CSCI 1230 Project Structure Guide

Staff

July 12, 2021

1 Introduction

This document describes the parts of the CSCI 1230 stencil that are relevant to each assignment.

2 Files Relevant to Every Assignment

The file `main.cpp` in the root of the CSCI 1230 stencil is the entry point of the program. **You will never need to modify** `main.cpp`.

Navigate to the "Projects" panel of Qt Creator, and scroll down to the "Forms" section. Under the `ui` directory, you will find `mainwindow.ui`. This is an `xml` file used by Qt Creator that defines the panels, sliders, labels, and buttons that make up the CSCI 1230 stencil's user interface. If you click on `ui/mainwindow.ui` in the "Projects" panel, you will be taken to Qt Creator's "Design" view, which allows you to modify the user interface defined by this file, while using a fancy graphical interface! **You will not need to modify** `ui/mainwindow.ui` **in any way for any assignment in this course**. However, you will likely want to create your own user interface for your final project. Feel free to use `ui/mainwindow.ui` as a guide when you create your own user interface! Click the "Edit" button on the leftmost panel of your Qt Creator window to exit the "Design" view.

After compiling the project for the first time, you may have noticed that a file called `ui_mainwindow.h` was generated and placed in the root directory of the project. This file essentially takes the design of the user interface defined in `ui/mainwindow.ui` and translates it into C++ code. **There is no need to ever modify** `ui_mainwindow.h` **because it is overwritten every time** `ui/mainwindow.ui` **is modified**.

Next, take a look at `ui/mainwindow.h` and `ui/mainwindow.cpp`. These files take the user interface defined by `ui/mainwindow.ui` and specify the functionality of each panel, button, and slider in the interface. Most of the methods here are callback functions (called "slots" in Qt-speak) that say "when this specific button is pushed, this is what should happen". You may look at these methods and wonder what UI element (i.e. button, slider) each of these methods corresponds to. These UI element-to-function connections are actually defined in `ui/mainwindow.ui` and then translated into C++ code in `ui_mainwindow.h`. It is also possible to de-

fine such connections outside of `ui_mainwindow.h` and `ui/mainwindow.h`. **You will have to modify `ui/mainwindow.cpp` in the Intersect assignment**.

Look over `ui/Settings.h` and `ui/Settings.cpp`. The `Settings` struct contains the state of the user interface. The `Settings` struct stores information such as whether a feature is turned on or off, the values of parameters set by sliders and text boxes, and what type of filter or brush is currently selected. There is one *global* instance of the `Settings` struct, which is called `settings`. This global variable is initialized by the `MainWindow` class and is accessible to any file that `#include`s the file `ui/Settings.h`. **You will be using the global `settings` variable in all of the assignments to access the state of the user interface**. It's not necessary to query the sliders for their values, because the values will be passed to the static `settings` object every time they are changed.

Finally, the code in `ui/Databinding.h` and `ui/Databinding.cpp` helps connect visual interface elements like sliders or text boxes to the values stored in the global `settings` struct. **You will never have to modify `ui/Databinding.h` or `ui/Databinding.cpp`**

3 Brush

First, let's look at the `ui` directory. As described above, the global `settings` object is relevant here, and you will want to `#include` the file `ui/Settings.h` anywhere you need to access the state of the user interface. For example, you will get the value specified by the "Radius" slider from `settings.brushRadius`.

Next, let's look at the `Canvas2D` class. **You will have to modify this class as part of the Brush assignment**. This class has methods that allow you to interact with the 2D panel of the user interface. **This class extends the `SupportCanvas2D` class, which you will never need to modify**. You will need to fill in the TODOs relevant to the Brush assignment in the `Canvas2D` class.

Finally, look at the classes in the `brush` directory. You will need to fill in all the TODOs in these classes.

Note that the `RGBA` struct in the `lib` directory will be useful for this project.

For more information about the `Settings` struct and `Canvas2D` class, refer to the Brush assignment handout.

4 Shapes

First, it would be good to take a quick look at `scenegraph/Scene.cpp` and `scenegraph/Scene.h`, the base class for `OpenGLScene` (which is also in the same directory). Generally, you do not need to modify the `OpenGLScene` class, but you should be aware that it's the base class for `ShapesScene`, which is where you will render your shapes. **Specifically, you will need to edit the constructor and the `renderGeometry` method in `ShapesScene.cpp`**.

For this project you will have to copy your `VBO` and `VAO` files from Lab 1 into the `gl/datatype` folder. In addition, a good starting point for your shapes is the `OpenGLShape` class from Lab 2. We do not give you starter files for this project, so feel free to create a separate directory for holding

your Shapes files.

The camera with which you view your shapes is the `OrbitingCamera` in the `camera` directory. You do not need to modify this file, but once you complete the camtrans lab, you will understand all of the code in it!

You are also relying on code in the `shaders` directory to view your shapes in this project. The code that displays the wireframe is the shader in the `shaders/wireframe` directory. The shaders in the `shaders/normals` directory display the normal vectors. The actual shape is displayed using the shader defined by `shaders/shader.vert` and `shaders/shader.frag` You do not have to modify these shaders at all.

5 Intersect and Ray

You will need to modify `ui/mainwindow.cpp` in the Intersect assignment. Look for the TODO in that file.

In these assignments, you will fill in the TODOs in the `Canvas2D` that are relevant to the Intersect and Ray assignments.

You will be working in `scenegraph/RayScene.h` and `scenegraph/RayScene.cpp` to implement your raytracer. You will want to create a separate `ray` directory that contains your raytracer code. You will use the `RGBA` struct in the `lib` directory for this project.

6 Camtrans

For the Camtrans lab, you will be working primarily in `camera/CamtransCamera.cpp`. You do not need to modify the `OrbitingCamera` or `QuaternionCamera`.

7 Sceneview

For Sceneview, you will be primarily working in the `scenegraph` directory. First, take a look at `scenegraph/Scene.h` and `scenegraph/Scene.cpp`. This is the base class for all your scenes, and because Ray and Intersect will be reusing some Sceneview code, you should add modifications to the base class `Scene`. Additionally, you will probably need to use code from your Shapes project in `scenegraph/SceneviewScene.cpp`. Also, you should examine `scenegraph/SceneviewScene.h` to get an idea of the general structure of the class and potentially edit it if you want to add `#include`s, new member variables, or methods.

You will depend on code in the `CS123ISceneParser`, `CS123XmlSceneParser`, and `ResourceLoader` classes in the `lib` directory for this project, but you do not need to modify anything there. You will also depend on `lib/CS123SceneData.h`, but you do not need to modify that file.

8 Filter

Back to `Canvas2D`! In this project, you will fill in the TODOs in the `Canvas2D` that are relevant to the filter assignment. You will also need to create a new `filter` directory in your CSCI 1230 project, where you can place all of your filter-specific code. You will use the `RGBA` struct in the

`lib` directory here. You may also find it useful to copy over code from the filter lab to this new `filter` directory.

9 The gl Directory

In the `gl` directory, you will find the `GLDebug` class. This class is one of your tools for debugging shaders and OpenGL calls. Refer to the debugging lab for more information.

The `gl/datatype` directory contains classes relevant to OpenGL. You may notice that there are lots of TODOs in these files! You will fill in these TODOs during the labs. However, the labs are separate Qt projects (that contain identical `VBO`, `VAO`, and `FBO` classes), so you will need to copy your work from the labs into the files in `gl/datatype`.

The `gl/shaders` directory contains classes for interfacing with shaders. You will not have to edit these files.

The `gl/textures` directory contains classes that are relevant to rendering textures in OpenGL. As with the `gl/datatype` directory, there are lots of TODOs in these files. You will fill in these TODOs during the labs, and you will need to copy your work from the labs into the files in `gl/textures`.

Finally, in `gl/util`, there is a class `FullScreenQuad`, which allows you to easily render a full-screen quad. This can be helpful when implementing certain algorithms.

10 The glew Directory

To quote the documentation, GLEW provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform.

11 The resources.qrc File

The `resources.qrc` in the root of CSCI 1230 project defines aliases for files (such as shaders) that can be loaded into the program at runtime using Qt methods. **You do not need to ever modify the `resources.qrc` file**.