Interactive 3D Application Development
CSC_4IG01_TP Exercise - OpenGL Programming

Kiwon Um, Telecom Paris

The objective of this exercise is to develop an interactive graphics application using modern OpenGL (version 3.3 and later). The suggested goal is to implement a small artificial solar system, but open-ended thus not limited. Once you finish all the tasks described here, you can extend your codes further as you want.

1 Codebase

Your first task is to understand the overall structure of the provided codebase. Read through the `main.cpp` and the other files. It may take approximately 20-30 minutes. It is a good idea to keep the OpenGL documentation opened all the time. The provided codebase is written in C/C++; you may refer to any resources about C/C++ programming. The codebase uses the following libraries:

- OpenGL for accessing your graphics processor
- GLFW to interface OpenGL and the window system of your operating system
- GLM for the basic mathematical tools (vectors, matrices, etc.).

1.1 Build and run

The codebase uses *cmake* as a build system. You can easily build an executable via general cmake commands. (See Code 1.) You should make sure that you are running the commands under the right path.

Code 1. Build and run

```
cmake -B build    # under your source directory where CMakeLists.txt exists
make -C build
./tpOpenGL        # You should be careful with your working directory!
                  # When running, it will try to load ./fragmentShader.glsl, ./vertexShader.glsl, and ./media/*.jpg.
```

Congratulations! If everything goes well, you should be able to see a simple OpenGL window as on the left of Fig. 1; the middle and right of Fig. 1 are examples you may achieve. You can press `q` to quit. For the following tasks, note that pressing `w` allows visualizing the wire structure of your mesh (useful for debugging); you can use `f` to visualize the surfaces again.

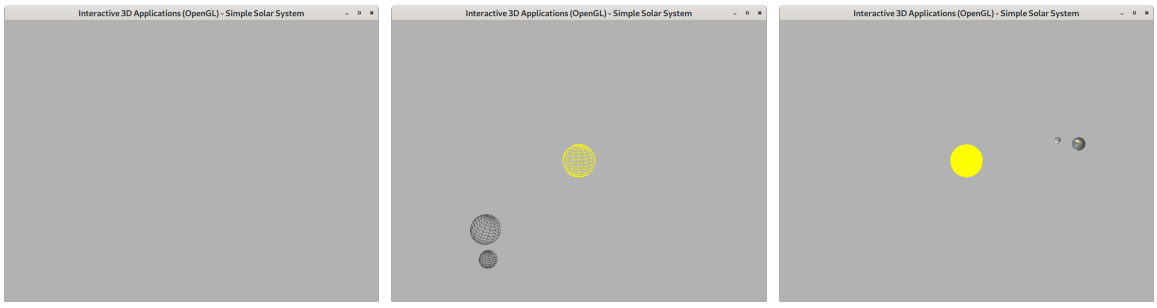


Fig. 1. Screen captures of (left) the very first run and (middle and right) two examples of this exercise.

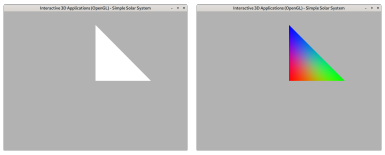
2 A single triangle

Let's start with drawing a triangle. You need to insert the coordinates of three vertices in the `g_vertexPositions` vector and its connectivity to the `g_triangleIndices` vector. You can do so in the `initCPUGeometry()` function. Note that these vectors are just arrays on the CPU side and that the function `initGPUGeometry()` takes care of creating the geometry on GPU side.

Code 2. Initialization of a triangle

```
// main.cpp ...
std::vector<float> g_vertexPositions;
std::vector<unsigned int> g_triangleIndices;
// ...
void initCPUgeometry() {
  g_vertexPositions = { // the array of vertex positions [x0, y0, z0, x1, y1, z1, ...]
    0.f, 0.f, 0.f,
    1.f, 0.f, 0.f,
    0.f, 1.f, 0.f
  };
  g_triangleIndices = { 0, 1, 2 }; // indices just for one triangle
  // ...
}
// ...
```

Code 2 will draw a single white triangle. Let's make it colorful by adding an attribute to your vertices. In addition to positions, each vertex needs to carry an RGB color value. To do so, you can add a new vector called `g_vertexColors` to your geometry definition on the CPU side first as in Code 3.



Code 3. Color attribute of a triangle

```
// main.cpp ...
std::vector<float> g_vertexColors;
// ...
void initCPUGeometry() {
  // ...
  g_vertexColors = { // the array of vertex colors [r0, g0, b0, r1, g1, b1, ...]
    1.f, 0.f, 0.f,
    0.f, 1.f, 0.f,
    0.f, 0.f, 1.f
  };
  // ...
}
// ...
```

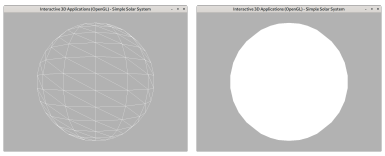
You need to pass the color data to your GPU via a new vertex buffer. You can simply replicate what is already done for the positions in `initGPUGeometry()`, but you should pay attention to the index of the vertex attribute. (Hint: `glEnableVertexAttribArray()` and `glVertexAttribPointer()`.)

Then, you need to update your shader codes to take into account this new input. To do so, you can edit the provided vertex shader program, `vertexShader.glsl`, as in Code 4.

Code 4. Vertex shader for the color attribute

```
// vertexShader.glsl ...
layout(location=0) in vec3 vPosition;
layout(location=1) in vec3 vColor;
uniform mat4 viewMat, projMat;
out vec3 fColor;
// ...
void main() {
  gl_Position = projMat * viewMat * vec4(vPosition, 1.0); // mandatory to rasterize properly
  // ...
  fColor = vColor; // will be passed to the next stage
}
```

3 Sphere mesh

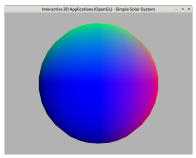


Let's aim for more interesting mesh, sphere. This will be used for rendering your planets later. Using a new class called `Mesh`, you can encapsulate the geometry manipulation codes such as initializing data (e.g., vertex positions, normals, texture coordinates, triangle indices, etc.) and binding buffers to GPU. Among others, let's equip this class with a static method, called `genSphere()`, that generates a sphere centered at the origin of radius 1. The structure of this class is shown in Code 5. (Hint: You can use the spherical coordinate system. At each pole, multiple vertices are collocated.)

Code 5. Mesh class

```
// main.cpp ...
class Mesh {
public:
  void init();   // should properly set up the geometry buffer
  void render(); // should be called in the main rendering loop
  static std::shared_ptr<Mesh> genSphere(const size_t resolution=16); // should generate a unit sphere
  // ...

private:
  std::vector<float> m_vertexPositions;
  std::vector<float> m_vertexNormals;
  std::vector<unsigned int> m_triangleIndices;
  GLuint m_vao = 0;
  GLuint m_posVbo = 0;
  GLuint m_normalVbo = 0;
  GLuint m_ibo = 0;
  // ...
};
```

3.1 Shading

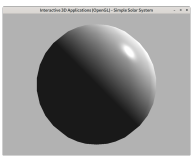


You will shade your sphere using the Phong lighting model. (See the lecture slides for more details.) To calculate correct color values, your mesh should have a correct normal vector at each vertex. Before diving into the lighting, make sure that your normals are properly calculated for your sphere. (Hint: You may not need any complex calculations since it is a sphere.) To debug, you can render the normals as colors. To do so, you can simply pass your normals through the two shaders as in Code 6. Again, do not forget the prerequisite codes in `main.cpp` handling a vertex buffer for this attribute.

Code 6. Shaders for coloring with normal vectors

```
// vertexShader.glsl ...
layout(location=1) in vec3 vNormal; // Now, the 2nd input attribute is the vertex normal.
out vec3 fNormal;

void main() {
  // ...
  fNormal = vNormal; // just pass to the next stage
}

// fragmentShader.glsl ...
in vec3 fNormal;
out vec4 color; // shader output: color of this fragment

void main() {
  // ...
  color = vec4(normalize(fNormal), 1.0);
}
```

If you confirm that your sphere geometry is all correct, it is time to start shading your sphere using the Phong lighting model. To this end, you need to define additional properties for light and material. You are considering only one instance of your sphere at this moment, and let's assume one directional light source; thus, you can write those properties directly into your shaders for now. Afterward, you will make your program more flexible such that you can pass different material properties per instance to the GPU. Code 7 shows the example codes.

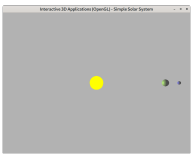


Code 7. Example codes for a quick implementation of the Phong lighting model

```
// main.cpp ...
void render() {
  // ...
  const glm::vec3 camPosition = g_camera.getPosition();
  glUniform3f(glGetUniformLocation(g_program, "camPos"), camPosition[0], camPosition[1], camPosition[2]);
  // ...
}
// ...

// fragmentShader.glsl ...
uniform vec3 camPos;
in vec3 fPosition;
in vec3 fNormal;
out vec4 color;
// ...
void main() {
  vec3 n = normalize(fNormal);
  vec3 l = normalize(vec3(1.0, 1.0, 0.0)); // light direction vector (hard-coded just for now)
  // TODO: vec3 v = calculate view vector
  // TODO: vec3 r = calculate reflection vector

  // TODO: vec3 ambient = set an ambient color
  // TODO: vec3 diffuse = calculate the diffuse lighting
  // TODO: vec3 specular = calculate the specular lighting

  color = vec4(ambient + diffuse + specular, 1.0); // Building RGBA from RGB.
}
```

4 Three planets

So far, you have not really made good use of different transformations for your scene. It was okay because you had been handling only a single object in your scene. Let's extend it to a small solar system of three planets: sun, earth, and moon. To this end, you will use the same sphere mesh yet three transformations and different properties. You need to adapt your codes for both the `main.cpp` file and shader programs such that you can use different values per planet. (Hint: `glUniform...()`) You can use the predefined constants for each transformation. See Code 8. Your scene setups are as follows:



- Your sun is at the origin $[0, 0, 0]^\intercal$ with the size of 1 in an arbitrary unit of your world space.
- Your sun is the only light source with the color of (1, 1, 1).
- Your sun is lit by only its own ambient lighting. Your sun is yellowish.
- Your earth is at $[10, 0, 0]^\intercal$ relative to your sun and half the radius of your sun's. Your earth is greenish.
- Your moon is at $[2, 0, 0]^\intercal$ relative to your earth and half the radius of your earth's. Your moon is blueish.

You will make your earth and moon rotate and orbit. But, for now, you can focus on rendering three static planets. You may need to adjust your camera to shoot all your three planets. *(By the way, these scales and transformations are not real. Do not get confused with your knowledge in astronomy.)*

Code 8. Predefined constants and variables for three planets

```
// Constants
const static float kSizeSun = 1;
const static float kSizeEarth = 0.5;
const static float kSizeMoon = 0.25;
const static float kRadOrbitEarth = 10;
const static float kRadOrbitMoon = 2;

// Model transformation matrices
glm::mat4 g_sun, g_earth, g_moon;
```

5 Animation

Let's make your scene more dynamic. Your earth is orbiting your sun and, at the same time, rotating itself around an axis that is slightly tilted. Simultaneously, your moon is rotating itself and orbiting your earth. For the scene setup, see Figure 2 and the following:

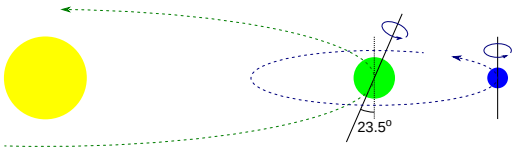


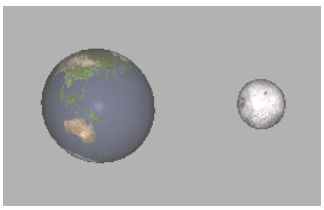
Fig. 2. Scene setup

- The orbital period of your earth is twice longer than its rotation period.
- The orbital period of your moon is twice shorter than the rotation period of your earth.
- The rotation period of your moon is the same as its orbital period; thus, your earth only sees the same side of your moon.

To create a smooth animation of your planets, you need a time source. GLFW provides a nice timer via the `glfwGetTime()` function. You can simply update the `update()` function where you can access the time in second since initialization.

6 Planet textures

You might notice that you cannot really see the rotations of your planets due to their smooth shading unless you switch to the wire-frame visualization. Let's make your planets look more realistic using textures. To use a texture with your mesh, the first key is the texture coordinates of each vertex. You need to add the texture coordinates to your `Mesh` class. You should update your `genSphere()` function accordingly as well. See Code 9.



Code 9. Texture coordinates of the `Mesh` class

```
// main.cpp ...
class Mesh {
  // ...
  static std::shared_ptr<Mesh> genSphere(const size_t resolution=16) {
    // ...
    // TODO: fill m_vertexTexCoords
    // ...
  }
  // ...
  std::vector<float> m_vertexTexCoords;
  GLuint m_texCoordVbo = 0;
  // ...
};
```

Once you have assigned the texture coordinates to your vertices, you need to load your texture images and make them ready to use. First, you can use the provided `stb_image.h` codes to load JPEG images, which are also provided for your planets. Then, you need to transfer the image data to your GPU memory. See Code 10.

Code 10. Load texture from file

```
// main.cpp ...
GLuint loadTextureFromFileToGPU(const std::string &filename) {
  // Loading the image in CPU memory using stb_image
  int width, height, numComponents;
  unsigned char *data = stbi_load(filename.c_str(), &width, &height, &numComponents, 0);

  GLuint texID; // OpenGL texture identifier
  glGenTextures(1, &texID); // generate an OpenGL texture container
  glBindTexture(GL_TEXTURE_2D, texID); // activate the texture
  // Setup the texture filtering option and repeat mode; check www.opengl.org for details.
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
  // Fill the GPU texture with the data stored in the CPU image
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);

  // Free useless CPU memory
  stbi_image_free(data);
  glBindTexture(GL_TEXTURE_2D, 0); // unbind the texture

  return texID;
}
```

Again, do not forget to update your rendering codes and shader programs such that you bind the right texture and access the texture coordinates and image when shading. The last hints are shown in Code 11.

7 Extensions

This is an open-ended exercise. You are strongly encouraged to further investigate any extensions you are interested in. You can find a list of potential directions as follows:

- Adding different shadings, e.g., the Gouraud shading and Blinn-Phong model
- Adding interactive navigation by manipulating the camera, e.g., zooming in/out and rotating the scene
- Adding more planets with a variety of shapes, sizes, etc.
- ...

8 Submission guideline

Once you finalize your exercise, you need to submit a packed/compressed file via Moodle:

- **by the midnight (23:59) on Thursday 31 October 2025**

Please make sure that your implementation compiles without errors and the application runs as you programmed. Your package must contain:

(1) Your final implementation files: **`vertexShader.glsl`**, **`fragmentShader.glsl`**, and **`main.cpp`**
(2) Any additional files **only if** you added on purpose
(3) A short PDF (maximum 2 pages) **report** (written in English) that contains a summary of what you achieved and screenshots you took from each task.
(4) A short **video** file (maximum 1 minute) that records an animation of your final implementation.

IMPORTANT: DO NOT include unnecessary files such as the executable and object files generated from your build.

Code 11. Example codes for using textures

```
// main.cpp ...
GLuint g_earthTexID;

void initGPUprogram() {
  // ...
  g_earthTexID = loadTextureFromFileToGPU("media/earth.jpg");
  // ...
  glUniform1i(glGetUniformLocation(g_program, "material.albedoTex"), 0); // texture unit 0
  // ...
}
// ...
void render() {
  // ...
  glActiveTexture(GL_TEXTURE0); // activate texture unit 0
  glBindTexture(GL_TEXTURE_2D, g_earthTexID);
  // ...
}

// fragmentShader.glsl ...
struct Material {
        // ...
        sampler2D albedoTex;  // texture unit, relate to glActivateTexture(GL_TEXTURE0 + i)
};
uniform Material material;

in vec2 fTexCoord;
// ...
out vec4 color;   // Shader output: the color response attached to this fragment

void main() {
  vec3 texColor = texture(material.albedoTex, fTexCoord).rgb; // sample the texture color
  // ...
  color = vec4(ambient + diffuse + specular, 1.0); // Building an RGBA value from an RGB one.
}
```