Real-Time Interactive Block World Preliminary Report

1st Bowen Wu
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
fsdi321@mit.edu

2nd Ali Cy
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
califyn@mit.edu

Fig. 1. A world consisting of a grid of cubes being rendered by the FPGA

***Abstract*—We present a design for a real-time interactive block world that is rendered entirely on an FPGA, which implements a graphics pipeline not unlike the modern OpenGL graphics pipeline, as well as a physics engine to allow the world to be interacted with in real-time. The rendered frame is then transmitted over HDMI at 74.25MHz to render the world at 60fps at a resolution of 320x180, upscaled to 1280x720. It utilizes the Bresenham Line Draw algorithm to render triangles and lines efficiently into one of two frame buffers and implements custom fixed-point vector math modules to perform perspective projection. It also features a barycentric coordinate computation module to perform screen-space vertex attribute interpolation, as well as a custom physics engine to handle movement, collision, and world editing. The main challenges with implementing a rendering engine include but are not limited to lack of native floating point support, limited memory, and fixed point precision loss.**

Keywords—Computer Graphics, Rendering, Real-Time, Interactive

I. WORLD REPRESENTATION

To render a block world, we must first define how it is to be represented in System Verilog.

TABLE 1 World Block RAM Contents

Memory Size	Content	Details
4900 bits	100 Cube Positions	1 valid bit 3 * 16 bit for (x,y,z) coordinates of the cube in world space. We only allow integer positions.

II. FRAME BUFFER

A. *Specifications*

- 320x180
- Unsigned 8bit format, 3R3G2B

B. *Double Buffer*

To allow for simultaneous HDMI transmission and rendering, we will make use of a double buffer system, where the front buffer is a finished frame, while the back buffer is cleared and used for rendering the next frame.

At the start of a new frame signal from the HDMI signal generator, the buffers will be swapped, and the new front buffer will be cleared and drawn to, while the back buffer will have its write port disabled and output pipelined to HDMI output. In total, we will require $320 * 180 * 8 * 2 = 921600$ bits of data to store both buffers.

III. DEPTH BUFFER

A. *Specifications*

- 320x180
- Unsigned 16bit format

B. *Details*

To render triangles in the correct order without requiring the polygons to be sorted and drawn in a specific order, we make use of a depth buffer to enable depth testing. This buffer has the same resolution as the frame buffer but uses 16 bits to store the depth of the triangles rendered; depth is defined as distance of where the pixel is in 3D space from the camera.

The depth values are fixed points, with the upper 8 bits representing the integer part of the depth and the lower 8 bits representing the fractional part. High precision is needed to render objects close together in the correct order.

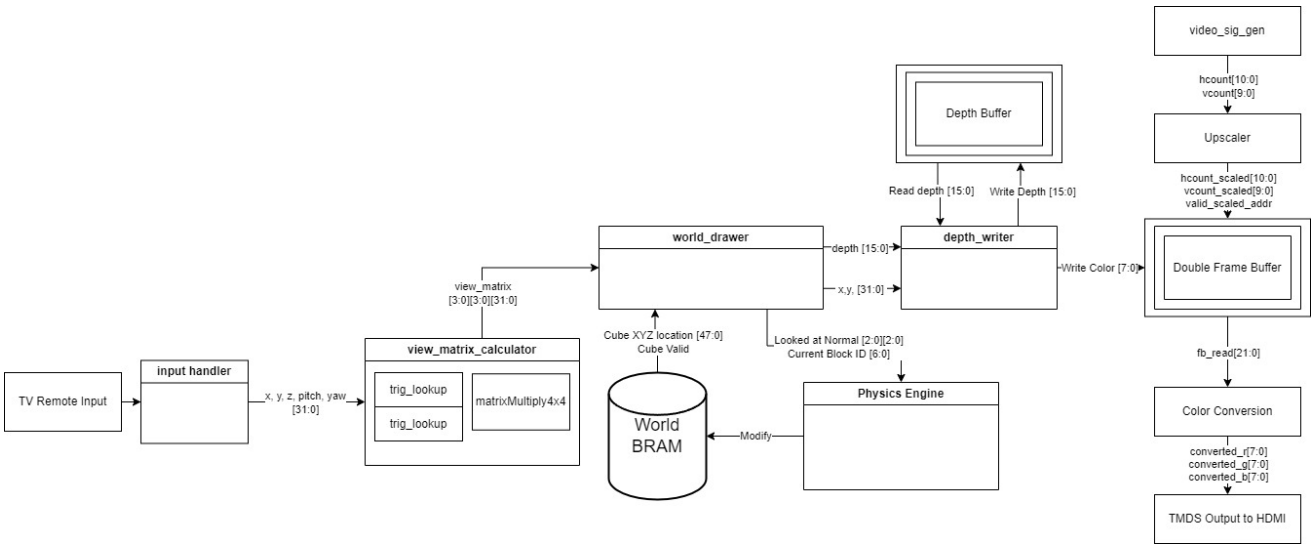

Fig. 2. High Level Diagram of the Modules

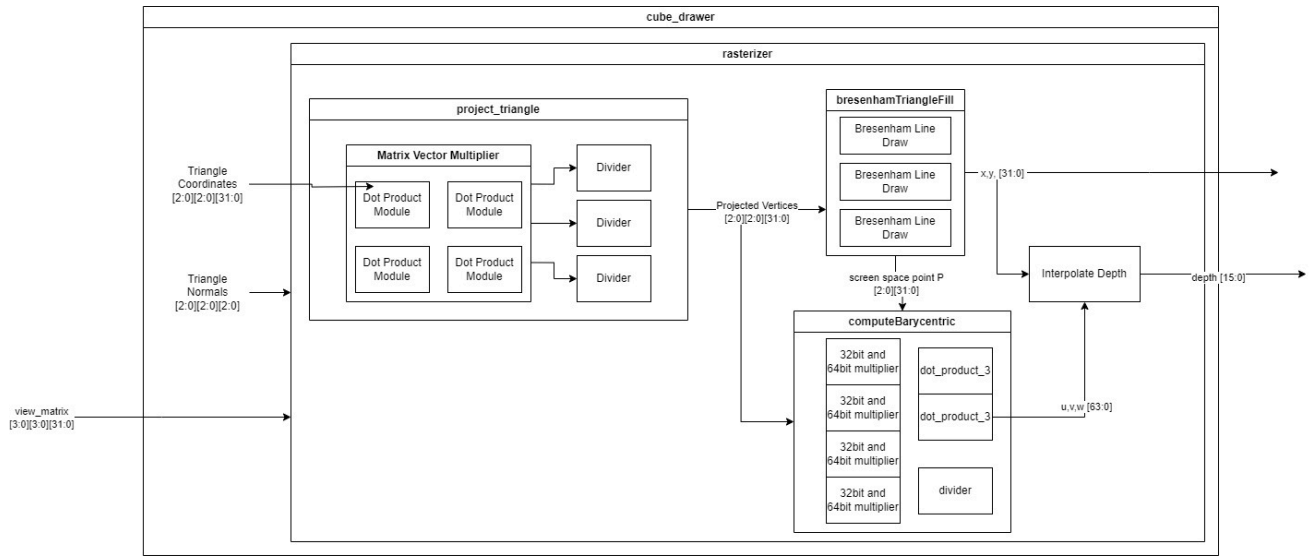

Fig. 3. Detailed block diagram of the cube_drawer

We do not need to use a double buffer for this, since we only need one working depth buffer to store the depth of the triangles rendered so far to compare against when rendering new triangles. In total, the depth buffer will require $320 * 180 * 16 = 921600$ bits of the BRAM.

IV. RENDERING

The rasterizer models after the modern OpenGL rendering pipeline. Valid cubes to render are found by iterating through the world BRAM. To render an object composed of triangular meshes, each triangle in the object is rendered in sequence. The triangle is first projected from its object space to screen space using vector math, then it is drawn into the frame buffer with depth testing, such that the triangles have correct overlap according to their positions in the world.

A. Main State Machine (Bowen)

The primary rendering state machine works as follows:

1. IDLE: wait for *new_frame* signal to go to INIT
2. INIT: swap the front and back buffer and initialize clearing the new front buffer. Go to CLEARING.
3. CLEARING: clear the front buffer and depth buffer over several cycles until we reach the end of the buffers. Go to DRAW
4. DRAW: pulse *start_render* signal when view matrix finishes computing and render cubes into the world using the *world drawer* module. Wait for *render_done* signal from *world drawer* module and go to DONE.
5. DONE: return to IDLE

B. World Drawer (Ali)

To draw multiple cubes and fill up the 3D world, we wrote a **world drawer** module that encapsulates the cube drawer module.

It iterates through the world BRAM, terminating when the next cube is at WORLD_SIZE (which we set to 128, so maximum 128 blocks). It retrieves *x, y, z* of each block from the world BRAM as well as whether the block is valid. If the

block is valid, then it proceeds by enabling the cube drawer on the read *x, y, z*.

C. Cube Drawer (Ali)

The **cube drawer** module to draw a cube given its (*x,y,z*) position in world space and a view matrix. The cube is hard coded as 12 triangles with their coordinates in object space along with their normal data. They are then rendered into screen space with the rasterizer module via the projection and triangle fill algorithms.

D. View Matrix Calculator (Ali)

The **view matrix calculator** computes the view matrix given the location (*x_input, y_input, z_input*) and orientation (*rot_angle*(yaw), *side_angle*(pitch)). First, it obtains the cosine and sine of the angles through two 8-bit lookup tables. Then, it instantiates two matrix multipliers to multiply the yaw matrix, pitch matrix, and translation matrix together. It also outputs the vector direction the camera is facing for the world controller.

E. Vector Math (Bowen)

To make the most out of each cycle, we implement vector math modules in a pipelined manner. We have the following modules:

1. Vector4 Dot Product: 3 cycle delay
2. Vector3 Dot Product: 3 cycle delay

Matrix math modules are not pipelined, but make use of the pipelined vector math modules to achieve low latency.

1. 4x4 Matrix Vector4 Multiplier: 6 cycle delay
2. 4x4 Matrix Multiplier: 9 cycle delay

F. Rasterizer: Projection (Bowen)

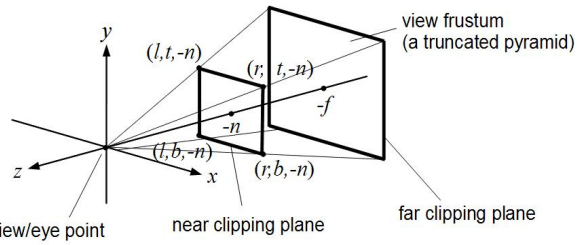

Fig. 4. The view frustum used to render a 3D world. Objects within the near and far clipping plane are rendered, while anything outside of them are "clipped" and discarded from the final image.

When rendering a triangle, first we must project it into screen space. Our triangle projection module, it follows these steps in an FSM:

1. The triangle's vertices are transformed from *object space* to *world space* by multiplying its vertices with its **Model Matrix.**
    a. Each vertex is represented by a 4x1 vector {*x,y,z,w*}. for perspective division, *w*=1.
    b. Each matrix is 4x4 in dimension.
2. The resulting vectors are transformed from *world space* to *view space* by multiplying them with the **View Matrix.**
    a. This matrix is recomputed at each frame to allow for movement and rotation of the camera, with camera position and yaw/pitch as input, computed over several cycles (Ali).
    b. Since computing rotation requires trigonometry, we opted to make use of a lookup table to quickly compute *sin* and *cos* values.
3. Results from (2) are then multiplied with the **Projection Matrix** to transform them to *clip space.*
    a. Because we do not intend to modify the value of the near or far plane, nor change the field of view, the projection matrix is hard coded to have:
        i. **FoV**: 45 degrees
        ii. **Near Plane**: 0.1
        iii. **Far Plane**: 100
4. In *clip space*, we then compare the *x,y,z* coordinate values to *w*.
    a. If $abs(z) > w$ for any vertex, then the triangle has parts outside of the near or far plane and is discarded.
    b. If $abs(x) > 2w \,||\, abs(y) > 2w$ for all three vertices, then the triangle is discarded.
5. Triangles that are not discarded are then transformed from *clip space* to *Normalize Device Coordinates (NDC)* using **perspective division**, where the *x,y,z* values are divided by *w*.
    a. As an optimization to reduce the number of divisions we need to do, we compute the reciprocal of *w* over multiple cycles via a

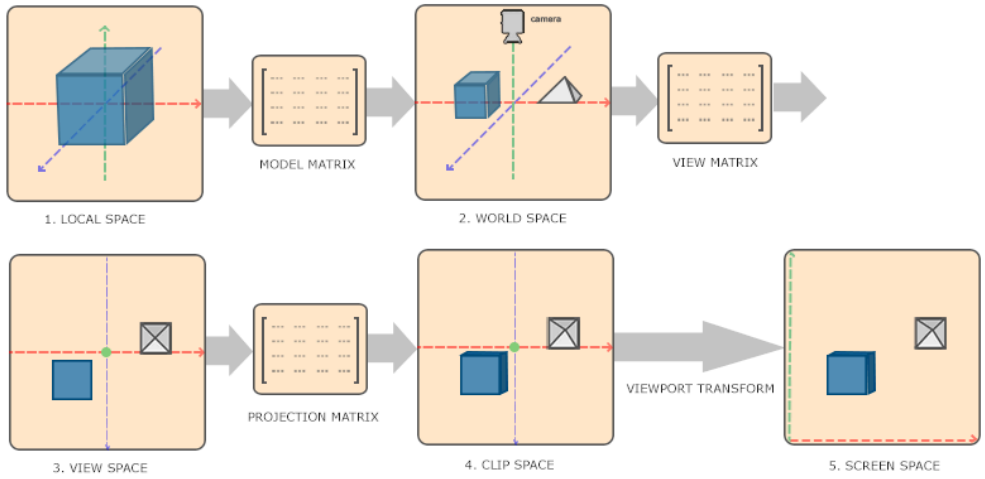

Fig. 5. Visualization of the rendering pipeline, courtesy of *learnopengl.com*

fixed-point divider. Then, *x,y,z* are multiplied by the result.

b. In our implementation, *z* values has the range of [0,1], and *x,y* values have the range of [-2,2]

6. Finally, the *NDC* coordinates are linearly mapped to screen coordinates.
   a. -1 in *x* and *y* of NDC space maps to 0 on the screen.
   b. +1 in *x* and *y* of NDC space maps to 320 and 180 on the screen, respectively
   c. 0 in *z* of NDC maps to 0.1, the value of the near plane.
   d. +1 in *z* of NDC maps to 100, the value of the far plane.

G. Rasterizer: Rendering Triangles (Bowen)

After a triangle is projected, its screen space coordinates are known and can be used to render triangles into the frame buffer. The typical approach is to draw horizontal lines within the triangle row by row starting from the top of the triangle, until the entire triangle is filled. To do this, we chose Bresenham's line algorithm [1] for its speed and simplicity, since it makes use of only integer addition, subtraction, and bit shifting to draw lines.

To draw and fill a triangle onto a 2D screen, where (0,0) is defined to be at the top-left corner, we use the following algorithm.

Algorithm 1: Bresenham Triangle Fill

1. Sort the triangle's vertices by *Y* value.
2. Start drawing two lines from the vertex with the lowest Y value to the other two vertices.
3. Whenever both lines have advanced 1 step in the *Y* direction, draw a horizontal line from the left line's position to the right line. Repeat from (2) until one of the two higher vertices is reached, in other words when a line has finished drawing.
4. Once a line has reached its endpoint, begin drawing a new line between the two higher vertices.
5. Repeat the same procedure as (3) between the two remaining lines until the triangle is filled, in other words when both lines are finished drawing.

H. Rasterizer: Barycentric Coordinates (Bowen)

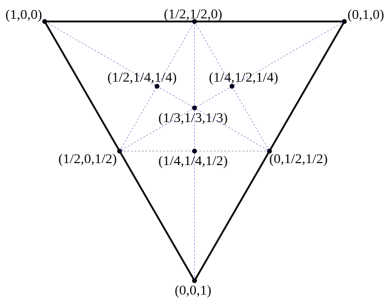

Fig. 6. Example of barycentric coordinates on a triangle.

Barycentric coordinates are necessary to perform interpolation of vertex attributes within a triangle. In triangles, barycentric coordinates are three numbers (*u, v, w*) that correspond to weight for vertices (*a, b, c*). These weights have the property that: $u + v + w = 1$, and are usually in the range of [0,1]

If all three weights are <1, then:

$$u * a_p + v * b_p + w * c_p = P,$$

where a_p, b_p, c_p represents the (*x, y, z*) coordinate of each vertex, and *P* represents a point that lies directly on the surface of the triangle.

This property of triangles can be used to linearly interpolate any attribute of the three vertices. In this case, we use it to compute the depth of each pixel within the triangle by linearly interpolating the depth of the 3 vertices.

We compute barycentric coordinates in a pipeline via the following algorithm, taken from Christer Ericson's book [2]. This computation occurs in parallel with Algorithm 1 in a pipelined fashion, with the output of Algorithm 1 as input.

Algorithm 2: Barycentric coordinates

1. Given a triangle with 3 vertices *a, b, c* in screen space with (*x,y,z*) coordinates, precompute the following:
   a. $v0 = b - a$
   b. $v1 = c - a$
   c. $d00 = dot(v0, v0)$
   d. $d01 = dot(v0, v1)$
   e. $d11 = dot(v1, v1)$
   f. $inv\_d00 = 1/d00$
   g. $inv\_d01 = 1/d01$
   h. $inv\_d11 = 1/d11$
2. Once precomputation is finished, compute the following in a pipeline to output a new set of barycentric coordinates per cycle:
   a. $v2 = p - a$
   b. $d20 = dot(v2, v0)$
   c. $d21 = dot(v2, v1)$
   d. $prod_a = inv\_d11 * d20$
   e. $prod_b = inv\_d01 * d21$
   f. $prod_c = inv\_d00 * d21$
   g. $prod_d = inv\_d01 * d20$
   h. $v = prod_a - prod_b$
   i. $w = prod_c - prod_d$
   j. $u = 1 - (v + w)$

Note that we are computing barycentric coordinates in screen space, so the z value is always set to 1.

This module was difficult to implement, because some of the values that it calculates would overflow a Q16.16 fixed point number. So, we had to make use of some high bit-width registers to store intermediate data, which required us to use a multiplier IP to keep our timing within constraints.

Further, we also suffer from precision loss because of the inherent nature of fixed-point numbers. So, when a point lands on the edge of a triangle or a vertex, the precision loss would result in negative values close 0 to be calculated. We

account for this by allowing negative barycentric coordinates, which would only result in slight deviation of the interpolated depth from its true value.

I. *Depth Testing (Bowen & Ali)*

During Algorithm 1, before we write a new pixel value to the frame buffer, we must check the depth value stored in the frame buffer and determine whether the new value goes in front of the existing pixel.

When a new frame signal is received from the HDMI signal generator, the depth buffer is cleared and initialized with values of 0xFFFF. Before a new pixel with depth (D_N) at screen coordinate (X_S, Y_S) is written into the front buffer, we read the depth buffer at (X_S, Y_S), and compare the existing depth value (D_E) to D_N. If $D_N < D_E$, that means the new pixel should be rendered as it is closer to the pixel.

However, for this to work we must be able to compute the depth of each pixel within the triangle. We interpolate the depth of the pixel using the barycentric coordinates computed by Algorithm 2.

Note: Since the output of Algorithm 2 is computing screen space barycentric coordinates, linear interpolation of depth, or any other vertex attributes, is **not perspective correct**. Perspective corrected barycentric coordinates can be computed but will require a pipelined division module to maintain throughput. This will result in some visual artifacts of colors and textures (if implemented) as the camera moves around. This is akin to the PlayStation 1, which also did not perform perspective correct interpolation of vertex attributes.

J. *Publishing Outputs (Bowen)*

To assist in the physics engine's task of placing new blocks and destroying the block in the center of the screen, the rasterizer will also publish two pieces of information after each frame is finished rendering:

1. The block ID of the current block that is currently at the center of the screen.
2. The normal of the face at the center of the screen.

K. *Upscaling (Bowen)*

Since our frame buffer is 320x180 but our HDMI output is written for 1280x720, we need an upscale module to upscale our frame buffer to the output resolution. This can be achieved by dividing the *hcount* and *vcount* signals from the HDMI video signal generator by 4.

L. *HDMI Output*

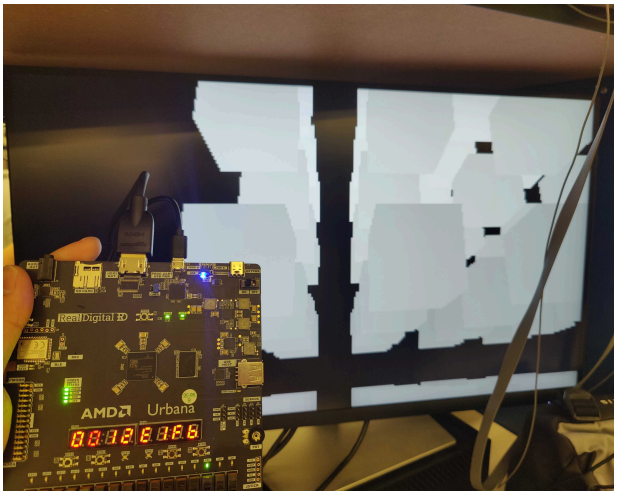

Fig. 7. The world being rendered at 60 fps

For outputting the frame buffer into a monitor over HDMI, we will re-use existing code from week4's lab assignment to generate the appropriate signals needed.

M. *Rendering Wireframes*

To render wireframes instead of triangles, we can simply draw 3 lines between the vertices of each triangle using Bresenham's line algorithm. The drawback of this naïve approach is that we will be drawing lines that are already drawn (overdraw). For now, solving overdrawing is out of the scope of this project.

Wireframe rendering is being worked on at time of writing.

V. PHYSICS ENGINE (ALI)

At each frame, the physics engine receives the following input:

1. User input. There are nine possible user inputs (view up, view down, view left, view right, forward, backward, jump, place, destroy).
2. A wire to read from the world BRAM.
3. From the physics engine—the previous *x, y, z,* normal for the center of the screen.

A. *Movement*

Movement is basic. The camera can move forward, backward, and vertically up or down on the Y axis. We integrated code written for week 3's labs to map inputs from a Roku remote to move and rotate the camera, as well as create and destroy cubes.

B. *View Controls*

The camera viewpoint can be controlled with pitch (horizontal) or yaw (vertical). The camera angle also controls which direction the player moves forward in. We enable wrapping around of pitch and yaw about 360deg=0deg.

C. *Placing and Destroying Blocks*

The x, y, z coordinates (as well as normal) published by the previous frame's rasterizer are used to judge if the proposed block is too far away, or if the proposed block collides with the player, the block is not placed. Otherwise,

Fig. 8. The world being rendered at 30 fps

the world BRAM is updated upon the new_frame signal from the HDMI signal generator.

First, the *xyz* of the block with the given block ID is retrieved. While iterating through the BRAM, the engine does two things. If a block is to be destroyed, it first obtains the *xyz* of each block, and if it is equal to the one to be destroyed, it removes it by setting its valid bit to zero. Second, if there are any invalid blocks, and a block is to be placed, the placed block is calculated and inserted at that position.

VI. EVALUATION

A. FPS

The primary performance of this project will be evaluated by maintaining a moving average of the number of cycles between each frame's finish signal and displaying it on the FPGA's eight segment modules. These values can then be manually converted to average frame time by dividing it by the clock frequency, then converted to average frames per second by taking its reciprocal.

We take the moving average over two frames and found that the average cycles between each frame is either $1237499\ (12E1FB)$ (Figure 6) or $2474999\ (25C3F7)$ (Figure 7), which is about 16.67ms and 33.33ms, which corresponds to 60 frames per second and 30 frames per second, respectively. This meets our initial throughput goal of at least 20-30 frames per second.

B. Timing

In terms of timing, we aimed to avoid negative slack. However, some small negative slack is acceptable since it would at most result in minor visual artifacts. We were able to stay within timing constraints, with a WNS (Worst Negative Slack) of 0.201, TNS (Total Negative Slack) of 0, WHS (Worst Hold Slack) of 0.014, and THS (Total Hold Slack) of 0.

C. Resources

In terms of resources, we aimed to stay within the budget of 75 BRAM blocks. We were successful in that we used 66 out of 75 BRAM blocks for the double frame buffer, depth buffer, and world BRAM.

D. Usecase and Goals

Since our design is written to render general triangle meshes, with support for vertex attribute interpolation, we can do more than just render cubes. In fact, given enough memory, we can render any triangle mesh with support for vertex colors and minimal changes to our design.

As for goals, we have reached our minimal goal. We can render our cube world at 60fps most of the time at 320x180, and we are able to allow the user to interact with the world by deleting or creating blocks.

We have also partially reached one of our stretch goals: an efficient barycentric coordinate calculator. Designing this module such that it pipelines the calculation allowed us to achieve a high throughput to not slow down the triangle rendering process and allowed us to implement vertex attribute interpolation to calculate depth per pixel.

VII. INSIGHTS & REFLECTION

Writing a rasterizer on an FPGA was indeed no trivial task. The most surprising part of this was learning that the triangle fill is the part of the rendering process that takes the most time, so optimizing our program for that was the most important part. Everything that came before we start drawing a triangle can take more cycles, but we must maintain a throughput of at about one new pixel every cycle once a triangle fill starts, otherwise the cost of filling triangles will significantly increase the more triangles we have to render. Pipelining the barycentric coordinate computation module was crucial to this, since there was no possibility of spending twelve cycles (the latency of the barycentric module) per pixel to compute them.

If we had another chance or more time, we would have pursued perspective correct barycentric coordinates to interpolate vertex attributes correctly. This would require a division to be performed on every pixel output after its screen space barycentric coordinate is computed, so we would have needed a pipelined division module to maintain our pixel throughput. The PlayStation 1 also did not have perspective correct interpolation, so if we would see some of the same visual artifacts present on the console.

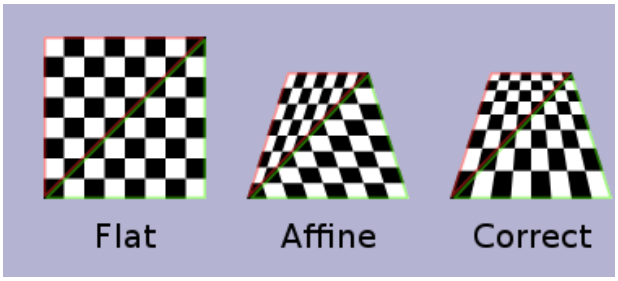

Fig. 9. Screen space (Affine) interpolation of UVs vs perspective correct interpolation. Ours and the PS1 would look like the Affine one.

Another direction we can take if there was another opportunity, was to write the FPGA such that it behaves like a GPU and draws pixels in parallel. Perhaps the FPGA itself can connect to an external device such as a computer and receive data over seral via Manta as commands to render things onto the screen.

Alternatively, we could have gone in the direction of rendering vector graphics instead of rasterizing triangles next time.

VIII. SOURCE CODE

The source code for the project can be found here: https://github.com/19829984/6.111_Final_Project

IX. CONTRIBUTIONS

Bowen worked on writing the skeleton of the rendering pipeline, specifically the vector math modules, projection module, triangle fill and line draw modules, barycentric module.

Ali worked on building the cube world and implementing interactivity, specifically the world BRAM, world drawer and cube drawer, view matrix calculator, sin and cos lookup table, and remote-control input.

Both Bowen and Ali worked on getting the depth buffer to work and contributed to writing the report.

Thanks for Project F for providing source code for a floating point divider: https://projectf.io/posts/division-in-verilog/

We also made use of the Multiplier IP from Vivado to pipeline high bit-width multiplication within the barycentric module.

ACKNOWLEDGMENT

Thanks to Project F for inspiring this project and providing reference material for graphics on a FPGA: https://projectf.io/posts/fpga-graphics

REFERENCES

[1] J. E. Bresenham “Algorithm for computer control of a digital plotter”, IBM Systems Journal. 4 (1): 25–30, 1965

[2] Christer Erison “Real-Time Collision Detection”, 2004. https://gamedev.stackexchange.com/questions/23743/whats-the-most-efficient-way-to-find-barycentric-coordinate