OpenGL + Qt Tutorial

Andreas Nicolai

Version 0.1.0, März 2020

Table of Contents

1. Einführung

Dieses Tutorial ist ***kein*** OpenGL Tutorial. Man sollte also OpenGL selbst schon ganz gut kennen. Natürlich kann man die hier vorgestellten Beispiele als Vorlage nehmen, aber es geht hier wirklich darum, die Qt-Klassen und vorbereitete Funktionalität zu verstehen und sinnvoll zu nutzen.

Es gibt eine PDF-Version des Tutorials:

https://github.com/ghorwin/OpenGLWithQt-Tutorial/raw/master/docs/OpenGLQtTutorial.pdf

1.1. Kernthemen

In diesem Tutorial geht es primär um folgende Themen:

- Integration von OpenGL in eine Qt Widget Anwendung (es werden verschiedene Ansätze diskutiert), einschließlich Fehlerbehandlung
- Verwendung der Qt-Wrapper-Klassen als Ersatz für native OpenGL Aufrufe (die Dokumentation vieler OpenGL-Qt-Klassen ist bisweilen etwas dürftig)
- Implementierung von Keyboard- und Maussteuerung
- Rendering-on-Demand mit Fokus auf CAD/Virtual Design Anwendungen, d.h. batterieschonendes Rendern nur, wenn sich Camera oder Scene ändern

Es wird eine hinreichend aktuelle Qt-Version vorausgesetzt, mindestens **Qt 5.4**. Bei meinem Ubuntu 18.04 System ist Qt 5.9 dabei, das dürfte also eine gute Basisversion für dieses Tutorial sein. Funktionen neuerer Qt Versionen betrachte ich nicht.

Qt enthält aus Kompatibilitätsgründen noch eine Reihe von OpenGL-Implementierungsklassen (im OpenGL Modul), welche alle mit `QGL...` beginnen. Diese sind veraltet und sollten in neuen Programmen nicht mehr verwendet werden. In aktuellen Qt Programmen sind die Hilfsklassen für OpenGL-Fenster im GUI-Modul enthalten.

1.2. Plattformunterstützung und OpenGL-Version

Das Tutorial addressiert Desktopanwendungen, d.h. *Linux*, *Windows* und *MacOS* Widgets-Anwendungen. Daher ist OpenGL ES (ES für Embedded Systems) kein Thema für dieses Tutorial. Das Wesentliche sollte aber übertragbar sein.

Hinsichtlich der OpenGL-Version wird Mac-bedingt Version 3.3 angepeilt. Hinsichtlich der Einbettung von OpenGL in Qt Widgets-Anwendungen spielt die OpenGL-Version eigentlich keine Rolle.

Im Rahmen dieses Tutorials wird für die Beispiele das Qt bzw. qmake Buildsystem verwendet. Das Thema *Compilieren mit CMake* und *Deployment von OpenGL-basierten Anwendungen* wird in einem speziellen Tutorial erklärt.

1.3. Grundlagen

Als Einstieg in OpenGL empfehle ich folgende (englischsprachige) Webseiten:

- https://learnopengl.com : ein gutes und aktuelles Tutorial mit guten Abbildungen und guter Mischung aus C++

und C, mein Tutorial orientiert sich inhaltlich an den hier dargestellten Themen

- http://antongerdelan.net/opengl : englisch, gute Illustrationen und Erklärungen zu einzelnen Themen
- http://www.opengl-tutorial.org : eher grundlegendes Tutorialset, C und GLUT werden verwendet

Mein Tutorial selbst basiert zum Teil auf folgenden Webtutorials:

- https://www.trentreed.net/blog/qt5-opengl-part-0-creating-a-window : in diesem Tutorial und den Forumkommentaren gibt es einige Anregungen, allerdings ist dies eher eine Dokumentation eigener Versuche grafisch optimale Effekte zu erziehlen. Es gibt aber interessante Anregungen. Manche Quelltextumsetzung sind nicht ganz optimal, daher mit Vorsicht als Vorlage für eigene Programme verwenden. (Diese Kleinigkeiten, über die ich selber auch gestolpert bin, sind u.A. der Grund für dieses Tutorial.)

2. Tutorial 01: OpenGL innerhalb eines QWindow

Das Ziel ist erstmal einfach: ein einfarbiges Dreieck mit OpenGL in einem `QWindow` zu zeichnen.

Das sieht dann so (noch ziemlich langweilig) aus, reicht aber aus, um mehrere Seiten Tutorialtext zu füllen :-)

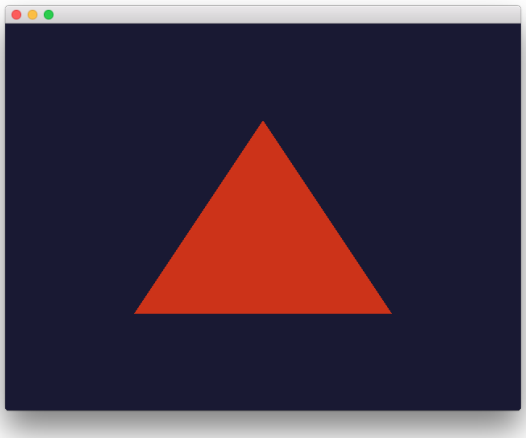

Figure 1. Ausgabe: Tutorial_01 (Mac OS Screenshot)

Quelltext für dieses Tutorial liegt im github repo: Tutorial_01

`.pro`-Datei in Qt Creator öffnen und compilieren.

Das Tutorial selbst basiert zum einen auf dem Qt Beispiel "OpenGLWindow" und auf dem Tutorial https://learnopengl.com/Getting-started/Hello-Triangle.

Beim Rendern von OpenGL Inhalten mit Qt gibt es verschiedene Möglichkeiten. Hier soll zunächst ein `QWindow` verwendet werden, welches ein natives Fenster des jeweiligen Betriebssystems kapselt. Damit kann man also ziemlich direkt und plattformnah zeichnen.

2.1. QWidget näher betrachtet

Um ein QWidget zu verwenden, muss man die Klasse ableiten und sollte dann einige Funktionen implementieren. Eine minimalistische Klassendeklaration sähe z.B. so aus:

```
class OpenGLWindow : public QWindow {
    Q_OBJECT
public:
    explicit OpenGLWindow(QWindow *parent = 0);

    // ... other public members ...

protected:
    bool event(QEvent *event) Q_DECL_OVERRIDE;
    void exposeEvent(QExposeEvent *event) Q_DECL_OVERRIDE;

private:
    // ... private members ...
};
```

Das Makro `Q_DECL_OVERRIDE` wird zum Schlüsselwort `override`, wenn der Compiler dies unterstützt (C++11 erlaubt). Da das eigentlich bei Qt 5 vorausgesetzt werden kann, könnte man eigentlich immer gleich `override` schreiben.

Man kann entweder mit einem rasterbasierten QPainter zeichnen, oder eben mit OpenGL. Dies legt man am besten im Constructor der Klasse fest, wie beispielsweise:

```
OpenGLWindow::OpenGLWindow(QWindow *parent) :
    QWindow(parent)
{
    setSurfaceType(QWindow::OpenGLSurface);
}
```

Durch Aufruf der Funktion setSurfaceType(QWindow::OpenGLSurface) legt man fest, dass man ein natives OpenGL-Window erstellen möchte.

Das Qt Framework sendet nun zwei für uns interessante Events:

- `QEvent::UpdateRequest` - wir sollten das Widget neu zeichnen
- `QEvent::Expose` - das Fenster (oder ein Teil davon) ist nun sichtbar und sollte aktualisiert werden

Für letzteres Event existiert eine überladene Funktion void exposeEvent(QExposeEvent *event), welche wir implementieren:

```
void OpenGLWindow::exposeEvent(QExposeEvent * /*event*/) {
    renderNow(); // simply redirect call to renderNow()
}
```

Wir leiten einfach die Anfrage an das Zeichnen des Bildes an eine Funktion weiter, die das macht (dazu kommen wir gleich).

In der Implementierung der generischen Ereignisbehandlungsfunktion `event()` picken wir uns nur das `UpdateRequest`-Ereignis heraus:

```
bool OpenGLWindow::event(QEvent *event) {
    switch (event->type()) {
        case QEvent::UpdateRequest:
            renderNow(); // now render the image
            return true;
        default:
            return QWindow::event(event);
    }
}
```

Damit wäre dann unsere Aufgabe klar - eine Funktion `renderNow()` zu implementieren, die mit OpenGL zeichnet.

2.2. Allgemeingültige Basisklasse für OpenGL-Render-Fenster

Die nachfolgend beschriebene Funktionalität kann man für beliebige OpenGL-Anwendungen nachnutzen, daher wird das ganze in Form einer abstrakten Basisklasse `OpenGLWindow` implementiert.

Wir erweitern die Klassendeklaration geringfügig:

```
class OpenGLWindow : public QWindow, protected QOpenGLFunctions {
    Q_OBJECT
public:
    explicit OpenGLWindow(QWindow *parent = 0);

    virtual void initialize() = 0;
    virtual void render() = 0;

public slots:
    void renderLater();
    void renderNow();

protected:
    bool event(QEvent *event) Q_DECL_OVERRIDE;
    void exposeEvent(QExposeEvent *event) Q_DECL_OVERRIDE;

    QOpenGLContext *m_context; // wraps the OpenGL context
};
```

Der Zugriff auf die nativen OpenGL Funktionen ist in Qt in der Klasse `QOpenGLFunctions` gekapselt. Diese kann entweder als Datenmember gehalten werden, oder eben wie oben gezeigt als Implementierung vererbt werden. Da es sich ja um ein `OpenGLWindow` handelt, fühlt sich das mit der Vererbung schon richtig an.

Es gibt zwei pur virtuelle Funktionen, `initialize()` und `render()`, ohne die kein OpenGL-Programm auskommt. Daher verlangen wir von Nutzern dieser Basisklasse, dass sie diese Funktionen bereitstellen (Inhalt wird später erläutert).

Neben der Funktion `renderNow()`, welche ja oben bereits aufgerufen wurde, und deren Aufgabe das *sofortige* OpenGL-Zeichnen ist, gibt es noch eine weitere Funktion `renderLater()`. Deren Aufgabe ist es letztlich, einen Neu-Zeichen-Aufruf passend zum Vertical-Sync anzufordern, was letztlich dem Absenden eines `UpdateRequest`-Ereignisses in die Anwendungs-Ereignis-Schleife entspricht. Das macht die Funktion requestUpdate():

```
void OpenGLWindow::renderLater() {
    // Schedule an UpdateRequest event in the event loop
    // that will be send with the next VSync.
    requestUpdate(); // call public slot requestUpdate()
}
```

Man kann sich strenggenommen die Funktion auch sparen, und direkt den Slot `requestUpdate()` aufrufen, aber die Benennung zeigt letztlich an, dass erst beim nächsten VSync gezeichnet wird.

Zur Synchronisation mit Bildwiederholraten kann man an dieser Stelle schon einmal zwei Dinge vorwegnehmen:

- es wird doppelgepuffert gezeichnet
- Qt ist standardmäßig zu konfiguriert, dass das `QEvent::UpdateRequest` immer zu einem VSync gesendet wird. Es wird natürlich bei einer Bildwiederholfrequenz von 60Hz vorausgesetzt, dass die Zeit bis zum Umschalten des Zeichenpuffers nicht mehr als ~16 ms ist.

Die Variante mit dem Absenden des `UpdateRequest` in die Ereignisschleife hat den Vorteil, dass mehrere Aufrufe dieser Funktion (z.B. via Signal-Slot-Verbindung) innerhalb eines Sync-Zyklus (d.h. innerhalb von 16ms) letztlich zu einem Ereignis zusammengefasst werden, und so nur *einmal* je VSync gezeichnet wird. Wäre sonst ja auch eine Verschwendung von Rechenzeit.

Zuletzt sei noch auf die neuen private Membervariable `m_context` hingewiesen. Dieser Kontext kapselt letztlich den nativen OpenGL Kontext, d.h. den Zustandsautomaten, der bei OpenGL verwendet wird. Obwohl dieser dynamisch erzeugt wird, brauchen wir keinen Destruktor, da wir über die QObject-Eltern-Beziehung auch automatisch `m_context` mit aufräumen.

Im Konstruktor initialisieren wir die Zeigervariable mit einem nullptr.

```
OpenGLWindow::OpenGLWindow(QWindow *parent) :
    QWindow(parent),
    m_context(nullptr)
{
    setSurfaceType(QWindow::OpenGLSurface);
}
```

2.2.1. Initialisierung des OpenGL-Fensters

Es gibt nun verschiedenen Möglichkeiten, das OpenGL-Zeichenfenster zu initialisieren. Man könnte das gleich im Konstruktor tun, wobei dann allerdings alle dafür benötigten Resourcen (auch eventuell Meshes/Texturen, ...) bereits initialisiert sein sollten. Für ein schnellen Anwendungsstart wäre das hinderlich. Besser ist es, dies später zu machen.

Man könnten nun eine eigene Initialisierungsfunktion implementieren, die der Nutzer der Klasse anfänglich aufruft. Oder man regelt dies beim allerersten Anzeigen des Fensters. Hier gibt es einiges an Spielraum und je nach Komplexität und Fehleranfälligkeit der Initialisierung ist die Variante mit einer expliziten Initialisierungsfunktion sicher gut.

Hier wird die Variante der Initialisierung-bei-erster-Verwendung genutzt (was nebenbei ja ein übliches Pattern bei Verwendung von Dialogen in Qt ist). Damit ist die Funktion `renderNow()` gefordert, die Initialisierung anzustoßen:

```
void OpenGLWindow::renderNow() {
    // only render if exposed
    if (!isExposed())
        return;

    bool needsInitialize = false;

    // initialize on first call
    if (m_context == nullptr) {
        m_context = new QOpenGLContext(this);
        m_context->setFormat(requestedFormat());
        m_context->create();

        needsInitialize = true;
    }

    m_context->makeCurrent(this);

    if (needsInitialize) {
        initializeOpenGLFunctions();
        initialize(); // call user code
    }

    render(); // call user code

    m_context->swapBuffers(this);
}
```

Die Funktion wird einmal von `exposeEvent()` und von `event()` aufgerufen. In beiden Fällen sollte nur gezeichnet werden, wenn das Fenster tatsächlich sichtbar ist. Daher wird über die Funtion `isExposed()` zunächst geprüft, ob es überhaupt zu sehen ist. Wenn nicht, dann raus.

Jetzt kommt die oben angesprochene Initialisierung-bei-erster-Benutzung. Zuerst wird das `QOpenGLContext` Objekt erstellt. Als nächstes werden verschiedene OpenGL-spezifische Anforderungen gesetzt, wobei die im QWindow-gesetzten Formate an den QOpenGLContext übergeben werden.

Die Funktion requestedFormat() liefert das für das `QWindow` eingestellte Format der Oberfläche (`QSurfaceFormat` zurück. Dieses enthält Einstellungen zu den Farb- und Tiefenpuffern, und auch zum Antialiasing des OpenGL-Renderes.

Zum Zeitpunkt der Initialisierung des OpenGL-Context muss also dieses Format bereits für das QWindow festgelegt worden sein, d.h. *bevor* das erste Mal `show()` für das OpenGLWindow aufgerufen wird.

Wenn man diese Fehlerquelle vermeiden will, muss man die Initialisierung unter Anforderung des gewünschten `QSurfaceFormat` tatsächlich in eine spezielle Funktion verschieben.

Mit dem Aufruf von `m_context->create()` wird der OpenGL Kontext (also Zustand) erstellt, wobei die vorab gesetzten Formatparameter verwendet werden.

Falls man später die Formatparameter ändern möchte (z.B. Antialiasing), so muss zunächst wieder das Format im Kontextobjekt neu gesetzt werden und danach `create()` neu aufgerufen werden. Dies löscht und ersetzt dann den vorherigen Kontext.

Nachdem der Kontext erzeugt wurde, stehen die wohl wichtigsten Funktionen `makeCurrent()` und `swapBuffers()` zur

Verfügung.

Der Aufruf `m_context->makeCurrent(this)` überträgt den Inhalt des Kontext-Objekts in den OpenGL-Zustand.

Der zweite Schritt der Initialisierung besteht im Aufruf der Funktion QOpenGLFunctions::initializeOpenGLFunctions(). Hierbei werden letztlich die plattformspezifischen OpenGL-Bibliotheken dynamisch eingebunden und die Funktionszeiger auf die nativen OpenGL-Funktionen (`glXXX...`) geholt.

Zuletzt wird noch die Funktion `initialize()` mit nutzerspezifischen Initialisierungen aufgerufen.

Das eigentliche Rendern der 3D Szene muss der Anwender dann in der Funktion `render()` erledigen (dazu kommen wir gleich).

Am Ende tauschen wir noch mittels `m_context->swapBuffers(this)` den Fensterpuffer mit dem Renderpuffer aus.

Nachdem der Fensterpuffer aktualisiert wurde, kann das Fenster beliebig auf dem Bildschirm verschoben oder sogar minimiert werden, *ohne* dass wir neu rendern müssen. Dies gilt zumindest solange, bis wir anfangen, in der Szene mit Animationen zu arbeiten. Bei Anwendungen ohne Animationen ist es deshalb sinnvoll, nicht automaisch jeden Frame neu zu rendern, wie das bei Spieleengines wie Unity/Unreal/Irrlicht etc. gemacht wird.

Falls wir dennoch animieren wollen (und wenn es nur eine weiche Kamerafahrt wird), dann sollten wir am Ende der Funktion `renderNow()` die Funktion `renderLater()` aufrufen, und so beim nächsten VSync einen neuen Aufruf erhalten. Ach ja: wenn das Fenster versteckt ist (nicht *exposed*), dann würde natürlich die Funktion schnell verlassen werden, und die Funktion `renderLater()` wird nicht aufgerufen. Damit wäre dann die Animation gestoppt. Damit sie wieder losläuft, gibt es die implementierte Ereignisfunktion `exposeEvent()`, die das Rendering wieder anstößt.

Damit wäre die zentrale Basisklasse für OpenGL-Renderfenster fertig. Wir testen das jetzt mit dem ganz am Anfang erwähnten primitiven Dreiecksbeispiel.

2.3. Implementierung eines konkreten Renderfensters

Vor der Lektüre diese Abschnitts sollte man den Tutorialteil https://learnopengl.com/Getting-started/Hello-Triangle überflogen haben (oder sich zumindest soweit mit OpenGL auskennen).

Das konkrete Renderfenster heißt in diesem Beispiel `TriangleWindow` mit der Headerdatei `TriangleWindow.h`. Die Klassendeklaration ist recht kurz:

```
/*  This is the window that shows the triangle.
    We derive from our OpenGLWindow base class and implement the
    virtual initialize() and render() functions.
*/
class TriangleWindow : public OpenGLWindow {
public:
    TriangleWindow();
    ~TriangleWindow() Q_DECL_OVERRIDE;

    void initialize() Q_DECL_OVERRIDE;
    void render() Q_DECL_OVERRIDE;

private:
    // Wraps an OpenGL VertexArrayObject (VAO)
    QOpenGLVertexArrayObject    m_vao;
    // Vertex buffer (only positions now).
    QOpenGLBuffer               m_vertexBufferObject;

    // Holds the compiled shader programs.
    QOpenGLShaderProgram        *m_program;
};
```

Interessant sind die privaten Membervariablen, die nachfolgend in der Implementierung der Klasse näher erläutert werden.

2.3.1. Shaderprogramme

Die Klasse `QOpenGLShaderProgram` kapselt ein Shaderprogramm und bietet verschiedene Bequemlichkeitsfunktionen, die in nativen OpenGL-Aufrufe umgesetzt werden.

Zuerst wird das Objekt erstellt:

Funktion: TriangleWindow::initialize()

```
void TriangleWindow::initialize() {
    // this function is called once, when the window is first shown, i.e. when
    // the the window content is first rendereds

    // build and compile our shader program
    // ------------------------------------

    m_program = new QOpenGLShaderProgram();

    ...
```

Dies entspricht in etwa den folgenden OpenGL-Befehlen:

```
unsigned int shaderProgram;
shaderProgram = glCreateProgram();
```

Es gibt nun eine ganze Reihe von Möglichkeiten, Shaderprogramme hinzuzufügen. Für das einfache Dreieck brauchen wir nur ein Vertex-Shader und ein Fragment-Shaderprogramme. Die Implementierungen dieser Shader sind in zwei Dateien abgelegt:

Vertex-Shader: shader/pass_through.vert

```
#version 330 core

// vertex shader

// input:  attribute named 'position' with 3 floats per vertex
layout (location = 0) in vec3 position;

void main() {
  gl_Position = vec4(position, 1.0);
}
```

Fragment-Shader: shaders/uniform_color.frag

```
#version 330 core

// fragment shader

out vec4 FragColor;  // output: fertiger Farbwert als rgb-Wert

void main() {
  FragColor = vec4(0.8, 0.2, 0.1, 1);
}
```

Der Vertexshader schiebt die Vertexkoordinaten (als vec3) einfach als vec4 ohne jede Transformation raus. Und der Fragmentationshader gibt einfach nur die gleiche Farbe (dunkles Rot) aus.

Compilieren und Linken von Shaderprogrammen

Die nächsten Zeilen in der `initialize()` Funktion übersetzen die Shaderprogramme und linken die Programme:

Funktion: TriangleWindow::initialize(), fortgesetzt

```
    if (!m_program->addShaderFromSourceFile(
        QOpenGLShader::Vertex, ":/shaders/pass_through.vert"))
    {
        qDebug() << "Vertex shader errors :\n" << m_program->log();
    }

    if (!m_program->addShaderFromSourceFile(
    QOpenGLShader::Fragment, ":/shaders/uniform_color.frag"))
    {
        qDebug() << "Fragment shader errors :\n" << m_program->log();
    }

    if (!m_program->link())
        qDebug() << "Shader linker errors :\n" << m_program->log();
```

Es gibt mehrere überladene Funktionen `addShaderFromSourceFile()` in der Klasse QOpenGLShaderProgram, hier wird die Variante mit Übernahme eines Dateinamens verwendet. Die Dateien sind in einer `.qrc` Resourcendatei referenziert und daher über die Resourcenpfade `:/shaders/...` angeben. Wichtig ist die Angabe des Typs des Shaderprogramms, hier `QOpenGLShader::Vertex` und `QOpenGLShader::Fragment`.

Erfolg oder Fehler wird über den Rückgabecode signalisiert. Das Thema Fehlerbehandlung wird aber in einem späteren Tutorial noch einmal aufgegriffen.

Letzter Schritt ist das Linken der Shaderprogramme, d.h. das Verknüpfen selbstdefinierter Variablen (Kommunikation zwischen Shaderprogrammen).

Die Funktionen der Klasse `QOpenGLShaderProgram` kapseln letztlich OpenGL-Befehle der Art:

Native OpenGL Shaderprogramm-Initialisierung

```
// create the shader
unsigned int vertexShader;
vertexShader = glCreateShader(GL_VERTEX_SHADER);

// pass shader program in C string
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);

// compile the shader
glCompileShader(vertexShader);

// check success of compilation
int  success;
char infoLog[512];
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);

// print out an error if any
if (!success) {
    glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
    std::cout << "Vertex shader error:\n" << infoLog << std::endl;
}


// ... same for fragment shader

// attach shaders to shader program
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);

// and link
glLinkProgram(shaderProgram);
```

Verglichen damit ist die Qt Variante mit "etwas" weniger Tippaufwand verbunden.

2.3.2. Vertex-Buffer-Objekte (VBO) und Vertex-Array-Objekte (VBA)

Nachdem das Shaderprogramm fertig ist, erstellen wir zunächst ein Vertexpufferobjekt mit den Koordinaten des Dreiecks. Danach werden dann die Zuordnungen der Vertexdaten zu Attributen festgelegt. Und damit man diese Zuordnungen nicht immer wieder neu machen muss, merkt man sich diese in einem VertexArrayObject (VBA). Auf den ersten Blick ist das alles ganz schön kompliziert, daher machen wir das am Besten am Beispiel.

Vertexpufferobjekte (engl. *Vertex Buffer Objects (VBO)*) beinhalten letztlich die Daten, die an den Vertex-Shader gesendet werden. Aus Sicht von OpenGL müssen diese Objekte erst erstellt werden, dann gebunden werden (d.h. nachfolgende OpenGL-Befehle beziehen sich auf den Puffer), und dann wieder freigegeben werden.

Funktion: TriangleWindow::initialize(), fortgesetzt

```
    float vertices[] = {
        -0.5f, -0.5f, 0.0f,
         0.5f, -0.5f, 0.0f,
         0.0f,  0.5f, 0.0f
    };

    // create a new buffer for the vertices
    m_vertexBufferObject = QOpenGLBuffer(QOpenGLBuffer::VertexBuffer); // VBO
    m_vertexBufferObject.create(); // create underlying OpenGL object
    m_vertexBufferObject.setUsagePattern(QOpenGLBuffer::StaticDraw); // must be called before allocate

    m_vertexBufferObject.bind(); // set it active in the context, so that we can write to it
    // int bufSize = sizeof(vertices) = 9 * sizeof(float) = 9*4 = 36 bytes
    m_vertexBufferObject.allocate(vertices, sizeof(vertices) ); // copy data into buffer
```

Im obigen Quelltext wird zunächst ein statisches Array mit 9 floats (3 x 3 Vektoren) definiert. Z-Koordinate ist jeweils 0. Nun erstellen wir ein neues VertexBufferObject vom Typ `QOpenGLBuffer::VertexBuffer`. Der Aufruf von `create()` erstellt das Objekt selbst und entspricht in etwa dem OpenGL-Aufruf:

```
unsigned int VBO;
glGenBuffers(1, &VBO);
```

Dann wird dem QOpenGLBuffer-Pufferobjekt noch die geplante Zugriffsart via `setUsagePattern()` mitgeteilt. Dies führt keinen OpenGL Aufruf aus, sondern es wird sich dieses Attribute für später gemerkt.

Mit dem Aufruf von `bind()` wird dieses VBO als Aktiv im OpenGL-Kontext gesetzt, d.h. nachfolgende Funktionsaufrufe mit Bezug auf VBOs beziehen sich auf unser erstelltes VBO. Dies entspricht dem OpenGL-Aufruf:

```
glBindBuffer(GL_ARRAY_BUFFER, VBO);
```

Zuletzt werden die Daten im Aufruf von `allocate()` in den Puffer kopiert. Dies entspricht in etwa einem memcpy-Befehl, d.h. Quelladresse des Puffers wird übergeben und Länge in Bytes as zweites Argument. In diesem Fall sind es 9 floats, d.h. 9*4=36 Bytes. Dies entspricht dem OpenGL-Befehl:

```
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

Hier wird der vorab gesetzte Verwendungstyp (usagePattern) verwendet. Deshalb ist es wichtig, `setUsagePattern()` immer *vor* `allocate()` aufzurufen.

Der Puffer ist nun gebunden und man könnte nun die Vertex-Daten mit den Eingangsparametern im Shaderprogramm verknüpfen. Da wir dies nicht jedesmal vorm Zeichnen erneut machen wollen, verwenden wir ein VertexArrayObject (VBA), welches letztlich so etwas wie ein Container für derartige Verknüpfungen darstellt. Man kann sich so ein VBA wie eine Aufzeichnung der nachfolgenden Verknüpfungsbefehle vorstellen, wobei der jeweils aktive Vertexpuffer und die verknüpften Variablen kollektiv gespeichert werden. Später beim eigentlichen Zeichnen muss man nur noch das VBA einbinden, welches unter der Haube dann alle aufgezeichneten Verknüpfungen abspielt und so den OpenGL-Zustand entsprechend wiederherstellt.

Konkret sieht das so aus:

Funktion: TriangleWindow::initialize(), fortgesetzt

```
    // Initialize the Vertex Array Object (VAO) to record and remember subsequent attribute assocations with
    // generated vertex buffer(s)
    m_vao.create(); // create underlying OpenGL object
    m_vao.bind(); // sets the Vertex Array Object current to the OpenGL context so it monitors attribute assignments

    // now all following enableAttributeArray(), disableAttributeArray() and setAttributeBuffer() calls are
    // "recorded" in the currently bound VBA.

    // Enable attribute array at layout location 0
    m_program->enableAttributeArray(0);
    m_program->setAttributeBuffer(0, GL_FLOAT, 0, 3);
    // This maps the data we have set in the VBO to the "position" attribute.
    // 0 - offset - means the "position" data starts at the begin of the memory array
    // 3 - size of each vertex (=vec3) - means that each position-tuple has the size of 3 floats (those are the 3
coordinates,
    //     mind: this is the size of GL_FLOAT, not the size in bytes!
```

Zunächst wir das Vertex-Array-Objekt erstellt und eingebunden. Danach werden alle folgenden Aufrufe von `enableAttributeArray()` und `setAttributeBuffer()` vermerkt.

Der Befehl `enableAttributeArray(0)` aktiviert ein Attribut (bzw. Variable) im Vertex-Puffer, welches im Shaderprogramm dann mit dem layout-Index 0 angesprochen werden kann. Im Vertex-Shader dieses Beispiels (siehe oben) ist das der *position* Vektor.

Mit `setAttributeBuffer()` wird nun definiert, wo im Vertex-Buffer die Daten zu finden sind, d.h. Datentyp, Anzahl (hier 3 floats entsprechend den 3 Koordinaten) und dem Startoffset (hier 0).

Diese beiden Aufrufe entsprechen den OpenGL-Aufrufen:

```
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
```

Damit sind alle Daten initialisiert, und die Pufferobjekte können freigegeben werden:

Funktion: TriangleWindow::initialize(), fortgesetzt

```
    // Release (unbind) all
    m_vertexBufferObject.release();
    m_vao.release(); // not really necessary, but done for completeness
}
```

Dies entspricht den OpenGL-Aufrufen:

```
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);
```

Man sieht also, dass die Qt-Klassen letztlich die nativen OpenGL-Funktionsaufrufe (mitunter ziemlich direkt) kapseln.

Die Qt API fühlt sich hier nicht ganz glücklich gewählt an. Aufrufe wie `m_programm->enableAttributeArray(0)` sugggerieren, dass hier tatsächlich Objekteigenschaften geändert werden, dabei wird tatsächlich mit dem OpenGL-Zustandsautomaten gearbeitet. Entsprechend ist bei etlichen Befehlen die Reihenfolge der Aufrufe wichtig, obgleich es bei individuell setzbaren Attributen eines Objekts eigentlich egal sein sollte, welches Attribut man zuerst setzt. Daher habe ich oben im Tutorial auch noch einmal explizit die dahinterliegenden OpenGL-Befehle angegeben.

Es ist daher empfehlenswert, dass man die Qt API nochmal in eigene Klassen einpackt, und dann eine entsprechend schlange und fehlerunanfällige API entwirft.

2.3.3. Rendern

Das eigentliche Render erfolgt in der Funktion `render()`, die als rein virtuelle Funktion von der Basisklasse `OpenGLWindow` aufgerufen wird. Die Basisklasse prüft ja auch, ob Rendern überhaupt notwendig ist, und setzt den aktuellen OpenGL Context. Dadurch kann man in dieser Funktion direkt losrendern.

Die Implementierung ist (noch) recht selbsterklärend:

Funktion: TriangleWindow::render()

```
void TriangleWindow::render() {
    // this function is called for every frame to be rendered on screen
    const qreal retinaScale = devicePixelRatio(); // needed for Macs with retina display
    glViewport(0, 0, width() * retinaScale, height() * retinaScale);

    // set the background color = clear color
    glClearColor(0.1f, 0.1f, 0.2f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    // use our shader program
    m_program->bind();
    // bind the vertex array object, which in turn binds the vertex buffer object and
    // sets the attribute buffer in the OpenGL context
    m_vao.bind();
    // now draw the triangles:
    // - GL_TRIANGLES - draw individual triangles
    // - 0 index of first triangle to draw
    // - 3 number of vertices to process
    glDrawArrays(GL_TRIANGLES, 0, 3);
    // finally release VAO again (not really necessary, just for completeness)
    m_vao.release();
}
```

Die ersten drei `glXXX` Befehle sind native OpenGL-Aufrufe, und sollten eigentlich in dieser Art mehr oder weniger immer auftauchen. Die Anpassung des ViewPort (`glViewport(...)`) ist für resize-Operationen notwendig, das Löschen des Color Buffers (`glClear(...)`) auch (später werden in diesem Aufruf noch andere Puffer gelöscht werden). Die Funktion devicePixelRatio() ist für Bildschirme mit angepasster Skalierung interessant (vornehmlich für Macs mit Retina-Display).

Solange sich die Hintergrundfarbe (clear-color) nicht ändert, könnte man diesen Aufruf auch in die Initialisierung verschieben.

Danach kommt der interessante Teil. Es wird das Shader-Programm gebunden (`m_programm->bind()`) und danach das Vertex Array Objekt (VAO) (`m_vao.bind()`). Letzteres sorgt dafür, dass im OpenGL-Kontext auch das Vertex-Buffer-

Objekt und die Attributzuordnung gesetzt werden. Damit kann dann einfach gezeichnet werden, wofür mit `glDrawArrays(...)` wieder ein nativer OpenGL-Befehl zum Einsatz kommt.

Dieser Teil des Programms sähe in nativem OpenGL-Code so aus:

```
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
glDrawArrays(GL_TRIANGLES, 0, 3);
glBindVertexArray(0);
```

Ist also ziemlich ähnlich.

2.3.4. Resourcenfreigabe

Bleibt noch das Aufräumen der reservierten Resourcen im Destructor.

```
TriangleWindow::~TriangleWindow() {
    // resource cleanup

    // since we release resources related to an OpenGL context,
    // we make this context current before cleaning up our resources
    m_context->makeCurrent(this);

    m_vao.destroy();
    m_vertexBufferObject.destroy();
    delete m_program;
}
```

Da einige Resourcen dem OpenGL-Kontext des aktuellen Fenster gehören, sollte man vorher den OpenGL-Kontext "aktuell" setzen (`m_context->makeCurrent(this);`), damit diese Resourcen sicher freigegeben werden können.

Damit wäre dann die Implementierung des `TriangleWindow` komplett.

2.4. Das Hauptprogramm

Das `TriangleWindow` kann jetzt eigentlich direkt als Top-Level-Fenster verwendet werden. Allerdings ist zu beachten, dass *vor* dem ersten Anzeigen (und damit vor der OpenGL-Initialisierung und Erstellung des OpenGL-Kontext) die Oberflächeneigenschaften (`QSurfaceFormat`) zu setzen sind:

```
int main(int argc, char **argv) {
    QGuiApplication app(argc, argv);

    // Set OpenGL Version information
    QSurfaceFormat format;
    format.setRenderableType(QSurfaceFormat::OpenGL);
    format.setProfile(QSurfaceFormat::CoreProfile);
    format.setVersion(3,3);

    TriangleWindow window;
    // Note: The format must be set before show() is called.
    window.setFormat(format);
    window.resize(640, 480);
    window.show();

    return app.exec();
}
```

Das wäre dann erstmal eine Grundlage, auf der man aufbauen kann. Interessanterweise bietet Qt selbst eine Klasse an, die unserer OpenGLWindow-Klasse nicht unähnlich ist. Diese schauen wir uns in *Tutorial 02* an.

3. Tutorial 02: Alternative: die Klasse QOpenGLWindow

In diesem Teil schauen wir uns die Klasse QOpenGLWindow an. Mit Hilfe dieser Klasse (die letztlich die Klasse OpenGLWindow aus dem *Tutorial 01* ersetzt) erstellen wir ein leicht modifiziertes Zeichenprogramm (2 Dreiecke, welche ein buntes Rechteck ergeben und via Element-Index-Array gezeichnet werden). Zuerst aber schauen wir an, was die Klasse unter der Haube macht.

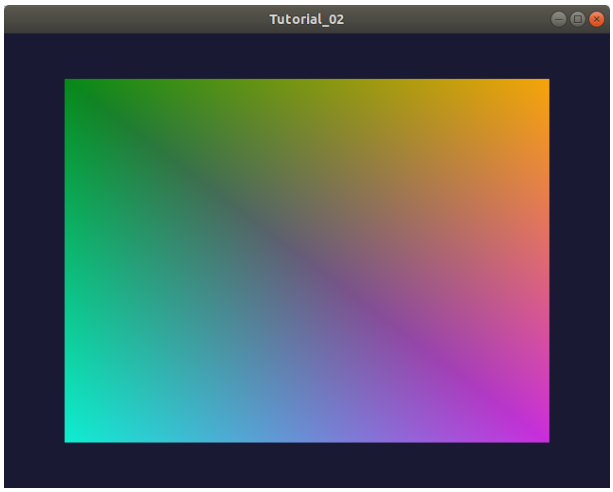

Figure 2. Ausgabe: Tutorial_02 (Linux/Ubuntu Screenshot)

Wer mit der Funktionalität des OpenGLWindows aus *Tutorial 01* zufrieden ist, kann gleich mit *Tutorial 03* weitermachen.

3.1. Verwendung der Klasse

Eine interessante Eigenschaft des `QOpenGLWindow` ist die Möglichkeit, nur einen Teil des Fensters neu zu zeichnen. Das wird über die `UpdateBehavior`-Eigenschaft gesteuert. Interessant ist das eigentlich nur, wenn man mittels rasterbasiertem QPainter Teile des Bildes aktualisieren möchte. Es gibt 3 Varianten:

- `QOpenGLWindow::NoPartialUpdate` - das gesamte Bild wird jedes Mal neu gezeichnet (es wird kein zusätzlicher Framebuffer erzeugt und verwendet)
- `QOpenGLWindow::PartialUpdateBlit` - man zeichnet nur einen Teil des Bildes neu, und das in einem zusätzlichen, automatisch erstellten Framebuffer. Nach Ende des Zeichnens wird einfach der neu gezeichnete Teil in den eigentlichen Framebuffer kopiert.
- `QOpenGLWindow::PartialUpdateBlend` - im Prinzip wie die 2. Varianten, nur dass diesmal der Inhalt nicht kopiert, sondern überblendet wird.

Ob man die 2. oder 3. Funktion braucht, hängt sicher von der Anwendung ab. Für viele OpenGL-Anwendungen wird das vielleicht nicht notwendig sein, daher schauen wir uns hier mal Variante mit `QOpenGLWindow::NoPartialUpdate` an.

Die Klasse `QOpenGLWindow` bietet 5 interessante virtuelle Funktionen an:

```
virtual void initializeGL();            // initialization stuff
virtual void paintGL();                 // actual painting
virtual void paintOverGL();             // not needed for NoPartialUpdate
virtual void paintUnderGL();            // not needed for NoPartialUpdate
virtual void resizeGL(int w, int h);    // to update anything related to view port
                                        // size (projection matrix etc.)
```

Die Funktion `initializeGL()` macht eigentlich das Gleiche, wie in Tutorial 01 die Funktion `initialize()`.

Die Funktion `paintGL()` macht das Gleiche, wie in Tutorial 01 die Funktion `render()`, d.h. hier wird das Bild mit OpenGL gezeichnet.

Die Funktionen `paintOverGL()` und `paintUnderGL()` werden im Modus `QOpenGLWindow::NoPartialUpdate` nicht benötigt.

Letztlich ist die Funktion `resizeGL(int w, int h)` nur eine Bequemlichkeitsfunktion, aufgerufen aus der `event()` Funktion für das `QEvent::ResizeEvent`. Hier kann man z.B. die Projektionsmatrix an den neuen Viewport anpassen oder sonstige Größenanpassungen vornehmen.

3.2. Die Implementierung der Klasse QOpenGLWindow

Um die Gemeinsamkeiten und Unterschiede zur OpenGLWindow-Klasse aus Tutorial 01 zu verstehen, schauen wir uns mal die Klassenimplementierung an. Die Quelltextschnipsel stammen aus der Qt Version 5.14, sollten aber im Vergleich zu vorherigen Versionen nicht groß verändert sein.

Wichtigster Unterschied ist schon die Vererbungshierarchie. `QOpenGLWindow` leitet von `QOpenGLPaintDevice` ab, welches hardwarebeschleunigtes Zeichnen mit dem rasterbasierten `QPainter` erlaubt. Allerdings gibt es einen kleinen Haken. Zitat aus dem Handbuch:

> Antialiasing in the OpenGL paint engine is done using multisampling. Most hardware require significantly more memory to do multisampling and the resulting quality is not on par with the quality of the software paint engine. The OpenGL paint engine's strength lies in its performance, not its visual rendering quality.
>
> — Qt Documentation 5.9 zu QOpenGLPaintDevice

Das hat insofern Auswirkung auf das Gesamterscheinungsbild der Anwendung, wenn im OpenGL Fenster verwaschene Widgets oder Kontrollen gezeichnet werden, daneben aber klassische Widgets mit scharfen Kanten. Man kennt das Problem vielleicht von den verwaschenen Fenstern in Windows 10, wenn dort die Anwendungen letztlich in einen Pixelpuffer zeichnen, welcher dann als Textur in einer 3D Oberfläche interpoliert gerendert wird. Sieht meiner Meinung nach doof aus :-)

Hilfreich kann das dennoch sein, wenn man existierende Zeichenfunktionalität (basierend auf QPainter) in einem OpenGL-Widget verwenden möchte. Falls man die Funktionalität nicht braucht, bringt das PaintDevice und die dafür benötigte Funktionalität *etwas unnützen Overhead* (vor allem Speicherverbrauch) mit sich.

Schauen wir uns nun die Gemeinsamkeiten an.

3.2.1. Constructor

Der Konstruktor sieht erstmal fast genauso aus, wie der unserer `OpenGLWindow`-Klasse. abgesehen davon, dass die Argumente in die private Pimpl-Klasse weitergeleitet werden.

```
QOpenGLWindow::QOpenGLWindow(QOpenGLWindow::UpdateBehavior updateBehavior, QWindow *parent)
    : QPaintDeviceWindow(*(new QOpenGLWindowPrivate(nullptr, updateBehavior)), parent)
{
    setSurfaceType(QSurface::OpenGLSurface);
}
```

3.2.2. Ereignisbehandlungsroutinen

Interessanter sind schon die Ereignisbehandlungsroutinen:

```
void QOpenGLWindow::paintEvent(QPaintEvent * /*event*/ ) {
    paintGL();
}

void QOpenGLWindow::resizeEvent(QResizeEvent * /*event*/ ) {
    Q_D(QOpenGLWindow);
    d->initialize();
    resizeGL(width(), height());
}
```

Das `paintEvent()` wird einfach an die vom Nutzer zu implementierende Funktion `paintGL()` weitergereicht. Insofern analog zu der Ereignisbehandlung im OpenGLWidget, welches auf `QEvent::UpdateRequest` wartet. Allerdings sind auf dem Weg bis zum Aufruf der `paintEvent()` Funktion etliche Zwischenschritte implementiert, bis zum Erzeugen des QPaintEvent-Objekts, welches gar nicht benötigt wird. Der Aufwand wird deutlich, wenn man sich die Aufrufkette anschaut:

```
QPaintDeviceWindow::event(QEvent *event)  // waits for QEvent::UpdateRequest
QPaintDeviceWindowPrivate::handleUpdateEvent()
QPaintDeviceWindowPrivate::doFlush()  // calls QPaintDeviceWindowPrivate::paint()

    bool paint(const QRegion &region)
    {
        Q_Q(QPaintDeviceWindow);
        QRegion toPaint = region & dirtyRegion;
        if (toPaint.isEmpty())
            return false;

        // Clear the region now. The overridden functions may call update().
        dirtyRegion -= toPaint;

        beginPaint(toPaint); // here we call QOpenGLWindowPrivate::beginPaint()

        QPaintEvent paintEvent(toPaint);
        q->paintEvent(&paintEvent); // here we call QOpenGLWindowPrivate::paintEvent()

        endPaint(); // here we call QOpenGLWindowPrivate::endPaint()

        return true;
    }
```

Alternativ wird `paintGL()` noch aus der Ereignisbehandlungsroutine `QPaintDeviceWindow::exposeEvent()` aufgerufen, wobei dort direkt `QPaintDeviceWindowPrivate::doFlush()` gerufen wird. Die Funktionen `beginPaint()` und `endPaint()` kümmern sich um den temporären Framebuffer, in dem beim UpdateBehavior `QOpenGLWindow::PartialUpdateBlit` und `QOpenGLWindow::PartialUpdateBlend` gerendert wird. Ohne diese Modi passiert in der Funktion sehr wenig.

3.2.3. Initialisierung

Interessant ist noch der Initialisierungsaufruf, der in der `resizeEvent()` Ereignisbehandlungsroutine steckt.

```
void QOpenGLWindowPrivate::initialize()
{
    Q_Q(QOpenGLWindow);

    if (context)
        return;

    if (!q->handle())
        qWarning("Attempted to initialize QOpenGLWindow without a platform window");

    context.reset(new QOpenGLContext);
    context->setShareContext(shareContext);
    context->setFormat(q->requestedFormat());
    if (!context->create())
        qWarning("QOpenGLWindow::beginPaint: Failed to create context");
    if (!context->makeCurrent(q))
        qWarning("QOpenGLWindow::beginPaint: Failed to make context current");

    paintDevice.reset(new QOpenGLWindowPaintDevice(q));
    if (updateBehavior == QOpenGLWindow::PartialUpdateBlit)
        hasFboBlit = QOpenGLFramebufferObject::hasOpenGLFramebufferBlit();

    q->initializeGL();
}
```

Eigentlich sieht die Funktion fast genauso wie der Initialisierungsteil der Funktion `OpenGLWindow::renderNow()` aus *Tutorial 01* aus. Abgesehen natürlich davon, dass noch ein `QOpenGLWindowPaintDevice` erzeugt wird.

3.2.4. Zusammenfassung

Das `QOpenGLWindow` ist im Modus `QOpenGLWindow::NoPartialUpdate` eigentlich vergleichbar mit unserem minimalistischen `OpenGLWindow`. Etwas Overhead ist vorhanden, allerdings sollte der in realen Anwendungen keine Rolle spielen. Es spricht also eigentlich nichts dagegen, direkt mit dem `QOpenGLWindow` anzufangen. Für spätere Erweiterungen (Maus- und Tastatureingabebehandlung) ist dennoch eine von `QOpenGLWindow` abgeleitete Klasse nötig. Wenn man also die zusätzlichen Funktionen (QPainter-Zeichnen, Buffer-Blenden etc.) von `QOpenGLWindow` nicht braucht, kann man auch mit dem schlanken `OpenGLWindow` aus *Tutorial 01* weitermachen.

3.3. Zeichnen mit Indexpuffern

Als Erweiterung zum *Tutorial 01* soll im Anwendungsbeispiel für `QOpenGLWindow` einmal ein Indexpuffer verwendet werden. Zwei Erweiterungen werden vorgestellt:

- interleaved Vertex-Puffer (d.h. Koordinaten und Farben zusammen in einem Puffer)
- indexbasiertes Elementzeichnen (und den dafür benötigten Elementpuffer)

Die Implementierung des `RectangleWindow` ist zunächst mal fast identisch zum `TriangleWindow` aus *Tutorial 01*:

RectangleWindow.h

```
/*  This is the window that shows the two triangles to form a rectangle.
    We derive from our QOpenGLWindow base class and implement the
    virtual initializeGL() and paintGL() functions.
*/
class RectangleWindow : public QOpenGLWindow {
public:
    RectangleWindow();
    virtual ~RectangleWindow() Q_DECL_OVERRIDE;

    void initializeGL() Q_DECL_OVERRIDE;
    void paintGL() Q_DECL_OVERRIDE;

private:
    // Wraps an OpenGL VertexArrayObject (VAO)
    QOpenGLVertexArrayObject    m_vao;
    // Vertex buffer (positions and colors, interleaved storage mode).
    QOpenGLBuffer               m_vertexBufferObject;
    // Index buffer to draw two rectangles
    QOpenGLBuffer               m_indexBufferObject;

    // Holds the compiled shader programs.
    QOpenGLShaderProgram        *m_program;
};
```

Die wesentlichsten Erweiterungen sind:

- die Klasse erbt von `QOpenGLWindow`
- die Initialisierung erfolgt in der Funktion `initializeGL()` (vormals `TriangleWindow::initialize()`)
- das Rendern erfolgt in der Funktion `paintGL()` (vormals `TriangleWindow::render()`)
- es gibt eine neue Variable vom Typ `QOpenGLBuffer`, welche wir für den Indexpuffer verwenden.

3.3.1. Initialisierung von gemischten Vertex-Puffern

Die Initialisierung beginnt wie in *Tutorial 01* unverändert mit dem Erstellen und Compilieren des Shaderprogramms. Nun wird der Vertex-Buffer erstellt. Diesmal werden nicht nur Koordinaten in den Buffer geschrieben, sondern auch Farben, und zwar abwechselnd (=*interleaved*) (siehe https://learnopengl.com/Getting-started/Hello-Triangle für eine Erläuterung).

Es wird ein Rechteck gezeichnet, und zwar durch zwei Dreiecke. Dafür brauchen wir 4 Punkte. Der Vertexpuffer-Speicherblock soll am Ende so aussehen: `p0c0|p1c1|p2c2|p3c3`, wobei p für eine Position (vec3) und c für eine Farbe (vec3) steht. Die Daten werden zunächst in statischen Arrays separat definiert.

```
// set up vertex data (and buffer(s)) and configure vertex attributes
// ------------------------------------------------------------------

float vertices[] = {
     0.8f,  0.8f, 0.0f,  // top right
     0.8f, -0.8f, 0.0f,  // bottom right
    -0.8f, -0.8f, 0.0f,  // bottom left
    -0.8f,  0.8f, 0.0f   // top left
};

QColor vertexColors [] = {
    QColor("#f6a509"),
    QColor("#cb2dde"),
    QColor("#0eeed1"),
    QColor("#068918"),
};
```

Die noch getrennten Daten werden jetzt in einen gemeinsamen Speicherbereich kopiert.

```
// create buffer for 2 interleaved attributes: position and color, 4 vertices, 3 floats each
std::vector<float> vertexBufferData(2*4*3);
// create new data buffer - the following memory copy stuff should
// be placed in some convenience class in later tutorials
// copy data in interleaved mode with pattern p0c0|p1c1|p2c2|p3c3
float * buf = vertexBufferData.data();
for (int v=0; v<4; ++v, buf += 6) {
    // coordinates
    buf[0] = vertices[3*v];
    buf[1] = vertices[3*v+1];
    buf[2] = vertices[3*v+2];
    // colors
    buf[3] = vertexColors[v].redF();
    buf[4] = vertexColors[v].greenF();
    buf[5] = vertexColors[v].blueF();
}
```

Es gibt sicher viele andere Varianten, die Daten in der gewünschten Reihenfolge in den Speicherblock zu kopieren. Es fällt vielleicht auf, dass der gemeinsame Pufferspeicher in einem lokal erstellen `std::vector` liegt. Das wirft die Frage nach der (benötigten) Lebensdauer für diese Pufferspeicher auf.

```
// create a new buffer for the vertices and colors, interleaved storage
m_vertexBufferObject = QOpenGLBuffer(QOpenGLBuffer::VertexBuffer);
m_vertexBufferObject.create();
m_vertexBufferObject.setUsagePattern(QOpenGLBuffer::StaticDraw);
m_vertexBufferObject.bind();

// now copy buffer data over: first argument pointer to data, second argument: size in bytes
m_vertexBufferObject.allocate(vertexBufferData.data(), vertexBufferData.size()*sizeof(float) );
```

Im letzten Aufruf wird der Pufferspeicher tatsächlich *kopiert*. Der Aufruf zu `allocate()` ist sowohl Speicherreservierung im OpenGL-Puffer, als auch kopieren der Daten (wie mit `memcpy`).

Danach wird der Vector `vertexBufferData` nicht mehr benötigt, oder könnte sogar für weitere Puffer verwendet und verändert werden.

3.3.2. Element-/Indexpuffer

In ähnlicher Weise wird nun der Elementpuffer erstellt, allerdings gibt es eine OpenGL-Besonderheit zu beachten:

Das *Vertex Array Object* verwaltet nicht nur die Attribute, sondern auch gebundene Puffer. Daher muss das VAO *vor* dem Elementpuffer gebunden werden, um dann den Zustand korrekt zu speichern.

Deshalb wird nun zuerst das VAO erstellt und gebunden (kann man auch ganz am Anfang machen)

```
// create and bind Vertex Array Object - must be bound *before* the element buffer is bound,
// because the VAO remembers and manages element buffers as well
m_vao.create();
m_vao.bind();
```

und dann erst der Elementpuffer erzeugt:

```
unsigned int indices[] = {  // note that we start from 0!
    0, 1, 3,   // first triangle
    1, 2, 3    // second triangle
};

// create a new buffer for the indexes
m_indexBufferObject = QOpenGLBuffer(QOpenGLBuffer::IndexBuffer); // Mind: use 'IndexBuffer' here
m_indexBufferObject.create();
m_indexBufferObject.setUsagePattern(QOpenGLBuffer::StaticDraw);
m_indexBufferObject.bind();
m_indexBufferObject.allocate(indices, sizeof(indices) );
```

Qt (und auch OpenGL) unterscheidet nicht zwischen Pufferobjekten für verschiedene Aufgaben. Erst beim Binden des Puffers an den OpenGL Kontext (beispielsweise durch den Aufruf `glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);`) wird die Verwendung des Puffers festgelegt.

In Qt muss man die Art des Puffers als Construktor-Argument übergeben, wobei `QOpenGLBuffer::VertexBuffer` der Standard ist. Für den Index-/Elementpuffer muss man `QOpenGLBuffer::IndexBuffer` übergeben.

Wie gesagt, es ist lediglich die Reihenfolge des Bindens wichtig, also man könnte auch die Puffer erst erstellen und befüllen und zum Schluss die

4. Tutorial 03: Renderfenster eingebettet in einen QDialog

In diesem Teil des Tutorials geht es darum, wie das QWindow-basierte Renderfenster (siehe *Tutorial 01* und *02*), in eine QWidgets-Anwendung eingebettet wird.

Man könnte auch die Bequemlichkeitsklasse `QOpenGLWidget` verwenden. In *Tutorial 04* schauen wir uns an, wie diese Klasse intern funktioniert und ob es ggfs. Performancenachteile geben könnte, wenn man diese Klasse verwendet.