State of the Art OpenGL and Qt

Dr. Roland Krause
Integrated Computer Solutions

©2015 Integrated Computer Solutions, Inc. All Rights Reserved

Abstract

OpenGL is a powerful, low level graphics toolkit with a steep learning curve that allows access to accelerated GPU hardware. Using OpenGL developers achieve high-fidelity, animated graphics ubiquitous in games, screen productions and scientific software. Due to OpenGL's native C-language API large scale, professional software development endeavors wanting to utilize the advantages of direct and accelerated graphics quickly become expensive and hard to maintain.

This presentation gives a comprehensive overview of the many aspects of OpenGL development where Qt provides advanced interfaces that let the developer focus on the tasks at hand instead of dealing with repetitive and error-prone, platform dependent issues. From handling of Window related tasks, providing type-safe data-structures for Vertex Array Objects to managing Shader Programs, dealing with Textures, Frame Buffer Objects all the way to support for Debugging and Profiling of OpenGL Applications we show solutions to the most common issues any OpenGL program has to address.

We are able to demonstrate why Qt is the best C++ framework for development of modern OpenGL based graphics applications.

ICS

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

Modern OpenGL with Qt

- OpenGL is an Application Programming Interface (API) for Rendering Graphics in 2D and 3D
  - Widely used in CAD, Virtual Reality, Scientific Visualization, Information Visualization, Flight Simulation, and Video Games.
  - Over 500 commands in Version 4.3
  - Managed by the non-profit technology consortium Khronos Group.
- Designed as a Streamlined, Hardware Independent Interface
  - To be efficient usually implemented on graphics hardware (GPU)
  - Independent of Operating System or Windowing System, Cross-platform
  - No functions for windowing tasks, managing state, user input, 3D models or reading image files

That's where Qt comes into play!

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

Steps to Render a Scene with OpenGL

1. Create OpenGL Context and Window
2. Create and Manage the OpenGL Scene
    - Create Geometry
        - Store Geometry in Vertex Buffer Objects (VBOs)
    - Define, Compile and Link Shaders into a Shader Program
    - Configure the Rendering Pipeline
    - Set Attribute Arrays, Uniforms, Textures, etc…
3. Render the Scene using OpenGL Primitives

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

Create OpenGL Context and Window

- Since Qt 5 the class QWindow represents a window in the underlying windowing system
  - In Qt 4, QWidget had both Window and Widget functionality
  - In Qt 5, QWindow is part of the gui module, QWidget is in a separate and now optional module (widgets)
  - Applications will typically use QWidget or QQuickView to create user interfaces
- It is possible to render directly to a QWindow with a QOpenGLContext
- QWindow can be embedded in a QWidget

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

Qt's OpenGL Support - QWindow

- Provides OpenGL and OpenGL ES Integration
  - QWindow supports rendering using (Desktop) OpenGL and OpenGL ES, depending on platform support
- Derive class from QWindow
  - In Constructor set QWindow's surface type:

    void QWindow::setSurfaceType(SurfaceType surfaceType)
  - Choose Format Attributes using QSurfaceFormat
    - OpenGL version and profile, sample buffer size, depth buffer size, etc..
  - Create a QOpenGLContext to manage the native OpenGL context

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

Qt's OpenGL Support - QWindow

- Create by calling QWindow::create()
- Implement a set of necessary functions
  - By convention: `initializeGL(), updateGL(), resizeGL()`
  - Needed to initialize and render OpenGL content
- Embed QWindow in a widget using `QWidget::createWindowContainer(...)` returns a QWidget with the QWindow embedded inside it
- Additionally Qt has QOpenGLPaintDevice
  - Enables use of OpenGL accelerated QPainter rendering

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

OpenGL Context

- State machine that stores all data related to the OpenGL rendering state
  - Most OpenGL functions set or retrieve some state
  - Creating a window and an OpenGL context is *not* part of the OpenGL specification
- QOpenGLContext represents a native OpenGL context
  - Enables OpenGL rendering on a QSurface.
  - Context allow to share resources with other contexts with setShareContext()

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

QOpenGLContext

To create:

- Create a QSurfaceFormat object
  - Set desired OpenGL version and profile
- Create the QOpenGLContext
  - Set the context format using the QSurfaceFormat object
- Finally call QOpenGLContext::create()
  - Use return value or isValid() to check if the context was successfully initialized
- Before using any OpenGL QOpenGLContext must be made current against a surface
  - `QOpenGLContext::makeCurrent(QSurface*)`

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

QOpenGLContext

- When OpenGL rendering is done
  - Call swapBuffers() to swap the front and back buffers of the surface at the end of the update function
  - Newly rendered content becomes visible
  - QOpenGLContext requires to call makeCurrent() again before starting rendering a new frame, after calling swapBuffers()
- Use QOpenGLContext::format() to retrieve info on the context
  - Returns a QSurfaceFormat
  - OpenGL version, profile, etc...

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

Qt's OpenGL Support

- QOpenGLFunctions are convenience classes
- Simplify writing of OpenGL code
- Hide complexities of extension handling
- Hide differences between OpenGL ES 2 and desktop OpenGL
  - Allow the use of OpenGL ES 2 functions on Desktop OpenGL
  - No need to manually resolve OpenGL function pointers
  - Allowing cross-platform development of applications targeting mobile or embedded devices

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

QAbstractOpenGLFunctions

- Family of classes that expose all functions for a given OpenGL version and profile
  - OpenGL implementations on different platforms are able to link to a variable number of OpenGL functions depending upon the OpenGL ABI on that platform
  - On many platforms most functions must be resolved at runtime, Options are:
    - Work with raw function pointers: `QOpenGLContext::getProcAddress()`
    - Use QOpenGLFunctions and only expose those functions common to OpenGL ES 2 and desktop OpenGL

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

QAbstractOpenGLFunctions (cont.)

- Provides better support for newer versions of OpenGL (especially 3.0 and higher)
- Ease development of desktop applications relying on modern, desktop-only OpenGL features
- QOpenGLFunctions_X_Y_PROFILE
  - Core and Compatibility Profiles
  - Expose every core OpenGL function by way of a corresponding member function
  - Class for every valid combination of OpenGL version and profile following the naming convention:

```
QOpenGLFunctions_<MAJOR VERSION>_<MINOR VERSION>[_PROFILE]
```

www.ics.com

ICS

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

Using QOpenGLFunctions_X_Y_PROFILE

- Ensure QOpenGLContext is current before using it
- Call `QOpenGLFunctions_X_Y_PROFILE::initializeOpenGLFunctions()` once before using it to resolve function pointers

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

Qt's OpenGL Support

- Classes that wrap native OpenGL Resources
  - QOpenGLBuffer, QOpenGLFramebufferObject
    QOpenGLShaderProgram, OpenGLTexture, QOpenGLVertexArrayObject
- Qt GUI Module Contains
  - QMatrix4x4, QVector4D and QQuaternion
  - Support common mathematical operations for 3D graphics
- Miscellaneous
  - Debugging, QOpenGLDebugLogger
  - Timing, QOpenGLTimeMonitor, QOpenGLTimerQuery

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

Qt 5 and OpenGL Classes

QOpenGLContext
QOpenGLContextGroup
QOpenGLVersionProfile
QOpenGLFunctions*
QOpenGLVertexArrayObject
QOpenGLBuffer
QOpenGLShader
QOpenGLShaderProgram
QOpenGLFrameBufferObject
QOpenGLFrameBufferObjectFormat

QOpenGLDebugLogger
QOpenGLDebugMessage
QOpenGLTimeMonitor
QOpenGLTimerQuery
QOpenGLPaintDevice
QOpenGLTexture (Qt 5.3)
QOpenGLWidget(Qt 5.4)
QOpenGLWindow(Qt 5.4)

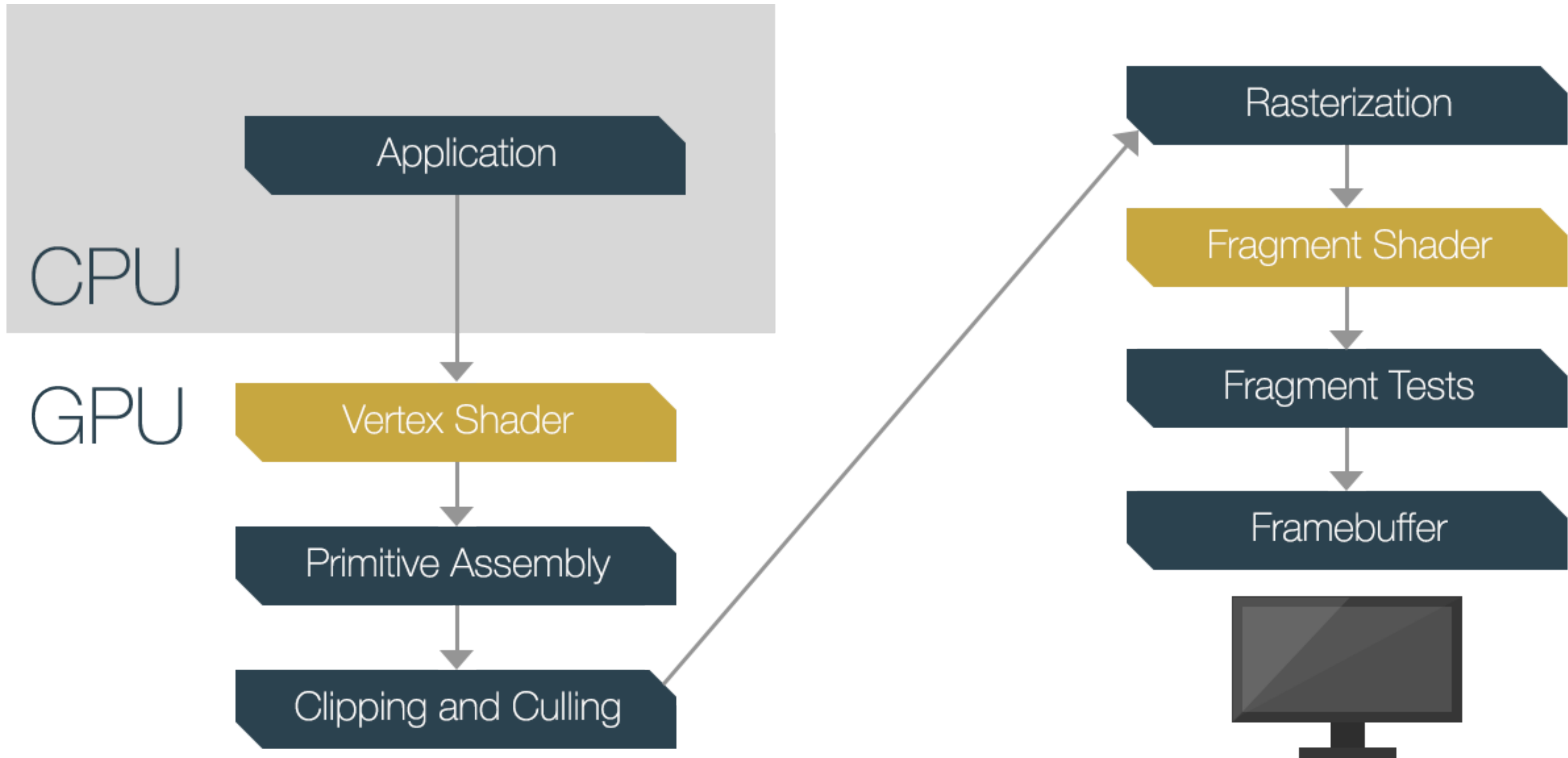
© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

What is a Shader?

- Small program that runs on the GPU
- Ran as part of the OpenGL pipeline
- Programmable
- Coded in GLSL (OpenGL Shading Language)
- Makes rendering infinitely flexible

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

Simplified Pipeline



© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

Shaders

- Two kinds of shaders:
  - Shaders that deal with Vertices, i.E:

    Vertex, Tessellation and Geometry shaders determine *where on the screen* a primitive is.
  - Fragment Shader uses that information to determine *what color* that fragment will be.
- Must have a version identifier at top of file
- Must have a main() function
- Each shader is compiled, then they are linked together to make a shader program
- Input/output interfaces must match, and are checked at link time

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

Shader Basics

- Vertex Shader
  - Executed once for every vertex
  - Input: Content of Vertex Buffer Arrays and Uniforms
  - Output: Vertex position
- Fragment Shader
  - Executed once for every fragment
  - Input: Result of Rasterization after Vertex Shader
  - Output: Candidate pixel color (aka Fragment)

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

Prepare Shaders

- Preparing Shaders
  - Compile vertex shader
  - Compile fragment shader
  - Configure attribute locations before linking
  - Link both shaders into a shader program
- Preparing shaders with Qt vs pure OpenGL
  - Much less code
  - Less error prone

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

Example: Use Qt to Create Shader Program

```
void OpenGLUtilities::createShaderProgram() {
    const GLubyte * _v=glGetString(GL_SHADING_LANGUAGE_VERSION);
    QByteArray _vstr;  _vstr.append(_v[0]); _vstr.append(_v[2]); _vstr.append(_v[3]);
    int _vnumber=_vstr.toInt();
    QByteArray version = "#version " + _vstr;
    bool isOpenGLES=(QOpenGLContext::openGLModuleType() == QOpenGLContext::LibGLES);
    if (_vnumber < 300) {
        version = isOpenGLES ? "#version 100" : "#version 120";
    }
    version += "\n";
    QFile vtFile(":/vertex.vsh");    vtFile.open((QIODevice::ReadOnly | QIODevice::Text));
    QFile fsFile (":/fragment.fsh"); fsFile.open((QIODevice::ReadOnly | QIODevice::Text));
    if (!m_shaderProgram.addShaderFromSourceCode(QOpenGLShader::Vertex,version+vtFile.readAll())){
        qDebug() << "Error in vertex shader:" << m_shaderProgram.log();  exit(1);
    }
    if (!m_shaderProgram.addShaderFromSourceCode(QOpenGLShader::Fragment,version+fsFile.readAll()))
    {
        qDebug() << "Error in fragment shader:" << m_shaderProgram.log();  exit(1);
    }
    if (_vnumber < 300) {
        m_shaderProgram.bindAttributeLocation("vertexPosition", 0);
        m_shaderProgram.bindAttributeLocation("vertexNormal", 1);
    }
    if ( !m_shaderProgram.link() ) {
        qDebug() << "Error linking shader program:" << m_shaderProgram.log(); exit(1);
    }
}
```


© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

Creating a VBO

- Create buffer object
- Bind the buffer, making it the active buffer
- Copy the data to the buffer

```
// Triangle vertices
float vertices[] = {
    -1.0f, -1.0f, 0.0f,
     1.0f, -1.0f, 0.0f,
     0.0f,  0.6f, 0.0f
};
// Create a static buffer for vertex data
m_vertexBuffer.create();
// Set usage pattern to Static Draw, (the data won't change)
m_vertexBuffer.setUsagePattern( QOpenGLBuffer::StaticDraw );
// Bind the buffer to the current OpenGL context
m_vertexBuffer.bind();
// Copy the data to the buffer
m_vertexBuffer.allocate( vertices, 3 * 3 * sizeof( float ) );
```

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

Vertex Attributes

- We have the data (in VBOs)
- We have the shaders compiled
- How do we map the data to the shader attributes?

After compiling shaders before linking:

```
m_shaderProgram.bindAttributeLocation("vertexPosition", 0);
```

This assigns the attribute *vertexPosition* the first location (0)

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

Vertex Attribute Arrays

- VBO data is mapped to shader attribute locations

```
m_shaderProgram.bind();
m_vertexBuffer.bind();
int vertexLocation = m_shaderProgram.attributeLocation("vertexPosition");
m_shaderProgram.setAttributeBuffer(
                     vertexLocation, // layout location
                     GL_FLOAT,       // data's type
                     0,              // Offset to data in buffer
                     3);             // number of components (3 for x,y,z)
```

  - Bind the shader program
  - Bind the VBO containing the attribute data
  - Enable the desired vertex attribute array location
  - Set the attribute buffer to desired attribute location, set number of components and stride
  - Supports VBOs with interleaved data

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

Defining Uniform Values in Qt

- Yep, It is that simple! OpenGLShaderProgram has a myriad ways to do it, e.g.: `m_shaderProgram.bind();`

```
// Get the location of uniform value "uni" in the shader.
int uniLocation = m_shaderProgram.uniformLocation("uni");
// Then update the value    m_shaderProgram.
setUniformValue(uniLocation,uniValue);
// Or in one step    m_shaderProgram.setUniformValue
("uni",0.8f,0.5f,0.5f);
```

- If the value changes during an animation this code would go in the updateGL function
- If it is static it could go into initializeGL after the shader program has been linked and bound

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

Deprecated OpenGL Matrix Stack

- OpenGL Desktop version < 3 used to have "built in" matrix stacks and related functionality for dealing with transformations and projections
  - glRotate*, glTranslate*, glScale*
  - glMatrixMode(), glPushMatrix(), glPopMatrix()
  - glLoadIdentity()
  - glFrustum(), gluPerspective(...), gluLookAt(..)
- All of these are now deprecated and should/can no longer be used

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

Matrices, Qt to the Rescue

- Fortunately, it is very easy to achieve the same functionality with more flexibility using Qt
- There are functions to:
  - Create or set a matrix to the identity matrix
    - Identity matrix is a diagonal matrix with all elements being 1. When multiplied with a vector the result will be the same vector.
  - Translate, Scale, Rotate
  - Create a (view) matrix representing a “camera”
  - Create perspective or orthographic projection matrix
- And then one can use QStack, QVector, QList and gain ultimate flexibility

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

QMatrix4x4

- Contains convenient functions for handling Model, View, and Projection Matrices
  - `QMatrix4x4::translate()`
  - `QMatrix4x4::scale()`
  - `QMatrix4x4::rotate()`
  - `QMatrix4x4::lookAt()`
  - `QMatrix4x4::perspective()`
  - `QMatrix4x4::ortho()`

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

QOpenGLTexture

- Qt- 5.2 introduces QOpenGLTexture to encapsulate an OpenGL texture object
  - Makes it easy to work with OpenGL textures
  - Simplifies dealing with dependencies upon the capabilities of an OpenGL implementation
- Typical usage pattern for QOpenGLTexture is
  - Instantiate the object specifying the texture target type
  - Set properties that affect storage requirements e.g. storage format, dimensions
  - Allocate server-side storage
  - Optionally upload pixel data
  - Optionally set any additional properties e.g. filtering and border options
  - Render with texture or render to texture
  - In the common case of simply using a QImage as the source of texture pixel data most of the above steps are performed automatically.

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

Qt Support for FBO

- Qt simplifies the process with:
  - QOpenGLFramebufferObject class
    - Represents OpenGL FBO
    - By default creates 2D texture for rendering target
    - Function to return the OpenGL texture id
      - Can be used for texture rendering
    - Function to return rendered scene as a QImage
  - QOpenGLFramebufferObjectFormat()
    - Specify format and attachments of FBO

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

Qt and OpenGL Extensions

- A list of all OpenGL extensions supported by the current context can be retrieved with a call to
  `QSet<QByteArray> QOpenGLContext::extensions() const`
  The context or a sharing context must be current.
- Resolve the entry points if the extension introduces a new API: `QOpenGLContext::getProcAddress()`.
- QtOpenGLExtensions module contains a class for every OpenGL extension in the Khronos registry that introduces new API.

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

OpenGL Debugging with Qt

- OpenGL programming can be error prone
  - Black screen syndrom. There is no indication what is going on?
  - To be sure that no errors are being returned from OpenGL implementation check `glGetError` after every API call
  - OpenGL errors stack up so need to use this in a loop.
  - Additional information e.g. performance issues, warnings about using deprecated APIs are not reported through the ordinary OpenGL error reporting mechanisms
- QOpenGLDebugLogger enables logging of OpenGL debugging messages
  - Provides access to the OpenGL debug log if OpenGL implementation supports it (by exposing the GL_KHR_debug extension)
  - Messages from the OpenGL server will either be logged in an internal OpenGL log or passed in "real-time", i.e. as they're generated from OpenGL, to listeners

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

OpenGL Debugging with Qt

- Creating an OpenGL Debug Context
  - OpenGL implementations are allowed not to create any debug output at all, unless the OpenGL context is a debug context
  - Set `QSurfaceFormat::DebugContext` format option on the QSurfaceFormat used to create the QOpenGLContext object:

```
format.setOption(QSurfaceFormat::DebugContext);
```

- Creating and Initializing a QOpenGLDebugLogger
  - QOpenGLDebugLogger is a simple QObject-derived class
  - Create an instance and initialize it before usage by calling `initialize()` with a current OpenGL context:

```
QOpenGLContext *ctx = QOpenGLContext::currentContext();
QOpenGLDebugLogger *logger = new QOpenGLDebugLogger(this);
logger->initialize();
```

  - Note that `GL_KHR_debug` extension must be available in the context in order to access the messages logged by OpenGL
  - You can check the presence of this extension by calling:

```
ctx->hasExtension(QByteArrayLiteral("GL_KHR_debug"))
```

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

Qt OpenGL Debug Messages

- Reading the Internal OpenGL Debug Log
  - Messages stored in the internal log of debug messages can be retrieved by using the **`loggedMessages()`** function

```
QList<QOpenGLDebugMessage> messages = logger->loggedMessages();
foreach (const QOpenGLDebugMessage &message, messages)
    qDebug() << message;
```

  - Internal log has limited size; Older messages will get discarded to make room for new incoming messages
- Real-time logging of messages
  - Receive a stream of debug messages from the OpenGL server as they are generated by the implementation
  - Connect a slot to the **`messageLogged()`** signal, and start logging by calling **`startLogging()`**:

```
connect(logger, &QOpenGLDebugLogger::messageLogged, receiver,
&LogHandler::handleLoggedMessage);
logger->startLogging();
```

- Similarly, logging can be disabled at any time by calling the **`stopLogging()`** function.

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

QOpenGLTimeMonitor

- Measure GPU execution time of OpenGL calls
- Use to profile an application's rendering performance
- Timed results in nanoseconds
- Create and set number of samples that will be taken, e.g:

```
m_timeMonitor = new QOpenGLTimeMonitor(this);
m_timeMonitor->setSampleCount(3);
if (!m_timeMonitor->create())
        ...Handle error
```

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

QOpenGLTimeMonitor

- QOpenGLTimeMonitor::recordSample() to record interval

```
m_timeMonitor->recordSample();
glClear( GL_COLOR_BUFFER_BIT );
m_timeMonitor->recordSample();
glDrawArrays( GL_TRIANGLES, 0, 3 );
m_timeMonitor->recordSample();
```

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

QOpenGLTimeMonitor

- Call `waitForSamples()` or `waitForIntervals()` to retrieve samples or intervals (in nanoseconds)

```
QVector<GLuint64> samples = m_timeMonitor->waitForSamples();
QVector<GLuint64> intervals = m_timeMonitor->waitForIntervals();
```

  - These functions block until values are ready
  - Call `isResultAvailable()` to prevent blocking
- Reset to use again

```
m_timeMonitor->reset();
```

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved

Conclusion

- Qt has many classes that make working with OpenGL much more efficient.
- Cross platform capabilities of Qt enhance the portability of OpenGL applications greatly.
- Developer efficiency translates directly to maintenance costs and time to market.
- Ideal for scientific applications, visualization and even some games.

© 2015 Integrated Computer Solutions, Inc. All Rights Reserved