A Data-Parallel Graphics Pipeline
Implemented in OpenCL

Examensarbete utfört i informationskodning
vid Tekniska högskolan vid Linköpings universitet
av

Joel Ek
LiTH-ISY-EX--12/4632--SE

Linköping 2012
A Data-Parallel Graphics Pipeline
Implemented in OpenCL

Examensarbete utfört i informationskodning
vid Tekniska högskolan vid Linköpings universitet
av

Joel Ek

LiTH-ISY-EX--12/4632--SE

Handledare: Harald Nautsch

ISY, Linköpings universitet

Examinator: Ingemar Ragnemalm

ISY, Linköpings universitet

Linköping, 27 november 2012
A Data-Parallel Graphics Pipeline Implemented in OpenCL
En Data-Parallell Grafikpipeline Implementerad i OpenCL

Författare
Author
Joel Ek

Sammanfattning
Abstract

This report documents implementation details, results, benchmarks and technical discussions for the work carried out within a master's thesis at Linköping University. Within the master's thesis, the field of software rendering is explored in the age of parallel computing. Using the Open Computing Language, a complete graphics pipeline was implemented for use on general processing units from different vendors. The pipeline is tile-based, fully-configurable and provides means of rendering visually compelling images in real-time. Yet, further optimizations for parallel architectures are needed as uneven work loads drastically decrease the overall performance of the pipeline.

Nyckelord
Keywords

computer graphics, GPGPU, OpenCL, graphics pipeline, rasterization, texture filtering, parallelization
A Data-Parallel Graphics Pipeline Implemented in OpenCL

Joel Ek
contact@joelek.se

October 29, 2012
Abstract
This report documents implementation details, results, benchmarks and technical discussions for the work carried out within a master’s thesis at Linköping University. Within the master’s thesis, the field of software rendering is explored in the age of parallel computing. Using the Open Computing Language, a complete graphics pipeline was implemented for use on general processing units from different vendors. The pipeline is tile-based, fully-configurable and provides means of rendering visually compelling images in real-time. Yet, further optimizations for parallel architectures are needed as uneven work loads drastically decrease the overall performance of the pipeline.
Contents

A Introduction ................................................. 1
  1 Report Structure ........................................ 3

B Background .................................................. 4
  1 Related Research ........................................ 4
    1.1 Samuli Laine and Tero Karras ...................... 4
    1.2 Larry Seiler et al. ................................ 6
  2 The Open Computing Language ............................ 6

C Mathematical Foundation ................................. 9
  1 The Fundamental Shapes ................................ 9
    1.1 Triangle .............................................. 9
    1.2 Quadrilateral ....................................... 10
  2 Line Normals ............................................. 11
  3 Geometric Series ....................................... 12
  4 Orthonormalization ..................................... 12
  5 Matrix Inversion ....................................... 14
    5.1 Four Elements ....................................... 14
    5.2 Nine Elements ....................................... 15
    5.3 Sixteen Elements .................................... 16
  6 Transformation Matrices ................................. 16
    6.1 Identity .............................................. 18
    6.2 Translation .......................................... 18
    6.3 Scaling ............................................... 19
    6.4 Basis .................................................. 20
    6.5 Rotation .............................................. 22
    6.6 Projection ............................................ 27
  7 Line Segment-Plane Intersection ....................... 28
  8 View Frustum Clipping .................................. 31
    8.1 Clipping Planes ..................................... 31
    8.2 Example ............................................... 33
    8.3 Algorithm ............................................. 35
  9 Barycentric Coordinates ................................ 37
    9.1 Cramer’s Rule ....................................... 39
6 Differentiation of Texture Coordinates .......................... 96
7 Fragment Illumination ............................................. 98

F Results ........................................................................ 101
1 Rendered Images ......................................................... 101
2 Texture Filtering ......................................................... 104
3 Benchmark Data ........................................................ 108
  3.1 Rasterization ......................................................... 108
  3.2 Load Balancing ...................................................... 108
  3.3 Texture Mapping ................................................... 109

G Discussion .................................................................. 111
1 Memory Requirements ................................................ 111
  1.1 Triangles ............................................................... 111
  1.2 Tiles ...................................................................... 112
  1.3 Fragments ............................................................. 113
2 Performance ............................................................. 114

H Conclusion .................................................................. 116
Chapter A

Introduction

With the introduction of consumer-level graphics processing units during the late 1990’s, three-dimensional computer graphics has become increasingly advanced and visually compelling. Today, we are on the verge of generating photo-realistic synthetic images in real-time. These synthetic images can sometimes be hard to distinguish from real photography, all thanks to the research in computer graphics and to the technical evolution of the graphics processing unit.

Modern graphics processing units are extremely powerful for parallel computing tasks such as rendering images. To perform such embarrassingly parallel tasks in serial on the central processing unit is considered a thing of the past. The visual quality expected in modern computer graphics imposes a requirement on the performance too large to be feasible in a serial architecture.

As there are multiple vendors of graphics processing units, the hardware will differ too much between vendors and models to program the hardware directly. Instead, modern three-dimensional computer graphics makes heavy use of application program interfaces which serve as abstraction layers of the hardware. Two well-used interfaces are the Open Graphics Library from the Khronos Group and Direct3D from Microsoft. Using these interfaces, the application programmer can set up the rendering procedure in a desired way, supply rendering data and instruct the hardware to perform the rendering. This without any knowledge about the actual graphics hardware present to the end user of the application.

Graphics hardware was once designed to render computer graphics in an almost unmodifiable fashion. The possibilities of specifying how a certain task should be performed were limited to a fixed set of pre-selected choices. For instance, lighting could previously only be computed at the geometry level using either flat or Gouraud shading as introduced in 1971 by Gouraud in [1]. Neither were
there any possibilities of multi-pass rendering. The need for programmability was evident.

Graphics processing units introduced some programmability during the early 2000’s. Previously fixed functionality was opened up to the application programmers through the introduction of small programs modifying the behavior of a certain stage in the graphics pipeline. These small programs became known as shaders since they were originally intended to modify the shading behavior in the graphics pipeline.

The shaders opened up for a whole range of new rendering techniques, not only limited to actual shading. As illustrious application programmers discovered what was possible, they also discovered what the limitations were. During the mid 2000’s, the concept of shaders evolved as vendors of graphics processing units introduced the unified shader model. Instead of having separate hardware sub-systems executing the different shader types, a unified array of processing units was introduced. These were capable of executing all shader types supported by the graphics hardware.

The unified model allowed application programmers to utilize the hardware as a massively parallel processing system. As such, the hardware was no longer limited to rendering images. Embarrassingly parallel tasks became accelerated by the new architecture.

The hardware vendors and interface developers realized the possibilities of the new architecture and released application program interfaces for the new hardware. Notably, Nvidia released the Compute Unified Device Architecture in 2007, the Khronos Group the Open Computing Language in 2008 and Microsoft DirectCompute in 2009.

The Compute Unified Device Architecture will only run on Nvidia hardware, DirectCompute only through Direct3D on Microsoft platforms whereas the Open Computing Language runs on many different systems and on hardware from different vendors.

These interfaces were intended for general-purpose heterogeneous computing on consumer-level graphics processing units. Through these interfaces, the units are now used for a wide array of massively parallel tasks.

An interesting set of question arises. Does the new tool set reopen the possibility of explicitly programming the entire graphics pipeline as done on the central processing unit in the mid 1990’s? If so, what are the benefits and drawbacks of a fully-configurable pipeline and how does its performance compare to the industry standard interfaces intended for computer graphics?
1 Report Structure

This master’s thesis will explore the structure of the graphics pipeline, evaluate the related techniques and discuss implementation details using the Open Computing Language. The following chapters of this report are structured as shown in the list below.

- Background
- Mathematical Foundation
- Texture Filtering
- Pipeline Overview
- Results
- Discussion
- Conclusion

The background chapter will discuss related research and give an introduction to the Open Computing Language. The following mathematical foundation will explore techniques and theorems used in computer graphics from a mathematical standpoint.

Texture filtering is a large subject in computer graphics and as such discussed in a separate chapter. In the chapter, details on the common texture filters are provided and as well as details on a set of non-standard filters such as the summed-area table.

The pipeline overview provides the actual details on the implemented graphics pipeline which uses the Open Computing Language. All major stages are explained using the mathematical foundation.

In the results chapter, the produced results are presented through rendered images and tables of benchmark data. The results are discussed in the following discussion chapter.

The final chapter concludes this master’s thesis and lists some drawbacks that were observed in the results chapter. It also suggests future research topics and how some stages of the pipeline could be implemented more efficiently.
Chapter B

Background

The graphics pipeline for rasterized graphics consists of several distinct stages. Each stage receives input data, processes the data and outputs it to the next stage in the pipeline. For a general pipeline, the stages are transformation, clipping, projection, culling, rasterization and shading followed by blending. The structure of the graphics pipeline varies between different application program interfaces and between different hardware and it is constantly evolving. Recently, a stage that allow tessellation of geometry data was introduced. This stage alters the linear layout of the traditional pipeline and is a complex stage, unfortunately out of scope for this master’s thesis.

The set of stages previously listed is the bare minimum of stages in the graphics pipeline required in order to correctly render three-dimensional computer graphics. The task of implementing a pipeline in a data-parallel fashion is straight forward for some stages. Other stages pose a great challenge as the parallel execution may cause data contamination. This is true especially for the rasterization and blending stages.

1 Related Research

1.1 Samuli Laine and Tero Karras

This master’s thesis is heavily influenced by the research presented in 2011 by Laine and Karras in [2]. In their paper, they present a data-parallel rasterization scheme running on the Compute Unified Device Architecture for Nvidia devices. Their paper is heavily focused on the rasterization stage as it is the stage in which most design decisions affecting performance can be made. As such, rasterization is a close-kept industrial secret of the different hardware manufacturers.
Laine and Karras employ a batch rendering scheme to enable high occupancy of the device. The data to be rendered is transferred to device memory and is composed of a vertex list and an indexed triangle list. They state that their intentions were to explore the raw rasterization performance and as such, no texturing occurs and no texture data is transferred. In addition, the vertex transformation stage is also left out of their pipeline. This as it is trivial to implement in parallel and its performance theoretically equal to that of the hardware pipeline. Lighting is computed at each individual vertex and not included in the benchmarks they present.

In the pipeline of Laine and Karras, the triangle list is processed in parallel. They employ a clipping pass which clips every triangle against the six planes of the view frustum, producing zero to seven sub-triangles. These are projected into an integer grid with sub-pixel resolution and potentially culled before the rasterization stage is entered.

For rasterization, they employ a sort-middle hierarchical rasterization refinement scheme. The screen is divided into bins of 128 by 128 pixels and every bin into tiles of 8 by 8 pixels. In addition, they limit the screen to 16 by 16 bins which corresponds to a resolution of 2048 by 2048 pixels. This yields a maximum number of bins of 256 where each bin in turn corresponds to 256 tiles and each tile to 64 pixels.

In the first rasterization pass of their pipeline, overlap between triangles and bins are computed in parallel. The ID of every triangle that overlaps a bin is appended to one of the local queues of that bin. This enables bin queues to be updated without synchronization between thread arrays and bins to be processed independently in the next pass, hence the sort-middle term.

In the second rasterization pass, bin queues are processed and the coverage between triangles and tiles computed. Similar to the first pass, the ID of triangles overlapping a tile are stored in one of the local queues of that tile. This is followed by the actual pixel rasterization pass in which coverage is computed using a look-up table approach.

Instead of computing a 64-bit coverage mask for each triangle and tile, the cases in a look-up table are determined for each of the three edges of the triangle. This can be represented using seven bits per edge according to Laine and Karras. In the shading pass, the actual coverage of the tile can be determined through bit-wise operations on the values of the look-up table.

Laine and Karras observe that their pipeline performs roughly 2 to 8 times slower in comparison to the hardware pipeline. This for a number of different test scenes with depth testing enabled, perspective-correct color interpolation and no multi-sampling.
1.2 Larry Seiler et al.

The Larrabee was a proposed microarchitecture by Intel, designed for applications with sections of highly parallel tasks. The platform was based on the x86 instruction set and introduced a range of new vector instructions to enable parallel processing of those tasks.

The new vector instructions were able to operate on a set of 32 new vector registers, each 512 bits wide. The width allowed each vector register to store eight 64-bit or sixteen 32-bit real-valued numbers or integers. The set of new instructions operated on those registers, effectively processing the stored values in parallel.

To demonstrate the capabilities of the Larrabee platform, a tile-based rasterization scheme was presented in 2008 by Seiler et al. in [3]. The rasterization employed a sort-middle hierarchical rasterization refinement scheme, very similar to that later presented by Laine and Karras.

As the new instructions were designed to handle at most 16 elements in parallel, Seiler et al. grouped 4 by 4 pixels in a tile and 4 by 4 tiles in a bin. In their pipeline, the rasterization was performed through evaluating the edge functions as detailed in 1988 by Pineda in [4]. The evaluated edge functions were compared to the trivial reject and accept values for each rasterization level. This was all done in parallel using the new instruction set.

The Larrabee platform will never reach any end users as the project was discontinued in 2010. This is unfortunate as it would have been beneficial to render three-dimensional computer graphics using the native x86 platform. However, their paper opened up a discussion on the feasibility of software rendering in the age of parallel computing and inspired the work of Laine and Karras.

2 The Open Computing Language

The Open Computing Language is an application program interface designed to abstract the hardware of general purpose computing units. As previously mentioned, the Khronos Group designed the specification to run on multiple platforms and hardware from different vendors. This makes it an ideal tool for implementing a hardware-accelerated graphics pipeline.

The interface allows small programs known as kernels to be executed in parallel on an Open Computing Language-enabled device. These devices can actually be of any architecture, including multi-core central processing units. However, the best performance is achieved on graphics processing units, provided that the task is highly parallel in nature.
Kernels are written using a subset of the C99 programming language and may be compiled in advance if the target device is known or at run-time to support the device available to the end user. The programming language does not support memory allocation nor function pointers. In addition, pointer arithmetic is allowed but could potentially break the code if memory alignment is not considered.

The kernels are executed in parallel as threads on the target device. How the threads are distributed within the device is determined by the implementer of the specification and as such will depend on the type of device. For instance, the Nvidia Fermi architecture consists of 1 to 15 streaming multi-processors, each with 32 cores. The threads are grouped into cooperative thread arrays and in turn into groups of 32 threads. Such a group is known as a warp and two warps may be issued and executed at each streaming multi-processor simultaneously. For each clock cycle, 16 cores receive an instruction from the first warp and the other 16 from the second warp [5].

For the Open Computing Language, a thread is known as a work item and group of threads as a work group. When executing a kernel on Nvidia hardware, work items translate to threads and work groups to cooperative thread arrays. As such, it is desirable to partition the work items into groups of \( n \cdot 32 \) for the Fermi architecture. This is somewhat problematic in the Open Computing Language as the total number of work items must be a multiple of the work group size. However, it can be achieved through issuing a greater number of work items than required for the task and ensuring that kernels return instantly for the extra work items. Since the total number of work items often is large, the issuing of additional work items should not have a significant impact on the overall performance.

The Open Computing Language specification defines four different memory address spaces. These are private memory for each work item, local memory that is shared between the work items of a work group, global memory that is shared between all work groups and constant memory which is a read-only variant of the global memory. The specification requires that the amount of local memory is at least 16 kilobytes but does not specify that the local memory needs to be located on-chip [6].

On the Fermi architecture, the local memory is located in the L1 cache of each streaming multi-processor and can be configured to either 16 or 48 kilobytes of the total 64 kilobytes. This makes the local memory extremely fast on the Fermi architecture. As such, kernels should store data shared between the work items of a work group in local memory whenever possible.

The graphics pipeline implemented was designed with Fermi as the target architecture. As such, all kernels are executed in work groups of 32 work items and local memory is utilized whenever possible.
The data transfers and the execution of kernels in the Open Computing Language are handled by a device queue. The queue employs the just in time execution model and queued commands are only guaranteed to have completed if the queue is explicitly instructed to complete all commands. As the data to be transferred often depends on the results of previously executed kernels, data transfers can be enqueued as blocking. This ensures that the application program interface does not return control to the application until the commands have completed.

The Open Computing Language is able to inter-operate with application program interfaces designed for computer graphics such as the Open Graphics Library or Direct3D. This allows generic data to be processed by kernels and displayed using a graphics interface. In order to utilize this functionality, the context of the Open Computing Language must be shared with that of the graphics interface. This is performed in the initialization stage in which the context of the graphics interface is created, often through additional interfaces for context and window handling. The context of the Open Computing Language is then instructed to use the same context as the graphics interface. Resources can then be acquired and released using the command queue.

One notable inter-operation feature is the possibility of sharing an image object between the two interfaces. This allows an image to be generated using the Open Computing Language and displayed using the graphics interface. It also allows the image to be rendered using the graphics interface, processed by kernels of the Open Computing Language and displayed by the graphics interface. For these examples, no image data needs to be transferred between the host and the device, making inter-operation features extremely useful.
Chapter C

Mathematical Foundation

In order to fully comprehend the concepts discussed throughout the rest of this report, a mathematical foundation is needed. This chapter assumes that the reader is familiar with linear algebra and specifically the notation of vectors and matrices.

1 The Fundamental Shapes

Two shapes commonly used in three-dimensional computer graphics are triangles and quadrilaterals.

1.1 Triangle

The fundamental shape of three-dimensional computer graphics is the triangle. It is formed from three vertices and its edges span a plane if they are linearly independent. Should two edges be linearly dependent (parallel), the triangle will collapse and have zero area. Figure 1.1 shows three vertices $\vec{U}$, $\vec{V}$ and $\vec{W}$ and how they form a proper triangle.
Equation 1.1 shows how the area of a two-dimensional triangle $A_t$ can be computed using the cross-product between two of the edges. Depending on the choice of edges, the resulting area may be negative. However, the cross-product between edges $\vec{U}$ to $\vec{V}$ and $\vec{U}$ to $\vec{W}$ is guaranteed to be positive if the vertices are specified in clock-wise order for a left-handed coordinate system or in counter clock-wise order for a right-handed coordinate system. A triangle with negative area is back-facing.

$$A_t = \frac{(V_x - U_x) \cdot (W_y - U_y) - (W_x - U_x) \cdot (V_y - U_y)}{2}$$ (1.1)

### 1.2 Quadrilateral

Quadrilaterals are formed from four vertices and are slightly more complex than their three-vertex counterparts. In contrast to triangles, the vertices of a quadrilateral are not guaranteed to coincide with a single plane and cannot be arranged into arbitrary shapes. This makes their use in defining three-dimensional geometry limited. Yet there are other areas of computer graphics in which proper quadrilaterals are formed, making them useful. Figure 1.2 shows four vertices $\vec{K}$, $\vec{L}$, $\vec{M}$ and $\vec{N}$ and how they form a proper quadrilateral.
Equation 1.2 shows how the area of a two-dimensional quadrilateral $A_q$ can be computed using the cross-product between two diagonals. Similar to triangles, the sign of the area depends on the choice of diagonals. In the equation, the cross-product between diagonals $\vec{K}$ to $\vec{M}$ and $\vec{L}$ to $\vec{N}$ is computed. This choice of diagonals guarantees that the resulting area is positive if the quadrilateral is front-facing with respect to the order of the vertices and to the reference coordinate system.

\[
A_q = \frac{(M_x - K_x) \cdot (N_y - L_y) - (N_x - L_x) \cdot (M_y - K_y)}{2}
\]

(1.2)

2 Line Normals

A two-dimensional line separates space into two sub-spaces. These two can be referred to as the positive and negative sub-spaces and are defined by selecting a line normal. For two-dimensional lines and line segments, there are two possible line normals of unit length. Figure 2.1 shows a line segment from $\vec{S}$ to $\vec{E}$ and the corresponding line normals $\vec{N}_1$ and $\vec{N}_2$. 

Figure 1.2: A quadrilateral is formed from four vertices.

Figure 2.1: A line segment and its corresponding line normals.
Equation 2.1 shows the relationship between the coordinates of the starting point \( \vec{S} \) and the ending point \( \vec{E} \) and the unnormalized line normals \( \vec{N}_1 \) and \( \vec{N}_2 \). Selecting one of these will effectively determine which sub-space is positive and which is negative.

\[
\vec{N}_1 = (E_y - S_y, S_x - E_x) \\
\vec{N}_2 = (S_y - E_y, E_x - S_x)
\]

(2.1)

3 Geometric Series

A geometric series is an infinite sum where each term is the previous term multiplied by a constant factor \( q \). These series are formed from recursive patterns and can be used to analyze whether a sum will grow indefinitely or if there is an upper bound at which the series converges. Equation 3.1 states that the series will converge if the absolute value of the constant factor \( q \) is less than one. It also implies that the sum of a finite number of terms is bounded by the expression \( \frac{1}{1-q} \).

\[
\lim_{N \to \infty} \sum_{i=0}^{N} q^i = \begin{cases} 
\frac{1}{1-q} & \text{if } |q| < 1 \\
\infty & \text{otherwise}
\end{cases}
\]

(3.1)

4 Orthonormalization

In order to span the entire three-dimensional space, a set of three linearly independent basis vectors is needed. This set does not need to be orthogonal,
though it is a convenient property in three-dimensional computer graphics. Even more convenient is to ensure that the set is both orthogonal and that each basis vector is of unit length. This is known as an orthonormal set of basis vectors.

To force any given set of linearly independent basis vectors to be orthonormal, the Gram-Schmidt process can be used. The process operates on each basis vector in sequence and may start at any vector. Equation 4.1 shows how the first basis vector $\vec{X}'$ is selected from the set and normalized into $\vec{X}''$.

$$\vec{X}' = \vec{X}$$
$$\vec{X}'' = \frac{\vec{X}'}{|\vec{X}'|} \quad (4.1)$$

Equation 4.2 shows how the second basis vector $\vec{Y}'$ is found by subtracting the projection of $\vec{Y}$ on $\vec{X}''$ from itself. The vector is then normalized into $\vec{Y}''$ and the two vectors $\vec{X}''$ and $\vec{Y}''$ now form an orthonormal pair of basis vectors.

$$\vec{Y}' = \vec{Y} - (\vec{Y} \cdot \vec{X}'') \cdot \vec{X}''$$
$$\vec{Y}'' = \frac{\vec{Y}'}{|\vec{Y}'|} \quad (4.2)$$

Equation 4.3 shows how the process continues with the final basis vector $\vec{Z}'$. The basis vector orthogonal to both $\vec{X}''$ and $\vec{Y}''$ is found by subtracting the projection of $\vec{Z}$ on $\vec{X}''$ and on $\vec{Y}''$ from itself. Finally, the vector is normalized and $\vec{Z}''$ is obtained.

$$\vec{Z}' = \vec{Z} - (\vec{Z} \cdot \vec{X}'') \cdot \vec{X}'' - (\vec{Z} \cdot \vec{Y}'') \cdot \vec{Y}''$$
$$\vec{Z}'' = \frac{\vec{Z}'}{|\vec{Z}'|} \quad (4.3)$$

After the process, the set of basis vectors $\vec{X}''$, $\vec{Y}''$ and $\vec{Z}''$ is orthonormal and each basis vector is of unit length. This process is useful if the set is initially orthonormal but subject to multiple transformations as small errors are introduced with every transformation in a computational context. Figure 4.1 illustrates the orthonormalization process.
5 Matrix Inversion

Inverting small square matrices is a common task in three-dimensional computer graphics and should therefore be detailed here. Equation 5.1 shows the general formula of the adjoint method. The method is often performed in three steps and produces an analytical inverse to a square matrix of any size. However, the complexity of the method grows fast, making it unsuitable for larger matrices. For such matrices, numerical inversion methods exist.

\[ M^{-1} = \frac{\text{adj}(M)}{\det(M)} = \frac{\text{cof}(M)^T}{\det(M)} \text{ if } \det(M) \neq 0 \]  

(5.1)

Firstly, the minor matrix of \( M \) is computed. The minor at index \((i, j)\) is defined as the determinant of the matrix formed by excluding row \( i \) and column \( j \) from \( M \). Secondly, the cofactor matrix is computed. The cofactor at index \((i, j)\) is defined as the minor at index \((i, j)\) multiplied by \((-1)^{i+j}\). Finally, the cofactor matrix is transposed and the adjoint matrix created. Dividing every element of the adjoint matrix by the determinant of the original matrix results in the inverse matrix \( M^{-1} \). This provided that the determinant is not equal to zero.

5.1 Four Elements

Equation 5.2 shows a square matrix \( M_f \) and its four elements \( a, b, c \) and \( d \).

\[ M_f = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \]  

(5.2)
Equation 5.3 shows how the determinant of $M_f$ is defined.

$$\det(M_f) = a \cdot d - b \cdot c$$ (5.3)

Equation 5.4 shows the cofactor matrix of $M_f$. For four-element matrices, it seems as if the cofactor matrix is formed through reorganizing the elements of $M_f$. However, every element at index $(i, j)$ is actually the minor at index $(i, j)$ with the corresponding sign adjustment.

$$\text{cof}(M_f) = \begin{bmatrix} +d \quad -c \\ -b \quad +a \end{bmatrix}$$ (5.4)

Transposing the cofactor matrix and dividing every element by the determinant of $M_f$ gives the inverse matrix $M_f^{-1}$. This is seen in equation 5.5.

$$M_f^{-1} = \frac{1}{\det(M_f)} \begin{bmatrix} +d \quad -b \\ -c \quad +a \end{bmatrix} \text{ if } \det(M_f) \neq 0$$ (5.5)

### 5.2 Nine Elements

Equation 5.6 shows a square matrix $M_n$ and its nine elements $a, b, c, d, e, f, g, h$ and $i$.

$$M_n = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$ (5.6)

Equation 5.7 shows how the determinant of $M_n$ is defined.

$$\det(M_n) = a \cdot (e \cdot i - f \cdot h) - b \cdot (d \cdot i - f \cdot g) + c \cdot (d \cdot h - e \cdot g)$$ (5.7)

The cofactor matrix of $M_n$ is formed by computing the minors of $M_n$ and adjusting the sign as previously stated. In contrast to $M_f$, the complexity is higher as every minor is computed as the determinant of a square four-element matrix. This is seen in equation 5.8.
\[
\text{cof}(M_n) = \begin{bmatrix}
+ (e \cdot i - f \cdot h) & -(d \cdot i - f \cdot g) & +(d \cdot h - e \cdot g) \\
-(b \cdot i - c \cdot h) & +(a \cdot i - c \cdot g) & -(a \cdot h - b \cdot g) \\
+(b \cdot f - c \cdot e) & -(a \cdot f - c \cdot d) & +(a \cdot e - b \cdot d)
\end{bmatrix}
\]

Transposing the cofactor matrix and dividing every element by the determinant of \(M_n\) gives the inverse matrix \(M_n^{-1}\). This is seen in equation 5.9.

\[
M_n^{-1} = \frac{1}{\det(M_n)} \begin{bmatrix}
e \cdot i - f \cdot h & c \cdot h - b \cdot i & b \cdot f - c \cdot e \\
f \cdot g - d \cdot i & a \cdot i - c \cdot g & c \cdot d - a \cdot f \\
d \cdot h - e \cdot g & b \cdot g - a \cdot h & a \cdot e - b \cdot d
\end{bmatrix}
\text{ if } \det(M_n) \neq 0
\]

(5.9)

### 5.3 Sixteen Elements

As the method is valid for any square matrix, it can be used to invert sixteen-element matrices. However, the complexity is significantly higher when compared to nine-element matrices, making the method unsuitable for real-time purposes. Yet, sixteen-element matrices are the most commonly used matrices in three-dimensional computer graphics and the need for clever inversion methods is evident.

### 6 Transformation Matrices

Three-dimensional computer graphics uses three-dimensional vectors to define various geometric properties. Such properties are, amongst others, the topology of a surface, the location of a light source and the orientation of a camera or an observer.

In order to fully model a transformation of a three-dimensional vector, the vector is often stored and handled in its four-dimensional homogeneous form. This embeds the three-dimensional vector in a four-dimensional super-space for which the fourth component is almost always equal to one. Equation 6.1 shows a homogeneous four-component vector \(\vec{P}\).

\[
\vec{P} = (P_x, P_y, P_z, 1)
\]

(6.1)
The homogeneous form enables affine transformations of the vector using a square sixteen-component transformation matrix. Equation 6.2 shows an affine transformation matrix $A$. For most affine transformations, the bottom row will be equal to $(0, 0, 0, 1)$.

$$A = \begin{bmatrix} X_x & Y_x & Z_x & W_x \\ X_y & Y_y & Z_y & W_y \\ X_z & Y_z & Z_z & W_z \\ X_w & Y_w & Z_w & W_w \end{bmatrix} \quad (6.2)$$

The vector $\vec{P}$ is defined in its local coordinate system. If this coordinate system is embedded in a global coordinate system, the matrix $A$ represents a transformation from the local coordinate system into the global one. The three vectors $\vec{X}$, $\vec{Y}$ and $\vec{Z}$ represent the basis vectors of the local coordinate system expressed in the global coordinate system. Conversely, the fourth vector $\vec{W}$ can be viewed as the translation vector which represents the offset of the local coordinate system with respect to the global one.

Figure 6.1 shows this relationship. The vector $\vec{P}$ is defined in its local coordinate system with base vectors $\vec{X}$, $\vec{Y}$ and $\vec{Z}$. Applying matrix $A$ to the vector $\vec{P}$ results in the vector being expressed in $\vec{X}', \vec{Y}'$ and $\vec{Z}'$. To transform in the opposite direction, the inverse of $A$ is applied to the vector $\vec{P}$ expressed in the global coordinate system.

![Figure 6.1: A transformation matrix transforms vectors between coordinate systems.](image)

A vector may be transformed between multiple coordinate systems and by different transformation matrices before reaching the coordinate system of a camera or an observer. This forms a chain of transformations. This transformation chain is often premultiplied so that the vector only needs to be transformed once. This is shown in equation 6.3.
\[ \vec{P}' = A_n \cdot A_{n-1} \cdot \ldots \cdot A_2 \cdot A_1 \cdot \vec{P} = (A_n \cdot A_{n-1} \cdot \ldots \cdot A_2 \cdot A_1) \cdot \vec{P} \]  \hspace{1cm} (6.3)

### 6.1 Identity

The identity matrix is a transformation matrix which when applied to a vector does not alter the vector in any way. Its only function in the transformation chain is to initialize it so that subsequent matrices in the chain can be premultiplied. Equation 6.4 shows an identity matrix \( I \) and its inverse \( I^{-1} \). As seen in the equation, the matrix and its inverse are identical.

\[
I = I^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \hspace{1cm} (6.4)
\]

### 6.2 Translation

The translation matrix is a transformation matrix that when applied to a vector adds another vector to it. When applied to a three-dimensional model, the model will be offset by the translation vector. Equation 6.5 shows a translation matrix \( T \) and how the three vector components \( x, y \) and \( z \) are embedded into the matrix.

\[
T(x, y, z) = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \hspace{1cm} (6.5)
\]

Inverting the translation matrix is done by translating in the opposite direction. Equation 6.6 shows the relationship between the translation matrix \( T \) and its inverse \( T^{-1} \).

\[
T(x, y, z)^{-1} = T(-x, -y, -z) = \begin{bmatrix} 1 & 0 & 0 & -x \\ 0 & 1 & 0 & -y \\ 0 & 0 & 1 & -z \\ 0 & 0 & 0 & 1 \end{bmatrix} \hspace{1cm} (6.6)
\]
6.3 Scaling

Uniform Scaling

The uniform scaling matrix is a transformation matrix that when applied to a vector scales all of its components equally. When applied to a three-dimensional model, the model will shrink or grow depending on the magnitude of the scaling factor \( f \). This provided that the model is located around the origin of the local coordinate system. Equation 6.7 shows a uniform scaling matrix \( S_u \) and how the scaling factor \( f \) is embedded into the matrix.

\[
S_u(f) = \begin{bmatrix}
  f & 0 & 0 & 0 \\
  0 & f & 0 & 0 \\
  0 & 0 & f & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix}
\]

(6.7)

Inverting the uniform scaling matrix is done by scaling with the inverse of \( f \). Equation 6.8 shows the relationship between the uniform scaling matrix \( S_u \) and its inverse \( S_u^{-1} \). As seen in the equation, the inverse does only exist if the scaling factor is not equal to zero. This as a scaling factor of zero would collapse the transformed vector into the null vector, eliminating the possibility of returning to the original vector.

\[
S_u(f)^{-1} = S_u \left( \frac{1}{f} \right) = \begin{bmatrix}
  \frac{1}{f} & 0 & 0 & 0 \\
  0 & \frac{1}{f} & 0 & 0 \\
  0 & 0 & \frac{1}{f} & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix}
\] \quad \text{if } f \neq 0
\]

(6.8)

Non-Uniform Scaling

The non-uniform scaling matrix is a transformation matrix that when applied to a vector scales all of its components differently. When applied to a three-dimensional model, the model will become wider or smaller in each of the three directions. As for uniform scaling matrices, this requires that the model is located around the origin of the local coordinate system. Equation 6.9 shows a non-uniform scaling matrix \( S \) and how the scaling factors \( f_x, f_y \) and \( f_z \) are embedded into the matrix.

\[
S(f_x, f_y, f_z) = \begin{bmatrix}
  f_x & 0 & 0 & 0 \\
  0 & f_y & 0 & 0 \\
  0 & 0 & f_z & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix}
\]

(6.9)
Inverting the non-uniform scaling matrix is done in the same fashion as with uniform scaling matrices. The inverse is created by scaling with the inverses of $f_x$, $f_y$ and $f_z$. Equation 6.10 shows this relationship. The inverse requires that each of the three scaling factors is not equal to zero as this would collapse one or several of the components of the vector.

\[
S(f_x, f_y, f_z)^{-1} = S \left( \frac{1}{f_x}, \frac{1}{f_y}, \frac{1}{f_z} \right) = \begin{bmatrix}
\frac{1}{f_x} & 0 & 0 & 0 \\
0 & \frac{1}{f_y} & 0 & 0 \\
0 & 0 & \frac{1}{f_z} & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \quad \text{if } f_x, f_y, f_z \neq 0 \quad (6.10)
\]

### 6.4 Basis

The basis matrix is a transformation matrix that when applied to a vector transforms the vector from its local coordinate system to a global coordinate system. The local coordinate system is defined by the three base vectors $\vec{X}$, $\vec{Y}$ and $\vec{Z}$ expressed in the global coordinate system as previously detailed. Equation 6.11 shows a basis matrix $B$ and how the three base vectors are embedded into the matrix.

\[
B(\vec{X}, \vec{Y}, \vec{Z}) = \begin{bmatrix}
X_x & Y_x & Z_x & 0 \\
X_y & Y_y & Z_y & 0 \\
X_z & Y_z & Z_z & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \quad (6.11)
\]

Inverting the basis matrix can be done in a few different ways depending on the properties of the set of base vectors. The upper-left nine components can be inverted using the inversion method for nine-component matrices discussed earlier. This as the other elements of the matrix are equal to those of an identity matrix.

Using this method will amount to finding a set of modified base vectors $\vec{X}'$, $\vec{Y}'$ and $\vec{Z}'$ and embedding this set in a transposed fashion. This can be seen in equation 6.12.

\[
B(\vec{X}', \vec{Y}', \vec{Z}')^{-1} = \begin{bmatrix}
X'_x & X'_y & X'_z & 0 \\
Y'_x & Y'_y & Y'_z & 0 \\
Z'_x & Z'_y & Z'_z & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \quad (6.12)
\]
When using this method, the inverse matrix $B^{-1}$ will only exist if the determinant of $B$ is not equal to zero. Computing the determinant from the three original base vectors is done as shown in equation 6.13.

$$\det(B) = X_x \cdot (Y_y \cdot Z_z - Z_y \cdot Y_z) - Y_x \cdot (X_y \cdot Z_z - Z_y \cdot X_z) + Z_x \cdot (X_y \cdot Y_z - Y_y \cdot X_z) \quad (6.13)$$

Equation 6.14 shows how the set of modified base vectors is defined from the original set of base vectors. The components of these vectors are embedded into the inverse matrix and the inversion is complete.

$$\vec{X}' = \frac{\vec{Y} \times \vec{Z}}{\det(B)} \cdot \begin{pmatrix} Y_y \cdot Z_z - Y_z \cdot Z_y \\ Y_z \cdot X_x - X_z \cdot Y_x \\ Y_x \cdot Z_y - Z_x \cdot Y_y \end{pmatrix} \quad \text{if } \det(B) \neq 0$$

$$\vec{Y}' = \frac{\vec{Z} \times \vec{X}}{\det(B)} \cdot \begin{pmatrix} Z_y \cdot X_z - Z_z \cdot X_y \\ Z_z \cdot X_y - X_z \cdot Z_y \\ Z_x \cdot X_y - Z_y \cdot X_x \end{pmatrix} \quad \text{if } \det(B) \neq 0 \quad (6.14)$$

$$\vec{Z}' = \frac{\vec{X} \times \vec{Y}}{\det(B)} \cdot \begin{pmatrix} X_y \cdot Z_z - Z_y \cdot X_z \\ X_z \cdot X_y - X_y \cdot X_z \\ X_x \cdot Y_y - Y_x \cdot Y_y \end{pmatrix} \quad \text{if } \det(B) \neq 0$$

If the set of base vectors is orthogonal and the bases not of unit lengths, the modified base vectors will correspond to the original ones divided by their square lengths. This assumes that no base vector is equal to the null vector, which would result in a determinant of zero. Equation 6.15 shows this process.

$$\vec{X}' = \frac{\vec{X}}{\vec{X} \cdot \vec{X}} \quad \text{if orthogonal } \land \vec{X} \cdot \vec{X} \neq 0$$

$$\vec{Y}' = \frac{\vec{Y}}{\vec{Y} \cdot \vec{Y}} \quad \text{if orthogonal } \land \vec{Y} \cdot \vec{Y} \neq 0 \quad (6.15)$$

$$\vec{Z}' = \frac{\vec{Z}}{\vec{Z} \cdot \vec{Z}} \quad \text{if orthogonal } \land \vec{Z} \cdot \vec{Z} \neq 0$$

If the set of base vectors is orthonormal, the inverse matrix $B^{-1}$ will always be equal to the transpose of $B$. If this is a know property of the set of base vectors, the inverse will always exist and full inversion is unnecessary. The inverse matrix can instead be created through embedding the original base vectors in a transposed fashion. This is shown in equation 6.16.
\[ B(\vec{X}, \vec{Y}, \vec{Z})^{-1} = B(\vec{X}, \vec{Y}, \vec{Z})^T = \begin{bmatrix} X_x & X_y & X_z & 0 \\ Y_x & Y_y & Y_z & 0 \\ Z_x & Z_y & Z_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ if orthonormal} \quad (6.16) \]

### 6.5 Rotation

Rotational matrices are transformation matrices that when applied to a vector rotates the vector around a rotational axis. They are closely related to basis matrices and can be used to rotate a three-dimensional model around an axis.

#### About the X-Axis

Rotating about the x-axis by an angle \( \alpha \) is done using equation 6.17. In the equation, the base vectors \( \vec{Y} \) and \( \vec{Z} \) are formed using the two trigonometric functions sine and cosine. The base vector \( \vec{X} \) is left unmodified as it is the axis about which the rotation is performed.

\[
R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (6.17)
\]

The rotation can be undone by rotating with the negative angle as seen in equation 6.18. The inverse matrix \( R_x^{-1} \) is also equal to the transpose of \( R_x \) as the set of base vectors form an orthonormal basis.

\[
R_x(\alpha)^{-1} = R_x(-\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(-\alpha) & -\sin(-\alpha) & 0 \\ 0 & \sin(-\alpha) & \cos(-\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = R_x(\alpha)^T \quad (6.18)
\]
About the Y-Axis

Rotating about the y-axis by an angle $\alpha$ is done using equation 6.19. In the equation, the base vectors $\vec{X}$ and $\vec{Z}$ are formed using the two trigonometric functions sine and cosine. The base vector $\vec{Y}$ is left unmodified as it is the axis about which the rotation is performed.

$$R_y(\alpha) = \begin{bmatrix} \cos(\alpha) & 0 & \sin(\alpha) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\alpha) & 0 & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ (6.19)

The rotation can be undone by rotating with the negative angle as seen in equation 6.20. The inverse matrix $R_y^{-1}$ is also equal to the transpose of $R_y$ as the set of base vectors form an orthonormal basis.

$$R_y(\alpha)^{-1} = R_y(-\alpha) = \begin{bmatrix} \cos(-\alpha) & 0 & \sin(-\alpha) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(-\alpha) & 0 & \cos(-\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos(\alpha) & 0 & -\sin(\alpha) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\alpha) & 0 & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = R_y(\alpha)^T$$ (6.20)

About the Z-Axis

Rotating about the z-axis by an angle $\alpha$ is done using equation 6.21. In the equation, the base vectors $\vec{X}$ and $\vec{Y}$ are formed using the two trigonometric functions sine and cosine. The base vector $\vec{Z}$ is left unmodified as it is the axis about which the rotation is performed.

$$R_z(\alpha) = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ (6.21)

The rotation can be undone by rotating with the negative angle as seen in equation 6.22. The inverse matrix $R_z^{-1}$ is also equal to the transpose of $R_z$ as the set of base vectors form an orthonormal basis.
\[ R_z(\alpha)^{-1} = R_z(-\alpha) = \begin{bmatrix}
\cos(-\alpha) & -\sin(-\alpha) & 0 & 0 \\
\sin(-\alpha) & \cos(-\alpha) & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} = \begin{bmatrix}
\cos(\alpha) & \sin(\alpha) & 0 & 0 \\
-\sin(\alpha) & \cos(\alpha) & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} = R_z(\alpha)^T \quad (6.22) \]

**About an Arbitrary Axis**

In order to rotate about an arbitrary axis, an orthonormal set of base vectors that describes the change of basis needs to be found. These base vectors can be found analytically if the process is decomposed into several rotation matrices. Equation 6.23 shows this process and the five rotational matrices involved.

\[ R(x, y, z, \alpha) = R_1^{-1} \cdot R_2^{-1} \cdot R(\alpha) \cdot R_2 \cdot R_1 \quad (6.23) \]

The first step is to rotate the rotational axis into one of the three axis-aligned planes. This is done using the \( R_1 \) matrix. The second step is to rotate the rotational axis in that plane so that it coincides with one of the two standard axes that span the plane. This is done using the \( R_2 \) matrix. The rotation is then performed as a rotation about that standard axis by an angle \( \alpha \). This is done using the \( R \) matrix. The final two steps involves inverting the first two rotations and applying them in inverse order so that the original coordinate system is restored.

The following example uses the x/z-plane as its axis-aligned plane and performs the rotation about the z-axis. Given a rotational axis with components \( x \), \( y \) and \( z \), two additional attributes \( k \) and \( l \) are computed. This is seen in equation 6.24 where \( k \) is the length of the rotational axis projected in the x/y-plane and \( l \) the full length of the vector.

\[ k = \sqrt{x^2 + y^2} \]
\[ l = \sqrt{x^2 + y^2 + z^2} \quad (6.24) \]

The projection of the rotational axis in the x/y-plane is needed as the rotation into the x/z-plane is performed about the z-axis in this example. Equation 6.25 shows this rotation where \( R_1 \) forms an orthonormal basis.
$$R_1 = \begin{bmatrix} \frac{x}{k} & \frac{y}{k} & 0 & 0 \\ \frac{-y}{k} & \frac{x}{k} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ if } k \neq 0$$  \hspace{1cm} (6.25)

As the basis is orthonormal, the inverse of $R_1$ is equal to its transpose. This is shown in equation 6.26.

$$R_1^{-1} = R_1^T = \begin{bmatrix} \frac{x}{k} & \frac{-y}{k} & 0 & 0 \\ \frac{y}{k} & \frac{x}{k} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ if } k \neq 0$$  \hspace{1cm} (6.26)

Rotating the rotational axis, now in the x/z-plane, into the z-axis is performed as rotation about the y-axis. This is shown in equation 6.27 in which $R_2$ also forms an orthonormal basis.

$$R_2 = \begin{bmatrix} \frac{z}{l} & 0 & -\frac{k}{l} & 0 \\ 0 & 1 & 0 & 0 \\ \frac{k}{l} & 0 & \frac{z}{l} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ if } l \neq 0$$  \hspace{1cm} (6.27)

The orthonormal basis provides a simple way of inverting the matrix $R_2$. Equation 6.28 shows how the inverse is equal to its transpose.

$$R_2^{-1} = R_2^T = \begin{bmatrix} \frac{z}{l} & 0 & \frac{k}{l} & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{k}{l} & 0 & \frac{z}{l} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ if } l \neq 0$$  \hspace{1cm} (6.28)

Premultiplying the two matrices $R_2$ and $R_1$ gives the result shown in equation 6.29.

$$R_2 \cdot R_1 = \begin{bmatrix} \frac{x}{kl} & \frac{y}{kl} & \frac{-z}{l} & 0 \\ \frac{-y}{k} & \frac{x}{k} & 0 & 0 \\ \frac{-z}{k} & \frac{y}{k} & \frac{-x}{l} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ if } k, l \neq 0$$  \hspace{1cm} (6.29)
As the inverses of $R_1$ and $R_2$ are equal to their respective transposes, premultiplying the two matrices $R_1^{-1}$ and $R_2^{-1}$ can be done by transposing the product of $R_2$ and $R_1$. Equation 6.30 shows this procedure and the resulting matrix.

$$R_1^{-1} \cdot R_2^{-1} = R_1^T \cdot R_2^T = (R_2 \cdot R_1)^T = \begin{bmatrix} \frac{x}{k} & \frac{y}{l} & \frac{z}{l} & 0 \\ -\frac{x}{k} & \frac{y}{l} & \frac{z}{l} & 0 \\ \frac{y}{l} & \frac{x}{l} & \frac{k}{l} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{if } k, l \neq 0 \quad (6.30)$$

Multiplying the entire chain of rotations with $R$ as a rotation about the z-axis by an angle of $\alpha$ gives a set of base vectors $\vec{X}$, $\vec{Y}$ and $\vec{Z}$. Equation 6.31 shows how these vectors are embedded into the rotational matrix.

$$R(x, y, z, \alpha) = \begin{bmatrix} X_x & Y_x & Z_x & 0 \\ X_y & Y_y & Z_y & 0 \\ X_z & Y_z & Z_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{if } l = 1 \quad (6.31)$$

In order to simplify the expressions of the base vectors, the rotational axis is assumed to be of unit length. Additionally, the attributes $k$ and $l$ were required to be non-zero for the individual rotational matrices. However, simplifying the entire transformation chain removes these criteria from the final matrix. Equation 6.32 shows the simplified expressions of the base vectors.

$$X_x = x^2 + (1 - x^2) \cdot \cos(\alpha)$$
$$X_y = x \cdot y \cdot (1 - \cos(\alpha)) + z \cdot \sin(\alpha)$$
$$X_z = x \cdot z \cdot (1 - \cos(\alpha)) - y \cdot \sin(\alpha)$$
$$Y_x = x \cdot y \cdot (1 - \cos(\alpha)) - z \cdot \sin(\alpha)$$
$$Y_y = y^2 + (1 - y^2) \cdot \cos(\alpha)$$
$$Y_z = y \cdot z \cdot (1 - \cos(\alpha)) + x \cdot \sin(\alpha)$$
$$Z_x = x \cdot z \cdot (1 - \cos(\alpha)) + y \cdot \sin(\alpha)$$
$$Z_y = y \cdot z \cdot (1 - \cos(\alpha)) - x \cdot \sin(\alpha)$$
$$Z_z = z^2 + (1 - z^2) \cdot \cos(\alpha) \quad (6.32)$$

Inverting the compound matrix can be done in one of two ways. Either through a rotation by the negative angle or by transposing the compound matrix. This is shown in equation 6.33.
\[ R(x, y, z, \alpha)^{-1} = R(x, y, z, -\alpha) = R(x, y, z, \alpha)^T = \begin{bmatrix} X_x & X_y & X_z & 0 \\ Y_x & Y_y & Y_z & 0 \\ Z_x & Z_y & Z_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ if } l = 1 \]

(6.33)

6.6 Projection

Adjusting the perspective for a three-dimensional vector when viewed by a camera or an observer is one of the fundamental processes in three-dimensional computer graphics. Objects located near the observer appear bigger than identical objects located further away. The projection process is often modeled using matrices, despite the inability of performing this process solely by applying a matrix to a vector.

The projection operator is dependent on the actual components of the vector it is to project, making it impossible to model using matrices. However, projection can be achieved by introducing a concept known as the homogeneous divide. Equation 6.34 shows a simple projection matrix.

\[ P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \]  

(6.34)

Applying this matrix to a four-dimensional homogeneous vector \((x, y, z, 1)\) results in the vector \((x, y, z, z)\). As seen, its fourth component is now possibly different from one. By applying the dividing operator which divides each component by the fourth component, the vector becomes \((\frac{x}{z}, \frac{y}{z}, 1, 1)\). It is now homogenized with respect to the fourth component and perspectively projected into the axis-aligned plane at \(z = 1\).

It should be emphasized that the projection matrix is irreversible as its determinant is equal to zero.

Perspective projection is better viewed as a geometric operation. Figure 6.2 shows a vector \(\vec{P}\) and the orthonormal basis \(\{\vec{X}, \vec{Y}, \vec{Z}\}\). Deriving an expression for projecting the vector into the axis-aligned plane at \(z = 1\) can be done using the properties of similar triangles.
Equation 6.35 shows the relationship between the vector $\vec{P}$ and its perspective projected counterpart $\vec{P}'$. As seen in the equation, the result is equal to that produced when using matrices and the homogeneous division operator.

$$
\begin{align*}
\frac{P'_x}{1} &= \frac{P_x}{P_z - 1 + 1} \iff P'_x = \frac{P_x}{P_z} \\
\frac{P'_y}{1} &= \frac{P_y}{P_z - 1 + 1} \iff P'_y = \frac{P_y}{P_z} \\
\frac{P'_z}{1} &= \frac{P_z}{P_z - 1 + 1} \iff P'_z = 1
\end{align*}
$$

(6.35)

7 Line Segment-Plane Intersection

There are three possible outcomes for the intersection between a line and a plane. The line may be embedded into the plane, producing infinitely many intersection points. It may be located a distance from the plane but its direction perpendicular to the normal of the plane. For this case, no intersection points exist. For all other cases, there is exactly one intersection point.

Line segments require that the two points forming the line are located on opposite sides of the plane in order to produce exactly one intersection point. This section is dedicated to locating that unique intersection point, should it exist.

A plane is defined through a plane normal $\vec{N}$ and the minimum distance to that plane $d$. Equation 7.1 shows how the signed distance between a plane $P$ and a vector $\vec{X}$ can be computed as a scalar projection between the normal and the vector and by subtracting the minimum distance. Naturally, the plane is located where the distance is zero.
\[ \text{dist}(P, \vec{X}) = \vec{N} \cdot \vec{X} - d \] (7.1)

Line segments are composed of a starting point \( \vec{S} \) and an ending point \( \vec{E} \). Determining whether the line segment intersects a plane is done by evaluating the signed distances to the two points. If the two signed distances are of opposite signs, the line segment will intersect the plane at a point \( \vec{I} \). For all other cases, no intersection point exists. Figure 7.1 shows a line segment intersecting a plane.

![Figure 7.1: A line segment intersecting a plane from the positive half-space.](image)

From the figure, a number of relationships can be observed through the use of similar triangles. Equation 7.2 shows these relationships.

\[ \frac{a}{\text{dist}(P, \vec{S})} = \frac{b}{-\text{dist}(P, \vec{E})} = \frac{a + b}{\text{dist}(P, \vec{S}) - \text{dist}(P, \vec{E})} \] (7.2)

Finding the intersection point is essentially the process of scaling the direction vector \( \vec{S} \) to \( \vec{E} \) by an amount proportional to the two signed distances and adding it to the starting point. Equation 7.3 shows how the intersection point is found where \( t_1 \) is a fraction of the length of the direction vector.

\[ \vec{I} = \vec{S} + t_1 \cdot (\vec{E} - \vec{S}) \] (7.3)

The fraction \( t_1 \) is computed using the previously observed relationships. Equation 7.4 shows the definition of this fraction.
The line segment may just as well start in the negative half-space of the plane and end in the positive half-space. Though the intersection point may be found using the previously defined equations in a mathematical context, it will depend on the orientation in a computational context. Due to rounding errors, the intersection point may be slightly different if the line segment starts in the negative half-plane compared to when it starts in the positive half-plane.

In a computational context, the interpolation scheme should be defined in a consistent fashion. That is, the fractional $t$ should always be defined as either the fraction of the directional vector from the inside point to the outside point or the other way around. Figure 7.2 shows a line segment intersecting a plane from the outside and in.

The relationships found using similar triangles are defined slightly different for this case. Equation 7.5 shows these relationships.

$$\frac{a}{\text{dist}(P, E)} = \frac{b}{\text{dist}(P, S)} = \frac{a + b}{\text{dist}(P, E) - \text{dist}(P, S)}$$  \hspace{1cm} (7.5)

As with the relationships, the interpolation scheme is slightly different. Equation 7.6 shows how interpolation is done when the line segment starts in the negative half-space.

$$\vec{I} = \vec{E} + t_2 \cdot (\vec{S} - \vec{E})$$  \hspace{1cm} (7.6)
The fraction \( t_2 \) of the negative direction vector \( \vec{E} \) to \( \vec{S} \) is defined as shown in equation 7.7 using the previously observed relationships.

\[
t_2 = \frac{a}{a + b} = \frac{\text{dist}(P, \vec{E})}{\text{dist}(P, \vec{E}) - \text{dist}(P, \vec{S})}
\]

\( (7.7) \)

8 View Frustum Clipping

The view frustum or view volume is a sub-space of the virtual scene in which triangles are partially or completely visible when projected onto the screen. It is formed from six planes, defined by the orientation of a camera or an observer and a few additional parameters such as draw distance. Its shape is in the form of a truncated pyramid as seen from above in figure 8.1. Four of the six planes and their corresponding normals are shown in the figure where \( \vec{N}_l, \vec{N}_r, \vec{N}_n \) and \( \vec{N}_f \) are the normals of the left, right, near and far planes.

![Figure 8.1: The view frustum is in the shape of a truncated pyramid.](image)

View frustum clipping is the process of removing complete triangles or parts of triangles which will not contribute to the final image when projected onto the screen. This often in order to save computational power.

8.1 Clipping Planes

By transforming the triangles into camera space before clipping, the definition of the six planes forming the view frustum will become simplified. Equation 8.1 shows the equations of the left, right, bottom, top, near and far planes, respectively. All equations are defined in camera space.
\begin{align*}
  (\alpha_x, 0, 1) \cdot \vec{X} &= 0 \\
  (-\alpha_x, 0, 1) \cdot \vec{X} &= 0 \\
  (0, \alpha_y, 1) \cdot \vec{X} &= 0 \\
  (0, -\alpha_y, 1) \cdot \vec{X} &= 0 \\
  (0, 0, 1) \cdot \vec{X} &= d_n \\
  (0, 0, -1) \cdot \vec{X} &= -d_f
\end{align*}
(8.1)

In the equation, \( d_n \) and \( d_f \) are the near and far draw distances and \( \vec{X} \) an arbitrary point. Equation 8.2 states a requirement on the relationship between the near and far draw distances. As seen in the equation, non-positive draw distances are not allowed and the far draw distance must be greater than the near draw distance.

\[
0 < d_n < d_f < \infty
\] (8.2)

The two constants \( \alpha_x \) and \( \alpha_y \) are stretch factors which are determined from the aspect ratio of the target image. This in order to account for non-square displays by extending the field of view of the camera along the major axis. As such, the field of view of the camera becomes defined along the minor axis of the target image.

Equation 8.3 shows how the horizontal stretch factor \( \alpha_x \) is determined from the width and height of the target image.

\[
\alpha_x = \begin{cases} 
  \frac{h}{w} & \text{if } h < w \\
  1 & \text{otherwise}
\end{cases}
\] (8.3)

The vertical stretch factor \( \alpha_y \) is determined in a similar way as shown in equation 8.4.

\[
\alpha_y = \begin{cases} 
  1 & \text{if } h < w \\
  \frac{w}{h} & \text{otherwise}
\end{cases}
\] (8.4)

The six plane equations are fully determined from the equations above and the clipping of triangles in camera space can be performed.
8.2 Example

The clipping procedure is done in sequence for each of the six planes and operates on a vertex list as clipping a triangle against a plane may produce additional vertices. The vertex list is initialized with the three original vertices of a triangle and may store up to four additional vertices. In addition, a secondary vertex list is used to store the results of a clipping pass.

Figure 8.2 shows a triangle which intersects the view frustum and its three initial vertices $\vec{V}_1$, $\vec{V}_2$ and $\vec{V}_3$.

Clipping against the left plane produces an additional vertex and the vertex list now contains a total of four vertices. This is shown in figure 8.3.

The clipping procedure may process the six planes in any order as long as the order is consistent for all triangles. The triangle will eventually become clipped against the right plane which does not produce any additional vertices for this
example. However, the vertices are moved to the respective intersection points as shown in figure 8.4.

Figure 8.4: Clipping against the right plane moves two of the vertices.

After the clipping process has completed, the final vertex list may not form a single triangle. It is possible for the process to produce as much as seven vertices, which together form a planar polygon. This polygon can be triangulated into several triangles and the number of triangles determined from the expression in equation 8.5. In the equation, $t$ is the number of triangles and $v$ the number of vertices. As the clipping process produces at most seven vertices, the number of triangles is at most five.

$$t = \max(v - 2, 0) \quad (8.5)$$

For this example, the polygon is split up into two triangles. Figure 8.5 shows the triangulated polygon and the two triangles $\{\vec{V}_1, \vec{V}_2, \vec{V}_3\}$ and $\{\vec{V}_1, \vec{V}_3, \vec{V}_4\}$.

Figure 8.5: The planar polygon can be triangulated into several triangles.
8.3 Algorithm

The Sutherland-Hodgman algorithm for line clipping can be used to clip each line segment implicitly defined by the vertex list against each of the six planes of the view frustum. The algorithm was introduced in 1974 by Sutherland and Hodgman in [7].

For every vertex in the vertex list, the signed distance to the current plane $P$ is computed. The signed distances of the starting point $\vec{S}$ and ending point $\vec{E}$ are used to determine how the line segment is oriented with respect to the plane. This is done through two boolean tests for which the results are embedded into a bit mask. Two tests give four cases in total, all of which are shown in equation 8.6 [7].

$$\text{case} = \begin{cases} 
0 & \text{if dist}(P, \vec{S}) < 0 \land \text{dist}(P, \vec{E}) < 0 \\
1 & \text{if dist}(P, \vec{S}) > 0 \land \text{dist}(P, \vec{E}) < 0 \\
2 & \text{if dist}(P, \vec{S}) < 0 \land \text{dist}(P, \vec{E}) > 0 \\
3 & \text{if dist}(P, \vec{S}) > 0 \land \text{dist}(P, \vec{E}) > 0 
\end{cases} \quad (8.6)$$

As seen in the equation, case 0 occurs when both the starting and ending points are located in the negative half-space created by the plane. Neither the points nor the line segment is visible and as such, nothing should be appended to the resulting vertex list. This is illustrated in figure 8.6.

![Figure 8.6: Illustration of case 0 for the Sutherland-Hodgman algorithm.](image)

If the starting point is located in the positive half-space and the ending point located in the negative half-space, case 1 occurs. This is illustrated in figure 8.7. For this case, the intersection point $\vec{I}$ needs to be computed. The intersection point is computed with respect to the orientation of the line segment as discussed in the section about line segment-plane intersections. As the original
starting point is visible, it is appended to the resulting vertex list followed by the intersection point.

Conversely, case 2 occurs when the starting point is located in the negative half-space and the ending point located in the positive half-space. For this case, the intersection point is computed and appended to the resulting vertex list. This is illustrated in figure 8.8.

The final case occurs when both points are located in the positive half-space. For this case, no intersection point exists. The only thing appended to the resulting vertex list is the original starting point. The case is illustrated in figure 8.9.
Figure 8.9: Illustration of case 3 for the Sutherland-Hodgman algorithm.

The clipping against the current plane $P$ is completed by clipping the final line segment between the last vertex and the first vertex in the same fashion as for the other line segments. The resulting vertex list describes the intermediate planar polygon and can be used as input for the next clipping plane and so forth until all clipping planes have been processed.

9 Barycentric Coordinates

For three-dimensional computer graphics, a triangle is formed from three vertices located in three-dimensional space. If the vertices are arranged so that a proper triangle is formed, the triangle will create a two-dimensional sub-space. This sub-space is a plane and the triangle is the small fraction of that plane enclosed within the convex hull of the three vertices. Figure 9.1 shows a triangle and its three vertices $\vec{U}$, $\vec{V}$ and $\vec{W}$.

Figure 9.1: A triangle is formed from three vertices.
If the plane is divided by the three lines formed when extending the edges infinitely, the triangle can be expressed as the union of the three positive half-spaces. This can be seen in figure 9.2 where the arrows indicate the positive half-spaces.

![Figure 9.2: A triangle can be seen as the union of three positive half-spaces.](image)

Attributes are commonly defined at the three vertices of the triangle and may be shared with adjacent triangles. In order to interpolate these values across the entire surface, a clever coordinate system is needed.

Barycentric coordinates are a set of three coordinates that can be used to refer to all points within the plane spanned by a proper triangle. This set of coordinates is often denoted \((u, v, w)\) and specifies the normalized perpendicular distances between the opposite edge of a vertex and the vertex itself.

Let an arbitrary point \(\vec{P}\) be defined in the plane by the three coordinates \((u, v, w)\). Equation 9.1 shows two fundamental properties for this point and for the barycentric coordinates.

\[
\vec{P} = \vec{U} \cdot u + \vec{V} \cdot v + \vec{W} \cdot w \\
1 = u + v + w
\]  

(9.1)

The \(u\) coordinate is defined by the normalized perpendicular distance from the edge between vertices \(\vec{V}\) and \(\vec{W}\). The \(v\) and \(w\) coordinates are defined in a similar fashion as illustrated in figure 9.3.
As the coordinates are normalized with respect to the size of the triangle, the coordinates gain a series of useful properties. If the point $\vec{P}$ coincides with vertex $\vec{U}$, its position in the barycentric coordinate system is $(1, 0, 0)$. The same is true for vertices $\vec{V}$ and $\vec{W}$, which are located at $(0, 1, 0)$ and $(0, 0, 1)$, respectively.

If the point lies anywhere along the line between $\vec{V}$ and $\vec{W}$, its coordinates will be $(0, v, w)$. Conversely, its coordinates along the other two lines will be $(u, 0, w)$ or $(u, v, 0)$. The normalization also ensures that the sum of all three coordinates is exactly equal to one, regardless of the position of the point. This is always true, even if the point lies outside of the triangle.

This set of coordinates and their properties form an efficient interpolation scheme for attributes defined at the three vertices.

9.1 Cramer’s Rule

The fundamental properties of the barycentric coordinates form a system of equations and can be expressed in matrix form. An input vector $\vec{X}$ is transformed by a matrix $M$ into an output vector $\vec{Y}$. Equation 9.2 shows a general system of equations in matrix form.

$$\vec{Y} = M \cdot \vec{X}$$  \hspace{1cm} (9.2)

The input vector will be the three barycentric coordinates $(u, v, w)$ and the matrix and output vector will correspond to the statements from equation 9.1. Expressing these statements in matrix form results in equation 9.3 where computing the barycentric coordinates is analogous to solving the equation for $(u, v, w)$. 

![Figure 9.3: The isolines of the barycentric coordinates are parallel to the edges.](image-url)
\[
\begin{bmatrix}
P_x \\
P_y \\
1
\end{bmatrix} =
\begin{bmatrix}
U_x & V_x & W_x \\
U_y & V_y & W_y \\
1 & 1 & 1
\end{bmatrix}
\begin{bmatrix}
u \\
v \\
w
\end{bmatrix}
\tag{9.3}
\]

Cramer’s Rule states that any system of equations with as many unknowns as there are equations (i.e. square matrices) can be solved using quotients of determinants. Equation 9.4 states that the unknown \( X_i \) can be expressed as the fraction between the determinant of the matrix \( M_i \) and that of the original matrix \( M \).

\[
X_i = \frac{\det(M_i)}{\det(M)} \text{ if } \det(M) \neq 0
\tag{9.4}
\]

The determinant of the system is computed as shown in equation 9.5. The bottom row simplifies the calculations as it is equal to \((1, 1, 1)\).

\[
\det(M) = (V_x \cdot W_y - W_x \cdot V_y) - (U_x \cdot W_y - W_x \cdot U_y) + (U_x \cdot V_y - V_x \cdot U_y)
\]

\[
= (V_x - U_x) \cdot (W_y - U_y) - (W_x - U_x) \cdot (V_y - U_y)
\tag{9.5}
\]

The additional matrices \( M_i \) are formed by replacing column \( i \) of the matrix \( M \) with the vector \( \vec{Y} \). Equation 9.6 shows this matrix.

\[
M_1 =
\begin{bmatrix}
P_x & V_x & W_x \\
P_y & V_y & W_y \\
1 & 1 & 1
\end{bmatrix}
\tag{9.6}
\]

Computing the determinant of \( M_1 \) is performed in the standard way and results in the expression seen in equation 9.7.

\[
\det(M_1) = (V_x \cdot W_y - W_x \cdot V_y) - (P_x \cdot W_y - W_x \cdot P_y) + (P_x \cdot V_y - V_x \cdot P_y)
\]

\[
= (P_x - W_x) \cdot (V_y - W_y) - (V_x - W_x) \cdot (P_y - W_y)
\tag{9.7}
\]

The matrix \( M_2 \) is formed by replacing the second column of \( M \) with the vector \( \vec{Y} \). Equation 9.8 shows this matrix.
\[ M_2 = \begin{bmatrix} U_x & P_x & W_x \\ U_y & P_y & W_y \\ 1 & 1 & 1 \end{bmatrix} \] (9.8)

Its determinant is similar to the determinant of \( M_1 \). Equation 9.9 shows an expression for the determinant of \( M_2 \).

\[
\det(M_2) = (P_x \cdot W_y - W_x \cdot P_y) - (U_x \cdot W_y - W_x \cdot U_y) + (U_x \cdot P_y - P_x \cdot U_y)
\]
\[
= (P_x - W_x) \cdot (W_y - U_y) - (W_x - U_x) \cdot (P_y - W_y)
\] (9.9)

The final matrix \( M_3 \) is formed by replacing the third column of \( M \) with the vector \( \vec{Y} \). This is seen in equation 9.10.

\[ M_3 = \begin{bmatrix} U_x & V_x & P_x \\ U_y & V_y & P_y \\ 1 & 1 & 1 \end{bmatrix} \] (9.10)

An expression for the determinant of the matrix \( M_3 \) is shown in equation 9.11.

\[
\det(M_3) = (V_x \cdot P_y - P_x \cdot V_y) - (U_x \cdot V_y - V_x \cdot U_y) + (U_x \cdot V_y - V_x \cdot U_y)
\]
\[
= (V_x - U_x) \cdot (P_y - U_y) - (P_x - U_x) \cdot (V_y - U_y)
\] (9.11)

Using Cramer’s Rule, the three unknowns \((u, v, w)\) can now be expressed as the respective quotients. Equation 9.12 shows the expression for the three barycentric coordinates.

\[
(u, v, w) = \frac{1}{\det(M)} \cdot (\det(M_1), \det(M_2), \det(M_3)) \text{ if } \det(M) \neq 0
\] (9.12)

### 9.2 Ratios of Areas

Barycentric coordinates can also be viewed as area-oriented coordinates. The coordinates \( \left( \frac{1}{3}, \frac{1}{3}, \frac{1}{3} \right) \) will always correspond to the center of any proper triangle. It is therefore possible to define the coordinates as ratios of areas. Figure 9.4
shows the three areas formed by the three vertices $\vec{U}$, $\vec{V}$, $\vec{W}$ and an interior point $\vec{P}$.

![Figure 9.4: Barycentric coordinates can be defined through ratios of areas.](image)

The area of the entire triangle is computed as shown in equation 9.13. This will become the normalizing factor.

$$A = \frac{(V_y - U_y) \cdot (W_y - U_y) - (W_x - U_x) \cdot (V_y - U_y)}{2} \quad (9.13)$$

The three areas formed by introducing the interior point $\vec{P}$ are computed in a similar fashion. Equation 9.14 shows this process in which $A_w$ is defined by the other two areas $A_u$ and $A_v$ in conjunction with the entire area $A$. Naturally, $A_w$ could be defined explicitly but the computational complexity of the method would be higher.

$$A_u = \frac{(P_x - W_x) \cdot (V_y - W_y) - (V_x - W_x) \cdot (P_y - W_y)}{2}$$

$$A_v = \frac{(P_x - W_x) \cdot (W_y - U_y) - (W_x - U_x) \cdot (P_y - W_y)}{2} \quad (9.14)$$

$$A_w = A - A_u - A_v$$

The barycentric coordinates are obtained through a division by the total area. This is shown in equation 9.15 where the actual coordinates $(u, v, w)$ are computed. For efficiency reasons, the total area can be precalculated, stored and reused in every calculation for that triangle. It is also unnecessary to divide every area by two as it is the relative areas that are of importance.

$$(u, v, w) = \frac{1}{A} (A_u, A_v, A_w) \text{ if } A \neq 0 \quad (9.15)$$
It can be observed that the two partial areas $A_u$ and $A_v$ actually correspond to half the signed perpendicular distances to the respective edges. That is, a projection of the relative vector from $\vec{W}$ to $\vec{P}$ on the two edge normals containing $\vec{W}$. Should the point $\vec{P}$ be located outside of the triangle, some areas and coordinates will become negative. This in accordance with the previous observation.

10 The Separating Axis Theorem

The hyper-plane separation theorem is often referred to as the separating axis theorem in two dimensions. The theorem can be applied to determine if two convex shapes are overlapping and is heavily used in collision detection for computer graphics.

10.1 Candidate Axes

If the two shapes are defined by vertices and the connecting edges in between, there exists a finite number of axes along which the two objects can be separated. In two dimensions, these candidate axes correspond to the edge normals present in either of the two shapes. Figure 10.1 shows two convex shapes and their edge normals.

Figure 10.1: A box and a triangle with their corresponding edge normals.

Figure 10.2 shows the connection between the edge normals of the two shapes and the axes for which a separation is possible. There are seven candidate axes for the two shapes in the figure but four of them are duplicates, leaving three unique axes.
Note that the separating axes are directionless. Two edge normals differing only in magnitude will correspond to the same separating axis, even if the magnitudes have opposite signs. This as it is the relative magnitudes of the projected shapes that are compared using the theorem.

### 10.2 Collision Detection

The theorem states that if two shapes are separated, there must exist at least one axis for which the projections of the two shapes do not overlap. If the two objects overlap, no such axis will exist.

Figure 10.3 shows a case where two shapes do not overlap and how this is reflected in the projections.

This case can be determined using the condition in equation 10.1 in which $B_i$ and $T_i$ are the sets of projected vertices of the box and the triangle along candidate axis $i$. 

Figure 10.2: Candidate axes are formed from unique edge normals.

Figure 10.3: Two convex shapes with one separating axis (diagonal).
\[ \exists i : (\max(B_i) < \min(T_i) \lor \min(B_i) > \max(T_i)) \quad (10.1) \]

Figure 10.4 demonstrates a different case where the two shapes partially overlap. The overlap is also visible in the projections of the two shapes.

Figure 10.4: Two convex shapes with no separating axis.

This case can be determined using the condition in equation 10.2. The sets of projected vertices of the box and of the triangle along candidate axis \( i \) are denoted by \( B_i \) and \( T_i \), respectively.

\[ \forall i : (\max(B_i) > \min(T_i) \land \min(B_i) < \max(T_i)) \quad (10.2) \]

An overlooked feature of the separating axis theorem is the possibility to determine if a shape is completely enclosed within another shape. If the projection of the smaller shape is enclosed in the projection of the larger shape for every candidate axis, the smaller shape will be completely enclosed inside the larger shape.

Figure 10.5 demonstrates this feature. All projections of the smaller shape are enclosed within the projections of the larger shape.
This case can be determined using the condition in equation 10.3. As with the other conditions, $B_i$ and $T_i$ denotes the sets of projected vertices of the box and of the triangle along candidate axis $i$, respectively.

$$\forall i : (\min(B_i) > \min(T_i) \land \max(B_i) < \max(T_i))$$ (10.3)

### 11 Tangent Space

When the vertices of a triangle are coupled with texture coordinates, a local coordinate system known as the tangent space must exist for that triangle. Its bases are the axes along which the texture coordinates are defined as well as the normal of the triangle. A common notation for these axes are $\vec{T}$, $\vec{B}$ and $\vec{N}$, corresponding to the tangent, bitangent and normal, respectively. However, the notation $\vec{S}$, $\vec{T}$ and $\vec{N}$ will be used from here on as texture coordinates often are denoted by $s$ and $t$.

Figure 11.1 shows a triangle and its three vertices $\vec{U}$, $\vec{V}$ and $\vec{W}$. It also shows two of the tangent space bases, $\vec{S}$ and $\vec{T}$. These bases can be of arbitrary orientation, depending on the different texture coordinates defined at the three vertices but are always located in the same plane as the triangle.
Figure 11.1: A tangent space exists for triangles with proper texture coordinates.

For any point $\vec{P}$ in the same plane as a triangle, it is required for the tangent space of that triangle to be consistent with the spatial coordinates and with the texture coordinates of the point. Equation 11.1 shows this relationship.

$$(P_{xyz} - U_{xyz}) = (P_s - U_s) \cdot \vec{S} + (P_t - U_t) \cdot \vec{T}$$  \hspace{1cm} (11.1)

The relationship is illustrated in figure 11.2 in which the point $\vec{P}$ local to vertex $\vec{U}$ is described using the tangent space bases and the relative texture coordinates.

Figure 11.2: The tangent space is a coordinate system local to the triangle.

Defining the tangent space from the attributes of the three vertices can be done through observing an important feature. The vertices $\vec{V}$ and $\vec{W}$ relative to $\vec{U}$ must be uniquely described using the tangent space bases and the relative texture coordinates. Equation 11.2 shows these criteria.
\begin{equation}
(V^{xyz} - U^{xyz}) = (V_s - U_s) \cdot \vec{S} + (V_t - U_t) \cdot \vec{T} \tag{11.2}
\end{equation}

\begin{equation}
(W^{xyz} - U^{xyz}) = (W_s - U_s) \cdot \vec{S} + (W_t - U_t) \cdot \vec{T} \tag{11.2}
\end{equation}

These criteria can be simplified by introducing two local vectors \( \vec{A} \) and \( \vec{B} \) as the vectors from \( \vec{U} \) to \( \vec{V} \) and \( \vec{U} \) to \( \vec{W} \), respectively. The two vectors will correspond to the two edges local to \( \vec{U} \) as shown in figure 11.3.

![Figure 11.3: Two vectors \( \vec{A} \) and \( \vec{B} \) local to \( \vec{U} \) are introduced.](image)

The previous equation can now be rewritten as shown in equation 11.3. This is a system of six equations with six unknowns.

\begin{equation}
\begin{bmatrix}
A_x \\
A_y \\
A_z \\
B_x \\
B_y \\
B_z \\
\end{bmatrix} = \begin{bmatrix}
A_s & A_t \\
B_s & B_t \\
S_x & S_y & S_z \\
T_x & T_y & T_z \\
\end{bmatrix} \tag{11.3}
\end{equation}

Obtaining the tangent space bases is analogous to solving this system of equations for the two bases \( \vec{S} \) and \( \vec{T} \). If the system of equations is rewritten in matrix form, equation 11.4 is obtained.

\begin{equation}
\begin{bmatrix}
A_x & A_y & A_z \\
B_x & B_y & B_z \\
\end{bmatrix} = \begin{bmatrix}
A_s & A_t \\
B_s & B_t \\
\end{bmatrix} \begin{bmatrix}
S_x \\
S_y \\
S_z \\
T_x \\
T_y \\
T_z \\
\end{bmatrix} \tag{11.4}
\end{equation}

All attributes belonging to either of the two vectors \( \vec{A} \) and \( \vec{B} \) are known. It is therefore possible to determine whether there exists a tangent space. A matrix \( M \) consisting of the texture coordinates is defined as shown in equation 11.5.

\begin{equation}
M = \begin{bmatrix}
A_s & A_t \\
B_s & B_t \\
\end{bmatrix} \tag{11.5}
\end{equation}
This matrix needs to be inverted in order to compute the tangent space bases. As with other matrices, it is only invertible if its determinant is not equal to zero. Using the inverse formula for square four-element matrices produces the result in equation 11.6.

\[
M^{-1} = \frac{1}{A_s \cdot B_t - A_t \cdot B_s} \begin{bmatrix} B_t & -A_t \\ -B_s & A_s \end{bmatrix} \text{ if } A_s \cdot B_t - A_t \cdot B_s \neq 0 \tag{11.6}
\]

The matrix equation can be solved for \( \vec{S} \) and \( \vec{T} \) by multiplying equation 11.4 with \( M^{-1} \) from the left. Rearranging the equation gives the result shown in equation 11.7.

\[
\begin{bmatrix} S_x & S_y & S_z \\ T_x & T_y & T_z \end{bmatrix} = M^{-1} \begin{bmatrix} A_x & A_y & A_z \\ B_x & B_y & B_z \end{bmatrix} \tag{11.7}
\]

The explicit definitions of the two tangent space bases are shown in equation 11.8 where \( M^{-1}_{i,j} \) corresponds to the element of the inverse texture coordinate matrix at row \( i \) and column \( j \).

\[
\vec{S} = \begin{bmatrix} M^{-1}_{1,1} \cdot A_x + M^{-1}_{1,2} \cdot B_x \\ M^{-1}_{1,1} \cdot A_y + M^{-1}_{1,2} \cdot B_y \\ M^{-1}_{1,1} \cdot A_z + M^{-1}_{1,2} \cdot B_z \end{bmatrix} \\
\vec{T} = \begin{bmatrix} M^{-1}_{2,1} \cdot A_x + M^{-1}_{2,2} \cdot B_x \\ M^{-1}_{2,1} \cdot A_y + M^{-1}_{2,2} \cdot B_y \\ M^{-1}_{2,1} \cdot A_z + M^{-1}_{2,2} \cdot B_z \end{bmatrix} \tag{11.8}
\]

The normal is by definition computed as the three-dimensional cross-product between two of the edges and is not connected to the texture coordinates. The two edges must be selected in accordance with the order in which the vertices are defined. Throughout this report, left-handed coordinate systems are used requiring the vertices to be defined in clock-wise order. Equation 11.9 shows the definition of the normal \( \vec{N} \) using the two local vectors \( \vec{A} \) and \( \vec{B} \).

\[
\vec{N} = \begin{bmatrix} B_z \cdot A_y - B_y \cdot A_z \\ B_x \cdot A_z - B_z \cdot A_x \\ B_y \cdot A_x - B_x \cdot A_y \end{bmatrix} \tag{11.9}
\]
12 Perspective-Correct Interpolation

When three-dimensional triangles are perspectively projected into a plane, the relationship between the attributes associated with each vertex becomes distorted. In three dimensions, the relationship is linear over the surface of the triangle. This is no longer true when the triangle is projected into a plane. However, the inverse projection is linear and this can be utilized.

This section provides details on how to correctly interpolate attributes across a projected triangle using the two texture coordinates $s$ and $t$ as an example. The method detailed here is in no way limited to texture coordinates. Any attributes may be interpolated using the method, even vectors and matrices.

Equation 12.1 shows the projected texture coordinates $s'$ and $t'$ and the inverse depth $z'$, all of which are linear over the surface of the projected triangle.

$$z' = \frac{1}{z}$$
$$s' = \frac{s}{z}$$
$$t' = \frac{t}{z}$$

(12.1)

These three projected attributes can be computed at the three vertices $\vec{U}$, $\vec{V}$ and $\vec{W}$ of the triangle, providing a set of nine projected attributes. This is shown in equation 12.2.

$$z'_U = \frac{1}{z_U} \quad z'_V = \frac{1}{z_V} \quad z'_W = \frac{1}{z_W}$$
$$s'_U = \frac{s_U}{z_U} \quad s'_V = \frac{s_V}{z_V} \quad s'_W = \frac{s_W}{z_W}$$
$$t'_U = \frac{t_U}{z_U} \quad t'_V = \frac{t_V}{z_V} \quad t'_W = \frac{t_W}{z_W}$$

(12.2)

An arbitrary point $\vec{P}$ on the projected surface is defined by the three barycentric coordinates $u$, $v$ and $w$ as stated previously. Equation 12.3 shows this point.

$$\vec{P} = (u, v, w)$$

(12.3)

As the nine projected attributes are linear over the surface, the barycentric coordinates can be used to linearly interpolate these to the point $\vec{P}$. This procedure is shown in equation 12.4.
\[ z'_P = z'_U \cdot u + z'_V \cdot v + z'_W \cdot w \]
\[ s'_P = s'_U \cdot u + s'_V \cdot v + s'_W \cdot w \]
\[ t'_P = t'_U \cdot u + t'_V \cdot v + t'_W \cdot w \]

Equation 12.5 shows how the texture coordinates \( s_P \) and \( t_P \) are recovered at the point together with the correct depth value \( z_P \).

\[ z_P = \frac{1}{z'_P} \]
\[ s_P = \frac{s'_P}{z_P} \]
\[ t_P = \frac{t'_P}{z_P} \]

13 Rounding Modes

Proper handling of the rounding of real-valued numbers is an important aspect of computer graphics. Real-valued numbers are used to describe an array of different properties of a three-dimensional scene, ranging from geometry definitions to texture coordinates. These will have to map to a discrete raster of discrete colors sampled from discrete textures, making the need evident.

Rounding can be performed in a number of different modes where each mode is beneficial in a number of different situations. These rounding modes are implemented in most programming languages and in hardware but can also be defined using the truncating function. The truncating function truncates all decimals from a real-valued number \( x \). Figure 13.1 shows this function and how it rounds its input towards zero.
Defining the truncating function in a computational context can be done in a few different ways. Either as a bit manipulation of the real-valued number or as a type conversion to an integer data type. This provided that the integer data type used is able to represent the truncated value.

### 13.1 Flooring

Flooring is a rounding mode which rounds its input towards negative infinity. A real-valued number $x$ is rounded towards the integer to the left on the x-axis. This is shown in figure 13.2.

The flooring function can be defined as shown in equation 13.1 in which the definition properly handles positive as well as negative values.

$$\text{floor}(x) \in \mathbb{Z} = \begin{cases} 
\text{trunc}(x) - 1 & \text{if } x < \text{trunc}(x) \\
\text{trunc}(x) & \text{otherwise}
\end{cases} \tag{13.1}$$
**Fractional**

The flooring function has a complimentary function known as the fractional function. It represents the value that was removed using the flooring function and is shown in figure 13.3.

![Figure 13.3: The fractional function of a real-valued number.](image)

The fractional function can be defined using the flooring function or using the truncating function. Equation 13.2 shows the fractional function defined using the latter. As with the flooring function, this definition properly handles positive as well as negative values.

\[
\text{frac}(x) \in \mathbb{R} : [0, 1] = \begin{cases} 
    x - \text{trunc}(x) + 1 & \text{if } x < \text{trunc}(x) \\
    x - \text{trunc}(x) & \text{otherwise}
\end{cases} \quad (13.2)
\]

The two functions obey an important relationship. The sum of the flooring function and the fractional function individually applied to any real-valued number \(x\) is always equal to \(x\) as shown in equation 13.3.

\[
x = \text{floor}(x) + \text{frac}(x) \quad (13.3)
\]

The sum of the two functions is as a continuous line as expected. This is shown in figure 13.4.
13.2 Ceiling

Ceiling is a rounding mode which rounds its input towards positive infinity. A real-valued number $x$ is rounded towards the integer to the right on the x-axis. This is shown in figure 13.5.

The ceiling function can be defined as shown in equation 13.4 in which the definition properly handles positive as well as negative values.

$$
\text{ceil}(x) \in \mathbb{Z} = \begin{cases} 
\text{trunc}(x) + 1 & \text{if } x > \text{trunc}(x) \\
\text{trunc}(x) & \text{otherwise}
\end{cases} \quad (13.4)
$$
Reverse Fractional

Similar to the flooring function, the ceiling function has a complimentary function known as the reverse fractional function. It represents the value that was introduced using the ceiling function and is shown in figure 13.6.

![Figure 13.6: The reverse fractional function of a real-valued number.](image)

The reverse fractional function can be defined using the ceiling function or using the truncating function. Equation 13.5 shows the reverse fractional function defined using the latter. As with the other functions, this definition properly handles positive as well as negative values.

\[
\text{rfrac}(x) \in \mathbb{R} : [0, 1] = \begin{cases} 
\text{trunc}(x) + 1 - x & \text{if } x > \text{trunc}(x) \\
\text{trunc}(x) - x & \text{otherwise}
\end{cases} \quad (13.5)
\]

The two functions obey a relationship similar to that of the flooring and fractional functions. The difference between the ceiling function and the reverse fractional function individually applied to any real-valued number \(x\) is always equal to \(x\). This is shown in equation 13.6. This difference also forms a continuous line.

\[
x = \text{ceil}(x) - \text{rfrac}(x) \tag{13.6}
\]

13.3 Nearest

Rounding to the nearest integer can be performed in a simple way utilizing the flooring or ceiling functions. Shifting the input by \(\frac{1}{2}\) before flooring will create a rounding mode that produces the closest integer, left or right on the
x-axis. Values at the exact midpoint between two integers will be rounded upwards. Conversely, shifting the input by $-\frac{1}{2}$ before ceiling will also produce a rounding towards the nearest integer. However, values at the midpoints will be rounded downwards.
Chapter D

Texture Filtering

Texture mapping plays an important role in computer graphics as it greatly enhances the visual quality of a rendered image. Texture mapping is traditionally used to modulate color values for three-dimensional models but can also be used to introduce detail in a variety of ways. This includes introducing geometric detail through normal mapping as well as modulating material properties other than color.

Throughout this chapter, the two words pixel and texel will be heavily used. Pixel will be used whenever discussing the sample points of the rendered image, whereas texel will refer to the sample points of a texture image. It is important to differentiate between these two, even if they technically represent the exact same thing.

1 Short on Discrete Images

Pixels and texels are indeed sample points, with emphasis on points. Points are zero-dimensional, infinitely small and should not be thought of as higher-dimensional shapes such as circles or squares. As such, they cannot be associated with a size.

The sample points together form a discrete image which has a horizontal and a vertical resolution equal to the amount of sample points in the respective direction. The discrete image is best viewed through a reconstruction filter which extends the image into the continuous domain according to Smith in [8]. The footprint of this filter will be two-dimensional and therefore have a size and a shape.
Texture mapping is a process in which a set of texture coordinates $s$ and $t$, along with the partial derivatives $\partial s$ and $\partial t$ are used to reconstruct a discrete texture image. The texture coordinates $s$ and $t$ are known as the resolution-independent texture coordinates for which the intervals between zero and one cover the entire texture. In order to access individual texels, the coordinates are commonly scaled by the actual resolution and the scaled texture coordinates $x$ and $y$ are obtained. The partial derivatives are scaled in the same fashion, giving a hint on the optimal size for the footprint of the reconstruction filter.

Pixels in the rendered image will rarely match the texels of a texture image, neither in orientation nor in resolution and by that distance between sample points. In three-dimensional computer graphics, the problem is worsened as the footprint of a pixel in image space may become distorted in texture space. This as a result of the perspective projection.

There are two extreme cases which pose great challenges for the texture filtering sub-system. Large regions of the rendered image might produce almost identical small footprints which fall between four texels. Choosing the nearest neighboring texel as a reconstruction filter will select the same texel for many pixels in that region of the rendered image. Clearly, this is not desirable and better reconstruction filters are needed.

When the distance between two texels is larger than the footprint of a pixel, the texture sub-system needs to magnify the texture image. Hence the term texture magnification. Unfortunately, not many techniques suited for real-time applications exist for this case.

Figure 2.1 shows the footprint of a pixel in texture space. In the figure, the footprint is smaller than the distance between two texels and magnification is needed.
The other extreme case presents itself when the footprint of a pixel is mapped to a region covering multiple texels in texture space. Choosing the nearest neighboring texel as a reconstruction filter will select a texel seemingly at random. In addition, slight changes in perspective will cause the filter to select different texels, causing visual artifacts such as Moiré patterns in the rendered image.

With the distance between two texels being smaller than the footprint of a pixel, the texture sub-system needs to minify the texture image. This is known as texture minification and multiple techniques suitable for real-time applications exist to combat the problem.

Figure 2.2 shows the footprint of a pixel in texture space. The footprint is many times larger than the distance between two texels and texture minification is needed.

![Figure 2.2: The footprint of a pixel in texture space needing minification.](image)

Most minification filters introduce a pre-processing pass in which the texture is processed and additional data stored. The pre-processing can often be done efficiently in a parallel computational context and only needs to be done once per texture.

### 3 Wrapping Modes

When a pair of texture coordinates exceed the domain of a texture, a consistent way of remapping these coordinates is needed. Generally, there are two different wrapping modes which can be used for the two texture dimensions. Both dimensions need not use the same wrapping mode, even if it often is convenient to treat both dimensions equally. However, it is crucial that a wrapping mode is defined as it is near impossible to prevent domain exceedings.
Wrapping modes can operate on the resolution-independent texture coordinates or on the integer texel coordinates. This section will define wrapping modes for the latter. As such, wrapping is performed whenever the texture filter requests a texel using the integer texel coordinates.

Given a set of texel coordinates \( x_i \) and \( y_i \), the look-up function \( I \) can be defined as shown in equation 3.1. In the equation, \( I \) corresponds to the discrete texture image whereas \( W_x \) and \( W_y \) are the two wrapping functions.

\[
I[x_i, y_i] = I[W_x[x_i], W_y[y_i]] \quad (3.1)
\]

### 3.1 Repeating

If a texture is meant to be used in a tiling fashion where texel coordinates exceeding the domain should remap to the corresponding texels in the tile, the repeat mode is used. Equation 3.2 shows how a remapping of the texel coordinates \( x_i \) and \( y_i \) is done for repeating textures. The method used in the equation correctly remaps \( x_i \) and \( y_i \) into the intervals between \([0, w - 1]\) and \([0, h - 1]\), even for negative coordinates.

\[
W_x[x_i] = ((x_i \mod w) + w) \mod w \\
W_y[y_i] = ((y_i \mod h) + h) \mod h \quad (3.2)
\]

Graphics pipelines such as the Open Graphics Library generally support two different repeating modes. One which mirrors or inverts every other tile and one which repeats every tile in a consistent fashion [9]. The expressions in the above equation are used to repeat every tile consistently.

### 3.2 Clamping

If a pair of texel coordinates are meant to never exceed their domain, the clamping mode is used. Clamping remaps texel coordinates outside of the domain to the edge texels of the texture image. Equation 3.3 shows how a remapping of the texel coordinates \( x_i \) and \( y_i \) is done for clamping textures.

\[
W_x[x_i] = \min(\max(x_i, 0), w - 1) \\
W_y[y_i] = \min(\max(y_i, 0), h - 1) \quad (3.3)
\]

Graphics pipelines generally support a few different clamping modes which allow fine control over the texels that actually get sampled. For instance, the
Open Graphics Library allows a virtual border of constant color or an actual border of varying colors to be specified around the texture image. The different clamping modes specify whether the clamping should be done to the edge of the texture, to the border of the texture or to in between the edge and the border [9].

Borders can be used to show that some pixels in the rendered image will map to texels outside of the domain. Should this not be the intention, the border texels visible in the rendered image will indicate that something is wrong with the texture coordinates specified.

4 Nearest Neighboring Texel

The nearest neighboring texel filter is a texture filter in its most simple form. It selects a single texel from the texture which corresponds to the one in closest proximity to the scaled texture coordinates $x$ and $y$. Equation 4.1 shows the definition of the nearest neighboring texel filter $N$. For efficiency, the rounding can be implemented as a flooring operation shifted by $\frac{1}{2}$.

$$N(x, y) = I_t[\text{round}(x), \text{round}(y)] \quad (4.1)$$

The filter is generally not used since it is neither suitable for magnification nor minification. However, it extends the discrete texture image into the continuous domain which is the minimum requirement for a texture filter.

5 Bilinear Interpolation

Bilinear interpolation is a filter which averages four texels weighted in accordance with distance in an aim to reconstruct the missing data. This has proven to be sufficient for the perceived visual quality of rendered images when the texture is magnified. Hence, bilinear interpolation is one of the most commonly used texture magnification filters.

In bilinear interpolation, the scaled texture coordinates $x$ and $y$ are split into integer parts and fractional parts using the flooring and fractional functions. The integer parts are used for texel indices and the fractional parts are used to weigh the texels accordingly.

Equation 5.1 shows how bilinear interpolation extends an image into the continuous domain. In the equation, $x_i$ and $y_i$ correspond to the integer parts
while \( x_f \) and \( y_f \) correspond to the fractional parts of the scaled texture coordinates. The equation clearly shows that the filter uses four texel look-ups.

\[
B(x, y) = I_l[x_i, y_i] \cdot (1 - x_f) \cdot (1 - y_f) \\
+ I_l[x_i + 1, y_i] \cdot (x_f) \cdot (1 - y_f) \\
+ I_l[x_i, y_i + 1] \cdot (1 - x_f) \cdot (y_f) \\
+ I_l[x_i + 1, y_i + 1] \cdot (x_f) \cdot (y_f)
\]

(5.1)

Bilinear interpolation does not work well as a minification filter. When the footprint of a pixel covers multiple texels, interpolating four of them provides little to no gain in the visual quality. Better reconstruction filters are needed for minification.

6 Image Pyramids

Image pyramids were introduced in 1983 by Williams in [10]. Williams used the term mip-maps where mip is an abbreviation of the latin phrase *multum in parvo*, meaning many things in a small place. The reconstruction filter utilizes a pre-processing pass in which the original image is down-sampled recursively by a factor of four (one half in each dimension) into an image with multiple resolutions, or levels. This forms a pyramidal structure, hence the term image pyramids.

As images with multiple resolutions are created and stored, the memory requirement for image pyramids is higher in comparison to that of ordinary images. Equation 6.1 shows that the upper bound of the memory requirement is \( \frac{4}{3} \) when compared to ordinary images. This justifies the mip prefix as only \( \frac{1}{3} \) more data is required for the storage.

\[
\lim_{N \to \infty} \sum_{i=0}^{N} \left( \frac{1}{4} \right)^i = \frac{4}{3}
\]

(6.1)

Figure 6.1 illustrates how the image pyramid reconstruction filter creates a series of recursively down-sampled images for three-channel images. It also illustrates how the memory requirement is bounded by \( \frac{4}{3} \) as all down-sampled copies fit inside an additional channel.
Image pyramids are in no way limited to three-channel images, they work just as well with an arbitrary number of channels. Additionally, they can be used for both magnification and minification if the sampling is implemented in a robust way.

### 6.1 Pre-processing

The pre-processing and sampling of image pyramids can differ slightly between different implementations and between different hardware vendors. Yet, its fundamental algorithm is to pre-process the image into multiple resolutions and sample the appropriate level or levels as determined by the size of the footprint.

As every level of the image pyramid is isotropically down-sampled, the total number of levels can be determined using equation 6.2. The formula used in the equation selects the axis with minimum extent in the image and down-samples until only a single row or column remains. Should the image be perfectly square, the final level becomes a single pixel which approximates the entire image.

\[
\text{levels} = \lceil \log_2(\min(w,h)) \rceil + 1
\]  

(6.2)

Re-sampling an image into half its resolution along both axes can be done in two passes provided that the filter used is separable. First along the x-axis and then along the y-axis or the other way around. This two-pass method uses fewer memory accesses than a one-pass method would but is reliant on an intermediate buffer storing the results from the first pass. It might also be subject to additional quantization noise if the intermediate results are stored in integer data types.
The recursive down-sampling process is initialized with the original image. Each recursion level uses the image at the previous level as its input and initializes the next recursion provided that there are more levels still to be processed.

### 6.2 Sampling Method

The set of resolution-independent texture coordinates $s$ and $t$ as well as their partial derivatives $\partial s$ and $\partial t$ together provide enough information to sample the image pyramid. The texture coordinates of a pixel are obtained through (perspectively-correct) interpolation across the triangle while the partial derivatives can be approximated using finite differences. This differentiation is done between adjacent pixels and can be expressed in a number of different ways.

In hardware, pixels are commonly textured in groups of four. Such a group is known as a quad pixel and its structure makes differentiation possible. Usually, the four partial derivatives $\frac{\partial s}{\partial x}$, $\frac{\partial s}{\partial y}$, $\frac{\partial t}{\partial x}$ and $\frac{\partial t}{\partial y}$ are approximated through differentiation and used for the entire quad pixel. For tile-based renderers, other differentiation schemes become possible as entire tiles are textured simultaneously.

The approximate size of an optimal footprint is related to the magnitude of the partial derivatives. A metric commonly used for image pyramids is shown in equation 6.3 [10] in which the footprint size $f_s$ is approximated from the maximum length of two gradients. The length is scaled by the number of samples along the minor axis to produce a footprint size which is expressed in distances between texels. Choosing the maximum gradient length is a trade-off which favors blur over aliasing. Hence, image pyramids often produce an over-blurred sample.

$$f_s = \max \left( \sqrt{\left( \frac{\partial s}{\partial x} \right)^2 + \left( \frac{\partial t}{\partial x} \right)^2}, \sqrt{\left( \frac{\partial s}{\partial y} \right)^2 + \left( \frac{\partial t}{\partial y} \right)^2} \right) \cdot \min(w, h) \quad (6.3)$$

The common metric comes with an assumption about the resolution of a texture. Namely, it assumes that every texture is perfectly square and that the partial derivatives contribute equally to the approximation of the optimal footprint. Williams stated that image pyramids were intended for square, power-of-two resolutions. Yet, it is useful to extend the definition to properly include rectangular textures and preferably arbitrary resolutions as well.
To properly include rectangular textures, the partial derivatives can be weighted in accordance with resolution when computing the length of the two gradients. This modification is shown in equation 6.4.

\[
f_s = \max \left( \sqrt{\left( \frac{\partial s}{\partial x} \cdot w \right)^2 + \left( \frac{\partial t}{\partial x} \cdot h \right)^2}, \sqrt{\left( \frac{\partial s}{\partial y} \cdot w \right)^2 + \left( \frac{\partial t}{\partial y} \cdot h \right)^2} \right) \tag{6.4}
\]

Approximating the footprint size from the lengths of the two gradients is computationally costly as it involves two square root computations at each pixel. The square root can be extracted from the maximum function without altering the expression and the footprint size computed using a single square root. Yet, further optimizations may be made as the size of the footprint is approximated. Equation 6.5 shows a method for which the size of the footprint is approximated independently in width and height.

\[
f_w = \max \left( \left| \frac{\partial s}{\partial x} \right|, \left| \frac{\partial s}{\partial y} \right| \right) \cdot w
\]

\[
f_h = \max \left( \left| \frac{\partial t}{\partial x} \right|, \left| \frac{\partial t}{\partial y} \right| \right) \cdot h \tag{6.5}
\]

The independent approach also aids in the actual sampling process for rectangular images. The appropriate level of the pyramid is determined as shown in equation 6.6. By scaling the resolution-independent texture coordinates \(s\) and \(t\) by the resolution of the image at level \(l_i\), the texture at resolution \(l_i\) can be sampled. The sampling may be performed using a simple texture filter such as nearest neighboring texel or bilinear interpolation with good results due to the pre-processing pass.

\[
l_i = \begin{cases} 
[\log_2(f_w)] & \text{if } f_w > f_h \\
[\log_2(f_h)] & \text{otherwise} 
\end{cases} \tag{6.6}
\]

When sampling a texture using the image pyramid, transitions between the different levels of the pyramid will become visible in the rendered image. To remedy this, trilinear interpolation was introduced. Trilinear interpolation is essentially two bilinear interpolations applied independently to the closest two levels in the pyramid followed by a linear interpolation between the two samples produced. The linear level fractional \(l_f\) is computed as shown in equation 6.7.

\[
l_f = \begin{cases} 
\frac{f_w}{2^r} - 1 & \text{if } f_w > f_h \\
\frac{f_h}{2^r} - 1 & \text{otherwise} 
\end{cases} \tag{6.7}
\]
The trilinear interpolation can be defined as shown in equation 6.8 where 
\((x_0, y_0)\) are the scaled texture coordinates of level \(l_i\) and \((x_1, y_1)\) of level \(l_i + 1\). The two bilinear interpolations are denoted by \(B\) and are weighted in accordance with \(l_f\). Together they form the image pyramid filter \(P_i\). The two bilinear interpolations use four texel look-ups each, amounting to eight look-ups in total.

\[
P_i(x, y) = B(x_0, y_0) \cdot (1 - l_f) + B(x_1, y_1) \cdot (l_f)
\] (6.8)

### 7 Anisotropic Image Pyramids

Anisotropic image pyramids is an extension to image pyramids. Instead of down-sampling the image isotropically, the image is down-sampled independently in both directions by a factor of two. Clearly, this requires more memory when compared to image pyramids and ordinary images. Equation 7.1 shows that this requirement is bounded by four times the original memory requirement.

\[
\left( \lim_{N \to \infty} 2 \sum_{i=0}^{N} \left( \frac{1}{2} \right)^i - 1 \right) \cdot \left( \lim_{M \to \infty} \sum_{j=0}^{M} \left( \frac{1}{4} \right)^j \right) = 3 \cdot \lim_{M \to \infty} \sum_{j=0}^{M} \left( \frac{1}{4} \right)^j = 4
\] (7.1)

The process is illustrated in figure 7.1 which forms a two-dimensional structure of levels with different resolutions. As with image pyramids, there is no limitation in the number of channels that can be used. A single channel was used here only to illustrate the memory requirement. This structure is often referred to as a rip-map, where rip corresponds to rectangular mip.

---

Figure 7.1: An illustration of rip-mapping for one-channel images.
It should be emphasized that the anisotropic extension includes the resolution levels of regular image pyramids. They are present across the diagonal in the figure.

### 7.1 Pre-processing

Similar to regular image pyramids, anisotropic image pyramids are recursively down-sampled in a pre-processing pass. As a two-dimensional structure of resolutions is to be generated, multiple recursive processes need to be performed where each process down-samples the image along either dimension until only a single row or column remains. Between every recursive process, the image is down-sampled once along the other dimension until only a single row or column remains. Equation 7.2 shows the number of levels an image will have in each direction. The total number of levels is the product of these two numbers.

\[
\begin{align*}
\text{levels}_x &= \lfloor \log_2(w) \rfloor + 1 \\
\text{levels}_y &= \lfloor \log_2(h) \rfloor + 1
\end{align*}
\] (7.2)

### 7.2 Sampling Method

The fundamental idea behind anisotropic image pyramids is that the optimal resolution of the texture can be computed using the two partial derivatives \( \partial s \) and \( \partial t \) and selected from the structure. Using the partial derivatives, the width and height of the footprint is computed as shown in equation 7.3.

\[
\begin{align*}
\text{f}_w &= \max \left( \left| \frac{\partial s}{\partial x} \right|, \left| \frac{\partial s}{\partial y} \right| \right) \cdot w \\
\text{f}_h &= \max \left( \left| \frac{\partial t}{\partial x} \right|, \left| \frac{\partial t}{\partial y} \right| \right) \cdot h
\end{align*}
\] (7.3)

Using the size of the footprint, the x-level \( l_{i,x} \) and the y-level \( l_{i,y} \) is computed as shown in equation 7.4.

\[
\begin{align*}
l_{i,x} &= \lfloor \log_2(f_w) \rfloor \\
l_{i,y} &= \lfloor \log_2(f_h) \rfloor
\end{align*}
\] (7.4)

In contrast to regular image pyramids, there are two linear level fractionals for anisotropic image pyramids. These are defined as shown in equation 7.5.

67
\begin{align*}
  l_{fx} &= \frac{f_w}{2^l_{i,x}} - 1 \\
  l_{fy} &= \frac{f_h}{2^l_{i,y}} - 1
\end{align*}

(7.5)

Sampling the anisotropic image pyramid is done in the same fashion as for regular image pyramids, only with a slight alteration. Four bilinear interpolations are performed for the four different level combinations. As the four samples need to be weighted, an additional bilinear interpolation weights these four samples using the two linear levels fractionals. The final filter $P_a$ is shown in equation 7.6 for which the number of texel look-ups amount to sixteen.

\begin{align*}
P_a(x, y) &= B(x_0, y_0) \cdot (1 - l_{fx}) \cdot (1 - l_{fy}) \\
&\quad + B(x_1, y_1) \cdot (l_{fx}) \cdot (1 - l_{fy}) \\
&\quad + B(x_2, y_2) \cdot (1 - l_{fx}) \cdot (l_{fy}) \\
&\quad + B(x_3, y_3) \cdot (l_{fx}) \cdot (l_{fy})
\end{align*}

(7.6)

In the equation $(x_0, y_0)$ correspond to the scaled texture coordinates in level $(l_{i,x}, l_{i,y})$, $(x_1, y_1)$ in level $(l_{i,x} + 1, l_{i,y})$, $(x_2, y_2)$ in level $(l_{i,x}, l_{i,y} + 1)$ and $(x_3, y_3)$ in level $(l_{i,x} + 1, l_{i,y} + 1)$.

8 Summed-Area Tables

Summed-area tables were conceived in 1984 by Crow and presented in [11]. Crow observed that the sum of all pixels in an image enclosed within an axis-aligned rectangle could be computed using a simple formula. By altering the representation of the image into the summed-area table, the sum could be computed using only four table accesses for rectangles of arbitrary sizes. In other words, an arbitrary box-filter could be computed in constant time.

Crow stated that an element of the summed-area table $S$ should be computed as the sum of all pixels in the image with lower or equal indices to that of the element. This is shown in equation 8.1 where $I$ is the discrete texture image.

\begin{align*}
  S[x, y] &= \sum_{j=0}^{y} \sum_{i=0}^{x} I[i, j]
\end{align*}

(8.1)

A box-filter of a rectangular region is the defined as the total sum within the region divided by the total area of the rectangle. Equation 8.2 shows a box-filter $T$ for the region enclosed within $(x_0, y_0)$ and $(x_1, y_1)$. This is equivalent to the mean of the image within the region.
\[ I[x_0, y_0, x_1, y_1] = \sum_{j=y_0}^{y_1} \sum_{i=x_0}^{x_1} I[i, j] \]
\[
\frac{1}{(x_1 - x_0) \cdot (y_1 - y_0)} \] (8.2)

Figure 8.1 shows a rectangular region defined by the coordinates \((x_0, y_0)\) and \((x_1, y_1)\) for which the box-filter is to be applied.

The method that Crow presented involves replacing the double sum of the image with four table look-ups in the summed-area table. This is shown in equation 8.3 [11].

\[ I[x_0, y_0, x_1, y_1] = \frac{S[x_1, y_1] - S[x_0, y_1] - S[x_1, y_0] + S[x_0, y_0]}{(x_1 - x_0) \cdot (y_1 - y_0)} \] (8.3)

The four table look-ups in the summed-area table correspond to sums over different regions in the original image. Figure 8.2 shows the region between \((0, 0)\) and \((x_1, y_1)\), corresponding to the table look-up \(S[x_1, y_1]\).

Figure 8.2: An element in the table corresponds to the sum over a region.
The table look-up $S[x_0, y_1]$ corresponds to the sum over the region between $(0, 0)$ and $(x_0, y_1)$. This is shown in figure 8.3. This sum is subtracted from the sum over the previous region as it is not part of the box-filter.

Figure 8.3: An element in the table corresponds to the sum over a region.

Conversely, the table look-up $S[x_1, y_0]$ corresponds to the sum over the region between $(0, 0)$ and $(x_1, y_0)$. This sum is also subtracted as the region is not part of the box-filter. Figure 8.4 shows this region.

Figure 8.4: An element in the table corresponds to the sum over a region.

The final table look-up $S[x_0, y_0]$ is added back to the sum as it has been removed twice by subtracting the two previous sums. It corresponds to the sum over the region between $(0, 0)$ and $(x_0, y_0)$ as shown in figure 8.5.
Using Crow’s original definition for sampling the summed-area table, the box-filter is actually computed over the region enclosed within \((x_0 + 1, y_0 + 1)\) and \((x_1, y_1)\). This is shown in figure 8.6.

This can be corrected using a slightly different definition of the box-filter. Equation 8.4 shows the new definition for which one has been subtracted from the coordinates \(x_0\) and \(y_0\) and the area adjusted accordingly.

\[
T[x_0, y_0, x_1, y_1] = \frac{S[x_1, y_1] - S[x_0 - 1, y_1] - S[x_1, y_0 - 1] + S[x_0 - 1, y_0 - 1]}{(x_1 - x_0 + 1) \cdot (y_1 - y_0 + 1)}
\]

(8.4)

Accessing elements above and to the left of the \(x_0\) and \(y_0\) coordinates requires that the original image is padded with a row of zeros above and a column of zeros to the left. The summed-area table can then be constructed as described by Crow. However, the introduction of an additional row and an additional
column will alter the memory layout of the image and of the summed-area table. This has to be considered when accessing the elements in a computational context.

### 8.1 Construction

Constructing the summed-area table can be done efficiently in two passes, each handling one of the two sums in the original definition. An intermediate summed-area table $S_x$ is created in the first pass. Its elements store the sum of all pixels in the original image with lower or equal indices in the same row. Equation 8.5 shows this process.

$$S_x[x, y] = \sum_{i=0}^{x} I[i, y] \quad (8.5)$$

The process can be implemented efficiently in parallel as it is independent between rows. Figure 8.7 shows how the cumulative sum of every row is computed through an iterative process.

![Figure 8.7: An intermediate summed-area table is created in the first pass.](image)

Utilizing the intermediate summed-area table, the final summed-area table $S$ can be constructed as shown in equation 8.6.

$$S[x, y] = \sum_{j=0}^{y} S_x[x, j] \quad (8.6)$$

Following the same procedure as the first pass, the final summed-area table can be constructed independently between columns. Figure 8.8 shows this iterative process.
As cumulative sums of the original image need to be stored, the summed-area table requires wider data types for its elements. Assuming integer images with a bit depth per channel of $b_i$, the summed-area table needs a bit depth per channel $b_s$ corresponding to the expression in equation 8.7. This in order to account for the rare worst case scenario in which every pixel of the original image is at maximum intensity. The additional row and column are for the bottom and right borders which are used for repeating textures, explained momentarily.

$$b_s = \lceil \log_2((2^{b_i} - 1) \cdot (w + 1) \cdot (h + 1)) \rceil \approx \lceil b_i + \log_2(w) + \log_2(h) \rceil$$  \hspace{1cm} (8.7)$$

Given an image bit depth and a summed-area table bit depth, the maximum resolution can be computed as shown in equation 8.8.

$$(w + 1) \cdot (h + 1) = \left\lfloor \frac{2^{b_s}}{2^{b_i} - 1} \right\rfloor$$  \hspace{1cm} (8.8)$$

Using a bit depth per channel of 8 for the image and 32 for the summed-area table yields a maximum resolution of roughly 16 million pixels. Textures are rarely this large and a few bits could be saved by limiting texture resolutions to a few megapixels. However, 32 bits will be nicely aligned in memory and the data type is native in most computational contexts, suggesting that 32-bit elements should be used for the summed-area table. This is a four-fold increase in memory requirement in comparison to the original image.

It is worth noting that for integer images, the summed-area table can be constructed without introducing any quantization noise. This in contrast to image pyramids, both regular and anisotropic.
The original summed-area table is monotonically increasing with each element. For real-valued images, this becomes a problem as the floating-point data type was not designed to handle operations between values with large magnitude differences.

An improvement was introduced in 2005 by Hensley et al. in [12]. If the texels of a channel in the original image are biased by subtracting the mean of all texels in that channel, the summed-area table is no longer monotonically increasing. This can be exploited for real-valued images as the box-filter is computed between samples with potentially large magnitude differences.

For biased images, the box-filter is computed as normal and the bias added as a final step. This minimizes the precision issues when computing the summed-area table as well as the issues that may arise when bilinearly interpolating the summed-area table.

8.2 Sampling Method

The previous filters have all been point-sampling filters, approximating the entire footprint by a single sample. This makes any assumptions about the shape of the footprint obsolete. For image pyramids, the size of the footprint was used only to select the appropriate texture levels in which to perform point-sampling.

Sampling in summed-area tables is area-based. This makes both the size and the shape of the footprint of importance. As previously discussed, Crow conceived summed-area tables with axis-aligned box-filters in mind. As such, the original summed-area tables are only able to filter rectangular footprints aligned with the axes of the texture.

With knowledge about the partial derivatives of the texture coordinates $\frac{\partial s}{\partial t}$, the axis-aligned rectangle can be computed in conjunction with the texture coordinates of the sample point $s_c$ and $t_c$. This is shown in equation 8.9.

\[
\begin{align*}
    s_0 &= s_c - \frac{\partial s}{2} \\
    s_1 &= s_c + \frac{\partial s}{2} \\
    t_0 &= t_c - \frac{\partial t}{2} \\
    t_1 &= t_c + \frac{\partial t}{2}
\end{align*}
\] (8.9)
For tile-based rendering pipelines, it is convenient to define the footprint as a square around the sample point in image space. This as a box-filter is to be applied to the underlying texture data. Figure 8.9 shows a tile to be textured where the grid of dots correspond to the sample points of the image.

This requires that the texture coordinates are perspective interpolated to the corners of all footprint squares for each tile. For each tile, this results in \((n + 1)^2\) interpolations instead of \(n^2\) interpolations where \(n\) is the number of sample points in a tile along one dimension.

In image space, the square is bounded by the four vertices \(\vec{K}, \vec{L}, \vec{M}\) and \(\vec{N}\). In texture space, it can be distorted into a quadrilateral. This is shown in figure 8.10 in which each vertex contains the resolution-independent texture coordinates \(s\) and \(t\).

Before the summed-area table can be sampled, the optimal axis-aligned rectangle footprint needs to be computed in texture space. This is done by computing the bounding box of the quadrilateral. Equation 8.10 shows how this is done.
\[ s_0 = \min(K_s, L_s, M_s, N_s) \]
\[ s_1 = \max(K_s, L_s, M_s, N_s) \]
\[ t_0 = \min(K_t, L_t, M_t, N_t) \]
\[ t_1 = \max(K_t, L_t, M_t, N_t) \] (8.10)

The bounding box will in most cases have an area greater than that of the quadrilateral which will result in a blurry sample if it is used as the actual footprint. Blur is often preferred over visual artifacts, suggesting that the bounding box should be used as a footprint.

However, it is possible to adjust the bounding box and by that its area so that it corresponds to the area of the quadrilateral. This can be achieved by first computing the center \((s_c, t_c)\) of the bounding box as shown in equation 8.11.

\[ s_c = \frac{s_0 + s_1}{2} \]
\[ t_c = \frac{t_0 + t_1}{2} \] (8.11)

The bounding box of the quadrilateral and its center are now defined by a three sets of resolution-independent texture coordinates. This is illustrated in figure 8.11.

![Figure 8.11: The bounding box of the quadrilateral and its center.](image)

The area of the quadrilateral \(A_q\) and the area of the rectangle \(A_r\) are computed as shown in equation 8.12. The absolute value is required for the area of the quadrilateral as it is possible for its orientation to be back-facing in texture space.
\[ A_q = \frac{|(M_s - K_s) \cdot (N_t - L_t) - (N_s - L_s) \cdot (M_t - K_t)|}{2} \]
\[ A_r = (s_1 - s_0) \cdot (t_1 - t_0) \]  

(8.12)

A shrinking factor \( f \) is introduced as the square root of the ratio between the two areas as shown in equation 8.13. This factor will be less than or equal to one as the area of the rectangle always is greater or equal to that of the quadrilateral.

\[ f = \sqrt{\frac{A_q}{A_r}} \]  

(8.13)

The bounding box of the quadrilateral can be adjusted in accordance with the shrinking factor. Equation 8.14 shows a process which shrinks the bounding box around its center.

\[ s'_0 = s_c + f \cdot (s_0 - s_c) \]
\[ s'_1 = s_c + f \cdot (s_1 - s_c) \]
\[ t'_0 = t_c + f \cdot (t_0 - t_c) \]
\[ t'_1 = t_c + f \cdot (t_1 - t_c) \]  

(8.14)

An adjusted bounding box, defined by the two sets of texture coordinates \((s'_0, t'_0)\) and \((s'_1, t'_1)\) is obtained from this process. Figure 8.12 illustrates this result.

Figure 8.12: The adjusted bounding box of the quadrilateral and its center.

The area of the adjusted bounding box can be computed as shown in equation 8.15. The equation also shows that this area will be exactly equal to that of the quadrilateral.
\[ A_r' = (s'_1 - s'_0) \cdot (t'_1 - t'_0) \]
\[ = (f \cdot (s_1 - s_c) - f \cdot (s_0 - s_c)) \cdot (f \cdot (t_1 - t_c) - f \cdot (t_0 - t_c)) \]
\[ = f^2 \cdot (s_1 - s_c - s_0 + s_c) \cdot (t_1 - t_c - t_0 + t_c) \]
\[ = f^2 \cdot (s_1 - s_0) \cdot (t_1 - t_0) \]
\[ = \frac{A_q}{A_r} \cdot A_r \]
\[ = A_q \tag{8.15} \]

As previously stated, adjusting the bounding box will cause visual artifacts. As such, the original rectangular footprint should be used for sampling the summed-area table, provided that the bounding box does not exceed the domain of the texture. The next section will provide details on how to bilinearly interpolate the summed-area table.

### 8.3 Bilinear Interpolation

Ordinary bilinear interpolation can be used to sample the summed-area table provided that the image is padded with the corresponding wrapped texels at the bottom and right borders before the summed-area table is constructed. Similar to point-sampling, a table look-up function \( S_l \) is defined as shown in equation 8.16 where \( W_x \) and \( W_y \) are the wrapping functions previously defined and \( S \) the summed-area table.

\[ S_l[x_i, y_i] = S[W_x[x_i], W_y[y_i]] \tag{8.16} \]

The scaled texture coordinate pair \((x, y)\) is obtained through scaling the resolution-independent texture coordinate pair \((s, t)\) by \( w \) and \( h \), respectively. This is shown in equation 8.17.

\[ x = s \cdot w \]
\[ y = t \cdot h \tag{8.17} \]

The integer and fractional parts are computed using the flooring and fractional functions as shown in equation 8.18.

\[ x_i = \text{floor}(x) \]
\[ x_f = \text{frac}(x) \]
\[ y_i = \text{floor}(y) \]
\[ y_f = \text{frac}(y) \tag{8.18} \]
If the image was padded before constructing the summed-area table, bilinear interpolation is performed in the same way as for point-sampling. Equation 8.19 shows how bilinear interpolation is performed using the table look-up function $S_l$ four times.

\[
B(x, y) = S_l[x_i, y_i] \cdot (1 - x_f) \cdot (1 - y_f) \\
+ S_l[x_i + 1, y_i] \cdot (x_f) \cdot (1 - y_f) \\
+ S_l[x_i, y_i + 1] \cdot (1 - x_f) \cdot (y_f) \\
+ S_l[x_i + 1, y_i + 1] \cdot (x_f) \cdot (y_f)
\]  

(8.19)

### 8.4 Clamping Textures

The bounding box of the quadrilateral is described by two resolution-independent texture coordinate pairs $(s_0, t_0)$ and $(s_1, t_1)$, both in texture space. For clamping textures, no sampling should occur past the four edges. Figure 8.12 shows a footprint which exceeds the domain of the texture at the bottom and right edges.

![Figure 8.13: Clamping textures only sample region A.](image)

For clamping textures, the only region of interest is the rectangle marked with $A$ in the figure. This region can be computed by limiting the resolution-independent texture coordinate pairs to the domain of the texture as shown in equation 8.20

\[
s_0' = \min(\max(s_0, 0), 1) \\
s_1' = \min(\max(s_1, 0), 1) \\
t_0' = \min(\max(t_0, 0), 1) \\
t_1' = \min(\max(t_1, 0), 1)
\]  

(8.20)
The sum of all texels in region $A$ is computed using bilinear interpolation as shown in equation 8.21 where the scaled texture coordinates $x_0$, $x_1$, $y_0$ and $y_1$ are obtained as normal. This is the modified definition for sampling the summed-area table as previously discussed.

$$\sum_A = B(x_1, y_1) - B(x_0 - 1, y_0 - 1) - B(x_1, y_0 - 1) + B(x_0 - 1, y_0 - 1)$$

(8.21)

The box-filter of the image enclosed within the region defined by $A$ is computed as shown in equation 8.22.

$$I = \frac{\sum_A}{(x_1 - x_0 + 1) \cdot (y_1 - y_0 + 1)}$$

(8.22)

In total there are 16 table look-ups for clamping textures. A number equal to that of anisotropic image pyramids.

### 8.5 Repeating Textures

Crow mentioned that special considerations had to be made when sampling the summed-area table past the edges. As the summed-area table stores a cumulative sum, elements on the other side of the edges must be increased or decreased by the corresponding values. If the footprint intersects multiple edges, the elements must be increased or decreased with multiples of those values. This section provides a method which properly handles an arbitrary amount of edge intersections in constant time.

For repeating textures, the sampling process becomes a little more complex. Figure 8.14 shows the same footprint exceeding the texture domain at the bottom and right edges. For repeating textures, all four regions $A$, $B$, $C$ and $D$ are of interest.
To compute the sum in each region independently, two additional coordinates are needed. These two coordinates correspond to the width and height of the texture as shown in equation 8.23.

\[
x_e = w \\
y_e = h
\]  

(8.23)

The sum of all texels in region \( A \) is computed as shown in equation 8.24, requiring four bilinear interpolations.

\[
\sum_A = B(x_e, y_e) - B(x_0 - 1, y_e) - B(x_e, y_0 - 1) + B(x_0 - 1, y_0 - 1)
\]  

(8.24)

The sum of all texels in region \( B \) is computed using only two bilinear interpolations. This is shown in equation 8.25.

\[
\sum_B = B(x_1, y_e) - B(x_1, y_0 - 1)
\]  

(8.25)

Conversely, the sum of all texels in region \( C \) is computed using two bilinear interpolations. Equation 8.26 shows this process.

\[
\sum_C = B(x_e, y_1) - B(x_0 - 1, y_1)
\]  

(8.26)

The sum of all texels in the final region \( D \) is computed using a single bilinear interpolation as shown in equation 8.27.
\[
\sum_{D} = B(x_1, y_1) \tag{8.27}
\]

It is possible for the rectangle footprint to span across multiple entire textures, forming additional regions. Though, the sample positions will be exactly the same as in the above example. Computing the correct sum is done by first determining the number of edges that are intersected by the footprint. Equation 8.28 shows how this is done.

\[
N_x = \text{floor}(s_1) - \text{floor}(s_0)
\]
\[
N_y = \text{floor}(t_1) - \text{floor}(t_0) \tag{8.28}
\]

Using these two numbers, the total sum for any rectangular footprint is computed as shown in equation 8.29. It should be emphasized that the expression in the equation will be exactly equal to that of clamping textures if both \(N_x\) and \(N_y\) are equal to zero, meaning that no edges were intersected in either direction.

\[
\sum = B(x_0 - 1, y_0 - 1) - B(x_1, y_0 - 1) - B(x_0 - 1, y_1) + B(x_1, y_1) + B(x_e, y_1) \cdot N_x - B(x_e, y_0 - 1) \cdot N_x + B(x_1, y_e) \cdot N_y - B(x_0 - 1, y_e) \cdot N_y + B(x_e, y_e) \cdot N_x \cdot N_y \tag{8.29}
\]

Identical to clamping textures, the box-filter is the total sum divided by the total area. This is shown in equation 8.30.

\[
I = \frac{\sum}{(x_1 - x_0 + 1) \cdot (y_1 - y_0 + 1)} \tag{8.30}
\]

The number of table look-ups will vary for this filter. In most cases, it will be 16 as for clamping textures. Should the footprint intersect the edges, the number of look-ups can be as large as 36. However, the number of look-ups is
not related to the number of texels in the region of the box-filter. As such, it is performed in constant time with respect to the size of the footprint.

The extra coordinate pair \( x_e \) and \( y_e \) was defined to be exactly equal to the width and height of the texture. This makes it completely unnecessary to bilinearly interpolate samples which contain either \( x_e \) or \( y_e \). Instead, two linear interpolation functions are defined as shown in equation 8.31.

\[
L_x(x, y) = \text{Sl}[x_i, y_i] \cdot (1 - x_f) + \text{Sl}[x_i + 1, y_i] \cdot (x_f)
\]

\[
L_y(x, y) = \text{Sl}[x_i, y_i] \cdot (1 - y_f) + \text{Sl}[x_i, y_i + 1] \cdot (y_f)
\] (8.31)

In addition, samples which contain both \( x_e \) and \( y_e \) only require a single table look-up. The two linear interpolation functions provide a more efficient way of computing the sum of all texels in region of the box-filter. This is shown in equation 8.32 where the sum is computed using at most 25 table look-ups, saving a total of 11 look-ups.

\[
\sum = B(x_0 - 1, y_0 - 1)
\]
\[
- B(x_1, y_0 - 1)
\]
\[
- B(x_0 - 1, y_1)
\]
\[
+ B(x_1, y_1)
\]
\[
+ L_y(x_e, y_1) \cdot N_x
\]
\[
- L_y(x_e, y_0 - 1) \cdot N_x
\]
\[
+ L_x(x_1, y_e) \cdot N_y
\]
\[
- L_x(x_0 - 1, y_e) \cdot N_y
\]
\[
+ \text{Sl}(x_e, y_e) \cdot N_x \cdot N_y
\] (8.32)

8.6 Higher-Order Filters

As stated, the original summed-area tables were designed with axis-aligned box-filters in mind. In many cases, box-filters are not sufficient as they weigh all samples within the support region of the filter equally. The need for higher-order filters is evident.

A generalization of the summed-area table was presented in 1986 by Heckbert in [13]. Heckbert recognized that the summed-area table was in fact an efficient implementation of box-convolution and extended the method to higher-order filters through convolution theory.

Using the new method, a variable width Gaussian-like filter could be implemented in constant time through integrating the original image multiple times.
This followed by applying a slightly different evaluation scheme on the resulting higher-order summed-area table.

However, handling borders still seems to be an unresolved issue, making their use for regular texturing in three-dimensional computer graphics limited. Additionally, the required bit depth is significantly higher and as such, precision becomes an issue for higher-order summed-area tables.

Equation 8.33 [12] shows the required bit depth per channel $b_s$ of a second-order summed-area table (equivalent to a Bartlett filter when sampled) for an integer image with a bit depth per channel of $b_i$.

$$b_s = \left\lceil \log_2 \left( \left( \frac{2^h - 1}{2} \right) \cdot w \cdot \left( \frac{w + 1}{2} \right) \cdot h \cdot \left( \frac{h + 1}{2} \right) \right) \right\rceil$$

\approx \left[ b_i + 2 \cdot \log_2(w) + 2 \cdot \log_2(h) - 2 \right] \tag{8.33}$$

For an image with a resolution of 512 by 512 pixels and with a bit depth per channel of 8, the amount of required memory per channel and pixel is 42 bits [13]. For alignment purposes, 64 bit wide data types should be used for the second-order summed-area table. This is an eight-fold increase in memory in comparison to the original image.
Chapter E

Pipeline Overview

So far, this report has provided a useful tool set which will come to use for a general graphics pipeline. The chapter on texture filtering discussed different problems associated with texture mapping and how to combat them. This chapter will detail the different stages of the actual graphics pipeline implemented.

1 Design Considerations

1.1 Deferred Rendering

The implemented pipeline employs a deferred rendering scheme for a number of different reasons. Mainly because of the common use in modern computer graphics but also due to a number of benefits introduced by the deferred design.

A deferred renderer postpones all lighting computations until the entire scene has been rendered and by that, visibility correctly determined. Lighting computations often require a great amount of processing power and it is beneficial to only light fragments that are guaranteed to be visible for the observer. Deferred rendering makes the complexity of lighting computations independent of overdraw and by that only dependent on the resolution of the target image and on the number of light sources.

The deferred rendering scheme also allows for a variety of post-processing effects such as field of view and screen-space ambient occlusion. However, postponing the lighting computations requires the pipeline to store fragments in a separate buffer. Rasterized fragments generally contain a great number of attributes such as albedo, normal and depth. These attributes must be stored on a per-pixel basis, increasing the memory requirement of the pipeline. If the
fragment record is stored in an uncompressed format, its memory requirement may impose a drawback too large to motivate the deferred design.

The implemented pipeline introduces a compressed record format which stores albedo (24 bits), normal (24 bits), specular power (8 bits), specular intensity (8 bits) and depth (32 bits). The compressed record requires 96 bits of storage in addition to the regular pixel buffer required to accumulate light contributions from the different light sources in the scene.

As opposed to forward rendering, deferred rendering suffers from the inability to properly handle blending. As such, blending is not included in the implemented pipeline and all models are assumed to be manifold. It should be noted that even traditional forward rendering suffers from problems with transparency. This as transparency is dependent on the order in which the fragments are produced, something that cannot be correctly determined for forward rendering nor for deferred rendering.

1.2 Rasterization

It is crucial to carefully evaluate the choice of rasterization method as its compliance with the underlying hardware will have a huge impact on the overall performance of the pipeline.

For serial architectures, a number of different rasterization methods would be suitable. For parallel architectures, data contamination may become a serious problem. If triangles are rasterized on a parallel architecture using a method suitable only for serial architectures, two executed elements may produce fragments at the same location in the target image at roughly the same time. A race condition occurs where a seemingly random process determines which fragment that is stored in the target buffer. This introduces spatial noise in the image and temporal noise between different frames and is an unacceptable flaw.

For serial architectures, scanline rasterization is a common method. For the method, horizontal spans of fragments between the edges of the triangle is generated for each scanline. The scanline method iteratively processes each scanline and each span and stores the resulting fragments or pixels in the target buffer. As the rasterization is performed by a single process, no data contamination can occur, making it suitable for serial architectures only.

Pipelines executing on parallel architectures must take data contamination into consideration. As such, the implemented pipeline employs a hierarchical tile-based rasterization scheme where the screen is subdivided into multiple levels of tiles and no two processes may operate on the same tile at the same time. This completely eliminates the data contamination problem.
1.3 Batch Rendering

The pipeline operates on batches of isolated triangles. Each triangle consists of three vertices and stores additional data required in the different stages of the pipeline. To supply a triangle, the application programmer invokes the corresponding interface method. The method buffers every triangle queued for rendering in host memory and ensures isolation by only accepting a single triangle at once.

Isolating the triangles will enable parallel processing of each individual triangle at the expense of processing a greater number of vertices during the initial stages. This is a trade-off which favors parallelism and data locality and by that helps to utilize the parallel architecture of the graphics processing unit and its cache hierarchy.

When the full capacity of the host buffer is reached or when the pipeline is explicitly instructed through the interface to perform the rendering, the buffer is flushed and the rendering is initialized. This process starts with the triangle buffer being copied to device memory and proceeds through a series of kernels which will be explained in detail throughout the rest of this chapter. The entire process is transparent to the application programmer in a fashion similar to that of the Open Graphics Library.

2 Vertex Transformation

Whenever a triangle is queued for rendering, the active transformation matrix is polled from the transformation sub-system and stored together with the triangle.

The transformation matrix corresponds to the pre-multiplied transformation chain representing the previous transformations as handled by the transformation sub-system. It is used to transform a triangle from object space to camera space. This is equivalent to the model view matrix in the Open Graphics Library with the addition of a camera system.

The vertex transformation stage applies the object space to camera space matrix to the three vertices of every triangle currently in the pipeline. This is
performed through ordinary matrix-vector multiplication. The stage is executed as a one-dimensional kernel with each work item processing a single triangle. Work items are issued in groups of 32.

3 View Frustum Clipping

As the triangles have been transformed to camera space, view frustum clipping can be performed as detailed in the mathematical foundation. The stage is executed as a one-dimensional kernel where each work item processes a single triangle and 32 work items form a work group.

The triangulation associated with view frustum clipping appends every generated triangle to an additional triangle buffer in global memory using atomic operations. In theory, this buffer needs to be five times larger than the batch size to account for the worst case scenario in which every triangle is clipped into a polygon with seven vertices. However, this is considered extremely rare and as such, the two buffers are of equal size.

As the process completes, the number of generated triangles needs to be read from device memory in order to initialize subsequent kernels. Should the number of triangles be equal to zero, no further kernels are launched, the interface is reset and control is returned to the application.

4 Triangle Projection

With triangles clipped against the view frustum, no vertices will be located in front of the near plane. As such, it is safe to project the vertices into the axis-aligned plane of the target image.

The projection stage is executed similarly to the previous stages. A one-dimensional kernel is launched with each work item processing a single triangle and with 32 work items grouped into a work group. Every vertex is projected into an integer raster with sub-pixel precision using the expressions in equation 4.1. In the equation, $p$ corresponds to sub-pixel precision while $(x_r, y_r)$ is the vertex in raster space and $(x_c, y_c, z_c)$ the vertex in camera space. Sub-pixel precision of the geometry is crucial for good results in the subsequent rasterization stage.
The expressions in the equation effectively projects the triangles into the axis-aligned plane of the target image. For each vertex, the inverse depth is also stored for use when interpolating the vertex attributes in a perspective-correct fashion as described in the next section.

The projection kernel also computes the signed double area of every triangle using the integer coordinates in raster space. The double area is stored together with the triangle and will be used heavily in a subsequent kernel. This is one of the major reasons for clipping every triangle against the entire view frustum. With high sub-pixel precision, the raster coordinates will be of large magnitudes and the signed double area of even larger magnitude. Without view frustum clipping, these variables would eventually overflow for triangles extending far beyond the left, right, bottom or top edges of the target image.

The signed double area also provides an efficient method for performing back-face culling. Since only triangles with a positive area will face the camera, all triangles with zero or negative area can be culled from further processing. This as the pipeline is designed for deferred rendering with no transparency or blending effects. Every triangle with an area greater than zero is appended to a final triangle buffer in global memory using atomic operations.

In addition, front-face culling can be achieved by culling all triangles with a non-negative signed double area. However, this is not implemented in the pipeline for the time being.

The number of triangles in the final triangle buffer is read from device memory and used to initialize subsequent kernels. Should this number be zero, no further kernels are launched, the interface is reset and control is returned to the application.

5 Rasterization

Rasterization is essentially the process of breaking down the projected triangles into pixel-sized fragments. This process uses integer arithmetics whenever possible to avoid precision issues associated with floating point arithmetics.
The stage is initialized through a one-dimensional kernel where each work item processes one of the visible triangles in the final triangle buffer. The kernel is executed in work groups of 32 work items and is responsible for the preparation of different attributes required for the rasterization process.

The bounding box of every triangle is computed for three different levels. These three levels are pixel, small tile and large tile. This is computed through a series of minimum and maximum functions on the raster coordinates of the three vertices of every triangle. The three levels form a sort-middle hierarchical rasterization refinement scheme, similar to the scheme described in 2011 by Laine and Karras in [2].

The kernel is also responsible for computing the three edge normals of every triangle as well as projecting the vertex attributes for interpolation across the surface.

### 5.1 Large Tiles

The kernel responsible for large tile rasterization is one-dimensional with one work item for each triangle and is enqueued with a work group size of 32. The group size of 32 is used primarily since it enables efficient use of the local memory as described momentarily. Figure 5.1 shows a triangle projected onto the plane of the target image and how the screen is sub-divided by large tiles.

![Figure 5.1: The target image is sub-divided by large tiles.](image)

Every work item is assigned the task of determining which large tiles that are partially or completely overlapping with the triangle. To achieve this, a work item iterates through the large tiles inside of the bounding box at the large tile level. For every overlapping triangle, a local memory buffer is updated with the corresponding information. Figure 5.2 shows the tiles that are processed.
In a first pass, every large tile corner in the bounding box is projected against
the three edge normals of the triangle and the resulting magnitudes stored in
private memory. For the second pass, the separating axis theorem is applied
and tiles are classified as either outside, inside or overlapping. This information
is stored in a local memory bit matrix using atomic operations on a layer of
32-bit integers which corresponds to the triangles of the current work group.

It should be emphasized that the three edge normals of the triangle are suffi-
cient for the separating axis theorem as only large tiles within the bounding
box are processed. As such, no separation can occur along the edge normals
of the large tiles.

The first work item in a work group clears the bit matrix before any other
work items are allowed to proceed with their tasks. It is also responsible for
writing the resulting bit matrix into global memory when all work items in the
work group have completed. This requires two synchronizations between the
work items of a work group and is achieved through the use of barriers.
No other work group will modify the layer of the bit matrix corresponding to the current work group. As a result, it can be written into global memory without having to rely on atomic operations.

This step of the rasterization process introduces a limitation on the graphics pipeline. As the amount of local memory on a streaming multi-processor is limited and an entire tile layer needs to be resident simultaneously, the resolution will be directly limited by the size of the large tiles.

To compute the maximum possible resolution with a given aspect ratio $\alpha$, a large tile size $s_{\text{large}}$ and a maximum number of large tiles $t_{\text{max}}$, equation 5.1 is used.

$$w_{\text{max}} = \left\lfloor \sqrt{t_{\text{max}} \cdot \alpha} \right\rfloor \cdot s_{\text{large}}$$
$$h_{\text{max}} = \left\lfloor \sqrt{t_{\text{max}} / \alpha} \right\rfloor \cdot s_{\text{large}} \quad (5.1)$$

It is safe to assume that the amount of local memory $m_{\text{local}}$ is at least 16 kilobytes as specified as a requirement in specification of the Open Computing Language. A tile layer is composed of 32 triangles, each requiring two bits of memory. This amounts to 8 bytes per tile and layer. Using this information, the maximum number of horizontal and vertical tiles in a layer can be computed using equation 5.2.

$$t_{\text{max}} = \left\lfloor \frac{m_{\text{local}}}{8} \right\rfloor \quad (5.2)$$

Assuming 16 kilobytes of local memory yields a maximum of 2048 large tiles. If these are arranged according to the common 16:9 (1.78) aspect ratio, the maximum number of large tiles is 60 by 33. This imposes a limit on the resolution which with a large tile size of 64 pixels is 3840 by 2112. As this is well over modern monitor resolutions, the limitation is minimal. It should also be noted that the large tile size can be increased to support higher resolutions.

### 5.2 Small Tiles

The first rasterization refinement step is executed as a three-dimensional kernel. Every work group is responsible for sub-dividing a single large tile of a layer into multiple small tiles. As with the previous step, a work group consists of 32 work items, each corresponding to one of the original triangles. This allows modifications of the bit matrix to be carried out with atomic operations on a local memory buffer. Figure 5.4 shows a large tile to be processed by a work group and the small tiles enclosed within.
The small tile rasterization process shares many fundamental ideas with the previous step and starts by querying the overlap status of the current large tile. Tiles without overlap are not processed further and the work item completes its task. For tiles with complete overlap, every enclosed small tile is set to complete overlap as implied by the large tile. The tiles that were classed as partially overlapping are the ones in need of an actual refinement. This is identical to how Seiler et al. successively refine the rasterization of a triangle.

The large tile in the previous figure is partially overlapping with the triangle. As such, it needs refinement. Figure 5.5 shows the small tiles within the large tile and within the bounding box of the triangle.

An iteration is made through all corners of all small tiles within the large tile and within the bounding box of the triangle at the corresponding level. The corners are projected against the three edge normals and the magnitudes are stored in private memory. A second iteration is made where the separating axis theorem is used to classify the small tiles as either inside, outside or partially overlapping. This is essentially the same process as for large tiles.
Figure 5.6 shows the tile corners that are projected against the edge normals of the triangle.

![Figure 5.6: The corners of the small tile are projected against the edge normals.](image)

The first work item in the work group clears the local memory buffer and is responsible for writing all the small tiles into global memory in the same fashion as earlier. As for large tiles, no atomic operations are needed due to the work group size.

### 5.3 Pixels

The final refinement step is the actual pixel rasterization. This is done differently in comparison to the two earlier rasterization steps. A two-dimensional kernel is launched with each work item operating on a single small tile. The size of a work group is equal to the number of small tiles in a large tile and each work item is responsible for computing the pixel coverage of every triangle located in that small tile. This provides exclusive access to the target image in the region of that tile. As such, every work item is able to read and write the target image without conflicting with other work items. This at the expense of load balancing which will come into the discussion at a later stage in this report.

Figure 5.7 shows a small tile that a single work item is processing. The pixel coverage is to be determined for all triangles intersecting that tile. This is done in sequence for all triangles marked as intersecting the specific tile.
Different to the two other rasterization steps, all computations are performed at the sample points of each pixel as shown in figure 5.8. In a first pass, the double signed areas $2 \cdot A_u$, $2 \cdot A_v$, and $2 \cdot A_w$ are computed. The signed areas are computed using integer coordinates in the sub-pixel raster which completely eliminates problems with floating point precision.

A set of rasterization rules must be defined which together determine if the triangle covers the sample point. From the definition of the barycentric coordinates, the inside of the triangle is defined where all three signed areas are positive as shown in equation 5.3. This is also true for the double signed areas.

$$
A_u > 0 \\
A_v > 0 \\
A_w > 0
$$

(5.3)

If an edge is shared by two triangles, there might be some pixels along the edge where none of the two triangles cover the sample point when using the above
rasterization rules. However, both triangles will tangent the point and the corresponding signed area will be zero for the two triangles. This will generate holes in the rasterization and must be corrected.

The set of rasterization rules must incorporate additional criteria in order to determine which of the two triangles that overlap the unrasterized pixels for the above example. This must be done in a consistent fashion. No pixels should be rasterized twice and no pixels should be left unrasterized.

The edge normals of the two triangles sharing an edge will be oriented in opposite directions if both triangles are front-facing. Since the pipeline only accepts front-facing triangles, this information can be used to determine pixel coverage in a consistent fashion.

Equation 5.4 shows a set of stronger rasterization rules which ensure that a pixel is rasterized exactly once by incorporating the information of the edge normals. In the equation, $\vec{N}_u$, $\vec{N}_v$ and $\vec{N}_w$ are the edge normals of the edges opposite to the triangle vertices $\vec{U}$, $\vec{V}$ and $\vec{W}$, respectively.

\[
\begin{align*}
A_u & > 0 \lor (A_u = 0 \land (N_{u,x} > 0 \lor (N_{u,y} = 0 \land N_{u,y} > 0))) \\
A_v & > 0 \lor (A_v = 0 \land (N_{v,x} > 0 \lor (N_{v,y} = 0 \land N_{v,y} > 0))) \\
A_w & > 0 \lor (A_w = 0 \land (N_{w,x} > 0 \lor (N_{w,y} = 0 \land N_{w,y} > 0)))
\end{align*}
\] (5.4)

The barycentric coordinates of a covered pixel are obtained through dividing the three signed double areas $A_u$, $A_v$ and $A_w$ by the pre-computed signed double area of the entire triangle. This set of coordinates is used to interpolate all vertex attributes.

6 Differentiation of Texture Coordinates

For every covered pixel, the vertex attributes are interpolated in a perspectively-correct fashion as detailed in the mathematical foundation. This includes the texture coordinates, the depth value and the tangent space bases when normal mapping is activated.

The pipeline supports the three advanced texture filters detailed in the chapter on texture filtering. In order to use these filters, the derivatives of the texture coordinates are needed. In a computational context, derivatives are approximated through finite differences. This differentiation can be done in a number of different ways since the pipeline is tile-based.

Figure 6.1 shows a single pixel of a small rasterization tile. One set of texture coordinates is not enough for differentiation. Information from adjacent pixels is required.
In the pipeline, the problem is solved by differentiating the interpolated texture coordinates between groups of four pixels. The differentiation is done along the diagonals as shown in figure 6.2. This is different to how the hardware pipelines handle differentiation of the texture coordinates.

Equation 6.1 shows how the partial derivatives are approximated from the maximum absolute differences between the texture coordinates along the diagonals. Using the maximum absolute differences is a trade-off which favors blur over noise. The division by $\sqrt{2}$ is present due to the distance over which the differentiation is made. The diagonal is $\sqrt{2}$ units long and this needs to be accounted for.

$$\partial s \approx \frac{\max(|C_s - A_s|, |D_s - B_s|)}{\sqrt{2}}$$

$$\partial t \approx \frac{\max(|C_t - A_t|, |D_t - B_t|)}{\sqrt{2}}$$

(6.1)
The resulting partial derivatives of the texture coordinates are used for all four pixels in the group. All active texture maps are sampled using the texture coordinates at pixel \((i, j)\) and the partial derivatives for the group in which the pixel is a member. When normal mapping is activated, the sampled normal is transformed by the interpolated tangent space bases followed by a transformation by the object space to camera space matrix. The results are stored in the fragment buffer and are ready for lighting in a separate pass.

7 Fragment Illumination

Fragment illumination is launched as a two-dimensional kernel with each work item processing a single pixel in the fragment buffer. The kernel operates in work groups of 32 work items and uses the Phong illumination model as introduced in 1975 by Phong in [14]. Phong observed that the spatial frequencies of the specular highlights often were low for photographic images and aimed to improve the standard illumination model used for computer generated images at that time.

For the Phong illumination model, the vertex normals are linearly interpolated across the surface of a triangle. For every fragment to be illuminated, the interpolated normal \(\vec{N}\) is used in conjunction with the light vector \(\vec{L}\) and the reflected observer vector \(\vec{R}\). These vectors are shown in figure 7.1. The observer vector \(\vec{O}\) is not used explicitly in the illumination model, only its reflected counterpart.

![Figure 7.1: The vectors used in the Phong illumination model.](image)

The point \(\vec{X}\) is obtained through reversing the projection of the raster coordinates as shown in equation 7.1.
By using the point in conjunction with the position of the light source $\vec{S}$, the light vector $\vec{L}$ can be computed as shown in equation 7.2. The equation also shows how the point is reflected against the unit normal to obtain $\vec{R}$.

$$\vec{L} = \vec{S} - \vec{X}$$
$$\vec{R} = \vec{X} - 2 \cdot (\vec{X} \cdot \vec{N}) \cdot \vec{N}$$  \hspace{1cm} (7.2)

The square length of the light vector is stored and the vectors $\vec{L}$ and $\vec{R}$ are normalized to unit lengths. The two angles present in the previous figure are computed from these unit vectors as shown in equation 7.3.

$$\alpha = \max(\vec{L} \cdot \vec{N}, 0)$$
$$\beta = \max(\vec{L} \cdot \vec{R}, 0)$$  \hspace{1cm} (7.3)

The ambient, diffuse and specular illuminations are computed from the material properties as shown in equation 7.4 [14]. In the equation, $n$ is the glossiness factor. Materials with high glossiness will produce small specular highlights with high intensities while materials with low glossiness will produce large specular highlights with low intensities.

$$i_a = M_a$$
$$i_d = M_d \cdot \alpha$$
$$i_s = M_s \cdot \beta^n$$  \hspace{1cm} (7.4)

Due to the nature of a deferred rendering pipeline, varying material properties have a great impact on the memory requirement as opposed to traditional forward rendering. Every material property needs to be stored per-pixel. This is not desirable. As such, the ambient factor $M_a$ and the glossiness factor $n$
are constant for all fragments. In addition, the diffuse factor $M_d$ is computed from the specular factor $M_s$ as $1 - M_s$.

The total intensity $i$ is computed as shown in equation 7.5 where $S_s$ is the strength of the light source. The strength is uniform across the spectrum of the light source.

\[
i = i_a + \frac{S_s}{|\vec{L}|^2} \cdot (i_d + i_s)
\]  

(7.5)

Finally, the target image is updated with the light contribution of the current light source. The spectral distribution of the light source is mixed with the spectral distribution of the material and illuminated by the intensity $i$. The light contribution is added to the light contribution from previous fragment illumination passes and stored in the target image. This is shown in equation 7.6.

\[
\begin{align*}
    r' &= r + M_r \cdot S_r \cdot i \\
    g' &= g + M_g \cdot S_g \cdot i \\
    b' &= b + M_b \cdot S_b \cdot i
\end{align*}
\]  

(7.6)

For multiple light sources, the fragment illumination pass is run in sequence. Once for every light source until all light sources have contributed to the final image.
Chapter F

Results

1 Rendered Images

Using the functionality of the implemented three-dimensional graphics pipeline, a set of images were rendered. These images are intended to demonstrate the capabilities of the pipeline and were rendered with a resolution four times greater than the target image. The target image was bilinearly interpolated using the texturing filtering of the Open Graphics Library when displayed on the screen.

The first set demonstrates the deferred rendering of the pipeline. The tank zombie from the game Left for Dead was used as a test model. It contains 2894 vertices and 5784 triangles. Figure 1.1 shows the model illuminated by three white light sources.
The different channels of the fragment buffer can be visualized using the application. Figure 1.2 shows the albedo (color) channels.

Figure 1.3 shows the normal channels. As the coordinate system is left-handed, red intensities indicate surface normals directed to the right. Conversely, green intensities indicate surface normals directed upwards and blue intensities surface normals directed into the image plane. The normals are defined in observer space.
Figure 1.3: The normal channels of the fragment buffer.

Figure 1.4 shows the specular channel of the fragment buffer. It indicates the specular reflectance of the visible fragments. Higher intensities will transfer light specularly while lower intensities will transfer light diffusely.

The final channel is the depth channel. It is shown in figure 1.5. Inverse, non-linear depth is used as it can be interpolated linearly across the surface of every projected triangle. Higher intensities correspond to fragments located closer to the observer.
2 Texture Filtering

The second set of rendered images demonstrate the texture filtering techniques, starting with the filters suitable for both texture magnification and minification. Figure 2.1 shows a surface textured using the image pyramid filter. This is equivalent to how textures are filtered in the Open Graphics Library [9].

Figure 2.1: A surface textured using the image pyramid filter.
Figure 2.2 shows a surface textured using the extended anisotropic image pyramid filter. Please note how the far details become sharper and more distinct.

![Surface textured using the anisotropic image pyramid filter.](image)

Figure 2.2: A surface textured using the anisotropic image pyramid filter.

Figure 2.3 shows the same surface textured using the final of the three texture filters suitable for both texture magnification and minification, the summed-area table. Similar to anisotropic image pyramids, the far details become sharper. Yet, there are slight differences.

The image produced using summed-area tables is slightly more bright at far distances in comparison to the previous two images. This as the image pyramids were generated without gamma-correction. As such, the brightness will drift with every down-sampled texture level. This does not happen when using summed-area tables.
The different texture levels of the two image pyramids can be visualized using the application. Figure 2.4 shows the distinct transitions between the texture levels of the regular image pyramid.

Figure 2.4: The texture levels for the regular image pyramid.

Figure 2.5 shows the corresponding transitions for the anisotropic image pyramid. As the levels are selected from a two-dimensional structure of textures with different resolutions, the visualization uses two colors. Red corresponding to the horizontal level and green corresponding to the vertical level.

Figure 2.5: A surface textured using the summed-area table.
The third and final set of images demonstrate the texture filters suitable for texture magnification. Figure 2.6 shows the nearest neighboring texel filter. At close range, the individual texels become visible.

The bilinear interpolation filter is far better at magnifying textures as shown in figure 2.7. When implemented correctly, the two image pyramids and the summed-area table actually default into bilinear interpolation filters at close range, producing results equal to those shown in the figure.
### 3 Benchmark Data

#### 3.1 Rasterization

The raw rasterization performance is of great importance for the graphics pipeline. As such, the rasterization scheme was evaluated through a series of tests. For the tests, the large tiles were fixed at 64 by 64 pixels and the small tiles at 4 by 4 pixels. A total of 5,000 frames were rendered with no textures bound and no view-frustum clipping. For every frame, a single quadrilateral covering the entire screen was rendered. Table 3.1 shows the resulting data.

<table>
<thead>
<tr>
<th>Resolution (px)</th>
<th>Large (ms)</th>
<th>Small (ms)</th>
<th>Pixel (ms)</th>
<th>Total (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128x128</td>
<td>0.10</td>
<td>0.70</td>
<td>0.39</td>
<td>1.2</td>
</tr>
<tr>
<td>256x256</td>
<td>0.12</td>
<td>0.78</td>
<td>0.76</td>
<td>1.7</td>
</tr>
<tr>
<td>512x512</td>
<td>0.24</td>
<td>0.93</td>
<td>1.86</td>
<td>3.0</td>
</tr>
<tr>
<td>1024x1024</td>
<td>0.65</td>
<td>1.22</td>
<td>6.09</td>
<td>8.0</td>
</tr>
<tr>
<td>2048x2048</td>
<td>1.13</td>
<td>1.63</td>
<td>10.91</td>
<td>13.7</td>
</tr>
</tbody>
</table>

Table 3.1: Benchmark data for different resolutions.

#### 3.2 Load Balancing

In order to test the load balancing of the tile-based rasterization scheme, two separate tests were conducted. The first test was designed to evaluate how the...
pipeline handles a fully covered screen with no overdraw. For this, a single quadrilateral composing of two triangles was rendered. These two triangles covered the entire screen and view frustum clipping was inactivated. A series of 5000 frames with a resolution of 800 by 600 pixels were rendered in total.

Table 3.2 shows the average processing time per frame for the three rasterization steps and for a few different tile sizes. No textures were bound to the pixel rasterization kernel.

<table>
<thead>
<tr>
<th>Large (px)</th>
<th>Small (px)</th>
<th>Large (ms)</th>
<th>Small (ms)</th>
<th>Pixel (ms)</th>
<th>Total (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64x64</td>
<td>4x4</td>
<td>0.38</td>
<td>1.05</td>
<td>3.17</td>
<td>4.6</td>
</tr>
<tr>
<td>64x64</td>
<td>8x8</td>
<td>0.38</td>
<td>0.34</td>
<td>5.27</td>
<td>6.0</td>
</tr>
<tr>
<td>64x64</td>
<td>16x16</td>
<td>0.38</td>
<td>0.15</td>
<td>9.64</td>
<td>10.2</td>
</tr>
<tr>
<td>128x128</td>
<td>8x8</td>
<td>0.17</td>
<td>0.80</td>
<td>5.17</td>
<td>6.1</td>
</tr>
<tr>
<td>128x128</td>
<td>16x16</td>
<td>0.17</td>
<td>0.26</td>
<td>7.65</td>
<td>8.1</td>
</tr>
</tbody>
</table>

Table 3.2: Benchmark data for different tile sizes with a few large triangles.

The second test was designed to evaluate how the pipeline handles the case where only a small region of the screen is covered by a large number of triangles and with overdraw. A series of 500 frames with a resolution of 800 by 600 pixels were rendered. No textures were bound when measuring the average execution time of the rasterization kernels, shown in Table 3.3. The tank zombie model was rendered from a viewpoint where it covered 6.7 percent of the total screen pixels. All triangles were rendered in a single batch.

<table>
<thead>
<tr>
<th>Large (px)</th>
<th>Small (px)</th>
<th>Large (ms)</th>
<th>Small (ms)</th>
<th>Pixel (ms)</th>
<th>Total (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64x64</td>
<td>4x4</td>
<td>1.65</td>
<td>9.55</td>
<td>27.20</td>
<td>38.4</td>
</tr>
<tr>
<td>64x64</td>
<td>8x8</td>
<td>1.65</td>
<td>4.45</td>
<td>171.52</td>
<td>177.6</td>
</tr>
<tr>
<td>64x64</td>
<td>16x16</td>
<td>1.65</td>
<td>3.38</td>
<td>640.06</td>
<td>645.1</td>
</tr>
<tr>
<td>128x128</td>
<td>8x8</td>
<td>1.53</td>
<td>4.16</td>
<td>111.97</td>
<td>117.6</td>
</tr>
<tr>
<td>128x128</td>
<td>16x16</td>
<td>1.53</td>
<td>2.94</td>
<td>658.32</td>
<td>662.8</td>
</tr>
</tbody>
</table>

Table 3.3: Benchmark data for different tile sizes with many small triangles.

### 3.3 Texture Mapping

The texture mapping sub-system is likely to be dependent on the level of cache utilization. The Fermi architecture uses cache lines of 128 bytes, has an L2 cache of 768 kilobytes and a configurable L1 cache of 64 kilobytes [5]. A series of tests were conducted to measure how these caches were utilized for the linear memory layout of the textures.

For the tests, a single quadrilateral covering the entire screen was rendered. The quadrilateral was textured by an albedo texture with a resolution of 512
by 512 pixels. The screen resolution was set to the identical resolution and the image pyramid filter used. The texture was set to the repeat wrapping mode and the number of tiles was altered for each test.

In theory, performance should scale with the number of tiles as the texture resolution will decrease, providing better cache utilization. A total of 5 000 frames were rendered for each test and a separate test with no textures bound conducted. With no textures, the average execution time of the pixel rasterization kernel was 1.88 ms. The execution times for the texturing sub-system were computed as the differences between the total execution times and this duration. This is shown in table 3.4.

<table>
<thead>
<tr>
<th>Level</th>
<th>Pixel (ms)</th>
<th>Texturing (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2.76</td>
<td>0.88</td>
</tr>
<tr>
<td>1</td>
<td>2.74</td>
<td>0.86</td>
</tr>
<tr>
<td>2</td>
<td>2.67</td>
<td>0.79</td>
</tr>
<tr>
<td>3</td>
<td>2.61</td>
<td>0.73</td>
</tr>
<tr>
<td>4</td>
<td>2.56</td>
<td>0.68</td>
</tr>
<tr>
<td>5</td>
<td>2.53</td>
<td>0.65</td>
</tr>
<tr>
<td>6</td>
<td>2.50</td>
<td>0.62</td>
</tr>
<tr>
<td>7</td>
<td>2.50</td>
<td>0.62</td>
</tr>
<tr>
<td>8</td>
<td>2.50</td>
<td>0.62</td>
</tr>
<tr>
<td>9</td>
<td>2.49</td>
<td>0.61</td>
</tr>
</tbody>
</table>

Table 3.4: Benchmark data for different texture levels.
Chapter G

Discussion

1 Memory Requirements

As a result of the tile-based rasterization scheme and the deferred design of the graphics pipeline, memory requirements have to be carefully evaluated. In addition, the triangles in the active rendering batch require a memory buffer in device memory.

1.1 Triangles

The triangle record occupies a total of 428 bytes of device memory. Of this, 64 bytes are used to store the transformation matrix and 108 bytes to store the three per-vertex tangent space bases. Other data stored in the record is the three edge normals, different vertex attributes and bounding boxes for the different levels of the hierarchical rasterization scheme.

The 428 byte memory footprint per triangle record is large but the triangle buffer only needs to be allocated to contain a reasonably small number of triangles. High occupancy levels can be reached using a batch size greater or equal to a small multiple of the number of cores available on the device. This provided that the pipeline is able to distribute the work load evenly. For instance, the Fermi architecture contains at most 480 cores. Rendering all triangles in batches of a few thousand triangles should be sufficient. With this in mind, the pipeline was set to process 8,192 triangles in each batch. This amounts to a total memory footprint of 3,506,176 bytes. Two such buffers are required for the different stages in the pipeline.

The memory requirement for the triangle buffers is not extremely large, yet it can be limited through the introduction of a separate vertex buffer. A large portion of the triangle record stores per-vertex data. Ordinary valence of a
vertex is six, meaning that each vertex is shared by approximately six triangles. The introduction of a vertex buffer could decrease the size of the triangle record significantly as well as increase the overall performance of some stages in the pipeline. In addition, the pipeline could be restricted to only handle triangle batches with equal transformation matrices. This would remove the need of storing the transformation matrix per-triangle and through that decrease the size of the record.

1.2 Tiles

During the rasterization process, triangle and tile overlap is stored using 2 bits per triangle and tile as mentioned earlier. As such, the amount of required data is dependent on the sizes of the two different tiles, on the batch size as well as on the screen resolution. With knowledge of the screen resolution $w$ and $h$, the size of a large tile $s_l$ and the batch size $s_b$, the memory requirement for the large tiles $m_l$ can be computed using equation 1.1.

$$m_l = \left\lceil \frac{w}{s_l} \right\rceil \cdot \left\lceil \frac{h}{s_l} \right\rceil \cdot \left\lceil \frac{s_b}{32} \right\rceil \cdot 8 \quad (1.1)$$

Using this equation with a batch size of 8 192 for a few different resolutions and large tile sizes yields the data in table 1.1. As seen in the table, the memory requirements are fairly low, even for higher resolutions.

<table>
<thead>
<tr>
<th>Resolution (px)</th>
<th>Large (px)</th>
<th>Required Memory (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>800x600</td>
<td>64x64</td>
<td>266 240</td>
</tr>
<tr>
<td>800x600</td>
<td>128x128</td>
<td>71 680</td>
</tr>
<tr>
<td>1024x768</td>
<td>64x64</td>
<td>393 216</td>
</tr>
<tr>
<td>1024x768</td>
<td>128x128</td>
<td>98 304</td>
</tr>
<tr>
<td>1920x1080</td>
<td>64x64</td>
<td>1 044 480</td>
</tr>
<tr>
<td>1920x1080</td>
<td>128x128</td>
<td>276 480</td>
</tr>
</tbody>
</table>

Table 1.1: Memory requirements for the large tiles for a few different resolutions.

The size of a large tile is defined in the pipeline using the size of a small tile and a small to large tile ratio. This to ensure that an even number of small tiles fit inside every large tile during the small tile rasterization pass. As such, the number of small tiles is directly related to the ratio $r$ and by that also the memory requirement. The relationship between the memory required for the large tiles and the memory required for the small tiles is $m_l \cdot r^2$. For the pipeline to fully utilize the cores of the Fermi architecture, the squared tile ratio should be as high as possible and preferably an even multiple of 32. This
is one reason to why the pipeline performs better with 64 small tiles in each large tile. However, this implies a large memory requirement for the small tiles.

As the screen should be divided into as many large tiles as possible and in turn the large tiles into as many small tiles as possible, the memory requirements become a problem. Using equation 1.2, the total memory requirement of the tiles can be computed.

\[ m = m_l \cdot (1 + r^2) \]  

(1.2)

The amount of required memory grows to unacceptable numbers. Table 1.2 shows the total memory required for all tiles for a few different resolutions and large tile sizes. For the computations, the small to large tile ratio \( r \) was set to 8.

<table>
<thead>
<tr>
<th>Resolution (px)</th>
<th>Large (px)</th>
<th>Required Memory (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>800x600</td>
<td>64x64</td>
<td>17 305 600</td>
</tr>
<tr>
<td>800x600</td>
<td>128x128</td>
<td>4 659 200</td>
</tr>
<tr>
<td>1024x768</td>
<td>64x64</td>
<td>25 559 040</td>
</tr>
<tr>
<td>1024x768</td>
<td>128x128</td>
<td>6 389 760</td>
</tr>
<tr>
<td>1920x1080</td>
<td>64x64</td>
<td>67 891 200</td>
</tr>
<tr>
<td>1920x1080</td>
<td>128x128</td>
<td>17 971 200</td>
</tr>
</tbody>
</table>

Table 1.2: Memory requirements for all tiles for a few different resolutions.

Unfortunately, these numbers hint that the rasterization stage should be redesigned as its preferences regarding tile sizes and number of tiles are in direct conflict with the memory requirements. Specifically, the amount of memory required for the small tiles.

1.3 Fragments

The fragment buffer is the common culprit of deferred rendering pipelines. It requires large amounts of allocated memory to store the per-pixel attributes of every visible fragment. As such, its memory requirement is directly related to the resolution of the target image. The only way of limiting the memory footprint is to limit the number of components in each individual fragment record.

Fragment buffers often store albedo (color), surface normal, depth, specular intensity and specular power. Commonly, most of these attributes are compressed into integer data types as the intervals of most attributes are known
and limited. This allows compression with minimal quantization noise. During the lighting passes, the fragment record is decompressed and its attributes used in the illumination model.

A compressed fragment may be stored in a 96-bit record. For a resolution of 1920 by 1080 pixels, this amounts to 24 883 200 bytes. This is a large memory requirement. Unfortunately, any further improvements other than removing individual attributes from the fragment records are prevented by the deferred design of the pipeline. In addition, memory alignment must be considered. Removing a single attribute compressed into 8 bits will require an additional three attributes to be removed. Adhering to this discussion, the use of the 96-bit fragment record in the pipeline is well-motivated.

2 Performance

When it comes to rasterization, the tile-based approach provides an efficient way of partitioning the task. As seen in table 3.1 of the results chapter, the pipeline performs well for high resolutions with few triangles and no overdraw. This as the amount of large tiles and by that threads increase with resolution when the large tile size is fixed, allowing high occupancy levels of the hardware. If the total execution times from the table are divided by the number of large tiles at each resolution, an interesting pattern emerges. This is shown in figure 2.1.

![Figure 2.1: Linear plot of average rasterization time per large tile.](image)

Only a small number of large tiles fit within the smaller resolutions. As the resolution is increased, so is the number of total large tiles. At sufficient resolutions, the performance gain evens out as expected. This as high occupancy levels have been reached for resolutions above 1024 by 1024 pixels. This hints that tile-based rendering has great potential, provided that the resolution is high enough to be partitioned into a high number of large tiles.
From the results in the previous chapter, uneven load balancing seems to be a serious issue for the implemented pipeline. When the pipeline is fed with the reasonably small test model consisting of 5,784 triangles, the only rasterization kernel with acceptable execution times is the one used for rasterizing the large tiles. The other two kernels do not handle the uneven work load well.

The behavior of the large tile rasterization kernel was expected as it operates per-triangle and as such is not subject to an uneven work load. The other two kernels operate per-tile, with possible uneven work loads. As no specific load balancing measures were implemented for the two kernels, some problematic behavior was to be expected. Yet, the data presented in the results chapter proved the bottle-neck to be narrower than expected. Clearly, a redesign of that specific stage is needed in order to achieve good performance.

The texturing sub-system yielded results far better than expected. As the memory layout of the textures are linear per row and planar per channel, texturing was expected to have a great impact on performance due to poor cache coherency. This was not the case as demonstrated in the results chapter. Texturing 262,144 pixels could be performed in 610 to 880 microseconds using the regular image pyramid filter. As shown in table 3.4 of the results chapter, the performance increases with each texture level. However, overdraw was not considered in order to isolate the cache performance. Typical scenes have overdraw and are usually rendered at much higher resolutions. In addition, each fragment is often generated using multiple textures.

The measured fill rate of 297,890,909 to 429,744,262 textured pixels per second does not compare well to the fill rate specified at 42,000,000,000 for the Fermi architecture [5]. However, the number stated in the specification is the theoretical texture filtering performance computed as one filtered look-up per clock cycle.

The Open Computing Language provides functionality to store images in a non-linear layout and to use the hardware texturing sub-system to filter the images. This would most certainly increase the performance of the texturing sub-system. However, the intention of this master’s thesis was to explore if the entire graphics pipeline could be implemented using solely the programmable units. As such, the image objects of the Open Computing Language were not used apart for inter-operability with the window interface.
Chapter H

Conclusion

This experiences with the graphics pipeline and with the graphics processing unit have been incredibly interesting. Much was learned through the continuous literature study and even more through implementing the different graphics algorithms. Through doing this, I have observed a few important aspects of tile-based rendering and of texture filtering. Notably, the increase in visual quality when using more sophisticated texture filters such as the anisotropic image pyramid or the summed-area table. These two come with a four-fold increase in memory requirement, something which is considered unacceptable within the industry. This as a multitude of textures often need to be simultaneously resident in device memory. A single texture should not be allowed to occupy four times the memory in comparison to the original texture.

Device memory is still a huge limitation for graphics processing units. The amount of memory on a typical consumer-level device is often far less than the main memory of the system. In addition, render data needs to be transferred to the device and techniques requiring a dynamic amount of memory become troublesome. This is an issue which as of today remains unsolved for the Open Computing Language. In many cases, device memory needs to be allocated for the worst case scenario. This is not a good solution as much of the memory remains unused while preventing other data from residing in that region of the device memory.

The tile-based rendering scheme proved to handle high resolutions elegantly but the non-existent work load balancing proved to be a bottle-neck narrower than anticipated. As such, a redesign of the rasterization stage should be considered. This redesign must account for uneven work loads, possibly handling this in a way similar to that of the pipeline presented by Laine and Karras.

Many of the conducted experiments indicated that device occupancy is an important aspect. A great increase in performance was observed until the number of sub-tasks were a few multiples of the total number of device cores.
This in accordance with the technical documents presented by Nvidia and the Khronos Group as well as stated in the paper by Laine and Karras [2].

It saddens me to conclude the work carried out within this master’s thesis. From the conducted experiments and the acquired knowledge about the hardware, many design choices would have been made differently, should the pipeline had been designed today. Unfortunately, the development of this graphics pipeline stops here and now.

In conclusion, I wish to thank the authors of the different research papers quoted throughout this report. They have inspired me and given me ideas of my own to explore.
Bibliography


[8] Alvy Ray Smith. A pixel is not a little square, a pixel is not a little square, a pixel is not a little square! (and a voxel is not a little cube). Technical report, Microsoft Research, 1995.


