FPGA-based 3D Graphics Pipeline and Advanced Rendering Effects

Diploma Thesis Report

by **Fotis Pegios**

Advisor: Prof. Aikaterini Mania
Co-Advisors: Prof. Ioannis Papaefstathiou and
Prof. Apostolos Dollas

Electronic and Computer Engineering Department
Technical University of Crete
Greece
February 17, 2016

Abstract

Since their invention by Xilinx in 1984, FPGAs have gone from being simple glue logic chips to actually replacing custom application-specific integrated circuits (ASICs) and processors for signal processing and real-time applications. As they enhance in speed, size and abilities, FPGAs are becoming more useful in areas where ASICs were used before. 3D graphics rendering is one such area, where research is underway as to how FPGAs can help to improve the performance of graphics processing units with less energy consumption.

In this project we represent an FPGA-based Graphic Processor for low power applications and three advanced 3D rendering effects, implemented using a hardware description language (VHDL). Approaching such a project, we have the ambition to fully understand how a simple GPU really works in depth and how graphics algorithms are implemented on low-level programming language. On the grounds that this thesis will be available to the programming society, there is promise for future upgrade and expansion.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisor Prof. Aikaterini Mania for the continuous support of my undergraduate studies, for her patience, motivation, enthusiasm, and immense knowledge. Her guidance helped me in all the time of developing and writing of this thesis. Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Ioannis Papaefstathiou and Apostolas Dollas for their encouragement and accurate comments to my work.

Also, I would like to thank Dr. George-Alex Koulieris and PhD candidate Pavlos Malakonakis for the continuous guidance, help and knowledge they gave me during the development of this thesis. I wish them all the best for their future careers.

Special thanks to Mr. Markos Kimionis and the Microprocessor and Hardware Lab of the University, which gave me the the best equipment to develop this project.

I thank all my friends and classmates for their continuous interest and encouragement during all the years of my studies.

I am also grateful to Mr. Mike Field from New Zealand, a good person and an FPGA expert, who helped me deal with a great difficulty I encountered and took me several days to overcome.

Finally, I thank my parents for supporting me throughout my studies at University, providing me all the conveniences in order to complete my undergraduate degree successfully.

Contents

List of Figures

List of Acronyms

ASIC Application Specific Integrated Circuits

FPGA Field Programmable Gate Array

GPU Graphics Processing Unit

HDL Hardware Description Language

Microblaze A soft-core 32 bit RISC microprocessor designed specifically for Xilnix FPGAs

Pipeline A sequence of functional units which performs task in several steps

Pixel Contraction of Picture element

RAM Random Access Memory

RTL Register Transfer Level

Vertex The point of intersection of lines

VHDL Very High Speed Integrated Circuit (VHSIC) Hardware Description Language

Xilinx Company invented the FPGA

2D Two Dimensional

3D Three Dimensional

1 Introduction

While the popularity of mobile devices and embedded systems that use 3D animations and graphics is escalating, it makes sense to look for new ideas that can help and improve their evolution and better operation. In contrast with Application-Specific Integrated Circuits (aka ASICs) that the majority of modern devices use, FPGAs have the capability to be reprogrammed over the time. With FPGAs, developers are able to improve or modify the initial design depending on their current requirements at any time and without any cost.

Apart from that, handheld systems use batteries in order to operate effectively. If the internal graphics processor of any portable device requires great amounts of energy, many issues can be faced due to the power management. On the other hand, FPGAs have little energy consumption when they operate. This means that FPGA-based graphics processors could be a solution to encounter such issues.

The main goal of this project was to implement a graphics processor unit on an FPGA. Apart from the processor, three advanced 3D rendering effects were implemented too, in order to demonstrate the capabilities of this system. Implemented entirely in Hardware Description Language (HDL) without the use of any processor, this system is open and configurable, providing a graphics system that could be easily adapted or optimized for specific requirements. The implementation was based on a tutorial series written by David Rousset and called "Learning how to write a 3D soft engine from scratch in `C#`, TypeScript or JavaScript"[27].

To achieve the goal of this project, three primary objectives had to be met. The graphics processor should be able to:

- Render any object depending on the coordinates imported
- Generate the final image at reasonable speeds
- Provide a display output interface

These objective were completed through the design and implementation of various necessary components. In this project we did not focus so much on the performance of the system, but on its successful operation. Thus, several optimizations can be done in the future to enhance the current performance.

The remainder of this report provides the reader with some background research in the relevant areas of graphics processing and algorithms, FPGA technology and High-level Synthesis (chapter 2), chapter 3 presents some related works, chapter 4 then describes an overview of the project including the goals of this project, the high level design, the target device and software platform were used for the implementation. Chapter 5 includes the hardware implementation, chapter 6 describes the evaluation of the system after performing several tests, and finally chapter 7 contains the conclusions we drew about the process we used as well as our recommendations for future work.

2 Background Research

Before proceeding to the design and implementation, it is vital to give the reader all the necessary mathematical background information about the graphics processing and the concept we used in this project. This section discusses the computer graphics pipeline, the rendering effects we implemented, an introduction to FPGAs and some information about High-Level synthesis.

2.1 Graphics Rendering Pipeline

In 3D computer graphics, the graphics pipeline or rendering pipeline refers to the sequence of steps used to create a 2D raster representation of a 3D scene[10]. Plainly speaking, once a 3D model has been created, for instance in a video game or any other 3D computer animation, the graphics pipeline is the process of turning that 3D model into what the computer displays. A complete rendering pipeline consists of these stages: Vertex Processing, Primitive Assembly, Clipping, Rasterization, Z-buffering, Shading and Texture Mapping.

It is called a pipeline because vertices are put through each step one at a time, and at each step along the "pipe", the vertex is rendered into its image form.

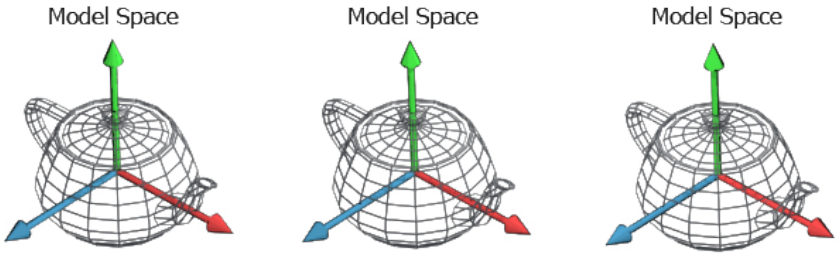


Figure 1: Three teapots each one in its own model space

2.1.1 Vertex Processing

In 3D graphics processing, a vertex is typically specified in the form of an (x, y, z) triple as a discrete position within the 3D coordinate system. These are often accompanied by many other parameters and vectors, representing

such data as shading color, texture, coordinates or normal direction. Vertex processing is the responsible process for decoding this data and preparing them to be assembled as primitives and then rasterized. The vertex positions must be transformed from their 3D coordinates into 2D space as approximated by a display device for this to be done.

When a model first starts out, it is generally centered on the origin, meaning the center of the object is at (0, 0, 0). This also means that all the objects in a scene will be in the exact center of the world, all piled up on top of each other, which does not make a very excellent scene. The first step is to sort out the new positions of all the models in relation to each other. This is called **World Transformation**.

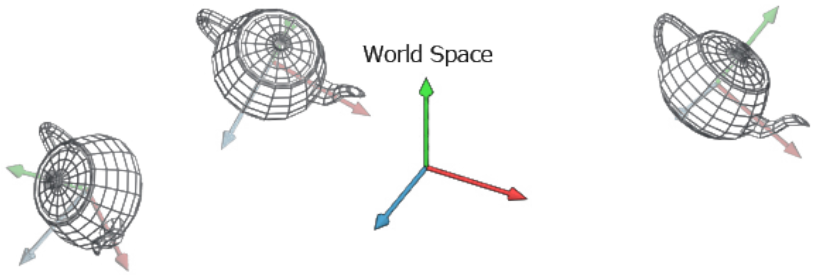


Figure 2: Three teapots set in World Space

Once all the objects in the world have been sorted out and the vertices' positions have been altered to represent world coordinates, we have to transform them into the View space. In graphics we usually use a camera analogy, i.e. the viewer, located at the View Reference Point (VRP), observes the scene through a camera and can move around the scene. This is accomplished by defining a viewing coordinate system (VCS) which has the position and orientation of the camera. Then we align the VCS with the world coordinate system (WCS), creating a viewing transformation matrix. This matrix is then applied to all the objects in the scene which moves them into the proper position as seen from the VRP. This process is known as **View Transformation**.

After the coordinate system has been rearanged, then it is time to convert all the 3D models into 2D images. In this step, the 3D coordinates are converted into screen coordinates. This process is known as **Projection Transformation**[18].

World Transformation World Transformation, in essence, changes coordinates from model space to world space. In other words, it places a model in a world at an exact point defined by coordinates.

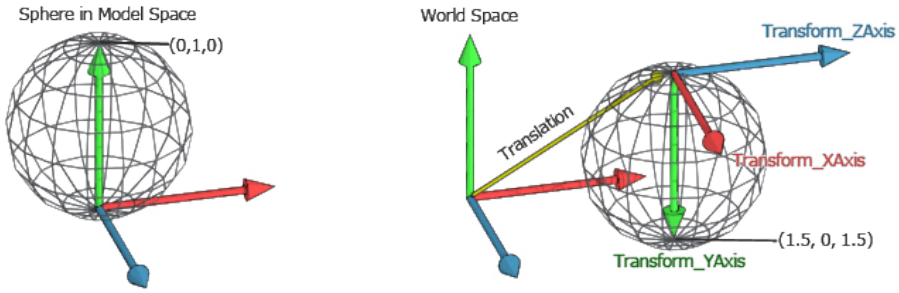


Figure 3: Sphere transformed from model space to world space

In order to apply the transformation, we have to multiply all the vertices that we want to transform against the transformation matrix. In Figure 4, we can see how we represent a generic transformation in matrix form. Where $Transform_X Axis$ is the X axis orientation in the new space, $Transform_Y Axis$ is the Y axis orientation in the new space, $Transform_Z Axis$ is the Z axis orientation in the new space and Translation describes the position where the new space is going to be relatively to the current space.

$$\begin{bmatrix} Transform\_XAxis.x & Transform\_YAxis.x & Transform\_ZAxis.x & Translation.x \\ Transform\_XAxis.y & Transform\_YAxis.y & Transform\_ZAxis.y & Translation.y \\ Transform\_XAxis.z & Transform\_YAxis.z & Transform\_ZAxis.z & Translation.z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 4: Generic World Transformation Matrix

Sometimes we want to do simple transformation, like translations, rotations or scalings.

- Translation refers to the movement of a vertex along a coordinate system axis.
- Rotation is the process of spinning 3D objects along an axis. Like translation, it can be done along multiple axes simultaneously, allowing the user to position the model as desired.

- Scaling is the action of making a 3D object larger or smaller. When an object is scaled, each vertex in the object is multiplied by a given number. These numbers can be different for each axis, resulting in various stretching effects.

In these cases we may use the following matrices which are special cases of the generic form we have just presented.

Translation Matrix In Figure 5, Translation is a 3D vector that represent the position where we want to move our space to. A translation matrix leaves all the axis rotated exactly as the active space.

$$\begin{bmatrix} 1 & 0 & 0 & Translation.x \\ 0 & 1 & 0 & Translation.y \\ 0 & 0 & 1 & Translation.z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 5: Translation Matrix form

Scale Matrix In Figure 6, Scale is a 3D vector that represent the scale along each axis. If you read the first column you can see how the new X axis it is still facing the same direction but it is scaled by the scalar Scale.x. The same happens to all the other axis as well. Also we can notice how the translation column is all zeros, which means no translation is required.

$$\begin{bmatrix} Scale.x & 0 & 0 & 0 \\ 0 & Scale.y & 0 & 0 \\ 0 & 0 & Scale.z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 6: Scale Matrix form

Rotation Matrix around X Axis In Figure 7, θ is the angle we want to use for our rotation. Notice how the first column will never change, which is expected since we are rotating around X axis. Also notice how change theta to 90° remaps the Y axis into Z axis and the Z axis into Y axis.

The rotation matrices for the Z axis and the Y axis behave in the same way of the X axis matrix.

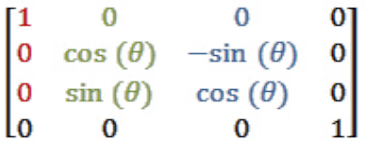


Figure 7: Rotation Matrix form around X Axis

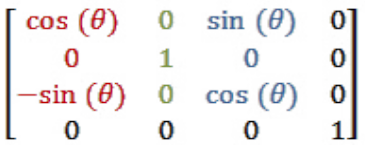


Figure 8: Rotation Matrix form around Y Axis

All the matrices we have just presented are the most used ones we need to describe rigid transformations. We can chain several transformations together by multiplying matrices one after the other. The result will be a single matrix that encodes the full transformation and it is called **World Matrix**.

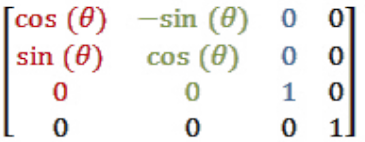


Figure 9: Rotation Matrix form around Z Axis

View Transformation With all the objects at the right place we now need to project them to the screen. This is usually done in two steps. The first step moves all the object in another space called the View Space. The second step performs the actual projection using the projection matrix. This last step is a bit different from the others and we will see it in detail in next.

The View Space is an auxiliary space that we use to simplify the math and keep everything elegant and encoded into matrices. The idea is that we need to render to a virtual camera, which implies projecting all the vertices onto the camera screen that can be arbitrarily oriented in space. The math simplifies a lot if we could have the camera centered in the origin and watch-

ing down one of the three axis, let's say the Z axis to stick to the convention.

So we create a space that remaps the World Space so that the camera is in the origin and looks down along the Z axis. This space is the View Space (sometimes called Camera Space) and the transformation we apply moves all the vertices from World Space to View Space.

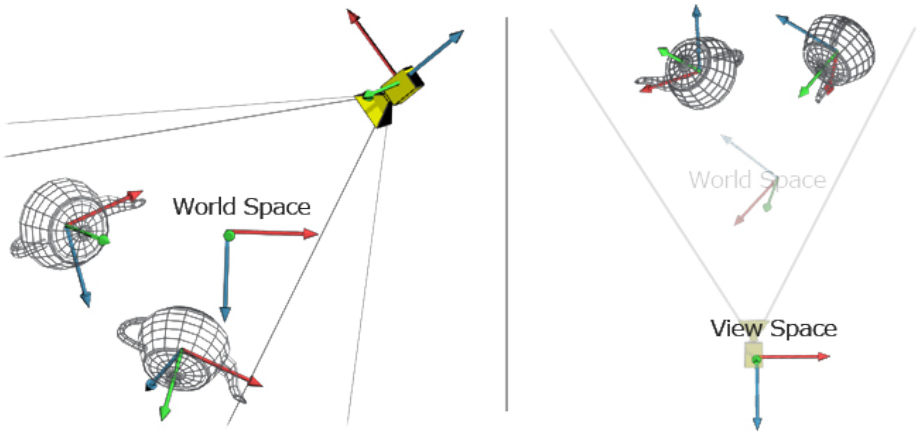


Figure 10: On the Left two teapots and a camera in World Space; On the right everything is transformed into View Space (World Space is represented only to help visualize the transformation)

Now, if we imagine we want to put the camera in World Space we would use a transformation matrix (here **View matrix**) that is located where the camera is and is oriented so that the Z axis is looking to the camera target. The inverse of this transformation, if applied to all the objects in World Space, would move the entire world into View Space. Notice that we can combine the two transformations Model To World and World to View into a single transformation Model To View.

View Matrix A common method to derive the view matrix is to compute a **Look-at matrix** given[19]

- The position of the camera in world space (usually referred to as the eye position)
- The current world's up-direction: usually [0, 1, 0]
- The camera look-at target: the origin [0, 0, 0]

A software typical implementation of Look-at function which exists in Babylon.math.js library looks like this:

Listing 1: Look-at function by Math.babylon.js

```
Matrix.LookAtLH = function LookAtLH(eye, target, up) {
           var zAxis = target.subtract(eye);
           zAxis.normalize();
           var xAxis = Vector3.Cross(up, zAxis);
           xAxis.normalize();
           var yAxis = Vector3.Cross(zAxis, xAxis);
           yAxis.normalize();
           var ex = -Vector3.Dot(xAxis, eye);
           var ey = -Vector3.Dot(yAxis, eye);
           var ez = -Vector3.Dot(zAxis, eye);
           return Matrix.FromValues(xAxis.x, yAxis.x, zAxis.x, 0,
               xAxis.y, yAxis.y, zAxis.y, 0, xAxis.z, yAxis.z,
               zAxis.z, 0, ex, ey, ez, 1);
       };
```

Projection Transformation If view transformation can be thought of as a camera, then this step can be thought of as a lens. Once view transformation is completed, we have to change the 3D scene into one big 2D image so it can be drawn on the screen. The process that does this is called projection transformation. It is simply the converting of 3D coordinates to screen coordinates and moving from View space to Projection Space.

To go from the View Space into the Projection Space we need another matrix, the Projection matrix, and the values of this matrix depend on what type of projection we want to perform. The two most used projections are the Orthographic Projection and the Perspective Projection.

In Orthographic projection, projection space is a cuboid which dimensions are between -1 and 1 for every axis. This space is very handy for clipping (anything outside the 1:-1 range is outside the camera view area) and simplifies the flattening operation (we just need to drop the z value to get a flat image). To do the Orthographic projection we have to define the size of the area that the camera can see. This is usually defined with a width and height values for the x and y axis, and a near and far z values for the z axis. Given these values we can create the transformation matrix that remaps the box area into the cuboid.

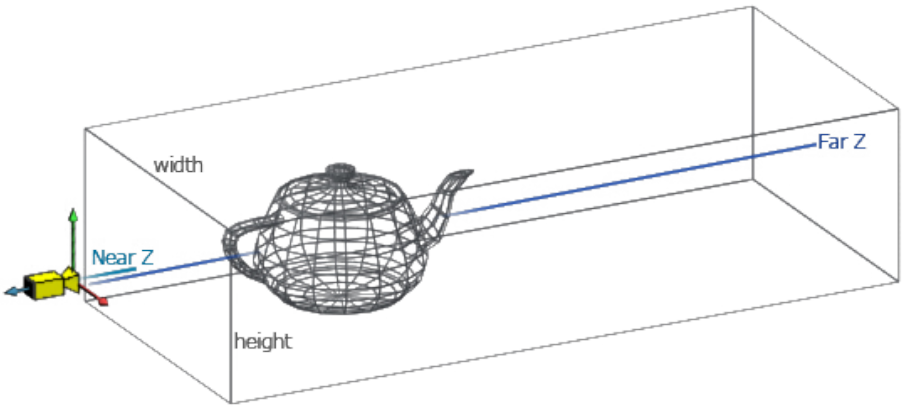


Figure 11: Orthographic Projection

The matrix that follows is used to transform vectors from View Space into Ortho-Projected Space and assumes a right handed coordinates system.

$$\begin{bmatrix} \frac{1}{width} & 0 & 0 & 0 \\ 0 & \frac{1}{height} & 0 & 0 \\ 0 & 0 & -\frac{2}{Z_{far} - Z_{near}} & -\frac{Z_{far} + Z_{near}}{Z_{far} - Z_{near}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 12: Ortho-Projection Matrix

The other projection is the Perspective projection. The idea is similar to the orthographic projection, but this time the view area is a frustum and therefore it is a bit more tricky to remap. Before proceeding to perspective projection, we need to discuss about the homogeneous coordinates and projective Geometry[20].

Most of the time when working with 3D, we are thinking in terms of Euclidean geometry (that is, coordinates in three-dimensional space (X, Y, and Z)). However, there are certain situations where it is useful to think in terms of projective geometry instead. Projective geometry has an extra dimension, called W, in addition to the X, Y, and Z dimensions. This four-dimensional space is called "projective space" and coordinates in projective space are called "homogeneous coordinates".

Before we move on to 3D, we have to look at how projective geometry works in 2D. Imagine a projector that is projecting a 2D image onto a screen. It is easy to identify the X and Y dimensions of the projected image (Figure 13).

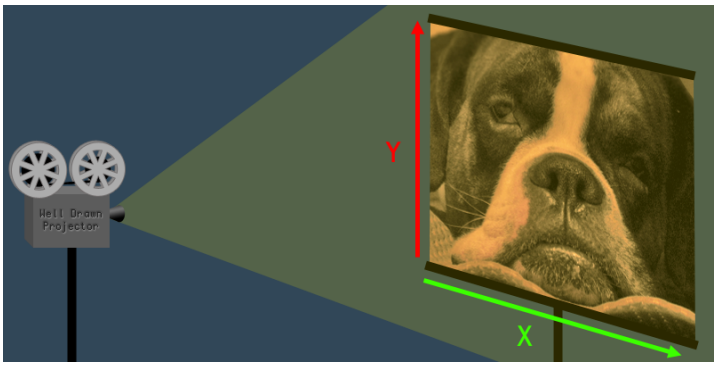


Figure 13: A projector that is projecting a 2D image onto a screen

Now, if we step back from the 2D image and look at the projector and the screen, we can see the W dimension too. The W dimension is the distance from the projector to the screen (Figure 14).

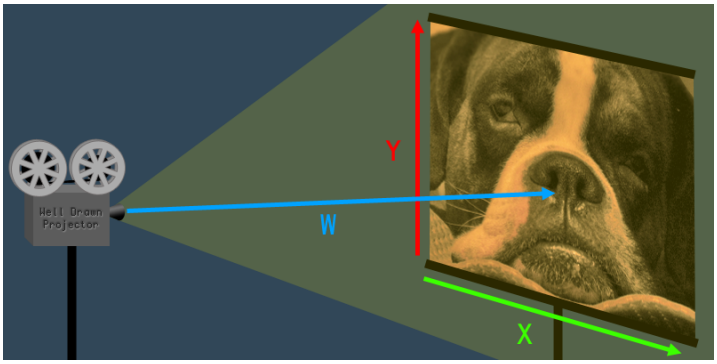


Figure 14: The W dimension is the distance from the projector to the screen

Now, if we move the projector closer to the screen, the whole 2D image becomes smaller. If we move the projector away from the screen, the 2D image becomes larger. As we can see, the value of W affects the size (a.k.a. scale) of the image (Figure 15).

There is no such thing as a 3D projector, so it is harder to imagine projective geometry in 3D, but the W value works exactly the same as it does in 2D. When W increases, the coordinate expands (scales up). When W decreases, the coordinate shrinks (scales down). The W is basically a scaling transformation for the 3D coordinate. The usual advice for 3D program-

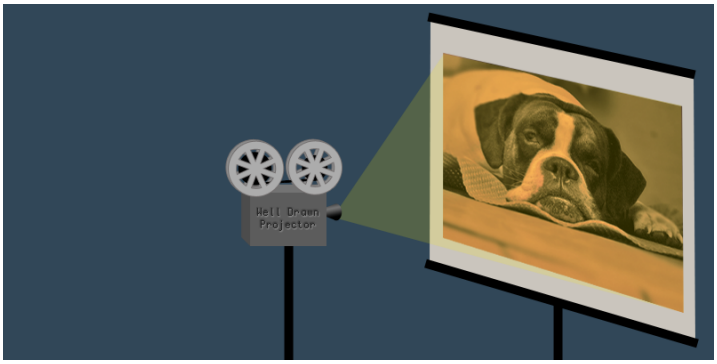


Figure 15: The projector close to the screen

ming beginners is to always set W=1 whenever converting a 3D coordinate to a 4D coordinate. The reason for this is that when you scale a coordinate by 1 it does not shrink or grow, it just stays the same size. So, when $W = 1$ it has no effect on the X, Y or Z values.

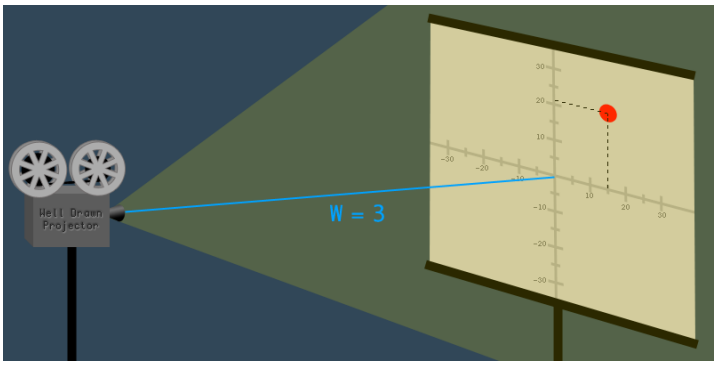


Figure 16: The projector is 3 meters away from the screen

For this reason, when it comes to 3D computer graphics, coordinates are said to be "correct" only when $W = 1$. If we rendered coordinates with $W > 1$ then everything would look too small, and with $W < 1$ everything would look too big. If we tried to render with $W = 0$ our program would

crash when it attempted to divide by zero. With $W < 0$ everything would flip upside-down and back-to-front.

Mathematically speaking, there is no such thing as an "incorrect" homogeneous coordinate. Using coordinates with W=1 is just a useful convention for 3D computer graphics. Now, Let us say that the projector is 3 meters away from the screen (Figure 16), and there is a dot on the 2D image at the coordinate $(15, 21)$. This gives us the projective coordinate vector:

$$(X, Y, W) = (15, 21, 3)$$

Now, we can imagine that the projector was pushed closer to the screen so that the distance was 1 meter. The closer the project gets to the screen, the smaller the image becomes. The projector has moved three times closer, so the image becomes three times smaller. If we take the original coordinate vector and divide all the values by three, we get the new vector where $W = 1$:

$(\frac{15}{3}, \frac{21}{3}, \frac{3}{3}) = (5, 7, 1)$ The dot is now at coordinate $(5, 7)$.

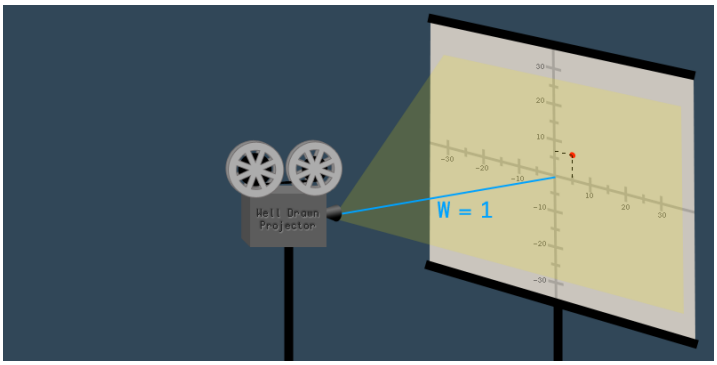


Figure 17: The projector is 1 meter away from the screen

This is how an "incorrect" homogeneous coordinate is converted to a "correct" coordinate: divide all the values by W. The process is exactly the same for 2D and 3D coordinates. Dividing all the values in a vector is done by scalar multiplication with the reciprocal of the divisor. Here is a 4D example:

$$(\tfrac{1}{5}(10, 20, 30, 5) = (\tfrac{10}{5}, \tfrac{20}{5}, \tfrac{30}{5}, \tfrac{5}{5}) = (2, 4, 6, 1)$$

In regard to 3D computer graphics, homogeneous coordinates are useful in certain situations. The 4th dimension W is usually unchanged, when using homogeneous coordinates in matrix transformation. W is set to 1 when

converting a 3D coordinate into 4D, and it is usually still 1 after the transformation matrices are applied, at which point it can be converted back into a 3D coordinate by ignoring the W. This is true for all translation, rotation, and scaling transformations, which are by far the most common types of transformations. The notable exception is projection matrices, which do affect the W dimension.

In 3D, "perspective" is the phenomenon where an object appears smaller the further away it is from the camera. A far-away mountain can appear to be smaller than a cat, if the cat is close enough to the camera.

Figure 18: The far-away mountain appears to be smaller than the cat

Perspective is implemented in 3D computer graphics by using a transformation matrix (Figure 19) that changes the W element of each vertex. After the the camera matrix is applied to each vertex, but before the projection matrix is applied, the Z element of each vertex represents the distance away from the camera.

$$\begin{bmatrix} \tan^{-1}(\frac{FOVx}{2}) & 0 & 0 & 0 \\ 0 & \tan^{-1}(\frac{FOVy}{2}) & 0 & 0 \\ 0 & 0 & -\frac{Z_{far}+Z_{near}}{Z_{far}-Z_{near}} & -\frac{2\,(Z_{near}\,Z_{far})}{Z_{far}-Z_{near}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Figure 19: Perspective-Projection Matrix

Therefore, the larger Z is, the more the vertex should be scaled down. The W dimension affects the scale, so the projection matrix just changes the W

value based on the Z value. Here is an example of a perspective projection matrix being applied to a homogeneous coordinate:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 2 \\ 3 \\ 4 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 4 \\ 4 \end{bmatrix}$$

It is easy to notice how the W value is changed to 4, which comes from the Z value. After the perspective projection matrix is applied, each vertex undergoes "perspective division". Perspective division is just a specific term for converting the homogeneous coordinate back to W=1, as explained earlier. Continuing with the example above, the perspective division step would look like this:

$$\frac{1}{4}(2, 3, 4, 4) = (0.5, 0.75, 1, 1)$$

After perspective division, the W value is discarded, and we are left with a 3D coordinate that has been correctly scaled according to a 3D perspective projection.

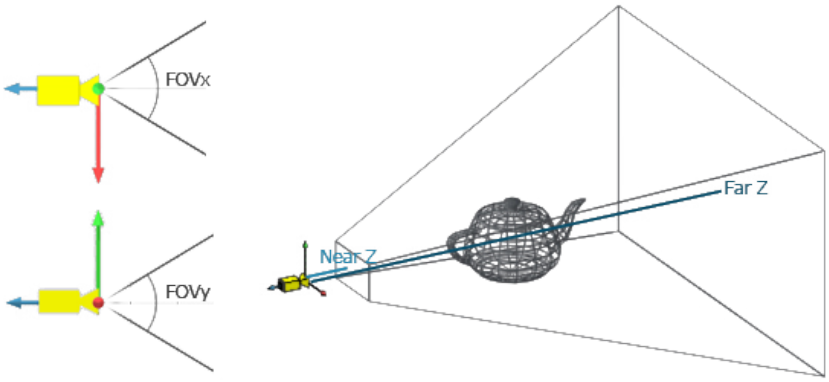


Figure 20: Perspective Projection

In this project we used the perspective projection and here is the function from Babylon.math.js that was implemented to compute the projection matrix.

Listing 2: Perspective function by Math.babylon.js

```
Matrix.PerspectiveFovLH = function PerspectiveFovLH(fov, aspect,
    znear, zfar) {
```

```
        var matrix = Matrix.Zero();
        var tan = 1.0 / (Math.tan(fov * 0.5));
        matrix.m[0] = tan / aspect;
        matrix.m[1] = matrix.m[2] = matrix.m[3] = 0.0;
        matrix.m[5] = tan;
        matrix.m[4] = matrix.m[6] = matrix.m[7] = 0.0;
        matrix.m[8] = matrix.m[9] = 0.0;
        matrix.m[10] = -zfar / (znear - zfar);
        matrix.m[11] = 1.0;
        matrix.m[12] = matrix.m[13] = matrix.m[15] = 0.0;
        matrix.m[14] = (znear * zfar) / (znear - zfar);
        return matrix;
    };
```

Consequently, the final matrix we multiply with the vertices in order to display a 3D scene into a 2D image is:

$$[ProjectionMatrix] \times [ViewMatrix] \times WorldMatrix = [ModelViewProjectionMatrix]$$

2.1.2 Primitive Assembly

A primitive consists of one or more vertices to form a point, line or closed polygon. Our system uses triangles to form faces. This stage takes the transformed vertices from the vertex processing stage and groups them into primitives (triangles).

2.1.3 Clipping

In the stage immediately following primitive assembly, primitives are clipped to fit just within the viewport or view volume and then prepared for rasterization to the display device. Once triangle vertices are transformed to their proper 2D locations, some of these locations may be outside the viewing window, or the area on the screen to which pixels will actually be written. Clipping is the process of truncating triangles to fit them inside the viewing area[11].

2.1.4 Rasterization

Rasterization is the task of taking an image described in a vector graphics format (shapes) and converting it into a raster image (pixels or dots) for output on a video display or printer, or for storage in a bitmap file format[12].

The most basic rasterization algorithm takes a 3D scene, described as polygons, and renders it onto a 2D surface, usually a computer monitor. Polygons are themselves represented as collections of triangles. Triangles are represented by 3 vertices in 3D-space. At a very basic level, rasterizers simply take a stream of vertices, transform them into corresponding 2-dimensional points on the viewer's monitor and fill in the transformed 2-dimensional triangles as appropriate.

The Rasterization Algorithm that was implemented in this project is very simple but also efficient[13]. If we are sorting the three vertices of each triangle on the Y coordinates in order to always have P1 followed by P2 followed by P3, we will finally have two possible cases as we see in Figure 21.

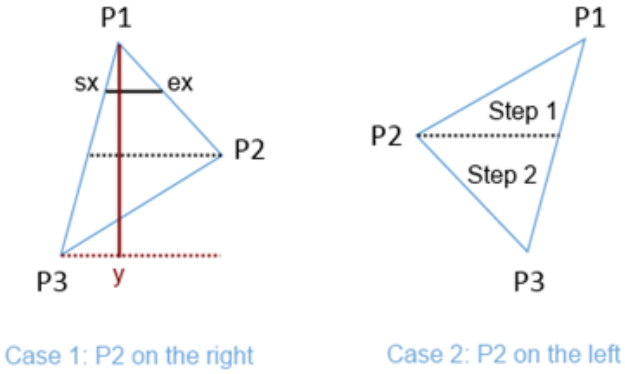


Figure 21: Cases of Triangles

- P2 is on the right of P1P3 line or
- P2 is on the left of P1P3 line

In our case, as we want to always draw our lines from left to right, from sx to ex, we have to handle these situations. Moreover, we are drawing from left to right by moving down from P1.y to P3.y following the red line drawn on the left case of the figure. But we need to change our logic reaching P2.y as the slope will change in both cases. For this reason, we have got two step in the scan line process.

- Moving down from P1.y to P2.y and

- Moving down from P2.y to P3.y, which is the final destination

To be able to sort the cases between case 1 and case 2, we simply need to compute the inverse slopes in this way:

$$dP1P2 = \frac{P2.x - P1.x}{P2.y - P1.y} \text{ , } dP1P3 = \frac{P3.x - P1.x}{P3.y - P1.y}$$

If dP1P2 > dP1P3, then we are in the first case with P2 on the right, otherwise if dP1P2 < dP1P3, we are in the second case with P2 on the left.

Now that we have seen the basic logic of our algorithm, we need to know how to compute X on each line between SX (Start X) and EX (End X). So we need to compute SX and EX first. As we know the Y value and the slope P1P3 and P1P2, we can easily find SX and EX we are interested in.

Let's take the step 1 of the case 1 as an example. First step is to compute our gradient with the current Y value in our loop. It will tell us at which stage we are in the scan line processing between P1.y and P2.y in Step 1.

$$gradient = \frac{currentY - P1.Y}{P2.Y - P1.Y}$$

As X and Y are linearly linked, we can interpolate SX based on this gradient using P1.x and P3.x, and interpolate EX using P1.x and P2.x. Finally, knowing the SX and EX we are able to draw the line between them.

2.1.5 Z-buffering

In computer graphics, z-buffering, also known as depth buffering, is the management of image depth coordinates in 3D graphics[14]. It is one solution to the visibility problem, which is the problem of deciding which elements of a rendered scene are visible, and which are hidden.

When an object is rendered, the depth of a generated pixel (z coordinate) is stored in a buffer (the z-buffer or depth buffer). This buffer is usually arranged as a two-dimensional array (x-y) with one element for each screen pixel. If another object of the scene must be rendered in the same pixel, the method compares the two depths and overrides the current pixel if the object is closer to the observer. The chosen depth is then saved to the z-buffer, replacing the old one.

Finally, in our implementation in the same way we are interpolating X value between each side of the triangles, we need to interpolate also Z values using the same algorithm for each pixel.

2.1.6 Shading

In computer graphics, shading refers to the process of altering the color of an object/surface/polygon in the 3D scene, based on its transparency, material, fresnel, subsurface scattering, angle to lights and distance from lights to create a photorealistic effect. Shading is performed during the rendering process by a program called a shader[15].

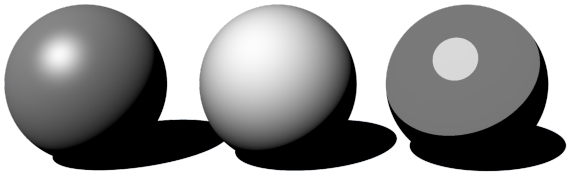


Figure 22: Shading in Computer Graphics

Flat Shading Flat shading is a lighting technique used in 3D computer graphics to shade each polygon of an object based on the angle between the polygon's surface normal and the direction of the light source, their respective colors and the intensity of the light source. It is usually used for high speed rendering where more advanced shading techniques are too computationally expensive. As a result of flat shading all of the polygon's vertices are colored with one color, allowing differentiation between adjacent polygons.

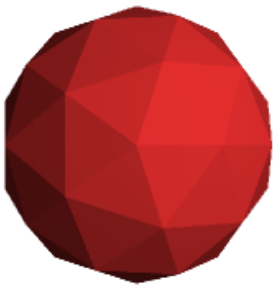


Figure 23: Flat Shading

To be able to apply the flat shading algorithm, we first need to compute the normal vector of the face. Once we have got it, we need to know the angle between this normal vector and the light vector. To be more precise, we use a dot product which give us the cosine of the angle between those 2 vectors. As this value could be -1 and 1, we cut it between 0 and 1. This final value will be used to apply the quantity of light to apply to every face based on its current color. In conclusion, the final color of the face will be:

$$finalcolor = color * max(0, cos(angle))$$

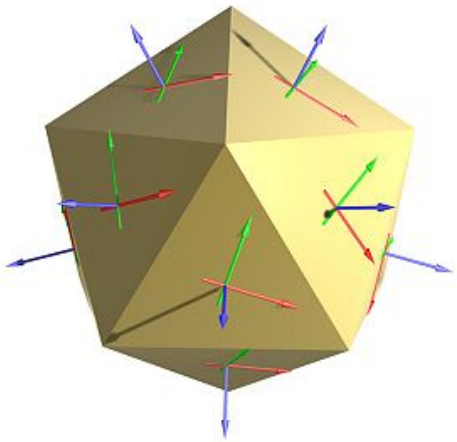


Figure 24: Blue Arrows: normals of the face, Green and Red Arrows: any edge vector of the face

After the calculation of the normal coordinates we need to define a light. In our project we used the simplest one: the **point light**. The point light is simply a 3D point (a Vector3). The quantity of light that every face receives is be the same whatever the distance from the light is. We will then simply vary the intensity based on the angle between the normal vector and the vector made of this point light and the center of the face.

So the light direction vector will be:

$$lightDirection = lightPosition - centerFacePosition$$

This will give us the light direction vector. To compute the angle between this light vector and the normal vector and finally the intensity of the color, we use a dot product.

$$angle = normalVector \cdot lightDirection$$

Gouraud Shading Gouraud shading, named after Henri Gouraud[16], is an interpolation method used in computer graphics to produce continuous shading of surfaces represented by polygon meshes. In practice, Gouraud shading is most often used to achieve continuous lighting on triangle surfaces by computing the lighting at the corners of each triangle and linearly interpolating the resulting colors for each pixel covered by the triangle.

Figure 25: Gouraud Shading

In this type of shading, rather than using 1 unique normal per face, and thus a unique color per face, we are using 3 normals: 1 per vertex of our triangles. We will then have 3 level of colors defined and we will interpolate the color of each pixel between each vertex using the same algorithm. Using this interpolation, we will then have a continuous lighting on our triangles.

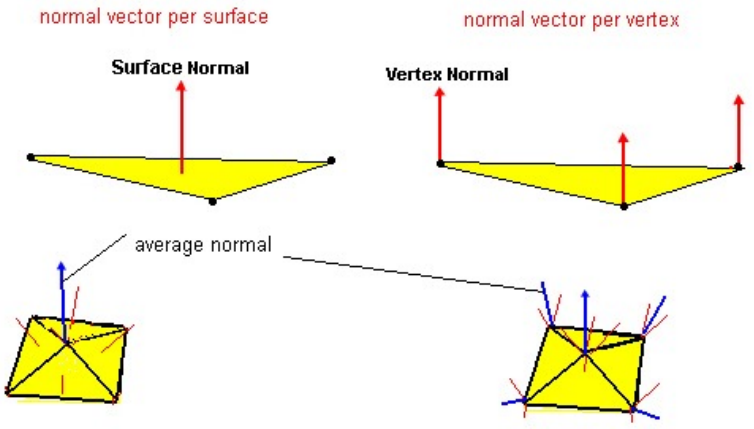


Figure 26: Face and Vertex Normals

We can see in Figure 26 the differences between flat shading and gouraud shading. The flat uses a centered unique normal and the gouraud uses 3 normals. It can be seen also on a 3D mesh (the pyramid) that the normal is per vertex per face. This means that the same vertex will have different normals based on the face it is currently being drawn.

Another good way to understand what we do with this shading is illustrated in Figure 27. In this figure, if the upper vertex normal has an angle > 90 degrees with the light direction, its color should then be black (minimum level of light = 0). If the two other vertex normal has an angle of 0 degree with the light direction, this means that they should receive the maximum level of light (1). To fill the triangle, we interpolate to color level between each vertex in order to have a nice gradient.

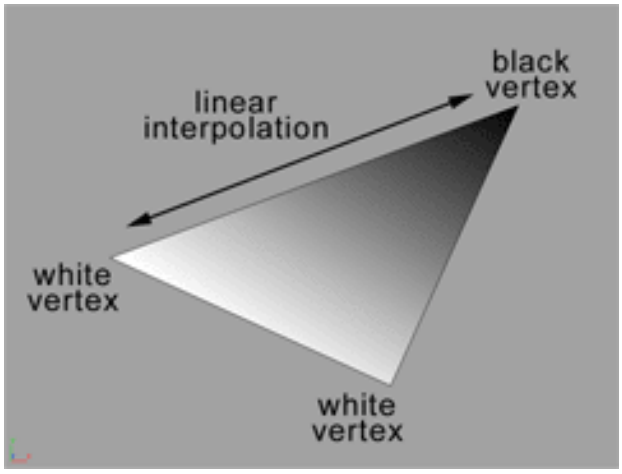


Figure 27: Gouraud shading linear interpolation

2.1.7 Texture Mapping

Texture mapping means applying any type of picture on one or more faces of a 3D model[17]. The picture, also known as texture can be anything but is often a pattern such as bricks, foliage, barren land, etc. that adds realism to the scene.

For example, compare the images in Figure 28. The left pyramid uses texture to fill in the faces and the right one uses gradient colors.

To get texture mapping working, three things need to be done:

- Load a texture into a memory.
- Supply texture coordinates with the vertices (to map the texture to them) and
- Perform a sampling operation from the texture using the texture coordinates in order to get the right pixel color

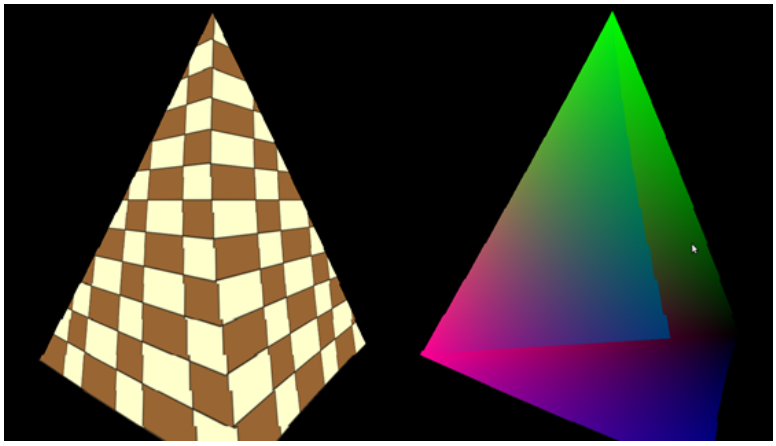


Figure 28: Left pyramid: with texture, Right pyramid: without texture

Since a triangle is scaled, rotated, translated and finally projected it can land on the screen in numerous ways and look very different depending on its orientation to the camera. What the GPU needs to do is make the texture follow the movement of the vertices of the triangle so that it will look real (if the texture appear to "swim" across the triangle it will not). To do that, a set of coordinates known as 'texture coordinates' have to be supplied to each vertex. As the GPU rasterizes the triangle it interpolates the texture coordinates across the triangle face and in the fragment shader these coordinates are mapped to the texture. This action is known as 'sampling' and the result of sampling is a texel (a pixel in a texture). The **texel** often contains a color which is used to paint the corresponding pixel on the screen.

In this implementation, a single 2D image was used as a texture to map every face of the mesh in 3D. A 2D texture has a width and height that can be any number within the limitations of the specification. Multiplying the

width by height tells you the number of texels in the texture. The texture coordinates of a vertex are not the coordinates of a texel inside the texture. That would be too limiting because replacing a texture with one that has different width/height means that we will need to update the texture coordinates of all the vertices to match the new texture. The ideal scenario is to be able to change textures without changing texture coordinates.

Therefore, texture coordinates are specified in **texture space** which is simply the normalized range [0, 1]. This means that the texture coordinate is usually a fraction and by multiplying that fraction with the corresponding width/height of a texture we get the coordinate of the texel in the texture. For example, if the texture coordinate is [0.5, 0.1] and the texture has a width of 320 and a height of 200 the texel location will be (160, 20) (0.5 * 320 = 160 and 0.1 * 200 = 20).

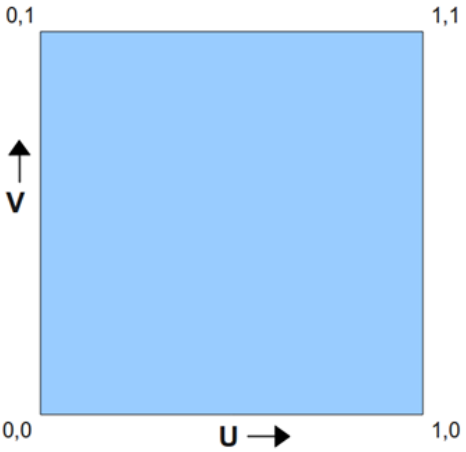
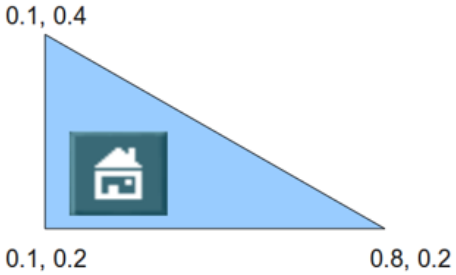


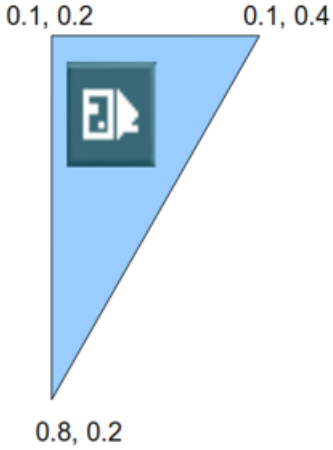
Figure 29: U and V texture coordinates

The usual convention is to use **U and V** as the axis of the texture space where U corresponds to X in the 2D cartesian coordinate system and V corresponds to Y. System treats the values of the UV axes as going from left to right on the U axis and down to up on the V axis.

Figure 29 presents the texture space and you can see the origin of that space in the bottom left corner. U grows towards the right and V grows up. Now consider a triangle whose texture coordinates are specified in the following picture:



Let us say that we apply a texture such that when using these texture coordinates we get the picture of the small house in the location above. Now the triangle goes through various transformations and when the time comes to rasterize it, it looks like this:



It can be seen, that the texture coordinates "stick" to the vertices as they are a core attributes and they do not change under the transformations. When interpolating the texture coordinates most pixels get the same texture coordinates as in the original picture (because they remained in the same place relative to the vertices) and since the triangle was flipped so is the texture which is applied to it. This means that as the original triangle is rotated, stretched or squeezed the texture diligently follows it.

Note that there are also techniques that change the texture coordinates in order to move texture across the triangle face in some controlled way.

2.2 3D Rendering Effects

2.2.1 Perlin Noise

Perlin noise is a type of gradient noise developed by Ken Perlin in 1983 as a result of his frustration with the "machine-like" look of computer graphics at the time[21]. In 1997, Perlin was awarded an Academy Award for Technical Achievement for discovering the algorithm.

Figure 30: Fire is created using the Perlin noise function

Perlin noise is a procedural texture primitive, a type of gradient noise used by visual effects artists to increase the appearance of realism in computer graphics. The function has a pseudo-random appearance, yet all of its visual details are the same size. This property allows it to be readily controllable; multiple scaled copies of Perlin noise can be inserted into mathematical expressions to create a great variety of procedural textures. Synthetic textures using Perlin noise are often used in CGI to make computer-generated visual elements, such as object surfaces, fire, smoke, or clouds, appear more natural, by imitating the controlled random appearance of textures of nature.

It is also frequently used to generate textures when memory is extremely limited, such as in demos, and is increasingly finding use in graphics processing units for real-time graphics in computer games.

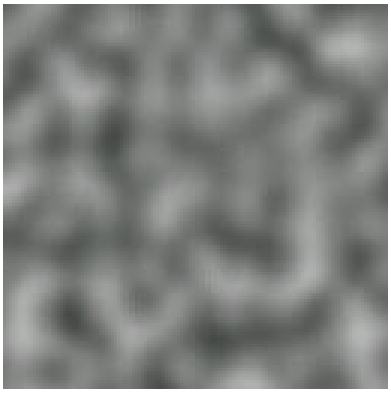


Figure 31: Two dimensional slice through 3D Perlin noise at z=0

Perlin noise is most commonly implemented as a two-, three- or four-dimensional function, but can be defined for any number of dimensions. An implementation typically involves three steps: grid definition with random gradient vectors, computation of the dot product between the distance-gradient vectors and interpolation between these values. In this project, we implemented the 3D version of the function.

Grid definition Define an n-dimensional grid. At each grid node assign a random gradient vector of unit length in n dimensions. For a one-dimensional grid each node will be assigned either +1 or -1, for a two-dimensional grid each node will be assigned a random vector on the unit circle, and so forth for higher dimensions.

Computation of the (pseudo-) random gradients in one and two dimensions is trivial using a random number generator. For higher dimensions a Monte Carlo approach can be used where random Cartesian coordinates are chosen in a unit cube, points falling outside the unit ball are discarded, and the remaining points are normalized to lie on the unit sphere. The process is continued until the required number of random gradients are obtained.

In order to negate the expensive process of computing new gradients for each grid node, some implementations use a hash and lookup table for a finite number of precomputed gradient vectors. The use of a hash also permits the inclusion of a random seed where multiple instances of Perlin noise

are required.

Dot product Given an n-dimensional argument for the noise function, the next step in the algorithm is to determine into which grid cell the given point falls. For each corner node of that cell, the distance vector between the point and the node is determined. The dot product between the gradient vector at the node and the distance vector is then computed.

For a point in a two-dimensional grid, this will require the computation of 4 distance vectors and dot products, while in three dimensions 8 distance vectors and 8 dot products are needed. This leads to the $O(2^n)$ complexity scaling.

Interpolation The final step is interpolation between the 2^n dot products computed at the nodes of the cell containing the argument point. This has the consequence that the noise function returns 0 when evaluated at the grid nodes themselves.

Interpolation is performed using a function that has zero first derivative (and possibly also second derivative) at the 2^n grid nodes. This has the effect that the gradient of the resulting noise function at each grid node coincides with the precomputed random gradient vector there. If $n = 1$, an example of a function that interpolates between value a_0 at grid node 0 and value a_1 at grid node 1 is

$$f(x) = a_0 + \text{smoothstep}(x) \cdot (a_1 - a_0), \text{ for } 0 \leq x \leq 1$$

The following is pseudocode for a two-dimensional implementation of Classical Perlin Noise[22].

Listing 3: Pseudocode for 2D Implementation of Perlin Noise

```
// Function to linearly interpolate between a0 and a1
 // Weight w should be in the range [0.0, 1.0]
 function lerp(float a0, float a1, float w) {
     return (1.0 - w)*a0 + w*a1;
 }

 // Computes the dot product of the distance and gradient vectors.
 function dotGridGradient(int ix, int iy, float x, float y) {

     // Precomputed (or otherwise) gradient vectors at each grid node
```

```
    extern float Gradient[IXMAX][IYMAX][2];
    // Compute the distance vector
    float dx = x - (float)ix;
    float dy = y - (float)iy;
    // Compute the dot-product
    return (dx*Gradient[iy][ix][0] + dy*Gradient[iy][ix][1]);
}

// Compute Perlin noise at coordinates x, y
function perlin(float x, float y) {

    // Determine grid cell coordinates
    int x0 = (x > 0.0 ? (int)x : (int)x - 1);
    int x1 = x0 + 1;
    int y0 = (y > 0.0 ? (int)y : (int)y - 1);
    int y1 = y0 + 1;

    // Determine interpolation weights
    // Could also use higher order polynomial/s-curve here
    float sx = x - (float)x0;
    float sy = y - (float)y0;

    // Interpolate between grid point gradients
    float n0, n1, ix0, ix1, value;
    n0 = dotGridGradient(x0, y0, x, y);
    n1 = dotGridGradient(x1, y0, x, y);
    ix0 = lerp(n0, n1, sx);
    n0 = dotGridGradient(x0, y1, x, y);
    n1 = dotGridGradient(x1, y1, x, y);
    ix1 = lerp(n0, n1, sx);
    value = lerp(ix0, ix1, sy);

    return value;
}
```

2.2.2 Particle System

A particle system is a technique in game physics, motion graphics, and computer graphics that uses a large number of very small sprites, 3D models, or other graphic objects to simulate certain kinds of "fuzzy" phenomena, which are otherwise very hard to reproduce with conventional rendering techniques - usually highly chaotic systems, natural phenomena, or processes caused by chemical reactions[23].

Examples of such phenomena which are commonly replicated using particle systems include fire, explosions, smoke, moving water (such as a waterfall), sparks, falling leaves, rock falls, clouds, fog, snow, dust, meteor tails, stars and galaxies, or abstract visual effects like glowing trails, magic spells, etc. - these use particles that fade out quickly and are then re-emitted from the effect's source. Another technique can be used for things that contain many strands - such as fur, hair, and grass - involving rendering an entire particle's lifetime at once, which can then be drawn and manipulated as a single strand of the material in question.

Typically a particle system's position and motion in 3D space are controlled by what is referred to as an emitter. The emitter acts as the source of the particles, and its location in 3D space determines where they are generated and where they move to. A regular 3D mesh object, such as a cube or a plane, can be used as an emitter. The emitter has attached to it a set of particle behavior parameters. These parameters can include:

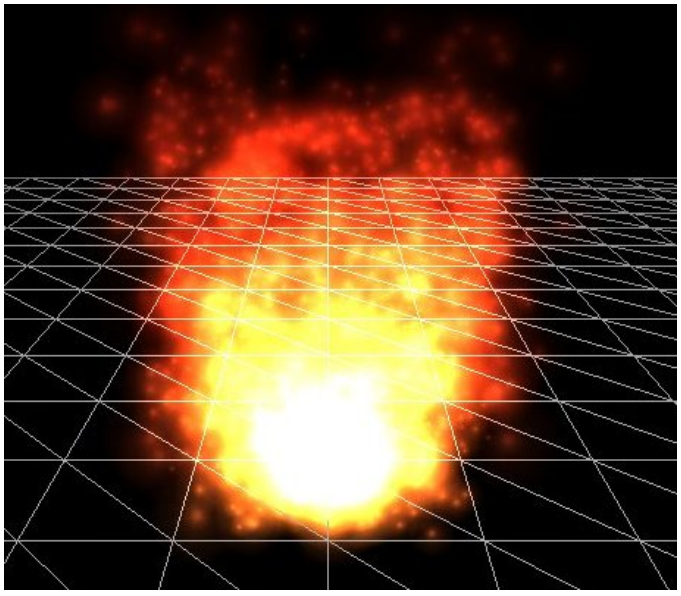


Figure 32: A particle system used to simulate a fire, created in 3dengfx

- The spawning rate (how many particles are generated per unit of time)
- The particles' initial velocity vector (the direction they are emitted

upon creation)

- Particle lifetime (the length of time each individual particle exists before disappearing)
- Particle color, and many more.

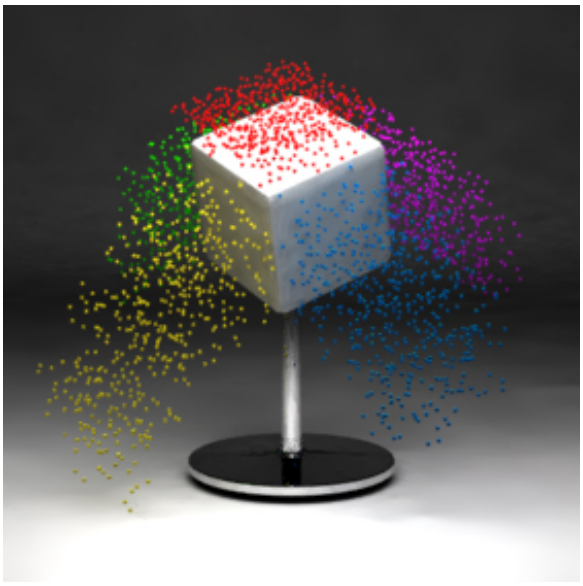


Figure 33: A cube emitting 5000 animated particles, obeying a "gravitational" force in the negative Y direction.

It is common for all or most of these parameters to be "fuzzy" instead of a precise numeric value, the artist specifies a central value and the degree of randomness allowable on either side of the center (i.e. the average particle's lifetime might be 50 frames 20%). When using a mesh object as an emitter, the initial velocity vector is often set to be normal to the individual face(s) of the object, making the particles appear to "spray" directly from each face but optional.

2.2.3 Displacement Mapping

Displacement mapping is an alternative computer graphics technique using a (procedural-) texture- or height map to cause an effect where the actual geometric position of points over the textured surface are displaced, often

along the local surface normal, according to the value the texture function evaluates to at each point on the surface[24]. It gives surfaces a great sense of depth and detail, permitting in particular self-occlusion, self-shadowing and silhouettes; on the other hand, it is the most costly of this class of techniques owing to the large amount of additional geometry.

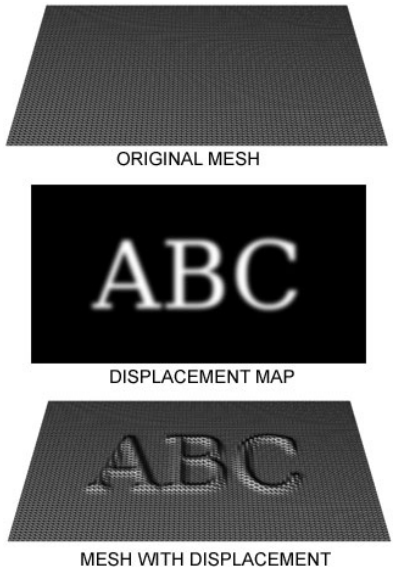


Figure 34: Displacement mapping in mesh using texture

In this project we used the idea of displacement mapping in combination with Perlin noise function in order to generate pseudorandom terrains like the scene we can see in Figure 35.

2.3 Introduction to FPGAs

At the highest level, FPGAs are reprogrammable silicon chips. Using prebuilt logic blocks and programmable routing resources, we can configure these chips to implement custom hardware functionality without ever having to pick up a breadboard or soldering iron. We develop digital computing tasks in software and compile them down to a configuration file or bitstream that contains information on how the components should be wired together. In addition, FPGAs are completely reconfigurable and instantly take on a brand new "personality" when you recompile a different configuration of circuitry. In the past, FPGA technology could be used only by engineers with a deep understanding of digital hardware design. The rise of high-level

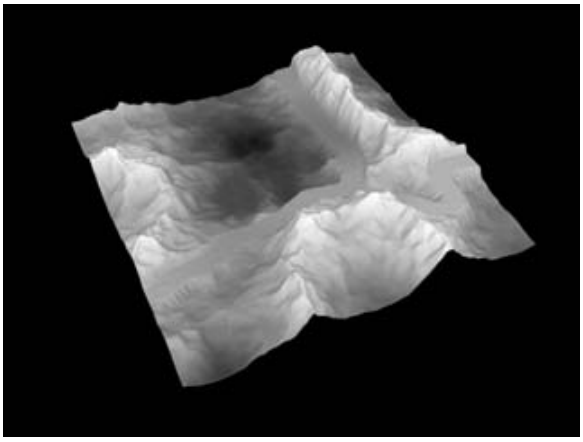


Figure 35: Displacement Mapping for creating multilevel terrain

design tools, however, is changing the rules of FPGA programming, with new technologies that convert graphical block diagrams or even C code into digital hardware circuitry[25].

FPGA chip adoption across all industries is driven by the fact that FPGAs combine the best parts of ASICs and processor-based systems. FPGAs provide hardware-timed speed and reliability, but they do not require high volumes to justify the large upfront expense of custom ASIC design. Reprogrammable silicon also has the same flexibility of software running on a processor-based system, but it is not limited by the number of processing cores available. Unlike processors, FPGAs are truly parallel in nature, so different processing operations do not have to compete for the same resources. Each independent processing task is assigned to a dedicated section of the chip, and can function autonomously without any influence from other logic blocks. As a result, the performance of one part of the application is not affected when you add more processing.

Here are the top 5 benefits of FPGAs.

1. Performance
2. Time to Market
3. Cost
4. Reliability

5. Long-Term Maintenance

Performance Taking advantage of hardware parallelism, FPGAs exceed the computing power of digital signal processors (DSP's) by breaking the paradigm of sequential execution and accomplishing more per clock cycle. BDTI, a noted analyst and benchmarking firm, released benchmarks showing how FPGAs can deliver many times the processing power per dollar of a DSP solution in some applications. Controlling inputs and outputs (I/O) at the hardware level provides faster response times and specialized functionality to closely match application requirements.

Time to market FPGA technology offers flexibility and rapid prototyping capabilities in the face of increased time-to-market concerns. We can test an idea or concept and verify it in hardware without going through the long fabrication process of custom ASIC design. We can then implement incremental changes and iterate on an FPGA design within hours instead of weeks. Commercial off-the-shelf (COTS) hardware is also available with different types of I/O already connected to a user-programmable FPGA chip. The growing availability of high-level software tools decreases the learning curve with layers of abstraction and often offers valuable IP cores (prebuilt functions) for advanced control and signal processing.

Cost The nonrecurring engineering (NRE) expense of custom ASIC design far exceeds that of FPGA-based hardware solutions. The large initial investment in ASICs is easy to justify for OEM's shipping thousands of chips per year, but many end users need custom hardware functionality for the tens to hundreds of systems in development. The very nature of programmable silicon means you have no fabrication costs or long lead times for assembly. Because system requirements often change over time, the cost of making incremental changes to FPGA designs is negligible when compared to the large expense of respinning an ASIC.

Reliability While software tools provide the programming environment, FPGA circuitry is truly a "hard" implementation of program execution. Processor-based systems often involve several layers of abstraction to help schedule tasks and share resources among multiple processes. The driver layer controls hardware resources and the OS manages memory and processor bandwidth. For any given processor core, only one instruction can execute at a time, and processor-based systems are continually at risk of

time-critical tasks preempting one another. FPGAs, which do not use OS's, minimize reliability concerns with true parallel execution and deterministic hardware dedicated to every task.

Long-term maintenance As mentioned earlier, FPGA chips are field-upgradable and do not require the time and expense involved with ASIC redesign. Digital communication protocols, for example, have specifications that can change over time, and ASIC-based interfaces may cause maintenance and forward-compatibility challenges. Being reconfigurable, FPGA chips can keep up with future modifications that might be necessary. As a product or system matures, you can make functional enhancements without spending time redesigning hardware or modifying the board layout.

To sum up, the adoption of FPGA technology continues to increase as higher-level tools evolve to deliver the benefits of reprogrammable silicon to engineers and scientists at all levels of expertise.

2.4 High-Level Synthesis

High-level synthesis (HLS), sometimes referred to as C synthesis, electronic system-level (ESL) synthesis, algorithmic synthesis, or behavioral synthesis, is an automated design process that interprets an algorithmic description of a desired behavior and creates digital hardware that implements that behavior. Synthesis begins with a high-level specification of the problem, where behavior is generally decoupled from e.g. clock-level timing. Early HLS explored a variety of input specification languages, although recent research and commercial applications generally accept synthesizable subsets of ANSI C/C++/SystemC/Matlab. The code is analyzed, architecturally constrained, and scheduled to create a register-transfer level (RTL) hardware description language (HDL), which is then in turn commonly synthesized to the gate level by the use of a logic synthesis tool. The goal of HLS is to let hardware designers efficiently build and verify hardware, by giving them better control over optimization of their design architecture, and through the nature of allowing the designer to describe the design at a higher level of abstraction while the tool does the RTL implementation. Verification of the RTL is an important part of the process[26].

Hardware design can be created at a variety of levels of abstraction. The commonly used levels of abstraction are gate level, register-transfer level (RTL), and algorithmic level.

While logic synthesis uses an RTL description of the design, high-level synthesis works at a higher level of abstraction, starting with an algorithmic description in a high-level language such as SystemC and Ansi C/C++. The designer typically develops the module functionality and the interconnect protocol. The high-level synthesis tools handle the micro-architecture and transform untimed or partially timed functional code into fully timed RTL implementations, automatically creating cycle-by-cycle detail for hardware implementation. The (RTL) implementations are then used directly in a conventional logic synthesis flow to create a gate-level implementation.

In this project we used high-level synthesis to generate specific components of the implementation.

3 Related Work

Several projects have attempted to implement FPGA-based three dimensional graphics rendering. One of the works have been done includes "Implementation of a Simple 3D Graphics Pipeline" by Vladimir Kasik and Ales Kurecka[1]. This project presents a hardware-based graphics pipeline based on FPGA that utilizes parallel computation for simple projection of a wireframe 3D model. Pipelining and strong parallelism were commonly used to obtain the target frequency of 100MHz.

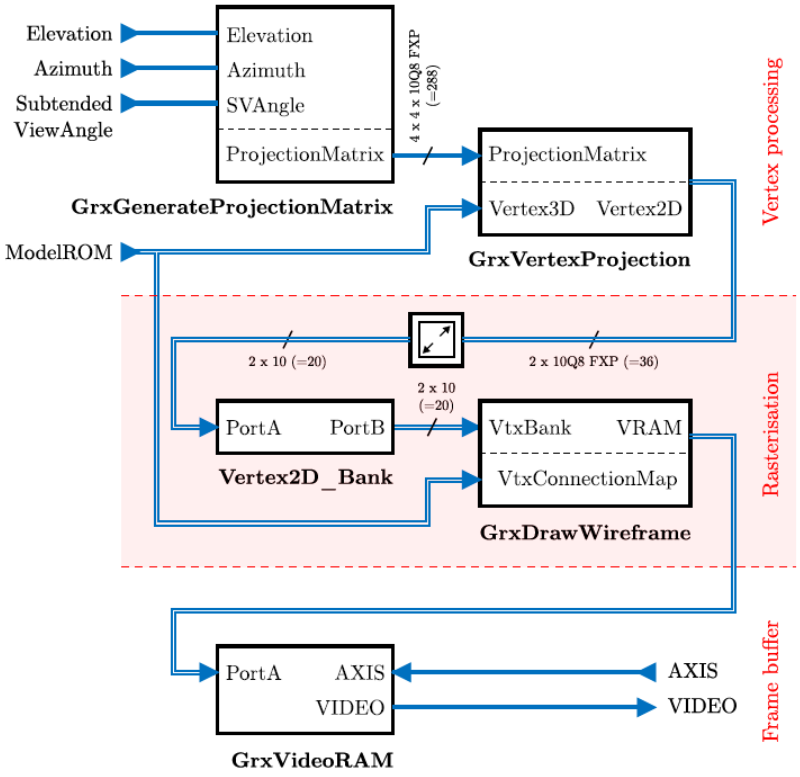


Figure 36: Simplified diagram of the implemented graphics pipeline.

This graphics pipeline consists of the unit for computing the projection matrix, the vertex unit, the simple rendering unit and a two-port video memory. The pipeline first deletes video RAM, computes the projection matrix with transferred parameters (azimuth, elevation, viewing angle) and reads the normalized 3D model from the vertex memory and their interconnections.

This is then displayed in 2D and rescaled for display on the screen. The wireframe model is then drawn in the video memory from the computed 2D vertices. The implementation of the project was for the Xilinx Spartan-6 circuit with a graphics output to VGA. The tests carried out resulted in real-time rendering at over 5000 FPS.

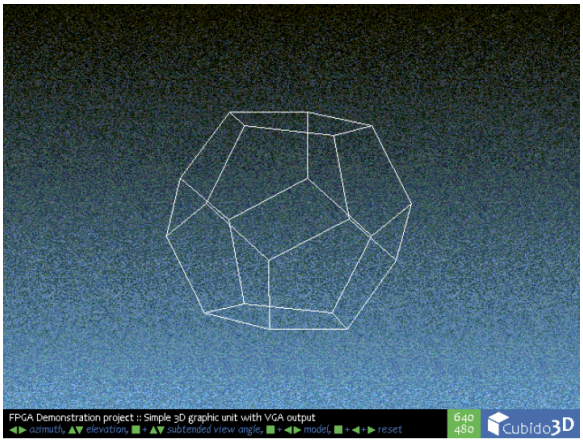


Figure 37: Example of an output by Kasik and Kurecka work.

Another one work was the "Implementation of a 3D Graphics Rasterizer with Texture and Slim Shader on FPGA" by Ajay Kashyap and Ashish Sharma[2]. They designed a 3D graphics hardware with rasterizer having texture and slim-shader for the efficient 3D graphics accelerator. All stages of the graphics pipeline were developed on FPGA using RTL design.

The main rendering operation such as texturing, shading, blending and depth comparison was performed by a component called slim shader. In slim shader vertical strip assignment was done, as shown in Figure 38.

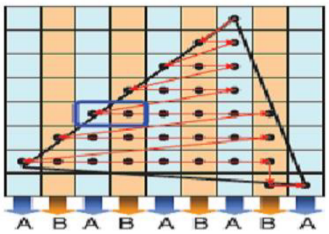


Figure 38: Slim shader Triangle Setup

Each triangle was divided into two alternative strips of A and B, and two pixel processors were used to rasterize the triangles in horizontal order. This triangle distribution to two pixel processors enhanced the overall performance of 3D pipeline by accelerating setup operation.

Kyungsu Kim et al. at Electronics and Telecommunications Research Institute in Korea have done research in 3D graphics hardware with title "Implementation of 3D Graphics Accelerator Using Full Pipeline Scheme on FPGA"[3]. The research was based on the graphics pipeline shown in Figure 39 which contains Geometry stage and Rasterization stage. Geometry stage consists of vertex shader, clipping engine and viewport mapping. Rasterization stage is composed of triangle setup engine, rasterizer, pixel shader and raster operators.

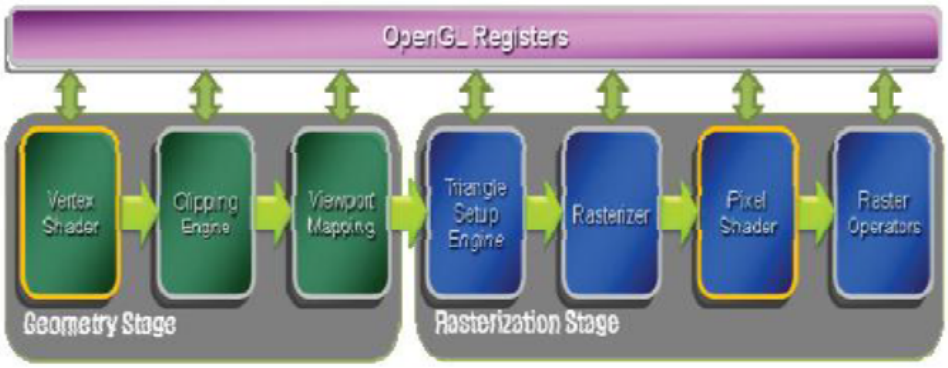


Figure 39: 3D graphics full pipeline

This system was synthesized and implemented on Virtex 5 FPGA and operated with 35MHz clock frequency. It was able to render 70,000 triangles of Stanford Bunny at 30 FPS.

Peter Szanto and Bela Feher designed an FPGA implementation which supports programmable pixel computing with title "Implementing a Programmable Pixel Pipeline in FPGAs"[4]. The implemented graphics accelerator supports hidden surface removal and shading in hardware. Transformation from local model coordinate system into the screen space's coordinate system, clipping and vertex based lighting are computed by the host processor. The hidden surface removal (HSR) part was based on the idea originally created by PowerVR: tile based deferred rendering [7]. This method ensures that no processing power is wasted on computing non-visible

pixels, and also saves memory bandwidth.

The implementation was not done using one of the most common hardware description languages (such as Verilog or VHDL), but an ANSI-C based tool, Celoxica's DK1 Design Suite[5].

Finally, Jeong-Joon Yoo et al. presented "Tile-based Path Rendering for Mobile Device"[6]. In this project, they designed a novel tile-based path rendering scheme that provides a fast rendering on mobile device. The proposed tile-based approach which effectively reduced memory I/O and computation simultaneously, gave them an acceptable high performance of path rendering on mobile device. Although tile-based rendering had been already used in 3D graphics or GPU, it was the first time to introduce the concept of tile-based scheme for path rendering.

Tile-based path rendering was composed of two steps as shown in Figure 40; i) Tile Binning step that generates Tile Bin Data, ii) Rendering step that renders path data located in each tile using the Tile Bin Data.

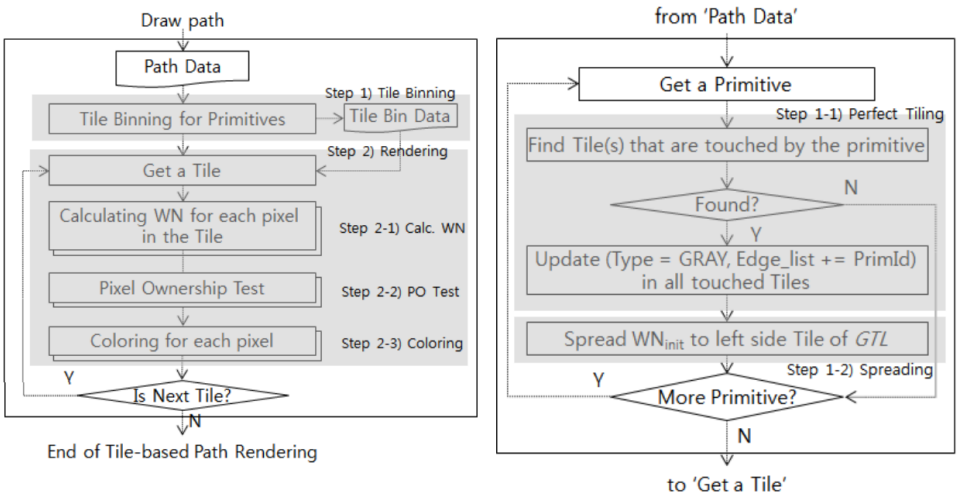


Figure 40: Overall Flow of Tile-based Path Rendering (Left: Overall Flow, Right: Tile Binning

As a result, they got a scalable performance (FPS) not only on high resolution but also on normal resolution pixel size. This scheme could be used not only in middle-end smart phone but also in FHD and QHD smart phone.

Apart form FPGA-based implementations, it is vital to mention a rendering pipeline implemented in CUDA[8], because CUDA is a powerful tool for general purpose computing that provides more flexible control over the graphics hardware.

Fang Liu et al. presented a system for fully programmable rendering architecture on current graphics hardware with title "FreePipe: a Programmable Parallel Rendering Architecture for Efficient Multi-Fragment Effects"[9]. This system bridges between the traditional graphics pipeline and a general purpose computing architecture while retaining the advantages of both sides. The core of the system is a z-buffer based triangle rasterizer that is entirely implemented in CUDA. All stages of the pipeline can be mapped to CUDA programming model. The system is fully programmable while still having comparable performance to the traditional graphics pipeline.

Two novel schemes had been proposed within this architecture for efficient rendering of multi-fragment effects in a single geometry pass using CUDA atomic operations. Both schemes had significant speed improvement over the stat-ofthe- art depth peeling algorithm. In Figure 41, we can see the performance of this implementation.

Model	Dragon	Buddha	Lucy	Neptune	Bunny
Tri No.	871K	1.0M	2.0M	4.0M	70K
FreePipe	322fps	256fps	183fps	98fps	645fps

Figure 41: Frame rates(fps) for various scenes by FreePipe

4 Project Overview and Design

This section presents an overview of the project's design. More specifically, this section presents the parts of the implementation, a high level design describing the system's various components, the target device and the software platform we used for the implementation.

4.1 Parts of Implementation

In this project we implemented a full 3D rendering pipeline in three separate phases and four advanced 3D rendering effects that use the previously implemented graphics pipeline.

4.1.1 Graphics Pipeline Implementation Phases

The graphics pipeline was implemented in three phases.

1. **Vertex Processing and Rasterization**: The vertices of the objects are joined three each time in order to create a triangle and then the inside are of the triangle is filled with color. Apart from lighting and texture mapping, all steps of the graphics pipeline are executed.

2. **Lighting**: First we define a light source at a point in 3D space. Then computing the angle between the light direction and the normal per face (or normal per vertex in Gouraud shading), every pixel's color is multiplied by an intensity value creating shading and giving photorealism to the scene.

3. **Texture Mapping**: In contrast with phases 1 and 2 where there is a fixed color for filling triangles, at this phase a texture is shown on every face of the mesh.

4.1.2 3D Rendering Effects

Perlin Noise Mapping to Ramp Texture In this part, we use Perlin-Noise function in order to create textures for the mesh. The color of every pixel in each face of the mesh depends on its coordinates (x, y, z). Perlin-Noise function is called with the coordinates of the pixel as inputs and a decimal number between 0 and 1 comes as output. This number is mapped with a ramp texture that contains 256 color values (shades of grey). The color we get after the mapping operation corresponds to the current pixel.

Particle System In this effect, 300 triangles (particles) are emitted from the same position in the 3D world (0, 0, 0.5) First of all, running a pseudorandom function we get each particle's parameters: lifespan, speed and direction. Particle coordinates are changed per frame depending on these three parameters and gravity force which is the same for all. The output on the screen looks like an emission of white particles falling down.

This part has a second version with the placement of a Repeller Object in the 3D world. In our example, this object is a cube with fixed coordinates. When a particle collides on the cube, it gets an opposing force depending on its direction. So, the particle bounces on the repeller and continues falling.

Displacement Mapping using Perlin Noise 3200 triangles have been initialized in order to create a terrain. When the user pushes the confirm button on the board, we get a random number r from 0 to 4095. This number is used as input to the PerlinNoise function PerlinNoise(x, r, z) (x and z are the coordinates of the current vertex).

The result of the perlin noise function is used as y value for every vertex of the mesh. Consequently, after rendering we can see a terrain with multiple levels on the screen. We can generate different terrains (4095 different terrains) randomly by just pushing the confirm button.

4.2 High Level Design

The high level design of this project includes four components. These components and their responsibilities are described below.

- Face Vertices ROM
- Rendering Processor
- Frame Buffer
- Display Driver

Each component had its own specific responsibilities. *FaceVerticesROM* is a memory where the coordinates of the 3D object we want to render are stored. Apart from that, this module is responsible for giving to processor three vertices a time that describe a triangle. In Lighting and Texture Mapping phases, FaceVerticesROM contains also the normals per vertex and the texture coordinates for each vertex.

Rendering Processor is the module in which all the graphics pipeline steps and raster operations are executed. All these operations are controlled by a component called Controller which is a part of Processor. The rasterization results (frame) is written to the frame buffer. As we proceed to Shading and Texture Mapping phases, this module will be larger and more complex due to various algorithms we launch.

Frame Buffer is the responsible component for containing the current frame. It is dual port because both processor and display unit are allowed to read from it but only processor can write.

Display Driver is a projection module which reads the frame from frame buffer and pixels one by one are displayed on the screen. This module operates independently of other operations are being executed on the processor unit.

It can be seen in Figure 42 that the system uses the Shared Memory model of communication.

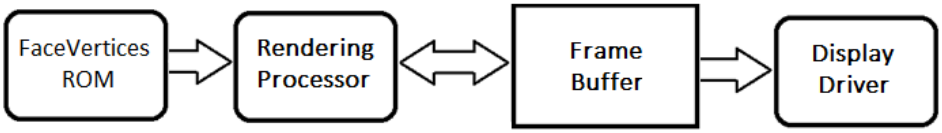


Figure 42: High Level Design

4.3 Target Device

The FPGA development board upon the system was developed is Kintex-7 FPGA KC705 Evaluation Kit[29]. The Kintex-7 FPGA KC705 evaluation kit provides a comprehensive, high-performance development and demonstration platform using the Kintex-7 FPGA family for high-bandwidth and high-performance applications in multiple market segments. The kit enables designing with DDR3, I/O expansion through FMC, and common serial standards, such as PCI Express, XAUI, and proprietary serial standards through the SMA interface.

Xilinx, the leading FPGA manufacturer, has designed the Kintex-7 with

high performance and is one of its ideal offering for FPGA-based graphics acceleration as it has the necessary resources for GPU computations. The selected XC7K325T-2FFG900C FPGA includes 326080 logic cells, 840 DSP slices, 407600 CLB Flip-flops, 4000Kb maximum distributed RAM and supports integrated hard memory, high performance clocking, serial IO and an integrated PCI-Express (PCI-E) endpoint block.

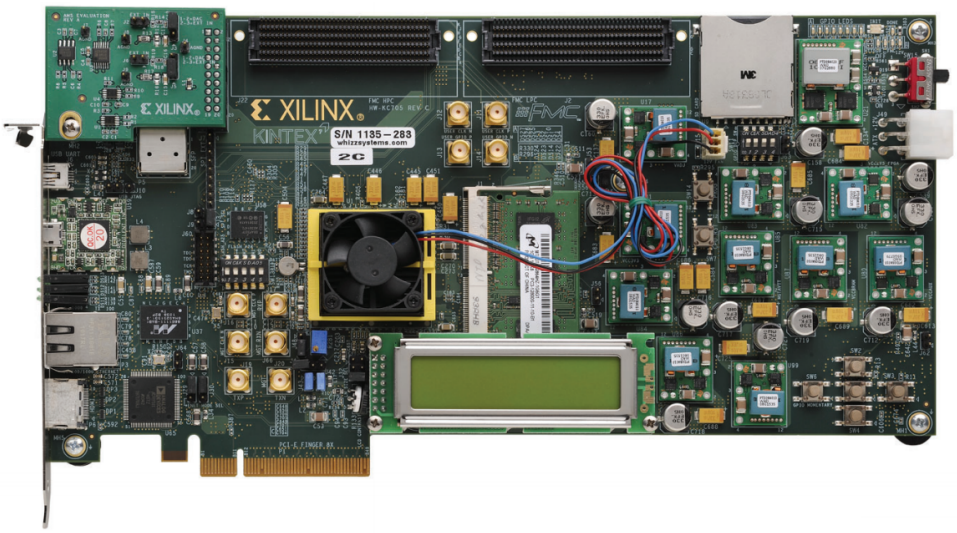


Figure 43: Kintex-7 KC705 Evaluation Kit

The KC705 Evaluation Kit enables developers to easily prototype designs with the XC7K325T-2FFG900C Kintex-7 FPGA. The kit includes all the basic components of the Xilinx Base Targeted Design Platform in one package. With the KC706, developers can easily take advantage of the features of the Kintex-7. Additionally, the kit includes HDMI video output, a Fixed Oscillator with differential 200MHz output, 1GB of DDR3 memory SODIMM , and various expansion connections. The support for the Kintex-7 FPGA,

Device	Logic Cells	Configurable Logic Blocks (CLBs)		DSP Slices	Block RAM Blocks		
		Slices	Max Distributed RAM (Kb)		18 Kb	36 Kb	Max (Kb)
XC7K325T	326,080	50,950	4,000	840	890	445	16,020

Figure 44: XC7K325T Device Feature Summary

digital video output interface, and 200MHz clock all made the KC705 an appropriate choice for FPGA-based graphics acceleration. Additionally, its support for PCI-E created an additional advantage for potential applications in a desktop PC environment.

4.4 Software Platform

As mentioned previously, the language used for this project was VHDL and the the target platform was Kintex-7. Because we used a 7-series board, the software platform used for the design was Vivado Design Suite 2014.1[28] (for lower series board we should use Xilinx ISE). Vivado Design Suite is a software suite produced by Xilinx for synthesis and analysis of HDL designs, superseding the older Xilinx ISE with additional features for system on a chip development and high-level synthesis.

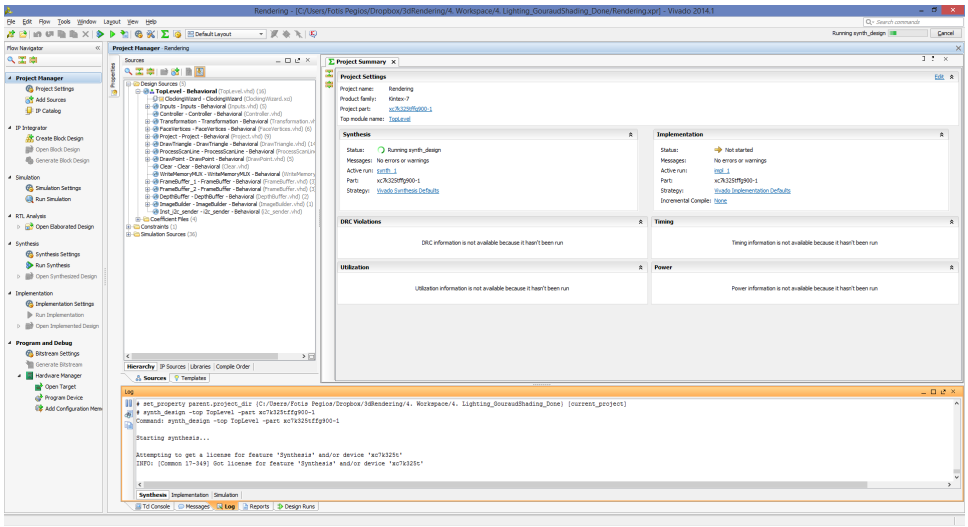


Figure 45: Xilinx Vivado Design Suite 2014.1 running synthesis

This software includes an in-built logic simulator and introduces high-level synthesis, with a toolchain that converts C code into programmable logic. Vivado has been described as a "state-of-the-art comprehensive EDA tool with all the latest bells and whistles in terms of data model, integration, algorithms, and performance".

Vivado enables developers to synthesize (compile) their designs, perform **timing analysis**, examine RTL diagrams, simulate a design's reaction to

different stimuli, and configure the target device with the programmer. So Vivado includes four components: Vivado High-Level Synthesis, Vivado Simulator and Vivado IP Integrator.

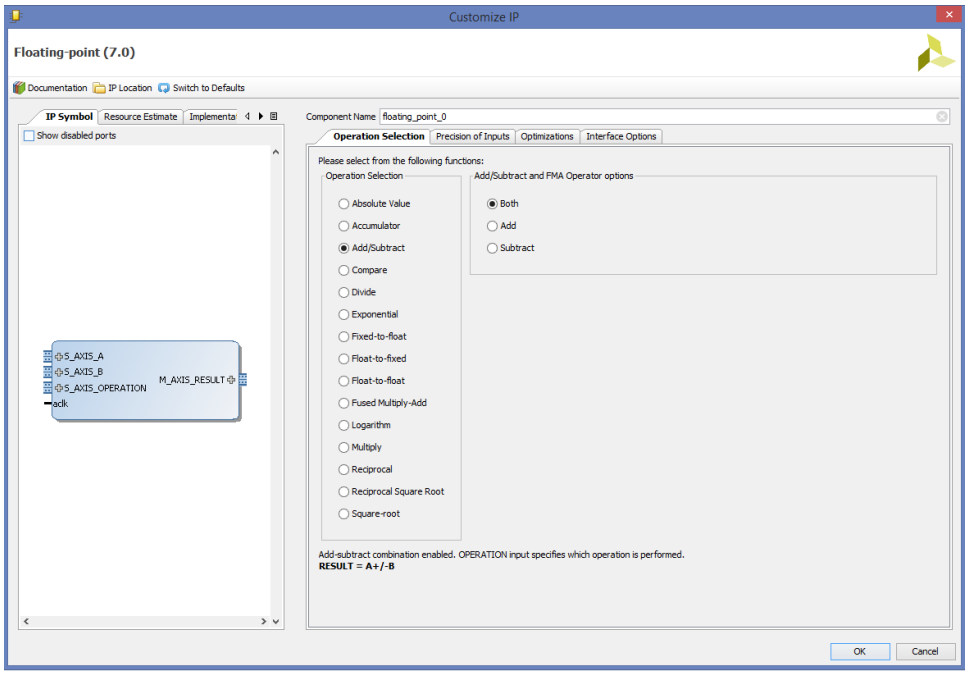


Figure 46: Xilinx floating-point IP that provides a range of floating-point operations.

Vivado High-Level Synthesis The Vivado High-Level Synthesis compiler enables C, C++ and SystemC programs to be directly targeted into Xilinx devices without the need to manually create RTL. Vivado HLS is widely reviewed to increase developer productivity, and is confirmed to support C++ classes, templates, functions and operator overloading.

Vivado Simulator The Vivado Simulator is a compiled-language simulator that supports mixed-language, TCL scripts, encrypted IP and enhanced verification. It gives the ability to developer to simulate his design before programming the device.

Vivado IP Integrator The Vivado IP Integrator allows engineers to quickly integrate and configure IP from the large Xilinx IP library. In-

tellectual Property library provides numerous implemented components to help developers focus on their application design and be more productive. For instance, there already implemented ip's that perform multiplication and other mathematical operations. In this project, the majority of mathematical operations were implemented using the floating point ip core

Vivado HLS was used in this project for specific computations, that are mentioned in the next section, and C was the preferred language. All these C codes were verified using simulation inside HLS and then we synthesized them to get the HDL code (in VHD files) and some TCL scripts. In order to import this RTL to our project, we added these VHD files to the implementations and ran the TCL scripts from the TCL console in Vivado. When TCL files are executed, some IP cores are created in order the RTL of this design to run properly and give the correct output.

Having an RTL description of the design in VHDL, we use Vivado to synthesize it, targeted for the Kintex-7 FPGA architecture. Once synthesized, a net-list is exported. This is used by the Xilinx Alliance Design Manager (XADM) FPGA layout tool to map the design to the FPGA using placement and routing in order to create an FPGA configuration bit-file. Finally, this bit-file is used to configure the FPGA to form the designed digital system.

5 Implementation

In this section, we will discuss about the architecture of all the parts and phases of the implementation in depth. On the grounds that the rendering effects use the graphics pipeline in order to be rendered, it is vital to begin with implementation of the pipeline.

In order to discuss the rendering pipeline's implementation, we have to analyze the individual components of the high level design, separately for every phase of the implementation.

5.1 Vertex Processing and Rasterization Phase

In this phase, we display the mesh on the screen using one solid color to fill its faces.

5.1.1 FaceVertices

This component is responsible for the coordinate production of three vertices that declare a triangle in order to draw a face of a 3D object.

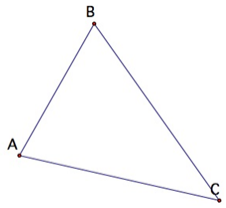


Figure 47: A, B and C are the vertices that describe triangle

In FaceVertices component there is a memory called VerticesROM. This ROM contains the x, y and z coordinates of all the vertices of a mesh. In order to find the appropriate three vertices that declare a triangle we need a second memory that is called IndicesROM. This ROM contains three numbers per triangle. These numbers correspond to the addresses in VerticesROM. The content of VerticesROM in these addresses are the x, y, z coordinates of the three vertices of the triangle.

All ROM memories were implemented using Distributed Memory Generator IP form Vivado. They were initialized with .coe files in binary radix. Coe files were generated with the use of Matlab.

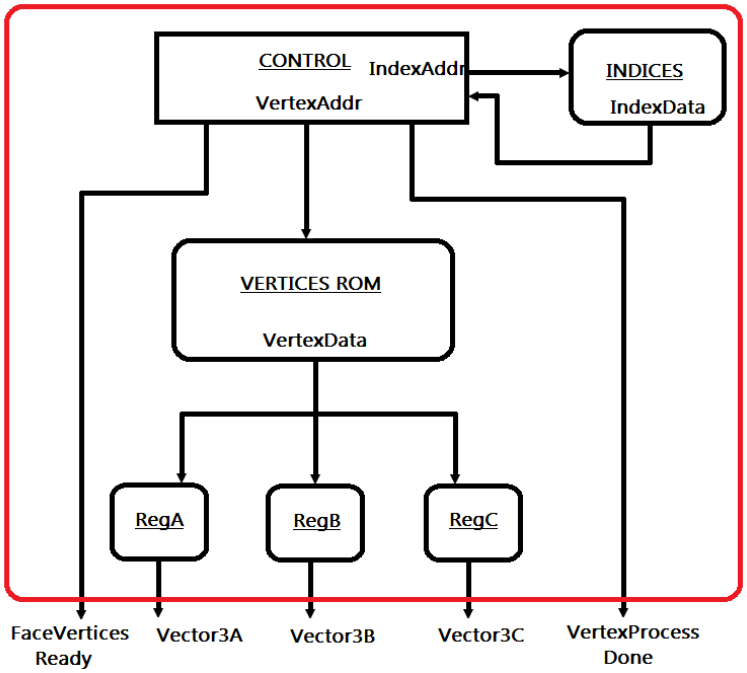


Figure 48: Block Diagram of FaceVertices component

Registers are used to store the vertex coordinates to be ready for output. The control component contains an FSM that is responsible for the right operation of the vertices production.

5.1.2 Processor Unit

This is the most complex and biggest component of the implementation. In this component all the operations are performed in order to create the frame. It is comprised of eight modules which all are controlled by the controller component. The mathematical operations were described in VHDL with the use of several basic IP cores.

Transformation Component This component performs all the operations of the vertex processing stage. Specifically, it performs this multiplication:

$$[ProjectionMatrix] * [ViewMatrix] * [WorldMatrix]$$

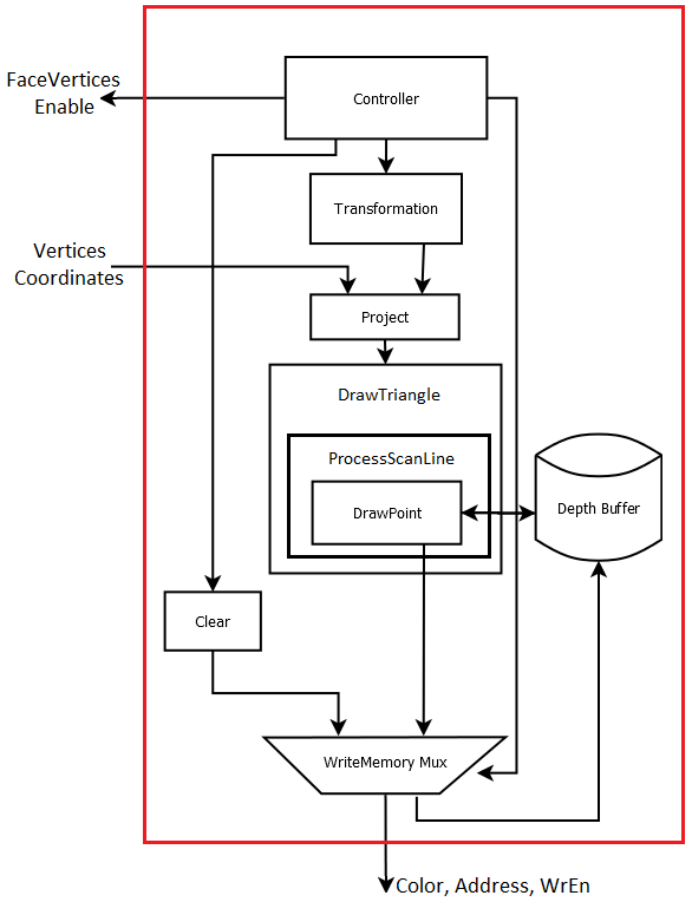


Figure 49: Block Diagram of Processor Unit

In order to compute the projection and view matrices, we implemented the perspective algorithm (Listing 2) and Look-at algorithm (Listing 1) mentioned earlier using high level synthesis (HLS) and placed the the appropriate RTL in Projection and View component respectively. Also, here are the input parameters we used as example in the project for View and Projection component:

$$cameraPosition = (0, 0, 5), cameraTarget = (0, 0, 0) \text{ and}$$

$$\text{fov} = 0.78, \text{ aspect} = 400/200, \text{ znear} = 0.01, \text{ zfar} = 1.0$$

These two matrices start to be computed in parallel when Controller enables

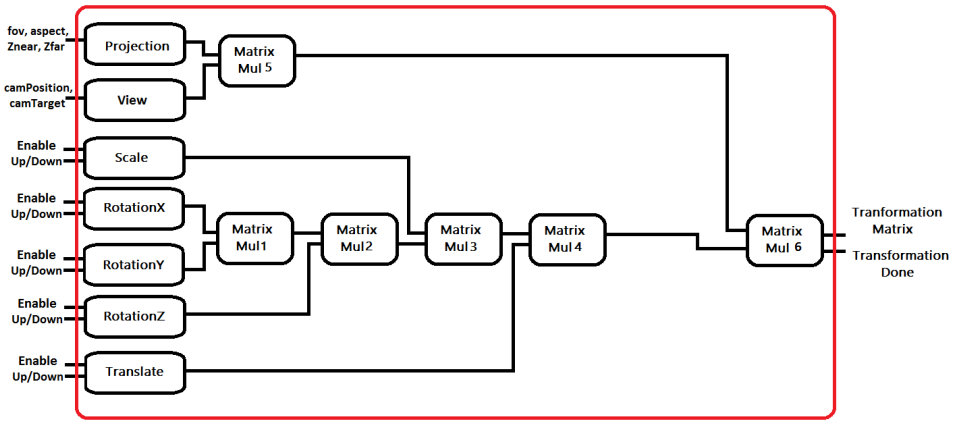


Figure 50: Block Diagram of Transformation Component

the Transformation component. After this computation, they are multiplied using the MatrixMul5 component. in this phase, all MatrixMultiplication components (MatrixMul in Figure 50) were implemented using high-level synthesis (Appendix 8.1). Also, the world matrix starts to be computed the same time with projection and view matrices. It is computed by multiplying Scale Matrix, Rotation Matrix and Translation Matrix. Because Rotation matrix comes from the multiplication below, it must be computed first.

$$[RotationMatrix] = [RotationXMatrix] * [RotationYMatrix] * [RotationZMatrix]$$

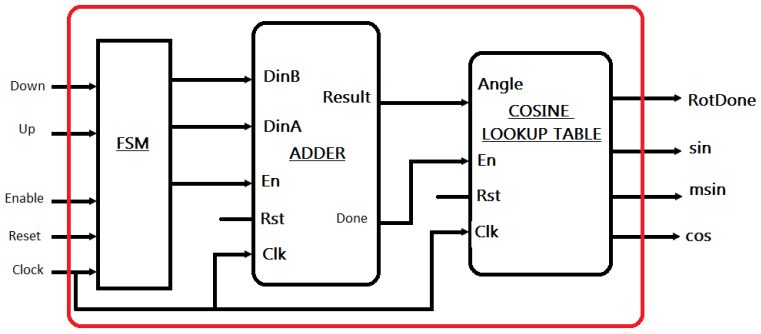


Figure 51: Block Diagram of Rotation Component

Each Rotation component that computes the separate rotation matrix, consists of two modules: an Adder and a Cosine Lookup table. The adder is initialized with zero and is increased or decreased depending the inputs. The output value from the adder is the angle θ and must be always in this range [0, 358] (angle is increased or decreased 2 degrees each time). Then, the Cosine Lookup table gives the appropriate positive cosine, positive sine and negative sine as output, depending always on the angle θ.

In the same way, Translation and Scale matrix are computed (without the cosine lookup table). The order of the matrix multiplications for the World Matrix is: 1. MatrixMul1, 2. MatrixMul2, 3. MatrixMul3, 4. MatrixMul4. World matrix is computed faster than the multiplication of projection and view, because the latters' matrices are more complex. So when MatriMul5 has finished, the result from MatrixMul4 is ready. The last step is to multiply these two matrices in order to get the final transformation matrix.

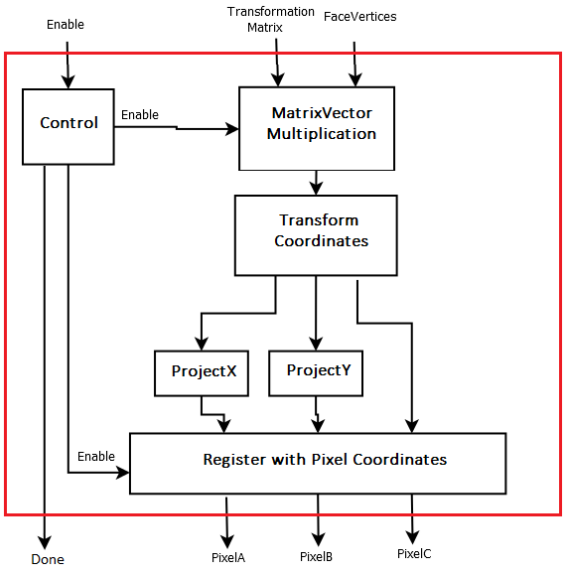


Figure 52: Block diagram of Project component

Project Component Project component takes some 3D coordinates and transform them in 2D coordinates using the transformation matrix. As it can be seen from the block diagram (in Figure 52), it is consisted of six modules.

In MatrixVectorMultiplication component (Appendix 8.2), we multiply the transformation matrix with the face vertices of a point of a triangle. The result is divided by w value in TransformCoordinates component (Appendix 8.3) and final x and y values are scaled to the viewport width and height in ProjectX and ProjectY components respectively. Because FaceVertices component gives each time the coordinates of the three vertices of a triangle we need to do this process three times for each vertex. That is why this register exists, in order to store the final coordinates of every vertex.

MatrixVectorMultiplication and TransformCoordinates components were implemented using high-level synthesis.

DrawTriangle, ProcessScanLine, DrawPoint and DepthBuffer Components These four components are responsible for the rasterization of a triangle depending on the pixel coordinates that come out from project module. ProcessScanLine is a part from DrawTriangle and DrawPoint is a part from ProcessScanLine. In other words, DrawTriangle calls ProcessScanLine component multiple times in order to raster all the lines of the triangle and ProcessScanLine calls DrawPoint component many times in order to draw all the points of each line.

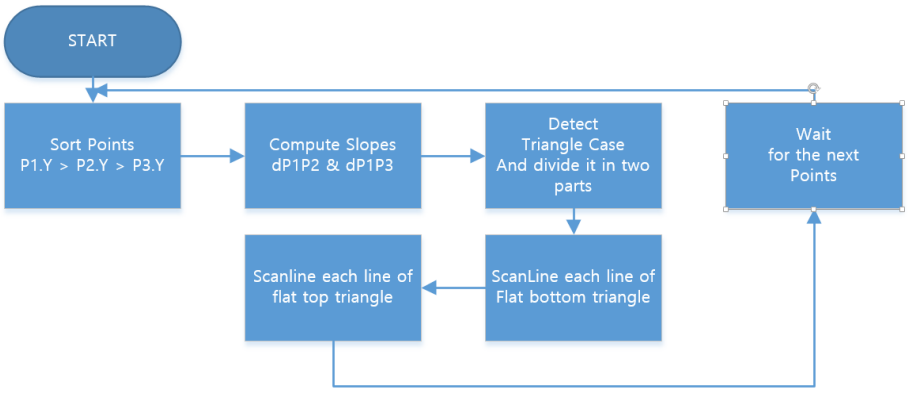


Figure 53: Flowchart of DrawTriangle component in Rasterization Phase

First of all, in the **DrawTriangle** component, we sort the points that are given as inputs from project component in order to have them always in the same order - with P1 always up, and then P2 between P1 and P3. Then, we detect which case of a triangle we have (with P2 on the right or P2 on the left) and divide the triangle into two parts, to a flat bottom and a flat top

triangle. Now for every line of these triangles, the appropriate coordinates of every line are given to ProcessScanline component in order to raster the lines.

The **ProcessScanLine** component in order to draw a line has to find the sarting X and Z, and ending X and Z of it. At first, we compute the gradient of the triangle edges (left and right) thanks to current Y. Then, we interpolate the coordinates with the given gradients and we have the values we want to draw the line. After that, we use the DrawPoint component in order to write every point of a line on the frame buffer.

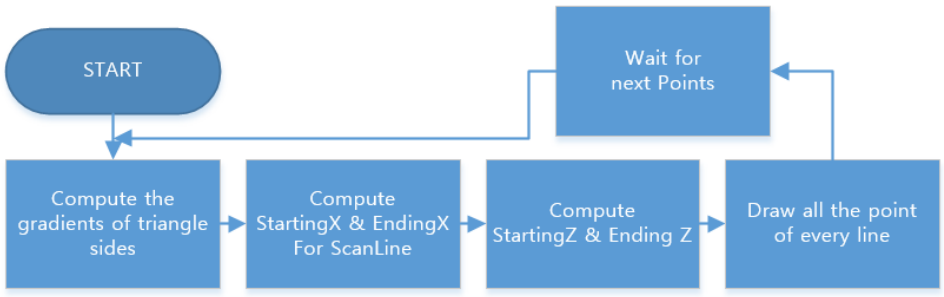


Figure 54: Flowchart of ProcessScanLine component in Rasterization Phase

The **Drawpoint** component, before writing the appropriate value to the frame buffer, does the clipping operation by checking if the current point of the triangle is visible on screen. Then, it checks the value of the depth buffer in current's x and y coordinates in order to identify if another object exists in front of it in the scene. Depending on that, the color value is written on the frame buffer and the z value in the depth buffer. Finally, **Depth buffer** is the memory in which we store the z value of a point which exists in the same x, y coordinates in frame buffer. Frame buffer and depth buffer have the same size.

Clear component This component is responsible for clearing the frame and depth buffers. When it is called, it writes black color to all addresses of the frame buffer and a great value to the addresses of the depth buffer. This is the initialization of the buffers.

Write Memory Multiplexer This multiplexer is responsible for allowing which component (Clear or DrawPoint) to write to frame and depth buffer. It decides depending on the input it gets from the controller component. In

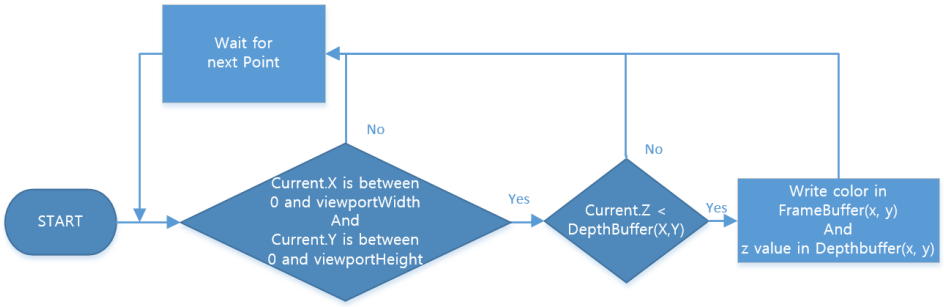


Figure 55: Flowchart of DrawPoint component

the beginning of each frame, buffers are cleared, so multiplexer allows to clear component to write into buffers. After that, drawpoint is the component that writes into buffers until current frame is completed. And this operation continues.

Controller Controller is a Frequent State Machine (FSM) and the main module of processor unit which gives the right values to the previously referred components in order to execute the pipeline successfully.

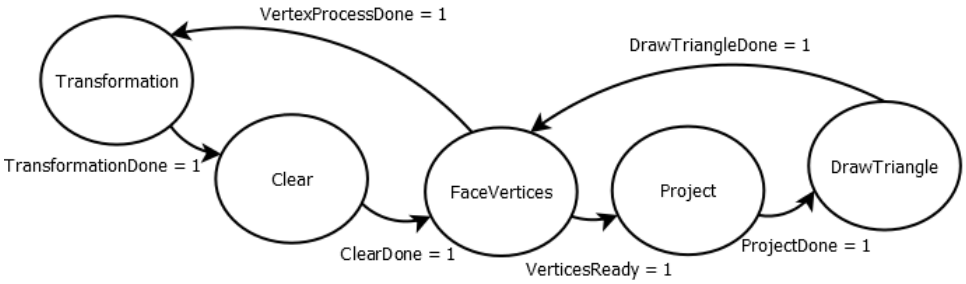


Figure 56: Controller FSM

At first, Controller enables the transformation component. When it finishes, enables the Clear component to clear the buffer and then it enables the FaceVertices component to get the first vertices that describe a triangle. When vertices are ready, Controller enables the Project component and then the DrawTriangle. When this process finishes, Controller enables the FaceVertices again to get the next three vertices and the same process continues. If FaceVertices component has given all the vertices of the mesh (VertexProcessDone = 1), Controller enables Transformation component and the

whole process is repeated from the beginning. In Figure 56, we can see the execution order of the components.

5.1.3 Dual-Port RAM

Dual-Port RAM includes the frame buffer. This memory is dual port because display unit needs always to read color values and processor unit needs to write whenever it has to. Because the viewport of the frame we used in this project is 400x200, memory's size is $400 * 200 * 8bits$ (the color depth is 8 bits). In order to translate the x, y coordinates in addresses to the buffer we use a component which is called CoordsToAddr. Our dual-port RAM

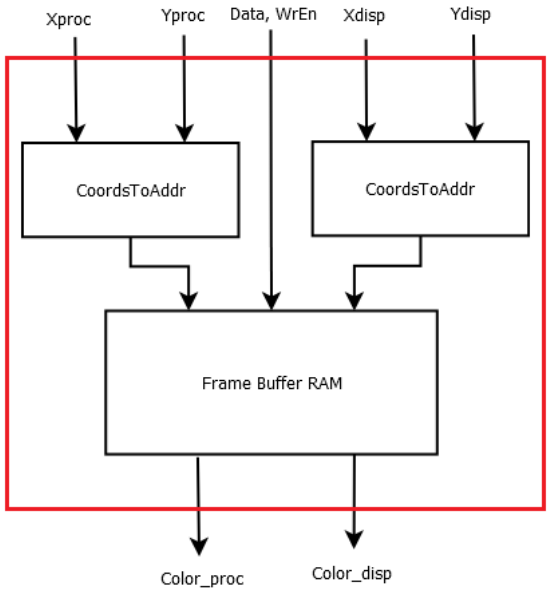


Figure 57: Block diagram of Frame Buffer

consists of two same frame buffers allowing full memory bandwidth to both the processor display unit. When processor unit writes on the first frame buffer, display unit reads from the second one. When processor unit finishes writing the mesh, it starts writing to the second buffer and display unit reads from the first one. This swap process continues.

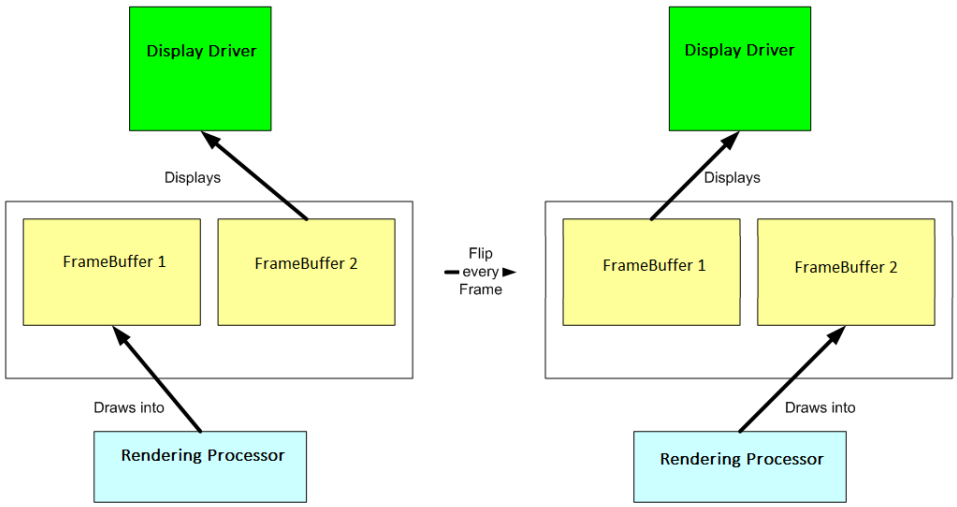


Figure 58: Swap process of Frame buffers

5.1.4 Display Unit

The final component of the implementation is the display unit. It includes a module called ImageBuilder, which is responsible for reading the color values of every address from framebuffer serial and gives the screen the appropriate color for every pixel at the right time. First of all, it is necessary to compute the pixel clock. Depending on VGA timings and specifications, the screen resolution we used in this project (1024x720) has 806 lines in a "whole frame" and the size of a whole line is 1344 pixels. That size includes the porches, sync and visible area (parameters for vga interface)[31]. So, we need to display 1344∗806=1083264 pixels on each frame. Now, in order to find the suitable pixel clock, we multiply the total amount of pixels with 60 (Hz) Industry standard timing.

$$1.083.264 pixels * 60 Hz = 64.99 MHz$$

That is why we selected to use 65MHz as the pixel clock's frequency.

Imagebuilder component sends binary color values and signals for horizontal and vertical synchronizing to an digital to analog converter that exists in KC705 board called ADV7511 [30]. Then this analog device sends the appropriate values to the HDMI transmitter. In order to get access to the converter, we have to use I2C bus. This part of VHDL code was implemented by Mike Field (Appendix 8.4). I2C bus is used to configures the

chip once, soon after at power on, and from then on it does nothing. Because the interface to the ADV7511 is running at Double Data Rate (DDR), data is transferred on both the rising edge and falling edge of pixel clock. So, the rest design has to tick twice as fast. That is the reason why we used 130 MHz as the main frequency of the system.

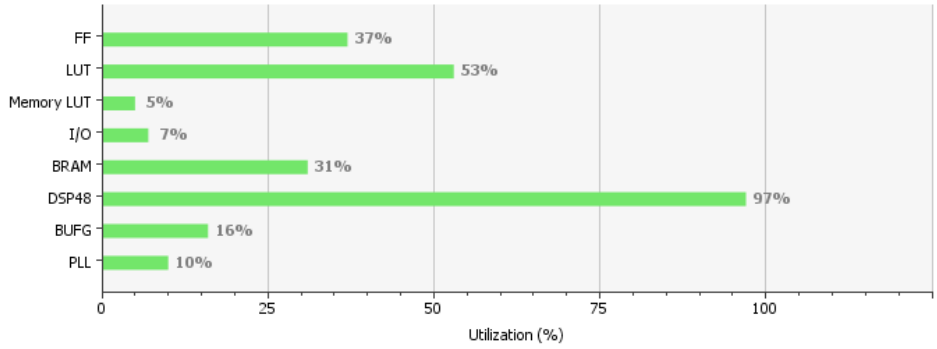


Figure 59: Resources utilization in Vertex Processing and Rasterization phase

As we can see in Figure 59, the implementation of Vertex Processing and Rasterization phase uses about the half of available Look-Up tables, 31% of available blockram and 97 % of DSP slices. The great amount of DSP slices is logical because we use numerous floating point IP cores for all the transformations. In the next phases, we will see that some replacements had to done to decrease the DSP's in order to fit the logic in the device.

5.2 Shading Phase

In the second phase of the implementation, we added shading to the already implemented in previous phase rendering pipeline. The two types of shading that were implemented are Flat and the Gouraud Shading. In both shading types, we had to add in FaceVertices component one further memory that keeps the normals per vertex of the triangles. So FaceVertices component has as outputs the normals per vertex alongside the coordinates of each vertex.

In Project component, apart from transforming the vertex coordinates into 2D space using MatrixVectorMultiplication and TransformCoordinates components, we needed also the world matrix and the normals in the 3D world.

So we transformed them too. We added these two transformations, because their output values are necessary in DrawTriangle component.

Due to limited resources of the device, we did not use the HLS RTL inside MatrixVectorMultiplication component. It needed more DSP slices from the available, so we implemented a new MatrixVectorMultiplication component that needed less resources. But it was much slower. In this phase, we faced that problem because we added more complexity to other components to implement shading.

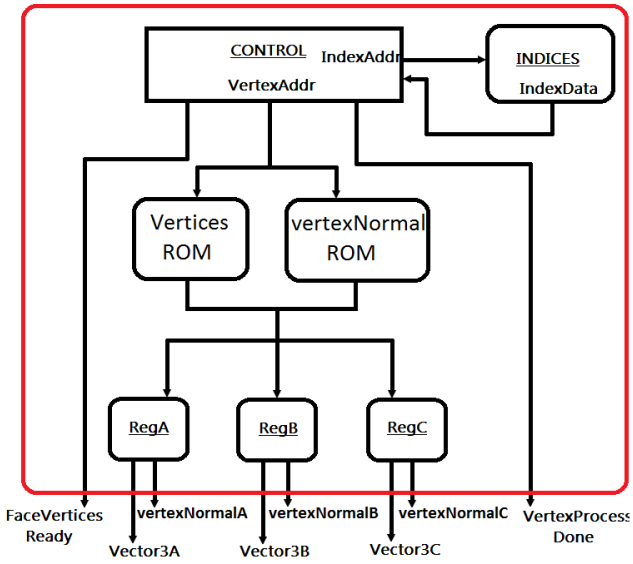


Figure 60: FaceVertices Component in Shading Phase

The MatrixVectorMultiplication component multiplies a matrix with a vector. As we know from linear algebra, if we want to multiply M[4][4] matrix with N[4] vector, here are the values that equal to K[4] vector.

$$K_1 = (M_1 1 * N_1) + M_1 2 * N_2 + M_1 3 * N_3 + M_1 4 * N_4$$
$$K_2 = (M_2 1 * N_1) + M_2 2 * N_2 + M_2 3 * N_3 + M_2 4 * N_4$$
$$K_3 = (M_3 1 * N_1) + M_3 2 * N_2 + M_3 3 * N_3 + M_3 4 * N_4$$
$$K_4 = (M_4 1 * N_1) + M_4 2 * N_2 + M_4 3 * N_3 + M_4 4 * N_4$$

So in this component, we compute the four values of K vector one by one and store them in a register. When we have computed all of them, the K vector is ready.

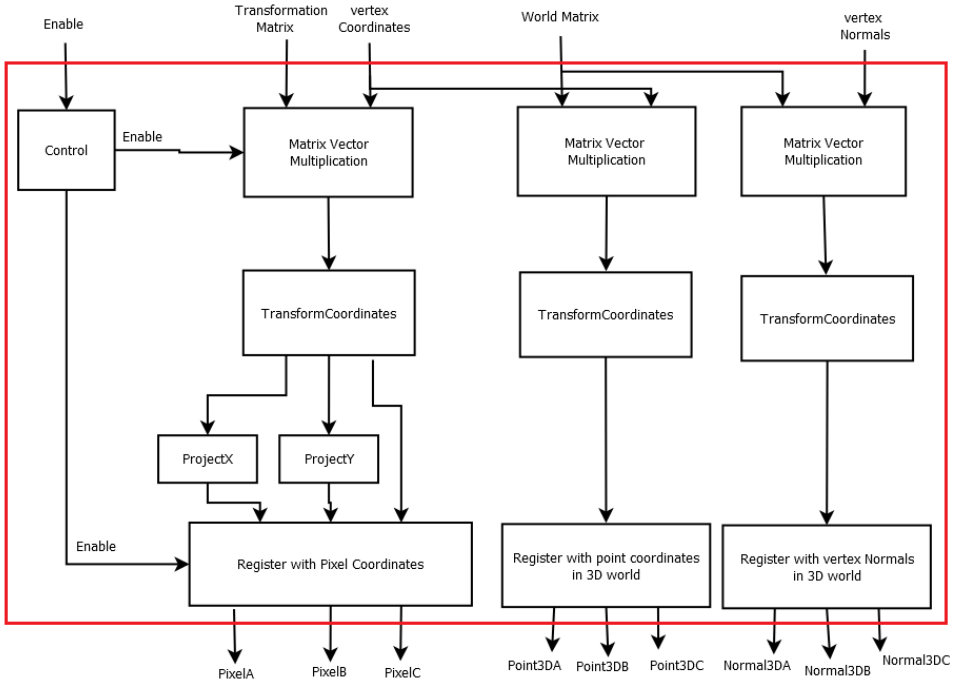


Figure 61: Project component in Shading Phase

The block diagram of MatrixVectorMultiplication component can be seen in Figure 62.

In Transformation component we faced also the same problem with resources. We removed two MatrixMultiplication components were implemented using HLS and replaced them with two other components implemented with the same logic as we saw in Figure 62. But because this time we want 16 values for the K matrix, latency is about 640 cycles. The components that we replaced were MatrixMul4 and MatrixMul5 (see in Figure 50. (The rest project's implementation uses these new components, MatrixMultiplication and MatrixVectorMultiplication, that we created without HLS)

The further changes that we had to do in Transformation, DrawTriangle and ProcessScanLine components are different for Flat and Gouraud shading so it will be discussed separately.

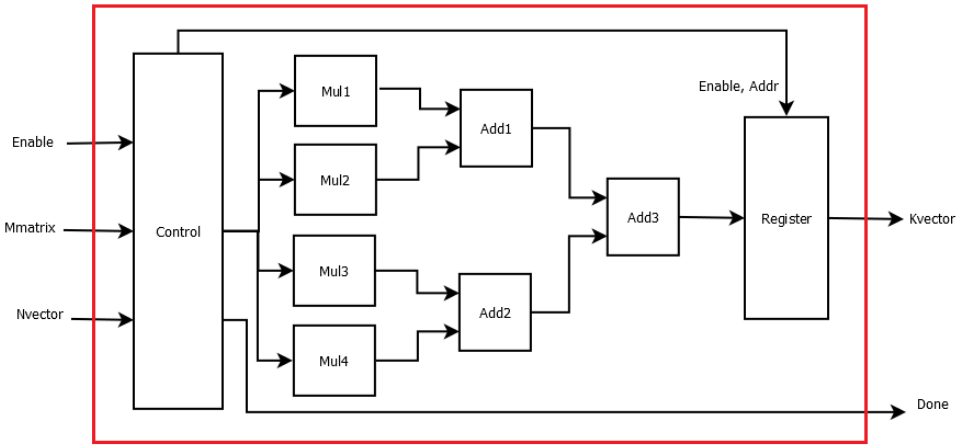


Figure 62: MatrixVectorMultiplication component in Shading Phase (Latency = 160 cycles)

5.2.1 Flat Shading

In this type of shading, DrawTriangle component includes some computations that had to be done in order to find the intensity of the color of the face. Intensity is a value between 0 and 1.

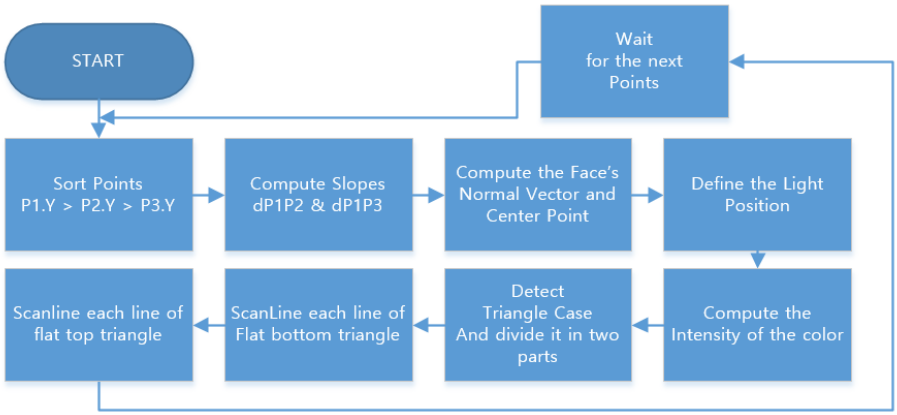


Figure 63: Flowchart of DrawTriangle in Flat Shading

So, first of all we have to find the face normal vector which is the average normal between each vertex's normal (Appenix 8.5). That is why we needed to compute the vertex normals in 3D world in Project component. Then

using the point coordinates in 3D world, we compute the center point of the face (Appendix 8.6). After declaring the light position (in our project light source is in this coordinate (2, -2, 10)), we compute the cosine of the angle between the light vector and the normal vector of the face using dot product (Appendix 8.7). The cosine value is the intensity of face's color. It is multiplied with the initial color and the result is being written later into the FrameBuffer.

The components that compute the face normal vector, the center point of the face and the dot product were implemented using high-level synthesis.

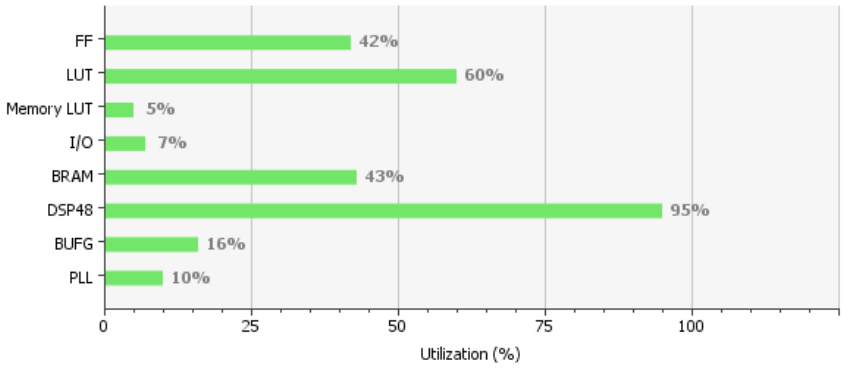


Figure 64: Resources utilization in Flat Shading phase

After the removal of HLS components and their replacements with the new ones described previously, we decreased the number of DSP slices successfully. Comparing to the previous phase's table, we can see that in this phase more resources are used due to the greater complexity of the flat shading implementation.

5.2.2 Gouraud Shading

In Gouraud Shading, DrawTriangle component does not include the same computations as in Flat Shading. Here, we compute only the cosine of the angle between the light vector and the normal vector for all vertices of atriangle separately. So, we have three intensity values of color for each vertex. In ProcessScanLine component now, we have to find the intensity value of the color of each pixel. We did this by interpolating the vertex normals first in the sides of the triangle and then in the line we are rasterizing. The final interpolated value is the intensity of the color. Intensity is multiplied with

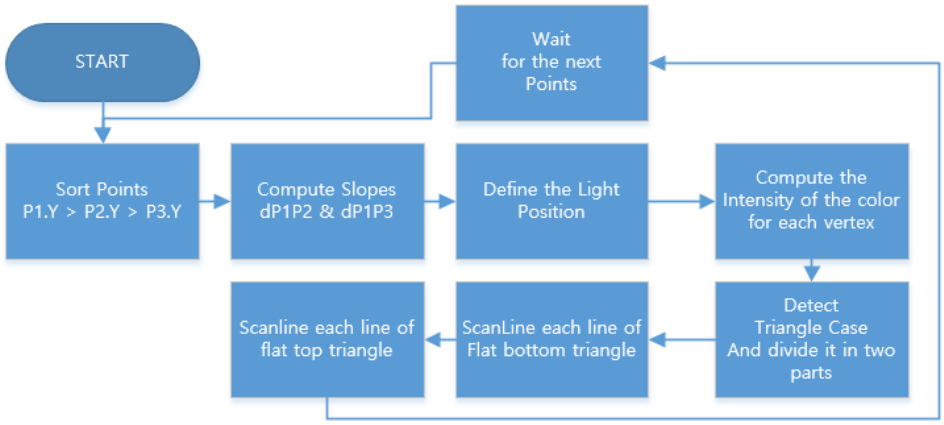


Figure 65: Flowchart of DrawTriangle in Gouraud Shading

the initial color and the result is being written into the FrameBuffer later using the DrawPoint component for every pixel.

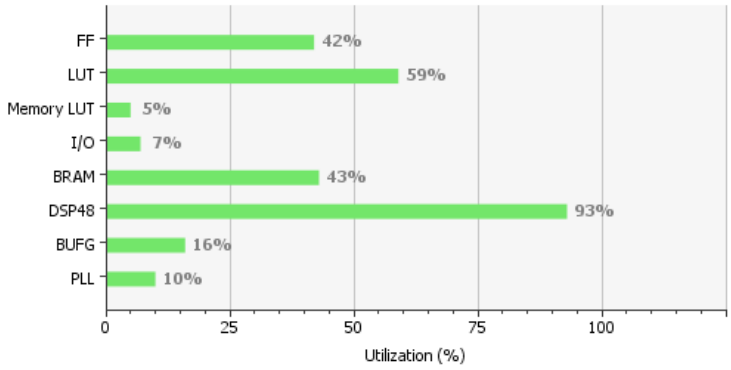


Figure 66: Resources utilization in Gouraud Shading phase

In Figure 66, we can see that this implementation uses a little less resources than Flat Shading. We expected that, because in this phase we do not use the two implemented in HLS components for the center point of the face and the face normal. This means less DSP slices and logic cells.

5.3 Texture Mapping Phase

In Texture Mapping phase, we used the Gouraud shading implementation and added some extra logic. At first, we needed a new memory to contain the texture coordinates of each vertex.

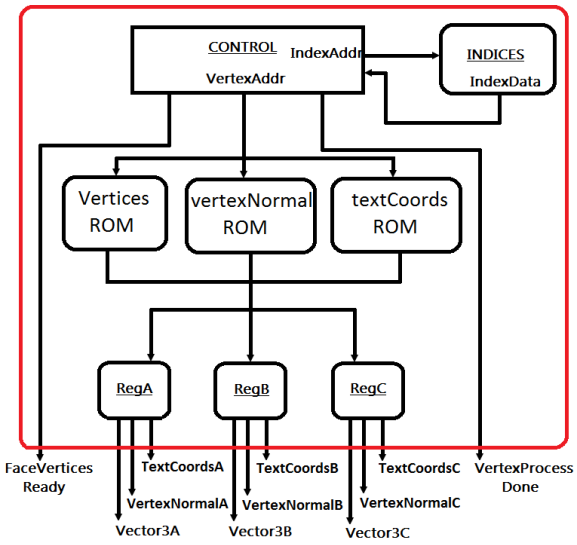


Figure 67: FaceVertices component in Texture Mapping Phase

For this reason, in FaceVertices component, we created the textureCoordinatesROM. These texture coordinates have been added as outputs to FaceVertices component. Texture coordinates are numbers between 0 and 1. Project and DrawTriangle components have remained the same.

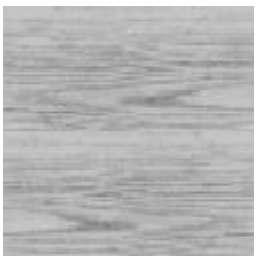


Figure 68: Wood Grayscale Texture (64x64)

The only change we did was the addition of some outputs with the texture coordinates of each vertex. All the computations for the final color of every

pixel depending on the texture are being done in ProcessScanLine component.

In ProcessScanLine component, a new memory called TextureROM was created to contain the texture color values. The texture image we used for the demonstration of the example mesh is a simple wooden surface with dimensions: 64 pixels width and 64 pixels height. Using Matlab code, a COE file was created that describes the texture colors with 8bit values. It contains only grayscale colors.

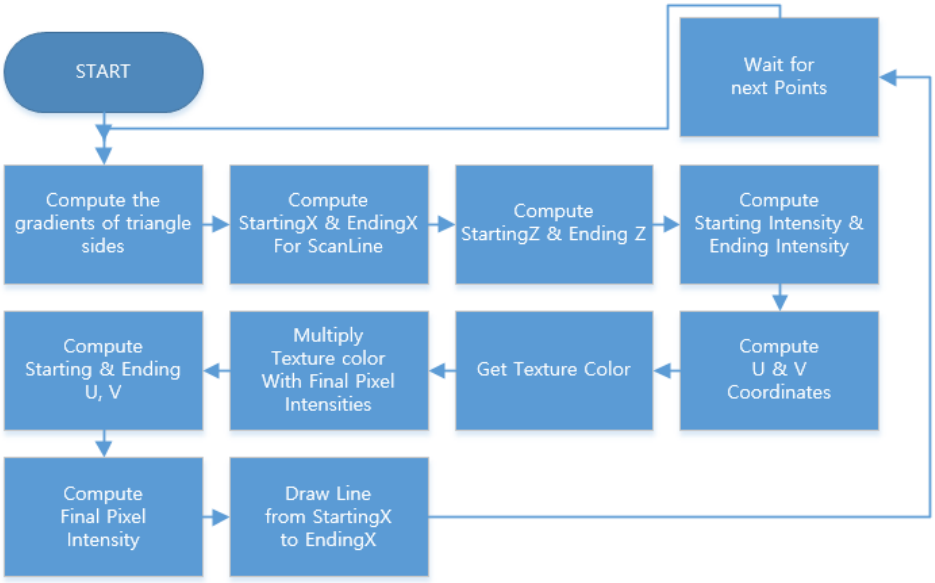


Figure 69: Flowchart of ProcessScanLine in Texture Mapping Phase

In ProcessScanLine component, after the interpolation of normals on Y axis, texture coordinates are also interpolated on the same Y axis in order to compute the Starting U and Starting V values. Moreover, another two values, U and V, have to be computed for every pixel of the line using interpolation again. These values are multiplied with width and height of the texture image respectively, in order to get the corresponding pixel color of the texture from TextureROM. Knowing now the right texture color, we multiply it with the intensities values. The result is the color value that will be later written into FrameBuffer using DrawPoint component.

As we expected, it can be seen in Figure 70 that we use more resources

than previous phases. Due to the use of more floating points for the computations and the new memories that added, the utilization of DSP slices, Flip-flops and LUT's has been increased.

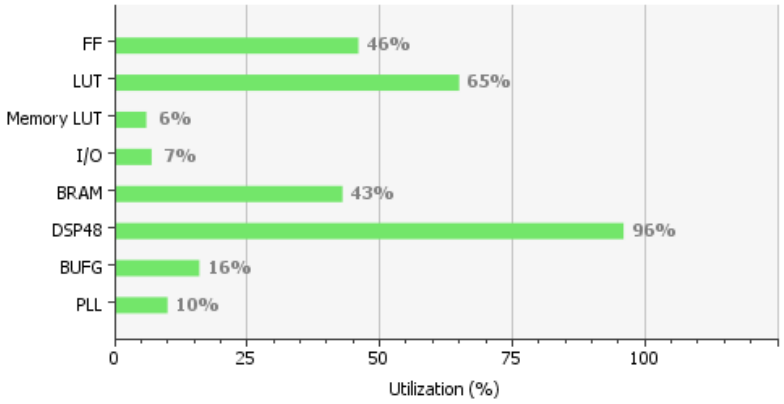


Figure 70: Resources utilization in Texture Mapping phase

5.4 3D Rendering Effects

For the implementation of the 3D rendering effects we used the rasterization phase and added there the appropriate components.

5.4.1 Perlin Noise Mapping to Ramp Texture

In this effect, the only component that we had to change, in comparison to the rasterization phase, is the ProcessScanLine component. After the

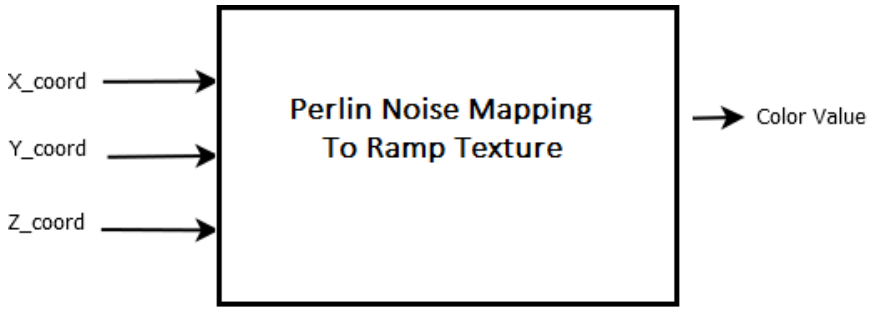


Figure 71: Perlin Noise Mapping to Grayscale Ramp Texture

implementation of the Perlin Noise function using high-level synthesis (Appendix 8.8), we used the exported component as the latest component in ProcessScanLine before calling the DrawPoint. This component gets the coordinates in 3D space of the point that is going to be drawn and gives a decimal value between 0 and 1 as output. This value is mapped with an 1D texture 1x256 pixel to get a color value, which will be passed alongside with the pixel coordinates to DrawPoint component. Despite the fact that we use only grayscale colors to these pixels, any color we want can be put in this 1D texture in order to have a real color mapping (i.e. colors from blue to red). In Figure 72, we can see an example of a grayscale ramp texture (1D texture).

Figure 72: Grayscale Ramp Texture

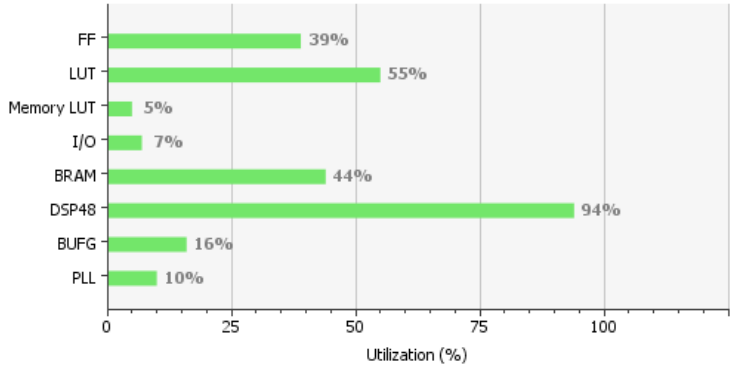


Figure 73: Resources utilization in Perlin Noise Mapping to Ramp Texture effect

In Figure 73, we can see that this implementation does not utilize more resources than the previous implementations.

5.4.2 Particle System

In Particle System implementation we removed the FaceVertices component and created a new one called ParticleVertices. This component is responsible for generating the vertices of the particles (triangles) and also updating

their coordinates in order to see them moving per frame. The ParticleVertices component consists of three modules: Triangle, MapColor and ParticleSystem.

- The *Triangle* module includes memories that keep the vertices, velocity in each axis and lifespan of the particles.
- The *MapColor* module maps the lifespan of the particles with a grayscale ramp texture in order to generate their color. With this mapping, particles look like disappearing as they reach their lifespan.
- The *ParticleSystem* module updates the characteristics of the particles and writes them back to memories in Triangle module.

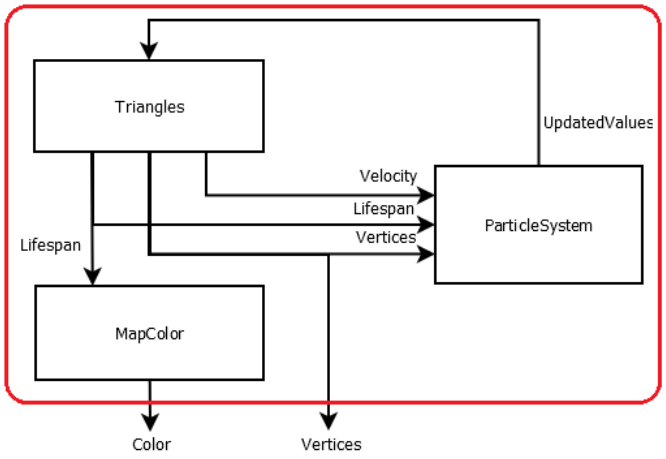


Figure 74: ParticleVertices component

ParticleSystem (Appendix 23) and MapColor (Appendix 8.10) modules were implemented using high level synthesis.

All particles begin the emission from the same coordinates in the 3D space and depending their velocity in each axis they are moving. As particles reach their maximum lifetime, their color changes from white to black in order to give realism to the scene. In Figure 76, we can see the steps that ParticleVertices component follows.

As mentioned in the previous section, we implemented a second version

of this effect called Particle System with Repeller. Both versions have the same components, but we made some changes in Triangle and PartcileSystem component. In Triangle component, we define in the first lines the vertex coordinates of the repeller object. In Figure 75, we can see the world coordinates of the repeller object we used in this implementation.

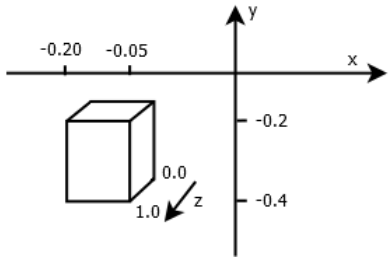


Figure 75: Repeller Object (Cube) in World Space

Now in ParticleSystem component, we made some changes in order the particles to react when colliding with the object. Before updating the vertices of the particle, we check if the particle collides with the cube. If it collides on the top face, it bounces up and then continues falling down. If it collides on the right face of the cube, it changes direction to the right. Particles change direction due to an instant counter force we added to the particle when it hits the object. This updated ParticleSystem component was implemented using high level synthesis too (Appendix 8.11).

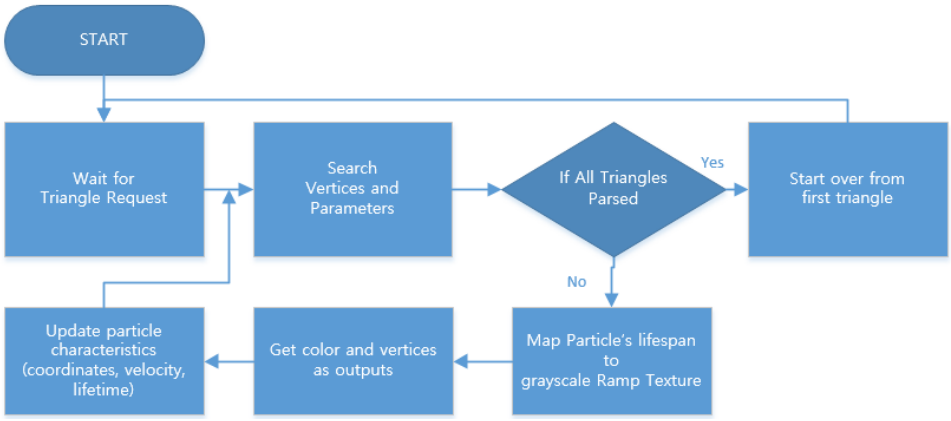


Figure 76: Flowchart of ParticleVertices component

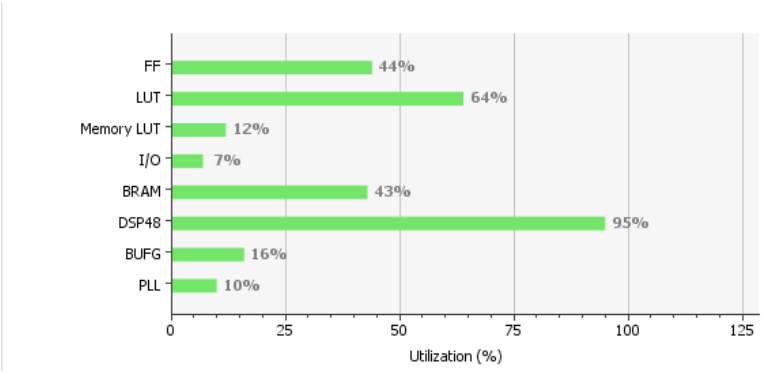


Figure 77: Resources utilization in Particle System effect

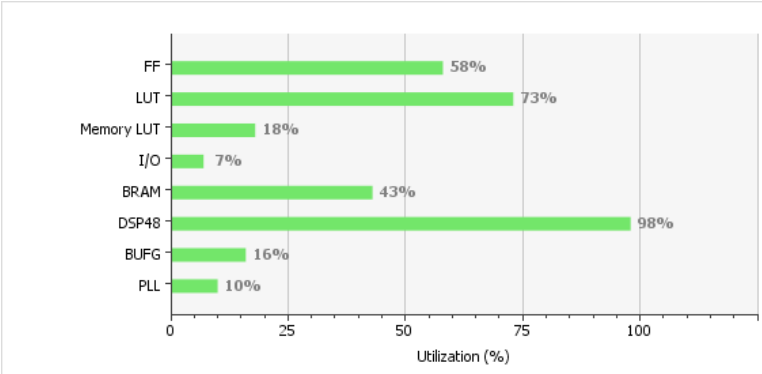


Figure 78: Resources utilization in Perlin System with Repeller effect

In Figures 77 and 78, we can see that the implementation of Particle System with Repeller uses more resources than the implementation without the repeller object. We expected that because we need extra logic.

5.4.3 Displacement Mapping using Perlin Noise

In this last rendering effect, we do not use any memory that holds vertices or other parameters that help to render the terrain. In FaceVertices component, a new module called TerrainVerticesGenerator was implemented in order to produce vertices that describe a horizontal grid (terrain). While x and z values are produced according to the grid, y values of these vertices are chosen using the Random component we implemented using high-level synthesis.

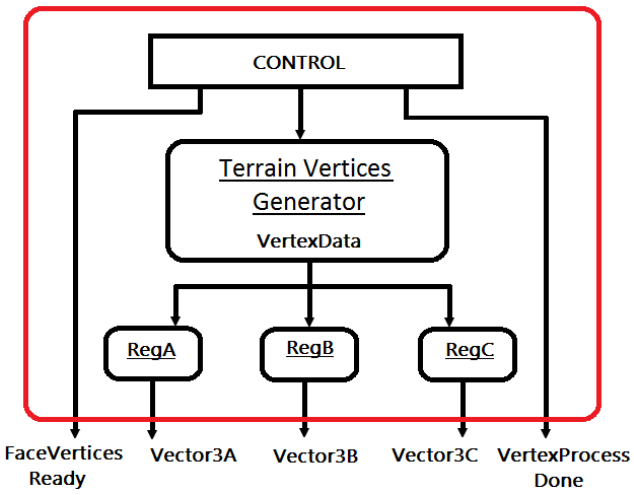


Figure 79: FaceVertices component in Displacement Mapping Terrain using Perlin Noise effect

Since system's startup, a counter is running from 0 to 4095. When user pushes the center button of the D-pad on the board, a value is chosen in this range (0 - 4095).

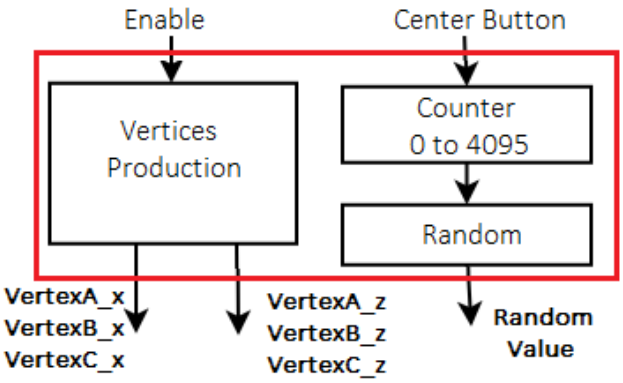


Figure 80: TerrainVerticesGenerator component

This value is the input to Random component and the output of this component is a random value used to compute later the y value of the current triangle's vertices. Knowing now the coordinates of the vertices and having a random number, we want to give to y of each vertex a 'pseudorandom'

value in order to create a terrain with multiple levels. As we do not want these values to be irrelevant to their 'neighboring' vertices, we use Perlin Noise function once again. The Perlin Noise component uses x and z value from vertices coordinates as inputs and takes the random value was chosen from the counter as the y input. The output value of the PerlinNoise component is used as the final y value of the vertex that is going for projection alongside with the initial x and y coordinates.

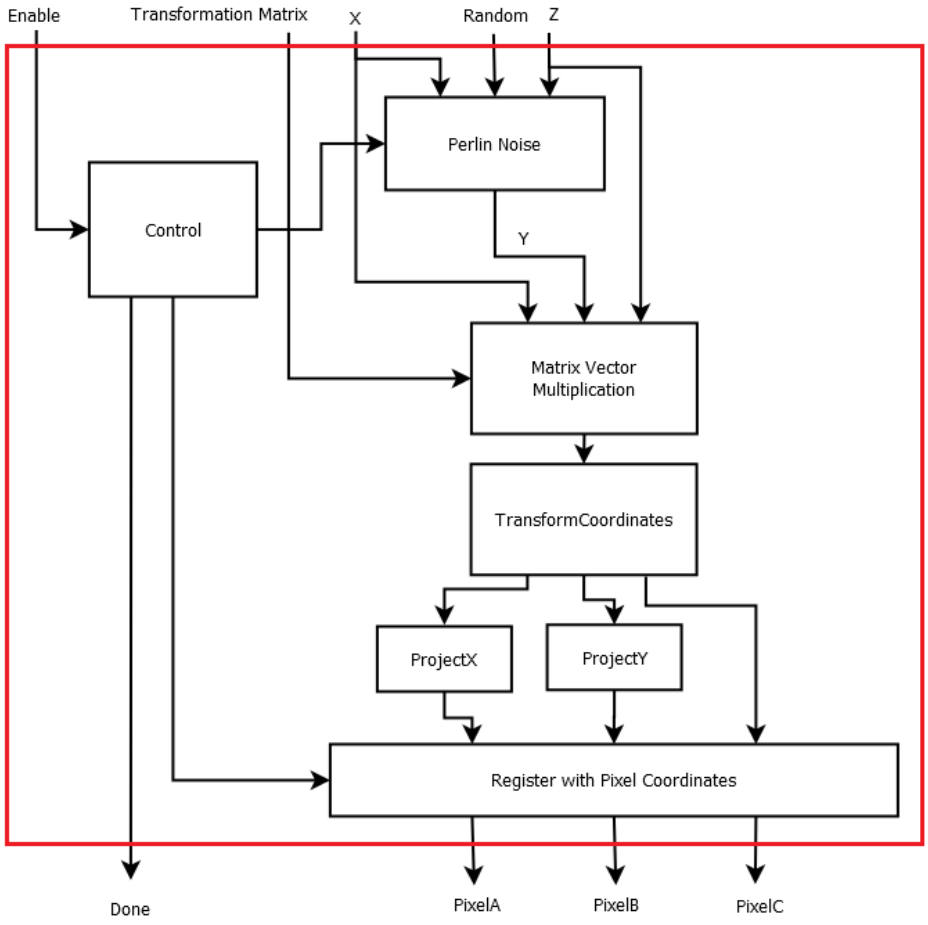


Figure 81: Project component in Displacement Mapping Terrain using Perlin Noise effect

The grid we create consists of 9600 vertices and 3200 triangles. These triangles describe the whole grid. In Figure 82, we can see an example of the

grid and how the triangles create it.

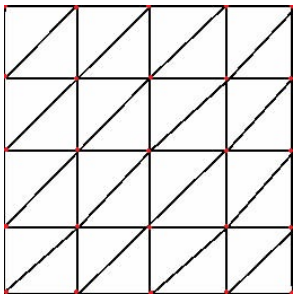


Figure 82: Mini version of the grid (seeing from top)

Also, in Figure 83, we can see the resources utilization for this implementation. As we expected, we do not need much more resources than the previous implementations.

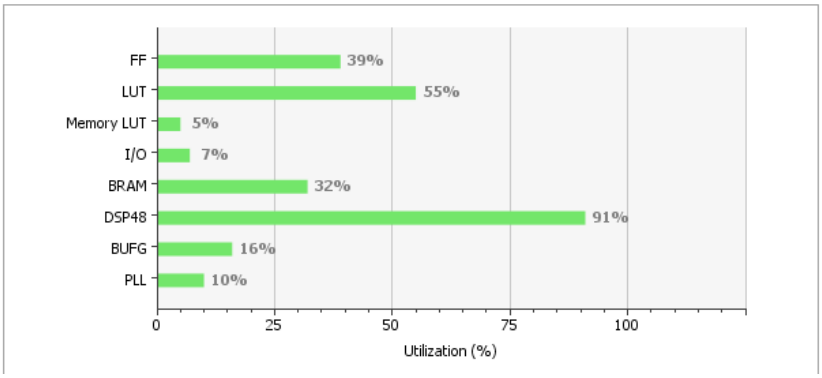


Figure 83: Resources utilization in Displacement Mapping Terrain using Perlin Noise effect

6 System Evaluation

To verify the functionality and efficiency of the system described in the previous sections, we carried out a number of tests. These tests measured the performance of the graphics pipeline's implementation phases (Rasterization, Flat and Gouraud Shading) and the rendering effects in frames-per-second (FPS), both in software and hardware. These phases were already developed in Javascript and we just executed them using the browser. Also, it must be stated that these demos are CPU accelerated in single threaded implementations.

Software demos were launched on an AMD FX-8120 8-Core 3.1GHz computer, with 8GB of RAM and Windows 8 Professional 64-bit. Due to the huge speed difference between the computer and the embedded platform (which ran at 130MHz), FPGA was expected to perform much slower than software. Hardware demos were run on the XC7K325T-2FFG900C Kintex-7 FPGA and KC705 development board using HDMI cable to display the results on a 19" LG Flatron LCD TV-Monitor.

6.1 Vertex Processing and Rasterization

In the first demo, we tested the performance of the rasterization phase. This demo used our implementation to draw a white cube which is being rotated in all axes. Here is a caption of this test running in software.

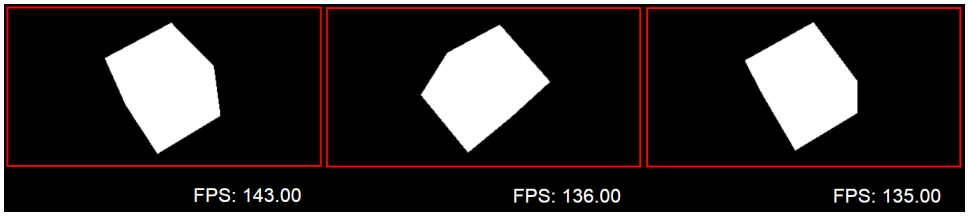


Figure 84: Vertex Processing and Rasterization phase running in software

As we can see in Figure 84, the average performance in software was about 140 FPS.

Next, the same test was launched in FPGA, seen in Figure 85. We can see that as more triangles are drawn the performance decreased. However, we can see that triangle primitives were drawn successfully. The frame rate peaked at 44 FPS and the average was about 39 FPS.

Figure 85: Rasterization phase running on the FPGA

6.2 Shading

The purpose of the Flat and Gouraud shading demos was to test the functionality of the shading implementations.

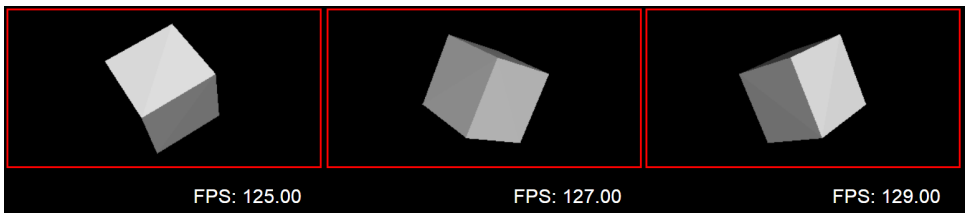


Figure 86: Flat Shading phase running in software

At first, we ran Flat Shading phase after setting the light position in (2, 2, 10). The cube was being rotated in all axes. As we can see in Figure 86, the average frame rate of Flat shading running in software was about 130 FPS. Then we tested the same demo on the FPGA and we can see the result in Figure 87.

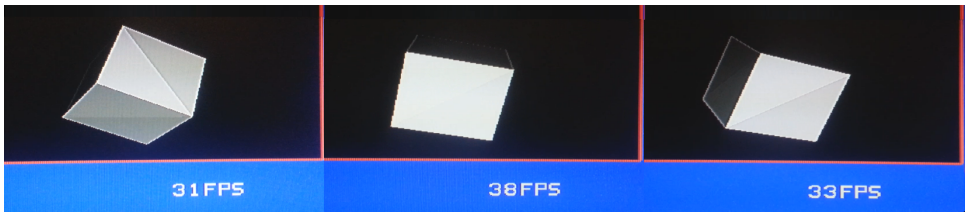


Figure 87: Flat Shading phase running on the FPGA

The frame rate on the FPGA peaked at 38 FPS and the average was about 33 FPS. In Figure 88, we can see the Gouraud shading running in software. The average frame rate was about 127 FPS. A screenshot of the same demo

Figure 88: Gouraud Shading phase running in software

can be seen running on the actual graphics hardware FPGA implementation in Figure 89.

Figure 89: Gouraud Shading phase running on the FPGA

Gouraud shading algorithm was also implemented successfully on the FPGA. The frame rate peaked at 36 FPS and the average was about 33 FPS.

6.3 Texture Mapping

The fourth demo included the texture mapping phase. As mentioned in the previous section, we used an image of wood as texture for the faces of the cube. In Figure 90, we can see the demo running in software.

Figure 90: Texture Mapping phase running in software

The average frame rate was about 40 FPS. A screenshot of the same demo can be seen running on the actual graphics hardware FPGA implementation in Figure 91.

Figure 91: Texture Mapping phase running on the FPGA

After seeing the results, we can say that texture mapping was implemented successfully. The frame rate peaked at 33 FPS and the average was about 28 FPS.

6.4 Perlin Noise Mapping to Ramp Texture

Our first 3D rendering effect, included a demo with the same cube as we used for the graphics pipeline's phases. We expected low performance on this test, as the perlin noise component takes many cycles in order to give output. In Figure 92, we can see the demo running in software.

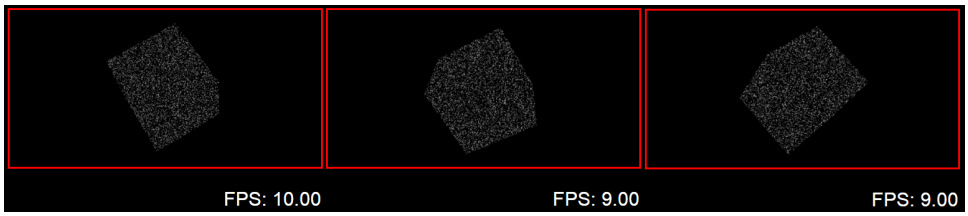


Figure 92: Perlin Noise Mapping to Ramp Texture running in software

The average frame rate was about 10 FPS in software. In Figure 93, we can see the demo running on the FPGA.

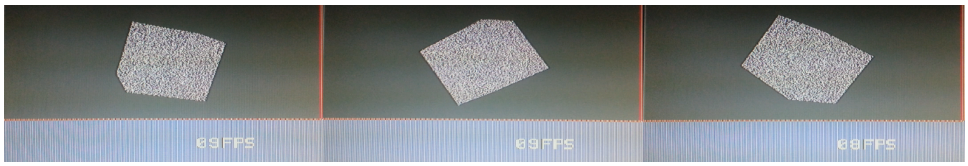


Figure 93: Perlin Noise Mapping to Ramp Texture running on the FPGA

The result is what we expected. A pseudorandom texture has been applied to the faces, but we see wrong colors on the screen. We assumed that Vivado

may change the color signal in order to optimize the final circuit. The frame rate on the FPGA peaked at 10 fps and the average was about 8 FPS.

6.5 Particle System

In our particle system test, we expected high performance because the graphics processor has to render very small triangles. In Figure 94, we can see a screenshot of the particle system running in software with average frame rate 180 FPS.

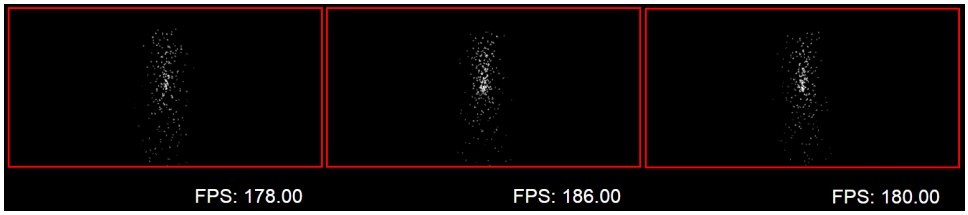


Figure 94: Particle System running software

In Figure 95, we see the hardware implementation on the FPGA.

Figure 95: Particle System running on the FPGA

The average frame rate was about 220 FPS.

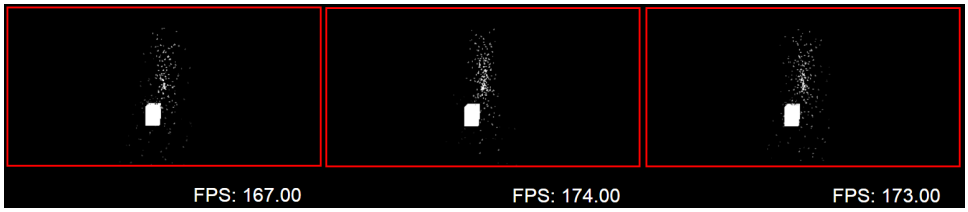


Figure 96: Particle System with Repeller running in software

In Figure 97, we can see the particle system with repeller running in software. The average frame rate in software was about 170 FPS. In Figure 95, we can see the hardware implementation on the FPGA.

Figure 97: Particle System with Repeller running on the FPGA

In this demo we had an average frame rate about 193FPS.

6.6 Displacement Mapping using Perlin Noise

The last demo we ran was about the Displacement Mapping using Perlin Noise. In Figure 98, we can see the demo running in software.

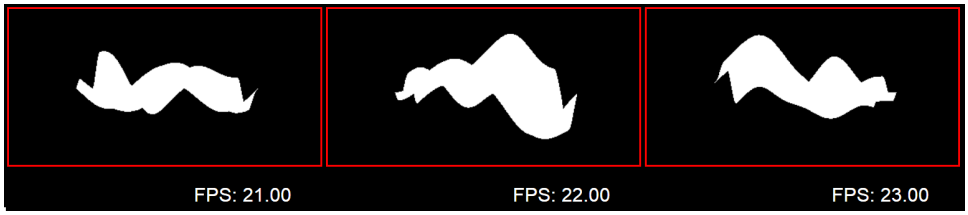


Figure 98: Displacement Mapping using Perlin Noise running in software

The average frame rate in software was about 22 FPS. In Figure 99, we can see the hardware implementation on the FPGA.

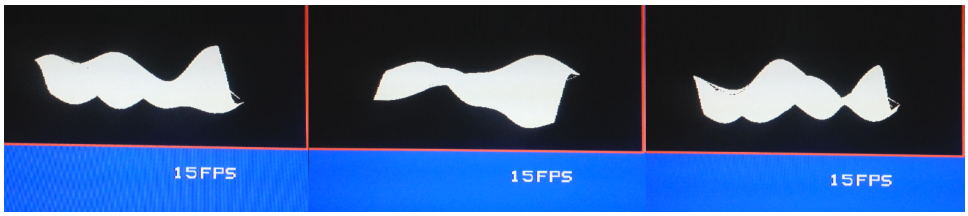


Figure 99: Displacement Mapping using Perlin Noise running on the FPGA

After seeing the results in Figure 99, we can say that our last rendering effect was implemented successfully too. By clicking the center button of the D-pad on the board, we got random terrains. The frame rate peaked at 18 FPS and the average was about 15 FPS. This terrain consists of 3200 triangles and 9600 vertices.

Table 1: Measurements Results

Implementation	**Average FPS (SW)**	**Average FPS (FPGA)**	**SpeedUp (FPGA/SW)**
Vertex Processing and Rasterization	140	39	0.28
Flat Shading	130	34	0.26
Gouraud Shading	127	33	0.26
Texture Mapping	41	28	0.69
Perlin Noise Mapping to Ramp Texture	10	8	0.8
Particle System	180	220	1.22
Particle System with Repeller	170	193	1.14
Displacement Mapping using Perlin Noise	22	15	0.68

As we can see from the table 1, our system has speed-up only in particle system effect. This means that it is more efficient for small faces rendering and when we have great number of triangles, such as particles, than a single threaded CPU implementation. In first three phases we have very low performance. In Texture Mapping phase, Displacement Mapping and Perlin Noise Mapping the performance on the FPGA is a little lower than in software. In hardware, when we use the perlin noise function, we can see that performance is not much lower than in software. It can be concluded that Perlin Noise function decreases the frame rate in software to a great extent, while the FPGA-based Perlin Noise implementation seems to be more efficient.

In order to see the performance in a larger resolution, we used the same implementation as we did in Vertex and Rasterization Phase, replacing frame buffer and depth buffer with larger block ram. We ran the demo both in software and FPGA. The selected resolution was $640 * 480$ due to the limited block ram resources of the board. In Figure 100 we can see the demo running both in software and on the FPGA.

The performance was very low in this resolution. In software, the average frame rate was about 22 FPS and on the FPGA about 7 FPS. These results

show us that our system is more efficient for screens with low resolution, such as small handheld devices.

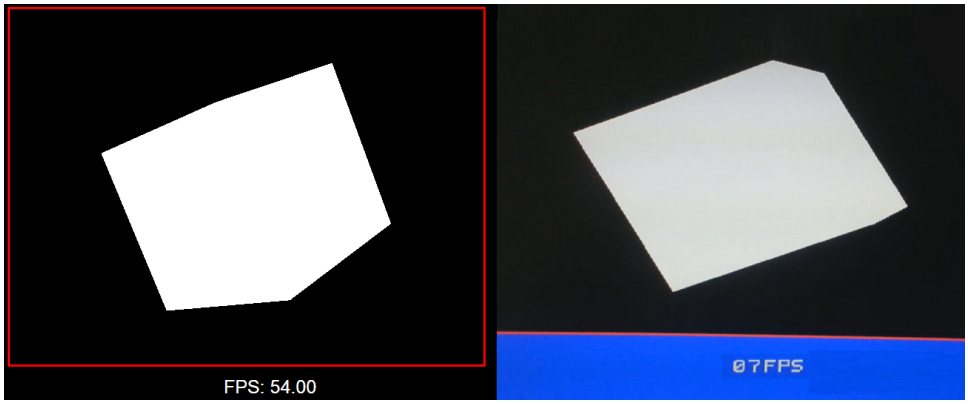


Figure 100: Vertex Processing and Rasterization phase running in software (left) and on the FPGA (right) with $640 * 480$ pixels resolution

7 Conclusions and Recommendations

In conclusion, we can say that our main goal of developing three advanced rendering effects using an FPGA-based 3D graphics pipeline has been achieved. This graphics processor is able to render any 3D object, as we can see from the last two rendering effects and independently from the complexity of the mesh (the terrain in the third effect consists of 9600 vertices).

In comparison to modern commercial graphics platforms, the performance of our system in frames per second was very low. However, additional features could be added to the FPGA implementation presented in this thesis in order to optimize the current design and make it function even more efficiently.

One feature that could be added to definitely improve the performance would be the application of pipeline to the implementation. In our system, new vertices enter the graphics pipeline when the rasterization of the previous ones has been completed successfully. In order to avoid this great delay, we could have some pipeline stages with the appropriate control unit and pipeline registers in order to pass the next vertices earlier.

Moreover, as we have seen from the resources utilization figures, we have used almost all the available DSP slices of Kintex-7 which are essential for the floating point operations. If we had a stronger FPGA with more DPS slices, we would be able to render two different triangles at the same time. For instance, we could use two Project, DrawTriangle, ProcessScanLine and DrawPoint components (Figure 103). In this case we would need also a DrawControl component to control the writing process of the pixels into the buffers, because processor unit can write only one pixel per cycle. This implementation would offer, as mentioned, the rasterization of two triangles simultaneously so we could have about double performance.

We could also raster more triangles in parallel using more times the previously referred components, considering also the number of triangles that describe the mesh.

Furthermore, a finding that came during the implementation of this project was that HLS can help extremely in optimizing the performance of a module. More particularly, the matrix multiplication component implemented with HLS requires 36 cycles to give the right output, while the implementation in VHDL needs 610 cycles (Figure 101). This is due to the optimizations

that the HLS makes with the directives we have chosen.

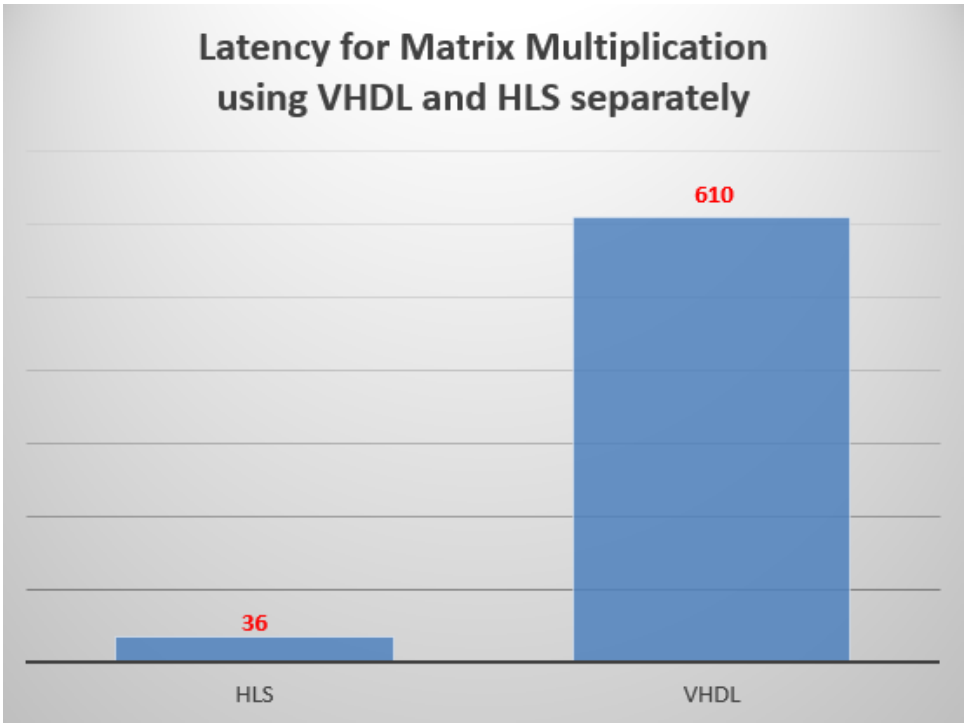


Figure 101: Latency for Matrix Multiplication using VHDL and HLS separately

However, the HLS implementation uses 80 DSP slices while the respective one in VHDL uses 14 DSPs (Figure 102). This means that in order to achieve higher performance by using HLS, we need quite more resources.

Because the difference in efficiency between the two ways of implementation is particularly high, we could reduce the circuit performance we take from HLS in order to need less resources and have a balanced situation.

With the experience of suing both implementation methods, we can can conclude that the decision of choosing between HLS and VHDL depends on the complexity of the problem we need to implement. It is worth noting that the implementation of matrix multiplication in VHDL took us two days with its verification, while HLS just two minutes.
To sum up, a great variety of potential portable devices that display 3D

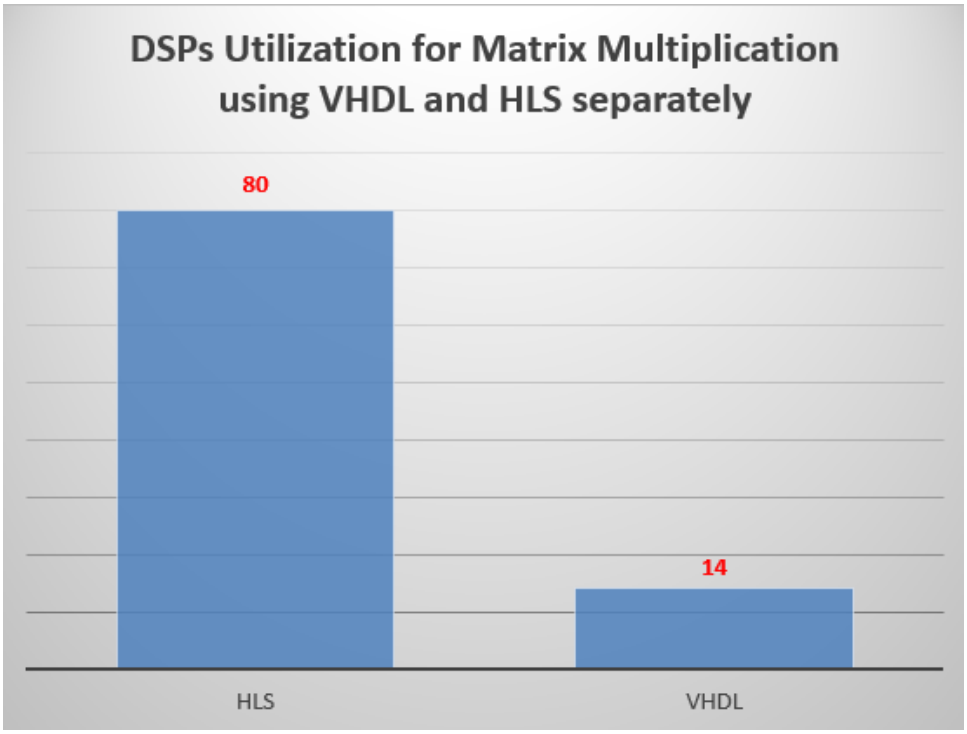


Figure 102: DSPs Utilization for Matrix Multiplication using VHDL and HLS separately

animations and graphics could use such a graphic processor. Devices such as Google Glasses, GPU navigators, smartphones and tablets need great performance and low energy consumption. As FPGAs need little energy to operate and based on the results of this project, we believe that an improved implementation of this 3D graphics pipeline with better performance is attainable in the future for handheld devices.

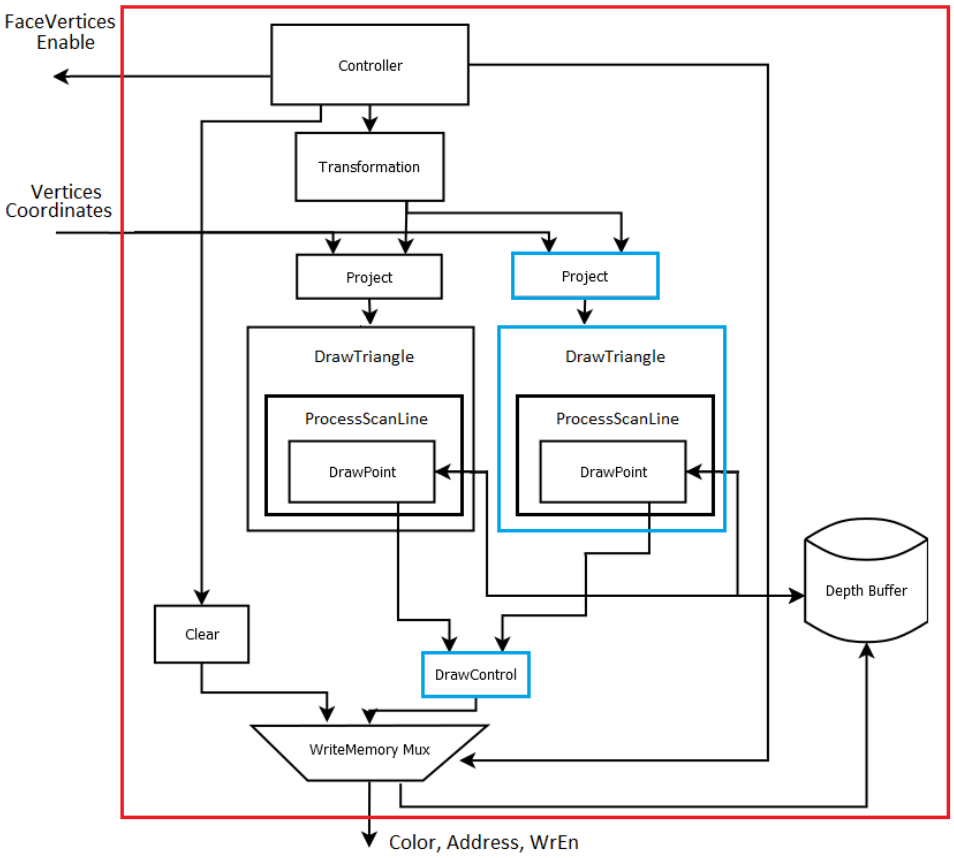


Figure 103: Creating two Triangles in parallel

8 Appendix

8.1 MatrixMultiplication - HLS code

In this component we multiply two matrices, M[4][4] and N[4][4] and we get the K[4][4] matrix. We used pipeline, dataflow and unroll directives to optimize the performance of the implementation.

Listing 4: MatrixMultiplication HLS code

```
#include <math.h>
#include "MatrixMul.h"

void MatrixMul(
```

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	80	8183	14209
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	1	-
Total	0	80	8184	14209
Available	1430	1440	445200	222600
Utilization (%)	0	5	1	6

Latency		Interval		
min	max	min	max	Type
36	36	37	37	dataflow

Figure 104: Resources Utilization and Latency for MatrixMultiplication component

```
   // Matrix M[4][4] - Input
   float m_11_in, float m_12_in, float m_13_in, float m_14_in,
   float m_21_in, float m_22_in, float m_23_in, float m_24_in,
   float m_31_in, float m_32_in, float m_33_in, float m_34_in,
   float m_41_in, float m_42_in, float m_43_in, float m_44_in,
   // Matrix N[4][4] - Input
   float n_11_in, float n_12_in, float n_13_in, float n_14_in,
   float n_21_in, float n_22_in, float n_23_in, float n_24_in,
   float n_31_in, float n_32_in, float n_33_in, float n_34_in,
   float n_41_in, float n_42_in, float n_43_in, float n_44_in,
   // Matrix K[4][4] - Output
   float *k_11_out, float *k_12_out, float *k_13_out, float
       *k_14_out,
   float *k_21_out, float *k_22_out, float *k_23_out, float
       *k_24_out,
   float *k_31_out, float *k_32_out, float *k_33_out, float
       *k_34_out,
   float *k_41_out, float *k_42_out, float *k_43_out, float
       *k_44_out
)
{
#pragma HLS DATAFLOW
#pragma HLS PIPELINE
   // We write M and N input values to
   // arrays in C
   float m[4][4], n[4][4];
   m[0][0] = m_11_in;
   m[0][1] = m_12_in;
   m[0][2] = m_13_in;
```

```
  m[0][3] = m_14_in;
  m[1][0] = m_21_in;
  m[1][1] = m_22_in;
  m[1][2] = m_23_in;
  m[1][3] = m_24_in;
  m[2][0] = m_31_in;
  m[2][1] = m_32_in;
  m[2][2] = m_33_in;
  m[2][3] = m_34_in;
  m[3][0] = m_41_in;
  m[3][1] = m_42_in;
  m[3][2] = m_43_in;
  m[3][3] = m_44_in;
  n[0][0] = n_11_in;
  n[0][1] = n_12_in;
  n[0][2] = n_13_in;
  n[0][3] = n_14_in;
  n[1][0] = n_21_in;
  n[1][1] = n_22_in;
  n[1][2] = n_23_in;
  n[1][3] = n_24_in;
  n[2][0] = n_31_in;
  n[2][1] = n_32_in;
  n[2][2] = n_33_in;
  n[2][3] = n_34_in;
  n[3][0] = n_41_in;
  n[3][1] = n_42_in;
  n[3][2] = n_43_in;
  n[3][3] = n_44_in;
  // t_k array contains the values for the
  // final K matrix
  float t_k[16];
  float sum = 0;
  int cnt = 0;
  int row, column, p;

  for (row = 0; row < 4; row++) {
#pragma HLS PIPELINE
#pragma HLS UNROLL factor=4
    for (column = 0; column < 4; column++) {
#pragma HLS PIPELINE
#pragma HLS UNROLL factor=4
     // in this for loop we compute values of K matrix
     for (p = 0; p < 4; p++) {
#pragma HLS PIPELINE
```

```
#pragma HLS UNROLL factor=4
       sum = sum + m[row][p]*n[p][column];
     }
     t_k[cnt] = sum;
     cnt++;
     sum = 0;
    }
  }
  // we assign the computed K matrix to output
  *k_11_out = t_k[0];
  *k_12_out = t_k[1];
  *k_13_out = t_k[2];
  *k_14_out = t_k[3];
  *k_21_out = t_k[4];
  *k_22_out = t_k[5];
  *k_23_out = t_k[6];
  *k_24_out = t_k[7];
  *k_31_out = t_k[8];
  *k_32_out = t_k[9];
  *k_33_out = t_k[10];
  *k_34_out = t_k[11];
  *k_41_out = t_k[12];
  *k_42_out = t_k[13];
  *k_43_out = t_k[14];
  *k_44_out = t_k[15];
}
```

8.2 MatrixVectorMultiplication - HLS code

In this component we multiply matrix M[4][4] with vector N[4] and we get the K[4] final vector. We used pipeline, dataflow and unroll directives to optimize the performance of the implementation.

Listing 5: MatrixVectorMultiplication HLS code

```
#include <math.h>
#include "MatrixArrayMul.h"

void MatrixArrayMul(
  // Matrix M[4][4] - Input
  float m_11_in, float m_12_in, float m_13_in, float m_14_in,
  float m_21_in, float m_22_in, float m_23_in, float m_24_in,
  float m_31_in, float m_32_in, float m_33_in, float m_34_in,
  float m_41_in, float m_42_in, float m_43_in, float m_44_in,
```

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	20	2051	3553
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	1	-
Total	0	20	2052	3553
Available	1430	1440	445200	222600
Utilization (%)	0	1	~0	1

Latency		Interval		
min	max	min	max	Type
36	36	37	37	dataflow

Figure 105: Resources Utilization and Latency for MatrixVectorMultiplication component

```
   // Vector N[4] - Input
   float n_1_in, float n_2_in, float n_3_in, float n_4_in,
   // Vector K4] - Output
   float *k_1_out, float *k_2_out, float *k_3_out, float *k_4_out
){
#pragma HLS DATAFLOW
#pragma HLS PIPELINE
   // We write M and N input values to
   // arrays in C
   float m[4][4], n[4];
   m[0][0] = m_11_in;
   m[0][1] = m_12_in;
   m[0][2] = m_13_in;
   m[0][3] = m_14_in;
   m[1][0] = m_21_in;
   m[1][1] = m_22_in;
   m[1][2] = m_23_in;
   m[1][3] = m_24_in;
   m[2][0] = m_31_in;
   m[2][1] = m_32_in;
   m[2][2] = m_33_in;
   m[2][3] = m_34_in;
   m[3][0] = m_41_in;
   m[3][1] = m_42_in;
   m[3][2] = m_43_in;
   m[3][3] = m_44_in;
   n[0] = n_1_in;
   n[1] = n_2_in;
   n[2] = n_3_in;
```

```
   n[3] = n_4_in;

   // t_k array contains the values for the
   // final K vector
   float t_k[4];
   float sum = 0;
   int cnt = 0;
   int row, p;

   for (row = 0; row < 4; row++) {
#pragma HLS PIPELINE
#pragma HLS UNROLL factor=4
      // in this for loop we compute values of K vector
      for (p = 0; p < 4; p++) {
#pragma HLS PIPELINE
#pragma HLS UNROLL factor=4
        sum = sum + m[row][p]*n[p];
      }
       t_k[cnt] = sum;
       cnt++;
      sum = 0;
   }
   // we assign the computed K vector to output
   *k_1_out = t_k[0];
   *k_2_out = t_k[1];
   *k_3_out = t_k[2];
   *k_4_out = t_k[3];
}
```

8.3 TransformCoordinates - HLS code

In this component we divide x , y and z with w value. We used pipeline and dataflow directives to optimize the performance of the implementation.

Listing 6: TransformCoordinates HLS code

```
#include "TransformCoordinates.h"

void TransformCoordinates(
   // X, Y and Z - Input
   float x_in, float y_in, float z_in,
   // W value - Input
   float w_in,
   // Updated X, Y and Z - Output
```

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	0	2384	3082
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	1	-
Total	0	0	2385	3082
Available	890	840	407600	203800
Utilization (%)	0	0	~0	1

Latency		Interval		
min	max	min	max	Type
15	15	16	16	dataflow

Figure 106: Resources Utilization and Latency for TransformCoordinates component

```
   float *x_out, float *y_out, float *z_out)
{
#pragma HLS DATAFLOW
#pragma HLS PIPELINE II=2
   *x_out = x_in / w_in;
   *y_out = y_in / w_in;
   *z_out = z_in / w_in;
}
```

8.4 I2C sender - VHDL code

This component sends register writes over an I2C-like interface to the ADV7511 HDMI transmitter. It was implemented by Mike Field.

Listing 7: I2C sender VHDL code

```
----------------------------------------------------------------------------------
-- Engineer: <mfield@concepts.co.nz
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity i2c_sender is
    Port ( clk   : in   STD_LOGIC;
           resend : in   STD_LOGIC;
           sioc   : out  STD_LOGIC;
           siod   : inout STD_LOGIC
    );
```

```
end i2c_sender;

architecture Behavioral of i2c_sender is
   signal  divider         : unsigned(7 downto 0) := (others =>
       '0');
    -- this value gives nearly 200ms cycles before the first
        register is written
   signal  initial_pause   : unsigned(22 downto 0) := (others =>
       '0');
   signal  finished        : std_logic := '0';
   signal  address         : std_logic_vector(7 downto 0) :=
       (others => '0');
   signal  clk_first_quarter : std_logic_vector(28 downto 0) :=
       (others => '1');
   signal  clk_last_quarter : std_logic_vector(28 downto 0) :=
       (others => '1');
   signal  busy_sr         : std_logic_vector(28 downto 0) :=
       (others => '1');
   signal  data_sr         : std_logic_vector(28 downto 0) :=
       (others => '1');
   signal  tristate_sr     : std_logic_vector(28 downto 0) :=
       (others => '0');
   signal  reg_value       : std_logic_vector(15 downto 0) :=
       (others => '0');
   constant i2c_wr_addr    : std_logic_vector(7 downto 0) := x"72";

   type reg_value_pair is ARRAY(0 TO 63) OF std_logic_vector(15
       DOWNTO 0);
   signal switch_config_sent : std_logic := '0';
   signal reg_value_pairs : reg_value_pair := (
           -------------------
           -- Powerup please!
           -------------------
           x"4110",
           ---------------------------------------
           -- These valuse must be set as follows
           ---------------------------------------
           x"9803", x"9AE0", x"9C30", x"9D61", x"A2A4", x"A3A4",
               x"E0D0", x"5512", x"F900",

           ---------------
           -- Input mode
           ---------------
           x"1506", -- YCbCr 422, DDR, External sync
           x"4810", -- Left justified data (D23 downto 8)
```

```
    x"1637", -- 8 bit style 2, 1st half on rising edge -
        YCrCb clipping
    x"1700", -- output aspect ratio 16:9, external DE
    x"D03C", -- auto sync data - must be set for DDR modes.
    ---------------
    -- Output mode
    ---------------
    x"AF04", -- DVI mode
    x"4c04", -- Deep colour off (HDMI only?) - not needed
    x"4000", -- Turn off additional data packets - not needed

------------------------------------------------------------------
-- As a start, here's the identity Matrix a <= A, b <= B, c
    <= C
------------------------------------------------------------------
--   -- ([R or Cr]* A1 + [G or Y] * A2 +  [B or Cb] *
    A3)/4096 + A4   =  Red or Cr
 --   x"18A8", x"1900", x"1A00", x"1B00", x"1C00", x"1D00",
     x"1E00", x"1F00",
--
--   -- ([R or Cr]* B1 + [G or Y] * B2 +  [B or Cb] *
    B3)/4096 + B4   =  Green or Y
--   x"2000", x"2100", x"2208", x"2300", x"2400", x"2500",
    x"2604", x"2700",
--
--   -- ([R or Cr]* C1 + [G or Y] * C2 +  [B or Cb] *
    C3)/4096 + C4   =  Blue or Cb
--   x"2808", x"2900", x"2A00", x"2B00", x"2C08", x"2D00",
    x"2E04", x"2F00",

------------------------------------------------------------
-- Here is the YCrCb => RGB conversion, as per programming
    guide
-- This is table 57 - HDTV YCbCr (16 to 255) to RGB (0 to 255)
------------------------------------------------------------
     -- (Cr * A1     +     Y * A2      +    Cb * A3)/4096 +
         A4   =  Red
      x"18E7", x"1934", x"1A04", x"1BAD", x"1C00", x"1D00",
          x"1E1C", x"1F1B",

     -- (Cr * B1     +     Y * B2      +    Cb * B3)/4096 +
         B4   =  Green
     x"201D", x"21DC", x"2204", x"23AD", x"241F", x"2524",
         x"2601", x"2735",
```

```
            -- (Cr * C1     +     Y * C2     +    Cb * C3)/4096 +
                C4   =  Blue
            x"2800", x"2900", x"2A04", x"2BAD", x"2C08", x"2D7C",
                x"2E1B", x"2F77",


            -- Extra space filled with FFFFs to signify end of data
            x"FFFF", x"FFFF", x"FFFF", x"FFFF", x"FFFF", x"FFFF",
                x"FFFF",
            x"FFFF", x"FFFF", x"FFFF", x"FFFF", x"FFFF", x"FFFF",
                x"FFFF", x"FFFF",
            x"FFFF", x"FFFF", x"FFFF", x"FFFF", x"FFFF", x"FFFF",
                x"FFFF"
   );
begin

registers: process(clk)
   begin
      if rising_edge(clk) then
         reg_value <= reg_value_pairs(to_integer(unsigned(address)));
      end if;
   end process;

i2c_tristate: process(data_sr, tristate_sr)
   begin
      if tristate_sr(tristate_sr'length-1) = '0' then
         siod <= data_sr(data_sr'length-1);
      else
         siod <= 'Z';
      end if;
   end process;

   with divider(divider'length-1 downto divider'length-2)
      select sioc <= clk_first_quarter(clk_first_quarter'length -1)
          when "00",
                   clk_last_quarter(clk_last_quarter'length -1)
                       when "11",
                   '1' when others;

i2c_send:  process(clk)
   begin
      if rising_edge(clk) then
         if resend = '1' then
            address         <= (others => '0');
            clk_first_quarter <= (others => '1');
```

```
   clk_last_quarter <= (others => '1');
   busy_sr          <= (others => '0');
   divider          <= (others => '0');
   initial_pause    <= (others => '0');
   finished <= '0';
end if;

if busy_sr(busy_sr'length-1) = '0' then
   if initial_pause(initial_pause'length-1) = '0' then
      initial_pause <= initial_pause+1;
   elsif switch_config_sent = '0' then
      ------------------------------------------------------------
      -- the HDMI chip is behind a bus switch
      -- That needs to be configured - so once we have waited
      -- for a while we jam the command into the shift
          registers
      -- ready for transmission once the pause is over.
      ------------------------------------------------------------
      clk_first_quarter <= (others => '0');
      clk_first_quarter(clk_first_quarter'length-1) <= '1';
      clk_first_quarter(8 downto 0) <= (others => '1');

      clk_last_quarter <= (others => '0');
      clk_last_quarter(9 downto 0) <= (others => '1');

      --            Start   Address  Ack   Register Ack
          Stop Padding
      tristate_sr <= "0" & "00000000" & "1" & "00000000" &
          "1" & "0" & "000000000";
      data_sr    <= "0" & "11101000" & "1" & "00100000" &
          "1" & "0" & "111111111";
      -- We don't care if we wait too long whnconfiguring
          the switch....
      busy_sr           <= (others => '1');
      switch_config_sent <= '1';
   elsif finished = '0' then
      if divider = "11111111" then
         divider <= (others =>'0');
         if reg_value(15 downto 8) = "11111111" then
            finished <= '1';
         else
            -- move the new data into the shift registers
            clk_first_quarter <= (others => '0');
                clk_first_quarter(clk_first_quarter'length-1)
                <= '1';
```

```
                clk_last_quarter <= (others => '0');
                    clk_last_quarter(0) <= '1';

                --            Start   Address       Ack
                    Register          Ack         Value        Ack
                      Stop
                tristate_sr <= "0" & "00000000" & "1" &
                    "00000000"        & "1" & "00000000"
                    & "1" & "0";
                data_sr    <= "0" & i2c_wr_addr & "1" &
                    reg_value(15 downto 8) & "1" & reg_value( 7
                    downto 0) & "1" & "0";
                busy_sr    <= (others => '1');
                address    <=
                    std_logic_vector(unsigned(address)+1);
              end if;
            else
              divider <= divider+1;
            end if;
          end if;
        else
          if divider = "11111111" then -- divide clkin by 255 for
              I2C
            tristate_sr     <= tristate_sr(tristate_sr'length-2
                downto 0) & '0';
            busy_sr         <= busy_sr(busy_sr'length-2 downto 0)
                & '0';
            data_sr         <= data_sr(data_sr'length-2 downto 0)
                & '1';
            clk_first_quarter <=
                clk_first_quarter(clk_first_quarter'length-2
                downto 0) & '1';
            clk_last_quarter <=
                clk_last_quarter(clk_first_quarter'length-2 downto
                0) & '1';
            divider         <= (others => '0');
          else
            divider <= divider+1;
          end if;
        end if;
      end if;
    end process;
end Behavioral;
```

8.5 FaceNormalVector - HLS code

In this component, we compute the face normal from three vertex's normals. Normal face's vector is the average normal between each vertex's normal. In orde to compute our normal vector, we just need to take the 3 vertices normals, add them to each other and divide them by 3. We used pipeline and dataflow directives to optimize the performance of the implementation.

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	6	3369	4579
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	1	-
Total	0	6	3370	4579
Available	890	840	407600	203800
Utilization (%)	0	~0	~0	2

Latency		Interval		
min	max	min	max	Type
31	31	32	32	dataflow

Figure 107: Resources Utilization and Latency for VnFace component

Listing 8: Face Normal Vector HLS code

```
#include <math.h>
#include "VnFace.h"

void VnFace(
     // Vertex Normals V1, V2, V3 - Inputs
     float v1_x_in, float v1_y_in, float v1_z_in,
     float v2_x_in, float v2_y_in, float v2_z_in,
     float v3_x_in, float v3_y_in, float v3_z_in,
     // Face Normal - Output
     float *vnFace_x, float *vnFace_y, float *vnFace_z ){
#pragma HLS DATAFLOW
#pragma HLS PIPELINE II=2

     // v2.add(v3)
     float temp1_x = v2_x_in + v3_x_in;
     float temp1_y = v2_y_in + v3_y_in;
     float temp1_z = v2_z_in + v3_z_in;

     // v1.add(v2.add(v3))
     float temp2_x = v1_x_in + temp1_x;
```

```
    float temp2_y = v1_y_in + temp1_y;
    float temp2_z = v1_z_in + temp1_z;

    // (v1.add(v2.add(v3))).scale(1 / 3);
    *vnFace_x = temp2_x / 3;
    *vnFace_y = temp2_y / 3;
    *vnFace_z = temp2_z / 3;
}
```

8.6 CenterPoint - HLS code

In this component, we compute the center point of the face from the three vertices. We used pipeline and dataflow directives to optimize the performance of the implementation.

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	6	3369	4579
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	1	-
Total	0	6	3370	4579
Available	1910	1920	597200	597200
Utilization (%)	0	~0	~0	~0

Latency		Interval		
min	max	min	max	Type
31	31	32	32	dataflow

Figure 108: Resources Utilization and Latency for CenterPoint component

Listing 9: CenterPoint HLS code

```
#include <math.h>
#include "CenterPoint.h"

void CenterPoint(
    // Vertices V1, V2, V3 - Inputs
    float v1_x_in, float v1_y_in, float v1_z_in,
    float v2_x_in, float v2_y_in, float v2_z_in,
    float v3_x_in, float v3_y_in, float v3_z_in,
    // CenterPoint - Output
    float *CenterPoint_x, float *CenterPoint_y, float
        *CenterPoint_z ){
#pragma HLS DATAFLOW
#pragma HLS PIPELINE II=2
```

```
    // v2.add(v3)
    float temp1_x = v2_x_in + v3_x_in;
    float temp1_y = v2_y_in + v3_y_in;
    float temp1_z = v2_z_in + v3_z_in;

    // v1.add(v2.add(v3))
    float temp2_x = v1_x_in + temp1_x;
    float temp2_y = v1_y_in + temp1_y;
    float temp2_z = v1_z_in + temp1_z;

    // (v1.add(v2.add(v3))).scale(1 / 3);
    *CenterPoint_x = temp2_x / 3;
    *CenterPoint_y = temp2_y / 3;
    *CenterPoint_z = temp2_z / 3;
}
```

8.7 ComputeNdotL - HLS code

This component computes the cosine of the angle between the light vector and the normal vector using dot product. It returns a value between 0 and 1, which is the intensity of the color for shading. We used pipeline and dataflow directives to optimize the performance of the implementation.

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	21	2440	3492
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	1	-
Total	0	21	2441	3492
Available	1430	1440	445200	222600
Utilization (%)	0	1	~0	1

Latency		Interval		
min	max	min	max	Type
90	90	91	91	dataflow

Figure 109: Resources Utilization and Latency for ComputeNdotL component

Listing 10: ComputeNdotL HLS code

```
#include <math.h>
#include "ComputeNDotL.h"
```

```
void ComputeNDotL(
      // Vertex coordinates - Input
      float vertex_x_in, float vertex_y_in, float vertex_z_in,
      // Normal Vector - Input
      float normal_x_in, float normal_y_in, float normal_z_in,
      // Light Positin - Input
      float lightPos_x_in, float lightPos_y_in, float lightPos_z_in,
      // Final Color Intensity - Output
      float *colorIntensity_out){
#pragma HLS DATAFLOW
#pragma HLS PIPELINE II=2

       // lightDirection = lightPosition.subtract(vertex)
      float lightDir_x = lightPos_x_in - vertex_x_in;
      float lightDir_y = lightPos_y_in - vertex_y_in;
      float lightDir_z = lightPos_z_in - vertex_z_in;

      // normal.normalize();
      float normalLength;
      normalLength = sqrt((normal_x_in*normal_x_in) +
          (normal_y_in*normal_y_in) + (normal_z_in*normal_z_in));

      float num;
      float normal_x;
      float normal_y;
      float normal_z;

      if (normalLength == 0) {
         normal_x = normal_x_in;
         normal_y = normal_y_in;
         normal_z = normal_z_in;
      }
      else {
         num = 1.0 / normalLength;
         normal_x = normal_x_in * num;
         normal_y = normal_y_in * num;
         normal_z = normal_z_in * num;
      }

      // lightDirection.normalize();
      normalLength = sqrt((lightDir_x * lightDir_x) + (lightDir_y *
          lightDir_y) + (lightDir_z * lightDir_z));
      num = 1.0 / normalLength;
```

```
      float lightDir2_x = lightDir_x * num;
      float lightDir2_y = lightDir_y * num;
      float lightDir2_z = lightDir_z * num;
      // return Math.max(0, BABYLON.Vector3.Dot(normal,
          lightDirection));
        float Dot = (normal_x * lightDir2_x) + (normal_y *
            lightDir2_y) + (normal_z * lightDir2_z);

        if (Dot > 0) {
         *colorIntensity_out = Dot;
        }
        else {
         *colorIntensity_out = 0;
        }
}
```

8.8 PerlinNoise - HLS code

This component maps perlin noise value to a ramp texture. At first we compute the noise value depending on the coordinates (x, y, z) given as inputs. The value we get is a decimal between 0 and 1, so we multiply it with 255 in order o map it to the 0x255 grayscale texture. Finally, this component returns a color value. We used pipeline, dataflow and unroll directives to optimize the performance of the implementation.

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	3989
FIFO	-	-	-	-
Instance	-	119	16160	25982
Memory	6	-	0	0
Multiplexer	-	-	-	886
Register	-	-	2873	-
Total	6	119	19033	30857
Available	890	840	407600	203800
Utilization (%)	~0	14	4	15

Latency		Interval		
min	max	min	max	Type
4	432	5	433	none

Figure 110: Resources Utilization and Latency for PerlinNoise component

Listing 11: PerlinNoise HLS code

```
#include <math.h>
#include "PerlinNoise.h"
```

```
void PerlinNoise(
     // Pixel Coordinates - Inputs
     float x_in, float y_in, float z_in,
     // Color value - Output
     float *rampTextureAddr){

  // DEFINE PERLIN'S CONSTANT ARRAY
  int permutation[256] = {151,160,137,91,90,15,
       131,13,201,95,96,53,194,233,7,225,140,36,103,30,69,142,8,99,37,240,21,10,23,
       190,
           6,148,247,120,234,75,0,26,197,62,94,252,219,203,117,35,11,32,57,177,33,
       88,237,149,56,87,174,20,125,136,171,168,
           68,175,74,165,71,134,139,48,27,166,
       77,146,158,231,83,111,229,122,60,211,133,230,220,105,92,41,55,46,245,40,244,
       102,143,54, 65,25,63,161, 1,216,80,73,209,76,132,187,208,
           89,18,169,200,196,
       135,130,116,188,159,86,164,100,109,198,173,186,
           3,64,52,217,226,250,124,123,
       5,202,38,147,118,126,255,82,85,212,207,206,59,227,47,16,58,17,182,189,28,42,
       223,183,170,213,119,248,152, 2,44,154,163,
           70,221,153,101,155,167, 43,172,9,
       129,22,39,253, 19,98,108,110,79,113,224,232,178,185,
           112,104,218,246,97,228,
       251,34,242,193,238,210,144,12,191,179,162,241,
           81,51,145,235,249,14,239,107,
       49,192,214, 31,181,199,106,157,184,
           84,204,176,115,121,50,45,127, 4,150,254,
       138,236,205,93,222,114,67,29,24,72,243,141,128,195,78,66,215,61,156,180};

  float x23, y23, z23;
  float tempX, tempY, tempZ;
  int X, Y, Z;
  float x2, y2, z2;
  float u, v, w;
  int p[512], i;
  int A, AA, AB, B, BA, BB;
  float PerlinNoise_out;

  // viewportWidth is 400 and viewportHeight is 200
  // we check if the current pixel is inside this area
  // to avoid computing perlin noise value for pixels that won't
      be drawn
  // in the screen. Trick for running faster!
  if ((x_in >= 0) && (x_in <= 399) && (y_in >= 0) && (y_in <=199)){
```

```
x23 = (x_in)+0.285;
y23 = (y_in)+0.285;
// We multiply z_in with 100000 because it is a very little
    value
// and in order to avoid the component to get it as zero.
// That's the reason we also added a random value (0.285) to
    all input
//coordinates
z23 = (z_in*100000)+0.285;

// FIND UNIT CUBE THAT CONTAINS POINT.
tempX = floor(x23);
X = (int)tempX & 255;
tempY = floor(y23);
Y = (int)tempY & 255;
tempZ = floor(z23);
Z = (int)tempZ & 255;

// FIND RELATIVE X,Y,Z OF POINT IN CUBE.
x2 = x23 - floor(x23);
y2 = y23 - floor(y23);
z2 = z23 - floor(z23);

// COMPUTE FADE CURVES FOR EACH OF X,Y,Z.
u = fade(x2);
v = fade(y2);
w = fade(z2);

for (i=0; i < 256 ; i++){
   #pragma HLS UNROLL factor=4
   #pragma HLS PIPELINE
   p[256+i] = p[i] = permutation[i];
}

// HASH COORDINATES OF THE 8 CUBE CORNERS,
A = p[X]+Y;
AA = p[A]+Z;
AB = p[A+1]+Z;
B = p[X+1]+Y;
BA = p[B]+Z;
BB = p[B+1]+Z;

PerlinNoise_out = lerp(w, lerp(v, lerp(u, grad(p[AA ], x2 , y2
    , z2 ), // AND ADD
                              grad(p[BA ], x2-1, y2 , z2  )),
```

```
                                        // BLENDED
                                    lerp(u, grad(p[AB ], x2 , y2-1, z2 ),
                                        // RESULTS
                                          grad(p[BB ], x2-1, y2-1, z2
                                              )),// FROM 8
                                lerp(v, lerp(u, grad(p[AA+1], x2 , y2 ,
                                    z2-1 ), // CORNERS
                                          grad(p[BA+1], x2-1, y2 , z2-1
                                              )), // OF CUBE
                                    lerp(u, grad(p[AB+1], x2 , y2-1, z2-1
                                        ),
                                          grad(p[BB+1], x2-1, y2-1, z2-1
                                              ))));

      // Perlin noise Mapping to Ramp Texture
      *rampTextureAddr = abs(PerlinNoise_out*255);
   }
   else{
      // else we give black color.
      *rampTextureAddr = 0.00;
   }

}

////////////  FUNCTIONS  ////////////////////////
float fade(float t){
#pragma HLS PIPELINE
   return t * t * t * (t * (t * 6 - 15) + 10);
}

float lerp(float t, float a, float b){
#pragma HLS PIPELINE
   return a + t * (b - a);
}

float grad(int hash, float x, float y, float z){
#pragma HLS PIPELINE
   // CONVERT LO 4 BITS OF HASH CODE
   int h = hash & 15;
   // INTO 12 GRADIENT DIRECTIONS.
   float u = h<8 ? x : y;
   float v = h<4 ? y : h==12||h==14 ? x : z;

   return ((h&1) == 0 ? u : -u) + ((h&2) == 0 ? v : -v);
}
```

8.9 ParticleSystem - HLS code

In this component we update the particle parameters. We used pipeline and dataflow directives to optimize the performance of the implementation.

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	10
FIFO	-	-	-	-
Instance	-	121	28270	41352
Memory	-	-	-	-
Multiplexer	-	-	-	832
Register	-	-	4684	2222
Total	0	121	32954	44416
Available	890	840	407600	203800
Utilization (%)	0	14	8	21

Latency		Interval		
min	max	min	max	Type
136	136	1	1	function

Figure 111: Resources Utilization and Latency for ParticleSystem component

Listing 12: ParticleSystem HLS code

```
#include <math.h>
#include "ParticleSystem.h"

void ParticleSystem(
      // Inputs
      float addr_in, // number of current face
      float v1_x_in, float v1_y_in, float v1_z_in, // coordinates of
          vertex 1
      float v2_x_in, float v2_y_in, float v2_z_in, // coordinates of
          vertex 2
      float v3_x_in, float v3_y_in, float v3_z_in, // coordinates of
          vertex 3
      float x_step_in, float y_step_in, float z_step_in, // rate of
          change in x, y and z axes
      float check_in, // we use this flag type variable to check if
          it the first time
                  // this particle is parsed
      float lifespan_in, // remaining lifetime
      // Outputs
      // these variables are the updated inputs
```

```
      float *addr_out,
      float *v1_x_out, float *v1_y_out, float *v1_z_out,
      float *v2_x_out, float *v2_y_out, float *v2_z_out,
      float *v3_x_out, float *v3_y_out, float *v3_z_out,
      float *x_step_out, float *y_step_out, float *z_step_out,
      float *check_out, float *lifespan_out){
#pragma HLS PIPELINE

   // Output temp variables
   float tv1_x_out, tv1_y_out, tv1_z_out;
   float tv2_x_out, tv2_y_out, tv2_z_out;
   float tv3_x_out, tv3_y_out, tv3_z_out;
   float tx_step_out, ty_step_out, tz_step_out;
   float tcheck_out, tlifespan_out;

   // we get a random value for maximum
   // lifespan of current particle
   float lifespan_const = prandLifespan(addr_in);

   // a is the value for the acceleration
   // of the particles
   float a = 2.5;

   // t is the current time variable for the
   // equation of motion
   float t = 0;

   // when t variable reaches its maximum time
   // it dies
   float max_time = lifespan_const/1000;

   // check is zero only the first time we update
   // current's particle parameters, then is always 1
   if (check_in == 0){

      // Generate Random Numbers X, Y
      // prX, prY and prZ are the coordinates where the
      // particle starts moving from. It is also the rate
      // of change in each axis.
      float prX = prandX(addr_in);
      float prY = prandY(addr_in);
      float prZ = prandZ(addr_in);

      // we initialize time -> 0
      t = max_time - (lifespan_const/1000); // Time Variable
```

```
   // setting the coordinates of the vertices
   tv1_x_out = prX; // VertexA
   tv1_y_out = (prY*t) - (a*t*t); // y depends on kinematic
       equation
   tv1_z_out = 0.50;

   tv2_x_out = tv1_x_out - 0.01; // VertexB
   tv2_y_out = tv1_y_out - 0.02; // we place it lower then vertexA
   tv2_z_out = 0.50;

   tv3_x_out = tv1_x_out + 0.01; // VertexC
   tv3_y_out = tv1_y_out - 0.02; // we place it lower then vertexA
   tv3_z_out = 0.50 - 0.02;

   ////////////////////////////////////////
   // these values are being written to memory
   // in order to get them when we want to update
   // the same particle again
   tx_step_out = prX;
   ty_step_out = prY;
   tz_step_out = prZ;
   tcheck_out = 1;
   tlifespan_out = lifespan_const;
   ////////////////////////////////////////
}
else if (check_in == 1){

   // lefespan_in = 0 and we initilize particle's
   // paramaeters (do the same as we did when check was 0.
   if (lifespan_in == 0){
      // Generate Random Numbers X, Y
      float prX = prandX(addr_in);
      float prY = prandY(addr_in);
      float prZ = prandZ(addr_in);

      t = max_time - (lifespan_const/1000); // Time Variable

      tv1_x_out = 0.00; // VertexA
      tv1_y_out = (prY*t) - (a*t*t);
      tv1_z_out = 0.50;

      tv2_x_out = tv1_x_out - 0.01; // VertexB
      tv2_y_out = tv1_y_out - 0.02;
      tv2_z_out = 0.50;
```

```
      tv3_x_out = tv1_x_out + 0.01; // VertexC
      tv3_y_out = tv1_y_out - 0.02;
      tv3_z_out = 0.50 - 0.02;

      ////////////////////////////////////////
      tx_step_out = prX;
      ty_step_out = prY;
      tz_step_out = prZ;
      tcheck_out = 1;
      tlifespan_out = lifespan_const;
      ////////////////////////////////////////
   }
   else{// here we update the parameters
      // t variable depends on remaining lifetime
      t = max_time - (lifespan_in/1000);

      // Updating the coordinates of the vertices
      tv1_x_out = v1_x_in + x_step_in; // VertexA
      tv1_y_out = (y_step_in*t) -a*t*t;
      tv1_z_out = 0.50 + z_step_in;

      tv2_x_out = tv1_x_out - 0.01; // VertexB
      tv2_y_out = tv1_y_out - 0.02;
      tv2_z_out = 0.50 + z_step_in;

      tv3_x_out = tv1_x_out + 0.01; // VertexC
      tv3_y_out = tv1_y_out - 0.02;
      tv3_z_out = 0.50 + z_step_in - 0.02;

      ////////////////////////////////////////
      tx_step_out = x_step_in;
      ty_step_out = y_step_in;
      tz_step_out = z_step_in;
      tcheck_out = 1;
      tlifespan_out = lifespan_in - 1;
      ////////////////////////////////////////
   }
}
*addr_out = addr_in;
*v1_x_out = tv1_x_out;
*v1_y_out = tv1_y_out;
*v1_z_out = tv1_z_out;
*v2_x_out = tv2_x_out;
*v2_y_out = tv2_y_out;
```

```
   *v2_z_out = tv2_z_out;
   *v3_x_out = tv3_x_out;
   *v3_y_out = tv3_y_out;
   *v3_z_out = tv3_z_out;
   *x_step_out = tx_step_out;
   *y_step_out = ty_step_out;
   *z_step_out = tz_step_out;
   *check_out = tcheck_out;
   *lifespan_out = tlifespan_out;
}


////////////// FUNCTIONS ////////////////////////////////
float prandX(float addr){
#pragma HLS PIPELINE

   int x, p, x1, i;
   x = 123456789;
   p = 289; // top Limit
   i = addr + 200;
   x1 = (((i*x*234525)/53112) % p);

   // Make it Float
   float temp = x1;
   float prand = temp/800000;

   return prand;
}

float prandY(float addr){
#pragma HLS PIPELINE

   int y, p, y1, i;
   y = 123456789;
   p = 10789; // top Limit
   i = addr + 200;
   y1 = (((i*y*219745)/53112) % p);

   // Make it Float
   float temp = y1;
   // Make it positive
   float prandt1 = temp/4000;
   float prandt2 = prandt1 * prandt1;
   float prand = sqrt(prandt2);
```

```
   return prand;
}

float prandZ(float addr){
#pragma HLS PIPELINE

   int x, p, x1, i;
   x = 123456789;
   p = 289; // Right/Left Limit
   i = addr + 200;
   x1 = (((i*x*234525)/53112) % p);

   // Make it Float
   float temp = x1;
   float prand = temp/600000;

   return prand;
}

float prandLifespan(float addr){
#pragma HLS PIPELINE

   int y, p, y1, i;
   y = 123456789;
   p = 156; // Up Limit
   i = addr + 200;
   y1 = (((i*y*234525)/53112) % p);

   // Make it Float
   float temp = y1;
   // Make it positive
   float prandt1 = temp + 1100;
   float prandt2 = prandt1 * prandt1;
   float prand = sqrt(prandt2);

   return prand;
}
////////////////////////////////////////////////
```

8.10 MapColor - HLS code

In this component, we map particle's lifespan to grayscale ramp texture. As particle is dying (remaining lifetime $\rightarrow$ 0), its color becomes darker. We

used pipeline directive to optimize the performance of the implementation.

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	2
FIFO	-	-	-	-
Instance	-	20	4009	5953
Memory	-	-	-	-
Multiplexer	-	-	-	43
Register	-	-	1341	100
Total	0	20	5350	6098
Available	890	840	407600	203800
Utilization (%)	0	2	1	2

Latency		Interval		
min	max	min	max	Type
101	101	1	1	function

Figure 112: Resources Utilization and Latency for MapColor component

Listing 13: MapColor HLS code

```
#include <math.h>
#include "MapColor.h"

void MapColor(
     // Inputs
     float addr_in, // particles's number/address
     float check_in, // check if particle has ever be drawn
     float lifespan_in, // particle's remaining lifetime
     // Output, final color value
     float *color_out ){
#pragma HLS PIPELINE

   // Output temp Variables
   float tcolor_out;
   // compute the maximum lifespan for cureent particles
   float lifespan_const = prandLifespanMC(addr_in);

   // if particle has never parsed again, we draw black color
   if (check_in == 0){
      tcolor_out = 255;
   }
   else{
      if (lifespan_in == 0){ // particle dies
         tcolor_out = 255; // black color
      }
      else{
         // mapping lifespan with grayscale 1x255 ramp texture
```

```
        tcolor_out = ((lifespan_in - 1) / lifespan_const) * 255;
      }
   }
   *color_out = tcolor_out;
}
////////// FUNCTIONS ////////////////////////////////
float prandLifespanMC(float addr){
#pragma HLS PIPELINE

   int y, p, y1, i;

   y = 123456789;
   p = 156; // top Limit
   i = addr + 200;
   y1 = (((i*y*234525)/53112) % p);

   // Make it Float
   float temp = y1;
   // Make it positive
   float prandt1 = temp + 1100;
   float prandt2 = prandt1 * prandt1;
   float prand = sqrt(prandt2);

   return prand;
}
//////////////////////////////////////////////////////
```

8.11 ParticleSystem (Repeller) - HLS code

In this component we update particle parameters and change their directions in case of collision with the repeler object (cube). We used pipeline and dataflow directives to optimize the performance of the implementation.

Listing 14: ParticleSystem (Repeller) HLS code

```
#include <math.h>
#include "ParticleSystem.h"

void ParticleSystem(
     // Inputs
     float addr_in, // number of current face
     float v1_x_in, float v1_y_in, float v1_z_in, // coordinates of
         vertex 1
     float v2_x_in, float v2_y_in, float v2_z_in, // coordinates of
```

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	148
FIFO	-	-	-	-
Instance	-	203	46792	72227
Memory	-	-	-	-
Multiplexer	-	-	-	2529
Register	-	-	44198	5103
Total	0	203	90990	80007
Available	890	840	407600	203800
Utilization (%)	0	24	22	39

Latency		Interval		
min	max	min	max	Type
155	155	1	1	function

Figure 113: Resources Utilization and Latency for ParticleSystem (Repeller) component

```
        vertex 2
    float v3_x_in, float v3_y_in, float v3_z_in, // coordinates of
        vertex 3
    float x_step_in, float y_step_in, // rate of change in x, y
        and z axes
    float collision_time_in, // collision time
    float check_in, // we use this flag type variable to check if
        it the first time
            // this particle is parsed
    float lifespan_in, // remaining lifetime

    // Outputs
    // these variables are the updated inputs
    float *addr_out,
    float *v1_x_out, float *v1_y_out, float *v1_z_out,
    float *v2_x_out, float *v2_y_out, float *v2_z_out,
    float *v3_x_out, float *v3_y_out, float *v3_z_out,
    float *x_step_out, float *y_step_out,
    float *collision_time_out,
    float *check_out, float *lifespan_out){
#pragma HLS PIPELINE

// Output temp Variables
float tv1_x_out, tv1_y_out, tv1_z_out;
float tv2_x_out, tv2_y_out, tv2_z_out;
float tv3_x_out, tv3_y_out, tv3_z_out;
float tx_step_out, ty_step_out;
float tcollision_time_out;
float tcheck_out, tlifespan_out;
```

```
// we get a random value for maximum
// lifespan of current particle
float lifespan_const = prandLifespan(addr_in);

// a is the value for the acceleration
// of the particles
float a = 2.5;

// t is the current time variable for the
// equation of motion
float t = 0;

// when t variable reaches its maximum time
// it dies
float max_time = lifespan_const/1000;

// if addr_in is between 0 and 5, vertex coordinates
// belong to the cube we use as the repeller object
// cube's vertices have been initialized in .coe file
if (addr_in == 0 || addr_in == 1 || addr_in == 2 || addr_in == 3 ||
    addr_in == 4 || addr_in == 5){

   // ...so we do not update anything
   // we write the same values to memory
   tv1_x_out = v1_x_in;
   tv1_y_out = v1_y_in;
   tv1_z_out = v1_z_in;

   tv2_x_out = v2_x_in;
   tv2_y_out = v2_y_in;
   tv2_z_out = v2_z_in;

   tv3_x_out = v3_x_in;
   tv3_y_out = v3_y_in;
   tv3_z_out = v3_z_in;
   ////////////////////////////////////////
   tx_step_out = 0;
   ty_step_out = 0;
   tcollision_time_out = 0;
   tcheck_out = 0;
   tlifespan_out = lifespan_const;
   ////////////////////////////////////////
}
else{ // if we have particle
```

```
// check is zero only the first time we update
// current's particle parameters, then is always 1
if (check_in == 0){

   // Generate Random Numbers X, Y
   // prX, prY are the coordinates where the
   // particle starts moving from. It is also the rate
   // of change in each axis.
   float prX = prandX(addr_in);
   float prY = prandY(addr_in);

   // we initialize time -> 0
   t = max_time - (lifespan_const/1000); // Time Variable

   // setting the coordinates of the vertices
   tv1_x_out = prX; // VertexA
   tv1_y_out = (prY*t) - (a*t*t); // y depends on kinematic
       equation
   tv1_z_out = 0.50;

   tv2_x_out = tv1_x_out - 0.01; // VertexB
   tv2_y_out = tv1_y_out - 0.02; // we place it lower then vertexA
   tv2_z_out = 0.50;

   tv3_x_out = tv1_x_out + 0.01; // VertexC
   tv3_y_out = tv1_y_out - 0.02; // we place it lower then vertexA
   tv3_z_out = 0.50;

   ////////////////////////////////////////
   // these values are being written to memory
   // in order to get them when we want to update
   // the same particle again
   tx_step_out = prX;
   ty_step_out = prY;
   tcollision_time_out = 0;
   tcheck_out = 1;
   tlifespan_out = lifespan_const;
   ////////////////////////////////////////
}
else if (check_in == 1){
   // lefespan_in = 0 and we initilize particle's
   // paramaeters (do the same as we did when check was 0.
   if (lifespan_in == 0){
      // Generate Random Numbers X, Y
      float prX = prandX(addr_in);
```

```
    float prY = prandY(addr_in);

    t = max_time - (lifespan_const/1000); // Time Variable

    tv1_x_out = 0.00; // VertexA
    tv1_y_out = (prY*t) - (a*t*t);
    tv1_z_out = 0.50;

    tv2_x_out = tv1_x_out - 0.01; // VertexB
    tv2_y_out = tv1_y_out - 0.02;
    tv2_z_out = 0.50;

    tv3_x_out = tv1_x_out + 0.01; // VertexC
    tv3_y_out = tv1_y_out - 0.02;
    tv3_z_out = 0.50;

    ////////////////////////////////////////
    tx_step_out = prX;
    ty_step_out = prY;
    tcollision_time_out = 0;
    tcheck_out = 1;
    tlifespan_out = lifespan_const;
    ////////////////////////////////////////
}
else{
    // here we update the parameters
    // t variable depends on remaining lifetime
    t = max_time - (lifespan_in/1000);

    tv1_x_out = v1_x_in + x_step_in; // VertexA
    tv1_y_out = (y_step_in*t) -a*t*t;
    tv1_z_out = v1_z_in + x_step_in;

    tv2_x_out = tv1_x_out - 0.01; // VertexB
    tv2_y_out = tv1_y_out - 0.02;
    tv2_z_out = v1_z_in + x_step_in;

    tv3_x_out = tv1_x_out + 0.01; // VertexC
    tv3_y_out = tv1_y_out - 0.02;
    tv3_z_out = v1_z_in + x_step_in;

    // TopCollision
    if (tv1_y_out >= -0.205 && tv1_y_out <= -0.2 && tv1_x_out
        <= 0.20 && tv1_x_out >= 0.05 && tv1_z_out >= 0.0 &&
        tv1_z_out <= 1.0){
```

```
         tcollision_time_out = t;
         tcheck_out = 2;
      }
      // RightCollision
      else if (tv1_y_out < -0.205 && tv1_y_out >= -0.4 &&
          tv1_x_out >= 0.05 && tv1_z_out >= 0.0 && tv1_z_out <=
          1.0){
         tcollision_time_out = 0;
         tcheck_out = 3;
      }
      else{
         tcollision_time_out = 0;
         tcheck_out = 1;
      }
      ////////////////////////////////////////
      tx_step_out = x_step_in;
      ty_step_out = y_step_in;

      tlifespan_out = lifespan_in - 1;
      ////////////////////////////////////////
   }

}
else if(check_in == 2){ // Top Collision
   if (lifespan_in == 0){
      // Generate Random Numbers X, Y
      float prX = prandX(addr_in);
      float prY = prandY(addr_in);

      t = max_time - (lifespan_const/1000); // Time Variable

      tv1_x_out = 0.00; // VertexA
      tv1_y_out = (prY*t) - (a*t*t);
      tv1_z_out = 0.50;

      tv2_x_out = tv1_x_out - 0.01; // VertexB
      tv2_y_out = tv1_y_out - 0.02;
      tv2_z_out = 0.50;

      tv3_x_out = tv1_x_out + 0.01; // VertexC
      tv3_y_out = tv1_y_out - 0.02;
      tv3_z_out = 0.50;

      ////////////////////////////////////////
      tx_step_out = prX;
```

```
      ty_step_out = prY;
      tcollision_time_out = 0;
      tcheck_out = 1;
      tlifespan_out = lifespan_const;
      ////////////////////////////////////////
   }
   else{
      t = max_time - (lifespan_in/1000);

      tv1_x_out = v1_x_in + x_step_in*1.5; // VertexA
      tv1_y_out = -0.2 +((y_step_in/2)*(t-collision_time_in))
          -a*(t-collision_time_in)*(t-collision_time_in);
      tv1_z_out = v1_z_in + x_step_in;

      tv2_x_out = tv1_x_out - 0.01; // VertexB
      tv2_y_out = tv1_y_out - 0.02;
      tv2_z_out = v1_z_in + x_step_in;

      tv3_x_out = tv1_x_out + 0.01; // VertexC
      tv3_y_out = tv1_y_out - 0.02;
      tv3_z_out = v1_z_in + x_step_in;

      ////////////////////////////////////////
      tx_step_out = x_step_in;
      ty_step_out = y_step_in;
      tcollision_time_out = collision_time_in;
      tcheck_out = 2;
      tlifespan_out = lifespan_in - 1;
      ////////////////////////////////////////
   }
}
else if(check_in == 3){ // Right Collision
   if (lifespan_in == 0){
      // Generate Random Numbers X, Y
      float prX = prandX(addr_in);
      float prY = prandY(addr_in);

      t = max_time - (lifespan_const/1000); // Time Variable

      tv1_x_out = 0.00; // VertexA
      tv1_y_out = (prY*t) - (a*t*t);
      tv1_z_out = 0.50;

      tv2_x_out = tv1_x_out - 0.01; // VertexB
      tv2_y_out = tv1_y_out - 0.02;
```

```
            tv2_z_out = 0.50;

            tv3_x_out = tv1_x_out + 0.01; // VertexC
            tv3_y_out = tv1_y_out - 0.02;
            tv3_z_out = 0.50;

            ////////////////////////////////////////
            tx_step_out = prX;
            ty_step_out = prY;
            tcollision_time_out = 0;
            tcheck_out = 1;
            tlifespan_out = lifespan_const;
            ////////////////////////////////////////
         }
         else{
            t = max_time - (lifespan_in/1000);

            // change direction on x axis
            tv1_x_out = v1_x_in - x_step_in; // VertexA
            tv1_y_out = (y_step_in*t) -a*t*t;
            tv1_z_out = v1_z_in + x_step_in;

            tv2_x_out = tv1_x_out - 0.01; // VertexB
            tv2_y_out = tv1_y_out - 0.02;
            tv2_z_out = v1_z_in + x_step_in;

            tv3_x_out = tv1_x_out + 0.01; // VertexC
            tv3_y_out = tv1_y_out - 0.02;
            tv3_z_out = v1_z_in + x_step_in;

            ////////////////////////////////////////
            tx_step_out = x_step_in;
            ty_step_out = y_step_in;
            tcollision_time_out = 0;
            tcheck_out = 3;
            tlifespan_out = lifespan_in - 1;
            ////////////////////////////////////////
         }
      }
   }
      *addr_out = addr_in;
      *v1_x_out = tv1_x_out;
      *v1_y_out = tv1_y_out;
      *v1_z_out = tv1_z_out;
      *v2_x_out = tv2_x_out;
```

```
   *v2_y_out = tv2_y_out;
   *v2_z_out = tv2_z_out;
   *v3_x_out = tv3_x_out;
   *v3_y_out = tv3_y_out;
   *v3_z_out = tv3_z_out;
   *x_step_out = tx_step_out;
   *y_step_out = ty_step_out;
   *collision_time_out = tcollision_time_out;
   *check_out = tcheck_out;
   *lifespan_out = tlifespan_out;
}

////////////// FUNCTIONS ///////////////////
float prandX(float addr){
#pragma HLS PIPELINE

   int x, p, x1, i;
   x = 123456789;
   p = 289; // Up Limit
   i = addr + 200;
   x1 = (((i*x*234525)/53112) % p);

   // Make it Float
   float temp = x1;
   float prand = temp/800000;

   return prand;
}

float prandY(float addr){
#pragma HLS PIPELINE

   int y, p, y1, i;
   y = 123456789;
   p = 10789; // Up Limit
   i = addr + 200;
   y1 = (((i*y*219745)/53112) % p);

   // Make it Float
   float temp = y1;
   // Make it positive
   float prandt1 = temp/4000;
   float prandt2 = prandt1 * prandt1;
   float prand = sqrt(prandt2);
```

```
   return prand;
}

float prandLifespan(float addr){
#pragma HLS PIPELINE

   int y, p, y1, i;
   y = 123456789;
   p = 156; // Up Limit
   i = addr + 200;
   y1 = (((i*y*234525)/53112) % p);

   // Make it Float
   float temp = y1;
   // Make it positive
   float prandt1 = temp + 1100;
   float prandt2 = prandt1 * prandt1;
   float prand = sqrt(prandt2);

   return prand;
}
//////////////////////////////////////////
```

Bibliography

[1] Kasik V. and A. Kurecka, *FPGA Implementation of a Simple 3D Graphics Pipeline, 2015.*

[2] Ajay Kashyap and Ashish Sharma, *Implementation of a 3D Graphics Rasterizer with Texture and Slim Shader on FPGA, 2013.*

[3] Kyungsu Kim, Hoosung-Lee, Seonghyun Cho, Seongmo Park *Implementation of 3D graphics accelerator using full pipeline scheme on FPGA.* SoC Design Conference, 2008. ISOCC '08. International.

[4] Peter Szanto, Bela Feher *Implementing a Programmable Pixel Pipeline in FPGAs, 2008.*

[5] DK Design Suite.
`https://www.mentor.com/products/fpga/handel-c/dk-design-suite/`

[6] Jeong-Joon Yoo, Jaedon Lee, Sundeep Krishnadasan, Wonjong Lee, John Brothers, Soojung Ryu*Tile-based Path Rendering for Mobile Device.* ACM SIGGRAPH ASIA 2015, Symposium on Mobile Graphics and Interactive Applications (MGIA).

[7] Imagination Technologies, PowerVR Software Development Kit, Imagination Technologies.
`http://www.pvrdev.com`

[8] NVIDIA CUDA.
`http://www.nvidia.com/object/cuda_home_new.html`

[9] Fang Liu, Meng-Cheng Huang, Xue-Hui Liu, En-Hua Wu *FreePipe: a Programmable Parallel Rendering Architecture for Efficient Multi-Fragment Effects.* ACM SIGGRAPH 2010 symposium on Interactive 3D Graphics and Games.

[10] Graphics pipeline.
`https://en.wikipedia.org/wiki/Graphics_pipelin`

[11] Clipping.
`https://en.wikipedia.org/wiki/Clipping_(computer_graphics)`

[12] Rasterization.
`https://en.wikipedia.org/wiki/Rasterisation`

[13] Software Rasterization Algorithms for Filling Triangles.
http://www.sunshine2k.de/coding/java/TriangleRasterization/TriangleRasterization.html

[14] Z-buffering.
https://en.wikipedia.org/wiki/Z-buffering

[15] Shading.
https://en.wikipedia.org/wiki/Shading

[16] Gouraud shading.
https://en.wikipedia.org/wiki/Gouraud_shading

[17] Basic Texture Mapping.
http://ogldev.atspace.co.uk/www/tutorial16/tutorial16.htm

[18] World, View and Projection Transformation Matrices.
http://www.codinglabs.net/article_world_view_projection_matrix.aspx

[19] Matrix.LookAtLH(Vector3,Vector3,Vector3) Method (Microsoft.DirectX).
https://msdn.microsoft.com/en-us/library/windows/desktop/bb281710(v=vs.85).aspx

[20] Explaining Homogeneous Coordinates and Projective Geometry.
http://www.tomdalling.com/blog/modern-opengl/explaining-homogenous-coordinates-and-projective-geometry/

[21] Perlin noise.
https://en.wikipedia.org/wiki/Perlin_noise

[22] The original code from Perlin was originally published in java.
http://rosettacode.org/wiki/Perlin_noise

[23] Particle system.
https://en.wikipedia.org/wiki/Particle_system

[24] Displacement mapping.
https://en.wikipedia.org/wiki/Displacement_mapping

[25] Introduction to FPGA Technology.
http://www.ni.com/white-paper/6984/en/

[26] High-level synthesis.
https://en.wikipedia.org/wiki/High-level_synthesis

[27] Learning how to write a 3D Soft Engine from scratch in TypeScript or JavaScript.
https://blogs.msdn.microsoft.com/davrous/2013/06/13/tutorial-series-learning-how-to-write-a/.../-javascript

[28] Xilinx Vivado Design Suite.
http://www.xilinx.com/products/design-tools/vivado.html

[29] Xilinx Kintex-7 FPGA KC705 Evaluation Kit.
http://www.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html

[30] ADV7511 HDMI Transmitter.
http://www.analog.com/en/products/audio-video/analoghdmidvi-interfaces/analog-hdmidvi-display-interfaces/adv7511.html

[31] VGA timings.
http://hamsterworks.co.nz/mediawiki/index.php/VGA_timings