OpenGL + Qt Tutorial

Andreas Nicolai
andreas.nicolai@tu-dresden.de

Version 0.5.0, März 2020

Inhaltsverzeichnis

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.

Diese Anleitung soll auch ***nicht*** zeigen, wie man mit Qt ein Spiel- oder eine Spieleengine schreibt. Es geht eher um technische Anwendungen, in denen *Animationen keine Rolle spielen*. Fokus liegt eher darauf, effizient und resourcenschonend (und damit Laptop-Akku-schonend) 3D Grafik zu verwenden.

Es gibt eine PDF-Version des Tutorials:

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

Die Quelltexte (und Inhalte dieses Tutorials) liegen auf github:

https://github.com/ghorwin/OpenGLWithQt-Tutorial

Fragen und Anregungen kann man in der Issues-Seite auf Githab eintragen, die kann man ja wie ein Diskussionsforum verwenden, nur dass die nie geschlossen werden :-)

https://github.com/ghorwin/OpenGLWithQt-Tutorial/issues

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.

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.

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, eine Lektüre der ersten paar Kapitel dieses Tutorials sollte eigentlich ausreichen, um alle in meinem Tutorial verwendeten OpenGL-Befehle und Techniken zu verstehen.
- 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 durchaus 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).

1. 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 :-)

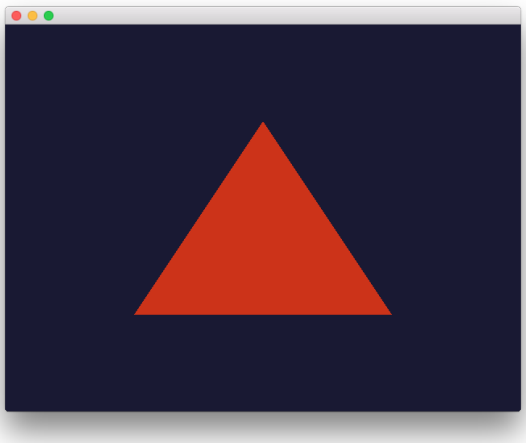

Figure 1. Ausgabe: Tutorial_01 (Mac OS Screenshot)

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.

1.1. 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.

1.2. 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);
}
```

1.2.1. 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.

1.3. 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.

1.3.1. 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.

1.3.2. 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.

1.3.3. 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.

1.3.4. 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.

1.4. 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.

2. Tutorial 02: Alternative: die Klasse QOpenGLWindow

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

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-Puffer gezeichnet werden).

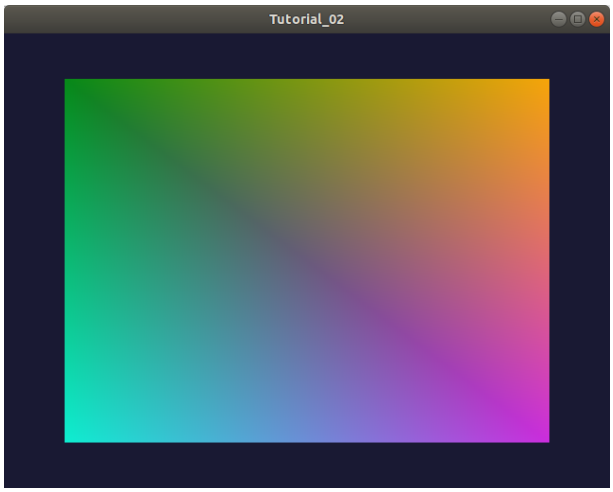

Figure 2. Ausgabe: Tutorial_02 (Linux/Ubuntu Screenshot)

Quelltext für dieses Tutorial liegt im github repo: Tutorial_02

Zuerst schauen wir an, was die Klasse `OpenGLWindow` unter der Haube macht.

2.1. 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.

2.2. 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.

2.2.1. 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);
}
```

2.2.2. 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.

2.2.3. 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.

2.3. Zeichnen mit Index-/Elementpuffern

Als Erweiterung zum *Tutorial 01* soll im Anwendungsbeispiel für `QOpenGLWindow` 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.

2.3.1. Shaderprogramm

Die Initialisierung beginnt wie in *Tutorial 01* unverändert mit dem Erstellen und Compilieren des Shaderprogramms. Da diesmal Farben verwendet werden, müssen beide Shaderprogramme angepasst werden:

Vertexshader "shaders/pass_through.vert"

```
#version 330 core

// vertex shader

// input:  attribute named 'position' with 3 floats per vertex
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 color;

out vec4 fragColor;

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

Es gibt nun zwei Vertex-Attribute:

- layout location 0 = Position (als vec3 Koordinate)
- layout location 1 = Farbe (auch als vec3, rgb Farbwerte je im Bereich 0..1)

Der Farbwert eines Vertex wird als Ausgabevariable *fragColor* einfach als vec4 weitergereicht und kommt dann, bereits fertig interpoliert, als *fragColor* im Fragmentshader an. Dort wird er unverändert ausgegeben.

Fragmentshader "shaders/simple.frag"

```
#version 330 core

// fragment shader

in vec4 fragColor;    // input: interpolated color as rgba-value
out vec4 finalColor;  // output: final color value as rgba-value

void main() {
  finalColor = fragColor;
}
```

Das Laden, Compilieren und Linken der Shader im Shaderprogramm wird genauso wie in *Tutorial 01* gemacht.

2.3.2. Initialisierung von gemischten Vertex-Puffern

Als nächstes 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.

2.3.3. 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. Der eigentliche Pufferinhalt wird wieder beim Aufruf von `allocate()` kopiert.

2.3.4. Attribute im gemischten Vertexarray

Bei der Verwendung gemischter Vertexarrays (mehrer Attribute je Vertex) muss man dem Shaderprogramm die Speicherstruktur und die Abbildung der Attribute angeben (zur Erläuterung siehe wiederum Hello-Triangle Tutorial).

Die Syntax von `QOpenGLShaderProgram::setAttributeBuffer` entspricht im wesentlichen dem nativen OpenGL-Aufruf `glVertexAttribPointer`:

```
// stride = number of bytes for one vertex (with all its attributes) = 3+3 floats = 6*4 = 24 Bytes
int stride = 6*sizeof(float);

// layout location 0 - vec3 with coordinates
m_program->enableAttributeArray(0);
m_program->setAttributeBuffer(0, GL_FLOAT, 0, 3, stride);

// layout location 1 - vec3 with colors
m_program->enableAttributeArray(1);
int colorOffset = 3*sizeof(float);
m_program->setAttributeBuffer(1, GL_FLOAT, colorOffset, 3, stride);
```

Wie gesagt, für die korrekte Komposition des VAO es ist lediglich die Reihenfolge des Bindens und der `setAttributeBuffer()`-Aufrufe wichtig. Man könnte also auch die Puffer erst erstellen und befüllen und zum Schluss die folgenden Aufrufe in der geforderten Reihenfolge schreiben:

```
m_vao.bind(); // VAO binden
// Puffer binden und Daten kopieren
m_vertexBufferObject.bind();
m_vertexBufferObject.allocate(vertexBufferData.data(), vertexBufferData.size()*sizeof(float) );
m_indexBufferObject.bind();
m_indexBufferObject.allocate(indices, sizeof(indices) );
// Attribute setzen
m_program->setAttributeBuffer(...)
```

In ähnlicher Art und Weise werden Bufferdaten auch aktualisiert (wird noch in einem späteren Tutorial besprochen).

2.3.5. Freigabe der Puffer

Bei der Freigabe der Puffer ist die Reihenfolge wichtig. Damit sich das VAO den Zustand des eingebundenden Elementpuffers merkt, darf man diesen *nicht vor* Freigabe des VAO freigeben. Am Besten man gibt nur Vertexbuffer und VAO frei, und auch das nur, wenn es notwendig ist. Es wird im Beispiel auch nur der Vollständigkeithalber gemacht.

```
// Release (unbind) all
m_vertexBufferObject.release();
m_vao.release();
```

Explizites Freigeben von VBO oder VAO ist eigentlich nur notwendig, wenn man mit verschiedenen VAOs arbeitet und/oder verschiedenen Shadern. Dann sollte man auf Zustand im aktuellen OpenGL-Kontext achten und bewusst OpenGL-Objekte einbinden und freigeben.

2.3.6. Rendern

Das eigentliche Zeichnen erfolgt in der `paintGL()` Funktion, welche fast genauso aussieht wie die `TriangleWindow::render()` Funktion aus *Tutorial 01*.

```
void RectangleWindow::paintGL() {
    // 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();
    // For old Intel drivers you may need to explicitely re-bind the index buffer, because
    // these drivers do not remember the binding-state of the index/element-buffer in the VAO
    //  m_indexBufferObject.bind();

    // now draw the two triangles via index drawing
    // - GL_TRIANGLES - draw individual triangles via elements
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, nullptr);
    // finally release VAO again (not really necessary, just for completeness)
    m_vao.release();
}
```

Das Anpassen des Viewports (OpenGL-Aufruf `glViewport()`) kann entfallen, da das bereits in der Basisklasse gemacht wurde.

Dann folgen eigentlich die üblichen 4 Schritte:

1. Shaderprogramm binden
2. Vertex Array Objekt binden (und damit Binden des Vertex- und Elementpuffers, und setzen der Attribut-Zeiger)
3. Rendern, diesmal mit `glDrawElements` statt `glDrawArrays`, und
4. freigeben des VAO (damit danach weitere Objekte gezeichnet werden können).

Bei einigen älteren Intel-Treibern wurde der Zustand des eingebundenen Elementpuffers noch nicht korrekt im VAO gespeichert und wiederhergestellt. Daher musste man den Index-/Elementpuffer vor dem Zeichnen immer nochmal explizit einbinden (siehe auskommentierter Quelltext).

Bei aktuellen Treibern scheint das aber kein Problem mehr zu sein (zumindest nicht unter Ubuntu).

2.4. Zusammenfassung

Das `QOpenGLWindow` ist im Modus `QOpenGLWindow::NoPartialUpdate` eigentlich vergleichbar mit unserem minimalistischen `OpenGLWindow` aus *Tutorial 01*. 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.

Wie man nun ein solches QWindow-basiertes (natives) OpenGL-Fenster in eine Widgets-Anwendung integriert bekommt, beschreibt *Tutorial 03*.

3. Tutorial 03: Renderfenster in einem QDialog eingebettet

In diesem Teil des Tutorials geht es darum, ein QWindow-basiertes OpenGL-Renderfenster (siehe *Tutorial 01* und *02*), in eine QWidgets-Anwendung einzubetten.

Der erste Teil des Tutorials beschäftigt sich allein mit der Einbettung (und ist recht kurz). Damit das Tutorial aber noch etwas interessanter wird, gibt es im 2. Abschnitt noch zwei Interaktionsvarianten mit und ohne Animation.

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.

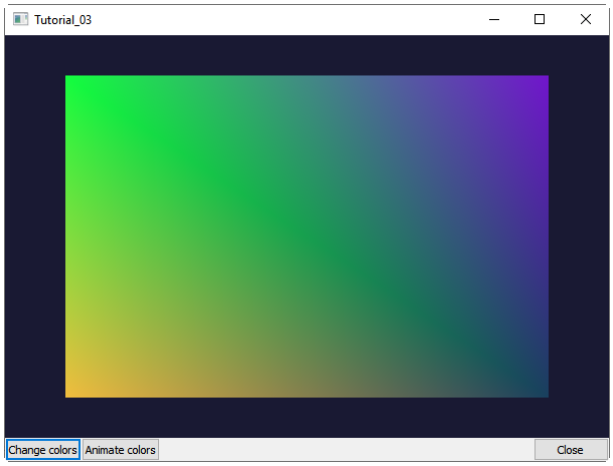

Figure 3. Tutorial_03 (Windows 10 Screenshot)

Quelltext für dieses Tutorial liegt im github repo: Tutorial_03

3.1. Window Container Widgets

Die Funktion `QWidget::createWindowContainer` erstellt ein `QWidget`, welches das als Argument übergebene `QWindow` einbettet. So einfach kann's sein:

TestDialog.cpp:Konstruktor

```
// *** create OpenGL window

QSurfaceFormat format;
format.setRenderableType(QSurfaceFormat::OpenGL);
format.setProfile(QSurfaceFormat::CoreProfile);
format.setVersion(3,3);

m_rectangleWindow = new RectangleWindow;
m_rectangleWindow->setFormat(format);

// *** create window container widget

QWidget *container = QWidget::createWindowContainer(m_rectangleWindow);
container->setMinimumSize(QSize(600,400));
```

`m_rectangleWindow` ist ein Zeiger auf die aus Tutorial 02 bekannte `RectangleWindow` Klasse. Das so erstellte Container-

Widget muss natürlich noch in ein Layout gesteckt werden. Aber mehr ist eigentlich nicht zu tun.

3.2. Interaktion und Synchronisation mit dem Zeichnen

Grundsätzlich ist folgende Aktualisierungslogik anzustreben:

```
OpenGL zeichnet Bild

<Anwendung wartet in Ereignis-Schleife>

Ein Event wird abgearbeitet, ändert für die Darstellung relevante Daten. Registriert ein "UpdateRequest" in der
Ereignisschleife.

<Anwendung wartet in Ereignis-Schleife>

Ein Event wird abgearbeitet, ändert für die Darstellung relevante Daten. Registriert ein "UpdateRequest" in der
Ereignisschleife. Dieses wird mit dem bereits existierenden "UpdateRequest" verschmolzen.

Passend zum VSync wird das UpdateRequest-Event verschickt, was zum OpenGL Rendern führt. Und wieder von vorne...
```

D.h., das potenziell zeitaufwändige Aktualisieren der Puffer und Zeichendaten erfolgt stets dann, wenn noch auf den nächsten VSync gewartet wird. So hat man ca. 16 ms Zeit (by üblichen 60 FPS), alles Notwendige zu erledigen.

3.2.1. Einmalige Änderungen: Farbwechsel auf Knopfdruck

Eine typische Anwendung, vor allem in technischen Anwendungen (d.h. nicht in Spielen), ist die diskrete Änderung der 3D Szene, sei es durch eine Kamerabewegung, Auswahl und Hervorhebung einzelner Elemente, oder Transformation der dargestellten Geometrie. Innerhalb des Qt Frameworks wird also zunächst ein Ereignis (OnClick, Maus- oder Tastatureingabe, ...) in die Ereignisschleife gelangen und dort abgearbeitet werden.

Ein Beispiel ist der "Change Color" Button im Dialog im Tutorial 03. Es gibt eine OnClick-Ereignisbehandlungsroutine:

TestDialog.cpp:TestDialog::onChangeColors()

```
// randomize the colors and change them in the OpenGL window
for (unsigned int i=0; i<4; ++i)
    m_rectangleWindow->m_vertexColors[i].setRgbF(
            rand()*1.0/RAND_MAX, rand()*1.0/RAND_MAX, rand()*1.0/RAND_MAX );

// now update the scene -> this will also request an update
m_rectangleWindow->updateScene();
```

Die Membervariable `m_vertexColors` wird mit zufälligen Farbwerten befüllt. Dann wird die Funktion `updateScene()` aufgerufen.

Zum Verständnis kann man noch einmal die geänderte Klassendeklaration von `RectangleWindow` anschauen:

RectangleWindow.h

```
class RectangleWindow : public QOpenGLWindow {
public:
    ....

    // updates the scene and requests a paint update
    void updateScene();

    // holds the vertex colors set on next call to updateScene()
    std::vector<QColor>         m_vertexColors;

private:

    // ....

    std::vector<float>          m_vertexBufferData;

};
```

Der im *Tutorial 02* noch als temporärer lokaler Speicherbereich verwendete Vector `m_vertexBufferData` ist jetzt eine Membervariable. Die zu verwendenden Farben sind in dem öffentlichen Vector `m_vertexColors` abgelegt.

Der Quelltext in diesem Tutorial-Beispiel ist natürlich sehr fehleranfällig und unsicher. Darauf kommt es aber nicht an und die notwendigen Fehlerprüfungen wurden der Übersichtlichkeit wegen weggelassen.

Die Vertexfarben werden im Konstruktor mittels C++11 Initialisierungsliste initialisiert:

RectangleWindow.cpp: Konstruktor

```
RectangleWindow::RectangleWindow() :
    m_vertexColors{         QColor("#f6a509"),
                            QColor("#cb2dde"),
                            QColor("#0eeed1"),
                            QColor("#068918") },
    m_program(nullptr),
    m_frameCount(5000)
{
}
```

Die OpenGL-Initialisierung ist minimal verändert:

RectangleWindow.cpp:initializeGL()

```
....

// resize buffer for 2 interleaved attributes: position and color, 4 vertices, 3 floats each
m_vertexBufferData.resize(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 = m_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] = m_vertexColors[v].redF();
    buf[4] = m_vertexColors[v].greenF();
    buf[5] = m_vertexColors[v].blueF();
}

....
```

Der Vertex-Puffer wird auf die richtige Größe gebracht (und bleibt so), und wird dann wie bisher belegt, wobei diesmal die Farben aus der Membervariable `m_vertexColors` kommen. Sonst ändert sich nichts.

Wenn jetzt in der Ereignisbehandlungsroutine der "Change Color" Schaltfläche die Farben in `m_vertexColors` geändert werden, hat das keinerlei Einfluss auf das OpenGL-Zeichnen. Die neuen Werte müssen erst in den OpenGL-Vertexpuffer kopiert werden.

Das passiert in der Funktion `updateScene()` (hätte auch `updateColors()` heißen können):

RectangleWindow.cpp:updateScene()

```
void RectangleWindow::updateScene() {
    // for now we only update colors

    // first update our vertex buffer memory, but only those locations that are actually changed
    float * buf = m_vertexBufferData.data();
    for (int v=0; v<4; ++v, buf += 6) {
        // colors
        buf[3] = m_vertexColors[v].redF();
        buf[4] = m_vertexColors[v].greenF();
        buf[5] = m_vertexColors[v].blueF();
    }

    // make this OpenGL context current
    makeCurrent();

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

    // and request an update
    update();
}
```

Erst wird der Puffer aktualisiert. Aber anstelle diesen komplett neu aufzubauen (und eventuell noch

Speicherbereiche neu zu reservieren), verändern wir einfach nur die Farbwerte.

Danach muss der OpenGL-Vertexpuffer die Daten bekommen. Damit der OpenGL-Context stimmt, wird `QOpenGLWindow::makeCurrent()` aufgerufen. Dann wird der Vertexpuffer eingebunden und schließlich die Daten kopiert.

Ganz zuletzt wird `QPaintDeviceWindow::update()` aufgerufen (`QOpenGLWindow` ist durch Vererbung auch ein `QPaintDeviceWindow`). Dies hängt letztlich ein `QEvent::UpdateRequest` an die Ereignisliste an, wodurch beim nächsten VSync neu gezeichnet wird.

Man kann mal eine Test-Debug-Ausgabe in die Zeichenroutine einfügen. Wenn man nun in der OnClick-Ereignisbehandlungsroutine die Funktion `updateScene()` mehrfach aufruft, wird dennoch stets nur einmal je VSync gezeichnet.

3.2.2. Animierte Farbänderung

Anstelle neue Farben sofort zu setzen, kann man diese auch animiert verändern, d.h. in jedem Frame nur ein Stück von der Ursprungsfarbe zur Zielfarbe gehen.

Man benötigt zusätzliche Membervariablen und zwei neue Funktionen:

RectangleWindow.h

```
class RectangleWindow : public QOpenGLWindow {
public:

    ....

    void animateColorsTo(const std::vector<QColor> & toColors);

private:
    // modifies the scene a bit and call updateScene() afterwards
    // when already in the final state, doesn't do anything
    void animate();

    ....

    // Stores the target colors that we animate towards
    std::vector<QColor>         m_toColors;
    // Stores the target colors that we animate from
    std::vector<QColor>         m_fromColors;
    // number of frames used for the animation
    unsigned int                m_frameCount;

};
```

Die Funktion `animateColorsTo()` wird wieder durch eine Schaltfläche angestoßen. Die Implementierung überträgt nur die Daten in die Membervariablen und ruft `animate()` auf:

RectangleWindow.cpp:animateColorsTo()

```
void RectangleWindow::animateColorsTo(const std::vector<QColor> & toColors) {
    // current colors are set to "fromColors", toColors are store in m_toColors and
    // animation counter is reset

    m_fromColors = m_vertexColors;
    m_toColors = toColors;
    m_frameCount = 0;

    animate();
}
```

Die Variable `m_frameCount` zählt die animierten Frames seit Beginn der Animation. In der Funktion `animate()` wird dann zwischen den Anfangsfarbwerten `m_fromColors` und Zielfarbwerten `m_toColors` linear (im HSV Farbraum) interpoliert:

RectangleWindow.cpp:animate()

```
void RectangleWindow::animate() {
    const unsigned int FRAMECOUNT = 120;
    // if already at framecount end, stop
    if (++m_frameCount > FRAMECOUNT)
        return; // this will also stop the frame rendering

    // update the colors
    double alpha = double(m_frameCount)/FRAMECOUNT;

    // linear blending in HSV space will probably look "interesting", but it's simple
    for (unsigned int i=0; i<m_vertexColors.size(); ++i) {
        double fromH, fromS, fromV;
        m_fromColors[i].getHsvF(&fromH, &fromS, &fromV);
        double toH, toS, toV;
        m_toColors[i].getHsvF(&toH, &toS, &toV);

        m_vertexColors[i] = QColor::fromHsvF( toH*alpha + fromH*(1-alpha),
                                              toS*alpha + fromS*(1-alpha),
                                              toV*alpha + fromV*(1-alpha));
    }

    updateScene();
}
```

Wichtig ist die Abfrage nach dem Überschreiten der Animationslänge (Anzahl von Frames). Sobald das Animationsende erreicht ist, wird die Funktion sofort verlassen und es finden keine weiteren Farbanpassungen und, was vielleicht wichtiger ist, keine weiteren UpdateRequest-Events statt. Dann wartet die Anwendung wieder einfach auf Nutzerinteraktion und verbraucht keine Resourcen.

Diese Art der Animation ist gekoppelt an *tatsächlich gezeichnete Frames*. Wenn das Fenster im Hintergrund ist (d.h. nicht *exposed*) wird die Ausführung des UpdateRequest-Events ausgesetzt, bis das Fenster wieder sichtbar ist. Damit wartet auch die Animation.

3.2.3. Zusammenfassung

Die Einbettung eines `QWindow` in eine Widgets-Anwendung ist dank Widget-Container denkbar einfach. Und was das Zusammenspiel zwischen normalen `QWidget`-basierten Eingabeereignissen und der Aktualisierung der OpenGL-

Ausgabe (synchron zur Bildwiederholfrequenz) betrifft, so sind die beiden Farbanpassungsvarianten in diesem Tutorial Beispiele, wie man das machen kann.

4. Tutorial 04: Verwendung des QOpenGLWidget

In Tutorialteil wird das QOpenGLWidget anstelle des `QOpenGLWindow` verwendet. Das Programm macht das Gleiche wie in *Tutorial 03* (nur etwas langsamer :-), aber dazu kommen wir gleich).

Damit der Screenshot nicht ganz genauso wie im letzten Tutorial aussieht, habe ich mal einen halbdurchsichtigen Hintergrund eingeschaltet - das geht aber mit dem bisherigen Implementierungsvarianten auch (siehe letzter Teil des Tutorials).

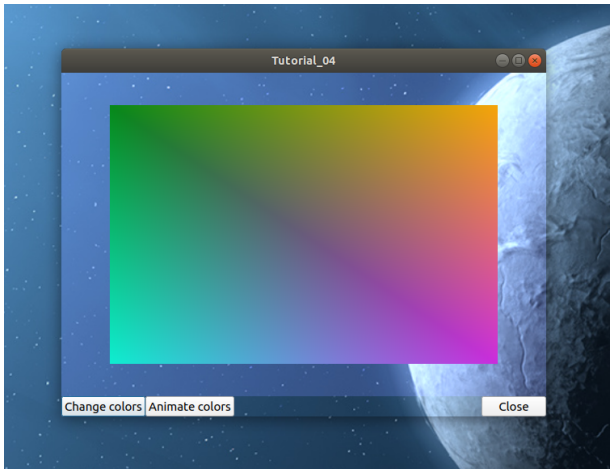

Figure 4. Tutorial_04 (Linux Screenshot, mit Transparenz)

Quelltext für dieses Tutorial liegt im github repo: Tutorial_04

4.1. Was bietet das QOpenGLWidget

Von den ganzen Qt OpenGL-Klassen ist das `QOpenGLWidget` die mit Abstand am besten dokumentierte Klasse. Es gibt ein paar interessante Details in der Dokumentation, hier ein paar Zitate:

> All rendering happens into an OpenGL framebuffer object.

...

> Due to being backed by a framebuffer object, the behavior of QOpenGLWidget is very similar to QOpenGLWindow with the update behavior set to PartialUpdateBlit or PartialUpdateBlend. This means that the contents are preserved between paintGL() calls so that incremental rendering is possible.

...

> **Note:** Most applications do not need incremental rendering because they will render everything in the view on every paint call.

Und vielleicht am Interessantesten:

> Adding a QOpenGLWidget into a window turns on OpenGL-based compositing for the entire window. In some special cases this may not be ideal, and the old QGLWidget-style behavior with a separate, native child window is desired. Desktop applications that understand the limitations of this approach (for example when it comes to overlaps, transparency, scroll views and MDI areas), can use QOpenGLWindow with QWidget::createWindowContainer(). This is a modern alternative to QGLWidget and is faster than QOpenGLWidget due to the lack of the additional composition step. It is strongly recommended to limit the usage of this approach to cases where there is no other choice. Note that this option is not suitable for most embedded and mobile platforms, and it is known to have issues on certain desktop platforms (e.g. macOS) too. The stable, cross-platform solution is always QOpenGLWidget.
>
> — Qt Documentation (5.9)

Grundlegend: Ein OpenGL bild wird beim `QOpenGLWidget` *immer* erst in einen Buffer gerendert, und dann entsprechend der Zusammensetzungsregeln (Compositing) auf den Bildschirm gezeichnet. Das dauert natürlich entsprechend länger als direktes Zeichnen (siehe Performance-Test unten).

Der wesentliche Vorteil des gepufferten Zeichnens ist die Möglichkeit des inkrementellen Renderns. Ob man das braucht, hängt wesentlich von der eigentlichen Anwendung ab. Eigentlich ist dies nur von Belang, wenn das zu rendernde Fenster aus mehreren individuellen Teilbereichen besteht. In diesem Fall könnte man aber auch die Anwendung aus mehreren OpenGL-Fenstern zusammensetzen, in in jedem Fenster individuell zeichnen.

Die letzte Anmerkung über die Portabilität und Stabilität ist vielleicht nicht ganz unwichtig. Man kann das Ganze also von 2 Seiten betrachten:

- mit `QOpenGLWidget` beginnen, und beim Auftreten von Performanceproblemen wechseln,
- mit `QOpenGLWindow` oder einer selbstgeschriebenen leichtgewichtigen Klasse wie in *Tutorial 01*, beginnen, und im Falle von Kompatibilitätsproblemen auf `QOpenGLWidget` wechseln

Hinsichtlich der Programmierschnittstelle sind die verschiedenen Klassen sich sehr ähnlich. Nachfolgend sind die einzelnen Anpassungen von *Tutorial 03* zur Verwendung von `QOpenGLWidget` aufgeführt.

4.1.1. Anpassung der Vererbungshierarchie

Der erste Schritt ist das Austauschen der Basisklasse.

RectangleWidget.h

```
class RectangleWindow : public QOpenGLWidget, protected QOpenGLFunctions {
public:
    RectangleWindow(QWidget * parent = nullptr);

    ....

protected:
    void initializeGL() Q_DECL_OVERRIDE;
    void paintGL() Q_DECL_OVERRIDE;

    ....
};
```

Die Klasse `QOpenGLWidget` erbt selbst nicht von `QOpenGLFunctions`, weswegen man diese Klasse als weitere Basisklasse angeben muss (geht auch noch anders, aber so muss im Quelltext sonst nicht viel angepasst werden). Der Konstruktor nimmt, wie andere Widgets auch, ein parent-Zeiger als Argument.

Die Funktionen `initializeGL()` und `paintGL()` sind bei `QOpenGLWidget` protected. Das war's auch schon.

4.1.2. Initialisierung

Der Konstruktor ist entsprechend zu erweitern, sodass der `parent` Zeiger an die Basisklasse weitergereicht wird:

RectangleWidget.cpp:Konstruktor

```
RectangleWindow::RectangleWindow(QWidget * parent) :
    QOpenGLWidget(parent),
    m_vertexColors{         QColor("#f6a509"),
                            QColor("#cb2dde"),
                            QColor("#0eeed1"),
                            QColor("#068918") },
    m_program(nullptr),
    m_frameCount(5000)
{
    setMinimumSize(600,400);
}
```

Da die Klasse nun ein Widget ist, kann man die minimale Größe auch gleich hier setzen.

Das Setzen der Größe muss vor dem ersten Anzeigen gemacht werden, da sonst das Widget nicht sichtbar ist (und auch nicht vergrößert werden kann).

Die Verwendung der vererbten `QOpenGLFunctions` Funktionen verlangt auch eine Initialisierung, die muss aber durch Aufruf der Funktion in `initializeOpenGLFunctions()` in `initializeGL()` erfolgen.

RectangleWidget.cpp:initializeGL()

```
void RectangleWindow::initializeGL() {
    initializeOpenGLFunctions();

    ....
}
```

Mehr ist nicht zu machen, und schon ist das `RectangleWindow` ein vollständiges Widget.

Das `UpdateBehavior` ist beim `QOpenGLWidget` standardmäßig auf `QOpenGLWidget::NoPartialUpdate` gesetzt, muss also nicht extra angepasst werden.

4.1.3. Einbettung in ein anderes QWidget

Der Widget-Container (siehe *Tutorial 03*) kann entfallen, und die Einbettung des Widgets wird wie mit jedem anderen Widget gemacht.

TestDialog.cpp:Konstruktor

```
....

m_rectangleWindow = new RectangleWindow(this);
m_rectangleWindow->setFormat(format);

// *** create the layout and insert widget container

QVBoxLayout * vlay = new QVBoxLayout;
vlay->setMargin(0);
vlay->setSpacing(0);
vlay->addWidget(m_rectangleWindow);

....
```

4.2. Performance-Vergleich

Die spannende Frage ist, wieviel langsamer ist das `QOpenGLWidget` im Vergleich zum direkten Zeichnen via `QOpenGLWindow` oder der eigenen `OpenGLWindow` Klasse aus *Tutorial 01*?

Im direkter Vergleich zwischen *Tutorial 03* und *Tutorial 04* fällt sofort auffällt auf, dass das Resize-Verhalten unterschiedlich ist. Es gibt eine merkliche Verzögerung bei der Größenänderung eines Widgets (sowohl unter Windows, als auch auf anderen Plattformen) und auch, wenn die Programme im Releasemodus kompiliert sind.

Da in diesen Testfällen nicht gerendert wird, liegt der Unterschied nur allein in der Widget-Compositing-Funktionalität im `QOpenGLWidget`.

Bei einem kleinen Benchmarktest (ca. 30 Sekunden lang mit dem Mauszeiger die Fenstergröße verändern, dabei die Anzahl der paintEvents() aufzeichnen und dann durch die Laufzeit teilen) kommt man auf:

- 25 Fensteraktualisierungen/Sekunde bei der Variante mit `QOpenGLWindow`, und
- 15 Fensteraktualisierungen/Sekunde bei der Variante mit `QOpenGLWidget`.

Das wohlgemerkt ohne OpenGL Zeichenaufrufe.

Interessant wird es, wenn man OpenGL-Animationen dazuschaltet. Dies kann man bei den Beispielen ganz einfach machen, wenn man die Frames für die Farbanimation von 120 auf, ca. 800 ändert. Dann läuft die Animation nach Klick auf "Animate Colors" ein paar Sekunden länger und man kann den CPU Overhead testen.

Bei beiden Varianten dauert die Animation exakt gleich lang, da jeweils mit nahezu 60 Frames pro Sekunde gerendert wird (bei mir zumindest).

Allerdings zeigen beide Varianten unterschiedliche CPU Auslastungen:

- 2.4% (single-core) CPU Load bei der Variante mit `QOpenGLWindow`, und
- 7.9% (single-core) CPU Load bei der Variante mit `QOpenGLWidget`.

Ein Unterschied ist da, aber sicher nicht der Rede wert. Da dürfte der optimische Verzögerungseffekt beim Vergrößern/Verkleinern eines Fensters während der Animation eher noch stören.

4.3. Transparenz

Wie schon im Screenshot zu sehen, kann man auch halb-transparente Widgetanwendungen bauen, oder auch Anwendungen mit recht unregelmäßigen Formen.

4.3.1. Mit QOpenGLWidget

Bei Verwendung des `QOpenGLWidgets` ist das recht einfach. Zunächst gibt man dem obersten Widget das Attribut `Qt::WA_TranslucentBackground`. Wer keine Titelleiste und keine Rahmen um das Fenster haben möchte, muss dem obersten Widget auch noch die Eigenschaft `Qt::FramelessWindowHint` geben, also z.B.:

main.cpp

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

    TestDialog dlg;
    // transparent window
    dlg.setAttribute(Qt::WA_TranslucentBackground, true);
    // no frame and flags.
    dlg.setWindowFlag(Qt::FramelessWindowHint, true);
    dlg.show();

    return app.exec();
}
```

In der eigentlichen Zeichenfunktion muss man nur noch die Hintergrundfarbe auf Transparent umstellen (zumindest einen Alpha-Wert < 1):

RectangleWindow.cpp:paintGL()

```
void RectangleWindow::paintGL() {
    // set the background color = clear color
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // fully transparent
    glClear(GL_COLOR_BUFFER_BIT);

    ....
}
```

Normalerweise würde man bei einem Alpha-Wert von 0 erwarten, dass der Hintergrund unverändert durchscheint, auch wenn die RGB Farbanteile der Hintergrundfarbe (clear color) irgendwelche Werte haben. Das ist aber nicht so - die Farben des Hintergrundes erscheinen etwas verblasst. Daher sollte man, wenn man wirklich den Hintergrund unverändert durchscheinen lassen möchte, die clear Color stets auf 0,0,0,0 setzen.

4.3.2. Mit QWindow-basierten OpenGL Renderfenstern

Bei den Varianten aus *Tutorial 01 .. 03* geht Transparenz auch, allerdings mit minimal mehr Aufwand. Bei der Konfiguration des `QSurfaceFormat` muss man einen AlphaBuffer festlegen (hier gezeigt beim Beispiel aus *Tutorial 01*).

main.cpp

```
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);
    format.setAlphaBufferSize(8);

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

    return app.exec();
}
```

In der Render-Funktion muss man noch Alphablending einschalten, hier gezeigt am Beispiel aus *Tutorial 01*.

TriangleWindow.cpp:render()

```
void TriangleWindow::render() {
    ....

    // Set the transparency to the scene to use the transparency of the fragment shader
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

    // set the background color = clear color
    glClearColor(0.0f, 0.0f, 0.0f, .0f);
    glClear(GL_COLOR_BUFFER_BIT);

    ....
}
```

5. Tutorial 05: Maus- und Tastatureingaben

In diesem Tutorial geht es primär um Maus- und Tastatureingaben. Und damit das irgendwie Sinn macht, brauchen wir ein (schön großes) 3D Modell, und deshalb ist dieses Tutorial auch *sehr sehr lang*. Und nebenbei geht es noch um Verwaltung von Shaderobjekten, Zeichenobjekten, Nebeleffekt beim Gitterraster und und und...

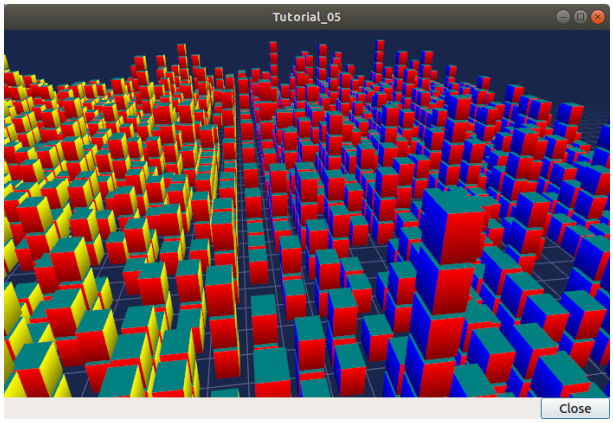

Figure 5. Tutorial_05 (Linux Screenshot), "Die Welt aus 10000 und einer Box"

Quelltext für dieses Tutorial liegt im github repo: Tutorial_05

In diesem Tutorial werden viele neue Dinge verwendet:

- zwei Modelle (eins für die Boxen und eins für das Gitter), nebst dazugehörigen, unterschiedlichen Shaderprogrammen (das vom Gitter verwendet in die Tiefe abgeblendete Farben)
- Tiefenpuffer, sodass Gitterlinien/Boxen korrekt vor/hintereinander gezeichnet werden
- Model2World und World2View-Matrizen (mit perspektivischer Projektion)
- Shaderprogramme und Renderobjekte (bzw. Objektgruppen) sind in Klassen zusammengefasst, wodurch der Quelltext deutlich übersichtlicher wird
- eine Maus+Tastatursteuerung mit WASDQE + Mauslook, incl. Shift-Langsam-Bewege-Modus
- und das Ganze wieder mit dem Schwerpunkt: Rendern nur wenn notwendig (Akku sparen!)

5.1. Überblick

Das Tutorial ist sehr lang, und der Quelltext entsprechend auch. Daher gehen wir in diesem Tutorial schrittweise vor. Die gezeigten Quelltextausschnitte stimmen daher nicht immer 100% mit dem finalen Quelltext überein (ich hab da aus didaktischen Gründen immer mal was weggelassen).

Folgende Implementierungsschritte werden besprochen:

- Anpassung der Klasse `OpenGLWindow` an die in `QOpenGLWidget` bzw. `QOpenGLWindow` verwendeten Funktionsnamen
- Vorstellung der Klasse `SceneView`, die das bisherige `TriangleWindow` oder `RectangleWindow` ersetzt
- Transformationsmatrizen: Model → World → Kamera → Projektion (Klassen `Transform3D` und `Camera`)
- Tastatur- und Mauseingabebehandlung (Klasse `KeyboardMouseHandler`)
- Kapselung der Shaderprogramme und Initialisierung und Verwendung derselben (Klasse `ShaderProgram`)
- Kapselung der Zeichenroutinen für das Gitterraster, Abblendeffekt am Horizont im Shader (Klasse `GridObject` und Shader `grid.vert` und `grid.frag`)
- Kapselung der Zeichenroutinen für die Boxen (Klassen `BoxObject` und `BoxMesh`, und Shader `withWorldAndCamera.vert` und `simple.frag`)

5.2. Fenster-Basisklasse OpenGLWindow

Als Grundlage für die Implementierung wird die Klasse `OpenGLWindow` aus *Tutorial 01* verwendet, allerdings etwas abgewandelt. Letztlich wird die Schnittstelle angepasst, um ungefähr der des `QOpenGLWidget` zu entsprechen:

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

    void initOpenGL();

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

protected:
    bool event(QEvent *event) override;
    void exposeEvent(QExposeEvent *event) override;
    void resizeEvent(QResizeEvent *) override;

    virtual void initializeGL() = 0;
    virtual void resizeGL(int width, int height) { Q_UNUSED(width) Q_UNUSED(height) }
    virtual void paintGL() = 0;

    QOpenGLContext *m_context;
};
```

Die Funktionen `initializeGL()` und `paintGL()` sind aus den vorangegangen Tutorials bekannt. Die Funktion `resizeGL()` ist eigentlich nur eine Bequemlichkeitsfunktion, welche aus dem Eventhandler `resizeEvent()` aufgerufen wird.

Neu ist jedoch die Funktion `initOpenGL()`, mit der die OpenGL-Initialisierung gezielt angestoßen werden kann. Normalerweise wird die Initialisierung beim ersten Anzeigen des Fensters (genaugenommen beim ersten ResizeEvent) aufgerufen. Dies kann aber für eine sinnvolle Fehlerbehandlung zu spät sein, weil dann das Fenster wahrscheinlich leer angezeigt wird. Daher ist es sinnvoll, die Funktion nach dem Erstellen des Renderfensters, aber vor Aufruf von `show()` auszuführen.

Macht man das nicht, so wird diese Funktion wie bisher automatisch beim ersten Anzeigen aufgerufen, konkret im ResizeEvent-Handler:

OpenGLWindow.cpp: Funktion resizeEvent()

```
void OpenGLWindow::resizeEvent(QResizeEvent * event) {
    QWindow::resizeEvent(event);

    // initialize on first call
    if (m_context == nullptr)
        initOpenGL();

    resizeGL(width(), height());
}
```

Unabhängig von dieser Initializierungsfunktion muss man natürlich die Funktion `initializeGL()` implementieren. Alles andere in der Klasse ist altbekannt.

5.3. Klasse SceneView - die konkrete Implementierung

5.3.1. Klassendeklaration

Zwecks Überblick ist hier zunächst die Klassendeklaration in Teilen. Zuerst die üblichen Verdächtigen:

SceneView.h, Deklaration der Klasse SceneView

```
class SceneView : public OpenGLWindow {
public:
    SceneView();
    virtual ~SceneView() override;

protected:
    void initializeGL() override;
    void resizeGL(int width, int height) override;
    void paintGL() override;
```

Dann kommen die Ereignisbehandlungsroutinen für die Tastatur- und Mauseingaben. Dazu gehören auch die Hilfsfunktionen `checkInput()` und `processInput()`, die im Abschnitt zur Tastatur- und Mauseingabe erklärt sind. Die Member-Variablen `m_keyboardMouseHandler` und `m_inputEventReceived` gehören auch dazu.

SceneView.h, Deklaration der Klasse SceneView, fortgesetzt

```
    void keyPressEvent(QKeyEvent *event) override;
    void keyReleaseEvent(QKeyEvent *event) override;
    void mousePressEvent(QMouseEvent *event) override;
    void mouseReleaseEvent(QMouseEvent *event) override;
    void mouseMoveEvent(QMouseEvent *event) override;
    void wheelEvent(QWheelEvent *event) override;

private:
    void checkInput();
    void processInput();

    KeyboardMouseHandler        m_keyboardMouseHandler;
    bool                        m_inputEventReceived;
```

Dann kommt die Funktion `updateWorld2ViewMatrix()` zur Koordinatentransformation und die dazugehörigen Member-Variablen.

SceneView.h, Deklaration der Klasse SceneView, fortgesetzt

```
    void updateWorld2ViewMatrix();

    QMatrix4x4                  m_projection;
    Transform3D                 m_transform;
    Camera                      m_camera;
    QMatrix4x4                  m_worldToView;
```

Zuletzt kommen Member-Variablen, die die Shader-Programme und Zeichenobjekte kapseln (beinhalten Shader, VAO, VBO, EBO, etc.)

SceneView.h, Deklaration der Klasse SceneView, fortgesetzt

```
    QList<ShaderProgram>        m_shaderPrograms;

    BoxObject                   m_boxObject;
    GridObject                  m_gridObject;
};
```

Und das war's auch schon - recht kompakt, oder?

5.3.2. Das Aktualisierungskonzept

Erklärtes Ziel dieser OpenGL-Implementierung ist nur dann zu rendern, wenn es wirklich notwendig ist. Also:

- wenn die Fenstergröße (Viewport) verändert wurde,
- wenn das Fenster angezeigt/sichtbar wird (exposed),
- wenn durch Nutzerinteraktion die Kameraposition verändert wird, und
- wenn die Szene selbst transformiert/verändert wird (z.B. programmgesteuerte Animation...)

Wenn man jetzt bei jedem Eintreffen eines solchen Ereignisses jedesmal neu zeichnen würde, wäre das mit ziemlichem Overhead verbunden. Besser ist es, beim Eintreffen eines solchen Ereignisses einfach nur ein Neuzeichnen anzufordern. Da die `UpdateRequest`-Ereignisse normalerweise mit der Bildschirmfrequenz synchronisiert sind, kann es natürlich sein, dass mehrfach hintereinander `UpdateRequest`-Events an die Eventloop angehängt werden. Dabei werden diese aber zusammengefasst und nur ein Event ausgeschickt. Es muss ja auch nur einmal je angezeigtem Frame gezeichnet werden.

Grundsätzlich muss man also nur die Funktion QWindow::requestUpdate() (oder unsere Bequemlichkeitsfunktion `renderLater()`) aufrufen, damit beim nächsten VSync wieder neu gezeichnet wird.

Leider funktionier das Verfahren im Fall des `ExposeEvent` bzw. `ResizeEvent` nicht perfekt. Gerade unter Windows führt das beim Vergrößern des Fensters zu unschönen Artefakten am rechten und unteren Bildschirmrand. Daher muss man in diesem Fall tatsächlich sofort in der Ereignisbehandlungsroutine neu zeichnen und dabei den OpenGL Viewport bereits an die neue Fenstergröße anpassen. Das Neuzeichnen wird direkt im ExposeEvent-Handler von `OpenGLWindow` ausgelöst:

OpenGLWindow.cpp:exposeEvent()

```
void OpenGLWindow::exposeEvent(QExposeEvent * /*event*/) {
    renderNow(); // update right now
}
```

Bei Größenveränderung des Fensters sendet Qt immer zuerst ein `ResizeEvent` gefolgt von einem `ExposeEvent` aus. Daher sollte man in der Funktion `SceneView::resizeEvent()` *nicht* `renderLater()` aufrufen!

Ohne eine Aufruf von `renderLater()` im ResizeEvent-Handler erhält man folgende Aufrufreihenfolge bei der Fenstervergrößerung:

```
OpenGLWindow::resizeEvent()
OpenGLWindow::exposeEvent()
SceneView::paintGL(): Rendering to: 1222 x 891
OpenGLWindow::resizeEvent()
OpenGLWindow::exposeEvent()
SceneView::paintGL(): Rendering to: 1224 x 892
```

Ruft man stattdessen `renderLater()` auf, erhält man:

```
OpenGLWindow::resizeEvent()
OpenGLWindow::exposeEvent()
SceneView::paintGL(): Rendering to: 1283 x 910
SceneView::paintGL(): Rendering to: 1283 x 910
OpenGLWindow::resizeEvent()
OpenGLWindow::exposeEvent()
SceneView::paintGL(): Rendering to: 1288 x 912
SceneView::paintGL(): Rendering to: 1288 x 912
```

Wie man sieht, wird jedes Mal doppelt gezeichnet, was eine deutlich spürbare Verzögerung bedeutet. Grundsätzlich hilf es zu wissen, dass:

- beim ersten Anzeigen eines Fensters immer erst ein `ResizeEvent`, gefolgt von einem `ExposeEvent` geschickt wird
- beim Größenändern eines Fensters ebenfalls immer ein `ResizeEvent`, gefolgt von einem `ExposeEvent` geschickt wird
- beim Minimieren und Maximieren eines Fensters nur je ein (oder auf dem Mac mehrere) `ExposeEvent` geschickt werden. Dies kann man nutzen, um eine Animation zu stoppen und beim erneuten Anzeigen (`isExposed() == true`) wieder zu starten. Dies ist aber nicht der Fokus in diesem Tutorial. Daher könnte man auch das `ExposeEvent` komplett ignorieren und `renderNow()` direkt am Ende von `OpenGLWindow::resizeEvent()` aufrufen. So wie es aktuell implementiert ist, wird beim Minimieren und Maximieren mehrfach `ExposeEvent` mit `isExposed() == true` aufgerufen und damit wird mehrfach gezeichnet, trotz unverändertem Viewport und unveränderte Szene. Das ist aber nicht weiter bemerkbar und verschmerzbar.

5.3.3. Verwendung der Klasse *SceneView*

Die Klasse `SceneView` wird als `QWindow`-basierte Klasse selbst via Widget-Container in den Testdialog eingebettet (siehe *Tutorial 03*).

Bei der Analyse des Tutorialquelltextes kann man sich von außen nach innen "arbeiten":

- `main.cpp` - Instanziert `TestDialog`
- `TestDialog.cpp` - Instanziert `SceneView` und bettet das Objekt via Window-Container ein.

Es gibt im Quelltext von `TestDialog.cpp` nur ein neues Feature: Antialiasing (siehe letzter Abschnitt "Antialiasing" dieses Tutorials).

5.3.4. Implementierung der Klasse *SceneView*

Und da wären wir auch schon bei der Implementierung des Klasse `SceneView`.

Im Konstruktor werden letztlich 3 Dinge gemacht:

- dem Tastatur/Maus-Eingabemanager werden die für uns interessanten Tasten mitgeteilt, siehe Abschnitt "Tastatur- und Mauseingabe"
- die beiden ShaderProgramm-Container Objekte werden erstellt und konfiguriert, siehe Abschnitt "Shaderprogramme"
- die Kamera- und Welttransformationsmatrizen werden auf ein paar Standardwerte eingestellt, siehe Abschnitt "Transformationsmatrizen"

SceneView.cpp, Konstruktor

```
SceneView::SceneView() :
    m_inputEventReceived(false)
{
    // tell keyboard handler to monitor certain keys
    m_keyboardMouseHandler.addRecognizedKey(Qt::Key_W);
    m_keyboardMouseHandler.addRecognizedKey(Qt::Key_A);
    m_keyboardMouseHandler.addRecognizedKey(Qt::Key_S);
    m_keyboardMouseHandler.addRecognizedKey(Qt::Key_D);
    m_keyboardMouseHandler.addRecognizedKey(Qt::Key_Q);
    m_keyboardMouseHandler.addRecognizedKey(Qt::Key_E);
    m_keyboardMouseHandler.addRecognizedKey(Qt::Key_Shift);

    // *** create scene (no OpenGL calls are being issued below, just the data structures are created.

    // Shaderprogram #0 : regular geometry (painting triangles via element index)
    ShaderProgram blocks(":/shaders/withWorldAndCamera.vert",":/shaders/simple.frag");
    blocks.m_uniformNames.append("worldToView");
    m_shaderPrograms.append( blocks );

    // Shaderprogram #1 : grid (painting grid lines)
    ShaderProgram grid(":/shaders/grid.vert",":/shaders/simple.frag");
    grid.m_uniformNames.append("worldToView"); // mat4
    grid.m_uniformNames.append("gridColor"); // vec3
    grid.m_uniformNames.append("backColor"); // vec3
    m_shaderPrograms.append( grid );

    // *** initialize camera placement and model placement in the world

    // move objects a little bit to the back of the scene (negative z coordinates = further back)
    m_transform.translate(0.0f, 0.0f, -5.0f);
    m_camera.translate(0,5,0);
    m_camera.rotate(-30, m_camera.right());
}
```

Im Konstruktor werden nur Eigenschaften für die Shaderprogramme festgelegt, die eigentliche Initialisierung (OpenGL-Aufrufe) findet in `initializeGL()` statt.

Im Destruktor der Klasse werden die OpenGL-Objekte wieder freigegeben:

SceneView.cpp, Destruktor

```
SceneView::~SceneView() {
    m_context->makeCurrent(this);

    for (ShaderProgram & p : m_shaderPrograms)
        p.destroy();

    m_boxObject.destroy();
    m_gridObject.destroy();
}
```

Wichtig ist hier, dass der OpenGL-Context für das aktuelle Fenster aktuell gesetzt wird (`m_context->makeCurrent(this)`). Damit können dann die OpenGL-Objekte freigegeben werden. Dies erfolgt in den `destroy()` Funktionen der Shaderprogramm-Wrapper-Klasse und Zeichen-Objekt-Wrapper-Klassen.

5.3.5. OpenGL-Initialisierung

Die eigentlich Initialisierung der OpenGL-Objekte (Shaderprogramme und Pufferobjekte) erfolgt in `initializeGL()`:

SceneView.cpp:initializeGL()

```
#define SHADER(x) m_shaderPrograms[x].shaderProgram()

void SceneView::initializeGL() {
    // initialize shader programs
    for (ShaderProgram & p : m_shaderPrograms)
        p.create();

    // tell OpenGL to show only faces whose normal vector points towards us
    glEnable(GL_CULL_FACE);
    // enable depth testing, important for the grid and for the drawing order of several objects
    glEnable(GL_DEPTH_TEST);

    // initialize drawable objects
    m_boxObject.create(SHADER(0));
    m_gridObject.create(SHADER(1));
}
```

Dank der Kapselung der Shaderprogramm-Initialisierung in der Klasse `ShaderProgram` und der Kapselung der Zeichenobjekt-spezifischen Initialisierung in den Objekten ist diese Funktion sehr viel übersichtlicher als in den bisherigen Tutorials.

Das Makro `SHADER(x)` wird verwendet, um bequem auf das `QOpenGLShaderProgram` Objekt in der Wrapper-Klasse zuzugreifen.

Die beiden `glXXX` Befehle in der Mitte der Funktion schalten zwei für 3D Szenen wichtige Funktionen ein:

- `GL_CULL_FACE` - Zeichne Flächen nicht, welche mit dem "Rücken" zu uns stehen
- `GL_DEPTH_TEST` - Führe beim Zeichnen der Fragmente einen Tiefentest durch, und verwerfe weiter hintenliegende Fragmente. Das ist wichtig dafür, dass die gezeichneten Boxen das dahinterliegende Gitter überdecken. Der dafür benötigte Tiefenpuffer wird über `QSurfaceFormat` konfiguriert (QSurfaceFormat::setDepthBufferSize()).

Die Funktion `glDepthFunc(GL_LESS)` muss nicht aufgerufen werden, da das bei OpenGL der Standard ist.

Man kann testweise mal das Flag `GL_DEPTH_TEST` nicht setzen - die etwas verwirrende Darstellung ist, nun ja, verwirrend.

5.4. Tastatur- und Mauseingabe

Qt stellt in `QWindow` und `QWidget` Ereignisbehandlungsroutinen für Tastatur- und Mauseingaben zur Verfügung. Die Deklaration dieser Funktion sind oben in der `SceneView` Klassendeklaration zu sehen.

Wenn man eine Taste auf der Tastatur drückt wird ein `QEvent::KeyPress` ausgelöst und die Memberfunktion `keyPressEvent(QKeyEvent *event)` aufgerufen. Das passiert auch, wenn man die Taste *gedrückt* hält. Unterscheiden kann man dieses durch Prüfen der Eigenschaft `AutoRepeat` (`QKeyEvent::isAutoRepeat()`).

Für die Navigation in einer 3D Umgebung hält man die Tasten (z.B. WASD oder ähnliche) längere Zeit gedrückt (d.h. über mehrere Frames hinweg). Man benötigt also einen Zustandsmanager, der sich den aktuellen Zustand der Tasten merkt.

Ein solcher "Inputmanager" hält intern also für jede (berücksichtigte) Taste einen Zustand:

- Nicht gedrückt
- Gerade gedrückt
- Wurde gedrückt

Letzterer ist eigentlich nur dann wichtig, wenn auf einzelne Tastendrücke reagiert werden soll, während eventuell eine aufwändige Neuzeichenroutine läuft.

5.4.1. Der Tastatur- und Maus-Zustandsmanager

Man könnte die gesamte Tastatur- und Mausbehandlung natürlich auch direkt in der Klasse `SceneView` implementieren, in der auch die Ereignisbehandlungsfunktionen aufgerufen werden. Es ist aber übersichtlicher, diese in der Klasse `KeyboardMouseHandler` zu kapseln.

Die Aufgabe dieser Klasse ist letztlich sich zu merken, welche Taste/Mausknopf gerade gedrückt ist. Die Implementierung der Klasse ist für das Tutorial eigentlich nicht so wichtig, vielleicht lohnt aber ein Blick auf die Klassendeklaration:

KeyboardMouseHandler.h

```
class KeyboardMouseHandler {
public:
    KeyboardMouseHandler();
    virtual ~KeyboardMouseHandler();

    // functions to manage known keys
    void addRecognizedKey(Qt::Key k);
    void clearRecognizedKeys();

    // event handler helpers
    void keyPressEvent(QKeyEvent *event);
    void keyReleaseEvent(QKeyEvent *event);
    void mousePressEvent(QMouseEvent *event);
    void mouseReleaseEvent(QMouseEvent *event);
    void wheelEvent(QWheelEvent *event);

    // state changing helper functions
    bool pressKey(Qt::Key k);
    bool releaseKey(Qt::Key k);
    bool pressButton(Qt::MouseButton btn, QPoint currentPos);
    bool releaseButton(Qt::MouseButton btn);

    // query functions
    bool keyDown(Qt::Key k) const;
    bool buttonDown(Qt::MouseButton btn) const;
    QPoint mouseDownPos() const { return m_mouseDownPos; }
    int wheelDelta() const;

    // state reset functions
    QPoint resetMouseDelta(const QPoint currentPos);
    int resetWheelDelta();
    void clearWasPressedKeyStates();

private:
    enum KeyStates {
        StateNotPressed,
        StateHeld,
        StateWasPressed
    };

    std::vector<Qt::Key>    m_keys;
    std::vector<KeyStates>  m_keyStates;

    KeyStates               m_leftButtonDown;
    KeyStates               m_middleButtonDown;
    KeyStates               m_rightButtonDown;

    QPoint                  m_mouseDownPos;

    int                     m_wheelDelta;
};
```

Eine `KeyboardMouseHandler`-Klasse wird nach der Erstellung durch Aufrufe von `addRecognizedKey()` konfiguriert (siehe Konstruktor der Klasse `SceneView`).

Für die Tastatur- und Maus-Ereignisbehandlungsroutinen gibt es passende Hilfsfunktionen, sodass man von den Event-Funktionen der eigenen View-Klasse einfach diese Hilfsfunktionen aufrufen kann. Die Zustandsänderungslogik (auch das Prüfen auf AutoRepeat) wird in diesen Funktionen gemacht. Bei bekannten Tasten wird der `QKeyEvent` oder `QMouseEvent` akzeptiert, sonst ignoriert.

Den Zustand einzelner Tasten kann man auch programmgesteuert durch die `pressXXX` und `releaseXXX` Funktionen ändern.

Danach kommen die Funktionen zum Abfragen des Zustands. Bei Tasten ist die Abfrage mit `keyDown()` oder `buttonDown()` recht klar (sowohl der Zustand "gerade gedrückt", als auch "gedrückt und wieder losgelassen" liefern hier `true` zurück).

Bei der Mausbewegung und Scroll-Rad muss immer die *Veränderung* zwischen zwei Abfragezeitpunkten angeschaut werden. Bei Verwendung einer Free-Mouse-Look-Taste (hier rechte Maustaste), wird beim Drücken dieser Taste die globale Cursorpostion abgelegt, welche über `mouseDownPos()` abgefragt werden kann. Bei Mouse-Wheel-Ereignissen werden die Drehstufen (Winkel/Ticks) addiert.

Wenn man diese Änderungen nun in eine Bewegung umwandelt, muss man diese nach dem Auslesen wieder zurücksetzen. Dies erfolgt mit den Funktionen `resetMouseDelta()` und `resetWheelDelta()`, welche beide die bislang erfassten Differenzen zurückliefern. Die const-Abfragefunktionen `mouseDownPos()` und `wheelDelta()` können also verwendet werden, um zu Testen, ob es eine Maus-/Scrollradbewegung gab. Und beim Anwender der Änderungen ruf man die `resetXXX()` Funktionen auf.

Zuletzt muss man die Funktion `clearWasPressedKeyStates()` nach Abfrage der Tasten aufrufen, um die "wurde gedrückt" Zustände wieder in den "Nicht gedrückt" Zustand zurückzusetzen.

Die Implementierung der Klasse ist recht einfach und selbsterklärend und muss hier nicht näher ausgeführt werden. Interessant ist die Verwendung der Klasse. Dazu müssen wir uns zunächst den Programmauflauf der Ereignisschleife und Auswertung der Tasteneingabe genauer anschauen.

5.4.2. Die Ereignisschleife und Tastatur-/Mausevents

Zwischen zwei Frames (also Aufrufen von `paintGL()`) läuft das Programm in der Ereignisschleife. Sobald eine Taste gedrückt oder losgelassen wird, ruft Qt die entsprechende Ereignisbehandlungsfunktion auf, d.h. `keyPressEvent()` bzw. `keyReleaseEvent()`. Ebenso werden bei Mausaktionen die entsprechenden Aktionen ausgelöst.

Die Aufrufe werden an die gleichnamigen Funktionen in Zustandsmanager (`KeyboardMouseHandler`) weitergereicht. Wenn die betreffende Taste dem Zustandsmanager bekannt ist, wird der aktuelle Zustand im Zustandsmanager entsprechend geändert.

Nun wird noch geprüft, ob die Taste eine Szenenveränderung (bspw. Kamerabewegung) bewirkt. Dies erfolgt in der Funktion `SceneView::checkInput()`.

SceneView.cpp:checkInput()

```
void SceneView::checkInput() {
    // trigger key held?
    if (m_keyboardMouseHandler.buttonDown(Qt::RightButton)) {
        // any of the interesting keys held?
        if (m_keyboardMouseHandler.keyDown(Qt::Key_W) ||
            m_keyboardMouseHandler.keyDown(Qt::Key_A) ||
            m_keyboardMouseHandler.keyDown(Qt::Key_S) ||
            m_keyboardMouseHandler.keyDown(Qt::Key_D) ||
            m_keyboardMouseHandler.keyDown(Qt::Key_Q) ||
            m_keyboardMouseHandler.keyDown(Qt::Key_E))
        {
            m_inputEventReceived = true;
            renderLater();
            return;
        }

        // has the mouse been moved?
        if (m_keyboardMouseHandler.mouseDownPos() != QCursor::pos()) {
            m_inputEventReceived = true;
            renderLater();
            return;
        }
    }
    // scroll-wheel turned?
    if (m_keyboardMouseHandler.wheelDelta() != 0) {
        m_inputEventReceived = true;
        renderLater();
        return;
    }
}
```

In dieser Funktion werden nun die Abfragefunktionen verwendet, d.h. der Zustand des Tastatur-/Maus-Zustandsmanagers wird nicht verändert. Auch ist zu beachten, dass die Abfrage nach dem Mausrad separat erfolgt.

Wird eine relevante Taste oder Mausbewegung erkannt, wird durch Aufruf von `renderLater()` ein Zeichenaufruf in die Event-Schleife eingereiht (kommt beim nächsten VSync) und das Flag `m_inputEventReceived` wird gesetzt dann geht die Kontrolle wieder zurück an die Ereignisschleife.

Es sollte wirklich nur neu gezeichnet werden, wenn dies durch Tastendruck- oder Mausbewegung notwendig wird. Dadurch, dass das `UpdateRequest` nur bei Bedarf gesendet wird, kann man ansonsten wild auf der Tastatur herumhämmern, ohne dass auch nur ein OpenGL-Befehl aufgerufen wird.

Es ist nun möglich, dass ein weiteres Tastaturereignis eintrifft, *bevor* das `UpdateRequest`-Ereignis eintritt. Bspw. könnte dies das `QEvent::KeyRelease`-Ereignis eines gerade zuvor eingetroffenen `QEvent::KeyPress`-Ereignisses derselben Taste sein. Deshalb wird der Zustand einer Taste beim `keyReleaseEvent()` auf "Wurde gedrückt" geändert, und nicht einfach wieder zurück auf "Nicht gedrückt". Sonst hätte man im Zustandsmanager keine Information mehr darüber, dass die Taste in diesem Frame kurz gedrückt wurde. Das ist zwar bei hohen Bildwiederholfrequenzen hinreichend unwahrscheinlich, kann aber bei sehr komplexen Szenen (bzw. schwacher Hardware) hilfreich sein.

5.4.3. Auswertung der Eingabe und Anpassung der Kameraposition- und Ausrichtung

Die eigentliche Auswertung der Tastenzustände und Bewegung der Kamera erfolgt am Anfang der `SceneView::paintGL()`-Funktion:

SceneView.cpp:paintGL()

```
void SceneView::paintGL() {
    // process input, i.e. check if any keys have been pressed
    if (m_inputEventReceived)
        processInput();

    ...
```

Da die Zeichenfunktion aus einer Vielzahl von Gründen aufgerufen werden kann, dient das Flag `m_inputEventReceived` dazu, nur dann die Eingaben auszuwerten, wenn es tatsächlich welche gab.

Der Zeitaufwand für die Auswertung der Eingaben ist nicht wirklich groß. Da aber einige Matrizenoperationen involviert sind, kann man sich die Arbeit auch sparen, daher das "dirty" Flag `m_inputEventReceived`.

Die Auswertung des Tastatur- und Mauszustandes erfolgt in der Funktion `SceneView::processInput()`:

SceneView:processInput()

```
void SceneView::processInput() {
    m_inputEventReceived = false;

    if (m_keyboardMouseHandler.buttonDown(Qt::RightButton)) {

        // Handle translations
        QVector3D translation;
        if (m_keyboardMouseHandler.keyDown(Qt::Key_W))      translation += m_camera.forward();
        if (m_keyboardMouseHandler.keyDown(Qt::Key_S))      translation -= m_camera.forward();
        if (m_keyboardMouseHandler.keyDown(Qt::Key_A))      translation -= m_camera.right();
        if (m_keyboardMouseHandler.keyDown(Qt::Key_D))      translation += m_camera.right();
        if (m_keyboardMouseHandler.keyDown(Qt::Key_Q))      translation -= m_camera.up();
        if (m_keyboardMouseHandler.keyDown(Qt::Key_E))      translation += m_camera.up();

        float transSpeed = 0.8f;
        if (m_keyboardMouseHandler.keyDown(Qt::Key_Shift))
            transSpeed = 0.1f;
        m_camera.translate(transSpeed * translation);

        // Handle rotations
        // get and reset mouse delta (pass current mouse cursor position)
        QPoint mouseDelta = m_keyboardMouseHandler.resetMouseDelta(QCursor::pos());
        static const float rotatationSpeed  = 0.4f;
        const QVector3D LocalUp(0.0f, 1.0f, 0.0f); // same as in Camera::up()
        m_camera.rotate(-rotatationSpeed * mouseDelta.x(), LocalUp);
        m_camera.rotate(-rotatationSpeed * mouseDelta.y(), m_camera.right());

        // finally, reset "WasPressed" key states
        m_keyboardMouseHandler.clearWasPressedKeyStates();
    }
    int wheelDelta = m_keyboardMouseHandler.resetWheelDelta();
    if (wheelDelta != 0) {
        float transSpeed = 8.f;
        if (m_keyboardMouseHandler.keyDown(Qt::Key_Shift))
            transSpeed = 0.8f;
        m_camera.translate(wheelDelta * transSpeed * m_camera.forward());
    }

    updateWorld2ViewMatrix();
}
```

Auch in dieser Funktion werden Bewegungen der Kamera durch Tastendrücke und Schwenker durch Mausbewegung unabhängig vom Scrollrad-Zoom behandelt. Am Ende der Funktion werden die Welt-zu-Perspektive-Transformationsmatrizen angepasst. Die relevanten Matrizen und auch das Kamera-Objekt (Klasse `Camera`) sind im Abschnitt "Transformationsmatrizen und Kamera" weiter unten beschrieben.

Die Bewegung der Kamera ist recht einfach nachvollziehbar - je nach gedrückter Taste wird eine Verschieberichtung auf den Vektor `translation` addiert. Der tatsächliche Verschiebevektor wird durch Multiplikation mit einer Geschwindigkeit `transSpeed` berechnet. Hier ist auch die "Verlangsamung-bei-Shift-Tastendruck"-eingebaut.

Die Geschwindigkeit ist hier als "Bewegung je Frame" zu verstehen, was bei stark veränderlichen Frameraten (z.B. bei komplexer Geometrie) zu einer variablen Fortbewegungsgeschwindigkeit führen kann. Hier kann man alternativ eine Zeitmessung einbauen und den Zeitabstand zwischen Abfragen des Eingabezustands in die Berechnung der Verschiebung einfließen lassen.

Die Drehung der Kamera hängt von der Mausbewegung ab. Hier wird die Funktion `resetMouseDelta()` aufgerufen, welche zwei Funktionen hat:

- die Bewegung der Maus seit dem Druck auf die rechte Maustaste bzw. seit letztem Aufruf von `resetMouseDelta()` wird zurückgeliefert, und
- `mouseDownPos` wird auf die aktuelle Maus-Cursorposition gesetzt (sodass beim nächsten Aufruf

Bei der Bewegung erfolgt die Neigung der Kamera um die x-Achse des lokalen Kamerakoordinatensystems (wird zurückgeliefert durch die Funktion `m_camera.right()`. Analog könnte man die Kamera auch um die lokale y-Achse der Kamera schwenken (wie in einem Flugsimulator üblich), dies führt aber zu recht beliebigen Ausrichtungen. Möchte man die Kamera eher parallel zum "Fußboden" halten, dann dreht man die Kamera um die y-Achse des Weltenkoordinatensystems (Vektor 0,1,0).

Am Ende des Tastaturabfrageteils werden noch die "wurde gedrückt"-Zustände zurückgesetzt.

Das Scrollrad soll in diesem Beispiel ein deutlich schnelleres Vorwärts- oder Rückwärtsbewegen durch die Szene ermöglichen. Deshalb werden die Mausradbewegungen mit größerer Verschiebegeschwindigkeit skaliert. Wie auch bei der Abfrage der Mausbewegung wird in der Funktion `resetWheelDelta()` der aktuell akkumulierte Scrollweg zurückgeliefert und intern im Zustandsmanager wieder auf 0 gesetzt.

5.4.4. Auf gedrückte Tasten reagieren

Wie oben erläutert wird das Neuzeichnen nur bei Registrieren eines Tastendrucks angefordert. Nehmen wir mal an, die rechte Maustaste ist gedrückt und die Vorwärtstaste W wird gedrückt gehalten. Dann sendet das Betriebssystem (bzw. Window-Manager) in regelmäßigen Abständen KeyPress-Events (z.B. 50 je Sekunde, je nach Einstellung). Diese sind dann als `AutoRepeat` gekennzeichnet und führen damit nicht zu einer Änderung im Eingabe-Zustandsmanager, aber zu einer erneuten Prüfung der Neuzeichnung (Aufruf von `checkInput()`). Und da eine Kamera-relevante Taste gedrückt gehalten ist, wird ein Neuzeichnen via `renderLater()` angefordert. Als Konsequenz ruckelt das Bild dann im Rythmus der Tastenwiederholrate… nicht sehr angenehm anzusehen.

Daher muss das Prüfen auf gedrückte Tasten regelmäßig, d.h. einmal pro Frame erfolgen. Und der geeignete Ort dafür ist das Ende der `paintGL()`-Funktion:

SceneView.cpp:paintGL()

```
void SceneView::paintGL() {
    ...

    checkInput();
}
```

Ganz zum Schluss wird nochmal auf eine Tasteneingabe geprüft und damit bei Bedarf ein `UpdateRequest` eingereiht.

Damit wäre die Tastatur- und Mauseingabe auch schon komplett.

5.5. Shaderprogramme

Die Verwaltung der Shaderprogramme macht Qt ja eigentlich schon durch die Klasse `QOpenGLShaderProgram`. Wenn man eine weitere Wrapper-Klasse außen herum packt, dann wird der Quelltext noch deutlich übersichtlicher. In der Deklaration der Wrapper-Klasse `ShaderProgram` findet man die gekapselte Qt Klasse wieder:

ShaderProgram.h

```
class ShaderProgram {
public:
    ShaderProgram();
    ShaderProgram(const QString & vertexShaderFilePath, const QString & fragmentShaderFilePath);

    void create();
    void destroy();

    QOpenGLShaderProgram * shaderProgram() { return m_program; }

    // paths to shader programs, used in create()
    QString     m_vertexShaderFilePath;
    QString     m_fragmentShaderFilePath;

    QStringList m_uniformNames; // uniform (variable) names
    QList<int>  m_uniformIDs;   // uniform IDs (resolved in create())

private:
    QOpenGLShaderProgram    *m_program;
};
```

Zur Verwaltung von Shaderprogrammen gehören auch die Variablen, die man dem Vertex- und/oder Fragment-Shaderprogramm übergeben möchte (siehe Shaderprogramme in Abschnitt "Zeichenobjekte"). Die Verwendung der Klasse sieht vor, dass man erst alle Eigenschaften setzt (Resourcen-Pfade zu den Shaderprogrammen, und die uniform-Namen im Vektor `m_uniformNames`). Dies wird im Konstruktor der `SceneView`-Klasse gemacht:

SceneView.cpp:SceneView()

```
SceneView::SceneView() :
    m_inputEventReceived(false)
{
    ...

    // Shaderprogram #0 : regular geometry (painting triangles via element index)
    ShaderProgram blocks(":/shaders/withWorldAndCamera.vert",":/shaders/simple.frag");
    blocks.m_uniformNames.append("worldToView");
    m_shaderPrograms.append( blocks );

    // Shaderprogram #1 : grid (painting grid lines)
    ShaderProgram grid(":/shaders/grid.vert",":/shaders/grid.frag");
    grid.m_uniformNames.append("worldToView"); // mat4
    grid.m_uniformNames.append("gridColor"); // vec3
    grid.m_uniformNames.append("backColor"); // vec3
    m_shaderPrograms.append( grid );

    ...
}
```

Die Konfiguration aller Shaderprogramme kann vor der eigentlichen OpenGL-Initialisierung erfolgen. Diese erfolgt für jedes Shaderprogramm beim Aufruf der Funktion `ShaderProgram::create()`. Die macht dann die eigentliche Initialisierung, die in den vorangegangenen Tutorials in der `initializeGL()` Funktion gemacht wurde:

ShaderProgram.cpp:create()

```
void ShaderProgram::create() {
    Q_ASSERT(m_program == nullptr);

    m_program = new QOpenGLShaderProgram();

    if (!m_program->addShaderFromSourceFile(QOpenGLShader::Vertex, m_vertexShaderFilePath))
        qDebug() << "Vertex shader errors:\n" << m_program->log();

    if (!m_program->addShaderFromSourceFile(QOpenGLShader::Fragment, m_fragmentShaderFilePath))
        qDebug() << "Fragment shader errors:\n" << m_program->log();

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

    m_uniformIDs.clear();
    for (const QString & uniformName : m_uniformNames)
        m_uniformIDs.append( m_program->uniformLocation(uniformName));
}
```

Dank der netten Hilfsfunktionen `QOpenGLShaderProgram::addShaderFromSourceFile()` und `QOpenGLShaderProgram::uniformLocation()` ist das auch recht übersichtlich. Die Fehlerbehandlung könnte noch besser sein, aber das kann man ja schnell nachrüsten.

Beim Aufruf von `QOpenGLShaderProgram::addShaderFromSourceFile()` ist das erste Argument zu beachten, welches den Typ des Shaderprogramms festlegt!

Die Funktion `uniformLocation()` sucht in beiden Shaderprogrammen nach `uniform` Deklarationen, also Variablen, die unabhängig von Vertex oder Fragment dem Shaderprogramm zur Verfügung stehen. Diese werden beim compilieren und linken durchnummeriert und den zu einem uniform-Variablennamen passenden Index kann man mit `uniformLocation()` ermitteln.

Bei der Verwendung des Shaders kann man dann mit setUniformValue() den entsprechenden Wert setzen (siehe auch Shaderprogramm-Beispiele im Abschnitt "Zeichenobjekte").

Die Shaderprogramme wissen selbst nicht, für welche Objekte sie zum Zeichnen gebraucht werden. Auch werden die Variablen (uniforms), die sie zur Funktion benötigen, meist woanders gespeichert. Daher gibt es in der Klasse nicht mehr zu tun.

5.6. Transformationsmatrizen und Kamera

5.6.1. Transformationen

Das Thema *Transformationsmatrizen* ist in den in der Einleitung zitierten Webtutorials/Anleitungen ausreichend beschrieben. Die Format zur Transformation eines Punktes/Vektors `pModel` in den Modellkoordinaten zu den View-Koordinaten `pView` benötigt 3 Transformationsmatrizen:

```
pView = M_projection * M_World2Camera * M_Model2World * pModel
```

Dies entspricht den Schritten:

1. Transformation des Punktes von Modellkoordinaten in das Weltenkoordinatensystem. Dies ist bei

bewegten/animierten Objekten sinnvoll, d.h. eine Objekteigenschaft. Manchmal möchte man auch die gesamte Welt transformieren, auch dafür nimmt man die Model-zu-Welt-Transformationsmatrix.

2. Transformation von Welt- zu Beobachterkoordinatensystem (Kamera). Ist eigentlich das Gleiche, jedoch ist die Kamera, deren Ausrichtung und Position modellunabhängig.
3. Projektionstransformation (othogonal, perspektivisch, ...), kann z.B. durch near/far-plane und Angle-of-View definiert werden.

Da die Objekte in Modell bzw. Weltkoordinaten definiert und verwaltet werden, sollte besser OpenGL die Transformationen durchführen (dafür ist es ja gemacht). Je nach Anzahl der zu transformierenden Objekte kann nun den objektspezifischen ersten Transformationsschritt in das Weltenkoordinatensystem auf der CPU durchführen (idealerweise parallelisiert). Die Transformation von Weltkoordinaten in die projezierte Darstellung macht dann OpenGL. Da diese Matrix für *alle* Objekte gleich ist, kann man diese auch bequem den Shaderprogrammen übergeben. D.h. die Matrix:

```
M_World2View = M_Projection * M_World2Camera * M_Model2World
```

wird als uniform-Variable an die Shaderprogramme übergeben. Die Transformieren dann damit hocheffizient auf der Grafikkarte alle Vertex-Koordinaten.

5.6.2. Aktualisierung der World2View Matrix

Die Projektionsmatrix ändert sich bei jeder Viewport-Änderung, da sich damit zumeist das Breite/Höhe-Verhältnis ändert. Sonst ändert sich diese Matrix eigentlich nie, außer vielleicht in den Benutzereinstellungen (wenn z.B. Linseneigenschaften wie Öffnungswinkel oder Zoom verändert werden).

Die Model2World-Matrix bleibt wie oben geschrieben außen vor, da objektabhängig.

Die Kameramatrix (World2Camera) ändert sich jedoch ständig während der Navigation durch die Szene. Da die Navigation am Anfang der Neuzeichenroutine ausgewertet wird, erfolgt die Neuberechnung der Matrix (falls notwendig) auch direkt vorm Neuzeichnen.

Es ist denkbar, dass ein MouseMove-Event mehrfach während eines Frames ausgelöst wird. Wenn man nun die Neuberechnung der Matrix daran koppelt, führt das mitunter zu unnützer Rechenarbeit. Daher ist es sinnvoller, die Berechnung erst zu Beginn des Zeichenzyklus durchzuführen.

Die eigentliche Berechnung erfolgt in der Funktion `updateWorld2ViewMatrix`. Dank der Funktionalität der Matrixklasse `QMatrix4x4` eine sehr kompakte Funktion.

```
void SceneView::updateWorld2ViewMatrix() {
    // transformation steps:
    //   model space -> transform -> world space
    //   world space -> camera/eye -> camera view
    //   camera view -> projection -> normalized device coordinates (NDC)
    m_worldToView = m_projection * m_camera.toMatrix() * m_transform.toMatrix();
}
```

Die Multiplikation mit der Modell-Transformationsmatrix (`m_transform`) ist eigentlich nicht zwingend notwendig, dient aber der Demonstration der Animationsfähigkeit (konstantes Rotieren der Welt um die y-Achse). Dazu den `#if 0` Block in `paintGL()` nach `#if 1` ändern.

Die ganze Arbeit der Konfiguration und Erstellung der Translations, Rotations, und Skalierungsmatrizen macht die Klasse `Transform3D`. In der Funktion `toMatrix()` werden diese einzelnen Matrizen zur Gesamtmatrix kombiniert (implementiert mit Lazy-Evaluation):

Transform3D.cpp:toMatrix()

```
const QMatrix4x4 &Transform3D::toMatrix() const {
    if (m_dirty) {
        m_dirty = false;
        m_world.setToIdentity();
        m_world.translate(m_translation);
        m_world.rotate(m_rotation);
        m_world.scale(m_scale);
    }
    return m_world;
}
```

Die Kamera-Klasse ist davon abgeleitet und beinhaltet letztlich nur die inverse Transformation vom Welten- zum Beobachterkoordinatensystem (siehe auch https://www.trentreed.net/blog/qt5-opengl-part-3b-camera-control). Im Prinzip hilft es sich vorzustellen, dass die Kamera ein positioniertes und ausgerichtetes Objekt selbst ist. Nun wollen wir dieses Kamera-Objekt nicht mittels einer Model2World-Transformationsmatrix in das Weltenkoordinatensystem hieven, sondern uns eher aus der Weltsicht in die lokale Sicht des Kamera-Objekts bewegen. Dies bedeuted, wir müssen alle Weltkoordinaten mittels der Inversen der Kamera-Objekt-Model2World-Matrix multiplizieren. Das macht dann die entsprechend spezialisiert `toMatrix()`-Funktion:

Camera.h:toMatrix()

```
const QMatrix4x4 & toMatrix() const {
    if (m_dirty) {
        m_dirty = false;
        m_world.setToIdentity();
        m_world.rotate(m_rotation.conjugated());
        m_world.translate(-m_translation);
    }
    return m_world;
}
```

Daneben bietet die Kameraklasse noch 3 interessante Abfragefunktionen, welche die Koordinatenrichtungen des lokalen Kamera-Koordinatensystems im Weltenkoordinatensystem zurückliefern:

Camera.h

```
// negative Kamera-z-Achse
QVector3D forward() const {
    const QVector3D LocalForward(0.0f, 0.0f, -1.0f);
    return m_rotation.rotatedVector(LocalForward);
}

// Kamera-y-Achse
QVector3D up() const {
    const QVector3D LocalUp(0.0f, 1.0f, 0.0f);
    return m_rotation.rotatedVector(LocalUp);
}

// Kamera-x-Achse
QVector3D right() const {
    const QVector3D LocalRight(1.0f, 0.0f, 0.0f);
    return m_rotation.rotatedVector(LocalRight);
}
```

Die eigentliche Arbeit macht hier die Klasse `QQuaternion`, welche man dankenswerterweise nicht selbst implementieren muss.

5.7. Zeichenobjekte

In diesem Abschnitt geht es um die Verwaltung von Zeichenobjekten. Dies ist nicht wirklich ein Qt-Thema, da diese Art von Datenmanagement in der einen oder anderen Art in jeder OpenGL-Anwendung zu finden ist. Wen also nur die Qt-spezifischen Dinge interessieren, kann dieses Kapitel gerne überspringen.

5.7.1. Effizientes Zeichnen großer Geometrien

Es gibt eine wesentliche Grundregel in OpenGL:

Wenn man effizient große Geometrien zeichnen möchte, dann muss man die Anzahl der `glDrawXXX` Aufrufe so klein wie möglich halten.

Ein Beispiel: wenn man **2** Würfel zeichen möchte, hat man folgende Möglichkeiten:

- alle 12 Seiten einzeln Zeichen (12 `glDrawXXX` Aufrufe), z.B. als:
  - `GL_TRIANGLES` (6 Vertices je Seite)
  - `GL_TRIANGLE_STRIP` (4 Vertices je Seite)
  - `GL_QUADS` (4 Vertices je Seite)
- jeden Würfel einzeln zeichnen (2 `glDrawXXX` Aufrufe), dabei alle Seiten des Würfels zusammen zeichnen via:
  - `GL_TRIANGLES` (8 Vertices, 6*6 Elementindices)
  - `GL_QUADS` (8 Vertices, 6*4 Elementindices)
- beide Würfel zusammen zeichnen (1 `glDrawXXX` Aufruf), dabei alle Seiten beider Würfels zusammen zeichnen via:
  - `GL_TRIANGLES` (2*8 Vertices, 2*6*6 Elementindices)
  - `GL_QUADS` (2*8 Vertices, 2*6*4 Elementindices)

Die oben angegebene Anzahl der Vertexes gilt natürlich nur für einfarbige Würfel. Sollen die Seitenflächen unterschiedlich gefärbt sein, braucht man natürlich für jede Seite 4 Vertices, also bspw. bei `GL_TRIANGLES` brauch man für die 2 Würfel 2*6*4 Vertices.

Wenn man Objekte mit gemischten Flächenprimitiven hat (also z.B. Dreiecke und Rechtecke, oder Polygone), dann kann man entweder nach Flächentyp zusammenfassen und je Flächentyp ein `glDrawXXX` Aufruf ausführen, oder eben alles als Dreiecke behandeln und nur einen Zeichenaufruf verwenden. Kann man mal durch Profiling ausprobieren, was dann schneller ist. Der Speicherverbrauch spielt auch eine Rolle, da der Datentransfer zwischen CPU und GPU immer auch an der Geschwindigkeit der Speicheranbindung hängt.

Die Gruppierung von Zeichenelementen erfolgt im Prinzip nach folgenden Kriterien:

- Vertexdaten bei interleaved Storage (z.B. nur Koordinaten wie beim Gitter unten, Koordinaten-und-Farben, Koordinaten-Normalen-Texturcoords-Farben)
- Geometrietyp (siehe oben)
- Objektveränderlichkeit

Das Ganze hängt also stark von der Anwendung ab. Im *Tutorial 05* gibt es zwei Arten von Objekten:

- das Gitter, bestehend aus Linien und ausschließlich Koordinaten, gezeichnet via `GL_LINES`
- die Boxen, mit `GL_TRIANGLES` gezeichnet.

5.7.2. Verwaltung von Zeichenobjekten

Eine Möglichkeit, die für das Zeichnen derart gruppierter Daten benötigten Objekte, d.h. VertexArrayObject (VAO), VertexBufferObject (VBO) und ElementBufferObject (EBO), zu verwalten, ist eigene Datenhalteklassen zu verwenden. Diese sehen allgemein so aus:

Deklaration einer Zeichenobjektklasse

```
class DrawObject {
public:
    DrawObject();

    // create native OpenGL objects
    void create(QOpenGLShaderProgram * shaderProgramm);
    // release native OpenGL objects
    void destroy();

    // actual render objects
    void render();

    // Data members to store state
    ....

    QOpenGLVertexArrayObject    m_vao;
    QOpenGLBuffer               m_vbo; // Vertex buffer
    QOpenGLBuffer               m_ebo; // Element/index buffer

    // other buffer objects

    ....
};
```

Die drei wichtigen Lebenszyklusphasen der Objekte sind durch die Funktionen `create()`, `destroy()` und `render()`

abgebildet.

Speichermanagement bei OpenGL Objekten sollte explizit erfolgen, und nicht im Destruktor der Zeichenobjekt-Klassen. Es ist beim Aufräumen im Destruktor durch die automatisiert generierte Aufrufreihenfolge der einzelnen Destruktoren schwierig sicherzustellen, dass der dazugehörige OpenGL-Kontext aktiv ist. Daher empfiehlt es sich, stets eine explizite `destroy()` Funktion zu verwenden.

Außerdem werden die Zeichenobjekte so kopierbar und können, unter anderem, in `std::vector` oder ähnlichen Container verwendet werden.

Am Besten wird das Datenmanagement in einer Beispielimplementierung sichtbar.

5.7.3. Zeichenobjekt #1: Gitterraster in X-Z Ebene

Beginnen wir mit einem einfachen Beispiel: Ein Gitterraster soll auf dem Bildschirm gezeichnet werden, sozusagen als "Boden". Es werden also Linien in der X-Z-Ebene (y=0) gezeichnet, wofür der Elementtyp `GL_LINES` zum Zeichnen verwendet wird.

Für jede Linie sind Start- und Endkoordinaten anzugeben, wobei die y-Koordinate eingespart werden kann.

Man muss nicht immer alle Koordinaten (x,y,z) an den Vertexshader übergeben, wenn es nicht notwendig ist.

Wir stellen also den Vertexpuffer mit folgendem Schema zusammen:

`x1sz1sx1ez1ex2sz2sx2ez2e...` also jeweils x und z Koordinatentuple für je Start- (s) und Endpunkt (e) einer Linie nacheinander.

Diese Geometrieinformation wird in der Klasse `GridObject` zusammengestellt:

GridObject.h, Klassendeklaration

```
class GridObject {
public:
    void create(QOpenGLShaderProgram * shaderProgramm);
    void destroy();

    void render();

    unsigned int                m_bufferSize;
    QOpenGLVertexArrayObject    m_vao;
    QOpenGLBuffer               m_vbo;
};
```

Die Implementierung der `create()` Funktion ist das eigentlich Interessante:

GridObject.cpp:create()

```
void GridObject::create(QOpenGLShaderProgram * shaderProgramm) {
    const unsigned int N = 100; // number of lines to draw in x and z direction
    // width is in "space units", whatever that means for you (meters, km, nanometers...)
    float width = 500;
    // grid is centered around origin, and expands to width/2 in -x, +x, -z and +z direction

    // create a temporary buffer that will contain the x-z coordinates of all grid lines
    std::vector<float>          gridVertexBufferData;
    // we have 2*N lines, each line requires two vertexes, with two floats (x and z coordinates) each.
    m_bufferSize = 2*N*2;
    gridVertexBufferData.resize(m_bufferSize);
    float * gridVertexBufferPtr = gridVertexBufferData.data();
    // compute grid lines with z = const
    float x1 = -width*0.5;
    float x2 = width*0.5;
    for (unsigned int i=0; i<N; ++i, gridVertexBufferPtr += 4) {
        float z = width/(N-1)*i-width*0.5;
        gridVertexBufferPtr[0] = x1;
        gridVertexBufferPtr[1] = z;
        gridVertexBufferPtr[2] = x2;
        gridVertexBufferPtr[3] = z;
    }
    // compute grid lines with x = const
    float z1 = -width*0.5;
    float z2 = width*0.5;
    for (unsigned int i=0; i<N; ++i, gridVertexBufferPtr += 4) {
        float x = width/(N-1)*i-width*0.5;
        gridVertexBufferPtr[0] = x;
        gridVertexBufferPtr[1] = z1;
        gridVertexBufferPtr[2] = x;
        gridVertexBufferPtr[3] = z2;
    }
```

Im ersten Teil wird ein linearer Speicherbereich (bereitgestellt in einem `std::vector`) mit den Liniendaten gefüllt. Das Raster besteht aus Linien in X und Z Richtung (2), jeweils N Linien, und jede Linie hat einen Start- und einen Endpunkt (2) und jeder Punkt besteht aus 2 Koordinaten. Dies macht 2*N*2*2 floats (=NVertices).

Es ist ok an dieser Stelle den Speicherbereich in einem temporären Vektor anzulegen, da beim Erzeugen des OpenGL-Vertexpuffers die Daten kopiert werden und der Vektor danach nicht mehr benötigt wird. Dies ist im Falle von veränderlichen Daten (siehe BoxObjekte unten) anders.

Im zweiten Teil der Funktion werden dann wie gehabt die OpenGL-Pufferobjekte erstellt:

GridObject.cpp:create(), fortgesetzt

```
    // Create Vertex Array Object
    m_vao.create();     // create Vertex Array Object
    m_vao.bind();       // and bind it

    // Create Vertex Buffer Object
    m_vbo.create();
    m_vbo.bind();
    m_vbo.setUsagePattern(QOpenGLBuffer::StaticDraw);
    int vertexMemSize = m_bufferSize*sizeof(float);
    m_vbo.allocate(gridVertexBufferData.data(), vertexMemSize);

    // layout(location = 0) = vec2 position
    shaderProgramm->enableAttributeArray(0); // array with index/id 0
    shaderProgramm->setAttributeBuffer(0, GL_FLOAT,
                                0 /* position/vertex offset */,
                                2 /* two floats per position = vec2 */,
                                0 /* vertex after vertex, no interleaving */);

    m_vao.release();
    m_vbo.release();
}
```

Die Aufrufe von `shaderProgramm->enableAttributeArray` und `shaderProgramm->setAttributeBuffer` definieren, wie der Vertexshader auf diesen Speicherbereich zugreifen soll. Deshalb muss die Funktion `create()` auch das dazugehörige Shaderprogramm als Funktionsargument erhalten.

Nachdem nun die Puffer erstellt und konfiguriert wurden, ist der Rest der Klassenimplementierung recht übersichtlich:

GridObject.cpp:destroy() und render()

```
void GridObject::destroy() {
    m_vao.destroy();
    m_vbo.destroy();
}


void GridObject::render() {
    m_vao.bind();
    // draw the grid lines, m_bufferSize = number of floats in buffer
    glDrawArrays(GL_LINES, 0, m_bufferSize);
    m_vao.release();
}
```

Die Funktion `destroy()` ist sicher selbsterklärend. Und die Render-Funktion ebenso.

Beachte, dass die Funktion `glDrawArrays()` als drittes Argument die Länge des Puffers als Anzahl der Elemente vom Typ des Puffers (hier GL_FLOAT) erwartet, und *nicht* die Länge in Bytes.

Die Funktion `render()` wird direkt aus `SceneView::paintGL()` aufgerufen. Hier ist der entsprechende Abschnitt aus der Funktion:

SceneView.cpp:paintGL()

```
void SceneView::paintGL() {
    ...

    // set the background color = clear color
    QVector3D backColor(0.1f, 0.15f, 0.3f);
    glClearColor(0.1f, 0.15f, 0.3f, 1.0f);

    QVector3D gridColor(0.5f, 0.5f, 0.7f);

    ...

    // *** render grid ***

    SHADER(1)->bind();
    SHADER(1)->setUniformValue(m_shaderPrograms[1].m_uniformIDs[0], m_worldToView);
    SHADER(1)->setUniformValue(m_shaderPrograms[1].m_uniformIDs[1], gridColor);
    SHADER(1)->setUniformValue(m_shaderPrograms[1].m_uniformIDs[2], backColor);
    m_gridObject.render(); // render the grid
    SHADER(1)->release();

    ...
```

Hier sieht man auch, wie die Variablen an die Shaderprogramme übergeben werden. In Abschnitt "Shaderprogramme" oben wurde ja gezeigt, wie die IDs der `uniform` Variablen ermittelt werden. Nun müssen diese Variablen *vor jeder Verwendung* des Shaderprogramms gesetzt werden. Dies erfolgt direkt vor dem Aufruf der `GridObject::render()` Funktion.

Das Ergebnis dieses Zeichnens (mit uniformer Gitterfarbe) ist zunächst ganz nett:

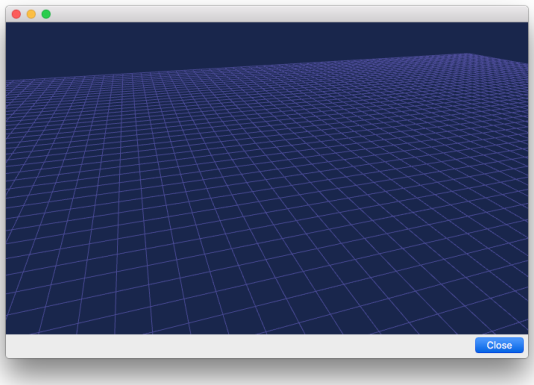

Figure 6. Einfaches Gitterraster (einfarbig) mit sichtbarer endlicher Ausdehnung

Aber schöne wäre es, wenn das Gitter mit zunehmender Tiefe verblasst.

Gitter mit Abblendung in der Tiefe

Das Gitter sollte sich nun in weiter Ferne der Hintergrundfarbe annähern. Man könnte das zum Beispiel erreichen, wenn man die Farbe des Gitters an weiter entfernten Punkte einfärbt.

Den Vertexshader könnte man wie folgt erweitern:

```
#version 330

// GLSL version 3.3
// vertex shader

layout(location = 0) in vec2 position; // input:  attribute with index '0'
                                       //         with 2 floats (x, z coords) per vertex
out vec4 fragColor;                    // output: computed vertex color for shader

const float FARPLANE = 50;             // threshold
float fragDepth;                       // normalized depth value

uniform mat4 worldToView;              // parameter: the view transformation matrix
uniform vec3 gridColor;                // parameter: grid color as rgb triple
uniform vec3 backColor;                // parameter: background color as rgb triple

void main() {
  gl_Position = worldToView * vec4(position.x, 0.0, position.y, 1.0);
  fragDepth = max(0, min(1, gl_Position.z / FARPLANE));
  fragColor = vec4( mix(gridColor, backColor, fragDepth), 1.0);
}
```

Es gibt 3 Parameter, die dem Shaderprogramm gegeben werden müssen (das passiert in `SceneView::paintGL()`, siehe Quelltextausschnitt oben):

- `worldToView` - Transformationsmatrix (von Weltkoordinaten zur perspektivischen Ansicht)
- `gridColor` - Farbe des Gitters
- `backColor` - Hintergrundfarbe

Die Variable `gl_Position` enthält nach der Transformation die normalisierten Koordinaten. In der Berechnung wird die zweite Komponente des Vertex-Vektors (angesprochen über `.y`) als z-Koordinate verwendet.

Für die Abblendefunktionalität ist die Entfernung des Linienstart- bzw. -endpunktes interessant. Nun sind die z-Koordinaten dieser normalisierten Position alle sehr dicht an 1 dran. Deshalb werden sie noch skaliert (entsprechend der perspektivischen Transformationsregeln etwas wie eine Farplane). Nun kann man diese Tiefe, gespeichert in der Variable `fragDepth` nutzen, um zwischen Gitterfarbe und Hintergrundfarbe linear mit der GLSL-Funktion `mix()` zu interpolieren.

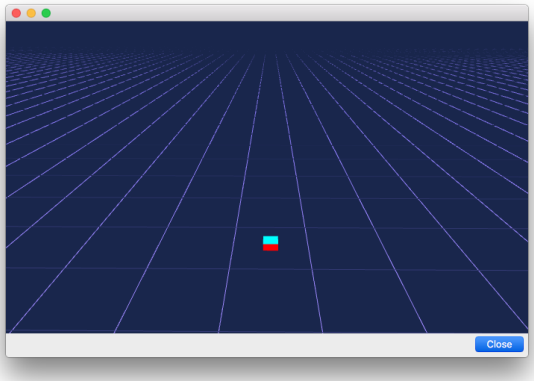

Figure 7. Gitterraster mit Vertex-basierter Abblendung

Das Ergebnis geht schon in die richtige Richtung, aber es gibt einen unschönen Effekt, wenn man parallel zu den Linien schaut. Die Koordinaten der Endpunkte der seitlich laufenden Linien sind sehr weit weg (in der

perspektivischen Projekten), sodass beide Linienenden nahezu Hintergrundfarbe bekommen. Und da die Fragmentfarbe eine lineare Interpolation zwischen den Vertexfarben ist, verschwindet die gesamte Linie.

Das Problem lässt sich nur beheben, wenn man die Ablendfunktionalität in den Fragment-Shader steckt.

Der Vertex-Shader wird dadurch total einfach:

grid.vert (Vertexshader)

```
#version 330

// GLSL version 3.3
// vertex shader

layout(location = 0) in vec2 position; // input:  attribute with index '0'
                                       //         with 2 floats (x, z coords) per vertex

uniform mat4 worldToView;              // parameter: world to view transformation matrix

void main() {
  gl_Position = worldToView * vec4(position.x, 0.0, position.y, 1.0);
}
```

Letztlich werden nur noch die Vertex-Koordinaten transformiert und an den Fragment-Shader weitergereicht. Der sieht dann so aus:

grid.frag (Fragmentshader)

```
#version 330

out vec4 fColor;

uniform vec3 gridColor;                // parameter: grid color as rgb triple
uniform vec3 backColor;                // parameter: background color as rgb triple
const float FARPLANE = 150;            // threshold

void main() {
  float distanceFromCamera = (gl_FragCoord.z / gl_FragCoord.w) / FARPLANE;
  distanceFromCamera = max(0, min(1, distanceFromCamera)); // clip to valid value range
  fColor = vec4( mix(gridColor, backColor, distanceFromCamera), 1.0 );
}
```

Die Variable `gl_FragCoord` wird für jeden einzelnen Bildpunkt von OpenGL bereitgestellt und enthält die Normalized Device Coordinates (NDC). Wenn man beachtet, dass diese Koordinaten durch Division mit w berechnet werden, dann bekommt man die originale z-Koordinate durch Multiplikation mit w. Das ganze wird dann noch mit einem Begrenzungswert (`FARPLANE`) skaliert. Falls bei der Definition des View-Frustums andere Werte für Near/Farplane verwendet werden, muss man die Formel entsprechend anpassen.

Damit sieht das Ergebnis dann wie gewünscht aus:

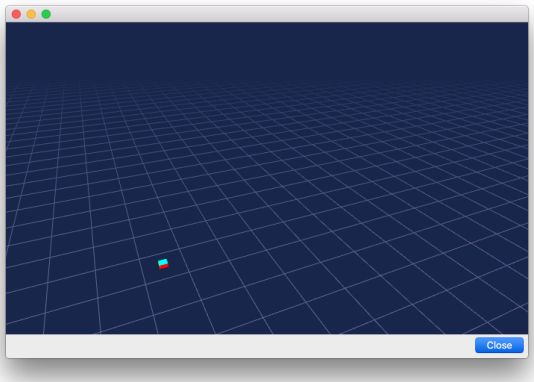

Figure 8. Gitterraster mit Fragment-basierter Abblendung (Fog/Nebeleffekt)

5.7.4. Zeichenobjekt #2: Viele viele Boxen

Um die Performance der Grafikkarte (und der Anwendung) zu testen, kann man sehr viele Boxen modellieren und dann mittels eines einzigen `glDrawElements()`-Aufrufs zeichnen lassen. Bei modernen Grafikkarten sollten locker Millionen von Boxen flüssig gezeichnet werden können.

Die Aufgabe besteht nun darin, die Vertexdaten aller Boxen und die dazugehörigen Elementindexe in die zwei Puffer (VBO und EBO) zu stecken, und den Quelltext auch noch einigermaßen verstehen zu können.

Zunächst wird wie beim Gitter ein Boxen-Zeichenobjekt erstellt:

BoxObject.h

```
class BoxObject {
public:
    BoxObject();

    void create(QOpenGLShaderProgram * shaderProgramm);
    void destroy();

    void render();

    std::vector<BoxMesh>        m_boxes;

    std::vector<Vertex>         m_vertexBufferData;
    std::vector<GLuint>         m_elementBufferData;

    QOpenGLVertexArrayObject    m_vao;
    QOpenGLBuffer               m_vbo;
    QOpenGLBuffer               m_ebo;
};
```

Sieht erstmal fast genauso aus wie bei der Klasse `GridObject`.

Beide Klassen stellen ja die gleichen Funktionen zur Verfügung. Man könnte also auf die Idee kommen, hinsichtlich Initialisierung und Aufräumen alle Zeichenobjekte gleich zu behandeln. Geht sicher, hängt aber vom Programm ab (und der Datenveränderlichkeit), ob das sinnvoll ist. Beim *Tutorial 05* wäre das sicher gut gewesen (hab ich mir aber wegen nur zwei Objekten gespart).

Vielleicht noch ein Hinweis zu den Puffern. Neben OpenGL-Pufferobjekten `m_vbo` und `m_ebo` sind die ursprünglichen Datenpuffer `m_vertexBufferData` und `m_elementBufferData` dauerhaft als Membervariablen vorhanden. Dies ermöglicht eine nachträgliche Aktualisierung eines Teils der Daten (z.B. Farben einer einzelnen Box oder einer Seite), ohne dass neu Speicher reserviert werden muss und die Puffer erneut aufgebaut werden.

Teilweise Aktualisierung von Pufferdaten spielt in diesem Tutorial keine Rolle. Es lohnt sich aber, die Funktion QOpenGLBuffer::mapRange anzuschauen (bzw. die darunterliegenden nativen OpenGL-Funktionen `glMapBuffer` und `glMapBufferRange`).

Die eigentliche Geometrie, d.h. Größe und Position der Boxen wird durch die `BoxMesh`-Objekte bereitgestellt, welche im Vektor `m_boxes` vorgehalten werden.

Die Implementierung der 3 Funktionen ist dann auch recht ähnlich wie beim `GridObject`.

BoxObject.cpp:destroy() und render()

```
void BoxObject::destroy() {
    m_vao.destroy();
    m_vbo.destroy();
    m_ebo.destroy();
}


void BoxObject::render() {
    m_vao.bind();
    glDrawElements(GL_TRIANGLES, m_elementBufferData.size(), GL_UNSIGNED_INT, nullptr);
    m_vao.release();
}
```

Die Funktionen `destroy()` und `render()` sind selbsterklärend (wie schon beim `GridObject`. Zur Vollständigkeit sei nocheinmal der Aufruf der Zeichenfunktion gezeigt:

SceneView.cpp:paintGL()

```
void SceneView::paintGL() {
    ...

    // *** render boxes
    SHADER(0)->bind();
    SHADER(0)->setUniformValue(m_shaderPrograms[0].m_uniformIDs[0], m_worldToView);
    m_boxObject.render(); // render the boxes
    SHADER(0)->release();

    ...
```

Erstellung der OpenGL-Puffer - struct *Vertex*

Interessanter ist dann schon die `create()`-Funktion, in der die Puffer befüllt werden:

BoxObject.cpp:create()

```
void BoxObject::create(QOpenGLShaderProgram * shaderProgramm) {
    // create and bind Vertex Array Object
    m_vao.create();
    m_vao.bind();

    // create and bind vertex buffer
    m_vbo.create();
    m_vbo.bind();
    m_vbo.setUsagePattern(QOpenGLBuffer::StaticDraw);
    int vertexMemSize = m_vertexBufferData.size()*sizeof(Vertex);
    m_vbo.allocate(m_vertexBufferData.data(), vertexMemSize);

    // create and bind element buffer
    m_ebo.create();
    m_ebo.bind();
    m_ebo.setUsagePattern(QOpenGLBuffer::StaticDraw);
    int elementMemSize = m_elementBufferData.size()*sizeof(GLuint);
    m_ebo.allocate(m_elementBufferData.data(), elementMemSize);

    // set shader attributes

    // index 0 = position
    shaderProgramm->enableAttributeArray(0); // array with index/id 0
    shaderProgramm->setAttributeBuffer(0, GL_FLOAT, 0, 3, sizeof(Vertex));
    // index 1 = color
    shaderProgramm->enableAttributeArray(1); // array with index/id 1
    shaderProgramm->setAttributeBuffer(1, GL_FLOAT, offsetof(Vertex, r), 3, sizeof(Vertex));

    m_vao.release();
    m_vbo.release();
    m_ebo.release();
}
```

Die `create()`-Funktion ist inzwischen sicher gut verständlich (ansonsten siehe *Tutorial 03* und *Tutoral 04*):

1. das Vertex Array Objekt wird erstellt,
2. die Pufferobjekte werden erstellt und die Inhalte der bereits initialisierten Puffer (`m_vertexBufferData` und `m_elementBufferData` werden in die OpenGL-Puffer kopiert)
3. die Attribute im Shaderprogramm werden gesetzt, d.h. die Zusammensetzung des Puffers

Hier kommt das erste Mal die Struktur `Vertex` zum Einsatz. Diese gruppiert alle Attribute eines einzelen Vertex:

Vertex.h

```
struct Vertex {
    Vertex() {}
    Vertex(const QVector3D & coords, const QColor & col) :
        x(float(coords.x())),
        y(float(coords.y())),
        z(float(coords.z())),
        r(float(col.redF())),
        g(float(col.greenF())),
        b(float(col.blueF()))
    {
    }

    float x,y,z;
    float r,g,b;
};
```

Die Klasse enthält derzeit lediglich 6 floats, 3 für die Koordinaten, und 3 für das rgb-Farbtuple.

Beim Erstellen eines Puffers im *interleaved*-Modus werden nun die Vertex-Daten nacheinander in den Puffer kopiert (Details dazu im nächsten Abschnitt).

Dem Shaderprogramm muss man nun mitteilen, wo in diesem kontinuierlichen Speicherbereich die einzelen Attribute zu finden sind. Der `stride`-Parameter ist die Größe eines Vertex-Datemblocks in Bytes, welches `sizeof(Vertex)` zurückliefert. Das `offset` Argument (3. Argument in `setAttributeBuffer()`) ist die Anzahl der Bytes seit Beginn eines Vertexblocks, bei dem das jeweilige Datenelement beginnt. Im Fall des rgb-Farbtuples beginnt dieser Speicherbereich bei dem float `r`, und das passende Byte-Offset liefert `offset(Vertex, r)` zurück.

Man könnte statt `offset(Vertex, r)` auch `3*sizeof(float)` oder `12` schreiben. **ABER** dann besteht die Gefahr, dass bei komplexeren Strukturen durch implizites Padding ungewollt eine Speicherbereichsverschiebung auftritt und das Shaderprogramm dann auf einen falschen Speicherbereich zugreift (siehe auch http://www.catb.org/esr/structure-packing). Dies ist auch der Grund, warum `sizeof(Vertex)` statt `6*sizeof(float)` als stride verwendet wird. Solange nur floats in der Struktur verwendet werden, wird der Compiler (normalerweise) kein Padding einfügen.

Initialisieren der Vertex- und Elementpuffer für die Boxen

Die ganze Arbeit der Vertex- und Index-Puffer-Erstellung wird im Konstruktor der Klasse `BoxObject` und der Hilfsklasse `BoxMesh` gemacht.

BoxObject.cpp:Konstruktor

```
BoxObject::BoxObject() :
    m_vbo(QOpenGLBuffer::VertexBuffer), // actually the default, so default constructor would have been enough
    m_ebo(QOpenGLBuffer::IndexBuffer) // make this an Index Buffer
{

    // create center box
    BoxMesh b(4,2,3);
    b.setFaceColors({Qt::blue, Qt::red, Qt::yellow, Qt::green, Qt::magenta, Qt::darkCyan});
    Transform3D trans;
    trans.setTranslation(0,1,0);
    b.transform(trans.toMatrix());
    m_boxes.push_back( b);

    const int BoxGenCount = 10000;
    const int GridDim = 50; // must be an int, or use cast below

    // initialize grid (block count)
    int boxPerCells[GridDim][GridDim];
    for (unsigned int i=0; i<GridDim; ++i)
        for (unsigned int j=0; j<GridDim; ++j)
            boxPerCells[i][j] = 0;
    for (unsigned int i=0; i<BoxGenCount; ++i) {
        // create other boxes in randomize grid, x and z dimensions fixed, height varies discretely
        // x and z translation in a grid that has 500 units width/depths with 5 m grid line spacing
        int xGrid = qrand()*double(GridDim)/RAND_MAX;
        int zGrid = qrand()*double(GridDim)/RAND_MAX;
        int boxCount = boxPerCells[xGrid][zGrid]++;
        float boxHeight = 4.5;
        BoxMesh b(4,boxHeight,3);
        b.setFaceColors({Qt::blue, Qt::red, Qt::yellow, Qt::green, Qt::magenta, Qt::darkCyan});
        trans.setTranslation((-GridDim/2+xGrid)*5, boxCount*5 + 0.5*boxHeight, (-GridDim/2 + zGrid)*5);
        b.transform(trans.toMatrix());
        m_boxes.push_back(b);
    }

    unsigned int NBoxes = m_boxes.size();

    // resize storage arrays
    m_vertexBufferData.resize(NBoxes*BoxMesh::VertexCount);
    m_elementBufferData.resize(NBoxes*BoxMesh::IndexCount);

    // update the buffers
    Vertex * vertexBuffer = m_vertexBufferData.data();
    unsigned int vertexCount = 0;
    GLuint * elementBuffer = m_elementBufferData.data();
    for (const BoxMesh & b : m_boxes)
        b.copy2Buffer(vertexBuffer, elementBuffer, vertexCount);
}
```

Wichtig ist zunächst die Initialisierung der `QOpenGLBuffer` Objekte. Als Konstruktorargument wird der Typ des Buffers angegeben (`VertexBuffer` ist der Standard, aber beim `m_ebo` Objekt muss man `IndexBuffer` festlegen).

Dann wird zunächst eine Testbox erstellt. Dies beinhaltet die folgenden Schritte:

1. Erstellung eines `BoxMesh` Objekts mit den Ausdehnungen 4x2x3 (die Box wird zentriert um das eigene Koordinatensystem erstellt, also x=-2…-2, y=-1…1, z=-1,5…1,5):

   ```
   BoxMesh b(4,2,3);
   ```

2. Festlegen der Seitenfarben:

```
b.setFaceColors({Qt::blue, Qt::red, Qt::yellow, Qt::green, Qt::magenta, Qt::darkCyan});
```

3. Verschiebung der Box in das Weltenkoordinatensystem (erst Erstellung der Transformationsmatrix, dann anwenden der Transformation auf die Box):

```
Transform3D trans;
trans.setTranslation(0,1,0);
b.transform(trans.toMatrix());
```

4. Zuletzt ablegen der Box im Vektor `m_boxes`:

```
m_boxes.push_back( b);
```

Die Klasse `BoxMesh` merkt sich zunächst nur die Koordinaten und Farbzuordnungen.

Als nächstes werden noch eine Reihe weiterer Boxen erstellt, und in einem Raster mit Dimension *GridDim x GridDim* gestapelt. Wenn man mal die eienen Grafikkarte testen will, kann man gerne `BoxGenCount` auf eine Million erhöhen und/oder das Gitterraster vergrößern (z.B. `GridDim=500`) um eine etwas größere "Stadt" zu bekommen.

Bei größeren Rasterdimensionen sieht man auch gut den Effekt des Tiefenclippings, d.h. Objekte hinter der FARPLANE werden nicht mehr gerendert.

Nun kommt der eigentlich interessante Teil. Es werden erst Pufferspeicher reserviert. Dabei liefern die Funktionen `BoxMesh::VertexCount` und `BoxMesh::IndexCount` die je Meshobjekt benötigte Anzahl von Elementen zurück. Man hätte hier auch gleich die Anzahl eintragen können, aber so bleibt der Code hinreichend universell und kann auf beliebige andere Meshobjekte übertragen werden.

Zuletzt kommt das Befüllen der Puffer in traditioneller C-Methodik zum Befüllen kontinuierlicher Speicherbereiche mit Elementen:

```
Vertex * vertexBuffer = m_vertexBufferData.data();
unsigned int vertexCount = 0;
GLuint * elementBuffer = m_elementBufferData.data();
for (const BoxMesh & b : m_boxes)
    b.copy2Buffer(vertexBuffer, elementBuffer, vertexCount);
```

Es werden erst Zeiger auf den Beginn des Pufferspeichers geholt und der Startindex der Vertices auf 0 gesetzt. Dann werden in jedem Schleifendurchlauf die Daten eines BoxMeshes in die Puffer geschrieben und die Zeigervariablen entsprechend vorgerückt. Ebenso wird der Startindex der Vertexes erhöht (`vertexCount`), sodass bei er nächsten Box neue Vertexnummern vergeben werden.

In dieser Art ließen sich ohne weiteres andere Objekttypen verwalten und zusammengefasst in einen Zeichenpuffer kopieren. Die ganze objektspezifische Geometriearbeit passiert im jeweiligen Mesh-Objekt, in diesem Fall in der Klasse `BoxMesh`.

Die Klasse BoxMesh

Inzwischen sollte die Aufgabe der Klasse `BoxMesh` klar sein:

- speichern der originalen Geometrie (im lokalen Koordinatensystem)
- speichern/anwenden der Transformation zum Weltenkoordinatensystem
- befüllen des linearen Vertexpuffer-Speichers und Elementpuffer-Speichers

Auch hier gibt es wieder verschiedene Möglichkeiten. Man kann sich, nach dem Prinzip der *lazy evaluation* erst einmal nur die für die Schritte benötigten Parameter merken, also z.B. Breite, Höhe und Länge der Box, und die Transformationsmatrix. Wenn dann der Vertexpuffer gefüllt werden soll, erstellt man die Vertexkoordinaten, führt die Transformation aus und kopiert dann die resultierenden Koordinaten. Das Verfahren ist sinnvoll, wenn sich die Transformation (also Model-zu-Weltkoordinaten) häufig ändert.

Alternativ kann man, wie hier in Tutorial 05, auch die Koordinaten gleich berechnen, d.h. beim Erstellen des Objekt die Vertexkoordinaten im lokalen Koordinatensystem festlegen, und dann bei Ausführen der Transformation sofort an Ort und Stelle transformieren. Dies reduziert die Arbeit beim eigentlichen Befüllen des OpenGL-Vertex-Puffers, führt aber zu witzigen Effekten bei mehrfacher Anwendung der in-place Transformation (wegen der unvermeidlichen Rundungsfehler... einfach mal mehrere 100 Mal im Kreis drehen und sich über die Geometrieveränderung freuen). Da Animation oder Transformation in diesem Tutorial keine Rolle spielt, werden die Boxen gleich zu Beginn ins Weltenkoordinatensystem transformiert.

Bevor wir uns der eigentlichen Implementierung widmen, hift vielleicht die eine oder andere Skizze, die Box-Geometrie zu verstehen:

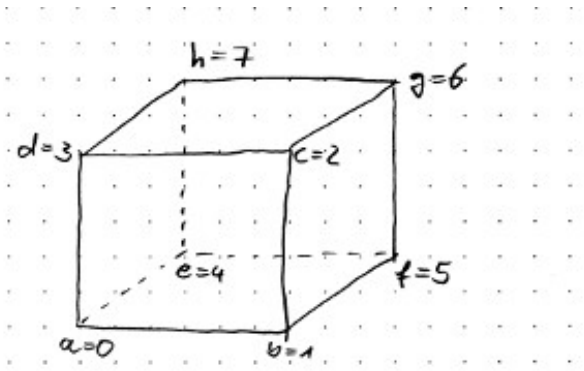

Figure 9. Nummerierung der Knoten (Vertices) der Box

Die Nummerierung der Vertexes ist zunächst einmal für die Datenhaltung in der `BoxMesh`-Klasse notwendig. Es werden nämlich im Konstruktor schon einmal die Vertexkoordinaten berechnet:

BoxMesh.cpp, Konstruktor

```
BoxMesh::BoxMesh(float width, float height, float depth, QColor boxColor) {

    m_vertices.push_back(QVector3D(-0.5f*width, -0.5f*height,  0.5f*depth)); // a = 0
    m_vertices.push_back(QVector3D( 0.5f*width, -0.5f*height,  0.5f*depth)); // b = 1
    m_vertices.push_back(QVector3D( 0.5f*width,  0.5f*height,  0.5f*depth)); // c = 2
    m_vertices.push_back(QVector3D(-0.5f*width,  0.5f*height,  0.5f*depth)); // d = 3

    m_vertices.push_back(QVector3D(-0.5f*width, -0.5f*height, -0.5f*depth)); // e = 4
    m_vertices.push_back(QVector3D( 0.5f*width, -0.5f*height, -0.5f*depth)); // f = 5
    m_vertices.push_back(QVector3D( 0.5f*width,  0.5f*height, -0.5f*depth)); // g = 6
    m_vertices.push_back(QVector3D(-0.5f*width,  0.5f*height, -0.5f*depth)); // h = 7

    setColor(boxColor);
}
```

Die Knotenkoordinaten sind zunächst in einem Vektor von `QVector3D` abgelegt. Bei einem nachfolgenden Aufruf zur Transformation werden diese Koordinaten einfach *verändert*:

BoxMesh.cpp:transform()

```
void BoxMesh::transform(const QMatrix4x4 & transform) {
    for (QVector3D & v : m_vertices)
        v = transform*v;
}
```

Bei mehrfacher Ausführung von `transform()` auf die Rundungsfehler achten!

Nun sind die Boxen also bereits im Weltenkoordinatensystem verankert und der Vertexpuffer und Indexpuffer können befüllt werden.

Für das weitere Vorgehen ist es hilfreich, das Speicherlayout des Vertexpuffers einmal gesehen zu haben. Die folgende Abbildung zeigt das Ziel dieser Kopieraktion.

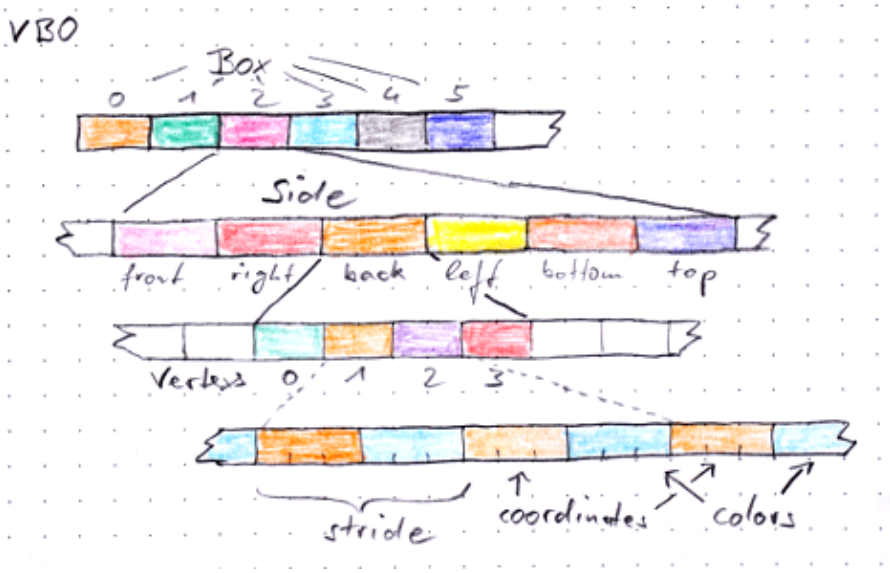

Figure 10. Speicherlayout des Vertexpuffers

Alle Boxen werden nacheinander im VBO abgelegt. Je Box sind das 6 Seiten, wobei für jede Seite 4 Vertexes mit je Koordinaten und Farbwerten abgelegt werden. Das Kopieren erfolgt in der Funktion `copy2Buffer()`, wobei jeweils die Daten für eine einzelne Box kopiert werden. In der Abbildung ist auch der *stride* (Länge eines Vertexdatenblocks) gezeigt.

In der Funktion `copy2Buffer()` wird zunächst ein temporärer Vektor `cols` mit Farben für jede Seite angelegt, für den Fall, dass einfarbige Boxen verwendet werden:

BoxMesh.cpp:copy2Buffer()

```
void BoxMesh::copy2Buffer(Vertex *& vertexBuffer, GLuint *& elementBuffer, unsigned int & elementStartIndex) const {
    std::vector<QColor> cols;
    Q_ASSERT(!m_colors.empty());
    // three ways to store vertex colors
    if (m_colors.size() == 1) {
        cols = std::vector<QColor>(6, m_colors[0]);
    }
    else {
        Q_ASSERT(m_colors.size() == 6);
        cols = m_colors;
    }

    ...
```

Nun werden die Seiten nacheinander in der Reihenfolge *vorne, rechts, hinten, links, unten* und *oben* in die Puffer geschrieben:

BoxMesh.cpp:copy2Buffer(), fortgesetzt

```
void BoxMesh::copy2Buffer(Vertex *& vertexBuffer, GLuint *& elementBuffer, unsigned int & elementStartIndex) const {
    ...

    // front plane: a, b, c, d, vertexes (0, 1, 2, 3)
    copyPlane2Buffer(vertexBuffer, elementBuffer, elementStartIndex,
            Vertex(m_vertices[0], cols[0]),
            Vertex(m_vertices[1], cols[0]),
            Vertex(m_vertices[2], cols[0]),
            Vertex(m_vertices[3], cols[0])
        );

    // right plane: b=1, f=5, g=6, c=2, vertexes
    // Mind: colors are numbered up
    copyPlane2Buffer(vertexBuffer, elementBuffer, elementStartIndex,
            Vertex(m_vertices[1], cols[1]),
            Vertex(m_vertices[5], cols[1]),
            Vertex(m_vertices[6], cols[1]),
            Vertex(m_vertices[2], cols[1])
        );

    // back plane: g=5, e=4, h=7, g=6
    copyPlane2Buffer(vertexBuffer, elementBuffer, elementStartIndex,
            Vertex(m_vertices[5], cols[2]),
            Vertex(m_vertices[4], cols[2]),
            Vertex(m_vertices[7], cols[2]),
            Vertex(m_vertices[6], cols[2])
        );

    // left plane: 4,0,3,7
    copyPlane2Buffer(vertexBuffer, elementBuffer, elementStartIndex,
            Vertex(m_vertices[4], cols[3]),
            Vertex(m_vertices[0], cols[3]),
            Vertex(m_vertices[3], cols[3]),
            Vertex(m_vertices[7], cols[3])
        );

    // bottom plane: 4,5,1,0
    copyPlane2Buffer(vertexBuffer, elementBuffer, elementStartIndex,
            Vertex(m_vertices[4], cols[4]),
            Vertex(m_vertices[5], cols[4]),
            Vertex(m_vertices[1], cols[4]),
            Vertex(m_vertices[0], cols[4])
        );

    // top plane: 3,2,6,7
    copyPlane2Buffer(vertexBuffer, elementBuffer, elementStartIndex,
            Vertex(m_vertices[3], cols[5]),
            Vertex(m_vertices[2], cols[5]),
            Vertex(m_vertices[6], cols[5]),
            Vertex(m_vertices[7], cols[5])
        );
}
```

Beim Aufruf der Funktion `copyPlane2Buffer()` stehen die Zeiger `vertexBuffer` und `elementBuffer` stehts am Anfang des Speicherbereichs, in den die nun folgenden Seitendaten geschrieben werden.

Ebenso enthält die Variable `elementStartIndex` den Vertexindex, bei dem die Nummerierung beginnt. Bei der ersten Box beginnt die Nummerierung auf der Vorderseite mit 0 (d.h. Vertexes 0...3 sind auf der Vorderseite), siehe auch folgende Abbildung:

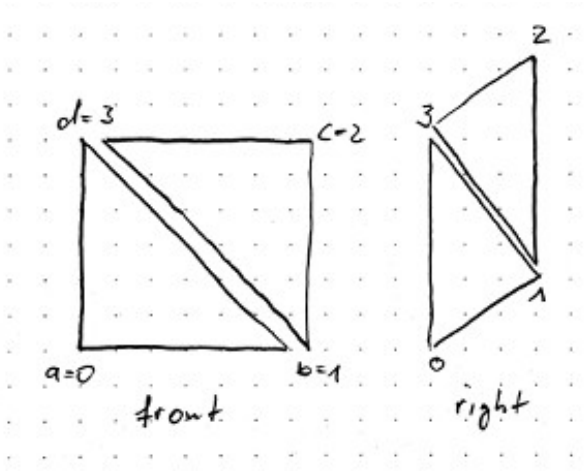

Figure 11. Seitennummerierung und generierte Dreieckselemente

Die Koordinaten und Farben werden beim Aufruf in die Vertex-Struktur kopiert.

Nachdem die Daten für die Vorderseite kopiert wurden, sind die Zeiger entsprechend verschoben worden und zeigen nun auf den Speicherbereich der nächsten Seite. Beim Aufruf der Funktion `copyPlane2Buffer()` muss auf die korrekte Reihenfolge der Vertexes geachtet werden, sodass die Vertices immer entgegen des Uhrzeigersinns übergeben werden.

Die letzte Abbildung zeigt auch die zwei Dreiecke, welche die Seite bilden. Deshalb wird in dieser Funktion sowohl der Vertexpuffer als auch der Indexpuffer befüllt. Innerhalb der Funktion `copyPlane2Buffer()` wird die Nummerierung relativ durchgeführt, d.h. die Vertices sind *immer* 0 bis 3, wobei allerdings stets der Startindex addiert wird (siehe Abbildung, rechte Seite).

BoxMesh.cpp:copyPlane2Buffer()

```
void copyPlane2Buffer(Vertex * & vertexBuffer, GLuint * & elementBuffer, unsigned int & elementStartIndex,
    const Vertex & a, const Vertex & b, const Vertex & c, const Vertex & d)
{
    // first store the vertex data (a,b,c,d in counter-clockwise order)

    vertexBuffer[0] = a;
    vertexBuffer[1] = b;
    vertexBuffer[2] = c;
    vertexBuffer[3] = d;

    ...

    // advance vertexBuffer
    vertexBuffer += 4;

    // we generate data for two triangles: a, b, d  and b, c, d

    elementBuffer[0] = elementStartIndex;
    elementBuffer[1] = elementStartIndex+1;
    elementBuffer[2] = elementStartIndex+3;
    elementBuffer[3] = elementStartIndex+1;
    elementBuffer[4] = elementStartIndex+2;
    elementBuffer[5] = elementStartIndex+3;

    // advance elementBuffer
    elementBuffer += 6;
    // 4 vertices have been added, so increase start number for next plane
    elementStartIndex += 4;
}
```

Hier machen wir uns nun eine nette Eigenschaft von C/C++ zu Nutze. Wenn wir einen Speicherbereich als Vektor einer Struktur behandeln, und via Index Objekte zuweisen, dass wird automatisch der Speicherbereich mit den Inhalten der Strukturen in der Reihenfolge der Deklaration der Variablen befüllt.

Da die Addressen und der Startindex als Referenzvariablen übergeben wurden, können wir die Zeiger "weiterschieben" und die Vertexanzahl entsprechend erhöhen.

Das schöne an der Funktion `copyPlane2Buffer()` ist, dass sie unverändert auch funktioniert, wenn die `Vertex` -Struktur später um Normalenvektoren und/oder Texturkoordinaten erweitert wird.

Mehr gibt es auch zur Klasse `BoxMesh` nicht zu sagen, womit wir am Ende des *Tutorial 05* angelangt wären. Um das ganze aber noch abzurunden (und etwas schicker aussehen zu lassen) fehlt noch Kantenglättung.

5.8. Antialiasing

Es gibt hier verschiedene Möglichkeiten, Antialiasing (Kantenglättung) zu verwenden. Die wohl einfachste aus Sicht der Programmierung ist das Einschalten von Multisampling (MSAA) (siehe Erläuterung auf https://www.khronos.org/opengl/wiki/Multisampling).

Dazu muss man beim Konfigurieren des `QSurfaceFormat`-Objekts nur folgende Zeile hinzufügen:

```
format.setSamples(4);   // enable multisampling (antialiasing)
```

Multisampling braucht mehr Grafikkartenspeicher und ist durch das mehrfache Samplen von Pixeln/Fragmenten

natürlich langsamer. Daher gibt es auch die Möglichkeit, Antialiasing in das Shaderprogramm einzubauen. Das ist aber, ebenso wie ein Drahtgittereffekt, ein Thema für ein anderes Tutorial.