Assignment 1 - Software Rasterizer

You must work individually.

Goal

Create a program that reads in and renders an indexed face set meshes (of type .obj) to an image via software rasterization. You may use existing resources to load in the mesh and to write out an image. You must write your own rasterizer. In general the required steps for the program are:

- Read in triangles.
- Compute colors per vertex.
- Convert triangles to window coordinates.
- Rasterize each triangle using barycentric coordinates for linear interpolations and in-triangle test. Write interpolated color values per pixel using a z-buffer test to resolve depth.

Task 1

Download the base code, which has a mesh loader and an image writer from PolyLearn.

The mesh loader is an obj loader from http://github.com/syoyo/tinyobjloader, and the image writer is the same as the one used in prior labs.

Example mesh files are included in the base code, and you can create your own to represent a single triangle, etc. In addition, there are numerous OBJ meshes on the web. For grading purposes, your program will be run using the provided Stanford Bunny and Utah Teapot.

Ultimately you will want each triangle to be represented in a C/C++ structure/class, with 3 vertices and a color per vertex. In addition, your triangle data should include a 2D bounding box, which will represent the triangle's extents in window coordinates.

Task 2

Add a command line argument to accept the following command line arguments.

- Input filename of the .obj file to rasterize
- Output image filename
- Image width
- Image height
- Coloring mode (see Task 5)

For example, your program should be able to be run as follows (Linux/Windows):

```
>  ./Assignment1 bunny.obj output.png 512 512  1
> Assignment1 bunny.obj output.png 512 512  1
```

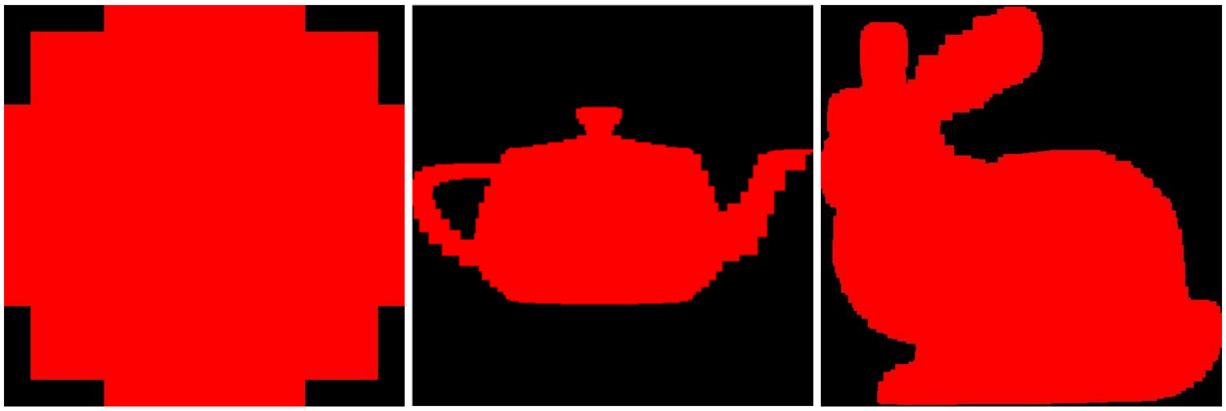
Add error checking to specify the required command line arguments if an incorrect number are given – and include an example correct command line run. Your program should not dump core if no input file is specified, nor fail without an error message! Follow the golden rule, treat your user/grader/instructor the way you'd like to be treated as a user/grader/instructor.

Task 3

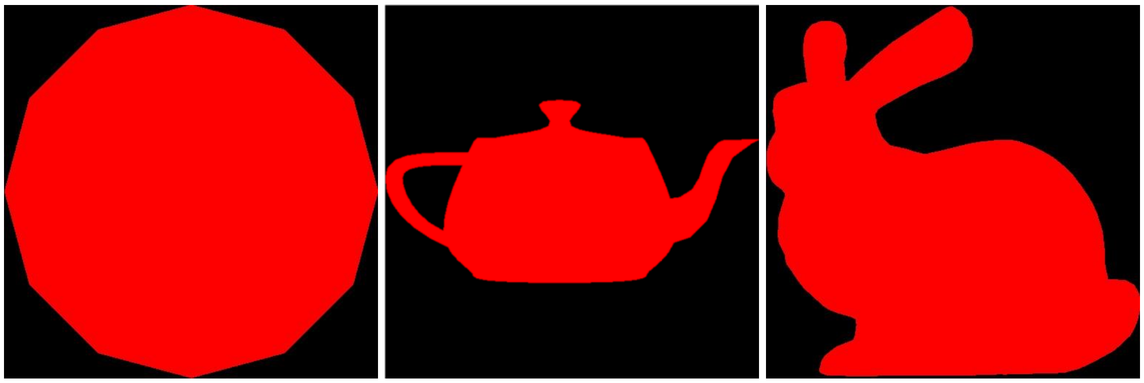
Write code to convert each 3D coordinate into 2D window coordinates. Assume the camera is at the origin looking down negative z. Make sure the object *completely* fills the image. Some tips for starting out:

- Color everything with a single color (e.g., red, yellow, etc.).
- Create a small OBJ file representing a single triangle. Using this small test case, make sure the numbers from the conversion are reasonable.

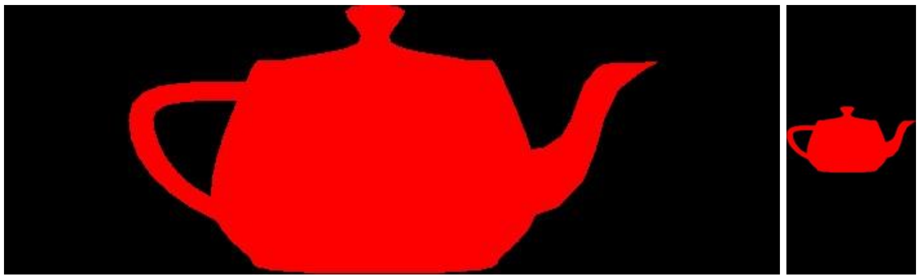
Write out the bounding box, rather than the triangles, to the image. If you do this with the provided sphere, teapot, and bunny, you should see blocky images like below. Make sure the object takes up the whole image and is centered.



Once the bounding boxes are being displayed correctly, you can add the barycentric test to write out the triangles as in Lab 2. You should not see any gap between the triangles.



Make sure you test nonuniform window sizes. As show below, the aspect ratio of the object should be preserved no matter what the image size is, and the object should fill out the image.



Task 4

Create a data structure to support z-buffer tests. Your z-buffer should be a *separate* buffer from your image pixel buffer, and it should be the same size as your pixel buffer. Its fine to just do another “image” object with the same resolution.

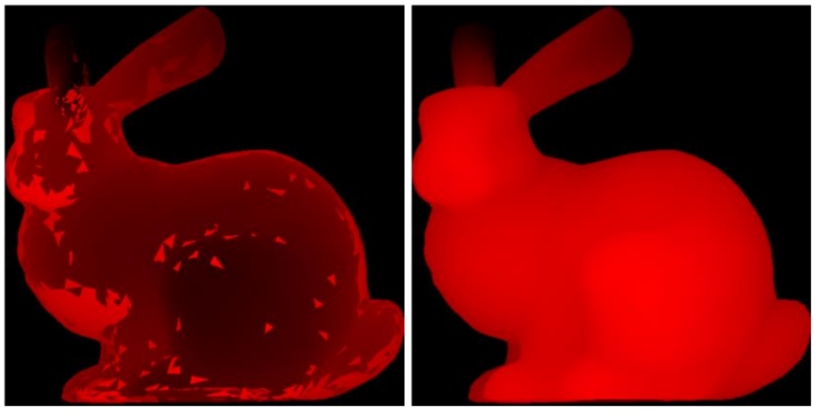
Incorporate your lab code to rasterize a triangle with a color defined per vertex. Any point within the triangle should be drawn with colors interpolated via the barycentric coordinates. Likewise, depth should also be interpolated using the barycentric coordinates and written to the z-buffer.

Use the z value of the vertices as the color. (You can choose any color, not just red.) To do this, you have to map the z-value to the range 0 to 255. If your z-buffer test is not working, you'll see some strange results, since some pixels that are farther from the camera may be drawn on top of closer pixels.

For testing if a pixel is in the foreground, you need to compare the pixelcolor on the position of the new pixel:

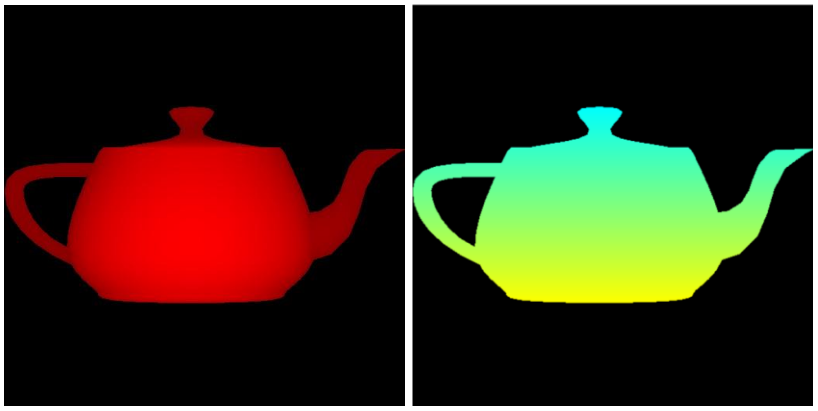
- If there is none, write the new pixel color.
- If its less colored (farther away) write the new pixel color.
- If its more colored (closer), do not overwrite the pixel.

For that, you need a possibility to GET the pixel color on a certain x/y position in the image. On the google doc workspace, there is a ready written method getPixel(…) which you can implement.


Picture shows the bunny w/o and with z-testing.

Task 5

Use the "color mode" command line argument to switch between two modes. If mode is 1, then use the z-value as in the previous task. If mode is 2, then use the y-value to linearly interpolate two colors of your choice. For example, in the right figure below, I am interpolating between yellow and cyan. Make sure to *specify these two colors in your README*. The color should vary smoothly from top to bottom.



Important Note

Make sure to pass your std::vector by reference rather than by value. (E.g., void foo(std::vector<float> &bar)) Otherwise, your program may become too slow.

Point breakdown

- 20 points for window coordinate transforms for square images.
- 10 points for window coordinate transforms for non-square images.
- 20 points for correct rasterization of triangles.
- 15 points for correct z-buffer implementation.
- 10 points for mode 1 color interpolation.
- 10 points for mode 2 color interpolation.
- 15 points for coding style and general execution. For example, do not put everything in main().

Total: 100 points

What to hand in

Failing to follow these points may decrease your "general execution" score. Make sure that your code compiles and runs by typing.

Unix/Linux:

```
> mkdir build
>  cd build >  cmake ..
> make
>  ./Assignment1 <ARGUMENTS >
```

Windows:

```
> Assignment1 arg1 arg2 arg3 …
```

- Make sure the arguments are exactly as specified.
- Include a README file that includes:
    - Your name
    - The two colors for the second coloring mode
    - Citations for any downloaded code (e.g., barycentric)
      Plus anything else of note
- Make sure you don't get any compiler warnings.
- Remove unnecessary debug printouts.
- Remove unnecessary debug code that has been commented out.
- Create a single zip file of all the required files. The filename of this zip file should be
- CALPOLY_USERNAME.zip (e.g., ceckhard.zip) . The zip file should extract everything into a folder named CALPOLY_USERNAME (e.g. ceckhard).
- Upload it to the submitting portal on PolyLearn.
- Specific for CMake/Vusial Studio see next page.

CMake

Please submit the project folder including the required source files *plus a makefile* and a README.txt and the .OBJ files to the submitting portal on PolyLearn.

All these files should be compressed into a single archive file (zip or tar only).
If it's too big, upload it to your one-drive or google-drive, share it with everyone, write the link into a text file and upload it.

The code should compile by typing make at the command prompt on a lab Linux machine. *Submit a README with your program which describes which features of the program work or do not work.*

A completely functional program is worth 100 points total. Points are awarded for a combination of functioning code and decent code design – i.e. I will look at your code design for this application.

Visual Studio

Please zip your project (the whole folder containing the Visual Studio project) including the required source files *plus a makefile* and a README.txt and the .OBJ files and upload it to the submitting portal on PolyLearn. *Submit a README with your program which describes which features of the program work or do not work.*

All these files should be compressed into a single archive file (zip or tar only).
If it's too big, upload it to your one-drive or google-drive, share it with everyone, write the link into a text file and upload it.

A completely functional program is worth 100 points total. Points are awarded for a combination of functioning code and decent code design – i.e. I will look at your code design for this application.