# Report

## Run

Generate a solution for vs2022 using premake5 tool, build and run it through Visual Studio

## Warning

The project generates warnings when building, but most of them are from third-party libraries such as rapidobj, except one from main.cpp: *'aWindow': unreferenced formal parameter*, which is because the glfw_callback_motion_ function body does not use the aWindow variable. However, the function declaration is fixed and the aWindow variable cannot be removed, so this warning is not resolved

## Task

### Object Groups

The theme of this project is puppy, so all the items are puppy-related, including puppy bed, puppy house, toy ball, dog food bows and puppy pad.

### Complex Objects

Of the above objects, three are constructed in code, which are consisted of combinations of basic geometry. Five pillars, four walls, one roof in the house is made of cubes, the wall sandwiched between the two roofs is made of trigonal cones and the puppy pad is made of several cylinders. As it is necessary to draw them many times, these are abstracted as functions which read the vertices corresponding to the geometry (or necessary parameters such as radius, height, etc.), the *SimpleMeshData* structure, the colour, texture, and positions are added to the *SimpleMeshData* structure in counterclockwise order according to the provided or calculated vertices. Normal vectors are computed on the edges of triangles.

```
SimpleMeshData makeCube(Vec3f vertices[], SimpleMeshData simpleMesh, Vec3f color,
bool texture)
SimpleMeshData makeTriangularCone(Vec3f vertices[], SimpleMeshData simpleMesh,
Vec3f color)
SimpleMeshData makeCylinder(Vec3f topCentre, float radius, float height, Vec3f
color, SimpleMeshData simpleMesh)
```

The food bowl is drawn in a different method. Based on two approximate circles on each of the bottom and top surfaces, vertices of these four approximate circles are connected in order to end up with a thick bowl. Note that the vertices of inside and outside of the bowl are added in a different order (the normal vectors are also in opposite directions).

## Perspective Projection

The effect of a perspective projection is that the near is large and the far is small. The **glm** library used in the project has a function to create a perspective projection matrix, which is used in the project in the following way

```
float aspect = fbwidth / fbheight;
if (abs(aspect - std::numeric_limits<float>::epsilon()) > static_cast<float>(0)) {
          projection = glm::perspective(0.78f, aspect, 0.1f, 100.0f);
}
```

Its four parameters are the field of view, the aspect, the near plane and the far plane, where the aspect value is adjusted every time the window has been resized so that it can adapt to changes in the window, and the other three parameters are set by convention.

## Camera

The camera is implemented based on the learnopengl tutorial and is adapted to CW's requirements and code structure. It is a simulation of the effect of moving the scene in the opposite direction. The first step is to build a right-handed coordinate system with the camera as the origin and a custom camera position, front vector and up vector.

```
glm::vec3 frontCam = glm::vec3(0.f, 0.f, -1.f);
glm::vec3 upCam = glm::vec3(0.f, 1.f, 0.f);
```

The **glm** library supports changing the world space into a camera-defined view space using the view matrix, and positionCam + frontCam is the object to observe

```
glm::mat4 view = glm::lookAt(positionCam, positionCam + frontCam, upCam);
```

The camera allows the user to move the camera via the keyboard, moving it in the corresponding direction when **W/A/D/S/E/Q** is pressed, the implementation is to change the corresponding position of the camera. To achieve frame-independent, the project tracks the time difference (the time it takes to render a frame) and multiplies this time difference by the speed. In addition, the multiplier of the camera speed can be adjusted via **crtl** and **shift** to accelerate and decelerate by adjusting the spped multiplier.
The mouse can also control the steering by reading the offset from the previous frame, changing the pitch and yaw angles (with upper and lower limits) and calculating a new unit vector pointing in front from the pitch and yaw angles.

## Diffuse and ambient shading

Ambient light refers to the product of the ambient light colour of the object, the ambient light constant of the light source, and the light colour

```
vec3 ambient = material.ambient * light.ambient * light.color;
```

Diffuse lighting requires the calculation of the angle between the normal vector of the object's surface and the vector from that fragment to the position of the light source, the smaller the angle between the two the brighter the object will be. The normal vector of the object is already stored in the *SimpleMeshData* structure, the position of the light source can be set by the uniform variable, the position of the vertex is passed from the vertex shader to the fragment shader, and the dot product of the two (greater than zero) is the component of the diffuse shading.

```
vec3 lightDir = normalize(light.position - v2fPos);
float diff = max(dot(norm, lightDir), 0.0);
```

## Animated Object

The animated objects in the scene are two cubic light sources that move in a circle, the angle of the circle is calculated according to time, so the animation is frame-independent. The user controls the animation of the objects via **P** in the keyboard. It is worth noting that each time the object stops the corresponding angle needs to be recorded so that the effect of the object resuming its movement is coherent. The principle is similar to the implementation of the camera speed adjustment, with key **J** accelerating and key **K** accelerating.

## Blinn-Phong Lighting Model

In addition to the previous ambient and diffuse shading, the complete Blinn-Phong lighting model includes specular and emissive shading. The specular shading uses a half-way vector, which refers to the unit vector on the angular bisector of the angle between the light to fragment and the fragment to view, and the closer the this vector to the normal vector, the more pronounced the effect of specular shading will be. Emissive shading is the light emitted The final full Blinn-Phong lighting is as follows

```
vec3 result = (ambient + diffuse + specular) * v2fColor + material.emissive;
```

## Material

Materials are implemented by defining material structures in the fragment shader and passing uniform variables. The structure contains the components of the intensity of the different shading effects, and the different materials are distinguished in the main program by setting different values.

```
struct Material {
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    vec3 emissive;
};
```

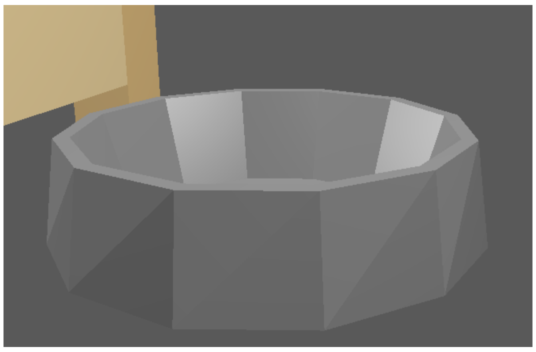

Figure1 Figure2

House is the main diffuse material, as it is mostly made of wood and rarely able to produce specular shading reflect, so there are almost no high light spots in the house (Figure1). The food bowl is mainly a specular material, the metal material is much more reflective so the food bowl will look brighter (Figure2). The dog bed is predominantly emissive material, it is more like it glows on its own and because the diffuse and specular reflections are minimal it does not have a strong sense of dimension, the image below shows a comparison between a normal dog bed (Figure3) and a predominantly emissive dog bed (Figure4).

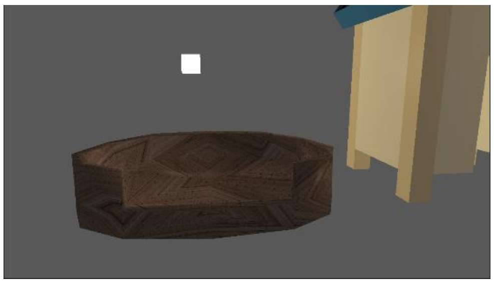
Figure3

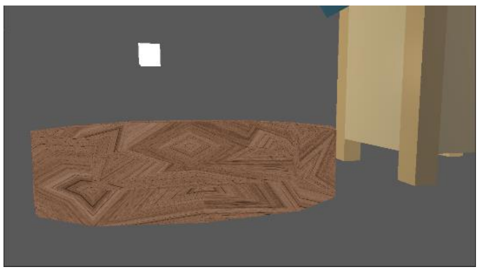
Figure4

## Texture Mapping

There are two objects in the scene that implements textures, the roof of the house and the dog bed. As the assignment required a portrait image to implement the texture, a regular roof (rectangule) was chosen as the object, with the four vertices on the left or right side corresponding to the four points of the texture image, and the other faces corresponding to the points in the lower left corner of the portrait, the result being a portrait on the sides of the roof and blue on the other faces.

For the dog bed, the wood image is utilized to represent that it is a dog bed made of wood. As the dog bed is an imported object, the coordinates of the texture are already included so it can be rendered directly.

The effect of the shading is mixed with the effect of the texture in the fragment shader to output the final fragment colour

```
oColor = texture(iTexture, v2fTexCoord) * vec4(result, 1.0);
```

## Multiple Light Sources

There are three cubic point light sources in the scene and the lighting effect of the object is a combination of these three sources, each of which is in a different position, so the vertex shader defines a light source structure with the necessary positions and a point light array containing all the point light sources. Since uniform variable is a complex array of structures when setting the information for the different light sources, the next statement is used.

```
glUniform3fv(glGetUniformLocation(programID, "pointLights[0].color"), 1,
glm::value_ptr(lightColor));
```

For ease of calculation, a function is defined in the shader to calculate the effect of a point light source.

```
vec3 pointLightEffect(PointLight light, vec3 v2fNormal, vec3 v2fPos, vec3 viewDir);
```

The main function iterates through the array of point sources and adds up the effects of all point sources to give the final effect.

```
vec3 result = vec3(0.f, 0.f, 0.f);
for(int i = 0; i < POINT_LIGHT_NUMBER; i ++){
       result += pointLightEffect(pointLights[i], norm, v2fPos, viewDir);
}
```

## External Wavefront File

The objects imported into the scene are the dog bed and the toy ball, both of which include vertex coordinates, texture coordinates and normal vector coordinates. The import implementation is based on the *load_wavefront_obj* function already provided in the exercise file. The **rapidobj** interface is fully encapsulated, so it is easy to add the corresponding coordinates of the objects to the *SimpleMeshData* structure by calling the different types of indices. The texture effects and lighting effects of the imported objects are displayed correctly.

## Object Movement

The object moving along the complex route in the scene is a toy ball. It moves in a counter clockwise loop in the shape of a square, each side moving in 5.f s. The program calculates the number of loops by obtaining the current time and thus determines where the ball is based on the shape of the square.

## Multi-texture

The object in the scene that achieves the effect of multiple textures is the roof, where it has previously applied the texture of the portrait image. In addition to this it has applied a specular texture map, the specular map is a picture with a white frame on a black background, the effect achieved is that the specular effect of side borders of the roof are more pronounced than tha of the middle (Figure5).

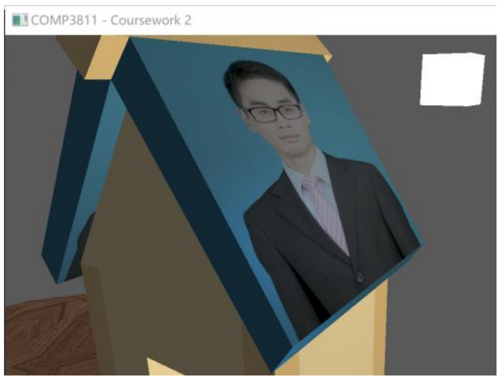

Figure5

When implementing a specular texture, the specular type in the material structure is set to the sampler2D and this texture and texture coordinates are applied when calculating the specular component

```
vec3 specular = light.specular * spec * texture(material.specular,
v2fTexCoord).rgb;
```

## Custom Shading Model

In addition to the Blinn-Phong lighting model, the scene also implements the Phong model, which treats specular shading differently. It considers that the smaller the angle between vector of light sources reflected by the object and the direction of observation, the more pronounced the light spot will be, and if the angle is greater than 90 degrees, the specular light component becomes 0. This is normal in most cases, but when the object has a very small reflection, the specular highlight radius will allow these opposite directions of light to have a significant effect on the luminance, at which point the Phong model is no longer accurate. However, due to the limitations of the scene in this project, there is little difference in the effect of the two lighting models. The roof is implemented as a Phong model and can be switched to the Blinn-Phong lighting model by pressing key **B**. The differences between the two model implementations are as follows.

```
if(blinn){
        vec3 halfDir = normalize(lightDir + viewDir);
        spec = pow(max(dot(norm, halfDir), 0.0), 64);
}
else{
        vec3 reflectDir = reflect(-lightDir, norm);
        spec = pow(max(dot(viewDir, reflectDir), 0.0), 16);
}
```

## Reference

JoeyDeVries. “Learn OpenGL, Extensive Tutorial Resource for Learning Modern OpenGL.” *Learnopengl.com*, 2019, learnopengl.com/.