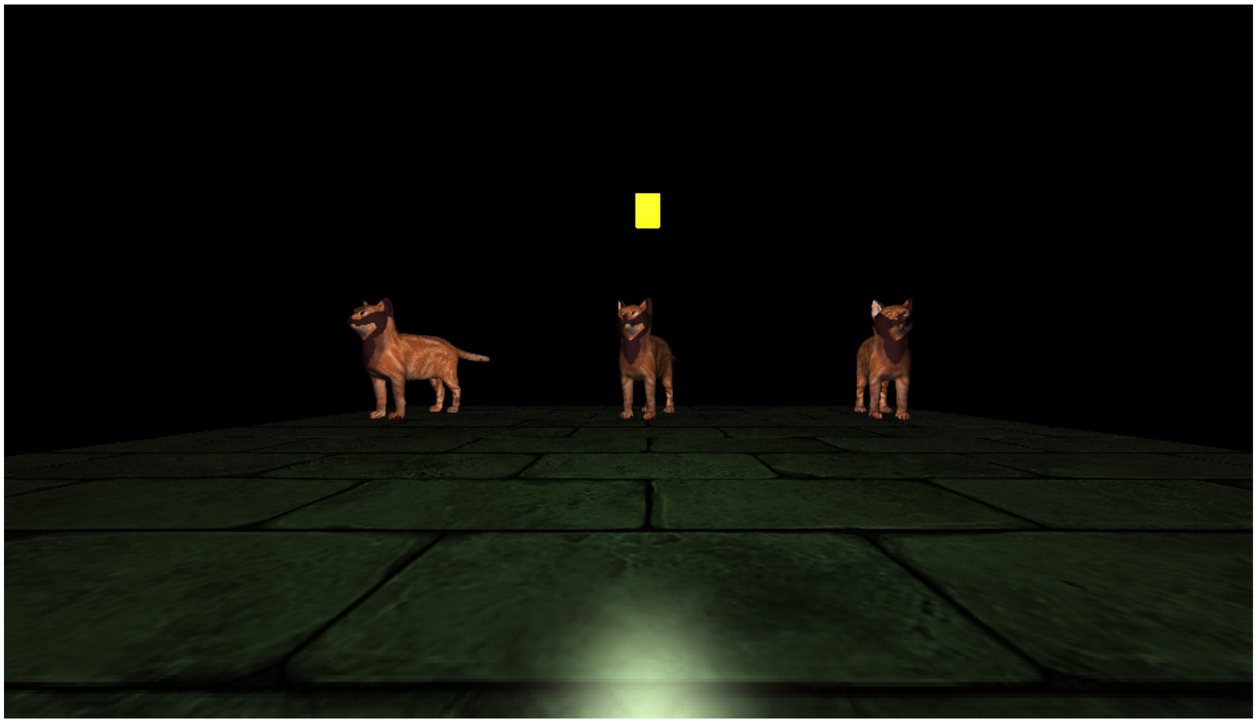
COMP3811-hw2-report

Overview

This work realizes the basic 3D scene, which includes the procedurally generated Cube model and the complex model read by OBJ files and corresponding textures.

All materials are shaded using BlinnPhong. The system has an animation function and calculates the motion function value according to time. The system also has basic interactive functions, which can control the mouse and keyboard to move the camera.



The Scene

'vmlib' Library

`mat22`, `mat44`, `vec2`, `vec3`, `vec4` are implemented and used in codes.

- Matrices are stored and accessed in row-major order.

When matrices are upload to GPU, it should be transposed to cow-major order, function glUniformMatrix3fv can do this by set the parameter 'transpose' to GL_TRUE.

```
void setMat4(const std::string& name, const Mat44f& mat) const
{
    glUniformMatrix4fv(glGetUniformLocation(ID, name.c_str()), 1, GL_TRUE, &mat(0, 0));
}
```

- Basic Transformations like rotation round axis, translation, scaling are implemented.
- Perspective projection matrix can be calculated in `mat44.hpp`

Camera

Camera.h implemented a first-person style 3D camera, which can move and rotate to capture the scene. The members `Front`, `Up`, `Right` indicate axis of camera coordinates, and `Yaw`, `Pitch` can easily rotate the camera orientation.

We calculate View Matrix in camera.h, which is the same implemented as `glm::lookAt(…)`:

```
// returns the view matrix calculated using Euler Angles and the LookAt Matrix
    Mat44f GetViewMatrix()
    {
        Vec3f f = normalize(Front);
        Vec3f u = normalize(Up);
        Vec3f s = normalize(cross(f, u));
        u = cross(s, f);
        return Mat44f{ {
                s.x, s.y, s.z, -dot(s, Position),
                u.x, u.y, u.z, -dot(u, Position),
                -f.x, -f.y, -f.z, dot(f, Position),
                0, 0, 0, 1} };

    }
```

Interactions

The following interactive features are supported:

- W/A/S/D to move camera position;
- LEFT SHIFT/CONTROL to change the movement speed;
- Dragging mouse to rotate view direction of camera.

Interactions are implemented mostly based on callbacks of GLFW.

GLFW provides many kinds of input. While some can only be polled, like time, or only received via callbacks, like scrolling, many provide both callbacks and polling. Callbacks are more work to use than polling but is less CPU intensive and guarantees that you do not miss state changes.

All input callbacks receive a window handle. By using the window user pointer, you can access non-global structures or objects from your callbacks.

Mouse Moving is implemented in `WindowControl::glfw_callback_mouse_(…)`

```
float lastX;
float lastY;
bool firstMouse;
void glfw_callback_mouse_(GLFWwindow* window, double xpos, double ypos)
  {
    if (glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_LEFT) == GLFW_RELEASE)
    {
      lastX = xpos;
      lastY = ypos;
      return;
    }
    if (firstMouse)
    {
      lastX = xpos;
      lastY = ypos;
      firstMouse = false;
    }

    float xoffset = xpos - lastX;
    float yoffset = lastY - ypos; // reversed since y-coordinates go from bottom to top

    lastX = xpos;
    lastY = ypos;

    camera.ProcessMouseMovement(xoffset, yoffset);

  }
```

When mouse moving in screen the function is called, and offsets of X/Y coordinates in screen between frames are calculated. If the left button of mouse is pressed, camera will process movements.

Callbacks of keys are implemented in the similar way, in function `WindowControl::glfw_callback_key_(…)`

Shading

Shading functions is based on 'Blinn-Phong' Illumination model.

Phong shading may also refer to the specific combination of Phong interpolation and the Phong reflection model, which is an empirical model of local illumination. It describes the way a surface reflects light as a combination of the diffuse reflection of rough surfaces with the specular reflection of shiny surfaces. It is based on Bui Tuong Phong's informal observation that shiny surfaces have small intense specular highlights, while dull surfaces have large highlights that fall off more gradually. The reflection model also includes an ambient term to account for the small amount of light that is scattered about the entire scene.

Blinn-Phong shading improves specular calculation in Phong shading by using halfway vector.

Following code shows blinn-phong algorithm in fragment shader:

```
// diffuse
vec3 lightDir = normalize(light.position - fs_in.WorldPos);
vec3 normal = normalize(fs_in.Normal);
float diff = max(dot(lightDir, normal), 0.0001);
vec3 diffuse = light.color * diff * material.diffuse * color;

// specular
vec3 viewDir = normalize(viewPos - fs_in.WorldPos);
vec3 halfwayDir = normalize(lightDir + viewDir);
float spec = pow(max(dot(normal, halfwayDir), 0.0001), material.shininess);
vec3 specular = spec * material.specular;

vec3 result = (diffuse + specular) * light.indensity;
```

Animation

Rotation of cats are calculated based on `sin(deltaTime)` , so cats are rotating periodically.

```
scene[i].world_transform = scene[i].world_transform * rot;
```

Textures

Textures are loading via `stb_image`.

```
unsigned char* data = stbi_load(path, &width, &height, &nrComponents, 0);
```

I initialize every texture by `glGenTextures(…)`, and send it to the model for shading:

```
glBindTexture(GL_TEXTURE_2D, textureID);

glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, format, GL_UNSIGNED_BYTE, data);
glPixelStorei(GL_UNPACK_ALIGNMENT, 4);
glGenerateMipmap(GL_TEXTURE_2D);


glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

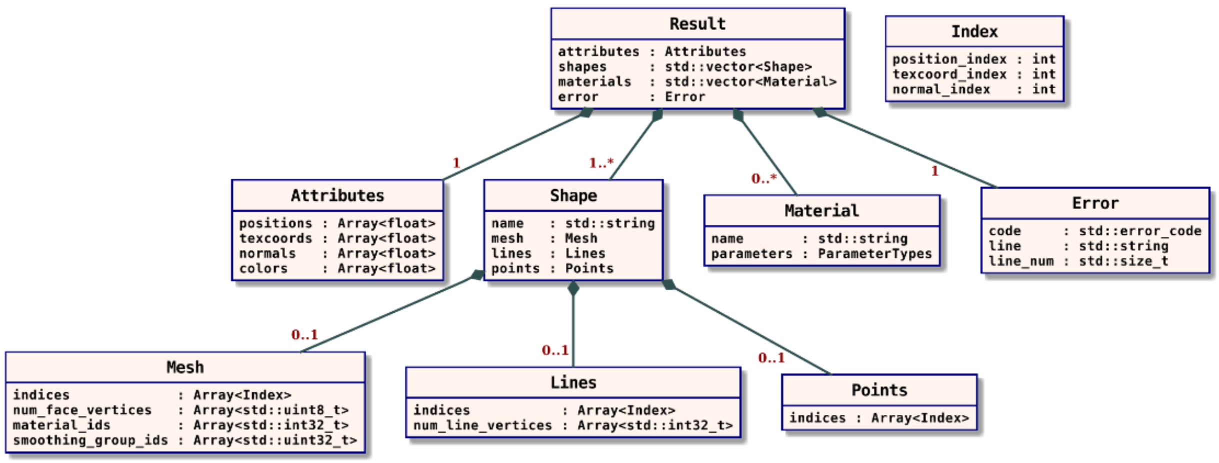
textures can be sampled in shader, and be used as albedo of 3D model:

```
vec3 color = texture(texture_diffuse, fs_in.TexCoords).rgb;
```

‘RapidObj’

Cats in the scene are from .obj files.

RapidObj can parse .obj files and store data in following way:



Structure of RapidObj

So we load files via following codes:

```
rapidobj::Result result = rapidobj::ParseFile(path);
```

```
if (result.error) {
  std::cout << result.error.code.message() << '\n';
  return nullptr;
}

bool success = rapidobj::Triangulate(result);

if (!success) {
  std::cout << result.error.code.message() << '\n';
  return nullptr;
}

std::vector<Vertex> vertices;
std::vector<unsigned int> indices;
for (const auto& face : result.shapes) {
  for (const auto& id : face.mesh.indices) {
    indices.push_back(indices.size());
        vertices.push_back(
          Vertex{
          result.attributes.positions[id.position_index*3],
          result.attributes.positions[id.position_index*3 + 1],
          result.attributes.positions[id.position_index*3 + 2],
          result.attributes.normals[id.normal_index*3],
          result.attributes.normals[id.normal_index*3 + 1],
          result.attributes.normals[id.normal_index*3 + 2],
          result.attributes.texcoords[id.texcoord_index*2],
          result.attributes.texcoords[id.texcoord_index*2 + 1]
      }
    );
  }
}
```