Programming Assignment 0: OpenGL Mesh Viewer

MIT 6.837 Computer Graphics

Fall 2021

Due September 17 at 8pm, Boston Time

Please note that this assignment is not at all included in your grade (even the extra credit will not count). This assignment is meant to get you familiar with the codebase and the problem set style in this class. You don't have to submit the assignment, but if you do (and have done a reasonable job for the requirements) **you will receive an extra late day** for the class.

1 New Changes

In Fall 2021, the starter code of the assignments contains a lightweight object-oriented OpenGL library called GLOO (openGL Object-Oriented) that is designed for this course. Previously the starter code was written primarily in C-style. The purposes of this change are:

1) To make use of modern C++11 features such as smart pointers (safe memory management) and move semantics (efficient copying of temporary objects).

2) To encourage object-oriented programming (OOP) to write modularized code while maximizing code reuse.

3) To separate the application-specific logic from the low-level OpenGL wrapper code. GLOO uses a component-based design at the application level that resembles scripting in Unity[1] game engine.

GLOO is influenced by the source code for CS148 offered at Stanford University (`http://web.stanford.edu/class/cs148/`). In comparison, GLOO is more lightweight and is targeted more at interactive applications. An overview of GLOO is given in the appendix.

For all the assignments, feel free to implement everything from scratch as long as your code meet the requirements detailed in each assignment.

2 Getting Started

This section details how to run the starter code on common platforms. We will be using a minimum version of OpenGL 3.3. On MacOS this should be automatically supported (see `https://support.apple.com/en-us/HT202823`). On Linux/Windows, you might need to upgrade your graphics drivers.

First make sure you have downloaded the assignment 0 starter code from Canvas. We use CMake[2] to generate build systems on different platforms.

2.1 Install building tools

Before compiling our code, we need to download the necessary building tools (e.g. C++ compiler, CMake).

[1] `https://unity.com/`
[2] `https://cmake.org/`

Linux Run (assuming you're on Ubuntu; otherwise it's analogous)

```
sudo apt-get install g++-5 build-essential cmake
```

Mac OS First make sure you have Xcode installed. Next install command-line tools if you haven't already:

```
xcode-select --install
```

It is recommended to use Homebrew[3] to download and install CMake. Once you have Homebrew installed, run

```
brew update
brew install cmake
```

Windows Download CMake Windows installer at `https://cmake.org/download/`. Download newest version of Visual Studio (community version is free at `https://visualstudio.microsoft.com/vs/`), and install the necessary C++ tools through Visual Studio (or in your preferred way).

2.2 Download source code

We use the following external libraries for a graphics application:

- GLAD[4] – OpenGL loader-generator
- GLFW 3.3.2 [5] – Window and context management
- GLM 0.9.9.8 [6] – OpenGL math library
- stb image [7] – single-file image loader/writer
- Dear ImGui [8] – C++ graphical user interface with no external dependency

First we need to download the sources (starter code + external libraries) - they are packaged in a single zip file for you on Canvas. Download that zip file and uncompress it to a folder.

2.3 Build assignment code

Linux & Mac OS To build the code for assignment 0, at the project's root directory, run

```
mkdir -p build/assignment0
cd build/assignment0
cmake ../..
make
```

The default build type is release (i.e. the code will be optimized for performance by the compiler). If you also want debug build (so that you can use gdb or lldb), simply add flag `-DCMAKE_BUILD_TYPE=Debug` when calling `cmake`.

After `make` finishes, an executable named `assignment0` is created in the current folder.

Windows You can use CMake-GUI to generate the build system of your choice. Run CMake-GUI will give you a window like Figure 1.

Set "Where is the source code" to the root directory of the uncompressed source code zip (containing `CMakeLists.txt`). Set "Where to build the binaries" to previous directory followed by `build/`

[3] `https://brew.sh`
[4] `https://github.com/Dav1dde/glad`
[5] `https://www.glfw.org/`
[6] `https://glm.g-truc.net/0.9.9/index.html`
[7] `https://github.com/nothings/stb`
[8] `https://github.com/ocornut/imgui`

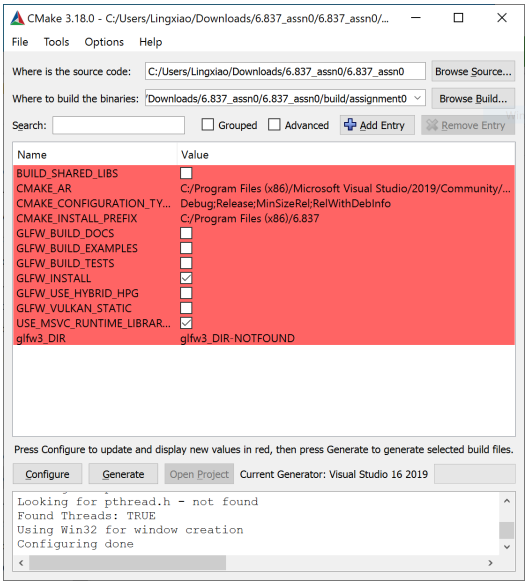

Figure 1: CMake GUI

`assignment0` (or wherever you want). Next click Configure. The default configuration should suffice. After configuring is done, click Generate. Here you will choose the build system you like, for instance Visual Studio 2019. After generation is done, you will find your favorite build system files (e.g. `*.sln` for VS2019) at the build directory you specified.

Now if you run `./assignment0` (or if you are on Windows double-click or run from an IDE) you should see a red teapot. You can use `WASD` keys to move around a bit (effectively the camera is being moved), but the range of motion is limited to a horizontal plane. It's now your job to make this application a bit more interesting by modifying the code.

Note: if you add a new source file (`.cpp`) in a directory, you will need to add it to your project. Otherwise you might encounter the linker error of "undefined symbols." To add a source file to the project, if your source file is under `assignment_code/assignmentX` or `gloo/` then you can rerun CMake and then build your project (if it is in some other directory you might need to modify `CMakeLists.txt` yourself). If you are using an IDE it might be able to automatically detect the new files, or you might have to import the new files manually.

3 Requirements

It is strongly recommended to take a look at the appendix where the starter code and GLOO library are explained in great details before working on the following requirements.

3.1 Color Changes

Add the ability to change the color of the displayed model. Right now, the color is set to $(0.5, 0.1, 0.2)$ (RGB), which is a boring darkish red. This is the default color if the node being rendered has no `MaterialComponent` attached. A *material* determines how the surface of an object will look and react to the light. Check out `gloo/shaders/PhongShader.cpp` for details.

Your task is to wire the `C` key to toggle through several other colors (feel free to choose which colors you want). To handle user inputs (e.g. mouse click, key press), GLOO uses a singleton[9] class `InputManager`. An example of handling keyboard events is in `BasicCameraNode::Update`. Remember that all the `Update` methods of every node in the scene tree are called in every frame by the `Application::Tick`. Hence `InputManager` will store events received between the previous frame and the current one.

[9]Singletons are like static classes but with more control over when to initialize.

A recommended way to implement color changes is to create a new `SceneNode` derived class and override the `Update` method. Let's say we create such a class called `TeapotNode`. As its name suggests, this class should be responsible for loading the teapot mesh, shader, and create the respective components. Creating a new class to assume responsibility and encapsulate part of the functionality of the system is a good software practice called single-responsibility principle[10]. The steps for writing the `TeapotNode` class involves creating a header file `TeapotNode.hpp` and a source file `TeapotNode.cpp` under `assignment_code/assignment0/`. Don't forget to add header guard in `TeapotNode.hpp`. You will also need to correctly handle the namespace scoping correctly. The simplest way would be to wrap everything in the `GLOO` namespace. Alternatively you can put a `using namespace GLOO` at the beginning (not recommended). Your `TeapotNode.hpp` should look like this:

```
#ifndef TEAPOT_NODE_H_ // This is the header guard
#define TEAPOT_NODE_H_

#include "gloo/SceneNode.hpp"

namespace GLOO {
class TeapotNode : public SceneNode {
  public:
    TeapotNode();
    void Update(double delta_time) override;
};
}

#endif
```

And your `TeapotNode.cpp` should look like this:

```
#include "TeapotNode.hpp"

// More include here

namespace GLOO {
TeapotNode::TeapotNode() {
  // Constructor
}

void TeapotNode::Update(double delta_time) {
  // Update
}
}
```

This new TeapotNode class you've written is meant to replace the additional `SceneNode` at the bottom of `MeshViewerApp::SetupScene`. Move the corresponding code from `MeshViewerApp::SetupScene` to your new class's constructor. This includes instantiation of a `PhongShader`, a `VertexObject` (including loading a mesh), and creating components `ShadingComponent` and `RenderingComponent` for them.

Lastly, don't forget to add an instance of it to the root in `MeshViewerApp::SetupScene`.

```
#include "TeapotNode.hpp"

// ...
void MeshViewerApp::SetupScene() {
  // ...
  root.AddChild(make_unique<TeapotNode>());
}
```

Before handling color changes, we need to first create a `MaterialComponent` for our `TeapotNode`. This can be done by (within the constructor of `TeapotNode`)

[10] https://en.wikipedia.org/wiki/Single-responsibility_principle

```
CreateComponent<MaterialComponent>(
    std::make_shared<Material>(Material::GetDefault()));
```

where `Material::GetDefault()` returns the default red material. You should now build and run your code to see if everything still works. You may need to rerun `cmake` to include the new source file `TeapotNode.cpp` you just created.

Next, you will implement the key handling in `TeapotNode::Update`. The attached `MatrialComponent` of `TeapotNode` can be accessed via `GetComponentPtr<MaterialComponent>()`, which returns a pointer to the requested component if it exists, and `nullptr` otherwise. To change the material of the teapot, you can get a reference to the material by `MaterialComponent::GetMaterial()`. The `Material` class provides getter/setter methods that can be used to change material properties like colors.

How do you handle color switches? A reasonable way to do this is to have the `C` key increment some sort of counter variable private to `TeapotNode` (don't forget to initialize it in the constructor) and use that variable to select a color. The ambient and diffuse colors of the material describe the color of the surface and should be the ones to modify. These colors are represented by `glm::vec3`, which is a vector of 3 floating-point components (check out the GLM webpage `https://glm.g-truc.net/0.9.9/index.html` for how to use GLM; it should be pretty straightforward). Feel free to also change other material properties.

There's a small caveat for handling toggling: you probably only want to toggle the color once when the user presses a button (unless you want to change the color in a continuous way). However the user typically will press down the same button across multiple frames, so you will need to record if the key in the last frame is released. Hence, inside `Update` you will need to write something like the following:

```
static bool prev_released = true;
if (InputManager::GetInstance().IsKeyPressed('C')) {
  if (prev_released) {
    ToggleColor();
  }
  prev_released = false;
} else if (InputManager::GetInstance().IsKeyReleased('C')) {
  prev_released = true;
}
```

3.2 Light Position Changes

Add the ability to change the position of the point light. In the starter code, a point light is placed at (0.0, 4.0, 5.0). Wire the arrow keys to change the position of the light in ways you want. This can be done quite similarly to the suggested method for the previous requirement (i.e. create a new derived class of `SceneNode` and override the `Update` method) except the quantity you want to change is the `Transform` associated with the node with the `LightComponent`. Check out `PhongShader::SetLightSource` in `gloo/shaders/PhongShader.cpp` to see how the point light properties get passed to the shader program. You may need to look up `https://www.glfw.org/docs/latest/group__keys.html` to find the correct key codes for the arrow keys. You will need to include `gloo/external.hpp` for the keycode related macros.

3.3 Directional Light

Add a new light type: directional light, which can be seen as a point light but is infinitely far away (such as the sun) that casts parallel light rays. This is a good opportunity to familiarize yourself with the basic GLSL and how GLOO works with shaders. First add a new derived class `DirectionalLight` of `LightBase` that has a `glm::vec3`-type member variable to represent the direction. Be sure to add a proper constructor and getter-setter methods similar to those in `PointLight`. Then you will need to modify the fragment shader `phong.frag` corresponding to `PhongShader`. Notice that a subclass of `ShaderProgram` (`PhongShader` in this case) always comes in pair with GLSL shaders (vertex shader `phong.vert` and fragment shader `phong.frag` in this case). In `phong.frag`, you will need to

- Add a new `struct` called `DirectionalLight` with fields similar to those of `PointLight`, except with `direction` of type `vec3` instead of `position`. Then declare a `uniform`-qualified parameter called `directional_light`. The `uniform` qualifier indicates the parameter will not change during the rendering call whereas the other parameters will change depending on the vertex (in vertex shader) or the fragment (in fragment shader). Fragment shaders such as `phong.frag` are small programs that run in parallel for each input fragment (a fragment is not exactly but almost like a pixel) on the GPU.

- Create a new function named `CalcDirectionLight`. The implementation of `CalcDirectionLight` should differ from that of `CalcPointLight` only in that the light direction `light_dir` is the negation of the directional light's ray direction. There is also no attenuation for the directional light.

Lastly you will need to update `PhongShader::SetLightSource` to pass the right parameters from `DirectionalLight` to the shader. If you follow the style of the code for point light, you will need to set `directional_light.enabled` first to false (because GLSL `uniform` parameters have undefined default values) and only set it to true if the light within `SetLightSource` is directional. In GLOO we use a simple forward renderer, so we need a rendering pass per mesh per light. That is why we have fields like `point_light.enabled` in `phong.frag`.

You can test if your code works by adding a new node to the scene, attach a `DirectionalLight` to it through `LightComponent`, set a proper light direction for it (the convention is that the light direction is towards the object, away from the light source), and set the light colors accordingly. Don't forget to add your new node to the root of the scene tree. The resulting scene should be much brighter now.

3.4 Populate the Scene (Optional)

Add more meshs, lights, etc. to the scene to make it look more interesting. For example, you can download meshes from the internet of OBJ format and load them using `MeshLoader::Import` that takes a path to the OBJ file relative to the `assets` folder in the root directory. Note that the implementation of `MeshLoader::Import` is quite basic and can only handle triangulated mesh. It loads materials but so far textures are not supported (you will need to implement texture support in assignment 5). You may need to extend it if you want to load complex meshes.

4 Extra Credit

Here are some ideas (sorted roughly by increasing level of difficulty) that might spice up your project. The amount of extra credit given will depend on the difficultly of the task and the quality of your implementation. In addition, feel free to suggest your own extra credit ideas! Just because it's not on this list doesn't mean we won't give you some extra points (although if it's a big addition, make sure you run it by the course staff first just to make sure).

To reiterate, in this assignment the extra credit does not count towards your grade. This is just to give you a feel of the format of assignments.

Easy

- Program the `R` key to spin the model as a rotating display would show.

- Modify the code so that the `C` key smoothly transitions between different colors (rather than just toggling it).

Medium

- Use Dear ImGui to allow editing teapot. This can include editing the position, rotation, scale of the teapot (using sliders) as well as the material color of the teapot using a color palette UI (e.g. `ImGui::ColorEdit4`).

- Implement a mouse-based camera control to allow the user to rotate and zoom in on the object. Credit will vary depending on the quality of the implementation.

Hard

- Large meshes are quite difficult to draw and process. For interactive applications, such as video games, it's often desirable to simplify meshes as much as possible without sacrificing too much quality. Implement a mesh simplification method, such as the one described in Surface Simplification Using Quadric Error Metrics (Garland and Heckbert, SIGGRAPH 97).

5 Submission

As a final step, write a `README.txt` or a PDF report including brief answers to the following questions:

- How do you compile and run your code? Have you encountered any difficulty compiling and/or running the starter code? If so please report the issues on piazza as soon as you found them. *This is very important for us because we are using a completely new codebase this semester, and the new remote setting has made it extra difficult for troubleshooting.*
- Did you collaborate with anyone in the class? If so, let us know who you talked to and what sort of help you gave or received.
- Were there any references (books, papers, websites, etc.) that you found particularly helpful for completing your assignment? Please provide a list.
- Are there any known problems with your code? If so, please provide a list and, if possible, describe what you think the cause is and how you might fix them if you had more time or motivation. *This is very important, as we're much more likely to assign partial credit if you help us understand what's going on.*
- Did you do any extra credit? If so, let us know how to use the additional features. If there was a substantial amount of work involved, describe what and how you did it.
- Got any comments about this assignment that you'd like to share? Was it too long? Too hard? were the requirements unclear? Did you have fun, or did you hate it? Did you learn something, or was it a total waste of your time? Feel free to be brutally honest; we promise we won't take it personally.

Upload the following to Canvas in a `.zip` or `.tar.gz` file

- Your code (just modified files will be fine).
- A compiled executable file built from your code (be sure to mention your platform in your `README.txt` or PDF report for the first question).
- The aforementioned `README.txt` or a PDF report file.
- Any additional files necessary to run your program.

A simple way to make sure you have included all the files is to upload the entire project folder (including `README.txt` or PDF report as `.zip` or `.tar.gz`) but **be sure to remove `external/` directory because it is too big - you shouldn't modify files in `external/` anyway**.

A An Overview of GLOO

This section gives a brief overview of GLOO and the assignment 0 starter code. It assumes you have basic knowledge of C++. If not, please check out the C++ tutorial tailored for this course (slides + video) on Canvas. Certain modern C++ features that are widely used in GLOO are further discussed in grey boxes, although the discussion here is by no means extensive. Good C++ references are `https://en.cppreference.com/`, stackoverflow, and your best friend Google. If you're never done object-oriented programming before, it is recommended to check out some online tutorials to make sure you understand common features of OOP such as encapsulation and polymorphism. For learning about OpenGL and the graphics pipeline in general, check out `https://www.khronos.org/opengl/wiki`. A good website for learning modern OpenGL with self-contained tutorials is `https://learnopengl.com/`. The official OpenGL reference website is `https://www.khronos.org/registry/OpenGL-Refpages/gl4/`.

A.1 Working with GLOO

A.1.1 Building a scene tree

A fundamental concept in GLOO is the tree structure of a scene. A *scene* is where you create your world of objects (e.g. meshes and materials to determine how they look) together with a camera and light sources. In GLOO the scene is represented as a tree, where the hierarchical structure of the tree could be interpreted both in the logic sense (e.g. a parent node can be viewed as a container for child nodes) or in the spatial sense (e.g. moving the parent node will also change the world positions of its children). This tree structure is implemented by the `SceneNode` class in `gloo/SceneNode.*pp`. Each `SceneNode` object can have an abitrary number of children `SceneNode` (implemented as `std::unique_ptr` for exclusive ownership of the child nodes). Every `SceneNode` has a `Transform` member variable as well as a dictionary of components which we will introduce next. For an illustration of the scene tree structure similar to the one in assignment 0, see Figure 2.

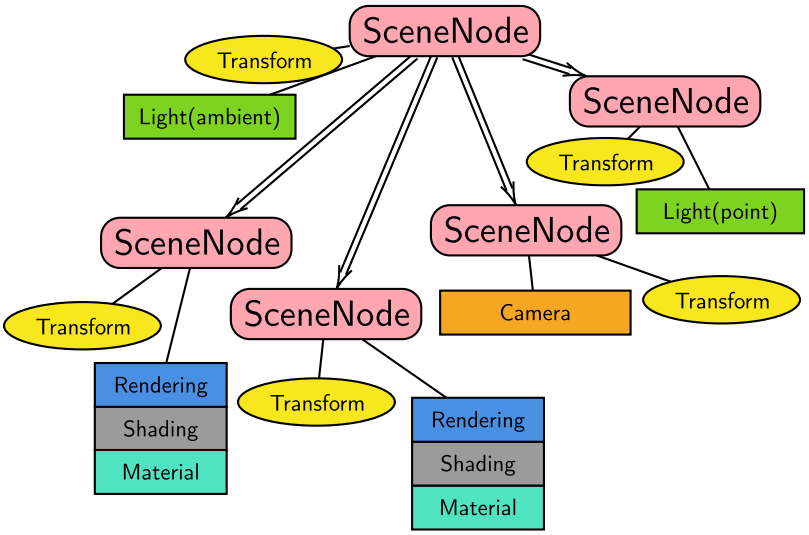

Figure 2: Scene tree structure of an example GLOO program similar to that of assignment 0. The double arrow $\Longrightarrow$ indicates parent-child relation, and the solid line indicates composition. Yellow ellipses represent `Tranform` objects associated with `SceneNode`. Rectangles represent components of a node (without the `Component` suffix). There are two lights in this scene (ambient light and a point light) and one camera.

- The `Transform` object attached to a `SceneNode` records the position, rotation, and the scale of the node. It provides getter and setter methods to change these fields, as well as methods to convert them to a 4-by-4 composite transformation matrix. In this assignment, the position (translation),

rotation, and the scale are absolute (so position will be the node's final position in the world space). You will implement hierarchical modeling in assignment 2 in which the transform could also be relative.

- A `SceneNode` contains *components* which enrich the node's behavior. A component is a derived class of `ComponentBase`, and a `SceneNode` manages the lifecycle of all its components via a hash map from `ComponentType` to `std::unique_ptr<ComponentBase>`. A minimal set of components are implemented in `gloo/components/` and you are encouraged to implement more to suit your needs. A node will be rendered (more on renderer a bit later) if it has a `RenderingComponent` which encodes mesh information. It will also need to have a `ShadingComponent` which sets the GLSL shaders and pass in the related `uniform` fields to the shaders. A node can optionally have a `MaterialComponent` to define how it looks with lighting; if there is no `MaterialComponent` a default material will be used.

  It is designed that a node will have at most one component of a certain type.

  - To create a component, you will need to use template functions `CreateComponent<T>(...)` and pass in the initializing argument list (similalr to `std::make_shared<T>(...)`). Alternatively you can use `AddComponent<T>(...)` and pass in a component as an rvalue of type `std::unique_ptr<T>` - i.e., you need to `std::move` it first.
  - To get a pointer to an existing component of type `T` on a `SceneNode`, use `GetComponentPtr<T>()`. If the requested component type does not exist, it will return `nullptr`. There is also a method `GetComponentPtrsInChildren<T>()` that gets all components of type `T` in the subtree. For instance, `GetComponentPtrsInChildren<LightComponent>()` will get all the lights in the subtree.
  - The base class `ComponentBase` provides a method `GetNodePtr()` which returns a pointer to the associated `SceneNode`, and `nullptr` if it is not associated with any `SceneNode`.

- The `SceneNode` class has a virtual method called `void Update(double delta_time)`. At each frame (i.e. one iteration of the main loop), `Update` of every node in the scene tree will be called where `delta_time` is the time elapsed since the last frame. The `Update` function will be where you process user input, update animations if there is any, and so on.

The `Scene` class is a thin wrapper around the root `SceneNode` of the scene tree with additional methods to set the active camera of the scene.

A.1.2 Rendering procedure

The code for rendering the scene has been provided for you and you don't need to touch those parts of the code until the last assignment. In GLOO we use a multi-pass forward renderer in which we render every mesh (`SceneNode` with a `RenderingComponent`) once per light (`SceneNode` with a `LightComponent`). The final color per pixel is the sum of all colors during each pass[11]. There must be a node with a `CameraComponent` (with camera specifications) that is set to be the active camera for the scene, so the scene will be rendered as if seen from this camera. The details of the renderer is in `gloo/Renderer.cpp`.

A.1.3 Application layer

In GLOO, the `Application` class manages the OpenGL context (via GLFW), the GUI, as well as the scene tree (as `Scene`) and the renderer (as `Renderer`). This is the top-level class and should only have one instance created in the `main` function. It also contains a method `Tick` that is called by the main loop at each frame. In `Tick` it will process UI and window events, call `Update` on the scene tree, and then render the scene and the UI. The `Application` class has a pure virtual method named `SetupScene`, and all its derived classes must implement this method to describe how to populate the scene at startup.

We have integrated a third-party GUI library called Dear ImGui (`https://github.com/ocornut/imgui`) in GLOO. We will only need ImGui for assignment 2, but you are encouraged to use it to add

[11] This does not scale well with a large number of lights and meshes. A better way is deferred rendering (`https://en.wikipedia.org/wiki/Deferred_shading`) in which we defer the light rendering to a latter stage after we have all the geometry information.

more interactive elements or for debugging purposes. To use it, override the virtual method `DrawGUI` in your application class (there is one per assignment).

A.2 Walkthrough of starter code

In this section we will give a walkthrough of the assignment 0 starter code to help you familiarize with the concepts in the previous section.

A.2.1 Main loop

Let's start by looking at the entry point of the starter code. In C++ the function called at program startup is the `main` function. In this case it is located in `assignment_code/assignment0/main.cpp`.

```
std::unique_ptr<MeshViewerApp> app =
     make_unique<MeshViewerApp>("Assignment0", glm::ivec2(1440, 900));
app->SetupScene();
```

In the first couple of lines in `main`, we create a `MeshViewerApp` object owned by a `std::unique_ptr` named `app`. Here we pass the name of the application and a window size of 1440×900 to the constructor of `MeshViewerApp`. We then call `app->SetupScene()` to set up the scene. The `MeshViewerApp` is a derived class of `Application`.

> Notice we did not use the usual `new` operator to create objects on heap — `std::unique_ptr` is one of the smart pointers introduced in C++11 that can automatically dispose the owned object when it goes out of scope. The goal of using smart pointers is to avoid having to use `new` and `delete` operators to manually manage the heap memory which is known to be error-prone even for the most experienced programmers. The smart pointer `std::unique_ptr` is "unique" in the sense that the ownership is exclusive. For instance you cannot copy a `std::unique_ptr`. To transfer ownership, you need to convert a `std::unique_ptr` to a rvalue reference `std::unique_ptr&&` via the move semantics `std::move`. We will soon see examples of this in the implemetation of `MeshViewerApp`. The function `make_unique<T>(...)`[a] is a shortcut for `std::unique_ptr<T>(new T(...))`. The goal is not to have occurrences of `new` and `delete` at all in your code.
>
> [a] This is added to `std` in C++14, but not in C++11, so we provide our own implementation

```
while (!app->IsFinished()) {
    TimePoint current_tick_time = Clock::now();
    double delta_time = (current_tick_time - last_tick_time).count();
    double total_elapsed_time = (current_tick_time - start_tick_time).count();
    app->Tick(delta_time, total_elapsed_time);
    last_tick_time = current_tick_time;
}
```

The second half of the `main` function is a top-level while loop that appears in any interative graphics application. The `std::chrono` library allows tracking time with high precision. The details here are not important. The body of the while loop repeatedly calls `app->Tick(delta_time, total_elapsed_time)` until `app->IsFinished()` returns true, where `delta_time` is the time in seconds passed since the last call to `app->Tick`, and `total_elapsed_time` is the total time in seconds since the program started.

A.2.2 Scene setup

As mentioned above, to write your own application (typically one assignment needs one application), you need to create a derived class inheriting `Application` in which you must define an overriding `SetupScene` method. In this assignment it is the `MeshViewerApp` class defined in `assignment_code/assignment0/MeshViewerApp.*pp`.

Let's take a closer look at the implementation of `MeshViewerApp::SetupScene`. This function is called in `main` after all necessary initializations (OpenGL, window context, etc.) are completed.

```
SceneNode& root = scene_->GetRootNode();
```

First we find the root of the scene tree and store it in `root`. The constructor of the base class `Application` already created a `std::unique_ptr` named `scene_` of an empty scene, so you only need to populate it in `SetupScene`.

```
auto camera_node = make_unique<BasicCameraNode>();
camera_node->GetTransform().SetPosition(glm::vec3(0.0f, 1.5f, 10.0f));
scene_->ActivateCamera(camera_node->GetComponentPtr<CameraComponent>());
root.AddChild(std::move(camera_node));
```

These lines add a camera node to the scene tree, set the node's position (using a `glm::vec3` that represents a vector with three `float` components), activate the camera, and then add the node to the root. We create a derived class `BasicCameraNode` of `SceneNode` because we want to handle keyboard events and move the camera accordingly. The constructor of `BasicCameraNode` adds a `CameraComponent` already so we don't need to add it again. As you might have suspected, a `CameraComponent` turns a node into a camera that inherits the position and the rotation of that node's `Transform`. Manipulating components is done via template helper methods in `SceneNode` as discussed in the previous section.

> The keyword `auto` is introduced in C++11 to automatically deduce the variable type from its initializer. Using it properly can make the code more readable (e.g. when the initializer is `make_unique` or an STL iterator). However abusing `auto` can obscure the intent of the code.

> Notice we pass `std::move(camera_node)` to `root.AddChild`. As mentioned before, directly passing `camera_node` is not allowed because `std::unique_ptr` cannot be copied. Here `std::move` is the move semantics introduced in C++11: it marks `camera_node` as an xvalue (an "eXpiring" value), so the move constructor of `std::unique_ptr` can simply move the underlying address of `camera_node` to itself instead of making a copy that is a lot more expensive. After `std::move(camera_node)` the variable `camera_node` should not be used again.

```
auto ambient_light = std::make_shared<AmbientLight>();
ambient_light->SetAmbientColor(glm::vec3(0.2f));
root.CreateComponent<LightComponent>(ambient_light);

auto point_light = std::make_shared<PointLight>();
point_light->SetDiffuseColor(glm::vec3(0.8f, 0.8f, 0.8f));
point_light->SetSpecularColor(glm::vec3(1.0f, 1.0f, 1.0f));
point_light->SetAttenuation(glm::vec3(1.0f, 0.09f, 0.032f));
auto point_light_node = make_unique<SceneNode>();
point_light_node->CreateComponent<LightComponent>(point_light);
point_light_node->GetTransform().SetPosition(glm::vec3(0.0f, 4.0f, 5.f));
root.AddChild(std::move(point_light_node));
```

The next portion of the code adds two types of light: ambient light and a point light. The lights themselves are managed by `std::shared_ptr` so their light properties can be shared.

> Unlike a `std::unique_ptr`, there can be multiple `std::shared_ptr`'s pointing to the same object on the heap. Reference counting is implemented for `std::shared_ptr` so that the count increases by one when copying a `std::shared_ptr`, and decreases by one when a `std::shared_ptr` goes out of scope. When the count reaches zero the object is disposed. It is recommended to use `std::unique_ptr` if you can as it is more lightweight, and only use `std::shared_ptr` when you need shared responsibilities for the allocated object. The function `std::make_shared` is similar to `make_unique` we just saw, except it returns a `std::shared_ptr`.

The properties of lights (e.g. diffuse/specular color), together with the material properties of the surface, determine how a surface would look. Technically the ambient light is not a light source. In GLOO, however, we still need to attach it to some node to make it visible to the renderer. The lights must be attached to scene nodes via `LightComponent`.

```
std::shared_ptr<PhongShader> shader = std::make_shared<PhongShader>();
std::shared_ptr<VertexObject> mesh = MeshLoader::Import("assignment0/teapot.obj").vertex_obj;
if (mesh == nullptr) {
    return;
}

auto node1 = make_unique<SceneNode>();
node1->CreateComponent<ShadingComponent>(shader);
node1->CreateComponent<RenderingComponent>(mesh);
node1->GetTransform().SetPosition(glm::vec3(0.f, 0.f, 0.f));
node1->GetTransform().SetRotation(glm::vec3(1.0f, 0.0f, 0.0f), 0.3f);
root.AddChild(std::move(node1));
```

Finally we are ready to create the teapot. First we create a `PhongShader` and put it in a `std::shared_ptr` named `shader`. Roughly speaking, a shader is a collection of programs that run on the GPU. The programming language for shaders is GLSL (`https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language`). See `https://www.khronos.org/opengl/wiki/Shader` for more details. In the starter code `PhongShader` implements Phong reflection model[12]. We also load an OBJ[13] file named `teapot.obj` using the static method `MeshLoader::Import`, which returns a `struct` containing mesh vertex information (plus information about groups and materials, but we won't need that here). The path we passed to `MeshLoader::Import` is relative to `assets/` in the root directory. In this case, multiple consumers can use the same shader or mesh, so we use `std::shared_ptr`. Finally we create the node for the teapot and attach the shader and the mesh via `ShadingComponent` and `RenderingComponent` respectively. The first thing you might want to do in assignment 0 is to add a `MaterialComponent` to this node and play with different material configurations. If there is no `MaterialComponent` attached, the default behavior, specified by the implementation of `PhongShader`, is to use a default dark red material. We set its position to be the origin and rotate it by 0.3 (radian angle) around the x-axis.

[12] `https://en.wikipedia.org/wiki/Phong_reflection_model`
[13] `https://en.wikipedia.org/wiki/Wavefront_.obj_file`