

# Simple Graphics Processing Unit on an FPGA

Anthony Bogedin, Michael Lohrer

Electrical and Computer Engineering Department  
School of Engineering and Computer Science  
Oakland University, Rochester, MI

e-mails: ajbogedi@oakland.edu, mflohrer@oakland.edu

**Abstract**—A Graphics Processing Unit (GPU) is important for many user interface based systems. By offloading a specific set of mathematically intense image generation operations from the Central Processing Unit (CPU), it frees up processing time for more general operations. In this project we developed a subset of these operations to run on the ZYNQ PL hardware and interfaced this newly created Simple GPU with the ZYNQ’s built in ARM microprocessor. The first and secondary goals were achieved successfully; the Simple GPU was capable of projecting CPU-designated three dimensional coordinates onto a display and optionally draw lines between them. Due to time constraints, the system lacks stability in several cases and future work could center on correcting these design flaws. The system could also be extended with more advanced functionality, such as color interpolation, triangle filling, and UV mapping.

## I. INTRODUCTION

An increasingly common requirement for many embedded systems is the necessity of providing a visually appealing interface for a user. Since a CPU must perform the primary function for a system, these older interfaces were typically limited to fixed function displays or basic text displays. Newer systems overcome this limitation by adding a coprocessor, the GPU, which handles the generation of images for user feedback.

A GPU consists of two primary functions. First, as a vector processor, it performs the *four* dimensional math necessary to project *three* dimensional points onto a plane. Second, as a pixel processor, it directly accesses memory in order to store correctly colored pixels onto a frame buffer that can be displayed on a screen without CPU involvement. A more feature rich system would provide more specific functionality under these two categories, but the focus of this project was to produce a system with the minimum number of component necessary to provide the basic functionality of a GPU.

The Simple GPU consists of a several stage pipeline, consisting of abstract mathematical functions and

graphics focused algorithms not covered in class. Namely, a four dimensional matrix multiply, a divider, a transform from normalized camera space to screen dimensions, a memory address calculator, and Bresenham’s line drawing algorithm. This system was developed in VHDL for the purpose of being placed on the programmable logic of the ZYNQ chip, a CPU-FPGA hybrid. Because this was designed for an FPGA and consisted of many mathematical operations, the components were written to operate on fixed point numbers as the LUT requirements for floating point support on an FPGA are quite large.

The inputs to the GPU are the memory address of the current frame buffer, the Model View Perspective (MVP) matrix and the vertices to draw. The GPU transforms the vertices into the camera’s view, and then calculates the address to store the output. The output of the GPU is stored in a frame buffer in the DDR memory onboard the Zybo. This frame buffer is then read by the display controller to show on the screen. The display controller was created using IP cores from Xilinx [2] and Digilent [3-4].

To send data to the GPU from the processor, two different interfaces are used. An AXI Lite interface provides easy configuration of parameters that do not change often, such as the MVP matrix. A Slave AXI Full interface allows sending of vertices and their color quickly to the GPU. Finally, a Master AXI Full interface allows sending pixel data straight to DDR memory through the High-Performance AXI ports of the Zynq PS.

## II. METHODOLOGY

Each major component in the System pictured in figure 1.

### A. Matrix Multiplication

A matrix multiplication is a binary operation that consists of multiplying every row of one matrix with the column of another matrix element wise, then summing the

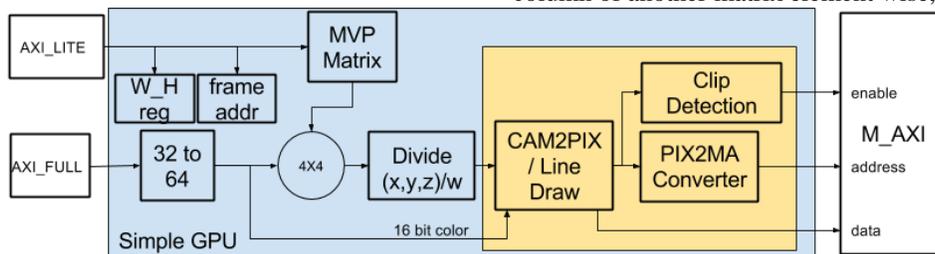


Figure 1. Simple GPU Architecture

results of each these operations to create a new entry into the output matrix. For the purposes of this project that consisted of multiplying a 1X4 matrix (the vertex) and the 4X4 MVP matrix. The equation for this is presented below.

$$\begin{bmatrix} v_x & v_y & v_z & v_w \end{bmatrix} \begin{bmatrix} m_{xx} & m_{xy} & m_{xz} & m_{xw} \\ m_{yx} & m_{yy} & m_{yz} & m_{yw} \\ m_{zx} & m_{zy} & m_{zz} & m_{zw} \\ m_{wx} & m_{wy} & m_{wz} & m_{ww} \end{bmatrix} = \\
 [v_x m_{xx} + v_y m_{yx} + v_z m_{zx} + v_w m_{wx} \quad \dots]$$

A full 4X4 matrix is necessary to perform all the transformations necessary to move a vertex from world space to view space, as shown in [5-6]. The input vertices are only three dimensional, so they are all placed on the same plane in the fourth dimension. Choosing this plane to be one simplifies later logic. Since it was known that the vertices streaming in were going to take longer than one clock cycle to transfer, the matrix multiply operation was only partially parallelized. Figure 2 shows a single multiply accumulate unit. Four of these were placed in hardware to handle each column of the 4X4 matrix. The state machine shown to the left controls each of the multiply accumulate units. As shown this will take 4 clock cycles to complete a single matrix multiplication.

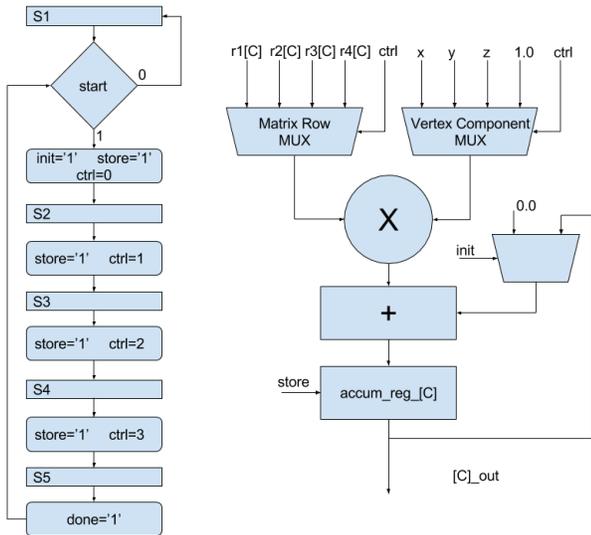


Figure 2. Matrix multiply state machine and datapath.

### B. Divider

The divider uses the pipelined version provided by Llamocca, but wrapped with some additional logic to handle negative numbers. The divider is important because the results while in view space after the matrix multiplication are not normalized to the view edges [6]. The scales can change dramatically depending on the distance between the camera and the vertex in question. Since the w component was known to be one, it was used

to normalize the other three. After this the values of x, y, and z will be between -1 and 1 if they are viewable.

The pipelined divider can take a new value each clock cycle, but the matrix multiply takes four clock cycles to complete. Thus, x, y, and z could be placed consecutively on a single divider and not slow the system.

### C. Camera to Screen Space

With the vertices in view space, they need to be scaled to the current screen resolution. Since the values are normalized this simply entails multiplying by half of the screens pixel width and height. It finishes with adding the width and height to bring the values positive if they are visible. Positive values are necessary for addressing into memory.

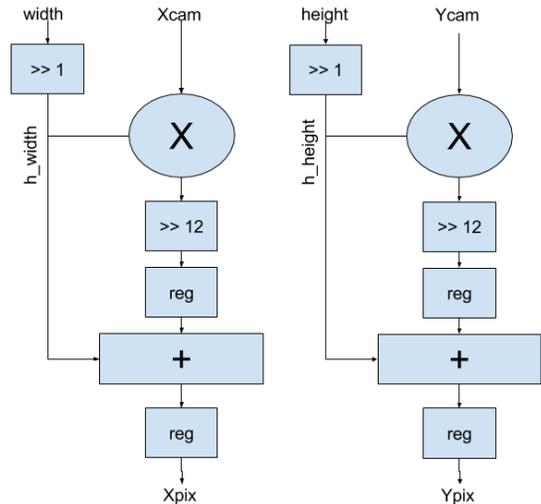


Figure 3. Camera to screen space datapath.

### D. Screen Clip Detection

As mentioned above, vertices can be off screen. Since drawing consists of writing to an array, it must not write outside the boundaries and corrupt main memory. The clip detection checks the position of the vertex and only sets the write signal high if the vertex is visible on screen.

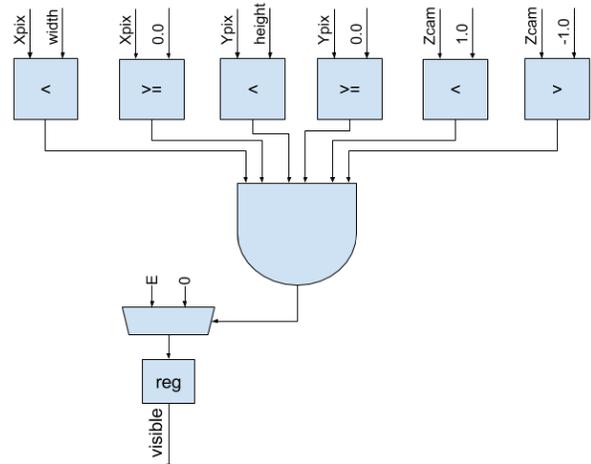


Figure 4. Screen clip detection datapath.

### E. Screen Space to Memory

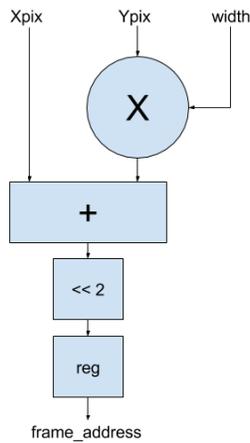


Figure 5. Screen space to framebuffer datapath.

The final step is to convert the screen coordinates of the vertex into an offset into an array. This is multiplying the y coordinate by the width to bring the index to the correct row and adds the x coordinate to bring the index to the correct column. This is a pixel index, but AXI is expecting a byte address. Since a pixel consists of 4 bytes, two least significant zeros are added to the address.

### F. Line Drawing

Line drawing takes two screen space vertices and draws a single pixel width line between them. The full algorithm is presented in the figure 6 pseudocode. "DIF" represents a replacement for error from normal fractional based methods, allowing this to be integer math. Every loop of the algorithm x is incremented and the difference between the y coordinates is added to DIF. When DIF reaches zero y is also incremented and the difference between the x coordinates is subtracted from DIF. A simple example shows that if the x distance is three times the y distance, DIF will reach zero every three increments of x. Thus a line with the proper slope of  $\frac{1}{3}$  will be drawn.

```

plotLine(x0,y0, x1,y1)
dx=x1-x0
dy=y1-y0

DIF = 4*dy - dx
y=y0

for x from x0 to x1
  plot(x,y)
  DIF = DIF + (2*dy)
  if DIF > 0
    y = y+1
    DIF = DIF - (2*dx)
  
```

Figure 6. Bresenham's line drawing algorithm pseudocode.

This was implemented using three data paths and a 5-state machine. First, the DIF reg is initialized with the negative of the x distance and is then added or subtracted as specified in the above pseudocode. The x and y outputs are then incremented in the cases also as specified in the above pseudocode.

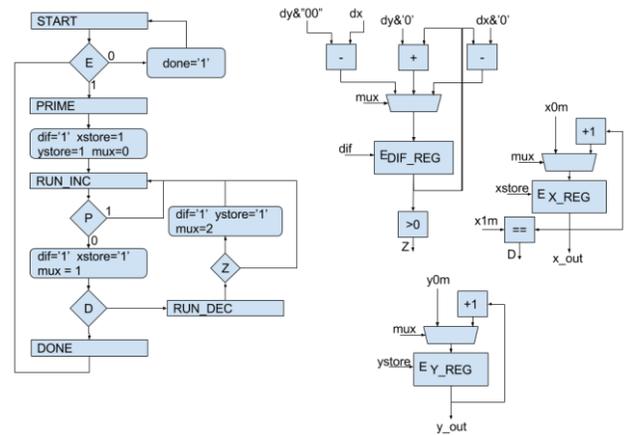


Figure 7. Line drawing state machine and datapath.

This has a major limitation. It only increments x, and y; limiting the direction the line can be drawn to the first quadrant of the coordinate space. Also since y only increments when x increments, the greatest slope is limited to 1, so lines can only be drawn within the first 45° of the coordinate space as shown in figure 8. Thus, the blue line cannot be drawn.

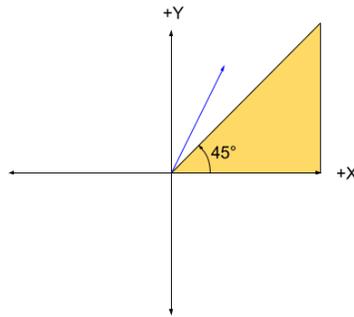


Figure 8. Line drawing limitation.

To overcome this limitation, the rest of the coordinate space needs to be folded into that 45° space. Then the output needs to be unfolded to the correct quadrant at the output for each pixel of the line. The block diagram is shown in figure 9. The important part is the three comparison operators, which detects the eighth of the coordinate space that contains the line.

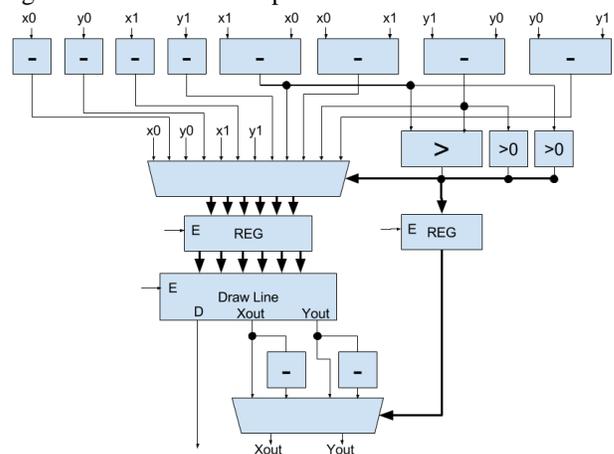


Figure 9. Folding MUX datapath.

A parameter of the Simple GPU component allows it to operate in either line-drawing or dot-drawing mode. This parameter is set at design-time so that the minimal resources are used for the desired operation in any given GPU.

### G. Configuration Interface

The Simple GPU needs several parameters to be set before it can be used, including the MVP matrix. Since the MVP matrix changes at most every frame, a high-speed interface is not required for setting its values. Therefore a set of registers with an AXI-Lite interface is used to set the 16 values of the matrix. Each matrix value is a 16-bit number, and so two could be packed into the same 32-bit register, but for simplicity only the lower 16 bits of each register was used. This also leaves room for upgrading to 32-bit fixed-point or floating-point numbers in the future without changing the interface. Two other parameters that are set over the configuration interface are the screen height and width. These are used in the conversion process to pixel coordinates, in detecting screen clip, and for generating the memory address of the pixel. The last parameter set with the configuration interface is the address to the start of the frame buffer in memory. Without this address, the GPU would not know where to store the pixels.

### H. Vertex Interface

The vertex interface has the job of receiving vertices from the CPU and storing them for use by the GPU. It achieves this by using a full AXI bus interface with support for bursts. Bursts would provide for faster transfers of vertices, however while the GPU supports burst writes our implementation has no DMA and so only single writes occur.

The vertices are sent in two 32-bit writes; the X and Y coordinate in the first and the Z and Color in the second. This limits the GPU to 16-bit coordinates and color, so the vertex-sending process would need to be modified to increase the resolution or world size. After a vertex is received on the AXI bus, it is stored in a FIFO and the read data and enable are exposed to the GPU.

### I. Memory Interface

The output of the GPU is an address into the frame buffer and a color to write to that address. In order to write the colors to the framebuffer in DDR memory quickly, a High-Performance (HP) AXI port is used. The HP ports only pass through one AXI interconnect in the PS system, which means faster access to the DDR controller than the General Purpose (GP) ports - about 3,200 Mb/s in both read and write. The AXI interface on the GPU side has to be a master, since the HP ports are slaves. Burst modes are not supported for the GPU master AXI port, since all burst modes require addresses either to be the same or sequential. The GPU will not generally be writing to sequential addresses so unfortunately burst modes are not possible without more buffering and logic. Two FIFOs are created to store the

addresses and colors generated from the GPU, with the read interfaces exposed to the GPU. The write interfaces are utilized by the master AXI, and whenever the both have data a write is started. The addresses stored in the FIFO are relative to the start of the framebuffer, so as a last step before writing, the address coming out of the FIFO is added to the frame buffer's base address to transform it into global memory address space.

### J. High-Level Schematic

A schematic of the important components in the entire system is shown in the figure 10. This is one possible configuration; as many GPUs as fit in the FPGA fabric could be used. Here we use two: one for dot-drawing and one for line-drawing. The dot-drawing GPU uses about 2000 LUTs, 1.5 36k BRAM tiles, and 7 DSPs, while the line-drawing GPU uses about 2500 LUTs, 2.5 BRAM tiles, and 7 DSPs. The two GPUs share an HP port, and the Display Controllers share another. Since the the display controllers are only reading the memory and the GPUs are only writing to it, they really could have all shared the same port and not sacrificed any bandwidth. GP0 is used for configuring both GPUs as well as the VDMA and Display Controllers - which is not shown in the diagram for simplicity. GP1 is used solely to send vertices to both GPUs.

The two display controllers allow for both VGA and HDMI output. Each VDMA receives a sync signal from its associated Display Controller to signal the beginning of a new frame. When a VDMA sees a pulse on the sync signal, it begins reading the framebuffer directly from DDR memory. There are three framebuffers, and which one the VDMA is reading from is configured by the CPU. The sync signal is also sent to the PS as an interrupt input.

Each frame, the CPU switches the framebuffer of the GPU, writes vertices to the GPUs, the GPUs write pixels into the new framebuffer, and the CPU waits for the Display Controller to signal an end of its current frame on the sync signal. Then the CPU updates the VDMA's to the new framebuffer, causing them to send the new frame to the Display Controllers. After each frame the CPU switches the framebuffer and starts over. The whole operation happens at a rate of 60 frames per second, and is capable of displaying up to 1080p resolution on both the VGA and HDMI outputs.

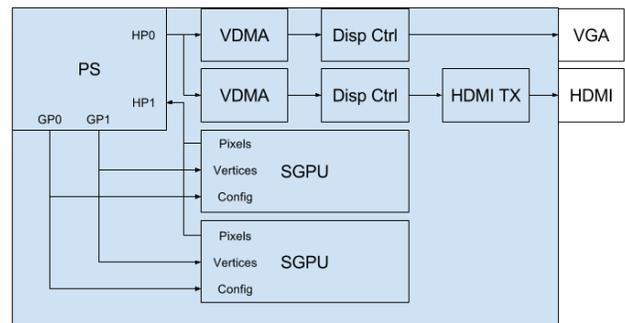


Figure 10. System level design.

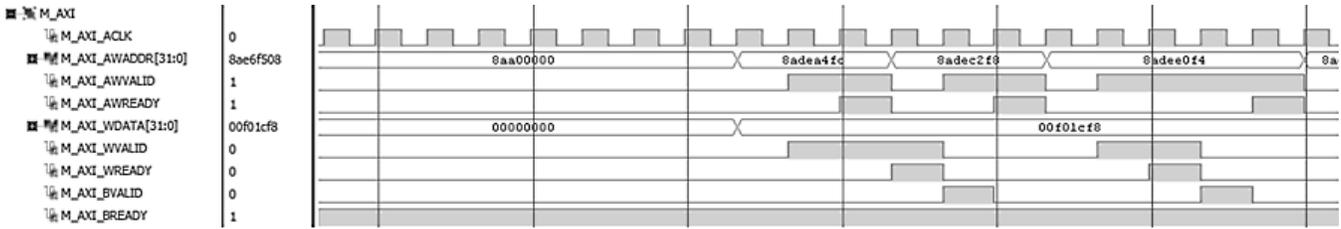


Figure 11. Master AXI write from GPU.

### III. EXPERIMENTAL SETUP

Each non-trivial component of the GPU that we created has an associated testbench. This includes the 4x4 matrix multiply, the line-drawing component, and one to test several representative values of the entire GPU. To test the entire GPU including AXI interfaces however requires a bit more work. A basic AXI lite master testbench model from [7] provided a good start, but the Simple GPU has a slave full and a master full interface as well. The slave full AXI interface - requiring a master full interface testbench to test - is easy to simulate because all of the extra inputs of the full over the lite interface can be set to constants. The master full AXI interface is a bit trickier. For that a slave AXI full model had to be created, which was based off of the example master AXI interface generated by Vivado. Using all three AXI interface testbench models, it is possible to simulate the entire GPU system from start to end. Using this testbench, the configuration registers were set up, and several vertices written to the GPU. The entire system was observed as well the final pixel output being written into the slave representing the HP ports. Below is shown an AXI write of a pixel from the GPU.

In addition to the simulations that were run, debugging of the hardware while it was running was used to verify the functionality of the AXI communication with the processor. Figure one shows the result of an AXI write on the Master AXI bus created in the Simple GPU component. In this case, the processor's High-Performance AXI port was the slave. The slave can be seen setting AWREADY, AREADY, and BVALID indicating a successful write.

Debugging of running hardware was achieved using Chipscope Pro debug cores, now also known as Vivado

ILA. This method places a debug core on-chip, which samples the desired signals every clock cycle. The sampled signals are put into a FIFO and can be sent to the PC for viewing. Since the bandwidth between the debug core and the PC isn't typically high enough to send all the signal's data, a trigger is used to tell the core when to send data. For our purposes, we set the trigger to the AWVALID signal so that we would see an AXI transaction occurring.

### IV. RESULTS

The resulting system was made up of two GPUs: one capable of dot-drawing and one capable of line-drawing. As can be seen in the video of the project, available at [1], several long lines and many dots can be drawn on the screen via the GPUs. Figure 13 shows a picture of the running system as well. The line-drawing GPU was used to form a cube, while one hundred dots were drawn with the dot-drawing GPU to create a sparkler effect. If the number of dots drawn increased too much, beyond 200, the colors became offset by a couple dots: so a dot that was supposed to be red may show up green for example.

Line drawing also showed issues in two cases. If lines with different colors were drawn, the colors would bleed into the next line. Also if lines were to go off the edge of the screen, a flickering effect was observed, with lines disappearing and sometimes connecting the wrong endpoints. These are most probably resulting from design issues with the FIFOs. If the two separate streams of data containing color and memory addresses were to get out of sync these effects could be expected.

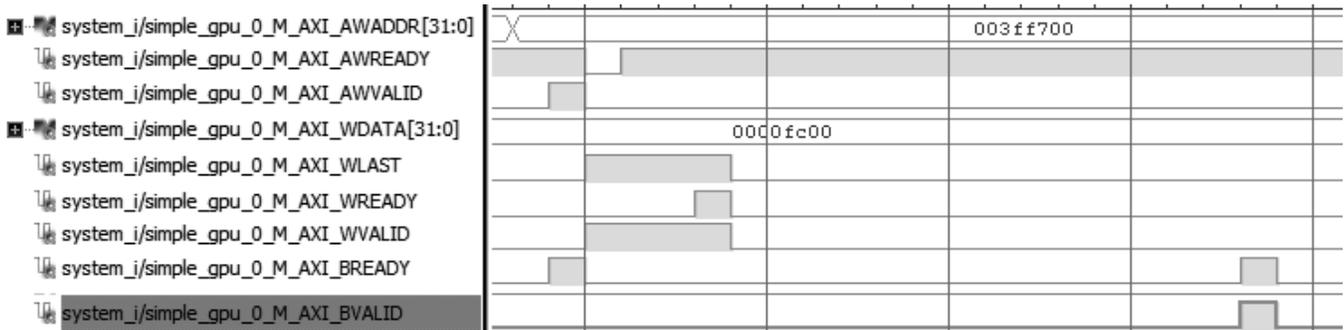


Figure 12. Master AXI write viewed on chip with Vivado ILA.

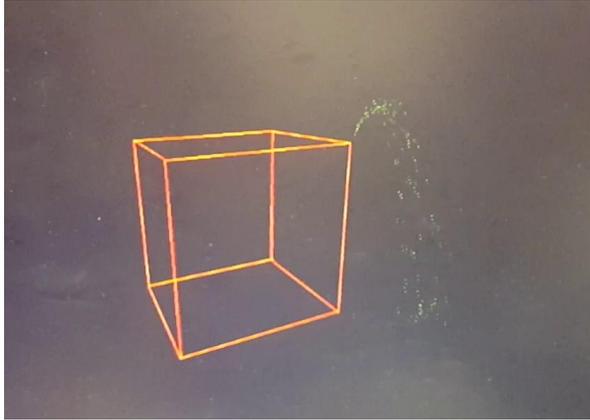


Figure 13. On-screen drawing of cube and sparkler.

### CONCLUSIONS

The next step for the GPU would be to allow for entire triangle drawing. Just as dot drawing was upgraded to line drawing, line drawing can be upgraded to triangle drawing. Three vertices would be input instead of two, and the pixel processor would have to fill in the area between the vertices. This is a big step up in complexity and in the time required to process the vertex stream.

Another improvement for the GPU system as a whole would be to add multiple pipelines of vertex calculations and pixel processing. For the single dot mode, this is not necessary as the CPU cannot send vertices fast enough to overflow a single GPU pipeline. But in the line drawing or proposed triangle drawing modes, the pixel processor can overflow its FIFOs. Adding more GPU pipelines would allow sharing of the workload between the multiple pipelines.

### REFERENCES

- [1] Video of project: [https://drive.google.com/open?id=0B4z\\_15QSfhUSOWNWWjhEZ24xTEE](https://drive.google.com/open?id=0B4z_15QSfhUSOWNWWjhEZ24xTEE)
- [2] Xilinx AXI VDMA datasheet [http://www.xilinx.com/support/documentation/ip\\_documentation/axi\\_vdma/v6\\_2/pg020\\_axi\\_vdma.pdf](http://www.xilinx.com/support/documentation/ip_documentation/axi_vdma/v6_2/pg020_axi_vdma.pdf)
- [3] Digilent Zybo Base System Design [http://www.digilentinc.com/Data/Products/ZYBO/zybo\\_base\\_system.zip](http://www.digilentinc.com/Data/Products/ZYBO/zybo_base_system.zip)
- [4] Digilent Vivado Library <https://github.com/DigilentInc/vivado-library/tree/master/ip>
- [5] OpenGL Projection Matrix [http://www.songho.ca/opengl/gl\\_projectionmatrix.html](http://www.songho.ca/opengl/gl_projectionmatrix.html)
- [6] OpenGL Transformation [http://www.songho.ca/opengl/gl\\_transform.html](http://www.songho.ca/opengl/gl_transform.html)
- [7] AXI Testbenches [https://github.com/Architech-Silica/Designing-a-Custom-AXI-Master-using-BFMs/tree/master/HDL\\_sources/Testbench\\_Sources](https://github.com/Architech-Silica/Designing-a-Custom-AXI-Master-using-BFMs/tree/master/HDL_sources/Testbench_Sources)