Schritt 2: Shader & GLSL Programme erzeugen & das Basis Setup zum Rendern

Shader & Programmobjekte

GLSL Shader werden wir als Textdateien mit entsprechender Endung speichern: *vert/frag/geo* usw. - dies erlaubt es Ihnen, Syntax Highlighting Tools zu verwenden. Es gibt, soweit ich weiß, ein Plugin für MS Visual Studio. Ich selber verwende den Texteditor Notepad++. Wie auch immer, die Speicherung in Textdateien ist dem *Hardcoden* als `std::string` vorzuziehen, Shader Programmierung ist an sich schon frustrierend genug: Es gibt zwar Regeln, wann ein GLSL Shader syntaktisch korrekt ist, unterschiedliche Treiberhersteller halten diese Regeln jedoch unterschiedlich genau ein. Auch der Output des Compilers beschreibt Fehler unterschiedlich genau.

Mehrere GLSL Shader (Standard: 1 Vertex- und 1 Fragment Shader) werden zu einem GLSL Programmobjekt verlinkt, fürs Rendern brauchen wir immer in dieser LVA immer mindestens ein solches Programmobjekt, später im Semester dann auch mehrere.

Für Erstellen des Programmobjektes bekommen Sie eine C++ Hilfsklasse von mir. Die Begründung dafür ist dieselbe wie schon bei der OpenGL Context Erstellung: der Erkenntnisgewinn beim Implementieren ist gering, die Wahrscheinlichkeit dass Sie einen subtilen Fehler irgendwo einbauen ist dagegen sehr hoch. Grundsätzlich begrüße ich es, wenn Sie Wrapper Klassen schreiben (wie etwa für Geometrie/Vertex Array Objects oder Bilder/2D Texturen), nur bei den Shadern eben nicht. Die Hilfsklasse finden Sie im Ordner *codesamples/cg2/glsl.h* - bzw. *glsl.cpp* - wobei das .cpp File eigentlich für Sie nicht von Interesse ist – Sie sollten lieber die Funktionsbeschreibungen im Header lesen.

Erzeugen eines GLSL Programmobjekts (INIT)

> `GlslProgram::createProgram`

Setzen von Uniform Variablen (RENDER)

> `GlslProgam::setUniformX` (→ X: Mat4, Mat3, Vec4, Vec3, Texture, etc.)

Auswählen / Aktivieren eines GLSL Programmobjekts, sodass es für die darauffolgenden Zeichen Operationen benutzt wird: (RENDER)

> `GlslProgram::setActiveProgram`

Zerstören des Programmobjekts (UN-INIT)

> Müssen Sie nicht selbst tun, weil Sie beim Anlegen einen Smart Pointer bekommen haben.

Grund-Setup fürs Rendering

Das zweite, was wir brauchen, ist die Initialisierung von zumindest ein paar OpenGL Settings, wie z.B. dem Z-Test. Darüber, ob diese Settings ins INIT gehören oder ins RENDER, kann man übrigens streiten... In einer echten Anwendung würden Sie alles, was konstant ist, schon in den INIT Teil packen – wenn z.B. der Z-Test immer aktiv sein soll. Ich halt es in meinen Code Samples lieber so, dass im INIT wirklich nur Ressourcen angelegt werden, das hilft meiner Erfahrung nach mit der Lesbarkeit. Daher wird in meinen Samples aller fürs Rendern relevante OpenGL State im RENDER gesetzt, und am Ende – vorm UPDATE - auch wieder auf den Default State rückgesetzt.

Der Code aus Schritt 1 erweitert sich bei uns also ein wenig. Im INIT kommt erst mal das Anlegen des Programobjektes hinzu.

```
std::shared_ptr< cg2::GlslProgram > tutorialProg = cg2::GlslProgram::create( … );
```

Im RENDER kommt etwas OpenGL Code hinzu… es wird aber noch nichts gezeichnet.

```
glEnable( GL_DEPTH_TEST );
glEnable( GL_MULTISAMPLE );

glViewport( 0, 0, windowWidth, windowHeight );

glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

cg2::GlslProgram::setActiveProgram( tutorialProg );
tutorialProg->setUniformMat4( "modelTf", glm::mat4( 1.f ) );
tutorialProg->setUniformMat4( "viewTf", glm::mat4( 1.f ) );
tutorialProg->setUniformMat4( "projectionTf", glm::mat4( 1.f ) );

// here, we will draw... using glDrawElements

cg2::GlslProgram::setActiveProgram( nullptr );

glDisable( GL_DEPTH_TEST );
glDisable( GL_MULTISAMPLE );
```

Wir aktivieren also alles, was wir für den Standard Ablauf beim Rendern von 3D Objekten brauchen werden. Ok, das Multisampling ist jetzt nicht unbedingt nötig, die Grafik schaut halt schöner/glatter aus... Aber den Z-Test wollen wir fast immer, und den Viewport (*) auf Fenstergröße setzen wollen wir eigentlich auch immer. Zum Z-Test: Die Default Settings in OpenGL sind, dass der Z-Buffer auf 1 *gecleared* wird, und die Vergleichsfunktion GL_LESS ist, also Fragmente den Z-Test bestehen, wenn ihr Z-Wert kleiner ist als der Gespeicherte. Mit diesen Einstellungen ist man meistens gut unterwegs, darum ändern wir diese auch nicht.

Des Weiteren setzen wir unser einziges GLSL Programmobjekt aktiv, und füllen die Uniform Variablen mit brauchbaren Werten. Wenn Sie den Vertex Shader öffnen, sehen Sie, dass dieser nichts weiter durchführt, als die Standard Transformation des Vertex mit Model-, View-, und Projektionsmatrix. Wenn wir alle drei auf die Einheitsmatrix setzen, bedeutet dies, dass jeder Vertex genauso im *Normalized Device Space* ankommt, wie er auf der CPU definiert wurde, in anderen Worten: Vertices, die im *Object Space* innerhalb

des Einheitswürfels definiert sind, werden gerendert werden, alle anderen werden *geclipped*.

*) Die Viewport Matrix ist die einzige 4x4 Matrix aus der klassischen CG Pipeline, welche wir nicht explizit spezifizieren müssen. Sie erinnern sich: Model Transformation → View Transformation→ Projection Transformation → Viewport Transformation. Die ersten drei Transformationsmatrizen werden bei uns typischerweise Uniforms sein und im Vertex Shader Anwendung finden. OpenGL leitet sich die Viewport Matrix von alleine aus dem `glViewport` Befehl her. Außerdem sollten Sie folgendes wissen: In OpenGL ist (0,0) des Viewports links unten. In SFML ist (0,0) des Fensters links oben, d.h. Wenn Sie Maus Koordinaten auf den Viewport beziehen wollen, müssen Sie diese entsprechend umrechnen.