A shader unit

Architecture, OpenGL-specific aspects, simulator implemented using SystemC, adaptions for embedded systems

Philipp Klaus Krause

January 27, 2008

Contents

Chapter 1

Introduction

1.1 Computer graphics

Computer graphics is the subfield of computer science concerned with the creation of digital images using computers.

The first computer known to have graphical input/output and thus computer graphics capabilities was the Whirlwind I, built at the Massachusetts Institute of Technology in the late 40s and early 50s. It used a cathode ray tube to display geographical and aircraft movement information and a light pen for input.

From the beginning [26] video games used graphical output, so computer graphics naturally became a necessity for video games using computers.

Computer graphics found their place in many areas, from computer aided design, games, movies to graphical user interfaces.

1.2 Graphics hardware

The rising quality of computer graphics required special hardware. Initially graphics output devices started to natively support simple 2D operations like superimposing sprites on a background. Such devices found widespread adoption in second generation video game consoles such as the Colecovision and the Intellivision in the early 80s.

Hardware acceleration for 3D graphics was first proposed in 1982 [7] and soon implemented in the SGI IRIS workstations. In the 90s consumer-level graphics cards gained 3D acceleration. Today it is being adopted in embedded systems such as smartphones.

Hardware for 3D graphics today is typically structured like the rendering pipeline in figure 1.1 (see figure 4.1 for a more details): Its input are vertices (leftmost in figure 1.2, mostly the positions of corners of polygons and additional data associated with them). The hardware first performs some transformations of these vertices (associated data including position can change, but not the number of vertices) and then rasterizes the corresponding polygons, creating

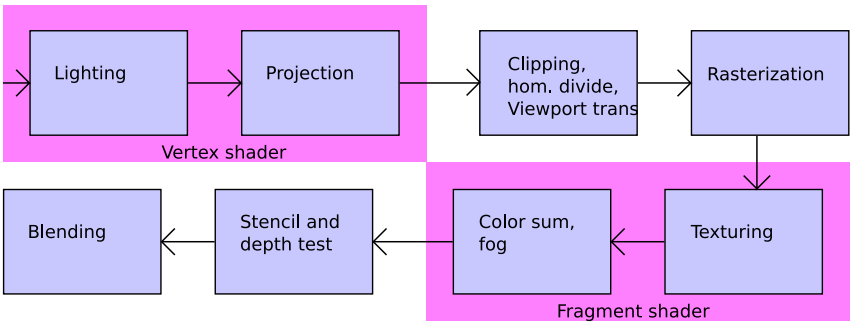

Figure 1.1: Rendering pipeline

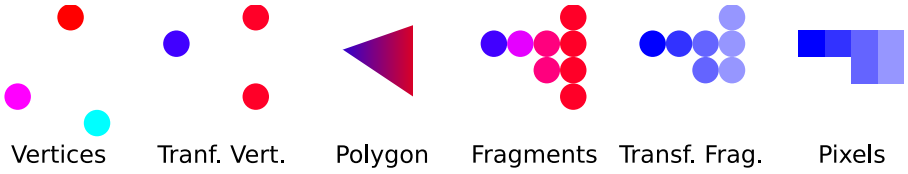

Figure 1.2: Data in the rendering pipeline

fragments. The fragments are further transformed (associated data, excluding position can be changed, fragments can be thrown away) and finally end up in the framebuffer as pixels (rightmost in figure 1.2).

However while compared to software rendering hardware acceleration provided vast speed improvements, flexibility was lost: New realtime rendering algorithms in computer graphics could be used only when hardware manufacturers implemented them.

1.3 Shaders

To gain back flexibility parts of the fixed-function graphics pipeline were replaced by shaders (see figure 4.1), executed on special-purpose, but programmable processors called shader units.

In the beginning shader units were specialized for shaders of a certain type such as vertex or fragment shaders. This is no longer the case allowing dynamic reassignment of shader units between shaders (unified shader architecture). Different shader architectures are in use: Nvidia's shader units use SIMD scalar processors, ATI uses MIMD vector processors with VLIW.

1.4 Abstract

An architecture for a shader unit is presented (part I). The architecture has been optimized for efficient execution of shaders written in high-level shading languages. Implementation of the OpenGL shaders (part II) including possible implementations of built-in functions of shading laguages (chapter 6) is discussed. Some interesting aspects of possible hardware-implementations of the architecture are presented (chapter 7). A cycle-accurate simulator of such a hardware-implementation has been created (chapter 8). Some adaptions of the architecture for embedded systems are discussed briefly (part IV).

1.5 Related work

The Attila project at the Technical University of Catalonia has created a simulator of a graphics processor including simple shader units [29] to analyze effects a unified shader architecture has on performance [28]. The Open Graphics Project [3] tries to create a synthesizeable Verilog description of a graphics processor for OpenGL 1.5 (fixed-function pipeline only). There is a register transfer level SystemC model of most of the fixed-function OpenGL 1.5 fragment pipeline with some texture combining extensions [13].

Available information on the architecture of commercial shader units is not very detailed.

Part I

Architecture

Chapter 2

Execution environment

2.1 Data Types

Operations in shaders typically operate on coordiantes in four-dimensional space or homogenous coordinates representing points in three-dimensional projective space. To implement these efficiently the shader unit proposed here uses four component vectors as data types.

All data types are 128 bit wide 4 component vectors. Other data types offered by higher-level languages can be implemented by using only some components of the hardware's data types (scalars, 2- and 3-component vectors) or using multiple registers (matrices).

2.1.1 vec4

vec4 is the most important data type, a 4 component vector of 32 bit floating-point values. Many programs will use this data type only.

2.1.2 ivec4

ivec4 is a 4 component vector of 32 bit signed integers in two's complement representation. Instructions that treat their operands as ivec4 have a z prefix.

2.1.3 bvec4

bvec4 is a 4 component vector of boolean values. All bits except 0, 32, 64 and 96 are zero.

2.1.4 conversion

There are instructions for conversion between the different data types: qtoz for vec4 to ivec4 conversion, ztoq for ivec4 to vec4 conversion. The ztob for ivec4 to bvec4 conversion and qtob for vec4 to bvec4 conversion instructions map

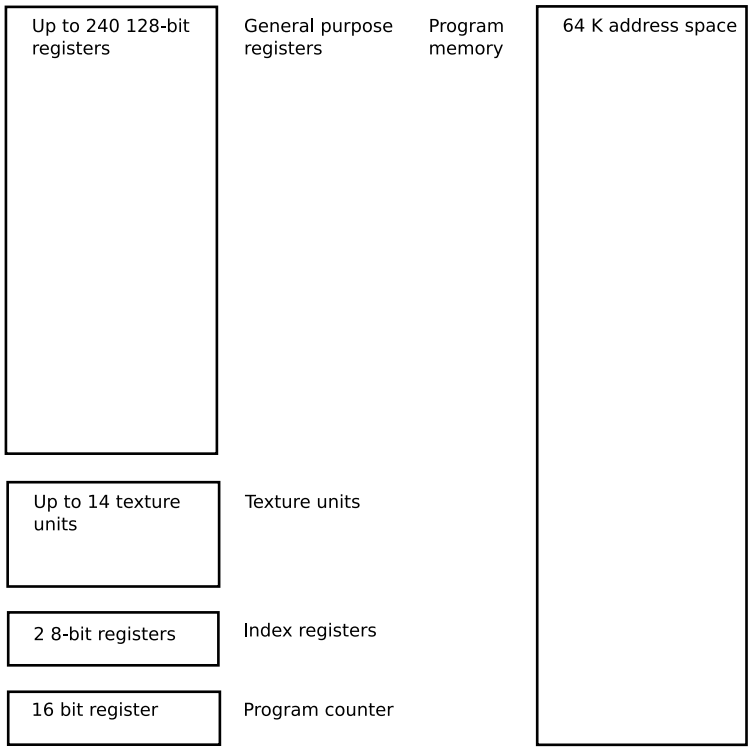

Figure 2.1: Execution Environent

nonzero input values to true. A bvec4 can be used as ivec4 directly, so no bvec4 to ivec4 conversion instruction is needed.

2.2 Registers

General purpose registers, texture units and index registers for indirect addressing are mapped into a single address space: Instructions use 8-bit values to specify operands. The lower 240 addresses are mapped to general-purpose registers, 240 to 253 are mapped to texture units, 254 and 255 are used for indirect addressing.

2.2.1 general purpose registers

The 128-bit general purpose registers are treated as 4-component vector registers by most instructions. All of these registers are available for general storage of operands. Data is passed between the processor and the rest of the graphics pipeline through these registers.

2.2.2 index registers

The two 8 bit index registers can be written by the inx, ini and deci instructions only. Instructions can use indirect addressing through index registers by using 254 or 255 as operand. An operand of 254 will be replaced by the register the first index register is referring to, an operand of 255 will be replaced by the register the second index register is referring to.

2.2.3 program counter

The 16 bit program counter is incremented after reading an instruction. Writing the program counter is possible using jump instructions (end, jump, kill).

2.3 Texture units

The texture units are not part of the processor, but their interface needs to be suitable for the processor's needs.

2.3.1 interface

Texture units are mapped into register space. There is one 128 bit register per texture unit. The processor can supply texture coordinates to the texture unit by using the unit's register as destination operand. It retrieves texture data by reading the register.

The first three components of the data written are the texture coordinates (for projective texture coordinates the dehomogenization will have to be implemented in the shader). The fourth component can be used as LOD parameter

or LOD bias. The type of texture (one-, two-, three-dimensional, cubemap, if a bias or LOD parameter is expected) is not under shader control. It will have to be specified by the driver.

2.3.2 level of detail and derivatives

$$\lambda = \log_2(\rho(x,y) + b) \quad (2.1)$$

$$\rho(x,y) = \max\{\sqrt{(\frac{\partial u}{\partial x})^2 + (\frac{\partial v}{\partial x})^2 + (\frac{\partial w}{\partial y})^2}, \sqrt{(\frac{\partial u}{\partial y})^2 + (\frac{\partial v}{\partial y})^2 + (\frac{\partial w}{\partial x})^2}\} \quad (2.2)$$

Choice of mipmaps and texture minification / magnification depends on a level of detail parameter λ (equation 2.1 with b being a user-defined value, x and y as screen coordiantes, u, v, w as texture coordinates; see equation 10.1 for an approximation allowed by the OpenGL specification). It can be specified explicitly by a shader or computed from the texture coordinates' derivatives. There are two possible implementations of the λ calculation that don't affect processor architecture, but they affect texture units and shader programs. A third option would require changes in processor architecture.

The rasterizer could provide derivatives of input variables in general purpose registers. This solution offers the highest quality derivatives, but requires additional computations in the shader program to calculate derivatives of texture coordinates from input variables and their derivatives.

A second, cheaper alternative would be to ensure that multiple processors operate in lockstep. The texture units could then access the texture coordinates used by neighbouring texture units resulting in a piecewise linear approximation of derivatives and use it to calculate λ. Derivatives would be undefined inside if/else statements and loops. Higher-order derivatives would be undefined.

The third alternative would require lockstep operation of multiple processors, too, but instead of doing the λ calculation in the texture units processors would directly access their neighbour's registers to calculate derivatives and λ, supplying λ to the texture units like in the first case.

Unless the third alternative is chosen fast calculation of λ in the shader is necessary and as in part IV two additional instructions should be considered.

All of these alternatives conform to the OpenGL and GLSL specifications. The specifications allow for undefined higher-order derivatives and undefined derivatives inside conditionals.

2.4 Dependencies

Hardware implementation will typically use pipelining to improve processor speed. Many processors use additional hardware to avoid data and control hazards. Since shaders will be compiled in the driver for this processor binary compability is not an issue. The compiler shall avoid pipelining hazards.

2.4.1 data dependencies

An instruction reading a general-purpose register following an instruction writing the same general purpose register directly or with only one or two instructions in between will read undefined values in the parts of the register that have been written (reading another part of the same register is fine).

2.4.2 control dependencies

Up to three instructions following a jump instruction might be executed if the jump is taken (they will always be executed otherwise).

Chapter 3

Instructions

There are some instructions that might seem a bit unusual for a floating-point processor with four component vector data types. Their existence is justified by the purpose of this processor, executing shaders in graphics hardware:

Dot products are common in shaders, thus special dot product instructions are provided. They are useful for matrix multiplication and approximating functions using power series, too. The rules on data dependencies (subsection 2.4.1) are tailored to pipelined implementation of dot products.

There is only one instruction that is expensive hardware-wise: rsq, the reciprocal square root. It is used to implement some functions which are common in shaders, but expensive: Division, square root and reciprocal square root.

Exponentiation is difficult to implement in software and would be expensive to implement in hardware, too, but since it's not as important as rsq another approach has been chosen instead: The exp2 instruction, which isn't expensive hardware-wise. It provides a rough approximation of base 2 exponentiation which can be made more accurate using a taylor series which is easy to implement in software.

Since many functions that are implemented use power series additional support (beyond dot product instructions) is provided by the poly instruction.

Instructions (figure 3.1) are 32 bit values.

Instructions typically have a destination write mask, which is used to selectively write only some components of the destination operand. Scalar instructions have a source selection which is used to select the component of the source operand that is used as input value.

In the following example only the first and the third component of register 1 are doubled, while the second and fourth component retain their values:

```
add 1.xz, 1, 1
```

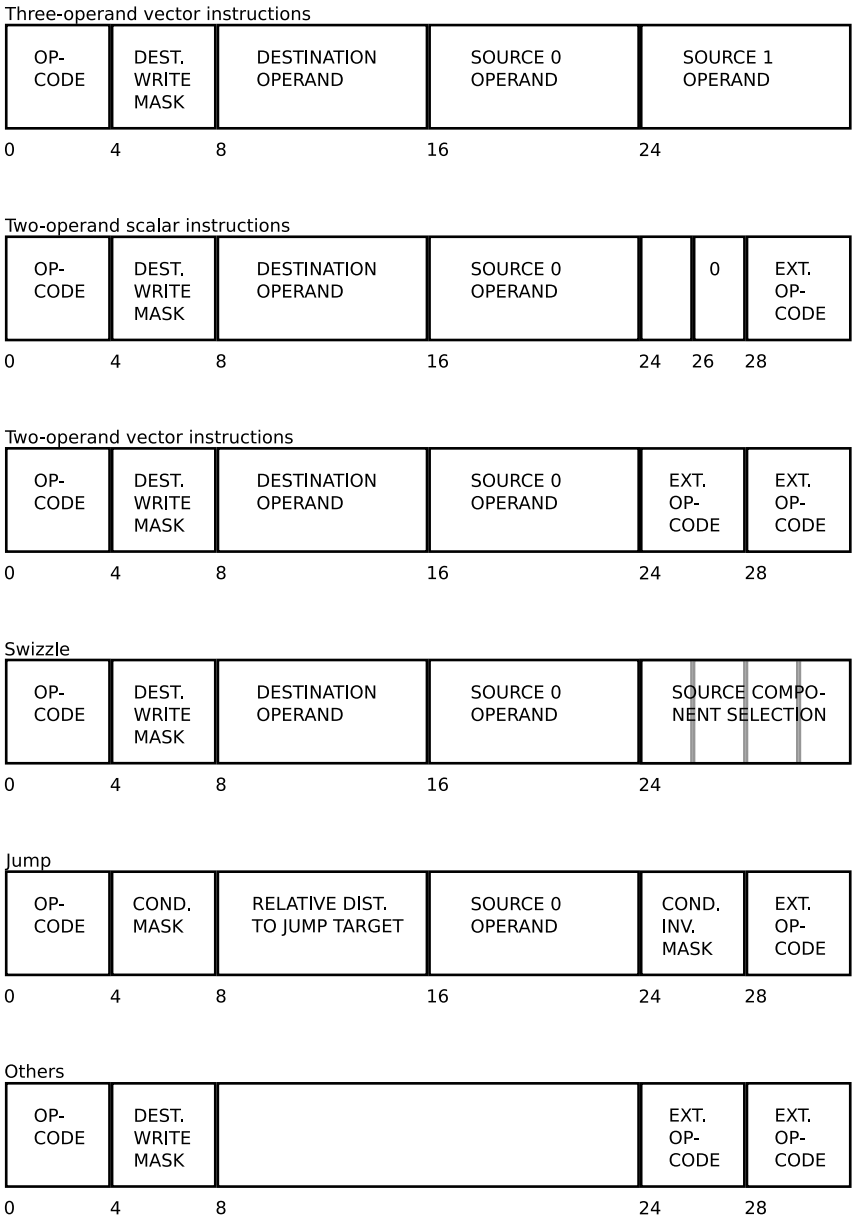

Figure 3.1: Instructions

3.1 Floating point three-operand vector instructions

These instructions have two 128 bit source operands which are treated as quadruples of 32 bit floating-point values. The result is a quadruple of 32 bit floating-point values.

3.1.1 add

Addition.

3.1.2 max

Component-wise maximum.

3.1.3 min

Component-wise minimum.

3.1.4 mul

Component-wise multiplication.

3.1.5 sub

Subtraction.

3.2 Floating point two-operand vector instructions

These instructions have a single 128 bit source operand which is treated as a quadruple of 32 bit floating-point values. The result is a quadruple of 32 bit floating-point values.

3.2.1 eexp

Component-wise extraction of exponent of floating-point representation:

$$x \mapsto \lfloor \log_2 x \rfloor$$

3.2.2 eman

Component-wise extraction of mantissa of floating-point representation:

$$a2^b \mapsto a$$

3.3 Integer three-operand vector instructions

These instructions have two 128 bit sourceoperands which are treated as quadruples of 32 bit integers in two's complement representation. The result is a quadruple of 32 bit integers in two's complement representation.

3.3.1 addz

Addition

3.3.2 mulz

Component-wise multiplication.

3.3.3 subz

Subtraction.

3.4 Integer two-operand vector instructions

This instruction has a single 128 bit source operand which is treated as a quadruple of 32 bit integers in two's complement representation. The result is a quadruple of 32 bit integers in two's complement representation.

3.4.1 absz

Component-wise absolute value.

3.5 Logic instructions (all three-operand)

All logic instructions have two 128 bit source operands which are treated as 128 bit registers. The result is a 128 bit value.

3.5.1 and

Bitwise and.

3.5.2 or

Bitwise or.

3.5.3 xor

Bitwise exclusive or.

3.6 Comparison

These instruction have two 128 bitsource operands. They are treated as a quadruple of 32 bit floating-point values. The result is a quadruple of boolean values.

3.6.1 less

The instruction does a component-wise comparison.

3.7 Dot products

These instructions have two 128 bit source operands. The result is a 32 bit floating-point value, which is placed in all components of the resulting quadruple.

3.7.1 dp3

The source operands are treated as triples of 32 bit floating-point values (the upper 32 bit are ignored). The instruction calculates the dot product.

3.7.2 dp4

The source operands are treated as a quadruple of 32 bit floating-point values. The instruction calculates the dot product.

3.8 Floating point scalar instructions

These instructions take a single 32 bit source operand. It can be selected from any component of an input vector. The result is a single 32 bit floating-point value, which is placed in all components of the resulting quadruple.

3.8.1 exp2

Approximation of base 2 exponentiation. The result is exact for integer input values.

$$x \mapsto y, 2^{\lfloor x \rfloor} \leq y \leq 2^{\lceil x \rceil}$$

3.8.2 flr

Returns the largest integer which is less than or equal to x:

$$x \mapsto \lfloor x \rfloor$$

3.8.3 frac

Returns the fractional part of the input value, that is the input value minus the largest integer which is less than or equal to x:

$$x \mapsto x - \lfloor x \rfloor$$

3.8.4 rsq

The instruction returns the reciprocal square root of the input value:

$$x \mapsto x^{-\frac{1}{2}}$$

3.9 Jumps

All jump instructions are executed with some delay: Two more instructions will be loaded and executed before jumping.

3.9.1 end

Jump to program start. This instruction signals that the fragment has been processed and can flow further into the graphics pipeline.

3.9.2 ijmp

Unconditional indirect jump. The program counter is set to the lower 16 bits of the 32 bit integer source operand.

3.9.3 jump

Conditional relative jump. It takes a single 4 bit source operand (bits 0, 32, 64, 96 of input vector) and two 4 bit masks: For each bit set in the first mask the corresponding input bit is inverted. All input bits for which the corresponding bit in the second mask is set are anded together with constant true. If the result is true the target location is loaded into the program counter.

3.9.4 kill

Jump to program start. This instruction signals that the fragment has been processed and discarded.

3.10 Special

3.10.1 deci

Decrement an index register.

3.10.2 inci

Increment an index register.

3.10.3 inx

Load the lower 8 bits of a 32 bit integer into an index register.

3.10.4 nop

No-op. Do nothing for one clock cycle.

3.10.5 poly

The source operand is a 32 bit floating point value. The result is a quadruple of 32 bit floating point values.

$$x \mapsto (x, x^2, x^3, x^4)^T$$

3.10.6 qtob

The instruction has a 128 bit source operand which is treated as a quadruple of 32 floating-point values. It converts them component-wise to integer. The result is a quadruple of boolean values: True for nonzero floating-point numbers, false otherwise.

3.10.7 qtoz

The instruction has a 128 bit source operand which is treated as a quadruple of 32 floating-point values. It converts them component-wise to integer. The result is a quadruple of 32 bit integers.

3.10.8 swiz

The instruction has a 128 bit source operand which is treated as a quadruple of 32 bit values. It is used to swizzle the input's components, that is for each component of the output vector a user-defined component of the input vector can be chosen.

In the following example the first and second component of register 1 are swapped with each other, the third component is placed in both the third and fourth component:

```
swiz 1, 1.yxzz
```

3.10.9 ztob

The instruction has a 128 bit source operand which is treated as a quadruple of 32 bit integers. It converts them component-wise to bool: Entries that are zero remain so, all other entries are converted to one.

3.10.10 ztoq

The instruction has a 128 bit source operand which is treated as a quadruple of 32 bit integers. It converts them component-wise to floating-point. The result is a quadruple of 32 bit floating-point values.

Part II

Implementing OpenGL

Chapter 4

OpenGL pipeline

Today there exist two important competing APIs for hardware-accelerated 3D computer graphics: Microsoft's Direct3D, and OpenGL. The OpenGL standard is governed by the OpenGL architecture review board (ARB), once an independent consortium, now a working group within the Khronos group [1]. New hardware functionality is often exposed through so-called extensions before being exposed within the core API. There are vendor-specific extensions (their name has a vendor prefix like NV for Nvidia), multi-vendor extensions (prefixed by EXT) and extensions approved by the ARB (prefixed by ARB). Applications can check for the presence of extensions at runtime.

The Khronos Group has created OpenGL ES, a variant of OpenGL optimized for embedded systems.

OpenGL and OpenGL ES 1.x have a fixed rendering pipeline which can be customized to some extend by parameters, while in OpenGL 2.x parts of the fixed functionality can be replaced by shaders [19, section 2.15 (page 71) and section 3.11 (page 196)] as can be seen in figure 4.1. In OpenGL ES 2.x the fixed functionality has been removed so that the use of shaders is mandatory [4, section 2.15 (page 11) and section 3.11 (page 27)]; OpenGL ES removes the alpha test, too; its functionality can be implemented in a fragment shader.

There have been multiple proposals for further replacement of fixed functionality by shaders. The most important proposals are texture shaders and geometry shaders.

Texture shaders have been implemented by Nvidia. They are meant to make texture filtering programmable. Texture shaders are available as vendor-specific extensions [15], [14], [16], but have never been popular with other vendors. Since they do not add functionality beyond what could be done in a fragment shader anyway, they only allow to optimize shader organization for Nvidia hardware.

Geometry shaders (also called topology shaders or primitive shaders) are added as an additional stage beyond vertex processing. They can access connectivity information and are executed per primitive (providing read access to neighbour primitives). They are available to the programmer through extensions [22], [23], [24] and part of Direct3D 10 [21].

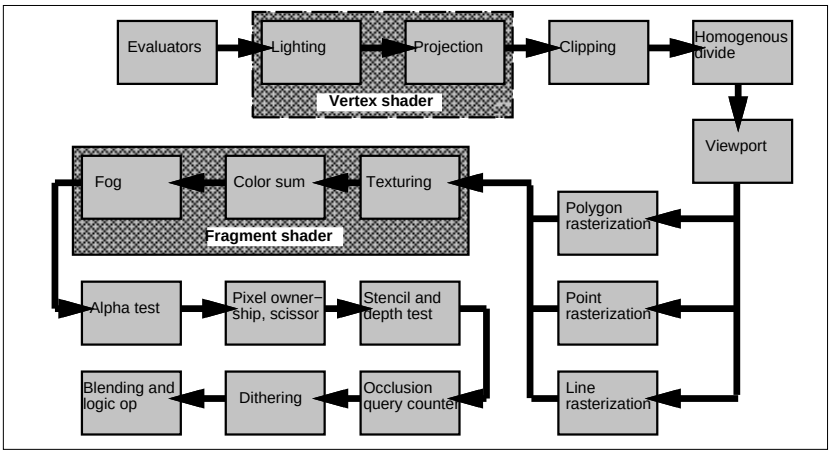

Figure 4.1: OpenGL graphics pipeline

4.1 OpenGL 1.x

4.1.1 vertex processing

Let $\mathfrak{M}$ the modelview matrix, $\mathfrak{P}$ the projection matrix, $\mathfrak{a}$ the position of an input vertex. A shader will have to implement the following transformation to mimic the fixed pipeline's vertex position transformation [18, page 38]:

$$\mathfrak{a} \mapsto \mathfrak{P}\mathfrak{M}\mathfrak{a} \tag{4.1}$$

Using a premultiplied $\mathfrak{P}\mathfrak{M}$ this can be implemented using four dp4 instructions. Other parts of the shader depend on the number and types of lights enabled. A suitable shader has to be generated in the driver.

4.1.2 fragment processing

A shader implementing the fixed fragment pipeline depends on the texturing, color sum and fog settings. As above a suitable shader has to be generated in the driver.

4.2 OpenGL ES 1.x

OpenGL ES is a variant of OpenGL for embedded systems. It is based on OpenGL with some functionality removed (mostly redundant functionality and support for high-precision floating point variables) and some added.

4.3 OpenGL 2.x

In OpenGL 2.x the fixed vertex and fragment processing can be replaced by GLSL shaders [19, page 71ff]. See the chapter on shading languages for information on implementing shaders.

4.4 OpenGL ES 2.x

OpenGL ES 2.x adds support for shaders. It removes the fixed vertex and fragment processing that is part of OpenGL ES 1.x. Unlike OpenGL 2.x shaders OpenGL ES 2.x shaders replace the alpha test, too.

4.5 OpenGL 3.x

While OpenGL up to 2.x was a state-based API OpenGL 3.x is object based. Like in OpenGL ES 2.x there is no support for a legacy fixed pipeline for parts of the pipeline where shaders are available.

Chapter 5

Shading languages

5.1 ARB style vertex and fragment programs

5.1.1 description

These were the first ARB-approved shading extensions. They expose programmability of the graphics pipeline in a vendor-independent way using a low-level pseudo-assembly language [10] [5]. Since its functionality is a subset of GLSL's functionality it is discussed less verbose here.

5.1.2 example - Doom 3

OpenGL might be dead by now were it not for id software and it's lead programmer, John Carmack, which used it in their popular Quake ego-shooters thus forcing graphics hardware vendors to provide OpenGL support through the dark age of Direct3D's dominance. John Carmack initiated initiatives to improve OpenGL support and correctness [8] [6] [27] in drivers and operating systems.

id's latest rendering engine was first used in Doom 3; it is used in third-party titles such as Quake 4. It provides a unified lighting / shadow model providing realtime dynamic shadowing using stencil shadows. As far as shaders are concerned the major part lives in glprogs/interaction.vfp as ARB style vertex / fragment program in the main archive of Doom 3 and the Doom 3 demo [9]:

```
!!ARBvp1.0 OPTION ARB_position_invariant ;

# VPROG_INTERACTION
#
# input:
#
# attrib[8] TEX0 texture coordinates
# attrib[9] TEX1 normal
```

Figure 5.1: Doom 3

```
# attrib[10] TEX2 tangent[0]
# attrib[11] TEX3 tangent[1]
# COL vertex color
#
# c[4] localLightOrigin
# c[5] localViewOrigin
# c[6] lightProjection S
# c[7] lightProjection T
# c[8] lightProjection Q
# c[9] lightFalloff S
# c[10] bumpMatrix S
# c[11] bumpMatrix T
# c[12] diffuseMatrix S
# c[13] diffuseMatrix T
# c[14] specularMatrix S
# c[15] specularMatrix T
# c[16] vertex color modulate
# c[17] vertex color add
#
# output:
#
# texture 0 is the cube map
```

```
# texture 1 is the per-surface bump map
# texture 2 is the light falloff texture
# texture 3 is the light projection texture
# texture 4 is the per-surface diffuse map
# texture 5 is the per-surface specular map
# texture 6 is the specular lookup table

TEMP R0, R1, R2;

PARAM defaultTexCoord = { 0, 0.5, 0, 1 };


# calculate vector to light in R0
SUB R0, program.env[4], vertex.position;

# put into texture space for TEX0
DP3 result.texcoord[0].x, vertex.attrib[9], R0;
DP3 result.texcoord[0].y, vertex.attrib[10], R0;
DP3 result.texcoord[0].z, vertex.attrib[11], R0;

# textures 1 takes the base coordinates by the texture matrix
MOV result.texcoord[1], defaultTexCoord;
DP4 result.texcoord[1].x, vertex.attrib[8], program.env[10];
DP4 result.texcoord[1].y, vertex.attrib[8], program.env[11];

# texture 2 has one texgen
MOV result.texcoord[2], defaultTexCoord;
DP4 result.texcoord[2].x, vertex.position, program.env[9];

# texture 3 has three texgens
DP4 result.texcoord[3].x, vertex.position, program.env[6];
DP4 result.texcoord[3].y, vertex.position, program.env[7];
DP4 result.texcoord[3].w, vertex.position, program.env[8];

# textures 4 takes the base coordinates by the texture matrix
MOV result.texcoord[4], defaultTexCoord;
DP4 result.texcoord[4].x, vertex.attrib[8], program.env[12];
DP4 result.texcoord[4].y, vertex.attrib[8], program.env[13];

# textures 5 takes the base coordinates by the texture matrix
MOV result.texcoord[5], defaultTexCoord;
DP4 result.texcoord[5].x, vertex.attrib[8], program.env[14];
DP4 result.texcoord[5].y, vertex.attrib[8], program.env[15];

# texture 6's texcoords will be the halfangle in texture space
```

```
# calculate normalized vector to light in R0
SUB R0, program.env[4], vertex.position;
DP3 R1, R0, R0;
RSQ R1, R1.x;
MUL R0, R0, R1.x;

# calculate normalized vector to viewer in R1
SUB R1, program.env[5], vertex.position;
DP3 R2, R1, R1;
RSQ R2, R2.x;
MUL R1, R1, R2.x;

# add together to become the half angle vector in object space (non-normalized)
ADD R0, R0, R1;

# put into texture space
DP3 result.texcoord[6].x, vertex.attrib[9], R0;
DP3 result.texcoord[6].y, vertex.attrib[10], R0;
DP3 result.texcoord[6].z, vertex.attrib[11], R0;

# generate the vertex color, which can be 1.0, color, or 1.0 - color
# for 1.0 : env[16] = 0, env[17] = 1
# for color : env[16] = 1, env[17] = 0
# for 1.0-color : env[16] = -1, env[17] = 1
MAD result.color, vertex.color, program.env[16], program.env[17];

END


#=============================================================================

!!ARBfp1.0
OPTION ARB_precision_hint_fastest;

# texture 0 is the cube map
# texture 1 is the per-surface bump map
# texture 2 is the light falloff texture
# texture 3 is the light projection texture
# texture 4 is the per-surface diffuse map
# texture 5 is the per-surface specular map
# texture 6 is the specular lookup table

# env[0] is the diffuse modifier
# env[1] is the specular modifier

TEMP light, color, R1, R2, localNormal, specular;
```

```
PARAM subOne = { -1, -1, -1, -1 };
PARAM scaleTwo = { 2, 2, 2, 2 };

# load the specular half angle first, because
# the ATI shader gives a "too many indirections" error
# if this is done right before the texture indirection

#-----------------
#TEX specular, fragment.texcoord[6], texture[0], CUBE;
#MAD specular, specular, scaleTwo, subOne;


# instead of using the normalization cube map, normalize with math
DP3 specular, fragment.texcoord[6],fragment.texcoord[6];
RSQ specular, specular.x;
MUL specular, specular.x, fragment.texcoord[6];
#-----------------


#
# the amount of light contacting the fragment is the
# product of the two light projections and the surface
# bump mapping
#

# perform the diffuse bump mapping

#-----------------
TEX light, fragment.texcoord[0], texture[0], CUBE;
MAD light, light, scaleTwo, subOne;

# instead of using the normalization cube map, normalize with math
#DP3 light, fragment.texcoord[0],fragment.texcoord[0];
#RSQ light, light.x;
#MUL light, light.x, fragment.texcoord[0];
#-----------------

TEX localNormal, fragment.texcoord[1], texture[1], 2D;
MOV localNormal.x, localNormal.a;
MAD localNormal, localNormal, scaleTwo, subOne;
DP3 light, light, localNormal;

# modulate by the light projection
TXP R1, fragment.texcoord[3], texture[3], 2D;
MUL light, light, R1;
```

```
# modulate by the light falloff
TXP R1, fragment.texcoord[2], texture[2], 2D;
MUL light, light, R1;

#
# the light will be modulated by the diffuse and
# specular surface characteristics
#

# modulate by the diffuse map and constant diffuse factor
TEX R1, fragment.texcoord[4], texture[4], 2D;
MUL color, R1, program.env[0];

# perform the specular bump mapping
DP3 specular, specular, localNormal;

# perform a dependent table read for the specular falloff
TEX R1, specular, texture[6], 2D;

# modulate by the constant specular factor
MUL R1, R1, program.env[1];

# modulate by the specular map * 2
TEX R2, fragment.texcoord[5], texture[5], 2D;
ADD R2, R2, R2;
MAD color, R1, R2, color;


MUL color, light, color;

# modify by the vertex color

MUL result.color, color, fragment.color;

# this should be better on future hardware, but current drivers make it slower
#MUL result.color.xyz, color, fragment.color;


END
```

Implementing the vertex shader is rather straightforward, with a little reordering and the introduction of a new temporary register (R1') the no-ops have been eliminated:

```
sub R1', c5, vertexposition
```

```
sub R0, c4, vertexposition
dp4 rt2.x, vertexposition, c9
dp4 rt3.x, vertexposition, c6
dp3 R2, R1', R1'
dp4 rt3.y, vertexposition, c7
dp3 R1, R0, R0
dp4 rt3.w, vertexposition, c8
rsq R2, R2.x
dp3 rt0.x, a9, R0
dp3 rt0.y, a10, R0
rsq R1, R1.x
mul R1', R1', R2
dp3 rt0.z, a11, R0
dp4 rt1.x,a8, c10
mul R0, R0, R1
dp4 rt1.y,a8, c11
dp4 rt4.x, a8, c12
dp4 rt4.y, a8, c13
add R0, R0, R1'
dp4 rt5.x, a8, c14
dp4 rt5.y, a8, c15
mul R1, vertexcolor, c16
dp3 rt6.x, a9, R0
dp3 rt6.y, a10, R0
dp3 rt6.z, a11, R0
add resultcolor, R1, c17
end
nop
nop
```

This vertex shader assumes that attribute i is stored in ai, constant i is stored in ci and that the contant defaultTexCoord has been stored into the rti (has to be done only once, since the values modified by the program are overwritten on each execution anyway).

The fragment shader produces only one result, the fragment color, which depends on everything else; furthermore this fragment shader has a rather long critical path, thus some no-ops, mostly at the end of the fragment shader, survive:

```
register 240, tex0
register 241, tex1
register 242, tex2
register 243, tex3
register 244, tex4
register 245, tex5
register 246, tex6
```

```
register 240, ft0
register 241, ft1
register 244, ft4
register 245, ft5

swiz localNormal, tex1.xyzx
dp3 specular, ft6, ft6
rsq tmp1, ft3.w
rsq tmp2, ft2.w
mul localNormal, localNormal, scaleTwo
rsq specular, specular.x
mul ft3, ft3, tmp1
mul ft2, ft2, tmp2
add localNormal, localNormal, subOne
mul specular, specular, ft6
mul tex2, ft2, tmp2
mul tex3, ft3, tmp1
dp3 light, tex0, localNormal
dp3 tex6, specular, localNormal
nop
add R2, tex2, tex2
mul light, light, tex3
mul R1, tex6, e1
nop
mul color, tex4, e0
mul light, light, tex2
mul R1, R1, R2
nop
nop
mul resultcolor, light, fragmentcolor
add color, R1, color
nop
nop
nop
mul resultcolor, color, resultcolor
end
nop
nop
```

This fragment shader assumes that texture coordinate i is stored in fti, env[i] in ei.

Figure 5.2: Quad rendered using steep parallax mapping

5.2 GLSL

5.2.1 description

The OpenGL Shading Language (GLSL) is a high-level language with a C-like syntax for both vertex and fragment shaders. See [11] for the specification. It has datatypes for floating-point, integer and boolean vectors and matrices. With the exception of recursion advanced flow control is supported as in C. It has lots of built-in functions including functions providing texture access. Unlike Direct3D's HLSL, which is compiled into an partly hardware independent low-level language GLSL programs are compiled by the graphics driver.

5.2.2 example - steep parallax mapping

The most advanced singlepass bumpmapping algorithm today is steep parallax mapping [20]. The algorithm's inner loop responsible for self-shadowing can be seen in the following GLSL code (before the loop NB.a ¡ height and height ¡ 1 are true):

```
while ((NB.a < height) && (height < 1))
{
height += step;
offsetCoord += delta;
```

```
NB = texture2D(normalBumpMap, offsetCoord);
}

// We are in shadow if we left the loop because
// we hit a point
selfShadow = (NB.a < height);
```

It is difficult to implement due to the loop's shortness and the data dependencies. Nevertheless hand-optimizing yields an acceptable implementation:

```
add height, height, step
add tex, offsetCoord, delta
%1:
add offsetCoord, offsetCoord, delta
less flags.x, height.x, one
less flags.a, tex.a, height
nop
add height, height, step
add tex, offsetCoord, delta
jump %1, flags.xa
nop
nop
```

The code assumes that height (step) from the GLSL shader is stored in both height.x and height.a (step.x and step.a). one contains 1.0 in ts first component. selfShadow lives in flags.a. Knowledge that the loop condition is always true on the first iteration and that height and the texture are not used after the loop has been used in optimizing the code.

5.2.3 example - Ogre

Ogre [2] is a free scene-orientated 3D engine, which is used in a large number of both free and non-free games including Ankh - Heart of Osiris and Pacific Fighters. In some of Ogre's vertex shaders large arrays of input data are accessed using indices calcualted from per-vertex attributes. Implementing these would be very difficult without index registers. A sample shader included in the Ogre 1.4.5 package is shown below:

```
attribute vec4 blendIndices;
attribute vec4 blendWeights;

uniform vec4 worldMatrix3x4Array[72];
uniform mat4 viewProjectionMatrix;
uniform vec4 ambient;

void main()
{
```

Figure 5.3: Ankh - Heart of Osiris

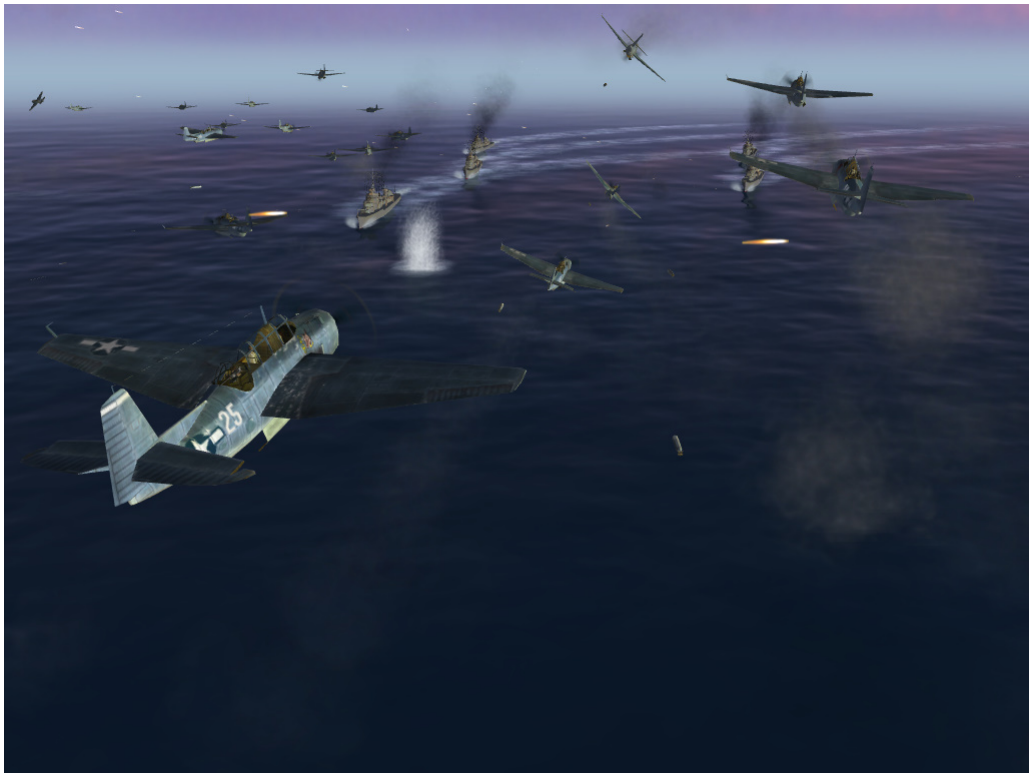

Figure 5.4: Pacific Storm

```
vec3 blendPos = vec3(0,0,0);

for (int bone = 0; bone < 2; ++bone)
{
int idx = int(blendIndices[bone]) * 3;
mat4 worldMatrix;
worldMatrix[0] = worldMatrix3x4Array[idx];
worldMatrix[1] = worldMatrix3x4Array[idx + 1];
worldMatrix[2] = worldMatrix3x4Array[idx + 2];
worldMatrix[3] = vec4(0);
blendPos += (gl_Vertex * worldMatrix).xyz * blendWeights[bone];
}
gl_Position = viewProjectionMatrix * vec4(blendPos, 1);

gl_FrontSecondaryColor = vec4(0,0,0,0);
gl_FrontColor = ambient;
gl_TexCoord[0] = gl_MultiTexCoord0;
}
```

The loop has been unrolled. It is assumed that register 1 has been initialized to 0, register 8.w to 1, register 7.xy to (3,3) and 10.xy to (16,16), worldMatrix3x4Array shall be stored in the registers starting at register 16. The rows of viewProjectionMatrix shall be stored in registers 12 to 15.

```
register 0, ambient
register 0, gl_FrontColor
register 1, gl_FrontSecondaryColor
register 2, gl_Position
register 3, gl_TexCoord[0]
register 3, gl_MultiTexCoord0
register 4, blendIndices
register 5, blendWeights

register 6, blendPos
; Use first three components of register 4 for blendPos.
register 7, three

register 8, tmp
register 9, tmp2
register 10, twelve

main:
qtoz tmp.xy, blendIndices
swiz blendPos.xyz, 1.xyzw
nop
nop
```

```
mulz tmp.xy, tmp, three
nop
nop
nop
addz tmp.xy, tmp, twelve
nop
nop
nop
inx 0, tmp.x
inx 1, tmp.y
nop
nop
dp4 tmp.x, gl_Position, 254
inci 0
dp4 tmp2.x, gl_Position, 255
inci 1
nop
dp4 tmp.y, gl_Position, 254
inci 0
dp4 tmp2.y, gl_Position, 255
inci 1
nop
dp4 tmp.z, gl_Position, 254
nop
dp4 tmp2.z, gl_Position, 255
nop
nop
nop
add tmp.xyz, tmp, tmp2
; tmp now contains the value of blendPos at the end of the loop.
nop
nop
nop
dp4 gl_Position.x, 12, tmp
dp4 gl_Position.y, 13, tmp
dp4 gl_Position.z, 14, tmp
dp4 gl_Position.w, 15, tmp
end
nop
nop
nop
```

5.3 GLSL ES

5.3.1 description

The OpenGL ES Shading Language (GLSL ES) [25] is similar to GLSL. Advanced GLSL features like derivatives and perlin noise are not present in GLSL ES. The main restriction is flow control though: Only unrollable loops are allowed to ensure that hardware without branching instructions can implement GLSL ES. There are some additional features like precision qualifiers.

Any device that can execute GLSL shaders should have no problems with GLSL ES.

Unlike other shaders GLSL ES shaders replace the alpha test, too.

Chapter 6

Functions

Shading languages provide various built-in functions. This chapter presents implementations of the GLSL built-in functions that have scalar, three- or four-dimensional floating-point operands. GLSL was chosen since it is the most advanced language and contains the most built-in functions. Scalar, three- and fourdimensional floating-point functions are the most common in shader programs. In many cases the generalization to integer operands is straightforward, like using the absz instruction instead of abs for calculating abs(). Many functions will work on twodimensional operands unmodified. In the case of abs() the abs instruction will implement the function for one to fourdimensional floating-point operands, while absz will implement it for one to fourdimensional integer operands. This chapter contains implementations of many of these functions.

6.1 Trigonometric

The trigonometric functions are neither very common nor difficult to implement in software. A hardware implementation seems not necessary. They have rapidly converging Taylor series; combined with a preceding range reduction short programs ($\approx$ 9 instructions) are sufficiant to get a result of reasonable precision. The dp4 instruction is very useful in implementing the series.

6.1.1 sin()

$$\sin(x) = \sum_{i=0}^{\infty} \frac{(-1)^i x^{2i+1}}{(2i+1)!} \tag{6.1}$$

The series definition (equation 6.1) of sine, can be used for approximation. It is identical to the taylor series for sine, sine being an uneven function all the even coefficients of the taylor series are 0. Approximation using a taylor polynomial is most exact near 0; a fast and not too unprecise implementation can be obtained using range reduction and scaling (equation 6.6):

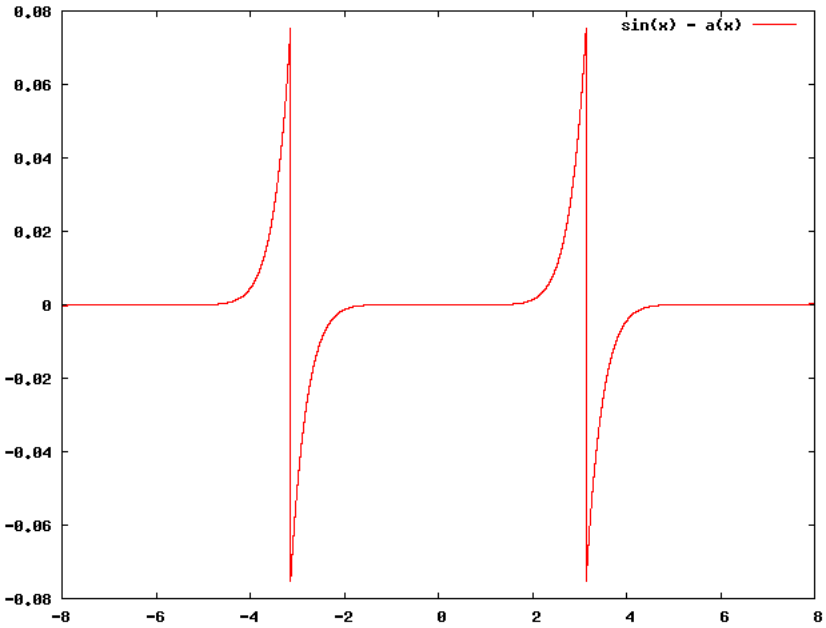

Figure 6.1: Sine approximation error

$$\sin(x) = t(g(\frac{x}{2\pi} + \frac{1}{2}) - \frac{1}{2}) \tag{6.2}$$

$$g(x) = x - \lfloor x \rfloor \tag{6.3}$$

$$t(x) = \sin(2\pi x) \approx \sum_{i=0}^{3} t_i x^{2i+1} = s(x) \tag{6.4}$$

$$t_i = \frac{t^{(2i+1)}(0)}{(2i+1)!} = \frac{sin^{(2i+1)}(0)(2\pi)^{2i+1}}{(2i+1)!} = \frac{(-1)^i(2\pi)^{2i+1}}{(2i+1)!} \tag{6.5}$$

$$a(x) = s(g(\frac{x}{2\pi} + \frac{1}{2}) - \frac{1}{2}) \tag{6.6}$$

The argument is transformed to range $[-\frac{1}{2}, \frac{1}{2}[$ using g. The scaled sine t can be easily approximated by its taylor series, as sine itself it is an uneven function, thus every other coefficient of the taylor polynomial is zero.

Assuming that the coefficients (equation 6.5) of the taylor polynomial (equation 6.4) have been placed in register $\mathfrak{p}$ the following program places an approximation of x in $\mathfrak{r}$:

$$\mathfrak{p} = \begin{pmatrix} t_0 \\ t_1 \\ t_2 \\ t_3 \end{pmatrix}, \mathfrak{c} = \begin{pmatrix} \frac{1}{2} \\ * \\ * \\ * \end{pmatrix}, \mathfrak{d} = \begin{pmatrix} \frac{1}{2\pi} \\ * \\ * \\ * \end{pmatrix}, \mathfrak{b} = \begin{pmatrix} x \\ * \\ * \\ * \end{pmatrix}$$

```
mul t.x, b, d
nop
nop
nop
add t.x, t, c
nop
nop
nop
frac t.x, t.x
nop
nop
nop
sub t.x, t, c
nop
nop
nop
poly r, t.x
nop
nop
nop
mul r.w, r, t
mul t.zw, t, t
nop
nop
nop
mul r.yzw, r, t
nop
nop
nop
dp4 r, r, p
```

An alternative would be to use the cosine implementation, since it is easier to approximate (see subsection 6.1.4).

$$\sin(x) = \cos(x - \frac{\pi}{2})$$

6.1.2 radians()

Can be implemented by a multiplication with a constant.

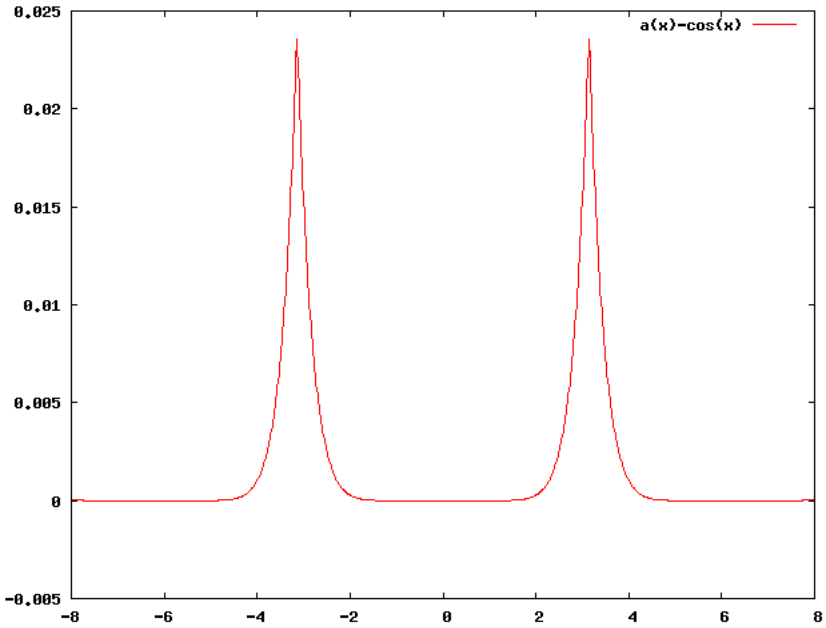

Figure 6.2: Cosine approximation error

6.1.3 degrees()

Can be implemented by a multiplication with a constant.

6.1.4 cos()

$$\cos(c) = \sum_{i=0}^{\infty}(-1)^n \frac{x^{2i}}{(2i)!} \tag{6.7}$$

$$\cos(x) = t(g(\frac{x}{2\pi} + \frac{1}{2}) - \frac{1}{2}) \tag{6.8}$$

$$g(x) = x - \lfloor x \rfloor \tag{6.9}$$

$$t(x) = \sum_{i=0}^{\infty}(-1)^n \frac{(2\pi x)^{2i}}{(2i)!} \approx \sum_{i=0}^{4}(-1)^n \frac{(2\pi x)^{2i}}{(2i)!} = s(x) \tag{6.10}$$

$$\cos(x) := a(x) \approx s(g(\frac{x}{2\pi} + \frac{1}{2}) - \frac{1}{2}) \tag{6.11}$$

The following program places an approximation of $\cos(x)$ in $\mathfrak{x}$.z.

$$\mathfrak{p} = \begin{pmatrix} \frac{(2\pi)^2}{2!} \\ \frac{(2\pi)^4}{4!} \\ \frac{(2\pi)^6}{6!} \\ \frac{(2\pi)^8}{8!} \end{pmatrix}, \mathfrak{c} = \begin{pmatrix} \frac{1}{2} \\ * \\ 1 \\ * \end{pmatrix}, \mathfrak{d} = \begin{pmatrix} \frac{1}{2\pi} \\ * \\ * \\ * \end{pmatrix}, \mathfrak{x} = \begin{pmatrix} x \\ * \\ * \\ * \end{pmatrix}$$

```
mul t.x, x, d
nop
nop
nop
add t.x, t, c
nop
nop
nop
frac t.x, t.x
nop
nop
nop
sub t.x, t, c
nop
nop
nop
mul t.x, t, t
nop
nop
nop
poly t, t.x
nop
nop
nop
dp4 t.z, t, p
nop
nop
nop
add x.z, t, c
```

6.1.5 atan()

$$s(x) := \frac{x}{1 + 0.28x^2} \tag{6.12}$$

$$t(x) := \frac{\pi}{2} - \frac{x}{x^2 + 0.28} \tag{6.13}$$

$$a(x) := \begin{cases} s(x), & |x| \leq 1 \\ t(|x|)\,\mathrm{sign}(x), & |x| > 1 \end{cases} \tag{6.14}$$

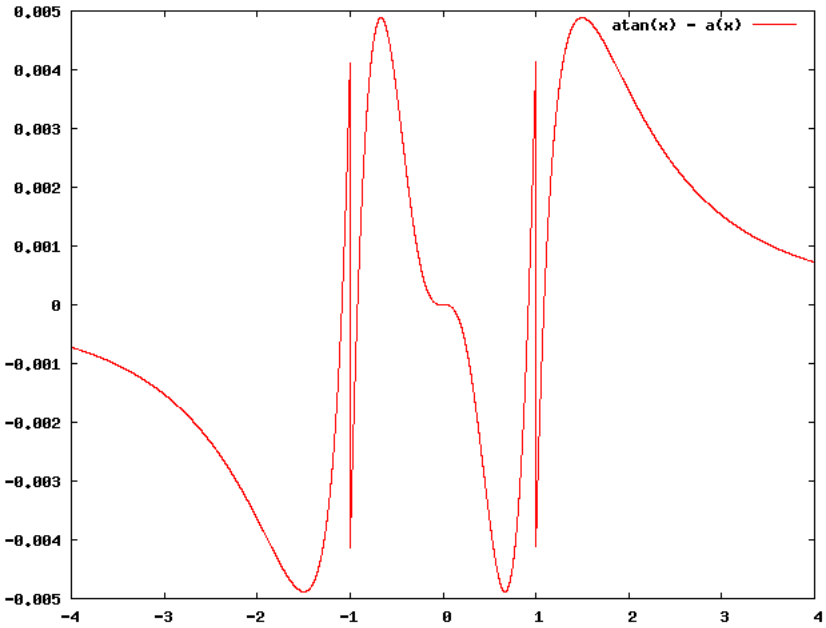

Figure 6.3: Arctangent approximation error

6.1.6 Others: tan(), asin(), acos(), atan(,)

These can be implemented using the other functions in this section or using similar programs.

6.2 Exponential

6.2.1 pow(,)

$$\mathrm{pow}(x, y) = x^y = 2^{y \log_2 x} = \mathrm{exp2}(y \log 2(x)).$$

Exponentiation can be implemented using base 2 exponentiation and logarithm.

6.2.2 exp()

$$\exp(x) = e^x = 2^{x \log_2 e} = \mathrm{exp2}(x \log_2 e).$$

The exponential function can be implemented using base 2 exponentiation and multiplication with a constant.

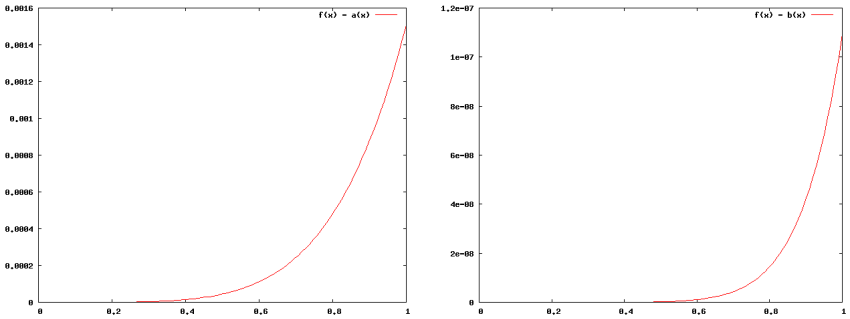

Figure 6.4: Base 2 exponentiation approximation error

6.2.3 log()

$$\log x = \frac{\log_2 x}{\log_2 e} = \log 2(x)\frac{1}{\log_2 e}.$$

The natural logarithm can be obtained using base 2 logarithm and a multiplication with a constant.

6.2.4 exp2()

Base 2 exponentiaton is approximated using a taylor series. To increase precision the taylor series is used for the fractional part only:

$$2^x = 2^{\lfloor x \rfloor} 2^{x - \lfloor x \rfloor} \tag{6.15}$$

$$2^x = \sum_{n=0}^{\infty} \frac{f^{(i)}(0)}{n!} x^n \tag{6.16}$$

$$f(x) := 2^x = e^{x \ln 2} \tag{6.17}$$

$$f^{(i)}(x) = e^{x \ln 2} (\ln 2)^i \tag{6.18}$$

$$2^x = \sum_{n=0}^{\infty} \frac{(\ln 2)^n}{n!} x^n = 1 + \sum_{n=1}^{\infty} \frac{(x \ln 2)^n}{n!} \tag{6.19}$$

$$2^x \approx a(x) = 1 + \sum_{n=1}^{4} \frac{(x \ln 2)^n}{n!} \tag{6.20}$$

$$2^x \approx b(x) = 1 + \sum_{n=1}^{8} \frac{(x \ln 2)^n}{n!} \tag{6.21}$$

The approximation above (equation 6.20) will be sufficient in most cases, if more precision is required another approximation (equation 6.21) can be used, which conforms to OpenGL requirements on precision of floating point operations [17, page 6].

The following program is based on equation 6.20:

$$\mathfrak{f} = \begin{pmatrix} \frac{1}{1!} \\ \frac{1}{2!} \\ \frac{1}{3!} \\ \frac{1}{4!} \end{pmatrix}, \mathfrak{z} = \begin{pmatrix} \ln 2 \\ * \\ * \\ * \end{pmatrix}, \mathfrak{x} = \begin{pmatrix} x \\ * \\ * \\ * \end{pmatrix}$$

```
frac t.x, x.x
flr r.x, x.x
nop
nop
mul t.x, t, z
exp2 r.x, r
nop
nop
nop
poly t, t.x
nop
nop
nop
dp4 t.x, t, f
nop
nop
nop
add t.x, t, f
nop
nop
nop
mul r.x, r, t
```

6.2.5 log2()

The logarithm's power series (equation 6.22) converges slowly. Other approximation include one based on the areatangens hyperbolicus (equation 6.27) and on the arithmetic-geometric mean (equation 6.23), [12, page 220f].

$$\ln(1+x) = \sum_{k=0}^{\infty} \frac{x^{k+1}}{k+1} \tag{6.22}$$

$$|\ln(x) - k(10^{-n}) + k(10^{-n}x)| \leq \frac{n}{10^{2(n-1)}} \tag{6.23}$$

$$k(x) = \frac{\pi}{2M(1, 10^{-n})} \tag{6.24}$$

$$M(x,y) = \lim_{n\to\infty} a_n(x,y) = \lim_{n\to\infty} b_n(x,y) \tag{6.25}$$

$$a_0 = x, b_0 = y, a_{n+1} = \frac{a_n + b_n}{2}, b_{n+1} = \sqrt{a_n b_n} \tag{6.26}$$

$$\ln(x) = \sum_{k=0}^{n} \frac{2}{2k+1}(\frac{x-1}{x+1})^{2k+1} + R_{n+1}(x) \tag{6.27}$$

$$|R_{n+1}(x)| \leq \frac{(x+1)^2}{2|x|}(\frac{x-1}{x+1})^{2n} \tag{6.28}$$

$$\log_2(a2^b) = \log_2(a) + b, \log_2(x) = \frac{\ln(x)}{\ln(2)} \tag{6.29}$$

$$\Rightarrow \log_2(a2^b) \approx l(a2^b) := b + \sum_{k=0}^{3} \frac{2}{\ln(2)(2k+1)}(\frac{a-1}{a+1})^{2k+1} \tag{6.30}$$

Equation 6.23 is the fastest approximation of the logarithm at high precision [12]. At precisions sufficient for computer graphics equation 6.27 will result in a shorter and faster program due to use of dp4 and poly instructions; it's most precise when x is near 1. Using equation 6.29 the logarithm of mantissa and exponent are computed separately resulting in the approximation in equation 6.30. It's implemented in the program below:

$$\mathfrak{c} = \begin{pmatrix} \frac{2}{\ln 2} \\ \frac{2}{3\ln 2} \\ \frac{2}{5\ln 2} \\ \frac{2}{7\ln 2} \end{pmatrix}, \mathfrak{e} = \begin{pmatrix} 1 \\ * \\ * \\ * \end{pmatrix}$$

```
eman u.x, ɾ
nop
nop
nop
add ʊ.x, u, ɕ
sub u.x, u, ɕ
nop
nop
mul ʊ.x, ʊ, ʊ
nop
nop
nop
```

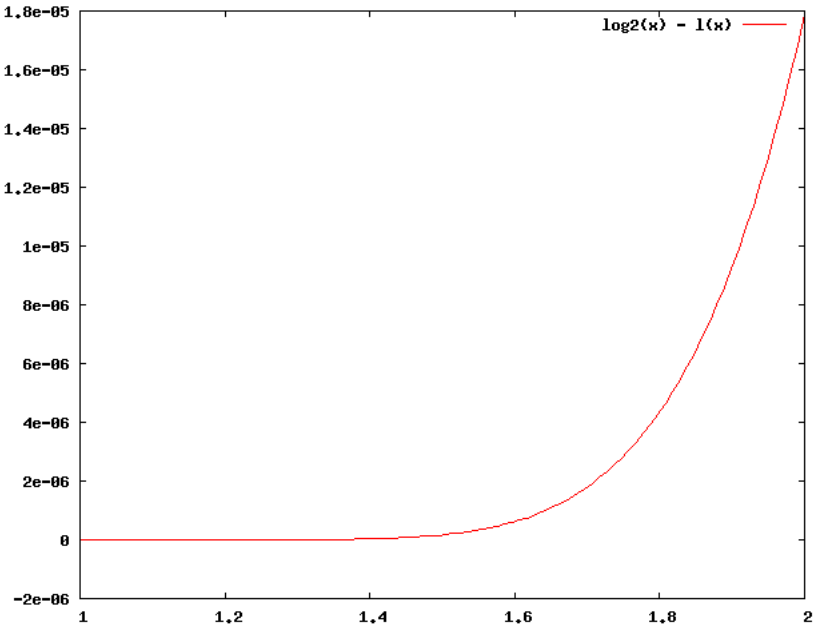

Figure 6.5: Base 2 logarithm approximation error

```
rsq v.x, v.x
nop
nop
nop
mul u.x, u, v
nop
nop
nop
mul u.x, u, u
nop
nop
nop
rsq v, u.x
poly u, u.x
nop
nop
nop
mul u, u, v
nop
nop
eexp v.x, r
dp4 u.x, u, c
```

```
nop
nop
nop
add r.x, v, u
```

6.2.6 sqrt()

$$\sqrt{x} = x^{\frac{1}{2}} = xx^{-\frac{1}{2}}.$$

The square root can be obtained using a reciprocal square root and a multiplication.

6.2.7 inversesqrt()

This function is implemented in hardware by the rsq instruction.

6.3 Common

6.3.1 abs()

This function can be implemented using and with a constant (clearing the sign bits).

6.3.2 sign()

qtob followed by ztoq will give a quadruple of floating-point values which is one for the nonzero components, zero otherwise. and followed by or can be used to give a quadruple of floating-point values which is minus one for negative components, one otherwise. Multiplying these gives the sign function.

6.3.3 floor()

This function is implemented by the flr instruction.

6.3.4 ceil()

$$\text{ceil}(x) = -\,\text{floor}(-x);$$

6.3.5 fract()

This function is implemented by the frac instruction.

6.3.6 mod(,)

$$\text{mod}(x, y) = x - y\lfloor\frac{x}{y}\rfloor = x - y\,\text{floor}(\frac{x}{y}).$$

6.3.7 min(,)

This function is implemented by the min instruction.

6.3.8 max(,)

This function is implemented by the max instruction.

6.3.9 clamp(,,)

$$\text{clamp}(x, y, z) = \min(\max(x, y), z)$$

6.3.10 mix(,,)

$$\text{mix}(x, y, a) = x(1 - a) + ya.$$

6.3.11 step(,)

Can be implemented using less followed by xor and ztoq.

6.3.12 smoothstep(,,)

$$\text{smoothstep}(x, y, z) = t^2(3 - 2t), t = \text{clamp}((z - x)/(y - x), 0, 1). \quad (6.31)$$

The multiplication used above in equation 6.31 is component-wise multiplication of vectors.

6.4 Geometric

6.4.1 length()

$$\text{length}(x) = |x| = \sqrt{< x, x >} = \text{sqrt}(\text{dot}(x, x)).$$

For onedimensional operands this function is the same as abs().

6.4.2 distance(,)

$$\text{distance}(x, y) = |x - y| = \text{length}(x - y).$$

6.4.3 dot(,)

This function is implemented by the dp3 and dp4 instructions for three- and fourdimensional operands. mul will do for scalars.

6.4.4 cross(,)

The cross product $\mathfrak{r}$ of $\mathfrak{x}$ and $\mathfrak{y}$ can be calculated as in its definition:

```
swiz tmp0, x.yzxx
swiz tmp1, y.zxyx
swiz tmp2, y.yzxx
swiz tmp3, x.zxyx
nop
mul tmp0, tmp0, tmp1
nop
mul tmp2, tmp2, tmp3
nop
nop
nop
sub r, tmp0, tmp2
```

6.4.5 normalize()

$$\text{normalize}(x) = \frac{x}{|x|} = \frac{x}{\sqrt{< x, x >}} = x\,\text{inversesqrt}(\text{dot}(x, x)).$$

A vector x in four- or threedimensional space can be normalized by a dp4 or dp3 and reciprocal square root and multiplication.

6.4.6 fttransform()

This function implements equation 4.1.

6.4.7 faceforward(,,)

$$\text{faceforward}(x, y, z) = x\,\text{sign}(\text{dot}(z, y))$$

6.4.8 reflect(,)

$$\text{reflect}(x, y) = x - 2 < x, y > y = x - 2\,\text{dot(x, y)}\,x.$$

6.4.9 refract(,,)

6.5 Matrix

6.5.1 matrixCompMult(,)

Component-wise multiplication can be implemented using mul instructions.

6.5.2 outerProduct(,)

The outerProduct requires n swiz and n mul instructions to implement. The following code fragment computes $\mathfrak{M}$ = outerProduct($\mathfrak{a}, \mathfrak{b}$).

```
swiz 𝔐1, 𝔞.xxxx
swiz 𝔐2, 𝔞.yyyy
swiz 𝔐3, 𝔞.zzzz
swiz 𝔐4, 𝔞.wwww
mul 𝔐1, 𝔐1, 𝔟
mul 𝔐2, 𝔐2, 𝔟
mul 𝔐3, 𝔐3, 𝔟
mul 𝔐4, 𝔐4, 𝔟
```

6.5.3 transpose()

Implementing this function is expensive. Transposing a $x \times y$ matrix requires xy swiz instructions. Compiler optimizations will eliminate some transpositions.

6.6 Vector Relational

6.6.1 lessThan(,)

Implemented by the less instruction.

6.6.2 greaterThan(,)

$$x > y \Leftrightarrow -x < -y \Rightarrow \text{greaterThan}(x, y) = \text{lessThan}(-x, -y)$$

6.6.3 notEqual

xor followed by ztob will implement this function.

6.6.4 not()

Negation can be implemented using xor with a constant.

6.6.5 Others: lessThanEqual, greaterThanEqual, equal, any, all

These are easy to implement using the other functions or jumps.

6.7 Texture Lookup

For each texture lookup function there exist variants for 1D, 2D, 3D textures and cube maps. Since the type of texture is determined by the texture unit used, 1D, 2D, 3D and Cube have been replaced by ? in the following function names.

6.7.1 Texture?

This function is implemented by writing the texture coordinates and reading the texel.

6.7.2 Texture?Proj

$$\text{Texture?DProj}(x, y) = \text{Texture?D}(x, \frac{y}{y_3}) = \text{Texture?D}(x, \text{inversesqrt}(y^2)).$$

6.7.3 Texture?Lod

This function is implemented by writing the texture coordinates and reading the texel.

6.7.4 Texture?ProjLod

$$\text{Texture?ProjLod}(x, y, z) = \text{Texture?DLod}(x, \frac{y}{y_3}, z) = \text{Texture?DLod}(x, \text{inversesqrt}(y^2), z).$$

6.7.5 Shadow textures

The texture lookup functions for depth textures are implemented just like the corresponding functions for ordinary textures.

6.8 Fragment processing

See subsection 2.3.2 above for a discussion of potential implementations of derivatives.

6.9 Noise

Since there are no special instructions for Perlin noise implementations will be slow. If unused textured units are available use of a noise texture would be a faster alternative.

Part III

Hardware implementation, supporting software

Chapter 7

Selected aspects of hardware implementation

7.1 Pipeline

The processor's architecture has been created with the possibility of a pipelined implementation (figure 7.1) in mind. The number of execution stages has been optimized for dot products (subsection 7.3.1).

In the first stage a command is fetched from program memory. The operands are selected, possibly involving a read access to the index registers.

In the second stage operands are fetched from general purpose registers or texture units.

Next come three execution stages. Most instructions are distributed over them. However the result of index register writes and indirect jumps needs to be available earlier, so these instructions have to complete in the first execution stage.

In the last stage a result can be written to a general purpose register or used as coordinates (and level of detail parameter) for a texture unit.

7.2 Environment

The processor will be embedded into a graphics pipeline to execute vertex or fragment shaders.

Interaction with the rest of the graphics pipeline mostly takes place through the general purpose registers. The graphics pipeline will place input values in them making the processor run the program and read output values from there after an end instruction.

Large amounts of data are typically stored in textures. The processor will supply texture coordinates (and possibly a level of detail parameters) to texture units and read filtered texels from them.

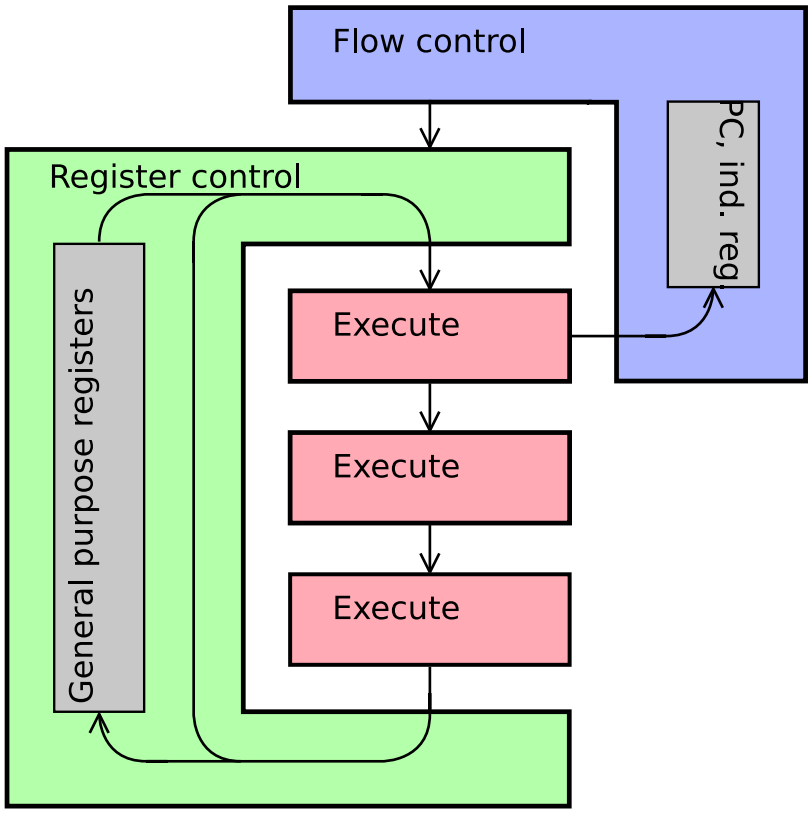

Figure 7.1: Pipeline

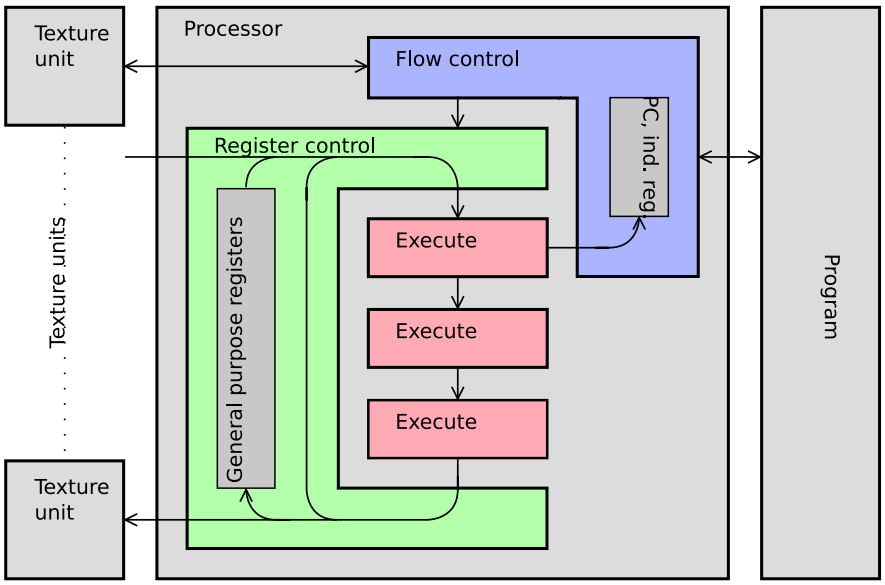

Figure 7.2: Processor and parts of its environment

7.3 Instructions

7.3.1 dp4

The dp4 instruction can be subdivided into floating-point additions and multiplications which easily map to the pipeline's execution stages as shown in figure 7.3. Since dot products are one of the most common operations in shaders the pipeline has been designed for efficient implementation of dot products.

7.3.2 rsq

$$(a2^b)^{-\frac{1}{2}} = a^{-\frac{1}{2}}2^{-\frac{b}{2}} = 2^{\lfloor\frac{b}{2}\rfloor}\begin{cases}a^{-\frac{1}{2}}, b\,\text{even}\\(2a)^{-\frac{1}{2}}, b\,\text{odd}\end{cases} \tag{7.1}$$

$$f(x) := x^{-\frac{1}{2}}, f'(x) = -\frac{1}{2}x^{-\frac{3}{2}} \tag{7.2}$$

$$f(a) \approx f(l(a)) + (a - l(a))f'(l(a)) \tag{7.3}$$

$$e < f(1+\epsilon) - f(1) + \epsilon f'(1) =: E(\epsilon) \tag{7.4}$$

$$T(x) := E(\frac{1}{x}) \tag{7.5}$$

The reciprocal square root is the only instruction that requires relatively complex hardware.

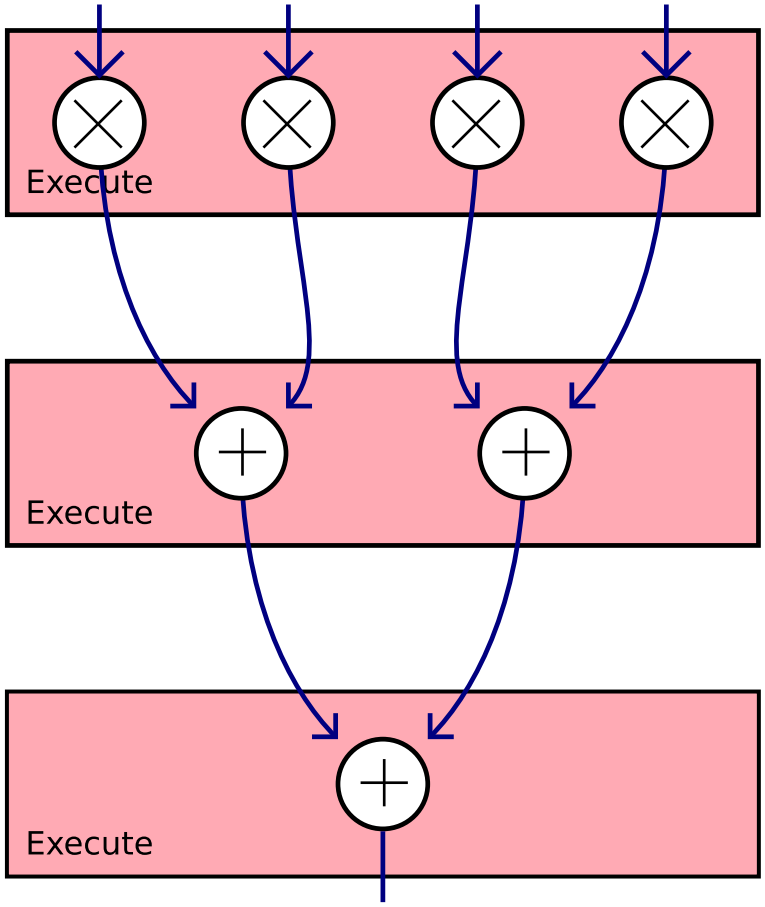

Figure 7.3: dp4 instruction in execution stages of pipeline

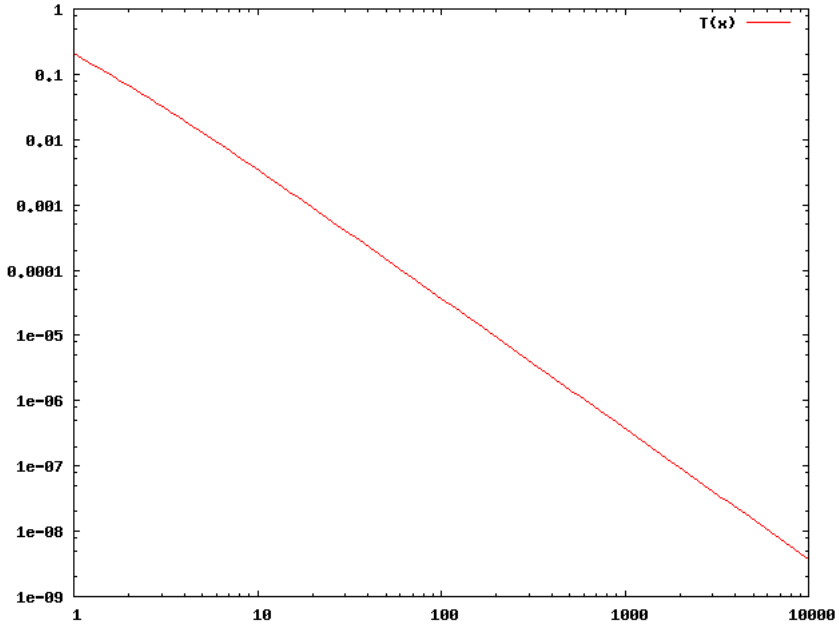

Figure 7.4: Reciprocal square root approximation error upper bound as function of table size

As can be seen in equation 7.1 calculating the reciprocal square root of a number in floating-point representation can be reduced to calculating the reciprocal square root of a number in $[1, 4[$ and shifting the exponent by one bit. Since $f([1, 4[) \subseteq]\frac{1}{2}, 1]$ calculation of the decreasing the shifted exponent by 1 will yield the correct exponent. The mantissa can be approximated (equation 7.3) using tables for the f and its derivative (equation 7.2) where the function l gives the nearest lower value for which a table entry exists. Since $|f'|$, as a function defined on $[1, 4[$ has it's maximum at 1 we can estimate the error e using equation 7.4 where ϵ is the maximum distance between two values for which table entries exist.

The structure of equation 7.3 is well-suited for implementation using the hardware pipeline for dot products.

Chapter 8

Cycle-accurate SystemC model

A simulator of the processor has been implemented in SystemC. It is structured like figure 7.2 with the processor itself implemented cycle-accurate and partially at register transfer level. The simulator emulates the processor's environment including program memory and texture units. It has a textual user-interface that allows changing simulation parameters and observing the processor's behaviour.

8.1 Simulator Commands

8.1.1 run

The run command takes one integer argument. It runs the simulation for as many clock cycles as the argument specifies.

8.1.2 program

The program command takes one argument, which is interpreted as a filename. It makes the simulator use the specified file as program memory (files output by the assembler can be used).

8.1.3 data

The data command takes one argument, which is interpreted as a filename. The file is used to set the data in the general-purpose registers. The file content is interpreted as follows: Each line should start with a data type specifier (one of vec4, ivec4 and bvec4) and contain 4 whitespace-separated values to be placed in a register. The line number corresponds to the register written. If the number of lines in the file is less than the number of general-purpose registers the contents of the other registers is undefined.

8.1.4 register

This command takes one integer argument. It displays the contents of the specified register as vector of floating-point values, integers and booleans.

8.1.5 registers

The registers command takes two arguments, which are interpreted as a range of registers. It performs the operation of the preceding command for all registers in the specified range.

8.1.6 help

Display the list of commands.

8.1.7 quit

Exit the simulation.

8.2 Example session

Assuming assembler output from the following program in the norm.bin file:

```
; Normalization
register 0, input
register 1, temp
register 2, temp'
register 3, output

dp4 temp.x, input, input
nop
nop
nop
rsq temp', temp.x
nop
nop
nop
mul output, input, temp'
nop
nop
nop
nop
nop
```

And the following register data in the normdata file:

```
vec4 42 23 0 0
```

A simulator session might look as follows:

```
>program norm.bin
>data normdata
>registers 0 3
vec4: (42, 23, 0, 0) ivec4: (1109917696, 1102577664, 0, 0) bvec4: (1, 1, 0, 0)
vec4: (0, 0, 0, 0) ivec4: (0, 0, 0, 0) bvec4: (0, 0, 0, 0)
vec4: (0, 0, 0, 0) ivec4: (0, 0, 0, 0) bvec4: (0, 0, 0, 0)
vec4: (0, 0, 0, 0) ivec4: (0, 0, 0, 0) bvec4: (0, 0, 0, 0)
>run 15
>registers 0 3
vec4: (42, 23, 0, 0) ivec4: (1109917696, 1102577664, 0, 0) bvec4: (1, 1, 0, 0)
vec4: (2293, 0, 0, 0) ivec4: (1158631424, 0, 0, 0) bvec4: (1, 0, 0, 0)
vec4: (0.0208832, 0.0208832, 0.0208832, 0.0208832) ivec4: (1017844566, 1017844566, 1017844566, 10
vec4: (0.877096, 0.480315, 0, 0) ivec4: (1063291233, 1056304076, 0, 0) bvec4: (1, 1, 0, 0)
>_
```

Chapter 9

Assembler

A two-pass assembler using the mnemonics in chapter 3 has been written. It supports the declaration and use of symbolic names for registers and can assemble the programs contained in this document.

The assembler takes exactly two arguments when invoced: The name of the input and output file.

Part IV

Adaptions for use in embedded systems

Embedded systems typicallly have low graphics quality requirements (such as low resolution, low color depth diplays). Often vertex processing isn't hardware-accelerated. Adaptions for embedded systems should thus focus on the fragment shader. For embedded systems only the OpenGL ES specification, not the full OpenGL specification is relevant. OpenGL ES has lower graphics quality requirements and less features; this can be exploited for architecture changes resulting in reduced hardware complexity in implementations.

Chapter 10

Execution environment

10.1 Data Types

Half precision floating point (16 bit) is sufficient for OpenGL ES in the fragment shader [25, section 4.5.1 (page 31)]. Thus size of data types can be reduced: All data types are 64 bit wide 4-component vectors. The individual data types correspond to the types in part I with precision reduced.

10.2 Registers

10.2.1 general purpose registers

Since data types are smaller (see above) the size of the general purpose registers can be reduced to 64 bit treated as 4-component vector by most instructions.

10.2.2 index registers

Array indexing is severely restricted in GLSL ES [25, section 10.25 (page 95) and appendix A (page 102)]. However index registers aren't expensive to implement and could thus be kept even for embedded systems.

10.3 Texture Units

Large parts of the texture unit are used during texture lookup only. To reduce the amount of this most of the time unused hardware part of the texture unit's functionality can be moved to the shader.

10.3.1 level of detail and derivatives

In contrast to GLSL GLSL ES does not require access to derivatives in the fragment shader. However if the derivatives can be acessed in the shader level

of detail calculations can be done in the shader instead of the texture unit.

The following approximation could be used to calculate the level of detail parameter λ from texture coordinates s and t (b is a user-defined value):

$$d := \begin{pmatrix} |\frac{\partial s}{\partial x}| \\ |\frac{\partial s}{\partial y}| \\ |\frac{\partial t}{\partial x}| \\ |\frac{\partial t}{\partial y}| \end{pmatrix}, \rho := \max\{d_0, d_1, d_2, d_3\}, \lambda := \log_2(\rho + b). \tag{10.1}$$

In subsection 2.3.2 multiple ways to access derivatives from shaders have been introduced. In OpenGL ES only control constructs that can be unrolled to contain only forward jumps are allowed [25, appendix A (page 101)]. Furthermore texture accesses are not allowed in conditionally executed code [25, appendix A (page 103)]. Thus lockstep operation would be the preferred way to obtain derivatives.

10.3.2 filtering

Trilinear filtering is expensive both in memory bandwidth and texture filtering hardware. Due to the cost in memory bandwidth bilinear filtering is preferred in embedded systems. However the OpenGL ES spec mandates support for both filtering modes. Since trilinear filtering requires more than twice as much hardware as bilinear filtering it can be desirable to move support for trilinear filtering to the shader. The shader would do two bilinearily filtered texture lookups to different mipmap levels of the same texture and interpolate the results. An additional instruction allowing for a faster implementation of the mix function (subsection 6.3.10) should be considered in this case.

Chapter 11

Instructions

Due to the restrictions in control flow and indexing mentioned in the last chapter integer instructions are less important in embedded systems; integer instructions that aren't used often or can be rather easily emulated using floating-point instructions could be left out. If index registers aren't implemented all integer instructions and the ijmp indirect jump should be left out.

To ease implementation of equation 10.1 two additional instructions should be considered: An instruction that returns the maximum of the entries of its source operand vector and a base 2 logarithm, which could be implemented in hardware similar to the reciprocal square root.

Bibliography

[1] Khronos Group. http://www.khronos.org.

[2] OGRE. http://www.ogre3d.org.

[3] Open Graphics Project. http://www.opengraphics.org.

[4] Aaftab Munshi. OpenGL ES Common Profile Specification 2.0 (Version 1.19). http://www.khronos.org/files/opengles_spec_2_0.pdf.

[5] Cass Everitt et al. ARB_fragment_program (revision 27), 2006. http://opengl.org/registry/specs/ARB/fragment_program.txt.

[6] Chris Hecker. Top Game Developers Call on Microsoft to Actively Support OpenGL. http://chrishecker.com/OpenGL/Press_Release.

[7] James H. Clark. The Geometry Engine: A VLSI Geometry System for Graphics. In *SIGGRAPH '82: Proceedings of the 9th annual conference on Computer graphics and interactive techniques*, pages 127–133, New York, NY, USA, 1982. ACM.

[8] John Carmack. Plan. http://www.exaflop.org/docs/d3dogl/d3dogl_jc_plan.html.

[9] John Carmack et al. Doom 3 Linux Demo. ftp://ftp.idsoftware.com/idstuff/doom3/linux/old/doom3-linux-1.1.1286-demo.x86.run.

[10] John Carmack, Mark Kilgard, Brian Paul et al. ARB_vertex_program (revision 45), 2004. http://opengl.org/registry/specs/ARB/vertex_program.txt.

[11] John Kessenich. The OpenGL Shading Language (Language version 1.20, document revision 8, 2006. http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf.

[12] Jonathan M. Borwein, Peter B. Borwein. *Pi and the AGM*. 1987.

[13] Philipp Klaus Krause. SystemC model of the OpenGL pipeline. http://colecovision.eu/graphics/fragment_pipeline.tar.gz.

[14] Mark J. Kilgard. NV_texture_shader2 (revision 9), 2004. http://www.opengl.org/registry/specs/NV/texture_shader2.txt.

[15] Mark J. Kilgard. NV_texture_shader (revision 29), 2007. http://www.opengl.org/registry/specs/NV/texture_shader.txt.

[16] Mark J. Kilgard. NV_texture_shader3 (revision 9), 2007. http://www.opengl.org/registry/specs/NV/texture_shader3.txt.

[17] Mark Seegal, Kurt Akeley. The OpenGL graphics system. A specification (Version 1.4), 2002. http://opengl.org/documentation/specs/version1.4/glspec14.pdf.

[18] Mark Seegal, Kurt Akeley. The OpenGL graphics system. A specification (Version 1.5), 2003. http://opengl.org/documentation/specs/version1.5/glspec15.pdf.

[19] Mark Seegal, Kurt Akeley. The OpenGL graphics system. A specification (Version 2.1 - December 1, 2006), 2006. http://www.opengl.org/registry/doc/glspec21.20061201.pdf.

[20] Morgan McGuire and Max McGuire. Steep parallax mapping. *I3D 2005 Poster*. http://www.cs.brown.edu/research/graphics/games/SteepParallax/index.html.

[21] Microsoft corporation. Pipeline Stages (Direct3D 10). http://msdn2.microsoft.com/en-us/library/bb205123.aspx.

[22] Pat Brown. NV_geometry_program4 (revision 6), 2006. http://www.opengl.org/registry/specs/NV/geometry_program4.txt.

[23] Pat Brown, Barthold Lichtenbelt. EXT_geometry_shader4 (revision 16), 2007. http://www.opengl.org/registry/specs/EXT/geometry_shader4.txt.

[24] Pat Brown, Barthold Lichtenbelt. NV_geometry_shader4 (revision 16), 2007. http://www.opengl.org/registry/specs/NV/geometry_shader4.txt.

[25] Robert J. Simpson, John Kessenich. The OpenGL ES Shading Language (language version 1.00, document revision 14). http://www.khronos.org/files/opengles_shading_language.pdf.

[26] Estle Ray Mann Thomas T. Goldsmith Jr. Cathode-Ray Tube Amusement Device, 1948. US Patent 2455992.

[27] Tony Smith. id Software's Carmack calls for OpenGL watchdog. http://www.theregister.co.uk/1999/12/14/id_softwares_carmack_calls/.

[28] Victor Moya, Carlos Gonzalez, Jordi Roca, Agustin Fernandez, Roger Espasa. Shader Performance Analysis on a Modern GPU Architecture. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 355–364, Washington, DC, USA, 2005. IEEE Computer Society.

[29] Víctor Moya, Carlos González, Jordi Roca, Agustín Fernández, Roger Espasa. A Single (Unified) Shader GPU Microarchitecture for Embedded Systems, 2005. `https://attila.ac.upc.edu/wiki/images/a/af/Tilarin.pdf`.