Rendering
Practical Assignment

Introduction to real time rendering with OpenGL

The objective of this assignment is to develop a modern interactive graphics application using OpenGL (version 4.5 and up). We will start from the archive and work mainly in the Main.cpp file. It's a good idea to keep the OpenGL documentation opened all the time: **www.opengl.org**. Note that **underlined text provide hyperlinks to useful documentations.** The base code (BaseGL application) is written in C++, one can refer to the resources from INF224 for details about C++ and Object-oriented programming. The following libraries are used by BaseGL:

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

For building BaseGL, we will use cmake:

```
mkdir build
cd build
cmake ..
cmake --build .
```

The main program and its dependencies will be generated using your local compiler. This should work on all platforms with cmake properly installed and configured. It has been tested on Linux and Microsoft Windows 10 (with Visual Studio 2015 and 2017). For now, the program is simply rendering a black background. The general philosophy of this program is:

- to define and manipulate a set of C++ objects representing the application entities e.g., scene's objects, such as camera, geometry and light, as well as the application's parameters – as global variables
- to initialize them in init ()
- to use them in render () to synthesize an image

I. Transform and Mesh Rasterization

Take a look at Main.cpp and take your time to read it entirely (should take at least 15 mins).

I.a Let's start by drawing a triangle. To do so, we need to insert the coordinates of its vertices in the vertexPosition vector, and its connectivity to the triangleIndices vector. We can do so in the initCPUGeometry function. Note that these vectors are CPU vectors and that the function init**GPU**Geometry takes care of creating the geometry on GPU side from the CPU geometry.

```
initCPUGeometry () {
	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
	};
	triangleIndices = { 0, 1, 2 };
}
```

This code now draws a single white triangle. Let's add some color by adding an attribute to our vertices: on top of a position, each vertices will now carry an RGB color value. To do so, let's first enrich our geometry definition on the CPU side by adding a new vector called vertexColors:

```
static std::vector<float> vertexColors;
```

and filling it with one RGB triplet per vertex:

```
void initCPUGeometry () {
...
	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
	};
...
}
```

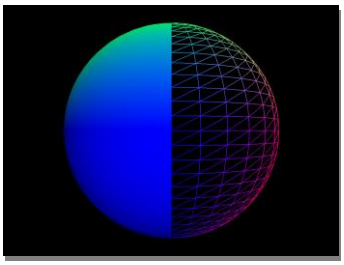
To account for this new attribute, we need to pass it to the GPU in a new vertex buffer by simply replicating what is already done for the positions in initGPUGeometry (). Last, we need the GPU code of our graphics pipeline to account for this new input. To do so, we will edit one of the two GLSL shaders provided with the archive, VertexShader.glsl :

```
...
layout(location=1) in vec3 vColor; // The 2nd input attribute is the vertex color (CPU
side: glVertexAttrib 1)

...

void main() {
   ...
    fColor = vColor; // Output passed to the next stage
}
```

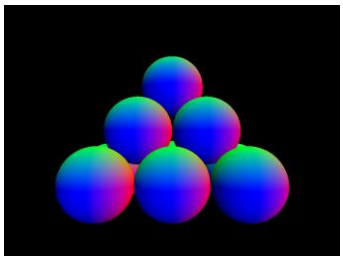
I.b We now want to generate a more interesting shape. Let's start by generating the triangulation of the surface of a sphere. To do so, encapsulate all the CPU geometry code (initialization, rendering) and data (vertexPositions, vertexColors, triangleIndices, GPU buffer identifiers) in a new class called Mesh and let's equip this class with a static method that generates a sphere centered at the origin, of radius 1.0:



```
class Mesh {
public:
	// Should properly set up the geometry buffer
	void init ();
	// Should be called in the main rendering loop
	void render ();
	// Should generate a unit sphere, meshed with resolution² vertices
	static std::shared_ptr<Mesh> genSphere (size_t resolution = 16);
};
```

Note that pressing F1 allows visualizing the wire structure of the mesh (useful for debugging).
Let's use this method to create and render a sphere in our program. For now, it is fine to set the position of the each vertex, normalized in the unit cube, as its color. It's a good idea to create and manipulate each Mesh using a shared pointer (std::shared_ptr), stored as a global variable.

I.c Add methods to generate a cube, a cone, a cylinder and a torus. We assume that each of them is normalized in the cube {[-1,1],[-1,1],[-1,1]}. Transformation may later be used to rescale them.



I.d We now want to place several objects in the 3D space. For that, we will add a transform to each mesh by making the Mesh class deriving from a new Transform class. This class should manipulate a translation vector, 3 angles defining rotation around the X, Y and Z axis, and a single scale value. At any time, this class should be able to provide a **model** transform matrix (i.e., a 4x4 matrix) built from its parameters. To do so, the GLM library does the heavy lifting: translate matrices, rotations matrices and scaling matrices can easily be generated and multiplied to yield the combined transform matrix provided on-demand by the Transform class. Let's use this new set of features to locate several meshes at different positions. And let's take this opportunity to improve the Camera class: make it a child of Transform to position the camera (remove the translation in the current computeProjectionMatrix) and use now the projection and view matrix of the camera, together with the model matrix of each mesh to update the current projection and modelView matrices sent to the GPU:

```
Void render {
        ...
        glm::mat4 projectionMatrix = camera.computeProjectionMatrix ();
        GLint projLoc = glGetUniformLocation (program, "projectionMat");
        glUniformMatrix4fv (projMatLoc, 1, GL_FALSE, glm::value_ptr (projectionMatrix));
        glm::mat4 viewMatrix = camera.computeViewMatrix ();
        glm::mat4 modelMatrix = mesh->computeTransformationMatrix ();
        glm::mat4 modelViewMatrix = viewMatrix * modelMatrix;
        GLint modelViewLoc = glGetUniformLocation (program, "modelViewMat");
        glUniformMatrix4fv (modelViewLoc, 1, GL_FALSE, glm::value_ptr (modelViewMatrix));
        ...
        mesh->render ();
        ...
}
```

II. Basic lighting, material and textures

In this part, we will start computing illuminated surfaces. In particular, to compute reflected light, each mesh vertex needs to carry a normal vector defining its local tangent plane. To do so, we can compute the normal vector at generation time (e.g., genSphere) and store it in a specific vertex buffer (normalVbo). Vertex colors can safely be ignored from now on.

II.a Let's start by defining a new structure in the FragmentShader.glsl file: a LightSource should have a position, a color and intensity. To do so, we add a new structure to the shader file and declare an instance of this structure as a uniform variable i.e., a GPU variable that can be modified from the host C++ application (CPU) :

```
struct LightSource {
        vec3 position;
        vec3 color;
        float intensity;
};
uniform LightSource lightSource;
```

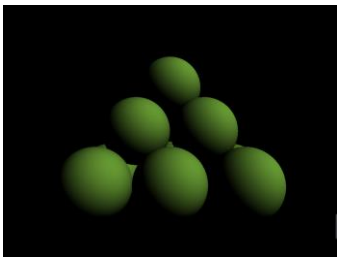
On the C++ side, in the initGPUProgram function, we can now fill up the new GPU uniform variables

```
glUseProgram (program);
glm::vec3 lightSourcePosition (3.0, 3.0, 3.0);
glm::vec3 lightSourceColor (0.4, 0.6, 0.2);
float lightSourceIntensity = 10.f;
glUniform3f (glGetUniformLocation (program, "lightSource.position"), lightSourcePosition[0],
lightSourcePosition[1], lightSourcePosition[2]);
glUniform3f (glGetUniformLocation (program, "lightSource.color"), lightSourceColor[0],
lightSourceColor[1], lightSourceColor[2]);
glUniform1f (glGetUniformLocation (program, "lightSource.intensity"), lightSourceIntensity);
```

Finally, let's use this new GPU structure in the fragment shader. To compile and test our new version of the program, let's do something simple: let's affect to the fragment shader output the color of the light source.

FragmentShader.glsl:

```
…
void main () {
	color = vec4 (lightSource.color, 1.0);
}
```



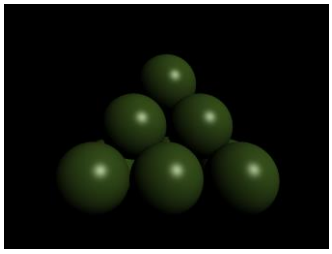
II.b Now that everything is ready, let's shade the surface of the objects. To start, we need an interpolated normal vector at each fragment, to compute light reflection. We can use the in/out mechanism to pass the normal from the vertex to the fragment shader. Note that the current model and view transforms should be taken into account to express the normal in the view frame. To do, we need to pass the normalMatrix i.e., the inverse *transpose* of the modelView matrix (viewMatrix * modelMatrix) to the vertex shader at rendering time, similarly to the way the modelView matrix is currently passed. This matrix reflects 3D transformations on vectors: let's use it on the normal in the vertex shader (be careful, transformation are 4x4 matrices, normal vectors are 3D). Again, GLM helps to easily compute this normalMatrix:

```
glm::mat4 normalMatrix = glm::transpose (glm::inverse (viewMatrix*modelMatrix));
GLint normalMatrixLoc = glGetUniformLocation (program, "normalMatrix");
glUniformMatrix4fv (normalMatrixLoc, 1, GL_FALSE, glm::value_ptr (normalMatrixi));
```

The position and normal vectors can be passed from the vertex shader to the fragment shader using the in/out mechanisms. In the fragment shader, they can now be used to compute a simple lambertian reflection:

FragmentShader.glsl:

```
void main () {
	vec3 wi = normalize (lightSource.position – fPosition);
	float lambertianTerm = max (0.0, dot (fNormal, wi));
	float radiance = lambertianTerm * lightSource.color * lightSource.intensity;
	color = vec4 (radiance, 1.0);
}
```



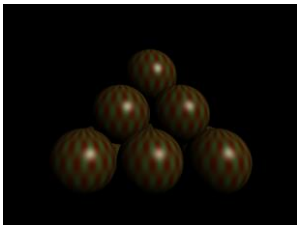
II.c For now, our meshes do not have any specific material and are basically "diffuse white". Just such as we did for light sources, let's enrich our GPU fragment shader with a new Material structure storing:

- an rgb albedo value (diffuse base color)
- a shininess coefficient (between 1 and 128, 16 as a default value)
- a diffuse coefficient kd and a a specular coefficient ks

With such a material model in hand, we can now evaluate reflectance using a variation of the Blinn-Phong shading model for each fragment:

```
FragmentShader.glsl:
void main() {
	vec3 n = normalize (fNormal); // Linear interp. does not preserve unit vectors
	vec3 wi = normalize (lightSource.position - fPosition);
	vec3 wo = normalize (-fPosition);
	vec3 fd = material.kd * material.albedo;
	vec3 wh = normalize (wi + wo);
	vec3 fs = vec3 (1.0) * material.ks * pow (max (0.0, dot (wh, n)),
material.shininess);
	vec3 Li = lightSource.color *lightSource.intensity;
	vec3 radiance = Li * (fd + fs) * max (0.0, dot (n, wi));
	color = vec4 (radiance, 1.0); // Building an RGBA value from an RGB one.
}
```

Let's experiment with different objects having different materials: to do so, we will define a Model class that encapsulates a Mesh and a Material, and use it to create a vector of objects modeling a simple scene. Again, models should be initialized in init () and used in render ().



II.d So far, materials are constant on an entire mesh. We now want to make their properties (e.g., diffuse albedo) vary over the surface. To do so, we will use textures. To map a mesh with a texture, the mesh needs to provide an extra per-vertex attribute: *texture coordinates* (or *UV coordinates*), which are 2D and indicate, for each vertex, which point in the unit square it corresponds to. This **parametrization** then allows to compute procedurally a value in 2D space or query a texture element (texel) in a normalized 2D image (i.e., bitmap texture), within in the fragment shader. This spatially varying value can be used instead of the constant value we had so far in our material definition.

Let's consider the case of a sphere: we will take its polar coordinates (phi and theta) as a basic parameterization. Therefore, we need to create a new vertex buffer attached to our VAO and storing, for each vertex, the angular components of its polar coordinates. Let's do the appropriate modifications to our program by modifying both the C++ code and the GLSL code. Then, to test it, we will simply define a procedural variation of the diffuse albedo in the fragment shader, based on the interpolated texture coordinates (see the images on the right). In particular, we can define a checkerboard function in the fragment shader, to alternate between 2 albedo colors on the surface based on the texture coordinates.

II.e Although procedural textures are useful, it is often handy to use actual bitmap images (grids of pixels) as textures. Let's do so by creating a new GPU object to store the texture that we will initialized in initGPUMemory () from an image (that can either be loaded or artificially generated):

```
GLuint texture; // OpenGL texture identifier
glEnable (GL_TEXTURE_2D); // Activate 2D texturing
glGenTextures (1, &texture); // Generate an OpenGL texture container
glBindTexture (GL_TEXTURE_2D,  texture); // Activate the texture
// the 4 following lines setup the texture filtering option and repeat mode, check www.opengl.org
for details
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
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);
// fills 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, image);
```

During rendering, we now need to bind the texture to the fragment shader.

```
Void render () {
        …
        glUseProgram (program);
        glActiveTexture (GL_TEXTURE0);
        glBindTexture (GL_TEXTURE_2D, texture);
        GLint texLoc = glGetUniformLocation (program, "material.albedoTex") ;
        glUniform1i (texLoc, 0);
        …
```

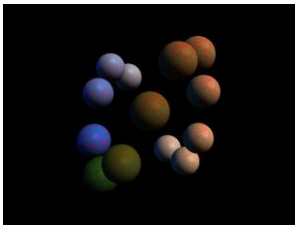
Of course, the fragment shader needs now to be updated to fetch texels from the newly defined texture and use them in the shading model. The "texture" function in GLSL can be used for this purpose, as it allows accessing textures stored in GPU memory.

II.Bonus

- Add several light sources
- Fill up the image on CPU side using Perlin noise.
- Use stb_image to load textures from files (bmp, jpg, png, etc).

III. Animation & Interaction

Up to now, everything is static. Let's add animation and interaction.



III.a The update function in the program is executed at each rendering pass. So far, it only gathers the time delay since the last call. Let's use it to animate various parameters in the application, such as:

- the transforms associated to each mesh to create animation trajectories for models
- the position and color of the light
- the material properties
- the texture coordinates (be careful, the GPU vertex buffers then need to be updated)

Let's be creative: simple basic algebra can yield impressive motions!

III.b Time can be passed to the GPU shaders to let them compute time evolving effects on the fly. Let's define simple wavy deformations in the vertex shader e.g., sums of sinus with various frequency and amplitude can easily produce non-trivial deformations when applied to move vertices along their normal vector.

III.c We now want to control the shininess of our materials using the keyboard, using the UP key to increase it and the DOWN key to decrease it. Let's use the `keyCallback` that we passed to GLFW to perform such interactions. We can then further define additional key bindings for controlling other parameters in the application.

III.d GLFW provide other *callbacks*, in particular for mouse event handling. Let's use them to control the camera view matrix and enable 3D navigation in the scene. Pressing the left button and moving the mouse, the camera should rotate, while the right button should provide zooming functionalities. Note that callbacks should be passed to the GLFW library in the initGLFW () function.