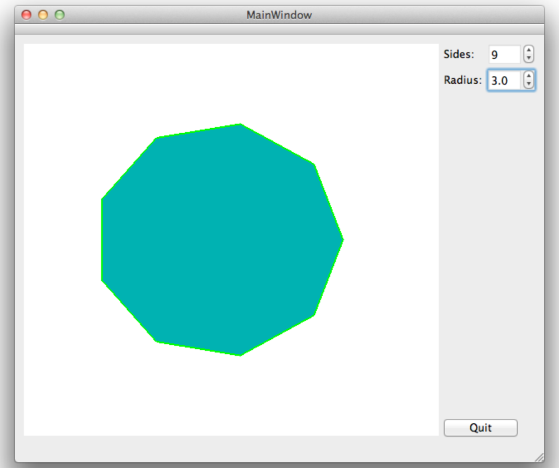
OpenGL and Qt Creator: a Gentle Introduction

Comp175: Introduction to Computer Graphics – Fall 201

September 9th, 2011

1 Introduction

In this lab[1] you will take your first steps into the world of OpenGL and Qt Creator. We will be building a simple polygon drawing system that will allow the user to dynamically control the number of sides as well as the size of the polygon. When you're done, the system will look something like this:



After this lab, you will understand the basics of creating, moving, and coloring objects using OpenGL. You will also gain some familiarity working within the Qt Creator IDE to build a simple, interactive system. While a fully-functional UI for your assignments has been provided as part of the support code, you are free to modify the interface as you see fit. This lab will also introduce you to some of the tools necessary for editing a UI using Qt Creator.

This lab will walk you step-by-step through the building process. Things you need to do will be highlighted in yellow, and we will do our best to go into detail as often as possible. This is intended as a crash course in using this environment, and is not a graded assignment. If at any time you feel like skipping ahead or trying something different, feel free!

[1]This lab is based on Márcio Bueno's tutorial "Tutorial de OpenGL com Qt Creator", available online at:

- Part 1: `http://www.youtube.com/watch?v=XzZH2uC8tWs`
- Part 2: `http://www.youtube.com/watch?v=9A1YwqJ2PqA`
- Part 3: `http://www.youtube.com/watch?v=-dBrGN3Kx6A`

2 Getting Started

Begin by opening the QtCreator IDE, and creating a new project by selecting:

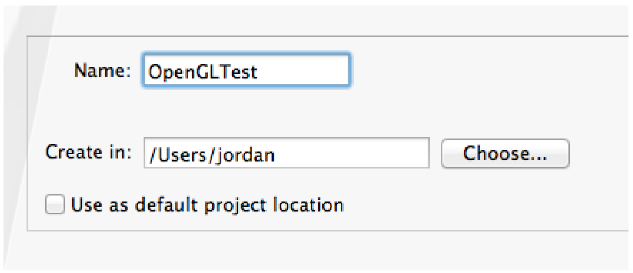
`File` $\rightarrow$ `New File or Project...`

2.1 Specify the project type

We would like to create a new GUI Application for this lab. In the dialog box that opens, select:

`Qt Widget Project` $\rightarrow$ `Qt Gui Application`

In the dialog box that opens, enter the name of your project and the location where you'd like to store it. If you're working on a lab machine, be sure to put this in your home directory. I'll call my project `OpenGLTest`:



Name: OpenGLTest

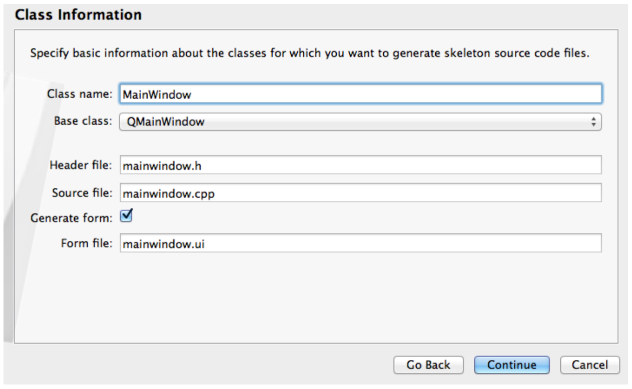
Create in: /Users/jordan

2.2 Specify the target

Qt Creator can be used to build applications for several targets, including both desktop computers and mobile devices. For this course, we only care about building desktop applications, but feel free to play around with other platforms if you're curious. For this project, check the box marked: `Desktop`.

2.3 Set up the `MainWindow` class

Your first class for this project will be called `MainWindow`, which will extend the `QMainWindow` base class. Note that when you fill in the `Class name` field, the header, source, and form file names will be automatically generated:



Class Information

Specify basic information about the classes for which you want to generate skeleton source code files.

Class name:	MainWindow
Base class:	QMainWindow
Header file:	mainwindow.h
Source file:	mainwindow.cpp
Generate form:	✓
Form file:	mainwindow.ui

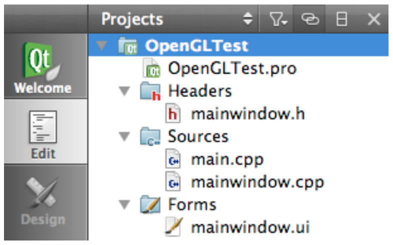
On the next screen, you will see 5 files that will be added to your project:

- `main.cpp`: the standard `main` setup area
- `mainwindow.cpp`: the source file for your `MainWindow` class
- `mainwindow.h`: the header file for your `MainWindow` class
- `mainwindow.ui`: the form file, which we will later edit using Qt Creator's built in WYSIWYG interface editor
- `OpenGLTest.pro`: the Qt project file

Click `Finish`, and you're ready to start editing!

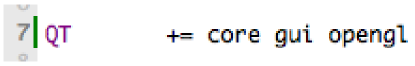
3 Qt Creator

You will now see your project open in the Qt Creator editing window. Your `OpenGLTest` project and all the associated files should appear in the `Projects` window:



3.1 Add the `opengl` module

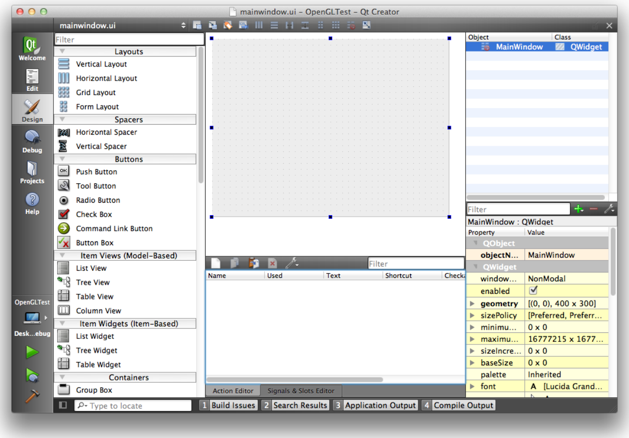
Because we are going to want to use OpenGL to draw the polygons, we need to add the `opengl` module to the project. To do this, double click `OpenGLTest.pro` to open it in the editor and add `opengl` to the list of packages, right after the two packages included already (`core` and `gui`):



```
QT       += core gui opengl
```

3.2 Editing the UI

Now we'll introduce you to Qt Creator's built in WYSIWYG interface editor. To get started, double click `mainwindow.ui`. You'll notice that unlike `.pro`, `.cpp`, and `.h` files, `.ui` files open in the Design window:



Let's start by resizing the `MainWindow` object (the big grey box) until it is around 650 pixels wide and 520 pixels high. You can do this by dragging the lower righthand corner.

On the lefthand side of the window you'll notice a massive list of widgets, all of which are at your disposal for setting up your UI. We'll use these to set up the user controls for the `number of sides` and the `radius` of the polygon to be displayed.

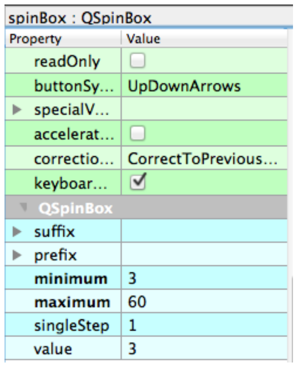
Drag and drop the appropriate widgets onto the UI:

- a `Label` Display Widget marked `Sides`
- a `Spin Box` Input Widget for entering the number of sides as an `int`
- another `Label` Display Widget marked `Radius`
- a `Double Spin Box` Input Widget for entering the radius as a `double`
- and a `Push Button` marked `Quit`

We will also need to add a generic `Container Widget`, which will later be used as a canvas on which OpenGL will draw the polygon. Feel free to play around with the various mechanisms for aligning and arranging these widgets. There are lots of fun springy `Spacer` tools as well as vertical, horizontal, grid, and form `Layouts`. You may also want to play with setting the `Size Constraints` for some of your widgets (like buttons) in conjunction with these layouts.

3.3 Constraining Spin Box Values

Because it wouldn't make sense to draw a polygon with fewer than 3 sides, or with a radius larger than our canvas can reasonably display, we want to limit the range of values that may be entered by the user. To do so, select the spin box you want to limit, and then scroll to the bottom of the `Property | Value window` in the lower righthand corner.



spinBox : QSpinBox

Property	Value
readOnly	☐
buttonSy...	UpDownArrows
▸ specialV...	
accelerat...	☐
correctio...	CorrectToPrevious...
keyboar...	✓
QSpinBox	
▸ suffix	
▸ prefix	
minimum	3
maximum	60
singleStep	1
value	3

There you can set the **minimum** and **maximum** values for each spin box, as well as the number of decimals and the size of each step. We recommend:

- `Sides`: ranging $[3, 60]$
- `Radius`: ranging $[1, 5]$ with 1 decimal place, and each step being 0.1

4 Building a canvas

Before we can use OpenGL commands to draw the polygon, we first need to define a canvas on which to do the drawing. To do this, let's add a new class to the project by selecting:

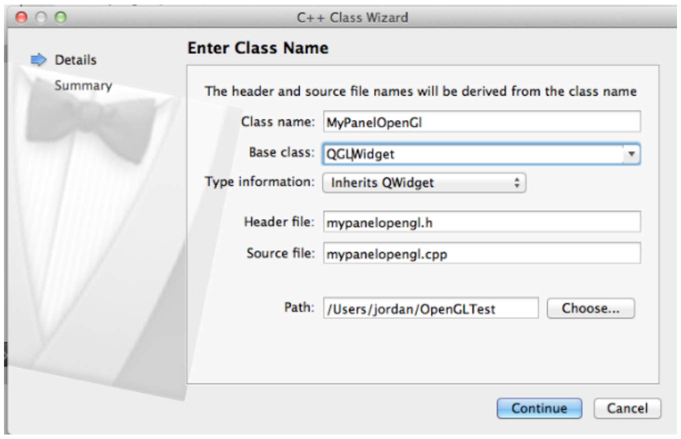
`File` → `New File or Project...`

And in the dialog box that opens, select:

`C++` → `C++ Class`

4.1 Define the `MyPanelOpenGL` Class

In the dialog box that opens, you will define a class called `MyPanelOpenGL` that extends the base class `QGLWidget` (note that this must be typed in manually) and also inherits `QWidget`. Note that again, the header and source files are automatically declared for you:



C++ Class Wizard

Enter Class Name

The header and source file names will be derived from the class name

Class name: MyPanelOpenGl
Base class: QGLWidget
Type information: Inherits QWidget
Header file: mypanelopengl.h
Source file: mypanelopengl.cpp
Path: /Users/jordan/OpenGLTest

As you might have guessed, the `QGLWidget` base class is the class we will extend to be able to use OpenGL within Qt. The three methods we will need to include are:

- `initializeGL()` which will be called to initialize the GL Context
- `resizeGL()` which will be called when the widget is resized
- `paintGL()` which will be called when the OpenGL widget needs to be redrawn

Declare all three as protected methods in the `mypanelopengl.h` header file:

```
protected:
    void initializeGL();
    void resizeGL(int x, int h);
    void paintGL();
```

We'll also need to declare the private variables:

```
private:
    int sides;
    double radius;
```

As well as a pair of public methods:

```
public slots:
    void changeSides(int s);
    void changeRadius(double r);
```

and define all the methods in the `mypanelopengl.cpp` source file:

```
void MyPanelOpenGL::initializeGL(){}
void MyPanelOpenGL::resizeGL(int x, int h){}
void MyPanelOpenGL::paintGL(){}
void MyPanelOpenGL::changeSides(int s){}
void MyPanelOpenGL::changeRadius(double r){}
```

While we're here, we will also `#include <cmath>` at the top of the `mypanelopengl.cpp` source file, and initialize the private variables in the constructor:

```
sides = 3;
radius = 1.0;
```

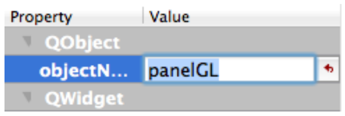
4.2 Promote `Widget` to `MyPanelOpenGL`

Hey, remember that generic `Widget` we added back when we were editing the UI? It's time to transform it into a useful `MyPanelOpenGL` object, because soon we'll be ready to draw!

To promote the generic `Widget` to a useful `MyPanelOpenGL` object:

- return to the `mainwindow.ui` file, which will open in the `Design` editor
- right-click the generic `Widget`
- select `Promote to...`

In the dialog box that opens, enter `MyPanelOpenGL` into the `Promoted class name:` field. Notice how the header file is automatically filled in for you, and click `Add`. With the `MyPanelOpenGL` class selected, click `Promote`. You'll now be able to see that your `Widget` now displays the class `MyPanelOpenGL` in the `Object | Class` window. For clarity, let's rename this object to `panelGL`:



Property	Value
QObject	
objectN...	panelGL
QWidget	

Now, to add some functionality!

5 Enabling the UI

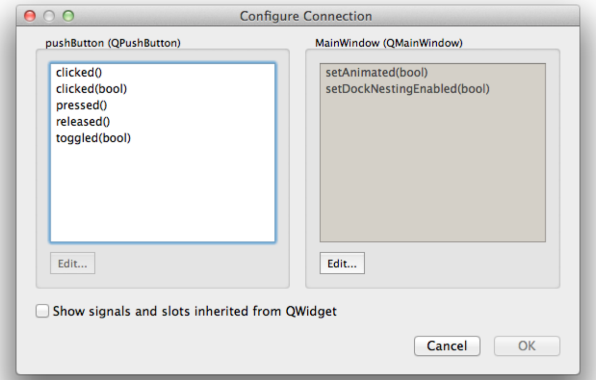
Now it's finally time to make those buttons do something. While still in the `Design` editor, select:

`Edit` → `Edit Signals/Slots...`

You'll notice that every object now has a blue box around it, and you're able to click and drag arrows to indicate interactive relationships. We'll start with an easy one:

5.1 Enabling the `Quit` button

Click and drag on the `Quit` button object, releasing the mouse somewhere that the head of the arrow is touching only the `mainwindow`. When you release the mouse, a dialog box will pop up:



Configure Connection

pushButton (QPushButton)
clicked()
clicked(bool)
pressed()
released()
toggled(bool)

MainWindow (QMainWindow)
setAnimated(bool)
setDockNestingEnabled(bool)

Select the `Show signals and slots inherited from QWidget` checkbox, and configure the connection so that when the button is `clicked()`, the `close()` method is called.

5.2 Enabling the `Spin Boxes`

Dragging an arrow from the `Sides` spin box to the `panelGL` object, configure the connection so that:

when the `Sides` spin box has its `valueChanged(int)`,
the `changeSides(int)` method is called.

Unfortunately, you will notice that the `changeSides(int)` method isn't available from the list of slots, and so you will need to add it by clicking `Edit...` and then adding it to the list.

Using this same method, configure another connection so that:

when the `Radius` spin box has its `valueChanged(double)`,
the `changeRadius(double)` method is called.

6 Filling in Functions

Now all that's left is to fill in the functions! We will leave the coding of the straightforward methods `changeSides(int s)` and `changeRadius(double r)` to you without additional instruction, and move on to the more interesting OpenGL methods. It is important, however, to call `updateGL()` after updating either the sides or the radius to prompt OpenGL to redraw.

6.1 `initializeGL()`

We will call the `initializeGL()` method to set up the GL environment. To fill out this method, we will call:

- `glShadeModel(GL_SMOOTH)`, which lets OpenGL know that we would like to use the `GL_SMOOTH` shading model as opposed to `GL_FLAT`
- `glClearColor(1.0f, 1.0f, 1.0f, 0.0f)` and `glClearDepth(1.0f)`, which zero out the color and depth buffers
- `glEnable(GL_DEPTH_TEST)`, which allows us to do depth comparisons and update the depth buffer and `glDepthFunc(GL_LEQUAL)`, which sets the depth function to the standard less than or equal, and finally
- `glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST)` which is a fancy trick for making things look a little nicer. You can read more about `glHint` online if you're curious.

6.2 paintGL()

We will call the `paintGL()` method to do the actual drawing. To fill out this method, we will first call:

```
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glLoadIdentity();
```

which will clear the current buffers and set the current matrix to the identity. Next, we'll call:

```
glTranslated(5.0, 5.0, 0.0);
```

which produces a translation by (5.0, 5.0, 0.0). Now, we'll set the width and color of the pen:

```
glLineWidth(1);
glColor3f(0, 0.7f, 0.7f);
```

And now we're ready to draw the polygon. To draw the solid polygon, we will use `glBegin()` and `glEnd()` to define a `GL_POLYGON` primitive, and use some basic geometry to define the location of the `glVertex2f` vertices:

```
glBegin(GL_POLYGON);
for (int i = 0; i < sides; i++){
   glVertex2f(radius*cos(i*2*3.14159265/sides),
                   radius*sin(i*2*3.14159265/sides));
}
glEnd();
```

Let's add an outline around the polygon to make it look nice. First, we'll change the width and color of the pen:

```
glLineWidth(2);
glColor3f(0, 1, 0;
```

and then we will repeat the exact same code as above, only this time, instead of a `GL_POLYGON` primitive, we will define a `GL_LINE_LOOP` primitive:

```
glBegin(GL_LINE_LOOP);
for (int i = 0; i < sides; i++){
   glVertex2f(radius*cos(i*2*3.14159265/sides),
                   radius*sin(i*2*3.14159265/sides));
} glEnd();
```

Now all that's left is to take care of resizing the window and its contents to make sure that everything appears in the right place. We'll do this using:

6.3 resizeGL(int width, int height)

Let's start by setting up some variables:

```
double xMin = 0, xMax = 10, yMin = 0, yMax = 10;
```

Then we'll want to let OpenGL know that the window size has changed by calling:

```
glViewport(0,0,(GLint)width, (GLint)height);
```

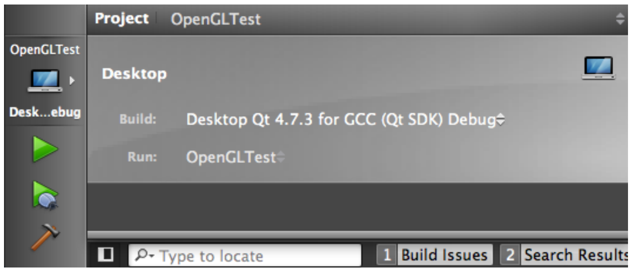
Now we'll want to adjust the viewing region by switching to GL_PROJECTION mode and modifying the min and max values for the size of the region. We'll give you the code for doing this below, and recommend that you check out each of the functions used to get an idea of what it does:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(-1,1,-1,1);

if (width > height){
    height = height?height:1;
    double newWidth = (xMax - xMin) * width / height;
    double difWidth = newWidth - (xMax - xMin);
    xMin = 0.0 + difWidth / 2.0;
    xMax = 10 + difWidth / 2.0;
} else {
    width = width?width:1;
    double newHeight = (yMax - yMin) * width / height;
    double difHeight = newHeight - (yMax - yMin);
    yMin = 0.0 + difHeight / 2.0;
    yMax = 10 + difHeight / 2.0;
}
gluOrtho2D(xMin, xMax, yMin, yMax);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

7 Compile and Test!

And that's it! Congratulations, you've finished coding your first complete OpenGL project using Qt Creator. Now it's time to take it for a spin. You'll notice that by default, the creator is set to build your projects in `Debug` mode:



Build: Desktop Qt 4.7.3 for GCC (Qt SDK) Debug

Run: OpenGLTest

You can build and run your project by pressing the green arrow in the lower lefthand corner. You might also want to switch to `Release` mode, which will suppress any warnings, by selecting it from the drawdown menu. The project will build, and the executable will launch for you to play around with. Viòla!