


UNIVERSITEIT GENT

FACULTEIT INGENIEURSWETENSCHAPPEN

SoftWire

Efficiënte emulatie van 3D-hardware

door

Nicolas Capens

Promotor: Prof. dr. ir. Koen De Bosschere

Scriptie ingediend tot het behalen van de academische graad van
Burgerlijk ingenieur in de computerwetenschappen
Optie: ingebedde systemen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: Prof. dr. ir. Jan Van Campenhout
Onderzoeksgroep Parallelle Informatiesystemen

Academiejaar 2006-2007

SoftWire

Efficiënte emulatie van 3D-hardware

door

Nicolas Capens

Promotor: Prof. dr. ir. Koen De Bosschere

Scriptie ingediend tot het behalen van de academische graad van
Burgerlijk ingenieur in de computerwetenschappen
Optie: ingebedde systemen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: Prof. dr. ir. Jan Van Campenhout
Onderzoeksgroep Parallelle Informatiesystemen

Academiejaar 2006-2007

Voorwoord – Dankwoord

ein·de·lijk (bn.)
1 op het eind, ten slotte [vaak als uiting van ongeduld] => *uiteindelijk*

Eindelijk, mijn scriptie. Voor mij is het meer dan enkel een verplicht afstudeerwerk. Het is de versmelting van al waar ik mij de laatste jaren op intellectueel vlak voor ingezet heb. Naast het drukke lessenschema had ik ook een eigen agenda. Veel van mijn 'vrije tijd' spendeerde ik aan mijn passie voor computergrafiek. Daar kroop op momenten zodanig veel tijd in dat het zelfs een gevaar vormde voor mijn studies. Toch sta ik nu voor de eindstreep, en dat met een scriptie waarin ik mijn volledige zelf heb kunnen tonen. Een kans om te bewijzen dat de afgelopen jaren de moeite waard waren. Eindelijk.

Ik wens dan ook de personen te bedanken die rechtstreeks of onrechtstreeks hun steentje hebben bijgedragen aan de totstandkoming van deze scriptie. Mijn dank gaat uit naar professor Koen De Bosschere voor het ondersteunen van mijn ideeën, alle proffen en assistenten die tijdens mijn opleiding meer deden dan enkel hun job, de collega's van TransGaming Technologies voor het professionele kader om mijn projecten uit te werken, familie en vrienden dichtbij en veraf voor hun motivatie, en in het bijzonder mijn vriendin Sarah voor haar liefdevolle steun in tijden die voor ons beide niet makkelijk waren.

Wanneer het in de scriptie raar leest om "we" te zien staan in plaats van "ik", denk dan aan deze personen.

Toelating tot bruikleen

Overzicht

SoftWire: Efficiënte Emulatie van 3D-hardware

Door Nicolas Capens

Promotor: Prof. dr. ir. Koen De Bosschere

Scriptie ingediend tot het behalen van de academische graad van
Burgerlijk ingenieur in de computerwetenschappen
Optie: ingebedde systemen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: Prof. dr. ir. Jan Van Campenhout
Onderzoeksgroep Parallelle Informatiesystemen

Universiteit Gent
Faculteit Ingenieurswetenschappen
Academiejaar 2006-2007

Samenvatting

3D-weergave gebeurt doorgaans met de hulp van een grafische co-processor (GPU). In deze scriptie onderzoeken we de mogelijkheid om deze taak efficiënt door de CPU te laten uitvoeren. Dit is interessant om kosten te besparen, en software is ook flexibeler dan hardware.

Grafische hardware wordt gekenmerkt door een hoge efficiëntie. Zeer veel elementen, waaronder geometrie en beeldpunten, worden in parallelle pijplijnen verwerkt. Hierbij maakt men veelvuldig gebruik van krachtige vectorbewerkingen. Moderne GPU's zijn ook grotendeels programmeerbaar.

Hoewel de hardware over zeer veel functionaliteit beschikt verandert de configuratie of 'toestand' slechts zelden. Dit maakt het mogelijk om in een software-implementatie de code te optimaliseren voor de gebruikte toestanden. Hiervoor wordt de code tijdens de uitvoering gegenereerd en tot nieuwe functies gecompileerd.

Deze techniek, de dynamische specialisatie, wordt in deze scriptie uitgebreid met een hogere programmeertaal en interne optimalisaties. De hogere programmeertaal is in C++ geïmplementeerd en laat toe gebruik te maken van de vectorbewerkingen van de CPU. De syntaxis is geïnspireerd door de HLSL taal, gebruik voor het programmeren van de GPU.

De optimalisaties zijn specifiek afgestemd op de grafische verwerking. Vooral dode-code-eliminatie en sterktereducties met constanten zijn efficiënt en verwijderen praktisch alle overbodige bewerkingen. Optimalisaties waarover de programmeur goede controle

heeft worden niet door het raamwerk voor dynamische specialisatie uitgevoerd om de compilatietijd laag te houden.

In deze scriptie wordt ook de vergelijking gemaakt met gerelateerd werk. De raakpunten en verschilpunten met het nieuwe raamwerk worden belicht. Hiermee wordt aangetoond dat het geschikter is voor grafische verwerking dan andere bestaande projecten. Dit is vooral te danken aan interne specialisatie, interne codegeneratie, en de ondersteuning voor vectorbewerkingen.

Uit de meetresultaten met 3D-toepassingen blijkt dat de prestaties vergelijkbaar zijn met die van een geïntegreerde GPU. We concluderen dus dat CPU's in de nabije toekomst in staat zullen zijn de taken van de GPU over te nemen voor bepaalde budgetsystemen. Tot slot is er nog zeer veel potentieel om de prestaties verder te verbeteren.

Trefwoorden

3D-weergave, emulatie, dynamische codegeneratie, dynamische specialisatie, optimalisatie

Extended abstract

(volgende twee pagina's)

SoftWire

Efficient Emulation of 3D Hardware

Nicolas Capens

Supervisor: Prof. Koen De Bosschere

***Abstract*: This article shows how 3D rendering can be emulated efficiently by using dynamic specialization extended with a high-level language that supports vector operations, and SSA-based optimizations.**

***Keywords*: 3D rendering, emulation, dynamic code generation, dynamic specialization, optimization**

I. Introduction

Real-time 3D rendering is typically accelerated by graphics hardware. However, the processing power of CPUs continues to increase steadily, which makes us question whether hardware acceleration is still always necessary. Software 3D rendering could reduce the cost of certain systems. It's also easier and cheaper to upgrade than hardware.

Efficient software rendering can be achieved with SoftWire, a framework for run-time code generation. It creates new functions that are specialized for the operations required for every draw command. This way the overhead of naive emulation is avoided.

However, SoftWire uses an assembly programming syntax, with makes it hard to write code, optimize it, and maintain it well. In this work we focus on extending SoftWire with a high-level programming language and a set of efficient optimizations tailored for graphics processing.

II. Background

The floating-point processing power of today's most powerful CPU is comparable to that of a mid-range graphics processor (GPU). However, a GPU has parallel pipelines dedicated to graphics processing and is thus very efficient, while the CPU is fully generic. A naive emulation of 3D functionality on the CPU results in very low performance because of the abundance of control flow operations.

Graphics processing is characterized by performing the same operations on many elements like vertices and pixels. During a drawing command the configuration or 'state' of the graphics device does not change. Another important characteristic is that graphics operations mainly consist of vector arithmetic. Small programs called shaders are executed for each element.

The key to efficient emulation is to generate processing functions specialized for the graphics state during each drawing command. Only the operations required to be performed for each specific state are added to the newly created functions. By using the CPU's vector instructions the shaders can be translated to efficient code.

III. Design and Implementation

The enhanced framework for dynamic specialization is called SoftWire++. Figure 1 below illustrates its design:

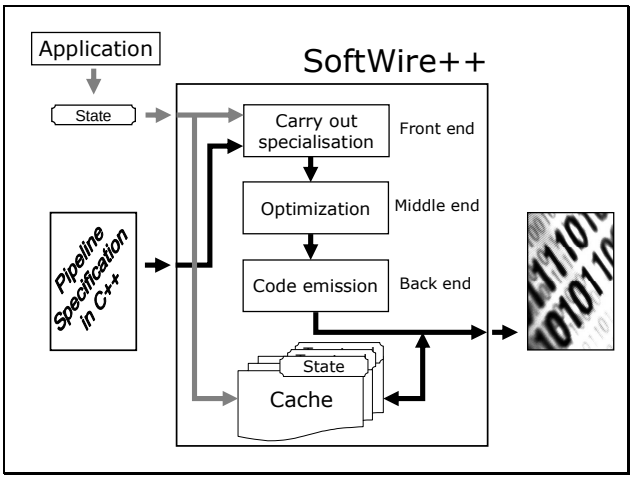


Figure 1: SoftWire++

The front end uses the state set by the application and the specification of the graphics pipeline written in C++, to create a list of operations to be performed. This is achieved by actually executing the C++ code and recording the operations that are being performed. C++ operator overloading and macros allowed creating a syntax very close to that of regular C.

By using classes such as `Float4`, which represents a vector with four components, it is possible to explicitly work with vector operations. The syntax of HLSL, a high-level language for shader programming, has been closely imitated.

In the middle end the list of operations is optimized. Two optimizations have been identified as particularly interesting for software graphics processing: dead code elimination, and strength reduction of operations with constants.

Dead code elimination allows simplifying the C++ specification by only having to control at the end of the processing pipeline which results are written to memory. Strength reduction is used to eliminate operations that use the default values of unused input parameters.

Both optimizations have been implemented using an intermediate representation in static single assignment (SSA) form. This ensures high efficiency and effectiveness [1].

Other generic optimizations are considered the task of the programmer who writes the pipeline specification. This way no time is wasted during run-time on optimizations that can be performed in advance.

In the back end of SoftWire++ the optimized list of operations in intermediate form is translated to CPU-specific instructions. Peephole optimizations clean up inefficiencies caused during the translation.

The cache stores newly generated functions and their corresponding graphics state for later reuse. This is crucial because generating the functions takes some time and applications use roughly a hundred states per frame. When the scene is rendered again the next frame, many of the states will be used again. If the state is found in the cache, the corresponding function is used instead of generating a new one.

IV. Related Work

Dynamic code generation and specialization are also used in other projects, albeit mostly for other purposes.

Historically, 'self-modifying code' was a popular way to reduce code size for machine code and assembly code programming. Later it was criticized because it makes code harder to maintain. In Henry Massalin's ground-breaking PhD thesis [2] it was revived as a means to optimize an OS kernel for the active configuration.

Code stitching is a form of dynamic specialization in which fragments of code are combined to form functions. It has been used successfully for a software 3D renderer but it is not flexible enough for programmable shaders.

Just-in-time compilation adaptively optimizes code at run-time. Although it can specialize for the CPU it runs on, it generally does not optimize for specific 'semi-constants'.

Projects which implicitly specialize for semi-constants exist, but are still mostly experimental. Their results vary because of the significant run-time overhead and because it's practically impossible to specialize for states consisting of many parameters.

Explicit specialization is more successful as the programmer generally knows what to specialize for. However, existing projects are less suited for graphics processing than SoftWire because they take external detours that cause longer compilation times and/or don't support vector operations.

V. Evaluation and Results

SoftWire is used in the SwiftShader software 3D rendering project, which is also developed by the author [3]. Performance was evaluated using the 3DMark benchmarks. Figure 2 summarizes the scores for a powerful GPU, a popular integrated GPU, an older generation integrated GPU (incapable of running the tests), SwiftShader using SoftWire and SoftWire++ on a Core 2 Duo CPU at 2.0 GHz, and an emulator used as hardware specification reference:

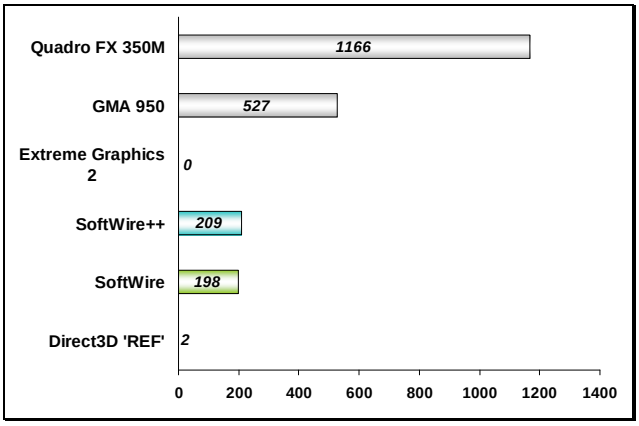


Figure 2: 3DMark05 scores

Figure 3 below shows a scene from the popular Half-Life 2 game running at low quality on the 2.0 GHz CPU. The game was generally playable.

Figure 3: Half-Life 2 at 19 frames per second

VI. Conclusions

Software 3D rendering could become a viable alternative to hardware rendering in the near future.

A high-level programming language with vector operations makes dynamic specialization for graphics processing easier to work with and retains performance.

Dead code elimination and strength reduction with constants in SSA-form are efficient and effective optimizations for software rendering.

References

[1] Keith D. Cooper and Linda Torczon, *Engineering a Compiler*, Morgan Kaufmann, 2003, ISBN 1-55860-698-X.

[2] Henry Massalin, *Synthesis: An Efficient Implementation of Fundamental Operating System Services*, PhD thesis, Department of Computer Science, Columbia University, 1992.

[3] SwiftShader software 3D renderer, TransGaming Technologies, http://www.transgaming.com/swiftshader.

Inhoudstafel

Tabel van afkortingen en symbolen

2D	tweedimensionaal
3D	driedimensionaal
ALU	*Arithmetic Logic Unit* – rekenkundige en logische eenheid
API	*Application Programming Interface* – applicatieprogrammeerinterface
CAD	*Computer-Aided Drawing (software)* – professioneel tekenpakket
CPU	*Central Processing Unit* – centrale verwerkingseenheid
FPU	*Floating-Point Unit* – verwerkingseenheid voor vlottende-kommagetallen
GFLOPS	*Giga FLoating point Operations Per Second* – miljard vlottende-kommabewerkingen per seconde
GPGPU	*General-Purpose computation on GPU's* – generieke verwerking op de GPU
GPU	*Graphics Processing Unit* – grafische verwerkingseenheid
HLSL	*High Level Shader Language* – hoog-niveau grafische verwerkingstaal
JIT	*Just-In-Time (compilation)* – dynamische compilatie
MP3	*MPeg-1 audio layer 3* – geluidsformaat
MMX	*MultiMedia eXtension* – x86 instructiesetuitbreiding
OS	*Operating System* – besturingssysteem
PC	*Personal Computer* – computer voor persoonlijk gebruik
ROP	*Raster OPeration (unit)* – rasterbewerkingseenheid
SFU	*Special Function Unit* – verwerkingseenheid voor transcendente functies
SIMD	*Single Instruction, Multiple Data* – vectorberekening
SISD	*Single Instruction, Single Data* – scalaire berekening
SSA	*Static Single Assignment (form)* – statische unieke-toewijzingsvorm
SSE	*Streaming SIMD Extension* – x86 instructiesetuitbreiding
x86	populaire instructieset van Intel voor CPU's

1
Inleiding

"*Question everything*"

— Maria Mitchell

1.1 Situering

Krachtigere CPU's openen nieuwe mogelijkheden. Voorbeelden hiervan zijn de opkomst van MP3, fotobewerking en recentelijk hoogresolutie video. In de huidige systemen wordt de CPU echter ook bijgestaan door een grafische co-processor (GPU). Deze is voornamelijk bedoeld voor 3D-weergave, een zeer rekenintensieve taak. Door een sterke specialisatie is hardware meer geschikt voor de uitvoering hiervan dan software. De rekenkracht van de CPU blijft echter gestaag groeien. We kunnen ons dus afvragen in welke mate huidige en toekomstige CPU's de grafische taken kunnen overnemen.

Om een eerste idee te krijgen van de verhoudingen van de rekenkracht, bekijken we de theoretische prestaties voor vlottende-kommabewerkingen. We vergelijken de laatste nieuwe CPU van Intel met een grafische kaart uit de nieuwste generatie van NVIDIA. De Intel Core 2 Extreme QX6800 levert tot 93,8 GFLOPS[1], terwijl de NVIDIA GeForce 8600 GTS theoretisch 92,8 GFLOPS haalt. We merken meteen op dat de CPU op dit vlak niet moet onderdoen voor een moderne GPU. De prestatiemaat voor vlottende-kommabewerkingen bepaalt echter slechts een deel van de nodige rekenkracht, en er bestaan ook krachtiger modellen van grafische kaarten (tot 475 GFLOPS op het moment van schrijven). Toch kunnen we verwachten dat de CPU in staat moet zijn bepaalde toepassingen met 3D-weergave zelfstandig uit te voeren.

Dit is echter niet vanzelfsprekend. Het efficiënte gebruik van de beschikbare rekenkracht is een grote uitdaging. Moderne GPU's hebben veel complexe functionaliteit die in de hardware ingebouwd zit. In software beschikken we enkel over de generieke instructies die de CPU ondersteunt. Met een naïeve aanpak (zoals bij klassieke emulators) worden de verschillende bewerkingen geïmplementeerd met veel controlestructuren. Dit leidt tot een grote inefficiëntie waardoor onvoldoende prestaties behaald worden voor vlotte interactiviteit. Er zullen dus extra inspanningen nodig zijn om bruikbare prestaties te behalen.

1.2 Motivatie

Waarom zouden we echter trachten de grafische taken door de CPU te laten uitvoeren, wanneer de GPU hier reeds voor instaat? De grafische hardware is immers specifiek ontworpen voor deze taken en we kunnen aannemen dat ze op alle vlakken hogere

[1] Giga FLoating point Operations Per Second

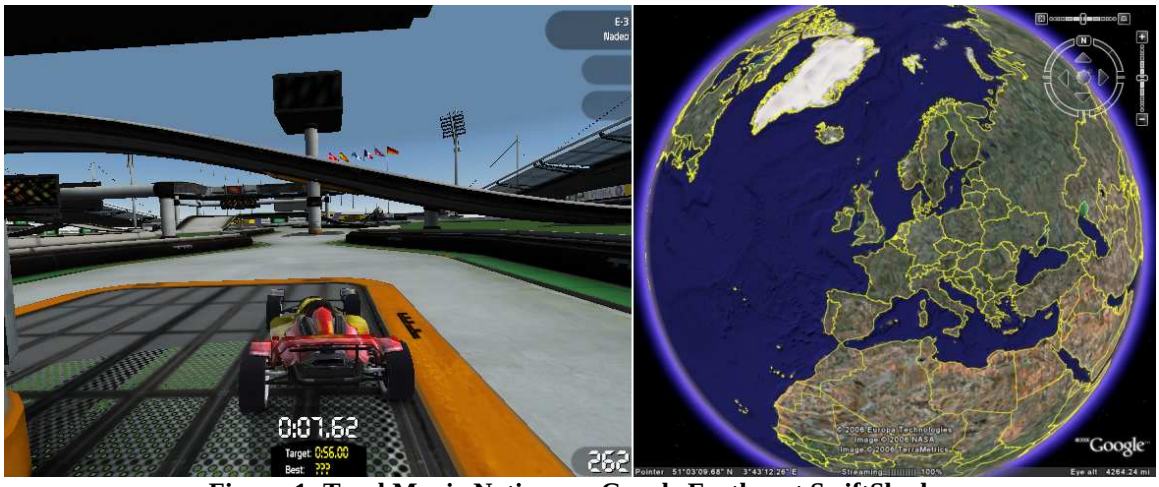
efficiëntie behaalt dan de generieke CPU. De maximale prestaties zijn niet te evenaren, en GPU's verbruiken verhoudingsgewijs ook minder stroom voor de geleverde prestaties. Toch zijn er verscheidene redenen om een softwarematige ondersteuning te wensen:

- Hardware heeft altijd een prijskaart. De ontwerpkost loopt in de ettelijke miljoenen, en de aankoopprijs van een grafische kaart varieert van enkele tientallen euro's tot honderden euro's. Software heeft een relatief lage ontwikkelkost en zeer lage kost per kopie.
- De prestaties van CPU's blijven voorlopig exponentieel stijgen dankzij het gebruik van meerdere kernen. Binnen afzienbare tijd overstijgen ze dan ook de praktische rekenkracht van elke GPU van dit moment.
- Met het voorgaande gaat ook een notie van "voldoende prestaties" gepaard. Er komen wel steeds zwaardere applicaties, en voor sommige gebruikers kan het nooit snel genoeg, maar voor een steeds groeiende verzameling applicaties zal de CPU in staat zijn de nodige prestaties te leveren.
- Software heeft het grote voordeel dat het eenvoudig op te waarderen is. Hardware opwaarderen is telkens een aanzienlijke investering, terwijl geschikte grafische software bij iedere applicatie meegeleverd kan worden. Voor veel systemen, in het bijzonder laptops, is opwaarderen van de hardware zelfs helemaal geen optie (tenzij het hele systeem).
- Software is ook uniform. Eén versie levert overal hetzelfde resultaat (op de snelheid na). GPU's kunnen echter belangrijke verschillen vertonen omdat er veel modellen bestaan en de versies van de stuurprogramma's niet altijd even stabiel zijn. Software vereist dan ook minder ondersteuning.
- Met twee processors limiteert één altijd de prestaties van de andere. Door een krachtige CPU ook de 3D-functionaliteit te laten uitvoeren wordt de rekenkracht gebalanceerd en verdwijnt het knelpunt.
- Nieuwe compacte ingebedde systemen zoals mobiele telefoons, handcomputers en navigatietoestellen zijn sterk door ruimte en kostprijs beperkt, zodat men genoodzaakt is het aantal chips te beperken. Functionaliteit zoals 3D-weergave kan soms enkel via software aangeboden worden.

Er zijn dus voldoende redenen om een softwarematige ondersteuning van grafische taken te verantwoorden. In deze scriptie onderzoeken we in hoeverre een CPU de taken van de GPU kan overnemen, en welke technieken hiervoor nodig zijn.

1.2.1 SwiftShader

Projecten die goede prestaties voor softwarematige 3D-weergave nastreven bestaan reeds. Met name SwiftShader [1] volgt de nieuwste grafische ontwikkelingen en implementeert ze in software. Het project vloeit voort uit eerder werk van de auteur van deze scriptie, en wordt verder ontwikkeld bij TransGaming Technologies. Het is tot vijftigmaal efficiënter dan een klassieke emulator zoals de *Direct3D Reference Rasterizer* van Microsoft. Het is in staat gelegenheidsspelletjes en kantoortoepassingen vlot weer te geven, waarvan in figuur 1 elk een voorbeeld:



Figuur 1: TrackMania Nations en Google Earth met SwiftShader

1.3 Dynamische Specialisatie

De hoge efficiëntie van SwiftShader is vooral te danken aan het gebruik van zogenaamde dynamische specialisatie. Deze techniek combineert specialisatie met dynamische codegeneratie.

Specialisatie is een optimalisatie waarbij de code afgestemd wordt op een specifieke variant van een bewerking. In plaats van generieke code te gebruiken die alle varianten van de bewerking aankan, gebruiken we voor elke variant afzonderlijk geoptimaliseerde code. Onderstaande C++ functies (fragment 1 en fragment 2) illustreren dit concept voor matrixvermenigvuldiging:

```
Matrix multiply(const Matrix &X, const Matrix &Y)
{
    Matrix R = Matrix::zeros(X.rows, Y.columns);

    for(int i = 0; i < R.rows; i++)
        for(int j = 0; j < R.columns; j++)
            for(int k = 0; k < X.columns; k++)
                R[i][j] += X[i][k] * Y[k][j];

    return R;
}
```

Fragment 1: Generieke code

```
Matrix multiply2x2(const Matrix &X, const Matrix &Y)
{
    Matrix R = Matrix(X.rows, Y.columns);

    R[0][0] = X[0][0] * Y[0][0] + X[0][1] * Y[1][0];
    R[0][1] = X[0][0] * Y[0][1] + X[0][1] * Y[1][1];
    R[1][0] = X[1][0] * Y[0][0] + X[1][1] * Y[1][0];
    R[1][1] = X[1][0] * Y[0][1] + X[1][1] * Y[1][1];

    return R;
}
```

Fragment 2: Gespecialiseerde code

De generieke versie (fragment 1) kan matrices van elke dimensie met elkaar vermenigvuldigen. Merk op dat de het grootste deel van de code uit controlestructuren bestaat. Deze voeren geen eigenlijke berekening uit en vormen dus 'ballast' die de uitvoering vertraagt. Indien we echter op voorhand weten dat de matrices van dimensie 2×2 zijn, kunnen we de functie uit fragment 2 gebruiken. Deze code voert louter de gewenste berekeningen uit en is dus veel sneller.

Specialisatie is een krachtige optimalisatie maar heeft ook beperkingen. Wanneer we optimaal met matrices van verscheidene dimensies willen werken zouden we voor elke combinatie van dimensies een gespecialiseerde functie moeten schrijven. Dit kan snel oplopen tot een onhandelbaar aantal functies. Men kan ze moeilijk allemaal manueel schrijven, en ze nemen veel geheugen in.

Een oplossing voor het manuele schrijven is het proces te automatiseren, bijvoorbeeld door gebruik te maken van C++ *templates* (sjablonen). Hierbij construeert de compiler zelf de nodige gespecialiseerde varianten. Men blijft echter met het probleem zitten dat al deze functies veel geheugenruimte innemen. Zoals we later in detail zullen zien zijn er zoveel specialisaties van grafische bewerkingen mogelijk dat ze onmogelijk in het geheugen passen. De oplossing is om de gespecialiseerde code pas te genereren wanneer ze nodig is, tijdens de uitvoering van de applicatie.

Dynamische codegeneratie is de veralgemeende term om nieuwe uitvoerbare code te genereren tijdens de uitvoering van de applicatie. Het biedt de oplossing voor bovenstaand geheugenprobleem van 'statische' specialisatie. Enkel de specialisaties die op het moment van uitvoering nodig blijken, moeten gegenereerd worden. Door dus code te genereren die enkel en alleen de gewenste bewerkingen bevat verkrijgen we 'dynamische' specialisatie.

Om dit te realiseren hebben we nood aan een raamwerk voor dynamische codegeneratie. Een dergelijk raamwerk bestaat uit een kleine compiler die door de applicatie kan aangeroepen worden. Bestaande raamwerken voor dynamische codegeneratie hebben echter hun beperkingen. Ze zijn niet volledig afgestemd op het genereren van code voor grafische bewerkingen, hebben beperkte optimalisaties of zijn moeilijk in gebruik. Hoewel dynamische specialisatie dus de sleutel vormt tot hoge efficiëntie is er nog grote vooruitgang te boeken op vlak van dynamische codegeneratie.

1.3.1 SoftWire

SwiftShader maakt gebruik van een raamwerk voor dynamische codegeneratie, dat afgeleid is van het SoftWire project [2]. Het is een project met open broncode, dat ook ontwikkeld werd door de auteur. De naam van dit raamwerk is afgeleid van *hard-wire*, wat staat voor het permanent verbinden van hardwarecomponenten. Bemerk de gelijkenis tussen specialisatie van hardware voor een specifieke taak via *hard-wiring*, en softwarematige specialisatie via *soft-wiring*.

Dynamisch code genereren met SoftWire gebeurt door functies aan te roepen die instructies toevoegen aan een geheugenbuffer. Door enkel die instructies toe te voegen die de nodige grafische bewerkingen implementeren, verkrijgt men in het geheugenbuffer een efficiënte nieuw gecreëerde verwerkingsfunctie. Deze functie kan dan aangeroepen worden om grafische data te verwerken [9].

De functies die de instructies genereren dragen de naam van de corresponderende assembleerinstructie van de CPU. Hoewel het C++-functies zijn is de abstracte ‘taal’ voor dynamische codegeneratie dus de assembleertaal. Hierdoor heeft men complete controle over elke instructie.

Toch is het niet eenvoudig om volledig optimale code te genereren voor elke specialisatie. De programmeur moet zelf rekening houden met alle aspecten om goede prestaties te verkrijgen. Bijvoorbeeld of een instructie al dan niet nodig is hangt doorgaans af van zeer veel parameters. Bovendien hebben deze parameters complexe afhankelijkheden. Dit zal ook duidelijk worden in het volgende hoofdstuk. Kleine onzorgvuldigheden of vereenvoudigingen van de programmeur leiden snel tot minder optimale code.

Naast de complexiteit om geoptimaliseerde code te genereren, heeft het programmeermodel van SoftWire het nadeel dat het moeilijk te beheren is. Assembleercode is lastig te lezen en schrijven, wat ook het opsporen en corrigeren van fouten bemoeilijkt. Bovendien zou men voor een andere processorarchitectuur alles moeten herschrijven, hoewel de grafische functionaliteit niet verandert.

We wensen dus op een hoger abstract niveau code te kunnen genereren die geschikt is voor grafische bewerkingen. Daarnaast moeten enkele optimalisaties door het raamwerk uitgevoerd worden, zodat de programmeur zich meer kan concentreren op het functionele en algoritmische aspect.

1.4 Doelstelling

Bovenstaande beschrijvingen geven een idee van de problemen die we in meer detail willen behandelen. Het zou ons echter te ver leiden om alle aspecten van softwarematige 3D-weergave gedetailleerd te bespreken. Voor enkele implementatieaspecten van de voorloper van SwiftShader verwijzen we de lezer naar [10]. We concentreren ons in deze scriptie voornamelijk op de bijdragen die we kunnen leveren door verbeterde dynamische specialisatie.

We merken ook op dat dynamische codegeneratie een op zich staand doel kan zijn. SoftWire wordt ook gebruikt door andere projecten die codegeneratie vergen. We zullen

ons in deze scriptie echter beperken tot dynamische codegeneratie in de context van 3D-weergave.

Concreet hebben we de volgende zaken tot doel:

- Het ontwikkelen van een hoog-niveau programmeertaal voor dynamische specialisatie in C++[1], geschikt voor grafische bewerkingen.
- De optimalisatie van de gegenereerde code en het minimaliseren van de compilatietijd.
- De efficiënte implementatie van een grafische bibliotheek met behulp van dynamische specialisatie.

1.5 Opbouw van de scriptie

Hoofdstuk 1 schetste met een minimum aan technische details de probleemstelling en het doel van deze scriptie.

Hoofdstuk 2 gaat dieper in op de theorie voor 3D-weergave, hoe de GPU functioneert, en de vertaling naar software.

Hoofdstuk 3 beschrijft het ontwerp van de verbeteringen aan SoftWire. We gaan ook in op enkele implementatieaspecten.

Hoofdstuk 4 biedt een overzicht van technieken en ideeën die we aantreffen in de literatuur en andere raamwerken voor dynamische specialisatie.

Hoofdstuk 5 evalueert het vernieuwde raamwerk en somt de meetbare resultaten op van de invloed op SwiftShader.

Hoofdstuk 6 concludeert de scriptie en sluit af met prospectieve opmerkingen en ideeën.

[1] C++ is de *de facto* standaard voor grafische toepassingen.

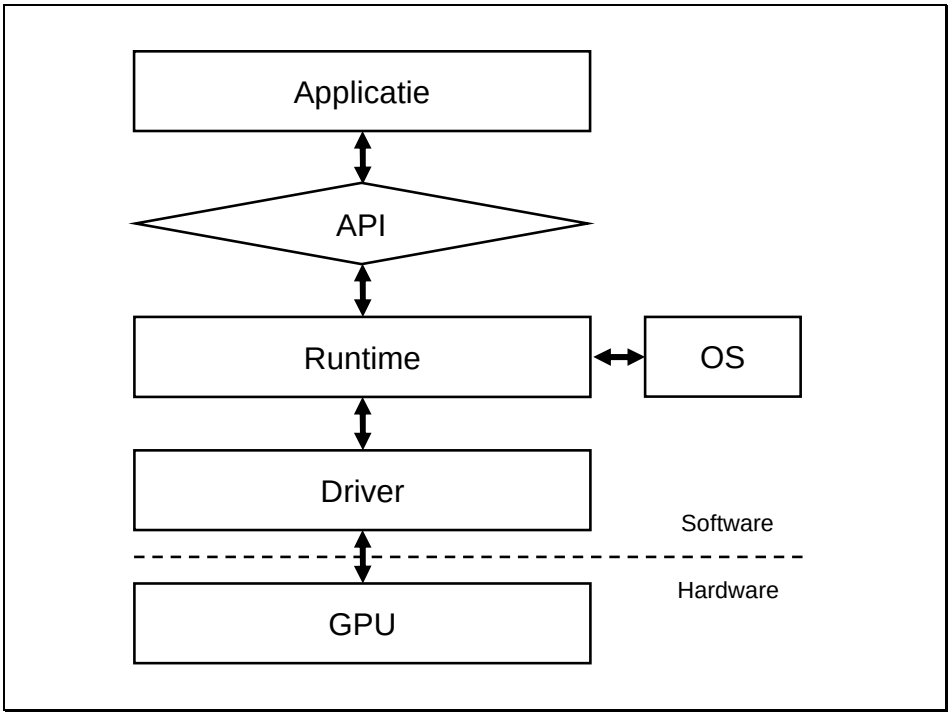
2
Achtergrond

"*Jack of all trades, master of none,*
though ofttimes better than master of one"

— Engels gezegde

2.1 Grafische architectuur

De weergave van een 3D-scène vereist een samenwerking tussen software- en hardwarecomponenten. Onderstaande figuur illustreert de algemene architectuur voor hardwarematige weergave als een lagenmodel:



Figuur 2: De grafische architectuur (GPU)

De applicatie bepaalt het creatieve aspect van de 3D-weergave. Het beheert wat getekend wordt, en hoe, maar doet zelf geen grafische berekeningen[1]. Alle functies voor het doorsturen van data naar de GPU, en de commando's voor de verwerking, verlopen via een applicatieprogrammeerinterface (API).

[1] Behalve geometrische berekeningen die niet ondersteund worden door de GPU. De nieuwste generatie kan echter praktisch alles.

Objecten tekenen gebeurt steeds in twee stappen: eerst wordt de ‘toestand’ ingesteld en data aangelegd, dan wordt een tekencommando gegeven. De toestand is de verzameling van alle parameters die het tekenproces beïnvloeden. Eens de gewenste toestand is ingesteld wordt het commando gegeven om de aangelegde data te verwerken volgens de gekozen parameters. De verwerking kan niet onderbroken of gewijzigd worden; het volgende commando wordt pas uitgevoerd wanneer het huidige afgelopen is.

Voor het tekenen van de hele scène worden deze stappen herhaald voor elk object. Daarna wordt het commando gegeven om het resultaat op het scherm te tonen. Dit levert dus één beeld op. Om interactiviteit te verkrijgen moeten meerdere beelden per seconde berekend worden.

De API schermt de applicatie af van de implementatie. Hoewel het niet veel meer is dan een lijst van functienamen speelt het een belangrijke rol. Het moet voldoende abstractie bieden zodat de applicatieprogrammeur zich vooral kan concentreren op het creatieve aspect, maar moet ook de mogelijkheden van de hardware beschikbaar stellen. Twee API’s zijn momenteel populair:

- Direct3D is ontwikkeld door Microsoft en dominant voor het Windows besturingssysteem. Het kent en snelle evolutie dankzij de sterk groeiende spelletjesindustrie, en is dan ook al aan versie 10 toe. Direct3D 10 speelt een belangrijke rol in de uniformisering van de grafische hardware door de minimumspecificaties zeer hoog te leggen en compatibiliteit met oudere hardware te verbreken. De meeste applicaties van dit moment gebruiken echter nog Direct3D 9.
- OpenGL staat voor *Open Graphics Library* en wordt op meerdere platforms ondersteund. Het was aanvankelijk bedoeld voor professionele werkstations en dus zeer krachtig. Het evolueerde echter trager dan Direct3D en speelt dus geen leidende rol meer.

Aangezien ze niet fundamenteel verschillen in functionaliteit gaan we in de rest van de scriptie uit van de Direct3D API. De werkwijze voor OpenGL is analoog.

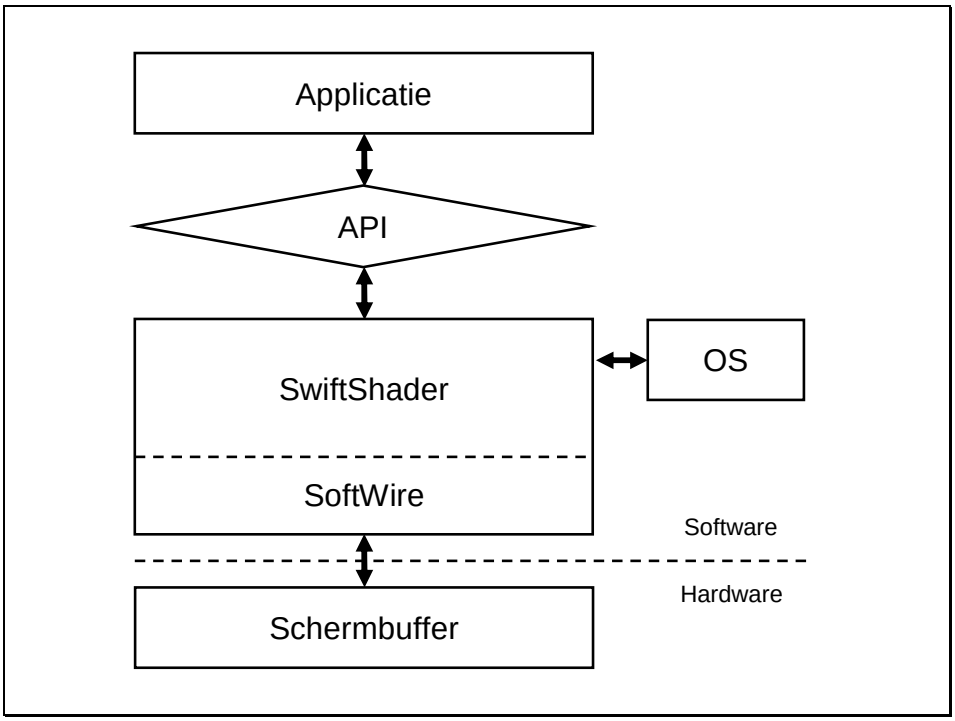
De *runtime* is de bibliotheek die door de applicatie ingeladen wordt om de API te kunnen gebruiken. De *runtime* zal doorgaans meerdere versies van de API ondersteunen. Verder staat het in voor de functionaliteit die niet hardwarespecifiek is. Bijvoorbeeld het beheren van de toestand is in eerste instantie de taak van de *runtime*. Het werkt ook samen met het besturingssysteem om meerdere grafische applicaties toe te laten.

De *driver* (stuurprogramma) is de software die instaat voor de correcte werking van een specifiek model van GPU. Het wordt dan ook meegeleverd door het bedrijf dat de GPU ontwerpt. De grafische commando’s, afkomstig van de *runtime*, worden in de *driver* geïmplementeerd voor de specifieke hardware. Hier zal men dan ook trachten enkele optimalisaties uit te voeren om de hardware niet onnodig te belasten.

De GPU voert de uiteindelijke grafische berekeningen uit en zorgt voor de uitvoer naar het scherm. Naast de grafische insteekkaarten, die over eigen geheugen beschikken, bestaat er ook geïntegreerde grafische hardware. Deze zit in de chipset van het moederbord en gebruikt het hoofdgeheugen. Om ze te onderscheiden van discrete GPU’s

worden ze aangeduid met IGP (*Integrated Graphics Processor*). De werking en het ontwerp van de grafische hardware komt verder in meer detail aan bod.

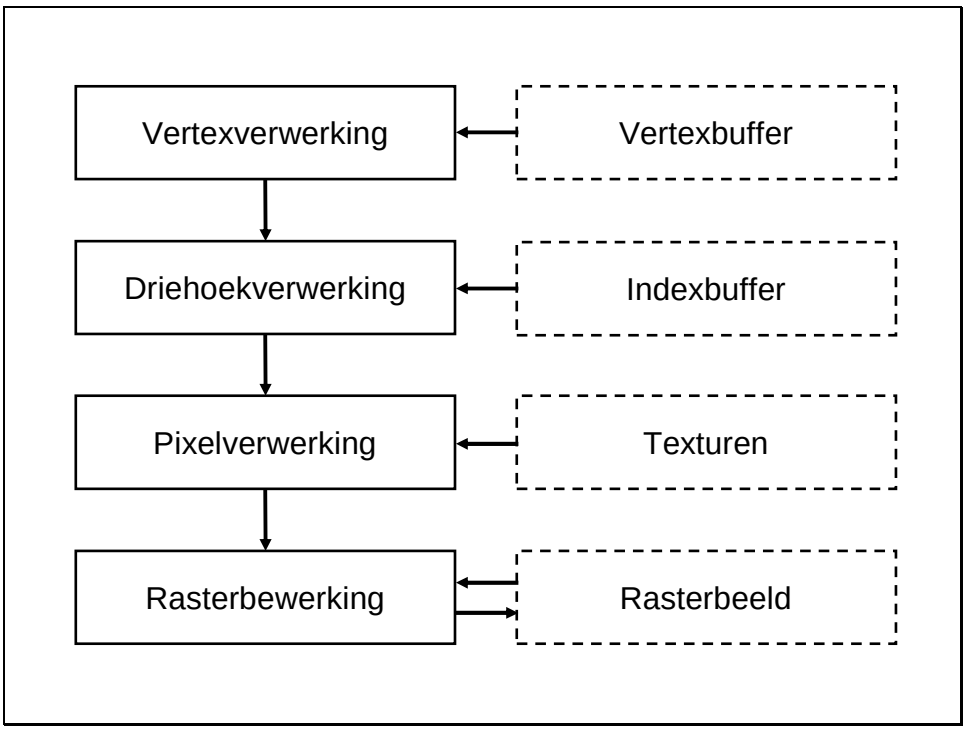
Bij softwarematige verwerking zoals met SwiftShader wordt de rol van de *runtime*, de *driver* en de GPU in één bibliotheek geplaatst (zie figuur 3). Zo heeft men volledige controle over het databeheer en de optimalisaties. Door het implementeren van de Direct3D API is men in staat bestaande applicaties die gebruik maken van deze API te draaien. Enkel voor de uitvoer van de berekende beelden naar het scherm is nog eenvoudige hardware nodig:



Figuur 3: De grafische architectuur (SwiftShader)

2.2 Grafische pijplijn

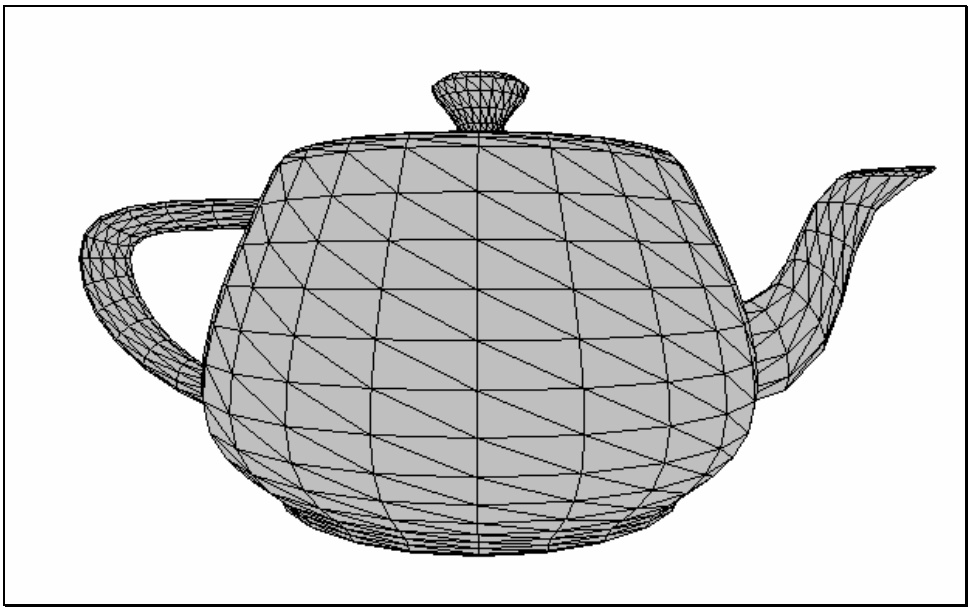
De grafische verwerking gebeurt in een aantal stappen die samen een pijplijn vormen. We zullen deze bestuderen om de soort bewerkingen en de karakteristieken van grafische berekeningen te achterhalen. De pijplijn kan opgesplitst worden in grote trappen die elk een pijplijn zijn van kleinere trappen. Op het hoogste niveau onderscheiden we volgende trappen en de voornaamste datastromen:



Figuur 4: De grafische pijplijn

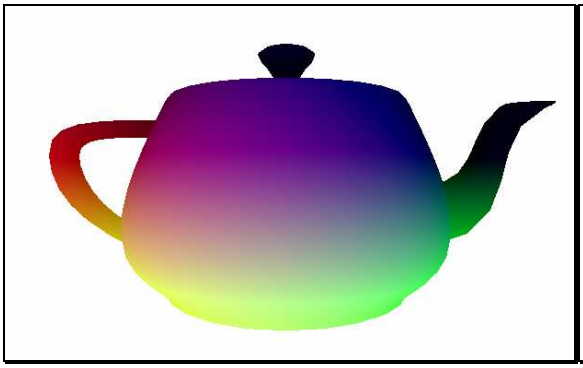
2.2.1 Vertexverwerking

Een 3D-scène wordt opgebouwd uit driehoeken. Met driehoeken kan elk object realistisch weergegeven worden. Ze zijn de eenvoudigste geometrische vorm met een oppervlak, en dus snel te verwerken. Onderstaande figuur 5 illustreert een theepot opgebouwd uit een groot aantal driehoeken om een relatief vloeiend oppervlak te vormen:

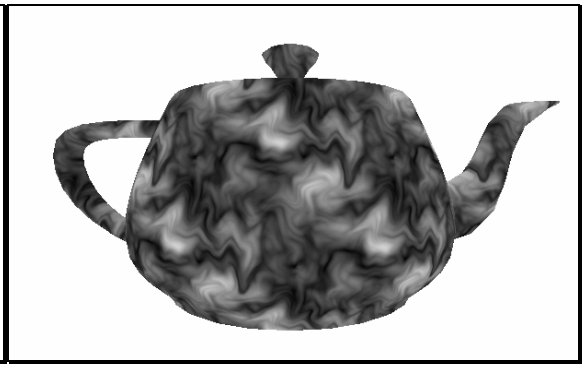


Figuur 5: Teapotahedron

Driehoeken zijn gedefinieerd aan de hand van hun hoekpunten, ook wel vertices genaamd (enkelvoud vertex). In de computergrafiek hebben vertices niet enkel een positie, ze krijgen ze ook attributen mee zoals kleur en textuurcoördinaten. Deze attributen worden geïnterpoleerd over de driehoek. Onderstaande figuren figuur 6 en figuur 7 illustreren wat hiermee bereikt kan worden:



Figuur 6: Kleur



Figuur 7: Textuur

Vertexverwerking omvat drie taken:

- Positie berekenen: De posities van de vertices van een object zijn opgeslagen in een vertexbuffer, maar moeten nog getransformeerd worden ten opzichte van de camerapositie. Dit gebeurt met behulp van matrixtransformaties.
- Belichting: De kleur van een vertex wordt bepaald door de belichtingsformule. De berekening is afhankelijk van materiaalparameters, lichtparameters, en de camerapositie. Figuur 6 toont een witte theepot met een rood, blauw en groen licht.
- Textuurcoördinaten berekenen: Deze zijn doorgaans ook reeds opgeslagen in de vertexbuffer van het object en rechtstreeks bruikbaar. Met matrixtransformaties kunnen echter enkele effecten verkregen worden zoals rollende tekst.

Op de precieze wijze van berekening wordt verder ingegaan wanneer we het ontwerp van een GPU nader bestuderen. We merken nu echter reeds op dat veel matrix- en vectorbewerkingen nodig zijn. Dit is niet verwonderlijk gezien de geometrische aanpak. De posities van de vertices worden opgeslagen als een vector met vier componenten: *(x, y, z, w)*. De *w*-component laat toe om ook translaties en projecties uit te voeren met een matrix van dimensie 4×4. De wiskunde erachter is hier minder van belang[1], het volstaat op te merken dat vectoren van vier elementen zeer typisch zijn.

Kleuren en textuurcoördinaten vormen ook vectoren. Kleuren worden opgeslagen als rode, groene en blauwe component. Een vierde component, doorgaans alfa genaamd, wordt gebruikt voor transparantie en andere effecten. Textuurcoördinaten hebben slechts twee componenten nodig voor 2D-texturen, maar er bestaan ook 3D-texturen en kubustexturen die drie componenten vereisen. Tot slot kunnen texturen ook geprojecteerd worden, wat net zoals bij de positievector een extra component vereist.

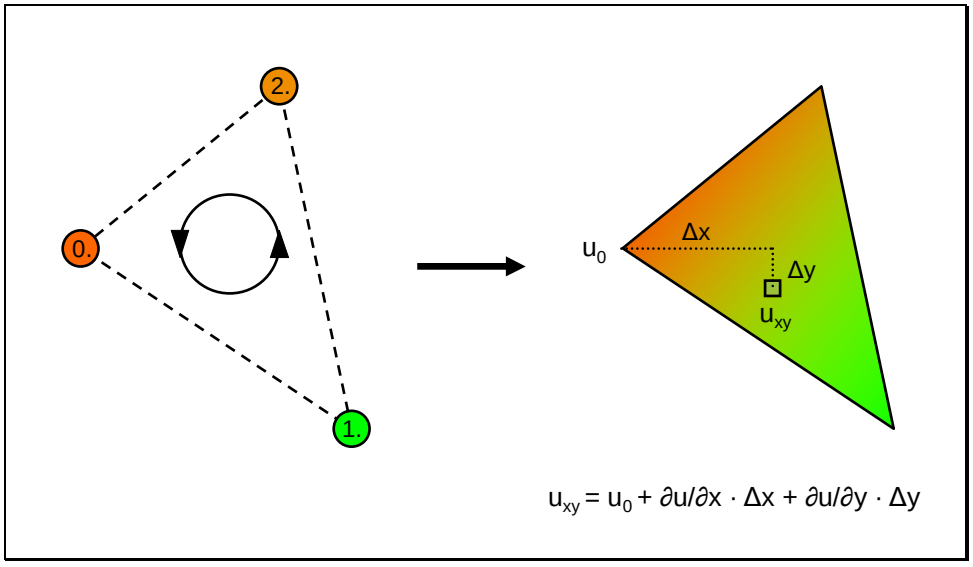
[1] Voor de volledigheid bevat Appendix A een inleiding tot homogene coördinaten.

2.2.2 Driehoekverwerking

Eens de vertices verwerkt zijn kunnen de driehoeken geassembleerd worden. Omdat vertices gedeeld kunnen worden tussen driehoeken, slaat men de indices van de vertices die de driehoeken vormen op in een index buffer. Het assembleren van de driehoeken vormt de eerste trap in de driehoekverwerking.

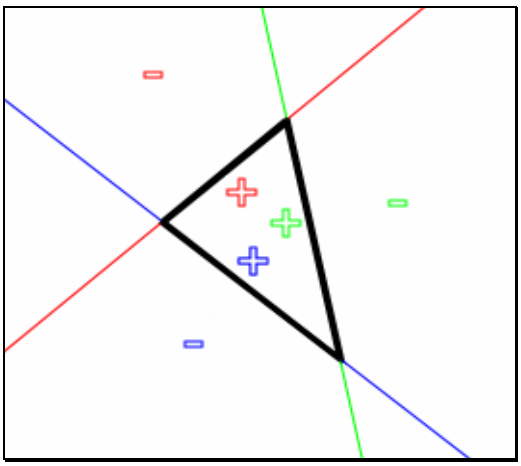
In de tweede trap wordt bepaald of de driehoek zichtbaar is. In de computergrafiek hebben driehoeken een voor- en achterzijde. Dit wordt gedefinieerd door de kurkentrekkerregel toe te passen, waarbij de volgorde van de vertices van belang is. Driehoeken die met hun achterzijde naar de camera gedraaid zijn worden uit de pijplijn verwijderd. Vervolgens wordt getest of de driehoek binnen het gezichtsveld van de camera ligt, en indien niet wordt hij ook verwijderd.

Om de kleur en textuurcoördinaten van de vertices te kunnen interpoleren over de driehoek dienen de gradiënten berekend te worden. Dit gebeurt in de derde trap van de driehoekverwerking. Een attribuut dat men interpoleert wordt een interpolant genoemd. De berekening van de gradiënten is gebaseerd op de parametervergelijking van een vlak door drie punten. Onderstaande figuur 8 illustreert de assemblage van een voorwaarts gerichte driehoek en de gradiënten voor een interpolant ‘u’:



Figuur 8: Voorzijdetest en gradiënten

De vierde trap in de driehoekverwerking is de zogenaamde rasterisatie. De taak van rasterisatie is de beeldpunten (*pixels*) te vinden die binnen de driehoek vallen. Dit gebeurt doorgaans door de parametervergelijking voor de drie zijden van de driehoek te bepalen. Wanneer een punt zich aan de positieve zijde bevindt van de drie zijden ligt het binnen de driehoek, zoals geïllustreerd in figuur 9:



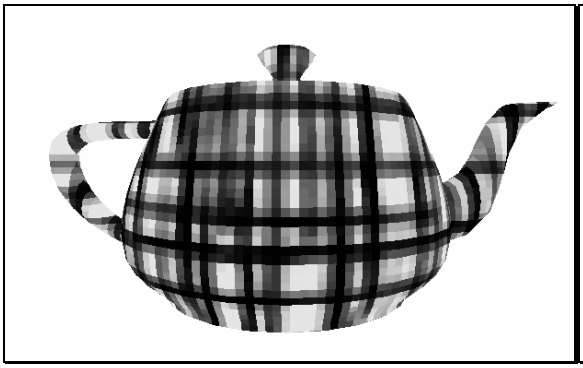
Figuur 9: Rasterisatie

We concluderen dat driehoekverwerking net zoals vertexverwerking uit geometrische berekeningen bestaat. De uitgevoerde bewerkingen tijdens vertexverwerking hangen echter af van een groot aantal toestandsparameters, terwijl ze bij driehoekverwerking volledig vastliggen. Dit zal ook een invloed hebben op de implementatie, waar we verder op terugkomen.

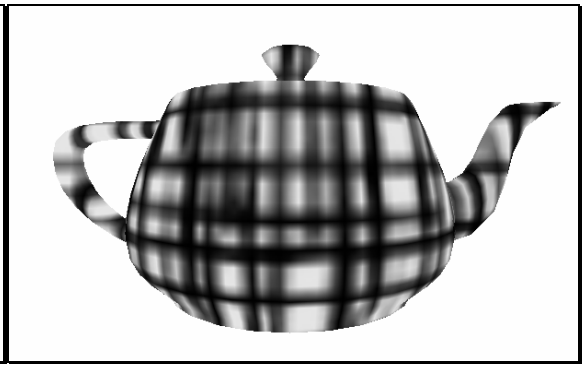
2.2.3 Pixelverwerking

De invulling van de driehoek gebeurt door elke pixel (beeldpunt) in de driehoek te verwerken. De eerste stap bestaat erin de waarde van de interpolanten te bepalen voor de huidige pixelpositie, zoals reeds geïllustreerd in figuur 8. Deze vormen de rechtstreekse invoer van de pixelverwerking.

Terwijl kleurinterpolanten onmiddellijk bruikbaar zijn om beelden zoals figuur 6 te produceren, dienen textuurcoördinaten om een kleur te bemonsteren uit een textuur. Hierbij heeft men keuze uit verschillende filters. Onderstaande figuren figuur 10 en figuur 11 illustreren puntfiltering en bilineaire filtering:



Figuur 10: Puntfilter



Figuur 11: Bilineaire filter

Puntfiltering bemonstert de meest nabijgelegen textuurkleur, terwijl bilineaire filtering de vier omringende kleuren interpoleert. Ook hogere orde filters zijn vaak beschikbaar. De functionaliteit van textuurbemonstering wordt dus sterk beïnvloed door bijkomende parameters.

Naast een aantal filtermethodes wordt ook een groot aantal verschillende textuurformaten ondersteund (één tot vier componenten van variërende kleurdiepte). Texturen kunnen daarenboven twee- of driedimensionaal zijn of een kubus vormen (zes zijden). Ze worden tegenwoordig voor veel meer gebruikt dan enkel afbeeldingen, en moeten eerder aanzien worden als algemene databuffers.

De interpolanten en textuurmonsters worden vervolgens gecombineerd tot een finale kleur voor de pixel. Ook hier bepalen toestandsparameters welke bewerkingen precies uitgevoerd worden. Voorbeelden van bewerkingen zijn optellen, vermenigvuldigen, de logaritme, het inwendig product, enz.

Naast de kleur wordt tijdens de pixelverwerking ook de nauwkeurige zichtbaarheid van de pixel berekend. Wanneer een pixel zich achter een ander object bevindt moet het uit de pijplijn verwijderd worden (tenzij bij transparantie en sommige andere effecten). Dit wordt gerealiseerd door voor elke pixel de kleinste afstand bij te houden. Deze waarden worden opgeslagen in de *z*-buffer, waarbij *z* de coördinaat is die in het scherm gaat en dus de loodrechte afstand tot het scherm vormt.

2.2.4 Rasterbewerking

Nadat de kleur en diepte (*z*) van de pixel berekend zijn, moeten deze waarden nog weggeschreven worden. Naast het eenvoudigweg overschrijven van de oude waarden in het rasterbeeld en de *z*-buffer kan men ook andere bewerkingen kiezen.

Volgens enkele voorgedefinieerde formules (te selecteren via een toestandsparameter) kan men de oude en nieuwe kleur van een pixel mengen. Zo kan men transparantie creëren of een zeer complexe berekening uitvoeren door meerdere ‘lagen’ over elkaar te leggen. Verder is het ook mogelijk om selectief de kleurcomponenten (rood, groen, blauw en/of alfa) weg te schrijven.

Voor de *z*-waarde wordt sowieso de oude waarde uitgelezen om de zichtbaarheid van de pixel te bepalen (tenzij dit volledig uitgeschakeld wordt). Hoewel men doorgaans enkel de pixels met *z*-waarde groter dan de oude in de *z*-buffer opgeslagen waarde uit de pijplijn wil verwijderen, heeft men ook de keuze tussen andere vergelijkingsoperators. Tot slot heeft men ook de optie om het wegschrijven van de nieuwe waarde uit te schakelen.

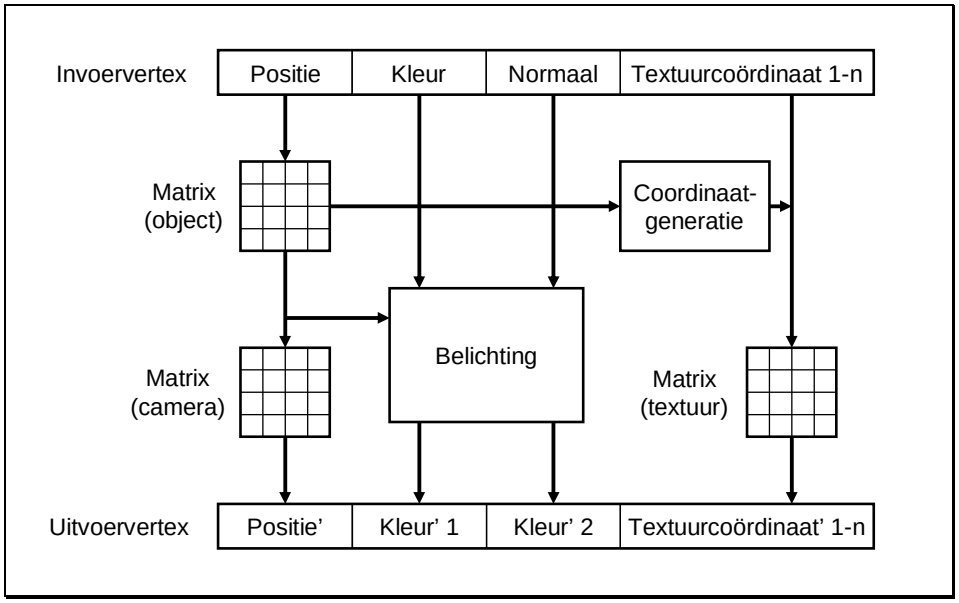
2.3 Hardware-implementatie

Tot nu toe was de grafische pijplijn een abstracte structuur die stap voor stap doorlopen werd. In hardware komt ze overeen met een echte pijplijn, waarbij op ieder moment in elke trap een bewerking gebeurt. Iedere klokcyclus schuift de data één elementaire pijplijntrap op, die typisch veel kleiner zijn dan de logische trappen.

Toch vormt het niet één doorlopende pijplijn zoals figuur 4 zou doen vermoeden. Eén vertex vormt immers geen driehoek, en één driehoek vormt geen pixel. De vier trappen op het hoogste niveau zijn echter wel pijplijnen op zich, waarbinnen op hetzelfde type data gewerkt wordt. Tussen deze pijplijnen worden buffermechanismen voorzien om elke pijplijn zo veel mogelijk ‘gevuld’ te houden.

2.3.1 Vertexverwerking

Het verwerken van vertices kent twee programmeermodellen; starre verwerking (*fixed-function processing*) en programmeerbare verwerking. In het eerste en oudste model zijn de gebruikte formules vooraf gedefinieerd. Met toestandsparameters kan de gewenste variant van de formule gekozen worden, maar daarbuiten heeft men geen vrijheden. Dit model werd aanvankelijk geïmplementeerd door voor elke taak afzonderlijke hardware te voorzien, zoals geïllustreerd in figuur 12:



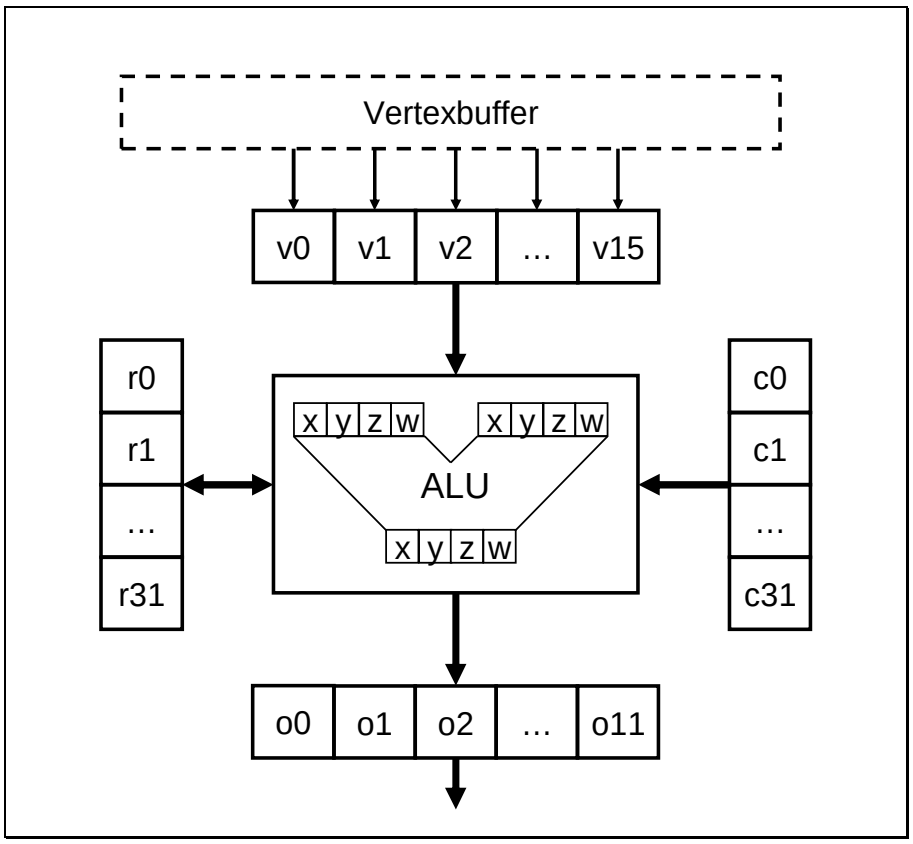
Figuur 12: Starre vertexverwerking

Dit toont slechts één mogelijke, geabstraheerde configuratie. Toch kunnen we hieruit enkele eigenschappen afleiden:

- Deze structuur is eenvoudig als een (groot) aantal pijplijntrappen in hardware uit te voeren.
- Iedere klokperiode wordt een aantal elementaire bewerkingen uitgevoerd, en dat voor iedere trap. Iedere klokperiode hebben we dus een afgewerkte vertex.
- De vaste bedrading en componenten leiden tot bovengenoemde beperking in verwerkingsformules.
- Wanneer bepaalde berekeningen uitgeschakeld zijn (bv. figuur 6 gebruikt helemaal geen textuurcoördinaten) blijft de hardwarecomponent hiervoor ongebruikt.

De twee laatste punten zijn grote nadelen. Wenste men meer mogelijkheden te bieden dan waren er meer componenten nodig, die vaker ongebruikt bleven. Dit was dus een verspilling van transistors. Een andere oplossing drong zich op.

Deze kwam er in de vorm van programmeerbare verwerking. Hierbij worden net zoals bij de CPU instructies uitgevoerd. Volgende figuur toont een typische architectuur voor programmeerbare vertexverwerking:



Figuur 13: Programmeerbare verwerking

Voor iedere vertex worden de componenten uit de vertex buffer gelezen en in de ‘v’ registers geplaatst. Deze vormen de primaire invoer van de verwerking. De ‘r’ registers zijn zowel lees- als schrijfbaar en worden gebruikt voor tijdelijke opslag van tussenresultaten. Variabelen die niet veranderen tijdens de uitvoer van een tekencommando worden in de ‘c’ registers opgeslagen en zijn dus enkel leesbaar. De finale resultaten worden weggeschreven in de ‘o’ registers. Al deze registers hebben vier componenten: xyzw of rgba afhankelijk van de semantiek (coördinaat of kleur).

De per vertex uitgevoerde programma’s worden door Direct3D *vertex shaders* genoemd, en de architectuur het *Shader Model*. De voornaamste instructies die de ALU en de instructiecontrole kunnen uitvoeren zijn:

Naam	Beschrijving	Type
`abs`	absolute waarde	rekenkundig
`add`	vectoren optellen	rekenkundig
`call`	subroutine aanroepen	controle
`crs`	kruisproduct	macro
`dp3`	inwendig product (xyz)	rekenkundig
`dp4`	inwendig product (xyzw)	rekenkundig
`exp`	exponentiële functie	rekenkundig
`frc`	Fractie	rekenkundig
`if`	test en conditionele sprong	controle

`log`	logaritmische functie	rekenkundig
`lrp`	lineaire interpolatie	macro
`m3x3`	matrixtransformatie (3×3)	macro
`m4x4`	matrixtransformatie (4×4)	macro
`mad`	vermenigvuldigen en optellen	rekenkundig
`max`	maximum	rekenkundig
`min`	minimum	rekenkundig
`mov`	kopiëren	rekenkundig
`mul`	vermenigvuldigen	rekenkundig
`nrm`	normaliseer vector	macro
`pow`	machtsfunctie	macro
`rcp`	omgekeerde (inverse)	rekenkundig
`ret`	einde van subroutine	controle
`rsq`	omgekeerde vierkantswortel	rekenkundig
`sge`	groter dan of gelijk	rekenkundig
`sincos`	sinus en cosinus	macro
`sub`	aftrekken	rekenkundig

Tabel 1: Shader Model 3.0 instructies

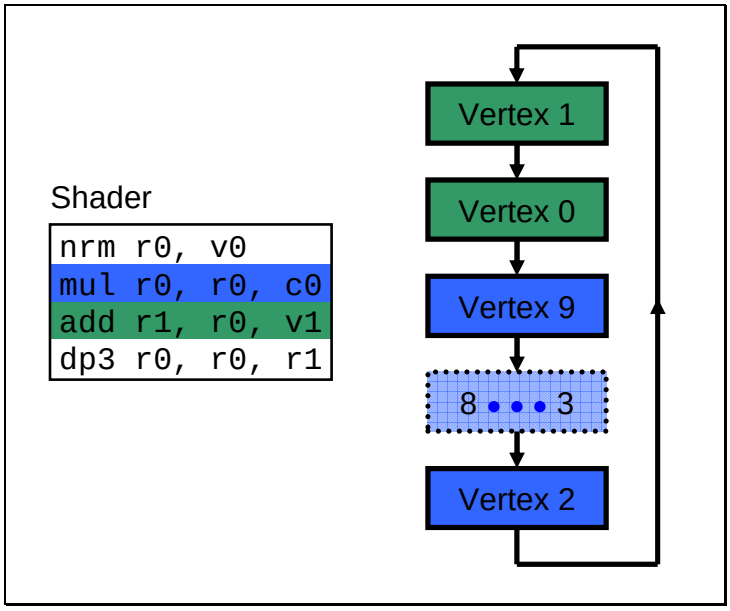
Instructies van het macrotype worden op hardwareniveau opgedeeld in kleinere instructies. Bijvoorbeeld `m4x4` bestaat uit vier `dp4` instructies. De `pow` instructie is geïmplementeerd met een `log`, `mul` en `exp` (dankzij de wiskundige gelijkheid $x^y \equiv 2^{y \cdot \log_2 x}$).

Met al deze instructies kan men dezelfde functionaliteit verkrijgen als de starre verwerking (en veel meer). Zo kunnen de bewerkingen die de matrices uit figuur 12 uitvoeren, verkregen worden met `m4x4` instructies. De parameters van de matrices zelf worden in de constante registers c0-c31 opgeslagen. Ze zijn dan ook deel van de toestand die de applicatie instelt vóór het tekencommando gegeven wordt.

Programmeerbare verwerking heeft echter een nadeel ten opzichte van starre verwerking: het duurt meerdere klokcycli om een vertex te verwerken. Er zijn immers meerdere instructies nodig om de verschillende taken uit te voeren. Dit is echter op te lossen door meerdere programmeerbare verwerkingseenheden in parallel te doen werken. Vertices zijn immers onafhankelijk van elkaar. De ALU's zijn ook relatief klein en goed benut. Grafische kaarten uit de Direct3D 9 generatie hadden tot 8 *vertex shader units*. Samen met een hoge kloksnelheid levert dit de gewenste prestaties.

Parallelle eenheden zijn niet de enige sleutel tot hoge prestaties. Merk op dat de instructies uit tabel 1 relatief complex zijn. Hun uitvoering duurt meer dan één klokcyclus. Bij een klassieke architectuur zou dit problematisch zijn gezien de afhankelijkheden tussen de instructies. Men zou vaak moeten wachten op het resultaat van een instructie voor de volgende gestart kan worden. Dit wordt echter opgelost door niet de volgende instructie te starten, maar dezelfde instructie, voor de volgende vertex. Vertices zijn immers niet alleen onafhankelijk van elkaar, er wordt dezelfde *shader* op uitgevoerd.

Concreet groepeert men vertices in een batch. Voor elke vertex in de batch voert men dan dezelfde instructie uit, alvorens naar de volgende instructie in de *shader* over te gaan. Indien de hardwarepijplijn van de ALU bijvoorbeeld 10 trappen lang is, heeft men een batch van 10 vertices nodig om alle instructielatentie op te vangen. Figuur 14 illustreert het principe:



Figuur 14: Batchverwerking

Merk op dat hierdoor wel de benodigde registerruimte ook vertienvoudigt. Bij moderne GPU's voorziet men echter minder registers per vertex dan het *Shader Model* voorschrijft. Veel *shaders* gebruiken ze toch niet allemaal. Wanneer het aantal voorziene registers toch overschreden wordt, verlaagt men het aantal vertices per batch zodat er meer registerruimte beschikbaar komt.

2.3.2 Driehoekverwerking

Zoals in het overzicht van de grafische pijplijn reeds opgemerkt werd bestaat de driehoekverwerking uit een aantal vaste bewerkingen. Hier is geen variatie gewenst, en de hardware-implementatie is dan ook een starre pijplijn[1]. Het voornaamste voordeel is dat iedere klokcyclus een afgewerkte driehoek de pijplijn verlaat[2]. Hier is geen parallellisatie meer nodig om voldoende prestaties te halen.

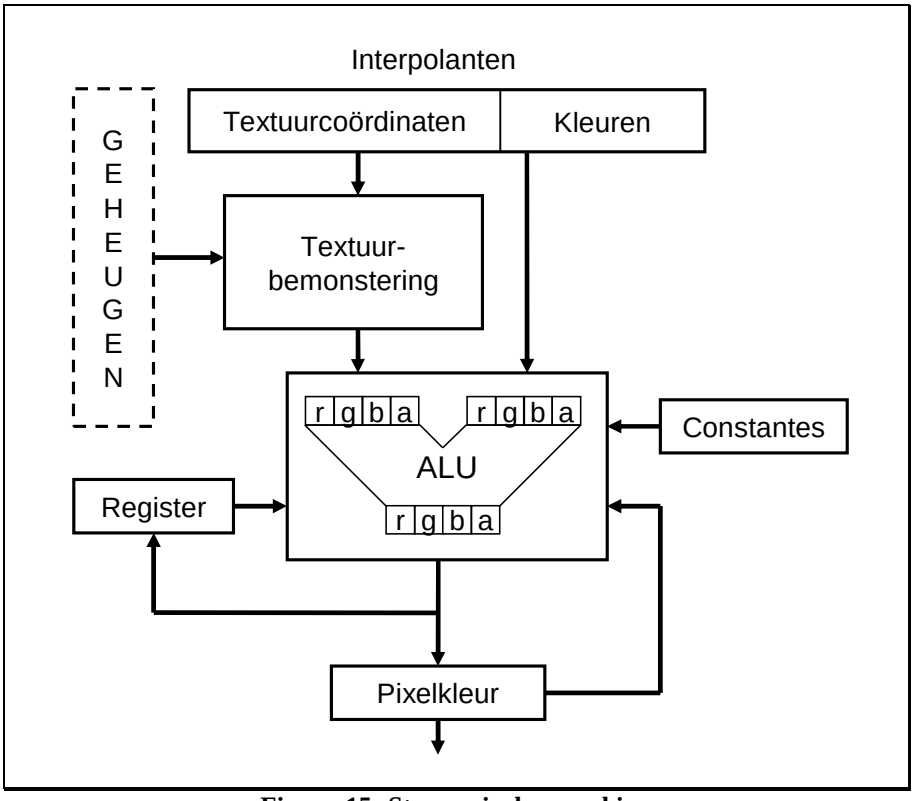
We kunnen wel nog opmerken dat in een hardwarepijplijn geen data kan verwijderd worden. Een driehoek die niet zichtbaar blijkt te zijn zal dus ook de rest van de pijplijn doorlopen.

[1] De X3000 IGP van Intel is een uitzondering en (her)gebruikt programmeerbare eenheden.
[2] Driehoeken met zeer veel interpolanten doorlopen de pijplijn meermaals.

2.3.3 Pixelverwerking

De grootste werklast van een GPU is de pixelverwerking. Een modern scherm bestaat immers uit circa één miljoen beeldpunten, terwijl een typische scène typisch veel minder vertices telt (en driehoeken - zie ook figuur 5).

Net zoals vertexverwerking kende pixelverwerking aanvankelijk enkel een starre implementatie. Onderstaande figuur toont een typische configuratie:



Figuur 15: Starre pixelverwerking

Hoewel we een ALU kunnen onderscheiden spreken we toch van starre verwerking. Er zijn immers slechts twee registers voor tijdelijke opslag. Erger nog, oudere modellen implementeerden er vaak slechts één (hier aangeduid met 'Pixelkleur'). Maar zelfs mét het extra register kunnen we moeilijk van programmeerbaarheid spreken. Er is maar beperkte vrijheid in het aantal formules (en dus effecten) dat kan verkregen worden met deze configuratie.

Volledige programmeerbaarheid vergt bovendien ondersteuning voor flexibele instructieopslag. Architecturen met starre pixelverwerking ondersteunden hoogstens acht bewerkingen per pixel. Bovenstaand ontwerp wordt ook vaak een *texture combiner* genoemd, omdat het in staat is tot acht textuurmonsters te 'combineren'. Iedere klokcyclus wordt een andere textuur bemonsterd met één van de geïnterpoleerde textuurcoördinaten[1].

[1] Bemerkt dat er maximaal acht textuurcoördinaten en texturen zijn, en maximaal acht bewerkingen.

Texturen bemonsteren is een complexe bewerking, en heeft bovendien een zeer hoge latentie. De texturen zijn immers opgeslagen in geheugen dat zich buiten de GPU bevindt. Enkele honderden klokcycli latentie, bovenop de latentie van de filtering en de ALU, is geen uitzondering. Dit probleem wordt opnieuw opgelost met *batching*. Ze zijn wel veel groter dan bij vertexverwerking.

Om de nodige prestaties te halen werden twee tactieken toegepast. De eerste was om twee texturen te bemonsteren en te combineren per pijplijn. Dit verdubbelt de pijplijnlengte. De tweede tactiek is om meerdere pijplijnen in parallel te plaatsen. Dit is efficiënter wanneer een oneven aantal texturen gebruikt wordt.

Volledige programmeerbaarheid was de volgende logische stap. De architectuur hiervan komt sterk overeen met die van programmeerbare vertexverwerking (figuur 13 en figuur 14). Met honderden pixels in een batch en 32 temporele registers vergt dit echter een gigantisch registerbestand. Men beperkt het aantal fysische registers per pixel tot een iets kleiner aantal, maar tracht er toch genoeg te voorzien om de batches zoveel mogelijk vol te houden.

Naast de instructies uit tabel 1 hebben *pixel shaders* nog één belangrijke toevoeging: `tex`. Deze instructie bemonstert een textuur op de opgegeven coördinaten. De coördinaten zelf zijn volledig aanpasbaar in de *shader*, wat veel meer creativiteit in de effecten toelaat.

Merk op dat de latentie van `tex` honderden klokcycli bedraagt, terwijl de latentie van rekenkundige bewerkingen een fractie daarvan is. De verhouding van bemonsteringsinstructies tot rekenkundige instructies is tegenwoordig typisch 1/3 of lager. Men heeft dus niet elke klokcyclus een textuurmonster nodig.

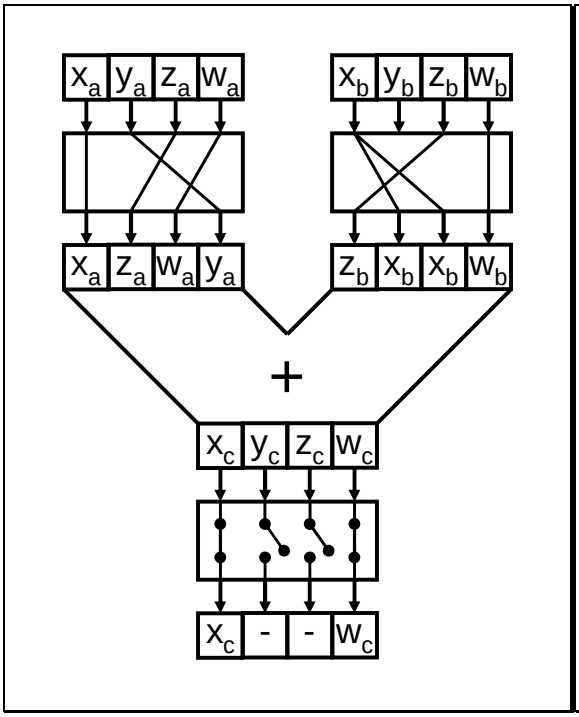
Moderne GPU's maken hier gebruik van om het probleem van het grote registerbestand effectiever op te lossen. Ze ontkoppelen de pijplijn voor bemonstering van de kortere pijplijn van de ALU. Zo lang er rekenkundige bewerkingen uitgevoerd moeten worden kan men op een kleine batch blijven werken. Wanneer men een `tex` instructie tegenkomt schakelt men over op de verwerking van de rekenkundige instructies voor andere batches[1]. Wanneer men terug bij de eerste batch aanbeland is het resultaat van de bemonstering (meestal) beschikbaar. Samen zijn deze batches kleiner dan die voor een architectuur zonder ontkoppelde bemonstering, en spaart men dus registers uit. Bovendien zijn er zo ook minder eenheden nodig voor textuurbemonstering dan *pixel shader units*. Daarnaast is het nu ook nuttig een cachegeheugen in te voeren om de latentie te verkleinen[2].

De aanpak met meerdere kleine batches heet *scheduling*, en vormde de basis voor een andere zeer recente evolutie: unificatie. Met een ontkoppelde pijplijn voor bemonstering houdt men enkel nog een ALU over die praktisch identiek is aan die voor vertexverwerking. Het was dus een logische stap om slechts één stel ALU's te voorzien die alle instructies kunnen uitvoeren. Dit heeft als grote voordeel dat de werklast van vertex- en pixelverwerking gebalanceerd kan worden via *scheduling*. Er kan dus geen bottleneck optreden. Er wordt op hardwareniveau geen onderscheid meer gemaakt tussen

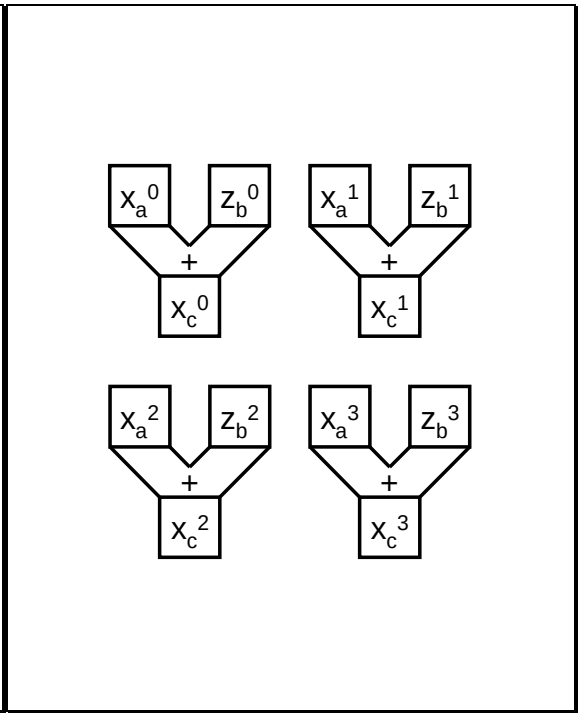
[1] De batches worden nu ook soms *threads* genoemd gezien de overeenkomsten met *multi-threading*.
[2] Caches voor texturen bestonden reeds eerder, maar enkel om de geheugenbandbreedte te beperken.

vertices en pixels, het zijn allemaal ‘elementen’. Merk ook op dat de vertexverwerking hiermee toegang krijgen tot de texturen, wat extra effecten toelaat.

Tot slot bekijken we nog een belangrijke verandering binnen de ALU:



Figuur 16: Vectoriële ALU



Figuur 17: Vier scalaire ALU’s

De linkerfiguur toont de eigenlijke structuur van een vectoriële ALU zoals ze in de GPU geïmplementeerd zijn, in een voorbeeldtoestand. Vóór de rekenkundige bewerking uitgevoerd wordt (hier een optelling), kunnen de vectorcomponenten ‘geklutst’ worden (*swizzle*). Na de bewerking, kunnen de componenten conditioneel weggeschreven worden (*mask*). In het voorbeeld wordt dus $x_c = x_a + z_b$ en $w_c = y_a + w_b$ uitgevoerd. De helft van de resultaten wordt weggegooid.

Deze inefficiëntie wordt opgelost met scalaire ALU’s. In plaats van vector per vector te verwerken, wordt component per component verwerkt. Dit verhoogt het aantal instructies, maar er kunnen vier scalaire ALU’s op het oppervlak van één vectoriële ALU zodat de rekenkracht minstens even hoog is. Merk op dat de componenten van vier verschillende pixels uit een batch komen. In het voorbeeld van de rechter figuur wordt de eerste component van de in de linker figuur geïllustreerde bewerking uitgevoerd. De volgende klokcyclus kan de berekening van de laatste component gestart worden. De gemaskeerde componenten y_c en z_c worden dus nooit berekend.

Concreet heeft de vectoriële ALU vier klokcycli nodig om deze instructie voor vier pixels uit te voeren, terwijl de vier scalaire ALU’s er slechts twee klokcycli over doen. Bovendien gebruiken de scalaire ALU’s minder registerruimte. Merk ook op dat omdat de vier scalaire ALU’s dezelfde bewerking uitvoeren de controle van het instructieverloop niet complexer is. Ten slotte heeft de scalaire versie geen extra hardware nodig voor *swizzle* en *mask.*

De GeForce 8800 GTX van NVIDIA heeft niet minder dan 128 zogenaamde *stream processors*, elk uitgerust met één scalaire ALU voor optelling+vermenigvuldiging en één voor transcendente functies (`exp`, `log`, `sin`, `cos`, `rcp` en `rsq`).

2.3.4 Rasterbewerking

Verwerkte pixels worden naar het beeldraster en de *z*-buffer geschreven door de ROP's (*Raster OPeration units*). Ze hebben nog drie andere functies:

- Pixelfragmenten combineren: Om kartelranden te vermijden worden vaak rasters gebruikt met hogere resolutie dan het schermraster. Pixels zijn dus opgedeeld in fragmenten die afzonderlijk verwerkt zijn en gefilterd (uitgemiddeld) moeten worden[1].
- Geheugen lezen: Om de kleur te kunnen mengen en de *z*-waarde te vergelijken moet de ROP eerst de oude waarden uit het geheugen lezen. Dit heeft (zoals elke textuurbemonstering) een hoge latentie, die opgevangen wordt met een lange pijplijn.
- Compressie: Om bandbreedte en geheugen te besparen zijn de waarden gecomprimeerd opgeslagen. Deze moeten gedecomprimeerd worden na het lezen en weer gecomprimeerd worden vóór het schrijven.

Net zoals bij de driehoekverwerking vormen de ROP's sterk gespecialiseerde hardware zonder programmeerbaarheid. Gezien de grote werklast aan pixels zijn het er echter wel meerdere in parallel.

2.4 Software-implementatie

2.4.1 Naïeve aanpak

De vertaling van hardware naar software kan op verschillende manieren. Een eerste, eenvoudige maar 'naïeve' aanpak is om elke hardwarecomponent op generieke wijze te emuleren, na te bootsen. De code voor elke component wordt dan sequentieel uitgevoerd door de CPU.

Een ALU wordt geïmplementeerd door eerst in de *shader*-instructie na te gaan welke argumenten ingeladen moeten worden. Deze worden dan uit het geheugen opgehaald. Daarna gaat men na welke bewerking uitgevoerd moet worden, waarna men naar de code gaat die ze effectief uitvoert op de ingeladen argumenten. Het berekende resultaat wordt dan weggeschreven naar de geheugenlocatie die overeenkomt met het doelargument in de instructie. Onderstaand codefragment illustreert deze aanpak voor de vertexverwerking uit figuur 13:

[1] OpenGL gebruikt de term *fragment program* in plaats van *pixel shader*, wat dus correcter is.

```
void executeALU(const ShaderInstruction &instruction)
{
    Float4 res, a0, a1, a2;   // Result and arguments

    // Read first source argument
    switch(instruction.src0.type)
    {
    case REG:   a0 = r[instruction.src0.index]; break;
    case CONST: a0 = c[instruction.src0.index]; break;
    case INPUT: a0 = v[instruction.src0.index]; break;
    }

    [...]   // Read second and third source argument

    // Perform operation
    switch(instruction.operation)
    {
    case ADD: res = a0 + a1;      break;
    case MUL: res = a0 * a1;      break;
    case MAD: res = a0 * a1 + a2; break;
    case [...]
    }

    // Write result to destination
    switch(instruction.dest.type)
    {
    case REG:    r[instruction.dest.index] = res; break;
    case OUTPUT: o[instruction.dest.index] = res; break;
    }
}
```

Fragment 3: Klassieke emulatie van een ALU

Hierbij is `Float4` een klasse voor een vector van vier componenten. Merk op dat hier nog geen ondersteuning voor *swizzle* en *mask* is toegevoegd, en een dergelijke ALU ook bewerkingen met drie argumenten moet kunnen uitvoeren. Toch puilt deze code reeds duidelijk uit van de controlestructuren, en dat louter om één instructie uit te voeren. Bovendien wordt telkens opnieuw data uit het geheugen gehaald en teruggeschreven.

Appendix B: Direct3D 9 poster geeft een idee hoeveel toestandsparameters Direct3D kent buiten de *shaders*. Ook daarvoor zijn met de naïeve aanpak controlestructuren nodig die een hoge efficiëntie sterk belemmeren.

De prestaties van deze aanpak zijn dan ook zeer beperkt. Toch wordt ze veelvuldig toegepast in emulatieprojecten, ook voor 3D-weergave. De eerder vernoemde *Direct3D Reference Rasterizer* van Microsoft (ook gekend als ‘REF’) is een typisch voorbeeld. Het bevestigt de verwachte prestatiekarakteristieken. Doorgaans duurt het tekenen van één beeld meerdere seconden, terwijl er meerdere beelden per seconde nodig zijn voor bruikbare interactiviteit. REF werd dan ook niet ontwikkeld als mogelijke vervanging van hardware, maar zoals de naam al doet vermoeden als referentie voor hardware-implementaties. Ze vormt een ondubbelzinnige specificatie en bewijst ook dat nieuwe toevoegingen aan de API implementeerbaar zijn.

Bovenstaande code kan in de huidige vorm nog enigszins geoptimaliseerd worden. Men kan bijvoorbeeld alle ‘registers’ in één rij plaatsen en de indices aanpassen. Dan hoeft

men ook niet meer expliciet de tijdelijke variabelen `a0`, `a1` en `res` te gebruiken en wordt de code een stuk compacter. Vele controlestructuren blijven echter nodig en zijn niet weg te werken zonder een fundamenteel andere aanpak.

2.4.2 Efficiënte aanpak

De essentie van het probleem met de naïeve aanpak is dat wanneer men duizend elementen verwerkt, er duizend keer alle betrokken toestandsparameters (inclusief de *shader*-code) worden nagegaan. Deze zijn echter voor ieder element tijdens de uitvoering van één tekencommando gelijk. Men zou dus in staat moeten zijn om slechts één maal de toestand na te gaan voor alle elementen samen. In de inleiding werd reeds gehint naar een techniek die dit mogelijk maakt: specialisatie.

Wanneer we echter naar de software-implementatie van de ALU kijken (fragment 3), merken we geen onmiddellijke mogelijkheid tot specialisatie. De uitvoering van één instructie voor één element vergt nu eenmaal het nagaan van de instructievelden. Specialisatie is pas mogelijk wanneer we de verwerking van meerdere elementen of meerdere instructies beschouwen. We zullen dit respectievelijk horizontale en verticale specialisatie noemen.

Horizontale specialisatie is het directe equivalent van *batching* in hardware. Onderstaand fragment illustreert het principe:

```
void processHorizontal()
{
    for(int i = 0; i < batchCount; i++)
    {
        loadInput(i);

        for(int j = 0; j < instructionCount; j++)
        {
            switch(instruction[j].operation)
            {
            case ADD: addBatch(instruction[j].resIndex,
                               instruction[j].arg0Index,
                               instruction[j].arg1Index);
                break;
            case [...]
            }
        }

        storeOutput(i);
    }
}

void addBatch(int res, int a0, int a1)
{
    r[0][res] = r[0][a0] + r[0][a1];
    r[1][res] = r[1][a0] + r[1][a1];
    r[2][res] = r[2][a0] + r[2][a1];
    [...]
    r[9][res] = r[9][a0] + r[9][a1];
}
```

Fragment 4: Horizontale specialisatie

Deze code werkt met batches van 10 elementen. Voor elke instructiebewerking wordt een functie geschreven die deze bewerking uitvoert op een volledige batch elementen (in dit voorbeeldfragment enkel voor de optelling). Merk op dat we alle registertypes in één geheugenrij hebben geplaatst, en we de tien bewerkingen expliciet 'ontrold' hebben. Het aantal controlestructuren dat doorlopen wordt is hierbij gereduceerd tot één per tien elementen, per instructie.

Grotere batches kunnen de overlast van de controlestructuren verder verminderen, maar dit levert steeds minder op. Bovendien is het dan lastiger om volle batches te verkrijgen. Mogelijks nog belangrijker is dat dit een aanzienlijke hoeveelheid geheugen vergt. Tien elementen van 92 vectorregisters komt neer op 14 KiB, terwijl de CPU's op het moment van schrijven over maximaal 32 KiB L1 datacachegeheugen beschikken. Ander geheugen is te traag voor veelvuldige toegang. Hoewel een doorsnee *shader* niet alle registers zal gebruiken, kunnen we dus niet zo veel grotere batches gebruiken.

Naast de batchgrootte brengt het gebruik van geheugen voor de opslag van de virtuele *shader*-registers nog een beperking met zich mee. Geheugentoegangen zijn steeds trager dan het gebruik van registers (de fysieke registers van de CPU, welteverstaan). De *load* en *store units* van de CPU worden zwaar belast en het kost enkele extra instructies. Wanneer de volgende *shader*-instructie van dezelfde argumenten gebruik maakt moeten ze toch telkens weggeschreven en terug ingeladen worden (net zoals bij de naïeve aanpak het geval was). Ook de indexering kost extra instructies.

Merk op dat hier in theorie nog verder gespecialiseerd zou kunnen worden. Men zou bijvoorbeeld `addR`*`i`*`R`*`j`*`R`*`k`* functies kunnen implementeren, waarbij *`i`*, *`j`* en *`k`* elke mogelijke registerindex doorlopen. Het is echter onmiddellijk duidelijk dat dit tot een 'explosief' groot aantal functies zou leiden (denk ook aan *swizzle* en *mask*). Het is dus niet praktisch, maar in theorie kan het de indexeringskost van de implementatie uit fragment 4 elimineren.

Ten slotte merken we hier op dat de parallelle pijplijnen van moderne CPU's niet efficiënt gebruikt worden. De meeste architecturen beschikken over afzonderlijke ALU's voor optelling en vermenigvuldiging. Maar in het bovenstaande voorbeeld zou de ALU voor vermenigvuldiging ongebruikt blijven, ook al is de volgende *shader*-instructie een `mul` (en dan blijft op zijn beurt de opteller ongebruikt). Dit is theoretisch gedeeltelijk op te lossen door zo veel mogelijk optellingen en vermenigvuldigingen samen te nemen tot `mad` bewerkingen.

Verticale specialisatie beschouwt niet een groepje elementen maar een groepje instructies als specialisatieobject. Wanneer we naar de *shader* als geheel kijken merken we op dat deze invariant is voor een groot aantal elementen. Indien we dus de gebruikte *shaders* op voorhand kenden zouden we voor elk een functie kunnen schrijven die exact die *shader* implementeert op de CPU. Tijdens het draaien van de applicatie hoeft dan enkel nog nagegaan te worden welke *shader* op dat moment actief is, en de corresponderende gespecialiseerde functie kan gebruikt worden.

In de praktijk kennen we echter de *shaders* niet op voorhand. We bevinden ons achter de API en de applicatie kan eender welke *shader* inladen. Tijdens het draaien van de applicatie kennen we de *shaders* echter wel, en dus kan op dat moment de specialisatie uitgevoerd worden. Concreet houdt dit in dat de vertaling van *shader*-code naar binaire

code uitvoerbaar op de CPU uitgesteld moet worden tot het moment dat de elementen verwerkt moeten worden. Onderstaande abstracte codefragment illustreert dit idee:

```
void processVertical()
{
    Function function = generateCode(shader);

    for(int i = 0; i < elementCount; i++)
    {
        function(element[i]);
    }
}
```

Fragment 5: Verticale specialisatie

De complexiteit zit uiteraard in het genereren van de op de CPU uitvoerbare code. Hier gaan we in het volgende hoofdstuk uitvoerig op in. Voorlopig volstaat het om het als een gebruikelijk compilatieproces te beschouwen.

Merk op dat bovenstaande code bij ieder gebruik van de *shader* nieuwe code zal genereren. Dit is niet nodig. Door een cachestructuur te gebruiken kan men eerst nagaan of de code voor de *shader* reeds eerder gegenereerd werd. Indien wel wordt de functie uit de cache gebruikt. Indien niet genereert men de functie en slaat ze op in de cache. De volledige *shader* doet hier telkens dienst als cachesleutel.

Verticale specialisatie lost de voornaamste problemen van horizontale specialisatie op. De *shader*-registers hoeven niet in het geheugen opgeslagen te worden. Hoewel een typische CPU van Intel slechts 8 of 16 vectorregisters heeft kan men doorgaans de meeste tussenresultaten in registers houden. Hiermee voorkomt men dus dat ze telkens weggeschreven en terug opgeladen moeten worden. Verder is er totaal geen sprake meer van extra overlast door controlestructuren (het zoeken in de functiecache is zeer snel). Daarenboven kunnen dicht opeenvolgende instructies wel gebruik maken van de parallelle pijplijnen van de CPU.

Er is echter één noemenswaardig nadeel: de latenties van de instructies kunnen de uitvoering vertragen. Gelukkig beschikken moderne CPU's over *out-of-order execution*. Deze technologie zorgt ervoor dat de CPU verder in de instructiestroom kan zoeken naar onafhankelijke instructies. Ze werkt vooral goed voor lange code met weinig sprongen, en dat is hier het geval. Toch geeft het geen volledige garantie dat code met veel afhankelijkheden niet gelimiteerd wordt door de latenties. We kunnen echter wel trachten de afhankelijkheden te minimaliseren. Hiervoor komen we opnieuw bij *batching* uit.

Door geen code te genereren voor de verwerking van één element, maar voor meerdere onafhankelijke elementen tegelijk, worden de latenties ingevuld. Ook hier is de batchgrootte cruciaal. De latentie van rekenkundige bewerkingen bedraagt gemiddeld een viertal klokcycli. Geheugenlatenties zijn hoger, en afhankelijk van de locatie van de data. Toch moet de batchgrootte klein blijven om een goed registergebruik te garanderen. *Out-of-order execution* helpt ook nog steeds bij hoge geheugenlatenties. Vier elementen blijkt dan ook een goed compromis.

Er is nog een veel doorslaggevender reden waarom vier elementen per batch ideaal is: scalaire verwerking. Deze techniek verwerkt component per component, maar kan dit

voor meerdere elementen in parallel. Omdat CPU's ook uitgerust zijn met vectoreenheden van vier componenten komt dit zeer goed uit. Merk bovendien op dat we hiermee net zoals bij de GPU zonder extra bewerkingen *swizzle* en *mask* verkregen, en er geen berekeningen uitgevoerd worden waarvan de resultaten weggegooid worden.

2.4.3 Niet-programmeerbare pijplijntrappen

De andere trappen van de grafische pijplijn, driehoekverwerking en rasterbewerking, verschillen niet fundamenteel van de wel programmeerbare trappen. Er zijn minder toestandsparameters (shaders zijn een dichte verzameling van toestandsparameters), maar ook hier is specialisatie van nut om controlestructuren geen significant deel van de uitvoeringstijd in te laten nemen.

Hier zijn de nadelen van horizontale specialisatie niet zo uitgesproken. Er is geen grote virtuele registerset die in het geheugen moet geplaatst worden. De bewerkingen zijn ook lang genoeg om een goed gebruik van de fysieke registers en de parallelle pijplijnen van de CPU te garanderen. Voordelen van horizontale specialisatie die nog niet vernoemd werden zijn dat het geen functiecache vergt en dat het eenvoudiger te debuggen is.

Textuurbemonstering (`tex`) verschilt sterk van rekenkundige bewerkingen en kan dus ook aanzien worden als een component die een aangepaste implementatie vergt. Vooral de hoge geheugenlatentie kan de uitvoering van de rest van de *shader* remmen. Een moderne GPU gebruikt hiervoor *scheduling* van kleinere batches, wat ook in zekere zin mogelijk is op de CPU. Door meermaals de code voor de rekenkundige verwerking van vier elementen achter elkaar te plaatsen, verkrijgen we grotere (totale) batches zonder het voordeel van goed gebruik van de (fysieke) registers te verliezen. Deze grotere batches zijn vervolgens beter geschikt om de geheugenlatenties tijdens textuurbemonstering op te vangen.

Door de krachtige geheugenhiërarchie en *prefetching* mogelijkheden van de CPU blijft het nut van deze complexe aanpak echter beperkt. Toch kunnen we concluderen dat elke techniek die gebruikt wordt op de GPU, vertaald kan worden naar de CPU, mits de nodige inspanning.

2.5 Vectorinstructies

Essentieel voor een efficiënte software-implementatie is het gebruik van vectorinstructies. De maximale prestaties van de CPU gelden immers enkel bij het gebruik van deze instructies. Ze bieden een beperkte doch belangrijke vorm van parallellisme voor de CPU. Bij x86 processors vindt men de vectorinstructies terug in de MMX en SSE instructiesetuitbreidingen.

MMX staat voor *MultiMedia eXtension*[1] en biedt de mogelijkheid om vectoren van 64-bit te verwerken. Ze bestaan uit acht componenten van één byte, vier componenten van twee byte, of twee componenten van vier byte. De componenten zijn steeds gehele getallen. Hierdoor is MMX vooral geschikt voor video-, audio-, en beeldverwerking.

[1] Volgens sommigen staat het voor *Matrix Math eXtensions*. Intel geeft er zelf geen vaste betekenis aan.

SSE staat voor *Streaming SIMD Extension*, waarbij SIMD een acroniem is voor *Single-Instruction Multiple Data*. SSE werkt met 128-bit vectoren van vier vlottende-kommagetallen van enkelvoudige precisie. Hierdoor is het zeer geschikt voor de implementatie van *shader*-instructies. SSE werd later uitgebreid met SSE2. Deze laat ook toe te werken met vectoren van twee vlottende-kommagetallen van dubbele precisie. Bovendien biedt het alle MMX-instructies voor gehele getallen, maar dan uitgebreid tot 128-bit vectoren. Later verschenen ook SSE3 en SSSE3 (*Supplemental* SSE3), die een klein aantal instructies toevoegen voor 'horizontale' bewerkingen, zoals het optellen van componenten binnen een vector. Op het moment van schrijven is SSE4 in ontwikkeling, dat onder meer een instructie voor een inwendig product zal bevatten. Merk op dat de CPU hierbij een (zoveelste) stapje dichter zet bij de instructieset van de GPU.

Hoewel ze zeer krachtig zijn in hun toepassingsdomein zijn vectorinstructies nooit goed ontvangen, zeker niet voor x86 processors. MMX kende enkele beperkingen die het meteen onpopulair maakten. Onder andere hergebruikt MMX de registers van de scalaire *floating-point unit* (FPU) waardoor men het niet zomaar samen kan gebruiken met vlottende-kommabewerkingen. SSE is minder problematisch, maar slaagt er ook niet in de gemiddelde programmeur te bekoren. Weinig compilers kunnen ermee overweg, en deze die het wel kunnen (via automatische vectorisatie) hebben veel hints nodig van de programmeur om kwaliteitsvolle code te genereren. Vaak valt men dan ook terug op het gebruik van assembleercode, wat voor velen een drempel is. Bovendien zijn niet alle processors uitgerust met ondersteuning voor de laatste SSE instructiesetuitbreidingen, waardoor men van functies die er gebruik van maken verschillende versies moet schrijven (een vorm van specialisatie).

We wensen dus via dynamische codegeneratie ook volledige toegang te hebben tot de vectorinstructies, maar wel op een gebruiksvriendelijke wijze.

2.6 Gedetailleerde prestatieanalyse

Dynamische specialisatie laat ons toe de totale rekenkracht van de CPU te benutten. Samen met de eerdere detaillering van de werking van een GPU loont het dus de moeite om enkele concrete modellen nader te vergelijken.

Bemerk vooraf dat GPU's en CPU's op sommige vlakken een verbluffende overeenkomst vertonen, veel meer dan op eerste zich vermoed zou kunnen worden. GPU's zijn immers ontwikkeld met grafische verwerking in gedachte. De programmeerbaarheid en het voordurende streven naar meer flexibiliteit brengt ze dichter bij de architectuur van de CPU. Recent worden ze dan ook gebruikt voor generieke rekenintensieve toepassingen, of GPGPU[1] toepassingen. Anderzijds winnen de CPU's aan rekenkracht door de toevoeging van vectorinstructies en parallelle kernen. De geplande SSE4 instructiesetuitbreiding zal zelfs het equivalent van de `dp4` instructie toevoegen.

De NVIDIA GeForce 8600 GTS uit de inleiding beschikt over 32 scalaire *stream processors* op 1,45 GHz. De *stream processors* van deze generatie uit één ALU voor optelling+vermenigvuldiging en één ALU voor speciale functies en interpolatie (beter

[1] General-Purpose computation on GPU's

gekend als de SFU). Wanneer we enkel de optellingen en vermenigvuldigingen meerekenen (dit is standaard) verkrijgen we de eerder vernoemde 92,8 miljard bewerkingen per seconde of GFLOPS. De SFU's bieden deze GPU echter een significant voordeel. Iedere klokperiode kunnen ze een complexe bewerking zoals een deling uitvoeren, dankzij ingenieuze benaderende berekeningen [3].

Naast de stream processors beschikt deze GPU over 16 *texture samplers* (eenheden voor textuurbemonstering) en 8 ROP's die aan 675 MHz draaien. Hij kan dus 10,8 miljard textuurmonsters per seconde leveren, en maximaal 5,4 miljard pixels per seconde uitschrijven.

De Intel Core 2 Extreme QX6800 uit de inleiding is een CPU met vier kernen. Hij heeft een klokfrequentie van 2,93 GHz en afzonderlijke vector-ALU's voor optelling en vermenigvuldiging. In totaal zijn dit dus beschouwd worden als 16 scalaire ALU's voor optelling en 16 voor vermenigvuldiging. Bijgevolg haalt het de eerder vernoemde 93.8 GFLOPS. Deze rekenkracht vertaalt zich zeer goed naar de prestaties voor optellingen en vermenigvuldigingen binnen de *shaders*.

Transcendente functies worden echter slecht ondersteund. Delingen en vierkantswortels zijn zeer traag. Logaritmes, exponentiëlen en geometrische functies zijn zelfs niet aanwezig als processorinstructie (in vectorvorm). Ze moeten dan ook geïmplementeerd worden via een reeks andere instructies. Hoewel Direct3D geen volle precisie vereist kan dit toch 20 instructies in beslag nemen. In vergelijking met onderstaande IGP volstaan echter deze prestaties.

Textuurbemonstering is veel kritischer. De CPU is totaal niet uitgerust voor deze taak en het kan dan ook 50 instructies kosten voor de basiskwaliteit (bilineair filteren). Toch is dit voldoende voor een groeiend gamma aan applicaties. De vier kernen en hoge frequentie van deze CPU compenseren gedeeltelijk het hoge aantal instructies (en dus klokcycli) dat nodig is. Zonder andere werklast kan ruwweg een half miljard textuurmonsters per seconde geleverd worden.

Anderzijds is de CPU niet gelimiteerd door het aantal ROP's voor rasterbewerking. Het een relatief eenvoudige bewerking, waardoor de rekenkracht volstaat om meer dan de bovenstaande 5,4 miljard pixels per seconde te verwerken. Ironisch genoeg limiteert de geheugenbandbreedte van de CPU de maximaal haalbare prestaties voor rasterbewerking tot 2,13 miljard pixels per seconde[1]. Dit is nog meer dan ruim voldoende, en de echte beperking ligt hoe dan ook bij de textuurbemonstering.

Laten we een gewenste resolutie van 1280x1024 beeldpunten beschouwen, en 30 beelden per seconde voor zeer vlotte interactiviteit. Dit komt neer op bijna 300 klokcycli per beeldpunt per beeld. Dit is dus ruim voldoende om enkele texturen te kunnen bemonsteren. Toch komt een beeldpunt en een pixel niet strikt overeen. Pixels worden met de standaard grafische pijplijn veelvuldig overschreven. Een zogenaamde *overdraw* van vijf is niet ongewoon. Dit brengt de gewenste resolutie of interactiviteit in gedrang.

De veelzijdigheid van de CPU laat echter toe om 'intelligente' algoritmes te gebruiken. Dit in tegenstelling tot de 'brute kracht' aanpak van de GPU. Zoals eerder opgemerkt kan een GPU niet zomaar data uit z'n pijplijnen halen, terwijl dit met de CPU op

[1] Tenzij men het beeldraster in 'tegels' opsplitst die in de cache passen en apart verwerkt worden.

softwareniveau wel kan. Geavanceerde algoritmes om de *overdraw* te minimaliseren bestaan (bv. *deferred rendering*), maar zullen hier niet verder besproken worden.

Bemerk ten slotte dat een CPU vrijwel automatisch voor unificatie en *scheduling* en een verdeling van de werklast zorgt. Taken worden sequentieel en onmiddellijk na elkaar uitgevoerd, zonder mogelijke bottlenecks zoals bij de GPU nog steeds mogelijk is in de driehoekverwerking, elementverwerking of de ROP's. We spreken enkel van hete code of *hotspots*, code die in significante mate de effectieve prestaties bepaalt. Verder vormen de caches van de CPU een krachtig automatisch opvangnet voor de geheugenlatentie.

De Intel Graphics Media Accelerator X3000 is een IGP uit de G965 chipset. Hij is uitgerust met 8 scalaire ALU's voor optelling en 8 voor vermenigvuldiging, en draait op 667 MHz. De theoretische prestaties zijn liggen dus op 10,7 GFLOPS. De speciale functies worden uitgevoerd op een "*Mathbox*" die zich buiten het pad van de andere ALU's bevindt en meerdere klokcycli nodig heeft om het resultaat te berekenen. De theoretische prestatie voor textuurbemonstering bedraagt 2,13 miljard monsters per seconde.

Bemerk dat een IGP het hoofdgeheugen deelt met de CPU. Dit bewijst meteen dat de bandbreedte ook volstaat voor een software-implementatie.

De X3000 is de krachtigste IGP van Intel van dit moment, en Intel is markleider op het vlak van IGP's. We kunnen dus uit de theoretische prestaties concluderen dat een vierkernige CPU een reële concurrent kan zijn voor IGP's. De snelle evolutie van CPU's en het "voldoende prestaties"-principe garanderen dan ook dat in de toekomst systemen zonder GPU voor 3D-weergave een optie vormen.

Dit alles gaat echter uit van een hoge efficiëntie van de dynamische codegeneratie. Het volgende hoofdstuk beschrijven we dan ook SoftWire++, het nieuwe raamwerk voor dynamische codegeneratie dat aan de nodige en gewenste vereisten voldoet.

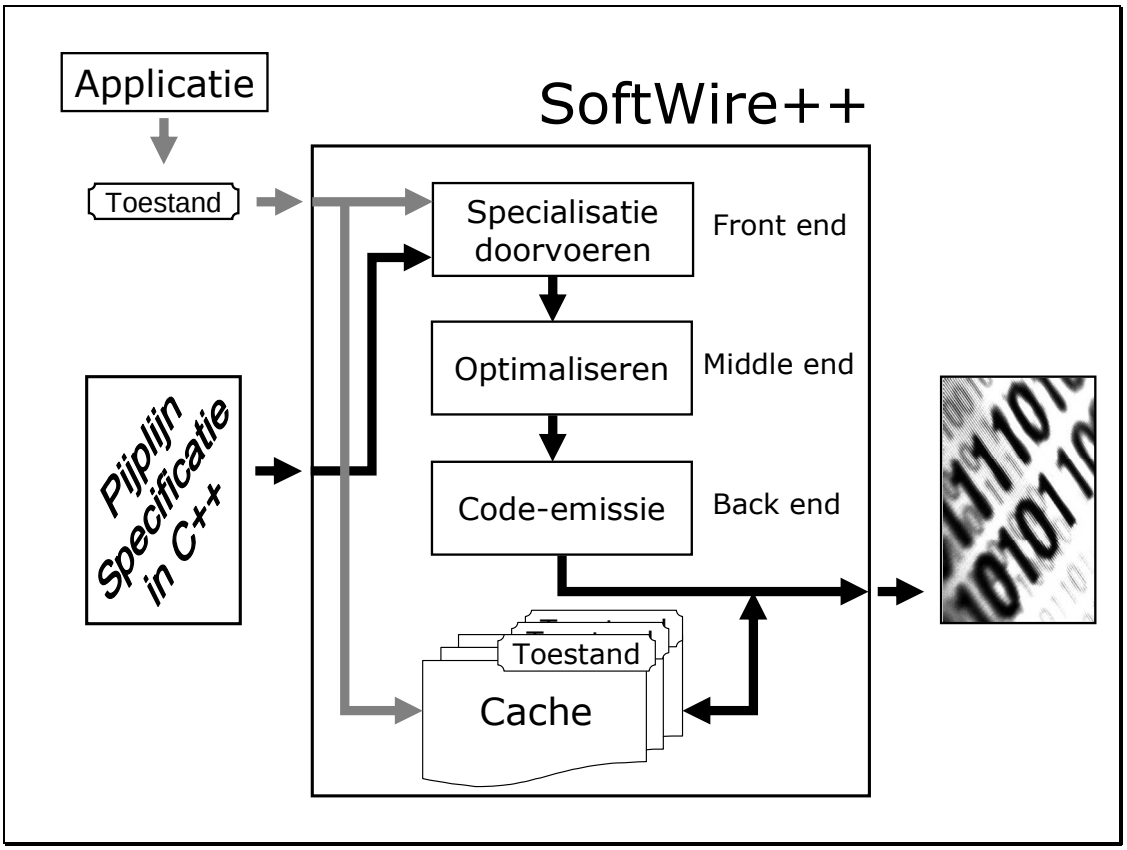
3
Ontwerp en implementatie

"All problems in computer science can be solved by another level of indirection"

— Butler Lampson

3.1 Overzicht

Het SoftWire++ project is een raamwerk voor dynamische specialisatie[1]. Het werd specifiek voor grafische doeleinden ontwikkeld, maar zou ook ruimer ingezet kunnen worden. Onderstaande figuur biedt een overzicht van de onderdelen van SoftWire++ en hun samenwerking:



Figuur 18: SoftWire++

[1] Wanneer het niet nodig is een onderscheid te maken met de vorige versie spreken we ook kortweg van SoftWire.

De applicatie die gebruik maakt van de grafische bibliotheek stelt via de API de toestand van de grafische pijplijn in. Bij het ontvangen van een tekencommando maakt de grafische bibliotheek gebruik van SoftWire++ om een generieke specificatie van de grafische pijplijn te compileren en te optimaliseren voor de huidige toestand. Dit levert binaire code op die als functie aan te roepen is (zie ook fragment 5).

Binnen het SoftWire++ raamwerk onderscheiden we vier onderdelen:

- Specialisatie doorvoeren (*front end*): De generieke specificatiecode wordt 'herschreven' tot een gespecialiseerde versie. Hierbij wordt gebruik gemaakt van de toestand om enkel de nodige bewerkingen over te houden.
- Optimaliseren (*middle end*): De gespecialiseerde code wordt geoptimaliseerd via enkele specifieke algoritmes. Dit gebeurt in een tussenvorm die onafhankelijk is van het processortype.
- Code-emissie (*back end*): De geoptimaliseerde code in tussenvorm wordt vertaald naar processorspecifieke binaire code. Hier worden ook enkele laatste optimalisaties uitgevoerd dit pas op dit niveau mogelijk zijn.
- Cache: De binaire code van reeds uitgevoerde dynamische specialisaties wordt opgeslagen in een cache voor later hergebruik. De (relevante) toestandsparameters worden als zoeksleutel samen met de functie opgeslagen.

3.2 Specialisatie via C++

Het uitvoeren van generieke code komt overeen met het nagaan van de toestand om via controlestructuren de gewenste bewerkingen uit te voeren. Merk op dat indien we dus bijhouden wat de uitgevoerde bewerkingen zijn, we meteen een gespecialiseerde voorstelling van de code verkrijgen.

SoftWire++ maakt gebruik van het overladen van C++ operatoren om bij te houden welke bewerkingen uitgevoerd worden. Onderstaande code (fragment 6) illustreert dit krachtige concept:

```
struct Float4
{
    Float4 operator+(const Float4 &rhs)
    {
        Float4 res;
        list.append(ADD4, res, *this, rhs);
        return res;
    }

    Float4 operator*(const Float4 &rhs)
    {
        Float4 res;
        list.append(MUL4, res, *this, rhs);
        return res;
    }

    [...]
};
```

```
void generateALU(const ShaderInstruction &instruction)
{
    Float4 res, a0, a1, a2;   // Result and arguments

    [...]   // Read source arguments

    switch(instruction.operation)
    {
    case ADD: res = a0 + a1;      break;
    case MUL: res = a0 * a1;      break;
    case MAD: res = a0 * a1 + a2; break;
    case [...]
    }

    [...]   // Write result to destination
}
```

Fragment 6: Overladen van operatoren

Merk de sterke overeenkomst met fragment 3 op. Daar voerden we effectief de ALU-bewerking uit. Bovenstaand voorbeeld bouwt echter een lijst op van de uitgevoerde bewerkingen (om ze later te compileren naar uitvoerbare code).

Op het eerste zicht lijkt dit niet te verschillen van de *shader*-instructie zelf. Wanneer we echter de implementatie van de `MAD` *shader*-instructie (*Multiply-ADd*) beschouwen, zien we dat één maal `operator*` en één maal `operator+` aangeroepen wordt. Er vindt dus een vertaling naar meer elementaire bewerkingen plaats. Bovendien zal automatisch het tijdelijke resultaat van de vermenigvuldiging opgeteld worden bij het laatste argument. Elke instantie van de `Float4` klasse krijgt hiervoor een nummer dat het uniek identificeert.

De (uitgewerkte) `Float4` klasse is deel van het SoftWire++ raamwerk, terwijl `generateALU` deel is van de grafische bibliotheek. Dankzij SoftWire++ hoeft voor de implementatie van de grafische bibliotheek enkel intuïtieve generieke code geschreven te worden. Hoe complex de formules voor een bepaalde bewerking ook zijn, de C++

compiler doet het zware werk om de juiste overladen operatorfuncties aan te roepen en de tijdelijke variabelen te beheren. De programmeur van de grafische bibliotheek kan zich concentreren op de functionaliteit.

De controlestructuren van C++ doen hierbij dienst als constructies voor conditionele compilatie. Ze zijn met ander woorden het 'dynamische' equivalent van `#if`, `#elif`, `#else` en `#endif`. Naast `if` en `else` beschikken we echter ook over `swich`, `for`, `while`, `do`…`while`, `goto`, `continue` en `break`. Dit geeft ons krachtige instrumenten om het verloop van de dynamische codegeneratie te regelen. We kunnen bijvoorbeeld bovenstaande `generateALU` in een lus aanroepen om voor elke instructie in een *shader* de gewenste code te genereren.

Dynamische conditionele compilatie kan ook complexere beslissingen uitvoeren. Men beschikt immers over alle functionaliteit van C++. Bijvoorbeeld de implementatie van de `tex` instructie hangt af van een samenhang tussen de gewenste filter, het textuurformaat en het textuurtype. De transcendente functies hebben ook een relatief complexe implementatie die afhangt van de gewenste precisie, en kan vereenvoudigd worden wanneer constante parameters gebruikt worden (bv. $3^y = 2^{y \cdot \log_2 3} = 2^{y \cdot 1{,}585}$). Enkel de grafische bibliotheek heeft toegang tot al de toestandsparameters om deze beslissingen te maken.

Daar waar we bij klassieke emulatie de complexiteit van de controlestructuren moesten minimaliseren, kunnen we hier in detail de toestand nagaan om enkel de effectief nodige bewerkingen in de gegenereerde functie op te nemen. Ook voor starre pijplijnen kan men op natuurlijke wijze de gewenste functionaliteit specificeren.

Merk op dat dit veel moeilijker te implementeren was indien de specificatie van de te genereren code in een afzonderlijk tekstbestand was geschreven. Men was dan beperkt tot eenvoudiger conditionele compilatie, of moest dan via interpretatie de complexere afhankelijkheden van toestandsparameters evalueren.

We slaan dus effectief enkele klassieke compilatiestappen over: We hoeven geen tekst te ontleden, geen syntaxis na te gaan, geen semantische analyse uit te voeren, enz. We genereren onmiddellijk code in tussenvorm die klaar is voor optimalisatie en vertaling naar uitvoerbare code.

3.3 Taalontwerp

SoftWire++ definieert een eigen taal. Het raamwerk steunt wellicht volledig op C++, maar types zoals `Float4` en het gedrag van hun operatoren vormen een abstractie die zich gedraagt als een totaal nieuwe taal.

Het ontwerp van de SoftWire++ taal reikt dan ook verder dan enkel `Float4`. De nood aan een vectortype met vier componenten is duidelijk, maar ook andere types zijn noodzakelijk. In de eerste plaats hebben we ons geïnspireerd op C om de taal verder uit te bouwen.

Het lezen en schrijven van het geheugen, met name de vertexbuffer en texturen, vergt wijzers en gehele getallen als index. Onderstaande fragment illustreert de syntaxis die SoftWire++ hiervoor ondersteunt:

```
Float4 readInput(const Pointer<Float4> &vertexBuffer,
                 int vertexStride,   // Size of a vertex
                 int offset,         // Component offset
                 const Int &index)   // Vertex index
{
    Pointer<Float4> stream = vertexBuffer + offset;

    return *(stream + vertexStride * index);
}
```

Fragment 7: Wijzers en gehele getallen

Deze code kan dienen om de 'v' invoerregisters in te lezen (zie ook figuur 13). De syntaxis om een wijzer (*pointer*) te definiëren wijkt duidelijk af van die van C/C++. Dit komt doordat `Float4*` niet als aparte klasse gedefinieerd kan worden met overladen operatoren. C++ sjablonen of *templates* laten echter toe een `Pointer` klasse te definiëren die de semantiek van wijzers ondersteunt. Tussen < en > kan eender welke type gebruikt worden, ook samengestelde types en andere wijzers (bv. `Pointer<Pointer<Int>>`).

Merk op dat 'dynamische' types bij conventie met een hoofdletter worden geschreven (zoals gebruikelijk bij klassen), en de types met kleine letters de ingebouwde C types zijn. Dit verschil is van groot belang. In bovenstaande code is bijvoorbeeld `offset` een toestandsparameter, terwijl `index` een waarde is die enkel gekend is tijdens het uitvoeren van de functie.

Naast Int ondersteunt de SoftWire++ taal ook de andere C basistypes in 'dynamische' vorm. Wat het nog van C overneemt is de controlestructuren. `IF`, `ELSE`, `FOR`, `WHILE`, `GOTO`, `BREAK` en `CONTINUE` zijn de equivalenten van de in C gelijknamige maar met kleine letters geschreven controlestructuren. Het zijn macro's, die bij conventie altijd volledig in hoofdletters worden geschreven. Onderstaande code verduidelijkt hoe `FOR` werkt:

```
#define FOR(init, cond, loop)                          \
init;                                                  \
int loopTop = newLabel();                              \
label(loopTop);                                        \
for(int loopEnd = newLabel();                          \
    loopEnd >= 0 && branchFalse(cond, loopEnd);        \
    loop, jump(loopTop), loopEnd = label(loopEnd))
```

```
FOR(Int x = xLeft, x < xRight, x++)
{
    processPixel(x);
}
```

Fragment 8: Dynamische lus

Het bovenste deel toont de implementatie van `FOR` in SoftWire++, het onderste deel hoe het gebruikt kan worden in de grafische bibliotheek.

`FOR` genereert de code voor een lus. Eerst voert het de initialisatie uit. In dit geval wordt `xLeft` toegekend aan een nieuw geheel getal `x`. Vervolgens wordt een label aangemaakt om de top de lus aan te duiden. `label(loopTop);` plaatst dit label in de lijst met te genereren code. Hierna wordt een echte `for`-lus gebruikt. Deze maakt echter slechts één iteratie. We willen immers dat de code binnen de `FOR`-lus slechts één maal gegenereerd

wordt. De `for`-lus is nodig omdat de C++ compiler op het einde van de lus de `loop`-uitdrukking uitvoert. Concreet zal `x++` slechts gegenereerd worden na `processPixel`. `newLabel` zal steeds een positief getal retourneren, terwijl `label` steeds een negatief getal retourneert. Dit garandeert dat de `for`-lus slechts één maal itereert. Het laatste deel van deze puzzel is de `branchFalse` functie, die de code genereert die ervoor zorgt dat naar het einde van de lus gesprongen wordt wanneer de voorwaarde `x < xRight` vals is geworden.

De andere 'dynamische' controlestructuren worden via gelijkaardige macro's gedefinieerd. Voor `IF`, `ELSE` en `WHILE` wordt ook een `for`-lus gebruikt, gezien de mogelijkheid om aan het einde van het blok een label en/of voorwaarde te kunnen plaatsen.

Hoewel deze constructies relatief artificeel en gecompliceerd ogen zijn ze voor de programmeur van de grafische bibliotheek eenvoudige abstracties voor het genereren van controlestructuren. Anders moest hij zelf conditionele sprongen en labels plaatsen. Ze vormen dan ook een krachtige toevoeging aan de SoftWire++ taal. Men kan met de types, operatoren en controlestructuren praktisch elke gewenste functie genereren.

Voor de volledigheid vermelden we ook nog dat een functieproloog via de `prologue` functie gegenereerd wordt. Aan deze functie wordt ook het aantal argumenten van de te genereren functie meegedeeld. De argumenten zelf kunnen gelezen worden via de `argument` functie. Om een 'dynamische' functie af te sluiten en een waarde te retourneren wordt de `epilogue` functie gebruikt.

SoftWire++ stopt echter niet bij de functionaliteit van C. Het `Float4` type is geïnspireerd op HLSL, de hoog-niveau *shader*-taal van Direct3D. HLSL biedt de applicatieprogrammeur de mogelijkheid *shaders* in een hogere taal dan de assembleerinstructies van tabel 1 te schrijven. `float4` is een basistype van HLSL. Daarnaast ondersteunt het ook kortere vectoren (`float3` tot `float1`), en matrices van diverse dimensies (`float4x4` tot `float1x1`). Van al deze types wordt ook een variant met gehele getallen en booleaanse waarden ondersteund.

SoftWire++ implementeert deze types, plus alle vectortypes die zich rechtstreeks naar MMX en SSE laten vertalen. Hierdoor wordt gegarandeerd dat geen prestaties verloren gaan door de vertaling van de hogere datatypes naar x86. Daarnaast neemt SoftWire++ ook de syntaxis van HLSL voor *swizzle* en *mask* over:

```
float4 a = {0, 0, 0, 0};
float4 b = {1, 2, 3, 4};

a.x   = b.w;      // a = {4, 0, 0, 0}
a.yzw = b.xzyx;   // a = {4, 3, 2, 1}
```

Fragment 9: *Swizzle* en *mask* in HLSL

Voor variabelen aan de rechterkant van de toekenning werkt de `.{xyzw}` operator als *swizzle*. Een tijdelijke vector wordt gevuld met de componenten van de variabele, in de volgorde aangegeven door de componenten die achter de punt volgen. Wanneer minder dan vier componenten opgegeven worden, wordt de laatste component herhaald. Voor variabelen aan de linkerkant van de toekenning werkt de `.{xyzw}` operator als *mask*.

Enkel de componenten die achter de punt volgen worden overschreven. Merk op dat bovenstaande voorbeeld ook korter te schrijven is: `a = b.wzyx`.

Exact dezelfde syntaxis werd verkregen voor SoftWire++ door gebruik te maken van *properties*. Deze (onofficiële) functionaliteit van C++ laat toe om bij het lezen en schrijven van variabelen impliciet een functie aan te roepen. Net zoals bij het overladen van operatoren wordt dus de uitgevoerde bewerking opgeslagen in de tussenvorm.

Ten slotte kent de SoftWire++ taal het concept van intrinsieke functies. Dit zijn bewerkingen zoals `sqrt(·)` die geen operator hebben (binnen C++) maar ook geen functieaanroep voorstellen. Ze vertalen zich onmiddellijk naar één of slechts enkele instructies. Bemerk de overeenkomst met *inline* functies.

We concluderen dat de SoftWire++ taal zo dicht mogelijk aansluit bij de syntaxis en semantiek van zowel C als HLSL. De C-functionaliteit maakt de taal veelzijdig, terwijl HLSL-functionaliteit krachtige vectorbewerkingen biedt voor grafische toepassingen.

3.4 Optimalisatie

De *front end* van SoftWire++ levert een lijst op van instructies in tussenvorm. De tussenvorm is een lager niveau dan de SoftWire++ taal maar hoger dan processorspecifieke instructies. Een rekenkundige instructie in tussenvorm is opgebouwd als `x = y` □ `z`, waarbij □ een bewerkingsoperator voorstelt (zie ook fragment 6). Er is ook geen limiet op het aantal 'registers'. In de *middle end* worden optimalisaties uitgevoerd op de tussenvorm met als voornaamste doel onnodige bewerkingen te elimineren.

SoftWire++ dient de compileertijd zo laag mogelijk te houden. De compilatie gebeurt immers tijdens de uitvoering van de applicatie. Daar waar voor een klassieke compiler enkele seconden per bestand aanvaardbaar is, moet SoftWire++ in milliseconden klaar zijn. De cache reduceert wel het aantal functies dat moet gegenereerd worden, maar bij plotse veranderingen in de scène (b.v. nieuwe objecten die getekend moeten worden) moeten er toch typisch een tiental functies gegenereerd worden. Om de interactiviteit niet te schaden moeten dit in een fractie van de tijd voor het tekenen van één beeld gebeuren.

Zoals we reeds eerder opgemerkt hebben slaat SoftWire++ de compilatiestappen voor tekstontleding, syntaxiscontrole en semantische analyse over. Het doorvoeren van de specialisatie (constructie van variabelen, aanroepen van de operatoren, enz.) vormt slecht een geringe kost. Zoals we verder zullen zien is ook de code-emissie geen zware stap. Het grootste deel van de compileertijd zit dus in de optimalisaties.

Anderzijds zijn de optimalisaties cruciaal voor hoge prestaties van de gegenereerde functies. Een kleine extra inspanning tijdens de optimalisatiefase kan leiden tot een merkbaar hoger aantal beelden per seconde.

Bijgevolg maakt SoftWire++ een compromis wat betreft de gebruikte optimalisatiealgoritmes. Enkel optimalisaties die de programmeur van de grafische bibliotheek niet of moeilijk manueel in de specificatiecode kan toepassen werden geïmplementeerd. Enkele niet geïmplementeerde optimalisaties en de argumenten voor hun ontbreken zijn:

- Lusontvouwing: De algoritmes om lussen te detecteren die geschikt zijn voor ontvouwing zijn relatief complex. Bovendien is de programmeur van de grafische bibliotheek in staat om zelf de ontvouwing uit te voeren, eventueel met het gebruik van een `for`-lus. Iedere iteratie zal immers de code binnen de lus opnieuw aan de lijst met instructies in tussenvorm toevoegen.
- Lusinvariante code verplaatsen: Detecteren of een berekening bij iedere iteratie hetzelfde resultaat oplevert is niet triviaal. De programmeur van de grafische bibliotheek kan deze berekeningen eenvoudig zelf buiten de lus laten uitvoeren.
- Eliminatie van gelijke subexpressies: Een typisch voorbeeld hiervan is `(x + y) * (x + y)`. Dit kan herschreven worden met één optelling door een tijdelijke variabele te gebruiken voor de som, of een `sqr` functie te schrijven die de kwadratering uitvoert.
- Constante uitdrukkingen uitwerken: Uitdrukkingen met constanten worden reeds door de C++ compiler uitgewerkt. Dit is niet het geval voor bijvoorbeeld `2 * x * 3`, waarbij `x` een dynamische variabele is. Hierbij is het aan de programmeur van de grafische bibliotheek om dit te herschrijven als `2 * 3 * x` of meteen `6 * x`.
- *Inline* functies: Alle functies die we aanroepen generen per definitie *inline* code.

Merk op dat we ervan uitgaan dat de programmeur van de grafische bibliotheek voldoende kennis over deze optimalisaties bezit. Dit is uiteraard geen onredelijke aanname. Toch kunnen we niet veronderstellen dat hij alle mogelijke optimalisaties op zich neemt. Hoewel SoftWire++ toelaat om zelf alles als zeer elementaire bewerkingen te genereren en men voor elk van die bewerkingen kan nagaan of ze werkelijk nodig zijn voor een bepaalde toestand, wordt dit snel onwerkbaar.

Stel bijvoorbeeld dat een *vertex shader* een kleurattribuut berekent, en de *pixel shader* hier nooit gebruik van maakt. Men zou dan alle bewerkingen in de *vertex shader* die de kleur berekenen kunnen elimineren. Op het niveau van de grafische bibliotheek zorgt dit echter voor een aanzienlijke complexiteit. De specificatiecode wordt hiermee onbeheersbaar.

SoftWire++ lost dit echter op door de eliminatie van dode (nutteloze) instructies in de tussenvorm. Enkel de berekeningen van variabelen die naar het geheugen weggeschreven worden, blijven bewaard. In het genoemde voorbeeld hoeft men dus enkel te beslissen niet de kleur weg te schrijven, en alle (onafhankelijke) bewerkingen die leiden tot de berekening van de kleur worden automatisch geëlimineerd binnen SoftWire++.

De implementatie hiervan gebeurt door de tussenvorm in SSA-vorm [5] te transformeren. SSA, of *Static Single Assignment*, vereist dat elke variabele slechts éénmaal toegekend wordt. In deze vorm kent elke variabele dus één plaats van definitie. Dit maakt het zeer eenvoudig om afhankelijkheden na te gaan. Het elimineren van dode instructies vergt enkel nog na te gaan of de gedefinieerde variabele ergens hergebruikt wordt.

SSA en het gebruik ervan voor het elimineren van dode code zijn uitvoerig beschreven en geëvalueerd in de literatuur [7]. Relevant voor SoftWire++ is dat het efficiënt is en alle dode instructies kan elimineren.

Een andere optimalisatie die snel is in SSA-vorm is de eliminatie van onnodige kopieën. Vaak worden variabelen gekopieerd naar een andere variabele, maar wordt de eerste niet verder gebruik. Men kan dan de kopie verwijderen en voor verdere bewerkingen de oorspronkelijke variabele gebruiken. Dit wordt ook het propageren van kopieën genoemd.

De laatste optimalisatie die SoftWire++ op dit niveau uitvoert is sterktereductie van bewerkingen met constanten. Dit gebeurt volgens enkele vervangingsregels:

- `x = y * 1` → `x = y` (analoog voor `x = 1 * y`)
- `x = y + 0` → `x = y` (analoog voor `x = 0 + y`)
- `x = y * 0` → `x = 0` (analoog voor `x = 0 * y`)
- `x = y - y` → `x = 0`

De laatste twee vervangingen kunnen verdere reducties teweegbrengen, terwijl de eerste twee propagatie van kopieën kunnen teweegbrengen. Beide worden dus herhaald tot er geen vervangingen of eliminaties meer mogelijk zijn.

Sterktereductie van bewerkingen met constanten is belangrijk omdat in de grafische pijplijn vaak verstekwaarden gebruikt worden. Wanneer bijvoorbeeld een invoerregister 'v' niet verbonden is met de vertexbuffer krijgt hij de waarde (0, 0, 0, 1). Het is niet praktisch om in de specificatiecode bij te houden welke bewerkingen in sterkte reduceerbaar zijn of geëlimineerd kunnen worden. Deze taak is echter efficiënt uit te voeren in de *middle end* in SSA-vorm.

3.5 Code-emissie

In de *back-end* van SoftWire++ wordt de code in (SSA) tussenvorm vertaald naar uitvoerbare binaire x86 code[1]. Deze taak wordt in feite geïmplementeerd met behulp van het 'oude' SoftWire project.

Daar waar SoftWire++ een C & HLSL syntaxis ondersteunt, bootst SoftWire de x86 assembleertaal na. Ook hier wordt gebruik gemaakt van klassen voor de datatypes, maar dit maal stemmen ze volledig overeen met de types die in x86 registers geplaatst kunnen worden. Onderstaande code illustreert de implementatie van een fractiebewerking:

[1] *Back-ends* voor andere architecturen zijn uiteraard ook mogelijk.

```
void BackEnd::frc(Float4 &dst, Float4 &src)
{
    static const float4 one4f = {1, 1, 1, 1};

    Float4 tmp0;
    Float4 tmp1;

    Dword2 xy;
    Dword2 zw;

    movaps(tmp0, src);

    cvttps2pi(xy, tmp0);
    movhlps(tmp1, tmp0);
    cvttps2pi(zw, tmp1);
    cvtpi2ps(tmp1, zw);                    free(zw);
    movlhps(tmp1, tmp1);
    cvtpi2ps(tmp1, xy);                    free(xy);
    subps(tmp0, tmp1);
    xorps(tmp1, tmp1);
    cmpnleps(tmp1, tmp0);
    andps(tmp1, xword_ptr [&one4f]);
    addps(tmp1, tmp0);                     free(tmp0);

    movaps(dst, tmp1);                     free(tmp1);
}
```

Fragment 10: Instructieselectie voor frc

Merk op dat dit overeenstemt met de instructieselectiestap van de compiler. Elke functie met de naam van een x86 instructie genereert meteen de corresponderende binaire code.

De `Float4` en `Dword2` types hier zijn verschillend van die uit de *front end* (ze zijn gedefinieerd in hun eigen C++ naamruimte). Ze corresponderen met respectievelijk de SSE en MMX registers. Hun gebruik resulteert in de allocatie van één van de acht (of zestien) fysieke registers. Bij onvoldoende vrije registers wordt er één naar het stapelgeheugen teruggeschreven.

Het gebruikte algoritme voor registerallocatie is gebaseerd op *Linear Scan Register Allocation* [8]. Daarnaast wordt ook een reductie van het stapelgeheugen uitgevoerd. Merk in bovenstaande fragment het gebruik van de `free` functie op. Deze geeft expliciet aan wanneer een tijdelijk register vrijgegeven mag worden. Dit versnelt de *liveness analysis* voor deze registers.

De *back end* voert ook enkele (andere) processorspecifieke optimalisaties uit. Via *peephole* optimalisaties worden sterktereducties uitgevoerd en (lokale) overbodige instructies gedetecteerd en verwijderd.

Tijdens de implementatie van SoftWire++ werd de *back end* uitgebreid met x86-64. Hierbij zijn de registers voor generiek gebruik (gehele getallen en wijzers) 64-bit breed en is het aantal registers verdubbeld (behalve voor MMX). De nodige aanpassingen aan de registerallocatie, het beheer van het stapelgeheugen, de adresseermodes en de functieproloog/epiloog werden aangebracht.

3.6 Cache

De cache is een cruciaal onderdeel om hoge prestaties te behalen. Hij houdt recent gegenereerde functies bij. Voor elke functie wordt ook de bijhorende toestand bijgehouden. Bij ieder tekencommando wordt eerst de huidige toestand vergeleken met de toestanden die zich in de cache bevinden. Wanneer een overeenstemmende toestand gevonden wordt gebruikt men de opgeslagen functie. In het andere geval wordt de functie gegenereerd en samen met de huidige toestand opgeslagen in de cache.

Het opzoeken in de cache is vele malen sneller dan het nieuw genereren van een functie. Het is dan ook belangrijk een hoge kans te creëren dat de functie in de cache gevonden wordt. Anderzijds kan de cache niet te groot zijn: een gegenereerde grafische functie is typisch een tiental kilobytes groot. Toch gebruikt een scène al snel een kleine honderd toestanden (herhaald per beeld).

Een hoge vindkans moet dus mede gerealiseerd worden door een goede vervangstrategie. SoftWire++ vervangt bij het toevoegen van een nieuwe functie telkens de langst niet gebruikte functie. De iets hogere complexiteit van deze strategie ten opzichte van andere loont in de praktijk.

Daarnaast blijkt het ook nodig de opslagruimte van de toestand te minimaliseren. Deze bestaat immers uit een zeer groot aantal parameters en men kan enkel een in de cache opgeslagen functie gebruiken wanneer de hele toestand exact overeenkomt. De eerste aanpak hiervoor is om enkel de relevante toestand op te slaan. Een *pixel shader* functie bijvoorbeeld wordt niet (rechtstreek) beïnvloed door het al dan niet aanwezig zijn van een kleurattribuut in de vertex buffer. De functionaliteit wordt wél beïnvloed door het al dan niet aanwezig zijn van een kleurinterpolant (de uitvoer van de *vertex shader*). Het is dus van groot belang de juiste toestandsparameters te selecteren. Soms zijn dit ook afgeleide parameters.

Het opslaan van enkel de relevante toestandsparameters is momenteel de taak van de programmeur van de grafische bibliotheek. SoftWire++ helpt echter bij de compressie van deze parameters in een minimaal aantal bytes. Toestandsparameters kunnen immers vaak slechts een beperkt aantal waarden aannemen. Denk hierbij aan enumeraties.

Hebben we bijvoorbeeld drie parameters –`a`, `b` en `c`– die zes waarden kunnen aannemen, dan kunnen ze opgeslagen worden in één byte als `a + 6 * (b + 6 * c)`. Bij een achteloze implementatie zouden tot twaalf bytes gebruikt zijn. Merk op dat het delingen en modulobewerkingen vereist om de ‘gecomprimeerde’ versie terug op te delen in afzonderlijke parameters. Dit is echter niet nodig. Voor de cache is de toestand immers een betekenisloos label dat enkel in z’n geheel vergeleken wordt.

Ten slotte wijzen we op het belang van het onderscheid tussen parameters en data. Men zou in principe bijvoorbeeld een *vertex shader* kunnen genereren waarbij de wijzers naar de vertexbuffer ‘hard’ gecodeerd worden. Dit is ook een vorm van specialisatie, waarbij de functie enkel geldig is voor die ene combinatie van *shader* en data. Het is echter duidelijk dat deze aanpak tot een explosie in het aantal functies zou leiden.

Bij het gebruik van SoftWire++ dient dus de werking van de cache in acht genomen te worden om tot goede prestaties te komen.

4
Gerelateerd werk

"Premature optimization is the root of all evil"

— Donald Knuth

4.1 Inleiding

De ideeën die geleid hebben tot SoftWire++ zijn niet uit een vacuüm ontstaan. Voor we het raamwerk op haar waarde kunnen evalueren dienen we dus een overzicht te hebben van gerelateerde projecten en literatuur.

4.2 Zelfaanpassende code

De voorloper van dynamische codegeneratie is zelfaanpassende code, ook gekend als *self-modifying code*. In een Von Neumann-architectuur wordt zowel data als code in hetzelfde geheugen opgeslagen. Er is dus geen wezenlijk verschil tussen de twee; code is data, maar krijgt de betekenis van code door het gebruik als instructiestroom.

Deze architectuur impliceert dat de software via het schrijven van data in het geheugen, op elk moment code kan creëren, of aanpassen. Nieuwe code creëren gebeurt reeds sinds het ontstaan van de eerste compilers (voor assembleerinstructies), maar zelfaanpassende code is mogelijks nog ouder. Het programmeren in binaire code maakt het een kleine stap om reeds geschreven instructies tijdens de uitvoering van de software te overschrijven met andere instructies.

Het doel hiervan was aanvankelijk om geheugenruimte te besparen. Geheugen was duur en elke methode was acceptabel om code in te krimpen. Wanneer bijvoorbeeld twee functies bijna identiek zijn, op één instructie na, is het voordeliger om slechts één functie te schrijven en de betreffende instructie te (over)schrijven vlak voor de aanroep van de functie.

Met het goedkoper worden van geheugen verviel de voornaamste reden voor het gebruik van zelfaanpassende code. De ontwikkeling van geavanceerde compilers en de toenemende complexiteit van software maakten het bovendien lastiger in gebruik. Er ontstond dan ook een sterk afkeurende houding tegenover zelfaanpassende code.

Baanbrekend werk rond zelfaanpassende code kwam er in 1992 met Henry Massalin's doctoraatscriptie [4]. Hierin worden methodes van objecten overschreven met code die de gewenste functionaliteit geeft. Iedere instantie van een object heeft zijn eigen kopie van de methodes, en in de scriptie worden dit soort instanties quajects gedoopt. Ze worden gebruikt om een efficiënte *kernel* voor het UNIX besturingssysteem te implementeren. Deze *kernel* past zich dynamisch aan de behoeften van de draaiende applicaties aan.

De scriptie bracht zelfaanpassende code terug in de aandacht, dit maal als krachtige optimalisatietechniek. Het bijhorende project had echter één groot nadeel: het was volledig geschreven in assembleercode voor een (inmiddels achterhaalde) Motorola processor. Het is dan ook niet zomaar over te zetten, waardoor het ook snel weer uit de belangstelling verdween. Massalin verdedigde zijn keuze door te stellen dat hoog-niveau programmeertalen van dat moment geen dynamische codegeneratie ondersteunen.

Hoewel dit niet volledig correct is was Massalin de eerste die dynamische codegeneratie als sleutel tot hogere prestaties gebruikte. Het was effectief een vorm van dynamische specialisatie. De applicaties bepalen door hun gedrag de 'toestand' van de *kernel*, en de *kernel* specialiseert dynamisch de methodes van de gebruikte objecten (quajects) voor deze toestand.

4.3 Rijgen van code

Het zelf schrijven van binaire instructies is lastig. Men kan echter ook een fragment schrijven in assembleercode, en de compiler inzetten om de binaire code te genereren. Onderstaand fragment illustreert het principe:

```
int stitchAddEaxEbx(void *buffer)
{
    int byteLength;

    __asm
    {
        jmp endStitch
    beginStitch:
        add eax, ebx
    endStitch:
        mov edi, buffer
        mov esi, beginStitch
        mov ecx, endStitch
        sub ecx, beginStitch
        mov byteLength, ecx
        rep movsb
    }

    return byteLength;
}
```

Fragment 11: Rijgen van add eax, ebx

Deze functie plaatst alles wat tussen `beginStitch` en `endStitch` staat, in binaire vorm, in de `buffer`. In dit specifieke voorbeeld schrijft het slechts één instructie, maar het kunnen ook grotere blokken code zijn. Door herhaaldelijk stukken code aan elkaar te rijgen kan een volledige functie opgebouwd worden.

Pixomatic, ook een grafische bibliotheek voor 3D-weergave, maakt gebruik van deze techniek, waar het *code stitching* wordt genoemd [6]. Voor elke bewerking (belichting, bemonstering van textuur, rasterbewerking, enz.) schreef men handmatig de gewenste code. Aan de hand van de toestand van de grafische pijplijn worden dan de gewenste fragmenten aaneengerijgd tot een functie.

Het rijgen van code heeft enkele voordelen. Allereerst is het bijzonder snel. Er is immers geen dynamische compilatiestap meer. De code moet enkel nog in een buffer gekopieerd worden. Ten tweede kan men, ten opzichte van pure zelfaanpassende code, relatief eenvoudig lange en complexe functies genereren. Ten derde en ten laatste kan men de fragmenten zeer sterk manueel optimaliseren. Het is immers exact deze code die in de functie terecht komt.

Hét nadeel van de techniek is echter dat het weinig flexibel is. De registers van de aaneengerijgde fragmenten moeten overeenkomen. De complexiteit neemt hierdoor snel toe wanneer er meer en kleinere fragmenten gebruikt worden. Voor Pixomatic heeft dit tot gevolg dat het geen *shaders* ondersteunt, enkel starre verwerking.

4.4 JIT-compilatie

Een aantal hogere programmeertalen werden aanvankelijk gecompileerd naar bytecode. Dit is een relatief eenvoudige binaire voorstelling van de uit te voeren bewerkingen. De uitvoering zelf gebeurde via een virtuele machine die de bytecodes interpreteert (vergelijkbaar met de emulatie in fragment 3). Een voordeel hiervan is dat de virtuele machine eenvoudig op meerdere platforms te implementeren is en dus de applicatie als bytecode op al deze platformen kan draaien.

Het grootste nadeel van interpretatie is de lage prestaties. Door echter de bytecode dynamisch te compileren naar rechtstreeks uitvoerbare code verkrijgt men prestaties vergelijkbaar met statisch gecompileerde code. Deze techniek wordt ook *Just-In-Time*-compilatie genoemd.

Eén van de grondlegger van JIT-compilatie is Urs Hölzle. Zijn werk rond de dynamische compilatie van objectgeoriënteerde talen [11] droeg rechtstreeks bij aan de doorbraak van de Java programmeertaal [12]. Hij werkte mee aan de Sun HotSpot virtuele machine, die nog steeds als primaire Java virtuele machine dienst doet [13].

Het project dankt zijn naam aan de wijze waarop het hete code optimaliseert. De uitvoering van de Java bytecode gebeurt eerst via interpretatie, en wanneer een fragment vaak uitgevoerd wordt beslist de virtuele machine om hiervoor rechtstreeks uitvoerbare code te genereren. Het loont immers slechts de moeite om dynamische compilatietijd te investeren in code die een reductie in uitvoeringstijd zal opleveren. Dit is ook gekend als adaptieve optimalisatie.

De virtuele machine houdt een cache bij van de vaakst uitgevoerde fragmenten. Bij een wijziging van het gedrag van de applicatie worden de minst frequent gebruikte fragmenten uit de cache verwijderd. De meest frequent uitgevoerde fragmenten ondergaan verdere optimalisatiestappen. Langere code, door agressieve *inlining* en lusontvouwing, is hierbij gerechtvaardigd. Dit kan dus ook gezien worden als een beperkte vorm van specialisatie.

Merk op dat de compilatietijd van de dynamische compiler kritiek is voor de prestaties. Java en andere JIT-gecompileerde talen hebben dan ook heel wat onderzoek naar snelle en doeltreffende optimalisaties op gang gebracht. Eén resultaat hiervan is het eerder vernoemde *Linear Scan Register Allocation* algoritme [8].

Een bijzonder voordeel van dynamische compilatie tegenover statische compilatie is dat het automatisch optimaliseert voor het precieze processormodel. Bij C/C++ is het nog gebruikelijk om voor de zeer oude 386 processor te compileren om compatibiliteit met alle x86 processors te garanderen. Met JIT-compilatie kan een oude applicatie echter gebruik maken van alle instructiesetuitbreidingen die de specifieke processor waar het op gedraaid wordt ondersteund. Enkel de virtuele machine moet af en toe opgewaardeerd worden.

4.5 Automatische specialisatie

Wanneer bij adaptieve optimalisatie ook gebruik gemaakt wordt van variabelen die niet of zeer zelden van waarde veranderen (zogenaamde semi-constanten), kan men hiervoor ook specialiseren tijdens de agressievere optimalisaties. Bij het opzoeken van codefragmenten in de cache moet dus ook nagegaan worden wat de waarden van de semi-constanten zijn. Wanneer ze gewijzigd zijn kan een hercompilatie nodig zijn.

Hierbij wordt één van de volgende strategieën toegepast:

- Eén specialisatie per codefragment: Er wordt vanuit gegaan dat de semi-constanten werkelijk constant zijn voor de hele uitvoering van de applicatie, of zodanig weinig wijzigen dat hercompilatie geen probleem vormt. Variabelen die vaker wijzigen (ook al is het slechts tussen twee waarden) worden niet gebruikt voor specialisatie.
- Meerdere specialisaties per codefragment: De semi-constanten nemen slechts een klein aantal waarden aan, en voor elke waarde (of combinatie van waarden) wordt gespecialiseerde code bijgehouden. Dit heeft echter als nadeel dat de cache gecompliceerder is.

Automatische specialisatie vereist dat de waarden van potentiële semi-constanten gevolgd worden. Merk op dat booleaanse waarden en enumeraties duidelijk betere kandidaten zijn als semi-constante dan gehele getallen en vlottende-kommagetallen. Daarnaast zijn ook functieargumenten interessanter dan tussenwaarden. Een bijzonder geval is memoisatie [14]: het vervangen van functies (zonder neveneffecten) door hun retourwaarden bij gegeven argumenten.

Toch kan niet vermeden worden dat bij een te kleine selectie van potentiële semi-constanten kansen tot nuttige specialisaties gemist worden. Anderzijds zorgt een te grote selectie semi-constanten voor een aanzienlijke meerkost voor het bijhouden van de wijzigingsfrequenties.

De meest geavanceerde projecten proberen dynamisch te bepalen welke variabelen gevolgd moeten worden. Aanvankelijk wordt elke variabele als potentiële semi-constante beschouwd. Tijdens het lopen van de applicatie worden te vaak veranderende variabelen (eventueel ook rekening houdend met hoe veel waarden aangenomen worden) niet meer als semi-constante beschouwd. Deze techniek is echter zeer complex, en eens een variabele niet meer gevolgd wordt mist men latere kansen tot specialisatie.

Deze problemen wordt deels opgelost door potentiële semi-constanten door de applicatieprogrammeur aan te laten duiden in de code via annotaties. Dit is geen volledig

automatische specialisatie meer, maar biedt ook geen garanties. De compiler kan nog steeds beslissen om de code niet te specialiseren. Bovendien worden de annotaties voor semi-constanten vaak slecht begrepen. Dit is niet verwonderlijk aangezien de syntaxis geen voeling geeft met hoe de code effectief uitgevoerd wordt, met en zonder de annotaties.

Hoewel automatische specialisatie via adaptieve JIT-compilatie enig potentieel vertoont, blijven de prestaties vaak achter op die van statisch gecompileerde programma's. In C++ bijvoorbeeld is het relatief eenvoudig om de prestaties te analyseren en handmatig gespecialiseerde functies te schrijven voor vaak voorkomende bewerkingen (zie ook het voorbeeld met matrixvermenigvuldiging in de inleiding). Via automatisch gespecialisatie wordt geen snellere code verkregen.

Bovendien beschikt C++ over statische door de compiler geholpen specialisatie in de vorm van *templates*. Voor elke combinatie van (statische) argumenten zal de compiler een gespecialiseerde instantie van de functie genereren. De argumenten kunnen ook types zijn, zoals we reeds gezien hebben voor de `Pointer` klasse.

Automatische specialisatie wordt dan ook eerder gebruikt ter verbetering van de relatief lage prestaties van JIT-gecompileerde code, dan om de prestaties van statisch gecompileerde code te overstijgen. Bovendien is het in de praktijk niet in staat om code te specialiseren voor een groot aantal parameters. Denk hierbij aan de specialisatie voor een volledige *shader*.

4.6 Expliciete specialisatie

Enkel de programmeur heeft een volledig beeld van de parameters waarvoor het nuttig is te specialiseren. In het geval van een grafische pijplijn weet hij dat er niet veel meer dan een honderdtal verschillende toestanden zullen optreden, en deze bijna ieder beeldje hergebruikt zullen worden. Specialisatie voor de hele toestand kan enkel verkregen worden via het expliciete genereren van nieuwe functies.

Daar waar automatische specialisatie gebruik maakt van één taal, maakt expliciete specialisatie gebruik van een tweede taal om de te genereren code in te specificeren. We zullen dit de dynamische taal noemen, terwijl de code die de gegenereerde functies aanroep in de statische taal geschreven is. De statische en dynamische taal kan dezelfde zijn, maar dit is eerder uitzonderlijk.

Expliciete dynamische specialisatie kent twee vormen, die we externe en interne specialisatie zullen noemen.

Externe specialisatie laat de specialisatie zelf over aan de dynamische compiler. Een voorbeeld hiervan is SoftWire versie 1.x. De ontwikkeling van SoftWire startte in 2001 als een eenvoudige compiler voor assembleercode in tekstvorm. Er was immers nood aan een "lichte" compiler die onmiddellijk de binaire code in het geheugen schrijft, en beschikte over conditionele compilatie die vanuit andere software te controleren was. Onderstaand fragment is een voorbeeld van de dynamische taal die SoftWire 1.3.3 ondersteunde:

```
inline det(a, b, c, d, x)   // Determinant of 2x2 matrix
{
    fld   a
    fmul  b
    fld   c
    fmul  d
    fsubp st1, st0
    fstp  x
}

#define x(v) dword [v+0]
#define y(v) dword [v+4]
#define z(v) dword [v+8]

inline cross(v1, v2, v3)
{
#if !SSE
    det(y(v1), z(v2), z(v1), y(v2), x(v3))   // x component
    det(z(v1), x(v2), x(v1), z(v2), y(v3))   // y component
    det(x(v1), y(v2), y(v1), x(v2), z(v3))   // z component
#else
    movaps xmm2, xword [v1]
    movaps xmm3, xword [v2]
    shufps xmm0, xmm0, 0x09
    shufps xmm1, xmm1, 0x12
    mulps  xmm0, xmm1
    shufps xmm2, xmm2, 0x12
    shufps xmm3, xmm3, 0x09
    mulps  xmm2, xmm3
    subps  xmm0, xmm2
    movaps xword [v3], xmm0
#endif
}

CrossProduct:
    mov eax, [esp+4]
    mov ecx, [esp+8]
    mov edx, [esp+12]

    cross(eax, ecx, edx)

    ret
```

Fragment 12: Externe specialisatie in assembleercode

In dit voorbeeld wordt een functie gegenereerd die een kruisproduct uitrekent. Afhankelijk van de `SSE` variabele maakt deze functie al dan niet gebruik van SSE vectorinstructies. De waarde van deze variabele kan ingesteld worden in de statische taal, vóór de functie gegenereerd wordt. Bemerk dat SoftWire 1.3.3 ook verschillende vormen van macro's ondersteunde.

In principe is het mogelijk eender welke compiler (en taal) te gebruiken voor externe specialisatie, zo lang de code in uitvoerbare vorm in het geheugen terecht kan komen, en we controle hebben over de conditionele compilatie. Voor grafische toepassingen zijn de vectorinstructies echter van zodanig groot belang dat enkel assembleercode voldoet.

Interne specialisatie gebruikt de statische taal om te beslissen welke bewerkingen van de dynamische taal in de gegenereerde functie terechtkomen. Dit is hoe SoftWire vanaf versie 2.0 te werk gaat (inclusief de nieuwe SoftWire++). Een voorbeeld hiervan is te zien in fragment 10. C++ code roept functies aan met de syntaxis van assembleerfuncties, die *run-time intrinsics* gedoopt werden. Daarnaast maakte het ook reeds gebruik van het overladen van de `[]` operator om de syntaxis voor geheugenadressering na te bootsen. Vanaf versie 3.0 werd ook registerallocatie ondersteund, en versie 4.0 voegde eenvoudige 'lineaire' optimalisaties toe [15].

Deze vorm van expliciete specialisatie heeft de eerder genoemde voordelen dat de specialisatie zonder tekstuele analyse kan gebeuren en alle constructies uit de statische taal ter beschikking zijn om de codegeneratie te sturen. Als nadeel geldt wel dat de dynamische taal beperkt is tot wat de statische taal toelaat te abstraheren. Uit het vorige hoofdstuk is echter gebleken dat C++ tot nog toe geen echte beperkingen vertoonde.

Bij interne specialisatie kunnen we nog een onderscheid maken tussen drie vormen van codegeneratie:

- Interne codegeneratie: De code in tussenvorm wordt verder verwerkt tot uitvoerbare code binnen hetzelfde raamwerk. Dit geeft volledige controle over de optimalisaties, zodat enkel de snelle en doeltreffende algoritmes toegepast worden. Het laat ook toe vectorinstructies in de tussenvorm op te nemen.
- Externe codegeneratie via tekstvorm: Hierbij zullen de *run-time intrinsics* of andere codegenererende functies bewerkingen in tekstvorm schrijven. Wanneer alle bewerkingen toegevoegd zijn wordt de tekstuele code door een externe compiler tot een uitvoerbare functie verwerkt. Deze aanpak combineert de kracht van interne specialisatie met de eenvoud van het gebruik van een bestaande (optimaliserende) compiler. Het neemt echter een 'omweg' via de tekstvorm en er is slechts beperkte controle over de optimalisaties. Vectorinstructies zijn zelden goed ondersteund.
- Externe codegeneratie via tussenvorm: Hierbij worden de bewerkingen meteen in de tussenvorm van een bestaande compiler opgeslagen. De 'omweg' via tekstvorm wordt dus vermeden. Een interessante compiler voor deze vorm is LLVM [16], dat ook een beperkte ondersteuning voor expliciete vectorbewerkingen biedt[1].

Voor SoftWire is duidelijk voor interne codegeneratie gekozen, voornamelijk voor de volledige controle over optimalisaties en vectorinstructies.

4.7 Overzicht van raamwerken

Onderstaande tabel geeft een overzicht van waar SoftWire zich bevindt en hoe het geëvolueerd is ten opzichte van andere raamwerken. We beperken ons hierbij tot een aantal typerende projecten voor expliciete dynamische specialisatie:

[1] Wat eventueel uit te breiden is; LLVM heeft open broncode onder een zeer vrije licentie.

Raamwerk	Statische taal	Dynamische taal	Specialisatie	Codegeneratie
SoftWire 1.0	C++	assembleercode	extern	intern
SoftWire 2.0	C++	assembleercode	intern	intern
SoftWire++	C++	C & HLSL	intern	intern
`C	`C	C	extern	intern
TaskGraph	C++	C	intern	extern (SUIF)
VCODE	C/C++	tussenvorm	intern	intern
.NET	C#	C#	extern	extern (C#)
Sh	C++	HLSL	intern	extern (HLSL)

Tabel 2: Enkele raamwerken voor expliciete dynamische specialisatie

De laatste vier raamwerken verdienen elk nog een woordje commentaar binnen de context van grafische toepassingen, vooraleer we overgaan tot de gedetailleerde evaluatie van SoftWire++:

**`C** (*Tick C*) [17] is een uitbreiding van de C programmeertaal waarbij het ` symbool gebruikt wordt om dynamisch te genereren code aan te duiden. Dit is dus explicieter dan de eerder genoemde annotaties voor semi-constanten bij automatische specialisatie. Het grootste nadeel van `C is echter dat het een nieuwe compiler vereist.

TaskGraph [18] maakt net zoals SoftWire gebruik van macro's en het overladen van C++ operatoren om een op C lijkende dynamische taal te definiëren. Het schrijft de uitgevoerde bewerkingen in de SUIF tussenvorm [19], die dan verwerkt kan worden door een compiler die dit formaat ondersteund. SUIF werkt echter niet met vectortypes.

VCODE [20] maakt gebruik van functies die een rechtstreekse voorstelling zijn van de tussenvorm. Dit lijkt sterk op het programmeren in assembleercode. VCODE ondersteunt echter geen registerallocatie (deze complexe taak wordt overgelaten aan de applicatie), en x86 code-emissie ontbreekt.

.NET [21] maakt gebruik van JIT-compilatie en biedt de mogelijkheid om vanuit de code de compiler aan te roepen. Dit kan gebruikt worden om code in tekstvorm te laten compileren tot een soort bibliotheek. Objecten gedefinieerd in de dynamische code kunnen dan geïnstantieerd worden via hun naam.

Sh [22] biedt de mogelijkheid om in C++ *shaders* te schrijven voor de GPU. De *shaders* worden uitgeschreven in GLSL syntaxis (de HLSL-variant van OpenGL) om dan te laten compileren door de OpenGL API. Dynamische controlestructuren zijn niet ondersteund. Het kan ook C code genereren en laten compileren, maar enkel voor vertexverwerking.

Al deze raamwerken (buiten Sh) missen goede ondersteuning van vectorbewerkingen doorheen het hele compilatieproces. Hoewel sommige automatische vectorisatie ondersteunen is dit nog een experimentele techniek met wisselende resultaten. Bovendien zou het een omweg zijn om van vectorcode naar een scalaire tussenvorm te gaan en daar terug vectorbewerkingen uit te halen.

5
Evaluatie en resultaten

"Almost all programming can be viewed as an exercise in caching"

— Terje Mathisen

5.1 Grafische prestaties

SoftWire++ is enerzijds gericht op maximale prestaties en anderzijds op een groot gebruiksgemak. In vorige hoofdstukken werd reeds duidelijk dat de C & HLSL syntaxis gebruiksvriendelijker is ten opzichte van de assembleersyntaxis gebruikt in vorige versies. Er rest ons dus enkel nog een objectieve evaluatie te maken van de prestaties.

Hoewel we SoftWire++ zouden kunnen beoordelen door bijvoorbeeld te meten hoeveel instructie het per seconde kan genereren, werd bewust gekozen dit niet te doen. Het werd niet ontwikkeld om dat soort prestatiematen te maximaliseren, maar wel die van de grafische bibliotheek die er gebruik van maakt. We zullen dus via de evaluatie van SwiftShader het beste beeld krijgen van de capaciteiten van SoftWire++. Merk wel op dat SoftWire++ via ontwerpkeuzes zoals interne specialisatie en interne codegeneratie reeds vele 'omwegen' vermijdt.

Voor de prestatieanalyse van SwiftShader maken we gebruik van de 3DMark evaluatieprogramma's [23]. Deze programma's zijn populair voor het evalueren van de prestaties van grafische hardware. Ze maken gebruik van de Direct3D API en SwiftShader ondersteunt de edities '01 '03 en '05 (maar nog niet editie '06). Op het moment van testen had SwiftShader voor de programmeerbare vertexverwerking de keuze tussen een implementatie via SoftWire[1] of SoftWire++. Dit stelt ons dus in staat om gedeeltelijk de impact van de nieuwe taal en de optimalisaties te beoordelen.

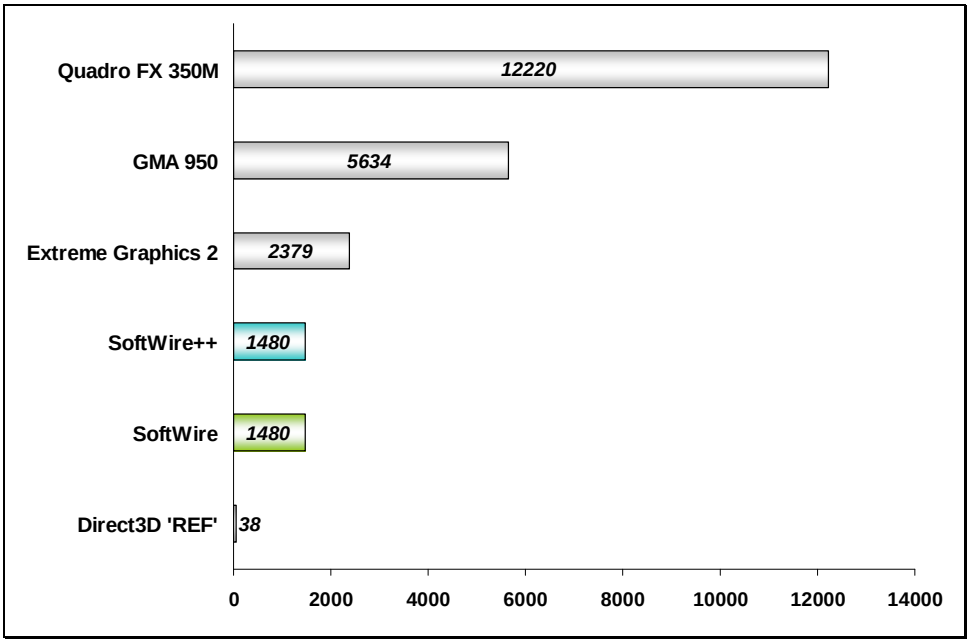
Drie testsystemen werden gebruikt:

- Een Dell Precision M65 laptop met Intel Core 2 Duo T7200 processor van 2,0 GHz. Deze CPU werd gebruikt voor het testen van SwiftShader en de *Direct3D Reference Rasterizer* (REF). Daarnaast is hij uitgerust met een NVIDIA Quadro FX 350M, een matig krachtige GPU, die ook in de tests is opgenomen.
- Een Apple Mac Mini met Intel CPU en geïntegreerde Intel GMA 950 GPU, de voorloper van de eerder vermelde GMA X3000. Met behulp van Boot Camp [24] werden Windows XP en de 3DMark programma's geïnstalleerd.

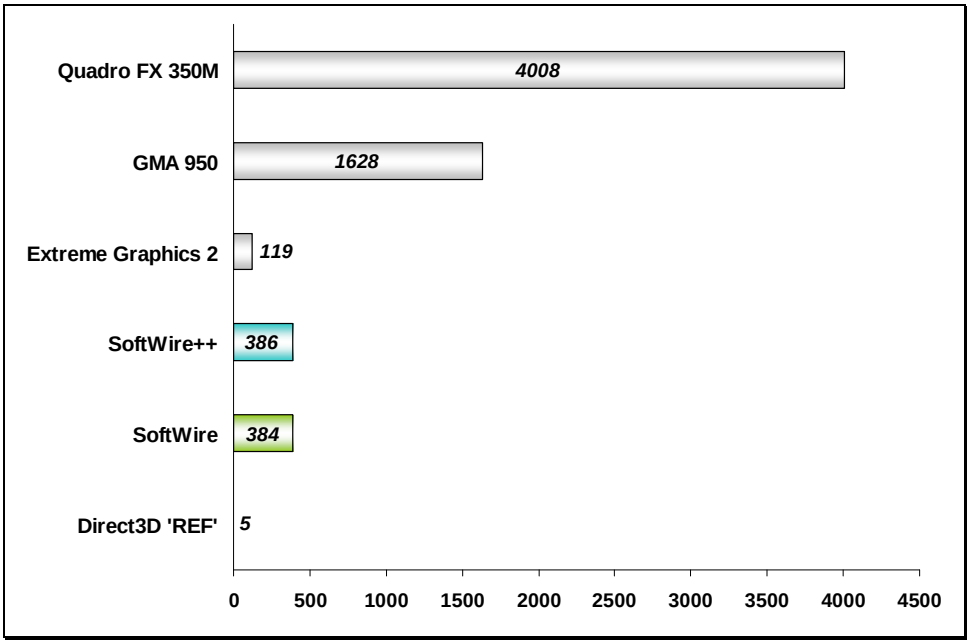
[1] Eigenlijk SwiftAsm, de commerciële doorontwikkeling van SoftWire 4.0.

- Een Dell Inspiron 600m laptop met Intel Extreme Graphics 2 IGP.

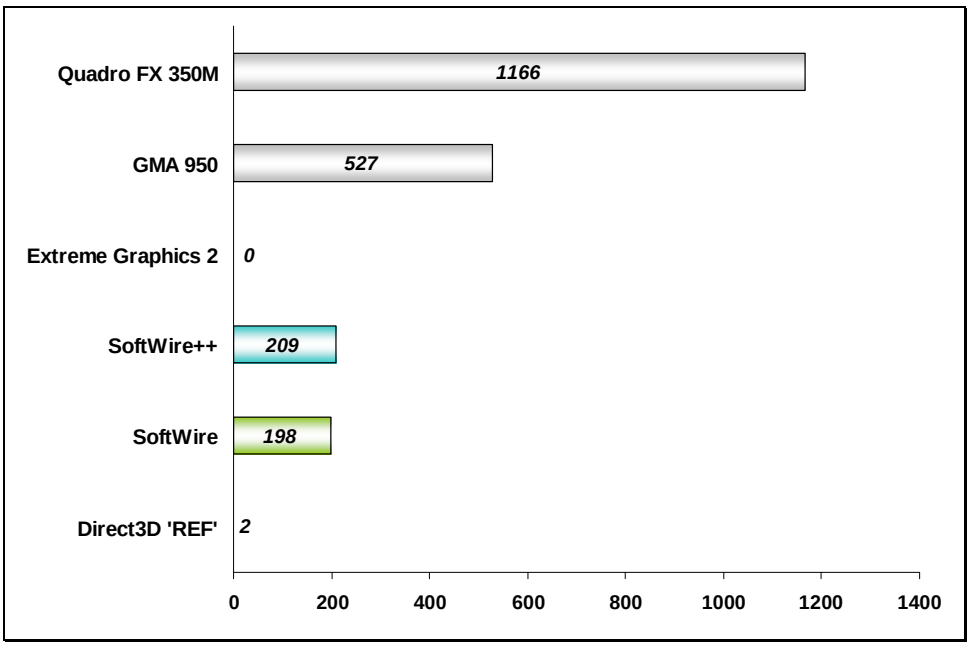
Onderstaande grafieken 1, 2, en 3 vatten de resultaten samen:



Grafiek 4: 3DMark01 scores



Grafiek 5: 3DMark03 scores



Grafiek 6: 3DMark05 scores

De NVIDIA Quadro FX 350M domineert de drie evaluaties. Het is dan ook een recente chip met een efficiënte architectuur gericht op het draaien van moderne 3D-spellen en CAD-toepassingen.

De Intel GMA 950 is ruwweg vier maal sneller dan SwiftShader in 3DMark01 en 3DMark03, maar minder dan drie keer sneller in 3DMark05. Het kleinere verschil in de '05 editie is te verklaren door een duidelijke evolutie naar langere *shaders* met meer rekenkundige bewerkingen en in verhouding minder textuurbemonstering. Dit komt SwiftShader ten goede, althans in relatieve cijfers, door de vele instructies die nodig zijn voor softwarematige textuurbemonstering.

Merk op dat de Core 2 Extreme QX6800 CPU praktisch drie keer meer rekenkracht heeft dan de Intel Core 2 Duo T7200. We kunnen dus verwachten dat de scores behaald door de krachtigste CPU van dit moment sterk kunnen rivaliseren met die van nog steeds veelgebruikte geïntegreerde grafische chips.

De Extreme Graphics 2 is een verouderde IGP die er enkel in slaagt SwiftShader op een T7200 te verslaan in 3DMark01. In 3DMark03 ligt de score veel lager omdat slechts één van de vier tests ondersteund wordt. Bovendien gebeurt de vertexverwerking op de CPU met slecht geoptimaliseerde software. De Extreme Graphics 2 ondersteunt geen enkele test van 3DMark05.

Voor 3DMark01 is er geen verschil tussen het gebruik van SoftWire en SoftWire++. Dit komt doordat het enkel gebruik maakt van starre vertexverwerking[1]. In 3DMark03 is een kleine winst waar te nemen voor SoftWire++, en voor 3DMark05 is het verschil nog iets duidelijker. Deze winst is het gevolg van de optimalisaties in de *middle-end*. Alle *run-time intrinsics* werden probleemloos vervangen door hun equivalent in HLSL syntaxis, waardoor er enkel een prestatiewinst kon zijn. De optimalisaties elimineren dode code en

[1] Het heeft ook tests met shaders, maar die worden niet bij de hoofdscore opgeteld.

voeren sterktereducties uit die niet via conditionele compilatie verkregen werden. 3DMark05 heeft langere vertex-*shaders* en meer geometrie, wat het verschil in winst verklaart.

Tot slot is de *Direct3D Reference Rasterizer* de enige andere software die 3DMark kan draaien, maar liggen de prestaties extreem laag. De tests duurden soms uren en vaak werd niet meer dan één beeld geproduceerd.

5.2 Kwaliteit en speelbaarheid

3DMark is ontworpen om de nieuwste generaties grafische kaarten met elkaar te kunnen vergelijken. Het hoeft dan ook niet te verwonderen dat SwiftShader tijdens de 3DMark tests vaak niet meer dan enkele beelden per seconde produceerde. In echte spellen kan men echter het detail aanpassen om systemen met lage prestaties ook voldoende vlot de software te laten draaien. SwiftShader beschikt ook over de mogelijkheid om de kwaliteit enigszins te verlagen om de prestaties te verhogen (niet gebruikt in bovenstaande 3DMark tests).

Onderstaande schermafdrukken werden gemaakt tijdens het spelen van Half-Life 2 [25] met verlaagd detail. Het aantal beelden per seconde werd in de rechterbovenhoek afgebeeld met behulp van FRAPS [26]:

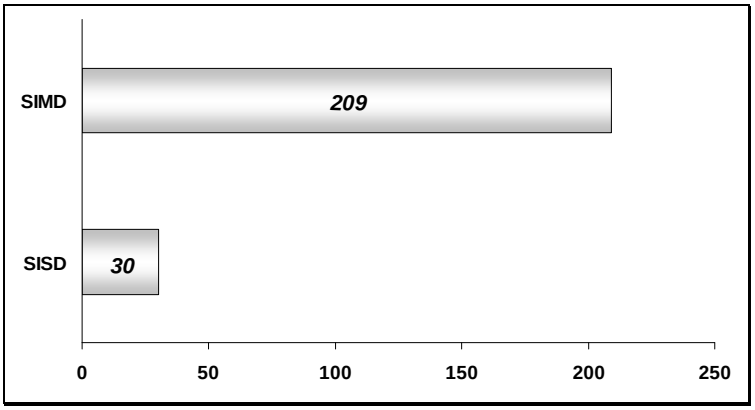
Figuur 19: Half-Life 2 eenvoudige scène

Figuur 20: Half-Life 2 complexe scène

Hoewel er regelmatig minder dan 30 beelden per seconde getoond konden worden bleef het speelbaar. Meerdere niveaus werden succesvol uitgespeeld. Met een krachtigere CPU (zoals de QX6800) wordt verwacht dat wel 30 beelden per seconde aangehouden kan worden, en de kwaliteit niet op de minimuminstelling moet gehouden worden.

5.3 Invloed van vectorbewerkingen

De versie van SoftWire gebruikt in SwiftShader ondersteunt ook de vertaling van vectorbewerkingen naar scalaire bewerkingen voor de x87 FPU. Dit werd tijdens het onderzoek van deze scriptie geïmplementeerd om ook zeer oude CPU's te ondersteunen, en laat ons toe de impact van vectorinstructies te evalueren:



Grafiek 7: 3DMark05 scores SIMD/SISD

Het kan verbazen dat de prestaties voor scalaire verwerking (SISD) meer dan vier keer lager liggen dan de vectoriële verwerking (SIMD). Dit is te verklaren doordat er ook vier keer minder registerruimte is voor scalaire verwerking. De acht registers bieden slechts ruimte voor twee vectoren van vier componenten, en er moet dus zeer vaak data uit het geheugen gehaald worden en later weer weggeschreven worden.

6
Besluit

"The best performance improvement is the transition from the nonworking state to the working state"

— John Ousterhout

6.1 Conclusies

In deze scriptie onderzochten we de mogelijkheid om 3D-weergave niet in hardware maar in software uit te voeren. Hiervoor werd gebruik gemaakt van het dynamische genereren van gespecialiseerde verwerkingsfuncties. Deze techniek werd verder uitgebreid met een hogere programmeertaal en optimalisaties.

Concreet kunnen we volgende zaken concluderen:

- De rekenkracht van moderne CPU's volstaat om verscheidene 3D-applicaties te ondersteunen.
- Dynamische specialisatie is onontbeerlijk om de CPU goed te benutten. Door de specialisatie en codegeneratie binnen C++ uit te voeren beschikt men over krachtige abstracties en snelle compilatie.
- Een dynamische programmeertaal gebaseerd op C & HLSL maakt de implementatie van de grafische bibliotheek eenvoudiger. De vectorsyntaxis van HLSL verzekert dat geen merkbare prestaties verloren gaan tijdens de vertaling.
- Optimalisaties in de tussenvorm maken het eenvoudiger efficiënte code te genereren. Bovendien zijn ze effectiever dan manuele optimalisaties.

6.2 Toekomstig werk

SoftWire++ is op het moment van schrijven nog een prototype. Het werd reeds succesvol gebruikt voor vectorverwerking en het is een kwestie van tijd om het ook in de andere trappen van de grafische pijplijn te gebruiken. De verwachting is dat het ook daar de complexiteit verlaagt en een prestatieverhoging oplevert.

Een idee voor snellere codegeneratie dat hier niet onderzocht werd is het gebruik van metaprogrammeren met behulp van C++ *templates*. Deze techniek zou kunnen toelaten om een deel van het werk reeds tijdens de statische compilatiestap te laten gebeuren.

We zouden ook de optimalisatie in fases kunnen laten verlopen, zoals bij adaptieve compilatie. Weinig gebruikte functies (*shaders*) kunnen bijvoorbeeld enkele van de

minder effectieve *peephole* optimalisaties overslaan. Veel gebruikte functies kunnen zwaardere optimalisatiestappen ondergaan.

Er werd opgemerkt dat het belangrijk is een onderscheid te maken tussen parameters waarvoor gespecialiseerd kan worden, en data die uit het geheugen moet gelezen worden, om geen te groot aantal functies te moeten genereren en opslaan. Bijvoorbeeld de adressen van texturen veranderen (veel) vaker dan typische toestandsparameters. Toch kunnen we ook hiervoor enige specialisatie verkrijgen, met behulp van zelfaanpassende code. Meer bepaald kunnen we de adressen in de (dynamische) code laten schrijven vlak voor ze uitgevoerd wordt.

Het doorgeven van semi-constanten aan de cache (voor de labels) is momenteel ook de taak van de programmeur. Fouten hierin leiden tot zeer lastige problemen die vaak pas laat opgemerkt worden. Het kan dus interessant zijn om te onderzoeken hoe de labels volautomatisch beheerd kunnen worden. Eén idee daarvoor is om de onverwerkte lijst met bewerkingen, in gecomprimeerde vorm, als label te gebruiken. Men genereert dus als het ware de functies telkens opnieuw, maar wanneer de lijst met bewerkingen overeenstemt met één die zich reeds in de cache bevindt wordt de bijhorende volledig geoptimaliseerde en rechtstreeks uitvoerbare functie gebruikt.

Verder hebben we in deze scriptie niet het testen en debuggen van dynamisch gegenereerde code beschouwd. Zelfs met een grondige kennis van assembleercode is dit een lastige taak. Toch is ook hier een mogelijkheid om dit veel eenvoudiger te maken: We hebben opgemerkt dat de code om een functie te genereren zeer sterk overeenkomt met de emulatiecode. Door dus de dynamische controlestructuren te vervangen door hun statische equivalent en de bewerkingen op veranderlijken werkelijk uit te voeren in plaats van hun corresponderende code te genereren verkrijgen we dus een emulator die wél eenvoudig te debuggen is.

Het aanroepen van andere functie binnen een gegenereerde functie wordt nog niet ondersteund. Daarnaast ontbreekt ook de mogelijkheid om datastructuren als samenhangend geheel te beschouwen. Men dient zelf de posities van componenten in het geheugen te kennen. Voorlopig zijn dit geen noemenswaardige beperkingen voor grafische taken, maar het zou van groot nut kunnen zijn voor andere toepassingen.

Zulke toepassingen zijn bijvoorbeeld encryptie, videobewerking en *scripting*. SoftWire werd reeds met enig succes toegepast in projecten met zo'n toepassing. We zouden dus kunnen onderzoeken hoe SoftWire++ ook tegemoet kan komen aan de noden van andere projecten. Het zou ook interessant zijn te weten of er nog meer toepassingen te bedenken zijn die er goed gebruik van kunnen maken.

6.3 Perspectieven

Met elke generatie CPU's wordt het aantrekkelijker om 3D-weergave in software uit te voeren. Het verhogen van de kloksnelheid is niet langer meer de primaire aanpak om meer rekenkracht te realiseren, maar de huidige evolutie naar meerkernige CPU's is minstens even interessant. Grafische verwerking bevat van nature zeer veel parallellisme.

We merken nogmaals op dat de CPU en GPU steeds sterker op elkaar gaan gelijken. Moderne GPU's zijn in staat zowel met gehele getallen als vlottende-kommagetallen te

rekenen, en ondersteunen relatief generieke controlestructuren en datatoegang. De CPU verhoogt op zijn beurt het parallellisme en wordt uitgerust met vectorinstructies die duidelijk geïnspireerd zijn op die van de GPU.

Het is dan ook enkel een kwestie van tijd voordat de twee geïntegreerd worden. Zowel Intel als AMD hebben plannen om CPU's te produceren met nog meer kernen, waarvan één of meer dienst kan doen als GPU of een meer generieke krachtige coprocessor. Het is echter nog onbekend in welke mate deze kernen (al dan niet) zouden verschillen van de klassieke kernen. Hoe dan ook zullen ze voor de programmeur toegankelijk zijn en zal dynamische specialisatie een veelgebruikte techniek zijn om hoge prestaties te behalen.

Tot slot is er nog zeer veel onaangeboord potentieel binnen de grafische bibliotheek. De stuurprogramma's van GPU's worden voltijds geoptimaliseerd door hele teams getalenteerde programmeurs. SwiftShader startte echter als hobbyproject en is slechts recentelijk gecommercialiseerd en deeltijds verder ontwikkeld door de auteur. Vooral algoritmes om de *overdraw* te reduceren kunnen nog een aanzienlijke prestatiewinst opleveren.

Appendix A: Homogene coördinaten

Homogene 3D coördinaten laten toe om niet-lineaire transformaties voor te stellen met matrices. Ze hebben vier componenten, die doorgaans *x*, *y*, *z* en *w* genoemd worden, en de matrices zijn van dimensie 4×4.

Een eerste niet-lineaire transformatie die met homogene coördinaten in één matrix voorgesteld kan worden is de affiene transformatie. In cartesiaanse coördinaten hebben we een 3×3 matrix *M* nodig met elementen m_{ij} en een afzonderlijke translatievector *t*:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

Met homogene coördinaten kunnen we dit schrijven als:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & t_x \\ m_{21} & m_{22} & m_{23} & t_y \\ m_{31} & m_{32} & m_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Merk op dat de homogene component (*w*) hier gelijk gesteld wordt aan 1, en ook voor de resulterende vector geldt $w' = 1$. Het belang van de homogenisering is dat men nu verschillende 4×4 matrices, bestaande uit rotaties, schaling, translaties, enz. kan vermenigvuldigen. De resulterende matrix stelt dan al deze transformaties samen voor. In de context van computergrafiek komt dit de efficiëntie ten goede omdat een groot aantal punten getransformeerd moeten worden en dit slechts één matrix meer vergt.

Naast een elegante en efficiënte voorstelling van translaties worden homogene coördinaten ook gebruikt voor perspectief. Objecten worden kleiner afgebeeld naarmate ze zich verder van de camera bevinden. Wanneer *z* de afstand tot het beeldvlak voorstelt gelden volgende formules:

$$X = x / z$$
$$Y = y / z$$

Hierbij vormt *(X, Y)* de coördinaten in het tweedimensionale beeldvlak. Deze formules kunnen voorgesteld worden in een matrix met behulp van homogene coördinaten, door per definitie volgende projectie aan te nemen:

$$\begin{pmatrix} x & y & z & w \end{pmatrix} \mapsto \begin{pmatrix} x/w & y/w & z/w \end{pmatrix} \equiv \begin{pmatrix} X & Y & Z \end{pmatrix}$$

De perspectieftransformatie kan dan geschreven worden als:

$$\begin{bmatrix} X \\ Y \\ 1/z \end{bmatrix} \Leftarrow \begin{bmatrix} x \\ y \\ 1 \\ z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} . \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Merk op dat de homogene component hier niet langer 1 blijft. Anderzijds zijn homogene coördinaten met $w = 1$ triviaal te projecteren naar cartesiaanse coördinaten door de vierde componenten te laten vallen. De definitie van de projectie werkt dus ook voor bovenstaande affiene transformatie.

Naast *(X, Y)* krijgen we bij de perspectieftransformatie ook $1/z$. Dit wordt doorgaans gebruikt als waarde om op te slaan in de *z*-buffer (zie sectie 2.3.3 pagina 13). Het opslaan van $1/z$ zorgt voor minder precisieproblemen dan het opslaan van *z*.

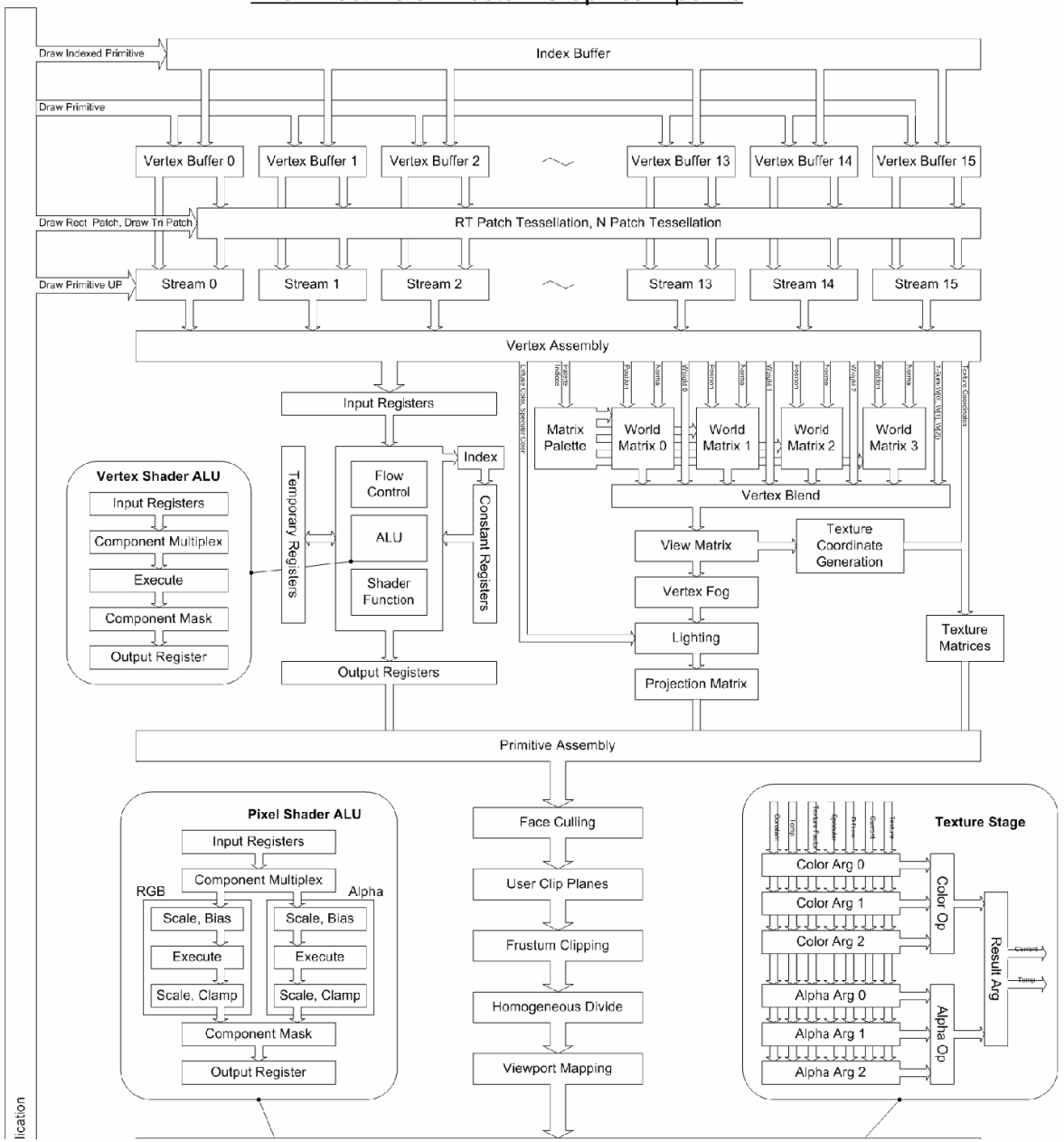
Affiene transformaties en perspectief kunnen dankzij het gebruik van homogene coördinaten ook samen als één 4×4 matrix voorgesteld worden. De vertices van objecten kunnen dus onmiddellijk van het coördinatensysteem van het object naar schermcoördinaten getransformeerd worden.

Appendix B: Direct3D 9 poster

Op de volgende twee pagina's wordt praktisch de hele Direct3D 9 architectuur afgebeeld. De poster is afkomstig van http://www.xmission.com/~legalize/book/preview/poster.

Deze bijlage dient louter ter illustratie van de grote complexiteit en de vele toestandsparameters van een moderne grafische API.

The DirectX 9.0 Direct3D Graphics Pipeline



Methods	Capabilities	Render, Sampler & Texture Stage States
CreateIndexBuffer DrawIndexedPrimitive SetIndices	Max Primitive Count Max Vertex Index	
DrawPrimitive	Dev Caps_Draw Prim TL Vertex Max Primitive Count	
CreateVertexBuffer SetStreamSource SetStreamSourceFreq	Dev Caps_TL Vertex System Memory Dev Caps_TL Vertex Video Memory Dev Caps 2_Stream Offset Dev Caps 2_Vertex Elements Can Share Stream Offset Max Streams FVF Caps Max Stream Stride Decl Types	
DeletePatch DrawRectPatch DrawTriPatch SetNPatchMode SetVertexDeclaration	Dev Caps_RT Patches Dev Caps_Cubic RT Patches Dev Caps_RT Patch Handle Zero Dev Caps 2_Adaptive Tess RT Patch Dev Caps 2_Adaptive Tess N Patch Dev Caps 2_D Map N Patch Max N Patch Tessellation Level	RS Adaptive Tess X, Y, Z, W RS Enable Adaptive Tessellation RS Max Tessellation Level RS Min Tessellation Level RS Normal Degree RS Patch Edge Style RS Position Degree
DrawIndexedPrimitiveUP DrawPrimitiveUP	Max Primitive Count	
SetFVF SetRenderState SetVertexDeclaration SetVertexShader SetSoftwareVertexProcessing	Dev Caps_HW Transform And Light Vertex Processing Caps_Tweening	RS Color Vertex RS Point Size
MultiplyTransform SetRenderState SetTransform	Max Vertex Blend Matrices Max Vertex Blend Matrix Index	RS Tween Factor RS Indexed Vertex Blend Enable RS Vertex Blend
CreateVertexShader	Vertex Shader Version VS 2.0 Caps Vertex Texture Filter Caps Max V Shader Instructions Executed Max Vertex Shader 3.0 Instruction Slots	
MultiplyTransform SetVertexShaderConstant SetRenderState SetTextureStageState SetTransform	Vertex Processing Caps_No Tex Gen Non Local Viewer Vertex Processing Caps_Tex Gen Vertex Processing Caps_Tex Gen Sphere Map Max Vertex Shader Const	TSS Tex Coord Index
SetRenderState	Raster Caps_Fog Range Raster Caps_Fog Vertex	RS Fog Vertex Mode RS Range Fog Enable
SetTransform MultiplyTransform SetLight SetLightEnable SetMaterial	Vertex Processing Caps_Directional Lights Vertex Processing Caps_Local Viewer Vertex Processing Caps_Material Source 7 Vertex Processing Caps_Positional Lights Max Active Lights	RS Ambient RS Ambient Material Source RS Diffuse Material Source RS Emissive Material Source RS Local Viewer RS Lighting RS Normalize Normals RS Shade Mode RS Specular Material Source TSS Texture Transform Flags
MultiplyTransform SetTransform		
SetRenderState	Primitive Misc Caps_Fog Vertex Clamped	RS Point Sprite Enable RS Point Size Min, Max RS Point Scale Enable RS Point Scale A, B, C
SetRenderState	Primitive Misc Caps_Cull CW Primitive Misc Caps_Cull CCW Primitive Misc Caps_Cull None	RS Cull Mode
SetRenderState SetClipPlane	Primitive Misc Caps_Clip Plane Scaled Points Max User Clip Planes	RS Clip Plane Enable
SetRenderState SetClipStatus	Primitive Misc Caps_Clip TL Verts	RS Clipping
SetViewport	Dev Caps_HW Rasterization Line Caps_Antialias Line Caps_Texture Primitive Misc Caps_Fog And Specular Alpha Raster Caps_Color Perspective Raster Caps_Slope Scale Depth Bias	RS Fill Mode

Application
Rasterization and Component Interpolation
Input Registers
Diffuse
Specular
Texture Coordinates
Constant Registers
Flow Control
ALU
Shader Function
Temporary/Output Registers
Texture Registers
Texture Addressing
Texture Sampling
Textures
Texture Coordinates
Texture 0
Texture 1
Texture 7
Bump Mapping
Texture Stage 0
Texture Stage 1
Texture Stage 7
Texture Factor
Constant 0
Constant 1
Constant 7
Add Specular
Color, Alpha
Depth, Fog Factor
Color, Alpha, Depth
Fog Blend
Scissor Test
Alpha Test
Depth/Stencil Test
Alpha Blend
Dither
Channel Mask
Render Target
Presentation
Video Scanout
Cursor
Gamma Ramp
D/A Converter
CRT Monitor
Scissor Rect
Alpha Ref
Color, Alpha, Depth, Stencil
Color
Reference
Mask
Test Source
Test Dest
Stencil Function
Stencil Fail Operation
Depth Function
Stencil Pass Operation
Stencil Z Fail Operation
Depth/Stencil Test
Source Color
Destination Color
Src Blend
Dest Blend
Blend Op
Source Alpha
Destination Alpha
Src Blend Alpha
Dest Blend Alpha
Blend Op Alpha
Alpha Blending
Device Resources
Index Buffer
Vertex Buffer
Volume
Volume Texture
Surface
Texture
Cube Texture
Lock, Unlock
Lock Box, Unlock Box
Lock Rect, Unlock Rect
This chart depicts the same logical function of the pipeline and not any particular implementation.
Methods: Most Get and object creation (CreateRenderTarget, etc.) methods omitted for clarity.
SetPixelShaderConstant and SetVertexShaderConstant refer to all three forms (B, F and I) of the method.
Capability bits: Hallmark bit flags are listed before relevant D3DCAPS9 members.
Parameter flag name definitions: "D3DCAPS2: MemberName" & D3DCAPS3: MemberName"
"Primitive Misc Caps_Blend Op" - "D3DCAPS9: PrimitiveMiscCaps & D3DPMISCCAPS_BLENDOP"
"VS 2.0 Caps" - "D3DCAPS9: VS20Caps"
Render Samples & Texture Stage States:
"RS Name" denotes "D3DRS_NAME", "RS Z Write Enable" - "D3DRS_ZWRITEENABLE"
"SS Name" denotes "D3DSAMP_NAME", "SS Border Color" - "D3DSAMP_BORDERCOLOR"
"TSS Name" denotes "D3DTSS_NAME", "TSS Color Arg 1" - "D3DTSS_COLORARG1"
SetPixelShader
SetRenderState
CreatePixelShader
SetCurrentTexturePalette
SetPaletteEntries
SetPixelShader
SetPixelShaderConstant
SetRenderState
SetTexture
SetTextureStageState
SetRenderState
SetScissorRect
SetRenderState
SetRenderState
Clear
ColorFill
SetDepthStencilSurface
SetRenderState
SetRenderTarget
SetMaterial
LightEnable
UpdateTexture
Present
SetDialogBoxMode
GetRasterStatus
SetCursorPosition
SetCursorProperties
SetGammaRamp
ShowCursor
Line Caps_Texture
Primitive Misc Caps_Fog And Specular Alpha
Raster Caps_Color Perspective
Raster Caps_Slope Scale Depth Bias
Texture Caps_Projected
Extents Adjust
Guard Band Left, Top, Right, Bottom
Max Pixel Shader Value
Max Point Size
Max Vertex W
Pixel Shader Version
Shade Caps
Caps 2_Can Auto Gen Mip Map
Dev Caps_Separate Texture Memories
Dev Caps_Texture Non Local Vid Mem
Dev Caps_Texture System Memory
Dev Caps_Texture Video Memory
Primitive Misc Caps_Per Stage Constant
Raster Caps_Anisotropy
Raster Caps_Mip Map LOD Bias
Max Anisotropy
Max Texture Aspect Ratio
Max Texture Height
Max Texture Width
Max Simultaneous Textures
Max Texture Blend Stages
Max Texture Repeat
Texture Caps
Texture Op Caps
Texture Filter Caps
Cube Texture Filter Caps
Volume Texture Filter Caps
Pixel Shader 1.x Max Value
PS 2.0 Caps
Max P Shader Instructions Executed
Max Pixel Shader 30 Instruction Slots
RS Specular Enable
RS Fog Enable
RS Fog Color
RS Fog Start
RS Fog End
RS Fog Density
RS Fog Table Mode
RS Scissor Test Enable
RS Alpha Test Enable
RS Alpha Ref
RS Alpha Func
RS Stencil Enable
RS Stencil Func
RS Stencil Ref
RS Stencil Mask
RS Stencil Fail
RS Two Sided Stencil Mode
RS CCW Stencil Fail
RS CCW Stencil Pass
RS Src Blend
RS Dest Blend
RS Blend Op
RS Separate Alpha Blend Enable
RS Alpha Blend Enable
RS Blend Factor
RS Dither Enable
RS Color Write Enable
RS Color Write Enable 1
RS Color Write Enable 2
RS Color Write Enable 3
RS sRGB Write Enable
RS Multisample Antialias
RS Multisample Mask
RS Stencil Write Mask
RS Z Write Enable
RS Fill Mode
RS Last Pixel
RS Depth Bias
RS Slope Scale Depth Bias
RS Specular Enable
RS Texture Factor
RS Wrap 0-15
SS Address U, V, W
SS Border Color
SS Mag Filter
SS Min Filter
SS Mip Filter
SS Mip Map LOD Bias
SS Max Mip Level
SS Max Anisotropy
TSS Tex Coord Index
TSS Texture Transform Flags
TSS Color Arg 0, 1, 2
TSS Color Op
TSS Alpha Arg 0, 1, 2
TSS Alpha Op
TSS Constant
TSS Result Arg
TSS Bump Env Mat
TSS Bump Env L Scale
TSS Bump Env L Offset
Line Caps_Fog
Raster Caps_Fog Table
Raster Caps_Fog Vertex
Raster Caps_Z Fog
Raster Caps_Scissor Test
Line Caps_Alpha Comp
Alpha Comp Caps
Line Caps_Z Test
Raster Caps_Buffer Hierarchy HSR
Raster Caps_Z Test
Stencil Caps
Z Comp Caps
Caps 3_Alpha Fullscreen Flip Op Discard
Line Caps_Blend
Primitive Misc Caps_Blend Op
Primitive Misc Caps_MRT Post Pixel Shader Blending
Primitive Misc Caps_Separate Alpha Blend
Src Blend Caps
Dest Blend Caps
Raster Caps_Dither
Primitive Misc Caps_Color Write Enable
Primitive Misc Caps_Independent Write Masks
Caps 3_Copy To Vid Mem
Caps 3_Copy To System Mem
Primitive Misc Caps_Mask Z
Primitive Misc Caps_MRT Independent Bit Depths
Raster Caps_Multisample Toggle
Raster Caps_Wireframe
Stretch Rect Filter Caps
Caps 3_Linear To sRGB Presentation
Dev Caps_Can Render After Flip
Presentation Intervals
Caps_Read_Scanline
Caps 2_Can Calibrate Gamma
Caps 2_Full Screen Gamma
Cursor Caps
Max Num Simultaneous RTs
Adapter Ordinal In Group
Number Of Adapters In Group

Appendix C: SoftWire++ prototype

Bij de gebonden papieren versie van deze scriptie komt een CD met daarop de bronbestanden van het SoftWire++ prototype. Het prototype of een recentere versie kan ook verkregen worden via:

http://studwww.ugent.be/~ncapens/
https://gna.org/projects/softwire/

Referenties

[1] *SwiftShader software 3D renderer*, TransGaming Technologies, http://www.transgaming.com/swiftshader.

[2] Nicolas Capens, *SoftWire*, https://gna.org/projects/softwire/.

[3] Stuart F. Oberman en Michael Y. Siu, *A High-Performance Area-Efficient Multifunction Interpolator*, IEEE Symposium on Computer Arithmetic 2005: 272-279.

[4] Henry Massalin, *Synthesis: An Efficient Implementation of Fundamental Operating System Services*, Ph.D. thesis, Colombia University, 1992.

[5] Andrew W. Appel, *Modern Compiler Implementation in C*, Cambridge University Press, 1998, ISBN 0-521-60765-5.

[6] *Pixomatic*, RAD Game Tools, Inc. http://www.radgametools.com/pixomain.htm.

[7] Keith D. Cooper en Linda Torczon, *Engineering a Compiler*, Morgan Kaufmann, 2003, ISBN 1-55860-698-X.

[8] M. Poletto en V. Sarkar, *Linear Scan Register Allocation*, TOPLAS 1999.

[9] Nicolas Capens, *x86 Shaders–ps_2_0 Shaders in Software*, verschenen in ShaderX^2, bewerkt door Wolfgang Engel, Wordware Publishing, Inc. 2004, 396-412, ISBN 1-55622-988-7.

[10] Nicolas Capens, *Software Shaders and DirectX DLL Implementation*, verschenen in ShaderX^3, bewerkt door Wolfgang Engel, Charles River Media, 2005, 279-311, ISBN 1-58450-357-2.

[11] Urs Hölzle, Craig Chamers en David Ungar, *Optimizing Dynamically-Typed Object-Oriented Programming Languages with Polymorphic Inline Caches*, ECOOP '91 proceedings, Springer Verlag Lecture Notes in Computer Science 512, July, 1991.

[12] *Java*. Sun Microsystems, Inc. http://www.java.com/.

[13] *The Java HotSpot Performance Engine Architecture*. Sun Microsystems, Inc. http://java.sun.com/products/hotspot/whitepaper.html.

[14] Marty Hall en J. Paul McNamee, *Improving Software Performance with Automatic Memoization*, Johns Hopkins APL Technical Digest, 18(2), april 1997.

[15] Nicolas Capens, *Linear Optimization: Implementing an efficient high-performance back-end compiler for Just-In-Time compilation*, http://www.devmaster.net/articles/linear-optimization/.

[16] *The LLVM Compiler Infrastructure Project*. http://llvm.org/.

[17] Dawson R. Engler, Wilson C. Hsieh en M. Frans Kaashoek, *`C: A Language for High-Level, Efficient, and Machine-Independent Dynamic Code Generation*, Thirteenth ACM Symposium on Principles of Programming Languages, 1996, 131-144.

[18] Olav Beckmann, Alastair Houghton, Michael Mellor en Paul H. J. Kelly, *Run-time code generation in C++ as a foundation for domain-specific optimization*, Domain-Specific Program Generation, 2003, 291-306.

[19] *The SUIF Compiler Infrastructure*. http://suif.stanford.edu/.

[20] Dawson Engler, *VCODE: A Retargetable, Very Fast Dynamic Code Generation System*, SIGPLAN Conference on Programming Language Design and Implementation, 1996.

[21] *Dynamic Source Code Generation and Compilation*, Microsoft Corporation, http://msdn2.microsoft.com/en-us/library/650ax5cx.aspx.

[22] *Sh Embedded Metaprogramming Language*, RapidMind Inc. http://libsh.org/.

[23] *3DMark Benchmarks*, Futuremark Corporation, http://www.futuremark.com/.

[24] *Boot Camp*, Apple Inc. http://www.apple.com/macosx/bootcamp/.

[25] *Half-Life 2*, Valve Corporation, http://half-life2.com/.

[26] *FRAPS*, http://www.fraps.com/.