CSCE 441 - Computer Graphics

Programming Assignment 3 - Part 1

Deadline: Oct. 12th (11:59 pm)

1 Goal

The goal of this assignment (part 1) is to become familiar with rasterization process.

2 Starter Code

The starter code can be downloaded from here.

3 Task 1

Download the code and run it. You should be able to see a white bunny as shown in Fig. 1. Make sure you write your name in the appropriate place in the code, so it shows up at the top of the window. Here is a brief explanation of the starter code:

- There are two folders in the package. "resources" contains the a few obj files that contain the geometry information for different objects. It also contains an image which is related to the second part of this project. The other folder "src" contains the source files. For this assigment, you'll be mainly modifying the "main.cpp" and the triangle class. "tiny_obj_loader.h" is a simple header file for loading the obj files and you will use it as is. Moreover, "stb_image.h" and "stb_image_resize.h" are header files for reading and resizing images and are related to the second part of the project.
- The `main` function in "main.cpp" is similar to the one in all the previous assignments. The `Init` function, in this case, loads the model (as well as the texture; for the second part) in addition to initializing the window and events. The `LoadModel` function, reads the vertices of triangles from the desired obj file and writes them into the `vertices` vector. The `CreateTriangleVector` function then creates an instance of the triangle class for each three vertices in the `vertices` vector and pushes them into the `triangleVector` vector.
- The display code then constructs the modelview and projection matrices and draws the triangles one by one by calling the appropriate drawing function in the triangle class. There are two modes; one drawing using OpenGL (`RenderOpenGL`) and the other is drawing with CPU (`RenderCPU`). The function for drawing using OpenGL is already provided, but you have to implement the rasterization process in `RenderCPU` function. You can toggle between rendering using OpenGL and CPU using the space key. In the skeleton code, OpenGL draws a white bunny, since the color of all the vertices are set to white in the triangle class constructor. The CPU mode, however, does not draw anything since the `RenderCPU` function is empty.
- You can move closer and further away from the object using 'w' and 's', respectively. The code simply adjusts the distance of the camera to the coordinate center where the object is located at. You can also rotate the object to be able to see it from different angles by pressing "a" and "d".

Currently, the path to the obj file is hardcoded. You should set up your code so it takes the path to the obj file as the input argument. This way you can test your system on different models without changing the source code.

4 Task 2

In this part, you will be implementing different coloring modes. You should be able to switch between different modes by pressing '0', '1', and '2'. Note that, you should assign the colors to the vertices before rendering, so the color of vertices doesn't change each time the bunny is rendered on the screen.

Figure 1: Running the skeleton code should produce a white bunny as shown here.

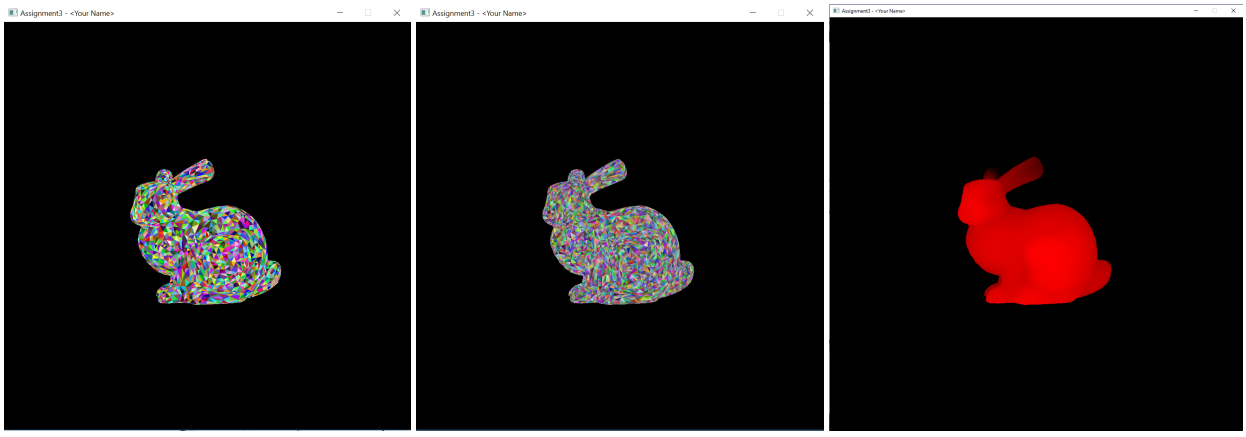

Figure 2: The coloring modes 0, 1, and 2 are shown on the left, middle, and right, respectively.

- **Mode 0:** Assign a random color to each triangle, as shown in Fig. 2 (left). You can use the "rand()" function which generates a random integer between 0 and RAND_MAX. Dividing "rand()" by RAND_MAX will give you a random number between 0 and 1. Note that your version is not going to be exactly like the one shown in this figure as you'll be assigning the colors randomly.
- **Mode 1:** Assign a random color to each vertex, as shown in Fig. 2 (middle). Again your version is going to be different from the one shown in this figure.
- **Mode 2:** Use the z value of each vertex to generate its color, as shown in Fig. 2 (right). You can do this by setting, for example, the z value to the r channel and set the g and b channels to zero. You can choose any color you want for this. Note that, you have to map the z values to range 0 and 1 (min-z mapped to 0 and max-z to 1) before using it as the color.

5 Task 3

Here you implement the graphics pipeline on CPU. A call to `RenderCPU` should draw the current triangle on to the `color` array (defined on top of the starter code). You should use `depth` array to implement the

z-buffer algorithm. Note that, you have to pass appropriate variable to the `RenderCPU` function since the triangle class does not have access to all the necessary variables. You have to implement the followings:

- **Transform the triangle to the screen –** The vertices of the triangle are in object coordinate. To project them onto the screen you first need to apply model view projection transformation to bring them to normalize device coordinate (NDC). Finally you apply viewport transformation to go from NDC to screen space. Here, we don't have a model transformation (it is basically identity) and view and projection matrices are provided using `glm::lookAt` and `glm::perspective`, but you have to create the viewport matrix based on the screen resolution. Note that, you have to perform perspective division (division by the w coordinate) to get the final transformed coordinates.

- **Rasterize the triangle –** The previous step places the triangle on screen. Now, you have to loop over the pixels on the screen and determine if a pixel is inside or outside the triangle. To make sure the code runs at a decent speed, you have to implement bounding rasterizer. This means you need to first compute the min and max of the x and y coordinates of the triangle vertices to find the box that contains the triangle. Then you only look over the pixels in the box and perform the inside test.

  Note that, you do not need to implement the edge cases that we discussed in the class. Since we do not have any transparent objects, you can double rasterize the pixels shared with two triangles (Incorrect solution #1 in the slides). Make sure there are not any gaps in between your triangles.

- **Interpolate the color of the pixel using Barycentric coordinates –** If a pixel is inside the triangle, you need to compute its color by interpolating the color of the three vertices. For this, you need to implement Barycentric coordinates (α, β, γ from the slides). Once you obtain these, you can compute the color as the weighted sum of the color of vertices.

- **Implement the z-buffer algorithm –** In this stage, you make sure that only the triangles that are closest to the camera are drawn. This can be done using the z-buffer method as discussed in the class. The basic idea is to use a depth buffer (you can use `depth`) to keep track of the closest depth drawn so far. You basically initialize this buffer with infinity. Then before drawing each pixel onto the color buffer, you first check if its depth is less than the depth in the depth buffer. If it is, then you color the pixel in the color buffer and update the depth buffer. If it is not, then you do nothing. Note that, to obtain the depth at every pixel, you have to interpolate it from the depth of the three triangle vertices using Barycentric coordinate.

- **[Extra]** Implement the clipping algorithm.

You need to test your implementation by comparing against GPU rendering (by pressing space) on different objects and various conditions. Make sure you move the camera closer to the object and further away to test the accuracy of your code. If you implement the clipping, test your algorithm by getting really close to the object. Your rendering with clipped triangles should match that of OpenGL.

6 Deliverables

Please follow the instruction below or you may lose some points:

- You should include a README.txt file that includes the parts that you were not able to implement, any extra part that you have implemented, or anything else that is notable. Note that the README file should be in "txt" file format and you should place it in the root folder next to the "src" folder and "CMakeLists.txt" file.

- Your submission should contain the "src" folder. DO NOT include the "build" and "resources" folders as well as the CMakeLists.txt.
- Zip up the whole package and call it "Firstname_Lastname.zip". Note that the zip file should extract into a folder named "Firstname_Lastname". **So you should first put your package in a folder called "Firstname_Lastname" and then zip it up.** The zip file structure should be exactly as follows:

  "`firstname_lastname.zip`"
  - "`firstname_lastname`"
    - "src"
    - README.txt
- Submit the package through Canvas.

Note: Once you upload your zip file, Canvas might adds extra information to the name of your zip file, which is fine. The most important thing is to put everything in a folder "Firstname_Lastname" and then zip this folder up. Canvas will not change this folder name.

7 Ruberic

Total credit: [100 points]

[05 points] - Taking the obj file as an argument
[20 points] - Implementing color modes
    [05 points] - Mode 0
    [05 points] - Mode 1
    [10 points] - Mode 2
[75 points] - Implementing the full rasterization pipeline on CPU
    [10 points] - Transforming the triangles
    [25 points] - Implement bounding rasterizer with no gaps between the triangles. To get the full point, your code should not take seconds (more than 3 seconds) to render a frame. The speed requirement is at the original position of the bunny. Your code gets slower as you zoom into the scene.
    [20 points] - Barycentric interpolation
    [20 points] - Implement Z-buffer

Extra credit: [20 points]

[20 points] - Implement clipping

8 Acknowlegement

The color modes is based on Shinjiro Sueda's assignment.