ECE 385

FA 2020

Prof. Chushan Li

Final Project Report
FPGA-based 3D Graphics Renderer

Xie Tian 3180111631
Guan Zimu 3180111630

Date: Jan 01, 2021

Table of Contents

1 Introduction

In this project, we desgin and implement a FPGA-based 3D graphics renderer, which contains a basic 3D graphics pipeline to render an object, including model, view, projection transformation, rasterization and certain further functionality. This project uses on-chip memory to store the raw information of the 3D objects, projected triangles data in every frame and frame buffers. The rendered 3D objected will then be displayed on the monitor through VGA port mentioned in Lab 8, and we can use keyboard input to control the movement of the object, e.g. moving horizontally, vertically, back and towards, rotating in three directions. Except for the NIOS-II CPU that controls the USB keyboard, all the components are implemented purely in SystemVerilog. It marks a successful end of our ECE 385 course, where we accumulates certain valuable experiences for project development.

1.1 A Brief Introduction to Graphic Rendering Pipeline

A 3D graphics pipeline is a process to display 3D objects on a 2D scene. In computer, an 3D object is usually stored as a bunch of triangles because their "discrete" property is easy for computer to handle with. To be specific, these data are usually 3D coordinates of vertices, sets of points that can form triangle surfaces, normal vectors of surfaces and so on. However, what we see on the display is just 2D images, so we need to project 3D objects onto a 2D scene, according to some rules (usually perspective relations). This step is called model, view, projection transformation, or MVP transformation. Obviously, it is not enough, because the resolution of a display is not infinity, it is composed of bunch of pixels. Therefore, we should convert the nearly continuous object data into pixels, which is called rasterization. During the rasterization, we would have some algorithm, usually Bresenham algorithm, to draw lines. Besides, we probably need to compute and display the illumination of the object according to its properties. This is the so called shading. In real-time rendering, speed is significant, so engineers use texture images instead of real physical properties to give objects' color (The latter method is used in off-line rendering, usually done by ray tracing). After the calculation of one triangle is finished, the depth of it should be consider because what we would see is the nearest object. That's why we use a z-buffer to store the depth of the pixel. When we done all these step for all triangles and write colors into the frame buffer, an object can be displayed on the screen. Actually, there are numerous other technologies in 3D graphics pipeline, for example, perspective correction, homogeneous space clipping, anti-aliasing, etc. Due to the limit time and complexity of hardware design, we are not going to implement all these functionalities.

2 Project Description

2.1 Overview of the Circuit

2.1.1 Purpose of the Circuit

The purpose of the circuit is to load information of a provided 3D object consisting of a sequence of triangles, project those 3D triangles into 2D plane with the scale of the monitor, finally display it on the monitor with a frame buffer. During rendering, the user can use a USB keyboard as controller to change the position of the rendered object and rotate it in accordance to 3 axises. The circuit can be roughly divided into two parts, first of which is a NOIS-II SoC that serves as the driver of the keyboard to control

the display configurations. The other part, also the major rendering modules, are purely written in SystemVerilg, which contains mostly the parallel computation to accelerate the large-scale calculation of the projection and coordinates. The entire circuit is organized in the form as a graphic rendering pipeline introduced in the first section, and for design details, refer to the Implementation Section.

2.1.2 Design Features

- Able to use NIOS II as a CPU to give the information of the rendered object to the hardware design.
- Able to calculate the model, view, projection matrices according to the given location and perspective of camera and apply matrix multiplication to get the final MVP transformation matrix in hardware design.
- Able to rasterize the triangle (basically draw lines) using line drawing Bresenham algorithm in hardware design.
- Able to do simple clipping of triangles when they are out of the screen.
- Able to use frame buffer and output the rendered image to the display through VGA port in hardware design.
- Able to control complex data flow during rendering through multiple finite state machines in hardware design.
- Able to use USB keyboard to control the view of the camera in NIOS II.
- Able to read .obj file, analyse it and generate triangles data in NIOS II (We program a Python script)
- Able to apply some parallel calculation features in hardware design. (e.g. matrix multiplication)

2.2 General Flow of the Circuit

2.2.1 Inputs/Outputs Description

Generally speaking, the input of our system is the key board input of object's motion, the output of our system is the image of a moving 3D object displayed on the monitor.

Specifically, inputs and outputs of the top level module are these:

- CLOCK_50: input, 50MHz clock
- KEY: input, on-board key signal, used for reset
- SW: input, on-board switch signal, used for debugging
- LEDR: output, signal controls on-board LEDR, used for debugging
- HEX0, HEX1: output, signal controls on-board HEX display which shows the keycode

- **VGA Interface**

- VGA_R: output, VGA Red
- VGA_G: output, VGA Green
- VGA_B: output, VGA Blue
- VGA_CLK: output, VGA Clock
- VGA_SYNC_N: output, VGA Sync signal
- VGA_BLANK_N: output, VGA Blank signal
- VGA_VS: output, VGA vertical sync signal
- VGA_HS: output, VGA horizontal sync signal

- **CY7C67200 Interface**

- OTG_DATA: inout , CY7C67200 Data bus 16 Bits
- OTG_ADDR: output, CY7C67200 Address 2 Bits
- OTG_CS_N: output, CY7C67200 Chip Select
- OTG_RD_N: output, CY7C67200 Write
- OTG_WR_N: output, CY7C67200 Read
- OTG_RST_N: output, CY7C67200 Reset
- OTG_INT: input , CY7C67200 Interrupt

- **SDRAM Interface for Nios II Software**

- DRAM_ADDR: output, SDRAM Address 13 Bits
- DRAM_DQ: inout , SDRAM Data 32 Bits
- DRAM_BA: output, SDRAM Bank Address 2 Bits
- DRAM_DQM: output, SDRAM Data Mast 4 Bits
- DRAM_RAS_N: output, SDRAM Row Address Strobe
- DRAM_CAS_N: output, SDRAM Column Address Strobe
- DRAM_CKE: output, SDRAM Clock Enable
- DRAM_WE_N: output, SDRAM Write Enable
- DRAM_CS_N: output, SDRAM Chip Select
- DRAM_CLK: output, SDRAM Clock

2.2.2 How Are the Inputs/Outputs Processed

This would be introduce in the overview of Pipeline Design.

2.2.3 Expected Outputs Shown

The expected output is the rendered 3D objected on the monitor through VGA port, which is written to on-chip memory during compile and program. The rendered 3D object consists of a sequence of projected triangles, and we can use USB keyboard as outer controller to control the real-time position, movement and the rotation of the object. To be more specific, upward and downward control the scale of the object (far or close); leftward and rightward control the horizontal movement of the object; "Z" and "X" control the vertical movement of the object; "Q, E", "W, S" and "A, D" control the rotation of the object with respect to the 3 axises. When the movement of the rendered object is to be out of the boundary of the screen, the overflow triangle will be clipped, which would cause the object to be "truncated". For the smoothness of the keyboard controller, we introduce the angular velocity and angular acceleration of the movement/rotation in design.

3 Design Procedure

3.1 Overview of Pipeline Design

The foundation of this project is the graphics rendering pipeline, as we briefly introduced before. Different from the general rendering pipeline, our design made modifications to make use of the advantage of FPGA and simplifications due to the limit time. Our pipeline design contains 3 stages.

1. **Application**: Motions of camera and object changing according to user's control are processed in this stage. Information generated in this stage would be used for Vertex Processing.
2. **Vertex Processing**: 3D vertexes of triangles are processed in this stage, including model, view, projection transformation and clipping. Projected triangles (2D vertexes in screen space) would be used for Rasterization.
3. **Rasterization**: Projected triangles would be drawn on the screen (rasterized) in this stage.

3.2 High Level Design & Consideration

This section include main objectives and some consideration of the project.

- **Motion Control**: We use a NIOS II system as a USB interface, so that user could control the motion of the object using USB keyboard. After getting the keycode from NIOS II system, a display controller module would generate position and angle information of the object, than pass to the render processor.
- **Render Processor**: Main processor, triangles are processed then projected and drawn in this part. Finally the pixel data (pixels of line) would be written into frame buffer.
- **Frame buffer**: To avoid flashing of the picture, we use 2-buffer and ping-pong strategy to display. During displaying, one buffer is updating while the other is

read by VGA display logic. Because we only need 1 bit to represent one pixel (whether is a line or not), the storage would not be so big. Therefore we choose on-chip memory to store 2 frame buffers. Dual port is used so that read and write can be done in one cycle.

- **Triangle Storage**: We use a list to store the original triangle data (3D coordinates of vertexes), which is initialized at the initialization of the processor, read-only after initialized. We use a fifo to store projected triangles (2D coordinates of vertexes in screen space) for each frame. Projected triangles would be pushed into triangle fifo then popped out for rasterize in each frame. Both list and fifo are stored in on-chip memory due to its convenience.
- **Real Number Calculation**: We use fixed point number instead of float point number to represent real number in our design. Though fixed point number is not accurate enough compared with floating point number, it is easier to handle with and is faster to calculate. Because speed is more important than accuracy in our design, we decided to use fixed-point number.
- **Parallelism**: We make use of the parallelism of FPGA mainly in render processor to accelerate the rendering. For Rasterization, because the storage of the frame buffer does not support write simultaneously to one pixel (otherwise we need to consider parallel programming technology such as thread lock, which is beyond our current ability), we decide to do this serially. However, for projection, we could compute matrix-matrix/matrix-vector multiplications and other computation simultaneously. Besides, projection calculation could be done when the frame is being clear. Projected triangle would be firstly pushed into triangle fifo then popped out for rasterizing when clear is done.

3.3 Research/Background Study

To realized a simple 3D renderer, members of our team studied basic knowledge in computer graphics, mainly in on-line course

- GAMES101:Introduction to Computer Graphics, taught by prof.Lingqi Yan

We also read some existing project on github to get some idea (though our final design is quite different from them)

- FPGA-3D-Renderer-ECE385, Kevin Palani
- Graphics-Renderer, Ishaan Patel

Besides, we read a Diploma Thesis Report by Fotis Pegios to get some idea of the pipeline design

- FPGA-based 3D Graphics Pipeline and Advanced Rendering Effects, Fotis Pegios

Finally, we researched online about fixed point/floating point calculation, trigonometric function implementation on FPGA and other details. References are scattered, we ignore them here.

4 Implementation

This section would describe the detailed implementation of the renderer, including algorithm implementation, state machines, block diagrams, etc.

4.1 Top Level Block Diagram

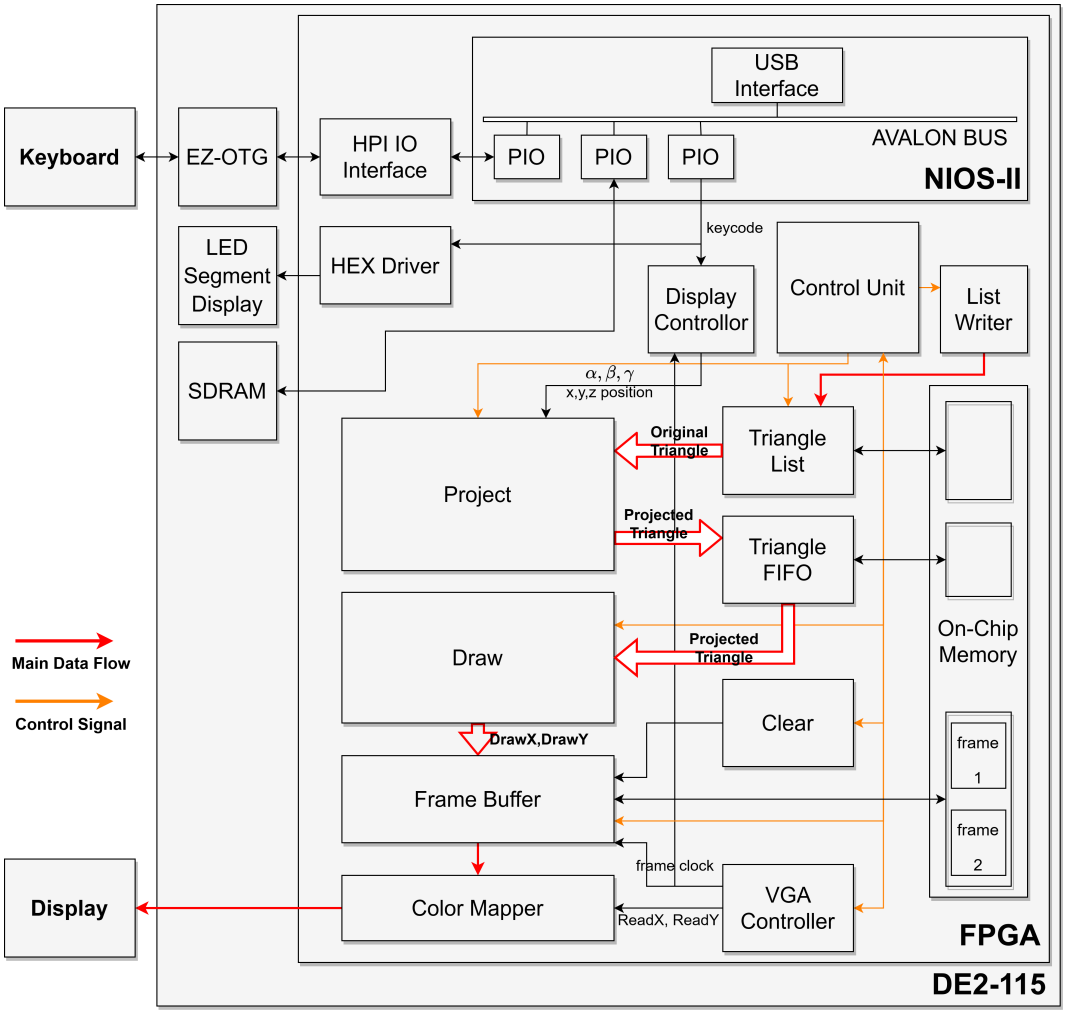

FIGURE 1. Top level diagram

We would introduce each component according to the main data flow.

4.2 Control Unit

We first introduce the control unit to illustrate what the renderer done in every frame. The control unit is a state machine control most of the signal of the renderer.

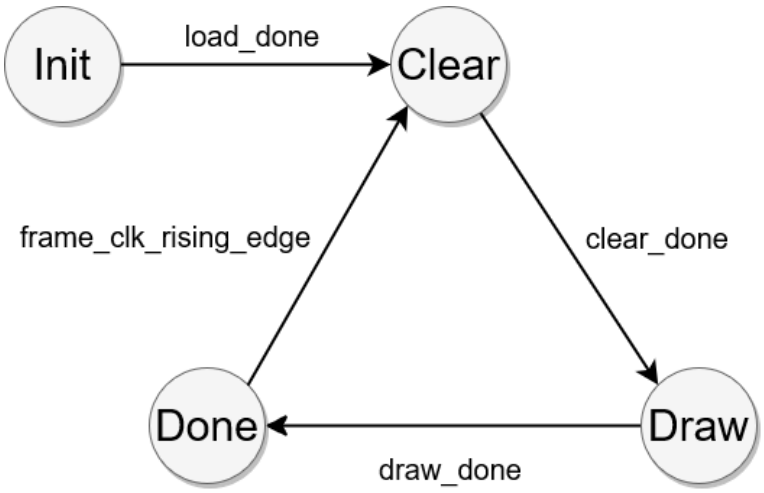

FIGURE 2. State machine diagram of the control unit

As we can see in the state machine diagram, the control unit has 4 states, **Init**,**Clear**, **Draw**, **Done**. Details are shown below.

- **Init**: The machine would be in this state only when the processor is initialized. In this state, object model (3D vertexes data) would be load into triangle list by triangle writer. Therefore, load_obj is high in this state.
- **Clear**: At the beginning of every frame, the machine needs to clear the previous frame so that new triangles displayed in current frame would not be drawn on the old triangles. clear_start would be high to trigger the clear module to clear the frame. draw_data would be 0 to clear the frame. Meanwhile, to make use of the parallelism of FPGA, vertexes processing mainly projection would also be triggered by the machine (proj_start would be high). So that clear and vertexes processing would be done simultaneously in this state.
- **Draw**: After all triangles are projected (pushed into triangle fifo) and frame is cleared, the processor starts to draw all triangles into frame buffer. draw_start would be high to trigger the draw module. draw_data would be 1 to indicate the line pixel.
- **Done**: After all procedure is done, the machine would halt until next frame is coming.

4.3 List Writer

This module writes original triangle data (3D coordinates of vertexes) into triangle list. The module would write only when the processor is initialized. By default, we write a cube into the list.

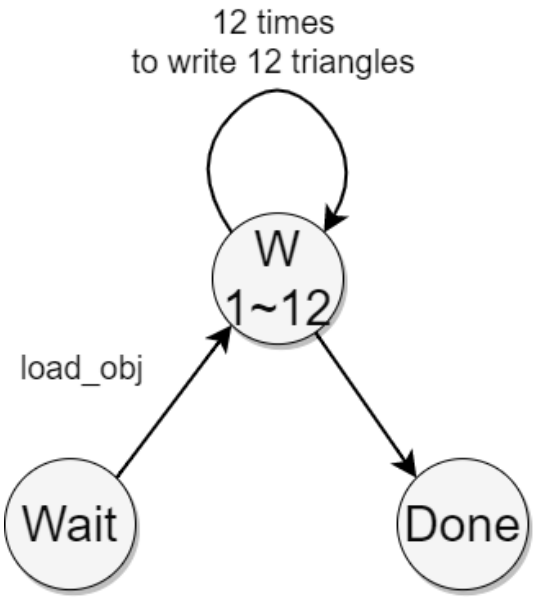

FIGURE 3. State machine diagram of the list writer

As we can see in the state machine diagram, the control unit has 14 states, **Wait**,**W1-12**, **Done**. Details are shown below.

- **Wait**: Wait to write until triggered by load_obj.
- **W1-12**: Write 12 triangles of a cube into triangle list.
- **Done**: Done.

4.4 Triangle List

This module stores original triangle data using on-chip memory. It contains a module called triangle list ram which would be automatically convert to block ram by Quartus II compiler. The storage of the list can be customized by setting the parameter. This module has 2 ways to use.

- **way1**: write triangles into list then read
- **way2**: initial on-chip memory by txt file while being compiled then read

Different way needs user comment/uncomment different lines. We use a register **num** to record the number of elements in the list. While the read pointer reach the top of the list (using **num** to justify), it would come back to the bottom of the list waiting for next read.

4.5 Display Controller

This module control the motion of the object and camera according to the keycode passed by the NIOS II system. It updates the motion every frame (not every cycle). To be specific, x,y,z position and α, β, γ , angles rotation along x,y,z axis (of the object's

local coordinates system), which would be passing to project module

Key ”up”,”down”,”left”,”right” would let the object go back, ahead, right, left perspectively. Because motions is relative, we let this be the motion of camera by invert the change of position.

Besides, we add angular acceleration to the rotation of the object. When press ”w”,”s”,”a”,”d”,”q” or ”e” on keyboard, the angular velocity of the object would increase or decrease until reaching the maximum magnitude. When the key is released, a friction would decrease the magnitude of the angular velocity until it becomes 0. So that the object could rotate smoothly and show some ”inertia” feature.

4.6 Project

This module is a rather complex module, containing a state machine and 2 nested sub module. Original triangle data (3D coordinates of vertexes) would be transformed using model, view, projection matrix calculated in sub model, then clipped if it is out of the bound of the screen. After that, projected triangle (2D coordinates of vertexes in screen space) would be written into triangle fifo. Except the state machine, all calculation are done without timing, i.e. calculated simultaneously. This can highly accelerate the procedure, though the disconvergence of timing would cause unstable of the system.

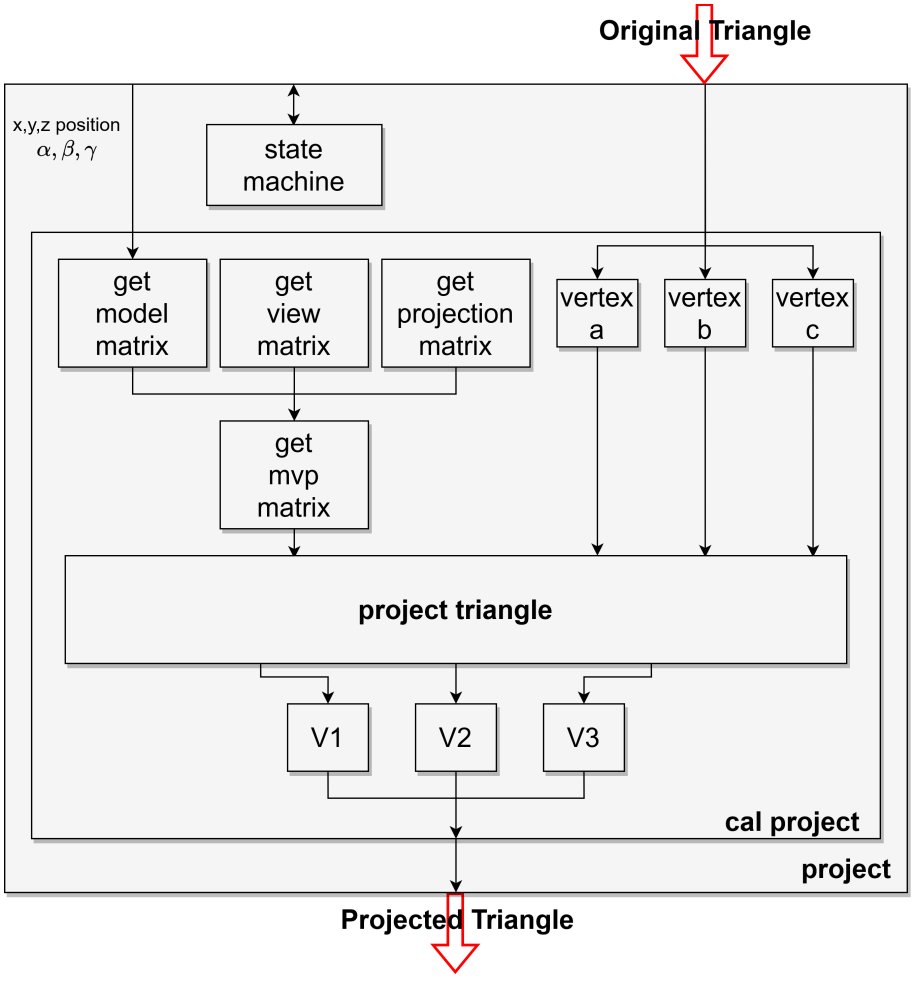

FIGURE 4. Project module diagram

4.6.1 State Machine of Project Module

The state machine control the read of triangle list and write of triangle fifo.

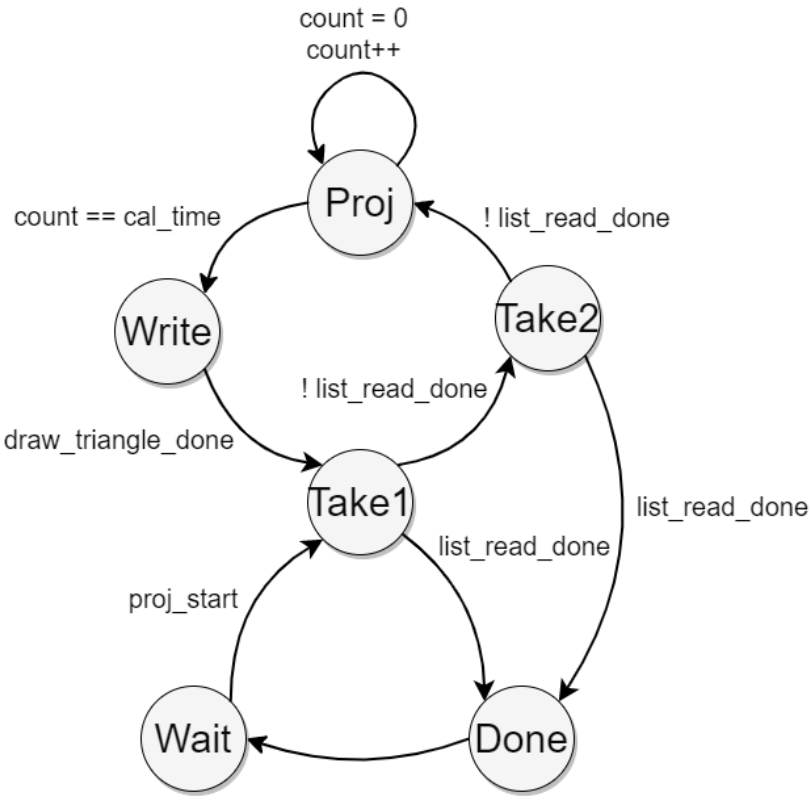

FIGURE 5. State machine diagram of project module

As we can see in the state machine diagram, the control unit has 6 states, **Wait, Take1, Take2, Proj, Write, Done**. Details are shown below.

- **Wait**: Wait to project triangles until triggered by proj_start.
- **Take1**: Trigger to take original triangles from triangle list. Read enable signal of triangle list is high in this state. If all triangles in triangle list are processed, end projection.
- **Take2**: Take original triangles from triangle list. If all triangles in triangle list are processed, end projection.
- **Proj**: Calculate projection (project vertexes). It wait for several cycles depend on the cal_time so that all calculation are done and the result is stable. We use a counter to implement this. Through our test, when cal_time is 4, i.e. machine waiting for 4 cycles, the result would be stable.
- **Write**: Write the projected triangle into triangle fifo. Write enable signal is high in this state.
- **Done**: Done. proj_done is high in this state.

4.6.2 cal_projection & proj_triangle

cal_projection.sv is the second highest module, mainly computing the mvp matrix then do the projection. It will first compute the model matrix of the object, view matrix of the camera, projection matrix that projects the object to $[-1, 1]^3$ canonical cube by instantiating get_model_matrix.sv, get_view_matrix.sv and get_projection_matrix. Then it instantiates get_mvp_model.sv to calculate the mvp matrix.

The lowest level module is project_triangle.sv. project_triangle.sv accepts the mvp matrix, the coordinates of three vertexes of a triangle and the parameter of the monitor (i.e. width, height). This module will call the matrix multiplication module to do 3 matrix multiplication to project the points in 3D to 2D, after which it will apply the normalization and scale to make the projected triangle fit the screen. When a point is out of the screen, clip the triangle containing this point.

4.6.3 Breif Introduction to MVP Transformation

To introduce MVP transformation, we need to first show what is the **homogeneous coordinate system**. In computer graphics, a 3D vector is represented in 3D homogeneous coordinate system with the form $\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$, where the last coordinate represents the vector is a point. The advantage of such coordinates system is that, we could apply transition (so called affine transformation in mathematics) to an object through matrix multiplication while in normal 3D coordinates, matrix multiplication could only represents linear transformation without transition. An affine transformation matrix in 3D homogeneous coordinates system has the following form:

$$\begin{bmatrix} a & b & c & \textbf{x\_translate} \\ d & e & f & \textbf{y\_translate} \\ g & h & i & \textbf{z\_translate} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where the left upper 3x3 submatrix represents linear transformation and the right 3x1 submatrix represents the translation vector. Beyond that, matrix multiplication in homogeneous coordinate system can also apply other transformation such as perspective projection, which would be introduced below.

Then we begin our introduction to MVP transformation, which has the full name, model, view, projection transformation. To display a 3D object on the screen, a model, view, projection transformation should be apply to the 3D object.

At first, we need to place the object in the world coordinates system, which is done by the **model matrix**. It consists of rotation, scale and translation transform:

$$[\textbf{Rotation\_matrix}] =$$

$$\begin{bmatrix} \cos\alpha\cos\gamma - \cos\beta\sin\alpha\sin\gamma & \sin\alpha\cos\gamma + \cos\beta\cos\alpha\sin\gamma & \sin\beta\sin\gamma & 0 \\ -\cos\alpha\sin\gamma - \cos\beta\sin\alpha\cos\gamma & -\sin\alpha\sin\gamma + \cos\beta\cos\alpha\cos\gamma & \sin\beta\cos\gamma & 0 \\ \sin\beta\sin\alpha & -\sin\beta\cos\alpha & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This is a rotation matrix of the 3 Euler angles. i.e. the object can rotate freely along 3 axises, and this matrix represents the rotation status. For more details see Euler angles.

$$[\textbf{Scale\_matrix}] = \begin{bmatrix} \textbf{scale} & 0 & 0 & 0 \\ 0 & \textbf{scale} & 0 & 0 \\ 0 & 0 & \textbf{scale} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This matrix is used to compute the model matrix, where scale represents the zoom factor of the object. This is a simple linear transformation, just multiplying each of the vector with the scale.

$$[\textbf{Translate\_matrix}] = \begin{bmatrix} 1 & 0 & 0 & \textbf{x\_translate} \\ 0 & 1 & 0 & \textbf{y\_translate} \\ 0 & 0 & 1 & \textbf{z\_translate} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This matrix is used to compute the model, where x_translate, y_translate and z_translate represents the position of the object. This is just a simple linear transformation that applies the translation.

The final **model matrix** would be

$$[\textbf{Model\_matrix}] = [\textbf{translate\_matrix}] * [\textbf{rotation\_matrix}] * [\textbf{scale\_matrix}]$$

Next, we need to place the camera, which is done by the **view matrix**. It consists of rotation and translation transformation. Rotation transformation sets the orientation of the camera while translation transformation set the position of the camera. In our design, the camera does not have view angle, therefore the view matrix is in this form:

$$[\textbf{View\_matrix}] = \begin{bmatrix} 1 & 0 & 0 & -\textbf{x\_pos} \\ 0 & 1 & 0 & -\textbf{y\_pos} \\ 0 & 0 & 1 & -\textbf{z\_pos} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

x_pos, y_pos and z_pos represents the position of the camera. In this matrix, we just apply a simple linear transformation to translate the camera back to the original point.

Finally, we need to project the whole scene onto the screen because what we see on the screen is a 2D image. This is done by **projection matrix**. However, we cannot simply get rid of the z coordinate because in reality, everything looks small in the distance and big on the contrary, i.e. there is a perspective relation between object and distance. To get a perspectively projected image, we apply the perspective matrix to the object, which has the following form:

$$[\textbf{Perspective\_matrix}] = \begin{bmatrix} \textbf{zNear} & 0 & 0 & 0 \\ 0 & \textbf{zNear} & 0 & 0 \\ 0 & 0 & \textbf{zNear} + \textbf{zFar} & -\textbf{zNear} * \textbf{zFar} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Basically, it transform a frustum of view with all objects inside the frustum into a cuboid (closer plane remains unchanged, farther plane being compressed), so that objects would

be small in the distance and big on the contrary. **zNear**, **zFar** here is the near and far z coordinate of the frustum perspectively.

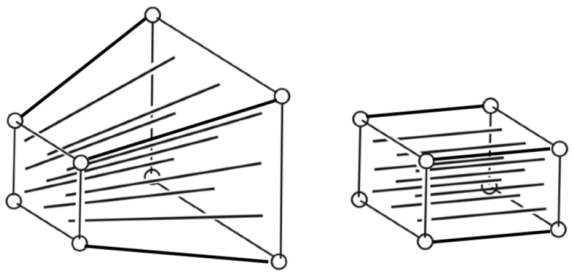

FIGURE 6. Frustum of view and cuboid after transformation

After apply this to the point, we could get rid of z coordinates of the point. However, in computer graphics, we usually transform the cuboid to a $[-1,1]^3$ cube (so called canonical) cube, for easier processing of vertexes such as clipping. Therefore, we need another translate matrix and scale matrix to do this:

$$[\textbf{translate\_matrix}] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -\frac{(\textbf{zNear}+\textbf{zFar})}{2} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$[\textbf{scale\_matrix}] = \begin{bmatrix} \frac{1}{\textbf{zNear}*\tan(\textbf{fov}/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{\textbf{zNear}*\tan(\textbf{fov}/2)*\textbf{aspect\_ratio}} & 0 & 0 \\ 0 & 0 & \frac{2}{\textbf{zNear}-\textbf{zFar}} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where fov is the angle of field of view and aspect ratio is the ratio of the width and height of the screen.

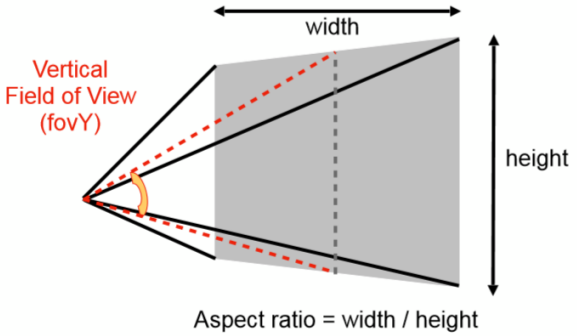

FIGURE 7. fov and aspect ratio

The final **projection matrix** would be

$$[\textbf{Projection\_matrix}] = [\textbf{scale\_matrix}] * [\textbf{translate\_matrix}] * [\textbf{perspective\_matrix}]$$

In our design, to save resources, we precompute the projection matrix so that matrix multiplication is not needed in this part.

$$[\mathbf{Projection\_matrix}] =$$

$$\begin{bmatrix} \mathbf{inv\_tan/aspect\_ratio} & 0 & 0 & 0 \\ 0 & \mathbf{inv\_tan} & 0 & 0 \\ 0 & 0 & (\mathbf{zFar} + \mathbf{zNear}) \cdot \mathbf{k} & 2\mathbf{k} \cdot \mathbf{zFar} \cdot \mathbf{zNear} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

where $\mathbf{inv\_tan} = \frac{1}{\mathbf{tan(eye\_fov)}}$, $\mathbf{k} = \frac{1}{\mathbf{zNear} - \mathbf{zFar}}$

The final MVP transformation matrix would be

$$[\mathbf{MVP\_matrix}] = [\mathbf{projection\_matrix}] * [\mathbf{view\_matrix}] * [\mathbf{model\_matrix}]$$

4.7 Triangle FIFO

This module is a FIFO which stores projected triangle data (2D coordinates of vertexes in screen space) using on-chip memory. It contains a module called triangle fifo ram which would be automatically convert to block ram by Quartus II compiler. The storage of the list can be customized by setting the parameter.

We use a register **num** to record the number of elements in the fifo. If a new element is pushed, the write pointer would increase; if an element is popped, the read pointer would increase. Write or read pointer would back to the first address if they reach the last address, so the fifo has a ring structure. When the read pointer and write pointer point to the same address (using **num** to justify), it would indicate the fifo is empty. When **num** is max, fifo would indicate itself is full.

4.8 Draw

This module is a rather complex module, containing a 3-level hierarchy. Projected triangles (2D coordinates of vertexes in screen space) would be drawn in this module. The top level module draws all triangles, the second top level module draw_triangles.sv draws one triangles, the bottom level module draw_line.sv draws one line. To be specific, this module would output DrawX, DrawY signals indicating the coordinates of line pixel, and with the help of the control unit, it would write line pixel into frame buffer for every triangles in triangle FIFO.

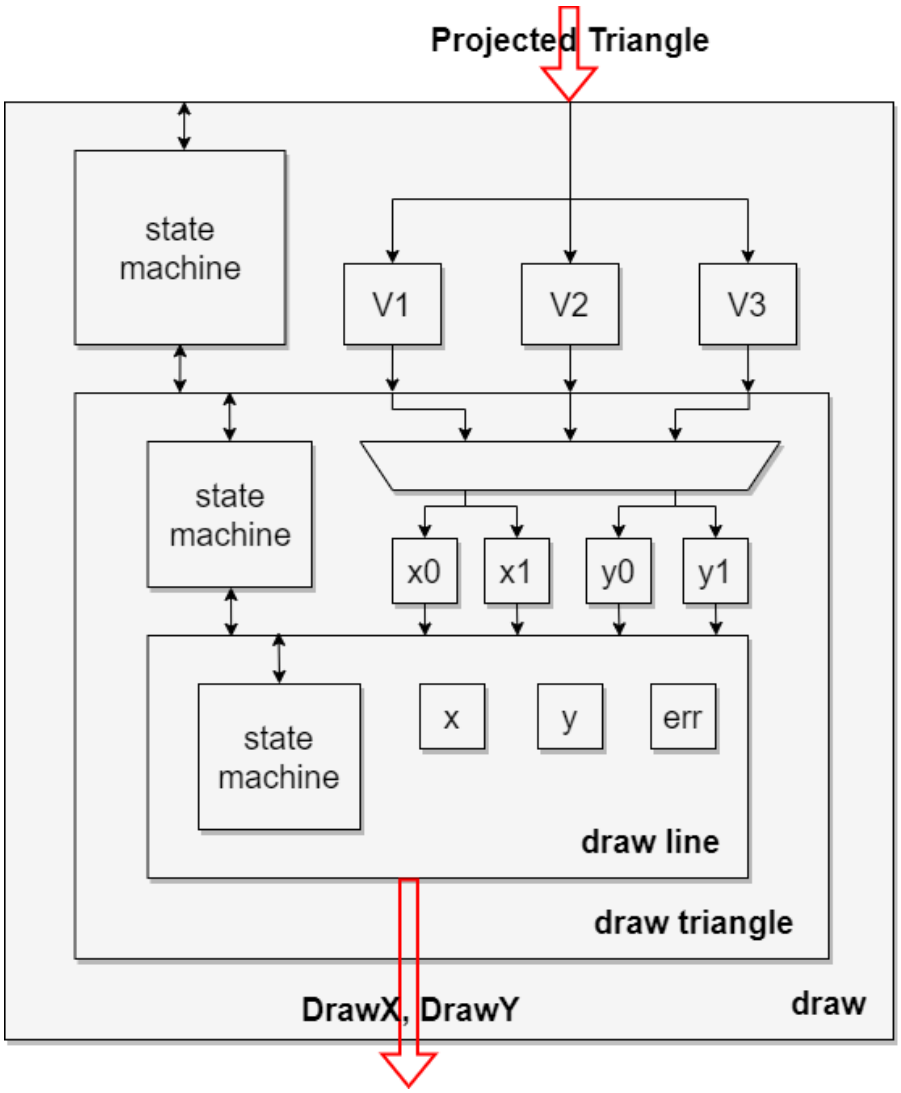

FIGURE 8. Draw module diagram

4.8.1 State Machine of Draw Module

The state machine reads projected triangle data from the triangle fifo then pass to draw_triangle module.

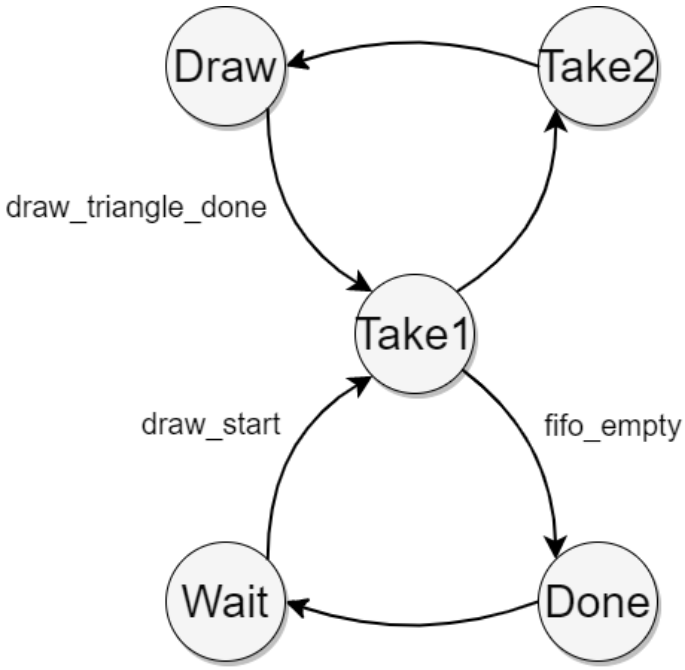

FIGURE 9. State machine diagram of draw module

As we can see in the state machine diagram, the control unit has 5 states, **Wait, Take1, Take2, Draw, Done**. Details are shown below.

- **Wait**: Wait to draw triangles until triggered by draw_start.
- **Take1**: Trigger to take projected triangles from the triangle fifo. Read enable signal of triangle fifo is high in this state. If all triangles in triangle fifo are drawn (fifo is empty), end drawing.
- **Take2**: Take projected triangles from the triangle fifo.
- **Draw**: Trigger draw_triangle, draw the triangle. Transfer to Take1 if the drawing of current triangle is done.
- **Done**: Done. draw_done is high in this state.

4.8.2 Draw Triangle

This module draws a triangle by triggering draw_line module 3 times to draw 3 edges of the triangle.

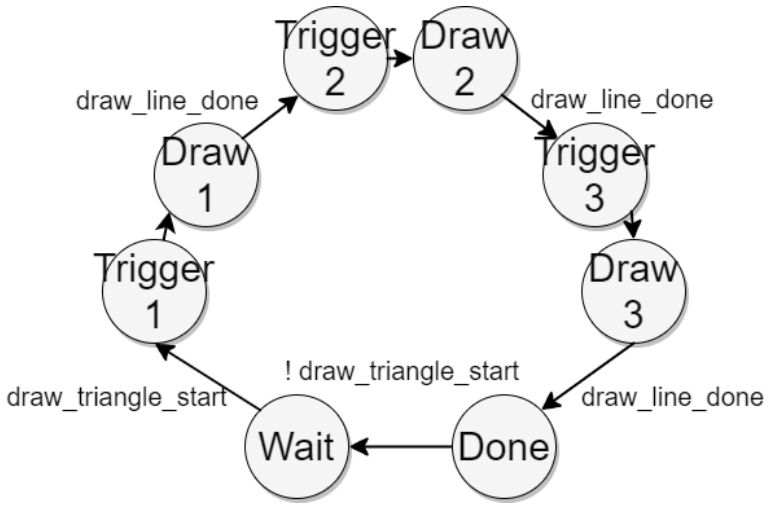

FIGURE 10. State machine diagram of draw triangle module

As we can see in the state machine diagram, the control unit has 8 states, **Wait, Trigger1, Draw1, Trigger2, Draw2, Trigger3, Draw3, Done**. Details are shown below.

- **Wait**: Wait to draw a triangle.
- **Trigger1/2/3**: Trigger the drawing of one edge. $V_i, V_j, i \neq j, 1 \leq i \leq j \leq 3$ would be unpacked then stored in register x0,x1,y0,y1 for draw line module to draw in each state.
- **Draw1/2/3**: Drawing edges of the triangle using draw line module. $V_i, V_j, i \neq j, 1 \leq i \leq j \leq 3$ would be unpacked then stored in register x0,x1,y0,y1 for draw line module to draw in each state.
- **Done**: Done. Halt until current trigger signal is low. draw_triangle_done is high in this state.

4.8.3 Draw Line& Breif Introduction to Bresenham Algorithm

This module draws a line according to the given vertexes using Bresenham line drawing algorithm. To be specific, it outputs coordinates of line pixels in serial. When we consider to draw a line on the screen, calculating the analytic expression of the line then traversing all pixels in a rectangular to decide whether a pixel is on a line would be a natural idea. However, this needs floating/fixed point calculation including multiplication and would waste many times on traversing blank pixels, which costs big resources and is inefficient. Bresenham algorithm is powerful because it only uses integer addition, subtraction and could generate coordinates of a line serially. Basically, it record a position starts from one vertexes, increase/decrease x or y coordinates of it every step according to its relative position with the line, until the position meets another vertexes. To avoid floating point calculation, it converts the original line expression

$$y = kx + b = \frac{\Delta y}{\Delta x}x + b$$

into

$$\Delta yx - \Delta xy + C = 0$$

, and uses a middle variable **err** to record the error between the position and the line so that every step we only need to update **err** instead of compare the position with the line.

Here is the pseudocode of this algorithm from Wikipedia.

```
plotLine(int x0, int y0, int x1, int y1)
    dx =  abs(x1-x0);
    sx = x0<x1 ? 1 : -1;
    dy = -abs(y1-y0);
    sy = y0<y1 ? 1 : -1;
    err = dx+dy;  /* error value e_xy */
    while (true)   /* loop */
        plot(x0, y0);
        if (x0 == x1 && y0 == y1) break;
        e2 = 2*err;
        if (e2 >= dy) /* e_xy+e_x > 0 */
            err += dy;
            x0 += sx;
        end if
        if (e2 <= dx) /* e_xy+e_y < 0 */
            err += dx;
            y0 += sy;
        end if
    end while
```

In this module, we implement this algorithm by a state machine.

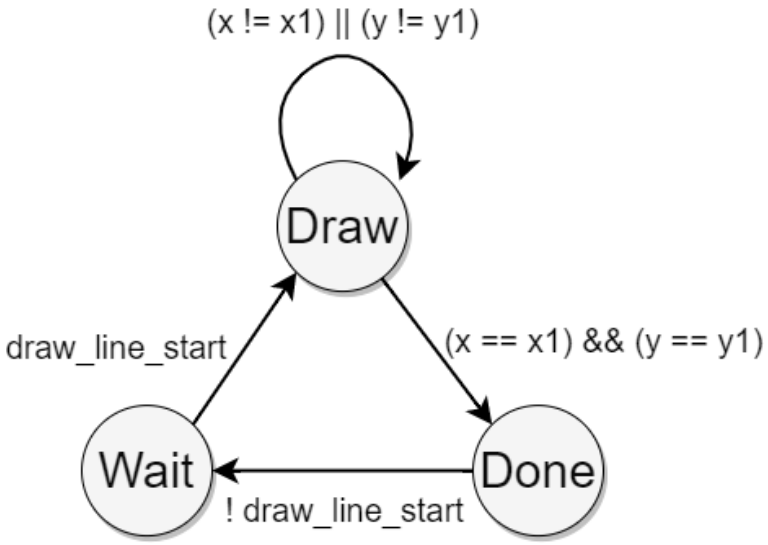

FIGURE 11. State machine diagram of draw line module

As we can see in the state machine diagram, the control unit has 3 states, **Wait, Draw, Done**. Details are shown below.

- **Wait**: Wait to draw a line.

- **Draw**: In this state, the machine would continuously update x,y coordinates of the position and textbferr from one end of the line, output the coordinates of line pixel until the position meets the other end of the line.
- **Done**: Done. Halt until current trigger signal is low. draw_line_done is high in this state.

4.9 Frame Buffer

To avoid flashing of the picture, we use 2-buffer and ping-pong strategy to display. During displaying, one buffer is updating while the other is read by VGA display logic. We uses a register **sw** to indicate which frame is to read/write. When next frame is come and rendering of current frame is done, the **sw** would update. Therefore, our renderer has a dynamic frame rate.

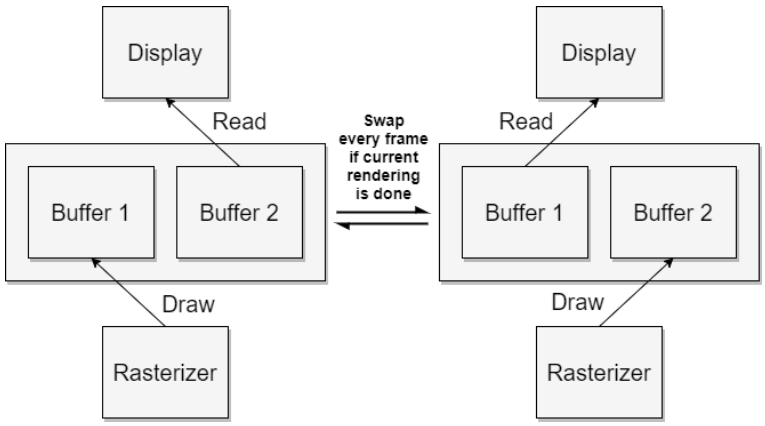

FIGURE 12. Swap of 2 frame buffer

For storage, because we only need 1 bit to represent one pixel (whether is a line or not), the storage would not be so big, the total storage of 2 buffers are

$$640 \times 480 \times 1bits \times 2\text{frame} = 614400bits \approx 600kbits$$

Therefore we choose on-chip memory to store 2 frame buffers. Dual port is used so that read and write can be done in one cycle.

4.10 Clear Frame

Every frame needs to be cleared before triangles of next frame are drawn on it. Therefore a clear module is needed to generate signal to traverse every pixel to clear the frame.

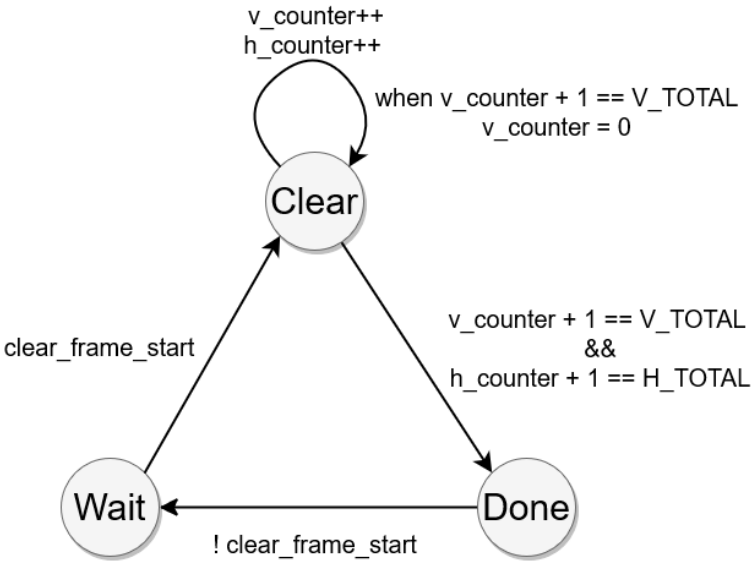

FIGURE 13. State machine diagram of clear frame module

As we can see in the state machine diagram, the control unit has 3 states, **Wait, Draw, Done**. Details are shown below.

- **Wait**: Wait to clear the frame.
- **Clear**: Generate signal traversing every pixel in a frame to clear it. A vertical counter and horizontal counter would be updated every cycle. When the horizontal counter reach 640, it would update to 0 and the vertical counter would increse by 1. When the vertical counter reach 480, clear is done and the machine would transfer to **Done** state.
- **Done**: Done. Halt until current trigger signal is low. clear_frame_done is high in this state.

4.11 VGA Controller

This module generates read signal of the VGA display and the a frame clock indicating whether a new frame is come. This module is from lab8 and is slightly modified to satisfied our requirement.

4.12 Color Mapper

This module mapped pixel data (whether a pixel should be drawn) in frame to different color. This module is from lab8 and is slightly modified to satisfied our requirement.

5 Module Descriptions

5.1 Third-Party Open-Source Modules

We only use one third-party open-source library in this project, which is a fixed point number calculation library. Between fixed point number and floating point number, we finally decide to use the fixed point number because it is more simple and fast to compute. Despite it may not be so accurate, we estimate the possible

errors and found them affordable. The fix point number library we use is fixed-point_lib.sv, which supports parameterized bit width and addition, subtraction, multiplication, division and sin. It works for most of the cases, but still has certain drawbacks. First, its division and sin modules are not likely to be timing closure, which causes us great trouble in the integration of components. The idea of our solution is to add a WAIT state similar to Lab6. Another problem is that the sin module works only for $[0, \frac{\pi}{2}]$ and would crash around 0 and $\frac{\pi}{2}$. Our solution is to make transformation to turn arbitrary angles in $[0, 2\pi]$ to $[0, \frac{\pi}{2}]$ and use Taylor expansion around 0 and $\frac{\pi}{2}$. See module description and documentation of bugs encountered for more details.

5.2 All .sv Modules

Most of modules here has been described in implementation part. Therefore we simplify the description here. **Module**: clear_frame.sv
Inputs: Clk, Reset, clear_frame_start
Outputs: [9:0] DrawX, [9:0] DrawY
Description: Generating signal to clear a frame by traversing every pixel in a frame.
Purpose: Clear the frame at the beginning of every frame clock cycle

Module: color_mapper.sv
Inputs: is_pixel, [9:0] ReadX, [9:0] ReadY
Outputs: [7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B
Description: Same as the file in Lab 8, decide which color to be output to VGA for each pixel.
Purpose: We need this module to render the line of triangles on the screen, drawing the monitor with colors. So we can visually see the movement of the 3D object on the monitor.

Module: control_unit.sv
Inputs: Clk, Reset, frame_clk, draw_done, clear_done, load_done, [9:0] draw_DrawX, [9:0] draw_DrawY, [9:0] clear_DrawX, [9:0] clear_DrawY
Outputs: load_obj, draw_start, clear_start, proj_start, draw_data, [9:0] DrawX, [9:0] DrawY, frame_clk_rising_edge, frame_done
Description: Control unit for the renderer, a state machine
Purpose: Control most of the signal of the renderer.

Module: display.sv
Default Parameters: WI=8, WF=8
Inputs: Clk, Reset, frame_clk_rising_edge, [7:0] keycode
Outputs: [11:0] alpha, [11:0] beta, [11:0] gamma, [WI+WF-1:0] x, [WI+WF-1:0] y, [WI+WF-1:0] z
Description: Control the motion of the object and camera.
Purpose: This module control the motion of the object and camera according to the keycode passed by the NIOS II system. It updates the motion every frame (not every cycle).

Module: draw_line.sv

Inputs: Clk, draw_line_start, Reset, [9:0] x0, [9:0] x1, [9:0] y0, [9:0] y1
Outputs: [9:0] DrawX, [9:0] DrawY, draw_line_done
Description: Use Bresenham line drawing algorithm to draw a line.
Purpose: This module draws a line according to the given vertexes using Bresenham line drawing algorithm. To be specific, it outputs coordinates of line pixels in serial.

Module: draw_triangle.sv
Inputs: Clk, Reset, draw_triangle_start, [1:0][9:0] V1, [1:0][9:0] V2, [1:0][9:0] V3
Outputs: [9:0] DrawX, [9:0] DrawY, draw_triangle_done
Description: Draw a triangle (only edges).
Purpose: This module draws a triangle by triggering draw_line module 3 times to draw 3 edges of the triangle.

Module: draw.sv
Inputs: Clk, Reset, draw_start, [2:0][1:0][9:0] triangle_data, fifo_empty
Outputs: fifo_r, [9:0] DrawX, [9:0] DrawY, draw_done
Description: Draw all triangles on the screen.
Purpose: All projected triangles (2D coordinates of vertexes in screen space) in triangle FIFO would be drawn in this module.

Module: euler_angle.sv
Default Parameters: WII=4, WIF=8, WOI=2, WOF=12
Inputs: [WII+WIF-1:0] alpha, [WII+WIF-1:0] beta, [WII+WIF-1:0] gamma
Outputs: [15:0][WOI+WOF-1:0] euler_angle_matrix
Description: This module calculates the 4x4 rotation matrix of the given 3 Euler angle. For Euler angle and its rotation matrix, see Euler angles for detail.
Purpose: The rotation matrix of the Euler angle is used in the module get_model_matrix.sv, which need a rotation matrix to set the angles of the object itself.

Module: fifo_writer.sv
Inputs: Clk, Reset, clear_start,
Outputs: fifo_w, [2:0][1:0][9:0] proj_triangle_in
Description: [test module] for triangle fifo which stores projected triangles in screen space, which writes projected triangles into fifo (x, y coordinates of vertexes in screen space).
Purpose: test triangle the functionality of triangle FIFO.

Module: frame_buffer.sv
Inputs: Clk, Reset, frame_clk_rising_edge, [9:0] DrawX, [9:0] DrawY, draw_data, frame_done, [9:0] ReadX, [9:0] ReadY
Outputs: read_data
Description: On-chip memory frame buffer using 2-buffer (ping-pong method). keep the frame if drawing does not finished.
Purpose: Store the frame and provide 2-buffer swapping logic.

Module: get_model_matrix.sv
Default Parameters: WIIA=4, WIFA=8, WIIB=8, WIFB=8, WOIA=2, WOFA=12,

WOI=8, WOF=8
Inputs: [WIIA+WIFA-1:0] alpha, [WIIA+WIFA-1:0] beta, [WIIA+WIFA-1:0] gamma, [WIIB+WIFB-1:0] scale, [WIIB+WIFB-1:0] x_translate, [WIIB+WIFB-1:0] y_translate, [WIIB+WIFB-1:0] z_translate
Outputs: [15:0][WOI+WOF-1:0] model_matrix
Description: Calculate the 4x4 model matrix, which reflects the position of the object. It equals translate_matrix · rotation_matrix · scale_matrix.
Purpose: The model matrix is needed to calculate the mvp matrix, which equals projection_matrix · view_matrix · model_matrix.

Module: get_mvp_matrix.sv
Default Parameters: WII=8, WIF=8, WOI=8, WOF=8
Inputs: [15:0][WII+WIF-1:0] model_matrix, [15:0][WII+WIF-1:0] view_matrix, [15:0][WII+WIF-1:0] projection_matrix
Outputs: [15:0][WOI+WOF-1:0] mvp_matrix
Description: This module calculates the mvp matrix by matrix multiplications, which equals projection_matrix · view_matrix · model_matrix.
Purpose: mvp matrix is used to project a point from 3D to 2D by a 4x4 · 4x1 matrix multiplication, where the 4x4 matrix is the mvp matrix, and the 4x1 vector is the 4D homogeneous coordinates of a 3D point in the space.

Module: get_view_matrix.sv
Default Parameters: WII=8, WIF=8, WOI=8, WOF=8
Inputs: [WII+WIF-1:0] inv_tan, [WII+WIF-1:0] aspect_ratio, [WII+WIF-1:0] z_near, [WII+WIF-1:0] z_far
Outputs: [15:0][WOI+WOF-1:0] projection_matrix
Description: Calculate the 4x4 view matrix, which reflects the position of the camera.
Purpose: The view matrix is needed to calculate the mvp matrix, which equals projection_matrix · view_matrix · model_matrix.

Module: get_projection_matrix.sv
Default Parameters: WII=8, WIF=8, WOI=8, WOF=8
Inputs: [WII+WIF-1:0] x_pos, [WII+WIF-1:0] y_pos, [WII+WIF-1:0] z_pos
Outputs: [15:0][WOI+WOF-1:0] view_matrix
Description: Calculate the 4x4 projection matrix, which projects the object from its ordinary position to a standard space of $[-1, -1]^3$ unit space.
Purpose: The projection matrix is needed to calculate the mvp matrix, which equals projection_matrix · view_matrix · model_matrix.

Module: HexDriver.sv
Inputs: [3:0]In0
Outputs: [6:0]Out0
Description: This is a hexadecimal display driver.
Purpose: This module is used to drive the LED segment displays on FPGA to display the hexadecimal number of certain results.

Module: hpi_io_intf.sv

Inputs: Clk, Reset, [1:0] from_sw_address, [15:0] from_sw_data_out, from_sw_r, from_sw_w, from_sw_cs, from_sw_reset,
Outputs: [15:0] from_sw_data_in, [1:0] OTG_ADDR, OTG_RD_N, OTG_WR_N, OTG_CS_N, OTG_RST_N
Inouts: [15:0] OTG_DATA
Description: This module is able to deliver and process data between NIOS and OTG chip.
Purpose: We need this module to interfaces between NIOS and OTG chip.

Module: list_writer.sv
Default Parameters: WI=8, WF=8
Inputs: Clk, Reset, load_obj
Outputs: list_w, load_done, [2:0][2:0][WI+WF-1:0] orig_triangle_in
Description: This module writes original triangle data into the triangle list.
Purpose: This module writes original triangle data (model data) into the triangle list.

Module: project_cal.sv
Default Parameters: WIIA=4, WIFA=8, WI=8, WF=8
Inputs: [2:0][2:0][WI+WF-1:0] orig_triangle, [WIIA+WIFA-1:0] alpha, [WIIA+WIFA-1:0] beta, [WIIA+WIFA-1:0] gamma, [WI+WF-1:0] x_translate, [WI+WF-1:0] y_translate, [WI+WF-1:0] z_translate
Outputs: [2:0][1:0][9:0] proj_triangle, clip
Description: This module calculates mvp matrix, then project triangles.
Purpose: We need this module to projects all the triangles to screen every frame so that the graphics is rendered. This module is instantiated and controlled by a state machine in project.sv.

Module: project_triangle.sv
Default Parameters: WIIA=8, WIFA=8, WIIB=12, WIFB=0, WI=12, WF=8, WOI=12, WOF=0
Inputs: [3:0][WIIA+WIFA-1:0] vertex_a, [3:0][WIIA+WIFA-1:0] vertex_b, [3:0][WIIA+WIFA-1:0] vertex_c, [15:0][WIIA+WIFA-1:0] mvp, [WIIB+WIFB-1:0] width, [WIIB+WIFB-1:0] height
Outputs: [1:0][WOI+WOF-1:0] V1, [1:0][WOI+WOF-1:0] V2, [1:0][WOI+WOF-1:0] V3, clip
Description: This module projects one single triangle based on the input mvp matrix and coordinate vector.
Purpose: We need to instantiate this module and continuously refresh it to project all the triangles to screen, which is done in project_cal.sv.

Module: project.sv
Default Parameters: WIIA=4, WIFA=8, WI=8, WF=8
Inputs: Clk, Reset, proj_start, [2:0][2:0][WI+WF-1:0] orig_triangle, [WIIA+WIFA-1:0] alpha, [WIIA+WIFA-1:0] beta, [WIIA+WIFA-1:0] gamma, [WI+WF-1:0] x_translate, [WI+WF-1:0] y_translate, [WI+WF-1:0] z_translate, list_read_done
Outputs: [2:0][1:0][9:0] proj_triangle, list_r, fifo_w, proj_done
Description: This module projects original triangles (vertexes) in 3D space into

2D screen space.
Purpose: This module projects original triangles (vertexes) in 3D space into 2D screen space.

Module: renderer_top.sv
Inputs: CLOCK_50, [3:0] KEY, [7:0] SW, OTG_INT
Outputs:[7:0] LEDR, [6:0] HEX0, [6:0] HEX1, [7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B, VGA_CLK, VGA_SYNC_N, VGA_BLANK_N, VGA_VS, VGA_HS, [1:0] OTG_ADDR, OTG_CS_N, OTG_RD_N, OTG_WR_N, OTG_RST_N, [12:0] DRAM_ADDR, [1:0] DRAM_BA, [3:0] DRAM_DQM, DRAM_RAS_N, DRAM_CAS_N, DRAM_CKE, DRAM_WE_N, DRAM_CS_N, DRAM_CLK
Inouts: [15:0] OTG_DATA, [31:0] DRAM_DQ
Description: This module is the top Level Module For FPGA-based 3D Graphics Renderer that renders arbitary simple 3D module (approximately less than 100 triangles) and controls the rotation, translation of the module smoothly using keyboard.
Purpose: We need this top level module to combine all the components together.

Module: triangle_fifo_ram.sv
Default Parameters: Waddr=7, size=100
Inputs: Clk, r_en, w_en, [Waddr-1:0] r_addr, [Waddr-1:0] w_addr, is_empty, is_full, [6*10-1:0] data_in
Outputs: [6*10-1:0] data_out
Description: This module is an on-chip memory fifo.
Purpose: Store the triangle FIFO.

Module: triangle_fifo.sv
Default Parameters: Waddr=7, size=100
Inputs: Clk, Reset, r_en, w_en, [2:0][1:0][9:0] triangle_in
Outputs: [2:0][1:0][9:0] triangle_out, is_empty, is_full
Description: This module is a customized triangle fifo using On-chip memory.
Purpose: Store the projected triangle data for rendering in every frame.

Module: triangle_list_ram.sv
Default Parameters: WI=8, WF=8, Waddr=7, size=100
Inputs: Clk, r_en, w_en, [Waddr-1:0] r_addr, [Waddr-1:0] w_addr, is_empty, is_full, [(WI+WF)*9-1:0] data_in
Outputs: [(WI+WF)*9-1:0] data_out
Description: This is an on–chip memory list, which has 2 ways to use. Way1: write triangles into list then read. Way2: initial on-chip memory by .txt file while compiled then read. Different ways need you to comment/uncomment some lines.
Purpose: Store the triangle list.

Module: triangle_list.sv
Default Parameters: WI=8, WF=8, Waddr=7, size=100
Inputs: Clk, Reset, r_en, w_en, [2:0][2:0][(WI+WF)-1:0] triangle_in
Outputs: [2:0][2:0][(WI+WF)-1:0] triangle_out, is_empty, is_full, read_done
Description: This module is a customized triangle list using On-chip memory that

reads traverse the list. This module has 2 ways to use. Way1: write triangles into list then read. Way2: initial on-chip memory by .txt file while being compiled then read. Different ways need you to comment/uncomment some lines.
Purpose: Store the original triangle data (3D coordinates of vertexes) for vertex process.

Module: VGA_controller.sv
Inputs: Clk, Reset, VGA_CLK
Outputs: VGA_HS, VGA_VS, VGA_BLANK_N, VGA_SYNC_N, [9:0] ReadX, [9:0] ReadY
Description: Original code is the provided VGA_controller.sv in Lab 8, and we slightly modify it.
Purpose: We need this module to run the monitor, refreshing the screen to display our project and generating frame clock.

Module: trigonometric_lib.sv/cal_sin
Default Parameters: WII=4, WIF=8, WOI=2, WOF=12, ROUND=1
Inputs: [WII+WIF-1:0] in
Outputs: [WOI+WOF-1:0] out
Description: This module is the modification of the sin module in fixedpoint_lib.sv, now it supports arbitrary input angle within range of [0, 2π]
Purpose: We need this module to get the sin value of an angle within range [0, 2π]

Module: trigonometric_lib.sv/cal_cos
Default Parameters: WII=4, WIF=8, WOI=2, WOF=12, ROUND=1
Inputs: [WII+WIF-1:0] in
Outputs: [WOI+WOF-1:0] out
Description: Since the fixedpoint_lib.sv does not provide a cos function module, we need this module to convert cos values to sin values and then compute.
Purpose: We need this module to get the cos value of an angle within range [0, 2π]

Module: trigonometric_lib.sv/fix_sin
Inputs: [11:0] in
Outputs: [13:0] out
Description: Because the sin function in third-party open-source library fixedpoint_lib.sv does not support sin values near 0 or $\frac{\pi}{2}$, we modify to fix the module to work properly. Around 0 or $\frac{\pi}{2}$, this module applies Taylor expansion directly to obtain values.
Purpose: We need this module to deal with the boundary condition of the provided sin function module in fixedpoint_lib.sv so that we can access to the trigonometric values of angles near 0 or $\frac{\pi}{2}$.

Module: matrix_lib.sv/dot_product
Default Parameters: WII=8, WIF=8, WOI=8, WOF=8
Inputs: [WII+WIF-1:0] a0, [WII+WIF-1:0] a1, [WII+WIF-1:0] a2, [WII+WIF-1:0] a3, [WII+WIF-1:0] b0, [WII+WIF-1:0] b1, [WII+WIF-1:0] b2, [WII+WIF-1:0] b3
Outputs: [WOI+WOF-1:0] res

Description: This module calculates the dot production of 2 vectors. vectorA = (a0, a1, a2, a3); vectorB = (b0, b1, b2, b3), return vectorA · vectorB.
Purpose: We need dot product as basic calculation to compute matrix multiplication.

Module: matrix_lib.sv/
Default Parameters: WII=8, WIF=8, WOI=8, WOF=8
Inputs: [15:0][WII+WIF-1:0] matA, [15:0][WII+WIF-1:0] matB
Outputs: [15:0][WOI+WOF-1:0] res_mat
Description: This module calculates the multiplication of two given matrices, the index of the matrix is row-major order.
Purpose: We need matrix multiplication as basic computation unit in projection and rendering.

- ## 5.3 Qsys Generated File
  - clk_0: Default 50 MHz clock.
  - nios2_gen2_0: The NIOS-II processor (CPU) which controls most of the signal.
  - sdram: SDRAM controller for the on-board SDRAM which is used to store the software in NIOS II. The SDRAM cannot be interfaced to the Avalon bus of NIOS II directly, as it has a complex row/column addressing scheme and requires constant refreshing to retain data, therefore, we need a SDRAM controller to interface the SDRAM to the Avalon bus.
  - sdram_pll: Phase Locked Loop (PLL) component to provide the precise timing clock signal for SDRAM which is required by SDRAM
  - sysid_qsys_0: A system ID checker to ensure the compatibility between hardware and software.
  - keycode: Parallel I/O interface(PIO) for the keycode. It is an output PIO.Through this interface, we could transmit the keycode of keyboard (which stores the code of keys being pressed) to the hardware design.
  - otg_hpi_address: Parallel I/O interface (PIO) for the address signal of otg_hpi. It is an output PIO. Through this interface, software in NIOS II could set the address of the HPI address register in the OTG chip to achieve I/O with the chip.
  - otg_hpi_data: Parallel I/O interface (PIO) for the data of otg_hpi. It is an inout PIO. Through this interface, software in NIOS II could write and read data from the HPI Data register in OTG chip to achieve I/O with chip.
  - otg_hpi_r: Parallel I/O interface (PIO) for the read enable signal of otg_hpi. It is an output PIO. Through this interface, software in NIOS II could control the read operation of the HPI register in OTG chip, in order to achieve I/O with the chip.
  - otg_hpi_w: Parallel I/O interface (PIO) for the write enable signal of otg_hpi. It is an output PIO. Through this interface, software in NIOS II could control the write operation of the HPI register in OTG chip, in order to achieve I/O with the chip.

- otg_hpi_cs: Parallel I/O interface (PIO) for the chip select signal of otg_hpi. It is an output PIO. Through this interface, software in NIOS II could control the I/O of HPI register in OTG chip, to be specific, the chip select signal.
- otg_hpi_reset: Parallel I/O interface (PIO) for the reset signal of otg_hpi. It is an output PIO. Through this interface, software in NIOS II could reset the HPI register in OTG chip in order to achieve I/O with the chip.
- jtag_uart_0: The JTAG UART peripheral. With this, we can use the terminal of the host computer (the one running eclipse) to communicate with the NIOS II (using print and scan statements in c).

6 Design statistics and Resources

LUT	22643
DSP	41
Memory (BRAM)	633,656
Flip-Flop	2,861
Frequency	3.64 MHz
Static Power	109.78 mW
Dynamic Power	738.27 mW
Total Power	926.91 mW

Table 1: Design Statistics Table.

6.1 Documentation of Problems Encountered and Our Solutions

The significant bugs we encountered and solved are listed below:

- The trigonometric calculation was wrong at first because the open-source library we adopted only supported sin function within range of $[0, \frac{\pi}{2}]$. Besides, that module will crash once the input angle is close to 0 or $\frac{\pi}{2}$. Here are two step solutions we apply to solve this problem. First we wrote testbenches to measure the boundary line of the module's crash empirically around 0 and $\frac{\pi}{2}$. Then we introduce Taylor expansion mechanism: $sin(x) = x$ and $sin(x) = 1$ when $x \approx 0$ or $x \approx \frac{\pi}{2}$. In this way, the sin module can compute the sin value for all angle within $[0, \frac{\pi}{2}]$. Then we make use of the period property of the trigonometric functions to transform arbitrary angle within $[0, 2\pi]$ into another angle within $[0, \frac{\pi}{2}]$ to compute. In this way, we successfully modify the existing module to compute all the cos, sin value of angle in range of $[0, 2\pi]$.
- The division operation and sin operation in the open-source library are not timing closure, so unstable cases occurs frequently when integrating all the components together in top level. So we modify each of our control logic, mainly the state machine. Inspired by the WAIT-another-state when accessing the SRAM in Lab6, we added additional 4 clock cycle as WAIT state in our state machine. And the problem was solved.
- Originally we use our own sv logic to generate frame buffer ram but failed. It is mainly because the swap logic and ram read/write logic is mixed together

so that the compiler cannot convert reg automatically into BRAM. We fixed the bug by using RAM IP provided by Quartus II for framebuffer.

7 Conclusion

7.1 Functionality Discussion

Our final project meets almost all the features and functions in the proposal. With a Python script that prase the .obj file into binary data, the 3D graphic renderer can load arbitrary models and render it on the screen, where users can control the movement and rotation of the object through USB keyboard. Due to our special design that takes angular velocity and angular acceleration into consideration, the keyboard control signal works quite smoothly. Therefore, we regard our project to be successful.

7.2 Future improvements

Except for the existing features and functions, we believe there are still many improvements that can be done. And we list them as follows:

+ In our current design, certain mathematical operations of the fixed point number are not timing closure. e.g. division and sin. And we think we can either modify the library itself, or apply some state machine strategies to make those calculations to be timing closure. In this way, our project will be much more stable with better robustness.
+ In our implementation, mostly, we only apply 8-bit integer and 8-bit decimal, which are not precise enough. Further, the overflow occurs so often that the size of the rendered objects are also limited. We think we can widen the data bits to increase its applicability. Such as 16-bit integer and 16-bit decimal.
+ We can integrate more sophisticated and further knowledge on computer graphic into this project. Such as shading with illumination and texture mapping.