Assignment 4: Pipeline and Shading
CS180 Fall 2021

Professor: Lingqi Yan

University of California, Santa Barbara

Assigned on Oct 22nd, 2021 (Friday)

Due at 23:59 on Nov 1st, 2021 (Monday)

Notes:

- Be sure to read the Programming and Collaboration Policy on course website.
- Any updates or correction will be posted on Piazza, so check there occasionally.
- You must do your own work independently.
- Read through this document carefully before posting questions on Piazza.
- Submit your assignment on GauchoSpace before the due date.

---

1 Overview

In this assignment, we will move one step forward to simulate the modern graphics pipeline. We have updated our skeleton code by adding the object loader, vertex shader and fragment shader stage, and also support texture mapping. The task for you in this assignment is to interpolate the attributes of triangle (normals, colors, texture coordinates, etc.), and use them to implement the Blinn-Phong shading model in the `phong_fragment_shader()` function for only the shading part. When you get Blinn-Phong reflectance model implemented, you can move on to the textured shading fragment shader. When you are working on the textured shading, write your code into the `texture_fragment_shader()` function in main.cpp file. Then, you actually need to change the active shader of the rasterizer. You should do it this way: in the main.cpp file, there is a variable called `active_shader()`. You can take a look at how we set it to different shaders in the code, and update it to your shader. Also, if you are using the shaders we have provided for you, you can use the command line interface to choose the shader that you want to use. The details of the command line interface is explained below in more detail.

Functions that you need to modify:

- `get_projection_matrix()` in `main.cpp`: Use your implementation for the projection matrix from the previous assignments.
- `phong_fragment_shader()` in `main.cpp`: Compute the fragment color according to blinn-phong reflection model.
- `texture_fragment_shader()` in `main.cpp`: Compute the fragment color according to blinn-phong reflection model. Use the texture color as the kd coefficient in the formula.
- `rasterize_triangle(const Triangle& t)` in `rasterizer.cpp`: Interpolate the needed attributes, and pass them to the fragment shader payload.

Initially, the code is compilable, and you can run the program directly by saying `./Rasterizer`, and it will initiate with an already implemented shader that visualizes the normals of the mesh, although since you will not have implemented the `rasterize_triangle()` function, you will not see anything. When you implement the function, you should be able to see the normals of the shape. After you implement your own shaders `phong_fragment_shader()` and `texture_fragment_shader()`, you should look for a variable called `active_shader` inside the main function. That

variable sets the shader to be passed to the rasterizer. Go ahead and update this variable to your shader (or, you can use the command line tool to choose among already implemented shaders), and voila! You're ready to run your cool shader.

2 Getting started

Same as the previous assignments, you can either choose to work in your own system, or to use the virtual machine. Download the skeleton code package for assignment 4. When building the project from command line, this time we require you do the following, create a folder named build under SoftwareRasterizer directory:

```
$ mkdir build
$ cd ./build
$ cmake ..
$ make
```

This will generate the executable name `Rasterizer`. There is a slightly extended command line tool that you can use with this executable. As usual, the second argument you give to the executable will be the output image name. The third argument can be:

- texture: Activates the **texture** shader in your code.
  Example usage: `./Rasterizer output.png texture`
- normal: Activates the **normal** shader in your code.
  Example usage: `./Rasterizer output.png normal`
- phong: Activates the **blinn-phong** shader in your code.
  Example usage: `./Rasterizer output.png phong`

Whenever you make changes to the code and want to see the new result, you need to type make again.
We have made several changes to the framework:

1) We have included a third party .obj files loader library to load more complex model from files. This exist in the `OBJ_Loader.h file.` You don't have to understand how this loader works in detail. Just be aware of that it generates a vector of triangles for us, which we call `TriangleList` in the program. Each triangle will also be assigned per vertex normals and texture coordinates.

Meanwhile, texture-associated with this model will will also be loaded. **Note: If you want to load other objects, you have to change the path manually for now.**

2) We have introduced a new texture class to create textures from images. And provided the interface to perform the texture color lookup function: `Vector3f getColor(float u, float v)`.

3) We created the Shader.hpp header file and defined the `fragment_shader_payload`, which contains the attributes that are will use by your fragment shaders. Currently there are three fragment shaders in the `main.cpp`. The `fragment_shader` is the example shader that shade the fragment according to its normal vector. The other two are left for you to implement.

4) The main rendering pipeline starts in `rasterizer::draw(std::vector<Triangle> &TriangleList)`. We perform a series of transformations here. Usually it is the job for the vertex shader. Then we call the `rasterize_triangle` function.

5) The `rasterize_triangle` function is similar to what you did in assignment 3. Instead of assigning constant value, you need to do interpolation for normals, colors, texture colors and shading colors using Barycentric Coordinates. Recall `[alpha, beta, gamma]` we provided late time to calculate the interpolated z value, now it is time for you to apply them to more other properties. What you need to do is to calculate interpolated color, and write the color computed by fragment shader into the framebuffer. This requires you to set up the fragment shader payload first using the interpolated attributes and to call the fragment shader to get result.

After you copy-pasted your code from the last assignment and make revision based on the instruction above (**Please make sure that you read through what we have revised from last time**). You can run the program with the default normal fragment shader, you will see this:

The shading result after your implementation of the Blinn-Phong reflectance model will look like:

Result after your implementation of the textured shading will look like:

3 Grading and Submission

Grading:

1. (5 points) Submission is in the correct format, include all necessary files. Code can compile and run.

2. (10 points) Attributes interpolation:
   Given the attributes at each vertex, we need to interpolate them to get the attribute for the pixels inside. Interpolate color, normal, texture coordinates, view space position (shading position) correctly and pass them to the `fragment_shader_payload` structure.

3. (20 points) Blinn-phong reflection model:
   Implement the shading model correctly in the fragment shader, which is represented by the `phong_fragment_shader` function in `main.cpp`.

4. (5 points) Texture mapping:
   Copy the code from the `phong_fragment_shader` to `texture_fragment_shader`, replace the kd coefficient with the color from your texture mapping result. You only need to sample the nearest neighbour value for now.

5. (Bonus 3 points) Load more models: Find other usable .obj files for our framework. Submit the image you rendered and save the models in the `/models` folder. Please note that those obj files you use should contain vertex normal information as well. If the .obj file you downloaded does not contain vertex normal information, you can use additional softwares such as MeshLab to process the .obj file, calculate and store the vertex normal information. If you are interested, please learn how to do so on your own. Also, our skeleton code does not support quadrilateral faces. So please make sure not to read .obj files containing quadrilateral faces directly.

6. (Bonus 5 points) Bilinear texture interpolation: Use the bilinear interpolation to sample the texture value, implement a new function in the Texture class: `Vector3f getColorBilinear(float u, float v)`. Call it from your fragment shader. Note that in order for the bilinear interpolation to be more obvious, you may consider using a smaller texture. Submit two comparison rendered images.

Submission:
After you finish the assignment, clean your project, AND remember to include the CMakeLists.txt in your folder, whether you modified it or not. Add a **SHORT** README.md file in the directory, write down your full name and perm number inside it. Tell us whether you did the bonus part or not, and **briefly describe what you have imlemented in each function.** Apart from the code and report, we also require you to include the following output images:

1. Three rendered images of `spot_triangulated.obj`, using normal, phong and texture shaders.

2. (For Bonus 1) Three rendered images of another .obj file you found elsewhere, using normal, phong and texture shaders.

3. (For Bonus 2) Two renderd images of `spot_triangulated.obj` (texture shader) with a compressed texture image, one using nearest neighbour, the other using bilinear interpolation.

Then compress the entire folder and rename it with **YOUR NAME**, like "Goksu.zip". Submit the .zip file on GauchoSpace.