En tilebaserad triangelutritare i hårdvara

A tile-based triangle rasterizer in hardware

Fredrik Ehnbom

Examensarbete
Stockholm 2005
ISSN 1651-5633
TRITA-2IT Ex arb 2005:xxx
ISRN KTH/2IT/–SE

TRITA: 2005: xxx

KTH Tillämpad informationsteknik

Examensarbete
vid Institutionen för tillämpad IT

Thesis project
at Department of Applied Information Technology

En tilebaserad triangelutritare i hårdvara
A tile-based triangle rasterizer in hardware

Fredrik Ehnbom

Godkänt	Examinator Bengt Koren	Underskrift

Sammanfattning

Detta dokument skrevs som en rapport för mitt examensarbete och beskriver implementationen av en triangelutritare skriven i VHDL. Ett av projektets syften är att dokumentera hur triangelutritaren fungerar, då lite tidigare har dokumenterats på denna detaljnivå. Triangelutritaren är tilebaserad och använder edge-funktioner och areala koordinater för parameterinterpollering. Tre delkomponenter utgör kärnan, en enhet som hittar tiles som triangeln överlappar, en enhet som hittar pixlar innuti triangeln i en tile, och en enhet som slutligen beräknar pixelns slutgiltiga färgvärde. Hela kretsen är syntetiserbar på en Virtex-II FPGA i 100 MHz samt klarar av bilinjärfiltrerad och perspektivkorrekt texturering, gouraud-shading och z-buffring.

Abstract

This document was written as a report for my thesis project and describes the implementation of a triangle rasterizer written in VHDL. One of the projects purposes is to document how the triangle rasterizer operates, as little has earlier been written at this detailed level before. The triangle rasterizer is tile-based and uses edge functions and areal coordinates for parameter interpolation. Three subcomponents represents the core, one unit that finds the tiles that the triangle overlaps, one unit that finds pixels inside the triangle in a tile, and one unit that finally calculates the pixels final color. The whole chip is synthesizable on a Virtex-II FPGA at 100 MHz and can render triangles with bilinear filtered perspective correct texture mapping, goraud shading and z-buffering.

Thank you Tomas Akenine-Möller, Jon Hasselgren and Anders Nilsson for your help and support during this project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

The Lund University Graphics Group (LUGG), constantly develops new algorithms for computer graphics. Previously they have been limited to test these algorithms in software only.

1.2 Purpose

The purpose with this project and this report is to create a prototype for rasterization of triangles in hardware and to document the functionality of the hardware. There has not been many public documents describing graphics hardware at this level of detail before.

The prototype will be used as a base for further research where LUGG can test their algorithms in hardware.

1.3 Goals and simplifications

The goal is to have a working prototype of a tile based rasterizer which can render z-buffered triangles with perspective correct texture mapping. The implementation will be in VHDL and should later run on an FPGA board.

As depicted in Figure 1.1, only a small subset of the pipeline will be implemented in hardware. Chapter 5 gives suggestions for what could be improved in the future.

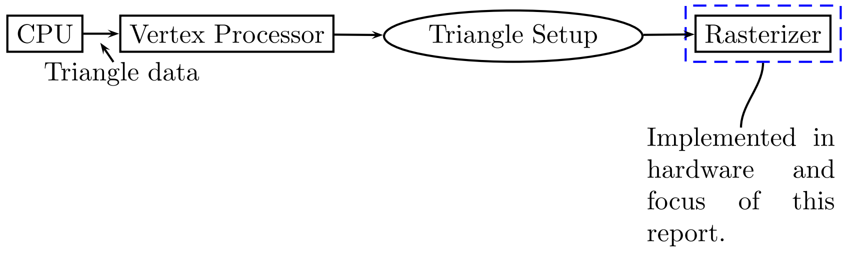

Figure 1.1: Rendering pipeline overview. Only the rasterization step is handled in this implementation.

Chapter 2

Theoretical background

Before the implementation details are shown, some theoretical background for the techniques used are needed.

2.1 Tile based triangle traversal

In *tile based triangle traversal* [7], the pixels of the triangle are handled in a tiled fashion, meaning that the screen is divided into tiles of n×n pixels, and that the pixels of one tile is processed first before moving on to the next. This tiled division allows for simple occlusion culling schemes [2] and increases the locality of pixels which improves hit rates in texture caches [5].

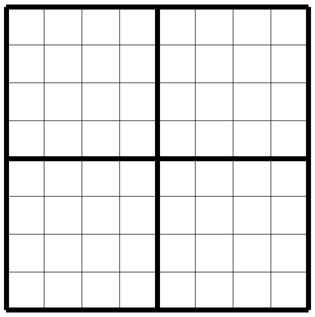

Figure 2.1: Tiles consisting of 4×4 pixels each.

2.2 Edge functions

An edge function [9], as defined by a line between two points in its implicit form

$$E(x, y) = ax + by + c = 0, \tag{2.1}$$

where *(a,b)* can be interpreted as the *normal*, a vector perpendicular to the line itself, and c is the line's distance from the centre along this normal.

When the result for testing a point (x,y) with an edge function is

=0 the point is exactly on the edge.

>0 the point is in the positive halfspace of the edge, ie. the same halfspace as the normal points to.

<0 the point is in the negative halfspace, the opposite halfspace of the one the normal points to.

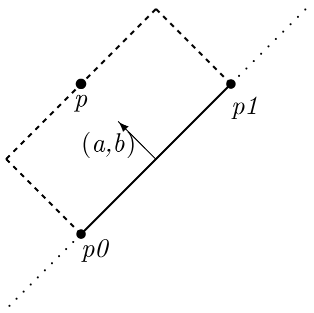

Figure 2.2: An edge formed between points *p0* and *p1* with normal *(a,b)*. When tested with a point *p*, the area of the dashed rectangle is the result, presupposed the length of the normal corresponds to the length of the line.

Equation (2.2) is used to calculate the edge function between the two points *(x0,y0)* and *(x1,y1)*.

$$\begin{aligned} a &= -(y1 - y0) \\ b &= x1 - x0 \\ c &= -(a * x0 + b * y0) \end{aligned} \tag{2.2}$$

As depicted in Figure 2.2, the result of an edge function is the rectangular area formed between the points *p0*, *p1* and the testing point *p*. Of course, this is only valid when the length of the normal corresponds to the length of the line formed between the two points, as it does when using equation (2.2).

Three edge functions can be used together to form the three edges of a triangle as can be seen in Figure 2.3. Then, to test if a point is inside the triangle, the point's coordinates are used with the edge functions to determine in which halfspace of the line the point is in. If all three edges give a positive result, the point is inside the triangle.

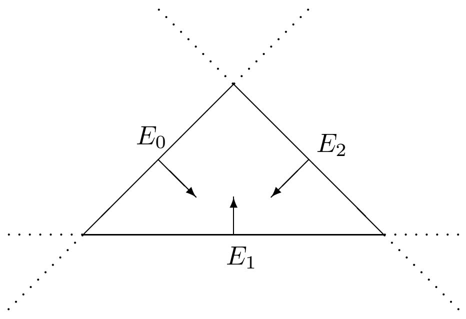

Figure 2.3: Three edges, E_0, E_1 and E_2, forming a triangle.

2.3 Barycentric coordinates

Barycentric coordinates are used to find the center of mass for a geometric object [11] and "Homogenous barycentric coordinates are barycentric coordinates normalized such that they become the actual areas of the subtriangles." [12] From Figure 2.4 we see the three homogenous barycentric coordinates *u, v, w*, representing the areas of the subtriangles in the larger triangle $\triangle ABC$.

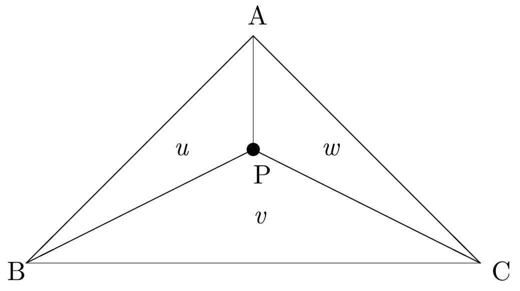

Figure 2.4: Homogenous barycentric coordinates (u,v,w).

Common in computer graphics applications, the homogenous barycentric coordinates are normalized by the area of the whole triangle. This means that the coordinates have a value between zero and one, and that

$$u + v + w = 1, \tag{2.3}$$

for any point inside the triangle. This specific case of normalized barycentric coordinates where equation (2.3) holds true, are called *Areal coordinates* [10].

From Figure 2.4, we can see that as the point P moves closer to the triangle corner A, the area of the subtriangle v will grow larger. The same holds true for u when P moves towards C and w when P moves towards B.

This observation together with the normalization by the whole triangle area can be used to find interpolated values for any point inside the triangle. Let *i,*

j and k be any parameter specified at vertex A, B and C respectively, then the interpolated value for the point P would be

$$inter = u * k + v * i + w * j. \tag{2.4}$$

2.4 Perspective correction

When performing perspective projection, objects that are further away appears smaller than objects near the camera. The parameters specified at each vertex in a triangle should also have different effect on the triangle depending on their weight, otherwise, the parameters at each vertex will contribute equally and will thus destroy the illusion of perspective. The weight at each vertex is the w component of a homogenous vector $p = (x, y, z, w)$.

Section 3.2.4, on page 12, explains how perspective correction has been implemented.

2.5 Fixed point number representation

Fixed point numbers is a way to represent fractional numbers which can be used with existing integer arithmetic units. Numbers are said to be on *i.f* form, where i are the number of bits used to represent the integer part and f are the number of bits used for the fractional. Table 2.1 shows the smallest possible fractional representation given some number of bits.

Bits	Resolution	Resolution (numeric)
1	$\frac{1}{2}$	0.50000000
2	$\frac{1}{4}$	0.25000000
3	$\frac{1}{8}$	0.12500000
4	$\frac{1}{16}$	0.06250000
5	$\frac{1}{32}$	0.03125000
6	$\frac{1}{64}$	0.01562500
7	$\frac{1}{128}$	0.00781250
8	$\frac{1}{256}$	0.00390625
9	$\frac{1}{512}$	0.00195313
10	$\frac{1}{1024}$	0.00097656
11	$\frac{1}{2048}$	0.00048828
12	$\frac{1}{4096}$	0.00024414
13	$\frac{1}{8192}$	0.00012207
14	$\frac{1}{16384}$	0.00006104
15	$\frac{1}{32768}$	0.00003052
16	$\frac{1}{65536}$	0.00001526

Table 2.1: Growth of possible resolution as the number of bits are increased.

Fixed point mathematics is used everywhere in the designed hardware where fractional numbers are needed, but as it is not necessary to understand for this report, a detailed explanation has been left out. The interested reader can instead refer to [3].

Chapter 3

Implementation

3.1 Overview

The implementation has three larger tasks, each implemented in its own chip; handling input from the programmer, rasterizing triangles and a RAMDAC[1].

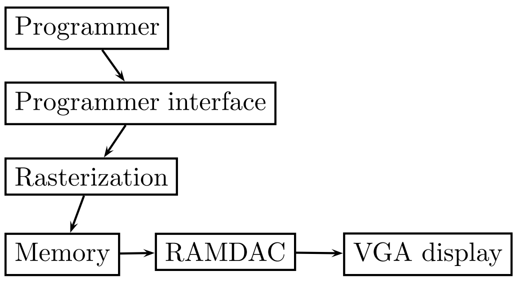

Figure 3.1: Data flow.

3.2 Rasterization

3.2.1 Overview

In short, there are three steps involved in the rasterization process. First, tiles overlapping the triangle are found, then each pixel in the overlapping tile is tested if it is inside the triangle, and finally the pixel is shaded, textured and possibly put into the framebuffer.

Figure 3.2 gives an overview of the rasterization process.

[1]A RAMDAC is a digital to analog convertor that gets its data from a Random Access Memory. This is used for displaying the framebuffer on a VGA display.

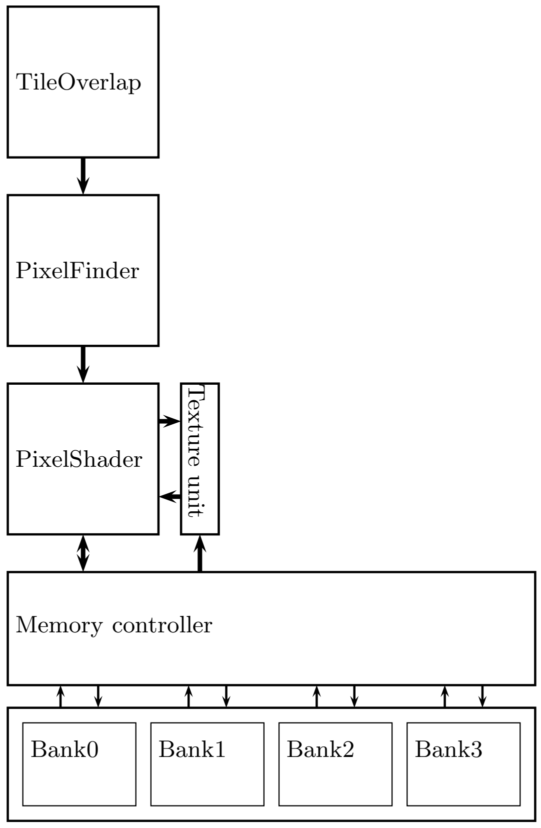

Figure 3.2: Rasterization unit block diagram.

3.2.2 Synchronization

All core components depends on data calculated by a previous component, and thus synchronization between the components are needed to know when it is possible to output new data and when the component is receiving new data. Figure 3.3 explains how synchronization has been implemented.

3.2.3 Tile overlap unit

The *Tile overlap unit* finds the tiles that the current triangle overlaps. Starting at the topmost vertex, the algorithm walks left until it finds a tile outside the triangle, then it proceeds to the right until it finds a tile outside on the other side of the triangle, and thus a full line of tiles inside the triangle has been processed. The algorithm proceeds down to the next row of tiles and repeats the procedure once again with reversed forward and backward directions. Figure 3.5 depicts the algorithm.

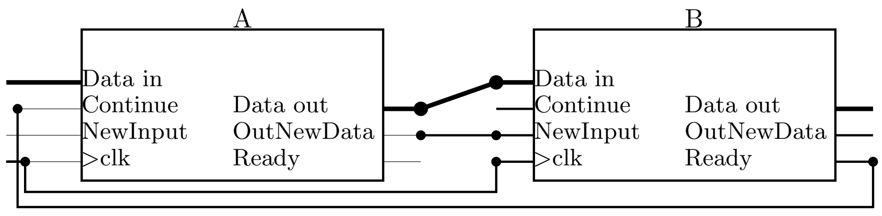

Figure 3.3: Synchronization between two units. When unit B is *Ready*, unit A can *Continue* to send new data. When unit A does send new data, unit B is notified with its *NewInput* connection to unit A's *OutNewData*.

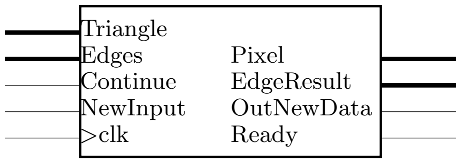

Figure 3.4: Tile overlap unit.

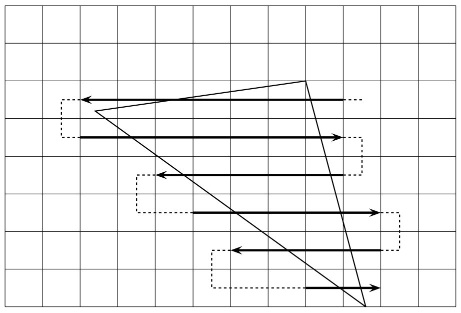

Figure 3.5: The zigzag testing pattern used by the *Tile Overlap unit*. The dashed lines represent tiles that are visited by this unit, but not sent for processing to the next.

To conclude that the tile is completely outside of the triangle, only one corner, for each of the edge functions needs to be tested [1].

Once an overlapped tile has been found, the tile's top left corner, together with the results of the edge functions for that corner, is sent to the next unit, the *Pixel Finder*.

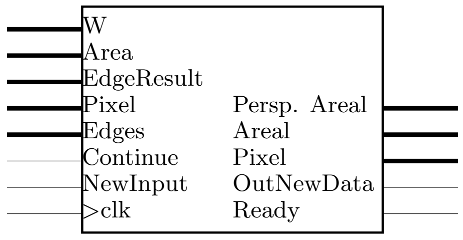

Figure 3.6: Pixel Finder unit.

3.2.4 Pixel Finder unit

The *Pixel Finder unit* is responsible to test all the pixels in a tile to see if they are inside the triangle. Once a pixel has been found inside the triangle, its screen coordinates as well as the non perspective correct and perspective correct areal coordinates for that pixel are output.

The unit starts in the top left corner of the tile and goes right until it reaches the end of the tile. It then goes down to the next row and proceeds to test the pixels to the left. This produces a zigzag pattern [9][2] as can be seen in Figure 3.7. The benefit of this pattern is that each testing step the tested pixel is moved in only one direction and only one pixel at a time, and thus the change from testing one pixel to the next can be done using only additions. Other patterns [6] have this benefit also, but this was chosen because of its simplicity.

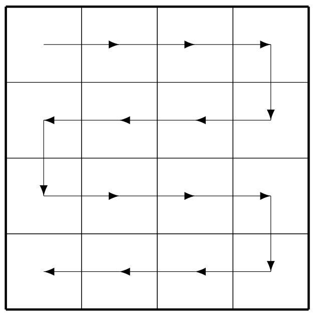

Figure 3.7: The zigzag testing pattern used by the *Pixel finder unit* when searching for pixels inside a triangle.

Whether to move right or left is just a matter of determining whether the current y-position is even or not.

Input data are the coordinates to the tiles top left pixel, as well as the results for the edge functions for that pixel. The task of testing a pixel against an edge is delegated to three *Edge Evaluator* units, which each tests one of the three edges of the triangle. Another subcomponent, the *Areal coordinate calculator* has the task of calculating the areal coordinates and the *Perspective correct areal coordinate calculator* does exactly what its name suggests.

Edge Evaluator

This unit's inputs are the result of the edge function for the top left corner of a tile, as well as the edge-normal (a,b), and the output is the new edge result based on the current pixel position.

The very same zigzag pattern mentioned above is also used by this unit, and thus it inherits the property of being able to find the next steps result by just using additions. To be more specific, either a,$-a$ or b is added to the current edge result making a move to the right, left or down respectively.

When an edge passes exactly through a pixels center, and the edge is shared between two triangles, both triangles will draw to that pixel. This will cause problems when rendering translucent polygons, rendering to the stencil buffer and can cause z-fighting between the two triangles. To prevent this, tie-breaking tests are used as suggested in [6].

Areal coordinate calculator

The result from the edge functions are the areas of the subtriangles multiplied by two. Since we are interested in the areal coordinates we need to divide by two times the triangle area.

Given the results of the three edge functions, E_1, E_2 and E_3, and the triangle area $\triangle A$, the areal coordinates u,v,w, for that pixel can be calculated:

$$u = \frac{E_1}{2\triangle A},\ v = \frac{E_2}{2\triangle A},\ w = \frac{E_3}{2\triangle A}. \tag{3.1}$$

Alternatively, as the areal coordinates have the property of (2.3), the calculation can be performed as

$$u = \frac{E_1}{2\triangle A},\ v = \frac{E_2}{2\triangle A},\ w = 1 - u - v, \tag{3.2}$$

instead.

Since division is a more expensive operation than multiplication[2], $\frac{1}{2\triangle A}$ is precalculated in the triangle setup and the edge results are then multiplied by this term instead.

Perspective correct areal coordinate calculator

Calculating the perspective correct areal coordinates is done in two steps. First, the Edge results are divided by their respective weights W.

$$u_1 = \frac{E_1}{W_3},\ v_1 = \frac{E_2}{W_1},\ w_1 = \frac{E_3}{W_2} \tag{3.3}$$

[2]The FPGA board used in our implementation can perform multiplication with a latency of one cycle [13], whereas divisions have at best a latency of $n+2$ cycles, where n is the number of bits in the dividend [14].

Then, to get areal coordinates, we need to normalize the components with the full triangle area as explained in section 2.3.

$$u_{pc} = \frac{u_1}{u_1 + v_1 + w_1},\ v_{pc} = \frac{v_1}{u_1 + v_1 + w_1},\ w_{pc} = \frac{w_1}{u_1 + v_1 + w_1} \tag{3.4}$$

Again, as the W values are constant for the triangle, $\frac{1}{W_1}$, $\frac{1}{W_2}$ and $\frac{1}{W_3}$ are precalculated and multiplication instead of division is used. However, the division with the full area to get areal coordinates can not be prevented as the sum of the homogeneous barycentric coordinates is not constant over the whole triangle.

3.2.5 Pixel Shader unit

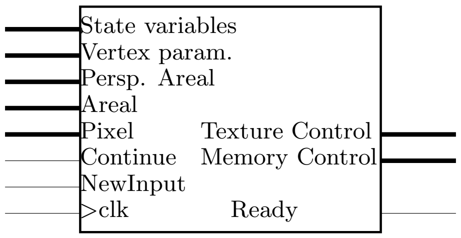

Figure 3.8: Pixel Shader unit.

Once a pixel has been determined to be inside the triangle and the perspective and non-perspective areal coordinates have been calculated, it is time to do the actual drawing of the triangle.

Certain properties are attached to each of the triangles vertices; Color, depth and texture coordinates. The interpolated property is attained with equation (2.4).

First of all, the depth value is calculated:

$$Z_{pixel} = u * Z_3 + v * Z_1 + w * Z_2. \tag{3.5}$$

This value is compared to the value in the Z-buffer, and the pixel is then either rejected or accepted as visible depending on the current z-test setting. If the pixel fails the depth test, no further calculations are done, and the unit is ready for the next pixel. If the pixel passes the depth test, the depth value is written to the z-buffer and the other vertex properties are calculated.

The z-test can be configured as one of the following settings, accept when *greater*, *less*, *greater or equal*, *less or equal*, *always* or *never*.

For texture coordinates and colors, since these properties have not been projected, the perspective corrected areal coordinates must be used for interpolation to keep the illusion of perspective.

Calculating the color, for example, is executed as:

$$Color_{pixel} = u_{pc} * Color_3 + v_{pc} * Color_1 + w_{pc} * Color_2. \tag{3.6}$$

If texturing has been enabled, the final pixel color is determined by either *modulating*

$$final = color * texture, \tag{3.7}$$

or *replacing*

$$final = texture, \tag{3.8}$$

the interpolated vertex color. If there is no texture, the final color is simply the interpolated vertex color.

Once the color has been written to the framebuffer, the unit is ready to take care of the next pixel.

3.2.6 Texture unit

The texture unit takes a texture coordinate pair as input parameters and outputs a color. There can only be one active texture at a time.

3.2.7 Memory controller

All memory operations done by the other components are done through the *Memory Controller*. This unit makes sure that only one component at a time accesses the memory and does so with a similar handshaking process, to each of the units accessing it, as the one described in section 3.2.2.

3.3 Programming the chip

3.3.1 Communication unit

Between the programmer and the core components there is a unit taking care of the data the programmer sends over the data bus. This unit takes care of some simpler triangle setup tasks and makes sure that the correct data is sent to the correct units.

3.3.2 Data formats

This section describes the formats used in the different parts of the implemented design. All numbers are in fixed point format and unless other specified, the notation is s for sign bits, i for integer bits and f for fractional bits.

texture mode	*disable, modulate, replace*
texture width	*iii*
texture height	*iii*
texture address offset	*iiii iiiiiiii iiiiiiii*
texture filtering	*none, bilinear*
depth test	*greater, less, greater or equal, less or equal, always, never*
shading	*flat, gouraud*

Table 3.1: Rendering states.

Rendering states

Different rendering states can be set when rendering the triangles.

The texture width and texture height value indicates how many extra bits to use when calculating the texture addresses. Setting these values to zero configures the texture unit to use the minimum sized texture of size $2^{3+0} \times 2^{3+0}$, while a value of 7 chooses the maximum texture size $2^{3+7} \times 2^{3+7}$.

Color

All colors are handled as 32-bit ARGB values, where ($0.FF000000_{16}$) is fully visible black and ($0.FFFFFFFF_{16}$) is fully visible white.

argb	a: *ffffffff*	r: *ffffffff*	g: *ffffffff*	b: *ffffffff*

Table 3.2: Bit configuration used by colors.

Vertex coordinates

The maximum screen resolution we have planned for is 1024×768, which means that the integer parts for the x and y coordinates fits in 10 bits, as $2^{10} = 1024$. For the fractional part, 4 bits have been chosen, which gives us 16×16 subpixel locations. When moving the triangle, this subpixel accuracy makes the pixels slowly move across the screen with less error and should be good enough for future antialiasing schemes.

The x,y and z components are the coordinates after homogenization, division by w, has been performed. Also the z coordinate has been normalized to the interval [0..1] where 0 is the most distant and 1 is the nearest value.

Texture coordinates

The u and v components represent the coordinate in the texture in the x and y directions. A value of (0.0,0.0) indicates the top left corner, while

x	*iiiiii iiiiffff*
y	*iiiiii iiiiffff*
z	*ffffffff ffffffff ffffffff*
$\frac{1}{w}$	*ffffffff ffffffff ffffffff*

Table 3.3: Bit configuration used by vertices.

($0.\text{FFFF}_{16}$,$0.\text{FFFF}_{16}$) indicates the lower right. The integer part is used for different wrapping modes.

u	*iiiiiiii ffffffff ffffffff*
v	*iiiiiiii ffffffff ffffffff*

Table 3.4: Bit configuration used by texture coordinates.

For 256×256 sized textures, 8 bits are used to select the correct texel, and then we still have 8 bits left for values between two texels. This means that when using bilinear interpolation, each texel in a 256×256 sized texture needs to cover more than 255 pixels before the image becomes pixelated once again:

$$\underbrace{iiiiiiii}_{\text{Wrapping}} \quad \underbrace{ffffffff}_{\text{Texel}} \quad \underbrace{ffffffff}_{\text{Sub-texel.}}$$

Using 1024×1024 sized textures, the bit usage changes to:

$$\underbrace{iiiiiiii}_{\text{Wrapping}} \quad \underbrace{ffffffff\ ff}_{\text{Texel}} \quad \underbrace{ffffff}_{\text{Sub-texel.}}$$

Triangle area

The reciprocal of twice the triangle area, as explained in section 3.2.4 on page 12, is needed to calculate the areal coordinates. The largest triangle area possible is

$$\triangle A = \frac{1023.9375 * 1023.9375}{2} \tag{3.9}$$

because of the maximum vertex coordinate values.

The later multiplication of two in the constant $\frac{1}{2\triangle A}$ suggests that the division in formula (3.9) can be skipped, and thus 28 bits are used to store it as that is the concatenation of the bits of two vertex coordinates.

$\frac{1}{2\triangle A}$	*ffff ffffffff ffffffff ffffffff*

Table 3.5: Bit configuration used by $\frac{1}{2\triangle A}$.

If the area times two is smaller than one, the reciprocal would need bits for integer representation. However, as a triangle of that size only would cover one pixel on the screen, the exact choice of area does not matter much.

One case where the area of a small triangle does matter is the case with a long but narrow triangle. Here many pixels can be covered by it, and thus it is important to get an accurate rasterization. This has not been experimented with for this report.

3.3.3 Internal data formats

Edge functions

It is important that the edge functions have a high precision as they are used to check whether or not a pixel is inside the triangle. With a bad precision, pixels shared between two triangle edges might be drawn twice, or not drawn at all and thus introducing cracks.

Since the a and b components are computed using the vertex coordinates, they have the same bit configuration as the x and y components of these. There is only one exception, a sign bit is also needed. This is so since the normal can point to the negative x and y axis.

As c is computed as in formulae (2.2), its bit configuration is the concatenation of the bits for the edges a and vertex x or, if you prefer, the edges b and vertex y.

a	*siiiiii iiiiffff*
b	*siiiiii iiiiffff*
c	*siiii iiiiiiii iiiiiiii ffffffff*

Table 3.6: Bit configuration for edge functions.

One could believe that the addition used when calculating c would introduce one extra bit for the carry, but this is not true. To get the maximum value of $|c|$, a and b needs to have their maximum values. In this implementation, this would be when using the two points p_0=(0,1023.9375) and p_1=(1023.9375,0), but for readability purposes the fractional part has been discarded in the following text as it does not change any of the proofs.

Now, using formulate (2.2), we get:

$$\begin{array}{l} a = -(p_1.y - p_0.y) = -(0 - 1023) = 1023 \\ b = p_1.x - p_0.x = 1023 - 0 = 1023. \end{array} \tag{3.10}$$

To calculate c, any point on the line can be chosen, and as it does not matter which point we choose, p_0 is used:

$$c = -(a * p_0.x + b * p_0.y) = -(1023 * 0 + 1023 * 1023) = -1023 * 1023. \tag{3.11}$$

From the above equation, it can be read that $|c|$ cannot have a value larger than $(2^{10} - 1)^2$.

$$\forall E(x,y)\neg\exists x\neg\exists y((E(x,y)=0)\wedge(|ax+by|>(2^{10}-1)^2)) \tag{3.12}$$

For the result of the edge function, a carry bit still does not have to be introduced. Using the above values for a, b and c, and the largest possible coordinate $p=(1023,1023)$, the result would be:

$$\begin{aligned}&a*p.x+b*p.y+c=\\&=1023*1023+1023*1023-1023*1023=\\&=1023*1023=\\&=(2^{10}-1)^2.\end{aligned} \tag{3.13}$$

Now that it has been proven that this is true for the maximum possible values of a, b and c, together with the maximum possible coordinate, any change in either the edge function or the testing coordinate would just make the resulting value smaller, and hence:

$$\forall E(x,y)\neg\exists x\neg\exists y(|E(x,y)|>(2^{10}-1)^2). \tag{3.14}$$

Areal coordinates

Multiplication between an edge function and the triangle area results in a 20.36 fixed point number. The fractional part is the only interesting part, so precise areal coordinates are 36 bits wide.

For perspective correct areal coordinates the number of bits have been specified as 16. When performing fixed point division, to get a quotient with $n.x$ bits, the dividend must have x more fractional bits than the divisor [3]. As the maximum number of bits of the divisor and dividend in our implementation is 32 [14], there is a limitation to the possible accuracy of the final coordinates.

As the number of bits for the quotient increases, the number of bits in the divisor has to decrease and the other way around, both which affects the accuracy of the final result. To get as good accuracy as possible, the 32bits have been split equally between the divisor and the quotient, and hence 16 bits are used for perspective correct areal coordinates.

To make the implementation a little bit easier the regular areal coordinates use 16 bits too.

u	*ffffffff ffffffff*
v	*ffffffff ffffffff*
w	*ffffffff ffffffff*

Table 3.7: Bit configuration for areal coordinates.

If u is $(0.FFFF_{16})$ and there is a value x=255 at its respective corner, then let r be $u\times x$ and the result of $\frac{r}{65535}$ should be 255. This division is not wanted and one very inexpensive approach would be to use bit shifting to avoid it. The result could then be calculated as $\frac{r}{65536} = r \gg 16$, but this would give the value 254 instead of 255.

Blinn [4] shows a way to avoid this division without loosing too much precision. Here the final result is calculated as $(r+(r \gg 8)) \gg 16$, where r's high bits are added to its low bits, and thus the algorithm is a little bit more accurate.

A similar method is used to get the perspective areal coordinates in the range $(0 \rightarrow FFFF_{16})$. If in equation (3.3), $u_1 = 1, v_1 = 0$ and $w_1 = 0$, then u_2 should be $FFFF_{16}$ while v_{pc}, w_{pc} should be zero. However, u_{pc} will be 10000_{16} as $\frac{u_{pc}}{u_{pc}} = 1$. Solving this is done with $u_{pc} = ((u_{pc} \ll 16) - u_{pc}) \gg 16$, where u_{pc}'s higher bits are subtracted from its lower ones. The same bit shifting operation is also performed on v_{pc} and w_{pc}.

Chapter 4

Results

The result is a tile based triangle rasterizer with support for z-buffering, perspective correct texturing and is fully synthesizeable on a Virtex-II FPGA at about 100 MHz.

Bilinear interpolation

When zooming in on a texture, the individual pixels in that texture covers a larger area on the screen, and thus making it clear that it is a sampled image.

To prevent this problem, bilinear interpolation is used to calculate a value between four of the textures pixels, and thus produce a smooth transition between them.

Using the sub-texel parts, U_f and V_f, of the texture coordinates, four weight values are calculated:

$$\begin{aligned} ul &= (1 - U_f) * (1 - V_f), \\ ur &= (U_f) * (1 - V_f), \\ ll &= (1 - U_f) * (V_f), \\ lr &= (U_f) * (V_f), \end{aligned} \tag{4.1}$$

where ul, ur, ll and lr are acronyms for upper left, upper right, lower left and lower right respectively. These four values are then multiplied with the four fetched texels to produce the final color:

$$Color_{final} = ul * Color_{ul} + ur * Color_{ur} + ll * Color_{ll} + lr * Color_{lr}. \tag{4.2}$$

Figure 4.1 gives an example.

Perspective correct texturing

The illusion of depth can be maintained with perspective correct texturing as depicted in Figure 4.2.

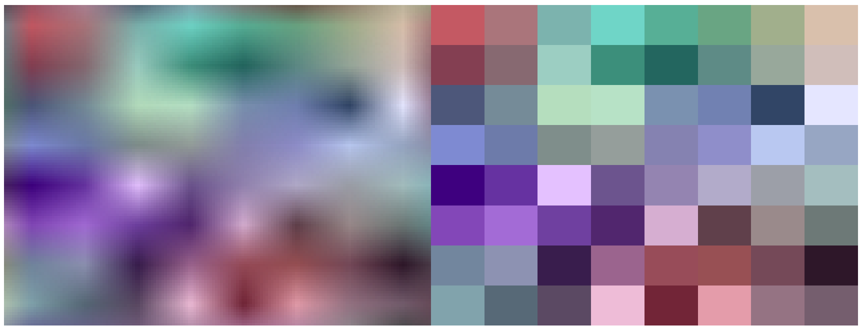

Figure 4.1: Bilinear interpolation vs no interpolation.

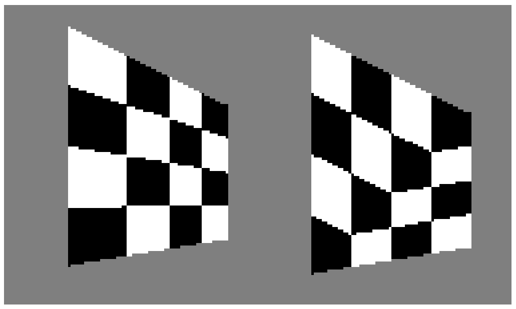

Figure 4.2: Perspective correct vs non-perspective correct texturing. To the right all corners in the triangle contribute equally to the texture coordinates, while on the left consideration has been taken to the corners depth.

Gouraud Shaded triangles

Gouraud shading smoothly interpolates the vertex colors across the triangle whereas *flat shading* has the same color over the whole triangle.

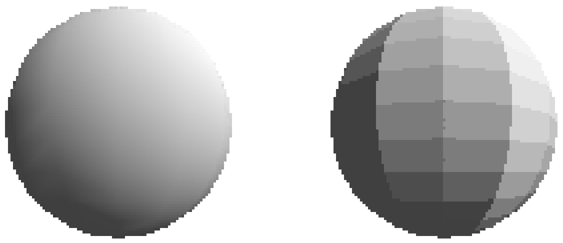

Figure 4.3: Gouraud vs flat shading.

Z-buffering

Triangles are drawn one at a time and sometimes two triangles intersect each other, meaning that parts of the triangle needs to be both in front of and behind the other. Z-buffering solves this problem as displayed in Figure 4.4.

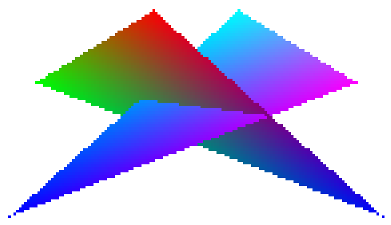

Figure 4.4: Two intersecting triangles.

Summary

The implementation can draw perspective correct and bilinear filtered textures, with z-buffering and gouraud shading. A couple of example models are shown in Figure 4.5.

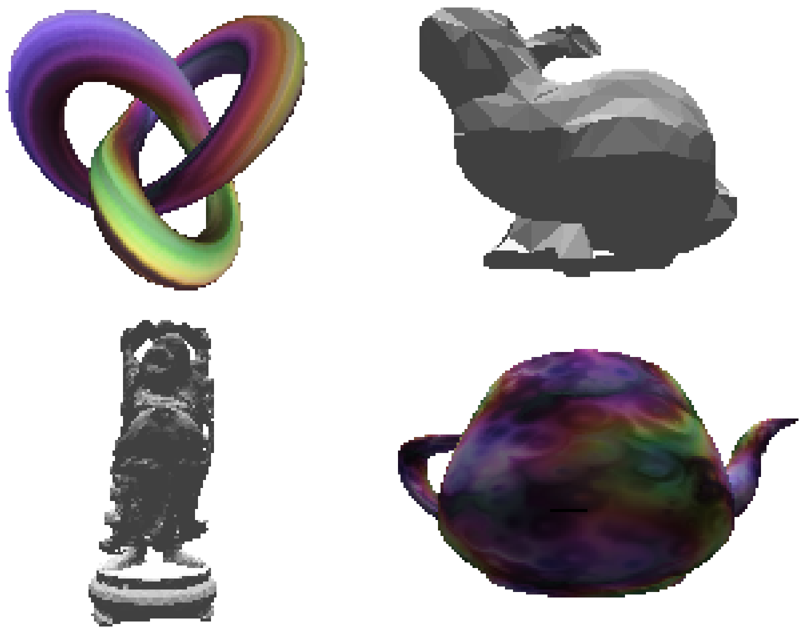

Figure 4.5: A plate displaying a couple of different models rasterized using the described implementation.

Chapter 5

Future improvements

5.1 General improvements

Bit depths

All buffers are hardcoded for 32bits access. Meaning that even if the graphics mode would be 16bit per pixel, a whole 32bit area is used for that pixel.

Dynamic rendering resolution

The rendering resolution is at the moment hardwired, but it should be programable.

Memory controller

It has been shown that for a resolution of 1024×768 in 32bit, in average the memory bandwidth in a modern graphical architecture is 2.4GB per second [8]. Thus the memory controller is a critical component when it comes to getting as much speed out of the architecture as possible and much care should be taken to make it perform as optimal as possible.

Also, this memory controller only interfaces simple one cycle read/write memories and this memory is a very limited resource on most FPGA boards. A better solution would be to use the DDR memory modules available.

Pipelines

As long as the memory controller isn't used to its full potential, several more pixel finder and pixel shader units can be set to work in parallel.

Also, the pipelining within the different components themselves is pretty much none at all and here huge improvements can be made for increased pixel throughput.

5.2 New features

Looking at the graphics cards of today they are a lot more advanced than the solution implemented here. Listing everything that is missing in our implementation would fill a book on its own, so in this section, the most important features that could be implemented in a near future are enumerated.

Alpha blending

To be able to render transparent triangles, alpha blending is needed.

Antialiasing

As can be seen in Figure 4.4, the edges of the triangles have a noticeable sawtooth look. To solve this problem, antialiasing can be implemented, where how much of a pixels area is covered by the triangle is taken into consideration when coloring it and thus produces smooth edges.

Geometry step

Most modern GPUs perform the geometry step in hardware.

Primitives

Only triangle primitives are supported. To be more feature complete at least triangle fans and triangle strips should also work.

Stencil buffer

With a stencil buffer you can implement effects such as masking and shadow volumes.

Bibliography

[1] Akenine-Möller, Tomas, and Aila, Timo, "Conservative and Tiled Rasterization Using a Modified Triangle Setup", To appear in Journal of graphics tools

[2] Akenine-Möller, Tomas, and Ström, Jacob, 2003, "Graphics for the masses: a hardware rasterization architecture for mobile phones", Proceedings of ACM SIGGRAPH 2003

[3] Astle, David and Durnhil, David, 2004, *OpenGL ES Game Development*, Premier Press, ISBN: 1-59200-370-2

[4] Blinn, Jim, 1998, *Jim Blinn's Corner: Dirty Pixels*, Morgan Kaufmann Publishers, Inc. ISBN: 1-55860-455-3

[5] Hakura, Ziyad S., and Gupta, Anoop, 1997, "The Design and Analysis of a Cache Architecture for Texture Mapping", International Conference on Computer Architecture

[6] McCool, Michael D., Wales, Chris, and Moule, Kevin, 2001. "Incremental and Hierarchical Hilbert Order Edge Equation Polygon Rasterization". Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware

[7] McCormack, Joel, and McNamara, Robert, 2000. "Tiled Polygon Traversal Using Half-Plane Edge Functions". In Workshop on Graphics Hardware, ACM SIGGRAPH/ Eurographics.

[8] NVIDIA Corporation, 2001, Lightspeed Memory Architecture, http://developer.nvidia.com/object/feature_lightspeed.html

[9] Pineda, Juan, 1988, "A parallel algorithm for polygon rasterization", ACM SIGGRAPH Computer Graphics Volume 22 , Issue 4 (August 1988)

[10] Weisstein, Eric W. "Areal Coordinates." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/ArealCoordinates.html

[11] Weisstein, Eric W. "Barycentric Coordinates." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/BarycentricCoordinates.html

[12] Weisstein, Eric W. “Homogeneous Barycentric Coordinates” From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/HomogeneousBarycentricCoordinates.html

[13] Xilinx Logicore, 2005, Multiplier Generator v7.0, Product specification, http://www.xilinx.com/ipcenter/catalog/logicore/docs/mult_gen.pdf

[14] Xilinx Logicore, 2005, Pipelined divider v3.0, Product specification, http://www.xilinx.com/bvdocs/ipcenter/data_sheet/sdivider.pdf