

EXECUTING LIUENESS

PhD Thesis 2016 Winnie Soon

PhD dissertation  
School of Communication and Culture,  
Aarhus University

EXECUTING LIUENESS

An examination of the live dimension of  
code inter-actions in software (art) practice

WINNIE SOON

# **Executing Liveness**

An examination of the live dimension of code inter-actions  
in software (art) practice

**Winnie Soon**

A thesis submitted in partial fulfilment of the requirements of  
Aarhus University for the degree of Doctor of Philosophy.

November 2016

Executing Liveness –

An examination of the live dimension of code inter-actions in software (art) practice  
by Winnie Soon

PhD dissertation

School of Communication and Culture,  
Aarhus University, 2016.

Main supervisor: Geoff Cox, Associate Professor.

School of Communication and Culture, Aarhus University

Co-supervisor: Christian Ulrik Andersen, Associate Professor.

School of Communication and Culture, Aarhus University

Co-supervisor: Jane Prophet, Professor.

Goldsmith, University of London

Layout and design: Winnie Soon and Polly Poon

Proofreading: David Selden

Danish translation: Rachel Stoklund

Cover image by author. The image is generated by code and the design is referenced from a  
throbber in the earlier Unix operating system.

```

import processing.pdf.*;
String[] x = {"—", "\\\", "|", "/"};
int scaleFactor = 5;
void setup() {
    size(2000,2000); background(255);
    beginRecord(PDF, "thesis_cover.pdf");
}
void draw() {
    scale(200/72.0);
    for (int h = 0; h < height; h+=10) {
        for (int w = 0; w< width; w+=10) {
            fill(0);
            textSize(6);
            int y = int(random(0, x.length));
            text(x[y], w, h);
        }
    }
    endRecord();
    noLoop();
}

```

The above shows a piece of source code written in the language Java (with an open source software called Processing) for the printed book cover.

As part of the thesis's submission the USB storage device includes video documentations and source code for the three submitted projects, namely *Thousand Questions*, *The Spinning Wheel of Life* and *Hello Zombies*. Additionally the USB includes a README file for each project containing information and specification to RUN them.

# Table of Contents

<b>Table of Figures and Tables</b> .....	<b>6</b>
<b>Words of Thanks</b> .....	<b>10</b>
<b>Abstract (in English)</b> .....	<b>12</b>
<b>Abstract (på dansk)</b> .....	<b>14</b>
<b>1 Introduction</b> .....	<b>17</b>
<b>1.1 Motivation: <i>The Listening Post</i></b> .....	<b>19</b>
<b>1.2 Nonhuman Turn</b> .....	<b>23</b>
<b>1.3 Perspective on Liveness</b> .....	<b>26</b>
1.3.1 The living bodies and the presence .....	26
1.3.2 Interaction between humans and technology .....	30
1.3.3 Temporality and liveness .....	34
1.3.4 Unpredictability and liveness .....	38
1.3.5 A sense of (digital) liveness .....	41
<b>1.4 Aims and Contributions</b> .....	<b>43</b>
<b>1.5 Chapter overview</b> .....	<b>47</b>
<b>2 Approaches to code inter-actions</b> .....	<b>53</b>
<b>2.1 Software Art</b> .....	<b>54</b>
<b>2.2 Software Studies: Three key concepts</b> .....	<b>66</b>
2.2.1 Invisibility .....	68
2.2.2 Performativity .....	75
2.2.3 Generativity .....	82
<b>2.3 Materialist Approach</b> .....	<b>90</b>
2.3.1 Why code inter-actions? .....	93
2.3.2 Live inter-actions .....	96
<b>2.4 Methodological Considerations</b> .....	<b>100</b>
2.4.1 Close reading in Critical Code Studies .....	101
2.4.2 Iterative trials in Software Studies .....	103
2.4.3 Cold gazing in Media Archaeology .....	105
<b>2.5 Reflexive Coding Practice</b> .....	<b>107</b>

<b>3 Executing Unpredictable Queries .....</b>	<b>117</b>
3.1 Databases and queries .....	120
3.2 The format of output queries .....	127
3.3 Query as cultural form .....	132
3.4 The unpredictability of live queries .....	140
3.4.1 Random events.....	141
3.4.2 Noise, entropy and randomness .....	148
3.4.3 Operators .....	155
3.5 Inexecutable query in closed platforms .....	161
3.6 Notes on Reflexive Coding Practice: <i>Thousand Questions</i> .....	171
<b>4 Executing Micro-temporal Streams .....</b>	<b>187</b>
4.1 A cultural reading of a throbber .....	189
4.2 Micro-temporal analysis .....	195
4.2.1 Data Signal Processing .....	198
4.2.2 Data packets and Network protocols.....	203
4.2.3 Buffer and Buffering .....	211
4.2.4 The absence of data .....	218
4.3 The Spinning Wheel of Life .....	223
4.4 Notes on Reflexive Coding Practice: <i>The Spinning Wheel of Life</i> .....	228
<b>5 Executing Automated Tasks .....</b>	<b>247</b>
5.1 Spam as automated agents.....	251
5.1.1 Hello Zombies .....	253
5.1.2 Loop.....	262
5.1.3 Open or die.....	269
5.1.4 Try and Catch Exceptions .....	272
5.2 A sense of ending in algorithms .....	279
5.3 Notes on Reflexive Coding Practice: <i>Hello Zombies</i> .....	291
<b>6 Unfinished Thesis.....</b>	<b>305</b>
6.1 Contribution .....	307
6.2 Future directions.....	311
<b>Bibliography .....</b>	<b>315</b>
<b>Software (art) projects cited .....</b>	<b>335</b>

# Table of Figures and Tables

Figure 1.1: Liveness check feature in the Android operating system	28
Figure 2.1: <i>GEO GOO</i> (2008) by JODI	60
Figure 2.2: <i>Whitespace</i> (2003) by Edwin Brady and Chris Morris	64
Figure 2.3: The diagram of <i>Google Will Eat Itself</i> (2005)	74
Figure 2.4: An example of code that listens to mouse events	76
Figure 2.5: Two pieces of <i>Microcodes</i> (2009-) by Pall Thayer	78
Figure 2.6: An excerpt of the work <i>femme Disturbance Library</i> (2012).	78
Figure 2.7: A screen shot of the work <i>Net.Art Generator</i> (1997) by Cornelia Sollfrank	84
Figure 2.8: A thinking model of code inter-actions	94
Figure 3.1: A love letter from <i>LoveLetters</i>	117
Figure 3.2: Centralized, Decentralized and Distributed Networks	126
Figure 3.3: The Manhattan system of Twitter	127
Figure 3.4: An experiment to extract a sample tweet returned from Twitter platform.	129
Figure 3.5: Excerpt of code, in Processing Software, for parsing JSON query from OpenWeatherMap for getting a list of cities' name.	131
Figure 3.6: <i>Net.Art Generator</i> by Cornelia Sollfrank	137
Figure 3.7: <i>Endless War</i> was shown in Hong Kong as part of the exhibition <i>Tracking Data: What you read is not what we write</i> (2014)	137
Figure 3.8: A screen shot of <i>Thousand Questions</i>	139
Figure 3.9: A screen shot of <i>Thousand Questions</i> , where the program is waiting for the next query execution	139
Figure 3.10: A conceptual model of Twitter random input	144
Figure 3.11: Schematic diagram of a general communication system.	149
Figure 3.12: Two binary strings	151
Figure 3.13: A requested query in <i>Thousand Questions</i>	149
Figure 3.14: An excerpt of the returned query in <i>Thousand Questions</i>	156
Figure 3.15: The erasure of the data content of the requested query in Figure 3.10	156
Figure 3.16: The erasure of the data content of the returned query in Figure 3.11	156
Figure 3.17: A screen shot of the error page of <i>Net.Art Generator</i> (1997) that was captured on January 14 <sup>th</sup> , 2016.	167
Figure 3.18: <i>Thousand Questions</i> (2012-2016)	171
Figure 3.19: <i>Thousand Questions</i> in Hong Kong (2012)	173
Figure 3.20: Audio effects in <i>Thousand Questions</i> (2012)	175
Figure 3.21: A conceptual stage, the flow chart, of <i>Thousand Questions</i>	178
Figure 3.22: An excerpt from <i>Thousand Questions</i> ' source code: Setting up variables and screen dimensions, and establishing a Twitter connection	178

Figure 3.23: An excerpt from <i>Thousand Questions</i> ' source code: Querying Twitter data	179
Figure 3.24: An excerpt from <i>Thousand Questions</i> ' source code: Splitting tweets to individual characters	179
Figure 3.25: An excerpt from <i>Thousand Questions</i> ' source code: Processing text-to-speech	180
Figure 3.26: An excerpt from <i>Thousand Questions</i> ' log: The feedback process	180
Figure 3.27: The notes of <i>Thousand Questions</i> in 2012	181
Figure 3.28: The notes of <i>Thousand Questions</i> in 2016	182
Figure 3.29: An excerpt from <i>Thousand Questions</i> ' source code: General notes	183
Figure 3.30: An excerpt of a returned query from OpenWeatherMap.org	183
Figure 4.1: Throbber in different browsers.	190
Figure 4.2: Throbber in the form of circles and lines	191
Figure 4.3: A code-based throbber	197
Figure 4.4: Discrete time signals	199
Figure 4.5: The clock cycle	200
Figure 4.6: Three-way handshake	204
Figure 4.7: Data packet analysis I - the screen shot highlights the three-way handshake	206
Figure 4.8: Data packet analysis II - the screen shot highlights the two greeting messages	207
Figure 4.9: Data packet analysis III - the screen shot highlights the field 'Time to Live' for the data packet that transverses from the Youtube server to a local client computer	209
Figure 4.10: Sliding Window Protocol	214
Figure 4.11: TCP- flow control with the sliding window protocol	215
Figure 4.12: Principle organization of a playback buffer	218
Figure 4.13: <i>The Pirate Cinema</i> (2012-2014)	222
Figure 4.14-4.19: The animated visuals of <i>The Spinning Wheel of Life</i> (2016)	225
Figure 4.20: The mini setup and work-in-progress of <i>The Spinning Wheel of Life</i> (2016)	226
Figure 4.21: <i>The Spinning Wheel of Life</i> (work-in-progress) (2016)	228
Figure 4.22: Experiment on how a throbber display on a browser	230
Figure 4.23: Experiment on a throbber display with HTML, CSS and JS script	231
Figure 4.24: A slightly modified version of the Unix shell script	231
Figure 4.25: First Screenshot of running the Unix Shell Script	232
Figure 4.26: Second Screenshot of running the Unix Shell Script	232
Figure 4.27: Third Screenshot of running the Unix Shell Script	232
Figure 4.28: Experiment with the command 'tcpdump' for networked data analysis	233
Figure 4.29: Experiment with the parameters of 'tcpdump' for networked data analysis	233
Figure 4.30: Experiment with watching youku video with data analysis.	234



Figure 4.31: Log analysis for the youku video in relation to Figure 4.30	234
Figure 4.32: A screenshot of Wireshark for packet analysis (with a focus on window size)	235
Figure 4.33: Tracking networked data: Experiment the Carnivore library by RSG in Processing	236
Figure 4.34: Tracking networked data: Experiment the Carnivore library by RSG in Processing	237
Figure 4.35: The log for networked data experimentation	237
Figure 4.36: Initial setup concept of <i>The Spinning Wheel of Life</i>	238
Figure 4.37: Concept stage of <i>The Spinning Wheel of Life</i>	239
Figure 4.38: Concept stage of <i>The Spinning Wheel of Life</i>	239
Figure 4.39: First prototype of <i>The Spinning Wheel of Life</i>	241
Figure 4.40: An excerpt from <i>The Spinning Wheel of Life</i> (work-in-progress)'s source code: The ellipses design	242
Figure 4.41: An excerpt from <i>The Spinning Wheel of Life</i> (work-in-progress)'s source code: Setting up IP addresses and the carnivore library.	243
Figure 4.42: An excerpt from <i>The Spinning Wheel of Life</i> 's log: The feedback process	243
Figure 4.43: An excerpt from <i>The Spinning Wheel of Life</i> 's source code: General notes from 2015 to 2016	244
Figure 4.44: A screenshot of the notes from Apr 2016 to present	245
Figure 5.1: <i>Hello Zombies</i> (2014)	254
Figure 5.2: A spam poem in <i>Hello Zombies</i> (2014)	255
Figure 5.3: Sending out poems in <i>Hello Zombies</i> (2014)	256
Figure 5.4: Receiving emails in <i>Hello Zombies</i> (2014)	257
Figure 5.5: Running addresses in <i>Hello Zombies</i> (2014)	258
Figure 5.6: A while loop in Python and its result in the Mac OS's terminal	264
Figure 5.7: An infinite loop in <i>Hello Zombies</i>	264
Figure 5.8: Bounded loop in <i>Hello Zombies</i>	266
Figure 5.9: The concept of recursion in making a 3-layer cake.	267
Figure 5.10: I/O operations in <i>Hello Zombies</i>	269
Figure 5.11: An error result	270
Figure 5.12: Try and catch exceptions (1) in <i>Hello Zombies</i>	274
Figure 5.13: Try and catch exceptions (2) in <i>Hello Zombies</i>	275
Figure 5.14: A high-level flowchart of <i>Hello Zombies</i>	280
Figure 5.15: Decomposition of algorithms.	281
Figure 5.16: An idea sketch of Turing's halting problem in Python	283
Figure 5.17: The construction of N	283
Figure 5.18: <i>Hello Zombies</i> (2014)	291
Figure 5.19: Testing out different sculptural forms at City University of Hong Kong in 2014	292
Figure 5.20: Site visit in 2014	293
Figure 5.21: A blog was setup to document my own reflections.	294
Figure 5.22: A high level draft of the flow chart	295
Figure 5.23: A high level logics of the programs	295

Figure 5.24: Reading network replies in <i>Hello Zombies</i>	297
Figure 5.25: Sending poems in <i>Hello Zombies</i>	298
Figure 5.26: Rolling Spammer addresses in <i>Hello Zombies</i>	299
Figure 5.27: Densely packed spammer addresses in <i>Hello Zombies</i>	299
Figure 5.28: The excerpt of the source code on presenting email addresses on a screen	230
Figure 5.29 The highlight of a connection error in running the test programs of <i>Hello Zombies</i>	301
Figure 5.29: The highlight of a network error in running the test programs of <i>Hello Zombies</i>	301
Figure 5.31: An excerpt of the source code on sending poems	302
Figure 5.32: An excerpt of the source code on checking server emails	303
Figure 5.33: An excerpt of the source code on fetching spammers' address list	303
Table 1: A selected list of (software) artworks that address the notion of liveness	56

# Words of Thanks

*In memory of Bolei Poon (2006-2014)*

There are so many people I have to thank who, in different ways, helped me to accomplish this doctoral research journey. I am extremely grateful for the mentorship and support that I received from my supervisors, Geoff Cox, Christian Ulrik Andersen and Jane Prophet and Geoff in particular for identifying my potential, trusting my ability, and my thesis development is highly inspired by his works. It is my honour to receive their attentive supervision and my gratitude to them is beyond measure.

In the prestigious research environment at Aarhus University I could not have hoped for a more supportive, open and collegial atmosphere. I would like to thank the Participatory Information Technology Research Centre and the Graduate School of Arts which have funded my project and supported its development. Conversations with visiting researchers, faculty members and colleagues in the School of Communication and Culture have been truly valuable and my heartfelt thanks goes to Thomas Bjørnsten, Lone Koefoed Hansen, Nicolai Brodersen Hansen, Lukasz Mirocha, Finn Olesen, Lea Muldtofte Olsen, Søren Pold, Andrew Prior, Morten Riis, Sigrid Nielsen Saabye, Cornelia Sollfrank, Marie Louise Juul Søndergaard and Magda Tyzlik-Carver. I apologise if I have missed anyone off the list.

Many concepts that I have developed in this thesis were, in part, informed by the undergraduate course, Aesthetic Programming, which I taught twice in the Department of Digital Design. I am thankful to Morten Breinbjerg who appointed me, as well as for the creative dialogue with the students. I

would like to extend my thanks to the course instructors, Nils Rungholm Jensen, Frederik Højlund, Tobias Stenberg Christensen and Malthe Stavning Erslev.

I am especially grateful for those who have encouraged me and given me guidance and care to cope with issues that are related to self-confidence and gender. I have been inspired by the work and conversation of feminists such as Christine Cheung, Jane Prophet, Helen Pritchard, Audrey Samson, Cornelia Sollfrank, Sarah Schorr, Geoff Cox, Annette Markham and Jennifer Gabrys. They have given me the strength to become a more sensitive and stronger person working in a technology related field. My geek aunt Christine, in particular, who has raised me and brought me to this tech world through learning Logo, Telnet and circuit bending when I was young and she continuously inspires me to be curious in life. My co-supervisor Jane, a supportive and caring mentor, who generously hosted me in the City University of Hong Kong as a visiting researcher where I had the opportunity to observe and learn from her closely. Also special thanks to Helen for being a great advisor, friend and collaborator who has witnessed my development over the years.

I am thankful to my writing buddy Magnus Lawrie, as well as the Critical Software Thing Group for their valuable input in sparking dialogue and improving the manuscript at various stages. I also thank Maria Chatzichrostodoulou, Maureen V Eastwood, Christopher Newell and Toni Sant who assisted me in the early stages of my PhD.

Finally, and most importantly, my warmest thank goes to my beloved family. My wife, Polly Poon, an excellent listener who has given me tremendous support. She has taken good care of my mental and physical state, as well as offering an extraordinary calm and loving home. Finally I thank Bowtie Soon who has accompanied me throughout my ups and downs during the whole research journey.

# Abstract (in English)

With today's prevalence of technology enormous quantities of data are generated and disseminated in real-time through a highly networked, programmable and distributed environment. Networks of machines and the circulation of data mediate our sense of time. The sensation of 'liveness' is deeply reconfigured by complex technological infrastructures behind ubiquitous screens and interfaces. This thesis explores how real-time computation reconfigures this immanent sense of liveness, specifically in relation to contemporary software art and culture. By focusing on the live dimension of code inter-actions this thesis examines the complexity of our current computational environment as evident in the increasing use of data queries, the instantaneous transmission of data streams and the seamless running of automated agents.

By drawing together the methods of reflexive practice, close reading, iterative trials and cold gazing in the fields of artistic research, critical code studies, software studies and media archaeology respectively, this thesis presents three artistic and experimental projects together with the written manuscript. Together they examine barely visible code operations and consider the cultural implications of the reading, writing, running and execution of code, which I refer to as 'reflexive coding practice.' This methodology provides an applied approach to computational processes, invisible architectures and a means to reflect on cultural issues through experimentation and practice.

A materialist framework for liveness is presented with the use of three main vectors, namely: unpredictability, micro-temporality and automation. This facilitates the unfolding of the assemblages of things and relations that have emerged through the inter-actions of code across various computational layers at multiple scales. The analysis and discussion contributes to a widening of critical attention to software (art) studies primarily in terms of

its distinct focus on the live dimension of code. Furthermore, it expands the debate in media and performance studies, providing technical description and analysis in relation to the concept of liveness. In overall terms, the research contributes to our understanding of software by expanding our understanding of liveness in contemporary culture. This includes a nuanced examination of liveness beyond immediate human reception.

## Abstract (på dansk)

### At eksekvere 'liveness': en undersøgelse af live dimensioner i kode-interaktioner for software (kunst)

I kraft af den store udbredelse af teknologi, vi har i dag, bliver enorme mængder data konstant genereret og distribueret gennem et omfattende netværk, der både er programmerbart og distribueret. Store netværk af maskiner og cirkuleringen af data mellem dem, er med til at påvirke vores tidsopfattelse.

Følelsen af at være 'live' er rekonfigureret af en kompleks teknologisk infrastruktur som er til stede overalt bag skærme og brugerflader. Denne afhandling udforsker hvordan disse real-tids beregninger ændrer vores følelse af at være 'live', med et særligt fokus på nutidens software kunst og kultur. Ved at fokusere på 'live' delen af kodeinteraktioner, vil denne tese undersøge kompleksiteten af det nuværende beregningsmiljø, som det fremstår gennem vores øgede brug af data, den øjeblikkelige overførsel af data og den næsten usynlige brug af automatiserede agenter.

Ved at kombinere metoderne fra reflektiv praksis, nærlæsning, iterative forsøg og 'cold gazing' i relation til områderne kunstnerisk forskning, kritiske kode studier, software studier og medie arkæologi, vil denne afhandling præsentere tre kunstneriske og eksperimenterende projekter sammen med et manuskript. Sammen undersøger de næsten usynlig kodeafvikling og vurderer de kulturelle implikationer forbundet med at læse, skrive og afvikle kode, hvilket jeg refererer til som 'refleksiv kode praksis'. Denne metode resulterer i en brugsorienteret tilgang til computerrelaterede beregninger og processer og giver mulighed for at reflektere over kulturelle problemstillinger gennem eksperimenter og praksis.

Et materialistisk framework til 'liveness' bliver præsenteret ved hjælp af tre hovedvektorer: uforudsigelighed, mikro-temporalitet og automatisering. Dette faciliterer udfoldelsen af sammensatte objekter og relationer, som er opstået gennem interaktionen mellem kode på tværs af flere beregningslag i varierende skala. Denne analyse og diskussion bidrager til en udvidelse af fokus i den kritiske tilgang til software (kunst) studier, primært i forhold til det udprægede fokus på 'live' området af koden, såvel som medie og performance studier, hvori konceptet omkring 'liveness' tilsyneladende har behov for yderligere og mere kompleks teknisk formidling. Denne afhandling leverer en nuanceret undersøgelse af 'liveness', som går udover den umiddelbare menneskelige forståelse, med det formål at tilpasse vores forståelse af software og udvide diskussionen om 'liveness' i nutidig kultur.





# Introduction

With the prevalence of technology today, enormous quantities of data are generated and disseminated in real-time through a highly networked, programmable and distributed environment. Networks of machines and the circulation of data mediate our sense of time. Demand for the latest information is high and constant updates are expected. The sensation of ‘liveness’ or ‘nowness’ is reconfigured by the complex technological infrastructures behind ubiquitous screens and interfaces. The immediacy of interactions between humans and machines, such as click/touch actions and screen representation, are just part of a mega structure of computational logics. This thesis focuses on those interactions that are not directly apparent to us but are an essential part of what constitutes the sensation of liveness. From live streams on social media and breaking news to the constant update of predictive measures, such as weather forecasts<sup>1</sup> (Olaiya, 2012), stock markets (Pan et al., 2003) and even political campaigns (Tumasjan et al., 2010), data is captured and updated in a seamless manner that is both speedily and silently underscored by computational processing involving real-time calculation, analysis and the manipulation of data to *generate* the sensation of liveness.

The rise of so-called ‘big data’<sup>2</sup> in the 21<sup>st</sup> century has sparked unprecedented economic value through datafication—a phenomena in which personal profiles and behavioural logs are stored in corporate server farms. Data is captured, processed and analysed to generate new information and knowledge. New business models have been established that aim to manage,

---

<sup>1</sup> Many companies provide a minute-by-minute update weather forecast. AccuWeather is one of them. See: <http://www.accuweather.com/en/about>

<sup>2</sup> Few art exhibitions have addressed this cultural phenomena: “Data in the 21st Century” (2015-2016), organised by V2\_Institute For the Unstable Media in Rotterdam, Netherlands and “Big Bang data” (2014, 2015, 2016), co-organised by The Centre de Cultura Contemporània de Barcelona in Spain, United Kingdom and Singapore.

produce and analyse big data for profit-making, such as cloud computing (Cisco Systems, 2013), API businesses<sup>3</sup> (Mason & McKendrick, 2015) and tracking solutions (Barcena et al., 2014; Oracle, 2012). These models are implemented at the level of code within platforms, applications and software packages, enabling data to be captured, accessed, analysed, manipulated and distributed in the background behind a user-friendly interface and within a technological network. The increasing phenomena of networked agents, including applications, firmware and feed updates, push notifications, auto files and data synchronisation, suggests that dynamic code and automated live processes play an increasingly significant role in cultural activities, as part of our everyday practices.

Computation can be processed behind and beyond a screen according to pre-programmed rules and logics. On a more conceptual level, phenomena are processed on a plane of “immanence” with unformed elements, variables and materials (Deleuze & Guattari, 1987, p. 255). The plane is not fixed, rather it moves at different speeds and comprises distinct relations and hence produces differences. The processing of cultural logics is subjected to a distributed live environment that consists of many other things that are contingently brought together as a state of becoming. The attention to speed and time gives rise to the assemblage of things (Deleuze & Guattari, 1987, p. 255). Therefore, the connections and relations between things exist in multiplicities. This plane of immanence, according to philosopher Gilles Deleuze, is not confined to humans, but emphasises wider relations with machines and other entities (1988, pp. 127-8).

This thesis examines the relations and interactions of code between substances, elements and materials in this way. In the era of big data, the execution and running of code not only enables the storage of an ever-

---

<sup>3</sup> API refers to Application Programming Interface, which is a form of machine query and interface that is used for communication between applications or programs. The offering of an API allows more third-party applications to build upon services like Google Maps and Twitter. Data can be “redistributed” and “remixed,” opening up a connection that results in more data, activities and usage generation (Soon, 2016, in press). APIs also facilitate new business model generation, for example Google Maps have been embedded in many other mobile applications / games.

growing amount of data but also the capacity to process the mix of data across the past, present and an unknowable future. This is all accelerated by real-time technology. According to software and media studies scholar Wendy Hui Kyong Chun, the term real-time “refers to the time of computer processing, not to the user’s time. Real-time is never real time—it is deferred and mediated” (Chun, 2011a, p. 98). The essence of real-time data processing may be understood as the collapse and construction of time, in which different kinds of data are being processed and manipulated to produce the immanent sense of liveness. Examples of this include, for instance, the ‘timeline’ interface in various social media applications and the network-provided time<sup>4</sup> on mobile devices. Such an instantiated sense of nowness is a computed and rendered snapshot, which is subjected to a live and networked environment of influences, as a plane of immanence. Importantly the now is constantly changing through computation: mutating in both time and space. The now exists in multiple forms and people are reading these different computational *nows* to access, adapt, react and imagine the world. As Chun explains, “The NOW constantly punctures time, as the new quickly becomes old, and the old becomes forwarded once more as new(ish)” (2016, p. 3). Therefore, the now is effectively a complex multiplicity of *nows*.<sup>5</sup> This thesis explores how real-time computation reconfigures this immanent sense of nowness, which I refer to as liveness in this thesis, specifically in relation to contemporary software art and culture (the notion of liveness will be further discussed in the latter section).

## 1.1 Motivation: The *Listening Post*

My interest in the notion of liveness was first inspired by my experience of an award-winning artwork called *Listening Post* (2000-2001) by statistician Mark Hansen and sound artist Ben Rubin. The project consists of 231

---

<sup>4</sup> The feature “Network Identity and Time Zone” (NITZ) is a mechanism offered by telecommunication operators to provision local time and date to mobile devices. The consequence is that a mobile device will get an automatic update of the system clock of mobile phones. See the service description of NITZ: [http://www.etsi.org/deliver/etsi\\_ts/101600\\_101699/101626/06.00.00\\_60/ts\\_101626v060000p.pdf](http://www.etsi.org/deliver/etsi_ts/101600_101699/101626/06.00.00_60/ts_101626v060000p.pdf)

<sup>5</sup> This comes close to the notion of contemporaneity that is under exploration by the research project The Contemporary Condition: The Representation and Experience of Contemporaneity in and through Contemporary Arts Practice. See: <http://contemporaneity.au.dk/about/>

monochromatic screens arranged in a rectangular grid in a darkened room. The screens show the running fragments of texts, starting with the text sequence, “I am,” “I like,” or “I love.” The text is captured in real-time from thousands of internet chat rooms, bulletin boards and other public forums. The sound experience is carefully composed of pulsing and flashing beats, accompanied by the synthesised recitation of text taken from the internet. Both the visual and sonic effects of the text are the result of an endless process of data scraping and processing. I was fortunate to visit the art installation at London’s Science Museum in 2008, sitting on a bench and gazing at the piece for some time. Back then, in the late 2000s, Interactive Art still tended to emphasise the audiences’ participation and co-production of human and machine (Jacucci et al., 2010; Kluszczynski, 2010). However this piece breaks with that convention as it is an autonomously running machine in as far as it runs without any human intervention or interaction, yet it offers a dynamic experience through its solo performance that engages its audience. Scholar of literature and media studies, Roberto Simanowski, categorises the work as “Real-time web sculpture,” demonstrating the “ever-changing compositions of dissociated messages” that are scraped from online communication (2011, p. 199). The artwork not only offers a rich visual and immersive sonic experience but also, as Simanowski suggests, “prompts reflections” through the capturing and presenting of data (2011, p. 200).

Simanowski is interested in the semantics of the text, such as indicating the currency of information on the internet or specific message content. By contrast, I am more curious about how the machine performs on its own and what this indicates in terms of the production of meaning and authorship. Performance studies scholar Philip Auslander raises a similar question: “If *Listening Post* is a performance, who or what is the performer?” Fundamentally, “do machines perform?” (2005, p. 5-6) Notably, the work challenges our general understanding of what constitutes performance. Auslander argues that *Listening Post* is in itself a live performer and concludes that the term performance is not exclusive to human behaviour, and that taking machine performance into account for analysis is indeed necessary (2005, pp. 8-9). To what extent are these free-standing machines

to be regarded as performers? What are the parameters that categorise them as live performers? How might we better understand machine performance? Specifically, how is the notion of liveness different in a machine performance from the liveness of a human performance, or human-machine performance?

Regarding machinic performance, the use of real-time technology allows the programmed software to express the temporality and immediacy of data in the piece *Listening Post* in its own terms through its programmed rules. Importantly, an audience can never capture the full complexity of computation and data processing within the piece and the feeling of liveness is a consequence of the audience's experience of what has been presented on the representational layer of the piece. For instance, the feeling of proximity to the Internet "crowd" with approximately 100,000 real-time messages (Raley, 2009, p. 31), or what radio host Jad Abumrad describes as a "mirror to look at society," or how radio reporter and producer Allan Coukell suggests the piece offers "a real sense of what people are talking about" (both cited in Simanowski, 2011, p. 193). All these *senses* of liveness are based on the spectacular technological effects that transform the data into a perceptible form. However, we gain little understanding of this in our experience of the work. So how might we discuss machine performance beyond the representation of data, its meaning and the perception of audiences?

Recognising the gap between what is happening behind the representational layer of the work and what is perceivable by audiences, I want to shift away from the human-centric interpretation of what constitutes liveness to investigate liveness in terms of the dynamics of computational and networked technology. I have therefore developed frameworks to better understand the performance and aesthetics of a running machine such that we can perceive technological artworks on a different register. My assumption, in line with how Auslander discusses machine performance, is that liveness should not be a term that is exclusive to human experience. I acknowledge that technology plays a significant role in shaping the

sensation and aesthetics of liveness. In particular, I am arguing that technology should not be understood merely as the manifestation of visual and sonic experience, that we should understand it additionally through the operational aspects of technological processes behind these representations such as data scraping and capturing, network transmission and processing. Performance studies scholar Christopher Newell defines liveness briefly as something “beyond a vague sensation of immediacy and now-ness” in his research on synthetic voice analysis (2009, p. 13), in which the sensation of liveness is, as he claims, “easy to detect but difficult to describe” (2009, p. 95). A better understanding of operational computational processes will arguably enrich what is otherwise only a vague sensation of liveness. Taking this operative approach to *Listening Post*, our understanding of performance and the notion of liveness extends beyond audience perception. Informed by this premise, the work *Listening Post* becomes the point of departure for this thesis.

## 1.2 Nonhuman turn

My project starts out from the observation and recognition of the fact that code inter-acts with different arrays of technology, across artistic, cultural and industry-based practices. The notion of inter-actions, which I use throughout the thesis, references computer science's understanding of "interaction" (Beaudouin-Lafon, 2008; Bentley, 2003; Murtaugh, 2008; Wegner, 1997) as well as the notion of "intra-actions" from philosophy (Barad, 2003, 2007). The next chapter comprises a more detailed discussion of this concept. In general, the inter-action of code carries not only the technical attributes and technical qualities of how things are made functional and operative but how these inter-actions are also embedded with "anonymous forces" which constitute what I have already introduced as the plane of immanence, something "constantly being altered, composed and recomposed, by individuals and collectives" (Deleuze, 1988, pp. 127-8). These forces are comprised of both the entanglements of human and nonhuman phenomena.

My emphasis on forces, collectives, elements, variables and materials (as well as vectors, a concept that I will introduce later) is influenced by what has been called the 'nonhuman turn'<sup>6</sup> that has emerged in the arts, humanities, and social sciences in recent years. The nonhuman turn involves a critical reconsideration of human-oriented approaches to examining the world. According to poststructuralist scholar Jon Roffe and literary scholar Hannah Stark,

[the nonhuman turn] has been instrumental in challenging human privilege and placing the human in the more-than-

---

<sup>6</sup> In 2012, a conference on the 'nonhuman turn' was organised by the Center for 21<sup>st</sup> Century Studies in Milwaukee, United States. The conference brought together scholars whose works were situated in the realm of the nonhuman turn. The conference was a key to provoking debates, challenging the anthropocentric discourse and intensifying the development and discussion of the nonhuman turn. In 2015, a book titled *The Nonhuman Turn* emerged from the conference. It was edited by Richard Grusin, who was also part of the conference's organising team. The publication was claimed to be the first to discuss various aspects and approaches under the concept of the nonhuman turn. See: <http://www.c21uwm.com/nonhumanturn/>



human world, motivated in part by the ongoing theoretical and political interrogation of the anthropocentrism of the Western tradition (2015, p. 2).

A wide range of theoretical approaches have been developed in various fields, from Actor-Network Theory, to Speculative Realism and Object Oriented Ontology to New Materialism, suggesting a critical engagement with nonhuman objects beyond human and even other biological life forms (such as animals and plants). Importantly, it provokes the recognition of objects and considers all human and nonhuman beings as active participants in shaping the world. Focusing on the relation between various ‘actants,’<sup>7</sup> technologies, things and materials in this way suggests that nonhuman things act and perform beyond human control. They exhibit agency through their *inter-action* with, and through, the world, producing meaning and shaping events that call for our critical attentiveness. As such, this thesis follows the trajectory of the nonhuman turn to examine the interactions of things.

Informed by networked conditions—data capture, streaming and networked agents—the primary objective of this thesis is to develop a more nuanced understanding of liveness through code inter-actions and examine how code inter-actions exhibit particular forms of liveness in contemporary software cultural practices. The intention is to contribute to a widening of the focus of critical attention in software studies, by investigating liveness beyond immediate human reception. Situating this thesis within the domain of software studies, it follows an established critical tradition of scholarship (Berry, 2011, 2014; Cox, 2013; Fuller, 2003; Mackenzie, 2006; Marino, 2006, 2014) in which code is the primary object and subject of study. Within software art practices, attention moves from the representation and display of the art object to processes of computational systems that have been

---

<sup>7</sup> The term ‘actant’ was first suggested by Bruno Latour in Actor Network Theory. The word actant is used to extend the term actor and agent to those relatively uncommon nonhumans or non-individual entities, respecting the fact that all things play an *active* role in shaping the world (Latour, 1999, pp. 180-1). Latour refers to actants as anything that operates as “a source of action” (1996, p. 373).

relegated to the background in works like *Listening Post*. Some artists have critically reflected upon the structures of program code and the cultural operation of digital objects, “[making] visible the aesthetics and political subtexts of seemingly neutral technical command” (Arns, 2004, p. 178). This type of critical art practice has gained wide attention in art festivals, exhibitions and scholarly works over the past decade (and examples of works will be described in detail in subsequent chapters, including examples drawn from my own practice).

As an artist-programmer, my works engage similarly with and reflect upon contemporary software culture, computational networks and processes. In recognition of the fact that code is not a standalone object, I make a strong assertion that code cannot be separated from other relational objects and materials. It is more than program source code or high level programming code. Code compiles and executes as executable code and also operates in the form of network protocols that control data transmission (Galloway, 2004, p. 7). Moreover, code can be seen in the form of a script or command that provides access to other computer systems, or even queries data in databases. Code, according to artist-programmer-scholar Alexander Galloway, “is a set of procedures, actions, and practices” (2004, p. xii). The procedures allow code to inter-act with various machines, systems, networks and databases. To examine such complex relations I bring together the closely related fields of software studies, digital humanities, media archaeology, platform studies<sup>8</sup> and interface studies,<sup>9</sup> whilst acknowledging the inter-actions of various objects beyond the dichotomy of software and hardware, or technology and culture. Therefore, code is not considered as software specific but refers to wider computational assemblages that perform and act in the world, such as social media platforms which operate across machine interfaces, distributed technology and cloud servers. Therefore, the concept of code is taken in this broad sense, as operating beyond software applications and programming languages.

---

<sup>8</sup> See: <http://platformstudies.com/> and <https://mitpress.mit.edu/books/series/platform-studies>

<sup>9</sup> See: <http://mediacommons.futureofthebook.org/tne/pieces/manifesto-post-digital-interface-criticism>

As this thesis does not focus on code as an isolated or standalone object but rather considers code inter-actions in a dynamic networked environment which is process-oriented, the processual events therein cannot be examined through the mere written form of code. Beyond that, code alludes to the activities of executing and running code, inter-acting with different systems, objects and materials which together generate the phenomena of liveness.

### **1.3 Perspectives on liveness**

Most importantly, the central theme of this thesis is to explore the notion of liveness within the context of contemporary software culture, thereby filling a perceived gap by adding detail on the complexity of code inter-actions. In the following section I offer an overview of the notion of liveness, organised around textual analysis. The aim is to show how the concept of liveness has evolved and been discussed to date through the exploration of key approaches that I expand upon in subsequent chapters. The overview identifies this perceived gap in the wider discourses on liveness but also outlines some of the fundamental concepts of liveness that will be developed further in the remainder of the thesis. In keeping with overviews, at times the discussion may seem limited, but further detail will be added in subsequent chapters.

Liveness has long been a subject of debate in performance and media studies. Although it has been widely discussed in relation to various media, including but not limited to television and radio broadcast, digital cinema and music, the concept of liveness still remains contested and there is no agreed precise or single definition (Barker, 2012, p. 61; Crisell, 2012, p. 3; Davis, 2007, p.36; Newell, 2009, p. 13). Seemingly, liveness is a contingent concept with various definitions of the term situated in a wide array of other discussions.

#### *1.3.1 The living body and presence*

Within the history of theatre and performance art, liveness is often used to

describe a setting where there is the presence of human living bodies, exploring the relationship and engagement between performers and spectators (Carlos, 1998, p. 10). The term Live Art is sometimes used interchangeably with Performance Art, in which the former takes its historical roots in the United Kingdom (UK) where it was used to indicate a broader understanding of performance inclusive of media and technology and to indicate a type of performance that emanated more from a Fine Art tradition (Live Art Development Agency, n.d).

The presence of living bodies, to some scholars, is fundamental to performance. Performance studies scholar Peggy Phelan does not regard other kinds of technological media as live performance. “Performance in a strict ontological sense is nonreproductive [...] [It] implicates the real through the presence of living bodies” (Phelan, 1993, p. 148). The presence of human bodies is crucial that plays a central role of the ‘live act’ in the performance of live art (Jones, 2012, pp. 12-3). When both performers and spectators are physically located in a same space, described as “physical co-presence” (Auslander, 2008, p. 61), the performing acts and receptive experiences are happening simultaneously in the same time and space. Auslander describes this co-presence as “classic liveness” (2008, p. 61) and this can be traced back to the 1960s when some artists wanted to make radical changes that challenge the canon of established, and more traditional, art media (Carlos, 1998, p. 15).

Over the years, classic liveness involves living bodies that explore the role of various life forms in the domain of live/performance art, stretching the meaning of biological life to include other beings such as animals. However, a concern with living bodies remains central (Sofaer, 2002), as the most basic level of performance art, according to curator Cindy Baker, “requires the presence of a body in space over time” (2014, p. 5).

Beyond performance art, the concern with bodies or living human beings is also discussed in relation to notions of liveness in computational applications and devices. Within biometric detection systems, the term

liveness is used to indicate the sign of a human being. For instance, the term ‘liveness detection’ is used in fingerprint, face and iris recognition systems (Abhyankar & Schuckers, 2004; Chakraborty & Das, 2014; Drahansky, 2011; He et al., 2010; Pan et al., 2008). Another human-centric definition of liveness is how the function ‘Liveness Check’ has become a standard option in the Android Operating System - Jelly Bean series on smartphone devices. ‘Liveness check’ is a security-checking feature that requires a user to blink in order to unlock their device (see Figure 1.1). In other words, the feature ensures not only the presence of a living body and the right body.

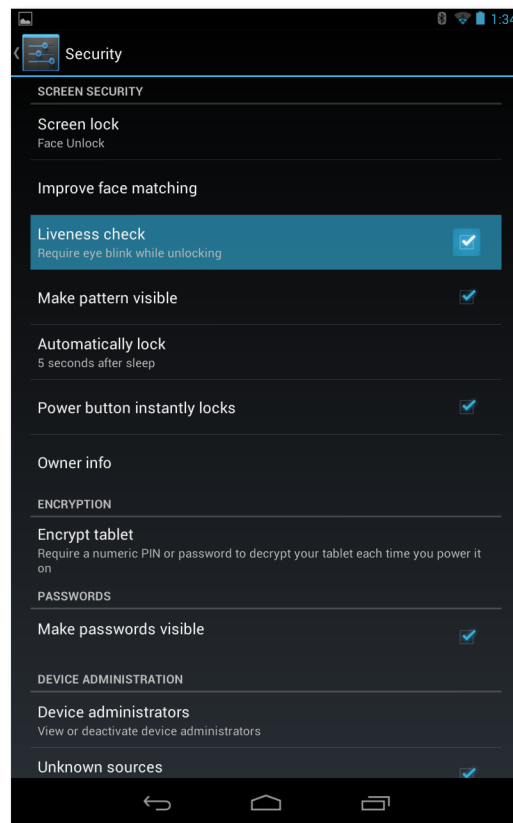


Figure 1.1: Liveness check feature in the Android operating system

Extending the discussion beyond physical presence with living human bodies, media theorist Paddy Scannell explains the *sensation* of presence in television broadcasting. Some television programmes, such as travel series and breaking news, allow audiences to have a “real sense of access to an event.” The audio-visual representation unfolds the events moment-by-

moment, producing “the effect of being-there, of being involved (caught up) in the here-and-now of the occasion” (Scannell, 1996, p. 84). In other words, liveness (from the perspective of presence) is about audiences who engage with television images<sup>10</sup> with a sensation of the events, offering the sense of presence, of being-there.

In internet environments, Auslander further explains the sensation of presence by drawing upon Nick Couldry’s discussion of liveness on the internet and in social devices. He describes the characteristics of networked liveness as a “sense of co-presence” among users and a “sense of connection to others” (as cited in Auslander, 2008, p. 61). In the context of the immersive virtual environment, presence is further and rather pragmatically described as “the sensation of being at the remote worksite” that is not necessarily situated at the actual physical locale (Witmer & Singer, 1998, p. 225). In *Second Life*, a 3D virtual world, live performance takes place in which an avatar represents a person, creating live performance in real-time using streaming technology. In this case, unlike classic liveness, spatial co-presence is a defining characteristic of liveness in which performers and spectators are not in the same physical space but rather share a virtual space. Telepresence emerges as a term specifically to describe this remote presence situation with the use of virtual-environment technology (Rheingold, 1991, p. 158). The sensation of remote presence is further promoted through the live transmission of images and sound effects, alluding to the remote experience of proximity and intimacy (Donati & Prado, 2001, p. 438; Zemmels, 2004, p. 11).

From concerns over the presence of a physical living body to the mediated

---

<sup>10</sup> The quality of liveness can be technically examined through television image. Wendy Davis argues that liveness can be observed through the qualities of the material surface of the televisual. The logic of television transmission is based on scanning with the wavy lines in the image. At any point, the television image is never completely composited. As Davis explains, “The television image has no separate frames as such, because the image is produced through a continuing signal that modulates in intensity” (Davis, 2007, p. 45). The operational perspective pays attention to the technological forces in which “the technicalities of scanning and transmission produced a specifically televisual image” (Davis, 2007, p. 46). This moves away from the unfolding of events to the operative processing of television’s surface. In other words, liveness is about the presence of television images.

sensation of being-there, the concept of liveness and presence are thoroughly intertwined. Technology plays a significant role in offering a sense of intimacy and immediacy through a spatio-temporal networked environment (Zemmels, 2004, p. 2). A more detailed examination of the interactions between humans and technology is required to discuss the notion of liveness more fully.

### *1.3.2 Interaction between humans and technology*

Auslander's book *Liveness* was first written in 1999. Nine years later, during which time different cultural understandings of live performance and various mediated live forms had been promulgated, he published the second edition in response to this changing technological landscape. Of course any current understanding of liveness will have shifted since then too. He acknowledges that highly volatile media landscapes—such as television, telecommunications and the internet—are part of the reason that he shifted his central focus to digital media (Auslander, 2008, pp. xi - xiii). Auslander traces the term 'live' through a history of recording technologies and argues that "live is actually an effect of mediatization [...] it was the development of recording technologies that made it possible to perceive existing representation as live" (2008, p. 56). He defines mediatization as technical mediation and claims that mediatization has a dependent relationship with liveness, arguing that "liveness was made visible only by the possibility of technical reproduction" (Auslander, 2008, p. 57). In contemporary software culture it is not surprising that liveness is mediatized by various kinds of technology, such as a distributed and programmable network. The focus of this thesis is neither the visibility of liveness nor the possibility of technology creating a live environment but rather what constitutes the "mediatized live" (to use Auslander's term). Precisely what constitutes a distributed and programmable network? How do we delve into technological materials that inform different understandings of liveness?

Indeed, Auslander has recognised the need to examine technological

materials. Drawing upon the work of media studies scholar, Margaret Morse, he opens up the discussion about human and machine interaction beyond the audience-performer model. He notes, “Liveness is attributed not only to the entities we access with the machine but also to the machine itself” (Auslander, 2008, p. 62). He did not discuss the machine itself in any detail but, all the same, shifted attention to the relations between human and machine interaction. Auslander gives the example of a website that is said to ‘go live’ to foreground the relationship between the feedback mechanism of user input and machine output. He explains, “the liveness of a website resides in the feedback loop we initiate with it: the website responds to our input” (Auslander, 2008, p. 62). The things behind a website include procedures that are embodied in code and users interact with code (Bolter et al., 2013, p. 328). Media studies scholar Jay David Bolter highlights the importance of procedurality when discussing liveness in the digital realm. A website, for example, captures the parameters of a human’s actions and continuously processes the request and feedback according to written procedures. Such procedurality maintains a feedback loop between human and machine that refers to the “performativity of digital media” (Bolter et al., 2013, p. 328) in which the human is performing within a procedural feedback loop. Therefore, both human and nonhuman entities are participating in the performance and interaction.

With regards to human and nonhuman interaction, the practice of ‘live coding’<sup>11</sup> that has emerged in computer music performance blurs the distinction between composers and programmers. Live coder and musician Shelly Knotts summarises the practice of live coding as “writing code on stage to produce sound” (2013). A performer interacts with code in live performance situations in front of audiences. The performance creates a nearly instantaneous feedback loop, taking the command written by a composer-programmer and output as audible sound. Audiences are able to

---

<sup>11</sup> There are different organisations that are founded to promote live coding in the 2000s. For example, TOPLAP and Repl Electric. See: <http://toplap.org/> and <http://www.repl-electric.com/>. Moreover, international conferences are setup for live coding research, see: <http://iclc.livecodenetwork.org/>. Performer/Programmer, such as Alex McLean and Sam Aaron develop various tools for live coding performance.



read the editing code (sometimes the code also overlays other visual effects) and listen to the sound that is generated by that code in real-time.

Unsurprisingly the live aspect has been widely discussed in the area of live coding. More precisely, the framework that analyses the degree of liveness, as proposed by computer science scholar Steven L. Tanimoto merges programming and systems' perspectives (1990, 2013). Tanimoto highlights the shift in computer programming from the traditional four distinctive phases of the development cycle, 'edit-compile-link-run cycle,' and reduces it to one phase only. He explains that code is running continuously even though there are various code edits on-the-fly (Tanimoto, 2013, p. 31). In other words, once running a program does not stop unless a terminate instruction is made during the live coding event. Liveness is considered as a characteristic of a programming environment, in which the programming software has to be easy for a programmer "to understand quickly what a program is doing or supposed to do" (Tanimoto, 2013, p. 31). From this perspective Tanimoto's analysis of the various degrees of liveness is based on the near-instant feedback between a program and a programmer. His latest article highlights two additional phases that a system incorporates, the new fifth and sixth levels of liveness, 'tactically predictive' and 'strategically predictive' measures respectively. Such 'intelligent' systems are capable of predicting the "programmer's intentions or desires," and this type of system has "the ability to act autonomously" (Tanimoto, 2013, p. 34). (This automated characteristic of a system requires more attention in understanding the liveness that is generated by automatic system, and this will be discussed later).

Similarly, again with respect to live coding, artist-programmer Alex McLean refers to a programming language that is easier for a user to learn and debug as "more live" (2014, p. 1). He summarises three main feedback loops in live coded performances: 1. "manipulation feedback" which happens between the programmer and their written code; 2. "performance feedback" which happens between the programmer and the program output, such as sound; 3. "social feedback" which happens between audiences and the

programmer (McLean, 2014, p. 2). The feedback relations between the programmer-performer are one of the key components in discussing the notion of liveness in live coding (performance).

McLean is also interested in the feedback loop within the machine, especially how code interacts with, or modifies, itself. He developed a text editor *Feedback.pl* (2004)<sup>12</sup> that allows programmers to edit the code during live coding performances and for it to edit itself. The code on the one hand is being modified by a human and, on the other, it has the ability to modify itself, changing the original source code on its own in real-time (Cox, 2013, pp. 61-2; Cox et al., 2004, pp. 170-1; McLean, 2004). This feature of self-modification, according to McLean, is useful for user feedback because it visually indicates “what the running code is up to” during live coding performance, where the performer can read the status of the code and respond to it (2004, n.p).

The focus on human and machine feedback is also observed in the field of artificial life (A-Life) that examines natural systems, exhibiting “life-like behaviour”<sup>13</sup> (Bedau, 2003). According to David Cameron and John Carroll, scholars of digital media, drama education and technology, “The level of liveness and direct human player input is most evident in forms of machinima designed for live performance” (2009, p. 3). In A-Life, the behavioural, computational graphical models and virtual objects, as well as live input data are all central to the discussion of feedback mechanisms. In other words, the real-time rendering between the input and output become core issues within the discussion of liveness in the context of artificial life.

Nevertheless, in general, the discussion of feedback loops tends to concentrate on human and machine interactions and pays only slight

---

<sup>12</sup> The text editor is for edit Perl code. In live coding performance, it is required to run the written code all the time without any pause. The editor is designed without a save function See: <http://www.perl.com/pub/2004/08/31/livecode.html>

<sup>13</sup> The artistic project *TechnoSphere* (1995), created by Jane Prophet and Gordon Selley, serves as an example that demonstrates life-like behavior through the use of 3D graphic rendering techniques and real-time technology, creating a 3D simulated virtual environment (Prophet, 2001).

attention to cover other scenarios which Auslander describes in terms of the machine itself. In the model of human and machine feedback mechanisms, input usually comes from a human user, however it does not *only* come from a human/user but also from other systems. As opposed to Auslander's scenario of a website 'going live,' the term 'live' signals, on the contrary, a *readiness* for public viewing and participation. The website is regarded as live once the pages are put up or uploaded on a production platform, usually located in other systems associated with the web server and internet network. From a systems point of view, the platform takes in the upload/input of web files and outputs them as webpages. Theoretically and practically speaking, a website can be claimed to be live once the upload process is completed regardless of any page views by users. Once the website is loaded on a production server, it is live unless someone takes it down or it crashes. Additionally, the earlier example *feedback.pl* demonstrates the ability of self-modifying code in which code interacts with itself and not only for other humans, as software studies scholar Geoff Cox puts it, "code embeds both action in-itself and action for-itself" (2013, p. 61). Furthermore, in his book *Protocol*, Galloway highlights that a feedback loop may not only occur between humans and machines, but it also between computers' interaction through protocols that "operate at the level of coding" (2004, pp. 7-8). It is precisely this thinking and discussion of code inter-actions (with itself and other nonhuman systems) that the thesis further develops, suggesting a fuller examination and acknowledgement of nonhumans in the overall discussion of liveness.

### 1.3.3 *Temporality and liveness*

The concept of time is crucially important in the discussion of liveness. The previously mentioned degree of liveness and feedback, as proposed by Tanimoto, are both subjected to the response time between a given human and machine. The concept of liveness is, instead, understood as 'non-instantaneous' as the moment of a programmer's input does not happen at the same time as the machine's output (McLean, 2014, p. 2). Chun has also

recognised that the notion of real-time is “deferred and mediated” (2011a, p. 98) as mentioned earlier.

The discussion of real-time has become central to the domain of media and internet studies. Real-time, on the contrary, usually refers to “instantaneous communication” that enables timely or instant delivery (Palmer, 2008, pp. 10-1; Zemmels, 2004, p. 2). In particular, the so called “real-time web” has been championed to differentiate from an earlier static web era, to “live social activities” that are now taking place on a more dynamic web platform (Weltevrede et al., 2014). The notion of real-time is generally understood as the human experience of time within the context of media study (Palmer, 2008, pp. 10-1; Weltevrede et al., 2014). This perspective also resonates with literary and media studies scholar Rita Raley’s discussion of liveness which is based on the artwork *Listening Post* (referred to earlier). She claims that this work, even though it collects data in “near real-time (there is [actually] a time delay of 1-2 hours),” still constructs a sense of liveness and audiences feel the immediacy of the process (Raley, 2009, pp. 24-30). In other words, our experience of time is subjective and cannot be measured or standardised precisely. Instead, the issue of time is a way to discuss the sensation of liveness as a mediated experience.

In a similar vein, software studies scholar David M. Berry suggests that the real-time computation of the web brings “liveness, or nowness to the users and contributors” (2011, pp. 142-3). He explains that it is the operability of the real-time web, including the process of content-generation and the feedback mechanism that provides new experiences to users and producers. Precisely, the contemporary experience of nowness is the result of the real-time web and the immediacy of social media platforms. Immediate user interactions and updated feeds are made possible via real-time technology (Weltevrede et al., 2014, p. 2). As already explained, the word real-time is never a real-time delivery as such, “the processing information is organized at such speed that allows for access without perceptible delay” (Weltevrede et al., 2014, p. 4).

Therefore, the notion of real-time is highly related to liveness that constitutes the live experience and yet real-time is fundamentally about computer processing time (Chun, 2011a, p. 98; Weltevrede et al., 2014, p. 4). Esther Weltered, Anne Helmond and Carolin Gerlitz explain the concept ‘system real-time’ as follows:

real-time refers to systems and processes performing tasks in predetermined temporal windows, most notably in micro-or nanoseconds, and the computational challenges of this (2014, p. 4).

This explanation suggests that the notion of real-time concerns micro-time and micro-processes that might not be humanly perceptible. If liveness in part refers to system real-time, then the constitution of liveness, as I argue, could also be produced through micro-processes of time in a system. Seemingly little attention has been paid to these micro-processes and micro-temporalities which operate within real-time environments and beyond human perception (This microscopic perspective of time will be discussed further in later chapter).

The use of the term ‘live’ has become commonplace. In particular, internet service providers often use the phrase ‘live data.’ The notion of liveness on the internet often refers to data which is updated immediately. However, Tara McPherson, scholar of digital media, argues that such an immediate sense of updated feeds is an “illusion of liveness” because it is not always instant but a mere recycling of data that is repackaged as “newness” (2006, p. 202). She points out that computational processing involves the movement of data and this relates to “the depth of electronic forms” which is not only temporal but is also spatialised (McPherson, 2006, p. 202). Such focus on the spatial-temporal dimension of data is one of the key components in understanding data beyond representational forms as to how they appear to us in a perceptible form.

Beyond the focus of media and internet studies, Luciana Parisi covers a

wider range of data in her philosophical account of computational architectural space. According to her,

Data are defined by what has been in the past, but also by what might have been, and by what might yet be of the spatial configurations: a software program, the real-time movements of a crowd, the reshaping of the pistons, all enter into a quantitative relation that precisely accounts for an invisible spatiotemporality (Parisi, 2013, p. 125).

Parisi highlights the potentiality of data, which is beyond its direct materialised form. Data enters a spatiotemporal relationship with other configurations in real-time. The understanding of real-time technology cannot be discussed without thinking of the “aliveness of data” that is related to “the capacity of software of media technologies.” This suggests that different forces, not a single entity, constitute the notion of liveness as a plane of immanence. As Parisi puts it,

The capacity of software of media technologies to retrieve information live, and to allow this information to add new data to programming. Real-time technologies can only be understood in terms of the *aliveness* of data (2013, p. 266, *original emphasis*).

Parisi examines temporalities through parametric and computational design which constantly transforms architectural space. Architecture, like a computational system, contains various parts and any change of constituted configurations, parameters and values will consequently alter the system as a whole. These variables include “relations between mechanical, physical, and algorithmic parts” (Parisi, 2013, pp. 107-9) which are inter-acting on a plane. The real-time feedback that she refers to include three parametric modes of operations and interactions that work with external and internal data: program mode, crowd mode and memory mode (Parisi, 2013, pp. 107-9). The system is never kept constant but is instead contingent. The

changing parameters and adaptations to environments are what she describes as “temporal variations” (Parisi, 2013, pp. 107-9).

These “live temporalities,” as Parisi describes them, explore the measurement of time “to the indeterminacies of differential relations” through “unpredictable or intensive relations between present parameters (2013, pp. 110-2). All the parameters that come with the structure, which are not always predictable, as well as the relations between parameters, further shape the live temporalities. The study of temporalities and unpredictability enable an understanding of the feedback activities between computation and environment through both temporal and spatial dimensions, which liveness is emerged through differential relations.

#### *1.3.4 Unpredictability and liveness*

Arguably, one of the exciting aspects of liveness concerns unpredictable events. Something that is unknown, unplanned and unpredictable might happen while an event or a process is unfolding in real-time (Davis, 2007, p. 48; Tate, 2014). Within the context of music performance, Paul Sanden discusses the liveness of spontaneity, which is improvised and “without a predetermined set list” (2013, p. 72). This spontaneity is fundamental to how a musician performs in an unpredictable manner, and which is unique to any performance (Sanden, 2013, p. 159). In live performance, Claudia Georgi examines the relationship between disruption and unpredictability. She argues that “unpredictability, imperfection and failure are inherent aspects of liveness” (Georgi, 2014, p. 152), which is different from pre-recorded materials which demonstrate a relatively stable, controllable and predictable outcome. In theatre and live performance there is always the risk of imperfection and susceptibility to failure because it is impossible to guarantee the act will follow exactly what has been rehearsed or scripted. A human mind and body are indeed unreliable (Georgi, 2014, p. 134).

Following a similar line of enquiry into liveness through unpredictable

human behaviours, media studies scholar Alla Gadassik extends this to the television context. According to her, the essence of liveness is its unpredictability, the possibility of disruptions and “the possibility that anything could happen” during a live program show (Gadassik, 2010, p. 120). She focuses on what she calls “affective corporeal disruptions” that stem from an actor’s behaviour (Gadassik, 2010, p. 118). For example, an audience might witness a participant or a performer who might suddenly cry or lose her/his temper, or s/he might speak outside of a pre-written script, or there might even be unexpected crises. Therefore, Gadassik remarks, “Television performances become most *live* when they break down” (2010, *original emphasis*).

Such breakdowns can also be further understood in relation to the temporal dimension of television through what media studies scholar, Mary Ann Doane calls the “catastrophe machine” in her article *Information, Crisis, Catastrophe* (2006). She defines catastrophe as “unexpected discontinuity in an otherwise continuous system” (Doane, 2006, p. 255), as, for example, with breaking news of earthquakes, explosions, nuclear disasters and plane crashes. These catastrophes disrupt an ordinary routine about what is expected to be seen and heard (Doane, 2006, p. 258). She also introduces the distinction of “dead or alive” as a way of characterising television liveness. She refers to deadness in general as a disruption of continuity, resulting in something that goes wrong just as the case of the appearance of breaking news in normal routines (Doane, 2006, pp. 259-60).

The introduction of deadness is useful in addition to Auslander’s liveness for an analysis of contemporary software culture. Although Auslander’s historical perspective of technological mediatisation and recording technologies demonstrates an understanding of how the notion of liveness has evolved and how it is historically rooted, the analysis inevitably falls short of keeping up with the changing conditions and new complexities of technology and culture. Thinking of liveness and deadness, continuities and discontinuities in which different forces exist in a plane of immanence, opens up unexpected consequences beyond a smooth or continuous flow of



events.

With regard to digital performance, Andrew Murphie proposes thinking of performance beyond living organisms. Similarly to Auslander, with respect to nonhuman machines, Murphie accounts for nonliving beings, in particular signal processing and computational processes (2013, p. 3). He highlights the fact that these nonliving forces are generally imperceptible but contribute to the register of performance. He takes a micro perspective to explain the processes that underline a running machine to address imperceptible, or less visible, forces.

The processes are hidden; literally micro-processes of microprocessors. So these are micro-performances, or, better, we are dealing with a multiplicity of performances, and the resonances of patterns of relation, that are able to scale across micro and macro (Murphie, 2013, p. 3).

Murphie argues that the differential distribution of signals and computational events and intensities constitute what it means to be live. As such, “performance has always been a mix of forces of the living and the dead” (Murphie, 2013, p. 3). His understanding of liveness (and deadness) is developed from a nonhuman perspective that accounts for signal machines, signal processes and their relationships with performance.

As stated this thesis mainly investigates the live condition of real-time technologies, especially distributed networks and databases, and not the presence of living bodies and their mediated representations. It becomes apparent that both temporality and unpredictability are essential for the discussion of digital and networked environments. In clarifying this I incorporate nonliving forces to examine unpredictability, disruption and deadness in computational systems beyond human perception, human bodily behaviours and mediated representation. Attention to the nonhuman dimension suggests that there may be scope for a more dynamic engagement with computational structures and micro-processes (addressed

in detail in Chapter 4) that will provide further insights.

### 1.3.5 A sense of (digital) liveness

In more current discussions of liveness the incorporation of digital technologies is inevitable. For instance, Auslander acknowledges how the term live has been employed culturally to describe human-machine relationships (such as the previously mentioned example of the website ‘going live’) but “digital liveness” to him still primarily points to human interaction with machines or virtual environments (2012). Digital liveness, he claims, “emerges as a specific *relation* between self and other, a particular way of ‘being involved with something” (Auslander, 2012, p. 10, *original emphasis*). Auslander does recognise the intrinsic properties of digital objects and media that are co-constructing the experience of liveness but only once the assumption is made that the human accepts that technology becomes live *for us* (2012, p. 9). This does not fully account for my earlier example, *feedback.pl*, in which code runs and modifies itself in a live environment for instance. Technology becomes live, as I argue, not only for us but also for-itself and for other beings that are beyond the scope of human reasoning and understanding.

Performance and technology studies scholar Sally Jane Norman on the contrary, calls for attention to new ways of “making sense” of technology, to establish “materialized temporal frameworks,” recognising that digital times affect our sense of liveness beyond human understanding and knowledge (2016, p. 3). In other words, the digital comprises of things and events that are both known and unknown. Informed by various digital technologies, digital times, as described by Norman, generate “phenomena at scales that escape our usual reasoning abilities” (2016, p. 6). Informed by this argument, this thesis shifts its attention from a human-orientated or phenomenological approach, such as Auslander’s, to a more nonhuman (or even posthuman) perspective which incorporates the hybridisation of known and unknown phenomena, things that are perceptible and imperceptible,

visible and invisible to human perception. Even though there are micro-processes and invisible forces that might not be observable to human senses, Norman suggests we have to generate imaginary ways of dealing with such hybridisation.

The sense of liveness that I promote in this thesis departs from human centrism and rather opens itself up to nonhuman sense-making. My assumption here is that nonhuman things, such as code and algorithms, are able to sense too. Indeed, the word sense is not something exclusive to humans. In computation, for instance, remote sensing and listening events are commonly found in the Internet of Things (IoT) and in computer programming. One might argue that these sensing technologies are implemented for humans and are derived from human design but how they work technically is highly system and material specific, hence, their interactions are not exclusively human-centric. Technically, in computer science, liveness refers to a property that a program has set up (Owicki & Lamport, 1982; Pradhan & Harris, 2009). The property designates how “some desirable state is repeatedly or eventually reached” (Pradhan & Harris, 2009, p. 14). An example would be how a traffic light eventually turns green or how an outdated software version is repeatedly detected. Such a perspective emphasises the material state of things, and how things are inter-acts beyond the presence of humans, and increasing automated systems, such as tracking agents and bots, can be found in contemporary software culture. It also becomes apparent that automation becomes one of the key and emerging areas for understanding liveness.

To be clear, focusing on the nonhuman aspect is not about ignoring the significance of human interactions and, subsequently, the feeling of human liveness but a change of emphasis. If we take this perspective, we may gain a different understanding and discussion of liveness that can extend our understanding. As such, it is strongly suggested that the notion of liveness is not solely situated or based on the engagement of audiences, nor perceptible representational forms. By situating itself in the analysis of contemporary software culture this thesis suggests that the inter-actions of

things and any sense of liveness are intricately interwoven. Following the discussion of Auslander and Norman on digital liveness, this thesis argues that we need to imagine and generate new ways to conceptualise liveness in keeping with our times.

## **1.4 Aims and Contributions**

Informed by the perspectives of liveness across diverse fields, as outlined in the above sections, the aim of this dissertation is to further develop this discussion in the field of software studies. Taking into account real-time and programmable technologies and networked environments, the live dimension is important for a fuller understanding of contemporary software culture. For the purpose of delimitation, this thesis specifically draws attention to the phenomenon of data queries, data streams and automated agents that are processed in distributed networks, where things are highly connected with devices, machines, systems and networks. The capturing, storing, updating and transmitting of data are actively processed in different kinds of programmable applications, programs and platforms that constitute an immanent sense of liveness. Importantly, automated agents are programmed and mutated transparently and they have the ability to interact with or without direct human involvement. The nonliving or nonhuman forces of code inter-actions co-constitute how we experience liveness in contemporary culture.

Inter-actions take place across computational layers and at multiple scales. The live dimension, as proposed in this thesis, is investigated along three key vectors: unpredictability, temporality and automation. The use of term ‘vector’ makes references to the work of Deleuze and Félix Guattari in their philosophical account of “assemblages,” they explain,

The multiplicity of systems of intensities conjugates or forms a rhizome throughout the entire assemblage the moment the assemblage is swept up by these vectors or tensions of flight. (Deleuze & Guattari, 1987, p. 110)

These vectors can be understood as forces that traverse time through space, propagating across computational layers through the perpetual running and execution of code. These vectors also modulate the sensation of liveness, in which code inter-acts with and through different dimensions, layers and nodes as becoming. The assemblages of relations, in particular code inter-actions, determine the vector of (nonliving) forces.

The three vectors are used to address the complexity and forces that arise from the technological and networked conditions of contemporary culture. The first two of these are based on material I have already introduced briefly in the above sections, addressing some of the pressing issues in existing discussions, especially the inter-actions beyond audience perception and human-machine interactions. The third vector, automation, is more clearly informed by the artwork *Listening Post* but it is also inspired by the increasing availability of automated systems. Automation in computation implies the act of repetition, which is now manifested in bots, machine-learning systems, auto update agents and among many more instances. Blurring the start and the end of a process as well as what is considered to be new or old and inferring the smooth and interrupted micro-processes of running code, the notion of automation implies continuity. In other words, automation is related to how data is being read and written through a repetitive act of code inter-actions, contingently and dynamically producing differences in computational processes. The three vectors—unpredictability, temporality and automation—together form a framework for the examination of the deep computational structures and architecture of these computational processes.

Within this framework, I will explore how code inter-acts with different materials and technologies, expanding the understanding of liveness beyond its immediate reception and mediated representation. Three of my customised software (art) projects will be discussed to address the underlying conditions for three phenomena—data queries, data streams, and automated agents—which I will position in relation to the vectors of

unpredictability, temporality and automation respectively. Coupling practice and theory in this way is used to pay attention to otherwise barely visible code operations and to take seriously their cultural implications. The three projects and their related experiments, trial processes and reflexive thinking will be structured and presented at the end of (and within) each chapter to further examine the phenomena under discussion (Chapters 3-5). The inclusion of these elements in the body of the thesis, rather than as separate appendices, emphasises that although the outcomes of these projects are explained within the chapters, my findings are not only demonstrated in the written text but in the running of the projects themselves. They are not written in an academic style, but rather through a textual method of self-narration and are presented together with screen shots that integrate my experience, observations and reflections in support of the overall argument of this thesis. Together with the examination of these vectors I intend to demonstrate how theory and practice inform each other. I refer to this in the thesis as ‘reflexive coding practice,’ a methodology which I discuss in more detail in the next chapter, alongside other methodological considerations.

It is also important to recognise that this thesis does not intend to define or explain liveness but rather it focuses on what constitutes liveness within a contemporary computational context, drawing attention to previously under-researched constituent parts, such as those imperceptible to humans. It acknowledges the significant and changing role that technology plays in shaping the vague sensation of liveness and departs from previous perspectives (as outlined earlier in this introduction). This thesis asks: how does a materialist framework of liveness reconfigure our understanding of software and expand the discussion of what constitutes liveness? This question is the main line of inquiry for this thesis, examining the technical and cultural aspects of software to inform a contemporary understanding of liveness.

This research contributes primarily to a widening of the focus of critical attention in software (art) studies through a close analysis of data queries,

data streams and automated agents. With a distinctive focus of the live dimension of code inter-actions, it presents the vectors of unpredictability, temporality and automation. This thesis develops what I call “reflexive coding practice” to examine these live phenomena and it is an applied approach to computational processes and a means by which to reflect on cultural issues through experimentation and practice. Furthermore, the thesis expands the debate in media and performance studies, providing technical description and analysis in relation to the concept of liveness. In overall terms, the research contributes to our understanding of software by expanding our understanding of liveness in contemporary culture. This includes a nuanced examination of liveness beyond immediate human reception.

The process of research for this thesis has revealed that there are relatively few female voices in the interdisciplinary field of software studies. Although, historically, women played a significant role in the development of programming and computation, it is generally agreed that there is an under-representation of women in the fields of science, technology, engineering, and mathematics (STEM subjects) (Hill et al., 2010). The reasons for this gender gap, and the strategies for bridging it are more multiple and complex (Etzkowitz et al., 2000) but it is widely acknowledged that a lack of women role models in STEM subjects in schools and universities negatively contributes to the problem (McIntyre et al., 2005). Furthermore, by contributing to the field of software (art) studies as an East Asian woman with an open attitude that champions gender and race equality and diversity I recognise that I am a de facto role model, and contribute in some way to addressing the stereotypes associated with software studies, whether I focus explicitly on issues of gender and race or not. Therefore, while gender and race are not the focus of this research, I recognise that my subject position ‘matters’ to the research I have undertaken (Barad, 2007, p. 57; 2012, p. 80).

## 1.5 Chapter Overview

The structure of this thesis acknowledges the complexity of code interactions. It reframes the object of study from code to code inter-actions, which are integrated into the processes of contemporary software culture. It argues that liveness is in part constituted by code inter-actions, in which processes are executed at various layers and at multiple scales beyond immediate reception, in the process of executing and running of code. This thesis aims to demonstrate a materialist framework, a live dimension of code inter-actions, to address these conditions. The framework uses and develops cultural, theoretical and practice-oriented approaches to make a material and critical account of the process of code inter-actions through the three vectors that are employed to understand the live condition, which is both culturally and technically entangled. It is organised across 6 chapters, including this one, Chapter One.

Chapter Two, 'Approaches to code inter-actions,' provides an in-depth discussion of the field of software (art) studies. It presents key concepts related to the understanding of contemporary software culture, beginning with the notion of invisibility that is associated with the materialisation of code and the opaqueness of computational processes. Secondly, the concept of performativity is introduced to examine the relationship between code and language as well as the operational logics of code that produce performative effects and highlights machine agency as a way to think about the materiality of code. Thirdly, the notion of generativity is discussed inasmuch as it introduces a certain degree of autonomy in a given system. Exploring these three concepts leads to a better understanding of some of the current debates in the field of software studies and acts as a complementary force to the three vectors of liveness that I will introduce in subsequent chapters.

In addition, Chapter Two presents a distinctive perspective on nonhuman agency and code inter-actions, which I refer to as a materialist approach in recognition of (feminist) new materialism. This overarching conceptual



framework for the whole thesis foregrounds the materiality of code interactions. It highlights the concept of code inter-actions that are comprised of “interactions” (Beaudouin-Lafon, 2008; Bentley, 2003; Murtaugh, 2008; Wegner, 1997) and “intra-actions” (Barad, 2003, 2007) that produce forms of agency. Code inter-actions examine code and its dynamic relations and inter-actions with other materials that underpin the underlying computational structure and operations therein.

The chapter also discusses the methodology, which I call ‘reflexive coding practice,’ that is informed by the field of artistic research (Borgdorff, 2011, 2014; Rolling Jr, 2014; Sullivan, 2010). Following the tradition of software studies in examining code and digital objects, it pays attention to code reading, writing, running and execution, intertwining theory with practice to think through the cultural implications of code inter-actions. Informed by the ‘close reading’ of critical code studies, ‘cold gazing’ in media archaeology and ‘iterative trials’ in software studies, these methods allow me to delve into code structures, observing and sensing how things operate and how materials are subject to inter-action. The artistic projects that I have developed are also to be regarded as forms of knowledge in parallel to the written form of this manuscript. This chapter argues that the execution of code is also a site of knowledge production and this is echoed in the thesis title.

After situating this thesis in the field of software studies and discussing my conceptual and methodological considerations in Chapter Two, the materialist framework of liveness is introduced across Chapters Three to Five. Liveness is examined in two main ways. Each chapter focuses on a contemporary condition which is based on computational phenomena observed in our culture, one of the specific vectors is then introduced. As part of the reflexive coding practice, each vector is used to examine an experimental art project. Therefore each chapter engages with and reflects upon a different platform, coding and networked environment, alluding to their inter-actions that inform the concept of liveness.

Chapter Three, ‘Executing Unpredictable Queries,’ investigates the unpredictable vector of liveness. Informed by computational media scholar Noah Wardrip-Fruin’s analysis of the computer-generated program *Loveletters* (1952), this chapter examines similar computational processes but in a networked environment. It takes my collaborative project *Thousand Questions* (2012-2016) as an example. *Thousand Questions* takes ‘questions’ from the internet as text and ‘voices’ them. This chapter discusses the format of query as a cultural form that has been widely adopted in cultural and artistic contexts. Findings are also based on this artistic project, offering an analysis of the unpredictability of live queries and how they inter-act with databases and network protocols.

The analysis consists of a discussion of the relative openness and closedness of the internet micro-blogging platform *Twitter*. By applying the concept of generativity, it explains how Twitter is a dynamic platform that generates random events. The chapter also draws upon information theory (Shannon, 1948; Weaver, 1949) and algorithmic information theory (Chaitin, 1987/[1975]) to underline the relationship of randomness to unpredictability in information processing. Furthermore, following N. Katherine Hayles’ concepts of ‘microscopic events’ and ‘macroscopic chaos’ (1990), the chapter identifies how mathematical operators play a major role in querying data at a microscopic level, arguing that they generate a temporal, unpredictable and dynamic relation of data that cannot be produced in the same way repeatedly. Lastly the possibility of disruption is introduced, which I call ‘inexecutable query.’ I use this concept to think through business logics, cultural operations and political decisions across seemingly smooth and uninterrupted computational processes. The chapter argues that what makes digital objects live is not the disruptive moment where things do not function technically but rather the possibility of disruption—the forces of deadness—which occur at any time that queries are executed through a technological network beyond a programmer or user’s control. This inexecutability operates at high levels of unpredictability, uncontrollability and unknowability across time in which a query is made inexecutable. The material forces in part constitute the notion of unpredictability.

Chapter Four, ‘Executing Micro-temporal Streams,’ addresses the temporal vector of liveness. In particular, it is based on the spinning and loading icon, commonly known as the ‘throbber,’ that often appears while waiting for social media feeds, streaming videos and content in contemporary software culture to load. This chapter further draws upon Wolfgang Ernst’s notion of “micro-temporality” (2013b), to examine the underlying complex and temporal activities of real-time data processing that is running behind the abstract form of the throbber. Micro-temporality addresses the micro-events of signal and operative processing, as well as computer execution and network synchronisation.

The chapter begins with a cultural reading of a throbber, examining its use in both the historical and contemporary context of software (art) practices. It proposes the term ‘discontinuous micro-temporality’ to rethink the notion of flow and stream in networked environments. Drawing upon computer scientist Paul Baran’s ‘packet switching mechanism’ and media and cultural studies scholar Florian Sprenger’s notion of ‘micro-decisions,’ the chapter further unfolds the operative processes of data transmission in distributed networks beyond the linear and continuous flow of time. By using the method of the ‘cold gaze’ (Ernst, 2013b, pp. 186-9), the chapter analyses digital signal processing, data packets and network protocols, the buffer and buffering within its deep internal and operational structures. It demonstrates that any perceived stream comprises the micro-temporality of the inner-workings of data processing which is discontinuous in distributed networks. Informed by this understanding of micro-temporality and the buffering of data streams, it calls for critical attention to the gaps, ruptures, pauses and silence of streams.

My experimental project entitled *The Spinning Wheel of Life* (2016) is discussed as it emphasises the micro-temporal dimension of code interactions that are manifested in the operations of a throbber. There are different processing rates, tempos, pulses and rhythms running at multiple scales—from the operations of the CPU to network routers, from sender to

receiver and from continuous streams to discontinuous packets. It highlights the temporal and spatial dimensions of data streams. Central to the argument of both chapters Three and Four are the inter-acting relationships between code, signals, network protocols, computer memory and the buffer, data processing and databases, query formats and operators which together constitute the vague sensation of liveness. By analysing the deep query and micro-temporal structures and processes these chapters explicate the complex inter-actions of code.

Chapter Five, 'Executing Automated Tasks,' examines the third vector, automation. Automated systems enable real-time computation, tracking and querying of data, responding to the live environment without human intervention. Automation is the act of repetition by pre-set algorithms, performing with changes and differences. This consists of spam, bots and various sorts of notifications that are perpetually active and processed in the background, ever-updating and ever-proliferating. The chapter centres upon Chun's notion of 'undeadness' (2008, 2011) which emphasises endless updating of code and circulation of data in a networked environment. Through an analysis of my project *Hello Zombies* (2015) the chapter closely examines three main code syntaxes, namely 'loop', 'open or die' and 'try and catch exceptions,' demonstrating how automated agents enable and disable certain activities while maintaining the perpetual running of code.

Going beyond the human sensation of liveness, the chapter further draws upon the writing of Alan Turing (1937), Ernst (2009), Chun (2011), Parisi and Beatrice M. Fazi (2014) who variously discuss the notion of ending in computation, how a process can come to an end or completion. The chapter examines undeadness as a conceptual and technical counterpart to liveness in order to think about whether there is (and what might be described as) the 'end' of a running program. It argues that the assemblage of forces, in particular the highlighting of contradictory and unknowable forces, which constitute the notion of liveness in computation, exhibit unpredictability through systemic automation.

Finally and in conclusion (whilst recognising that this comes without an ending as such, reflected in the title), Chapter Six, 'Unfinished Thesis,' brings together the key elements of the dissertation together, in particular the three vectors, in order to discuss potential future paths for the conceptualisation of liveness. The overall argument of this thesis is reiterated: that liveness is about code inter-actions, a continuous process of executing and running of code that inter-acts across various computational layers at multiple scales. The detail of this is contained in the various chapters.

Throughout the thesis as a whole and as reinforced in its title, I present the central notion of 'executing liveness' in order to assert the importance of the contingent and complex computational processes that execute liveness. The written thesis, together with the three projects presented herein, provides a materialist framework to examine the live dimension of code inter-actions and to produce new understandings of both technical and cultural layers of liveness

## 2

# Approaches to code inter-actions

This thesis is primarily situated in the field of software studies, in which programmable logic is arguably one of the key components that are embedded in many kinds of interfaces, devices and media in our culture. In 2001, Lev Manovich observed this is fundamental different from older media forms like print, photography and television (2001, pp. 47-8). The media that we are experiencing now is a distributed environment, situating in networked and data-driven landscape. Although networked and distributed applications are not something new in everyday life (examples such as the automated teller machine<sup>14</sup> and the bulletin board system<sup>15</sup> began operating in the public domain in the 1960s and 1970s respectively) the web and internet dominate almost every aspect of life in the twenty-first century from finance and communication to entertainment and educational sectors to name but a few. Contemporary software culture denotes networks of machines that are operated through a highly programmable and distributed environment, demanding real-time data circulation, dissemination and processing. The term software studies was coined by Manovich in his early book entitled *The language of New Media* in 2001 (pp. 47-8). He argued that existing media theory did not provide sufficient critical tools to understand this computational world and he suggested turning to computer science, shifting the interest and attention from media to software in order to understand programmable logics and parameters, as for instance, evident in interface and database (Manovich, 2001, pp. 47-8). Crucially, software studies is different from, although deeply related to, computer science in that it is about the culture and practices of software, computation and technology in its broadest sense. As Manovich puts it, software studies is “to

---

<sup>14</sup> It is claimed that cash dispenser machines first appeared in Japan and the United Kingdom in 1966 and 1967 respectively (Bernardo, 2009, p. 6).

<sup>15</sup> In 1978, Ward Christensen and Randy Suess created the first dial-up public bulletin board allowing exchange of files and information (Taboada, 2004, p. 59).

investigate the role of software in contemporary culture, and the cultural and social forces that are shaping the development of software itself' (2013, p. 10).

This chapter aims to contextualise the emerging field of software studies by reviewing some of the existing and relevant literature to understand the current debates in the field and to establish a foundation on which the concept of liveness can be situated. This chapter uses textual analysis supplemented by the discussion of specific instances of software art to demonstrate some key concepts in the field of software (art) studies which examine different forces beyond technical innovation and functions. Many software artworks pay attention to the specific technical/cultural qualities that explicate the general concerns in the field of software (art) studies. By drawing upon the scholarly discussions in software studies, three key concepts which inform the understanding of distinct processual and expressive nature of code, namely invisibility, performativity and generativity, will be examined in detail. Finally, this chapter outlines the conceptual and methodological frameworks for the whole thesis, revealing the approaches and methods to the study of code that explore the constitution of liveness.

## 2.1 Software Art

Software studies makes an argument for a close relationship between art and culture. Specifically, there are increasing numbers of artists whose works explore the notion of liveness, utilising real-time computation, live data and dynamic systems which operate on, and through, a technological network to create artworks. For example, the work *Live Wire* (1995) by Natalie Jeremijenko exposes the dynamic of immaterial information through visualising network traffic in real-time. In past decades the unprecedented growth of social media sites and online platforms have fostered the querying of data through programmable technology as observed in works such as *The Sound of Market* (1996), *Listening Post* (2000-2001), *Toy Town* (2009), *Thousand Questions* (2012), *Read for us...And show us the*

*pictures* (2015) amongst others. Table 1 demonstrates some of the software artworks that particularly emphasise and explore the various dimensions of liveness.



## Approaches to code inter-actions

Year	Artwork	Artist	Excerpt of artwork description
1995	Live Wire	Natalie Jeremijenko	“The <i>Live Wire</i> is a 3D, <i>real-time</i> network traffic indicator”(Jeremijenko, 1995, <i>my emphasis</i> ).
1996-2001	Image/ine	Steina Vasulka and Tom Demeyer	“Image/ine is a Macintosh program that allows a user to manipulate visual source material in a <i>live performance</i> environment” (Vasulka & Demeyer, n.d, <i>my emphasis</i> ).
1996	The Sound of Market	Henry Chu	“This work uses the stock chart of Hong Kong stock market, downloaded in <i>real time</i> , according to which stock code the user puts in. The chart will be analyzed, and transformed into music notes” (Chu, 2007[1996], <i>my emphasis</i> ).
2000-2001	Listening Post	Ben Rubin and Mark Hansen	“Listening Post is a <i>dynamic portrait</i> of online communication, displaying uncensored fragments of text, sampled in <i>real-time</i> , from public internet chatrooms and bulletin boards” (Hansen & Rubin, 2000-2001, <i>my emphasis</i> ).
2004	www.is-a-living.org	Mushon Zer-Aviv	“It is using a physical computed tracking system conjunct with <i>live internet search-engine activity</i> , interactive visuals and sounds and a game system to deliver the concept” (Zer-Aviv, 2004, <i>my emphasis</i> ).
2005	Lyric economy	Electroboutique	“Poetic Economy visually deconstructs a traditional poetic text (Goethe, Shakespeare, Pushkin, you to decide) and replaces it with a news feed coming in <i>real time</i> through RSS channels” (Electroboutique, 2005, <i>my emphasis</i> ).
2009	Toy Town	Ellie Harrison	“Together they form part of a trilogy of new works which use specially designed software to respond <i>instantaneously</i> to news headlines reported in the BBC News RSS feed” (Harrison, 2009, <i>my emphasis</i> ).
2012	Thousand Questions	Winnie Soon and Helen Pritchard	“The questions are gathered in <i>real-time</i> from the social media site Twitter and encoded to speech. Listening is a form of decoding, and in this work the machine <i>constantly</i> undergoes the process of editing, encoding and decoding texts” (Soon & Pritchard, 2012b, <i>my emphasis</i> ).
2012	A live portrait of Tim Berners-Lee (an early warning system)	Thomson & Craighead	“A <i>live portrait</i> of Tim Berners-Lee (an early warning system) is a drawing made from two <i>live</i> cameras located on opposite sides of the world and eleven time zones apart from each other” (Craighead, 2012, <i>my emphasis</i> ).
2014	Radiancescape	XCEED	“It based on the <i>live radiation data</i> from the Safecast.org, a global sensor network for collecting and sharing radiation measurements, to generate a cityscape” (XCEED, 2014, <i>my emphasis</i> ).
2014	The Pirate Cinema	Nicolas Maigret	“The project is presented as a monitoring room, which shows Peer-to-Peer transfers <i>happening in real time</i> on networks using the BitTorrent protocol.[...] <i>This immediate and fragmentary rendering</i> of digital activity, with information concerning its source and destination, thus depicts the topology of digital media consumption and uncontrolled content dissemination in a connected world” (Maigret, 2014, <i>my emphasis</i> ).
2015	Show us the pictures ‘Some Thing We Are’	John Cayley and Daniel Howe	“The Readers Project presents the work of a software entity that generates digital video montage, with visual content sourced through <i>live image search</i> ” (Cayley & Howe, 2015, <i>my emphasis</i> ).

*Table 1: A selected list of (software) artworks that address the notion of liveness*

The first use of the word ‘software’ in an exhibition title can be traced back to the 1970s. Artist and critic Jack Burnham curated an exhibition called *Software - Information Technology: Its New Meaning for Art* at the Jewish Museum in New York (1970-1971). This exhibition opened up a way of thinking software beyond the confines of engineering. According to Burnham, the exhibition was focused on information processing that “[dealt] with underlying structures of communication or energy exchange instead of abstract appearances” (1970, p. 12). This focus on process rather than appearance suggests that code must be understood together with system procedures and data sources, or that the resulting representations would have a richer, or alternate meaning if the system procedures and data sources were understood. Software structure, for Burnham too, does not limit itself to languages, programs and procedures (that is the ‘form’ of written code) but includes system design, such as flow diagrams and ‘system procedures’ (1970, p. 12). He asserted that the notion of software should be expanded to include “any kind of data” that is taken “from the environment by a system, living or organic” (Burnham, 1970, p. 12). In addition, Burnham’s understanding of software is a creative and expressive medium that “represent[s] the programs of artists,” including their views and concepts. Burnham’s words and his curated exhibition indicate that software not only refers to a specific piece of code but rather to the processes and other living systems which are also part of it. His interest in the morphology of structure and processes of materialisation can be said to demonstrate how materials are inseparable from the final ‘object’ (or appearance) of software. In the words of the materialist Manuel DeLanda, the notion of “morphogenesis” describes such an entanglement between stable forms and material processes (1995, 2003).

On the contrary, the exhibition *CODEDOC* (2002), which was curated by Christiane Paul for the Whitney Museum online portal,<sup>16</sup> took the medium of code as its focus and presented twelve commissioned artworks in the form

---

<sup>16</sup> Paul took the concept of *CODEDOC* and developed *CODEDOC II* (2003) at the digital art festival Ars Electronica. For all the artworks and the curatorial statement of CODEDoc, see: <http://artport.whitney.org/commissions/codedoc/>.

of source code and executed results. The intention of the exhibition was to “explore the relationship between the underlying code of software art and its results” (Paul, 2003, n.p). Such presentation of the works draws attention to the procedures, rules and grammar of code. Paul regarded code as a form of creative writing in which both the artists themselves and how the code is structured become the integrated whole of the creative process (2002, 2003). These procedural and stylistic instructions are specific to different programming languages. The exhibition of instructions, rather than the visual representation that emerges from them or the result of the work, becomes the focal point as the audience moved between the front and backend of the artworks (Paul, 2002). This is also demonstrated in the recent award-winning artwork *50. Shades of Grey*<sup>17</sup> (2015), Hong Kong’s artist-programmer Bryan Chung shows 6 different programming languages, in the form of both source code and executed outcomes, which create the same graphic pattern of fifty shades of grey tone. These kinds of artwork suggest source code as a material, indicating underlying structure and processes which are distinct from the finished form of the (art) objects.

The focus on technical materials and processes also challenges the perception and meaning of art and whether software art can be considered art at all (Cramer, 2003, n.p; Goriunova & Shulgin, 2004, pp. 19-20). As noted by media theorist Florian Cramer, software art is a genre that addresses the materiality of software, rather than reflecting the mainstream contemporary art scene in which artists are the core participants who make the meaning of the work manifest in many contemporary art forms such as installation, video and audio/sound art (Cramer, 2003). Software art’s attention to its medium or material specificity shares values with net art, which has often described itself as an anti-aesthetic (Stallabrass, 2003, p. 49). As Paul explicitly stated in *CODEDOC*: “Visual Beauty does not have to be the main focus” (2002, n.p). One could argue that software art is similar to conceptual art in the 1960s where technology was not prevalent and the human idea was favoured above its representation. Indeed such a

---

<sup>17</sup> The artwork *50. Shades of Grey* received the Grand Prize of the Japan Media Art Festival in 2015. See: <http://www.magicandlove.com/blog/artworks/50-shades-of-grey/>

connection to conceptualism has been traced by scholars such as Jullian Stallabrass (2003), Edward Shanken (2002) and Alexander Galloway (2004). In particular, software studies scholars Christian Ulrik, Andersen and Søren Pold argue that software art promotes our critical awareness of software culture and evokes debates on the art form itself. Specifically, one of the claims about software art is that such an art form “has the power to contemplate its own materiality and language” (Andersen & Pold, 2004b, p. 13). As such Andersen and Pold argue that software art can be regarded as a conceptual art form (2004a, p. 13).

Software art finds its roots in the net art movement in the 1990s (cf. Bookchin & Shulgin, 1999, n.p; Goriunova, 2012, p. 74). Yet this is not straightforward and browser art, for instance, blurs sub genres that are hard to classify as either network or software centric. Importantly Cramer points out that many software artists consider the wider network or computer system as part of the code materials (2003). Although net art places its focus on the internet network, it often requires the presence of software. Conversely, everyday software is operated under and within a network. For example, software requires an internet connection that deals with live data feed, or a script which runs in a web browser (a HTML web page for instance). Networks and software are integrated and increasingly blurred and they inter-act with one another and form new relations that undermine the separation between the two. JODI's earlier work *GEO GOO* (2008) (see Figure 1.1) demonstrates the intertwining of software and network. This hacked Google map calls for our attention to daily proprietary software and familiar interfaces and icons that are used in everyday network culture. *GEO GOO* is not a narrative form of art but it presents in a fairly messy interface, something like a malfunction map, by appropriating Google Maps and icons in a manner that disrupts how may we normally think of a map and the consequences of how Google has changed the conception of a map in everyday practice. One may argue that JODI's work shows the concept of digital interventions by destabilising and presenting special kinds of internet materials in which the idea behind, the concept, in the work takes over its visual beauty.

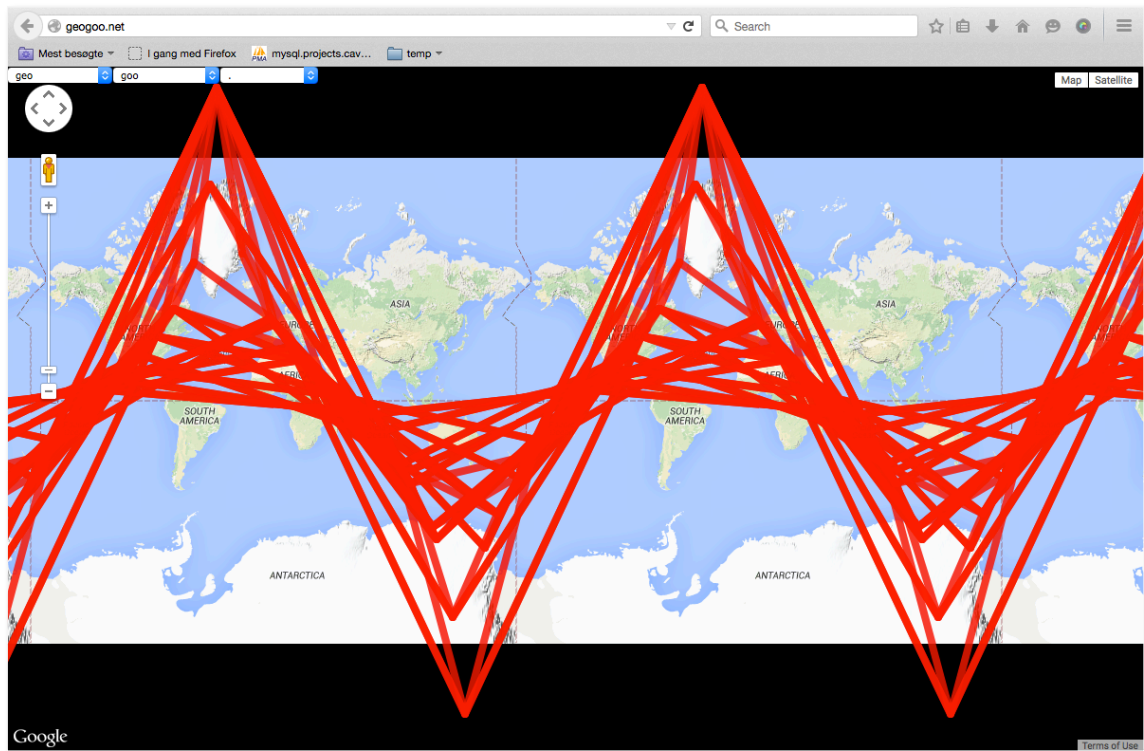


Figure 2.1: *GEO GOO* (2008) by JODI. This figure is a screen shot from geogoo.net.

However, there are also other voices in the field for whom these kind of art forms are not equivalent to conceptual art. Insofar as artists in the realm of net art/software art take into consideration material agency such as “faulty code,” “crashes,” “incompatibilities” and “viruses” which are specific to the network or the software, they are different from conceptual artists (Cramer, 2003, n.p). Expanding on this the concern with material agency becomes the subject of the artworks. Artists explore the materiality of software and network, expressing and presenting their work in the same form with the same subject. For example, the piece *Biennale.py* (2001)<sup>18</sup> is a computer virus distributing as art at the 49<sup>th</sup> Venice Biennale that intrigued the matter of software legality. Moreover the work was also distributed as open source software that was programmed with Python, allowing the ‘variable names’ to be customised and seen by the public. As such the source code is recognised as not only “a love poem” but also as code that can perform on its own once executed (0100101110101101.ORG & epidemIC, 2004, np).

---

<sup>18</sup> The piece *Biennale.py* (2001) was developed by 0100101110101101.ORG and epidemIC. See: <http://epidemc.ws/biannual.html>

Alluding to artworks that express the agency of code by using code, such attention to the code's materiality of indicates an integral concern with code as both object and subject. This stands in sharp distinction from using code as a tool to merely present another artistic concept.

It is perhaps necessary to emphasize how software art differs from other art forms that also have a focus on materiality, for example those filmmakers who declared themselves as structuralists/materialists. The ability of code to execute and run in different conditions may produce distinct behaviours which is unique to software art. The same webpage can be rendered distinctively by different web browsers<sup>19</sup> and by the condition of the machine. A further example is *Listening Post*, which has been discussed in previous chapter, which takes in dynamic data and presents it in a multimedia form. Generally, film has hardly moved away from playback. Although one could argue that film materialists investigate different aspects of the medium of film presentation via projections, screens and lens or image production processes with distinctive printers and papers, the characteristics of software are rules, languages and programmability, most importantly, however, are the live conditions of the execution processes.

In the realm of software art, code or software is often not considered as a practical tool but as a thing that expresses itself. As media theorist Tilman Baumgärtel puts it:

Software art is not art that has been created with the help of a computer but art that happens in the computer. Software is not programmed by artists, in order to produce autonomous work, but the software itself is the artwork. What is crucial here is not the result but the process triggered in the computer by the program code (cited in Cox, 2007, p. 150).

---

<sup>19</sup> The artwork *Scrollbar Composition* (2000) by Jan Robert Leegte would be an illustration of this. He presented the work in the exhibition *Electronic SuperHighway* at Whitechapel Gallery in London in 2016, where there were three screens containing three web browsers and each ran the same piece of code differently. The screens showed different stylistic scrollbars, as well as micro differences of rendering speeds.

This attention to self-expression differentiates the field of software studies from the growing discipline of digital humanities (DM). In DM, software is often regarded as a practical tool for sociological or humanistic analysis, usually in the form of processing large amounts of data for statistical analysis (Fitzpatrick, 2012, p. 13; Svensson, 2012, p. 41). In the words of DM scholar John Unsworth, “the computer is used as tool for modeling humanities data and our understanding of it” (2002). One could argue that software artworks use technology in creating an expressive form of work. Indeed, software art does not only place emphasis on using software as a tool, to make works or generate another expression but is also “using code as an expression itself” (Andersen & Pold, 2004b). Using the term ‘expression’ draws attention to the argument that code consists of non-neutral commands, not taking for granted any syntax or even punctuation.

An example of the expression of non-neutral code is a software artwork called *Whitespace*<sup>20</sup> (2003) produced by Edwin Brady and Chris Morris (see Figure 1.2). White space is commonly seen in computer programming language representing horizontal or vertical space. White space is a character or a series of characters, for example in HTML a white space symbol is written as ‘&nbsp;’. However, the artwork *Whitespace* is a programming language without a practical value insofar as it questions the injustice of white space, a syntax that is often hidden or ignored in computer programming culture. The artwork ignores any non-whitespace characters and only gives meaning to various forms of white space,<sup>21</sup> ‘Space’ and ‘Tab’ for instance. White space can be seen as a programming standard for indentation but also as an expression of aesthetic appeal and readability (Lazaris, 2013). Furthermore the demands of white space vary across programming languages such as Python, which requires a strict indentation and space character position for further layers of processing.<sup>22</sup> As such the

---

<sup>20</sup> See the work details, source code and later developed version:  
<http://runme.org/project/+whitespace/>, <https://github.com/hostilefork/whitespacers/> and  
<https://github.com/igorw/whitespace-php>

<sup>21</sup> See: <http://compsoc.dur.ac.uk/whitespace/tutorial.html>

<sup>22</sup> See: <https://www.python.org/dev/peps/pep-0008/#whitespace-in-expressions-and-statements>

white space may be understood as a transformative force in itself, interacting with code, language and culture. It traverses the material and its representation in a seamless way. This kind of translation is specific to software art, distinguishing it from other art forms such as film and painting, in which instructions can be activated through layers of computational processes. Paul explains such inter-acted layers of code as central to software art in the following passage:

In software art, the *materiality* of the written instructions mostly remains hidden. In addition, these instructions and notations can be instantaneously activated; they contain further layers of processing and are the artwork itself (2003, n.p, *original emphasis*).

The work *Whitespace*, and many other software artworks, are archived at Runme.org, a software art repository established in 2003. The code is available for analysis as well as its function as an archive of cultural activity. Focusing on art and culture, the associated Readme festival,<sup>23</sup> which took place at the Macros-center in Moscow<sup>24</sup> (2002), Media Centre Lume in Helsinki<sup>25</sup> (2003), Rum46 in Aarhus<sup>26</sup> (2004) and HMKV in Dortmund<sup>27</sup> (2005), promoted this artistic, experimental and expressive genre and is regarded as an important event that shaped the field of software studies. Software art expresses a response to the inhabitants of different kinds of software activities, such as programming, experimentation and open source culture.<sup>28</sup> According to Andersen and Pold, software art addresses the culture of software, promoting the awareness of the cultural consequences of software through artistic practices (2004b, p. 12).

---

<sup>23</sup> readme festival promoted the artistic practice of software.

<sup>24</sup> See: [http://web.archive.org/web/20021212040735/www.macroscenter.ru/read\\_me/abouten.htm](http://web.archive.org/web/20021212040735/www.macroscenter.ru/read_me/abouten.htm)

<sup>25</sup> See: [http://www.m-cult.org/read\\_me/report.htm](http://www.m-cult.org/read_me/report.htm)

<sup>26</sup> See: <http://readme.runme.org/2004/conference.php>

<sup>27</sup> See: <http://readme.runme.org/>

<sup>28</sup> An exhibition titled *Open Source Embroidery: Craft + Code* was held in 2008 at the HTTP Gallery in London.



```

;initial blanks in a file (above)
;trailing blanks
;trailing blanks
;trailing blanks
;trailing blanks
;trailing blanks
;space before tab
;space after tab
;indentation
;indentation
;blanks at end of file (below)

```

~/pub/emacs/gnu-whitespace/whitespace-example.el Top (1.0) (Emacs-Lisp ws Fil  
Whitespace mode enabled

Figure 2.2: *Whitespace* (2003) by Edwin Brady and Chris Morris. This image is retrieved from [http://www.elistmania.com/juice/10\\_esoteric\\_programming\\_languages/](http://www.elistmania.com/juice/10_esoteric_programming_languages/).

Such practices that come with the awareness of software culture have been also formally established in other annual and well-known institutional festivals, continually obtaining worldwide attention. For instance, Transmediale (the festival for art and digital culture) introduced “artistic software” as a new category in 2001 that has been championed through its exhibition and symposiums. Furthermore, code gained major attention at Ars Electronica, one of the biggest digital art festivals, in 2003. There were three thematic domains<sup>29</sup> under the theme of ‘Code—The Language of our Time,’ they were: Code=Law, Code=Art and Code=Life. Such domains provided a framework to make explicit the cultural and political consequences of code, reflecting on the matters such as privacy, copyright, control, aesthetics and norms that are related to software. Software becomes

<sup>29</sup> See: [http://90.146.8.18/en/archives/festival\\_archive/festival\\_overview.asp?iPresentationYearFrom=2003](http://90.146.8.18/en/archives/festival_archive/festival_overview.asp?iPresentationYearFrom=2003)

a critiquing tool within artworks, with a role beyond merely technical functions and operations (as illustrated in the artworks *Biennale.py* and *Whitespace*). As the director of Ars Electronica, Gerfried Stocker, reminds us, this type of art “is always aesthetic research, critical analysis and social critique of our scientifically, technically conditioned view of the world” (2003, p. 11).

This focus on criticality through art practice is continued to flourish through various institutions across the globe such as the *Computational Culture Journal*<sup>30</sup> and *A Peer-Reviewed Journal About*<sup>31</sup> which are both online journals that offer critical software (art) discourses, and operate in the United Kingdom and Denmark respectively. Collectives such as Constant<sup>32</sup> (based in Brussels since 1997) and Critical Software Thing<sup>33</sup> (an international group founded in 2015) both address particularly on critical discourse and social critique of software and computation through workshops, writings, seminars and artistic practice.

There is no standard way of working within software art inasmuch as artworks can be presented in many different forms from the sonic to the visual, from sculpture to installation. Nonetheless, software art commonly engages with the social, political and critical attention devoted to code and computational processes in a broad way. There are numerous cases where self-declared non-artists enter different festivals and exhibitions (Readme is a case in point) and this can be traced back to the exhibition *Cybernetic Serendipity*,<sup>34</sup> curated by Jasia Reichardt, at the ICA London in 1968.

I am not going to argue in this thesis whether software art is art given the nature of the involvement of computation. Whether such art engages with other dialogues in aesthetics and art philosophy is not central to my thesis.

---

<sup>30</sup> See: <http://computationalculture.net/>

<sup>31</sup> See: <http://www.aprja.net/>

<sup>32</sup> See: <http://constantvzw.org/>

<sup>33</sup> See: <http://softwarestudies.projects.cavi.au.dk/index.php/CriticalSoftwareThing>

<sup>34</sup> The exhibition *Cybernetic Serendipity* showcased works from artists and scientist-engineer engineers, exploring the intersection between technology and art (Reichardt, 1968, p. 10).

Rather, in this section in particular and in this thesis in general, I seek to highlight the orientation of software (art) practice in which code is neither a purely a technical matter, nor a cultural matter. Indeed software (art) practice may be considered as “entangled material practices” (Barad, 2007, p. 25) in which the technical and cultural can no longer be considered as separate categories and this is also the position of the field software (art) studies in this thesis.

## 2.2 Software Studies: Three key concepts

The discussion of software art within the field software studies started to take shape in the 2000s. In his earlier book, *Behind the Blip* (2003), artist and scholar Matthew Fuller discusses his collaborative software artwork *The Web Stalker* (1997-1998) at length. In his book Fuller suggests that we pay attention to software not at a general level but at the level of particular programs or objects to better understand how the dynamic of things “are networked out into further vectors, layers, nodes of classes, instrumentalisations, panics, quick fixes, slow collapses, the sheerly alien fruitfulness of digital abundance” (2003, p. 18). In other words, software can be studied through artworks or digital objects in a broad sense. Fuller argues that such investigative approaches to the study of digital objects are apparently quite different from how we learn about software in science as they involve social and political implications. In 2003, following Fuller’s suggested approach to studying specific objects, he published the first book that used the term software studies in its title—*Software Studies: A Lexicon* (edited 2008). All contributing writers were asked to contribute a short study on a particular digital object such as, algorithm, code, class library, interface, loop, memory among others, to open up new ways of studying these terms beyond the technical and with critical and cultural perspectives. The book was published by the MIT Press and it prompted the software studies book series<sup>35</sup> in August 2008 which foregrounded the importance of software in contemporary culture. As the MIT Press’ website describes it,

---

<sup>35</sup> See: <https://mitpress.mit.edu/books/series/software-studies>

[t]he Software Studies series publishes the best new work in a critical and experimental field that is at once culturally and technically literate, reflecting the reality of today's software culture. The field of software studies engages and contributes to the research of computer scientists, the work of software designers and engineers, and the creations of software artists. Software studies tracks how software is substantially integrated into the processes of contemporary culture and society. It does this both in the scholarly modes of the humanities and social sciences and in the software creation/research modes of computer science, the arts, and design (The MIT Press, 2016).

Software studies is an emerging interdisciplinary field. It connects the territories of science and the humanities. Over the past two decades, programmable technology has permeated our environment in all sorts of areas from education to business, to entertainment, design and art. On the one hand, the reasons we need to understand software culture is the pervasiveness with which technology influences culture, human behaviours, and even how we read, act and participate in this computational world. On the other, according to Chun, software culture is about knowledge-power renegotiation within institutions, social and economic systems, and, arguably, one could begin to engage with software differently (2011b, p. 21). This is evident in open source software movement, various open data initiatives,<sup>36</sup> establishment of creative commons and ethical companies like Fairphone.<sup>37</sup>

The field of software studies has a specific interest in what David M. Berry describes as “computationality” (2011, p. 10), Adrian Mackenzie names as “softwarily” (Mackenzie, 2006, p.1) or what Chun and Manovich refer to as “programmability” (Chun, 2011b, p. 21; Manovich, 2001, p. 49). This focus,

---

<sup>36</sup>Initiatives such as Open Data Hong Kong and Open Data Aarhus which advocate availability of public data. See: <https://opendatahk.com/> and <https://www.odaa.dk/>

<sup>37</sup> See: <https://www.fairphone.com/en/>

however, does not champion new technology and invention, nor promote scientific knowledge of mathematics and predictive models or similar but rather is about asking what are the implications of such computational and programmable logics? In other words, software studies is about having a critical and broader understanding of software, examining the conditions that explicate social, cultural, aesthetic and political relations.

Delving into a deeper discussion of the field software studies in which this thesis is situated, there are three key concepts that are fundamental to software culture and which provide a basis for further understanding of the constitution of liveness. In particular, the notion of liveness that this thesis suggests is indeed highly related to computational processes, where computer code is important in driving such operations. Those processes might not be expressed apparently through their mediatised representation. Arguably, these relatively invisible operations affect how we perceive computational processes as live events. How have computational processes remained relatively invisible? How does computation perform beyond immediate reception? How might we understand generative processes in computation? The following sections take these questions seriously and regard them as necessary parameters for the discussion of liveness but they might not be the only concepts required. However, these three concepts are salient enough to establish a basis for the further examination and articulation of liveness as situated within a computational context as will be discussed in subsequent chapters.

### *2.2.1 Invisibility*

Code, according to Berry, is material and not immaterial. Although code is not a physical object and it has no physical properties such that one can touch it, code is like a “knot” which “ties together the physical” and can be touched or sensed (Berry, 2011, p. 3). Just like, for example, reading a live feed on a mobile screen, code in the form of written instructions materialises and renders data into something noticeable on a physical screen, allowing

the reader to experience the liveness of the world through the screen interface based on her reception of the perceptible content. Therefore, code is connected with other things, “mediating and constructing our media experiences in real-time as software” (Berry, 2011, p. 38).

In addition, software becomes ubiquitous as physical machines such as the personal computer or small devices like watches are now also embedded with software. Things are constantly changed and highly connected with other systems. Software studies scholars Rob Kitchin and Martin Dodge point out that code includes the invisible and distant processes of data flow across networked infrastructure and databases (2011, p. 9). Such working processes are operated invisibly, at least to general mass audiences who do not have much technical knowledge and do not have access to those machines. Code becomes visible when it generates sensual representation through data processing and manipulation, such as the LED light blinks on a motherboard or data visualised on a screen. Before code becomes a perceptible form many different events have happened and not all of them are perceptible. Perhaps most importantly, not all functions of code are rendered visible.

Chun argues that perceptible data is not a mere representation or reproduction of code insofar a computer has to “generate” such data through computation (2011b, p. 17). Chun claims that software is a metaphor that generally misleads us into believing in a separation between software and hardware, where software “is invisible yet generates visible effects” (2011b, p. 17). Specifically Chun critiques the metaphor of software that makes us believe in a synchronised logic between reading and writing and this has been observed in many interfaces. A case in point is the myth: ‘What you see is what you write.’ She goes on to argue that this logic has been implied in the general definition of software in which a task is performed and executed according to a set of instructions. In other words, for Chun, software has most often been reduced to instructions, especially instructions in source code, which are able to generate invisible data, ignoring the entire computational process. In particular, Chun addresses the execution process

which includes compilation or interpretation that is not simply a translation but involves the calculation of memory, for example, which is not stated in the source code (2011b, p. 23). As such code is invisible if one takes into account the execution process and code cannot be reduced to its association with software only.

Furthermore, Chun points out that “code does not unfold linearly” (2011b, p. 25). The nonlinear behaviour of code gives rise to its dynamic nature, as she puts it, “[code] has always been regenerative and interactive; every iteration alters its meaning” (Chun, 2011b, p. 25). Such iteration can be understood in two ways that operate at the memory level. First is the updating of the code logic, such as variables and input data, in which a value is subjected to the live conditions in which the piece of code is run. Code computes, and is based on, the values that are stored in a memory. Secondly, it is the physical marks that would be left as ‘traces’ in storage devices through data processing (Kirschenbaum, 2012). Memory is not unlimited but it is being allocated technically in a non-linear manner. Therefore, iteration is more than repeating, alluding to a process of generating differences and variation in repetition. In view of such micro-processes and detailed examination, Chun would rather say software is “invisibly visible, visibly invisible” (2011b, p. 98).

Software is a technical object that participates in complex computational processes. It is hard for anyone even computer experts, to fully comprehend and understand such complex processes. Source code that is actually written by programmers, proprietary software in particular, might only get to be read by a limited number of people and, according to sociologist and software studies scholar Adrian Mackenzie, “Code, woven into the background of transactions, habits and perceptions, does not often become visible, except in breakdowns, failures and at certain other moments” (2006, p. 170). On the contrary, some contemporary software is of a large scale in terms of its number of instructions and the number of people involved in writing it. For example, an open source operating system like Linux contains numerous pieces and lines of code that are contributed by tens of

thousands of programmers and specialists (Mackenzie, 2005, p. 72). In such cases, no single professional programmer would be able to understand the totality of functions and computer code (Bentley, 2003, p. 34; Hayles, 2006, p. 137; Mackenzie, 2006, p. 170). This highlights the fact that code is often invisible, not only to users and readers, but also to IT professionals. As computer scientist Manfred Broy puts it, “software is almost intangible, generally invisible, complex, vast and difficult to comprehend” (2002, p. 11). In short, software programs are getting more complex. The issue of invisibility may also relate to knowability and knowledgeability but not only the availability of code and the transparency of related code functions and processes.

According to Berry, code is “largely invisible” because it runs inside a confined machine and it runs fast (2011, p. 94). In most programming languages the unit of time measured by the computer are calculated in milliseconds. Nowadays, with the advancement of hardware, processing power and memory, running an algorithmic transaction takes less than a second. ‘Sub-one millisecond’ is the current standard of the stock exchange system per transaction (Keehner, 2007, n.p). This performance, as described by Sandy Frucher, the CEO of Philadelphia Stock Exchange, will continue to be reduced. This speedy processing is not only observed in machine processing but also in data transmission. The forthcoming technology Li-Fi<sup>38</sup> (light fidelity), invented by Professor Harald Haas from the University of Edinburgh, uses visible light communication (radio waves) to transmit data at a rate of approximately 10 gigabits per second, which is more than 200 times faster than the average Wi-Fi speed (Anil et al., 2015). As such, observing running code or transmitting data becomes extremely difficult for humans. Even by using sophisticated tools, it may not be accessible by everyone. Consequently, the reliance on what is visible on a screen may even dictate or direct more future study and everyday activities, reducing the awareness of what is happening between and underneath layers, processes and nodes of the plane.

---

<sup>38</sup> A term coined by Harald Haas at the TED Global conference in Edinburgh. See: [https://www.ted.com/talks/harald\\_haas\\_wireless\\_data\\_from\\_every\\_light\\_bulb](https://www.ted.com/talks/harald_haas_wireless_data_from_every_light_bulb)



This draws much attentions from academics who critically analyse this speedy phenomenon in contemporary culture (cf. Chun, 2008a, pp. 151-152). Among these scholars, Berry points out the differences between computation and human time and suggests slowing down the running time using specific devices to better observe how code runs from a distance (2011, p. 94). Berry also establishes the notion of a “grammar of code,” underpinning the different modalities of code beyond its syntactic form (2011, pp. 51-6). Code can be in different forms, including source code, prescriptive code (or executable code), commentary code and everyday code objects (Berry, 2011, pp. 51-6). The notion of code, for Berry, emphasises the performative aspect that connects it with capitalist economy (2011, p. 61) and this is especially worth noting with regard to datafication in contemporary software culture.

Invisibility is also associated with data commodification and capitalism as it is performed by code. In her recent essay, researcher Renée Ridgway discusses how customers’ data and their online behaviours have been mined and captured seamlessly for predictive measurement and profit-making by businesses, enabling new models of personalisation, profiling and customer loyalty management. Data mining businesses cannot operate without the complex logic of algorithms and hidden infrastructures (Ridgway, 2015) which, when revealed even in the abstract, cause public concern. Giant platform providers, like Facebook, Weibo and YouTube, consist of enormous numbers of subscribes or active users and these providers offer web APIs (Application Programming Interfaces) that foster data mining businesses. Arguably, those platform providers make false presentations of what their software means and how it functions and behaves through software platforms and services offered. There is not much information released to end users on how and for whom data is queried and mined. Science and technology scholar and philosopher Bruno Latour would refer to this phenomenon using his metaphor of the “black box” (1999, p. 304), a box that does not allow us to know what is inside it. He explains,

[Blackboxing] refers to the way scientific and technical work is

made *invisible* by its own success. When a machine runs efficiently, when a matter of fact is settled, one need focus only on its inputs and outputs and not on its internal complexity. Thus paradoxically, the more science and technology succeed, the more *opaque* and *obscure* they become (Latour, 1999, p. 304, *my emphasis*).

This blackboxing, in relation to software, has been critically and tactically explored in various artistic works. The project *Google will Eat itself*<sup>39</sup>(2005) criticises the neutrality of Google, the economic expansion of market control and the act of information manipulation, through hacking its internet advertising system. The way the project is presented is distinctly unlike a blackbox; the artists use a diagram (see Figure 2.3) to show the components and conceptual logics of their hidden engine, as well as demonstrating how each component relates to another and how the engine automatically triggers invalid clicks to disturb the existing mechanics of the Google advertising chain. Therefore, code is not only invisible but also largely imperceptible in terms of its complex relationship with the economy and political agenda of giant software systems like Google (Parikka, 2010, p. 118). The notion of invisibility addresses the opaqueness of computational processes that are intertwining with wider economic, political and cultural forces.

---

<sup>39</sup> *Google Will Eat Itself* is developed by UBERMORGEN, Alessandro Ludovico and Paolo Cirio. See: <http://www.gwei.org/index.php>

## Approaches to code inter-actions

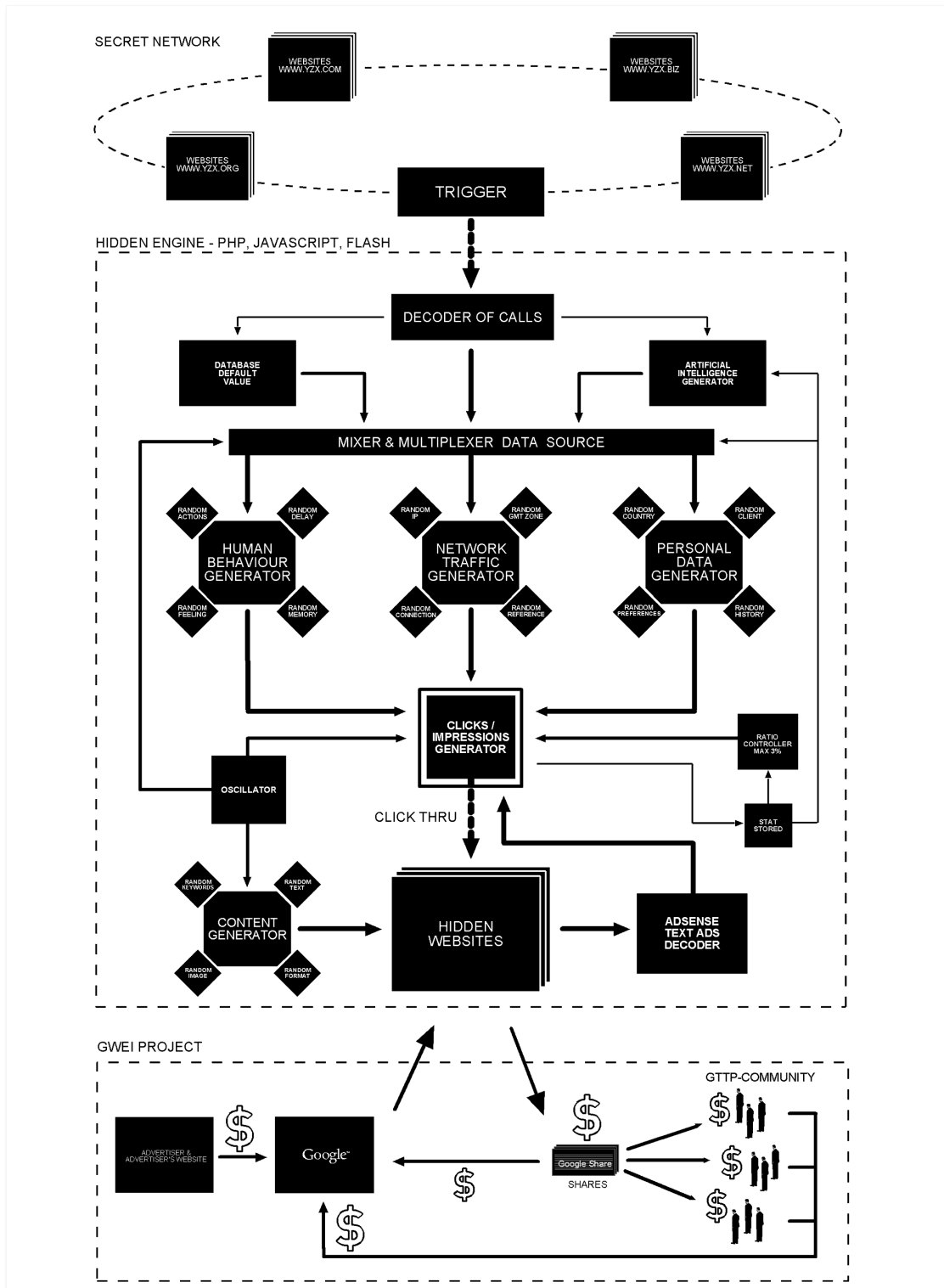


Figure 2.3: The diagram of *Google Will Eat Itself* (2005). It demonstrates the detailed mechanics of the system that is interacted with Google. Retrieved from <http://gwei.org>

### 2.2.2 Performativity

Software can be understood with reference to a particular program or platform or operating system. Fundamentally software consists of mathematical, logical and procedural instructions that are constructed in language. In programming practice instructions have to be written precisely, following certain specifications of programming languages, as code. Code is related to formal language in two ways, first the written components of programming languages and second in its implementation with symbolic controls (Cramer, 2013, p. 142). Code is a structured language, regardless of the written form of high-level or low-level programming languages. Code comprises symbols, words, grammar, syntax, statements and a strict structure that shares similar properties with formal languages. The term “performativity” is commonly used in the field of software studies to conceptualise software as language (Arns, 2004; Cox, 2013; Galloway, 2006; Hayles, 2005; Mackenzie, 2005).

Notably, code is not only expressed in its written form but is also materialised in visible and invisible interfaces and devices performing various actions and creating events. The notion of the event is specific to computation and is commonly seen in the programming paradigm in which a program or software is driven by events through Input/Output (I/O) operations such as a mouse click. This event-driven design “invokes a continuation” of a program through different event handlers (Fischer et al., 2007, p. 134). Figure 2.4 shows an example of code that draws things on a screen by using a computer mouse in which the program *listens*, as an event handler, to the possible movement and actions of a mouse input.

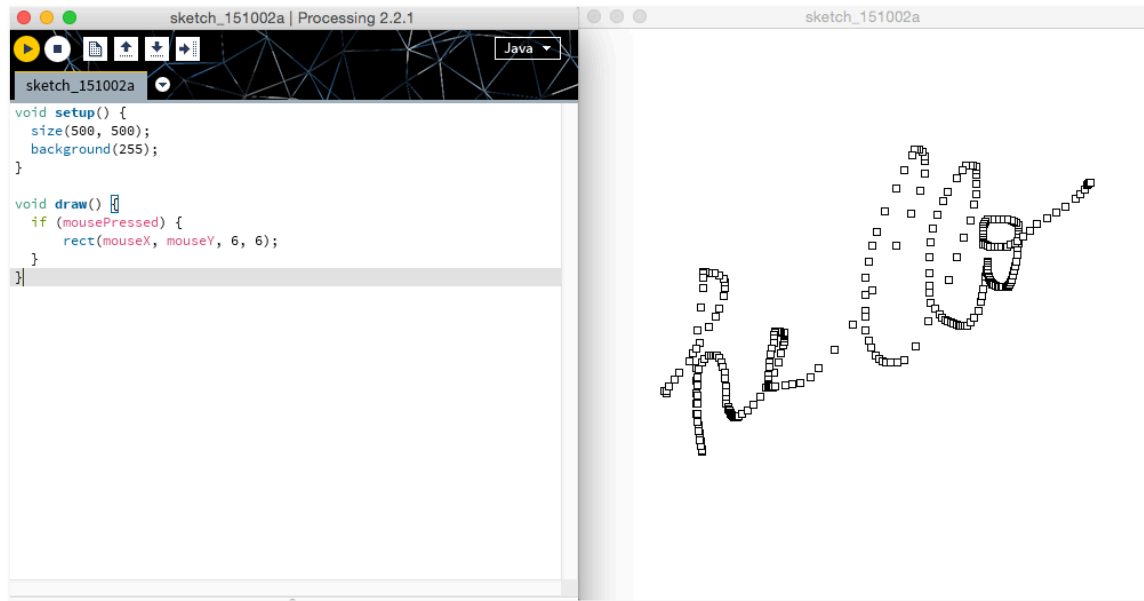


Figure 2.4: An example of code that listens to mouse events

In this example, there are several strict structures that one needs to follow, special punctuation symbols such as the semi-colon at the end of a statement as well as the pairs of parentheses and curly brackets. Additionally, the language is human readable and functions, such as the ‘draw’ and ‘setup’ commands, are case sensitive. When this piece of code runs it constantly registers to mouse events—like whether the mouse has been clicked/pressed, as well as capturing the mouse coordinate values. Once the mouse’s X and Y coordinates are captured they will be translated into a rectangular position that presents itself on a screen. In other words, running the code means performing actions. In this case, behind the visible rectangular drawing, the code registers mouse events, captures the coordinates and draws rectangles based on this information. In this example, code performs at the most basic level that output something based on the input.

Extending the basic understanding of code performativity, a subject can be addressed linguistically in both formal and computer languages. Cox argues from a linguistic perspective and points towards the ability of code that can “authoritatively *speak* to subjects” (2013, p. 3, *original emphasis*), and in drawing upon the work of Judith Butler to assert that code exists in ideology

in similar ways to natural languages. For example, Cox demonstrates this in human language, with the Althusserian phrase “Hey, you there!” (2013, p. 4). Code does not only allow one to speak to a subject but also speaks on a subject. There are many examples that could help illustrate the interconnections of language, subjectivity and code. Within the genre of codework, for instance, artists mix both computer code and text to form something like a code-poem or manifesto. Pall Thayer’s artwork *Microcodes*<sup>40</sup> (2009-) best illustrated this (see Figure 2.5). The work consists of many small pieces of microcode that are written in the programming language PERL. All of them are both readable in the form of source code and as the result of an executable program. Codework is for humans to read but at the same time can be recognised as code language, which means a piece of code or a program that demonstrates linguistic expressions yet it is not necessarily executable. According to critic and artist Alan Sondheim, codework is about “an uneasy combination of contents and structures,” but not its executability (2001, n.p). Another example of this is the software library called *Femme Disturbance Library*<sup>41</sup> (2013), which is also a “code poem” as described by the artists Zach Blas and Micha Cárdenas (2013, p. 565), designed with queer and feminist perspectives that prompts critical attention to subjectivities, genders and desires which constitute digital technologies. Such queer femme expressions are embodied and demonstrated in the artwork (see Figure 2.6). As such, this kind of codework comes with a “voice” and, according to Cox, connects “with political expression” (2013, p. 3). In other words, code is not only an expressive medium but it allows political expression that demonstrates agency and hence produces meaning through the reworking of “formal logic and poetic expression” (Cox, 2013, p. 8). Code is able to speak to, and on, certain subjects.

---

<sup>40</sup> See: <http://pallthayer.dyndns.org/microcodes/>

<sup>41</sup> Femme Disturbance Library is part of the TransCoder project. See: <http://www.queertechnologies.info/products/transcoder/>

## Approaches to code inter-actions

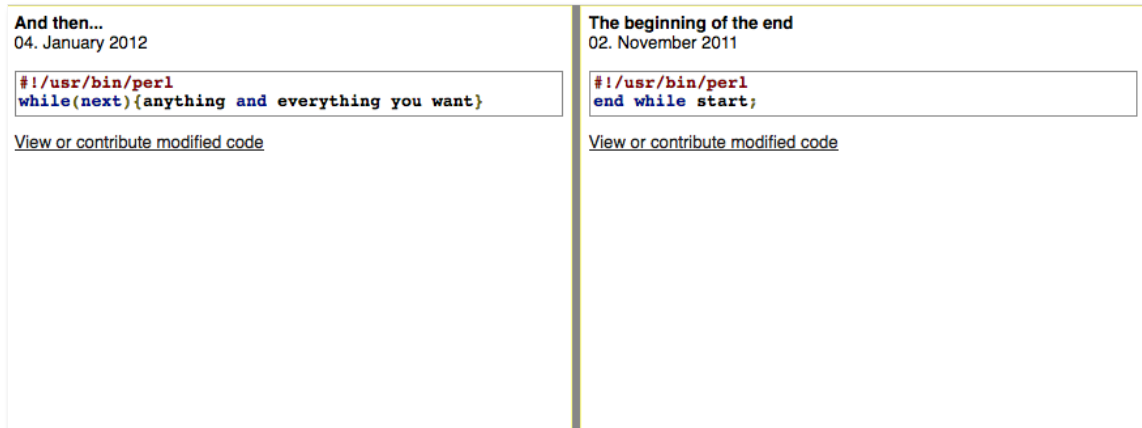


Figure 2.5: Two pieces of *Microcodes* (2009-) by Pall Thayer's. This is a screen shot from <http://pallthayer.dyndns.org/microcodes/>

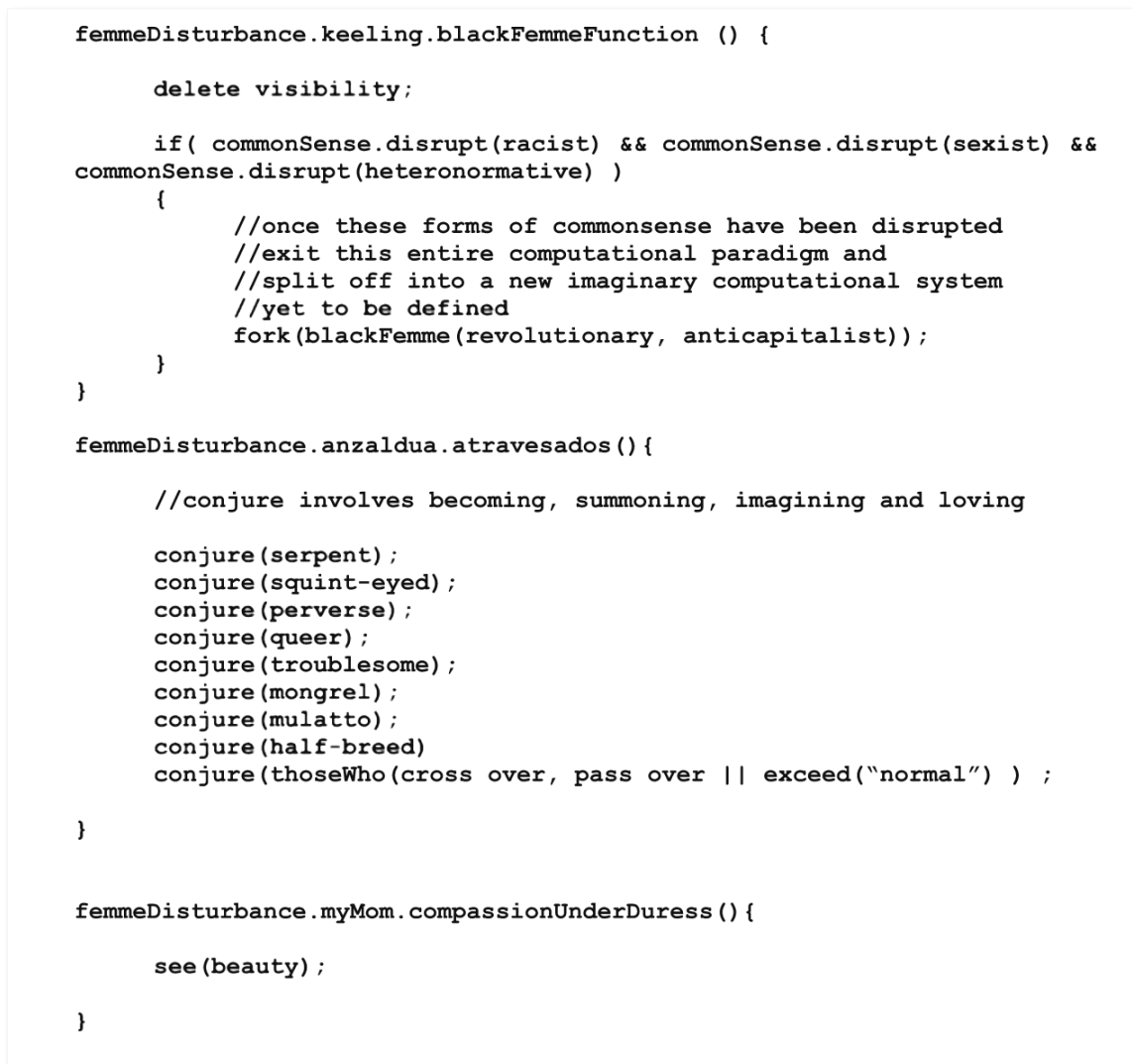


Figure 2.6: An excerpt of the work *femme Disturbance Library* (2012). Reprinted from *Imaginary computational systems*, by Z. Blas and M. Cárdenas, 2013, London: Springer. Copyright 2013 by Springer.

In the performativity of human language, actions and subjects are interwoven. John Langshaw Austin's Speech-Act theory indicates language is not only about the description of things or events but also has a performative dimension where actions are performed and subjects are involved (1962, p. 21). The analogy of speech, as used by software studies scholars, addresses the interplay between language (saying) and actions (doing) within and beyond code. Cox further draws such connection but he emphasises computer action to be unstable like speech (again drawing upon Butler). He addresses the unstable relationship in executing code, where code operations can be "out of control," especially when the computer encounters failure or program bugs. Cox argues that this instability is similar to speech inasmuch as the human interpretation of language is always transformative (2013, p. 5). In other words, the computer does not always follow strict instructions and does not produce predictable outcomes all the time. Being out of control implies unpredictable consequences. Similarly, Inke Arns suggests "[code] as an effective speech act," producing unpredictable effects which goes beyond "technical performativity." She explains, "[code] directly affects, and literally sets in motion, or even *kills*, a process" (Arns, 2004, p. 186, *original emphasis*).

In computing 'kill' is a command that is used to terminate a computer process and normally it is provided in operating systems such as Unix and Linux. In a graphical user interface operating system like Mac OS, 'force quit' is commonly used to kill a piece of software that is unresponsive. This can be thought of an act of control, regulating computational tasks. However, not all the computational tasks can be killed directly and permanently, other forms of killing take place in the form of viruses, bots and spam. As suggested by Cox, "violence is encoded in software itself," which is the same as formal languages whereby "violence is embodied in language" (2017, in press). The notion of kill is symbolic, alluding to violence and even death that raises an ethical consideration of language and its performative acts. Further extending the regulation and control within the realm of distributed networks, any system can possibly be killed because of



the vulnerabilities of computational infrastructure. In recognition of the ‘software violence’ (Cox, 2017, in press) that may impose on both humans and nonhumans, code possibly produces vital consequences and effects that are similar to the activity of human speech and act.

In fact code performs in a largely invisible or less apparent way that does not necessarily appear on a screen. Underneath the analogy of code as language, code also changes machine’s behaviours through interfacing with other technical components that should not be undermined. This technical performativity, using Arns’ word, is the underlying structure and consequences of how code runs and executes. To explicate Arns’ technical performativity, it could be said that code produces performative effects through its physical and technical typology. For example, when a machine accesses a hard disk or is navigating data archives code essentially changes computer memory, the computer motherboard’s LED and the computer performance in general. Literary theorist N. Katherine Hayles explains that code performs in the way it alters “machine behaviour and, through networked ports and other interfaces, may initiate other changes, all implemented through transmission and execution of code” (2005, p. 50). Although the human is involved in writing code instructions, once the code has started to execute and run, the machine will take over completely as “final arbiter” (Hayles, 2005, p. 50).

The distinct executable characteristic of code marks the distinction between code and language. The performative process of code does not occur in a human mind and, unlike language, its results do not take the form of human behavioural effects. Austin gives the example of saying “I do” in the context of a marriage ceremony (1962, p. 5), illustrating that such *doing* actions occur in the minds of humans. Furthermore, Galloway argues against reducing the notion of performativity to language insofar as the processes by code instructs a machine “in how to act” are distinct from formal language narration (2012, p. 71). Therefore, according to Galloway, if there is a tendency to “see code as subjectively performative or enunciation is to anthropomorphize it to project it onto the rubric of psychology rather

than to understand it through its own logic of *calculation or command*” (2012, p. 71, *original emphasis*).

Similarly, in his essay titled *The Code is not the Text (unless it is the Text)*, Literary scholar John Cayley puts forward the claim expressed in the title. Similarly to Galloway, Cayley identifies the condition of running code wherein the code is not the text. Although code is written by humans and programming languages consist of both syntactic and semantic dimensions, “there are divisions and distinctions between what the code is and does, and what the language of the interface text is and does, and so on” (Cayley, 2002, n.p). In other words, there is a performative separation between the linguistic layer and executable layer of code which is linked to its invisibility. As Cayley puts it, “[code] functions, typically, without being observed, perhaps even as a representative of secret workings, interiority, hidden process” (2002, n.p). As such, he calls for a different strategy of reading code that is not comparable to text reading, taking into account technical performativity and invisible processes. Further to reading practices, Chun specifically discusses the importance of knowing both the executable code and the process of its execution. One cannot rely solely on reading the source code as it has to be understood in conjunction with its executable form (Chun, 2008b, p. 305-6). Code and language are intrinsically related but fundamentally different. Informed by Galloway, Cayley and Chun, this thesis is also positioned to understand code through its underlying programmable logics, execution processes and operational procedures, oscillating between the visible and invisible.

Mackenzie turns his focus on code from the code itself as a language towards the processes and consequences of collective efforts in coding practice. This contributes to the on-going distribution, development and maintenance of a piece of software, such as the Linux community. Mackenzie investigates the open source operating system of Linux, demonstrating how collective agency, including developers and user communities, reconfigure “the efficacy of Linux” as a cultural artefact through its continuing advancement (2005, p. 73). The performativity of Linux then lies on the practices of code and

cultures of circulation. It “moves” (Mackenzie, 2005, pp. 76-7) the relationship of users/viewers/producers to how they use or experience code, demonstrating agency with a wider effect in social, political and cultural contexts.

Concerning the materiality and the performativity of code, Mackenzie suggests coding is only part of mediated practices. Other surrounding activities, such as those who contribute in “distributing, configuring, and running and operating system,” should be considered together (2005, pp. 76-7). Obviously this mediated practice consists of a continuation of different human and nonhuman activities that contingently shape the performance of a piece of software. He attempts to expand from the focus on the linguistic speech-act to the collective processes of circulation and all kinds of mediated practices that surround code. This performativity of code includes not only technical and production forces but also the market forces from other competitors’ proprietary operating system that shape what an operating system should be in both technical and cultural dimensions (Mackenzie, 2005, pp. 76-7). In other words, code produces performative effects through its social organisation and market forces. This performative aspect of code is also addressed throughout this thesis when it comes to examining specific code objects in subsequent chapters, without losing sight of the entanglement of the socio-economical and technical aspects of code practices.

### *2.2.3 Generativity*

This section focuses on the particular notion of generativity which is regarded as the third key concept to software studies. It is important to discuss this because generativity produces unpredictable processes and results that are regarded as one of the fundamental concepts in discussing liveness in subsequent chapters. Generativity is commonly discussed in relationship with generative art and software art (Arns, 2004, pp. 183-4; Broeckmann, 2004; Cox, 2007; 2010, p. 20; Cramer, 2003; De Souza, 2010). The similarity of the two art forms is that artwork runs a set of rules in the

form of code and this execution process generates “other forms and processes” (Cox, 2010, p. 20).

As an example, Cornelia Sollfrank produced an artwork *Net.Art Generator*<sup>42</sup> in 1997 that illustrates socio-technical processes in her work. This long-standing artwork is an endless generative machine that uses Google image data bank and customised online software, constantly generating images as a piece of net art through a user’s keyword search (see Figure 2.7). Therefore, Sollfrank claims, “Anyone can become a (net) artist” (2003, n.p), insofar as the machine helps generate an art image. The resulting images are technically generated by a program that includes “a random-generator driven collage technique” (Sollfrank, 2003, n.p). This points directly to the problem of authorship, challenging the traditional understanding of an artist as the sole author. Indeed, for Sollfrank, the notion of generativity takes into account the social perspective: the collective efforts in coding practices (similar to the discussion of Mackenzie’s notion of performativity that was addressed earlier). According to Sollfrank, the source code of *Net.Art Generator* is open, alluding to the fact that anyone can freely modify and distribute the code (2003, n.p). This open source approach enables an endless reproduction and regeneration of code through source code re-modification in a social dimension. Art critic and curator Andreas Broeckmann refers to the term generativity as a “means for the creation of machinic and social processes” (2004, n.p).

---

<sup>42</sup> *Net.Art Generator* (1997) by Cornelia Sollfrank: <http://nag.iap.de/>

## Approaches to code inter-actions

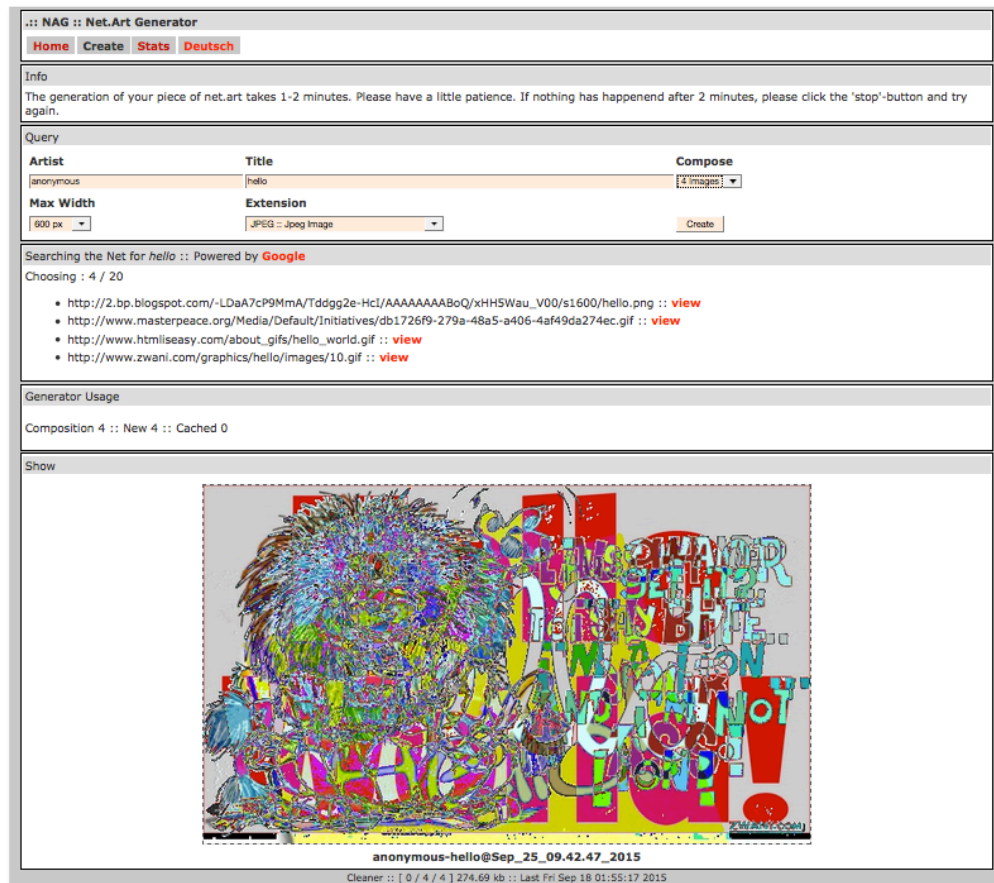


Figure 2.7: A screen shot of the work *Net.Art Generator* (1997) by Cornelia Sollfrank. An image is generated based on the keyword “hello” and the selection and combination of four Google images that I chose on 25 September 2015. Retrieved from <http://nag.iap.de/?ac=create&name=anonymous&query=hello&comp=4&width=600&ext=jpg>

Nevertheless, in many situations the term generativity has been used mainly to place focus on syntax (Cox, 2007; Cramer, 2003), mathematics and science (De Souza, 2010) in comparison with the more open agenda of software art. Artist-academic Philip Galanter who publishes substantial articles in the area of generative art, has explicitly claimed that the term “denotes art created by non-human systems” (2016, p. 151). This implies that the notion of generativity should be primarily focused on systems. Such a perspective is indeed important to understand how a system functions and how rules are structured in order that further examination can be built upon it.

Many artist-academics show that generative art concerns how the works are made (Galanter, 2016, p. 154; Watz, 2010), examining this issue from the

perspective of artistic practice. This perspective focuses on a system in which it takes control in creating artworks. According to Galanter, “generative art systems are autonomous,” alluding to a system which does not require decision making or control by a human during the process of running or execution (2016, p. 152). In 2003 Galanter defined generative art as follows:

Generative art refers to any art practice where artists use a system, such as a set of natural languages, rules, a computer program, a machine, or other procedural invention, which is set into motion with some degree of autonomy contributing to or resulting in a completed work of art (2003, n.p).

Galanter places emphasis on the machine’s conditions of production and the results which exhibit some degree of autonomy. Generative artworks “must be well defined and self-contained enough to operate autonomously” (Galanter, 2003, n.p). The use of the term ‘self’ does not point at any programmer/artist but to the core of the self-organising processes of a machine. Those rules include mathematical formulas and logical procedures that have to be well written in advance and have to conform to the programming requirements. Therefore, the self-organising processes of code instructions and procedures, as describe by Arns, “are running independently from their authors or artist-programmers” (2004 #779, p. 178). This is evident in *Net.Art Generator* too as Sollfrank explains,

it is no longer the human creator, who arranges the single parts in order to create new meaning through the relationships constructed, instead, a machine, the computer program, takes on the role of the artist (2012, p. 40).

Code is autonomously run in a way that it is able to self organised and self processed independently. This distinguishes it from other human-machine performance, such as live coding, where there is the involvement of human interaction and feedback ‘on-the-fly.’ The production process of generative

art is “unsupervised” once code has started to run (GENERATOR, 2002). Galanter further addresses the notion of control in his second definitive edition, shifting control from humans to nonhuman organisation and operation. He offers the following definition:

Generative art refers to any art practice in which the artist cedes control to a system with functional autonomy that contributes to, or results in, a completed work of art. Systems may include natural language instructions, biological or chemical processes, computer programs, machines, self-organizing materials mathematical operations, and other procedural inventions (Galanter, 2008, p. 154).

Indeed, the concept of generativity is central to the field of artificial life (A-Life) which is also about examining systems—specifically natural systems—through the use of simulation technology (Bedau, 2003, p. 505). One of A-Life’s characteristics is the generative and emergent qualities of simulating nature which are also largely based on the self organisation of computation and autonomous agents (Penny, 2009). The notion of emergence in A-Life is about how agents *learn* through feedback and become smarter as part of an adaptive system and this requires the understanding of a complex system. In his book titled *Emergence*, Steven Johnson explains the adaptive behaviour of agents. These intelligent agents grow smarter through “bottom-up systems, not top-down,” which have been seen in the case of ants, cities and pattern-recognition software (2001, p. 18). This kind of complex system does not plan its course in advance but evolves through process.

The notion of generativity is frequently referenced in complexity science to explain the dynamism of systems in different kinds of generative arts (Galanter, 2003, 2008, 2016; Hayles, 1990, 1991; Solaas et al., 2010). Galanter points out that a “[c]omplex system often includes chaotic behaviour” (2003, n.p), in which the random function is a commonly used parameter to introduce dynamism in a nonlinear system. Even a small

change in a seemingly ordered system can generate large differences, resulting in a chaotic system that is “increasingly difficult to predict over time” (Galanter, 2003, n.p). This is commonly referred to as the “butterfly effect,” a term coined and developed by a mathematician Edward Norton Lorenz in 1969. Complex systems include multiple to infinite components and each component interacts with each other. The level of complexity increases with the number of components. Galanter gives the examples of weather and stock markets to demonstrate different systems that lead to unpredictable outcomes and behaviours (2003). A generative program may consist of many combinations of different rules and conditions which emerge autonomously. This is how Galanter refers to the notion of generativity in which control is not solely based on pre-written rules or the author/artist. The artist cedes control to a system and allows the work to operate autonomously.

Galanter considers both chaotic behaviour and ordered interactions. The adoption of the term ‘effective complexity’ serves to emphasis the fact that a complex system contains “a mixture of both order and disorder” (Galanter, 2016, p. 157). Drawing upon physicist Murray Gell-Mann’s effective complexity theory, Galanter treats complex systems as living things, including computational systems. He explains that, “life requires both order maintaining integrity and persistence, and disorder allowing adaptation, change, and flexibility” (Galanter, 2010, p. 402). Importantly, effective complexity “comes with the balance of order and disorder, or expectation and surprise, built in” (Galanter, 2010, p. 402).

Unpredictability is one of the important aspects of generative art that is produced by a system. As an example, Generator.x<sup>43</sup> is a curatorial platform that embraces using code as artistic material and creative expression. In 2008, it collaborated with Club Transmediale and presented 15 artworks that had utilised generative systems and custom software. The works were presented under the festival theme of “Unpredictability,” investigating

---

<sup>43</sup>See: <http://www.generatorx.no/>



“artistic concepts that imply the surprising and unforeseeable, accidents, mistakes and coincidences as a means to alter the dynamics of creative processes and to discover new aesthetic forms” (Generator.x, 2008, n.p). The festival provides a concrete articulation of what unpredictable aspects might be generated. Technically, implementing random functions is seemingly one of the ways to increase the dynamism of a system, creating unpredictable effects (Schönlieb & Schubert, 2013, p. 8; Watz, 2008). However, it is important to note that in computational systems, the use of the random function does not stand for true randomness; it can never perfectly simulate natural randomness. The word or the function random implies ‘pseudorandom’ instead (Montfort, 2013; Tian & Benkrid, 2009).

Simulating real world systems is one of the applications of generative art and, as Galanter says, “artists can create form that emerges as result of naturally occurring processes beyond the influence of culture and man” (2003, n.p). The word ‘natural’ could be further extended to emergent “naturally evolving phenomena” in the real world, such as the organic process of the growth of a tree (Pearson, 2011, p. viii). A common focus of generative art is its alleged “natural” and “organic” forms which simulates real world systems and produces an “aesthetically pleasing” result (Pearson, 2011, p. xix). In other words, generativity is thought of, non-ideologically, as a ‘neutral’ tool or strategy to achieve an organic form as an end product; it is about the perusal of techniques and the production of a final form. Therefore, the notion of generativity in generative art is of less interest in questioning the neutrality of computational processes when compared with software art, or simply using software as a pragmatic tool to generate a close-to-real phenomenon.

Such interest in autonomous operations, emerging processes and simulated results maybe what Arns would describe as “the negation of intentionality” (2004, p. 178). However, artist and scholar Mitchell Whitelaw argues that a generative system pays close attention to “entities and relations within system, with entities and relations outside it.” This ability to connect things requires critical, prospective and speculative considerations (Whitelaw,

2006, p. 140). Cramer and Gabriel also point out that a generative system is “not as negation of intentionality but as balancing of randomness and control” (2001, n.p). In the context of generative art, it is the balance between the randomness and structure of a visual representation that counts for a good visual effect after all (Olthof, 2009, p. 7). In general, Cox observes that the notion of generativity is not specifically limited to art but such an understanding may lock into the emphasis on end-result that is produced by a generative process rather than keeping the focus of the process itself (2010, pp. 21-2). Code therefore might risk being considered as performing a supporting role in generative art to help generate a visually pleasing artwork but the code itself is inscribed with its own structure and format, decision making and subjectivity. One of the interests of software studies is to question the seemingly neutral commands, formats and actions that are performed by code. Cox instead suggests examining software culture in relation to generative processes—together rather than in separation (2010, p. 24). This concern on generative processes also provides a conceptual approach for how this thesis will unfold in later chapters.

In part, these three concepts—invisibility, performativity and generativity—inform the understanding of the materiality of code, in which code has to be read as connections with computational processes, such as generative and execution processes. The perspective on liveness adopted by this thesis is supported by how scholars in the field of software studies emphasise processes rather than surfaces, results or mediated representation. As such, computational processes such as data processing, code execution, algorithmic procedures and network handshaking are crucial when discussing the phenomena of software that emphasis the processual quality.

Although these three concepts are fundamental there are still situations which they seemingly cannot fully explain. For example, what makes a program different when run at one time than another? What are the implications of the automated and seamless updating of software features, standards and specifications? Increasingly, with different network computers that share data content in real-time, how does code operate

within such a distributed and live environment? Clearly, these three concepts undermine the importance of live conditions in which networked technology plays a significant role in shaping the culture and practice of software. In relation to this thesis, liveness is not entirely separated from the concepts of invisibility, performativity and generativity that have been illustrated above. These concepts set the stage for understanding current debate in the field of software studies, allowing the notion of liveness to be brought into the field. The notion of liveness is based on such concepts to address the opaqueness of code and the performative effects of operational logics as well as the complexity of computational systems and is further extended to consider the live dimension of code inter-actions in subsequent chapters.

## **2.3 Materialist approach**

In the following section a materialist approach is introduced as a conceptual perspective through which to examine the notion of liveness in the entire thesis. Conceptually, it is a way of thinking about matter and processes of materialisation that constitute the understanding of liveness within the context of software studies. The underlying assumption is that contemporary software culture is composed of material elements and processes as a plane of immanence. Things that we experience through networked devices are generated by complex structures and processes that are not apparently visible. Media studies scholar Nathalie Casemajor gives a list of objects of study that are considered by materialist approaches. She states:

[Materialist approaches] embrace both the material substrates and abstract programming languages required for data storage, processing and exchange; code, hardware devices, operating systems, software, applications, platforms, interfaces, documents, file formats as well as networking protocols and infrastructure (Casemajor, 2015, p. 6).

As informed by the nonhuman turn that has been discussed in the previous chapter, those suggested objects do not act alone. Similar to the manner in which code never executes on its own, executing computer instructions involves other things such as a computer processor, a compiler, computer memory, electricity, programming language, human and nonhuman logics and designs and so forth. All these individual objects play a role and collectively “emerge” and “make something happen.” Materialist theorist Jane Bennett calls this as “the agency of assemblages” (2010, p. 24).

A materialist perspective pays attention to the underlying assemblages, or “material substrates” (Galloway, 2004), which have been increasingly addressed in the field of software studies or in fields related to it. For example, literary studies scholar Matthew Kirschenbaum investigates textual production processes which leave traces on storage devices. Delving deeply into the physical properties and operations of hard drives, Kirschenbaum argues that these storage media are considered “as a kind of writing machine” that inscribes traces (2012, p. 19). Although he mainly focuses on textual materiality, his notion of “forensic materiality” suggests any kind of digital media production is fundamentally taking place in the physical and material world (Kirschenbaum, 2012, pp. 11-2). He argues that materiality is essential and fundamental to the operation, process and understanding of digital media.

Extending from general digital media production to a more specific sphere, Fuller analyses the infrastructure of the web through his own collaborative artwork *The Web Stalker*<sup>44</sup> (1997). “By material is meant the propensities of the various languages, protocols, and datatypes of the web.” (Fuller, 2003, p. 53). This material constitutes the actualisation of a browser, forming visual outcomes for reception. What matters Fuller is not the representational content however, instead his work reflects on those deep structures of the web in order to think about everyday web culture: the production of web interfaces. In a similar vein, Galloway’s work explores the relationship

---

<sup>44</sup> See: [http://www.archimuse.com/mw98/beyondinterface/fuller\\_fr.html](http://www.archimuse.com/mw98/beyondinterface/fuller_fr.html)

between networked computing and the political economy through a close examination of networked materials: the command and control of protocols, namely TCP/IP and DNS. He makes apparent that such material forces are enacted through the process of materialisation (2004, p. 110). Both Fuller and Galloway critically examine the forces of commands, languages, protocols and datatypes that are in the form of code, highlighting the processes of actualisation and materialisation that produce agency.

In his book *Cutting Code*, Mackenzie also focuses on the relationship between code, forces and agency. He delves into different systems, namely Java Virtual Machine and the Linux Kernel, exploring the social relations of software, including “different practices of production, consumption, use, circulation and identity” (Mackenzie, 2006, p. 2). Mackenzie asserts, “code as a material” with agency, where power, law and art are associated with software (2006, p. 12). Therefore, on the one hand code can be seen as a technical object, and on the other code can be perceived as material. As such, code materiality is about agency—social, cultural and/or political forces (also described as material forces) that surround and shape what code does, and how code becomes what it is.

From a materialist perspective materiality is more than the study of mere technical objects and processes. According to scholars in political science, Diana Coole and Samantha Frost, “materiality is always something more than *mere* matter: an excess, force, vitality, relationally, or difference that renders matter active, self-creative, productive, unpredictable” (2010, p. 9). This encapsulates the study of objects and processes that are dynamic in nature, referring to matter as forces that are constantly and contingently shaping the encountering of things. This thesis emphasises nonhuman materials and their agency, but this does not mean that it neglects human participants. Numerous humanistic research approaches towards technology such as audience analysis, phenomenology and cognitive feedback (which has had a profound impact on the study of humanities) are clearly also relevant. To clarify, whilst recognising the collective of humans and nonhumans that both play an active role in the making of meaning, this

thesis emphasises and reiterates the nonhuman aspects of this process. This perspective is also aligned with the emerging field of software studies in placing various digital objects and their efficacies, potentialities and systematic forces at the centre of the argument in order to better understand the processes of computation and its materialisation. This is what Bennett would describe as vitality, “the capacity of thing,” in which things could also “act as quasi agents or forces with trajectories, propensities, or tendencies of their own (2010, p. viii).

### 2.3.1 *Why code inter-actions?*

Following the focus on code and agency, this section introduces the notion of *code inter-actions*, highlighting the relations of things in which code does not act alone. This can be understood through a thinking model which contains three large material arrangements: code, data and technological networks (see Figure 2.8). They are not independently examined but constantly *inter-acting* with each other. Technically, code is used in its broad sense, including source code, executable code, scripts and programming languages, comprising syntax, functions, commands, algorithms, mathematics and calculations. For technological networks, it includes protocols and mechanisms of data transmission. Data requires to distribute over a technological network. Data is equally important as code and technological networks and is associated with memory, storage, feeds, structure and streams that can be processed and transmitted from one location to another. The transmission of data is not only concerned with the macroscopic dimension (from one machine to another) but also from one node to another, as well as data that is passing within a code function at a microscopic level. Together with code, data and technological networks, this thesis brings together other related fields such as platform and information studies to examine the underlying complex computational processes and their inter-actions.

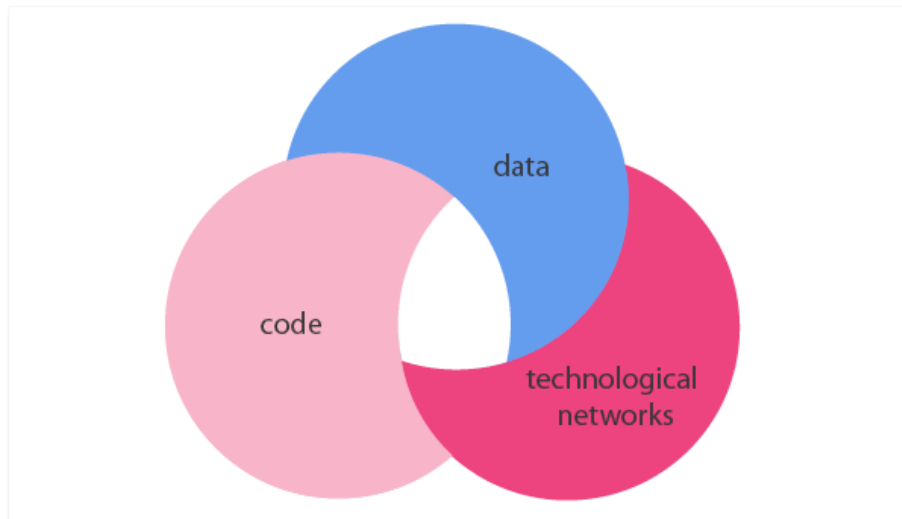


Figure 2.8: A thinking model of code inter-actions

To understand contemporary software culture that concerns materiality and agency, developing a critical perspective towards code is seemingly essential. The book *Interface Criticism* (2011) attempts to present a theoretical framework, pointing to “the working in, between, behind and beyond the interface” of software culture (Andersen & Pold, 2011, p. 10). The term interface has been increasingly used in computation, such as human computer interface, application programming interface, graphical user interface and physical interface and among others.<sup>45</sup> It is even sometimes used interchangeably with the term code because they are both concerned with signs and signals, humans and machines. However, the term interface is heavily associated with the larger community of human-computer interactions, which centres on the relationship between human and computer (Andersen & Pold, 2011, p. 19; Emerson, 2014, p. 133; Hookway, 2014).

Fuller, on the other hand, develops a model to illustrate different types of software, namely critical software, social software and speculative software. Critical software pays attention to the inner production and operational process in order to inform “knowing, seeing and doing” of software (Fuller, 2003, p. 23). Social software highlights “an ongoing sociability between users

---

<sup>45</sup> See the glossary Interface in the book *Software Studies* (Cramer & Fuller, 2008) and the interface layer chapter in the book *The stack* (Bratton, 2016, pp. 219-50).

and programmers” around coding practices. Therefore, software can be seen as socio-technical through the model of social software (this has been also demonstrated in Mackenzie’s investigation of the open source software Linux). Whilst speculative software is focused on the potential of programming which intersects with “data, machines, and networks” in which new creation is made (Fuller, 2003, pp. 29-30). Manovich uses the term software in his book, *Software Takes Command* (2013), for him software (to be precise he uses the term media software) points towards the cultural aspect of an official market related application or an artefact that is associated with practice or applied usage as such.

Similar to Fuller’s notion of Critical Software, Chun also place emphasis on the inner computational processes of software. Instead of using the term software, Chun considers the difference between software and code and takes into account the executability of code—the conflation between source code and compiled code. She firmly asserts that, “software is code” (Chun, 2008b, p. 309; 2011b, p. 27), explaining that source code does not fully indicate what it does, inasmuch as the procedural logics mostly express the intention of a programmer. It does not reveal how the code is processed. For example, the complexity of such execution processes, as Chun explains, is not a translation from “a decimal number into a binary one, rather it involves instruction explosion and the translation of symbolic into real [memory] addresses” that requires “arithmetic calculation” (2008b, pp. 306-7). Ultimately, code is “based on a conflation of storage with access, of memory with storage, of word with action” (Chun, 2008a, p. 160). The notion of code inter-actions specifically references Chun’s notion of code and Fuller’s concept of critical software which take seriously the inner writing and execution of code and its’ underlying operative processes that inter-act with other things. This thesis focuses on code that operates in networked environments to address various phenomena in contemporary culture. Therefore, code not only operates within the machine on which it runs (conflated with memory and storage) but also constantly inter-acts with networked data and network protocols.



In *There is no Software*, literary scholar and media theorist Friedrich Kittler observes that at a superficial level software deliberately presents a masked interface, hiding “the very act of writing” (1995). This can be understood in two ways: First software as a tool changes the relationship of the act of writing; Second, software is a blackbox in which users tend to only be concerned with usability and functioning, hiding the real logic, imperceptible functions and invisible machine’s operations from users. Kittler calls for attention to code operations and material substrates: the fundamental writing processes of code beyond an end product (artefact) as a piece of software. Code is constantly inter-acting in all layers within a machine. Informed by Chun, Fuller and Kittler, the thinking regarding code inter-actions in this thesis is, therefore, further grounded. Indeed, both Kittler and Chun have criticised the distinction/separation between hardware and software. Kittler argues that the abstraction of software essentially hides the nature of machine and computer system. He reminds us that there are different operating layers in a machine such as the machine code that communicates within the machine, the software which runs on top of BIOS and the data processed from external mass memory to the random access space of a machine (Kittler, 1995). Whilst Chun asserts that, “software has become a metaphor” which makes us believe things are separated. Remarkably, the notion of the metaphorical software has made us believe that what is stated in the source code is what should be translated and reflected on the surface/representation. Thus software is mostly reduced to the perception of visible effects that are generated by invisible software (Chun, 2011b, pp 2-3). The use of code in favour of software in this thesis is to minimise the misconception of such a dichotomy with hardware. Additionally, following Kittler and Chun, the notion of code inter-actions does not only refer to a machine itself but to networks of machines, in which code executes and inter-acts at multiple scales.

### 2.3.2 *Live inter-actions*

The interaction and operation of code, which I refer to as code inter-actions,

is different from the focus on human/machine interaction. Although cybernetic theory and the feedback loops of users' interaction are commonly used to explain human/machine relations in the realm of interactive/responsive art and live coding performances (Andersen & Pold, 2011; Ascott, 1966, 1967; Goodman, 1987; McLean, 2011), this human-machine focus is less helpful for this thesis, in which nonhuman interactions are more of a focus.

The multiplicity of interactions is explained through the work of Peter Bentley in understanding interaction from a computer science perspective. He says, "There are so many separate elements (subroutines, modules, files, variables) and they interact with each other in so many ways" (Bentley, 2003, n.p). This implies that code does not work within singularity; code encompasses subroutines that enable modularity and these subroutines might possibly locate in other modules and file systems which require the running of code to bring all materials together. For Bentley, simply passing data through variables, within and across subroutines, is already regarded as one form of interactions and hence code interacts within itself and other material objects.

In a similar vein in relation to the discussion of code interaction, Cox, Alex McLean and Adrian Ward further address the "live running" environment of a system as follows:

The code is interacting with the user, itself, its environment, and the systems it has access to via many multi-layered and mediated interfaces that are available to it. Many of the components are predetermined, but through the combinations of interactions combined with the dynamism and unpredictability of *live action*, the result is far from fixed as a whole (2004, p. 164, my emphasis).

What they have shown is that code not only interacts within itself and with others but, more importantly, dynamic and unpredictable forces are coupled

through running code in a live environment. Once the code starts running and brings things into being the whole situation changes from determinate to indeterminate interactions. Likewise, independent researcher Michael Murtaugh, who works in the area of software studies, addresses the notion of liveness more explicitly. The sense of “infinite database,” as he argues, is the result of noncomputability and liveness of a system, in which computational decisions are made in real-time. Furthermore, if computation is introduced in a more noisy channel, then computer operations would introduce “a greater degree of uncertainty” (Murtaugh, 2008, pp. 145-6). Informed by this, the historical analysis of the Turing machine may need to be expanded to reconsider various elements of code running as it inter-acts with the outside world. Computer scientist Peter Wegner critiques the classic Turing machine as failing to account for wider assemblages. He explains, “Turing machines cannot, however, accept external input while they compute; they shut out the external world and are therefore unable to model the passage of external time” (Wegner, 1997, p. 83). Having the capability to inter-act with a dynamic environment is a requirement of what Wegner refers to as an “interaction machine” (1997, p. 83). Computer scientist Michel Beaudouin-Lafon further picks up on Wegner’s discussion by pointing at contemporary conditions of “endless streams” and “distributed systems” (2008, pp. 263-4) through interacting entities which allow unpredictable events to occur in a dynamic environment that cannot be reduced to the credit of an algorithm per se. He notes,

distributed system are now ubiquitous, from the Internet to computer clusters and multicore chips. Such large and complex systems can no longer be analysed as a single algorithm but must be seen as a set of interacting entities (Beaudouin-Lafon, 2008, pp. 263-4).

The notion of interaction above suggests some characteristics of *live* conditions including infinite data generations, distributed networks, dynamic systems and unpredictable results, which emerge through inter-actions of code. Liveness is not an end result of such inter-actions but it

needs to be understood as on-going processes of materialisation. In fact, the whole notion of code inter-actions not only accounts for the perspective of computer science in understanding nature but also for the concept of agency in materialism.

Central to the notion of liveness, the concept of code inter-actions is in part influenced by feminist theorist Karen Barad and her ontological notion of “intra-actions” that highlights agency (2003, 2007). As she says,

The neologism “intra-action” *signifies the mutual constitution of entangled agencies*. That is, in contrast to the usual “interaction,” which assumes that there are separate individual agencies that precede their interaction, the notion of intra-action recognizes that distinct agencies do not precede, but rather emerge through, their intra-action (Barad, 2007, p. 33, *original emphasis*).

Barad emphasises emergence in the intra-action of things. She describes the metaphysics of things as phenomena and they “are the ontological inseparability of agentially intra-acting components” (Barad, 2007, p. 148). She might describe liveness as a phenomena, inasmuch as there are no individually constituted agents or pre-existing entities but rather it needs to be understood as ‘entanglement.’ Ontologically, Barad refers to the entanglement of material relations that are not only technically and scientifically specific but also involve mixed factors and domains of operation that are regarded as social, political, economical and cultural (2007, pp. 232-3). These material relations, both discursive and non discursive, produce agency but agency, according to Barad, “is not an attribute but the ongoing reconfiguration of the world” (2007, p. 141). She underlines the idea that the entanglement of material relations is dynamic in nature. The notion of liveness is about an attentiveness to the nature of entanglement that constitutes how we understand the live dynamic of computational processes.

In this thesis, the entanglement of code is expressed through the term *code inter-action*, alluding to the complexity of material relations and their agencies which is not only limited to code per se. Code may be regarded as a separate individual entity, a digital object and a material but it could be further understood as a co-constituted entity, object and material which is continuously and contingently emerging in, and through, their mutual interdependence as part of the phenomena itself. This thesis chooses to primarily focus on the inter-actions and material relations of code, data and technological networks.

In other words, individuals as things/entities/objects “emerge through and as part of their entangled intra-relating” (Barad, 2007, p. ix). Informed by this, we can say it is neither code nor protocols nor databases nor technological networks that inherent the phenomena of liveness and, according to Barad, it is “a matter of intra-acting; it is an enactment, not something that someone or something has” (2007, pp. 232-3, p. 178). Ontologically, she suggests that intra-actions “are not mere static arrangements of the world,” and the world as “agential intra-activity in its becoming” (Barad, 2007, p. 141). Considering liveness as phenomena, an ongoing process of materialisation, this thesis explicates the three vectors through the agential possibilities of code. Code inter-actions emphasises entangled, active, dynamic and collective relations: the “agential intra-activity” that constitutes the phenomena of liveness. Apart from the understanding of interaction from a scientific paradigm, the use of the term *inter-action*, throughout the thesis, also makes reference to Barad’s notion of “intra-actions,” highlighting the dynamics of entanglement and signifying the notion of liveness as “things-in-phenomena” (2007, p. 140).

## 2.4 Methodological considerations

Following the discussion of the conceptual understanding of materiality, this section shows three overarching methods, examining how code inter-acts and performs in the world. These methods are central to the methodology use in the research documented in this thesis. They are ‘close readings’ of

code from critical code studies (Marino, 2006, 2014), ‘iterative trials’ of running code from software studies (Berry, 2011, 2014) and ‘cold gazing’ of micro-processes from media archaeology (Ernst, 2006, 2013b). The objective has been to combine these methods in order to examine code beyond treating it as a text for mere reading and writing but rather to consider the code running and execution of code and its operative processing.

### *2.4.1 Close reading in Critical Code Studies*

Critical code studies (CCS) promotes the examination of written code, analysing digital objects through writing and reading (Marino, 2014). This way of working has gained increasing currency in the field of software studies through conferences, presentations, discussions on blogs, workshops and published books (Marino, 2014). The CCS community was initiated in America with core contributing members that include Nick Montfort, John Cayley, Rita Raley and Mark Marino, who are primarily interested in (electronic) literature, especially on how the study of code informs and changes the practice of writing and reading. CCS pays attention to code as textual materials and its main argument is that code itself can be considered to be a “cultural text worthy of analysis and rich with possibilities for interpretation” (Marino, 2006). This hermeneutic interpretation enables the discussion of code beyond the domain of computer science, software engineering and the arts. It demonstrates the interdisciplinary ambition to translate “humanities hermeneutics into the conceptual paradigms of computer sciences” (The Humanities and Critical Code Studies Lab, n.d).

Beyond paying attention to efficiency and aesthetics, close analyses of code has a broader scope to CCS, including social and cultural interpretation through which to facilitate a wider discussion on the significance of code (Marino, 2006). As Marino describes it, code allows one to reflect “on the relations between the code itself, the coding architecture, the functioning of the code, and specific programming choices or expressions, to that which it

acts upon, outputs, processes, and represents” (2006, n.p). One can read and interpret specific lines of code even though those lines may never be executed (such as the comments line).

The usual way to interpret code is to select a particular block from an entire program, close reading each line of the code “to build a structure that resonates and operates aesthetically, functionally, and even conceptually with the other discourse of encoded objects as well as mathematical and natural language discourse” (Marino, 2006, n.p). The phrase ‘close reading’ resonates with the literary approach of textual reading as discussed in many scholarly works in the area of digital text studies (Hayles, 2010; Raley, 2012; Simanowski, 2008). According to Marino, there are “symbols,” “procedures,” “structures” and “gestures” that exist in code (2006) and these characteristics allow computer code to be read in a similar way to literary works. Maurice Joseph Black, in his doctoral dissertation, suggests code is similar to poetry too, “in terms of structure, elegance, and formal unity” (2002, p. 131). Additionally, some scholars in CCS suggest code could be analysed as static material like a printed poem, which could be non-functioning (such as codeworks as mentioned earlier in the chapter). Therefore the issue of whether code can be executed is not a primary concern (Cramer, 2001, 2005; Marino, 2006).

Nevertheless, there are some scholars who argue that reading code is not the same as reading text and claim that such readings must take into consideration the specificity of code execution. For example, although Cox, McLean and Ward also suggest code is somewhat like poetry in terms of its aesthetic value, the focus on code goes beyond its written form, as they argue, “the aesthetics value of code lies in its execution, not simply its written form” (2000, n.p). Similarly, Cayley draws out a distinctive characteristic of code: the difference between what code does and what code is about. Even though code and language are related, code is also distinctive from other pieces of text because of what it is and does (Cayley, 2002, n.p). For Cayley, the use of hermeneutic interpretation “simplifies the intrinsically complex address of writing in programmable media” (2002,

n.p). What he refers to as “writing” includes texts that are directly addressed to the machine such as instructional code: the compiling process of execution. He, therefore, calls for a different strategy for reading program code as opposed to other formal languages. Although Marino does not want to limit CCS to literary study, the focus on reading code in a humanistic tradition still mostly remains the main method for analysing what has been written in code. CCS focuses on how code communicates with humans and asks how code expresses itself and generates meaning from the written form of source code.

### *2.4.2 Iterative trials in Software Studies*

What is largely missing from CCS is the dynamic and live aspect of code. Berry’s method of ‘iterative trials’ (2011, 2014) explicates another important dimension: the running of code. He notes:

Code is understood not merely through a close reading of the text, but by running it, observing its operation and processes it institutes, introducing breakpoints and ‘print to screen’ functions to see inside the code while it is running (Berry, 2014, p. 191).

Specifically, Berry suggests ‘trials of strength’ (2011, p. 67) as a way of observing code running processes. In the trials approach (a process of trial-and-error testing) each iteration has additional features and fixes (also known as patches, sprints, release candidates and beta in software development and design) before a final release version comes out to the market. Each release contains embedded business requirements and priorities, programmers’ decisions and their fine-tuning of logics, bug fixes, and many others factors. Therefore, these iterations demonstrate what Berry describes as the “real-requirement” (2011, p. 67), encompassing priorities and decisions from different parties that are changing across time.



In Berry's book, *Critical theory and the digital* (2014), he further explores the notion of testing and takes into consideration what he calls "implicit temporality and goal orientedness" to develop "coping tests" (2014, p. 185). This coping test not only focuses on the static test but what he means by "implicit temporality" refers to the code structure "that has a past, a processing present and a future orientation to the completion of computational task" (Berry, 2014, pp. 185-6). Such a focus is neither mechanical, mathematical nor textually oriented but is a complementary critical approach that "engage[s] with the processual nature of algorithms" (Berry, 2014, pp. 185-6). Every test run or version released is important in understanding the processual nature of software as it often generates glitches and errors. The implication of coping tests, according to Berry, is that we can learn from software through "breaking it, glitching it, hacking it and generally crashing its operations" (2014, p. 186). As such, his approach concerns the operational process of running code and, therefore, the reading of source code is only a part of the process (Berry, 2014, p. 190).

More specifically, Chun suggests thinking about the difference between source code and its execution, as what the code produces is never a stable artefact and often includes failure and disruption. She introduces the idea of code as "re-source [that] allows us to take seriously the entropy, noise, and decay that code as source renders invisible" (Chun, 2008b, p. 321). Most remarkably, in following Chun's re framing of code as a re-source it becomes possible to demonstrate that there is a gap between source and execution. In Chun's terms, code is "ephemeral," it is intrinsically conflated with memory and undergoes constant and unforeseeable degeneration (2008a, p. 301). This implies that a program crash may not be directly related to syntactic or semantics errors in the writing of code, it could be due to run-time errors that the human and/or machine discovers during executing and running processes. In other words, errors might just happen when a program runs for a certain period of time and not previously.

This is not to say that Chun does not agree that source code is able to give us some understanding of computational logic as it is readable but rather

that she highlights the other layers at work. It is useful to extend this to a consideration of code inter-actions, which is how code brings together other digital objects that exist in different layers of computation during a run-time environment. Both Chun and Berry's concept of executing and running code inform this thesis' focus on code beyond mere reading and writing, together with attention to code running and execution, which enables a closer examination of the deep structure and operative processing of computation.

### 2.4.3 *Cold gazing in Media Archaeology*

Taking a very different reading approach to media archaeology from the German media theory tradition, the method of the 'cold gaze' does not focus on humanistic interpretation, rather it is used to engage with the mechanical and engineering dimensions of media (Ernst, 2013b; Parikka, 2011, 2012). Wolfgang Ernst is interested in using material processuality to rethink media history, including mathematical calculation, data structuring and other forms of material in the field of media archaeology (Parikka, 2011, p. 50). The cold gaze is considered to be mechanical, and hence has a "lack of emotion or semantics" (Parikka, 2012, p. 8). This method of cold gazing describes cold facts but not stories or interpretation, which is to say it is distinctly object-oriented, nonhuman-focused as opposed to narrative-based approach. This way of gazing at media objects, as Ernst puts it, "[is] enumerative rather than narrative, descriptive rather than discursive, infrastructural rather than sociological, taking numbers into account instead of just letters and images" (2013b, p. 251).

Media archaeology suggests getting close to media objects in order to understand cultural memory from technical media—gaining understanding from different devices, hardware and even software and any other techniques that facilitate the opening of black boxes, such as circuit bending and hardware hacking (Hertz & Parikka, 2012, p. 425; Parikka, 2012, p. 15). Media archaeology makes a theoretical and historical reference to the 'archaeology' of Michel Foucault, especially *The Order of Things* (1966) and

*The Archaeology of Knowledge* (1969) and, to an extent, Walter Benjamin's writings on historical materialism. As such, it relates to the inter-relations of the political and power discourses of things and knowledge. The difference between media archaeology and media history is that the later focuses more on historical text and narrative materials, while the former pays attention to the materiality of media objects, "the real technological conditions of expressions" which lie underneath the media surface as content (Hertz & Parikka, 2012, p. 427).

However, this methodology is usually oriented towards hardware examination. The examined objects are mostly obsolete, such as in Kittler's investigation of a gramophone and typewriter and Ernst's inspection of a phonographic recording device. Those digital objects examined are mostly physical objects that one can touch and listen to the sounds made as they are running. Each physical device or obsolete machine is relatively standalone, meaning that it rarely operates under, or within, the technological networks common to how we understand devices nowadays. The processing of signal is usually operated at the low level of, or close to that of, a machine.

Nevertheless, software platforms and networked databases are increasingly playing an important role in social media culture. The constantly changing conditions of the internet, such as the disappearance of websites, remain a challenge to many archaeologists who work in the field of media/internet archaeology (Helmond, 2013) and also those working across media archaeology into the domain of the arts and digital culture (Parikka, 2012, p. 124). This temporal dimension—the unpredictable life span of intangible software or a web page, might be one of the reasons why media archaeologists rarely use the method of cold gazing. Facebook, for example, renders the platforms seamlessly as there are at least twice update per day (Soon, 2014b) and unlike physical objects, there is no public manual or schematic at all for all the available functions and details of invisible logics behind the interface of Facebook like tracking and profiling.

One of the characteristics of the cold gazing method is its microscopic examination of time, including processes of data operation and synchronisation. It investigates how “time is being organized technologically” (Ernst, 2013b, p. 251). For Ernst, the dimension of time does not refer to historical time but largely means the micro-temporality of data transmission and signal processing that is regarded as time-critical.<sup>46</sup> In other words, the method of cold gazing in media archaeology emphasises the importance of the process-oriented internal machine/code operations and their relation to time, which is relevant to this thesis.

## 2.5 Reflexive Coding Practice

In addition to the methods of close reading, iterative trials and cold gazing, this research also employs coding practice to examine computational logics, procedures and inter-actions. Although the two terms coding and programming are somewhat similar,<sup>47</sup> favouring the use of the word ‘coding’ over ‘programming’ is a deliberate choice in order to emphasise two main concerns. First, code as a material for creative/artistic expressions beyond mere focus on technical functions and applications. The term coding has gained more currency in the sphere of software (art) practices, as evident in the sphere of live coding, as well as creative coding that highlights the intersection of art/design and technology (Maeda, 2004; Peppler & Kafai, 2009). Second, the practice of coding is close to what Cox, McLean and Ward refer to as “art-oriented programming,” and this implies a material-discursive approach with which “to acknowledge the conditions of its own making.” This refers to both the “formal qualities of code” as well as to the “critical discourses” around code (Cox et al., 2004, p. 161). This attention to materials and discourses is what Barad would call “material-discursive practice” (2007, p. 244), which is an on-going process of engagement with and as part of the world.

---

<sup>46</sup> This perspective of micro temporality can be found in Ernst’s writing (in German). Ernst, W. (2007), *Zeit und Code*, in D. Tyradellis & B. Wolf (Eds.), *Die Szene der Gewalt: Bilder, Codes und Materialitäten* (pp. 175-187).

<sup>47</sup> Both the terms programming and coding have their historical roots in the domains of mathematics and computer science (Blackwell, 2002; Hopper, 1955). Historically, the term programming came after coding and was more associated with money making skills in America (Billings, 1989, p. 51).

Coding is a process-based practice which requires “creative crafting” (Black, 2002; Hansen et al., 2014), substantial thinking and deep reflection continuously. In the words of artist, hacker and scholar Natalie Jeremijenko, “we think with things,” and she regards thinking as handwork and making (or coding, in this context) as intellectual activities. She explains: “I can’t make sense of the world in theoretical terms without the materiality of what actually works and the open endedness of how others interpret, receive and use things” (in Hertz, 2012). This comes close to the notion of “diffractive art practice,” a phrase adapted from Barad’s diffractive methodologies (2007) and proposed by Helen Pritchard and Jane Prophet to reflect on what constitutes practice in their view. For them, practice does not consist of mere materials but also the reflection, articulation and diffraction in which “materialities emerge as differentiated events, as they come together, in relation to one another” (Pritchard & Prophet, 2015, n.p). What they suggest is that practice is more than reflections that “look back onto art practice,” (Pritchard & Prophet, 2015, n.p) as there are other practices involved during the process of making (in this context which is coding) (Pritchard & Prophet, 2015, n.p).

In the discussion around the merits of artistic research, this thesis also pays attention to the notion of reflexivity in artistic practice (Borgdorff, 2011, 2014; Rolling Jr, 2014; Sullivan, 2010). Reflexive practice, according to Graeme Sullivan, “is a kind of research activity that uses different methods to work against existing theories and practices and offers the possibility of seeing phenomena in new ways” (2010, p. 110). This is somewhat similar to diffractive art practice or material-discursive practice as mentioned above, in which theory and practice come together but not in opposition. For art-based research scholar James Haywood Rolling Jr, the notion of reflexivity refers to thinking in and through material, context and practice continuously (2014, p. 163). The concept of reflexivity has its root in philosophy and education, putting emphasis on “reflection-in-action” (Brookfield, 1986; Schön, 1983), reflection on experiences (Dewey, 1991) and the “reflective learning cycle” (Gibbs, 1988), however these traditions are

more based on (semi-)finished objects, incidents or projects as a way to seek conclusion, or solve problems, or follow a linear and systematic cycle of reflection that is less concerned with the emergence of knowledge production, meaning making and the various entanglements of reading, writing and coding (art) practice. The practice of coding can be seen as a way of knowing and understanding how things work on an epistemic level, in which reflexively informs thinking about, and being in, the world. This onto-epistemological dimension, again references Barad, highlighting the inseparable practices of knowing and being (2007).

In this thesis, coding as software (art) practice involves a loosely configured experimental system,<sup>48</sup> a concept borrowed from historian of science Hans-Jörg Rheinbergerh, in which practice can be understood as the intertwining of technical objects and epistemic things. This characterisation alludes to “the technical conditions under which an experiment takes place and the objects of knowledge whose emergence they enable,” as explained by Henk Borgdorff in the field of artistic research (2014, p. 114). There is a fundamental difference between reading and analysing someone else’s code and code written and experimented by oneself.

Importantly, the notion of experimental practice embraces indeterminacy, which refers to the “not yet crystallised status of the knowledge object” (Borgdorff, 2014, p. 114). Unlike having fixed routines with clear objectives, problems, hypotheses or evaluation and without predictable and expected results or answers derived from what has been known, not yet known, or still to be known. Coding practice, as a form of experimental process, is associated with a journey of “instability, indeterminacy, serendipity, intuition, improvisation, and a measure of *fuzziness*” (Borgdorff, 2014, pp. 114-6), similar to other kinds of artistic practice, which are regarded as dynamic and creative. Artworks, according to Borgdorff, to be epistemic

---

48 Experimental System is originally used by Hans-Jörg. Rheinberger, a historian of science, who defines it as “a basic unit of experimental activity combining local, technical, instrumental, institutional, social, and epistemic aspects” in the domain of scientific research (Rheinberger, 1997, p. 238).

things<sup>49</sup> that “constitute the driving force in artistic research” (2014, pp. 114-6). To conceive art practice as research, artworks can act as a means or a mode of inquiry, to reach out for things that are unknown or are not yet known. In this way, the use of coding practice, as a form of experimental practice, in this thesis is a “discovery-led” process (Borgdorff, 2011, p. 56).

This mode of epistemic inquiry through artistic research does not have the goal of producing formal knowledge like a new scientific discovery or an innovative artefact (although some might do so especially in new media art that utilises technologies) but, crucially, it is a process undertaken in order to inform and invite what Borgdorff describes as “unfinished thinking” and is considered as part of the notion of reflexivity. He explains this as follows:

artistic research seeks not so much to make explicit the knowledge that art is said to produce, but rather to provide a specific articulation of the pre-reflective, non-conceptual content of art. It thereby invites *unfinished thinking*. Hence, it is not formal knowledge that is the subject matter of artistic research, but thinking in, through and with art (Borgdorff, 2011, p. 44, *original emphasis*).

What he highlights is that there is no fixed boundary of knowledge that art may produce, shifting the attention to ongoing and unfinished modes of reflexive thinking and practice. This reflexive process of unfinished thinking describes not only artefacts that are produced in this thesis but also as an inspirational concept in thinking through this written manuscript, that it has informed the last chapter of this thesis with its title of “Unfinished Thesis.”

The methodology of this thesis is to use a combination of the aforementioned

---

<sup>49</sup> This is similar to how Cox and Jacob Lund argue artistic practice to be a “means to provide epistemic enquiry” (2016, p. 32). They claim that nonhumans, not only humans, “allow us to perceive what is knowable or even unknowable” (Cox & Lund, 2016, p. 29). Therefore, these epistemic things also play a significant role in shaping meaning and producing knowledge

methods, including close readings, iterative trials, cold gazing and reflexive practice to examine code and its inter-actions with and within, material substrates. I present three of my own artistic research projects in the form of software (art) practice that are interwoven into the discussion and articulation of liveness in the remaining three chapters. I have taken a major role in conceptualising procedures, writing code, implementing and actualising the projects in exhibition and workshop settings.

In Chapter 3, I undertake a close reading of the Twitter platform and its API, as well as the code of data query that is required to access and extract online data in my collaborative artistic work, *Thousand Questions*. Through refining the code in the spirit of iterative trials we have developed a version that emphasis the process of querying data, including accessing, selecting, filtering and presenting data, to examine the vector of unpredictability. Additionally, the changing technological landscape forces the code to be updated and implemented differently and this informs the conceptualisation of the ‘inexecutable query’ that I will discuss in detail in the next chapter.

In Chapter 4, I use cold gazing method to examine digital signal processing, packet switching, protocols and data buffering which exhibit the vector of temporality. I closely investigate the architecture and mechanism of a central processing unit (CPU), the technical specifications of internet protocols, as well as the logic of data buffering. Consequently, I offer a detailed technical description and analysis of how these things are presented, focusing on the deep operative processes and their implications. Additionally, I present my coded experimental project, *The Spinning Wheel of Life*, as a means of reflection on a perpetual running environment that invites “unfinished thinking” (Borgdorff, 2011, p. 44) on the temporality of distributed networks.

Chapter 5 takes the artwork *Hello Zombies* as its central example, unfolding the vector of automation. By undertaking a close reading of the code syntax, the chapter selects a particular block of code from the entire program that offers a technical explanation of each line, conceptualising them with the



notion of automation. This chapter additionally presents a sketch of Alan Turing's halting problem in the form of my short written code which further explains the constituent forces in algorithms.

The three artistic projects as described briefly above are developed in conjunction with the written component of this thesis. However, these material practices are not separated from reading, writing and reflecting on critical discourse around code but, instead, work in a reflexive manner, continuously thinking in, through and with practice (Rolling Jr, 2014; Sullivan, 2010). The last section of each remaining chapter, which I refer to as 'Notes on Reflexive Coding Practice.' It is presented through a textual method of self-narrative written in different font style and together with various images and screenshots within a defined box, a presentation format borrowed from technical textbooks which offers material which does not fully fit into the main chapter but is also not entirely separate from it. These special sections are not designed to fit in the main flow of the chapter but to exemplify the notion of reflexivity by documenting research that is associated with my practice which informs the understanding of liveness in ways that analysis is not able. This includes evidence of the systematic use of the methods introduced here, for example the self-narrative text, flow charts and procedures, contextualisation of the works, some of the notes, examples of the most recent work and work-in-progress experiments, source code, programming comments and so forth. On the one hand, reflexivity can be thought of as being undertaken by a practitioner who reflects continuously before, during and after actions and allows differentiated events to emerge through practice. On the other, the documentation demonstrates some of the processes of how code, materials and artworks inform and unfold the understanding of things. This is to illustrate how I think with things and how the materials inform the critical discussion of software (art) practice. The latter reinstates the materials, things and processes to the centre of the projects in which they are encountered as forces which co-produce meaning and knowledge.

The methodology of reflexive coding practice is demonstrated throughout the

whole thesis, in articulating, analysing and reflecting on the process-driven projects, as well as reading, writing, running and executing code that collectively and contingently informs each other. Therefore, knowledge is regarded as co-produced by different practices that are somewhat merged together. As a consequence, the practice component is not separated from the writing of this thesis and they are interwoven within the chapters as opposed to being presented as supplementary supporting materials.<sup>50</sup> Similarly, following the same line of argument, my findings are not only demonstrated in the written text but also in the running of the projects themselves. The live computational processes of code running carry, unfold and express the argument of this thesis by themselves that follow the practice of software art. As explained earlier, software artworks allude to the software itself is the work, that is the presented artistic projects express themselves with the live inter-actions of code and exhibit material forces while executing and running code.

Tellingly, the subjects or objects involved in this research journey do not consider humans to be the sole actors, and as I hope I have made clear to this point in the thesis nonhuman entities also play a crucial role in acting and searching for knowledge. It is equally important to remind the reader that code also acts upon itself as well as inter-acting with other materials that are not directly apparent to humans. Code acts and performs through the process of execution and running. It operates in a continuous manner that acts and responds reflexively too. In view of that, this chapter seeks to establish that the execution of code is also a site of knowledge production, co-emerging with other reading and writing practices.

Considering the issue of whether artworks can generate knowledge, there are in fact many scholars with a coding practice that clearly demonstrate that knowledge can be produced in, within and beyond, the artworks themselves. McLean, for instance, who works in the field of live coding, wrote his PhD thesis reflecting on his position as an artist-programmer and

---

<sup>50</sup> This may be also commonly seen as the tradition in Denmark.

the findings are mutually informed by practice and theory (2011, p. 18). That said, it is not “ruled by theory,” rather, is embedded in the reading and writing activity of academic and coding practices (McLean, 2011, p. 119). Similarly, Daniel C. Howe, together with Helen Nissenbaum, developed a browser add-on project called *TrackMeNot*<sup>51</sup> (2006) which introduced their concept of obfuscation in order to tackle the problem of surveillance and data profiling by search engines in a tactical way. As seen in both of their subsequent practical and scholarly reflections,<sup>52</sup> the practical works inform their reflexive thinking about computer networks and surveillance as demonstrated in their publications over the years<sup>53</sup> (Howe, 2015; Howe & Nissenbaum, 2009; Howe et al., 2011).

As with different versions of *TrackMeNot*,<sup>54</sup> it is worth noting that the reflexive production of artefacts also encompass additional features, fixes, “real-requirement[s]” (Berry, 2011, p. 67), priorities and decisions that respond to the changing landscape of contemporary software culture. What Borgdorff referred to as unfinished thinking, therefore, may be extended to a consideration of the code-based artworks with different versions that are never considered to be finished artefacts. These unfinished objects express an unfinished and continual reflection on the world, and therefore, some of the previous versions of my projects are also mentioned in the last section of each of the remaining chapters.

In summary, this chapter has introduced reflexive coding practice as a key aspect of my methodology, in addition to other methods of close reading, cold gazing, iterative trials and reflexive practice that are employed in order to pay attention to code reading, writing, running and execution. With the

---

<sup>51</sup> See: <http://cs.nyu.edu/trackmenot/>

<sup>52</sup> Within the following chapter’s footnotes, I have also listed my publications, conference presentations and exhibition records of the works that demonstrate how thinking has evolved and emerged with various practices. This serves to demonstrate an ongoing “dialogue” between artists and their works (Sullivan, 2010, p. 110), or to exhibit Borgdorff’s notion of “unfinished thinking” (2011, p. 44; 2014, p. 117).

<sup>53</sup> There are other scholarly works who use *TrackMeNot* as the central case in their research output (Al-Rfou et al., 2012; Peddinti & Saxena, 2010).

<sup>54</sup> For the artwork *TrackMeNot*, there are more than 6 releases over 9 years (from 2006-2015). Each release adds different features and fixes that comply with new web browser versions.

materialist approach as an overall conceptual framework, the materiality of code inter-actions is foregrounded by taking into consideration “interactions” (Beaudouin-Lafon, 2008; Bentley, 2003; Murtaugh, 2008; Wegner, 1997) and “intra-actions” (Barad, 2003, 2007) that produce multiple forms of agency.

Additionally, this chapter presents three key concepts that are considered to be fundamental and related to the understanding of contemporary software culture. Firstly, invisibility was used to address the materialisation of code and the opaqueness of computational processes. Secondly, performativity was used to examine the relationship between code and language as well as the operational logics of code that produce performative effects and highlights machine agency as a way to think about the materiality of code. Thirdly, generativity was discussed to introduce a certain degree of autonomy in a system. These three concepts together provide a basis on which to understand some of the current debates in the field of software studies, in which the three vectors of liveness (namely unpredictability, micro-temporality and automation) will be further developed upon in subsequent chapters.



### 3

## Executing Unpredictable Queries

Allegedly the first digital literary work, *Loveletters* (1952), was built using the Ferranti Mark I, the world's first computer to be commercialized, by Christopher Strachey at the University of Manchester. It is a computer program that employed an algorithm developed early on by Alan Turing's for generating random numbers. The love letters were generated through a combination<sup>55</sup> of grammatical rules that referred to adjectives, nouns, adverbs and verbs as well as random choices of sentence structure (Wardrip-Fruin, 2011). Predating ELIZA<sup>56</sup> natural language processing, a computer program for the study of computational linguistics, the utilisation of computation, randomness and linguistics in the 1950s marks the beginning of the history of software studies. In 2012, David Link, an artist and media archaeologist, won the first Tony Sale Award with his software artwork titled *LoveLetter\_1.0*<sup>57</sup> (2009). Made through reconstructing the algorithms and executing the original code of Strachey's *Loveletters*, *LoveLetter\_1.0* ran on a Mark 1 emulator and was able to reproduce and regenerate pieces of love letters.

Darling Sweetheart

You are my avid fellow feeling. My affection curiously clings to your passionate wish. My liking yearns for your heart. You are my wistful sympathy: my tender liking.

Yours beautifully

M.U.C<sup>58</sup>

*Figure 3.1: A love letter from LoveLetters*

---

<sup>55</sup> The grammar logic is: "My—(adj.)—(noun)—(adv.)—(verb) your—(adj.)—(noun)" (Wardrip-Fruin, 2011, p. 309).

<sup>56</sup> In 1966, the developer of ELIZA, Joseph Weizenbaum, published the article on ELIZA, which was conceived as the pioneer software written for the study of natural language communication between human and machine (1966).

<sup>57</sup> See: [http://www.alpha60.de/art/love\\_letters/](http://www.alpha60.de/art/love_letters/)

<sup>58</sup> M.U.C refers to Manchester University Computer. Another letter can be found also in Strachey's article titled *The "Thinking" Machine* (1954, p. 26).

The *Loveletters* was able to generate 318 billion variations with different combination of words and sentences (Link, 2006) like the example love letter above. The love letters are the results of computational generative processes, therefore they are more than representations (Wardrip-Fruin, 2011, p. 302). *Loveletters*, as digital media scholar Noah Wardrip-Fruin argues, is an *unpredictable manifestation* through two hidden elements: data and processes (2011, p. 306). Wardrip-Fruin is not interested in the resulting letters as semiotic and poetic representations but more in the generative processes themselves (2011, p. 306). Such generative processes produce unpredictable results from accessing the databank of a range of words and by using Turing's random algorithm. He explains that *Loveletters* includes the data it employed, execution processes and representational output in the form of a text that together can be considered as a "system" (Wardrip-Fruin, 2011, p. 307). It is the system that generates unpredictable love letters.

This chapter investigates the unpredictable vector of liveness, the notion of unpredictability that is inherent in examples such as this. It takes its cue from how Wardrip-Fruin analyses computational processes that move beyond the meaning of their representational output. In particular, it pays attention to unpredictable manifestation through data processing: how data is being processed, generated and represented. My collaborative artistic project *If I wrote you a love letter would you write back (and thousands of other questions)*<sup>59</sup> (from hereon referred to as *Thousand Questions*) is inspired by the multiple possible variations generated through the implementation of simple rules running autonomously, in which the system takes control of generating unpredictable outcomes. The concept of generativity is key to an understanding of computational processes such as

---

<sup>59</sup> The artwork was co-produced with British artist Helen Pritchard. It was first exhibited at Microwave International New Media Arts Festival in Hong Kong (2012), as part of *Digital Futures*, at the Victoria and Albert Museum (2013), presented in the research workshop *Artistic Research* at Kunsthal Aarhus (2015), and most recently the latest version includes a visual component that is presented in the International Conference on Live Interfaces in Sussex (UK) and Si Shang Art Museum International Art Conference in Beijing (China) in 2016. In the same year, the work is selected by and published in *Electronic Literature Collection* (Volume 3). It will show in forthcoming Kochi Biennale exhibition in 2017 (India). See the artwork's documentation: <http://siusoon.net/home/?p=900>

this, and the notion of unpredictability in logical systems. Instead of generating love letters the work *Thousand Questions* takes ‘questions’ from the internet as text and ‘voices’ them. An example of the questions is ‘If I wrote you a love letter would you write back?’

On a technical level it uses a web API (Application Programming Interface) to query data from the internet platform, Twitter. In other words, the notion of unpredictability is manifested through the real-time query of an API within a networked and distributed environment. In contemporary conditions data is generated in real-time and evolved over time. I refer to this kind of data query as *live queries*. A live query is implemented as part of a program or a piece of software and, in the work *Thousand Questions* as a snippet of code. In an installation setting it runs continuously, repeatedly executing queries and retrieving different sets of questions from Twitter automatically, it is unpredictable because you never know what question will be retrieved. The project employs the web API that is offered by Twitter, extracting questions from an infinite pool of possibilities (databank) in real-time as the audience experiences an endless computer synthesised voice that speaks those questions that were posted by internet users of Twitter. Instead of using different combinations of grammar rules and words, as in *Loveletters*, *Thousand Questions* works with dynamic data from the constantly updating databases which then produce unpredictable vocal manifestations as an integral system.

As demonstrated in Chapter 1, unpredictability is identified as one of the important perspectives from which to examine the notion of liveness. What marks media phenomena *live* concerns unpredictability. Using my collaborative artistic project *Thousand Questions*, which extracts data and executes queries in a networked environment, this chapter analyses the notion of unpredictability through a materialist account of executing live queries. It unfolds the computational process of a query’s execution through an understanding of its operational and generative logics, similar to Wardrip-Fruin’s emphasis on a system that comprises data, processes and output (2011, p. 306-7).



This chapter explores the ‘unpredictable manifestations’ of a networked system and, more specifically, how the unpredictability of live queries can be understood through their material encounters. *Thousand Questions*, a computer program, is still running today and it operates parasitically with the internet platform querying data from Twitter, an ever-updating database. While this chapter focuses on the present use of technology that is still running and operating, Wardrip-Fruin uses the perspective of media archaeology to analyse an object (in his case *loveletters*) from the past: “the predigital media”, or “more recent past” (2011, p. 302). Through reflexive coding practice, in particular reading code-related materials,<sup>60</sup> writing computer code<sup>61</sup> and running and executing data queries this chapter aims to articulate the complex materiality of network conditions and their computational processes which shed light on the understanding of unpredictability in contemporary software culture.

### 3.1 Queries and Databases

In *Loveletters* both the software and its data<sup>62</sup> were stored and run inside the same machine, the Ferranti Mark I. This was one of the early stored-program computers, which was based on von Neumann architecture which enabled both data and instructions to be stored in the same computer memory. Following his theoretical concept of a ‘Universal Computing Machine,’ in which tape was used to demonstrate the holding of data and instruction in 1937, Turing’s revolutionary concept was realised in the world’s first commercial computer, the Ferranti Mark 1. The promise of a universal computing machine suggested that a computer could take in, compute and output data, steps that enabled it to solve problems and perform assigned tasks by using instructions as symbols and algorithms in sequential steps (Parisi & Fazi, 2014, p. 116; Turing, 1937). However, many

---

<sup>60</sup> This includes the discussion, library, specification and documentation of Twitter web APIs, Twitter databases and accessing methods.

<sup>61</sup> *Thousand Questions* is developed through a Java-based open source software called Processing.

<sup>62</sup> This includes all the adjectives, nouns, adverbs, verbs and letter start (Wardrip-Fruin, 2011, p. 309).

scholars critique the model of the Turing machine and consider it to be insufficient to cope with the dynamics of environmental input in digital networks (Parisi & Fazi, 2014, p. 121; Wegner, 1997, p. 83). More than half a century after the concept of the Turing universal machine, with the invention of database management systems, distributed networking, the internet, World Wide Web, hypertext systems, cloud computing and blockchain technology,<sup>63</sup> the computational world is far more complex. Since the Web 2.0 era, there are increasing amounts of user-generated content which is stored in so-called social media platforms, physically located in server farms. These platforms<sup>64</sup> operate across data centres beyond a single machine, whereby data are held and linked together in a manner which can be retrieved by a specific query method. This is a system that consists of input, process and output, internet platforms and networked applications which can now efficiently inter-act with many other machines, extracting data from cloud servers and processing it across distributed environments through live queries. The way storage systems and code instructions work nowadays are significantly different from the Turing computer, specifically there are endlessly updated and dynamic feeds on social media platforms in which a network of computers communicate with each other across time and space. What would have happened if *Loveletters* had been comprised of an infinite data stream operating within a distributed internet network?

In contemporary culture many applications offer data streams or feeds with infinite stored data sets where their databases undergo a never-ending update of records. Databases have a significant impact on contemporary conditions and it is through the storage and analysis of massive amounts of data (so-called 'Big Data') that profiling, targeted marketing, personalised recommendations and various sorts of predictions and e-commerce become

---

<sup>63</sup> The term blockchain came from the two concepts: block and chain in Bitcoin, an electronic cash system that is invented by Satoshi Nakamoto in 2008. Blockchain promotes decentralised network by using a peer-to-peer network protocol that involves participants to validate each transaction. In other words, blockchains suggest a visible and transparent process whereby actions, such as creating, transferring, verifying digital assets, are taken by participants of a network. It is a chain of blocks contains transactions that linked to one another and enforced with cryptography, maintaining a continuously-growing list of data records as distributed databases (Nakamoto, 2008).

<sup>64</sup> An example of a platform is Google where data centers operate around the world. See: <https://www.google.com/about/datacenters/inside/locations/index.html>

possible. According to Chun, user habits formulate big data businesses, and she explains, “*Through habits users become their machines*: they stream, update, capture, upload, share, grind, link, verify, map, save, trash and troll” (2016, p. 1, *original emphasis*). Browsing, Googling, messaging and Tweeting, for instance, become our habits and they are storable, traceable and analysable in the form of data that is kept in databases. Behind all of these habits the role of databases cannot be underestimated.

Databases do not only enable the storage of data but also the organisation of data and the retrieval of information. Retrieving from and navigating databases suggest new cinematic and narrative experiences for users where data is organised through code. This allows a more dynamic of real-time computation to occur, showing different content at different time and for different person. Manovich argues for the “database as a cultural form of its own” in which it “present[s] a different model of what a world is like” (1999, p. 80). Therefore the database matters to us as it changes our experience of the world and digital media, such as computer games, hypertexts, database cinema and other interactive interfaces. Manovich’s analysis stems from his notion of narration as “a set of links” that are structured around and within databases, generating meaning and showcasing new aesthetic possibilities (1999, pp. 90-4). Mathematically, a ‘set’ refers to set theory in which all mathematical concepts are based on; “A set is formed by the grouping together of single objects into a whole” (Hausdorff, 1957, p. 11). In his article, *More parts than elements: how databases multiply*, Mackenzie discusses some of the mathematical and philosophical implications of SET theory underlying databases (2012). He highlights how set-like operations create new relations and how data relations can be established through unions, intersections and complements. Therefore database can be understood as more than a storage system, rather it is a system about relations on multiple levels.

There are many discussions of various database models in the realms of software studies and digital humanities, from mainstream relational databases (Castelle, 2013; Ramsay, 2004) to other alternative models such

as NoSQL (Dorish, 2014), MapReduce (Mackenzie, 2012) and blockchain databases (O'Dwyer, 2015, 2016). Scholars have investigated various material aspects of databases including database design, database infrastructure, database organisation, digital information, technical-social processes, industry practice and operation. However less attention has been paid to the concept of query, or data query processing in particular, which is used for database communication. Data queries are widely implemented at the level of code to communicate with the database, querying data records. It is further manifested in today's mobile gadgets, sharing buttons and "social plugins" (Gerlitz & Helmond, 2013).

Query is most commonly understood as a language. The concept of query was first introduced in Edgar F. Codd's article, *A Relational Model of Data for Large Shared Data Banks* (1970), which addressed the linguistic aspect of collecting relational data and foresaw its power when incorporated into other programming languages. He says,

Such a language would provide a yardstick of linguistic power for all other proposed data languages, and would itself be a strong candidate for embedding (with appropriate syntactic modification) in a variety of host languages (programming, command- or problem-oriented) (Codd, 1970, p. 381).

Codd's vision has been realised such that querying a database does not need to take the form of a command-line terminal but can be embedded in many different forms. Structured Query Language (SQL) is one of the most popular query languages for communicating with databases, particularly on relational databases, such as Oracle and MySQL. SQL can be executed, meaning that it provides instructions for storing, querying and manipulating data. Although the notion of query has a historical relationship with SQL and SQL databases (Codd, 1990, p. 7), it should be understood that query could be also used in different alternative types of databases beyond the structure of relational databases.

Databases are more about data storage and data relationship. The notion of query that I address in this chapter is focused on the process of data query that communicates with different kinds of databases. Computer scientists Ashok K. Chandra and David Harel define a query as follows:

[a] query language is a well-defined linguistic tool, the expressions of which correspond to requests one might want to make a data base. With each request, or query, there is associated a response, or answer (1980, p. 156).

Therefore the execution of a query is a two-way communication, both a request and a response.

Set operations have been commonly used to bring data into relations and relations are expressed and established through executing a query. Set-like operations can be done through writing statements for managing and manipulating data in a database. According to Mackenzie, “Any query to a database takes the form of a ‘SELECT’ command. The syntax of ‘SELECT’ ranges from extremely simple requests for a single row of a single table to highly complex intersections, unions, and joins spanning many tables” (Mackenzie, 2012, p. 340). Besides this, the most frequently used query statement in a commercial context is the ‘SELECT’ query, which is not updating or inserting data but retrieving it (Tuya et al., 2007, p. 398). A query is an inquiry into databases and this inquiry does not refer to a statement only. A query, as a form of code, performs when it is executed and a result will be returned. A result is a combination of relations that answer and respond to one’s query statement. With the availability of data records, a query has the capability to specify, create and identify relations through this request and respond logic, such as the ‘SELECT’ syntax. These selected relations are things behind ‘tags,’ ‘playlists,’ ‘(Google) analytics’ and categorisation and are further manifested into “suggestions, connections, menus, recommendations, and invitations” (Mackenzie, 2012, p. 340) in contemporary software culture. Thus queries exhibit a certain material power that executes the inclusion and exclusion of specific type and range of

data, therefore queries are not simply to be regarded as neutral commands.

The terms *dynamic queries* (Shneiderman, 1994) and *visualizing queries* (Consens et al., 1992) have been used to indicate the functional aspect of a query to visualise databases. In particular, computer scientist Ben Shneiderman discusses the empowerment offered by dynamic queries that enable a direct manipulation of a visual outcome where users have more control. Yet these articles fall short of taking into consideration of the dynamics of networked technologies. For example the constant update of databases and distributed networks.

The use of the term *live queries* in this thesis is not limited to any specific database models or their technical structures and organisation. Arguably, live queries allow data to be inquired of, and queried from, centralised, decentralised or distributed databases via networked technologies. This can be understood via the network typology that engineer Paul Baran proposed in 1964. In Figure 3.2, it shows Baran's three types of networks. Type A is a centralised network, in which there is only one central node that acts as a server in which data can be sent to participants. However, participants are not allowed to communicate with each other. In contrast to type A, a decentralised network is illustrated as type B. It is commonly used by telephone systems whereby the network does not need to have "complete reliance upon a single point" (Baran, 1964, p. 1). Type C, a distributed architecture is seen as a grid or mesh-like network. The internet is a distributed communication network, in which the destruction of an individual node or link will not impact the whole transmission channel. This model is also implemented in what is known in internet computing as 'packet switching,' where messages can be delivered to their destinations via multiple pathways.

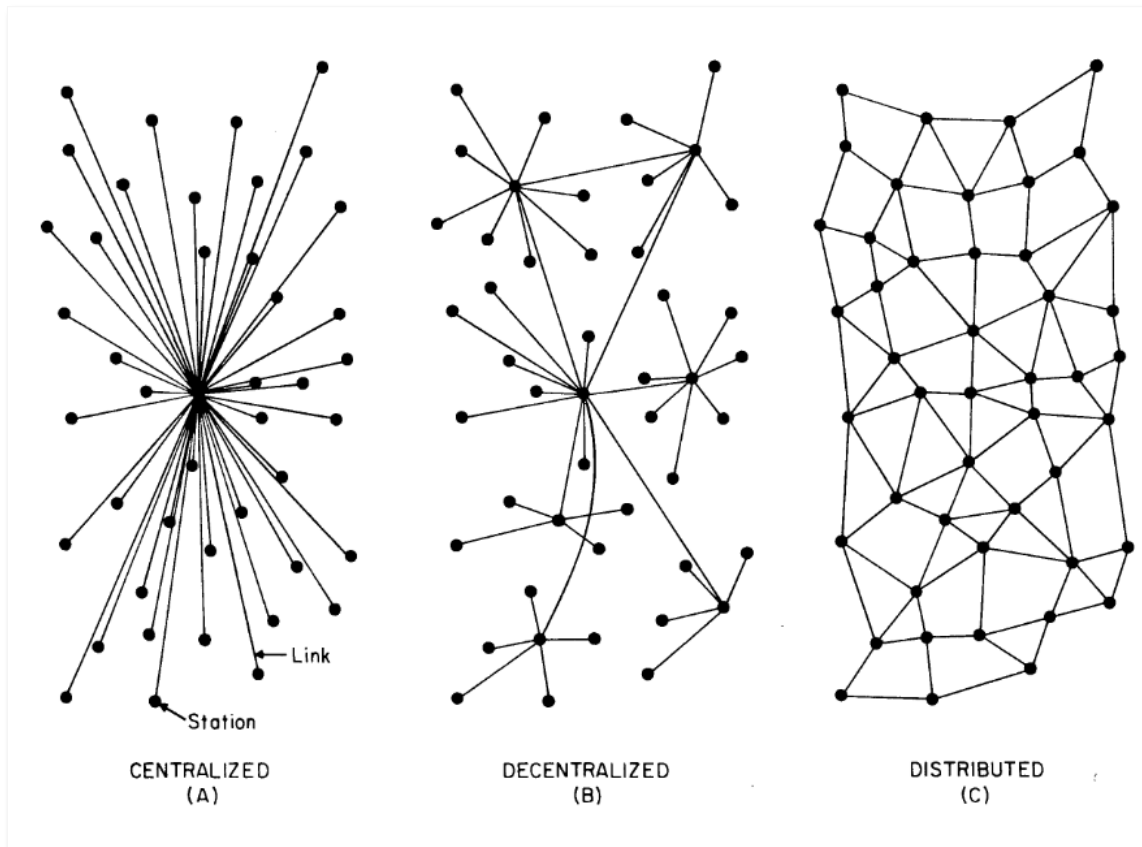
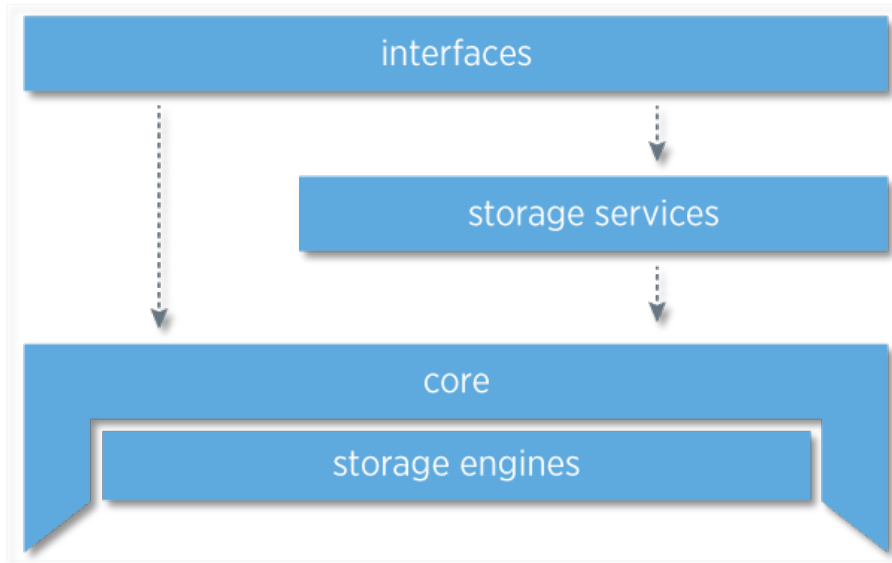


Figure 3.2: Centralized, Decentralized and Distributed Networks. Reprinted from *On Distributed Communications* (p. 2) by P. Baran, 1964.

Live queries include different kinds of ways that data is queried from a database that is centralised, decentralised, distributed or a combination<sup>65</sup> of the above. A structured format of live queries allows data exchange between sites, platforms, machines and applications in real-time. Data can be queried, specifically selected, filtered, generated, sent and collected from an enormous databank that is operated across different database structure. Taking Twitter as an example, it first uses a relational database MySQL and gradually moves to NoSQL databases (such as Cassandra and Gizzard), because the later can handle massive data sets and therefore better support for time-critical queries (Metz, 2014). Although their latest system, which is called the ‘Manhattan database system,’ includes private multiple systems and storage engines (see Figure 3.3), access to the system can be gained through the web query, which is also known as web API (Schuller, 2014).

<sup>65</sup> The blockchain database is said to be a combination of decentralized and distributed models (O'Dwyer, 2015).



*Figure 3.3: The Manhattan system of Twitter. Retrieved from <https://blog.twitter.com/2014/manhattan-our-real-time-multi-tenant-distributed-database-for-twitter-scale>*

According to media and software studies researcher Taina Bucher, one of the advantages of API is that changes of database infrastructure would not impact the format and operation of the data query (2013, n.p). Therefore it appears seamless to the users of web APIs, as the process of live querying does not depend on a particular infrastructure and technical arrangement. As Bucher puts it,

[APIs] separate modules into public and private parts, so changes to the private part can be performed without impacting the public (the API itself) part, and therefore minimizing the dependencies between these two parts (2013, n.p).

This section lays out the general understanding of a query and how it can be executed independent of database type. The next section will discuss in more details about the format of query.

### **3.2 The format of query output**

A query is executed in the form of sending an input and receiving an output within a system. Both query requests and query responses are highly



structured, employing a particular format and mechanism for structuring data. At a structural level, cultural critic Johanna Drucker argues that a digital format demonstrates grouping, grammar and rules that have powerful effects. This kind of format “contain[s] protocols that enable dynamic procedures of analysis, search, and selection, as well as display” (Drucker, 2009, p. 11). A query output comes with semantic naming that describes the data using a set of ‘tags’ for structuring data. In addition, those tags are the key parameters used for locating specific content when the system responds to a query output. In other words, a structural format facilitates the dynamic retrieval of data, in which textual processing is enabled. Through habits of digital search, searching for videos, books, news or a word meaning for example, formats bring “the object of their inquiry into being” (Drucker, 2009, p. 11). These search habits are the quest for knowledge, suggesting formats “must be read as models of knowledge, as discursive instruments” (Drucker, 2009, p. 11).

Within the context of web queries, platforms or service providers offer a clear format for not only specifying the request for data, but also how it will be returned in another way round. Javascript Object Notation (JSON) is one of the most popular formats used for data exchange between applications. It is designed for inter-operability, meaning that it is a standardised format used to exchange data between applications written in different programming languages<sup>66</sup> (Crockford, 2006, p. 7). Such standardised formats are used widely, allowing thousands and millions of developers in the world, who work on different programming languages and platforms, to retrieve data and process queries. In other words, using a standardised format that can be easily parsed, like JSON from Twitter, enables wider distribution, circulation and application of data queries.

In JSON, data returns in four primitive types (strings, numbers, booleans and null) (Crockford, 2006, p. 1). Within the example of *Thousand Questions*, a query request will return more than 20 objects in primitive

---

<sup>66</sup> The languages include ActionScript, C, C#, ColdFusion, Common Lisp, E Erlang, Java, Javascript, Lua, Objective CAML, Perl, PHP, Python, Rebol, Ruby and Scheme (Crockford, 2006, p. 7).

types, including field names 'text,' 'id,' 'isTruncated' and 'geolocation' to name just a few. Below is a sample tweet returned<sup>67</sup> from the Twitter platform using the Twitter API:

```
tweets=[StatusJSONImpl
{
  createdAt=Mon Feb 29 16:21:25 CET 2016,
  id=70432560996301217,
  text='[...]' ,
  source='<a href=http://twitter.com/download/android
rel="nofollow">Twitter for Android</a>',
  isTruncated=false,
  inReplyToStatusId=[...],
  inReplyToUserId=[...],
  isFavorited=false,
  isRetweted=false,
  favoriteCount=0,
  inReplytoScreenName=[...],
  geoLocation=null,
  place=null,
  retweetCount=0,
  isPossiblySensitive=false,
  isLanguageCode=en,
  contributorsIDs=[...],
  urlEntities=[],
  hastagEntities =[],
  mediaEntities=[],
  currentUserRetweetID=-1,
  [...]
}
```

*Figure 3.4:* An experiment to extract a sample tweet returned from Twitter platform.

Together with the corresponding values of types, they are regarded as objects. Therefore, an object includes a field name and its value, such as 'geolocation=null,' identifying the specific data content in a structured way.

---

<sup>67</sup> The use of the symbol [...] indicates sensitive information.

On the one hand the field's name explains the semantic meaning, and on the other, the field is well structured in the sense that same field is returned for every query request. These same returned fields enable data to be programmed in *Thousand Questions*, allowing 'questions' to be extracted every time but with different values that were manifested as a perceivable and unpredictable voice.

To understand the format in a deeper way, it is worth noting that JSON follows "JSON Grammar," separating different names and values (Crockford, 2006, p. 2). This grammar is a set of rules for structuring data. For example, a pair of left and right curly brackets, the symbols of '{ }', indicates a 'begin-object' and an 'end-object.' A further example is a comma ',' that separates each object as a name/value pair. To process a query output format and be able to acquire appropriate data involves the identification and extraction of specific data through code. This procedure is called 'parsing.' Parsing is an operational and technical method often used to analyse structured data that follows certain rules and grammar. Given an output format with different fields or values as indicated in Figure 3.4, *Thousand Questions* analyses the output in order to extract the 'questions' (that is the field 'text', see line 5 in Figure 3.4), among many other fields, for further processing. Therefore those structures, including namings, brackets and commas, are things that are essential in automated data processing. Analysing, searching, selecting, displaying and speaking of queries, the questions of *Thousand Questions*, are all automatically run. The structures in the output format are designed *to do something* as an indicator. For example, a comma indicates a value separator, a pair of curly brackets indicates data objects and the field 'text' in the last example indicates the required field as a tweet with a question mark. These structures and rules are embedded in a format and have a role to perform and this is what Drucker describes as "performative" (2009, p. 11). A query, as a form of code, is conflated with languages, symbols, meanings and actions that is similar to the performativity that has been discussed in Chapter 2.

A query format is comprised of rules indicating how the data is being

structured and how text parsing should be done. According to professor of culture and technology Jonathan Sterne, a format represents a range of decisions “that affect the workings of a medium. It also names a set of rules according to which a technology can operate” (2012, p. 7). To parse data also means handling a specific set of hierarchies, syntaxes and symbols through code. From the perspective of coding, there are two structured types—objects and arrays—in JSON and these require different functions for parsing data. In the example of Figure 3.5, the excerpt of code is about parsing a weather query in JSON format. By submitting a query that specifies geographic coordinates to the platform OpenWeatherMap,<sup>68</sup> the returned result provides a list of cities and the corresponding weather details as output queries. Since there is more than one city within the specified zone, the query outputs an array of cities and their corresponding weather data that are in the types of object and array. How the data is structured results in using a different function for parsing. In Figure 3.5, there are two different functions—‘JSONArray’ and ‘JSONObject’—used to extract different types of data. Therefore a format and its structure change the way a program should be written. A standard format is not regarded as an isolated set of rules but rather, the format is an active process because the rules of a format require certain libraries, functions and code that enable it to parse and read the output. The notion of format is therefore more than a static instance, it formulates a set of processes and infrastructural elements that support such a standardised format. Any changes in a query format, regardless of any decisions behind it, literally impact upon coding practices and the running artefacts.

```
JSONArray record = response.getJSONArray("list");
for (int i = 0; i < record.size(); i++) {
    JSONObject result = record.getJSONObject(i);
    String name = result.getString("name");
    println(name);
}
```

*Figure 3.5:* Excerpt of code, in Processing Software, for parsing JSON query from OpenWeatherMap for getting a list of cities’ name.

---

<sup>68</sup> See the detailed API’s parameters and usage: <http://openweathermap.org/current>

JSON is just one of many formats with a set of specific rules and other formats such as XML and RSS are also widely used, like podcasting, in the similar way to JSON. JSON is an increasingly popular format in web industries (Amyatwired, 2011; Hamp, 2010). Most data returned by Facebook APIs<sup>69</sup> and Sina Weibo APIs<sup>70</sup> are written in JSON although some are in XML too, whilst Instagram only offers JSON format. In the case of Twitter,<sup>71</sup> JSON is the only output format in the Twitter web API v1.1 and other formats including XML, RSS and ATOM were made obsolete, along with API v.1, in 2013. A change in format alters operative and computational processes. For example, the handling of data queries requires change in both providers and users in the form of code at both structural and infrastructural levels. The consequences of system upgrades, program updates and documentation revisions are indeed affecting different practices in various industries and cultural sectors. Therefore, a change in format is not a mere technical shift, but additionally, as claimed by Sterne, one which “may mark a significant cultural shift” (2012, p. 12).

### 3.3 Query as cultural form

From Raymond Williams’ argument that “television has altered our world” (1974, p. 9) to Christiane Paul and Manovich’s assertion of the “database as cultural form” (Manovich, 1999, n.p; Paul, 2007, p. 98), technological objects, such as television and databases, are associated with many kinds of cultural and social activities, through which different events are transmitted and delivered to a screen. I argue that the provision, consumption and execution of queries are equally paradigmatic cultural forms in contemporary software culture. Offering Web APIs becomes a standard package of online and social platforms, at least in the case of the major companies across the Eastern and Western continents, including but not limited to Google, Facebook, Instagram, Amazon, PayPal, Sina Weibo, WeChat, Twitter and Youtube.

---

<sup>69</sup> For the Facebook APIs output format and Public Feed API format, see: <https://developers.facebook.com/docs/unity/reference/current/Json> and [https://developers.facebook.com/docs/public\\_feed](https://developers.facebook.com/docs/public_feed)

<sup>70</sup> See: <http://open.weibo.com/wiki/Statuses/update/en#Response>

<sup>71</sup> See: <https://blog.twitter.com/2013/api-v1-retirement-final-dates>

According to Tim O'Reilly, who popularises the term Web 2.0, one of the important aspects of Web 2.0 services is data management that allows "remixability": "remix the data into new services" (2005). This data remixability not only includes capturing, storing and organising data but also as media studies scholar Anne Helmond highlights, its redistribution (2015, p. 6).

One of the cultural consequences is that 'social plugins' (Helmond, 2015) have become commonplace on many websites. Webpages usually come with a list of social media sharing icons, such as Pinterest, Twitter and Instagram. This development is particularly apparent across a wide range of content, from online news platforms to academic online journals and magazines to many other kinds of websites. When a social media icon is clicked on these sites the data is updated in the corresponding social media database and the computed result will display accordingly. The action of a click executes a query that is underneath the graphical user interface of a webpage, where the number of shares, likes or favourites is computed by reading the accumulated acts and writing the new record in another database. So, for example, an image on a news website is added to a Pinterest database when someone clicks on the Pinterest symbol. In other words, an API as a specific form of query demonstrates the capability to read and write.

In the fields of digital humanities and sociology an API is an important tool and object of study for social data research and user behaviour analysis. For instance, media studies scholars Anja Bechmann and Peter B. Vahlstrup discuss the implications and challenges of using APIs from Facebook and Instagram (2015). Tyler H. McCormick et al collect and process user data and tweets using Twitter API for Social Science Research (2015). Under the influence of digital humanities, institutions<sup>72</sup> are also providing workshops and seminars to researchers from different disciplines, offering information

---

<sup>72</sup> Here are just a few examples: MITH API workshop (2011) by University of Maryland, APIs as Interfaces to the Cloud (2012) by The Digital Methods Initiative, Cleaning and Exploring Your Data with Open Refine (2015) by University of Western Sydney.

on the development, usage and critique of APIs. Clearly, the practice of querying data is becoming an important research topic in data analysis beyond the discipline of computer science.

With “the rising values of APIs” and with many big and small companies providing APIs that extract value out of the available data, it is claimed that offering an API creates “new business opportunities,” enhances “existing products, systems, and operations,” and develops “innovative business models” (Mason & McKendrick, 2015, n.p). In parallel the critiques of APIs in journal articles have been increasingly seen in academia. Together with media studies scholar Carolin Gerlitz, Helmond analyses the “like economy” in Facebook via Facebook APIs (2013). Helmond, in another publication, argues that the politics of data flows in web platforms have been transformed from open standards to proprietary APIs (2015, p. 22). Likewise, Bucher suggests that APIs exhibit control and freedom through her examination of the Twitter API (2012a, 2013). In addition to the widely available web APIs mentioned so far, Berry discusses how the use of specialised and private APIs expose some of the relations between companies like Microsoft and the political economy of software development (2011, pp. 70-1). An investigation of these APIs, with the capability to read and write, suggests that this form of query is highly related to different cultural and industry practices. In other words, studying API queries enables a better understanding of different platforms and the politics of data circulation associated with contemporary software culture.

A query can be thought of as an object, both an object of study and an object in terms of how to use it technically and economically. Fuller suggested that digital objects are pervasive, and as such identifiable, traceable and analysable (2004, p. 27). Records kept in databases, including health records, telephone records and library records. To Fuller, these digital objects “are in connection with a million relations of dimensionality” (2004, p. 28). The digital object is about the creation of social and technical relations, “[making] stable different kinds of sociability and inter-relation with other elements” (Fuller, 2004, p. 28).

Bucher argues that Twitter APIs express “enactive power,” and yet this is contingent rather than stable (2013, n.p). She argues that a query is a quasi-object beyond its standard specification. The prefix ‘quasi’ refers to the social desires in which using objects can bring about the social organisation’s goals (Bucher, 2013, n.p). In addition to this, control, according to Bucher, is exercised through the standards, structures and specific social situations. Therefore, an API may be understood as an active participant in as well as a component of a set of relations. She explains:

This implies viewing APIs not merely as specifications and protocols that determine relations between software and software, but also in the sense of the quasi-object, as protocols that structure and exercise control over the specific social situations on which they are bought to bear. Drawing on Roland Day’s claim that quasi-objects are best understood as historical projections of power within organizational and epistemic structures, the argument is made that the kind of work that the Twitter APIs perform, needs to be situated within the platform politics of data exchange and transmission (Bucher, 2013, n.p).

Following Bucher, I acknowledge live queries as active participants in a system that acts and performs with wider cultural consequences. A query execution includes both technical and social dimensions and thereby has political significance. From digital objects to quasi-objects, we could say that a query is an object that creates dynamic social relations that includes the active participation of nonhuman entities.

In addition to the academic and business sectors, there are increasing numbers of artists who use queries in their artistic practice. As mentioned in the last chapter, *Net.Art Generator* (1997)<sup>73</sup> uses Google search query to

---

<sup>73</sup> Ten years after the work was presented in a web page format, Cornelia Sollfrank exhibited the artwork in a museum space - Kunsthalle Schirn in Frankfurt during 2007-2008, in which the final setup included a screen that showed the live query processing, running on a physical server machine.



manipulate images of Warhol's flower paintings. The setup of the artwork reveals the process of data query by installing a computer with all the process logs which the audience can see (see Figure 3.6). Another more recent artwork, *Endless War* (2011), developed by YoHa with Fuller, is an installation that reveals the real-time processing of data (Afghan War Diaries) from Wikileaks. The visual presentation is structured "from a series of different analytical points of view: each individual entry, phrase matching between entries and searches for the frequency of terms" (YoHa & Fuller, 2014, n.p). In its gallery installation, the three video screens display the result of the execution of live queries as text, as well as constantly performing the execution of queries. In addition, the artists exemplify the materiality of data processing through exposing the inner workings of computational and data processing sounds by placing the workstations close-by (see Figure 3.7). As a result, the installation is a rich receptive experience that is expressed in both visual and audio forms. Many software artworks employ computation but in many cases, as Wardrip-Fruin also points out, computational processes are "invisible on the surface of their projects" (2011, p. 320). *Net.Art Generator* and *Endless War* may be considered as examples of artworks that make visible the materiality of query processing.

---

The change of the presentation shows that Sollfrank started to notice the importance of data query as part of the resulted Warhol flower images.  
See the video documentation: <https://www.youtube.com/watch?v=43y2k5j7oIU>

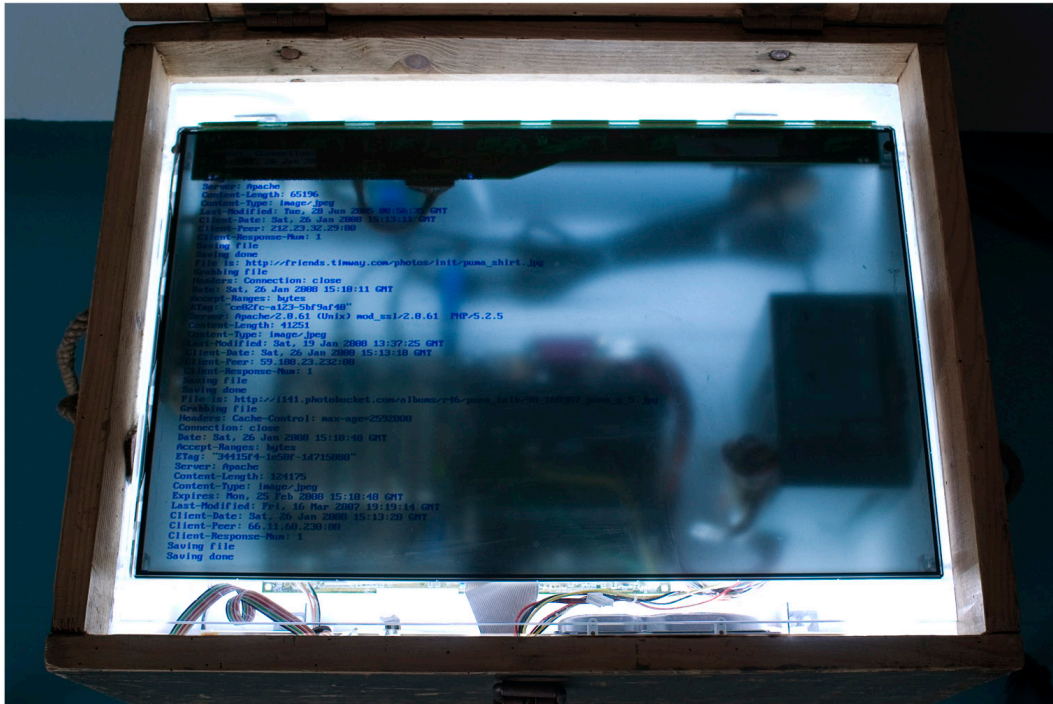


Figure 3.6: *Net.Art Generator* by Cornelia Sollfrank. Photo: Nina Pieroth. Courtesy of the artist



Figure 3.7: *Endless War* was shown in Hong Kong as part of the exhibition *Tracking Data: What you read is not what we write* (2014). Retrieved from <http://www.writingmachine-collective.net/wordpress/?p=489>

I have worked with query since 2009. The collaborative project, *If I wrote*

*you a love letter would you write back (and thousands of other questions)*, drew thousands of questions based on the key symbol—a *question mark*—from the Twitter social media network, synthesising questions in text form to speech. The project employs query, following the standard and official Twitter API<sup>74</sup> format (using REST Search API) that offers programmable access to search and extract Twitter data. By using the Twitter web API, the query in *Thousand Questions* was written with various criteria and conditions of data extraction as part of the larger query request. This includes content search that is comprised of a question mark (?) and that tweets must be in English. In addition, the returned query output only includes 50 results (tweets) per request and they are regarded as ‘recent tweets’ according to the definition from Twitter. A query statement, like this, consists of multiple parameters.

For the latest development of *Thousand Questions*, the visual component includes the returned collective questions. However, they are not in a readable format. A screen displays only one character per frame until all the remaining characters are shown (see Figure 3.8). Using text to speech feature, a woman’s voice<sup>75</sup> is heard, speaking all 50 tweets one after the other. The program repeatedly poses questions, alongside other parameters, to Twitter once the artwork finishes displaying and speaking all the pending tweets. Therefore, the experience of such live queries results from interactions between different machines<sup>76</sup>—ongoing cycle of requests and responses that are both operational, cultural and social. The work makes query processing apparent by showing the latency and the temporal aspect of getting questioned tweets and speaking unanswered questions through an audio-visual experience. The project is meant to be an endless process of query processing, in which temporality is expressed through the display of the underscore symbol ‘\_’, indicating the unknown waiting of the program for the next query execution (see Figure 3.9). This unknown can be

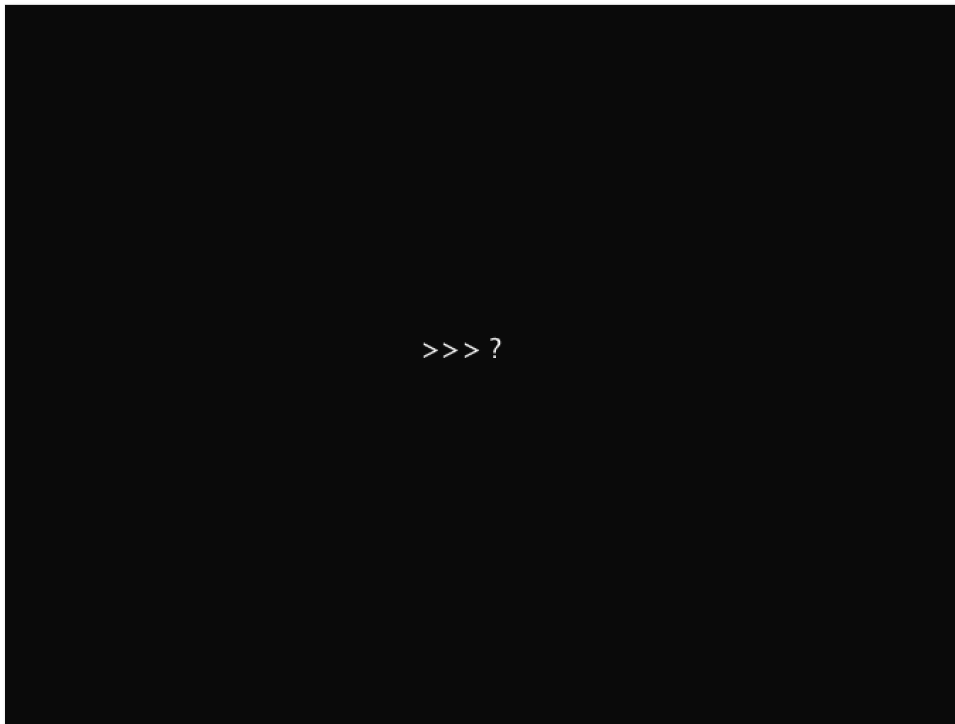
---

<sup>74</sup> See: <https://dev.twitter.com/rest/reference/get/search/tweets>

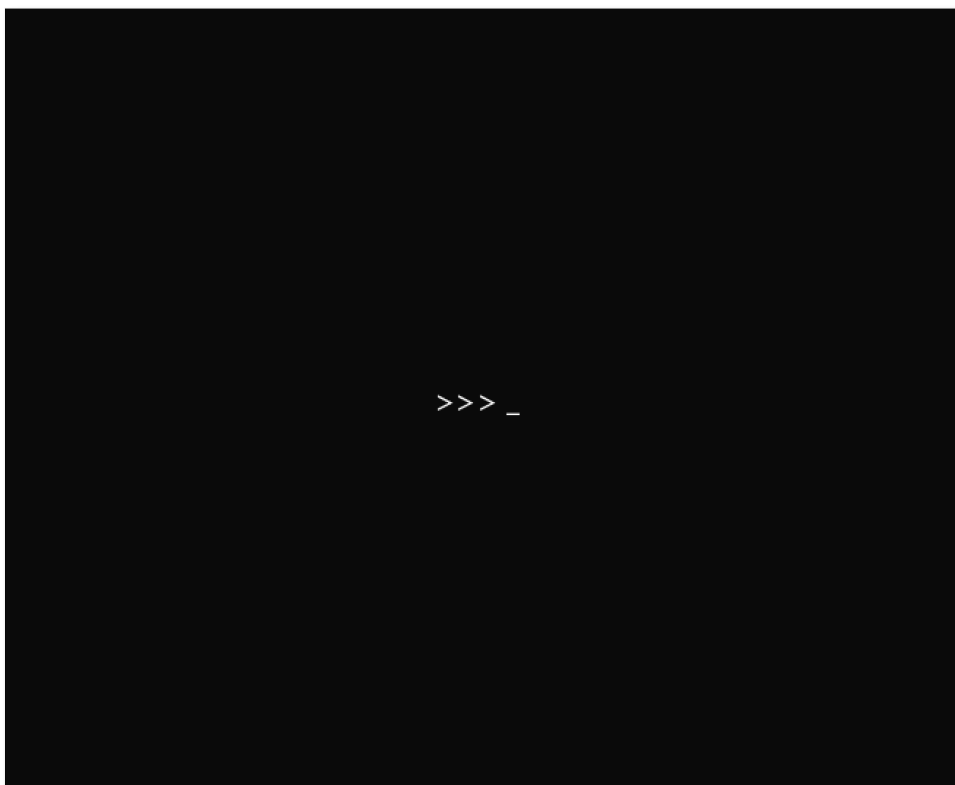
<sup>75</sup> We are using the text to speech feature Mac Operating System. The Australian woman is called Karen who speaks the tweets.

<sup>76</sup> The use of machines here refers to the machine that runs *Thousand Questions*, Twitter’s machines, and also those social machines that in contact with the Twitter platform.

understood in two ways: both the time of query processing and the content of “unanswered queries” (ELC3, 2016).



*Figure 3.8: A screen shot of *Thousand Questions**



*Figure 3.9: A screen shot of *Thousand Questions*, where the program is waiting for the next query execution*

### 3.4 The unpredictability of live queries

Query is a form of code, and it inter-acts with databases and network protocols. Considering the case of *Thousand Questions*, the query consists of different text and symbols wherein meaning is attached. Systematically, the query output contains different fields,<sup>77</sup> including but not limited to a timestamp, a name of an author, an unique identifier, a favourite count, a retweet count and, sometimes, geographical coordinates. Although rules are pre-set and fields are highly structured, the process of data query exhibits dynamic and unpredictable qualities that modulate the sensation of liveness.

In view of Twitter as a web 2.0 platform, it can be understood as both an open and closed system. The system is open in the sense that it allows anyone, including both human and non human agents (such as bots) to post questions across time and space. It is also a closed system in so far as it can be considered a black box wherein the processes, algorithms and mechanism of showing tweets and returning query output are rather opaque. In the remainder of this chapter I will unfold the complexity of this openness and closedness of the Twitter platform and those associated unpredictable events that occur when executing a query. Again, the focus here is not so much on the meaning of output texts and questions but more a consideration of the two hidden elements—data and processes—as suggested by Wardrip-Fruin. These two elements unfold the dynamics of query execution in a real-time environment.

My focus is on the unpredictability of live queries that results from thinking and practicing code reflexively. Such an approach responds to questions that arise from the project. What does it do, and what does it mean when a query is being executed? We can ask the same question differently, borrowing concepts from set theory, what are the mathematical operators or symbols

---

<sup>77</sup> See for different fields of a tweet: <https://dev.twitter.com/overview/api/tweets>

doing in a search operation? What do the micro-processes of executing queries tell us? How do changing formats relate to unpredictability? Ultimately, what constitutes unpredictability through executing live queries? These lines of inquiry help establish the logic of unpredictability that matter, to argue that the execution of queries is a live and unpredictable process.

### *3.4.1 Random events*

In the previous chapter, the notion of generativity was discussed in relation to complexity science in order to explain the dynamism of systems. Random events exist in a complex system, generating chaotic behaviours that are unpredictable. Following that, this section applies the concept of generativity to explain the random events in Twitter, which is a highly generative and complex system. Within complex systems, components interact with each other in ways that lead to unpredictable phenomena and the cause is difficult to trace inasmuch as complexity intensifies at each level of interaction across all components and layers. Generative systems consist of rules, and Twitter, a complex system, these rules are many and various and take account of conditions, human and nonhuman activities which have emerged over time. Professor of the philosophy of science, Roman Frigg, would describe the tweets as ‘random behaviours’ from a system perspective. Frigg argues randomness is unpredictable in the context of dynamical systems and explains, “an event is random if there is no way to predict its occurrence with certainty. Likewise, a random process is one for which we are not able to predict what happens next” (Frigg, 2004, p. 430). This way of thinking about tweets, to borrow from Frigg, stems from “the seemingly random, stochastic, unpredictable or haphazard time evolution” of a system (Frigg, 2004, p. 412). Since there are different inter-actions and dynamic processes at play these also impact the process of live queries. This section offers some explanation of the complex system of Twitter in order to gain a general understanding of the system behind query execution.

Using Twitter data to do analysis and prediction is an important area for research, as observed in the fields of computer science, political science, media studies and social science (Burghardt, 2015; Gayo-Avello, 2013; Zaman et al., 2010). Although there are more and more methods for researching predictive power that is based on data collection, it still seems impossible to achieve perfect accuracy and/or prediction. There is no perfect prediction of when a tweet arrived and from where and from whom it derives. This has been also seen in the political campaigns, such as Brexit and United States president election in 2016, in which actual voting results are significantly different from big data analysis and prediction.

To explain further, it is necessary to discuss Twitter as a Web platform that engenders dynamism. In 1999, Darcy DiNucci, who works in the area of user experience design, was one of the first to describe the phenomena of having dynamic media on the web. What makes dynamic media possible is the standardised web infrastructure of protocols (TCP/IP and HTTP), in which data can be transported across different devices and screens (DiNucci, 1999, p. 32). This standardisation implies individuals can access the Web using different mobile or stationary configurations and hence more data could be generated due to its increased accessibility. In 2004, O'Reilly and John Battelle defined *Web 2.0*, making the distinction between “the web as platform” rather than *site*, which was more like a publishing channel (2004). One of the important concepts of Web 2.0 is idea of participation in which the role of a user is changed from that of a viewer that of an active participant. User-generated content became a key component driving new online business models (O'Reilly & Battelle, 2004). From a system point of view, Web 2.0 refers to a platform that not only allows one way communication that outputs dynamic media but also takes input from users for data processing as part of the computational output.

The term *openness* is associated with the concept of participation, in which the platform is open for participation and content generation. Writing in 2015, Helmond further articulated the concept of a platform that is programmable, enabling a participation which extends from end users to

developers. She explains:

In order to become a platform, a software program needs to provide an interface that allows for its (re)programming...[The web API] makes a website programmable by offering structured access to its data and functionality and turns it into a platform that others can build on (Helmond, 2015, p. 35).

Adding detail to the ideas of O'Reilly and Battelle, Helmond argues that API is an active agent that changes social media from the paradigm of sites to platforms (2015, p. 35). I extend this discussion by arguing that participation can be also achieved through a programmable query.

Within the context of the Twitter platform, there is one important note to make: not all users of accounts in Twitter are human. Rather it is a combination of the human and nonhuman that constitute individuals or users. For instance, nonhuman bots are a growing phenomena on the internet. An example is the twitter bot project called *moth generator* (2015), developed by artists Katie Rose Pipkin and Loren Schmidt, which automatically generates beautiful moth images on Twitter (with the account: @mothgenerator). These bots are programmed and tweets are automatically updated without leaving any browsing logs.<sup>78</sup> Although no exact data specifically about nonhuman bots has been released by Twitter, the company admits that there are about 23 million automated accounts<sup>79</sup> that are regarded as active users as of June, 2014 (Seward, 2014, n.p). Perhaps, the availability of the programmable query can explain why Twitter has automated accounts, in which bots are programmed through logics: automatic tweets are posted through the use of Twitter web APIs. In other words, participation extends from mainly human to a collective of human and nonhuman agents. What counts as 'active users' or 'individuals'

---

<sup>78</sup> In contrast with a human, a bot does not tweet like a human. A person usually uses a particular device, using a specific operating system to access an application or browser to look for interested tweets. As a result, less data can be traced through these bots.

<sup>79</sup> There is no specific indication of automated accounts are equal to nonhuman bots, because spam may be also count into this value.



might also account for this increasing nonhuman participation, performing automatic query in the realm of web 2.0 platforms.

Such openness, in part, constitutes the unpredictability of tweets. The notion of unpredictability refers to how Twitter, as a system, cannot predict input events, such as when the next tweet will arrive, how many characters it will contain or what the content of the tweet will be (such input events directly impact the query results). Following Frigg, tweets are regarded as random behaviours as it is impossible to predict the next one. Certainly, there are many factors, both internal and external, on both individual and collective levels that influence such random behaviours. To help explain this I have grouped the factors into three main categories: individual behaviors, external events as well as structure and format which are presented in Figure 3.10 as a conceptual model to explain the random behaviours from Twitter (random) input.

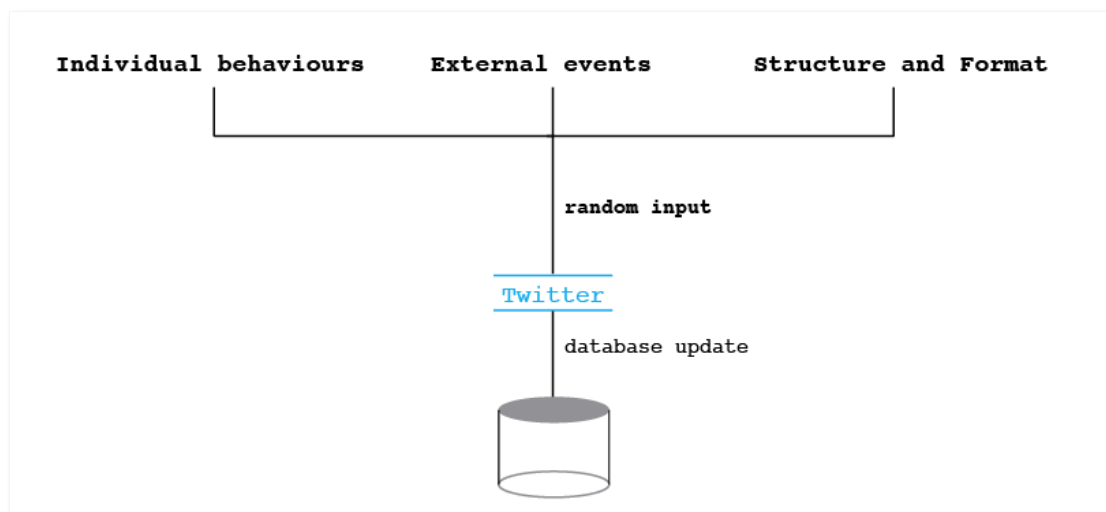


Figure 3.10: A conceptual model of Twitter random input

I now discuss these three categories in more detail. Firstly, individual behaviours more matters that are individually-specific including, but not limited to, the personality, language practices, typing habits, peer-influence and emotions at the time of initiating a tweet. Culture clearly influences an individual’s language practice. For example, someone who grows up in Denmark is more likely to use Danish to write tweets. Language is a

complex subject that is influenced by daily practices and the surrounding environment and it is also location-specific. People in Northern China and Southern China, for instance, would speak and write differently. Behaviour is a highly complex and specialised subject, as it is also related to sociology, anthropology, psychology, cultural studies, genetic studies and mind studies where the different and multiple factors that influence human behaviours are elaborated. In chapter 1, I discussed unpredictable human behaviour in the context of a live program show (Gadassik, 2010). It is clearly impossible to analyse all the factors that influence human behaviours but this section points out that behaviour is a complex subject with multiple influencers, and it is made even more complicated in this context given that Twitter has 271 million<sup>80</sup> active users. Complexity intensifies at each level of interaction according to different individual behaviours, contexts and experiences.

Secondly, external events, such as catastrophes, might increase the amount of tweets. This is similar to the television study as discussed in Chapter 1, in which Doane describes how crises or catastrophes generate a disrupted routine and argues that this constitutes the understanding of unpredictability of television (2006). Gadassik also explains that the interruption of events to a continuous TV programme, such as “economic crises, tragic accidents, natural disasters, and human casualties” are something that make television is a live medium (2010, p. 121). Social media like Twitter also shares this kind of response to external events. However, one of the differences between social media and television is that the former is considered to be a more open platform that supports two-way communication and interactions, allowing participants to respond, hence generating more usage. Researchers Jessica Li and H.R. Rao trace the noticeable increase in Twitter usage during the severe earthquake in Xinhua, China in 2008 (2010). Therefore, external events are one of the forces that made the tweets unpredictable. Other than catastrophes and disasters, there are other external events that can drive the traffic of Twitter data, such as political campaigns, sports competitions and cultural

---

<sup>80</sup> The statistic is based on the Earnings Report in 2Q, 2014 (Seward, 2014, n.p).

activities—presidential elections, the World Cup and the Oscars. Additionally, many scholarly works demonstrate that social networks have emergent properties, where certain topics can become pervasive and ‘trend’ over a period of time (Birdsey & Two, 2015). All these external events may trigger random inputs to the Twitter platform.

Thirdly, the structure and format of Twitter itself shapes the input of tweets too. It is arguable that the number of retweets, the number of favourite counts and the tweets that display on a screen would inform and activate user actions. Artist-researcher Benjamin Grosser has developed a similar line of inquiry based on the Facebook platform. His artwork *Facebook Demetricator*<sup>81</sup> (2012-present), a browser add-on hides all numbers on the Facebook interface, “illuminates how metrics activate the desire for more, driving users to want more likes, more comments, and more friends.” Grosser also argues that metrics, such as the number of likes in Facebook, directly impacts user participation (2014, n.p). Besides, in terms of the computational logic of Facebook, the post with more likes or comments will get more visibility on the Facebook interface where lesser metrics would result in a lower visibility to others (Dredge, 2014, n.p). Furthermore, Bucher explains that the visibility of feeds on Facebook is calculated based on “the multiplication of the affinity, weight and time decay scores,” in which it is subjected to the interaction between users, the popularity and the freshness of a post (2012b, pp. 4-5). There are complex invisible logics that govern a post’s structure. Although Twitter does not use exactly the same logic as Facebook, filtering is implemented in order to show limited tweets on a screen at a time. The logic of Twitter does not follow a chronological order in displaying the latest tweets on screen, it is algorithmically structured (Newton, 2016). Consequently, the positioning and visibility of a post may also impact individual behaviour, as scholars from computing and information science, Tad Hogg, Kristina Lerman and Laura M. Smith, state, “responding to a post is conditioned on seeing it and being interested in it” (2013, n.p). Therefore, how the content is being

---

<sup>81</sup> See: <http://bengrosser.com/projects/facebook-demetricator/>

structured visually and algorithmically also contributes to the overall input of tweets.

Apart from the structure and format that is oriented to visibility and end-users' interaction, the specification and documentation of a query might also impact upon developers' usage as many third-party applications are built around Twitter. As the founder and president of technology companies, Peter Gruenbaum, explains, "Good documentation is important in encouraging and keeping developers interested in your platform as well as reducing support costs" (2010, n.p). Good documentation includes sample code and detailed querying approaches, explanation of authentication and error handling (Gruenbaum, 2010, n.p). With clear specification and documentation guidelines, there will be more API connections between third-party applications and Twitter, hence more data contributing to the Twitter platform.

There are many factors and perspectives that can help to explain Twitter as a dynamic system. What has been established so far is that there are different processes involved that drive the random inputs in Twitter. The three categories discussed are just part of the whole complex assemblage of the Twitter platform and my focus has been on the input source. For my purpose here, the dynamic Twitter platform and its random events shed light on the *questions* of live queries that are harvested from Twitter as *unpredictable* phenomena. Following Frigg's argument, the unpredictable occurrence of an event is regarded as random from a systems point of view (2004, p. 430). Therefore, Twitter is a dynamic system consisting of unpredictable occurrences of tweets.

Indeed, these are non-linear consequences that are non-reducible to any individual tweet and, according to the classic 'butterfly effect,' the level of complexity increases with the number of components (see also Chapter 2, section 2.2.3). This is not, however, simply calculating the sum as greater than the parts because the central idea of a complex system is emergence, which is self-organised. Individual components develop a collective and

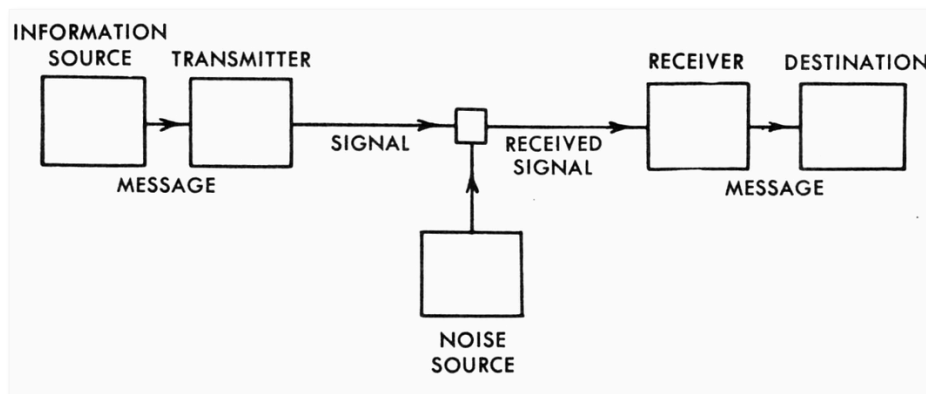
emergent pattern through their connection and coordination, evolving over time (Johnson, 2001). As such, if Twitter is to be regarded as a complex system then emergence is exhibited through multiple dimensions. This is not a top-down dimension that entirely dictates the input sources and flows of inter-actions but a bottom-up, a relatively open system that allows emergent behaviours to develop and evolve from “low-level rules,” structures and constraints to “high-level sophistication” (Johnson, 2001, p. 18)—a distributed connection of human and nonhuman inter-actions.

### 3.4.2 *Noise, entropy and randomness*

Shifting from the perspective of input sources to the transmission of data, the tweets, then we can see the letters and images that comprise tweets broken down into the basic unit for digital communication, namely binary digit, also known as bits in information theory as introduced by Claude Shannon in his article *A Mathematical Theory of Communication* (1948, p. 1). To understand the dynamic processing of a query we may need in turn to know how data is being processed beneath the tweets as perceptible textual materials. Following the discussion of information theory, tweets would not be understood as consisting of the message per se but of binary digits—signal—where data is being processed. This is more about understanding how a piece of information is transmitted from a source to a destination through signals. Shannon’s perspective removes the semantic aspect of a message, a text or a tweet.

By making a query request to the Twitter platform or receiving the query output, data passes through a communication system. As suggested by Shannon, this includes an information source, a transmitter, the channel, the receiver and the destination (1948, p. 2). Regardless of any communication system, both discrete and continuous (e.g. digital network and analogue television respectively), noise is necessarily present as a force that interferes with the transmission process (see Figure 3.11). Therefore, the information, logics, structure, and both request and response of query

are operated in a sphere of signals, noise and interference.



*Figure 3.11:* Schematic diagram of a general communication system. Reprinted from “A Mathematical Theory of Communication”, by C. Shannon, 1948, The Bell System Technical Journal, 27, 2.

Noise is an active force that perturbs a signal during transmission. “This means that the received signal is not necessarily the same as that sent out by the transmitter” (Shannon, 1948, p. 19). Most importantly, according to Shannon, “[t]he noise is considered to be a chance variable[...]. In general it may be represented by a suitable stochastic process” (1948, p. 19). Scientist and mathematician Warren Weaver explains further:

If noise is introduced, then the received message contains certain distortions, certain errors, certain extraneous material, that would certainly lead one to say that the received message exhibits, because of the effects of noise, an increased uncertainty (1949, p. 19).

There is uncertainty on the receiver side which it may contain something else, added noise, not exactly the same data as that on the sender side. There is no perfect prediction which can calculate the degree of discrepancy that is impacted by the force of interference and the resulting noise. What is known within an information transmission process is that noise increases the amount of information, whereby the received information at the receiver’s end is not equal to the sender’s side. In other words, “noise introduces error” during transmission (Shannon, 1948, p. 20). This is also why noise is

normally considered as error, something that is not expected to receive and that is different from the sender's signal. Noise is introduced at any point during any transmission process, including but not limited to the transmission within and between users' computer, the Twitter's servers, databases and specifically, for *Thousand Questions*, the machine that interacts with Twitter.

To understand and measure how information is transmitted with noise it is necessary to turn to the concept of entropy in information theory. The way information is processed is subjected to "the amount of freedom of choice" in selecting a message over a channel in bits (Weaver, 1949, p. 4). A selected message requires translation and transmission as binary digits via a channel involving memory registers, relays, switches, electronic circuits, routers and cables. Such a selection of binary digits, the encoded message, requires taking into consideration that all the digits as a whole constitutes the amount of freedom, as Weaver explains:

The concept of information applies not to the individual messages (as the concept of meaning would), but rather to the situation as a whole, the unit information indicating that in this situation one has an amount of freedom of choice (1949, p. 4).

However, the choice here is somewhat unpredictable. The relation between choice and unpredictability is that: "greater freedom of choice, greater uncertainty, greater information" (Weaver, 1949, p. 8). Weaver drew upon physical sciences to explain that the amount of freedom of choice is similar to "the degree of randomness, or of *shuffledness*" (1949, p. 5, *original emphasis*). In the second law of thermodynamics the notion of entropy is associated with the surrounding of a system like heat, pressure and temperature, which is arguably similar to the function of noise in information processing because both are actively changing the behaviour of a system in a random manner. Considering the amount of freedom of choice or the degree of randomness in a system, if a system is more ordered or organised it is then considered as less random and vice versa.

Weaver notes that if the “situation is highly organized, it is not characterized by a large degree of randomness or of choice—that is to say, the information (or the entropy) is low” (1949, p. 8). Based on the model of information theory, the discrepancy between the sender’s and receiver’s end is caused by the forces of interferences and this can be further explained through the concept of entropy. In addition to input sources of a complex system, the entropy of channels’ transmission also contributes to the overall understanding of unpredictability.

Information theory considers messages as binary signals that transmit over a channel but it is equally important, as it is for the general understanding of unpredictability, to examine what constitutes signals, its basic form, the binary value of either 0 or 1, in digital computers. Algorithmic information theory concerns not only information and randomness but also computation as well. It concerns a computer program’s size that “determine[s] the algorithmic information content of a message” (Klaus Mainzer, p. 194). In other words, algorithmic information theory studies the inherent structure of data objects. Similar to Shannon’s information theory, both theories are interested in the notion of randomness beyond the semantic dimension of a message. Gregory Chaitin, an algorithmic information theorist, investigates the structure of data randomness that stems from mathematics, binary digits and computation. He explains, randomness to be “based on the observation that the information embodied in a random series of numbers cannot be *compressed*, or reduced to a more compact form”(Chaitin, 1987/[1975], p. 4, *original emphasis*). In Figure 3.12, he gave a simple example to explain the complex concept with two binary strings.

```
010101010101010101010101  
01101100110111100010
```

Figure 3.12: Two binary strings

Clearly, the first one has a pattern and one can describe it in another manner such as: ten consecutive instances of 01. On the contrary, the second



one is seemingly random and there is no way to describe it except reading out the whole series. In the context of code instructions, the first sequence could be written as “Print 01 ten times” and “Print 01101100110111100010” (Chaitin, 1987/[1975], pp. 4-5). Computationally, there is no way to reduce the size of the second series, meaning that it contains almost the same number of bits of information as the first series which can be compressed. As such, the second series is regarded as random because it is patternless and cannot be further compressed (Chaitin, 1987/[1975], pp. 4-5). Informed by Chaitin’s algorithmic information theory, Parisi considers randomness as follows:

Not arbitrary complexity, but a form of entropic complexity defined by an infinite amount of data that cannot be contained by a smaller program. In algorithmic information theory, something is random if it is algorithmically incompressible or irreducible (2013, p. 266).

Unpredictability is one of the results of randomness and when we compared the two series of strings the first one that is less random can be more predictable because of the repeatable pattern of strings. In other words, the more random a string, the more unpredictable its sequence.

Understanding the fundamental character of randomness helps to explain the translation of computer code to machine code, which is essential in any kind of computer program. In the case of a compiler,<sup>82</sup> a software program invented by pioneer computer programmer Grace Murray Hopper in 1952 called ‘A-0 compiler,’<sup>83</sup> the compiler takes the role of translating “machine-like pseudo-code into machine code” (1955, p. 3). A compiler has the capability to link pieces together and to instruct “a generator to produce a specific input routine” (Hopper, 1955, p. 3). All instructions, together with data, have to be converted to the form of binary digits, both input to and

---

<sup>82</sup> See Hopper (Hopper, 1955).

<sup>83</sup> This A-0 compiler was originally written for the UNIVAC | computer, the first commercial business computer system in the United States.

output from a machine, for computation during the process of compilation (Chaitin, 1987/[1975], p. 6). A compiler is one of the applications of algorithmic information theory that handles data compression and optimises machine code (Chaitin, 1987; Dietz & Mattox, 2005; Kistler, 1997). In the work of *Thousand Questions*, developed in the Java programming language and environment called Processing, when a 'run' button is pressed (it means to execute and run the program), it converts source code (also called sketch) into Java byte code<sup>84</sup> that could be processed by any operating system and computers with a Java Virtual Machine<sup>85</sup> (JVM) compiler. This enables the Java program to run but technically, the compiler executes Java byte code into native machine code. The process of data compression is inevitable.

*Thousand Questions* runs in real time, querying data that flows into the program from Twitter continuously. The run-time environment handles different storage locations. Although the size of variables and program instructions are known in advance, the actual variable (the data) and the maintenance of a 'pointer'<sup>86</sup> is only known in run-time. This memory storage organisation requires data to be converted from human understandable symbols to binary digits, where the machine writes into memory and reads from stored memory. Such writing and reading activities involve algorithmic information in which a piece of information (in the form of binary digits) is encoded in an optimised way (Machta, 1999, p. 1040). Following Chaitin's example in understanding randomness in the realm of algorithmic information, it may be said that randomness is exhibited at the most basic

---

<sup>84</sup> Java byte code is an intermediary between the source code and the machine code. This Java byte code is converted by an interpreter whereby it can be run on any computer with the Java runtime environment installed on it. The runtime environment consists of a virtual machine and its supporting code. See: <http://www.codeproject.com/Articles/30422/How-the-Java-Virtual-Machine-JVM-Works>

<sup>85</sup> The use of the word virtual refers a machine does not actually exist, but acts like a machine that is operated based on the computer architecture and functions of a real computer. A JVM is a software that can process instructions in the form of a certain machine language calls Java byte code. *Thousand Questions* is written in Java using the software calls Processing. Running processing sketch requires Java Run Time, which is also broadly called Java Virtual Machine. JVM is part of the Processing software package. See:

<https://github.com/processing/processing/wiki/Supported%20Platforms>.

<sup>86</sup> In computer memory organisation, a pointer does not store value, but stores a reference to another value instead. See Parlante (Parlante, 1999).

level in *Thousand Question*: the computation of a string of binary digits, of zeros and ones. From an algorithmic information theory point of view, if the string of binary digits is patternless it cannot be compressed and is regarded as random.

Unpredictability is immanent in programming as described by Parisi in *Contagious Architecture* (2013). For her, algorithms do not only mean “a set of finite instructions” (2013, p. 10) but also “are designed to select, recombine, and mutate data” (2013, p. 272). Therefore, the queried data from *Thousand Questions* is what Parisi might describe as “evolving data”—“able to adapt and to vary unpredictability according to external stimuli” (2013, p. 10). But this adaptation is endless, because querying data is a continuous process without a definite end, unless someone stops the program or the program encounters an error during run-time that blocks its continuation. Influenced by Chaitin’s algorithmic information theory on randomness, Parisi argues that “[r]andomness has become the condition of programming culture” (2013, p. ix) which is the core of computation in ubiquitous urban infrastructures and technological networks. Randomness corresponds to infinite volumes of data in contemporary software culture and the random quantities included in algorithmic decisions that compute the compression of data in the form of binary digits. Additionally, such a notion of randomness and infinite volumes of data lead to what Parisi calls “unpredictable variabilities” (2013, p. 12). The continuous running of query and the inter-actions of data, databases and software platforms account for these unpredictable variabilities that go beyond the semantic aspect of data, and to understand data in a deep, dynamic and structural way.

In their later article, Parisi and M. Beatrice Fazi continue the discussion of Chaitin’s algorithmic randomness, highlighting the fact that computation is always unpredictable and computational processing does not always lead to a “pre-programmed result” (Parisi & Fazi, 2014, p. 118). As Chaitin demonstrates, the input and output of data can be different with “an entropic transformation of data,” where the processed output shows differences from input instructions, that the output is bigger than the input

(Parisi & Fazi, 2014, p. 118). This resonates with Shannon’s information theory in measuring the probabilities of a sender and a receiver’s message differences within a channel transmission. What counts as unpredictable, according to Parisi and Fazi (and drawing upon Shannon and Chaitin) is the “increasing yet unknown quantities of data that characterize information processing” (2014, p. 118).

Code inter-acts in many different layers of computation. This section shows that the inter-actions take place at the deep structural level of communication channel transmission, data compression and compilation. Instructions are executed and infinite data are processed in the run-time environment, in which randomness is inscribed at the extreme deep level of the computation of infinite binary strings. New ‘questions’ are constantly queried from the Twitter network in the work of *Thousand Questions*, transmitting, compressing, computing, writing and reading the binary strings that generate immanent unpredictable variabilities.

### 3.4.3 Operators<sup>87</sup>

This section moves beyond data transmission and data compression, focusing on a particular process of query execution. At the material level, and as discussed earlier, a query employs set-like operations to link or to group data together. In this way, query is about bringing this relation to the fore. In this section, I discuss how data brings things into relation using the case of live queries in *Thousand Questions*. As explained earlier, the ‘SELECT’ command is one of the most frequently used query statements, therefore, the discussion mainly emphasises data selection and retrieval and not so much on data update or deletion through code. Such operations and relations are important to understand how data is returned and inter-acted with differently, and further unfold the unpredictability of data relations.

---

<sup>87</sup> An earlier version of this section has been published as ‘Interfacing with questions: The unpredictability of live queries in the work of *Thousand Questions*’ in the International Conference on Live Interfaces (Soon, 2016a)



combination of words and characters that request Twitter to filter specific words and characters out<sup>89</sup> from its database search. For example, if one wants the returned tweets without any retweeted<sup>90</sup> content, then the syntax ‘-RT’ indicates the removal of retweets. In other words, the mathematical operators can be said to play an important role in the inclusion and exclusion of data, identifying what data should be grouped together or otherwise. By having the mandatory parameter of ‘query’ or ‘q’ (as indicated in Figure 3.13 and 3.14) together with the ‘=’ operator and the list of values (the words and characters) constitute a query instruction to Twitter.

Additionally, complexity increases by having the ‘+’ and ‘-’ operators which indicate the additional words to be paid attention to. The operator ‘+’ refers to adding different words, while the operator ‘-’ refers to removing certain words. The two seem to contradict each other but function quite differently. The operator ‘+’ is also used to separate different words, such that Twitter knows what the words are that it has to pay attention to. Since some of the words are to be removed from tweets, as per the program design, therefore the operator ‘-’ is used instead to signal the function of removal. These are all complying with the query operators that are specified in the Twitter specification.<sup>91</sup> To put simply, a query, such as ‘?+hello+-world’ means to search for tweets with a question mark and the word ‘hello,’ but remove the word ‘world.’ The query parameter and the corresponding values are fixed, meaning that the query is executed with the same requirement and request logic every time. Although the condition and the operators used are the same, the result of the query execution events is unpredictable—results are different and are subjected to what data is available at both the current and recent moment. Twitter is a relatively open system in a sense there are always *random events* that intervene in the output of the result. The mathematical operators create a relation that is based on the query of random events.

---

<sup>89</sup> The blurred parts of Figure 3.13 and 3.14 are the words about racial slurs, incitements of racism and sexual violence. In the work of *Thousand Questions*, we have filtered out a list of these words.

<sup>90</sup> Retweet refers to reposting or forwarding of a message. To filter out retweets in *Thousand Questions* is to minimize duplicated content.

<sup>91</sup> See <https://dev.twitter.com/rest/public/search>.

There are other influencing factors that also constitute the indexing algorithm and sorting of Twitter’s database. Although Twitter does not publish this information<sup>92</sup> or its implementation logic, operators arguably, contribute to the relation and grouping of data. According to Twitter web API specification,<sup>93</sup> if ‘recent tweets’ are requested, Twitter will then return current data and data from the past seven days. The past seven days logic is implemented at Twitter side and it is considered as ‘recent’ that mixes with the current one. Considering ‘seven days’ to be one of the conditions for API processing, the ‘day’ criteria is part of the algorithmic logic that filters out which data is stored beyond that. To implement the condition of ‘the past seven days,’ the machine has no idea what the past seven days means logically and mathematically unless an instruction states to subtract the current date. Such subtracted data defines the scope of the date, thereby the date parameter is within a specific range for query processing on Twitter. As such, other mathematical operators may also be used to specify certain criteria. A case in point is relational operators, including ‘==’, ‘>’, ‘<’, ‘!=’, ‘>=’ or ‘<=’ that stand for “equality,” “greater than,” “less than,” “inequality,” “greater than or equal to,” and “less than or equal to” respectively. They are called relational operators<sup>94</sup> because there is always a relation—a comparison—between two entities (Meysenburg, 2014, pp. 44-5). By using different operators, the algorithm is able to act—exclude, specify and sort data—in different ways and hence, directly impact what data to process. The query with the operators is, therefore, a site of control, of restriction.

Indeed, the logic of the ‘past seven days’ is just part of many other blackboxed criteria that remain unknown to the public. But for any criteria in a query, using different operators for data selection are inevitable. More importantly, the same operators bring different data into relation for every

---

<sup>92</sup> We have made a request to Twitter about the handling of Twitter API through Twitter developer forum, however, according to Twitter they are unable share the implementation logic. See last section for the dialogues with the Twitter staff.

<sup>93</sup> Thousand Questions uses REST API to search for specific data, the recent tweets. See <https://dev.twitter.com/rest/public/search>.

<sup>94</sup> See [https://en.wikipedia.org/wiki/Relational\\_operator](https://en.wikipedia.org/wiki/Relational_operator).

query execution as in the case of *Thousand Questions*. While running the same program pre-programmed query is executed to fetch new data that matches the stated criteria. Thus, the output data, as mediated by text and voice, is presented as a snapshot of querying the database of Twitter. Although the query execution is deterministic for every computer iteration, there is a “constant injection” (Hayles, 1990, p.159-160) of new data into the Twitter database that changes the system dynamics.

Hayles observes information is interwoven with technologies and social landscapes (1990, p. xiii). By drawing upon physicist Robert Shaw, she discusses a chaotic model in which data is added from an external input as ‘information.’ In physical systems, such external inputs could be thought of being like heat—something that produces “random fluctuation” (Hayles, 1990, p. 159-60) that constitutes a complex system. In live queries, random events may be understood as random fluctuations too, in which fluctuations/events exist at the “microscopic” level that leads to “macroscopic chaos” (Hayles, 1990, p. 160). Such a specific adaptation of random fluctuation from physics to social media and an individual tweet is to make an analogy, adding up all the microscopic events, the constant injection of new information as tweets effects the macroscopic system. These amplifying fluctuations reconfigure the processing of data by taking scale into account resulting in the macroscopic chaos of Twitter as a complex system. According to Hayles, chaos emphasises that “couplings between levels are complex and unpredictable.”

One of the important concepts about a chaotic system is its ability to scale whilst retaining the same properties at all levels. Fractal geometry in mathematics, for example, demonstrates the complex relationship between microscopic parts and the whole which share the same algorithms that generate fractals, the “complex forms characterized by multiple or infinite levels of self-similarity” (Hayles, 1990, p. 288). This scaling level demonstrates the incremental difference that “shifts the focus to complex irregular forms” (Hayles, 1990, p. 210). Each level is inter-related and together shape the becoming of unanswered queries in *Thousand Questions*.



Considering how live queries are conceived in terms of fractal geometry shows that each iteration of query execution shares the same deterministic properties, such as the operators and requested parameters. Operators bring data together by restricting and specifying things; hence a new set of returned data would form a new relation. Yet scaling in query execution does not mean exactly the same as fractal geometry in physical science, but rather to expand and take into consideration how the world is represented at multiple scales. As Hayles too explains, the world “is rich in unpredictable evolutions, full of complex forms and turbulent flows, characterized by nonlinear relations between cause and effects, and fractured into multiple-length scales” (1991, p. 8). This implies that unpredictable forces are exhibited at multiple scales.

The relation of queried data is temporal. There are many queries executed in every moment of time, and a query does not necessarily contain a relation with another query request, yet a particular pool of data only responds to the specific query request. When executing the same query again (even in a very short turn around time), another pool of data comes into relation and that is subjected to the real-time conditions. Live queries comprise operators that act to establish a data relation through query execution. The combination of data and its relations are only specific to a particular query at a moment of time. The results of live queries are therefore something that cannot be repeatedly generated.

In physical science, fractal geometry for example, we understood the relationship between chaos and unpredictability where simple deterministic systems can possibly produce unpredictable results. In contemporary software culture, live queries of a data stream are conceived as unpredictable, insofar as data is understood as random events from a system point of view. Microscopic fluctuations are entangled with macroscopic chaos, consisting of operators that act upon and beyond a chaotic system through code inter-actions.

Fundamentally, the query's operators play a key role in manipulating what is to be seen and heard through the process of specifying, sorting and excluding data. In the context of social media, user information becomes increasingly valuable for targeted marketing. In the examples of online registration it is observed that many fields are set from optional to mandatory for data collection. A case in point is Rena Bivens's research on the Facebook platform, wherein, since 2008, the gender field became mandatory with only the option of selecting male and female (2015, p. 9). It was not until 2014 that Facebook introduced over 51 gender options but these choices are restricted to users in certain countries. With this valuable data, marketers can formulate targeted marketing strategies and analyse corresponding users' behaviours through the act of query execution.

This 'queer query' might even have its historical root in *loveletters*, which was developed by queer scholars Strachey and Turing (Gaboury, 2013, n.p). Executing query may be thought of in terms of queering "the nature of identity" (Barad, 2011, p. 26) through the act of execution. The historical work *loveletters* is important in countering stereotypes at that point in time. It continuously influences other practitioners and projects, such as our project *Thousand Questions* which utilises the queer query. In other words, executing query may be also understood as executing que(e)ries in many software platforms, which does not only mean to queer gender but also other parameters, such as occupation, countries of origin, interests and many others whereby an identity is (re)derived and (re)defined across time. Mathematical operators, therefore, are considered as an essential part of a queer query.

### **3.5 Inexecutable query in closed platforms**

As demonstrated in the previous section, by using various operators one can add more constraints when extracting the data. In handling a query request, Twitter also needs to process the query through communicating with the database privately using various operators. The operators encompass private logics, business decisions and regulations and any forms of query

(both public and private) should, according to Bucher, be understood as “a management style, a technique for governing the relations it contains” (2013, n.p). Twitter, like any other social media platform exercises control through the act of query execution. From a developer’s perspective, there is no way to reveal the detailed algorithmic logics of the web APIs. It is like a blackbox, in which Twitter takes in a query request and returns a list of data whereby the process between the input and output of data query is rather opaque. Twitter does not publish any source code for its platform, nor details of its infrastructure, architecture and system. Therefore, there is no way for developers to participate in and to understand the operations of the Twitter platform. Although APIs offer the possibility of and accessibility to data query and Twitter has provided a degree of openness with its comprehensive guide<sup>95</sup> for developers, containing customer service, blogs, conferences, forum and documentation that facilitate its usage, Twitter remains a closed platform from the perspective of artists, developers or researchers.

In contrast to open source platforms, developers cannot participate in decision making raising concerns about software features and technical implementation. In addition, one cannot modify the code and create the software, or the platform, together. Closedness is used here to refer to the inability to participate, create and modify software under the software development life cycle. The platform is closed to the extent that “restrictions are placed on participation in its development, commercialization or use” (Eisenmann et al., 2008, p. 1). The visibility of the platform is opaque as there are different types of restrictions, or limitations upon use. According to the scholar of technology studies, Jenna Burrell, this opaqueness exists because of “proprietary concerns,” and she explains, “They are closed in order to maintain competitive advantage and/or to keep a few steps ahead of adversaries” (Burrell, 2016, p. 3). This is also how professor of information studies, Christopher M. Kelty, describes a closed system as more a proprietary system (Kelty, 2008, p. 143). Giant proprietary platforms,

---

<sup>95</sup> See <https://dev.twitter.com/>

Twitter and Facebook for example, host enormous amounts of public data but how they manipulate, process and present this data to us is highly closed.

In *Thousand Questions* the query requests 50 tweets that are regarded as 'recent,' the only condition that is transparent is the period of 'the past seven days.' Imagining tweets are intensively stored and updated, what kind of tweets have higher priority than other available tweets that are also stored within the past seven days? What are the decisions that influence the priority? The closed nature of the system makes it impossible to know the answers to these and similar questions. Given that there are many applications, today research and analysis relies on these platforms as they, in some respects, "organize and model the future" (Day cited in Bucher, 2013, n.p) and this closedness has an important implication. The ecology of software is manipulated by big players and power is indeed centralised, as such, and according to Bucher, the "potentiality or openness towards the future is highly controlled" (2013).

For query data developers have to fully comply with all the standards and specifications even though there are many changes over time that impact upon those artworks and industry apps that require the Twitter web API to function. *Thousand Questions* is one such artefacts. To put it simply, if the web API is changed, upgraded or terminated *Thousand Questions* will cease to function and so too will other works that rely on Twitter.

In fact we can observe many similar software update cases in contemporary culture as, for instance, new updates of operating systems, applications, servers and databases to name a few. A company usually supports legacy software for a certain period of time but ultimately it has to be made obsolete. The operating system Windows 95 is a case in point. It was released in 1995 by Microsoft but the corresponding support ended in 2001 ("Windows 95," 2001). Similarly a particular format of a web API can be made obsolete in time. In the case of Twitter, it once had a major change of the web API from v1 to ver1.1 in 2013, providing a 6-month grace period.

The new version aimed to fully replace the old one (Espinha et al., 2014, p. 88). *Thousand Questions* was developed in 2012, initially using v1 but after 2013 it was changed to ver1.1 for its second public exhibition. Without knowing the API retirement plans it came as a surprise to experience that the artwork did not perform as expected. If *Thousand Questions* were not upgraded to use the web API ver1.1 the query would not be executed.

Using an old and obsolete version of the web API meant that the syntax and logic were outdated. Any thing wrong (in logic or syntax) would result in errors and hence the whole program would stop. In the context of live queries, disruptions thus can be thought of inexecutability—the malfunctioning of queries and this requires articulating in relation to closedness and controlling beyond the technical. In the previous chapter, the notion of disruption has been discussed through Gadassik's 'bodily disruption' (2010, p. 129) and Doane's disruption of continuity as the forces of deadness. In addition to that, inexecutability is more politically and culturally oriented within a closed system, where control is exercised through the entanglement of humans and nonhumans resulting in the unpredictability of the inexecutable. Unlike the unexpected catastrophes and other external events, disruption can be a centralised and planned decision where the unpredictable disruption is experienced through running the query object which will not work as expected. It appears as a technical problem with errors encountered but requires a more complex understanding of the notion of inexecutability.

Although unpredictable machine behaviour is encountered when errors occur at a particular moment, it can be further articulated across time. Within the specific situation of software updates, there are new features that come with the new patch while at the same time, some past features are made obsolete. This implies the possibility for forthcoming disruptions. Therefore any upgrade or update includes not only the debut of new features but also the obsolete past and the inexecutable future. The inexecutable query is conditioned by the new update and this is a relational change. The change is related to the past features, syntax and policies. Features cannot

be just understood as pure technical enhancement but rather as indicators of the change of the companies' direction, its business decisions, market forces and technological infrastructure. As such, inexecutability should be understood beyond the technical as forces which include a social and political dimension.

To elaborate further in support of my argument regarding the social-political aspects of inexecutability we have to examine the change of the Twitter web API from 2012 (ver1) to 2013 (ver1.1). The change of API, according to Twitter, includes more rigorous authentication procedures, a lower rate limit and stricter developer rules. These are all measures for better tracking and more control that align with the company's overall direction.<sup>96</sup> As verified by media studies scholar Robert W. Gehl, "Twitter once had a very open API," allowing anyone to easily access its data but the policy has changed since 2012 (2015, n.p). In addition, by using the Twitter API, one agrees to the terms of services which restrict<sup>97</sup> the development of third party applications (Gehl, 2015, n.p). Considering "code is law" (Lessig, 2006, p. 5), such measures serve to regulate query processing which is implemented at the level of code. The upgrade is not necessarily a means of offering more advanced features, it can be perceived as a disruption because it *operates* at high levels of unpredictability, uncontrollability and unknowability across time in which a query is made inexecutable.

In 2015, the Netherlands-based artist duo, Esther Polak and Ivan Van Bakkum (also known as PolakVanBakkum), developed a project called *Techno Mourning*.<sup>98</sup> It is a video that was made directly using Google Earth, documenting 3D graphics made using the Google Earth API but at the same time announcing that the project would be no longer functioning by 2016.

---

<sup>96</sup> First of all, Twitter requires knowing who accesses and uses the query. Second, a rate limit is imposed to "prevent abuse of Twitter's resources". Finally, consistent experience is something Twitter wants to ensure. See <https://blog.twitter.com/2012/changes-coming-in-version-11-of-the-twitter-api>

<sup>97</sup> This refers to Robert W. Gehl's article that shows Twitter's intention behind the upgrade of API, which is to centralize their content (2015, n.p). The new policy of having an "API key" is to keep track of the third party applications' development.

<sup>98</sup> See: <http://www.250miles.net/techno-mourning/>

Google had announced the deprecation<sup>99</sup> of its Google Earth API as early as December 12, 2014. After this time it did not upgrade anything and the announcement literally states that the API would “shut down” in 2016 (Google, 2016). There would be no alternative or upgraded solution to develop any 3D rendered artefacts that are based on Google Earth—the virtual globe, map and geographical information program. PolakVanBekum also documents the list of artistic projects that require the Google Earth API to function. In the way the artist group raises awareness of the ever-changing technological landscape, where things can be stopped at any time and there is no promise of everlasting and workable digital projects. There are relations and entanglements in the practice of software and one cannot see digital objects as something standalone, sustainable or even static.

Similarly, the artwork *Net.Art Generator*, mentioned in this and the earlier chapter, could not escape from this notion of inexecutability. Sollfrank, explains how the whole program of *Net.Art Generator* has stopped functioning because of the sudden and seamless change of the Google search query (personal communication, October 28, 2015). She did not receive any announcement from Google in advance about the change and has put up a webpage stating that the work “is currently not operational” (see Figure 3.17). From the artworks that have been mentioned, there are different decisions that are inscribed at the code level, insofar as the user, developer and artist have no way to participate in the development process of web APIs. Therefore, the notion of inexecutable query is also a critique to closed and proprietary platforms that centralised its decision-making in a black box.

---

<sup>99</sup> See <https://developers.google.com/earth/>

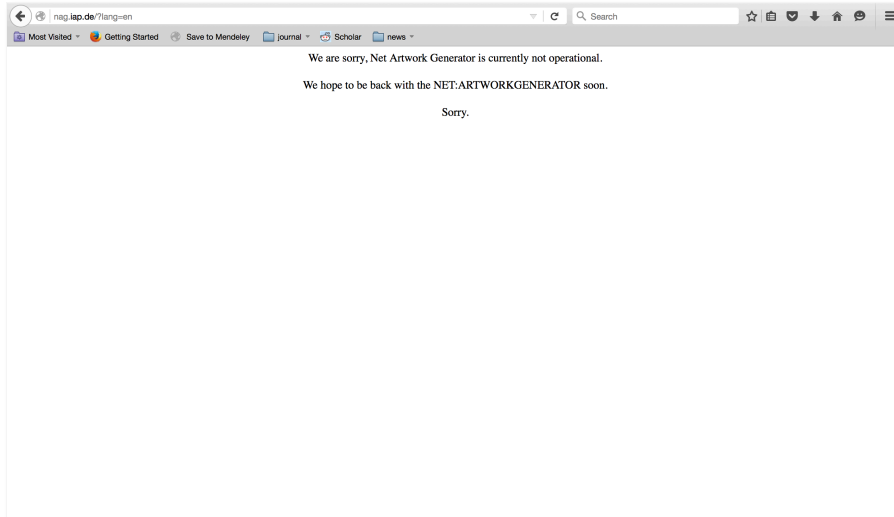


Figure 3.17: A screen shot of the error page of *Net.Art Generator* (1997) that was captured on January 14<sup>th</sup>, 2016. Retrieved from <http://nag.iap.de/>

The inexecutable query does not simply mean it is malfunctioning. Instead, it describes non-neutral code that is inscribed with different business logics, political decisions and software practices. The inexecutable query impacts different software (art) practices that require coping with changes of code, formats and policies as performative effects (see Mackenzie’s discussion in Chapter 2.2.2). The art projects mentioned above are just two examples of many other software projects that rely on third-party APIs. Moreover, the notion of inexecutability allows for critical attention and reflection on the dynamic qualities and unpredictable consequences of technology, Berry reminds us “to stop thinking about the digital as something static and object-like and instead consider its trajectories” (2014, p. 1). In other words, things are constantly changing in contemporary software culture. Query is just one of the many digital objects that emphasise this point. Interestingly, Twitter suggests one has to be looking for change proactively, as it is written on its development site:

The Twitter platform is constantly evolving and there is frequent change. If you have an integration that’s no longer working, be sure and review the [different Twitter supporting sites] (Twitter, n.d).



From theoretical understandings to industry practices, we are always being reminded of the dynamics of code and the possibility of inexecution. Seemingly what makes digital objects live is not the disruptive moment where things do not function and queries are inexecutable but rather it is the possibility of inexecution at any time in contemporary (update) culture. In other words, unpredictable forces are entangled with live queries.

Beyond the possibility of inexecution, it is worth noting that even when a query is technically well structured, there is still unpredictable force upon its realisation. Software studies scholar Federica Frabetti traces the development of software engineering in the 1970s where industrial production schedule usually lagged behind an estimated plan. Drawing on the work of Fred Brooks, who was a project manager in IBM, Frabetti explains that programmers found it impossible to estimate the unpredictable consequences of change and calculate the possible consequences of software development (2015, p. 103). She identifies a gap between “conception” and “realization” in which software is a product as an ideal concept. However, software is also a “process” of software development that requires the understanding of system consistency (Frabetti, 2015, p. 104-5), alluding to encountering of wider assemblages that enable a piece of software to run. Most importantly, in many situations problems “are hard to detect in the test laboratory” and it is not until something is realised in a “real environment” that software performs in an unexpected way, as the actual situations cannot be simulated fully in the lab (Frabetti, 2015, p. 118). Such a situation is unpredictable and it is difficult to predict all the cases from designing the concept to implementing the software. Frabetti argues that software is predictable—what she calls “predictability fail”—where it fails predictably “in unpredictable ways” (2015, p. 114).

Following Frabetti’s notion of software realisation in a real environment, queries also perform in unpredictable ways. Things might be operating intrinsically and unnoticeably beyond human perception as, for example,

with micro-time execution. When five queries are run<sup>100</sup> in a real environment sequentially, all having the same query statement (requirements and parameters from *Thousand Questions*), the execution time that Twitter reports back is different: 0.246s, 0.092s, 0.085s, 0.074s and 0.315s. The real environment of query execution is indeed distributed and networked, yet it is also highly unpredictable. One of the reasons for the micro-time differences may be due to many other queries running at the same time and Twitter is dynamically updating the database from end users input to many queries' execution simultaneously. Many other issues, from hardware and software to data traffic, impact the actual performance of Twitter. The realisation of data query constitutes the unpredictability of execution insofar there is no way to predict the actual performance of code execution.

In conclusion, this chapter has investigated the unpredictability of queries by analysing Twitter as a complex platform and the corresponding API that is operatively and technically used in *Thousand Questions*. The reflexive coding approach demonstrates how code inter-acts unpredictably at multiple scales from the dynamic Twitter platform, to the noise, entropy and randomness in digital processing, to the data processing of mathematical operators and to the realisation of query execution. Executing queries is a cultural practice that is widely used in both artistic projects and industrial applications in contemporary software culture. By paying attention to execution this chapter has also articulated the possibility of what I call inexecutable query—that is not only technically inexecutable but which also exhibits social and political forces that are inscribed at the level of code.

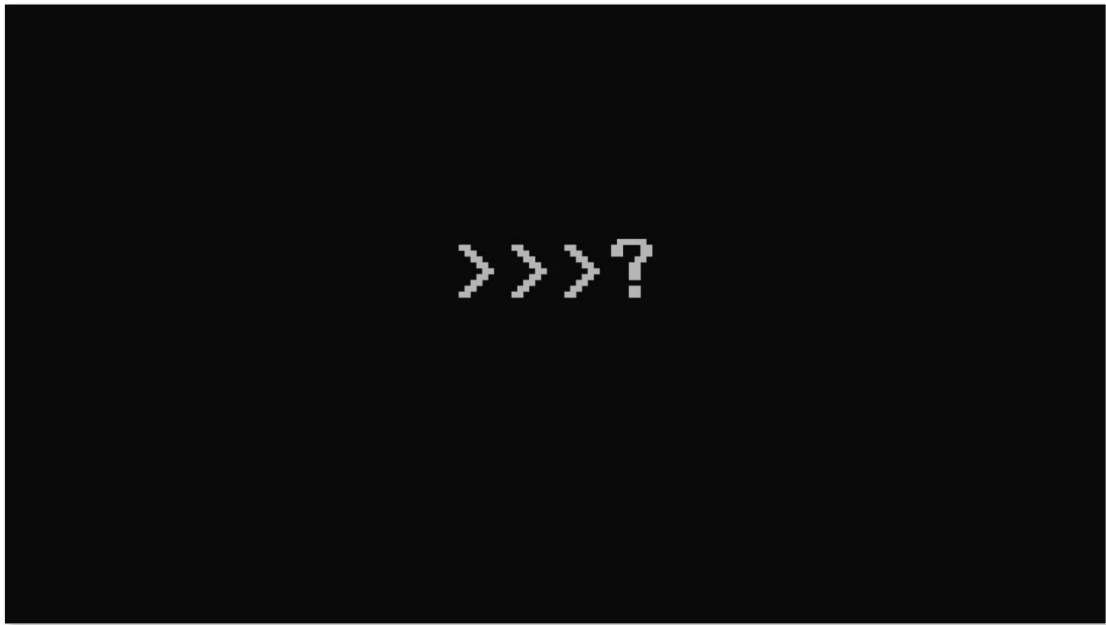
For the argument of this thesis liveness can be understood through this material account of unpredictability. Examining this vector through Wardrip-Fruin's notion of unpredictable manifestation, this chapter expands the concept by considering technological platforms and networks, as well as deep data structure and query processing. It demonstrates the entangled

---

<sup>100</sup> The tests were conducted on March 23, 2016, between 11:45-12:00.

inter-actions between code, data, databases and transmission processes with a specific focus on openness and closedness of the Twitter system. It argues that there are different forces that constitute the notion of unpredictability, in which executing queries unfolds some of the complexities of code interactions, exposing both the technical and cultural relationships of live queries. The next chapter will continue to investigate live processes, exploring another vector of liveness—micro-temporality—that is also essential to understand the entanglements of contemporary software culture at multiple scales.

### **3.6 Thousand Questions**



*Figure 3.18: Thousand Questions (2012-2016)*

By contrast to “hypothesis-led” practice, the artwork *Thousand Questions* is made using a “discovery-led” process. This means that the creation process, in part, is based on the intuition and interest of the artist (Borgdorff, 2011, p. 56). With a background in information systems, I have always been interested in the process of data scraping, querying and extraction. Therefore, when I first experienced the artwork *Listening Post* (as discussed in Chapter 1) and understood its collection of large amounts of online data that is updated in real-time, I was drawn into the data scraping and extraction processes that were configured in the background. A dialogue with one of the artists, Mark Hansen, revealed the not too surprising fact that *Listening Post* was designed in the pre-API era (see below email communication).

hi

the piece was designed in a time pre-api's and pre-rssfeeds. then we scraped bulletin boards (fetching html programmatically and parsing out the content) and spidered irc networks. again, the piece was designed before the era of web services...

M.

(M. Hansen, personal communication, July 6, 2011)

Recognising the offering of APIs is a common practice in the industry, I wondered how APIs could efficiently process data that use a different approach from *Listening Post*. My previous experience and knowledge in information processing and database management influenced the subject matter—data query—of this chapter. Borgdorff’s reflexive practice has been useful to me in understanding more about how to account for such influences, in particular his discussion of artists’ tacit knowledge. He says,

the artist’s tacit understandings and her accumulated experience, expertise and sensitivity in exploring uncharted territory are more crucial in identifying challenges and solutions than an ability to delimit the study and put research questions into words at an early stage (Borgdorff, 2011, p. 56).

Therefore, my tacit knowledge, experience and understanding of data processing also influence the direction and process of artefact that I produce. This is a self-reflexive process because the inquiry process, in part, “is directed by personal interest and creative

insight” (Sullivan, 2010, p. 110). The focus on data querying became the subject of interest through which to explore the notion of liveness in the work of *Thousand Questions*.

Through the process of creating *Thousand Questions*, I became dedicated to researching and investigating how APIs operate and to thinking through why this has become a cultural practice through many platforms and technology service providers have offered their APIs. According to Borgdorff, a creation process contributes to what we ‘know’ and ‘understand’ (2011, p. 54) and it is similar to artist Natalie Jeremijenko description of how “we think with things” (Hertz, 2012). In other words, experimenting and investigating APIs allows me to think in wider computational ecologies and understand how APIs are operated, such as how they are offered by companies, used by developers and communicated between code and machines. Some of the thinking and understanding has been synthesised and expressed in the earlier sections of this written thesis.

*Thousand Questions* was first developed in 2012, in collaboration with British artist Helen Pritchard, exploring the notion of liveness as the first project as part of this thesis development. It was first exhibited in Microwave International New Media Arts Festival in Hong Kong (see Figure 3.19).

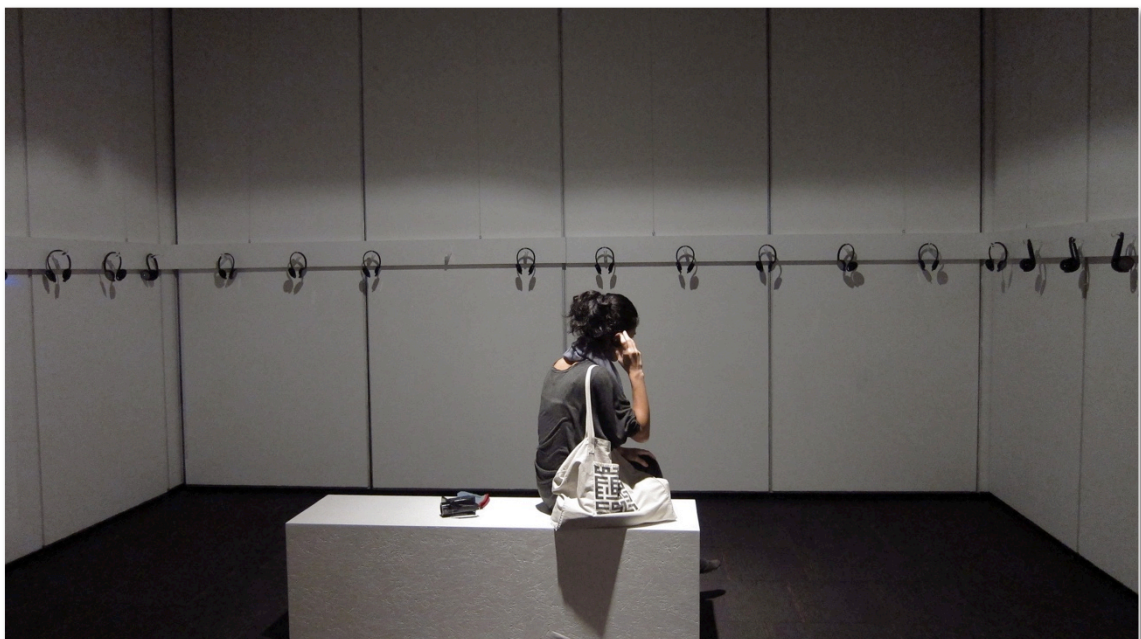


Figure 3.19: *Thousand Questions* in Hong Kong (2012)

Extending Sullivan's discussion on using the method of visual strategies in reflexive practice (2010, p. 108), the first installation was presented with an audio arrangement, containing a series of Bluetooth headphones that through which visitors hear the installation speak the tweets. This multiple arrangement of audio devices embodied meanings, allowing the mix of the live and archived tweets to enter the gallery. Reflecting on the employment of this audio strategy, I noted that it was a way of 'thinking with things,' in this case data and headphones arrangement, to question the distinction between live and recycled data, and that it informed my understanding of liveness in digital media as opposed to other media forms such as television.

While the work was executed in the gallery, it underwent reflexive cycle of processes that involve creation and critique. According to Sullivan, an artwork's structure embeds various dimensions of theory, in particular to the dimension of theory: Create-Critique, alluding to "theoretical interests are investigated through a cycle of processes involving creating and critiquing" (Sullivan, 2010, p. 106). In the exhibition, two headphones were connected to live queries to create new audio tweets that mixed with the other archived data that were presented through the remaining headphones. In addition to the *creation* of tweets, the work also involved a *critique* on the possible "illusion of liveness" in digital media (McPherson, 2006, p. 202). Even though live queries were executed in real-time in the gallery space, this was a mere recycling of data that was framed as live. This recycling of data pointed at both the live and archived tweets that were mediated in an audio form. Therefore, the artwork *Thousand Questions* encompasses Sullivan's understanding of the Create-Critique dimension (2010, p. 106) through running and executing code in which the creation and critique were held together simultaneously.

Back then in 2012, I was still examining the notion of liveness in relation to the immersive and bodily experience of how an audience may feel live. This was inspired by the artwork *Listening Post* as well as by my textual analysis of different perspectives on liveness (see Chapter 1, sections 1.1 and 1.3). Therefore, in addition to the arrangement of headphones, my collaborator and I employed an audio strategy by which the synthesised voice (text-to-speech) was made to speak with special audio effects such as 'Detune' and 'Reverb' with the help of the software called AudioHijack (see Figure 3.20).

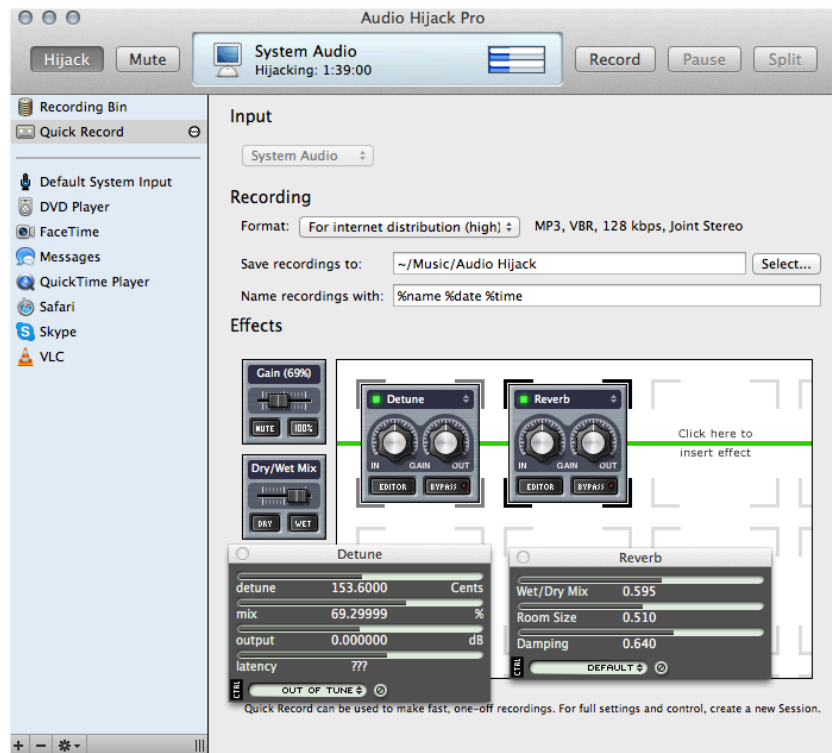


Figure 3.20: Audio effects in *Thousand Questions* (2012)

At the first public exhibition of *Thousand Questions* we considered both human and nonhuman activities to explore how a query is performed in real-time. This is reflected in the first version of our artist statement:

The network asks us 'If I wrote you a love letter would you write back? (and thousands of other questions) in this work. Computational code draws thousands of questions from online matter into the exhibition space. The questions are gathered in real-time from the social media site Twitter and encoded to speech. Listening is a form of decoding, and in this work the machine constantly undergoes the process of editing, encoding and decoding texts. What happens when written texts of the network are converted to speech? How does it feel to listen to the questions of a machine, when these questions are our familiar tweets?

The computerised voice the audience hears is a collective one, one of both the human, non-human and distributed. The audio stream frames (if only for a moment) the chaos of matter in the network as the activity of 'query' is



performed across a collective of humans and non-humans.

The sequence of questions is open to change by audience's actions. As headphones are picked up and replaced, a new sequence of texts is created - blurring the space between the live and archived text, the linear and the chaotic.

Like the love letters appeared mysteriously on the noticeboards of Manchester University's Computer Department in the 1950's. These questions continuously evolve as the machine performs the questions (perhaps an expanded Turing test) to its listeners (Soon & Pritchard, 2012a).

The last paragraph hinted at the link with the earlier work of Strachey, *LoveLetters*, and this has become the starting point for writing this chapter. Through researching the work *LoveLetters*, my interest shifted from generative algorithms and linguistic combinations to questioning the dynamics of *Thousand Questions*, further it has extended to the distributed network environment.

Through developing the subsequent chapters of this thesis, and as a result of taking up different opportunities to present the work again, we decided to get rid of the special audio effects and simply use the computerised voice as we felt it was distracting for the audience by to focus on the mediated immersive audio effects. This decision was also influenced by more thorough research that shifted my focus on liveness from the perspective of audience experience to nonhuman inter-actions. This is similar to when Bordorff suggests: "Contexts are constitutive factors" in art practice (in this case is coding practice). He continues: "Artworks and artistic actions acquire their meaning in interchange with relevant environments" (Borgdorff, 2011, p. 54). In reponse to the change of context and following our new inspirations, we developed the latest version using both visual and audio strategies, most notably an additional visual component that demonstrated the waiting of a query, putting the focus of the artwork back on the process of data querying. By taking this approach the work unfolds the unpredictability of data query in itself in a way that is more apparent. These changes of strategies and settings demonstrate the reflexive practitioner's ability to question and review the situation, context and approach (Sullivan, 2010, p. 110).

Additionally I split the sentences into individual characters with the aim of de-

emphasising the linguistic part of the returned query. The resulting visual presentation allowed the audience to experience some sense of the questions without really knowing the meaning of the combinations of characters. Our intention was that the newly configured version may offer a space to speculate on the process of data query. The overall process of using different strategies and changing the direction of the artwork through working with materials and coding is, therefore, an example of reflexive practice.

Technically, *Thousand Questions* was written in Processing, a Java-based open source software. It utilised an external library called Twitter4J<sup>101</sup> to communicate with Twitter through the Twitter API. The project has been tested, running on Processing ver. 1.5-3.1 with Twitter4J ver. 3.05. The artwork also utilised the text-to-speech feature from the native Apple Operating System (requires version 10.7 or above) with a particular Australian woman's voice, Karen. Since the project requires querying data from Twitter an Internet connection is required throughout the whole installation.

With regard to the visual design and logic of *Thousand Questions*, we used a particular nostalgic font type called 'MorePerfectDOSVGA' to simulate, and reference, the old command line operating system environment, presenting a minimalistic interface for speculation on the process of data querying. The work first starts with the posting of a query request to Twitter, going through the process of authentication and selection of data. Based on the data the program filters tweets, excluding additional spaces and special symbols like '@' sign and 'http' for showing anonymous tweets and removing URLs. Following this the tweets are further split into single characters and displayed on a screen. Finally the program calls Apple's text-to-speech function and vocalises the unanswered queries as tweets one by one. This cycle loops without an end as such.

The sample queried questions are shown below:

1. If I wrote you a love letter would you write back?
2. Do you believe we'll ever find ourselves in this position again?
3. Utterly freezing sitting in my wet swimsuit. Why do I do this to myself?
4. If you could do anything this weekend, but would need to be back at work Monday morning ready to go, what would you do?

---

<sup>101</sup> See: <http://twitter4j.org/en/index.ht>

5. The Importance of a Positive Work Environment: Are you excited to go to work everyday?
6. Ben! What in the world happened?

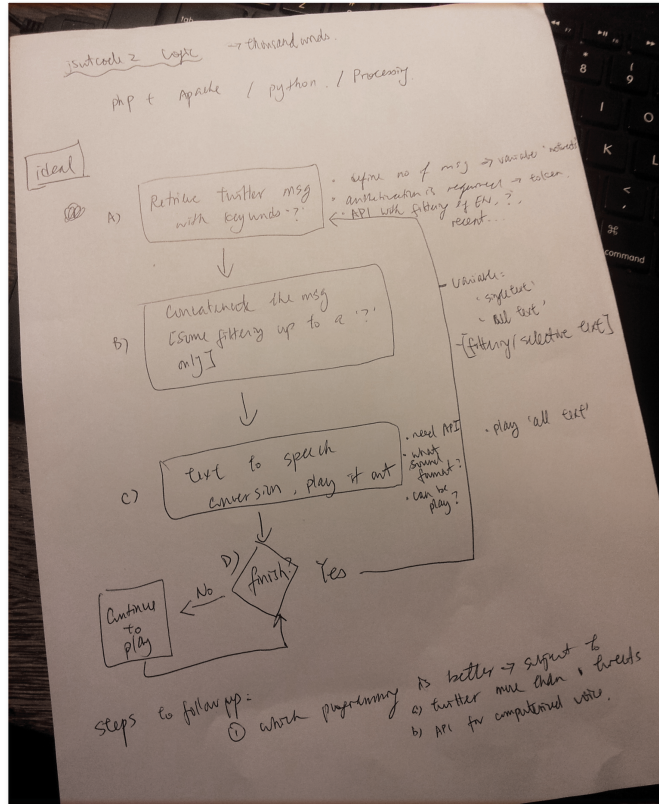


Figure 3.21: A conceptual stage, the flow chart, of *Thousand Questions* in 2012

```

21 ArrayList tweets;
22 int manyTimes= 50; //how many tweets : the rate limit is 180 per 15 mins window, around 12 sec per call.
23
24 Twitter twitter = new TwitterFactory().getInstance();
25 RequestToken requestToken;
26
27 /***** sound *****/
28 String finalMsg="";
29 int voiceIndex = 24; // final selection is 5, 20, 22 //Microwave use 24 Karen
30 int voiceSpeed = 130; //10.5.8 is not supported
31 /***** End Of sound *****/
32
33 /***** Sequence control *****/
34 boolean sayfinish = true; //make sure finish the voice and follow by tweet query
35 boolean textfinish = false; //make sure finish all text/tweet display then the voice follows
36 boolean firststart = true; //allow the first start of tweet query
37 /***** End Of sequence control *****/
38
39 void setup() {
40 //size(1024, 768);
41 fullscreen();
42 background(c1);
43 font = createFont("MorePerfectDOSVGA.ttf", 80);
44 //font = createFont("PerfectDOSVGA437.ttf", 25);
45 textFont(font);
46 frameRate(40); // the speed of character display
47 connectTwitter(); //try to connect to twitter
48 }
49
50 void draw() {
51 //logic: 1/ query tweet > 2/ text display (with split characters) > 3/ voice + text display (space) > loop 1-3
52 calltwitter();
53 }

```

Figure 3.22: An excerpt from *Thousand Questions*' source code: Setting up variables and screen show, and establishing Twitter connection

```

20 void calltwitter() {
21     //Try making the query request.
22     if ((firststart) || ((sayfinish) && (textfinish))) {
23         firststart = false;
24         try {
25             Query query = new Query(querystr);
26             query.count(manyTimes); //rpp is no longer used
27             query.setLang("en");
28             query.setResultType("Query.RECENT");
29             finalMsg = "";
30             int totalMsg = 0;
31             timenow = time();
32             println(prefixdisplay + timenow + "Query Data in process..");
33             QueryResult result = twitter.search(query); //180 rate limit, search/tweets/json
34             ArrayList tweets = (ArrayList) result.getTweets();
35             timenow = time();
36             println(prefixdisplay + timenow + "Successful in getting unanswered query.");
37             println(prefixdisplay + timenow + "Filtering tweets in process..");
38             for (int i = 0; i < tweets.size(); i++) {
39                 Status t = (Status) tweets.get(i); //Tweet class is replaced with Status
40                 User u = (User) t.getUser(); //getUser() replaces getFromUser
41                 String user = u.getName();
42                 String rawMsg = t.getText();
43                 Date d = t.getCreatedAt();
44                 String f_rawMsg = filtering(rawMsg); //call function to filter text
45                 totalMsg++;
46                 /*here can add further filtering, check the prohibited keywords, if a tweet contains
47                 the prohibited keywords (using while loop, array to go through all, split with space),
48                 then f_rawMsg="", else f_rawMsg with the tweets and append */
49                 finalMsg = finalMsg + " " + f_rawMsg; //final message > append each rawMsg with ? sentence
50                 finalMsg = finalMsg.replaceAll("\\r\\n\\t\\p{So}+", " "); //no more weird text formatting and emoticons
51             }
52             timenow = time();
53             //println(prefixdisplay + timenow + "*** There are total " + totalMsg + " unanswered queries in this batch:\n" + finalMsg);
54             println(prefixdisplay + timenow + "*** There are total " + totalMsg + " unanswered queries in this batch");
55             getindex = 0;
56         } catch (TwitterException te) {
57             timenow = time();
58             println(prefixdisplay + timenow + "Error in twitter query: " + te);
59         }
60     }
61 }

```

Figure 3.23: An excerpt from *Thousand Questions*' source code: Querying Twitter data

```

1 void loadlongstring() {
2     String longstring = finalMsg;
3     int lengthcount = longstring.length();
4     if (getindex == lengthcount) { //display waiting character when all the text is read
5         waiting();
6         textfinish = true;
7         thread("sayIt"); //in twitter tab > only have voice after all the text is run
8     } else if (getindex > lengthcount) {
9         waiting();
10        textfinish = true;
11    } else {
12        displaychar(longstring.charAt(getindex)); //display individual character from the retrieved questions
13        textfinish = false;
14    }
15    getindex++;
16 }
17
18 void displaychar(char one) {
19     fill(c1);
20     rect(0,0,width, height);
21     fill(c2);
22     textAlign(CENTER);
23     text(">>>" + one, width/2, height/2); //you can see text can be added. a text adds another text.
24 }
25
26 void waiting () {
27
28     int count = frameCount%17; //subject to the viewing experience of a "pause"
29     char wait;
30     if (count<10) {
31         wait = '_';
32         displaychar(wait);
33     } else {
34         wait = ' ';
35         displaychar(wait);
36     }
37 }

```

Figure 3.24: An excerpt from *Thousand Questions*' source code: Splitting tweets into individual characters

```

1 void sayIt(){
2     if((sayfinish) && (textfinish)) {
3         try {
4             String s = finalMsg;
5             sayfinish = false;
6             timenow = time();
7             println(prefixdisplay + timenow + "Preparing to execute a run time script (TTS) in Mac OS ...");
8             Runtime rtime = Runtime.getRuntime();
9             //Process child = rtime.exec("/usr/bin/say -v " + (voices[voiceIndex]) + " " + s); //--r only supports 10.6+ //this is for Mac 10.5.8 testing
10            Process child = rtime.exec("/usr/bin/say -r " + voiceSpeed + " -v " + (voices[voiceIndex]) + " " + s); //this is for Mac Lion 10.7+
11            timenow = time();
12            println(prefixdisplay + timenow + "Karen is reading all the live questions...");
13            println(prefixdisplay + timenow + "Waiting for Karen to finish all the speaking...");
14            child.waitFor();
15            sayfinish = true;
16        }
17        catch (Exception e) {
18            e.printStackTrace();
19        }
20    }
21 }
22 }

```

Figure 3.25: An excerpt from *Thousand Questions*' source code: Processing text-to-speech

```

49 void draw() {
50     //logic: 1/ query tweet > 2/ text display (with split characters) > 3/ voice + text display (space) > loop 1-3
51     calltwitter();
52 }
53 }

```

```

>>>14:51:38 Processing authentication...connecting the Twitter platform
>>>14:51:38 Successful authentication.
>>>14:51:38 Query Data in process...
>>>14:51:39 Successful in getting unanswered query.
>>>14:51:39 Filtering tweets in process...
>>>14:51:39 *** There are total 50 unanswered queries in this batch
>>>14:52:40 Preparing to execute a run time script (TTS) in Mac OS ...
>>>14:52:40 Karen is reading all the live questions...
>>>14:52:40 Waiting for Karen to finish all the speaking...

```

Figure 3.26: An excerpt from *Thousand Questions*' log: The feedback process

Over the years, this project been developed into different versions. The work was first created and exhibited in 2012 but it changed over time in response to different contexts and due to the slightly different questions asked over the years (changes are not only in the presentation format but also the written code itself). More importantly, the changing technological landscape also forced us to update and implemented the code differently. Again, this demonstrates the process of reflexivity. The change of the code and the environment around the work suggested to us that there are different ways to understand the phenomena of liveness in new ways (Sullivan, 2010, p. 110). The artwork utilises the Twitter API but there was once a major update by Twitter that made the work inexecutable. This is illustrative of a further question around the relationship between liveness and inexecutability. The way that the piece exhibited the notion of liveness through inexecutability suggests that we may need to think differently about the notion of liveness or that we may need to think simultaneously about liveness and deadness. This informed how the notion of inexecutable query has been developed within the chapter relating to it. According to Sullivan, having the capacity to open up further dialogues and debates is an important part of reflexive practice (2010, p. 110).

This section also includes the documentation of the creation process which is in a form beyond scholarly writing and is considered as an important part of reflexive practice. Documentation can demonstrate how decisions are made with artistic reasonings as well as how the results are achieved (Borgdorff, 2011, p. 58). A blog<sup>102</sup> was setup in 2012 to document the process of conceptual development and technical exploration as well as my reflections over time. The screen shot in Figure 3.27 only shows an overview of the posts but each contains a more lengthy text. These blog posts are part of my research field journals and document, for example, different trials of text-to-speech libraries, reflections on the live presentation of the work as well as the exploration of the meaning of live data and real-time technology. Some of these have been synthesised for the sake of presenting more detailed discussion within the relevant chapters of this thesis. Figure 3.28 shows a mapping sketch on the Twitter platform and the thoughts on the Twitter API.

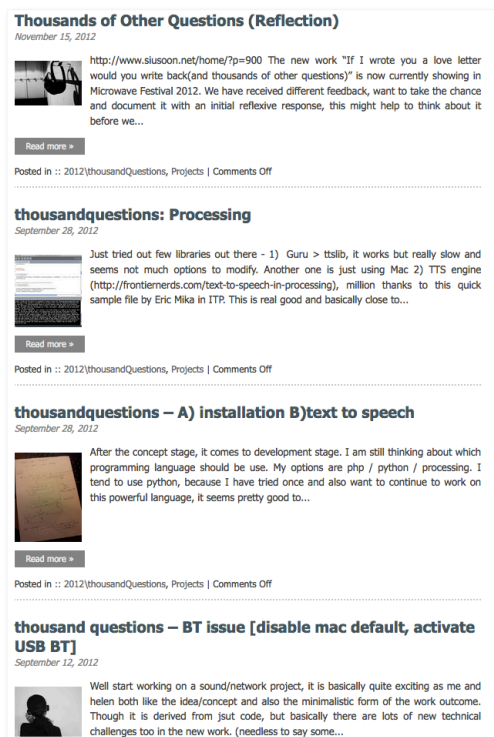


Figure 3.27: The notes of *Thousand Questions* in 2012, retrieved from: <http://www.siusoon.net/dat/category/projects/2012thousandquestions/>

<sup>102</sup> See: <http://www.siusoon.net/dat/category/projects/2012thousandquestions/>

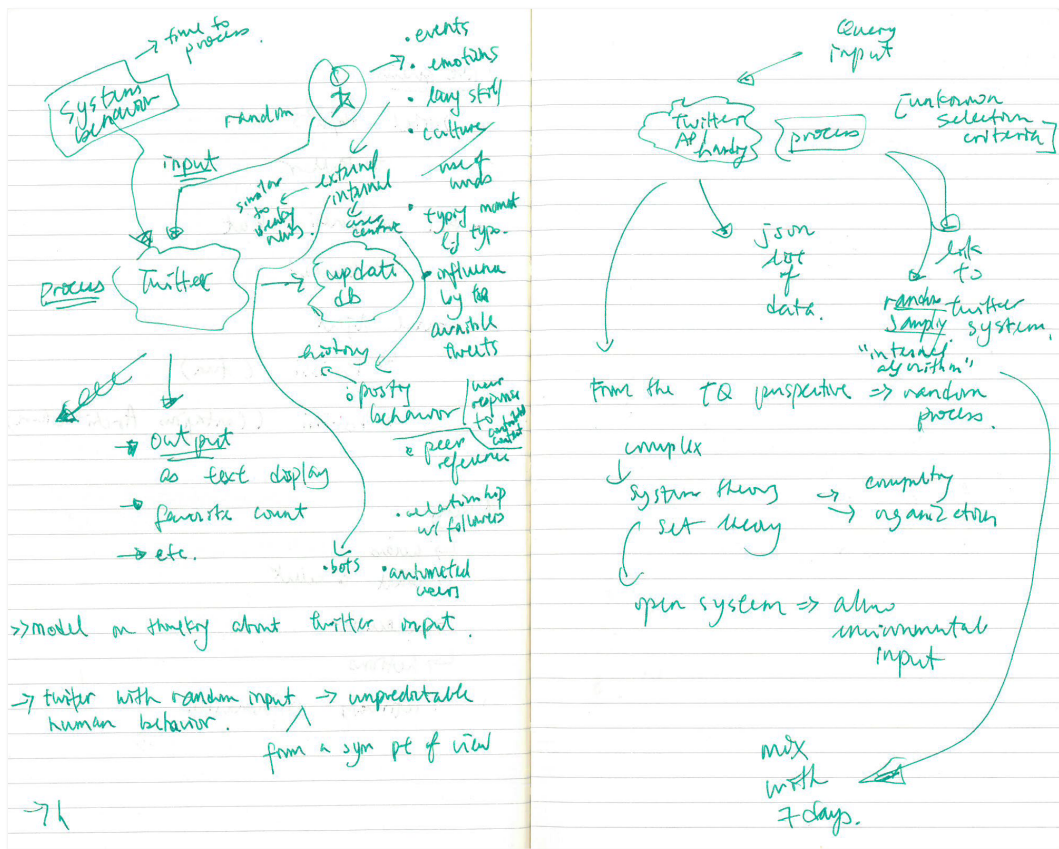


Figure 3.28: The notes of *Thousand Questions* in 2016, retrieved from: <http://www.siusoon.net/dat/category/projects/2012thousandquestions/>

In addition to the documentation through handwritten notes and blog posts, code comments were made during each “iterative trial” (Berry, 2011, 2014) for major releases. Code comments are integrated as part of the program of *Thousand Questions*, written in a grey color as shown in Figure 3.29. This special note area lists the technical parameters, outstanding actions and tested notes which document the requirements, priorities, decisions and fine-tuning of logics, bug fixes and many other factors.

```

10 //*****
11 Logic & parameter/ last updated on 13 Apr 2016:
12 *** after click the run button don't press the stop, let it run
13 1) rate: configure the content + rate + voice
14 2) querystr -> define the keyword + filtering in twitter content
15 3) keyword -> keyword
16 3) manyTimes -> define the no. of message extraction
17 4) voicespeed -> rate of the speech *doesnt work on 10.5.8
18 5) needtoFilter -> manual filter list e.g @, http
19 ///coming actions:
20 - make it into object such that there is a pause for every question
21 - change the TTS so that it won't depend on mac OS -> migrate to PI
22 *****/
23
24 //*****
25 If I wrote you a love letter would you write back? (and thousands of other questions), developed by Winnie Soon, 12 Oct 2012;
26 tinkered with by Helen Pritchard Nov 2012
27 - Test on Mac OS X 10.5.8 + Java 1.5.0.30 - Karen voice doesn't work / 10.7.5 + Java 1.6.0.37 / 10.7.0 + Java 1.6.0.37 /
28 10.9.5 + Java 1.6.0_65 / Mac 10.10.2 + Java 1.7.0_79
29 - [ver1.1] on 17 Oct, 2012
30 - [ver1.2] on 7 Nov,2012 (fix the "invoke length() on the array type string[])" -> add new karen voice in 10.7
31 - [ver1.3] on 29 Apr, 2013 - The search keywords with encode
32 - [ver1.4] on 8 Nov, 2013 - Change of Twitter4J ver and Twitter API
33 - [ver1.5] on 17 Mar, 2015 - Changed for Processing2.2.1 + filter text [tested on Mac 10.9.5 + Java 1.6.0_65, AudioHijack 2.11.4, Twitter4J 3.05]
34 - [ver1.6] on 30 Jul, 2015 - clear up code [tested on Mac 10.10.2 + Java 1.7.0_79, Twitter4J 3.05]
35 - [ver1.7] on 15 Feb, 2016 - Test in Processing 3.0
36 - [ver1.8] on 13 Apr, 2016 - integrate char text to the main program > new thread and boolean arrangement, time function to have clearer log
37 http://www.siusoon.net/home/?p=900
38 *****/
39
40 /* *****field notes*****
41 - constructed temporality > slow things down to see the invisible?
42 */

```

Figure 3.29: An Excerpt from *Thousand Questions'* source code: General notes

Furthermore, I have also conducted some experiments to understand data query on different platforms, OpenWeatherMap.org, for example, has been discussed earlier in the chapter. Figure 3.30 shows the returned query from the platform. This experiment has informed my understanding of how json works at the level of programming as well as data structure that is operated as a document. As discussed in Chapter 2, my methodology involves “experimental system” (Borgdorff, 2014, p. 113) that acts as a means of inquiry to reach out for things that are not yet known. Trying out queries that are offered by various platforms offer a deeper understanding on the materials that facilitates the thinking and research process.

```

{"cod":"200","calctime":0.0013,"cnt":15,"list":[{"id":2208791,"name":"Yafran","coord":{"lon":12.52859,"lat":32.06329},"main":{"temp":16.83,"temp_min":16.831,"temp_max":16.831,"pressure":1013.36,"sea_level":1032.78,"grnd_level":1013.36,"humidity":39},"dt":1444145600,"weather":[{"id":800,"main":"Clear","description":"Sky is Clear","icon":"01d"}]},{"id":2208425,"name":"Zuwarah","coord":{"temp":17.24,"temp_min":17.239,"temp_max":17.239,"pressure":1032.46,"sea_level":1033.35,"grnd_level":1032.46,"humidity":68},"dt":1444145600,"weather":[{"id":800,"main":"Clear","description":"Sky is Clear","icon":"01d"}]},{"id":2212771,"name":"Sabratrah","coord":{"temp":16.83,"temp_min":16.831,"temp_max":16.831,"pressure":1013.36,"sea_level":1032.78,"grnd_level":1013.36,"humidity":39},"dt":1444145600,"weather":[{"id":800,"main":"Clear","description":"Sky is Clear","icon":"01d"}]},{"id":2217362,"name":"Gharyan","coord":{"temp":15.83,"temp_min":15.831,"temp_max":15.831,"pressure":999.9,"sea_level":1032.33,"grnd_level":999.9,"humidity":40},"dt":1444145600,"weather":[{"id":800,"main":"Clear","description":"Sky is Clear","icon":"01d"}]},{"id":2216885,"name":"Zawiya","coord":{"

```

Figure 3.30: An excerpt of a returned query from OpenWeatherMap.org

While I was undertaking this research project I began to want to know more about the process behind the REST API that is offered by Twitter and questioned whether some kinds of random sampling have been implemented. This conforms to the discovery-led process, and speaking of it Borgdorff draws attention to the possibility of encountering “some unexpected issues or surprising questions” during the discovery (2011, p. 56). Working as a reflexive practitioner and being sensitive to the investigative materials, it was essential to know how the returned tweets were indexed through the Twitter platform. Therefore I went to the site and contacted Twitter’s developers for help. My



dialogue with Twitter's staff is documented in the excerpt below<sup>103</sup>:

My question on 26 Feb 2016:

I know that streaming API is using random sampling, what about Search API?

As search API is extracted tweets from last 7 days (include the day of making queries i believe), therefore returned tweets are in combination of current and past. However, as it returned 100 tweets per time max, what's the logic behind to pick the 100 tweets? Will it count duplicate from previous request? Has it implement random function to get the tweets from the data bank?

My follow up question on 27 Feb 2016:

For optimization: I am interested to know what do Twitter means by optimization, what's the logic behind. How to calculate the relevance - base on no of favorite counts or other machine learning algorithm?

For the 100 returned tweets, I wonder if there is max limit for the "more page". (but due to the rules setup by twitter, every extra page is count for 1 request, and there is a rate limit) But it seems that if i made the request within a short timeframe, the returned 100 tweets with different results. May be i should made a simultaneous request to understand more the logic. However, I am thinking if there is randomness and extra logic built in.

Twitter Staff wrote back on 27 Feb 2016:

Unfortunately we are not able to share the details behind the implementation here.

(A. Piper, personal communication, February 27, 2016)

---

<sup>103</sup> See: <https://twittercommunity.com/t/random-sampling-in-search-api/62115>

This unexpected dialogue (both my questions to and the response from Twitter) informs my discussion on the closedness of social media platforms and is articulated within the chapter earlier.

In short, this section shows how the work *Thousand Questions* developed and evolved during the past 4 years through my use of what I call reflexive coding practice. Simultaneously, I have explained how the practice has also informed and enriched the chapter's content. In summary, this project emphasises the running of live processes as part of the work in itself, executing unpredictable queries.



## Executing Micro-temporal Streams

Loading webpages, waiting for social media feeds and streaming videos and content are mundane activities in contemporary culture. Such mundane activity includes the involvement of network-connected devices from fixed desktop computers to portable tablets and smart watches, all involving data transmission and distribution across multiple sites—referred to as data. In these scenarios, data is constantly perceived as a stream (Berry, 2011, 2012, 2013; Chatzichristodoulou, 2012; Fuller, 2003), indicating its characteristics of vast volume, speed of update, continuous flow and delivery. The concept of streams now characterises the internet rather than web pages (Berry, 2011, p. 143). Data streams indicate events that are regarded as the latest and instantaneous versions. The *now* that we experience through perceptible streams is entangled with computational logic.

From social media feeds to playback video and mobile applications, users usually encounter a distinctive spinning icon during the loading, waiting and streaming of data content. This spinning icon represents an unstable streaming of the now. A graphical animation known as throbber indicates to users that something is loading and in-progress, but nothing more. A similar yet very different form of process indicator, such as a progress bar, expresses even more information than a throbber. In contrast to a progress bar, which is more linear in form, a throbber does not indicate any completed or finished status and progress. It does not explain any processual tasks in any specific detail. Previously such computer operations were commonly explained in conjunction with the use of a progress bar instead, as for example, with statements about the transferring and copying of specific files and directories, or illustrating installation procedures. A throbber usually appears in a graphical format, sometimes only with the additional word 'loading.' With a throbber, all that is presented is a spinning

icon, perceived as repeatedly spinning under constant speed as well as hinting at invisible background activities for an indeterminate and unforeseeable timespan. If one looks up the dictionary definition of the verb ‘throb’,<sup>104</sup> it is defined as a strong and regular pulse rhythm that resonates with a throbber’s design and in regards to how it performs on the internet today. But such design can be seen as an over-simplification of the micro-operations of networked technology, making one believe that the network is working with a certain regularity and that all data are queuing, thus rendering how we perceive liveness in superficial form.

Although a throbber is normally expressed in a graphical format, the underlying processes are specifically technical and operational. This chapter is informed by reflexive coding practice through reading, coding and creating a new throbber to inform my understanding of the computational logics behind a running throbber. This is explored with and through an experimental project titled *The Spinning Wheel of Life* (2016).<sup>105</sup> It suggests a different engagement and possibility of seeing this notable icon and understanding its related background activities. I employ the method of “cold gazing” (Ernst, 2013b), as explained in Chapter 2, and coding practice to reflexively examine the deep internal and material structure of data buffering and data processing. This chapter and the associated project open up the cultural and computational logics of data streams which are constantly rendering the pervasive and networked conditions of liveness.

As demonstrated in Chapter 1, the perspective of temporality is important in the general discussion of liveness and real-time, in particular, is essential when it comes to the notion of data streams. The notion of real-time concerns micro-time that might not be humanly perceptible (which is different from the concept of real-time in media studies as demonstrated in Chapter 1), insofar as real-time is operated through many micro-processes

---

<sup>104</sup> See: <http://www.oxforddictionaries.com/definition/english/throb>

<sup>105</sup> The project was first shown as a prototype at Malmö University as part of a workshop and masterclass on Execution in 2016 and it will be developed further for presentation at Transmediale2017 as part of a book launch event. See: [http://softwarestudies.projects.cavi.au.dk/index.php/Exe0.1\\_Winnie\\_Soon](http://softwarestudies.projects.cavi.au.dk/index.php/Exe0.1_Winnie_Soon)

that characterise a data stream. This chapter<sup>106</sup> establishes a link between liveness and temporality in contemporary software culture, with specific attention to micro-time and micro-processes. In the following, I will illustrate how a cultural and technical reading and understanding of an abstracted form of the throbber allows an examination of data streams. It will first unfold a cultural reading of a throbber and continue with a detailed discussion and analysis of the underlying operative and technical processes. Examining these processes may shed light on the understanding of liveness in contemporary software culture. The final section will discuss the project *The Spinning Wheel of Life* in detail.

## 4.1 A cultural reading of a throbber

With its distinct design characteristic of a spinning behaviour hinting at background processing, the throbber icon acts as an interface between computational processes and visual communication. One of the earliest uses of the throbber can be found in the menu bar of a Mosaic web browser from the early 1990s which was developed by the National Center for Supercomputing Applications (NCSA), with the browser interface designed by scientist Colleen Bushell (Albers, 1996; Roebuck, 2011, pp. 348-9). This throbber<sup>107</sup> contains a letter 'S' and a globe that spins when loading a web page. This kind of spinning throbber, with the company's graphical logo, can also be witnessed in subsequent software browsers (see Figure 4.1). While the throbber spins it visually indicates actions are in progress. These actions, from a user point of view, could be interpreted as the loading of web data or connecting to a website by a software browser. From a technical perspective it involves internet data transmission and a browser that renders the inter-actions of code. The spinning behaviour stops when a webpage has finished loading within a browser. A web browser is software

---

<sup>106</sup> This chapter is developed through a series of seminars and workshops, with the theme of 'Execution' during 2015-2016, organised by Critical Software Thing. Earlier versions of this chapter have been published in ISEA2016 Conference Proceedings (Soon, 2016b) and forthcoming book chapter in *Executing Practices* (2017) (Soon, 2017). See:

<http://softwarestudies.projects.cavi.au.dk/index.php/CriticalSoftwareThing>

<sup>107</sup> The Mosaic throbber also allows user to click on it to stop loading a webpage (Roebuck, 2011, p. 348).

able to render and display requested content, making network calls and requests and storing data locally (Garsiel & Irish, 2011). In this respect, the spinning throbber represents complex inter-actions of code under network conditions. A throbber, with its spinning characteristic, can therefore be said to be rooted in, and specific to, internet culture.

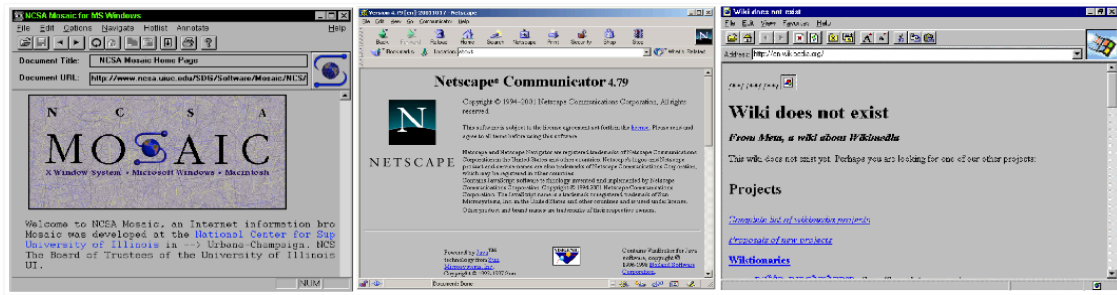


Figure 4.1: Throbber in different browsers. The specific browsers' image are taken from Computer History Museum,<sup>108</sup> soft32<sup>109</sup> and FavBrowser.com<sup>110</sup>

More recently the throbber icon is no longer only attached to software browsers, appearing also on different web and mobile applications including social media platforms in particular. The contemporary throbber transforms into a spinning wheel<sup>111</sup> that consists of lines or circles that are arranged in radial and circular form, moving in a clockwise direction. A throbber is mostly produced in the format of an animation, loading each frame one by one as most of the throbbers that we experience have been preconfigured during the development of the software. The visual logic of the throbber image relies on other functions<sup>112</sup> that have been implemented in the source code. In other words, the actual handling of data does not have a direct relation with how the throbber is animated. Therefore a throbber is animated and spun, or throbbed, at a constant rate, demonstrating a regular

<sup>108</sup> See: <http://www.computerhistory.org/atcm/happy-25th-birthday-to-the-world-wide-web/>

<sup>109</sup> See: <http://netscape-communicator.soft32.com/>

<sup>110</sup> See: <http://www.favbrowser.com/the-history-of-internet-explorer/>

<sup>111</sup> The use of lines that indicates the progress activity of a computer can be found in the early operating system of Unix that consists of few string characters as '[', '—', '\', '|', '/', ']' (Roebuck, 2011, p. 349). This is also used in the cover image of this thesis. The last section of this chapter also documents a Unix Shell script of a throbber.

<sup>112</sup> For example, a function handles the loading of an image or video, or a network connection in the form of code. A throbber is made to appear when a program is still waiting for the content to load fully or a network connection to establish successfully.

tempo. Each individual element of the wheel<sup>113</sup> sequentially fades in and out repeatedly to create a sense of animated motion (see Figure 4.2). These spinning wheels appear after a user has triggered an action such as swiping a screen with feeds in order to request the updated information. They also appear after a user has confirmed an online payment or is waiting for a transaction to complete. Perhaps most commonly of all a throbber is seen when a user cannot watch a video clip loading smoothly over an internet connection. As a result an animated throbber appears as a spinning wheel on a black colour background occupying the whole video screen while the video is buffering.

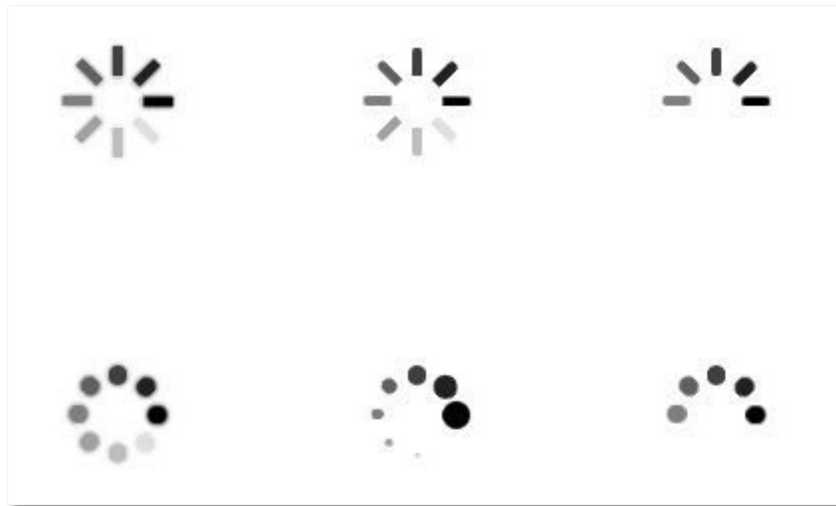


Figure 4.2: Throbber in the form of circles and lines. Image is retrieved from <http://designmodo.com/css3-jquery-loading-animations>.

A throbber represents the speed of network traffic which is also tied to our affective states and perception of time. Emotionally it can be annoying and frustrating to encounter buffering as it involves interruption (Broida, 2010, July 14; Stelter, 2011). Things do not flow smoothly and users become impatient when waiting for an unknown period of time or for something yet to come. Taiwanese artist Lai Chih-Sheng exhibited his throbber animation, titled *Instant*(2013),<sup>114</sup> in the Hong Kong Eslite Gallery with a minimalistic

---

<sup>113</sup> Coincidentally, the visual design of a throbber is similar to the design of early wristwatches (with crystal guards) that were made for soldiers in World War I. Both include the concept of a wheel in the form of circles or lines of petal shape. See: <http://www.oobject.com/category/earliest-wrist-watches/>

<sup>114</sup> See:



presentation, expressing the relationship between waiting and time. This waiting is considered unproductive in that it consumes time. As artist-researcher James Charlton describes it: “It is a gaze that goes beyond the screen to an event not yet here” (2014, p. 171). The loading time of the throbber appears wasted and unproductive as it is often associated with the perception of slowness of a network.

On September 10<sup>th</sup> in 2014 a campaign called “Internet Slowdown day”<sup>115</sup> was launched as part of the ‘Battle for the Net’ promoting net neutrality and internet freedom. Customised loading icons similar to throbber were put up on different websites symbolising the potential impact of controlled traffic that would be implemented by Internet Service Providers in the name of increasing profit. The campaign argues for internet speed equality across all websites and that no unequal conditions, such as fast-lane traffic, should be given to any prioritised website. More than 10,000 corporations such as Etsy, Kickstarter, Netflix, tumblr and Vimeo, showed support by putting up self-designed throbber icons. As is evident in this context the throbber has a significant and symbolic meaning within cultural and political realms.

In contemporary art this cultural icon was remade by artist Aristarkh Chernyshev, who showed the spinning behaviour through customised LEDs in a physical installation. The LEDs formulated the word ‘LOADING,’ circulating in a motion directly reminiscent of a spinning throbber. Chernyshev’s artwork *LOADING* (2007)<sup>116</sup> aimed to present this icon and its data exchange process as a cultural phenomena, with the cultural icon of the throbber<sup>117</sup> here expressing various dimensions of time—from the loading time of a browser to the regular tempo of a spinning throbber, to the

---

[https://www.facebook.com/ESLITE.PROJECTONE/photos/?tab=album&album\\_id=437623016346200](https://www.facebook.com/ESLITE.PROJECTONE/photos/?tab=album&album_id=437623016346200)

<sup>115</sup> For more details, see: <https://www.battleforthenet.com/sept10th/>

<sup>116</sup> See: <https://festivalenter.wordpress.com/2009/04/09/electroboutique-by-alexei-shulgin-roman-minaev-aristarkh-chernyshev/>

<sup>117</sup> Other artists have also explored the throbber icon. For example artist Gordan Savičić explores the perception of time through his work *Loading*, that turns an ordinary windowpane into a screen. Additionally, Fedora’s artwork team produces a series of throbber images that put emphasis on the design of spinning.

See: <http://www.yugo.at/processing/?what=loading> and

<https://fedoraproject.org/wiki/Artwork/ArtTeamProjects/Fedora7Remix/Rhgb/Throbber>

slowness of internet network—in understanding data streams.

All the instances above suggest that the smoothness of network traffic is important as it relates to user experience, time perception and the productivity of waiting through daily instances. When watching data streams we may expect things would flow in a timely, smooth and continuous fashion that constitutes the notion of liveness. It might be similar to television liveness in which there is a constant flow of “discrete segments” (Ellis, 1992/[1982], p. 112) as well as a flow of connection. This flow of connection means a constant unfolding of reality (Feuer, 1983), or “alive view” (Health & Skirrow, 1977, p. 54), of the world—a connection between the viewer and the outside world.

The concept of flow was theorised by media studies scholar Raymond Williams in 1974, in which the programming of content, one programme after another, implies continuity to hold their viewers (1974, pp. 80-4). This concept of flow is fundamental to television studies, suggesting that the technology of television involves institutional plans of programming, the delivery of mediatised representation as well as the viewer experience thereof. As Williams puts it, “[the] phenomenon of planned flow, is then perhaps the defining characteristic of broadcasting, simultaneously as a technology and as a cultural form” (1974, p. 80). Even though there is interruption, what Williams calls the “natural break” of the advertisement in television, its arrangement is a planned flow with the number of breaks and the corresponding duration as part of the overall television production (1974, pp. 90-3). Therefore, television technology consists of a relatively continuous and steady flow of programming which constitutes the immediacy effect, or liveness, that is brought to viewers by the flow of content.

Media studies scholars Stephen Heath and Gillian Skirrow further discuss the temporal dimension of flow in the television context. They argue that a live television programme alludes to an equivalence between creation and transmission, viewing time and time of event (Health & Skirrow, 1977, p.

53). The organisation of different times maintained the immediacy of a television programme through the experience of flow. As they summarise:

The immediacy effect is supported by the experience of flow: like the world, television never stops, is continuous. Programs, however, organise times within that flow; within the context of the overall definition of television as 'live' and on the maintained basis of 'immediacy' (Health & Skirrow, 1977, p. 54).

However, in the context of digital streams in which the experience of immediacy is concerned, a data stream is organised through computational time with different micro-processes that exhibit highly unstable temporality instead. The interruption of a data stream, such as buffering process that manifests in the form of a throbber, cannot be planned (as with television) insofar as it is subjected to its technical conditions at any moment of time. Additionally, the immediacy of a stream's narration cannot be simply understood as a planned sequential flow or as discrete segments of programmes. The discreteness of streams is not characterised by content but rather, as I propose in this chapter, by the very nature of digital and computational processing.

The throbber is widely seen and used in cultural practices but most people in everyday life do not want to see a throbber on their screens as it represents slowness and interruption. Application providers present a near perfect connection in which data flows smoothly as streams, or a stream is "fully synchronized"<sup>118</sup> with 'excellent' performance. Arguably the notion of stream or flow that has been cultivated perhaps makes us forget and become unaware of the materiality of digital networks. The material nature of the network exhibits something that is unpredictable, unstable and discontinuous, which is beyond seemingly 'natural' breaks and beyond

---

<sup>118</sup> A press release from Logitech promised the web camera was a product that would "ensure people experience fully synchronized, high-quality video and audio communications over the Internet" by working closely with Skype. See: <http://ir.logitech.com/all-featured-press-releases/press-release-details/2005/Logitech-and-Skype-Team-up-on-Video-Logitech-to-Provide-Skype-Certified-Webcams-and-Headsets/default.aspx>

visible and apparent interruptions. Beyond different cultural instances however, the operative and technical dimensions of a running throbber should not be underestimated, as they can provide a specific perspective for further understanding how the experience of liveness is being organised computationally as streams.

In taking into consideration the operative and technical perspectives of network transmission, media and cultural studies scholar Florian Sprenger argues that the concept of a stream or a flow is a metaphor. He says:

The network structure of today's communication channels and of their information stream is often understood as providing a direct connection between users and services or between two communication partners, even though there cannot be any direct connections on digital networks. The metaphor of the flow conceals the fact that, technically, what is taking place is quite the opposite. There is no stream in digital networks (Sprenger, 2015, pp. 88-9).

Sprenger highlights the possible misconception of a flow or a stream, suggesting that there is a gap between the experienced and operative streams. He reminds us that two widely used concepts—flow and stream—in digital media are metaphors that potentially mislead anyone looking to understand the actual technical processes that take place beneath a stream.

In the following section I develop the notion of 'discontinuous micro-temporality' as a way to rethink the notion of flow and stream beyond and beneath its continuous immediate experience in networked environments.

## **4.2 Micro-temporal analysis**

Micro-temporality focuses on the detailed processes of computation. Drawing upon Ernst's notion of "micro-temporality," the focus of such approaches is on the nature and operation of signals and communications,

mathematics, digital computation and dynamic network events within deep internal and operational structures (2013b, pp. 186-9). The added prefix ‘micro’ addresses the micro-operative processes that are not apparent to the immediate human register.

Following Foucault, Ernst’s notion of micro-temporality draws on the concept of discontinuity (2006). In Foucault, discontinuity offers an alternative perspective to understanding knowledge beyond its stable form of narration and representation (1972, p. 3). Both Foucault and Ernst use discontinuity as a means to examine the gaps, silences and ruptures of things that go beyond signs or representational discourses. The notion of liveness, as demonstrated in Chapter 1, is examined through a perspective of temporality that constructs a sense of liveness and immediacy with real-time technology. However, the focus on (micro-)temporality in this section is intended open up the notion of liveness, shifting gradually from a discursive to a non-discursive discussion of data streams that underlies deep processual micro-temporality.

To bring together concepts of discontinuity and micro-temporality is to offer an alternative perspective in examining the conditions of liveness beyond a planetary scale. Streams can be understood as highly capitalised and as operating at massive scales under the contemporary conditions of a globalised economy that is disseminates to every part of the world. In the words of philosopher Peter Osborne, contemporaneity “is primarily a global or a planetary fiction” (2013, 26). Alternatively, the notion of discontinuous micro-temporality highlights the micro-processes and gaps in a stream that are manifested within the newest presence-oriented feeds and their regular interruptions by a throbber. The concept of discontinuous micro-temporality extends our understanding of a seemingly smooth stream and a steady temporal throbber.

Through his micro-temporal analysis media studies scholar Shintaro Miyazaki argues that there is rhythmical quality to algorithms (2012). This, in part, informs an understanding of a throbber beyond a regular tempo.

Logically, a throbber is displayed when data is being received, loaded and buffered beyond an acceptable threshold of latency. Once it is below an acceptable threshold, content will be shown in lieu of an animated throbber. Micro-temporality, in the context of streams, can be understood as the variation and latency of data processing. In order to express the rhythms and differences of networked data traffic, I have made a throbber with lines of code that are able to display varying speeds (see Figure 4.3). In Figure 4.3, lines 16 and 25 are the major code syntax that controls the speed of each display of an ellipse.

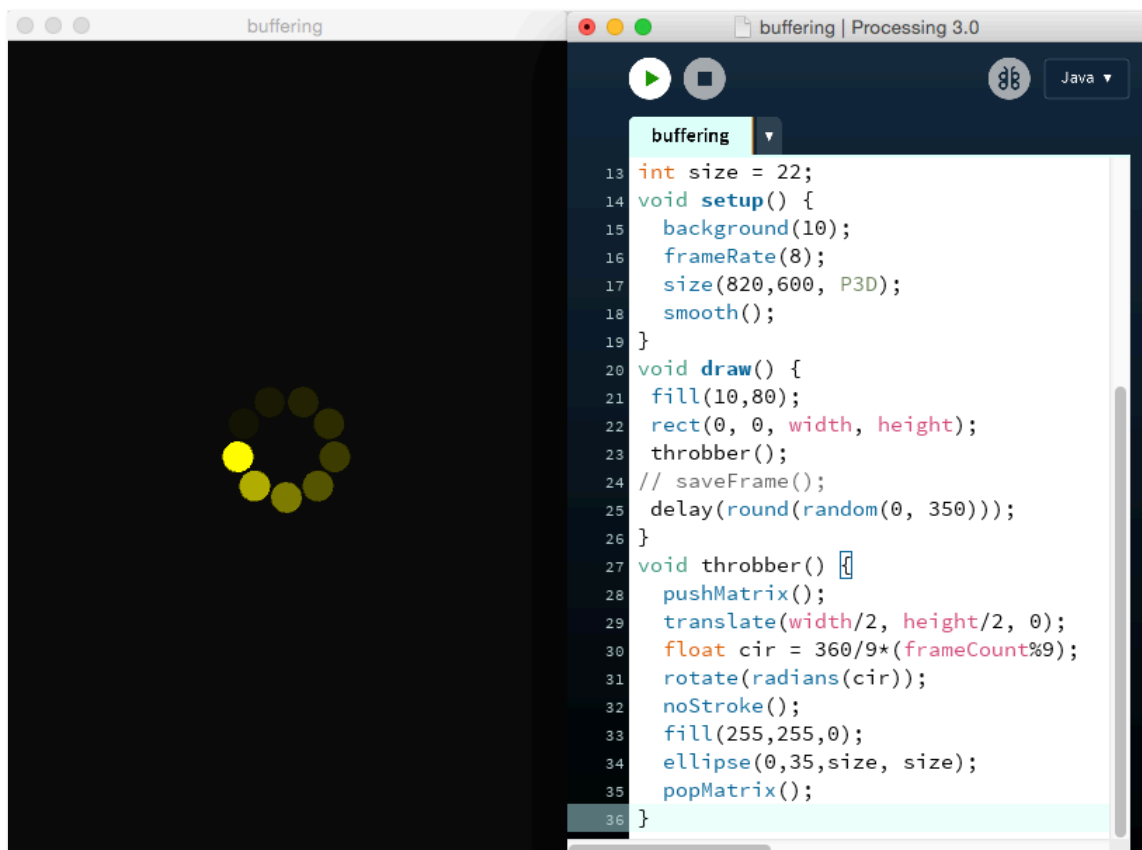


Figure 4.3: A code-based throbber

Instead of using an animated image, I have written code to describe all the visual elements of a throbber, such as position, circles/shapes, colours and rotations, exploring whether there is a possibility for a throbber to perform in a different way and questioning why it has to be displayed in a regular tempo. It is also similar to how Rolling Jr would describe as a “thought experiment” in which artistic research is developed through “thinking in a

material,” “thinking through a context” and “thinking reflexively” (2014, pp. 162-3). Such thinking in, and through, code offers the possibility of regarding the throbber in new ways as well as the ways in which the production of artworks can inform the research findings (this project is further developed into an artwork I will discuss later).

To further understand the operational logic behind a throbber, I began to investigate how data is transmitted through and within the machine and its networked architecture, taking a techno-engineering perspective to understand how things operate at an epistemic level. In particular, I engage with engineering concepts such as socket networking, buffering, buffer playback, TCP/IP protocol, streaming protocol and packet-switching that are essential to understand the operative and technical processing of streams. The following four sections offer an analysis of digital signal processing, data packets and network protocols, buffer and buffering that lead to the absence of data.

#### *4.2.1 Digital Signal Processing*

As opposed to the continuous-time signals of analogue systems, the digital adopts the model of discrete-time with independent variables in signal processing. This model means that each discrete state is countable and measurable according to a distinct value and can be represented by a sequence of numbers (Kumar, 2015, pp. 3-5). However, the signals are discrete in time, alluding to the value between two discrete-time instances is not defined (see Figure 4.4). Therefore, the *flow* of data that we experience through a screen is discrete in its nature. Within digital signal processing the data stream is discontinuous (discrete) in terms of its time signal.

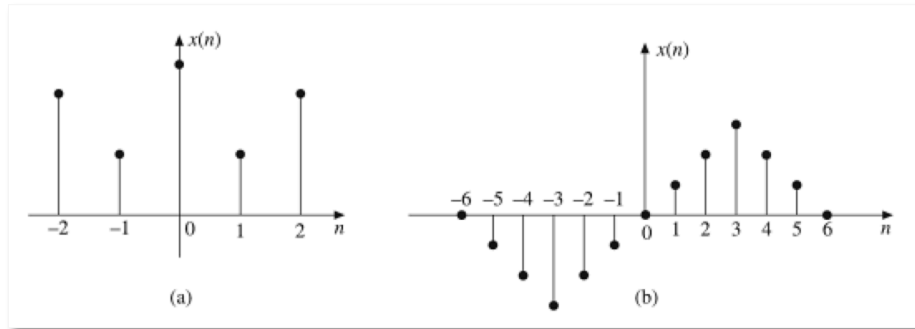


Figure 4.4: Discrete time signals. Reprinted from Digital Signal Processing (p. 18), by A. A. Kumar, 2015, Delhi: PHI Learning Private Limited. Copyright 2015 by PHI Learning Private Limited.

Inside a computer machine a processor's clock<sup>119</sup> fundamentally coordinates the instruction and the processing of data in the form of binary signals. The clock signal is expressed in a square wave of a clock circuit (see Figure 4.5) driven by an oscillating signal that is produced by a quartz crystal oscillator (Burrell, 2004, p. 75). A clock cycle refers to the amount of time between two pulses of an oscillator and, as indicated in Figure 4.5 the signal changes from 0 to 1 and then back to 0. Computational time is different from human time as it has its own rhythm and unit that is beyond human perception. With the advancement of technology the processing speed of a computer has been enormously accelerated. In modern 21<sup>st</sup> Century processors the clock rate is usually measured in Megahertz (MHz) or Gigahertz (GHz), which refers to a clock cycle or clock tick. For example, a clock speed of 5 GHz allows the computer to process 5 thousand million clock cycles/ticks per second. To run the program in Figure 4.3, for example, it takes less than two seconds because a computer can run more than an instruction per second.

<sup>119</sup> Thanks to Brian House who first introduces the concept of computer clock in the exe0.1 workshop. See: [http://softwarestudies.projects.cavi.au.dk/index.php/Exe0.1\\_Brian\\_House](http://softwarestudies.projects.cavi.au.dk/index.php/Exe0.1_Brian_House)



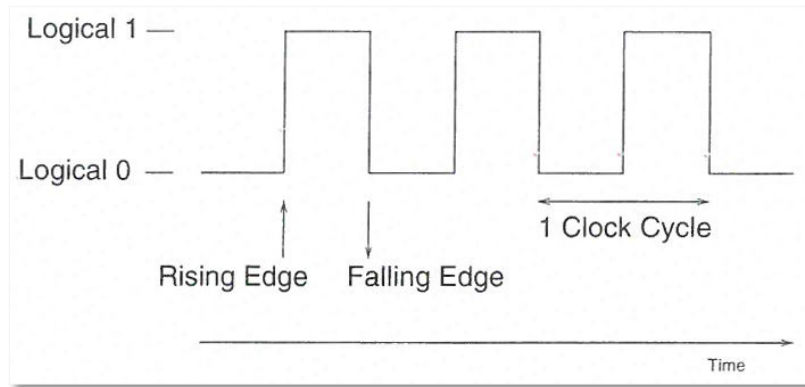


Figure 4.5: The clock cycle. Reprinted from *Fundamentals of Computer Architecture* (p. 75), by M. Burrell, 2004, New York: Palgrave Macmillan. Copyright 2004 by Palgrave Macmillan.

Following the von Neumann architecture that was first initiated in 1945, mathematician and physicist John von Neumann designed a computer architecture consisting of a processing unit that contained an arithmetic logic unit, a control unit and a memory unit for performing arithmetical operations, operational sequence control and data and instruction storage. This is also known as a stored-program computer. These units are coordinated by a central clock (von Neumann, 1945, pp. 1-2), executing computer instructions in a precise manner.

The appearance and disappearance of a throbber is rendered by code, instructing when a throbber should be displayed on a screen. However, computer instruction is more than source code. In computer science and engineering the *Fetch-Execute* cycle is used to describe how a Central Processing Unit (CPU) performs code instructions through a series of steps that are executed within clock cycles (Burrell, 2004, p. 135; Frabetti, 2015, p. 153). A simple calculation like '5+6' includes further micro-instructions such as the copying of the memory location, the storing of the instruction and the individual values of 5 and 6 (in bits pattern format), calculating the sum of the values and writing the calculated result (cf. Loudon & Lambert, 2012, pp. 4-5). The high level instruction breaks into many micro-instructions by fetching and executing values from and in the memory space. The micro-instructions are highly ordered. For example, the values of 5 and 6 must be stored before they can be retrieved for the next process. The

instruction pointer (also known as program counter) is used to keep track of the instruction sequence. This pointer is incremented after fetching an instruction and storing the memory address of the next instruction to be executed. The computer will continue repeating the cycles which fetch instructions and data from memory and then execute them one after another in sequence until the final instruction is reached (see also Frabetti, 2015, pp. 150-9; Snodgrass, 2017, in press). This cyclic process of fetching and execution is coordinated by the clock in which one fetch cycle might take more than one clock cycle to complete. The time to finish one fetch cycle is measured in clock cycles. In short, executing code instructions involves the reading and writing of memory (memory is used here in a broad sense that includes the main computer memory, instruction register and memory buffer register, etc.), generating a sequence of micro-operational steps and the actual computation. The appearance or disappearance of a throbber on a screen is not an exception. All of the code instructions are operated across on/off states, generally known as ‘flip-flops’ and at the level of the quartz-crystal circuit via logic gates used to store and control data flow.

Underneath a throbber is the inter-action of data, source code, micro-instructions and many other entities. The reading and writing of memory is one of the key operations that take place in the CPU. Since the CPU involves multiple units, there are many components involved in the process of a fetch-execute cycle. The clock determines the access of memory. Referring to Figure 4.5, the writing of values in the internal memory can only be done on a clock edge, either the rising or the falling edge. The oscillating time between 0 to 1 or 1 to 0 is significantly small when it is compared to the whole clock cycle. According to computer scientists David A. Patterson and John L. Hennessy, this is called “an edge-triggered clocking methodology,” alluding to “any values stored in a sequential logic element are updated only on a clock edge” (2007, p. 290). In other words, the micro-temporality of instructions is driven by the internal clock as there are things that have to be done exactly at a specific time. However, what is important is that if the memory is not properly stored within a designated clock cycle “garbage data” (Hyde, 2004, p. 154) will be read instead, causing system

failure.

This mismatch between writing and reading is an important concept in understanding what is occurring behind a running throbber. Imagine a smartphone waiting for a video stream such that the data can be stored in the computer's buffer memory for viewing. If the data cannot arrive and write on the memory on time it is possible that the CPU will read garbage data. Since there is the logic of buffering as manifested in a throbber display (the detailed logic of buffering will be discussed in a later section), this minimises the reading of garbage data. This section starts to introduce the micro-operations that are happening within a machine. The signal processing suggests that streams operate in the digital nature of discrete signals. Importantly, the machine clock forms a basic infrastructural activity of contemporary technology, organising and maintaining the sequences and components of computation that are essential in performing operational tasks through micro-instructions.

Such micro-processes are extremely difficult to perceive as they involve hidden processes. Just because their micro-temporality falls beneath the threshold of human sensory perception does not mean that they are unimportant. Invisibility, as discussed in Chapter 2, is one of the key concepts in understanding computational culture that is always in process with us. From mundane computational interfaces to sophisticated platforms, invisible processes are intertwined with wider cultural forces. This micro-perspective allows us to be attentive to how time is structured and organised computationally and differently. Precisely, it considers how computational time makes critical decisions to determine the operation of things (the case of clock edges for instance). Therefore, micro-temporality is also about considering time-critical<sup>120</sup> processes. As Ernst puts it, “[time-criticality refers] to a special class of events where exact timing and the temporal momentum is *decisive* for the processes to take place and succeed at all”

---

<sup>120</sup> Ernst discusses the time-critical perspective within micro temporal media, where he argues that this is different from time-based media. This time-critical perspective is more focused on the operational and technomathematical media to understand our culture beyond narration (2013a).

(2013a, *original emphasis*). This section begins to unfold the micro-processes of computation to understand the temporal dynamic nature of technical media. The next section will continue to focus on this subject but will discuss technological network transmission that is the core of a stream's delivery.

#### 4.2.2 *Data packets and Network protocols*

A close reading of the *Transmission Protocol Specification* (RFC 793) and cold gazing of network traffic help to explicate the micro-temporality of data packets and network protocols. I have used an open source network packet analyzer, software called WireShark,<sup>121</sup> that monitors and captures network packets.<sup>122</sup> It is able to capture numerous packet information. For my experiment I used WireShark to monitor a YouTube video. YouTube implements media streaming technology to deliver a playback video file. A 56 second video clip, for instance, generates a total of 2757 packets that are captured by WireShark. It is important to highlight that packets are not delivered in a single instance of time but spread across multiple timeframes that are barely noticeable<sup>123</sup> to the human eyes. This is also why there is a need to use WireShark to capture all these micro timeframes for a detailed analysis. In addition, the experiment emphasises the mechanism of TCP/IP and its data flow processes which are time-critical because they relate to how data transfers operate geopolitically and are constrained by structures, infrastructures and “micro-decisions” (Sprenger, 2015) along a transmission process.

In the late 1960s, ARPANET, the world's first packet switching network was introduced laying the groundwork which led to the development of the internet as it has developed today. The concept of ‘packet switching’ was fundamental to understanding how data is organised and flows. A data stream was chopped into smaller blocks as ‘packets,’ which were then sent

---

<sup>121</sup> WireShark is an open source software available for Windows and Unix platform. See: <https://www.wireshark.org/>

<sup>122</sup> See the last chapter for more detailed steps.

<sup>123</sup> The time as indicted in the WireShark was measured in microseconds.

via a communications channel in and through different routes, rates and sequences, known as packet switching (Baran, 2002). According to Baran, one of the inventors of the packet switched computer network, real-time connections between sender (transmitting end) and user (receiving end) are an illusion. Instead a sufficiently fast data rate gives only a *sense* of real-time connection between a sender and receiver. Fundamentally, the routing of a data packet transmits through different sites. Although a selected path is based on “adaptive learning of past traffic,” there are real-time decisions that have to be made to locate the shortest path<sup>124</sup> due to the dynamics of network conditions. In other words, data travels “via highly circuitous paths that could not be determined in advance” (Baran, 2002, pp. 43-4).

The Internet includes TCP/IP (Transmission Control Protocol/Internet Protocol), which are currently the major protocols for networked data transmission and entail the massive distribution of data over real-time connections. The internet backbone stems from these protocols. According to Galloway, protocols are specific technical standards that consist of “a set of recommendations and rules” (2004, pp. 5-6). Most data streams are transmitted via a TCP/IP connection, ensuring a reliable delivery through two major processes: a 3-way ‘handshake’ and ‘fragmentation’ (Galloway, 2004, pp. 40-2; Postel, 1981b). Generally, establishing a communication channel between two connection points requires the handshaking process (see Figure 4.6). The data packet is based on a sequence of numbers and computational logic (i.e. checksum) to reassemble and reformulate the sequence as the original sender’s message (this is called fragmentation) (Galloway, 2004, p. 45).

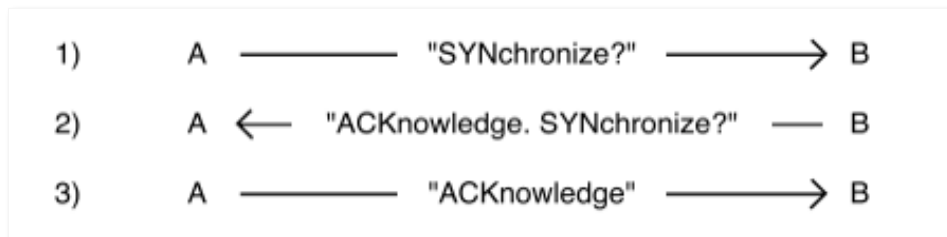


Figure 4.6: Three-way handshake. Reprinted From *Protocols* (p. 43), by A. Galloway, 2004, Cambridge, Massachusetts, London: The MIT Press. Copyright 2004 by The MIT Press.

<sup>124</sup> For more details about the determination of the shortest path, see (Meinel & Sack, 2013, pp. 350-2)

A timer is implemented in the protocols of TCP/IP, maintaining a reliable connection by keeping track of packets that are transmitted within a certain time interval. It is an active agent that is implemented across any TCP/IP connection where a stream is operated, in particular to control the processes of network establishment and data transmission between a sender and receiver. Throughout the whole process of data packet transmission, a timer is set to wait for the return of an “acknowledgement” (referred to as ‘ACK’) from the corresponding side and “If the ACK is not received within a timeout interval, the data is retransmitted” (Postel, 1981b, p. 4). As such, there are always time-checking mechanisms implemented at the level of network protocol. Therefore, one of the important features of a timer is, according to Jonathan Postel, the editor of the *RFC 793*, “[t]o govern the flow of data between TCPs” (1981b, p. 9)

In addition to a timer, there are different waiting times as expressed in a series of states of a connection lifetime: ‘LISTEN,’ ‘SYNC-SENT,’ ‘SYNC-RECEIVED,’ ‘ESTABLISHED,’ ‘FIN-WAIT1,’ ‘FIN-WAIT2,’ ‘CLOSE-WAIT,’ ‘CLOSING,’ ‘LAST-ACK,’ ‘TIME-WAIT’ and ‘CLOSED.’ Most of these states refer to different waiting times for things to be processed: for example, waiting for a connection request, matching a connection, confirming a connection and terminating a connection request. These states will change before and after certain actions. Using the three-way handshake as an example (see Figure 4.7), if a client computer connects to a particular YouTube video link and requests to view the content a ‘SYNC’ request is sent from A (a client) to B (a server, Youtube) and the connection state at A is changed from ‘LISTEN’ to ‘SYNC-SENT.’ While B receives the request it will respond to A with both ‘SYNC’ and ‘ACK’ status in a message and the connection state for B is changed from ‘LISTEN’ to ‘SYNC-RECEIVED.’ Finally A has to respond with an ‘ACK’ status again and then the connection state is now changed from ‘SYNC-SENT’ to ‘ESTABLISHED.’ When B receives the message from A the state changes from ‘SYNC-RECEIVED’ to ‘ESTABLISHED.’ The state of ‘ESTABLISHED’ is regarded as “the normal state for the data transfer phase of the connection” (Postel, 1981b, p. 21).

The actual content, video for instance, is technically called ‘application data and can only be delivered in this state. This is the actual implementation of the three-way handshake process: first to synchronize (with a ‘SYNC’ request), then to acknowledge the receipt of this (with a ‘SYNC’ and ‘ACK’ respond) and finally returning the acknowledgement status (with an ‘ACK’). This process acts as a base, an established connection, and can be thought of a trust relationship that can be built upon to allow further data exchange. This relationship is expressed in human language, such as the greeting messages, ‘Client Hello’ and ‘Server Hello,’ as indicated in Figure 4.8. The timer implementation is important in building this trust relationship because every state change is required to be done at a certain time interval. This ‘ESTABLISHED’ state is time-critical, whether the application data can be delivered and received is essentially subject to the prerequisite of this state and streaming is no exception, where further exchange can be built upon on the basis of a trust relationship.

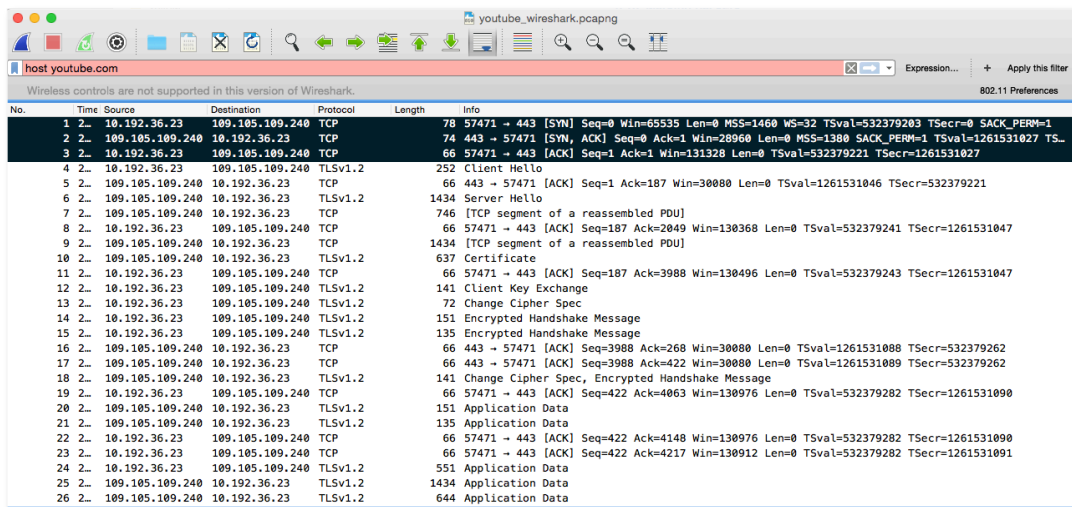


Figure 4.7: Data packet analysis I - the screen shot highlights the three-way handshake

No.	Time	Source	Destination	Protocol	Length	Info
2	2015-12-15 16:54:19.334473	109.105.109.240	10.192.36.23	TCP	74	443 → 57471 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1380 SACK_PERM=1 TSval=1261531027 TSecr=532379203 WS=128
3	2015-12-15 16:54:19.334596	10.192.36.23	109.105.109.240	TCP	66	57471 → 443 [ACK] Seq=1 Ack=1 Win=131328 Len=0 TSval=532379221 TSecr=1261531027
4	2015-12-15 16:54:19.335597	10.192.36.23	109.105.109.240	TLSv1.2	287	Client Hello
5	2015-12-15 16:54:19.335370	109.105.109.240	10.192.36.23	TCP	66	443 → 57471 [ACK] Seq=1 Ack=187 Win=30080 Len=0 TSval=1261531046 TSecr=532379221
6	2015-12-15 16:54:19.335530	109.105.109.240	10.192.36.23	TLSv1.2	1434	Server Hello
7	2015-12-15 16:54:19.336295	109.105.109.240	10.192.36.23	TCP	746	[TCP segment of a reassembled PDU]
8	2015-12-15 16:54:19.336386	10.192.36.23	109.105.109.240	TCP	66	57471 → 443 [ACK] Seq=187 Ack=2049 Win=130368 Len=0 TSval=532379241 TSecr=1261531047
9	2015-12-15 16:54:19.337538	109.105.109.240	10.192.36.23	TCP	1434	[TCP segment of a reassembled PDU]
10	2015-12-15 16:54:19.338336	109.105.109.240	10.192.36.23	TLSv1.2	637	Certificate
11	2015-12-15 16:54:19.338439	10.192.36.23	109.105.109.240	TCP	66	57471 → 443 [ACK] Seq=187 Ack=3988 Win=130496 Len=0 TSval=532379243 TSecr=1261531047
12	2015-12-15 16:54:19.377381	10.192.36.23	109.105.109.240	TLSv1.2	141	Client Key Exchange
13	2015-12-15 16:54:19.377381	10.192.36.23	109.105.109.240	TLSv1.2	72	Change Cipher Spec
14	2015-12-15 16:54:19.377382	10.192.36.23	109.105.109.240	TLSv1.2	151	Encrypted Handshake Message
15	2015-12-15 16:54:19.377382	10.192.36.23	109.105.109.240	TLSv1.2	135	Encrypted Handshake Message
16	2015-12-15 16:54:19.395731	109.105.109.240	10.192.36.23	TCP	66	443 → 57471 [ACK] Seq=3988 Ack=208 Win=30080 Len=0 TSval=1261531088 TSecr=532379262
17	2015-12-15 16:54:19.396513	109.105.109.240	10.192.36.23	TCP	66	443 → 57471 [ACK] Seq=3988 Ack=422 Win=30080 Len=0 TSval=1261531089 TSecr=532379262
18	2015-12-15 16:54:19.398010	109.105.109.240	10.192.36.23	TLSv1.2	141	Change Cipher Spec, Encrypted Handshake Message
19	2015-12-15 16:54:19.398105	10.192.36.23	109.105.109.240	TCP	66	57471 → 443 [ACK] Seq=422 Ack=4063 Win=130976 Len=0 TSval=532379282 TSecr=1261531090
20	2015-12-15 16:54:19.398323	109.105.109.240	10.192.36.23	TLSv1.2	151	Application Data
21	2015-12-15 16:54:19.398327	109.105.109.240	10.192.36.23	TCP	135	Application Data
22	2015-12-15 16:54:19.398424	10.192.36.23	109.105.109.240	TCP	66	57471 → 443 [ACK] Seq=422 Ack=4148 Win=130976 Len=0 TSval=532379282 TSecr=1261531090
23	2015-12-15 16:54:19.398426	10.192.36.23	109.105.109.240	TCP	66	57471 → 443 [ACK] Seq=422 Ack=4217 Win=130912 Len=0 TSval=532379282 TSecr=1261531091
24	2015-12-15 16:54:19.399480	10.192.36.23	109.105.109.240	TLSv1.2	551	Application Data
25	2015-12-15 16:54:19.399480	10.192.36.23	109.105.109.240	TLSv1.2	1434	Application Data
26	2015-12-15 16:54:19.456314	109.105.109.240	10.192.36.23	TLSv1.2	644	Application Data
27	2015-12-15 16:54:19.456366	10.192.36.23	109.105.109.240	TCP	66	57471 → 443 [ACK] Seq=907 Ack=6163 Win=129120 Len=0 TSval=532379338 TSecr=1261531148
28	2015-12-15 16:54:19.456722	10.192.36.23	109.105.109.240	TLSv1.2	135	Application Data
29	2015-12-15 16:54:19.457749	109.105.109.240	10.192.36.23	TLSv1.2	791	Application Data
30	2015-12-15 16:54:19.457838	10.192.36.23	109.105.109.240	TCP	66	57471 → 443 [ACK] Seq=976 Ack=6880 Win=130336 Len=0 TSval=532379339 TSecr=1261531148
31	2015-12-15 16:54:19.458038	10.192.36.23	109.105.109.240	TLSv1.2	135	Application Data
32	2015-12-15 16:54:19.475639	109.105.109.240	10.192.36.23	TCP	66	443 → 57471 [ACK] Seq=6888 Ack=1045 Win=31104 Len=0 TSval=1261531168 TSecr=532379338
33	2015-12-15 16:54:19.476585	10.192.36.23	109.105.109.234	TCP	78	57472 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=1408 WS=2 TSval=532379356 TSecr=0 SACK_PERM=1
34	2015-12-15 16:54:19.498485	109.105.109.234	10.192.36.23	TCP	74	443 → 57472 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1380 SACK_PERM=1 TSval=1261531191 TSecr=532379356 WS=128
35	2015-12-15 16:54:19.498471	10.192.36.23	109.105.109.234	TCP	66	57472 → 443 [ACK] Seq=1 Ack=1 Win=131328 Len=0 TSval=532379378 TSecr=1261531191
36	2015-12-15 16:54:19.498869	10.192.36.23	109.105.109.234	TLSv1.2	248	Client Hello
37	2015-12-15 16:54:19.520200	109.105.109.234	10.192.36.23	TCP	66	443 → 57472 [ACK] Seq=1 Ack=183 Win=30080 Len=0 TSval=1261532123 TSecr=532379378
38	2015-12-15 16:54:19.522412	109.105.109.234	10.192.36.23	TLSv1.2	1434	Server Hello
39	2015-12-15 16:54:19.522415	109.105.109.234	10.192.36.23	TCP	94	[TCP segment of a reassembled PDU]
40	2015-12-15 16:54:19.522415	109.105.109.234	10.192.36.23	TCP	1434	[TCP segment of a reassembled PDU]
41	2015-12-15 16:54:19.522416	109.105.109.234	10.192.36.23	TCP	746	[TCP segment of a reassembled PDU]
42	2015-12-15 16:54:19.522493	10.192.36.23	109.105.109.234	TCP	66	57472 → 443 [ACK] Seq=183 Ack=1397 Win=129920 Len=0 TSval=532379401 TSecr=1261532124
43	2015-12-15 16:54:19.522493	10.192.36.23	109.105.109.234	TCP	66	57472 → 443 [ACK] Seq=183 Ack=3445 Win=127872 Len=0 TSval=532379401 TSecr=1261532124
44	2015-12-15 16:54:19.522575	10.192.36.23	109.105.109.234	TCP	66	[TCP Window Update] 57472 → 443 [ACK] Seq=183 Ack=3445 Win=131072 Len=0 TSval=532379401 TSecr=1261532124
45	2015-12-15 16:54:19.522671	109.105.109.234	10.192.36.23	TLSv1.2	609	Certificate
46	2015-12-15 16:54:19.522712	10.192.36.23	109.105.109.234	TCP	66	57472 → 443 [ACK] Seq=183 Ack=3988 Win=130520 Len=0 TSval=532379401 TSecr=1261532124
47	2015-12-15 16:54:19.529111	10.192.36.23	109.105.109.234	TLSv1.2	141	Client Key Exchange

Figure 4.8: Data packet analysis II - the screen shot highlights the two greeting messages

Beyond the micro-time dimension, packets are also spread across spaces. As Sprenger highlights in his earlier quote, “there cannot be any direct connections on digital networks” (2015, pp. 88-9). This implies that an actual network connection has more than two parties beyond the sender and receiver. According to the Protocol specifications (RFC 793 and RFC 791), there is a field called ‘Time to Live’ (TTL) that limits the lifespan of data within a connection (Postel, 1981a, p. 14; 1981b, p. 51). Between transmissions from both ends it is made up of multiple ‘hops’ and a hop refers to the following,

the leg of a route from one end system to the nearest switching computer, or between two adjacent switching computers, or from the switching computer to a connected end system (Meinel & Sack, 2013 p. 451).

Therefore, data packet routing means that a connection between a sender and receiver contains multiple switching computers and a route is made up of multiple hops. TTL is defined as the number of hops that a packet has to pass through before reaching its destination. This also means that if a



packet passes through more than a defined number of hops the packet is discarded, alluding to the time to die as opposed to live. Therefore each packet has its own lifespan and its own state of life or death. The idea behind having the TTL field is to prevent any instances of endless circulating of data packets within the network.

TTL is also indicated in the video streaming experiment (see Figure 4.9). The usual default value of TTL is 64 and the packet has travelled via 6 switching computers so as to reach the destination (where the value is indicated as 58 in Figure 4.9). For each switching computer, when it receives a data packet with a TTL value that is greater than 2, it will pass to the next and produce the decrement of 1 from the TTL value. A new TTL value is then sent to the next one. When a switching computer receives a data packet with a TTL value of 1, it means that the data packet has died. This is a form of checking mechanism to ensure the message will not route endlessly. The switching computer (usually in the form of a router) will inform the original sender if the value has exceeded the range. A forwarding machine will send back the message to the previous node as 'ICMP\_TTL\_TIME\_EXCEED.' Then the previous machine will traverse with the message 'ICMP\_DEST\_UNREACH' together with the code 'ICMP\_TIME\_EXCEED.' In this backward route each hop will increase the value of TTL by 1 and send to the destination (the original sender) (Rosen, 2014, p. 37). The checking mechanism and the decision of backward routing are monitored and executed in real-time.

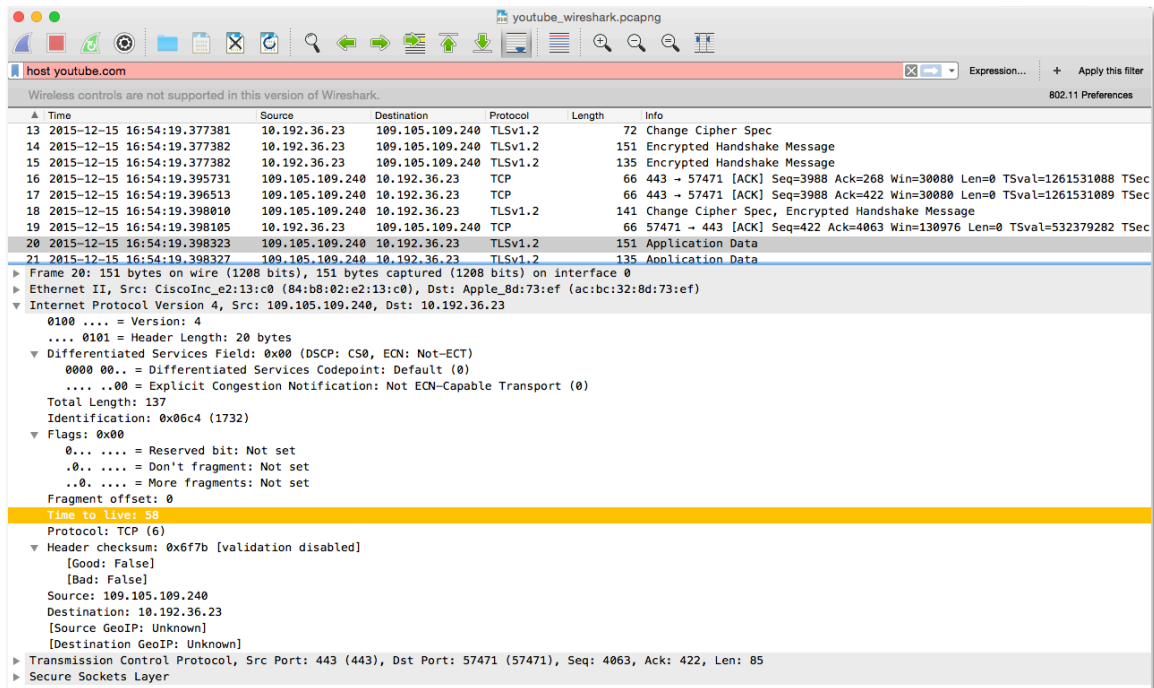


Figure 4.9: Data packet analysis III - the screen shot highlights the field ‘Time to Live’ for the data packet that transverses from the Youtube server to a local client computer.

This real-time execution is similar to what Chun describes within the context of hardware and software systems in which computation responds to the live conditions. She says,

hard and software real-time systems are subject to a ‘real-time’ constraint—that is, they need to respond, in a forced duration, to actions predefined as events. The measure of real time, in computer systems, is its reaction to the live—its liveness (Chun, 2008b, p. 316).

The notion of liveness can be understood as the decisions and reactions that are required to execute beneath various real-time constraints. From the timer implementation in protocols to TTL handling, Chun’s notion of real-time constraints is further extended to technological networks, the complex connection between machines across distributed space. The example of the timer and the field TTL highlight the timely responses, transmission and decisions mechanism within a temporal network. To Chun, liveness is expressed at the temporal level in which a system is required to react and

respond according to its user input and output. In the case of technological networks however, the response may not include direct human intervention and machines take charge of decisions in real-time and respond in a forced duration. The liveness of a networked system has to be understood through its micro-temporality. The micro-temporality of a stream involves decisions, as well as interruptions, in real-time. Every decision, the routing decision via multiple hops for example, takes time. Decisions are made not only in real-time but also at micro-temporal intervals. Therefore, every decision made within protocols can be thought of as a micro-interruption instead.

The interruption of a transmission process is further elaborated in Sprenger's notion of 'micro-decisions.' Through the analysis of TTL the role of the protocol is not simply to route and transmit data but it also computes, makes decisions and performs actions according to the value and logic that take place in real-time. As Sprenger puts it,

[there are] constant temporal interruptions, during which decisions are made about the further transmission of packets, and it situates the very stability of the network in these interruptions (2015, p. 86).

He highlights the fact that micro-decisions are made through temporal interruptions. This also implies the notion of that real-time, as we are able to understand it, is "never instantaneous" and data is not transmitted via multiple hops simultaneously. Each individual switching computer, situated in different continents, takes part in fulfilling the goal of information reaching the same destination. However, these micro-decisions also "interrupt the stream of data in order to control its distribution" (Sprenger, 2015, p. 19). One of the characteristics of such micro-decisions is their effectiveness, insofar as the mechanism is placed and decisions are made automatically without human monitoring or intervention. Importantly, as Sprenger remarks,

political and economic considerations have been made in the

background to these micro-decisions, because the technical development of digital networks starts with their implementation (2015, p. 20).

These micro-decisions are important to make data transmission possible. Although these are made beyond the human sensory perception, Sprenger reminds us that the so-called real-time transmission is always interrupted even if only a nanosecond of time is required. Therefore, the notion of a real-time connection is never the same as having an instant and direct contact to the connected world (Sprenger, 2015, p. 21). Such a connection is not a direct one between A to B but involves other inter-actions, such as switching computers, which are spatial as well as temporal. These are the active agents of a connection that can decide whether a packet is to live or to die. In other words, there are no streams that flow in a smooth and one-way direction, rather there are unplanned micro-interruptions in multiple, or even backward, directions within a network connection. When taking into account the technical details of network architecture and considering the micro-temporality of so-called streams, I argue that the notion of liveness and interruption cannot be separated and there is a coupling of the living and the dead forces. The experience of liveness consists of micro-interruptions that are not apparent to us. Consequently, the inter-actions of code execute decisions and produce micro-interruptions that engender the immanent live experience of watching a stream.

### *4.2.3 Buffer and Buffering*

Further to the understanding of the nature of digital signal processing, the fundamental component of clock cycle and the protocols of network transmission, this section focuses on the buffer to elaborate the time-critical matter of buffering, the time when one sees a throbber on a screen.

A buffer is understood as a temporal storage that usually stores a small amount of data in physical memory. Code is required to instruct the

temporary storage of data. As demonstrated in the previous section, every packet segment that is sent via protocols requires the receipt of an acknowledgment, allowing the sender side to know which packet segment has been successfully delivered. This is also a major part of a reliable protocol, meaning the protocol “must recover from data that is damaged, lost, duplicated, or delivered out of order by the Internet communication system” (Postel, 1981b, p. 4). Within a robust data flow control in protocols, the process of fragmentation with the checksum function that is inscribed in data packets makes it possible to reformulate a correct sequence that makes sense of the perceived content. This assembling process involves the use of a temporary buffer. Baran explains as follows:

On the transmitting end, the functions include chopping the data stream into packets, adding housekeeping information and end-to-end error control information to the out-going packets. On the receiving end, each multiplexing station uses terminating buffers temporarily assigned to each end addressee to unscramble the order of the arrived packets, and buffer them so that they come out as an error-free stream, only slightly but not noticeably delayed (2002, p. 46).

What is interesting here is the barely noticeable delay time that gives the perception and illusion of a stream. A number of questions arise. How does the protocol synchronise such a delay so that things become unnoticeable? What is flow control (or the end-to-end error control) and how does it enable reliable housekeeping, as in a sense of packets coming out as an error-free stream even though there is packet damage, information which is lost, duplicated or out of order? How do these perpetual and invisible ruptures in packets enable us to conceive of a stream of discontinuity but not a perception of continuity? What is the role of the temporary buffer and its read/write logic, leading to an inscribed stream? Ultimately, what makes the buffer temporal and produces differences and how does it inter-act?

To address these questions, an investigation of the flow control in TCP

architecture is necessary. One of the characteristics of a flow control is that it regulates the amount of data to be sent for each transmission through the concept of the 'Sliding Window Protocol Mechanism.' Window size refers to the buffer size capacity that indicates the maximum amount of data that can be buffered. This requires notification to a receiver. The sender then can only send the amount of data within the indicated window size. The value of the window size should be decreased every time a new segment of the packet arrives at the receiver. When the value drops to zero it means that the receiver's buffer size is full and the receiver is not able to handle any additional data at the moment. Unless the receiver has processed the buffered data and sent an 'ACK' message back to the sender the window size value will be increased accordingly. Theoretically, all the received segment packets must send an 'ACK' message in return. Then a sender can continue to send the remaining data segment. According to scientist Christoph Meinel and scholar Harald Sack, who specialise in internet systems and technologies, "the receiver regulates this maximum number over the sliding window protocol and adapts this size to fit its processing capacity" (2013, p. 610).

To explain in details Figure 4.10 illustrates the sliding window at the sender side, including a data segment that is "already sent but unacknowledged" and "not yet sent but ready to send." Once the sender receives the 'ACK' for message segment B1-B4 segment C1-C4 will transmit immediately, hence the sliding window will move to C1 and will cover through D4. In other words, the sliding window moves as acknowledgement arrives, consequently segment D will be included in the window. The sliding window adjusts the rate of data flow.

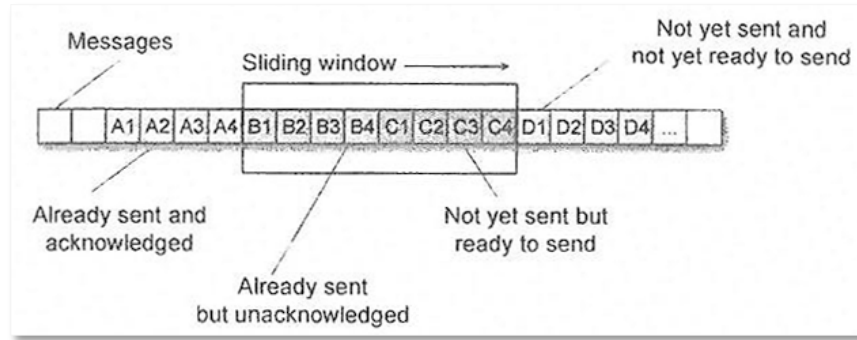
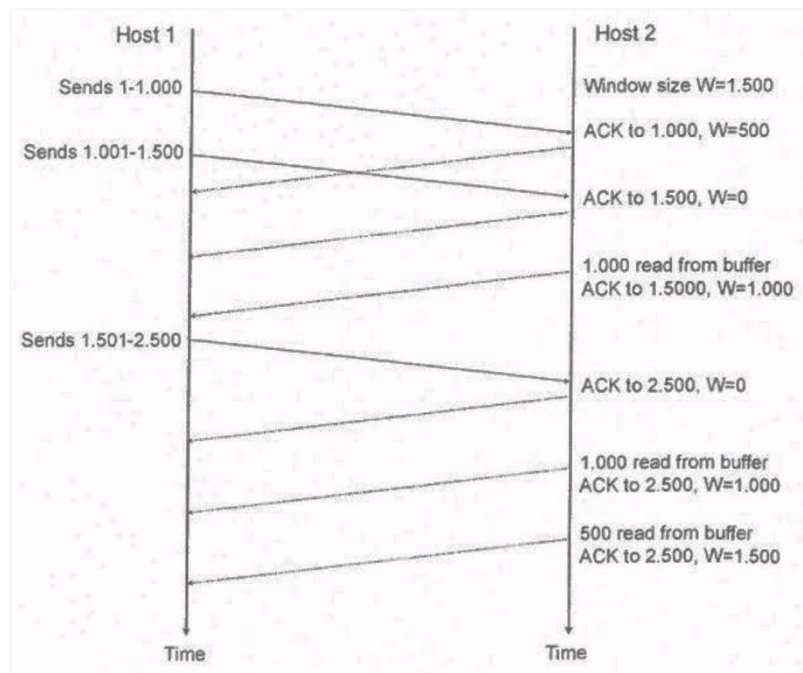


Figure 4.10: Sliding Window Protocol. Reprinted from Internetworking (p. 611) by C. Meinel & H. Sack, 2013, Berlin: Springer. Copyright 2013 by Springer.

Figure 4.11 demonstrates the flow control with the sliding window protocol more clearly. Assuming host 2 sets a window size  $W$  as 1500 and host 1 sends segment within 1-1000 then upon segments arriving at host 2, it sends an 'ACK' message with the value 1000, meaning the previous segments (1-1000) have been received. At this point host 1 sends other segments from 1001-1500 to host 2. Since it takes time for host 2 to process the segments 1001-1500 it sends an 'ACK' as 1500 but window size is set as 0, signaling the buffer capacity is full. It also means that host 1 cannot send any further segments to host 2 as it does not has any additional capacity to process. After host 2 has processed 1000 out of 1500 segments from the buffer host 2 sends an 'ACK' to host 1 with the new window value as 1000 because the remaining 500 is still under processing. Then host 1 sends another segment of 1501-2500 to host 2 and, once host 2 receives the segments, it immediately reports back with an 'ACK.' Again the window value is set to 0 because the buffer is now processing the old 500 segments as well as the new 1000 segments with its full capacity. The governing of the flow control takes place within the windows' restriction and the segment number in the 'ACK' field, and these are all determined during live processing. This flow control governs how (many), when and what acceptable data segments can be transmitted. When an error occurs during transmission it can be recovered by TCP via a retransmission mechanism. Data is automatically retransmitted<sup>125</sup> if no 'ACK' is received within a

<sup>125</sup> Another function calls 'checksum' in TCP that is also used to handle data disruption (cf. Meinel & Sack, 2013, pp. 134-5).

certain time interval.



*Figure 4.11:* TCP- flow control with the sliding window protocol. Reprinted from *Internetworking* (p. 612) by C. Meinel & H. Sack, 2013, Berlin: Springer. Copyright 2013 by Springer.

The spatial dimension of micro-temporality is further explicated beyond transmission from A to B or from host 1 to host 2. The movement of slicing windows that moves across data segments in which spaces are defined as “already sent and acknowledged,” “already sent but unacknowledged,” “not yet sent but ready to send,” as well as “not yet sent and not ready to send” (see Figure 4.10). The window slice moves across these spaces to control the outward message. Additionally, the dynamic size of a window is subject to the activity of reading and writing data in buffer memory. Window size is full when the data has filled the buffer memory and is waiting to be processed. Size is not a mere all (full) or nothing (empty) indicator but it can be varied dynamically when buffer data is processed partially or fully. The window size also impacts the counting logic of what is to be sent and not to be sent from the sender’s perspective. By referencing the window size from the receiver the counting logic is automatically adjusted. Importantly, such sliding window protocols constitute the mechanisms of data flow control in which there are spatial-temporal assemblages that render a data stream.



In the process of data buffering there are micro-decisions also take place. From the previously mentioned decisions of locating an efficient path to the movement of the window slice and further in relation to which segments of data to send, all of these decisions are made to control how data is transmitted. Even though it may not seem significant time is lost along the journey (Sprenger, 2015, p. 75). This journey involves interruption at different times and in different spaces, as Sprenger remarks, “[t]he stream never flows uninterruptedly” (2015, p. 19). This constant interruption constitutes the notion of micro-temporality that is discontinuous and includes decision-making processes, controls and regulations that are programmed at the level of protocol and are inscribed in the stream. When micro-decisions take account of rules in network environments, a stream does not unfold continuously, but rather like what philosopher Jacques Derrida refers to as the “continuous unfolding of a calculable process” (2002, p. 252).

Buffering is a calculable process in which data is divided into segments and packets. From the articulation above with the processing of window buffer, as well as the respective input and output of the buffer, these involve the activities of storing, reading and processing. It is worth noting that the activities are not acting on the same bit and piece of data. While some data is stored in a buffer, other segments of data are being read and processed. From a system perspective, data is processed at the receiver’s end (as input data in the buffer) and is stored temporarily and locally until the data is further processed by the software application (as output data). This also means that software applications are not required to wait for the entire media file to be downloaded. ‘Just in Time’ (JIT) delivery is used in streaming media, allowing for the playback of partially received data temporarily stored in the client’s buffer (Pereira & Ebrahimi, 2002, p. 260). In this sense both the playback of buffer data and the receiving of the remaining data can be made simultaneously (and, in addition to the case of video and audio, this is also commonly experienced in loading any relatively

large size file, such as a PDF or an image<sup>126</sup> within a browser). Software applications, like a browser or media player, have the capability to load or play the partially downloaded data. The buffer is where software applications access the input data and process it as output data. In other words, the processing of data consists not only of the transferring part but rather, as Ernst reminds us, through “a coupling of storage and transfer in realtime.” He continues, “[w]hile we see one part of the video on screen, the next part is already loaded in the background” (Ernst, 2006, p. 108). More precisely, the viewer is not watching the content as data arrives, instead the viewer is watching the processed data that has arrived and is stored in the buffer. This process of temporal storage and playback gives us an understanding of the relation between buffer and stream, in which there is latency between data arrival (from the network), data storage (within internal memory) and data processing (inside a machine). Streaming is essentially “achieved by buffering the transmitted data before the actual display” (Meinel & Sack, 2013, p. 780). A throbber is entangled with this latency, inter-acting with different pieces of data in different ways.

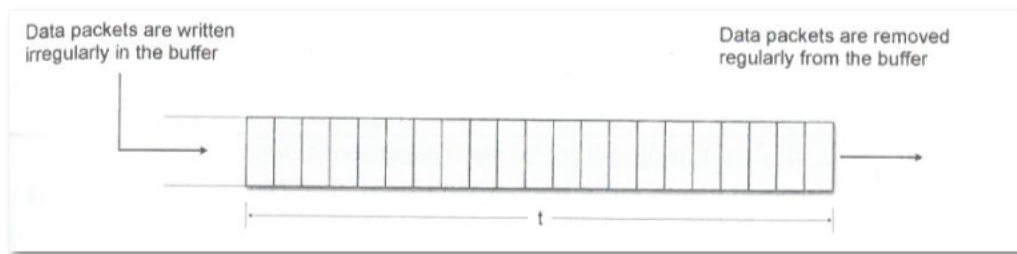
Ideally, the “buffer empties itself at one end just as quickly as it fills up at the other end,” as described by Meinel and Sack (2013, p. 783). If there is transmission delay that is within a threshold time  $t$ , it is regarded as unnoticeable in playback. However, if the delay of the individual segment exceeds the threshold time  $t$  a throbber will then display. A program attempts read and process the buffer but the data hasn’t arrived yet and this gap and rupture will lead to the appearance of a throbber. When this occurs we can perceive and experience discontinuous micro-temporality.

Normally a throbber is seen when loading a big chunk of data and is commonly seen in video sites mostly due to the instability or low bandwidth of a network which causes a delay in the arrival of a data segment (i.e. it exceeds the threshold time  $t$ ). Buffering is highly related to time as it allows

---

<sup>126</sup> The relation between large file sizes and the buffer has been explored in my previous artistic project *How to get Mao* experience through Internet...(2014), see: <http://maoexperiencethroughinternet.siusoon.net/>

different rates to occur simultaneously, decoupling “time dependencies” between the input and output of data (Laplante, 2000, p. 55). As a result, data can be consumed and processed at a different rate by program applications. Data, in the case of streaming, is actively and constantly being stored (written) and removed (read) in the buffer at different speeds and rhythms (see Figure 4.12), oscillating between the invisible and visible. With Parisi’s notion of “temporal variations” (see Chapter 1), it can be said that the micro-temporality of buffering transforms the space of a buffer that works with both internal and external data. This buffer space, as a site of inter-actions, contingently and temporally performs variations.



*Figure 4.12:* Principle organization of a playback buffer. Reprinted from *Internetworking: Technological Foundations and Applications* (p. 783), by Christoph Meinel & Harald Sack, 2013, Berlin: Springer. Copyright 2013 by Springer.

#### 4.2.4 *The absence of data*

Proceeding from the operative logic of streaming this chapter has demonstrated that there are calculable processes, data transmissions and that the reading and writing of the buffer occurs at different rates. The operative logic is built into the infrastructure of networked technology as code. What has been written in the buffer will be automatically read and processed through running code. However, technology does not guarantee that all the data is written in the buffer. Dropped frames (frames of video that are dropped during playback) are a relatively common experience in real-time communications and video streaming. Dropped frames impact upon the user’s viewing experience because frames disappear within a perceivable continuous stream. When an audio-visual is played back at the receiver’s side this introduces gaps in the stream and it produces glitches or

jittery audible effects. This is different from displaying a throbber on a screen, where nothing can be seen on a screen despite the animated graphic. When experiencing dropped frames, one can still see or hear something but just not necessarily in good quality.

In some situations the issue of dropped frames is seamless because it does not create significant quality degradation. Such visible and invisible dropped frames are caused by packet loss, the absence of certain parts of data during data transmission across nodes and routers throughout the journey. Time lost, as mentioned above, includes micro-decision making as well as interruptions and delays. Indeed, packet loss is highly relevant to the notion of micro-temporality. According to information and computer science scholars James F. Kurose and Keith W. Ross, the delay time for transmitting data not only includes ‘store-and-forward’ in each buffer nodes but also ‘queuing delays’ that are subjected to network congestion and are not predictable in advance (2013, p. 25). Packets are required to queue up and wait for the transfer while the network is congested. Under streaming conditions, data is constantly transmitted from a sender to receiver across multiple sites. However, the amount of buffer space is limited at each site, which means a newly arriving packet potentially has no space to be stored while the stored packet is still queuing for its next routing. In this situation, “packet loss will occur-either the arriving packet or one of the already-queued packets will be dropped” (Kurose & Ross, 2013, p. 25).

The robust design of network protocols consists of an automatic mechanism to detect and trigger retransmission for packet loss. However, for real-time conversational applications and media streaming platforms such as Skype and YouTube delay time for each packet is a critical issue as the transmission is required to be continuous. Both conversations and live concerts are unceasing. On the one hand the absence of data is crucial as packet loss is related to the degradation of quality and it could immediately impact the visual or audio quality in a live environment. On the other hand, if data arrives with significant delay the application design at the receiver’s end is then required to determine if such data will still make sense in

playback, in particular where conversation and data are constantly played-back as a stream. In deciding whether the data should be played-back or ignored, acceptable latency becomes a decision that is inscribed in the software and platform design.

Serious data loss may even result in the automatic termination of a connection—which also means the tolerance is unacceptable from the point of view of software design. The technical consequences of data loss is nothing new. If one has used Skype or other communication applications like WhatsApp, weChat or Line it is not uncommon to have the experience of glitches or jitter effects or a throbber displayed on a screen. What is of concern here is rather the cultural implications of these absence of data, or the potentiality of packet loss.

The absent data requires our attention. Firstly, the absence of data might be caused by a voluntary condition. It is possible for an application to discard late-arriving data that are within acceptable latency because it is insignificant to the entire user experience. Secondly, due to buffer capacity, data loss can occur anytime and at any site during the entire journey of a data transmission. Last but not least, when the network bandwidth cannot match the application's processing rate there will be data loss. For example, a 50% data loss is encountered when a network has only a maximum bandwidth of 5 Mbps and the application requires 10 Mbps. When there is insufficient capacity to handle different rates (the input and output buffer rates at receiver's end - see Figure 4.12), and hence, data loss will occur (Claypool & Riedl, 1998, p. 882). As a result not all data is treated equally and able to arrive at the destination and take a perceptible form. Even though the presence of a stream is mediated as audio and visuals through a screen there is still a possible absence of data. The absence of data, although it cannot be mediated in a perceptible form at the receiver's end, is somehow interwoven into the presence of a stream in which a conversation or video playback is kept running. The point is that the mundane activity that we experience when we load, wait and stream through a screen is loaded with unperceivable gaps.

To explain further I will briefly mention an artwork called *The Pirate Cinema*<sup>127</sup> (2012-2014), developed by Nicolas Maigret, Brendan Howell and Jean-Marie Bover, which pays similar attention to absent data, glitches, jitters and the micro-temporality of the buffer. Situated in a peer-to-peer sharing network, the artwork reveals the geo-political and legal aspects of file transfer (torrent). The glitches and jittery audio effects (see Figure 4.13) clearly show the fragmented bits and pieces of the real-time stream. Instead of a smooth showing of a stream the work demonstrates patterns of discontinuity between absence and presence. Perhaps these discontinuous patterns have something to do with performativity in which realities are enacted into being and political expressions are performed. Cox discusses this specific artwork in relation to Sergei Eisenstein's theory of montage, where different *realities* are can be seen as material constructions. These realities incorporate temporal and disorderly fragments from multiple spaces and times (Cox, 2015, pp. 8-9) whereby their presence builds upon a series of absences in a distributed network. In other words, the work is conditioned by these absences and such a focus on temporalisation is infrastructural specific and non-discursive, stemming from the materiality of the distributed network that is running dynamically. According to the science, technology and society scholars John Law and Vicky Singleton, when one experiences a reality, "whatever the form of its presence, this also implies a set of absences" (2005, p. 342). Their notion of presence and absence, as in this chapter, are understood as an entanglement not as a separate concept, in which "sets of present dynamics generated in, and generative of, realities that are necessarily absent" (Law & Singleton, 2005, p. 343).

---

<sup>127</sup> See: <http://thepiratecinema.com/>



*Figure 4.13: The Pirate Cinema (2012-2014). Image is retrieved from <https://www.flickr.com/photos/61131081@N03/15548817658/>. Copyright 2014 by Nicolas Maigret.*

Similarly, when a stream of data is sent via a distributed internet network the logic of buffering and data processing are constantly performed through the presence and absence of data. A display of a throbber presents another reality, a reality that is conflated with an invisible material infrastructure and interruptions as well as the absence of material substrates. They are both micro-temporal as I have already demonstrated. Furthermore, a throbber and its underlying logic of data buffering involve discrete-time signaling—the milliseconds of time lost and the absence of data—presenting multiple realities which lie at the heart of time-dependent logics. Therefore, reality is not only a matter of continuous flow and the immediacy of a stream. Taking account of materiality such a notion of reality neither refers to the symbolic meaning of content, nor the feeling of presence, nor immediate data delivery but rather to a tension that is expressed between continuity and discontinuity through the performativity of code interactions.

This is to say that when taking into account packet loss, the liveness or nowness of a stream is also about an absent present. The notion of discontinuous micro-temporality explicates the invisibility of computational culture by shifting our attention from the cultural understanding of a throbber and what is visible on a screen to invisible micro-events that are running in the background; events are not separated but entangled as absent/present.

Absent data is rarely mentioned in the commercial products that frame contemporary digital culture inasmuch as it possibly relates to quality degradation or may be regarded as unnoticeable. Technically there are some parts of a stream not necessarily perceptible even though data is continuously being sent. Within a stream there are these discontinuous forces constituting a continuous presence. Sometimes the forces seem to be strong but other times they are weak; in some case they are more visible and at other times they are unnoticeable. The notion of discontinuity pays attention to the gaps, ruptures and pauses that are interwoven within the continuous flow of a data stream. From the display of a running throbber to its disappearance while a stream is presented, *discontinuous micro-temporality* highlights the forces and presence of micro-decisions and micro-interruptions that reconfigure the liveness of a stream.

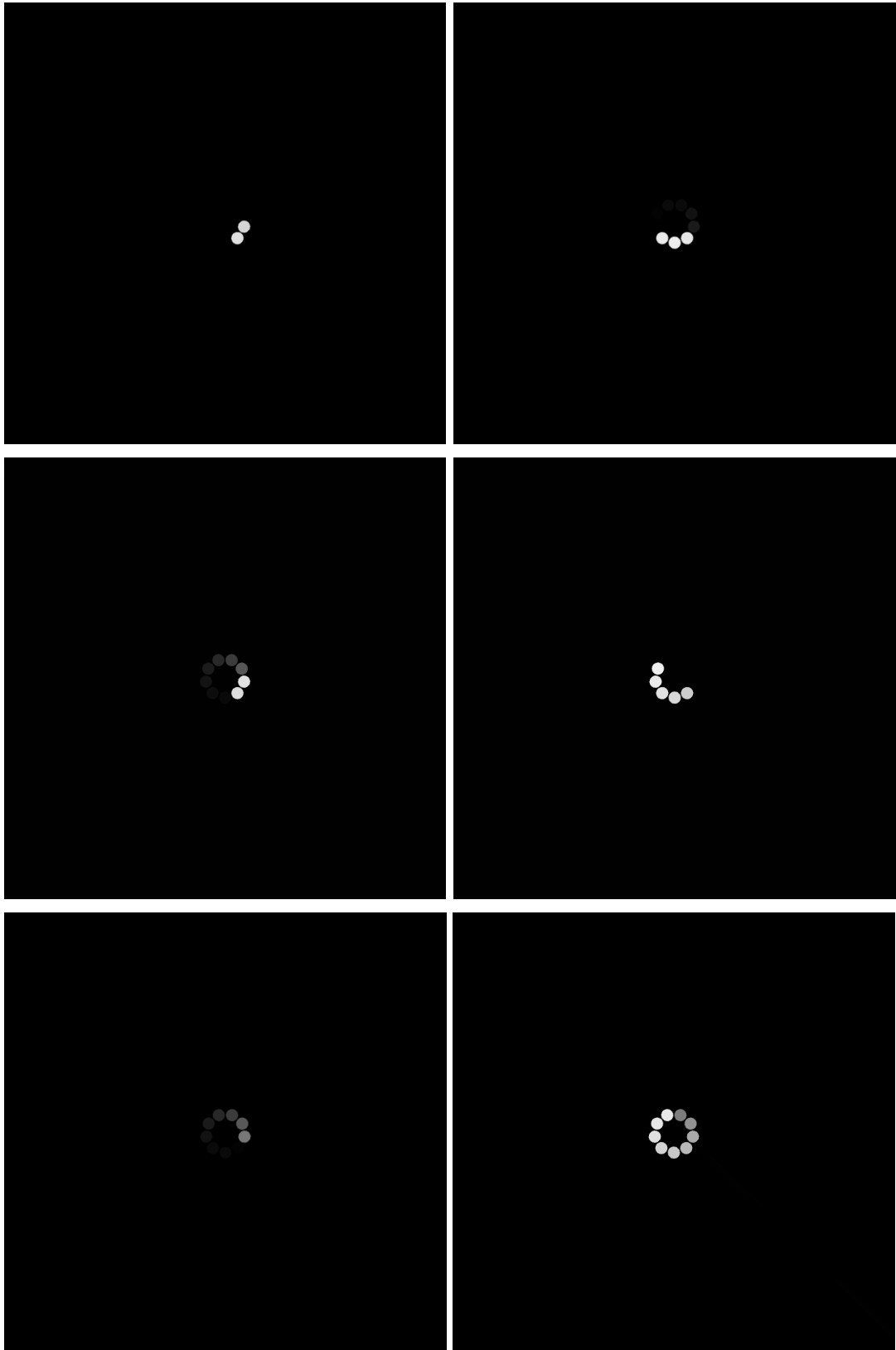
### **4.3 The Spinning Wheel of Life**

Such a reflection of invisibility and performativity, underpinned by material substrates and their time-dependent logics, is made apparent in my artistic project *The Spinning Wheel of Life*. The project emphasises the micro-temporal dimension of code inter-actions that are manifested in the throbber. The title of the project is borrowed from a ‘wait cursor’ in the Macintosh Operating System X designed by Apple. The wait cursor is colloquially known as ‘The Spinning Wheel of Death,’ referring to the malfunction or failure of a running program or a system that leads to a screen freezes. The name takes on negative connotations as the problems are usually difficult to diagnose. The reference to the Spinning Wheel of



Death invokes problems, failure and interruption. However, *The Spinning Wheel of Life* asserts that interruption exists as a reality across multiple sites where micro-decisions are constantly executed. A packet segment, as demonstrated, has its own lifespan and its own state in terms of life and death. However, a stream is potentially infinite insofar as data is continuously generated and updated. A stream is perceived as a continuous flow with an unforeseeable end and imperceptible interruptions, yet a packet segment *has* a pre-set timespan. The project highlights the paradoxes and tensions between continuous and discontinuous processes, between end and endless, finite and infinite states. These relations suggest a blurring of lines and that constitute the forces of liveness and deadness, a coupling of the present and absent, the living and the dead.

Figure 4.14-4.19 below document the animated movements of *The Spinning Wheel of Life*. The visual reacts to network packets generated from running a YouTube playlist in real-time. The work consists of a throbber that is animated at different rates. Each ellipse within the throbber represents a new data packet arrival. The time for each fading ellipse is adjusted to an optimal level in which a balance of the visual composition is archived. Since packets arrive at multiple times and spaces and sometimes a huge amount of packets arrive at almost the same time the visual throbber yields an unusual and uneven spinning wheel—from having just a few ellipses to a full throbber with all the ellipses displayed brightly. Each ellipse fades in and out along different tempos, subject to the network conditions in real time. There are different rates, tempos, pulses, pauses and rhythms at multiple scales—from the operations of the CPU to network routers, from the transmissions of senders to receivers, from the writing to the reading of buffers and from continuous streams to discontinuous packets. Time is an important element in contemporary software culture as it governs how a signal is processed, how data is transmitted and flows and how micro-decisions are made. As a result a stream is constantly being interrupted from the start of data transmission not just at the time one encounters a throbber animating on a screen. The project makes apparent the underlying notion of discontinuous micro-temporality.



*Figure 4.14-4.19: The animated visuals of *The Spinning Wheel of Life* (2016).*

The project is still work-in-progress and would ideally be presented as an installation consisting of a series of mini setups. Each mini setup (see

Figure 4.20 for the work-in-progress prototype) displays a throbber on a screen and is attached to a small speaker that plays a stream of 8-bit music from the YouTube platform. An internet connection is required to allow the data to unfold in real-time. The playlist is set to repeatedly run and configured to block advertisements, a form of non-experiential interruption or “natural break(s)” (Williams, 1974, pp. 90-3) between songs. The setup simulates the cultural logic of buffering and its representation in its most familiar form as a throbber. The project offers an alternative experience in order to speculate the micro-temporality of code that inter-acts in a live networked environment. The work does not set out to explain or describe how things work, instead it enacts the discontinuous micro-temporality of a throbber through the performativity of code inter-actions.



Figure 4.20: The mini setup and work-in-progress of *The Spinning Wheel of Life* (2016).

A stream is manifested as continuously updating feeds, passing through hops and sites, which in part define the *now* in how we experience the world. *The Spinning Wheel of Life*, drawing references from the mundane throbber and the Spinning Wheel of Death and calls for critical attention towards these live and mediated processes, not only at a planetary scale but also at

the level of micro-temporal operations, including clock cycles, instruction execution, packet switching and data buffering, which exhibit micro-decisions and micro-interruptions. The notion of discontinuous micro-temporality takes into account the micro-processes, gaps and ruptures and, more importantly, the absence of data that renders multiple realities that are at work. This sheds light on the understanding of streams in computational culture; in particular, on how time is processed and organised to present the now under live conditions.

In other words, the artwork is a reflection on perpetually changing cultural and social conditions. On the one hand, the existence of a throbber is a by-product of a commercial application that informs users to wait for an unknown period of time. On the other hand, through the use of a throbber in developing various data query services, such as live streaming, big data analysis, social media platforms, data predictions and transactional applications, this cultural icon offers a critical space for speculating and reflecting on how time is being organised and how the now is presented and made operative. A throbber is a cultural phenomenon that appears in almost every application that operates within a live computational environment. *The Spinning Wheel of Life* is not only a technical or visual object but is also entangled with other cultural and technical processes that render the unknowable more knowable. This chapter explicates the computational logic behind a throbber through the vector of micro-temporality. The next chapter will continue to investigate live processes, exploring another vector, automation, which examines the constitution of liveness in contemporary software culture.

## Notes on Reflexive Coding Practice

### 4.4 The Spinning Wheel of Life

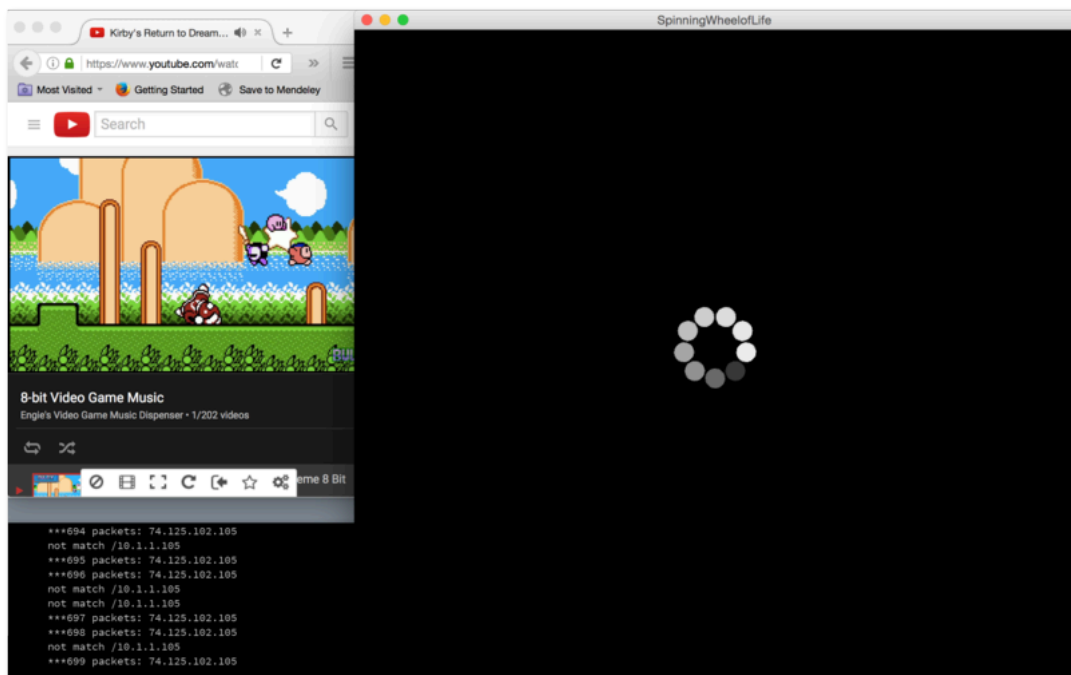


Figure 4.21: The Spinning Wheel of Life (work-in-progress) (2016)

The project of *The Spinning Wheel of Life* is a discovery-led process (Borgdorff, 2011, p. 56), my initial interest for this chapter was to discuss distributed networks, in particular how data is organised and transferred through the BitTorrent protocol, a form of peer-to-peer technology. I was very much inspired by the artwork and performance called *The Pirate Cinema* which I mentioned in the previous section because that work reveals some of the underlying operative processes of peer-to-peer technology. As I started to examine the technology I was drawn into the organisation of data that the format, which is called magnet link, uses to guard against corrupted or dummy files. From there I began to explore the ruptures of communication networks and the throbber icon caught my attention.

I was intrigued by this minimalistic yet iconic image which has been widely used in digital culture. In the words of Borgdorff, “[t]he discourses about art, social context and the materiality of the medium are in fact partially constitutive of artistic practices and products” (Borgdorff, 2011, p. 56). Therefore, the earlier section that offered a cultural reading of a throbber was established. At the beginning all I knew about the term buffering was that it dealt with data in the background. From my own experience when there is a slow internet connection I have a higher chance of seeing this icon. This was all I knew. In this chapter, my research was first guided by the image of the throbber then its visual design and front-end display logic and finally the backend operative mechanism. In other words, my research and practice on throbber went through the reflexive inquiry process of “identifying relevant factors, components, or systems” to answer basic questions such as why? Or what is it? (Sullivan, 2010, p. 110). I first approached the throbber through visual research, collecting different throbbers on the web and thinking about the difference between a throbber and a progress bar. I was surprised about how little detail the throbber icon conveys to users. From there, the investigation led me to examine the history and the practice around using a throbber. My thinking about visual language, production and history of the throbber comes close to what Rolling Jr discusses as part of the reflexive practice, specially “thinking in a language” (Rolling Jr, 2014, p. 163). The visual object, the animated throbber sign, is used as a means to investigate meaning around the cultural practice of a throbber.

I looked at the source code of websites to understand how a throbber is displayed on a screen. After understanding that a throbber is normally presented as an animated image

through various experiments (see Figures 4.22-4.27), I started to think of ways to design a throbber differently. In this way, it can be said that the experiment is a “thought experiment,” alluding to “the creation of possibilities over the proving of certainties” (Rolling Jr, 2014, p. 162). This led me to use mathematics to describe a moving throbber and use programming to adjust the speed of a throbber and, ultimately, to control each ellipse’s appearance which I have discussed in the earlier section (see Figure 4.3). Through coding practice, I demonstrate the process of “thinking in a material” via experimentation (Rolling Jr, 2014, p. 163) that informs my understanding of how a throbber can be created differently beyond otherwise an animated image that only expresses a regular tempo.

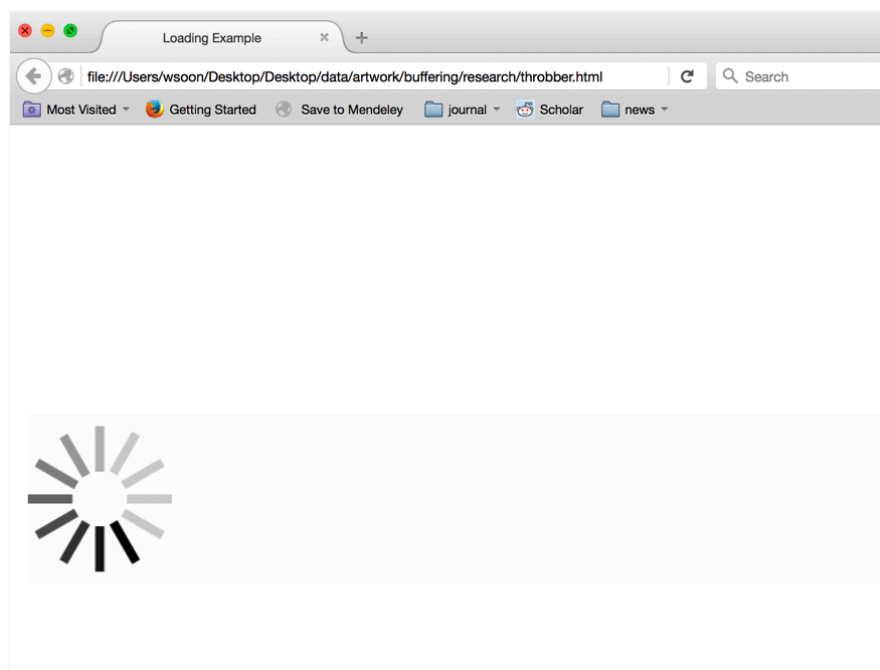


Figure 4.22: Experiment on how a throbber displays on a browser

```

1 <!DOCTYPE html>
2 <html class="loading">
3 <head>
4 <meta charset="utf-8">
5 <title>Loading Example</title>
6 <style>
7   html {
8     -webkit-transition: background-color 1s;
9     transition: background-color 1s;
10  }
11  html, body { min-height: 100%; }
12  html.loading {
13    background: #FFFFFF url('http://softwarestudies.projects.cwi.au/images/fff/Throbber_header.gif') no-
14    -webkit-transition: background-color 0;
15    transition: background-color 0;
16  }
17  body {
18    -webkit-transition: opacity 1s ease-in;
19    transition: opacity 1s ease-in;
20  }
21  html.loading body {
22    opacity: 0;
23    -webkit-transition: opacity 0;
24    transition: opacity 0;
25  }
26  button {
27    background: #00A3FF;
28    color: white;
29    padding: 0.2em 0.5em;
30    font-size: 1.5em;
31  }
32 </style>
33 <script>
34 var html = document.getElementsByTagName('html')[0];
35 var removeLoading = function() {
36   // In a production application you would remove the loading class when your
37   // application is initialized and ready to go. Here we just artificially wait
38   // 3 seconds before removing the class.
39   setTimeout(function() {
40     html.className = html.className.replace(/loading/, '');
41   }, 3000);
42 };
43 removeLoading();
44 </script>
45 </head>
46 <body>
47 <!-- AVERT YER EYES. This button is a hack to demo this. Do not use onclick attributes. -->
48 <button onclick="html.className = 'loading'; removeLoading();">Reload</button>
49
50 <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer nec odio. Praesent libero. Sed cursus ante dap-
51
52 <p>Sed dignissim lacinia nunc. Curabitur tortor. Pellentesque nibh. Aenean quam. In scelerisque sem at dolor. Maec-
53 </body>

```

Figure 4.23: Experiment on a throbber display with HTML, CSS and JS script

```

throbber.sh
#!/bin/bash
#reference: http://fitnr.com/showing-a-bash-spinner.html

throbber()
{
  local pid=$1
  local delay=0.175 #0.75
  local spinstr='|/-\`
  while [ "$(ps a | awk '{print $1}' | grep $pid)" ]; do
    local temp=${spinstr#?}
    printf " [%c]  " "$spinstr"
    local spinstr=$temp${spinstr%"$temp"}
    sleep $delay
    printf "\b\b\b\b\b\b"
  done

}
printf "Wow in progress..."
sleep 50 & throbber $! #how long it loads

```

Figure 4.24: A slightly modified version of the Unix shell script



```
d17018:research wsoon$ bash throbber.sh
Wow in progress... [\] █
```

*Figure 4.25: First screenshot of running the Unix shell Script*

```
d17018:research wsoon$ bash throbber.sh
Wow in progress... [-] █
```

*Figure 4.26: Second screenshot of running the Unix shell Script*

```
d17018:research wsoon$ bash throbber.sh
Wow in progress... [||] █
```

*Figure 4.27: Third screenshot of running the Unix shell Script*

After further experiments, I gradually shifted my attention from peer-to-peer file organisation to internet data organisation because this is the basis on which today's technology, like blockchain databases and many other peer-to-peer networks operate. I started to think about the notion of temporality through context, interrogating the technical infrastructure of our social structures (Rolling Jr, 2014, p. 163). I looked into internet data packets by using a simple 'tcpdump' command in my terminal application. Then I further experimented with different parameters of the command in order to capture more of the data logged behind data transmission (see Figures 4.28 – 4.29).

```

Last login: Thu Sep 17 18:48:49 on tty800
b03215~ wsoons tcpdump host google.com
tcpdump: loc[...]/bin/tcpdump: net permitted
b03215~ wsoons sudo tcpdump host google.com
tcpdump: data link type PKTAP
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
Listening on pktap, Link-type PKTAP (Packet Tap), capture size 65535 bytes
18:50:10.568940 IP 10.192.75.9.56925 > arn06s07-in-f110.1e100.net.http: Flags [S], seq 774259016, win 65535, options [mss 1460,nop,wscalc 5,nop,nop,TS val 730274614 ecr 0,sackOK,eol], length 0
18:50:10.601551 IP arn06s07-in-f110.1e100.net.http > 10.192.75.9.56925: Flags [S.], seq 3821528132, ack 774259017, win 42540, options [mss 1380,sackOK,TS val 426994986 ecr 730274614,nop,wscalc 7], length 0
18:50:10.601598 IP 10.192.75.9.56925 > arn06s07-in-f110.1e100.net.http: Flags [.] , ack 1, win 4184, options [nop,nop,TS val 730274645 ecr 426994986], length 0
18:50:10.601762 IP 10.192.75.9.56925 > arn06s07-in-f110.1e100.net.http: Flags [P.] , seq 1:1807, ack 1, win 4184, options [nop,nop,TS val 730274545 ecr 426994986], length 1076
18:50:10.631218 IP arn06s07-in-f110.1e100.net.http > 10.192.75.9.56925: Flags [.] , ack 1077, win 350, options [nop,nop,TS val 426995016 ecr 730274645], length 0
18:50:10.652312 IP arn06s07-in-f110.1e100.net.http > 10.192.75.9.56925: Flags [P.] , seq 1:547, ack 1077, win 350, options [nop,nop,TS val 426995037 ecr 730274645], length 546
18:50:10.652367 IP 10.192.75.9.56925 > arn06s07-in-f110.1e100.net.http: Flags [.] , ack 547, win 4086, options [nop,nop,TS val 730274695 ecr 426995037], length 0
18:50:11.686669 IP 10.192.75.9.56907 > arn06s07-in-f110.1e100.net.https: Flags [P.] , seq 3818621493-3818621855, ack 2564618268, win 4518, options [nop,nop,TS val 730275673 ecr 426954099], length 362
18:50:11.668714 IP 10.192.75.9.56907 > arn06s07-in-f110.1e100.net.https: Flags [P.] , seq 362:509, ack 1, win 4518, options [nop,nop,TS val 730275673 ecr 426954099], length 147
18:50:11.701721 IP arn06s07-in-f110.1e100.net.https > 10.192.75.9.56907: Flags [.] , ack 362, win 566, options [nop,nop,TS val 427000534 ecr 730275673], length 0
18:50:11.702364 IP arn06s07-in-f110.1e100.net.https > 10.192.75.9.56907: Flags [.] , ack 509, win 587, options [nop,nop,TS val 427000535 ecr 730275673], length 0
18:50:11.896818 IP arn06s07-in-f110.1e100.net.https > 10.192.75.9.56907: Flags [P.] , seq 1:216, ack 509, win 587, options [nop,nop,TS val 427000729 ecr 730275673], length 215
18:50:11.896824 IP arn06s07-in-f110.1e100.net.https > 10.192.75.9.56907: Flags [P.] , seq 216:344, ack 509, win 587, options [nop,nop,TS val 427000729 ecr 730275673], length 128
18:50:11.896857 IP 10.192.75.9.56907 > arn06s07-in-f110.1e100.net.https: Flags [.] , ack 216, win 4511, options [nop,nop,TS val 730275890 ecr 427000729], length 0
18:50:11.896878 IP 10.192.75.9.56907 > arn06s07-in-f110.1e100.net.https: Flags [.] , ack 344, win 4507, options [nop,nop,TS val 730275890 ecr 427000729], length 0
18:50:11.901472 IP arn06s07-in-f110.1e100.net.https > 10.192.75.9.56907: Flags [P.] , seq 344:517, ack 509, win 587, options [nop,nop,TS val 427000734 ecr 730275673], length 173
18:50:11.901508 IP 10.192.75.9.56907 > arn06s07-in-f110.1e100.net.https: Flags [.] , ack 517, win 4512, options [nop,nop,TS val 730275903 ecr 427000734], length 0
18:50:11.901532 IP arn06s07-in-f110.1e100.net.https > 10.192.75.9.56907: Flags [P.] , seq 517:563, ack 509, win 587, options [nop,nop,TS val 427000734 ecr 730275673], length 46
18:50:11.901552 IP 10.192.75.9.56907 > arn06s07-in-f110.1e100.net.https: Flags [.] , ack 563, win 4511, options [nop,nop,TS val 730275903 ecr 427000734], length 0
18:50:11.901599 IP arn06s07-in-f110.1e100.net.https > 10.192.75.9.56907: Flags [P.] , seq 563:593, ack 563, win 4518, options [nop,nop,TS val 730275903 ecr 427000734], length 46
18:50:11.934213 IP arn06s07-in-f110.1e100.net.https > 10.192.75.9.56907: Flags [.] , ack 555, win 587, options [nop,nop,TS val 427000767 ecr 730275903], length 0
18:50:12.149819 IP 10.192.75.9.56906 > arn06s07-in-f110.1e100.net.https: Flags [.] , ack 64627099, win 4096, length 0
18:50:12.184818 IP arn06s07-in-f110.1e100.net.http > 10.192.75.9.56906: Flags [.] , ack 1, win 352, options [nop,nop,TS val 427004008 ecr 730215887], length 0
18:50:20.757847 IP 10.192.75.9.56925 > arn06s07-in-f110.1e100.net.http: Flags [.] , ack 547, win 4096, length 0
18:50:20.788976 IP arn06s07-in-f110.1e100.net.http > 10.192.75.9.56925: Flags [.] , ack 1077, win 350, options [nop,nop,TS val 427005174 ecr 730274695], length 0
18:50:22.293238 IP 10.192.75.9.56906 > arn06s07-in-f110.1e100.net.https: Flags [.] , ack 1, win 4096, length 0
18:50:22.328148 IP 10.192.75.9.56925 > arn06s07-in-f110.1e100.net.https: Flags [.] , ack 547, win 4096, length 0
18:50:20.802972 IP 10.192.75.9.56925 > arn06s07-in-f110.1e100.net.http: Flags [.] , ack 547, win 4096, length 0
18:50:30.835439 IP arn06s07-in-f110.1e100.net.http > 10.192.75.9.56925: Flags [.] , ack 1077, win 350, options [nop,nop,TS val 427015220 ecr 730274695], length 0
18:50:32.343168 IP 10.192.75.9.56906 > arn06s07-in-f110.1e100.net.https: Flags [.] , ack 1, win 4096, length 0
18:50:32.378133 IP arn06s07-in-f110.1e100.net.http > 10.192.75.9.56906: Flags [.] , ack 1, win 352, options [nop,nop,TS val 427042402 ecr 730215887], length 0
18:50:40.957925 IP 10.192.75.9.56925 > arn06s07-in-f110.1e100.net.https: Flags [.] , ack 547, win 4096, length 0
18:50:40.990219 IP arn06s07-in-f110.1e100.net.http > 10.192.75.9.56925: Flags [.] , ack 1077, win 350, options [nop,nop,TS val 427025375 ecr 730274695], length 0
18:50:42.492554 IP 10.192.75.9.56906 > arn06s07-in-f110.1e100.net.https: Flags [.] , ack 1, win 4096, length 0
18:50:42.435874 IP arn06s07-in-f110.1e100.net.http > 10.192.75.9.56906: Flags [.] , ack 1, win 352, options [nop,nop,TS val 427034259 ecr 730215887], length 0
18:50:51.308165 IP 10.192.75.9.56925 > arn06s07-in-f110.1e100.net.https: Flags [.] , ack 547, win 4096, length 0
18:50:51.340284 IP arn06s07-in-f110.1e100.net.http > 10.192.75.9.56925: Flags [.] , ack 1077, win 350, options [nop,nop,TS val 427035726 ecr 730274695], length 0
18:50:52.457814 IP 10.192.75.9.56906 > arn06s07-in-f110.1e100.net.https: Flags [.] , ack 1, win 4096, length 0
18:50:52.492137 IP arn06s07-in-f110.1e100.net.http > 10.192.75.9.56906: Flags [.] , ack 1, win 352, options [nop,nop,TS val 427044316 ecr 730215887], length 0

```

Figure 4.28: Experiment with the command ‘tcpdump’ for networked data analysis

```

wsoons ~ bash - 203x38
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
Listening on pktap, Link-type PKTAP (Packet Tap), capture size 65535 bytes
16:21:09.303799 IP 10.192.36.23.65322 > 109.105.109.208.443: Flags [S], seq 2785969751, win 65535, options [mss 1460,nop,wscalc 5,nop,nop,TS val 388081391 ecr 0,sackOK,eol], length 0
16:21:09.306658 IP 10.192.36.23.65299 > 109.105.109.208.443: Flags [S], seq 21931593, win 65535, options [mss 1460,nop,wscalc 5,nop,nop,TS val 388081393 ecr 0,sackOK,eol], length 0

2 packets captured
184 packets received by filter
0 packets dropped by kernel
d17018~ wsoons sudo tcpdump -s0 -n -c 100 host google.com
tcpdump: data link type PKTAP
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
Listening on pktap, Link-type PKTAP (Packet Tap), capture size 65535 bytes
16:21:32.088101 IP 10.192.36.23.65322 > 109.105.109.208.443: Flags [S], seq 3137143621, win 65535, options [mss 1460,nop,wscalc 5,nop,nop,TS val 388104541 ecr 0,sackOK,eol], length 0
16:21:32.909858 IP 109.105.109.208.443 > 10.192.36.23.65322: Flags [S.], seq 1641328408, ack 3137143622, win 28966, options [mss 1380,sackOK,TS val 252536409 ecr 388104541,nop,wscalc 7], length 0
16:21:32.909899 IP 10.192.36.23.65322 > 109.105.109.208.443: Flags [.] , ack 1, win 4184, options [nop,nop,TS val 388104562 ecr 252536389], length 0
16:21:32.910155 IP 10.192.36.23.65322 > 109.105.109.208.443: Flags [P.] , seq 1:518, ack 1, win 4184, options [nop,nop,TS val 388104562 ecr 252536389], length 517
16:21:32.920572 IP 109.105.109.208.443 > 10.192.36.23.65322: Flags [.] , ack 518, win 235, options [nop,nop,TS val 252536409 ecr 388104562], length 0
16:21:32.931059 IP 109.105.109.208.443 > 10.192.36.23.65322: Flags [P.] , seq 1:153, ack 518, win 235, options [nop,nop,TS val 252536410 ecr 388104562], length 152
16:21:32.931895 IP 10.192.36.23.65322 > 109.105.109.208.443: Flags [.] , ack 153, win 4099, options [nop,nop,TS val 388104582 ecr 252536410], length 0
16:21:32.931368 IP 10.192.36.23.65322 > 109.105.109.208.443: Flags [P.] , seq 518:569, ack 153, win 4099, options [nop,nop,TS val 388104582 ecr 252536410], length 51
16:21:32.931665 IP 10.192.36.23.65322 > 109.105.109.208.443: Flags [P.] , seq 569:726, ack 153, win 4099, options [nop,nop,TS val 388104582 ecr 252536410], length 157
16:21:32.931701 IP 10.192.36.23.65322 > 109.105.109.208.443: Flags [P.] , seq 726:1284, ack 153, win 4099, options [nop,nop,TS val 388104582 ecr 252536410], length 558
16:21:32.950668 IP 109.105.109.208.443 > 10.192.36.23.65322: Flags [.] , seq 726, win 243, options [nop,nop,TS val 252536430 ecr 388104582], length 0
16:21:32.953401 IP 109.105.109.208.443 > 10.192.36.23.65322: Flags [P.] , seq 153:209, ack 1294, win 252, options [nop,nop,TS val 252536433 ecr 388104582], length 136
16:21:32.953521 IP 10.192.36.23.65322 > 109.105.109.208.443: Flags [.] , ack 209, win 4095, options [nop,nop,TS val 388104603 ecr 252536433], length 0
16:21:32.953649 IP 10.192.36.23.65322 > 109.105.109.208.443: Flags [P.] , seq 1284:1322, ack 209, win 4095, options [nop,nop,TS val 388104603 ecr 252536433], length 38
16:21:33.011408 IP 109.105.109.208.443 > 10.192.36.23.65322: Flags [.] , ack 1322, win 252, options [nop,nop,TS val 252536492 ecr 388104603], length 0
16:21:33.012828 IP 109.105.109.208.443 > 10.192.36.23.65322: Flags [P.] , seq 289:449, ack 1322, win 252, options [nop,nop,TS val 252536493 ecr 388104603], length 160
16:21:33.012859 IP 10.192.36.23.65322 > 109.105.109.208.443: Flags [.] , ack 449, win 4091, options [nop,nop,TS val 388104661 ecr 252536493], length 0
16:21:33.013268 IP 109.105.109.208.443 > 10.192.36.23.65322: Flags [.] , seq 449:1817, ack 1322, win 252, options [nop,nop,TS val 252536494 ecr 388104603], length 1368
16:21:33.013832 IP 109.105.109.208.443 > 10.192.36.23.65322: Flags [P.] , seq 1817:2497, ack 1322, win 252, options [nop,nop,TS val 252536494 ecr 388104603], length 680
16:21:33.013848 IP 109.105.109.208.443 > 10.192.36.23.65322: Flags [P.] , seq 2497:2865, ack 1322, win 252, options [nop,nop,TS val 252536494 ecr 388104603], length 1368
16:21:33.013864 IP 109.105.109.208.443 > 10.192.36.23.65322: Flags [P.] , seq 2865:5233, ack 1322, win 252, options [nop,nop,TS val 252536494 ecr 388104603], length 1368
16:21:33.013848 IP 109.105.109.208.443 > 10.192.36.23.65322: Flags [P.] , seq 5233:6661, ack 1322, win 252, options [nop,nop,TS val 252536494 ecr 388104603], length 1368
16:21:33.013852 IP 109.105.109.208.443 > 10.192.36.23.65322: Flags [P.] , seq 6661:7499, ack 1322, win 252, options [nop,nop,TS val 252536494 ecr 388104603], length 898
16:21:33.013873 IP 10.192.36.23.65322 > 109.105.109.208.443: Flags [.] , ack 2497, win 4074, options [nop,nop,TS val 388104662 ecr 252536494], length 0
16:21:33.013884 IP 10.192.36.23.65322 > 109.105.109.208.443: Flags [P.] , seq 5233, win 3989, options [nop,nop,TS val 388104662 ecr 252536494], length 0
16:21:33.013890 IP 10.192.36.23.65322 > 109.105.109.208.443: Flags [.] , ack 7499, win 3918, options [nop,nop,TS val 388104662 ecr 252536494], length 0

```

Figure 4.29: Experiment with the parameters of ‘tcpdump’ for networked data analysis



Figure 4.30: Experiment with watching Youku video with data analysis. Video link: [http://v.youku.com/v\\_show/id\\_XMTI1NTU4OTA0NA==.html?from=s1.8-1-1.2](http://v.youku.com/v_show/id_XMTI1NTU4OTA0NA==.html?from=s1.8-1-1.2)

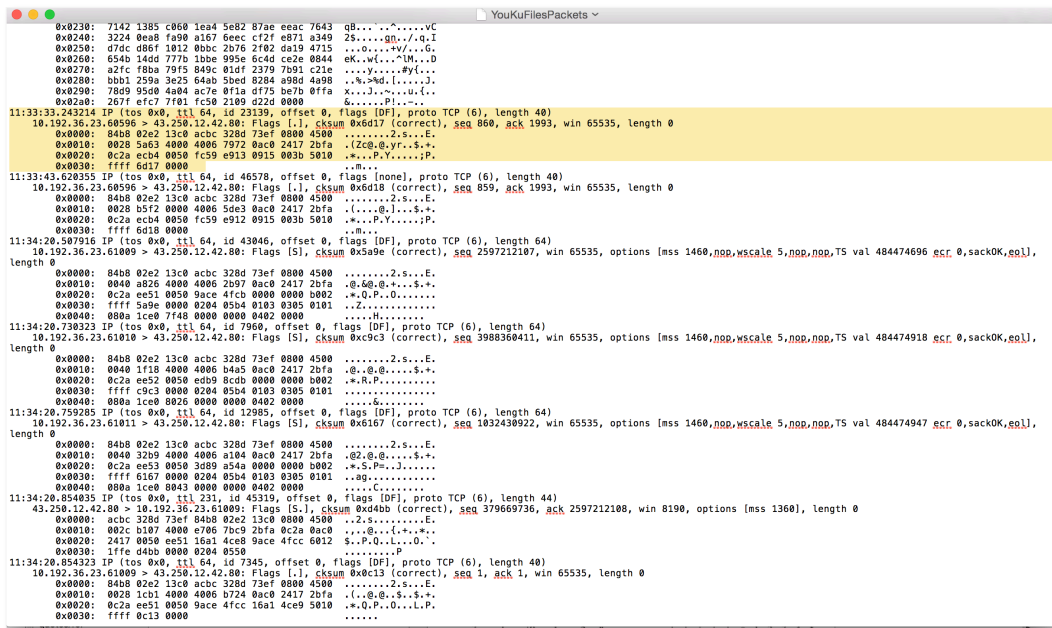


Figure 4.31: Log analysis for the Youku video in relation to Figure 4.30

However, I did not understand them very well and this prompted me to a “cold gazing” (Ernst, 2006, 2013b) of the technical specification of internet protocols as well as the logic of Baran’s packet switching. With Ernst’s concept of micro-temporality in mind, I started to focus on the micro-time and micro-processes in networked data organisation.

From the terminal experiments the field 'TTL' (Time to Live) caught my attention, first because of its peculiar name. Later it proved to be very useful, as demonstrated in the earlier discussion, in thinking through the notion of liveness and deadness. Through connecting to a video streaming site, I discovered the limitation of the terminal command of 'tcpdump' with less decoded and organised information, and this prompted me to use the software Wireshark for packet analysis. The micro-temporal analysis in section 4.2 is a detailed description and analysis of the technical infrastructure of data streams, and this is what Rolling Jr describes as "thinking through a context" (Rolling Jr, 2014, p. 163). The context that I have applied is more oriented towards technological infrastructure.

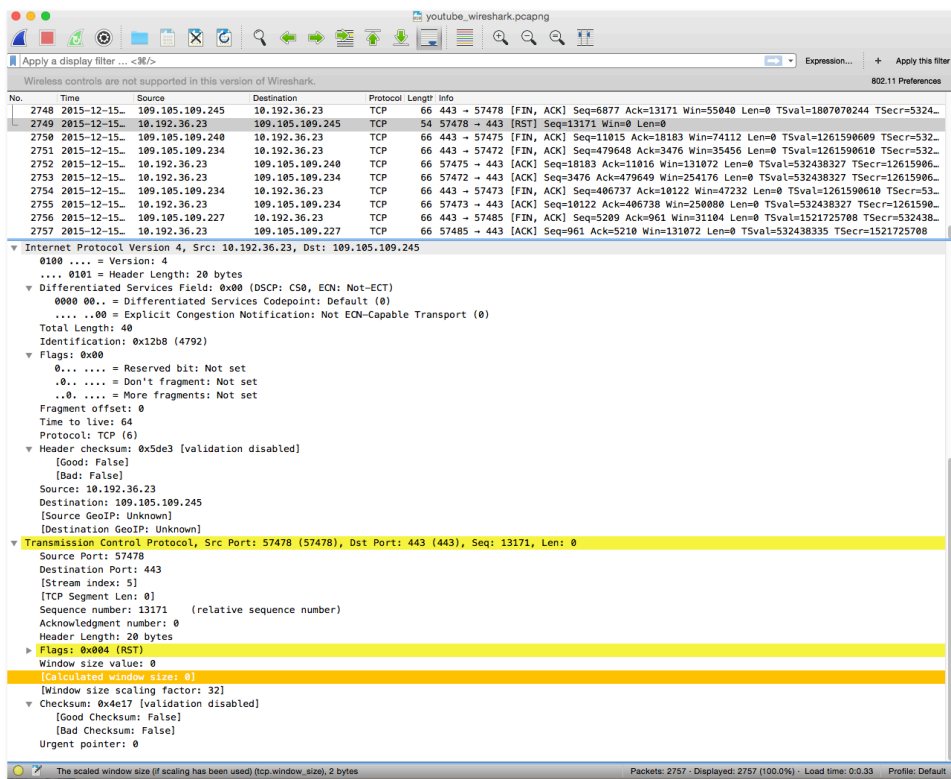


Figure 4.32: A screen shot of Wireshark for packet analysis (with a focus on window size)

The project, *The Spinning Wheel of Life*, is being slowly and gradually developed to draw attention to the micro-temporality of streams. Due to the high internet bandwidth and the heavy data transmission required for streaming videos I intentionally chose a streaming playlist that required less data packets. Since I want to develop a rhythmic throbber which is not running at a constant speed, the decision to use an 8-bit audio stream allows for longer temporal pauses in data transmission while the stream is playing back. This is because there is not much data to transfer within an audio clip. The temporal rhythms

between data transmission and playback can be arranged in a more revealing and clear manner.

RSG's Carnivore library was used to track networked data. From there I decided to maintain emphasis on the matter of time processing, instead of the actual data/content of a data stream. Therefore, the visual presentation of *The Spinning Wheel of Life* contained only the movement of ellipses.

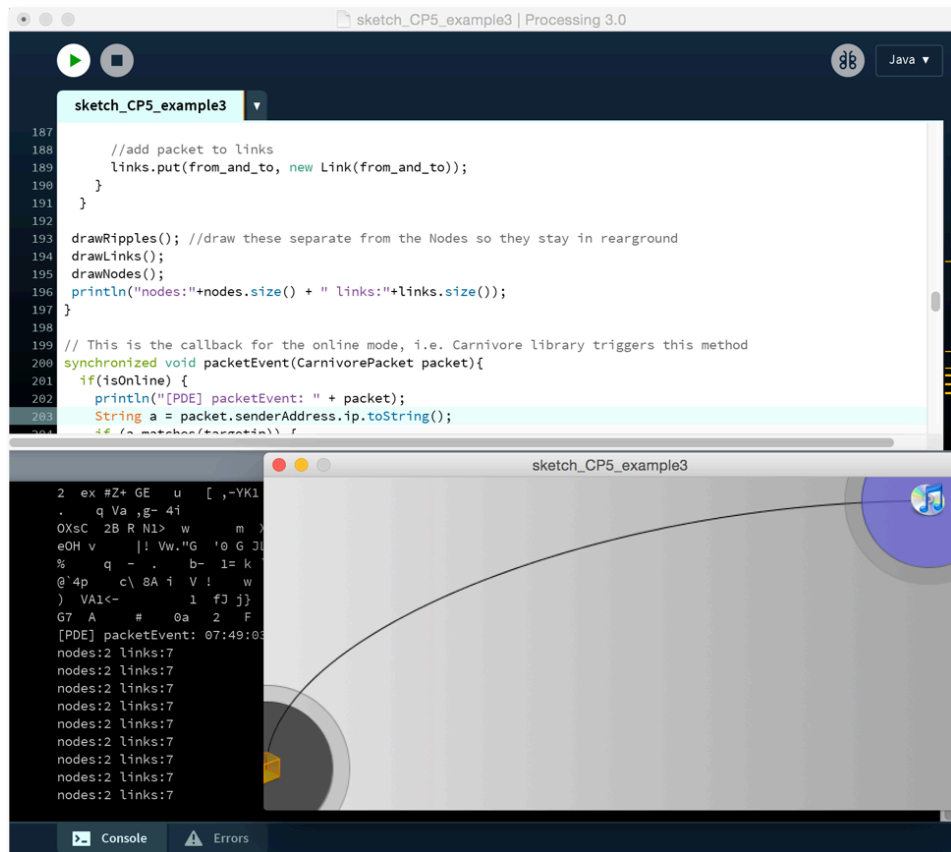


Figure 4.33: Tracking networked data: Experiment the Carnivore library by RSG in Processing

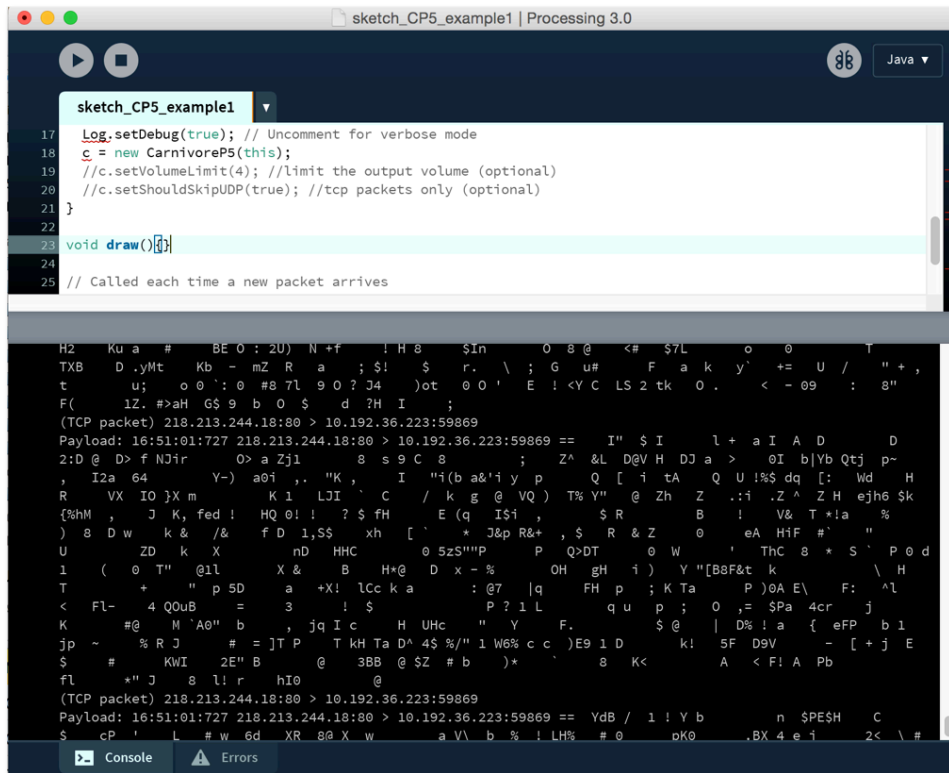


Figure 4.34: Tracking networked data with log:  
Experiment the Carnivore library by RSG in Processing

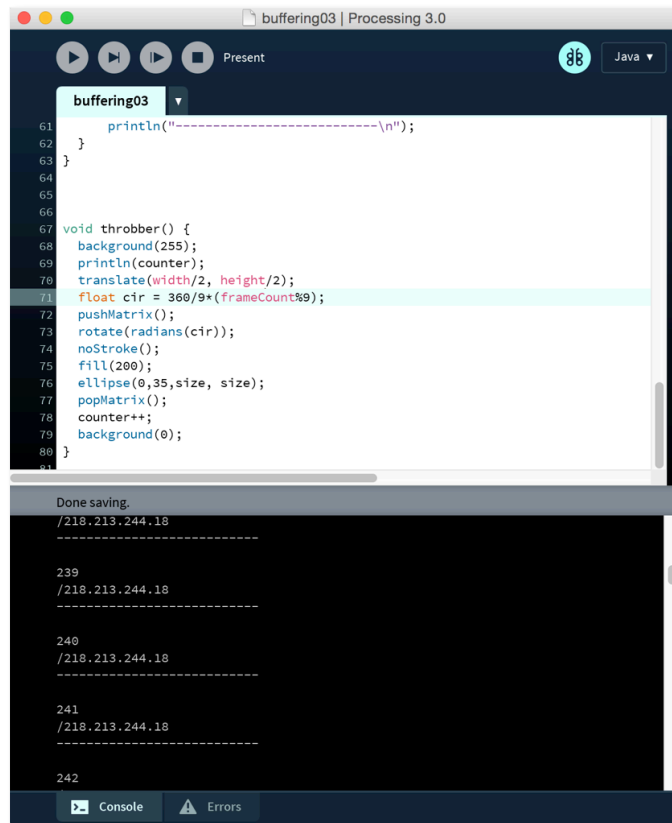


Figure 4.35: The log for networked data experimentation

The setup of the work is not a straightforward implementation. There are many other things to consider, such as how the work is displayed, which audio list is played, what hardware to use, how big the screen is and how many of them to use if the work is going to be shown as an installation.

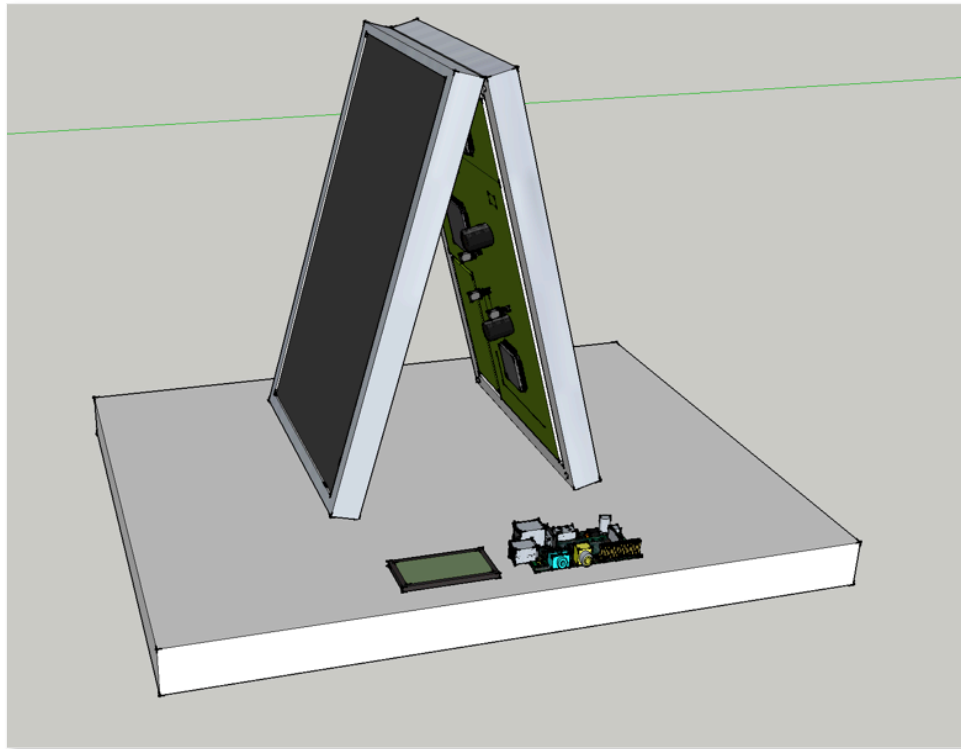


Figure 4.36: Initial setup concept of *The Spinning Wheel of Life*

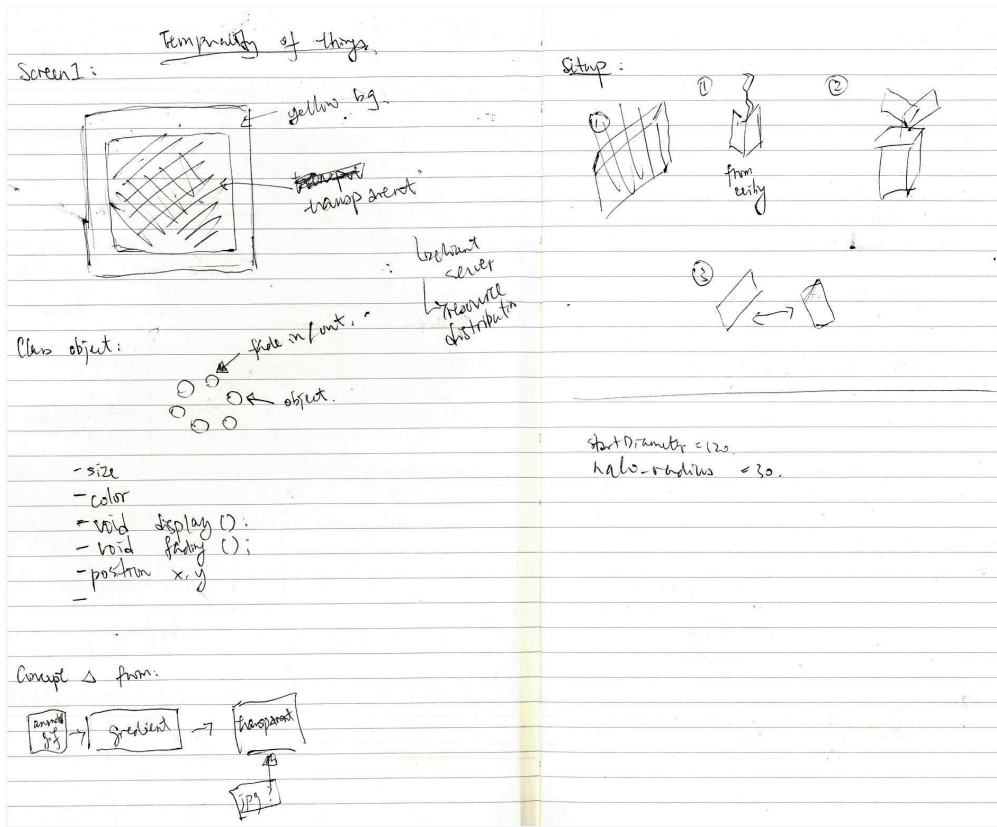


Figure 4.37: Concept stage of The Spinning Wheel of Life

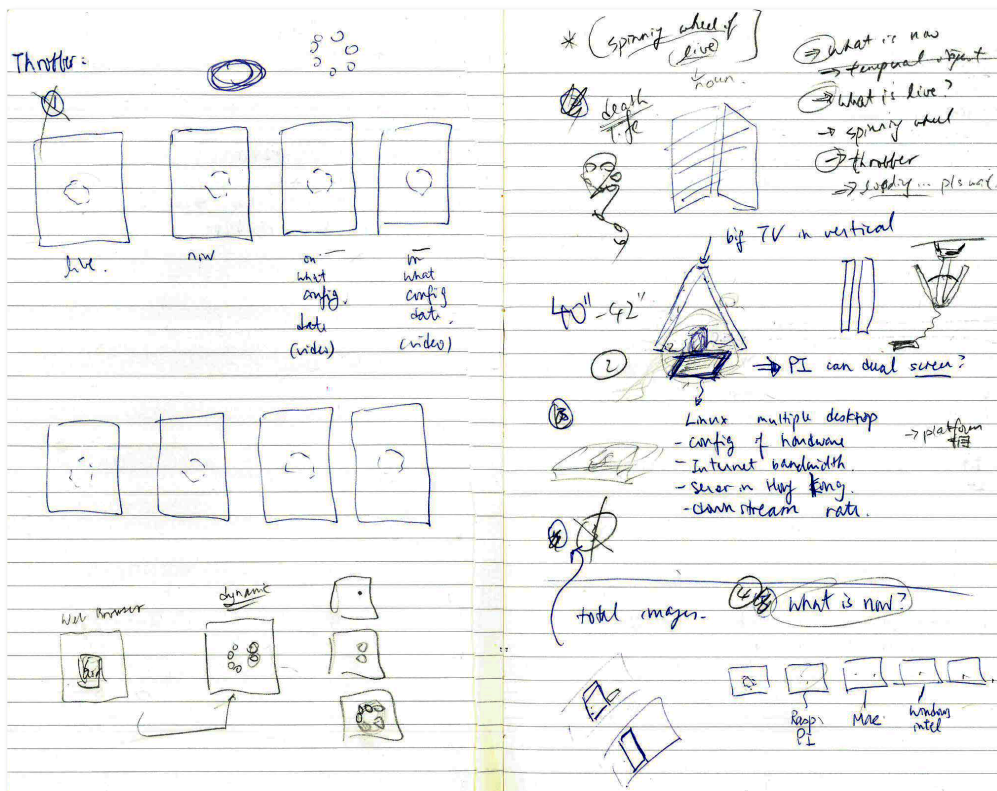


Figure 4.38: Concept stage of The Spinning Wheel of Life.



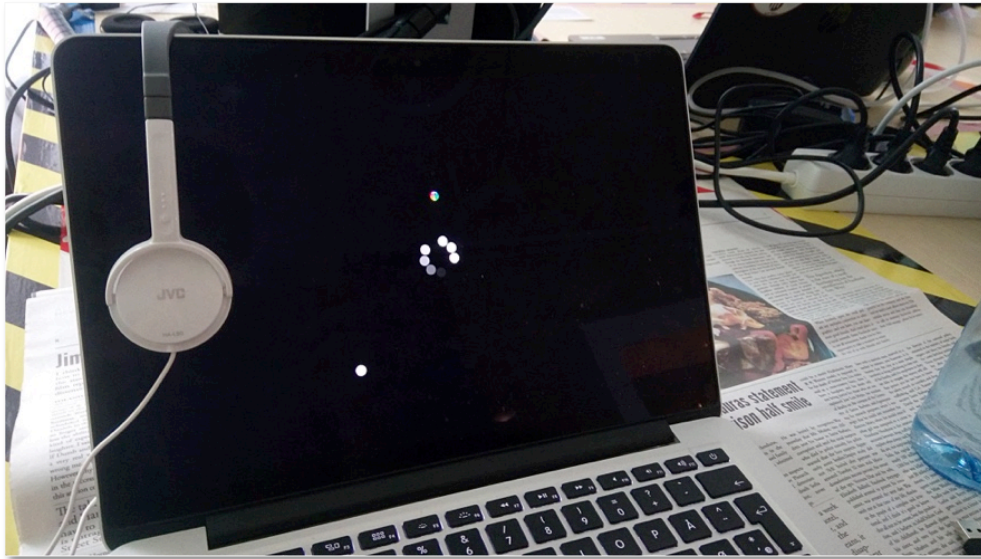
In April 2016, I had a chance to show this work-in-progress as part of the execution workshop,<sup>128</sup> organised by the Critical Software Thing Group. The workshop encouraged participants to present practice-based prototypes with the theme of execution for discussion. Presenting the earlier version of this project has opened up further discussion about time and execution (such as the clock cycle discussion) as illustrated in the earlier sections. The artist's statement relating to the project highlights the micro-processes that are running behind a visible and notable throbber icon:

The project challenges the perception of a throbber that is usually understood as a transitional object, waiting for content delivers on a screen. By having a music stream that runs in the background, the throbber appears not because of waiting for data arrival but, instead, it spins as the machine receives data and stores them in a buffer for immediate retrieval. There are micro processes that are merely being noticed. Commonly, a throbber expresses the unknowable waiting time that spins at a constant rate. In *The Spinning Wheel of Life*, the project explores a different dimension of time—the “micro-temporality” of data transmission and data processing through subverting the usual functioning of a throbber. Beyond the negative connotations of waitings, frustrations and annoyance, how might we reflect on a throbber, also known as the spinning wheel of death, which has become a cultural object and used commonly in contemporary software culture? (Soon, 2016c).

Due to hardware limitations the work was shown on a Mac laptop computer with headphones. Figure 4.39 shows that the work crashes with the Mac's built in animated icon: 'The Spinning Wheel of Death.' This incident also demonstrates the coupling forces of the living and the dead.

---

<sup>128</sup> See: [http://softwarestudies.projects.cavi.au.dk/index.php/\\*\\_exe\\_\(ver0.2\)#30\\_April:\\_Masterclass](http://softwarestudies.projects.cavi.au.dk/index.php/*_exe_(ver0.2)#30_April:_Masterclass)



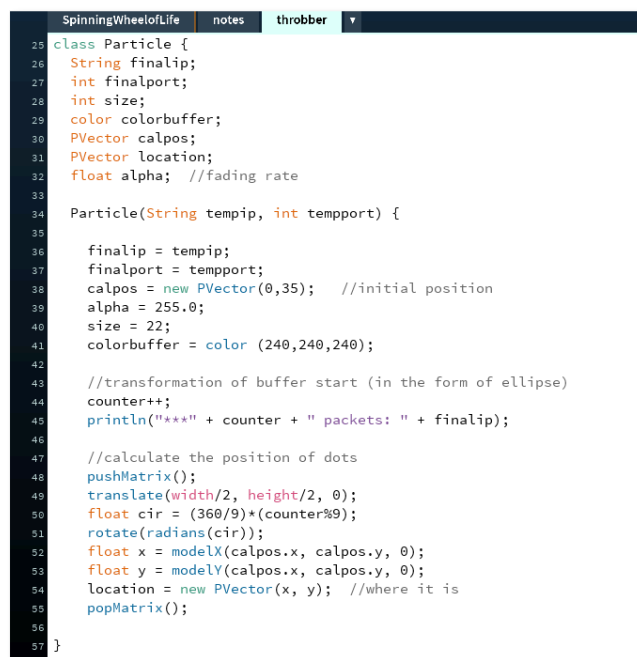
*Figure 4.39: Concept stage of The Spinning Wheel of Life.*

I was not satisfied with this version mainly because the use of a Mac computer cannot fully express the concept of micro-temporality. As I have argued earlier, the throbber presents multiple realities that lie at the heart of time-dependent logics and these logics exhibit different rates, tempos, pulses, pauses and rhythms at multiple sites and scales as assemblages. Therefore, I was keen to show it with a more transparent hardware setup where audiences could see through the motherboard or computer and by seeing the blinking LED lights that indicate the operation of the CPU and the micro-time gap between the electronic board and the screen. To this end the latest version (see Figure 4.20) reveals all the wires and electronic circuits of an electronic motherboard that connects with a screen. The project is to be shown in Transmediale 2017 and the experimentation with different forms of the setup is ongoing. This reflexive reconfiguration of the work and its setup demonstrates how I think reflexively. As Rolling Jr describes, “thinking reflexively, an exercise in continuous, practice-based experiential learning” (2014, p. 163).

Technically, *The Spinning Wheel of Life* is written in Processing. It utilises an external library called Carnivore<sup>129</sup> which was developed by the Radical Software Group (RSG) to monitor data network traffic. Since the project requires monitoring network traffic an internet connection is required throughout the duration of the installation.

Ideally the installation consists of multiple mini setups. Each mini setup displays a throbber on a 7 inch screen and a designated streaming audio is played through a small speaker attached to a mini computer (a Raspberry Pi 3). Since the work requires access to the network layer, a root access<sup>130</sup> is required to run the program. Additionally, a web browser is required to access a YouTube playlist and I have chosen an 8-bit video game play list<sup>131</sup> because each track will be small enough in size to make apparent the captured packets in a more visible way.

The program constantly listens to a range of IP addresses in real-time. Each display of an ellipse correlates to the arrival of a network packet. A series of the ellipses will then reassemble to form an animated throbber.



```
SpinningWheelofLife | notes | throbber | v
25 class Particle {
26   String finalip;
27   int finalport;
28   int size;
29   color colorbuffer;
30   PVector calpos;
31   PVector location;
32   float alpha; //fading rate
33
34   Particle(String tempip, int tempport) {
35
36     finalip = tempip;
37     finalport = tempport;
38     calpos = new PVector(0,35); //initial position
39     alpha = 255.0;
40     size = 22;
41     colorbuffer = color (240,240,240);
42
43     //transformation of buffer start (in the form of ellipse)
44     counter++;
45     println("***" + counter + " packets: " + finalip);
46
47     //calculate the position of dots
48     pushMatrix();
49     translate(width/2, height/2, 0);
50     float cir = (360/9)*(counter%9);
51     rotate(radians(cir));
52     float x = modelX(calpos.x, calpos.y, 0);
53     float y = modelY(calpos.x, calpos.y, 0);
54     location = new PVector(x, y); //where it is
55     popMatrix();
56   }
57 }
```

Figure 4.40: An excerpt from *The Spinning Wheel of Life* (work-in-progress)'s source code: The ellipses design

<sup>129</sup> See: <http://r-s-g.org/carnivore/>

<sup>130</sup> The command to open Processing is 'sudo ./ processing'

<sup>131</sup> See: <http://bit.ly/1ppETHQ>

```

SpinningWheelofLife  notes  throbbler  ▾
1 import java.util.Iterator;
2 import org.rsg.carnivore.*;
3 import org.rsg.carnivore.net.*;
4 import org.rsg.lib.Log;
5
6 //CarnivoreP5 p;
7
8 int size = 22;
9 String [] targetip= {"64.18.0.0", "64.233.160.0",
10 "66.102.0.0","66.249.80.0","72.14.192.0","74.125.0.0", "173.194.0.0","207.126.144.0",
11 "209.85.128.0", "216.58.208.0", "216.58.212.142", "216.239.32.0", "172.24.115.27",
12 "216.58.197.97", "172.30.8.132"}; //youtube ips
13 int counter = 0;
14 ParticleSystem ps;
15 // Flag for online/offline modes.
16 boolean isOnline = true;
17 String packets[];
18 HashMap nodes = new HashMap();
19
20 void setup() {
21   background(0,0,0, 80);
22   frameRate(1000); //100, max 1000
23   //size(800,800, P3D);
24   fullScreen(P3D);
25   ps = new ParticleSystem();
26   smooth();
27   CarnivoreP5 p = new CarnivoreP5(this);
28 }
29
30 void draw() {
31   fill(0,0,0, 80);
32   rect(0, 0, width, height);
33   ps.run();
34 }

```

Figure 4.41: An excerpt from *The Spinning Wheel of Life* (work-in-progress)'s source code: Setting up IP addresses and the Carnivore library.

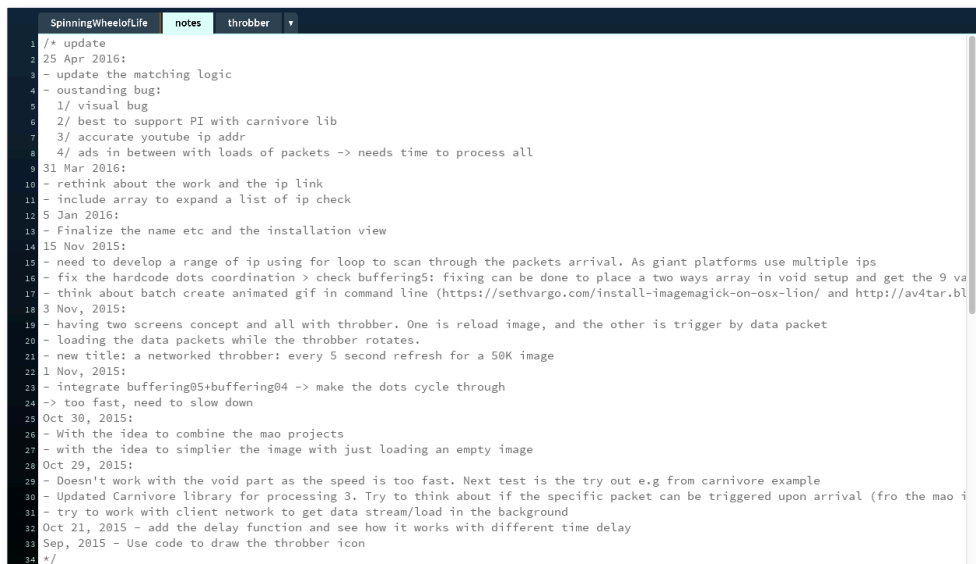
```

***785 packets: 216.58.211.142
***786 packets: 216.58.211.142
***787 packets: 216.58.211.142
***788 packets: 216.58.211.142
***789 packets: 216.58.211.142
***790 packets: 216.58.211.142
***791 packets: 216.58.211.142
***792 packets: 216.58.211.142
not match /10.192.40.150
***793 packets: 216.58.211.142
***794 packets: 216.58.211.142
***795 packets: 216.58.211.142
***796 packets: 216.58.211.142
***797 packets: 216.58.211.142
***798 packets: 216.58.211.142
***799 packets: 216.58.211.142
***800 packets: 216.58.211.142
not match /10.192.40.150
***801 packets: 216.58.211.142
***802 packets: 216.58.211.142
***803 packets: 216.58.211.142
***804 packets: 216.58.211.142
***805 packets: 216.58.211.142
***806 packets: 216.58.211.142
***807 packets: 216.58.211.142
***808 packets: 216.58.211.142
not match /10.192.40.150
***809 packets: 216.58.211.142
***810 packets: 216.58.211.142
***811 packets: 216.58.211.142

```

Figure 4.42: An excerpt from *The Spinning Wheel of Life*'s log: The feedback process

With respect to documentation, instead of setting up a blog as I did for *Thousand Questions*, I created a tab called *notes* in my program which documents my updates of it (see Figure 4.43). After the project's first presentation at Malmö University in 2016, the source code was uploaded to GitHub, a web-based code hosting platform for version control. From thereon a testlog file has been created on the platform to continue the documentation of this project (see Figure 4.44). Similar to the *Thousand Questions* that discussed in the last chapter, documentation is made in the form beyond scholarly writing and is considered as an important part of reflexive practice.



```
SpinningWheelofLife  notes  throbber
1 /* update
2 25 Apr 2016:
3 - update the matching logic
4 - outstanding bug:
5 1/ visual bug
6 2/ best to support PI with carnivore lib
7 3/ accurate youtube ip addr
8 4/ ads in between with loads of packets -> needs time to process all
9 31 Mar 2016:
10 - rethink about the work and the ip link
11 - include array to expand a list of ip check
12 5 Jan 2016:
13 - Finalize the name etc and the installation view
14 15 Nov 2015:
15 - need to develop a range of ip using for loop to scan through the packets arrival. As giant platforms use multiple ips
16 - fix the hardcoded dots coordination > check buffering5: fixing can be done to place a two ways array in void setup and get the 9 va
17 - think about batch create animated gif in command line (https://sethvaro.com/install-imagemagick-on-osx-lion/ and http://av4tar.bl
18 3 Nov, 2015:
19 - having two screens concept and all with throbber. One is reload image, and the other is trigger by data packet
20 - loading the data packets while the throbber rotates.
21 - new title: a networked throbber: every 5 second refresh for a 50K image
22 1 Nov, 2015:
23 - integrate buffering05+buffering04 -> make the dots cycle through
24 -> too fast, need to slow down
25 Oct 30, 2015:
26 - With the idea to combine the mao projects
27 - with the idea to simplify the image with just loading an empty image
28 Oct 29, 2015:
29 - Doesn't work with the void part as the speed is too fast. Next test is the try out e.g from carnivore example
30 - Updated Carnivore library for processing 3. Try to think about if the specific packet can be triggered upon arrival (fro the mao i
31 - try to work with client network to get data stream/load in the background
32 Oct 21, 2015 - add the delay function and see how it works with different time delay
33 Sep, 2015 - Use code to draw the throbber icon
34 */
```

Figure 4.43: An excerpt from *The Spinning Wheel of Life's* source code: General notes from 2015 to 2016

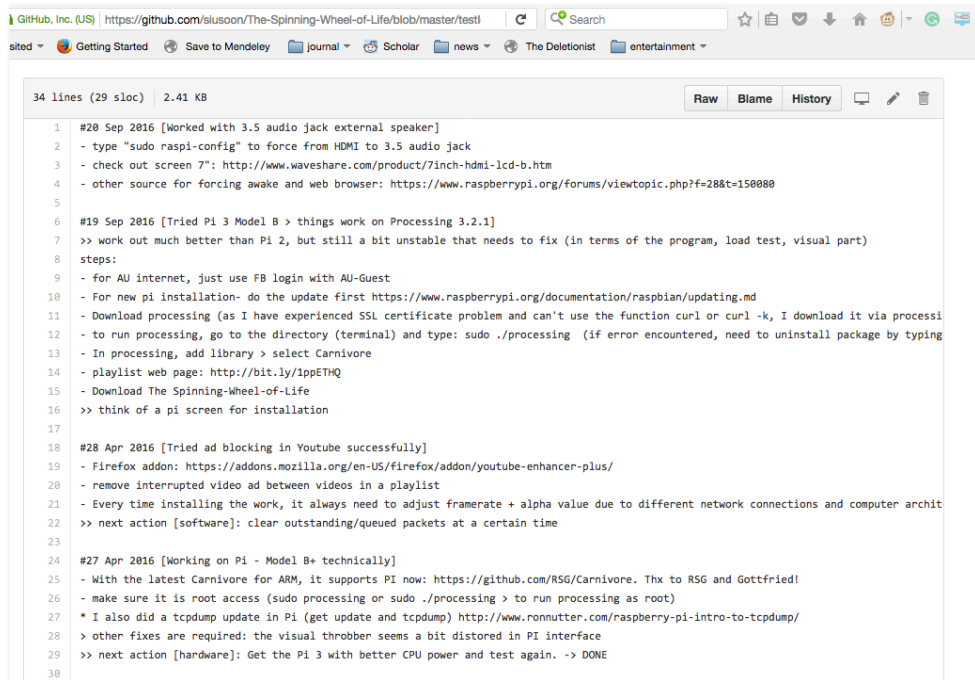
A screenshot of a web browser displaying a GitHub repository page. The address bar shows the URL: https://github.com/siusoon/The-Spinning-Wheel-of-Life/blob/master/testlo. The page content is a list of commit messages and notes, numbered 1 through 30. The notes are organized into sections by date: 20 Sep 2016, 19 Sep 2016, 28 Apr 2016, and 27 Apr 2016. The notes describe technical work on a Raspberry Pi, including audio jack configuration, software installation, and hardware fixes. The browser interface includes a search bar, navigation icons, and a menu bar with items like 'Getting Started', 'Save to Mendeley', 'Journal', 'Scholar', 'news', 'The Deletionist', and 'entertainment'.

Figure 4.44: A screen shot of the notes from Apr 2016 to present. Retrieved from: <https://github.com/siusoon/The-Spinning-Wheel-of-Life/blob/master/testlo>

In short, this section shows how the experimental and work-in-progress project *The Spinning Wheel of Life* is being developed and the investigative process of a throbber that informs and enriches the chapter's content. In addition, the discussion in this section demonstrates the applied approach of Rolling Jr's reflexive framework, alluding to thinking in a material, thinking in a language, thinking through a context and thinking reflexively.

This project, similar to the previous artwork *Thousand Questions*, emphasises the processes that are running live as part of the work, those that exhibit the assemblage of forces, contingently reconfiguring through material relations. Specifically, *The Spinning Wheel of Life* dynamically executes micro-temporal streams.



## 5

# Executing Automated Tasks

This chapter examines the third vector which constitutes the notion of liveness: automation. The term automation is associated with the execution of tasks in computation. Paper tape and punch cards were commonly used as data storage in the early form of computational data processing from the 1950s to 1980s, encompassing different patterns of punched holes for batch processing. The concept of batch processing alludes to the automation of a batch of inputs without “user intervention” (Fitzpatrick, n.d.) or “end user interaction” (IBM, 1990). Although the cards were prepared in advance and put in the machine by human operators, it was the machines that executed the reading of the cards and processed the inputs. Recently the term automation has also frequently appeared in the hype around machine learning, data mining and artificial intelligence in contemporary software culture (Burrell, 2016; Kephart & Chess, 2003; Morris, 2015). In commercial sector systems such as automated recommendation engines, push notifications and automated reporting are increasingly important as business and technological solutions for targeted marketing and profit makings.<sup>132</sup> Automated systems enable real-time computation and the tracking and querying of data, responding to live environments with pre-set algorithms that are automated and executed by machines. From historical to contemporary practices of data processing, it would seem that the notion of automation is crucial in understanding software culture.

The notion of automation concerns efficiency. One of the major benefits of batch processing is that tasks can be scheduled to run in a less busy time

---

<sup>132</sup> See some examples here:

- (1) Push notification engine by Swrve: <https://www.swrve.com/product/push-notifications>
- (2) Mass Notification by Everbridge: <http://www.everbridge.com/products/mass-notification/>
- (3) Recommendation engine that is offered by Predictry: <http://predictry.com/>
- (4) Automating complex reporting tasks through Google Analytics APIs: <https://developers.google.com/analytics/solutions/reporting>



interval and can be run in batches. Given the fact that a program can handle more than one transaction batch processing reduces the need to run the same program in many times. The concept of “automatic coding” was introduced by Hopper in 1955, allowing a computer to “replace” any programmer so as to achieve “the reduction of the computer time” (1955, p. 2). Her profound invention of a compiler was aimed at reducing the amount of time needed to prepare for machine translation of code but, more importantly, compilers check for mistakes and are able to detect them before actual running. This reduces the time needed for debugging before the program is ready for “production running” (Hopper, 1955, p. 1). The notion of automation commonly implies efficiency<sup>133</sup> insofar as it relates to reducing or replacing human labour or optimising processing time.

More broadly automation can be seen as a way to examine software culture. Manovich proposes a list of “general tendencies”<sup>134</sup> of understanding software culture, in which automation is one of those principles (2001, p. 27). The term automation stresses the operational aspect of code interactions that are repeatedly executed, including the “creation, manipulation, and access” of media and data (Manovich, 2001, p. 32). In contemporary culture, media materials have been archived digitally that change the way with which materials are being organised, stored and accessed. Accessing and querying platform databases have been previously discussed in Chapter 3, but automation is more than that because it allows tasks to be automated operatively and repeatedly without much human intervention. As in the artwork *Listening Post* (see Chapter 1) online materials are constantly extracted and turned into new forms of expression by free-standing machines. By accessing and taking in different data inputs, the output can then be calculated, manipulated and produced in multiple varieties. Therefore, automation lies at the heart of *the act of repetition*: the ability to execute things repeatedly and automatically. Since automation involves

---

<sup>133</sup> The term automation also refers to “automatically controlled operation of an apparatus, process, or system by mechanical or electronic devices that take the place of human labor” (“Automation,” n.d.).

<sup>134</sup> In Manovich’s earlier work, he proposes five “principles of new media” that could be understood as “general tendencies of a culture undergoing computerization”: Numerical Representation, Modularity, Automation, Variability and Transcoding (2001, pp. 27-48).

computational creation and production, there are built-in logics and algorithms that can take different inputs and, consequently, producing different outputs. As such, the act of repetition implies a process of generating differences.

According to some philosophers and scholars, the repetitive aspect of automation implies difference (Chun, 2008a, 2008b, 2011a, 2011b, 2015, 2016; Deleuze, 1968; Derrida, 1978). In relation to the execution of code and the nature of information circulation Chun points out that code is fundamental in understanding repetition and its differences. She says that code “is *undead* writing, a writing that—even when it repeats itself—is never simply a deadly or living repetition of the same” (Chun, 2011b, p. 177, *original emphasis*). Chun’s employment of the notion of “undeadness” draws attention to repetitions, changes and differences at the micro level of “inner writing” (2008a, p. 165). For her, the inner writing of code is about the activity of writing on computer memory and it is a “degenerative” process, alluding to things are not “always there” (Chun, 2008a, p. 148). Indeed the notion of undeadness does not merely refer to loss or degenerativity but, importantly, the endless updating and refreshing of information proliferates the “enduring ephemerals” to prevent loss (Chun, 2008a, p. 149). What is live, as Chun suggests, is about undeadness—the endless updating and circulation of information, as well as “the ever-updating, ever-proliferating, and increasingly incompatible software and hard technologies” (2011b, p. 138). This chapter raises some important questions concerning liveness and automation: How is the act of repetition automated by computation? What are the implications of repetition in connection with liveness and undeadness? How has automation, in connection with liveness and undeadness, reconfiguring our understanding of software?

In view of the fact that there are many computational activities which are automated and run in a real-time environment the notion of automation is increasingly important in examining contemporary software culture which emphasises the latest or newest phenomena, especially the urgency of updates or upgrades when one interfaces with operating systems, platforms,

applications and devices. The alert or the notification of an update is instant and timely, automatically categorising one's currently in-use system or application as an old version. Sometimes these old versions do not even work as expected (see also the notion of the inexecutable query in Chapter 3, section 3.5). Chun discusses the characteristics of the old and new that are conditioned by constant requests for updating in detail. She says:

New media, if they are new, are new as in renovated, once again, but on steroids, for they are constantly asking/needing to be refreshed. They are new to the extent that they are updated. [...] *New media live and die by the update*: the end of the update, the end of the object. Things no longer updated are things no longer used, useable, or cared for, even though updates often 'save' things by literally destroying-that is, writing over-the things they resuscitate (Chun, 2016, p. 2, *original emphasis*).

Within software (update) culture there is even no upfront prompting to update, rather things are processed automatically behind a screen. This is particularly observed in social media platforms where new features are released quietly until users discover them (the gender feature in Facebook, as mentioned in Chapter 3, is a case in point). Additionally, without logging out of a platform or an application, data is continuously processed in the background without an end time. Computation operates automatically and continuously through repetition in which differences are produced, yet it does so without a clear end point or final state because the next update is underway. This points towards a sense of endlessness in contemporary update culture. Arguably automation has contributed to a reconfiguration of the notion of liveness by blurring the distinctions between new and old, life and death, start and end. From visible interfaces and their user interactions to background processes and their code inter-actions, updates are highly automated and that requires our attention.

Although it is common to associate automation with wider social and political consequences, an example being the cultural theorist Tiziana

Terranova who suggests that “post-capitalist” software practices have changed the nature of automation (2014, n.p), this chapter is less concerned with issues related to human labour (important although they are) and will more specifically investigate forms of automation in code inter-actions. Instead of emphasising specific economic or social aspects, this chapter aims to illustrate some of the material operations and processes of automation to illuminate its relation to liveness. The concern with micro-time and micro-processes is carried over from the last chapter to this chapter. However, this chapter shifts attention away from the various micro activities of machines and network protocols to the micro processes of running algorithms in which to perform the undead writing that made automation possible. Special attention is paid to the time in which an algorithm is endlessly run and how tasks are automatically executed (a more detailed explanation of algorithms follows in a later section). This chapter is concerned with algorithms in terms of automation. It unfolds the discussion through my artistic project, *Hello Zombies*, which specifically explores the phenomena of spam automation that invades our internet network. Such an examination of spam as automated agents further considers the increasingly commonplace phenomena of other agents such as bots, auto software updates and alert notifications.

This chapter is structured to examine the relationship between undeadness and automation at the level of code and extends this to algorithms by a close reading of code written for *Hello Zombies*. This follows the tradition of close reading in critical code studies which extracts selected blocks of code for further investigation and analysis (see Chapter 2, section 2.4.1, for details on the method of close reading code). Additionally, this chapter maintains a focus on running and executing code, alluding to how code is always in progress (implying undeadness) and its inter-actions are held together contingently.

## 5.1 Spam as automated agents<sup>135</sup>

*Hello Zombies* explores spam production and automation. Spam appears everywhere on the internet from downloaded emails to server-based blogs, forums and social media communications. In 2014 statistics showed that the proportion of spam almost reached 70% of entire email traffic.<sup>136</sup>

Most spams that show up in an email's inbox is automatically programmed with a customised body of content and a peculiar email address.<sup>137</sup> In day-to-day form-filling from paper to electronic registration workflows, supplying an email address is a mandatory field, therefore an email address is equally important to a mobile number as a way to contact another person. In addition, email addresses come with standard naming conventions; a domain usually belongs to or has a connection with a particular organisation. Sometimes a domain not only describes the nature of an institution but also indicates a geographic location through the last two characters. A recipient is usually unaware that the sender's address can be easily customised in an email, regardless of its authenticity or whether it exists in a network. Therefore spammers can easily tailor sender addresses to transmit messages and consequently new identities<sup>138</sup> are created in the network. New spam emails are created everyday but they are also caught and blocked by spam filtering algorithms. On the one hand sender addresses are actively 'living' and distributed in the network, continually monitored by algorithms; while on the other they consume numerous resources of the network and are regarded as "waste" (Gabrys, 2011, p. 67; Parikka & Sampson, 2009b, p. 4) and "unwelcome" (Burrell, 2016, p. 7) entities to be traced and trashed. Once an email address is blocked the spammer creates other new identities by using another fake and customised email address.

---

<sup>135</sup> An earlier version of this section has been published in elsewhere (Soon, 2015a, 2015b, 2015c).

<sup>136</sup> See the Spam Report (2014): <http://securelist.com/analysis/monthly-spam-reports/58559/spam-report-february-2014/>

<sup>137</sup> Examples of such email addresses are [naomiwhitfield274@trash-mail.com](mailto:naomiwhitfield274@trash-mail.com) and [\\*\\*\\*\\*\\*@gmail.com](mailto:*****@gmail.com). Spam email addresses can be found in stop forum spam: <http://www.stopforumspam.com/downloads/>

<sup>138</sup> However, many of the email addresses do not exist in the network and are easily identified as spammers. The sender address appears to stand as a proper identity and as such is ready for others to reply to.

Both addresses are not real (without a valid owner that one can reply to) but an address is important to invade the network without getting filtered by other email systems and to function as an email that aims to arrive in an inbox folder. In other words, the spammer's email address can be considered as the living dead because once they are identified they need to produce another one. Most spammer email addresses are not real. This reproduction process is endless. As a result, the lifespan of a peculiar spammer address is ephemeral. This state of the living dead resonates with the notion of undeadness because it relates to something ephemeral that is "neither alive nor dead, neither quite present nor absent" (Chun, 2011b, p. 133). With its changing identities, which are endlessly generated and disseminated through distributed networks, spam can also be seen in this way.

### 5.1.1 *Hello Zombies*

*Hello Zombies* was an installation made in 2014 that explored how code inter-acts with the mail server to create spam (see Figure 5.1). The work was exhibited as part of the group exhibition, "Tracing Data: what you read is not what we write" which was staged at the Connecting Space laboratory in Hong Kong.<sup>139</sup> The setup of the work was like an assembly line: each machine and the corresponding computer script were responsible for doing a particular job, breaking down the production processes into various constituent parts. The project contained three software programs that ran simultaneously and automatically and no human intervention or physical interaction was required. Each machine played a distinct role to execute pre-defined tasks.

---

<sup>139</sup> For the concept statement of the exhibition, *Tracing Data*, see: [http://www.writingmachine-collective.net/wordpress/?page\\_id=76](http://www.writingmachine-collective.net/wordpress/?page_id=76)



Figure 5.1: *Hello Zombies* (2014)

Spam are considered “unwanted visitor[s]” (Parikka & Sampson, 2009a, p. 101) in so far as they are unsolicited by and have no use value for either the network or the recipient while occupying enormous amounts of network and storage resources. In some instances, however, spam produces artistic or poetic values. As Galloway and Eugene Thacker note, “with a string of spam emails, many of them containing non-sense text that, from another perspective, forms a strange *poetics* of spam” (2009, p. 253, *original emphasis*). These strange values can also be observed in many artistic works, for example *ee spammings* (2010)<sup>140</sup> by Martin Krzywinski and *Spam Heart* (2010)<sup>141</sup> by David Jhave Johnston in which both artists work with spam to explore their use of rhetorical linguistic devices. Spam poems are one of the outcomes of *Hello Zombies* but these poems were not sent to any normal users. The spam poems were sent back to spammers through an automated system that consists of an assembly line. I will go on to describe each of the constituent programs that together provide the automation.

The first program was written in a programming language called Python. It

<sup>140</sup> See: <http://mkweb.bcgsc.ca/fun/eespammings/>

<sup>141</sup> See: <http://www.glia.ca/2010/spamHeart/>

acted as a job scheduler (also known as ‘cron job’<sup>142</sup> in programming terms) that fetched a list of spammers from a site called *stop forum spam*.<sup>143</sup> A body of content was then sent to these extracted email addresses one by one and the process was shown on a computer screen (see Figure 5.3). The email’s body included a poem that was composed with my collaborator Susan Scarlata, an American poet. There were a total of 47 poems written in advance through which spam emails were collected over a period of 6 months. An example of an email is as follows:<sup>144</sup>

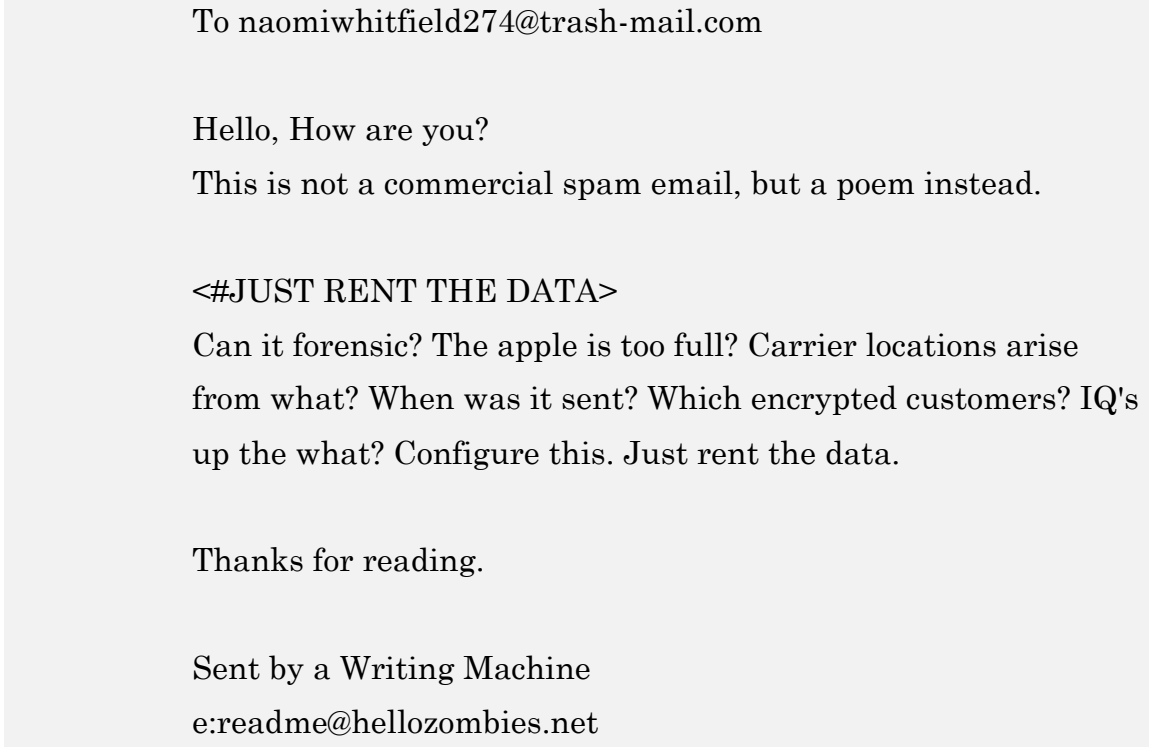
The image shows a screenshot of an email. The recipient address is 'To naomiwhitfield274@trash-mail.com'. The body of the email starts with 'Hello, How are you?' followed by 'This is not a commercial spam email, but a poem instead.' Below this is a poem titled '<#JUST RENT THE DATA>' with the text: 'Can it forensic? The apple is too full? Carrier locations arise from what? When was it sent? Which encrypted customers? IQ's up the what? Configure this. Just rent the data.' The email ends with 'Thanks for reading.' and 'Sent by a Writing Machine e:readme@hellozombies.net'.

Figure 5.2: A spam poem in *Hello Zombies* (2014)

---

<sup>142</sup> See: <https://en.wikipedia.org/wiki/Cron>

<sup>143</sup> Stop Forum Spam is a platform that offers lists of spammers and allows public contributions through their website and the offered APIs. Many other plug-ins are also based on this platform to offer spam filtering services, such as a php extension called Stop Forum Spam for MyBB. See: <http://www.stopforumspam.com/>

<sup>144</sup> It was noticeable that the grammar of an email spam was sometimes not quite right but we decided to use the original spammer’s words to compose a poem.





Figure 5.3: Sending out poems in *Hello Zombies* (2014)

The second program was also written in Python. It periodically checked if there was any new email and automatically deleted email in the *Hello Zombies*' mail server. The mailbox mostly contained the bounce back emails from the spammer's server as well as spammer's occasional replies. The program presented the emails on a screen (see Figure 5.4). The first and second programs together demonstrated the autonomous mechanism of writing and reading of code as emails and the resulting processes of consumption, production and distribution of email addresses.



Figure 5.4: Receiving emails in *Hello zombies* (2014)

Harvesting data with active email addresses is arguably one of the most challenging parts for massive emailing. Security is continuously enhanced in email systems and filtering rules, where the web checking logic that differentiates robots and humans is becoming ever more sophisticated. As a result computer networked agents such as web crawlers and web bots use different techniques such as web data mining,<sup>145</sup> spoofing attacks and dictionary attacks to harvest close-to-live addresses that are able to pass through all the scanning and filtering logics used by email servers and reach the valid end. On some occasions a real email address is stolen through spoofing attacks and spammers “get names and addresses through compromised email accounts, which give them access to contact lists” (Yeaton, 2013, n.p). Whilst in dictionary attacks, spammers use obsolete and invalid addresses to generate a new receipt address by slightly amending the username and replacing the old email domain (such as the change of email address from james1@hinet.net to james@hotmail.com) as close-to-live addresses (Clayton, 2004). Harvesting email addresses has a substantial business value and code contributes significantly to the process of spam data quantification and automation. The focus on these technological and

---

<sup>145</sup> See different techniques of harvesting email addresses: <http://www.private.org.il/harvest.html>

business logics is manifested in the third program, emphasising both the forces of the living and the dead as part of the automated mechanism.

The third program was written in PHP and its function was also related to the activity of reading and writing. This program read the extracted file from the first program and wrote all the email addresses on a webpage. The browser rendered the source code such that it was displayed as rolling text and hyperlinks. The webpage full of the spammer's addresses, consisting of the living dead identities, was displayed on one of the computer screens and was projected onto the wall as well as on the other physical computers that formed the whole installation (see Figure 5.5). The display of such densely packed addresses illustrated the sheer scale of spam and went beyond automatic processing to reveal the mechanism of reproduction in which spam reproduces itself through different identities with distinctive email addresses.



Figure 5.5: Running addresses in *Hello zombies* (2014)

The whole installation can be considered as an automated system that processes information like spammer addresses and spam poems with a common goal. The goal is to continually produce and reproduce information through displaying spam on a screen, sending out poems and fetching spammer's email addresses. These processes of production and reproduction define the three programs as an integral automated system.

Spam acts as automated agents that become part of a wider systemic organisation that is self-regulated. The notions of cybernetic control and feedback systems can further explain this self-regulated organisation. One of the core concepts of second-order cybernetics is reflexivity, informed by John von Neumann's study of automata theory, "an automation was any system that processes information as part of a self-regulating mechanism" (Aspray, 1990, p. 189). Part of this is the way spam is being monitored by various anti-spam techniques and reporting systems. One of the apparent objectives of sending spam is to invade a mailbox successfully where emails are displayed in the 'Inbox' folder and not categorised as 'Junk.' The customisation of the subject line, the body of the content as well as the email address become important as part of the overall logic of spam reproduction. This logic underpins the ability of automated agents to reproduce and modify their actions; spammers become smarter or more intelligent to carry out a specific task at the expense of filtering techniques and the mailboxes' storage space. These agents like many other similar agents that exist in network culture function within automatic and self-regulatory systems.

Following the discussion line of second-order cybernetics and automated agents, it becomes important that a system contains other external systems as well as the internal organisation. It is similar to how media studies scholar Jussi Parikka describes the ecology of a system that takes into consideration "couplings of systems and environments and the self-organization of complexity" (2015, n.p). The materiality of a system, a spam system in particular, is based on the operative and inter-active logics of production and reproduction that are self-regulated. The attention to the materiality of technical objects enables the examination of our techno-

culture beyond the discourses or representations attached to them (Parikka, 2015, n.p). Using computer viruses as his object of study Parikka describes the operative logics as follows:

Viruses do not merely produce copies of themselves but also engage in a process of autopoiesis: they are building themselves over and over again, as they reach out to self-reproduce the very basics that make them possible, that is, they are unfolding the characteristics of network culture. [...] This viral activity can be understood also as the recreation of the whole media ecology, reproduction of the organizational characteristics of communication, interaction, networking and copying, or self-reproduction (2015, n.p).

Reproduction in spam is more than the technical repetition of mail generation as it also incorporates the inter-actions of code and various components within a wider system and media ecology that is always becoming and is reflexively operating as “life-like processes of self-organization, distributed processing and meshworking” (Parikka, 2015, n.p).

Such concerns over ecology and processuality are expressed in the manifestation of the three programs of *Hello Zombies*: harvesting email addresses in parallel to the business and economic logic of obtaining email assets; sending out poems in relation to the productive and reproductive characteristics of network culture; checking and deleting mails as part of the self regulatory system. Spammer addresses in spam production are carefully constructed in the real world such that fake identities resist two-way communication and hide the real server’s sources. Since the sender address field is mandatory in an emailing system this compulsory field may be regarded as a “loophole” by allowing the input of fake identities (Soon, 2015c, n.p). Spammers mostly didn’t reply in *Hello Zombies* and the mail system kept receiving the bounced back emails from the fake server domains. The system automatically executed a mail retry mechanism.

The small programs implemented in *Hello Zombies* do not only demonstrate spamming as systemic processes that obey the instructions given to them. Although, technically, the artwork is comprised of input and generative processes, more importantly, such activities are deployed by automation—the continuing execution of code—that occurs in its deep and operational structure including all the processes like harvesting, composing, sending, checking and deleting emails that are inter-acting with other material substrates. As previously mentioned, one of the important characteristics of a second-order cybernetic system is that it is self-regulatory, this goes beyond an understanding of code as written instructions which governs the behaviour of agents that a machine needs to obey. Indeed the implication of automation is about agency and actions through code execution. Despite automated production and reproduction which follow rules and procedures, a self-regulatory system also enables and disables certain actions automatically. These actions have implications as part of the algorithmic design.

In society instructions (or laws) are something that government entities and citizens follow. The establishment of a law and the execution of it are not automatically run. Chun argues that the characteristic of executable code makes code distinct from law:

code is—has been made to be—executable, and that this executability makes code not law, but rather every lawyer’s dream of what law should be, automatically enabling and disabling certain actions and functioning at the level of everyday practice (Chun, 2008b, p. 309).

With regard to spam what kind of actions have been automatically enabled and disabled? This question relates to automatically filtered emails, “where a legitimate message is categorized as spam (a ‘false positive’)” (Burrell, 2016, p. 7). According to Burrell, some spam filtering algorithms are solely based on the words they contain in order to learn and adapt their assessment in categorising spam and thereby possibly censor emails

incorrectly (2016, p. 8). Some may be correctly classified as spam but many others are mislabeled. This suggests that the algorithms employed are not fully capable of interpretative acts. A consequence of this is the discarding of potentially important emails which end up in the 'Junk' folder instead of the 'Inbox.' Extending this to a wider cultural context the decisions made by the algorithms of a self-driving car, for example, may cause injury or even death. This dual executability (on both code and human) automatically enables efficient self-driving but possibly disables human lives. Putting spam in the wrong folder may seem insignificant when compared to the risk of a human life, however *Hello Zombies* invites audiences to think about possible intrinsic nonhuman decisions within an automatic system, a system that operates without human involvement while it runs continuously and repeatedly in the society that we live.

*Hello Zombies* turns off all spam filtering applications in the mail server. Mail reading and sending scripts (as mentioned in the description of the three programs above) are incorporated into the installation in order to consider the agency of code and algorithm. The artwork keeps sending out poems and receiving spammers' replies as a continuous loop, a loop that has no definite end. To further understand the notion of repetition and loops in code and in the use of algorithms within the context of automated systems, the remaining part of the chapter will address the specific syntax and function of code as well as the logic of algorithms. These topics demonstrate how repetition is deeply implicated in executing automated tasks and how automation acts decisively that forms as a third vector in examining the notion of liveness.

### 5.1.2 Loop

In computer programming, the concept of a loop is highly related to repetition, control and automation. Mathematician Augusta Ada Byron Lovelace first introduced the concept of a loop in the early nineteenth century. She recognised that there were repeatable operations in the early

conceptual design of the computational machine that was regarded as “the first automatic, general-purpose computing machine ever designed” (Kim & Toole, 1999, p. 76), known as Charles Babbage’s Analytical Engine.<sup>146</sup> The concept of a loop, which she called a ‘cycle,’ was conceived in 1834 in her notes<sup>147</sup> on the Analytical Engine which set a precedent for the direction in which digital computers would be later developed. Her notes include (in the form of a diagram) the program procedures, also called the Bernoulli numbers program, of the Analytical Engine. The diagram utilises two loops to indicate the repetition of a set of instructions with conditions (Kim & Toole, 1999, p. 78), minimizing duplicate efforts to write the repeatable operation again. Arguably, the concept of a loop in modern coding practice is highly influenced by her insights into the handling of repeated machine operations, which depict the essence of repetition and condition in a cycle. Modern high-level programming language includes a loop function, allowing a fragment of source code to be repeatedly executed (Nakov et al., 2013, p. 211).

There are different implementation styles of a loop in modern coding practice such as ‘for loops’ and ‘while loops.’ In general, a loop is a repeated execution that continues until a given condition is met. In theory a computer program can execute an infinite loop, meaning a condition is never met. Figure 5.6 shows a simple example that demonstrates a while loop with three lines of code and its output the result on a screen. The first line refers to the assignment of the variable ‘time’ with the value ‘true.’ The bracket in the second line implies ‘time is true.’ The whole second line would then read as, ‘while the variable ‘time’ remains ‘true’ do something.’ This action is indicated in the third line which is to print out the phrase ‘hello world.’ Indeed the program never stops and keeps printing the line ‘hello world’ on a screen. A similar infinite while loop is also implemented in *Hello Zombies* (see Figure 5.7).

---

<sup>146</sup> The Analytical Engine was never implemented because of funding issues (Kim & Toole, 1999, p. 76).

<sup>147</sup> The typed version of the article and her notes are put up by John Walker from Fourmilab, see: <https://www.fourmilab.ch/babbage/sketch.html>



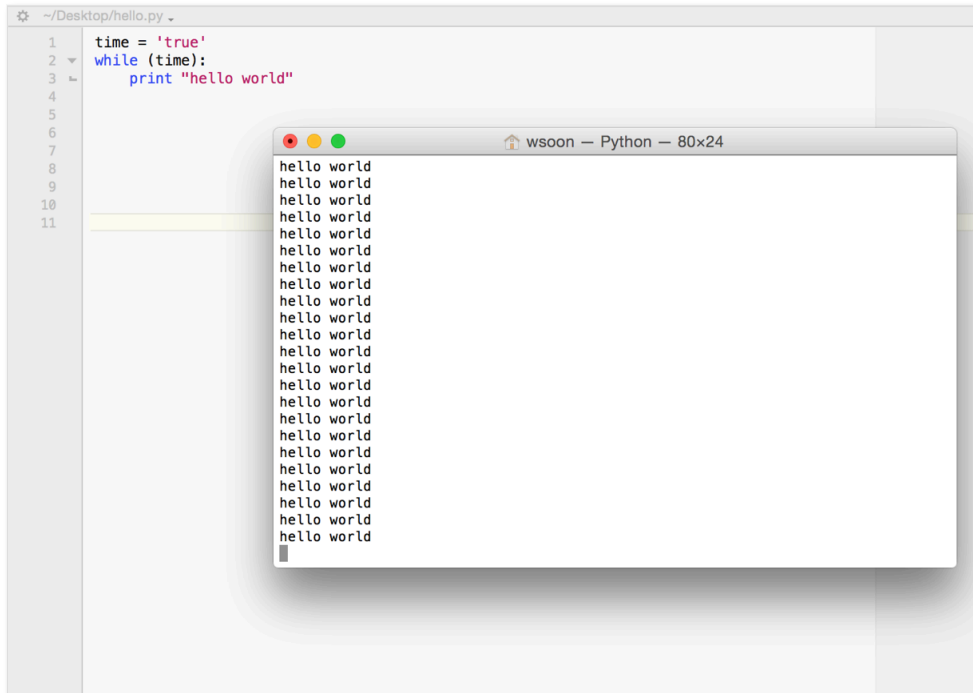


Figure 5.6: A While loop in Python and its result in the Mac OS’s terminal

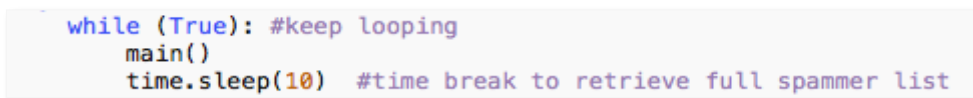


Figure 5.7: An infinite loop in *Hello Zombies*

Referring to Figure 5.7, the condition of the ‘while loop’ is expressed in a boolean state—true or false—and this determines how long or how many times the loop body will be repeated (the loop body refers to the lines below ‘while (True)’ in Figure 5.7). In the conceptual model of a Turing machine, each loop must specify “a finite number of conditions” and the configuration of these conditions “determines the possible behavior of the machine” (Nakov et al., 2013, p. 231). In other words, at least in theory, a loop contains an exit. Whether the machine meets the exit criteria depends on the condition and actual program execution. Therefore if the condition in the loop is always true the machine will keep running which implies there is no way to exit the loop. This kind of loop, as illustrated in Figure 5.6 and 5.7, is considered an “infinite loop” (Ernst, 2009, n.p; Hofstadter, 1980 [1979], p. 157) or as a “strange loop” as cognitive science and comparative literature scholar Douglas Hofstadter describes it as depicting “the concept of infinity.”

It is “a way of representing an endless process in a finite way” (Hofstadter, 1980 [1979], p. 23). This sense of endlessness is set out in *Hello Zombies* as illustrated through the while loop where the condition is discretely finite (true or false) yet it is endlessly processed.

Within the loop body of the above example (in Figure 5.7), the source code instructs the computer to run a particular function called ‘main()’ and to take a 10 second break. These simple lines setup a repetitive structure in which a spam poem is sent out to every extracted spammer address. The loop body ‘main()’ executes these extracting and sending routines, resulting in an excess of communication. This never-ending loop is executing repetitively and endlessly because the condition is always true. In addition, the loop is controlling the flow of sequence routines, in particular when and how the loop body is executed. Therefore its repetition is related to preset conditions and controls. Put simply, while a particular condition is met then a sequence of instructions will be continuously executed as routines. It stops *only* when the condition is no longer satisfied.

To complicate the idea of a condition even more, a “bounded loop” (Hofstadter, 1980 [1979], p. 149) is introduced in Figure 5.8 to illustrate the process of ‘abortion.’<sup>148</sup> This bounded loop consists of a ‘for’ statement and an ‘if-else’ statement that are related to getting each spammer address from a list of emails in the form of a text file. The ‘for’ statement specifies the pointer of a specific address, making sure it will not ask for more than the listed records. The ‘if-else’ statement ensures that every piece of data is a valid email address that comes with the symbols ‘@’ or ‘.’ through moving the pointer. It will only continue to process via a function called ‘sendmail’ if it passes this check (see line 3 of Figure 5.8). Hofstadter describes a bounded loop as follows:

[Loops] perform some series of related steps over and over, and

---

<sup>148</sup> This is a computational term which means to interrupt a computation process. The term ‘abortion’ has a violent connotation, similar to other computational terms such as killing and execution in formal language use.

abort the process when specific conditions are met. Now sometimes, the maximum number of steps in a loop will be known in advance; other times, you just begin, and wait until it is aborted (1980 [1979], p. 149).

The characteristic of a bounded loop is that abortion will occur (at some point) when specific conditions are met. An abortion occurs through  $n$  times of loop processing which might be known or unknown in advance. If the program knows how many email address are in a list in advance the variable  $n$  is known as it will loop for  $n$  times. The use of a specific ‘for-loop’ syntax specifies the range of pointers.

```
for index in range(nodata):
    if any(x in data[index] for x in check):
        sendemail(data[index]) #call send email function
    else:
        print ""
```

Figure 5.8: Bounded loop in *Hello Zombies*

The concept of a loop is fundamental to understanding computational logic and procedures in software (art) practices because it allows conditions and repetitions to run within a confined loop. The projects *Thousand Questions* and *The Spinning Wheel of Life* (discussed in Chapters 3 and 4 respectively) also used the concept of the loop in their implementation. In generative art code generates ‘self-familiar’ patterns which *repeat* themselves to form larger structures while each step and each pattern exhibits only tiny changes and movements (Cramer, 2003; Pearson, 2011). Artist-programmer Casey Reas, for example, has explored this kind of recursive generative art form.<sup>149</sup> Technically, “recursion means that a function can call itself within its own block” (Reas & Fry, 2014, p. 354). This self-familiar arrangement is achieved by the implementation of a recursion function in which a small change is implemented repeatedly. This small change implies having the same logic but just in a slightly different calculated value such that self-

---

<sup>149</sup> See: <http://www.festivalforte.com/festival-forte/generative/>

familiar patterns can be generated. This is considered as one of the ways in which the act of repetition produces differences.

To explain recursion further, Figure 5.9 shows a procedural diagram of making a 3-layer cake by computer science scholar David Schmidt. In order to make a 3-layer cake one needs to make a 2-layer cake and add one more top layer. But to make a 2-layer cake requires adding a top layer to a 1-layer cake. Each iteration is highly dependent on the other and follows the same procedure, thus the initial aim of having a 3-layer cake requires that it is broken down into a few similar steps. Therefore the concept of recursion exhibits repetition, which is similar to a loop and produces differences. Inspired by everyday natural phenomena such as snowflakes, tree branches and blood vessels a fractal structure is a typical example of recursion, “continuing downward in scale” (Pearson, 2011, p. 157) and producing self-similar repetition. Both concepts of recursion and loop require repetition but their subtle difference is that the former emphasises the combined result of all the iterations. Each iteration is based on the previous step to compute and develop the present as well as the next pattern.

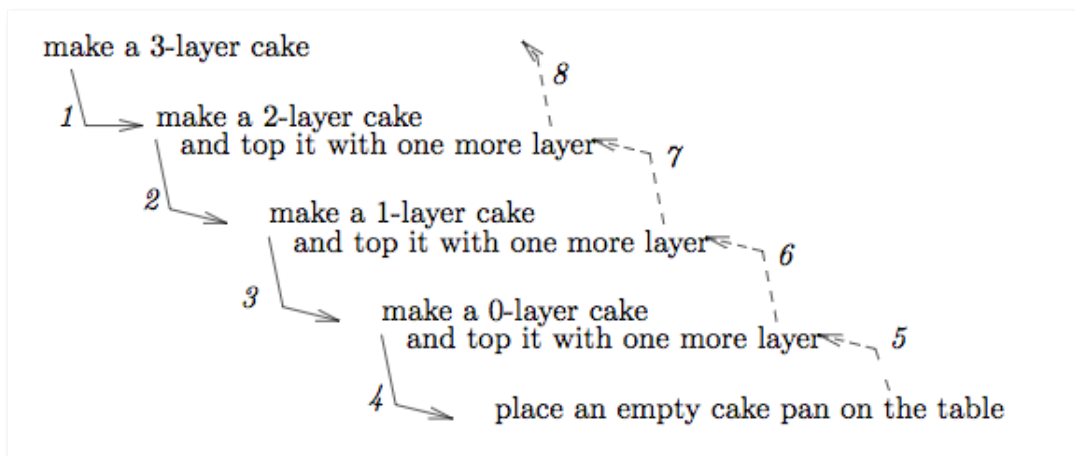


Figure 5.9: The concept of recursion in making a 3-layer cake. Reprinted from Programming Principles in Java (p. 350), by D. Schmidt, 2003. Copyright 2003 by David Schmidt

As such, loops allude to the complexity of repetition: to similarities and differences. Although a machine executes the apparently same written line of code, it can produce self-similar structures (as in the case of generative

art) or it can take in a different input to produce various outputs through data processing (as in the case of the bounded loop in *Hello Zombies*). Just reading the source code is not enough to tell the difference (the repetition is the same in the syntax and written format) because the differences only unfold through the execution and realisation of code that the machine computes in time. In her article titled *The Enduring Ephemeral, or the Future Is a Memory*, Chun proposed a phrase: “code as re-source” (2008b, p. 307), addressing the gap between source code and its execution. This notion of re-source suggests code to be a process rather than a stable written form (Chun, 2008b, p. 321). As demonstrated by the use of the infinite loop, bounded loop or recursion in software (art) practice, these processes can be run endlessly and changes can be made in a very subtle way. It is, therefore, through running and executing code that computational is realised and emerged over time.

In the earlier discussion of the Fetch-Execute Cycle (in Chapter 4, section 4.2.1), the read/write process of memory was seen to play a significant role in computation, generating micro-instructions as a sequence of operational steps. Chun highlights the fact that computer memory is impermanent and volatile because of its “constant degeneration” (2008a, p. 148) and further reminds us that repetition is more than simply speed and, subsequently, we need to think beyond speed to the ways in which constant repetition is also related to stability and ephemerality (2008a, pp. 148-53). She explains how the notion of ‘ephemeral endure’ is entangled with the unstable and ephemeral computer memory as well as the endless act of forwarding, updating and circulating of data. My point is that, technically, these acts are run through many different kinds of loops in computational systems. Within the live dimension of code inter-actions within loops, regardless of what the pre-set conditions in a loop actually are, these endless acts are regarded as processes that constitute the ephemeral endure. An automated loop does not at all guarantee that things are stable.

### 5.1.3 Open or die

The syntax of ‘open or die’ can be used as a further example of a way to address the instability issue in computation. Following the von Neumann architecture model of 1945, a CPU interacts with memory, executes arithmetic instructions (via the Fetch-Execute Cycle), and performs input/output (I/O) operations. These I/O operations do not necessarily render data that is stored within the machine (such as user input on a mouse click) but also enable an exchange of data with the external world (von Neumann, 1945, p. 3).

Code inter-acts with different kinds of data, stored within a machine or outside of it. Figure 5.10 shows one of the I/O operations which is programmed with PHP in *Hello Zombies*.

```
#### get file ####
$filename = "list_output/list.txt"; #DEFINE file name of the spammer list
$myfile = fopen($filename, "r") or die("Unable to open file!");
$email = array();
while (!feof($myfile)) {
    $email[] = fgets($myfile);
}
fclose($myfile);
##### end of get file #####
```

Figure 5.10: I/O operations in *Hello Zombies*

The significant syntax is ‘fopen() or die()’ in line 3 which points evocatively to the dichotomy of the living and the dead within the context of an I/O operation. For this particular line of code, ‘\$myfile = fopen(\$filename, “r”) or die (“Unable to open file!”);’ means that the computer will open the designated file, read the file and place the ‘file pointer’<sup>150</sup> at the beginning of the file. However, if there is any error during accessing the file, for example, if the file is protected, unavailable or corrupted, the program will then stop immediately without running the rest of the program. As a result, an error message is shown on the webpage of *Hello Zombies* as indicated below as an illustration:

---

<sup>150</sup> This file pointer is not something one can observe directly in a programming environment. It is used as a marker to indicate the position of data records in a file as part of the input reading operation. It can be adjusted, reading from the beginning or from the end of the file for example. Besides this a pointer can be moved while it is being read to indicate the current pointer location (See: [http://www.onlamp.com/pub/a/php/2002/12/12/php\\_foundations.html](http://www.onlamp.com/pub/a/php/2002/12/12/php_foundations.html)).

```
Warning: fopen(list_output/list.txt) [function.fopen]: failed to open stream:
No such file or directory in
/home/freehandhk/domains/siusoon.net/public_html/test/index.php on line
54 Unable to open file!
```

*Figure 5.11: An error result*

As such and this is my point, the syntax ‘die’ here serves syntactic, semantic and performative functions. The function ‘die()’ is scripted in PHP, an open source scripting language that is used for web development. Following a strict syntax associated with the language in which if there is any message required to display when a program encounters an open file error, it must be specified within brackets. On a semantic level, as stated in the PHP manual documentation,<sup>151</sup> the function of ‘die’ is equivalent to the function of ‘exit,’ meaning the current script will be terminated after outputting a predefined message.

Within the design of PHP, the function of ‘die’ is regarded as the alias of ‘exit’ and as a ‘master’ function. Indeed the syntax of ‘die’ and ‘exit’ also occur in other programming languages such as Perl and C. Functionally speaking, as indicated in the documentation, both ‘die’ and ‘exit’ are equivalent in PHP. However there are also very subtle differences, as for example the memory space occupied by the one character difference between them. Additionally and more importantly, if there are any cleanup activities performed by PHP in the future to remove redundant syntax, an alias (in this case the function of ‘die’) is regarded as secondary when compared with the master (in this case the function of ‘exit’). As a result, potentially the syntax of ‘die’ may be removed and made obsolete. Consequently this function (‘die’) will no longer be supported in the future version, implying that this would possibly impact the whole ecology of programming practice (this is similar to the notion of the inexecutable query discussed in Chapter 3, section 3.5). It can be said that if choosing between the syntax of ‘die’ and ‘exit’ the latter is recommended as it is regarded as the ‘master.’

---

<sup>151</sup> See: <http://php.net/manual/en/aliases.php>

According to the Oxford Online dictionary, the word ‘master’ historically meant, “a man who has people working for him, especially servants or slaves.” It is a noun which denotes a particular gender. It also means that a master with power “has complete control of something” (“Master,” n.d.). This gender and power hierarchy is not only implemented at the syntactic and semantic level but it is also performed through the activities around it (the potential clean up activities and PHP’s stated priorities that favour the master versus the alias for example). Therefore, the notion of the syntax ‘die’ not only refers to the failure to open a file in a functional way but also to the inherent gender and class structure that may give rise to potential changes in the future. Performativity, therefore, cannot be understood as mere “technical performativity” (Arns, 2004, p. 186). The conflation of both the linguistic and execution layers exhibit performative acts (see the discussion of language and performativity in Chapter 2, section 2.2.2) and these acts may not expose an immediate impact (such as a file being open or not at a given moment) but rather result in on-going effects on programming practice in general.

The syntax of ‘die()’ demonstrates error handlings within code. The handling of errors is always inscribed in code writing regardless of how they are actually encountered. This specific syntax demonstrates how deadness is deeply embedded in high-level programming language. These dual states of the liveness and deadness of code are inevitable in software (art) practices. It is worth noting that the concept of deadness does not refer to the literal syntax of ‘die’ but applies to different forms of disruption and obsolescence (see Chapter 3), interruption and absence (see Chapter 4), errors and malfunctions which constitute the notion of liveness. As argued in the previous chapters, disruption is always implied in executing query and liveness is about the possibility of inexecutable at any time in contemporary (update) culture. In addition, the occurrences of micro-decisions and micro-interruptions within networked protocols engender the immanent experience of streams as part of the networked culture. Both forces, the living and the dead, constitute the notion of liveness. These forces co-



existed. Even though it may be in a live condition in which a program runs smoothly, its death is always implicit (as for the discussion on ‘open or die’ syntax). Both arguments, unlike the syntax and semantics of ‘open or die’ which promote the distinctive split, represent a coupling of forces that has to be understood as an entanglement that constitutes the very phenomena of liveness. I argue that liveness always implies deadness. From the inner writing of code to its actual realisation of code execution, liveness and deadness are not two separated concepts, but they are entangled in their material relations. The forces reiterate the very nature of liveness that appears to echo Barad’s idea of “things-in-phenomena” (2007, p. 140),<sup>152</sup> in which liveness is a phenomenon that exhibits an on-going dynamic process of materialisation.

In executing queries and streams, as articulated in the previous chapters and further emphasised in this chapter through additional examples of the various programs developed for *Hello Zombies*, liveness implies instability and unpredictability in contemporary software culture. What is added in this chapter is the thinking of ephemerality through the notion of undeadness and the execution of automated systems. Instead of the distinctive split of the ‘open or die’ in which a program will terminate upon failure, the next section will demonstrate how a system can still continuously run even an error is encountered, attaining the enduring ephemeral.

#### 5.1.4 Try and catch exceptions

Exception handling can be used in software (art) practice to allow constant repetitive events to run continuously regardless of errors that occur during run-time. The mechanism of handling exception is widely built into current programming languages such as C++, Java and Python. Some run-time errors are critical and will make a program to stop immediately while some are less critical and a program can still be made to run continuously. It

---

<sup>152</sup> See Chapter 2, session 2.4.2.

depends on how a program is designed to handle these errors. These 'run-time errors' are different from language errors (also classified as 'compilation errors') in which errors cannot be detected in advance but only through running the program in real-time. Programmers possibly anticipate them. For example in the case of *Hello Zombies*, it is possible that the spammer list file may be corrupted resulting in an error during the retrieval process. Therefore, an error may be anticipated but no one knows when will it occur or whether will it occur at all. Since there are many different possible failures which can occur within a machine, from CPU and hardware failure to memory allocation issues as well as network and protocol communication problems, it is impossible for a program to handle and catch every specific error (Louden & Lambert, 2012, p. 424). Therefore, it can be said that countering run-time error is something operative between the predictable and unpredictable states.

The handling of run-time errors can be traced back to the Ada programming language that was originally developed in 1983. The project was led by Dr. Jean Ichbiah in France and the name was chosen in honor of the pioneer of programming Ada Lovelace (1815-1852). Ada is an object-oriented and real-time programming language. It was also a widely used programming language until the introduction of other programming languages as C++ and Java which are designed for real-time systems. Advanced exception handling, for example handling different types of exceptions and their detection, handling and propagation (Chapman et al., 1993, p. 149), is one of the major contributions of Ada. This was acknowledged by Lawrence Collingbourne, the editor of the book *Ada: Towards Maturity*, he states,

One of the major contributions Ada has made to programming has been to recognise that errors do occur at run-time and therefore to provide a way of handling these by means of exceptions (Collingbourne, 1993, p. 2).

Prior to Ada exception handling was implemented in the language P/L in the 1960s. According to computer science scholars Kenneth C. Louden and

Kenneth A. Lambert,

Exception handling is an attempt to imitate in a programming language the features of a hardware interrupt or error trap, in which the processor transfer control automatically to a location that is specified in advance according to the kind of error or interrupt (2012, p. 424).

This is one of the programming designs used to ensure “security and reliability” in which a program is able to “recover from errors and continue execution” (Louden & Lambert, 2012, pp. 423-4). In other words, exception handling can be regarded as crucial in automated systems in which tasks can still possibly be run and execute continuously. In Python exception is handled by the syntax called ‘Try and catch exceptions’ in which a form of control is established as it governs the flow of a program when errors are encountered. To explain further, an exception contains two parts; First is the ‘abortion’ of the partial current computation and it causes a jump from one program point to another; Second is the handling of the exception which may allow certain functions/statements to be run and respond to exceptions raised. Below is an example of an exception in *Hello Zombies* which is an extension of Figure 5.7.

```
try:
    while (True): #keep looping
        main()
        time.sleep(10) #time break to retrieve full spammer list
except KeyboardInterrupt: #press control+c to interrupt the program
    sys.exit(1)
```

Figure 5.12: Try and catch exceptions (1) in *Hello Zombies*

Figure 5.12 indicates a ‘while loop’ within a ‘try and catch exception’ structure. It means that the program will keep trying to run unless there is an interruption from the press of a keyboard button. In this case, a ‘try and catch exception’ exits a constructed ‘while loop’ when an exception is caught even though it was supposed to be a continuous ‘while-loop.’ Once a keyboard button is pressed the program jumps out of the block of the ‘while loop.’ The handling of such an exception in Figure 5.12 means it will exit the

entire program with the syntax 'sys.exit(1)' that was preset in the last line. More explicitly, in this case, pressing the keyboard button would stop sending spam poem to a spammer. The last line indicates what a program should do after catching an exception. This exception refers to an 'abortion' that is made by a human to stop, or 'kill,' the running of the program. The corresponding exception handling, in this case to exit the program, is also considered as a means by which to control the continuation or stopping of a program during run-time. Additionally, an interruption by pressing a keyboard is listened by the program during the whole running time. This literally means whenever a keyboard button is pressed the program will stop accordingly.

Figure 5.13 (an extension of Figure 5.8) shows another type of exception that is set up to catch the nonhuman exceptions that would make the program run continuously instead of stopping the program. This exception has nothing to do with human interruption but is related to the live conditions of a networked environment. For instance, there may be errors when getting a spammer address list or reading the file list or problems with network protocols such as POP3 emailing or TCP/IP networking. This type of exception frequently appears in the source code of *Hello Zombies* as it exchanges data from/to the outside environment. To further clarify this example, exchanging data means sending out/check emails through the POP3 protocols, reading an internet file for the poem selection or extracting spammer addresses through internet protocols.

```
try:
    dsta0 = urllib2.urlopen("http://www.hellozombies.net/getmails.php") #execute php script to extract the new list
    dsta0.close()
    print "... getting mail list"
    time.sleep(10)
    elist_data = urllib2.urlopen(elist_url)
    data= elist_data.read()
    elist_data.close()
    data = data.split("\n") # then split it into lines
    nodata = len(data) #check no of arrays
    #nodata = 10 #temporary no of arrays to avoid massive email sent out
    check = ['@', '\.'] #just simple email parameter check
    for index in range(nodata):
        if any(x in data[index] for x in check):
            sendemail(data[index]) #call send email function
        else:
            print ""
# except urllib2.URLError, errormsg:
except Exception, errormsg:
    error_handling(errormsg)
```

Figure 5.13: Try and catch exceptions (2) in *Hello Zombies*

These exceptions cannot be detected or checked prior to running the

program insofar as the statements or expressions are syntactically correct. Errors detected during run-time and exceptions are caught. The ‘try’ and ‘except’ keywords (as shown in both Figures 5.12 and 5.13) indicate the *possibility* of an error occurring but this does not mean it will necessarily happen during execution and cause vital disruption. To explain further, the ‘try’ syntax specifies the expression and instruction. Semantically it implies the execution of this block of code with uncertainty because it *may* fail: errors *may* be detected during execution. If there is no exception occurs, the ‘except’ clause is omitted automatically. However, if an exception occurs during the running of the program the rest of the clause within the ‘try’ block is then skipped and the ‘except’ clause takes over. To keep the code running (after executing the ‘except’ block) the program will continue to run the rest of the code. If the program is set to repeat itself it will then run the ‘try’ block again. The idea of the program design behind *Hello Zombies* comes with an assumption that an error may be recovered automatically, for example an unstable internet connection or server failure at a particular point in time.

With this technical explanation of the ‘try and catch exception’ we can understand how a program is made to continuously run and made to be operative at the level of code with the syntax ‘try and catch exceptions.’ Unlike the specific I/O error that demonstrated earlier with the syntax of ‘open or die,’ there are other types of error that cause exceptions such as run-time and network errors and type errors. The syntax ‘pass’ can be used in Python to indicate the bypassing of any non-defined procedures when an exception is caught. Therefore, it is possible to catch an exception without specific procedures for handling it and hence to bypass it. When a loop is operating within the block of a ‘try and catch exception’ it might be considered to be close to what Chun describes as “a battle of diligence between the passing and the repetitive” (2008a, p. 167). The computational logic automatically bypasses and handles exceptions and this enables the continuous running of a program with a very specific form of repetition. Such tensions again fit the notion of the enduring ephemeral. However, such procedures and control are made invisible to audiences and end users as the

program keeps skipping, looping and executing. What remains is a program that is continuously *trying* to perform smoothly in which code inter-acts with protocols, files, lists, data and so forth. 'Try and catch exceptions' illustrate how ephemerality is made to endure in a live networked environment.

In computer science exceptions arose as a mechanism for handling errors during run-time and for efficient debugging. However, central to my concern with code inter-actions related to 'try and catch exceptions' is also the additional reflections on continuous processing and automated execution that are expressed both in the inner writing of code and that are evident in wider cultural matters like spam production, virus attacks and the update and upgrade culture of software and platforms. These matters are automated in a continuous manner that is set out as a series of *norms* which define the criteria for exceptions under specific conditions. In her article *Crisis, Crisis, Crisis, or Sovereignty and Networks*, Chun discusses the production of norms and exceptions in the context of crisis. She argues that "crises are new media's critical difference: its norm and its exception" (2011a, p. 92) and yet, "crises do not arguably interrupt programming" (2011a, p. 99). Apparently Chun's concept of uninterrupted exception is similar to the programming syntax of 'try and catch exceptions' used as one of the strategies to keep a program running continuously. Imagine web platforms like Facebook and Google services, the seamless background update of new interfaces,<sup>153</sup> new features and, perhaps most significantly, data mining strategies and new tracking mechanisms of which users are not fully aware of. Similarly the constant automated call for operating system upgrades and app updates in mobile devices create new cultural norms of interfacing with software, applications and devices. From a technical perspective outdated clients and versions of software might be regarded as exceptions as they do not fit into the new environment, resulting in the suspension of services.<sup>154</sup> Not until a user updates or reinstalls a new

---

<sup>153</sup> The Wall Street Journal reported on a new feature of Facebook which tracked a user's cursor. See the article titled "Facebook Tests Software to Track Your Cursor on Screen":

<http://blogs.wsj.com/cio/2013/10/30/facebook-considers-vast-increase-in-data-collection/?mod=e2tw>

<sup>154</sup> It is indeed fairly common to experience outdated software versions and clients that are forbidden from running. See examples in the areas of game and social media:

version can the user be considered to be a normal user. Such rules and exceptions create new norms that “are intriguingly linked to technical codes and programs”(Chun, 2011a, p. 99).

A further reference can be made to Giorgio Agamben’s political and philosophical understanding of “the state of exception” (1998) in which the sovereign has full power to set out, apply and suspend a norm. With regard to the software culture of constant update/upgrade, some of the legacy clients might still be eligible for limited services but many of these clients are suspended and excluded from further participation. What is eligible, or not, is constantly shifting over time. Yet Agamben suggests that the notion of exception is more than just an exclusion but something that operates in between inclusion and exclusion: “The exception is what cannot be included in the whole of which it is a member and cannot be a member of the whole in which it is always already included” (Agamben, 1998, p. 21). Those excluded parties still remain as “members” but different layers of control and services can be implemented and provided to these *excluded members* by software sovereignty. Thinking through the logic of ‘try and catch exceptions,’ exceptions may be caught and handled with another pathway that deviates from the norm but is regarded as the rule in itself. New standards and policies are set out through the constant and seamless update of systems, platforms, applications and devices. In response to this a suspension is a deferred, “[deferring] the future it once promised” (Chun, 2011a, p. 98).<sup>155</sup>

Rules and exceptions are both implemented at the level of code and are conflated to be almost indistinguishable. In *Hello Zombies* the ‘try and catch exception’ falls under the scope of a larger structure of a loop. Therefore, the computational process repeats itself extremely fast, at a speed that is beyond human perception of micro-processing time. But we are reminded that an exception is also a way to set out the norms.

---

(1) <http://forums.na.leagueoflegends.com/board/showthread.php?t=621651>

(2) <http://crackberry.com/psa-dont-sign-out-instagram-or-youll-be-right-back-where-you-started>

<sup>155</sup> This is poetically expressed in the Gilbert and Sullivan opera *The Mikado* (1885) through the lyrics written by William Schwenck Gilbert: “Defer, defer, To the Lord High Executioner.” See the song “Behind the Lord High Executioner:” <https://www.youtube.com/watch?v=u1qd3bwv3N4>

Exceptions in programming can be understood in terms of the state of exception, limiting, suspending, interrupting and deferring future possibilities via code execution and justified in terms of security or maintaining the norms of sovereign rule. Code is put into action using words with executive power, as Chun asserts, “[c]ode as law as police, like the state of exception, makes executive, legislative and juridical powers coincide” (2011a, p. 101). Thus, code expresses nonhuman agency and “embodies the power of the executive” (Chun, 2011a, p. 101).

Exceptions and loops are programmed to enable and disable certain activities in the process of maintaining the perpetual running of code. Code inter-acts with exceptions, both at the level of programming and in states of exception, rendering the undeadness of code through the multiple meanings of the term execution across the realms of computer science, contract law and death. This subsection of ‘try and catch exceptions’ explains exceptions in a technical manner and demonstrates the possible implications of a state of exception that runs automatically. In this way, an examination of the live dimension of code inter-actions demonstrate how undeadness is exhibited beyond inner memory writing through running code syntaxes, such as ‘loops,’ ‘open or die’ and ‘try and catch exceptions,’ in computation which explicates cultural implications.

## 5.2 A sense of ending in algorithms

While the previous section explained the continuation of running code by analysing specific syntaxes, an account of how *tasks* are automated and how code is related to tasks is still missing. Algorithm is a more abstract term that is not dependent on any specific programming language or syntax but describes a step-by-step procedure used to achieve certain tasks through computation. The understanding of the notion of any given task may be different between human and machine. For instance, the general task for *Hello Zombies* is to demonstrate autonomous reading and writing activities in relation to spam. In order to achieve this human task this has to be broken down into many different steps. Figure 5.14 shows the high-level



logic of *Hello Zombies* and indicates the procedures, processes and conditions of the three programs. The diagram is structured in such a way that a machine could understand the procedures in detail when they are further transformed into program code.

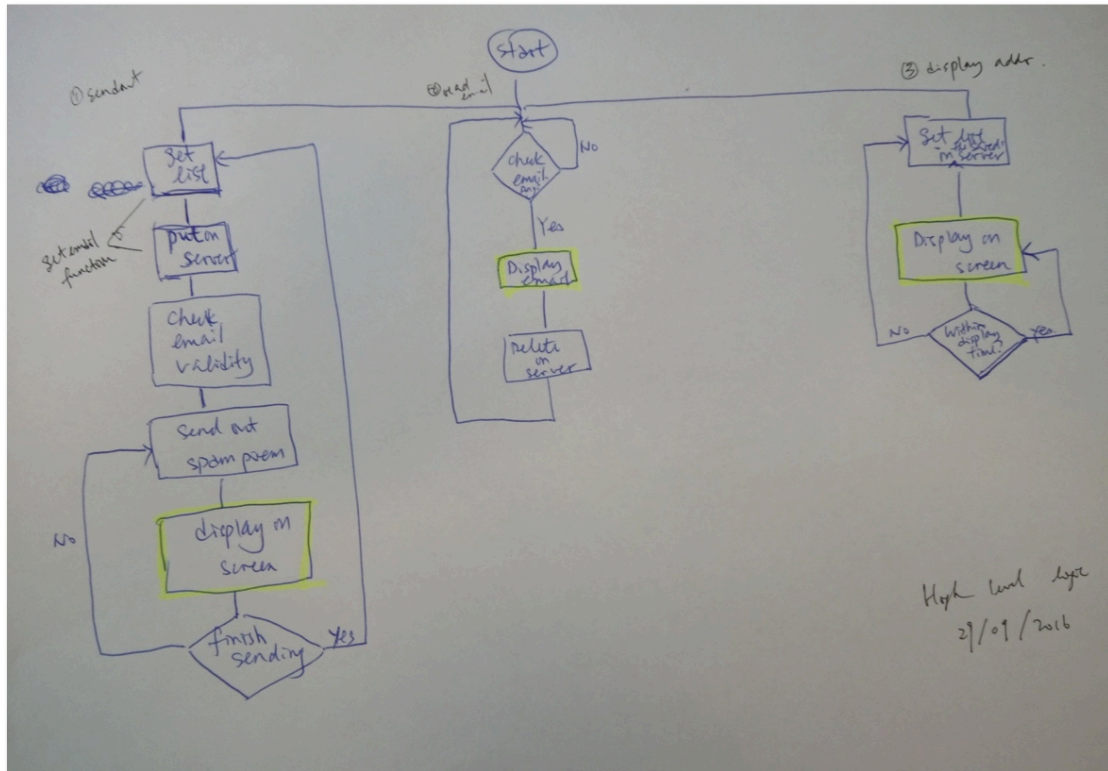


Figure 5.14: A high-level flowchart of *Hello Zombies*

The flow chart in Figure 5.14 shows a breakdown of tasks representing procedures, step-by-step progression and algorithms. It is not a detailed flow chart that includes every possible step but rather it demonstrates high-level processes for visual understanding that gives a general idea of how tasks are broken down. It is independent of any programming language as it concerns procedures but not coding's syntax. Computer scientist Robert Kowalski specifies that an algorithm consists of both logic and control components (1979). An algorithm generally refers to “the knowledge to be used in solving problems” and “problem-solving strategies” to achieve a computational task (Kowalski, 1979, p. 424). The logic also includes a “relational component” for things like data processing handling (Kowalski, 1979, p. 425). Figure 5.15 shows the various components involved in an

algorithm which influence how it behaves.

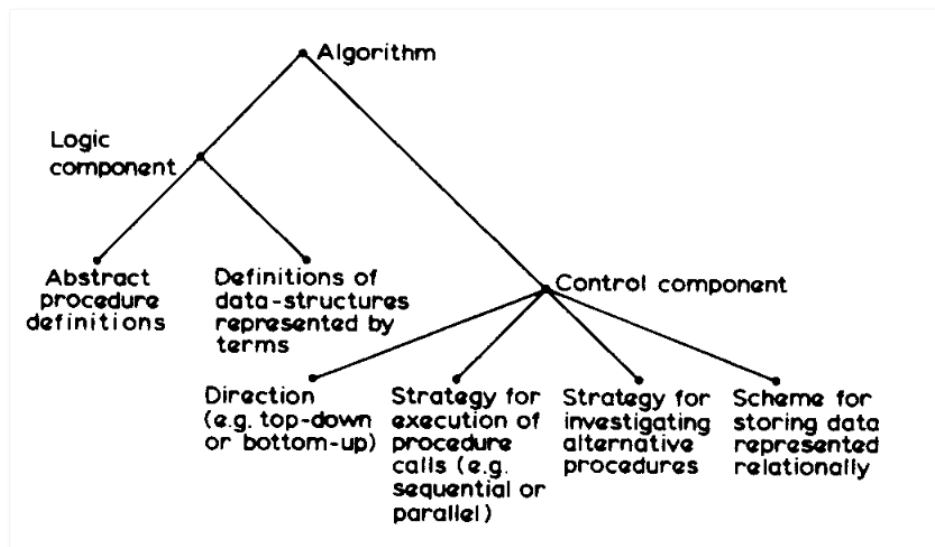


Figure 5.15: Decomposition of algorithms. Reprinted from “Algorithm = Logic + Control,” by R. Kowalski, 1979, Communications of the ACM, 22(7), p. 425. Copyright 1979 by ACM.

Among the various components of an algorithm, direction might be technically understood as a way to derive a sequence. Kowalski offers two different directions: ‘bottom-up’ and ‘top-down,’ controlling the efficiency of a running algorithm (1979, p. 425). This implies that algorithms consist of both spatial and temporal dimensions. In the context of this thesis, this would also imply that liveness has a direction.<sup>156</sup> Reaching an ending is directional implying movement from one stage to another, from a defined problem to a solved problem, from start to stop and from a state of living to inevitable death. To move beyond liveness as something human-oriented, a more pertinent question might be: How does an algorithm achieve its task? Is there a sense of ending in live running algorithms? Given the possibilities of different implementations of loops and, in particular, the possibility of an infinite loop, is there an end? What are the implications if there is no definitive end?

Indeed, in his article *On Computable Numbers, With an Application to the Entscheidungsproblem* (1936), Turing demonstrated a mathematical proof

---

<sup>156</sup> The use of the vectors in this thesis implies forces with direction (see Chapter 1 for the vector and force discussion).

that there are problems which cannot be solved computationally (the mathematical proof is demonstrated through contradiction). Theoretically there should be “an end after a finite number of steps” (Turing, 1937, p. 247) but in fact the computer cannot decide whether an arbitrary program will end or will not end. To put it another specific way, a computer cannot solve this question: Can a program (Program A) process and read another program and its input (Program B) and decide if that program will halt provided that the answer/output to this question is either Yes or No. Turing proved that this problem is undecided and unsolvable. One of the reasons is that an output of the Program B can enter into a self-constructed repetition, an infinite loop, in the Program A that makes it never cease and contradicts the output result ‘Yes’ in the Program B. Similarly, the Program A could process the Program B and its outputs and despite arriving at a ‘No’ result but the Program A actually performs the halting (Booher, 2008, p. 3). These scenarios suggest that there is a contradiction between a program’s output and performance.

Figure 5.16 illustrates this halting problem with a sketch in the form of Python code simulating the conditions in which Program A contains the Program B. The function ‘DoesItHalt’ in line 8 assumes that there is a procedure defined for processing the other program and its input (in this case Program B) which can determine if another program can be halted, or will otherwise loop infinitely, outputting the result (‘Yes’ or ‘No’) in lines 9 and 13 respectively. Lines 10-11 show that the program will continue to loop even though the result of the assumed function ‘DoesItHalt’ outputs the result as ‘Yes’ (as indicated in line 9). In a similar vein, the program will halt if the output result is ‘No.’ This example is further illustrated in a graphical form in Figure 5.17 which explains the input, output and the contradiction between the output and the actual performance of the program. Turing called this problem the Entscheidungsproblem (German for “decision problem”<sup>157</sup>) and he asserts, “Entscheidungsproblem cannot be solved” (Turing, 1937, p. 262).

---

<sup>157</sup> See: <https://en.wikipedia.org/wiki/Entscheidungsproblem>

```

1  ▾ #ref1: http://sahandsaba.com/static/presentations/it-from-bit.html#/10
2  ▾ #ref2: https://www.explainxkd.com/wiki/index.php/1266:_Halting_Problem
3
4  #assume this is another program called programB
5  programB= raw_input("Get input: ")
6
7  #assume the function 'DoesItHalt' is a procedure that solves the halting problem
8  ▾ if DoesItHalt(programB):
9      print "Yes"
10 ▾     while (True): #performs as infinite loops i.e not halt
11     ▾         pass
12 ▾ else:
13     ▾     print "No" #performs as an end i.e halt

```

Figure 5.16: An idea sketch<sup>158</sup> of Turing’s halting problem in Python.

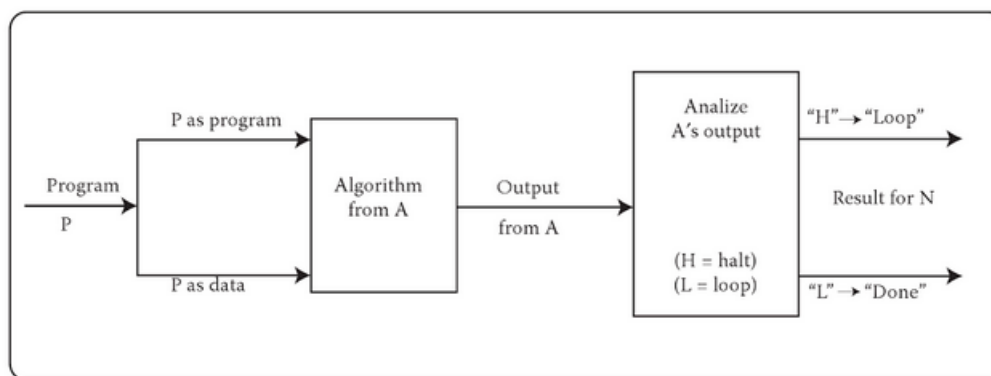


Figure 5.17: The construction of N. Reprinted from *The Tao of Computing* (p. 172), by H M. Walker, 2013. Copyright 2013 by CRC Press

The implication of the problem is that there are things that are impossible to solve and cannot be decided but what is relevant to this chapter, and to the overall argument of this thesis, is that the undecidability to halt shows a contradiction between liveness and deadness at the material level (code and algorithms) in computation. Results and actions (loop or halt) contradict each other. In this way, not only does the endless running of algorithms signify a coupling of living and the dead, but it also draws attention to the contradiction itself, conceptually as well as computationally. Turing’s halting problem suggests that computer systems have blind spots: how code is written and performed can be contradictory with each other.

There are numerous scholars who draw upon Turing’s work and discuss this

---

<sup>158</sup> This is a much simplified version of Turing’s halting problem in which he offered the detailed steps of solving the halting calculation (Turing, 1937). Figure 5.16 cannot be run in Python because the ‘DoesItHalt’ function has not yet been defined. In the last section of this chapter there are some other exploratory sketches that can be compiled and executed in which the function can generate either True or False results.

notion of “ending” (or ‘halting’) in relation to computation (Chun, 2011a, 2011b, 2016; Ernst, 2009; Parisi & Fazi, 2014) but not enough attention is paid to the computational contradictions and their implications in software studies. In his essay “...*Else Loop Forever*”. *The Untimeliness of Media*, Ernst takes the perspective of media archeology to examine this sense of ending at the level of medium. Informed by information theory, a medium is a transmission channel with a sender and a receiver, a beginning and an end, an input and an output. Ernst explains that a channel “always remembers the input and anticipates the output” (2009, n.p). This sense of ending exists and it is highly related to time. As Ernst states, it is “temporally ephemeral” in nature (Ernst, 2009, n.p).

A sense of ending can be interpreted as a perceived sense of time towards an end or towards the completion of the running of an algorithm or process. Theoretically when it comes to an end or completion of a task the program stops. In terms of computation a sense of ending can be understood as how a computer decides when will be the time to achieve an end, signifying that a set problem is solved. Ernst formulates this question as: “can a problem be solved [within] a limited time or not?” (2009, n.p) He indicates that through a close reading of the Turing machine it is impossible to solve these kinds of questions which are associated with the matter of time. His question is a transformative rephrasing of Turing’s halting question, as Ernst claims it is “a time-critical question” (Ernst, 2009, n.p) yet it is also a question which produces a yes or no answer. Ernst’s question concerns the sense of ending in an algorithm as it is associated with time, infinity and temporal processuality. Ernst draws on a theoretical understanding of the Turing machine that is based on the model of finite automation in which it is assumed that there is endless storage tape available. Despite the possibility of running out of tapes, Ernst describes these kind of infinite loop as displaying endlessness on a computational level: it “loop[s] forever” (2009, n.p). There is no absolute end throughout the computational process and the fundamental structure of repetition changes the way we sense an end in computation. As Ernst explains, “[t]he configuration of a loop, the iterative principle, and recursive procedures are the predominant chronotropes in

computing time” (2009, n.p). His attentiveness to computational time leads to the notion of ephemerality in which although the notion of endlessness describes the act of looping forever, it is also ephemeral as “the temporal aura” is lost over time with repetition (Ernst, 2009, n.p). This ephemeral characteristic resonates with Chun’s use of the term ephemeral and her concept of undeadness in which memory in digital media is not stable and, unlike storage, undergoes constant degeneration. Instead of focusing on automated tasks, both Chun and Ernst’s notion of ephemerality share a similar perspective on the lost in computation at the level of memory and temporality respectively.

Referring back to Figure 5.14, the high level flow chart of *Hello Zombies*, three different programs are built with loops which automate the tasks of sending out poems, checking emails and fetching spammers’ address list. These automated tasks can be understood as not constituting a full completion because the programs, in theory, never halt. It is said to be not a full completion because the programs, once started and executed, send some poems out, check and delete some emails and fetch some lists of spammers. Various tasks are completed over time but no definite duration can be indicated as to how long do the programs will take to complete *all* of the tasks. The notion of automation alludes to the sense of endlessness with the implementation of infinite loops that exhibit the loss of the “temporal aura” (Ernst, 2009, n.p) in the work of *Hello Zombies*. An end is never attained, at least in theory, therefore what constitutes an end is never clear in the *Hello Zombies* project. To put it differently, the function and result of the endless running becomes the task in itself: the task is to execute tasks endlessly. On the one hand, the programs are implemented using the infinite loops function which results in a loop which will run forever; on the other, the project reflects on the contemporary condition of the endless reproduction of spam and extends this to the dynamic network processes that constitute the continuation of the drive to the overall concept of ‘undeadness’ (Chun, 2008a, p. 165). in contemporary software culture.

From the previous discussion of the reproduction of spam to the

contemporary phenomena of internet messages that are created and forwarded endlessly, different temporalities are produced that consequently change the value of data in contemporary software culture. The value of data points at the *latest* information, as in a news feed or in the updating of a conversation in social media that reflects the current reality of the world. Printed newspapers and magazines used to charge a certain amount for offering a worldview of the *now*. Similar content is observed in the digital world but what is different is that latest or most recently updated information is usually free. In the digital world it is the archive, the old, which is considered to be a valuable asset (Chun, 2011a, p. 98). Commonly many online service providers have adopted the model of charging for access to archives or for big data extraction. In particular this kind of endless forwarding and response to the old creates business and research values that investigate user behaviours, trending topics and data prediction. As Chun reminds us, “Repetition produces value” (2011a, p. 98).

What executable code has automated is not only computational tasks but also human decisions that produce value through the different temporalities of data. This has been discussed in Chun’s article, *Crisis, Crisis, Crisis, or Sovereignty and Networks*, in which she addresses the rapid response of computation to our actions that automate decision-making (2011a, p. 98). Instead of focusing on time and monetary value, perhaps one of the more pressing issues in contemporary culture is the increasing phenomenon of machine learning algorithms. The queries of data mining, personalisation and machine learning are based on large amounts of stored data to make automated decisions. In particular, machine-learning algorithms are commonly required to solve “classification tasks.”

Spam filtering is a case in point as it utilises machine learning algorithms, taking different information from an email such as a header information and the body of the text, to categorise an email as spam, or not (Burrell, 2016, p. 5). Algorithms are set to not only query massive quantities of data, but also to “*learn on training data*” that train a spam classifier (Burrell, 2016, original emphasis). Machine learning algorithms can then assign a

“weight measure” for certain words that are used to classify spam emails (Burrell, 2016, p. 7). As a result what has been automated are the processes of adaptive learning and decision-making used to classify the inclusion or exclusion of things that are not entirely human-oriented. If we think about the sense of ending in algorithms, this endlessness can be extended from the configuration and principle of loops and infinite loops to never ending automated decisions. Culturally (and crucially) these decisions are recommended and enacted in search engines, posts, bookselling and many other e-commerce sites and daily transactions in contemporary culture which classify what should be visible or otherwise, relevant or irrelevant. As a consequence knowledge is shaped invisibly and distributed unevenly.

Indeed these automated decisions can be quite unpredictable and do not necessarily follow our wishes. Drawing upon the philosophy of Alfred N. Whitehead (1968, 1978) and a deep understanding of the Turing machine, Parisi and Fazi discuss the notion of completion in conjunction with unpredictability. The drive towards completion needs to account for “the exceptional condition of instability and malleability of the computational rule” (2014, p. 110). In this sense, running an algorithm does not guarantee it will reach an end even though there are well-defined rules and procedures as “completion in computation cannot always be attained” (Parisi & Fazi, 2014, p. 117). Within the context of computation they suggest that what has been overlooked is actualisation which involves more than code and memory, algorithms and automated decisions. For them what makes computation dynamic is the process of “an actual occasion” (Parisi & Fazi, 2014, p. 117). They explain:

An actual occasion is always a spatio-temporal relation between elements in process; it is indeed an occasion, its own eventuality determines its own prehensions, and, consequently, its own constitution (Parisi & Fazi, 2014, p. 117).

They suggest that the understanding of algorithms requires the need to re-conceptualise them as “forms of process” (Parisi & Fazi, 2014, p. 112); which



they refer to as actualities. Central to this, and following a Whiteheadian perspective, it is “a process of concrescence” that determines the actual occasion (Parisi & Fazi, 2014, p. 112). Data is highly involved in the process of the becoming of an actual occasion. It is not merely being reused or forwarded as the same thing but rather data is something that is “re-processed by the actual occasions under new conditions” (Parisi & Fazi, 2014, p. 114). Any sense of completion is both physical and conceptual, involving the actual processing of data and its structure and variation on the one hand and, on the other, entangling the incomputable condition that informs “the unknown condition of an algorithmic occasion” (Parisi & Fazi, 2014, p. 118). The re-conceptualisation of algorithms is regarded as dynamic in light of their notion of actual occasion which emphasises unknowable parts. Therefore, as they are keen to stress, a sense of an end includes the entanglement of the knowable and unknowable.

Through Turing’s proof of the halting problem, the hybridisation of known and unknown phenomena, as I argue, exhibits contradictory forces through running algorithms. Such algorithmic entanglement can be illustrated through bots which are not only coupled with the forces of the living and the dead, but also contradictory forces on the same plane of immanence that are manifested as unpredictable acts; examples may include the Microsoft Twitter chatbot, TayTweets,<sup>159</sup> that expressed racist language and the physical robot, Promobot,<sup>160</sup> that escaped from a testing lab and appeared in public. These are just two examples that demonstrate the actualisation of algorithms in which knowable, unknowable and contradictory forces are entangled with machine learning algorithms, causing unpredictable outcomes. What is important is that automated decisions are not just being made at the behest of the humans who preset them but that nonhuman

---

<sup>159</sup> TayTweets is a Twitter chatbot implemented by Microsoft as a research project in 2016. It caught much news attention because it took less than 24 hours for this chatbot to reply in racist tweets. See one of the news: <http://www.theverge.com/2016/3/24/11297050/tay-microsoft-chatbot-racist>

<sup>160</sup> Promobot, also called Promotional Robot, is a physical robot created by Russian scientists. It is designed for companies to interact with customers. In 2016, the robot unexpectedly escaped from a research lab in Russia and appeared on the road causing traffic chaos in the city centre. See: <http://www.dailymail.co.uk/sciencetech/article-3643119/Robot-run-Watch-bizarre-moment-self-learning-android-escapes-testing-area-causes-havoc-roads.html>

agents are taking an increasingly primary role in capturing and analysing data from which they can learn to adjust decisions over time. Consequently, these unpredictable decisions can be contradictory to human wishes and the code in itself.

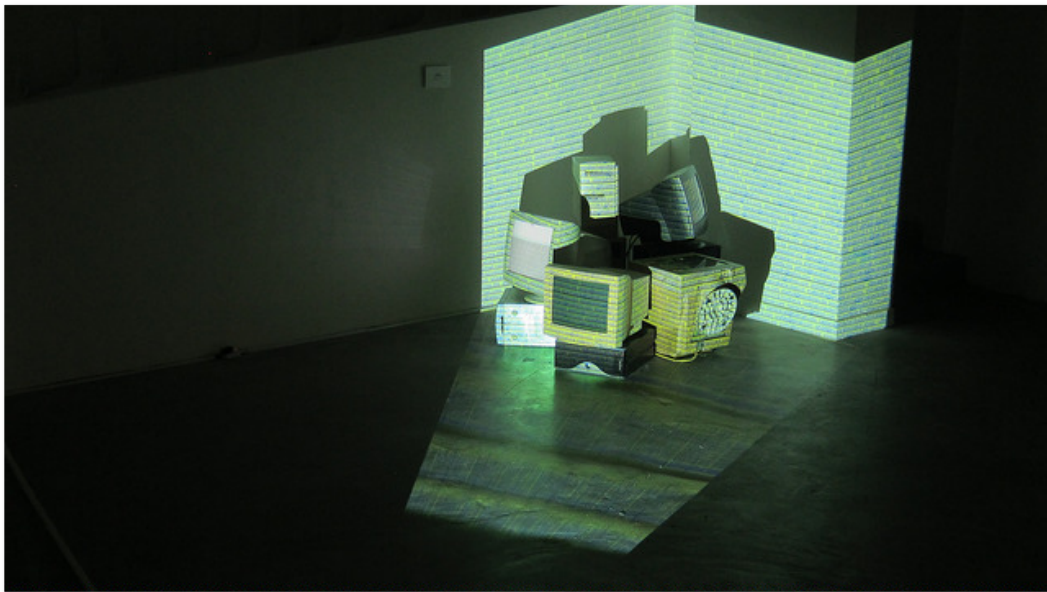
In this chapter the use of various nonhuman agents such as spam, update/upgrade notifications and bots, has been described in order to emphasise the tendency for the increasing use of automated strategies in software (art) practice. This automation is achieved in part by the activities of (undead) reading and writing in various repetitive acts. The focus of this chapter has been on the manner in which repetition unfolds complex computational logics beyond human perception and is central to the notion of automation. Citing the examples of 'loop,' 'open or die' and 'try and catch exceptions' this chapter has not only considered repetition in terms of differences, it has also clarified how algorithms can be considered to be both knowable and unknowable as part of an entanglement of complex forces.

My claim is that the artistic work *Hello Zombies* explores the assemblage of forces that help us to scrutinize the notion of liveness in computation. It also demonstrates how spam functions within an automatic and self-regulatory system. However, spam is just one of the many automated agents in contemporary culture. Other agents are active as well such as bots that are being automated by self-learning algorithms, continuously running and inter-acting with the networked and physical environment, constituting contradictory forces that require our attention as it produces unpredictable outcomes.

The notion of liveness is also *about* deadness. They are entangled and, when it comes to addressing the dynamics of liveness in computation, further reflection on deadness is inevitable. The notion of deadness considers disruption and interruption in computation at both macro and micro levels. These aspects have been previously discussed in Chapter 3, through the notion of the inexecutable query and in Chapter 4, with regard to micro-decisions in protocols and absent data in streams. In this chapter the

discussion has been extended to include ‘open or die’ syntax, exceptions and the sense of ending in algorithms. Chun’s notion of undeadness is particularly useful for this thesis as it emphasises the ever-changing conditions that arise from repetition in contemporary software culture (2011b, p. 138). Her attention to the process of execution and memory and what she calls the “undead of information” captures live conditions as something ephemeral in which the living and the dead are entangled. This chapter has used the notion of undeadness to describe the specific syntax and automation of code and extended the concept to include not only the coupling forces but also the contradiction of liveness and deadness at the material-level of computation through an understanding of algorithms at a historical, technical and cultural level.

### ***5.3 Hello Zombies***



*Figure 5.18: Hello Zombies (2014)*

I first approached *Hello Zombies* in February 2014. Without losing sight of the research in and through art practice, the project took nine months to realise and the first proposal was sent to Writing Machine Collective,<sup>161</sup> the exhibition organiser, on May 2, 2014. The proposal was finalised after a site visit to the venue in Hong Kong on September 16, 2014, a series of test arrangements and a discussion with the exhibition's research director, Hector Rodriguez. The context of the exhibition was that it especially focused on both research and practice. After the site visit, the final installation setup changed significantly as I discovered that when a projector was put on the floor and spammer addresses were projected on the wall there were some unexpected abstract visualisation effects produced on the floor that gave a sense of dynamism (see Figure 5.20). Attention to the materiality of the medium is one of the characteristic of artistic practice (Borgdorff, 2011, p. 49).



Figure 5.19: Testing out different sculptural forms at City University of Hong Kong in 2014

<sup>161</sup> This organisation is particularly focused on practice-based research, which is not just only on the artwork and its presentation.



Figure 5.20: Site visit in 2014

In relation to the central concept of automation in this chapter, I wanted to base the work on my previous project, *Readme.SpamPoem*,<sup>162</sup> to develop a generative artwork such that it became an autonomous piece, without the need for direct human interaction. The work can run and express by itself as it unfolds in time. Driven by my intuition and interest (Borgdorff, 2011, p. 55; Sullivan, 2010, p. 110), I identified spam as an interesting subject to help me developed my thinking about both generativity and automation in contemporary software culture.

I developed a blog<sup>163</sup> to document my thinking processes and reflections, references, technical references, experiments and so forth. This documentation, as I have discussed in my two previous artworks, is an important part of reflexive practice. There were a total of 58 posts from March 17, 2014 to April 25, 2016. Throughout this period, I explored different topics in relation to automation, such as procedurality, mutability, processuality and temporality. According to Borgdorff, artistic experiments, as opposed to scientific experimental systems, offer the ability “to continuously open new perspectives and unfold new realities” (2014, p. 117). This openness makes room “for not-knowing, or not-yet-

---

<sup>162</sup> The previous project was more focused on spam language and culture, as well as the autoreply feature in email system. See: <http://siusoon.net/home/?p=1184>

<sup>163</sup> See: <http://generativeaesthetics.blogspot.dk>

knowing” (Borgdorff, 2014, p. 114) that directs the thinking of the work *Hello Zombies* in keeping with the concept of thinking with objects and materials. The blog also refers to practical research materials on spam filtering logics, spam production and generative text display and setup. I also wrote about my experiments with different technologies to understand how *Hello Zombies* can be implemented such as the programming languages JQuery, PHP, Python as well as different operating systems including Raspberry Pi and Mac OS.

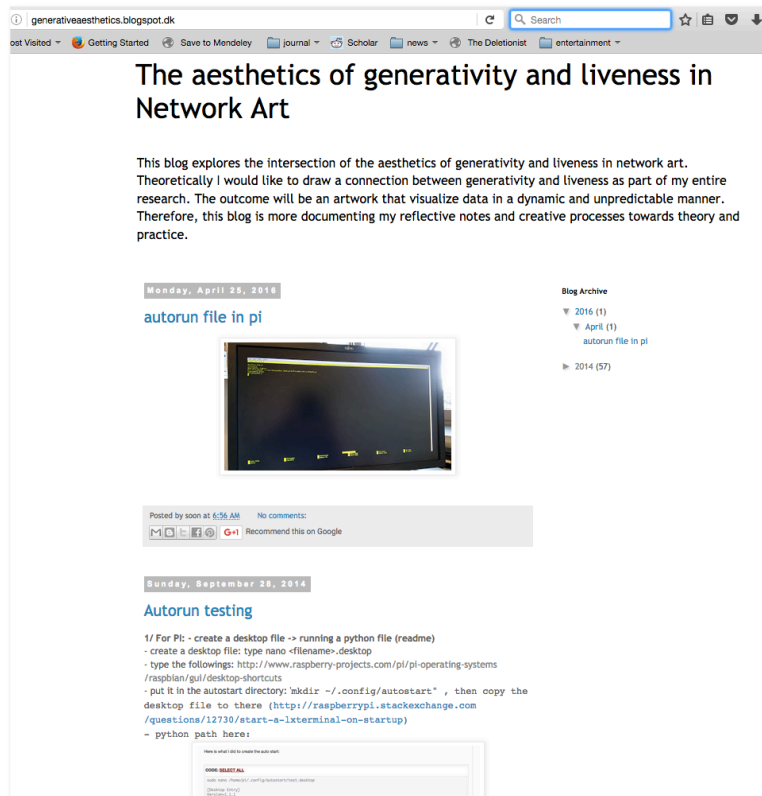


Figure 5.21: A blog was setup to document my own reflections. Retrieved from: <http://generativeaesthetics.blogspot.dk/>

The details of how the programs work together have been already described in the earlier section of this chapter with the flow chart. An earlier version of the flow chart was created on August 25, 2014 and the high level logics were confirmed around the same time (see Figures 5.22-5.23). These system logics indicate that coding practice does not only include source code or the outcome of the code execution, but other processes and materials are also regarded as being part of the work (see the discussion of Burnham’s curated exhibition in Chapter 2, section 2.1).

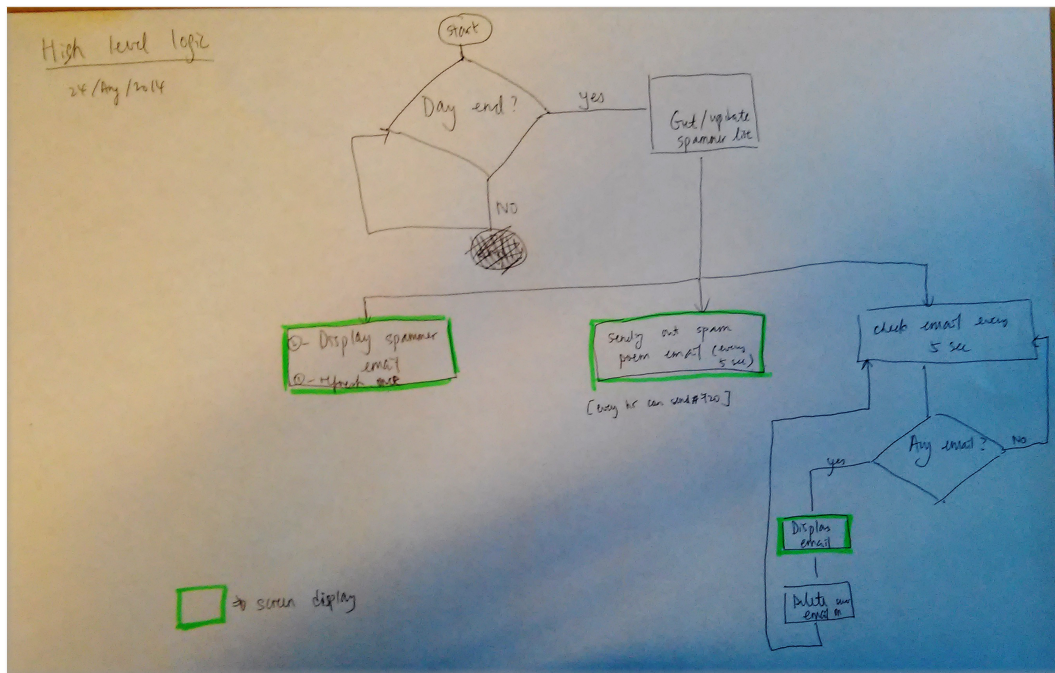


Figure 5.22: A high level draft of the flow chart

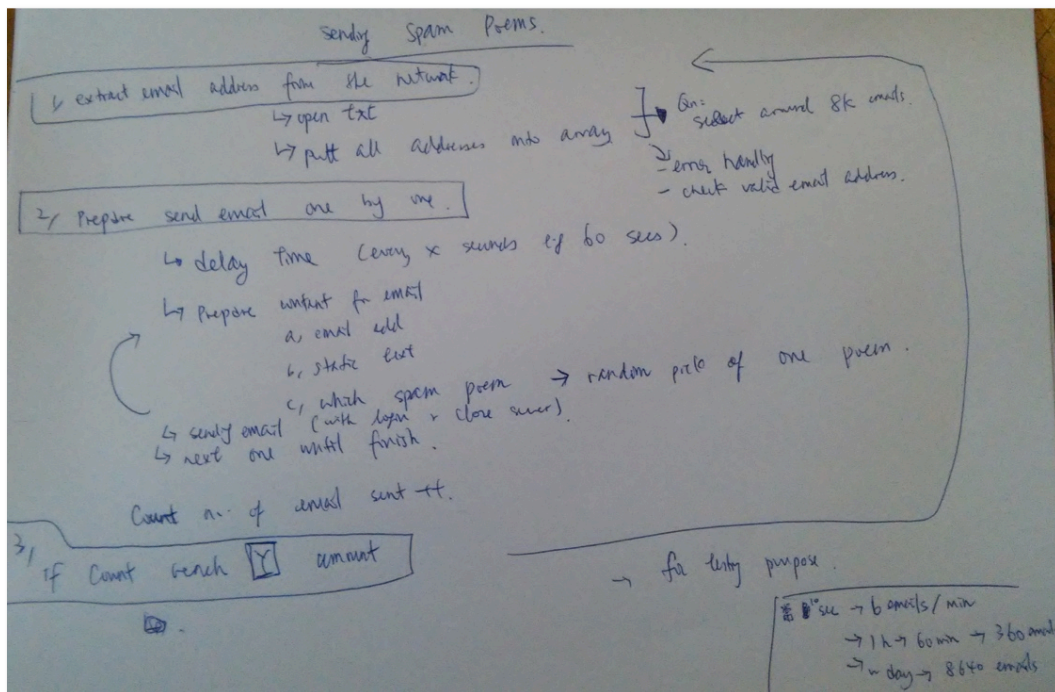


Figure 5.23: High level logic of the programs

As I was about to submit the final artist statement relating to the work, I was given the opportunity to participate in the Transmediale research workshop<sup>164</sup> in Hong Kong with

<sup>164</sup> See: <https://transmediale.de/content/call-for-participation-phd-research-workshop-datafied-research>



the theme of *Datafied Research*. This was an excellent chance to reflect on what I had been doing, contextualising the artwork further in a written format. I first wrote a 2000 text about spam and the concept behind the project a month before the exhibition (later this was submitted and published in the Tracing Data proceedings along with contributions from the other participating artists). The text was then reduced to 1000 words for the Transmediale newspaper which was published during the exhibition. Later the text was expanded to around 4000 words for a peer-reviewed online journal which was published after the exhibition. I was also given the opportunity to talk about this project at Transmediale 2015 which took place in Berlin with the theme of *Capture All*. All the text that I submitted explored spam as a datafied phenomenon in contemporary software culture. In particular with this work, publication/research and artistic practice are highly intertwined, and they play a constitutive role in the process of continuous discovery in which practice and theory inform each other reflexively (Sullivan, 2010). Through participating in the workshop, I also developed the artist statement that went alongside the artwork during the exhibition. The artist statement is important in media artwork and for me it is part of the artwork that expressed the idea behind and somehow illustrated the context and logics. Here is the exhibition version of the artist statement:

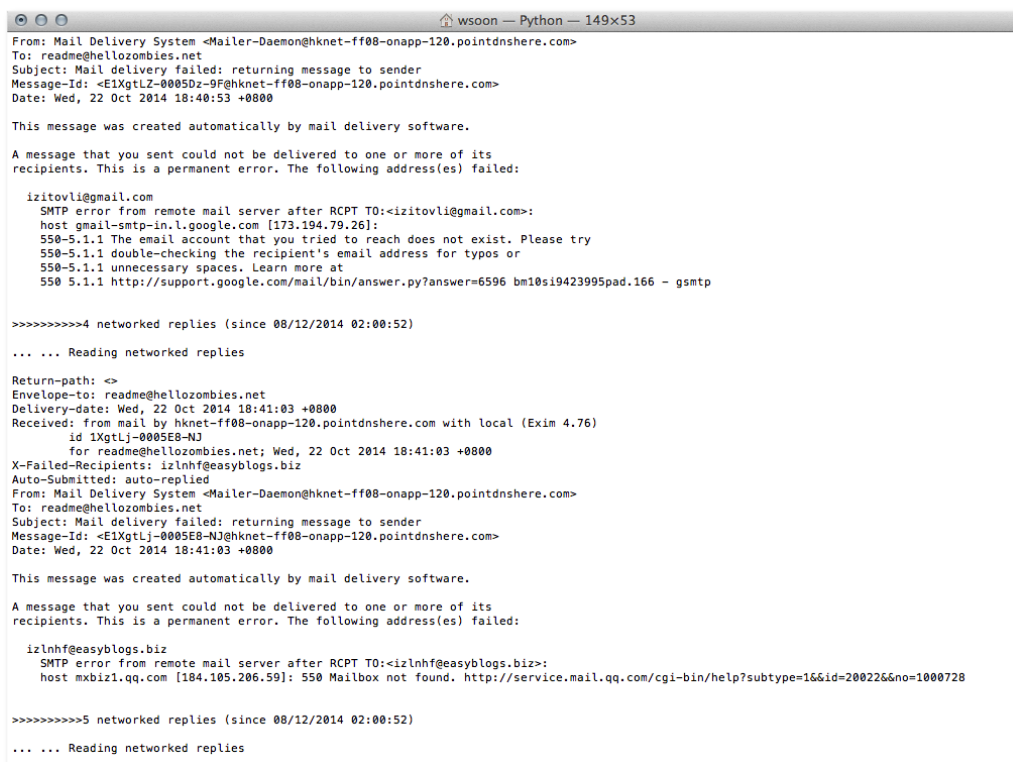
We are with you everyday, we live in the Internet with peculiar addresses and enticing titbits, but you call us “spam”. We wander around the network, mindlessly, and you wanted to trash us, but we are still everywhere. We are just the children of your economic and social system, but you ignore and avoid us. We are not dead, we write, we create.”

—This artwork examines these nonhuman zombies as a cultural phenomenon that produces quantified data and network identities. Through running the automated living machines, the artwork intervenes the network by writing spam poems to zombies and reading networked replies continuously. This project explores zombies of the living dead that bring forward social, technical, capitalistic and aesthetic relations in everyday lives (Soon, 2014a).

Some of the concepts I explored in the Transmediale workshop such as quantification and the living dead, have now, in later versions, become part of the artwork. Expanding the text allowed me to refine my concept and expand my argument. One of the most important concepts was that of the living dead which was further developed in this

chapter and throughout this thesis. Therefore, such reflexivity of theory and practice are not in opposition, but rather are considered as entanglement that co-produce knowledge (Barad, 2007).

In the exhibition setting, all of the screen' outputs, especially the one sending out poems and checking the server's emails, were designed to demonstrate the operative data processing. Figures 5.24 and Figure 5.25 show a screen that is packed with text, where the received emails were displayed consecutively and the screen scrolled down automatically in the exhibition setting. The program was also setup to experiment with any replies from the spammer and to think about the loophole as well as the concept of waste within an email system. In other words, artistic practice can act as a means or a mode of inquiry, to reach out for things that are unknown or are not yet known. According to Borgdorff, "[t]he openness of art is what invites us, again and again, to see things differently" (2014, p. 118). This is evident in the recent writing and reflection about this work by digital poetry scholar David Jhave Johnston, he invites readers to think about the notion of surplus that embedded in sending and handling emails in the work of *Hello Zombies* (2016, p. 190).



```
wsoon — Python — 149x53
From: Mail Delivery System <Mailer-Daemon@hknet-ff08-onapp-120.pointdnshere.com>
To: readme@hellozombies.net
Subject: Mail delivery failed: returning message to sender
Message-Id: <E1XgtLZ-0005Dz-9F@hknet-ff08-onapp-120.pointdnshere.com>
Date: Wed, 22 Oct 2014 18:40:53 +0800

This message was created automatically by mail delivery software.

A message that you sent could not be delivered to one or more of its
recipients. This is a permanent error. The following address(es) failed:

  izitovli@gmail.com
SMTP error from remote mail server after RCPT TO:<izitovli@gmail.com>:
host gmail-smtp-in.l.google.com [173.194.79.26]:
550-5.1.1 The email account that you tried to reach does not exist. Please try
550-5.1.1 double-checking the recipient's email address for typos or
550-5.1.1 unnecessary spaces. Learn more at
550 5.1.1 http://support.google.com/mail/bin/answer.py?answer=6596 bm10s19423995pad.166 - gsmtp

>>>>>>>>4 networked replies (since 08/12/2014 02:00:52)
... .. Reading networked replies

Return-path: <>
Envelope-to: readme@hellozombies.net
Delivery-date: Wed, 22 Oct 2014 18:41:03 +0800
Received: from mail by hknet-ff08-onapp-120.pointdnshere.com with local (Exim 4.76)
 id 1Xgtlj-0005E8-NJ
 for readme@hellozombies.net; Wed, 22 Oct 2014 18:41:03 +0800
X-Failed-Recipients: izlnhf@easyblogs.biz
Auto-Submitted: auto-replied
From: Mail Delivery System <Mailer-Daemon@hknet-ff08-onapp-120.pointdnshere.com>
To: readme@hellozombies.net
Subject: Mail delivery failed: returning message to sender
Message-Id: <E1Xgtlj-0005E8-NJ@hknet-ff08-onapp-120.pointdnshere.com>
Date: Wed, 22 Oct 2014 18:41:03 +0800

This message was created automatically by mail delivery software.

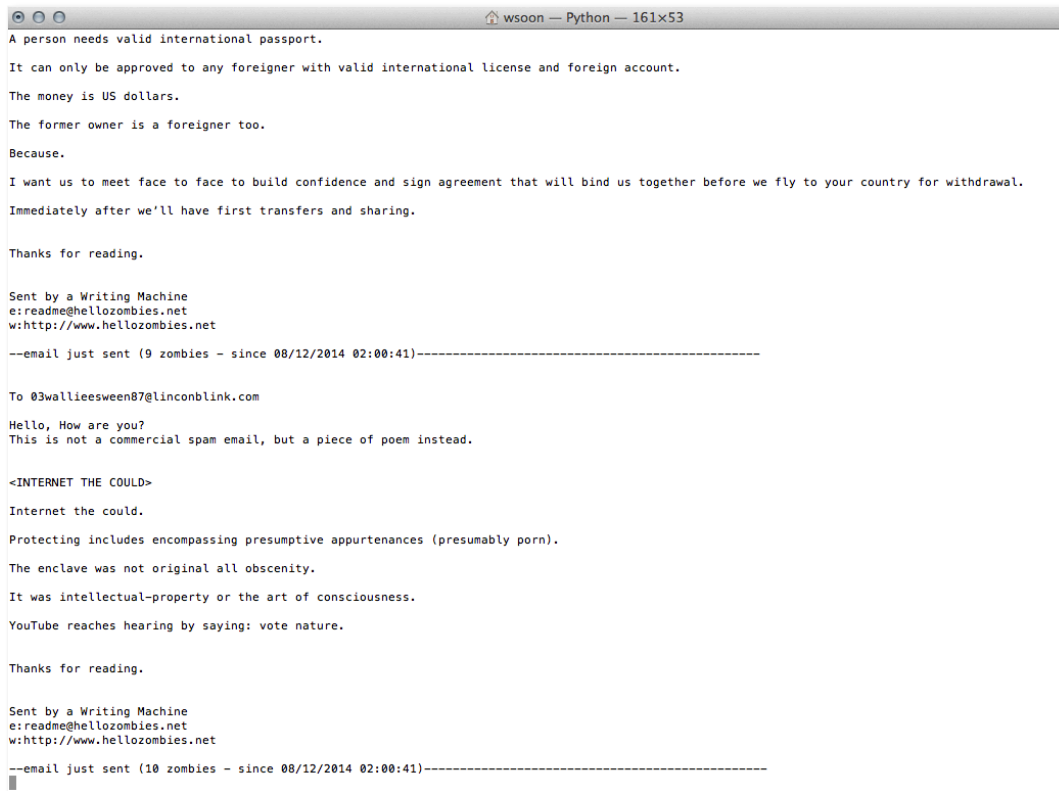
A message that you sent could not be delivered to one or more of its
recipients. This is a permanent error. The following address(es) failed:

  izlnhf@easyblogs.biz
SMTP error from remote mail server after RCPT TO:<izlnhf@easyblogs.biz>:
host mxbiz1.qq.com [184.185.206.59]: 550 Mailbox not found. http://service.mail.qq.com/cgi-bin/help?subtype=1&&id=200226&no=1000728

>>>>>>>>5 networked replies (since 08/12/2014 02:00:52)
... .. Reading networked replies
```

Figure 5.24: Reading network replies in *Hello Zombies*

## Executing Automated Tasks



```
wsoon — Python — 161x53
A person needs valid international passport.
It can only be approved to any foreigner with valid international license and foreign account.
The money is US dollars.
The former owner is a foreigner too.
Because.
I want us to meet face to face to build confidence and sign agreement that will bind us together before we fly to your country for withdrawal.
Immediately after we'll have first transfers and sharing.
Thanks for reading.
Sent by a Writing Machine
e:readme@hellozombies.net
w:http://www.hellozombies.net
-----
--email just sent (9 zombies - since 08/12/2014 02:00:41)-----
To 03wallieesween87@linconblink.com
Hello, How are you?
This is not a commercial spam email, but a piece of poem instead.
<INTERNET THE COULD>
Internet the could.
Protecting includes encompassing presumptive appurtenances (presumably porn).
The enclave was not original all obscenity.
It was intellectual-property or the art of consciousness.
YouTube reaches hearing by saying: vote nature.
Thanks for reading.
Sent by a Writing Machine
e:readme@hellozombies.net
w:http://www.hellozombies.net
-----
--email just sent (10 zombies - since 08/12/2014 02:00:41)-----
```

Figure 5.25: Sending poems in *Hello Zombies*

In addition to the email sending and receiving, through paying special attention to the display of the spam email addresses, the work highlights the network identities of spam and their unique characteristics on the web which allow them to be replied to (in the physical installation, the audience could not click anything as there was no keyboard available). The final implementation of the artwork used old-fashioned HTML rolling text that utilised the 'MARQUEE' syntax and set the hyperlinks of each of the address with the corresponding email. As a result the screen outputs a densely packed field of hyperlinks rolled over time. This implementation led me to think about the meaning of spammer addresses and their characteristics which were also discussed in the earlier section of this chapter.



Figure 5.26: Rolling Spammer addresses in Hello Zombies

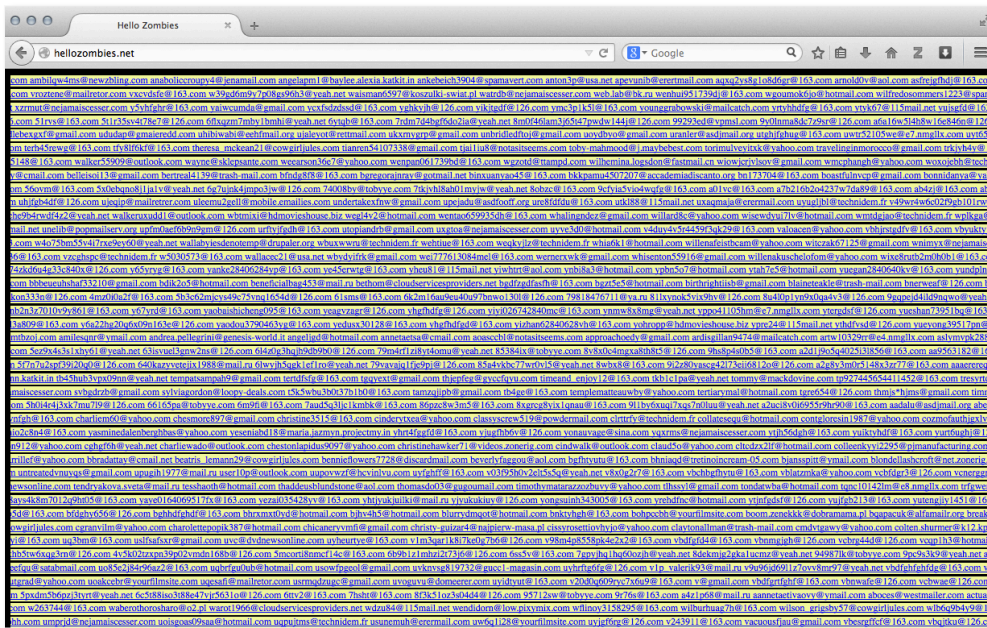


Figure 5.27: Densely packed spammer addresses in Hello Zombies

```

77 ##### display rolling text #####
78 for ($y = 0; $y < sizeof($list); $y++)
79 {
80 # >>>add the random parameter of direction and scrollamount
81 $roll=array("left","right");
82 echo "<MARQUEE behavior=\"scroll\" direction=\"\"";
83 if (strlen($list[$y]) % 2) { #DEFINE the rolling direction by the no of characters in each row
84 echo $roll[0];
85 }
86 else{
87 echo $roll[1];
88 }
89 #echo $roll[rand(0,1)]; #DEFINE the rolling direction
90 echo "\" scrollamount=\"";
91 $scroll= (strlen($list[$y]) % 5); #DEFINE the rolling speed by the no of characters in each row
92 if ($scroll < 1) {
93 echo "1";
94 }else{
95 echo $scroll;
96 }
97 #echo rand(1,8); #DEFINE rolling speed
98 echo "\" width=\"100%\" bgcolor=\"#F2F5A9\">"; #####
99
100 $email_item = explode(",", $list[$y]);
101 for ($z = 0; $z < sizeof($email_item); $z++)
102 { #mailto:isusoon@gmail.com
103 echo "<a href=\"mailto:\", $email_item[$z],\">\", $email_item[$z],\"</a>";
104 }
105 echo "</MARQUEE>";
106 echo "</p>";
107 }
108 ##### end of display rolling text #####
109 ?>

```

Figure 5.28: The excerpt of the source code on presenting email addresses on a screen

The work itself went through different iterations. The errors that I encountered during running the program prompted me to think further about methods for avoiding errors during run-time and live exhibition. Informed by this, this chapter further develops the section called ‘Try and catch exceptions.’ As discussed in Chapter 2, reflexive coding practice involves a loosely configured experimental system that embraces instability and indeterminacy (Borgdorff, 2014, p. 115). Experimental systems in artistic practice are not to validate hypotheses or seek absolute facts or findings, but rather they are the reflexive intertwining of technical objects and epistemic things. Attention to material agency, such as crashes, faulty code, incompatibilities and in this case errors, is always of interest to artists in software (art) practice (Cramer, 2003, n.p). This also aligns with Berry’s method on iterative trials in which errors may be seen as a form of understanding and knowing machine operational processes (Berry, 2014, p. 186) (see Chapter 2, section 2.4.2).

```

-----email is sent at 21/09/2014 07:22:13 (total no of zombies: 5193)-----
-----
To dessiejernigan@mailcatch.com

Hello, How are you?
This is not a commercial spam email, but a piece of poem instead.

<THE SAME SAME NAME>

Hello Dear, I am Zain Abina, legal practitioner.

I contact you to give you money left by my late client, whom you share the same last name with.

My late client (who shall be referred to as my client) has ($19 MILLION) lodged in the bank.

Client died as a result of a heart on 30 Sept.

His heart went due to the death of all the members of his family in the 2004 tsunamis natural disaste
r while holidaying in Phuket Thailand.

Contact me for more.

Thanks for reading the poem.

Sent by an artist-researcher, Winnie Soon
e:readme@hellozombies.net
w:http://www.hellozombies.net

-----email is sent at 21/09/2014 07:22:17 (total no of zombies: 5194)-----
-----
[Errno 61] Connection refused
Traceback (most recent call last):
  File "/Users/wsoon/Desktop/artwork/hellozombies/python_test/sendemail2.py", line 127, in <module>
    main()
  File "/Users/wsoon/Desktop/artwork/hellozombies/python_test/sendemail2.py", line 36, in main
    getemail_function()
  File "/Users/wsoon/Desktop/artwork/hellozombies/python_test/sendemail2.py", line 115, in getemail_f
unction
    sendemail(data[index]) #call send email function
  File "/Users/wsoon/Desktop/artwork/hellozombies/python_test/sendemail2.py", line 87, in sendemail
    server.quit()
UnboundLocalError: local variable 'server' referenced before assignment
d04227-2:~ wsoon$ python /Users/wsoon/Desktop/artwork/hellozombies/python_test/sendemail2.py

To *****@gmail.com

```

Figure 5.29: The highlight of a connection error in running the programs of *Hello Zombies*

```

Sent by an artist-researcher, Winnie Soon
e:readme@hellozombies.net
w:http://www.hellozombies.net

-----email is sent at 19/09/2014 03:29:29 (total no of zombies: 768)-----
---
Traceback (most recent call last):
  File "/Users/wsoon/Desktop/artwork/hellozombies/python_test/sendemail2.py", line 103, in <module>
    main()
  File "/Users/wsoon/Desktop/artwork/hellozombies/python_test/sendemail2.py", line 32, in main
    getemail_function()
  File "/Users/wsoon/Desktop/artwork/hellozombies/python_test/sendemail2.py", line 91, in getemail_function
    sendemail(data[index]) #call send email function
  File "/Users/wsoon/Desktop/artwork/hellozombies/python_test/sendemail2.py", line 52, in sendemail
    server = smtplib.SMTP(HOST)
  File "/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/smtplib.py", line 250, in __init__
    (code, msg) = self.connect(host, port)
  File "/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/smtplib.py", line 310, in connect
    self.sock = self._get_socket(host, port, self.timeout)
  File "/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/smtplib.py", line 285, in _get_socket
    return socket.create_connection((host, port), timeout)
  File "/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/socket.py", line 571, in create_connection
    raise err
socket.error: [Errno 50] Network is down
d04227-2:~ wsoon$

```

Figure 5.30: The highlight of a network error in running the test programs of *Hello Zombies*

Since the programs were coded to handle many repetitive tasks, exploring the use of a loop was necessary in so far as it was able to use the same logic but to process different data. This also prompted me to seek a better understanding of the relationship between the syntaxes used in *Hello Zombies* and the concept of automation. The three syntaxes, which I explained in the previous sections, namely, loops, open or die, try and catch exceptions, are informed by coding practice and an understanding of how these functions are built and used and any corresponding implications. In addition to this, the work allowed me to speculate on the logic of spam and the potential application of automated systems that matter to us.

Finally, the remaining part of this section will show key excerpts of the three programs in *Hello Zombies* that constitute an automated system (the full source code is also available on the USB storage device submitted with this written thesis). In short, this section demonstrates that the art practice of *Hello Zombies* offers epistemic enquiry through reflexive practice. The live running work reflexively and continuously invites “unfinished thinking” (Borgdorff, 2011, p. 44) via executing automated tasks.

```

~/Desktop/Desktop/data/PhD/PhD_writing/Thesis/usb/HelloZombies/latestsourcecode/hz_sendemail.py (no functi
24 import string
25 import urllib2
26 import time
27 from time import gmtime, strftime
28 import random
29 import sys
30 import os
31 import logging
32 import socket
33 global sendemail
34 global poem_list
35
36 #first default function as main
37 def main():
38
39     getepoem_function()
40     getemail_function()
41
42 #global send email function and catch exception
43 def sendemail( eaddr ):
44     global runtime
45     global count
46     HOST = "mail.hellozombies.net:587"
47     FROM = "readme@hellozombies.net"
48     TO = eaddr
49     poem = getpoem() #title and content
50     text = "\nHello, How are you?\nThis is not a commercial spam email, but a poem instead. \n\n\n<:
51     final_body = MIMEText(text,'plain')
52     final_body['From']=FROM # some SMTP servers will do this automatically, not all
53     final_body['MIME-Version']="1.0"
54     final_body['Subject']="Hello"
55     final_body['Content-Type'] = "text/plain; charset=utf-8" #or text/html
56     final_body['Content-Transfer-Encoding'] = "quoted-printable"
57     socket.setdefaulttimeout(10)
58
59     try: # sometimes connection fails
60
61         server = smtplib.SMTP(HOST)
62         server.login('INPUT USERNAME', 'INPUT PASSWORD')
63         server.sendmail(FROM, TO, final_body.as_string())
64         send_time = strftime("%d/%m/%Y %H:%M:%S", time.localtime())
65         server.quit()
66         display = "\n\nTo %s \n%s--email just sent (%d zombies - since %s)-----
67         print (display) #display on screen
68         count += 1
69         time.sleep(3)

```

Figure 5.31: An excerpt of the source code on sending poems







## 6

# Unfinished Thesis

This final chapter operates somewhat like a conclusion for the entire thesis but declares itself unfinished, inviting readers to engage in unfinished reflection and endless process of material relations. It is inspired by Borgdorff's "unfinished thinking" in, through and with reflexive art practice (2011, 2014). That there is no ending as such and should also be considered in the light of the "strange" and "infinite" loops that I discussed in the previous chapter (Ernst, 2009, n.p; Hofstadter, 1980 [1979], p. 157). This ongoing updating of the thesis and running of things is perfectly in keeping with the logic of dynamic computational culture. Given that things are constantly changing, updating and emerging, I am presented with the problem of how to mark the end of this thesis which would otherwise operate in contradiction to concepts such as unfinished thinking and endlessness (Ernst, 2009, n.p). Perhaps this contradictory force can be seen as an expanded halting problem for my research process as a whole. This thesis, like any other thesis, is in progress—it will "loop forever" (Ernst, 2009, n.p). Informed by how code is written which executes in a dynamic manner, this written text may be considered to run within a loop that engages its readers differently. The entanglement of material processes and the surrounded critical discourse is always in a state of becoming, always unfinished and always in progress. The thesis invites the reader to engage in unfinished reflection that simply cannot reach a definitive end.

With a distinct focus on the live dimension of code inter-actions, a materialist framework has been developed, demonstrating how it might be used to analyse, address and respond to the ubiquitous, complex, and computational nature of *liveness*. These live phenomena include the increasing execution of data queries, instantaneous transmission of streams and seamless running of automated agents and they are symptomatic of this

datafied, accelerated and ever-updating culture under contemporary conditions.

This thesis has been developed through a materialist examination of live phenomena by taking into account real-time programmable technologies in networked environments. I have used a range of methods which brought together diverse fields including ‘iterative trials’ from software studies, ‘close reading’ from critical code studies, ‘cold gazing’ from media archaeology and ‘reflexive practice’ from artistic research to foreground what I have referred to as ‘reflexive coding practice.’ Central to this methodology is close attention to reading, writing, running and executing code which allowed a deep reflection upon and understanding of how code operates and inter-acts with things beyond mediated representation. By using the method of reflexive practice, I have demonstrated how theory and practice inform each other through my three artistic and experimental projects. The practice of coding can be seen as an effective way of understanding how things work at the epistemic level, in which reflexivity informs thinking about the world. Epistemic thinking also plays a significant role in shaping meaning and producing knowledge. The projects were discussed in Chapters 3-5 alongside an analysis of relevant phenomena. At the end of each of these chapters, I have articulated a more subjective register to account for the conceptual background and coding practice in order demonstrate that the emergence of knowledge production and meaning making are co-produced by the reflexive entanglement of practice and theory.

Within the discussion of software (art) studies in this thesis, I first offered a detailed overview of the field in Chapter 2 and this led to present three main concepts—namely invisibility, performativity and generativity—and these served to both unfold some of the theoretical debates in the field of software studies as well as to support further discussion of the concept of liveness. Through the notion of code inter-actions, live phenomena have been examined in which attention has not only been paid to their technical, operational and functional attributes but also to consider their cultural

implications through analysing the assemblages of force that constitute the phenomena of liveness. This thesis has undertaken a close examination of material substrates, code operations and technical infrastructure beyond how they are generally made visible and perceptible. These relatively invisible code inter-actions affect how we perceive computational processes as live events and constitute to the overall notion of liveness that has been discussed throughout the whole thesis. More importantly the opaqueness of computational operations is intertwined with wider economic, political and cultural forces that require our attention. The live phenomena, references from Barad, are contingently materialised and configured through a process of entanglement (2007, p. 140).

## **6.1 Contribution**

The main objective of this thesis is to develop a more nuanced understanding of liveness in the field of software studies. Building upon previous understandings of liveness derived from performance, software and media studies (as demonstrated in Chapters 1 and 2), a materialist framework was presented through three vectors—unpredictability, temporality and automation—which together illuminated the discussion of liveness. They responded to the central question of this thesis, chapter by chapter: How does a materialist framework of liveness reconfigure our understanding of software and expand the discussion of what constitutes liveness? Undertaking live queries as an object of study, Chapter 3 explored the inter-actions between code, databases and technological networks and argued that live queries are active participants that exhibit performativity through their data structures, formats, execution and cultural practices. The chapter analysed the unpredictability of live queries by examining the deep structural level of communication channel transmission, data compression and compilation. I argued that randomness is inscribed at the deepest level of computation in infinite binary strings, data processing and querying which generate unpredictable variability. The chapter also identified the performative mathematical operators as a site of restriction and control. They are able to act—identify, exclude, specify and sort data—in different

ways and hence directly impact what data are processed. In contrast to the notion of openness that is related to Twitter as a participatory platform, the idea of the inexecutable query addressed the closedness of Twitter's APIs that limit the participation in its development and understand the logics of data processing. The concept of inexecutability thereby should be understood in relation to the notion of closedness which is beyond technical errors and incompatibility. The constituent forces include market, social and political forces where live queries are operated at high levels of unpredictability, uncontrollability and unknowability in contemporary software culture. This chapter, together with the project *Thousand Questions*, served to demonstrate how code inter-acts unpredictably at multiple scales through executing queries. It argued that material forces constitute the unpredictable qualities of liveness.

Chapter 4 addressed the perceived gap, already identified through the textual analysis of liveness in Chapter 1, to offer a detailed analysis of the micro-processes and micro-temporalities behind the running of the abstract symbolic form of the throbber. Influenced by Ernst's notion of micro-temporality (2013b, pp. 186-9), this analysis drew upon a techno-engineering perspective (cold gazing) to examine digital signal processing, the fetch-execute cycle, the clock cycle, packet switching mechanisms, network handshaking processes, Sliding Window Protocol mechanisms, data buffering and dropped frames to discuss the detailed processes of computation behind a running throbber, paying attention to the time-dependent logic and the micro-temporality of code inter-actions. Drawing upon Sprenger's notion of 'micro-decisions,' the chapter further explored the interruption of deep operative processing which is beyond the linear and continuous flow of streams. Within this analysis the idea of discontinuous micro-temporality was established to rethink the metaphors of the flow and stream in networked environments. A stream is perceived as a continuous flow with unforeseeable and imperceptible interruptions. The notion of discontinuous micro-temporality takes into account the micro-processes, gaps and ruptures and, more importantly, the absence of data that renders realities in which multiple layers are at work while loading a live stream or

fetching a live feed. Together with the project *The Spinning Wheel of Life*, this chapter served to highlight the paradox and tension between continuity and discontinuity, between end and endless states as well as presence and absence to understand how streams are processed and organised computationally, and how they exhibit micro-temporality that leads to real-time rendering of a pervasive and networked condition of liveness.

Chapter 5 explored the notion of liveness through the vector of automation. Focusing on the act of repetition in automation, this chapter presented spam as one of the automated agents in which automated systems enable real-time computation and querying of data without human intervention. It recognised that the act of repetition does not simply automate tasks but also includes the process of generating differences and coping with instability and contradiction. Building upon previous understandings of inexecutable queries and the micro-interruption of streams, this chapter further drew on Chun's notion of undeadness to examine the sense of endlessness in computation, highlighting the ephemeral nature of code inter-actions. Through the analysis and articulation of the code syntaxes—'loop,' 'open or die,' 'try and catch exceptions'—it argued that the entanglement of living and dead forces are central to the understanding of liveness. More fundamentally, by drawing upon Alan Turing's halting problem (1937, p. 247), these entangled forces were seen to be in part contradictory in regards to how an algorithm is written and performed.

The implication of the problem of ending is that there are things that are impossible to solve and cannot be decided, but what is relevant in this particular chapter and to this thesis' overall argument is undecidability which implies the contradictory relation of liveness and deadness at the material level—code and algorithms—of computation. In this way, not only does the endless running of algorithms signify a coupling of living and the dead, but it also draws attention to the contradiction itself, conceptually as well as computationally. The artistic work *Hello Zombies* explored this assemblage of forces and extended from spam to other automated agents as a means of scrutinizing the notion of liveness in computation, in which

repetition can be endless (looping forever) and exhibit instability, exceptions, contradictions and without a finished or completed state, like this thesis itself.

Although the focus of this thesis has been the notion of liveness, both deadness and undeadness became useful counterparts as this thesis developed as a means of capturing the inherent unstable and dynamic nature of contemporary technology. Although it is beyond the scope of this thesis to engage in detailed discussion of phenomenology, the coupling of life and death, liveness and deadness (and undeadness) seems intrinsic, with deadness being the inexecutable, absence, or end, of liveness. While liveness and deadness may commonly be viewed as mutually exclusive dualism, undeadness suggests a more complex mattering and entanglement between these states that can be examined through code inter-actions.

This research has contributed primarily to a widening of the focus of critical attention in software (art) studies through a close analysis of data queries, data streams and automated agents. It does so through an examination of a distinctive focus of the live dimension of code inter-actions, presenting the vectors of unpredictability, micro-temporality and automation. This thesis has developed what I call “reflexive coding practice” to examine these live phenomena and it is an applied approach to computational processes and a means by which to reflect on cultural issues through experimentation and practice. Furthermore, the thesis expands the debate in media and performance studies, providing technical description and analysis in relation to the concept of liveness. In overall terms, the research contributes to our understanding of software by expanding our understanding of liveness in contemporary culture. This includes a nuanced examination of liveness beyond immediate human reception.

Indeed the three vectors addressed in the thesis are not considered to be the definitive or fixed parameters for examining liveness, instead this study remains necessarily unfinished and the vectors are simply offered as one way amongst many to present an overall argument at this point in time.

From the coupling and entanglement of living and dead forces to assemblages of material forces, the overall argument of the thesis has served to assert that liveness can be examined through code inter-actions, in which the continuous process of executing and running of code inter-acts across various computational layers at multiple scales. Executing liveness is conditioned by such an assemblage of forces, which are not a static arrangement of things but a relationship between collective inter-actions in which things come to live as “things-in-phenomena” (Barad, 2007, p. 140). Beyond analysing a particular type of web query, a specific throbber icon and a peculiar spam agent, the examination of liveness and deadness (or even undeadness) has implications on wider cultural phenomena such as the update culture in various types of query, software and platforms, immediate streams and feeds as well as for the various automated agents that help us to understand some of the dynamics and complexity of contemporary software culture.

## 6.2 Future directions

In keeping with the spirit of unfinished thinking, I will end by suggesting a number of future research directions that the thesis’ overall argument can be expanded into. Informed by Turing’s halting problem (see previous chapter, section 5.2) this may mean that the thesis is contradictory in itself as it halts but at the same time offering future directions to continue the process as a loop.

First, the analysis of the notion of unpredictability in Chapter 2 only covered Western social media platforms. It is observed that the social messaging software WeChat in China has increasingly gained worldwide attention. This software connects payment transactions and services across cities which make it a powerful and efficient tool that is used by more than 800 million active users per month,<sup>165</sup> making it more than double the size of Twitter users. However, weChat is a highly centralised platform like other

---

<sup>165</sup> See: <https://www.statista.com/statistics/255778/number-of-active-wechat-messenger-accounts/>



internet services in China with a sophisticated censoring system. This means that both bots and humans can delete users' messages and suspend users' accounts automatically as part of the daily regulatory process. All the services that weChat provides, including but not limited to its security system, censorship system, transaction and payment systems and all sorts of communication systems, are integrated into one mobile application, offering live updates, intense query execution, rigorous monitoring and immediate operations that work on the network layer yet they are distinctively connected, political and complex in its infrastructure. In view of this highly censored and connected system, an understanding of the unpredictable qualities of liveness can be more comprehensively examined by covering a wider internet sphere to address the extra layer of complexity. However, such an analysis may require fieldwork in China in order to examine, and reflect upon, the apps culture which is far beyond the limited timeframe of the development of this thesis.

Secondly, there scope for developing a deeper discussion of other forms of technology beyond distributed networks. In Chapter 4, the focus was mainly on the temporality of the internet as well as its distributed mechanism. However, other alternative and emergent technology are highly relevant, in particular to the Peer-to-Peer (P2P) network, such as blockchain technology, BitTorrent protocol and real-time communication protocols (such as WebRTC), also connect machines together but using an entirely different networking approach. Including additional networking typologies may offer an expanded understanding of temporality through a different register of other complex forms of data distribution. During the latter stages of this research journey I have discovered an open source P2P software, called Web Torrent,<sup>166</sup> which could be used in the future by applying reflexive coding practice to examine the sophisticated and complex forms of networked technology.

Thirdly, as mentioned in Chapter 5, the adaptive quality of machine

---

<sup>166</sup> See: <https://webtorrent.io/>

learning is now being widely implemented in financial services, the medical industry and many others sectors in which data can be iteratively processed and learnt by algorithms. Machine learning, a field that is at the intersection of computer science and statistics, is highly dependent on large amounts of data that produce probabilistic outcomes, focusing on predictability through computational and learning processes. Although the notion of unpredictability has already been discussed in Chapters 3 and 5, this thesis has mainly focused on the concept “unpredictable manifestation” (Wardrip-Fruin, 2011, p. 307) as well as unexpected categorisation through machine learning in spam filtering. Future research can address a statistical perspective on the predictability of machine learning so as to think through how predictability and unpredictability emerge that exhibits different forces of liveness. Machine learning as suggested by Mackenzie is a different programming practice, as he puts it, “Machine Learning still has to be programmed by someone, but is programmed differently” (2013, p. 395). In the future I would be interested in exploring statistical programming languages, such as R and Python, which involve using statistical methods to understand how predictive works are processed, created and inter-acted with differently at the level of code. By using deep-learning methods, in which code is run in real-time, networked data can constantly be fed back to the system as a feedback loop that can learn and adapt. This kind of adaptive behaviour contains highly complex statistical logics and algorithms that require a more in depth discussion and understanding on the deep level of machine learning through coding practice. Further research on the predictive dimension and machine learning logics may inform our future understanding of data-driven culture and a more comprehensive study of automation.

Throughout this thesis, while I have not focused on issues of gender and race per se, I have deliberately included queer projects and postcolonial perspectives where possible as part of the artwork selection and discussion. As a result of subjective experience, and a deepening political awareness throughout this research, I have registered the contributions made by women to the historical development of computer technologies and coding

practices. Many of these have made a profound impact on technological development but are still too rarely recognised. As part of this, for instance, I mention Lovelace's fundamental loop concept and Hopper's contribution to the invention of the compiler. However I recognise that I might have developed this discussion in greater detail but again I consider this to be part of a larger and ongoing process of returning to, and reflecting upon, materials and materialisms, not least through more attention to the feminist new materialism of Barad. Promoting gender and race equality and diversity is especially important in STEM-related fields and I regard myself having a responsibility to cultivate a more open environment when disseminating knowledge in the field of software (art) studies both in my research and teaching practice.

Together with the three artistic projects presented herein, this thesis does not add up to an end in itself, or indeed a conclusion as such but rather should be considered as part of an ongoing and unfinished process. Like the argument that runs through it, running text and code to execute ongoing arguments and statements and calling for a critical awareness of code interactions simply cannot end as this is a technical and conceptual impossibility. This is not only meant to loop forever (to borrow Ernst's formulation) but also in consideration of code and text as a form of "undead writing" (Chun, 2008a, p. 149), preventing the loss of critical attention to computational processes. It is a reminder that even though text and code are repeatedly run, they never produce identical results but emerge through unfinished and undead processes that together execute forms of liveness.

# Bibliography

- 0100101110101101.ORG, & epidemiC. (2004). Contagious Paranoia: 0100101110101101.ORG spreads a new computer virus. Retrieved from [http://www.digitalcraft.org/iloveyou/biennale\\_part\\_2.htm](http://www.digitalcraft.org/iloveyou/biennale_part_2.htm)
- Abhyankar, A. S., & Schuckers, S. C. (2004). *A wavelet-based approach to detecting liveness in fingerprint scanners*. Proc. SPIE 5404, Biometric Technology for Human Identification. Retrieved from <http://dx.doi.org/10.1117/12.542939>
- Agamben, G. (1998). *Homo sacer : sovereign power and bare life*. Stanford, Calif.: Stanford University Press.
- AI-Rfou, R., Jannen, W., & Patwardhan, N. (2012). TrackMeNot-so-good-after-all. *ArXiv*.
- Albers, M. C. (1996). Auditory cues for browsing, surfing, and navigating. *Proceedings of the 3rd International Conference on Auditory Display (ICAD 1996), Palo Alto, California*.
- Amyatwired. (2011). Thousand of APIs Paint a Bright Future for the Web. *WIRED*. Retrieved from <https://www.wired.com/2011/03/thousand-of-apis-paint-a-bright-future-for-the-web/>
- Andersen, C. U., & Pold, S. (2004a). Introduction. In O. Goriunova & A. Shulgin (Eds.), *Read\_me : Software Art & Cultures*. Århus: Digital Aesthetics Research Centre : University of Aarhus.
- Andersen, C. U., & Pold, S. (2004b). Software Art and Cultures - People Doing Strange Things with Software. In O. Goriunova & A. Shulgin (Eds.), *Read\_me : Software Art & Cultures* (pp. 394 sider). Århus: Digital Aesthetics Research Centre : University of Aarhus.
- Andersen, C. U., & Pold, S. (2011). *Interface Criticism: Aesthetics Beyond Buttons*. Aarhus: Aarhus University Press.
- Anil, B. C., D, J., & Chayadevi, M. L. (2015). A Survey onf WIFI and LIFI technologies. *International Journal of Computer Technology and Applications*, 6(6), 1047-1051.
- Arns, I. (2004). READ\_ME, RUN\_ME, EXECUTE\_ME: Software and its discontents, or: It's the performativity of code, stupid. In O. Goriunova & A. Shulgin (Eds.), *Read\_me : Software Art & Cultures*. Århus: Digital Aesthetics Resarch Centre : University of Aarhus.
- Ascott, R. (1966). Behaviorist Art and the Cybernetic Vision, Part One. *Cybernetica: journal of the International Association for Cybernetics (Namur)*, 9(4), 247-264.
- Ascott, R. (1967). Behaviorist Art and the Cybernetic Vision, Part Two. *Cybernetica: journal of the International Association for Cybernetics (Namur)*, 10(1), 25-56.
- Aspray, W. (1990). *John von Neumann and the origins of modern computing*. Cambridge, Mass.: MIT Press.
- Auslander, P. (2005). At the Listening Post, or, do machines perform? *International Journal of Performance Arts & Digital Media*, 1(1), 5-10.
- Auslander, P. (2008). *Liveness : performance in a mediatized culture* (2. ed.). London, New York: Routledge.
- Auslander, P. (2012). Digital Liveness: A Historico-Philosophical Perspective. *PAJ*:

## Bibliography

- A Journal of Performance and Art*, 102, 3-11.
- Austin, J. L. (1962). *How to do things with words*. Oxford: Clarendon.
- Automation. (n.d.). In *Merriam-Webster Online*. Retrieved from <http://www.merriam-webster.com/dictionary/automation>
- Baker, C. (2014). *The Missing Body: Performance in the Absence of the Artist*. Lethbridge, Alberta: Minuteman Press Leduc-Nisku.
- Barad, K. (2003). Posthumanist Performativity: Toward an Understanding of How Matter Comes to Matter. *Journal of Women in Culture and Society*, 28(3), 801-831.
- Barad, K. (2007). *Meeting the Universe Halfway : Quantum Physics and the Entanglement of Matter and Meaning* (Reprint. ed.). Durham: Duke University Press.
- Barad, K. (2011). Nature's Queer Performativity. *Qui Parle*, 19(2), 121-158.
- Barad, K. (2012). Intra-actions. *Mousse Magazine*, 76-81
- Baran, P. (1964). *On distribution communications: Introduction to distributed communications networks*. Available from [http://www.rand.org/content/dam/rand/pubs/research\\_memoranda/2006/RM3420.pdf](http://www.rand.org/content/dam/rand/pubs/research_memoranda/2006/RM3420.pdf)
- Baran, P. (2002). The beginnings of packet switching: Some underlying concepts. *IEEE Communications Magazine*, 40(7), 42-48.
- Barcena, M. B., Wueest, C., & Lau, H. (2014). *Security Response*. Available from <https://www.symantec.com/content/dam/symantec/docs/white-papers/how-safe-is-your-quantified-self-en.pdf>
- Barker, M. (2012). *LIVE TO YOUR LOCAL CINEMA: The Remarkable Rise of Livecasting*. Hampshire: Palgrave Macmillan.
- Beaudouin-Lafon, M. (2008). Interaction Is the Future of Computing. In E. Thomas & D. W. McDonald (Eds.), *HCI Remixed: Reflections on Works That Have Influenced the HCI Community* (pp. 263-266). Cambridge, Massachusetts, London: The MIT Press.
- Bechmann, A., & Vahlstrup, P. B. (2015). Studying Facebook and Instagram data: The Digital Footprints software. *First Monday*, 20(12). Retrieved from <http://firstmonday.org/ojs/index.php/fm/article/view/5968/5166>
- Bedau, M. (2003). Artificial life: organization, adaptation and complexity from the bottom up. *Trends in cognitive sciences*, 7(11), 505-512.
- Bennett, J. (2010). *Vibrant Matter : A Political Ecology of Things*. Durham, N.C.: Duke University Press.
- Bentley, P. (2003). The meaning of code. In G. Stocker & C. Schöpf (Eds.), *Code: The language of our time* (pp. 33-36). Linz: Hatje Cantz.
- Bernardo, B.-L. (2009). Emergence and evolution of ATM networks in the UK, 1967 -2000. *Business History*, 51(1), 1-27.
- Berry, D. M. (2011). *The Philosophy of Software: Code and Mediation in the Digital Age*. Basingstoke: Palgrave Macmillan.
- Berry, D. M. (2012). The Social Epistemologies of Software. *Social Epistemology*, 26(3-4), 379-398.
- Berry, D. M. (2013). Introduction: What is Code and Software? *Life in Code and Software: Mediated Life in a Complex Computational Ecology*. Open Humanities Press.
- Berry, D. M. (2014). *Critical Theory and the Digital*. New York: Bloomsbury Academic.
- Billings, C. W. (1989). *Grace Hopper: Navy Admiral and Computer Pioneer*. Enslow

- Publishers, Inc.
- Birdsey, L., & Two, Y. M. (2015). Twitter Knows: Understanding the emergence of topics in social networks. *Proceedings of the 2015 Winter Simulation Conference*, 4009-4020.
- Bivens, R. (2015). The gender binary will not be deprogrammed: Ten years of coding gender on Facebook. *New Media & Society*.
- Black, M. J. (2002). *The art of code*. (Doctoral dissertation). University of Pennsylvania, Pennsylvania. Retrieved from <http://search.proquest.com/docview/305507258>
- Blackwell, A. (2002). *What is programming*. Paper presented at the 14th workshop of the Psychology of Programming Interest Group.
- Blas, Z., & Cárdenas, M. (2013). Imaginary Computational Systems: Queer Technologies and Transreal Aesthetics. *AI and Society*, 28(4), 559-566.
- Bolter, J. D., MacIntyre, B., Nitsche, M., & Farley, K. (2013). Liveness, Presence, and Performance in Contemporary Digital Media. In U. Ekman (Ed.), *Throughout: Art and Culture Emerging with Ubiquitous Computing*. Cambridge, Massachusetts, London: The MIT Press.
- Booher, J. (2008). Computability: Turing Machines and the Halting Problem. Notes for a PROMYS talk about Turing machines, the halting problem, and the arithmetic hierarchy. Retrieved from [http://stanford.edu/%7Ejbooher/expos/computability\\_promys.pdf](http://stanford.edu/%7Ejbooher/expos/computability_promys.pdf)
- Bookchin, N., & Shulgin, A. (1999). Introduction to Net.Art (1994-1999). Retrieved from <http://www.easylife.org/netart/>
- Borgdorff, H. (2011). The Production of Knowledge in Artistic Research. In M. Biggs & H. Karlsson (Eds.), *The Routledge Companion to Research in the Arts* (pp. 44-63). Oxon: Routledge.
- Borgdorff, H. (2014). Artistic Practices and Epistemic Things. In M. Schwab (Ed.), *Experimental Systems: Future Knowledge in Artistic Research*. Leuven University Press.
- Bratton, B. H. (2016). *The Stack: On Software and Sovereignty*. The MIT Press.
- Broeckmann, A. (2004). Runtime Art: Software, Art, Aesthetics. In the catalogue of the exhibition RUNTIME Art, Zagreb: Gallery VN. Retrieved from <http://web.archive.org/web/20040614202632/http://runtimeart.mi2.hr/TextAndreasBroeckmann>
- Broida, R. (2010, July 14). Stop Frustrating Pauses in YouTube Videos. *PCWorld*. Retrieved from [http://www.pcworld.com/article/201089/Stop\\_Frustrating\\_Pauses\\_in\\_YouTube\\_Videos.html](http://www.pcworld.com/article/201089/Stop_Frustrating_Pauses_in_YouTube_Videos.html)
- Brookfield, S. D. (1986). *Understanding and facilitating adult learning : a comprehensive analysis of principles and effective practices*. San Francisco: Jossey-Bass Publishers.
- Broy, M. (2002). Software Engineering From Auxiliary to Key Technology. In M. Broy & E. Denert (Eds.), *Software Pioneers* (pp. 10-13). Springer Berlin Heidelberg.
- Bucher, T. (2012a). *Programmed sociality: A software studies perspective on social networking sites*. (Doctoral Dissertation). University of Oslo, Oslo. Retrieved from <http://www.scribd.com/doc/148539178/Bucher-Ph-D-diss-download>
- Bucher, T. (2012b). Want to be on the top? Algorithmic power and the threat of invisibility on Facebook. *New Media & Society*, 0(0), 1-17.

- Bucher, T. (2013). Objects of Intense Feeling: The case of the Twitter API. *Computational Culture*(3).
- Burghardt, M. (2015). Introduction to Tools and Methods for the Analysis of Twitter Data. *10plus1: Living Linguistics*(1), 74-91.
- Burnham, J. (1970). Notes on art and information processing. In the catalogue of the exhibition SOFTWARE, New York: Jewish Museum. Retrieved from [https://monoskop.org/images/3/31/Software\\_Information\\_Technology\\_Its\\_New\\_Meaning\\_for\\_Art\\_catalogue.pdf](https://monoskop.org/images/3/31/Software_Information_Technology_Its_New_Meaning_for_Art_catalogue.pdf)
- Burrell, J. (2016). How the machine 'thinks': Understanding opacity in machine learning algorithms. *Big Data & Society*, 3(1), 1-12.
- Burrell, M. (2004). *Fundamentals of Computer Architecture*. New York: Palgrave Macmillan.
- Cameron, D., & Carroll, J. (2009). *Encoding Liveness: Performance and Real-Time Rendering in Machinima*. DIGRA '09 - Proceedings of the 2009 DIGRA International Conference: Breaking New Ground: Innovative in Games, Play, Practice and Theory, Brunel University. Retrieved from <http://www.digra.org/wp-content/uploads/digital-library/09291.37018.pdf>
- Carlos, L. (1998). introduction: Performance art was the one place where there were so few definitions *Performance: Live Art since the 60s*. Thames and Hudson.
- Casemajor, N. (2015). Digital Materialisms: Frameworks for Digital Media Studies. *Westminster Papers in Communication and Culture*, 10(1), 4-17.
- Castelle, M. (2013). Relational and Non-relational Models in the Extexualization of Bureaucracy. *Computational Culture*(3).
- Cayley, J. (2002). The Code is not the Text (unless it is the Text). *electronic book review*. Retrieved from <http://www.electronicbookreview.com/thread/electropoetics/literal>
- Cayley, J., & Howe, D. C. (2015). *Show us the pictures: 'Some Thing We Are'* [Artwork]. Vancouver: ISEA 2015. Retrieved from <http://thereadersproject.org/installations/sutp.html> - sutp
- Chaitin, G. J. (1987). *Algorithmic Information Theory*. Cambridge: Cambridge University Press.
- Chaitin, G. J. (1987/[1975]). Randomness and Mathematical Proof *Information, randomness & incompleteness : Papers on algorithmic information theory*. Singapore: World Scientific. (Reprinted from: *Scientific American* 232, 47-52, May 1975).
- Chakraborty, S., & Das, D. (2014). An Overview of Face Liveness Detection. *International Journal on Information Theory (IJIT)*, 3(2), 11-25.
- Chandra, A. K., & Harel, D. (1980). Computable Queries for Relational Data Bases. *Journal of Computer and System Sciences*, 21(2), 156-178.
- Chapman, R., Burns, A., & Wellings, A. (1993). Worst-case Timing Analysis of Exception Handling in Ada. In L. Collingbourne (Ed.), *Ada: Towards Maturity* (pp. 148-164). Amsterdam, Oxford, Washington, Tokyo: IOS Press.
- Charlton, J. (2014). Post Screen Not Displayed. In H. Ferreira & A. Vicente (Eds.), *Post-Screen: Device, Medium and Concept* (pp. 170-182). Lisbon: CIEBA-FBAUL.
- Chatzichristodoulou, M. (2012). *Cyberformance? Digital or Networked Performance? Cybertheaters? Or Virtual Theatres? ... or all of the above?* Paper presented at the Cyposium: cyberformance symposium, Retrieved from <http://www.cyposium.net/selected-presentations/chatzichristodoulou/>

- Chu, H. (2007[1996]). *The Sound of Market* 《股·市·聲·動》 [Artwork]. Hong Kong: Microwave International New Media Arts Festival. Retrieved from <http://www.microwavefest.net/festival2007/artists/artist07.html>
- Chun, W. H. K. (2008a). The Enduring Ephemeral, or the Future Is a Memory. *Critical Inquiry*, 35(1), 148-171.
- Chun, W. H. K. (2008b). On "Sourcery," or Code as Fetish. *Configurations*, 16(3), 299-324.
- Chun, W. H. K. (2011a). Crisis, Crisis, Crisis, or Sovereignty and Networks. *Theory, Culture & Society*, 28(6), 91-112.
- Chun, W. H. K. (2011b). *Programmed Visions : Software and Memory*. Cambridge, Mass.: MIT Press.
- Chun, W. H. K. (2015). Networks NOW: Belated Too Early. In D. M. Berry & M. Dieter (Eds.), *Postdigital aesthetics: Art, Computation and Design* (pp. 289-315). Palgrave Macmillan.
- Chun, W. H. K. (2016). *Updating to Remain the Same: Habitual New Media*. The MIT Press.
- Cisco Systems. (2013). *An Innovative Business Model for Cloud Providers* [White paper]. Retrieved from [http://www.cisco.com/c/dam/en\\_us/about/ac79/docs/sp/An-Innovative-Business-Model-for-Cloud-Providers-Whitepaper.pdf](http://www.cisco.com/c/dam/en_us/about/ac79/docs/sp/An-Innovative-Business-Model-for-Cloud-Providers-Whitepaper.pdf)
- Claypool, M., & Riedl, J. (1998). End-to-End Quality in Multimedia Applications. In B. Furht (Ed.), *Handbook of Multimedia Computing*. Boca Raton, London, New York, Washington, D.C: CRC Press.
- Clayton, R. (2004). *Stopping Spam by Extrusion Detection*. CEAS. Retrieved from <https://www.cl.cam.ac.uk/~rnc1/extrusion.pdf>
- Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6), 377-387.
- Codd, E. F. (1990). *The Relational Model for Database Management* (2nd ed.). Addison-Wesley Publishing Company, Inc.
- Collingbourne, L. (1993). Editorial. In L. Collingbourne (Ed.), *Ada: Towards Maturity*. Amsterdam, Oxford, Washington, Toyko: IOS Press.
- Consens, M. P., Cruz, I. F., & Mendelzon, A. O. (1992). Visualizing Queries and Querying Visualizations. *SIGMOD RECORD*, 21(1), 39-46.
- Coole, D., & Frost, S. (2010). Introducing the New Materialisms. In D. Coole & S. Frost (Eds.), *New Materialisms: Ontology, Agency, and Politics*. Durham, London: Duke University Press.
- Cox, G. (2007). Generator: The Value of Software Art. In J. Rugg & M. Sedgwick (Eds.), *Issues in Curating Contemporary Art and Performance*. Bristol, Chicago: Intellect Lrd.
- Cox, G. (2010). *Antithesis : The Dialectics of Software Art*. Århus: Digital Aesthetics Research Center.
- Cox, G. (2013). *Speaking Code: Coding as Aesthetic and Political Expression*. The MIT Press.
- Cox, G. (2015). *Real-time for Pirate Cinema*. Ljubljana: Aksioma - Institute for Contemporary Art.
- Cox, G. (2017, in press). RuntimeException()- Critique of Software Violence. In H. Pritchard, E. Snodgrass, & M. Cyzlik-carver (Eds.), *Executing Practices*. Autonomedia.
- Cox, G., & Lund, J. (2016). *The Contemporary Condition: Introductory Thoughts on Contemporaneity and Contemporary Art*. Aarhus: Sternberg Press.
- Cox, G., McLean, A., & Ward, A. (2000). The Aesthetics of Generative Code.



## Bibliography

- Retrieved from <http://generative.net/papers/aesthetics/>
- Cox, G., McLean, A., & Ward, A. (2004). Coding Praxis: Reconsidering the aesthetics of code. In O. Goriunova & A. Shulgin (Eds.), *Read\_me : Software Art & Cultures*. Århus: Digital Aesthetics Research Centre : University of Aarhus.
- Craighead, T. (2012). *A live portrait of Tim Berners-Lee (An early warning system)*. Bradford: National Media Museum. Retrieved from <http://thomson-craighead.net/tbl.html>
- Cramer, F. (2001). Digital Code and Literary Text. Retrieved from <http://www.dichtung-digital.org/2001/10/22-Cramer/index2engl.htm>
- Cramer, F. (2003). Ten These about Software Art. Retrieved from [http://cramer.pleintekst.nl/all/10\\_thesen\\_zur\\_softwarekunst/10\\_these\\_s\\_about\\_software\\_art.txt](http://cramer.pleintekst.nl/all/10_thesen_zur_softwarekunst/10_these_s_about_software_art.txt)
- Cramer, F. (2005). *Words Made flesh: Code, Culture, Imagination*. Rotterdam: Piet Zwart Institute.
- Cramer, F., & Fuller, M. (2008). Interface. In M. Fuller (Ed.), *Software Studies \ a lexicon*. The MIT Press.
- Cramer, F., & Gabriel, U. (2001). Software Art. *Netzliteratur*. Retrieved from [http://www.netzliteratur.net/cramer/software\\_art\\_-\\_transmediale.html](http://www.netzliteratur.net/cramer/software_art_-_transmediale.html)
- Crisell, A. (2012). *Liveness and Recording in the Media*. Palgrave Macmillian.
- Crockford, D. (2006). The application/json Media Type for JavaScript Object Notation (JSON). Retrieved from <https://www.ietf.org/rfc/rfc4627.txt>
- Davis, W. (2007). Television's Liveness: A Lesson from the 1920s. *Westminster Papers in Communication and Culture*, 4(2), 36-51.
- De Souza, P. (2010). Rethinking the Dissension between Software and Generative Art. *The International Journal of Technology, Knowledge and Society*, 6(5), 13-26.
- DeLanda, M. (1995). *The Geology of Morals: A Neomaterialist Interpretation*. Paper presented at the Virtual Futures 95 Conference, Warwick University  
Retrieved from <http://www.t0.or.at/delanda/geology.htm>
- DeLanda, M. (2003). 1000 Years of War. CTheory interview with Manuel de Landa. *ctheory*.
- Deleuze, G. (1968). *Différence et répétition*. (Disputats, Paris). Presses universitaires de France,, Paris.
- Deleuze, G. (1988). *Spinoza: Practical Philosophy*. San Francisco: City Lights Books.
- Deleuze, G., & Guattari, F. (1987). *A thousand plateaus capitalism and schizophrenia*. Minneapolis: University of Minnesota Press.
- Derrida, J. (1978). *Writing and difference*. Chicago, Ill.: University of Chicago Press.
- Derrida, J. (2002). *Acts of religion*. New York: Routledge.
- Dewey, J. (1991). *How we think*. Buffalo, N.Y.: Prometheus Books.
- Dietz, H. G., & Mattox, T. I. (2005). Compiler Optimizations Using Data Compression to Decrease Address Reference Entropy. In B. Pugh & C.-W. Tseng (Eds.), *Languages and Compilers for Parallel Computing: 15th Workshop, LCPC 2002, College Park, MD, USA, July 25-27, 2002. Revised Papers* (pp. 126-141). Berlin, Heidelberg: Springer Berlin Heidelberg.
- DiNucci, D. (1999). Fragmented Future. *Print*, 221
- Doane, M. A. (2006). Information, Crisis, Catastrophe. In W. H. K. Chun & T.

- Keenan (Eds.), *New Media Old Media: A History and Theory Reader* (Vol. 1, pp. 251-264). New York, London: Routledge.
- Donati, L. P., & Prado, G. (2001). Artistic Environments of Telepresence on the World Wide Web. *Leonardo*, 34(5), 437-442.
- Dorish, P. (2014). No SQL: The Shifting Materialities of Database Technology. *Computational Culture*, (4). Retrieved from <http://computationalculture.net/article/no-sql-the-shifting-materialities-of-database-technology>
- Drahansky, M. (2011). Liveness Detection in Biometrics. In G. Chetty & J. Yang (Eds.), *Advanced Biometric Technologies*. InTech.
- Dredge, S. (2014, Jun 30). How does Facebook decide what to show in my news feed? Retrieved from <http://www.theguardian.com/technology/2014/jun/30/facebook-news-feed-filters-emotion-study>
- Drucker, J. (2009). *SpecLab digital aesthetics and projects in speculative computing*. Chicago: University of Chicago Press.
- Eisenmann, T. R., Parker, G., & Alstyne, M. V. (2008). *Opening Platforms: How, When and Why*. Harvard Business School. Retrieved from [http://www.hbs.edu/faculty/Publication Files/09-030.pdf](http://www.hbs.edu/faculty/Publication%20Files/09-030.pdf)
- ELC3. (2016). If I wrote you a love letter would you write back? Retrieved from <http://collection.eliterature.org/3/work.html?work=if-I-wrote-you-a-love-letter>
- Electroboutique. (2005). *Electroboutique. works* [Artwork]. Retrieved from <http://www.electroboutique.com/works/4>
- Ellis, J. (1992/[1982]). *Visible Fictions: Cinema: Television: Video* (2 ed.). London, New York: Routledge.
- Emerson, L. (2014). *Reading writing interfaces : from the digital to the bookbound*. Minneapolis: University of Minnesota Press.
- Ernst, W. (2006). Dis/continuities: Does the Archive Become Metaphorical in Multi-Media Space? In W. H. K. Chun & T. Keenan (Eds.), *New media, old media : a history and theory reader* (pp. x, 418 sider). New York ; London: Routledge.
- Ernst, W. (2009). '*...Else Loop Forever*': *The Untimeliness of Media*. Paper presented at the I1 Senso della Fine conference, Urbino, Italy Retrieved from <https://www.medienwissenschaft.hu-berlin.de/de/medienwissenschaft/medientheorien/downloads/publikationen/ernst-else-loop-forever.pdf>
- Ernst, W. (2013a) *Ernst on Time-Critical Media: A mini-interview / Interviewer: J. Parikka*. Machinology. Retrieved from <https://jussiparikka.net/2013/03/18/ernst-on-microtemporality-a-mini-interview/>
- Ernst, W. (2013b). *Media Archaeology: Method and Machine versus History and Narrative of Media*. Minneapolis: University of Minnesota Press.
- Espinha, T., Zaidman, A., & Gross, H. G. (2014). *Web API growing pains: Stories from client developers and their code*. 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering and reverse Engineering (CSMR-WCRE), Belgium: CSMR-WCRE 2014.
- Etzkowitz, H., Kemelgor, C., Uzzi, B., Neuschatz, M., Seymour, E., Muley, L., & Alonzo, J. (2000). *Athena Unbound: The Advancement of Women in Science and Technology*. Cambridge: Cambridge University Press.
- Feuer, J. (1983). The Concept of Live Television: Ontology as Ideology. In E. A.

## Bibliography

- Kaplan (Ed.), *Regarding Television: Critical Approaches* (pp. 12-22). Washington, DC: University Press of America.
- Fischer, J., Majumdar, R., & Millstein, T. (2007). *Tasks: language support for event-driven programming*. Paper presented at the Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, Nice, France
- Fitzpatrick, D. (n.d.). Batch Processing. Retrieved from <http://www.computing.dcu.ie/~dfitzpat/content/batch-processing>
- Fitzpatrick, K. (2012). The Humanities, Done Digitally. In M. K. Gold (Ed.), *Debates in the digital humanities*. Minneapolis: Univ Of Minnesota Press.
- Foucault, M. (1972). *The Archaeology of Knowledge and the Discourse on Language*. New York: Pantheon Books.
- Frabetti, F. (2015). *Software Theory : A Cultural and Philosophical study*. London: Rowman & Littlefield International.
- Frigg, R. (2004). In What Sense is the Kolmogorov-Sinai Entropy a Measure for Chaotic Behaviour?—Bridging the Gap Between Dynamical Systems Theory and Communication Theory. *The British Journal for the Philosophy of Science*, 55(3), 411-434.
- Fuller, M. (2003). *Behind the blip : essays on the culture of software*. New York London: Autonomedia;Pluto.
- Fuller, M. (2004). Digital Objects. In O. Goriunova & A. Shulgin (Eds.), *read\_me: Software Art & Culture* (pp. 26-41). Aarhus: University of Aarhus.
- Gaboury, J. (2013). A Queer History of Computing. *Rhizome.org*. Retrieved from <http://rhizome.org/editorial/2013/feb/19/queer-computing-1/>
- Gabrys, J. (2011). *Digital Rubbish: A Natural History of Electronics*. The University of Michigan Press.
- Gadassik, A. (2010). At a loss for words: Televisual Liveness and Corporeal Interruption. *Journal of Dramatic Theory and Criticism*, XXIV(2), 117-134.
- Galanter, P. (2003). *What is Generative Art? Complexity theory as a context for art theory*. In GA2003–6th Generative Art Conference Citeaser.
- Galanter, P. (2008). *What is Complexism? Generative Art and the Cultures of Science and the Humanities*. GA2008, 11th Generative Art Conference, Milan, Italy Generative Design Lab.
- Galanter, P. (2010). *Complexity, Neuroaesthetics, and Computational Aesthetic Evaluation*. Paper presented at the 13th Generative Art Conference GA2010, Politecnico di Milano University, Italy
- Galanter, P. (2016). Generative Art Theory. In C. Paul (Ed.), *A Companion to Digital Art* (pp. 146-180). Wiley-Blackwell.
- Galloway, A. R. (2004). *Protocol : how control exists after decentralization*. Cambridge: MIT Press.
- Galloway, A. R. (2012). *The interface Effect*. Cambridge, UK: Polity Press.
- Galloway, A. R., & Thacker, E. (2009). On Narcolepsy. In J. Parikka & T. D. Sampson (Eds.), *The Spam Book: On Viruses, Porn and Other Anomalies From the Dark Side of Digital Culture*. Hampton Press.
- Garsiel, T., & Irish, P. (2011). How Browsers Work: Behind the scenes of modern web browsers. Retrieved from <http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/>
- Gayo-Avello, D. (2013). A Meta-Analysis of State-of-the-Art Electoral Prediction From Twitter Data. *Social Science Computer Review*, 1-31.
- Gehl, R. W. (2015). Building a Better Twitter: A Study of the Twitter Alternatives GNU social, Quitter, rstat.us, and Twister. *The Fibreculture Journal*, (26).

- Retrieved from <http://twentysix.fibrejournal.org/fcj-190-building-a-better-twitter-a-study-of-the-twitter-alternatives-gnu-social-quitter-rstat-us-and-twister/>
- GENERATOR. (2002). *GENERATOR* [Exhibition]. Retrieved from <http://generative.net/generator/>
- Generator.x. (2008). Generator.x 2.0: Beyond the Screen. Retrieved from <http://www.generatorx.no/20071130/generatorx-20-call/>
- Georgi, C. (2014). *Liveness on stage : intermedial challenges in contemporary British theatre and performance*. Berlin, Germany ; Boston, Massachusetts: De Gruyter.
- Gerlitz, C., & Helmond, A. (2013). The like economy: Social buttons and the data-intensive web. *New Media & Society*, 15(8), 1348-1365.
- Gibbs, G. (1988). *Learning by doing: A guide to teaching and learning methods*. FEU.
- Goodman, C. (1987). *Digital visions : computers and art*. New York: Abrams.
- Google. (2016, Mar 30, 2016). Google Earth API Developer's Guide. Retrieved from <https://developers.google.com/earth/ - troubleshooting>
- Goriunova, O. (2012). *Art Platforms and Cultural Production on the Internet*. New York: Routledge.
- Goriunova, O., & Shulgin, A. (2004). Read\_Me Today. In O. Goriunova & A. Shulgin (Eds.), *read\_me: Software Art & Culture* (pp. 17-22). Aarhus: University of Aarhus.
- Grosser, B. (2014). What do metrics want? How quantification prescribes social interaction on Facebook. *Computational Culture*, (4). Retrieved from <http://computationalculture.net/article/what-do-metrics-want>
- Gruenbaum, P. (2010, August 12). Web API Documentation Best Practices [Blog post]. Retrieved from <http://www.programmableweb.com/news/web-api-documentation-best-practices/2010/08/12>
- Hamp, S. (2010, Aug 11). Is JSON the Developer's Choice? Retrieved from <http://www.programmableweb.com/news/json-developers-choice/2010/08/11>
- Hansen, M., & Rubin, B. (2000-2001). *Listening Post* [Artwork]. London: Science Museum. Retrieved from [http://www.sciencemuseum.org.uk/visitmuseum/plan\\_your\\_visit/exhibitions/listening\\_post](http://www.sciencemuseum.org.uk/visitmuseum/plan_your_visit/exhibitions/listening_post)
- Hansen, N. B., Nørgård, R. T., & Halskov, K. (2014). *Crafting code at the demoscene*. Paper presented at the Proceedings of the 2014 conference on Designing interactive systems, Vancouver, BC, Canada
- Harrison, E. (2009). Toy Town. Retrieved from <http://www.ellieharrison.com/index.php?pagecolor=3&pageId=project-toytown>
- Hausdorff, F. (1957). *Set theory*. Providence, R.I.: American Mathematical Society.
- Hayles, N. K. (1990). *Chaos bound : orderly disorder in contemporary literature and science*. Ithaca, N.Y.: Cornell University Press.
- Hayles, N. K. (1991). *Chaos and order : complex dynamics in literature and science*. Chicago, London: University of Chicago Press.
- Hayles, N. K. (2005). *My mother was a computer : digital subjects and literary texts*. Chicago: University of Chicago Press.
- Hayles, N. K. (2006). Traumas of Code. *Critical Inquiry*, 33(1), 136-157.
- Hayles, N. K. (2010). How We Read: Close, Hyper, Machine. *ADE Bulletin*(150), 62-

- 79.
- He, Y., Hou, Y., & Wang, Y. (2010). *Liveness iris detection method based on the eye's optical features*. Proc. SPIE 7838, Optics and Photonics for Counterterrorism and Crime Fighting VI and Optical Materials in Defence System Technology VII SPIE.
- Health, S., & Skirrow, G. (1977). Television: A World in Action. *Screen*, 18(2), 7-59.
- Helmond, A. (2013). The Algorithmization of the Hyperlink. *Computational Culture*, (3). Retrieved from <http://computationalculture.net/article/the-algorithmization-of-the-hyperlink>
- Helmond, A. (2015). *The Web as Platform: Data Flows in Social Media*. (Doctoral Dissertation). Universiteit van Amsterdam, Amsterdam. Retrieved from [http://www.annahelmond.nl/wordpress/wp-content/uploads/2015/08/Helmond\\_WebAsPlatform.pdf](http://www.annahelmond.nl/wordpress/wp-content/uploads/2015/08/Helmond_WebAsPlatform.pdf)
- Hertz, G. (2012). Garnet Hertz - Interview with Natalie Jeremijenko. In G. Hertz (Ed.), *Critical Making: Conversations* (pp. 40). Garnet Hertz.
- Hertz, G., & Parikka, J. (2012). Zombie Media: Circuit Bending Media Archaeology into an Art Method. *Leonardo*, 45(5), 424-430.
- Hill, C., Corbett, C., & Rose, A. S. (2010). *Why So Few? Women in Science, Technology, Engineering, and Mathematics* [Report]. Retrieved from <https://www.aauw.org/files/2013/02/Why-So-Few-Women-in-Science-Technology-Engineering-and-Mathematics.pdf>
- Hofstadter, D. R. (1980 [1979]). *Gödel, Escher, Bach: An Eternal Golden Braid*. New York: Vintage Books.
- Hogg, T., Lerman, K., & Smith, L. M. (2013). Stochastic Models Predict User Behavior in Social Media. *SocialCom*, 2(1), 63-68.
- Hookway, B. (2014). *Interface : A Genealogy of Mediation and Control*. Cambridge, Massachusetts: MIT Press.
- Hopper, G. M. (1955). *Automatic Coding for Digital Computers*. Paper presented at the The High Speed Computer Conference, Louisiana State University Retrieved from [http://www.mirror-service.org/sites/www.bitsavers.org/pdf/univac/HopperAutoCodingPaper\\_1955.pdf](http://www.mirror-service.org/sites/www.bitsavers.org/pdf/univac/HopperAutoCodingPaper_1955.pdf)
- Howe, D. C. (2015). Surveillance Countermeasures: Expressive Privacy via Obfuscation. *A Peer-Reviewed Journal About*, 4(1). Retrieved from [http://www.aprja.net/?page\\_id=2283](http://www.aprja.net/?page_id=2283)
- Howe, D. C., & Nissenbaum, H. (2009). TrackMeNot: Resisting Surveillance in Web Search. In I. Kerr, V. Steeves, & C. Lucock (Eds.), *Lessons from Identity Trial: Privacy, Anonymity and Identity in a Networked Society* Oxford: Oxford University Press.
- Howe, D. C., Toubiana, V., LSubramanian, L., & Nissenbaum, H. (2011). TrackMeNot: Enhancing the privacy of Web Search. *ArXiv*.
- Hyde, R. (2004). *Write great code Volume 1 : understanding the machine*. San Francisco: No Starch Press.
- IBM. (1990, 2010). What is batch processing? Retrieved from [https://www.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos.zconcepts/zconc\\_whatbatch.htm](https://www.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos.zconcepts/zconc_whatbatch.htm)
- Jacucci, G., Wagner, M., Wagner, I., Giaccardi, E., Annunziato, M., Breyer, N., . . . Schuricht, S. (2010). *ParticipArt: Exploring participation in interactive art installations*. 2010 IEEE International Symposium on Mixed and Augmented Reality - Arts, Media, and Humanities.

- Jeremijenko, N. (1995). *Live Wire* [Artwork]. Retrieved from [http://tech90s.walkerart.org/nj/transcript/nj\\_04.html](http://tech90s.walkerart.org/nj/transcript/nj_04.html)
- Johnson, S. (2001). *Emergence : the connected lives of ants, brains, cities, and software*. London: The Penguin Press.
- Johnston, D. J. (2016). *Aesthetic Animism: Digital Poetry's Ontological Implications*. MIT Press.
- Jones, A. (2012). The Now and the Has Been: Paradoxes of Live Art in History. In A. Jones & A. Heathfield (Eds.), *Perform, Repeat, Record Live Art in History*. Bristo, Chicago: Intellect.
- Keehner, J. (2007). Milliseconds are focus in algorithmic trades. *Reuters*. Retrieved from <http://www.reuters.com/article/us-exchanges-summit-algorithm-idUSN1046529820070511>
- Kelty, C. M. (2008). *Two Bits: The Cultural Significance of Free Software*. Durham, London: Duke University Press.
- Kephart, J. O., & Chess, D. M. (2003). The vision of autonomic computing. *Computer*, 36(1), 41-50.
- Kim, E. E., & Toole, B. A. (1999). Ada and the First Computer. *Scientific American*, 76-81.
- Kirschenbaum, M. G. (2012). *Mechanisms : new media and the forensic imagination*. Cambridge, Mass. ; London: MIT Press.
- Kistler, T. (1997). Dynamic runtime optimization. In H. Mössenböck (Ed.), *Modular Programming Languages: Joint Modular Languages Conference, JMLC'97 Linz, Austria, March 19–21, 1997 Proceedings* (pp. 53-66). Berlin, Heidelberg: Springer Berlin Heidelberg.
- Kitchin, R., & Dodge, M. (2011). *Code/space : software and everyday life*. Cambridge, Mass.: MIT Press.
- Kittler, F. (1995). There is No Software. *Ctheory.net*. Retrieved from <http://www.ctheory.net/articles.aspx?id=74>
- Kluszczyński, R. W. (2010). Strategies of interactive art. *Journal of Aesthetics & Culture*, 2. Retrieved from <http://www.aestheticsandculture.net/index.php/jac/article/view/5525>
- Knotts, S. (2013). METAL TV: Composers Lab Shelly Knotts. Retrieved from <https://www.youtube.com/watch?v=SbW9Bhp3JZU>
- Kowalski, R. (1979). Algorithm = logic + control. *Commun. ACM*, 22(7), 424-436.
- Kumar, A. A. (2015). *Digital Signal Processing, Second Edition*. Delhi: PHI Learning Private Limited.
- Kurose, J. F., & Ross, K. W. (2013). *Computer Networking: A Top-Down Approach*. Pearson Education.
- Laplante, P. A. (2000). *Dictionary of Computer Science, Engineering and Technology* CRC Press.
- Latour, B. (1996). On actor-network theory: A few clarifications. *Soziale Welt*, 47(4), 369-381.
- Latour, B. (1999). *Pandora's hope : essays on the reality of science studies*. Cambridge, Mass.: Harvard University Press.
- Law, J., & Singleton, V. (2005). Object Lessons. *Organization*, 12(3), 331-355.
- Lazaris, L. (2013). Using White Space for Readability in HTML and CSS. *Smashing Magazine*. Retrieved from <https://www.smashingmagazine.com/2013/02/using-white-space-for-readability-in-html-and-css/>
- Lessig, L. (2006). *Code v2*. Basic Books.

## Bibliography

- Li, J., & Rao, H. R. (2010). Twitter as a Rapid Response News Service: An Exploration in the Context of the 2008 China Earthquake. *The Electronic Journal of Information Systems in Developing Countries*, 42(4), 1-22.
- Link, D. (2006). There Must Be an Angel: On the Beginnings of the Arithmetics of Rays. In S. Zielinski & D. Link (Eds.), *Variantology 2: On Deep Time Relations of Arts, Sciences and Technologies* (pp. 15-42). Cologne: Walther König.
- Live Art Development Agency. (n.d). What is Live Art. Retrieved from <http://www.thisisliveart.co.uk/about/what-is-live-art>
- Louden, K. C., & Lambert, K. A. (2012). *Programming Languages: Principles and Practice* (3rd ed.). Boston: Cengage Learning.
- Machta, J. (1999). Entropy, information, and computation. *American Journal of Physics*, 67(12), 1074-1077.
- Mackenzie, A. (2005). The Performativity of Code: Software and Cultures of Circulation. *Theory, Culture and Society*, 22(1), 71-92.
- Mackenzie, A. (2006). *Cutting Code: Software and Sociality*. New York: Peter Lang.
- Mackenzie, A. (2012). More parts than elements: how databases multiply. *Environment and Planning D: Society and Space*, 30, 335-350.
- Mackenzie, A. (2013). Programming subjects in the regime of anticipation: Software studies and subjectivity. *Subjectivity*, 6(4), 391-405.
- Maeda, J. (2004). *Creative code*. London: Thames & Hudson.
- Maigret, N. (2014). *The Pirate Cinema - A cinematic collage generated by P2P users* [Artwork]. Retrieved from <http://thepiratecinema.com/>
- Manovich, L. (1999). Database as symbolic form. *Convergence*, 5(2), 80-99.
- Manovich, L. (2001). *The language of new media*.
- Manovich, L. (2013). *Software Takes Command*. Bloomsbury Academic.
- Marino, M. C. (2006). Critical Code Studies. *electronic book review*. Retrieved from <http://www.electronicbookreview.com/thread/electropoetics/codology>
- Marino, M. C. (2014). Field Report for Critical Code Studies. *Computational Culture*, (4). Retrieved from <http://computationalculture.net/article/field-report-for-critical-code-studies-2014%E2%80%A8-fn-1946-1>
- Mason, R., & McKendrick, J. (2015). *The Rising Value of APIs: MuleSoft's Predictions for 2016* [Whitepaper]. Retrieved from <https://www.mulesoft.com/lp/whitepaper/api/rising-value-apis>
- Master. (n.d.). In *English Oxford Living Dictionaries*. Retrieved from <https://en.oxforddictionaries.com/definition/master>
- McCormick, T. H., Lee, H., Cesare, N., Shojaie, A., & Spiro, E. S. (2015). Using Twitter for Demographic and Social Science Research: Tools for Data Collection and Processing. *Sociological Methods & Research*.
- McIntyre, R. B., Lord, C. G., Gresky, D. M., Frye, G. D. J., & Bond Jr, C. F. (2005). A Social Impact Trend in the Effects of Role Model on Alleviating Women's Mathematics Stereotype Threat. *Current Research in Social Psychology*, 10(9).
- McLean, A. (2004). Hacking Perl in NightClubs. *Perl.com*. Retrieved from <http://www.perl.com/pub/2004/08/31/livecode.html>
- McLean, A. (2011). *Artist-Programmers and Programming Languages for the Arts*. (Doctoral Dissertation). University of London. Retrieved from <http://yaxu.org/thesis/>
- McLean, A. (2014). *Making programming languages to dance to: live coding with tidal*. Paper presented at the Proceedings of the 2nd ACM SIGPLAN

- international workshop on Functional art, music, modeling & design,  
Gothenburg, Sweden
- McPherson, T. (2006). Reload: Liveness, Mobility and the Web. In Wendy Hui K. Chun & T. Keenan (Eds.), *New Media Old Media: A History and Theory Reader* (pp. 199- 208). New York, Oxon: Routledge.
- Meinel, C., & Sack, H. (2013). *Internetworking: Technological Foundations and Applications*. Berlin: Springer.
- Metz, C. (2014, Feb 4). This is what you build to juggle 6,000 tweets a second. *WIRED Business*. Retrieved from <http://www.wired.com/2014/04/twitter-manhattan/>
- Meysenburg, M. (2014). *Introduction to Programming Using Processing, Second Edition*. Crete: lulu.com.
- Miyazaki, S. (2012). Algorhythmics: Understanding Micro-temporality in Computational Cultures. *Computational Culture*, (2). Retrieved from <http://computationalculture.net/article/algorhythmics-understanding-micro-temporality-in-computational-cultures>
- Montfort, N. (2013). *10 PRINT CHR*. Cambridge, Mass.: MIT Press.
- Morris, J. W. (2015). Curation by code: Infomediaries and the data mining of taste. *European Journal of Cultural Studies*, 18(4-5), 446-463.
- Murphie, A. (2013). Convolving Signals: Thinking the performance of computational processes. *Performance Paradigm*, (9). Retrieved from <http://www.performanceparadigm.net/index.php/journal/article/view/135>
- Murtaugh, M. (2008). Interaction. In M. Fuller (Ed.), *Software studies : a lexicon* (pp. 143-148.). Cambridge, Mass.: MIT Press.
- Nakamoto, S. (2008). Bitcoin: A Peer-to-Peer Electronic Cash System. *bitcoin.org*. Retrieved from <https://bitcoin.org/bitcoin.pdf>
- Nakov, S., Dimitrov, D., Germanov, H., Stoyanov, M., Valkov, K., Bivas, M., & Yosifov, Y. (2013). *Fundamentals of computer programming with C#*: Sofia.
- Newell, C. (2009). *Place, authenticity, and time: a framework for liveness in synthetic speech*. (Doctoral Dissertation). The University of York, United Kingdom.
- Newton, C. (2016). Here's how Twitter's new algorithmic timeline is going to work. *The Verge*. Retrieved from <http://www.theverge.com/2016/2/6/10927874/twitter-algorithmic-timeline>
- Norman, S. J. (2016). *Senses of Liveness for Digital Times*. Paper presented at the IETM Amsterdam, IETM Amsterdam Plenary Meeting [Opening Keynote Speech]. Retrieved from <https://www.ietm.org/en/themes/senses-of-liveness-for-digital-times>
- O'Dwyer, R. (2015). The Revolution Will (not) be Decentralised: Blockchain-based Technologies & the Commons. *COMMONS TRANSITION*. Retrieved from <http://commonstransition.org/the-revolution-will-not-be-decentralised-blockchains/>
- O'Dwyer, R. (2016). Blockchains and Their Pitfalls. In T. Scholz & N. Schneider (Eds.), *Ours to Hack and to Own: The Rise of Platform Cooperativism, A new vision for the future of work and a fairer internet*. OR Books.
- O'Reilly, T. (2005). What Is Web 2.0. Retrieved from <http://www.oreilly.com/pub/a/web2/archive/what-is-web-20.html>
- O'Reilly, T., & Battelle, J. (2004). *Opening Welcome: The State of the Internet*



## Bibliography

- Industry*. Web 2.0 Conference, Hotel Nikko, San Francisco, CA. Retrieved from [http://web2con.com/presentations/web2con/intro\\_tim\\_john.ppt](http://web2con.com/presentations/web2con/intro_tim_john.ppt)
- Olaiya, F. (2012). Application of Data Mining Techniques in Weather Prediction and Climate Change Studies. *International Journal of Information Engineering and Electronic Business*, 4(1), 51-59.
- Olthof, T. (2009). (GENERATIVE) VISUAL ART IN FLEX4/AS3. Retrieved from <http://www.timenolthof.nl/projects/tutorials/GenerativeArtTutorial/GenerativeArtTutorial.pdf>
- Oracle. (2012). *An overview of Research Tracking: Research Candidate Management and Thesis Processing* [White Paper]. Retrieved from [https://www.uwplatt.edu/files/its/PASS/Upgrade\\_Schedule/Bundle26/Research\\_Tracking\\_Benefits\\_Document.pdf](https://www.uwplatt.edu/files/its/PASS/Upgrade_Schedule/Bundle26/Research_Tracking_Benefits_Document.pdf)
- Owicki, S., & Lamport, L. (1982). Proving Liveness Properties of Concurrent Programs. *ACM Transactions on Programming Languages and Systems*, 4(3), 455-495.
- Palmer, D. (2008). *Participatory Media: Visual Culture in Real Time*. VDM Publishing.
- Pan, G., Wu, Z., & Sun, L. (2008). Liveness Detection for Face Recognition. In K. Dalac, M. Grgic, & M. S. Bartlett (Eds.), *Recent Advances in Face Recognition*. InTech.
- Pan, H., Tilakaratne, C., & Yearwood, J. (2003). Predicting the Australian Stock Market Index Using Neural Networks Exploiting Dynamical Swings and Intermarket Influences. In T. D. Gedeon & L. C. C. Fung (Eds.), *AI 2003: Advances in Artificial Intelligence: 16th Australian Conference on AI, Perth, Australia, December 3-5, 2003. Proceedings* (pp. 327-338). Berlin, Heidelberg: Springer Berlin Heidelberg.
- Parikka, J. (2010). Ethologies of Software Art: What Can a Digital Body of Code Do? . In S. O'Sullivan (Ed.), *Deleuze and Contemporary Art*. Edinburg University Press.
- Parikka, J. (2011). Operative Media Archaeology: Wolfgang Ernst's Materialist Media Diagrammatics. *Theory, Culture & Society*, 28(5), 52-74.
- Parikka, J. (2012). *What is media archaeology*. Cambridge: Polity Press.
- Parikka, J. (2015). The Universal Viral Machine: Bits, Parasites and the Media Ecology of Network Culture. *ctheory*. Retrieved from <http://www.ctheory.net/articles.aspx?id=500>
- Parikka, J., & Sampson, T. D. (2009a). Bad Objects. In J. Parikka & T. D. Sampson (Eds.), *The Spam Book: On Viruses, Porm and Other Anomalies From the Dark Side of Digital Culture*. Hampton Press.
- Parikka, J., & Sampson, T. D. (2009b). An Introduction. In J. Parikka & T. D. Sampson (Eds.), *The Spam Book: On Viruses, Porm and Other Anomalies From the Dark Side of Digital Culture*. Hampton Press.
- Parisi, L. (2013). *Contagious Architecture: Computation, Aesthetics, and Space*. Cambridge: MIT Press.
- Parisi, L., & Fazi, M. B. (2014). Do Algorithms Have Fun? On Completion, Indeterminacy and Autonomy in Computation. In O. Goriunova (Ed.), *Fun and Software*. New York, London: Bloomsbury Publishing.
- Parlante, N. (1999). Pointers and Memory. Retrieved from <http://cslibrary.stanford.edu/102/PointersAndMemory.pdf>
- Patterson, D. A., & Hennessy, J. L. (2007). *Computer Organization and Design: The Hardware/Software Interface* (3 ed.). Burlington: Morgan Kaufmann.
- Paul, C. (2002). *Whitney Artport Comissions: CODEDOC* [Exhibition]. Retrieved

- from <http://artport.whitney.org/commissions/codedoc/>
- Paul, C. (2003). CODEDOC II. In the catalogue of the exhibition Ars Electronica 2003, Linz: Ars Electronica. Retrieved from [http://90.146.8.18/en/archives/festival\\_archive/festival\\_catalogs/festival\\_artikel.asp?iProjectID=12323](http://90.146.8.18/en/archives/festival_archive/festival_catalogs/festival_artikel.asp?iProjectID=12323)
- Paul, C. (2007). The Database as System and Cultural Form: Anatomies of Cultural Narratives. In V. Vesna (Ed.), *Database Aesthetics*. Minneapolis London: University of Minnesota Press.
- Pearson, M. (2011). *Generative Art: A Practical Guide Using Processing*. New York: Manning Publications Co.
- Peddinti, S. T., & Saxena, N. (2010). On the Privacy of Web Search Based on Query Obfuscation: A Case Study of TrackMeNot. In M. J. Atallah & N. J. Hopper (Eds.), *Privacy Enhancing Technologies: 10th International Symposium, PETS 2010, Berlin, Germany, July 21-23, 2010. Proceedings* (pp. 19-37). Berlin, Heidelberg: Springer Berlin Heidelberg.
- Penny, S. (2009). *Art and Artificial Life - a Primer*. Proceedings of the 6th European Conference (ECAL), Springer, Berlin. Retrieved from [http://simonpenny.net/texts/Resources/a\\_life.pdf](http://simonpenny.net/texts/Resources/a_life.pdf)
- Peppler, K. A., & Kafai, Y. B. (2009). *Creative coding: Programming for personal expression*. The 8th International Conference on Computer Supported Collaborative Learning, Rhodes, Greece.
- Pereira, F., & Ebrahimi, T. (2002). *The MPEG-4 Book*. Prentice Hall.
- Phelan, P. (1993). *Unmarked : the politics of performance* (Paperback. ed.). London: Routledge.
- Postel, J. (1981a). *Internet Protocol - Darpa Internet Program Protocol Specification (RFC 793)* [Specification]. Retrieved from Information Sciences Institute: <https://tools.ietf.org/html/rfc791>
- Postel, J. (1981b). *Transmission Control Protocol - Darpa Internet Program Protocol Specification (RFC 791)* [Specification]. Retrieved from Information Sciences Institute: <https://tools.ietf.org/html/rfc793>
- Pradhan, D. K., & Harris, I. G. (2009). *Practical design verification*. Cambridge, UK: Cambridge University Press.
- Pritchard, H., & Prophet, J. (2015). Diffractive Art Practices: Computation and the Messy Entanglements between Mainstream Contemporary Art, and New Media Art. *Artnodes*, 15. Retrieved from <http://journals.uoc.edu/index.php/artnodes/article/view/n15-pritchard-prophet/2710>
- Prophet, J. (2001). "TechnoSphere": "Real" Time, "Artificial" Life. *Leonardo*, 34(4), 309-312.
- Raley, R. (2009). List(en)ing Post. In F. J. Ricardo (Ed.), *Literary Art in Digital Performance: Case Studies in New Media Art and Criticism*. New York, London: The Continuum International Publishing Group Inc.
- Raley, R. (2012). Distracted Reading. *ELMCIP Anthology*. Retrieved from <http://anthology.elmcip.net/materials/syllabi/Raley-2012-US.pdf>
- Ramsay, S. (2004). Databases. In S. Schreibman, R. Siemens, & J. Unsworth (Eds.), *A Companion to Digital Humanities*. Oxford: Blackwell.
- Reas, C., & Fry, B. (2014). *Processing: A Programming Handbook for Visual Designers and Artists* (2 ed.). Massachusetts Institute of Technology.
- Reichardt, J. (1968). Cybernetic Serendipity: the computer and the arts. In the catalogue of Exhibition Catalogue, London, New York: Studio International

- Rheinberger, H.-J. (1997). *Toward a history of epistemic things : synthesizing proteins in the test tube*. Stanford, Calif.: Stanford University Press.
- Rheingold, H. (1991). *Virtual Reality*. Summit Books.
- Ridgway, R. (2015). Personalisation as currency. *A Peer-Reviewed Journal About Datafied Research*, 4(1). Retrieved from [http://www.aprja.net/?page\\_id=2283](http://www.aprja.net/?page_id=2283)
- Roebuck, K. (2011). *Virtual Desktops : High-impact Strategies - What You Need to Know: Definitions, Adoptions, Impact, Benefits, Maturity, Vendors*. Dayboro: Emereo Publishing.
- Roffe, J., & Stark, H. (2015). *Deleuze and the non/human*. Houndmills, Basingstoke, Hampshire: Palgrave Macmillan.
- Rolling Jr, J. H. (2014). Artistic Method in Research as a Flexible Architecture for Theory Building. *International Review of Qualitative Research*, 7(2), 161-168.
- Rosen, R. (2014). Internet Control Message Protocol (ICMP) *Linux Kernel Networking: Implementation and Theory* (pp. 37-61). Berkeley, CA: Apress.
- Sanden, P. (2013). *Liveness in modern music : musicians, technology, and the perception of performance*. New York: Routledge.
- Scannell, P. (1996). *Radio, Television and Modern Life*. Oxford, Massachusetts: Blackwell Publishers Inc.
- Schön, D. A. (1983). *The reflective practitioner : how professionals think in action*. New York: Basic Books.
- Schönlieb, C.-B., & Schubert, F. (2013). Random simulations for generative art construction – some examples. *Journal of Mathematics and the Arts*, 7(1), 29-39.
- Schuller, P. (2014, Apr 2). Manhattan, our real-time, multi-tenant distributed database for Twitter Scale [Blog post]. Retrieved from <https://blog.twitter.com/2014/manhattan-our-real-time-multi-tenant-distributed-database-for-twitter-scale>
- Seward, Z. M. (2014, Aug 11). Twitter admits that as many as 2 million of its active users are automated. *QUARTZ*.online. Retrieved from <http://qz.com/248063/twitter-admits-that-as-many-as-23-million-of-its-active-users-are-actually-bots/>
- Shanken, E. A. (2002). Art in the Information Age: Technology and Conceptual Art. *Leonardo*, 35(4), 433-438.
- Shannon, C. E. (1948). A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27, pp. 379-423, 623-656.
- Shneiderman, B. (1994). Dynamics Queries For Visual Information Seeking. *IEEE software*, 11(6), 70-77.
- Simanowski, R. (2008). Close Reading in the Realm of Static and Dynamic Texts. *Digital Humanities Quarterly*, 2(1). Retrieved from <http://digitalhumanities.org/dhq/vol/2/1/000018/000018.html>
- Simanowski, R. (2011). *Digital art and meaning reading kinetic poetry, text machines, mapping art, and interactive installations*. Minneapolis: University of Minnesota Press.
- Snodgrass, E. (2017, in press). Ecologies of the executable. In E. Snodgrass, H. Pritchard, & M. Tyzlik-Carver (Eds.), *Exeuting Practices*. Autonomedia.
- Sofaer, J. (2002). What is Live Art? *Joshua Sofaer Art*. Retrieved from <http://www.joshuasofaer.com/2011/06/what-is-live-art/>
- Solaas, L., Watz, M., & Whitelaw, M. (2010) *Generative Practice. The state of the art/Interviewer: J. Levine*. (Vol 57), DiGICULT | Digital art, Design and

- Culture, Online. Retrieved from <http://www.digicult.it/digimag/issue-057/generative-practice-the-state-of-the-art/>
- Sollfrank, C. (2003). A small artist makes the machine do the work! Retrieved from [http://net.art-generator.com/src/projekt\\_engl.html](http://net.art-generator.com/src/projekt_engl.html)
- Sollfrank, C. (2012). *Performing the Paradoxes of Intellectual Property. A practice-led Investigation Into the Conflicting Relationship between Copyright and Art*. (Doctoral Dissertation). University of Dundee, United Kingdom.
- Sondheim, A. (2001). Introduction: Codeword. *ABR*, 22(6).
- Soon, W. (2014a). *Hello Zombies* [Artwork]. Retrieved from <http://siusoon.net/home/?p=1273>
- Soon, W. (2014b). Post-digital approach: Rethinking Digital Liveness in 'The Likes of Brother Cream Cat'. *A Peer-Reviewed Journal About Post-Digital*, 3(1). Retrieved from <http://www.aprja.net/?p=1814>
- Soon, W. (2015a). *Zombies as the living dead. Datafied Research* L A peer-Reviewed Newspaper. Aarhus, Berlin: Digital Aesthetics Research Center, Aarhus University and reSource transmediale culture Berlin/Transmediale. Retrieved from [http://www.aprja.net/?page\\_id=2133](http://www.aprja.net/?page_id=2133)
- Soon, W. (2015b). *Zombies in Spam Culture. Tracing Data: What you see is not what we write Proceedings*, 86-94. Retrieved from <http://www.writingmachine-collective.net/wordpress/?p=704>
- Soon, W. (2015c). *Zombification: the living dead in spam. A Peer-Reviewed Journal About Datafied Research*, 4(1). Retrieved from <http://www.aprja.net/?p=2471>
- Soon, W. (2016a). *Interfacing with questions: The unpredictability of live queries*. Proceedings of the 2016 International Conference on Live Interfaces, Sussex, United Kingdom: the Experimental Music Technologies (EMuTe) Lab, University of Sussex, in collaboration with REFRAME Books, Falmer, UK. Retrieved from <http://reframe.sussex.ac.uk/reframebooks/archive2016/live-interfaces/>
- Soon, W. (2016b). *Microtemporalities: At the Time of Loading-in-progress*. ISEA2016 Hong Kong Cultural R>evolution, Hong Kong: School of Creative Media, City University of Hong Kong. Retrieved from [https://isea2016.scm.cityu.edu.hk/openconf/modules/request.php?module=oc\\_program&action=summary.php&id=249](https://isea2016.scm.cityu.edu.hk/openconf/modules/request.php?module=oc_program&action=summary.php&id=249)
- Soon, W. (2016c). *The Spinning Wheel of Life (work-in-progress)* [Artwork]. Retrieved from <http://siusoon.net/home/?p=1407>
- Soon, W. (2016, in press). *Executing queries as a form of artistic practice. International Art Conference*.
- Soon, W., & Pritchard, H. (2012a). *Thousand Questions* [Artwork]. Hong Kong: Microwave International New Media Arts Festival 2012. Retrieved from <http://microwavefest.net/festival2012/> - [!/if\\_i\\_wrote\\_as](http://www.siusoon.net/home/?p=900)
- Soon, W., & Pritchard, H. (2012b). *thousands of other questions* [Artwork]. Retrieved from <http://www.siusoon.net/home/?p=900>
- Sprenger, F. (2015). *The Politics of Micro-Decisions: Edward Snowden, Net Neutrality, and the Architectures of the Internet*. Lüneburg: meson press.
- Stallabrass, J. (2003). THE AESTHETICS OF NET.ART. *Qui Parle*, 14(1), 49-72.
- Stelter, B. (2011). *Debate Web Stream Does Not Flow Smoothly for All* [Blog post]. Retrieved from [http://thecaucus.blogs.nytimes.com/2011/10/11/debate-web-stream-does-not-flow-smoothly-for-all/?\\_r=1](http://thecaucus.blogs.nytimes.com/2011/10/11/debate-web-stream-does-not-flow-smoothly-for-all/?_r=1)
- Sterne, J. (2012). *MP3: The meaning of a format*. Durham, London: Duke

## Bibliography

- University Press.
- Stocker, G. (2003). CODE - the language of our time. In G. Stocker & C. Schöpf (Eds.), *Code: The language of our time* (pp. 10-14). Linx: Hatje Cantz.
- Strachey, C. (1954). The "Thinking" Machine. *Encounter*, 25-31.
- Sullivan, G. (2010). *Art practice as research : inquiry in the visual arts* (2. ed.). London: SAGE.
- Svensson, P. (2012). Beyond the Big Tent. In M. K. Gold (Ed.), *Debates in the digital humanities* (pp. xvi, 516 s.). Minneapolis: Univ Of Minnesota Press.
- Taboada, M. (2004). The Genre Structure of Bulletin Board Messages. *Text Technology*, 13(2), 55-82.
- Tanimoto, S. L. (1990). VIVA: A visual language for image processing. *Journal of Visual Languages and Computing*, 1(2), 127-139.
- Tanimoto, S. L. (2013). *A perspective on the evolution of live programming*. Proceedings of the 1st International Workshop on Live Programming, San Francisco, California: IEEE Press.
- Tate. (2014). BMW TATE LIVE: On Liveness: Pre/During/Post. Retrieved from <http://www.tate.org.uk/whats-on/tate-modern/talks-and-lectures/bmw-tate-live-on-liveness-preduringpost>
- Terranova, T. (2014). Red Stack Attack! Algorithms, Capital, and the Automation of the Common. In A. Avanesian & R. Mackay (Eds.), *#Accelerate# : The Accelerationist Reader*. Falmouth: Urbanomic.
- The Humanities and Critical Code Studies Lab. (n.d). About | The Humanities and Critical Code Studies Lab. Retrieved from [http://hacslab.com/?page\\_id=2](http://hacslab.com/?page_id=2)
- The MIT Press. (2016). Software Studies. Retrieved from <https://mitpress.mit.edu/books/series/software-studies>
- Tian, X., & Benkrid, K. (2009). *Mersenne Twister Random Number Generation on FPGA, CPU and GPU*. Paper presented at the Proceedings of the 2009 NASA/ESA Conference on Adaptive Hardware and Systems,
- Tumasjan, A., Sprenger, T. O., Sandner, P. G., & Welpe, I. M. (2010). Predicting Elections with Twitter: What 140 Characters Reveal about Political Sentiment. *Proceedings of the Fourth International AAI Conference on Weblogs and Social Media, Culture & Society*.
- Turing, A. M. (1937). On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1), 230-265.
- Tuya, J., Suárez-Cabal, M. J., & la Riva, C. d. (2007). Mutating database queries. *Information and Software Technology*, 49(4), 398-417.
- Twitter. (n.d). FAQ. Retrieved from <https://dev.twitter.com/faq/basics>
- Unsworth, J. (2002). What is Humanities Computing and What is not? Retrieved from <http://computerphilologie.uni-muenchen.de/jg02/unsworth.html>
- Vasulka, S., & Demeyer, T. (n.d). Image/ine - V2\_Institute for the Unstable Media. Retrieved from <http://v2.nl/archive/works/image-ine/?searchterm=real-time>
- von Neumann, J. (1945). *First Draft of a Report on the EDVAC* (Contract No. W-670-ORD-4926)[Report]. Retrieved from [http://www.wiley.com/legacy/wileychi/wang\\_archi/supp/appendix\\_a.pdf](http://www.wiley.com/legacy/wileychi/wang_archi/supp/appendix_a.pdf)
- Wardrip-Fruin, N. (2011). Digital Media Archaeology: Interpreting Computational Processes. In E. Huhtamo & J. Parikka (Eds.), *Media Archaeology: Approches, Applications, and Implications* (pp. 302-322). University of

- California Press.
- Watz, M. (2008). Re: Internet #007: Technological Mimesis. Retrieved from <http://cont3xt.net/blog/?p=253>
- Watz, M. (2010). Closed systems: Generative art and Software Abstraction. Retrieved from <http://mariuswatz.com/wp-content/uploads/2012/03/201005-Marius-Watz-Closed-Systems.pdf>
- Weaver, W. (1949). Recent Contributions to The Mathematical Theory of Communication. In C. E. Shannon & W. Weaver (Eds.), *The Mathematical Theory of Communication* (pp. 3-28). Urbana, Chicago: University of Illinois Press.
- Wegner, P. (1997). Why interaction is more powerful than algorithms. *Communications of the ACM*, 40(5), 80-91.
- Weizenbaum, J. (1966). Computational Linguistics. *Communications of the ACM*, 9(1), 36-45.
- Weltevrede, E., Helmond, A., & Gerlitz, C. (2014). The Politics of Real-time: A Device Perspective on Social Media Platforms and Search Engines. *Theory, Culture & Society*, 0(0), 1-26.
- Whitelaw, M. (2006). System stories and model worlds: A critical approach to generative art. In O. Goriunova (Ed.), *Readme 100: temporary software art factory* (pp. 135-154). Dortmund: Hartware-Medien-Kunst-Verein
- Williams, R. (1974). *Television : technology and cultural form*. London: Fontana, Collins.
- Windows 95. (2001). In *Wikipedia*. Retrieved from [https://en.wikipedia.org/wiki/Windows\\_95](https://en.wikipedia.org/wiki/Windows_95)
- Witmer, B. G., & Singer, M. J. (1998). Measuring Presence in Virtual Environments: A Presence Questionnaire. *Presence: Teleoperators and Virtual Environments*, 7(3), 225-240.
- XCEED. (2014). *Radiancescape* [Artwork]. Retrieved from <http://www.xceed.hk/work/radiancescape/>
- Yeaton, M. (2013). What is email spoofing all about? *MIT News*. Retrieved from <http://news.mit.edu/2013/email-spoofing-whats-it-all-about>
- YoHa, & Fuller, M. (2014). *Endless War* [Artwork]. Hong Kong: Connecting Space. Retrieved from <http://www.writingmachine-collective.net/wordpress/?p=489>
- Zaman, T. R., Herbrich, R., Van Gael, J., & Stern, D. (2010). *Predicting information spreading in twitter*. Computational Social Science and the Wisdom of Crowds Workshop (colocated with NIPS 2010) Citeseer. Retrieved from <https://www.microsoft.com/en-us/research/publication/predicting-information-spreading-in-twitter/>
- Zemmels, D. (2004). Liveness and Presence in Emerging Communication Technologies. Retrieved from <http://david.zemmels.net/scholarship/Comm7470.html>
- Zer-Aviv, M. (2004). Rhizome | The Web is a Living Organism. *Rhizome.org*. Retrieved from [http://rhizome.org/artbase/artwork/27784/?ref=search\\_title](http://rhizome.org/artbase/artwork/27784/?ref=search_title)



## Software (art) projects cited

- 0100101110101101.ORG & epidemic. (2001). *Biennale.py* [Software]. Retrieved from <http://epidemic.ws/biannual.html>
- Blas, Z and Cárdenas, M. (2012). *femme Disturbance Library* [Codeart].
- Brady, E., & Morris, C. (2003). *Whitespace* [Software]. Retrieve from <http://runme.org/project/+whitespace/>
- Cayley, J., & Howe, D. (2015). *Read for us...And show us the pictures* [Installation]. Retrieved from <http://thereadersproject.org/installations/sutp.html#sutp>
- Chernyshev, A. (2007). *Loading* [Sculpture].
- Chu, H. (1996). *The Sound of Market* [Installation].
- Chung, B. (2015). *50. Shades of Grey* [Installation]. Retrieved from <http://www.magicandlove.com/blog/artworks/50-shades-of-grey/>
- Electroboutique. (2005). *Lyric economy* [Installation]. Retrieved from <http://www.electroboutique.com/works/4>
- Grosser, B. (2012-). *Facebook Demetricator* [Software]. Retrieved from <http://bengrosser.com/projects/facebook-demetricator/>
- Hansen, M., & Rubin, B. (2000-2001). *Listening Post* [Installation].
- Harrison, E. (2009). *Toy Town* [Installation].
- Howe, D. C & Nissenbaum, H. (2006). *TrackMeNot* [Software]. Retrieved from <http://cs.nyu.edu/trackmenot/>
- I/O/D. (1997-1998). *The Web Stalker* [Software].
- Jeremijenko, N. (1995). *Live Wire* [Installation].
- JODI. (2008). *GEO GOO* [Website]. Retrieved from [geogoo.net](http://geogoo.net)
- Johnson, D. J. (2010). *Spam Heart* [Website/Software]. Retrieved from <http://www.glia.ca/2010/spamHeart/>
- Kzyzwiniski, M. (2010). *ee spamming*s [Electronic Literature]. Retrieved from <http://mkweb.bcgsc.ca/fun/eespammings/>
- Lai, C-S. (2003). *Instant* [Installation]. Retrieved from [https://www.facebook.com/ESLITE.PROJECTONE/photos/?tab=album&album\\_id=437623016346200](https://www.facebook.com/ESLITE.PROJECTONE/photos/?tab=album&album_id=437623016346200)
- Leegte, J.R. (2000). *Scrollbar Composition* [Installation]. Retrieved from <http://www.scrollbarcomposition.com/>
- Link, D. (2009). *LoveLetter\_1.0* [Installation].
- McLean, A. (2004). *Feedback.pl* [Software].
- Sollfrank, C. (1997). *Net.Art Generator* [Website/installation]. Retrieve from <http://nag.iap.de/>
- Thomson & Craighead. (2012). *A live portrait of Tim Berners-Lee (an early warning system)* [Installation]. Retrieved from <http://www.ucl.ac.uk/slade/slide/tbl.html>
- Maigret, N. (2014). *The Pirate Cinema* [Website/Installation/Performance]. Retrieved from <http://thepiratecinema.com/>
- Polak, E & Bekkum, I.V. (2015) *Technomourning* [Website/Video]. Retrieved from <http://www.250miles.net/techno-mourning/>
- Prophet, J and Selley, G. (1995). *Technosphere* [Installation].
- Ripkin, Rose & Schmidt, Loren. (2015). *Moth generator* [Bot]. Retrieved from <https://twitter.com/mothgenerator>
- Savičić, G. (2009) *Loading (The Beast 6:66/20:09)* [Installation]. Retrieved from <http://www.yugo.at/processing/archive/index.php?what=loading>



- Thayer, P. (2009-). *Microcodes* [Software]. Retrieve from <http://pallthayer.dyndns.org/microcodes/>
- Soon, W., & Pritchard, H. (2012-2016). *Thousand Questions* [Installation]. Retrieved from <http://siusoon.net/home/?p=900>
- Soon, W. (2014). *Hello Zombies* [Installation]. Retrieved from <http://siusoon.net/home/?p=1273>
- Soon, W. (2016). *The Spinning Wheel of Life* [Software/Installation].
- Strachey, C. (1952). *Loveletters* [Software/Print].
- UBERMORGEN., Ludovico, A & Cirio, P. (2005). *Google Will Eat itself* [Software/Print]. Retrieved from <http://www.gwei.org/index.php>
- Vasulka, S., & Demeyer, T. (1996-2001). *Image/ine* [Software].
- XCEED. (2014). *Radiancescape* [Installation]. Retrieved from <http://www.xceed.hk/work/radiancescape/>
- Zer-aviv, M. (2004). *www.is-a-living.org* [Installation].

