

UCSD CSE 167 Assignment 3: 3D OpenGL Rendering

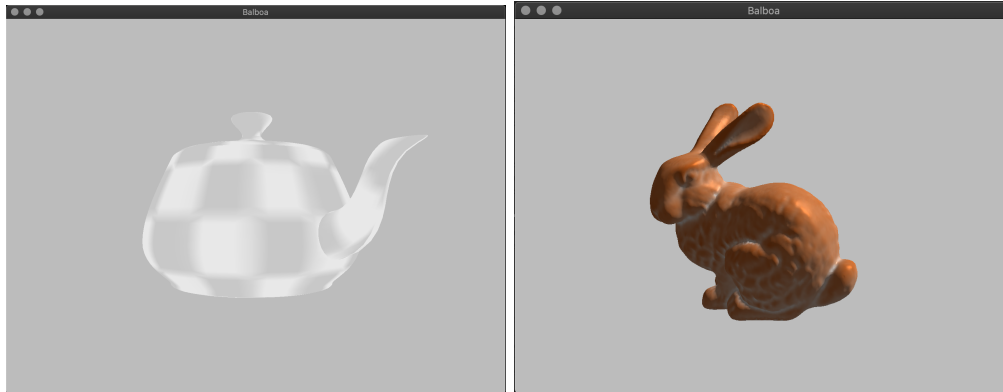


Figure 1: We will develop an interactive interface for inspecting 3D models in this homework.

As you can probably tell from the previous homeworks, rendering requires computing interactions between millions of pixels and billions of triangles. This leads to significant challenges in performance, especially when we want to interact with the content in real-time. To make things really fast, pioneers in computer graphics came up with the solution to use *domain-specific hardware* to speedup rendering. Instead of using a general purpose computer to compute everything, we build chips that specialize at rendering. These processors are called the Graphics Processing Units (GPUs). The idea of GPUs can be traced back to more than 40 years ago: The first GPU, **Geometry Engine** was developed by Jim Clark and Marc Hannah in 1981. Jim Clark formed the company Silicon Graphics Inc (SGI) in the same year and SGI was one of the most important computer graphics companies in the history. Nowadays, GPUs are found to be general enough to compute very wide-range of computation, including deep learning and many scientific computing tasks, and they are indispensable to the human society. GPU is one of the most successful examples of domain-specific hardware.

In this homework, we will write code to render things using GPUs on your computer. To command your GPUs, we need to send commands to it using some sort of “Application Programming Interface” (API). These interfaces are collectively decided by the GPU companies and some other organizations, and each hardware will come with some “drivers” that actually implement these interfaces using underlying hardware instructions. The most popular APIs are: **OpenGL**, **DirectX**, **Metal**, **Vulkan**, and **WebGPU**. Among these, DirectX is Windows only, Metal is MacOS only, WebGPU is only for browsers, and Vulkan is extremely low-level and very verbose for providing fine-grained control (it takes literally a thousand lines to render a single triangle in Vulkan). Therefore, we will use OpenGL in this homework: even though DirectX, Metal, and Vulkan are more update to date (the latest version of OpenGL is 6 years ago), OpenGL is still use in practice and supported by all major GPUs and OSes, and it is significantly easier to learn compared to other lower-level APIs. Just like programming languages, it’ll be a lot easier to learn other APIs once you’ve learned OpenGL.

In this homework, we will mostly follow an online tutorial: learnopengl.com, because they likely write significantly better tutorials than me. We will implement what we did in the previous homework in OpenGL and hopefully see significant speedup. We will also create a Graphics User Interface (GUI) and enable real-time interaction.

This homework is also more “open-ended” compared to the previous ones. We do not ask you to produce the exact same output as we do. At this point, you should be familiar with the theory of rasterization. We’re just wrangling with hardware interface, so allowing a bit of creativity seems reasonable.

1 Creating a window (10 pts)

Our first task, instead of rendering a single triangle, is to create a window! Read the chapters of [OpenGL](#), [Creating a window](#), and [Hello Window](#) in [learnopengl.com](#) to see how to create a window with OpenGL context using GLFW. Pick your favorite background color. We have included GLFW and glad in `balboa`, so you shouldn't have to download them. We're using OpenGL 3.3, but feel free to use the version you like.

Implement your code in `hw_3_1` in `hw3.cpp`. Test it using

```
./balboa -hw 3_1
```

Once you are done, take a screenshot of the window you created and save it as `outputs/hw_3_1.png`.

2 Rendering a single 2D triangle (20 pts)

Yeah, it's that time again! Read the [Hello Triangle](#) chapter and render a single triangle with constant color (pick one that you like the most). Make sure you've become familiar with the ideas of shaders, VAO, VBO, and EBO. Just to make things slightly different so that we are not just copy and pasting code, let the triangle rotate in the image plane over time (it can be clockwise or counterclockwise, your choice). For the rotation, you can do it whichever way you want, but I recommend you do it in the vertex shader. Read the [Shaders](#) chapter and understand how to pass in a `uniform` variable, then you can use the uniform variable as the rotation angle.

float vs. double By default, `balboa` uses double precision floats through the `Real` type. However, by default, GLSL uses single precision floats. Be careful of this discrepancy. You can use `Vector3f/Matrix3x3f` to switch to float in `balboa`. Also feel free to use the `glm` library which is used in the tutorial.

Implement your code in `hw_3_2` in `hw3.cpp`. Test it using

```
./balboa -hw 3_2
```

This time, do a screen recording of your rotating triangle and save it as `outputs/hw_3_2.mp4` (or whatever encoding you are using).

3 Rendering 3D triangle meshes with transformations (35 pts)

Next, we'll use OpenGL to render the type of scenes we handled in the previous homework. Read the chapters [Transformations](#), [Coordinate systems](#), and [cameras](#), and that should give you enough knowledge to render the JSON scenes like the ones in the previous homeworks.

This part is a big jump from the previous parts. I would recommend you to do things incrementally. E.g., handle two 2D triangles first, add projection matrix, add view matrix, add model matrix, handle multiple triangle meshes, and finally add camera interaction.

Below are some notes and tips:

Clip space. In Homework 2, our projection matrix convert from camera space directly to the screen space. In OpenGL, the hardware expects the projection to convert from camera space to the *clip space*, which by default ranges from -1 to 1 for x , y , and z axes. Everything outside of the clip space is clipped. Note that the clipping happens at the far side of z as well – we use the `z_far` parameter in the camera in our JSON scene to specify this. The difference in spaces means that we need to use a different projection matrix:

$$\begin{bmatrix} \frac{1}{as} & 0 & 0 & 0 \\ 0 & \frac{1}{s} & 0 & 0 \\ 0 & 0 & -\frac{z_{far}}{z_{far}-z_{near}} & -\frac{z_{far}z_{near}}{z_{far}-z_{near}} \\ 0 & 0 & -1 & 0 \end{bmatrix}, \quad (1)$$

where s is the scaling/film size parameter as before, and a is the aspect ratio. The first row and the second row scale the x and y clipping plane to $[-1, 1]$ respectively. The third row compresses z values from $-z_{\text{near}}$ to $-z_{\text{far}}$ to $[-1, 1]$. The fourth row is the perspective projection using homogeneous coordinates.

Depth test. By default, OpenGL does not reject triangles when they are occluded. Remember to turn on depth testing using `glEnable(GL_DEPTH_TEST)` and clear the Z buffer (e.g., `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)`).

Vertex colors. In contrast to the learnopengl tutorial, balboa stores the vertex color in a separate array. Therefore it's likely more convenient to create two VBOs:

```
unsigned int VBO_vertex;
glGenBuffers(1, &VBO_vertex);
glBindBuffer(GL_ARRAY_BUFFER, VBO_vertex);
glBufferData(GL_ARRAY_BUFFER, ...);
glVertexAttribPointer(0 /* layout index */,
                    3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
unsigned int VBO_color;
glGenBuffers(1, &VBO_color);
glBindBuffer(GL_ARRAY_BUFFER, VBO_color);
glBufferData(GL_ARRAY_BUFFER, ...);
glVertexAttribPointer(1 /* layout index */,
                    3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
```

You only need one VAO per mesh regardless.

Multiple meshes. To handle multiple meshes in a scene, create a VAO for each mesh.

Window resizing. We don't require you to handle window resizing in this homework. It's annoying because you'll need to regenerate the projection matrix every time the aspect ratio changes.

Gamma correction. When we save the image in balboa, we perform a gamma correction by taking a power of $\frac{1}{2.2}$. OpenGL does not by default do this. To enable gamma correction, use `glEnable(GL_FRAMEBUFFER_SRGB)`. Read the [gamma correction](#) chapter in learnopengl.com to learn more.

Camera interaction. Like the tutorial, you should also implement a simple camera interaction scheme, see the [Camera](#) chapter. A simple WSAD style translation suffices. To obtain the camera direction and right vector, you can look at the columns of the `cam_to_world` matrix.

As a **bonus** (15 pts), add camera rotation based on mouse input like the tutorial. Note that the rotation in the tutorial assumes a particular camera frame and would not work for our case. I recommend doing the following: 1) store `yaw` and `pitch` angles and the original `cam_to_world` matrix from the scene. 2) update the `yaw` and `pitch` based on the mouse movement offsets like in the tutorial. 3) form a rotation matrix R based on `yaw` and `pitch`, then form a new `cam_to_world` matrix by multiplying the original `cam_to_world` matrix with R . (Don't overwrite the original `cam_to_world` matrix!)

For rotation, it might be tempting to keep only one `cam_to_world` matrix by keep multiplying it with new rotation matrices. However, this is going to produce unintuitive behavior (try it!) since `yaw` and `pitch` rotations are not *commutative*: applying `yaw` first then `pitch` will produce different result compared to applying `pitch` first then `yaw`. As a result, when you chain together many `pitch`s and `yaw`s matrix rotations, they will not represent the desired rotation. Yes, rotation is weird. This is why you should explicitly store the `yaw` and `pitch` angles and modify those instead.

Passing parameters in callback functions. If you dislike global variables as much as me, you would like the functions `glfwSetWindowUserPointer` and `glfwGetWindowUserPointer`. You will use it like this:

```
void mouse_callback(GLFWwindow* window, double xpos, double ypos) {
    StructIWanttoPasstoCallback *data_ptr =
        glfwGetWindowUserPointer(window);
}

GLFWwindow* window = glfwCreateWindow(width, height, "Balboa", NULL, NULL);
StructIWanttoPasstoCallback data = ...;
glfwSetWindowUserPointer(window, &data);
glfwSetCursorPosCallback(window, mouse_callback);
```

Debugging. Debugging OpenGL (and other graphics API) programs is painful: if you do one thing wrong, you'll likely get a black screen. The learnopengl tutorial provides [useful tips](#) for debugging. To debug shaders, it's particularly useful to use a debugger such as [renderdoc](#). Unfortunately, none of the existing OpenGL debuggers work on MacOS anymore (Apple makes it extremely hard to develop OpenGL on MacOS because they want people to use Metal). For MacOS users, a potential debugging strategy is to emulate the shader on CPU: write the same code on CPU and print out the values, and see if it does what you expect. It's going to be painful regardless, I'm sorry. On the other hand, this is a fruitful research area that awaits innovation to make things better!

For the 3D transformation, copy your Homework 2 code to the `parse_transformation` function in `hw3_scenes.cpp`. Implement the rest in `hw_3_3` in `hw3.cpp`.

Test your OpenGL rendering using the following commands:

```
./balboa -hw 3_3 ../scenes/hw3/two_shapes.json
./balboa -hw 3_3 ../scenes/hw3/cube.json
./balboa -hw 3_3 ../scenes/hw3/spheres.json
./balboa -hw 3_3 ../scenes/hw3/teapot.json
./balboa -hw 3_3 ../scenes/hw3/bunny.json
./balboa -hw 3_3 ../scenes/hw3/buddha.json
```

For `two_shapes` and `cube`, they should render to the same images as the previous homework (before you move the camera yourself). The rest are new scenes. (`teapot.json` is a higher-resolution version that has 10 times more triangles!) Record a video of you moving the camera for each scene and save them as:

```
outputs/hw_3_3_two_shapes.mp4
outputs/hw_3_3_cube.mp4
outputs/hw_3_3_spheres.mp4
outputs/hw_3_3_teapot.mp4
outputs/hw_3_3_bunny.mp4
outputs/hw_3_3_buddha.mp4
```

Acknowledgement. The bunny model was scanned by Greg Turk and Marc Levoy back in 1994 at Stanford, so it is sometimes called the [Stanford bunny](#). The texture of the bunny model was made by KickAir_8p who posted the scene in [blenderartists.org](#). The buddha texture was generated by Kun Zhou et al. for their [Texturemontage](#) paper.

Bonus: textures (15 pts). Read the [Textures](#) chapter of learnopengl.com and implement textures for the shapes above. We have provided the UV maps for the models except `two_shapes` and `cube`. I have also included the original textures I used to produce the vertex colors for `teapot`, `bunny`, and `buddha`.

4 Lighting (25 pts)

For this part, read the chapters of [Colors](#) and [Basic Lighting](#) in the tutorial, and implement some basic lighting in our viewer. Be careful about the transformation of the normals! Use the vertex colors or texture colors as the `objectColor` equivalent in the tutorial. Let's assume `ambientStrength=0.1`, `specularStrength=0.5` and `lightDir` is at `normalize(vec3(1, 1, 1))`. Note that you can extract the camera position by looking at the fourth column of `cam_to_world`. Also note that the tutorial claimed that it is adding a point light. It is actually a fake point light since it does not have a distance squared falloff. I decided to simplify that part and just use a parallel directional light anyway, which means there is no `lightPos` in our case.

The way the tutorial does the lighting requires defining vertex normals (an alternative is to use face normals, but it often looks uglier). We have provided vertex normals for the following scenes:

```
./balboa -hw 3_4 ../scenes/hw3/spheres.json
./balboa -hw 3_4 ../scenes/hw3/teapot.json
./balboa -hw 3_4 ../scenes/hw3/bunny.json
./balboa -hw 3_4 ../scenes/hw3/buddha.json
```

Save your output as screenshots:

```
outputs/hw_3_4_spheres.png
outputs/hw_3_4_teapot.png
outputs/hw_3_4_bunny.png
outputs/hw_3_4_buddha.png
```

Bonus: lighting animation (10 pts). Add some animation to the light. Make it move the way you like, and submit a video recording of the animation.

Bonus: different types of lights (10 pts). Our light currently is a directional light. Implement point lights and spot lights (see the [Light casters](#) chapter) in your renderer, and support [multiple lights](#).

Bonus: shadow mapping (20 pts). Implement a basic shadow map. See the [Shadow Mapping](#) chapter in `learnopengl`. Support of directional lights is good enough.

5 Design your own scenes (10 pts)

We're at the fun part again. Design your own scene and render it using your new renderer!