CSE467 Checkpoint 0: Course Logistics and Graphics Algorithm Software Model

Vincent Lee, Mark Wyse, Mark Oskin

Winter 2015
Checkpoint Due Tuesday, January 13th @ 11:59PM
Demonstration Due Thursday, January 15th in lab
Last Updated: January 10, 2015

1 Change Log

A previous version of this document in section 3.2 erroneously instructed to extend the vertex vector by zero instead of one to convert the 3D vertex vector into R^4 before applying the transformation matrix.

2 Project Overview and Logistics

Welcome to CSE467! In this course, you will be designing and prototyping a simplified graphics processing unit (GPU) over the entire quarter. Some of the components of the GPU will be implemented as hardware accelerator units and others as software graphics commands (more on this later). In any case, by the end of the quarter you will have solid understanding of how hardware and software systems interact, and go from CSE352 Verilog N00b to Verilog Master.

There is no single solution for the project this quarter and we expect everyone's solution will be slightly different. We will provide high level specifications and some structure for each of the project phases to help guide you through each stage of the project. Since this is a project based class, we expect you to exercise your engineering intuition to solve many of the underlying implementation details yourselves. This course will require a significant time commitment to successfully debug and complete your project. This is a right of passage for any engineer; you have been warned.

Finally, this is a no brainer but people always forget...ALWAYS READ AND UNDERSTAND THE ENTIRE PROJECT SPECIFICATION DOCUMENT BEFORE WRITING ANY CODE.

2.1 Grading

There are no exams or homeworks for this course. As a result (fortunate or unfortunate), your entire grade will be derived from your work on the project. The majority of your grade will be whether or not you successfully complete the project according to the provided specifications. During the course of the project, you will also be required to demonstrate progress by completing a few checkpoints (explained below).

2.2 Teams

You are allowed to work in teams of two or individually to complete the course project. In fact, we encourage you to work with a partner. If you decide not to work with a partner, that is fine, but your project will be evaluated and graded the same as those working in teams.

2.3 Checkpoints and Falling Behind

Throughout the project, you will be required to complete checkpoints. These checkpoints will help you stay on track in the project and help us evaluate your work throughout the quarter. Each checkpoint will have a specified due date and most will also require a demonstration of work completed. Checkpoint demonstrations will usually be due about 48 hours after the checkpoint due date; specific dates will be provided for each required checkpoint. You may use late days, according to the Late Policy described below, on checkpoint deadlines, but not on demstration deadlines.

Each phase of the project builds on the previous, and we will not be releasing solutions for any part of the project, as we expect each team to have slightly different designs. As a result, we understand the project structure can be unforgiving if you fall behind. If you feel overwhelmed by the pace of the project at any point in the quarter, "please" alert one of the TAs so we can allocate more of our attention to you and help you succeed!

2.4 Late Policy

Each checkpoint will have a prescribed deadline that we expect work to completed by. Many checkpoints will also require a demonstration, with a separate deadline that will usually be 48 hours after the checkpoint deadline.

Each checkpoint may be turned in late at a penalty of 10% per 24 hours until the demonstration deadline is reached. After the demonstration deadline, checkpoints will incur a 50% score penalty. We reserve the right to possibly add additional tax free late days depending on the FML index.

To humor the TAs, for each checkpoint deadline you may submit one original xkcd-like comic to offset 10% worth of late penalties (one comic per team). For example, if you submit your checkpoint work 46 hours after the checkpoint deadline, you would only incur a 10% late penalty as opposed to 20%. The comic must be submitted to the TAs via email by the ORIGINAL CHECKPOINT DEADLINE. You may not submit more than one comic per deadline and comics must be rated G, PG, or PG-13. Please do not include any personally identifying information since we will release these to the rest of the class to read while they wait for their designs to compile.

2.5 Collaboration

You are encouraged to collaborate with each other in the lab. Talking through your ideas and convincing someone else your idea works is probably one of the best ways to tackle a problem. Since many of you will probably be spending long nights together in the lab, get to know each other; you never know who you'll be collaborating with in the future. If everyone does well on the project, everyone gets a 4.0.

That being said, there are certain things you may not do. "NEVER" send someone else your code or copy theirs. Cheating is a serious offense which gets you fired in industry. Don't bother looking on the Internet for solutions, they don't exist; we created this project specifically for this class. If you are using a version control systems, your repository must be private. Justice will be dispensed to individuals or teams that are caught in violation of academic dishonesty.

3 Project Introduction

3.1 A 5000 ft. View

This quarter we are going to build programmable GPUs. The high level architecture of the GPU we are building is shown in Figure 1.

This is the basic architecture and if you complete the checkpoints and make this work you will receive a 3.8 in the class. To receive a 4.0 you must parallelize the GPU processor by instantiating two copies of it and executing a WARP on similar data types. Note that this class has no other homework and no exams.

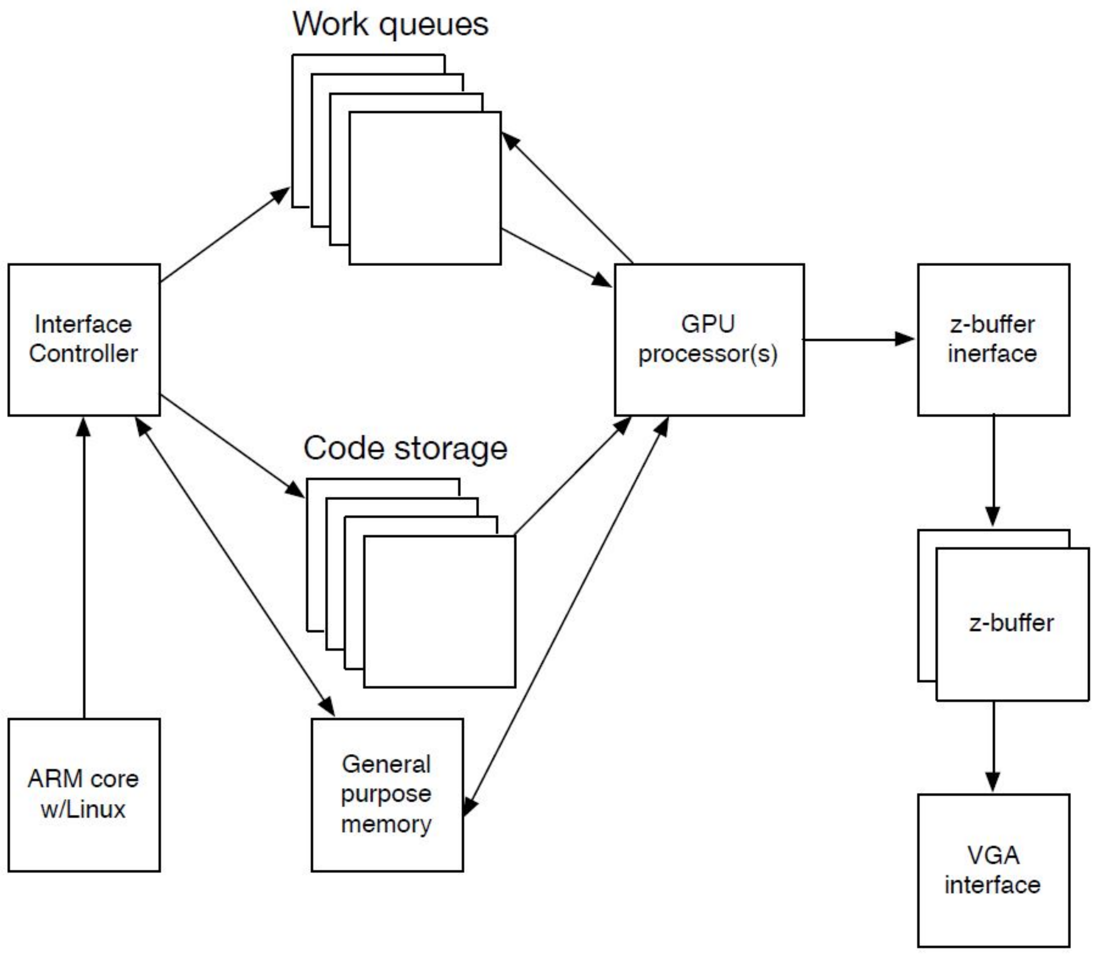

Figure 1: Rough System Diagram of GPU

This document is going to describe the function of GPUs in general, followed by a description of the architecture above, the instruction set, and finally the progression of work you should follow for this class. Note that this project is not precisely defined and there is a lot of room for you to make your own decisions about where the design should go. I hope throughout this project you embrace this flexibility and do cool things!

3.2 An Overview of Old School GPUs - 1992 to late 90s

In this class we are going to build more modern GPUs. But it is important to understand where GPUs come from to in order to understand where they are now. In this section we describe how older, fixed-function GPUs used to work. Graphics cards render a sequence of objects. These objects are passed to the graphics card over a FIFO link, usually in a DMA like fashion. They are stored to an object memory, essentially a polygon list. The CPU also controls the GPU and frame buffer, but this control is limited (reading and writing the frame buffer, and controlling the GPU).

The polygons are usually defined in three-dimensional space, in user coordinates. Usually user coordinates are defined as the volume of space spanned by (-1,-1,-1) to (1,1,1), centered at (0,0,0). In this model a polygon is simplified to be a triangle and is defined by:

(x1, y1, z1), (x2, y2, z2), (x3, y3, z3) and (nx,ny, nz) and (r,g,b)

In addition, under different lighting models either a triangle or each vertex of that triangle is given a color. For our class we will use the simpler shading model where each triangle has a color. The component of the triangle (nx,ny, nz) is the normal vector of the triangle. The normal vector is a vector that is perpendicular to the plane of the polygon and normalized to a magnitude of 1. The graphics card is going to use this to determine the color of the polygon. In this class you do not have to use textures (but you can!). Simple shades of gray are sufficient. By the end of the quarter, we will have implemented each of these stages of the pipeline either in dedicated hardware units or on the GPU cores.

In order to understand the underlying graphics algorithms you will be implementing for your GPU, we will first require you to implement a software model of the graphics pipeline. The model you build will plug into a simulation/debugger infrastructure which we will provide for you. Keep in mind that this model will eventually serve as a reference for you when designing the hardware and GPU core. You can use floating point values but avoid using division operators as dividers are costly to implement in hardware.

The GPU pipeline roughly consists of the algorithmic pipeline shown in Figure 2. You are not expected to have any background in graphics to complete the project. We will provide all necessary algorithmic background for you to complete the project.

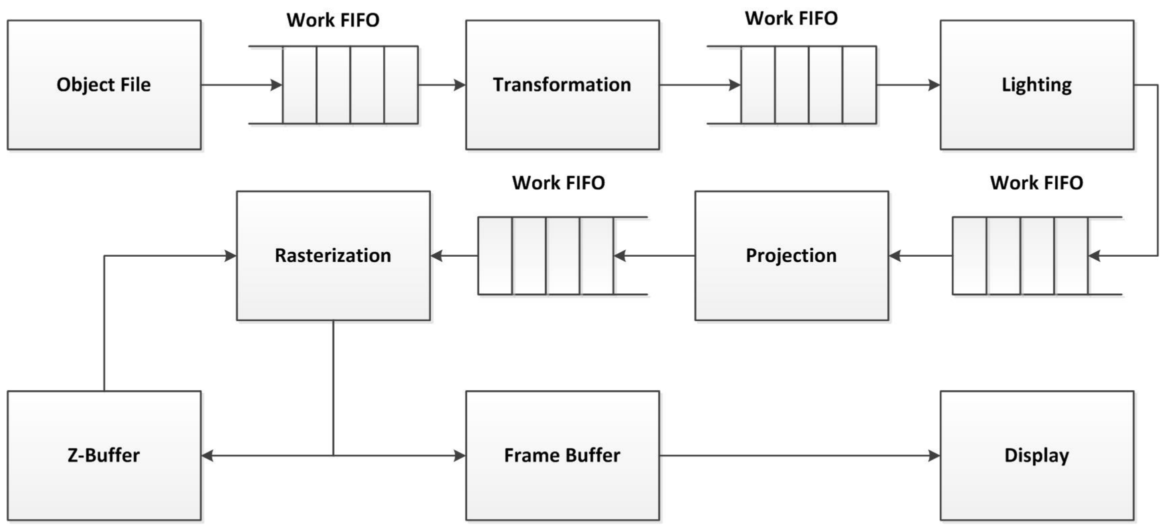

Figure 2: High level graphics pipeline

Throughout the course, we will incrementally build each of these components into a fully fledged modern GPU which will be presented in future phases of the project.

4 Checkpoint 0: Simulation Specification

In this first phase of the project, you are required to build a simulation model for the GPU. In other words, we are going to implement the above algorithms and pipeline in software. The goal of this first phase will be to render a teapot provided in the teapot.off file.

4.1 Reading the Object File Format

The teapot.off file contains a description of a teapot in 3 dimensional space. In order to render the teapot on a screen, it will have to undergo the appropriate transformations which we will handle later. Before we do that, we need to figure out how to read the file.

The OFF file format stands for Object File Format - not terribly interesting. The file is organized into three section: header, vertices, and faces. The header section contains information about how many vertices, faces, and edges there are in the file; essentially it tells us how large the file is. The vertices section contains the x, y, and z coordinates of each vertex in the model. Finally, the faces section contains the number of vertices in the face, and the indices of the verices composing the face. In the OFF files we provide, we will only be rendering triangles so the length of each entry in the faces section should be 4. Note that the indices that we refer to in the faces section are zero indexed.
For instance, if the OFF file looked like the following:

```
OFF
4 2 0
-0.5 -0.5 0.5
0.5 -0.5 0.5
-0.5 0.5 0.5
0 0 0
3 0 1 2
3 1 2 3
```

The first line is always "OFF" to indicate the file is an Object File Format. The second line says there are 4 vertices, 2 face, and 0 edges in the file. Lines 3 through 6 are the x, y, z coordinates for the 8 vertices specified in the header. For instance, in this example, the first vertex has coordinates x = -0.5, y = -0.5, and z = 0.5. Lines 7 and 8 are the face entries with the vertex indices composing the polygon. In this case, the first face entry is "3 0 1 2". This means there are 3 vertices composing the polygon associated with this face, and the vertices that make up the polygon are the 0th, 1st, and 2nd vertices in the vertex list. In other words, the entry describes a triangle with vertices (-0.5, -0.5, 0.5), (0.5, -0.5, 0.5), and (-0.5, 0.5, 0.5).

4.2 Perspective Transformation

The first task is to apply a perspective transformation matrix to the original cartesian coordinates of each vertex. This is to apply any scaling, rotation, or translations before we render the image. These transformation matrices can be arbitrarily complex so as a starting point, we will use the identity matrix:

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

To apply the transformation we extend the vertex vector with an additional one to make it live $\in R^4$. This is because the transformation matrix will be a 4 x 4 matrix which we simply left multiply the vertex vector by. Again note that this matrix is parametrizable to accomodate arbitrary rotation, scaling, and translation matrix operations. After applying the transformation, we simply discard the fourth dimension to convert the resulting vector back into R^3. After you have completed the implementation, you may go back and change this transformation matrix to see the effects of different transformations.

4.3 Lighting

After the transformation phase, we apply a very simple lighting technique. The intensity of each pixel in the image will be computed relative to a lighting vector. For your model, this lighting vector should be a parameterizable normal vector that indicates the direction of the lightning.

To determine the color that should be used for each polygon rendered in the image, a vector normal $\overleftarrow{N}$ to the plane of the surface will be projected onto this lightning vector $\overleftarrow{L}$. This amounts to a simple dot product between $\overleftarrow{N}$ and $\overleftarrow{L}$, the result of which we will define as $\overleftarrow{M}$. Given the intended color value of the polygon we are processing C, we compute the final color of the polygon C' as $C' = C * |\overleftharpoonup{M}|$ where the $|\overleftharpoonup{x}|$ operator denotes the magnitude of vector $\overleftharpoonup{x}$.

Therefore, in summary, the lighting intensity calculation can be described by:

$$C' = C * |\overleftharpoonup{N} \bullet \overleftharpoonup{L}|$$

Note that the vectors $\overleftharpoonup{N}$ and $\overleftharpoonup{L}$ are both of unit magnitude and thus the dot product should yield a value between 0 and 1. Also note that for now we will be using greyscale so the color value will be a single 8 bit value.

4.4 Projection

Now that the lightning for each polygon has been computed, we just have to project the 3D space onto the screen space and rasterize the polygons. To perform this projection, we will use the following transformations on each vertex in the object:

$$x' = (((x/4) + 1)/2) * screen\_width$$

$$y' = (((-y/4) + 1)/2) * screen\_height$$

$$z' = (((z/4) + 1)/2) * screen\_depth$$

The screen_width, screen_height, and screen_depth parameters will eventually be hard coded when we set out to design the GPU but for now you can assume each of these values are 8 bit or between 0 and 255. The divisions are used to shrink the polygons, while the addition of 1 is used to translate the space such that all vertex coordinates are positive. Notice that the transformation we provide above are using divisions by powers of two. This is because it simplifies the division by weakening it to a shift operation. As with the previous transformations, these values are not set in stone and you can tune them as you like.

4.5 Rasterization

WARNING: This section is the most involved step of the graphics pipeline. We suggest you go get a coffee, booster, or mana potion if you need one.

Now that the 3D object coordinates have been projected onto the screen space, we now just need to draw the pixels before we declare victory. Rasterization is the process of converting the vertices and polygons in the continuous coordinate space and discretizing them into the pixel space. The end result should be a screen buffer with pixel colors and z-buffer with pixel depths.

To do this, we need to discretize the lines between each set of vertices that form the triangles in our image since the pixel coordinates that compose each line cannot be fractional. Remember, we need to do this without arbitrary divisions because floating point division is costly to do in hardware. Furthermore, we need to be able to appropriately adjust for the depth at which each polygon lives. If multiple polygons end up coloring a given pixel coordinate, we should only keep the pixel value corresponding to the shallowest z depth and update the screen buffer with the new color only if the z depth is less than all prior depths. In essence, this suppresses pixels that belong to polygons that are obstructed by polygons that may be closer to the viewer.

So how does one manage to actually write the correct set of pixels to the screen buffer? One could imagine iterating through every polygon in the list for each pixel in the screen and determining which polygon occurs at the shallowest depth for that pixel coordinates. However, the runtime for this is $O(reallybig)$ so we're going to do something more efficient. Instead we're going to iterate over the polygons and draw only the pixels that correspond to each polygon. This should get us down to $O(notasbig)$.

As mentioned before, rasterizing a line in hardware without the division operation is not trivial. Luckily, someone smart already solved this problem for us... introducing the Bresenham line algorithm (google it). Using this algorithm, it becomes possible for us to compute the boundaries of polygon without using divisions. We recommend that you take a bit of time right now and convince yourself that this is true.

Using this line algorithm, we can now compute the depth of the pixels we need to render and the extent of their scanlines. To make this more efficient, we use scanlines to color all the pixels for a polygon that belong to the same y coordinate in the polygon (it can also be the x coordinate). Using the Bresenham line algorithm and the coordinates of each vertex in the polygon, we should be able to deduce the range of each scanline and depth of each pixel in the scanline (convince yourself this is true). We can then iterate through all the polygons in the list and draw all the pixels.

The general algorithm will therefore look like the following:

```
initialize z-buffer to 255 $\forall(x, y) \in R^2$
initialize color-buffer to 0 $\forall(x, y) \in R^2$
for all polygons P with color C do

  for all scanlines y in P do
    compute extents of scanline $(x_1, z_1)$ and $(x_2, z_2)$
    for all coordinate $(x, z)$ between $(x_1, z_1)$ and $(x_2, z_2)$ do

      if if $z <=$ z-buffer(x, y) then
        z-buffer(x, y) = z
        color-buffer(x, y) = C
      end if
    end for
  end for
end for
```

Figure 3: A high level rasterization algorithm

It is your job to figure out how to implement the scanline extent calculation. Since the scanlines are bound by their y-coordinates, you may want to break the triangle drawing into two steps. Think through the calculations involved in these steps carefully, getting the implementation is deceptively more involved than it looks.

4.6 Writing the Output File

For the simulation phase of the project, we expect the final result of your rendering to be written to a PGM (portable greyscale map) format. We have provided sample code for reading and writing PGM files that you are free to use and/or modify as needed. It “should” work as expected, however we provide no guarantees it is perfect!

As mentioned, we will be using the PGM file format (note: the sample code uses a strict interpretation of the standard, as described here). The first line of the file is a magic number, which for us is the literal “P2”. An optional comment line starting with “#” may follow the magic number. The third line is the width and height, separated by a space, e.g. “5 6”. The fourth line is the maximum grey value, or maximum pixel value, such as “255” for 8-bit grey pixels. Following these header lines are the data. Each data line is a space separated listing of pixel values. There must be exactly width*height number of pixel values, but rows of data are split across lines in the file. The pixel data in the file corresponds exactly as you would expect into the image, that is left to right and top to bottom.

A sample PGM file is shown in Figure 4. In the example, the PGM file has a width of 5 and height of 6 pixels with a maximum pixel magnitude of 255.

We encourage you to consult the Internet if these format specifications don’t make sense.

```
P2
# optional comment line
5 6
255
0 56 125 255 125
0 56 125 255 125
56 56 125 255 125
255 125 125 255 125
125 255 255 125 125
56 100 100 100 25
```

Figure 4: A sample PGM file

4.7 Hints and Gotcha's

While you build your model, keep in mind the following considerations as they can potentially be a source of frustratingly difficult errors:

- Normalize all appropriate vectors. The dot product of two normal vectors should always be less than or equal to one.
- The cross product between two vectors produces a vector normal to the plane formed by the two vectors.
- Make sure the value of all scalars are appropriately positive or negative. Remember, the lighting value is expected to be positive or zero.
- Check the dimensionality of matrices and vectors as they pass through subroutines in your model. Simple sanity checks can ensure that rogue dimensions aren't causing problems.
- Implementing and extending the Bressenham line algorithm to 3D can be tricky. Convince yourself the 2D algorithm works before you extend it to 3D.
- Be careful when generalizing the Bressenham line algorithm to arbitrary pairs of points as it can be tricky. Many of the implementations found online require pre-processing and post-processing before and after execution of the line algorithm.

If there is a "gotcha" that you end up running into, feel free to post on the class forum for help or ask the TAs.

5 Specifications and Demonstration

Your assignment for this checkpoint is to write a simulator and implement a model for the graphics pipeline outlined in previous sections. You will also be required to demo your simulator to the TAs. In the event your demonstration does not work correctly, we will require you to prove to us which components of the system work and explain why the rest is broken.

Your software model of the GPU pipeline must adhere to the specifications provided in order to receive full credit for this checkpoint:

1. Your implementation must render the teapot provided in teapot.off correctly and in a reasonable amount of time. Having some artifacts in the end result is okay.
2. You must be able to show different illuminations of the teapot by changing the lighting vector.

3. You must be able to show different scales, translations, and rotations of the original image by altering the transformation matrix.
4. You must not use graphics libraries such as OpenGL headers.
5. You are limited to only dividing by powers of 2.
6. Your model should output a PGM file with the resulting image that gets rendered.

You must demonstrate your working code by the end of lab (**5:30PM on Thursday, January 15th**). All demonstrations performed after that time will be subject to late penalty.

5.1 Submission Artifacts

Prior to the checkpoint due date, you are required to submit the following items to the course dropbox in the form of a tarball or zip file.

1. All source code required to make your design run. If you are using library files that are not part of the standard library, you must also include them. Your submission should be able to compile and stand on its own.
2. At the top level of your submission folder, you must include a README file which contains the first and last name, SID, and email of each member on the team. The README must also provide instructions on how to run and compile your code.

These artifacts are due **Tuesday, January 13th at 11:59PM**.