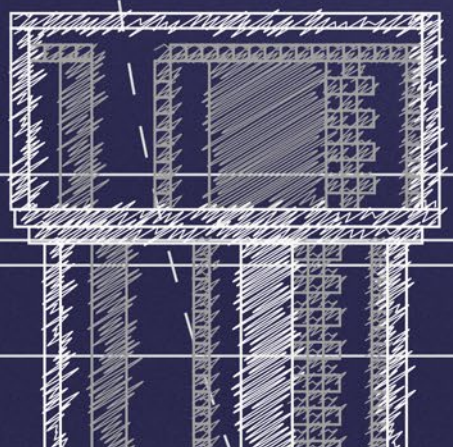
595.2pt

Codecraft

01010100011010100110000100101100001000000111011101
10000101110100001000000111001001100000101
10111001100100011011110110110100100
00001110100011001010110101011
01110011011101000001000
00011010000011010010
110010101110001
000101110

717pt

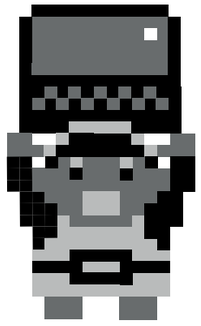
01000011011011111011001000110010101100011011100100110000101011001100



MA Thesis

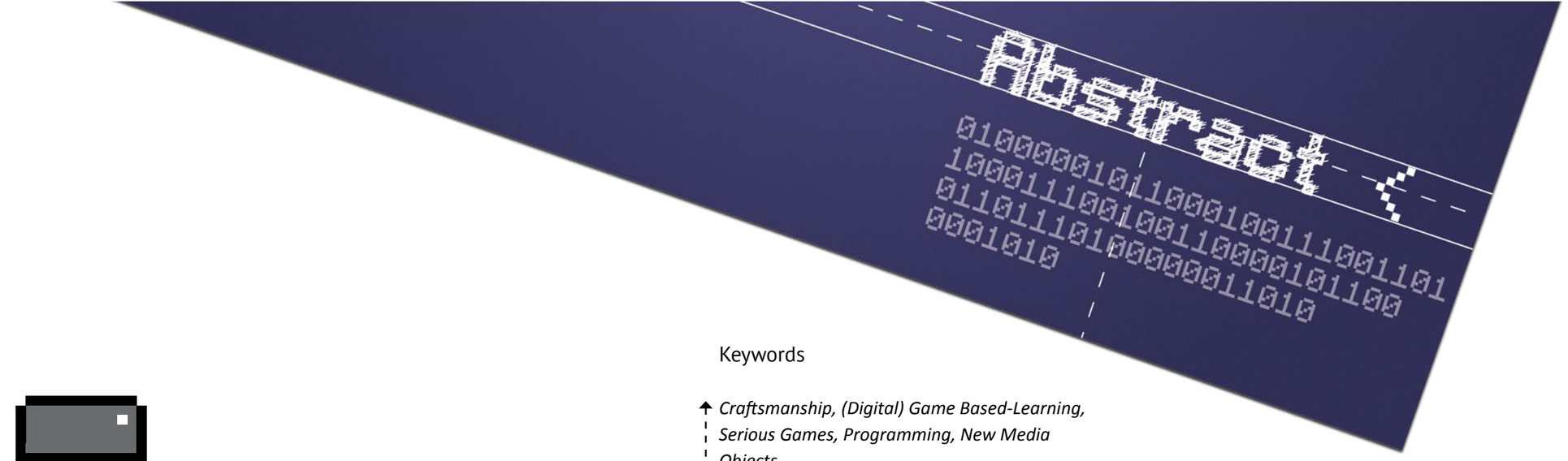
Esther Kool

595.2pt



Codecraft

MCMV10009
THE-Masterthesis/ MA NMDC
New Media & Digital Culture
Utrecht University
Esther Kool 3808858
Tutor: Dr. Michiel de Lange
Second reader: Dr. Mirko Tobias Schäfer
Hand in date: 11 February 2014



Keywords

Craftsmanship, (Digital) Game Based-Learning, Serious Games, Programming, New Media Objects

This paper discusses whether games are suitable to teach the craft of programming. This notion is two-fold: is programming a craft, and how can games help to teach a craft? By means of literature research and qualitative interviews, it is concluded that programming is indeed a craft, and that games are suitable to get someone started in the field. Though games are interesting to teach beginners their own learning style, they lack the possibility of transferring enough knowledge and skill to achieve mastery of a field, which is crucial to craft.

Dedication

01000100011001010110010001101
00101100011011000010111101
00011010010110111101
101110

A master thesis is quite a feat. Somehow, I've managed to spend my third summer in a row writing a thesis. First, one for the Utrecht University of Applied Sciences, and last year a final assignment to finish up my pre-master year in New Media & Digital Culture at Utrecht University. And now the time is finally there to present my magnum opus: the thesis that will end the time in my life where I can call myself a university student.

A very special thanks goes out to the people at SETUP, where the topic of this research originated. They also helped with its contents by thinking along and lending me books that were crucial. My interviewees Sergio van Pul, Peter Goes, Jan Willem Huisman, Evert Hoogendoorn, Dennis Rosenbaum and Douglas Rushkoff also deserve recognition for the time out of their schedule to help me understand the practice supporting the theory.

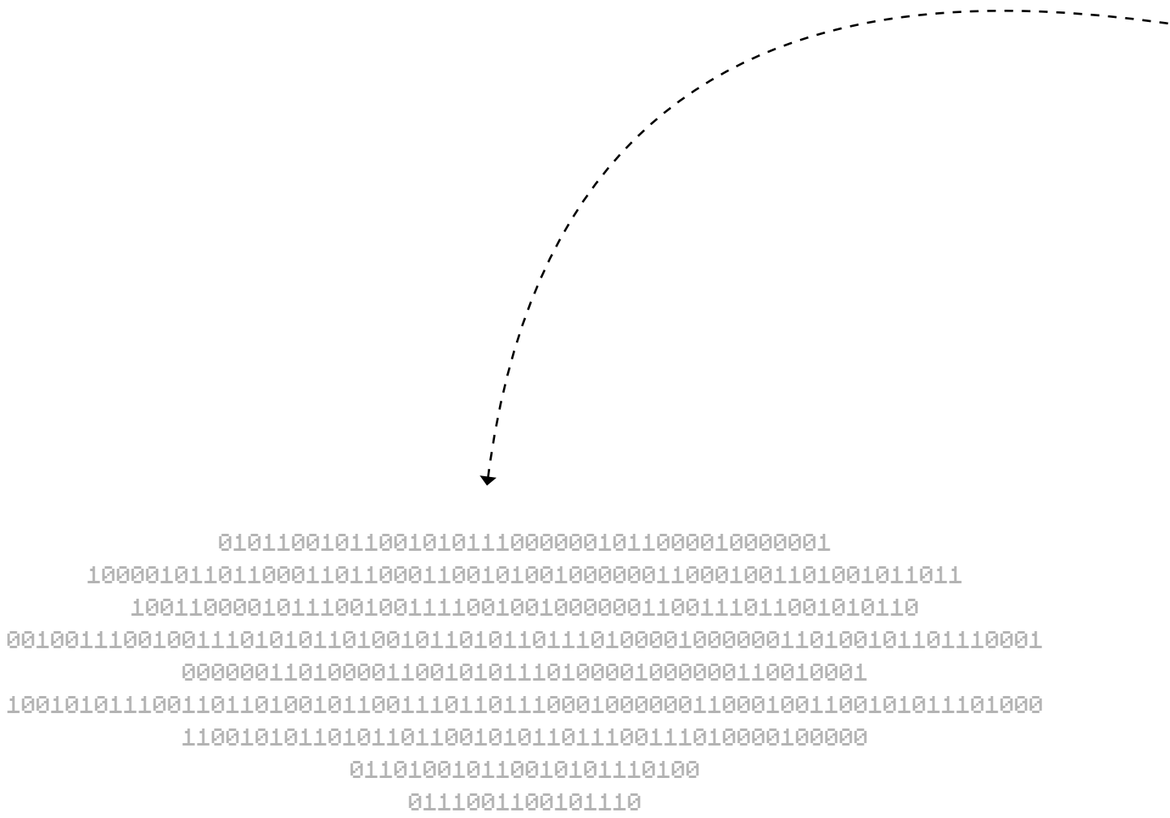
My eternal gratitude will also be aimed towards everyone who took the time to review my work. From my tutor Michiel de Lange and my second reader Mirko Tobias Schäfer to my peers at NMDC and my family members who corrected every apostrophe out of place. Especially my dad, who still believes you can learn cooking by pressing buttons on a microwave. On a more personal note, a thanks towards my 'To the rescue' crew for putting up with the infinite Whatsapp messages back and forth. Finally, a word of thanks for Jochem, who took me swimming when I wanted to kick everything related to craft, programming or games.

1190pt

Table of Contents

01010100011000010110001001101
10001100101001000000110111
10110011000100000001000
01101101111010110111
00111010001100
10101101110
0111010000
1110011

Table of Figures



Introduction

A quick search on Google tells me we are approaching the two billion mark in PC sales worldwide, this year (www.worldometers.info). However, it doesn't require Google to state that there are not two billion computer programmers. Societies worldwide are becoming increasingly dependent on computers, yet lack the skills to improve their modus operandi. Using computers is not a problem, controlling the manner in which they operate is. This is not a cry for global programming skills, but it is a plea to better understand what good programmers do and how they learn to do it.

Earlier this year I completed an internship at cultural media institution SETUP. They arranged a contest called Elegant Algorithms to underline the craftsmanship inherent to programming. Participants had to recreate Piet Mondrian's painting Victory Boogie Woogie in computer code. At SETUP it was believed that programmers are looked at with disdain. This can be recognised in usage of the term 'code monkey' for instance, implying that even monkeys could program computers. This is not so.

During my internship I investigated whether programming can indeed be dubbed a 'craft' and concluded it was so, provided we realise 'craftsmanship' has evolved. The relation of programming and craftsmanship is a complex one, so I felt my internship report should be expanded into my thesis. In the first chapter of this thesis I will discuss notions of craft and why the term should be refitted to current societal setting.

A craftsman distinguishes himself as such by producing quality objects with insight and expertise. Although I am already hinting at my working definition, craft and its many facets will be explored in more detail in chapter one. I will argue that learning to be a craftsman has changed, as has the notion of craft itself.

I decided to look at craft from three angles; (I) how one learns a craft, (II) how one performs a craft and (III) the objects that come forth out of the crafting process. By researching how these three areas come into play, the important bases needed to make a proper definition of modern (digital) craft are covered.

Following a dilemma set out by sociologist Richard Sennett in his book The Craftsman (2011, 81), I present the notion of craftsmanship in a contested space between man-made objects versus automated, machine-produced objects. When discussing working with- and the creation of- digital environments, this distinction becomes tricky. The (aspiring) craftsman works with a digital environment, where he or she starts the automated process to create new digital objects. By looking at the materiality of digital objects and at the way people learn to work with them, I aim to present a concept of the craftsman of programming.

This research will help solidify the importance of the area of computer programming in contemporary society and pull the programming field out of the prejudiced abstract. Computer programmers are not all code monkeys, nor are they high priests of a magical digital order. They are creators and modern craftsmen. In this thesis I will not only further investigate how programming and craftsmanship are interrelated, but also how one becomes a craftsman programmer.

Computer programming, or coding, entails translating human wishes to a machine-regulated environment. There are differences in programming languages and there are differences in the skill-level of each programmer. So programming is a skill that is taught. Investigating how this is done will shine some light onto programming as a field of expertise.

It is interesting to note that video-games and computer programming developed side by side. Programming is taught through games on several platforms. Because technology and games are pervading society at so many levels, some argue that video-games are becoming crucial to learning (Prensky 2005, 97; Schaffer & Gee 2005, 12; McGonigal 2011, 7). Because of this, I am going to discuss learning to be- and being- a craftsman computer programmer, specifically how to do so using games- and gamified learning trajectories.

Games for education come in many forms, from pure entertainment games that are used in the classroom to full-fledged serious games that are designed with educational goals in mind. In chapter three I will discuss why games are considered for education in the first place, as well as the different forms they come in.

Digital video-games, like programming, take place in a simulated environment. Digital video-games are also built by programmers themselves. The gaming industry is an industry that actually relies on outside programmers to do tinker with their products (modding). Game industry professionals will often be found to hire or finance these outside programmers to improve upon their products (Nieborg 2005, n.p.). These overlaps cause for an interesting choice of education in the context of programming. In chapter three and four I will refer to the process of learning to be a craftsmen programmer, and how games may--or may not contribute to this process.

Research questions

To guide this research, a research question was defined:

How can games teach one to become a craftsman of programming?

Two main themes can be delineated from that question, i.e. programming as craft and whether one can reach this state by playing (video) games. Important sub questions are:

- What defines craft?
- How do craft and technology relate?
- Is programming a craft?
- Why and how are games applied in learning?
- Can games teach a craft?

To answer these questions two methods of research are used: literature research and interviews.

Literature Research

Both programming and games for education are hot topics in current academic debate. To define where this thesis fits into that debate, a literature research was conducted. Although an explorative literature research was applied at the start of this research, during the course of reading and writing a more focussed approach was applied, based on the descriptions of social scientists William Gibson and Andrew Brown (2009, 38-42).

By casting a wide net, explorative research allowed me to find general research methods and questions posed in academic literature and secondary sources. Topics included new ways of working, specific ways of computer programming or particular cases of gamification (to name a few). This method doesn't provide the level of in-depth knowledge needed to discuss particular areas of this research, though (Gibson & Brown 2009, 40).

The focussed literature research came into play after the general orientation, which provided this in-depth knowledge and refined the original framework and research questions. Focussed literature research means reviewing literature in the same empiric and theoretical domains and includes literature asking similar questions (Gibson and Brown 2009, 41).

The theoretical framework in this thesis concerns three areas, namely craft, computer programming, and (serious) games for education. By focussing on craft, programming and games, the area of education is somewhat left behind. Deeper understanding of didactics would provide an interesting angle, but this was impossible to include at this time given the necessary focus within the format of this master thesis. For instance by looking at the persuasive nature of games in the manner that Ian Bogost suggests (see chapter three), I leave no room for other possible didactic approaches that could be taken in game-form.

My starting point was the book *The Craftsman* by Sennett (2008), which is the most elaborate contemporary work on craftsmanship available. By analysing Sennett's work on practices and attitudes prevalent in craft, I was able to do a comparative literary study on how these practices and attitudes are possibly mirrored within programming culture. Comparative literary study entails cross-cultural and interdisciplinary study of literature, says literature- and cultural professor Tötösy de Zepetnek (1998, 13). Although most of the literature in my framework originated in the social sciences, they did so with wildly varying ontologies.

While looking at programming culture, I focussed on literature that has the social and historical impact of programming and/or software as its ontology (e.g. Knuth (2007), Manovich (2001), Rosenberg (2007), Montfort et al (2012), et cetera), as opposed to literature that dictate best practices within the field of industry. Although my time working at SETUP allowed me to build my own knowledge of programming culture, I am not a programmer myself and this literature would have little value.

Besides, part of learning a skill (such as programming) requires internalising certain practices, creating tacit knowledge (Sennett 2008, 50). Although this tacit knowledge is quite useful when producing an object, it also means the one doing so may have trouble explaining why or how they are doing it. By looking at

programming and software as a cultural product as opposed to looking at it from a very practical perspective that blind spot is avoided.
Finally: software production is renowned for its troublesome planning and organisation (Rosenberg 2007, n.p.). Although some strides have been made to create methodologies for software projects (which I will discuss in chapter 2), analysing programming as a craft in a broader social context could shed some light on the root of these problematic issues.

By looking at both older and newer literature, conclusions can be reached that might be relevant at a future time. Especially when looking at topics such as computer programming and digital (video) games, it can be easy to forget that these topics have been under discussion for decades. For example: by combining the thoughts of computer programmers in the 1970's with those of today's computer programmers, the development of the field of programming is illustrated.

The third area of the literature research concerns learning through video-games. I started out by reading works by renowned authors in the field, i.e. Mark Prensky, Jane McGonigal, James Paul Gee and David Schaffer. These authors are all very enthusiastic about using games in educational settings. To uncover why, I address older, classic works on the role of play and games within culture, such as historian Johan Huizinga's *Homo Ludens* (1949) and Roger Callois' *Man, Play and Games* (1961).

Although the literature research provides us with a thorough understanding of craft, programming and games, as they are currently being discussed in academic circles, the theory needs to relate to the practice. With the exception of McGonigal, none of the authors mentioned above, whose ontology is games for education, put their hands in the dirt and created video-games themselves. To further highlight the aspects of video-games that could have an impact on behaviour (and thus learning), I refer to *Persuasive Gaming* (2011) by game designer and theorist Ian Bogost. His experience with game production gives him a very practical perspective of what it is game designers want to achieve and how their rhetoric influences the player.

This thesis is not the first to tie computer programming and gaming techniques together, so two cases that use gaming to teach programming are presented as examples. To complement the literature research and examples, I employ qualitative interviews with programmers and game designers, thus further strengthening the link between the academic theory and the current industry practices.

Interviews

The application of game-based learning in the context of programming will be based on several qualitative interviews. I decided on interviews because they provide an in-depth set of qualitative data. This way we can test whether programmers and game designers agree with the theory outlined in the theoretical framework.

Psychologist Svend Brinkmann states that interviews are a key research method in the social sciences, as conversation has been used as means to elicit knowledge for as long as we have known (2013, 1). Interviews should be carefully designed, however, as conducting interviews unprepared could mean ending up with a mountain of data that is almost impossible to analyse (Brinkmann 2013, 4).

By completing the literature research first, it was possible to format the interviews so the interviewees could substantiate or falsify claims made by academics. For example: Kurt Squire stated that the entertainment game *Civilization* could be used to teach history, yet it proved to be more difficult than traditional history lessons for some students (2005, n.p.). Based on this statement, the game designers were asked whether they thought entertainment games, serious games or gamification of education would be effective methods to teach. By combining the literature research and the empirical data from the interviews, a well-rounded conclusion can be made about the ability of games to teach the craft of programming.

By using interviewing as a method, a lot of data was retrieved. A challenge is that choices have to be made with regards to the relevance of certain statements in favour of others. Although I have tried to select the data most relevant to this research, another researcher might have chosen differently. Nevertheless, I have related the interviews to the literature research in the most productive way possible. The original transcripts can be found in the appendix.

Three interview templates were used, one aimed at programmers, one aimed at serious game designers and one specifically tailored (but based on the programming template) to Douglas Rushkoff, who is a media theorist and advisor at *Code Academy*. The questions were open-ended to ensure descriptive and lengthy responses.

The questions were peer reviewed by three fellow New Media & Digital Culture master students on relevance and wording beforehand. The questions were arranged according to certain themes. The programmer interviews centred around three general areas: (I) learning how to program, (II) process and (III) objects of programming, mirroring the manner in which I analysed craft in the literature research. The game interviews were roughly divided into two sections, i.e. how serious games can help someone learn and the process of making a serious games. This way both the game designer's reflection on the academic discourse and their practical approach to making these games come to light.

Due to the nature of their occupation, a general interviewing style was applied (based on the description of physician Daniel Turner (2010, 755)). This means the questions for both domains were prepared beforehand, but the interviewer did not stick to the exact order or formulation of each question, to ensure an in-depth situated interview for each interviewee. This method of interviewing means a possible lack of consistency in interviewee answers, but it also means questions could be formulated to better suit the interviewees' story and provided the opportunity for follow-up questions (Turner 2010, 755). Even though a general interviewing style was applied, the designed templates ensured the same general areas of information were discussed (Turner 2010, 755). Because the interviews were qualitative rather than quantitative and there is a wide range in experience and practical application of each interviewee, this style is best suited to generate the information needed.

The interviewed individuals are: Sergio van Pul, a programmer and former employee of *Scratchweb* (a web based platform designed to teach children computer programming), Dennis Rosenbaum, a computer programmer at *Winvision*, Douglas Rushkoff (as stated before), Peter Goes, a serious game developer and programmer at *Qlvr*, and finally Jan-Willem Huisman and Evert Hoogendoorn, serious game developers at *IJsfontein*. The interviewees were selected on the basis of their experience with computer programming and/or serious games. All the interviewed individuals are male.

The inclusion of female programmers or game designers would have perhaps provided a different data-set, yet unfortunately no female interviewees were available. No gender-related issues were discussed in the interviews, so the influence on the data should be minimal. The small sample size and the fact that all but one of the interviewees are from the Netherlands means that generalisations should be approached with care. The nature of their respective experiences means they have great insights into the questions posed by this research, however, so I consider the analysis of the interviews to be fruitful.

The interviews took place at a location of the interviewees choosing. Most interviews took place face to face, with the exception of Douglas Rushkoff who preferred to answer questions via e-mail. The audio of each interview was recorded and transcribed by the author. The interviews were conducted in Dutch with the exception of the interview with Rushkoff, which was written in English. All the spoken interviews took between approximately thirty and forty-five minutes. All interviewees were informed about the subject of this study. Their permission was asked to be recorded and transcribed. As a compensation for their help they will receive a digital copy of this thesis.

By means of the literature research and the interviews, a fairly robust conclusion can be reached. By applying the outlined methodology, the research will deliver a picture of the academic discourse surrounding programming craftsmanship and game-based learning. It will also validate whether this discourse is in congruence with current industry practice. I hypothesise that it will, because most of the academics who write about programming and/or games are actively involved in those fields themselves.

Following this line of research will provide a better picture of what does- or does not work in the specific context of craft, programming and games; a topic that has not been covered at this time. After answering the main research question, I hope to contribute to the understanding of games-based learning. I also aim to improve the understanding of the value of computer-programming and the arguably abstract notion of craftsmanship in contemporary society. Finally, an understanding of how one learns computer programming effectively (through games) will contribute to training of this increasingly important industry.

One cannot discuss whether games can teach someone the craft of programming without grasping what a craft is, exactly. This chapter will be devoted to understanding the concept of craft itself, in doing so leaning on Sennett's *The Craftsman* (2008).

A peculiar tendency struck me during the past few months. Whenever I would discuss first my internship research and later my thesis, I would ask people about their thoughts on the notion 'craft'. Without fail, everyone would mention carpentry and/or furniture builders. Something about a craftsman feeling his way across the grain of a particular type of wood and turning that wood into a utensil or piece of furniture has struck a collective chord and allowed us to associate craftsmanship with that specific field of expertise.

Another often-mentioned element is that of the guild, by which we take the craftsman out of the 21st century and back to the Middle Ages. Funnily enough, when I would ask the same people whether they thought a computer programmer was a craftsman, nearly all of them replied positively. When asked how this relates to the materiality of the craft in carpentry with which they were so enamoured, they were stumped. The fact that guilds are now mostly associated with role-playing games (though they do still exist in the context of craftsmanship) was not seen as a reason to dismiss programming as craft. So without the guilds, and looking beyond the sanded down borders of materiality, what does a modern notion of craft entail? This chapter will be devoted to answering that question.

In 2008, Sennett wrote The Craftsman. In this book he explores the notion of craft from a variety of perspectives and fields. These perspectives and fields are not ordered chronologically. Instead, Sennett divides his book into three parts: craftsmen, craft and craftsmanship. The three respective areas pertain to the manner in which the craftsman is situated into the world, the way in which he or she works and characteristics that set him or her apart from other workers. I will also look at craft from three perspectives. How one learns a craft, how it is practiced and the objects that come forth out of the crafting process. Sennett's work will serve as a guideline along the way.

1.1. Learning a craft

In the olden days, a craftsman would start out as an apprentice, work his way up to an assistant and would finally take on the role of the master. The work took place in the workshop. Often, the craftsman and his apprentices and/or assistants would live in that same workshop. There was a type of impersonality to behold in these workings, something that can still be recognised in the digital 'workshop' of Linux programmers, says Sennett. "In Linux online workshops, it's impossible to deduce, for instance, whether 'aristotle@mit.edu' is a man or a woman; what matters is what 'aristotle@mit.edu' contributes to the discussion" (Sennett 2008, 26-27).

So something of the old workshop system is still alive in the digital era, we might deduce. There are still novices and masters. This can be recognised in the Dutch hiring system where programmers are classified on their experience level, thereby falling into the 'junior', 'medior', or 'senior' level. There is more to craft than a ranking system, however.

Computer scientist C.A.R. Hoare discusses craftsmanship by referring to the practicality of their knowledge. Craftsmen learn by imitation, practice, and trial and error, often without a theoretical understanding of their field (Hoare 1984, 5). This causes them to be ill-equipped to explain what it is they are doing, exactly. Sennett touches upon the same subject, yet traces the cause not exactly to a lack of theoretical understanding, but to tacit knowledge.

Sennett discusses the creation of the first encyclopaedia by French philosopher Denis Diderot. When Diderot was researching its contents, Diderot would often find the workers he spoke with unable to explain what they did, or how they did it (Sennett 2008, 94). This does not suggest stupidity, on the contrary. The skills of the craftsman surpass their ability for expressing it in language (Ibid. 95). A theoretical understanding of something and tacit knowledge of the activities related to that same thing do not exclude each other. For example: someone can explain the historic origin of the French language, yet not be able to transfer their innate ability to speak it fluently without an accent.

This tacit knowledge of a trade comes to being through, perhaps, talent, but mostly through practice. To become a true master at something, Sennett cites the often mentioned ten thousand hours of practice rule (Sennett 2008, 20). One of the interesting things inherent to craft is the fact that the craftsman can practice the same thing over and over, even when it seems like a boring activity to the outsider.

Being completely consumed by a task is something which psychologist Mihaly Csikszentmihalyi described as flow (Nakamura and Csikszentmihalyi 2002, 89). Prerequisites to flow are experiencing perceived challenges that stretch existing skill, and clear and immediate feedback about the progress being made (Ibidem). So flow can only be achieved when someone is practicing something.

Daniel Pink (an author who mostly focusses on business, work and management) describes how flow is not enough to generate mastery, true mastery requires years of effort, whereas the state of flow can be achieved in an hour (Pink 2009, 200). That does not mean that flow vanishes in the hands of masters. True mastery is as an asymptote: you can get close to it, but you can never quite reach perfection (Pink 2009, 212).

All of this practice starts by refining skill. Skill, regardless of the level of abstraction, starts as a bodily practice, says Sennett (2008, 10). It should already serve as a clue that the word ‘practice’ and ‘practical’ share the same word root, praktikos in Greek. A word used in contrast to that gnostikos, the intellectual. So the practice of an art takes root in action and application.

Sennett describes physical sensation in relation to understanding. He refers to the saying ‘to grasp something’, generally interpreted as understanding something, stating this is directly related to implicit knowledge directed by touch (Sennett 2008, 153). The verb ‘to grasp’ also means to physically take something or hold something with your hand. Your body reacts with anticipation. This double meaning is not a coincidence, says Sennett. To grasp something mentally is also to anticipate meaning without having all the knowledge at hand (Ibid. 2008, 164).

To illustrate this Sennett discusses playing the piano. By practicing the same piece over and over, the brain gets more adept at getting the fingers to press the right keys, as do the hands. This is of course troublesome when we discuss programming as a craft, as the only tactile sensation will probably be found in the tapping of the keyboard and perhaps swiping the tablet surface.

Programming does entail more than just the ability to type blindly, though. Bodily practice in computer programming includes posture, positioning the screen and equipment, and knowing where to click and type within the computer’s interface without having to mull that over. More importantly, though, in programming this tendency manifests itself in the knowledge of code. This will be discussed in chapter two: Programming.

1.2. Practicing a craft

Once someone becomes proficient at their work, they can start to practice their craft in exchange for monetary reward. Or at least, that is what many have in mind when they think of a professional: someone who can make a living out of their skill. Pink discusses the motivation to do things in his book *Drive*. Paradoxically, doing something for the sake of doing it, is generally a better motivation then when one is offered compensation for their efforts, says Pink (2009, 23). Though extrinsic motivators like a hefty pay check and bonuses work in some small instances, most creative and cognitive work is done when the worker is offered intrinsic motivations. Do it because you love doing it, because you think it matters.

So proper motivation is indeed quite important when someone is trying to master a craft. This is where the first similarity between learning a craft and learning computer programming becomes clear. American journalist Scott Rosenberg spent several years with the development team of personal information manager *Chandler* and wrote about its creation in the book *Dreaming in Code: Two Dozen Programmers, Three Years, 4,732 Bugs, and One Quest for Transcendent Software* (2007). In it, he describes how professional programmers often find themselves being paid to do something they would have done anyway, just for the fun of it (2007, n.p). While interviewing Dennis Rosenbaum, I asked him how he got started. He replied: “Fun. I had fun. I wanted to build a website and got sucked in.”

According to Pink, the three elements one truly needs to be motivated to do good work are (I) Autonomy; (II) Mastery and (III) Purpose. The worker must feel a sense of control over their own situation, they must always work on their skill (yet accept that they will never reach perfection) and a greater sense of purpose for the task will also greatly increase motivation.

Especially the autonomy is important for craftsmen like software developers and designers, says Pink (2009, 142). The drive to keep working on the same practice comes from the human tendency to explore and to learn. Though this tendency is fragile and hard to nourish, it often yields the best results (Pink 2009, 23).

When a craftsman has achieved his or her autonomy by becoming proficient enough to no longer need a master to perform the job, when they have achieved mastery in their field, a craftsman will become proficient at developing his trade. With skill comes technical understanding, which develops through imagination, says Sennett (2008, 10). A craftsman will always show three foundational traits, he argues: “the ability to localize, to question and to open up” (Sennett 2008, 277). This means he or she will determine what materials they are working with, question its potential and finally they will expand upon its current form. Finding new applications for existing tools, that is what matters. The quality of the eventual object is the main priority.

1.3. Objects of craft

Architects build houses, cobblers mend shoes, carpenters build furniture and sheds, and computer programmers build structures that organise and store information. Coming back to the vision of the carpenter, Sennett states that: “Craftsmanship is poorly understood ... when it is equated only with the manual skill of the carpenter’s sort” (Sennett 2008, 20).

Indeed, in this paper I am not concerned with the more restrictive term of ‘handicraft’ in which all objects are created completely by hand. However, industrial automation is the other end of the spectrum. It would be wise to discuss where computer programming is placed on that scale from handicraft to automation.

Of course, when technology is introduced to the work floor, certain professions will have to adapt or even fall away. It is like German philosopher Martin Heidegger said: “The will to mastery becomes all the more urgent the more technology threatens to slip from human control” (Heidegger 1977, 5). Being a true master of a craft means no machine can do your work in your stead. Well, at least, that is what some believe. Because programming is so entwined with technology, let’s take a side step and discuss the relationship between craft and machinery.

1.3.1. Craft & the machine

Sennett refers to Victorian writer, thinker and art critic John Ruskin as the icon for the separation between human-made object (an earthenware jar) and mechanically produced object (a mechanically woven rug). Ruskin would only find beauty in objects with a human element and would stand firmly opposed towards the use of mechanical processes (Sennett 2008, 108). Although men like Ruskin would only praise natural, analogue objects as the product of true craft, Sennett also sees the work of craftsmen in the creation of digital objects (Ibid. 24).

Sennett is quick to proclaim that programmers of kernel Linux are *demioergoi*, which literally translates to ‘public worker’. About these programmers he states that their main objective is: “[A]chieving quality, doing good work, which is the craftsman’s primordial mark of identity” (Sennett 2008, 25), thereby elevating

their status from mere workers to craftsmen. Linux is an open source system, which means that anyone who can program is allowed to contribute to the entire system by writing or adapting its code. By finding bugs and fixing them, the programmers think of cunning ways to repair problems in the code.

The extent to which a machine undermines or supports craft is thus a grey area. Whether something is a complete mechanical production or whether we speak of a tool that is applied by a competent craftsman is something yet to be determined. Perhaps the difference can be found in the object’s agency.

Donald A. Norman, an academic in the fields of cognitive science and usability engineering, discusses agency in his book *The Design of Everyday Things*. Objects have their own psychology, he says. They are formed in such a way that its user is constrained in the way in which he can use them. A handle can be pulled, a chair can be sat on, a ball can be thrown or rolled, et cetera (Norman 1988, 12-13). That same handle cannot be swung, the chair cannot be used to lie on and the ball cannot be used to balance a stack of books. Their mappings and constraints create the objects’ affordance.

The nuance of affordance is something that is missing in Sennett’s work. Although he hails Linux programmers as demioergoi, he also says CAD (computer-aided design) is detrimental to human devotion, immersion and insight (Sennett 2008, 40). Linux shows us problems, CAD hides them (Ibid. 43). As long as the worker programs to build a computer system, and his work stays digital, Sennett perceives it to be craft, but as soon as CAD is applied in architecture, for instance, the architect loses his immersion and devotion and builds a lesser product. A strange turn of events, but only one that occurs in the manner in which the computer is applied in the process of creating an end product. Though Sennett arrives at a conclusion, he does so by skipping the why of it.

When looking at computers, it is thus good to take into account what the computer is used for, in order to decide whether it is an attribute to the process or detrimental to it. In the case of the Linux programmers, the computer is used in order to build something in computer code, something that stays digital. In the case of the architects, the computer is used to replace a method that could otherwise have stayed analogue. In this case the analogue version (the drawing of blueprints) has the added benefit of creating insight for the architect.

But before we conclude that a computer is only useful when the work that is done on it results in a digital construct, consider the following: the programmer does not necessarily need to program something that stays digital in order to best make use of the computers affordance (think of 3D printing, for instance). The fact that the computer in the setting of the architect is used to replace an already tested and approved method is more revealing. Considering how the computer is used, and what it is used for, is crucial to the understanding of the role of automation in the process.

Just like the craftsman is envisioned as a carpenter, the fear of human techniques being replaced by machinery is also a collective human vision. Heidegger has famously argued against modern techniques. (Heidegger stated that to view technology as mere instruments is not understanding technology at all.

Instead of machinery, we should look at technology's essence (1977, 32)). Through modern techniques, forces are violently claimed instead of letting things be (Ibid. 9); their natural *dasein*. Another aspect of modern technique is that it relies on itself (Ibid. 27). For instance, machines can only be turned on if they are connected to the electrical system. About Heidegger and machinery, Sennett states:

> Even Martin Heidegger eventually installed electricity and modern plumbing in his Black Forest hut. What Enlightenment writers worried more about was the machine's productive side, its influence on the experience of making—and these worries remain. (Sennett 2008, 83).

The fear of machinery stems from a sensation of domination by the machine, argues Sennett. Since the 18th century, when Vaucanson invented the automatic loom, a problematic relationship between man and machine arose (Sennett 2008, 87). The machine executed tasks formerly done by man, often better than man did.

Sennett describes that there is a sense of seduction: "The seduction of CAD lies in its speed, the fact that it never tires, and indeed in the reality that its capacities to compute are superior to those of anyone working out a drawing by hand" (Sennett 2008, 81). This shift from human work to complete automation is easily compared to the question when craft begins and when it ends.

Sennett himself does not answer this question well. According to him we still struggle with the abilities of the human craftsman and the invasion of the machine. Because machines often have higher production value, they are introduced to the production process without taking the craftsman into account. This way, a sense of submission to the machine starts to creep in when technological 'progress' is announced (Sennett 2008, 108).

The only positive note in this human-machine relationship is that we learn to recognise our own limitations, says Sennett (Sennett 2008, 84). By knowing our limitations, we can revel in leaving our shortcomings to automated processes (Ibid. 99). An enlightened craftsman will thus find pleasure in his or her shortcomings and will see value in the markings that human work leave behind, in contrast to mechanically produced objects (Ibid. 104).

Exclusively looking at craft in a comparison of human failure as opposed to mechanically produced perfection is problematic to me. Not only will a machine not always produce perfect objects (besides human errors in entering the code, things can also go wrong mechanically), but it also ignores the human aspect in controlling the machine.

Another way to look at computers is in the aid of 21th century craft. As we are changing into an information society, the need to restructure the manner in which we retrieve and consume that information becomes more important. The computer becomes a tool with which the programmer can design his architectural blueprint for the flow of information (Hoare 1984, 16).

Though Sennett affirms that the objects that come out of craft are worthy of regard themselves (Sennett 2008, 7), the tricky thing about objects of programming is that the eventual objects are all that users understand.

They understand how to use the software that's installed on their computer (mostly), but the way it came to be is a different matter. The way this software is built, and the technical details of digital objects that can be seen when you software up, are like gibberish to the average user.

Media theorist Douglash Ruskoff argues that everyone should learn how to code, because those who know how to use the tools of a technology will be the ones who decide how society is formed. Marshall McLuhan, a Canadian media philosopher, was best known for his phrase "the medium is the message" (McLuhan 1994, 7). With this phrase he sought to explain that the content of a particular medium is of no consequence. It is the influence the form of the medium has on society we should take into account. According to McLuhan, technology changes our very surroundings, much like the railway shaped our landscape and the light bulb changed our perception of day and night.

Rushkoff takes a similar stance towards programming. In fact, his book is called *Program or be Programmed: Ten Commandments for the Digital Age* (2010). In it he states that only the ones who know how to create or adapt software will be free of its constraints. "With the advent of a new medium, the status quo not only comes under scrutiny; it is revised and rewritten by those who have gained new access to the tools of its creation" (Rushkoff 2010, 12).

Defining programming as a craft would contribute to its understanding as a legitimate and even crucial profession. This is about more than legitimisation, because it would push new media discussions into ones of quality and application, rather than just re-affirming its existence. This discussion will be saved for the next chapter. First I would like to define craftsmanship as it is understood in the rest of this thesis.

1.4. Defining craft

This thesis will operate on the following provisional framework of what craft is, based on the earlier theory: craft is all about creation. Human intervention that causes certain materials to be transformed into objects of use. Hereby the quality of the eventual object is always the main focus of the craftsman.

Though they may practice their craft as such, craftsmen are never alone. To learn how to be one, a person has to accept a position of inferiority (apprentice/junior) towards those already proficient in a craft. Talent in a respective field, though beneficial, is not crucial on the road towards craftsmanship. In observing the craft as it is executed by one's superiors, by imitation, and by trial and error, the apprentice can work his way up towards mastery over his/her field. Practice is the key word. By practicing the art of creation over and over, a person can ingrain his- or herself with the tacit knowledge needed to create on a craftsmanship's level.

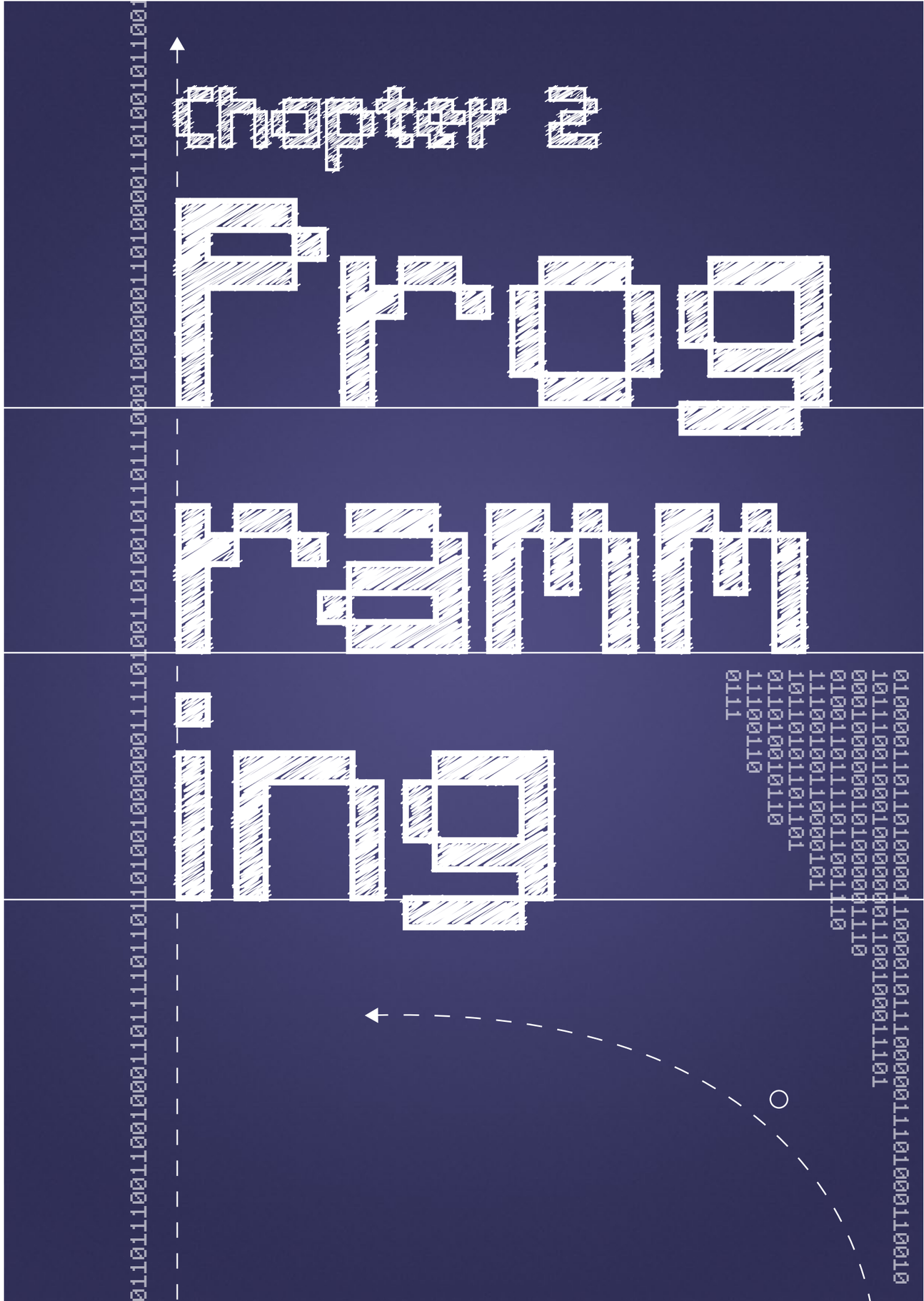
A craftsman will never reach perfection. Neither in his or her mastery of the art of creation, nor in the objects he or she produces. A true craftsman will always try to improve both his or her capabilities and his or her creation. Once a particular activity in the process of creation has been perfected, the craftsman will try to innovate, thus always working towards perfection but never reaching it. This is the shift from a mere clerical activity (medior) towards an activity of

insight and creativity (senior). Though the clerical aspects of craft will always remain and are even appreciated when it gets the craftsman into the flow, mastery means moving beyond it.

Finally: though automation is most definitely not part of craft, the complete dismissal of technology in craft is not productive either. Machinery, though often thought to replace human creation, is used to supplement it. The functions of computers in certain fields and occupations are complicated enough to warrant practice of use in themselves. Computer programming is one such field. Brian Hayes, a Senior Writer for *American Scientist* refers to this sentiment by mentioning a movement within programming culture called the pattern movement:

> The pattern movement ... suggests instead that programmers are like carpenters or stonemasons—stewards of a body of knowledge gained by experience and passed along by tradition and apprenticeship. This is a movement of practitioners, not academics. Pattern advocates express particular contempt for the notion that programming might someday be taken over entirely by the computer. Automating a craft, they argue, is not only infeasible but also undesirable. (Hayes 2003, 110).

Although not everyone agrees that programming is indeed a craft, the next chapter will highlight the possible similarities or discrepancies between craft and programming. I will demonstrate what programming is, what objects come out of the process and how one learns to be a programmer.



2.1. The act of programming

Now that we've established a framework of what being a craftsman in digital times entails, it is time to illustrate the craftsman programmer. Media theorist Adrian Mackenzie describes the act of programming as '[E]diting text according to rules expressed syntactically by a programming language' (Mackenzie 2003, 8). Stripped down to its most bare form, a computer programmer translates from human language to machine language. This process is called 'programming' or 'coding'. "As a developer, I create programs. I create websites, web applications, software that runs on machines. Basically, that's it: I create, I build" (Rosenbaum). Writing computer code can be done in several layers of abstraction, from directly operating on the hardware like the first 'computers' did (in this instance, the humans operating the machine were called computers) to the higher programming languages most often used today (Chun 2004, 29 - 30).

According media- and cultural academic Wendy Chun, the first computers were people, mostly women, who set machines (no more than calculators) to make calculations in certain orders. These days, the process is not too different, in essence. Fellow media- and cultural academic Lev Manovich states that all new media objects, whether created on a computer or converted from analogue sources, are numeric representations (Manovich 2001, 27).

The first computers did not have software. These were machines that consisted solely of hardware. They had to be set by hand by operators. Slowly, people began to find ways to set the machine to do iterations of the same calculation, based on instructions. These instructions took the form of mathematic algorithms. This developed into what we now refer to as software; operating the computer through so-called higher programming languages (Chun 2004, 29-30). Higher programming languages are instructions to the computer to set operations in the hardware into motion: a chain reaction of linguistic instructions by man to mechanical instructions within the machine.

This means that the physicality of computer programming has greatly reduced over the years. Does this also mean that programmers do not need to 'grasp' their field by practice, as Sennett stated? I asked Van Pul and Rosenbaum about their process, to uncover the physicality of their respective jobs. They both started speaking in the abstract: the conceptual process inherent to the programming process. Another question was needed to ask them how they physically work: their location, their actions. I wanted to relate this to the image of the workshop: the location where the craftsman of old lived and worked and trained his apprentices.

For Rosenbaum, the location where he works is inconsequential. Whether he works at home, at the client's office or on the train if need be, it does not matter. All that matters is whether he has his computer at hand. Van Pul mostly works out of his own home, provided he doesn't get distracted there. The only relation Van Pul could find in programming and physicality was that frustrating code could make him start cursing. Perhaps no true correlation can be found besides perhaps stating: both craftsmen and programmers can work from their home.

A sculptor molds clay, a programmer molds code. Computers are often described as sources of transparency, because of their ability to show us so much information. One important part is forgotten, argues Chun. A computer does not show us a document or an image. A computer shows us the result of a calculation that describes the document or image (Chun 2004, 27). The area of expertise of programmers is causing these calculations (notice a nuance between doing the calculations and causing them). Mathematician Ionica Smeets described it as cooking during a *Pauw & Witteman* airing devoted to the *Elegant Algorithms* project: the programmer writes the recipe and the computer bakes the pie (Smeets 2013). Who or what creates the end result, is hard to determine.

Last year, Nick Montfort, Patsy Baudoin, John Bell, Ian Bogost, Jeremy Douglass, Mark C. Marino, Michael Mateas, Casey Reas, Mark Sample, and Noah Vawter (media theorists, programmers and game developers alike) wrote *10 PRINT CHR$(205.5+RND(1)); : GOTO 10* (2012). In it, they carefully explore the cultural, historical, aesthetic and societal aspects of the line of BASIC (an old programming language) they named the book after. Using this line of code, a digital representation of a maze grows. In doing so, it provides a great metaphor of the murky area between human agency and that of the computer.

> Hedge mazes need to be planned and plotted, but unlike most other mazes, they must grow in order to fulfil that plan. 10PRINT's maze does as well, albeit in a different way than bushes do: once seeded, the computer-generated maze grows without tending, growing until the viewer interrupts it. (Montfort, et al. 2012, 39)

The computer programmer plans the layout of the plot and plants the seeds, the computer grows the bushes. For each type of different garden or different vegetation, a different language can be used. There are many programming languages to cause these operations within the machine. As I already stated, BASIC is an old one (though not completely retired). Which language is used is dependent on the hardware, the OS (operating system, a nice reference to the original operators) and on the preferences of the programmer or the company he or she works for. Also, there are differences between languages to develop an OS or program, and the scripting languages used to create web-objects. Within all of these languages, there are subcultures, dialects and different architectural approaches (Mackenzie 2003, 6).

For Rosenbaum the first step to 'building', as he calls it, is figuring out what his client actually wants. "Ten years ago, we did something called 'functional design'. We made a technical design based on the client's specifications. We then coded that, which took half a year. Two years later they had a product. Exactly what the client requested. Well, what they requested two years ago. And that was the problem: software was never any good" (Rosenbaum).

Now, he and his colleagues work in iterations, Rosenbaum says. They ask the client what they want and begin by building a very small part of that. They let the client test it, which always leads to new insights about how the product should

work. As an example Rosenbaum mentions an accounting application. Before they deliver the entire application, they deliver the option to enter data into a form. Nothing more. Based on that functionality, the client can specify what he or she likes or doesn't like. "So that time with the client, during which you are trying to get a clear picture of what they want, is very important" (Rosenbaum).

The methods applied here are fairly universal programming methods, e.g. *Prince2* or *Scrum*. What is interesting about this, is that like other crafts, programming is innovated upon. This happens not just by adapting the programming languages, but also in the manner in which they are used. As Sennett stated, the ability to localise, question and open up are indicative of true mastery and craftsmanship (Sennett 2008, 277).

After the concept is clear, the programmers both begin by building up their product piece by piece. In the case of Van Pul, this means he starts by asking questions about what the game he is coding will look like: "What are the elements? What are the rules? The first step is breaking it up. What are my characters? What are the power ups? What do the levels look like? Is it in 2D or 3D?" (Van Pul), et cetera. Rosenbaum stated that he used to take a very visual approach: build something, run it, and see what happens. These days he can go on for hours, simply typing code, without having to check the functionality in the meantime. Like in learning to play the piano, he has grasped the application of functions within his code.

A programmer's objective is thus to create reactions within the computer. This no longer happens directly on the machinery, but in the software that the programmer works with. This can happen directly in the BIOS, the 'layer' of software closest to the hardware and in the OS. Different programming languages have different affordances, so the choice for a specific programming language in the creation of an object is a well thought out choice (at least to the professional).

This tendency clearly showed in the spread of programming languages used in the Elegant Algorithms contest by SETUP; though in total fourteen languages were used, almost a third of the entries were programmed in visually oriented programming language Processing. This includes the winning entry. This language specifically lends itself to programming graphically focused content, and thus was particularly suited to the challenge of the contest: to recreate Mondrian's *Victory Boogie Woogie*.

The affordance of programming languages started out as a form of creativity within constraint. Back when computer programming first took place, the input devices included teletypewriters. These were not fit for the mathematical functions within the computer, so the mathematical functions were replaced with typographical characters (Montfort, et al. 2012, 13-14). Though the constraints and mappings of modern programming languages may present less clumsy, it does illustrate why programming languages develop the way they do. It is always an interplay between the wishes of the user and the options available on the hardware.

When the programming language suitable for the job is chosen, the programmer has to decide how to write his code. This is no different than spoken language, people choose a specific order of words in order to tell a story. Another familiar

example is that of the carpenter, having to take the direction of the grain of the wood into account to create particular shapes. Instead of grammar or grain, the programmer has to take the syntax into account.[1] We can relate that to Sennett's foundational elements of true craftsmanship, the ability to localise.

More and more programmers are defending code's aesthetic properties. Montfort et al. refer to esoteric languages, which are programming languages invented by hobbyists to test the limits of the usual languages (Montfort, et al. 2012, 58). Code is thus also a very cultural practice, rather than just utilitarian.

Computer scientist Donald Knuth gave a talk in 1974 which was published in 2007. In it, he talks about the aesthetic properties of the (earlier) languages too, saying one can write truly beautiful code, not unlike composing music or poetry (Knuth 2007, 39). Knuth picks up that trail of thought by referencing craftsmanship.

> Sometimes we're called upon to perform a symphony, instead of to compose; and it's a pleasure to perform a really fine piece of music, although we are suppressing our freedom to the dictates of the composer. Sometimes a programmer is called upon to be more a craftman [sic] than an artist; and a craftman's [sic] work is quite enjoyable when he has good tools and materials to work with. (Knuth 2007, 43, original emphasis)

It is interesting to note that Knuth compared programming to craftsmanship so many years ago, as well. This emphasises the importance of analysing programming from the perspective of craftsmanship yet again. Although Knuth is partially correct in his deduction that performing an act that doesn't require creativity is an aspect of craft, he only touches upon part of what craft entails. The seemingly menial acts that come with the practice of craft are part of the practice and improvement a craftsman, but creatively thinking of new ways to build digital objects is as well.

Sennett argues that craftsmen have a material consciousness (2011, 120), which embodies three concepts: metamorphosis, presence and anthropomorphosis. This means the craftsman transforms a material into an object, he/she leaves a personal mark onto the objects he/she creates, and finally, the object produced is spoken of as if it had human characteristics. This last aspect of material consciousness I consider to be inconsequential, as people tend to humanise non-human forms all the time.

Material consciousness is not without controversy when we try to apply it to programming. In 2011, technological- and organisational consultant Dan North wrote a critique called *PROGRAMMING IS NOT A CRAFT* on the so-called *Manifesto for Software Craftmanship* (2009). The manifesto decrees that programming is a craft. It can be signed by anyone who wants to do so. North opposed this manifesto for two reasons.

One: it can be signed by anyone, which means that people who have no true knowledge of programming get to decide that programming is a craft. According to North, there is a big difference between skilful programmers

1 // These are the rules of a programming language.

and laymen who simply muddle about. Programming should only be declared 'craft' when people recognise this distinction. I understand craft, however, as mastering a field of expertise throughout different stages of proficiency. The fact that anyone can sign the manifesto does nothing to delegitimise this. In fact: it simply means that outsiders understand that learning and performing programming is something that requires effort and insight.

Second: North believes that programming is not a craft because it should only be *serviceable*. No frills, no invitations to digitise the programmer's ego, just functionality. As soon as a programmer tries to spice up his code because he fancies himself a craftsman, he undermines this functionality. We can deduce that North equates the term 'craft' with grandiose objects that contain unnecessary aesthetic details to satisfy the craftsman's ego. A clear plea against the presence that Sennett ascribes to the craftsman's material awareness.

Looking back at the interviews we can indeed say that both gentlemen take various 'materials' (computer code) and turn it into objects. Their presence, although made more explicit by signing their work, is ingrained into the code as well. The interviewed gentlemen had very different opinions about this display of ownership within the code. Van Pul always signs his work, and attributes code he might have used off of others. Rosenbaum only filled out a small ownership declaration because his company requires him to, often even neglecting to fill out his name.

> There are movements that want you to code in a certain way. That is a good thing, because if you code in a certain way all your colleagues will understand your work and you can share it. However, I feel code should be functional. That you have your own style in programming, in your code; who cares? As long as it works. (Rosenbaum)

As Rosenbaum said: working in a particular style is sometimes required, which means styles are personal and recognisable. This does not necessarily mean that code is imbued with unnecessary aesthetic frills, it can simply be preference or affordance that dictates the form code will take. "An 'if…else' statement and a switch case statement do the same thing. It looks different, but you use one or the other. Sometimes they have advantages or disadvantages" (Rosenbaum).

Whether 'frills' are unnecessary is another argument we could take under scrutiny. Computer scientist Paul Graham compares computer programming with a painting by Leonardi Da Vinci (*Ginevra de' Benci*), in which Da Vinci painted each leaf in the background quite precisely. Graham says programmers should do the same. "Relentlessness wins because, in the aggregate, unseen details become visible" (2004, 28). This is where programmers can create their beauty, he argues. By looking at seemingly unimportant details like indents or variable names (ibid., 29).

After asking the programmers whether they thought programming was a craft, both of them replied positively. Van Pul struggled with the concept of innovation. Though in a craft there is a calibrated best practice of doing things, he says, in programming changes take place all the time. Looking at Pink's description of mastery and how this can never be achieved completely (2009, 212), I think it is fair to state that a calibrated best practice is not one of the pillars of craftsmanship.

Rosenbaum related craft to making things, visual and functional things. In practicing a craft, the craftsman create things of beauty that also function very well. The creation is important, he said. So let's look at the objects that come forth out of the programming process.

2.2. Object of programming: code and software

Code for software is written in two abstract layers: the 'front end' and the 'back end'. The front end concerns itself with what the eventual user of the software sees on his or her screen. The back end is the part of the software that interprets the interaction on the front end. "In *Star Wars* terms, the front end is the butlerish C3PO; the back end is the unintelligible R2D2" (Rosenberg 2007, n.p.).

Most of the code that constructs software is built up in parts. All these parts can be re-used by other programmers for their own constructs, provided the code is shared among programmers. Digital culture theorist Mirko Tobias Schäfer compares it to DJ culture, where different samples from songs are used to create new songs. Thus, both DJ's and programmers build on a larger cultural resource (Schäfer 2011, 67).

In keeping with Rosenbaum's emphasis on creation, I asked about the interviewees' objects of programming as well. What did they consider to be good code? Rosenbaum, who develops both the front end and the back end, underlined functionality and modularity. Good code is built up out of building blocks that you can take apart and rearrange, he said. Like Rosenberg said: "Software builders like to talk about laying bricks; sceptics see a house of cards. Either way, there's a steady accumulation going on. New layers pile on old" (Rosenberg 2007, n.p.).

Regarding good quality code, Van Pul emphasised clarity. As he works a lot with functionality built by others, it is important they provide clear instructions. Writing code with a lot of abbreviations, for instance, will make it compact, possibly faster, and very clear to its author, but makes it a lot harder for others to understand. Because code is often re-used or altered, this can be very problematic.

The quality of code is found in its resilience, whether it can be changed or used in unexpected ways, Rushkoff says. Functionality matters more than clarity, as clarity can be fixed later on. All three aspects, functionality, modularity and clarity were already named as the qualities of good code by Knuth in 1974 (2007, 40):

> In the first place, it's especially good to have a program that works correctly. Secondly, it is often good to have a program that won't be hard to change, when the time for adaption arises. Both of these goals are achieved when the program is easily readable and understandable to a person who knows the appropriate language. (Knuth 2007, 40)

Rosenbaum also touched upon bad variable names within the code (causing problems in both clarity and functionality), stating this is often times a way in which you can recognise a junior coder versus a medior or senior one. Van Pul stated writing an orderly piece of code is like writing a book. When you properly label your chapters and paragraphs it becomes a lot easier to understand. Sometimes you have to decide between clear descriptions and efficiency of the code, however, as leaving out the descriptions could make your code process a little faster. “Everyone will tell you something different, what is right and what is wrong. Some love writing very efficient code that processes quickly. Others want to know how things work and where those things would be beneficial” (Van Pul).

Both Van Pul and Rosenbaum agreed that code has changed over the years. Van Pul noticed how ActionScript became much more efficient in its third release, when it started focusing on object-oriented programming. Object-oriented programming (with the unfortunate acronym ‘OOP’) is a term coined by computer scientist Alan Kay in the 1970s. After its initial application, the methodology has been immensely popular since the 1980s.

In OOP, both the data and the means of using this data are combined in a single object. Hayes describes it using the example of a series of triangles. Very simply put: instead of describing all the elements of the triangles separately (the fact that it has three sides, the size of each side, the location of the triangle, the angle, et cetera), you only define the class triangle once. After defining the class, you can specify instances of your class and its possible alterations (rotation, different sizes, et cetera) later on (Hayes 2003, 107). Rosenberg tried to explain it as well: “Objects relate to other objects via strictly defined inputs and outputs, so that programmers writing code that must interact with them need not concern themselves with what’s happening inside them” (Rosenberg 2007, n.p.).

During his programming career, Rosenbaum saw the rise of object oriented programming languages *.NET* and *C#*. He thought it was interesting to note that the vision of how programming will evolve, has not changed. In keeping with what some call the ‘Lego Hypothesis’, this concerns the vision of software as being built up out of reusable parts (Rosenberg 2007, n.p.).

> Thirty years ago we said: ‘In thirty or forty years from now, we will only click things together… You have a window and you can drag a button on there and you can drag an action on there and you can basically tell it [the computer] what to do’. Ten years ago we said the same thing. We’re still actually at that level. We think that in ten years it will be easy to drag things together. Ultimately, that isn’t so. There will always be a need for people to know and understand code. (Rosenbaum).

Rosenbaum seems to be right, as enterprises to create lego-like software have not yet had any true success. Software needs to be fitted for a specific use and on a specific platform. It also needs to work alongside other software, work on the proper hardware and it needs to be understood by its users (Rosenberg 2007, n.p.). In Rosenberg’s book, it is compared to a “solitary cut-to-fit craft... lovingly handcrafted” (ibidem).

Code is used to create software, yet also created within software. A complex relationship that I will try to touch upon in this section. Functionality, clarity and modularity are three key points, but when we leave our understanding of code at that, we restrict ourselves only to its application. Media- and culture theorist Adrian Mackenzie argues that code oscillates between text and process (Mackenzie 2003, 5). It has a double form of existence. It is both text written by a programmer, as it is a process when the code is executed. The space in which the code works is a contested space. This space takes different shapes and forms based on the language or platform through which it is formed (Mackenzie 2003, 7).

As a text, it has many forms, dialects, engineering and architectural forms (Mackenzie 2003, 6), and as a process it works on heterogeneous matter such as text, audio-visual materials, structures, communications, et cetera (Ibid. 15). No longer simply a calculator, a computer has become a ‘media synthesizer and manipulator’ (Manovich 2001, 48). However, the computer does not understand the meaning of the objects being generated, it still relies on the numerical representations we build them out of (Ibid. 53).

Code creates objects through textual elements based on numeric representations, but at the same time the data that represents these objects is allocated on several places on the hardware during its execution. Because code moves between text and process, it is difficult to determine where it is, or what it is (Mackenzie 2003, 14). The importance of code, Mackenzie says, is interpreted in different ways. In academic circles the importance of code is mostly found in its abstraction.

Let’s begin this discussion with Friedrich Kittler’s radical statement that software does not exist at all. Kittler, a German media theorist, claimed that software is in fact non-existent, because all that software entails is layers of increasing abstraction that hides the simple act of chasing electronic pulses across a circuit board. He said “All code operations, despite their metaphoric faculties such as “call” or “return”, come down to absolutely local string manipulations and that is, I am afraid, to signifiers of voltage differences” (Kittler 1995).

Although Kittler argues that there is no software at all, effectively, he does so by delegitimising language in the process. By saying that the layering down of software towards voltage signifiers means software does not exist, he inadvertently argues that because language is composed of layered cultural constructs towards sounds, language does not exist either. By using software we give shape to those electrical pulses that run through the machine, thereby creating meaning. When Douglas Rushkoff was asked how working in a simulated environment relates to notions of materiality, he said: “I don’t really know. I think it’s harder for people to recognize the achievement sometimes. But now that programmers can get paid so much for figuring something out, society is coming to recognize that what they’re doing is real” .

Schäfer described software as being ‘in-material’. Although programming languages, like spoken languages, are not quite tangible and software can be copied, using the term ‘immaterial’ does not ring completely true. Besides residing in material objects such as hard drives, it also creates means of production (Schäfer 2011, 64).

Still, Kittler's description rung a chord with Chun, who started questioning the act of programming. She wondered whether programming was a clerical activity or a mastery of a field, when the computer is the one who actually calculates the outcome of the instruction and produces the result (Chun 2004, 32).

To answer this question, a closer look at software as a phenomenon is needed, say both she and Manovich (Manovich 2001, 48; Chun 2004, 28). According to Chun a shift from control over the machine towards a control over the engineering of software took place (Chun 2004, 33). This information engineering takes the shape of what we now often call 'new' media.

Manovich describes new media according to five principles. The first, I already mentioned: numerical representation (Manovich 2001, 49). New media objects, just like good code, are also modular. This means they are assembled out of several objects which can also be encountered in other places and configurations (for instance: a webpage consists of text and images that can also be viewed and edited on their own) (Ibid. 51).

New media objects can be automated. This is the element where the notion of craft becomes difficult. Because media objects are numerical representations, they are programmable and can be automated. This way processes without human intervention are possible (Ibid. 52). I would refer back to the example of the recipe or the maze, to illustrate what part of software needs to be looked at in order to understand the craftsmanship inherent to programming.

A fourth principle is variability. 'Old' media objects (photographs, newspapers), were created by man and stayed the way they were created. New media objects can be created in a variety of ways (Ibid. 55). For instance in the Elegant Algorithms contest, there were several entries which appeared visually different every time the script ran. Certain modules that have been written in code can also be used to process different programs when they are re-used in a different context, to create different applications. Thus even one bit of code can have a different affordance, dependent on the rest of the code it is a part of (Schäfer 2011, 70).

One interesting quote on the variability of new media is as follows: "If the logic of old media corresponded to the logic of industrial mass society, the logic of new media fits the logic of the postindustrial [sic] society, which values individuality over conformity" (Manovich 2001, 56). This suggests that we may be moving beyond the shift of handicraft towards industrialised processes by means of mass-customisation. According to Manovich, a programmer always tries to use variables instead of constants, to grant the user the chance to shape his or her own experience when they are presented with the results of the script (Ibid. 58).

With the fifth and last principle, Manovich hits the nail on the head. He calls this principle cultural transcoding. According to Manovich an exchange between technological elements and cultural artefacts takes place through new media, because we translate human-understandable media to computer code and vice versa. Because they are created in sync, a hybrid forms between the computer layer and the cultural layer (Manovich 2001, 62). This also affects how a culture such as the craft culture is understood and developed. This means programming is more than simply a clerical activity, as Chun discussed.

One thing that makes software development a difficult process, is that it is hard to define how long it will take to develop a particular piece of software. It is more than simply counting how many lines of code a programmer has written a day. In fact, often a programmers' time is invested in fixing bugs in code, or making code more effective (thus even decreasing the lines of code at times).

Because understanding code is (as of yet) not a universal skill, some troublesome cultural occurrences appear. Programmers are displayed as a type of superheroes with unknowable powers in popular culture (think of such characters as Neo in *The Matrix* or Garcia in *Criminal Minds*). In 1984, Hoare blatantly compared computer programming to high priesthood, saying: "[T]he high priest was the custodian of a weighty set of sacred books, or magician's manuals, which he alone was capable of reading" (Hoare 1984, 6), going on to explain how the same applies to programmers. In a more recent text by technology and surveillance academic David Murakami Wood, the same comparison can be found. Wood says:

> [A] new priesthood for the digital age whose access to the arcane world of code and algorithms provides them with the position from which to speak about the technologies they produce. These collectives are relatively opaque to outsiders. (Wood 2008, 16)

The comparison between high priests and programmers is thus a persistent one yet Montfort et al. disagree with this comparison. Though we've moved on thirty years since Hoare's comparison, programmers are still looked at as masters of the occult (which is quite ironic when we consider the other oft-used term for programmers is 'code monkey'). Montfort et al. say code is *knowable* (2012, 7). Code to them is not a mysterious entity, but something that can be explained and understood with proper effort and time.

Already in the basic make-up of the programming language BASIC (no pun intended), the concern for people becomes clear. The language includes the use of optional spaces to make the code easier to understand (Montfort, et al. 2012, 10). Programming languages are also adapted over time, to make reading and writing them easier (Ibid. 15). On explaining the line of code 10PRINT, the authors write: "A token-by-token explanation is like a clumsy translation from BASIC into English, naively hewing to a literal interpretation of every single character" (Montfort, et al. 2012, 16). Yes, computer code is difficult to understand if one is not acquainted with it, but given proper attention and study everyone can learn to understand how it works and how to use it.

2.3. Learning to code

Suelette Dreyfuss (a technology journalist and researcher) and Julian Assange (computer programmer, journalist and editor-in-chief of *Wikileaks*) discuss how hackers are often very intelligent people who fail to get proper intellectual stimulation in regular educational settings (Dreyfuss & Assange 1997, 14). Hackers are expert computer programmers, not to be confused with the

criminally intent crackers. Like Graham said: “Why do kids who can’t master high school end up as some of the most powerful people in the world?” (2004, x), by which he refers to the likes of Bill Gates and Steve Jobs.

Graham paints a very bleak picture of American high school, when he says it is mostly designed as a prison to keep children away from adults so they are free to do things that matter. All the while these children lean useless facts to keep them occupied (2004, 10). Rushkoff worriedly says that whenever schools do incorporate computer literacy into their program, they tend to teach programs, not computer programming (2010, 129). This needs to change, and people need to start learning programming.

These are two separate problems, but both of them involve the design of the educational space. Although it might be difficult to change the entire educational field for the logic-driven minds of the hackers, the level of technical education needs to be adapted across the board. Let’s start by looking at the way programmers learn their trade. Like Knuth before him, Hoare linked programming to craftsmanship, when he discussed the manner in which programmers get started:

> The programmer of today shares many attributes with the craftsman of yesterday. He learns his craft by a short but highly paid apprenticeship in an existing programming team engaged in some ongoing project, and he develops his skills by experience rather than by reading books or journals. (Hoare 1984, 6).

Hoare links programming to craftsmanship, not through its methods, but through the manner in which programming is taught. Although he wrote these words nearly twenty years before I wrote the words you are reading now, many programmers still learn their trade in the manner he describes.

Learning programming, like learning most things, is possible through a multitude of approaches. Students have their own preferences in learning skills, which is no different in learning programming. Programming does require a type of insight, which can be taught to those who are naturally talented, but also to those who simply keep at it. This insight is known as *procedural literacy*. Procedural literacy is the competency to understand processes and concepts, mostly associated with computational literacy but not necessarily limited to it (Bogost 2010, 245).

A ‘highly paid’ apprenticeship, as Hoare mentions, does suggest students of computer programming must be of a certain age to apply. I asked my interviewees whether they thought age was important when learning programming. Van Pul, who has experience in getting children to start programming, said that getting really small children to start coding would be difficult, unless they had a prior interest because ‘daddy works with computers’, for instance. Starting to teach procedural literacy around the age of ten or eleven would be good, he said.

In 1980 educator and computer scientist Seymour Papert wrote a book called *Mindstorms*, in which he explains how math problems become much easier to comprehend when they are modelled physically using gears. Instead of providing

every child with a set of gears, you can also use a computer, he argues (Papert 1980, vii - viii). Papert suggested children should build things and thereby develop algorithmic thinking (1980, vii - viii).

So one of the manners in which the knowledge of algorithmic thinking is conveyed is by building things (which is not too surprising as most programmers describe their efforts as building things as well). *LEGO* did so by using *Mindstorms*. Named after Papert’s book, this system allowed children to build robots out of *LEGO* and program them. As Bogost stated: “While robotics are integral to the process, they serve principally as a carrot to draw child interest; the educational value is understood in terms of their potential to develop general abilities in programming and creative expression” (Bogost 2010, 239).

Crucial to getting children to think algorithmically and develop their procedural literacy is to approach it differently in schools than it is now, Van Pul continued. Because teachers lack understanding themselves, they usually stop procedural literacy lessons after telling children that *Google* and *Facebook* can be dangerous. Children should also encounter the technical aspects, Van Pul says.

Van Pul used to do workshops in programming language *Scratch*, specifically aimed at children. Like the *LEGO Mindstorms*, *Scratch* emphasises building blocks in creating code. Although *Scratch* has a fairly low entry barrier, the reception among students is varied. Some students are immediately intrigued and want to build their own *Super Mario Bros*. Others get bored and quit as soon as they have finished their assignment. Sometimes disinterested students become interested as soon as they realise what they can create. “They realise: ‘Oh wait, I can draw my own characters and bring them to life on the screen. Hey, this is fun!” (Van Pul). The control the student has over their own coded little world makes a big difference. “Doing things yourself, building yourself. That stimulates children. Most children like making something” (Van Pul).

In 1972 Alan Kay wrote a bit of fiction called: *A Personal Computer for Children of All Ages*. In it, he described where he thought the personal computer was headed, based on the state of technology at the time. In doing so he roughly sketched a hybrid between a laptop and a tablet. Perhaps more importantly, however, he described how children would use such a device. The children play a game where they fly space ships.

In losing the game, the boy Jimmy’s spaceship disintegrates, causing a glorious win for his friend Beth. In trying to solve Jimmy’s conundrum, they try to do better by studying mathematics, physics and altering the program of the game. According to Kay, the benefits this system would deliver include: access to information that belongs to everybody, easy editing and the promotion of algorithmic thinking (Kay 1972, n.p.). Keep in mind this text is over forty years old, yet this is what we are working on today.

Ajit Joakar, founder of technology trends research company *Futuretext* recently started *Feynlabs*, a *Kickstarter* campaign in which he promotes his new system to teach children computer science called *Computer Science for your Child*. He focusses on computer science (not just programming on its own) to promote a better understanding of the subject matter which, according to Joakar, also leads children to pick up programming languages faster. Computational thinking is a fundamental skill for everyone, not just for computer scientists, argues Joakar.

To teach computer science to children, this system employs a three stage learning process: (I) concept; (II) computer; (III) extrapolate (see figure 1).

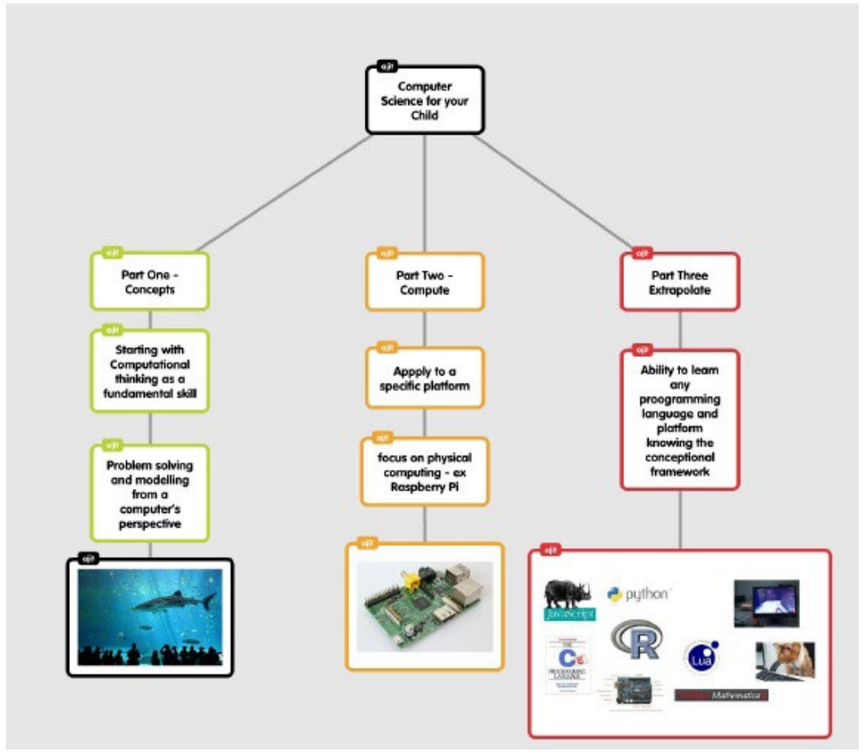


Figure 1. *Computer Science for your Child* explained in a diagram, image via http://www.opengardensblog.futuretext.com/archives/2013/03/feynlabs-using-the-raspberry-pi-to-teach-computer-science-part-two.html

In the conceptual phase, this system teaches children that computational thinking entails experiencing the world differently, based on different models of the world. The same way a shark experiences the world differently than we do (by sensing electrical impulses, in this case), a computer understands the world differently too: through numbers and algorithms. In the second stage the children are illustrated how computational thinking works by applying physically it to *Raspberry Pi* (a credit card sized computer designed to be used by children to learn programming) and *Arduino* (an open-source electronics prototyping platform). This fits nicely with Papert's assertion that working with physical objects enhances one's ability for abstract thought. In the final stage the children extrapolate the things they learned in the previous stages by applying it directly to a programming language.

Now, so far we've just discussed ways in which children can learn programming. Obviously, children learn faster, so starting earlier has its advantages. For adults there are solutions too, however. Of course there are books, there are computer programming courses, and so forth. Outside of the traditional spectrum, we have online platforms *Code Academy* and *Kahn Academy*. Both of these platforms are free to use and teach the user how to program through a feedback system.

These systems are not looked at without scrutiny. For instance Bret Victor, an interface designer (who worked at *Apple*, among other things), critiques Khan Academy's experimental approach. To see what certain values in a code do, they invite you to change the values, for instance. Victor states this is the same thing as asking someone to press unknown buttons on a microwave to learn cooking (Victor 2012).

This is reminiscent of Norman's description of affordances. Bad design means a lack of visibility (Norman 1988, 7). A function is only clear when it has only one possible purpose, which becomes clear though good feedback (Ibid. 22). According to Victor, the coding environment should make the meaning transparent and the coding environment should explain in context by annotating both the code and the data the code is used on (Victor 2012).

Graham says hackers learn to program much in the same way other makers learn their trade. By copying the works of earlier, talented makers. By reproducing already successful works, programmers learn what does and does not work. Only after they learn technique, they can start to become good at their profession (Graham 2004, 26-27). Being good, becoming masterful, is not something you can achieve and judge by yourself. Especially in programmer culture.

2.4 Programmer culture

Programmers, like perhaps many people in their own fields, have their own particular culture. Whether a programmer is any good, and can truly be considered a hacker, is often a judgement reserved for the community. Dreyfuss and Assange discuss the earliest hacker bulletin boards in Australia, and describe how they had an onion-like structure. Only when a hacker was deemed talented enough, had enough technical prowess and references from other established hackers, were they allowed into the deeper inner sanctum of the bulletin boards were the other talented hackers roamed (Dreyfuss & Assange 1997, 55-58).

Back then, the truly talented programmers would occupy themselves with activities such as phreaking, which means hacking the telephone system. "The purist hacker sees phreaking more as a way of eluding telephone traces than of calling his or her friends around the world for free" (Dreyfuss & Assange 1997, 64). The legal system had not yet adapted to hacking practices and the rules were fuzzy. There were three golden rules at play: "don't damage computer systems you break into (including crashing them); don't change the information in those systems (except for altering logs to cover your tracks); and share information" (Dreyfuss & Assange 1997, 67).

Today, hackers are mostly known in the media as criminally intent programmers who take down websites, often with political motive. But as Graham said: “Hackers? Aren’t those the people who break into computers? Among outsiders, that’s what the word means. But within the computer world, expert programmers refer to themselves as hackers” (2004, ix). As I am trying to sketch the craftsman programmer, I’d like to stick to the programming culture as they themselves describe it.

The sharing of information is still one of the cornerstones of computer programming. In the 1990’s the term ‘open source’ became commonly used (Rosenberg 2007, n.p.). Open source means openly sharing your computer code, letting the world see what you have made. This way programmers can learn from each other, improve upon each other’s work and share their expertise. Van Pul describes how this works in the game development scene:

A lot of indie developers share pieces. Not an entire game, but pieces of code and solutions they have conjured up. That’s shared freely. It’s also not considered something to make money off of, because it’s only building blocks. (Van Pul)

Programming culture stretches worldwide, from different nationalities, ages, but perhaps not gender. Unfortunately, programming is a very male-dominated sector. Back in the early days there were already only one or two women of note (Dreyfuss & Assange 1997, 67). Not much has changed. “Today, to walk into the management meeting of a software project and encounter a group of female faces is still an exotic experience” (Rosenberg 2007, n.p.).

It does have some characteristics that, although not shared by all perhaps, are shared by many. Rosenberg discusses the ‘geekiness’ of programming culture. Though it seems perhaps a bit odd to discuss the stereotypical geek in academic theory, Rosenberg is not wrong. “While not all geeks are programmers and not all programmers geeks, the overlap between the two groups is thorough enough that their shared identity is assumed as a matter of course” (Rosenberg 2007, n.p.).

According to Graham, who speaks of ‘nerds’ instead of geeks[2], this stems from a certain attitude in teenagers. While most teenagers are concerned with popularity, nerds prioritise being smart, even if they are not conscious of doing so (Graham 2004, 2-3). Because they don’t spend their time and effort at being popular, they are not. They do spend their time and effort on something else.

One thing a lot of programmers have in common is the way they *think.* Rosenberg describes a study in which IT professionals were surveyed. It came to light that 77% of the IT professionals preferred logical thinking decision-making over feeling decision-making. In the general population, this percentage is roughly 50% (Rosenberg 2007, n.p.). In fact, Rosenberg refers to the overlap between ‘geek’ thinking and autism.

2 // The two terms cause heated discussions regarding their differences and similarities, which I will not get into here. Suffice it to say they are both terms referring to outcasts of popular teenage culture.

> [O]bservers of the programming tribe have suggested that in order to commune more closely with the machines they must instruct, many programmers have cut themselves off from aspects of their humanity. But the Asperger’s/autism parallel suggests that, more likely, those programmers were themselves already programmed to hear machine frequencies as well as or better than human wavelengths. (Rosenberg 2007, n.p.)

This does not suggest that computer programmers purely think in terms of algorithms. Graham suggests just the opposite. He says that although computer programming is usually paired with computer science, it has less to do with science than it does with making. “I’ve found that the best sources of ideas are not the other fields that have the word “computer” in their names, but the other fields inhabited by makers. Painting has been a much richer source of ideas than the theory of computation” (Graham 2004, 21).

Empathy, understanding what the people want, is what distinguishes great programmers from good ones, he says. Graham even goes so far as to say that the above-average intelligence a lot of programmers seem to possess do nothing to diminish their capacity for empathy (2004, 32), thus the notion that programmers lack aspects of humanity is ludicrous.

What separates the good programmers from the great ones, what makes the difference between coding newbies and actual hackers, comes down to a set of capabilities. In the next section I will describe programming as craft, establishing what the word ‘craft’ actually contributes to the field.

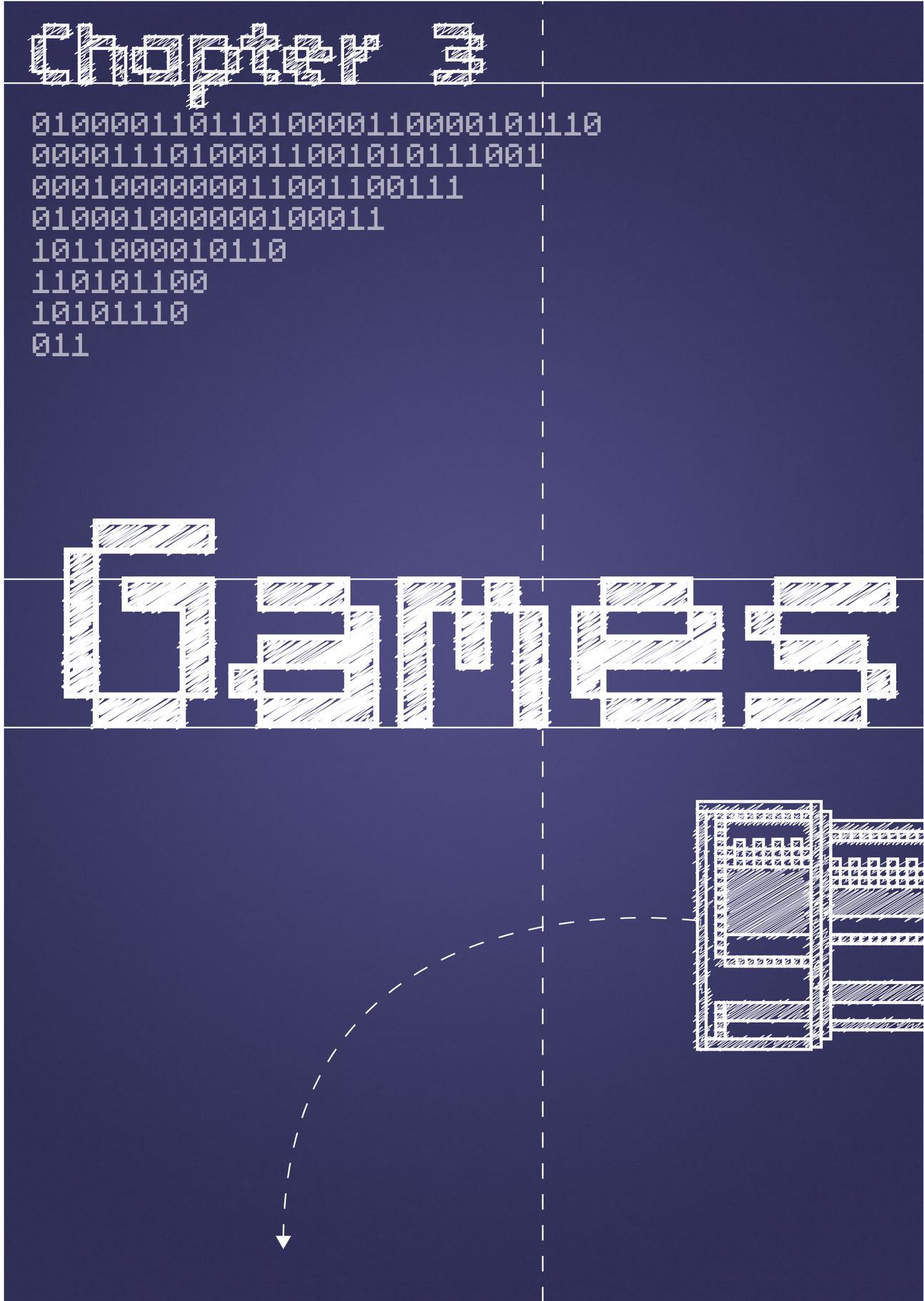
2.5. Programming as craft

There is a big difference between ‘programmers’ that pass themselves off as great developers and actual senior developers who excel at programming. They understand software development is a skill, in fact a whole portfolio of skills: understanding and modelling a problem domain, understanding programming languages, libraries, paradigms and idioms, choosing which to apply in a given situation, learning and understanding algorithms, mastering the ‘path to production’ (build, deployment, release), monitoring and availability, process automation, Lean theories of supply, production and product development, utility and cloud computing, concurrency and parallelism, I could go on. Those guys want a way to differentiate themselves (North 2011).

Unlike North, I believe this differentiation becomes clearer when we look at programming as a craft. North says software should be negative space; the area in which information is allowed to roam. A craftsman puts the quality of his or her code first. Code is written to create a clear and functional structure, whereby the language is chosen by its affordance. Programmers that add useless frills to code to stroke their own ego cannot possibly be called craftsmen. If software is indeed negative space for information to move around in, it still needs to be plotted in such a way that the information can do so effectively.

By means of software someone can built objects, either digital or analogue. It can thus be defined as a craft, because its raison d'être is to create. Though programming is an activity that takes place as an interaction between human and a machine, it is still the human that influences the machine to process code. Therefore the computer can be seen as the craftsman's tool: the operation of which is the skill that needs to be mastered. Though using a computer is a fairly universal skill, changing its modus operandi is not.

The materiality of software, although tricky, cannot be dismissed as non-existent. The computer programmer uses code to create software and other new media objects of use. To learn how to do so, he or she must practice the art of writing code until the code becomes knowable. By debugging software, writing more efficient code and implementing clarity into his/her design, a programmer keeps perfecting his or her creation. In doing so, like all other crafts, the programmer must work his or her way up from an apprentice level towards mastery, seeking guidance along the way.



Warcraft. *Minecraft*. Two insanely popular video games that have the word 'craft' right in their title. I would argue this is not just an empty use of the word indicating some aspect of the game's narrative. In the last chapter I discussed programming, looking at it from several angles. Among them, the ways in which one can learn computer programming. Some of the methods I discussed had quite playful elements. This chapter highlights that playfulness. In order to master anything, Richard Sennett mentions the ten thousand hour practice rule (Sennett 2008, 20). Another person who touches upon that ten thousand hour mark is game designer Jane McGonigal (McGonigal 2011, 267).

McGonigal discusses an untapped resource. The average young American has spent over ten thousand hours playing video-games. According to McGonigal, this means they are virtuosos of gaming. The skills they learn in these games could and should be put to good use.

Games might be an effective way to learn computer programming. For one thing, video games are created by computer programmers (among others). It is quite possible it could work the other way around, too. Before we delve deep into theories on games in the service of computer programming, let's take a look at why games matter in the first place.

3.1. Classic theories on games and play

Although programming is a relatively new phenomenon, play and games are not. Play itself must be understood before we delve into how games might be suitable to teach the craft of programming. Dutch historian Johan Huizinga argued in 1944 that play is one of the characteristics of culture in his book *Homo Ludens* (which means 'playing man'). In fact, play precedes culture. You can already see play when you watch a young pack of dogs play with each other. They may not have culture, but they already know how to play (Huizinga 1949, 1).

This inherent trait we share does not disappear when we grow older and more aware of society's rules. Contemporary civilisation shares the trait of play as well. Play must also not be understood as the opposite of seriousness, one can take their play very seriously (Huizinga 1949, 6). According to Huizinga, play has a few characteristics. It involves (I) Freedom; (II) it does not refer to ordinary or 'real' life; (III) it is constrained within time and place; (IV) the rules are fixed; and finally: (V) play creates order.

The first characteristic, freedom, relates to the fact that play cannot be forced. It has to be voluntary, a part of one's free time. Huizinga states that play can never be a task or moral duty. When we relate this notion to that of using games in education, we might run into a roadblock. Although education is not necessarily forced upon someone, the student does not pick the lesson plan either. Huizinga offers a small loophole by saying: "Only when play is a recognized cultural function—a rite, a ceremony—is it bound up with notions of obligation and duty" (Huizinga 1949, 8), yet that is of course not exactly what we are aiming for.

The second category is a little ambiguous when we look at it in the context of games for education. Although we want the game to refer to a very real aspect of life, i.e. the knowledge needed for computer programming, the game does not necessarily need to portray exactly that. The context of the game and the context of the real-life situation which it is supposed to influence are related, however, which I will discuss later on in paragraph 3.2.

The third category is not of true concern. The game may be played in a particular place and time (e.g. in the classroom or at home). Although the goal of the games is to alter behaviour outside of the game, this behaviour is no longer a part of play (although it may be perceived as playful by its practitioners due to their experiences in learning through play). The fourth and fifth categories complement an educational approach. The rules complement teaching styles and the order can assist the production of knowledge.

To the list of characteristics, French philosopher and literary critic Roger Callois added that play is never productive; it does not accrue wealth or new elements (Callois 1961, 9). When we encourage learning through play, however, we do want a new element to be acquired: knowledge. Huizinga explicitly speaks of play, however, while we might be more concerned with games. Callois further structured down the notion of play. He divided 'play' (*paidia*) and 'game' (*ludus*) on a series of scales. Where *paidia* was completely free-form (thus not equal to Huizinga's notion of play), *ludus* was the version that contained rules and structure, or a civilising quality (Callois 1961, 27).

About *ludus*, Callois states that it can be used in training to acquisition skills (Callois 1961, 29), which makes games yet again interesting for educational purposes. Huizinga's definitions of play has merit, but could do with some revision in the context of educational games. Educational games are not completely ingrained with freedom when they are included in lesson plans. Without any reference to real life, play may fail to educate about real-life situations. Callois' division of play and games is more beneficial. Games have potential to educate because of their rules and structure. Let's explore why and how this can be applied.

3.2. Games in education

To play games is an inherent trait of mankind, and could perhaps be quite functional when it is applied in education as well. An obvious first question is: why would we want to use games in education in the first place? That they are inherent to culture as Huizinga states does not primarily mean they are suited to teach.

To understand the ways in which games could enhance procedural literacy and teach someone programming I conducted interviews with serious game developers, such as *Qlvr*'s Peter Goes and *IJsfontein*'s Jan-Willem Huisman and Evert Hoogendoorn. To start with the negative: Hoogendoorn and Huisman stated that often times a traditional learning system would be a lot more useful than a game because they did not consider games learning to be the best means of *knowledge-transfer*. Rather, they felt games were a way in which people can decide on their own preferred learning style. To understand why they came to this conclusion, consider the use of standard entertainment games in the classroom.

Full-fledged entertainment games are games designed with the single purpose of entertaining the player. From role-playing games to first-person shooters to massive online multiplayer games, they take many forms. Some of these games are based on real-life events. The strategy-game *Civilization* for instance, allows the player to build an empire. To do so, they must pick a population and guide them through historical evolution, from a nomadic existence to a technologically advanced civilization. Some educators have tried using this game to teach players history. Kurt Squire, a games and learning theorist, described his experiences on using *Civilization III* in the classroom to teach history.

Although games are often associated with fun, Squire encountered his first resistance on this very topic when he introduced *Civilization* to teach. Students wanted to know why they were playing a game, when they were there to learn something. Some even considered it a waste of valuable time. Part of that frustration resulted from the time it took to get to know the basic game mechanics (Squire 2005, n.p.). Again the need to design a game well was underlined. As Squire said:

> Considering the difficulty and complexity of *Civilization III*, it is not surprising that students should have found the game challenging; what is noteworthy, however, is that students found this off-the-shelf computer game, which has been marketed toward a broad audience and which has sold millions of copies, more challenging than their traditional learning experiences in school. (Squire 2005, n.p.)

Hoogendoorn and Huisman also referred to using *Civilization* in the classroom: a student might be able to learn about the tension between the French and the English during the war, but he or she would learn nothing about historical facts like important dates, because those are not integral to the game. Tycoon game *Sim City* could also be used to explain economical systems, but it would take several tens of hours. “Well, if I can spend 48 hours in a classroom talking about how economical systems work, I will have achieved the same thing” (Hoogendoorn), “Sooner, even” (Huisman).

Education can have different goals. From teaching strategy to increasing skill to increasing fact-based knowledge, each requires a different approach. If you make the game too complicated, the learning goals might get lost in the process. Besides that, the fact remains that games do not appeal to everyone, certainly not one game (Squire 2005, n.p.).

Civilization provided Squire’s players with complex problems, and for some this indeed increased their knowledge of history. For others the game did not work as well as their regular curriculum did. The difference can be found in learning-types. Those who failed at the regular curriculum did well in the game and vice versa (Squire 2005, n.p.).

‘Traditional’ learners worried that playing the game would set them back in their skill set to apply for colleges, who test in the more traditional way with tests and scores. This is a problem that occurs more often.

> To give you an example, we were creating a game for primary education. It was about fractions. The client stated: ‘there needs to be a test, we need to see whether they can do the fractions or not.’ That’s a very old fashioned way of thinking. I know exactly what they can do, I am constantly testing and adapting their behaviour. I’m adjusting their knowledge level. …The system knows exactly what they can do, even more so than when they take a test. A test is just a snapshot. (Huisman)

Ironically, *Civilization* prepared students better for the new economy where students have to think creatively with digital tools, Squire says (Squire 2005, n.p.). Still, the game did not appeal to everyone. Part of the problem was that the games were made compulsory.

This does resonate with Huizinga’s theory that true play cannot be forced upon the student. Squire struggled with the question of how fixed learning goals correlate to the choices one would expect games to give to the player, as well. According to him educators need to reconfigure the education system to formulate personal learning goals to adapt to their learning styles.

Regardless of a preference of one style over the other, any method of education needs to be able to get a person to learn. Educational theorist David A. Kolb is renowned for his theory regarding learning. According to Kolb, learning is an experiential process. When someone learns they go through a four stage process, which is then repeated. The model below illustrates the different stages a person goes through to increase or adapt their knowledge.

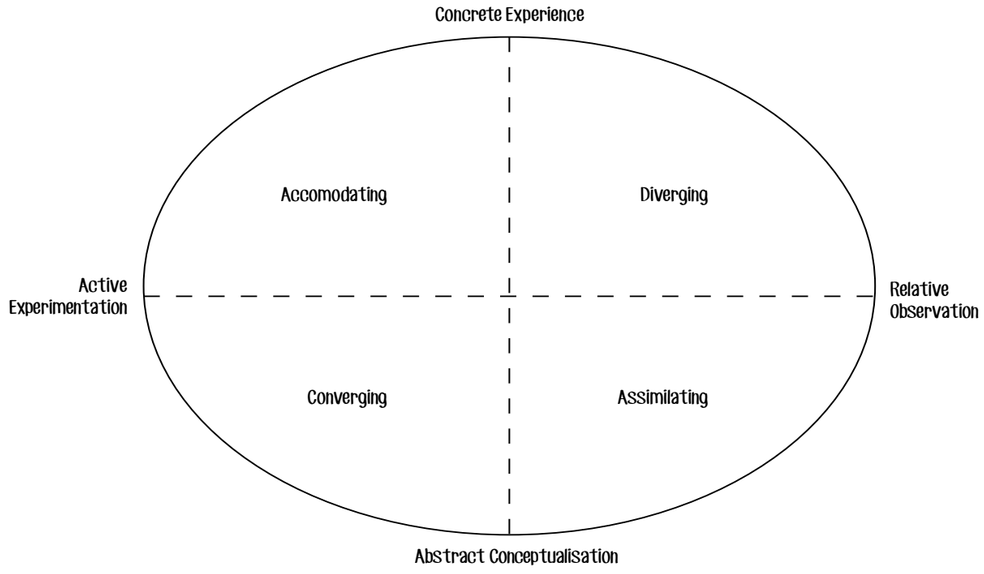


Figure 2. The Experential Learning Cycle and Basic Learning Styles, based on Kolb, 1984.

By experiencing something, a person can make observations about that particular situation and form a theory as to how this could situation could occur. The person then tests that theory by recreating the situation according to their theory, and then experiences and observes the outcome. The preference of one of these stages over the other determines their basic learning style (Kolb 1981, 238).

The process of learning and adapting is much more important than the eventual outcome, says Kolb. This is because learning entails adapting existing experience to new experiences, in which a person's knowledge is transformed. Knowledge is not a single thing to be added to a person's reservoir, but something that is constantly (re)created (Kolb 1984, 38).

The four stages of the experiential learning cycle could very well be integrated into games, with the added benefit that there is no 'test' at the end which focusses on a fixed outcome. The particular design of the game then determines which type of behaviour can be stimulated and/or changed.

According to Ian Bogost, education can be divided roughly into two camps: the behaviourist camp and the constructivist camp (2010, 233-234). Where the behaviourists follow the more traditional, classroom setting with positive- and negative reinforcement, the constructivist camp is based on one's environment, learning by doing and individual learning trajectories. In constructivist theory, each individual develops through a set of stages and each stage requires a different method for them to construct their knowledge. These two camps can also be spotted in serious game theory (Ibid. 236).

Roughly stated: behavioural thinkers believe video-games can transfer certain values through play, which then express themselves in behaviour outside of the game (Bogost 2007, 238). This is most commonly encountered in discussions on violence. A behaviourist approach would be to argue that the Columbine shootings happened because the shooters played too many violent video-games. This discussion, although interesting, is not relevant to this thesis, though. What is important, is whether games are suitable to transfer knowledge and skill onto the player.

Bogost says that behaviourist approaches tend to ignore the unique situation the player is in. Without the context in which the game takes place or taking the ambiguity of the meaning in the game into account, no behaviour may be conditioned at all (Ibid. 238-239). As Bogost states: "Behaviorism ascribes a singular, rationalist approach upon the content of video-games [sic]. Such a turn ignores Marshall McLuhan's suggestion that we understand media themselves as shapers of human experience, not just carriers of content" (Ibid. 240).

Constructionist approaches to game-based education, see games as a way to teach the underlying abstract values of a certain system (Ibid. 239). The superficial layer of content does not matter; whether you are playing with pirates or ducks, it matters that you learn how to navigate water. On the topic of behaviourism versus constructivism, all interviewees landed on the constructivism side of the argument.

The interviewees said children learn by doing. Because the players receive so much feedback as to what they are doing, they develop their own unique trajectories and strategies. Preference on whether games should be based on situated context or abstract models that are presented through random contexts, differed.

Goes stated most of the cases he handles require a very specific context because the problems they have to solve are quite specific as well. Huisman and Hoogendoorn related the level of abstraction not necessarily to the context, but to the role the player had to assume. Whether the game uses pirates or spacemen is inconsequential, their position within the context does.

> The role you present someone is the role they will assume in the game. So if you want someone to make a lot of decisions, you make him the head pirate. Whereas when you want the player to learn from others, you give him the role of ship's mate. The role you give someone generates the behaviour that suits that role. (Huisman)

A constructivist approach might be more aligned with McLuhan's ideas on media in general, but it has its faults as well. Because they are not concerned with particular situations or particular skills, they cannot convey their rhetoric effectively (Ibid. 241). This can be seen in the example of *Civilization*, like Huisman stated: it may teach you about the underlying notions of tensions between civilisations, but it will teach you nothing about specific dates in history.

This would suggest games are inadequate at transferring data, because their tie to reality is representational at best. When the transfer of strategy or understanding algorithmic thinking is the goal, however, games could be beneficial. Game designers need to carefully decide what type of learning their game needs to incite and the different methods different learner-types can employ to achieve it.

Educational author and speaker Marc Prensky stated learning through games would be easier for the so-called digital natives (Prensky 2005, 98). Unlike their digital immigrant counterparts, these people (who are generally born after 1980) have grown up in digital surroundings. This means they are better equipped to deal with these surroundings than the average digital immigrant. They are not very well equipped to deal with traditional learning methods, however. Linear thinking based on long texts fail to capture their attention or motivation, Prensky states.

Digital natives have different thinking patterns. (Prensky 2005, 98). They process information at a higher speed causing them to desire a faster pace of development. Information gets processed on a parallel basis rather than the digital immigrant's penchant for linear thinking. Visuals are preferred over text. Information gets accessed based on a hyper textual model rather than a linear one, i.e. via random access. This means information is ordered along a networked model rather than a sequential order.

Multiple sources can provide information rather than one or two authorities (such as a teacher). Their information (and the way they get it) is thus much more connected than it used to be (Prensky 2005, 99-100). Also: they expect results sooner rather than later, patience is not their strong suit. They prefer fantasy over reality, and finally: they have attitude (Ibid., 100-101). What a troublesome lot. The only way to reach them, it seems, is through (video) games.

The obvious features of video-games that can be used to teach digital natives include their highly visual-based form of existence, and fast pace with short feedback loops. According to Prensky, though, games work first and foremost because they are engaging. They provide enjoyment, passionate involvement, increase our adrenaline levels and elicit emotional responses. Perhaps more importantly, however, they have structure and clear goals. That motivates players to keep going and satisfy their need for ego gratification. Combined with the intense involvement, games can get us into the state of flow (Prensky 2005, 102). As discussed in chapter 1, the state of flow is crucial to practicing a skill if one aims at mastering that skill.

Though I don't agree with Prensky's sentiment that digital natives all have a sarcastic attitude that requires different ways of communicating with them, he does provide some insights as to why games are suited to help them process information. Of course not all people who want to learn programming are digital natives, but due to the relatively new arrival of the field of programming, a lot of the characteristics of the digital natives coincide with the ones that are needed to learn procedural literacy.

That still does not solve the conundrum that some students become annoyed when games are introduced in schools because they think they can only learn via the traditional classroom system. In the next section I will discuss games that are specifically created with education in mind.

3.2.1. Serious Games

One area in which the outcomes of playing a game often meticulously thought out, is the serious gaming industry. Discussing the particularities of every game ever made is not possible, so without dismissing situated contexts as being important, I would like to look at some characteristics of serious games for education in total. No one game is the same, of course, yet certain attributes can often, if not always, be recognised.

Tarja Susi, Mikael Johannesson & Per Backlund, academics in games, communication and information systems, wrote a text on the serious games field in 2007. In it, they provide an overview of the discourse surrounding the terminology applicable here.

Serious games are defined as games with a purpose other than mere entertainment (Susi, Johannesson & Backlund 2007, 1). Game-based learning are serious games with specific learning outcomes (Ibid. 2). The addition of the word 'digital' means games specifically based on video- or computer-platforms.

Terms such as E-learning and Edutainment also tend to rear their head, but have fallen into bad graces. E-learning concerns computer-enhanced learning, and Edutainment seeks to enhance education through entertainment.

Both types of education have been noted to result in "boring games and drill-and-kill learning", says Richard van Eck, an academic in instructional design (2006, 3). Digital game-based learning, however, is still an option worth discussing in both the games- and educational fields.

Michael Zyda, the founding director of the *USC GamePipe Laboratory* and a professor of engineering practice, defined this as: "a mental contest, played with a computer according to certain rules for amusement, recreation, or winning a stake" (Zyda 2005, 25).

Digital game-based learning is considered effective by the *QIvr*'s Peter Goes because of short feedback loops, repetition, interesting visuals, its ability to motivate and finally because it emphasises cooperative learning. "With digital techniques you have the possibility of providing a little more panache, fireworks. But, because children [or adults] can work together, it becomes more fun as well. Because they work together, it not only has a higher entertainment value, but it also stimulates learning" (Goes).

Games can also offer us an escape from reality. A purposeful and active one, says McGonigal (2011, 6). This does not mean their subject matter needs to be devoid of reality. The fact that games have a beginning, middle and end and that they are purposely played outside of 'regular' life provides the escape.

Unlike Prensky who felt the entertainment factor was one to games' biggest selling points, Huisman and Hoogendoorn warned against overly selling entertainment as an attribute to serious games. "Games are a tool for people to experiment with certain behaviour. This way they aren't subjected to the teacher's favourite strategy, but they can figure out the best strategy for themselves" (Hoogendoorn), "Motivation… I'm not too charmed by that myself. That it's fun to play is more of a by-product than the main goal. It's a lot more about your own learning strategies, direct feedback loops. That's what a game is about" (Huisman). Huisman also mentioned his experience with students being annoyed when a game was too much fun. They felt they were there to learn, not to play.

Papert gives two very important lessons on computer games as learning tools: (I) they are fun because they are hard, and (II) they should not hide the fact that they are trying to teach you something (Papert 1988, 88). Papert states that learning is hard, that shouldn't be masked from view. The very fact that games are challenging makes them fun, which correlates to the difficulty in learning something. Making it clear what it is they are doing exactly, also makes the players learn faster (Ibidem).

Whether the serious game is considered 'fun' or not, most people agree that games are very well suited to increase intrinsic motivation say Alke Martens, Holger Diener and Steffen Malo (academics in the field of game-based learning) (2008, 182-183), a concept highly valued in practicing and mastering a skill. Getting students motivated does not mean there is a transfer of knowledge, however, which is still the primary goal of education.

According to Papert, video-games are especially suitable for knowledge transfer, because unlike (for instance) basketball, video-games have a beginning, middle and end. This allows for a lesson domain to be formulated. Like I discussed

earlier, however, the game must be carefully designed before knowledge transfer can take place. When a game is presented in the wrong format for a particular learning type, not much of the rhetoric will make it across.

Martens, Diener & Malo discussed how three elements are the supporting pillars of game-based learning, namely: (I) technical aspects (i.e. games engineering); (II) pedagogical aspects of games; and (III) technology enhanced learning (i.e. game didactics) (2008, 172). They devised a model displaying the interplay between pedagogy, computer science and games which illustrates how learning, simulation and games rely on each other to enforce game-based training.

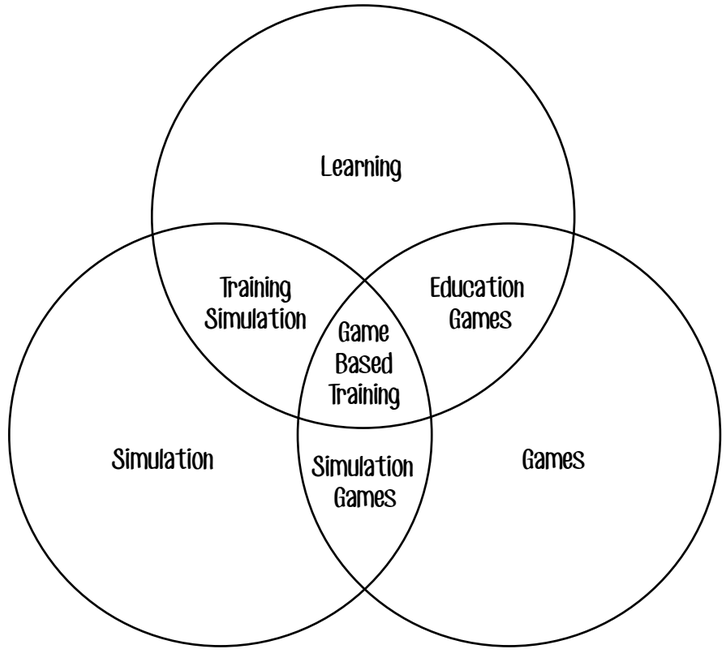


Figure 3. Interplay of pedagogy, computer science and games, based on Martens, Diener & Malo 2008, 174.

They also discuss transfer of knowledge in terms of *intramondial* and *intermondial*. Intramondial means using knowledge of one computer game to another. For instance in both *Age of Empires* and *Civilization*, one can upgrade technologies in order to give the player an economic, political or military advantage. A player who is familiar with the one game, will recognise this system in the other. Intermondial knowledge transfer occurs when a player can use his in-game knowledge in the 'real' world, outside of the game. For instance by building robots with *LEGO Mindstorms*, students learn skills they might apply in computer programming as well.

Media- and culture academic Stefan Wesener illustrates knowledge-transfer through different game types. Game worlds can be divided on a micro-, meso- and macro virtual level. At the micro-level, the world takes a linear approach where players and non-players are figurative substitutes, the entire interaction in the game us channelled through these substitutes. At the meso-level, a primarily linear structure is still in place yet most interaction takes place via quests. This includes direct interaction via menu structures. At the macro-level, the world is complex enough to allow different interaction styles. Some direct, some with figurative substitutes (Wesener 2006, n.p.; Martens, Diener & Malo 2008, 184-185).

Micro-virtual worlds only support the intramondial transfer of knowledge. In meso-virtual worlds some of the facts in the game might be adopted by the player too. In the macro-virtual world setting, intermondial knowledge may be transferred (Martens, Diener & Malo 2009, 185). Games become interesting from an educational point of view when they are complex rather than 'mini', a distinction Prensky made in 2005 (4).

Prensky defined this as follows: "Unlike mini-games, these 'complex' games typically require tens of hours of concentrated attention to master. They demand the learning of multiple skills, as well as the ability research and communicate outside the game" (Prensky 2005, 4). Martens, Diener & Malo state that simple games without simulation may still prove useful for primary education, and the removal of game elements leads to simulations such as applied in military and pilot training. However, all three elements are needed when higher cognitive skill needs to be developed. (Martens, Diener & Malo 2008, 173).

An added benefit of games is that they allow the student to take the reins instead of being the teacher's subject, says Papert (1988, 88). Autonomy is one of the pillars on which doing good work is based, so designing a game that takes this into account is one of the first steps. The interviewed game-designers also said that games give the player autonomy instead of being at the teacher's mercy. Games are not based on a pass- or fail principle. You can fail a level, but you can keep trying and improving until you get it. This sense of autonomy was touched upon by Pink in the context of mastery too, so it is an element of gameplay that can useful in the pursuit of a craft. There are more design choices to be made, however.

For instance, oddly, when we speak of children the term 'learning' is used, yet when we speak of adults the word 'training' is prevalent (Martens, Diener & Malo 2008, 175). Age in game-based learning no longer matters when the players are eighteen years or older, said the interviewees. Interestingly, Goes started calling it 'training' as well, once we started talking about adults. However, he felt that the only thing that matters in a game is the specific target audience, whether they are young or old. In the next section I will take a look at the way serious games are designed to 'fit' with the proper audience and to entice the *desired behaviour* the players ought to portray after playing.

3.2.2. Serious Game Design

I asked the serious game developers at *Qlvr* and *IJsfontein* how they design their games. Both companies are generally approached by other organisations. These organisations present a problem that they believe could be solved with a game. The serious game companies proceed to translate that problem into information relevant for a game. At IJsfontein they start with determining the desired behaviour the game should entice.

When educators use an entertainment game like *Civilization*, they are at the mercy of what the game designers put into the game. Their *procedural rhetorics* are prevalent, not the ones the educators feel would best suit their students. Serious games are specifically designed with certain goals in mind, entertainment notwithstanding, but also not as a priority.

Bogost discusses the properties of games to initiate a procedural rhetoric in his book *Persuasive Games* (2010). According to Bogost, video-games have a unique procedural rhetoric that can persuade and influence players. Of course persuading a student that a particular course of action or a certain form of information is the correct one is the goal of education, which makes the persuasive tendencies of games relevant. In using game-based processes, a game can persuade the player of a certain argument (Bogost 2007, 2-3).

As an example Bogost mentions a game called *Tenure*, which lets players take on the role of a teacher. By making choices in teaching styles, ways of dealing with colleagues and bosses, the outcome of your career will change. In doing so, the game argues that being a good teacher revolves around more than in-class contact with students. High school politics become important (Bogost 2005, 2). These arguments are compelling, as they are based not on words, but on procedures that the player must follow. Instead of passively hearing another's arguments, the player becomes actively involved in the process.

Although Bogost makes a good argument, Jonas Heide Smith & Sine Nørholm Just (games- and communications-theorists) fairly argue that the game still does not break the barrier where the subject must decide whether they agree with an argument or not. Even though they are acting out the argument in a game, it does not mean they agree with it. The proposal stays situational. Secondly, although procedural rhetorics are a very valid description of the rhetorical power of video-games, it does not always mean it is the most effective method of persuasion. An abstract, rationally stated argument might be better in some situations, they say (Smith and Just 2009, 57).

So in what situations are games exactly good at persuading? According to Bogost, especially when the game is specifically situated: "[R]hetorical positions are always particular positions; one does not argue or express in the abstract" (Bogost 2010, 243). David Williamson Schaffer (academic in education) and James Paul Gee (academic in the fields of linguistics, education and games) state this is exactly why games can be effective learning tools.

Education is in crisis and games provide a way out, they say (Schaffer and Gee 2005a, 12). Standardised testing produces standardised skills, so current educational systems are geared towards to creation of commodity workers, instead of innovative workers (Ibid. 5). Society needs craftsmen who have mastered a field. Games provide exceptional insights into particular situations because they allow the player to experiment with different identities and undergo situated learning (Schaffer and Gee 2005b, 106).

Instead of working with facts and figures, a game provides a player with an epistemology to handle a specific situation. "In game worlds, learning no longer means confronting words and symbols that are separated from the things those words and symbols refer to" (Schaffer and Gee 2005b, 106). Instead players are provided with the tools to solve problems specific to a type of situation: learn by doing, so to speak. This is reminiscent of how Hoare described craftsmanship: craftsmen learned their craft by imitating and practicing a certain act, yet lacked theoretical understanding to what it is they were doing.

Epistemic games, as Schaffer and Gee call them, do not leave the player completely in the dark, however. The player learns through the unique situation formulated in the game (for instance through surroundings, clothing, non-playable characters, possible actions and/or objects, et cetera) (Schaffer and Gee 2005b, 108). The facts 'come for free' during the course of gameplay, when they are presented in a coherent domain of knowledge (Ibid. 109). Based on the constructivist mentality of the interviewed game designers, however, this is arguably only true on a game by game basis. Not every game that is used to educate will include relevant, factual data.

Van Eck fairly stated that saying games are an effective educational method is not enough (Van Eck 2006, 17). Proper research needs to go into when and why they are effective. Part of the educational value games can offer lie in their ability to convey the proper information. To argue why a certain solution, method or bit of information is relevant to the player.

Both Qlvr and IJsfontein employ a fair bit of research into their subject matter. Clients and experts on the subject are consulted throughout the process. This means that the games developed at both companies are quite situational, which should make their procedural rhetoric more effective.

Eventual implementation is always part of the design process. When a game needs to be played on a person's downtime, yet cannot take up three hour sessions to play, the designers could decide a game should be played during downtime at work.

> So then you make an iPhone game. Something they can play in three minutes while they are standing near the coffee machine, so it doesn't interfere with their work. Or you put a four minute game time-cap on a session. Those choices have to be made at the start as preconditions, out of the implementation. (Hoogendoorn)

Testing the effectiveness of their games is often dependent on the client's time frame and budget. According to Goes, the effectiveness of the game is obvious because it has been built up out of these little pieces of functionality that were carefully designed. During the process of creating the game, the designers can already expect feedback from the client. After the game is finished, *Qlvr* conducts a month worth of tests, after which they ask the players what they thought.

At *IJsfontein* the desired behaviour is tested against the choices they made for the game's functionality. When the game is finished, they can enter a validation process with experiments and baseline testing, a process often taking up as much as two years. Though this trajectory occurs more and more, it is still up to the client whether they have the time and budget for this research. Although it is good that this occurs more often, overarching research into serious games for education would be quite helpful.

Schaffer and Gee say that with its careful design, the largest challenge lies not in the educational potential of games, but in integrating them into current educational systems. Teachers, mostly familiar with traditional schooling methods themselves, first need to believe the games will be effective (Schaffer and Gee 2005b, 110). After that the games still need to be paid for and developed. Obviously, development itself is not a straightforward situation, either.

One of the more serious problems in integrating games into an educational curriculum, however, is a financial one. Though most teachers are not digital natives and need to adapt to the idea of using games as an educational tool, the practical problem of money does not go away. Schaffer & Gee lament that epistemic games would fall into an extra-curricular system, because there is no mental or financial room for these games to fall inside the standard curriculum. This way, only the well-off would be able to play to learn (Schaffer and Gee 2005a, 23). Complete serious games are expensive to design, especially when both a game design team and a team of instructional scientists and subject matter experts are needed. A possible solution could be gamification of current educational practices.

3.2.3. Gamification

Software, information technology, and game theorists Sebastian Deterding, Dan Dixon, Rilla Khaled & Lennart Nacke defined gamification as: "the use of game design elements in non-game contexts" (Deterding, et al. 2011, 9). Before their description of gamification appeared, Ian Bogost already described gamification as being 'bullshit'. Gamification was simply a marketing tool. Slap a point system on something and because of it, your product is made more exciting for the consumer. Instead of gamification, Bogost suggested we use the term 'exploitationware' (Bogost 2011).

Gamification can be more intricate than simply applying a point system and it can be used outside of marketing as well. Of course, when we consider traditional education, some of the simpler gamification techniques can already be recognised. Grades can be compared to a scoreboard and stickers for a job well done can be compared to collectible badges. Some more thought needs to be placed on how game elements can provide a meaningful addition to education.

Gamification was only considered effective by the interviewed game designers when it was well-designed. "It's more than just a point-system. Psychologists and cognitives come into play, they have an idea of how the brain gets stimulated. When you let people play with everyday things, life can become a lot more interesting" (Goes). Hoogendoorn stressed that gamification is more than putting a game-layer on top of an existing system. You have to redesign reality from the ground up, and in doing so incorporating game mechanics. The same goes for education, if you want to incorporate gamification you need to re-design education from the ground up, not just add a 'sticker layer'. So unlike Bogost (2011), these game designers did not dismiss gamification completely, but they were adamant about the importance of the manner in which it was constructed.

Gamification is firmly rooted in games not play, say Deterding et al. (2011, 11). Within gamification, 'players' follow certain rules to get to a specific outcome, thus they are not located within the free-form paidia that Callois relates to play. Because games do not have a specific set of elements to them that are uniquely attuned to games and games alone, Deterding et al. suggest gamification is using game characteristics in order to get a gameful design (2011, 12). These characteristics can include interface design patters such as leaderboards or badges, but they can also include mechanics such as time constraints, resources, et cetera. Genres can also be used; challenges, fantasy, et cetera (Ibid. 12).

In a *Google Tech Talk*, Sebastian Deterding said the three elements important in gamification are Meaning, Mastery, and Autonomy (Deterding 2011). We've read that before. To be specific: Pink's pillars to do good work were Autonomy, Mastery and Purpose. Gamification and good work are able to thrive under the same circumstances, thus making it perfect to have people practice something.

In order to make gamification relevant, the gamified application has to have some form of meaning relevant to the user. In education this is fairly easily achieved by relating the application to the fact that the player is learning something. Like Papert said, hiding this fact is not constructive, being upfront about it actually helps people learn faster.

Concerning the improvement of skill: the challenges offered up in the gamified application should also be interesting to the player. Accruing points by filling out dull forms is not satisfying. Rules can be a way to make challenges interesting. By applying rules and by increasing the difficulty at each new level slightly, keeps playing interesting. Again, this is applicable to practice and getting into the flow as well. Like Pink said:

> Most important, in flow, the relationship between what a person had to do and what he could do was perfect. The challenge wasn't too easy. Nor was it too difficult. It was a notch or two beyond his current abilities, which stretched the body and mind in a way that made the effort itself the most delicious reward. (Pink 2009, 192).

The difficulty level should not rise at a steady pace, however. Spiking the difficulty level at times allows the player to fail and rethink his or her strategies. When they then finally do pass that difficult level, they will feel the sense of mastery so important to intrinsic motivation. Like I discussed in chapter 1, intrinsic motivation is a pillar on which the road towards

craftsmanship rests, without it good work cannot be done. Proper feedback is the only way in which players will understand what it is they have done to fail or how they achieved their success (Deterding 2011).

Finally, autonomy. Whereas work is not always done for pleasure, play is a freely chosen activity. Thus making autonomy one of the important elements of gameful design, says Deterding (2011). This does not correlate when we keep in mind that most good work is done when a sense of autonomy is registered, too. However, when we relate autonomy to the mastery of a field through intrinsic motivation instead of monetary gain, we arrive at roughly the same value.

3.4. Games & Programming

You might wonder why I discuss gamification. Of course Rushkoff argued that programming is a crucial skill and it should be taught in every school (Rushkoff 2010, 130). Combined with the possible benefits of game-based learning, and the financial hindrances to implement game-based learning, makes discussing the possible integration of games or gamified techniques relevant.

There was another motive, too. One of the more well-known avenues to learn programming is *Code Academy*. *Code Academy* uses gamification techniques in their platform, so taking a look at that case might give us some more insight into its application.

Code Academy, an online platform that was founded in 2011, makes use of a feedback system to teach its users computer programming. Subscribing is free. To learn the code, a short and to the point lesson sits next to a text editor. After entering a line of code and hitting the 'save & submit' button, the result of your work gets shown in a console. By following the lessons, users can earn points and badges. There is also a counter sitting on top of the page, counting how many days your coding streak has gone on for.

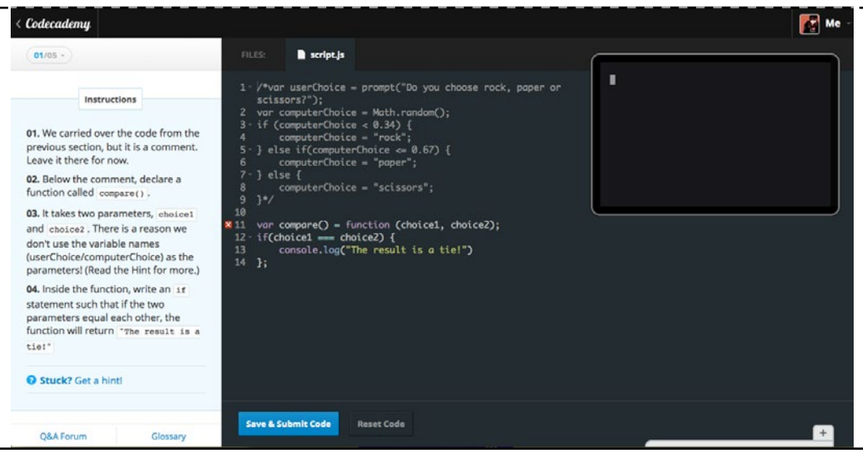


Figure 4. The *Code Academy* editor, screenshot via www.codeacademy.com

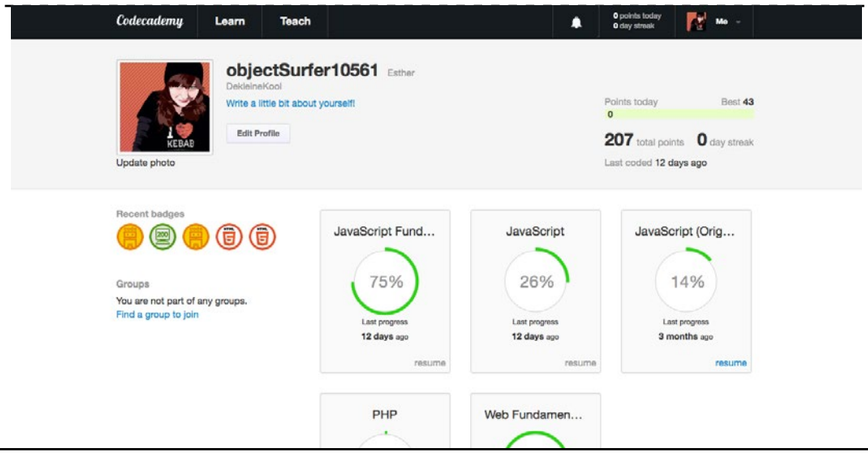


Figure 5. Points, streaks and badges in *Code Academy* screenshot via www.codeacademy.com

Code Academy is based on gamification techniques. In an e-mail to the author on July third, 2013, its community manager Linda Liukas stated:

> Gamification of education is obviously a big trend. The way we see it, the best learning experiences come down to motivation and reward systems. The problem with learning things from books and videos is that you're just reading or watching them by yourself, and there's no reward when you've finished. With *Codecademy* [sic] the interactive interface gives the user points and badges for completing exercises and keeps them motivated.

These points and badges may work for a group of people, but are largely reminiscent of what Bogost dubbed exploitationware. The badges and points are fairly empty compensation compared to the more meaningful intrinsic reward of mastering the craft of programming. In order for *Code Academy* to be true exploitationware, they would have to 'trick' the user into paying them for the platform's advantages, yet subscribing to the platform is free. Failing to encompass meaningful game characteristics does means their gamification is weak, at best.

Code Academy does come with a useful feedback system, however. Code is explained step by step. When the user fails to write the code necessary to proceed to the next lesson, hints are provided or the user can refer to the Q&A forums where users help each other out by asking and answering questions.

Goes stated he himself is a big fan of things like *Code Academy* because of its ease of use and quick feedback on a job well done. "It's fun to do because you get successful experiences quickly. It's also very accessible; I don't have to install a complete software development kit to get my first 'Hello World'[3]" (Goes).

3 // "Hello World" is typically one of the first things you learn in any programming language because it is easy to code and can be used to test whether a system works properly.

There is one element to video-games that I have purposely ignored as of yet. In order to create video-games, they have to be programmed. Interestingly, when one follows the JavaScript track on *Code Academy*, users learn how to make simple games such as rock paper scissors and later on even blackjack.

A lot of computer programmers get started by tinkering with computer games. Rushkoff described this as going from player to cheater to modder to programmer (2010, 137). After a player is done playing by the game's rules, he might go online to find cheat codes. At this point he will be able to break the rules of the game and thus the game itself. Like Huizinga said: "It is curious to note how much more lenient society is to the cheat than to the spoil-sport. This is because the spoil-sport shatters the play-world itself" (Huizinga 1949, 11).

Using cheat codes might be fun at first but in the end does nothing to really add to the game. At this point the player might start to 'mod' the game, modify it. This can happen in-game. Several strategy-games provide the player with the option to design their own levels, for instance. This might not be enough for our enthusiastic player however, and he could start editing the game's files. Keep this process going long enough and the player will be able to release his own version of a game, or build an entire game himself: through programming (Rushkoff 2010, 137-138).

This is exactly how Van Pul started out with computer programming. He started out by playing with *ZeldaQuest*, an editor that allows its user to edit the original *Zelda* game of the *Nintendo* environment within a PC-emulator.[4] Rosenbaum first learned HTML, though he doesn't really consider that to be a programming language. After HTML he moved on to ActionScript (Flash), Java and even assembly language.[5] He says: "I just liked doing it. I wanted to build a website and kept going. I kept trying to find all the available options" (Rosenbaum). Both programmers went on to pursue a degree in their respective fields; Rosenbaum in computer programming and Van Pul in game design.

Rosenbaum underwent a dual-approach at his university of applied sciences. They would alternate an hour of theory with an hour of practice. During his very first semester he was taught to build a game: backgammon, to be precise. "You have to start thinking in the abstract: object-oriented. You can move a pawn forward. No, rather, you can put down a pawn. And you create the board. Each lesson would be focussed on one of those things. Finally you combine the elements and make the game. That's how you learned how to program over there" (Rosenbaum).

Games and programming are thus very integrated. *iPad* turn-based game *Hakitzu* leans on this very fact. In Hakitzu, the player controls fighter robots. In order to control the robot, directions have to be entered in JavaScript.

4 // An emulator is a program that allows an application that was built on a specific set of hard- ware to run on a different set of hardware. In this case the developer made sure that PC-users could run a game on their PC's originally designed for a game-console by *Nintendo*.

5 // Assembly languages are low-level programming languages, meaning they are a lot closer to the hardware level than the abstracted higher-level languages mostly used to build soft- ware.

The JavaScript used is very basic in order to let people who have never coded before play the game and learn as they go along. The game is aimed at children aged 11-14. The game's developers state that the game will not get you a job in computer programming straight away, but offers players an introduction.

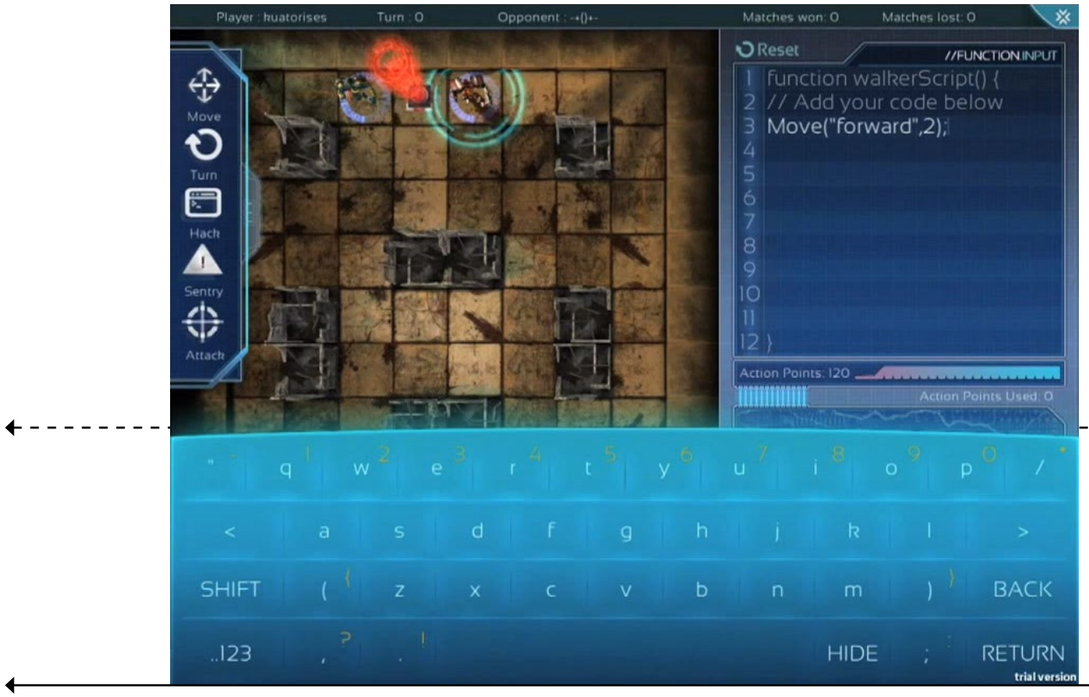


Figure 6. Screenshot of *Hakitzu*

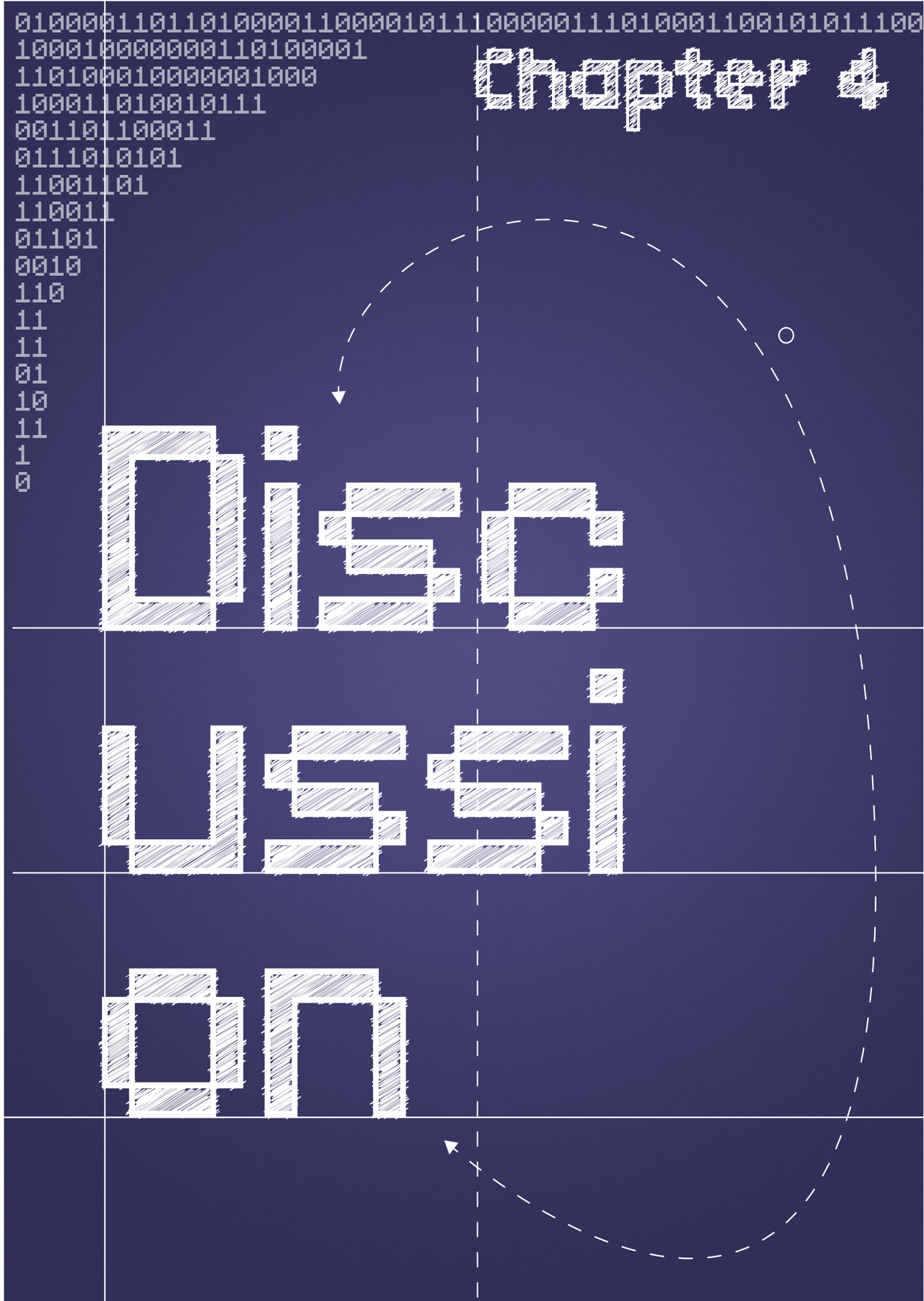
In order to get your robot to move or shoot, instructions have to be written in JavaScript. Each turn gives the player a certain amount of points which decrease with each move. Walking requires less points than shooting a laser, for instance. As the player learns more code, they get more control of the game. In a later stage, the player can edit his robot using code, for instance.

Strengths to this particular game are its clarity of purpose (learn to code), the social aspect (being better at code than your friend will allow you to a. destroy his robot and b. show him an amazing graphic of an explosion he hasn't seen before), and by and large, the game is situational. Because a lot of programming is geared towards technology and in fact programming games, players quickly understand what they are working towards by controlling robots in a video game.

An obvious downside to the game is its platform: not every student will be able to afford an *iPad*. The game also will not teach you to become a craftsman programmer, it just introduces you to the basics. This problem seems to be a big one in getting games to teach someone a craft, be it programming or something else. As discussed earlier, the more information needs to be conveyed, the more complicated the game will get, thus making it less suitable for a heterogeneous audience.

This is perhaps the crux of the story. Although *Code Academy* might not have the most interesting game-elements integrated into its system, it is simple enough to get the beginner programmer to write its first code. Like Goes said:

> The way *Code Academy* often works is to run scripts from one file. If you look at how we [game developers] write software: we have hundreds of files. So it becomes quite complicated to keep up. When you get to that higher level, the lessons will be more abstract as well. You will move into teaching ideas instead of providing examples.



4.1 Re: Codecraft

In chapter 1 I concluded that craft is all about creation, whereby practice makes perfect. Not literally, because true mastery includes never being satisfied with the end result and trying to find new ways to make the creation process even better. For none this rings true as for the master programmer. At MIT there is graffiti that states “I would rather write programs to help me write programs than write programs”(Rosenberg 2007, n.p.).

Hacker culture is a very geeky one, but also a social one. By sharing bits of code open source, programmers help each other improve. This social structure also ensures a hierarchy forms, based on technical aptitude. This does not have to mean aptitude in all code across the board.

In programming this is a bit difficult. Say you know C# and HTML. You can be a senior in HTML while you are a junior in C# or .NET. But generally these are the terms used, and when a junior writes code... You can tell. (Rosenbaum).

A true hacker knows programming languages have affordances and will be able to select one best suited for the job at hand. According to Sennett, someone learns his craft mostly through practice. Programmers can practice their skill by recreating works of others, not unlike other makers such as painters. To those truly motivated, practice has the added benefit of getting them into the flow, which is beneficial both for mental stimulation as for the learning of procedural literacy.

Of course there is the troublesome idea that craft is truly about the manipulation of very tangible materials, where technology does not help but only hinder the process. Some even believe that calling programming a craft diminishes the quality of code, because it will ensure programmers add unnecessary aesthetic details to their code, just to stroke their ego.

However, the devil is in the details. Someone who writes quality code makes sure it’s modular, clear and resilient. The craftsman knows how to localise, question and open up. The hacker defines the ‘problem’, determines the best programming language to solve it and creates an application that solves the problem.

From the early days when the input devices clumsily used typographic characters to portray mathematical equations to the object-oriented approach most used in programming today, even beyond that towards the vision of a Lego-like approach to computer programs, developers are constantly creating and improving. Although they do not work with tangible materials, their in-material objects do produce tangible results.

As Graham suggested, learning computer programming properly has more similarity with learning other ‘maker’ type skills such as painting, than it has with the ‘sciences’ most people think computer related skills belong. To cite Knuth:

> We have seen that computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty. A programmer who subconsciously views himself as an artist will enjoy what he does and will do it better. Therefore we can be glad that people who lecture at computer conferences speak about the state of the Art. (2007, 44).

4.2 Craft-based learning

It seems like such an easy deduction: ‘School is boring. Playing games is not. Computer programmers create video games. Practice make perfect. Therefore: computer games are better than traditional learning.’ Of course by looking at the ways in which people learn and the possibilities of knowledge-transfer and strategy forming through games, we saw that there is a lot more to take into account.

The work by Martens, Diener & Malo suggest that the more complicated a game is, the more a person could learn, but also the less heterogeneous its target audience could be. The interviewed game designers fairly averted this problem by targeting very specific behavioural changes in very specific target audiences. In an iterated process of research, trial and error, all games were designed for specific contexts and specific people.

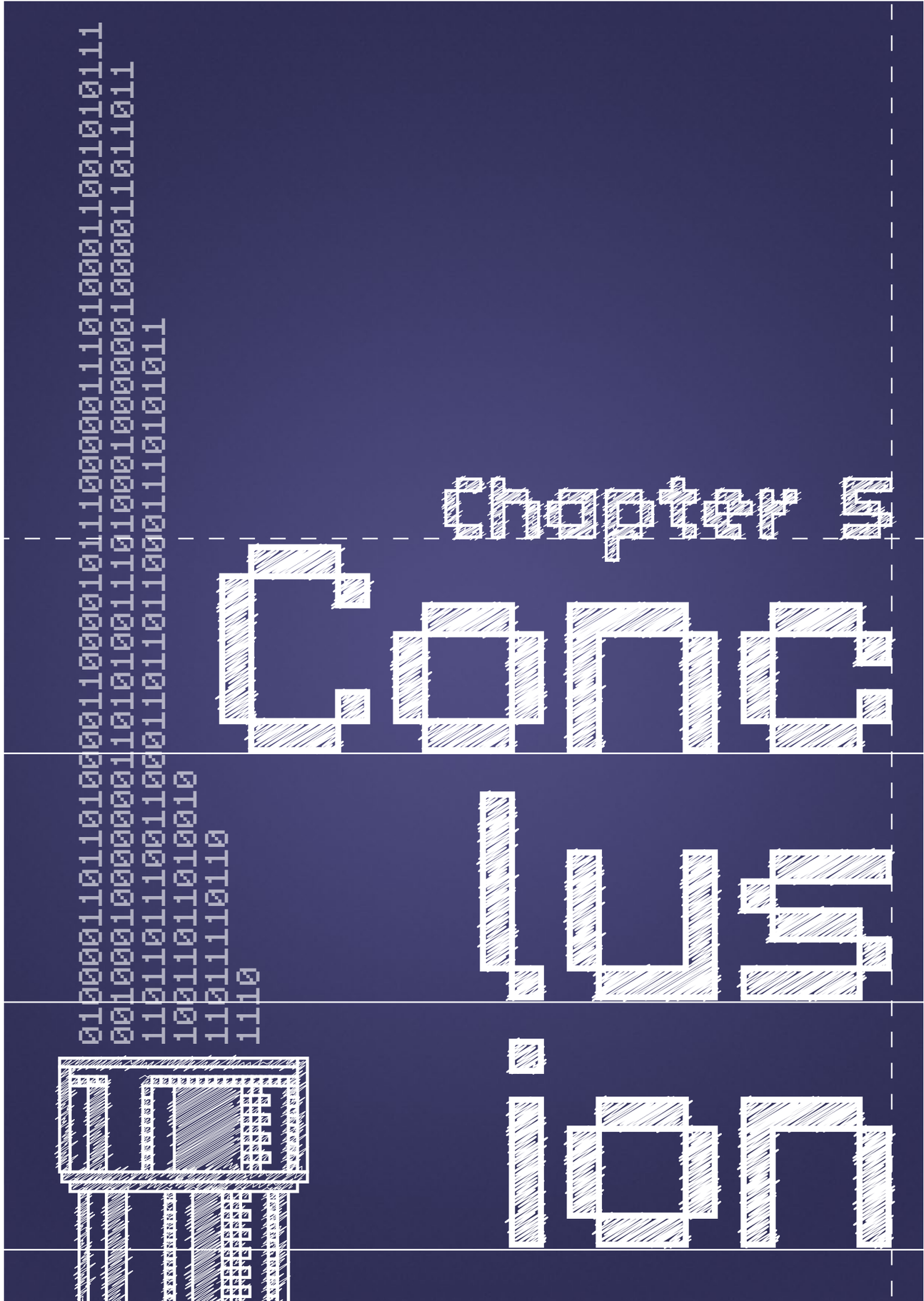
These games were not designed specifically to be fun. Although Prensky argues this to be one of the reasons why games appeal to digital natives, this aspect seemed to be under scrutiny upon further exploration. The game designers at *IJsfontein* were a bit apprehensive in using terms like ‘fun’ and ‘motivation’. They felt that students were in school to learn and that games should not discount that fact, which echoes Squire’s experiences in implementing *Civilization* in the classroom.

This seems to be in agreement with Papert’s assertion that games are only fun because they are hard, just like learning. Only in being clear about learning goals can a game trigger proper motivation. The aspect of fun was approached differently, yet again, by *Qlvr’s* Goes, who stated that because games can encourage cooperative learning, they are a lot more fun than traditional teaching styles

Moving past the aspect of fun, the subject of educational value of games was a subject all game designers seemed to agree upon. Games offer repetition, direct feedback systems, escape the ‘pass or fail’ that comes with traditional testing and allow the player to figure out their own strategy rather than being subjected to a teacher’s preferences. In doing so, the four stages of learning by Kolb can all be experienced sequentially while playing a game: experiencing, observing, conceptualising and experimenting.

However, not everyone enjoys working with games. Like Martens, Diener & Malo illustrated, the complexity and genre of the game matter a great deal. Where the more complicated games have a higher chance of transferring knowledge, the learning curve to play the games might be too high for some students. Ironically, playing the game can be more challenging than learning in a traditional setting.

Van Pul encountered the same problem in trying to teach children to program their own games in Scratch. Some children were simply not interested enough, though this could be turned around when they realised their sense of autonomy and mastery. Autonomy, mastery and purpose, the aspects needed to do good work according to Pink, can thus be present in both computer programming and game play, yet is this enough to allow games to teach someone the craft of programming?



I started out this research with the research question: How can games teach one to become a craftsman of programming? Further sub questions were organised and answered throughout the different chapters. To answer the research questions I conducted a literature research (first an explorative one, afterwards a focused one (Gibson & Brown 2009, 38-42). After reviewing the literature, I composed general interviewing protocols based on the literature. The interviews were chosen to provide a lot of qualitative data, yet were carefully designed to avoid coming up with a heap of data that would not be analysable (Brinkmann 2013, 4). The interview designs were based on the literature, to clarify how current industry practice reflects on academic debate.

These methods ensured that this thesis was in line with current academic discourse yet also provided novel information, i.e. the specific avenues in which games might be suitable to teach the craft of programming. By choosing the methods of literature research and interviews, other avenues of research have been left open, such as closer case studies of the current available means to learn computer programming (via games).

By interviewing all male industry professionals, most of whom live in the Netherlands, the empiric data gathered might be more heterogeneous than if a larger, more diverse sample group was used. However, by offsetting their statements to the literature and carefully analysing differences and similarities, the following conclusions have been reached.

In chapter three I discussed how games are ingrained with a procedural rhetoric, according to Bogost. Although Smith & Just say the arguments presented in the game remain situational, the general discourse suggests that games can indeed be used to trigger behaviour, if this behaviour is something the player believes to be beneficial to him- or herself. Of course the desired behaviour here is to get the player to learn how to program, to practice, and eventually to write quality code.

Becoming a true craftsman means mastering a field and combining tacit knowledge with clerical activity. Although the definition of craft tends to end there for a lot of people, such as for instance Knuth, innovation should not be forgotten. Sennett says a true craftsman localises, questions and opens up. This means a craftsman researches his starting material, questions the options it has to be transformed or used, and finally shapes the material in the best manner it could be shaped. Like I stated in the first chapter, craft is about creation.

Programmers are definitely craftsmen in the sense that they build structures.

Like Montfort et al. describe: programmers plot the shape and function of a digital object much like a gardener plans his garden. The computer, provided with the algorithms entered by the programmer, then grows the structure into the digital object. You could even compare the continued maintenance and debugging of software with the tending of a matured garden.

Although many people are hesitant to regard a technologically-laden activity as craft, understanding and writing computer code is a) technologically-aided but not driven, and b) complicated enough to be regarded as a craft. Hoare even says computer programmers share many characteristics with the craftsman of old: they often learn on a basis of imitation and practice, a ranking system

of junior programmers versus senior ones is still in place. Rosenbaum added that differences in the code written by either will be noticeable to those with experience in the field.

Hoare moves on to state craftsmen are actually modern high priests with volumes of text that is only understandable to them, but misses the mark with this statement. Like Monfort et al. stress: code is knowable. It simply requires practice to understand and write computer code. Rushkoff rightly argues that more people should do so. Computer programmers themselves created quite the hierarchy wherein those with technical skill are clearly more elite than those who lack this skill.

The emotional resistance towards looking at a technologically-based profession as a craft should be revised as the continued reliance on computers becomes more pervasive in society at every level. Computer programmers are alone in their ability to change the modus operandi of the very technology feared to replace human agency. That programmers work with and produce automation does not mean that their field is a completely technical one. Programming requires the ability to think procedurally, and in building blocks. The ‘negative space’ in which information is allowed to flow must be plotted carefully. Affordance, design, clarity, variability and resilience need to be incorporated. This is craft worthy.

Many of these characteristics can be taught by means of a game. Games offers players the option to try different strategies, all the while receiving immediate feedback on their actions. This is an excellent method of teaching players their own preferred learning style, the interviewed game designers suggested.

Games also offer the player a sense of autonomy, as the style they choose to play the game is their own and not one decided upon by the teacher. This sense of autonomy is mentioned by game theorists such as McGonigal and Deterding, and was confirmed by the interviewed game designers. Deterding suggests that by moving through the different levels and encountering parts that are challenging, the player can obtain a sense of mastery. By practicing over and over and trying different options, a game’s level can be won. This means a student can go through the game and learn at their own pace, instead of being dependent on exams that portray only a snapshot of a person’s abilities.

A well-designed game also offers relevant goals. Papert asserted that when games are used in an educational setting, making it clear that these goals are educational is crucial. Learning is hard, as are challenging game levels, and learning is serious. Although there are those such as Prensky who state that games are great motivational tools because they are fun, the interviews conducted suggested otherwise. This does not mean that there is no fun to be had, but most of the fun comes from cooperative play, said Goes. The social aspects of video-games make them yet again suitable to teach a craft. The road to craftmanship is also shaped by collaboration, imitation and sometimes rivalry. These parallels are not hard to find within gameplay.

Games should not hide behind a cloak of ‘fun’ in order to trigger players to learn, though. It seems Papert was right and educational games are only effective when they are clear about what they want to achieve. This means

that games will not be attractive to students across the board, because the students need to have the intrinsic motivation to start programming in the first place. Well, mostly.

Van Pul suggested that some children who started working with Scratch start to feel this intrinsic motivation only after they discover that they can create with it (some were motivated from the start and others never quite enjoyed working with computer code at all). And as I stated, craft is all about that creation.

All of the mentioned characteristics, autonomy, mastery, goals, practice and learning, are needed on the road towards craft. Intrinsic motivation is the first step in doing good work, which is something games can offer. The sense of flow, crucial to practice according to Pink, can be achieved when a game is designed well according to the situation and the player.

Games cannot lead the aspiring craftsman all the way, however. First off: McGonigal says that most young Americans have spent tens of thousands of hours playing games and that the skills they obtained should be put to good use. Sennett also mentions the ten thousand hour practice rule to become proficient at a craft. As I discussed before though, the procedural rhetoric of games is quite situational. This means a person would have to play an educational game about computer programming for ten thousands of hours to master that particular skill set.

It will be much more likely that a game can offer an introduction into computer programming and perhaps offer the player the means of discovering their own preferred basic learning method, as suggested by the game designers at *IJsfontein*.

Which brings me to the second problem: games are expensive to design. If done properly, desired behaviour is plotted, experts are consulted on the subject the player is supposed to learn, and effectiveness is tested after the game has been designed. This is a long and expensive process, which most educators cannot afford. The game would become particularly expensive when not only the introduction to computer programming and procedural literacy was designed, but also the entire complicated world that a person has to get to know to become a master at their craft.

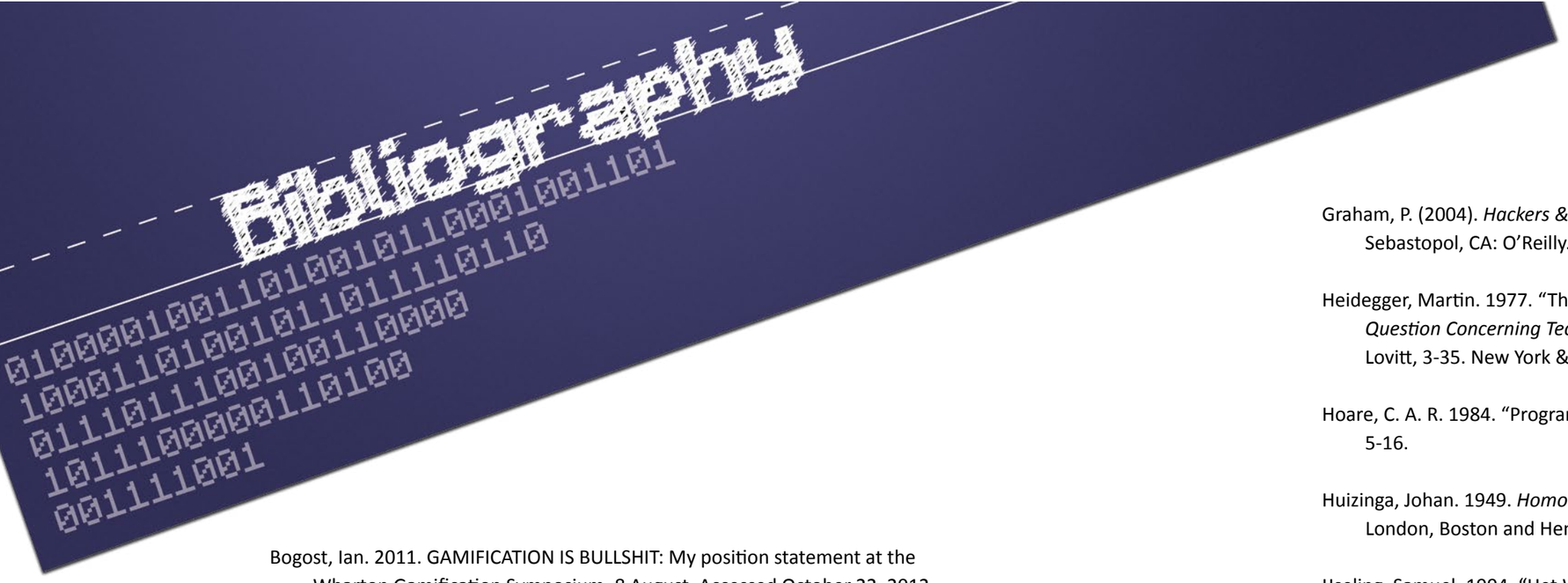
Educators also fail to see the added value of teaching through games, say Schaffer & Gee and Prensky, although this problem may lessen in the coming years. Prensky's reliance on the technical capabilities of digital natives may be overrated though, as a lot of them still lack procedural literacy.

Although games are an effective way to get someone acquainted with procedural literacy, the extent of knowledge that is required to master a field is beyond what can be incorporated in a game. The more knowledge has to be transferred intermondially, the more complicated the game becomes, say Martens, Diener & Malo. This creates hurdles both in the reception of the game across a heterogeneous field, as well as in the time it takes to transfer the knowledge that is constructive to the craft. Speaking of construction, a game will not allow the player to build objects outside of the game, unless the player makes the move towards modding the game.

Finally: games have a beginning, middle and an end. Although this makes them particularly suitable to design lesson plans, they also stop the ability of the craftsman to innovate. Like Pink says, the road to perfection, although unending, is the one the craftsman must follow to become a master at their craft.

This thesis has shed some light on three important areas: modern craftsmanship, computer programming and educational games. By providing an overview of the academic discourse and comparing it to the opinions of industry professionals, the way in which a programmer learns and executes his trade has become more clear, which could aid further debate in media studies.

I have discussed how one learns to be a craftsman programmer and in what manner this might be achieved with games. This subject would greatly benefit from further research including baseline studies. Students who want to learn programming could take different approaches and their results could be monitored. This would provide a lot of empiric data to make conclusions about what does- or does not work regarding learning programming, whether this is achieved with games or by other means. More in-depth case studies of various teaching methods or a closer look at different types of computer programming could be interesting angles for further research as well.



Bogost, Ian. 2011. GAMIFICATION IS BULLSHIT: My position statement at the Wharton Gamification Symposium. 8 August. Accessed October 22, 2012. http://www.bogost.com/blog/gamification_is_bullshit.shtml.

—. 2010. *Persuasive Games: The Expressive Power of Videogames*. Cambridge, Massachusetts & London, Engeland: The MIT Press.

Brinkmann, Svend. 2013. *Qualitative Interviewing: Understanding Qualitative Research*. Oxford: Oxford University Press.

Callois, Roger. 1961. *Man, Play and Games*. Translated by Meyer Barash. New York: The Free Press of Glencoe, Inc.

Chun, Wendy. 2004. "On Software, or the Persistence of Visual Knowledge." *Grey Room* 18: 26-51.

Deterding, Sebastian. 2011. "Meaningful Play: Getting Gamification Right." 18 February. Accessed October 11, 2012. http://www.youtube.com/watch?v=7ZGCPap7GkY.

Deterding, Sebastian, Dan Dixon, Rilla Khaled, and Lennart Nacke. 2011. "From Game Design Elements to Gamefulness: Defining "Gamification"." *MindTrek '11 Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments*. New York: ACM. 9-15.

Dreyfuss, S., & Assange, J. (1997). *Underground - hacking, madness and obsession on the electronic frontier.* Kew: Mandarin.

Gibson, William J, and Andrew Brown. 2009. *Working with Qualitative Data*. London, Thousand Oaks, CA: SAGE.

Graham, P. (2004). *Hackers & Painters: Big Ideas from the Computer Age.* Sebastopol, CA: O'Reilly.

Heidegger, Martin. 1977. "The Question Concerning Technology." In *The Question Concerning Technology: And Other Essays*, translated by William Lovitt, 3-35. New York & London: Garland Publishing, Inc.

Hoare, C. A. R. 1984. "Programming: Sorcery or Science?" *IEEE Software* 1 (2): 5-16.

Huizinga, Johan. 1949. *Homo Ludens: A Study of the Play Element in Culture*. London, Boston and Henley: Routledge & Kegan Paul.

IJseling, Samuel. 1994. "Het Wezen van de Techniek bij Martin Heidegger." In *Gegrepen door Techniek*, edited by Raoul Weiler and Dirk Holemans, 21-41. Kapellen: Uitgeverij Pelckmans.

Kay, Alan C. 1972. "A Personal Computer for Children of All Ages." *Proceedings of the ACM annual conference-Volume 1*. ACM.

Kittler, Friedrich. 1995. "There is No Software." *CTheory* 10 (18). http://www.ctheory.net/articles.aspx?id=74.

Knuth, Donald E. 2007. "1974 Turing Award Lecture: Computer Programming as an Art." In *ACM Turing award lectures*, 33-46. ACM.

Kolb, David A. 1984. *Experiential learning: Experience as the source of learning and development.* Vol. 1. Englewood Cliffs, NJ: Prentice-Hall.

Kolb, David A. 1981. "Learning styles and disciplinary differences." *The modern American college* 232-255.

Mackenzie, Adrian. 2003. "The problem of computer code: Leviathan or common power?" *Academia.edu*. Accessed February 4, 2013. Mackenzie, Adrian. "The problem of computer code: Leviathan or common power." non-pag. http://www. lancs. ac. uk/staff/mackenza/papers/code-leviathan.pdf.

2009. *Manifesto for Software Craftmanship*. Accessed april 15, 2013. http://manifesto.softwarecraftsmanship.org/.

Manovich, Lev. 2001. *The Language of New Media*. Cambridge, Massachusetts: The MIT Press.

Martens, Alke, Holger Diener, and Steffen Malo. 2008. “Game-Based Learning with Computers – Learning, Simulations, and Games.” In *Transactions on Edutainment I*, edited by Z. Pan et al., 172-190. Berlin & Heidelberg: Springer-Verlag.

McGonigal, Jane. 2011. *Reality is Broken: Why Games make Us Better and how they can Change*. New York: Penguin Press.

McLuhan, Marshall: The Extentions of Man. 1994. *Understanding Media*. Cambridge, Massachusetts & London, England: The MIT Press.

Montfort, Nick, Patsy Baudoin, John Bell, Ian Bogost, Jeremy Douglass, Mark C. Marino, Michael Mateas, Casey Reas, Mark Sample, and Noah Vawter. 2012. *10 PRINT CHR$(205.5+RND(1)); : GOTO 10*. Cambridge, Massachusetts & London, England: The MIT Press.

Nakamura, Jeanne, and Mihaly Csikszentmihalyi. 2002. “The concept of flow.” *Handbook of positive psychology* 89-105.

Nieborg, David. “Am I mod or not?—An analysis of first person shooter modification culture.” In *Creative Gamers Seminar—Exploring Participatory Culture in Gaming, University of Tampere, Finland (14–15 January)*. 2005.

Norman, Donald A. 1988. *The Design of Everyday Things*. New York: Doubleday.

North, Dan. 2011. *PROGRAMMING IS NOT A CRAFT*. 11 January. Accessed April 15, 2013. http://dannorth.net/2011/01/11/programming-is-not-a-craft/.

Papert, Seymour. 1988. “Does Easy Do It? Children, Games, and Learning.” *Game developer magazine* 88.

—. 1980. *Mindstorms: Children, Computers and Powerful Ideas*. New York: Basic Books, Inc.

Pink, Daniel H. 2009. *Drive*. New York: Riverhead Books.

Prensky, Marc. 2005. “Computer Games and Learning: Digital Game-Based Learning.” In *Handbook of Computer Game Studies*, edited by Joost Raessens and Jeffrey Goldstein, 97-122. Cambridge, MA: The MIT Press.

Rosenberg, Scott. 2007. *Dreaming in Code: Two Dozen Programmers, Three Years, 4,732 Bugs, and One Quest for Transcendent Software*. New York: Crown Punlishers.

Schäfer, Mirko Tobias. 2011. *Bastard Culture!* Amsterdam: Amsterdam University Press.

Schaffer, David Williamson, and James Paul Gee. 2005a. “Before every child is left behind: How epistemic games can solve the coming crisis in education.” Madison. Accessed October 22, 2012. http://s3.amazonaws.com/academia.edu.documents/31089769/learning_crisis.pdf?AWSAccessKeyId=AKIAIR6FSIMDFXPEERSA&Expires=1374145828&Signature=zZvuUpIsKvx%2BQS6UIAC5stM%2BluQ%3D&response-content-disposition=inline.

Schaffer, David Williamson, and James Paul Gee. 2005b. “Video-Games and Learning.” *Phi Delta Kappan* 87 (2): 104-111.

Sennett, Richard. 2008. *The Craftsman*. New Haven & Londen: Yale University Press.

SETUP. 2013. *Elegante Algoritmes*. Accessed January 30, 2013. http://elegant.setup.nl.

Slavin, Kevin. 2011. *How Algorithms Shape Our World*. Video. TED. http://www.ted.com/talks/kevin_slavin_how_algorithms_shape_our_world.html.

Smeets, Ionica. 2013. “Wiskundige Ionica Smeets over algoritmes in kunst.” *Pauw & Witteman*. Amsterdam: Vara, 19 April. http://pauwenwitteman.vara.nl/Cultuur-detail.215.0.html?&tx_ttnews%5BbackPid%5D=116&tx_ttnews%5Btt_news%5D=30373&cHash=5f3fceca7691ccb42b2ea0f0165cb59b.

Smith, Jonas Heide, and Sine Nørholm Just. 2009. “Playful Persuasion: The Rhetorical Potential of Advergames.” *Nordicom Review* 30: 53-68.

Squire, Kurt. 2005. “Changing the Game: What Happens When Video-Games Enter the Classroom?” *Innovate: Journal of online education* 1 (6).

Susi, Tarja, Mikael Johannesson, and Per Backlund. 2007. “Serious Games – An Overview.”

Tötösy de Zepetnek, Steven. 1998. *Comparative Literature: Theory, Method, Application*. Amsterdam: Rodopi B.V.

Turner, Daniel W. III. 2010. “Qualitative Interview Design: A Practical Guide for Novice Investigators.” *The Qualitative Report* 15 (3): 754-760.

Van Eck, Richard. 2006. “Digital Game-Based Learning: It’s Not Just the Digital Natives Who Are Restless...” *EDUCAUSE Review* 41 (2): 16-30.

Victor, Bret. 2012. *Learnable Programming: Designing a programming system for understanding programs*. September. Accessed July 11, 2013. http://worrydream.com/LearnableProgramming/.

Wesener, Stefan. 2006. “Spielen in virtuellen Welten: Übertragung von Inhalten und Handlungsmustern aus Bildschirmspielen.” Accessed July 20, 2013. http://www.thomasbauer.at/tab/media/pdf/paed/expose/praxis_artikel10b.pdf.

Zyda, Michael. 2005. “From Visual Simulation to Virtual Reality to Games.” *Computer* 38 (9): 25-32.

Affordance
Affordance concerns the options available to a certain subject. What actions does the object lend itself to? (21)

Algorithm
An algorithm is a mathematical equation which formulates the actions to be processed in a computer. (26)

Assembly
Assembly are lower-level programming languages that have a very direct correlation between the code typed and the actions performed on the machine. (60)

Basic Learning Styles
The basic learning style is the manner in which a person prefers to learn. The experiential learning cycle depicts the different stages of learning. Dependent on their preferred learning stage, a person learns by accommodating, diverging, assimilating or converging. (47)

Behaviourism
Behaviourism is a didactic approach whereby knowledge is transferred through a positive- and negative reinforcement system. In game-theory, behaviourism is associated with the thought that games have a direct effect on the player’s behaviour. (48)

Beta
Beta is a term often used in software to indicate a product is usable, but is most likely still in need of improvement. Users can indicate bugs in the software which programmers can then fix. (Front)

CAD
Computer Aided Design. (21)

Code
Code or computer code, is the language that is used to build digital objects and programs. Code is written by programmers and interpreted by the computer. (31)

Code Academy
An educational platform that teaches people computer program while employing gamification techniques. (58)

Coding
Coding is writing code. See programming. (26)

Constructivism
Constructivism is a didactic approach whereby an individual's context is taken into account in constructing their own unique learning approach. (48)

Craft
Craft is a creation process: human intervention that causes certain materials to be transformed into objects of use. (17)

Digital
As opposed to analogue, digital concerns new media objects. (34)

(Digital) Game-based learning
(Digita) Game-based learning is education based on games. These can be digital video-gamesvideo-games or board-games, the use of game mechanics to inspire learning is what matters. (50)

Elegant Algorithms
Elegant Algorithms was a contest at the start of 2013 by cultural media lab SETUP. In it programmers were asked to recreate a Mondrian painting in code. This contest, co-organized by the author, led to the topic for this thesis. (10)

Experiential Learning Cycle
Designed by David A. Kolb, this model shows the cycle a person goes through when they are learning. The four stages include experiencing, observing, conceptualizing and experimenting. A preference on any one of these stages indicates their basic learning style. (47)

Flow
Flow is a state of being where a person is completely absorbed in their task. Flow leads to intense focus and allows one to practice their art. (18)

Hackers
Hackers are expert computer programmers. Hackers are not to be confused with the criminally intent crackers. (35)

Hacker culture
Hacker or programmer culture entails the hierarchy within programmer culture, wherein only the truly talented and technically apt are considered hackers. It also involves a fair bit of geekiness. (39)

Hakitzu
Hakitzu is an iPad game in which players control virtual robots by typing in JavaScript, a programming language. (60)

Hardware
Hardware is the physical part of a computer. The circuit boards on which electrical impulses race back and forth, the casing, the monitor, et cetera. (26)

Higher-programming languages
Higher-programming languages are the abstracted forms of languages most programmers use to code today. (26)

IJsfontein
A serious game development company from the Netherlands. (15)

Intermondial knowledge-transfer
Intermondial knowledge-transfer is achieved when someone can use knowledge obtained in the game outside of the game world. (52)

Intramondial knowledge-transfer
Intramondial knowledge-transfer is achieved when someone can use knowledge obtained in a game in another game. (52)

Linux
An open source operating system. (17, 20-21)

Ludus
Ludus is the one end of the spectrum of games versus play. Ludus entails games, which means structured forms of play with rules and goals. (45)

Gamification
Gamification is the addition of game characteristic to a non-game context. (56)

Paidia
Paidia is the other end of the spectrum of games versus play. Paidia is a pure form of play, which is unencumbered by rules. Free-form play and pretend could be classified as paidia. (45)

Play
Play is a trait inherent to culture, or even preceding it. Play is separate from 'real life' and provides people with an escape of sorts. (44)

Procedural literacy
Procedural literacy is the competency to understand processes and concepts, mostly associated with computational literacy but not necessarily limited to it. (36)

Processing
A programming language that emphasises aesthetic appropriation. (28)

Programming
Programming means translating human thoughts and wishes into a language that can be processed by a computer. Programming means writing code, which is both a text as it is a process, as soon as it is interpreted by the machine. (26)

Open Source
Open source means sharing knowledge openly. Though this is not restricted to the software industry, it did originate there. By creating open source materials, coding becomes a social activity wherein expertise is shared. (21, 40)

OS
Short for operating system. (27)

Qlvr
A serious game company based in Utrecht, the Netherlands. (15)

Scratch
A programming language specifically aimed at children. (37)

Scratchweb
Aweb based platform designed to teach children computer programming. (15)

SETUP
A cultural media institution based in the Netherlands, where makers, lovers and co-operators of digital- and cultural society are brought together. (10)

Serious games
Serious games are games that are created with a purpose other than entertainment. Whether entertainment is still a part of serious games or not, is a subject under discussion. Serious games are most often applied in business- or educational contexts. (50)

Software
Software is all the content of a computer that cannot be directly traced to its physical components (hardware). Examples are computer code and applications. (31)

Mass-customisation
Mass-customisation means objects can be produced on a mass-scale, yet also specified to each specific user. An example are computer programs that can be run indefinitely, but each time its process its run it portrays a different visual outcome. (34)

Modding
Modifying a video game's code, thereby adapting the rules or content. (60)

New media objects
New media objects differ from their predecessors on a couple of levels. They are based on numeric values, they are modular, variable, automatable, and can be transcoded. (34)

Video-games
Games that are played digitally, mostly on a computer or console. (11)

Winvision
A software development company. (15)

Appendix B: Interviews

Transcript interview Sergio van Pul, Scratchweb

Interview date: June 17, 2013, 15:00 - 15:36
Location: *Seats2Meet*, Utrecht

1. [0:03] Nou vertel, wat doe je precies?

Nou waar ik voor op ben geleid is game design, op de kunstacademie, op de *HKU* [*Hogeschool voor de Kunsten Utrecht*]. En eigenlijk, wat je zegt [ik heb het onderwerp van mijn scriptie ingeleid], van die deling van gamedesign plus programmeren is in de praktijk vaal veel meer hybride. Vooral als je in een klein bedrijfje werkt, of als freelancer, want dan moet je gewoon alles doen. Zowel concept, als visueel design, als programmeren, om het productje af te maken. Dus ik spring een beetje heen en weer tussen alle verschillende disciplines.

Ik heb dus een tijd gefreelanced, daar ben ik eind dit jaar mee gestopt, helaas, door financiële beperkingen, ik had te weinig opdrachten om rond te kunnen komen. En met Scratchweb ben ik betrokken geweest ook min of meer als vrijwilliger en mede-ontwikkelaar, en daar doe ik gewoon af en toe nog dingen voor. Dus eigenlijk om de website opnieuw te herstructureren en op te bouwen.
Ik heb mee gedraaid in de workshops, dus praktisch kinderen aanleren om met Scratch spelletjes en animaties te maken. En af en toe doe ik mee met evenementen, dus gewoon voor publiciteit. Dat is meer business to business om te laten zien dat wij met Scratch mooie dingen maken, als Scratchweb. We willen dat bedrijven en scholen en andere instellingen van ons weten, en weten hoe interessant het is, en dat ze ons kunnen inhuren, om daar gewoon baat bij te hebben, voor personeel of voor kinderen die opgroeien en dan aan het werk komen bij zo'n bedrijf.

2. [1:22] Ok, dus je programmeert zelf alleen in Scratch?

Nee, ook op andere dingen. Scratch ben ik eigenlijk ingegroeid vanuit een workshop. Ik kreeg een vraag doorgespeeld van een ouddocent: 'Een school wil ons in een workshop met creativiteit, games, en voor klasleerlingen, voor dertien- veertienjarigen. Waar ga je dat combineren?' Bijvoorbeeld C++ is echt niks in het begin en Flash is ook al ingewikkeld. Hoewel de meesten het wel kennen van de voorkant, van het spelletje spelen op internet. Maar om ze die zelf te laten bouwen is alweer een drempel. Ook om het op zo'n school geïnstalleerd te krijgen. Scratch is dan een hele goede instap, om heel makkelijk, en heel snel te leren hoe je programmeert. Zo doende ben ik met Scratch in aanraking gekomen, hoewel ik ook werk met ActionScript, Flash en een aantal andere talen.

3. [02:11] Mag ik vragen hoe je die geleerd hebt?

Vooral, eerst gewoon zelfstudie en daarna door geleerd via mijn opleiding. Flash heb ik eigenlijk grotendeels zelf aangeleerd. Daarmee een begin gemaakt, voor ik aan de opleiding begon heb ik zelf gewoon boeken uit de kast gepakt en ben ik dingen gaan maken. In het begin animatie-achtige dingen en daarna steeds meer ook spelletjes, dus programmeren erbij. Dat doorgelopen en gewoon simpele techno-demo's gemaakt en pong-spelletjes. Gewoon zien hoe het werkt. Gewoon proberen en imiteren. Dat is eigenlijk de basis.

4. [02:47] In hoeverre is dat een sociale activiteit? Werk je samen met andere programmeurs, heb je nog les gehad ernaast, of is het meer in je eentje een beetje spelen en opzoeken?

Ik heb veel dingen toch in mijn eentje gedaan. Misschien dan nog fora gebruiken om dingen na te vragen of om iets op te zoeken wat mensen al hebben uitgezocht. Veel problemen zijn al bekend in die grote wereld. Dan is het alleen een kwestie van de goede vragen stellen, die in *Google* intikken en dan vind je meestal wel een antwoord of *een* methode. Soms vind je er ook drie of vier en dan mag je uit gaan zoeken wat de meest effectieve is. Wat het meest past op de situatie.

5. [03:32] In hoeverre kan je een programmeertaal vergelijken met 'gewone' taal? Is dat een zelfde manier van leren, of is het een hele andere manier van denken die je nodig hebt?

Het is eigenlijk een combinatie van taal en wiskunde. Het is inderdaad deels taal en grammatica, van hoe bouw je dingen op in een volgorde zodat het een logisch geheel vormt? En dat het begrepen wordt, in dit geval door de computer. De computer is dan veel stricter dan de mens, zeker in spreektaal ten opzichte van programmeer-talen. Schrijftaal is al veel formeler. Wil je een net boek schrijven, dan moet dat aan bepaalde regels voldoen. In dat opzicht is het met programmeren vergelijkbaar. Ik ben ook meer een logica-programmeur dan een wiskunde-programmeur. Ik schrijf meer logische verbanden op, hoe die aan elkaar te knopen, of met visuele trucjes dingen duidelijk maken, dan dat ik hele moeilijke wiskundige formules ga uitdenken. Dat zijn twee benaderingen die je kan kiezen. Het zwaartepunt kan dus de ene of de andere kant op vallen.

6. [04:28] Vind je het belangrijk dat dit ook meer aangeleerd wordt? Dat mensen wiskundiger en logischer leren denken?

Ja, dat is heel belangrijk omdat alles op computers draait tegenwoordig. Ik bedoel, een telefoontje draait op een computer, en een iPad, maar ook allerlei apparaten om je heen waarvan je niet eens weet dat er een computertje in zit. Een klok, of een magnetron, dienstroosters van allerlei instanties, dat draait allemaal op een computer. En wat je nu vaak ziet: 'oh, het doet het niet, het is stuk, stom apparaat', en vaak is het dan 'oh, iemand heeft een foutje gemaakt'. Misschien kan je het repareren, als je weet hoe het werkt.

7. [05:05] En heb je ideeën over hoe we dat soort media-wijsheid zouden kunnen vergroten? Scratch is bijvoorbeeld een goed instapmiddel. Is dit het enige wat je nodig hebt, of zouden er ook andere methoden moeten zijn?

Ik denk dat het meer praktisch geleerd moet worden, vooral door het *doen* van dingen. Je ziet dat mediawijsheid op scholen vaak stopt bij: 'kijk jongens, dit is internet, dat is *Google*, *Google* is gevaarlijk,' en zo kun je niet dingen opzoeken om dus jezelf of anderen te beschermen. Hoe ga je om met dingen als *Facebook* en sociale media? Niet meer op internet pesten, dat soort sociale dingen worden aangeleerd. Maar de techniek blijft een beetje achter. Dat komt vaak ook omdat de docent zelf vaak niet weten hoe het werkt. Die kunnen dus zelf niet die kennis overdragen.
Scratch is inderdaad wel een voorbeeld, er zijn er inderdaad meer, die je onder de motorkap laten kijken en je eigen laten bouwen. 'Bouw zelf een website, of een animatie,' dat je weet hoe het van binnen uit werkt. En dan ga je veel meer, inderdaad, door het scherm heen kijken. Weten *waarom* bepaalde dingen niet werken of waarom ze op een bepaalde manier georganiseerd staan. Omdat je zelf dingen gedaan en ervaren hebt.

8. [06:16] Denk je dat het leeftijd gebonden is, wanneer iemand met programmeren zou moeten beginnen?

Dat is lastig. In principe niet, je ziet ook, het ligt ook aan interesses. Sommige kinderen beginnen al vanaf zes, zeven jaar van: 'papa is IT'er, dus wij willen ook op de computer', en die rollen daar vanzelf in. En andere kinderen op hun vijftiende, zestiende, nog *steeds* niet achter een scherm krijgen. Als ze namelijk interesse hebben om te weten hoe het werkt...
Tja, als je hebt hebt over op welk niveau moet het in principe op school aangeboden worden? Misschien met een jaar of tien, twaalf, dat je een begin kan maken. Zoals je bijvoorbeeld ook begint met iets als Engels gaan leren, dat is ook net een stapje verder dan basis overlevingstechnieken, ook omdat je... [7:04 - 7:09 niet te verstaan] ... dat is een stapje verder om meer gespecialiseerde dingen aan te leren.

Ja en verder is het een kwestie van inderdaad 'hoeveel belangstelling is er?' en 'wat is de specialisatie?' Van een aantal kinderen kun je waarschijnlijk heel ver gaan en heel gecompliceerde dingen maken, ...[7:25, lastig te horen]... heel andere annotaties zullen ze genoeg hebben en vinden ze het ook niet leuk meer, zo van 'ik ga toch de andere kant op en ik hoef er niet meer over te denken'.

9. [07:34] Hoe kom je daar dan achter, is dat iets wat onmiddellijk duidelijk wordt? Of is dat een proces dat als je bijvoorbeeld met Scratch aan de gang gaat en je dan zo'n workshop doet, hoe pik je diegene eruit die er gewoon voor gaan?

Ja dat zie je vanzelf door de vragen die je terug krijgt. De kinderen die willen stellen vragen. Die zetten mij aan het werk omdat ik dingen voor ze uit moet gaan zitten. 'Hoe moet ik dit maken, hoe moet ik dat maken.' De meeste fanatiekelingen willen meteen Super Mario namaken. 'Ik wil een platform spel, met powerups en vijanden en dit en dat.' 'Ok jongens, één ding tegelijk. Laten we eerst eens met één stapje beginnen.' En andere kinderen heb je dan die zitten van: 'Oh leuk, *Google*! of, *Facebook*!' en die zijn dan afgeleid en die vragen ook niets meer dan 'Nou meneer, wat moet ik doen?'. Die hebben meer sturing nodig. Dat is dan nog steeds soms de manier waarop het kwartje valt. Eens even van 'Oh wacht, maar ik kan wel gewoon mijn eigen poppetjes tekenen en in zo'n scherm tot leven wekken. Hey, dat is leuk!' Dan heb je ze dus te pakken op iets wat hen al aanspreekt. Bijvoorbeeld dat tekenen.

10. [08:30] Denk je dat dat dan ook te maken heeft met dat ze die controle over hun eigen wereldje hebben?

Dat helpt wel ja. Het helpt om dingen te doen. Het zelf dingen doen, zelf kunnen bouwen, dat stimuleert voor veel kinderen. De meeste kinderen vinden het sowieso leuk om iets te maken, ...[8:47 slecht te verstaan]... voor het gebeurt is, dan ga ik puzzelen. Op het moment dat er een computer in huis is, dan trekt dat heel veel aandacht. Dan leidt dat soms af van hetgeen wat je echt wil leren omdat het dan is van 'oh de computer is voor spelletjes, Facebook en al die andere leuke dingen', die vooral afleiding zijn en waarmee ze niet zelf mee dingen produceren. Niet echt aan de gang gaan. Als producent in plaats van consument. Dat is een soort ding wat je moet kunnen doorbreken. Die computer is niet belangrijk. Misschien moet je ook gewoon leren programmeren met papier en pennen en wat pionnen. 'Ga maar eens een spelletje bouwen doormiddel van knipseltjes en tekeningetjes', en dat is eigenlijk die basis van die logica opzet.

11. [09:27] Naar ik begrijp is Scratch zo ook opgebouwd, toch? Dat je onderdelen neemt en die naast elkaar plaatst, omdat dat dezelfde denkwijze moet gaan triggeren die je ook bij andere programmeertalen gebruikt?

Ja. Ja, het is natuurlijk een soort visuele programmeertaal hè? Dus je gebruikt ... [9:41 slecht te verstaan]... dingen als sprites, als plaatjes, en daar plak je logica blokjes aan vast. Een beetje zoals je legoblokken op elkaar klikt. En dan zeg je van: 'Als ik op een knop druk, dan gaat deze die kant op lopen, en als hij tegen die wand botst, oh dan keert hij weer om.' En dat is dan een heel simpel stapje om iets heen en weer te laten marcheren. Dat kun je dus heel simpel houden, en daarna kun je het heel complex maken. Daana kun je hele spellen simuleren, die nu ook gewoon op de markt te koop zijn. Ze hebben manic man [?] gebouwd, en race spelletjes gebouwd, en zelfs dingen als *Portal* nagebouwd. Portalen die je afschiet waar je dan doorheen kan springen.
Goed, dat kunnen ze dus nabouwen in Scratch, terwijl dat dus plat en 2D is en heel veel beperkingen heeft, en dan is het een kwestie van door willen gaan en weten waar je fans liggen. Hoe zo'n systeem werkt, wat de beperkingen zijn en de mogelijkheden die ze hebben. Als je dat kan, dan heb je een programmeertaal geleerd. Dan kun je gaan programmeren.

12. [10:43] Dit waren voornamelijk vragen over het leerproces. Ik ben ook benieuwd naar het werkproces als je zelf programmeert. Kan je me vertellen, als je bijvoorbeeld iets in Flash maakt: wat doe je?

Ik begin in ieder geval met het idee. Specifiek is dat voor mij dan gamedesign. Het spel. Wat zijn de elementen? Wat zijn de regels? En die eerste stap is nog opbreken, dat is nog een stukje gamedesign werk. Wat zijn de karakters die ik inzet? Wat zijn de power-ups, hoe zien mijn levels er uit? Is het 2D, is het 3D? In wat voor aanzicht zit ik? Vanaf de zijkant of van bovenaf gezien? In wat voor structuur ga ik dat programmeren ingieten?
Dan is het zaak om op te delen in objecten. Object-georiënteerd programmeren. [De sprites?] Nou, dat is niet eens zo zeer de visuele dingen, maar ook logica dingen. Dat kan ook iets zijn om botsingen te detecteren. Als je in een spel heel duidelijk moet weten waar de muren zitten, dan kun je een soort virtueel 'ding' maken, wat die muren aantastten, zodat je weet dat je er nooit tegenaan loopt. Dat zie je nooit, als speler, maar het is wel een soort logica blok, wat je in bouwt en wat je er aan vast plakt aan zo'n zichtbare sprite.
Dat zijn dus onderdelen die je eerst definiëert, en dan ga je ze één voor één uitwerken.
Je begint met het belangrijkste, namelijk datgeen wat je zelf bestuurt, het speelkarakter. Dat je de knopjes aan elkaar kan koppelen, dat die kan bewegen. Een volgende stap is dan vaak de ruimte eromheen een beetje vormgeven, als is het maar een soort voorbeeldomgeving. Zo van waar zijn jullie en waar kan je wel en waar kun je niet bewegen? En dan komen dingen als vijanden, power ups, en dat soort dingen als dos elementen. En uiteindelijk plak je het aan elkaar ...[12:30 luidruchtig lawaai wat gesprek verstoord] ... op elkaar reageren. En dan is het vooral heel veel testen. Heel veel trial en error. En kijken of het werkt.

13. [12:37] Maar dat is echt het schrijven van de code. Hoe komen dat soort ideeën tot stand voordat je begint te bouwen? Doe je dat analoog, typ je dingen uit, teken je dingen?

Ik teken heel veel dingen. Ik maak ook vaak design-documentjes. Dat is vooral uitgeschreven, de logica van het spel. In gewoon bulletpoints: dit zijn de objecten, dit zijn de speldoelen, dit zijn de acties die je karakter kunnen doen, en dan heb je een soort logisch schema, wat al min of meer abstract is. Dan hoef je alleen nog maar uit te denken hoe je dat gaat vertalen naar programmeercode. Kan iets lopen? Kan iets vliegen? Kan ze schieten of niet? Dan kan je het naar blokjes vertalen. Dan kan je ook zien: ok, dit is een ding wat schiet. Dat is ook een ding wat schiet, dus ik kan daar een centraal blok voor bouwen wat zij dan beide kunnen gebruiken. Dan hoef ik alleen maar ergens een naam aan te passen. Wat dan achteraf kan veranderen in een spel.

14. [13:29] En werk je dan voornamelijk thuis? Heb je ergens een werkplaats?

Nee, ik werk voornamelijk vanuit huis. Dat is het makkelijkste. Mits je niet afgeleid raakt, natuurlijk.

15. [13:45] Is het belangrijk om code uit je hoofd te leren? Je weet de basisvorm van de programmeertaal, maar hoeveel van de daadwerkelijke code komt uit je hoofd: probeer je gewoon uit of zoek je op?

In mijn geval is het heel veel opzoeken. Uit je hoofd leren is eigenlijk niet belangrijk, want alles staat op het internet of in je computer. Van gewoon het help-document bij een programmeertaal tot aan alle fora op het internet. Als je nou ActionScript als voorbeeld neemt, de 'help' zelf, daar heb ik ook heel veel uit gehaald. Gewoon weten van, ok, ik wil iets doen, daar is vast een functie voor. Als ik dan in het help-document intik wat ik wil doen, dan vind ik vast die functie wel. Of ik kijk gewoon de lijst door met alle acties. Dan ga je zien: dit is de functie, dat komt er uit. Daar tussenin gebeurt er iets. Op die manier kun je stukjes aan elkaar knopen. Zo heb ik heel veel dingen vanaf het niets opgebouwd. Voor complexere problemen ga je op het internet kijken, om dan op fora na te vragen: 'Ik wil iets moeilijks in 3D doen. Hebben jullie goede tips over hoe je dat efficiënt oppakt? Welke blokken heb ik dan nodig?'

16. [14:52] Het is dus een heel sociaal proces?

Ja, je hebt wel interactie, maar het is niet real-time. Het gaat op afstand en je gaat heen en weer ping pongen. Soms start je een dialoog met iemand. Als iemand iets geeft en je hebt daar vragen over. En heel vaak is het gewoon opzoek werk. En dan eventueel een bedankje schrijven: 'Hey tof, het werkte, en hier is mijn link, van wat ik heb gemaakt.'

17. [15:11] Dus dat gebeurt op semi open-source basis? Je stuurt alleen je code op?

Tja, dat ligt eraan waar je het vandaan hebt. Heel veel commerciële bedrijven zullen dat soort van gesloten houden, dan doe je dus niets terug. Maar heel veel indie developers, die delen stukjes. Natuurlijk niet het hele spel, maar gewoon stukjes code en oplossingen die ze bedacht hebben. Dat wordt vrij gedeeld. Dat wordt ook niet gezien als waar je je geld uit haalt, want dat zijn maar blokjes. Alsof je één lego-steentje geeft en zegt: "Nou, speel er maar mee!" Dat werkt niet. Pas als je hele dingen hebt, dan kan je er wat leuks mee doen.

18. [15:44] Heb je dan ook bepaalde bronnen waar je vaak naar terug komt voor mogelijke oplossingen en dergelijke?

Dat wisselt wel een beetje afhankelijk van het platform. Stack Overflow is een hele bekende voor alle programmeertalen. Dus heel veel ActionScript, maar zelfs dingen voor Scratch als je goed zoekt. Scratch heeft ook zijn eigen community site met allemaal voorbeelden en oplossingen. Met ook voorbeelden van spellen die je dus wel helemaal open kan breken en kan bekijken. Als ik iets zoek in een bepaald genre dan zoek ik voorbeelden van dat spel. Dan kijk ik naar 'heeft die een leuk uitziende voorkant' 'lijkt die er veel op', en dan ga ik kijken hoe ze er vanuit de binnenkant uit zien. Of dat makkelijk te doorzien is en toe te passen, dat je een paar stukken code overneemt. En dat geldt ook heel veel voor wat meer commerciële games. Dat je stukjes code overneemt als je ze kan vinden. Maak je een platform game, dan hoef je die niet specifiek van de grond op te bouwen. Er zijn alweer platform engines. Mensen die iets hebben geschreven en gewoon een deel van de code gebruiken, dan hoef je alleen wat plaatjes aan te passen en wat logica toe te voegen wat het concept eigen maakt.

19. [16:58] Is een werk dan iets van jezelf? Signeer je het?

Het eindproduct is wel inderdaad míjn werk. Daar staat duidelijk mijn bedrijfsnaam op, mijn eigen naam op, gewoon 'gemaakt door'. En als ik een groot deel heb overgenomen staat er bij: 'mede mogelijk gemaakt door'. Sommige programmeurs vragen dat ook, die hebben ook een soort rol van 'online docent' zonder dat ze daar iets voor rekenen. Die vragen gewoon 'Als je mijn code gebruikt, mijn hele uitgewerkte functie, zet er dan boven waar je het vandaan hebt.'

20. [17:40] Krijg je dat alleen in de uiteindelijke game te zien of staat dat in de code?

Het kan gewoon in de code staan, in zo'n headertje. Veel van die codes beginnen met zo'n header: 'deze code is gemaakt door... op datum...'. Het is vaak fijn als je dat laat staan. Dat je het verzamelt bovenaan de code. Dat je weet: ik heb de onderdelen gebruikt van een aantal [...18:00-18:02 slecht te verstaan door piep...], ook als je je code zelf door zou geven, of iemand anders kijkt er eens naar, dat je ook kan nagaan waar het vandaan komt. Stel ik heb iets gemaakt en er zit een verschrikkelijke bug in, en ik krijg die er niet uit, ik geef hem door aan een andere programmeur om er nog eens naar te kijken, dan is het fijn dat ik weet waar ik al die logica vandaan heb gehaald. Of het mijn eigen idee was, of dat ik het ook van een andere bron heb overgenomen.

21. [18:23] Dus dan komt er nog een hoop studie bij, dat je kijkt: waar komt het vandaan en hoe is die logica opgebouwd? Of kijk je gewoon hoe de code er uit ziet?

Ik kijk in eerste instantie naar de code. Of het werkt of of het niet werkt. Soms is het makkelijk om een andere code op te zoeken die wel werkt, of dat je door hebt na een tijdje: 'dit doet het niet' of 'dit is gewoon zo onlogisch zonder comments'. Dat is dan heel vervelend. Het is leuk als het aangeboden wordt en als het echt werkt is het prima, als je een soort black box hebt waarvan je zegt: 'nou ik plak dit er in en ik gooi er dingen in en ik krijg er uit wat ik wil', maar als je dingen moet aan gaan passen, heb ik ook gedaan, dat zo'n functie wordt open gebroken voor zo'n object. En dan is het van: 'uhm ok, ik heb ongeveer door wat je van plan bent, iets met richting en dingen bewegen, of het omrekenen van het één naar het andere model,' maar dan is het soms heel erg zoeken om een variabele aan te passen. Dan moet je regel voor regel uit gaan zoeken: 'waar vindt die wisseling nou plaats, om juist dat ene stukje te doen wat ik van plan ben?' Het is dan makkelijker als ze er zelf even bij zetten: 'dit stukje code is om dingen op te zoeken, dit stukje code is een berekening, dit stukje code voert iets uit naar de rest van het programma.'

22. [19:33] Dus de structuur binnen de code is heel belangrijk?

Ja, dat is te vergelijken met de hoofdstukken in een boek. Bij paragrafen en hoofdstukken geef je alles een duidelijke titel. Zodat je weet, vooraf, waar je hoofdstuk over gaat.

23. [19:53] Hoe kan je zien of iets goede of slechte code is?

Vaak in eerste instantie door dat scannen. Staat er voldoende commentaar bij? Is het duidelijk? Zitten er bepaalde standaard elementen in de code, bijvoorbeeld manieren van opbouwen? Heeft iemand zijn dingen in functies opgedeeld? Verwijzen die functies goed naar elkaar? Zijn de variabele namen uitgebreid en logisch? Er komt bijvoorbeeld soms hele goede code uit, maar dat alle variabele namen afkortingen zijn. 'Ski A ES, IMG...' Voor zo'n programmeur is het dan heel logisch wat de hele woorden zijn die bij de code horen, maar als je dat overneemt, ja het is maar niet wat populair voor zo'n programmeur is.

24. [20:35] Heeft de kwaliteit van de code dan alleen maar te maken met hoe duidelijk die is of heeft dat ook te maken met functionaliteit?

Ook met functionaliteit, hoe effectief die is. Het zijn twee dingen, het ligt er maar aan wat je er mee wil. Als je alleen een black box wil die werkt, ja dan maakt het niet uit dat de variabelen zijn

afgekort. Als dat gewoon je ding is en dan werkt het gewoon heel efficiënt. Dat kan dus ook, dat je hele efficiënte code bouwt, juist door het in te pakken, dat je heel weinig tekens gebruikt en verschillende rekentruuks, zodat je computer in een fractie van een seconde een hele ingewikkelde berekening uit kan rekenen. Dan is het efficiënte code, en dan zou je hem kunnen gebruiken, maar dan moet je weten dat je er niets meer mee kan doen. Behalve alleen maar invoeren en uitvoeren.
Wil je een code gebruiken die je wil kunnen doorzien, veranderingen in toepassen, dan is het fijn als je standaard methoden gebruikt om te beschrijven: 'oh je schrijft je variabelen uit.' Zorg dat er in je variabelen staat wat het ding doet. Een uitgeschreven zinnetje, is het dan eigenlijk. Ook voor je functies, dat er staat, dat er staat wat de functie doet, en er boven staat waarom het zo is gedaan en waar die naar toe verwijst. Dus dat zijn twee afwegingen. En iedereen zal je waarschijnlijk iets anders vertellen, wat goed en wat slecht is. De één schrijft graag hele efficiënte code, wat heel snel door je computer heen loopt. De ander zegt: 'Ik wil wel weten hoe het werkt en hoe het van pas komt'. Ik vind het wel fijn om te weten hoe ik dingen moet toepassen, ook om ze te kunnen veranderen.

25. [21:56] Heb je ook wel eens een code gezien waarvan je van onder de indruk was? Heb je daar een voorbeeld van?

Nee, ik zou zo geen voorbeeld kunnen noemen. Ik heb wel eens dingen zien passeren waarvan ik denk: 'Oh, cool dat iemand dat gemaakt heeft!' Dat je er dan niet over na hoeft te denken, dat ik niet urenlang of dagenlang op een functie zit, want ik wil dit of dat bereiken, het moet simpel zijn, maar ik kom er niet uit. Dat je dan soms iets tegen komt van: 'oh ja, zo had ik het nog niet bekeken.' Dat is dan eigenlijk heel simpel en heel handig. Maar ik heb nog nooit een code geïntegreerd door het simpel over te nemen, vaak is het dan een tweespel.

26. [22:40] Dus dan pas je het eerst aan naar gelang je eigen wensen?

Ja, ik gebruik er vaak dan stukjes uit, in plaats van de hele code, een heel werk. Ik ben ook min of meer half game designer, half programmeur. Voor mij is het vaak zo dat als het werkt, is het goed genoeg. Ik ga niet voor de bleeding edge prestaties.

27. [23:07] Ik had het al eerder over de term ambacht. Wat denk jij dat dit inhoudt en denk je dat een programmeur een ambachtsman is?

Een ambachtsman is iemand die kennis heeft van een discipline, van een vak. Dat ook uitvoert. De vraag blijft: is er in ambacht ook sprake van vernieuwing? In traditionele ambachten is het vaak zo dat dingen gebeuren volgens een geeikte manier en daar gebeurt niet zoveel meer. Er is gewoon vastgesteld wat de regels zijn en de leerdoelen. Met programmeren, wat nog steeds gebonden is aan computers en ICT, wat heel erg snel ontwikkelt, is dat iets wat misschien wat minder traditioneel ambacht is. Wat toch meer onderzoek... Hoe moet je het dan noemen? Meer een soort onderzoekswetenschap. In het verleden was er een bepaalde manier van werken, object-georienteerd programmeren dat is er in de loop van de jaren tachtig ingebouwd van 'oh, dat is handig om dat zo te doen, om dingen op te delen in plaats van één lange code aan elkaar te schrijven.' Dat werkt nu nog steeds. Je ziet dus nu langzaamaan weer computers ontstaan waardoor dat idee weer gaat rafelen. Dat codes van servers komen en niet in één pakket staan, maar dus van allerlei verschillende plekken komen, of er dingen gestreamt worden, de variabelen zijn dus niet meer hetzelfde. Het is een andere manier van code gebruiken.
Maar goed, je hebt wel mensen die daar mee bezig zijn en het echt gebruiken, maar het gros van commerciële code dat is nog gewoon object georiënteerd.

28. [24:47] Dus je zou programmeren wel als ambacht beschouwen, behalve dat vernieuwende aspect?

Ja, ik denk dat dat wel iets is wat er van afwijkt. Maar dan is de vraag: zijn ambachten ook niet vernieuwend? Een timmerman is ambacht, maar daar bestaan natuurlijk ook nieuwe gereedschappen voor. Daar zit dan ook nog enigszins vernieuwing in. Aan de andere kant, een stoeltje is nog steeds hetzelfde. De techniek verandert, en ambacht verandert dus ook.

29. [25:15] Neem ActionScript nou als voorbeeld, heb jij daar veranderingen meegemaakt sinds jij er mee begonnen bent?

ActionScript is sowieso redelijk veranderd. Het is begonnen als animatieprogramma, dus eerst was het heel simpel. Dan kon je zeggen: 'start- of stop filmpje'. Daaromheen werden nog andere functies aangeboden. Dingen die wat dynamischer lijken. Toen werd het steeds dynamischer en dynamischer met de versies die volgden, zodat ze uiteindelijk een hele grote spaghetti aan code hadden met allerlei functies die naar elkaar verwezen en elkaar volgden. Op een gegeven moment zijn ze over gestapt van versie twee naar versie drie en hebben ze gezegd: 'ok, we gaan het even schoon vegen en we gaan al die functies opnieuw definiëren.' En nu is het opeens een object georiënteerde code geworden die in één keer veel efficiënter en veel makkelijker werd, om dingen door te lezen. Toen ben ik er ook veel meer mee gaan doen, omdat het toen opeens te begrijpen viel. Daarvoor moest je vaak hele lange omslachtige regels schrijven om iets heel simpels te realiseren.

30. [26:14] Maar als je zo'n taal aanpast ga je weg van machine-language en richting de higher programming languages. Wordt het dan allemaal niet steeds abstracter?

Ja, in principe wel.

31. [26:28] Is dat dan niet erg? Dat je minder weet over wat je precies doet binnen de machine?

Nee, volgens mij niet, want bijna niemand weet wat hij doet binnen de machine. Ik weet absoluut niet hoe binaire code werkt. Ik weet dat het bestaat. Ik weet dat het met nullen en éénen is, en ik weet hoe zo'n hexadecimaal stelsel rekent, maar ga mij niet vragen om bits te veranderen, want dat leidt tot chaos. Er zijn misschien een handjevol programmeurs die dat wel kunnen, op dat niveau programmeren, dus echt de machine aansturen, maar voor de meeste programmeurs is het makkelijker dat dingen gewoon in functies zijn ingedeeld.

Waarbij je dus doorverwijst naar en voor de grote menigte. Als je dan zegt: 'iedereen zou een beetje programmeur moeten zijn', is dat dan ook gewoon heel makkelijk. Als je gewoon een functie krijgt met een duidelijke beschrijving, dat je weet wat die wel en wat die niet kan. Binaire code hoef je niet te kennen.

32. [27:20] Maar hoe zie je dan het verschil tussen een professionele en een amateuristische programmeur? Of is dat verschil er niet?

Dit is grijs gebied. Je ziet nu, ook in de games industrie, dat er eigenlijk amateurs op hun zolderkamertje beginnen en professionele dingen maken en daarmee doorbreken. En dat soms professionele studio's moeite hebben om nog nieuw en innovatief te zijn, om iets te maken wat aan de verwachtingen voldoet. Want dan zijn ze een studio, en dan verwacht iedereen ook perfecte code, perfecte graphics en alles moet helemaal lopen en ze moeten vooral voldoen aan het marketingdeel. Dat is zelf heel hard roepen. Maar als het dan een beetje tegenvalt, dan valt het ook met een hard tegen.
En dan zie je dus dat er mensen als kinderen beginnen en zichzelf iets aanleren en in één keer iets heel leuks hebben ontwikkeld. ...[28:10-28:12 slecht te verstaan]... Zoals een Minecraft of zo'n soort game die gewoon een beetje onder de radar door loopt, die zet iemand dan op Steam of op een portal, of wat dan ook, en dan heeft hij ineens een studio. Want het is goed, dan willen mensen meer. Dus het is toch een soort schuivend gebied van 'wanneer is iemand nou professioneel?'
Iemand kan ook heel goed zijn in één onderdeel en dat helemaal beheersen, maar niets weten over iets anders. Zo is er ook onderscheid in onderdelen, iemand kan heel goed game logica maken, of interfaces bouwen, of heel goed met physics werken, of heel goed karakters besturen, maar allemaal tegelijk is weer lastig. Dan moet je je helemaal specialiseren op alleen maar één taal. Een basis. Daar kan je je op specialiseren, die taal helemaal doorleren, wil je daar je vak op maken of wil je het mechanics-deel, het physics-deel programmeren? Of juist 3D leren programmeren. En dat dan op verschillende manieren uit kunnen voeren. In principe ben je nooit uit geleerd. Iedereen is ook een beetje *n00b* [slang voor beginner, sukkel] op zijn eigen gebied. Dus je moet ook weer dingen erbij pakken of het uitbesteden naar iemand anders die het dan beter snapt.

33. [29:29] Nog even gerelateerd aan het leren met games: wat vind je van een platform als Code Academy, dus op een gamified manier mensen code proberen aan te leren?

Het kan een balans zijn. Het is redelijk zelf lerend. Als zij de lespakketten aanbieden en die kan je thuis volgen, en je gaat dan net als met zo'n game een stramien door, en je leventje loopt steeds verder vol... Het is inderdaad wel iets wat mensen trekt, om dat te sturen. Dat is ook waar heel Xbox Live op draait, badges verdienen. Door dingen in spellen te halen of contacten te leggen. Dus het is een stimulans, maar het zal niet de enige stimulans zijn. Mensen die echt willen halen ook heel veel voldoening uit 'het maken van'. Het leuke van een spelletje maken is dat uiteindelijk je best doet, het spelletje werkt en je het spelletje kunt spelen. Soms ben je het dan zelf helemaal zat, maar heb je nog vrienden die er plezier aan beleven. Dat is eigenlijk in zichzelf al de beloning. Als je iets bouwt en je uiteindelijk een keer op een startknop drukt, dan doet het wat je in gedachten had. Dat is het leuke aan programmeren. Het vervelende is de route om er te komen. Dagenlang tegen de muur op lopen omdat iets niet werkt zoals je het verwacht had. Of dat het veel moeilijker blijkt om te maken dan je gedacht had.

34. [31:25] Doet dat fysiek ook dingen met je?

Schelden tegen de computer, natuurlijk. Dat hoort erbij. Dat verwachten ze ook, programmeurs. Iemand die niet af en toe zijn computer uitscheldt, die is niet aan het werk.

Ik denk ook dat live contact wel belangrijk is, en dat het kan helpen. Je kan dingen zelf aan leren, in een schoolomgeving of in een cursus. Dan kan het veel sneller gaan, dan heb je direct interactie. op hele korte termijn. Je hebt dan als het goed is een expert die je kan vertellen hoe alles moet, dus de docent, en dan heb je heel veel peers die ook dingen aan elkaar doorgeven. Want je leert ook van elkaar, wat een ander aan het doen is, wat hij fout doet, of welke trucjes hij kent uit een ander verleden of andere bron die hij heeft.

35. [32:16] Merk je dan ook dat als er beginners zijn, mensen die niet zo heel erg goed zijn, dat die worden buitengesloten of is het juist heel inclusief?

Ligt aan de leeromgeving. Ligt aan de sociale omgeving. [Heb je veel van die situaties meegemaakt?] Het ligt een beetje aan de klassensituatie. Ga je naar een kamp voor allemaal *die hard* programmeurs om daar met een bootcamp mee te doen, [32:48-32:49 slecht te verstaan], dan kun je veel mee kijken, maar je kan weinig doen. [Een global game jam bijvoorbeeld?] Bijvoorbeeld. En de game jam is dan nog redelijk low level. Daar kun je ook heen van: 'ik kan niet programmeren, maar ik kan wel poppetjes tekenen.' Het is vaak zoeken, hoor. Ik heb ook een paar game jams meegedaan. Als een soort hybride game designer programmeur werd ik daar dan neergezet. Specialisatie, in zo'n hoge druk pan, is dan heel moeilijk om neer te zetten. 'Ik ga dit doen, want dat is precies mijn dingetje'. Ik ga daar dan heen als deels game design, deels programmeren, deels om te testen, en soms is het dan lastig om een plekje te veroveren. Omdat sommigen al een specialisme hadden.

Addendum: Email by Sergio van Pul, June 19th, 21:50

"Hoi Esther,

Tijdens het interview ben ik vergeten mijn echte eerste stap in de wereld van game design en programmeren te noemen. Daar dacht ik pas aan toen ik thuis was. Ruim een jaar voordat ik met de opleiding Game Design aan de *HKU* begon ontdekte ik *ZeldaQuest*: http://www.zeldaclassic.com/

Dit project is ooit begonnen als een PC emulatie van het originele 8-bit Zelda spel voor de NES. Tijdens het maken van het spel ontwierp de maker 'per ongeluk' een complete quest editor. Het leek hem wel een goed idee om behalve het spel ook de editor met de wereld te delen.

Al snel ontstonden er tientallen verschillende nieuwe Zelda varianten. Sommige ontwikkelaars bleven trouw aan het origineel en veranderden slechts het verhaal en de layout van de wereld. Maar ambitieuze ontwerpers bogen de editor naar hun wil en bouwden varianten van Final Fantasy, Super Mario of Megaman. En er zijn nog wel meer rariteiten te vinden.

De ZeldaQuest editor leerde mij op een praktische manier over sprites, animatie, scripted events, level design en game balance. Ik heb gewerkt met de inmiddels antieke 1.84 en 1.90 versie. Inmiddels is de editor verder ontwikkeld en kunnen er Quests van SNES niveau in gebouwd worden. Inmiddels heb ik de community (misschien ten onrechte) verlaten. Maar er zijn vast nog steeds spectaculaire staaltjes van 'bedroom coding' te vinden.

Vriendelijke groeten,

Sergio van Pul"

Transcript interview Dennis Rosenbaum, Winvisie

Interview Date: July 1st, 2013, 20:31 - 21:12
Location: Author's home, Utrecht

1. [00:32] Zou je eerst willen uitleggen wat je werk precies inhoudt?

Ja, als developer, software ontwikkelaar, welke vorm je het ook ziet, maak ik programma's. Ik maak websites, web-applicaties. Software die draait op apparaten. Eigenlijk is dat het, ik creëer, ik bouw.

[Wat is je precieze functie?]

Ik ben momenteel *Sharepoint* developer. *Sharepoint* is een product wat Microsoft heeft gemaakt waar mee je heel makkelijk andere websites kan neerzetten, die als portaal kan gebruiken. Vandaag ben ik voor mezelf bezig als, gewoon in de breedste zin van het woord, websites ontwikkelen. Webapplicaties maken, maar vooral gericht op het Web.

2. [01:20] Hoe ben je voor het eerst met programmeren in aanraking gekomen?

[Dennis lacht] Dat was met de *TI-83 Plus*. Dat was een rekenmachine waar je 'if...else' statements op kon neerzetten. Het eerste idee wat ik volgens mij heb gedaan was een app maken, nou geen app, maar gewoon een spiekbriefje. En wat ik daarmee wilde, met een spiekbriefje moest je met pijltjestoetsen naar beneden drukken om te kijken wat de volgende zin was, en ik wilde dat onder verdelen in categorieën. En dat was was volgens mij het eerste ding wat ik echt heb gemaakt met programmeren.

3. [02:03] Wat was de eerste programmeertaal die je echt geleerd hebt?

Goeie. Taal... Ja, als je HTML onder de 'talen' mag samenvatten, dan is dat HTML. Maar dat is geen programmeertaal. Anders zou dat zijn, ActionScript vijf, dat is van Flash, ken je dat? [Ja.] Daarna ben ik over gegaan naar Java, vast nog wel dingen er tussen, Assembly en uiteindelijk op C#. [Ook echt Assembly gewoon?] Ja, ja *die hard*! *Die hard* op de chip programmeren, ja.

4. [02:42] Wat was je motivatie om die talen te gaan leren?

Lol. Ik had er lol in. Ik wilde een website bouwen en daar ga je dan in door. Dan ga je kijken wat er allemaal mogelijk is en dan ga je naar een technische school. En dan leer je programmeren. En dan zie je wat er allemaal mee kan. Wat ik tof vond is dat je dingen kon maken die anderen niet kunnen maken. Dus je bouwt iets en je zet een webpagina neer en dat verander je van kleurtje en dat is aan het begin nog leuk. Vervolgens kan je er steeds meer mee, je kan het interactief maken, dat soort dingen.

5. [03:19] Je zegt dat je naar een technische school gegaan bent, hoe heb je daar leren programmeren? Was dat een sociaal proces? Had je een leraar die je vertelde wat je moest doen?

Theoretisch -- praktisch. Een uur theoretisch, theorie. Wat is het nou? Wat doet het? Wat gaan we straks behandelen bij de praktijk? HBO. Vervolgens ging je de praktijk in en ging je dat proberen en moest je bijvoorbeeld... De eerste les die we gedaan hebben was een backgammon spel. Die moesten we gaan programmeren. Het eerste kwartaal moesten we dat maken. Dan krijg je: 'Wat moet daarvoor gebeuren?' Eerst maken we e en pack, dus één pion. Wat doet die? Wat kan die? Dan ga je abstract denken, object-georiënteerd. Een pionnetje die kan je vooruit zetten, of nee, die kan je neerzetten. Dan maak je het bord. Elke les behandelden we dan één zo'n ding. Vervolgens ging je dat samenvoegen en ging je dat maken. Op zo'n manier leerde je daar programmeren.

6. [04:25] In hoeverre was dat een sociaal proces? Deed je dat samen of vertelde iemand je wat je moest doen en ging je dan zelf aan de slag?

In principe doe je het zelf, maar het maakt het sociaal dat je met mensen praat en het er over hebt en zo van: 'Hey, hoe doe jij het? Waarom doe je het zo? Weet jij hoe het in elkaar zit?" Je moet toch, zeker tijdens de lessen, tijdens het leren ervan moet je andere mensen hebben waaraan je vragen kan stellen. Dat hoeft niet, het is niet nodig. Je kan het als veertienjarig zoldergastje ook gewoon zelf leren. Dan heb je forums. In principe is een forum precies hetzelfde als de sociale processen die je hebt maar dan zonder dat je praat.

7. [05:15] Is het leren van programmeren gelijk aan het leren van een 'gewone' taal? Dus zeg: je wil Italiaans leren versus je wil Java leren?

Wow. Zo heb ik er nog nooit over nagedacht. *Let's see*... Ja, er zitten heel veel overeenkomsten in. En daarnaast ook helemaal niet. De overeenkomst is: als je een taal gaat leren, normaal leer je dat woord bij woord. Dus wat is 'ja' in het Italiaans? Wat is 'nee'?
Dat is grappig en dat leer je. Dat kan bij programmeren ook. Wat doet een if statement, wat doet een 'else statement'. Wat doet een 'switch case'? Wat er dan ook allemaal is. Uiteindelijk merk je dat het verbanden met elkaar heeft. If...else, dat is gelijk aan switch case, als je het op een bepaalde manier structureert. Dat is hetzelfde als met een taal. Een taal taal. Hoe noem je een taal taal? Een Italiaanse taal, een Spaanse taal. Ja, leuk. Dus er zijn verbanden. Er is grammatica, dat heb je dus ook met een programmeertaal. Maar ik denk dat het daarna wel ophoudt en dat een programmeertaal verder gaat.
Een taal kan op verschillende manieren geïnterpreteerd worden, een programmeertaal kan dat ook, maar een programmeertaal wordt altijd gecompileerd. Zo zeggen we dat dan. Het wordt omgevormd naar computercode. Wat een computer kan begrijpen. En dat omvormproces, of dat nou in C#, C++ of Java doet, daar komt altijd ongeveer hetzelfde uit. Dus die geïntegreerde taal, die gecompilede taal, dat kun je misschien wel weer vergelijken met een echte gesproken taal.
Een stukje kan je vergelijken. Je kan er delen uithalen, maar grotendeels toch ook niet.

8. [07:20] Denk je dat het leren programmeren leeftijd gerelateerd is of kan iedereen het altijd leren en pikken ze het dan net zo makkelijk op?

Nee. Nee, als jij het op je vierde gaat leren zal je een betere programmeur zijn dan wanneer je het op je twintigste gaat leren. Het enige verschil is dat nut. Want daarom zie je ook veertienjarige gastjes op hun zolderkamer het leren, omdat ze er lol in hebben. Waarom heb ik het geleerd? Omdat ik er lol in had.
Een programmeertaal leren is niet leuk. Een spreektaal leren is nodig. Maar een programmeertaal, dat doe je voor een functioneel nut. Dat is de reden waarom je het niet op je vierde gaat leren. Betekent niet dat je het niet kan oppakken op je veertiende. Een taal kan je ook oppakken op je veertiende.

9. [08:11] Je zegt: een spreektaal is nodig. Zou je daarmee ook zeggen dat programmeertaal niet cruciaal is?

Uiteindelijk is het een tweede behoefte. Bij programmeren gaat het er om dat je een computer verteld hoe hij zijn werk moet doen. Natuurlijk is dat belangrijk, maar we maken hier heel veel woorden aan vuil, die nodig zijn om het elkaar uit te leggen. Eigenlijk wil je bij een computer dat andere mensen er al heel veel woorden aan vuil hebben gemaakt, heel veel dingen hebben geprogrammeerd en dat *iedereen* het kan gebruiken. Kijk naar een Apple die met hun geweldige idee kwam om te zeggen: 'Hey weet je wat? We maken een device voor de gebruikers en *wij* bepalen wat de functionaliteiten zijn. De gebruiker moet gewoon op één knop kunnen drukken en dan willen ze bellen. De gebruiker wil dit, en de gebruiker hoeft niet meer na te denken over "hoe moet ik die computer nu aanspreken," die hoeft alleen maar te denken: "Oh, ik moet stom op zo'n knop drukken."' En andere rare mensen hebben dan al gezegd hoe die computer dat dan maar moet interpreteren. Ik denk niet dat dat dezelfde basis is, dat dat hetzelfde kaliber is.

10. [09:34] Maar computers zijn steeds wijder verspreid. Dan heb je dus maar een kleine hoeveelheid mensen die die apparaten kunnen beïnvloeden ten opzichte van alleen maar gebruiken. Is dat niet problematisch?

Nee. Ik denk het niet. Laat ik het zo zeggen: ik denk niet dat we nu in een wereld zitten waar we de functie kunnen zien dat als iedereen zou programmeren wat dat teweeg brengt. Want: als iedereen zou programmeren... Programmeren kun je zien als iets heiligs. Je maakt iets. Je zegt de computer wat hij doet en hij doet precies wat je wil.
Als iedereen dat zou kunnen dan krijg je allemaal *little pieces*. Waarom Microsoft iets heel geweldigs neer kan zetten en één persoontje iets minder geweldigs, is omdat Microsoft 15.000 man heeft die aan één ding zitten te puzzelen, om het werkend te krijgen, om het mooi te maken. Die hebben al héél veel geleerd. Ik denk dat als iedereen zou programeren en de computer zou aansturen op hun eigen maniertje, ja dan heb je wel heel in detail.... Dan kan iedereen... *Mass customization*, ken je die term? [Ja.] Nou, dan is dat het een beetje. Dat is dan niet helemaal zo, maar het heeft verband ermee. Maar ik vind het wel een mooie stelling.

11. [11:10] Ik zou graag wat meer willen weten over je werkproces. Wat is jouw proces als je iets gaat programmeren, wat ga je doen?

Ik ga nadenken of hetgeen wat gevraagd wordt ook wel echt is wat diegene wil. Dat is dat stukje persoonlijke, weer. Iemand vraagt om een fiets, maar die wil eigenlijk geen fiets, die wil een gemotoriseerd voertuig. Dus daar kijk je eerst naar. Dat heeft nog niet eens met programmeren te maken.
Vervolgens heb je dat in kaart gebracht en ik ga werkelijk aan de opdracht beginnen. Dan snijdt ik het in stukjes. Zie het als een huis. Dat is altijd een hele mooie metafoor die we ervoor gebruiken.
Eigenlijk is het bouwen van software het bouwen van een huis. Eerst bouw je de fundamenten. We weten wat we ongeveer willen en we zetten een basis neer wat bestaat uit data en de data tonen. Daartussen zit nog een laag die data overbrengt naar de getoonde data. Dat is een model, daar zijn *patterns* voor, daar zijn heel verschillende patronen in code. Dit is het *three-tier model*. Dat is mijn basis altijd voor het bouwen van software. Je hebt data, waar ik uit denk.
Bijvoorbeeld, men wil een TV show *tracken*. Dan ga ik kijken: welke TV shows zijn er? Nou dan heb je dus een stukje data. Welke tijden zijn er? Zijn er tijd-slots? Tijd-slots: dat moet ingevoerd worden, daar moet wat mee gedaan worden. Vervolgens wat er mee gedaan moet worden bouw ik om in een stukje code. In de logica. Ik probeer dat altijd heel gestructureerd te doen, waarbij je kan zeggen van: ik zou één module los kunnen trekken, het fundament staat al, maar dat is dat three-tier, zeg maar. En vervolgens kan ik daar modules

in kloppen en er uit halen zodat je heel makkelijk het kan hergebruiken. Want uiteindelijk gaat programmeren niet alleen om het maken van een stukje code, maar om het later hergebruiken van dat stukje code zodat je heel makkelijk iets veel groters kan bouwen. Is het dan een beetje duidelijk hoe ik te werk ga?

12. [13:39] Een beetje, maar dit is heel erg conceptueel. Ik heb het ook over het daadwerkelijke *zijn*: je locatie, je voorbereiding, hoe je achter je computer zit?

Ok, voorbereiding en dat soort dingen vaak niets. Ik heb die opdracht en stel ik ga er aan beginnen, dan pak ik mijn ontwikkel-omgeving erbij. Ik heb het al in stukjes gedeeld, en dan ga ik gewoon één ding uitprogrammeren. Hoe denk ik dat het werkt? Dan ga ik gewoon wat typen. Het maakt niet uit waar ik ben. Maakt me niets uit, zolang ik dat computerscherm maar heb en die toetsen voor me. Ik ga gewoon beginnen. Ik maak een schermpje, ik maak een knop op dat schermpje, ik druk er op, zodat ik een actie krijg. Vervolgens krijg ik niet hetgeen wat ik verwacht: ik druk op een knop en ik verwacht een venstertje wat zegt: 'Oh, je hebt op een knopje gedrukt', en hij geeft dat niet. Oh, dan heb ik een fout gemaakt, want dat is direct al een bug. Ja, dan los ik dat op en ga ik door met het volgende stukje.
Dan gaat het vaak wel van het visuele uit. Daarmee begon het. Tegenwoordig kan ik ook acht uur lang code kloppen en op het einde pas kijken of het werkt en of het wat doet. Maar dat ligt aan het niveau. Maar qua locatie, omgeving, dat soort dingen? Dat maakt helemaal niets uit. Voor mijn proces in mijn hoofd, om dat werkend te krijgen.

13. [15:07] Je werkt dus voornamelijk op kantoor of thuis?

Voor werk, werk ik op kantoor of bij de klant. Omdat dat toch een heel sociaal iets is, klanten weten niet wat ze willen. Dat is de helft van je werk, zo ongeveer. [Dat is interessant] Dat is een hele interessante. Klanten weten nooit wat ze willen. Klanten denken te weten wat ze willen, maar als ze dan krijgen wat ze wilden, willen ze het weer anders, want dan zijn ze tot inzichten gekomen. Om daar heel even op door te gaan: vroeger werkten we met waterval, met vroeger bedoel ik tien jaar geleden. Toen was dat helemaal hip. Je vraagt aan de klant wat ze willen, je maakt een functioneel ontwerp: 'wat wil je?' Vervolgens maak je een technisch ontwerp, dat doe je intern. "Hoe gaan we dat in de techniek integreren? Gaat dat allemaal werken?" Vervolgens gingen wat Indiërs of een paar mensen in Nederland gingen dat code-kloppen. Letterlijk.
Dan werd het opgeleverd, en dat was dan een project elk van een half jaar. Dus dat je na een twee-jarig traject, hadden ze een product. En dat product was precies wat ze wilden. Wat ze twee jaar geleden wilden. En dat is het probleem. Software was nooit goed.
Er was een miljoen euro in gestopt en dan moest er tien miljoen euro aan door gestopt worden, aan beheerkosten, om het echt tot een product te krijgen wat ze wilden. En dat was dan nog steeds niet wat ze wilden, want eigenlijk was die hele basis niet goed en daardoor was je uiteindelijk véél meer tijd kwijt dan nodig en verwacht.
Wat je nu heel veel ziet, *Scrum*, *Agile*, *Prince2*; project methodieken waarin je elke twee tot vier weken oplevert. Dus je begint en je hoort zeggen: 'Ik wil deze grote bak aan data, of deze grote applicatie willen we'. En dan zeg je als projectteam: 'Nou ja, dat krijg je niet direct. Je krijgt dit stukje. Dit stukje is de basis.' Bijvoorbeeld: een boekhoudprogramma. We gaan een heel nieuw boekhoudprogramma maken waar wat ze zeggen is: 'We willen rapporten maken van die gegevens! We willen dit soort rapporten! We willen grafiekjes! We willen allemaal inzichten over hoe die gegevens nou werken. En we willen ze in kunnen vullen.' Nou ja, dat is ook belangrijk. Dan zeggen wij: 'Ok. Om direct al dat nieuwe systeem te gaan gebruiken gaan wij alleen het invoergedeelte maken. En dat kunnen we binnen twee weken'. Of drie weken, maakt niets uit wat je opleverperiode is. Dus in drie weken ga je dat maken, dat lever je op. Binnen die drie weken heeft iemand anders al besproken met de klant wat ze na die drie weken willen, want drie weken is best snel. Vervolgens lever je na drie weken iets op wat werkt. Dat kunnen klanten mee testen, daar kunnen ze al mee spelen, daar kan echt data in de database komen. Vervolgens ga je in de volgende drie weken, iteraties of sprints noemen we dat, gaan we aan de nieuwe functionaliteit werken.
Dus in sprint één ben je aan het bouwen. In sprint twee is de gebruiker, de klant, aan het testen en ben jij de nieuwe functionaliteit aan het bouwen. In sprint drie zegt de klant: 'Dit is wat ik van sprint één niet mooi vond en dit wil ik net iets anders.' Omdat het zo klein is kunnen ze het ook allemaal goed testen. Begrijp je het, krijg je heel veel duidelijkheid over wat het product nou is en doet. Dus ze krijgen geen grote tractor die ze niet begrijpen, ze krijgen een leuk klein fietsje die werkt en waarop je modules kan installeren. Dus die tijd met die klant, waarin je bezig bent om dingen duidelijk te krijgen, dat is heel belangrijk.
Maar de vraag was...? [We hadden het over locatie.] Oh ja, precies. Nou ja, voor mijn werkgever nu, voor op mijn werk werk ik voornamelijk op kantoor en soms bij de klant. Gewoon op een laptop en een plek die ze ergens hebben. Maakt niets uit waar. Voor mezelf werk ik natuurlijk ook. En dat is gewoon op de bank. Gewoon op de bank, voor de TV, tijdens het TV kijken. Wat dan ook. Als ik echt mijn handen niet wil beschadigen doe ik dat achter een bureau, maar daar denk ik over twee jaar pas aan als ik RSI heb. Dat is een beetje het idee.
Of in de bus, of in de trein, of waar dan ook. Als ik maar tijd of mogelijkheid heb waar ik kan werken, waar ik mijn laptop, mijn tablet, of wat dan ook mee heb.

14. [19:48] Is het belangrijk om code uit je hoofd te leren?

Heel. Is een hele moeilijke vraag die je stelt, eigenlijk. Want wat is code? Uiteindelijk kan je heel veel, bijna alles doen met if...else statements. In Assembly heb je 32 operaties. Met 32 operaties kan je alles, want alles wordt naar Assembly omgezet. Dus dat betekent dat elk stukje code wat ik schrijf, omgezet wordt naar die 32 operaties. Dus eigenlijk hoef ik er maar 32 te weten. Maar om het te versimpelen hebben ze makkelijkere dingen gemaakt. Die operaties bevatten bijvoorbeeld: neem een byte, tel daar eentje bij op en schrijf die byte terug. Dan hebben we overigens al drie operaties beschreven van die 32.
Wat wil je? Je wil vermenigvuldigen. Wat is vijf keer drie? Wat is vijf keer acht? Of erger nog, wat is 20.322.221 keer vijftien? Maar dat wil je vermenigvuldigen. Ok, daar hebben we *libraries* voor. Libraries noemen we die dingen. Dat wil je wel weten, dat is handig, want dat ga je vaak gebruiken. Vervolgens is het handig om te weten hoe je een schermpje maakt. Een formulier, een website, een webpagina. Dus dan moet je dat ook weten. Vervolgens is het handig om te weten hoe ik daar makkelijk een knop op krijg. Hoe ik heel makkelijk een *event* er aan hang: oftewel, je klikt er op en dat doet dan wat.
Eigenlijk kan je het zo zeggen: hoe meer code je kent, hoe makkelijker je leven wordt, omdat je dan niet hoeft op te zoeken wat de mogelijkheden zijn. Er zijn heel vaak meerdere manieren om naar Rome te gaan, ook in programmeren. Zoals ik net al zei: een if...else statement tegenover een switch case statement doet precies hetzelfde. Het ziet er anders uit maar de één gebruik je of de ander gebruik je, soms heeft het een voordeel of nadeel. Dus hoe meer je weet, hoe beter je dingen kan inzetten en hoe minder vaak je zelf het wiel hoeft uit te vinden. Dus om een eenduidig antwoord te geven: ja, het is wel belangrijk om code te kennen.

[Dus het is meer dan alleen de syntax onder de knie hebben?]

Met de syntax onder de knie hebben, kom je een eind en kan je overal komen, maar het maakt je leven makkelijker als je meer weet.

16. [22:50] Signeer je je werk, en waarom of waarom niet?

Ik doe dat sinds ik bij dit bedrijf werk. Nou ja, ik doe het sinds een maand. We hebben een style kop, dat is een framework, of ja, ik weet niet hoe je dat moet noemen. En daardoor moet je je code signeren. De auteur, de copyright. De auteurs vul ik niet eens in, want het boeit mij niet. Maar de copyright staat er wel. Dus eigenlijk signeer ik mijn werk nooit. Maar, er staat commentaar boven dat het van *Winvision* is, het bedrijf waar ik werk.

[En waarom doe je dat dan niet?]

Omdat ik weer wil terugkomen op het begin: het is iets functioneels. Ik maak het om de computer iets te laten doen, het wordt weer omgevormd naar iets anders. Iedereen heeft zijn eigen programmeerstijl, je kan aan programmatuur zien wie het heeft geschreven. Share, in principe. Aan de andere kant zijn er ook stromingen die zeggen: 'Wij willen heel erg dat je op een bepaalde manier programmeert. Dat is iets heel goeds, want als je op een bepaalde manier programmeert kunnen al je collega's het begrijpen en kan je het heel makkelijk overdragen.' Weten hoe je moet programmeren maakt het leuk, want je bent sneller overal. Maar, ik vind het- en ik blijf het iets functioneels vinden. Dat je een eigen signatuur hebt in je programmeren, in je code, ja, *care*. Als het maar werkt.

17. [24:28] Als we het dan toch over functionaliteit hebben, waaraan kan je zien of iets goede of slechte code is?

Aan functionaliteit niet. Aan functionaliteit zie je het pas wanneer het werkt, danwel niet. Maar de code intern kan heel mooi zijn, of niet. Doet iets een operatie in 25 milliseconden, dan kun je zeggen: 'Nou, dat is best snel', dat is één veertigste van een seconde, dat zie je niet eens. Dat ziet je oog zelfs niet als je knippert. Maar die code zou het in twee milliseconde kunnen doen. Omdat je gewoon iets heel stoms hebt gedaan in de code. Dan is je code niet mooi. Functioneel zou het moeten werken, en werkt het ook, totdat er blijkbaar twintig mensen op zitten. Dan duurt het opeens in plaats van één vijfentwintigste van een seconde veel meer. Dan gaat het fout. Dus uiteindelijk zie je het pas bij grote getale.

[Dus het heeft te maken met de snelheid van de functionaliteit]

Dat is een voorbeeld.

[Of heeft het te maken met de helderheid van de code?]

Ik zou code moeten laten zien wil je het begrijpen. Je hebt bijvoorbeeld spaghetticode, niet herbruikbare code. Dus als je mooie code wil schrijven, gaat het er bijvoorbeeld functioneel om dat het snel gaat of dat het werkt zoals je verwacht. Maar in code gaat het er denk ik, vind ik ook zelf, meer om of het herbruikbaar is. Object-georiënteerd komt daar altijd in terug. Of het los te koppelen is en terug te koppelen naar de echte wereld.
Het is een beetje hetzelfde, om er weer een metafoor voor te geven, als het bouwen van een mens. Stel ik zou een mens gaan programmeren. Dan zou ik alle functionaliteiten in één code class kunnen doen. Nou ja, dat werkt. Want de mens doet precies wat de mens moet doen. Het kan lopen, het kan schieten, het kan wat dan ook. Maar ik zou het veel beter kunnen vervormen naar: een mens heeft twee armen, een romp en twee benen. Wat kunnen benen? Benen kunnen bewegen. Hoe kunnen ze bewegen? Ze hebben spieren. Ah, ze hebben spieren. Hoe sterk zijn de benen? Nou ja, dat ligt aan de botten. Want hoe sterker de botten, hoe minder snel het breekt, dus hoe meer het aan kan. Nou ja, dat ligt ook aan de aanhechting. Als je nou al die dingen apart definieert wordt het veel makkelijker om dat later weg te halen en in een veel beter mens te stoppen.
Stel je zou alles in één stuk, in één class in één code file neer hebben gezet, dan zou je dat niet kunnen extraheren. Ik denk dat mooie code er toch om gaat dat het herbruikbaar is, dat het leesbaar is, want je leest code ongeveer tien keer en je schrijft het maar één keer.
Bijvoorbeeld al een stukje naamgeving. Stel je gaat de tijd bijhouden, hoe lang iets duurt. Dan kan je zeggen: 'Time', wat bijvoorbeeld een object is. 'Time "T" = new time', en dan gebruik je 'T' als zijnde de variabele die je steeds terug laat komen. T punt hoeveel tijd is er eigenlijk vergaan, of wat dan ook.
Als je zou neerzetten "Time," + "aantal seconden", of beter: "Time," + "aantal seconden die dezeoperatie gaat duren," (wat dan eigenlijk weer te lang is, want dat lees je niet meer, want dat is te lang), daar moet je verband in zoeken, eigenlijk een concessie zelfs in doen. Dat heeft ook met mooie code te maken.
Er zijn heel veel stukjes. Dus: kan ik het uit elkaar trekken? Kan ik er goed doorheen lezen? Als een functie bijvoorbeeld uit meer dan twintig regels bestaat, dan kan ik het al niet meer lezen. Dus is het dan nog goede code? Het kan. Het ligt er aan of de code duidelijk is, die één functie heeft. Als het meerdere functies heeft in één code die, bijvoorbeeld, later weer gekopieerd wordt en ergens anders gebruikt, ja, dan is het in principe slecht.
Om een heel goed voorbeeld te noemen van wat slecht is: bij de ANWB hadden ze iemand die daar werkte en die website bestond uit 30.000 HTML files. Wilden ze een nieuwe pagina, dan kopieerden ze de oude pagina en die zetten ze nieuw neer, en dan wijzigden ze de content. Dat betekent dat als er een menu-wijziging was, dat er 30.000 pagina's aange-

past moesten worden. Een vorm van slechte code. Maak eerst alleen je *template* en verander alleen die content, want je template blijft wel bestaan. Dat is één van de slechtste voorbeelden die ik ken.

18. [29:20] Zie je ook duidelijk verschil tussen professionele en amateuristische programmeurs?

Ja, meestal wel. We spreken over junior, medior en senior. Als aangenomen wordt, word je als junior aangenomen: geen ervaring. Medior: drie jaar ervaring, of senior: vijf jaar ervaring. Nu is het bij programmeren een beetje lastig, je kan C# en je kan HTML. HTML kan je senior in zijn terwijl je in C# of .NET, je een junior bent. Maar over het algemeen zijn het die termen, dus junior, medior en senior. Maar als een junior code schrijft zie je dat. Dan kan je dat wel zien. Die doen domme dingen, die let niet op zijn inspringen, bijvoorbeeld. Als je een functie schrijft en je hebt vervolgens een if statement, dan moet je inspringen en dat gebeurd niet, of dat gebeurt op een oude manier, een andere manier, de Amerikaanse manier. De Europese manier. Daar heb je ook weer verschillende vormen van. Gebruikt slechte variabele namen, al dat soort dingen. Gebruikt zijn eigen code en niet alle andere dingen die bestaan. Bijvoorbeeld *frameworks*, of *DLL's* die andere mensen gebruikt hebben. Dus die heeft eigenlijk een heel veel bagage nog niet die andere mensen wel hebben die ze gewoon zo kunnen inzetten.
Je ziet het vooral aan de snelheid. De code kan je vaak zien, maar het is ook wel mogelijk dat een junior wel mooie code schrijft.

19. [31:06] Wat versta jij onder een ambacht en denk je dat programmeren een ambacht is?

Ja. Die definitie is bijna niet in te zetten in programmeren. Ik ben wel van mening dat programmeren een ambacht is, maar ik zou je niet kunnen zeggen waarom. Want programmeren *an sich* is uiteindelijk een stukje computer aanspreken. Is dat nou ambacht? 32 operaties, komen we daar weer op terug. Nee, dat niet. Maar het feit dat er zoveel verschillende programmeertalen zijn en dat mensen er constant mee werken, en er dingen mee doen, dat ze er mee bezig zijn, dat het mooi kan zijn, minder mooi, ik durf bijna zelfs te zeggen: aantrekkelijk, dan kan het wel. Dus in dat opzicht zou het wel een ambacht zijn en hoort het bij de ambachten.
Een ambacht is bijvoorbeeld het maken van een mooi sieraad. Het kúnnen maken van een mooi sieraad. Dat kan met programmeren ook. Ik kan een website maken, en ik kan een *mooie* website maken. Dan hoeft hij er niet mooi uit te zien, zolang de achterkant maar werkt. Op een hele mooie en elegante manier bijvoorbeeld. Dus ja, ik vind het wel een ambacht.

[Maar wat versta jij precies onder de term 'ambacht'? Je had het over een sieraad?]

Ja, dat beschouw ik als een ambacht, inderdaad. Ik heb het gevoel dat dat de algemene terminologie daarvan is. Het creëren van iets, wat andere mensen niet kunnen. Is dat zo, ja of nee? Want iedereen kan het leren. Timmerman is een ambacht. Vind ik. Want, je maakt huizen, en je maakt huizen mooi. Moet het wel een goede timmerman zijn. Eigenlijk is dat het een beetje. Een schilder is een ambacht. Het maken van sieraden. Eigenlijk kijk je dan al direct terug naar drie beroepen die al sinds de middeleeuwen bestaan en uitgevoerd worden. Dus misschien is dat wel mijn definitie van een ambacht. Dat het al heel lang handwerkmatig uitgevoerd wordt.
Maar als we dan kijken naar een ander beroep, laten we even zeggen: elektronicus, iemand die soldeert, die dingen in elkaar zet. Is dat ook een ambacht? Bij eerste gevoel zeg ik: nee. Maar aan de andere kant, die maakt ook dingen. Die maakt ook mooie dingen. Een filosoof, dat is geen ambacht. Want die maakt niets. Ik denk dat dat de definitie is. Je moet wat creëren en je moet wat maken. Het moet visueel zijn, het moet er uit zien, en het moet functioneel zijn. Het moet wat doen. Een timmerman maakt een huis, met hout. Het ziet er uit? Maakt niets uit. Een architect die tekent een huis, is dan ook weer een ambacht. Een programmeur maakt software, wat uiteindelijk op je tablet, op je computer of op je TV draait, wat dan ook.

20: [34:19] Nog even terugkomend op de code: is dat veranderd sinds jij er mee begonnen bent?

Dat vind ik heel moeilijk. Als je er in zit is het net zoals wanneer je een kind ziet opgroeien. Je ziet niet dat het groeit. Dus op zich kan ik niet zeggen dat in de tijd dat ik er mee bezig ben dat ik het heb zien groeien. Ik kan wel terugkijken naar wat er twintig en dertig jaar geleden was, qua programmeren, technieken. Omdat je dan een kind van één jaar tot tien jaar ziet, en dan is het van 'Oh wow, hij is echt onwijs veel veranderd.'
Als ik er op die manier naar terug kijk: ja, code is veranderd. Er zijn meer talen, er zijn verbeterde talen. Als ik dan even snel terug kijk naar: 'wat was er toen ik begon?' wat natuurlijk een geleidelijk iets is, laten we zeggen dat ik er tien jaar geleden aan begon, dat was in 2003, tot 2013, ja toen was de wereld nog anders. Alles wat ik nu doe, was er nog niet zelfs. C# werd net uitgevonden. Het .NET framework werd net uitgevonden. Betekend niet dat de middelen er al redelijk waren. Java kan hetzelfde als C#, bij wijze van spreken. Maar veranderd zal het zeker zijn. Het is geëvolueerd.
Dertig jaar geleden zeiden ze over programmeren:
'Over dertig jaar, of over veertig jaar, zullen we alleen maar dingen in elkaar gaan klikken.' Dat betekent dat je een scherm voor je had en dat je programma's ging maken en die programma's sleeptje je in elkaar. Je had een venstertje en daar kon je een knop op slepen en je kon er een actie achter slepen en je kon hem bij wijze van spreken vertellen wat hij zou moeten doen. Tien jaar geleden zeiden we hetzelfde. Nu zitten we eigenlijk nog steeds op hetzelfde niveau. We denken over tien jaar dat het makkelijk is om dingen in elkaar te slepen. Uiteindelijk is dat niet zo. Uiteindelijk zullen we altijd mensen moeten zijn die code kennen en snappen. Al is het maar om die basis te maken. Want je ziet die verschuiving van 'iedereen is programmeur' tot 'er zijn maar een aantal mensjes die programmeur zijn en die bouwen frameworks zodat wij andere programmeurs het makkelijk in elkaar kunnen slepen', dus het programmeurvak wordt weg gerangeerd. Maar we zullen altijd willen blijven tweaken, dingen zelf willen maken.

21. [36:56] Wat vind je van de toenemende mate van abstractie in code?

Dat is iets heel goeds. Vooral, er zullen heel veel banen door verloren gaan. Hoe abstracter het wordt, hoe meer banen er weg gaan. En er blijven wat 1337's over die programmeren heel leuk vinden. En dat is iets goed en iets slechts. Het goede wat het met zich mee brengt is dat wij ons kunnne richten op het hogere doel. Waar we nu mee zitten is: aanvallen van buitenaf, ...[37:42-37:43 slecht te verstaan, een vorm van aanval]..., hack attacks. Iedereen heeft een virus op zijn computer. Weten ze niet, maar dat is zo. Je kan een virus maken die gewoon achter de virusscanners blijft. Mensen weten het niet, maar als je daar dieper op induikt is het te doen. Dat betekent dat wij ons in die abstracte lagen kunnen neerzetten. Als niemand meer slechte code schrijft, want die geeks schrijven geen slechte code want die reviewen elkaar. Die nerds, want geeks zijn andere mensen. Die nerds. Heel belangrijk, terminologie. [Dennis lacht]. Dus die schrijven die code en vervolgens gaat daar boven een laag boven lopen die zegt: we willen authenticatie en we willen autorisatie. Die willen we er op zetten. Dat moeten een soort modules worden.
Dus die mate van abstractie is goed, maar je zal altijd die kleine programmeurs houden, nou, klein, genoeg programmeurs, want we zitten nog lang niet op dat niveau, dat alles abstract is. Ik vind het zelf heel goed om te weten welke code je schrijft, wat voor elektrische pulsen er in jouw flipflops gaan. Een flipflop is één transistortje die aan of uit kan staan. Ik vind dat belangrijk, want ik wil weten wat mijn computer doet met mijn code, waardoor ik snellere code kan schrijven, die beter werkt dan code waar mensen niet over nadenken, maar het is niet nodig om te weten. Het is niet nodig om te weten wat het precies doet.
Dus abstractie is iets goeds, zolang programmeurs maar altijd iets kunnen aandraaien. Het is een tegenstrijdigheid. Ik brabbel ook een beetje merk ik, omdat ik het er aan de ene kant mee eens ben dat het goed is, dat abstracte. Aan de andere kant zullen er altijd programmeurs zijn die willen programmeren en zullen programmeren en die moeten programmeren. Misschien omdat we daar pas over tien jaar zijn, dat moment dat het echt abstract wordt. Dat is het enige. Ik denk dat dat het is. Een mooie quote, er is geen quote.
Het programmeursyndroom. Dat is dat programmeurs altijd zelf hun code willen maken. Zelf het wiel willen uitvinden. Het is veel leuker om zelf precies diezelfde tractor te willen bouwen, maar hey, ik wil *gele* wielen. In plaats van dat je alleen de wielen vervangt, maak je die hele tractor zelf. Het is namelijk veel leuker om hem zelf te maken dan dat je hem koopt en alleen de wieltjes er op zet. Dat vat wel aardig samen wat een programmeur is, in basis.

E-mail interview Douglas Rushkoff

E-mail exchange on 23 juni 2013, 9:14 - 15:14

This questionnaire will aid in the research in the subjects programming, games and craft. The questions and answers will be used in my MA thesis, the entirety of the document posted in the appendix. Once my thesis is finished, you will receive a digital copy if so desired.

1. Could you tell me the first programming language you learned and how you learned it?

I learned Basic Extensive, on a terminal in my high school. It was connected to some sort of 1970's IBM mainframe for the school administration. I learned from my peers. They showed me the basics and then I just tried new commands.

2. Is learning how to programming a social experience; if so in what manner?

For me it was, because it was in a room with three terminals, a dozen kids, and limited resources. That's the way most programming languages were developed too: as a way of sharing the limited processing cycles available to us.

At this point, it doesn't have to be social. I mean, you can go online to a site or buy a book and learn the basics. It sure helps to have someone else there who knows more than you, though, for when you get stuck. And building a program is often much better a shared activity.

3. You have argued that learning computer programming is crucial. Have your thoughts on this subject changed or evolved while working at *Code Academy*?

I still think people who understand something about code and, more importantly, the biases of digital technology, will be better prepared to make conscious choices in a digital world.
I do think coding – real coding – is probably too much to ask of most adults, though. If you don't get exposed to algorithmic thinking at a young age, it can be quite difficult to get later on. Plus, we live in a very restrictive economy now. People work maybe three times as many hours as they did when I was a kid, so adults don't have any time for enrichment. It's just the fact. As unemployment goes up, it may give people more time to learn things, though.

4. As I understand it, software like the one used in *Lego Mindstorms* or Scratch teaches children to think in terms of building blocks to get them into the programming mindset. Could you explain why this is or isn't a proper approach?

It's a proper approach. Not the only one, but nothing wrong with it. I think it's good for people who don't yet know algebra. It's all good.

5. In the *iPad* game *Hakitzu*, people learn how to control their fighter-robot by commanding them in JavaScript. The more code you learn, the better your robot is equipped for a fight. How do you feel about initiatives like these? Is this a proper learning technique?

I think there are many proper things! I don't think there's one right way. I think salad is a proper food. I think beans are a proper food. The test of a good programming course for kids

is whether they like it, and whether they learn something. I think *Hakitzu* is great to the extent kids use it. *Mindstorms*, too. I think *Mindstorms* and *Hakitzu* are more limited in their appeal than we might have thought. They are hard for many kids. Weirdly enough, *Minecraft* ended up more successful, even though I don't think it was intended to be used as a programming language.

6. What do you think of when you hear the term 'craft' and do you think that computer programming fits that description?

I don't know. I don't usually think of computers with that word. My friends started two magazines – *Make* and *Craft*. *Make* was more for tech, *arduino*, and that stuff. *Craft* was more for yarn and such.

7. How does working in a simulated environment relate to notions of materiality (often associated with classic craftsmanship)?

I don't really know. I think it's harder for people to recognize the acehivement sometimes. But now that programmers can get paid so much for figuring something out, society is coming to recognize that what they're doing is real.

8. How do programming and ownership of code and/or software relate?

I don't know that they do, except insofar as we have a legal system to define these relationships. A programmer can work for hire, or work for himself.

9. How can you tell whether something is quality code? Is clarity or functionality more important?

I think it has more to do with its resilience. Can it be changed? Can it be used in unexpected ways? Functionality probably matters most. If it's not clear, you can just document it better.

10. How do you feel about the increase in abstraction in computer programming (i.e. the move away from machine language into increasingly abstracted functions fit for human understanding)?

This always happens. It's just a matter of recognizing that. And if we are going up a level, we had better make sure the underlying layer is really good. And that we trust the people who are writing it.

11. Could you share some insights into the gamification processes applied in *Code Academy*, why and how they work?

I don't really know so much about that. You should talk to them. I'm really just an advisor.

12. What are *Code Academy*'s main goals for the next five years?

I don't really know. I guess to become the place the world goes to learn and teach code.

13. Can you tell me about some of Code Academy's successes and failures?

You should probably talk to them. They've got a site with "stories" about what has happened there. Or email [edited out] who runs the community stuff.

14 . Are there any other thoughts, recommendations or questions you would like to add?

I think my book *Program or Be Programmed* shares my answers to all this better than I am doing here, so I encourage you to look at that. And good luck!

D

Addendum: E-mail by *Code Academy* community manager Linda Liukas July 3rd, 2013, 23:46

"Hey Esther,

Thanks for reaching out - I'm unfortunately unsure if any of these questions are something we can answer : (But in general I'd point you over to codecademy.com/press where you can find answers to question number one.

For gamification,

Gamification of education is obviosly a big trend. The way we see it, the best learning experiences come down to motivation and reward systems. The problem with learning things from books and videos is that you're just reading or watching them by yourself, and there's no reward when you've finished. With *Codecademy* the interactive interface gives the user points and badges for completing exercises and keeps them motivated.

And for 5 years,

With Codecademy we're seeing p==rogramming enter every venue of society. You can be a better financial analyst by learning to program. You can be a better journalist by learning to program. People are realizing that and learning it to get the upper edge.

Linda
Community Manager
Codecademy"

Transcript interview Peter Goes, Qlvr

Interview Date: June 19, 2013, 12:02 - 12:46
Location: *Dutch Game Garden*, *Qlvr* offices, Utrecht

1. [0:03] Wat doe je?

Hier heb ik de rol als creative developer. Dat houdt zoveel in als, we zijn een vrij klein team met vaste medewerkers en het management die de concepten ontwikkelen en ook daadwerkelijk uitvoeren. Dan ben ik programmeur. Met zijn allen denken we de concepten uit, bedenken we de flowcharts en bedenken we de structuur van de apps. Een deel van mijn werk is dus de conceptualisatie van een product. Daarna de realisatie doormiddel van software development.

2. [00:46] Ik begrijp dat jullie recentelijk een app hebben gemaakt voor het VWO, kan je daar wat over vertellen?

Dat was een simulatie van de papegaaienpopulatie op Bonaire. Het doel was om studenten te leren hoe ze, wat de invloed van de omgeving heeft op zo'n populatie en hoe de mens daar een grote rol in speelt. Het scenario was zo op Bonaire: een eiland en daar leefde een populatie papegaaien, en op een gegeven moment kwam daar de mens naartoe. Door allerlei wiskundige modellen hadden ze daar berekend wat de populatie was voordat de mens er kwam, tijdens dat de mens er kwam en een tijd nadat de mens daar leefde. Dan zag je de populatie veranderen. Jouw taak als gebruiker van de app was eigenlijk om een plan te schrijven om de invloed van de mens dusdanig terug te brengen zodat de populatie weer op een gezond niveau terug kwam. Dit kon je doen door stropers tegen te gaan en meer bomen te planten zodat ze konden nestelen.
Aan de hand van de simulaties kon je zien wat de effecten daarvan waren op een periode van twintig tot veertig jaar.
De opdrachten die daar bij kwamen was 'maak een plan om de populatie gezond te krijgen' en dat voor een aantal tijdsvlakken.
Wat ik daar specifiek in gedaan heb, was de samenwerking met de Radboud Universiteit, Universiteit van Nijmegen. Zij hadden alle modellen, wiskundige modellen. Allemaal gebaseerd ook op echte wetenschappelijke modellen. Die heb ik omgezet in software. Eén van de designers heeft daar een visuele schil omheen gemaakt. Dus ik heb eigenlijk een insteek geschreven om die modellen dusdanig aan te passen zodat je daar dan grafieken uit krijgt en er data mee kan genereren.
Omdat het voor VWO studenten was moet het ook aantrekkelijk genoeg zijn om er iets mee te kunnen. Grafieken op zich, dat is nogal een saai beeld. Dus we hebben er een soort van 'view' bij gemaakt die dan een scène laat zien en afhankelijk van populatie of tijd komen daar dus meer of minder papegaaien bij. Om het geheel nog een beetje aantrekkelijk te maken.
Datzelfde hebben we gedaan, eenzelfde soort simulatie maar dan voor economie studenten. Dan had je een staatje ter grootte van Nederland. Een eiland ter grootte van Nederland en die moest je voorzien van energie. Je moest ervoor zorgen dat een blackout uitbleef. Hoe doe je dat en hoe zorg je er nou voor dat CO2 waardes, dat daarmee rekening gehouden wordt. Welke energie vormen leveren er meer op, welke minder?

3. [04:06] Dat relateert mooi aan één van mijn vragen. In hoeverre is de content van de game van belang van de onderliggende lessen? Oftewel: heb je abstracte modellen die je aan probeert te leren doormiddel van willekeurige situaties of zijn de games contextueel toegepast op concrete voorbeelden?

Dat hangt heel erg van de opdrachtgever af. Over het algemeen is het vrij concreet. Die twee apps waar ik net over vertelde, die kwamen voort uit een vraag van de Universiteit van Wageningen. 'Dit is onze case: hier willen we iets mee. Dit zijn de modellen die we hebben. En dit is wat we die kinderen willen leren.' Dat nemen we dan en daar gaan we een app voor bouwen.
Zo zijn we laatst ook voor Naturalis bezig voor een applicatie. Is meer een onderzoekstool, maar ook dat is heel specifiek op een case gericht. Daarnaast hebben we in den beginne apps uit gegeven voor kleinere kinderen, die we helemaal zelf hebben uitgeschreven en helemaal zelf hebben ontwikkeld.
Een voorbeeld is dat we het karakter Jeffy hebben bedacht. Jeffy is een jongetje van een jaar of tien en die heeft allemaal vragen. 'Hoe zit de wereld in elkaar? Waar komen de zon en de maan vandaan? Waar komt eb en vloed vandaan?'
En zo gaat Jeffy een hele reis door het leven heen, eigenlijk. Leert hij rekenen, leert hij taal, dat soort dingen. Daar maken we dan allemaal kleine apps van. Die komen meer uit een globale behoefte van we willen kinderen leren rekenen. Nou, dat is het uitgangspunt. Hoe gaan we dat doen? Daar gaan we een concept omheen bouwen, dat gaan we uitwerken en dat gaan we maken.
Dus het kan twee kanten op, maar het fijnste is het werken met een echte case. 'Dit moet er komen,' dan kan je je heel specifiek richten op wat je moet doen. Wij zijn een app-development bedrijf, dus we zijn geen educatieve instelling. Dus de kennis die we over willen dragen komt vaak bij klanten vandaag. Wij bieden een medium om dat dan over te brengen.

4. [06:20] In 'traditionele' klaslokalen werk je vaak met een feedback systeem, 'nu doe je het goed' en 'nu doe je het fout.' Zou je zeggen dat dat in die games ook zo terug komt of is het meer een kwestie van de spelers hun eigen 'ding' laten doen en dan kun je dingen op die manier leren; meer een soort Montessori model?

Het hangt er een beetje van af. Mijn persoonlijke opvatting is dat het kind het meeste leert door heel veel te doen. In eerste instantie moet het leuk zijn om te doen. Als het niet leuk is, gaat een kind het niet uit zichzelf doen. Dus een competitief deel komt altijd om de hoek kijken. Maar tegelijkertijd heb je ook genoeg kinderen die daar juist niet van gecharmeerd zijn.
Gamification van een lesomgeving, daar zit een competitief deel in. Je kan iets niet goed doen, maar dan moet je er ook op gewezen worden *waarom* je iets niet goed doet. Waar wij ons op proberen te richten is: niet alleen zeggen dat iets niet goed

is en ‘dit is het goede antwoord’, maar ga nou eens kijken met deze en deze en deze hulpmiddelen kun je er dan zelf achter komen wat wel het goede antwoord is. Dan vertel je dus niet in eerste instantie ‘iets is niet goed,’ maar ‘iets kan beter.’ Dan probeer je op die manier een kind zijn eigen leerproces te doorlopen. Zonder meteen het antwoord in zijn gezicht te duwen, dan stopt het leren. Als ik denk aan een school, aan een klaslokaal waar een vragenboekje ligt met daarnaast een antwoordenboekje, dan denk ik aan wiskunde, bijvoorbeeld. ‘Dit zijn de opdrachten die je moet doen en die zijn fout, want kijk maar, het antwoord staat hier.’ Maar waarom is iets fout? Natuurlijk zijn er ook methodes die dat ook proberen uit te leggen, maar doormiddel van digitale functionaliteit kun je dat veel beter op het niveau van het kind aanbieden.
Denk aan een filmpje waarin iets wordt uitgelegd. Het filmpje kun je pauzeren, kun je terugspoelen en opnieuw laten spelen. Helemaal op het niveau van het kind. Op een snelheid die het kind fijn vindt. Misschien leren sommige kinderen veel sneller en willen ze dan dingen skippen omdat ze het dan al begrijpen. Maar het volgende kind heeft dat dan wel nodig, om het nog een keer te herhalen. Om het goed te leren. En dan zijn digitale technieken juist heel erg geschikt.

5. [08:55] Waar me dat dan een beetje aan doet denken is een stukje van Sebastian Deterding, een goede toepassing van gamification werkt het beste als je steeds een stapje verder gaat in de moeilijkheidsgraad. Hoe zorg je er voor dat dit voor kinderen ook leuk blijft? Soms heb je mensen die echt niet door een bepaald level heen komen. Hoe werk je met die frustratie?

Het voordeel van een computer is dat het soort van kan nadenken. De rekenapplicatie waar ik net over vertelde, van Jeffy, een adaptive learning system zat daar in. Ik zeg dat nu heel ingewikkeld, maar dat hield in dat een kind een aantal fouten op rij maakte, dan ging het systeem er van uit dat het te moeilijk was en dan bood hij dus simpelere oplossingen aan. En op het moment dat een kind heel snel de juiste antwoorden gaf, dan ging het systeem er van uit dat: ‘oh dit niveau is te makkelijk, dus ik ga moeilijkere opgaven voorschotelen.’
Wat je dan krijgt is dat twee mensen naast elkaar kunnen zitten, allebei op *hun* niveau aan het rekenen zijn, en het allebei leuk blijven vinden. Want er moet wel genoeg positieve feedback zijn, er moeten genoeg succesmomenten zijn, om een game leuk te houden.
Er moeten natuurlijk een aantal zijn om je wat meer uit te dagen, maar ik ben er van overtuigd dat de hoofdmoot moet er op gericht zijn dat het leuk is. Dus je hebt succeservaringen nodig. Op het moment dat je die kan afwisselen met uitdagende negatieve ervaringen, dan kun je een kind blijven stimuleren, maar wel op zijn of haar niveau.
Een tekstboek is inderdaad alleen, daar zit één niveau aan vast, en iedereen heeft zich daar maar aan te houden. Maar een digitaal systeem kan daar op inspelen.

[10:48] Zou je zeggen dat de toegevoegde waarde van games in educatie voornamelijk ligt bij het motiveren of zitten er ook andere elementen in?

Ja, voor een deel wel, want papieren opgaven zijn gewoon saai. Met digitale technieken heb je gewoon een mogelijkheid om er iets meer schwung aan te geven, iets meer vuurwerk aan te geven. Maar, omdat je met kinderen ook samen kan gaan werken op een leuke manier heeft het niet alleen dat wat je doet leuker is, maar omdat je het ook samen met mensen kan gaan doen, heeft dat niet alleen entertainment waarde, maar ook een stimulerende leerwaarde.
Nou, diezelfde rekenapplicatie weer, die deed je in principe met zijn tweeën. Twee setjes kaarten en op die kaarten staan getallen. Met die getallen moest je een som berekenen. Je had echt overleg nodig tussen elkaar om daar te komen. Ik had twee getalletjes en mijn vriendje had twee getalletjes. Ik kon daar niet in mijn eentje komen, maar hij ook niet. Samen moesten we kijken: ‘jij hebt dit, ik heb dit, hoe gaan we samen tot iets komen?’ Nou op het moment dat je daar samen mee bezig bent denk ik dat je daar echt veel meer van leert dan als je het in je eentje doet.
Dat kan op papier natuurlijk ook, maar mijn ervaring daarmee is dat papier nog te individueel gericht is. Een game is... Het begon natuurlijk in je eentje, maar al vrij snel kwamen daar tweede controllers bij. Dus multiplayer heeft altijd bij games er in gezeten. Nou, dat principe kan je nu ook op leren toepassen. Ik denk dat dat heel goed werkt.

7. [12:44] We hebben het tot nu toe voornamelijk over kinderen gehad. Denk je dat games dan ook specifiek geschikt zijn om kinderen dingen te leren of zou je er als volwassene ook nog baat bij hebben?

Dat hangt er maar helemaal van af waar je mee bezig bent. We zijn nu bezig met de ontwikkeling van een applicatie die psychiatrische patiënten ondersteuning biedt in hun therapie. Dus dat is niet zozeer het leren van dingen, maar wel de ondersteuning biedt voor een therapie doormiddel van een game. Nou zijn volwassenen wezenlijk anders dan kinderen, het is een hele andere doelgroep, ze hebben een heel andere feeling met digitale objecten, maar ook die volwassenen zijn uiteindelijk een game aan het spelen. Ze zijn uiteindelijk doormiddel van een spel hun gedrag aan het veranderen. Dus dat is ook zeker niet alleen gericht op kinderen, nee.
Ik heb voorbeelden van games waar een hele fysieke installatie omheen zat. Dat iemand met een spierziekte, bijvoorbeeld, zijn arm beter liet gebruiken. Er zat een hele constructie om de arm heen en door bewegingen van de arm te doen werd de patiënt getraind, en dat kan ook gewoon een volwassene zijn, maar dan moet er wel een aanleiding zijn om de bewegingen te maken. Daar heb je dan een spel voor, dan heb je een joystick in je hand, en dan moet ik mijn hele hand bewegen. Dan ben ik aan het bewegen, maar ik ben eigenlijk een spel aan het spelen. Dus dat is zeker niet alleen voor kinderen.

8. [14:22] Je hebt het nu over hele haptische spellen. Houden jullie ook rekening met de fysieke omgeving van de speler?

Ons bedrijf niet echt. Wij zijn meer gericht op tablets en telefoons en minder fysieke installaties. Alhoewel we dat wel graag zouden doen. Maar omdat iedereen een telefoon in zijn zak heeft, bijna iedereen heeft tegenwoordig een tablet. Het is vanuit commercieel oogpunt gewoon makkelijker om die mensen te bereiken. Het is natuurlijk ook zo dat op het moment dat je een fysieke installatie bouwt, dan staat hij op één plek. Dan kan hij alleen daar gespeeld worden. Dat is een afweging die je maakt als bedrijf zijnde.

9. [15:11] Als je een serious game gaat ontwikkelen, wat is dan het proces? Waar begin je?

Wij beginnen eerst met een klant. Die heeft een vraag van ‘Hey, ik heb een probleem. Ik heb een groep psychiatrische patiënten en de begeleiding daarvan kan beter. Ze zien elkaar elke week, elke twee weken, maar daartussen mis ik ze. Daartussen heb ik geen contact met ze.’ Nou, die vraag wordt bij ons neergelegd en dan wordt er doormiddel van het management team wordt er een eerste concept bedacht. ‘We zouden iets in deze richting kunnen doen, iets in die richting kunnen doen.’ Dan wordt er een heel grof idee gevormd.
Daarna wordt het team erbij gehaald. Ons team bestaat helemaal uit kunstacademie studenten. Die hebben sowieso een conceptuele inslag, van naast het werk wat ze doen. ...[16:16-16:17 slecht te horen]... Dan wordt het concept verder uitgewerkt in iteraties.
De eerste iteratie is een brainstorm sessie: ‘Wat kan ik er allemaal mee? Wat wil de klant eigenlijk? Wat voor doelgroep hebben we?’ Dan gaan we een soort onderzoek in. Op het moment dat dat concreter en concreter wordt gaan we delen van de functionaliteit gaan we invullen. Eerst op papier, dat kunnen complete flowcharts zijn maar dat kunnen ook steekwoorden zijn. ‘Ik wil dat mijn patiënt kan communiceren met zijn mede patiënten zonder daarvoor in de therapieruimte te zijn.’ ‘Ik wil dat de coach, want elke therapiegroep heeft een coach, ik wil dat de coach individueel contact kan onderhouden met zijn patiënten. Zonder dat ze fysiek contact hebben’
Nou, zo stellen we allemaal regels op waar de app aan moet voldoen, en elke regelset gaan we verder uit kristalliseren wat dan uiteindelijk features gaan worden van de app. Op het moment dat het team een bepaalde feature heeft uitgewerkt en uitgedacht, dan gaat het terug naar het management, die kijkt er nog een keer naar, dan gaat het eventueel nog terug naar de klant van ‘Hey, wat vinden jullie hier van?’ En op het moment dat je daar een ‘go’ op krijgt ga je dat stuk ontwikkelen. Zo kun je dat eigenlijk met elke feature, kun je dat doen. Dat kan dan klein worden terug gebracht tot ‘Ik heb een knopje nodig om snel hulp in te roepen,’ dat kan door het team worden uitgedacht, of door een subset van het team kan dat worden uitgedacht. Op het moment dat dat uitgedacht is gaat het terug naar het management, dus dat zijn ook iteraties.
Je bedenkt iets, je bouwt het uit tot een concept, het wordt vorm gegeven het wordt ontwikkeld, dan bedenk je weer iets, en daar wordt dan steeds uit opgebouwd. Net zolang tot het product af is. Of net zolang tot de klant zegt: ‘Dit is klaar om te gebruiken.’
Dan kan daarnaast nog een hele wenslijst liggen van ‘Dit zou eigenlijk ook nog fijn zijn.’ Dan kan je daar verder ook op gaan itereren. Op basis van wat je al gedaan hebt.

10. [18:32] Hoe weet je dat het werkt? Hoe test je de effectiviteit van je game?

Nou, omdat je alles heel erg opknipt in hele kleine stukjes. Je hebt een heel groot idee: ‘Ik wil mijn patiënten kunnen monitoren.’ Dat bouw je op in hele kleine stukjes, en elk klein stukje moet eigenlijk heel erg op zichzelf staan. Dat moet kleine functionaliteit zijn die op zichzelf testbaar is. En op zichzelf toepasbaar is. Nou op het moment dat het toepasbaar is kun je het gaan testen. Dat doe je dus met een testgroep, in dit geval, de app waar we nu mee bezig zijn. Dat is gewoon een groep patiënten die dat gewoon uit probeert. Op een gegeven moment zijn we op zo’n moment dat het geheel, in zijn geheel, bruikbaar is. Dan gaan ze dat testen, daar komt dan weer feedback op terug, en dan gaan we die iteratie weer in.

11. [19:25] Wat voor feedback is dat dan? Ze praten met jullie? Er komen cijfers terug over hoe ze het gedaan hebben? Hoe moet ik dat voor me zien?

Kan ook, ja. Het hangt er van af wat we op dat moment willen testen. In dit geval is het zo dat we de complete ervaring aan het testen zijn, dus dat duurt een maand. Daarna gaan we met een deel van de groep zitten van ‘Hey, hoe ging het? Wat ging er mis? Wat ging er goed?’ De coach die dat begeleidt, daar hebben we dan contact me, dus die onderhoudt dat ook. Daarvan krijgen we gaandeweg al feedback van ‘eigenlijk gaat dit niet zo goed, of dat zou beter moeten.’
Dat hangt er van af wat we willen testen. Als we echt data willen hebben dan bouwen we functionaliteit die die data vergaat, dan kunnen we dat geautomatiseerd geleverd krijgen. Of we hebben gewoon fysieke gespreken. ‘Dit ging zo, dit ging zo. Hoe kan het beter?’

12. [20:29] Je bent waarschijnlijk bekend met het media-effecten debat. [Ja.] Nou, dat heeft een negatieve connotaties. Als je in een game een pistool afvuurt, ga je dat in het echt ook doen en dan krijg je Columbine situaties. Maar als je een game ontwikkeld voor educatie, probeer je eigenlijk ook gevolgen te creëren in de ‘echte’ wereld. Hoe zie jij die grens?

Oh, dat vind ik een mooie, dit. Even denken. Ik denk dat heel erg de link gelegd wordt tussen als je het hebt over gewelddadige games en gewelddadige films of muziek, dan denk ik aan Marylin Manson die verantwoordelijk werd gehouden voor de Columbine incidenten. Als je een educatieve game maakt, of een serious game maakt, dan heb je een verbetering van de gebruiker voor ogen. Dus je wil graag beter gedrag gaat vertonen, beter gaat leren rekenen, je hebt echt een specifiek doel voor ogen. Dat je gaat aanpassen in de werkelijkheid. Ik denk de meeste games die gemaakt worden voor entertainment, die hebben pure entertainment waarde. Zoals films dat hebben. Ik heb zelf heel veel gegamed, dan speelde ik first-person shooters enzo, dat is vrij gewelddadig. Dat was voor mij een manier om mijn gedachten ergens anders op te zetten en de druk van de ketel halen. Dus ik vond entertainment games zeker... Die hebben ook een doel in de ‘echte’ wereld, namelijk om mijn frustratie weg te nemen. Dat kan heel vaak uitgelegd worden als, als ik zie dat er met een gun geschoten wordt, dan ga ik dat herhalen. Ik denk als je goed genoeg geleerd is om het onderscheid tussen fictie en non-fictie te zien, dat je weet dat dat niet de echte wereld is. En dat je weet dat dat puur een uitlaatklep is.

13. [22:56] Ok, dus die verantwoordelijkheid ligt buiten de game? Die ligt er bij hoe mensen uitleggen wat fictie- en non-fictie is?

Ja, dat denk ik wel. Dat kun je heel ver trekken, dan haal ik weer even Marylin Manson naar voren. Het is van de zotte dat iemand denkt omdat er één iemand op een podium staat en die schreeuwt dat verlichting behaalt wordt doormiddel van zelfkastijding, ik noem maar even iets heel grofs, de link leg je eigenlijk een soort van al, dan nog steeds ben ik

als persoon diegene die daar iets mee doet. Wat zijn idee daarbij is, dat moet hij weten. Wat ik daar als persoon mee doe, dat is een heel andere discussie. Dat is hoe ik de wereld interpreteer en wat ik zie als goed en fout.
Op het moment dat je die verantwoordelijkheid bij de gebruiker, en dan denk ik even aan iemand die muziek luistert, ook als gebruiker van muziek, of iemand die naar film kijkt, de gebruiker van de film, op het moment dat het *daar* niet goed ligt, dan krijg je daar conflicten mee.
Dus waar ik eigenlijk naartoe wil is dat ik vind dat iemand die een geweer oppakt omdat hij dat in een film gezien heeft, die heeft de connectie niet begrepen. Die heeft niet begrepen dat een film bedoeld is om jou frustratie weg te nemen, of jou te laten nadenken over dingen, of jou te beïnvloeden, maar wel op jouw eigen manier.
Als ik dan naar serious games kijk, die hebben heel specifiek als doel om jou te laten nadenken over dingen, maar jou ook daarmee te laten doen wat jij daarmee wil doen. Daarbij zie ik geweld niet heel snel terugkomen in een serious game. Dat is niet helemaal hetzelfde, maar jou vraag van 'in hoeverre heeft dat invloed?' ja, ik denk zeker dat dat invloed heeft, maar dat hangt er maar net van af hoe de gebruiker daarmee omgaat.

14. [25:02] Er zit dus niets in de structuur van de game die ook beargumenteerd wat je ermee wil bereiken, of is dat bij serious games juist weer wel een optie?

Ja, dat is meer een optie. Veelal omdat het door... De inhoud gemaakt wordt door docenten of mensen die cognitief weten waar ze mee bezig zijn. Dus daar zit dan een groter idee achter, ja. Ja, dat denk ik wel. Serious games zijn denk ik op dezelfde manier opgebouwd als dat een lesmethode is opgebouwd. Daar zit ook een idee achter, een soort van opbouw achter. Een verloop. Die zitten in serious games ook, juist omdat ze in samenwerking met docenten gemaakt worden en met experts in het vakgebied.

15. [26:05] Wat zijn jouw gedachten over gamification? Dan bedoel ik niet een game bouwen, maar iets nemen wat geen game is en daar game elementen tegenaan gooien. Werkt dat?

Ja, ik denk dat dat zeker werkt. Nou vind ik gamification... Het is een term die je heel vaak hoort en die redelijk zegt waar het voor staat. Maar ik denk dat het... Kijk, wat een game zo gaaf maakt is dat je continu uitgedaagd wordt en continu geprikkeld wordt om nieuwe dingen te gaan doen.
Dus je zou bij wijze van spreken een supermarkt kunnen gamificeren door er een soort van uitdaging in te stoppen. Dan denk je al snel aan: 'loop het snelst door de supermarkt en vind al je boodschappen,' maar dat kan natuurlijk ook op een andere manier. Zonder dat het 'gamey' aanvoelt. [dus het is meer dan alleen een puntensysteem?] Het is meer dan alleen een... [puntensysteem], ja, nee, zeker. En dan komen er inderdaad ook weer psychologen en cognietieven om de hoek kijken. Die hebben natuurlijk een idee van hoe het brein werkt. En boe het brein geprikkeld wordt. Op het moment dat je die mensen los kan laten op alledaagse dingen, dan kan het leven een stuk interessanter worden. Omdat het veel leuker is om met boodschappen om te gaan. Dat zou je ook veel leuker kunnen maken, denk ik. Ik weet niet precies hoe, maar ik kan me zeker voorstellen dat dat kan.
Daar wordt vaak de term 'gamification' aangehangen, maar ik denk dat het vooral het prikkelen van mensen is. Het uitdagen van mensen. Wat voor mij een game behelst is het uitdagen van iemand om verder te komen in wat die doet. Nou goed, dan zou fitness ook een gamification vorm zijn. Van je eigen lichaam. Dat kan zeker op alles toegepast worden, denk ik.

16. [28:11] Als je driedeling hebt van: full-fledged games die ze proberen om te vormen voor educatie, games die echt ontwikkeld worden voor educatie (serious games) en als laatste een 'traditionele' lesmethode waar gamification op toegepast is. Hoe zie jij die verschillen, is de een effectiever dan de ander?

Het ligt natuurlijk wel een beetje in mijn straatje om te zeggen dat de tweede variant effectiever zou zijn, maar dat komt denk ik omdat op het moment dat een serious game gemaakt wordt, het grote verschil met een entertainment game is dat het een lerende grondslag heeft. Het is niet gebouwd om te entertainen, het is gebouwd om te leren. Dat doe je doormiddel van entertainment. Maar het fundament is anders.
Zo zal een, hoe je het ook went of keert, hoe een leuke serious game je ook maakt, het zal nooit een zwaar interessante entertainment game worden. Het doel is gewoon anders in den beginne. Omdat het fundament anders is ga je een ander huis bouwen. Als je het legt naast een normaal, nou ja, normaal, een papieren methode. Die heeft natuurlijk dezelfde grondslag. Het is in het begin gemaakt door dezelfde mensen, maar papier heeft gewoon zijn beperking. Zoals ik in het begin al heb proberen aan te geven, dat we het veel meer... Toch stukken leuker kunnen maken om dingen te gaan leren. Waardoor het effectiever wordt.
Dus ik denk dat als je een papieren methode neemt die heel goed is uitgewerkt, heel goed bedacht is, dan is hij het effectiefste, als je de menselijke factor uitsluit, zeg maar. Maar als je wil dan een mens ermee overweg kan dan heb je toch net even iets meer entertainment erbij nodig om het over te brengen.
Daarom denk ik dat een serious game, vanaf de grond af aan ook bedoeld als serious game, het effectiefste is. Omdat het het beste van twee werelden kan zijn.

17. [30:50] Ben je bekend met *Code Academy*? [Ja.] Wat vind je van een platform als dat?

Ja, geweldig! Supergaaf. Ik gebruik het zelf regelmatig. Dat is eigenlijk de gamification van het leren code schrijven. Op twee manieren is dat heel interessant: het is leuk om te doen en het werkt ontzettend soepel en je hebt er vrij weinig basis kennis voor nodig om te beginnen. Dus je succeservaring is meteen al gezet. Het is overduidelijk gemaakt door mensen die weten waar ze het over hebben, dus de educatieve waarde is gewoon heel hoog. Het is leuk om te doen omdat je vrij snel succeservaringen hebt. En het is makkelijk toegankelijk om te doen. Ik hoef niet een complete software development kit te installeren om überhaupt mijn eerste 'Hello World' voor elkaar te krijgen. Dat is allemaal al voor je gedaan. Ja, ik ben ontzettend voor zulk soort vormen van educatie.

18. [32:00] Hoe ver denk je dat je zoiets kan doortrekken? Denk je dat met zo'n game alleen de beginselen kan aanleren of kan je daar een professional mee trainen?

Pfoe. Daar *kun* je heel ver in gaan. Maar dat hangt er maar net van af hoe goed diegene is die het heeft opgezet. Je zou bij wijze van spreken gewoon je niveau kunnen ophogen. Steeds moeilijkere cases voorleggen. Voorwaarde daarvan is wel dat het ding wat je maakt steeds complexer wordt. Dat moet het systeem dan wel aan kunnen. Dus dat is al een voorwaarde.
Hoe *Code Academy* nu vaak werkt is, scripts die uit één file ontstaan en die draaien. Maar als je kijkt naar hoe wij software schrijven, dat gaat in de honderden files zitten. Dus dat wordt ingewikkelder om dat bij te houden. Dan zul je ook eerder, op het moment dat je op een hoger niveau komt zal je abstracter gaan lesgeven. Dan zul je meer ideeën proberen over te brengen, en niet zozeer voorbeelden aan te trekken.

19. [33:14] Dus om dan even terug te komen op waar we het eerder over hadden: dan ga je in plaats van een specifieke context steeds meer abstracte modellen proberen aan te leren?

Ja, maar dat is dan juist wel weer voor die doelgroep een hele specifieke context. Dan hoef ik niet per sé de syntax van een programmatuur aan te leren, want dat heb ik al gehad in het begin. Dan ga ik opeens de architectuur van software aanleren. Ik denk dat op het moment dat je iets verder komt, dat dat dan belangrijker wordt. Want de syntax, ja, ik heb *Google* naast me open staan. Als ik even iets niet weet, dan zoek ik het op. Dus dat is niet heel erg interessant meer. Het is interessanter om te zien: 'Hoe ga ik om met de structuur van data,' of 'hoe ga ik om met memory-management?' En dat zijn weer concepten die niet in één stukje code te vangen zijn, maar waar een visie overheen gaat. En ik denk dat op het moment dat *Code Academy* dat wil aanbieden, dan zul je meer naar een *Kahn Academy*-achtige structuur gaan.
Ik weet niet of je *Kahn Academy* kent? [Nee.] *Kahn Academy* is briljant. Ik ben helemaal fan. Het is begonnen als een Iraniër die wilde zijn neefjes lesgeven, wiskunde huiswerk, meehelpen. Hij zat in Amerika aan de andere kant van de wereld, en we hadden Internet. En we hadden *YouTube*. Wat ging hij doen? Hij ging de cases die zij aangaven; 'Dit en dit moet ik leren, maar ik snap het niet,' ging hij in een filmpje van maximaal tien minuten, ging hij het uitleggen. Met voorbeeldjes, met gesproken tekst, met beeld, ging hij uitleggen wat de bedoeling was. Die zette hij op *YouTube* en zijn neefjes konden dat zien. Hij zag geen reden om dat 'private' te maken, dus dat is publiekelijk.
Eigenlijk, wat hij deed, was de basis-beginselen van algebra uitleggen. Want daar waren zijn neefjes toevallig mee bezig. Nou die neefjes keken het, en steeds meer mensen gingen dat eigenlijk bekijken. Veel mensen, en nog veel meer mensen. Want wat was het fijne daar aan? Zoals ik in het begin al zei: een filmpje kun je op je eigen snelheid bekijken. Het ene neefje was iets sneller dan het andere, maar dat maakte niet uit, want dan gingen ze even terug naar wat er al verteld was. Dan kon hij het op zijn eigen tempo doen.
Nou, dat is uiteindelijk uitgebreid tot het grootste, denk ik, kenniscentrum op wiskundegebied. Je moet maar eens kijken. *KahnAcademy.com*. Alles wat je wil leren, en heb ik het echt over alles wat met wiskunde te maken heeft, kun je daar vinden. Vanaf optellen en aftrekken tot echt de meest ingewikkelde algebra en de meest ingewikkelde algoritmes. Alles kun je vinden. In stapjes van tien minuten. En daar kan de professional zeker nog dingen bij opsteken.
Dus de vraag: kan *Code Academy* dat? Ja, dat kan. Het is dan maar net de vorm waarin je dat dan gaat doen. Hele ingewikkelde principes uitleggen doormiddel van video kan makkelijk.

20. [36:38] Nu hebben we het over het verschil tussen basis beginselen leren tot heel professioneel. Denk je dat je met een game een vol ambacht kan leren? En een volgende vraag: wat denk jij dat een ambacht inhoudt?

Ja. Nou, dan denk ik bijvoorbeeld aan technieken die chirurgen gebruiken tegenwoordig. Nou ben ik laatst niet in een behandelkamer geweest, maar ik zie wel steeds meer berichten voorbij komen van chirurgen die met een soort joystick met een cameratje naar een beeldscherm zitten te kijken om maar heel precies een beweging uit te voeren. Een heel ingewikkelde bypass operatie kunnen ze doormiddel van techniek veel beter doen, dan dat ze dat met de hand zouden kunnen. Tja, dan zit je naar het scherm te kijken en dan zit je met twee joysticks, dat kun je prima simuleren.
Zoals een straaljager piloot ook de eerste helft van zijn opleiding in een simulator doorbrengt voordat hij daadwerkelijk in een vliegtuig zit. Dan denk ik dat het wel een ambacht is.
Maar als je het hebt over houtbewerken of metaalbewerken? Dan wordt het anders denk ik, want dan is het veel meer de tactiele feedback die je terug krijgt van het object waar je mee bezig bent, die belangrijk wordt. Ik denk dat de techniek, nu zeker, lang o ver nog niet is dat het de werkelijkheid kan reproduceren. Waardoor je dus ook... Wanneer ik aan een ambachtsman denk, dan denk ik aan een meubelmaker. Die zijn houtkennis heeft. Die voelt hoe hout voelt. Die voelt wat de verschillende verschijningsvormen van hout zijn. Zolang een computer dat niet kan reproduceren kan je het niet leren doormiddel van een game.
Ik zei net: alles is te gamificeren. Het uitdagender maken om dat te leren, ja dan heb je een soort van gamification ervan, maar dat is niet doormiddel van een serious game. Nee, ik denk waar de digitale techniek tekort schiet om genoeg informatie te leveren, dan is een serious game niet geschikt, maar je kan wel gamification principes er op toepassen.

21. [39:18] Een ambacht heeft dus heel veel te maken met materiële gewaarwording?

Ja, ik zit er nu eens over na te denken en ik denk dat... Sommige wel, sommigen niet. Zoals traditionele ambachten, dingen maken, ja, dan moet je gewoon kennis hebben van berkenhout ten opzichte van beukenhout. Zonder dat te voelen denk ik niet dat je dat kan leren.

22. [39:45] Je programmeert zelf ook, een programmeur is volgens jou dan dus geen ambachtsman?

Jawel. Ja, het hangt van het ambacht af. Op het moment dat je met fysieke dingen bezig bent denk ik dat digitale techniek te kort schiet. Maar daar waar digitale technieken prima kunnen volgen, ja, dan is het meteen te leren doormiddel van een game.

23. [40:15] Dus volgens jou zouden games geschikt kunnen zijn om iemand het ambacht der programmeren aan te leren?

Oh ja. *Code Academy* goed voorbeeld. Ja, ik denk dat dat kan. Voorwaarde is natuurlijk wel dat je er een interessante game van maakt. Bijvoorbeeld dat je een leuke setting creëert, betekend niet meteen dat je ook daadwerkelijk informatie overbrengt. Je moet wel weten waar je het over hebt, je moet wel een goed fundament hebben. Op het moment dat je dat hebt is het best goed te doen.

24. [40:54] Ok, even terug kijkend op waar we het over gehad hebben. Je hebt het gehad over video's die je kan herhalen en vertragen. Je kan mensen motiveren door lessen interessant te maken. Je kan het er leuk uit laten zien. Wat zou jij noemen als de belangrijkste factoren in het ontwikkelen van een serious game; heb ik die nu opgenoemd?

Ik denk dat je coöperatief leren mist. Ik denk dat dat het belangrijkste is wat het het interessantste maakt. Samen met mensen leren, ik denk dat dat het belangrijkste is.

25. [41:42] Zijn games cruciaal om te leren?

Ooh, dat is een gewetensvraag. Nee, games zijn niet cruciaal om te leren. Ik geloof dat mensen al 3000 jaar aan het leren zijn. Serious games waar we het nu over hebben zijn pas van de laatste tien jaar. Dus ze zijn niet cruciaal om te leren, maar ze maken het wel verdomde makkelijk.

26. [42:14] Als we het hebben over traditionele klaslokalen waar ze nog niets met games doen, zou het daar een verrijking zijn? [Ja.] Hoe zouden we dat dan integreren?

Ik raad je nog een keer aan om *Kahn Academy* te gaan zien. Je moet het klasikaal omdraaien. Je moet kinderen thuis de stof laten leren, doormiddel van filmpjes, doormiddel van installaties wat mij betreft. En op school moet je het er samen over hebben, samen tot een hoger plan komen.
Kinderen kunnen elkaar dan helpen. Nu heb je een docent die voor de klas staat, en die broadcast eigenlijk zijn idee. Dus je geeft alle brokjes informatie: 'en zo is het.' Maar ik denk dat het leerproces van een kind niet zo werkt. Het leerproces van kinderen is dingen zelf ontdekken, op zijn of haar eigen tempo. Ik denk dat dat echt radicaal anders moet. En ik denk dat dat kan, doormiddel van tablets en computers, kunnen ze zelf op hun eigen niveau, hun eigen tempo, een leerproces doorlopen. Op het moment dat ze dan op school samen zijn, het er samen over hebben, en er samen mee gaan spelen, wat mij betreft, dan denk ik dat je zal zien dat het leerproces sneller gaat. Op het moment dat het kind het even terug wil trekken en het op zijn eigen tempo wil leren, omdat het te snel gaat, moet dat ook kunnen. Dus dan is de docent ineens geen leidende rol meer, maar een ondersteunende rol. Ik hoop niet dat ik nu een hoop docenten tegen de benen schop, maar ik denk dat we uiteindelijk die kant op moeten.

Transcript interview Jan Willem Huisman & Evert Hoogendoorn, IJsfontein

Interview date: June 27, 2013, 13:23 - 13:02

Location: *Mediacafe Westergasfabriek*, Amsterdam

(Huisman referred to as: "JW", Hoogendoorn referred to as: "E")

1. [00:30] Laten we een beginnen met hoe jullie heten en wat jullie precies doen.

JW: Oh ja, heel slim. Ik heet Jan Willem. Ik ben oprichter en creatief directeur van *IJsfontein*.

E: Ik ben Evert Hoogendoorn en ik ben strateeg serious games. Wij zijn beide... Bij allegames die *IJsfontein* maakt, is één van ons altijd minimaal betrokken bij hoe het uiteindelijk, hoe het op zijn minst uiteindelijk de deur uit gaat. Maar over het algemeen wat eerder.

2. [01:01] Wat zijn projecten waar jullie recentelijk aan gewerkt hebben?

E: Nou, even kijken, een lijstje. Eén van de dingen waar ik het meest mee bezig ben geweest is *ABCDEsim*, een simulatie voor spoedeisende hulp. Het is een game die we oorspronkelijk hebben gemaakt voor artsen op de spoed-eisende hulp om een protocol om patiënten te stabiliseren beter te leren. En daar zijn spin-off's voor gemaakt voor verpleegkundigen, nu voor huisartsen, en er komt er eentje aan die specifiek gaat over kinderen. Die is wel weer voor artsen. Over open brandwonden... Nou, zo bouwen we voort op het thema. Twee andere projecten waar ik mee bezig ben: één is om aan cannabis verslaafde jongeren te helpen in hun behandeling. En een andere is om VMBO leerlingen te leren over financieel besef. Wat is het verschil tussen lenen en een hypotheek? Hoe lees je een balans? Dat soort dingen.

JW: En een *iPad* spelletje waar kinderen spelen met de *Tumblies*. [Wat zijn de *Tumblies*?] Kleine karaktertjes [uit een] animatieserie.

E: Ja, pre-school animatieserie, het is dus eigenlijk een ...[02:28 woord slecht te verstaan]... case, *iPad* game voor kinderen van twee tot vijf.

[02:32] Dus jullie hebben veel projecten tegelijkertijd lopen?

E: Ik heb er op dit moment zeventien.

JW: Zo, dat zijn er dus ongeveer twaalf teveel, maar dat is een ander gesprek.

E: Ja.

[Beide grinniken]

3. [02:47] Wat is het standaard proces als jullie een opdracht binnen krijgen? Of gaan jullie naar bedrijven toe? Hoe gaat dat in zijn werk?

JW: Meestal komen bedrijven naar ons. Dat is in denk ik zo'n 90% van de gevallen. Eerst proberen we er achter te komen wat de doelstellingen zijn. Wat wil je eigenlijk bereiken? En is dat wat je zegt ook wat je daadwerkelijk wil bereiken? Dan gaan we met inhoudsdeskundigen praten, dus mensen die heel veel van dat onderwerp weten. Zo dat we begrijpen: "Wat is nou eigenlijk de essentie van het onderwerp, waar gaat het nou eigenlijk over?" Dan gaan we proberen om dat te vertalen naar een set spelregels.

E: Eerst naar gedrag.

JW: Eerst naar gedrag. Dan naar een set spelregels. Dan gaan we de spelregels zoveel mogelijk proberen te testen in een paper prototype. Dus voordat we gaan programmeren gaan we spelen. Dan gaan we het aanpassen, dan gaan we het bouwen en dan gaan we het implementeren. Dan gaan we bij sommige projecten ook kijken wat dat ontluikt: Hoe spelen mensen het? Wat kunnen we daarvan leren? En soms gaan we dan opnieuw.

E: Die cirkel 'opnieuw', als het goed is, in 90% van de gevallen beperkt die zich tot een stuk na het proces. Dus je blijft je concept verbeteren. Want als je er achter komt dat het écht niet werkt, moet je helemaal terug naar: "hebben we het gedrag wel goed gedefinieerd?" De versnelling aan het einde, die implementatie, die komt niet alleen maar aan het einde. Bij wenselijk gedrag, dat proberen te formuleren, hoort ook dat je de context waar in mensen spelen definieert. Daaruit komt vaak voort welk platform we kiezen. Dus: wil je browser-based, wil je een tablet? wil je een telefoon? Dus uit die implementatievraag komen al heel veel randvoorwaarden voor wat uiteindelijk het concept wordt.

5: [05:19] Kan je daar een voorbeeld van noemen?

E: Als je aan mensen gaat vragen om het in hun vrije tijd te doen, dan moet je gaan kijken naar: "wanneer hebben die mensen hun down-time?" Dat ze dat zouden kunnen. Dat betekent vaak dat je niet een game kan maken die drie-uur speel-sessies heeft. Het kan zijn dat je op basis daarvan bepaalt: "het moet wel in je vrije tijd, maar wel onder werktijd." Dus dan maak je een iPhone game. Omdat ze dat dan in drie minuten sessies dat bij de koffie-automaat kunnen doen, dan gaat het niet ten koste van hun werk. Of je zet een game- time cap van vier minuten op een spelsessie. Dat soort keuzes maak je allemaal al aan het begin als randvoorwaarde. Vanuit de implementatie.

6: [06:05] Hoe test je dan de effectiviteit van zo'n game? Hoe weet je dat iemand er iets van geleerd heeft?

E: Dat hele validatie-vraagstuk zit vervlochten in het hele proces. Dus op basis van de paper-prototype beslissen we of de mechanic klopt. Op basis van het om leren van het gedrag en daarbij het kiezen van je concept, dus het omzetten naar een set regels of mechanics, daar zit een validatie-stap in op basis van literatuuronderzoek. Op basis van literatuur, op basis van ervaring. Maar je moet kunnen beargumenteren waarom je denkt dat deze mechanic past bij het gewenste gedrag of bij het huidige gedrag, omdat het profiel daartussen eigenlijk is waar de game over gaat. Vervolgens als de game af is kan je nog, maar dat is meestal in de handen van de opdrachtgever zelf, een validatietraject starten waarin je met een experimenteel onderzoek, of een nul-meting, en een controlegroep en de hele keet stappen zet.

7: [07:18] En komt dat vaak voor?

E: Dat komt steeds vaker voor, maar dat is een vrij langdurig proces. Dat duurt gemiddeld twee jaar, en je moet wel een opdrachtgever hebben die adem heeft. Zoals een *ABCDEsim,* van een ziekenhuis, lange termijn project, elk jaar komen daar drie spin-off's van, dus daar is het waardevol om dat te doen. Als je het doet voor, weet ik veel, een consumptiebedrijf, die hebben gewoon niet de tijd voor twee jaar.

8: [07:46] Zijn die spel-elementen leeftijdsgebonden, of maken jullie spellen die voor iedereen speelbaar zijn?

E: Spellen maak je altijd leeftijdsgebonden. Probeer ze zo goed mogelijk te formuleren, en dan heb je niet te maken met alleen maar leeftijd. Eén van de minst belangrijke factoren, eigenlijk. Je hebt heel wat te maken met gender, je hebt te maken met: "Wat voor speler type is iemand?" Bij een bank heb je veel meer mensen die op competitie drijven dan in een ziekenhuis, maar als je het chirurgisch maakt, is het juist weer heel erg competitie gedreven. Als je voor volwassen werkt, laten we zeggen vanaf achttien jaar, maakt leeftijd niet zo heel erg meer uit. Maar het verschil tussen zes en twaalf dat is wel heel groot. Dat zit met name in abstractievermogen en in fijne motoriek. Dus een game voor twee tot vijf jarigen, ja, die range in fijne motoriek is niets. Maar als je eenmaal boven de achttien bent maakt het niet meer zo heel veel uit. In marketing wordt daar nog heel erg aan gehangen, maar wij maken eigenlijk niet heel veel mee. De selectie die door de opdrachtgever is gedaan is dan zo groot dat die leeftijd verwaarloost kan worden.

9. [09:12] Als je dingen aan probeert te leren in een traditionele klaslokaal-setting werk je veel met discipline. Je doet iets goed of je doet iets fout. Zit dat ook in jullie spellen verwerkt, of zijn ze meer contextueel, zoals het Montessori model?

E: Je doet het wel goed en fout, maar je doet het goed en fout, gemeten naar je eigen strategie. Dus je ontwikkelt. De game stelt jou doelen, die moeten bedrijven ontwikkelen tot hun eigen strategie, en in de executie daarvan kan je het fout doen. En daar krijg je dan direct feedback op. Maar wij gaan niet zeggen: de handeling is fout.

10. [09:58] Het is vaak zo bij games dat als je een bepaald level niet doorkomt, dat je vast loopt en hetzelfde stuk steeds opnieuw moet doen. Betekent dit dat tragere leerlingen de game niet uit kunnen spelen, of zorgt die herhaling er juist voor dat het op een gegeven moment opgepikt wordt?

E: Nou, de definitie van een wat tragere leerling is in het onderwijs... In het onderwijs word je meestal gedwongen om de favoriete strategie van de leraar of van de methode te zien. En als je in die strategie niet zo goed bent, dan

ben je een trage leerling. Een goede game geeft je zoveel alternatieven daar op, dat je met het vinden van een andere strategie toch weer bij kan komen. Dus als de leerling wat flexibeler is, word je veel minder vaak een trage leerling.

11. [10:54] Zou je zeggen dat games cruciaal zijn om te leren?

JW: Spelen is cruciaal om te leren, en games zijn daar een vastgelegde vorm van.

12. [11:10] Hoe zou je een game beter kunnen integreren in het klassieke lessysteem?

JW: Ik denk dat heel veel mensen games om de verkeerde redenen inzetten. Je ziet dat heel vaak games in het onderwijs worden gebruikt om de verkeerde redenen, dus: 'kinderen vinden het leuk om te spelen'. Terwijl wat Evert zegt, het is belangrijker dat je in een game je eigen strategie kunt kiezen. Dus dat je je eigen leerpad creëert terwijl je aan het spelen bent. Belangrijk in een spel is dat je heel veel korte feedback loops krijgt. Dus dat je op je gedrag en wat je doet feedback krijgt. Dat zijn allemaal aspecten waarom een game een heel krachtig instrument is. Maar dat zijn allemaal aspecten die over het algemeen niet gebruikt worden voor games die in het onderwijs ontwikkeld worden. Om je een voorbeeld te geven, we waren bezig met een game voor het basisonderwijs. Die ging over breuk, die kinderen waren alsmaar aan het leren leren breuken, en dan die opdrachtgever zegt: "Ja, maar er moet wel een toets. We moeten wel kijken of 'ie het kan of niet." Dat is een hele ouderwetse schoolmanier van denken. Ik weet namelijk heel erg goed wat hij kan, want ik ben continu zijn gedrag aan het toetsen en aan het aanpassen. Ik ben hem aan het aanpassen op zijn of haar kennisniveau. Dus die hele gedachte dat je die toets nodig hebt, bijvoorbeeld, toont aan dat je eigenlijk niet begrijpt wat er in het spelletje gebeurt. Want je wordt continu getoetst. Het systeem weet *precies* wat je kan, nog veel beter dan wanneer je een toets doet. Dat is maar een momentopname.

E: Om terug te komen op de *ABCDEsim*, een game voor spoedeisende hulp, dat zit in een traject waarbij je eerst E-learning doet, dan een game speelt en dan naar een live training mag. Dat is een heel intensief traject, en wat de game eigenlijk doet is dat minder kostbaar maken. Die live training is heel duur. En die wordt nu korter. De E-learning heeft een toets. Die E-learning... allemaal prima, maar het enige wat er gemeten wordt is die toets. De live training heeft ook een toets. Alleen de game niet. Want als je de game hebt uitgespeeld, dan ben je getoetst.

13. [13:52] Wat houdt die E-learning precies in?

E: Je krijgt een lap tekst met foto erbij, of een videootje, en vervolgens wat multiple choice vragen.

JW: En dan gaan ze E-learning gamificeren, want als je dan namelijk van de vijf vragen er vier goed hebt, krijg je een badge. En dan heb je ineens een game.

14. [14:17] Ik hoor hier Ian Bogost z'n: 'Gamification is bullshit' een beetje doorsijpelen?

JW: Gamification is vele malen complexer. Ik geloof heel erg in gamification. Alleen gamification is vele malen complexer dan de manier waarop het nu wordt uitgevoerd. Het verkrijgen van badges past bij één speeltype. Dat is één van de vier. Dat is 25%, nou het zal in percentage misschien een iets grotere groep zijn. Maar de andere drie vinden badges helemaal niet interessant. Ze krijgen een badge en denken: 'Nou, badge. Lekker belangrijk.' Zo zijn er heel veel mechanics die nu gebruikt worden in gamification die a. volgens mij maar beperkt houdbaar zijn, en b. maar een kleine groep van de hele populatie aanspreken. Dus als je alleen maar kijkt naar de trucjes die er zijn en die gaat toepassen, verander je eigenlijk in feite niets.

E: De meest gebruikte definitie van gamification is: "We leggen een game-laag op de werkelijkheid." Een betere definitie is, denk ik, en dat is eigenlijk de originele definitie maar die snappen marketeers niet, is: "We herontwerpen de werkelijkheid zodat we gebruik kunnen maken van de game-principes." Maar dat betekent dus dat je de werkelijkheid moeten herontwerpen.

JW: Ja, dus je moet heel het onderwijs ontwerpen.

E: Als je het onderwijs gaat gamificeren, moet je het onderwijs opnieuw ontwerpen, als game. En niet er een stickerlaag op plakken, waardoor het er uit ziet als een game.

15: [16:58] Je had het over bepaalde standaard mechanics die er gebruikt worden. Zou je er daar een paar van kunnen noemen? En waarom die niet houdbaar zijn?

JW: Badges zijn een veelgebruikte mechanic. Dat betekent: "Als je dat en dat en dat hebt gedaan, krijg je van mij een stickertje." Dus als jij heel vaak ergens komt krijg je een stickertje. Als je hoort dat iemand het woord 'zon' gebruikt, krijg je een stickertje. Dus dat is een mechanic. Er zijn heel veel mechanics.

E: Eigenlijk kun je mechanics definiëren als de kleinst mogelijke eenheden in een game. Kans is een mechanic. Maar er is bijna geen één spel wat alleen maar uit kans bestaat. Loterijen een beetje. Maar poker? Het heeft een kans element. Want het moment daarna, het moment nadat er gedeeld wordt, heeft eigenlijk een science mechanic. Het uitrekenen hoe het zit. Welke kaarten heb ik, welke kaarten zijn er gedeeld? Dan moet hij dus wel... Dan is de kans dat hij een goede hand heeft groter. Dat is een science mechanic.

JW: Je hebt ook een social mechanic.

E: De social mechanic. Met online poker gaat die er weer een beetje uit. Poker verandert heel erg doordat we het online spelen. De poker face is niet meer van belang.

JW: Het lezen van je tegenstanders. Wel jammer.

E: Maar je kan hem wel lezen in hoe hij speelt. Dus: "Oh wacht even. Altijd als hij drie keer past, dan gaat hij daarna risico nemen. Dat weet ik gewoon." Dan kan ik daar op mijn beurt in richten. Het sociale zit er in, competitie zit er in, dus dat zijn allemaal mechanics. Er zijn ook een aantal complexere. Gifting is een hele complexe, als ik jou iets cadeau geef. Dan vind jij dat heel erg leuk, en dan voel ik me weer beter omdat jij dat heel erg leuk vindt. Maar tegelijkertijd heb ik ook een contract met jou afgesloten dat jij ook iets terug gaat geven. En dat moet of weer dezelfde waarde hebben... [Tenzij je een vrouw in *World of Warcraft* bent] ... Tenzij je een vrouw in *WoW* bent. En ook daar in, alleen maar die vrouwen vinden dat geen contract. Die kerels die die vrouwen wat geven wel. En dat maakt het een hele interessante.

16: [18:49] Jullie hadden het net over gamification als het opnieuw opbouwen van de werkelijkheid. Als je dan een driedeling hebt: (I) full fledged entertainment games die voor educatie gebruikt worden, (II) serious games of (III) een klassieke lesmethode waar je wat game elementen aan toe voegt, wat werkt er dan het beste? Zouden er dingen sowieso geschrapt moeten worden?

JW: Van mij mag je die eerste sowieso wel schrappen. Maar ik weet niet of jij het daar mee eens bent [Evert]?

E: Nou, niet helemaal.

JW: Jammer.

E: Nou, waar het over gaat is, om te beginnen, en dat is iets wat in educatie eigenlijk nooit gebeurt, jij heel specifiek moet definiëren wat jij vindt dat er geleerd moet worden. En dat moet je definiëren in gedrag. Als je dat hebt en er blijkt bij de *Bart Smith* een game te liggen die precies dat doet, moet je dat vooral doen. Alleen over het algemeen wordt er niet gedefinieerd wat het gedrag is dat ze willen zien bij leerlingen. Ten opzichte van de content. En ontbreek te kennis, de échte kennis, van wat zo'n game doet. Want als je zo'n game neemt, dan kan het best zijn dat je leerdoelen er in zitten, maar dan zitten er ook nog een heleboel andere dingen in die je cadeau krijgt. En daar moet je wel ook weer je onderwijs op aanpassen. En vaak weten mensen dat niet. En dat betekent dat je, nou voorbeeld van vanochtend, je kan een heleboel leren over hoe een economisch model in elkaar zit van *Sim City*. Maar om dat te leren moet ik, weet ik wat, 48 uur spelen. Nou als ik het met een klas 48 uur kan hebben over hoe een economisch model in elkaar zit, dan ben ik er ook.

JW: Wel eerder.

E: Eerder. Dus dan is het wel een hele interessante manier, als je het wil onderzoeken of weet ik het wat, maar niet een hele efficiënte.

17. [21:06] En die gedragsmodellen die jullie noemen, hebben jullie daar standaard modellen voor of wordt dat per situatie anders beoordeeld?

E: Daar hebben we kennis voor. Wat we eigenlijk doen, is proberen die kennis die we zien samen met de opdrachtgever en domeindeskundigen vast te leggen in een model. Zelf maken we een soort werkdefinitie, een werkmodel. Of je haalt daar theorie bij, waarvan wij denken van: "Nou, dat is bij deze situatie passend."

18. [21:46] Maakt de context van de game uit? Welk verhaal je er aan vast plakt? Wat ik eigenlijk probeer te vragen: probeer je onderliggende modellen aan te leren of moet een game heel specifiek zijn toegepast op een situatie wil je daar kennis mee overbrengen?

E: Het maakt heel veel uit hoe jouw definitieve laag er uit ziet, maar dat maakt uit voor het perspectief wat de speler krijgt. Het maakt niet uit, je content verandert er vaak niet van. Ik wil een bepaald systeem uitleggen aan een leerling, nou dan kan ik dat met piraten doen of met buitenaardse wezens. Dat maakt niet zo heel veel uit. Als het allemaal jongetjes zijn die van piraten houden, dan is het handig om voor die piraten te kiezen. Wat wel uitmaakt, is maak ik hem de rover hoofdman, of het hulpje van de kok? Want die gaat zich op een andere manier gedragen. [dus je moet er een beetje tussenin hangen?] Nee, dat hoeft niet, dat is gewoon een keuze die je maakt.

JW: Je moet het er juist niet tussenin laten hangen, je moet kiezen. Want de rol die jij iemand geeft in het spel is ook de rol die hij gaat aannemen in het spel. Dus als jij wil dat iemand als rover hoofdman bijvoorbeeld heel erg beslissingen neemt, een bepaald profiel heeft, dan moet je hem rover hoofdman maken. Als je wil dat hij wat meer gaat leren van, dan maak je hem het knechtje van. De rol die je iemand geeft zorgt ook voor het gedrag wat hoort bij die rol.

E: En de strategieën.

JW: En de strategieën die je gaat ontwikkelen.

E: Als je iemand James Bond maakt, een geheim agent, dan gaat hij dingen geheim houden voor alle spelers. Dat krijg je cadeau bij dat perspectief. Die gaat in zijn eentje handelen. En dat kan heel bruikbaar zijn voor sommige dingen. Als ik dat niet wil, ja dan is het gewoon niet zo slim om een geheim agent te nemen als metafoor.

JW: Nee, dan doe je Charlie's Angels.

E: Dan doe je Charlie's Angels, en dan moeten ze met zijn drieën. En dat geheim houden, weet je wel, daar kan je in ... [24:01-24:02 slecht te verstaan]...

19. [24:09] Jullie zijn allebei bekend met het media-effecten debat neem ik aan? [Ja] Als je gewelddadige games speelt wordt je gewelddadig. Nou, dat kunnen we aan de kant zetten. Dat is niet zo. Maar als je iemand iets probeert te leren met een game, probeer je een effect uit te lokken. Je probeert gedrag aan te passen. Dus hoe zit die tweedeling? Mogen we het media effecten debat wel aan de kant zetten?

JW: Daar hadden we het toevallig vanochtend nog over. Ik denk niet dat je het aan de kant moet zetten. Alleen ik denk dat die veel te zwart-wit gevoerd wordt. Ik denk dat dat het grootste probleem eigenlijk is, maar dat komt omdat mensen het niet fijn vinden om naar een genuanceerd verhaal te luisteren. Volgens mij is dat meer het probleem van het debat.

E: Ja, en het debat wordt gevoerd door, meestal door mensen die er niet al teveel van weten, die er vooral een ander probleem hebben. Dus een gewelddadig kind, een verslaafden kliniek waar te weinig mensen zich hebben ingeschreven. Er is wel een duidelijke transfer van in-game naar real life. En dat heeft te maken met dat ik in-game met bepaald gedrag kan experimenteren. Dat ik kan experimenteren met bepaald gedrag, en die strategieën mee neem naar real life. En in een gewelddadige game gaan mijn strategieën eigenlijk nooit over geweld. Maar over *pathfinding*, of over samenwerking, en die neem ik mee. Het schieten zelf niet. Als je kijkt naar alle defensie apparaten wereldwijd, die gebruiken allemaal games. Maar nooit om te leren schieren. Wel om te navigeren in stedelijke gebieden. En dat schieten doe je op de schietbaan, dat is namelijk gewoon iets anders. Het debat moet je dus altijd blijven voeren. Zolang mensen behoefte hebben aan dat debat, ook die mensen waarvan wij denken van niet. Deels vergroten we daarmee de ...[26:24-26:26 slecht te verstaan.. 'boekgangen?']... Businesswise is dat slim om te doen. En ook fatsoenlijk. Alleen het debat zou moeten gaan over: 'Welke dingen worden er nou meegenomen uit een game?'

JW: Welke conclusie kan je trekken? Daar gaat het om, uiteindelijk.

E: Je ziet dus dat kinderen hun gedrag naar aanleiding van een gewelddadige game wel degelijk gewelddadiger worden, omdat ze hun gedrag gaan moduleren naar wat ze net gedaan hebben. Dat gebeurt ook met films, dat gebeurt ook met boeken. Als ik mijn zoontje voorlees uit een boek over een ridder die draken verslaat, is hij die week erna een ridder met een draak. Dan hoort daar dus een zwaard bij. Maar dat wil niet zeggen dat hij daarmee ook...

JW: Geen boeken meer moet lezen.

E: Nee.

20. [27:42] Zijn games alleen een motivatie-tool?

E: Nee, het is een tool om mensen te leren om bepaald een gedrag, daarmee te experimenteren, waardoor ze niet meer alleen maar door een leraar favoriete gevonden strategie opgelegd krijgen, maar zelf kunnen uitvinden wat voor hun de beste strategie is.

JW: Een game is denk ik een heel goed middel om te leren leren. Om te leren wat jouw leerstrategie is. Wat jouw manier van waarheidsvinding is. Veel meer dan... Die motivatie, daar ben ik zelf nooit zo van gecharmeerd. Dat het leuk is om te spelen vind ik veel meer een by-product van een spel dan dat dat de main toegevoegde waarde is. Het gaat veel meer over je eigen leerstrategieën, directe feedback loops. Daar gaat een game over. Dat het ook leuk is... Zelf als een game niet leuk is kan hij nog steeds heel effectief zijn. Het effect zit hem niet in het feit dat het leuk is.

E: Leuk heeft ook weer te maken met implementatie, meer. Dus willen we dat mensen dat in hun vrije tijd gaan doen, dan moet hij dus leuker zijn dan een game die je in een klaslokaal doet. Sterker nog, we krijgen heel vaak van leerlingen van basis- en middelbare scholen die dus in de les een game moeten doen, als die te leuk is, krijgen we commentaar.

JW: Dat willen ze niet.

E: "Ik ben hier wél om te leren, kom op nou. We gaan hier niet doen alsof het pauze is de hele wiskundeles." Echt waar. Dat klopt. Leerlingen hebben dat heel goed in de gaten.

JW: Die hebben zoiets van: "Als ik een spelletje wil spelen ga ik wel naar huis. Ik ben hier gewoon om te leren."

E: En dat hebben mensen op hun werk ook.

JW: "Ja."

21. [29:22] Denk je dat mensen door games een grotere mediawijsheid krijgen?

JW: Mediawijsheid van wat?

[Dat mogen jullie invullen]

E: Mediawijsheid in het algemeen heeft veel meer te maken met: "Hoe interpreteer ik informatie die uit verschillende kanalen komen?" En een game is over het algemeen maar één kanaal. Die is dus heel erg binnen een perspectief gefocussed.

22. Dus er moet meer bijkomen, je kan niet iets leren alleen met een game?

E: Mediawijsheid heeft veel meer te maken met: "Ik lees iets op *Wikipedia*, ik lees iets op de site van *Harvard* en ik lees iets op *GeenStijl*. Hoe verhoudt zich dat tot elkaar? Want ze zijn het niet met elkaar eens." Dat is veel meer mediawijsheid. Maar binnen een game heb je zo'n eigen universum...

JW: Je zou wel een game over mediawijsheid kunnen maken.

E: Sterker nog, *Six to Start* heeft een prachtige game over mediawijsheid gemaakt. Maar die [mediawijsheid en games] zijn niet één op één te trekken.

23. [30:22] En als we het breder trekken? überhaupt in leren, moet je naast een game nog iets hebben?

JW: Ja ik denk het wel. Ik denk niet dat een game per sé het meest geschikte middel is voor directe kennis overdracht. Ik weet niet of jij dat met me eens bent [Evert]?

E: Ja dat ben ik helemaal met je eens.

[Waar is het wel geschikt voor?]

JW: Het leren leren. En je eigen leerstijl ontwikkelen, en je eigen strategieën. Het is een veel pro-actievere vorm van leren dan het traditionele leren. In de klas heb je een ontzettend passieve vorm van leren. Als je probeert een leraar te begrijpen dan wordt je er ook moe van. Het is heel erg *leaned-back*. Terwijl in een spel: als je niets doet, dan gebeurt er niets. Dus elke stap die je maakt moet je zelf maken. Het is een hele andere manier, een veel proactievere manier van leren. Maar als je het er nou over hebt: "Je moet kennis opdoen," laten we zeggen de geschiedenis. Als je het hebt over het leren van geschiedenis is het helemaal uitspelen van een geschiedenis game niet per sé heel effectief. Dan is misschien een docent die ontzettend goed kan vertellen wel interessant, maar als je het wil hebben over "Hoe was het krachtveld tussen Duitsland en Frankrijk tijdens de tweede wereldoorlog of daarvoor?" Dan is een spel wel een heel geschikt middel en dan kun je daarmee experimenteren.

E: Als je het hebt over uitleg in een game ...[31:44-31:46 praatten de heren door elkaar dus de eerste helft van deze zin is een interpretatie]..., dan betekent dat automatisch dat de einddatum van de tweede wereldoorlog niet vast ligt. Omdat ik het zelf kan beïnvloeden. En dan leer ik dus nooit wanneer de bevrijding was. Want dat is mijn mijn game misschien wel pas in 1952.

JW: Misschien komt er wel helemaal geen oorlog.

E: Of wint Luxemburg. Spreken we nu allemaal Luxemburgs.

JW: Enfin, een game is niet een leermiddel wat geschikt voor elke doelstelling, dat geloof ik helemaal niet.

24. [32:27] Wat zijn de belangrijkste elementen? We hebben het over korte feedbackloops gehad, over persoonlijke leerstijlen, wat zijn andere elementen die echt terug moeten komen in een serious game voor educatie?

E: Meerdere strategieën kunnen hebben.

JW: Wat ik zelf een hele fijne vind is dat het spel de speler direct in control bent. Dus of je nou een held als *character* bent of niet, maar als jij een spel speelt ben jij de bepalende factor. Dat kom je in het onderwijs ook nergens tegen. In het onderwijs word je altijd in de afhankelijke factor geplaatst.

[Dus je zit altijd in de subjectpositie?]

JW: Ja. En hier ben je gewoon, vanaf het moment dat je speelt, relevant. Dat vind ik een heel mooi aspect van spellen wat in het onderwijs ook wat meer zou mogen. Een ander mooi voorbeeld vind ik dat waar het onderwijs zich bijna alleen maar richt op eindresultaat; "Heb je een vijf of een negen gehaald?", gaan spellen veel meer over progressie. En het belonen van progressie, en die korte feedbackloops die mee gaan in jouw persoonlijke progressie, is iets waar ik vind dat spellen vele malen beter in zijn dan ons onderwijs.

E: Ik zie ook in de speler beleving: het meest teleurstellende moment in een game is als je hem hebt uitgespeeld. Want dan houdt alles op. Terwijl met het onderwijs werk je daar naartoe. Naar het terug krijgen van je proefwerk. Een gamer is dan alweer verder, die is dan al begonnen aan de volgende game.

JW: En wat daar jammer aan is, aan de huidige manier van onderwijs, is dat je helemaal niet geïnteresseerd bent in wat je leert, omdat je alleen maar geïnteresseerd bent in het behalen van het eindresultaat. En dat is eigenlijk een gemiste kans. Dus door zo ontzettend de focus te leggen op het eindresultaat is de weg ernaartoe niet meer interessant. Terwijl in een spel is de weg er naartoe, is het spel. Ik denk dat onderwijs daar nog heel veel kan leren van ons. Maar dan moet je dus wel het onderwijs helemaal opnieuw ontwerpen. Je kan niet zeggen: we doen een gamelaag maar je uiteindelijke afrekening is een acht of een negen. Want leerlingen zijn niet gek, die hebben dat heel snel in de gaten. Die denken: "Oh, dit geloof ik allemaal wel". En dat is wat Evert zegt, en ik denk dat dat ook echt zo is, ik denk dat het spelend leren, want het gaat veel meer over het spelen dan over de game, het spelend leren pas echt gaat werken als partijen tot op dat niveau de boel gaan herontwerpen.

E: En dan zie je dus ook dat in zo'n korte feedbackloop, waar die ontworpen is om positieve feedback te krijgen, als het negatief is kan je weer terug. Als hij positief is kan je weer door, maar een heleboel spelers doen dat niet. Een heleboel spelers krijgen positieve feedback, maar gaan toch terug. Om te kijken of er nog een andere manier is. Of om te kijken of hun hypothese over hoe je het nou helemaal mis kan laten gaan ook klopt.

JW: Ze gaan een hele andere strategie proberen. Gewoon *just for the sake of it*.

E: Zeg maar de drie sterren van *Angry Birds*, er zijn echt maar een paar mensen in een klas die als ze een 6,5 hebben gehaald op hun proefwerk vervolgens vragen: "Mag ik hem nog een keer doen? Want misschien weet ik een manier hoe je een zeven krijgt." Dat is met *Angry Birds* doet *iedereen* dat. Als je één spel hebt uitgespeeld ga je door, en na een tijdje kom je terug om hem met twee of drie sterren uit te spelen.

25. [35:53] Zijn er dingen die we nu niet besproken hebben die jullie wel belangrijk vinden?

JW: Voor jouw scriptie, als het gaat over games als leermiddel. Toch? [Ja].

E: Dat is nog heel breed. [Voor programmeren].

JW: Nou, als je het hebt over programmeren, heel specifiek, wat ik denk programmeren begint in eerste instantie met het begrijpen van de syntax. De logica van de taal. Dat zou wat mij betreft een ander type spel zijn, want je kunt van programmeren namelijk een spel maken. Je kunt de strategie, even heel flauw, stel je moet in een game een balletje over een schuttinkje krijgen, en jouw tool is code, dan kan dat een heel leuk spelletje zijn. Maar dan kun je dus verschillende strategieën toepassen omdat balletje er overheen te krijgen, alleen je moet ze zelf programmeren. En dat zou volgens mij best een leuk spel kunnen worden.

E: Ik denk dat één ding wat we nog helemaal niet besproken hebben is het toepassen van game design als onderwijs. Dus na het spelen van de game komt eigenlijk

onmiddellijk het modificeren van de game. Want je hebt dan het systeem in een soort van vertrapte werkelijkheid, dus via die metafoor waar hij in zit, leren begrijpen. En dan, om dat begrip nog verder te verdiepen, kan je dat systeem gaan modificeren. Dat begint heel klein, dus eigenlijk niet eens in de code maar in de xml. Dus alles waar ik nu '+1 speed' krijg, dat ik daar +5 van maak.

JW: Dan wordt de game onspeelbaar.

E: Totdat je helemaal terug bent in de bron van die game. En dat zie je ook bij *Quest to Learn*, een basisschool gebaseerd op game principes. Daar zijn eigenlijk alle exacte vakken vervangen door game design. Maar dan moet je altijd ook weer, om dat perspectief te pakken, mag het niet over programmeren gaan maar over het maken van een game, en daarvppr heb je heel veel programmeer kennis nodig.

26: [38:10] Dus die twee concepten zijn wel goed aan elkaar gelinkt?

JW: Ik denk het wel.

27: [38:16] Bijvoorbeeld: zijn jullie bekend met *Code Academy***?**

E: Ja. Bij *Code Academy*, zoals ik het ken, dus alleen de buitenkant, ben je wel de leerling. En dat betekend dat op het moment dat ik verder wil, en ik snap het nog niet, dat ik ga afkijken. Want dat hoort bij mijn perspectief van de leerling.

JW: Ik denk dat het een goed eerste begin is.

E: Ik vind wel dat bij *Code Academy* de skill tree heel duidelijk is. Dus ik weet als ik dit doe, als ik dit doe en als ik dit doe, dan kom ik daar, maar als ik dit niet begrijp kan ik wel terug en dan zo, zo zo, ben ik er ook. En dat geeft dus die strategieën heel inzichtelijk om ergens te komen.