The Ultimate Ball Balancer

CSEE 4840 Embedded System Design

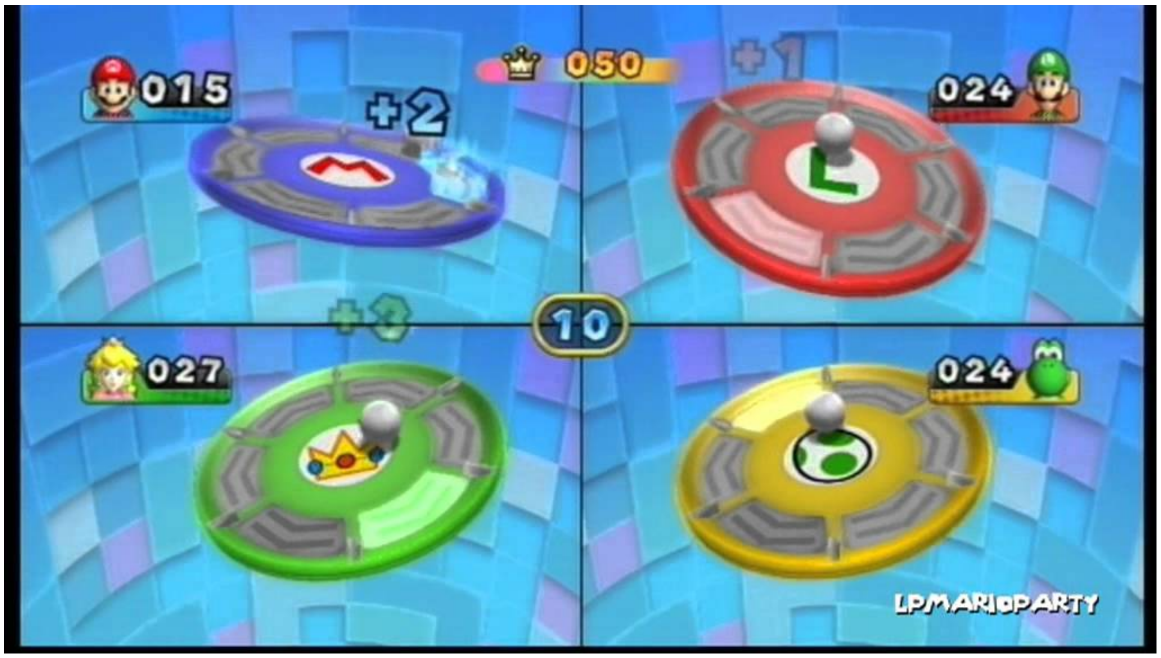
Spring 2014

Earvin Caceres (ec2946)
Gautham Vunnam (gv2226)
Garvit Singh (gs2731)
Annjana Ramesh (ar3303)

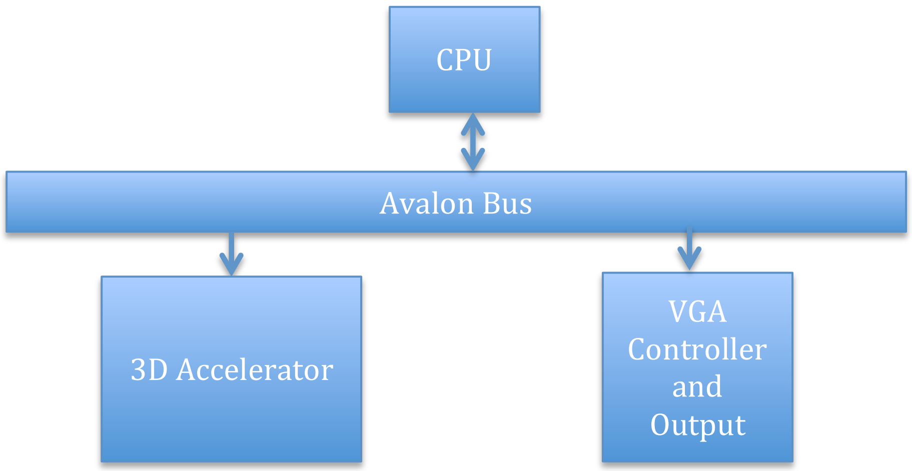
1. Introduction

Our Ball-balance video game is based on the Mario Party game. The game has a 3D-plate at the center of the screen. Sections of the plate will light up and the aim of the player is to balance a ball on that plate, while moving over the section that is lit up in a random color, in order to get more points. Certain colors that are lit up are to be avoided (e.g. red and white are to be avoided). If a player rolls over a penalty section, the player loses points. The player loses the game if the ball falls off the plate.

To implement this project, few challenges need to be overcome. The first task would be to successfully display a plate in 3D. In addition to that, the plate has to be able to rotate in 3D. The user via a PS3 controller specifies the direction of rotation. Effective VGA implementation is key in this aspect. Further, a ball is to move in accordance with the plate; while keeping in mind the effects of gravity. Minor challenges also include how the hardware and software are going to split/handle the workload.



2. Hardware Structure



The CPU has two main hardware peripherals: the 3D Accelerator and the VGA Controller/Output. They are connected to the Avalon bus to enable mutual communication to the VGA output from both the 3D accelerator and CPU.

3D ACCELERATOR

The 3D Accelerator is the main crux of our project. It renders 3D models so that they can be displayed on a 2D screen. It communicates with the VGA Controller, which deals with the VGA output. The major load on the FPGA is from this block. Figure 1 is a block diagram of the 3D Accelerator subsystem along with the main subcomponents it interfaces with. Figure 2 is a reference design of a typical FPGA/External Ram interface.

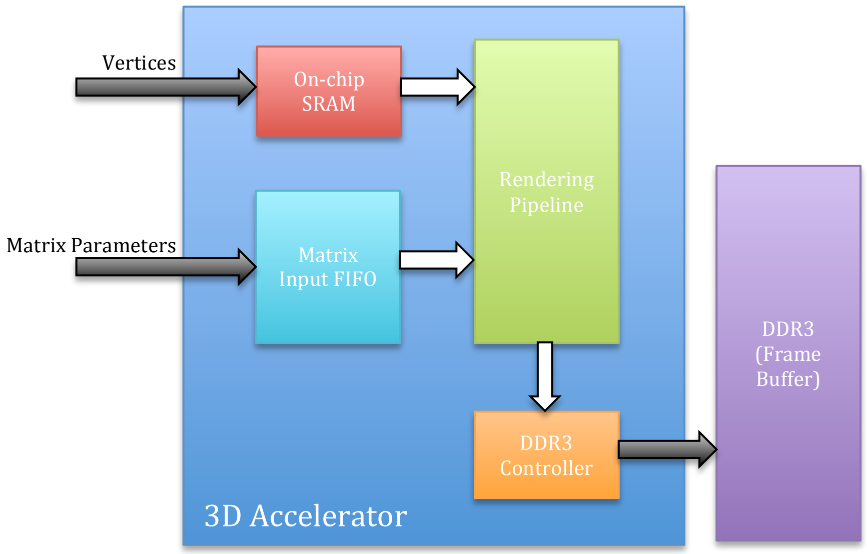


Figure 1: 3D Accelerator Block Diagram

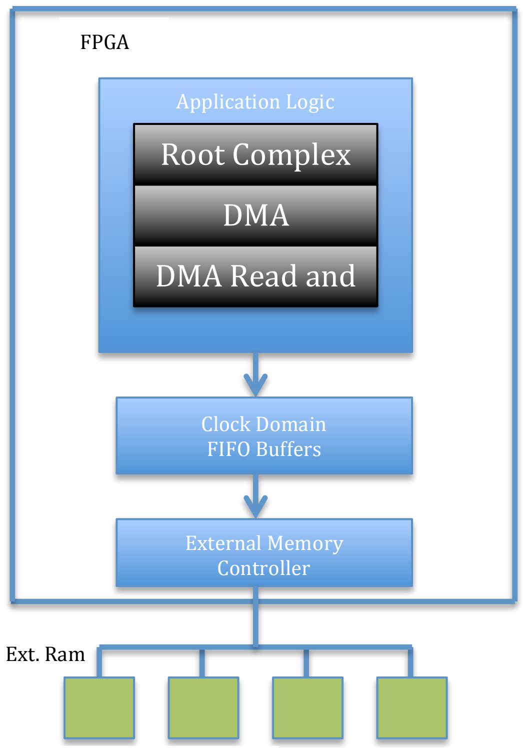
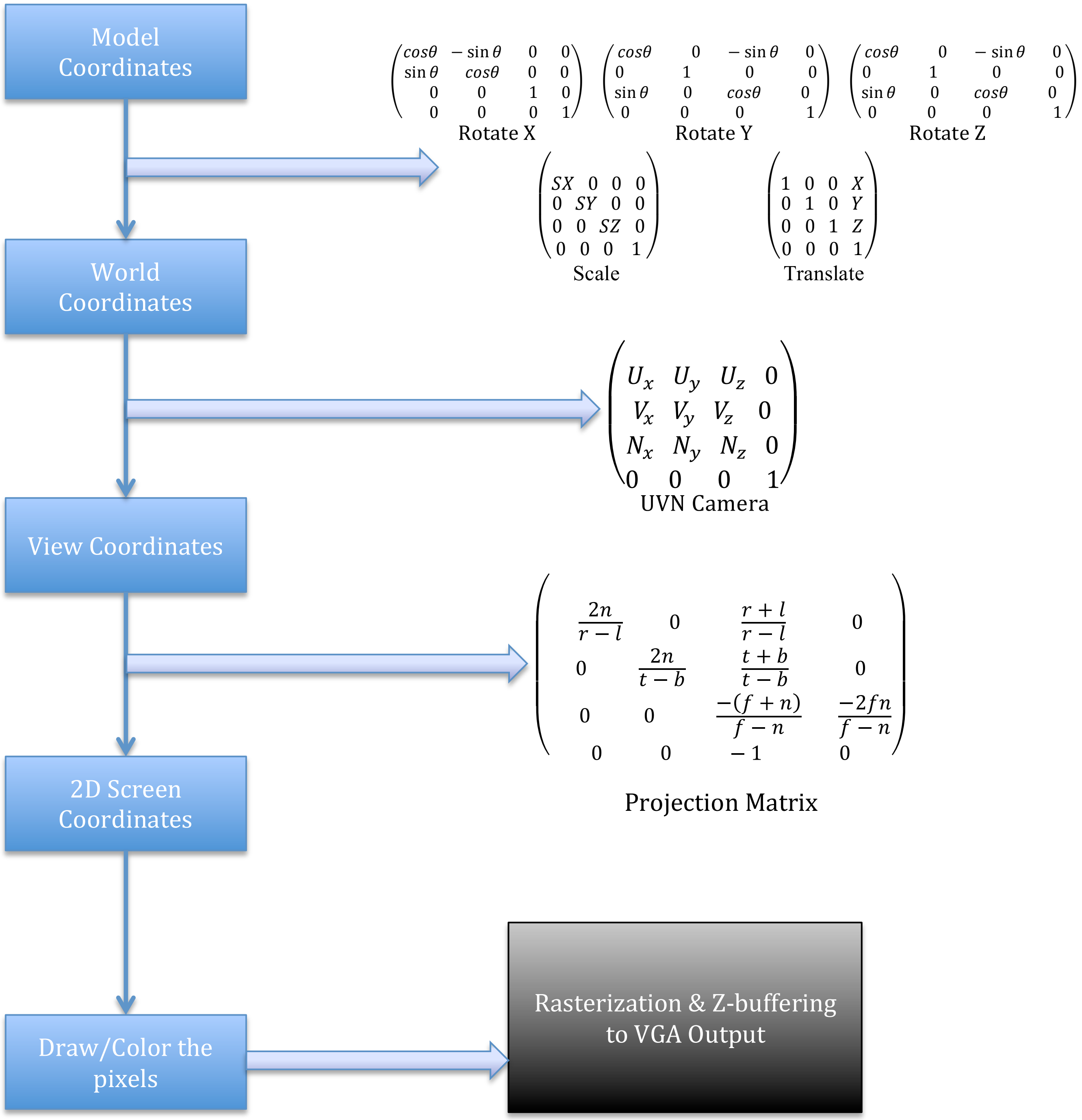


Figure 2: Altera Reference Design

Transformation Pipeline

A 3D object is modeled using a collection of vertices in 3D space (x,y,z). The vertices can be connected to form the surfaces of the 3d object. The surfaces can be filled in using any number of methods according to the model. To project a 3D object onto a 2D screen, the models vertices undergo a series of matrix multiplications to transform the 3D vertices into 2D coordinates that can then be displayed on the screen. The rendering pipeline we will use for our project is shown in Figure 3



$$\begin{pmatrix} cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} SX & 0 & 0 & 0 \\ 0 & SY & 0 & 0 \\ 0 & 0 & SZ & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 0 & X \\ 0 & 1 & 0 & Y \\ 0 & 0 & 1 & Z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} U_x & U_y & U_z & 0 \\ V_x & V_y & V_z & 0 \\ N_x & N_y & N_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Figure 3: Rendering Pipeline

In our model, the vertices are represented using four coordinates (x, y, z, w). The addition of the ‘w’ coordinate makes these Homogeneous coordinates, which are used to simplify the operations, involved for 3D rendering. The ‘w’ can be used for other rendering purposes such as shadows, which will be out of the scope of this project. Here we describe each of the coordinate blocks in the pipeline and the matrix operations that transform the vertices from one domain to another.

i. The Model Coordinates

Each surface in the 3D model to be displayed is considered to be a collection of triangles. The triangles are defined by a set of vertices. To start with, the x,y,z coordinates are defined relative to the center of the object which is a vertex at (0,0,0). We want to be able to rotate, translate and scale our model. Each of the above transformations requires a matrix to be multiplied with the vertices. The model matrix is a product of the following matrices; the 3 rotation matrices (for x, y and z axes) scale and translate matrices.

ii. The World Coordinates

After model matrices are applied to the vertices, the model went from being in Model Space (all vertices defined relative to the center of the model) to World Space (all vertices defined relative to the center of the world).

To understand the view matrix, consider a camera moving around your world space. If the camera looking at the entire world rotates 3 units to the right, it is equivalent to your world space rotating 3 units to the left. The view matrix defines which point in world space is being viewed. In other words, it describes the direction the camera is pointed in. In our project, we plan on keeping the camera/view fixed while moving the object/world.

The method we will use to generate the view matrix is the “UVN Camera”. The details of this method can be found in [1] and [2].

The U, V and N vectors that make up the view matrix are described according to []

> N - The vector from the camera to its target. Also known as the 'look at' vector in some 3D literature. This vector corresponds to the Z-axis.
>
> V - When standing upright this is the vector from your head to the sky. If you are writing a flight simulator and the plane is reversed that vector may very well point to the ground. This vector corresponds to the Y-axis.
>
> U - This vector points from the camera to its "right" side". It corresponds to the X-axis.

For our project, we plan to place the plate object along the z-axis so that the movement of the camera will be minimal. The world coordinates are transformed into the view coordinates (all vertices defined relative to the camera) after multiplying the vertices by the view matrix.

iii. The View Coordinates

Once we have the view coordinates, we need to convert the 3D Coordinates into 2D Coordinates that represent the 3D object being projected onto a 2D screen. This requires another matrix transformation. The matrix generated at this step is called the Projection matrix. It defines the window that the 3D coordinates will be projected onto. There are 6 main parameters: near, far, left, right, top and bottom. Near and far are used to specify the camera's field of view, that is, how near and far the camera is able to see. Left, right, top and bottom represent the boundaries of the display in their respective directions. Note that the left and right boundaries are dependent on the angle of view of the camera, which can be specified in the software. The top and bottom boundaries are dependent on the aspect ratio of the display. These parameters are listed in the equations below and the dependencies are reflected in these equations.

n = near
size=near*tan (angle_of_view/2.0)
left= -size = *l*
right= size = r
bottom= -size/aspect_ratio = b
top = size / aspect_ratio = t

See references [3] and [4] for details on how the projection matrix is derived.

iv. The 2D Coordinates:

After the projection transformation, we have the 2D space vertices, which can be used for displaying the model on screen. To fill in the model's surfaces, we use a basic rasterization algorithm that includes z - buffering. A set of three vertices represents a triangle, which is the primitive shape that makes up the surface of an object. Figure 4 shows a sample triangle defined by vertices P1, P2 and P3.

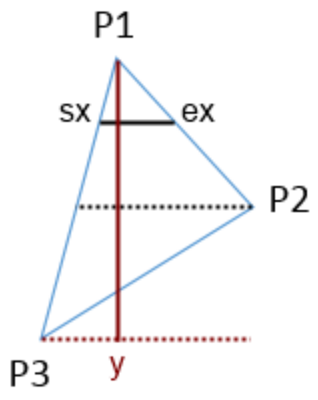


Figure 4: Triangle defined by 3 vertices

The three vertices allow us to calculate the gradient of lines P1-P2 and P1-P3. Using these gradients, we can determine sx and ex, which represent the start location and end location of the triangle, for each value y. Similarly, we can calculate a sz and ez which represents the start and end z-coordinates that correspond to the sx and ex values. Given these values, we can determine the boundaries of the triangle that needs to be rasterized. The addition of the z coordinate can tell us if particular pixels should reside in front of other pixels, which maintain the 3D look of the object when projected on the screen.

The details of the algorithm we are using are at [5].

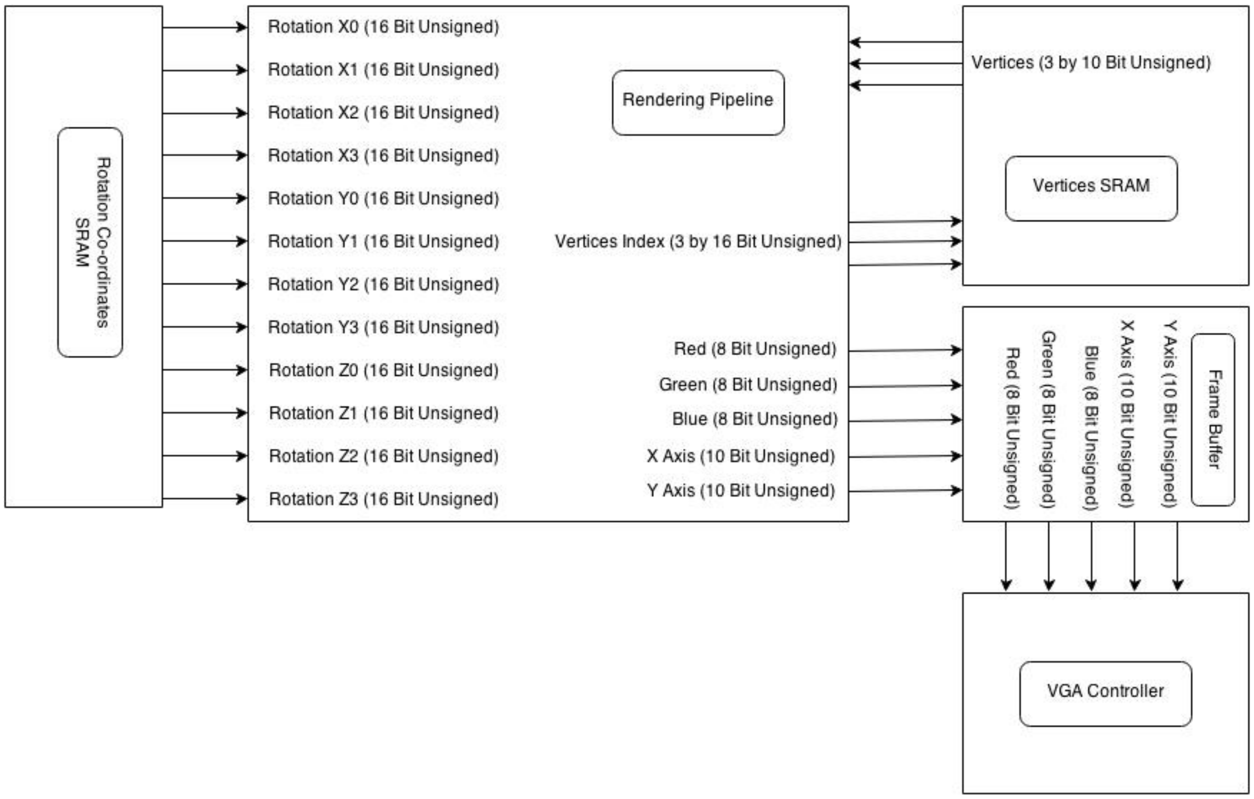
For sequencing the mathematical transforms, we can choose from the following options:

Avalon ST
Ready and valid signals (sink, src)
Tag method

The ready and valid signal method seems the most straightforward, and we plan on sequencing the transforms with the help of those signals.

Registers

This is a preliminary list of I/O registers we plan to implement. The hardware will be modeled as a standard memory-mapped I/O with the device driver reading/writing to these registers.


Rotation Co-ordinates SRAM
Rotation X0 (16 Bit Unsigned)
Rotation X1 (16 Bit Unsigned)
Rotation X2 (16 Bit Unsigned)
Rotation X3 (16 Bit Unsigned)
Rotation Y0 (16 Bit Unsigned)
Rotation Y1 (16 Bit Unsigned)
Rotation Y2 (16 Bit Unsigned)
Rotation Y3 (16 Bit Unsigned)
Rotation Z0 (16 Bit Unsigned)
Rotation Z1 (16 Bit Unsigned)
Rotation Z2 (16 Bit Unsigned)
Rotation Z3 (16 Bit Unsigned)
Rendering Pipeline
Vertices Index (3 by 16 Bit Unsigned)
Red (8 Bit Unsigned)
Green (8 Bit Unsigned)
Blue (8 Bit Unsigned)
X Axis (10 Bit Unsigned)
Y Axis (10 Bit Unsigned)
Vertices (3 by 10 Bit Unsigned)
Vertices SRAM
Frame Buffer
Y Axis (10 Bit Unsigned)
X Axis (10 Bit Unsigned)
Blue (8 Bit Unsigned)
Green (8 Bit Unsigned)
Red (8 Bit Unsigned)
VGA Controller

The 'Rendering Pipeline' block consists of multiple multipliers to perform the matrix multiplications required. The dedicated multipliers present on the board will provide this functionality. Each element of the rotation matrix is fed into the pipeline as a 16 bit word and operations are performed in parallel where possible.

All the modules are clocked at 50MHz.

Software prototype

We wrote a software prototype to test and verify this pipeline method. We use a program called Blender to generate the vertices of our 3D model. We used the tutorial [] as a guideline for our prototype. The tutorial was written in C# however we chose to implement our software prototype in python. The numpy module is used to do the matrix operations while the pygame module was used to draw pixels on the screen. No higher-level graphics libraries were used in our prototype. All of the transform matrices and matrix multiplications were generated and performed from scratch. We successfully implemented our pipeline in software, and Figure 5 is a sample object rotated in space using our software pipeline. Note that the colors used during rasterization were random to simply be able to distinguish between the different surfaces of the object.

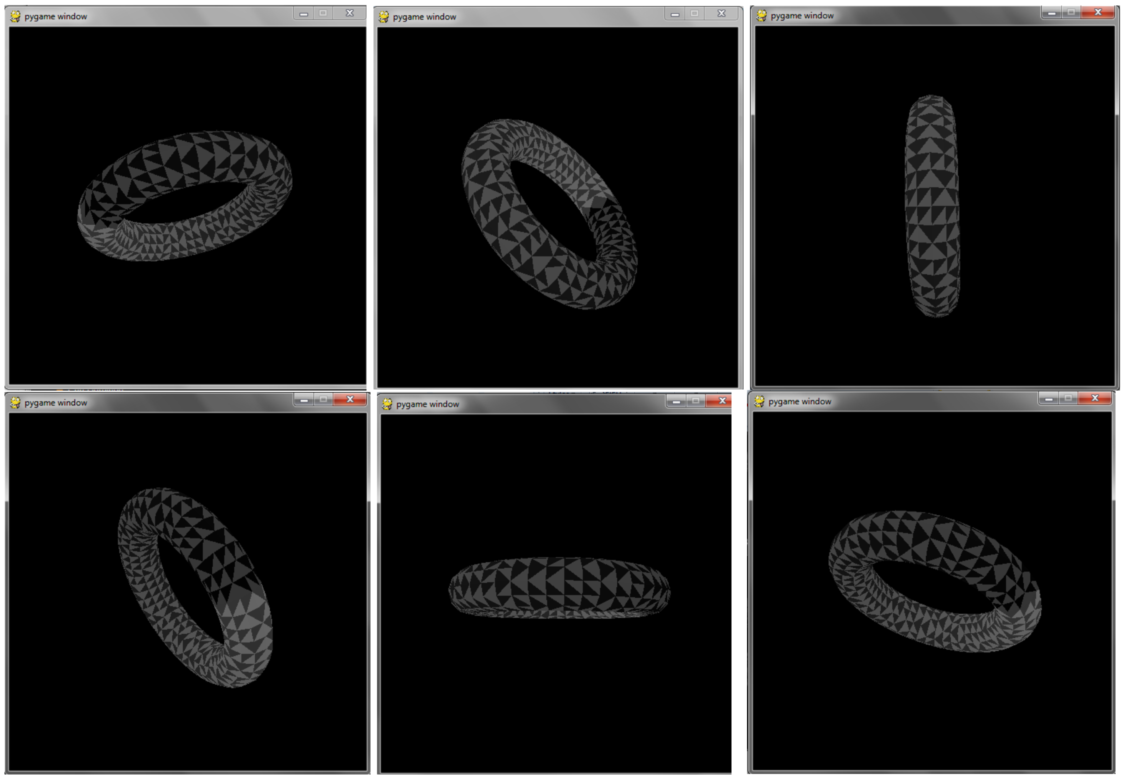
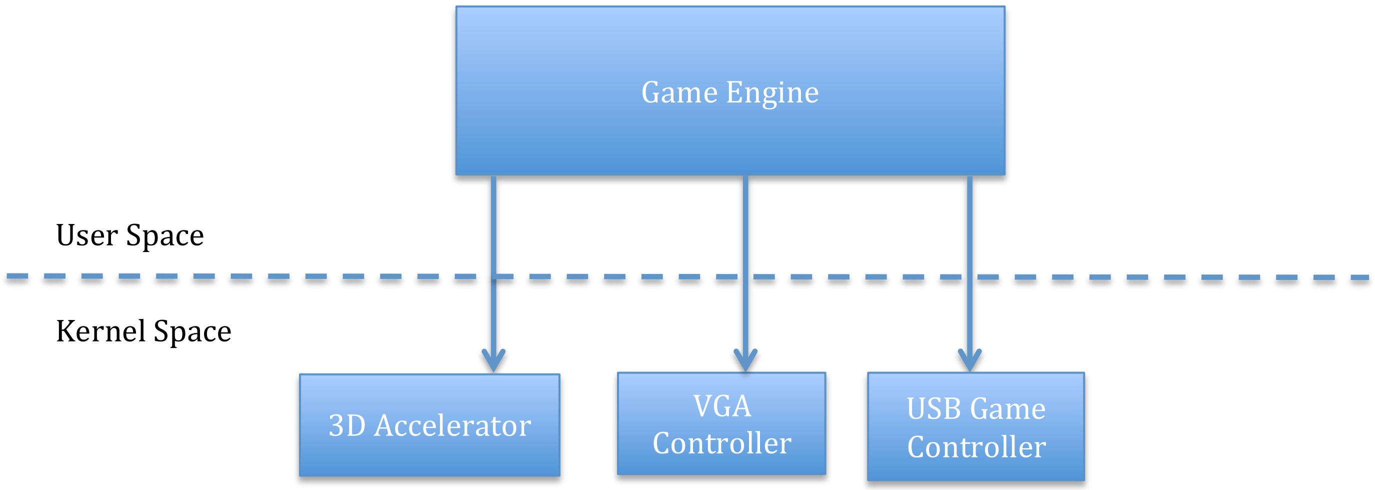


Figure 5: Software Pipeline Sample

- **VGA CONTROLLER AND OUTPUT**

The VGA module gets the input from the frame buffer, to display the graphics pixel by pixel on the screen. As of now, we plan on sticking to the default screen resolution of 640x480.

3. Software Structure



The user-space component consists of the software structure of the project. The software mainly handles the game logic. In our project, we have divided the software part into the controller and the graphics. The controller decides the movement of the plate. While doing this, the graphics play a significant role to display a nice rotating plate.

- Controller

This block deals with the USB Game controller. As this is a physically connected device, a device driver is necessary on software. We plan on using the PS3 controller, which has an inbuilt accelerometer. This segment detects the direction in which the user wants to move the plate, and interfaces this information with the device driver for the controller. The information fed by the controller leads to the calculation of the rotation angle, the depth, the vertices and faces.

- Graphics

The Graphics module defines the matrices necessary for the transformation from 3D to 2D, which is done by the 3D Accelerator of the hardware. The controller reads the directions. The graphics module interprets this in terms of rotational matrices. This block also deals with reading of the vertices to be rotated, and the faces defined by these vertices. All the transformation matrices are identified in this block and sent over to the hardware interface for calculation, processing and display.

4. Initial problems to address

- The board has only 224 multipliers. First, we have to get the mathematics involving the matrices working, and fit the overall transformation within this multiplier limit. Initial target would be to do a multiplier test, and get the mathematics to work.
- The format of the vertices is another issue. We need to figure out the number of bits necessary to represent the vertices. For now we will assume 16-bits for the vertices and 32-bits for the intermediate multiplication values. The 1GB DDR3 will be used to hold both intermediate calculate values and our output frame buffer, which should be sufficient memory for our purposes.

5. References

[1] http://ogldev.atspace.co.uk/www/tutorial13/tutorial13.html
[2] http://blog.db-in.com/cameras-on-opengl-es-2-x/
[3] http://www.songho.ca/opengl/gl_projectionmatrix.html
[4] http://blog.db-in.com/cameras-on-opengl-es-2-x/
[5] http://blogs.msdn.com/b/davrous/archive/2013/06/21/tutorial-part-4-learning-how-to-write-a-3d-software-engine-in-c-ts-or-js-rasterization-amp-z-buffering.aspx