Design of 3D Accelerator for Mobile Platform

Master’s Thesis performed at
Department of Electrical Engineering

Linköping University Sweden

by

Radha Krishna .R

Reg No: LiTH-ISY-EX--06/3811--SE

Linköping Date May 22 2006
Design of 3D Accelerator for Mobile Platform

Master’s Thesis performed at
Department of Electrical Engineering
Linköping University Sweden

By
Radha Krishna. Ramachandruni

For
Degree in Masters in Electrical Engineering

Reg No: LiTH-ISY-EX--06/3811--SE

Examiner
Dr. Dake Liu

Supervisors
Johan Eilert
Dr. Magnus Karlsson

Linköping May 22 2006
Abstract

Implement a high-level model of computationally intensive part of 3D graphics pipe-line. Increasing popularity of handheld devices along with developments in hardware technology, 3D graphics on mobile devices is fast becoming a reality. Graphics processing is essentially complex and computationally demanding. In order to achieve scene realism and perception of motion, identifying and accelerating bottle necks is crucial. This thesis is about Open-GL graphics pipe-line in general. Software which implements computationally intensive part of graphics pipe-line is built. In essence a rasterization unit that gets triangles with 2D screen, texture co-ordinates and color. Triangles go through scan conversion, texturing and a set of other per-fragment operations before getting displayed on screen.
With Love to:

Mom: Late R. Prema Vasantha Kumari
Dad: R.S.N. Murthy
Brother: R. Ravi Krishna
Abstract

Implement a high-level model of computationally intensive part of 3D graphics pipe-line. Increasing popularity of handheld devices along with developments in hardware technology, 3D graphics on mobile devices is fast becoming a reality. Graphics processing is essentially complex and computationally demanding. In order to achieve scene realism and perception of motion, identifying and accelerating bottle necks is crucial. This thesis is about Open-GL graphics pipe-line in general. Software which implements computationally intensive part of graphics pipe-line is built. In essence a rasterization unit that gets triangles with 2D screen, texture co-ordinates and color. Triangles go through scan conversion, texturing and a set of other per-fragment operations before getting displayed on screen.

Keywords: Open GL, Graphics Pipe-line, Rendering, Rasterization, Texturing and Anti-aliasing.
Acknowledgements

I would first like to thank Dr. Dake Liu and Di Wu for giving me an opportunity to work on 3D graphics, and my supervisor Johan Eilert for his support. And thanks to Ylva for helping me with a lot of administrative things. I would also like to thank my supervisor at University of Kalmar Dr. Magnus Karlsson. In addition to the above, thanks to my friends Bharat Gupta and Suri who all have supported me in all respects and my opponent Imtiyaz Pasha.
Table of Contents

Chapter 1 Introduction .................................................................................................... 16
1.1 Motivation and Purpose ....................................................................................... 16
1.2 Reading Instructions ......................................................................................... 17
1.3 Intended Audience ......................................................................................... 18
1.4 Result ............................................................................................................... 18

Chapter 2 The Graphics Pipe-Line ........................................................................... 19
2.1 Introduction ....................................................................................................... 19
  2.1.1 Geometry Engine ....................................................................................... 20
  2.1.2 Geometry Transformation ......................................................................... 20
  2.1.3 Culling and Clipping ................................................................................. 21
  2.1.4 Lightening ................................................................................................. 22
2.2 Rasterization Engine ....................................................................................... 23
  2.2.1 Scan Conversion ....................................................................................... 24
  2.2.2 Texturing ................................................................................................... 24
  2.2.3 Per-fragment Operations ......................................................................... 25
  2.2.4 Blending ................................................................................................... 26

Chapter 3 Implementing Rasterization Engine ....................................................... 28
3.1 Scan Conversion ............................................................................................... 28
  3.1.1 Triangle scan conversion ........................................................................... 30
  3.1.2 Shading - Interpolating Color ..................................................................... 33
  3.1.3 Interpolating Texture Co-ordinates ............................................................ 35
  3.1.4 Interpolating Fog ...................................................................................... 41
3.2 Texturing .......................................................................................................... 42
  3.2.1 Texture filtering ......................................................................................... 42
  3.2.2 Texture Blending ....................................................................................... 48
  3.2.3 Fog ............................................................................................................ 48
3.3 Per-fragment Operations .................................................................................. 49
  3.3.1 Scissor Test ............................................................................................... 49
  3.3.2 Alpha Test ................................................................................................ 49
  3.3.3 Depth Test ............................................................................................... 50
List of Figures

Figure 2:1  Wire frame model ................................................................. 19
Figure 2:2  A generic 3D graphics Pipe-line .............................................. 20
Figure 2:3  Geometry engine ................................................................. 20
Figure 2:4  flat shaded triangle .............................................................. 22
Figure 2:5  gouraud shaded triangle ....................................................... 22
Figure 2:6  Rasterization Pipe-line .......................................................... 23
Figure 2:7  Triangle covering pixel centers ................................................ 24
Figure 2:8  Other Per-fragment Operations ............................................... 25
Figure 2:9  Overlapping Triangles ............................................................ 26
Figure 2:10  Effect of Alpha blending ...................................................... 27
Figure 3:1  Triangle Edges and scan lines .................................................. 29
Figure 3:2  Types of triangles in rasterization context .................................. 31
Figure 3:3  Triangle with right middle ...................................................... 31
Figure 3:4  Scan converting Triangle ....................................................... 32
Figure 3:5  Triangle with left middle ........................................................ 33
Figure 3:6  Scan converting – Interpolating Color ....................................... 35
Figure 3:7  Texture mapping ................................................................. 36
Figure 3:8  Scan conversion – Interpolating Texture co-ordinates .................. 37
Figure 3:9  Demonstrating no perspective distortion in edges ...................... 38
Figure 3:10  demonstrating perspective distortion in edge ......................... 38
Figure 3:11  affine texture interpolation .................................................. 39
Figure 3:12  Perspective texture interpolation ........................................... 39
Figure 3:13  Affine texture mapping – checker board ................................. 40
Figure 3:14  perspective texture mapping – checker board ......................... 40
Figure 3:15  Scan conversion – Perspective texture interpolation ................. 41
Figure 3:16  Texture clamp modes Clamp and repeat ................................. 42
Figure 3:17  Texture Look-up ............................................................... 43
Figure 3:18  Point sampling Vs Bi-linear filtering ....................................... 44
Figure 3:19  Bi-linear filtering ............................................................... 45
Figure 3:20 Mip-map generation by averaging texels .......................................................... 46
Figure 3:21 Mip-maps chain ................................................................................................ 47
Figure 3:22 Tri-linear filtering between mipmaps .............................................................. 47
Figure 3:23 Two overlapping triangles ............................................................................... 52
Figure 4:1 Top blocky edges; Bottom edge pixels with light shade .................................... 53
Figure 4:2 Sub pixels – Ordered grid multi sampling ....................................................... 54
Figure 4:3 Sub pixels – Partially and fully covered a by triangle edge ............................... 55
Figure 4:4 Sub pixels – Edge through a single pixel .......................................................... 56
Figure 4:5 Left figure No edge anti-aliasing; Right figure ordered grid multi sample anti-aliasing ............................................................................................................. 56

List of Tables in Appendix A

Table A - 1 Computations per-triangle ............................................................................... 62
Table A - 2 Computations per-scan line ............................................................................. 63
Table A - 3 Computations per-fragment ............................................................................ 63
Table A - 4 Texturing computations per-fragment ............................................................. 63
Table A - 5 Fog – per-fragment .......................................................................................... 64
Table A - 6 Computations - Z - test .................................................................................. 64
Table A - 7 Computations - Alpha Blending ...................................................................... 64
Table A - 8 Results for 10K triangles ............................................................................... 65
Table A - 9 Texturing Results for 250K fragments ............................................................ 66
Table A - 10 Computational Cost for 25 frames per-second ............................................. 66
3D graphics and imaging is going through rapid advancement with developments in hardware technology. It has applications ranging from medical imaging, scientific visualization, and virtual reality simulations to high-end gaming and many more. 3D graphics has been limited to high-end graphics machines and desktop computers; however, with significant developments in hardware and silicon, graphics processing on handheld devices is fast becoming a reality. Major hurdle when implementing graphics algorithms is computational complexity and enormous amount of work load involved. In order to implement graphics algorithms first step is to investigate at least one of the many existing graphics APIs. Open GL is the first industry standard graphics system developed by Silicon Graphics Inc. A subset of Open GL is Open GL ES suitable for embedded application. Open GLES is optimized for low power consumption while retaining fully programmable graphics pipeline. Here we start by studying a generic graphics pipeline and Open GL pipeline and various implementation details. Open GL graphics pipeline is a very flexible and is very complex. Discussing the whole Open GL is beyond this work and will take a textbook as a result only some parts which are relevant for implementation on hardware are discussed here.

1.1 Motivation and Purpose

Consider a display screen with resolution 640 x 480 and assume that we are trying to render an interactive graphics scene. Frame rate for interactive speeds is 25 frames per second.

\[
\begin{align*}
\text{Number of Pixels} & = 640 \times 480 = 307200. \\
\text{Total Pixels per second} & = 307200 \times 25 = 7680000. \\
\text{Time for rendering each pixel} & = 130\text{ns}.
\end{align*}
\]

From the above it is evident that in order to render at interactive rates a rendering system has 130ns to compute final pixel color. The time required to render a pixel is far less than a typical combination of memory read write and floating point computations on a general purpose processor. (In order to render a scene rendering system has to perform a series of transformations, memory operations and arithmetic operations which will be detailed when we get to introduction to graphics pipeline). Apparently it is not possible to achieve interactive speeds in graphics with general purpose processors. An effective solution is to accelerate the whole process or at least some parts of the pipeline with dedicated hardware. A graphics processor is essentially a dedicated piece of hardware with special instruction set designed to accelerate parts in particular or the whole pipeline in general.

There are many constraints to meet in order to design graphics processor for use in mobile and portable devices. Graphics processing units are big beasts consuming enormous amounts of power. However a graphics processing unit for mobile platforms
can be built with minimum resources reducing power consumption. In effect a compromise between efficiency and power consumption. This thesis draws inputs from various resources which describe statistics of a typical graphics pipeline. A graphics processing can be described as a pipe-lined process comprising of geometry engine and rasterization engine. Geometry engine works with primitives or exactly the vertices associated with primitives (points, lines and polygons) in 3D coordinate system. Rasterization engine works with these same set of primitives in 2D co-ordinate system as a result of projection to 2D screen in geometry engine.

Consider a model with 10000 triangles each generating 100 fragments (pixels) on an average when rasterized. The geometry engine essentially works with 10000 triangles each having 3 vertices, so effectively 30000 vertices (Since some vertices are common between triangles the original vertex count is much less then 30000). The number of fragments generated in this typical example is:

\[ 10000 \text{ triangles} \times 100 \text{ fragments per triangle} = 1000000 \text{ fragments} \]

From above example it is apparent that work load increases dramatically as primitives are rasterized, generating huge number of fragments for processing further down the pipeline. This is where the real work is performed from scan converting, texturing, and fog computations to various other per-fragment operations which is conceivably suitable for acceleration.

The purpose of this work is to study the whole graphics pipeline and implement a high level model of computationally intensive part. As a result I implemented rasterization part of the pipeline. It involved detailed study of different scan conversion, texturing and anti-aliasing algorithms and choosing the most suitable implementation. This serves as a foundation for porting to hardware platform and verification. The software platform used is C.

1.2 Reading Instructions

Chapter 2:
This chapter introduces to 3D graphics pipeline and the background information. This chapter is divided into two parts. First part a brief description of rendering pipeline in general from coordinate transformation, culling, clipping to shading and projection. Second part deals with rasterization of primitives, texturing and various per-fragment operations.

Chapter 3:
Implementing the rasterization engine is discussed in chapter 2. The first section talks about scan converting triangles in particular with different shading modes and interpolation of color and texture values. The second section deals with texturing and various modes in Open GL texture mapping. And finally the per-fragment operations such as alpha test, z-test and alpha blending etc are discussed.

Chapter 4:
Anti-aliasing, how multisampling combined with coverage value of pixels is used to achieve smoother edges is discussed here.

1.3 Intended Audience

This document is a starting point for those of you who want to gain a brief understanding about graphics pipe-line in general. Third and fourth chapter give a detailed description about rasterization, can be of great help if trying to build rasterization hardware based on Open GL. Reader is expected to have some knowledge in graphics terminology and pipe line.

1.4 Result

A software rasterization unit is built based on Open GL specification. Inputs to rasterization unit are triangles. Each triangle is described by a set of vertices in screen co-ordinates and additional attributes such as color and texture co-ordinates. These triangles are converted to fragments or pixels with respective \(x, y\) screen co-ordinates, color and texture coordinates. These pixels are then fed through next stages of pipe line. Texturing, fog calculations and different sets of per-fragment operations are performed based on fragment rasterization context before it is written into frame buffer for display.
Chapter 2 The Graphics Pipe-Line

2.1 Introduction

OpenGL is a software API for 3D graphics, a combination of more than 120 functions used to specify 3D models, operations and various parameters that control rendering. Rendering is the process of generating a 2D image of a 3D scene. A 3D scene is composed of models in three-dimensional space and models or composed of primitives whom the graphics system supports; most common primitives in modern graphic systems are points, lines and polygons. OpenGL is a very flexible interface between underlying graphics engine or rendering engine and graphics application. A typical graphics application can be a game or a simulation environment. Rendering engine can be implemented as pure software or hardware or a combination of both. Any modern graphics engine is a pipelined process called rendering pipe-line where a 3D scene is converted into a 2D image. CPU which runs graphics application reads through model database and issues drawing commands by passing vertices to rendering pipe-line. Remember that any complex real world model or a scene in 3D graphics is constructed by combination of small number of primitives, rendering pipe-line essentially processes vertices associated with these primitives.

Figure 2:1 Wire frame model
As mentioned earlier rendering can be best explained as a pipe-lined process. The first stage in this process is the geometry engine which essentially works with vertex data associated with primitives in three-dimension. A scene is described in three-dimensions which is called the world co-ordinate system, and task of the geometry engine is to apply a set of transformations and additional operations such as projection, clipping, culling and lightening. Each of these operations is specified by the application to control the way a scene appears on display screen. Geometry transformations handle the position and orientation of models in the scene. Projection converts a 3D scene into a 2D scene so that it can be displayed on a screen. Clipping and culling reduce the work load by eliminating obscured models. All these operations are detailed below.

First stage in geometry pipe-line is co-ordinate transformation main goal of this stage is to perform a set of transformations on model’s vertex data. Starting point in per-vertex operations is model to world co-ordinate transformation. Models are individual entities described in their own coordinate system and a typical scene consists of many

**Figure 2:3 Geometry engine**

**2.1.2 Geometry Transformation**

First stage in geometry pipe-line is co-ordinate transformation main goal of this stage is to perform a set of transformations on model’s vertex data. Starting point in per-vertex operations is model to world co-ordinate transformation. Models are individual entities described in their own coordinate system and a typical scene consists of many
such models. Model to world co-ordinate space transformation brings all models into a unified world co-ordinate system. Second operation is world to camera space (view space) transformation. Given a point in world space form where we wish to view 3D scene, world to view space transformation determines which models are visible and which are not. This is where concept of viewing volume or viewing frustum comes into picture. Once all models are in view space models are tested against viewing frustum for culling and clipping (visibility tests). Models out of viewing frustum are not visible and need not be processed further. Culling and clipping discards considerable amounts of data sets or primitives which are not visible, thereby reducing computational load for further processing. Next stage in rendering pipe-line is lightening a process by which we add light to 3D scene. Lightening model helps to determine final color for each primitive in a scene. Final stage of pipe-line is projection to 2D screen. Each primitive is essentially projected on to the 2D screen to get screen co-ordinates which are sent for rasterization. Rendering process which performs all the above operations form different co-ordinate transformations, culling, clipping, lightening to projection is called geometry engine.

2.1.3 Culling and Clipping

Culling and clipping effectively reduce amount of data sent further down the pipe line by discarding parts of models or entire models which are not visible to the viewer. This is achieved by testing models position against viewing frustum. Two types of culling operations are performed before models are sent down to lightening and rasterization stages.

1) Model Culling
2) Back Face Culling.

In model culling each visible model is tested against viewing frustum and if it is outside will be discarded. For a model made of triangle mesh culling operation amounts to determining and discarding triangles that are outside viewing frustum.

Back face culling is a process by which primitives are discarded in relation to their surface normal with viewing direction. As a result primitives which are facing away form viewing direction will be discarded. Back face culling essentially throws away polygons which are facing back; this is determined by using direction of surface normal. Each primitive has a normal pointing away from surface which it is a part of. If a normal pointing away form viewing direction primitive is back facing.

Clipping operates on front facing triangles. A front facing triangle either is completely inside viewing frustum or intersects viewing frustum; triangles which are inside viewing frustums are not clipped however triangles which intersect viewing frustum are clipped. Culling aims at reducing data sent down pipe line by eliminating primitive which are outside viewing frustum or facing away from viewing direction. On the other hand clipping results in reducing number fragments generated per primitive in rasterization part of pipeline.
2.1.4 Lightening

Lightening refers to process of determining primitive color based on a set of light sources. Final primitive color is controlled by a lot of factors for example lightening model which describes in detail the color, position and direction of each light source. In addition material properties of model and shading also contribute to primitive color. Detailed discussion about lightening model and material properties is beyond this work however shading model has relevance to what I am going to discuss in rasterization part of pipe-line so I start with a brief description of shading modes. There are three standard shading modes.

Flat Shading

In flat shading mode color is computed per primitive using surface normal, lightening model and set of other parameters which contribute to color computation. As a result during rasterization each pixel which falls inside a primitive gets same color. For a triangle a single color is assigned to each vertex and finally to all pixels which fall inside triangle.

![Figure 2:4 flat shaded triangle](image)

Gouraud Shading

Gouraud shading on the other hand assigns separate color at each vertex in a primitive depending on vertex normal and lightening model and set of other parameters discussed above which contribute to vertex color. As a result during rasterization pixel color is computed by interpolating vertex colors. For a triangle vertex colors are interpolated across edges and across scan line to arrive at a final pixel color for each pixel inside triangle.

![Figure 2:5 gouraud shaded triangle](image)
**Phong Shading**

Phong shading is computationally very demanding and complex. So far color is computed using either surface normal (for flat shading) or vertex normal (for gouraud shading). Phong shading uses pixel normal to compute color for each pixel. Vertex normal are interpolated across primitive to arrive at pixel normal for each pixel inside primitive. Color of each pixel is thus computed using pixel normal lightening model. Phong shading results in much smooth and realistic images however considering computational complexity, Phong shading will not be considered here.

### 2.2 Rasterization Engine

So far the rendering pipe-line dealt with vertices, transformed them from model space to world space, transformed form world space to camera space. All necessary vertex and primitive processing and projection are done in geometry engine. Now we are ready to deal with 2D coordinates in rasterization engine. This stage of pipe-line gets vertices associated with each primitive as 2D screen co-ordinates; primitives are essentially triangles. In addition to 2D screen co-ordinates each vertex has color and texture co-ordinates which are used to color and texture the triangle. From this point on, pipe-line gets complex with enormous amounts of data sets and computations.

Rasterization is a process of taking a 2D image and converting it into pixels or dots for output on a video display. Task of rasterization algorithm is to identify all pixels that are covered by the primitive (triangle in our case), assign color and additional attributes to each such pixel. Scan converting a triangle generates set of pixels which fall inside triangle which here-after referred as fragments. Each fragment gets its own x, y screen coordinates, color and texture coordinates computed by interpolating color, texture values across triangle edges and across scan lines.

```plaintext
for (each pixel on screen) {
    // determine if pixel is inside triangle given vertices //
    // compute color and texture coordinates depending on rasterization context.//
    // Assign color and texture values to pixel //
    // generate fragments for further processing //
}
```

![Figure 2:6 Rasterization Pipe-line](image)
### 2.2.1 Scan Conversion

To start with 2D screen can be thought as a rectangular grid of pixels where each pixel is represented by its x and y values. The x value increases along horizontal and y increases along vertical starting form bottom left corner of the rectangular grid. Where each scan line is represented by y. Rasterization unit gets 2D screen co-ordinates of vertices associated with each primitive. It’s worth mentioning here a few points about why modern graphics systems use triangle as the fundamental primitive.

Triangles are convex.

Any polygon can be decomposed into triangles.

![Figure 2:7 Triangle covering pixel centers](image)

Scan conversion starts by sorting the vertices in decreasing order of y, which is starts with top most scan line traverses down to bottom most scan line that intersects the primitive. Generates fragments for each pixel that falls inside the primitive with respective x, y screen co-ordinates. Scan conversion also assigns texture co-ordinates, color and depth values to each fragment generated depending on the shading and texturing modes. For instance flat shading assigns same RGBA values to all the fragments on the other hand gouraud shading assigns RGBA values which are interpolated across edges and across scan lines. Texture co-ordinate computation depends on affine and perspective texturing modes. Detailed description can be found in chapter 3. Task of triangle setup, edge walk and span interpolation is to generate fragments that fall inside the given triangle at the same time assign color, depth and texture co-ordinates to each such fragment. Further fragments are passed down the pipe-line for texturing and various per-fragment operations.

### 2.2.2 Texturing

Texture mapping is a process of painting an image on to a rendered surface. It is one of the most useful techniques used in modern graphics machines to render realistic images. Assume rendering a marble surface with texture mapping this task is reduced to
rendering a simple polygon of desired size and pasting the marble texture image on top of the polygon. That is color assigned to each pixel that falls inside the polygon is read from marble texture image. Texture mapping proceeds through three phases.

1. Compute texture coordinates at each pixel inside the primitive.
2. Read texels from texture image.
3. Filter texels.
4. Modify the fragment color.

Textures are simple rectangular arrays of data for example textures can be represented as rectangular grid of texels each of which can be addressed with \(<u, v>\) texture co-ordinates just as screen pixels can be addressed with \(<x, y>\) screen co-ordinates. Texels contains color information such as RGB and in some cases an alpha value \(A\) is also present. Each fragment gets its texture co-ordinates and the location of the base texture image in the texture memory. The idea of texture mapping is to read respective texel RGBA values from memory filter according to filtering mode (Point Sampled, Bilinear and Tri-linear) to remove anti-aliasing artifacts and distortion, and apply to fragments RGBA as directed by texture blend mode (Replace, Modulate and Decal).

**2.2.3 Per-fragment Operations**

Before writing textured fragments to the frame buffer Open GL specifies a set of tests which are performed on each rasterized fragment to determine whether the fragment is visible and to compute a final blend color. Each rasterized fragment goes through these series of tests which are outlined in the block diagram below. Scissor test ensures that only fragments which are inside the viewing window are sent further down the pipe-line by comparing fragments \(<x, y>\) with specified window width and height. Alpha test discards incoming fragments based on comparing fragments alpha value with a specified alpha reference value. Depth test and Alpha Blending play a major role in the rasterization process. Depth test is way to perform hidden surface removal, that is incoming fragments are tested for depth value or z value in-order to ensure only visible fragments are drawn. Alpha blending is used to draw translucent and transparent objects. A detailed discussion of how depth testing is used to perform hidden surface removal is given below followed by brief explanation about blending.

**Hidden Surface removal**

When drawing multiple polygons it is important to draw only those visible to the viewer. That is if we have number of over-lapping polygons it is essential that final pixel
color is modified by polygon which is close to the view point. As objects close to the view point will obscure objects which are farther unless they are translucent. In case of translucent objects final pixel color is a blend of overlapping polygons which we will discuss in the next section. For now in order to correctly render multiple overlapping polygons there are many algorithms such as Painter’s algorithm, Warnock’s algorithm and Z-buffer algorithm. A detailed discussion of all these here is not a good idea however one very popular algorithm which I implement here is the z-buffer algorithm. The advantages with this are easy to implement in hardware and doesn’t have any limitations. In this algorithm in addition to a frame buffer which holds current color values we also have a depth buffer to keep track of distance from view point for each pixel on display screen. Initial z-buffer values are set to maximum and when a new fragment color arrives for a pixel its z-value is compared to the z-value in the depth buffer. Then the new color and z-values are only written to the respective buffers if they are close to view point. Unlike painter’s algorithm and others, z-buffering works at fragment level. Thus each rasterized fragment gets <x, y> screen co-ordinates and a depth value <z> which helps to determine if it is close to view point when compared to the fragment already existing at screen pixel <x, y>.

![Overlapping Triangles](image.png)

*Figure 2:9 Overlapping Triangles*

### 2.2.4 Blending

Alpha blending is used to draw transparent and translucent objects. When blending is enabled Alpha value form fragments RGBA is used to determine the opacity of that fragment. Fragments RGB values determine fragments color and Alpha value corresponds to opacity a zero alpha is completely transparent and one corresponds to opaque. Blending operation combines incoming fragment color with color already existing in frame buffer at fragments <x, y>. The way in which incoming fragment color is modified is controlled by source blend factor Fs and similarly destination blend factor Fd determines how frame buffer color is modified. Alpha blending is the final operation before writing fragments to frame buffer.
Figure 2:10 Effect of Alpha blending
Chapter 3 Implementing Rasterization Engine

The rasterization process can be understood as a two stage process: the first fragment generation part which comprises of triangle edge walk and span interpolation. Fragments are generated for each pixel covered by the triangle and assigned color and texture co-ordinates. The second part is fragment processing part which basically deals with texturing, fog and set of other operations as specified by the rendering context. Rendering context is a set of flags that specify how the fragments should be treated by the fragment processing unit. Fragment processing pipe-line modifies each fragment color based on these flags for instance applying texture color, fog color and blending and so on. Output of the fragment processing pipe-line is written to frame buffer for display. A detailed description of each process is given in this chapter starting with scan conversion which generates fragments.

So far presented in this document is a brief description of a typical graphics pipeline. As a part of my work I implement the rasterization part, which deals with rasterizing triangles given the vertex, color and texture data. This chapter and rest of the document describe how a triangle rasterizer is implemented in software with considerable amount of detail. The details include computing color, texture and fog values at each pixel inside a triangle and the set of per-fragment operations as listed in OpenGL are also implemented. There are lots of choices or compromises to make when designing a graphics pipe-line. For instance the way color and texture values are computed for each pixel inside the triangle, texture filtering scheme and many more which will become apparent as we proceed. Software rasterizer starts by creating a window on screen where rendered triangle is displayed. Each vertex is represented by a vertex data structure as shown below and rasterization engine gets three such vertices associated with a triangle.

```
struct vertex{
    float x, y, z;   <x, y> screen co-ordinates of vertex, z depth.
    float r, g, b, a; r, g, b, a specify vertex color value and
    float u, v;     u and v are texture co-ordinates.
}
```

3.1 Scan Conversion

Scan conversion is a process of identifying fragments that fall inside the triangle and assign <x, y> screen co-ordinates to each such fragment. There are plenty many ways to scan convert a triangle. I choose to implement one which is less computationally expensive way that is by interpolation using incremental edge algorithm. This algorithm has been around for quite some time, is very popular and easy to understand. Even though there are some new ways based on triangle edge functions and homogenous co-ordinates.

The task of filling triangle can be broken in to two: one is to decide which pixels to fill and the other is to decide fill values. Scan conversion takes on the task to decide
which pixels to fill and fill values depends on shading and texturing mode which will be
discussed later after scan conversion. In general determining which pixels to fill consists
of taking successive scan lines form top to bottom that intersect the triangle and filling
span of adjacent pixels that lie inside the triangle form left to right. Scan conversion starts
by sorting all the vertices in decreasing order of y and proceeds from y_max through
y_min.

    struct fragment{
        float x, y, z;  //<x, y> screen co-ordinates <z> depth
        float r, g, b, a;  //Color
        float texture_u, texture_v;  //texture co-ordinates
        float f  //fog factor
    }

Pseudo code showing the scan conversion process:

For a given triangle from top scan line to bottom that is y_max to y_min:

    Step 1: Find intersections for each scan line
    Step 2: Left Intersection is start_x and right intersection end_x.
    Step 3: Fill all the pixels form start_x to end_x.
    Step 4: Decrement scan line back and to step 1.

    for( y from y_max to y_min; y--)
    {
        for( x from start_x to end_x; x++)
        {
            Generate fragment (x, y, color value)
        }
    }

We start by sorting triangle vertices in decreasing order of y. In order to get
start_x and end_x for each scan line we use linear interpolation across edges.

Figure 3:1 Triangle Edges and scan lines
Assume we have an edge joining points form \((x_1, y_1)\) and \((x_2, y_2)\) the task is to find x intercept for each scan line between \(y_1\) to \(y_2\) that is find \(x_a, x_b, x_c, \ldots, x_m\):

\[
\begin{align*}
(x_1, y_1) \\
(x_a, y_1-1) \\
(x_b, y_1-2) \\
(x_c, y_1-3) \\
\ldots \\
\ldots \\
\ldots \\
(x_m, y_2+1) \\
(x_2, y_2)
\end{align*}
\]

Interpolation using Edge Slope:

The equation for line joining \((x_1, y_1)\) and \((x_2, y_2)\)

\[(y - y_1) = m \times (x - x_1)\]

Where \(m = (y_2 - y_1) / (x_2 - x_1)\)

Now consider finding \(x_a\) the x-intercept for scan-line \((y_1-1)\)

\[((y_1 - 1) - y_1) = m \times (x_a - x_1)\]

\[x_a = x_1 - (1/m)\]

Similarly x intercept for

\[
\begin{align*}
y_1 - 2 & \quad x_b = x_1 - (2/m) = x_a - (1/m) \\
y_1 - 3 & \quad x_c = x_1 - (3/m) = x_b - (1/m)
\end{align*}
\]

From the above it is apparent that for each successive scan line x intercept changes by a value equal to inverse of slope from the previous. This forms basis for incremental edge algorithm and all other interpolation techniques employed in computing color, depth, fog and texture co-ordinates. Given a variable \(P\) with initial value and slope \(dP/dy\) for an edge computing \(P\) at successive scan-lines comes down to a simple subtraction of slope operation.

### 3.1.1 Triangle scan conversion

Now we will discuss how to scan-convert a triangle in much detail. In general there are four different types of triangles:

1. Flat bottom
2. Flat Top
3. General Triangle
   i) Right Middle
   ii) Left Middle

![Figure 3:2 Types of triangles in rasterization context](image)

We start by discussing about a right middle general triangle in detail and rests of the triangles are scan converted in very much the same way. Consider a triangle as shown in the Figure 3.3. As stated above x intercept for each successive scan line is computed iteratively by subtracting edge slope. Edge 2 is towards the left and edges 3 and 1 are towards the right. Edge slope 2 and 3 determine start\_x and end\_x for top half and edge slopes 2 and 1 for bottom half respectively.

![Figure 3: 3 Triangle with right middle](image)

In principle we start form the top vertex that is x1, y1 proceed down to the middle vertex x2, y2 and down to bottom vertex x3, y3. Here edges are named according the vertex they are opposite to and edge slops are gives by:
Dx1 = (x3 - x2) / (y3 - y2)
Dx2 = (x3 - x1) / (y3 - y1)
Dx3 = (x2 - x1) / (y2 - y1)

Figure 3:4  Scan converting Triangle

//draws top half of triangle
start_x = end_x = x1
for (y from y1 to y2)
   for (x from start_x to end_x)
      Write pixel (x, y, fill value)
   end
start_x -= Dx2
end_x -= Dx3
end

//slope transition for end_x
//draws bottom half of triangle
for (y from y2 to y3)
   for (x from start_x to end_x)
      Write pixel (x, y, fill value)
   end
start_x -= Dx2
end_x -= Dx1
end

Edge slopes which determine start_x and end_x depend on the type of triangle which is being rasterized. For instance a flat bottom triangle Dx3 and Dx2 determines start_x and end_x, for a flat top triangle Dx2 and Dx1. On the other hand a general triangle can be seen as two triangles one top half and other bottom half, a transition in
edge slope at the middle vertex takes care of the scan-conversion process. Shown in figure 3.5 below a left middle general triangle where the edge slope that determines start_x changes form D3 to D1 after transition point V2.

![Figure 3:5 Triangle with left middle](image)

After sorting vertices in decreasing order of y:

```java
if (y1 == y2)
    //draw flat top
    start_x -= Dx2
    end_x -= Dx1

if (y2 == y3)
    //draw flat bottom
    start_x -= Dx3
    end_x -= Dx2

if (Dx3 > Dx2)
    //draw Right Middle
else
    //draw Left Middle
```

So far we have discussed what pixels to fill in and next sections of this document discuss about fill values. As mentioned before deciding on fill values depends on shading and texturing modes. Rest of the chapter discusses about shading modes such as flat and gouraud shading. Texturing is explained in detail in texturing section.

### 3.1.2 Shading - Interpolating Color

**Flat Shading**

A triangle’s vertex data structure in addition to screen co-ordinates x and y will also contain RGBA color values which are used to assign color to individual fragments
during scan conversion. As discussed in lightening section in chapter 2 flat shading assigns single color to a primitive depending on surface normal as a result, all the pixels that fall inside are assigned the same color. Flat shading is really very simple to implement. To fill a triangle with flat shading we set same color to each pixel inside the triangle.

**Gouraud shading**

Unlike flat shading gouraud shading is a type of smooth shading where color is computed per-vertex depending on the vertex normal. As a result each vertex gets a different color. To fill a triangle with smooth shading we interpolate vertex colors across edges and across scan lines much the same way as we interpolated x during scan-conversion.

For instance consider a general triangle with vertices v1, v2, and v3. Additionally each vertex has a RGBA color value. Remember scan converting a triangle where we computed start_x and end_x for each scan line as we move down the triangle, we simply subtract edge slope which is basically change in x with respect to change in y. This is what exactly we will do to compute left color and right color values for each scan line. Interpolate colors down the edges as a function of y. Now we have left color and right color value for each scan line in order to arrive at final color for each pixel fragment we interpolate end color values with respect to x. The pseudo code below shows how to render a smoothly shaded triangle. Here I have discussed smooth shading a general triangle with right middle similarly all the triangle types discussed previously can be rendered by simply interpolating color deltas across left and right edges and across scan lines.

\[
C = R, G, B \text{ and } A;
\]

\[
\text{Edge slopes}\]
\[
dC1 = (C3 - C2) / (y3 - y2)
\]
\[
dC2 = (C3 - C1) / (y3 - y1)
\]
\[
dC3 = (C2 - C1) / (y2 - y1)
\]

**Horizontal span**

\[
Cspan = (right\_C - left\_C) / (end\_x - start\_x)
\]

//draws top half of triangle
start\_x = end\_x = x1;
left\_C = right\_C = C1;
for(y from y1 to y2)
  fill value = left\_C
  Cspan = (right\_C - left\_C) / (end\_x - start\_x)
  for(x from start\_x to end\_x)
    Write pixel(x, y, fill value)
    fill value += Cspan;
end
start\_x -= Dx2
end\_x -= Dx3
left\_C -= dC2
right_C -= dC3
end
//slope transition for end_x
//draws bottom half of triangle
for(y form y2 to y3)
  fill value = left_c
  Cspan = (right_c – left_c)/(end_x – start_x)
  for(x form start_x to end_x)
    Write pixel(x, y, fill value)
    fill value += Cspan;
  end
  start_x -= Dx2
  end_x -= Dx1
  left_C -=d C2
  right_C -=d C1
end

Figure 3:6 Scan converting – Interpolating Color

3.1.3 Interpolating Texture Co-ordinates

In addition to RGBA color values each vertex gets two dimensional texture co-ordinates <u, v>. These texture co-ordinates are interpolated similarly as RGBA color values to arrive at the final pixel’s texture co-ordinates. Figure 3.7 below shows a
rectangular brick wall texture which is split into two triangles. Each vertex gets normalized texture co-ordinates which are interpolated to get texture co-ordinates at each pixel inside the triangle and are used to read texels from texture map. When these two triangles are scan converted and texture mapped will exactly look like a brick wall as shown in the texture image.

Figure 3:7 Texture mapping

Two popular texturing techniques are

1. Affine texture mapping.
2. Perspective texture mapping.

Affine texture mapping is by far the most easy and computationally less expensive technique. This simply interpolates texture values across left and right edges and across scan lines. This technique doesn’t take into consideration z depth at each vertex while interpolating texture co-ordinates as a result images look distorted when rendered. Interpolating <u, v> is done in a same way as RGBA color channels.

Compute du/dy and dv/dy for left and right edges.
Compute left <u, v> and right <u, v> for successive scan line
Interpolate left<u, v> and right <u, v> along the scan line with respect to x.
Assign interpolated <u, v> texture co-ordinate to each fragment.

Consider the triangle as shown in the figure 3.8 below. Now we have <u, v> in addition to x, and RGBA to interpolate so a total of seven values.

\[ T = u, v \]

Edge slopes
\[ dT1 = (T3 - T2) / (y3 - y2) \]
\[ dT2 = (T3 - T1) / (y3 - y1) \]
\[ dT3 = (T2 - T1) / (y2 - y1) \]

Horizontal span
\[ T_{span} = (right_T - left_T) / (end_x - start_x) \]
Figure 3:8  Scan conversion – Interpolating Texture co-ordinates

Scan converting a textured triangle is very similar to rendering a gouraud shaded triangle as discussed above. We start by sorting vertices in decreasing order of y and proceed from top scan line which intersects the triangle to bottom. The only difference is addition of two more variables <u, v> to the set of interpolating variables. Affine texture mapping woks fine for surfaces that make a small angle with the viewing direction as variation in z is minimal across the surface. However as the angle increases z depth varies considerably resulting in visible distortion in textured surface. Figures 3.9 and 3.10 below illustrates a two edges one perpendicular to the view direction and other inclined. It is apparent that edges that make an angle with the viewing direction contribute to visual distortion. As can be seen the second figure 3.10, as edge depth increases.

One way to remove distortions in rendered texture is to perspectively interpolate texture co-ordinates. Perspective texture mapping is akin to perspective projection that is it takes z depth at each vertex into consideration for interpolating <u, v>. Affine texture mapping and gouraud shading are all based on linear interpolation. The problem with linear interpolation is that we can’t use it to interpolate any thing that is not linear in screen space. That is uniform steps in screen space doesn’t equate to uniform steps in world space. As texturing is performed in screen space after geometry transformation (which other wise looks nice in world space) looks distorted. In perspective texturing we interpolate <u’ = u/z, v’ = v/z> instead of <u, v> and 1/z across triangle edges and scan lines. Finally to get back original <u, v> we divide <u’, v’> by 1/z.
Figure 3:9  Demonstrating no perspective distortion in edges

Figure 3:10  demonstrating perspective distortion in edge
The table below shows an example of perspective interpolation.

\[ y_1 = 1, z_1 = 10, u_1 = 0. \]
\[ y_2 = 7, z_2 = 20, u_2 = 12. \]
\[ y_1 = 1, z_1' = .1, u_1' = 0. \]
\[ y_2 = 7, z_2' = 0.05, u_2' = 0.6. \]

\[
d y = y_2 - y_1 \\
 dz' = \frac{1}{z_2} - \frac{1}{z_1} \\
 du' = \frac{u_2}{z_2} - \frac{u_1}{z_1}
\]

<table>
<thead>
<tr>
<th>y1</th>
<th>u (linear)</th>
<th>z’</th>
<th>u’</th>
<th>u = u’/z’ (perspective)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0.091</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0.083</td>
<td>0.2</td>
<td>2.4</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>0.075</td>
<td>0.3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>0.066</td>
<td>0.4</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>0.058</td>
<td>0.5</td>
<td>8.5</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>0.05</td>
<td>0.6</td>
<td>12</td>
</tr>
</tbody>
</table>

Figure 3:11    affine texture interpolation

Figure 3:12   Perspective texture interpolation
Perspective interpolation

As shown in the figure 3.15 below to render a perspectively correct textured triangle we start by computing \(u', v'\) and \(z'\) at each vertex and interpolate them along triangle edges and along scan lines. For each scan line in addition to \(x\) and color gradients we also get left and right \(u', v', z'\) Thus for each fragment inside the triangle we get interpolated \(u', v'\) and \(z'\). Final fragment’s texture co-ordinates are computed as:

\[
\begin{align*}
    u &= u'/z' \\
    v &= v'/z'
\end{align*}
\]

Figure 3:13 Affine texture mapping – checker board

Figure 3:14  perspective texture mapping – checker board

Thus far we have interpolated \(x\) screen co-ordinates, color RGBA and \(u, v\), perspective interpolation adds another component to the list of interpolates that is the depth \(z'\).
3.1.4 Interpolating Fog

Fog is used to simulate haze, mist, dust, smoke and different atmospheric effects. Addition of fog adds realism to rendered scene by blending the specified fog color with incoming fragment's color using fog blend factor. This factor increases with z distance from viewpoint. This makes distant objects blend with their surroundings, as the distance from viewpoint increases. The factor f is computed by one of the three equations shown below and clamped to [0, 1]. Where z is distance from viewpoint, values density, start and end are specified prior for each object for which fog is enabled. Fog is applied after geometry transformation, lightening and texturing, so it affects transformed lit and textured objects.

\[ f = e - (density \cdot z) \quad \text{EXP} \]
\[ f = e - (density \cdot z)^2 \quad \text{EXP2} \]
\[ f = (end - z) / (end - start) \quad \text{LINEAR} \]

After calculating fog factor f at each vertex fog for each fragment inside the triangle is computed by interpolating f across edges and across scan lines similarly as RGBA color and other variables. Final fragment color based on f will be modified later to texturing stage. After doing all the necessary interpolation each fragment gets its own screen co-ordinates <x, y>, color RGBA, fog and texture co-ordinates <u, v> either linearly interpolated or perspectively interpolated. Finally each fragment thus generated
will go through texturing, fog and set of other operations which are detailed in the following sections.

### 3.2 Texturing

The first stage after scan conversion is texture mapping. The task of texture mapping is to read texels from texture memory and modify the pixel color. This is performed by reading normalized texture co-ordinates \( <u, v> \) associated with each fragment, converting them to memory locations from where RGBA texel values are looked up for. Following are the steps performed in texturing.

1. **Read each fragments** \( <u, v> \) **texture co-ordinates.**
2. **Round off** \( <u, v> \) **depending on texture wrap mode.**
3. **Look up texels in texture memory.**
4. **Filter texels based on filtering mode.**
5. **Modify fragment color depending on texture blend mode.**

Two texture wrap modes are specified CLAMP and REPEAT, in clamp mode \( <u, v> \) values greater than 1.0 are set to 1.0 and values less than 0.0 are set to 0.0, on the other hand in repeat mode integer part of \( <u, v> \) if beyond \([0, 1]\) is ignored. Repeat mode is generally used to texture map a large polygon with same repeating texture. Figure 3.16 below shows a checker board pattern applied to a polygon in these two modes.

![Texture clamp modes Clamp and repeat](image)

**Figure 3:16  Texture clamp modes Clamp and repeat**

After rounding off \( <u, v> \) the way texels are read form texture space depends on one of three filtering modes. Three most common filtering modes are:

1. **Point sampling.**
2. **Bi-Linear filtering.**
3. **Tri-Linear filtering.**

#### 3.2.1 Texture filtering

Point sampling is the simplest and fastest technique to apply texture to a polygon. This starts by taking \( <u, v, c(u, v)> \) where \( c \) is the texture space. Rounds off \( <u, v> \) to nearest integer values and fetch nearest texel color.
Pseudo code showing point sample texture mapping:

\[
\begin{align*}
    u_{\text{temp}} &= \text{TexW} \times u; \quad \text{//convert normalized texture co-ordinates to-} \\
    v_{\text{temp}} &= \text{TexH} \times v; \quad \text{//-texture space values} \\
    u_{\text{temp}} &= \text{floor} \ (u_{\text{temp}}); \quad \text{//truncate off to nearest integer} \\
    v_{\text{temp}} &= \text{floor} \ (v_{\text{temp}}); \\
    (r, g, b) &= \text{TEX} \ [u_{\text{temp}}, v_{\text{temp}}]; \quad \text{//texture offset} \\
    \text{Fragment -> color} &= (r, g, b); \quad \text{//modify fragment color}
\end{align*}
\]

Where \( \text{TEX} \ [u, v] = tp \ [3 \times \text{TexW} \times v + 3u + c] \)
\( c \) is 0 for R, 1 for G, 2 for B color values.

For instance:

\[
\begin{align*}
    u &= 0.71 \\
    v &= 0.56 \\
    \text{TexH} &= 256 \\
    \text{TexW} &= 256 \\
    u_{\text{temp}} &= 181.76 = 181 \\
    v_{\text{temp}} &= 130.56 = 130
\end{align*}
\]

A texture of above size will occupy 256 x 256 x 3 bytes in memory which is equal to 196608 bytes.

Red offset = \ 101154.
Green = \ 101155.
Blue = \ 101156.

Point sampling as mentioned before is the simplest of all texture mapping techniques it takes into consideration the nearest texel specified by \( <u, v> \) texture co-ordinates. However it leads to a lot of undesirable effects such as texture distortion and
aliasing. As a triangle goes through the process of geometry transformation they get distorted as a result of translation, rotation and scaling, when these distorted triangles are textured the texture as well will look distorted. For instance as polygons are placed far away from the viewpoint they appear smaller thus covering less number of screen pixels on the contrary if the same polygon is drawn close to view point it appears bigger thus covering more number of screen pixels. When such polygons are texture mapped, then one texel may be mapped to a single screen pixel or a single texel is mapped to more then one screen pixel this leads to distortion. One way to avoid such distortion and aliasing effects is to filter texels. In addition texture mapping can be thought as a sampling process where \(<u, v>\) denote discrete sampling points in texture space. In order to render smooth textures filtering needs to be performed. For example if multiple texels map to a single fragment a weighted average of texels is computed.

Bilinear filtering helps generate smooth textures, figure 3.18 below shows the effect of point sampling and filtering. Point sampling introduces jagged edges and an alias effect on the other hand with bi-linear interpolation a much smoother image is obtained. As mentioned in the previous section when computing final sample texel we simply truncate the floating point co-ordinate values to integer values thus pointing to single nearest texel. In bilinear interpolation we use four nearest texel colors to texture a fragment. The key idea if we want to find texel color at \(<u, v>\) we use weighted combination of texels at \(<u+1, v>\), \(<u, v+1>\), \(<u+1, v+1>\). The process of bi-linear interpolation requires four weight factors to be computed. As each fragment arrives to the texturing unit it has its own normalized texture co-ordinates which are converted to normal texture pointers by multiplying by texture width and height.

\[
\text{Original Image}
\]

\[
\text{Point sampling} \quad \text{Bilinear Interpolation}
\]

*Figure 3.18 Point sampling Vs Bi-linear filtering*

Weights are computed from the fractional parts of these texture pointers this process is outlined in the pseudo code shown below. In addition to this the figures 3.19 show a visual illustration of the filtering process. Assume a texture where we want to compute the final pixel color from four neighboring texels \(p0, p1, p2, p3\) we take into
consideration the overlapping pattern to arrive at the final color. In case (a) the texture co-ordinates have no fractional part consequently the final pixel color is corresponding to a single texel p0, in other cases texture co-ordinates have fractional parts which are used to average out surrounding texels.

\[ \text{u}_t = u \times \text{texW} \quad \text{//convert normalized texture co-ordinates to real texture pointers} \]
\[ \text{v}_t = v \times \text{texH} \]

\[ x = \text{floor} \ (u_t) \quad \text{//rounding off} \]
\[ y = \text{floor} \ (v_t) \]

\[ u_{\text{fraction}} = u_t - x \]
\[ v_{\text{fraction}} = v_t - y \]

\[ u_{\text{opposite}} = 1 - u_{\text{fraction}} \]
\[ v_{\text{opposite}} = 1 - v_{\text{fraction}} \]

\[ \text{color} = \text{texture} \ [x, y] \times u_{\text{opposite}} \times v_{\text{opposite}} + \text{texture} \ [x+1, y] \times u_{\text{fraction}} \times v_{\text{opposite}} + \text{texture} \ [x, y+1] \times u_{\text{opposite}} \times v_{\text{fraction}} + \text{texture} \ [x+1, y+1] \times u_{\text{fraction}} \times v_{\text{fraction}}. \]

*Figure 3:19* Bi-linear filtering
The only problem with this kind of filtering is there is a lot of computational overhead that is reading four texels from memory and multiplying them with their respective weights. Bi-linear texturing works fine for most of the cases however when polygons are small or angled steeply to the viewing direction results in artifacts. One way to get rid of this problem is to use Tri-linear interpolation. Unlike bi-linear filtering which uses only one texture image to average out texels tri-linear filtering uses texels from two similar textures of different resolution to compute final texel color. Tri-linear filtering uses mipmapped images to achieve a fine result which is discussed in next section.

Mip-Mapping and Tri-Linear Filtering

Even Bi-linear filtering can result in aliasing when polygons far from view point are textured. As the distance from the view point increases the polygon appears small thus covering less number of pixels. And as the same texture co-ordinates are applied over a smaller number of pixels texture aliasing occurs. One way to overcome this aliasing is by use of mipmaps. Mipmaps are scaled down versions of the base texture image consider for instance a base texture of size 512 x 512, we generate multiple images at 256 x 256, 128 x 128, 64 x 64 and so on. If the original texture is a square of size \(2n \times 2n\) there are \(n\) down sampled versions. Mipmaps are built by averaging four neighboring texels down to one. Image with highest resolution is the base texture and is called level-0 texture and each reduced images are assigned level-1, level-2 and so on. One major advantage of mipmaps is images of high resolution are used to texture polygons close to the view point and images with smaller resolution for polygons that are far. Mipmaps considerably improve the quality of rendered image however with substantial increase in memory usage.

Another advantage of mipmaps is Tri-linear filtering where texels from neighboring mipmap levels are averaged to arrive at the final texel color. Bi-linear
interpolation works fine in most of the cases however some aliasing effects are still visible when polygons that make a large angle with viewing direction are textured. Tri-linear filtering can be thought as a simple extension to bi-linear filtering where texels form two neighboring mipmap levels are interpolated. For instance consider the computed mipmap level is 3.5 texels from level-3 and level-4 texels are combined to arrive at the final texel color.

![Mip Mapping and Tri-linear texturing](image)

**Figure 3:21 Mip-maps chain**

Texture Co-ordinate

$\langle u, v \rangle = (0.3, 0.7)$

![Tri-linear interpolation](image)

**Figure 3:22 Tri-linear filtering between mipmaps**
Given normalized texture co-ordinates <u, v> for a fragment tri-linear interpolation proceeds by computing texel color form individual mipmaps through bi-linear interpolation. That is multiply <u, v> by corresponding texture width and height and computing the weighted average of four neighboring texels. Final texel colors thus computed form both mipmap levels are averaged to arrive at the final fragment color.

### 3.2.2 Texture Blending

The fragment color is modified depending on texture blending modes. After computing texel color using one of the filtering modes, blending operation controls the way texel color is applied to fragment. Three blending modes are specified in RGBA mode.

- **Replace**
- **Modulate**
- **Decal**

Shown below how final fragment color is computed in various modes:

C and A are fragment color RGB and alpha A respectively.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPLACE</td>
<td>$C_f = C_t; A_f = A_t$</td>
</tr>
<tr>
<td>MODULATE</td>
<td>$C_f = C_i C_t; A_f = A_i A_t$</td>
</tr>
<tr>
<td>DECAL</td>
<td>$C_f = C_i (1 - A_t) + C_t A_t; A_f = A_i$</td>
</tr>
</tbody>
</table>

### 3.2.3 Fog

Once fragments are textured effect of fog is given by modifying the fragment color as:

$$C = f * C_i + (1-f) * C_f$$

Where $C_i$ is in coming fragments color (RGBA)

$C_f$ is fog color (RGB)

f is fragment’s fog factor.

There are two ways to compute fog factor:

- Per-vertex fog.
- Per-pixel fog.
Per-vertex fog computes fog factor at each vertex depending on EXP, EXP2 or LINEAR modes and interpolates these across triangle edges and scan lines similarly as RGBA color values are interpolated, arriving at a fog factor for each pixel inside the triangle. On the other hand per-pixel fog computes fog factor for each fragment inside the triangle using interpolated z values. Each fragment inside the triangle gets an interpolated z value that is used to compute fog factor depending on EXP, EXP2 or LINEAR mode. Per-pixel fog results in good amount of scene realism however it is computationally expensive keeping in mind the fog factor computation at each pixel. In both cases fragment’s final color is computed using the above equation for final fragment color.

3.3 Per-fragment Operations

So far we have discussed the scan conversion on per-primitive basis, texturing and fog computations on per-fragment basis. Scan conversion generates fragments; texturing and fog decide final color for each such generated fragment. The graphics pipeline before writing these fragments into frame buffer as pixels, performs a set of operations on individual fragments. Some tests decide fate of each fragment, if it passes the test it is sent further down the pipeline else discarded and some modify the fragment color to achieve more realism. List of these operations are listed below.

Scissor test
Alpha test
Z test
Alpha blending

3.3.1 Scissor Test

Scissor test determines if a fragment’s \(<x, y>\) are in-side scissor rectangle. Fragments are sent for further processing depending on the outcome of the test. Scissor rectangle is specified by setting values for left, bottom, width and height. If the fragments screen co-ordinates are between these values it passes the test. This test is enabled by setting scissor_test flag, if scissor test is enabled all incoming fragment’s \(<x, y>\) are tested against the scissor rectangle and only those pass will be processed further. If not enabled all fragments pass the test and are sent down the pipeline.

3.3.2 Alpha Test

Alpha test is enabled by setting alpha_test flag. If enabled each incoming fragment’s alpha \(A_i\) is tested against a pre-assigned alpha reference \(A_{ref}\) value. One of eight functions determines the way alpha test is performed, which are listed below. Fragments which pass the test are retained and the rest discarded.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEVER</td>
<td>Never pass fragments</td>
</tr>
<tr>
<td>ALWAYS</td>
<td>Always pass fragments</td>
</tr>
<tr>
<td>LESS</td>
<td>Pass if (A_i &lt; A_{ref})</td>
</tr>
<tr>
<td>LEQUAL</td>
<td>Pass if (A_i \leq A_{ref})</td>
</tr>
<tr>
<td>EQUAL</td>
<td>Pass if (A_i = A_{ref})</td>
</tr>
</tbody>
</table>
GEQUAL Pass if $Ai \geq A_{\text{ref}}$
GREATER Pass if $Ai > A_{\text{ref}}$
NOTEQUAL pass if $Ai \neq A_{\text{ref}}$

3.3.3 Depth Test

As mentioned in chapter-2 section-2.2.3, when drawing multiple over-lapping polygons it is necessary to consider each fragment’s depth. It ensures that pixel color is modified by fragment which is close to the view point. Depth test or z-test compares incoming fragment’s z against z value stored in the depth buffer specified by fragment <x, y>. Depth buffer as stated before is a rectangular array of cells, each cell stores current depth value (distance from view point) of its corresponding pixel in frame buffer. If the fragment passes the test its z and color values replace corresponding pixel in frame buffer. Depth test is generally used for hidden surface removal that is to discard pixels covered by opaque objects close to view point.

```c
Write Pixel (int x, int y, float z, color){
    if ( z < zbuf [x, y]){  
        zbuf [x, y] = z;
        Frame Buffer [x, y] = color;
    }
}
```

Depth test is enabled by setting z_test flag. One of eight ways to control the way z-test is performed are listed below:

NEVER Never pass fragment
ALWAYS Always pass fragments
LESS Pass if $Zi < Zbuf [x, y]$
LEQUAL Pass if $Zi \leq Zbuf [x, y]$
EQUAL Pass if $Zi = Zbuf [x, y]$
GEQUAL Pass if $Zi \geq Zbuf [x, y]$
GREATER Pass if $Zi > Zbuf [x, y]$
NOTEQUAL Pass if $Zi \neq Zbuf [x, y]$

Where $Zi$ is incoming fragments depth.

3.3.4 Alpha Blending

Alpha blending is the final operation performed on fragments before writing them into the frame buffer. Alpha blending if enabled helps to create various translucency and transparency effects. This is performed as a two stage process; first each incoming fragment color (source color) is modified according to source blend factor and the pixel color (destination color) which is already stored in frame buffer at fragments <x, y> is modified according to destination blend factor. Final fragment color is computed by combining both modified source and destination colors. For instance assume $Rs$ $Gs$ $Bs$ $As$, $Rd$ $Gd$ $Bd$ $Ad$ are source and destination colors respectively and $Fs$, $Fd$ are source and destination blend factors respectively. Then final fragment color is given by:
RGBA = FsRs + FdRd, FsGs + FdGd, FsBs + FdBd, FsAs + FdAd

Table below shows different combinations for computing source and destination blend factors. The first column lists name of each function and second column lists if the function is applicable to source color or destination color are both and final column lists the computed blend factor. For instance if source blend factor is SRC_ALPHA and destination blend factor is ONE_MIN_SRC_ALPHA, Fs, Fd and final fragment color are computed as follows:

SRC_ALPHA = source alpha As; ONE_MIN_SRC_ALPHA = one minus source alpha (1 - As)
Fs = As
Fd = 1 - As
Final fragment color is given by

As (Rs Gs Bs As) + (1 - As) (Rd Gd Bd Ad)

On the other hand if source blend factor and destination blend factor are DST_COLOR and SRC_COLOR respectively then Fs = Rd Gd Bd Ad; Fd = Rs Gs Bs As. Final fragment color is 2 (Rs Rd, Gs Gd, Bs Bd, As Ad).

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>FACTOR SRC/DST</th>
<th>Computed Blend factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZERO</td>
<td>src or dst</td>
<td>0</td>
</tr>
<tr>
<td>ONE</td>
<td>src or dst</td>
<td>1</td>
</tr>
<tr>
<td>DST_COLOR</td>
<td>src</td>
<td>Rd Gd Bd Ad</td>
</tr>
<tr>
<td>SRC_COLOR</td>
<td>dst</td>
<td>Rs Gs Bs As</td>
</tr>
<tr>
<td>ONE_MIN_DST_COLOR</td>
<td>src</td>
<td>1 - Rd, 1 – Gd, 1 - Bd, 1 - Ad</td>
</tr>
<tr>
<td>ONE_MIN_SRC-COLOR</td>
<td>dst</td>
<td>1 - Rs, 1 - Gs, 1 - Bs, 1 - As</td>
</tr>
<tr>
<td>SRC_ALPHA</td>
<td>src or dst</td>
<td>As</td>
</tr>
<tr>
<td>ONE_MIN_SRC_ALPHA</td>
<td>src or dst</td>
<td>1 - As</td>
</tr>
<tr>
<td>DST_ALPHA</td>
<td>src or dst</td>
<td>Ad</td>
</tr>
<tr>
<td>ONE_MIN_DST_ALPHA</td>
<td>src or dst</td>
<td>1 - Ad</td>
</tr>
<tr>
<td>SRC_ALPHA_SATURATE</td>
<td>src</td>
<td>Fs = min (As, 1-Ad)</td>
</tr>
</tbody>
</table>
One important thing to consider when rendering translucent polygons is the order they are rasterized. For instance consider two polygons A and B at depth Za and Zb respectively where Za < Zb. That is polygon A is closer to view point than polygon B. If polygon A is rasterized first it passes z-test and is written to screen with transparency. Polygon B should be visible when rendered correctly since polygon A is translucent however polygon B fails the depth test and will not be drawn on screen since it is behind polygon A see figure 3.23. The order in which polygons are rasterized and drawn greatly effects the blending effect in final result. While drawing translucent objects in three-dimension is necessary to consider depth testing. Depth testing is generally used for hidden surface removal depth buffer keeps track of distance between view point and the object pixel. When a new fragment arrives for the same pixel it is only drawn if closer to view point and depth value is replaced. If polygons are drawn with depth testing obscured polygons are not drawn and therefore aren’t used for blending.

In order to render both opaque and translucent polygons correctly perform blending and hidden surface. One way is to sort polygons form back to front thus farther polygon are drawn first. Thus polygons that are obscured by opaque ones are removed by hidden surface removal or depth test and translucent polygons are drawn with blending. This helps solve the problem however when objects are moving z-sorting polygons becomes too big a task. Another way is to draw all opaque polygons followed by translucent ones. In this case we first draw all opaque objects or fragments with z-testing and z-write. Thus pixel color is modified by fragments closer to the view point. However while drawing translucent polygons only depth test is performed. This ensures translucent objects or fragments can’t overwrite opaque objects as depth values are not changed and also objects that are close to view point are drawn with blending.

Figure 3:23 Two overlapping triangles
Chapter 4 Anti-Aliasing

So for we have discussed in detail about generating fragments by scan converting triangles and little attention is placed on the concept of triangle edge aliasing. One very important way to increase the image quality is to generate smoother triangle edges. What we have discussed so far results in blocky and jazzed edges. First thing triangle setup does when it gets a triangle is, determine which scan lines are located inside the triangle by checking all the three vertices. Then the scan conversion process iterates through the scan lines from maximum of y to minimum of y and successively computes for each scan line where the triangle starts and ends. The triangle setup we have discussed so far actually processes pixel centers that is if a pixel center in inside the triangle, the entire pixel is treated to be a part of the triangle and generates fragment with interpolated color and texture co-ordinates. However in reality majority of pixels on the triangle edges are partially covered. As a result of this rounding a triangle edge which is otherwise smooth appears as stepped or jazzed. These steps contribute to visual distortion in the rendered image. In the figure 4.1 below pixels with solid color are those with their centers inside the triangle and pixels with light shade are pixels which are partially covered by the triangle that is they fall on the edge.

![Figure 4:1 Top blocky edges; Bottom edge pixels with light shade](image-url)
There are plenty many ways to achieve smoother edges. One way is to render the whole scene at twice the resolution and filter back to original resolution before displaying on screen. This is done by, for instance consider a 128 x 128 scene which is rendered as 256 x 256 scene and finally before writing to the frame buffer four neighboring pixels are averaged down to single pixel much the same way as used to construct mipmaps. This technique is called over-sampled anti-aliasing. One major draw back of this technique is substantial increase in memory and computational load to the rendering pipe-line. However super sample anti-aliasing gives full screen anti aliasing and much better images at the cost of twice the computational load.

Another way is to sample a single pixel multiple times, a technique called multi-sample anti-aliasing. Both over-sampling and multi-sampling are super-sampling techniques where more then two samples are used to compute final pixel color. Unlike the previous technique where anti-aliasing operation is performed for the whole screen image; multi sample anti-aliasing process only triangle edges. Pixels along edges are tested for coverage that is how much of the pixel in question is inside the triangle and thus assigning color based on the coverage value. With this technique triangle edges look much smoother by achieving a proper blend between edge pixel color and back ground color. A detailed description of multi-sample rasterization and edge anti-aliasing is given below.

4.1 Multi-sample Anti-Aliasing

The basic idea behind multi-sample rasterization is to sample single pixel at four different points called sub pixels. A visual representation is shown in the figure 4.2 below. These four different sample points determine the final coverage value of the given pixel. A pixel totally inside the triangle has a coverage value of 100% and pixels on edges have varying coverage value. This coverage value is effectively used as a blending factor for blending triangle color with back ground color. As we understand the task of triangle setup is to generate fragments which are set down the pipe-line for texturing and various operations. One these fragments reach end of the fragment processing pipe-line just before writing them to the frame buffer; each fragments coverage value is checked to determine how fragments color will affect the pixel color. As edge fragments have varying coverage values, will result in color that blends with back ground which results in smoother edges.

![Sub pixels – Ordered grid multi sampling](image-url)
The multi sampling which I am going to discuss is called ordered grid multi-
sampling which smoothen edges which are not close to horizontal or vertical slopes. 
There are multiple sampling grid patterns such as rotated grid and random grid however 
ordered grid is relatively easy to implement and less complex and one major advantage is 
we can incorporate support to multi sampling to the rasterization base we have developed 
so far. The starting point for generating fragments is to start form top scan line 
intersecting the triangle and determine where the triangle starts and ends. Triangle setup 
process essentially passes scan line y, start and end of x to the fragment generating unit. 
Fragment generation unit starts by testing each sub pixel against the triangle edge 
equation. That is given a fragment <x, y> and the line equation of the edge; task is to 
determine each sub pixels position relative to the edge. Each sub pixel inside the triangle 
gives a 25% coverage value and consequently all four sub pixels inside give the fragment 
a 100% coverage value.

![Figure 4:3 Sub pixels – Partially and fully covered by triangle edge](image)

In the figure 4.3 above we can see each pixel represented by a bold border is divided 
into four sub pixels and a triangle edge. Sub pixels which are covered by the edge are 
shown as well. Assuming pixel center is <x, y> then the four sub pixel positions are 
offset by +/- 0.25 in both x and y directions. Now consider a scan line y with start_x and 
end_x, scan line y essentially passes through the pixel center at least that is what we 
assume. We start by determining start_x and end_x for what we call sub scan lines that 
pass through the sub pixels. Two sub scan lines are one which passes through top two sub 
pixels and other through bottom two sub pixels. Once we have start and end values for 
each scan line finding the coverage for a given fragment is done by comparing sub pixels 
against the start and end values. The figure 4.4 below shows a triangle left edge passing 
through a pixel fragment. Initial start_x usually corresponds to the pixel center and to 
compute start_x1 and start_x2 from start_x we can iteratively add and subtract 0.25 x left 
edge delta or use triangle edge equation. The triangle edge equation is simply equation of 
line joining two corresponding edge vertices. The coverage factor of the fragment shown 
above is 75%. This is determined by comparing sub pixel x co-ordinate against start_x1 
and start_x2.
Figure 4:4 Sub pixels – Edge through a single pixel

Figure 4:5 Left figure No edge anti-aliasing; Right figure ordered grid multi sample anti-aliasing
Chapter 5 Conclusions

This work is a first step towards understanding the graphics pipe-line and implementing the rasterization part in hardware. The second chapter gives a brief description of a general graphics pipe-line. Chapter three implementing the rasterization engine discusses specifically in detail how a triangle is rasterized given the rendering context and 2D screen co-ordinates. Rasterization context controls various operations performed while generating fragments for each pixel inside the triangle and set of operations performed on these individual fragments. Rasterization context can be best understood as a structure comprising of flags each of which controls coloring, texturing and fog operations. Rendering engine checks these flags for instance when computing fragment color, it checks for whether flat shading or gouraud shading is to be used similarly if texturing is enabled texture co-ordinates are interpolated and assigned to each generated fragment.

While rasterizing triangles there are plenty many choices to be made numerous flags to be checked. There are many ways to implement a rasterization engine one way is to have a dedicated routine for each set of combinations. For instance flat shading plus affine texturing with z buffering can be implemented as one routine and gouraud shading plus perspective texturing with alpha test and alpha blending can be implemented one routine. One advantage to have a dedicated routine for a set of combinations is speed as that single routine is optimized to perform a specific set of tasks specified by rasterization context. With this implementation we will get numerous functions implementing different combinations of rendering choices. However this is not a very encouraging idea for implementing in hardware.

What we essentially do during scan conversion is we generate fragments for each pixel covered by the triangle assign color and texture co-ordinates to these fragments. These fragments are stored in memory possibly a FIFO and the hardware texturing unit reads each fragment textures it and sends it to the next unit that is fog. If fog is enabled for that fragment, fog unit modifies the fragment color depending on fog mode. The whole process form generating fragments, texturing, fog, scissor, alpha test, z test to blending can be seen a pipe-lined process. Where each unit reads a fragment modifies it pushes it to the next unit. And also fragments that doesn’t need the attention of a unit can skip that unit and proceed to the next, for instance fragments for which fog is not enabled can proceed to next stage that is scissor test and other per-fragment operations. And finally after getting processed by all units fragments are written to the frame buffer.

By reading the chapter on implementing the rasterization engine one can get an idea about the set of operations that are performed. The whole rasterization process can viewed as a two stage process one which generates fragments and the next consumes fragments. In fragment generation process operations are limited to arithmetic computations and no access to texture memory or frame buffer memory is needed. On the other hand in the fragment consuming part specially texturing needs access to texture memory, z-test needs access to depth buffer in memory and blending needs access to the whole frame buffer memory.
Arithmetic computations in the first block are computing deltas along triangle’s three edges and along spans. A total of seven deltas need to be computed assuming gouraud shading and perspective texturing. In the fragment consuming part computations are usually interpolating RGBA color and access to different memory blocks.

5.1 Future Work

What I have implemented so far is a small yet very flexible rasterization engine. As it is apparent that graphics pipeline is relatively large with lot of choices to make. However there is always room for optimization and fine tuning the software. In addition to this computer graphics is an emerging field and there is always some thing new to be implemented and tested. For instance ordered grid multi sampling can be replaced with some new technique. And further more this rasterization engine can be converted into a portable API by adding modules in geometry engine.

On the other hand porting this onto some architecture can also be a good idea. And also the triangle set up process is based on edge interpolation which is relatively old yet computationally very less complex. This can be replaced with triangle edge functions with barycentric co-ordinates which might be suitable for implementing on some architecture.

On the whole 3D graphics pipe-line is very demanding with new ideas and algorithms added constantly for improving rendered image quality. More realistic image quality at faster rate places much more demand on triangle and pixel processing stage. Consequently there is constant need and room for algorithmic and architectural innovation.
Bibliography

[1] André LaMothe
Tricks of the 3D game programming Gurus Advanced 3D graphics and Rasterization
ISBN- 0-672-31835-0

3D Game Engine Design a practical Approach to Real Time Computer Graphics
ISBN- 1558605932

Computer Graphics: Principles and Practice in C
ISBN - 0201848406

ISBN – 0321335732

[5] Mark Segal, Kurt Akeley
Open GL Graphics System a Specification – GL Spec 2.0

Department of computer graphics - Lund University Sweden
Mobile Graphics Hardware – Lecture Notes
http://www.cs.lth.se/DAT075/notes/mgh_ch123.pdf

University of Waterloo, School of Computer Science, Computer Graphics Lab.

[8] Daniel Mc Keon
Computer Science Department, Trinity College

Transformation of Rendering Algorithms for Hardware Implementation – PhD Thesis
Faculty of Electrical Engineering and Informatics, Budapest University of Technology and Economics, Budapest 2002.
http://www.iit.bme.hu/~szirmay/abbas.pdf

The Department of Computer Science, Sheffield University, July 1992.

http://ce.et.tudelft.nl/publicationfiles/894_10_p1-antochi.pdf

www.opengl.org

www.khronos.org.opengles

http://ogl-es.sourceforge.net/references.htm

http://studierstube.icg.tu-graz.ac.at/klimt/

[17] ExtremeTech 3D Pipeline Tutorial by Dave Salvator.
http://www.extremetech.com/article2/0,1697,464440,00.asp

[18] Software Rendering Tutorial from DevMaster.com
http://www.devmaster.net/articles.php

[19] Texture Mapping Tutorial
http://www.geocities.com/SiliconValley/2151/tmap.html
http://tfpsly.free.fr/Docs/TomHammersley/tmaptute.htm
http://en.wikipedia.org/wiki/Texture_mapping

http://www.3dcenter.org/artikel/multisampling_anti-aliasing/index_e.php

[21] Bi-linear texture filtering
http://en.wikipedia.org/wiki/Bilinear_filtering
Tulika Mitra and Tzi-cker Chiueh
Three-dimensional computer graphics architecture Computer Science Department, State University of New York at Stony Brook, Stony Brook, NY 11794-4400, USA
Appendix A Static Profiling

In order to determine the exact statistics of the rasterization engine it is essential to perform profiling. Profiling helps us to determine the key computational elements of an application program. For instance when porting a software program on to hardware it is of key importance to determine the computational blocks frequently accessed by the source code. Profiling can be done in two ways one is static profiling and other dynamic profiling. Static profiling is effective for small source codes. In case of large programs dynamic profiling is done to quickly get the statistics for each computational block. In this case we start with brief static profiling of rasterization engine to evaluate the computational statistics.

For instance the rasterization engine has two fundamental blocks one that generates fragments and the other processes the fragments. If we consider profiling the fragment generation block major computations occurring in this section are computing deltas across triangle edges and across spans. Those are change in x, z, RGBA, <u, v> with respect to y for all the three edges in the triangle and also compute change in the same set of variables except x for each scan line. Computing deltas is a division operation that is ratio of difference in variables x, z color etc and difference in y.

The following tables give a detailed description of various arithmetic operations performed in both the fragment generation and fragment processing stages. Tables in the fragment generation unit highlight operations performed per-triangle, operations performed per scan line and operations performed per-fragment. Tables in the fragment processing part discuss about operations in Texturing, fog, alpha blending and z-test units. After these tables I will assume a 3D scene with about 10000 visible triangles, with average height and width of 5 pixels and give the number of operations performed. Assuming gouraud shading, perspective texturing.

Fragment Generation Unit

Per-Triangle

<table>
<thead>
<tr>
<th>For 3 Edges</th>
<th>Add/Sub</th>
<th>Mul</th>
<th>Inverse</th>
<th>Compare</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Difference</td>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;x, y&gt;; &lt;RGBA&gt;; &lt;u, v&gt;;&lt;f&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compute Deltas</td>
<td>1/dy</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Screen</td>
<td>&lt;dx, dz&gt; x 1/dy</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Color</td>
<td>&lt;dr, dg, db, da&gt; x 1/dy</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Texture</td>
<td>&lt;du, dv&gt; x 1/dy</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fog</td>
<td>&lt;df&gt;x 1/dy</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Perspective texturing</td>
<td>1/z</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table A - 1Computations per-triangle
### Per-Scan line

<table>
<thead>
<tr>
<th>Per-Scan line</th>
<th>Add/Sub</th>
<th>Mul</th>
<th>Inv</th>
<th>Compare</th>
<th>Memory</th>
</tr>
</thead>
</table>
| Compute Difference
  \(<x, z>: \langle\text{RGBA}\rangle; <u, v>; <f>\) | 9 | | | | |
| Compute Deltas
  \(<dz> \times 1/dx\) | | | 1 | | |
| Depth
  \(<dz> \times 1/dx\) | | | | | |
| Color
  \(<dr, dg, db, da> \times 1/dx\) | | | | 4 | |
| Texture
  \(<du, dv> \times 1/dx\) | | | | 2 | |
| Fog
  \(<df> \times 1/dx\) | | | | 1 | |
| Increment Edge Data – Left & Right
  \(<dx, dz>; <dr, dg, db, da>; <du, dv>; <df>\) | | | | 18 | |

*Table A - 2 Computations per-scan line*

### Per-Fragment

<table>
<thead>
<tr>
<th>Per-Fragment</th>
<th>Add/Sub</th>
<th>Mul</th>
<th>Inv</th>
<th>Compare</th>
<th>Memory</th>
</tr>
</thead>
</table>
| Span Increment
  \(<x, z>; <r, g, b, a>; <u, v>; <f>\) | 9 | | | | |
| Fragment depth
  \(<u, v> \times 1/z\) | | | 1 | | 2 |

*Table A - 3 Computations per-fragment*

Note: Computing deltas is done by computing inverse of dx or dy once and further multiplying with each variable.

### Fragment processing

### Texturing

<table>
<thead>
<tr>
<th>Texturing</th>
<th>Add/Sub</th>
<th>Mul</th>
<th>Inv</th>
<th>Compare</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point Sampling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bi-linear Filtering</td>
<td>12</td>
<td>20</td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>3</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Tri-linear Filtering</td>
<td>28</td>
<td>48</td>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>6</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Blending</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DECAL</td>
<td>5</td>
<td></td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MODULATE</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

*Table A - 4 Texturing computations per-fragment*
### Fog

<table>
<thead>
<tr>
<th>Fog Factor f</th>
<th>Add/Sub</th>
<th>Mul</th>
<th>Div</th>
<th>Compare</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXP</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXP2</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINEAR</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td><strong>Fog Blending</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f x Ci + (1-f) x Cf</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>7</td>
</tr>
</tbody>
</table>

*Table A - 5 Fog – per-fragment*

Ci is Incoming fragment color in RGBA
Cf is fog color in RGB.

### Z – Test

<table>
<thead>
<tr>
<th>Z – Test</th>
<th>Add/Sub</th>
<th>Mul</th>
<th>Div</th>
<th>Compare</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z - test</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

*Table A - 6 Computations - Z - test*

### Alpha Blending

<table>
<thead>
<tr>
<th>Blending Factor</th>
<th>Add/Sub</th>
<th>Mul</th>
<th>Div</th>
<th>Compare</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source and Destination factor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZERO</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ONE</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>DST_COLOR</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ONE_MIN_DST_COLOR</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRC_COLOR</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ONE_MIN_SRC_COLOR</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRC_ALPHA</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ONE_MIN_SRC_ALPHA</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DST_ALPHA</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ONE_MIN_DST_ALPHA</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRC_ALPHA_SATURATE</td>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Fragment Color</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fs (RGBA) + Fd (RGBA)</td>
<td></td>
<td>4</td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Table A - 7 Computations - Alpha Blending*

As mentioned above consider a 3D scene which comprises of 10K visible triangles. Each of these triangles is sent to the rasterization engine for scan conversion, texturing and other operations. If we look at the figures in Table A - 1 we have number of
computations per-triangle. In our case of 10000 triangles the numbers of computations are:

Per 10Ktriangles:
Add = 10K x 30 = 300K;
Mul = 10K x 27 = 270K;
Inv = 10K x 06 = 60K;

Now if we consider the average height of each triangle is 5 scan lines. From Table A – 2 we have:

Per-Scan line for 10Ktriangles:
Add = 10K x 5 x 27= 1350K;
Mul = 10K x 5 x 8 = 400K;
Inv = 10K x 5 x 1 = 50K;

Similarly consider average width of triangle is 5 pixels then form Table A - 3 we have:

Per-Span for 10000 Triangles:
Add = 10K x 5 x 9 = 450K;
Mul = 10K x 5 x 2 = 100K;
Inv = 10K x 5 x 1 = 50K;

Apparently in order to generate fragments for 10000 triangles we need to perform a minimum of:

Add = 1.425M;
Mul = 820K;
Inv = 160K;

N: Number of Triangles.
h: Average height of a triangle.
w: Average width of triangles.

<table>
<thead>
<tr>
<th>Fragment Generation</th>
<th>Add</th>
<th>Mul</th>
<th>Inv</th>
</tr>
</thead>
<tbody>
<tr>
<td>Per-Triangle</td>
<td>300K</td>
<td>270K</td>
<td>60K</td>
</tr>
<tr>
<td>Per-Scan Line</td>
<td>1350K</td>
<td>400K</td>
<td>50K</td>
</tr>
<tr>
<td>Per-Fragment</td>
<td>450K</td>
<td>100K</td>
<td>50K</td>
</tr>
<tr>
<td><strong>Total Cost</strong></td>
<td><strong>2.1 M</strong></td>
<td><strong>770K</strong></td>
<td><strong>170K</strong></td>
</tr>
</tbody>
</table>

Table A - 8  Results for 10K triangles

On the other hand consider 10K triangles each generating 25 fragments on an average, in total the fragment count is 250K. From Tables A – 4 and A - 6 the number operations performed in each unit are as follows:
Texturing: Bi-linear filtering + DECAL
Add       = 250K x 20  = 5M;
Mul       = 250K x 32  = 8M;
Memory    = 250K x 4   = 1M;

<table>
<thead>
<tr>
<th>Fragment Processing</th>
<th>Add</th>
<th>Mul</th>
<th>Inv</th>
<th>Compare</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Texturing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Point Sampling</td>
<td>500K</td>
<td>500K</td>
<td>250K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bi-linear</td>
<td>3.75M</td>
<td>6M</td>
<td>1M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tri-linear</td>
<td>8.5M</td>
<td>13M</td>
<td>2M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blending</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modulate</td>
<td>1M</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decal</td>
<td>125K</td>
<td>2M</td>
<td></td>
<td>250K</td>
<td>250K</td>
</tr>
</tbody>
</table>

Table A - 9  Texturing Results for 250K fragments

Alpha Blending and Fog are not considered in this fragment processing case since not all fragments are passed through the fog unit. Alpha blending unit is not used often for all the fragments. Only a fraction of the whole fragments go through the fog and alpha blending stages. However texturing and depth comparison are generally used for majority of fragments.

The total number of operations performed in the rasterization unit right form fragment generation through fragment processing, in case of 10K triangles and 250K fragments are:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Add</th>
<th>MUL</th>
<th>Inv</th>
<th>Compare</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fragment Generation</td>
<td>2.1M</td>
<td>770K</td>
<td>170K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Texture Filtering Bi-linear</td>
<td>3.75M</td>
<td>6M</td>
<td>1M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Texture Blending Decal</td>
<td>125K</td>
<td>2M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z-test</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>250K</td>
</tr>
<tr>
<td>Total</td>
<td>6M(approx)</td>
<td>8M(approx)</td>
<td>170K</td>
<td>250K</td>
<td>1.25M</td>
</tr>
<tr>
<td>For 25 Fps</td>
<td>150M/sec</td>
<td>200M/sec</td>
<td>4.25M/sec</td>
<td>6.25M/sec</td>
<td>31.25M/sec</td>
</tr>
<tr>
<td>Operations</td>
<td>42%</td>
<td>56%</td>
<td>2%</td>
<td>2%</td>
<td></td>
</tr>
</tbody>
</table>

Table A - 10  Computational Cost for 25 frames per-second
Note: Texture Filtering as discussed in section 3.2.1 can be thought as a dot product between four weights and four texel colors.

\[
\text{Color} = W_1 \ast T_1 + W_2 \ast T_2 + W_3 \ast T_3 + W_4 \ast T_4
\]

\[
\text{Color} = W \cdot T
\]

Where \(Ws\) are weights and \(Ts\) are texel colors (RGBA).

In our example of 250K fragments number of operations performed in bi-linear filtering are, considering texel filtering as one dot-product operation total number of operations performed is equal to 250K x 4 (RGBA) dot products. In case of Tri-linear filtering we perform the same operation on eight texels thus total computations performed here amounts to 500K x 4 (RGBA) dot-product operations.

Now for rendering a scene with this data at reasonable frame rates which is \textbf{25fps} the number of arithmetic operations performed per second is sum of 6M additions, 8M multiplications, 170K inverse, and 250K compare operations which is \textbf{360M} operations per second. And the memory operations are the sum of texturing 1M and z-test 250K which is \textbf{31.25M} memory operations per second. These values are only apparent or more visible computational costs and the hidden computations are not taken into consideration. The real computational costs may very depending on implementation.
Appendix B Screen Shots

A rendered Transparent Cube

A textured Polygon
Top triangle no Anti-aliasing. Bottom Triangle Ordered grid multi sampling on edges
Textured polygon with fog
Upphovsrätt


Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances. The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for noncommercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility. According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Radha Krishna . R