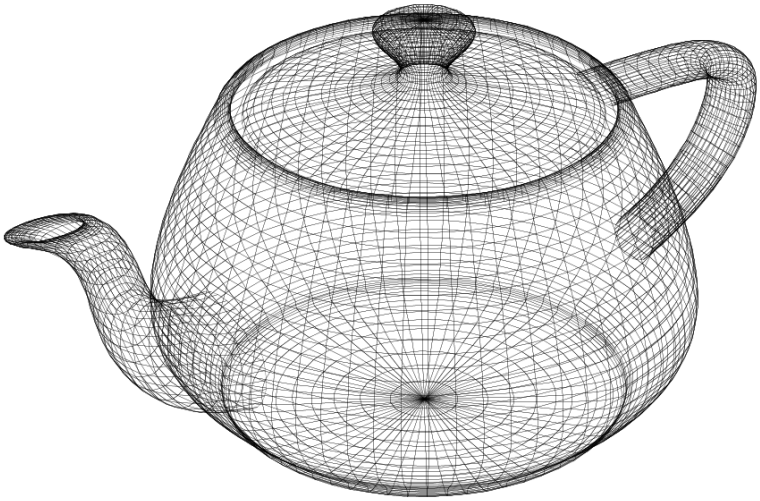
REAL-TIME RASTERIZATION ON THE STARBURST MPSOC

OĞUZ METEER



Computer Architecture for Embedded Systems
EWI
University of Twente

April 2017 – version 1.0

Supervising committee:

Prof.dr.ir. M.J.G. Bekooij
Ir. J. Scholten
G. Kuiper, MSc.

ABSTRACT

Formally proving real-time behavior of commercial graphics cards is difficult due to their closed nature and high complexity. This thesis researches the viability of developing a real-time graphics architecture on the Starburst Multi-Processor System on Chip (MPSoC) platform. Multiple graphics architectures are evaluated, and one is selected that fits the platform and allows real-time behavior. Based on the chosen architecture, a reference system that runs fully in software is implemented and analyzed. Also a hardware accelerated system is designed and built to remove bottlenecks of the reference system. The real-time behavior and performance, as well as the scalability of the hardware accelerated system are analyzed.

ACKNOWLEDGMENTS

First, I would like to thank my supervisor Marco Bekooij, for giving me the opportunity to combine what I love doing in my master thesis project, namely hardware and graphics. He also gave me the right insight which led me to trying different approaches to solve some of the challenges, and supported me when faced some health issues during my research.

I would also like to thank Cecill, for giving me the idea to pick a graphics related topic for my thesis, and sharing his ideas with me. And a special thanks is in order for my neighbor, Viktorio, for sticking with me throughout my research, and for letting me use the Bulgarian side of the whiteboard.

But the biggest thanks go to my family, who supported me throughout my years of studying. Especially my parents did everything they possibly could to make sure that I could pursue my quest for knowledge. I truly am proud and blessed to have such amazing parents.

CONTENTS

1 INTRODUCTION

1.1 BACKGROUND

Hardware for accelerating the rendering of 3D graphics has been around for many years. Advancements in technology and cheaper components made it possible to make 3D graphics accessible to use in regular personal computers. Graphics accelerators have since evolved and with each iteration, they became faster, more capable and programmable, and contain more and faster memory. As a result, this paved the way to achieve rendering more realistic looking graphics in order to immerse users in 3D worlds.

Graphics accelerators are developed being optimized for best *average* case performance, employing more tricks and complexity, which need to be taken into account as performance of graphics accelerators heavily depend on how they are used. This makes them difficult to classify as real-time systems. When playing a game or viewing or modeling highly complex 3D models, the consequences of potentially dropping frames temporarily might be experienced as annoying, but they are not disastrous. While it might seem that real-time acceleration of graphics, from a real-time systems view point, is not a necessity, there are however domains where it is required or at least would be beneficial.

One example is Virtual Reality (VR) applications. Just as graphics accelerators made it possible to render immersive 3D worlds at interactive frame rates, VR headsets can be seen as another tool to increasing the level of immersion by making users feel as if they really are within a fictitious 3D world. However, a prominent issue that early prototypes of VR headsets had, is that they caused motion sickness for some users [34]. Among the mentioned causes was low head-tracking rate, leading to a noticeable delay between moving the head and seeing the results, and low precision head-tracking. While manufacturers claim to have solved these problems, it doesn't change the fact that the brain is sensitive to inconsistencies between head movement and visual perception, and thus can lead to motion sickness for some users [34]. But high precision head-tracking and fast update rates are only a part of the chain, and VR headsets would greatly benefit having graphics accelerators that guarantee having a low and consistent latency, and a fast and consistent rendering rate. This is where a graphics accelerator that is formally proven to provide real-time guarantees given certain requirements comes into play, as current graphics accelerators cannot give such guarantees.

Augmented Reality is another example, for instance in military applications. Military vehicles equipped with various types of camera sensors combined with algorithms to create a 3D model of an area or target, can be a valuable tool for soldiers of the future. A practical example of this are pilots of fighter jets flying at night, where the human eye is incapable of seeing. They could wear a headset with a transparent screen that overlays a 3D model of a potential target, giving them another tool in dangerous situations. Real-time guarantees regarding a consistent rendering performance is an important requirement in such systems.

1.2 PROBLEM DESCRIPTION

Our goal is to design and implement a real-time rasterization algorithm on the Starburst platform, so naturally the following research question is defined:

Is the Starburst architecture suitable for implementing a real-time rasterization algorithm? If not, which hardware and/or software components need to be incorporated to make it suitable?

To answer this question, we first need to understand what obstacles there are to creating a such a system from a hardware and software perspective. After discovering which bottlenecks there are and where they are located, solutions to alleviate them have to be researched. This involves evaluating the balance in the usage of hardware and software that give higher, more predictable, and more consistent performance, and more flexibility respectively. We therefore define the following objectives for this thesis:

1. Research and evaluate various graphics architectures —
    a) Determine which architecture is the most suited to evaluate its real-time characteristics;
    b) Identify any obstacles that make it difficult to evaluate those characteristics.
2. As a reference, implement a software rasterizer on the Starburst MPSoC —
    a) Evaluate the performance and real-time characteristics of the software rasterizer implementation;
    b) Identify bottlenecks in the implementation that prevent us being able to give real-time guarantees.
3. Realize an embedded system that performs real-time triangle rasterization on the Starburst MPSoC —
    a) Develop efficient accelerators that lessen or remove previously identified bottlenecks in the software rasterizer implementation;

b) Evaluate the performance and real-time characteristics of the system.

c) Identify any bottlenecks that still remain in the system.

We consider a design and implementation suitable if we can successfully determine the worst-case performance and if we can guarantee that the system performance will never fall below the previously determined performance figures. The performance of graphics architectures is usually quantified as the amount of Frames Per Second (FPS). But since the goal is to have a low latency between the input (such as head movement in a VR application) and the rendered result, the execution time of each frame also has an upper bound and we cannot use the average of the amount of frames rendered per second.

1.3 CONTRIBUTIONS

The main contributions of this thesis are given below:

1. We evaluate different graphics hardware architectures and algorithms that best enable us to reason about them from a real-time systems perspective. (Chapter 2)
2. We reason that the approach of designing a real-time graphics architecture using the Worst-Case Execution Time (WCET) could yield a vastly under-utilized implementation. (Chapter 2)
3. We show that it is not possible to give temporal guarantees based on the real-time systems methodology.
4. We show that it is possible to create a *reproducible* system: *a system that for any specific input, it will have almost the same temporal behavior for each subsequent execution*. This enables us to still give real-time guarantees under the assumption that the inputs are known. (Chapter 2)
5. We implement a triangle rasterization algorithm with reproducible results on Starburst, a real-time system for streaming applications. (Chapter 4)

1.4 THESIS OUTLINE

This chapter has introduced the topic and research questions, and describes the organization of this report.

Chapter 2 first gives a brief history of graphics accelerators, and then describes the graphics rendering pipeline. Furthermore, several rasterization algorithms and graphics architectures are given, followed by which algorithm and architecture is chosen to implement a real-time graphics rasterizer suitable for implementation on our platform.

Chapter 3 gives an overview of the platform on which the project is implemented on, and continues with describing the proposed architecture.

In chapter 4, implementation details of the chosen architecture are described.

In chapter 5, the results are presented and the implementation is evaluated. Finally, in chapter 6, we answer the research questions, discuss future work, and conclude this thesis.

2 THE GRAPHICS RENDERING PIPELINE

In this chapter, the conceptual stages of the graphics rendering pipeline is explained, followed by various rasterization algorithms and graphics architectures. A rationale for choosing a specific algorithm and architecture is given, and finally we explain why using just the WCET to design a real-time graphics architecture can lead to vastly underutilized hardware.

2.1 THE GRAPHICS RENDERING PIPELINE

The graphics rendering pipeline takes three-dimensional objects or *models* forming a 3D environment, textures, light sources, and shading equations, and uses it to create a two-dimensional image as seen through a virtual camera. This process is called *rendering*, and there are several rendering methods that solve the shading equations in a different way, depending on the required results and available performance. All methods use a virtual camera and screen, that is the *viewport* into the 3D environment which displays the resulting 2D image. An example scene is illustrated in Figure 1.

The most often used rendering method employed in computers, gaming consoles, and mobile devices, is *rasterization*. It takes *rendering primitives* such as points, lines, and polygons, applies several transformations on them, rasterizes them on a 2D image, and *shades* each pixel of the visible polygons. Rasterization will be explained in more detail in the next section.

Ray tracing is another rendering method that is most often used in movies and game cinematics. The name of this method comes from the fact that each pixel is shaded by tracing a ray from the camera, through the virtual screen, bouncing off of the 3D objects until it hits a light source. The results usually are very realistic as ray tracing mostly resembles how our eyes see objects. Because of this, physical phenomena such as reflections and refraction basically come for free with ray tracing, whereas with rasterization, this requires tricks and workarounds to achieve. The downside of this method however, is that in general, it requires more performance compared to rasterization.

2.2 ARCHITECTURE

An often used method of increasing the throughput of a system is *pipelining*, where a large task is divided into several stages, and all

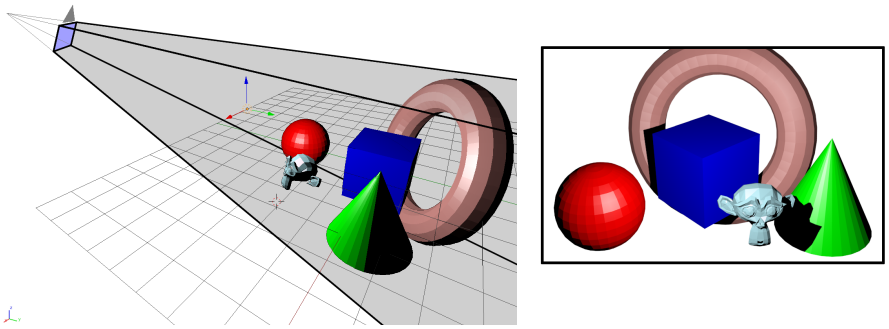


Figure 1: Displayed in the left image is an example of a scene with several 3D models, a light source, and a virtual camera. The camera position is de tip of the pyramid, the light blue plane in the pyramid is the *near plane* or *image plane*, which is the virtual screen or *viewport* into the world. The gray volume represents the *viewing frustum* or the volume in which models are visible. In the right image, the same scene is shown projected onto the virtual screen as seen through the virtual camera. The cube and the monkey head models cast shadows onto the torus and cone models respectively, and the the torus partially falls outside the viewing frustum, and is therefore clipped.

stages execute in parallel, consuming the output of the previous stage. While pipelining does not change the *latency*, i.e. the amount time needed for an input to be processed, it could potentially give a speedup in throughput proportional to the amount of stages. However, to achieve this speedup, the pipeline is required to be full at all times, meaning that most, if not all stages should be utilized. Since pipelining is inherently serial, this also means that the throughput of a pipelined system is determined by its slowest stage.

The throughput of the graphics rendering pipeline, also called the rendering speed, is the amount of new images the system can provide, and is usually expressed in *frames per second* (fps) and in Hertz. However, more important is the variance in rendering speed (REF) as it negatively impacts how humans perceive animations. For example, a system which outputs frames consistently in 33 ms (30 Hz) is usually perceived as more pleasant than a system which outputs frames with an average of 16.6 ms (60 Hz), but often has frames that take much longer to render.

Another way to potentially increase the performance of a system is *parallelization*, where a single task is broken up into multiple parts which are processed at the same time. As we will see in the remainder of this chapter, rendering and rasterization is an embarrassingly parallel process, which is why modern graphics hardware exploit this fact as much as possible.

From a high-level viewpoint, the graphics rendering pipeline can be divided into three *conceptual* stages: *application*, *geometry*, and *rasterizer*, and is shown in Figure 2. These conceptual stages often consist

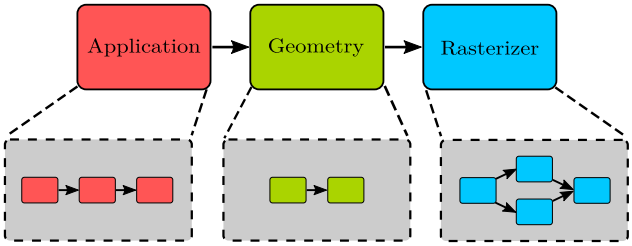


Figure 2: The conceptual stages of the graphics rendering pipeline. Each stage can have pipelined internal stages, as shown beneath the conceptual stages. Some internal stages can also contain parallelized stages.

of multiple, pipelined and parallel internal stages called *functional* stages.

2.3 APPLICATION

The rendering process starts at the *application* stage, and is tasked with setting up and providing the necessary data to the rest of pipeline. It is usually divided into at least two sub-stages: *user application* and *graphics library*. The user application creates a virtual world and a 3D scene to be rendered by setting up the coordinate system and a virtual camera, loading 3D models consisting of rendering primitives, placing these models in the world, loading materials associated with models, and setting up light sources and shading equations that determine how models should be lit. High-level optimizations can also be implemented in this stage such as hierarchical view frustum culling (REF). Other tasks not directly related to rendering that the user application stage can fulfill range from handling user input to performing physics calculations.

When graphics accelerators first entered the market, they all required different ways of setting up the hardware, which made it difficult to write portable software that runs on different hardware. To provide an abstract way of utilizing graphics accelerators, *graphics libraries* were developed, where user applications talk to a standardized Application Programming Interface (API). Hardware vendors provide an implementation of these graphics libraries, which generate batches of commands for the hardware to perform the necessary graphics operations.

Graphics hardware is heavily pipelined and massively parallel, and its efficiency therefore depends of the level of utilization of the stages. Because of this, graphics libraries aim to generate large batches of commands in order to increase hardware utilization. Some modern graphics hardware even support multiple command streams and asynchronous computing [1].

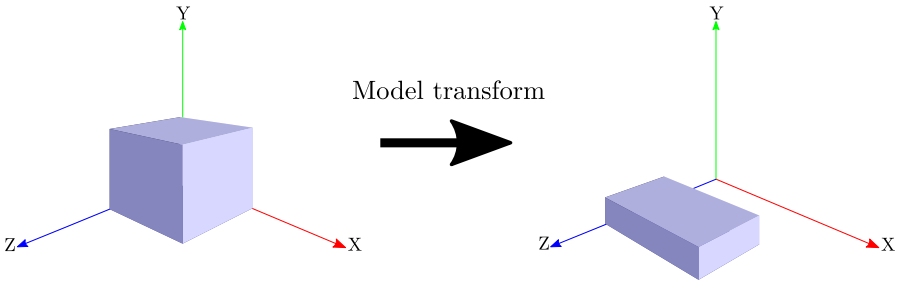


Figure 3: A 3D model risiding in model space can be transformed to world space using the model transform. On the left is a cube in the coordinate system of the model and on the right is the same model but translated in the z axis, and scaled in the x- and y-axis.

2.4 GEOMETRY

Rendering a 3D scene as seen from a virtual camera requires that models need to be positioned, oriented, and scaled correctly, and the light sources should light up the models as needed. In order to do so, the rendering primitives received from the application stage need to be processed, which is what the geometry stage is responsible for. This stage applies per-vertex and per-polygon operations on the received rendering primitives in order to prepare for the rasterizer stage.

The application stage delivers geometry in the form of rendering primitives such as points, lines, and polygons. We will assume that only polygons will be processed by the pipeline as it is the more involved case. Although we refer to polygons, the actual used polygon type is the triangle because they have several nice properties that simplify rasterization:

- It is the simplest polygon that defines a planar surface (i.e. it has 2D characteristics), and all other polygons can be constructed using triangles. This gives finer control when defining geometry.
- A triangle is a simple polygon, which means that cannot intersect with itself.
- It is also a convex polygon, meaning that no line between its vertices will be outside the triangle.

These properties simplify determining if a point lies inside a triangle, and interpolation of vertex properties over the surface, enabling efficient hardware implementations. Please see section **??** for more information on why these properties simplify rasterization.

The vertices of the rendering primitives are represented using 4D *homogeneous coordinates* [35] which have the usual x, y, and z components, and a fourth, w component which is normally set to 1. When

this w component equals 1, the x, y, and z components of homogeneous coordinates correspond to an equivalent, regular 3D vector. Using this representation allows the use of 4x4 matrices and enables a uniform way of applying transformations to vertices.

The geometry stage generally consists of the following functional stages which will be explained below: model and view transform, vertex shading, projection, culling and clipping, and screen mapping.

2.4.1 *Model & View Transform*

3D models are usually created or *modeled* in 3D modeling software, where it resides in its own coordinate space called *model space*. In model space, the vertices of the model are relative to origin of the coordinate system of the modeling software. To place a model in the world, its vertices and normals need to be transformed so that it will reside in *world space* or *world coordinates*. This transformation that transforms objects in model space to world space is called the *model transform*, and is often expressed in the form of a 4x4 matrix. This *model matrix* combines rotation around each axis, scaling, and translation, and because homogeneous coordinates are used, all of these operations can be done with one vector-matrix multiplication. Each model can be assigned a unique model matrix so that every model can be positioned, oriented and scaled individually. Figure 3 illustrates the effects of the model transform.

The virtual camera resides in the same world space, and has a position and a direction vector. As an optimization that makes clipping and screen mapping easier, another transform called the *view transform* is applied to the geometry. The effect of this transform is that all models are translated and rotated as if the camera is placed in the origin, with its direction vector is pointing in the positive or negative z-axis (depending on the graphics library). When the view transform is applied to models, they go from *world space* to *camera space* or *eye space*. Figure 4 shows an example of the view transform.

2.4.2 *Vertex Shading*

The way a model looks is not just determined by its shape, but also by its physical properties that affect how the model responds to light. Determining this response is called *shading*, and it involves *materials* and *shading equations*. A material is a collection of properties that characterize its response to a light source, and shading equations materialize these properties. There are many shading equations ranging from just calculating the color and intensity of a material, to also taking reflection, refraction, specular components, etc. into account.

To achieve realistic results, it is often necessary to evaluate the shading equations per pixel in the *pixel shader* stage. Since it is done per

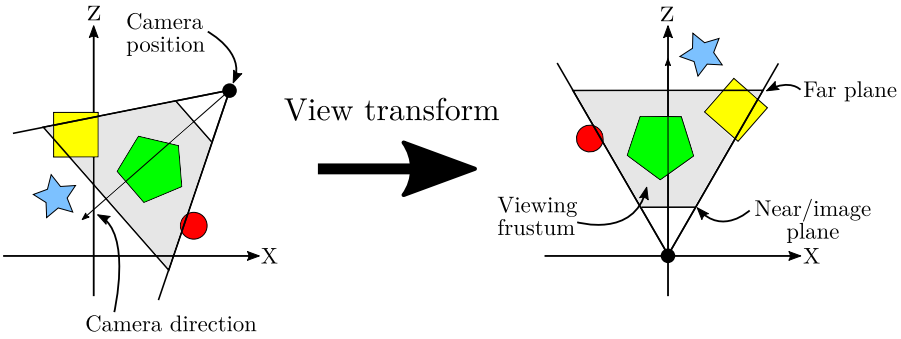


Figure 4: An example scene where four models and the virtual camera are transformed from world space to eye space using the view transform. The viewing frustum determines the volume in which models are visible, and is bound by the near and far planes. The near plane or image plane is the virtual screen on which the models in the viewing frustum are projected. In this case, all models except for the star intersect with the viewing frustum, and are therefore (partially) visible.

pixel or *fragment*, it is also called *per-pixel lighting*. In order to do per-pixel lighting, some setup has to be done per vertex and per polygon, which is the responsibility of the vertex shading stage. Depending on the shading equations and the desired result, the vertex shading stage can output data per vertex to help with shading, such as colors, normal vector, texture coordinates, etc. This output is then read by the rasterizer stage and interpolated over the surface of the polygon made up by those vertices.

2.4.3 *Projection*

The projection stage applies a *projection transform*, which *projects* visible models in the viewing volume onto the image plane. This projection is done in two steps:

- Vertices are transformed from eye space to *clip space* using the *projection matrix*. In this space *clipping* is performed which will be explained in the next section.
- Perspective division is performed. As noted before, 4D homogeneous coordinates correspond to 3D coordinates when the w component equals one, so this step divides all components of the 4D vertices by their w component.

The result of these operations is that the viewing volume is transformed into a unit cube called the *canonical view volume*, and the models inside are now in NDC space. Depending on the implementation and used graphics library, the size of the unit cube can vary but is usually (2,2,1) or (2,2,2).

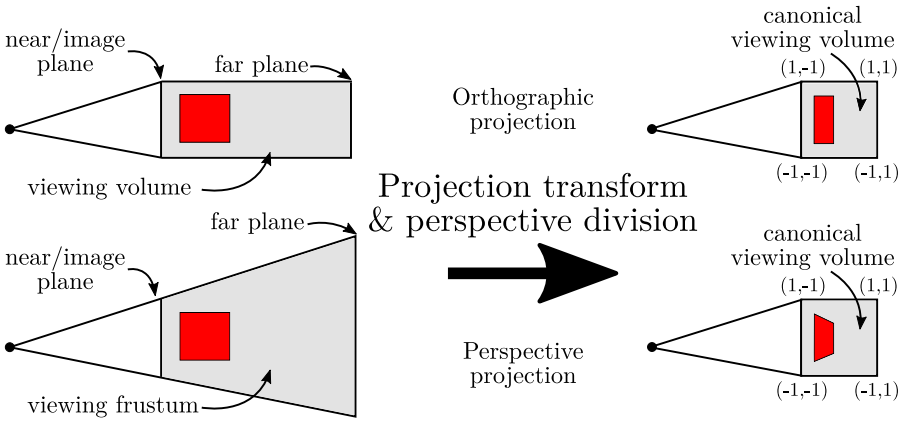


Figure 5: Application of projection transform and perspective division. Top image shows orthographic projection and bottom image shows perspective projection. In both cases, the models are transformed from eye space to NDC space.

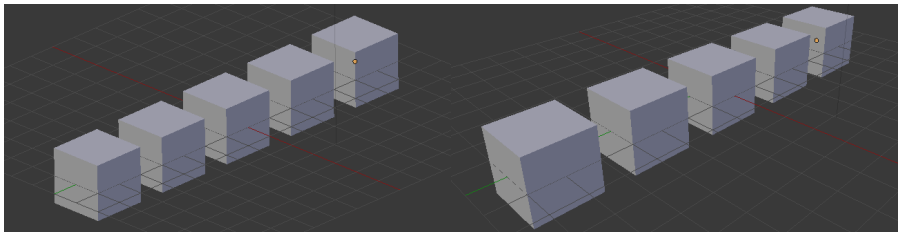


Figure 6: A scene rendered with orthographic projection on the left and with perspective projection on the right.

Two of the most often used projection methods are *orthographic* and *perspective* projection, illustrated in Figure 6.

Orthographic projection uses a rectangular viewing volume, and its associated 4x4 projection transform matrix consists only of translation and scaling, making it an affine transformation. Therefore, when transforming the viewing volume to a unit cube, the near and far planes of the volume are scaled equally. Orthographic projection simply disregards the depth or z component, which means that the projected size of a model is not influenced by the distance between that model and the image plane. Because parallel lines remain parallel after orthographic projection, it is also called *parallel projection*. See top image of Figure 5 for an example.

Contrary to orthographic projection, with perspective projection, models farther away from the image plane appear smaller than those that are closer, and parallel lines can converge at the horizon. This is because perspective projection uses a frustum as the viewing volume, meaning the projection transform compresses the far plane and expands the near plane, transforming the models accordingly. Bottom image of Figure 5 illustrates this.

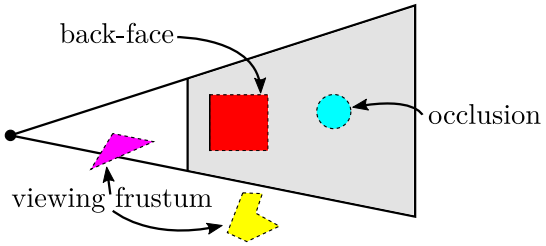


Figure 7: Example of three culling techniques: back-face, frustum, and occlusion, where culled primitives are represented by dashed lines. (Illustration inspired by [11])

2.4.4 *Culling and Clipping*

From a performance standpoint, it is desirable to only render models that are fully or partially visible, in order to limit performing unnecessary processing. *Culling* is the process of eliminating rendering primitives that are not visible at all. Commonly used culling techniques are:

- *Back-face culling* where back-facing polygons, whose normal vector points away from the camera, are eliminated.
- *View frustum culling* which removes entire models that fall outside the viewing frustum.
- *Occlusion culling* where models are removed that cannot be seen because they are entirely behind other models.

All of these culling techniques can be performed in the different transformation stages, and are therefore implementation dependent. Some techniques can be evaluated more easily in a specific space. For example, back-face culling is easy to perform in eye space as it only involves checking the dot product of the polygon normal and the camera direction vector which is simply (0,0,1,0) or (0,0,-1,0) depending on the implementation. View frustum culling on the other hand involves checking models against the six planes of the viewing frustum in eye space, but in clip space, an easier Axis-Aligned Bounding Box (AABB) check can be performed. Several culling examples are illustrated in Figure 7.

Clipping is the process of removing parts of rendering primitives that fall outside the region of interest. In the case of computer graphics, the vertices of triangles that intersect with the image plane or viewport are moved to the intersection points. If necessary, new vertices and thus triangles are added in order to preserve the same surface area that the intersecting part of the original triangle had. See Figure 8 for an example of clipping.

Clipping is an important operation to perform, as it prevents artifacts that can be caused when vertices with negative w components

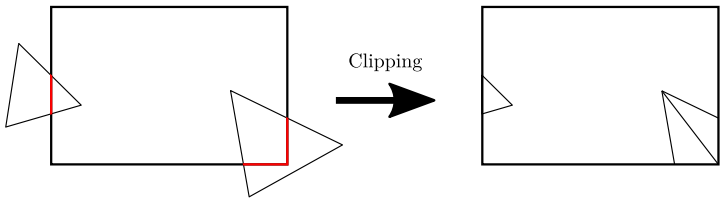


Figure 8: Triangles that partly fall outside the viewport are clipped against its borders. In this process, new vertices can be introduced as is the case for the right triangle.

are rasterized [26]. This can occur when vertices that lie behind the camera are transformed by the view matrix, which can create edges that "pass through infinity" and appear in front of the camera [32].

2.4.5 *Screen Mapping*

The final stage takes care of screen mapping where the unit cube is transformed to map to the screen that will display the rendered result. The w component of all vertices is 1 after perspective division and is dropped. The x- and y-coordinates are translated and scaled to match the resolution of the screen or window, which does not affect the z-coordinate (i.e. depth value). These values are then passed to the rasterizer stage. Models are transformed from NDC space to screen space with this final transform, and an overview of all transforms including screen mapping is given in Figure 9.

2.5 RASTERIZER

The rasterizer stage takes the transformed and projected vertices of all potentially visible models, and maps it to a raster of pixels that is the screen. This mapping process is called *rasterization*, and uses the vertices and vertex shading data from the previous stage to determine the color of each pixel that is occupied by models. The rasterizer stage is also responsible for solving the visibility problem, where the order to draw models is determined per pixel. It usually consists of the following functional stages which will be explained below: *triangle setup*, *triangle traversal*, *pixel shading*, and *merging*.

2.5.1 *Triangle Setup*

The triangle setup stage prepares per-triangle information necessary to be able to perform point-in-triangle checks necessary for the next stage. These checks are performed using *edge functions*, and before explaining their use, the choice of using triangles needs to be reiterated and explained in more detail.

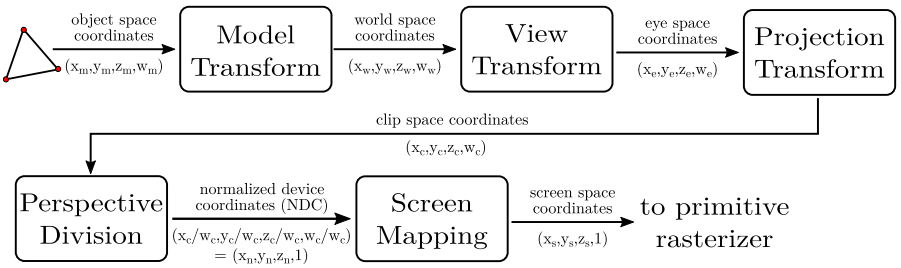


Figure 9: An overview of the transformations applied to a triangle in the geometry stage.

Triangles are polygons consisting of three vertices $(\mathbf{v_0}, \mathbf{v_1}, \mathbf{v_2})$ and edges $(\mathbf{e_{01}}, \mathbf{e_{12}}, \mathbf{e_{20}})$ between $(\mathbf{v_0}, \mathbf{v_1})$, $(\mathbf{v_1}, \mathbf{v_2})$, and $(\mathbf{v_2}, \mathbf{v_0})$ respectively. In the case of graphics rendering, the vertices coming from the geometry stage are three-dimensional, but the z-coordinates are not needed for performing point-in-triangle checks, therefore we view the vertices here as two-dimensional.

The order of these vertices determine the *winding order* or *orientation* of a triangle, which is clockwise or counterclockwise. Having defined an orientation allows us to view the edges of a triangle as directed edges. They are simple polygons, which means that they are not self-intersecting, and they define a planar surface meaning they have two-dimensional characteristics. Because of these properties, triangles divide a plane into a finite interior region and an infinite exterior region.

Edge functions are used to determine if a point lies on the "left" or "right" side of a line. They are implicit, affine equations of the form $ax + by + c = 0$, and given our definition of a triangle, we can describe the edge function through vertices $\mathbf{v_0}$ and $\mathbf{v_1}$ (i.e. for $\mathbf{e_{01}}$) as:

$$E(x,y) = (v_{1x} - v_{0x})(y - v_{0y}) - (v_{1y} - v_{0y})(x - v_{0x}) = ax + by + c \quad (1)$$

Equation 1 that can also be written as in the form:

$$E(x,y) = ax + by + c = \mathbf{n} \cdot (x,y) + c \quad (2)$$

A visual representation of the edge function is given in Figure 10.

Using Equation 1, we can describe edge functions for each of the three edges:

$$\begin{aligned} E_{01}(x,y) &= (v_{1x} - v_{0x})(y - v_{0y}) - (v_{1y} - v_{0y})(x - v_{0x}) \\ E_{12}(x,y) &= (v_{2x} - v_{1x})(y - v_{1y}) - (v_{2y} - v_{1y})(y - v_{1x}) \\ E_{20}(x,y) &= (v_{0x} - v_{2x})(y - v_{2y}) - (v_{0y} - v_{2y})(y - v_{2x}) \end{aligned} \quad (3)$$

When we rearrange the terms a bit we get:

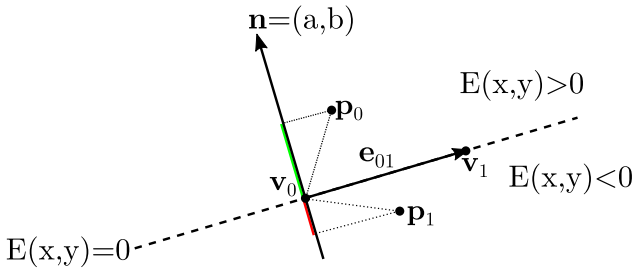


Figure 10: The edge function $E(x, y) = 0$ going through vertices $\mathbf{v_0}$ and $\mathbf{v_1}$ is shown as a dashed line. The normal vector comes from the edge function as defined in Equation 2. The edge function projects points onto the normal vector, and as an example two points are shown ($\mathbf{p_0}$ and $\mathbf{p_1}$). The projection of $\mathbf{p_0}$ yields a positive value shown as the green line, and the value for projecting $\mathbf{p_1}$ is negative as shown by the red line. Positive values (area above the dashed line) are on the "left" side of the edge $\mathbf{e_{01}}$ and negative values are on the "right" side and are not a part of the triangle.

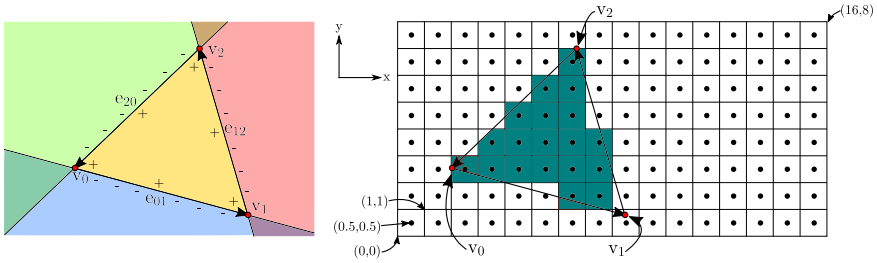


Figure 11: On the left: a triangle consisting of three vertices and edges as they are received from the geometry stage. For each edge, the "left" and "right" sides are shown with "+" and "-" signs and different colors. The yellow color marks the interior region of the triangle. On the right: the same triangle rasterized on a 16x8 raster of pixels. The dots in the center of the cells are the positions that are used for the point-in-triangle tests.

$$
\begin{aligned}
E_{01}(x,y) &= (v_{0y} - v_{1y})x + (v_{1x} - v_{0x})y + (v_{0x}v_{1y} - v_{0y}v_{1x}) \\
E_{12}(x,y) &= (v_{1y} - v_{2y})x + (v_{2x} - v_{1x})y + (v_{1x}v_{2y} - v_{1y}v_{2x}) \\
E_{20}(x,y) &= (v_{2y} - v_{0y})x + (v_{0x} - v_{2x})y + (v_{2x}v_{0y} - v_{2y}v_{0x})
\end{aligned} \tag{4}
$$

Because the vertices do not change within the rendering pipeline, we can consider them as constants that we can give them names (only constants for E_{01} are given here for brevity):

$$
\begin{aligned}
A_{01} &= v_{0y} - v_{1y} \\
B_{01} &= v_{1x} - v_{0x} \\
C_{01} &= v_{0x}v_{1y} - v_{0y}v_{1x}
\end{aligned} \tag{5}
$$

We can now rewrite Equation 4 as:

$$
\begin{aligned}
E_{01}(x,y) &= A_{01}x + B_{01}y + C_{01} \\
E_{12}(x,y) &= A_{12}x + B_{12}y + C_{12} \\
E_{20}(x,y) &= A_{20}x + B_{20}y + C_{20}
\end{aligned} \tag{6}
$$

Since edge functions are affine functions, the following holds true:

$$
\begin{aligned}
E_{01}(x+1,y) - E_{01}(x,y) &= A_{01} \\
E_{01}(x,y+1) - E_{01}(x,y) &= B_{01}
\end{aligned} \tag{7}
$$

This means that checking the pixel on the right of the current one requires adding A_{01}, and checking the pixel below requires adding B_{01} (in the case of edge $\mathbf{e_{01}}$). What the triangle setup stage therefore does is calculate these per-triangle constants, so that the edge functions do not have to be fully evaluated for every single pixel, enabling very efficient implementations.

2.5.2 *Triangle Traversal*

The triangle traversal stage performs the actual *rasterization* process, and determines *how* to look for candidate pixels in order to perform point-in-triangle tests. When a pixel inside the triangle is found, its visibility also needs to be checked, i.e. if it is occluded by other triangles or not. Triangle traversal is dependent on the implementation, and several of these rasterization methods will be explained below. More thorough and detailed explainations can be found in [5] and [37].

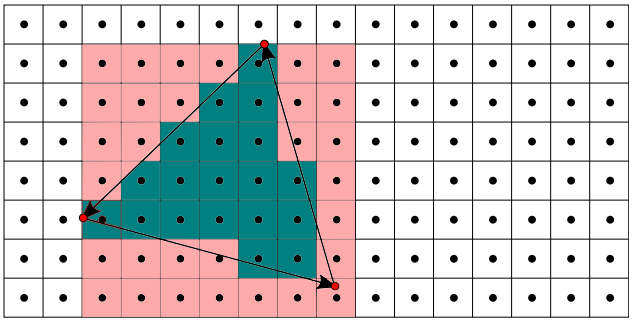


Figure 12: Bounding box traversal visits all pixels within the bounding of the triangle.

2.5.2.1 *Bounding Box Traversal*

The *bounding box traversal* is of the simplest traversal methods. It involves calculating the bounding box of a triangle and then evaluating the edge function for every single pixel inside it in any order, one pixel at a time. Although trivial to implement, its downside is it is potentially very inefficient as there might be many pixels that lie outside a triangle that will still be evaluated. See Figure 12 for an example.

2.5.2.2 *Backtrack Traversal*

A slightly more complex but also more efficient method is *backtrack traversal* that was used in older mobile hardware [48]. Traversal starts with the pixel occupied by the topmost vertex and processes pixels per scanline from left to right and top to bottom. When the pixel is outside to the right of the triangle, traversal jumps to the pixel below, and traverses to the left until the pixel is outside to the left of the triangle. This backtracking is done in order to find the starting point. During the backtracking step, no visibility checks are performed. This process is repeated until there are no more pixels to process within the bounding box.

2.5.2.3 *Zigzag Traversal*

An ever more efficient method is a variant of backtrack traversal is the zigzag traversal method described in [35] and [7]. As the name suggests, it traverses the pixels inside a triangle in a zigzag manner, instead of just during backwards traversal.

2.5.2.4 *Tiled Traversal*

Tiled traversal is a more expensive method of finding pixels inside a triangle, but also more efficient, since it visits every pixel inside a tile all at once. This is possible because determining whether a pixel lies inside a triangle can be performed independently of all other pixels, making it embarrassingly parallel.

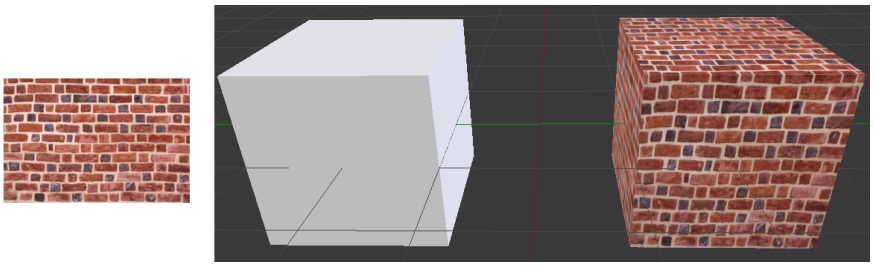


Figure 13: Two cubes are shown. The cube on the left has no texture, and the cube on the right is textured with the brick texture shown on the left.

Aside the performance increase of processing all pixels of a tile at once, tiled traversal also has the benefit of requiring less memory bandwidth. Since each tile can be processed independently, a tile can have its own small, on-chip memory for storing frame and depth buffers, requiring less round trips to the slower main memory. When all triangles that fall in a tile are processed, the content of the local buffers can be written to the actual frame and depth buffers.

Another benefit of tiled traversal is that memory accesses are more efficient. The other methods described above can access memory in irregular patterns depending on the position and shape of triangles. With tiled traversal, all pixels (or a regular pattern of pixels such as a row) are accessed in regular manner.

2.5.3 *Pixel Shading*

The pixel shading stage calculates per-pixel values using interpolated triangle data from the triangle setup stage. The minimum output of the this stage is per-pixel, or *fragment* colors, but it can have additional outputs that can be used in subsequent pixel shading passes. Modern GPUs have several fully programmable stages including the pixel shading stage, making it possible to implement various lighting equations and apply textures to achieve results ranging from photo realistic to cartoony. Figure 13 shows an example of texture mapping.

2.5.4 *Merging*

The output of the entire graphics pipeline is an image, which is a 2D array of colors called the *frame buffer*. The merging stage is responsible for composing this image, by merging fragments that come from the pixel shading stage with the color values currently stored in the frame buffer.

In almost all modern GPUs, the merging stage also solves the visibility problem, where objects closer to the camera should be drawn over objects that lie farther away. Traditionally, this problem was

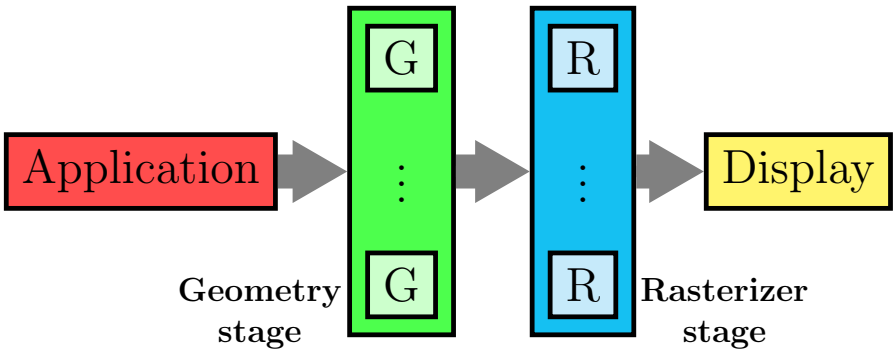


Figure 14: A general, parallel graphics processing architecture. Blocks marked **G** and **R** are *geometry* and *rasterizer* units respectively. *(Illustration after Möller et al. [6])*

solved in the application stage, using space partitioning and front-to-back sorting algorithms such as *BSP trees* [3], [27], [23], [22], and *k-d trees* [41].

Modern GPUs most often solve the visibility problem by using the *Z-buffer* [10], which is a 2D array of depth values for each fragment. During traversal, whenever a fragment inside the triangle is found, a depth lookup for that fragment is performed. If the depth of the fragment is closer to the camera than the retrieved value, then its color is written to the frame buffer. Being a trivial solution that works on a per fragment level, the added benefit is that no complex partitioning and sorting algorithms are needed.

The merging stage can also perform various blending operations such as *alpha blending*, where each fragment has an associated *alpha* value indicating how opaque or transparent it is. Using the Z-buffer, fully opaque fragments can be rendered in any order, but this is not the case with partially transparent fragments. They have to rendered in a back-to-front fashion after all opaque fragments have been rendered.

2.6 GRAPHICS ARCHITECTURES

Now that a general overview of the graphics rendering pipeline is given, we will discuss graphics architectures. Because each rendering primitive can be processed separately in the geometry stage, and the same applies to fragments in rasterizer stage, it makes sense that graphics architectures exploit the parallel nature of the rendering process. Figure 14 shows a general, parallel graphics processing architecture, where the geometry and rasterizer stages contain multiple processing units marked **G** and **R** respectively.

The idea is that multiple rendering primitives are processed in parallel, and that the partial results are merged into the final image. The application stage acts as a source of rendering primitives that are dis-

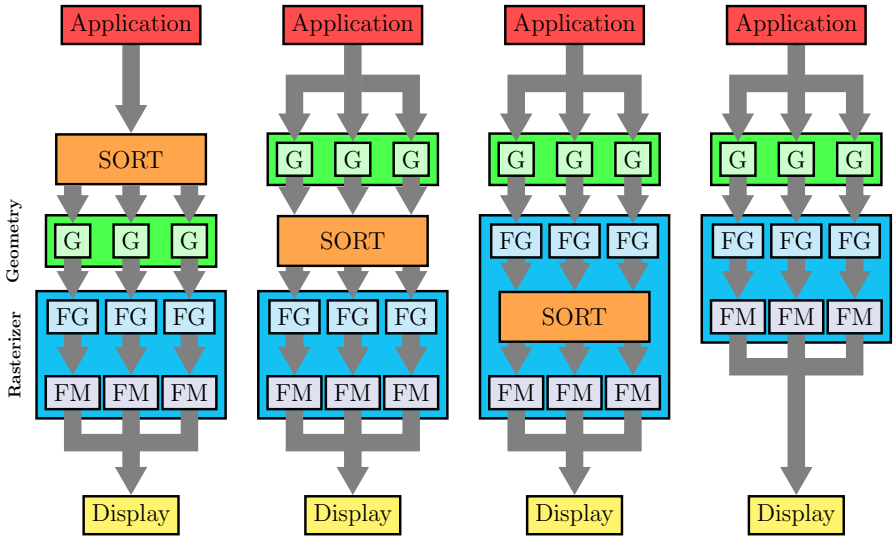


Figure 15: Four parallel graphics architectures, from left to right are *sort-first*, *sort-middle*, *sort-last fragment*, and *sort-last image*. The geometry stage consists of multiple blocks marked **G** that are geometry units. The rasterizer stage comprises of **FG** and **FM** blocks which are *fragment generation* and *fragment merging* respectively. *(Illustration after Möller et al. [6] and Eldridge et al. [19])*

tributed to the geometry units, which perform the geometry stage operations (transformation, vertex shading, etc.) on the primitives and then either cull them or send them to the rasterizer units. The rasterizer units then perform the rasterizer stage operations (triangle setup and traversal, shading, etc.), and finally the partial results of these processing units are merged into the final image.

Processing primitives in a parallel fashion introduces a couple of issues stemming from the fact that the locations and sizes of primitives on screen are arbitrary and are not known a priori. One issue is that primitives need to be sorted to determine which rasterizer units should render them [30]. This can be seen as sorting primitives to the screen [45].

Another issue is even work distribution of primitives and fragments to make efficient use of the hardware. A scene containing a few primitives that are close to the camera, i.e. cover a large part of the screen, can be computationally more expensive to render than many primitives that are far away or not in front of the camera.

Various parallel architectures dealing with these issues are presented in [30] and [19], and shown in Figure 15. These architectures are *sort-first*, *sort-middle*, *sort-last fragment*, and *sort-last image*. They differ in where the sorting takes place in the pipeline, and in all architectures the rasterizer stage is seperated into *fragment generation* (FG) and *fragment merging* (FM) units. We explain these architectures below.

2.6.1 *Sort-first*

The idea of the sort-first architecture is to sort primitives before the geometry stage. The screen is divided into disjoint regions, and primitives that fall into that region are handled by a full geometry and rasterizer pipeline. The sorting step involves performing just enough calculations to determine which regions or *tiles* a primitive falls into, usually by computing the screen-space bounding box of the primitive.

With sort-first, fragments can sometimes fall into regions of the wrong rasterizer units (because of the empty part of the bounding box for example), so they have to be either sent to another rasterizer, or to another pipeline. Both require interconnects, leading to increased hardware complexity. In the latter case, duplicate geometry computations need to be performed.

This architecture is of the least explored for single machine usage [30][31], and mostly sees usage in multi-screen setups where each screen is driven by a standalone system [40].

2.6.2 *Sort-middle*

In sort-middle architectures, the sorting happens between the geometry and rasterizer stages, when the primitives have been transformed into screen-space. On older hardware, geometry and rasterizer units were separate pieces of hardware, so it feels natural to sort the results of the previous pipeline stage before distributing the results to the next stage. After all primitives have been processed by the geometry units, the sorting step determines to which rasterizer(s) each primitive should be sent to. As with sort-first architectures, each rasterizer pipeline is responsible for its tile(s).

The benefits compared to sort-first are that there are no potentially duplicate geometry computations, and it is easier to load balance the geometry units because primitives can be arbitrarily assigned to them. One downside of this architecture is that the sorting step limits the primitive processing rate of the system, because it can only commence after all primitives have been processed before broadcasting them to the next stage. The impact depends on the tile size, and the right granularity has to be chosen find a balance between performance and hardware cost.

Being well researched [24][4][13], the sort-middle architecture has been used in both older and modern hardware, ranging from desktop computers [42] and gaming consoles [47][28] to embedded devices [43][2][36][17].

2.6.3 *Sort-last fragment*

The sort-last fragment architecture sorts processed fragments between the fragment generation and fragment merging units. The difference with sort-first and sort-middle architectures is that each primitive is fully processed by a single pipeline up until the FM unit, so there is no overlap with multiple rasterizer units and therefore no duplicate computations. Another added benefit is that there is no communication overhead in the form of broadcasting primitives to multiple rasterizer units compared to sort-first and sort-middle architectures. Once an FG unit is done processing, the resulting fragments are sent to FM units that merge fragments that fall into their region. This sorting step requires a simpler broadcasting method that in sort-middle architectures, as each fragment can only fall in one region and therefore only one FM unit has to merge it.

The downside of this architecture is that it can be difficult to load balance fragment generation work [18]. This can for example happen when a few large primitives are submitted to the same FG unit, which then needs to perform more work compared to FG units that process more primitives that however have a smaller area. Several examples of the sort-last fragment architecture are the PLAYSTATION®3 [6], the Evans & Sutherland Freedom 3000 [21], and the Kubota Denali [16].

2.6.4 *Sort-last image*

The sort-last image architecture performs the sorting step after the the rasterizer stage (FG and FM). It can be seen as multiple, independent pipelines that render the fragments that have been assigned to them into seperate, independent frame and depth buffers. The sorting step then composes the final image using these frame and depth buffers. Sorting is being performed last in order to scale and achieve higher performance, and this gives this architecture the the property that the order of fragments are not preserved.

Just like with the sort-last fragment architecture, a fragment is fully processed by a single pipeline, and therefore this architecture also inhibits the load-balancing problem. An additional downside is the vast amount of communication necessary to merge multiple full-size frame and depth buffers, though Roth and Reiners proposed an optimization [38]. One example of such an architecture is PixelFlow [20][29].

2.7 ARCHITECTURE CHOICE AND RATIONALE

Designing real-time rasterization hardware requires that we are able to analyze it such that real-time guarantees can be given. Often, more

elaborate designs have a higher average performance or throughput, but require more effort to analyze because they use complex hardware whose performance characteristics are hard to reason about. Examples are caches and DRAM memory controllers, and although we cannot always avoid using such hardware, we strive to minimize the use of such hardware.

As explained in Section 2.2, the performance of a system could be increased by the use of pipelining and parallelization. Older graphics hardware mostly focused on pipelining, such as the Geforce 3 GPU, consisting of hundreds of pipeline stages [**RTS3**]. More modern hardware exploit the fact that rendering and rasterization can be performed in parallel, as can be seen by the high core count of even low-end graphics hardware. The pipelined graphics architecture as described in this chapter results in a multiprocessing system. Such systems usually have two problems that were partly mentioned in Section 2.6: *load balancing* and *communication* [12]. While the choice of which architecture to use has an impact on the potential maximum performance of a system, it can also determine the minimum performance which is important in the case of real-time systems.

The restriction of minimizing hardware complexity aids in making the design easier and less costly to analyze it, but the worst-case performance must also meet the minimum requirements for the design to be viable. As with many designs, a trade-off between performance and complexity has to be made. In our case, any architecture that minimizes off-chip memory accesses is preferred, but a low variance in execution time is at least as important, and demonstrates the aforementioned trade-off which balances consistent performance and an analyzable architecture.

2.7.1 *Traversal Method*

Of all the traversal methods, tiled traversal is preferred because the rasterizer stage only uses local memory in the form of block RAM until the final result is written to the frame buffer. Reading and writing intermediate results therefore have consistent access times, easing analysis. We can also limit the amount of primitives each tile can process, which puts an upper bound on the execution time. Given that primitives are only rasterized by a fixed function stage, the WCET is trivial to compute. By changing the size of the tile, we can also change the amount of pixels that are processed in parallel. This gives us a trade-off between performance and hardware cost, without compromising the ease of analysis. Compared to the other traversal methods, tiled traversal does not suffer from the complexity of tracking the traversal position and direction, and has more consistent and potentially higher performance, depending on the tile size.

2.7.2 *Architecture*

For all graphics architectures, we put an upper limit on the amount of primitives to be rendered each frame, and we assume that each part of the pipeline has to fully process each primitive as this represents the worst-case scenario.

The sort-first architecture is the simplest of the four that we described, but because of its simplicity, it can lead to duplicate computations and requires interconnects between rasterizers or pipelines. This make analyzing the minimum performance of the system harder as it can be difficult to predict when and how often these duplicate computations and communication between pipelines have to be performed.

Sort-middle architectures somewhat retain the simplicity of sort-first architectures but offer more consistent performance. As primitives can be processed by any geometry units, there are no duplicate computations and no interconnects between pipelines are required. This improves load balancing and ease of analysis of the geometry processing part of the system. Because primitives can fall into arbitrary tiles, storing processed primitive data requires random memory accesses and potentially complex hardware for broadcasting primitive data to multiple rasterizer pipelines. While this analysis more difficult, we can make it easier by putting an upper limit on how many fragments each rasterizer pipeline may process, which also puts an upper bound on the number of memory accesses. Since the tiled traversal method is preferred and the rasterization step is well defined, it is easy to determine the WCET for fixed hardware implementations.

Both sort-last architecture variations offer higher potential performance and scalability. They also do not perform duplicate computations, and the sorting step involves less complex hardware compared to that in sort-middle architectures. One major disadvantage is that they suffer from load balancing problems as mentioned before. Because primitives can appear anywhere on screen and can have an arbitrary size, it makes it difficult to predict how much work every rasterizer pipeline will have to perform and consequently what the minimum performance is.

We believe that of all architectures, the sort-middle architecture offers the most consistent performance and is the easiest to analyze. It is therefore the preferred graphics architecture for developing a real-time rasterizer.

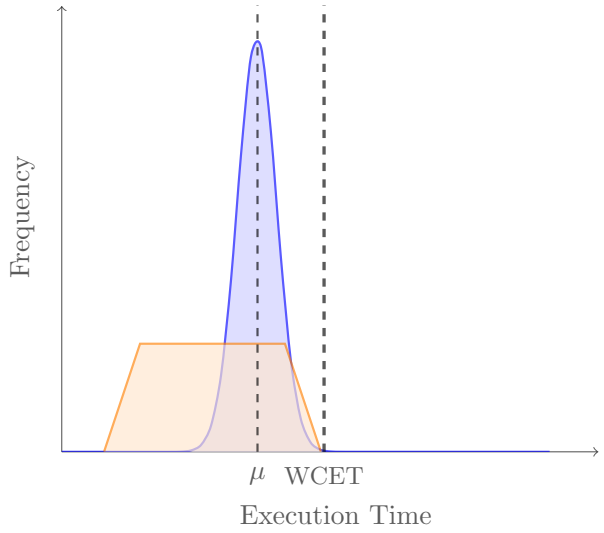


Figure 16: Distribution of execution times of a real-time system.

2.8 DESIGNING A REAL-TIME GRAPHICS ARCHITECTURE WITH REPRODUCIBLE RESULTS

Real-time systems are systems that give guarantees about temporal behavior such as maximum latency and minimum throughput. This is done by creating an abstract model of the system and analyzing it for all possible inputs. The difference between real-time and non real-time systems is that in real-time systems, the emphasis lies on predictability and not on attaining high average case performance. The most important aspect to determine is the WCET as it defines the upper bound and dictates what the real-time aspects such as latency and throughput will be. Complex systems with sophisticated interconnects and multiple levels of caches are difficult to analyze, making it hard to argue about their real-time performance as the WCET of such systems are difficult to determine.

A typical task in a real-time system could have execution time distribution similar to Figure 16, where it is guaranteed that the execution time of the task will never exceed the WCET. The execution time usually does not depend on *what* the data is, and the variance of the distribution is often caused due to unpredictability of for example shared resources, caches, and DRAM. Building a real-time system therefore involves designing an architecture that increases the predictability by minimizing complexity to a level where it can still be analyzed.

Unlike typical real-time systems, graphics systems have the property that the workload of the different stages *does* depend on *what* the input is. Also the workload of the stages are not directly correlated. Two examples that demonstrate this are illustrated in Figure 17. In the first example, a complex model is rasterized. Because the model has a

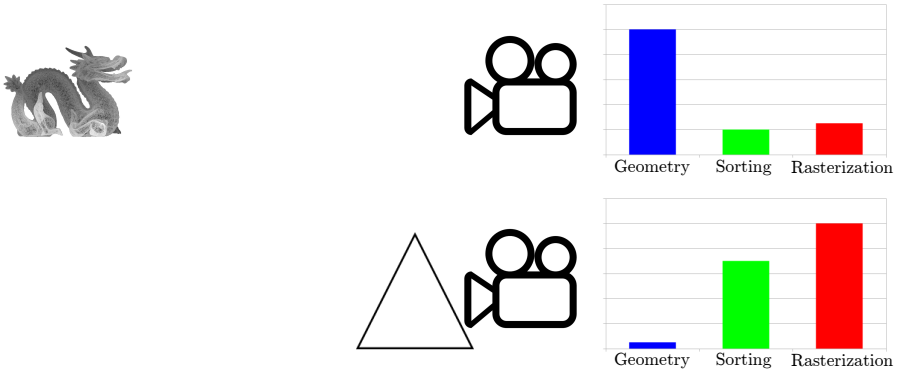


Figure 17: Examples of two different scenes and their influence on the workload of the geometry, sorting, and rasterization stages. Dragon model courtesy of Stanford University.

large number of primitives, the geometry stage has a high workload. Since the model is located far from the camera, it occupies a small portion of the image plane. Thus a few tiles will have primitives assigned to them and therefore the workload of the sorting stage is low. Consequently the rasterization stage also has a low workload since only a few tiles will have to be rasterized.

The second example consists of a single primitive being rendered. Naturally the geometry stage will have a low workload. However, because the primitive is placed so close to the camera, it will occupy the entire image plane. Therefore it has to be assigned to all tiles resulting in a high workload of the sorting stage. The rasterizer will also have a high workload because it needs to rasterize every pixel of every tile.

The execution time of the stages being weakly correlated has several consequences. Camera position and direction have a great influence on the workload, and the set of all camera positions and directions is virtually infinite in size, and only limited by the precision of the hardware. Also, there are a varying amount of models in different locations and orientations. Due to the infinite amount of ways to model a 3D world and place a camera in it, it is very difficult to predict the execution time and latency, and consequently to determine the WCET of such a system for all possible inputs with sufficient precision. The execution time distribution will be more similar to Figure 18, where we can see that the distribution has a high variance due to the unpredictability of the workload. Another consequence is that it is also difficult and impractical to create a model of the system for all possible inputs, as well as to analyze it. Therefore it is not possible to design a real-time system based on the real-time systems methodology, preventing us from giving temporal performance guarantees at design time.

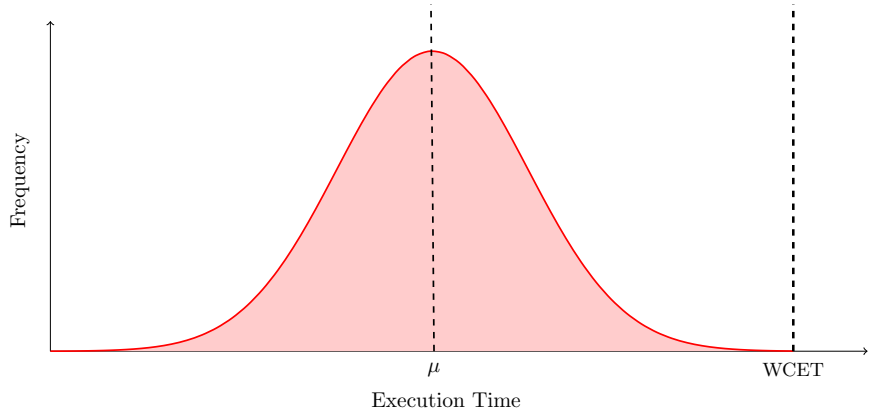


Figure 18: Distribution of execution times of a computer graphics application.

However, real-time systems are not the only type of systems that can give temporal performance guarantees. Another class of systems that can give the same are *reproducible* systems:

A system that for any specific input will have almost the same temporal behavior for each subsequent execution.

In other words, if we give the system a fixed input and measure the execution times of its parts, we will always get almost the same execution time. Tasks in a reproducible system have an execution time distribution similar to Figure 19, where the distribution is almost a single line. This implicitly means that the upper and lower bound of the execution times are almost equal, and that also the latency of a reproducible system is almost the same.

A reproducible system has several advantages:

1. An accurate model of the system is not required to give temporal performance guarantees.
2. Due to the upper and lower bounds being almost equal, it is not required to measure performance sufficiently many times to construct a distribution of execution times, latency, and throughput. What we measure is always very close to upper and lower bound.
3. The temporal performance is consistent. This is useful in systems where a fixed latency is required. An example would be airbags in cars, where having a variable latency between a crash and deployment of airbags could cause fatal injuries. Another example is performing remote surgery, where a fixed latency can help a surgeon anticipate the result of his/her next movement.

With this *reproducibility* property, we can then test inputs that we assume will have a high execution time. Since we know that there is

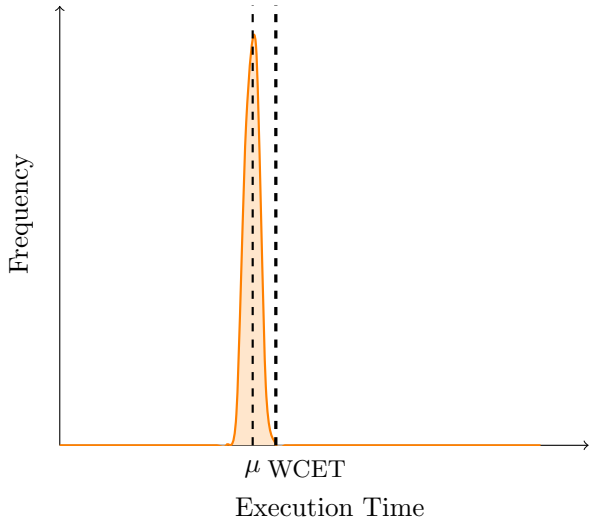


Figure 19: Distribution of execution times of a system with reproducible results.

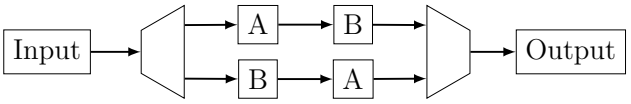


Figure 20: Example of a reproducible system. *A* and *B* are tasks with fixed execution times. Both paths from input to output have the same latency.

a very low variance in execution, we can guarantee that if the performance requirements are met, then they will also be met in future runs when the same input and hardware is used. Therefore we can still implement an architecture that gives temporal performance guarantees, under the condition that the set of inputs is known beforehand.

A perfect reproducible system is a system that has no non-deterministic parts, and the execution time distribution of its parts would be a single line. In practice, there are many systems that have sources of non-determinism like caches and shared resources such as DDR memory. A system can be classified as a reproducible system if it meets the following criterion: *For every specific input to the system, the total execution time for all paths that the input data can take through the system should be almost the same.* A trivial example of this is illustrated in Figure 20, where *A* and *B* are tasks with fixed execution times. No matter which path the input takes, the latency between the input and output will always be the same. If the tasks have similar execution time distributions like in Figure 19, then the latency of that system will not be fixed, but will have an upper and lower bound that almost the same.

In the case of graphics applications, building a reproducible system might not seem useful as we want interactivity and not a fixed 3D world and camera. However, due to the consistent temporal behavior, we can test inputs for which we assume that they will have

a high workload, and simply measure the execution time of the individual functional units of the system. We can then add up these numbers and acquire the latency of the system for that input. If we test the difficult cases, then we know that other inputs will have a lower workload and therefore a lower latency. This means that the latency for the set of inputs for a user application is not fixed.

What we want from our system is a fixed frame rate of N FPS. This gives an average frame time (total execution time to process a single frame) of $\frac{1}{N}$. In the system that we will design, the frame time and latency are equal, because we synchronize the system to the frame time. Therefore we do not allow processing the geometry for the next frame before the current frame is done processing. The latency not being fixed for the set of inputs is not a problem, as long as the maximum latency of difficult cases does not exceed the frame time. This means that a graphics system that is reproducible can give us a fixed frame rate with consistent frame times, and therefore the temporal guarantees that we want.

3

SYSTEM OVERVIEW

This chapter gives an overview of the *Starburst* MPSoC system on which the real-time hardware rasterization algorithm has been implemented, discusses several graphics architectures, and describes the proposed architecture.

3.1 THE STARBURST MPSOC

Computer Architecture for Embedded Systems

Starburst is a configurable, heterogeneous MPSoC developed by the CAES research group at the University of Twente. It is a tile based architecture, consisting of processor, gateway, and accelerator tiles connected to each other via a dual-ring interconnect. Starburst is used as a research platform for the development and analysis of multi-core, real-time, deterministic, streaming applications. Currently, it is implemented on the Xilinx ML-605 and VC-707 Field Programmable Gate Array (FPGA) platforms. An example Starburst architecture is given in Figure 21.

3.1.1 *Processor Tiles*

An overview of a processor tile is given in Figure 22. This tile consists of a RISC Microblaze processor, a timer used for interrupts, some local memory for a bootloader, scratchpad memory and FIFO memory for receiving and sending data from and to accelerator tiles or other processor tiles respectively. Software based C-FIFO communication [25] is used between all types of tiles. Each processor runs Helix OS, a real-time operating system developed for Starburst [39] that is POSIX compliant, and supports multi-threading using a real-time budget scheduler [44].

3.1.2 *Accelerator Tiles*

Accelerator tiles allow the use of dedicated logic for more efficient processing of streams such as CORDICs, FIR filters, etc. Some examples that use accelerators are a PAL decoder [15] where an accelerator is used for AM demodulation, and a Bluetooth Low Energy Long Range (BLR) receiver [46], where the RF front end, FIR filter, quadrature demodulator, and frame detector are implemented as accelerators. Figure 22 illustrates the general architecture of an accelerator.

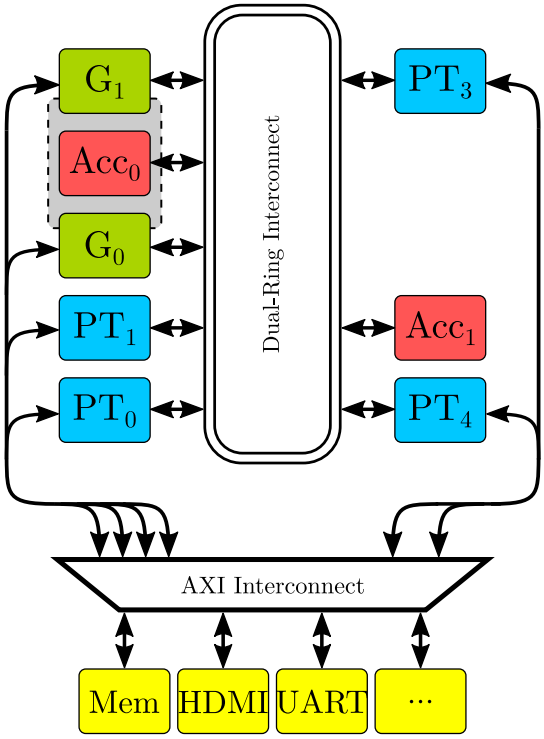


Figure 21: Example architecture overview with all tile types.

3.1.3 *Gateway Tiles*

To reduce hardware resource usage, accelerators can be shared between processor tiles. Gateway tiles are used to save and restore the state of shared accelerators as described in [14]. They consist of entry and exit gateway tiles that are placed before and after the accelerators that are shared respectively. Entry gateway tiles are the same as processor tiles with an added Direct Memory Access (DMA) peripheral on the AXI4-Lite bus, and exit gateway tiles are similar to accelerator tiles, where the accelerator itself is replaced with a DMA peripheral.

3.1.4 *Dual-Ring Interconnect*

Tiles are connected to a unidirectional, guaranteed-throughput dual-ring interconnect [15] that functions like a shift register. Each tile connects to the ring using a network interface that has a unique id, and each register on the ring is assigned a unique ring slot. Every clock cycle, the data and ring slots are shifted one hop. A tile can place data on the ring whenever a ring slot matches the id of the network interface, which is very similar to a Time-division Multiplexing (TDM) system. However, tiles can also send data even if the ring slot does not match their id as long as they do not overwrite data or interrupt the flow of data of other tiles. Therefore a much higher bandwidth can

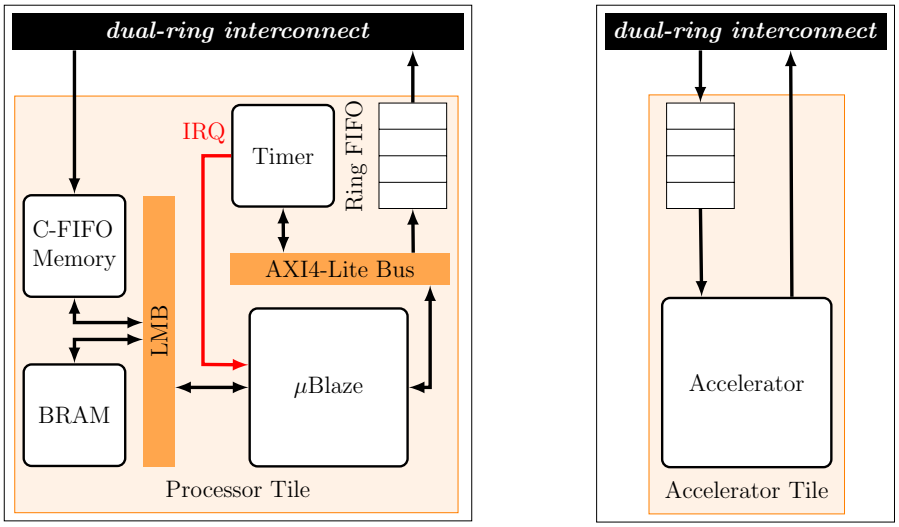


Figure 22: Overview of processor and accelerator tiles.

be achieved by clever placement of tiles on the ring, such as placing sending and receiving tiles right next to each other.

The current implementation is an improved version, where the configuration of accelerators is done over the ring itself, rather than using a separate configuration bus connected to gateway tiles. This enables the ability for any tile to configure any accelerator in the system.

3.2 FPGA PLATFORM

The Xilinx VC-707 FPGA platform is used for our implementation. It consists of a Virtex 7 FPGA and has many peripherals, including 1 GB of DDR3 RAM, Ethernet, HDMI output, GPIO, FMC connectors, etc. For video output, a resolution of 800x600 at 60 Hz was chosen.

3.3 REFERENCE ARCHITECTURE

The reference architecture consists of a single processor tile and no accelerator tiles. All the pipelines of the rasterization algorithm are fully implemented in software and run on the single processor. An implementation with multiple processor tiles that each run a single pipeline stage or a full, separate pipeline can also be realized but due to timing constraints, we have chosen to implement the most trivial setup that can execute the rasterization algorithm.

3.4 RASTERIZATION HARDWARE

This section gives an overview of the proposed architecture and accelerators, and describes how everything is integrated into the Starburst

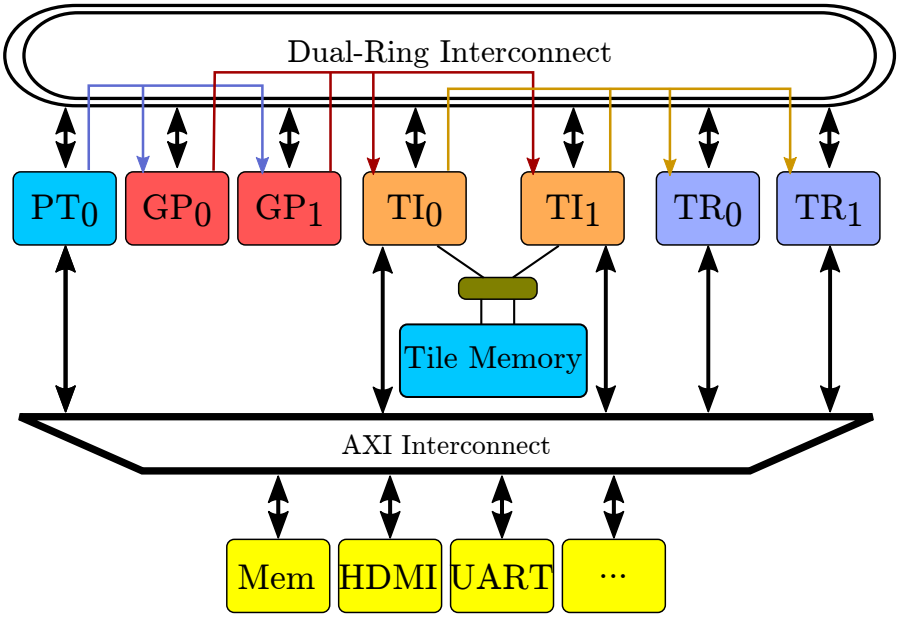


Figure 23: The proposed architecture.

plaform. Accelerators are placed on the ring in such a way that the connection between tiles has as much bandwidth as possible.

3.4.1 *Proposed Architecture*

The hardware accelerated rasterization architecture consists of a processor tile (PT), one or more geometry processors (GP), an equal amount of tile intersection accelerators (TI), and one or more tile rasterizers (TR). Implementations can be configured and scaled accordingly depending on the requirements such as frame rate, number of primitives, etc. Figure 23 shows a high-level overview of our architecture.

3.4.2 *Geometry Stage*

The geometry processor (GP) is the sole accelerator that handles the tasks of the geometry stage. A list of all vertices is streamed to geometry processors, which transform each vertex using the model, view, and projection matrices. Primitives that are back-facing or that fall outside the viewing frustum are discarded. Finally, it converts the final results from single-precision floating point values to fixed-point because the rest of the algorithm doesn't need floating point values. It is also cheaper to use fixed-point values from a hardware resource usage perspective. Parallelism in this stage can be achieved by simply placing more geometry processors back to back on the ring.

3.4.3 *Sorting Stage*

As parallelism is heavily exploitable in graphics architectures, we would ideally want to be able to scale every part of the graphics pipeline including the sorting step. With geometry processors we can simply add more to the ring, but this is not as trivial in the sorting stage. Sorting involves determining which primitives overlap with which tiles and storing this information in a list for each tile. Multiple accelerators need to access these lists meaning that the memory where they are stored is a shared resource. Therefore a safe and concurrent method of access is required to exploit parallelism in the sorting stage, including atomic read-modify-write transactions. To do this, the sorting stage consists of three components: *tile intersection accelerators*, a shared *tile memory* IP block, and a *binary tree interconnect* that connects the accelerators to the tile memory. These three components are shows in 23 and will be described in more detail below.

3.4.3.1 *Tile Intersection Accelerator*

Tile intersection accelerators (TI) perform the sorting step between the geometry and rasterizer pipeline stages of sort-middle architectures as described in Section 2.6.2. They receive processed primitives by geometry accelerators, determine which screen tiles intersect with each received primitive, and distribute primitives among *tile rasterizers*. There is a one to one mapping between geometry processors and tile intersection accelerators as it does not require an interconnect between them, simplifying the design. The *tile memory* IP block is used as temporary storage and is shared between all tile intersection accelerators.

3.4.3.2 *Tile Memory*

The tile memory IP block manages access to memory that stores three sets of data:

- *Primitive list* - Primitive data received from geometry accelerators.
- *Tile list* - A list per tile that consists of indices of primitives that intersect with the tile.
- *Tile counter* - A counter per tile that counts how many primitive indices are stored in the tile list.

Using indices conserves memory usage because the primitive data is not duplicated. When all sets of tiles have been determined, the list of indices is traversed linearly, and the primitive data associated with them is sent to the tile rasterizers.

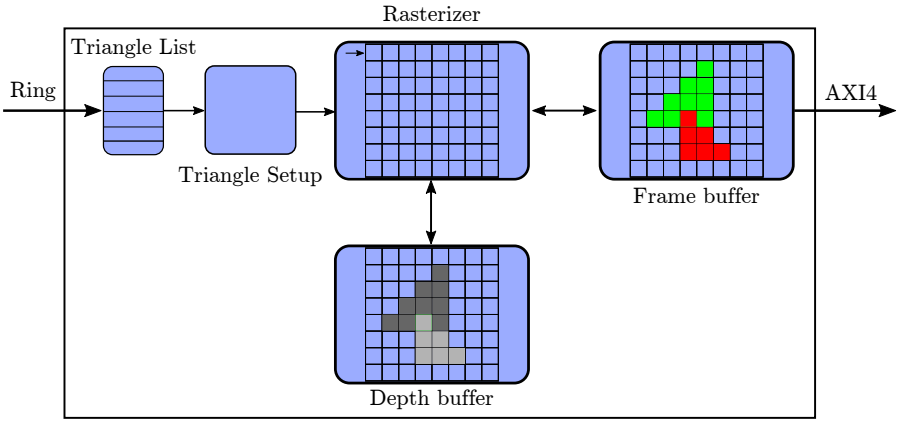


Figure 24: Simplified overview of the tile rasterizer.

3.4.3.3 *Binary Tree Interconnect*

In our architecture, a binary tree topology is used to connect tile intersection accelerators to the tile memory IP block. Internal nodes arbitrate read/write accesses using simple *valid/ready* handshaking similar to the AXI bus specification, and additional signals are used to ensure that transactions are atomic.

Using a binary tree for concurrent access has the benefit that data accesses are inherently serial due to the nature of the topology. Another benefit is that the path length of the leaves and the root grows logarithmically when adding more leaves, and aids in the scalability of the architecture.

3.4.4 *Rasterizer Stage*

The tile rasterizer (TR) is responsible for rasterizing a list of triangles for a given tile. This accelerator has three memories: triangle memory containing a list of primitive data, a local depth buffer, and a local frame buffer. Each accelerator is first configured by setting which tile of the frame buffer it will rasterize.

Rasterization begins by first clearing the frame and depth buffers, and performing triangle setup where the A_{01}, A_{12}, A_{20}, B_{01}, B_{12}, and B_{20} constants as well as the edge functions of Equation 6 are calculated. Then, each primitive is rasterized row by row, outputting the result to both the frame and depth buffers. When the rasterization of a tile is finished, the result is written to the global frame buffer residing in DDR RAM using the AXI4 bus. The tile position of the accelerator that is set during configuration is used to calculate the frame buffer address that corresponds with the tile. The HDMI peripheral IP uses a DMA to efficiently transfer the final image from DDR RAM to the onboard HDMI transmitter chip. Figure 24 shows the innards of this accelerator.

4

IMPLEMENTATION

In this chapter we describe the implementation of the reference system, the hardware accelerators, and the hardware accelerated system. We also explain why our architecture can be classified as a reproducible system.

4.1 REFERENCE SYSTEM

The reference system implements all the algorithms in software using one processor tile only. The application starts by allocating memory for the two frame buffers, the depth buffer, and tile buffer, followed by initializing the HDMI transmitter peripheral to enable double buffering using the allocated frame buffers. The output resolution is set to 800 by 600 pixels. Then, the view and projection matrices are initialized using the resolution and aspect ratio. To be able to manipulate the camera while the application is running, the GPIO peripheral that reads the five buttons on the boards is initialized. The last step consists of loading a 3D model consisting of just vertices and indices, and initializing the model matrix.

After initialization is done, the main loop starts. First, the state of the buttons is read which control the camera position and rotation. If a button was pressed, a new view matrix is calculated. Then the current frame buffer and depth buffer is cleared. Indices of the model are read and used to find the vertices that belong to that primitive, which are then used to perform the calculations of the geometry stage as described in Section 2.4.

After processing a primitive it is also immediately sorted by first calculating its bounding box and determining with which tiles the bounding box intersects. The index of the primitive is stored in the *tile list* section of the tile buffer, and the *tile counter* value that holds the number of primitives per list is increased by one.

When all primitives have passed the geometry and sorting stages, the rasterization stage loops over all tiles by reading the *tile counter* values. Then all primitives within tiles that have primitives in them are rasterized as described in Section 2.5. Finally, after looping over all tiles, a command is issued to the HDMI peripheral to *flip* the frame buffers so that the result of the rasterizer stage is made visible.

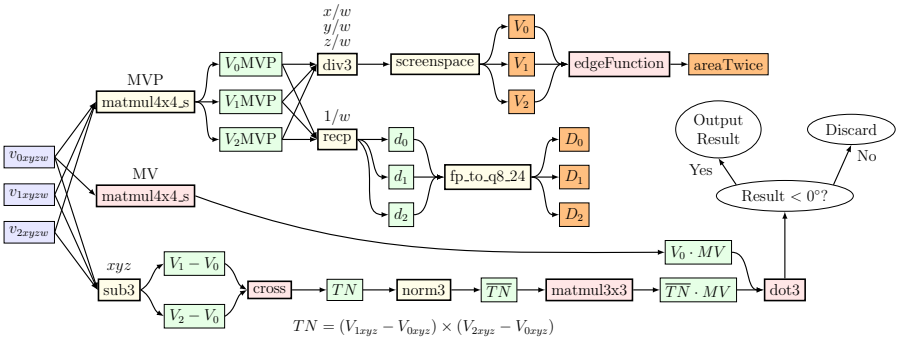


Figure 25: Internal overview of the geometry accelerator. Light blue nodes on the left are the three input vertices, green nodes are intermediate values, and orange nodes are the seven output words. Yellow and red nodes are IP blocks. Yellow blocks are shared resources and are used in multiple steps whereas red blocks are not.

4.2 ACCELERATORS

The detailed implementation of the accelerators are described below. All accelerators have configuration parameters that need to be set before they can be used.

4.2.1 *Geometry Accelerator*

The geometry accelerator performs the operations (with the exception of clipping) previously illustrated in Figure 9, transforming vertices of models defined in *model space* to *screen space*, which can then be rasterized by a primitive rasterizer. Figure 25 gives an overview of the individual components that the geometry accelerator comprises of.

Per primitive, three vertices consisting of four components each, are streamed to the geometry accelerator over the ring (light blue nodes in Figure 25). The input format is the single-precision, 32-bit floating-point format, as this is common in the computer graphics industry. To facilitate the usage of floating point values in logic, the Floating-Point Operator IP from Xilinx [49] is used. All IP blocks use simple *valid-ready* communication signals to synchronize data transfer between them.

The geometry accelerator has four configuration parameters that can be set: Model-View (MV) matrix, Model-View-Projection (MVP) matrix, and viewport width and height, all specified as floating point numbers. Both matrices are set to the identity matrix during reset, and the viewport width and height are 800 and 600 respectively, as this is the resolution used in our setup.

If a triangle is not discarded due to back-face culling, the geometry accelerator outputs the following seven, 32-bit words (orange nodes in Figure 25):

- The x- and y-coordinates of the three vertices in *screen space*. Each pair of coordinates is a concatenation of two 16 bit signed integers (3 words).
- $\frac{1}{2 \cdot A}$, 1 over twice the triangle area, using the Q8.24 fixed-point format (1 word).
- The *view space* depth of the three vertices, also using the Q8.24 fixed-point format (3 words).

4.2.1.1 *Vector-Matrix Multiplication*

First, the vertices are transformed from *model space* to *clip space* by multiplying the vertices with the MVP matrix. This is performed by the **matmul4x4_s** block, which accepts *streaming* data. Looking at Equation 8 that shows vector-matrix multiplication, we can see that the four rows of the result are simply a summation of partial products. For example, when only v_x is available, the partial products $v_x m_{11}$, $v_x m_{12}$, $v_x m_{13}$, and $v_x m_{14}$ can be calculated.

$$\begin{pmatrix} v_x & v_y & v_z & v_w \end{pmatrix} \cdot \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{pmatrix} = \begin{pmatrix} v_x \cdot m_{11} + v_y \cdot m_{21} + v_z \cdot m_{31} + v_w \cdot m_{41} \\ v_x \cdot m_{12} + v_y \cdot m_{22} + v_z \cdot m_{32} + v_w \cdot m_{42} \\ v_x \cdot m_{13} + v_y \cdot m_{23} + v_z \cdot m_{33} + v_w \cdot m_{43} \\ v_x \cdot m_{14} + v_y \cdot m_{24} + v_z \cdot m_{34} + v_w \cdot m_{44} \end{pmatrix} \tag{8}$$

The method of streaming in data and calculating partial results has several benefits. The first benefit is that less logic is required. The dual-ring interconnect can only transfer one word of data per clock cycle, therefore only one vertex component is available at the input. For each vertex component, one partial result for each of the four rows can be calculated, requiring four multipliers and adders. Calculating the full result in one go would require 16 multipliers and 12 adders.

The second benefit is the inherent pipelining, which enables running at higher clock frequencies. Figure 26 shows the how the partial results are calculated in a pipelined fashion. Every clock cycle, at most four multiplications and four additions are performed, and the final result is obtained in six clock cycles. Calculating the result when all vertex components are available would have to happen in at most two clock cycles, if the result should be obtained in six clock cycles. This would require a longer combinatorial path, lowering the potential clock frequency that this block could run at.

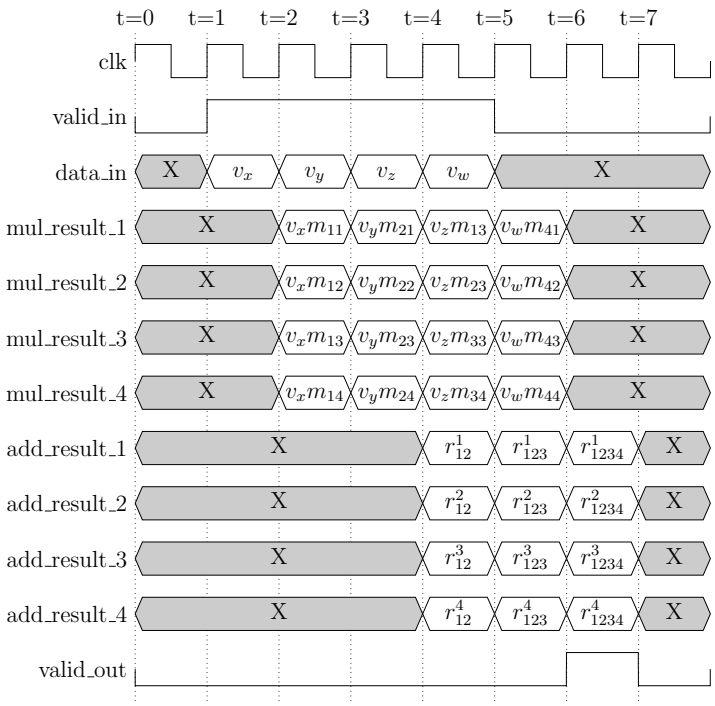


Figure 26: Pipelining of partial results in the **matmul4x4_s** IP. r^n_m is the cumulative result of the m partial products of row n.

The third benefit is that streaming requires less memory to store intermediate results. The four components of a vertex do not have to be stored before processing them, but also requires less memory for storing intermediate results. In the **matmul4x4_s** block, only four words of memory is required to store the results of the adders, and consequently, the final result.

After multiplying the three vertices with the MVP matrix, the results $V_n \cdot MVP$ are fed to the **div3** and **recp** blocks. The w component of the results contain the *view space* depth of the vertices of which we want the reciprocal. That is calculated by the **recp** block and the results are processed by the **fp_to_q8_24** block which converts the floating point input to the Q8.24 fixed-point format.

Additionally, the perspective divide is also performed for each vertex $V_n \cdot MVP$ using the **div3** block, which divides the x, y, and z components by the w component. After this division, the vertices are now in NDC space. The **screenspace** block transforms the vertices to *screen space* by scaling and translating the x and y components to match the viewport resolution that is stored in the configuration memory of the accelerator. It also drops the z and w components, and converts the x and y components into 16 bit signed integer formats.

The last calculation involves obtaining the reciprocal of twice the triangle area. First, twice the area of the triangle is calcuated by the **edge_function** block, giving us the result in 32 bit signed integer format. Next, the **int_to_fp** block converts the result into the single-

precision format, which is used by the **recp** block to obtain the reciprocal. Finally, the **fp_to_q8_24** block converts the result into the Q8.24 fixed-point format. The **int_to_fp** block is not shown in Figure 25 for brevity.

4.2.1.2 *Back-face Culling*

Back-face culling is performed by checking if the normal of the triangle is pointing towards the camera. This can be done in three steps (illustrated in the bottom path of Figure 25):

- Define a (normalized) vector with its base at the camera origin, pointing to one of the vertices.
- Calculate the (normalized) normal vector of the triangle.
- Check if the angle between the two vectors lies between 90° and 270° i.e. if the dot product is negative.

Performing these steps is the easiest in *view space*, where the first step is to simply multiply v_0 with the MV matrix to obtain $v_0 \cdot MV$. This multiplication is also performed by another instantiation of **matmul4x4_s** and is performed in parallel with the calculation of $v_0 \cdot MVP$ described above.

Calculating the normal vector of the triangle involves determining two edges $e_{10} = v_1 - v_0$ and $e_{20} = v_2 - v_0$, then performing a cross product to obtain the normal vector in *model space* $N_{model} = e_{10} \times e_{20}$. Finally, N_{model} is multiplied with the MV matrix using the **matmul3x3** IP, resulting in the normal vector in *view space* $N_{view} = N_{model} \cdot MV$.

matmul3x3 performs a vector-matrix multiplication by dropping v_w, implicitly setting it to 0. In homogeneous coordinates, a value of 1 or 0 for the w component represents point in space or a direction vector respectively. Since the normal vector is a direction, the fourth row and column of the MV matrix would be multiplied by zero, and therefore **matmul3x3** only uses the upper left 3x3 matrix of the MV matrix. This block calculates partial results just like **matmul4x4_s** except it is non-streaming, and stores N_{obj} temporarily.

The last step is to check if N_{view} points towards the camera. This is done by using the **dot3** block to calculate the dot product using on both vectors and checking if the result is negative. If it is, then the triangle is front-facing and the accelerator continues with the other operations it needs to perform. When the dot product is positive, the accelerator drops the triangle and starts processing a new triangle. A simple optimization to check if the result is negative is to check if bit 31 of the result is set, as negative numbers in both two's complement integers and floating point numbers set bit 31.

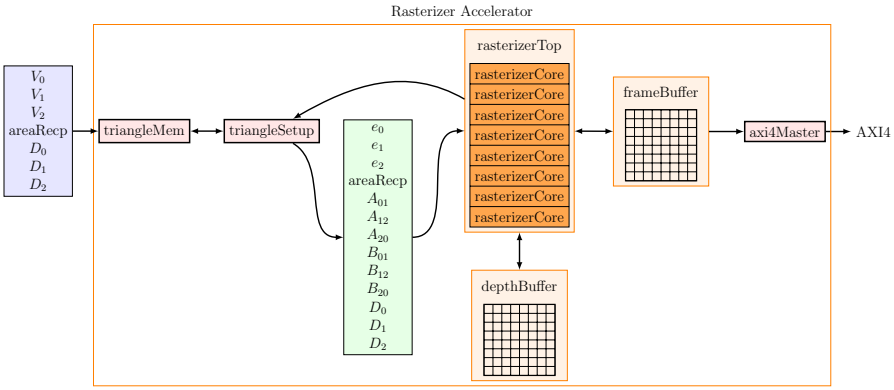


Figure 27: Internal overview of the rasterizer accelerator.

Optionally, both vertices can be normalized before performing the dot product. For simply performing back-face culling this is not necessary, but if lighting calculations need to be performed, then normalizing the normal vector is required. The geometry accelerator does normalize both vectors using the **norm3** block.

4.2.2 *Tile Rasterizer Accelerator*

The tile rasterizer accelerator stores received primitive data for up to a fixed amount of triangles (16 in our implementation), performs a precalculation on the triangle data and rasterizes them, and finally writes the result to the frame buffer residing in off-chip DDR3 memory. An overview of the accelerator is given in Figure 27. Aside from functional blocks there are also two memory blocks used as local frame and depth buffers.

The **triangleMem** block stores the received triangle data in block RAM. Whenever the maximum amount of triangles are received, or when a predefined bit pattern is received to indicate that the last triangle is being received, then the accelerator enables the rest of the IP blocks. The **rasterizerTop** block then requests the first triangle so that it can rasterize it. But in order to do this, some intermediate data needs to be calculated such as the A and B constants of Equation 5, as well as the edge functions E_{01}, E_{12}, E_{20} of Equation 6. This is calculated in the **triangleSetup** block that sits between the **triangleMem** and **rasterizerTop** blocks.

The **rasterizerTop** block drives eight **rasterizerCore** blocks that are each responsible for rasterizing a single column of pixels. Each core starts by first clearing the frame and depth buffers before using them. The state machine of each core is setup such that it takes the same amount of clock cycles whether it has to rasterize or not in order to simplify the synchronization between all eight cores and the **rasterizerTop** block.

When all primitives have been processed, the **rasterizerTop** block signals the **axi4Master** block to start writing the local frame buffer to the frame buffer stored in off-chip DRAM. Each pixel is represented as 32 bits, and they are written per row, meaning that the width of the data channel of the AXI4 port is 256 bits wide. We take advantage of the capabilities of the AXI interconnect peripheral provided by Xilinx to split the data into smaller chunks that the DDR memory controller can process.

The address of tile location in the frame buffer is calculated using several configuration words in the accelerator. They consist of base address of the frame buffer, number of tiles in the x and y direction, the size of each tile, and the current tile coordinate. After writing the local frame buffer to DDR memory, the cores are reset and they clear the local frame and depth buffers again.

4.3 HARDWARE ACCELERATED SYSTEM

The hardware accelerated system is shown in Figure 28. It consists of two processor tiles (PT0 and PT1), a geometry accelerator (GM) and a tile rasterizer accelerator (TR). Although the critical parts of the tile intersection accelerator were implemented, it was not in a state that allowed us to integrate it into the system due to a lack of time. Therefore the second processor tile uses the same sorting implementation as in the reference system.

In a Starburst system, each processor tile can write to any location in the C-FIFO memory of other processor tiles. However, accelerators simply receive data from the ring and send data to the ring without using any addressing. The network interface in accelerator tiles need to be configured so that the network interfaces know where the data comes from and where it needs to go. Processor tiles that receive data from accelerators do so through a network interface with a FIFO connected to the FSL bus of the Microblaze processor.

When the application starts, PT0 performs the same initialization as in the reference system. Additionally, it defines how data should flow (PT0, GM, PT1, TR) in the system by configuring the network interfaces of both accelerators. PT0 also starts a thread on PT1 that executes a function that sorts incoming primitive data from GM, and utilizes TR to rasterize the sorted primitives.

When data from GM arrives at PT1, the processor reads the data through the FSL link which is a blocking process. While the maximum amount of primitives that is sent to GM is known, PT1 might not read that many words as some primitives could be culled. This is circumvented by writing a special data word before sending the last primitive data to GM, which then outputs the same special data word to indicate that the last primitive was processed. When PT1 reads this, it can then starts to configure and use TR. The same method is also

used in TR. After all tiles have been processed, PT1 swaps the front and back frame buffers so that the HDMI peripheral displays the final rasterized result. Then it signals PT0 to start another iteration of its main loop by reading the button state, updating the view matrix and sending primitive data to GM again.

4.4 REPRODUCIBLE ARCHITECTURE

As a refresher, we repeat the criterion for a system to be classified as a reproducible system:

For every specific input to the system, the total execution time for all paths that the input data can take through the system should be almost the same.

The accelerators that we have implemented have fixed hardware that have the same execution time regardless of what the input is. Also, almost all communication goes through the dual-ring interconnect that has a guaranteed minimum throughput and maximum latency. In the topology that we use, all tiles on the ring can place a word of data on the ring every clock cycle, and no data streams interrupt each other. The only shared resource in the system is DDR memory, which is written to by the tile rasterizer accelerator, and read from by the HDMI transmitter peripheral. This is one of two non-deterministic parts of the system, with processors with caches being the second. Compared to the workload of the rasterization algorithm, we predict that the influence on execution times of the rest of the system will not be significant enough that it would violate our requirements. Therefore we are confident that our architecture is a system that has the reproducibility property.

5 RESULTS & EVALUATION

In this chapter, we begin the evaluation of our system by discussing the results concerning the execution time of the entire system and individual components. Due to the accelerators in the hardware accelerated system not working properly, we do not have results for the hardware accelerated system as a whole. However, we do have performance figures for the separate accelerators, which are used to extrapolate them and give a theoretical evaluation of the hardware accelerated system. Finally, we give an overview of the hardware and area costs and discuss the results.

5.1 PERFORMANCE EVALUATION

In this section, the performance of both the reference and the hardware implementations will be discussed. The Microblaze processor in the processor tiles in both the reference and hardware accelerated implementations use the following configuration:

- Version: 8.50c
- Instruction cache: 64 KiB
- Instruction cache line size: 8 words
- Data cache: 64 KiB
- Data cache line size: 8 words

All processor and accelerator tiles, the dual-ring interconnect, and all AXI buses run at 100 MHz, and the DDR3 memory controller runs at 400 MHz. The output resolution is 800x600 32 bits per pixel. Both implementations are tested with two 3D models:

- A flat square consisting of 4 vertices and 6 indices. It is primarily used for testing the throughput of the rasterizer stage as it has a low geometry processing workload.
- A model of a monkey named Suzanne used in Blender, an open source 3D graphics and animation application. It consists of 507 vertices, 2904 indices, and 968 faces, and is used as a real-world example.

With these two models, maxmimum execution times were measured for each of the three main parts of the graphics pipeline: geometry, sorting, and rasterization. The models were rendered at several

Number of pixels drawn	Percentage of frame buffer	Clearing (uS)	Geometry (uS/primitive)	Sorting (uS)	Rasterization (#cycles/pixel)	Total (min/max uS)
13225 (115x115)	2.76%	33749	113 (56.5)	557	31307 (237)	65550 / 65702
18225 (135x135)	3.80%			680	40954 (225)	75323 / 75457
25921 (161x161)	5.40%			951	59970 (231)	91667 / 94902
40401 (201x201)	8.42%			1391	86562 (214)	121674 / 121825
71289 (267x267)	14.85%			2366	150040 (210)	186066 / 186229
160801 (401x401)	33.50%			5104	331075 (206)	369809 / 370030
480000 (800x600)	100%			14759	976361 (203)	1018871 / 1019024

Table 1: Maximum execution times of each stage when rendering a simple plane model consisting of two triangles.

Number of pixels drawn	Percentage of frame buffer	Clearing (uS)	Geometry (uS/primitive)	Sorting (uS)	Rasterization (#cycles/pixel)	Total (min/max uS)
49163	10.24%	33774	32816 (33.9)	10048	160306 (326)	227234 / 236870
70763	14.74%			11478	212137 (300)	289060 / 290078
110223	22.96%			13771	311404 (283)	390560 / 391379
194327	40.48%			18618	527190 (271)	610682 / 611540
360840	75.18%			346318	2177047 (603)	2587084 / 2587828

Table 2: Maximum execution times of each stage when rendering the monkey model consisting of two triangles.

depths starting from far away up until the entire screen was covered. The results were obtained by rendering 1000 frames and measuring the maximum execution time.

5.1.1 *Reference System*

Tables 1 and 2 show the maximum execution times of the reference system for the plane and monkey models respectively. Looking at the results of the plane model, we can clearly see that the rasterization parts has the highest execution time by far. Sorting primitives takes longer when more tiles have primitives assigned to them as expected. Processing geometry also depends on the amount of primitives, ranging from 113 µs to 32 816 µs for the plane and monkey model respectively. The reason why the number of microseconds per primitive is lower when rendering the monkey model is because a set of primitives are culled, while that is not the case for the plane model.

While we do see a low variance in the total execution time of the system, giving real-time guarantees is difficult because of several factors. The memory controller is a black box, so cannot create a model of it to verify its real-time properties. The same applies to the Microblaze processors as well, which hinders giving proofs about their behavior. In this particular instance the processors process data with consistent execution times most likely because the caches are large enough.

However once the implementation is expanded to run multiple threads, then the processors can show unpredictable performance results as we witnessed while testing. Helix OS, which has a real-time budget scheduler to support multi-threading and runs on each processor, is configured to display various statistics every minute by default. These statistics are sent to the UART peripheral which has a FIFO which can hold up to 16 characters. This is not large enough to hold all the characters of the statistics and therefore the processor has to wait until the FIFO buffer is emptied. This of course is very slow compared to the clock frequency of the processor since the UART outputs at 115200 baud, and the processor runs at 100 MHz. During displaying the statistics, the execution time of the application increases by about 10 milliseconds, but only affects frame buffer clears and rasterization times. This is as expected as the second thread does not yield and uses its full budget when output when outputting statistics, degrading the performance of the rendering application.

5.1.2 *Hardware Accelerated System*

The hardware accelerated system consists of two processor tiles, one geometry processor, and one tile rasterizer as illustrated in Figure 28. Each accelerator has several counters to measure the amount of clock

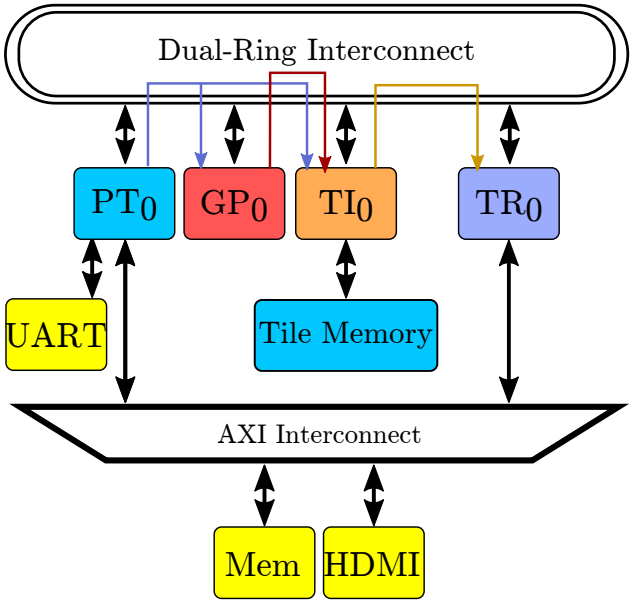


Figure 28: High-level overview of the implemented hardware accelerated system.

cycles it takes to receive data over the ring and to process that data. The rasterizer accelerator has an additional counter that counts the clock cycles it takes to write the local frame buffer to DDR memory. The counters store their values in configuration memory, which can then be accessed over the ring.

Unfortunately, because the accelerators did not work in the fully implemented hardware accelerated system, we could not measure its performance. This is due to the Xilinx tools not dealing well with the two-process method we used for all hardware that we built. However, we do have performance results of the accelerators when they were tested individually. During the development process, each accelerator was tested and evaluated separately in order to verify if they were working correctly.

While the geometry accelerator occasionally gave partially incorrect results, the rasterizer accelerator worked correctly. It is when both of them were used to build the hardware accelerated system that they stopped working. Synthesis results show that each accelerator has one bit of a state register that is driven by a combinatorial signal. This explains why the geometry accelerator would output incorrect results in a non-deterministic fashion.

Because we have performance figures of the isolated accelerators, and all tiles communicate over the dual-ring interconnect which is formally proven to have hard real-time behavior, we can still reason about the characteristics of the hardware accelerated system.

The process of sending data from a processor tile to an accelerator tile is common to both accelerators and are therefore described in this section. Writing to an accelerator happens through the AXI4-Lite bus

on the Microblaze processor. It is connected to an AXI4 interconnect peripheral in shared access mode and no registering is used. Since the processor is the only master on the bus, the interconnect does not add any latency to the processing of placing data on the dual-ring interconnect. The network interface IP with built-in FIFO does add a fixed latency and we have measured a minimum throughput of one word per 12 clock cycles but is usually between 9 and 10 clock cycles.

5.1.2.1 *Geometry Processor*

The geometry accelerator receives 12 words of data per primitive, which means that it takes at most $12 \cdot 12 = 144$ clock cycles for the processor to send it to the accelerator. Using a DMA would enable the processor to send a word each clock cycle resulting in 12 clock cycles per primitive. It calculates whether the current primitive should be culled while at the same time, it also transforms the primitive from model space to screen space. If the primitive is not culled, then it outputs seven words of data, one word per clock cycle.

Determining whether a primitive is back-facing and thus to be culled happens in 9 clock cycles, but the accelerator fully processes the primitive before checking if it should be culled or not. This was done to simplify the design, but could be modified to immediately stop processing after 9 clock cycles. Fully processing and outputting a primitive takes 33 clock cycles. This results in:

$$\frac{144 + 33}{100 \times 10^6} \cdot 1 \times 10^6 = 1.77\,\mu s \tag{9}$$

without using a DMA. This gives a maximum throughput of approximately 564971 primitives per second. The plane model consists of two primitives and would take $2 \cdot 1.77\,\mu s = 3.54\,\mu s$ to compute, and the monkey model consisting of 968 primitives would take $968 \cdot 1.77\,\mu s = 1713.36\,\mu s$. Compared to the reference system, this results in a speedup of approximately 31 and 19 times for the plane and monkey model respectively.

If a DMA would be used, then it would take:

$$\frac{12 + 33}{100 \times 10^6} \cdot 1 \times 10^6 = 0.47\,\mu s \tag{10}$$

to process a single primitive. This gives a speedup of approximately 120 and 72 times compared to the reference system when processing the plane and monkey model respectively.

5.1.2.2 *Tile Rasterizer*

The tile rasterization accelerator receives seven words of data per primitive with a maximum of 16 primitives. That gives a total of 112

words that the accelerator can accept for each tile. If less primitives fall within a tile, then a special data word is sent to let the accelerator know that no more primitives will be sent and that it can start rasterization. The minimum throughput of a processor placing data on the ring is one word per 12 clock cycles, which results in the worst case of 1344 clock cycles. If a DMA would be used, then the full bandwidth of the dual-ring interconnect could be used and it would take at most 112 clock cycles to send the data, which is more than an order of magnitude faster.

Since the rasterization process happens in fixed hardware, the amount of clock cycles it takes is always the same. For each primitive, a primitive setup has to be performed taking 6 clock cycles, and 8 rows of 8 pixels need to be rasterized, each row taking 3 clock cycles. This brings the total to 30 clock cycles per primitive. For the first primitive per tile, the local frame and depth buffers need to be cleared as well which takes 9 clock cycles. The primitive setup is done in parallel however, so the total is 33 clock cycles for the first primitive. The maximum amount of cycles needed for 16 primitives is therefore $33 + (15 \cdot 30) = 483$.

Calculating how much clock cycles are needed to write the local frame buffer contents to DDR memory is difficult since we cannot create an accurate model of the memory controller due to its closed nature. We therefore resort to measuring the maximum amount of clock cycles on the actual hardware. It ranges from 150 to 350 clock cycles, and therefore we take the maximum value as the WCET.

In the case of the plane model, we know that there can be at most two primitives per tile, and when it covers the entire screen, we know that all 7500 tiles have two primitives in them. Two primitives need 15 words (14 for the two primitives, and one to signal the accelerator that it is the last primitive), which at most can take $15 * 12 = 180$ clock cycles to send to the accelerator. Rasterizing the two primitives takes $33 + (1 \cdot 30) = 63$ clock cycles, and writing the result to DDR memory takes 350, giving a total of $180 + 63 + 350 = 593$ clock cycles. Compared to lowest number of cycles per pixel in Table 1, processing 64 pixels would take approximately $64 \cdot 203 = 12992$ clock cycles on the processor which makes the accelerator have a $\frac{12992}{593} \approx 21.9$ times lower execution time.

For the plane model, a frame takes just under one second to render on a processor while on the accelerator it takes:

$$\frac{7500 \cdot (180 + 63 + 350)}{100 \times 10^6} \cdot 1000 \approx 44\,\text{ms} \approx 22\text{FPS} \tag{11}$$

This is without using a DMA. If a DMA would be used, then it would take:

$$\frac{7500 \cdot (15 + 63 + 350)}{100 \times 10^6} \cdot 1000 \approx 32\,\text{ms} \approx 31\text{FPS} \tag{12}$$

Using the same approach, performance figures for the absolute worst case (every tile has 16 primitives) can be calculated:

$$\frac{7500 \cdot (1344 + 483 + 350)}{100 \times 10^6} \cdot 1000 \approx 163\,\text{ms} \approx 6\text{FPS} \tag{13}$$

Again, this is without the use of a DMA. Using a DMA results in:

$$\frac{7500 \cdot (112 + 483 + 350)}{100 \times 10^6} \cdot 1000 \approx 71\,\text{ms} \approx 14\text{FPS} \tag{14}$$

These results are summarized in Table 3. We can see that not only is the accelerator faster, but as long as it can be supplied with data fast enough, it is guaranteed that the performance will not fall below the calculated values due to the use of fixed hardware.

Using the rasterizer accelerator has another benefit in that it alleviates the need for clearing the screen before the rendering process. If we simply send a special word to the accelerator to indicate that there are no primitives to render, it will write its cleared local frame buffer to DDR memory. Clearing the local frame buffer takes 9 clock cycles, receiving the special word takes 12 clock cycles, and writing it to DDR memory takes at most 350 clock cycles, meaning that it takes at most:

$$\frac{7500 \cdot (12 + 9 + 350}{100 \times 10^6} \cdot 1000 = 27.15\,\text{ms} \tag{15}$$

to clear the frame buffer if nothing is to be rasterized. Compared to the roughly 33.7 milliseconds it takes for the reference system clearing the frame buffer, we can see that it is a bit faster. The main advantage is that clearing the screen in the reference system is mandatory in order to clear the result of the previous frame, whereas in the hardware accelerated system it is part of the rasterization process. This means that we save at least $33.7 - 27.15 = 6.55\,\text{ms}$ when nothing is rendered, and more when primitives are rendered.

5.2 HARDWARE COSTS

As explained before, all implementations are done on the Xilinx VC707 evaluation board, that contains a Virtex-7 VX485T FPGA. Using Xilinx terminology, the relevant types of hardware resources that this FPGA offers are:

- *Slice register*: a single bit flip-flop.

#primitives	Receive from ring	Rasterization	Write to DDR memory	Frame time	FPS
2	180	63	350	44	22
2	15 (DMA)	63	350	32	31
16	1344	483	350	163	6
16	112 (DMA)	483	350	71	14

Table 3: Performance figures for the rasterizer accelerator. Receiving data from the ring, rasterization, and writing data to DDR memory are in number of clocks. Frame times are based on the accelerator running at 100 MHz, and are given in milliseconds.

	Available	Reference System (Used/%)	Hardware Accelerated System (Used/%)
Slice Registers	607200	12912 / 2.13%	27210 / 4.48%
LUTs	303600	15305 / 5.04%	50550 / 16.65%
LUTRAM	130800	1421 / 1.09%	2120 / 1.62%
BRAM	1030	13 / 1.26%	41 / 3.98%
DSP48E1	2800	5 / 0.18%	362 / 12.93%

Table 4: Hardware resource usage of the reference system (software-only) and hardware accelerated system (one geometry and rasterizer unit).

- *Lookup Table (LUT)*: a multi-functional building block that can be used to implement logic, RAM, ROM, shift registers, etc.
- *LUTRAM*: a LUT that is used as RAM.
- *Block RAM (BRAM)*: on-chip static RAM with a size of 36 kbit per BRAM block. Can also be used as two separate 18 kbit RAMs.
- *DSP48E1*: an on-chip, dedicated Digital Signal Processor (DSP).

Table 4 shows the hardware resource usage of the reference system and the hardware accelerated system. The reference system does not use any accelerators and consists of one processor tile as described in Section 3.1.1, whereas the hardware accelerated system uses two processor tiles, and one geometry and rasterizer accelerator. We should note that the amount of LUTs used include LUTRAM, so to find the amount of LUTs used as logic, subtract the amount of LUTRAM from the amount of LUTs.

Table 5 gives the breakdown of the hardware resource usage of the accelerators. The rasterizer accelerator internally has multiple rasterizer cores, and the resource usage of an individual core is also shown in the same table. The first thing to notice is that the resource usage of the accelerators are higher than that of a single processor tile.

	Available	Geometry Accelerator (Used/%)	Rasterizer Accelerator (Used/%)	Rasterizer Core (Used/%)
Slice Registers	607200	4913 / 0.81%	3921 / 0.65%	327 / 0.05%
LUTs	303600	21604 / 7.12%	6346 / 2.09%	387 / 0.13%
LUTRAM	130800	0 / 0%	0 / 0%	0 / 0%
BRAM	1030	0 / 0%	19 / 1.84%	0 / 0%
DSP48E1	2800	105 / 3.75%	247 / 8.82%	30 / 1.07%

Table 5: Hardware resource usage of the used accelerators.

There are two reasons for this. The first is that a processor executes its instructions in a serial manner, whereas the implemented accelerators process data in a parallel fashion by having multiple data paths. The second reason is the consequence of a trade-off that was made, favoring higher performance at the cost of increased resource usage.

5.2.1 *Geometry Accelerator*

The geometry accelerator performs several mathematical operations using floating point arithmetic such matrix-vector multiplication, dot product, normalization, division, addition, subtraction, finding the reciprocal, and cross product. To map the vertices to screen space, it also converts the floating point results to fixed point formats. The amount of LUTs of the geometry accelerator is mostly used for screen space mapping, followed by matrix-vector multiplications, where the non-streaming one (Chapter 4.2.1.1) uses the most. The usage of DSPs is similarly distributed among the internal hardware blocks. Slice register usage is mostly for storing state and intermediate results.

Resource usage could be lowered by sharing some of the hardware used for arithmetic. For example, there are three instances of the *div3* IP block that simultaneously perform the homogeneous division operation. This could lowered to one instance and performing the division in serial.

5.2.2 *Rasterizer Accelerator*

Even though we have chosen to use 8 rasterizer cores in the rasterizer accelerator, the hardware resource usage is around three times less than that of the geometry accelerator. This is because only integer arithmetic is used so no expensive floating point operator hardware is needed. Each rasterizer core implements the equations of Section 2.5.1 and requires a fair bit of arithmetic operations, explaining the usage of 30 DSP blocks. The *triangle setup* IP performs a few operations

that all rasterizer cores have in common, therefore using just 7 DSP blocks.

Since tiles in our implementation are squares, and 8 cores are used, each accelerator processes blocks of 8 by 8 pixels, requiring storage for 64 color values and 64 depth values. Each data value is four bytes in size so a total of 512 bytes worth of storage is needed. Storage is also used for storing triangle data produced by the geometry accelerator. Each triangle requires 7 words of data and in the implementation, up to 16 triangles can be stored, requiring a total of 448 bytes.

The 19 BRAMs that are used can store up to 85.5 kB of data, of which only 960 bytes are actually used. The frame and depth buffers use 8 and 4 BRAMs respectively, and the triangle memory uses 7. To explain the reason why so many BRAMs are used, we need to look at the BRAM primitive. Each BRAM primitive can have two 36 bit ports (32 bits data and 4 bits parity) when used as a true dual-port memory, or one 72 bit port (64 bits data and 8 bits parity) when used as a single port memory. Integer multiples of 32 bit data widths were used in our implementation, so for single and dual-port memories, every 64 and 32 bits of data width will use one BRAM respectively.

The frame and depth buffers are 256 bits wide because of the 8 cores, and each core reading and writing 32 bit words. Even though the frame and depth buffers have the same data width and memory size, the discrepancy of the frame buffer using twice as many BRAMs comes from it being implemented as a dual-ported memory. The second port is connected to the AXI port of the accelerator. But since the frame buffer is never accessed by the cores and the AXI port simultaneously, a multiplexer could be used to decrease the BRAM usage.

For triangle memory, 7 words of data are stored per triangle, resulting in a 224 bit data width, and therefore 7 BRAMs are used. By using different settings, the synthesizer might come up with a more efficient implementation that could result in much less BRAM usage by implementing it using LUTRAM instead. With the current implementation on the other hand, many more triangles per rasterizer accelerator can be used without using more hardware.

6 CONCLUSION

The goal of this thesis was to design and implement a real-time triangle rasterization algorithm, and for this reason, three objectives were defined. The first was to research and evaluate various graphics architectures, and determine which architecture is most suited to reason about its real-time characteristics. Four major graphics architectures were evaluated and the *sort-middle* architecture was chosen because the stages are loosely connected, no duplicate calculations are performed, and the architecture is less complex due to not having extra interconnects for sending data to other pipelines. This makes reasoning about each separate stage easier than the other architectures. However, the complexity is concentrated in the sorting stage, as it requires atomic read-modify-write capabilities accessing tile memory when using multiple functional units in the sorting stage. Several triangle traversal methods were evaluated and the tiled traversal method was chosen because it has low complexity and exploits the parallel nature of rasterization. This gives the flexibility to use more hardware resources for higher performance without adding additional complexity.

The second objective was to implement a reference software implementation of a tiled triangle rasterizer on the Starburst platform. Having chosen to use the sort-middle architecture, we created a software implementation of it, running the rasterization algorithm on a single processor. This was a crucial objective because it gives us insight into real-life performance and bottlenecks of the sort-middle architecture. The performance of the reference implementation was evaluated, and based on the results, the geometry and rasterization stages that are the largest bottlenecks were selected to be replaced by accelerators.

The third objective was to design and implement an embedded system that performs real-time triangle rasterization on the Starburst platform. Based on the findings of the reference implementation, two hardware accelerators were implemented and an embedded system was realized. Unfortunately, we could not evaluate the final system because the system would hang. Synthesis logs indicated that this is due to a bug where a bit of a state register is driven by a combinatorial signal. This causes illegal state transitions in the state machines of the accelerators which result in deadlocks. Thankfully we have results from when the accelerators were tested separately during development, and based on those results, a theoretical evaluation of the hardware accelerated system was performed.

With the results of this evaluation, we can now answer our research question:

Is the Starburst architecture suitable for implementing a real-time rasterization algorithm? If not, which hardware and/or software components need to be incorporated to make it suitable?

We have shown that it is very difficult and impractical to design a real-time graphics architecture using the real-time systems methodology at design time. Due to the weak correlation of the workload between the stages of the graphics pipeline, we cannot predict the workload for all possible inputs. We provided a new classification of systems that also give temporal performance guarantees called *reproducible systems*. If we have a limited set of inputs that are known beforehand, we can implement a graphics architecture with *reproducible* results. Strong guarantees can be given because of the use of fixed hardware and a real-time ring interconnect. We can calculate exactly how long it takes to transform a single primitive in the geometry stage and how long it takes to rasterize a single tile. In Tables 1 and 2, we can see that in the reference implementation, the cost for rasterizing a single pixel changes based on the percentage of the screen that is being covered by primitives. In the case of the monkey model, the execution time per pixel increases significantly when the camera zooms so much that the model covers most of the screen. These kinds of outliers cannot occur when using accelerators, as they will always provide very similar execution times in subsequent runs. With this, we can state that the Starburst platform is suitable for implementing a rasterization algorithm with temporal performance guarantees.

There is one important improvement that can be made however. While we do have gateway entry tiles that are like processor tiles, but with an additional DMA engine, there are no easy to use functions to use the DMA in Helix OS. Also, these tiles are not yet an integral part of the system generation scripts of Starburst, and requires additional effort to get them up and running. Since we stated that developing a real-time graphics architecture with reproducible results depends on being able to generate and evaluate a system quickly, easy usage of a DMA engine would help this process. We should be able to easily add a DMA engine to any processor tile and use simple functions provided by Helix OS to use it.

6.1 FUTURE WORK

As a first step, we would like to correctly implement both accelerators, and compare the real-life performance and real-time properties compared to our partly theoretical results. The next step would be to implement a tile collision accelerator like it was described in Chapter 3. This would make it easier to determine the WCET since it would be fixed hardware with deterministic performance.

DDR memory is a resource that is difficult to analyze at a lower level, and it is shared between several components in the system, making it a major point of non-determinism. The four components that access DDR memory are both processors, the HDMI peripheral, and the rasterizer accelerator. Taking into consideration that the instruction and data caches of the processors are large enough to rarely access it, we can assume that their influence on the non-deterministic performance is negligible. Interestingly, the rasterizer accelerator only writes to memory, whereas the HDMI peripheral only reads from it, they never access the same memory region, and the accesses are linear and not random. To lessen the non-determinism of the system due to sharing, it would be interesting to create a FMC daughter board with two separate off-chip memories, which are accessed by the two components in an alternating manner. The memory controllers for these memories could be fairly trivial if SRAM were to be used. Even if DDR memories that need refreshing logic were to be used, a naive state machine in the memory controllers would be sufficient due to the linear memory accesses by the two components.

As was mentioned in Section 5.1.2 of Chapter 5, transferring data over the ring interconnect using a processor tile is very inefficient, and a DMA could be used to increase the latency, throughput, and therefore the performance of the system. The Starburst platform already has gateway entry tiles, which are similar to processor tiles, but contain an additional DMA peripheral. This was not used however due to them not being integrated well into the system generator flow.

Another example where performance could be increased is to use double buffering in the rasterizer accelerator. While it is rendering, another set of primitives could be stored in a second triangle memory block, which could then be rasterized while the result of the previous tile is being sent to memory over the Advanced Extensible Interface 4 (AXI4) port. Since only a fraction of the memory of the BRAM blocks is used, double buffering would not actually increase the amount of BRAM blocks. Allowing the multiple stages of the accelerator to be run in parallel should only increase LUT usage minimally. By using double buffering, the accelerator would be pipelined and achieve a higher throughput, with a slight increase in hardware resource usage.

The current hardware accelerated system uses only one of each accelerator, and we mentioned that the tile size of the rasterizer accelerator was chose somewhat arbitrarily. While its size can be increased to process more pixels at a time and decrease the number of tiles, we would like to scale the system up by adding more of each accelerator to the system. Another angle that we would like to explore is to use multiple rings where each ring implements a full graphics pipeline so that the system more closely resembles the sort-middle architecture of Figure 15.

An interesting step that lies further down the road is to implement fully programmable *pixel shaders* in the rasterizer accelerator. *Shaders* are small programs that run for each vertex (*vertex shaders*), primitive (*geometry shaders*), and rasterized pixel (*pixel shaders*). This way the system can also *render* instead of just rasterize. It would be an interesting challenge to modify the accelerator in such a way that we can give it a pixel shader, all while maintaining a guaranteed throughput and making it simple to analyze. Such an implementation could have multiple small and simple processors that run a pixel shader for each pixel that passes the rasterization step. A good candidate for such a processor is the J1, a small Forth CPU core [9]. Every instruction executes in a single cycle, and given a Forth application, it is fairly easy to determine how many clock cycles it takes to execute and reason about its real-time performance.

6.2 REFLECTION

This project started with my supervisor asking if the Starburst platform could be ported from the ML605 board, which has a Virtex 6 FPGA to the VC707 board, which has a Virtex 7 FPGA. This was attempted before by PhD students but was never a high priority. The main issue porting Starburst is that the Processor Local Bus (PLB) is used for connecting parts of the system together. Xilinx, the manufacturer of the FPGAs, discontinued the use of PLB for the 7-series FPGAs and newer, and switched over to AXI4. This raised hardware and software issues that needed to be addressed. Hardware wise, PLB and AXI4 are incompatible, and therefore all PLB connected hardware had to use AXI4 variants. Also, PLB is big-endian whereas AXI4 is little-endian, meaning that some piece of hardware and software had to be rewritten to accommodate this difference. There are many small differences between the old platform and one that would use AXI4, which led to a partial redesign of the platform.

Below is a list of things that needed to be modified and/or fixed in order to port the hardware to the VC707 board:

- The old platform uses a binary tree structure called *warpfield* to connect the Microblaze processors to the DDR memory controller. This is replaced with just an AXI4 interconnect where the processors and the memory controller are connected to as masters and slaves respectively. Although an initial implementation of an AXI4 compatible *warpfield* component was previously implemented but it did not work. It was deemed unnecessary to fix it since it is difficult to model the real-time behavior of the memory controller, and therefore it would not add much to the ability to analyze the system.

- The dual-ring interconnect network interface IP consists of two parts, one that connects to the ring called *dual ring acc*, and one that connects the Microblaze processor to *dual ring acc* called *plb2ring*. A version of *plb2ring* with an AXI4 port called *axi2ring* was implemented by a PhD student before, but it would occasionally hang. As this is a crucial part of the platform, it was debugged in simulation, and a timing error was found in the AXI4 handshaking where the IP expected to receive more data without the processor receiving an acknowledgement.

- A student created a new version of *dual ring acc* that allows configuring accelerators over the ring instead of using a separate configuration bus, and this is used in the new platform. While using this IP initially meant a change of just a few lines in the generator template, it turned out that it did not work correctly. The master thesis of this student was used in order to get a better understanding of the IP, and finding the bug took much longer than anticipated. This was because of disparity between the thesis and the implementation, which turned out to not to work when accelerators were present on the ring.

- The ML605 board uses a DVI transmitter chip, whereas the VC707 board uses an Analog Devices ADV711 HDMI transmitter. Although Analog Devices released the HDL code to use this chip, it was packaged for the Vivado toolchain, whereas the generator scripts for Starburst use the older ISE/XPS toolchain. Although not difficult, many small changes were necessary in creating an XPS pcore peripheral that can be used by the ISE toolchain. However, the accompanying clock generator IP that they also released does not synthesize with the older toolchain, while it does with Vivado. This IP is necessary to support multiple resolutions which use different pixel clocks. Since this IP does synthesize with Vivado, it is most likely a toolchain issue, and due to time limitations it was not pursued. Therefore only the resolution of 800x600 is supported as it uses an easy to generate 40 MHz pixel clock.

Software wise, several changes were necessary as well, and are listed below:

- GCC had to be configured to generate little-endian libraries and binaries. This involved adding a flag in a couple of makefile scripts.

- The checksum code that loops over the memory where Helix OS resides had to be modified to use little-endian byte ordering.

- The order of struct members that use bit fields had to be reversed, again due to endianness.

- Analog Devices released a library for the Microblaze processor to use the ADV7511 HDMI transmitter chip, but when used, it would hang in the initialization function because it waited for a timer that is not present. Because they did not release the source code, the solution was toreverse engineering their library to find out which registers of the ADV7511 should be set and which values should be used. This effort was documented, and custom functions that correctly initialize the ADV7511 were added to Helix OS.

Some general things that were done to lower the barrier to use the Starburst platform for future students:

- Using accelerators is a big part of using the Starburst platform. While there are multiple accelerators created by previous students, they are large and complex. Understanding them requires looking up the theses that describe them. There is one trivial accelerator that just passes its input to its output, so there is a gap in complexity between the accelerators. In order to address this, two additional simple accelerators were created including documentation and software examples: one that takes two input words and outputs the summed value, and one that allows configuring whether the two values should be summed or multiplied.
- Functions were added to Helix OS that simplify configuring accelerators along with examples that demonstrate this process.
- A commented example application was written to demonstrate how to use the HDMI peripheral.

There are also several changes that were attempted but were not realized in the end:

- On the old platform, one processor connected to the board peripherals runs Linux while the rest runs Helix OS. A separate version of GCC is used for compiling the kernel, and for the new platform, the little-endian flag is used. Linux also requires a DTS file which describes all the devices in the system, and to generate it for the new platform, Xilinx provides a plugin for the EDK tool that does just that. The problem is that when generating a DTS file for the new platform, and error about indexing with the interrupt controller is given. This is a known bug in the EDK tool, but since it is not maintained anymore, the bug will not be fixed. The Vivado version of EDK was used, and while it did generate a DTS file, the kernel would not boot because it thinks there is too little memory in the system. A researcher at NXP did build a working kernel, but getting it running on the

new platform as well as getting the userland applications such as busybox to compile, was put on halt because it was taking too much time to get it running.

- The binary tree interconnect for the tile intersection accelerator was implemented and tested in simulation, but I set a deadline for myself, and decided to spend the remaining time in trying to fix the geometry and rasterizer accelerators as those the biggest bottlenecks of the implementation.

My supervisor warned me not to spend too much time on porting everything over, but since I continuously made progress in small steps, I did not stop and ended up spending about 18 months on this process. The porting process as well as the research for the thesis has taken a lot of time, partially due to me working a part-time job, but also because of bug filled tools, faulty hardware, long synthesis times, and using a design methodology that is not supported and sensitive to making errors. But I learned a lot and gained experience, so I do not see this effort as a waste of time.

Looking back on the implementation of the accelerators, it was a very deceptive process in the sense that during development, everything worked and is still working fine in simulation. In the beginning, even the actual hardware worked most of the time, and when something did not work, it was due to a bug in the rest of the hardware or software. This makes sense as there were many subtle bugs that were present during the porting process. After fixing all bugs that I came across, it became apparent that the usage of the two-process method led to accelerator implementations that would work occasionally.

I assumed that it was because of my accelerators. However more often than not, it turned out to be a bug in the rest of the hardware or software. This gave the false sense that the accelerators were working correctly, and when an accelerator did not work, it was most likely a bug caused by IP developed by others or by the porting process. While this was the case a few times, in the end my implementation is incorrect because I used a design methodology that is prone to generating incorrect results. Both accelerators did work when used in a system where only one accelerator was present to test each accelerator separately, but would sometimes give incorrect results or hang the entire system. In the end, the ISE toolchain generated hardware that has one latch in each accelerator and it is most likely due to an incorrect use of the two-process methodology.

My recommendation for future students and users is to use the newer Vivado toolchain for Starburst on the VC707 board. Also, do not use the two-process design methodology as it makes is very easy to create unwanted combinatorial logic. Using a higher-level HDL such as CλaSH [8] and SpinalHDL [33] is also worth investigating to simplify the development process. If the developed hardware inhibits

non-deterministic behavior, then you most likely have a timing problem related to an inferred latch or incorrect combinatorial circuit. All efforts should be focused on solving that issue, and tested on actual hardware as simulations do not show non-deterministic behavior.

ACRONYMS

AABB	Axis-Aligned Bounding Box
API	Application Programming Interface
AXI4	Advanced Extensible Interface 4
BLR	Bluetooth Low Energy Long Range
BRAM	Block RAM
DMA	Direct Memory Access
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
FPS	Frames Per Second
LUT	Lookup Table
MPSoC	Multi-Processor System on Chip
MV	Model-View
MVP	Model-View-Projection
NDC	Normalized Device Coordinates
PLB	Processor Local Bus
TDM	Time-division Multiplexing
VR	Virtual Reality
WCET	Worst-Case Execution Time

LIST OF FIGURES

LIST OF TABLES

BIBLIOGRAPHY

[1] AMD. *AMD Graphics Cores Next (GCN) Architecture*. Tech. rep. 2012. URL: https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf.

[2] ARM. *Mali GPU OpenGL ES Application Development Guide - Mali Development Strategy*. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0363d/CJAEEJCF.html (visited on 12/15/2016).

[3] Michael Abrash. *Michael Abrash's Graphics Programming Black Book, with CD: The Complete Works of Graphics Master, Michael Abrash*. Coriolis group books, 1997.

[4] Kurt Akeley. "Reality Engine Graphics." In: *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '93. Anaheim, CA: ACM, 1993, pp. 109–116. ISBN: 0-89791-601-8. DOI: 10.1145/166117.166131. URL: http://doi.acm.org/10.1145/166117.166131.

[5] Kurt Akeley and Tom Jermoluk. "High-performance Polygon Rendering." In: *SIGGRAPH Comput. Graph.* 22.4 (June 1988), pp. 239–246. ISSN: 0097-8930. DOI: 10.1145/378456.378516. URL: http://doi.acm.org/10.1145/378456.378516.

[6] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering 3rd Edition*. Natick, MA, USA: A. K. Peters, Ltd., 2008, p. 1045. ISBN: 987-1-56881-424-7.

[7] Tomas Akenine-Möller and Jacob Ström. "Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones." In: *ACM Trans. Graph.* 22.3 (July 2003), pp. 801–808. ISSN: 0730-0301. DOI: 10.1145/882262.882348. URL: http://doi.acm.org/10.1145/882262.882348.

[8] Christiaan Pieter Rudolf Baaij. "Digital circuits in CλaSH: functional specifications and type-directed synthesis." PhD thesis. Enschede, 2015. URL: http://doc.utwente.nl/93962/.

[9] James Bowman. "J1: a small Forth CPU Core for FPGAs." In: pp. 43–46. URL: http://www.complang.tuwien.ac.at/anton/euroforth/ef10/papers/bowman.pdf.

[10] Edwin Catmull. *A subdivision algorithm for computer display of curved surfaces*. Tech. rep. DTIC Document, 1974.

[11] D. Cohen-Or, Y. L. Chrysanthou, C. T. Silva, and F. Durand. "A Survey of Visibility for Walkthrough Applications." In: *IEEE Transactions on Visualization and Computer Graphics* 9.3 (July 2003), pp. 412–431. ISSN: 1077-2626. DOI: 10.1109/TVCG.2003.1207447. URL: http://dx.doi.org/10.1109/TVCG.2003.1207447.

[12] Michael Cox, David Sprague, John Danskin, Rich Ehlers, Brian Hook, Bill Lorensen, and Gary Tarolli. "Developing High-Performance Graphics Applications for the PC Platform." In: *Course 29 notes at SIGGRAPH 98*. 1998.

[13] Michael F. Deering and Scott R. Nelson. "Leo: A System for Cost Effective 3D Shaded Graphics." In: *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '93. Anaheim, CA: ACM, 1993, pp. 101–108. ISBN: 0-89791-601-8. DOI: 10.1145/166117.166130. URL: http://doi.acm.org/10.1145/166117.166130.

[14] B. H. J. Dekens, M. J. G. Bekooij, and G. J. M. Smit. "Real-time multiprocessor architecture for sharing stream processing accelerators." In: *22nd Reconfigurable Architectures Workshop (RAW 2015), Hyderabad, India*. Hyderabad, India: IEEE Computer Society, 2015, pp. 81–89.

[15] B. H. J. Dekens, P. Wilmanns, M. J. G. Bekooij, and G. J. M. Smit. "Low-cost Guaranteed-Throughput dual-ring communication infrastructure for heterogeneous MPSoCs." In: *2014 Conference on Design and Architectures for Signal and Image Processing (DASIP), Madrid, Spain*. Madrid, Spain: ECSI Media, 2014, pp. 157–164.

[16] *Denali Technical Overview*. Tech. rep. Kubota Pacific Computer Inc., 1993.

[17] Jake Edge. *An update on the freedreno graphics driver*. 2015. URL: https://lwn.net/Articles/638908/ (visited on 12/16/2016).

[18] Matthew Eldridge. "Designing graphics architectures around scalability and communication." PhD thesis. STANFORD UNIVERSITY, 2001.

[19] Matthew Eldridge, Homan Igehy, and Pat Hanrahan. "Pomegranate: A Fully Scalable Graphics Architecture." In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 443–454. ISBN: 1-58113-208-5. DOI: 10.1145/344779.344981. URL: http://dx.doi.org/10.1145/344779.344981.

[20] John Eyles, Steven Molnar, John Poulton, Trey Greer, Anselmo Lastra, Nick England, and Lee Westover. "PixelFlow: The Realization." In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*. HWWS '97. Los Angeles,

California, USA: ACM, 1997, pp. 57–68. ISBN: 0-89791-961-0. DOI: 10.1145/258694.258714. URL: http://doi.acm.org/10.1145/258694.258714.

[21] *Freedom 3000 Technical Overview*. Tech. rep. Evans & Sutherland Computer Coorporation, 1992.

[22] Henry Fuchs, Gregory D Abram, and Eric D Grant. "Near real-time shaded display of rigid objects." In: *ACM SIGGRAPH Computer Graphics*. Vol. 17. 3. ACM. 1983, pp. 65–72.

[23] Henry Fuchs, Zvi M Kedem, and Bruce F Naylor. "On visible surface generation by a priori tree structures." In: *ACM Siggraph Computer Graphics*. Vol. 14. 3. ACM. 1980, pp. 124–133.

[24] Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. "Pixel-planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-enhanced Memories." In: *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '89. New York, NY, USA: ACM, 1989, pp. 79–88. ISBN: 0-89791-312-4. DOI: 10.1145/74333.74341. URL: http://doi.acm.org/10.1145/74333.74341.

[25] Om Prakash Gangwal, André Nieuwland, and Paul Lippens. "A scalable and flexible data synchronization scheme for embedded HW-SW shared-memory systems." In: *System Synthesis, 2001. Proceedings. The 14th International Symposium on*. IEEE. 2001, pp. 1–6.

[26] Jonas Gomes, Luiz Velho, and Mario Costa Sousa. *Computer Graphics: Theory and Practice*. 1st. Natick, MA, USA: A. K. Peters, Ltd., 2012. ISBN: 1568815808, 9781568815800.

[27] Dan Gordon and Shuhong Chen. "Front-to-back display of BSP trees." In: *IEEE computer Graphics and Applications* 11.5 (1991), pp. 79–85.

[28] Jon Jordan. *Develop 2011: PS Vita is the most developer friendly hardware Sony has ever made*. 2011. URL: http://www.pocketgamer.co.uk/r/Multiformat/PlayStation+Vita/news.asp?c=31682 (visited on 12/15/2016).

[29] Steven Molnar, John Eyles, and John Poulton. "PixelFlow: High-speed Rendering Using Image Composition." In: *SIGGRAPH Comput. Graph.* 26.2 (July 1992), pp. 231–240. ISSN: 0097-8930. DOI: 10.1145/142920.134067. URL: http://doi.acm.org/10.1145/142920.134067.

[30] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. "A Sorting Classification of Parallel Rendering." In: *IEEE Comput. Graph. Appl.* 14.4 (July 1994), pp. 23–32. ISSN: 0272-1716. DOI: 10.1109/38.291528. URL: http://dx.doi.org/10.1109/38.291528.

[31] Carl Mueller. "The Sort-first Rendering Architecture for High-performance Graphics." In: *Proceedings of the 1995 Symposium on Interactive 3D Graphics*. I3D '95. Monterey, California, USA: ACM, 1995, 75–ff. ISBN: 0-89791-736-7. DOI: 10.1145/199404.199417. URL: http://doi.acm.org/10.1145/199404.199417.

[32] Marc Olano and Trey Greer. "Triangle Scan Conversion Using 2D Homogeneous Coordinates." In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*. HWWS '97. Los Angeles, California, USA: ACM, 1997, pp. 89–95. ISBN: 0-89791-961-0. DOI: 10.1145/258694.258723. URL: http://doi.acm.org/10.1145/258694.258723.

[33] Charles Papon. *SpinalHDL*. URL: https://github.com/SpinalHDL/SpinalHDL (visited on 04/11/2017).

[34] Stephanie Pappas. *Why Does Virtual Reality Make Some People Sick?* 2016. URL: http://www.livescience.com/54478-why-vr-makes-you-sick.html (visited on 11/07/2016).

[35] Juan Pineda. "A Parallel Algorithm for Polygon Rasterization." In: *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '88. New York, NY, USA: ACM, 1988, pp. 17–20. ISBN: 0-89791-275-6. DOI: 10.1145/54852.378457. URL: http://doi.acm.org/10.1145/54852.378457.

[36] Qualcomm. *The Rise of Mobile Gaming on Android - Qualcomm Snapdragon Technology Leadership*. Tech. rep. 2014. URL: https://developer.qualcomm.com/qfile/27978/rise-of-mobile-gaming.pdf.

[37] David F. Rogers. *Procedural Elements for Computer Graphics (2Nd Ed.)* New York, NY, USA: McGraw-Hill, Inc., 1998. ISBN: 0-07-053548-5.

[38] Marcus Roth and Dirk Reiners. "Sorted Pipeline Image Composition." In: *Eurographics Symposium on Parallel Graphics and Visualization*. Ed. by Alan Heirich, Bruno Raffin, and Luis Paulo dos Santos. The Eurographics Association, 2006. ISBN: 3-905673-40-1. DOI: 10.2312/EGPGV/EGPGV06/119-126.

[39] Jochem Hendrik Rutgers. "Programming models for many-core architectures: a co-design approach." PhD thesis. Universiteit Twente, 2014.

[40] Rudrajit Samanta, Jiannan Zheng, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. "Load Balancing for Multi-projector Rendering Systems." In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*. HWWS '99. Los Angeles, California, USA: ACM, 1999, pp. 107–116. ISBN: 1-58113-170-4. DOI: 10.1145/311534.311584. URL: http://doi.acm.org/10.1145/311534.311584.

[41] Hanan Samet. *The design and analysis of spatial data structures*. Vol. 199. Addison-Wesley Reading, MA, 1990.

[42] Ryan Smith. *Hidden Secrets: Investigation Shows That NVIDIA GPUs Implement Tile Based Rasterization for Greater Efficiency*. 2016. URL: http://www.anandtech.com/show/10536/nvidia-maxwell-tile-rasterization-analysis (visited on 12/15/2016).

[43] Rys Sommefeldt. *A look at the PowerVR graphics architecture: Tile-based rendering*. 2015. URL: https://imgtec.com/blog/a-look-at-the-powervr-graphics-architecture-tile-based-rendering/ (visited on 12/15/2016).

[44] Marcel Steine, Marco Bekooij, and Maarten Wiggers. "A Priority-Based Budget Scheduler with Conservative Dataflow Model." In: *Proceedings of the 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools, DSD '09*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2009, pp. 37–44. URL: http://doc.utwente.nl/69797/.

[45] Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. "A Characterization of Ten Hidden-Surface Algorithms." In: *ACM Comput. Surv.* 6.1 (Mar. 1974), pp. 1–55. ISSN: 0360-0300. DOI: 10.1145/356625.356626. URL: http://doi.acm.org/10.1145/356625.356626.

[46] D. van der Veer. *Design of a GMSK Receiver Prototype on a Heterogeneous Real-time Multiprocessor Platform*. 2016. URL: http://essay.utwente.nl/69320/.

[47] Matthew White. *Sega Dreamcast Hardware Analysis*. 2014. URL: https://thesolidstategamer.wordpress.com/2014/07/26/sega-dreamcast-hardware-analysis/ (visited on 12/15/2016).

[48] Ramchan Woo, Sungdae Choi, Ju-Ho Sohn, Seong-Jun Song, and Hoi-Jun Yoo. "A low power 3D rendering engine with two texture units and 29Mb embedded DRAM for 3G multimedia terminals." In: *Solid-State Circuits Conference, 2003. ESSCIRC '03. Proceedings of the 29th European*. 2003, pp. 53–56. DOI: 10.1109/ESSCIRC.2003.1257070.

[49] Xilinx. *Floating-Point Operator*. URL: https://www.xilinx.com/products/intellectual-property/floating_pt.html (visited on 11/02/2016).