Ontwerp en Realisatie van Hoog Performante Multimedia en HCI-systemen

Robin Marx, Nick Michiels en Jimmy Cleuren

3 februari 2011

1 Inleiding

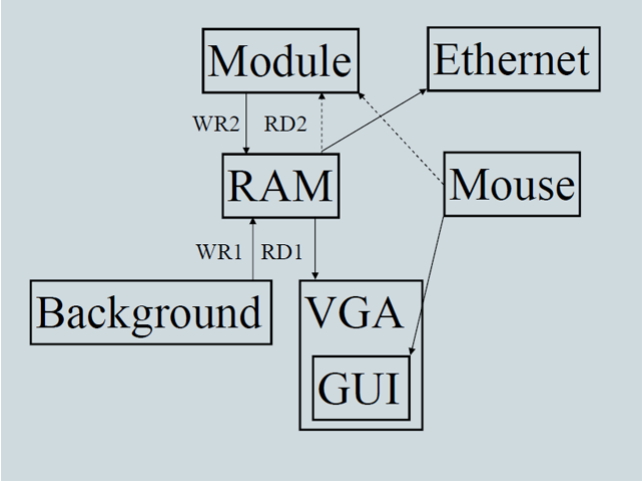
De naam van het project, *AGe*, staat voor *Art Generator* en beschrijft in wezen wat het project is. Door de opkomst van de computers en digitale apparatuur zijn er ook nieuwe kunstvormen ontstaan die daarvoor bijna onmogelijk waren. Zogenaamde procedurele, generative of random art is hier een voorbeeld van. Hiervoor gebruiken artiesten welbepaalde tekenalgoritmes om kunstwerken op te bouwen. Deze algoritmes worden dan gevoed door random waarden van een (pseudo) random number generator om telkens opnieuw een nieuw en origineel resultaat te geven. Afhankelijk van de gebruikte algoritmen zijn er een enorm aantal verschillende kunstwerken mogelijk. Onze inspiraties waren onder andere www.complexification.net en http://abandonedart.org/.

Wanneer we deze websites echter bekijken en openen in de browser (ze maken gebruik van Processing, een taal gebaseerd op Java), zien we dat deze toepassingen erg veel CPU kracht vragen en traag verlopen. Dit komt omdat er vaak heel veel elementen tegelijk moeten getekend worden wat moeilijk is door de sequentiële aard van de werking van een CPU. Een toepassing in hardware kan hier zeker in helpen omdat we de elementen parallel kunnen verwerken. De tekenalgoritmes zijn over het algemeen simpel genoeg dat ze goed doenbaar zijn in hardware, ookal gebruiken ze soms meer ingewikkelde wiskundige functies zoals trigonometrie. Een toepassing in goedkope hardware zou ook interessant zijn in commercieel toepasbare producten, zoals digitale fotokaders die verschillende generatieve kunstwerken kunnen weergeven die constant in evolutie zijn.

2 Algemene opbouw rond RAM

Het project is gebaseerd op een standaard template geleverd door Altera, namelijk de Camera toepassing, die beelden van de camera in het SDRAM zet en dan toont op een VGA scherm. Hoewel we in het begin lang hebben geprobeerd zelf de componenten op te bouwen rond enkel de bestaande

4-port SDRAM module bleek dit te moeilijk. Door de template te gebruiken hadden we meteen een goede VGA controller en ingebouwde werkende SDRAM module. De overbodige elementen zoals koppeling met NIOS en touchscreen werden verwijderd. De SDRAM module vormt de kern van



Figuur 1: Achterliggende structuur AGe voorgesteld in een schema.

het systeem. Hierin wordt de beeldbuffer bijgehouden. In de tekenmodules schrijven we naar deze buffer, terwijl de VGA controller hem uitleest en toont op het scherm. Door de ingebouwde round-robin methode voor de 4 poorten van de SDRAM module kan dit asynchroon gebeuren in aparte klokdomeinen. De 2 16-bit SDRAM modules worden naast elkaar gelegd zodat we 32-bit woorden krijgen, 1 woord per pixel (30 bits voor RGB, 2 statusbits die vrij te gebruiken zijn). Een andere oplossing zou zijn slechts 16-bit woorden te gebruiken. Hierdoor zouden we bijvoorbeeld makkelijker 2 beeldbuffers kunnen gebruiken voor dubbele buffering. Het zou echter ook het aantal beschikbare kleuren drastisch beperken. Omdat voor onze toepassing dubbele buffering niet nodig is, hebben we besloten bij de originele 32-bit woorden te blijven. Dubbele buffering met 32-bit woorden is theoretisch gezien ook mogelijk (er is genoeg geheugen ervoor) maar door de opzet van ons doel zou dit betekenen dat we onder andere dubbele writes zouden moeten doen naar beide buffers om consistent te blijven (of een manier zoeken om het anders te doen) wat de complexiteit gevoelig zou verhoogd hebben.

De tekenmodules hebben dus een 32-bit output en besturen het adres en flag van 1 read en write poort op het SDRAM. Omdat de tekenmodules vaak op random plaatsen moeten schrijven (random pixel locaties op het scherm) moesten we de SDRAM module wel aanpassen zodat deze random read/write aan kon. Dit was niet standaard mogelijk omdat het SDRAM

gebouwd is om lange buffers tegelijk te lezen en schrijven. Dit had als gevolg dat de snelheid waarmee we konden random access doen drastisch vertraagd werd ten opzichte van block access. Door experimenteren bleek dat we gemiddeld 1 operatie kunnen doen per 150 clockticks van de 50MHz clock als de VGA controller ook tegelijk bezig is op 1 read poort. Deze snelheid bleek een zeer beperkende factor voor het project.

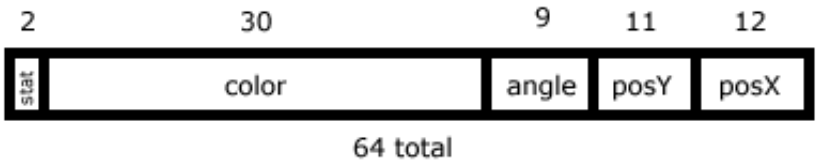
Hoewel het ons doel was parallel te tekenen met meerdere elementen tegelijk bleek dit niet mogelijk met het RAM. Niet alleen is de snelheid beperkt, ook zijn er slechts 2 read en write poorten die we tegelijk kunnen gebruiken, waarvan 1 read poort bezet is door de VGA controller. Dit betekent dat we zoiezo slechts 2 parallelle tekenmodules konden hebben, waarvan er maar 1 kon lezen. Hier zijn er enkele mogelijke oplossingen voor. We konden bijvoorbeeld extra verwerkingen doen tijdens de sync-periodes van de VGA controller, waardoor we wel 2 read poorten zouden hebben. We konden ook proberen de algoritmes zo te bouwen dat we toch block access konden doen op het SDRAM. We besloten echter dat beide methodes te weinig snelheidswinst zouden opleveren tegenover de toch wel zeer verhoogde complexiteit. Daarom hebben we slechts 1 actieve tekenmodule die 1 write en 1 read poort gebruikt en 1 particle tegelijk behandeld (de 2de write poort wordt gebruikt om de achtergrond te tekenen als de actieve module is uitgeschakeld). De tekenmodules werken dus eigenlijk sequentieel, in plaats van parallel, door de beperkingen van het RAM. Een sneller RAM met meer poorten zou dit kunnen oplossen en een meer parallelle oplossing mogelijk maken. Het is immers perfect mogelijk de particles parallel te berekenen, het is enkel niet mogelijk ze parallel random te schrijven.

Voor het omgaan met het RAM hebben we ook verwant werk geraadpleegd. De eerste is een sandgame implementatie (`http://instruct1.cit.cornell.edu/Courses/ece576/FinalProjects/f2010/ss868/ss868/index.html#skip_introduction`). De architectuur die de maker hier gebruikt is echter zeer specifiek voor het sandgame en was dus niet bruikbaar voor ons project. De tweede paper was interessanter en ging over een algemener particle systeem (`http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.89.7862&rep=rep1&type=pdf`). Hier kwamen ze tot dezelfde conclusie als wij over het gebruik van het RAM, namelijk dat het te traag is om veel particles te simuleren. Zij geven wat verschillende alternatieven, maar deze waren te geavanceerd voor ons om te implementeren. Hierover later meer. De laatste bron die we gebruikt hebben waren de verschillende projectpagina's van een vergelijkbaar vak aan Cornell University (`http://people.ece.cornell.edu/land/courses/ece5760/FinalProjects/`). Hier vonden we veel inspiratie voor verschillende onderdelen, waaronder de GUI en de tekenalgoritmes.

3 Algemene tekenmodules

Om de particles een behaviour te geven en deze te laten weerspiegelen aan een visueel resultaat hebben we een aantal tekenmodule's en hulpmodules moeten implementeren. Een aantal voorbeelden zijn drawRectangle.v, drawRectangleBorder.v, drawLine.v, cosLookUpTable.v en cosLookUpTable.v. De drawRectange en drawRectangleBorder gaan respectievelijk de volledige of enkel de border van een vierkant tekenen in een opgegeven kleur. De drawLine gaat met behulp van het Bresenham algoritme een lijn tekenen van een punt (p_1, p_2) naar (p'_1, p'_2). De cosinus en sinus zijn nodig om deze lijnen onder een bepaalde hoek te kunnen laten roteren. Beide zijn geïmplementeerd als een lookup table.

4 Art Generator modules basis

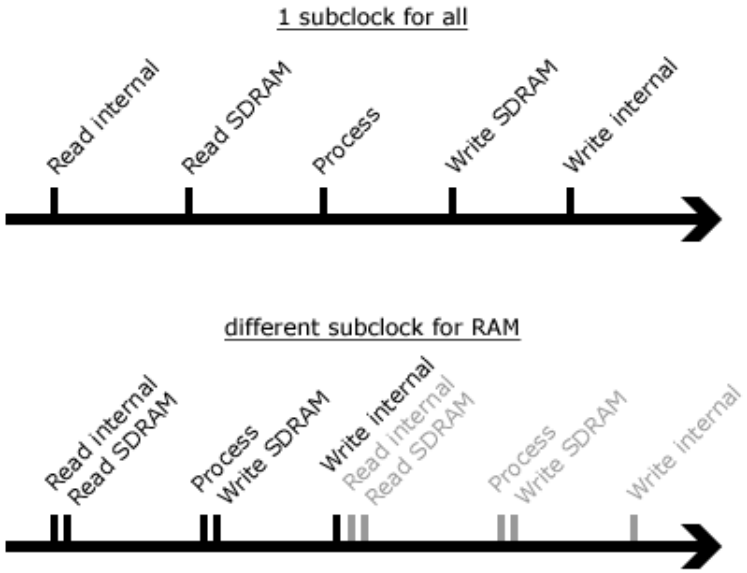


Figuur 2: Particle informatie voor de substrate-tekening past in 64 bit.

De sequentiele aard van de tekenmodules is weerspiegeld in de state-machines die we erin gebruiken. In elke module hebben we een aantal particles die elk een tekenelement van het kunstwerk voorstellen. Deze particles nemen 32 tot 64 bit in het intern geheugen in. Er zijn gemiddeld 10 - 200 particles per module actief, hoewel we ook getest hebben met 8000+ particles en dit perfect werkt. De particles moeten bijgevolg ook eerst uit het intern geheugen worden gelezen vooraleer te gebruiken, en daarna ook worden weggeschreven. Een typische state-machine voor 1 particle ziet er als volgt uit :

- lees particle van intern geheugen
- (lees kleurgegevens op particle pixelplaats in SDRAM)
- bereken nieuwe waardes voor particle (locatie, kleur, ...)
- schrijf nieuwe kleurwaarde op nieuwe locatie in SDRAM
- schrijf particle naar intern geheugen

Telkens 1 zulke loop is uitgevoerd wordt het adres van het interne geheugen verhoogd (of terug op 0 gezet) en de volgende particle verwerkt. Herinner u dat we slechts 1 read/write in het SDRAM kunnen doen per 100-150 clockticks. Dit betekent dat de states van de RAM operaties minstens zo lang moeten duren. De andere states zouden sneller kunnen in theorie (1 clocktick) maar om de implementatie makkelijk te houden hebben we alles geklokt op dezelfde subclock van 50MHz die dus slechts om de 100-150 keer een pulse geeft. Dit verhoogt de totale tijd per particle natuurlijk gevoelig, zeker als er meerdere tussenstappen zitten in de berekeningen. We merkten echter dat alles nog meer dan snel genoeg werkte dus hebben we gekozen voor de minste complexiteit in het ontwerp, hoewel we perfect weten hoe we de andere stappen kunnen versnellen (zie afbeelding hieronder).



Figuur 3: door enkel de SDRAM operaties lang te laten duren krijgen we mogelijk een grote snelheidswinst.

5 Tekenmodule basis

Het werken met de state machines was een hele aanpassing van onze traditionele manier van werken. Na elke berekening moeten we immers altijd

minstens 1 clocktick (of in ons geval 1 state) wachten tot het resultaat klaar is, waardoor er vaak tussenstates moesten worden toegevoegd. Ook het slechts beschikbaar zijn van 1 particle per keer betekende dat we creatief moesten zijn met de beschikbare 32 of 64 bits om de noodzakelijke informatie bij te houden. Met verilog is dit echter makkelijk op te lossen door DEFINES te gebruiken, waardoor we snel de bitranges voor de verschillende variabelen konden aanpassen.

Ondanks dat we zeiden dat het RAM de beperkende factor was qua snelheid en parallellisme besluiten we hier om het geheel nog trager te maken. Dit is echter vooral omdat het einddoel van het project dit perfect toeliet. Teveel particles tegelijk simuleren of ze simuleren tegen een heel hoge snelheid levert geen estetische resultaten op, wat toch uiteindelijk het doel is. In de praktijk is zelfs 100-150 clockticks nog veel te snel, dus vertragen we de subclock vaak nog (wat zelfs mogelijk is via de GUI, zie later) om een mooier kunstig resultaat te krijgen. Dit is niet het geval in andere toepassingen die zeer snel moeten werken, daarom hebben we ook de alternatieven besproken en methodes om het systeem gevoelig te versnellen mocht dit nodig zijn.

6 Art Generators

In dit project zijn er drie verschillende art generators ontwikkeld: Path Following, Substrate en BoxFitting. Beide zijn ze geïmplementeerd op basis van een state machine, maar de invulling van de particles en de behaviour ervan wordt voor elke generator apart gedefiniëerd.

6.1 Path Following

De eerste generator maakt gebruik van het lijn teken algoritme. Er worden random een aantal particles gegenereerd over het scherm. Deze particles zullen ofwel een pad volgen naar andere particles ofwel een pad volgen richting de muis.

6.2 Substrate

In de tweede generator, namelijk de Substrate, worden ook random particles doorheen het scherm gegenereerd van waaruit lijnen vertrekken. Op de lijnen worden loodrechte lijnen getekend met een kleur en een random lengte die uitfaden naar de background color. Deze kleur kan ofwel random worden gekozen ofwel op basis van een input image. Het is een uitgebreide state machine omdat er veel gebruik wordt maakt van verschillende tekenmodule's. Zo moet het zowel lezen en schrijven van RAM, image kleuren uitlezen, hoeken van de lijnen bepalen met een sinus en cosinus lookup table, alsook de gewone lijnen en loodrechte lijnen afhandelen. Een uitbreiding is mogelijk

door ook zogezegde POI's (Point Of Interest) bij te houden. Deze kunnen dan nieuwe startpunten zijn van nieuwe particles als andere particles doodgaan door te botsen. Deze POI's worden bijvoorbeeld op een lijn gekozen zodat nieuwe particles altijd loodrecht starten van al bestaande lijnen in de RAM. Door de keuze van een goed kleurenschema is het mogelijk om Art te maken die lijkt op de top view van een stad. Een voorbeeld is gegeven in Figuur 4.

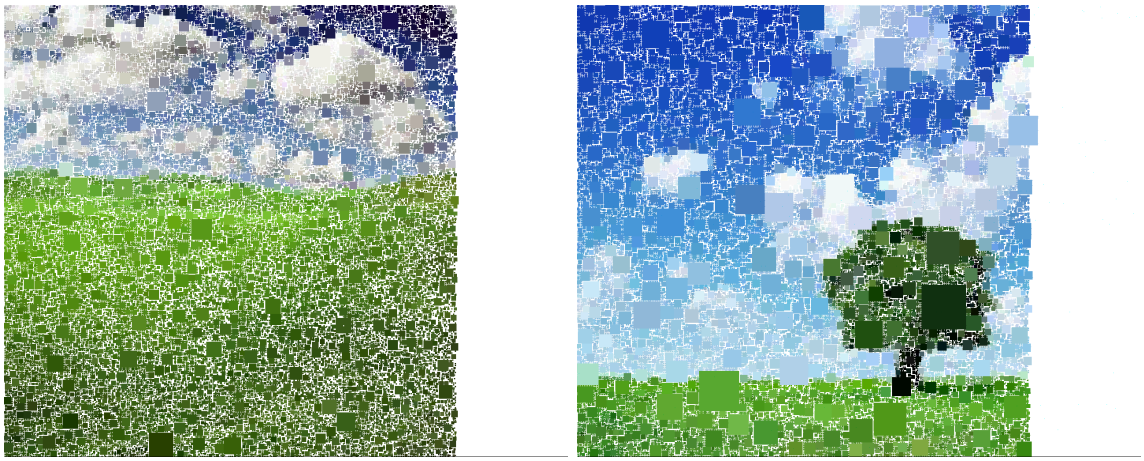
Figuur 4: Voorbeeld Substrate.

6.3 Box Fitting

Bij Box Fitting worden de particles voorgesteld als vierkanten. De positie ervan wordt random bepaald op een positie waar nog niet getekend is. Elke tijdstap groeien de boxen totdat ze botsen met andere. Ook hier wordt de kleur weer bepaald door een input image. Een uitbreiding zou bijvoorbeeld zijn dat de afbeelding gecaptured wordt van de aangesloten camera. Deze generator maakt gebruik van de tekenmodule *drawRectangleBorder.v*. Deze krijgt een grootte en dikte mee en zal enkel een border van het vierkant tekenen met die afmetingen. De dikte van de edges van de particles kan ingesteld worden met de switches, dit maakt het mogelijk interactief de generator aan te passen. Twee voorbeelden van deze generator vindt u terug in Figuur 5.

7 GUI

De GUI bestaat uit een aantal gekleurde knoppen die gedefinieerd zitten in de module. Elke knop krijgt een unieke code die wordt uitgestuurd zolang



(a) (b)

Figuur 5: Voorbeeld Boxfitting

er op geklikt wordt. Met een simpele debouncing wordt er maar 1 klik uitgevoerd. Naast de knoppen wordt er ook de muiscursor weergeven. Deze input komt van de ps2 controller die de muiscoordinaten en staat van de knoppen bijhoudt. Om de GUI te kunnen weergeven hebben we deze in de VGA controller geplaatst. De VGA controller stuurt de huidige pixel door naar de GUI en deze geeft de pixel terug, al dan niet aangepast. Hierdoor komen de knoppen en de cursor niet in het RAM en moeten we er dus ook niet op letten. Zo kan de GUI ook makkelijk afgezet worden.

8 PS2 muis

Omdat we tot nu toe alle functies aan de switches aan het koppelen waren, en dit er na een tijd veel werden om te onthouden besloten we ook maar een muis te interfacen. Deze konden we dan ook als input gebruiken voor bepaalde generators. Hiervoor hebben we een standaard ps2 muis gebruikt. Om deze te gebruiken moesten we dus eerst de mogelijkheid hebben om data te versturen en te ontvangen via ps2.

Deze communicatie verloopt via de ps2_clock en ps2_data. Dit zijn beide tristate pinnen, zodat ze zowel voor zenden als ontvangen gebruikt kunnen worden. Als we data willen ontvangen zetten ze beide pinnen op 1'bZ. Hierdoor zet de muis zijn eigen klok op de ps2_clock pin en op basis van die klok stuurt de muis zijn data. Als we zelf data willen sturen genereren we een klok en zetten deze zelf op de ps2_clock en sturen onze data op basis van deze klok. Om zelf niet met deze low level signalen bezig te zijn hebben we de UP block van Altera zelf gebruikt: *Altera_UP_PS2*. Deze neemt gewoon de data en clock van de ps2 poort en kan dan data versturen en ontvangen.

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 1	Y overflow	X overflow	Y sign	X sign	1	Middle	Right	Left
Byte 2	X Movement							
Byte 3	Y Movement							

Tabel 1: PS2 protocol

Als de module start moet er eerst gewacht worden tot de muis zelf opgestart is. Dan sturen we 8'hF4, wat de muis laat starten met zijn positie en knop updates te sturen. Een extra commando dat we ook nog sturen is 8'hE7, waardoor er een 2:1 scaling wordt ingesteld. Dus voor elke pixel dat de muis beweegt, beweegt de cursor op het scherm 2 pixels.

Eens we de data van de muis konden ontvangen moesten we deze nog goed interpreteren. Om te starten zijn we begonnen met het ps2 protocol te bestuderen op `http://www.computer-engineering.org/ps2mouse/`. Elke volledige update van de muis bestaat uit 3 bytes die er uitzien zoals in tabel 1.

We wachten totdat we deze 3 bytes ontvangen hebben en updaten dan de muis status in 1 keer, zodat we geen artefacten krijgen met eerst de x positie en daarna pas de y positie te updaten. Ook houden we hier rekening met de randen van het scherm zodat we daar niet buiten gaan. De ps2_mouse module houdt zelf de positie en button state bij, zodat andere modules deze gewoon kunnen uitlezen, zoals de GUI en bepaalde generators.

9 Ethernet

Omdat foto's nemen van een scherm nooit een goede kwaliteit geeft en we onze resultaten toch in redelijke kwaliteit wouden opslaan zochten we een manier om deze digitaal door te sturen. Ons eerste idee was om dit via de seriële poort te doen, maar aangezien geen van onze laptops deze aansluiting nog had was dit niet erg praktisch. De 2de optie was om dit via de ethernet poort te doen.

De ethernet controller op de DE2-70 is de dm9000a. Deze bezit zelf een MAC laag, waar we als eerste het MAC adres op instellen. Dan kunnen we nog kiezen voor promiscuous mode of niet. Als deze modus opstaat ontvangen we alle data die op de poort toekomt, als deze afstaat ontvangen we enkel data doe naar ons eigen MAC adres gestuurd is. Ook voor deze interface hebben we een controller gevonden, die full duplex communicatie voorziet. De controller interfacet met de buitenwereld via 2 fifo's. 1 voor de te versturen data en 1 waar de ontvangen data in komt. Daarnaast moeten de ethernet pinnen ook doorverbonden worden met deze controller. Omdat we de ontvangst functie niet gebruiken in ons project laten we de read flag

0	1	2	3	4	5	6	7	8	9	10	11	12	13
Destination Address						Source Address						Ethertype	

Tabel 2: Ethernet header

0	1	2	3
Version + Header length	Services	Total length	
Id		Flags + offset	
TTL	Protocol	Header Checksum	
Source IP			
Destination IP			

Tabel 3: IP header

van de receive buffer gewoon altijd op 1 staan, zodat deze nooit vol geraakt.

Om data te sturen moeten we hier de juiste headers voor plaatsen zodat de data op de juiste plek aankomt. De data die we moeten doorgeven aan de dm9000a controller begint met de totale lengte van het te versturen packet. Dit wordt niet expliciet verstuurd maar hierdoor weet de controller tot waar het packet in de fifo reikt. Vervolgens moeten we de Ethernet (tabel 2), IP (tabel 3) en UDP header (tabel 4) versturen. De UDP header heeft ook een checksum maar deze zetten we gewoon altijd op 0x00 (disabled).

Om niet telkens deze headers voor onze data te moeten zetten hebben we hierdoor een *udp* module gemaakt. Deze gedraagt zich heel gelijkend aan de dm9000a controller. Echter omdat we enkel data moeten kunnen verzenden heeft deze maar 1 fifo. Als deze module genoeg data in deze fifo heeft begint hij de headers te versturen en uiteindelijk ook de data. Daarna wacht hij opnieuw op voldoende data. Hier moeten we op 2 dingen goed letten: buffer underflow en buffer overflow. De module mag niets doorgeven aan de dm9000a controller als er te weinig data beschikbaar is en mag ook niet meer data aanvaarden als de fifo vol is. De dm9000a controller moet ook met beide situaties rekening houden. De snelheid waarmee de data wordt aangeleverd aan de *udp* module kan namelijk ook variëren door de snelheid

0	1	2	3
Source port		Destination port	
Length		Checksum	
Data			

Tabel 4: UDP header

in de gui aan te passen.

Er zitten wel nog een aantal bugs in de controller die we gebruikt hebben. Zo kan deze als er een packet ontvangen wordt volledig stoppen met zenden. Dit hebben we deels opgelost door promiscuous mode af te zetten zodat we enkel packets ontvangen die effectief naar ons verstuurd zijn, wat heel wat minder kans is om vast te lopen. Ook als we een reset uitvoeren loopt de controller vast, waardoor we de FPGA moeten herprogrammeren voordat deze weer werkt. Ook met de data volgorde moet opgelet worden. Aangezien de fifo in de dm9000a controller 16 bits binnenkrijgt en 8 bits uitgeeft, wordt de volgorde van deze 16 bits omgedraaid. 16'h1234 als input wordt dus verstuurd als 8'h34 gevolgd door 8'h12.

We hebben de FPGA een MAC en IP adres toegekend en broadcasten alle data. De source en destination poort zijn beide 2048. Hierdoor kan dus elke pc die op hetzelfde netwerk als de FPGA hangt en luistert op deze poort de data ontvangen. Zelf hebben we ook een kleine client geschreven die de netwerk data leest en de afbeelding die erin zit opnieuw opbouwt.

- Source MAC: 48'h010203040506
- Destination MAC: 48'hFFFFFFFFFFFF
- Source IP: 32'hC0A80309 (192.168.3.9)
- Destination IP: 32'hFFFFFFFF (255.255.255.255)
- Source port: 16'h4000
- Destination port: 16'h4000

10 Praktische info

10.1 Resources

In tabel 5 volgt een samenvatting van de gebruikte resources per tekenmodule.

10.2 Gebruik

10.2.1 Art Generators

Om van tekenmodule te wisselen moet er in de code de juiste module uit commentaar gehaald worden en de andere in commentaar zetten. De mogelijke tekenmodules zijn hier followMouse, substrate en boxFitting2.

Resource	Boxfitting	FollowMouse	Substrate
Logische elementen	6290	6809	8811
Registers	2754	2833	3106
Pins	534	534	534
Memory bits	573184	573184	587520
Multipliers	0	0	0
Pll's	2	2	2

Tabel 5: Resource gebruik

10.2.2 GUI

De 2 knoppen linksboven versnellen of vertragen de modules. Dit slaat zowel op de tekenmodules als op het doorsturen via het netwerk. Hier moet wel opgepast worden dat op de snelste stand het RAM soms niet kan volgen en er dus rare artefacten kunnen optreden.De knop in het midden kan de tekenmodule pauzeren. Helemaal rechts in de hoek hebben we de reset knop, welke hetzelfde doet als iKEY[0]. Links daarvan staat de knop om de achtergrond kleur te veranderen en nog verder naar links staat de clear knop. Als laatste is er nog een knop om het netwerk te activeren (rechts beneden). Deze neemt de 2de read port van het RAM over van de actieve tekenmodule, dus moet er opgepast worden dat als de tekenmodule terug gestart wordt met de netwerk module nog ingeschakeld. Dan kan de tekenmodule namelijk niet lezen van het RAM en kunnen er rare artefacten optreden.

11 Conclusie

Voor dit project wilden we echt iets doen dat grote voordelen had omdat het gedaan werd in hardware in plaats van software. Procedurele kunst leek ons heel geschikt omdat we verschillende elementen tegelijk zouden kunnen verwerken omdat ze onafhankelijk van elkaar dezelfde regels volgen en weinig tot geen input nodig hebben van elkaar. De enorme beperkende factor van het SDRAM zorgde er echter voor dat dit massieve parallellisme wel mogelijk was voor de berekeningen (enkel beperkt door het aantal logische elementen) maar niet om de beeldbuffer te updaten, vooral door het random karakter van de kunstwerken.

Het SDRAM heeft bijgevolg een grote invloed gehad op het design van de rest van het project. De modules spreken rechtstreeks 1 read/write poort aan voor random access en we gebruiken vertraagde state machines om het geheugen niet te overbelasten. Veel van deze elementen zijn conceptueel wel over te brengen naar een meer parallel systeem, maar zeker niet rechtstreeks

qua implementatie. We zijn er echter van overtuigd dat we, gegeven een andere SDRAM module met meer mogelijkheden, vrij snel een massieve versie met veel parallelle particles zouden kunnen bouwen, gebaseerd op de huidige code. De sequentiele aanpak die we nu hebben heeft ons veel meer inzicht gegeven in de werking van bestaande CPU's en doen ons inzien dat dingen als caching en block access zeer belangrijk zijn, omdat het RAM echt wel beperkend kan werken voor de snelheid van een programma. Deze kennis zal later nog van pas komen bij het ontwikkelen van traditionele applicaties.

Ookal hebben we het ganse systeem niet parallel kunnen opbouwen, toch hebben we veel geleerd van het omzetten van de bestaande Processing algoritmen (geinspireerd door www.complexification.net) naar hardware. Dit zal in de komende jaren meer en meer belangrijk worden naarmate we bewegen naar GPGPU systemen waarmee we software rechtstreeks op de GPU kunnen uitvoeren. Dit is toch altijd meer low-level dan op de CPU programmeren. Het programmeren in verilog en omzetten van de high-level algoritmen naar low-level beschrijvingen heeft ons betere inzichten gegeven in hoe de GPGPU implementaties kunnen worden aangepakt.

Maar het RAM en snelheid/parallellisme is natuurlijk niet het enige onderdeel van dit project. We hebben nog heel wat extra dingen toegepast die niet in de les waren besproken en die soms zeker even onvoordehandliggend waren. Zo hebben we het ps/2 protocol moeten doorgronden voor communciatie met de muis en zorgen voor een werkende UDP implementatie bovenop een ethernet stack. Deze twee onderdelen waren erg technisch gericht, terwijl de andere modules vaak redelijk high-level konden worden aangepakt. Dit gaf ons de kans om te zien hoe de low-level protocols en standaarden zijn opgebouwd en hoe hardware controllers in moderne besturingssystemen hiermee om moeten gaan. We hebben ook een terugkeer moeten maken naar de cursussen computer graphics en elementaire wiskunde om de lijn-teken algoritmes werkende te krijgen. Hiervoor moesten we onder andere zorgen voor een werkende cosinus en sinus lookup table. We kozen voor de lookuptables en niet voor een live uitrekening van de waardes (wat ook perfect mogelijk is via gespecialiseerde algoritmen die we onderzocht hebben) omdat deze veel minder complex zijn om op te bouwen en de extra preciesie weinig meerwaarde heeft in een grotendeels random-gedreven kunstomgeving. Het implementeren van de tekenalgoritmes geeft ons meer inzicht in hoe grafische kaarten dit soort probleem kunnen aanpakken en hoe we met beperkte middelen toch goede resulaten kunnen bekomen door wat slimme optimalizaties in de algoritmes en denkwijzen.

Uiteindelijk zijn we zeer blij met het project. Ookal hebben we de originele technische doelen niet helemaal kunnen bereiken, we hebben bruikbare random art gemaakt op een FPGA, wat het einddoel was. Hiernaast heb-

ben we enorm veel bijgeleerd over hoe hardware werkt en wat de beperkende elementen in zulk een systeem zijn. We hebben ook heel wat elementen toegepast die niet gehandeld waren tijdens de colleges en tevens kennis uit andere vakken toegepast in een totaal nieuwe omgeving. We zijn trots op de kunstwerken die eruit gekomen zijn en de algemene uitkomst van het project.