Qt Toolkit What You Need to Know

Zia Khan

Qt Overview

- Pronounced like “Cute”
- Cross Platform Development Framework
  - Desktop: Mac, Windows, Linux, etc.
  - Mobile: iOS, Android, etc.
- Originally C++, but bindings in Python, Java and Ruby
- LGPL license
  - can use in commercial software w/o releasing your own code
- In active development since 1991

Qt Users

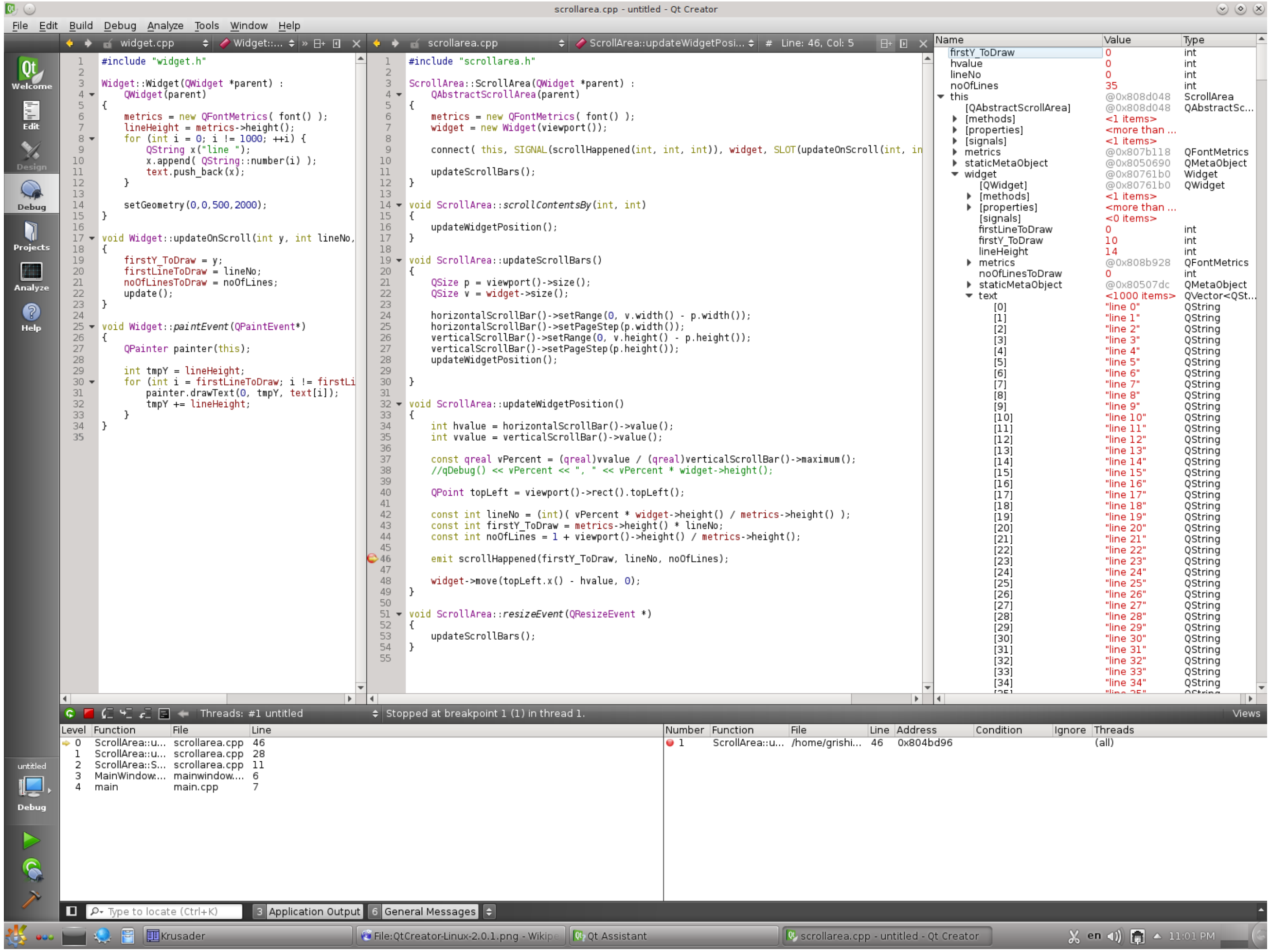
- Companies and govt. agencies
  - European Space Agency, Dreamworks, Lucasfilm,..
  - Your next startup. ;-)
- Software based on Qt
  - Skype Client, Spotify Client, Mathematica, Google Earth, AutoDesk's Maya, BitCoin Core client, Amazon.com's MacOS and Windows Client etc.
  - Tower control system in Germany. Ever fly to Germany? Your plane was guided to the ground using Qt.
  - Your next world changing and/or profit generating application. ;-)

Why are we using Qt in CMSC427?

- Need something to handle windowing and input operations on the applications we develop in class.
- Nice helper classes available for handling some OpenGL operations (e.g. QOpenGlBuffer, QMatrix4x4, QOpenGLShaderProgram, etc.)
- GLUT is commonly used in graphics classes, but the last release 3.7 was in 1998! YES 1998!!!!!!
- Some other alternatives like GLFW and FreeGLUT, but Qt has these nice helper classes.
- I wanted to introduce you guys to a toolkit that will help you build the next awesome world changing app.

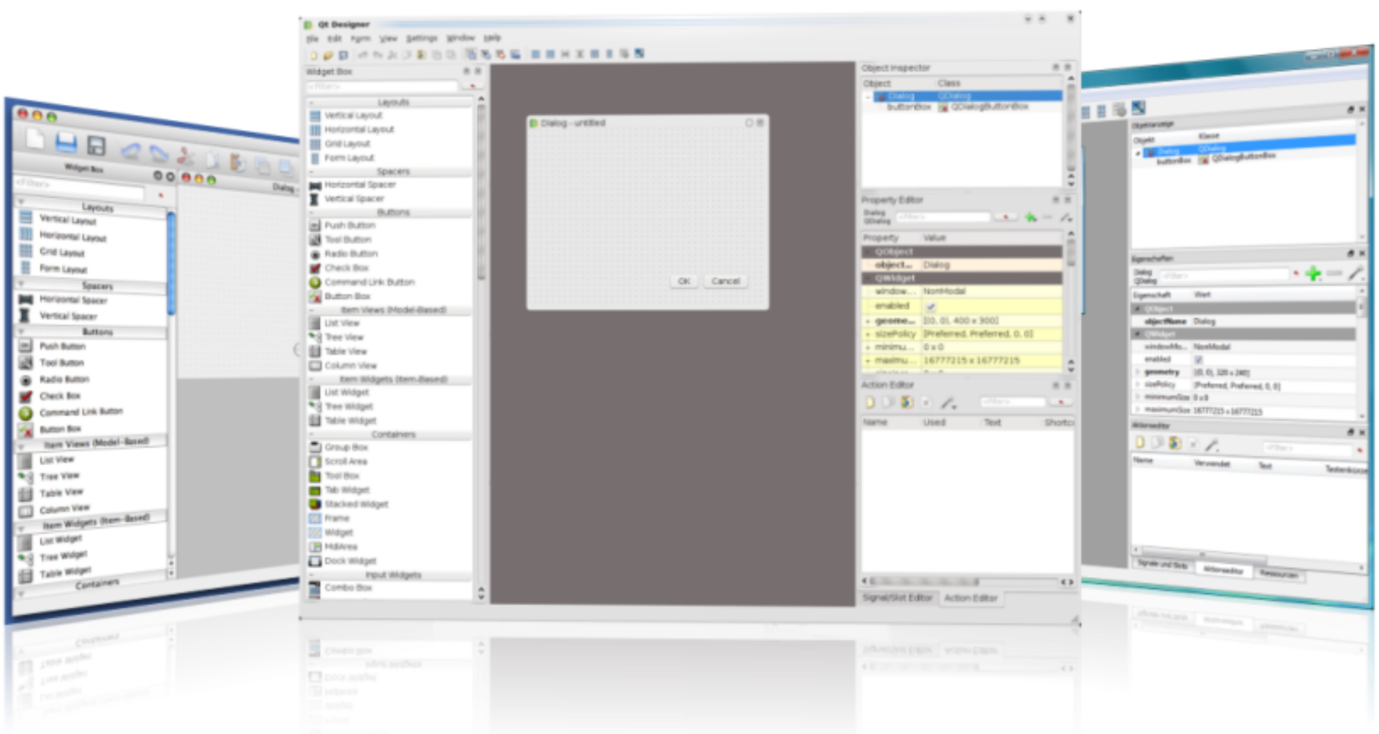
Qt Creator

- Eclipse-like IDE (but much simpler/easier to use)



Qt Designer – UI Widget Editor

- Can be used to created custom dialog boxes, windows, input boxes, add menus etc.



What are we going to cover today?

- Creating a simple command line application using Qt's build tools.
- Useful classes in the Qt library.
- Using some of Qt's classes in the command line application.

- Creating a minimal Shader Based OpenGL application.
- Using the Qt resource system.
- Responding to keyboard and mouse input.

`cmsc427.pro` (Qt Project File)

For a simple console application (Mac, Win, and Linux)
with access to GUI classes (e.g. QImage)

```
TEMPLATE = app
CONFIG += console warn_on release embed_manifest_exe
CONFIG -= app_bundle
QT += gui
SOURCES += cmsc427.cpp
HEADERS += cmsc427.hpp
QMAKE_CXXFLAGS += -I/usr/local/include
windows {
   QMAKE_CFLAGS += "/D_HAS_ITERATOR_DEBUGGING=0 /D_SECURE_SCL=0"
   QMAKE_CXXFLAGS += "/D_HAS_ITERATOR_DEBUGGING=0 /D_SECURE_SCL=0"
}
```

You will probably only change these if you add and remove header files.

+= adds an option/file and -= removes an option or file
Windows specific parameters like `embed_manifest_exe` assure meta data is added so standalone EXE runs.
Additional parameters remove bounds checking for STL classes for speed.
-= app_bundle, prevents creation of a .app folder, but you can

cmsc427.[h,c]pp Source Code

```cpp
#ifndef __CMSC427_HPP__
#define __CMSC427_HPP__

#include <QtCore>
#include <QtGui>

class Test {
public:
  Test(bool do_sort);
  void demo();
protected:
  bool sorted;
  QVector<int> num;
};


#endif
```

QtCore and QtGui provide access to Qt classes.

```cpp
#include <iostream>

#include "cmsc427.hpp"

using namespace std;

Test::Test(bool do_sort) {sorted = do_sort;}

void Test::demo() {
  num.append(4);
  num.append(2);
  num.append(7);
  if(sorted) qSort(num.begin(), num.end());
  for(int i = 0; i < num.size(); i++)
    cout << num[i] << " ";
}

// Application start point.
int main(int argc, char *argv[]) {
  Test sort_test(true), reg_test(false);
  sort_test.demo(); reg_test.demo();
  return 0;
}
```

Qt Modules

- Qt has several modules, Core, GUI, Multimedia, Network, QML, SQL, Widgets, etc.
- Can add QT += [module name] to .pro file to link these modules.
- In this class we'll use, mostly, `QT += gui widgets` in our .pro file. Core is included by default.
- Then in your header files add
  - #include <QtCore>
  - #include <QtGui>
  - #include <QtWidgets>

Useful stuff in Core: Qt Containers

Class	Summary
QList<T>	This is by far the most commonly used container class. It stores a list of values of a given type (T) that can be accessed by index. Internally, the QList is implemented using an array, ensuring that index-based access is very fast. Items can be added at either end of the list using QList::append() and QList::prepend(), or they can be inserted in the middle using QList::insert(). More than any other container class, QList is highly optimized to expand to as little code as possible in the executable. QStringList inherits from QList<QString>.
QLinkedList<T>	This is similar to QList, except that it uses iterators rather than integer indexes to access items. It also provides better performance than QList when inserting in the middle of a huge list, and it has nicer iterator semantics. (Iterators pointing to an item in a QLinkedList remain valid as long as the item exists, whereas iterators to a QList can become invalid after any insertion or removal.)
QVector<T>	This stores an array of values of a given type at adjacent positions in memory. Inserting at the front or in the middle of a vector can be quite slow, because it can lead to large numbers of items having to be moved by one position in memory.
QStack<T>	This is a convenience subclass of QVector that provides "last in, first out" (LIFO) semantics. It adds the following functions to those already present in QVector: push(), pop(), and top().
QQueue<T>	This is a convenience subclass of QList that provides "first in, first out" (FIFO) semantics. It adds the following functions to those already present in QList: enqueue(), dequeue(), and head().
QSet<T>	This provides a single-valued mathematical set with fast lookups.
QMap<Key, T>	This provides a dictionary (associative array) that maps keys of type Key to values of type T. Normally each key is associated with a single value. QMap stores its data in Key order; if order doesn't matter QHash is a faster alternative.
QMultiMap<Key, T>	This is a convenience subclass of QMap that provides a nice interface for multi-valued maps, i.e. maps where one key can be associated with multiple values.
QHash<Key, T>	This has almost the same API as QMap, but provides significantly faster lookups. QHash stores its data in an arbitrary order.
QMultiHash<Key, T>	This is a convenience subclass of QHash that provides a nice interface for multi-valued hashes.

http://qt-project.org/doc/qt-5/containers.html

QString and QStringList

http://qt-project.org/doc/qt-5/qstring.html

```
QString str1 = "ABCD";
char x = str[1].toLatin1();
// x is 'B' in ASCII format
string cppstr = str.toStdString();
// cppstr is now a C++ std::string
```

Note: QString's use Unicode characters.

```
QString str1 = "1234.56";
str1.toFloat(); // returns 1234.56
bool ok;
QString str2 = "R2D2";
str2.toFloat(&ok);
// returns 0.0, sets ok to false
```

Really handy functions to convert numbers as strings e.g. atof in C.

```
QString str = "a,,b,c";
QStringList list1 = str.split(",");
// list1: [ "a", "", "b", "c" ]
QStringList list2 = str.split(",", QString::SkipEmptyParts);
// list2: [ "a", "b", "c" ]
```

Tokenize a string. Way better than strtok in C.

QFile and QTextStream

http://qt-project.org/doc/qt-5/QFile.html

```
QFile file("in.txt");
if (!file.open(QIODevice::ReadOnly | QIODevice::Text))
  return;

QTextStream in(&file);
while (!in.atEnd()) {
  QString line = in.readLine();
  process_line(line);
 }
```

Reading one line of a text file in at a time.

QImage

http://qt-project.org/doc/qt-5/QImage.html

```
QImage output(image_width, image_height, QImage::Format_RGB32);
    for(int y = 0; y < image_height; y++) {
        for(int x = 0; x < image_width; x++) {
         float red = 1.0, green = 1.0, blue = 1.0;
         QRgb value = qRgb(red*255.0, green*255.0, blue*255.0) ;
        output.setPixel(x, y, value);
   }
}
output.save("test.png");
```

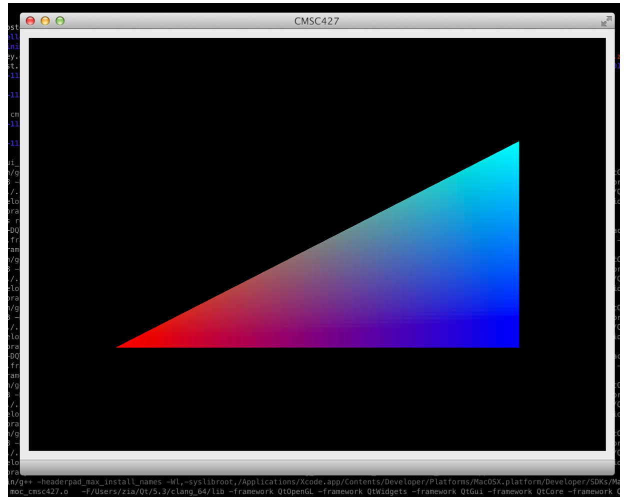
Extension is used to determine the format of the output image.

QImage::Format sets image red,green, blue, and alpha for each pixel with alpha set to 255 (no transparency).

Class Activity

- In your current written assignment, you’ve been asked to write a small Qt program.
- I’d like to get started on this in class.
- You may work alone or in groups of 2 or 3.
- Please note on your written assignment who you worked with.

Minimal 2D OpenGL Application



OpenGL = rich and complex cross platform API used for hardware accelerated 2D and 3D graphics on the desktop and on mobile devices

Shader Based OpenGL = programmable API, where you specify little programs called shaders to customize different parts of the drawing.

Shaders are written in GLSL, a C-like language with useful vector operations.

The catch: lot's of painful overhead to use this API.

OpenGL Context = state information of the
  Mental model: OpenGL API as a state machine.

Source Files

GLview.cpp
GLview.hpp
cmsc427.cpp
cmsc427.hpp
cmsc427.pro
cmsc427.ui
resources.qrc
simple.fsh
simple.vsh

Shader Based OpenGL code here

main(), QMainWindow, and keyboard capture

Qt Resource system (more in a few slides)

.fsh = fragment shader
.vsh = vertex shader

cmsc427.cpp: main()

```
// Application start point.
int main(int argc, char *argv[]) {
  // Launch main application.
  QApplication app(argc, argv);

  // Set OpenGL version and parameters
  QGLFormat fmt;
  fmt.setProfile(QGLFormat::CoreProfile);
  fmt.setVersion(3,3);
  fmt.setSampleBuffers(true);
  QGLFormat::setDefaultFormat(fmt);

  // Show main window
  CMSC427Win main_win; main_win.show();
  return app.exec();
}
```

Set OpenGL version and features.

initializeGL() and resizeGL()

Initialize OpenGL, set background color and compile your shaders

```
void GLview::initializeGL() {
  vao.create(); vao.bind(); // need this to initialize GL state.

  // Set the clear color to black
  glClearColor( 0.0f, 0.0f, 0.0f, 1.0f );

  // Prepare a complete shader program...
  if ( !prepareShaderProgram( ":/simple.vsh", ":/simple.fsh" ) ) return;

  ...
}
```

Called right after initializeGL(), sets viewport (see on the board).

```
void GLview::resizeGL( int w, int h ) {
  // Set the viewport to window dimensions
  glViewport( 0, 0, w, qMax( h, 1 ) );
}
```

An Aside: Qt's Resource System for your Shaders

```
TEMPLATE = app
CONFIG += qt warn_on release embed_manifest_exe
CONFIG -= app_bundle
QT += gui opengl xml widgets
FORMS += cmsc427.ui
SOURCES += GLview.cpp cmsc427.cpp
HEADERS += GLview.hpp cmsc427.hpp
QMAKE_CXXFLAGS += -I/usr/local/include
win32 {
   QMAKE_CFLAGS += "/D_HAS_ITERATOR_DEBUGGING=0 /D_SECURE_SCL=0"
   QMAKE_CXXFLAGS += "/D_HAS_ITERATOR_DEBUGGING=0 /D_SECURE_SCL=0"
}
RESOURCES += resources.qrc
OTHER_FILES += simple.frag simple.vert
```

The resource system allows you to keep your shaders in separate files, but also "compile in" the shader code into your binary..

An Aside, resources.qrc – XML file list

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
    <file>simple.fsh</file>
    <file>simple.vsh</file>
</qresource>
</RCC>
```

```
void GLview::initializeGL() {
  ...
  // Prepare a complete shader program...
  prepareShaderProgram( ":/simple.vsh", ":/simple.fsh" )
  ...
}
```

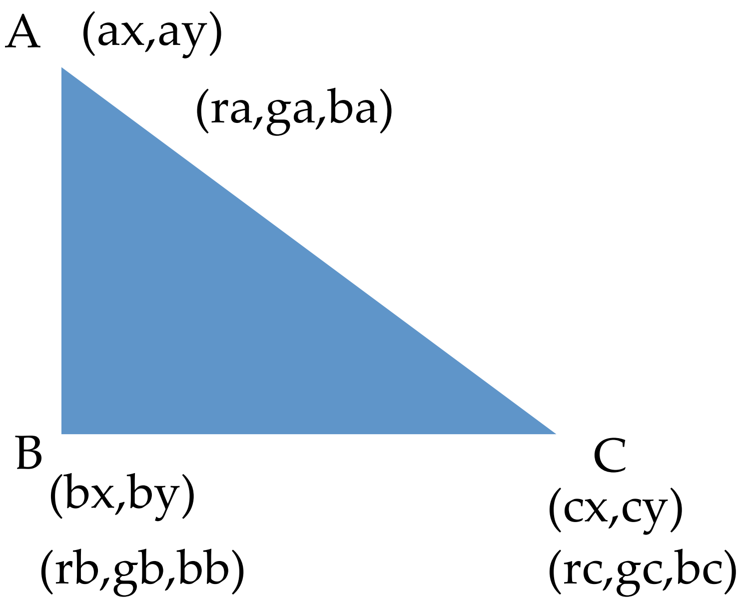
In your code prepend `:/` to tell QFile to get these from the resource system.

Setting up your triangle(s) vertices and vertex attributes.

We will focus almost exclusively on triangles throughout this class.

Vertex buffer (floats): ax, ay, bx, by, cx, cy

Attribute buffer (floats): ra, ga, ba, rb, gb, bb, rc, gc, bc



As we'll see later, it's best to specify your vertices and attributes in the counter clockwise direction.

initializeGL(): Setting up your vertex and attribute buffers

```
class GLview : public QGLWidget {
  …
  QOpenGLBuffer vertexBuffer;
  QOpenGLBuffer colorBuffer;
  …
};
```

```
void GLview::initializeGL() {
  ..
  // Set up vertex and attribute buffers
  int nVert = 3;
  float sqVerts[6] = {
    -.5, -.5,  .5,  .5, .5,  -.5,
  };
  vertexBuffer.create();
  vertexBuffer.setUsagePattern( QOpenGLBuffer::DynamicDraw );
  vertexBuffer.bind();
  vertexBuffer.allocate( sqVerts, nVert * 2 * sizeof( float ) );

  float sqCol[9] =  {
    1, 0, 0,   0, 1, 0,   0, 0, 1,
  };
  colorBuffer.create();
  colorBuffer.setUsagePattern( QOpenGLBuffer::DynamicDraw );
  colorBuffer.bind();
  colorBuffer.allocate( sqCol, nVert * 3 * sizeof( float ) );
  …
}
```

Order of calls to QOpenGLBuffer is important.

bind() , tells the OpenGL context to focus on this current API object
Direct State Access = future extension to OpenGL where this bind() stuff will go away.

initializeGL(): Telling your shaders about the the vertex and attribute buffers

```
shader.bind();

vertexBuffer.bind();
shader.setAttributeBuffer( "aVertex", GL_FLOAT, 0, 2 );
shader.enableAttributeArray( "aVertex" );

colorBuffer.bind();
shader.setAttributeBuffer( "aColor", GL_FLOAT, 0, 3 );
shader.enableAttributeArray( "aColor" );
```

void **QOpenGLShaderProgram**::setAttributeBuffer
(const char * ***name***, GLenum ***type***, int ***offset***, int ***tupleSize*****)**

Assign input variables to buffers of vertices and attributes you created.

Simple Vertex Shader

vec2 = 2d vector
vec3 = 3d vector
vec4 = 4d vector

```
#version 330

uniform float uVertexScale;

in vec2 aVertex;
in vec3 aColor;

out vec3 vColor;

void main( void )
{
  gl_Position =
    vec4(aVertex.x * uVertexScale, aVertex.y, 0, 1);
  vColor = aColor;
}
```

Uniform variable. Value shared between all shaders.

Input buffers.

Output buffer.

Special "built in" variable with new vertex position.

This little application will run independently on each vertex and attribute.

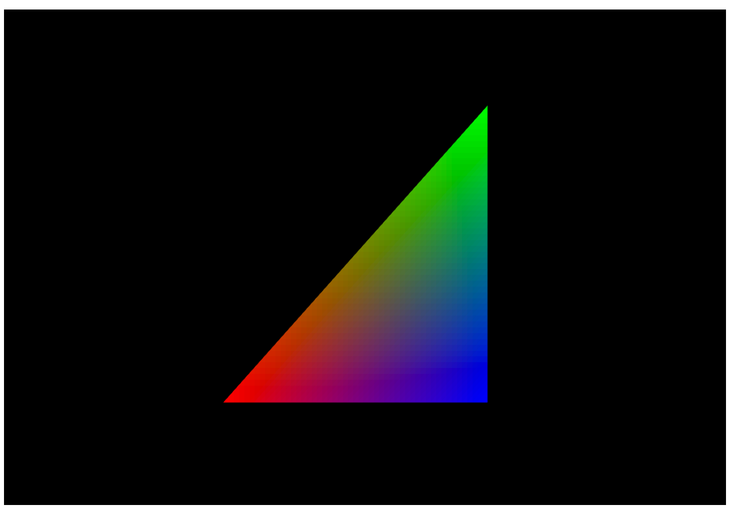
Simple Fragment Shader

simple.fsh – fragment shader, runs on each pixel independently in each triangle

```
#version 330

uniform float uVertexScale;

in vec3 vColor;

layout(location = 0, index = 0) out vec4 fragColor;

void main( void ) {
  fragColor = vec4(vColor.x, vColor.y, vColor.z, 1);
}
```

Shared with vertex shader.

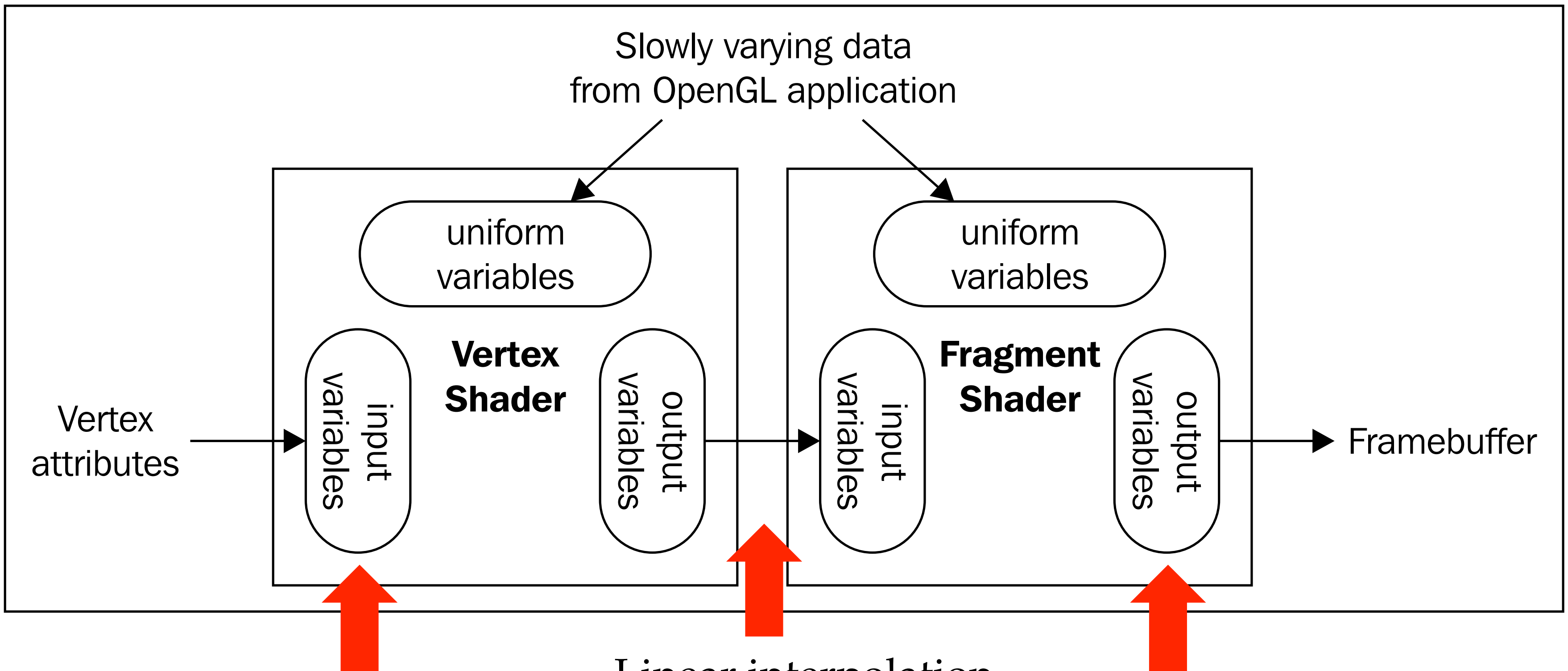
Output color for pixel.
layout stuff is essential.



```
in vec3 vColor;
```

Input variable for fragment shader. This variable is linearly interpolated between each vertex of the drawn triangle at each pixel.

Uniform, input, and output variables



Linear interpolation

Your input vertex
and attribute
buffers.

```
layout(location = 0, index = 0)
out vec4 fragColor
```

Adapted from OpenGL 4.0 Shading Language Cookbook

Running your shaders

```
void GLview::paintGL() {
    // Clear the buffer with the current clearing color
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    shader.bind();
    shader.setUniformValue("uVertexScale", (float)g_objScale);
    // Draw stuff and run shaders
    glDrawArrays( GL_TRIANGLES, 0, 3 );
}
```

Number of vertex attribute tuples.

Set your uniform variable (vertical scale of triangle).

Animating the Triangle via Uniform Variable

```
GLview::GLview(QWidget *parent)  : QGLWidget(parent) {
  …
  QTimer *timer = new QTimer(this);
  connect(timer, SIGNAL(timeout()), this, SLOT(timeout()));
  timer->setTimerType(Qt::PreciseTimer);
  timer->start(17);
}
```

Creates a callback to timeout(), ever 17 milliseconds.

```
void GLview::timeout(){
  if(mousePressed == false) {
    seed += 0.05;
    g_objScale = 0.5 * qSin(seed) + 1;
    updateGL();
  }
}
```

In timeout the object scale, which is a uniform variable that gets updated in paintGL() is modified. updateGL() repaints the OpenGL window.

Handling Mouse Press Events

```
class GLview : public QGLWidget {
  ...

  void mousePressEvent(QMouseEvent *event);
  void mouseReleaseEvent(QMouseEvent *event);
  void mouseMoveEvent(QMouseEvent *event);
  ...
};
```

http://qt-project.org/doc/qt-5/qmouseevent.html

```
void GLview::mouseMoveEvent(QMouseEvent *event) {
  const int newx = event->x();
  const int newy = height() - event->y() - 1;
  if (g_leftClicked) {
    float deltax = (newx - g_leftClickX) * 0.02;
    g_objScale += deltax;
    g_leftClickX = newx;
    g_leftClickY = newy;
  }
  event->accept();
  updateGL();
}
```

Use mouse position to update triangle.

Key press to update triangle

```
void GLview::keyPressGL(QKeyEvent* e) {
  switch ( e->key() ) {
  case Qt::Key_Up:
    updateGeometry();
    break;
  }
}
```

http://qt-project.org/doc/qt-5/qkeyevent.html

```
void GLview::updateGeometry() {
  float updated_vert[2] = {1,-1};
  vertexBuffer.bind();
  vertexBuffer.write(2*2*sizeof(float),
      updated_vert, sizeof(float) * 2);
  updateGL();
}
```

Bind and write to buffer of floats.

Lot's more to Qt

- Please use CMSC427 as a chance to explore. http://qt-project.org
- We'll learn more about Shader-Based OpenGL in coming lectures.