OpenGL und Shaderprogrammierung

Gerade für die Computergraphik ist die Tatsache entscheidend, dass sich die Lebenserwartungen aktueller Grafikarten in Bezug auf Neuerscheinungen von Generation zu Generation erheblich verkürzen. Um unseren Beispielen eine etwas längere Lebenserwartung zu ermöglichen wurde auf OpenGL und die dazugehörige Shadersprache GLSL gesetzt.

Die Funktionalität bleibt prinzipiell erhalten, lediglich der Sprachumfang erweitert sich mit jeder neuen Version. Eine Alternative ist sicherlich DirectX mit der Shadersprache HLSL, wobei sich die Sprache von Version zu Version stark unterscheiden kann.

Um mit Java den gleichen OpenGL-Code verwenden zu können, wie es beispielsweise mit GLUT unter C++ möglich ist, wird das Framework Lightweight Java Game Library (LWJGL) verwendet[1].

Dieser Abschnitt gibt eine sehr kurze Einführung in die Arbeit mit OpenGL und GLSL unter LWJGL. Für den kompletten Sprachumfang sollte allerdings das Buch „*OpenGL Programming Guide*“ [8] (alias „*the red book*“) nicht fehlen. Das Pendant für GLSL heißt „*OpenGL Shading Language*“ [9] (alias „*the orange book*“) ist aber im Gegensatz dazu didaktisch sehr gut aufgebaut und als Lehrbuch für GLSL sehr zu empfehlen.

15.9 Rendering-Pipeline bei OpenGL

Um 3D-Szenen darstellen zu können, werden zunächst je nach Bedarf 3D-Objekte mit typischen Modellierungswerkzeugen wie z. B. 3D-Max, modelliert. Ein Baukasten aus kleinen geometrischen Bausteinen, wie Dreiecken, Linien usw. steht zur Vefügung. Ein 3D-Modell besteht aus zusammengesetzten Primitiven und wird anschließend texturiert, indem die Primitiven Teilabschnitte einer gegebenen Oberfläche repräsentieren. Für das Rendering einer 3D-Szene werden die 3D-Objekte jetzt entsprechend der gewünschten Ausrichtung und Größe positioniert. Jetzt kommen auch Lichtquellen und die Kameraposition ins Spiel.

Allgemein und vereinfacht lässt sich die Rendering-Pipeline von OpenGL mit dem instruktiven Beispiel in Abbildung 15.15 erklären.

[1] http://lwjgl.org/

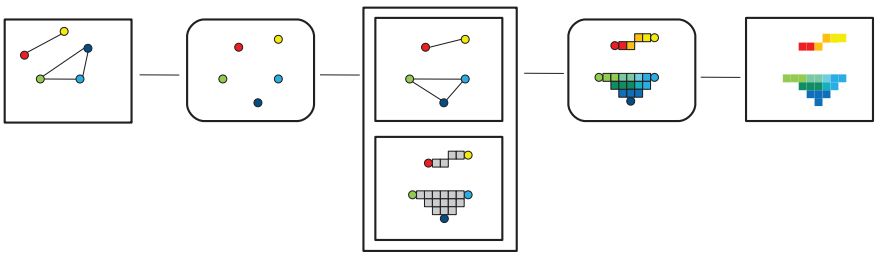

Abbildung 15.15: Typische Render-Pipeline von OpenGL (von links nach rechts): Gegeben sind geometrische Primitive mit unterschiedlichen Informationen (links als Farben dargestellt). Anschließend werden diese durch den Vertex-Shader an die gewünschten, neuen Positionen transformiert. Nach dem Zielraster werden die Inhalte der Primitiven diskretisiert und die Inhalte linear interpoliert (mittig). Der Fragment-Shader übernimmt die Färbung der interpolierten Werte. Schließlich wählen wir den Ausgabebereich, z. B. den Bildschirm (rechts)

In der ersten Phase werden die Daten der Primitiven bestehend aus Vertices definiert. In diesem Beispiel trägt jeder Vertex symbolhaft für eine ganze Reihe von möglichen Informationen eine unterschiedliche Farbe. Die Primitiven haben jeweils unterschiedliche Bezugssysteme.

Um diese Koordinatensysteme in der darzustellenden Welt zu vereinen, müssen diese transformiert werden. Diese Aufgabe übernimmt der Vertex-Shader. Jedem Vertex wird dabei eine Position in der zu generierenden Welt zugewiesen, es können sogar mehrere auf dieselbe Position fallen. Dabei spielen die Informationen über die dazugehörigen Primitiven zunächst keine Rolle. Alle Vertices werden quasi parallel transformiert. Sie kennen also die neuen Positionen der Nachbarknoten zu diesem Zeitpunkt nicht.

Im anschließenden Rasterisierungsschritt werden die Informationen über die Zusammenhänge der Vertices verwendet, um die innenliegenden Bereiche zu lokalisieren. Diese werden nach dem Zielraster, beispielsweise dem Bildschirm oder einfach einem definierten Speicherabschnitt (*Framebuffer*) diskretisiert. Dabei werden für alle innenliegenden Fragmente die bestehenden Vertexinformationen linear interpoliert.

Wir hatten in diesem Beispiel für die Vertexinformationen konkrete Farben verwendet. Nach dem Rasterisierungsprozess liegen nun für alle Fragmente interpolierte Werte vor. Der Fragment-Shader bestimmt parallel für jedes Fragment aus den interpolierten Werten eine Farbe. Verwenden wir also einfach die interpolierten Farbwerte, sehen wir sehr schnell den auftretenden Effekt.

Beispielsweise könnten wir das erhaltene Raster, das wir in einem Framebuffer gesammelt haben, auf dem Bildschirm ausgeben und erhalten einen fließenden Übergange der Farben in den Primitiven. Die abstrakten fünf Schritte der Pipeline werden in Abbildung 15.16 noch einmal gezeigt.

15.10 OpenGL als Zustandsautomat

OpenGL und DirectX wurden als Zustandsautomaten realisiert. Das bedeutet, dass alle Parameter solange verwendet werden, bis diese verändert werden.

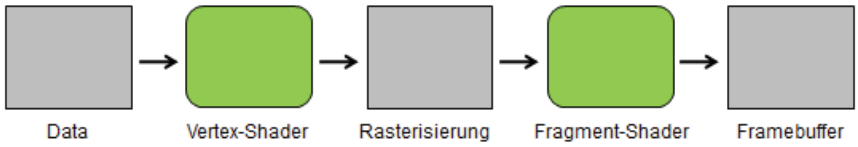

Abbildung 15.16: Bei den abstrakten fünf Schritten der Rendering-Pipeline in OpenGL sehen wir die markierten Bereiche für Vertex- und Fragment-Shader. Das sind genau die beiden Phasen, die wir in den folgenden Abschnitten direkt manipulieren werden

Stellen wir beispielsweise die aktuelle Farbe auf rot, werden im Anschluß alle Vertices in rot dargestellt. Durch dieses Konzept werden wir dazu gezwungen, uns mit den Defaulteinstellungen des Zustandsautomaten zu beschäftigen. Um Programmpassagen austauschbar zu machen sollten im Anschluß an die entsprechende Methode wieder die Defaulteinstellungen verwendet werden.

15.11 Einführung in Vertex- und Fragment-Shader

Wir starten jetzt für den Einstieg mit jeweils einer minimalistischen Version eines Vertex- und eines Fragment-Shaders. In einem Vertex-Shader werden parallel für alle vorhandenen Vertices neue Positionen berechnet:

```
1 void main(void) {
2    gl_Position  = gl_ModelViewProjectionMatrix * gl_Vertex;
3 }
```

Die Aufgabe eine neue Position `gl_Position` in Abhängigkeit zur alten Position `gl_Vertex` zu ermitteln ist sozusagen die Mindestanforderung dieses Shadertyps. Beide Vektoren sind vom Datentyp `vec4`. In `gl_Position` berechnen wir in diesem Beispiel die neue Position aus `gl_Vertex` einfach in Abhängigkeit unserer `ModelViewProjectionMatrix`.

Die wichtigste Aufgabe, die ein Fragment-Shader zu erledigen hat, ist die Entscheidung, welche Farbe für das aktuelle Fragment (wir können es als Pixel interpretieren) gesetzt werden soll:

```
1 void main(void) {
2    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
3 }
```

In diesem einfachen Beispiel setzen wir alle Fragmente auf rot, denn der angegebene Vektor `vec4` enthält die Farbinformationen: `(rot, grün, blau, alpha)`. Die folgenden Unterabschnitte sollen uns dazu dienen, einen kleinen Einblick in die Shadersprache GLSL zu erhalten und ein Schema für die Einbettung von Shadercode in LWJGL zu erarbeiten.

15.12 Vektoren und Matrizen in GLSL

Es gibt zunächst einmal drei zur Verfügung stehende Vektortypen mit unterschiedlichen Dimensionen: `vec2`, `vec3` und `vec4`. Zwei Vektoren gleicher Dimension lassen sich dabei einfach und intuitiv komponentenweise addieren:

```
1 vec3 v1, v2;
2 vec3 v3 = v1 + v2;
```

Für den Programmierer gibt es die Möglichkeit, den Vektortypen eine semantische Bedeutung zu geben. So können die drei Vektorbedeutungen Farbe, Position und Textur unterschieden werden. Die direkten Zugriffe auf die Komponenten erfolgen über die entsprechenden Attribute:

```
1 vec4 col, pos, tex;
2 vec4 col2         = vec4(col.r, col.g, col.b, col.a); //Farbe
3 vec4 pos2         = vec4(pos.x, pos.y, pos.z, pos.w); //Position
4 vec4 tex2         = vec4(tex.s, tex.t, tex.p, tex.q); //Textur
```

Dem Compiler ist die Bedeutung allerdings gleichgültig und verwendet im Hintergrund nur einen Vektortyp. Um mehrere Komponenten aus einem Vektor lesen und übernehmen zu können, schreiben wir einfach:

```
1 vec2 redGreen     = col.rg;
2 vec3 pos_xyz      = pos.xyz;
```

Sogar multiples Auslesen einer Position ist möglich:

```
1 vec4 v2           = v.xxyy;
```

Der Compiler erlaubt es allerdings nicht, die angebotenen semantischen Bedeutungen der Vektoren zu vermischen und wirft einen Fehler bei dem folgenden Versuch:

```
1 vec4 v2           = v.xgba;
```

Typumwandlungen sind nur über Konstruktoren möglich, dabei wird eine ganze Reihe von Kombinationsmöglichkeiten angeboten:

```
1 vec3 coord_xyz    = vec3(1.0f, 0.0f, -0.5f);
2 vec4 coord_xyzw   = vec4(coord_xyz, 1.0f);
```

Hier noch ein paar abschließende Beispiele für die Indizierung:

```
1 vec4 v            = vec2(1.0f, 2.0f, 3.0f, 4.0f);
2 float f           = v[2];  // liefert Element 3.0f
3 v.xw              = vec2(8.0f, 9.0f);
```

Nach dem letzten Beispiel trägt der Vektor `v` die Werte `(8.0f, 2.0f, 3.0f, 9.0f)`.

Der Umgang mit Matrizen und das Zusammenspiel von Matrizen und Vektoren sind ebenfalls sehr komfortabel und intuitiv in GLSL gelöst:

```
1 mat4 M            = mat4(3.0);  // Diagonale wird auf 3.0 gesetzt
2 vec4 v            = M[1];       // zweite Spalte wird kopiert
```

Als wichtigste Referenz sei an dieser Stelle aus das Buch von Rost verwiesen .

15.13 Modularisierung des Programmcodes

Analog zu fast allen gängigen imperativen Programmiersprachen ist es in GLSL ebenfalls möglich, Programmteile in Funktionen auszulagern. In dem folgenden Beispiel wollen wir zeigen, wie die Funktionen `TranslateMatrix` und `ScaleMatrix`, die jeweils aus einem Vektor die entsprechende homogene Matrix erzeugt, in einem Vertex-Shader erstellt werden können:

```
mat4 TranslateMatrix(const in vec3 t){
   return mat4(1.0f, 0.0f, 0.0f, 0.0f,
               0.0f, 1.0f, 0.0f, 0.0f,
               0.0f, 0.0f, 1.0f, 0.0f,
               t.x,  t.y,  t.z, 1.0f);
}

mat4 ScaleMatrix(const in vec3 s){
   return mat4(s.x,  0.0f, 0.0f, 0.0f,
               0.0f,  s.y, 0.0f, 0.0f,
               0.0f, 0.0f,  s.z, 0.0f,
               0.0f, 0.0f, 0.0f, 1.0f);
}

void main(void) {
   float sFac     = 0.5f;
   mat4  SCALE_M  = ScaleMatrix(vec3(sFac, sFac, sFac));
   vec4  res      = SCALE_M * vec3(1.0f, -1.0f, 2.0f);
   ...
}
```

In der `main`-Methode wird zunächst eine Skalierungsmatrix `SCALE_M` erzeugt und anschließend mit einem Vektor multipliziert. Mit dem Auslagern von Codefragmenten können wir übersichtlichere Programme und wiederverwendbare Programmabschnitte erstellen.

15.14 Shadercode mit LWJGL

Die Einbettung von GLSL-Code in Java werden wir mit Hilfe der freien Bibliothek LWJGL[2] vornehmen. Dazu müssen wir den GLSL-Code für die Vertex- und Fragment-Shader zunächst in einem String ablegen:

```
String vertShader =
   "void main(void) {"+
   "  gl_Position  = gl_ModelViewProjectionMatrix * gl_Vertex;"+
   "}";

String fragShader =
   "void main(void) {"+
   "   gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);"+
   "}";
```

Diese beiden Zeichenketten werden später direkt in den Speicher der Grafikkarte kopiert. Eine Übersicht zum Gesamtworkflow von Shader- und Program-Objekten in LWJGL zeigt Abbildung 15.17.

Wir wollen die einzelnen Schritte in Java mit einem Beispiel exemplarisch durchgehen. Die Zeichenketten für Vertex- und Fragment-Shader liegen bereits vor, jetzt wollen wir die beiden Shader-Objekte erzeugen und die entsprechenden Zeichenketten binden:

[2] http://www.lwjgl.org/

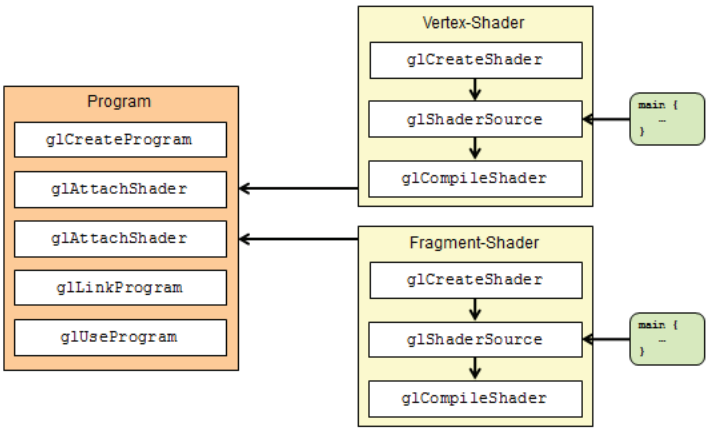

Abbildung 15.17: Kompletter Workflow, um in LWJGL GLSL-Shadercode einzubinden. Die beiden Zeichketten für die Vertex- und Fragment-Shader sind rechts zu sehen und werden in den jeweiligen Shader-Objekten mit glShaderSource gebunden. Nachdem die beiden Shader-Objekte kompiliert wurden, werden sie mit einem Program-Objekt verknüpft

```
1 int shaderObjectV = glCreateShader(GL_VERTEX_SHADER);
2 glShaderSource(shaderObjectV, vertShader);
3 glCompileShader(shaderObjectV);
4
5 int shaderObjectF = glCreateShader(GL_FRAGMENT_SHADER);
6 glShaderSource(shaderObjectF, fragShader);
7 glCompileShader(shaderObjectF);
```

Anschließend wird ein Program-Objekt erzeugt und die Shader-Objekte über die Funktion `glAttachShader` verknüpft:

```
1 int programObject = glCreateProgram();
2 glAttachShader(programObject, shaderObjectV);
3 glAttachShader(programObject, shaderObjectF);
4 glLinkProgram(programObject);
```

Mit `glUseProgramm` können wir das erstellte Shader-Programm in die Grafikkarte laden und verwenden:

```
1 glUseProgram(programObject);
```

Alle folgenden OpenGL-Befehle durchlaufen dann die Pipeline mit unseren Varianten der Vertex- und Fragment-Shader.