October 2018

An Embedded Approach to Volumetric Displays

Jared Alexander Goldman
Worcester Polytechnic Institute

An Embedded Approach to Volumetric Displays

A Major Qualifying Project

Submitted to the Faculty of Worcester Polytechnic Institute
in partial fulfillment of the requirements for the Degree in Bachelor of Science in

Electrical and computer Engineering

By

_____

Jared Goldman

Date 5/1/17

Project Advisor: Stephen J. Bitar

_____

Table of Contents

Table of Figures

Table of Tables

Abstract

The goal of this capstone project was to design and create and test a functioning volumetric display system. This project created an embedded solution for a volumetric display that can perform the necessary data manipulations required to prepare and slice a three-dimensional object for so that it can be displayed in real space.

1 Introduction

The goal of this MQP is to create a volumetric display unit. Two different approaches to creating this system were explored and compared against each other. Both approaches attempt to create a low cost, simple to use display unit. The goal is to provide a simple all-in-one device that can produce a volumetric display for a fraction of the cost compared to other systems that are available.

The first approach uses a combination of FPGA logic and embedded design to create and display the sliced images required for the display. Different tasks are divided between the two systems to best utilize the strengths of each system. The model manipulation and projection unit control were done on the embedded side, while image slicing was done on the FPGA side of the system. This has the adverse effect of requiring a large amount of data transfer between the systems. The nature of the FPGA allows for all the calculations to be done quickly and accurately. The major downside of this approach is that the process of transferring data between the systems is slow and adds large amount of latency to the entire unit.

This MQP will also explore a purely embedded approach to solve this problem. All the tasks that were previously performed on the FPGA will instead be performed on the embedded system. This will eliminate the need for the slow data transfer. However, this does mean that the powerful parallel processing power of the FPGA is lost. The results of this approach will be compared against the results of the FPGA to find the optimal solution.

This project was undertaken in order to discover a cheap, versatile solution to creating volumetric displays, which have been unable to see a commercial release. This project serves as an important step to creating a volumetric display that is capable of displaying real time

video. This report discusses the success of the embedded solution and how it compares to the FPGA version. Finally, it discusses potential areas for future work in the volumetric field and how the current display system could be improved.

2 Background

2.1 Volumetric Displays

A volumetric display is a three-dimensional projection that exists in a transparent volume and is viewable from multiple angles. It is very similar to the science fiction concept of a hologram. The first volumetric display was postulated in the early nineteen hundreds, however, as of yet they have not received the necessary development to see widespread implementation [8].

Methods to create volumetric displays include swept-volume, moving display, static volume, and plasma-point. A swept volume display uses a rotating object as screen that images are projected onto. It relies on the way humans perceive moving objects to create what appears to be a three dimensional object [20]. Moving-display is uses a similar method as the swept volume approach. The main difference is that the projector is rotating rapidly instead of the screen [20]. Static-volume approach uses no moving parts and instead projects into a semitransparent volume. The three dimensional image is then created inside of this volume [7]. The plasma point method uses an extremely powerful laser to create balls of plasma in the air. This creates glowing dots in the air that can then be used to draw an image [19].

The swept helix approach displays the three-dimensional object through a series of still two-dimensional images. These images are displayed in quick succession onto a rapidly rotating helix shaped screen. To achieve the volumetric effect, the three-dimensional object must be sliced into several two-dimensional images [23]. This works by finding the intersection

points between the object and the helix and projecting only those points. Figure 1, shows demonstrates how slices are found.

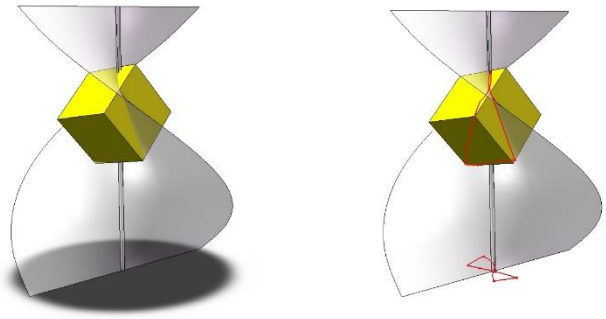

Figure 1: Helix object intersection [2]

In order to display the whole image multiple different intersections must be taken, each one at a different rotation position. By displaying each intersection at the correct time the full object can be displayed and viewed.

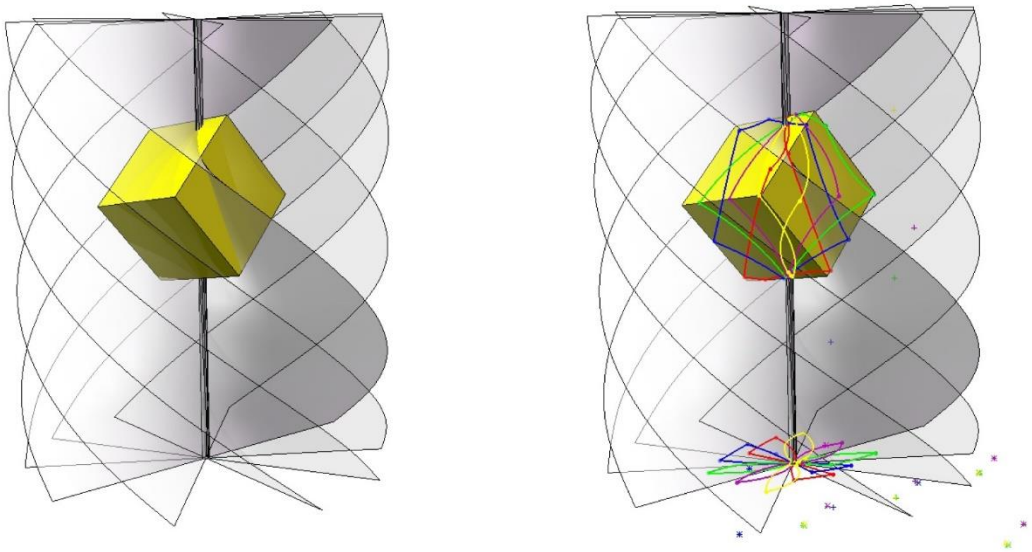

Figure 2: Full rotation intersections [3]

However, to achieve a volumetric image this way, one needs to project the slices at a rapid pace. For a normal two-dimensional image a minimum of fifteen frames per second are needed to display a stable image. In a volumetric display, all the slices must be displayed within the same period of a single

two-dimensional frame in order to get a stable image. This means that to display an object that has been divided into twenty different slices a total of three hundred frames per second would be needed [4]. When more slices are used, the vertical resolution the display will be greater. Swept helix approach does have the downside of having a sizeable 'dead zone' in the center of the object [23]. This is an area that cannot be projected onto due to the shape of the object. In the case of the swept helix approach, the center of the object, where the rotational shaft is placed, will not be able to have any projection on it. This will cause what will appear to be a hole in the object.

2.2 Model Manipulation

To convert a model into a volumetric image the model must first be processed into a voxel format. A voxel is a "volume pixel" that represents a parallelotope, on a regular grid of three-dimensional space [17]. Three-dimensional objects are usually represented as a mesh model. Three-dimensional mesh objects are a stored as a series of vertices, edges, and faces that define the surface of the object. Mesh models usually only represent the surface of the object, while voxels represent the volume of the object. To convert from a mesh format to a voxel format the mesh must undergo the process known as voxelization.

Mesh models are built from many different polygons, such as triangles and quadrilaterals. There are many file formats for storing a mesh object, including the stereolithography (STL) format. These files describe a raw unstructured triangulated surface. The more polygons that are used the more detailed the object becomes, and the larger the file.

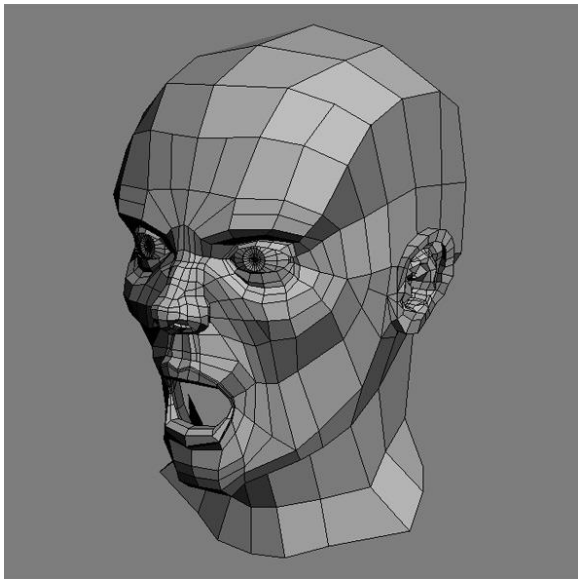

Figure 3: Mesh model [12]

Voxelization is the step of converting a mesh model to a model that is represented by voxels. This process takes the mesh and maps it into three-dimensional space. A volume of three-dimensional space that is just large enough to contain the entire mesh is found. This boundary space is then divided into a regular grid with specified values for the X, Y, and Z resolution. Each parallelotope in the grid is then checked to see if it contains a piece of the mesh model. If it does then that parallelotope is defined as being present in the voxelized model.

Figure 4: Voxelization example [7]

The resolution of the voxelization process decides the size of each voxel. Smaller resolutions result in bigger voxels that have less accuracy, but take up less storage space. Voxels are stored in the same fashion as bitmaps. This means they are stored in an array that only contains the color information about each voxel. The position of the voxel is inferred based on the size of the resolution of the object. For instance a 100x100x100 object would contain 1,000,000 voxels and the 30^{th} voxel in the array would be located on the top level of the object [17]. Since voxels contain depth, and meshes do not, this makes them the optimal tool for finding intersections between models.

2.3 Projection Technology

In 1987 Texas Instruments patented Digital Light Processing (DLP) projectors. These make uses of a technology known as Digital Micromirror Devices (DMD) to display images. A DMD is a matrix of microscopic mirrors that each represents a pixel. A special controller board usually controls the DMD. These mirror move rapidly to reflect light through a lens to display the image [14].

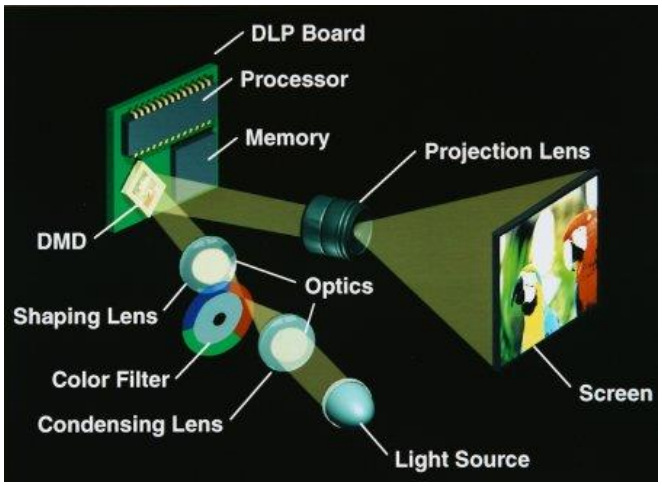

Figure 5: DLP Technology [11]

The entire image is displayed at the same time, instead of the pixel-by-pixel approach that is used on conventional projectors. Each color is displayed one at a time, and the intensity is decided by rapidly switching the projection on and off. High-end versions use multiple DMD's at the same time to produce the whole image in single clock cycle. This technology has the drawback of being harder to work with and having less color fidelity. It is also far more expense then standard digital projectors. However, it can offer much higher frame rates than the competing technologies. It is possible to even achieve tens of thousands of frames per second with this technology[14].

2.4 Hardware Platforms

There are three different hardware platforms that are used in this project. The Avnet Zedboard, the Texas Instruments DLP Lightcrafter EVM and the SolidRun Hummingboard-Gate I2eX. These hardware devices were chosen since they offer the best set of features needed to complete the design goals of the project.

2.4.1 Avnet Zedboard

The Avnet Zedboard is an evaluation board that utilizes a Zynq-7000 System-on-chip (SoC). The Zynq-7000 SoC is a combination of a field programmable gate array (FPGA) and an embedded system. The FPGA is the equivalent a Xilinx Artix-7 FPGA. It features 85,000 logic cells and 53,200 look-up tables. It also offers 106,400 flip-flops and 220 DSP slices. It has access to 140, 36 kb block memory modules, which can be shared with the embedded system. The embedded system is a Dual-core ARM Cortex-A9 MPCore processor. This is a 32-bit system with 256 KB of on-chip memory and 8 direct memory access channels, 4 of which are dedicated to interacting with the block memory of the FPGA. The board also has an addition two DDR3 RAM chips that provide another 512MB of memory. The board features a SD card slot to expand the flash memory or load another operating system.

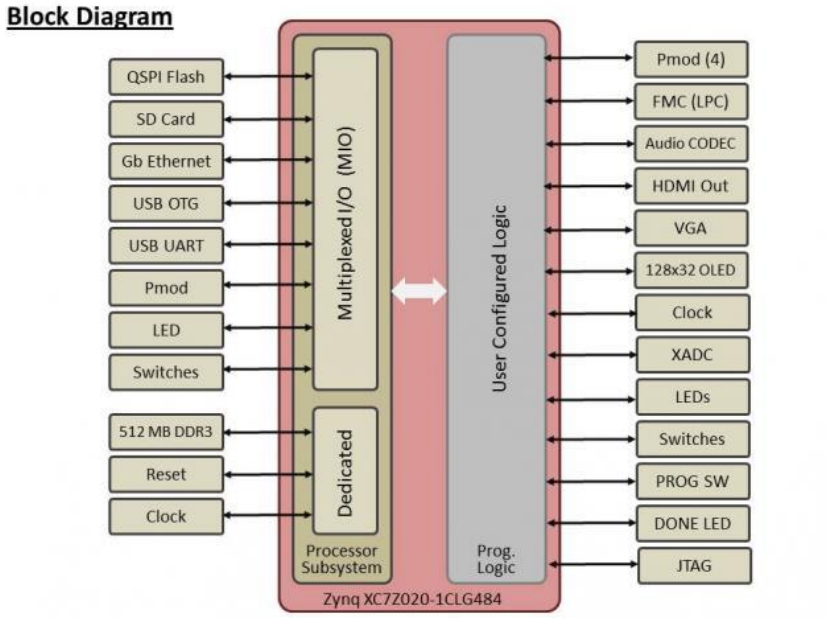

Figure 6: ZedBoard Block Diagram [25]

This device was mainly chosen due to its ability to interact between the FPGA and the embedded system. The original design of the project required that large amounts of data to be shared by the two separate systems, so it was imperative to have a hardware system that could handle the data volume. The Zedboard provides both this functionality with the required performance at reasonable price point.

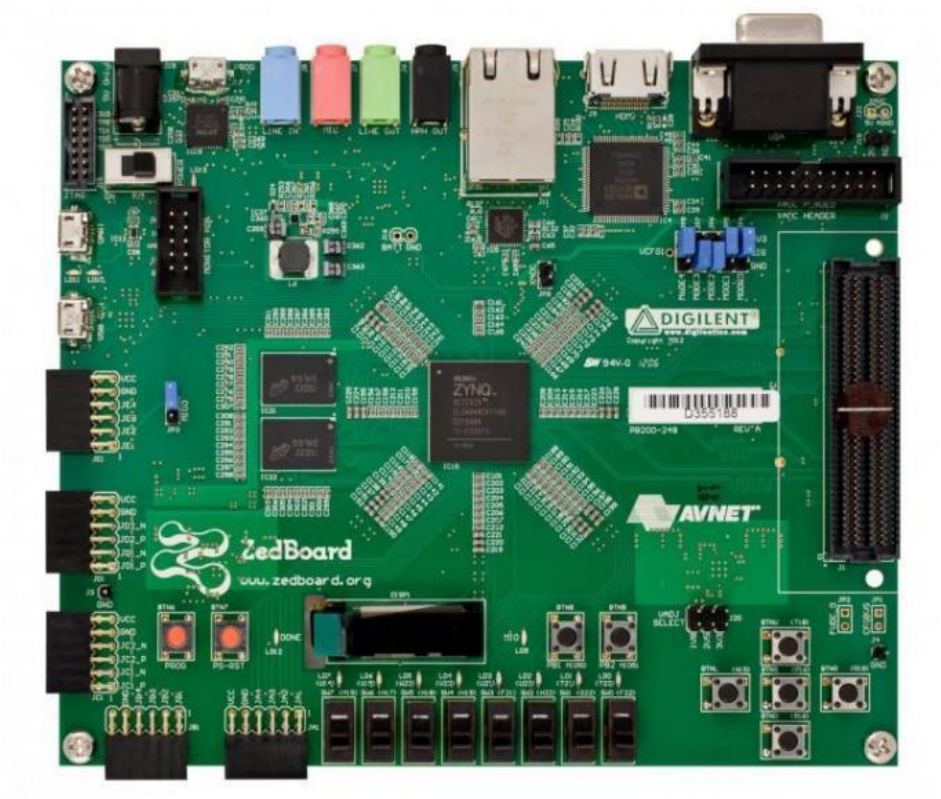

Figure 7: ZedBoard Front Side [26]

2.4.2 Texas Instruments DLP Lightcrafter EVM

The Lightcrafter EVM is an evaluation module for Texas Instrument's patented DLP technology. The unit's DMD features over 400,000 mirrors in a diamond pattern. The projector has a resolution of 608x684 and can achieve up to 4000 frames per second in binary pattern mode. The Lightcrafter is controlled using a combination of a FPGA and a digital processor that runs a version of embedded Linux. The FPGA directly receives inputs from the HDMI port and the external trigger. The USB, UART, and SD are sent to the processor. The processor is responsible for converting these inputs into a format the FPGA can work with and then sending them to the FPGA. The FPGA converts the data into two separate streams of data, a stream of LED color data and a stream of video data. That is then sent to the LED driver and the DLP chip so they can be displayed.

Figure 8: Lightcrafter EVM [6]

This device was chosen as the display unit due to its simplicity and usability. Other competing options called for more a far more complicated setup. The Lightcrafter also offers increased flexibility since it offers a great range of options and modes that could be useful. Finally, it offers all of this at a reasonable price point.

2.4.3 SolidRun HummingBoard-Gate i2eX

The SolidRun Humming board is a small form factor modular minicomputer. The hummingboard features a modular design that allows for the processor to be swapped out. The i2eX version of the board features a MX6 microSOM processor board. This side board features a 64 bit 1GHz dual core ARM A9 processor and a Vivante GC2000 Graphics processor. The board features 1 GB of DDR3 ram. The board offers a variety of inputs including four USB 2.0 and 30 GPIO pins. It also features a mPCIE port.

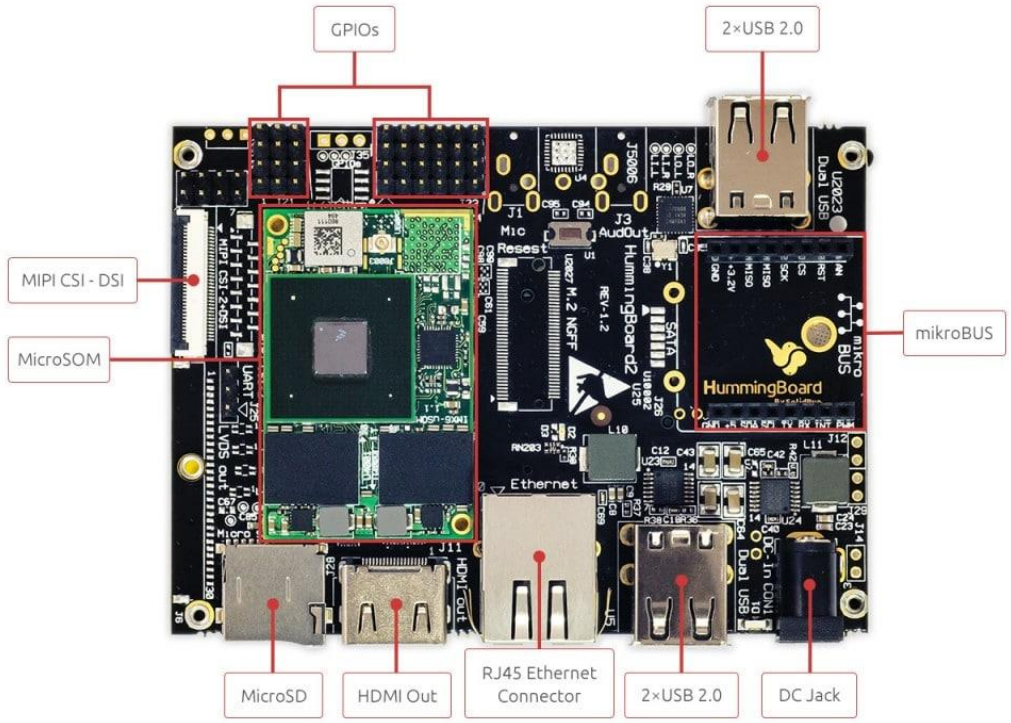

Figure 9: HummingBoard-I2EX and Components [11]

This device was chosen since it offers similar power as the Zedboard's processor. A dual core A9 processor was necessary to get an accurate comparison for the two different processing solutions. The HummingBoard offers the best price point for any A9 system.

3 Design

3.1 Design Goals

This project manipulated a three-dimensional object in a way that allowed it to be used to create a volumetric display using the swept helix approach. First, a three-dimensional model needed to be voxelized so that it could undergo image slicing. Intersections between the digital model and a digital version of the screen were found so that a series of image slices could be created and then converted into images. These images were then sent to the Lightcrafter projector unit so that they could be displayed. The image slices are then projected onto a rotating screen that is synced to the projector. The three-dimensional object is fully displayed on the rotating screen.

Three-dimensional objects start as mesh models. This model only contains the outside surface of the object, and not the complete volume. Since the project required that all intersection points between the model and the screen are found, the model needed to be converted to a format that has volume. The process is called voxelization, and the model is converted to a format that has volume along with correctly filling the model so that it was not empty. Instead of being represented by a series of triangles along its surface, the model Is now represented by a series of three dimensional pixels.

Next the object was sliced twenty times. Slices were created by finding the intersection points between object and a model helix. Rotating the helix by eighteen degrees and finding the new intersections calculate new slices. The model helix that is used to find intersections is identical to the real-world helicoid screen that the projection will be displayed on. These slices are used to figure out what parts of the model will need to projected at each frame.

After all the intersection points have been found the slice images need to be created. This is done by removing the Z dimension (or height) from all the points and displaying all the points on a single flattened image. Due to the nature of the shape of a helix there will not be multiple points of

intersection that share the same x and y coordinates, so there will be no overlapping points. This flattened image is what will need to be projected to create the three-dimensional illusion.

Each image needed to be saved as a 1-bit depth bitmap. This means that the image is a black and white image with no gradient, each pixel is either there or its not there and is defined by a single bit. This is so that the images can be displayed at high frame rates on the Lightcrafter projection unit. A series of twenty images needed to be created. This number was chosen to give the object a greater height resolution. Any more than twenty images and it starts becoming too much data to transfer. Each image needed to be twenty pixels by twenty pixels so that they are square. The size was chosen so that images can be transferred quickly to the Lightcrafter system.

The Lightcrafter projector and the motor display system needed to be kept in sync. Otherwise, an image that does not correspond to the current rotation of the screen will be displayed. Thus the projector needed to respond to the current position of the helix and change images accordingly. An encoder wheel and circuit were designed and implemented to keep the systems in sync. The projector will only switch to a new image when it gets the signal from the encoder that the screen has rotated that far.

3.2 Design Overview

3.2.1 Original Design

The original project planned to use device known as a ZedBoard to implement the design using a combination of a field programmable gate array (FPGA) and an embedded system. The original goal was to divide the work between the processor and the FPGA. This way each system could perform the functions it was best suited for. The original plan can be seen in figure 10.

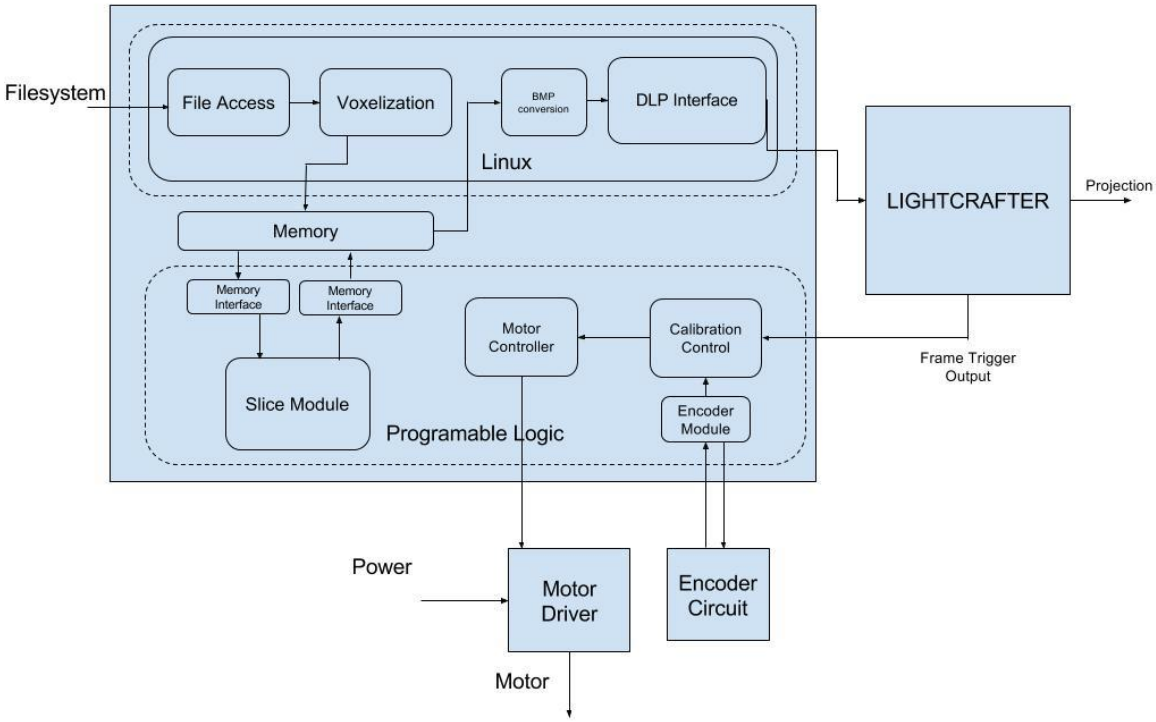

Figure 10: Original System Block Diagram

In this block diagram you can clearly see the four different systems and how they are connected. The large square represents the Zedboard; the two sections that define the embedded processor (linux) and the FPGA (programmable logic) are shown with the dotted lines. The operations performed by each

of those sections are shown, along with the block memory that the systems use to communicate. The other systems include the Lightcrafter projector, the encoder circuit and the motor driver.

The nature of FPGAs makes it the ideal system for performing slicing math on. The FPGA works completely in parallel, so all the intersection points could be found simultaneously, significantly reducing computation time. The FPGA was also chosen to be the motor controller, since it had the correct outputs to connect to the motor driver. However, it still has many downsides. The biggest of which is the inability to directly interact with the file system, where the display model is stored. This means that getting the data off of the file system has to be done on the embedded side, and then transferred over to the FPGA. The FPGA also cannot work with the original model file; the model must first be converted into a format that it understands. The steps required to convert the original model must also be performed on the embedded side. The FPGA is also unable to directly interact with the Lightcrafter projector. This means that after the slicing occurs the data then must be transferred back to the embedded system from the FPGA, and then sent to the Lightcrafter. The data is transferred by using Block Ram (B-Ram) that is shared between the two systems. This is a system of shared memory that both the FPGA and the embedded system can access. To transfer data, one system must save it to the ram, then the other one can access it and read the data. The process of transferring the data between the two systems is slow, particularly when performing a FPGA save and embedded load. The slow data transfer is what inspired another solution to be researched.

The slice is then saved as a bit map file with a height and width of twenty pixels. An example can be seen in figure 11. Each circle in the image represents a pixel that will be present in the projected image. This image is then sent to the Lightcrafter to be displayed. The Lightcrafter will stretch the image automatically so that it will fit the dimensions of the projector and then display it onto the spinning screen.

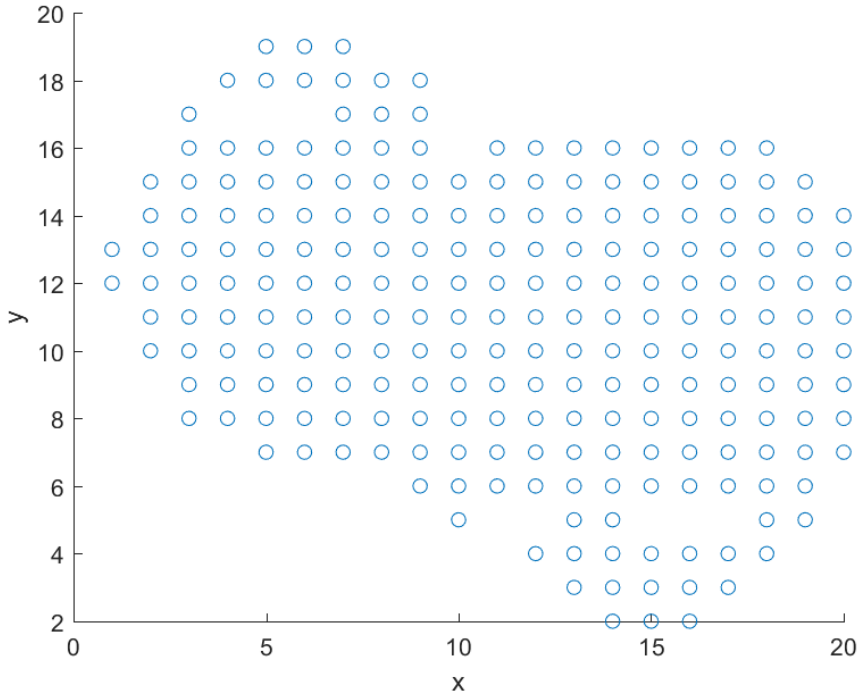

Figure 11: Example Slice

3.2.2 Embedded based design Overview

The other part of this project seeks to remove the FPGA components from the design. This means that the slicing module needed to be written so that it will run on an embedded system instead of the FPGA. The embedded system does not have anywhere near the parallel power of the FPGA, but instead has other benefits like better access to RAM and a more versatile architecture. The purely embedded design does confer some other obvious advantages. The entire operation is much simpler to execute on an embedded system when compared to an FPGA. It also has access to enough resources to load both object files at the same time. A block diagram showing how the slicing program works can be seen in figure 12.

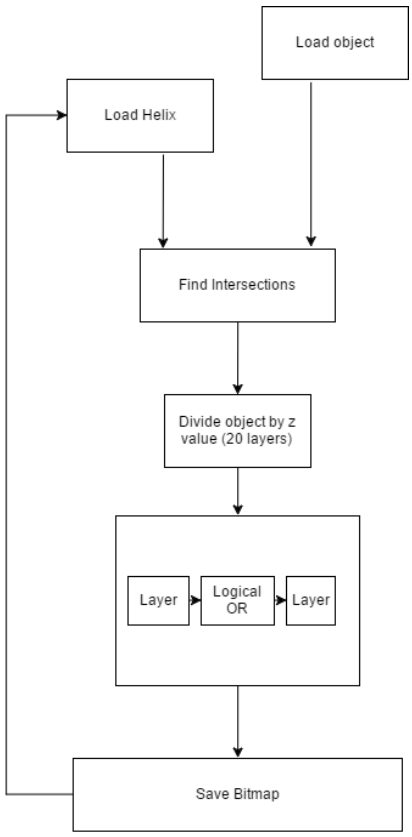

Figure 12: Program Block Diagram

The program works by first loading in a voxelized helix and the voxelized object. These are stored as a binary file each containing eight thousand bits. The intersections between the two objects can be easily found by performing an AND operation on them. Since each voxel is just one bit the helix can serve as a mask for the object and all the intersections can be found quickly.

After the intersections have been found the intersections need to be changed to a 2-dimensional bitmap. To do this we start by having the object divided into twenty layers by their z values. Theses layers can be found since each layer is stored as a consecutive 400 bits. This means finding the next layer is a simple task. The layer then undergoes and OR operation with the final image, which starts out blank. This means that each one of the twenty layers are effectively OR together to get the final image.

This operation replaces the slicing operation that was previously on the FPGA section of the Zedboard. The other operation that the FPGA preformed, the motor control, is now being performed by a separate dedicated system.

3.3 Voxelization

To begin the volumetric display process, first a three-dimensional model needs to be voxelized. This process is instrumental since it changes the model into a format that can be worked with. Without this process the model is just a hollow outline of the object and cannot be used to find intersection points.

The process begins by putting the chosen three-dimensional model in a cube shaped volume, just large enough to fit the entire model. This volume is then divided based on the chosen resolution. The X, Y and Z values are then in so that it is possible to decide the shape of each voxel. Next the voxels need to be filled in. First every voxel that contains any part of the mesh is filled in. Once all the points along the mesh are found then every point between the mesh is filled in. This process only works with three dimensional mesh objects that are a closed single interlocked shape. If the object is not interlocked the entire volume will fill in. Intricate shapes can sometimes lead to complications, which require the resolution be made higher. This gives a much rougher object that what the original mesh design can accomplish, and it also takes up significantly more file space. An example of the process can be seen in figure 13.

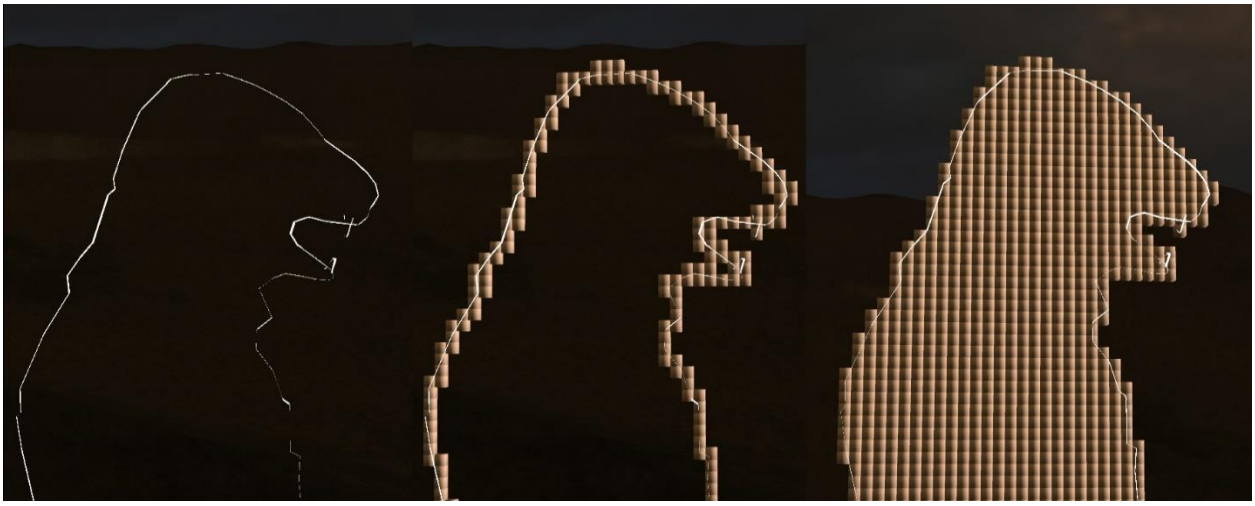

Figure 13: Voxelization Example [7]

This figure shows a crosshatch of a three-dimensional object that is being voxelized. The steps can clearly be seen. Each picture represents a step. The first image on the left shows the original mesh object. The second picture shows the object after the outside mesh has been fully turned into voxels. The last picture shows the filling in of the model.

A simulation voxelization was also done with Matlab. This clarified the process that was needed to create the voxelized model. This provided also provided a baseline to compare the other programs against. Matlab was able to perform the voxelization and state the exact points of every voxel. Models also had to be found or created for this project. The test model of a car seen in figure 14, has been voxelized in figure 15 to show the result of the process. This example make it is clear form this how much definition is lost in the voxelization process.

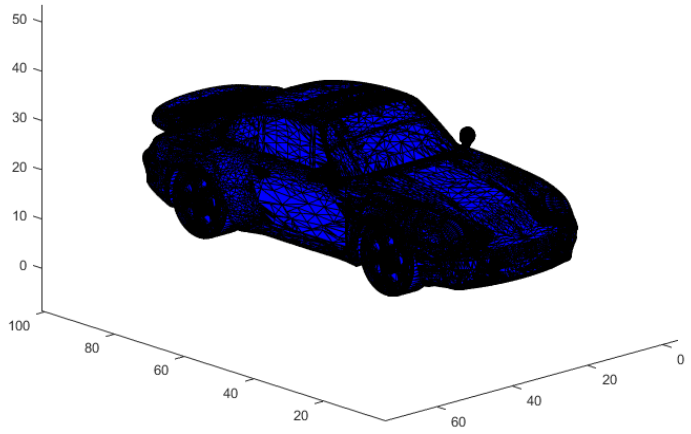

Figure 14: Mesh Model Car

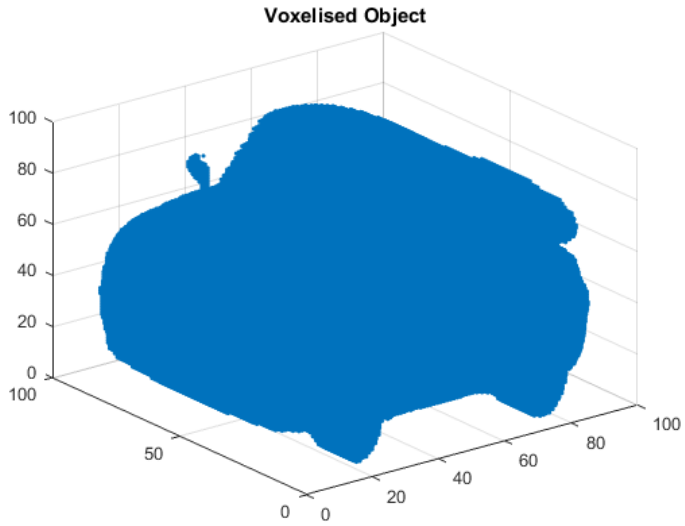

Figure 15: Voxelized Car Model

The helix model had to be created for this project. With the help of an outside source, a CAD model of the real-world helix was created, with the necessary thickness so that it could be used in a 3d printer. This model was eventually used to print the screen that the object would be projected onto. The model was then spun to create the twenty different angles of the mesh. Each one was saved as a

separate object and voxelized so that intersection can be found. The rotations of the helix can be seen in figures 16-18.

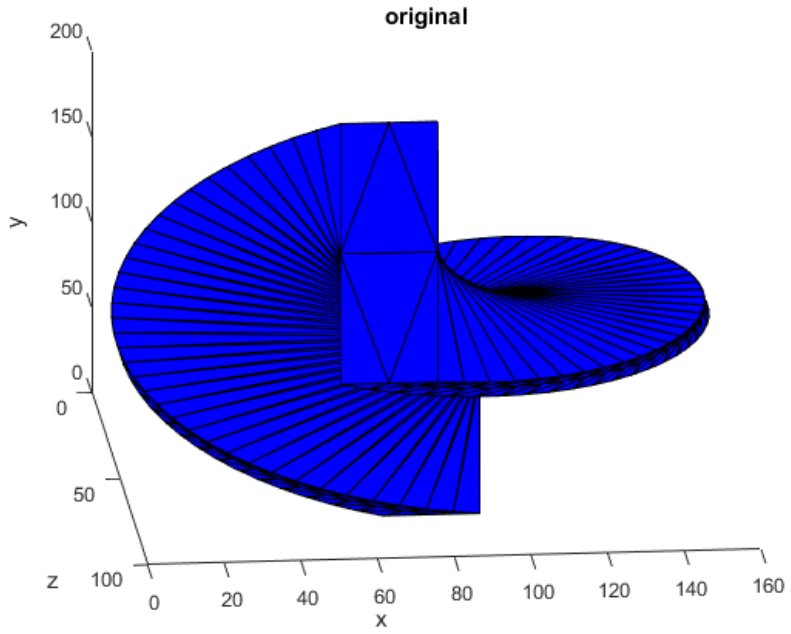

Figure 16: 0 Rotation Helix

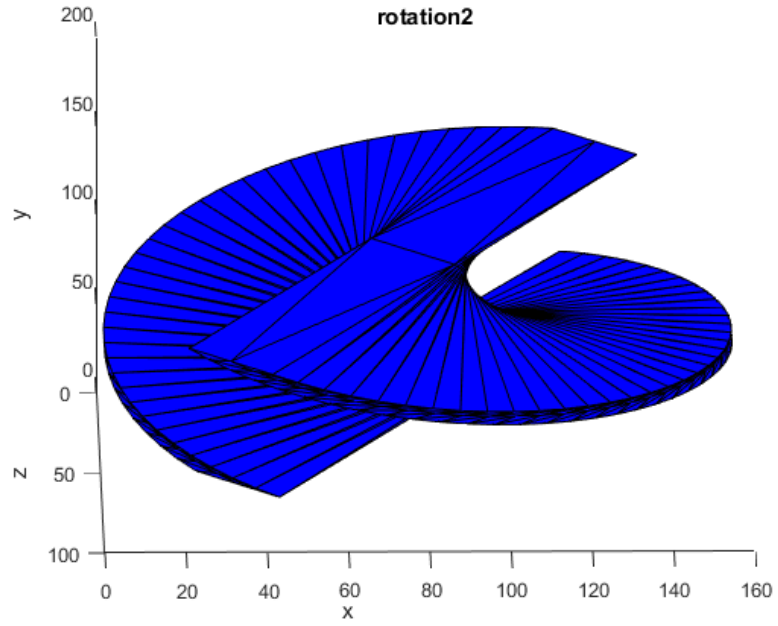

Figure 17: 45 Degree Rotation Helix

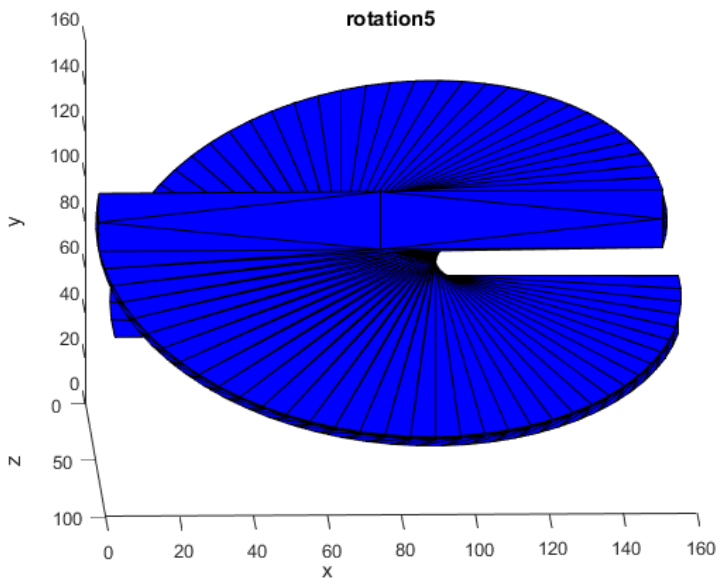

Figure 18: 90 Degree Rotation Helix

There already exists several tools to perform voxelization, and couple of them were considered for use in this project. Creating another program to do this was both out of the purview of the project and completely unnecessary.

The first one that was looked at was Polygon Mesh Voxelisation by Adam H. Aitkenhead [9]. While this system worked and proved to generate good models, it could not run on the embedded system. It relied on a direct control and did not provide any source code to allow for recompiling onto the arm. It was still used to create the models that were used for testing and simulation since it was easy to work with and provided a way to view the models before the final save. Even the final Helix models were voxelized with this program.

Binvox by Patrick Min provided a better solution[16]. His program was compatible onto the arm system and able to create a raw file for the voxelization that was easier to send to FPGA. It also allowed for the removing of all color parameters, which are unnecessary for this project and help save a large

amount of space and RAM. It used it own format for saving the file that made it hard to use with a viewer. This served as the final voxelization tool used in the project.

3.4 FPGA Desgin

The original design called for a multiplatform design that made use of two different kinds of systems. The FPGA part of that system preformed the calculation intensive processes, such as the object slicing, and the function that required GPIO connections, such as motor control.

3.4.1 Processing System Wrapper

For the Zedboard to enable the embedded processor built into the Zynq System on a Chip (SoC) a wrapper is created. This allows the Programming Logic to see the different connections of the embedded system. This allows for the vital connections, such as power and clock, to be made to the embedded system so it can run. It is also an important part is making it so that the embedded system will be able to access the FPGA since the wrapper allows for the two systems to see each other and interact.

3.4.2 Bram Controller

The FPGA/Embedded combination solution requires a way to transfer data between the two systems. The Zedboard provides this by having a set of special RAM that is shared between the two systems. This special block memory (BRAM) has to be initiated and controlled by the FPGA's programming logic. The Zedboard's Zynq SOC has one hundred and forty blocks of BRAM that can be addressed. Each block is 36kb is size and has a maximum bus width of 75. The BRAM controller allows for these blocks to be customized and filled as necessary. It was decided that one block would be used for each of the twenty-helix rotations. In addition, one block would hold the unsliced three-dimensional model and one last block would be used to store all of the sliced images. This resulted in twenty-two separate blocks of memory. Since the BRAM controller can preload data, the twenty helix slices could be preloaded into their RAM to reduce the number of load and save operations that the device will need to do. Figure 19 shows the block RAM setup.

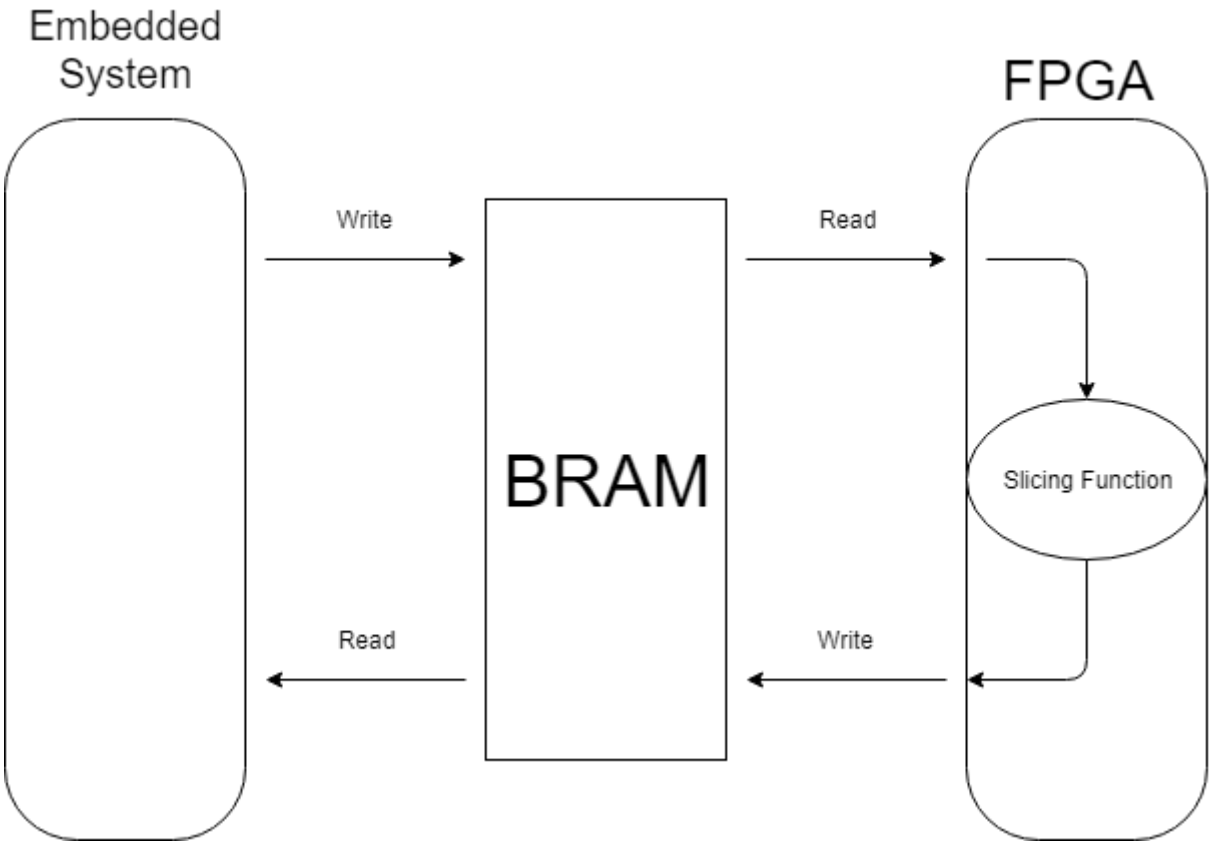

Figure 19: Block Ram Functioning

This figure shows the many different instances that the block ram must be accessed. Since the data cannot be accessed directly the data must be saved into special memory by one system and then loaded by the other. This process must happen twice, once to get the data to the FPGA and once to get the data back to the Embedded system.

3.4.3 AXI Interconnect

The different parts of the FPGA Programming logic are all connected together thanks to the Advanced eXtensible Interface (AXI)[24]. The AXI protocol was developed by Xilinx to serve as the interconnection between the different IP cores. An AXI interconnect module was generated in the Programming Logic so that the necessary connection to the Embedded system could be made. Namely

it is used to setup the BRAM and General-Purpose Input/Output (GPIO) connections. This allows the connection to the encoder and the Embedded connection to the BRAM. This setups the actual physical connections that are needed, much like plugging the correct devices in.

3.4.4 Slicing

The custom logic for the slicing is designed to allow for fast parallel computations. Once the slice processor is activated it begins to extract the voxel data of the object and helix from the BRAM. The helix is pre-saved onto twenty different BRAM blocks, one for each rotation. Preloading the helix rotations onto the BRAM reduces total run time by a massive number, since the embedded system is not required to save the additional twenty objects. Each helix rotations are read in one a time. Once a helix is read in the two-dimensional slice is generated with two operations. The first is a bitwise AND, to find all the helix and object intersections. Then each point is mapped based on their x and y positions into a twenty by twenty array. By directly mapping them to the final array the parallelization of the FPGA can be maximized, since all intersections can be found simultaneously. The slice is then saved into BRAM while the next helix is loaded in. All the slices are saved into the same block of BRAM to maximize loading efficiency. Once the all the rotations are completed and saved in, the embedded system is ready to take and convert the data for use in the projector.

3.4.5 Encoder

The FPGA encoder logic was very simple. The encoder circuit already generated the correct waveform to control the image transitions of the projector. This means that the encoder logic was mainly responsible for making sure the two systems synced up correctly. It would do this by making sure the helix was at the correct starting point before projection would begin. This is done by having a second ring on the encoder wheel that represented the start positions. This section of FPGA logic would spin the screen till it was at the correct speed. Once the motor hits the correct speed it would then wait

until the screen was at the starting position before beginning the projection. After the projection was started the encoder would make sure the motor would not go out of sync. It would do this by setting the projection back to the first frame whenever it passed the origin point.

3.5 Digital Solution

The second solution utilized a fully digital approach. The FPGA was completely removed from the system and all of it components were instead implemented digitally on the embedded system. This allowed for the two different methods to be directly compared against each other

3.5.1 Reasoning

The FPGA/embedded combination-based solution had several major flaws that inspired another approach towards this project. This second approached aimed to eliminate the worst of these issues by using a completely embedded based solution. It does this by taking a simpler and more flexible approach.

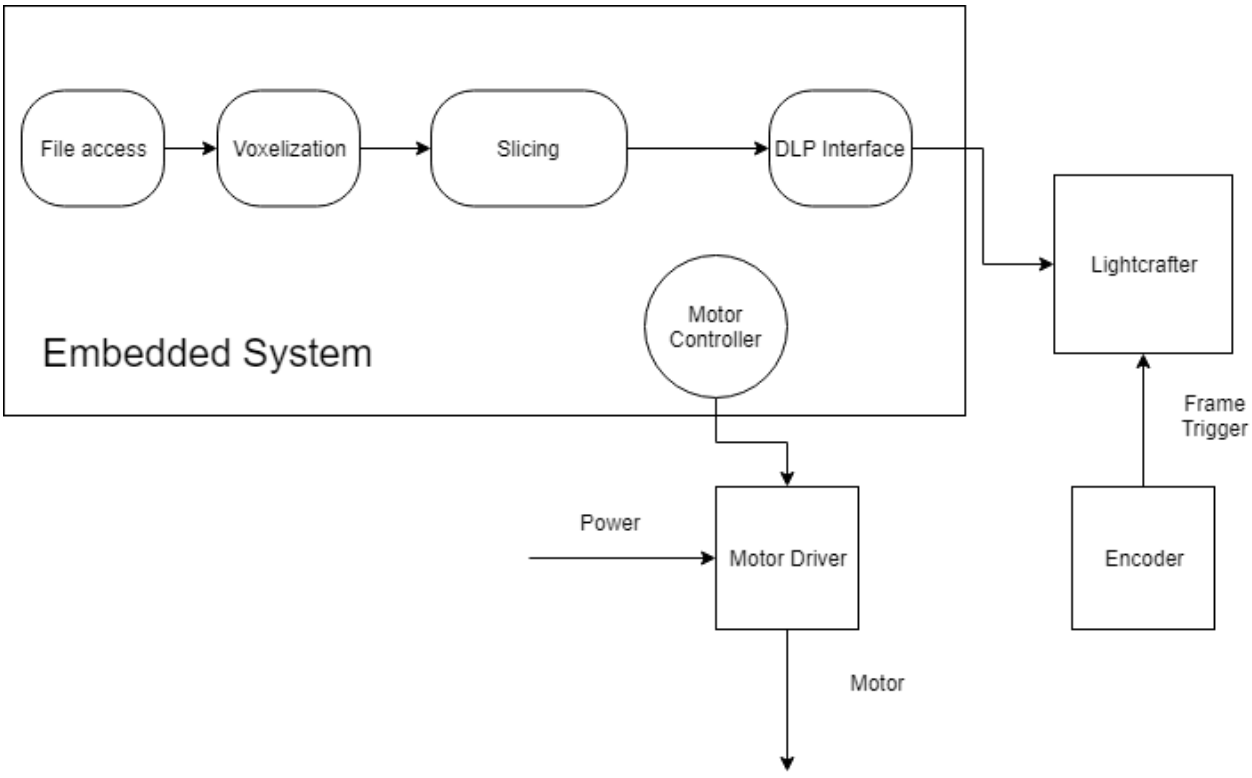

Figure 20: Embedded Overview

Figure 20 shows the embedded only system. The different blocks inside of the large embedded system block represents the different programs that will be run. The arrows show the information flow and how the different parts of the device interact with each other. The final embedded solution has five different parts that run on the embedded system, the file access, voxelization, slicing, DLP interface and the motor controller. These programs work together to complete the tasks that were previously completed by the two separate systems.

The existence of two different types of processing units adds a large amount of complexity to the project. A big part of this complexity is in the interaction between the multiple parts of the system. Since each part would be controlled by either the FPGA or the embedded system, the synchronization of the different parts of the system becomes much more complicated. The synchronization requires the addition of wait time to the system, further increasing the total time needed to create each frame for projection. In addition, the FPGA cannot handle complex data types, requiring that the data be broken down and delivered to it in a simpler form. This adds more steps to the process and makes it take even longer.

The biggest issue that the division of tasks causes is a need for data transfer. For the FPGA to get the data it must be first saved into a special kind of RAM known as BRAM (Block RAM). Both the embedded system and the FPGA can access the BRAM, however only one can access it at a time. This means that the one system ends up stalled and waiting for the other to transfer the data. Since three-dimensional models can be large, the transfer can take a long time. In fact, the transfer time for each frame is far greater than the time it takes for the intersections to be found. To worsen matters, each transfer is required to be made twice, once to get it to the FPGA and once to read it back to the embedded system. A purely embedded approach removes the need for this data transfer. The removal of the slow data transfers could allow for the purely embedded approach to be faster than the combination approach, despite losing the immense parallel processing power of the FPGA.

Each data transfer takes immense amounts time when compared to the other functions. The first transfer, and embedded write, requires the system to load the projection object into ram and then begin transferring it one byte at a time into the BRAM. This transfer must be done using the FPGA clock, which is significantly slower than the embedded clock. The next transfer is from the BRAM to the FPGA. This transfer takes more time then the entire slicing operation, since the data must be removed sequentially. Fortunately, this can occur while the FPGA loads the helix models, so little efficiency is lost. However, the next transfer requires that 20 separate flattened bitmaps be saved and retrieved from the BRAM. This is the slowest operation of them all and takes around 50 clock cycles for each slice to be saved into ram, for a total of 2000 clock cycles. This is almost ten times longer than the slicing operation took and serves as a massive bottleneck on the system. Finally, the embedded system must then load all the slices before it can transfer them to the light crafter projector. This takes slightly longer then the embedded load, but it is comparable.

The use of the FPGA in this way also significantly limits the flexibility and expandability of the project. The BRAM becomes a major limiting factor in both speed and size. The BRAM transfers are not fast enough to allow for real time video to be generated, since each transfer is longer than the frames duration. It also severely limits the size of the object that can be used. Since the BRAM is so small the objects that are going to be displayed must be relatively low resolution. This provides a significant hardware limitation to expanding on and improving the system.

On the contrary the embedded system can utilize much larger amounts of RAM and flash storage than the FPGA. This means it is possible to have much larger and more complex object sliced and displayed. It also makes the system more flexible in many ways, such as being able to quickly replace the screen or make other hardware modifications. By using only one system, wait times are significantly reduced, meaning the system will be far more time efficient.

3.5.2 Requirements

To fully replace the FPGA several functions, must be transferred to the Embedded system. On the hardware side the motor control and timing must be implemented. These systems are simple and recreating them is a basic task. The other more complicated change is to bring the slicing to the embedded system, and to make it as efficient as possible.

The slicing operation has three key steps that need to be replicated. Loading the models, finding the intersections, and saving the intersections as a bitmap. The load operation is very simple on the embedded side since it can directly access the flash unlike the FPGA. The slicing is the most complicated part for the embedded system. While the embedded system can make use of complicated data structures that the FPGA cannot make use, the its lack of power makes it difficult to make this an efficient operation.

3.5.3 Slice Program design

The slicing program went through two major iterations. The key difference between the designs was how the models would be loaded and read. The first design utilized linked lists, while the second design was a simple data stream.

First Program Design

The first iteration of the design used linked lists. The model was stored as a list of structs that contained the coordinates of the voxel. This has the benefit of only having to record the points that are there and not the points that are not. This is also easy to work with and easy to expand in the future. Furthermore, it is quicker to convert the mesh to a linked list instead of converting it to a to the full data stream

This design has a major fault, it is quite slow. To find all the intersection points a full search operation must be performed for each point. Even using an algorithm such as heap sort, searching is

incredible inefficient. It is possible that up to 8000 search operations will have to be performed for each slice. It is possible to always guarantee the best-case search time. Sorting the two objects in the same way does this. This means that we can be sure that the search will find each point as soon as possible and that the search can be ended early if that point is passed over. Even with these enhancements it is still too inefficient for use in this setting.

Final Program Design

A simpler approach to the slicing program needed to be created. The second solution relies on using a very simple data structure to store the object files. This is done by first reducing the amount of data that is stored. Instead of storing all the coordinates of the object each voxel is instead represent simply as a 1 or 0 to signify if it is present in the structure. This means that the data can be stored as a 1-bit data stream and can be loaded much quicker. Most importantly since we know that both objects contain all point in the same order it is possible to find all intersections without using a single sort or search function. In fact, if set up correctly the intersections can be found with a simple AND statement. Another benefit of this program is how easily it can be threaded. Since each slice is completely operation does not affect the previous data structures it is very easy to convert this program into a threaded one. This also mean that there are very few time where the threads will be waiting on each other for resources meaning the program can see massive improvements through threading. FA program flow chart can be seen in figure 21.

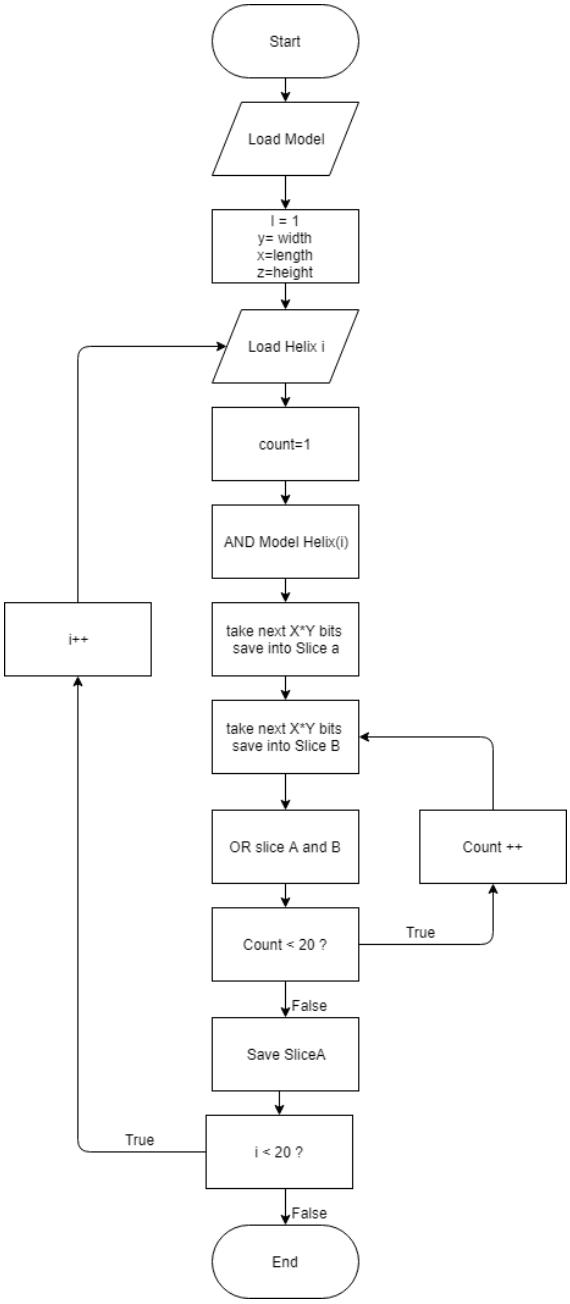

Figure 21: Program Flow Chart

As the chart shows, the majority of the program is done through a couple of simple loops. The first loops controls which slice is being created. There are twenty separate helix rotations, so this loop will need to run twenty times to create all the slices. This is the point where the threads are created in the multi-thread version. All the intersections are found taking each layer one at a time. The data

structure does not have any indication for end of layer, so instead the program determines where the layer ends by taking only a set number of bits. The layers are then OR together one at a time to create the final slice. After this the slices are ready to be converted to a bitmap. This is done by taking a set header string and adding it to the front of the saved slice. This string contains the header information for 20 by 20 1bit depth bitmap.

This makes a fast and reliable operation, but it does have some limitations. As the resolution of the objects grow the amount of memory used will grow faster, since all possible points are represented. This method also greatly limits the ability of the program to be expanded on, since things such as color cannot be represented when using only a 1-bit array. These limitations are necessary since speed is the most important aspect of this program

3.6 Lightcrafter

The images are displayed by a special kind of projector, known as a DLP projector. DLP projectors have the ability to project images at incredibly high frame rates since it uses lasers and a special mirror array know as a DMD to create the image. The DLP projector that was chosen for this project is the Lightcrafter evaluation module, since it was cost effective DLP projector option.

DLP projectors offer many advantages over the other options. The biggest being simplicity. The system can be treated as a normal projector or images can be loaded onto the projector for even faster frame rates. Getting the device to work with the embedded system was also clearly an option since it came with a well-documented API that could be loaded onto the system. It also provides more direct control of the images. Images did not have to be altered to be displayed, so it could use the basic image formats that are already in use by the computer and the projector would not distort them.

Other methods for the projection where also looked at. The biggest competitor to DLP was a pure laser solution. In theory the laser solution offers a clearer image and should cost less. However, the laser solution also would have been more complicated to implement and control. Furthermore, it would have been fragile, susceptible to slight shifts in the system. Finally, thanks to the generosity of Texas Instruments the DLP projector was able to be acquired at an incredibly cheap cost.

The Lightcrafter was used in 1bpp definition pattern mode. This means that it was able to display a series of 1bpp depth images at up to 4000 frames per second. This mode did have one major downside. The images had to be downloaded to the device before they could be displayed. The download process takes about 1 second per frame, which is far too slow to be able to display the images in real time as they are being created. This meant that goal of the project had to be changed from displaying the object in real time to displaying a clear steady image.

The Lightcrafter came with both GUI and an API as methods to control the system. The GUI had the capabilities to control every aspect of the system, but since it required a person to be actively controlling it a program that uses the API was written instead. The GUI still proved to be very useful for testing the system and settings, since it was quick and easy to use.

The API program that was written accomplished the intended goals. First off it was independent and able to be run on the embedded system without installing any additional libraries. It first established a connection to the Lightcrafter and then makes sure it has a solid connection. It then modifies the Lightcrafter setting so that the correct ones are being used. These settings include setting the display mode, bit depth, pattern number, and configuring the external trigger. It then loads the twenty slices into the Lightcrafter and waits for the external trigger. The external trigger is generated by the infrared encoder that is attached to the motor. This means that as the motor spins the Lightcrafter will change images to keep up with it. The light crafter was set to change at every falling edge of the encoder signal.

3.7 Hardware

For the swept helix approach to work, the sliced images need to be displayed onto a spinning helix shaped screen. This means that a motor system, a screen and a calibration tool all needed to be created for this project. A frame to hold all the pieces of the project together also needed to be created.

3.7.1 Screen

First the screen for the projection needed to be created. Several different methods to create the screen were looked at and compared.

One such method was using a wire frame and cloth screen. This method provided the easiest and cheapest implementation, however it came with several drastic limitations. The first limitation was the shape. For the image to be produced correctly the shape of the screen is very important. It is important to get the screen as close to exact as possible, and with the wire frame and cloth method it would be difficult to get anything to be exact. The cloth would also distort as the device spun. Finally, the screen would be fragile and vulnerable to being bent or otherwise damaged.

The other method, that was the one that was chosen, was to 3d print the helicoid screen. This could be costly since the screen is large for a 3d print so it takes up a large amount of material. It also would take a long time to print. The shape of the object also makes it tricky to print and can cause the print to have failures. However, the end product is a light and sturdy screen that that will maintain it shape.

3.7.2 Motor

To spin the screen a motor is needed. Two different kinds of motors where considered. The first option was a stepper motor. The stepper motor had the huge advantage of stopping after moving a

set distance. This would mean that when each slice was projected the screen would be stopped. Since the screen wouldn't be moving the image would have increased clarity and stability. The controller would also always have direct control of the position of the motor and would be able to know where the motor was at all time. This was considered the optimal solution; however, it came with some serious drawbacks. Stepper motors are far more difficult to control and provide a lot less power per watt. More importantly the stepper motor loses power the faster it is spinning [20]. A power to speed curve for a stepper motor can be seen in figure 22.

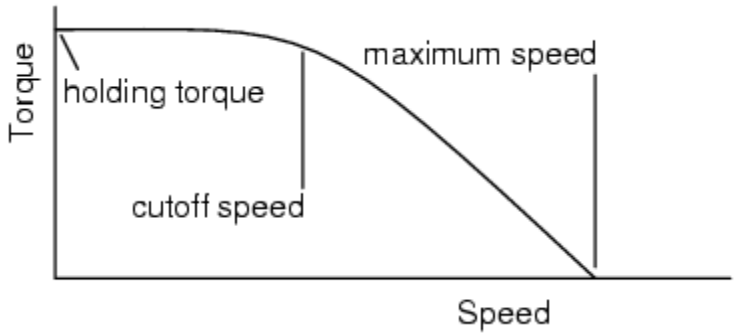

Figure 22: Stepper Motor torque curve [1]

Since our application would require at the minimum 1800 rpm it was not feasible for a stepper motor to be used. Instead a standard dc brushless motor was used. This provided the advantage of being able to get the need torque with very little power and easily being able to get up to the correct speed. It did come with the downside of requiring an additional encoder system to be developed so that the position of the motor could be controlled.

3.7.3 Encoder System

An encoder system for the motor needed to be developed so that the current position of the motor could be found. The encoder works by using an infrared sensor and an encoder wheel. The encoder wheel was a thin circle with two rows of holes cut out. The inside row had a hole for each position of the helicoid screen where display would occur. A total of forty holes where in this row, since

there were twenty positions and the shape were symmetrical so each half rotation went through all the positions. The outer row was the home row, it had two holes in it that represented the start positions. This ensured that the display could start at the right point each time it was turned on. The outline of the encoder wheel can be seen in figure 23.

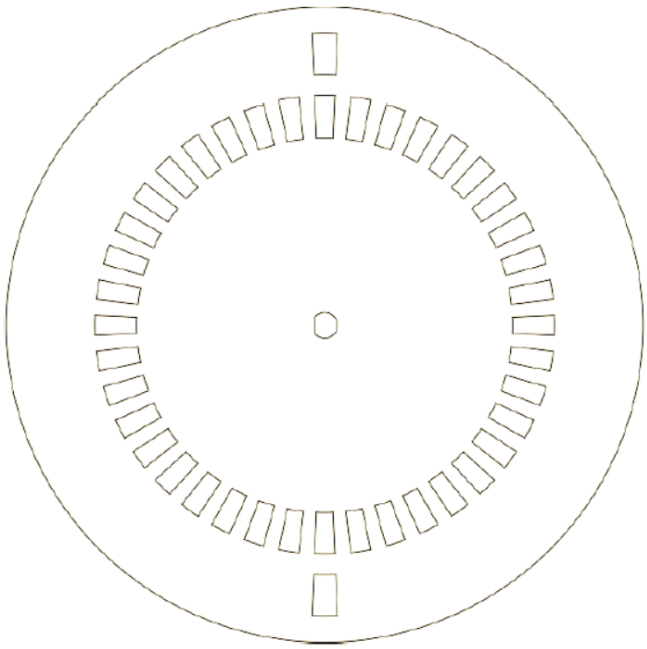

Figure 23: Encoder Wheel

The other part of the encoder was the sensor circuit. An infrared sensor circuit was designed a created. The system had an infrared LED on one side of the encoder wheel and an infrared sensor on the other side of the wheel. As the wheel spins it will allow the LED to shine through the holes when a new slice should be displayed. The sensor would then see this and cause the output to go high, creating a rising edge in the wave form. A second sensor was need so that the system could tell when it was at the home position. A schematic for the circuit can be seen in figure 24.

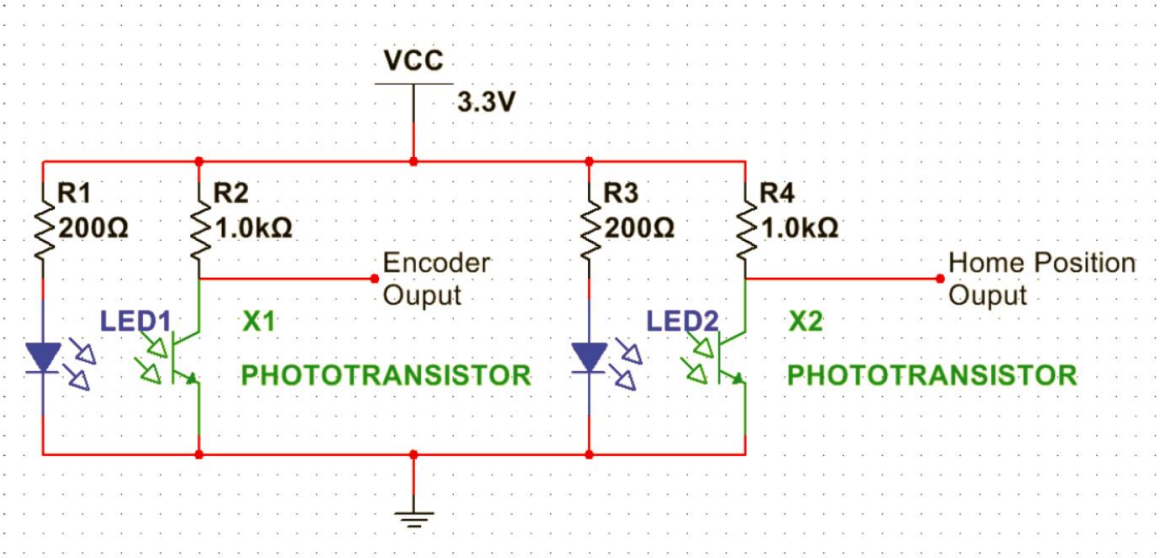

Figure 24: Encoder Circuit

3.7.4 Frame

Several parts in the project are position dependent. For the project to work consistently a frame for the system needed to be built. The screen, the motor, the encoder, and the projector are very particular about their positioning. A frame was designed and then built using punched metal. A housing for the encoder circuit was created by 3d printing and bolted onto the frame. The top of the frame holds the projector pointing down at the screen so that it can achieve maximum coverage of the screen. The encoder wheel sits below the screen and runs through the encoder circuit. An image of the built frame can be seen in figure 25.

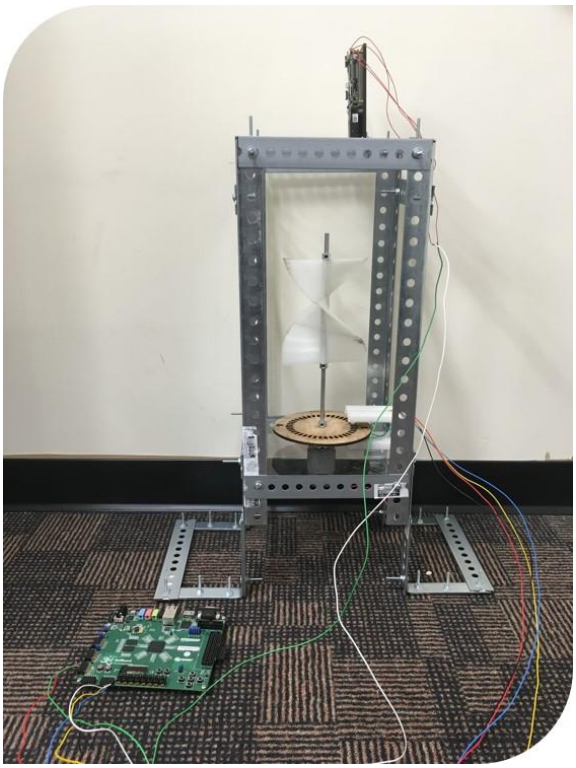

Figure 25: Built Frame

4 Discussion and Results

4.1 Program results

The program works as intended and produces slice images. The project was tested on a HummingBoard-i2ex system that was running an Ubuntu-16 Linux based system. This board features a dual core A9 processor. The program was run two different ways. The first way was with an unthreaded program. The second way was with the same program, but now threaded instead. The efficiency of the program was measured by seeing how many cpu cycles the program took to complete. This was chosen as a measurement, since it would give a direct comparison between the two and can easily be converted to time. Threading the program created a significantly more efficient program. The threaded program had a smaller variance and took on average about 36.5% less cycles to complete. A table of the different programs efficiencies is shown it table 1.

Table 1: Program efficencies

	Unthreaded Program										Threaded Program					
	Trial 1		Trial 2		Trial 3		TTrial 4		tTrial 5		Trial 6		Trial 7		TriaTl 8	
Frame	cycles / frame	total cycles	cycles / frame	total cycles	cycles / frame	total cycles	cycles / frame	total cycles	cycles / frame	total cycles	cycles / frame	total cycles	cycles / frame	total cycles	cycles / frame	total cycles
1	316	316	317	317	423	423	273	273	316	316	203	203	159	159	205	205
2	249	565	323	640	285	708	297	570	322	638	182	385	207	366	168	373
3	268	833	317	957	324	1032	236	806	303	941	151	536	207	573	178	551
4	323	1156	320	1277	313	1345	322	1128	318	1259	207	743	206	779	191	742
5	319	1475	316	1593	322	1667	317	1445	325	1584	202	945	208	987	208	950
6	338	1813	286	1879	314	1981	267	1712	311	1895	171	1116	216	1203	204	1154
7	260	2073	246	2125	316	2297	321	2033	315	2210	231	1347	183	1386	232	1386
8	301	2374	367	2492	268	2565	322	2355	202	2412	148	1495	201	1587	205	1591
9	326	2700	318	2810	309	2874	277	2632	361	2773	148	1643	205	1792	198	1789
10	358	3058	311	3121	291	3165	281	2913	314	3087	205	1848	129	1921	206	1995
11	319	3377	317	3438	361	3526	332	3245	260	3347	208	2056	204	2125	166	2161
12	322	3699	312	3750	244	3770	267	3512	315	3662	180	2236	200	2325	201	2362
13	310	4009	318	4068	314	4084	263	3775	263	3925	235	2471	201	2526	177	2539
14	314	4323	320	4388	316	4400	325	4100	231	4156	203	2674	208	2734	206	2745
15	322	4645	363	4751	318	4718	289	4389	248	4404	203	2877	159	2893	202	2947
16	232	4877	321	5072	325	5043	321	4710	279	4683	204	3081	148	3041	201	3148
17	317	5194	314	5386	315	5358	298	5008	322	5005	166	3247	201	3242	183	3331
18	338	5532	330	5716	318	5676	278	5286	317	5322	199	3446	204	3446	202	3533
19	278	5810	314	6030	262	5938	336	5622	262	5584	177	3623	215	3661	207	3740
20	312	6122	247	6277	313	6251	332	5954	321	5905	201	3824	205	3866	172	3912

There are eight sets of data present in the table. The first 5 sets represent different measurements from three separate runs of the unthreaded program. The last three sets of data represent the run times of the threaded program. Each data set is composed of two columns. The first column represents number of cycles to complete each slice. The second column is a running total for of cycles for the entire duration of the program.

It was found that the mean number of cycles for the unthread program to complete was 305.09 cycles and had a standard deviation of about 33.53. In contrast the threaded program had a mean of

193.36 cycles and a standard deviation of about 22.04. This shows that not only was the threaded program much faster it was much closer together.

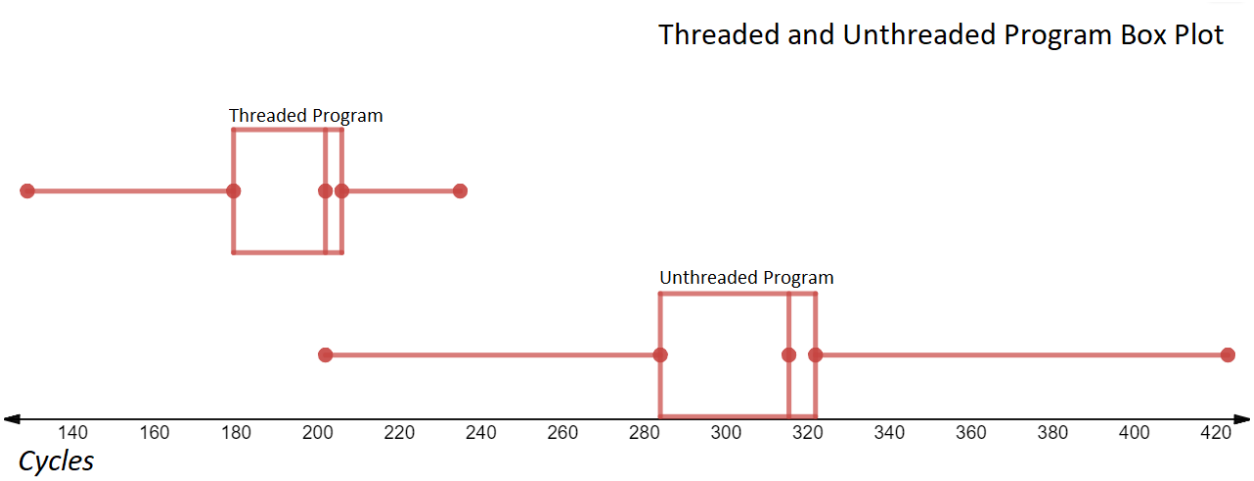

Figure 26: Box Plot for Threaded and Unthreaded Program

This data is also represented as a box and whisker plot is figure 26. The lower box plot represents the unthreaded program. The higher box plot represents the threaded version of the program. These data sets are compared by cycles to complete each slice, which can be seen as the x axis. Each plot represents all the trials done with the version of the program. This means that the unthreaded plot is composed of five trials, each with twenty data points, for a total of one hundred data points and the threaded plot is composed of three trials, each with twenty data points, for a total of sixty data points

This data shows that the system is reliable and efficient. It also shows that we were meet the original time goal. The original goal was to have all twenty slices created and loaded in under one twentieth second or 50 miliseconds. This goal was chosen so that it would be possible to create and load the slices in real time. To achieve this goal all slices had to be created in less than a quarter of the goal time, about 1.25 milliseconds since the other programs would need time to function as well. Even

though the Hummingboard has a dual core 1 Ghz processor it can be run at 1.2Ghz if need be [10]. This means that the program would need to run in under fifteen thousand cycles. When running at 1.2 GHZ and factoring in worst case scenarios the threaded program stays under these number by a significant amount.

Unfortunately, even though the slicing is under the goal time, the other functions are not. Namely the transfer of the images to the Lightcrafter projector takes about a half a second each. There was no way to circumvent this issue without drastic changes to the design. The voxelization of the model also takes far too long to complete, sometimes taking multiple seconds. This means that the system will not be able to achieve real time video with this current design.

4.2 Comparison to FPGA Hybrid

The embedded only solution was then compared directly to the FPGA/Embedded solution. The two different solutions share many different parts so only the parts that were different between the two systems were looked at. This consist of mostly the slicing component. For the FPGA/Embedded solution this starts when the embedded system begins writing the model into the block memory and ends when it is saved as a Bitmap. For the embedded solution this begins when it starts the slicing program, and when the program finishes and saves the slices.

The FPGA has a slower clock speed than the embedded system, so comparing cycles is does not provide useful information. For realistic comparison the data was converted from cycles to time for both the solutions.

This is more complicated for the FPGA/Embedded since the two systems need use different clock speeds. To begin the cycle amount for the FPGA and the embedded system are separated apart. Then each system is converted from cycles to µs based on their clock speeds. The FPGA has a maximum clock speed of 667 Mhz [24] and the embedded system can achieve 1.2 Ghz. These times are then

added back together to get the total time of the system. The breakdown of times can be seen in Table 2.

Table 2: FPGA/Embedded Time Breakdown

Frame	Total Cycles	Embedded cycles	FPGA cycles	Embedded Time (us)	FPGA Time (us)	Total Time (us)
1	1988	1941	47	1617.5	70.46	1687.96
2	2035	1941	94	1617.5	140.93	1758.43
3	2082	1941	141	1617.5	211.39	1828.89
4	2129	1941	188	1617.5	281.86	1899.36
5	2176	1941	235	1617.5	352.32	1969.82
6	2223	1941	282	1617.5	422.79	2040.29
7	2270	1941	329	1617.5	493.25	2110.75
8	2317	1941	376	1617.5	563.72	2181.22
9	2364	1941	423	1617.5	634.18	2251.68
10	2411	1941	470	1617.5	704.65	2322.15
11	2458	1941	517	1617.5	775.11	2392.61
12	2505	1941	564	1617.5	845.58	2463.08
13	2552	1941	611	1617.5	916.04	2533.54
14	2599	1941	658	1617.5	986.51	2604.01
15	2646	1941	705	1617.5	1056.97	2674.47
16	2693	1941	752	1617.5	1127.44	2744.94
17	2740	1941	799	1617.5	1197.90	2815.40
18	2787	1941	846	1617.5	1268.37	2885.87
19	2834	1941	893	1617.5	1338.83	2956.33
20	3037	2057	980	1714.17	1469.27	3183.43

As can be seen in the table, the embedded part of the solution has a large amount of upfront time that slows the whole system down. The individual slices on the FPGA are very quick, far quicker than the slices on the embedded system. It is also worth noting that most of the slices take the same amount of time (47 cycles) on the FPGA. This is because since the FPGA uses physical logic the slices always take the same amount of cycles to complete.

Table 3: Slicing Method Time Comparison

Frame	FPGA	Unthreaded 1	Unthreaded 2	Unthreaded 3	Unthreaded 4	Unthreaded 5	Threaded 1	Threaded 2	Threaded 3
	(us)	(us)	(us)	(us)	(us)	(us)	(us)	(us)	(us)
1	1687.96	263.33	219.44	352.50	352.50	263.33	169.17	132.50	170.83
2	1758.43	470.83	392.36	590.00	590.00	531.67	320.83	305.00	310.83
3	1828.89	694.17	578.47	860.00	860.00	784.17	446.67	477.50	459.17
4	1899.36	963.33	802.78	1120.83	1120.83	1049.17	619.17	649.17	618.33
5	1969.82	1229.17	1024.31	1389.17	1389.17	1320.00	787.50	822.50	791.67
6	2040.29	1510.83	1259.03	1650.83	1650.83	1579.17	930.00	1002.50	961.67
7	2110.75	1727.50	1439.58	1914.17	1914.17	1841.67	1122.50	1155.00	1155.00
8	2181.22	1978.33	1648.61	2137.50	2137.50	2010.00	1245.83	1322.50	1325.83
9	2251.68	2250.00	1875.00	2395.00	2395.00	2310.83	1369.17	1493.33	1490.83
10	2322.15	2548.33	2123.61	2637.50	2637.50	2572.50	1540.00	1600.83	1662.50
11	2392.61	2814.17	2345.14	2938.33	2938.33	2789.17	1713.33	1770.83	1800.83
12	2463.08	3082.50	2568.75	3141.67	3141.67	3051.67	1863.33	1937.50	1968.33
13	2533.54	3340.83	2784.03	3403.33	3403.33	3270.83	2059.17	2105.00	2115.83
14	2604.01	3602.50	3002.08	3666.67	3666.67	3463.33	2228.33	2278.33	2287.50
15	2674.47	3870.83	3225.69	3931.67	3931.67	3670.00	2397.50	2410.83	2455.83
16	2744.94	4064.17	3386.81	4202.50	4202.50	3902.50	2567.50	2534.17	2623.33
17	2815.40	4328.33	3606.94	4465.00	4465.00	4170.83	2705.83	2701.67	2775.83
18	2885.87	4610.00	3841.67	4730.00	4730.00	4435.00	2871.67	2871.67	2944.17
19	2956.33	4841.67	4034.72	4948.33	4948.33	4653.33	3019.17	3050.83	3116.67
20	3183.43	5101.67	4251.39	5209.17	5209.17	4920.83	3186.67	3221.67	3260.00

Table 3 shows the total times of all the different trials. It is clear from this table that the FPGA/embedded solution is the fastest option, but only by a small margin. The FPGA/Embedded solution is 55% faster than the unthreaded program. However, it is only 1.2% faster than the threaded program. This is only marginally better.

In figure 27 the data from Table 3 is graphed. For the graph the average times for each method are used. This graph clearly shows the large initial startup time of the FPGA, but its comparatively lower slope. It also clearly shows that on average the FPGA/embedded solution becomes

faster than the unthreaded program after about 10 different slices. The FPGA also becomes faster than the threaded program after about 18 slices.

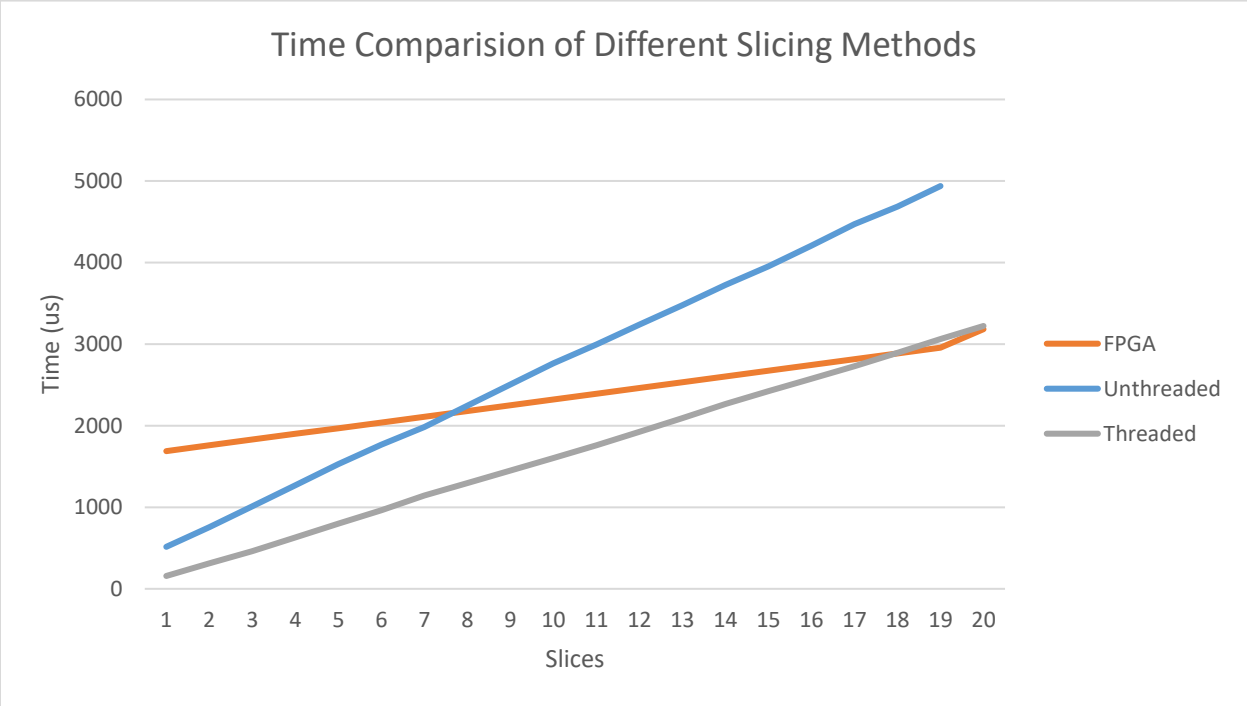

Figure 27: Graph of Time for Different Slicing Methods

5 Conclusion

This project was able to create a functional volumetric display. The display utilized an FPGA and an embedded system to display a 3d computer model. This was tested against a second solution that did the same work without the use of an FPGA, only using an embedded system. The embedded solution was shown to have be more efficient for smaller amounts of slices, but less effective when working with the full object. The purely embedded solution also provides a more modular and expandable design that can be improved upon in the future.

The project was able to develop several different components. The development of voxelization and slicing algorithms allowed for a complete simulation of the display to be created. FPGA logic was developed and tested that successfully was able to generate the correct slice images from a 3d model. The embedded system was proven to be able to do the slicing of the images correctly. A projector control program was also successfully created for the Lightcrafter. A projection control system was designed and created that included all the mechanical components necessary to accurately display the object. It included the helicoid screen and the encoder module that synchronized the motor to the projector, as well as a frame designed to hold everything together.

6 Future Work

There is much to explore in the field of volumetric displays and much that can still be improved on in this design. This project showed that a FPGA-embedded combination-based solution was not only viable, but preferable to that of a purely embedded solution. This means that there is still lots of promising research that can be done in FPGA display field. A variation of this project where the FPGA can connect directly to the projector system would greatly improve the throughput of the device, and make an FPGA based solution much more viable. Exploring the possibility of creating a higher resolution images for slicing can lead to huge breakthroughs in the field. Alternatively, another group would be able to expand on the work that has been done and potentially create a volumetric video device instead of one that can only display a still object. To do that a method that has much higher throughput would have to be developed, but this project could serve as the perfect starting point.

REFERENCES

[1]Engineering Stack Exchange. (2018). *Reading stepper motor datasheets to get torque and speed*. [online] Available at: https://engineering.stackexchange.com/questions/2802/reading-stepper-motor-datasheets-to-get-torque-and-speed [Accessed 1 Sep. 2017].

[2]M. David, *Cube model and helicoid surface*. 2017.

[3]M. David, *Intersecting cube curves*. 2017.

[4]M. David, "Volumetric Display", *The Critical Technologist*, 2017. [Online]. Available: https://thecriticaltechnologist.wordpress.com/audiovisual/. [Accessed: 25- Apr- 2017]

[5]"DLP® LightCrafter™ Evaluation Module," *DLPLIGHTCRAFTER DLP® LightCrafter™ Evaluation Module | TI.com*. [Online]. Available: http://www.ti.com/tool/DLPLIGHTCRAFTER. [Accessed: 25-Apr-2017].

[6]DLP, *Lightcrafter EVM*, 2017

[7]D. Rosen, *Triangle surface to voxel*. 2017.

[8]E. Downing, L. Hesselink, J. Ralston, and R. Macfarlane, "A Three-Color, Solid-State, Three-Dimensional Display," *Science*, vol. 273, no. 5279, pp. 1185–1189, 1996.

[9]H. Aitkenhead, A. (2018). *vigente/gerardus*. [online] GitHub. Available at: https://github.com/vigente/gerardus#svn%2Ftrunk%2Fmatlab%2FThirdPartyToolbox%2FMeshVoxelisation [Accessed 1 Sep. 2018].

[10]"HummingBoard," *SolidRun*. [Online]. Available: https://www.solid-run.com/nxp-family/hummingboard/. [Accessed: 25-Apr-2017].

[11]HummingBoard, *HummingBoard-I2EX and Components*, 2017

[12]Kozioo, *Male Head Mesh*. 2017.

[13]S. Luther, *DLP Projection*. 2017.

[14]S. Luther, "DLP Projectors", *pctechguide.com*, 2017. [Online]. Available: https://www.pctechguide.com/projectors/dlp-projectors. [Accessed: 25- Apr- 2017].

[15]M. Okamoto, K. Komatsu, Y. Kajiki, and E. Shimizu, "Three-dimensional display with volume/space expansion," *Three-Dimensional TV, Video, and Display II*, 2003.

[16]Min, P. (2018). *binvox 3D mesh voxelizer, keywords: voxelization, voxelisation, 3D model*. [online] Patrickmin.com. Available at: http://www.patrickmin.com/binvox/ [Accessed 1 Sep. 2017].

[17]S. Patil and B. Ravi, "Voxel-based Representation, Display and Thickness Analysis of Intricate Shapes", *Ninth International Conference on Computer Aided Design and Computer Graphics (CAD-CG'05)*.

[18]H. Refai, "Static Volumetric Three-Dimensional Display," *Journal of Display Technology*, vol. 5, no. 10, pp. 391–397, 2009.

[19]H. H. Refai, "3D Images Using CSpace Display for Air Traffic Control Applications," *Journal of Display Technology*, vol. 7, no. 4, pp. 186–192, 2011.

[20]L. Sawalha, M. P. Tull, M. B. Gately, J. J. Sluss, M. Yeary, and R. D. Barnes, "A Large 3D Swept-Volume Video Display," *Journal of Display Technology*, vol. 8, no. 5, pp. 256–268, 2012.

[21]R. Wright and B. Lipchak, *OpenGL superbible*, 1st ed. Indianapolis, Ind.: SAMS, 2005.

[22]W. Xu, J. Liu, M. Cai, and M. Wang, "A camera network for the voxel data acquiring of the Three-Dimensional Swept Volume Display," *2012 IEEE International Conference on Information Science and Technology*, 2012.

[23]J. Yue, F. Jiao and C. Shen, "The Key Technologies of Helix Rotating Screen Volumetric-Swept Display System", *2009 First International Conference on Information Science and Engineering*, 2009.

[24]ZedBoard," *ZedBoard | Zedboard*. [Online]. Available: http://zedboard.org/content/zedboard-0. [Accessed: 25-Apr-2017].

[25]ZedBoard, *Zedboard Block Diagram*, 2017

[26]ZedBoard, *Zedboard, Front side*, 2017