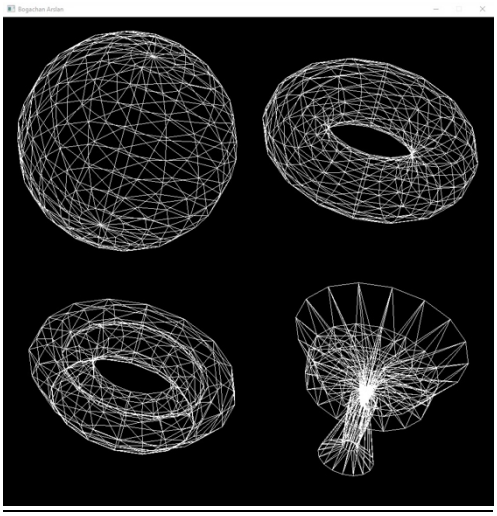
Boğaçhan Arslan
26569
CS405

20/12/20

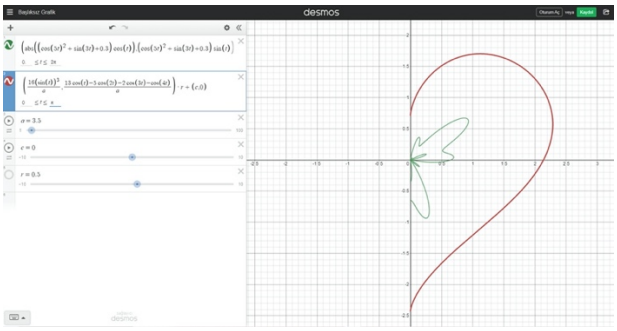
3D Project Part 1 – Report

First of all, to create an executable OpenGL application with multiple scenery, I decided that I should store the contents and features of each scene separately and use the same drawing functions for any scene as an input. To accomplish that behavior, I have created a struct called “SceneVars” that stored the VAO’s in a vector, rotation speed, program (shaders), transforms in a vector, the scene identifier etc. The keyboard input, received from the GLFW callback and stored as a global variable, determined which scene to be currently drawn based on its identifier, which also has a pointer to it that is globally stored. The bindings and drawings have been done accordingly.

Scene 1



For the first scene, we have 4 shapes (a sphere, a donut, a heart shaped donut, and the goblet of fire?). All the equations are parametric. The sphere’s equation is a half-circle and the donut’s is a full circle (which were already implemented in class as ParametricHalfCircle and ParametricCircle, respectively). The equations for the shapes in the bottom and their non-rotated 2D graphs can be found below in order for ParametricHeart and ParametricBat.



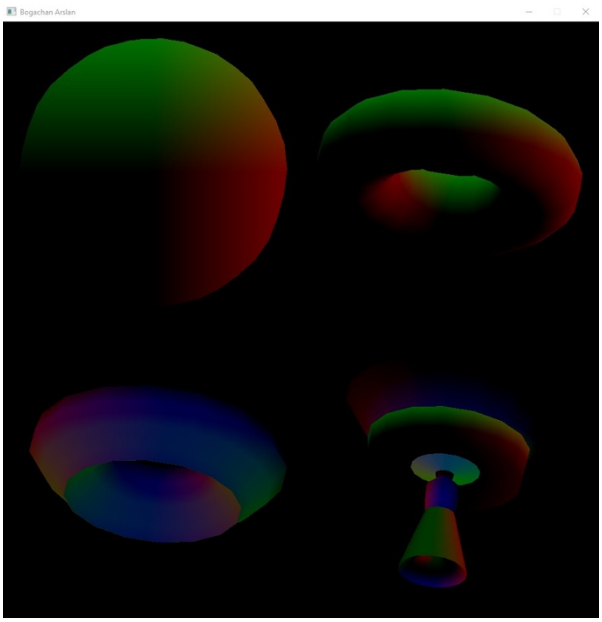
To fit the 4 shapes into the single frame, I have applied the following translations to the sphere, donut, heart, and goblet in this order:

```
// Sphere
    transform = glm::scale(glm::vec3(0.45));
    transform = glm::translate(transform,glm::vec3(-1.1, 1.1, 0));
// Donut
    transform = glm::scale(glm::vec3(0.45));
    transform = glm::translate(transform, glm::vec3(1.1, 1.1, 0));
// Heart
    transform = glm::scale(glm::vec3(0.30));
    transform = glm::translate(transform, glm::vec3(-1.6, -1.6, 0));
// Goblet
    transform = glm::scale(glm::vec3(0.40));
    transform = glm::translate(transform, glm::vec3(1.2, -1.3, 0));
```

The only thing left was to instruct OpenGL not to fill in our polygons (triangles) but only color the edges. This was done via the command:

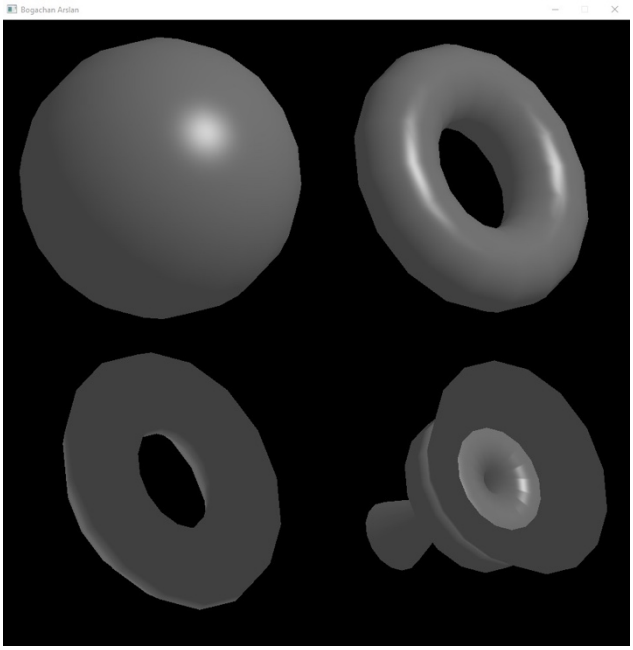
```
if(Globals.current->sceneidentifier == 'q')
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
else
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
```

Scene 2



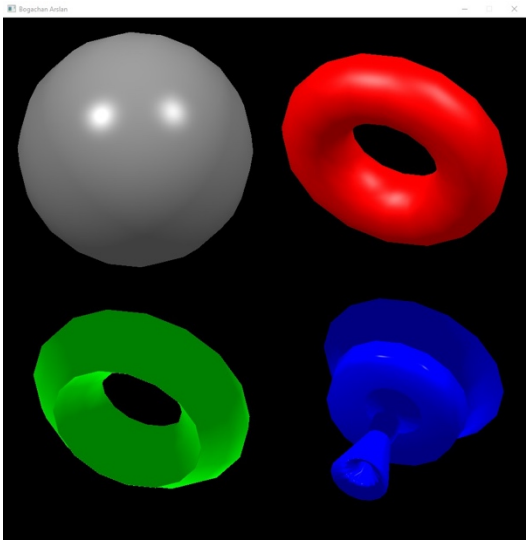
Scene 2 is initialized from scene 1 since the shapes and transformations are exactly the same. The program only differs in the sense that the out_color of the fragment shader is the normal vector, instead of white as in scene 1.

Scene 3



Scene 3 is also initialized from scene 2, yet the shader programs differ heavily. This time we have used the Blinn-Phong reflection model to illuminate the objects. We have set the surface colors to gray (0.5,0.5,0.5) and the shininess to 64 as instructed. The lighting model was accomplished by initially setting the color to black, and then adding the values of different lightings to it. The first is ambient, with a coefficient of 1 and a color of gray (same). The second is a directional light (-1,-1,1) with a color of (0.4,0.4,0.4). This light both contributes to the diffuse light and specular.

Scene 4

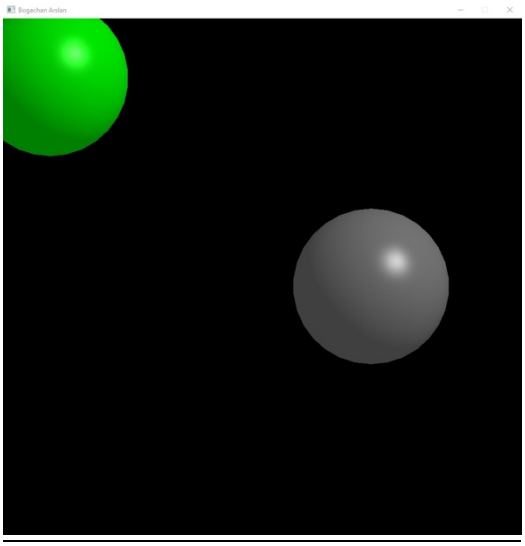


Scene 4 is initialized from scene 3. The program is yet again different. In the fragment shader, the objects (and the vertices) are assigned their colors (gray, red, green, blue) depending on which quartile of the screen they are in, via the glsl code provided below:

```
vec3 surface_color = vec3(0.5,0.5,0.5);
if(vertex_position.x > 0 && vertex_position.y > 0) surface_color = vec3(1,0,0);
else if(vertex_position.x > 0 && vertex_position.y < 0) surface_color = vec3(0,0,1);
else if (vertex_position.x <0 && vertex_position.y < 0) surface_color = vec3(0,1,0);
```

This could have also be achieved with assigning numbers to each shape and determining the color based on which object's vertex it is based on the number. However, I haven't been able to figure out passing in parameters to shaders until towards the very final tasks of the project so I came up with this workaround.
The lighting is the same as scene 3, with an additional directional light based on the mouse location. The light again contributes to diffused and specular lighting, but it has a directional vector of (mouse.x, mouse.y, -1). Also the shininess of the objects were determined via a similar approach of quantile detection. The sphere's is 128, donut's is 32, heart's is 64 and goblet's is 512.

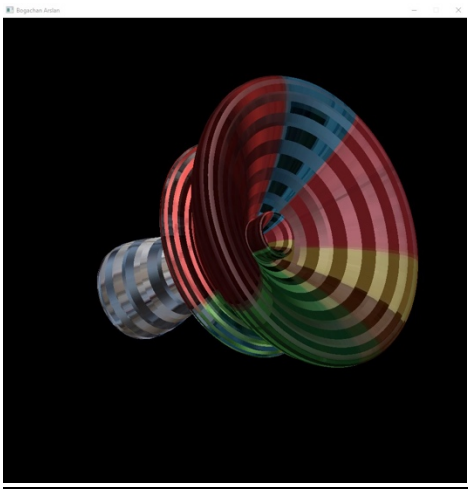
Scene 5



This scene is initialized empty. Two sphere VAO's are added with the same qualities as in scene 1, however they are scaled to a factor of 0.6 (not 0.3 because for some reason when dynamic positions were assigned, it seemed like the scaling was done twice so I had to enlarge the spheres to reverse that effect). Mouse position was already being tracked by GLFW's cursor callback and stored as global variables. They were also passed to the shaders as a uniform dvec2. To position the first sphere aligned with the mouse, the vertex positions in the vertex shader was incremented by (mouse.x *2, mouse.y *2). The multiplication with 2 was necessary because of the correction of the shrinking effect I've previously mentioned. To determine the next position of the chasing sphere was more complicated. For that, I had to use the previous position of it each time, so creating a global variable that stores the value and updates it with the glm::mix function as instructed as long as this scene is active

made sense. A uniform parameter is also created to pass the value into the shader, so the position of the chasing square is the vertex position, plus the chase position calculated by the mix function. The surface color of the running sphere is determined in the fragment shader by the distance between the mouse position and chase position, and is red if it is smaller than 0.6.

Scene 6



This scene contains a single object, which is a ParametricBat with 100x100 samples, instead of the 16x16,which makes it a goblet. However, this time it is covered with a texture. The texture is actually an image, wrapped around this parametric curve. The image is originally the following:

Obviously, the image is not directly stacked on the shape. The way it was done is benefited from an external library called stb image, which allowed an easy way of lading an image as a character array. However, to implement a texture, I had to extend the VAO struct (with an additional constructor) and the GenerateParametricShapeFrom2D function to receive another vector (dvec2) for the uv values. The only difference in drawing was that if this scene was active, the texture was activated and bound as well.
Both shaders also received another in parameter for the uv values. The fragment shader, using the texture method and uv values, calculated the surface color at each point. I also wanted the colorful part of the colorful part to stretch towards the top of the shape, so I. divided the uv y values by two, and to control repetition divided x's by 3.
This scene has ambient lighting with a color of (0.7,0.7,0.7) and a diffused and specular directional light that is controlled with the mouse.