Implementation of a GPU rasterization stage on a FPGA

Author:
Albert Navarro Torrentó

Grado en Ingeniería Informática
Especialidad en Ingeniería de Computadores

15 de Junio de 2015

Director:
Agustín Fernandez, Arquitectura de computadores

Facultad de Informática de Barcelona
Universidad Politécnica de Cataluña (UPC) - BarcelonaTech

Resumen

En el siguiente trabajo veremos los pasos que se han seguido para realizar la implementación de la etapa de rasterización de una GPU.

En primer lugar, el lector podrá encontrar toda aquella información necesaria para adquirí una base general sobre la rasterización.

Seguidamente se presentará el diseño del rasterizador que se ha implementado, el cual se analizará y ademas, se estudiaran algunos análisis referentes al funcionamiento del diseño

Finalmente se expondrá la planificación del proyecto y las conclusiones finales de este.

Resum

En el següent treball veurem els passos que s'han seguit per la realització de la implementació de l'etapa de rasterització d'una GPU.

En primer lloc, el lector trobarà tota aquella informació necessària per adquirí una base general sobre la rasterització.

Seguidament es presentarà el disseny del rasteritzador que s'ha implementat, el qual s'analitzarà en detall i a més, s'estudiaran algun anàlisis referents al funcionament del disseny

Finalment s'exposarà la planificació del projecte i les conclusions finals d'aquest.

Abstract

In the following document we will see all the necessary steps to make an implementation of the GPU rasterization stage.

First, the reader will find all of the necessary information to a get a basic knowledge on rasterization.

Next, we will show the implemented rasterizer design, which will be analyzed in detail as well as study some benchmarks of the design.

Finally, we will expose the planning of the project and the final conclusions of the project.

Contents

List of Figures

List of Tables

Acronyms

ASIC Application-Specific Integrated Circuit. 13

FPGA Field Programmable Gate Array. 9, 10, 13, 37, 38, 46, 48

GPU Graphics Processing Unit. 9, 10, 20, 22, 26, 27, 46–48

HDL Hardware Description Language. 9, 13, 14

LE Logical Element. 13

LUT Look-up table. 13

PLD Programmable Logic Devices. 13

TE Tile evaluator. 30

Chapter 1

Introduction

1.1 Motivation and objectives

In the last 20 years, the importance of *Graphics Processing Units (GPUs)* has increased. These devices are designed to process graphics, but they are also usually used in clusters as accelerators.

Initially, graphic scenes were composed by a few really big triangle. These devices use complex algorithms designed to rasterized big triangles effectively. Currently, graphics scenes are composed by many small triangles. however the algorithms on the *gpu* are not optimal.

The idea of our project is to make an implementation of a rasterization stage with a specialized part for the 2 types of triangles: One part that rasterizes big triangles and another part that rasterizes small triangles.

To do it, we will use a *Hardware Description Language (HDL)* to make the design and a *Field Programmable Gate Array (FPGA)* to test the design. Also this design can be used as part of a full *GPU* implementation.

The motivation to make this implementation is to have a fully functional rasterizer to study what is the performance achieved with the specialized way for small triangles.
implementation.

1.2 Actors

In this part we will list the involved parts on the project, directly or indirectly

1.2.1 Project Director

The project director is Agustín Fernandez, professor at the "Universitat Politecnica de Catalunya" his role is to supervise that the project objectives are being accomplished. Additionally he guides and helps the developer carry out the project.

1.2.2 Developer

This is the role developed by the student, in this case me. He has to carry out the project.

1.2.3 Researchers

Researchers interested in *GPU*s can find our project useful to make a full *GPU* implementation or to have a template to implement other rasterization algorithms.

1.2.4 Students and Teachers

*GPU*s are devices with very few public documentation so it's hard to learn about them. Students can find this implementation useful to learn more about a *GPU*s rasterization stage, how it works and how to implement it. Also teachers can show the implementation to their students to teach them how the stage works and how it's implemented.

1.3 Possible problems

1.3.1 Bugs

Bugs are a problem in any type of project and they can slow us down, which could be a real problem with planning. It's also possible that a bug can be produced by the testing hardware and not by the rasterizer hardware. This can be confusing when we are searching the bug and can make us waste a lot of time. We have to keep these problems in mind.

1.3.2 Hardware development difficulty

Hardware programming is more difficult than software programming. You have to program thinking that you are making an electric circuit when you are writing code. In addition, debugging is more difficult since the only way you have to do it, is using the simulator or using a display to show the value of some signals when you are using the *FPGA*.

1.4 Development tools

1.4.1 Quartus II

Quartus II is a software provided by Altera to design and implement hardware design on their *FPGA*s. This is a very powerful software, even in their free edition and lets you make your designs using a hardware description language tool that lets you make the schema of the hardware.

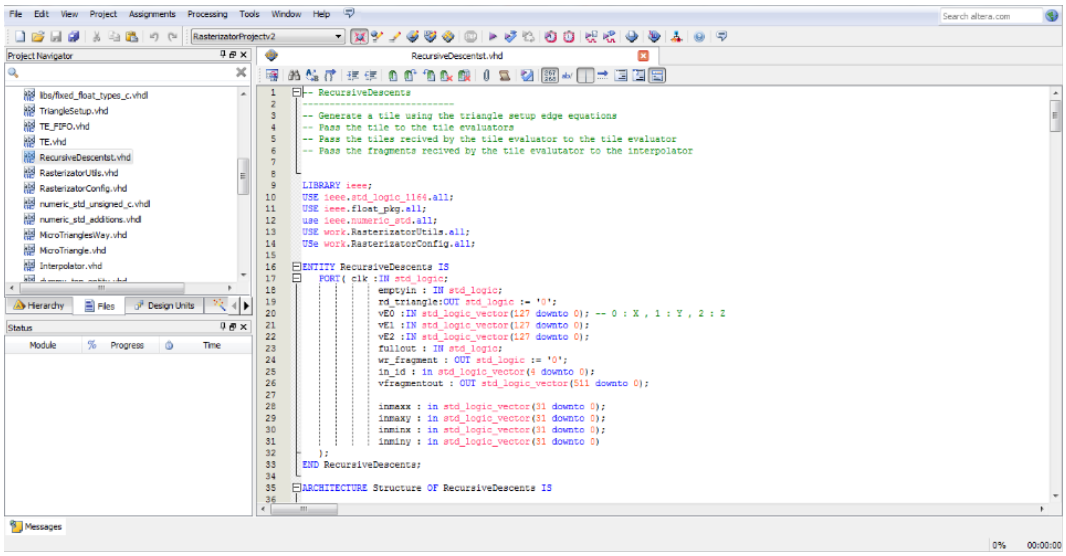

Figure 1.1: Quartus II IDE

1.4.2 ModelSim

ModelSim is a simulator provided by Altera to simulate the hardware designs without using the FPGA. ModelSim lets you see the state of every signal during the simulation, making it easier to debug and see if the signals values are what we expect.

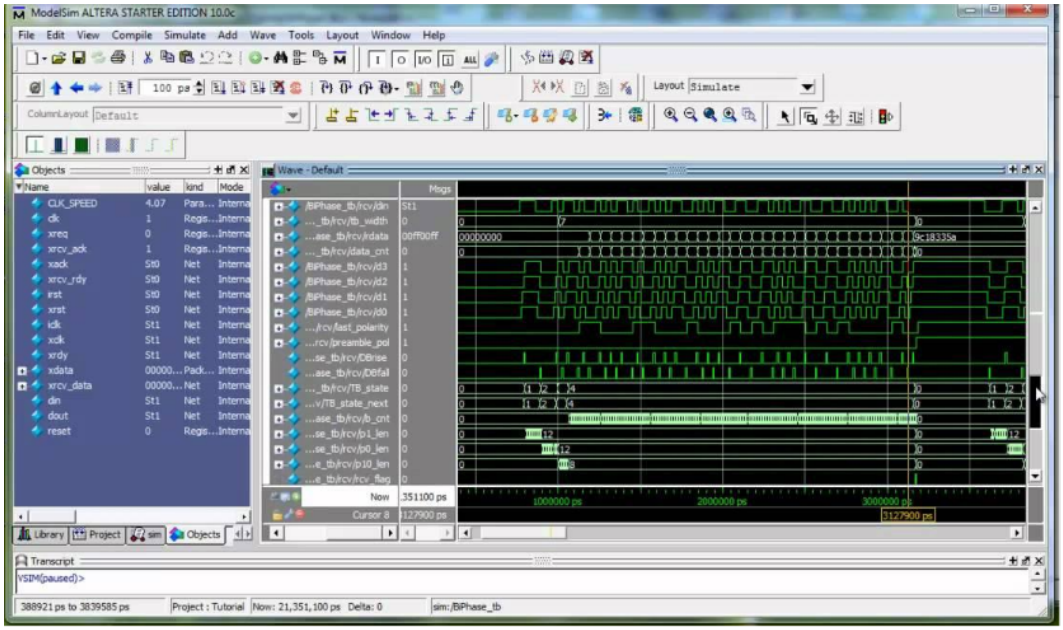

Figure 1.2: Modelsim Simulator

1.4.3 FPGA Altera DE2-115

This is the device provided for the project. The board components are :

- FPGA Altera Cyplone IV E with 115K LEs

- 3888 embeded memory (Kbits)
- 266 embedded 18x18 multipliers
- 528 User I/O
- 128MB SDRAM
- 2MB SRAM
- 8MB FLASH
- 7-segments displays
- VGA out
- Mic line
- Switches

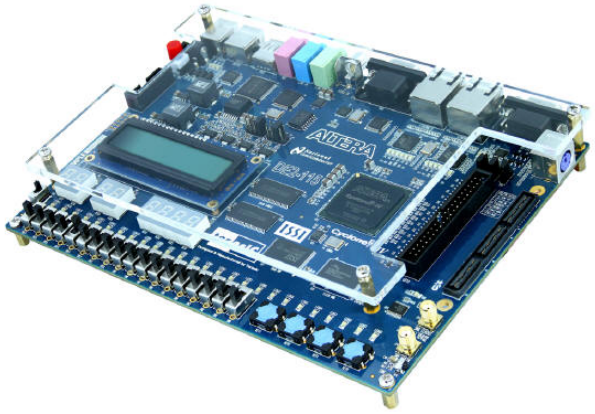

Figure 1.3: FPGA DE2-115

1.4.4 GNU Octave

GNU Octave is a high-level interpreted language, primarily intended for numerical computations. It provides capabilities for the numerical solution of linear and nonlinear problems, and for performing other numerical experiments. It also provides extensive graphics capabilities for data visualization and manipulation. Octave is normally used through its interactive command line interface, but it can also be used to write non-interactive programs. The Octave language is quite similar to Matlab so that most programs are easily portable.

We will use Octave to generate an image with the raw values generated by the pipeline.

Figure 1.4: GNU Octave

1.5 Potential Technologies

1.5.1 FPGA

An *FPGA*, is an integrated circuit designed to be configured by a designer after manufacturing. *FPGA*s are in between *Programmable Logic Devices (PLD)* and *Application-Specific Integrated Circuit (ASIC)*. *FPGA* are not as powerful as *ASIC* but they are cheaper and re-programmable and the *PLD* are not as powerful as *FPGA*.

The *FPGA*s are devices composed many *Logical Element (LE)*. These *LE*s are composed by a multiplexor, a register and a *Look-up table (LUT)*. The *LUTs* are very powerful because you can define an output for a given input without any logic. Every *LE* is connected with an interconnection network that communicates with the other *LE*s.

To program an *FPGA* you need a design written in *HDL*. The steps to generate a code for the FPGA are :

Synthesis:
Generate an implementation of the design using the board elements.

Placement:
Determinate the location of the board *LE*s.

Routing:
Determine the best routes to connect the devices.

With these steps we obtain a bitstream file, which is the binary configuration file of the *FPGA* for the compiled deign.

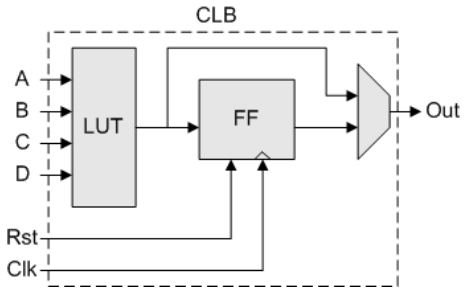

Figure 1.5: LU block

1.5.2 VHDL

VHDL (VHSIC Hardware Description Language) is a hardware description language used in electronic design. The HDL languages like VHDL are designed to describe how signals are transmitted and through time.

The *HDL* language is specifically designed to describe the structure of a design, that is how it is decomposed into sub-designs, and how those sub-designs are interconnected. The syntax is based on ADA and it was developed by the U.S Department of Defense. The alternative to VHDL is Verilog, an extended *HDL* language whose syntax is based on C.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity signed_adder is
  port
  (
    aclr : in   std_logic;
    clk  : in   std_logic;
    a    : in   std_logic_vector;
    b    : in   std_logic_vector;
    q    : out  std_logic_vector
  );
end signed_adder;

architecture signed_adder_arch of signed_adder is
  signal q_s : signed(a'high+1 downto 0); -- extra bit wide

begin -- architecture
  assert(a'length >= b'length)
    report "Port A must be the longer vector if different sizes!"
    severity FAILURE;
  q <= std_logic_vector(q_s);

  adding_proc:
  process (aclr, clk)
    begin
      if (aclr = '1') then
        q_s <= (others => '0');
      elsif rising_edge(clk) then
        q_s <= ('0'&signed(a)) + ('0'&signed(b));
      end if; -- clk'd
    end process;

end signed_adder_arch;
```

Figure 1.6: VHDL language

1.6 Methology

We have analyzed the possible work methodologies for this project, and we think that our best option is to apply some kind of agile methodology like SCRUM or Kanban. We have decided to use SCRUM with some modifications for our project. We will change the daily meetings to a meeting every 15 days as I can't meet every day with my tutor. We'll use SCRUM instead of Kanban because SCRUM defines some kind of iteration cycles of development which we think are good to track our project.

1.7 Validation

The idea of this project is to make a functional rasterizer with 2 specialized stages so we have a few points that we have to achieve.

Speedup with the specialization
: Obviously, if we are designing a specialized way for every type of triangle we want some kind of speedup

Correct Rasterization
: We want to generate correct frames with the resulting stage.

Chapter 2

State of the art

Our project objective is to make an implementation of a GPU rasterization stage. Therefore we are interested in the GPU graphics pipeline, which are the technology and algorithm to rasterize and if there are some usable rasterization stage or GPU implementation.

2.1 GPU pipeline

The objective of a GPU is to translate 3D triangles into a 2D image. This is divided in 3 steps : Geometry, Rasterization and Fragments Processing.

2.1.1 Geometry

In the geometry step, the GPU does all the necessary vertex operations. These are split into in many steps :

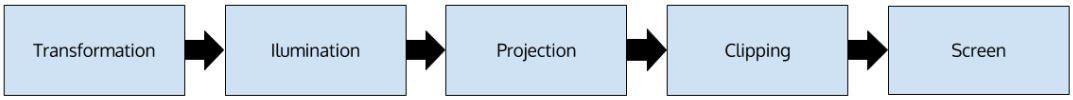

Figure 2.1: Geometry steps

2.1.1.1 Transformation

In this step, the transformations are applied in 3 spaces : model space,world space and view space.

The model space is the coordinate system in which the 3D object is defined. This coordinate system has the center defined in some point on the model and the transformations (move, rotate or scale the object) are applied relative to this point. This is very useful because we can use the same model with different transformations without

replicating the geometric information.

The world space is where the objects are placed. The operations in this space affect all the objects of the scene. This is only useful when you need to move the entire scene so many programmers skip this step and instead go directly to the view space.

In this space the camera, is positioned in the world by the programmer. The camera attributes define which are the objects that will be projected to the screen.

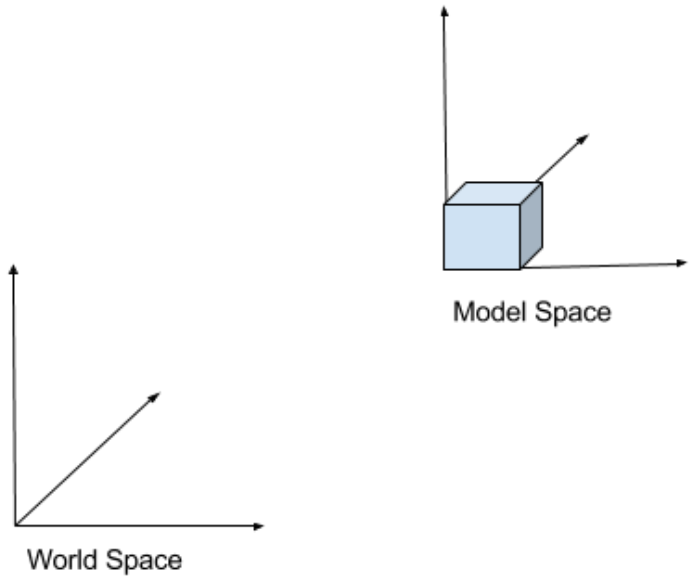

Figure 2.2: World space and model space

2.1.1.2 Illumination

When we render an image, usually the objective is to achieve the most realistic image possible. That's why in this step, the GPU emulates light using information stored in the vertex of the polygons. The information quantifies what the contribution of each vertex is.

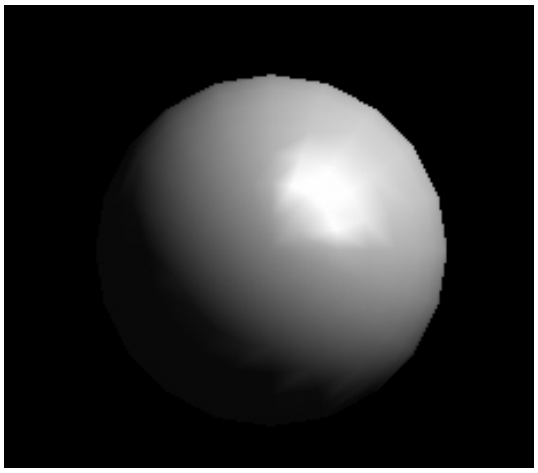

Figure 2.3: Illumination example

2.1.1.3 Projection

In this step, the GPU calculates the projection of the objects relative to the screen. The projection is defined by the camera and it can be orthographic or perspective. The ortographic projection respects the angles of the projection and the perspective camera emulates how the eye works.

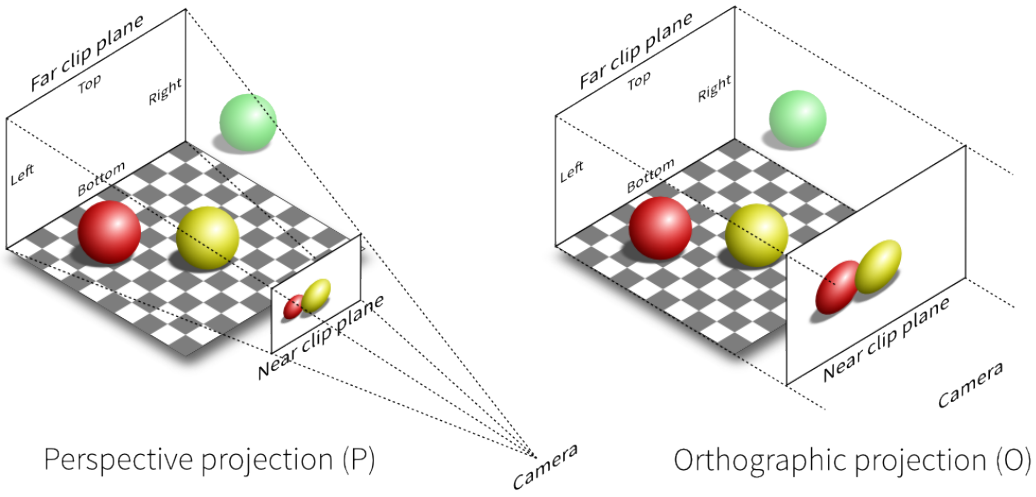

Figure 2.4: Orthographic and perspective camera

2.1.1.4 Clipping

In this step, the GPU removes the polygons that are out of view. This step is not necessary, but reduces the information that the GPU needs to process in future steps.

2.1.1.5 Screen Projection

Finally, in this step the coordinates x and y of the polygons are converted to screen coordinates.

2.2 Rasterization

The second step is rasterization, the process to translates triangles to fragments. There are many ways to do this process and in this section we describe a lot of them.

2.2.1 Scanline

This technique processes the framebuffer row by row and calculates if the pixel is inside a triangle using a table of points of the triangle edge that intersect the scanline. [7]

This a very primitive technique used in early 3D graphics because it is really cheap in memory. It's looks good for a software render or for a low memory implementation but is difficult to achieve a good parallelism and it's difficult to implement it on hardware.

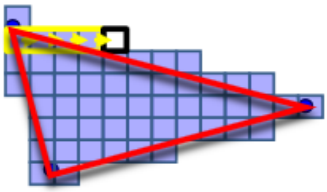

Figure 2.5: Scanline Rasterization

2.2.2 Centerline

This algorithm starts with the top-most vertex, processes all the left fragments and checks if any pixel is missing on the right. If something is missing, the algorithm processes it and then moves to the fragments under the initial one and repeats the process. With this type of scan, most of the operations produce useful data but it is hard to parallelize. [7]

The decision to start at the top vertex is arbitrary.

2.2.3 Cross-product

This algorithm tests directly the points using a test-in-polygon instead of the edge equations. This can be fully parallelized but needs a high amount of operations. It's useful when you have very small triangles (1-4 pixels) ,and the cost of building the edge equations testing that the is higher than the cross-product.[5][10]

The test is done in the following way: We compute the cross product between every vertex to vertex vector and the vertex to center vector. The result sign indicates if the point is at the left(+) or at the right(-) of the triangle edge.

$$e_0 = v_2 - v_1, e_1 = v_0 - v_2, e_2 = v_1 - v_0$$

$$p_0 = p - v_0, p_1 = p - v1, p_2 = p - v2$$

$$u_0 = e_0 x p_1 y - e_0 y p_1 x$$

$$u_1 = e_1 x p_2 y - e_1 y p_2 x$$

$$u_2 = e_2 x p_0 y - e_2 y p_0 x$$

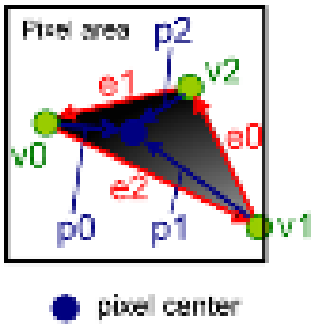

Figure 2.6: cross-product inside test

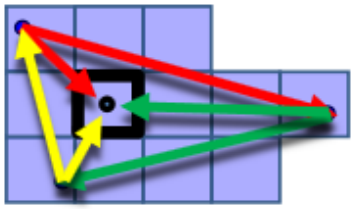

Figure 2.7: cross-product inside test

2.2.4 Edge equations algorithms

Edge equations provide a way to efficiently test if a point is inside a polygon. Building the edge equations is expensive but it is worth if you do many tests. Modern *GPUs* use edge equations. The edge equations define 2 half-planes, a negative plane and a positive plane. If a point is in the same plane for the 3 equations it means that the point is inside/outside the triangle[6][9][8].

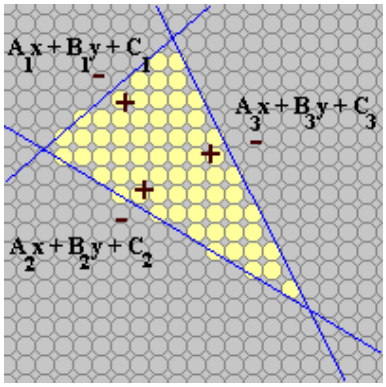

Figure 2.8: Edge equations inside test

2.2.4.1 Recursive

Used on Intel Larrabee [2]. This algorithm subdivides the bounding box of the triangle to search which parts of it contains fragments. The procedure is:

1. Build a top tile with the size of the triangle bounding box.
2. Split the tile in 4 equal parts (subtile)
3. Evaluate the edge equations of the triangle in the 4 points of each subtile that represents the bounding box of the subtile.
4. If each of the edge equations contains at least 1 point in the inside plane, there is a triangle inside the subtile so we repeat the process (starting at point 2).
5. If the subtile is of the side of a stamp (4 fragments) you don't have to subdivide, you have finished
6. If the subtile doesn't contain anything, discard it.

The evaluation of the points can be implemented like an incremental update since we know the distance between the points.

$$u = ax + by + c$$

$$c_new = c_old + (a << subtilelevel) + (b << subtilelevel)$$

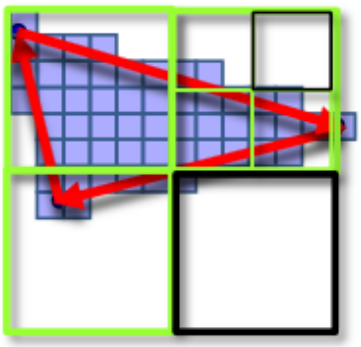

Figure 2.9: Recursive rasterization

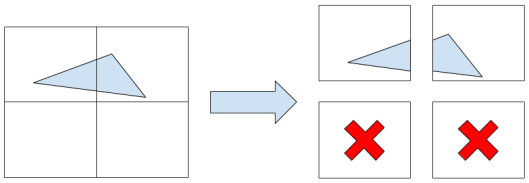

Figure 2.10: Recursive rasterization

2.2.4.2 Tile scan

This algorithm subdivides the bounding box of the triangles in fixed size tiles and tests the fragments of every tile using edge equations. The algorithm scales fine since all the tiles are independent and it is implemented on modern *GPU*s

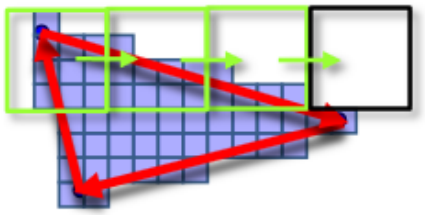

Figure 2.11: Tile scan rasterization

2.2.5 Fragments Processing

When the fragments are generated, the GPU applies transformations by the programmer to the fragments.This transformations are mostly : to calculate the color of the fragment based on the applied texture, transparency of the fragment, etc.

Finally, the GPU performs a set of reject tests before the fragments are sent to the framebuffer.

Scissor Test
Discard fragments outside the defined rending area.

Alpha test
Discard fragments with an alpha value that's too low. The minum value is defined by the programmer.

Stencil test
Discard fragments using the stencil buffer has a reference. The stencil buffer is defined by the programmer.

Z test
Discard the fragments that are behind other fragments. Use the Z buffer to compare the depth of the fragments.

Figure 2.12: Texture apply

2.3 GPU implementations

There are not many implementations of GPUs or parts of it. Mostly, they are not very well documented software implementations rasterization algorithms. In this section we will only talk about the GPU implementation.

2.3.1 Attila

Attila is a complete GPU emulator and simulator written in C++. Its supports DX9, OpenGL and Reyes. The emulator part can generate frames from commercial games and it's very well documented. The simulator part emulates the hardware of the GPU and supports many configurations which is really useful to study the performance of the design[1][4] .

Attila implements recursive and tile scan rasterizations.

2.3.1.1 Attila pipeline

The Attila pipeline has the following components:

Streamer
The streamer takes the input information (vertex and attributes) and sends them to the shaders to process them (vertex shading).

Primitive Assembly
The primitive assembly receives the vertex from the previous step and generates the polygons.

Clipping
The clipping module removes from the pipeline the polygons that are out of the frustrum.

Triangle Setup
The triangle setup builds the edge equations of the polygons.

Fragment Generator
This module traverses the triangle using the edge equations and generates stamps. The algorithms exploits the fragment-inside parallelism

Hierarchical Z
This module discards the fragments overlapped by other fragments.

Z and Stencil test
This module receives fragments in quads (4 fragments), and applies a Z test and a Stencil test to try to discard them.

Interpolator
This module interpolates the fragments with the vertex attributes using a perspective correction interpolation.

Shader
These are vectorial processors with 128 bits registers (4 32 bits floating point numbers). They process the vertex and the fragments with the program defined by the programmer.

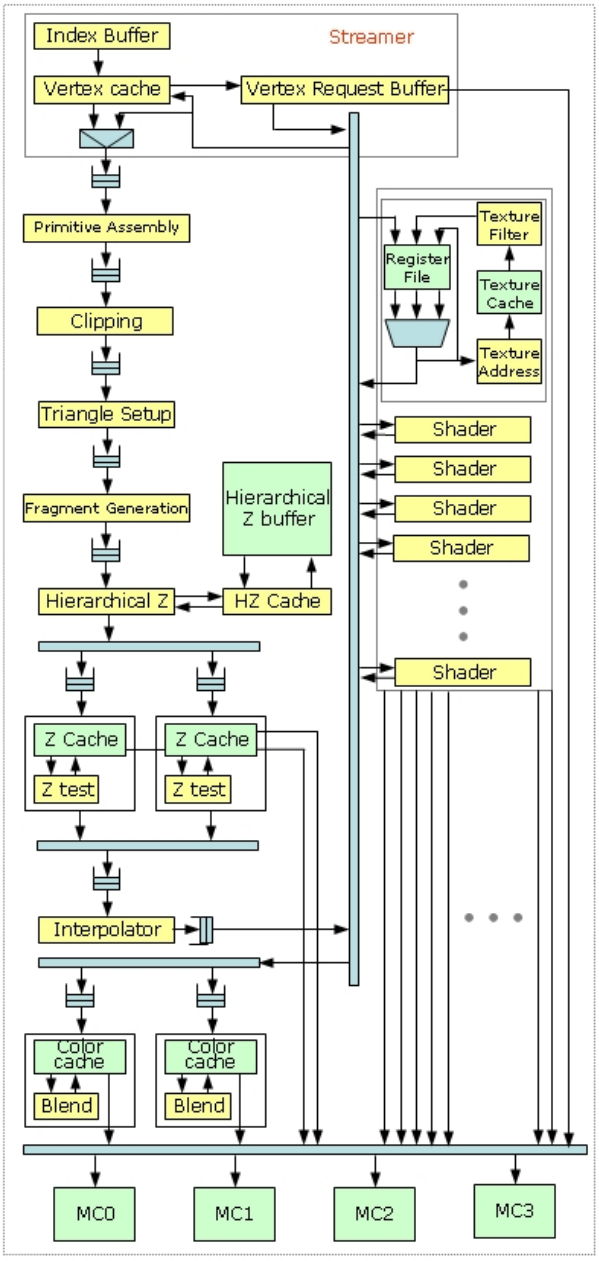

Figure 2.13: Attila pipeline

2.3.1.2 Attila microtriangles pipeline

Attila implements a specialized way to rasterize small triangles described in [10] .The document describes how a single triangle setup starves the pipeline for very small triangles because it produce very few pixels for the high setup cost. To fix this 2 solutions are proposed :

Replicate the triangle setup N times

This one, increase the area usage and doesn't scale very well for a high number of microtriangles since the classic pipeline is still designed to exploit the fragment-inside-triangle parallelism.

Use the shader unit

This one doesn't have any area cost.The triangles are sent to the shader unit if they are too small and they are rasterized there. This solution scales in the future since future *GPU*s will offer more shader cores.

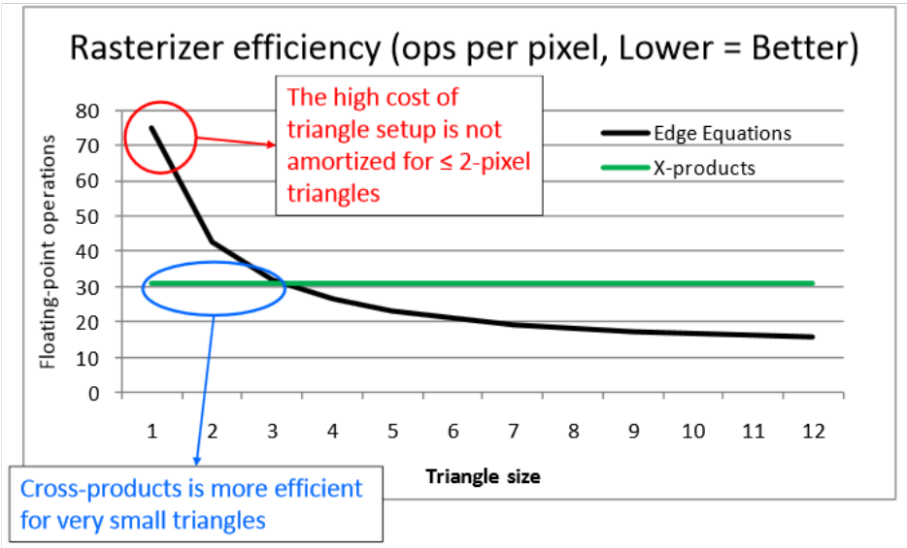

Figure 2.14: Attila rasterizer efficiency plot

Attila implements the second solution. The microtriangles are sent to the shaders, exploiting the triangle parallelism in a efficient way.

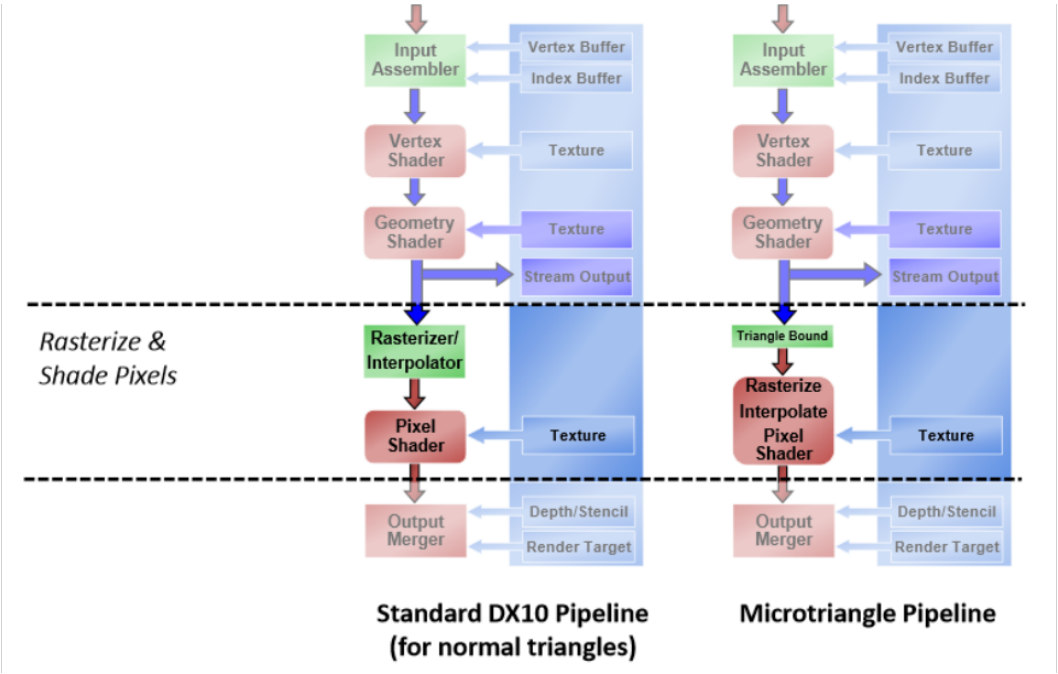

Figure 2.15: Attila microtriangles pipeline

2.3.2 Manticore

Manticore was a project that wants to make a full implementation of a *GPU* in Verilog.Since last year, the project looks dead and only the web page and the code repository remain with a part of the implementation [3].

Chapter 3

Hardware Design

With the documentation about the rasterization stage we can start designing the pipeline. We decide to use the Attila design as a blueprint. We are interested in the Attila fragment generation stage. Also, since Attila have a microtriangle specialized way we can use this design too. Since we are only implementing the rasterization stage we suppose that the input triangles have all the geometric transformations applied. In the following sections we will describe the 2 ways, how they are implemented and how the way selector is made.

3.1 Big Triangles Way

We based this stage in the Attila implementation. This stage tries to exploit the fragment-inside-triangle parallelism of the big triangles. To do this we will use an edge equations based algorithm to reduce the operations for each fragment. Attila implements 2 algorithms : a tile based fragment scanner and a recursive algorithm. We decide to implement the recursive algorithm since it's easier. This stage is composed by 3 main modules : the triangle setup, that builds the edge equations with the triangle vertex, the recursive descents that, using the edge equations, generates fragments using the recursive algorithm. Finally, the fragments are sent to the Interpolator which, using the vertex attributes interpolates the fragments, generation pixels.

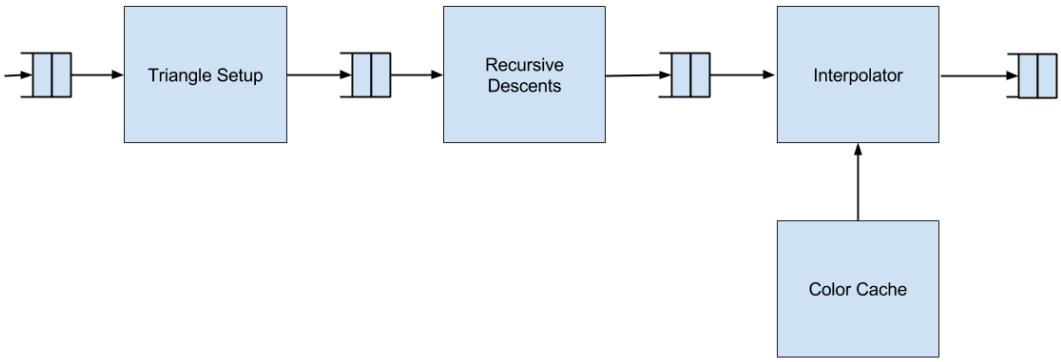

Figure 3.1: Big Triangles Pipeline

3.1.1 Triangle Setup

This module calculates the edge equations values using the adjoin matrix. The implementation of this module follows a shader design.The module receives a 3 triangle vertex and sends it to one of shader units. The module has 2 shaders units to support the high edge equations generation latency (11 intrs.). The implementation of the shader units will be discussed in a future section.

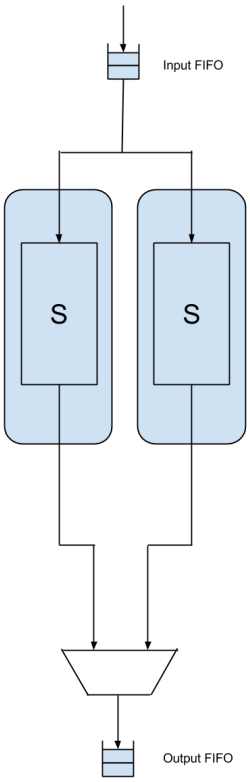

Figure 3.2: Triangle Setup

mul rC.xyz, rX.yzx, rY.zxy
mul rB.xyz, rX.zxy, rW.yzx
mul rA.xyz, rY.yzx, rW.zxy
mad rC.xyz, rX.zxy, rY.yzx, -rC
mad rB.xyz, rX.yzx, rW.zyx, -rB
mad rA.xyz, rY.zxy, rW.yzx, -rA
dp3 rD.x, rC, rW
rcc rI.x, rD.x
mul rC, rC, rI
mul rB, rB, rI
mul rA, rA, rI
	#adj(M)
	
	#det(M)
	#M^{-1}(M)
		rX, rY, rW -> vertex values
		rA, rB, rC -> edge equations values
		rD, rI -> determinant and reciprocal

Figure 3.3: Triangle Setup Program

3.1.2 Recursive Descents

This module generates fragments using the recursive algorithm described previously. The module generates a top-tile using the edge equations and the bounding box of the triangle. This tile is sent to the *Tile evaluator (TE)*s and this ones generates additional tiles. The Recursive descents module balances the work between the *TE*s, sending the tiles to the *TE* with less work to do. Finally the *TE*s generate stamps when the tile is small enough (2x2) and the module sends them to the next module.

The module has 4 *TE* since the tiles are split in 4 sub-tiles. The output rate is 1 stamp/clk since the interpolator only interpolates 1 stamp/clk but the module can produce a lot more.

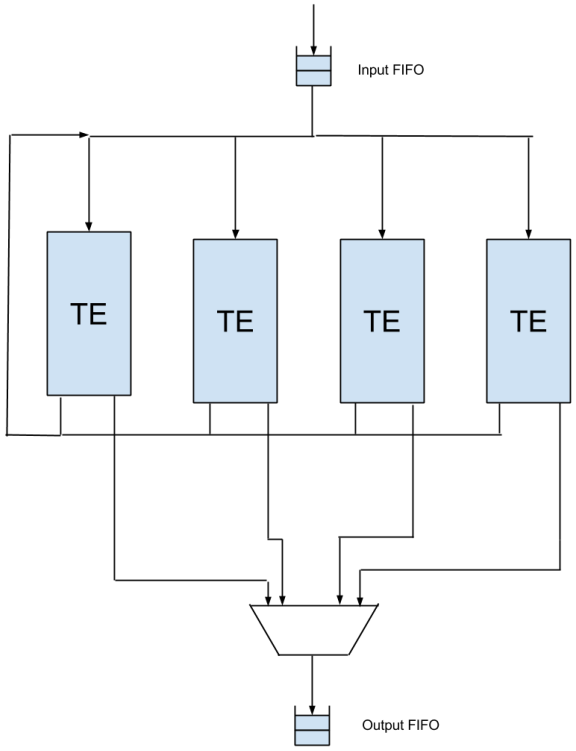

Figure 3.4: Recursive Descent

3.1.2.1 Tile Evaluator (*TE*)

This module splits the tiles from the recursive descents into 4 subtiles and tests if a triangle is inside them. To do it the module takes the 4 subtiles points and tests with the triangle edge equations to check if the subtile contains part of the triangle. If the subtile contains a triangle it's sent it back to the recursive descent. If the tile is small enough and contains a triangle the TE generates a stamp and writes it to the FIFO to send it to the Interpolator. If the subtile doesn't contain any triagle is discarded.

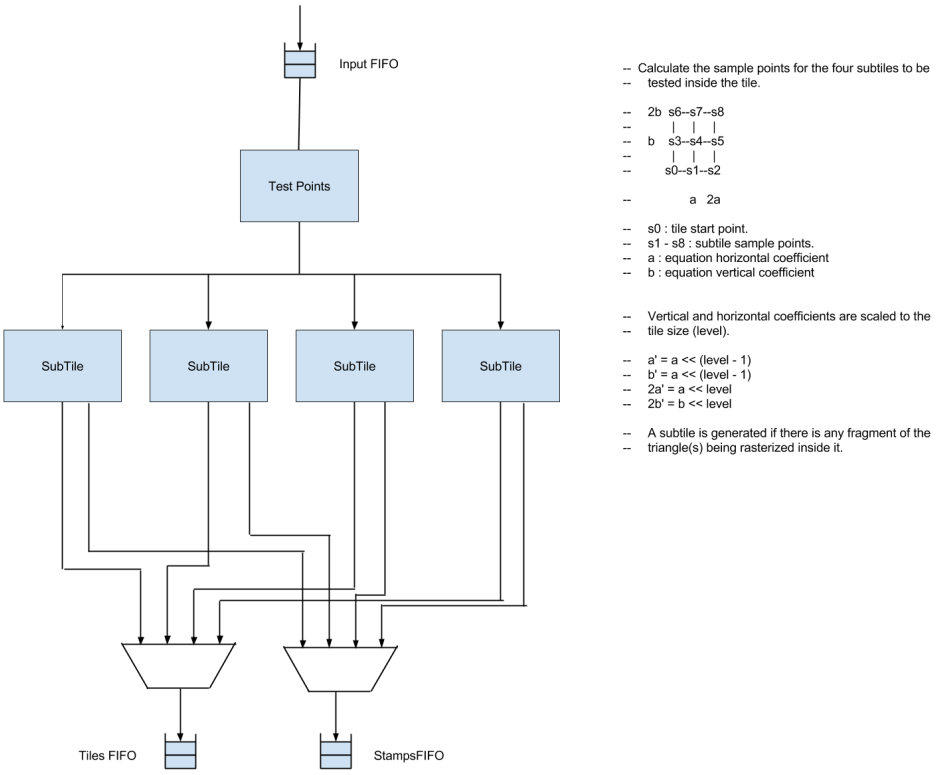

Figure 3.5: TE design

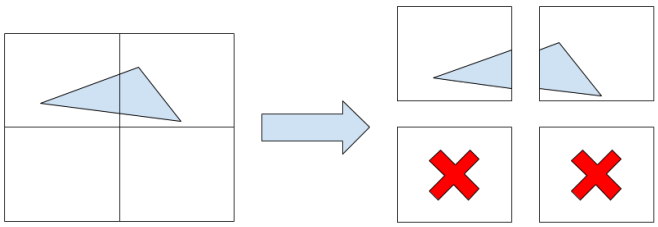

Figure 3.6: TE operation

3.1.3 Interpolation

This module interpolates the fragments. Basically, the module give the final color to the fragment. To do this the module uses barycentric coordinates (McCool) using the following expresion:

$$r = \frac{1}{E_0 + E_1 + E_2}$$

$$F_0 = r * E_0$$

$$F_1 = r * E_1$$

$$F_2 = r * E_2$$

$$color_0 = F_0 * vertex_0.r + F_0 * vertex_1.r + F_2 * vertex_2.r$$

$$color_1 = F_1 * vertex_0.g + F_1 * vertex_1.g + F_2 * vertex_2.g$$

$$color_2 = F_2 * vertex_0.b + F_2 * vertex_1.b + F_2 * vertex_2.b$$

The module obtains the vertex attributes from the color caches and the edge equations values from the previous stage. The module interpolates 1 stamp per clock.

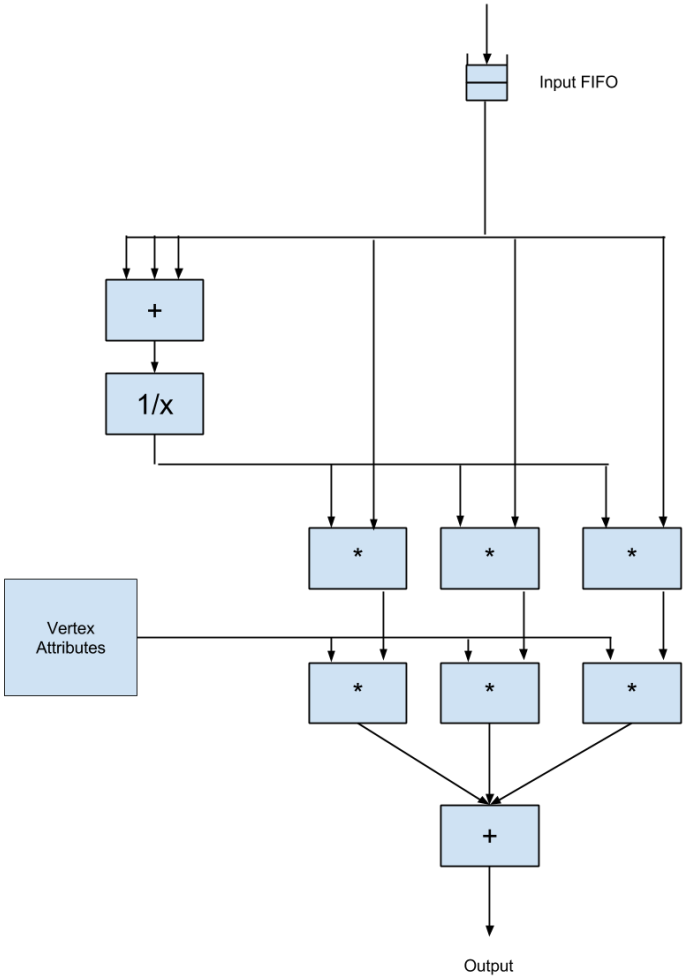

Figure 3.7: Interpolator

3.2 Small Triangles Way

This module rasterizes small triangles using a cross-product algorithm. The module is designed with a shader architecture. The module is composed by "Work Units" which are composed by 4 shader units. When the module receives a triangle it sends it to a free work unit. The work units generate 1 to 4 interpolated fragments for each microtriangle. This module is designed to exploit the triangle-parallelism, rasterizing a lot of triangles at the same time. Every shader unit runs the Attila microtriangles rasterization shader program.

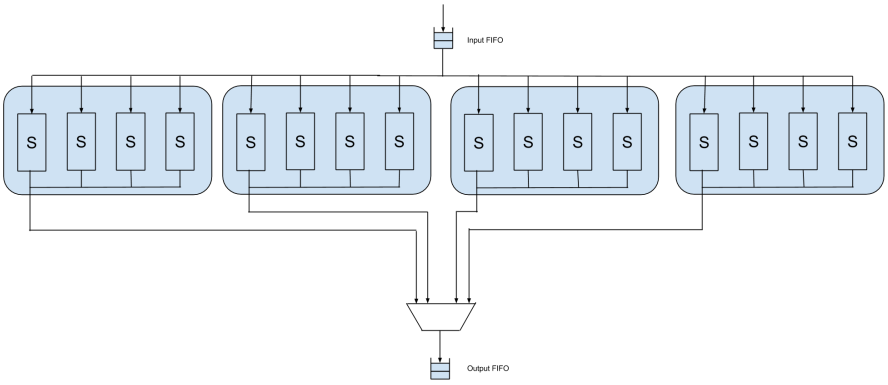

Figure 3.8: Microtriangles Way

a

```
# Compute e0, e1 and e2 directed edges in input vertex order:
a) ADD    t1.yzw,   i0.zzwy, -i0.wwyz;
b) ADD    t3.yzw,   i1.wwyz, -i1.zzwy;

# Compute p0, p1 and p2 vectors and –e1.
c) ADD    t2.xyzw,  i0.yxxx, -i0.zyzw;
d) ADD    t0.xyzw,  i1.yxxx, -i1.zyzw;

# Compute cross products (areal coordinates) and
# face orientation using fixed-point operations.
e) FXMUL   fx0.xyzw, t1.yyzw, t0.xzwy;
f ) FXMAD   t0.xyzw, t3.yyzw, t2.xzwy, fx0.xyzw;

# Test fragment-inside-triangle and cull CW faces.
g) KIL    t0.xyzw;

# Test the tie break rule for edges intersecting the pixel center.
h) CMP_KIL t2.yzw, -t0.yyzw, c255.xxxx, t1.yyzw;
i ) CMP_KIL t2.yzw, -t2.yyzw, c255.xxxx, t3.yyzw;

# Perspective interpolation of z value at the center
j ) DP3 t2.x,     t0.yzww, i2.yzww;
k) RCP t2.x,     t2.x;
l ) MUL t2.x,     t2.x,    t0.x;

# Compute 1/2A for attribute interpolation
m) RCP t0.x,     t0.x

# Output fragment's interpolated Z value.
n) MOV o0.z,     t2.x;
```

b

Shader inputs				
	x	y	z	w
i0	fr.y + 0.5	v2.y	v0.y	v1.y
i1	fr.x + 0.5	v2.x	v0.x	v1.x
i2	-	1/v2.z	1/v0.z	1/v1.z
i3	Microtriangle vertex attributes			
i4				
i5				
...				
Shader constants				
	x	y	z	w
c255	1.0	-	-	-

c

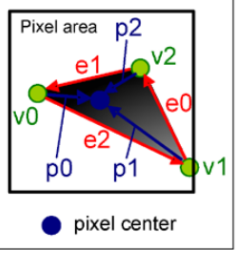

Figure 3.9: Attila microtriangles program

3.3 Shader Unit

The triangle setup and the Microtriangles Way modules are implemented with shader units. Every unit is a vectorial processor with registers of 128 bits. The module is pipelined in 2 steps to get better performance : the Fetch step and the Decode step. In the Fetch step, the next instruction pointed by the program counter is read. In the Decode step, the instruction is decoded and executed. We have only implemented the instructions needed by the 2 programs that we run.

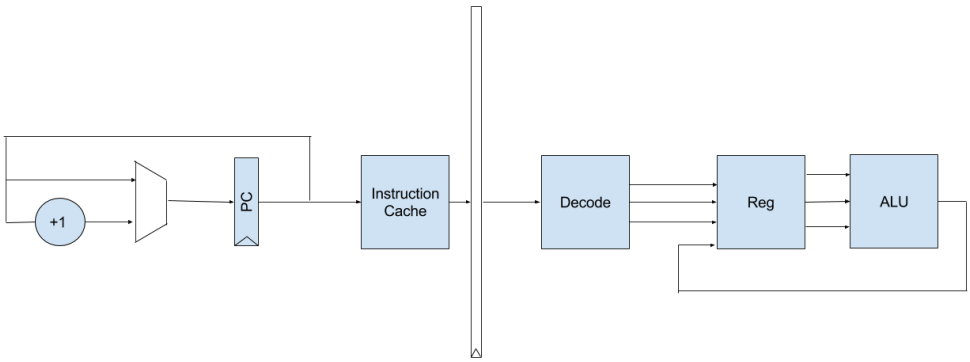

Figure 3.10: Shader Unit

3.4 Way selector

This is the first module of the pipeline. The module receives the vertex and the attributes from the previous stage. The selector computes the bounding box of the triangle, if the size of the triangle is 2x2 or less it is sent to the Microtriangles Way, otherwise it's sent to the triangle setup. If the triangle is sent to the triangle setup, the attributes are stored in the colors cache, otherwise the attributes are sent directly to the microtriangles way.

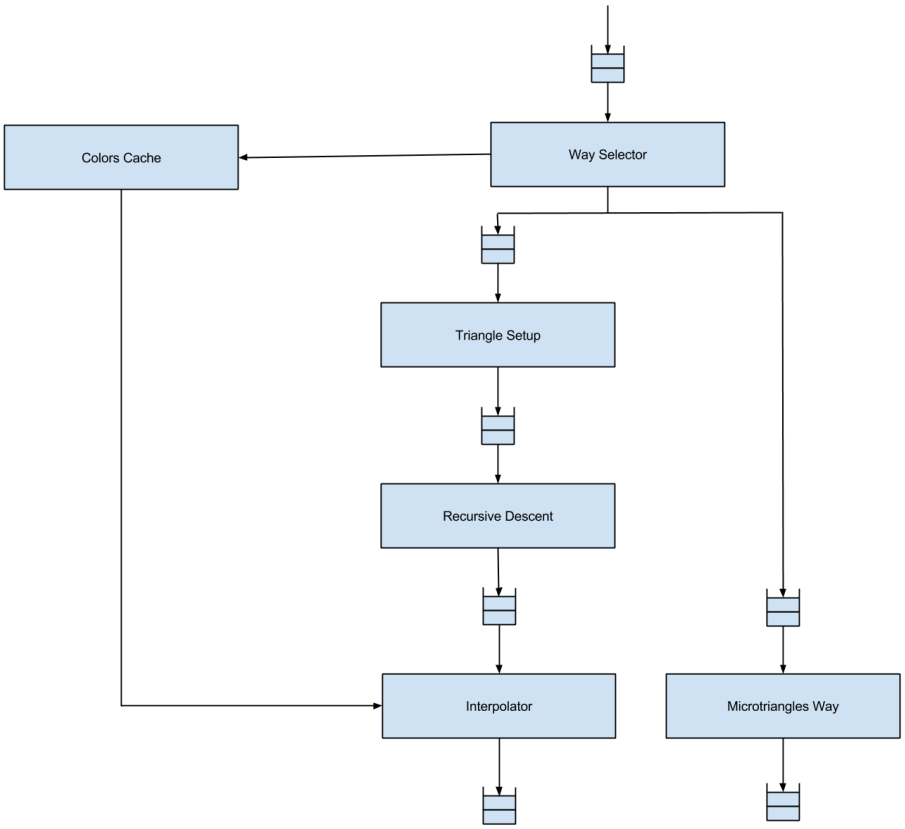

Figure 3.11: Rasterization Pipeline

Chapter 4

Hardware Evaluation

In this section we will explain how we test the pipeline and which tools we used to do it.

Initially, when we were implementing the pipeline we tested it using the simulator, Modelsim. Modelsim simulates every signal of the design, making it easier to see where the errors are. That's good to debug but to test the entire pipeline is not very useful. To test all of the pipeline, we write a module that writes fragments into files in JSON format.

```
[
 {
      "r" : 0.1,  //attributes
      "g" : 0.2,
      "b" : 0.3,
      "x" : 12,  //position
      "y" : 23,
 }

 ...

];
```

Figure 4.1: JSON format

With these files, we can generate an image using GNU Octave to test that the results are what we expected.

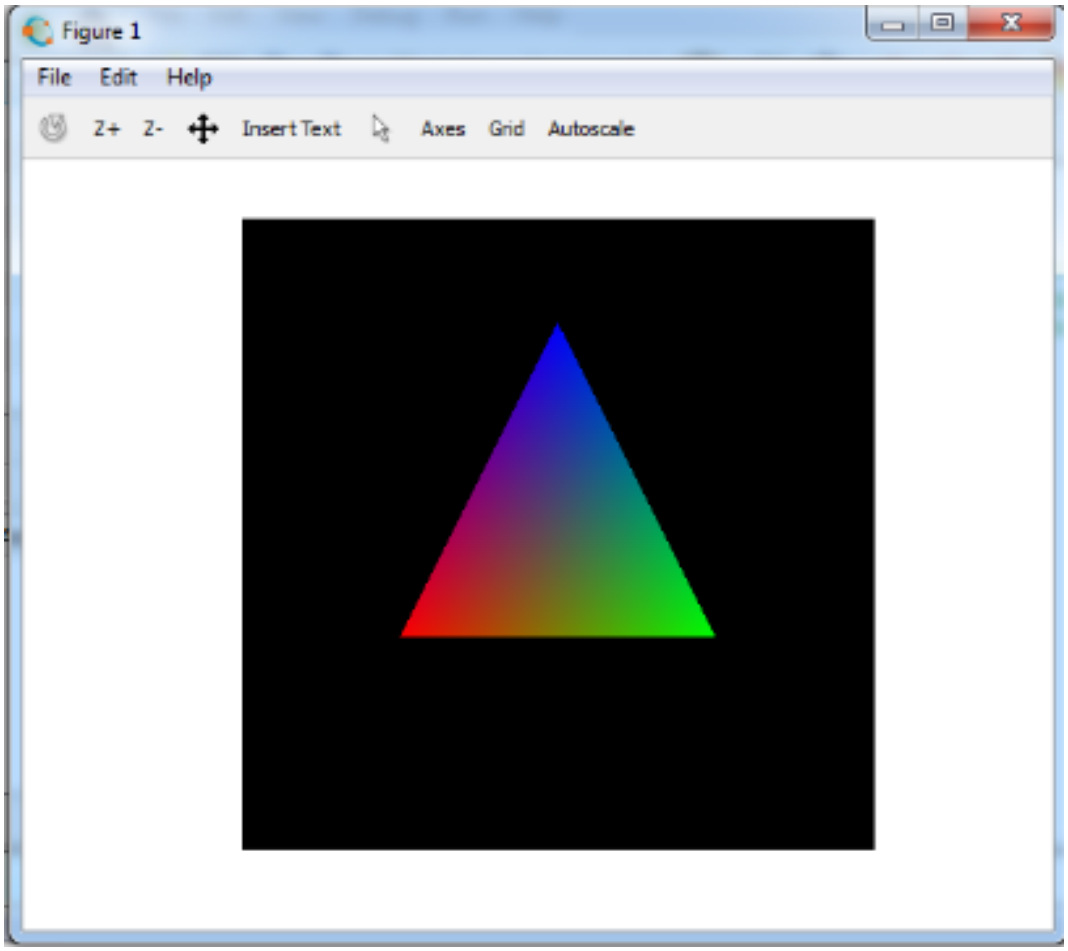

Figure 4.2: Interpolated Triangle - GNU Octave display

When we were ready to test the final design we saw that the design was too big for the *FPGA*!. When we realized, it was too late to get another *FPGA* so we could only test it in the simulator.

The way to test in the FPGA is like in the simulator but with a few changes. We can't write files in the FPGA so, we have to write raw data into the RAM, read it using the software provided by the FPGA manufacturer and use a script to format the data to send it to GNU Octave.The communication with the RAM is not implemented because we can't test it in our FPGA.

Figure 4.3: FPGA Test Protocol

We can't test it in the *FPGA* but as you can see in 4.2 it works in the simulator.

4.1 Specialization improvement

We designed the pipeline to exploit the microtriangles parallelism, by rasterizing a lot of them at the same time. We can see in the figure 4.4, that the pixels/clk when we rasterize microtriangles is much better with the new pipeline that exploits the triangle parallelism.

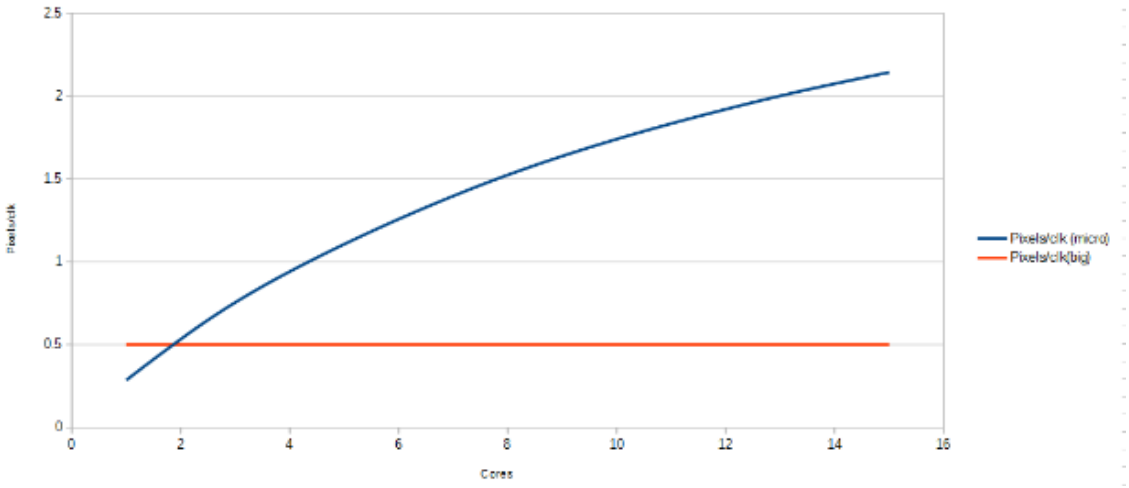

Figure 4.4: Comparative Pixels/clk

Chapter 5

Project Management

5.1 Temporal planning

The project was expected to end June 30th, the limit date to present the project. Complete the project in this time was not possible so the project was extended to October 26th. In the following sections we describe the hours expended in every development stage and which was the planned hours on each of it.

5.1.1 Planning

This is the initial part of the project and it's mostly covered by GEP deliveries. We will spend a month in this part. As you can see in 5.1, we will spend 76.5 hours in this part. This means that we will have a ratio of 2.64 hours/day.

Task	Planned Hours	Total Hours
Scope definition	9,25	9,25
Temporal planning	8,25	8,25
Economic management and sustainability	9,25	9,25
Preliminary presentation	6,25	6,25
Context and biography	15,25	15,25
Specialty	10	10
Initial milestone presentation	18,25	18,25

Table 5.1: Planning phase tasks and dedication hours

This part has been followed without any problem

5.1.2 Project Design

Our objective in this part is make an analysis of the project and design the pipeline. First we will search for documentation about the GPU pipeline and some related projects, if they exist. Once we know how to make a good design, we will do it. Also we will design the hardware to test the pipeline. With this part we want to know how to make a good design and try to avoid possible problems during the implementation phase.

Tasks	Planned Hours	Total Hours
Find and study documentation	40	50
Design the rasterizator	30	50
Design the hardware test	20	25
TOTAL	90	125

Table 5.2: Project design phase tasks and dedication hours

5.1.3 Implementation and validation

The objective of this part is to make a VHDL implementation of the pipeline and the hardware to test it. Once finished, we will validate it and fix all the bugs. This is the hardest and longest part of the project

Tasks	Planned Hours	Total Hours
Implementation of the rasterizator	80	275
Implementation of the hardware test	80	75
Validation and bugfix	50	175
TOTAL	90	525

Table 5.3: Project design phase tasks and dedication hours

5.1.4 Documentation and delivery

In this part the pipeline has been finished and it's working, so we only have to write the documentation of the pipeline and prepare the final presentation.

Tasks	Planned Hours	Total Hours
Make project documentation	22	15
Make presentation documentation	10	21
Practice presentation	7	30
TOTAL	39	66

Table 5.4: Project design phase tasks and dedication hours

5.2 Resources

We can see the resources used during the project in table 5.5.Everyone will we available at any moment of the project.

Resource	*Resource Type*	*Start phase*	*End phase*
Project Manager	Human	Planning	Documentation
Hardware Engineer	Human	Design	Documentation
Laptop	Hardware	Planning	Documentation
FPGA DE2-115	Hardware	Implementation	Implementation
Libreoffice	Software	Planning	Documentation
Quartus II	Software	Implementation	Documentation
Modelsim	Software	Implementation	Documentation

Table 5.5: Resources list

5.3 Estimate time and action plan

In this section we will make a short analysis of the total project time

Phase	**Total Planned hours**	**Planned Duration (in working days)**	**Hours/Day**
Planning	76,50	29.00	2.64
Design	90.00	18.00	5.00
Implementation and validation	210.00	37.00	5,60
Presentation	39.00	13.00	3.00
TOTAL	**415.50**	**102.00**	**4.06**

Table 5.6: Amount of planned hours

Phase	**Total Hours**	**Final Duration (in working days)**	**Hours/Day**
Planning	76,50	29.00	2.64
Design	125.00	25.00	5.00
Implementation and validation	525.00	105.00	5.00
Presentation	66.00	22.00	3.00
TOTAL	**792.50**	**181.00**	**4.37**

Table 5.7: Final amount of hours

As we can see in 5.6, the most critical phase of our project was the implementation and testing phase. This being about the half of the project hours and the higher hours/day ratio of the project.

In the table 5.7 we can see how the project final cost in hours, mostly of them are expended in the Implementation and validation phase.
In case of suffering delays in any part, the only option we have is to increase the hours/day ratio but these was not enough and we have to delay the project.

All resources that will be used in this project, will be available all days of the week, so material availability should not cause any additional delays.

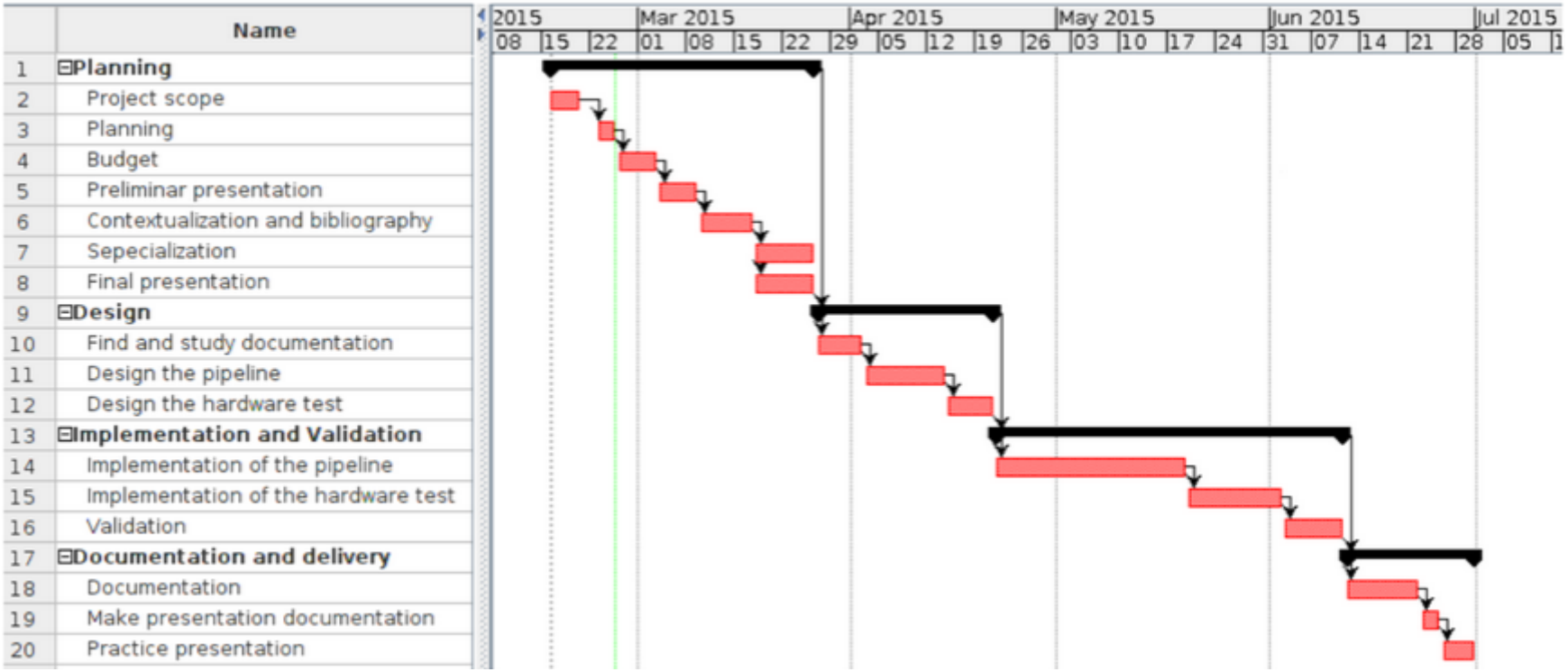

Figure 5.1: Gantt Diagram - Initial planning

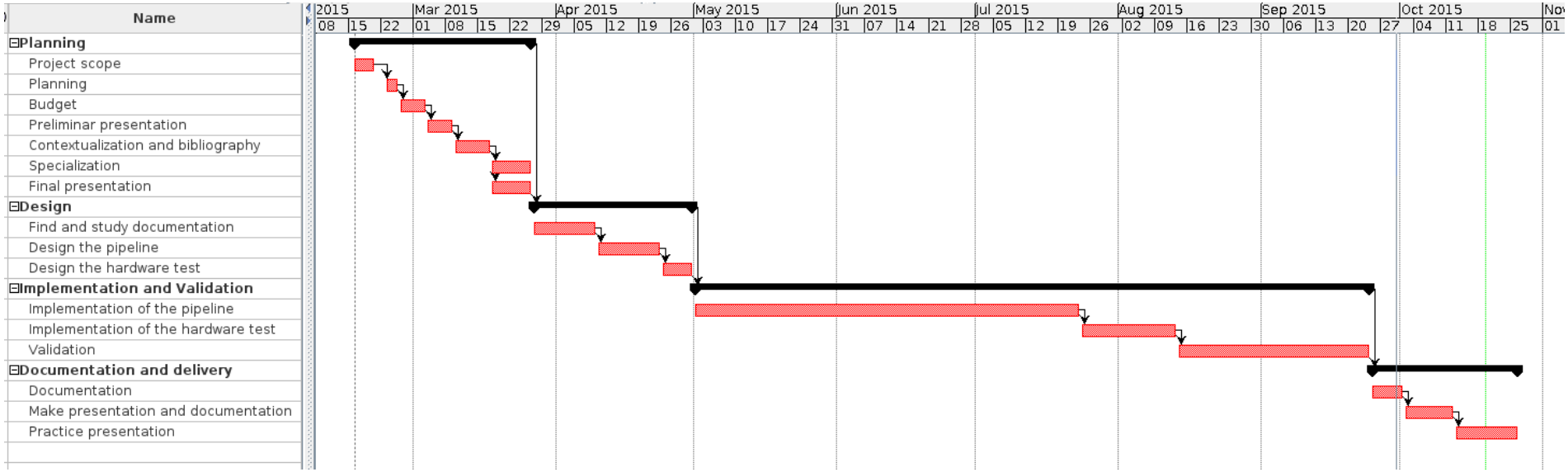

Figure 5.2: Gantt Diagram - Final planning

5.4 Economic Planning

In the following section we will make a budget using the tasks and their duration detailed on the previous section and in the Gantt. The exposed budget is the final cost of the project. In the 5.4.5 section we compare the planned costs with the final costs.

5.4.1 Planning

In this section we will detail the costs of the planning part. The costs of this part are mostly generated by human resources.

Direct Cost							
Resource	Resource type	Units	Price	Time	Depreciation	Total	Observations
Project Manager	Human	1	25€/h	76.5h		1912.50€	
Laptop	Hardware	1	500.00€		0.02%	10.00€	4 years of life span 1 month of usage in this stage
Libreoffice	Software	1	0.00€			0.00€	
Indirect Cost							
Resource	Resource type	Units	Price	Time	Depreciation	Total	Observations
Electricy	Energy	24.48 kwh	0.126€/Kwh	76.5h		3.08 €	300w laptop 20w light
TOTAL						1925.58€	

Table 5.8: Planning costs

5.4.2 Project design

In this section we will detail the costs of the project design part. Like in the previous part the costs of this part are mostly generated by human resources.

Direct Cost							
Resource	Resource type	Units	Price	Time	Depreciation	Total	Observations
Project Manager	Human	1	25€/h	30h		750.00€	
Hardware Engineer	Human	1	20€/h	95h		1900.00€	
Laptop	Hardware	1	500.00€		0.02%	10.00€	4 years of life span 1 month of usage in this stage
Libreoffice	Software	1	0.00€			0.00€	
Indirect Cost							
Resource	Resource type	Units	Price	Time	Depreciation	Total	Observations
Electricy	Energy	38.4 kwh	0.126€/Kwh	120h		4.83€	300w laptop 20w light
TOTAL						2664.83€	

Table 5.9: Design Costs

5.4.3 Implementation and validation

In this section we will detail the costs of the implementation and validation part. Like in the previous part the costs of this part are mostly generated by human resources. This is the hardest and longest part because, as you can see in the Possible Problems section of the table, there's likely to be bugs in the pipeline. These have to be solved with more work hours, which means more money and a possible change in the planning.

Direct Costs							
Resource	Resource type	Units	Price	Time	Depreciation	Total	Observations
Project Manager	Human	1	25€/h	10h		250.00€	
Hardware Engineer	Human	1	20€/h	375h		7500.00€	
Hardware Enginner for Testing	Human	1	18€/h	175h		3115.00€	
Laptop	Hardware	1	500€		0.06%	30.00€	4 years of life span. 3 month of usage in this stage.
FPGA DE2-115	Hardware	1	236.92€		0.0125%	3.60€	5 years of life span. 1 month of usage in this stage.
Quartus II	Software	1	0€			0.00€	
Indirect Costs							
Resource	Resource type	Units	Price	Time	Depreciation	Total	Observations
Electricy	Energy	156.24 kwh	0.126€/Kwh	525h		24.30€	300w laptop 24w FPGA 20w light
TOTAL						10922.90€	
Possible Problems							
Problem		Probability		Cost		Observations	
Pipeline Bugs		40%		1600.00€			
FPGA limitations		30%		500.00€		We need to buy another FPGA	

Table 5.10: Implementation and validation cost

5.4.4 Documentation and delivery

In this section we will detail the costs of the documentation and delivery part. This is the final part of our project and all the hard work is done. It's unlikely we'll have any problems here.

Direct Costs							
Resource	Resource type	Units	Price	Time	Depreciation	Total	Observations
Project Manager	Human	1	25€/h	66h		1650.00€	
Laptop	Hardware	1	500.00€		0.02%	10€	4 years of life span 1 month of usage in this stage
Quartus II	Software	1	0.00€			0.00€	
Libreoffice	Software	1	0.00€			0.00€	
Indirect Costs							
Resource	Resource type	Units	Price	Time	Depreciation	Total	Observations
Electricy	Energy	21.44 kwh	0.126€/Kwh	66h		2.70€	300w laptop 20w light
TOTAL						1662.70€	

Table 5.11: Documentation and delivery cost

5.4.5 Total Cost

In this section you can see the total cost of the project. As we can see we added a 10% of contingencies that we think are enough to deal with possible problems. Prices are rounded.

Phase	Planned Cost	Total cost
Planning	1925.60€	1925.60€
Design	2564.80€	2664.83€
Implementation and validation	4386.20€	10922.90€
Documentation and delivery	986.60€	1662.70€
Total without Contingency	**9863.20€**	**17176.03€**
Total with Contingency (10%)	**10849.52€**	**18863.63€**

Table 5.12: Total costs

As we can see in table 5.12 the final cost of the project is 18863.63€which means we need 8014.11 € (Mostly expended in the Implementation and validation phase) which is not covered by the planned contingency.

5.5 Sustainability and Viability

Economic	Social	Environment
7	7	9

Table 5.13: Sustainability matrix results

5.5.1 Economic

The budget of the project is pretty detailed and assesses many situations that the budget can increase. The budget is a bit expensive because the developers don't have much experience developing this type of projects.

The economic viability of this project is justified by the following reasons:

- The budget is pretty detailed and human and material resources are assessed.
- The budget is slightly expensive but is still competitive
- The time spent in each task is in accordance to its importance

5.5.2 Social

The political situation is not the best to make projects, but the *GPU* market has been rising in the last years. *GPU*s help improve the speed of modern computers, as well as contributing to many other things (Maths, Chemistry, Medicine...) that use computers to improve themselves.

The social viability of this project is justified by the following reasons:

- The political situation is not the best to make projects, but the *GPU* market has been rising in the last years.
- Therese not any law that regulates the *FPGA*s or *GPU*s
- The project helps students and researchers to study the *GPU* rasterization stage.
- The project doesn't harm any collective.

5.5.3 Environment

*GPU*s are devices that are made to do a lot of operations in parallel, thus making large problems faster to solve.Which in turns reduce the time that we are consuming energy. Since the goal of our project is to make a stage of a *GPU* and study it we are contributing to improve a device that is reducing the environmental impact of computers.

The environmental viability of this project is justified by the following reasons:

- The development of the project doesn't produce any toxic material.
- The environmental impact is positive since we are reducing the energy that computer consume.
- The resulting project is reusable by other projects (ex: Full *GPU* implementation)
- Time spent in each task is in accordance to its importance

Chapter 6

Conclusions

Finally we are going to expose the conclusion of the project results : The main objective was to implement a *GPU* rasterization stage on the FPGA. This objective is not fully achieved, the design works on the simulator but we can't test it on the *FPGA* because the design can't fit the *FPGA*.

The other objective was to improve the classic GPU pipeline adding microtriangles in a specialized way that exploits the triangle parallelism. This objective is fully accomplished as we exposed in the Hardware Evaluation section.
Despite the objectives, the project development process has been very helpful in my training.

First, I have learn a lot of about designing a hardware with special needs like this one. Also ,although the design doesn't fit the *FPGA* and finally we can't test it I have learn a lot about *FPGA*, how they work and how to use it correctly.

6.1 Other works

In this project we designed and implemented a GPU rasterization stage. Future works can continue to improve this device or by attaching more modules of the GPU.

- Tests in a FPGA.
- Improve the pipeline latency and bandwidth.
- Improve the resource usage of the implementation.
- Attach more components of the GPU to the design.
- Improve the shaders design.

Acknowledgements

As a final note I just want to thank all of those people who made this project possible and successful.

To Agustín, my director, for providing me this opportunity.

To my mother, who is always there.

To my family, without them I can't be here today.

To my childhood friends, for be on my side when I need them.

To my friends Artem, iVan and Oscar for help me with the mercenaries when the work's is too much.

To Elsa, Anna and Lucia, for make me laugh when I really need.

To Javi, for so many good moments with The Game.

Bibliography

[1] Attila Project - AttilaWiki. URL http://attila.ac.upc.edu/wiki/index.php/Attila_Project

[2] Rasterization on Larrabee - Intel Developer Zone. URL https://software.intel.com/en-us/articles/rasterization-on-larrabee.

[3] Manticore. URL https://webcache.googleusercontent.com/search?q=cache:GOW2wGYgANYJ

[4] Del Barrio, V Moya, C González, J Roca, Fernández A, and Espasa Roger. {ATTILA}: A Cycle-Level Execution-Driven Simulator for Modern {GPU} Architectures. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 231–241, 2006. ISBN 1-4244-0186-0. doi: 10.1109/ISPASS.2006.1620807. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1620807.

[5] Kayvon Fatahalian, Edward Luong, Solomon Boulos, Kurt Akeley, William R Mark, and Pat Hanrahan. Data-parallel rasterization of micropolygons with defocus and motion blur. *Proceedings of the 1st ACM conference on High Performance Graphics HPG 09*, 1:59, 2009. doi: 10.1145/1572769.1572780. URL http://portal.acm.org/citation.cfm?doid=1572769.1572780.

[6] Michael D. McCool, Chris Wales, and Kevin Moule. Incremental and hierarchical Hilbert order edge equation polygon rasterization. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware - HWWS '01*, pages 65–72, 2001. URL http://portal.acm.org/citation.cfm?doid=383507.383528.

[7] Joel McCormack and Robert McNamara. Tiled polygon traversal using half-plane edge functions. *SIGGRAPH/EUROGRAPHICS Conference On Graphics Hardware*, 2000.

[8] Marc Olano and Trey Greer. Triangle scan conversion using 2D homogeneous coordinates. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware - HWWS '97*, pages 89–95, 1997. URL http://attila.ac.upc.edu/wiki/index.php/Main_Page.

[9] Juan Pineda. A parallel algorithm for polygon rasterization, 1988. ISSN 00978930.

[10] Jordi Roca, Victor Moya, Carlos Gonzalez, Vicente Escandell, Albert Murciego, Agustin Fernandez, and Roger Espasa. A SIMD-efficient 14 instruction shader program for high-throughput microtriangle rasterization. *The Visual Computer*, 26(6-8):707–719, apr 2010. ISSN 0178-2789. doi: 10.1007/s00371-010-0492-4. URL http://link.springer.com/10.1007/s00371-010-0492-4.