Warmup Guide

Spring 2022

Introduction

Hi there! This guide is designed to help you get setup for taking CS1950U. It will go through the basics of using the support code. However, this guide is in no way comprehensive. After completing the work here, please keep the following things in mind:

- The TA staff encourages you to look at and play around with the support code. This tutorial will only prepare you for the first week's assignment!
- You will be writing C++ code for the entire semester, exercising concepts that are not covered here. If you have not written much C++ code before taking this class we recommend that you read over the CS123 help sessions for C++, found here: http://cs.brown.edu/courses/csci1230/.
- In addition, although the TA staff has attempted to write comprehensive graphics support code, it will still be helpful (though not necessary) to understand the graphics pipeline and some basic OpenGL concepts such as Vertex Buffer / Array objects, Shaders, Textures, and FBOs. A good introduction to OpenGL can be found here: https://open.gl/drawing.

Developing Locally

Visit https://www.qt.io/download-qt-installer and download the Qt Installer. Use the installer to install Qt 5.15.2 with Qt Creator. Feel free to follow this tutorial from CS1230 which recommends some packages to get in the installer:https://github.com/cs123tas/labs/tree/master/lab00

Step 1: Creating a New Project

1. You will find the engine code stencil in /course/cs195u/asgn/engine on the department filesystem. Copy this folder into your home directory (probably something like /course/cs195u/engine)

2. Open Qt Creator (you can use the command /course/cs195u/bin/cs195u_qtcreator if you are on a department machine) and then open the engine/cs195u_engine.pro file within Qt Creator. You will be prompted to select your compiler and version of Qt. Build and run the project by clicking the green button in the lower left corner. You should see a black screen. Press Escape to close the program window. Your mouse should be in the center of the screen at each tick. If this is not working, then something might be wrong with your security preferences. Make sure to give Qt Creator and your game engine permission to control your computer. Once the program is manipulating the mouse properly, open up view.cpp. The View class represents a Qt object which will display your game. On line 31 of view.cpp change Qt::ArrowCursor to Qt::BlankCursor. Now the cursor will be hidden when inside the window.

3. In C++, every method that you write will be declared in a header file. You will put the implementation of each method in the corresponding source (.cpp) file. You will also declare your instance variables in the header file. Public and private keywords are used in blocks rather than for each method or variable, so all your public methods and variables go in one place, and all your private ones go in one place. Protected methods and variables also exist.

4. Use the F4 key to switch between the .cpp and the header file quickly.

5. Take a look at the View::View() method. This is the definition of the constructor of the View class. The constructor sets up your window, hides your cursor (now that you've modified it), and sets up a timer.

6. Take a look at the View::View() method. This is the definition of a destructor of the View class. The destructor of a class is what is called when an object goes out of scope. In this class, the destructor of View will be called at the end of the program.

7. Navigate to View::initializeGL(): this method starts the timer (which triggers tick events) and is where the Graphics object is set up.

8. Navigate to View::paintGL(): this will be called 60 times per second, and is where you will draw the current state of your game. The line

```
m_graphics->setClearColor(glm::vec3(0, 0, 0));
```

makes it so that when the screen is cleared, it is painted black, since glm::vec3(...) represents a color specified as amounts of red, green, and blue. In general, we will use glm::vec's to represent colors, positions, and more. Change the above line to m_graphics->setClearColor(glm::vec3(1, 0, 0)); When you build and run the project, the screen should now be red.

9. Navigate to View::mouseMoveEvent(QMouseEvent *event): this will be called every time the user moves the mouse.

10. Navigate to View::keyPress/ReleaseEvent(QKeyEvent *event): this will be called whenever the user presses or releases a key on the keyboard.

11. Navigate to View::tick(): this gets called 60 times a second and is where you will update your game! This is where most of your game happens.

Setting Up a Camera

As we discussed in lecture, when working in 3D space, we will need a virtual "camera" that determines where in our 3D world we are looking. You can think of this as literally placing a video camera in the middle of a room and putting the video from that camera onto the screen.

The support code provides a default camera class in src/graphics/Camera.h and src/graphics/Camera.cpp which has the following instance variables:

- glm::vec3 m_eye; // Keeps track of the position of the camera
- float m_yaw; // rotation around the y axis (the vertical axis) – the horizontal direction you are looking in

- float m_pitch; // angle with the xz plane (horizontal plane) – how much up or down you are looking
- float m_fov; // field of view – how wide is the camera's lens
- glm::vec2 m_screenSize; // size of the screen

Some of its public methods (declared in the header file, and fleshed out in the .cpp file) are:

- void setEye(glm::vec3 eye); // Set the position of the camera
- void translate(glm::vec3 amount); // Move the position of the camera by the specified amount
- void setScreenSize(glm::vec2 screenSize); // Set the screen size / aspect ratio of the camera
- void setPitch/Yaw(float pitch/yaw); // Set pitch / yaw of the camera
- void setLook(glm::vec3 look); // Instead of setting the pitch and yaw, you can directly specify the direction you want the camera to face

By default, the camera position is the origin, i.e. glm::vec3(0, 0, 0), the fov is pi / 3 radians (60 degrees), and the pitch and yaw are both 0 radians (0 degrees). This means that it will be looking down the positive z-axis.

1. Navigate to the end of the method View::initializeGL() in view.cpp, and set the m_camera member variable to a new camera: m_camera = std::make_shared<Camera>();
2. In the same function, set the position of the camera to (0, 1, 0)
   usingm_camera->setEye(glm::vec3(0, 1, 0));
3. Call m_graphics->setCamera(m_camera) so that the graphics object will use m_camera for rendering.
4. We need the camera to have an accurate value for the screen size, so navigate to the end ofView::resizeGL() and add the line: m_camera->setScreenSize(glm::vec2(w, h));

Drawing a Quad

If you run the project, you should still see a blank screen. Let's change that.

In the function View::paintGL(), add the following code:

```
m_graphics->clearTransform(); m_graphics->scale(20);
m_graphics->drawShape("quad");
```

The call m_graphics->clearTransform() sets the matrix applied to all subsequent shapes equal to the identity matrix. This just means that all shapes drawn after this will be drawn at their "default" size and position. In this example, the quad has width 1 and height 1, and is oriented along the xz plane, with its center at (0, 0, 0).

The subsequent call to m_graphics->scale(20) adds a scale matrix to the current transform. This means that all shapes drawn after this will be scaled by a factor of 20. Finally, the call to m_graphics-

>drawShape("quad") draws a quad aligned with the xz plane. Running the project, part of your screen should now be white. This is your camera looking at a quad.

Setting the Material

This quad is great, but it'd be nice to have pretty-looking quads with different colors or even images on them, so let's see how to do that using Materials.

1. Open the file src/engine/graphics/Material.h. There are a ton of member variables here that relate to different material properties; for example, the color variable specifies the color you want your shape to be. The useLighting variable specifies if you want your shape to take lights into account when being drawn. The textureName variable refers to the name of an image stored in the Graphics object, which will be drawn onto your shape.

2. Let's create a material! Go back to the View::initializeGL() function and add the following code:

   Material myFirstMaterial; myFirstMaterial.color = glm::vec3(0, 1, 0);
   m_graphics->addMaterial("boringGreen", myFirstMaterial);

   The first line Material myFirstMaterial; creates a new default Material object (you can check the Material constructors in Material.h to see what all of the variables will be initialized to by default).

   The second line myFirstMaterial.color = glm::vec3(0, 1, 0); is just setting the color of your material to a bright green (the color variable is interpreted as an RGB triplet).

   The third line m_graphics->addMaterial("boringGreen", myFirstMaterial); stores your material in the graphics object under the name "boringGreen", so that we can recall it later.

3. So let's use our material! Go back to the View::paintGL() function and add the following line before you draw your quad: m_graphics->setMaterial("boringGreen");.

   This line sets the current material to "boringGreen", so that shapes will be drawn using this material.

4. Now let's create a Material that has a texture associated with it. Create a new material mySecondMaterial and add the line

   mySecondMaterial.textureName = "grass"

   Using the previous steps to guide you, store this material in the graphics object and use it to render the quad.

Camera Movement

In most games, we want the camera to be affected by player key inputs. In this guide, we'll use the W, A, S, D keys for moving.

1. In View::keyPressEvent(), add:

   glm::vec3 look = m_camera->getLook();

   glm::vec3 dir = glm::normalize(glm::vec3(look.x, 0, look.z));

   glm::vec3 perp = glm::vec3(dir.z, 0, -dir.x);

   // strafe movement

```
if(event->key() == Qt::Key_W) m_camera->translate(dir);

if(event->key() == Qt::Key_S) m_camera->translate(-dir);

if(event->key() == Qt::Key_A) m_camera->translate(perp);

if(event->key() == Qt::Key_D) m_camera->translate(-perp);
```

This makes it so that whenever a key is pressed, the player is moved forward one unit, backwards one unit, left one unit, or right one unit.

2. The camera should also be affected by mouse movements. Again, we'll use a standard mouse configuration. In View::mouseMoveEvent(), add:

```
m_camera->rotate(-deltaX / 100.f, -deltaY / 100.f);
```

The camera will now adjust its pitch and yaw proportionally to how much the mouse moves. Try running the project – you should be able to move the mouse around and walk around your green quad.

Drawing a second shape

Great! Now we have a beautiful quad and can run around the world. But 3D games usually have more than just a single 2D plane in them, so let's try to draw a *second* shape.

1. In the function View::paintGL(), add this line:

```
m_graphics->drawShape("cylinder"); Build and run your project.
```

But where's the cylinder...

It turns out that it's in the same place and has the same material as our quad. Because the camera is inside the cylinder, it can't see it (a consequence of how OpenGL renders shapes). This probably isn't what you want. Uh-oh!

The reason this happens is because OpenGL (which is what our graphics object uses) is a state machine. This means that unless you explicitly change a setting, such as the current material or transform applied to shapes, OpenGL will continue to use that setting. In other words, OpenGL stays in the same state until you give it a command to change that state. This is great when you want to draw many objects with near identical properties (100 red cubes, for example), but this also means that you need to explicitly reset things you are done using.

2. Try augmenting the above line of code with this:

```
m_graphics->clearTransform(); m_graphics->setDefaultMaterial();
m_graphics->translate(glm::vec3(1.f,1.f,10.f)); m_graphics->scale(5);
m_graphics->drawShape("cylinder");
```

You should end up with a plain cylinder in front of you and slightly to your left. Hooray!

Some Final Words

At this point, you should be prepared to do Warmup 1, but what you have done here will not pass for the first project. For example, you probably should not be doing all (or any) of these things in view.cpp. Again, we recommend playing around with what we have taught you in this guide!