Some notes on

Graphics Hardware

Originally a draft for EDA075 Mobile Computer Graphics,
and later used for EDAN35 High-Performance Computer Graphics

Tomas Akenine-Möller
Lund University
Sweden
tam@cs.lth.se

© *Draft date November 27, 2012*

Contents

Contents i

Preface v

1 Introduction 1
- 1.1 Notation and operations . . . 1
  - 1.1.1 Homogeneous Coordinates and Projections . . . 2
- 1.2 Pipeline Overview . . . 5
  - 1.2.1 Geometry: Vertex Processing . . . 5
  - 1.2.2 Pixel Processing . . . 5

2 Edge Functions and Triangle Traversal 7
- 2.1 Edge Functions . . . 8
  - 2.1.1 Fixed-Point Edge Functions . . . 12
- 2.2 Triangle Traversal . . . 15
  - 2.2.1 Bounding Box Traversal . . . 15
  - 2.2.2 Backtrack Traversal . . . 16
  - 2.2.3 Zigzag Traversal . . . 17
  - 2.2.4 Tiled Traversal . . . 17

3 Interpolation 21
- 3.1 Barycentric Coordinates . . . 21
- 3.2 Perspectively-Correct Interpolation . . . 23
- 3.3 Interpolation using Edge Functions . . . 26
  - 3.3.1 Barycentric Coordinates . . . 26
  - 3.3.2 Perspective-Correct Interpolation Coordinates . . . 27
- 3.4 Triangle Setup and Per-Pixel Computation . . . 29

4 Pixel-Processing 31
- 4.1 The Pixel-Processing Pipeline . . . 31
- 4.2 The Pixel-Processing Equation . . . 31

5 Texturing **33**
5.1 Texture Images 34
5.1.1 Wrapping 35
5.2 Texture Filtering 37
5.3 Magnification 38
5.3.1 Nearest Neighbor Sampling 38
5.3.2 Bilinear Filtering 39
5.4 Minification 41
5.4.1 Mipmapping 42
5.4.2 Anisotropic Mipmapping 49
5.5 General Caching 50
5.5.1 Cache Mapping, Localization, and Replacement 52
5.5.2 Replacement Strategy 54
5.5.3 Cache Misses 54
5.6 Texture Caching 55
5.7 Texture Compression 55
5.7.1 Background 55
5.7.2 S3TC/DXTC 55
5.7.3 PVR-TC 55
5.7.4 ETC1/ETC2 55
5.7.5 ASTC 55
5.7.6 Normal Map Compression 55

6 Culling Algorithms **57**
6.1 Z-min and Z-max Culling 57
6.1.1 Z-max Culling 58
6.1.2 Z-min Culling 61
6.2 Object Culling with Occlusion Queries 63
6.3 Delay Streams 64

7 Buffer Compression **65**
7.1 Compression System with Cache 66
7.2 Depth Buffer Compression 68
7.2.1 Depth Offset Compression 69
7.2.2 Layered Plane Equation Compression 73
7.2.3 DPCM Compression 74
7.3 Color Buffer Compression 78

8 Screen-space Antialiasing **79**
8.1 Theory 79
8.2 Inexpensive Antialiasing Schemes 79
8.3 High-Quality Antialiasing 79

9 Architectures for Mobile Devices **81**
9.1 Tiling Architectures 81
9.2 Bitboys?? Others? 81

A Fixed-point Mathematics **83**
A.1 Notation . . . 83
A.1.1 Conversion . . . 84
A.2 Operations . . . 85
A.2.1 Addition/Subtraction . . . 85
A.2.2 Multiplication . . . 86
A.2.3 Reciprocal . . . 87
A.2.4 Division . . . 88
A.2.5 Inexpensive division: special cases . . . 88
A.2.6 Expansion of integers and fixed-point numbers . . . 89

Bibliography **91**

Preface

These notes were written for the course *Mobile Computer Graphics* given at Lund University during 2005–2008, and after that it has been and is still used in the *High-Performance Computer Graphics* course given by Michael Doggett. The notes are mainly intended for those who take that class. Other persons are welcome to use these notes for teaching as well, but please email the author first at: `tam@cs.lth.se`. The reason for putting these notes together was simply that I could not find this type of explanatory text somewhere. The first draft was written in the summer of 2005, and it was updated during falls of 2006 and 2007, and bug fixing will continue.

Note that I would be glad to hear about corrections, updates, and suggestions on how to make the text better.

Thanks to Michael Doggett, Fredrik Enhbom, Jon Hasselgren, John Owens, and Jonas Åström for feedback, corrections, and bug fixes.

Tomas Akenine-Möller

Smaller bug fixes: 2012
Third draft: September 2007
Second draft: September 2006
First draft: September 2005
Lund, Sweden

Chapter 1

Introduction

Hardware for rendering has changed the way real-time graphics is done in a major way. Graphics hardware makes the generation of images from three-dimensional scenes orders of magnitudes faster than an ordinary CPU can produce the same images. The major reasons for this are that pipelining and parallelism can be exploited to a much higher degree, and latency can be hidden in the architecture. This latency can be tolerated since rendering a frame takes, say, 20 ms, and as long as images come through the system at that rate, users tend to be happy. However, if it would take an instruction 20 ms to finish in a CPU, performance would be terrible.

Graphics hardware, which also was rather capable, started to appear in mobile devices, such as mobile phones and mobile gaming units (e.g., the SONY PlayStation Portable) around 2004–2004. The application programming interface (API) OpenGL ES 2.0 was released in the summer of 2005, and the major target for this API is mobile devices. OpenGL ES 2.0 already has support for programmable vertex and pixel shaders, and so this API has been catching up extremely fast with OpenGL in general. Graphics hardware for mobile devices has changed the way graphics is done forever. Originally, these notes were just targeting mobile graphics, but in hindsight, the algorithms described here can also work in high-end graphics architectures.

This chapter will mainly contain the notation that will be used throughout this text, and as well as some prerequisites.

1.1 Notation and operations

We denote vectors as column vectors, that is, a three-dimensional vector, **p**, is:

$$\mathbf{p} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix}. \tag{1.1}$$

Usually we make no distinction between a vector and a point, unless mentioned explicitly. Note also that when a vector is written on a row, it need to be transposed: $\mathbf{p} = (p_x, p_y, p_z)^T$. However, sometimes our text is a bit sloppy with that notation. For example, the coordinates of a pixel are often written as (x, y), even though the correct notation is $(x, y)^T$. Matrices are denoted by bold, upper-case letters, for example, $\mathbf{M}$.

The dot product and the cross product are two operators that are commonly used in graphics. The dot product between two vectors, $\mathbf{a}$ and $\mathbf{b}$, is a scalar and is denoted $d = \mathbf{a} \cdot \mathbf{b}$. Recall that $d = ||\mathbf{a}||\,||\mathbf{b}|| \cos\alpha$, where $||\cdot||$ denotes the length of the vector, and α is the smallest angle between $\mathbf{a}$ and $\mathbf{b}$. If $\mathbf{a}$ and $\mathbf{b}$ are perpendicular, then $\mathbf{a} \cdot \mathbf{b} = 0$. Note that the dot product can also be computed as: $d = \mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z$. The cross product between two vectors, $\mathbf{a}$ and $\mathbf{b}$, is denoted $\mathbf{c} = \mathbf{a} \times \mathbf{b}$. The resulting vector, $\mathbf{c}$, is always perpendicular to both $\mathbf{a}$ and $\mathbf{b}$. However, when $\mathbf{a}$ is parallel to $\mathbf{b}$, the result becomes $(0, 0, 0)^T$.

Truncation of a number, n, is denoted $\lfloor n \rfloor$, and this function returns the "closest" integer, i, so that $i \leq n$. Put another way, truncation is done towards $-\infty$. Similarly, the ceiling of a number is denoted $\lceil n \rceil$, and this functions returns the closest integer, i, so that $i \geq n$, that is, truncation is done upwards towards ∞.

Shift operators are often used when doing binary math, and in this text right shift is denoted as $a >> s$ exactly as in C and C++, that is, all bits of a are shifted s steps to the right. In general, shifting one step to the right means a division by two. Left shift is denoted $a << s$, and here all bits in a are shifted s steps to the left. Shifting only one step to the left, means multiplying by two.

The fractional part of a number, n, is obtained by removing the integer part, and the definition we will use is shown below:

$$f = n - \lfloor n \rfloor = \mathrm{frac}(n), \tag{1.2}$$

where f is the fractional part of n. Note that $\mathrm{frac}(1.2) = 0.2$ and $\mathrm{frac}(-1.8) = 0.2$.

Sets of numbers are used for convenience in the notation. If an integer i can take on any of the numbers, 0, 1, or 2, then the shorthand notation for that is $i \in [0, 1, 2]$. For a real (e.g. floating point) number, f, we can used the following notation: $f \in [0, 1]$, which means that f can take on any real number between 0 and 1, including both 0 and 1. If we write $f \in [0, 1)$ then f cannot take on exactly 1, but otherwise it is exactly the same as $f \in [0, 1]$. Similarly, $f \in (0, 1]$ includes all numbers between 0 and 1 except for 0. The final construction is $f \in (0, 1)$, which again includes all numbers between 0 and 1, except for 0 and 1.

The decimal number system is used for the most part. Binary numbers are indicated by a subscript of $_b$ at the end of the number, for example, 01010010_b. Similarly, a subscript of $_x$ indicates a hexadecimal number, e.g., $\texttt{FFC1}_x$.

1.1.1 Homogeneous Coordinates and Projections

To be able to use matrices for all common transforms, such as rotations, shears, and scalings (and any combinations of those), plus translations and projections,

© 2005–2012 Tomas Akenine-Möller

we need to turn to homogeneous coordinates.

The problem with translations is illustrated in the equation below:

$$\begin{pmatrix} ? & ? & ? \\ ? & ? & ? \\ ? & ? & ? \end{pmatrix} \mathbf{v} = \mathbf{M}\mathbf{v} = \mathbf{v} + \mathbf{t}, \tag{1.3}$$

that is, we want to express the translation of the point $\mathbf{v}$ using a matrix. This is not possible using a 3×3 matrix. Instead, we turn to homogeneous coordinates. For a three-dimensional point, we simply augment the point with a fourth element, so that

$$\mathbf{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \\ 1 \end{pmatrix}. \tag{1.4}$$

The translation can then be expressed as the matrix, $\mathbf{M}$, below.

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \mathbf{v} = \begin{pmatrix} v_x + t_x \\ v_y + t_y \\ v_z + t_z \\ 1 \end{pmatrix} = \mathbf{v} + \mathbf{t} \tag{1.5}$$

The fourth element of a vector, $\mathbf{v}$ is in general called v_w, and can also be either zero or any number at all.

$$\mathbf{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \\ v_w \end{pmatrix}. \tag{1.6}$$

If $v_w = 0$ then we have a direction vector, and when $v_w = 1$, we interpret that as a point. The addition of a direction vector, $\mathbf{d}$, to a point, $\mathbf{p}$, automatically becomes a point as expected. This can be seen by examining what happens to the fourth element after the addition.

Homogeneous coordinates can also be used for projections. After we multiply a perspective projection matrix to a point, $\mathbf{e}$, the fourth component is in general not equal to one (nor zero). To obtain a point again, we say that we homogenize the result. This is done by dividing all elements in the vector by the fourth element, as shown below, where $\mathbf{M}$ is a projection matrix.

$$\mathbf{M}\mathbf{e} = \mathbf{h} = \begin{pmatrix} h_x \\ h_y \\ h_z \\ h_w \end{pmatrix} \Longrightarrow \begin{pmatrix} h_x/h_w \\ h_y/h_w \\ h_z/h_w \\ h_w/h_w \end{pmatrix} = \begin{pmatrix} h_x/h_w \\ h_y/h_w \\ h_z/h_w \\ 1 \end{pmatrix} = \mathbf{p} \tag{1.7}$$

The arrow, $\Longrightarrow$, is used to denote the homogenization process. As can be seen, we obtain a point again, since the fourth component of $\mathbf{p}$ is equal to one. Also, note that the homogenization involves a division, and that is usually not allowed.

© 2005–2012 Tomas Akenine-Möller

However, with homogeneous coordinates it is, and this fact makes it possible to project points as well using matrices.

When you have transformed a point into eye space, first using a model transform (also called object transform), and then a view transform (also called camera transform), a point $\mathbf{e}$ is obtained. The next step is to use a projection matrix, in order to handle perspective effects, for example. Assume the projection matrix is called $\mathbf{M}$, then the projection is done as shown below:

$$\mathbf{Me} = \mathbf{h}. \tag{1.8}$$

As can be seen, the result is denoted $\mathbf{h}$. When written out fully, i.e., without the vector and matrix shorthand, the contents of the projection matrix, $\mathbf{M}$, can be seen, and Equation 1.8 then becomes:

$$\mathbf{Me} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} e_x \\ e_y \\ e_z \\ 1 \end{pmatrix} = \begin{pmatrix} h_x \\ h_y \\ h_z \\ h_w \end{pmatrix} \tag{1.9}$$

The matrix $\mathbf{M}$ above is the perspective projection matrix according to OpenGL. The parameters are n and f for the near and far plane, and it must hold that $0 < n < f$. Note that in eye space, the viewer looks down the negative z-axis, but after the projection matrix, $\mathbf{M}$, has been applied, the view direction is along the positive z-axis ($\mathbf{M}$ includes a negative scaling along z). The other parameters specify the frustum size on the near plane by giving a bottom-left corner and a top-right corner. The parameters are b & l and t & r. Note that for a symmetric frustum, it holds that $r = -l$ and $t = -b$, which gives the simplified projection matrix below.

$$\mathbf{M} = \begin{pmatrix} \frac{2n}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2n}{t-b} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{1.10}$$

As can be seen in Equation 1.9 (or using the matrix from Equation 1.10), the fourth component, h_w, of the result is in general not equal to one. Therefore, homogenization must be done in order to make the result into a valid point again. This is shown below.

$$\mathbf{Me} = \begin{pmatrix} h_x \\ h_y \\ h_z \\ h_w \end{pmatrix} \Rightarrow \begin{pmatrix} h_x/h_w \\ h_y/h_w \\ h_z/h_w \\ h_w/h_w \end{pmatrix} = \begin{pmatrix} h_x/h_w \\ h_y/h_w \\ h_z/h_w \\ 1 \end{pmatrix} = \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \mathbf{p} \tag{1.11}$$

As can be seen above, the projected point is denoted $\mathbf{p}$ here.

© 2005–2012 Tomas Akenine-Möller

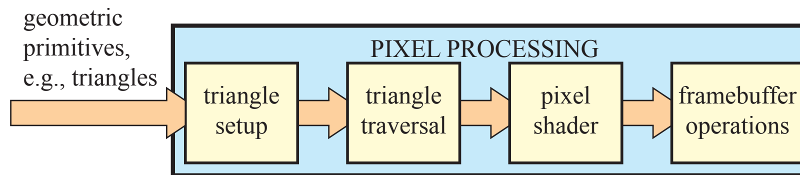

Figure 1.1: An overview of the pixel-processing stages.

1.2 Pipeline Overview

See first lecture: application + vertex processign + pixel processing.

1.2.1 Geometry: Vertex Processing

See first lecture.

1.2.2 Pixel Processing

A high-level overview of the pixel-processing stages is shown in Figure 1.1.

© 2005–2012 Tomas Akenine-Möller

© 2005–2012 Tomas Akenine-Möller

Chapter 2

Edge Functions and Triangle Traversal

The triangle is the geometrical *rendering atom* of real-time graphics. There are many reasons for this. As a counter example, consider a quadrilateral (polygon with four vertices). First of all, the four points need not be in the same plane, and this makes it very hard to render it in a simple way, especially since there are many different ways to define which points are on the surface of that primitive. Furthermore, if the quadrilateral is planar, then the vertices could form a concave polygon, which also makes hardware more complex. For example, a single scanline can intersect with more than two polygon edges. As another example, consider what would happen if you wanted to render a self-intersecting polygon (such as a star). Now suddenly, one need to be careful when defining the "inside" of the polygon.

The easiest way to provide a clean definition of a rendering primitive is to find the simplest geometrical entity that fulfils the requirements of representing a piece of a two-dimensional surface. The simplest such primitive is the triangle, since a triangle cannot loose another vertex and still be a two-dimensional primitive. If one vertex is removed, only two vertices remain, and these can only define a line, which is one-dimensional. Thus, it makes sense to build hardware for rendering triangles, and engineers and researchers have spent lots of time fine-tuning this process and optimizing the hardware.

A triangle is always convex and always reside in a single plane. Furthermore, interpolation of parameters (Chapter 3) is easy to define for a triangle, and fast to compute. In this chapter, we will study how a triangle can be *rasterized*, i.e., how all the pixels that are inside the triangle are found. These procedures are also called *triangle traversal* algorithms, since they describe how the pixels are traversed (visited). In Figure 2.1, the screen space coordinate system for rasterization is shown. Notice that the center of a pixel is given by $(x+0.5, y+0.5)$, where $x \in [0, w-1]$ and $y \in [0, h-1]$ are integers, and $w \times h$ is the screen resolution, e.g., 1920×1200 or 320×240 pixels. For now, we only consider the

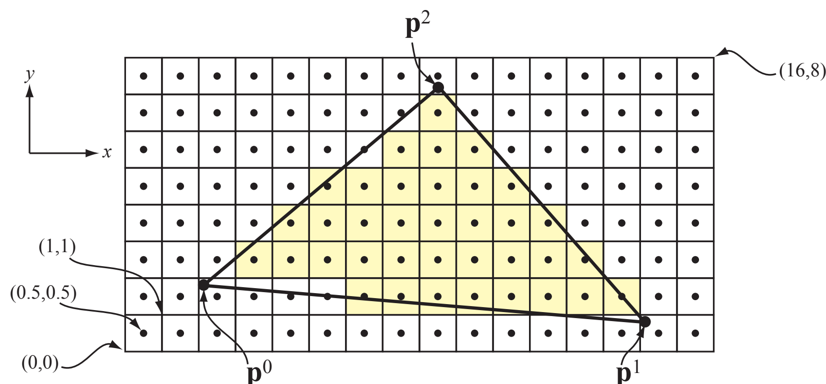

Figure 2.1: A triangle, $\Delta\mathbf{p}^0\mathbf{p}^1\mathbf{p}^2$, in screen space.

center of the pixel when testing whether a pixel is considered to belong to a triangle. In Chapter 8, supersampling algorithms will be studied that removes this constraint, and that increases the image quality.

This chapter starts with presenting the *edge function*, which is an excellent tool when one wants to rasterize triangles, and then discuss different triangle traversal algorithms, i.e., how the pixels inside a triangle can be found.

2.1 Edge Functions

In this section, we will describe edge functions [48], which are fundamental to rasterizing triangles in hardware.

Assume we have a two-dimensional triangle described by three points, $\mathbf{p}^0$, $\mathbf{p}^1$, and $\mathbf{p}^2$, where $\mathbf{p}^i = (p^i_x, p^i_y)$ (see Figure 2.1). Notice, that a z-coordinate could also be included in the points, $\mathbf{p}^i$, but those are not needed in this chapter, and so omitted. An *edge function* is the implicit equation, $ax + by + c = 0$, of the line through two of these points. For example, the edge function through $\mathbf{p}^0$ and $\mathbf{p}^1$ can be described as:

$$e(x, y) = -(p^1_y - p^0_y)(x - p^0_x) + (p^1_x - p^0_x)(y - p^0_y) = ax + by + c. \tag{2.1}$$

Equation 2.1 can easily be rewritten as $e(x, y) = ax + by + c = \mathbf{n} \cdot (x, y) + c$. In the last step, $\mathbf{n} = (a, b)$, and this can be interpreted as the normal of the line, i.e., it is a vector that is perpendicular to the line itself. For all points, (x, y), that are exactly *on* the line, it holds that $e(x, y) = 0$. Points that are on the same side as the normal gives $e(x, y) > 0$, and for points on the other side, $e(x, y) < 0$. It should be noted that the edge function could be defined so that the opposite holds as well. An example of an edge function is shown in Figure 2.2.

Next, we attempt to build some intuition on why the edge functions work like this. Note that in Equation 2.1, the normal is $\mathbf{n} = (-(p^1_y - p^0_y), p^1_x - p^0_x)$, i.e., a

© 2005–2012 Tomas Akenine-Möller

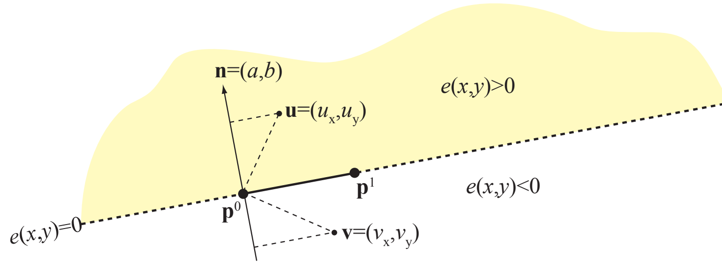

Figure 2.2: Illustration of an edge function, $e(x, y)$, defined by two points, $\mathbf{p}^0$ and $\mathbf{p}^1$. Imagine that you are located at $\mathbf{p}^0$ and look towards $\mathbf{p}^1$, then the positive halfspace of the edge is to the left, and the negative halfspace to the right. The latter space is not part of the triangle. The "normal" direction, $\mathbf{n} = (a, b)$, of the edge is also shown.

vector with the same length as $\mathbf{p}^1 - \mathbf{p}^0$, but perpendicular to it. Mathematically speaking, this means that $\mathbf{n}\cdot(\mathbf{p}^1 - \mathbf{p}^0) = 0$. Consider again Figure 2.2. When the point $\mathbf{u}$ is "plugged into" the edge function, we get $e(\mathbf{u}) = \mathbf{n}\cdot\mathbf{u} + c = \mathbf{n}\cdot(\mathbf{u} - \mathbf{p}^0)$, that is, the dot product between the vector from $\mathbf{p}^0$ to $\mathbf{u}$ and $\mathbf{n}$. This can be interpreted as the projection of $\mathbf{u} - \mathbf{p}^0$ onto the normal, and since $\mathbf{u}$ is on the same side of the edge as the normal, the projection must be positive. Similarly, the point $\mathbf{v}$ is also projected on $\mathbf{n}$ and $e(\mathbf{v})$ is negative since $\mathbf{v}$ is on the side of the edge where the normal is *not* located. For a point exactly on the edge, the projection becomes zero, and so $e(x, y) = 0$.

For each triangle, three edge functions can be created:

$$
\begin{aligned}
e_0(x, y) &= -(p_y^2 - p_y^1)(x - p_x^1) + (p_x^2 - p_x^1)(y - p_y^1) = a_0 x + b_0 y + c_0 \\
e_1(x, y) &= -(p_y^0 - p_y^2)(x - p_x^2) + (p_x^0 - p_x^2)(y - p_y^2) = a_1 x + b_1 y + c_1 \\
e_2(x, y) &= -(p_y^1 - p_y^0)(x - p_x^0) + (p_x^1 - p_x^0)(y - p_y^0) = a_2 x + b_2 y + c_2 \quad (2.2)
\end{aligned}
$$

Note that we have defined an edge function, $e_i(x, y)$, by using the two vertices, $\mathbf{p}_j$ and $\mathbf{p}_k$ such that $i \neq j$ and $i \neq k$. Put another way, an edge function, $e_i(x, y)$, is derived from the the vertices opposite of $\mathbf{p}_i$. As we will see, this is convenient when using the edge functions to compute barycentric coordinates (Section 3.3). In Figure 2.3, the three edge functions of a triangle are illustrated.

In general, we could say that a point, (x, y), is inside the triangle if $e_i(x, y) \geq 0$, for all $i \in [0, 1, 2]$. However, this may not always work as desired. Consider two triangles sharing an edge, and assume that we sample each pixel at the center of the pixel. For sample points that lie exactly on the shared edge, the pixel will belong to both triangles, and thus, those pixels will be visited twice. This gives incorrect results for shadow volume rendering [15] and transparency, for example, and it also makes for worse performance. Another approach would be to, instead, perform the test like this: $e_i(x, y) > 0$. However, a pixel lying exactly on the shared edge, will not be considered to belong to either triangle.

© 2005–2012 Tomas Akenine-Möller

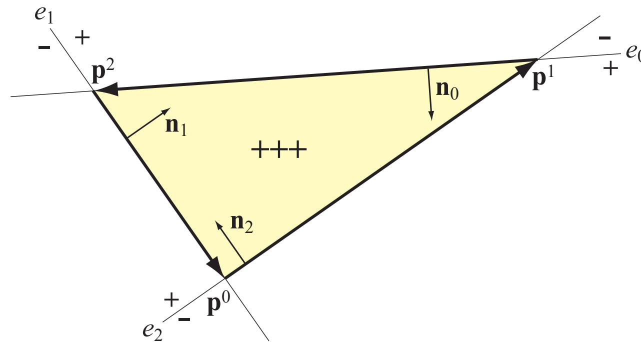

Figure 2.3: A triangle $\Delta\mathbf{p}^0\mathbf{p}^1\mathbf{p}^2$ with its three edge functions $e_0(x, y)$, $e_1(x, y)$, and $e_2(x, y)$. For points inside the triangle, all edge functions are positive. The edge functions normals are $\mathbf{n}_i = (a_i, b_i)$, where (a_i, b_i) comes from the respective edge function, $e_i(x, y) = a_i x + b_i y + c_i$. Note that the lengths of the normals are arbitrary scale in order to make the illustration readable.

bool INSIDE(e, x, y)
1 **if** $e(x, y) > 0$ **return** true;
2 **if** $e(x, y) < 0$ **return** false;
3 **if** $a > 0$ **return** true;
4 **if** $a < 0$ **return** false;
5 **if** $b > 0$ **return** true;
6 **return** false;

Figure 2.4: McCool et al's [38] tiebreaker rule for determining whether a point (x, y) is "inside" an edge, e.

Thus, a so called *pixel dropout* is generated, which simply is an undesired crack between connected triangles.

There is an easy work around [38] though, as shown in Figure 2.4. This is often called a tiebreaker rule. The idea is to only include points, (x, y), that are fully inside the triangle ($e > 0$), and exclude points that are fully outside ($e < 0$). If neither of these conditions are fulfilled then $e = 0$, and we choose to include points that are to the "left" of the triangle ($a > 0$), and we exclude points that are to the "right" of the triangle ($a < 0$). The remaining cases occur when $a = 0$, i.e., horizontal edges. There are two such cases; either a "top" edge or a "bottom" edge of a triangle. As long as we choose to only include one of these cases, either of them work. In Figure 2.4, the bottom horizontal edge is included, and the top horizontal edge is excluded. Owens [46] presents another way to think about this, as shown in Figure 2.5.

Since, a and b are constants for each edge, the code in Figure 2.4 can be

© 2005–2012 Tomas Akenine-Möller

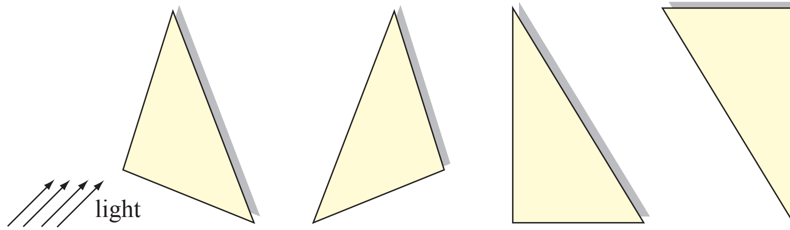

Figure 2.5: Shadowing rule for including/excluding point on edges. A directional light source emits light in the direction indicated by the arrows. If a point (x, y) is exactly on the edge, i.e., $e(x, y) = 0$, then the point is only included if the edge is not shadowed.

bool InsideOptimized(e, t, x, y)
1 **if** $e(x, y) > 0$ or $(e(x, y) == 0$ and $t)$ **return** true;
2 **return** false;

Figure 2.6: Optimized tie-breaking inside test using an edge function and a precomputed bit, t.

optimized by precomputing a boolean (i.e., a single bit), t for each edge as follows:

$$t = \begin{cases} \texttt{bool}(a > 0), & \text{if } a \neq 0 \\ \texttt{bool}(b > 0), & \text{else.} \end{cases} \tag{2.3}$$

where `bool`$(expr)$ returns 1 if the expression is true, and otherwise it returns 0. Given this bit, t, per edge, the inside-test simplifies to the code shown in Figure 2.6.

Assume for a while that we place a sample point at the center of each pixel. A nice property of the edge function is that it is inexpensive to determine whether a sample point is inside the triangle if you have just evaluated a neighboring sample point. In fact, this is one of the reasons why edge functions most often are used for traversing the pixels inside a triangle. Now, assume that we have just visited a pixel at (x, y), and would like to traverse to a neighboring pixel at $(x + 1, y)$. Let us see what happens to the edge function, e, from Equation 2.1:

$$e(x + 1, y) = a(x + 1) + by + c = e(x, y) + a. \tag{2.4}$$

As can be seen, it is just a matter of adding one of the constants, a, to $e(x, y)$ in order to evaluate e at $(x + 1, y)$. To traverse in the negative x-direction, simply subtract a. In the y-direction, the calculations are similar. The four possibilities

© 2005–2012 Tomas Akenine-Möller

are shown below.

$$\begin{aligned} e(x+1,y) &= e(x,y)+a \\ e(x-1,y) &= e(x,y)-a \\ e(x,y+1) &= e(x,y)+b \\ e(x,y-1) &= e(x,y)-b \end{aligned} \quad (2.5)$$

Thus, once the edge functions have been created and evaluated for the sample point in one pixel, any of the four neighboring pixels can be evaluated using only three (one per edge function) additions. All pixels can be reached by repeating this process, though care must be taken in order to remain sufficient accuracy (using, for example, enough bits in a fixed-point implementation).

More generally speaking, one can rewrite Equation 2.4 as shown below:

$$e(\mathbf{s}+\mathbf{t}) = e(\mathbf{s}) + \mathbf{n}\cdot\mathbf{t}, \quad (2.6)$$

where $\mathbf{s} = (s_x, s_y)$, $\mathbf{t} = (t_x, t_y)$, and $\mathbf{n} = (a, b)$.

2.1.1 Fixed-Point Edge Functions

Now, assume that all triangles have been clipped and transformed to screen-space, and the projected two-dimensional vertices of the triangle are called $\mathbf{p}^0$, $\mathbf{p}^1$, and $\mathbf{p}^2$. Furthermore, we assume that the maximum resolution of the screen is $w \times h$ pixels, and for simplicity, we set $w = h = 2^b$.

Two triangles sharing an edge must get exactly the same edge function for the shared edge. Otherwise, inconsistent behavior, e.g., pixel dropouts, can be expected. Therefore, an edge function derived from $\mathbf{p}^0$ and $\mathbf{p}^1$ must be identical to an edge function derived from $\mathbf{p}^1$ and $\mathbf{p}^0$ (only order differs). Thus, it is very important to maintain a sufficient number of bits for the constants, a, b, and c, because otherwise, pixel dropouts can occur. In this section, we look into the number of required bits for doing this correctly.

When the vertices have been transformed into screen-space, they are usually represented using floating-point representations. However, it is very convenient to use fixed-point mathematics (see Appendix A) to represent the numbers, a, b, and c, in the edge functions. The reason for this is that the dynamic behavior of floating point numbers is not needed for these computations due to the limited range of the input numbers, and fixed-point mathematics is less expensive to implement in hardware. Therefore, we must truncate or round off the floating-point coordinates. Simply truncating these to the positions at the center of the pixels will give a bad appearance. For example, a slowly translating triangle will appear to abruptly "jump" from one pixel to the next instead of showing a smooth movement. To get reasonable quality, it is preferable to use a grid of sub-pixel coordinates inside each pixel [36]. Assume we choose to use a sub-pixel resolution of $2^g \times 2^g$ grid locations inside each pixel.

This is illustrated in Figure 2.7. In general, the floating-point coordinates have simply been rounded off to g fractional bits (see Appendix A for rounding). In the figure, there are only two bits for the sub-pixel grid, i.e., $g = 2$.

© 2005–2012 Tomas Akenine-Möller

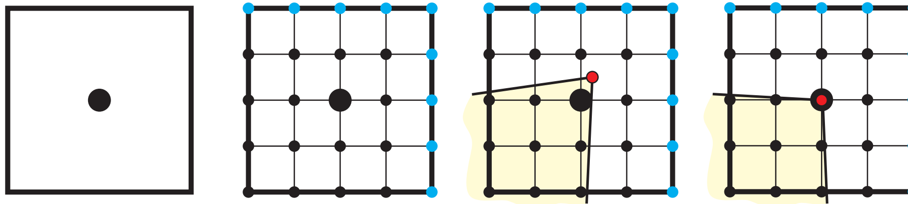

Figure 2.7: Left: a pixel with the sampling point at the center of the pixel. Middle-left: a 4×4 sub-pixel grid per pixel (blue points are shared by neighboring pixels). Middle-right: a triangle vertex with floating-point coordinates. Right: the floating-point vertex has been rounded off to the nearest sub-pixel coordinate (which happens to coincide with the center sampling point.

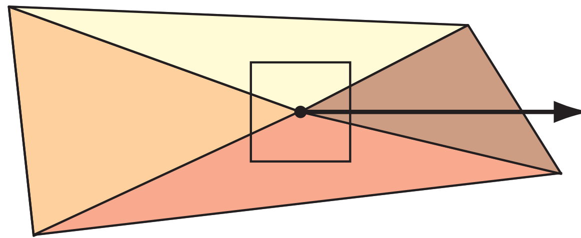

Figure 2.8: Four triangles are sharing a vertex, which coincides with the sampling point (center) of a pixel. The pixel should only belong to one of the triangles, and a robust triangle rasterizer handles this case correctly.

Unfortunately, as a floating-point vertex is transformed into a fixed-point vertex location, it may be located at the center of the pixel, and thus it coincides with the sampling point. Now, assume that the vertex is part of a mesh, and several triangles share that vertex. Since, the truncated vertex is located in the center, more than one triangle may visit this pixel during triangle traversal, and yet, the pixel should only belong to one of the triangles. Hence, the same type of problem arises here as when two triangles shared an edge. An example is shown in Figure 2.8.

One solution is to choose an *inclusion direction*. In Figure 2.8, this direction is a vector pointing to the right. It should be noted that any vector works, as long as the same vector is used for every triangle being rendered. Now, the trick is to let the pixel belong to the triangle which has the vector in its interior. In Figure 2.8, the dark triangle to the right includes the inclusion vector, and therefore, the pixel belongs to that triangle only. In practice, it is possible to determine this by looking at the edge functions for the two edges sharing the vertex. This is left as an exercise.

Another solution is to offset the sub-pixel grid by a half sub-pixel as shown in Figure 2.9. That is, the sub-pixel grid is placed so that no sub-pixel coordinate coincides at the sample point(s) of the pixel. The advantage of that is that the

© 2005–2012 Tomas Akenine-Möller

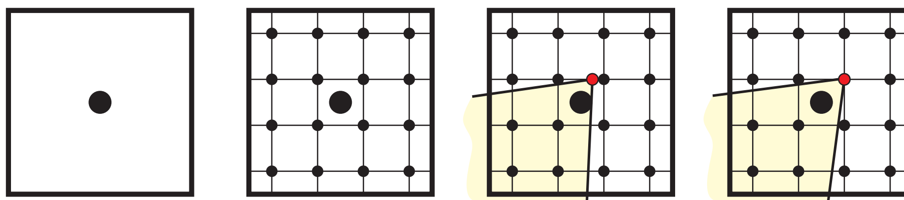

Figure 2.9: Left: a single pixel with the sample point in the center of the pixel. Middle-left: the pixel has been divided into a 4×4 sub-pixel grid offset half a sub-pixel. Middle-right: the same triangle vertex with floating-point coordinates as in Figure 2.7. Right: the triangle vertex has been "snapped" to the closest of the sub-pixel grid locations. This has two advantages: a pixel sample point will never coincide with a vertex, and a limited number of positions inside a pixel gives us the opportunity to use fixed-point mathematics for the edge functions.

edge function will not be evaluated at the vertices of the triangle, and thus, we can avoid pixel dropouts or double (unnecessary) pixel writes there. This means that after truncation to the sub-pixel grid, each two-dimensional vertex, $\mathbf{p} = (p_x, p_y)$, must be represented using two fixed-point numbers with $[b.g]$ bits per x and y. As can be seen in Equation 2.2, the normal of an edge function is typically computed as:

$$\mathbf{n} = (a, b) = (p_y^i - p_y^j, p_x^j - p_x^i), \tag{2.7}$$

which means that a and b will need $[b + 1.g]$ bits for exactly representing the difference using fixed-point numbers. The c-parameter of an edge function is computed as:

$$c = -ax - by, \tag{2.8}$$

for some point, (x, y), that lies exactly on the line. Both a and b are represented using $[b + 1.g]$ bits and both x and y are vertex coordinates, which have been truncated to $[b.g]$ bits. Due to Equation A.8, the terms ax and by will both need $[2b + 1.2g]$ bits, and thus $[2b + 2.2g]$ bits are needed for exactly representing the c-parameter. To evaluate an edge function, we get the following:

$$e(x, y) = \underbrace{\underbrace{\underbrace{\underbrace{ax}_{[b+1.g]\times[b.g]}}_{[2b+1.2g]} + \underbrace{\underbrace{by}_{[b+1.g]\times[b.g]}}_{[2b+1.2g]}}_{[2b+2.2g]} + \underbrace{c}_{[2b+2.2g]}}_{[2b+3.2g]} \tag{2.9}$$

However, if the sub-pixel grid is offset as shown in Figure 2.9, the sub-pixel grid does not coincide with the sample point, and one more bit of precision is needed to take that into account. A simple way to implement this to compute the edge function with the rounding shown in Figure 2.7, and is then simply

© 2005–2012 Tomas Akenine-Möller

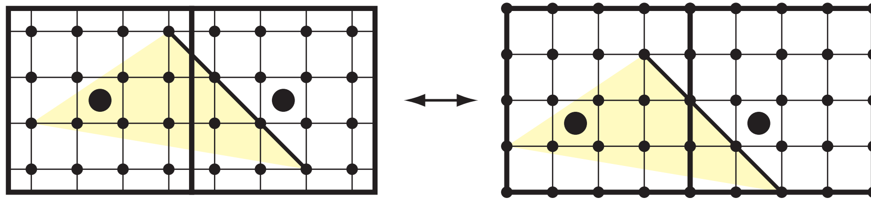

Figure 2.10: Left: two pixels with each a 4×4 sub-pixel grid, and a triangle with one edge (marked in black) that we want to evaluate at the center of the pixels. Right: one way of implementing that evaluation. Simply round off the vertices using the standard grid (Figure 2.7), and then offset the sampling point (big circle).

offset the sample point location half a sub-pixel. This is shown in Figure 2.10. So instead of using the point $(x + 0.5, y + 0.5)$, where x and y are integers, to evaluate the edge function, we must use $(x + 0.5 - 2^{-g-1}, y + 0.5 - 2^{-g-1})$. In Equation 2.9, this means that instead of using $[b.g]$ bits for the x and y, $[b.g + 1]$ should be used.

2.2 Triangle Traversal

"To traverse a triangle" is the procedure that finds the pixels that are inside the triangle. This process is often also referred to as *rasterization*. The positions of the pixels are sent down the pipeline for further processing, and eventually colors and depth values may be written to the color and depth buffer, respectively. In this section, we assume again that only a single sample is taken per pixel, and the sample is located in the center of the pixel. Several different strategies will be explored and explained here. All of them have one thing in common: they only test one sample (pixel) at a time. There exist more efficient algorithms (that visits fewer unnecessary pixels), but those execute the $\textsc{Inside}(x, y)$ for more than one pixel at a time and requires more storage.

2.2.1 Bounding Box Traversal

The simplest strategy is to compute a bounding box of the triangle with respect to the grid of center sample points, and then execute the $\textsc{Inside}(x, y)$ test for each edge for each center point of the pixels inside the bounding box. This determines which pixels are inside the triangle, and the procedure is illustrated in Figure 2.11. Once the bounding box has been found, the pixels can be visited in any order, e.g., left-to-right/bottom-to-top. Clearly, this is not a very efficient strategy since many more pixels are visited than are actually inside the triangle.

© 2005–2012 Tomas Akenine-Möller

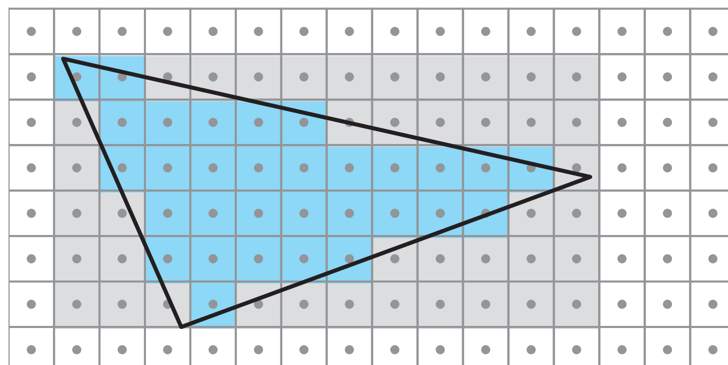

Figure 2.11: A triangle is traversed using the bounding box strategy. The pixels are shown as squares and the circles show where the sample points are located. The light gray pixels are included in the bounding box, but only the blue pixels are actually inside the triangle.

2.2.2 Backtrack Traversal

Another very simple traversal strategy is called backtrack traversal, and the basic idea is shown in Figure 2.12. This type of traversal has been used in graphics hardware for PDAs and other mobile devices [58].

Traversal starts at the pixel center below the topmost vertex, then the strategy is to process one scanline at a time, always from left to right. Therefore, on each scanline, one must make sure that we have traversed so that we start at a pixel that is to the "left" of the triangle. However, it should be noted that traversal never need to go outside the bounding box (as in Section 2.2.1) of the triangle. When a pixel that is outside to the left has been found, traversal can continue to the right until a pixel that is outside to the right of the triangle is found. At that point, traversal continues to the pixel below the current pixel. Backtracking to the left follows until a pixel that is outside to the left of the triangle is found, and so on.

In the text above, we have used a terminology of "outside to the left" and "outside to the right." Recall that the normal of a triangle points inwards the inside of the triangle, and that the normal is $\mathbf{n} = (n_x, n_y) = (a, b)$. By "outside to the left," we mean a point (or pixel) such that the point is outside at least one edge function with $a > 0$. Similarly, points "outside to the right" of the triangle are points that are outside at least one edge function with $a < 0$. An example of when this is clearly needed is shown to the right in Figure 2.12. On the third scanline from the top, we need to traverse down to the pixel below. As can be seen, that pixel is outside one edge function, and therefore the pixel is outside the triangle. However, to find the first pixel that is outside to the left of the triangle, this test is not sufficient. Instead, we need to traverse to the left until we find a pixel that is outside to the left at least one edge function.

© 2005–2012 Tomas Akenine-Möller

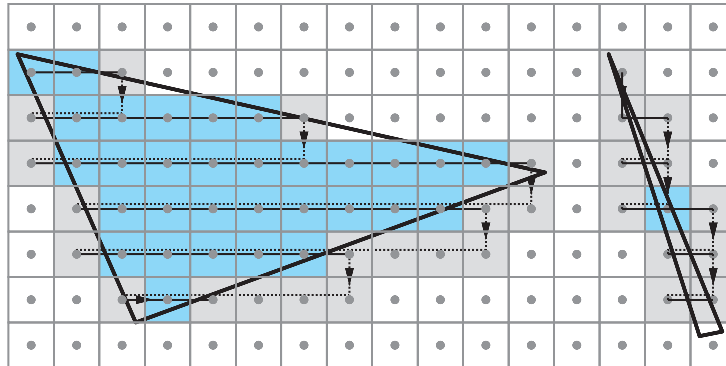

Figure 2.12: Backtrack traversal. Left: pixels are always visited one scanline at at time, and always from left to right. Therefore, when one scanline is finished, the next scanline starts its processing by backtracking until the current pixel is outside to the left of the triangle. Right: a thin triangle being traversed. Notice that only one pixel is included.

2.2.3 Zigzag Traversal

The backtrack traversal strategy visits pixel during its backtracking stage, but it is only doing this to find a pixel outside to the left. We do not compute depth, color, or anything else. A more efficient algorithm is called zigzag traversal as suggested by Pineda [48], and described in more detail by Akenine-Möller and Ström [7]. An example of this is shown in Figure 2.13. As can be seen, one scanline at a time is traversed, and the traversal order (left-to-right or right-to-left) is altered every scanline.

In that figure, some cases are illustrated that are not so efficient, but in general this algorithm visits fewer pixels than the backtracking traversal strategy. It should be noted that pixels are visited in backward order on every other scanline and this can make memory accesses less efficient.

Bounding box, backtrack and zigzag traversal can work by always traversing to a pixel that is either to the left, to the right, above or below the current pixel. Therefore, incremental updates of the edge functions (page 2.1) can be done which makes for less expensive implementation. Next, we will discuss more expensive and more efficient traversal strategies.

2.2.4 Tiled Traversal

In this section, tiled traversal strategies will be described. A tile is a rectangular region consisting of $w \times h$ pixels.Common choices of tile sizes are 4×4 or 8×8 pixels. Tiled traversal algorithms visit all pixels[1] inside a tile before proceeding to the next tile. An example is shown in Figure 2.14.

[1]Several tiles can also be processed in parallel, but the same principle applies.

© 2005–2012 Tomas Akenine-Möller

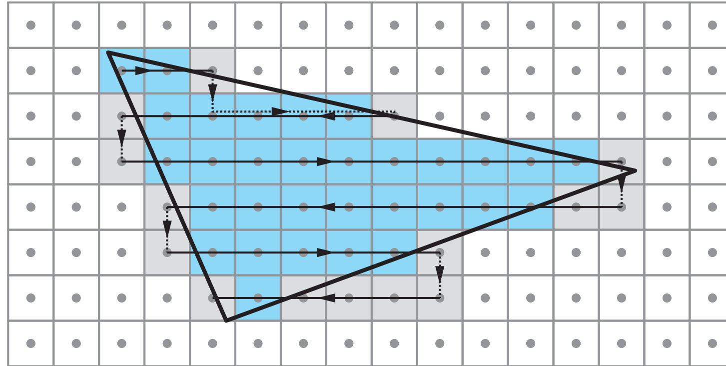

Figure 2.13: Zigzag traversal. Traversal order is changed on every scanline. Note that when we move down from the second to the third scanline, we need to traverse to the right until we find a pixel that is outside to the right of the triangle. Also, when moving down to the fifth scanline, two pixels that are outside are traversed before finding a pixel that is inside. On the seventh scanline, it is even worse; four pixels are visited that are outside.

There are several reasons why tiled traversal algorithms are to prefer over the traversal strategies that were described previously. Tiled traversal makes for better texture caching performance (Section 5.6) [24]. Furthermore, tiled traversal makes is possible to implement algorithms that reduces bandwidth usage. This includes, for example, depth- and color-buffer compression (Chapter 7), zmin- and zmax-culling (Section 6.1).

Finding tiles that overlap a triangle is very similar to finding pixels that are inside the triangle. However, an algorithm is needed for testing whether a tile overlaps with the triangle (compared to just testing whether a sample point is inside the triangle), and a traversal strategy for finding tiles that can potentially contain pixels inside the triangle. We start by describing a simple algorithm [5] for testing whether a tile overlaps a triangle, and then continue describing some simple traversal strategies.

Tile/Triangle Overlap Test

As it turns out, finding out whether a tile overlaps a triangle can be determined by using the edge functions again. A tile is excluded if either the tile is fully outside the bounding box of the triangle, or if the entire tile is outside at least one of the edge functions. The bounding box test is trivial, so we continue describing the second part. Testing whether a tile is outside an edge function can be implemented by evaluating the edge function for all four corners of the tile, and if all four corners are outside, then the tile is outside that edge. However, there is a smarter way [5] that will be described now.

Haines and Wallace [23] observed that when testing whether a three-dimensional box intersects with a plane, it suffices to test the two box corners that form a

© 2005–2012 Tomas Akenine-Möller

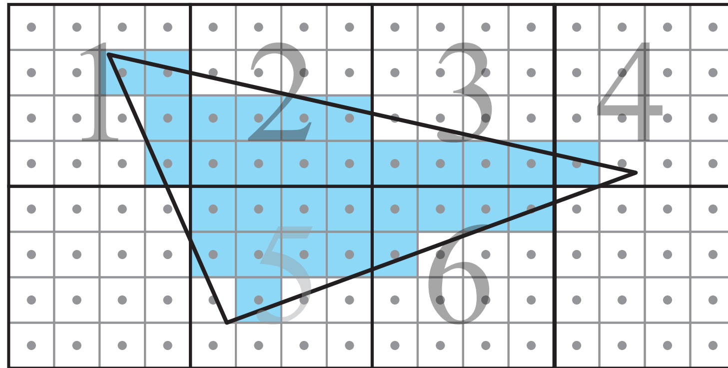

Figure 2.14: A possible traversal order when using tiled traversal. In this example, the tile size is 4×4 pixels, and the processing starts with the tile enumerated with one. The four pixels inside the triangle inside that tile are first processed, before moving on to tile two, and so on.

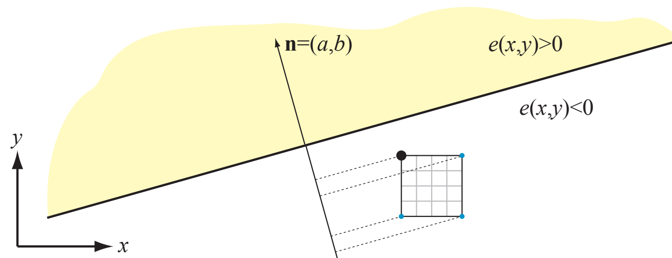

Figure 2.15: The corners of a 4×4 tile are projected onto the edge's normal. Only the corner with the black large circle needs to be tested against this edge, since its projection onto $\mathbf{n}$ is greatest.

diagonal that most closely aligns with the plane's normal. In addition, Hoff [32] suggests that only one of those corners need to be tested to determine if the box is in the negative halfspace (or in the positive halfspace). Since a plane is an edge function and a three-dimensional box is a tile in two-dimensional space, a similar test can be used here. Thus, we can reduce to testing a single corner of the tile in order to test whether the tile is outside an edge function. The corner that should be tested depends on the direction of the edge function.

Consider Figure 2.15 for a while. As can be seen, a tile's four corners have been projected onto the normal of an edge. The top left corner has the greatest projection, and this means that we only need to test whether that corner is outside the edge. If it is outside, the entire tile is guaranteed to be outside the edge. Otherwise, the tile is partially overlapping the edge or fully inside.

In order for this to be efficient, we cannot actually perform the projection. Instead, we use the normal, $\mathbf{n} = (n_x, n_y) = (a, b)$, of the edge to determine this.

© 2005–2012 Tomas Akenine-Möller

Looking again at Figure 2.15, we can conclude that one of the two topmost corners of the tile must be the correct corner since $n_y > 0$. Moreover, since $n_x < 0$, the correct corner must be the upper left corner of the tile.

Now, assume that the lower left corner of a tile is denoted $\mathbf{s} = (s_x, s_y)$, and that the edge evaluation there is $e(\mathbf{s})$. Any of the four corners can be evaluated using the edge function by adding a constant $\mathbf{t}$ to $\mathbf{s}$. The value of $\mathbf{t} = (t_x, t_y)$ should be either $(0, 0)$, $(w, 0)$, $(0, h)$, or (w, h), where $w \times h$ is the size of a tile [5]:

$$t_x = \begin{cases} w, & n_x \geq 0 \\ 0, & n_x < 0 \end{cases}, \quad t_y = \begin{cases} h, & n_y \geq 0 \\ 0, & n_y < 0 \end{cases}. \tag{2.10}$$

This type of setup is needed once per edge function and is therefore done in the triangle setup, before traversal starts. The value of $\mathbf{t}$ from Equation 2.10 is plugged into Equation 2.6. The involved computations are inexpensive: two comparisons (Equation 2.10), two shifts (when w and h are powers of two), two additions, and also the evaluation of the edge function at the bottom left corner, $e(\mathbf{s})$.

Tiled Traversal Strategies

At this point, we know how to determine whether a tile overlaps (partially or fully) a triangle. For such tiles, we need to traverse the pixels inside the tile, and we also need to traverse the tiles themselves in some order.

Both the bounding box strategy (Section 2.2.1) and the zigzag traversal technique (Section 2.2.3) are straightforward to adapt to traversing tiles instead of traversing pixels. However, again zigzag traversal is expected to perform much better.

Once a tile has been found that is partially or fully overlapping with a triangle, any technique can be used to traverse the pixels. Bounding box, back track traversal, or zigzag all work for this purpose.

A more sophisticated scheme is presented by McCormack and McNamara [40].

© 2005–2012 Tomas Akenine-Möller

Chapter 3

Interpolation

The purpose of this chapter is to explain interpolation of scalar values over a triangle, both in perspective and using a parallel (also called orthographic) projection. This is something that is fundamental to computer graphics, and in particular to rasterization algorithms.

3.1 Barycentric Coordinates

We start by studying how to interpolate over a two-dimensional triangle. Assume, we have a triangle, $\Delta \mathbf{p}^0\mathbf{p}^1\mathbf{p}^2$, with scalar values, (s^0, s^1, s^2), at the respective vertices. Given an arbitrary point, $\mathbf{p}$, inside a triangle, the goal of using barycentric coordinates is to interpolate the scalar s-values over the triangle in a continuous manner with respect to $\mathbf{p}$'s position. This situation is illustrated to the left in Figure 3.1.

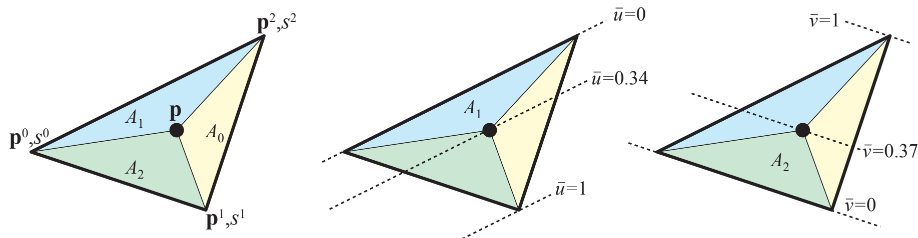

Figure 3.1: Left: a triangle with scalar values (s^0, s^1, s^2) at the vertices. The barycentric coordinates are proportional to the signed areas (A_1, A_2, A_0). Middle: illustration of the barycentric coordinate, $\bar{u}$. Right: illustration of the barycentric coordinate, $\bar{v}$. Note that on the dashed lines, the respective barycentric coordinate is constant.

Here, the barycentric coordinates for a triangle are denoted: $(\bar{u}, \bar{v}, \bar{w})$,[1] and those are proportional to the signed areas (A_1, A_2, A_0) of the sub-triangles formed by a triangle edge and $\mathbf{p}$. Note that we use the bar over these barycentric coordinates. This is in contrast to perspectively-correct interpolation coordinates, described in Section 3.2, which do not use the bars.

Computing the cross product of two edge vectors of a triangle gives the area of the parallelogram, and by dividing by two, the area of the triangle is obtained. For example, the signed area for A_1 is computed as:

$$A_1 = \frac{1}{2}((p_x - p_x^0)(p_y^2 - p_y^0) - (p_y - p_y^0)(p_x^2 - p_x^0)), \tag{3.1}$$

and similarly for the other areas. For computer graphics applications, we almost always use barycentric coordinates normalized with respect to the total triangle area, $A_\Delta = A_0 + A_1 + A_2$. In fact, this is a special case of barycentric coordinates called *areal coordinates* [55]. When we refer to barycentric coordinates in this text, however, we always mean these normalized barycentric coordinates. Hence, the barycentric coordinates are computed as:

$$(\bar{u}, \bar{v}, \bar{w}) = \frac{(A_1, A_2, A_0)}{A_\Delta}. \tag{3.2}$$

Note that this normalization process makes the terms sum to one: $\bar{u} + \bar{v} + \bar{w} = 1$. Therefore, we often only compute $(\bar{u}, \bar{v})$, and derive the third coordinate as $\bar{w} = 1 - \bar{u} - \bar{v}$.

Given the barycentric coordinates of a point, $\mathbf{p}$, we can compute an interpolated s-value at $\mathbf{p}$ as shown below:

$$\begin{aligned} s &= \bar{w}s_0 + \bar{u}s_1 + \bar{v}s_2 = (1 - \bar{u} - \bar{v})s_0 + \bar{u}s_1 + \bar{v}s_2 \\ &= s_0 + \bar{u}(s_1 - s_0) + \bar{v}(s_2 - s_0). \end{aligned} \tag{3.3}$$

Now, return to the middle part of Figure 3.1. As can be seen there, the $\bar{u}$-parameter, which is computed as A_1/A_Δ, reaches 0 for all points on the line through $\mathbf{p}^0$ and $\mathbf{p}^2$. This line has a direction, $\mathbf{d} = \mathbf{p}^2 - \mathbf{p}^0$. For all points on a line with origin $\mathbf{p}^1$ and direction $\mathbf{d}$, we see that $\bar{u} = 1$. Why is it that $\bar{u}$ is constant for all points on a line with direction $\mathbf{d}$? The answer is that the area, A_1, of all such triangles is constant, because they have the same base length (the edge from $\mathbf{p}^0$ to $\mathbf{p}^2$), and the same height (perpendicular distance from the edge to $\mathbf{p}$). This also means that the $\bar{u}$-parameter varies linearly inside the triangle from the edge $\mathbf{p}^0\mathbf{p}^2$ to $\mathbf{p}^1$, since the height (perpendicular to $\mathbf{p}^0\mathbf{p}^2$) of the triangle $\Delta\mathbf{p}^1\mathbf{p}^0\mathbf{p}^2$ grows linearly. The same case for the $\bar{v}$-parameter is illustrated to the right in Figure 3.1. These facts even hold for points outside the triangle. However, for a point, $\mathbf{p}$, inside the triangle, it must hold that

$$\begin{aligned} \bar{u} &\geq 0, \\ \bar{v} &\geq 0, \text{ and} \\ \bar{w} &\geq 0 \Leftrightarrow \bar{u} + \bar{v} \leq 1. \end{aligned} \tag{3.4}$$

[1] Note that the $\bar{w}$ is not the w that we use for the homogenization process (Section 1.1.1).

© 2005–2012 Tomas Akenine-Möller

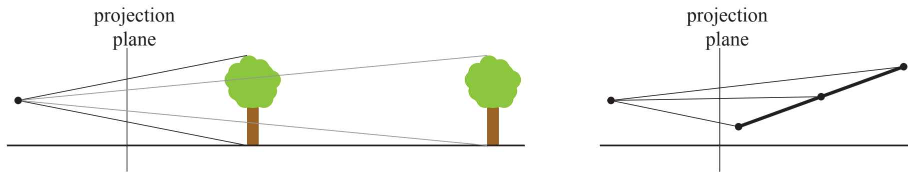

Figure 3.2: Left: per-object perspective foreshortening. The tree farther away appears smaller since its projection is smaller. Right: inside an object (in this case, a two-dimensional line), perspective foreshortening occurs as well and for the same reasons as the case to the left. Interpolation of, for example, color on the line must be done with perspective in mind.

Outside the triangle, the barycentric coordinates can be greater than one, and even negative.

Note that from now on, we avoid using the $\bar{w}$ and instead always use $1-\bar{u}-\bar{v}$ to avoid confusion with the fourth component in homogeneous coordinates.

3.2 Perspectively-Correct Interpolation

If you stand in a long corridor with doors on the walls, doors far away will appear smaller than the the ones being close. This is a simple example of perspective foreshortening, that is, objects' perceived size varies with the distance to the object. Another way of looking at this, is that objects far away project to a smaller size on your eye lobe, and thus naturally appear smaller.

Perspective foreshortening is something that we must handle in computer graphics as well. The reason for this is that some kind of perspective transform is often used, and these transforms mimic this behavior. However, even if the triangles appear with correct size, we often interpolate parameters inside the triangles, and this must also be done with perspective foreshortening in mind. See Figure 3.2 for two examples of these types of perspective effect. In Figure 3.3, the difference between perspective interpolation inside a polygon and linear interpolation is shown. Another example is shown in Figure 3.4, where another texture is used.

Perspective correct interpolation is known for its per-pixel division operation, that is, in order to compute perspectively-correct interpolation, one need to do one division per pixel. Here we will explain why, and how interpolation is done. From a high level, we want perspective-correct interpolation by using linear interpolation, which is inexpensive, as much as possible. Assume that each vertex has a parameter, s^i, $i \in [0,1,2]$, as shown in Figure 3.1. Blinn [11] and Heckbert & Moreton [28] first described how to do perspective-correct interpolation. Here, however, we will loosely follow the description by Olano and Greer [45].

As can be seen by looking at the perspective transform matrix, $\mathbf{M}$ (Equation 1.9 and 1.10), the h_x and h_y are simply the eye-space coordinates ($\mathbf{e}$) scaled

© 2005–2012 Tomas Akenine-Möller

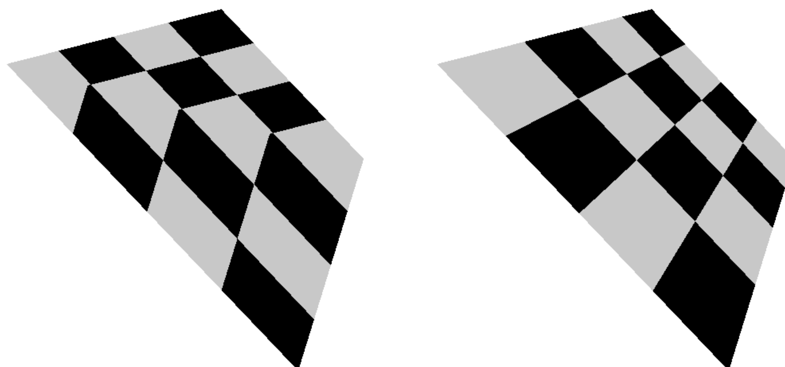

Figure 3.3: A quadrilateral (composed of two triangles) with a texture map of an 4×4 checker board. Left: without perspective correction. Right: with perspective correction. To the left, it is obvious which diagonal of the quadrilateral that is shared between the triangles.

by some factor, and possibly translated by some amount. Note also that h_w is simply the z-coordinate in eye space, which can be seen by looking at the same matrices. Therefore, the coordinates, (h_x, h_y, h_w), can be considered as $\mathbf{e}$ transformed into some "scaled eye space." Hence, we can write a formula for linearly interpolating the parameter s over a triangle in "scaled eye space" as:

$$s(h_x, h_y, h_w) = kh_x + lh_y + mh_w, \tag{3.5}$$

where k, l, m are some parameters that can be determined from the three vertices of the triangle. This is so, since $s^i = kh_x^i + lh_y^i + mh_w^i$ for all three vertices, $i \in [0, 1, 2]$, and thus we have three unknowns and three equations, and therefore the system is solvable for non-degenerate triangles. In Equation 3.5, the interpolation is done in a "scaled eye space." However, when we traverse the pixel inside a triangle, we do that in screen space, and therefore, we rewrite Equation 3.5 so that it contains the screen space coordinates, $(p_x, p_y) = (h_x/h_w, h_y/h_w)$, instead of (h_x, h_y, h_w):

$$\begin{gathered} s(h_x, h_y, h_w) = kh_x + lh_y + mh_w \Leftrightarrow \\ k\frac{h_x}{h_w} + l\frac{h_y}{h_w} + m\frac{h_w}{h_w} = kp_x + lp_y + m = \hat{s}(p_x, p_y) \end{gathered} \tag{3.6}$$

The surprising result is that we now can interpolate linearly in screen space, since the resulting function only depends on p_x and p_y. However, the function, denoted $\hat{s}(p_x, p_y)$, that can be interpolated in this way is s divided by w (or more correct, by h_w). The desired result from the perspective correct interpolation is not s/w but rather just s. To correct for that, the standard trick is to linearly interpolate the function $1/w$ over the triangle in screen space. Let us call that function $\hat{o}(p_x, p_y)$ (o is for *one*). So, if we interpolate s/w in screen space as

© 2005–2012 Tomas Akenine-Möller

Figure 3.4: Left: a square texture. Middle: without perspective correction. Right: with perspective correction. Notice that in the middle figure, the texture is distorted, and it does not behave as expected.

well as $1/w$, we can recover the correct parameter as:

$$\frac{s/w}{1/w} = \frac{sw}{w} = s. \tag{3.7}$$

This is where the per-pixel division comes from. More formally, Equation 3.7 is expressed as:

$$s(p_x, p_y) = \frac{\hat{s}(p_x, p_y)}{\hat{o}(p_x, p_y)} \tag{3.8}$$

Interpolation of $\hat{o}$ can be done in the same way as in Equation 3.6, and we then obtain:

$$\hat{o}(p_x, p_y) = k'p_x + l'p_y + m'. \tag{3.9}$$

Thus, we have showed that we can use linear interpolation in screen space of $\hat{s}$ and $\hat{o}$, and by computing $s = \hat{s}/\hat{o}$, we obtain the perspectively correct interpolation of s. This has been the goal of the entire chapter.

In theory, we can do perspectively correct interpolation by first computing s^i/h_w^i and $1/h_w^i$, $i \in [0, 1, 2]$, that is, for each vertex. This is typically done in the triangle setup. Then we compute the barycentric coordinates, $(\bar{u}, \bar{v})$, using Equation 3.2, for a particular pixel and use these to linearly interpolate both s/h_w and $1/h_w$. Finally, we correct for perspective by computing $s = \frac{s/h_w}{1/h_w}$. This generates a correct result, but usually this is not the best way to do it, especially when we have many parameters to interpolate in perspective. For example, we may have many texture coordinates per vertex, per-vertex color, fog, etc., that should be interpolated. For such cases, a better method is to compute perspectively correct interpolation coordinates once per pixel, and then interpolate all parameters using those. Such a technique is presented in Section 3.3.2.

© 2005–2012 Tomas Akenine-Möller

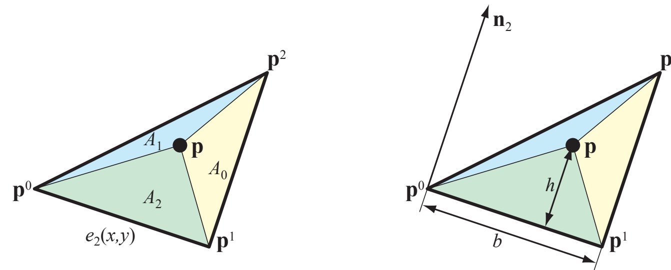

Figure 3.5: Left: a triangle with vertices $\mathbf{p}^0$, $\mathbf{p}^1$, and $\mathbf{p}^2$. We are interested in computing the barycentric coordinates, $(\bar{u}, \bar{v}) = (A_1, A_2)/(A_0 + A_1 + A_2)$. Right: for the $\bar{v}$-parameter, we want to compute $A_2 = bh/2$, which is the area of the darkest gray triangle.

3.3 Interpolation using Edge Functions

In this section, we will show how the edge functions from Section 2.1 can be used to compute barycentric coordinates, $(\bar{u}, \bar{v})$, and also perspectively-correct interpolation coordinates, (u, v).

3.3.1 Barycentric Coordinates

Again, assume we have a triangle $\Delta\mathbf{p}^0\mathbf{p}^1\mathbf{p}^2$, as shown to the left in Figure 3.5. Recall that the edge function, $e_2(x, y)$, for the triangle vertices $\mathbf{p}^0$ and $\mathbf{p}^1$ is computed as (see Equation 2.2):

$$e_2(x, y) = -(p^1_y - p^0_y)(x - p^0_x) + (p^1_x - p^0_x)(y - p^0_y) = a_2 x + b_2 y + c_2 \quad (3.10)$$

The "normal" of the edge is $\mathbf{n}_2 = (a_2, b_2)$. If we denote the point (x, y) by $\mathbf{p}$ then Equation 3.10 can be rewritten as:

$$e_2(x, y) = e_2(\mathbf{p}) = \mathbf{n}_2 \cdot (\mathbf{p} - \mathbf{p}^0) \quad (3.11)$$

From the definition of the dot product, this can be rewritten as follows:

$$e_2(\mathbf{p}) = ||\mathbf{n}_2||\,||\mathbf{p} - \mathbf{p}^0|| \cos\alpha, \quad (3.12)$$

where α is the angle between $\mathbf{n}_2$ and $\mathbf{p} - \mathbf{p}^0$. Note that $b = ||\mathbf{n}_2||$ must be equal to the length of the edge $\mathbf{p}^0\mathbf{p}^1$. The geometric interpretation of the second term $||\mathbf{p} - \mathbf{p}^0|| \cos\alpha$ is that it represents the length of the vector obtained when projecting $\mathbf{p} - \mathbf{p}^0$ onto $\mathbf{n}_2$, and that length must be exactly the height, h, of the sub-triangle $\Delta\mathbf{p}\mathbf{p}^0\mathbf{p}^1$. This is illustrated to the right in Figure 3.5.

This means that $e_2(x, y) = bh$, that is, the edge function returns twice the area, A_2. Thus, if the edge functions from Equation 2.2 are used,[2] then the

[2]Note that an edge function can be *any* implicit line equation through two vertices of a polygon. In this case, we assume that the *exact* definition from Equation 2.2 is used.

© 2005–2012 Tomas Akenine-Möller

computation of barycentric coordinates is particularly simple, as can be seen below:

$$\begin{aligned}\bar{u} &= \frac{e_1(x,y)}{2A_\Delta} \\ \bar{v} &= \frac{e_2(x,y)}{2A_\Delta}\end{aligned} \tag{3.13}$$

where A_Δ is the area of the triangle. It should be noted that $\bar{u}$ and $\bar{v}$ really are functions of (x, y), but that has been omitted in the equation above for clarity. The "third" barycentric coordinate is obtained as $1-\bar{u}-\bar{v}$ since all three should sum to one, or alternatively as $e_0(x, y)/(2A_\Delta)$. So, if edge functions are used to find pixels inside a triangle, we already have the evaluations of $e_1(x, y)$ and $e_2(x, y)$ for a particular pixel. The area is constant for the triangle, so $1/(2A_\Delta)$ can be computed as part of the triangle setup, and the division avoided.

These barycentric coordinates, $(\bar{u}, \bar{v})$, can be used to interpolate depth. If $d_0 = h^0_z/h^0_w$, $d_1 = h^1_z/h^1_w$, and $d_2 = h^2_z/h^2_w$ are the known depths for the three vertices, then $(\bar{u}, \bar{v})$ can be used to compute an interpolated depth value for a certain pixel. This is done as shown below:

$$d(x, y) = (1 - \bar{u} - \bar{v})d_0 + \bar{u}d_1 + \bar{v}d_2 = d_0 + \bar{u}(d_1 - d_0) + \bar{v}(d_2 - d_0). \tag{3.14}$$

3.3.2 Perspective-Correct Interpolation Coordinates

In this section, we show how to compute, what we call, *perspectively-correct interpolation coordinates*. What we mean by this is a set of coordinates, (u, v) (notice that the bars are missing now), that are similar to barycentric coordinates, $(\bar{u}, \bar{v})$, but (u, v) are computed with perspective in mind. Note, that these coordinates are *not* proportional to the areas of the subtriangles as previously described. Once, (u, v) have been computed, we can interpolate any parameters across the triangle in perspective. Thus, if we have some scalar s^i, $i \in [0, 1, 2]$ at each vertex, then, for a particular pixel, (x, y), we compute a perspective-correct s-value at (x, y) as:

$$s(x, y) = (1 - u - v)s^0 + us^1 + vs^2 = s^0 + u(s^1 - s^0) + v(s^2 - s^0). \tag{3.15}$$

A very nice property is that (u, v) can be used to interpolate any parameters that should be interpolated across the triangle using Equation 3.15.

The following is a derivation of how to compute (u, v), and the result is shown in Equation 3.23. In Figure 3.6, a triangle is shown (compare to the middle illustration of Figure 3.1), where the parameters $s^0 = s^2 = 0$ and $s^1 = 1$. If the s-parameters are interpolated with perspective taken into account, we will obtain the perspectively interpolated coordinate u. The reason for this is that $\bar{u} = 0$ at $\mathbf{p}^0$ and $\mathbf{p}^2$, and $\bar{u} = 1$ at $\mathbf{p}^1$ (again, see middle illustration of Figure 3.1).

According to Equation 3.8, u can therefore be computed as:

$$u(x, y) = \frac{\hat{s}(x, y)}{\hat{o}(x, y)}. \tag{3.16}$$

© 2005–2012 Tomas Akenine-Möller

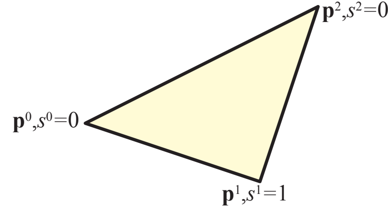

Figure 3.6: The triangle configuration for computing the "perspectively-correct" parameter, u. If the s^i-values, $i \in [0,1,2]$ are interpolated with perspective in mind, then we obtain u.

Both functions $\hat{s}$ and $\hat{o}$ are computed using linear interpolation, and so can be computed using $(\bar{u}, \bar{v})$:

$$\hat{s}(x,y) = (1-\bar{u}-\bar{v})\frac{0}{h_w^0} + \bar{u}\frac{1}{h_w^1} + \bar{v}\frac{0}{h_w^2} \tag{3.17}$$

$$\hat{o}(x,y) = (1-\bar{u}-\bar{v})\frac{1}{h_w^0} + \bar{u}\frac{1}{h_w^1} + \bar{v}\frac{1}{h_w^2} \tag{3.18}$$

The perspectively-correct u is then obtained as:

$$u(x,y) = \frac{\frac{\bar{u}}{h_w^1}}{\frac{(1-\bar{u}-\bar{v})}{h_w^0} + \frac{\bar{u}}{h_w^1} + \frac{\bar{v}}{h_w^2}}. \tag{3.19}$$

Using Equation 3.13, and some simplification, this reduces to:

$$u(x,y) = \frac{\frac{e_1}{h_w^1}}{\frac{e_0}{h_w^0} + \frac{e_1}{h_w^1} + \frac{e_2}{h_w^2}}, \tag{3.20}$$

where e_i, $i \in [0,1,2]$ are edge functions. To simplify notation a bit, we introduce the following functions:

$$f_0 = \frac{e_0(x,y)}{h_w^0}, \qquad f_1 = \frac{e_1(x,y)}{h_w^1}, \qquad f_2 = \frac{e_2(x,y)}{h_w^2}. \tag{3.21}$$

Using these, and doing a similar computation for the v-parameter, we obtain *perspectively-correct interpolation coordinates* as:

$$u(x,y) = \frac{f_1(x,y)}{f_0(x,y) + f_1(x,y) + f_2(x,y)} \tag{3.22}$$

$$v(x,y) = \frac{f_2(x,y)}{f_0(x,y) + f_1(x,y) + f_2(x,y)} \tag{3.23}$$

© 2005–2012 Tomas Akenine-Möller

	Notation	Description
1	$a_i, b_i, c_i,\ i \in [0,1,2]$	Edge functions
2	$\dfrac{1}{2A_\Delta}$	Half reciprocal of triangle area
3	$\dfrac{1}{h_w^i}$	Reciprocal of w-coordinates

Table 3.1: Triangle setup computations. These computation should include everything that can be performed only once per triangle, and thus factored out of the per-pixel computations.

These can be seen as rational basis functions [38]. Equations 3.22 and 3.23 are fundamental to triangle rasterization and interpolation, and constitute the core result of perspective-correct interpolation.

By using the least common divisor, Equation 3.20 can be rewritten as:

$$u(x,y) = \frac{(h_w^0 h_w^2)e_1}{(h_w^1 h_w^2)e_0 + (h_w^0 h_w^2)e_1 + (h_w^0 h_w^1)e_2}. \tag{3.24}$$

The same rewrite can be done for $v(x,y)$. This formulation may be beneficial, but the reciprocals $1/h_w^i$ are needed in the triangle setup anyway, in order to compute the depths h_z^i/h_w^i, and also the screen space coordinates, $(h_x^i/h_w^i, h_y^i/h_w^i)$, of the triangle. With the formulation in Equation 3.24, we also need to store three terms of the type: $h_w^i h_w^j$, or we can bake them into the edge function parameters, a, b, and c.

3.4 Triangle Setup and Per-Pixel Computation

In this section, we summarize what computations are done in the triangle setup, and what is done per pixel. It must be emphasized that there are endless variations on how to do this, and this is just one way of doing it.

The triangle setup calculations are shown in Table 3.1. As can be seen, the parameters for the edge functions, $e_i(x,y)$, $i \in [0,1,2]$ must be computed so that triangle traversal can be performed. Furthermore, half the reciprocal of triangle area is needed, since Equation 3.14 is used to compute per-pixel depth, and $\dfrac{1}{2A_\Delta}$ is used to compute $(\bar{u},\bar{v})$ as shown in Equation 3.13. Line 3 in Table 3.1 shows that the triangle setup should compute the reciprocal of the h_w^i for each vertex as well. These will be used when evaluating Equation 3.21.

In Table 3.2, the basic computations that need to be performed per pixel are shown. As can be seen, the edge functions need to be evaluated per pixel, so that we can find out whether the sample point, (x,y), is inside the triangle. After that, the depth test is usually done, and for that barycentric coordinates are computed (line 2), and then per-pixel depth is computed on line 3. If the depth test fails, the rest of the computations can be avoided. Otherwise, we

© 2005–2012 Tomas Akenine-Möller

	Notation	Description
1	$e_i(x, y)$	Evaluate edge functions at (x, y)
2	$(\bar{u}, \bar{v})$	Barycentric coordinates (Equation 3.13)
3	$d(x, y)$	Per-pixel depth (Equation 3.14)
4	$f_i(x, y)$	Evaluation of per-pixel f-values (Equation 3.21)
5	(u, v)	Perspectively-correct interpolation coordinates (Equation 3.23)
6	$s(x, y)$	Interpolation of all desired parameters, s^i (Equation 3.15)

Table 3.2: Per-pixel computations for pixel (x, y). Only the very basic computations are shown. In general, textures etc can be accessed and blended in arbitrarily in a pixel shader.

need to prepare for computation of perspectively-correct interpolation of any parameters. Therefore, f-values are computed (line 4), and this allows us to compute the (u, v). When this is done, we can interpolate any scalar parameter, s, with perspective in mind (line 6).

© 2005–2012 Tomas Akenine-Möller

Chapter 4

Pixel-Processing

This chapter is still not finished.

4.1 The Pixel-Processing Pipeline

TODO: Explain the simplest. Explain depth fail/pass.

4.2 The Pixel-Processing Equation

Message to students: the following is just quick, non-finished notes. Refer to the slides from the course for a more complete presentation.

Assuming the scene to be rendered has an average depth complexity of d, the average overdraw, $o(d)$, can be estimated as [14]:

$$o(d) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{d}. \tag{4.1}$$

FIX: explain the formula above. Interesting: the above formula shows the fragments that pass the depth test. The fragments that fail is d-o, but we can also present a formula for that, which could be interesting to look at (for better understanding).

Stupid formula.

$$b = d \times (Z_r + Z_w + C_w + T_r), \tag{4.2}$$

Without using any algorithms or special techniques to reduce bandwidth, the bandwidth required by a single pixel for a single view is [7]:

$$b = d \times Z_r + o \times (Z_w + C_w + T_r), \tag{4.3}$$

where Z_r and Z_w are the cost for reading and writing a depth buffer value, respectively. Furthermore, C_w is the cost for writing to the color buffer (this assumes that no blending is done, since the term C_r is missing) and T_r is the

total cost for accessing textures for a fragment. Standard values for these are: $Z_r = Z_w = C_w = 4$ bytes and a single texel is often stored in four bytes. Trilinear mipmap filtering [56] is commonly used for reducing aliasing artifacts, and this requires eight texel accesses, which makes $T_r = 8 \times 4 = 32$ bytes for a filtered color from one texture.

In a sense, Equation 4.3 is proof for that the rendering pipeline can be a true brute force (and hence kind of stupid) architecture. However, several chapters that follow, will reduce this cost. Continue here...

If a texture cache [24] with miss rate, m, is used Equation 4.3 becomes:

$$\begin{aligned} b &= d \times Z_r + o \times (Z_w + C_w + m \times T_r) \\ &= \underbrace{d \times Z_r + o \times Z_w}_{\text{depth buffer, } B_d} + \underbrace{o \times C_w}_{\text{color buffer, } B_c} + \underbrace{o \times m \times T_r}_{\text{texture read, } B_t} \\ &= B_d + B_c + B_t \end{aligned} \tag{4.4}$$

The equation above is what we call the *pixel-processing equation*.

© 2005–2012 Tomas Akenine-Möller

Chapter 5

Texturing

Texturing [9] is the process of applying some image, called a texture, to a primitive being rendered in order to add detail to a rendered scene. The most common example in real-time rendering is to "glue" a two-dimensional image onto a triangle that is being rendered. However, texturing is also used for many other things, and instead of storing image content in the texture, some other parameter can be stored. For example, when using bump mapping [10], a geometrical displacement, such as a normal vector, can be stored in the image instead. With programmable shaders, this trend has exploded, as the programmer has the possibility to use his/her imagination to come up with clever things to store in a texture, and to use this information to achieve some desired effect. This chapter will mainly deal with two-dimensional textures, but many of the concepts easily generalize to higher dimensions. Two examples of texturing are shown in Figure 5.1.

This chapter starts with a description of texture images, texture space, and

Figure 5.1: Left: a simple texture is "glued" onto a teapot. Right: a variety of texturing techniques are used to create an ocean surface. Bump mapping, reflection mapping and refraction mapping are used.

texture coordinates. Then follows a theoretical explanation of texture filtering in Section 5.2. For example, bilinear filtering and trilinear mipmapping are discussed, and these are techniques for obtaining better quality and reducing the effect of aliasing. A description of how caches work in general in presented in Section 5.5. When both caches and filtering are understood, texture caching can be explained. Finally, texture compression is a mean for further reducing bandwidth at some small loss of image quality, and several techniques for that is described at the end of this chapter (but for now, we will only give pointers to papers to be read—write about texture caching and texture compression later).

5.1 Texture Images

In this section, texture images, their corresponding texture space, and a triangle's texture coordinates will be discussed. In general, we refer to a texture as an n-dimensional image consisting of $w_1 \times w_2 \times \cdots \times w_n$ image elements called *texels*. To simplify texturing hardware, it is often assumed that the dimensions of a texture are powers of two,[1] that is, $w_i = 2^{e_i}$, where e_i is a positive integer. In the majority of cases, each texel can hold a color, for example stored as RGB (24 bits) or RGBA (32 bits). However, the principles described here can be applied to, e.g., textures containing only intensity content or floating point values, or the texture image can even be stored in a compressed format.

To render a triangle with a texture on it, texture coordinates, $\mathbf{c}^k$, $k \in 0, 1, 2$, need to be assigned to the triangle's vertices. Each texture coordinate is simply a set of two-dimensional coordinates (when two-dimensional textures are used), e.g., $\mathbf{c}^0 = (c_s^0, c_t^0)$. The texture space in OpenGL is denoted (s, t), and so that is used here as well. Texture coordinates and a texture image are illustrated in Figure 5.2. From the figure, it can be understood that it is the responsibility of the user to assign the texture coordinates to the triangle's vertices in order to achieve the desired result.

From Section 3.3.2, it is known that perspective correct interpolation coordinates, (u, v), can be computed for each fragment. These are then used together with the triangle's texture coordinates, $\mathbf{c}^k$, to compute the fragment's coordinate in texture space, (s, t), as shown below, which is simply the standard formula for interpolation of parameters in a triangle:

$$(s, t) = (1 - u - v)\mathbf{c}^0 + u\mathbf{c}^1 + v\mathbf{c}^2. \tag{5.1}$$

The different coordinate systems, including (s, t), that are involved in texturing are illustrated in Figure 5.3. The (s, t)-coordinates are floating point values, and to simplify notation a bit, we use:

$$\begin{aligned} \tilde{s}(x, y) &= w \times s(x, y), \\ \tilde{t}(x, y) &= h \times t(x, y), \end{aligned} \tag{5.2}$$

[1] Though, arbitrary rectangles can be handled by most desktop graphics hardware.

© 2005–2012 Tomas Akenine-Möller

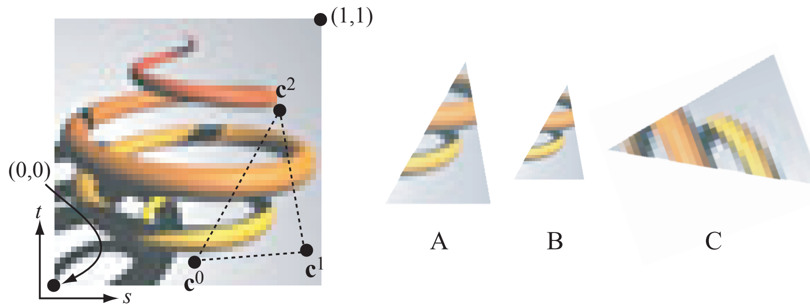

Figure 5.2: To the left, a simple two-dimensional texture is shown in its texture space, (s, t). To the right, three triangles with different positions, scales, and orientations have been rendered. However, note that they have the same texture coordinates, $\mathbf{c}^k$, $k \in [0, 1, 2,]$, and so the same part of the texture image is glued onto the rendered triangles.

which are simply the s-coordinate multiplied with the texture's width, w, and the t-coordinate multiplied with the texture height, h. In the OpenGL specification, $(\tilde{s}, \tilde{t})$ are instead denoted (u, v), but the former notation is used here, since (u, v) already have been used for the perspectively correct interpolation coordinates (see Equation 3.23).

For now, we assume that texels are stored contiguously in memory, and these can thus be enumerated based on memory location. Assume that first texel has number 0, and the following texel, number 1, and so on. If a texel has number n, then this can be converted to a set of two-dimensional integer coordinates, (i, j), as shown below.

$$\begin{aligned} i &= n \bmod w, \\ j &= \left\lfloor \frac{n}{w} \right\rfloor \bmod h, \end{aligned} \tag{5.3}$$

where again $w \times h$ is the texture resolution.[2] The (i, j) coordinate pair is also illustrated in Figure 5.3. Given (i, j), the texel number can be computed as $n = i + wj$, and from n the address of the desired texel can be computed. In the following section, we will see how these texture spaces and coordinates are used to access texels, and filter them in order to obtain better quality.

5.1.1 Wrapping

Interestingly, the texture coordinates, $\mathbf{c}^k$, at a triangle's vertices need not be in $[0, 1] \times [0, 1]$. Instead, larger numbers than 1.0 and/or smaller than 0.0 can be used. The most common usage for this is when one wants a texture to repeat itself over a triangle. For example, say an application models a 100×100

[2]Texture borders have been omitted from this presentation. Consult the OpenGL specification for more information [51].

© 2005–2012 Tomas Akenine-Möller

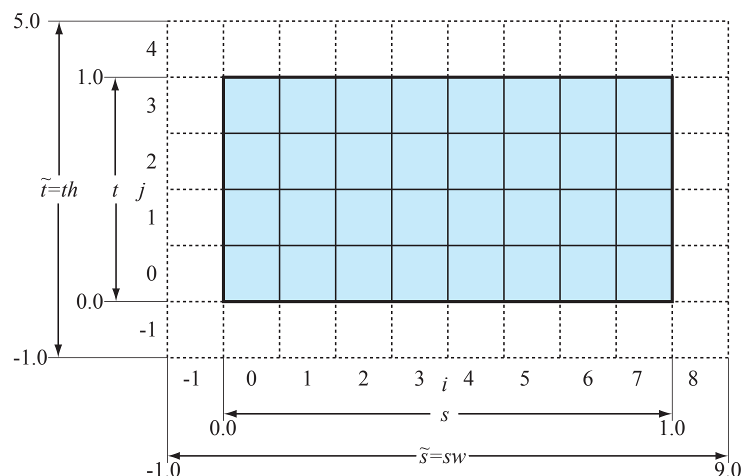

Figure 5.3: A texture image with resolution $w \times h = 8 \times 4$ texels. Notice the different coordinate systems. In texture space, there are unit coordinates, $(s, t) \in [0, 1] \times [0, 1]$, and when these are multiplied by the texture resolution, $w \times h$, we get $(\tilde{s}, \tilde{t}) = (sw, th) \in [0, w] \times [0, h]$. There are also the enumeration, (i, j), of the texels. Texel $(0, 0)$ is located at the bottom left corner, and $(w-1, h-1)$ is located in the top right corner. (Illustration after Segal and Akeley [51])

m^2 as a single quadrilateral, and that a 2×2 m^2 texture image of grass is available. Instead of stretching the small texture so that it fits over the entire quadrilateral, one can instead repeat the small texture 50×50 times over the quadrilateral. This can be done by setting texture coordinates to $(0, 0)$ in the lower left corner of the quadrilateral and $(50, 50)$ in the upper right corner (and appropriate values in the remaining two corners).

The most common wrapping modes are *repeat*, *clamp*, and *mirrored repeat*. Clamping sees to it that s and t always are clamped to the range $[0, 1]$. This means that if s is greater than 1.0, then set $s = 1.0$, and if it is smaller than 0.0 then set $s = 0.0$.

The repeat texture wrap mode (which would be used in the grass example above), the integer part of s and t are ignored, and only the fractional part is kept. The fractional part of s can be computed as:

$$s' = s - \lfloor s \rfloor, \tag{5.4}$$

where $\lfloor\rfloor$ truncates towards $-\infty$ to the closest integer. This means that s' never will reach 1.0, but rather $s \in [0, 1)$. In Section 5.3, we will see why this is convenient.

Mirrored repeat is just what it sounds like, the texture is repeated, but every other occurrence of the texture is mirrored. In OpenGL [51], mirrored repeat

© 2005–2012 Tomas Akenine-Möller

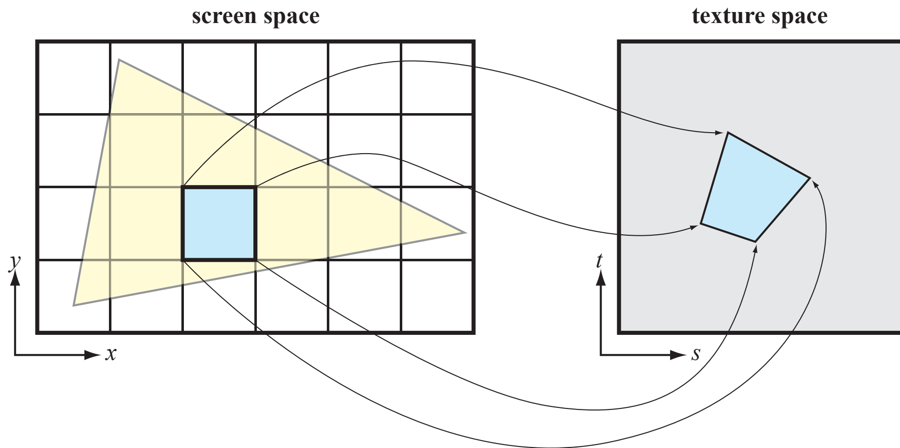

Figure 5.4: In screen space, the borders of a single pixels (blue) are projected back into texture space, (s, t), and there the borders become a convex quadrilateral with straight edges. This quadrilateral is called the footprint of the pixel. In texture filtering, one attempts to quickly estimate the colors of the texels under the dark gray quadrilateral to the right.

of s is defined as:

$$s' = \begin{cases} s - \lfloor s \rfloor & \lfloor s \rfloor \text{ is even} \\ 1 - (s - \lfloor s \rfloor) & \lfloor s \rfloor \text{ is odd} \end{cases} \tag{5.5}$$

5.2 Texture Filtering

In order to render textured scenes with good quality, some kind of *texture filtering* is needed. This is to avoid aliasing that can occur under *minification*, and avoid a blocky appearance under *magnification*. Recall from Chapter 3 that it is possible to compute perspectively-correct interpolation parameters, (u, v), for each pixel that is traversed. In this section, the (u, v)-parameters will be used to look up texels from a texture image, and techniques will be presented that take a set of these and filter them in order to produce a better result.

Here, the most common interpretation of how texture filtering should be done will be explained. In Figure 5.4, the focus is on a single pixel in screen space. When the borders of this pixel are transformed into texture space, the *footprint* of the pixel in texture space is obtained. The footprint takes the form of an arbitrary convex quadrilateral, and its shape depends on the texture coordinates and the vertices' positions and amount of perspective. It should be pointed out that we usually only use a single sample per pixel or a few samples, and thus using the pixel borders does not correspond to the sample(s) of the pixel [19]. Still, this is the model being used.

In Figure 5.5, two cases are illustrated. When many texels reside under the

© 2005–2012 Tomas Akenine-Möller

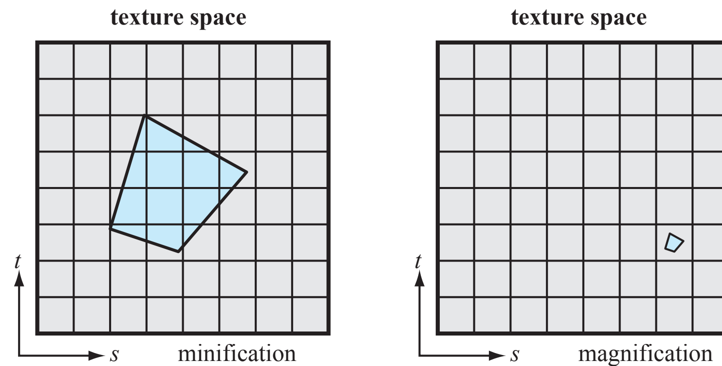

Figure 5.5: Left: when minification occurs the pixel footprint (blue quadrilateral) in texture space covers many texels. Right: when magnification occurs, few or only one (as in this case) texel is covered by the footprint.

area of the pixel footprint as shown to the left, minification occurs. Similarly, when few or only one texels is under the area of the pixel, magnification takes place. Intuitively, this latter case means that the texture is enlarged or magnified when glued onto a triangle. In the former case, the texture content is instead compressed, or minified onto the triangle.

In the following two sections, we describe magnification filtering and minification filtering techniques.

5.3 Magnification

Texture magnification occurs, for example, when we render a square that covers 512×512 pixels using a texture that is only, say, 64×64. Most graphics hardware only support two different filtering techniques:[3]

- Nearest neighbor sampling[4]
- Bilinear filtering

Examples of generated results using these two techniques are shown in Figure 5.6.

5.3.1 Nearest Neighbor Sampling

Nearest neighbor sampling is the simplest form of sampling that can be done since it involves finding only the single closest texel, (i, j). The actual compu-

[3]However, cubic filtering can be done by the user in a pixel shader. REF GPU Gems II.

[4]This is sometimes also called point sampling. However, that term is quite misleading as all filtering techniques use point sampling of the texture, and then possibly weigh several samples to produce the filtered color.

© 2005–2012 Tomas Akenine-Möller

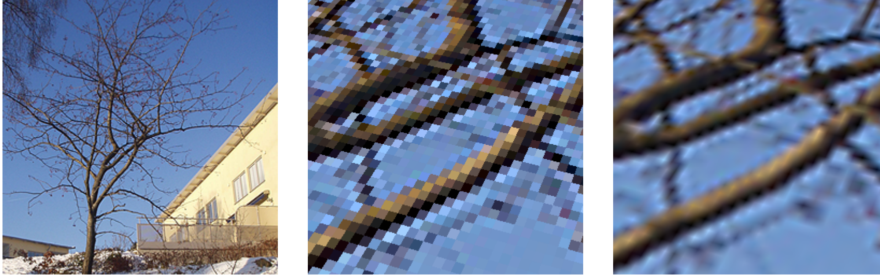

Figure 5.6: To the left, a 256×256 texture is shown. This image has been put on a quadrilateral, and zoomed in on in perspective. The middle image shows part of that scene with nearest neighbor sampling, while the image to the right shows bilinear magnification.

tation is done as shown below.

$$
\begin{aligned}
i &= \begin{cases} \lfloor \tilde{s} \rfloor & s < 1 \\ w-1 & s = 1 \end{cases} \\
j &= \begin{cases} \lfloor \tilde{t} \rfloor & t < 1 \\ h-1 & t = 1 \end{cases}
\end{aligned}
\tag{5.6}
$$

The $w \times h$ is the texture resolution as usual. The texel is accessed as $\mathbf{t}(i, j)$, where $\mathbf{t}$ is used to denote a texture look-up, and the texel color at (i, j) is returned from this function. Recall that neither s nor t ever reaches 1.0 when the wrap mode is *repeat*. Here, the advantage of that can be seen because $\tilde{s} = sw$ will never quite reach w, and hence addressing outside the texture will be avoided.

5.3.2 Bilinear Filtering

In the middle image in Figure 5.6, it can be seen that the appearance of using nearest neighbor sampling is quite blocky. This could be the desired effect, but often, it is quite disturbing and best avoided, and that can be done with bilinear filtering. The general idea is to use the closest 2×2 texels to the texture coordinates (s, t), and weight these texels according to the exact location of (s, t) inside this 2×2 texel block. This way a smoother transition is created, as shown to the right in Figure 5.6. To do this, two pairs of texels, (i_0, j_0) and (i_1, j_1) must be identified, and the other two texels are simply (i_0, j_1) and (i_1, j_0), as illustrated in Figure 5.7. The first texel is:

$$
\begin{aligned}
i_0 &= \begin{cases} \lfloor \tilde{s} - 1/2 \rfloor \bmod w, & \text{if wrap mode for } s \text{ is repeat,} \\ \lfloor \tilde{s} - 1/2 \rfloor, & \text{else.} \end{cases} \\
j_0 &= \begin{cases} \lfloor \tilde{t} - 1/2 \rfloor \bmod h, & \text{if wrap mode for } t \text{ is repeat,} \\ \lfloor \tilde{t} - 1/2 \rfloor, & \text{else.} \end{cases}
\end{aligned}
\tag{5.7}
$$

© 2005–2012 Tomas Akenine-Möller

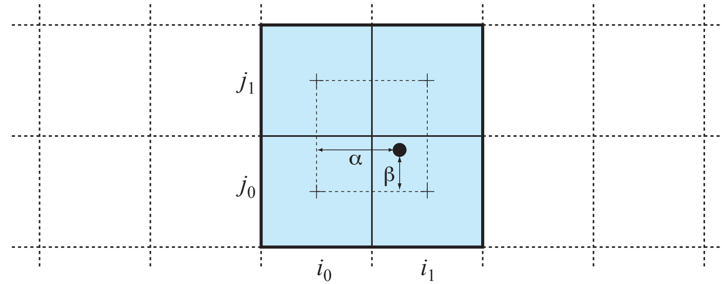

Figure 5.7: Illustration of bilinear filtering. The 2×2 gray texels are used by bilinear filtering. The actual sample point in the texture is marked with a black circle, and the fractional values (α, β) are used to weight the colors from these four texels.

The second texel is:

$$
\begin{aligned}
i_1 &= \begin{cases} (i_0+1) \bmod w, & \text{if wrap mode for } s \text{ is repeat,} \\ i_0+1, & \text{else.} \end{cases} \\
j_1 &= \begin{cases} (j_0+1) \bmod h, & \text{if wrap mode for } t \text{ is repeat,} \\ j_0+1, & \text{else.} \end{cases}
\end{aligned} \tag{5.8}
$$

In Figure 5.8, we compare nearest neighbor sampling and bilinear filtering again. Focus on the lower left corners where one white and one red texel are located. As can be seen in the bilinear filtered image, the white and red colors are only exactly white and red at the center of the respective zoomed up texels to the left. It is the factors $-1/2$ in Equation 5.7 that sees to that this happens. This is clearly just a matter of definition, but this is the way OpenGL defines it.

To actually compute the blending of the four texels, the following fractional values need to be computed:

$$
\alpha = \mathrm{frac}(\tilde{s} - 1/2), \qquad \beta = \mathrm{frac}(\tilde{t} - 1/2). \tag{5.9}
$$

To make the notation a bit shorter, we use $\mathbf{t}_{00} = \mathbf{t}(i_0, j_0)$, i.e., $\mathbf{t}_{00}$ is the texel color at texel position (i_0, j_0). The same goes for $\mathbf{t}_{01}$, $\mathbf{t}_{10}$, and $\mathbf{t}_{11}$. With bilinear filtering, linear filtering is done in two passes. The first pass can go in either direction, but we will use the i-direction (normally x) in Figure 5.7 This means that the α value is used to compute two new colors, $\mathbf{a}$ and $\mathbf{b}$ by linearly blending $\mathbf{t}_{00}$ & $\mathbf{t}_{10}$, and $\mathbf{t}_{01}$ & $\mathbf{t}_{11}$.

$$
\begin{aligned}
\mathbf{a} &= (1-\alpha)\mathbf{t}_{00} + \alpha\mathbf{t}_{10} = \mathbf{t}_{00} + \alpha(\mathbf{t}_{10} - \mathbf{t}_{00}) \\
\mathbf{b} &= (1-\alpha)\mathbf{t}_{01} + \alpha\mathbf{t}_{11} = \mathbf{t}_{01} + \alpha(\mathbf{t}_{11} - \mathbf{t}_{01})
\end{aligned} \tag{5.10}
$$

When $\mathbf{a}$ and $\mathbf{b}$ have been computed, linear interpolation is done once again, but this time in the other direction, and using β and the recently computed colors, $\mathbf{a}$ and $\mathbf{b}$.

$$
\mathbf{f} = (1-\beta)\mathbf{a} + \beta\mathbf{b} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) \tag{5.11}
$$

© 2005–2012 Tomas Akenine-Möller

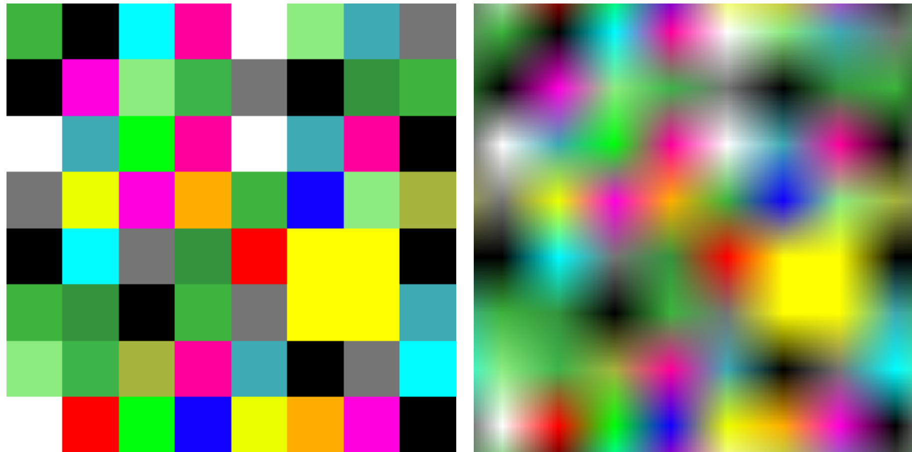

Figure 5.8: Left: an 8×8 random texture rendered onto a quadrilateral with nearest neighbor sampling. The results is that the texels are rendered as large squares onto the quadrilateral. Right: the same texture is rendered using bilinear filtering.

The formulae from Equations 5.10 and 5.11 can easily be merged into a single equation:

$$\mathbf{f} = (1-\alpha)(1-\beta)\mathbf{t}_{00} + \alpha(1-\beta)\mathbf{t}_{10} + (1-\beta)\alpha\mathbf{t}_{01} + \alpha\beta\mathbf{t}_{11}. \tag{5.12}$$

Assuming each texel has four components, RGBA, the use of Equations 5.10 and 5.11 imply that $3 \times 4 = 12$ multiplications and $6 \times 4 = 24$ adders need to be implemented for bilinear filtering. Using Equation 5.12 means that $4+4\times 4 = 20$ multipliers (note that the 4×4 of these need higher accuracy, since the scalar α- and β-terms are multiplied first) are needed and $3 \times 4 = 12$ adders.

5.4 Minification

When the texture is minified onto an object, the retrieved texel should be some weighted combination of texels under the footprint of the screen space pixel as previously discussed in this chapter. During minification, it is therefore not sufficient to use nearest neighbor sampling since only a single texel is accessed. This is shown to the left in Figure 5.9. Note that when, for example, the camera animates slowly, the aliasing artifacts becomes even more annoying since this results in serious flickering. Therefore, it is usually more acceptable to use a filter with more overblurring, rather than undersampling (resulting in flickering). Bilinear filtering can be used as a filter during minification, but that only helps a little bit. As soon as more than about four texels are in the footprint, the undersampling effects become very clear. In this section, better techniques are presented.

© 2005–2012 Tomas Akenine-Möller

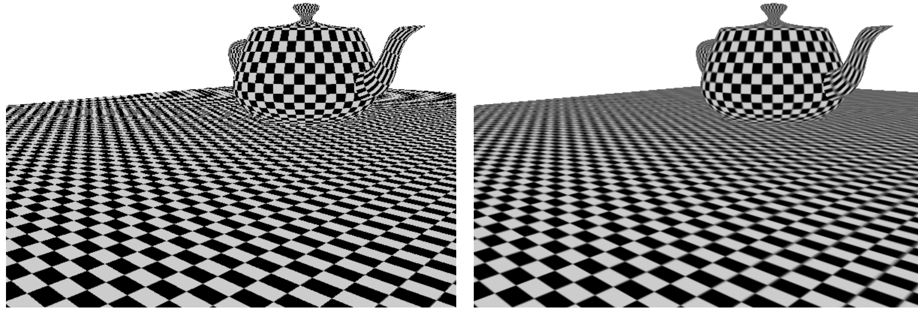

Figure 5.9: A checkered teapot on a checker plane. Left: with nearest neighbor sampling. Right: with trilinear mipmapping. When animated, the nearest neighbor sampling results in severe flickering, while the mipmapped scene animates smoothly.

In theory, the retrieved color, $\mathbf{f}$, can be expressed as shown in Equation 5.13 [50].

$$\mathbf{f}(x, y) = \int\int h(x - \alpha, y - \beta)\mathbf{t}(\alpha, \beta)d\alpha d\beta \tag{5.13}$$

In the equation above, h is a filter kernel, the integration domain is the footprint, and $\mathbf{t}$ accesses the texels in the texture as usual. Equation 5.13 can be approximated as:

$$\mathbf{f}(x, y) \approx \frac{1}{n}\sum_{i=1}^{n}\mathbf{t}_i, \tag{5.14}$$

but even then, the cost is way too high to be considered for graphics hardware.

One solution that achieves a better result is *summed-area tables* [16]. However, that technique requires more bits[5] per color component than the original image, and uses rather random accesses in the memory to produce a filtered color (and hence texture caching would not work that well). Next, mipmapping [56] will be presented, whose use is very widespread.

5.4.1 Mipmapping

Mip is an abbreviation for "multum in parvo," which is latin for "much in a small place." The key to mipmapping [56] is to use a hierarchy of images, called an image pyramid [18], or simply just a mipmap.[6] The images of a mipmap for a particular texture is shown in Figure 5.10. The corresponding mipmap

[5] For a 1024×1024 texture with 8 bits per color component, a summed-area table texture would require 28 bits per component [50].

[6] The use of an image pyramid for texture filtering was apparently first suggested by Ed Catmull in his PhD thesis (not easily available) as reported by Heckbert [27]. Dungan et al. [18] presented an algorithm that uses an image pyramid for texture filtering, but only a single mipmap level was used, and nearest neighbor sampling was used in that level. Williams [56] improved on that scheme by suggesting trilinear mipmapping, which is explained in detail in this chapter.

© 2005–2012 Tomas Akenine-Möller

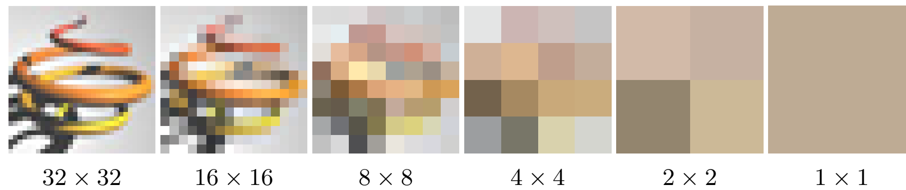

Figure 5.10: To the left, the original 32×32 texture is shown. The image is then low-pass filtered, and downsampled to 16×16, 8×8, 4×4, 2×2, and finally (to the right) to a singel texel. This hierarchy of images is used for a minification technique called mipmapping.

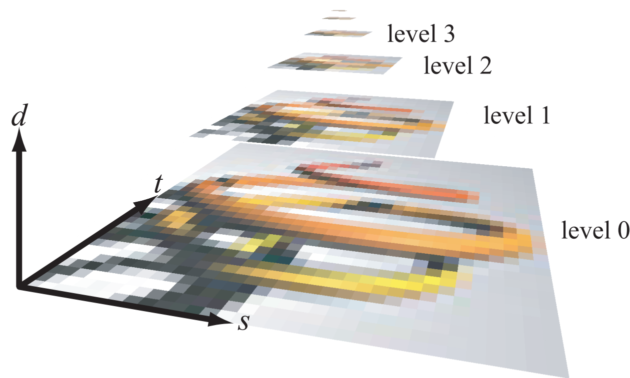

Figure 5.11: A mipmap pyramid using the images from Figure 5.10, and the pyramid's coordinate system, (s, t, d), which is used when accessing the texels during filtering.

pyramid is shown in Figure 5.11. Notice, that the original texture is called the level 0 base texture. The next level in the pyramid is a lowpass filtered and then downsampled version of the level 0 texture. The size is half the width and half the height of the level 0 texture. The level 2 texture, is a lowpass filtered and downsampled version of the level 1 texture, and so on. An advantageous feature of the mipmap pyramid is that it does not occupy that much extra memory. The amount of memory for the entire mipmap pyramid can be expressed as follows:

$$m + \frac{m}{2 \cdot 2} + \frac{m}{4 \cdot 4} + \frac{m}{8 \cdot 8} + \cdots = m \sum_{i=0}^{l} \frac{1}{4^i} \approx m \frac{1}{1 - \frac{1}{4}} = \frac{4}{3} m, \tag{5.15}$$

where m is the number of bytes for the base texture, and l is the level of the tip of the pyramid. Hence, it only occupies 33% more than the base texture, which is quite affordable.

In mipmapping, the footprint of a screen space pixel in texture space is used to estimate which level(s) should accessed and filtered to form the filtered

© 2005–2012 Tomas Akenine-Möller

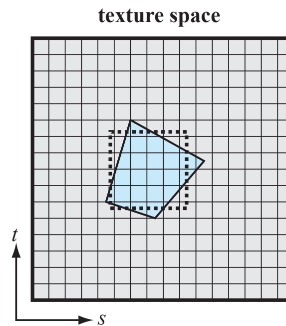

Figure 5.12: The blue footprint in texture space is shown. In mipmapping filters, the area of the footprint is used to estimate the size of a square filter, as outlined by the dashed square. The size of the square is used to determine the this texture space parameter, d.

color, $\mathbf{f}$. It should be noted that each texel in the mipmap pyramid is some weighted average of the texels in a square under the texel. This means that an *isotropic* square filter shape is used. The size of the filter can change, but not the shape. In fact, as can be seen in Figure 5.4, the filter shape should ideally be *anisotropic*. In Section 5.4.2, anisotropic minification algorithms will be presented.

Level-of-Detail Computation

A key to mipmapping is to compute which level(s) in the pyramid that should be accessed. As can be seen in Figure 5.11, another axis, d, has been added to the texture space, (s, t), which has been used so far. In Figure 5.12, the footprint in texture space is shown again.

The general idea is to compute the area, a, of the footprint, and since the mipmap can only use square filters, this area is assumed to be square as well. Hence, the side, b, of the square is computed as $b = \sqrt{a}$. The mipmap level to access is determined by the side, b, such that the side has similar size as the side of a texel. Since the most detailed level in the mipmap pyramid is enumerated 0, and the level above it, is level 1, and so on, the d-parameter is computed as:

$$d = \log_2 b = \log_2 \sqrt{a} = \log_2(a^{0.5}) = 0.5 \log_2 a \tag{5.16}$$

Assume, we have a 256×256 texture, which means that we have nine levels in the mipmap pyramid. The side of the square can be 256 at most, which means $d = \log_2 256 = 8$, which is the topmost mipmap level (recall that the bottommost mipmap level has number 0). Another example is when the side of the square is 1, and then $d = \log_2 1 = 0$, and thus the bottommost level is chosen. Intuitively, you need to locate the level in the mipmap pyramid where the side of a texel has approximately the same side as the footprint of a pixel in texture space.

© 2005–2012 Tomas Akenine-Möller

It is interesting to consider what happens when $d < 0$. The side of the footprint is then smaller than the side of a texel in the bottommost level. Thus, texture magnification takes place. Hence, the hardware can compute d, and then if $d < 0$ the texture magnification filter should be used, and otherwise the texture minification filter should be used. This can also be seen in Figure 5.5. Clamping on d should be done so that d never becomes larger that the topmost level in the mipmap hierarchy. Up until now, the discussion about how to compute d has been mostly theoretical. The text now continues with a presentation of different techniques on how to compute d in practice.

Recall that perspectively-correct interpolation coordinates, (u, v), can be computed as shown in Equation 3.23, and that the texture coordinates, (s, t), are computed using Equation 5.1. Furthermore, we have defined $\tilde{s} = sw$ and $\tilde{t} = th$. Thus, $\tilde{s}$ and $\tilde{t}$ are functions of the pixel coordinates (x, y): $\tilde{s}(x, y)$ and $\tilde{t}(x, y)$. The most common way to compute d is to use the partial derivatives, $\partial\tilde{s}/\partial x$, $\partial\tilde{s}/\partial y$, $\partial\tilde{t}/\partial x$, and $\partial\tilde{t}/\partial y$, and the presentation here will mostly concern those ways. An alternative would be to compute the coordinates of the footprint quadrilateral. It should also be noted that some implementations have computed only a single d for the entire triangle, and others have computed a per-vertex d-value, and then interpolated over the triangle with perspective taken into account. Here, we will only discuss per-pixel computations though.

As suggested by Heckbert [REF], and also recommended by the OpenGL specification [51], the level of detail, d, should be computed as:

$$d = \log_2 \left(\max \left[\sqrt{\left(\frac{\partial\tilde{s}}{\partial x}\right)^2 + \left(\frac{\partial\tilde{t}}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial\tilde{s}}{\partial y}\right)^2 + \left(\frac{\partial\tilde{t}}{\partial y}\right)^2} \right] \right) \tag{5.17}$$

Equation 5.17 can be simplified [41] so that the two expressions under the square roots are computed first. The equation then becomes:

$$d = \log_2 \left(\max\left[\sqrt{e_1}, \sqrt{e_1}\right] \right) = \log_2 \left(\max[e_1, e_1] \right)/2, \tag{5.18}$$

where $e_1 = (\partial\tilde{s}/\partial x)^2 + (\partial\tilde{t}/\partial x)^2$ and $e_2 = (\partial\tilde{s}/\partial y)^2 + (\partial\tilde{t}/\partial y)^2$. As can be seen, the square root has been lifted outside the $\log_2$ and there, it converts to a division by two.

Sometimes the following, less expensive approximation[7] is used to compute d:

$$d = \log_2 \left(\max \left[\text{abs}\left(\frac{\partial\tilde{s}}{\partial x}\right), \text{abs}\left(\frac{\partial\tilde{t}}{\partial x}\right), \text{abs}\left(\frac{\partial\tilde{s}}{\partial y}\right), \text{abs}\left(\frac{\partial\tilde{t}}{\partial y}\right) \right] \right). \tag{5.19}$$

However, as shown by McCormack et al. [41] it is a too gross approximation. Instead, they propose to use the following piecewise linear approximation of the expression $\sqrt{x^2 + y^2}$, where $x > y$:

$$\sqrt{x^2 + y^2} = \begin{cases} x + y/8, & \text{if } y < x/2 \\ 7x/8 + y/4, & \text{else} \end{cases} \tag{5.20}$$

[7]This is also the lower bound on the computation of d according to the OpenGL specification [51], i.e., the approximation may not be worse than this.

© 2005–2012 Tomas Akenine-Möller

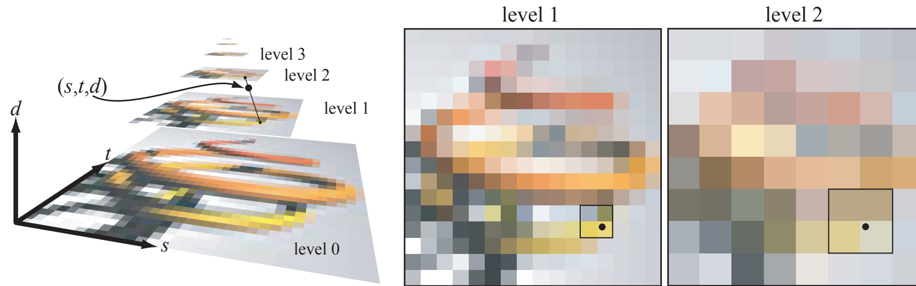

Figure 5.13: Trilinear mipmapping. A sample location is located at the black circle in the mipmap pyramid to the left. The two closest levels are level 1 and level 2. To the right, the four closest texel in each level are shown, and these are used for bilinear filtering at each level. When the two bilinearly filtered colors have been computed, linear blending is performed to compute the final color. This is further illustrated in Figure 5.14.

This approximation is within a 3% error margin of the exact value of $\sqrt{a^2+b^2}$.

The partial derivatives, $\partial\tilde{s}/\partial x$ etc., can be computed analytically by differentiating Equation 3.23. Ewins et al. present many other methods for this, including incremental techniques, and they also have a thorough survey of different ways to compute the level of detail [19].

At this point, we assume that (s,t,d) have been computed, and that a filtered texture color, $\mathbf{f}$, should be computed. Several filtering techniques are presented in the following subsections.

Trilinear Mipmapping

The most common form of mipmapping is trilinear mipmapping, and as can be understood from its name, linear filtering is performed three times. This is done using bilinear filtering in the two mipmap levels closest to d. Thus, levels l_0 and l_1 are computed as:

$$l_0 = \lfloor d \rfloor, \quad \text{and} \quad l_1 = l_0 + 1. \tag{5.21}$$

This situation is illustrated in Figure 5.13. Notice that care has to be taken so that only levels that exist in the pyramid are accessed, which can be done by simple clamping.

When those two levels have been identified, the (s,t) are scaled with the respective width and height of the two levels, so that two coordinate pairs are formed: $(\tilde{s_0},\tilde{t_0}) = (sw_0, th_0)$ and $(\tilde{s_1},\tilde{t_1}) = (sw_1, th_1)$, where $w_0 \times h_0$ is the resolution of level l_0 and $w_1 \times h_1 = [w_0/2] \times [h_0/2]$ is the resolution of level l_1. For level l_0, the parameters (α_0, β_0) are computed using Equation 5.9, and similarly for level l_1: (α_1, β_1). These are then used to compute two bilinearly filtered colors, $\mathbf{f}_0$ and $\mathbf{f}_1$, one for each level. All bilinear filtering is done using Equations 5.7–5.12. See Figure 5.14.

© 2005–2012 Tomas Akenine-Möller

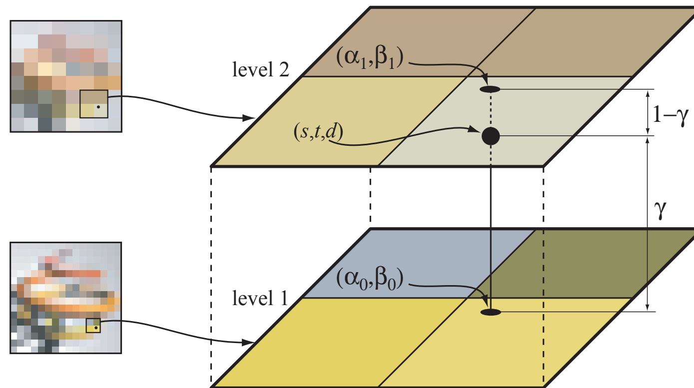

Figure 5.14: Illustration of trilinear filtering. Bilinear filtering is done in both the two mipmap levels. Using linear interpolation, those bilinearly filtered colors are blended to form the final color, $\mathbf{f}$, using γ.

The next step consists of linearly blending the colors $\mathbf{f}_0$ and $\mathbf{f}_1$, obtained from bilinear filtering in levels l_0 and l_1. This is done using a parameter, $\gamma \in [0, 1]$. According to the OpenGL specification, γ should be computed as:

$$\gamma = \text{frac}(d). \tag{5.22}$$

Note also that γ is illustrated in Figure 5.14. The final step is to compute the trilinearly interpolated color, $\mathbf{f}$:

$$\mathbf{f} = (1 - \gamma)\mathbf{f}_0 + \gamma \mathbf{f}_1. \tag{5.23}$$

To compute $\mathbf{f}_0$ and $\mathbf{f}_1$, four texel accesses per value were needed. Thus, $\mathbf{f}$ is computed using 8 texel accesses, that is in constant cost (in terms of texel accesses). Therefore, trilinear mipmapping is often referred to as a "constant time" filtering technique.

Recall that regardless of which formula is used to compute d, it is always of the form $d = \log_2 a$, where a is different for each technique. To compute the lower level, l_0 (see Equation 5.21), we need to evaluate:

$$l_0 = \lfloor d \rfloor = \lfloor \log_2 a \rfloor. \tag{5.24}$$

In a fixed-point representation of a, the level l_0 can be found very efficiently. It is simply a matter of locating the most significant bit that is set to one in a.

Example 5.4.1.1 Computation of l_0
For example, assume $a = 000101.011001_b$. As can be seen, the most significant bit is located in the integer part of a, and it is bit number two that is set to one (bit number 0 is immediately to the left of the decimal point). Therefore, $l_0 = 2$ since bit number two was the MSB and set to one. Converting the value of a to the decimal number system, we get $a = 5.390625$, and $\log_2 5.390625 = 2.43\ldots$, and thus $l_0 = 2$, as expected. □

© 2005–2012 Tomas Akenine-Möller

An approximation to estimating γ, instead of using Equation 5.22, is shown below [19]:

$$\gamma = \frac{a - 2^{l_0}}{2^{l_0}}. \tag{5.25}$$

Thus, a very simple method to compute a fixed-point representation of γ is at hand. The subtraction is implemented by zeroing out the most significant bit that is set to one, and then a right shift implements the division. However, note the γ here is not computed in the logarithmic domain as in Equation 5.22, and so will not provide us with the same results, but γ in Equation 5.25 is in the interval $[0, 1]$, and it is a monotonic, increasing function, and so can work quite well.

Trilinear mipmapping is done using bilinear interpolation in two levels, and then linear interpolation between these. Most APIs also support three related techniques that uses fewer texel accesses. These modes are obtained by simply replacing bilinear interpolation with nearest neighbor sampling, and/or linear interpolation among the two mipmap levels with nearest neighbor sampling as well. The first one uses only one mipmap level (the closest one), and then performs bilinear interpolation there. This means that only four texel accesses are needed. The second filter uses the two mipmap levels with linear filtering in between, but in each mipmap level, nearest neighbor sampling is used. Thus, only two texel accesses are needed. The third filter uses only one mipmap level and nearest neighbor sampling there. For all these three filters there will be noticeable artifacts as can be seen in Figure 5.15, and it gets even worse when animated. Hence, these are seldom used, unless quality can be sacrificed. In general it holds (at least for these filters), that the more texel accesses that are required, the better quality. Next, two techniques will be presented which saves some texel accesses, but with better quality than the three modes discussed here.

Brilinear mipmapping

NVIDIA's scheme. Add this later on. Not part of the course this year.

Bilinear-Average mipmapping

For mobile devices, it is of uttermost importance to preserve the usage of memory accesses, and therefore, a less expensive technique has been suggested [7]. This technique, called *bilinear-average mipmapping*, performs much better than that used in the top right corner of Figure 5.15, but not as well as trilinear mipmapping. The advantage of bilinear average mipmapping is that it only requires accesses to four texels.

Only the texels in level l_0 of the mipmap are accessed, and bilinear interpolation is performed there as usual. However, accesses to the level above (l_1) are avoided. Instead, it is noted that a texel in the level above usually is computed using some weighted average of the 2×2 texels below the texel in level l_1. In fact, quite often a 2×2-box filter is used, i.e., the exact average of the four

© 2005–2012 Tomas Akenine-Möller

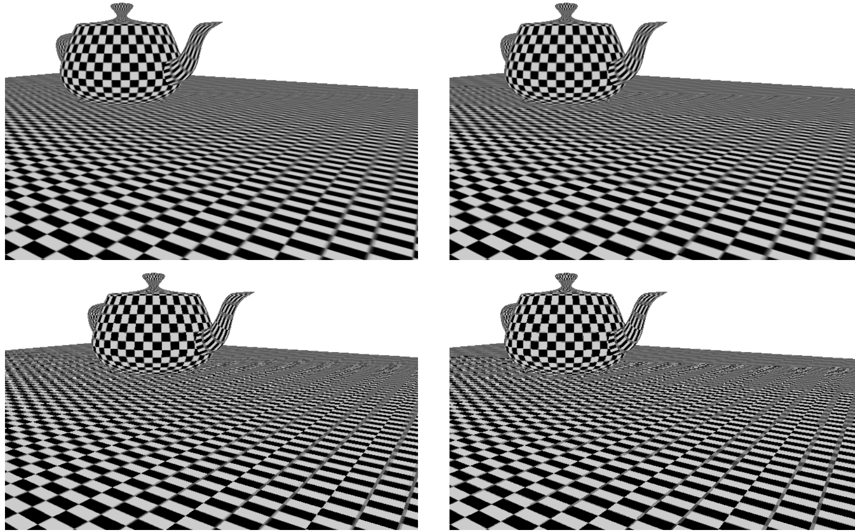

Figure 5.15: Different mipmapping filters. Top left: trilinear mipmapping. Top right: using a single mipmap level and bilinear filtering in that level. Bottom left: using two mipmap levels, but only nearest neighbor sampling in each. Bottom right: using one mipmap level, and nearest neighbor sampling there. In the top right image, it is quite clear from looking at the ground plane when the transitions from one mipmap level to another occurs.

texels is used, or a filter that gives a similar result. The idea of bilinear-average mipmapping is to compute a texel in level l_1 on-the-fly during filtering, from the 2×2 texels in level l_0. More specifically, the texels $\mathbf{t}_{00}$, $\mathbf{t}_{01}$, $\mathbf{t}_{10}$, and $\mathbf{t}_{11}$ are accessed in level l_0 (see Section 5.3.2 for an exact definition of the $\mathbf{t}_{ij}$). From these, a bilinearly filtered value, $\mathbf{f}_0$, is computed, and in addition, the average is computed as:

$$\mathbf{f}_1 = \frac{1}{4}(\mathbf{t}_{00} + \mathbf{t}_{01} + \mathbf{t}_{10} + \mathbf{t}_{11}). \tag{5.26}$$

Then interpolation is done with $\mathbf{f}_0$ and $\mathbf{f}_1$ using Equation 5.23 as in trilinear mipmapping. The result is that bilinear interpolation is done the the lower level, and a type of nearest neighbor sampling is done in the upper level. However, it is a bit better than nearest neighbor sampling, as illustrated in Figure 5.16.

5.4.2 Anisotropic Mipmapping

Texram [50] was first! Add this later on. Not part of the course this year.

© 2005–2012 Tomas Akenine-Möller

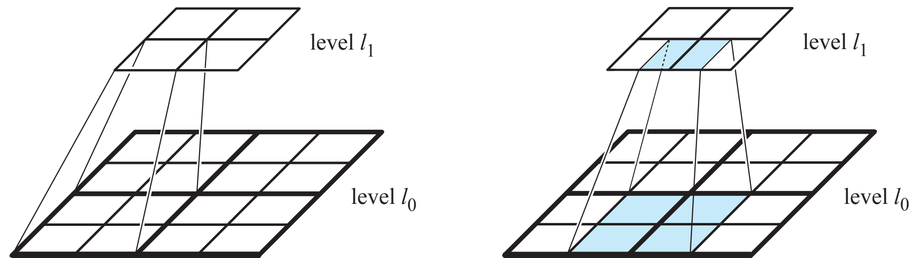

Figure 5.16: Left: two levels, l_0 and l_1, in a mipmap hierarchy. Only 4×4 texels are shown for level l_0. A texel in level l_1 is often computed as the average of the four texels in level l_0, i.e., "under" the texel itself. Right: the 2×2 gray texels are assumed to be accessed when bilinearly interpolating a color from the bottom level. Bilinear-average mipmapping uses the average of these gray texels as a texel from level l_1. Note, however, in this case, the average of the gray texel does not even exist in level l_1, and therefore, the technique is a bit better than using nearest neighbor sampling in level l_1. In addition, it does not access any texels in the upper level, and so is less expensive too.

5.5 General Caching

In computer system design, the memory system plays a very important role for performance. A memory system is often a hierarchy of different memory units, where the memory unit closest to the computational unit (e.g., GPU or CPU) using the memory is smallest and fastest. The memory unit "farthest" from the computational unit is largest and slowest. A *cache* is the first memory unit in a memory hierarchy, and it has thus the fastest access time and the smallest amount of memory. The underlying principle that makes caches work is *locality*.

The idea of locality can be understood by studying an efficient "over-the-counter" liquor store. In such a store, the most commonly used bottles are located immediately under the counter, closest to the sales person. The finest, most expensive wines are probably not being sold that often, and so they are located in the back of the store. Hence, in the majority of cases, the sales person finds what the buyer wants quickly (since it is located closely), and in some rare cases, the sales person need to go to the back of the store to get the fine wine. The organization of this imaginary store is based on locality.

For computer systems, caches also work under the assumption of locality. There are two major forms of locality in this context:

- *Spatial locality*: if a certain memory location is being accessed, it is likely that memory locations nearby will be accessed soon as well.
- *Temporal locality*: if a certain memory location has been accessed, it is also likely that the same location will be accessed again in the future.

Next, we will look into how a cache works and how these types of locality are being exploited.

© 2005–2012 Tomas Akenine-Möller

As mentioned, a cache is a small and fast memory. Common sizes may be 512 bytes up to hundreds of kilobytes, and the memory technology is quite often SRAM or simply a set of flipflops. When the CPU or GPU wants to access a particular memory address, a request is sent to the cache. If the content of that memory access is already in the cache, we have a *cache hit*, and the content can be delivered immediately. However, if it is not in the cache, we have a *cache miss*, and the cache must request the content from the slower main memory (or the next, lower level in the memory hierarchy), and the CPU or GPU must be delayed until the content is available in the cache.

To exploit spatial locality, a larger block of memory is fetched on a cache miss. A cache is divided into a number of *cache lines*, and a cache line is simply a number of words in the cache. For example, a 512 bytes cache can be divided into eight 64 bytes blocks. On a cache miss, a block of 64 bytes is therefore fetched from main memory. Due to spatial locality, memory locations nearby are likely to be accessed, and since some of them have already been fetched into the cache, the next access could perhaps be done directly from the cache. Furthermore, unless the content of this cache line has been replaced, temporal locality is being exploited the next time the same memory location is being accessed, because, the desired content will then (again) be in the cache and thus ready for immediate access.

A simple performance model for a cache is given below [29]:

$$t_{\text{avg}} = t_{\text{hit}} + r_{\text{miss}} \times t_{\text{miss}}, \tag{5.27}$$

where t_{avg} is the average number of clock cycles it takes to access memory, r_{miss} is the miss rate, which is simply a measure for how often a cache miss is obtained. t_{miss} is the time it takes to access main memory.

Example 5.5.0.1 Cache performance
A cache is often built so that a memory access in the cache can be done in a single clock cycle, and thus $t_{\text{hit}} = 1$. When we get a cache miss, assume that the cost of fetching a block (say 64 bytes) of memory into a cache line takes $t_{\text{miss}} = 40$ clock cycles (using for example DRAM memory). With a good cache and a well-behaved application, one might get a *miss rate* of $r_{\text{miss}} = 2.5\%$. With this data, the average time to access memory, becomes $t_{\text{avg}} = 1+0.025\times 40 = 2.0$ clock cycles. □

One goal of using caches is to make it appear as if all memory locations can be accessed with the same speed as provided by the fast (and expensive) cache memory, and the same time providing the user with lots of memory at a low cost. In Example 5.5.0.1 we saw that the cache was able to reduce the average access time to only two clock cycles, which is far better than the 40 clock cycles to access main memory directly without the cache.

Next, we will describe the cache in more detail. However, it should be noted that we do not cover anything about how memory writes are handled within a system with a cache. The reason for this is that graphics hardware most often only has read-only caches. For the interested reader, we refer to Hennessey and Pattersson's book [29].

© 2005–2012 Tomas Akenine-Möller

So far we have not covered in any detail how a cache really works. For example, where in the cache should a block of memory be placed? How can we test if a block is in the cache? Which block should be thrown out from the cache when a new block needs to come into the cache. Solutions will be presented in the next subsection.

5.5.1 Cache Mapping, Localization, and Replacement

We first explain different strategies for where to put a memory block in the cache. There are three different mappings that do this, and those are called *direct mapped*, *fully associative*, and *n-way set associative*. For this discussion, the address of a memory location is denoted A, and the number of cache lines in the cache is denoted $n_L = 2^{b_L}$. We also assume that the smallest accessible unit in a cache is a 32-bit word. Furthermore, each cache line holds $n_W = 2^{b_W}$ 32-bit words. Thus, the size of the cache is $s = n_L \times n_w$ 32-bit words.

Direct Mapped

This is the simplest type of mapping, and here, a block of memory is mapped to a unique position (cache line) in the cache. To oversimplify, the block is (often) put into the cache line described by $\texttt{mod}(A, n_L)$, where `mod` is integer modulo. It should be noted that the memory block of n_W words "surrounding" the address A is put into the cache, and not the block starting at address at A (unless they coincide).

A cache does not simply need storage for the actual cache lines, but it also needs to store a *tag* for each cache line. The purpose of the tag is to identify from where in the main memory the memory location comes from. For a direct mapped cache, an address, A, is divided into two parts as shown below:

$$A = [\texttt{TAG} \,|\, \texttt{lineIndex} \,|\, \texttt{blockOffset}], \qquad (5.28)$$

where `TAG` refers to the most significant bits of A, and `blockOffset` refers the least significant bits of A, and `lineIndex` to the bits in between.

The idea of the tag is to provide an identifier for all words in the same memory block. Thus given that $n_W = 2^{b_W}$ 32-bit words are stored in a cache line, the b_W least significant bits are only used to identify a particular 32-bit word in a cache line. Furthermore, two address bits are used for indicating which byte inside a 32-bit word is being referenced. The number of bits for `blockOffset` is thus $b_W + 2$. To check a particular cache line, b_L bits are needed for `lineIndex` since the number of cache lines is $n_L = 2^{b_L}$. For a 32-bit address, A, the number of bits for `TAG` it is $b_T = 32 - b_L - b_W - 2 = 30 - b_W - b_L$ bits.

In Figure 5.17, an illustration of a direct mapped cache can be seen.

Fully Associative

When a memory block can be placed anywhere in the cache it is called *fully associative*. In a direct mapped cache, there was only a single location, but

© 2005–2012 Tomas Akenine-Möller

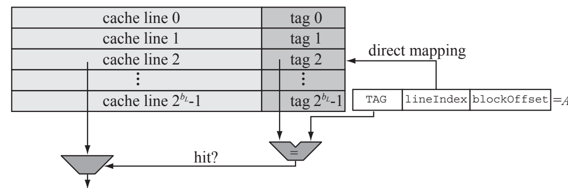

Figure 5.17: A direct mapped cache. The address A of a desired memory location is used to check whether the cache already holds the content of A. First, the b_L bits are used to directly map the a cache line, which in this example is cache line number 2. Cache line number 2's tag is compared for equality against the tag of A, and if those tags are equal we have a hit in the cache. Otherwise, the memory block containing A need to be fetched from main memory.

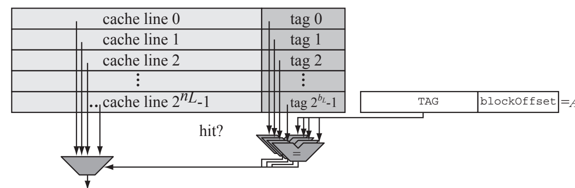

Figure 5.18: A fully associative cache. All tags in the cache are tested in parallel for equality with the tag of the address A. The result shows whether we have a cache hit, and in such a case, it also selects the corresponding cache line.

here, we have full degree of freedom to choose. However, that also means that the address of a memory location is described differently as shown below:

$$A = [\texttt{TAG} \,|\, \texttt{blockOffset}]. \tag{5.29}$$

As can be seen, the tag is now described with more bits than for a direct mapped cache. This is a consequence of the fact that we can place a memory block anywhere in the cache. In practice, all tags must be searched in order to test if a memory block is in the cache, as can be seen in Figure 5.18. This is done in parallel for efficiency reasons. However, that also means that this is an expensive type of cache because there are as many comparitors as lines in the cache, which can amount to a lot of physical hardware.

n-way Set Associative

This type of cache is somewhat in between direct mapped and a fully associative cache in the sense that there is a degree of freedom to place the memory block in a few places. Figure 5.19 shows an example with a set associative cache with "m" sets and 2 ways. This is also called a 2-way set associative cache. First the *set* that a memory block can go into is selected using `lineIndex`, the same as a direct mapped cache. Then one of any number of *ways* can be selected, the

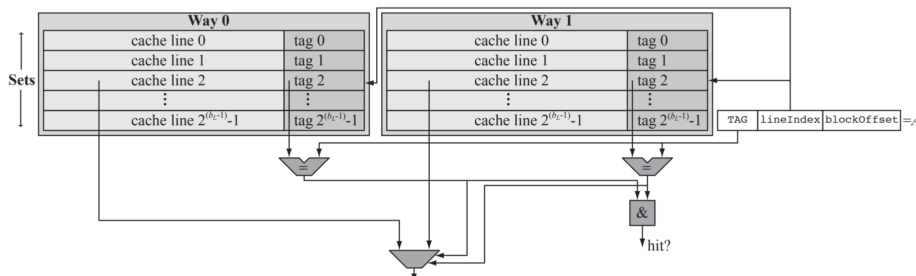

Figure 5.19: A 2-way set associative cache.

same as any location in a fully associative cache can be selected. The general idea is that we split the cache into m sets, and then split each set into n ways. The tags in these n cache lines are then searched in parallel to find whether we have a hit. For a set associative cache, with n ways, $n = 2^s$, the address A is split into the following pieces:

$$A = [\texttt{TAG} \,|\, \texttt{lineIndex} \,|\, \texttt{blockOffset}]. \tag{5.30}$$

`blockOffset` is exactly the same as for the other types of caches. However, `lineIndex` does not need b_L bits as in a direct mapped cache but rather only $b_L - s$ bits since there are $n = 2^s$ different ways to be searched. The `TAG` occupies the remaining bits of the address.

5.5.2 Replacement Strategy

For fully associative and set associative caches, there is a choice to be made when we get a miss in the cache, and need to fetch data from main memory. The choice is about which cache line to replace, and thus what data is to be thrown out of the cache. For a direct mapped cache, there is no such choice since an address always maps to exactly one cache line.

There are two major strategies for replacing (sometimes called "retiring") a cache line. These are *random* and *Least-Recently Used* (LRU). With a random replacement strategy a cache line is chosen in a pseudo-random way. This is the simplest of the two strategies. LRU on the other hand, attempts to make a smart choice by retiring the cache line that was not used for the longest period of time. If a block has not been used in a long time, it makes sense, due to the locality assumption, to throw out such blocks. A counter for each block is needed to implement LRU, which makes it more expensive. In addition, on replacement, one needs to find the block that is least-recently used, and that costs in hardware as well.

5.5.3 Cache Misses

To be able to analyze the performance and to discuss the characteristics of a cache, it is common to categorize the misses into three different types, namely,

© 2005–2012 Tomas Akenine-Möller

compuslory, *capacity*, and *conflict* misses. A compulsory miss is one that must always occur, and these happen when the data that is needed is not yet in the cache. Thus, these are often also called *cold start* misses. A *capacity* miss occurs because the size of the cache is too small. Finally, a *conflict* miss occurs because two blocks compete for the same cache line in the cache, and since one of them is thrown out from the cache, we get a conflict miss the next time the thrown block is needed.

A fully associative cache is good at avoiding conflict misses. To reduce capacity misses a larger cache is needed, and for cold start misses, one could use larger cache lines. Note that changing one parameter in the cache may affect more than one of these types of misses.

5.6 Texture Caching

See reading list on the course website.

5.7 Texture Compression

See reading list on website. Should be here eventually.

5.7.1 Background

5.7.2 S3TC/DXTC

5.7.3 PVR-TC

5.7.4 ETC1/ETC2

5.7.5 ASTC

5.7.6 Normal Map Compression

3Dc and our own algorithms.

© 2005–2012 Tomas Akenine-Möller

© 2005–2012 Tomas Akenine-Möller

Chapter 6

Culling Algorithms

In this chapter, we will describe some simple culling algorithms which are often used in graphics hardware. To *cull* means to *pick out*, and in graphics, we refer to algorithms that avoid unnecessary work when talking about culling algorithms. Here, we are only concerned with algorithms that do not affect the final rendered image, and therefore, "unnecessary work" is such that it does not influence the resulting image.

First, two simple algorithms are described (Section 6.1), which can avoid a large amount of depth buffer reads. Then, in Section 6.2, a technique is presented that performs culling at a higher level, which in this case means that an entire object can be culled even before sending it to the GPU. Finally, in Section 6.3, a delay stream architecture is presented.

6.1 Z-min and Z-max Culling

From the rasterization equation (4.4), we know that the memory bandwidth related to the depth buffer, B_d, is:

$$B_d = \underbrace{d \times Z_r}_{\text{reads}} + \underbrace{o \times Z_w}_{\text{writes}}, \tag{6.1}$$

where d is the depth complexity, o is the overdraw, and Z_r & Z_w are the cost of depth buffer reads and writes, respectively. It is known that $o \leq d$ (see Equation 4.1). For example, when $d = 4$, the overdraw is $o \approx 2$. Furthermore, since Z_r and Z_w are equally large, the term $d \times Z_r$ is larger or equal to $o \times Z_w$. The kind of algorithms described in this section, therefore aims at reducing the cost of $d \times Z_r$.

A great advantage of these algorithms is that they work without any kind of user intervention. The hardware takes care of it all. Furthermore, these two algorithms are surprisingly simple. The idea of both Z-max and Z-min culling is to perform some kind of culling on a per tile basis. Hence, tiled

traversal (Section 2.2.4) must be used.[1] Assume that tiles of size $w \times h$ pixels are used, and that the depth values of a particular tile are denoted $d(i, j)$, with $i \in [0, w-1]$ and $j \in [0, h-1]$. Next, we describe Z-max and Z-min culling.

6.1.1 Z-max Culling

This type of algorithm stems back to the hierarchical Z-buffer by Greene et al. [22], where a pyramid of depth values is maintained and culled against. However, this algorithm has not found its way into graphics hardware, probably due to the complexity of updating the depth pyramid. Instead, a simpler variant [42] has been developed, and will be described here. This algorithm is referred to as HyperZ by ATI, and NVIDIA has its own variant of it.

Assume we are currently rendering a triangle. The basic idea here is to avoid depth buffer reads in tiles covered by the triangle, where the triangle is definitely behind the contents of the depth buffer in that tile. To that end, each tile stores a value called, $z_{\max}$, which is defined as:

$$z_{\max} = \max_{ij}[d(i, j)], \tag{6.2}$$

that is, $z_{\max}$ holds the maximum of all the depth values in a tile. Since depth values, $d(i, j)$, are at least 16 bits, and often 24 or 32 bits, the $z_{\max}$ could have the same resolution. However, a conservative value can work almost as well, albeit never fully as well. In this case, it must be guaranteed that the $z_{\max}$-value of a tile is larger than (or equal to) $\max_{ij}[d(i, j)]$, which is the optimal value. If that holds, then the $z_{\max}$-values can be stored in, for example, only 8 bits per tile. However, note that there is a tradeoff in culling performance vs. number of bits per $z_{\max}$.

The actual culling works as follows. When the traversal algorithm arrives at a tile that overlaps with the triangle being rendered, a value called $z_{\min}^{\text{tri}}$ is computed. The goal of this is to compute a conservative value of the minimum of the depth values of the triangle in that tile. This can be done in a number of ways that will be discussed momentarily. First, we describe the important operation of Z-max culling. If the following expression is true:

$$z_{\min}^{\text{tri}} > z_{\max}, \tag{6.3}$$

then the triangle in that tile is guaranteed to be hidden by the contents of the depth buffer, i.e., hidden by primitives (e.g., triangles) that have already been rendered. When the triangle is hidden, we do not need to read the depth buffer, and this is the gain that this algorithm provides. Note that this algorithm does not have any advantage in terms of texture accesses,[2] color buffer reads/writes,

[1] An exception is the scanline-based zigzag traversal described by Akenine-Möller and Ström [7], where a Z-min culling algorithm is used, and adapted to scanline traversal. This is done using a small tile row buffer.

[2] This may not be true for all architectures. Especially high-end graphics hardware executes the pixel shader (and thus also texture accesses) before the per-pixel depth test. This means that such an architecture would also reduce the number of texture accesses and pixel shader executions with Z-max culling.

© 2005–2012 Tomas Akenine-Möller

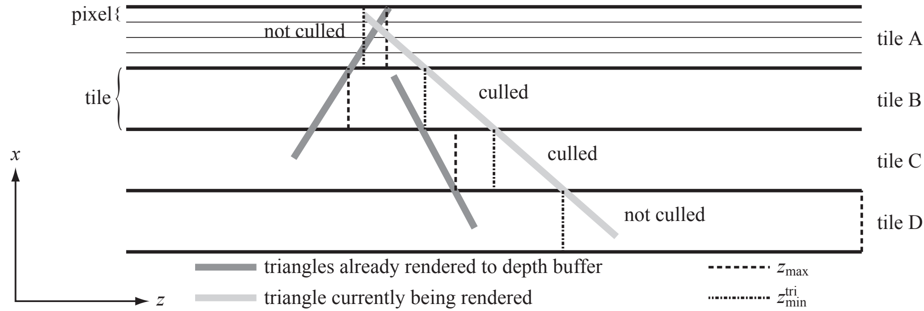

Figure 6.1: Z-max culling illustrated in two dimensions in post-perspective space. In tiles where $z^{\text{tri}}_{\min} > z_{\max}$ depth buffer reads can be avoided since it is guaranteed that the triangle being rendered is hidden with respect to the contents of the depth buffer. In this example, tile B and tile C can be culled. However, for tile A and D, the light gray triangle must be rendered as usual.

nor depth writes, because these operations would be avoided anyway due to that the depth test would fail. Z-max culling is also illustrated in Figure 6.1.

There are several different ways to compute conservative estimates of $z^{\text{tri}}_{\min}$. Recall that the depth values at the vertices are denoted d^k, $k \in [0, 1, 2]$ (not to confuse with the pixels' depths, called $d(i, j)$). A really simple way to compute $z^{\text{tri}}_{\min}$ is shown below:

$$z^{\text{tri}}_{\min} = \min_{k \in [0,1,2]} (d^k). \tag{6.4}$$

The advantage of this is that it is simple, and it can even be done in the triangle setup, since the evaluation of Equation 6.4 is constant over the entire triangle. However, this value can be overly conservative, and thus perform poorly. For example, in Figure 6.1, all tiles would get the same $z^{\text{tri}}_{\min}$-value, which would be equal to the value shown in tile A. Tile B would still be culled, but tile C would not be culled any longer, since the $z^{\text{tri}}_{\min}$-value is less than the tile's $z_{\max}$-value.

A better way is to compute the depth values, called c_i, $i \in [0, 1, 2, 3]$, at the corners of the tile, and then compute:

$$z^{\text{tri}}_{\min} = \min_{k \in [0,1,2,3]} (c_i). \tag{6.5}$$

If the entire tile is covered by the triangle, this gives the optimal value of $z^{\text{tri}}_{\min}$. However, when a tile is only partly covered by a triangle, this may not longer be the case. The reason is that the minimum must occur at one of the vertices of the polygon that results when the triangle is clipped against the tile borders. When the triangle overlaps the tile entirely, the resulting clipped polygon is a rectangle of the same size as the tile itself, and therefore Equation 6.5 is optimal. An extreme example of when this is not true is when all the triangle vertices lies inside a tile. Assuming the triangle's normal is not parallel to the viewing direction, it is apparent that Equation 6.5 produces overly conservative values.

© 2005–2012 Tomas Akenine-Möller

The optimal solution would be to clip the triangle against the polygon, compute the depth at all the vertices of the resulting polygon, and choose the minimum of these depths. Clipping is already considered an expensive operation when the actual triangle is clipped against the screen borders and near & far plane, so clipping the same triangle over and over again against all the tiles that are being overlapped by the triangle would be extremely expensive.

A hybrid method is to compute the maximum of Equation 6.4 and Equation 6.5:

$$z_{\min}^{\text{tri}} = \max\left[\min_{k\in[0,1,2]}(d^k), \min_{k\in[0,1,2,3]}(c_i)\right]. \tag{6.6}$$

This technique will avoid the example above when the triangle's vertices were all inside a single tile.

Assuming the screen resolution is $s_w \times s_h$ and the tile size is $w \times h$. There are $\lceil s_w/w \rceil \times \lceil s_h/h \rceil = t_w \times t_h$ tiles for the entire screen. Each tile needs to store its $z_{\max}$-value. One solution is simply to store them in on-chip memory directly in the GPU. Another is to access these values through a reasonably large cache. The first solution always deliver the $z_{\max}$ at constant time, but for large screen resolution it can be expensive. The second solution does not always get cache hits, and therefore, this solution could incur extra delays. However, it scales better with larger screen resolutions.

Now, consider a single tile and its $z_{\max}$-value. Under normal circumstances[3] the $z_{\max}$-value is a decreasing monotonic function, which means that it will get smaller and smaller the more triangles that are being rendered, or stay the same. Recall also that it suffices to have a conservative estimate of the optimal $z_{\max}$ (Equation 6.2). When fragments pass the depth test and are written to the depth buffer, the optimal $z_{\max}$-value may take on a new value. To update $z_{\max}$ one could read all depth values of the tile, and simply evaluate Equation 6.2. When depth buffer compression and depth caches are used (see Section 7.2), this is what is usually done due to that all depth values of a tile are readily available.

When this is not the case, reading all the depth values could defeat the purpose of this algorithm. Since $z_{\max}$ can be conservative, one solution could be to only update a tile's $z_{\max}$-value when, say, n different triangles have written to that tile. This is not a strategy that has been documented in the literature, so it is not guaranteed that it will work. However, a rather small value of n, perhaps $n = 4$, could provide a good trade-off. Note that this surely is scene-dependent though.

Normally, when a triangle is hidden, we avoid writing to the color and depth buffers, and texture accesses are also avoided. However, as have been shown in this section, with Z-max culling, the depth buffer reads can also be avoided when the triangle is definitely hidden by the contents of the depth buffer. This is the strength of this algorithm. However, it is interesting to investigate when the algorithm starts to pay off, and this can be done by rewriting Equation 6.1

[3]That is, when using "less than" or "less or equal" as depth test function.

© 2005–2012 Tomas Akenine-Möller

as shown below.

$$B_d = d \times Z_r + o \times Z_w = \underbrace{(d-o) \times Z_r}_{\text{fragments that only read}} + \underbrace{o \times Z_r + o \times Z_w}_{\text{fragments that read and write}} . \quad (6.7)$$

The "fragments that only read" are the fragments that fails the depth test, which means that they are occluded, i.e., hidden by the contents of the depth buffer. These are the kind of fragments that Z-max culling reduces work for. As can be seen in the Equation, there will be $d-o$ such fragments, and o fragments that both read and write. In terms of number of fragments, there will be more fragments that fail than pass when the following is fulfilled:

$$d - o > o \quad \Leftrightarrow \quad d > 2o. \quad (6.8)$$

According to Equation 4.1, $o \approx 2$ when $d = 4$, so there is a break-even point around $d = 4$, and for larger d there are more fragments that fail. What this means, however, is that a reasonably high depth complexity is needed before Z-max culling start to really work. Put another way, it takes a while to build up some occlusion power of a tile that can actually cull something. This lead researchers [7] to investigate whether a different algorithm could be use before the occlusion power has been built up. This is the topic of the following section.

Morein [42] reported that about 20–50% of the pixels failed the depth test in applications such as Quake, 3D Mark, and 3D Winbench. So, ideally all those pixels could be culled by Z-max culling. However, due to that tiles must be larger than pixels, and due to that the $z^{\text{tri}}_{\text{min}}$-value is conservative, only about 45–90% of the 20–50% could be avoided. In practice, this meant that 10–32% of the depth buffer reads were avoided.

6.1.2 Z-min Culling

The Z-min culling algorithm [7] is very similar to Z-max culling. An interesting feature of this couple, is that the algorithms are not competing, but they are rather complementary. Thus, if both are implemented, then performance will be at least as good, and often much better, as if only one of the algorithms exists in the hardware.

In Z-min culling, each tile stores a value called, z_{min}, which is defined as:

$$z_{\text{min}} = \min_{ij}[d(i,j)], \quad (6.9)$$

that is, z_{min} holds the minimum of all the depth values in a tile. Similar to Z-max culling, the z_{min}-value can be a conservative estimate of Equation 6.9. The conservative estimate of z_{min} must be less than or equal to that in Equation 6.9. Again, there is a tradeoff between the number of bits per z_{min}-value and performance.

The actual Z-min culling works as follows. When the traversal algorithm arrives at a tile that overlaps with the triangle being rendered, a value called $z^{\text{tri}}_{\text{max}}$ is calculated. The goal of this is to compute a conservative value of the

© 2005–2012 Tomas Akenine-Möller

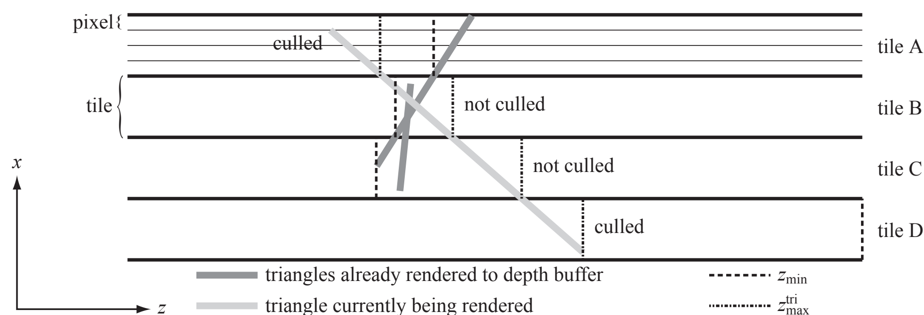

Figure 6.2: Z-min culling illustrated in two dimensions. In tiles where $z_{\max}^{\text{tri}} < z_{\min}$, depth buffer reads can be avoided since it is guaranteed that the triangle being rendered is definitely in front of primitives already rendered to the depth buffer. In this example, tile A and tile D can be culled. However, for tile B and C, the light gray triangle must be rendered as usual.

maximum of the depth values of the triangle in that tile. There are several ways to do this. Either of Equations 6.4, 6.5, or 6.6 work when the min-operator is replaced with max, and vice versa.

The major operation of Z-min culling takes place when the following expression is true:

$$z_{\max}^{\text{tri}} < z_{\min}, \tag{6.10}$$

because then the triangle in that tile is guaranteed to be completely in front of the contents of the depth buffer, i.e., in front of primitives (e.g., triangles) that have already been rendered. When this happens, it is certain that the depth test passes, and the depth buffer reads can be avoided. This is where the performance gain comes from in Z-min culling. In Figure 6.2, Z-min culling is illustrated.

Exactly as in Z-max culling, the $z_{\min}$-values of all the tiles on the screen need to be stored somewhere. The solutions for $z_{\max}$ works as well here, so either onchip memory is used for all tiles' $z_{\min}$-values, or they are accessed through a cache.

A great advantage of Z-min culling is that it is extremely simple to maintain an exact[4] $z_{\min}$-value for each tile. The reason for this is that when we know that a depth value, d, has been written inside the tile and $d < z_{\min}$, then the minimum depth value must be equal to d, i.e., $z_{\min} = d$. Such a simple update is not possible in Z-max culling because there may be several pixels with the depth being equal to $z_{\max}$, and all of them must be overwritten in order to change the value of $z_{\max}$. Even if that did happen, we would not know what the new $z_{\max}$-value would be, since the maximum depth value of the tile may reside in a pixel not overwritten just now. The conclusion is that Z-max culling must read all depth values in the tile, and compute the maximum. Since updating

[4]With respect to the number of bits that $z_{\min}$ is stored in.

© 2005–2012 Tomas Akenine-Möller

$z_{\min}$ is so simple, it is perfectly suited for mobile GPUs.

In Equation 6.7, it can be seen that o fragments can potentially pass the depth test, and for all of these, Z-min culling can provide a performance improvement by avoiding the depth buffer reads. Interestingly, Z-min culling starts to pay off immediately, and then works worse and worse, and when $d > 4$, Z-max culling starts to pay off more. Note though, that since $z_{\min}$ for a tile is set to 1.0 (assuming this is the maximum depth value) when the depth buffer is cleared, Z-min culling will improve performance from the first polygon that is being rendered (if it does not have all depth values set to 1.0).

For scenes with low depth complexity (d=0.65, 1.5, 2.5), Z-min culling has been found to reduce depth buffer reads with 84%, 49%, and 69% [7]. When depth complexity grows it makes more sense to implement Z-max culling, and for best performance both algorithms should be implemented.

6.2 Object Culling with Occlusion Queries

This is a also a very simple algorithm that needs extra support in the graphics hardware, and in addition, the programmer needs to use this capability in a clever way to actually gain performance. Unlike, Z-min and Z-max culling (Section 6.1), *object culling* can actually cull away an entire object at a time. Z-min and Z-max culling can only reduce the number of depth buffer reads. Object culling can even reduce the amount of data sent to the GPU, the number of vertices processed by the vertex shader unit, and of course also, fragment processing.

The extra hardware that is needed is basically just a couple of counters and some control logic. When a set of primitives (e.g., triangles) are rendered, one can issue an *occlusion query*, which counts the number of fragments that passes the depth test. If no fragments passes, then all primitives are occluded, i.e., hidden by objects already rendered into the depth buffer.

For occlusion culling of an entire object, say a character running around in a game, the bounding box of the object is drawn using an occlusion query. During this rendering, only depth testing is enabled—depth writes, color writes, texturing, and pixels shaders are typically disabled. If the entire bounding box is occluded, i.e., no fragments pass the depth test, then the entire object is also occluded, and hence the object need no further processing. Note that it is assumed that the object is much more complex than the bounding box of the object, otherwise, there is little to gain from this. These types of occlusion culling algorithms also usually benefit from drawing the objects in a (rough) front to back order.

In practice, there must be a possibility to handle several occlusion queries in the pipeline at the same time. This is due to the long turnaround time to get the result of the query. NVIDIA and others have hardware for this. This can also mean that a sophisticated algorithm is needed to run on the CPU in order to be able to use occlusion queries on a hierarchical spatial data structure, such as a bounding box hierarchy. See Wimmer and Bittner's article on this

© 2005–2012 Tomas Akenine-Möller

topic [57].

It should also be pointed out that occlusion queries are not only used for culling algorithms, but can be used whenever fragment counting is needed. For example, when implementing a ray tracer on a GPU [49], one needs to know many pixels that still need to execute the grid traversal, or if all pixels have found a grid cell with triangles in it.

6.3 Delay Streams

The student should read the paper by Aila et al. [2]. Another example that uses delay streams is the hierarchical shadow volume algorithm [1].

© 2005–2012 Tomas Akenine-Möller

Chapter 7

Buffer Compression

So far, we have seen that texture caching & texture compression (Chapter 5), and different culling techniques (Chapter 6) can be used to reduce the memory bandwidth usage in graphics hardware for a rasterizer. In this chapter, different algorithms for compressing and decompressing parts of the frame buffer will be explored. These techniques compress data and store it in a compressed form in external memory. When it need to be accessed, the compressed information is sent over the bus, and thus bandwidth is saved, and when it reaches the GPU, it is decompressed and stored temporarily in a cache. When it need to be evicted from the cache, it is compressed on-the-fly and sent back to external memory. Note that a particular part of the buffer can be compressed and decompressed several times when rendering a scene. First, a particular part of the buffer can be accessed, compressed, and sent back to the external buffer. Then at a later time, another triangle needs to access that part of the buffer again, and so the information is again read, decompressed, and stored in the cache. At an even later stage, the information can again be compressed and sent back to the external memory, and so on.

This chapter starts with a description of a general system that can be used with any type of lossless compression algorithm and with any part of the frame buffer. Then follows, in Section 7.2, different techniques for compressing and decompressing the depth buffer. Finally, in Section 7.3, some techniques for color buffer compression/decompression will be presented.

It should be pointed out that the description of such techniques are, surprisingly, non-existing in the academic literature. The only material found is by Morein, who discuss depth buffer compression in only four slides [42]. Therefore, the presentation here is based on several patents[1] by ATI and NVIDIA.

[1] The patents are: **I**: "Method and Apparatus for Compressing Parameter Values for Pixels in a Display Frame", by DeRoo et al., US Patent 6,476,811, November 2002, **II**: "Method and Apparatus for Compression and Decompression of Z Data" by van Hook, US Patent 6,630,933, October 2003, **III**: "Method and Apparatus for Controlling Compressed Z Information in a Video Graphics System" by Morein et al., US Patent 6,636,226, October 2003, **IV**: "System, Method, and Apparatus for Compression of Video Data using Offset Values" by Morein et al.,

7.1 Compression System with Cache

Both for depth compression and color compression, it is very important that the compression is lossless, i.e., no information can be lost during compression. This is in contrast to, for example, JPEG whose most common mode[2] is "lossy," which means that information may be lost when an image is compressed. With any lossy technique, the result will differ from using the original data (or a non-lossy technique), and for the depth buffer and color buffer, the results would not be acceptable.

For all algorithms described in this chapter, it is assumed that compression and decompression is done on a per tile basis (see Section 6.1 for the definition of a tile), and each tile is $w \times h$ pixels, and there are $t_w \times t_h$ tiles on screen.

The idea of this type of compression system is that some part of the rasterizer needs to access a buffer, and when possible, this information should be stored in compressed form, and hence bandwidth will be saved both to and from the external memory. Therefore, real-time compression and decompression units are needed. The basic components of a compression system in general are shown in Figure 7.1.

Two core components in this architecture is a cache (see Section 5.5) and a *control block*. A cache is needed because a full tile of pixels is compressed and compressed at a time, and therefore, it is more efficient to store the tiles that are accessed in a cache, since memory accesses to adjacent pixels sometimes are immediately available. In fact, without the cache it is very likely that this system would actually degrade performance. The control block receives read and write requests from the rasterizer, determines whether the information is in the cache, or sees to it that the information is loaded into the cache. When read and writes to the buffer in external memory are issued, the *decompression unit* unpacks the compressed information from a tile, and the *compression unit* attempts to compress the data, and sends it back to the external buffer.

The *tile table* is also central to this buffer compression system, and for each tile, it stores a number of bits of extra information. We call this information *tile info*. The usage of these will be explained here, but also in the following sections. When lossless compression is supported, there always needs to be a fall back that performs no compression at all. Otherwise, lossless compression cannot be guaranteed. These tile info bits can encode which type of compression mode is used for each block. This simplest example is to use a single bit for this, where 0 means uncompressed, and 1 means some compressed format. Another example assumes that two bits are used. The code 00_b could mean that the block is *not* compressed, 01_b could mean a mode with a 25% compression, 10_b could mean another mode with 50% compression, and finally, 11_b could mean that the block is cleared. This last mode is often used for fast buffer clears. So, to clear the buffer, one simply sets all tile info values to 11_b. When a read request is obtained, the tile info is first checked, and if it is equal to 11_b (i.e.,

US Patent 6,762,758, July 2004, and **V**: "System and Method for Real-Time Compression of Pixel Colors" by Molnar et al., US Patent 6,825,847, 2004.

[2]There is a mode in JPEG that is lossless too.

© 2005–2012 Tomas Akenine-Möller

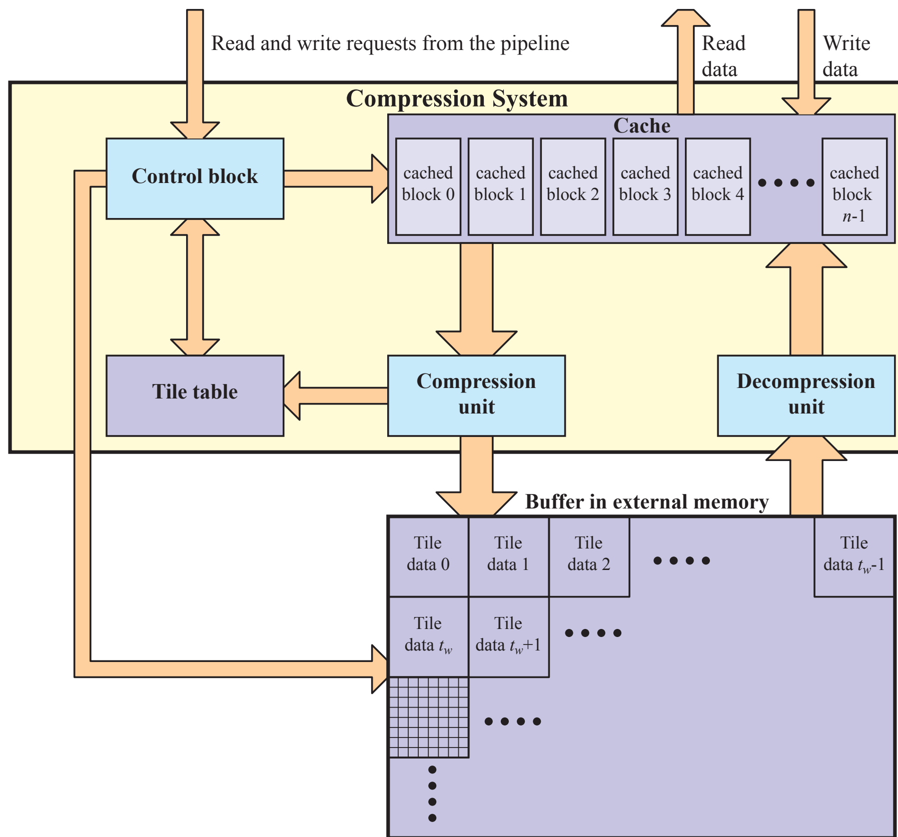

Figure 7.1: Compression system with cache. The buffer itself is located at the bottom, and it is divided into tiles, of say, 8×8 pixels. Read and write requests are coming from the rasterizer, and the cache delivers the information at the required locations, or updates the contents in the cache. When the cache needs to read from external memory, the control block first checks the tile table to see how much needs to be read from the memory, then reads, and finally decompresses. When the cache need to evict data from the cache, the compressor attempts to compress, and sends it back to the external memory in that form.

cleared), one need not read the buffer at all. Instead, the cache can immediately deliver a tile full of cleared values.

Now, assume that a tile stores four bytes per pixel, and that the tile is 8×8 pixels. This means that $8 \times 8 \times 4 = 256$ bytes are needed to store the information of a tile in uncompressed form. If there are $t_w \times t_h$ tiles, the external buffer still needs to store $256 \times t_w \times t_h$ bytes of information. This compression system does not make better use of external memory—it only saves bandwidth to and from

© 2005–2012 Tomas Akenine-Möller

the external memory.

With this information, imagine that a particular pixel in a particular tile need to be accessed, and a read request is issued, and that issue arrives to the control block. The control block determines whether that pixel's tile is in the cache, and if it is, then the information is immediately delivered to the issuing party. If it is not, some block is evicted from the cache, and the contents need to be sent back to the buffer in external memory. Therefore, an attempt is made at compressing the contents of that tile. This takes place in the compression unit. Say, the block could be compressed to 25%, i.e., 256 bytes is encoded using only 64 bytes. These 64 bytes are then sent to the external memory, which means that the other 192 bytes of that tile is not used. Since the tile was written back to external memory, the tile info is updated (in this case with 01).

Depending on the architecture, a new tile could be read in parallel or after the evicted block has been written back to memory. When reading from the external memory, the tile info in the tile table is first examined, to check to which compression mode is associated with that tile. Assume the current tile is compressed to 50%. This means that 128 bytes are read from external memory, and these bytes are then decompressed and stored in the cache. It should also be noted that the amount of memory for storing the tile table might be quite large, and therefore, its information might have to be accessed through a cache as well.

For efficiency reasons, it is also common to let each cache block store a *dirty bit*. The dirty bit indicates whether new information has been written to that tile. If not new info has been written to it, one need not write back that information to external memory when the tile is evicted. So, as soon as a single pixel has been updated in a tile, one need to set that tile's dirty bit in the cache.

Examples of buffers that can be compressed are the depth buffer (Section 7.2), the color buffer (Section 7.3), and even the stencil buffer.

7.2 Depth Buffer Compression

The depth buffer is probably the easiest buffer to compress, and it is also where one can make the biggest savings. The reasons why compression is likely to work well for the depth buffer are that depth over a triangle is interpolated linearly (see Equation 3.14), and that triangles tend to cover several pixels, or even entire tiles of pixels, or in case of really small (sub-pixel) triangles, the triangles often have a limited depth range anyway.

Notice that a depth cache (see Figure 7.1) is imperative to Z-max culling (Section 6.1.1), and since the system in Figure 7.1 also can provide compression, it is common to provide both Z-max culling and depth compression at the same time. Also, note that some depth compression schemes are likely to fail at compressing a tile if the geometrical complexity is high. However, the depth cache can still provide a significant gain since many triangles are written to that tile. For tiles with low geometrical complexity, depth buffer compression is likely to perform very well. Another advantage of depth compression with a

© 2005–2012 Tomas Akenine-Möller

cache is that semi-transparent geometry, such as particle systems etc, seldom write their depths to the depth buffer—those algorithms only test against the depth buffer. If the depths of those tiles are already in the depth cache, then the cost will be equal to zero in terms of depth buffer accesses.

An observation that appears to be used when designing compression schemes for the depth buffer is that when a triangle partially covers a tile, one can consider the tile to consist of two depth layers. Assume, for example, that a ground floor quadrilateral has been rendered, and that it covers a particular tile fully. Then an object is rendered, which makes half of that tile be overwritten by that object's triangles. The pixels in that tile that belongs to the ground floor all have similar depth values, and the other half also have similar depth values. Hence, depth compression schemes often try to take this fact into account.

Recall that, as in Section 6.1, the depth values in a tile are denoted $d(i, j)$, where $i \in [0, w-1]$ and $j \in [0, h-1]$. We also assume that the smallest possible depth value is $00\dots00_b$ (all bits are zero) and that the largest possible depth value is $11\dots11_b$ (all bits are one), where each of these numbers are, for example, 24 bits. Hence, the depths can be treated as integers. It should also be pointed out that the granularity of the memory accesses is not taken into account in the examples. Instead, the focus is on providing figures for the different algorithms that can be compared. If all memory accesses need to be, say, 128 bits, then that fact need to be taken into account as well. Next, some techniques for depth buffer compression are presented.

7.2.1 Depth Offset Compression

This compression scheme identifies a number of reference depth values, r_k, for a tile, and then iterates over all depth values in the tile. A test is made whether a depth value, $d(ij)$ is within a predetermined range, t, of r_k, and if so, that depth value can be coded as an *offset* to that reference value. Identifying more than two good reference depth values, r_k, appears quite difficult, at least if some kind of optimality is desired. Instead a simpler approach is often taken.

The idea is to compute the minimum depth value, $z_{\min}$, and the maximum depth value, $z_{\max}$, of a tile. This is done using Equations 6.2 and 6.9. Then it is assumed that there are two layers, one that is close to $z_{\max}$ (and less than or equal to $z_{\max}$), and another that is close to $z_{\min}$ (and larger than or equal to $z_{\min}$). To reduce the hardware complexity, the predetermined range, t, is often chosen as a power of two, i.e., $t = 2^p$. If the following is fulfilled:

$$d(i, j) - z_{\min} < t, \tag{7.1}$$

then $d(i, j)$ can be encoded as an offset relative to $z_{\min}$, and if the next expression is fulfilled:

$$z_{\max} - d(i, j) < t, \tag{7.2}$$

then $d(i, j)$ can be encoded relatively to $z_{\max}$. If at least one of Equation 7.1 or Equation 7.2 is fulfilled for every pixel inside a tile, then that tile can be compressed using this technique. Otherwise, it is stored in uncompressed form. In

© 2005–2012 Tomas Akenine-Möller

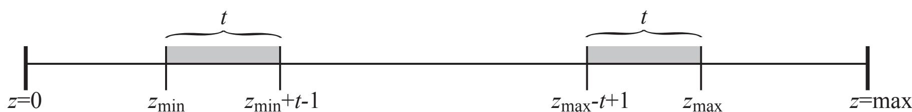

Figure 7.2: Offset depth buffer compression. When all depths, $d(i, j)$, of a tile are in either of the gray depth ranges, the tile can be compressed. Otherwise, the depth values of that tile are stored in uncompressed form.

Figure 7.2, the cases that can be compressed with this algorithm are illustrated.

If the tile can be compressed with this technique, the following information should be stored in order to be able to uncompress it without losses. The z_{min}- and z_{max}-values, must be stored, since these are the reference values, which we compress relatively to. Notice if Z-max and/or Z-min culling is combined with this depth buffer compression scheme, then sharing of computations and storage should be possible. Furthermore, each pixel needs to store one bit, which says whether its depth value is encoded relative to z_{min} or z_{max}, and each pixel also need to store its offset, $o(i, j)$, which is simply:

$$o(i,j) = \begin{cases} d(i,j) - z_{\text{min}}, & \text{if } d(i,j) - z_{\text{min}} < t, \\ z_{\text{max}} - d(i,j), & \text{if } z_{\text{max}} - d(i,j) < t. \end{cases} \tag{7.3}$$

The following example shows what kind of compression ratio can be expected from using depth offset compression.

Example 7.2.1.1 Depth Offset Compression
In this example, we assume that 8×8 tiles are used, and that each depth value is stored in 24 bits. Therefore, z_{min} and z_{max} need 6 bytes to store, and each pixel need one bit for indicating which reference value it is encoded to. Hence, another 8×8 bits $= 8$ bytes are needed. Finally, we assume that $t = 2^8$, and therefore, each offset value, $o(i, j)$, is encoded using 8 bits. This costs $8 \times 8 \times \times 8 = 512$ bits $= 64$ bytes. In total, this costs $64 + 8 + 6 = 78$ bytes. In uncompressed form, a tile occupies $8 \times 8 \times 3 = 192$ bytes. Thus, the compression reduces the data to $100 \times 78/192 \approx 41\%$, when the tile depths can be compressed. □

For the algorithm described above, a subtraction need to be performed both for comparing a depth value to z_{min} and to z_{max}. Next, a technique will be described that can reduce the computational complexity of compression and decompression. Instead of storing all bits of z_{min} and z_{max}, only the m most significant bits (MSBs) are stored as reference values. Call these values u_{min} and u_{max}. If z_{min} and z_{max} each is stored using k bits (e.g., 24 or 32 bits), the offset values are simply the $k - m$ least significant bits (LSBs) of the depth values. This value also determines the range, $t = 2^p$, i.e., $p = k - m$. The following example attempts to clarify this.

Example 7.2.1.2 Inexpensive Offset Computation
For simplicity, assume that $z_{\text{min}} = 010111001110_b$, and that the eight MSBs are

© 2005–2012 Tomas Akenine-Möller

Figure 7.3: A problem with the inexpensive variant of depth offset compression is illustrated here. Since $z_{\min}$ is truncated to u_{min} and $z_{\max}$ is truncated to u_{max}, the full range of $t = 2^p$ cannot be exploited for encoding offsets. In this figure, only the dark gray regions can actually be used. With the more expensive technique, all values in the light and dark gray regions could be encoded with offsets.

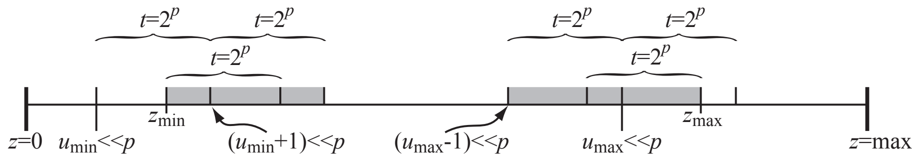

Figure 7.4: A solution to the inexpensive variant of depth offset compression. At the cost of one extra bit per offset, the depth range is increased by another range of $t = 2^p$, and thus all values in the gray areas can be encoded using this technique.

used as a reference value. Thus the reference value becomes $u_{min} = 01011100_b$, and the offset for encoding $z_{\min}$ is thus 1110_b, i.e., the remaining four bits. A major disadvantage with this scheme can be seen in this example. Since we use four bits for the offset, the offset should be able to encode offsets in $[0, 15]$. However, with this simplified scheme, the offset must be larger or equal to 1110_b, but at the same time it cannot be bigger than 1111_b. Thus, the depth range that can be used is very small, only two values. □

This loss of compressibility, as exemplified above, is illustrated in Figure 7.3. The problem arises since, the $z_{\min}$ and $z_{\max}$ are truncated, and hence, depending on the least significant bits of these values, there will be a different amount of possible values to encode relative $z_{\min}$ and $z_{\max}$. In the worst case, only a single value can be encoded relative to, say, $z_{\min}$, and this value must be equal to $z_{\min}$.

One solution is simply to increase the number of bits per offset. With just one more bit per offset, the range increases by a factor $t = 2^p$. This is illustrated in Figure 7.4. An example that uses this technique follows.

Example 7.2.1.3 Depth Offset Compression: inexpensive variant
Again, assume that 8×8 tiles are used, and that each depth value is stored in 24 bits. Furthermore, we assume that the offsets are encoded using 8 bits, which means that the truncated versions of $z_{\min}$ and $z_{\max}$ only need 16 bits=2 bytes each, i.e., 4 bytes in total. Each pixel needs two bits for indicating which

© 2005–2012 Tomas Akenine-Möller

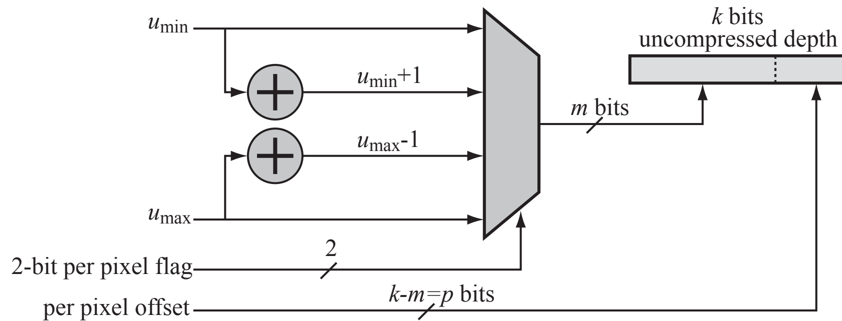

Figure 7.5: Decompression of a single depth value using the depth offset compression technique. One of the values u_{min}, $u_{min}+1$, $u_{max}-1$, and u_{max} are identified using the MUX and the 2-bit identifier that each pixel stores. When that is done, the result is concatenated with the pixel's offset value, which results in the uncompressed depth.

reference value it is encoded relative to, sine we can use u_{min}, $u_{min}+1$, $u_{max}-1$, and u_{max}. This costs $8 \times 8 \times 2$ bits=16 bytes. The offsets are of 8 bits, which costs 64 bytes for the entire tile. In total, this costs $64+16+4=84$ bytes. In uncompressed form, a tile occupies $8 \times 8 \times 3 = 192$ bytes. Thus, the compression reduces the data to $100 \times 84/192 \approx 44\%$, when the tile depths can be compressed. Compared to Example 7.2.1.1, 44% is still comparable to 41%, and since this technique is less expensive, it is worth that small amount of reduction in compression. □

In total, the amount of compression might actually become better with the inexpensive variant even though the example above shows otherwise. The reason for this is that the offset range that can be encoded is greater with this technique, and therefore more tiles should be possible to compress this way, and that might make up for its loss of compression efficiency. In the best case, the range is doubled, and in the worst case the range is only increased by one. Hence, it can be argued that, on average, the range is about 50% larger, and the extra cost for this is one bit per pixel. However, 50% corresponds to about half a bit, and so it cannot quite match up with the previous technique. It is still inexpensive in hardware though.

As shown in Figure 7.4, one can interpret this technique as having four different reference depth values, u_{min}, $u_{min}+1$, $u_{max}-1$, and u_{max}. Based on this, an example of how decompression hardware for a single depth value is done is illustrated in Figure 7.5.

It is possible that this the depth offset compression technique could be used for compressing colors as well. Each color component (R, G, B, and A) could be compressed separately. However, nothing has been documented about this.

© 2005–2012 Tomas Akenine-Möller

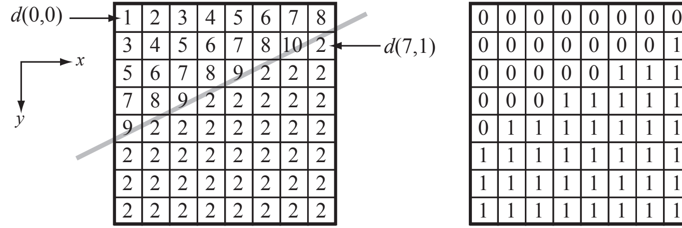

Figure 7.6: In this example, there are only two layers in this 8×8 pixel tile, and the two layers are identified using a single bit (0 or 1) as shown to the right. The depth values of the tile are shown to the left. As can be seen in the lower right area of the tile, a primitive with a constant depth of 2 has been rendered, and in the upper left region, a primitive has been rendered with linearly varying depth. The depth values in these two layers can be reconstructed from only knowing a single depth value ($d(0,0)$ for layer 0, and $d(7,1)$ for layer 1), and the differentials, $(\partial z^0/\partial x, \partial z^0/\partial y)$ for layer 0, and $(\partial z^1/\partial x, \partial z^1/\partial y)$ for layer 1. As can be seen for the upper left primitive, $\partial z^0/\partial x = 1$ and $\partial z^0/\partial y = 2$, and $\partial z^1/\partial x = 0$ and $\partial z^1/\partial y = 0$

7.2.2 Layered Plane Equation Compression

This algorithm exploits the fact that depth is interpolated linearly over a triangle, as in Equation 3.14. Thus, if only a single triangle is fully covering a tile, the plane equation of the triangle could be stored instead. This is the basic idea, and that idea is generalized to handling several plane equations (i.e., triangles) covering a tile.

An explicit plane equation is given as a function of screen space coordinates, (x, y):

$$d'(x,y) = d(x_0,y_0) + \frac{\partial z}{\partial x}(x - x_0) + \frac{\partial z}{\partial y}(y - y_0), \tag{7.4}$$

where $d(x_0, y_0)$ is the depth at a pixel (x_0, y_0), and the differential, $\partial z/\partial x$ could be calculated as:

$$\frac{\partial z}{\partial x} = d(x+1, y) - d(x - y), \tag{7.5}$$

and similarly for $\partial z/\partial y$. The idea is that all depth values of a layer, k, can be reconstructed as long as one depth value, $d(x_k, y_k)$, and the depth differentials, $(\partial z^k/\partial x, \partial z^k/\partial y)$ are known. Reconstruction is then done using Equation 7.4. It should be noted that the differentials must be constant for the entire tile, and care has to be taken so that this requirement is fulfilled.

An example of a compressed tile is shown in Figure 7.6. To reduce storage requirements, one can avoid storing the positions, (x_k, y_k), of the the depth value, $d(x_k, y_k)$. Instead this is assumed to be the first position when that layer is active. In Figure 7.6, this is $(0, 0)$ for layer 0, and $(7, 1)$ for layer 1, if the order is assumed to be left to right and top to bottom.

© 2005–2012 Tomas Akenine-Möller

Decompression is done by iterating over all the pixels, once per layer, k. The pixel coordinates, (x_k, y_k), are found when the the first pixel belonging to layer k is identified, and the depth value at that pixel is simply, $d(x_k, y_k)$, which is stored in the compressed representation. For each of the following pixels, (x_p, y_p), belonging to layer k, the depth is reconstructed using Equation 7.4.

Example 7.2.2.1 Layered Plane Equation Compression
In this example, we assume that 8×8 tiles are used, and that each depth value is stored in 24 bits. If we assume that only two layers are supported by one compression mode, each pixel need to store one bit as a layer identifier. This costs 8×8 bits $= 8$ bytes. Furthermore, the depth and depth differentials are stored using 3 bytes each, which means that each layer needs $3 \times 3 = 9$ bytes for that information. In total, this sums up to $9+9+8 = 26$ bytes. In uncompressed form, a tile occupies $8 \times 8 \times 3 = 192$ bytes. Thus, the compression reduces the data to $100 \times 26/192 \approx 14\%$, when the tile depths can be compressed. However, it should be noted that only tiles containing a single layer or two layers can be compressed with this technique. With four layers, this figure rises to $9 \times 4 + 2 \times 8 = 52$ bytes, and thus the compression ratio becomes $\approx 27\%$. With another four layers, i.e., a total of eight layers, this figure rises to $9 \times 8 + 4 \times 8 = 104$ bytes, and thus the compression ratio becomes $\approx 54\%$. □

The compression algorithm used for this technique is only described rather briefly here. Basically, a layer identifier need to be maintained for each pixel, and this could, for example, be a 4-bit register. This means that at most 15 different layers can be used, because the highest number using four bits is reserved for indicating that more layers than can be handled have been written to that tile. When the depth buffer is cleared, the layer identifier is set to zero for each pixel, and the depth values are set to the depth clear value. When a triangle is rendered over a tile, a layer counter is incremented for the first written pixel, and all other pixels in that triangle writes the value of the layer counter into the respective pixels' layer identifier. It must be possible to derive the differentials from the depth values alone, and to reduce hardware complexity, one can require that the first pixel, (x_k, y_k), belonging to a layer, k, must also have pixels at $(x_k + 1, y_k)$ and $(x_k, y_k + 1)$ belonging to layer k as well. Then the differentials are trivial to compute. However, this reduces the number of tiles that can be compressed using this technique.

7.2.3 DPCM Compression

This algorithm is based on a technique known as *differential pulse code modulation*, or DPCM, for short. The basic idea is to compute the second derivative of the depth function. If we have a linear function, such as depth over a triangle, the second derivative will be zero, and thus inexpensive to encode. However, since we are dealing with a discretization of of the depth, we need to compute the second derivative using differential techniques. The first two steps of this procedure is illustrated in Figure 7.7.

© 2005–2012 Tomas Akenine-Möller

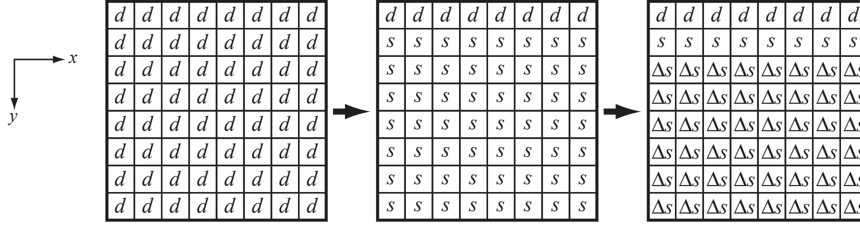

Figure 7.7: A tile with depths, d, is shown to the left. The first step of DPCM depth buffer compression is to compute the slopes, s, for all pixels below the first row. The second step is to compute the differential slopes, Δs, for all but the first two rows.

To explain this, let us focus for a moment on a single column of depth values. For simplicity, denote these depths, d_i, $i \in [0, 7]$, where d_0 is at the top of the column. The first step is to compute the slopes, s_i, $i \in [1, 7]$, as:

$$s_i = d_i - d_{i-1}. \tag{7.6}$$

The second step is to compute the differential slopes, Δs_i, $i \in [2, 7]$, as:

$$\Delta s_i = s_i - s_{i-1}. \tag{7.7}$$

This can also be expressed as:

$$\Delta s_i = s_i - s_{i-1} = (d_i - d_{i-1}) - (d_{i-1} - d_{i-2}) = d_i - 2d_{i-1} + d_{i-2}. \tag{7.8}$$

Now, given only d_0 and the slopes, s_i, it can be seen that all the depth values, d_i, can be reconstructed. This is done in order of the subscript, i, that is, first d_0 is computed (already known), then d_1, and then d_2, and so on. To reconstruct the second depth, compute $d_1 = d_0 + s_1 = d_0 + (d_1 - d_0)$. In general, it holds that:

$$d_i = d_{i-1} + s_i, \quad i \geq 1. \tag{7.9}$$

Unfortunately, the slopes, s_i, are not that simple to compress, since they might have large values (and for linear functions, they should be the same). Therefore, the differential slopes, Δs_i, are used.

Assume we know d_0, s_1, and the differential slopes, Δs_i, $i \in [2, 7]$. The second depth is reconstructed as before, $d_1 = d_0 + s_1$. The third depth could be evaluated as $d_2 = d_1 + s_2$. However, at this point, s_2 is not known, but it can be reconstructed in the same way as the depths in Equation 7.9. Hence, the slope s_2 is computed as $s_2 = s_1 + \Delta s_2 = s_1 + (s_2 - s_1)$, and in general, a slope can be calculated as:

$$s_i = s_{i-1} + \Delta s_i, \quad i \geq 2. \tag{7.10}$$

© 2005–2012 Tomas Akenine-Möller

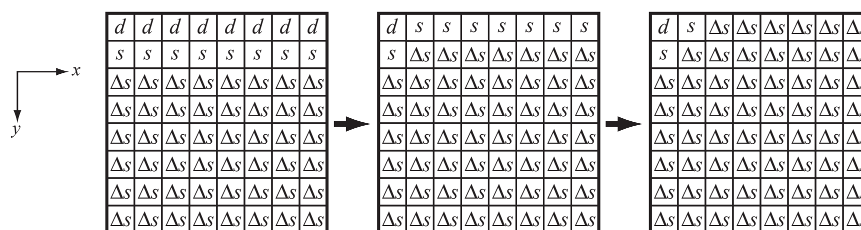

Figure 7.8: This illustration is a continuation of Figure 7.7, and it shows how the first two rows can be reduced to many differential slopes, Δs, as well. As can be seen, only one original depth value, d, is needed, and only two slopes, s. The remaining 61 values are differential slopes.

This means that each column in a tile is reconstructed, from d_0, s_1, Δs_i, $i \in [2, 7]$, as follows:

$$\begin{array}{ll} d_0, & \text{already known} \\ d_1 = d_0 + s_1, & s_1 \text{ already known} \\ d_2 = d_1 + s_2, & \text{where } s_2 = s_1 + \Delta s_2 \\ d_3 = d_2 + s_3, & \text{where } s_3 = s_2 + \Delta s_3 \\ \ldots \text{and so on} & \end{array} \tag{7.11}$$

Each column is compressed independently of the other columns, so parallel processing can be exploited to implement fast hardware for this. As previously touched upon briefly, it is most beneficial to reduce the depth values to as many differentials slopes as possible, since these tend to be close to zero, and thus easy to compress. Therefore, the first two rows, containing only d's and s's are transformed in the same was as the columns. The second row already consists of s's and therefore, the differential slopes, Δs, can be computed directly from that row of s's. For the first row, containing only d's, seven slopes, s, are first computed, and then six differential slopes, Δs, are calculated. That is, the first row is treated in exactly the same way as a column. This is all illustrated in Figure 7.8.

Due to the high correlation between the depth values over a tile, especially, when only one triangle has been rendered to the tile, it can be argued that the differential slopes, Δs, will be very small values. Typically, over a single triangle, the Δs are either -1, 0, or $+1$. In fact, if $\partial z/\partial x$ and $\partial z/\partial y$ from pixel to pixel, as assumed in Section 7.2.2, then the differential slopes should be 0—however, that can be tricky to obtain in practice, especially if Equation 3.14 is used directly to compute the depth per pixel. However, if more than one triangle is rendered to a tile, the differential slopes will have larger values, but this can be assumed to happen only for a small amount of the pixels, for the simple cases when only two triangles reside in a tile. Therefore, each pixel gets to store two extra bits, where 00_b means that the differential slope is $\Delta s = 0$, 01_b means that $\Delta s = +1$, and 10_b means that $\Delta s = -1$. As can be seen, the differential slopes are encoded using only two bits per value when this is fulfilled.

© 2005–2012 Tomas Akenine-Möller

This should be compared to 24 bits or even 32 bits that are often used for the per-pixel depth.

The last number, 11_b is used as an *escape* code. When an escape code is encountered, this means that the the differential slope is outside the range, $[-1, +1]$, and thus this case must be handled differently. Note that a one-bit value can be used instead of two bits. Then 0_b would mean that $\Delta s = 0$, and 1_b would be the escape code. More than two bits can also be used, and the advantage of that is that more differential slopes can be encoded without the escape. However, each pixel need to store more bits, so this is a tradeoff.

For the remainder of this text, we assume that two bits are used for this purpose. This means that 122 bits are needed for a tile's two-bit values, as there are 61 Δs-values per tile. To be able to encode any differential slope, one must investigate how many bits that are used during computing the Δs. The slopes, s, are computed as a difference of two depth values, and hence they need to be stored using 25 bits (see Section A.2.1), assuming the depth is stored in 24 bits per pixel. The differential slope is computed by subtracting two slopes, and therefore, a differential slope can be stored in 26 bits. This also means that if an escape code is encountered, then 26 bits should be used to to encode the differential slope, if it is required that every possible value of the differential slope should be possible to encode. This is an extreme case, and often much fewer bits suffice.

Example 7.2.3.1 DPCM Compression: Best case
When no escape codes at all are needed, the information that needs to be stored are $24 + 2 \times 25 + 2 \times 8 \times 8 = 202$ bits≈ 25 bytes. An uncompressed tile occupies 192 bytes, and so the compression rate is $100 \times 25/192 \approx 13\%$. This is a best case scenario, however. □

The following example illustrates what happens when two triangles are rendered to a tile.

Example 7.2.3.2 DPCM Compression: Common case
Let us focus on only a single column at first. Assume the depths, d_i, are 1, 2, 3, 4, 8, 10, 12, and 14. Clearly, this indicates that two primitives have been rendered into that column. The slopes would be 1, 1, 1, 4, 2, 2, and 2. Finally, the differential slopes become 0, 0, 3, -2, 0, and 0. As can be seen, this sequence of differential slopes would require two escape codes, one for 3 and one for -2. If we assume that a triangle edges cuts right through an 8×8 tile, there would be a need for two escape codes per column. This could be considered a common case, so let us investigate how much data is needed to store such a compressed block. As in Example 7.2.3.1, 202 bits are needed for everything but the differential slopes that are encoded when an escape could is encountered. We assume that $2 \times 8 = 16$ escape codes are needed, two per column. Thus, another 16×26 bits=52 bytes is needed. In total, this sums to $52 + 25 = 77$ bytes, which gives a compression ratio of $100 \times 77/192 = 40\%$. □

Since, the escape codes are quite expensive, as shown in the example above, another method is presented as well. The idea is to perform the coding both

© 2005–2012 Tomas Akenine-Möller

from the top row and down as already shown throughout this section, and also from the bottom row and up. When differential slopes are encountered that cannot be coded with the range $[-1, +1]$, we have detected the point where the slope and/or depth offset change. Assume that this happens at the third pixel from the top for one column. The encoding starting from the top row could be used until the third pixel, and the encoding from the bottom row could be used from the bottom pixel up to the fourth pixel. This technique would avoid these expensive differential slopes for the common case. This comes at a cost of three bits per column, since we need to know where the bottom row encoding should be used instead of the top encoding, and also another depth value and two slopes (for the bottom encoding).

$24 + 2 \times 25 + 2 \times 8 \times 8 = 202$ bits≈ 25

Example 7.2.3.3 DPCM Compression: Optimized common case
Using both a top encoding and a bottom encoding as explained above, the compression ratio can be lowered. This scheme costs $2 \times (24+25 \times 2)+2 \times 8 \times 8 = 276$ bits$= 34.5$ bytes for the two depth values and the four slopes, and the two bit codes per pixel. In addition, 3 bits are needed per column, which results in 3 bytes for an 8×8 tile. In total this sums to 37.5 bytes, which gives $100 \times 37.5/192 = 20\%$, which is about half of that in Example 7.2.3.2. □

Notice that neither of Examples 7.2.3.3 nor 7.2.3.1 actually use any of the escape codes, so it might be beneficial to use a differential slope range of $[-2, +1]$ instead of $[-1, +1]$.

To provide a compression mode where more complicated cases are handled, one can combine the techniques in Example 7.2.3.2 and 7.2.3.3. This would still provide a compression ratio of less than 50%.

Note also that it might be possible to encode the slopes using fewer bits, as well as the differential slopes that needs to be encoded when an escape code is encountered. This could provide even better compression ratios. Morein reports that this technique reduced the depth bandwidth with about 50% [42], but exactly what compression modes are used are not described.

7.3 Color Buffer Compression

Bengt-Olaf Schneider + someone else at NVIDIA. Look at non-lossy compression from JPEG as well: check copies from Petrik's book.

How much can you gain? Can a PACKMAN-like algorithm be used for this? Must be non-lossy, of course.

Depth offset compression can be used? Section 7.2.1.

© 2005–2012 Tomas Akenine-Möller

Chapter 8

Screen-space Antialiasing

8.1 Theory

8.2 Inexpensive Antialiasing Schemes

FLIPTRI [4], FLIPQUAD [3, 7], Hasselgren et al. [25].

8.3 High-Quality Antialiasing

Interleaved sampling: Heidrich and Keller, and ... Molnar.

© 2005–2012 Tomas Akenine-Möller

Chapter 9

Architectures for Mobile Devices

9.1 Tiling Architectures

ZR: A 3D API Transparent Technology for Chunk Rendering by Hsieh et al. 2001.

9.2 Bitboys?? Others?

© 2005–2012 Tomas Akenine-Möller

Appendix A

Fixed-point Mathematics

In this appendix, we will introduce the notation and operations that we use for fixed-point mathematics. For some platforms, e.g., low-cost mobile phones, fixed-point mathematics is used on the CPU due to lack of accelerated floating point instructions. It can be argued that floating point acceleration on the CPU will be added in the future, and so fixed-point math is nothing that is relevant to learn. However, in the hardware of a GPU, it often makes sense to use fixed-point math. To reduce the cost of an implementation, it is very important to reduce the needed accuracy (number of bits) in the computations without sacrificing quality.

It should also be pointed out that fixed-point math can be more accurate than floating point, especially if you know the maximum range of your numbers before starting.

A.1 Notation

In contrast to floating point, a fixed-point number has a decimal point with fixed position. Each number has a number of bits for the integer part, and another number of bits for the fractional part. With i integer bits, and f fractional bits, the notation for that representation is shown in Equation A.1.

$$[i.f] \tag{A.1}$$

Note that the most commonly used technique is to use two-complement if negative numbers are to be represented as well. Thus, there is no explicit sign bit as in floating point. The *resolution* of a fixed-point number is dictated by the number of fractional bits, f, and the resolution is 2^{-f}. This is the smallest unit that can be handled using this representation.

For f bits for the fractional part, the resolution is shown in Table A.1.

f	Resolution	Resolution
1	1/2	0.5
2	1/4	0.25
3	1/8	0.125
4	1/16	0.0625
5	1/32	0.03125
6	1/64	0.015625
7	1/128	0.0078125
8	1/256	0.00390625
12	1/4,096	0.000244140625
16	1/65,536	0.0000152587890625
24	1/16,777,216	0.000000059604644775390625
32	1/4,294,967,296	0.00000000023283064365386962890625

Table A.1: Resolution of fixed-point numbers with f fractional bits. The resolution is shown both as a rational number ($1/2^f$), and a decimal number.

A.1.1 Conversion

Converting between fixed-point and floating-point numbers is simple as shown in the examples below.

Example A.1.1.1 Conversion from floating-point to fixed-point
For example, assume you have a floating-point number, 2.345. Converting it to the $[5, 4]$-format is done as follows: $\texttt{round}(2.345 \times 2^4) = 38$. □

Example A.1.1.2 Conversion from fixed-point to floating-point
Now assume that we have a fixed-point representation in $[5, 4]$-format, and that the integer value of that representation is 38 (see example above). Converting it back to floating-point is done as follows: $\texttt{float}(38 \times 2^{-4}) = 2.375$. If truncation (using for example `int()` in C/C++) was used instead of `round()` in Example A.1.1.1, the fixed-point representation would become 37 instead, and when converting back to float the result would be 2.3125, which is a slightly worse approximation compared to 2.375. □

As can be seen in the two examples above, $2.345 \neq 2.375$, and this is one of the obstacles in using fixed-point representation: the accuracy can be quite bad, and therefore, extreme care has to be taken in order to provide sufficient accuracy. Usually, the indata and the needed accuracy in the result are analyzed, and fixed-point format is chosen thereafter.

In general, converting a floating-point number, a, to fixed point format, $[i.f]$, is done as shown below:

$$\texttt{round}(a \times 2^f), \tag{A.2}$$

where the result is an integer with $i + f$ bits. The corresponding formula for converting from fixed-point, b, to floating-point is:

$$\texttt{float}(b \times 2^{-f}) \tag{A.3}$$

© 2005–2012 Tomas Akenine-Möller

Note that conversion therefore often is implemented as shifting the numbers, since this is less expensive than multiplication and division.

It should be noted that we are not restricted to using scaling factors of 2^f and 2^{-f}. In general, any number can be used as scaling factor, but the powers of two are convenient because they can be implemented as simple left and right shifts. Furthermore, using scaling factors that are not powers of two makes for a representation that is not on $[i.f]$-format.

Rounding of a floating-point number, a, can be implemented as follows:

$$\texttt{round}(a \times 2^f) = \texttt{int}(\lfloor a \times 2^f + 0.5 \rfloor), \tag{A.4}$$

where `int()` is a function that simply truncates the fractional parts, i.e., throws them away. The floor function, $\lfloor x \rfloor$ simply truncates towards $-\infty$, e.g., $\lfloor 1.75 \rfloor = 1$ and $\lfloor -1.75 \rfloor = -2$.

A.2 Operations

In the following, we show what happens to the resulting fixed-point representation that is need to exactly being able to represent the result from an operation, e.g., multiplication. Such knowledge can help prevent overflow/underflow from occurring.

A.2.1 Addition/Subtraction

Addition and subtraction are performed by treating the fixed-point numbers as integers and adding them using standard addition/subtraction. In terms of the bit count of the result, that can be expressed as shown below:

$$[i.f] \pm [i.f] = [i + 1.f]. \tag{A.5}$$

As can be seen, all that happens is that the number of integer bits are increased by one. Intuitively, this makes sense, since adding the biggest number that can be represented in $[i.f]$ to itself, is the same as multiplying the number by two, which in turn is the same as shifting the number one step to the left. This always makes the number occupy one more bit, and correspondingly, the number of integer bits has been increased by one $(i + 1)$.

Example A.2.1.1
Adding two numbers represented in [8.8]-format gives a sum that is represented in [9.8], and hence, the result cannot be stored in 16 bits. □

If the two operands in the addition/subtraction has different number of bits, the bit count of the result is expressed as:

$$[i_1.f_1] \pm [i_2.f_2] = [\max(i_1, i_2) + 1.\max(f_1, f_2)], \tag{A.6}$$

that is, the number of integer bits of the results is one plus the maximum of the integer bits of the operands, and the fractional number of bits is equal to

© 2005–2012 Tomas Akenine-Möller

the maximum number of bits of the operands' fractional bits. However, in that case care must be taken so that standard integer addition can be used. This is done by aligning the number representations, as shown in the example below.

Example A.2.1.2
Assume we have two fixed-point numbers $a = [8.8]$ and $b = [4.4]$, and that we want to compute $c = a + b$. Furthermore, we assume that a and b are standard integers, and we want to add the numbers using standard addition. To make that work, the numbers must be aligned, and that is done by making sure they have the same number of fractional bits. Hence, we must see to it that b also has 8 fractional bits. This is done by simply shifting b four steps to the left. Therefore, the computation is done as $c = a + (b << 4)$, where c is on [9.8]-format. □

A.2.2 Multiplication

Multiplying two numbers is a bit more complicated. In terms of the number of bits in the result, Equation A.7 shows what happens.

$$[i.f] \times [i.f] = [2i.2f] \tag{A.7}$$

Multiplication of two fixed points numbers are performed as standard integer multiplication. However, as can be seen above, the number of fractional (and integer) bits have doubled. Thus, the decimal point is no longer at the same position.

The factor two comes from that multiplication can be thought of as a series of additions. For simplicity, consider an integer with i bits. The biggest number, h, is again when all bits are set to one. Multiplying this number by itself, $h \times h$, gives us the biggest possible product. This product can be expressed as: $h \times h = h + 2h + 4h + \cdots + 2^{i-1}h$. That is, first we have i bits in h, and we add $2h$, and that sum is represented using $i + 2$ bits according to Equation A.6. Then we add $4h$, which is represented with $i + 2$ bits as well. Thus, $h + 2h + 4h$ can be represented using $i + 3$ bits, and so on. After all terms have been added, this has grown to exactly i extra bits since $i - 1$ additions have to be done, and because another one is added due to addition. Hence the factor two in Equation A.7. Similar reasoning can be applied to the fractional bits as well.

In the general case, the resulting product is computed as shown below:

$$[i_1.f_1] \times [i_2.f_2] = [i_1 + i_2.f_1 + f_2]. \tag{A.8}$$

Again, one has to be careful when considering where the decimal point is located.

Example A.2.2.1 Multiplication of fixed-point numbers I
Assume that two fixed-point numbers [2.0] are to be multiplied. These numbers do not have any fractional bits, and only two integer bits. The biggest numbers that can be represented are 11_b, and the biggest product that can be produced is then $11_b \times 11_b = 1001_b$. As can be seen, the result has four bits as predicted. □

© 2005–2012 Tomas Akenine-Möller

Example A.2.2.2 Multiplication of fixed-point numbers II
Assume we have two floating-point numbers $r_1 = 0.79$ and $r_2 = 3.15$ that shall be multiplied using fixed-point representation [4.5]. Converting to fixed-point gives: $f_1 = \texttt{round}(0.79 \times 2^5) = 25$ and $f_2 = \texttt{round}(3.15 \times 2^5) = 101$. Multiplying gives: $f_3 = f_1 \times f_2 = 25 \times 101 = 2525$. Recall, that f_3 must be on $[4 + 4.5 + 5] = [8.10]$-format. Thus, when converting back to floating-point, we get: $r_3 = \texttt{float}(2525 \times 2^{-10}) = 2.465820\ldots$. The real result should be $0.79 \times 3.15 = 2.4885$. Note, that with more fractional bits in the initial conversion from floating-point to fixed-point, the end result will be more accurate. □

A.2.3 Reciprocal

Recall that the resolution of a fixed-point number using $[i.f]$-representation is 2^{-f}, that is, the smallest number that we can represent is the same as the resolution. When computing the reciprocal, $1/x$, then the biggest result that can be obtained must occur when the denominator, x, is smallest. Thus, the biggest number obtained through computing a reciprocal of a fixed-point number is $1/2^{-f} = 2^f$, and clearly there need to be f integer bits to represent this number. Hence, the result of computing $1/x$, where x is on $[i.f]$-format, must be on $[f.z]$-format. This means that in general, for reciprocal computation, it holds that:

$$\frac{1}{[i.f]} = [f.z]. \tag{A.9}$$

As can be seen in the in the following example, z, can, unfortunately, be infinitely large.

Example A.2.3.1 Reciprocal of fixed-point numbers
The fixed-point representation of 0.75 using three fractional bits is $\texttt{round}(0.75 \times 2^3) = 6$. The reciprocal is then $1/6 = 0.16666666666667$, but with infinitely many decimals. Clearly, this is not feasible to represent exactly using fixed-point. □

A reasonable representation of the reciprocal can be obtained by considering which is the biggest number that can be represented using $[i.f]$-format. Assuming we use signed fixed-point, the largest integer we can represent, using the i integer bits, is $2^i - 1$, and the fractional bits must represent a number, b, which is less than one, namely, $b \leq 1 - 2^{-f}$. Thus, the sum of these is $2^i - 1 + b \leq 2^i - 1 + 1 - 2^{-f} < 2^i$. If we assume that the number is exactly 2^i, the reciprocal will be 2^{-i}, that is, i fractional bits are needed. Hence, we have the following approximation:

$$\frac{1}{[i.f]} \approx [f.i]. \tag{A.10}$$

A more pragmatic way of thinking about this, is to decide which accuracy you need in the resulting number. The accuracy of the reciprocal is then:

$$\frac{\texttt{round}(1 \times 2^{f_1})}{[i_2.f_2]} = [f_2.f_1 - f_2], \tag{A.11}$$

© 2005–2012 Tomas Akenine-Möller

where the nominator is a fixed-point representation of 1 with f_1 fractional bits, and $f_1 \geq f_2$.

Example A.2.3.2
Assume that you want to compute the reciprocal of x and that you want the resulting number to have $f = 11$ fractional bits, because that is needed for subsequent computations. Now, if x has $f_2 = 5$ fractional bits, then the nominator in Equation A.11 must have $f_1 = 11 + 5$ bits, since that gives a result with $f_1 - f_2 = 16 - 5 = 11$ bits.

A.2.4 Division

TODO: This part of these notes is not finished, so the students can avoid this subsection. It is not needed in the course anyway, right now. □
Division:

$$\frac{[i_1.f_1]}{[i_2.f_2]} \approx [i_1 + f_2.f_1 - f_2], \tag{A.12}$$

where again $f_1 \geq f_2$.

A.2.5 Inexpensive division: special cases

For certain cases, computing the division by computing the reciprocal first, and then multiplying by that may be simpler. The special case of dividing by $2^n - 1$ is described by Blinn [12]. Looking at the fixed-point representation of such numbers, we find the following:

$$\begin{aligned}
1/3 &= 0.0101010101010 10\ldots \\
1/7 &= 0.001001001001001\ldots \\
1/15 &= 0.000100010001000\ldots \\
1/31 &= 0.000010000100001\ldots
\end{aligned}$$

As can be seen, the number of zeroes between the ones are becoming more and more, the higher the denominator. An approximation to $a/3$ can thus be implemented as $(a \times \mathsf{5555}_x) >> 16$. For 1/255, there are seven zeroes followed by a one, followed by seven zeroes and a one, and so on. Using 16 fractional bits, we can have the following approximation:

$$\frac{1}{255} \approx 1 << 8 + 1, \tag{A.13}$$

where the righthand side is on [0.16]-format. Thus, approximative division by 255 can be done by a shift and an add (and possibly some further shifting to get the result in the desired format). This is very inexpensive, compared to an integer division.

The original motivation [12] for such a division, was to compute the product of two eight-bit numbers, where 0 represented 0.0 and 255 represented 1.0. Assume we want to compute $c = a \times b$, where the result, c, is in the same format

© 2005–2012 Tomas Akenine-Möller

as the terms, a and b, in the product. This is done by computing $a \times b/255$. Assuming $d = a \times b$ (where d uses 16 bits), the trick above can be used to compute c as:

$$c = ((d << 8) + d) >> 16. \quad \text{(A.14)}$$

Now, since we wanted the result, c, to be stored in eight bits, there is yet another optimization to be done. Note that since d uses 16 bits, $d << 8$ must use 24 bits. Performing addition with more bits costs more if you are to build hardware for it, and therefore, it would be nice to reduce the size of the computations as much as possible.

Now, assume that $d = \texttt{HHLL}_x$, that is a 16-bit word (as above), with a higher-order byte, $\texttt{HH}_x$, and a lower-order byte, $\texttt{LL}_x$. Equation A.14 can be illustrated as follows:

```
 HHLL00        (d << 8)
+  HHLL        d
-------
 ssssLL
```

If we only need the eight significant bits of this sum, then the least significant bits can be safely ignored, since they cannot carry any information over to the rest of the sum (due to that the first operand as $\texttt{00}_x$ as the least significant bits). Equation A.14 can therefore be rewritten as:

$$c = (d + (d >> 8)) >> 8 \quad \text{(A.15)}$$

The nice property about this, is that the hardware for Equation A.15 is simpler than that for Equation A.14 since 24 bits plus 16 bits has been replaced by 16 bits plus 8 bits.

If you want include correct rounding as well, then Blinn [12] argues that it should be done like this:

$$c = (d + (d >> 8) + 1) >> 8. \quad \text{(A.16)}$$

A.2.6 Expansion of integers and fixed-point numbers

We start this section by a simple example that serves as motivation for how the expansion is done in general later on.

Example A.2.6.1 Integer expansion: from four to eight bits
Assume we have a four-bit integer, c_4, that we want to expand to eight bits in order to obtain c_8. The biggest number we can represent in four bits is $2^4 - 1 = 15$, and in eights bits, it is $2^8 - 1 = 255$. Thus, c_4 should be multiplied by $255/15 = 17$ in order to expand c_4 into c_8. This can be done like this: $c_8 = 17 \times c_4 = 16 \times c_4 + c_4 = (c_4 \ll 4)\ \texttt{or}\ c_4$, and this result is bit exact. □

In general, we do not get an integer when we compute $2^{i_1}/2^{i_2}$, $i_1 > i_2$, as we did in the example shown above, and therefore, we cannot get an exact expansion

© 2005–2012 Tomas Akenine-Möller

in the majority of cases. However, we can get a very good approximation using a trick that resembles what we did in Example A.2.6.1. Assume we have an integer, c, using i_2 bits that should be expanded into d using i_1 bits. This can be done as shown in Equation A.17.

$$\begin{aligned} d &= c \ll (i_1 - i_2) + c \gg (i_2 - (i_1 - i_2)) = \\ &= c \ll (i_1 - i_2) \text{ or } c \gg (2i_2 - i_1) \end{aligned} \tag{A.17}$$

Put another way, the original number is first shifted to the left so it occupies all the i_1 bits, and then we need to fill out the least significant bits of the number, and those are filled out using the most significant bits from c.

Example A.2.6.2 Integer expansion
Assume we have an integer in binary format, 10110_b, using five bits. To expand this to eight bits, we compute $(10110_b \ll 3) + (10110_b \gg 2) = 10110000_b$ or $101_b = 10110101_b$. Thus, to expand $10110_b = 22$ into eight bits, we get $10110101_b = 181$. The biggest number we can represent with five bits is $2^5 - 1 = 31$. To convert to eight bits, we would ideally like to multiply by $255/31 = 8.2258...$ Our expansion did a good job though, since the expansion is equivalent to a multiplication by $181/22 = 8.2272...$, which is far better than multiplying by eight. □

a

© 2005–2012 Tomas Akenine-Möller

Bibliography

[1] Timo Aila and Tomas Akenine-Möller. A Hierarchical Shadow Volume Algorithm. In *Graphics Hardware*, pages 15–23, 2004. On page(s) 64

[2] Timo Aila, Ville Miettinen, and Petri Nordlund. Delay Streams for Graphics Hardware. *ACM Transactions on Graphics,*, 22(3):792–800, 2003. On page(s) 64

[3] Tomas Akenine-Möller. FLIPQUAD: Low-Cost Multisampling Rasterization. Technical Report 02-04, Chalmers University of Technology, April 2002. On page(s) 79

[4] Tomas Akenine-Möller. An Extremely Inexpensive Multisampling Scheme. Technical report, Chalmers University of Technology/Ericsson Mobile Platfroms AB, August 2003. On page(s) 79

[5] Tomas Akenine-Möller and Timo Aila. Conservative and Tiled Rasterization Using a Modified Triangle Setup. *Journal of Graphics Tools*, 10(3):1–8, 2005. On page(s) 18, 20

[6] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering*. AK Peters Ltd., 2002. On page(s)

[7] Tomas Akenine-Möller and Jacob Ström. Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones. *ACM Transactions on Graphics,*, 22(3):801–808, 2003. On page(s) 17, 31, 48, 58, 61, 63, 79

[8] A.C. Beers, M. Agrawala, and Navin Chadda. Rendering from Compressed Textures. In *Proceedings of SIGGRAPH*, pages 373–378, 1996. On page(s)

[9] Jim Blinn. Texture and Reflection in Computer Generated Images. *Communications of the ACM*, 19(10):542–547, October 1976. On page(s) 33

[10] Jim Blinn. Simulation of Wrinkled Surfaces. In *Proceedings of SIGGRAPH*, pages 286–292, 1978. On page(s) 33

[11] Jim Blinn. Hyperbolic Interpolation. *IEEE Computer Graphics and Applications*, 12(4):89–94, July 1992. On page(s) 23

[12] Jim Blinn. Three Wrongs Make a Right. *IEEE Computer Graphics and Applications*, 15(6):90–93, November 1995. On page(s) 88, 89

[13] G. Campbell, T. A. DeFanti, J. Frederiksen, S. A. Joyce, L. A. Leske, J. A. Lindberg, and D. J. Sandin. Two Bit/Pixel Full Color Encoding. In *Proceedings of SIGGRAPH*, volume 22, pages 215–223, 1986. On page(s)

[14] Michael Cox and Pat Hanrahan. Pixel Merging for Object-Parallel Rendering: a Distributed Snooping Algorithm. In *Symposium on Parallel Rendering*, pages 49–56. ACM SIGGRAPH, November 1993. On page(s) 31

[15] Frank Crow. Shadow Algorithms for Computer Graphics. In *Computer Graphics (Proceedings of ACM SIGGRAPH 77)*, pages 242–248. ACM, July 1977. On page(s) 9

[16] Frank Crow. Summed-Area Tables for Texture Mapping. In *Computer Graphics (Proceedings of ACM SIGGRAPH 84)*, pages 207–212. ACM, July 1984. On page(s) 42

[17] E. Delp and O. Mitchell. Image Compression using Block Truncation Coding. *IEEE Transactions on Communications*, 2(9):1335–1342, 1979. On page(s)

[18] William Dungan, Anthony Stenger, and George Sutty. Texture Tile Considerations for Raster Graphics. In *Proceedings of SIGGRAPH*, pages 130–134, 1978. On page(s) 42

[19] Jon P. Ewins, Marcus D. Waller, Martin White, and Paul F. Lister. MIP-Map Level Selection for Texture Mpping. *IEEE Transactions on Visualization and Computer Graphics*, 4(4):317–329, 1998. On page(s) 37, 46, 48

[20] Simon Fenney. Texture Compression using Low-Frequency Signal Modulation. In *Graphics Hardware*, pages 84–91. ACM Press, 2003. On page(s)

[21] Richard Fromm, Styliannos Perissakis, Neal Cardwell, Christofors Kozyrakis, Bruce McCaughy, David Patterson, Tom Anderson, and Katherine Yelick. The Energy Efficiency of IRAM Architectures. In *24th Annual International Symposium on Computer Arhchitecture*, pages 327–337. ACM/IEEE, June 1997. On page(s)

[22] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical Z-Buffer Visibility. In *Proceedings of ACM SIGGRAPH 93*, pages 231–238. ACM Press/ACM SIGGRAPH, New York, J. Kajiya, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, August 1993. On page(s) 58

[23] Eric Haines and John Wallace. Shaft Culling for Efficient Ray-Traced Radiosity. In *Photorealistic Rendering in Computer Graphics (Proceedings of the Second Eurographics Workshop on Rendering)*, pages 122–138. Eurographics, 1994. On page(s) 18

© 2005–2012 Tomas Akenine-Möller

[24] Ziyad S. Hakura and Anoop Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. In *24th International Symposium of Computer Architecture*, pages 108–120. ACM/IEEE, June 1997. On page(s) 18, 32

[25] Jon Hasselgren, Tomas Akenine-Möller, and Samuli Laine. A Family of Inexpensive Sampling Schemes. *to appear in Computer Graphics Forum*, 2005. On page(s) 79

[26] Jon Hasselgren, Tomas Akenine-Möller, and Jacob Ström. Monosampling. *to be submitted*, 2005. On page(s)

[27] Paul Heckbert. Survey of Texture Mapping. *IEEE Computer Graphics and Applications*, 6(1):56–67, November 1986. On page(s) 42

[28] Paul S. Heckbert and Henry P. Moreton. Interpolation for Polygon Texture Mapping and Shading. In *State of the Art in Computer Graphics: Visualization and Modeling*, 1991. On page(s) 23

[29] John L. Hennessey and David A. Pattersson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990. On page(s) 51

[30] Homan Igehy, Matthew Eldridge, and Pat Hanrahan. Parallel Texture Caching. In *Graphics Hardware*, pages 95–106. ACM Press, 1999. On page(s)

[31] Homan Igehy, Matthew Eldridge, and Kekoa Proudfoot. Prefetching in a Texture Cache Architecture. In *Workshop on Graphics Hardware*. ACM SIGGRAPH/Eurographics, August 1998. On page(s)

[32] Kenneth E. Hoff III. A faster overlap test for a plane and a bounding box. Technical report, Available from World Wide Web (http://www.cs.unc.edu/ hoff/research/vfculler/boxplane.html, 1996. On page(s) 19

[33] Konstantine Iourcha, Krishna Nayak, and Zhou Hong. System and Method for Fixed-Rate Block-based Image Compression with Inferred Pixels Values. In *US Patent 5,956,431*, 1999. On page(s)

[34] Brian Kelleher. PixelVision Architecture. Technical report, Digital Systems Research Center, no. 1998-013, October 1998. On page(s)

[35] G. Knittel, A. Schilling, A. Kugler, and W. Strasser. Hardware for Superior Texture Performance. *Computers & Graphics*, 20(4):475–481, July 1996. On page(s)

[36] Olin Lathrop, David Kirk, and Doug Voorhies. Accurate Rendering by Subpixel Addressing. *IEEE Computer Graphics and Applications*, 10(5):45–53, September 1990. On page(s) 12

© 2005–2012 Tomas Akenine-Möller

[37] Dan McCabe and John Brothers. DirectX 6 Texture Map Compression. *Game Developer Magazine*, 5(8):42–46, August 1998. On page(s)

[38] Michael D. McCool, Chris Wales, and Kecin Moule. Incremental and Hierarchical Hilbert Order Edge Equation Polygon Rasterization. In *Graphics Hardware 2001*, pages 65–72, 2001. On page(s) 10, 29

[39] Joel McCormack, Bob McNamara, Christopher Gianos, Larry Seiler, Norman P. Jouppi, Ken Corell, Todd Dutton, and John Zurawski. Implementing Neon: A 256-Bit Graphics Accelerator. *IEEE Micro*, 19(2):58–69, March/April 1999. On page(s)

[40] Joel McCormack and Robert McNamara. Tiled Polygon Traversal Using Half-Plane Edge Functions. In *Workshop on Graphics Hardware*. ACM SIGGRAPH/Eurographics, August 2000. On page(s) 20

[41] Joel McCormack, Robert McNamara, Christopher Gianos, Larry Seiler, Norman P. Jouppi, and Ken Corell. Neon: A Single-Chip 3D Workstation Graphics Accelerator. In *Workshop on Graphics Hardware*. ACM SIGGRAPH/Eurographics, 1998. On page(s) 45

[42] Steve Morein. ATI Radeon HyperZ Technology. In *Workshop on Graphics Hardware, Hot3D Proceedings*. ACM SIGGRAPH/Eurographics, August 2000. On page(s) 58, 61, 65, 78

[43] Avi C. Naiman. Jagged Edges: when is Filtering Needed? *ACM Transactions on Graphics*, 17(4):238–258, 1998. On page(s)

[44] NVIDIA. HRAA: High-Resolution Antialiasing Through Multisampling. Technical report, 2001. On page(s)

[45] Marc Olano and Trey Greer. Triangle Scan Conversion using 2D Homogeneous Coordinates. In *Workshop on Graphics Hardware*. ACM SIGGRAPH/Eurographics, 1997. On page(s) 23

[46] John Owens. EEC 277: Graphics Architecture. Technical report, 2005. On page(s) 10

[47] Anton Pereberin. Hierarchical Approach for Texture Compression. In *Proceedings of GraphiCon '99*, pages 195–199, 1999. On page(s)

[48] Juan Pineda. A Parallel Algorithm for Polygon Rasterization. In *Computer Graphics (Proceedings of ACM SIGGRAPH 88)*, pages 17–20, August 1988. On page(s) 8, 17

[49] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics,*, 21(3):703–712, 2002. On page(s) 64

© 2005–2012 Tomas Akenine-Möller

[50] A. Schilling, G. Knittel, and W. Strasser. Texram: A Smart Memory for Texturing. *IEEE Computer Graphics and Applications*, pages 32–41, May 1996. On page(s) 42, 49

[51] Mark Segal and Kurt Akeley. The OpenGL 1.5 Specification. Technical report, 200. On page(s) 35, 36, 45

[52] Peter Shirley. *Physically Based Lighting Calculations for Computer Graphics*. PhD thesis, University of Illinois at Urbana Champaign, December 1990. On page(s)

[53] Jacob Ström and Tomas Akenine-Möller. PACKMAN: Texture Compression for Mobile Phones. In *Sketches program at SIGGRAPH*, 2004. On page(s)

[54] Jacob Ström and Tomas Akenine-Möller. *i*PACKMAN: High-Quality, Low-Complexity Texture Compression for Mobile Phones. In *submitted*, 2005. On page(s)

[55] Eric W. Weisstein. Areal Coordinates. *From MathWorld–A Wolfram Web Resource,* `http://mathworld.wolfram.com/ArealCoordinates.html`, 2005. On page(s) 22

[56] Lance Williams. Pyramidal Parametrics. In *Computer Graphics (Proceedings of ACM SIGGRAPH 83)*, pages 1–11. ACM, July 1983. On page(s) 32, 42

[57] Michael Wimmer and Jiri Bittner. Hardware Occlusion Queries Made Useful. *GPU Gems II,*, pages 91–108, 2005. On page(s) 64

[58] R. Woo, S. Choi, J. Sohn, S. Song, Y. Bae, and H. Yoo. A Low-Power 3-D Rendering Engine with Two Texture Units and 29-Mb Embedded DRAM for 3G Multimedia Terminals. *IEEE Journal of Solid-State Circuits*, 39(7):1101–1109, July 2004. On page(s) 16

[59] R. Woo, C. Yoon, J. Kook, S. Lee, and H. Yoo. A 120-mW 3-D Rendering Engine With a 6-Mb Embedded DRAM and 3.2 GB/s Runtime Reconfigurable Bus for PDA Chip. *IEEE Journal of Solid-State Circuits*, 37(19):1352–1355, October 2002. On page(s)

© 2005–2012 Tomas Akenine-Möller