An Optimized Triangle Rasterizer

by

Salem Haykal

Thesis submitted to the DigiPen Institute of Technology

at Redmond in partial fulfillment for the degree of

Masters in Computer Science

October 2008

Abstract

This paper is aimed at introducing the reader to a variety of techniques used in hardware triangle rasterization. While the rasterization process involves many additional steps (such as texturing, shading, visibility culling, anti-aliasing, shadowing…), we will be focusing on the core triangle rasterization algorithm and what can be done in this context to improve these additional sub-processes. Specifically, this paper will focus on algorithms that are suitable for current graphics hardware which is why a brief overview of the underlying hardware architecture is first presented. Triangle rasterization has been an active area of research since the early beginning of Computer Graphics. It is still an active topic and is continuously being optimized to suit the needs of current hardware technology. Our goal is to propose two new faster triangle rasterization algorithms, one suited to SISD hardware architectures and the other tailored to SIMD hardware architectures. However, unlike the many other papers and dissertations we have read, this paper is not geared towards specific systems (such as Pixel Planes 4, Pixel Planes 5, RealityEngine Graphics…). Rather, we will focus on the actual algorithms and not on the hardware specific implementation details. This implies that our proposed algorithms will work on any hardware system (consumer graphics card processors, handheld devices processors, dedicated rasterization processors…).

Acknowledgements

I am grateful for the invaluable advice and support given by my advisor, Dr. Xin Li, and by my committee members. I am especially thankful to Mr. Samir Abou Samra who has guided me during my years of study at DigiPen. Furthermore, I am grateful to Mr. Prasana Ghali who has given me precious knowledge and insight. Special thanks also go to Mr. Claude Comair for his much appreciated support and encouragement.

I am equally grateful for the unconditional support of my family and friends.

Table of Contents

1. Introduction

1.1. Background

I am a student at DigiPen Institute of Technology which is the leading institute in teaching game development. I major in Real Time Interactive Simulation. Ever since I studied the classical triangle rasterizer during my second year of undergraduate studies, I have been intrigued in developing new more efficient ways to handle this process. Therefore I chose this topic for my Master's thesis dissertation and I hope I can contribute to the ever-growing computer graphics community.

1.2. Purpose

Some programmers might wonder why reinvent the wheel when it's already been invented. Indeed, triangle rasterizers have been, since the early beginnings of computer graphics, the "wheel" behind all modern technology in this field. First, let's answer an obvious question that comes to mind "Why triangles?" I'm sure you all know that a triangle is formed by three points and that three non-collinear points are the minimum number of points required to define a plane. It is from this simple definition that a triangle is considered the base element from which any arbitrary 2D or 3D shape may be formed of. Furthermore, triangles can be rasterized fairly quickly and offer an attractive way of efficiently interpolating arbitrary attributes along the surface (colors, depth, normals, tangents, binormals…).

Now back to our original question, we reinvent the wheel for two reasons:

1- To understand how it works: imagine an F1 driver who doesn't know his "wheels"

2- To make it better: it's obvious that we tend to continuously evolve and create something new

As you might imagine, reinventing the wheel is not exactly a walk in the park. There are lots of technical details involved in triangle rasterization. In this survey, we try to cover as much of these details as possible as a prerequisite step to reach the level of understanding needed to optimize the triangle rasterizer.

1.3. Motivation

Yet another question that comes to mind is "Why Bother?" and this is a perfectly valid question. It is true that with current graphics accelerator cards, a huge number of triangles throughput (hundreds of millions per second) may be achieved. However, there are many reasons why optimizing the triangle rasterizer may prove very advantageous. First of all, imagine a device not having a graphics accelerator (such as a handheld device), displaying 3D graphics on this kind of devices requires a lot of optimization to squeeze out every bit of available processing power. Now some might claim that, with the advancements made in manufacturing computer chips, such devices will soon get their graphics accelerator and still maintain a reasonable price. It is true that some handheld devices already have graphics accelerators, and next generation cell phones technology will focus on this aspect. However, it is equally true that other devices will soon exhibit the need for displaying 3D (or at least 2D) graphics without the extra cost of having graphics accelerators (it would be nice for your home microwave to display some kind of visual representation of the food being heated rotating on a virtual platter and displaying visual cues of heat levels). The second reason is more oriented towards

real time simulations (notably games). With the advent of GPGPUs, it is possible to exploit the GPU hardware technology to perform general tasks. Why not create a really fast triangle rasterizer and use it in your simulation engine? Such a rasterizer would help in virtually every graphics task (and if you are creative in other tasks also, such as generating surface points to be used in collision engines, portal visibility determination, physics simulations…) A concrete example would be the shadow map building pass. Having highly tessellated models would hurt performance in this pass even though no heavy vertex or fragment shaders are running. So if we have 100,000 triangles render limit per frame into the shadow map, a 30% more optimized triangle rasterizer would allow us to render up to 130,000 triangles which is significantly more. So instead of lowering the models level of detail in the shadow map building phase, we can retain these details by using the optimized triangle rasterizer. Now imagine having three or four of such passes in one frame of your simulation (such as the initial depth fill pass (depth peeling), stencil fill passes, other shadow map passes for shadow casting lights or even multi-pass object rendering effects…) the gain of the optimized rasterizer scales even further.

1.4. Terms and Abbreviations

The following is a table listing the terms used in this paper with their respective definitions:

Rasterization	The process of mapping infinitesimal points into a finite set of image pixels
Scan Conversion	Another term used for Rasterization
Vertex	Triangle point
Pixel	Picture Element
Tessellation	Surface subdivision

The following is a table listing all the abbreviations used in this paper:

2D	Two Dimensional
3D	Three Dimensional
GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing Unit
SIMD	Single Instruction – Multiple Data
SISD	Single Instruction – Single Data
MISD	Multiple Instructions – Single Data
MIMD	Multiple Instructions – Multiple Data
DDA	Digital Differential Analyzer

1.5. Overview

This survey will be organized into four sections:

1- The first section will go over the mathematical background needed to understand triangle rasterization. The level of required math knowledge is not

very advanced. Therefore, readers who feel comfortable with the math may skim through this section or even skip it.

2- The second section will present current hardware architectures and the influence of their underlying technology on the triangle rasterization process. Also presented is the general 3D pipeline to fully "draw the big picture".

3- The third section will cover existing triangle rasterizer algorithms. We will particularly focus on hardware implementations, however the presented concepts also map to software implementations.

4- The fourth section will present two new approaches to triangle rasterization, each suited to a particular class of hardware. Since this paper is intended to be a survey, this section will be brief and geared only towards giving the general ideas.

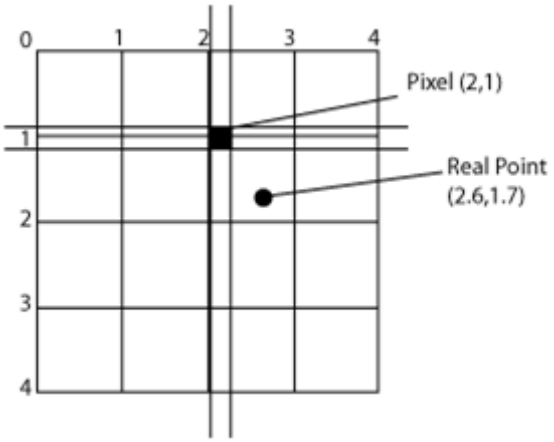
2. Mathematical Background

2.1. What Is A Pixel?

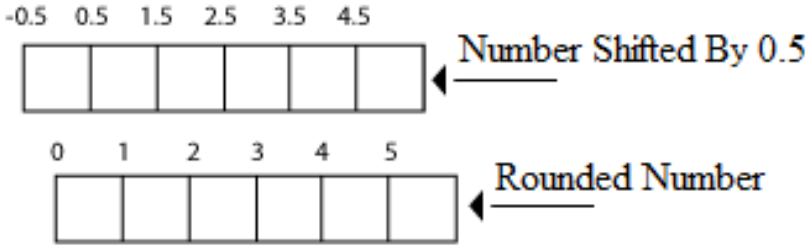
A common misconception is to think of a Pixel as a little square. Another misconception is to think of it as a little dot on the display monitor. Let's first attack the second misconception which is easier to refute. There is no fixed mapping between the monitor's dots and pixels, since most graphics cards support a variety of display resolutions (320x240, 640x480, 800x600, 1024x768…) but monitors may physically have only a fixed number of dots. Now for the first misconception, we all know that a pixel is a picture element but then what is a picture element? A picture element is a point sample and exists only as a point [1]. This leads us to define an image as an array of point samples discretized by a sampling filter. This filter's

shape determines the actual contributions to the pixel's color and that shape does not have to be a box (in fact most high quality filters are not box shaped). However, for the sake of speed, we will simplify and not use any filtering or anti-aliasing. These two rasterization simplifications are natural since our primary focus is the core triangle traversal and rasterization. All we have to do now, given these simplifications, is map real world geometric surface data into corresponding pixels. Technically, this means mapping floating-point quantities into integral quantities. Two methods exist to perform this mapping:

1- Truncation is achieved by keeping the integral part of the point coordinate i.e 1.2 -> 1 and 1.7 -> 1. The truncate operation takes the floor of the floating-point number. It is important to note that simply type casting the value to integer might not yield the desired result since the compiler may generate code to actually round the type casted number.



2- Rounding is achieved by taking the closest integral number to the point coordinate i.e 1.2 -> 1 and 1.7 -> 2. The round operation is achieved by first shifting the floating-point number by 0.5 and taking the floor of the result.



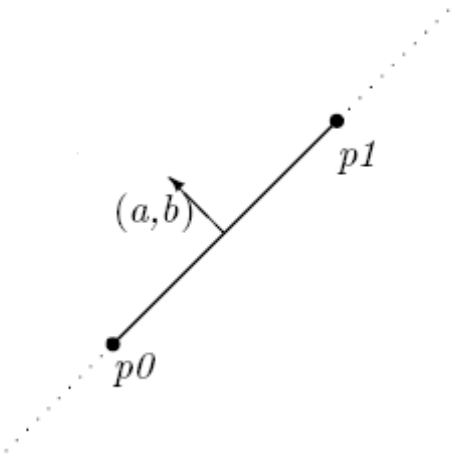
2.2. Edge Functions

An Edge function [2] is called a half-space function since it subdivides the space into two "half" regions based on the considered edge. An edge function is defined by a line in its implicit form:

E(x, y) = ax + by + c where (a, b) is the line normal and c is the line's distance from the center along its normal. The function yields three possible outputs based on the input point P(x, y):

- E(x, y) = 0 if point P is on the line
- E(x, y) > 0 if point P is on the positive line half-space which the line normal points to
- E(x, y) < 0 if point P is on the negative line half-space which the line normal points opposite to

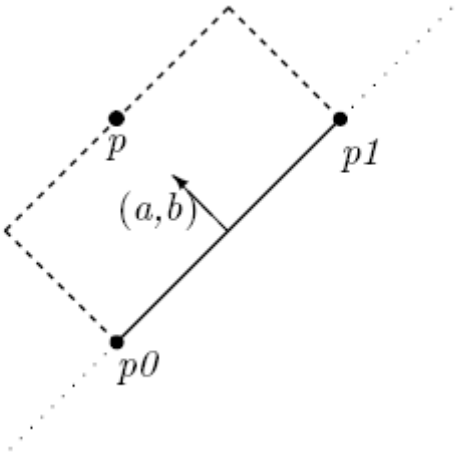
So if we consider a triangle edge formed between points P1 and P2 with normal (a, b) as in the following figure:



We can easily see that the edge function is computed by setting:

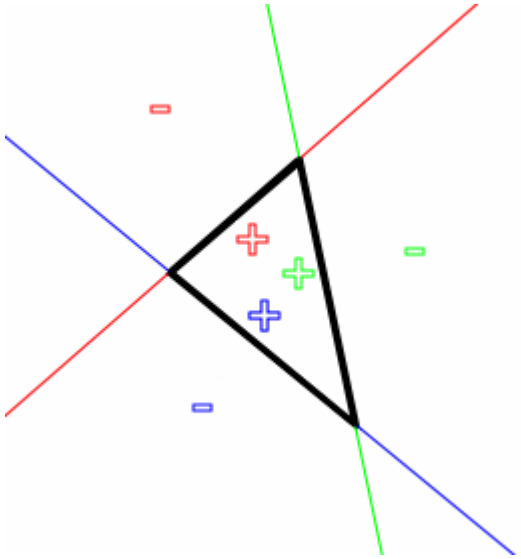
- a = - (y1 - y0)
- b = (x1 – x0)
- c = - (a * x0 + b * y0)

Once the edge function is computed, we can test any arbitrary point P as depicted in the following figure:



Now having three edges that define a triangle, we can test if a point P is inside the triangle given a consistent edge orientation. If we consider a counterclockwise edge orientation, then the point is inside if all three tests yield a positive result.

This is illustrated in the following figure:



So we now have a way to check whether the pixel is inside the rasterized triangle which will come in handy when we look at the tile-based triangle rasterizer (more on this later).

2.3. Digital Differential Analyzer (DDA)

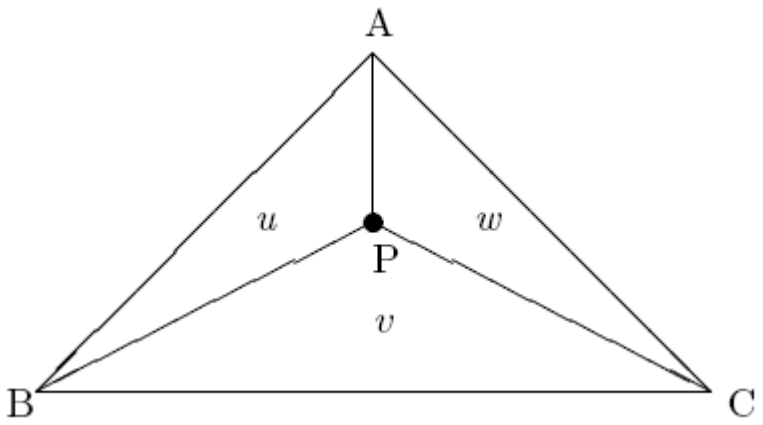
A common technique to minimize heavy arithmetic operations is the DDA method. For example, the "Edge Walking" based triangle rasterizers that we are going to see later in section 3 rely on DDA to incrementally update the position along the triangle edges. Let's consider an explicit line equation $\mathbf{y_i = m * x_i + b}$ where **i** denotes the current iteration. Assuming we want the value when y is incremented by 1 (which is the case in the "Edge Walking" algorithms), then we need $\mathbf{x_{i+1}}$. Let's replace in the explicit line equation $\mathbf{y_i + 1 = m * x_{i+1} + b} \Leftrightarrow \mathbf{x_{i+1} = (y_i - b) / m + 1 / m}$ but $\mathbf{x_i = (y_i - b) / m} \Leftrightarrow \mathbf{x_{i+1} = x_i + 1 / m}$. This last equation means that given the current value of **x** we can get the next value at **y + 1** by simply adding the inverse slope of the line segment. So instead of performing a multiplication and an addition (evaluating the line equation), we only have to perform an addition. This might not seem like a lot of optimization, but imagine having a huge number of triangles rendered at a high enough resolution. The number of saved multiplications quickly builds up and becomes significant.

Taking this concept a bit further, it can easily be shown that we can obtain any value of **x** offset by $\Delta\mathbf{y}$ from a given reference value. So we obtain $\mathbf{x_{i+\Delta y} = x + \Delta y / m.}$ This simple equation will prove very useful in the final section where we propose a new approach for rasterizing triangles. It is important to note that DDA is not only used to incrementally update spatial coordinates, but also any kind of attribute as we will

see in the next section. Also note that DDA may also work with integer arithmetic where an accumulator is used to track the floating point digits.

2.4. Barycentric Coordinates

Barycentric coordinates are actually used to find the center of mass for a geometric object [3]. We are interested in a subset of Barycentric coordinates called homogenous barycentric which are normalized such that they become the areas of the sub-triangles [4] as seen in the following figure where u, v, w are the barycentric coordinates of point P in the triangle ABC.



These coordinates are normalized by the entire triangle area. A consequence of this normalization is that any point inside the triangle has barycentric coordinates bounded between zero and one and that the sum of these coordinates must be equal to one (if they were not normalized it would be equal to the actual triangle area). So we have the equation:

- u + v + w = 1 for P inside the triangle

Areal coordinates [5] are another name for homogenous barycentric coordinates which verify the above equation.

The importance of areal coordinates in triangle rasterization lies in the fact that they provide a consistent way of interpolating vertex attributes independent of the triangle orientation. Another possible use of barycentric coordinates is that, similar to edge functions, they provide a way to check if a point P is inside the triangle (if all homogenous coordinates are within the range [0,1]). We will be using barycentric coordinates to interpolate vertex attributes. Assuming we are interpolating an arbitrary attribute S specified at each vertex S_A, S_B, S_C the value of S at point P would be:

- $S_P = u * S_A + v * S_B + w * S_C$

All we need to do now is to actually find u, v and w for point P. This can easily be done by realizing that the area of the triangle is the half cross product of two consecutive triangle edges:

- Area(ABC) = |AB ^ AC| / 2

And by definition we have:

- u = Area(ABP) / Area(ABC)
- v = Area(BCP) / Area(ABC)
- w = Area(CAP) / Area(ABC) or also w = 1 – u – v

So the result:

- Let $d = (x_B - x_A) * (y_C - y_A) - (x_C - x_A) * (y_B - y_A)$
- $u = ((x_B – x_A) * (y_P – y_A) - (x_P – x_A) * (y_B – y_A)) / d$
- $v = ((x_C – x_B) * (y_P – y_B) - (x_P – x_B) * (y_C – y_B)) / d$
- $w = ((x_A – x_C) * (y_P – y_C) - (x_P – x_C) * (y_A – y_C)) / d$

Computing these coordinates per pixel is very costly. Therefore, we need a way to incrementally update them along every xStep and yStep. We do that by first noticing that only x_P and y_P vary along the triangle surface. That means that the value of "d" is constant, and since division is expensive we can precompute 1/d. Another consequence is that for every xStep:

- The value of u is updated by xStep * $(y_B - y_A)$
- The value of v is updated by xStep * $(y_C - y_B)$
- The value of w is updated by xStep * $(y_A - y_C)$

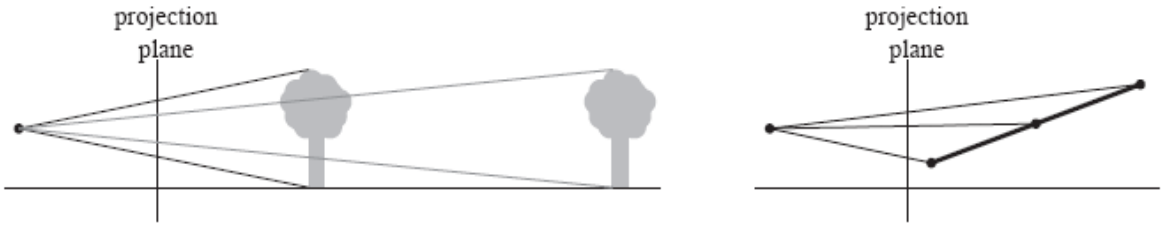
Similarly, for every yStep:

- The value of u is updated by yStep * $(x_B - x_A)$
- The value of v is updated by yStep * $(x_C - x_B)$
- The value of w is updated by yStep * $(x_A - x_C)$

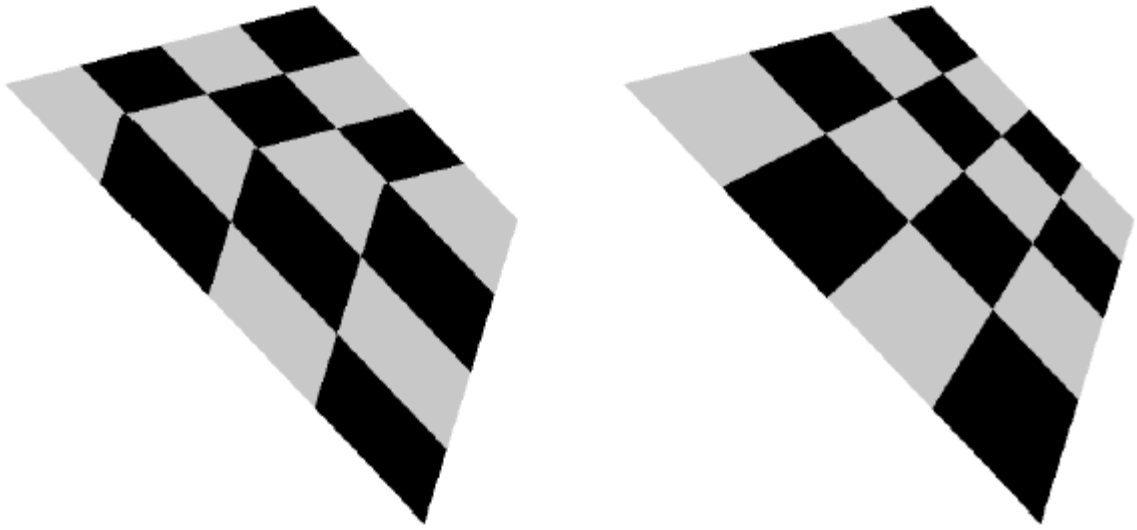
It is clear that by using this DDA method we can greatly optimize the computation of our areal coordinates.

2.5. Perspective Correction

Whenever perspective projection is used, we need to perform perspective correction for the interpolated attributes to account for the perspective foreshortening effect.



acceptable for some attributes (like color) but may lead to severe artifacts in some other attributes (like texture coordinates) as shown in the following figure:



In order to remedy this problem, we need to understand its source. Perspective projection is not an affine transformation since it involves a division by z (perspective divide). That means that screen space quantities are linear with respect to 1/z. As a consequence, we need to interpolate the barycentric coordinates along 1/z. We do that by first dividing the coordinates by the respective triangle vertices z (projection transform), and interpolate using the derived formula in the previous section. But we now need to recover the actual areal coordinates u, v and w. It becomes clear that we need to interpolate 1/z per pixel:

- $1/z = u / z_A + v / z_B + w / z_C$

All we need to do now is to divide by 1/z which is really multiplying by z, the perspective correct interpolated quantity S is then:

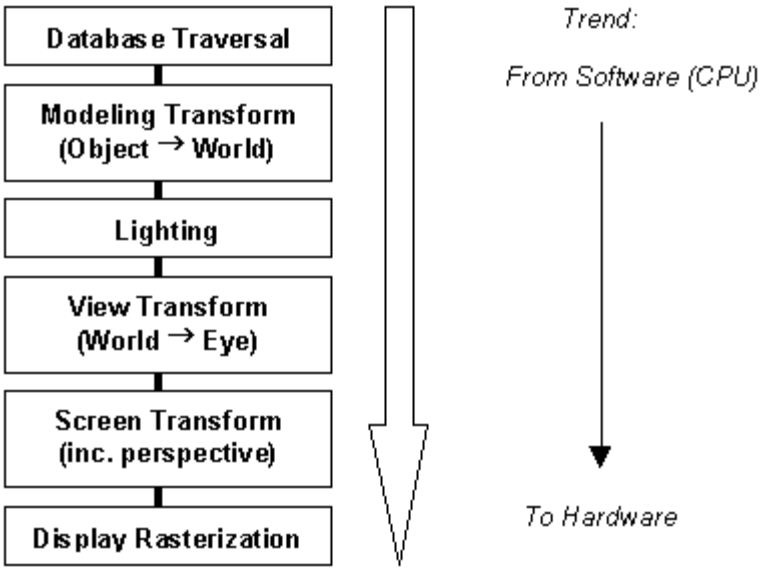
- $S_p = ((u * S_A / z_A) + (v * S_B / z_B) + (w * S_C / z_C)) / (1/z)$

Some methods have been developed to help reduce the cost of division by 1/z such as using table lookups [6].

3. Underlying Technology

3.1. 3D Graphics Pipeline

A view of the classical 3D graphics pipeline is presented in the following figure:



More and more stages of this pipeline have found their way into special purpose hardware over the past few years. This is a big evolution from the early days of graphics where the display was memory mapped and the general purpose CPU wrote pixels directly into the frame buffer. The hardware implementations have started from the Display Rasterization stage and gradually risen up to include all the pipeline stages except for the Application stage where the CPU feeds data into the graphics processor. Even more technological advancement was made with the development of micro-programmable processors. Developers can now write their own vertex and fragment programs (even geometry programs recently with DX10). Currently, General Purpose Graphics Processing Units (or GPGPUs) are being developed and promise a blend of efficiency and full programmability. However, all that is only for devices that support a heavy amount of power usage, big enough

hardware integration space, powerful cooling systems and cost efficiency. But since not all devices are created equally, we sometimes fall back to older technologies in some cases such as handhelds.

3.2. Hardware Architectures

The most popular nomenclature for the classification of hardware architectures is the one proposed by Flynn[7] in 1966. We are interested in Flynn's classification since he chose not to examine the explicit structure of the hardware, but rather how instructions and data flow through it. Flynn's taxonomy identifies whether there are single or multiple streams for data and for instructions in the hardware architectures. The term "stream" refers to a sequence of either data or instructions.

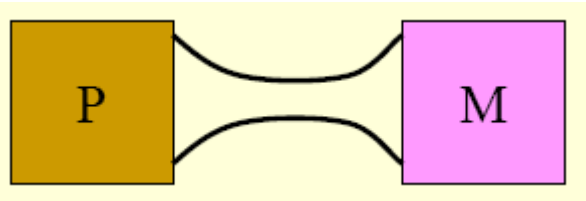
Flynn identifies four categories of hardware architectures:

- SISD Single Instruction Single Data
- SIMD Single Instruction Multiple Data
- MISD Multiple Instruction Single Data
- MIMD Multiple Instruction Multiple Data

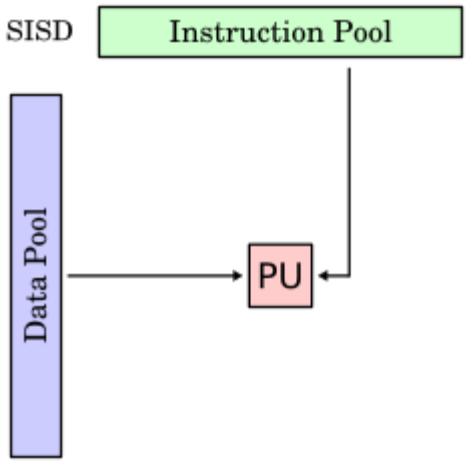
A brief study of each of these architectures is presented.

3.2.1. SISD

SISD based processors are conventional serial architectures that can process only one stream of instructions on one stream of data. SISD architectures are also called Von Neumann models since they suffer from the Von Neumann bottleneck which is essentially the bottleneck between processors and memory as in the following figure:



The architecture is illustrated in the following figure:



In this kind of architectures, instructions may be overlapped by pipelining. Additional functional units may be provided such as arithmetic coprocessors, vector units, I/O units…

Some examples of SISD machines:

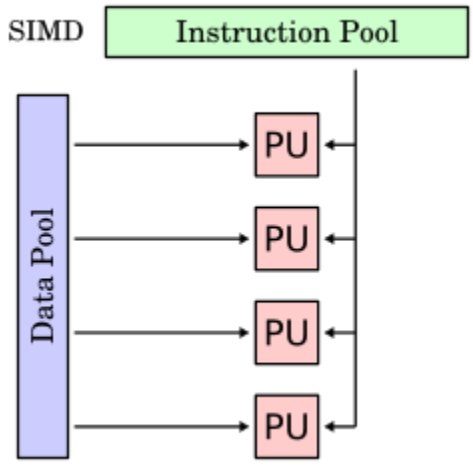
- CDC 6600 which is not pipelined but has multiple functional units, may be considered as the first supercomputer.
- CDC 7600 which has a pipelined arithmetic unit.
- Amdhal 470/6 which has pipelined instruction processing.
- Cray-1 which supports vector processing.

3.2.2. SIMD

In this kind of architectures, a single control processor acts as a supervisor for multiple identical inter-connected processors executing the same program with different data inputs. These inter-connected processors are said to operate in “lock-step”. Note that each processor has its own memory from

which it works on its own data, which means that multiple processors have different data streams.

The architecture is illustrated in the following figure:



Each processor completes its instruction before the next instruction is loaded leading to a synchronous mode of operation.

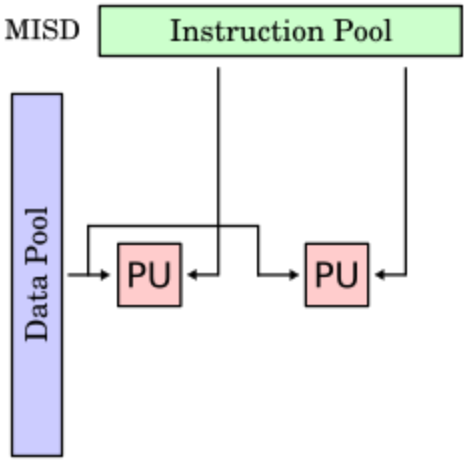
Some examples of SIMD machines:

- Illiac - IV
- BSP
- STARAN
- MPP
- DAP
- CM-1 and CM-2

3.2.3. MISD

MISD architectures have multiple processing elements each executing a different stream of instructions but on the same set of data.

The architecture is illustrated in the following figure:

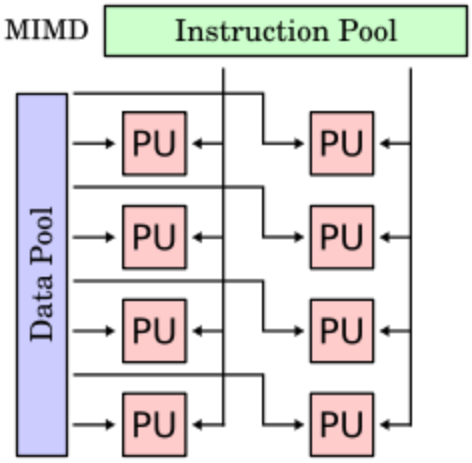


This model is not very useful or at least no useful way of using this model has been discovered yet. It has only been implemented in C.mmp which is built by Carnegie-Mellon University. This computer is reconfigurable and can operate in SIMD, MISD and MIMD modes.

3.2.4. MIMD

MIMD architectures have multiple processing units each executing a different stream of instructions on a different stream of data. The instructions executed by different processors may start and finish at different times so no lock-stepping is used as in SIMD architectures. This leads to an asynchronous mode of operation.

The architecture is illustrated in the following figure:



MIMD architectures are common in today's most powerful supercomputers. The only problem with this model is the synchronization overhead between different processors.

Some examples of MIMD machines:

- C.mmp
- Tandem/16
- S1
- Cray-2
- Cray X-MP
- Burroughs D825
- BBN Butterfly
- FPS T/40000
- iPSC
- HEP

3.3. Target Architectures

The SISD architecture is still being used in today's low-end devices such as handhelds for reasons of power saving and cost-efficiency. We will propose an optimized triangle rasterizer specific to this kind of architectures where triangle traversal speed is essential and parallel pixel processing is not needed.

Furthermore, the SIMD architecture is currently dominating consumer level graphics cards (mainly because of being more cost efficient than MIMD), we will also propose another optimized triangle rasterizer that exploits this architecture's

capabilities and allows multiple pixels to be handled in parallel based on the number of available stream processors.

4. Existing Triangle Rasterizers

4.1. Overview

Triangle rasterizers can be broadly categorized into two approaches:

- Edge Walking
- Edge Function Testing

The first approach "walks" on the triangle edges to find successive horizontal spans to fill, whereas the second approach relies on edge functions (see first section) to determine whether a point is inside the triangle and needs rasterization. At first glance, the edge walking algorithm seems much more efficient than the second more "brute force" approach. However, due to hardware efficiency issues I will cover in this section, the second approach is currently being used in contemporary graphics cards since it allows for "tile-based" rasterization where a tile is a block of m*n (or usually n*n) pixels.

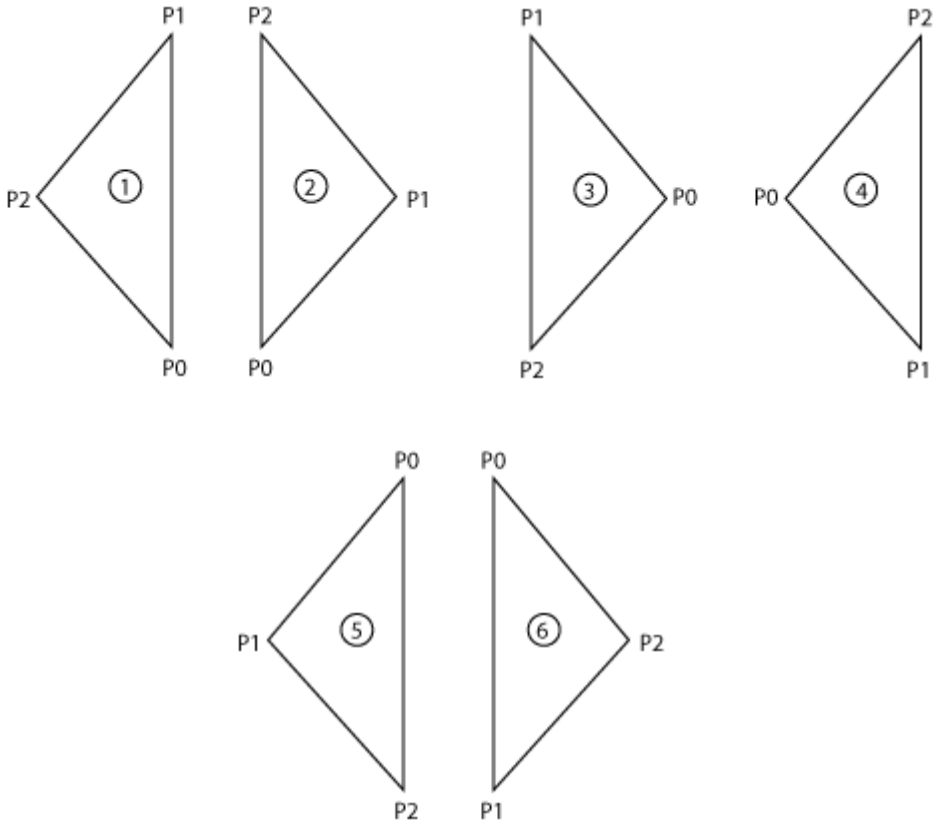
4.2. Edge Walking

Sometimes referred to as the classical triangle rasterizer, it offers very efficient triangle traversal which is why it is still being used in today's low end consumer devices such as handhelds. Its only downside is that it doesn't allow finding multiple pixels at the same time, therefore no parallel pixel processing may be executed which is why its main use is in SISD architectures.

In this section, we will outline the entire edge walking algorithm subdivided into its various steps [11].

4.2.1. Vertex Sorting

Our goal is to identify the top, middle and bottom vertices. Also we need to determine whether the middle vertex is to the left or to the right. If the triangle vertices are assumed to be in a counter-clockwise order, there are only six possible vertex configurations shown in the following figures:



By laying out these cases, it is possible to determine the input vertex configuration by performing two checks on the y-values of the triangle vertices. The following pseudo code illustrates these checks:

```
If (P0y < P1y)
    If (P2y < P0y)
        Top = 2
        Middle = 0
        Bottom = 1
        MiddleIsLeft = 1
    Else
        Top = 0
        If (P1y < P2y)
            Middle = 1
            Bottom = 2
            MiddleIsLeft = 1
        Else
            Middle = 2
            Bottom = 1
            MiddleIsLeft = 0
Else
    If (P2y < P1y)
        Top = 2
        Middle = 1
        Bottom = 0
        MiddleIsLeft = 0
```

Else

Top = 1

If (P0y < P2y)

Middle = 0

Bottom = 2

MiddleIsLeft = 0

Else

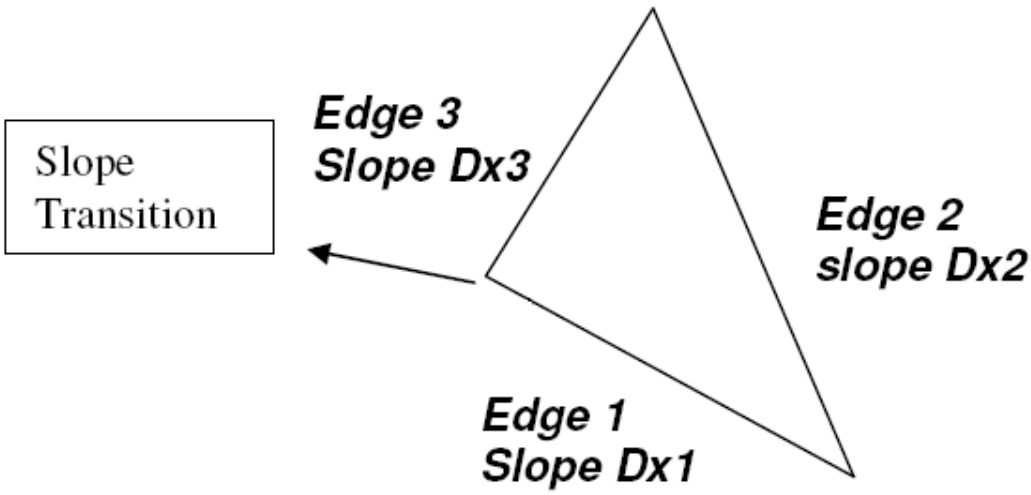
Middle = 2

Bottom = 0

MiddleIsLeft = 1

4.2.2. Top-Bottom Split

The triangle must be rasterized from top to bottom, it is split into two along the horizontal line passing by the middle vertex since a slope transition is needed between these two parts as shown in the following figure:



4.2.3. Update By Inverse Slopes

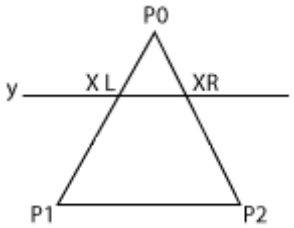
Using the DDA method, at each downward y step we update the x value by the inverse slope of the respective edge. The slopes are easily computed by setting:

InverseSlopeEdge0 = $(P_{Bottom}x - P_{Top}x) / (P_{Bottom}y - P_{Top}y)$

InverseSlopeEdge1 = $(P_{Middle}x - P_{Top}x) / (P_{Middle}y - P_{Top}y)$

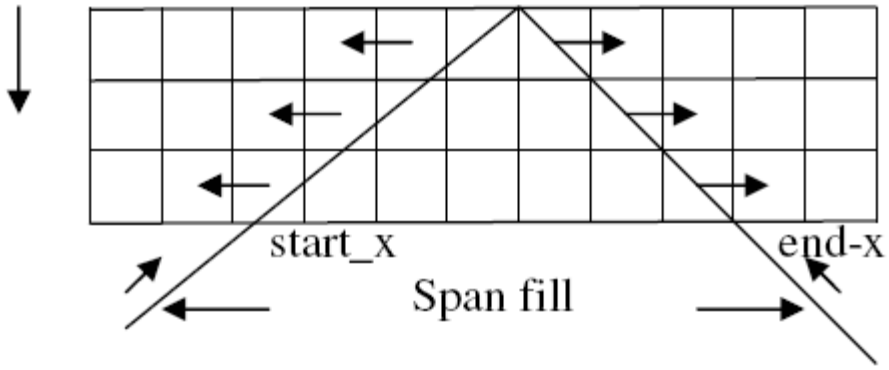
InverseSlopeEdge2 = $(P_{Bottom}x - P_{Middle}x) / (P_{Bottom}y - P_{Middle}y)$

Note that at any given time, we only have two active edges, one left and the other right as in the following figure where xL is current left x and xR is current right x.



4.2.4. Fill Spans

At every y step we have a left x value and a right x value. All we need to do is fill the horizontal span given by these two values as shown in the following figure:



Note that any number of attributes may be interpolated (see section 1) and determined per pixel so that they may be used in finding the final rasterized pixel color.

4.3. Edge Function Testing

Currently implemented in high end consumer graphics cards, or more precisely the tile based variant of this algorithm [12] is the one that's actually implemented since the raw algorithm is simply considered as a brute force method of finding pixels inside in the triangle. In this section, we will outline the basic edge function testing algorithm then present a highly optimized tile based version of it.

4.3.1. Basic Algorithm

The first step is to determine the triangle edges testing functions assuming we have a triangle formed by three points P0(x0, y0), P1(x1, y1) and P2(x2, y2):

- Edge 1: (x1 - x2) * (y - y1) - (y1 - y2) * (x - x1)
- Edge 2: (x2 - x3) * (y - y2) - (y2 - y3) * (x - x2)
- Edge 3: (x3 - x1) * (y - y3) - (y3 - y1) * (x - x3)

Next we need to determine which pixels actually need to be tested. Surely it would not be efficient to test all the screen pixels against these edge functions per triangle. The answer is to simply use the triangle bounding box computed as such:

- MinX = min(x1, x2, x3)
- MinY = min(y1, y2, y3)
- MaxX = max(x1, x2, x3)

- MaxY = max(y1, y2, y3)

So we now have the structure of a working triangle rasterizer as shown in the following pseudo-code:

Loop Vertical Extent MinY to MaxY

Loop Horizontal Extent MinX to MaxX

If EdgeFunction1(CurrentX, CurrentY) >= 0 &&

EdgeFunction2(CurrentX, CurrentY) >= 0 &&

EdgeFunction3(CurrentX, CurrentY) >= 0

SetPixel(CurrentX, CurrentY)

However the above rasterizer is not robust at all, it does not account for the Top-Left Fill rule nor does it account for the Shared Vertex issue (more on these two topics in the next section). Aside from the robustness issues which will be addressed later, this algorithm is not efficient since each per-pixel edge function test requires two multiplications and five subtractions making for a total of six multiplications and fifteen subtractions for the three edge function tests. The solution is to use the DDA method. It turns out that the update per horizontal and vertical step is exactly the same as the one for barycentric coordinates (see section 2), for convenience the results are summarized by the following:

For every xStep:

- The value of EdgeFunction1 is updated by xStep * (y2 – y1)
- The value of EdgeFunction2 is updated by xStep * (y3 – y2)
- The value of EdgeFunction3 is updated by xStep * (y1 – y3)

For every yStep:

- The value of EdgeFunction1 is updated by yStep * (x2 – x1)
- The value of EdgeFunction2 is updated by yStep * (x3 – x2)
- The value of EdgeFunction3 is updated by yStep * (x1 – x3)

It is clear that by using this DDA method we can greatly optimize the computation of our areal coordinates.

4.3.2. Tile-Based Rasterizer

Rasterizing tiles instead of pixels is the main goal behind using the edge function testing method. The reason for this is that current SIMD hardware handles tiles very efficiently (a single instruction is executed on multiple pixels inside the tiles). The Tile's size currently varies from 2*2 to 8*8 depending on the number of stream processors available in hardware. Furthermore, visibility detection efficiency, cache hit rates and texture throughput are increased by using tiles [16]. All we need to do is determine whether a tile is fully inside, fully outside or partially intersecting with the triangle. A simple way would be to check the tile's extremity points against the edge functions as in the following pseudo-code:

Loop Vertical Extent MinY to MaxY Inc by Tile Size

Loop Horizontal Extent MinX to MaxX Inc by Tile Size

MinTileX = CurrentX

MinTileY = CurrentY

MaxTileX = CurrentX + TileSize

MaxTileX = CurrentY + TileSize

If TileInside(MinTileX, MinTileY, MaxTileX, MaxTileY)

Set All Tile Pixels

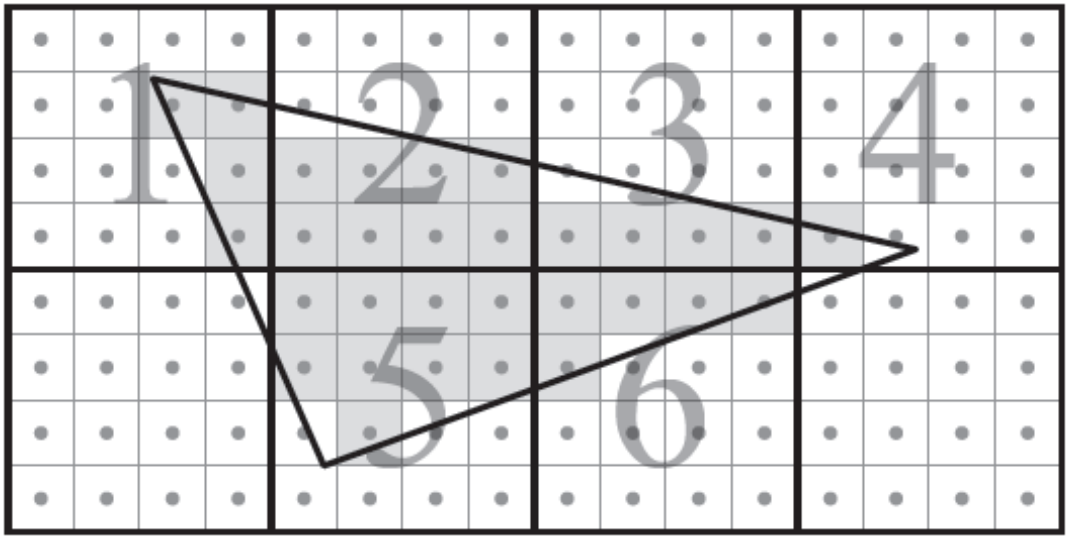
Else If TileOutside(MinTileX, MinTileY, MaxTileX, MaxTileY)

Skip Tile

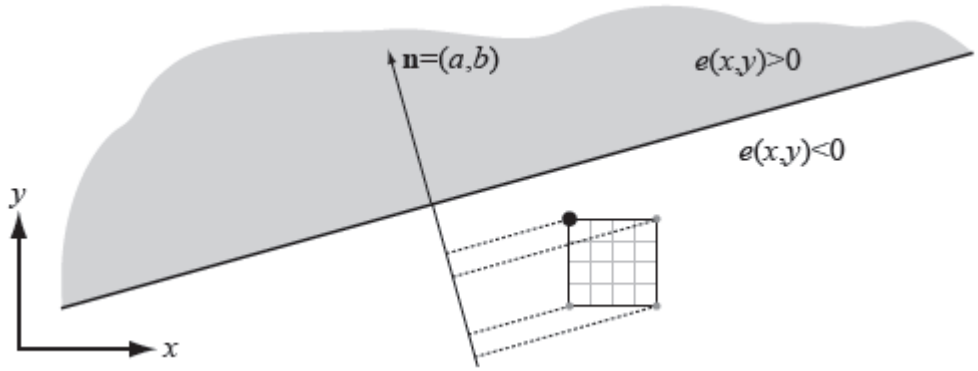
Else Tile Partially Covered

Test For and Set Tile Covered Pixels

The following figure shows a numbered tile traversal of a rasterized triangle:



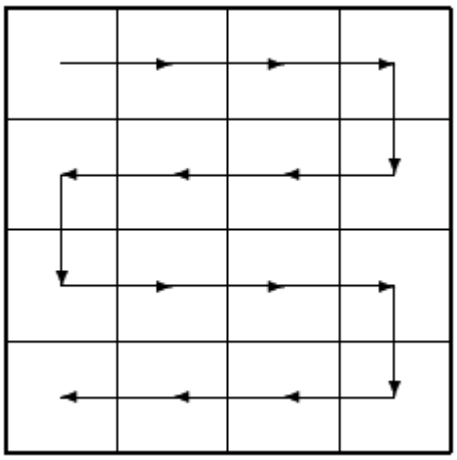
However this algorithm may be greatly optimized since it turns out that only one tile edge needs to be tested against one respective edge function instead of four [15]. Indeed, consider the following figure [16]:



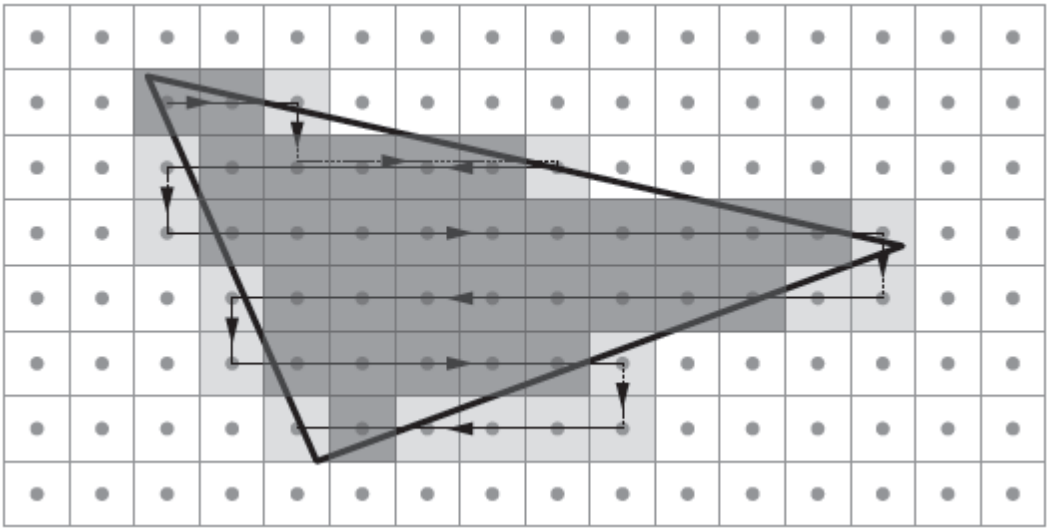
By projecting the tile's corners onto the normal of the tested edge, we can determine that, in this case, testing the top left corner is sufficient to indicate whether the tile is either fully outside or fully or partially inside the triangle. However, for efficiency, we effectively do not perform any projection, instead we pre-calculate an offset from the bottom left tile corner to the corner that needs to be tested in the triangle setup phase. The tile offsets T(w, h) are computed based on the edge normal N(x, y) where "w" is the tile width and "h" is tile's height:

- Tx = w if Nx >= 0
- Tx = 0 if Nx < 0
- Ty = h if Ny >= 0
- Ty = 0 if Ny < 0

These offsets need to be calculate during triangle setup for every triangle edge so we would have T1(w, h), T2(w, h) and T3(w, h). Furthermore, we can calculate corresponding edge offsets for both tile full rejection and full acceptance checks (reverse normal) to optimize both during tile testing. Another possible major optimization is to change the triangle traversal order. Instead of fully traversing the entire triangle bounding box, we can adapt a zigzag traversal scheme as proposed by [2] and later described in more detail by [17].



One scan-line is traversed at a time and the traversal order is altered every scan-line. The order is flipped whether a fully not covered tile is found as shown in the following figure:



This traversal scheme visits less tiles but does not necessarily fully eliminate redundancy as can be seen in the last rasterized scan-line in the above figure.

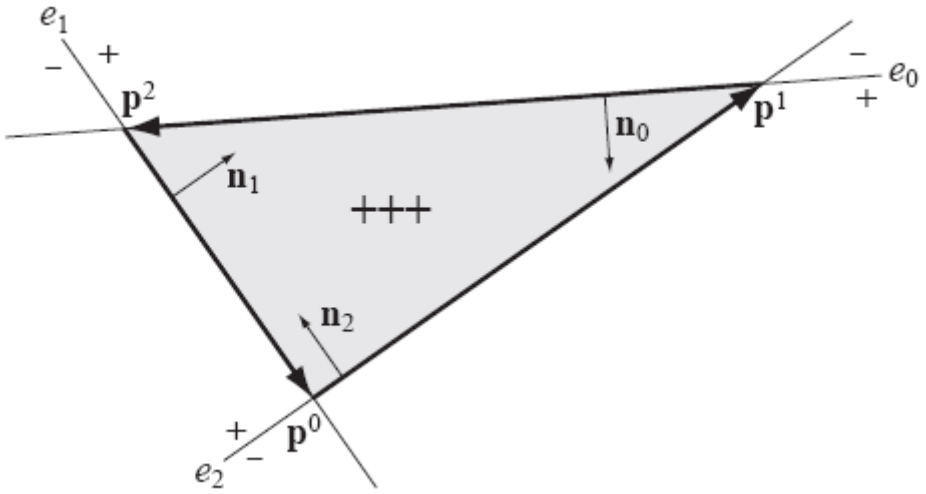
4.4. Achieving Robustness

We have deferred the discussion of a number of important issues to this section. These issues allow for more robust and consistent triangle rasterization.

4.4.1. Top-Left Fill Rule

Assume we have two triangles sharing the same edge. The shared edge will be drawn twice as each triangle is rasterized. This is both inefficient and inconsistent since it causes problems with alpha blending and stenciling operations. A simple solution to this is to only draw the top-left edges and leave the bottom right edges. That way, all connected triangle strip edges will be drawn consistently. Applying this rule in the Edge Walking triangle

rasterizer is fairly straightforward (simply subtract 1 from the right-bottom edge horizontal and vertical spans). It is however not as straightforward in the Edge Function Testing triangle rasterizer. The following figure illustrates the directed edges of a triangle (all triangles must have consistent orientation) [16]:



Let every edge have its corresponding normal N(a, b). McCool's tie-breaking rule follows [18]:

```
bool INSIDE(e, x, y)
    if e(x, y) > 0 return true;
    if e(x, y) < 0 return false;
    if a > 0 return true;
    if a < 0 return false;
    if b > 0 return true;
    return false;
```

This check relies on excluding points to the right of the triangle (a<0) and to its bottom (b<0) based on the edge normal. However observe that we can improve performance by pre-computing the constants check results as suggested by Owens [19].

Let Boolean EdgeCheck be:

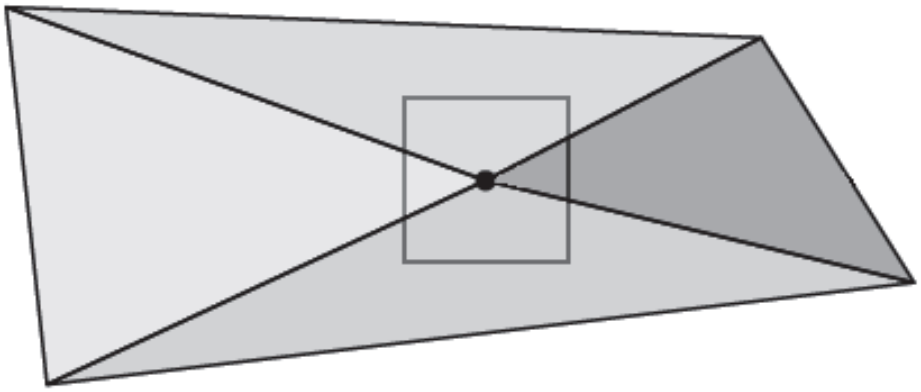
- EdgeCheck = bool(a > 0) if a != 0
- Else EdgeCheck = bool(b > 0)

The Inside test is then reduced to:

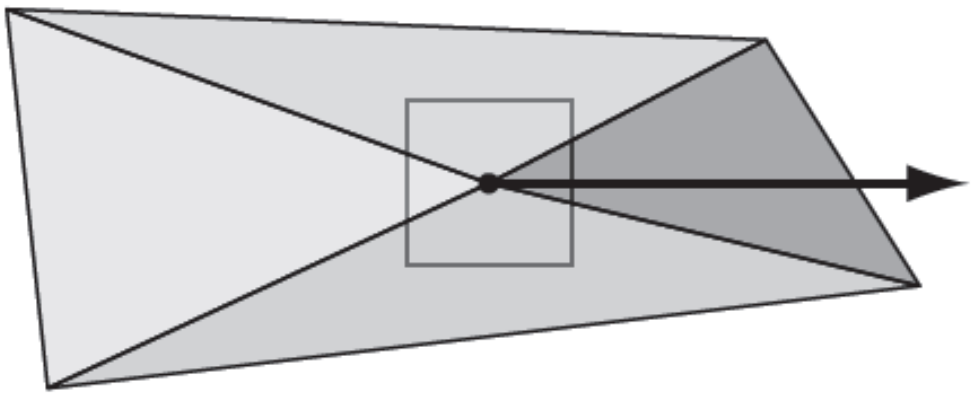
```
bool InsideOptimized(e, t, x, y)
    if e(x, y) > 0 or (e(x, y) == 0 and t) return true;
    return false;
```

4.4.2. Shared Vertex

The same problem as edge sharing arises when two or more triangles share the same vertex as shown in the following figure:



The solution is to choose an inclusion direction, any direction will work but we must be consistent. Then only the triangle having the inclusion direction totally inside will be the one that owns the shared vertex as in the following figure:



However, as the triangle fan rotates, vertex ownership will shift between triangles.

Another more aggressive solution would be to have a screen sized bit array which denotes the rasterized triangle vertices. A triangle vertex is only rasterized if its corresponding position in the bit array is zero then this position is set to 1. That way the triangles rasterization order determines the vertices to be rasterized.

4.4.3. Fixed Point Arithmetic

A fixed point number is an integral number representing a floating-point quantity. The number is said to be in i.f form [20] where "i" is the number of bits used to represent the integer part and "f" is the number of bits used to represent the fractional part. The following table sets the relation between "f" and the fixed point number resolution [12]:

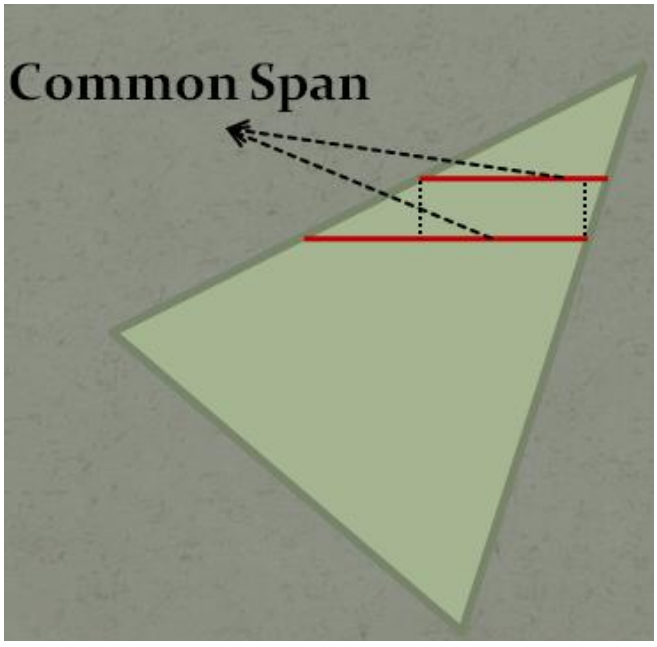
Bits	Resolution	Resolution (numeric)
1	$\frac{1}{2}$	0.50000000
2	$\frac{1}{4}$	0.25000000
3	$\frac{1}{8}$	0.12500000
4	$\frac{1}{16}$	0.06250000
5	$\frac{1}{32}$	0.03125000
6	$\frac{1}{64}$	0.01562500
7	$\frac{1}{128}$	0.00781250
8	$\frac{1}{256}$	0.00390625
9	$\frac{1}{512}$	0.00195313
10	$\frac{1}{1024}$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

Fixed point arithmetic is used in low end devices to reduce the implementation cost of accelerated floating-point instructions. Furthermore, using fixed point in some rasterizer operations such as edge function testing proves to be necessary to avoid holes or cracks in the final rendered mesh. The reason for this is that edge functions involve multiplications and subtractions which build up cumulative errors in the floating-point representation. Fixed point arithmetic, on the other hand, does not suffer much from this kind of error build-up.

5. Proposed Triangle Rasterizers

5.1. Idea

The main idea behind both proposed triangle rasterizers is to exploit successive span coherence as in the following figure:



In fact, two successive spans only vary by the sum of the absolute value of the inverse slopes of their two active edges.

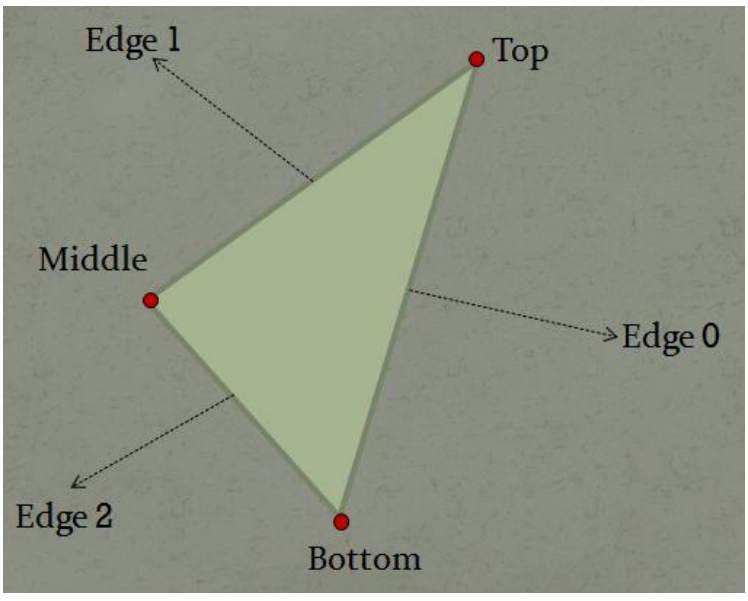
5.2. The Eight Cases

We have observed that triangles may be classified into eight cases (four by four symmetric based on middle vertex classification) according to their horizontal span coherence. Our classification is based on the DeltaX value of each triangle edge. Since there are two states per edge (<0 or >=0) there are 2^3 potential combinations per each middle vertex classification (either middle to the left or to the right). But since we are assuming CCW triangle orientation, these cases are split into half leaving us with a total of 4 cases when the middle is on the left and 4 other cases when the middle is on the right.

This section will outline the proposed classification:

5.2.1. Edge Indexing

Assuming the triangle's vertex ordering is determined as explained in the Edge Walking algorithm setup phase, the edge indexing we will follow is shown in the following figure:

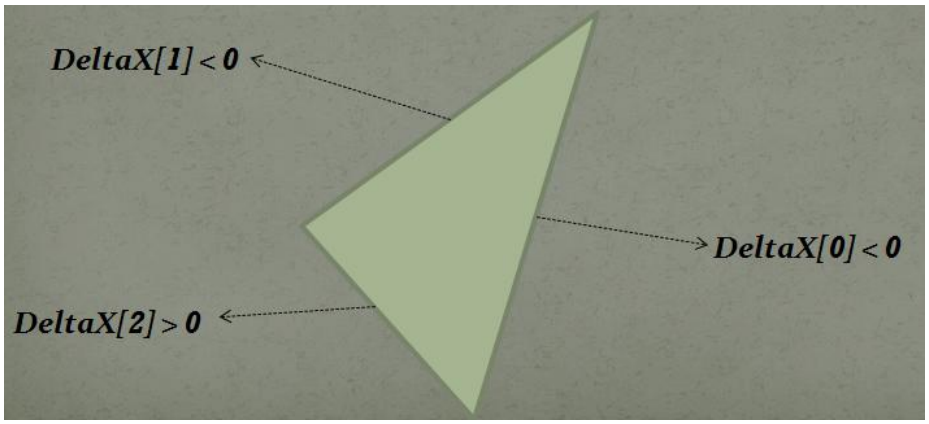


5.2.2. Middle On The Left

Four cases occur when the middle vertex is on the left:

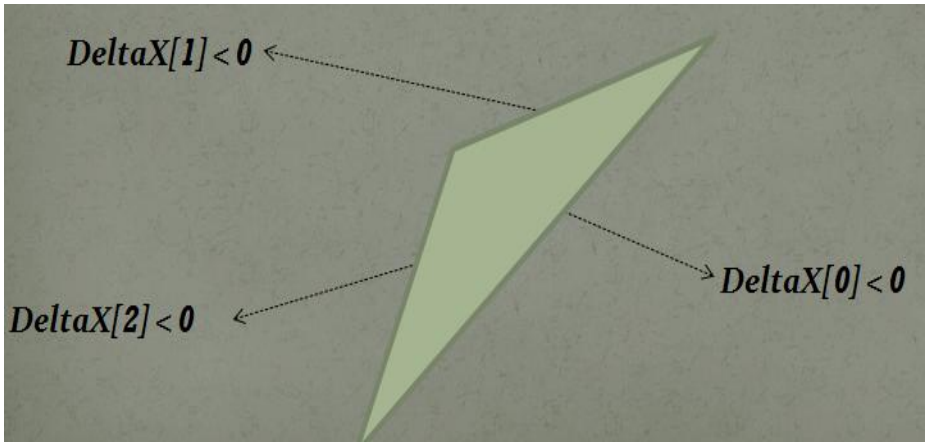
- **Case 1:**
    - DeltaX[0] < 0
    - DeltaX[1] < 0 (Implied because middle is on the left)
    - DeltaX[2] > 0

The case is shown in the following figure:



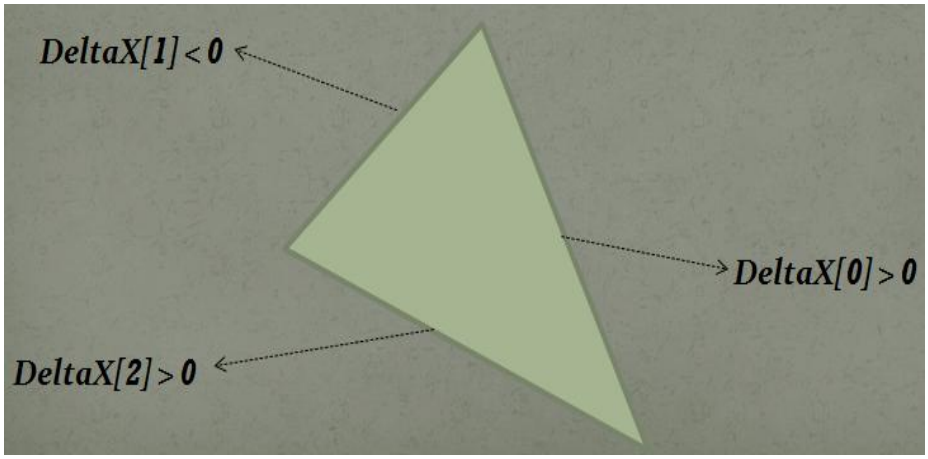
- **Case 2:**
    - DeltaX[0] < 0
    - DeltaX[1] < 0 (Implied because middle is on the left)
    - DeltaX[2] < 0

The case is shown in the following figure:



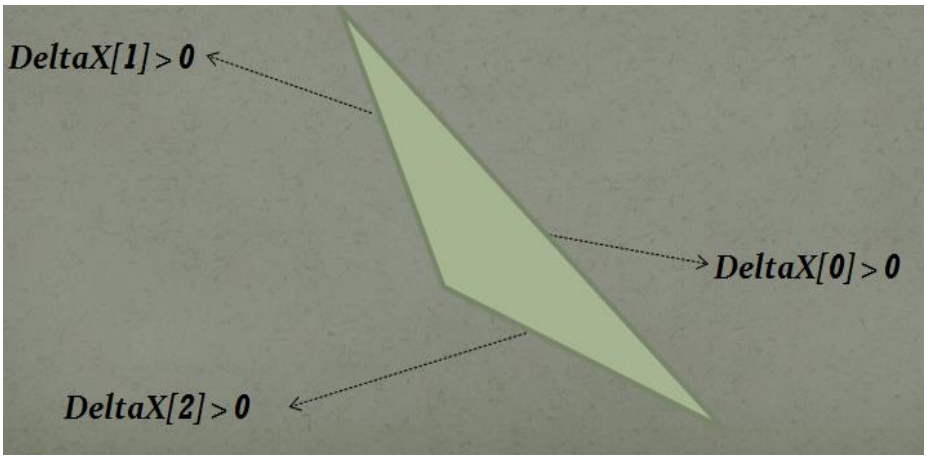
- **Case 3:**
    - DeltaX[0] > 0
    - DeltaX[1] < 0
    - DeltaX[2] > 0 (Implied because middle is on the left)

The case is shown in the following figure:



- **Case 4:**
    - DeltaX[0] > 0
    - DeltaX[1] > 0
    - DeltaX[2] > 0 (Implied because middle is on the left)

The case is shown in the following figure:

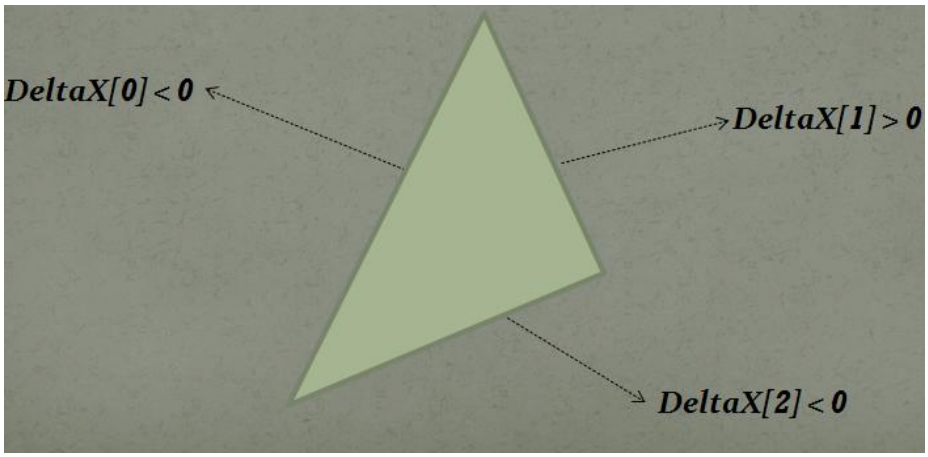


5.2.3. Middle On The Right

Four cases occur when the middle vertex is on the right:

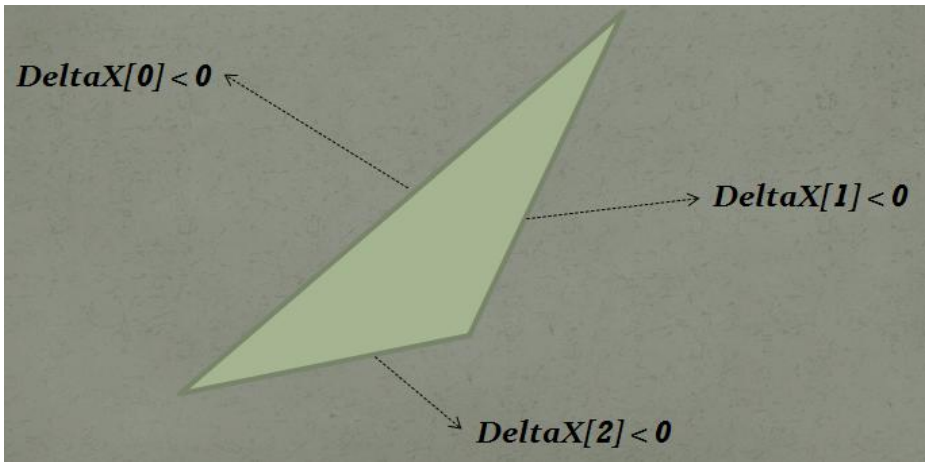
- **Case 5:**
    - DeltaX[0] < 0
    - DeltaX[1] > 0
    - DeltaX[2] < 0 (Implied because middle is on the right)

The case is shown in the following figure:



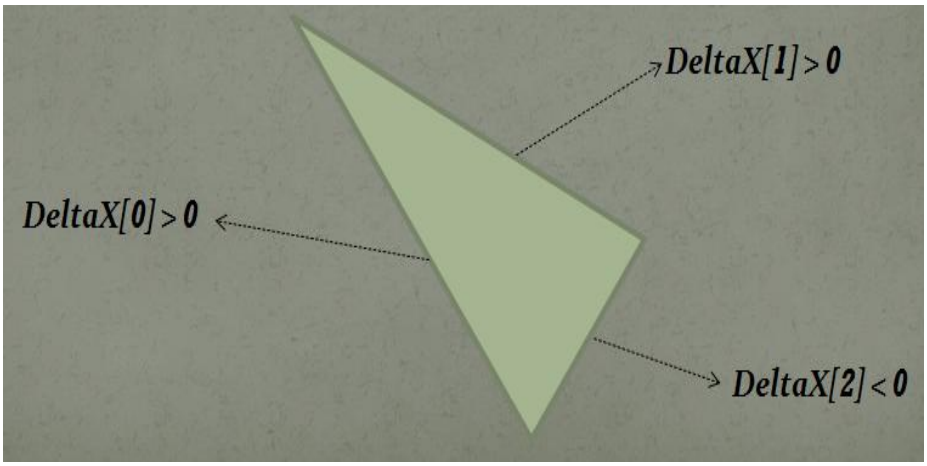
- **Case 6:**
    - DeltaX[0] < 0
    - DeltaX[1] < 0
    - DeltaX[2] < 0 (Implied because middle is on the right)

The case is shown in the following figure:



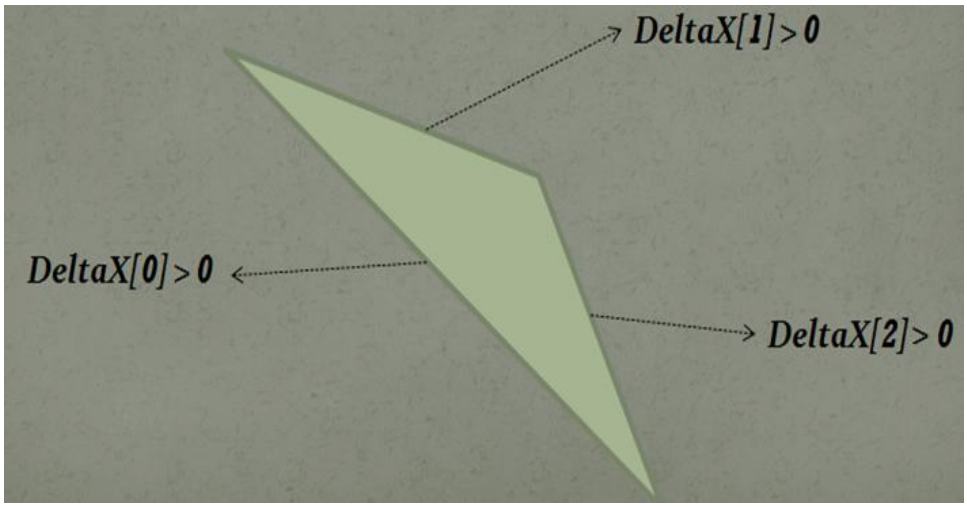
- **Case 7:**
    - DeltaX[0] > 0
    - DeltaX[1] > 0 (Implied because middle is on the right)
    - DeltaX[2] < 0

The case is shown in the following figure:



- **Case 8:**
    - DeltaX[0] > 0
    - DeltaX[1] > 0 (Implied because middle is on the right)
    - DeltaX[2] > 0

The case is shown in the following figure:



5.3. Triangle Setup

A slightly different triangle setup is needed in order to determine the triangle case. This setup is an extension to the one presented in the traditional Edge Walking rasterizer section. The following pseudo-code illustrates the new setup:

```
If (P0y < P1y)
    If (P2y < P0y)
        Top = 2
        Middle = 0
        Bottom = 1
        MiddleIsLeft = 1
    Else
        Top = 0
        If (P1y < P2y)
            Middle = 1
            Bottom = 2
            MiddleIsLeft = 1
        Else
            Middle = 2
            Bottom = 1
            MiddleIsLeft = 0
Else
    If (P2y < P1y)
        Top = 2
```

```
            Middle = 1
            Bottom = 0
            MiddleIsLeft = 0
        Else
            Top = 1
            If (P0y < P2y)
                Middle = 0
                Bottom = 2
                MiddleIsLeft = 0
            Else
                Middle = 2
                Bottom = 0
                MiddleIsLeft = 1
    If (MiddleIsLeft)
        If(Dx[0] < 0)
            Common Span Top Left X to Bottom Right X
            If(Dx[2] > 0)
                Set Update Flag for Left Edge in Second Half
        Else
            If(Dx[1] < 0)
                Common Span Top Left X to Top Right X
                Set Update Flag for Left Edge in Second Half
            Else
```

Common Span Bottom Left X to Top Right X

Else

If(Dx[0] < 0)

If(Dx[1] > 0)

Common Span Top Left X to Top Right X

Set Update Flag for Right Edge in Second Half

Else

Common Span Top Left X to Bottom Right X

Else

Common Span Bottom Left X to Top Right X

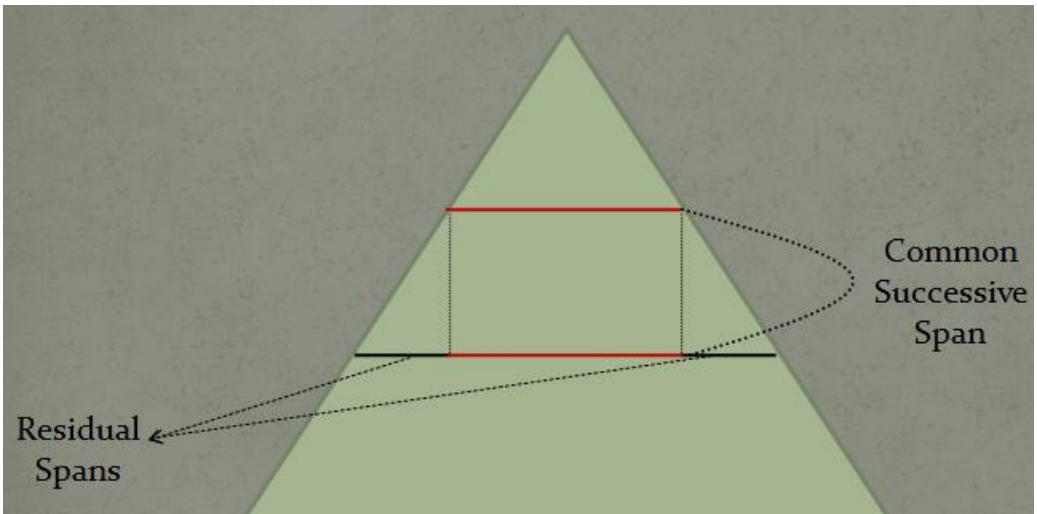
If(Dx[2] < 0)

Set Update Flag for Right Edge in Second Half

As can be seen, practically only two extra branches have been added during triangle setup wich is an acceptable cost.

5.4. Proposed Edge Walking Rasterizer

The propose Edge Walking rasterizer first determines the triangle case during triangle setup. Then, similar to the double step line rasterizer, it walks on span pairs, rasterizes the common span first then the two residual spans as shown in the following figure (successive spans are spaced apart for better visualization):



The gain is less range checking and better cache usage. In fact, the first implementation of this algorithm (with color interpolation) shows a 35% improvement over the traditional Edge Walking algorithm.

5.5. Proposed Tile-Based Rasterizer

The proposed Tile-Based algorithm is envisioned as some kind of a hybrid between the discussed Tile-Based algorithm in the previous section and the proposed optimized Edge Walking algorithm.

A good consequence of this hybrid walking scheme is that we do not encounter any redundant “ghost” tiles which is a step further from traversal schemes such as the presented zigzag method. Furthermore, since no (or little) edge function testing is used, we save up to three tests per tile (comparing with the most optimized one tile test per edge algorithm – see section 3). These two optimizations are expected to significantly boost the traditional Tile-Based rasterizer performance. But this algorithm is left as future work.

6. Implementation

6.1. Language Choice

First we needed to choose the most suitable programming language for the task. The choice was between C or C++. On one hand, the C++ implementation would be both well-structured and elegant. On the other hand, the C implementation would have little overhead in dealing with the C++ language complexities so it would be easier for the compiler to optimize the rasterizer code therefore yielding better performance. We chose to sacrifice elegance for performance as we are only implementing a triangle rasterizer not a full engine where code design and structuring is crucial.

The next step was to reach an optimal implementation of the normal triangle rasterizer so we can have something to compare the algorithm with.

6.2. Floating-Point Edge Walking Rasterizer

Since we are interested in getting the pure triangle traversal speed-up, we have not included any shading or texturing in the implementation. The first few implementations of the normal edge walking rasterizer performed well and were based on using floating-point arithmetic. We incrementally optimized this rasterizer by looking at the generated disassembly code. The major problem was that the compiler generated a ‘rep stos’ instruction to fill an entire span. That is due to the span having the same color since no color interpolation was being performed. The ‘rep stos’ instruction is used by the CPU to quickly fill memory by a uniform value (used in memset). This would be an unrealistic scenario for a triangle rasterizer

unless flat shading was being used. Further, we needed to emulate the per-pixel branching due to z-buffering. The following pseudo-code fragment is the result of fixing both of these issues:

Get VideoBuffer Pointer At Current Scan Line

Loop Through Scan Line

Set Color

Interpolate some value and apply non-logic changing conditional to it

The Interpolate value is a dummy value that emulates color variation. The conditional emulates z-buffering. The reason why we didn't fully implement color interpolation and z-buffering is simply because we're interested in keeping the color filling as light as possible to focus on the actual traversal speed gains.

The final implementation of the floating-point rasterizer is the Triangle_Float() function found in the Appendix.

6.3. Integer Edge Walking Rasterizer

Although the floating-point rasterizer was performing very well, we wanted to try to implement a rasterizer that runs exclusively on the CPU without any FPU intervention. With current technology, floating-point arithmetic on the FPU is as fast, if not faster, than integer arithmetic on the CPU, however when both processors communicate and synchronise with each other, vital cycles are lost.

The integer edge walking implementation turned out to be faster than the floating-point implementation. Looking at the dissassembly code, the compiler was keeping more variables in the high-speed CPU registers and no FPU synchronization instructions were being used.

6.4. Fast Floating-Point Edge Walking Rasterizer

The first attempt at implementing the previously discussed optimization ideas was on the floating-point rasterizer. First, we implemented the code to determine the triangle case (one of the eight above mentioned cases). Once the case is determined, it was crucial to avoid unecessary branching based on the case. we came up with the idea of keeping two consecutive spans along with their common and uncommon span indices. The following pseudo-code is used to determine the triangle case:

```
/*Setting leftEdge, rightEdge, Common and Uncommon edges indices*/
If Middle Is On the Left
    /*Left and Right edges*/
    leftEdge=1;
    rightEdge=0;
    If dx[0]<0
    /*What's sure in this case is that dx[1]<0 because the middle is on
    the left*/
        ucL=1;cL=0;
        ucR=0;cR=1;
        If dx[2]>0
            /*Update Value is for left edge*/
            eUpdate=1;
        Else
            /*No Update Needed*/
            eUpdate=0;
```

```
        Else
        /*What's sure in this case is that dx[2]>0 because the middle is on
        the left*/
            cR=0;ucR=1;
            If dx[1]<0
                ucL=1;cL=0;
                /*Update Value is for left edge*/
                eUpdate=1;
            Else
                ucL=0;cL=1;
                /*No Update Needed*/
                eUpdate=0;
    Else
        /*Left and Right edges*/
        leftEdge=0;
        rightEdge=1;
        if dx[0]<0
            ucL=1;cL=0;
            /*What's sure in this case is that dx[2]<0 because the
            middle is on the right*/
            If dx[1]>0
                ucR=1;cR=0;
                /*Update Value is for right edge*/
```

```
            eUpdate=1;
        Else
            ucR=0;cR=1;
            /*No Update Needed*/
            eUpdate=0;
    Else
        /*What's sure in this case is that dx[1]>0 because the
        middle is on the right*/
        ucL=0;cL=1;
        ucR=1;cR=0;
        if dx[2]<0
            /*Update Value is for right edge*/
            eUpdate=1;
        Else
            /*No Update Needed*/
            eUpdate=0;
```

The variables ‘ucL’ and ‘cL’ stand for uncommon left and common left respectively. Similarily the variables ‘ucR’ and ‘cR’ stand for uncommon right and common right respectively. Further, since the triangle is being split into two parts based on its middle point, the ‘eUpdate’ variable is used to actually update the edges indices when moving to the second part without having to re-check for the triangle case again. The following pseudo-code shows some clever binary manipulation to avoid branching during the edge indices update:

```
/*Updating the leftEdge, rightEdge, Common and Uncommon indices*/
leftEdge<<=1;rightEdge<<=1;
cL+=eUpdate&MidIsLeft;ucL-=eUpdate&MidIsLeft;
cR+=eUpdate&!MidIsLeft;ucR-=eUpdate&!MidIsLeft;
```

It would be hard to clearly understand the logic behing these indices and updates and the way to imagine how it works is to take concrete examples of triangles. We considered all possible triangle cases in the above code and made sure they were all handled correctly while optimizing speed as much as possible.

After this phase, the rasterizer now knows the common and uncommon spans by indexing without explicitly checking for them at every span. The spans are filled in two steps, first the uncommon spans are filled (generally a few number of pixels) then the common span is filled simulatenously for both successive spans (a relatively much larger number of pixels). This represents the core optimization startegy.

The comparison with the normal floating-point rasterizer shows on average a %20 increase in speed for arbitrary triangle shapes and sizes. This is an acceptable performance boost, however we were convinced that we could obtain a better performance gain.

6.5. Fast Integer Edge Walking Rasterizer

Because of the previously mentioned co-processor synchronization issues, we applied the algorithm on the integer rasterizer to compare the performance gains. Basically, the edge indexing and update optimizations are the same and were adapated to the integer rasterizer. It turns out that the performance boost is even greater when comparing the normal integer rasterizer with the optimized integer

rasterizer. We obtained on average a %30 performance boost for arbitrary triangle shapes and sizes.

6.6. Fast Edge Walking Rasterizer Variations

Since determing the triangle case and updating the indices all require a dereference operation which could lead to expensive cache misses, we had the idea of avoiding as much dereferencing as possible by expanding the triangle rasterizer cases in code therefore pushing as much branching operations as possible to the beginning of the algorithm.

The first variation was a ‘medium’ sized rasterizer, we have split the code to handle the triangle rasterization into two major parts based on whether the middle point is on the left or right. This allows us to save a few dereference operations however the benchmarking results were not encouraging.

Then we re-structured the rasterizer by expanding all the eight cases and avoiding a lot dereferencing operations.

This variation is the ‘large’ sized raterizer since the resulting function turned out to be 1082 lines of code which is a lot compared to the small algorithm’s 377 lines. However, we gained another %5-%10 average performance boost making the algorithm %35-%40 faster than the normal integer triangle rasterizer.

6.7. Robustness

6.7.1. Ill-Shaped Triangles

Because of polygon anomalies sometimes found in the 3D models and because of the variation in camera view, we are sometimes faced with

projected screen space triangles that are ill-shaped. By ill-shaped, we mean extremly slanted in such a way that two successive span lines share a very small common span (or even no common span) as shown in the following figure:

The above image is low-res in order to clearly show wut's happening on a per pixel basis. Notice how only the small red portion constitutes some common spans and the black part will make up all the residuals. The implication in this case is not incorrect triangle display using our algorithm, it is however less speed enhancement. The logic here is simple, since the core speed optimization relies on rasterizing common span pixels, the fewer these pixels, the less speed gain we get.

6.7.2. Pixel Fidelity

Another important issue to address here is which pixels get rasterized using our algorithm in comparison with the normal rasterizer. Due to the two algorithms being structurally different, this question actually makes sense since we may end up missing some pixels and adding others. In order to be consistent, It is important that our algorithm rasterizes the exact same pixels as the normal rasterizer and it does. The proof that our algorithm is consistent follows: First the edge slope computations are the same in both algorithms,

so is the starting point (top vertex). In our proposed algorithm we walk on pairs of spans in contrast to the normal rasterizer which only considers a single span at a time. The next span extents are found incrementally by adding the corresponding inverse slope value to the current span extents. In our algorithm we have two options to deal with this update:

NextPairExtents[0] = CurExtents + InvSlope

*NextPairExtents[1] = CurExtents + 2*InvSlope*

Or

NextPairExtents[0] = CurExtents + InvSlope

NextPairExtents[1] = NextPairExtents[0] + InvSlope

Due to floating-point precision issues, only the second option yields consistent results with the normal rasterizer since they both perform the exact same arithmetic operations.

7. Appendix

7.1. EdgeWalkTriangle()

Determine Positions of the three vertices assuming CCW ordering

If (Py0 < Py1)

If (Py2 < Py0)

Top = 2

Middle = 0

Bottom = 1

MiddleIsLeft = 1

Else

Top = 0

If (Py1 < Py2)

Middle = 1

Bottom = 2

MiddleIsLeft = 1

Else

Middle = 2

Bottom = 1

MiddleIsLeft = 0

Else

If (Py2 < Py1)

Top = 2

Middle = 1

Bottom = 0

MiddleIsLeft = 0

Else

Top = 1

If (Py0 < Py2)

Middle = 0

Bottom = 2

MiddleIsLeft = 0

Else

Middle = 2

Bottom = 0

MiddleIsLeft = 1

Set inverse slope values for all three triangle edges

$$InvSlope[0] = (Px_{Bottom} - Px_{Top})/(Py_{Bottom} - Py_{Top})$$

InvSlope[1] = (Px_{Middle} - Px_{Top}) /(Py_{Middle} - Py_{Top})

InvSlope[2] = (Px_{Bottom} - Px_{Middle}) /(Py_{Bottom} - Py_{Middle})

Determine left and right active edges

leftEdge = MidIsLeft

rightEdge = ***Reverse*** *MidIsLeft*

Split triangle from top to middle and set starting and ending y values

yStart = Ceil(Py_{Top})

yEnd = Floor(Py_{Middle})

Set initial x value to top vertex

xLeft = xRight = Px_{Top}

Fill Top To Middle Triangle Part

Loop *y = yStart* ***While*** *y<=yEnd* ***Do*** *y++*

Get Current Horizontal Span

xStart = Ceil(xLeft)

xEnd = Floor(xRight)

Loop *x = xStart* ***While*** *x<= xEnd* ***Do*** *x++*

Set Pixel at (x, y)

Update Left and Right x by Inverse Slope of respective edge

xLeft += InvSlope[leftEdge]

xRight += InvSlope[rightEdge]

Update left and right active edges

leftEdge = ***ShiftLeft*** *MidIsLeft* ***By*** *1*

rightEdge = ***ShitfLeft*** *(****Reverse*** *MidIsLeft)* ***By*** *1*

Split triangle from middle to bottom and set starting and ending y values

yStart = Ceil(Py_{middle}*)*

yEnd = Floor(Py_{Bottom}*)*

Fill Middle To Bottom Triangle Part

Loop *y = yStart* ***While*** *y<=yEnd* ***Do*** *y++*

Get Current Horizontal Span

xStart = Ceil(xLeft)

xEnd = Floor(xRight)

Loop *x = xStart* ***While*** *x<= xEnd* ***Do*** *x++*

Set Pixel at (x, y)

Update Left and Right x by Inverse Slope of respective edge

xLeft += InvSlope[leftEdge]

xRight += InvSlope[rightEdge]

7.2. TileBasedTriangle()

Determine Bounding Rectangle

MinX = Min (Px0, Px1, Px2)

MaxX = Max (Px0, Px1, Px2)

MinY = Min (Py0, Py1, Py2)

MaxY = Max (Py0, Py1, Py2)

Set Fixed Block Size n (could be statically or dynamically determined)

BlockSize = n

Go through n*n blocks in triangle bounding rectangle

Loop y = MinY While y<=MaxY Do y+=BlockSize

Loop x = MinX While x<=MaxX Do x+=BlockSize

Get Current Tile Corners

TileMinX = x

TileMaxX = x + BlockSize - 1

TileMinY = y

TileMaxY = y + BlockSize - 1

Check Tile Against Edge Functions

Bool InsideEdge1 = EdgeFunction1(TileMinX, TileMaxX,
TileMinY, TileMaxY) > 0

Bool InsideEdge2 = EdgeFunction2(TileMinX, TileMaxX,
TileMinY, TileMaxY) > 0

Bool InsideEdge3 = EdgeFunction3(TileMinX, TileMaxX,
TileMinY, TileMaxY) > 0

Do Full Rejection Test

*If (**Reverse** InsideEdge1) **AND** (**Reverse** InsideEdge2)*
***AND** (**Reverse** InsideEdge3)*

Go To Next Tile

Do Full Acceptance Test

*If (InsideEdge1) **AND** (InsideEdge2) **AND** (InsideEdge3)*

Rasterize Full Tile

Go To Next Tile

Otherwise Tile Is Partially Covered

Loop Ty = TileMinY While Ty<=TileMaxY Do Ty++

Loop Tx = TileMinX While Tx<=TileMaxX Do x++

Check If Pixel at (x, y) is inside triangle

If IsInside(Tx, Ty)

Set Pixel at (Tx, Ty)

7.3. FastEdgeWalkTriangle()

Determine Positions of the three vertices assuming CCW ordering

If (Py0 < Py1)

If (Py2 < Py0)

Top = 2

Middle = 0

Bottom = 1

MiddleIsLeft = 1

Else

Top = 0

If (Py1 < Py2)

Middle = 1

Bottom = 2

MiddleIsLeft = 1

Else

Middle = 2

Bottom = 1

MiddleIsLeft = 0

Else

If (Py2 < Py1)

Top = 2

Middle = 1

Bottom = 0

MiddleIsLeft = 0

Else

Top = 1

If (Py0 < Py2)

Middle = 0

Bottom = 2

MiddleIsLeft = 0

Else

Middle = 2

Bottom = 0

MiddleIsLeft = 1

Set DeltaX Values

$DeltaX[0] = Px_{Bottom} - Px_{Top}$

$DeltaX[1] = Px_{Middle} - Px_{Top}$

$DeltaX[2] = Px_{Bottom} - Px_{Middle}$

Set DeltaY Values

DeltaY[0] = Py_{Bottom} - Py_{Top}

DeltaY[1] = Py_{Middle} - Py_{Top}

DeltaY[2] = Py_{Bottom} - Py_{Middle}

Set inverse slope values for all three triangle edges

InvSlope[0] = DeltaX[0] / DeltaY[0]

InvSlope[1] = DeltaX[1] / DeltaY[1]

InvSlope[2] = DeltaX[2] / DeltaY[2]

If MiddleIsLeft

Update left and right active edges

leftEdge=1

rightEdge=0

If DeltaX[0]<0

Update Common/Uncommon Left and Right Indices

ucL=1 / cL=0

ucR=0 / cR=1

If DeltaX [2]>0

Update Value for left edge

eUpdate=1

Else

No Update Needed

eUpdate=0

Else

Update Common/Uncommon Right Indices

cR=0 / ucR=1

If DeltaX[1]<0

Update Common/Uncommon Left Indices

ucL=1 / cL=0

Update Value for left edge

eUpdate=1

Else

Update Common/Uncommon Left Indices

ucL=0 / cL=1

No Update Needed

eUpdate=0

Else

Update left and right active edges

leftEdge=0

rightEdge=1

if DeltaX[0]<0

Update Common/Uncommon Left Indices

ucL=1 / cL=0

If DeltaX [1]>0

Update Common/Uncommon Right Indices

ucR=1 / cR=0

Update Value for right edge

eUpdate=1

Else

Update Common/Uncommon Right Indices

ucR=0 / cR=1

No Update Needed

eUpdate=0

Else

Update Common/Uncommon Left and Right Indices

ucL=0 / cL=1

ucR=1 / cR=0

if DeltaX[2]<0

Update Value for right edge

eUpdate=1

Else

No Update Needed

eUpdate=0;

Split triangle from top to middle and set starting and ending y values

$yStart = Ceil(Py_{Top})$

$yEnd = Floor(Py_{middle})$

Set initial x value to top vertex

xLeft[0] = xRight[0] = PxTop

Step By one if needed since this algorithm relies on traversing edge pairs so the edges number in the main loop must be even

If IsOdd(yEnd – yStart)

Loop *x = xLeft[0]* ***While*** *x<=xRight[0]* ***Do*** *x++*

Set Pixel (x, yStart)

Update Left and Right x by Inverse Slope of respective edge

Xleft[0] += InvSlope[leftEdge]

XRight[0] += InvSlope[rightEdge]

Fill Top To Middle Triangle Part

Loop *y = yStart* ***While*** *y<=yEnd* ***Do*** *y+=2*

Get New Left and Right x

Xleft[1] = Xleft[0] + InvSlope[leftEdge]

XRight[1] = XRight[0] + InvSlope[rightEdge]

Fill First Residual Span

xStart = Ceil(xLeft[ucL])

xEnd = Floor(xRight[cL])

Loop *x = xStart* ***While*** *x<= xEnd* ***Do*** *x++*

Set Pixel at (x, y+ucL)

Fill Second Residual Span

xStart = Ceil(xLeft[cR])

xEnd = Floor(xRight[ucR])

Loop *x = xStart* ***While*** *x<= xEnd* ***Do*** *x++*

Set Pixel at (x, y+ucR)

Fill Common Span

xStart = Ceil(xLeft[cL])

xEnd = Floor(xRight[cR])

***Loop** x = xStart **While** x<= xEnd **Do** x++*

Set Pixel at (x, y)

Set Pixel at (x, y + 1)

Update Left and Right x by Inverse Slope of respective edge

Xleft[0] = Xleft[1] + InvSlope[leftEdge]

XRight[0] = XRight[1] + InvSlope[rightEdge]

Update left and right active edges

*leftEdge = **ShiftLeft** leftEdge **By** 1*

*rightEdge = **ShitfLeft** rightEdge **By** 1*

Update Common/Uncommon indices

*cL += eUpdate **AND** MidIsLeft*

*ucL -= eUpdate **AND** MidIsLeft*

*cR += eUpdate **AND** (**Reverse** MidIsLeft)*

*ucR -= eUpdate **AND** (**Reverse** MidIsLeft)*

Split triangle from middle to bottom and set starting and ending y values

yStart = Ceil(Py_{middle})

yEnd = Floor(Py_{bottom})

Step By one if needed

If IsOdd(yEnd – yStart)

Loop *x = xLeft[0]* ***While*** *x<=xRight[0]* ***Do*** *x++*

Set Pixel (x, yStart)

Update Left and Right x by Inverse Slope of respective edge

Xleft[0] += InvSlope[leftEdge]

XRight[0] += InvSlope[rightEdge]

Fill Middle To Bottom Triangle Part

Loop *y = yStart* ***While*** *y<=yEnd* ***Do*** *y+=2*

Get New Left and Right x

Xleft[1] = Xleft[0] + InvSlope[leftEdge]

XRight[1] = XRight[0] + InvSlope[rightEdge]

Fill First Residual Span

xStart = Ceil(xLeft[ucL])

xEnd = Floor(xRight[cL])

Loop *x = xStart* ***While*** *x<= xEnd* ***Do*** *x++*

Set Pixel at (x, y+ucL)

Fill Second Residual Span

xStart = Ceil(xLeft[cR])

xEnd = Floor(xRight[ucR])

Loop *x = xStart* ***While*** *x<= xEnd* ***Do*** *x++*

Set Pixel at (x, y+ucR)

Fill Common Span

xStart = Ceil(xLeft[cL])

xEnd = Floor(xRight[cR])

Loop *x = xStart* ***While*** *x<= xEnd* ***Do*** *x++*

Set Pixel at (x, y)

Set Pixel at (x, y + 1)

Update Left and Right x by Inverse Slope of respective edge

Xleft[0] = Xleft[1] + InvSlope[leftEdge]

XRight[0] = XRight[1] + InvSlope[rightEdge]

8. References

[1] Alvay Ray Smith, A Pixel Is Not a Little Square, Technical Memo 6, July 17, 1995

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

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

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

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

[6] Dan Crisu, Sorin Cotofana, Stamatis Vassiliadis and Petri Liuha, "Design Tradeoffs for an Embedded OpenGL-Compliant Hardware Rasterizer"

[7] Flynn MJ, Very High-Speed Computing Systems, Proceeding of the IEEE, 54(12), December 1966, p1901-1909

[8] Parallel Computer Taxonomy, Wasel Chemij, MPhil, Aberystwyth University, 1994

[9] Wikipedia, “Flynn Taxonomy” Article. http://en.wikipedia.org/wiki/Flynn's_taxonomy

[10] Larry Carter, CSE 260 – Introduction to Parallel Computation, 2001

[11] Radha Krishna, Master’s Thesis - Design of 3D Accelerator for Mobile Platform, May 2006

[12] Fredrik Ehnbom, Master’s Thesis - A Tile-Based Triangle Rasterizer in Hardware, 2005

[13] DevMaster, Advanced Rasterization, http://devmaster.net

[14] Dan Crisu, Sorin Cotofana, Stamatis Vassiliadis and Petri Liuha, “Efficient Hardware for Tile-Based Rasterization”

[15] Tomas Akenine-Möller, Aila and Timo, “Conservative and Tiled Rasterization Using a Modified Triangle Setup”, to appear in Journal of graphics tools

[16] Tomas Akenine-Möller, [Mobile] Graphics Hardware, Lund University, September 2007

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

[18] Michael D. McCool, Chris Wales, and Kecin Moule, Incremental and Hierarchical Hilbert Order Edge Equation Polygon Rasterization, In Graphics Hardware 2001

[19] John Owens, EEC 277, Graphics Architecture - Technical Report, 2005

[20] Tomas Akenine-Möller, Appendix A – Fixed Point Mathematics, 2007