PROCESS ART

Het software-proces als leidraad in de nieuwe-mediakunst

Arjan Scherpenisse

Afstudeerscriptie Gerrit Rietveld Academie, DOGTIME avondopleiding,
afdeling *Unstable Media / Interaction Design*, Juli 2009.

PROCESS ART

Het software-proces als leidraad in de nieuwe-mediakunst

Arjan Scherpenisse

Afstudeerscriptie Gerrit Rietveld Academie, DOGTIME avondopleiding,
afdeling *Unstable Media / Interaction Design*. Amsterdam, juli 2009.

Inhoudsopgave

A

Inleiding

A.1 Motivatie

Een deel van mijn motivatie voor deze scriptie, en ook voor mijn eindwerk, is gestoeld op een fundamentele twijfel. Ik heb lang niet geweten of ik mezelf wilde ontwikkelen tot een fulltime, “standalone” kunstenaar: ik dacht me prettiger te voelen bij een aanpak die meer “facilitair” was: samenwerkend met anderen, in collaboratieven, aan gezamelijke projecten. Niet alleen gedreven vanuit een artistiek idealisme maar ook vanuit een economische noodzaak.

Waar ik eerst dacht dat deze twijfel een persoonlijke zaak was, ben ik naarmate mijn research vorderde tot de conclusie gekomen dat deze twijfel een fundamenteel deel uitmaakt van de hedendaagse kunstwereld. Realistisch gezien is het voor een kunstenaar ondoenlijk om zelfstandig actief te zijn. Vooral in de nieuwe mediakunst wordt de schoonheid van kunst in grote mate bepaald door haar technische implementatie, en de benodigde kennis en expertise die hiervoor noodzakelijk is, is zó omvangrijk dat het niet meer door één persoon kan worden gedaan. Dit fenomeen stelt de auteurschap van een werk in een nieuw licht. In de klassieke hedendaagse kunst opereren kunstenaars nog wel onder hun eigen naam, maar worden ze ondersteund door een heel team aan technici en marketeers. Een meer open aanpak zijn de collaboratieven waarbij de namen van kunstenaars zélf er niet meer toe doen.

Deze overwegingen en twijfels brachten me terug bij mijn *roots*: waar is het allemaal voor mij begonnen? In wezen ben ik een geboren en getogen computerwetenschapper: dat is mijn vak. Een software-ontwikkelaar met een hang naar perfectionisme, maar met nog nét genoeg pragmatisme om effectief te kunnen werken. Kortom: een ambachtsman. Maar ik heb na mijn eerste studie wel voor de kunstacademie gekozen: was dit een poging tot ontsnapping? Een kreet diep van binnenuit? Er moest duidelijk nog iets boven komen borrelen: de ambachtelijkheid alléén was niet genoeg.

In deze scriptie kies ik ervoor mijn ambacht centraal te stellen: software. En wel software in de context waarin ik me het meest thuisvoel: de kunstwereld, met al haar idealen, creatieve uitspattingen, maar ook dieptepunten. Software fascineert me: codes, losgelaten op de processor. Zitten er bugs in? Zal het werken zoals het is bedoeld? Hebben de instructies het gewenste effect?

Software en de processen eromheen hebben een steeds groter wordende invloed op hoe de hedendaagse maatschappij functioneert, en dus ook op de kunstwereld. Ik voel de behoefte om software en kunst op één lijn te krijgen. Middels deze scriptie wil ik een overzicht scheppen dat mij kan helpen mijn positie op het snijvlak van deze velden de komende jaren te bepalen.

A.2 Doel van deze scriptie

De almaar toenemende belangstelling voor de zogenoemde “nieuwe-media-kunst” noopt tot een kritische beschouwing van de onderliggende radarwerken die deze nieuwe vormen van kunst draaiende houden: de software. In de kunstwereld is de achterliggende software vaak een onderbelicht aspect van een werk. Dit is niet meer dan logisch: software is van nature een medium dat niet snel aan de oppervlakte komt: software zélf is onzichtbaar: alleen de effecten van software kunnen worden waargenomen.

Het is tijd deze impasse doorbreken door de software in het middelpunt te stellen: als spil in het radar van elk werk dat zich onder nieuwe-mediakunst schaart. In deze scriptie verdedig ik daarom een stelling die deze positie van software rechtvaardigt. De stelling is de volgende:

> ***Een werk dat software tentoonstelt of gebruikt als controllerend proces heeft geen definieve vorm.***

Om dit standpunt te begrijpen, zullen we software niet als een statisch gegeven, als "pure tekst" moeten zien, maar moeten we het zien als een continu, altijd in beweging zijnde entiteit: het *software-proces*. Het werk dat software gebruikt neemt daardoor vanzelf ook deze proces-vorm aan. In deze scriptie kies ik drie benaderingen hoe de term "software-proces" kan worden geinterpreteerd, om als bouwstenen te dienen ter ondersteuning van de stelling zoals ik die heb geformuleerd.

Volgens goed software-gebruik begint de scriptie bij hoofdstuk nul. Dit omdat in software begint men altijd begint te tellen bij nummer 0, en niet bij nummer 1 zoals in de rest van de wereld. Ik begin ook bij hoofdstuk 0 omdat dit hoofdstuk er eigenlijk niet bijhoort: het is niet, zoals de andere hoofstukken een verklaring van het begrip "software-proces", maar het dient meer als schepper van de context.

Hoofdstuk 0 dus, de context: De kunstwereld, nieuwe-mediakunst, software-cultures. Dit hoofdstuk schetst een beeld van de hedendaagse kunstwereld vanuit het "new media art" perspectief. Wat is er "new" aan "media art"? En wat wordt er eigenlijk bedoeld met "media"? Het eerste hoofdstuk licht dit proces toe en introduceert ook "software-studies": de kritische benadering van software. Want bestáát software eigenlijk wel?

Het eerste échte hoofdstuk gaat over software als expressief proces: het behandelt de zelf-reflecterende natuur van software (recursie, zelfreplicatie), en vanuit een poetisch standpunt, zoals de expressieve kracht van de verschillende programmeertalen. Maar ook de schoonheid van software-architectuur. En: wordt software zélf eigenlijk al als kunst beschouwd? Ik doe dit aan de hand van een aantal werken en kunstvormen die het gebruik van software in een centrale positie stellen.

Hoofdstuk twee gaat in op het collaboratieve proces: wat brengt software teweeg, hoe komt het tot stand? Het behandelt de geschiedenis van open source en de auteurschapskwesties die bij kunst zich aandienen. Dit gebeurt ook vanuit een sociaal-economische hoek, om de deze kwesties in een grotere context te plaatsen.

In het derde hoofdstuk wordt het software proces vanuit het oogpunt van software constructie behandeld. Hoe wordt software gebouwd, wat zijn de verschillende methodologieën en strategieën die eraan ten grondslag liggen, en hoe worden deze gebruikt in de wereld van de (ontwikkeling van) nieuwe-mediakunst. Kortom: wat zijn de implicaties van het gebruiken van software als "materiaal".

Ik eindig met een conclusie waarin ik de bevindingen van deze drie hoofdstukken samensmelt om de centrale stelling van deze scriptie te rechtvaardigen.

Hoofdstuk 0

Nieuwe-media *meets* software

Nieuwe-mediakunst. Het is dezer dagen een gebruikelijke term om alle soorten van kunst te omschrijven die niet binnen een traditionele kunstvorm te classificeren valt. Vaak is dit soort kunst interactief, en in vrijwel alle gevallen wordt er gebruik gemaakt van nieuwe technologieën zoals de computer, internet en mobiele apparaten. Het is, zoals Geert Lovink [1] het formuleert, een "transnationale kunstvorm, een multi-disciplinaire 'wolk' van micro-praktijken".

Nieuwe-mediakunst is hip: het explodeert. De kunstwereld is er door aan het veranderen. In Nederland is dit dit bijvoorbeeld zichtbaar in de koersverandering van grote instituten: Het Nederlands Instituut voor de Mediakunst exposeert tegenwoordig vrijwel nog alleen maar interactieve, "nieuwe-media" werken, terwijl het uit een traditie van de videokunst is ontstaan. Pure videokunst is er nauwelijks meer te vinden. En andere, traditionelere instellingen zoals het Boymans van Beuningen durven het aan om puur videokunst te laten zien, zoals nu in de expositie van Pippiloti Rist (tot mei 2009).

De kunstvorm is eigenlijk simultaan gegroeid met de ontwikkeling van de computer en microchips. De mogelijkheid om een machine te laten doen wat je vertelt hem te doen is een krachtig principe dat vrijwel onmiddelijk niet meer alleen gebruikt werd voor het industriele / functionele doel waarvoor het was uitgevonden. Al in 1970 werd er een expositie georganiseerd getiteld *Software* [2]. In deze expositie was onder andere werk te vinden van Ted Nelson: een catalogus gebaseerd op het hypertext-principe (de eerste publieke uiting van hypertext!). Ook was er een installatie van Nicholas Negroponte (die later MIT's Media Lab zou oprichten) die met het werk *Seek* een robot tentoonstelde die een interactie aanging met een hamsterkolonie. De robot werd een soort "hand of god" die de wereld van de hamsters organiseerde. De gehele expositie was opgebouwd rond dit soort interactieve, technologie-gecentreerde werken. In de catalogus van de expositie schreef Jack Burnham al dat "... computers misschien wel primair niet bedoeld waren om kunst mee te maken, maar dat ze wel een sleutelrol zullen gaan spelen in het herdefiniëren ervan". Een visionair statement: nieuwe-mediakunst heeft inmiddels een geheel eigen positie veroverd in het kunstlandschap.

Maar hoe valt deze nieuwe kunstvorm te classificeren? Wat zijn deze "nieuwe media" nou eigenlijk? In het artikel uit 2005 heeft Lovink[1] heeft het over een "verzameling van micro-praktijken". Een "wolk" van kleine, zeer specialistische kunstuitingen, die soms overlappen en altijd aan verandering onderhevig zijn. Één factor die al deze praktijken in ieder geval gemeen hebben, is dat ze nieuwe technologieën verkennen met *als doel* het vinden van nieuwe manieren van artistieke expressie. De nadruk ligt dus op het verkennende aspect: zoals Lovink aangeeft is het niet het doel van deze stroming om tijdloze, universele kunstwerken te creëren, maar is het doel meer om een universele taal te vinden waarin toekomstige generaties zich kunnen uitdrukken. Hierin sluit hij zich aan bij Lev Manovich, die in het standaardwerk "The Language of New Media" [3] de eerste theorie over "Nieuwe Media" opstelt, door het te plaatsen in de lijn der geschiedenis van de visuele media zoals fotografie en film. Hij laat zien dat oude concepten zoals het kader van een camera nog steeds van toepassing kunnen zijn op nieuwe-media, en op nieuwe concepten in die media zoals databases. Manovich benadert de nieuwe media vanuit het standpunt dat ze hun oorsprong hebben in de "klassieke", visuele media: voornamelijk fotografie en film. Manovich' benadering gaat relaties aan met de geschiedenis van deze visuele media en met nieuwe ontwikkelingen hierin, maar het is geschreven vanuit het gelimiteerd perspectief. De nadruk blijft altijd liggen op voorbeelden rond cinema, virtual reality en games.

Alhoewel er wel wordt ingegaan op de ontwikkeling van computerprogrammatuur waar steeds meer

mee mogelijk is door een toenemende abstractie van de onderliggende processen, noemt Manovich software als fenomeen eigenlijk maar sporadisch. In het boek missen referenties naar de experimentele literatuur, naar de procesmatige performances uit de jaren '60, op fluxus. Ik denk dat dezen een niet te onderschatten invloed hebben op de nieuwe-media zoals we die vandaag de dag kennen. Ík zou zelfs durven stellen: *Software ís de taal van de nieuwe-media*.

Nu is Manovich' werk natuurlijk ook al weer redelijk gedateerd; in 2001 was de hele "software-revolutie" nog niet zo emergent als nu. De laatste jaren is software een meer prominenter plek in onze cultuur gaan innemen: alle electronica van tegenwoordig bevat op de een of andere manier software, al is het een wekker-radio, een wasmachine of een mobiele telefoon. Ook het publieke bewustzijn van de juridische implicaties van het gebruik van software wordt steeds groter: het gebruik van open-source alternatieven van gangbare software-pakketen, van alternatieven voor Microsoft Office tot complete besturingssystemen zoals Ubuntu neemt toe. En ook in de kunstwereld is dit merkbaar: het aanbod van "open" software vóór en dóór kunstenaars neemt toe.

Op de golf van deze revolutie is er recentelijk een nieuw onderzoeksveld ontstaan, *Software studies*, dat, op een breder vlak dan kunst alleen, deze erkenning dat de samenleving tegenwoordig voor een belangrijk deel wordt vormgegeven door software deelt. Door de belangrijke rol die het speelt moet software daarom onderhevig moet zijn aan kritiek en introspectie. Het gelijknamige boek dat deze stroming startte, samengesteld door Matthew Fuller [4], is een verzameling van artikelen die vanuit meerdere standpunten deze kritische blik probeert vorm te geven.

Friedrich Kittler, die ook een artikel heeft in *Software Studies*, is al langere tijd bezig met dit onderwerp. In 1995 schreef hij *There Is No Software* [5], waarin hij begint met het aankondigen van het eind van het fysieke schrijven: het geschrevene bestaat niet meer in een ruimte-tijd, maar alleen nog maar in de voltage-verschillen in het binnenste van een machine. Vervolgens bespreekt hij hoe de toenemende maten van abstractie die in een computer plaatsvinden, de feitelijke software helemaal verbergen en zelfs teniet doen. Deze argumenten maken het aannemelijk dat software zo lang verborgen is geweest in de kunst: als *materiaal*, als basis van een werk, is het niet zichtbaar. Software is blijkbaar zó onzichtbaar dat ook Geert Lovink in zijn artikel [1] er niet lang stil bij staat. Kort noemt hij de tendens van kunstenaars om zich te verzetten tegen bestaande (gesloten) systemen en de activistische hackersmentaliteit van kunstenaars, maar hij gaat niet in op de daad van het schrijven van software als artistieke basis van een werk.

Onder "nieuwe" media worden vaak ook de *interactieve* media geschaard. Met de komst van de computer is natuurlijk niet interactie als begrip uitgevonden, maar het heeft wel een hele nieuwe dimensie gekregen: de computer is bij uitstek een apparaat waarmee een interactie wordt aangegaan. Met de komst van de computer heeft interactie ook een prominenter plaats ingenomen in de kunstwereld: buiten performances en interventies die natuurlijk al bij uitstek een interactief medium waren, werden ook "autonome" werken in toenemende mate interactiever. Is een menselijke partij noodzakelijk voor een interactief werk? Andy Lippman vindt van niet: hij definieert interactie heel algemeen, als "mutual and simultaneous activity on the part of both participants usually working toward some goal, but not necessarily"[6]. Dit in tegenstelling to Manovich, die interactie echt ziet als een proces waar altijd een mens bij betrokken is.

Het concept interactiviteit met al zijn connotaties is zo breed dat het buiten de scope van deze scriptie valt. Deze scriptie gaat over software als opzichzelfstaand proces, niet over de interactie van de mens mét software. Daarom wil ik het begrip interactie in deze scriptie verder buiten beschouwing houden. Zoals Manovich het verwoord:

> *"Once an object is represented in a computer, it automatically becomes interactive. Therefore, to call computer media "interactive" is meaningless - it simple means stating the most basic fact about computers."*

Terug naar software. Alsof hij weet dat hij in *The Language of New media* uit 2001 iets vergeten is, stelt Manovich in zijn laatste boek, *Software takes command* [7], software wél helemaal centraal als middelpunt van de new media beweging. In het boek bespreekt hij, aan de hand van de geschiedenis van de personal computer, deze computer als manipulator voor media. Alan Kay speelt een centrale rol in het eerste hoofdstuk: hij was de eerste die de computer beschreef als een *meta-medium*: een medium waarmee andere media gecreëerd en gecombineerd kunnen worden. Het is niet alleen een gereedschap, maar het kan zich gedragen zoals gereedschappen. In dit licht heeft hij het over de

dynamische natuur van software: het is niet statisch, het kan altijd worden uitgebreid en verbeterd. In dat licht komt hij ook uit op waarom nieuwe-media eigenlijk de classificatie “nieuw” verdient:

> *“New media is “new” because new properties (i.e., new software techniques), can always be easily added to it.”*

Software is een *enabler* van deze nieuwe-media. De dynamische natuur van software, de veranderlijkheid van software door de tijd heen, maakt het mogelijk dat nieuwe-media zich blijven ontwikkelen. De komende drie hoofdstukken wil ik gebruiken om in te gaan op de precieze natuur van deze dynamiek: wat zijn de speciale eigenschappen die software tot deze katalysator maken?

Hoofdstuk 1

Software als expressief proces

Het is moeilijk je een voorstelling te maken van wat software nu precies is. Als je een persoon vraagt wat hij zich voorstelt bij software, dan krijg je hoogst waarschijnlijk een antwoord terug als “codes in de computer”, “enen en nullen”, of iets dergelijks. En inderdaad, het beschouwen van software als tekst is inderdaad het meest voor de hand liggende mentale model. Op een bepaald niveau is software ook niet meer dan tekst, het is de “view source” knop in je webbrowser, de rauwe HTML (al is dat strikt gezien geen software, maar opmaak), de binnenkant van de computer; de achterkant van de vensters op je beeldscherm. Tekst is de primaire manier van het schrijven van software.

Wanneer je software schrijft, doe je dit altijd in een bepaalde taal. De taal dicteert de syntax waarin je schrijft: de vereiste volgorde van symbolen. Een fout in de syntax (meestal een van de eerste meldingen die een beginnend programmeur te zien krijgt) resulteert in een *syntax error* en weerhoudt het programma van starten. Er zijn letterlijk duizenden programmeertalen. De populairsten (C, Java) zijn *general purpose* talen en worden door miljoenen programmeurs gebruikt; aan de andere kant van het spectrum zijn er de talen die geschreven zijn voor een specifiek doel, en die maar door enkelen wordt beheerst. Een klasse apart zijn de zgn. *Weird Languages* [8], programmeertalen die zijn geschreven met als doel het spelen met en verkennen van de mogelijkheden die bij het ontwerp van zo’n taal komen kijken: *Shakespeare*, een programmeertaal waarin alle codes moeten worden opgesteld alsof je een toneelstuk aan het schrijven bent en *Malbolge*,een taal die ontworpen is met als doel dat het vrijwel onmogelijk is om er een programma mee te schrijven. programmeren te zijn.

1 .1 Software als performance

Een trend die de laatste jaren sterker wordt, is het zogenaamde *live coding*. Live coding is een vorm van performancekunst waarin de kunstenaar software schrijft of herschrijft als performance voor een publiek. Het resultaat van de code-manipulaties is direct waarneembaar en wordt vaak gebeamd zodat het publiek kan meekijken met de performer. Een van de eersten die dit deed en die erover schreef was Alex McLean [9], die de taal Perl gebruikte om tijdens performances in nachtclubs live muziek te componeren door te programmeren.

Het publiek aanschouwt de performer, ziet hem “live” algorithmes bedenken en opschrijven, zichzelf verbeterend, en tegelijkertijd is het resultaat van die code te zien als visualisatie, en/of te horen als muziek. Het weergeven van dit “artistieke” proces is het centrale punt van live coding: McLean vergelijkt het proces van schrijven van software met dansen: het opbouwen en afbreken van abstracte patronen gedurende de tijd, telkens veranderend, en uiteindelijk een soort conclusie bereikend. Het coderen als een “flow”, een proces van creatie, met een enorme aantrekkingskracht.

Naast Perl zijn er nog meer talen geschikt voor live coding. Sommige talen zijn er zelfs speciaal voor ingericht: ze bevatten dan vaak een speciale editor waarin de code wordt geschreven. Fluxus is één van deze omgevingen: gebaseerd op Scheme, een LISP-variant, is het een omgeving waarin een 3D-game-engine live kan worden gemanipuleerd. Een andere interessante is ChucK, een live coding taal voor muziek. Deze twee talen zijn enkelen van de tools die gebruikt wordt door het TOPLAP collectief [10] - een community van kunstenaars die allen live coding gebruiken in hun performances. In het TOPLAP manifest wordt treffend beschreven wat hun instelling is.

> *We recognise continuums of interaction and profundity, but prefer: Insight into algorithms The skillful extemporisation of algorithm as an expressive/impressive display of mental dexterity.*

Dit is een behoorlijk *geeky* uitspraak die echter wel typerend is: er wordt van de kunstenaar verwacht dat hij een degelijke programmeur is en zijn instrument onder de knie heeft.

> *Live coding is not about tools. Algorithms are thoughts. Chainsaws are tools. That's why algorithms are sometimes harder to notice than chainsaws.*

Deze stelling duidt weer op de "onzichtbaarheid" van software: het zit voornamelijk tussen de oren, en is niet zichtbaar. Het maakt niet uit welke tool er gebruikt wordt, zolang het algoritme tot zijn recht komt.

> *It is not necessary for a lay audience to understand the code to appreciate it, much as it is not necessary to know how to play guitar in order to appreciate watching a guitar performance. Live coding may be accompanied by an impressive display of manual dexterity and the glorification of the typing interface.*

Met het eerste deel van deze stelling ben ik het niet helemaal eens. De vergelijking tussen een gitaarspeler en een softwaremaker is moeilijk te trekken: de moeilijkheid van gitaarspel kan wel worden afgeleid aan de snelheid van de bewegende vingers, maar aangezien programmeren voornamelijk tussen de oren gebeurt, is het voor een publiek moeilijk om te waarderen dat iemand hard aan het nadenken is. Hier komt deel twee van deze stelling naar boven: het visuele element van de performance verdient dus wel degelijk aandacht: onder *manual dexterity* interpreteer ik als het publiek showen hoe goed je met je text-editor overweg kan.

De reden waarom het fascinerend kan zijn om een live coding performance te zien, beschrijft Inke Arns treffend [11] als een *speech act*: terwijl de code wordt uitgesproken (in dit geval, ingetoetst), wordt het tegelijkertijd uitgevoerd. De consequenties van het geschrevene zijn direct zicht- of hoorbaar: het geschrevene (*speech*) valt samen met de uitvoering, de *act*. Mochten de geschreven software-instructies geen zichtbaar resultaat opleveren, dan "faalt" als het ware de uitvoering ervan. Op het eerste gezicht zijn dit soort software-instructies als het ware nutteloos: waarom zouden ze worden opgeschreven, als ze geen effect hebben?

Binnen live coding lijkt dit het geval te zijn: het gaat erom een band te creëren met het publiek, het publiek te laten merken wat de code doet: zodat een relatie wordt gemaakt tussen de code en de visuals.

1 .2 Reflectie op het medium

Binnen live coding wordt de software letterlijk beschouwd als instrument, en de software die geschreven is dus altijd "nuttig": het is gemaakt met als doel de uitvoering ervan. Dit is niet altijd het geval, er bestaat ook "nutteloze" software: de software óm de software. Er zijn kunstenaars die het schrijven van software waarvan het nut niet direct duidelijk is tot kunstvorm hebben verheven. In de *Codeworks* van JODI komt dit naar voren. De werken bestaan uit e-mails die gestuurd worden naar public mailing lists. De e-mails bevatten uitsluitend schijnbaar nutteloze software-code. Voor een lezer die niet op de hoogte is van de gebruikte programmeertaal (als het al een bestaande taal is), is het niet op te maken of de code wel of niet uitvoerbaar is. In werken zoals *Walkmonster_start* (dat Arns ook aanhaalt) gaat het erom dat de code *in potentie* kan worden uitgevoerd, en dus een speech act wordt. Syntax en semantiek zijn zodanig verweven dat (in ieder geval de geschoolde) lezers van de code zich een potentie aan mogelijke uitvoeringen van de code gaan voorstellen.

Schijnbaar nutteloze stukken software kunnen toch nut hebben, maar in een andere context: in de hoofden van de (menselijke) lezers ervan creëert deze software een geheel eigen belevingswereld. Hoe zou het "walkmonster" eruit zien?

Een stroming gerelateerd hieraan is de zogenaamde *net.art*: een kunstgenre waarvan de kunst uitsluitend op het internet bestaat. Vaak is het onderwerp van de kunst ook direct gerelateerd aan het medium: JODI's website `wwwwwwwww.jodi.org` is een schijnbaar eindeloze reis door HTML codes,

software glitches, CGI scripts, perl code, computer games, enzovoorts. JODI's werk laat de ingewanden van het web zien: het keert het web binnenstebuiten. Deze stortvloed van beelden blijven echter illustraties: het wordt niet duidelijk wat er bedoeld wordt; de code is (vaak) niet uitvoerbaar of ongeldig. Het doet zijn werk op een associatief niveau: JODI gebruikt software hier als illustraties: eindeloze variaties van codes en artefacten die een computercultuur representeren die de mens heeft geschapen.

Sommige *codework*s spelen daarentegen meer met de dubbele betekenis van software codes, en gaan daarmee in op de algoritmische kant van code: het feit dat code een proces beschrijft, een reeks instructies met beslismomenten, herhalingen en sprongen. `barszcz.c` [12] bijvoorbeeld, geschreven door Denis "Jaromil" Rojo. Het is een computerprogramma geschreven in C, dat tevens als recept gelezen kan worden. Hierin is dit werk is gestoeld op principe van *double coding*: het is zowel voor een computer begrijpbaar (dwz, syntactisch correct), als dat het voor een mens leesbaar is: met een beetje moeite kan er daarwerkelijk een soep mee worden gekookt. Dit werk was de eerste bijdrage aan het *barszcz* project: een poging om zoveel mogelijk verschillende broncodes (recepten) te verzamelen die het proces van het maken van de bietensoep borsjt beschrijven.

Door dit double-coding principe hóef je als lezer van de code dus niet noodzakelijk te begrijpen wat er algoritmisch gesproken gebeurt: door de keuze van variabelenamen en volgorde van dingen, alsmede het commentaar dat in de programmacode geschreven is, kan er toch op dit tweede niveau een betekenis gegeven worden.

1 .3 Generatieve software

Voor mij worden kunstzinnige uitingen die refereren aan software dus pas interessant als het geldige syntax heeft: het moet de mogelijkheid bevatten dat het uitvoerbaar is. Dit uitvoeren hoeft niet perse door een computer worden gedaan, zoals het recept van borsjt hierboven. `.walk` [13] van Wilfried Houjebek is een ander, eerder voorbeeld van een stuk code, een set instructies die door een mens uitgevoerd dienen te worden: de code dicteert de manier waarop een persoon door een stad kan dwalen: 1^{e}straat links, 2^{e}straat rechts, 2^{e}straat links; repeat. Zoals Cramer [14] observeert, grijpt dit werk terug op de fluxus-beweging van de jaren '60. Destijds werden algoritmische muziekstukken geschreven met als anticipatie dat ooit een computer het misschien zou kunnen uitvoeren. Het verschil met `.walk` is dat dit werk zichzelf (door de titel en de 40 jaar verschil) in deze computercultuur plaatst: de terugblik op fluxus is slechts ironisch.

De kracht van software is dat het een processuele kant heeft: het kan iets in gang zetten. Software beschrijft gedrag, het schrijft de grenzen voor waarbinnen gedrag zich kan afspelen. Zoals in `.walk`, dat een wandeling door een stad beschrijft. Deze wandeling is slechts afhankelijk van de startlocatie: ongeacht het tijdstip of het weer, elke keer zal de wandeling hetzelfde zijn wanneer er vanaf dezelfde locatie wordt begonnen. Mocht `.walk` dus door iemand worden uitgevoerd, dan kan het resultaat ervan worden beschouwd als *generatieve art*. Arns[15] beschouwt het als volgt: generatieve kunst refereert naar processen die autonoom of zelf-organiserend hun werk doen, volgens instructies die de kunstenaar heeft voorgeprogrammeerd. De kern van het gebruiken van generatieve processen in een werk is om elke vorm van intentionaliteit te vermijden: de "uitvoer" van een werk is gegenereerd, elke keer anders, dus kan een auteur niet worden verweten het werk er op een bepaalde manier te hebben vormgegeven.

Dat is natuurlijk de theorie, in de praktijk ligt het moeilijker want een auteur zet nog altijd de grenzen waarbinnen het werk zich kan afspelen. Een werk dat generatieve kunst maakt in het platte vlak zal nooit ineens 3-d objecten genereren: dit is nog steeds te danken aan de intentionaliteit van de auteur.

Susanne Jaschko merkt in een artikel over generatieve kunst op [16], dat de schoonheid van de achterliggende code buiten beschouwing blijft en irrelevant is voor de perceptie van de aanschouwer. De focus van deze stroming ligt altijd op de esthetische "front end", op het resultaat van het generatieve proces.

Er bestaat een klasse van software die hierin een stap verder lijkt te gaan: genetische algoritmes. Deze zijn een principe uit de kunstmatige intelligentie die programmacode zélf als variabel, generatief gegeven neemt. Programmacode wordt de uitvoer van een generatief proces.

Volgens een biologische metafoor wordt de programmacode (software dus) beschouwd als een DNA-reeks van enen en nullen. Een complete colonie van dna-reeksen wordt dan met elkaar gecombineerd

en gemuteerd, waardoor nieuwe software ontstaat uit de oude. Deze algoritmes worden zo generaties lang "geëvolueerd" om een optimale oplossing te vinden voor een gesteld probleem. Voor wélk probleem precies een oplossing wordt gezocht wordt bepaald door de zgn. "fitness-function" die de criteria waaraan een ideale kandidaat moet voldoen, beschrijft. En hier zit hem de crux: deze in fitness-function zit weer de intentionaliteit van de auteur van het algoritme.

Ik zou wel willen stellen dat het genereren van (mogelijk zelf-generatieve) software een stap verder is binnen de generatieve kunst, omdat er een extra abstractie tussen het werk en de auteur wordt geplaatst. Ik denk dat in genetische algoritmes, en speciaal gecombineerd met het beschouwen van software als expressief proces, voor mij nog een groot onderzoeksveld ligt.

Binnen deze scriptie wil ik het onderwerp van "generatieve kunst" verder buiten beschouwing laten omdat generatieve kunst software "slechts" beschouwt als een middel om een doel te bereiken, namelijk een meer of minder "onvoorzien" resultaat [15]. Deze scriptie behandelt software juist als centraal gegeven, compleet met alle connotaties en referenties die het met zich meedraagt.

1 .4 Een softwareproject als kunstwerk

Een fascinerend werk dat bij mij blijft hangen is het project *Time Based Text*[17] (TBT): een samenwerking tussen Jaromil en JODI. Opzich is de samenwerking al interessant: Jaromil is een (opensource) programmeur in hart en ziel, JODI houden zich bezig met de deconstructie van software: het afbreken en blootleggen ervan. Het concept van dit werk komt van JODI, die een *keylogger* wilde maken. Jaromil verbond dat daarna met de tijdgebaseerde natuur van het schrijfproces.

Time Based Text is een gereedschap voor het opnemen en terugspelen van het proces van het typen van een bericht, op de milliseconde nauwkeurig. Het idee erachter is dat in het proces van het typen zélf al betekenis verscholen zit: door de timings zichtbaar te maken, plus de spelfouten, wordt het duidelijk waar een persoon twijfelt, waar hij juist zeker van zijn zaak is, etc. De timing-informatie (die normaal gesproken in een "af" stuk tekst ontbreekt) is dus een belangrijk vehikel voor het communiceren van emotie. In theorie kunnen e-mail, blogs, elk digitaal medium, allen een "menselijke touch" worden gegeven door TBT te gebruiken.

In een interview wordt Jaromil gevraagd aan welk onderdeel in dit project hij nu de meest artistieke waarde hecht: aan het concept, of aan de uitvoering ervan in de software die hij heeft ontwikkeld. Hijzelf vindt het concept "artistieker" dan de uitvoering ervan. De opzet van de software en de potentieele resultaten die de poezie kan geven kunnen ook een artistieke waarde worden toegedicht, maar meer op een formeel niveau.

Ikzelf vindt ook de opzet van het project interessant. De software is gestructureerd opgezet: het is cross-platform en echt een stuk gereedschap in de UNIX traditie: simpel, minimalistisch, geschikt voor 1 taak. Én het is open source: het werk, met de GPL als licentie, dient als een referentie-implementatie en Jaromil hoopt dat zijn werk zal worden overgenomen in bv. e-mail clients en blogging software. De implicaties van een werk open source maken komen terug in hoofdstuk 2 .

Wat me ook interessant lijkt is het toepassen van het TBT principe op het schrijven van software toe te passen. Het schrijven van software is een *expressief proces*, en de toevoeging van de tijdsdimensie aan de code maakt het interessant voor een toeschouwer om de "flow" van een programmeur te begrijpen. Zoals live coding een toeschouwer ook in contact brengt met de coder, maar dan niet noodzakelijk *live*.

1 .5 Software art?

De bovenstaande werken en kunstvormen zijn allemaal nieuwe uitingen die onder verschillende termen kunnen worden geklassificeerd, van "new media art" tot "net art". Maar ik geef de voorkeur eraan om de term "software art" te gebruiken.

Bestaat "software art" wel? Verdient software wel het achtervoegsel "art"? Florian Cramer [14] schrijft hierover dat het naïef is om software puur als een constructief te beschouwen: het is niet alleen maar een materiaal waarop een kunstwerk gebouwd wordt. Veel programmeurs vinden de vraag zelfs niet relevant: het gaat niet zozeer om de vraag of iets "software kunst is", maar meer om of iets überhaupt "kunst" is. "Software art" is in feite een generieke term voor het soort kunst waarbij software centraal

staat, net zoals dat bij “video art” het geval is voor bewegend beeld. De term “software art” wordt vooral gebezigd door curators en critici, niet door de kunstenaars zelf. De vraag of software wel kunst is wordt is gerechtvaardigd doordat er genoeg bewonderenswaardige kunst wordt geproduceerd waarbij software centraal staat.

Software is een expressief middel dat in zijn pure vorm gebruikt kan worden als communicatief medium, al dan niet met een toegevoegde tijdsdimensie (live coding, TBT) die het gevoel van expressiviteit kan versterken. Hierdoor is software art gerechtvaardigd als op zichzelf staande kunstvorm. Software zou vaker een meer zichtbaardere rol in werken moeten innemen, om de notie van importantie van software bij een aanschouwer over te dragen. Voor de waardering van code is het niet noodzakelijk om die ook daadwerkelijk te doorgronden. Wanneer software in een werk zichtbaar is, toont het altijd de belofte, de mogelijkheid van een executie, en refereert zo aan de procesmatige eigenschappen van de code. Deze belofte van executie maakt de mogelijke uitkomst van het proces subjectief aan de beschouwer en daardoor zonder definitieve vorm.

Alhoewel ze beiden software centraal stellen, verschilt software art van generative art in de zin dat in generatieve werken de nadruk ligt op de uitvoering, op het resultaat: de software wordt beschouwd als een neutraal middel, als een gereedschap. Hoewel het interessante kanten heeft, worden de politieke, estetische en culturele subtexten van software niet ter sprake gesteld.

Hoofdstuk 2

Software als open, collaboratief proces

In dit hoofdstuk ga ik in op hoe het openstellen, het vrijgeven van software een natuurlijke collaboratie tussen auteurs tot stand brengt. opensource software. Het begrip zit bij mij al zo ingebakken, het is zo vertrouwd, dat ik me geen wereld kan voorstellen zónder. Het fenomeen heeft zich in de afgelopen decennia al vaak bewezen en vormt een geduchte concurrent voor de zng. *proprietary* software. Opensource en de cultuur eromheen hebben zich een belangrijke plaats verworven in de wereld van "computing".

Maar misschien is het goed om eerst een hypotetische uitstap te maken. Hoe zou een wereld eruit zien zónder FLOSS[1]? En dan met name, hoe zou de (nieuwe-media) kunstwereld er uit zien? Nieuwe-media kunst zou elitair zijn. Er moet immers duizenden euro's worden neergelegd voor een kopie van Max/MSP, Flash, de Adobe suite en andere grote softwarepakketten. Kunstinstituten zouden hun leerlingen aanraden om Apple laptops te kopen, en daarbij zouden ze, gesponsord door de grote softwarebedrijven, tegen gunstig tarief studenten-licenties van deze grote pakketten ter beschikking stellen aan de studenten. Zodat deze studenten na hun afstuderen wel de "Ultimate edition" van Adobe *Creative Suite* zouden móeten kopen, omdat ze geen enkel alternatief aangeboden hebben gekregen tijdens hun studie.

Oh, wacht. Dit is helemaal geen horrorbeeld, maar realiteit. Op de Gerrit Rietveld academie lopen écht alleen maar mensen rond met mac-laptops en je hoeft niet te proberen om bij systeembeheer aan te kloppen met een vraag over een ander besturingssysteem. In sommige lessen leggen de docenten uit hoe de trial-versie van *Flash* gekraakt kan worden zodat hij het langer doet dan 30 dagen. Op de academie ontbreekt iets: een frisse wind die door het instituut dat "systeembeheer" heet waait, een docent die doceert over de mogelijkheden van opensource software. Meer algemeen: het besef dat er méér is dan de "creatieve" mogelijkheden die proprietary software biedt.

2 .1 Open source

Rond 1996 kwam ik voor het eerst in aanraking met een van de principes waarop opensource software uit ontstaat: het overkomen van de frustraties van een vaakgebruikt programma. Tijdens een informatica-project op onze school stonden er gedurende 3 weken een aantal computers in de klas. Hierop leerden wij hoe tekstverwerken ging, en hoe een spreadsheet-programma werkte. Tenminste, dat deed de rest van de klas: ik was bezig om een spelletje te programmeren; een flipperkastspel. Ik had namelijk op een computer thuis ook zo'n spelletje, maar het frustreerde me altijd dat het maar één soort flipperkast had. Ik wilde iets maken waardoor het aantal flipperkasten onbeperkt was, zodat ik altijd weer een nieuw *level* kon spelen.

Er is hier een parallel te trekken naar het ontstaan van opensource software. In zijn *GNU manifesto* [18] beschrijft Richard Stallman hoe hij eind jaren '70 steeds meer werd gefrustreerd door de toenemende

[1] *Free Libre Open Source Software*; ik gebruik de termen in dit hoofdstuk door elkaar heen: wanneer ik *opensource* zeg bedoel ik altijd de meer algemenere term FLOSS

commercializering van de eerste software-tools en door het feit dat deze tools vaak niet deden wat hij wilde. Hij ging zijn eigen programma's schrijven en maakte ze openbaar als *Free Software*, waarbij "free" in dit geval "vrij" betekent, niet "gratis" (*free as in speech, not beer*). De software werd vrijgegeven onder de General Public License, die ervoor zorgt dat de software vrij is en altijd vrij zal blijven, ookal wordt gebruikt voor een ander product. Dit principe moedigde andere programmeurs ook weer aan om hun software ook onder deze licentie vrij te geven en zo ontstond langzaam maar zeker de collectie software die nu als het GNU project bekend staat, en als basis dient voor de meeste gratis operating systems ("GNU/Linux").

In het manifest dat hij in 1985 schreef verdedigt hij enkelen van de belangrijkste bezwaren tégen opensource software. Een van die bezwaren is het feit dat er geen geld te verdienen valt: "Won't everyone stop programming without a monetary incentive?" Nee, zegt Stallman, want naast het feit dat programmeren voor goede programmeurs gewoon iets is wat ze niet kunnen laten (net zoals musici toch blijven spelen, ookal zijn ze straatarm), is het niet zo dat je níet betaalt wordt, maar misschien is het wat minder. De beloning is ook immaterieel: Het feit dat je een baan hebt waarbij je je creativiteit op zo'n manier in kwijt kunt.

Twee jaar later, in 1998, kwam ik pas écht in aanraking met open source. Ik vroeg een van de begeleiders van het informatica-project in '96 te hulp bij mijn laatste werkstuk voor school over Artificiële Intelligentie, waarin ik een simulatie van een aantal robots wilde programmeren. Naast de nodige programmeertips introduceerde hij me ook met *Emacs* [19], een tekstverwerkingsprogramma. Het was een tekst-editor, een programma waarmee "platte" teksten werden getypt, met ondersteuning voor veel programmeertalen. In het begin vond ik het maar een vreemd programma: de toetsen voor copy-pasten werkten bijvoorbeeld niet zoals ik dat gewend was van Windows. In plaats daarvan moest ik toetsensequenties gaan leren zoals `ctrl-x ctrl-f` voor het openen van een bestand, en `ctrl-x ctrl-s` voor het opslaan. Ondanks deze steile *learning curve* bleef ik het programma toch gebruiken, steeds meer de kracht ervan ontdekkend, tot aan de dag van vandaag. Pas jaren na mijn introducie met Emacs kwam ik er pas achter dat de eerste versie van ervan al in 1979 was geschreven door Richard Stallman zelf, en een van de allereerste GNU programma's was.

Het feit dat een stuk software zo lang kan bestaan en nog steeds ontwikkelt om aan de standaarden van vandaag te blijven voldoen, kan volgens mij alleen worden gerealiseerd middels opensource software. Noem maar eens een commercieel softwarepakket dat al 30 jaar bestaat en nog steeds wordt ontwikkeld! Richard Stallman is al lang niet meer de *maintainer* van Emacs: ontwikkelaars komen en gaan in de loop der jaren, maar de groei blijft: gedreven vanuit de vraag van de gebruikers ervan. Diezelfde "itch" die ik had bij het flipperkastspel houdt elk stuk software in leven.

2 .2 Sociale aspecten van vrije distributie

Vrije distributie van opensource software. *Free, Libre* Open Source Software, om precies te zijn. De verschillende open source licenties geven de auteur allen de garantie dat hij vrij is om het programma te gebruiken en de broncode van het programma te bestuderen, of erop verder te bouwen.

De software is dus vrij ter beschikking, maar is het ook *gratis*? Nee. Alhoewel er niets betaald voor hoeft te worden, is het beter om opensource software te beschouwen als een *geschenk*. Zoals Marcel Mauss dat doet in het boek "The Gift" [20]. Vanuit een anthropologisch perspectief bekijkt Mauss hoe "primitieve" samenlevingen het geschenk gebruikten als een structurerend middel voor het vormgeven van hun maatschappij. De verplichting van het schenken toont de schenker zichzelf als genereus, gul; en verdient daarom respect. Het aannemen van een geschenk eerbiedigt juist dit respect. Het legt tegelijkertijd een soort belofte dat het geschenk ooit in equivalent terug zal worden gegeven: het legt een morele band tussen de schenker en de ontvanger, want de schenker schenkt niet alleen het object, maar daarmee ook een deel van zichzelf: "the objects are never completely separated from the men who exchange them".

Programmeurs en contributors aan FLOSS hebben deze notie volgens mij ingebakken. Een programmeur gebruikt voor zijn werk een ander programma of een library, het in het achterhoofd toch altijd de gedachte dat zijn eigen werk "rust" op het werk van anderen. Mocht hij dan een fout tegenkomen in een programma wat hij gebruikt, dan vindt hij het niet erg om erin te duiken, het te onderzoeken of de *bug* te rapporteren aan de makers ervan. Hij is het in feite aan ze verplicht, doordat zij hem het programma hebben geschonken. Zij beschouwen zijn *bugreport* dan weer als een geschenk.

In het licht van opensource software-ontwikkeling lijkt het gift-economy principe dus te werken. Ontwikkelaars en *power users* kunnen ermee overweg. "Maar werkt het ook voor mijn oma?" De opensource software wereld maakt een grote groei door, met name door "entry-level" linux-distributies als Ubuntu, die het installeren en gebruiken zó eenvoudig maken dat de *terminal* ver in de menus is weggestopt. In hoeverre gaat het principe van de gift-economy nog op met het licht op deze groei? "Gewone" eindgebruikers zullen nooit de verplichting voelen om iets terug te doen voor de community; sterker nog, ze hebben geen gevoel dat er een community ís. Dit fenomeen wordt gebruikt door de bedrijven die geld verdienen aan opensource software: het leveren van support en ondersteuning.

2 .3 Een FLOSS model voor nieuwe-media kunst

Van nature zijn "traditionele" kunstenaars beschermend naar hun werk toe. Het werk is een "darling", een ding van jezelf, er is lang en hard aan gewerkt. De kunstenaar wil dat het werk verkocht word, hij moet kunnen leven van zijn kunstenaarschap.

Wat gebeurt er als we de *gift economy* introduceren in de kunstwereld? Kan opensource van toepassing zijn op kunst, en wat is het voordeel van het "opensourcen" van een werk? Voordat deze vragen beantwoord kunnen worden moet ik eerst toelichten dat er een groot verschil is tussen kunst die in analoge vorm bestaat en kunst die uit digitale media bestaat.

Voordat de digitale media hun intrede deden, was er nog niets aan de hand: het was onmogelijk om een exacte kopie te maken van een werk. Schilders verdienden hun brood met het schilderen van schilderijen, en soms profiteerden de schilder-kopieerders hiervan mee. Het maken van een replica was echter zo veel werk, dat het nooit op zo'n grote schaal werd gedaan dat de auteur van het origineel er last van had.[2].

In het digitale tijdperk is er geen verschil meer tussen het origineel en de kopie ervan. De actie van het kopieren is zelfs onlosmakelijk verbonden met het feit dat iets digitaal is: er wordt continu gekopieerd. Wanneer we een bestand "verplaatsen", wordt het eerst gekopieerd, en pas daarna van de oorspronkelijke locatie verwijderd. Wanneer we een video kijken of een tekst lezen, worden de video-frames of paragrafen in het videogeheugen van de grafische kaart gekopieerd. Kopieren is alom aawezig: zonder de kopie kan een digitale entiteit niet bestaan.

Natuurlijk is het de entertainment- en software-industrie die de grootste moeite heeft met deze eigenschap, en zijn er allerlei kromme bochten bedacht om het kopieren tegen te gaan: van DRM[3] tot online-activatiesystemen zoals WGA[4]. Er heerst nog steeds het ouderwetse idee dat een digitaal product in feite hetzelfde is als een analoog product: dat iedere kopie een bepaalde economische waarde vertegenwoordigt, en dat daarom elke kopie beschermd moet worden.

De kunstwereld reageert al even krampachtig op de inherente kopieerbaarheid van de digitale media. Zelfs al in de videokunst begon dit: al kon een (analoog) videowerk niet *exact* gekopieerd worden, het kon wel geautomatiseerd worden gedaan en daarom werd er een certificaat van echtheid verkocht tezamen met de videoband. De gehele traditionele kunstmarkt is gebaseerd op het economische principe van de schaarste aan goederen, en een cultuur van "ongelimiteerd kopieren" kan hier niet in bestaan.

Laten we de traditionele kunstwereld achter ons liggen en niet meer proberen om de digitale kunsten in dit model te gieten. Laten we accepteren dat ons werk gekopieerd zal worden en laten we ons werk zoveel mogelijk vrijheden te geven door het meegeven van een *open content* licentie. Wat is er dán mogelijk?

De meest bekende familie licenties zijn de *Creative Commons*. Deze lijken een soort "open source" te zijn voor creatieve goederen: voor teksten, afbeeldingen audio, videofragmenten. Ze bieden een wijde reeks mogelijkheden voor artiesten die meer controle willen hebben over hoe hun werk mag worden gebruikt door derden. Dit loopt uiteen van even restrictief als copyright ("Attribution Non-Commercial No Derivaties") tot bijna *public domain* ("Attribution"). Ze doen het goed, sites als Flickr ondersteunen het, het wordt een soort merk. Maar, zoals Cramer opmerkt in [22], deze licenties rusten op geen enkele onderliggende ethische code, niet op een filosofisch manifest of politieke constitutie: ze garanderen geen enkele vrijheden voor het werk, ze geven alleen maar restricties aan. De licenties

[2] Walter Benjamin vraagt zich in deze context zelfs af, "Is the author of a copy, an author?"[21]
[3] Digital Restrictions Management
[4] Windows Genuine Advantage

zijn zó verschillend van elkaar dat het enige wat ze gemeen hebben het logo is. Het zeggen dat een werk een CC licentie heeft is dus op zichzelf een betekenisloze uitspraak.

Het is tijd voor een soort *GNU manifesto* voor de kunstenaar, waarin de vrijheden van een werk worden gegarandeerd en het auteursrecht van de mensen die aan het werk meewerkten wordt gerespecteerd. Waarin op alle prangende vragen een antwoord wordt gegeven. Een mogelijke licentie die beter de GNU filosofie nastreeft is de Free Art License[23, 24]. Deze licentie wil een GPL zijn voor kunstuitingen en lijkt tot op heden de beste keus voor het maken van open, vrije werken in het culturele domein. Het garandeert dat een werk vrij is, en altijd vrij zal blijven, ookal wordt het gekopieerd en/of aangepast. Het erkent de kracht van het internet door onbeperkte toegang tot het werk voor het publiek te garanderen. en pleit voor een economisch model voor kunst gebaseerd op geven, delen en uitwisseling.

Er zijn genoeg kunstenaars die software maken als onderdeel van hun kunstwerken, maar er zijn er nog niet veel die dit doen onder een open licentie. Twee collectieven wil ik noemen die dit wel doen: GOTO10 is zo'n kunstenaarscollectief dat een groot belang hecht aan de openheid van software in de kunst en zijn *roots* heeft in de muziek-scene. Dyne.org, aangevoerd door software-activist / kunstenaar "Jaromil" is meer een netwerk dat gevormd is rondom *hardcore* programmeurs, met links met het Blender project en het Freaknet medialab.

2 .4 Naar een open, collaboratief proces

Open-source processen zijn een krachtig, zelf-organiserend medium dat een ander sociaal-economisch model representeert: een economie van giften. Nu dit opensource-model zich in de software-industrie al bewezen heeft, ben ik van mening dat de kunstwereld een dergelijke mentaliteitsverandering ook moet ondergaan om weerstand te bieden aan de software-monopolies en corporate databanken. Hiertoe moeten "traditionele" kunstenaars eerst hun ego kwijt, en bereid zijn tot geven.

Net zoals opensource software een ecologie creëert van programmatuur die elkaar versterkt en aanvult, kan dit voor *open content* ook gebeuren. Een cultuur waarin wordt gecreëerd en geremixed, waarin kunstenaars hun werk "bevrijden", aan het grotere goed schenkend, en waarin anderen het werk nemen, de oorspronkelijke auteur(s) *credits* gevend, voortbouwend op hun werk.

Zeker wanneer een werk een "open" licentie heeft, is een werk dat software gebruikt onder continue verandering. Het maakt deel uit van de software-ecologie die inherent besloten ligt in het gebruik van een open licentie. Het werk heeft de potentie om, onder de invloed van gebruikers en andere ontwikkelaars, van gedaante te veranderen: het werk wíl veranderd worden, het wíl verder evolueren. Het auteurschap van zulke "open" projecten ligt bij alle deelnemers eraan. Door het gebruik van een écht open licentie zoals de Free Art License zullen de (mede)auteurs van een werk altijd erkenning blijven houden.

Hoofdstuk 3

Software als constructief proces

Wat *is* software precies? In "Words made flesh"[25] begint Florian Cramer met het analyseren van code aan de hand van de kaballah: een numerologische interpretatie waardoor er dichter bij "god"gekomen kan worden. Kabbalisten hebben een eindige set symbolen, namelijk de letters van de naam van god, JHWH, die worden gemanipuleerd en in volgorde gezet om nieuwe interpretaties van "god"te geven.

Dit is eigenlijk het pure principe van programmeren: met een beperkte, afgebakende set symbolen worden permutaties en nieuwe rangschikkingen gemaakt waardoor er betekenis ontstaat.

Je zou kunnen zeggen dat bovenstaande ook geldt voor de literatuur: door het rangschikken van symbolen (namelijk: letters, spaties en leestekens) wordt ook een eigen betekenis gecreëerd die bij het lezen ervan een eigen werkelijkheid doet ontstaan. Echter, literatuur kan ambigu zijn, en is subject aan de interpretatie door de lezer, waarentegen software nooit ambigu is (tenminste tot "quantum-computing"doorbreekt). Software is altijd deterministisch: gegeven een staat van het systeem, plus een programma, zal de uitvoer van het programma altijd hetzelfde zijn.

3 .1 Software leeft

Als software een reeks symbolen is, is software dan uitsluitend tekst? Nee. Naast de tekst van de software, zijn er nog andere aspecten die deel uitmaken van software: *Software is levend*. Deze stelling is op twee manieren op te vatten.

Ten eerste is er het feit dat software bedoeld is om uitgevoerd te worden, het bevat instructies om een machine aan te sturen. Tijdens het uitvoeren van een ervan gaat de software "leven": het doet waarvoor het gemaakt is. In besturingssystemen heet een software die wordt uitgevoerd, toepasselijk genoeg een "proces". Software “leeft” als het draait op een machine die daarvoor geschikt is. Op unix-systemen is zo’n proces te stoppen via het commando “kill”.

Na het schrijven van een programma moeten de instructies van de software eerst nog door de machine vertaald worden naar instructies van het allerlaagste niveau: naar machinetaal[1]. Machinetaal is de enige taal die de hardware van de computer “letterlijk” kan begrijpen, en software ondergaat dus altijd een vertaalslag. In sommige gevallen gebeurt deze vertaalslag tijdens het schrijven van de software (*C*, *C++*, etc. worden “gecompileerd”), in sommige gevallen gebeurt dit pas tijdens het uitvoeren (*Python*, *Perl*, *PHP*, etc. worden “geïnterpreteerd”), en in weer andere gevallen is het een mix tussen deze twee (*Java*, *C#*, etc. draaien in ”virtual machines”).

Software is een construct en steunt op andere constructen: de *dependencies*. Dit zijn andere stukken software, *libraries* genoemd, waar de software afhankelijk van is. Software kan de bestaande libraries gebruiken die al in het besturingssysteem zitten, of kan libraries nodig hebben die wordt aangeboden door derde partijen. Niet alleen het uitvoeren maar ook het bouwen (compileren) van software vereist deze een specifieke set aan al geinstalleerde onderdelen.[2] Naast dit pakket van eisen die aan de software wordt gesteld, geldt dit ook voor hardware: de *systeem-eisen*. Software komt altijd met een pakket van hardware-eisen waaraan een systeem waar de software op draait aan moet voldoen. De

[1] Tenzij de programmeur in “assembler” schrijft: dat ís al (een human-readable versie van) machinetaal

[2] Gelukkig maken *installers* het ons als gebruiker wél eenvoudig om software te installeren

systeem-eisen stellen welk soort hardware noodzakelijk is voor het draaien van de software. De eisen zijn afhankelijk van de software en varieren van heel simpel ("486 of hoger") tot zeer complex ("SPARC mainframe met redundante raid storage").

De tweede manier waarop software als “levend” kan worden getypeerd is het feit dat software nooit af is.

Bij een regulier literair proces (bv. het schrijven van een boek) is het zo dat een auteur zich een paar maanden of langer terugtrekt op een zolderkamer en met een meesterwerk er weer vanaf komt. Het moment van het ter perse gaan van het boek is meestal (zeker in het geval van fictie) het moment dat het corpus van woorden die het boek vormen, niet meer verandert, spel- en zetfouten daargelaten.

Het schrijven van software is daarentegen een continu proces. Het is een stelsel van cyclische processen. Tijdens het programmeren zelf wisselen code schrijven en de code testen elkaar af. Wanneer het “klaar” wordt bevonden, wordt er een “release” gedaan: een *snapshot* van de code zoals die op dat moment is wordt voorzien van een versienummer en in productie genomen: gebruikers gaan ermee aan de slag. Wanneer er bugs of security-issues in de release zich voordoen, of wanneer er een nieuwe functionaliteit af is, wordt er een nieuwe release gedaan. Doot dit cyclische karakter van software is deze eigenlijk nooit geheel “klaar”. Sterker nog, het is al verouderd wanneer het bij de gebruiker aankomt. Denk maar aan de software-update functie die tegenwoordig in vrijwel elk programma is ingebouwd.

Er is niet één manier waarop een softwareproject wordt ontwikkeld en beheerd. Hoewel dit cyclische proces zoals hierboven beschreven zich bij alle software voordoet, kent elk project zijn eigen vorm van organisatie. Wanneer het project uit meer dan 1 ontwikkelaar bestaat, is er vaak een soort kernteam van “core” ontwikkelaars die met elkaar in contact blijven via mailing lists en IRC [3]. Bij de grotere projecten kunnen dit ook meerdere teams zijn, die elk een stuk van het programma in beheer hebben. De natuurlijke organisatievorm van elk project is een netwerk. Dit komt voort uit het feit dat developers vaak overal ter wereld zitten en altijd het Internet als primair contactmiddel gebruiken. Echter, de topologie van het netwerk verschilt per project.

Over de organisatie van softwareprojecten gaat Eric S. Raymond’s klassieker “The cathedral vs. the bazaar” [26]. Dit essay uit 1997 beschrijft de twee uitersten in het ontwikkelproces van opensource software: Het “cathedral” model, waarbij de broncode wel wordt vrijgegeven, maar alleen bij de release van een nieuwe versie van het product, en het “bazaar” model: waarbij de software ook tussen releases altijd voor iedereen toegankelijk is over het internet. Een van de belangrijkste projecten die het “cathedral” ondersteunen is Emacs, waar Raymond zelf ook aan mee heeft gewerkt. Als “bazaar” model wordt de ontwikkeling van de Linux kernel genoemd: het hart van het operating system.

In de architectuur van de verschillende versiebeheer-systemen zoals die in de (opensource) softwarewereld worden gebruikt, is ook een analogie te trekken met deze vergelijking. Elke programmeur gebruikt een versiesysteem: programmeren is niet alleen maar schrijven, maar vooral herschrijven. Bugfixes, nieuwe functionaliteiten, ze vereisen dat oudere code wordt herzien. Een versiesysteem houdt al deze wijzigingen in de code nauwkeurig bij, en dwingt documentatie van de wijzigingen af. Op deze manier dient het als backup en als naslagwerk. Traditionele, “cathedral”-achtige versiebeheersystemen zijn allen *centraal* van opzet. Deze familie, komend van de systemen *SCCS* en *RCS* (’80), nog later *CVS* (’90) en tegenwoordig Subversion, werken vanuit het principe dat alle broncode op 1 centrale server staat: alleen hier worden de wijzigingen bijgehouden. Elke programmeur werkt met een kopie waar 1 versie op staat: zodra hij klaar is zet hij het op de centrale server. Een nieuwere trend is om het versiebeheer *decentraal* te doen. Elke programmeur heeft een eigen repository met versies erop, en kan naar hartelust nieuwe versies maken. Soms “pullt” hij informatie binnen van repositories van de andere programmeurs, soms “pusht” hij zíjn wijzigingen naar anderen terug. Voorbeelden hiervan zijn *git* (waarin de Linux kernel wordt beheerd), *bazaar* (gebruikt door Ubuntu) en *Mercurial* (o.a. Firefox).

3 .2 Een software-ecologie

De vergelijking “cathedral” versus “bazaar” is een mooi verhaal met een ideëel randje, maar de tekst is meer dan 10 jaar oud, en ontwikkelingen van de laatste jaren hebben de kijk hierop wel veranderd. De

[3] Internet Relay Chat

écht “pure” cathedral-projecten zijn er haast niet meer: de broncodes van de allerlaatste ontwikkelaarsversie van Emacs is tegenwoordig gewoon voor iedereen in te zien. Ik denk dat de analogie nu meer betrekking heeft op de vorm van ontwikkelen. De linux-kernel is dezer dagen meer een “cathedral” gaan lijken: qua organisatievorm een piramide, waarin elke wijziging van eniger betekenis echt moet worden gereviewed door de allerhoogste baas. Ditzelfde geldt voor Ubuntu. Grote projecten hebben echt een kapitein nodig om vooruit te kunnen: ze vereisen *visie*.

Het mooie van visies is dat ze kunnen verschillen, en opensource software laat dit gewoon toe. Verschilt een ontwikkelaar in een project zó erg van mening met de anderen in het project, dan kan hij de broncode nemen, en een *fork* maken. Een fork is een splitsing in het ontwikkelpad van een project: een ontwikkelaar gaat “ervandoor” met de broncode en begint zijn eigen project, gebaseerd op het oude project. In proprietary software zou dit echt een taboe zijn: diefstal zelfs. Maar in de opensource wereld is dit juist een goed ding. Door de opensource licentie blijft de fork blijft ook vrij beschikbaar, en draagt zo bij aan een rijker, gevarieerder aanbod van software: de *software-ecologie*. Want dat is de opensource wereld eigenlijk: een universum van meer en minder opzichzelf staande projecten. Een ecologie waarin software-projecten komen en gaan, zichzelf in stand houdend, of drijvend op het succes van een ander (soms proprietary) project, of ergens eenzaam aan de top.

Elk project, en dus ook forks van andere projecten, zijn in deze ecologie onderhevig aan de natuurlijke selectie. Dit selectieproces is een samenspel tussen de gebruikers van een project en de ontwikkelaars ervan: wanneer er te weinig ontwikkelaars zijn, zullen de gebruikers afhaken (door bugs of te weinig nieuwe features); wanneer er te weinig gebruikers zijn zullen ontwikkelaars afhaken (omdat ze te weinig feedback van gebruikers krijgen). [4] Wanneer er geen gebruikers en ontwikkelaars meer zijn voor een project, zal het “afsterven”, wat in de software-wereld zoveel betekent als er nog wel zijn maar niet meer beheerd worden, *orphaned* zijn. Het is dan in limbo: een project “verdwijnt” niet echt; het wacht op een nieuwe ontwikkelaar die zich aandient om het project weer op te pakken.

Wáár bevindt zich deze ecologie? Online. In de centrale, openbare “source code repository” sites die zijn opgezet: De grootste en bekendste hiervan is *Sourceforge*, maar er zijn ook *google code*, *github*, *GNU savannah*, *bitbucket*, *launchpad*, ... én natuurlijk in private repositories van mensen: persoonlijke websites, universiteiten die servers ter beschikking stellen voor *mirrors*, etc, etc. Het zou een mooi apart project zijn om dit precies in kaart te brengen.

3 .3 Software als materiaal

De laatste jaren kenmerken zich door een groei in verschillende tools en frameworks die speciaal zijn gemaakt voor het creëeren van nieuwe-mediakunst. Processing, Arduino, OpenFrameworks maken het eenvoudig voor kunstenaars om moderne technologieën te gebruiken binnen een klassiek kunstmodel. Een aspirerend kunstenaar, die niet veel van computers en programmeren weet, kan gewoon Processing installeren en de enorme collectie aan voorbeeldcode doorzoeken en zo een eigen “kunstwerk” bij elkaar copy-pasten.

Dit ligt in de lijn met Alan Kay’s gedachte dat de computer vooral een metamedium is: het verschaft toegang tot het creëren van nieuwe, nog onontdekte media. En door het aanbieden van makkelijke tools voor het maken van *sketches*, prototypes, wordt de gebruiker begeleid in dit proces. Maar leggen deze pakketten je ook weer niet een limiet op? Ze bieden een standaardpakket, een basisfunctionaliteit die een standaard (beeld)taal dicteert. Hier is wel omheen te komen door andere libraries te gebruiken dan de standaard, maar dat is gelijk een stap verder en ligt vaak buiten het bereik van de “amateur”. Binnen Kay’s visie past het wel, een standaard beeldtaal voor een varieteit aan programmatuur, maar voor het echt creeren van *nieuwe media* lijkt het me een limiet: een computer blijft dan een tool, maar staat niet in het middelpunt.

Ik ben van mening dat een kunstenaar in staat moet zijn om het medium wat hij gebruikt, te doorgronden. Mijn interesse gaat meer uit naar mediakunstenaars die de implicaties en connotaties die software met zich meebrengt, begrijpen, gebruiken en/of misbruiken voor hun eigen doeleinden. Die echt proberen diep door te dringen in het medium. Bijvoorbeeld JODI, die HTML binnenstebuiten gooiden. Of "glitch art"kunstenaars die bugs in software exploiteren en omtoveren in onverwachte schoonheid. En de andere al eerder aangehaalde voorbeelden: het ontwerp van *weird languages* of receptuur in code, live coding, et cetera.

[4] Natuurlijk zijn er ook projecten waarbij de ontwikkelaar ook de gebruiker ís: vaak ontstaan projecten op deze manier

Maar meer nog zie ik het *construeren* van software als een creatief proces. Het bedenken en uitvoeren van een project. Het opzetten van de verschillende modules, het in elkaar laten grijpen van onderdelen en op het juiste abstractieniveau kunnen beschouwen van de code. In de *elegantie* van de softwareconstructie. Matthew Fuller zelf schrijft hierover [27] het volgende:

> *"In order to achieve elegance use of materials should be the barest and cleverest."*

Het zo optimaal mogelijk gebruiken van de systeembronnen die je tot je beschikking hebt. De juiste afweging tussen abstractie en pragmatisme.

3 .4 Het constructief proces

Software leeft: het bevindt zich in een doorlopend constructief proces. Hierdoor is het nooit af, heeft geen definitieve vorm. Het is afhankelijk van de systeemeisen van hardware en van dependencies op andere software. Omdat computersystemen veranderen, zal de software ook moeten veranderen. Software is altijd gebouwd op andere software, die ook altijd in beweging is. Het maakt deel uit van de software ecologie. Het klinkt lastig, een werk dat is gebaseerd op iets dat altijd verandert: het moet worden bijgehouden, aangepast aan nieuwere versies, en daarbij rekening houdend dat het op oudere systemen nog steeds werkt. Maar de veranderingen in deze ecologie over tijd zijn niet alleen maar negatief, ze bieden juist ook nieuwe mogelijkheden voor de kunstenaar. Onder invloed van de veranderende systemen en dependencies kan het werk zélf evolueren: er kunnen nieuwe toevoegingen en mogelijkheden bijkomen waardoor het werk wezenlijk verandert, op onvoorziene manieren.

Natuurlijk is het mogelijk om een werk dat software gebruikt te "bevriezen": een *snapshot* te nemen van de software op dat moment, plus de hardware waar het op draait, en dit als het ware te archiveren. Ik beschouw dit echter niet als een interessante invalshoek: de software gaat dood, het werk wordt geconserveerd, en heeft alleen nog maar een archeologische waarde.

De computer is een *metamedium*, dat door kunstenaars de mogelijkheid biedt verkenningen te doen in de creatie van nieuwe, nog onontgonnen media, maar de taal waarmee die verkenningen plaatsvinden is altijd software. Werken die software gebruiken zijn pas interessant wanneer de werken het belang tonen van de software die ze gebruiken. In het constructieproces is een kritische houding ten opzichte van de "software-materialen" die ze gebruiken, noodzakelijk. Limitaties die software je als kunstenaar opleggen, moeten niet voor lief worden genomen maar goed worden overwogen, desnoods van materiaal wisselend of de onderliggende software *patchend*.

Hoofdstuk 4

Conclusie

In de inleiding van deze scriptie bracht ik de volgende stelling onder woorden:

> *Een werk dat software tentoonstelt of gebruikt als controllerend proces heeft geen definieve vorm.*

In drie hoofdstukken heb ik een beeld geschetst van drie verschillende eigenschappen van software art die ik centraal vind staan: het gebruik van software zélf als expressief middel, software als natuurlijke en open samenwerkingsvorm, en software als een constructief proces, een basismateriaal.

Uit het eerste hoofdstuk concludeerde ik dat software een expressief middel is dat in zijn pure vorm gebruikt kan en móet worden als communicatief medium. Een toegevoegde tijdsdimensie zoals in live coding en time-based-text, versterkt het gevoel van expressiviteit omdat het de emoties kan communiceren die met het proces van het schrijven van software gepaard gaan. Wanneer software zichtbaar is, toont het altijd de belofte van een mogelijke executie van de code. Zo refereert het aan de procesmatige eigenschappen van zichzelf. Deze belofte van executie maakt de mogelijke uitkomst van het proces subjectief aan de beschouwer: door het lezen van de code speculeert de toeschouwer over het mogelijke resultaat. Aldoende heeft het werk waarin de software zich bevindt geen definitieve vorm: het zet een speculatief proces in gang.

Hoofdstuk twee stelde de processen rond auteurschap aan de kaak. Opensource-processen zijn een krachtig, zelf-organiserend principe die een ander sociaal-economisch model presenteren: een economie van giften. Nu dit opensource-model zich in de software-industrie al bewezen heeft, ben ik van mening dat de kunstwereld een dergelijke mentaliteitsverandering ook moet ondergaan om weerstand te bieden aan de software-monopolies en corporate databanken. Hiertoe moeten “traditionele” kunstenaars eerst hun ego kwijt, en bereid zijn tot geven. Zeker wanneer een werk een “open” licentie heeft, is een werk dat software gebruikt onder continue verandering. Het maakt deel uit van de software-ecologie die inherent besloten ligt in het gebruik van een open licentie. Het werk heeft de potentie om, onder de invloed van gebruikers en andere ontwikkelaars, van gedaante te veranderen: het werk wíl veranderd worden, het wíl verder evolueren. Het auteurschap van zulke “open” projecten ligt bij alle deelnemers eraan. Door het gebruik van een écht open licentie zoals de Free Art License zullen de (mede)auteurs van een werk altijd erkenning blijven houden.

In het derde hoofdstuk bleek ook dat deze definitieve vorm voor een werk ontbreekt omdat software *leeft*. Omdat computersystemen veranderen en omdat software altijd is gebouwd op andere software, zal het werk mee moeten groeien om te kunnen blijven bestaan. Door software te gebruiken als controllerend proces in een werk, wordt de computer erkend als een *metamedium*, dat de mogelijkheid biedt verkenningen te doen in de creatie van nieuwe, nog onontgonnen media. Werken die software op deze manier gebruiken moeten het kritische belang tonen van de software die ze gebruiken: In het constructieproces is zo’n kritische houding ten opzichte van de “software-materialen” die gebruikt worden, noodzakelijk. Limitaties die software een kunstenaar opleggen, moeten niet voor lief worden genomen maar goed worden overwogen, desnoods van materiaal wisselend of de onderliggende software *patchend*.

Aldoende hoop ik een beeld te hebben gegeven van de aspecten die mij aan software en de relatie met kunst het meest bezig houden. De centrale stelling van deze scriptie, waarin ik stel dat een

werk dat software centraal stelt nooit een definitieve vorm bereikt, onderstreep ik nog steeds. In mijn eigen toekomstige praktijk als autonoom en/of faciliterend kunstenaar en/of ontwerper, wil ik er naar streven deze stelling altijd te doen gelden. Ik hoop in mijn werken het belang te benadrukken van een open kunst-ontwikkelproces, maar ook een openheid in het uiteindelijke werk door het tonen van het belang van de software dat een werk voortdrijft.

4 .1 Dankwoord

Tot slot wil ik de volgende mensen bedanken: Willem van Weelden voor het begeleiden van deze scriptie en voor het mentorschap in 3 jaar ID/UM, Jorinde Seijdel in haar rol als rijksgecommitteerde voor het (hopelijk) positief beoordelen van deze scriptie, Jan Robert Leegte en Esther Polak voor het begeleiden van mijn afstudeerwerk, Manel Esparbé i Gasca voor het coördineren van DOGTIME, en verder alle andere docenten van DOGTIME die mij hebben helpen komen waar ik nu ben. Als laatste mijn vriendin Ezra ten Bruin voor haar steun en geduld met mij tijdens de eindexamenperiode.

Arjan Scherpenisse,
Juni 2009

Bibliografie

[1] Geert Lovink. New media, art and science: Explorations beyond the official discourse. http://laudanum.net/geert/files/1129753681/, 2005.

[2] Noah Wardrip-Fruin and Nick Montfort, editors. *The New Media Reader*. The MIT Press, 2003.

[3] Lev Manovich. *The Language Of New Media*. The MIT Press, 2001.

[4] Matthew Fuller, editor. *Software Studies*. The MIT Press, 2008.

[5] Friedrich Kittler. There is no software. 1995.

[6] Michael Murtaugh. Interaction. In Fuller [4], pages 143–152.

[7] Lev Manovich. Software takes command. http://lab.softwarestudies.com/2008/11/softbook.html, 2008.

[8] Michael Mateas. Weird languages. In Fuller [4], pages 267–275.

[9] Alex McLean. Hacking perl in nightclubs. http://www.perl.com/pub/a/2004/08/31/livecode.html.

[10] The toplap collective. http://www.toplap.org/.

[11] Inke Arns. Code as performative speech act. *Artnodes*, July 2005.

[12] Denis "Jaromil" Rojo. Recept voor barszcz. http://software.kurator.org/recipes/stringbasedcooking.txt.

[13] Wilfried Houjebek. .walk. http://www.medienkunstnetz.de/works/dot-walk/.

[14] Florian Cramer. 10 theses about software art. 2003.

[15] Inke Arns. Read_me, run_me, execute_me. http://www.medienkunstnetz.de/themes/generative-tools/read_me/.

[16] Susanne Jaschko. Process as aethetic paradigm: a nonlinear observation of generative art. http://www.sujaschko.de/downloads/170/generatortalk, 2005.

[17] Denis "Jaromil" Rojo and JODI. Time based text. http://tbt.dyne.org/.

[18] Richard M. Stallman. The GNU manifesto. 1985.

[19] Richard M. Stallman. Emacs: The extensible, customizable, self-documenting display editor. 1979, updated 1981.

[20] Marcel Mauss. *The Gift*. 1923-24.

[21] Walter Benjamin. On copy. *HKMV catalog 2008*, 1936.

[22] Florian Cramer. The creative common misunderstanding. In Mansoux and de Valk [28], pages 128–137.

[23] The free art license 1.3.
http://artlibre.org/licence/lal/en/.

[24] Open content guide: The free art license.
http://pzwart.wdka.hro.nl/mdr/research/lliang/open_content_guide/06-chapter_5/#free_art_license.

[25] Florian Cramer. Words made flesh. 2005.

[26] Eric S. Raymond. The cathedral and the bazaar. 1997.

[27] Matthew Fuller. Elegance. In *Software Studies* [4], pages 87–92.

[28] Aymeric Mansoux and Marloes de Valk, editors. *FLOSS+Art*. GOTO10, 2008.

[29] Christopher Kelty. *Two Bits, The Cultural Significance of Free Software*. Duke University Press, 2008.

[30] Walter Benjamin. The work of art in the age of mechanical reproduction. 1936.