3D Applikationen in WebGL

Projektarbeit

von

Michael Legner
aus Ellwangen/Jagst

Matrikelnummer: 27487

Martin-Luther-Str. 1
73463 Westhausen
07363/4999

HTW Aalen

Hochschule für Technik und Wirtschaft
Betreuer: Prof. Dr. Ulrich Klauck

Abgabetermin: 04.10.2011

Inhaltsverzeichnis

1 Grundlagen

1.1 3D Computergraphik

3D Computergraphik ist im wesentlichen die Darstellung von dreidimensionalen Objekten auf einem zweidimensionalem Ausgabegerät. Der Prozess wird in drei Phasen aufgeteilt [Wika]:

1. Modellierung: Ein Model beschreibt die Form eines Objektes. Sie werden manuell in Form von Koordinaten-Angabe, von spezieller Software oder prozedural erzeugt.

2. Platzierung und Animation: Das Objekt muss innerhalb einer sog. Szene platziert werden, bevor es gezeichnet werden kann. Die Platzierung kann durch die Animation verändert werden.

3. Renderig: Das eigentliche Zeichnen des Objekts. Dabei wird zwischen Photorealistischem und nicht-photoreastischem Rendering unterschieden, was speziell auf die Art der Beleuchtungsrechnung Einfluss hat, welchen den größten Rechenaufwand erzeugt.

Das erzeugte Bild wird auf einem Ausgabegerät angezeigt. Da diese meist nur über zwei Dimensionen verfügen, muss die Szene auf ein zweidimensionales Bild reduziert werden.

1.2 OpenGL und OpenGL ES

OpenGL ist eine offene, standardisierte API zur Erzeugung von 2D und 3D Computergraphiken. OpenGL entstand aus der proprietären API IrisGL von Silicon Graphics (SGI), aus welchem Teile, die nicht direkt zur Computergraphik gehören, entfernt wurden, z.b. APIs zur Nutzung von Maus und Tastatur. OpenGL konzentriert sich allein auf die Erzeugung von Graphiken, was den Vorteil hat, dass diese Teile unabhängig von anderen Programmteilen laufen können. Die ersten Version 1.0 wurde im Januar 1992 vom *OpenGL architectural review board* (OpenGL ARB) veröffentlicht, einer Gruppe Firmen, die in den folgenden Jahren OpenGL gepflegt und weiterentwickelt hat. Die aktuelle Version 4.2 wurde am 8. August 2011 vom Khronos-Konsortium veröffentlicht, welches seit August 2006 die Weiterentwicklung der API verwaltet und das OpenGL ARB beheimatet. Das nicht kommerziell orientierte Konsortium besteht aus führenden Firmen in der Industrie wie Nvidia, AMD, Intel, ARM und weiteren.

OpenGL ES ist eine Abwandlung von OpenGL, dass speziell für den Einsatz in Embedded-Geräten wie Mobiltelefonen, PDAs und ähnlichem entwickelt wurde. Die API wurde für das neue Einsatzgebiet stark entschlackt, orientiert sich aber an den Hauptversionen von OpenGL und ist teilweise kompatibel. OpenGL ES 1.0 wurde gegen OpenGL 1.3 zertifiziert, Programm in dieser Version von OpenGL ES sollen sich leicht in Programme mit der zugehörigen OpenGL Version portieren lassen. Die Versionen sind normalerweise nicht abwärtskompatibel, sondern nur zur zugehörigen OpenGL-Version. Seit Version 2.0 von OpenGL ES wird zwingend der Einsatz von Hardware, die Vertex- und Fragmentshader unterstützt, vorausgesetzt.

1.3 WebGL

WebGL (Web Graphics Library) ist eine offene Low-Level-API zur Erzeugung von 2D- und 3D-Computergraphiken, die auf OpenGL ES 2.0 basiert und das *canvas*-Element im HTML-5 Standard nutzt. Das *canvas*-Element definiert einen Bereich im Browser, in den frei gezeichnet werden kann, auch nach dem Laden der Seite. WebGL entstand aus Experimenten mit canvas und 3D-Grafik beim Browser-Entwickler Mozilla und wird seit 2009 vom Khronos-Konsortium verwaltet, welches auch die *WebGL working group* beheimatet, in welcher alle großen Browser-Hersteller außer Microsoft vertreten sind. Da OpenGL ES nur für das Erzeugen der Graphiken geeignet ist, wird für alle anderen Programmteile JavaScript eingesetzt. Der Standard setzt zwingend das Vorhandensein von OpenGL ES 2.0 Hardware voraus. Version 1.0 der Spezifikation der API wurde im März 2011 veröffentlicht.

1.4 JavaScript

JavaScript ist eine Prototyp-basierte Skriptsprache, die in allen relevanten Browser für clientseitige Berechnungen verwendet wird. Es stützt sich auf den ECMA Skript-Standard. JavaScript wurde ursprünglich als LiveScript von Netscape entwickelt und war erstmals mit Version 2.0 des Browsers, der im September 1995 veröffentlicht wurde, enthalten. Später wurde sie in Absprache mit Sun Microsystems, dem Entwickler der Programmiersprache Java, in JavaScript umbenannt. JavaScript entleiht Java einige Konvetionen für Namen, sonst gibt es aber keine Gemeinsamkeiten. JavaScript ist dynamisch typisiert und objektbasiert, bietet aber keine Objektorientierung, da das Konzept von Klassen fehlt. Die Implementierung kann

aber ähnlich erfolgen, in dem das Programm nach dem Singleton-Pattern aufgebaut wird [Pet].

1.5 Shader

OpenGL ES setzt seit Version 2.0, welche WebGL zugrunde liegt, Hardware voraus, die Vertex- und Fragmentshader unterstützt. In der offenen Shadersprache GLSL geschriebene Programme werden in den entsprechenden Einheiten der Grafikkarte ausgeführt, für jedes Vertex- oder Fragment eine Szene. Fragment-Shader werden oft fälschlicherweise als Pixelshader bezeichnet. Die Farbwerte eines Pixels werden aus einzelnen Fragmenten berechnet, u.a. Position im Koordinatensystem, Farb- oder Texturwert, Alphawert und andere. GLSL wird innerhalb des OpenGL ARB entwickelt, neue Versionen erscheinen parallel zu OpenGL Versionen. Die Sytax ist einfach und orientiert sich stark an der Programmiersprache C.

1.6 Vereinfachte Renderpipeline von WebGL

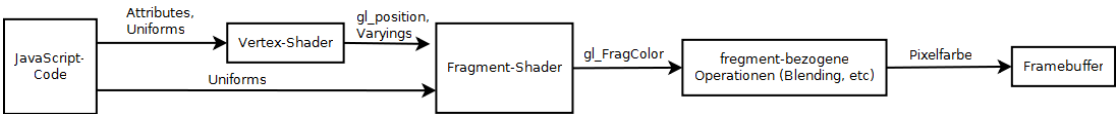


Abbildung 1.1: Vereinfachte Renderpipeline von WebGL - Quelle [Str11]

2 WebGL

2.1 Hardware Unterstützung

Da WebGL auf OpenGL ES 2.0 basiert, verlangt es zwingend nach Hardware, die Vertex- und Fragmentshader unterstützt, sowie passende Treiber. OpenGL ES 2.0 zertifizierte Hardware umfasst Grafikchips der großen Desktop-Hersteller Intel, Nvidia und AMD, die auch bereits einige Jahre alt sein können, da OpenGL ES 2.0 gegen OpenGL 2.0 zertifiziert wurde, zu welcher die Spezifikationen bereits am 7. September 2005 veröffentlicht wurden. Zudem wird die API von einer Reihe von Grafikchips für den mobilen Bereich sowie entsprechenden Betriebssystemen unterstützt.

2.2 Software Unterstützung

WebGL hängt im wesentlichen von der Implementierung des Browsers ab, welche sich je nach Hersteller stark unterscheiden kann.

- Mozilla Firefox unterstützt WebGL seit Version 4.0.
- Google Chrome unterstützt WebGL seit Version 9.
- Apple Safari unterstützt WebGL seit Version 5.1, die Unterstützung muss aber manuell aktiviert werden.
- Opera unterstützt WebGL in bislang keiner offiziellen Version. Ein Entwicklerbuild der Version 11.50 enthält eine experimentelle WebGL Implementierung.
- Microsoft plant keine Unterstützung von WebGL durch den Internet Explorer. Plugins von Drittherstellern ermöglichen die Darstellung von WebGL-Inhalten im Internet Exporer.

Quelle: [Wikb]

Zudem werden OpenGL 2.0-kompatible Treiber für die Hardware verlangt.

2.2.1 ANGLE

Die meisten WebGL-Implementierungen verlangen nach OpenGL Treibern im System. Auf Unix-basierten Plattformen sind diese Treiber normalerweise vorhanden, da OpenGL hier die einzige verfügbare und relevante API zur Erzeugung von

3D-Graphiken ist. Auf Windows-Betriebssystemen ist dies Direct3D, weshalb oft keine OpenGL-Treiber verfügbar sind. Software-Implementierungen wie Mesa3D [Pro] sind zwar möglich, kranken aber oft an schlechter Performance. Um dies zu beheben hat Google das Projekt ANGLE (Almost Native Graphics Layer Engine) [Inca] gestartet, was im wesentlichen OpenGL-Befehlen in Direct3D-9-kompatible umwandelt. Dies ist schneller als eine reine Software-Implementierung, hinkt einer nativen OpenGL-Implementierung an Performance und Unterstützung bestimmter Funktionen (z.b. die Unterstützung von Floating Point Textures) aber hinterher.

2.3 Sicherheitsaspekte

Seit einem Blogpost der Sicherheitsfirma *Context Informatioen Security* [Ltd] im Mai 2011 werden zunehmen die Sicherheitsaspekte von WebGL beleuchtet. Dabei geht es vor allem um die Fragen, ob einem Browser direkter Zugriff auf die Rechenwerke der Grafikkarte, genauer die Shader-Einheiten, gegeben werden darf. Ein Problem könnten die Treiber sein, die mehr auf Geschwindigkeit als auf Sicherheit ausgelegt sind, da bisher der Zugriff auf diese Einheiten nur lokal möglich war. Per WebGL kann aber ein Angriff über das Internet erfolgen. Bis zum Abgabe-Termin dieser Arbeit sind keine erfolgreichen Angriffe bekannt. Bisher kann nur über übermäßig große Geometrie das System zum Absturz (Windows XP und früher) oder zum Reset der Grafiktreiber (Windows seit Vista) gebracht werden. Ein weiteres Problem ist der sog. *cross domain image theft*, der Diebstahl von Daten über eine Domain hinweg. Einige WebGL-Implementierungen erlauben das Laden von lokalen Grafiken. Diese können mit etwas Aufwand von einem WebGL Programm mitgelesen werden. Google Chrome und Chromium Implementierung verlangen dazu aber Parameter *–allow-file-access-from-files*, der manuell spezifiziert werden muss. Firefox verbietet seit Version 5.0 grundsätzlich das Laden von Texturen von anderen Domains als der eigenen des WebGL-Programms.

Da WebGL noch ein sehr junger Standard und wenig auf Sicherheitslücken erforscht ist, kann noch keine abschließende Aussage über die Sicherheit der API gemacht werden. Grundsätzlich birgt jede Erweiterung eines Programms auch neue Angriffsfläche, die in vertrauenswürdigen und lokalen Umgebunden aber keine Rolle spielen. Da das Internet aber tendenziell nicht vertrauenswürdig ist, ist dies ein ernst zu nehmender Aspekt, der in Zukunft weiter Untersucht wird. Die Khronos Gruppe hat bereits Demos zu möglichen Angriffen per Denial-of-Service und Cross-Domain-Image-Theft bereitgestellt.

3 Programmierung

3.1 Ein einfaches Beispiel

Das nachfolgende Beispiel soll den grundlegenden Aufbau einer WebGL-Applikation beschreiben. Das Beispiel ist bewusst einfach gehalten und kommt soweit wie möglich ohne Bibliotheken von Drittanbietern aus. Es basiert auf den ersten WebGL-Lektionen von Giles Thomas [Thoa]

Der erste Teil des Programms ist der HTML-Rumpf, der hier sehr einfach gehalten ist. Listing 3.1 zeigt den sehr einfachen Aufbau. Im öffnenden Body-Tag wird die JavaScript-Funktion angegeben, die nach dem Laden der Datei aufgerufen werden soll, in diesem Fall *webGLStart*. Im Body-Teil wird das Canvas-Element beschrieben, in das später gezeichnet werden soll. Das Element soll dabei 500 Pixel in hoch und breit sein sowie von einer schwarzen, einen Pixel dicken Rahmen haben.

Listing 3.1: HTML Rumpf

```
1 <body onload="webGLStart();">
2 <canvas id="webgl-canvas" style="border: 1px solid;" width="500
      " height="500"></canvas>
3 </body>
```

Im JavaScript-Teil werden einige globale Variablen angelegt. Dies sollte in einer Einsatzapplikation vermieden werden, für dieses einfache Beispiel erspart es aber komplexere und damit tendenziell verwirrende Konstrukte. Zuerst wird Variable *gl* angelegt, die den WebGL-Context darstellt und über welche alle weiteren WebGL spezifischen Funktionen aufgerufen werden. Zudem werden Model-View- und Projektions-Matrizen erstellt. Diese sind vom Typ *mat4*, entnommen aus der Bibliothek **glMatrix** [Jon]. Diese schlanke und auf Performance optimierte Bibliothek stellt Matrizen-Operationen zur Verfügung und ist für WebGL geschrieben, im Gegensatz zu einigen anderen JavaScript-Bibliotheken wie *Sylvester.js*. Als letztes werden noch Puffer für die Geometriedaten erstellt.

Die erste Funktion, gewissermaßen die Hauptfunktion (Listing 3.2) des Programms, ist einfach gehalten und ruft weitere Funktionen auf. In der Funktion wird noch die Farbe, mit der das zu löschende *canvas*-Element überzeichnet werden soll, definiert.

Die erste Hilfsfunktion ist *initGL()*, welche in Listing 3.3 beschrieben wird. Die Funktion initialisiert die Variable *gl* mit den Daten des *canvas*-Elements und definiert den Viewport, in dem später gezeichnet werden soll. Es wird noch eine rudimentäre Fehlerbehandlung durchgeführt, sollte der WebGL-Context nicht verfüg-

Listing 3.2: webGLStart

```
1  function webGLStart(){
2      var canvas = document.getElementById("webgl-canvas");
3      initGL(canvas);
4      initShaders();
5      initBuffers();
6
7      gl.clearColor(1.0, 1.0, 1.0, 1.0);
8      gl.enable(gl.DEPTH_TEST);
9
10     drawScene();
11 }
```

bar sein, wenn der Browser WebGL nicht unterstützt deaktiviert ist.

Listing 3.3: initGL

```
1  function initGL(canvas){
2      try {
3          gl = canvas.getContext("experimental-webgl");
4          gl.viewportWidth = canvas.width;
5          gl.viewportHeight = canvas.height;
6      } catch(e){
7      }
8      if(!gl){
9          alert("WebGL konnte nicht initialisiert werden");
10     }
11 }
```

Als nächstes müssen die Shader initialisiert werden, dies geschieht in der Funktion *initShaders*, beschrieben in Listing 3.4. Das einfach Beispiel benötigt nur zwei einfach Fragment- und Vertex-Shader, die in den Listings 3.5 und 3.6 beschrieben sind. Im Vertex-Shader wird nur die Farbe statisch für das komplette Objekt festgelegt, dies wird später anders gelöst, siehe Abschnitt 3.2.

Zuerst werden mit der Hilfsfunktion *getShader* beide Shader geholt und an den WebGL-Context gebunden. Die Funktion ist in Listing 3.7 abgebildet und wird später beschrieben. Die Shader werden dort auch kompiliert, allerdings sind sie noch nicht ausführbar. Erst nach dem Linken in ein ShaderProgram können sie ausgeführt werden.

Listing 3.4: initShaders

```
1  function initShaders(){
2      var fragmentShader = getShader(gl, "shader-fs");
3      var vertexShader = getShader(gl, "shader-vs");
4
5      shaderProgram = gl.createProgram();
6          gl.attachShader(shaderProgram, vertexShader);
7      gl.attachShader(shaderProgram, fragmentShader);
8      gl.linkProgram(shaderProgram);
9
10     if(!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)){
11         alert("Shader konnten nicht initialisiert werden");
12     }
13
14     gl.useProgram(shaderProgram);
15
16     shaderProgram.vertexPositionAttribute = gl.
           getAttribLocation(shaderProgram, "aVertexPosition");
17     gl.enableVertexAttribArray(shaderProgram.
           vertexPositionAttribute);
18
19     shaderProgram.pMatrixUniform = gl.getUniformLocation(
           shaderProgram, "uPMatrix");
20     shaderProgram.mvMatrixUniform = gl.getUniformLocation(
           shaderProgram, "uMVMatrix");
21 }
```

Listing 3.5: Fragment Shader

```
1 #ifdef GL_ES
2 precision highp float;
3 #endif
4
5 void main(void){
6     gl_FragColor = vec4(0.0, 0.0, 0.0, 1.0);
7 }
```

Listing 3.6: Vertex Shader

```
1 attribute vec3 aVertexPosition;
2
3 uniform mat4 uMVMatrix;
4 uniform mat4 uPMatrix;
5
6 void main(void){
7     gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition,
          1.0);
8 }
```

Die Funktion getShader holt zunächst den jeweiligen Shader anhand der *id* aus dem Dokument und schreibt ihn (oder mehrere vom selben Typ) in eine String-Variable. Es wird ein passender Shader erstellt (hier entweder Fragment- oder Vertex-Shader) und der Shader kompiliert und zurück gegeben.

Die kleine Hilfsfunktion *setMatrixUniforms*, beschrieben in Listing 3.8 bereitet die Matrizen für die Übergabe an die Vertexshader-Einheiten des Grafikchips vor und übergibt sie an diesen, damit die Matrizen-Operationen dort ausgeführt werden.

In der Funktion *initBuffers* (Listing 3.9) werden die Geometriedaten eines Dreiecks und eines Quadrates erzeugt und in die zugehörigen Puffer geschrieben, welche an den Grafikchip übergeben werden, damit sie dort gezeichnet werden können.

Die Vertex-Informationen werden zuerst in einer einfachen JavaScript-Liste gespeichert. Da WebGL, was die vordefinierten Datentypen angeht nur sehr spärliche Möglichkeiten bietet, wird die Liste in eine Array vom Typ *Float32Array* gespeichert. Hier liegen die Informationen aber ohne logische Einteilung vor. Dafür werden in JavaScript die neue Attribute *itemSize* und *numItems* definiert, die die Anzahl der logischen Elemente sowie die Anzahl der Attribute pro Element angeben. Im Falle des Dreiecks sind es drei Vertices mit je drei Attributen für x-, y- und z-Koordinate im Raum. Beim Quadrat sind es entsprechend vier Verticies mit drei Attributen. Intern wird WebGL das Quadrat als zwei Dreiecke behandeln.

In der in Listing 3.10 beschriebenen Funktion *drawScene* wird die Szene gezeichnet. Als erstes wird der Viewport mit den bekannten Daten initialisiert und die Zeichenfläche einfarbig neu gezeichnet.

Anschließend muss die Perspektive festgelegt werden. Ohne diese Angaben würden alle Objekte gleich groß gezeichnet werden, egal wie weit sie von der Kame-

Listing 3.7: getShader

```
function getShader(gl, id){
    var shaderScript = document.getElementById(id);
    if(!shaderScript){
        return null;
    }

    var str = "";
    var k = shaderScript.firstChild;
    while(k){
        if(k.nodeType == 3){
            str += k.textContent;
        }
        k = k.nextSibling;
    }

    var shader;
    if(shaderScript.type == "x-shader/x-fragment"){
        shader = gl.createShader(gl.FRAGMENT_SHADER);
    } else if(shaderScript.type == "x-shader/x-vertex"){
        shader = gl.createShader(gl.VERTEX_SHADER);
    } else {
        return null;
    }

    gl.shaderSource(shader, str);
    gl.compileShader(shader);

    if(!gl.getShaderParameter(shader, gl.COMPILE_STATUS)){
        alert(gl.getShaderInfoLog(shader));
        return null;
    }

    return shader;
}
```

Listing 3.8: setMatrixUniforms

```
1 function setMatrixUniforms(){
2     gl.uniformMatrix4fv(shaderProgram.pMatrixUniform, false,
          pMatrix);
3     gl.uniformMatrix4fv(shaderProgram.mvMatrixUniform, false,
          mvMatrix);
4 }
```

Listing 3.9: initBuffers

```
1  function initBuffers(){
2      triangleVertexPositionBuffer = gl.createBuffer();
3      gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexPositionBuffer
           );
4      var vertices = [
5           0.0,  1.0,  0.0,
6          -1.0, -1.0,  0.0,
7           1.0, -1.0,  0.0
8      ];
9      gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices),
           gl.STATIC_DRAW);
10     triangleVertexPositionBuffer.itemSize = 3;
11     triangleVertexPositionBuffer.numItems = 3;
12
13     squareVertexPositionBuffer = gl.createBuffer();
14     gl.bindBuffer(gl.ARRAY_BUFFER, squareVertexPositionBuffer);
15     vertices = [
16          1.0,  1.0,  0.0,
17         -1.0,  1.0,  0.0,
18          1.0, -1.0,  0.0,
19         -1.0, -1.0,  0.0
20     ];
21     gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices),
           gl.STATIC_DRAW);
22     squareVertexPositionBuffer.itemSize = 3;
23     squareVertexPositionBuffer.numItems = 4;
24 }
```

ra entfernt sind und in welchem Winkel sie dazu stehen. Hier wird ein Gesichtsfeld von 45 Grad angenommen, sowie das Verhältnis zwischen Höhe und Breite des Viewports angegeben. Zudem wird festgelegt, dass Objekte die weniger als 0.1 Einheiten und weiter als 100 Einheiten entfernt sind nicht dargestellt werden sollen. Die Informationen werden in der Perspektive-Matrix gespeichert. Als Identity-Matrix, die die aktuellen Koordinaten zum Zeichnen angibt, wird die Model-View-Matrix verwendet. Sie enthält Informationen zur Position der Objekte und der Kamera.

Dann wird zuerst das Dreieck gezeichnet. Mit einer Einfachen Transformations-Matrix zur Translation werden die momentanen Koordinaten zum zeichnen nach links verschoben. Die Puffer mit den Geometriedaten werden an den WebGL-Context gebunden und die Vertex-Informationen verarbeitet. Anschließend werden die aktuellen Matrizen an den Grafikchip übergeben und der Inhalt des Arrays mit den Vertexinformationen gezeichnet. Danach wird das Quadrat gezeichnet, mit dem Unterschied dass es zuerst ein wenig nach rechts verschoben wird, damit beide Objekte nebeneinander im *canvas*-Element erscheinen.

Für beide Elemente wird der Zeichenmodus angegeben, das Dreieck wird als solches gezeichnet, während das Quadrat als Streifen von Dreiecken (*TRIANGLE_STRIP*) gezeichnet wird, da es intern als zwei aneinanderliegende Dreiecken behandelt wird.

Wenn alles korrekt abgearbeitet wurde, sollte im Browser das Bild eines schwarzen Dreiecks und Quadrates zu sehen ein, wie in Abbildung 3.1 zu sehen.

3.2 Farbe

Im vorherigen Beispiel wurden beide Objekte einfach in einer festen Farbe gezeichnet, die im Fragment Shader festgelegt wurde. Da dieser Shader gemäß der Renderpipeline, abgebildet in Abbildung 1.6, für jedes sichtbare Pixel eines Objektes ausgeführt wird, reichen bereits kleine Änderungen in den Shadern, um einen Farbverlauf zu erzeugen.

Im ersten Teil wurden Variablen vom Typ *uniform* verwendet, einem der beiden Typen, die OpenGL und damit auch WebGL kennt. Der andere ist *varying* und wie der Name bereits erahnen lässt, ist der Unterschied zwischen den Typen, dass uniform-Variablen sich zwischen den Aufrufen nicht ändern, im Gegensatz zu varying-Variablen. Da zuerst im Vertex-Shader verarbeitet werden, muss dieser zuerst angepasst werden, abgebildet in Listing 3.11. Neu hinzu gekommen ist

Listing 3.10: drawScene

```
1  function drawScene(){
2      gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
3      gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
4
5      mat4.perspective(45, gl.viewportWidth / gl.viewportHeight,
           0.1, 100.0, pMatrix);
6      mat4.identity(mvMatrix);
7
8      mat4.translate(mvMatrix, [-1.5, 0.0, -7.0]);
9      gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexPositionBuffer
           );
10     gl.vertexAttribPointer(shaderProgram.
           vertexPositionAttribute, triangleVertexPositionBuffer.
           itemSize, gl.FLOAT, false, 0, 0);
11     setMatrixUniforms();
12     gl.drawArrays(gl.TRIANGLES, 0, triangleVertexPositionBuffer
           .numItems);
13
14     mat4.translate(mvMatrix, [3.0, 0.0, 0.0]);
15     gl.bindBuffer(gl.ARRAY_BUFFER, squareVertexPositionBuffer);
16     gl.vertexAttribPointer(shaderProgram.
           vertexPositionAttribute, squareVertexPositionBuffer.
           itemSize, gl.FLOAT, false, 0, 0);
17     setMatrixUniforms();
18     gl.drawArrays(gl.TRIANGLE_STRIP, 0,
           squareVertexPositionBuffer.numItems);
19 }
```

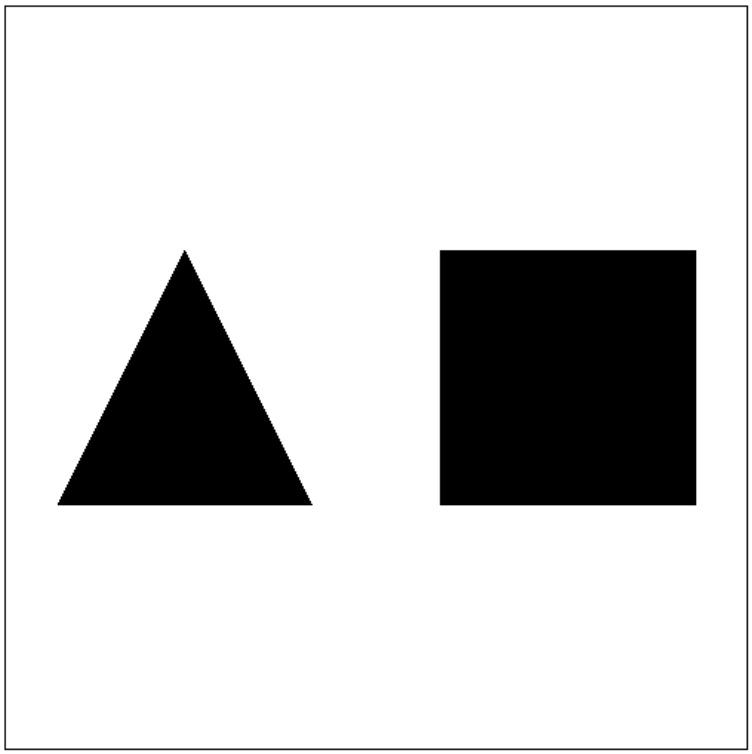


Abbildung 3.1: Einfaches Beispiel in WebGL: ein Dreieck und ein Quadrat

eine Varying-Variable für die aktuelle Farbe sowie eine Zuweisung an ein Vertex-Attribut.

Listing 3.11: Vertex-Shader für Farben

```
1  attribute vec3 aVertexPosition;
2  attribute vec4 aVertexColor;
3
4  uniform mat4 uMVMatrix;
5  uniform mat4 uPMatrix;
6
7  varying vec4 vColor;
8
9  void main(void){
10         gl_Position = uPMatrix * uMVMatrix * vec4(
               aVertexPosition, 1.0);
11         vColor = aVertexColor;
12 }
```

Auch im Fragment-Shader müssen Änderungen vorgenommen werden, diese sind aber noch geringer als im Vertex-Shader. Statt die Farbe statisch festzulegen, wird ebenfalls eine *varying*-Variable erzeugt und ihr Inhalt dem aktuellen Farbwert zugewiesen. Listing 3.12 zeigt den Shader mit den Änderungen.

Listing 3.12: Farbe Fragment-Shader

```
1  #ifdef GL_ES
2  precision highp float;
3  #endif
4
5  varying vec4 vColor;
6
7  void main(void){
8          gl_FragColor = vColor;
9  }
```

In der Initialisierungsfunktion für die Shader muss eine kleine Erweiterung hinzugefügt werden. Nachdem die Positionsdaten des aktuellen Verticies mit dem WebGL-Context verknüpft wurden, wird das selbe mit dem Farbinformationen gemacht. Ausschnitt mit Änderungen der Funktion ist Listing 3.13

Die Werte für die aktuelle Farbe werden wieder in globalen Puffern gespeichert,

Listing 3.13: Farbe initShaders Ausschnitt

```
1  gl.useProgram(shaderProgram);
2
3  shaderProgram.vertexPositionAttribute = gl.getAttribLocation(
       shaderProgram, "aVertexPosition");
4  gl.enableVertexAttribArray(shaderProgram.
       vertexPositionAttribute);
5
6  shaderProgram.vertexColorAttribute = gl.getAttribLocation(
       shaderProgram, "aVertexColor");
7  gl.enableVertexAttribArray(shaderProgram.vertexColorAttribute);
8
9  shaderProgram.pMatrixUniform = gl.getUniformLocation(
       shaderProgram, "uPMatrix");
10 shaderProgram.mvMatrixUniform = gl.getUniformLocation(
       shaderProgram, "uMVMatrix");
```

die neben den Puffern für die Geometriedaten angelegt werden. Siehe dazu Listing 3.14

Listing 3.14: Globale Puffer für Farbdaten

```
1  var triangleVertexColorBuffer;
2  var squareVertexColorBuffer;
```

Analog zu den Geometriedaten werden im Farbpuffer die Farbwerte für die jeweiligen Verticies festgelegt, was den Ecken des Dreiecks entspricht. Der Farbverlauf dazwischen wird von WebGL interpoliert. Farbwerte bestehen immer aus vier Attributen, nämlich den Anteilen für Rot, Blau und Grün sowie der Deckkraft.

Zuletzt muss noch in der Funktion *drawScene* (Listing 3.16) der gefüllte Farbpuffer mit dem WebGL-Context verbunden werden.

Als Resultat sollte ein Dreieck, dass in seinen Ecken Rot, Blau und Grün gefärbt ist mit entsprechenden Farbverläufen dazwischen, sowie ein Quadrat, das oben Blau nach unten hin Rot gefärbt ist angezeigt werden, vergleiche dazu Abbildung 3.2.

Listing 3.15: Ausschnitt initBuffers mit Farbpuffer

```
1  triangleVertexColorBuffer = gl.createBuffer();
2  gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexColorBuffer();
3  var colors = [
4                  1.0, 0.0, 0.0, 1.0,
5                  0.0, 1.0, 0.0, 1.0,
6                  0.0, 0.0, 1.0, 1.0
7                  ];
8  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors), gl.
       STATIC_DRAW);
9  triangleVertexColorBuffer.itemSize = 4;
10 triangleVertexColorBuffer.numItems = 3;
```

Listing 3.16: Ausschnitt DrawScene mit anbinden des Farbpuffers für das Dreieck

```
1  mat4.translate(mvMatrix, [-1.5, 0.0, -7.0]); // Dreieck nach
       links verschieben
2  gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexPositionBuffer);
3  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
       triangleVertexPositionBuffer.itemSize, gl.FLOAT, false, 0,
       0);
4
5  gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexColorBuffer);
6  gl.vertexAttribPointer(shaderProgram.vertexColorAttribute,
       triangleVertexColorBuffer.itemSize, gl.FLOAT, false, 0, 0);
7
8  setMatrixUniforms();
9  gl.drawArrays(gl.TRIANGLES, 0, triangleVertexPositionBuffer.
       numItems);
```

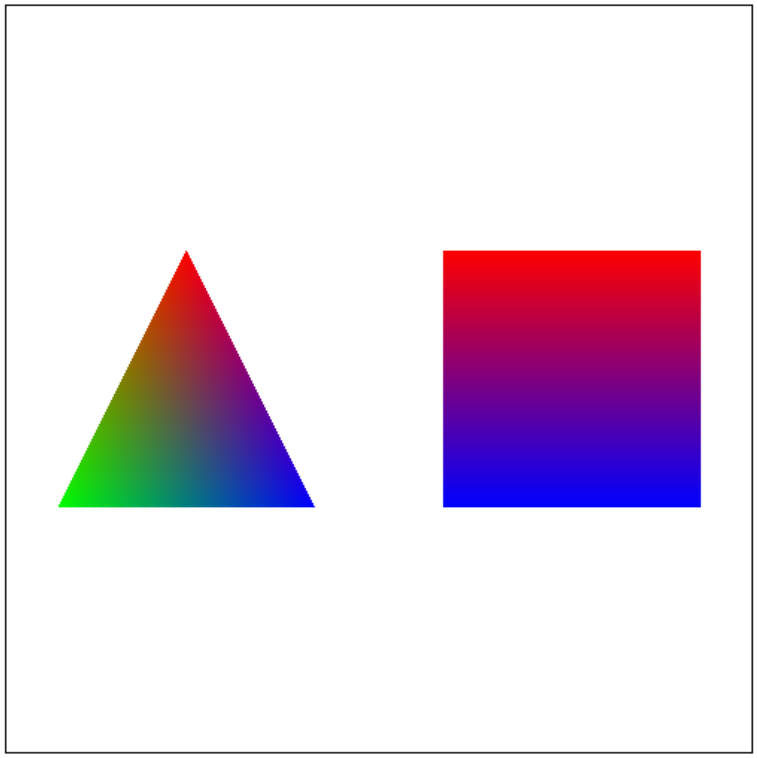


Abbildung 3.2: Farbiges Beispiel in WebGL: ein Dreieck und ein Quadrat

3.3 Animation

Animationen in OpenGL funktionieren nach einem einfachen Muster: jedes mal, wenn sich etwas im Bild ändert, wird es neu gezeichnet. Bisher wird im Beispiel alles von der Funktion *drawScene* gezeichnet, und zwar einmalig. Als Konsequenz muss die Funktion nun in einer Art Endlosschleife aufgerufen werden, für jede Änderung am Bild. Im folgenden Abschnitt wird eine einfache Animation anhand einer Rotation der beiden bekannten Objekte, dem Dreieck und dem Quadrat, beschrieben.

Zunächst wird der Aufruf der *drawScene*-Funktion geändert. In der Funktion *webGLStart* wird statt *drawScene* die neue Funktion *tick* aufgerufen, die eine implizite Endlosschleife erzeugt. Die Funktion ist in Listing 3.17 abgebildet. Sie ruft drei Funktionen auf: *requestAnimFrame* mit der Anweisung, *tick* zu starten wenn benötigt, also wenn die Szene neu gezeichnet werden soll. Weiterhin werden *drawScene* und eine weitere Funktion *animate* aufgerufen, welche später weiter beschrieben werden. *requestAnimFrame* ist eine Funktion aus der von Google entwickelten Bibliothek *webgl-utils* [Incb], die Vereinfachungen bei der Initialisierung sowie diese Hilfsfunktion für Animationen bereitstellt. Der Hintergrund für die Nutzung der Bibliothek ist, dass die Funktion *RequestAnimationFrame* von den Browsern mit unterschiedlichen Namen implementiert werden: In Firefox heißt die Funktion *mozRequestAnimationFrame*, in WebKit-basierten Browsern wie Chrome und Safari dagegen *webkitRequestAnimationFrame*. Für eine zukünftige Version ist angedacht dies zu vereinheitlichen. Bis zur Umsetzung ist man für eine browserunabhängige Lösung auf diesen oder ähnliche Umwege angewiesen.

Die Nutzung von *RequestAnimationFrame* hat einen entscheidenden Vorteil gegenüber anderen Lösungen: Die Animation wird nur neu gezeichnet, wenn das Browserfenster (oder Tab) geöffnet und sichtbar ist. Eine alternative Lösung würde die JavaScript-Funktion *setInterval* nutzen, was aber dazu führen würde, dass die Animation immer neu gezeichnet wird, auch wenn der Browser gerade nicht im Vordergrund ist und damit Ressourcen verschwendet werden. In den offiziellen FAQ der Khronos-Gruppe wird von dieser Lösung abgeraten und stattdessen empfohlen *RequestAnimationFrame* zu verwenden.

Für die Animation selber werden zwei globale Variablen definiert, die jeweils den aktuellen Grad der Rotation beinhalten. Diese Art der Definition ist eigentlich nicht zu empfehlen, für dieses sehr kleine Beispiel vermeidet es aber kompliziertere Lösungen, die in späteren Abschnitten genutzt werden und einen sauberen

Listing 3.17: Hilfsfunktion tick für Animation

```
1 function tick(){
2         requestAnimFrame(tick);
3         drawScene();
4         animate();
5 }
```

Programmierstil erlauben.

Zuerst werden die Änderungen an der Funktion drawScene, dargestellt in Listing 3.18 für das Dreieck, besprochen: Es kommt für die Rotation wieder eine Matrizen-Operation zum Einsatz, die eine Rotation um die vertikale Achse des Objektes definiert. Die Hilfsfunktion *degToRad* berechnet einen Winkel von Grad in das Bogenmaß um, welches hier benötigt wird. Der Programmteil für das Quadrat ist dazu analog, mit der Unterschied, dass eine Rotation um die horizontale Achse erstellt wird.

Die Funktionen *mvPushMatrix* und *mvPopMatrix* bilden einen Stack nach, auf dem die aktuelle Model-View-Matrix zwischengespeichert wird. Der Grund ist, dass in OpenGL wie in WebGL immer von relativen Positionen zur letzten weiter gezeichnet wird. Da hier allerdings die Model-View-Matrix verändert wird, würde sich auch die Koordinaten, an dem das Quadrat gezeichnet wird, verändern. Um das zu verhindern wird die Model-View-Matrix auf einem Stack zwischengespeichert, der ebenfalls global definiert ist. Die Definition der Funktion ist in den Listings 3.19 und 3.20 dargestellt.

Es fehlt noch die Funktion *animate*, die die Änderungen der Winkel berechnet. Hier wurde eine Lösung gewählt, die eine Rotation um einen fixen Wert bei jedem Aufruf erzeugt. Dadurch wird die Rotation nicht schneller auf leistungsstärkeren Geräten, auf langsameren wirkt die Animation etwas abgehackt, aber nicht langsamer. Der fixe Wert ist die Zeit seit dem letzten Aufruf der Funktion. Hier wird genauer das Dreieck um 90 Grad und das Quadrat um 75 Grad pro Sekunde rotiert. Für dieses kleine Beispiel wäre das nicht notwendig, aber es ist eine bewährte Methode, die in ihrer Komplexität einfache Lösungen nur margial übersteigt.

Als Ergebnis sollte das Dreieck um seine vertikale und das Quadrat um seine horzintale Achse rotieren. Ein Beispiel ist ein Abbildung 3.3 abgebildet.

Listing 3.18: Animation mit Auschnitt der Änderungen an drawScene

```
1  mat4.translate(mvMatrix, [-1.5, 0.0, -7.0]);
2
3  mvPushMatrix();
4  mat4.rotate(mvMatrix, degToRad(rTri), [0, 1, 0]);
5
6  gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexPositionBuffer);
7  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
       triangleVertexPositionBuffer.itemSize, gl.FLOAT, false, 0,
       0);
8
9  gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexColorBuffer);
10 gl.vertexAttribPointer(shaderProgram.vertexColorAttribute,
       triangleVertexColorBuffer.itemSize, gl.FLOAT, false, 0, 0);
11
12 setMatrixUniforms();
13 gl.drawArrays(gl.TRIANGLES, 0, triangleVertexPositionBuffer.
       numItems);
14 gl.drawArrays(gl.TRIANGLES, 0, triangleVertexPositionBuffer.
       numItems);
15
16 mvPopMatrix();
```

Listing 3.19: Animation Hilfsfunktion mvPushMatrix

```
1 function mvPushMatrix(){
2     var copy = mat4.create();
3     mat4.set(mvMatrix, copy);
4     mvMatrixStack.push(copy);
5 }
```

Listing 3.20: Animation Hilfsfunktion mvPopMatrix

```
1 function mvPopMatrix(){
2     if(mvMatrixStack.length == 0){
3         throw "Stack␣ist␣bereits␣leer!";
4     }
5     mvMatrix = mvMatrixStack.pop();
6 }
```

Listing 3.21: Animation Funktion animate

```
1  function animate(){
2          var timeNow = new Date().getTime();
3          if(lastTime != 0 ){
4                  var elapsed = timeNow - lastTime;
5
6                  rTri += (90 * elapsed) / 1000.0;
7                  rSquare += (75 * elapsed) / 1000.0;
8          }
9          lastTime = timeNow;
10 }
```

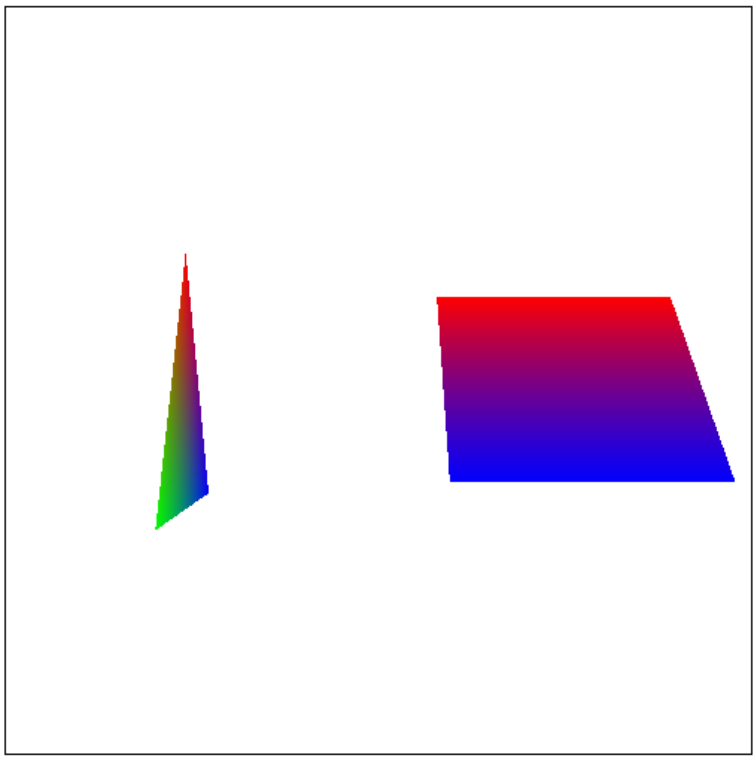


Abbildung 3.3: Animiertes Beispiel in WebGL: ein Dreieck und ein Quadrat

3.4 Dreidimensionale Objekte

Bisher waren alle Objekte nur zweidimensional und hätte (bis auf die Animation) auch mit anderen Methoden gezeichnet werden können. Jetzt sollen die Objekte dreidimensional werden, statt eines Dreiecks eine Pyramide mit einem Dreieck als Grundfläche sowie ein Würfel. Dabei sollen auf jeder Fläche der Pyramide wieder die Farbverläufe zu sehen sein und jede Seite des Würfels soll eine andere Farbe haben.

Für das Zeichnen der Pyramide sind keine weiteren Änderungen erforderlich, ausgenommen die zusätzlichen Geometriedaten, da die Pyramide aus vier Seiten mit je drei Verticies besteht, die sich aber überschneiden. Die neue Definition der JavaScript-Liste mit den Koordinaten in der Funktion *initBuffers* ist in Listing 3.22 abgebildet. Ebenso müssen die Farben für jede Seite definiert werden. Der Einfachheit halber werden in diesem Beispiel nur die bisherigen Angaben viermal in eine Liste und das Array geschrieben.

Listing 3.22: Definition der Verticies für Pyramide

```
var vertices = [
        // Seitenangaben jeweils von der Kamera aus
        // Vorderseite
         0.0,  1.0,  0.0,
        -1.0, -1.0,  1.0,
        1.0, -1.0,  1.0,

        //Rechte Seite
         0.0,  1.0,  0.0,
         1.0, -1.0,  1.0,
         1.0, -1.0, -1.0,

        // Rueckseite
         0.0,  1.0,  0.0,
         1.0, -1.0, -1.0,
        -1.0, -1.0, -1.0,

        // Linke Seite
         0.0,  1.0,  0.0,
        -1.0, -1.0, -1.0,
        -1.0, -1.0,  1.0
];
```

Für den Würfel sind weitere Überlegungen notwendig: Grundsätzlich gibt es drei Varianten, der Würfel zu zeichnen: In einem einzelnen *TRIANGLE_STRIP*. Diese sehr effiziente Methode funktioniert aber nur, wenn die Farb- bzw. Texturinformationen für jede Seite gleich sind, was in diesem Beispiel nicht der Fall ist. Eine etwas unschöne, aber mögliche Lösung wäre das Erstellen von sechs separaten Quadraten, die gezeichnet einen Würfel ergeben, aber keine logische Verbindung haben. Die gebräuchlichere Methode ist es, einen Würfel bestehend aus sechs Quadraten zu definieren, welcher dann in einem Schritt gezeichnet wird. Die Methode ähnelt stark der ersten mit *TRIANGLE_STRIP*, aber so lassen sich verschiedene Farben für jede Seite definieren. Die Definition der Verticies ist in Listing 3.23 abgebildet.

Auch die Farben müssen für den Würfel neu definiert werden. Da für jede Seite und damit für jede der vier Verticies die gleiche Farbe festgelegt wird und der Programmcode nicht ausufert, wird die JavaScript-Liste mit einer Schleife gefüllt, nachdem zuvor in einem Array die Farben festgelegt wurden. Dabei werden die Informationen einfach vier mal hintereinander in eine temporäre Liste geschrieben, aus der dann das Array für den WebGL-Context erstellt wird. Siehe dazu Listing 3.24.

Eine Problematik ist die Struktur des Würfels: Er besteht aus sechs Seite, welche aber jeweils aus zwei Dreiecken bestehen sollen. Es muss noch festgelegt werden, welche Verticies zu welchem Dreieck gehören. Dazu wird ein weiterer Puffer, *cubeVertexIndexBuffer*, mit den nötigen Informationen über die Zusammensetzung der Dreiecke gefüllt. Diese Puffer wird auch an den WebGL-Context gebunden und die logische Struktur festgelegt. Der JavaScript-Code dazu steht in Listing 3.25.

Da nun kein *TRIANGLE_STRIPE*, sondern einzelne Dreiecke gezeichnet werden, muss in der Funktion *drawScene* der Aufruf zum Zeichnen geändert werden. Statt den Inhalt eines Arrays werden nun logische Elemente gezeichnet, wofür der entsprechende Puffer mit den Zugehörigkeiten bekannt sein muss. Siehe dazu Listing 3.26

Am Ende sollte eine rotierende Pyramide und Würfel zu sehen sein, Beispiel in Abbildung 3.4.

3.5 Texturen

Bislang wurden die Objekte nur mit einfachen Farben gezeichnet. In der Praxis kommt es aber häufig vor, dass Texturen auf die Objekte aufgetragen werden sollen.

Listing 3.23: Definition der Verticies des Würfels

```
vertices = [
        // Vorderseite
        -1.0, -1.0,  1.0,
         1.0, -1.0,  1.0,
         1.0,  1.0,  1.0,
        -1.0,  1.0,  1.0,

        // Rueckseite
        -1.0, -1.0, -1.0,
        -1.0,  1.0, -1.0,
         1.0,  1.0, -1.0,
         1.0, -1.0, -1.0,

        // Obere Seite / Deckel
        -1.0,  1.0, -1.0,
        -1.0,  1.0,  1.0,
         1.0,  1.0,  1.0,
         1.0,  1.0, -1.0,

        // Untere Seite / Boden
        -1.0, -1.0, -1.0,
         1.0, -1.0, -1.0,
         1.0, -1.0,  1.0,
        -1.0, -1.0,  1.0,

        // Rechte Seite
         1.0, -1.0, -1.0,
         1.0,  1.0, -1.0,
         1.0,  1.0,  1.0,
         1.0, -1.0,  1.0,

        // Linke Seite
        -1.0, -1.0, -1.0,
        -1.0, -1.0,  1.0,
        -1.0,  1.0,  1.0,
        -1.0,  1.0, -1.0
];
```

Listing 3.24: Definition der Farben für Würfel

```
cubeVertexColorBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, cubeVertexColorBuffer);
var colors = [
                [ 1.0, 0.0, 0.0, 1.0 ], // Vorderseite
                [ 0.0, 1.0, 0.0, 1.0 ], // Rueckseite
                [ 0.0, 0.0, 1.0, 1.0 ], // Obere Seite / Deckel
                [ 1.0, 0.0, 1.0, 1.0 ], // Untere Seite / Boden
                [ 0.0, 1.0, 1.0, 1.0 ], // Rechte Seite
                [ 1.0, 1.0, 0.0, 1.0 ]  // Linke Seite
                ];

var unpackedColors = [];
for (var i in colors){
        var color = colors[i];
        for (var j=0; j < 4; j++){
                unpackedColors = unpackedColors.concat(color);
        }
}
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(unpackedColors)
    , gl.STATIC_DRAW);
cubeVertexColorBuffer.itemSize = 4;
cubeVertexColorBuffer.numItems = 24;
```

Listing 3.25: Festlegung der Zugehörigkeit der Verticies zu Dreiecken

```
cubeVertexIndexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, cubeVertexIndexBuffer);
var cubeVertexIndices = [
        0, 1, 2,                0, 2, 3,        // Vorderseite
        4, 5, 6,                4, 6, 7,        // Rueckseite
        8, 9, 10,               8, 10, 11,      // Obere Seite
           / Deckel
        12, 13, 14,             12, 14, 15,     // Untere Seite
            / Boden
        16, 17, 18,             16, 18, 19,     // Rechte Seite
        20, 21, 22,             20, 22, 23      // Linke Seite
];
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(
    cubeVertexIndices), gl.STATIC_DRAW);
cubeVertexIndexBuffer.itemSize = 1;
cubeVertexIndexBuffer.numItems = 36;
```

Listing 3.26: Neue Aufrufe zum zeichnen des Würfels

```
1 gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, cubeVertexIndexBuffer);
2 setMatrixUniforms();
3 gl.drawElements(gl.TRIANGLES, cubeVertexIndexBuffer.numItems,
      gl.UNSIGNED_SHORT, 0);
```

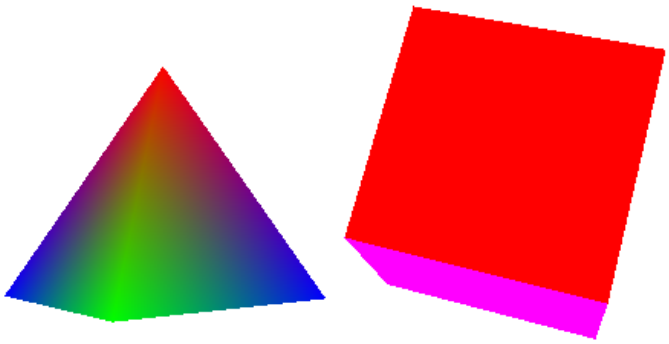


Abbildung 3.4: Dreidimensionales Beispiel in WebGL: eine Pyramide und ein Würfel

Um das Beispiel zu vereinfachen wird die Pyramide aus dem vorherigen Beispiel entfernt, nur der Würfel soll mit dem Logo der HTW Aalen texturiert werden. Dazu wird in der Hauptfunktion *webGLStart* eine neue Funktion aufgerufen, *initTexture* (Code in Listing 3.27), welche die Initialisierung der Textur übernimmt. Es wird das globale Objekt *htwTexture* genutzt, in einer Applikation für den Betrieb sollten lokale Variablen genutzt werden. Zuerst wird eine Textur gemäß des WebGL-Kontextres erstellt, dann wird der Variable ein neues Attribut *image* hinzugefügt, über das die Textur innerhalb des JavaScript-Codes angesprochen wird. Das Attribute *onload* definiert eine Call-Back-Funktion, die *handleLoadedTexture* ausführt sobald die Textur erstellt wurde.

Listing 3.27: initTexture

```
1 function initTexture(){
2         htwTexture = gl.createTexture();
3         htwTexture.image = new Image();
4         htwTexture.image.onload = function(){
5                 handleLoadedTexture(htwTexture)
6         }
7         htwTexture.image.src = "htwlogo256.png";
8 }
```

Die Funktion *handleLoadedTexture*, beschrieben in Listing 3.28 initialisiert die Textur gemäß des WebGL-Kontextes. Zuerst wird die aktuelle Textur festgelegt, ähnlich den *bindBuffer*-Operationen. Dann wird die Textur entlang der Y-Achse gespiegelt, um die Unterschiede im Koordinatensystem auszugleichen: Das Lokale Koordinatensystem ist ein Linkssystem, eine 2D-Grafik wird aber in einem Rechtssystem beschrieben. Die Funktion *texImage2D* sorgt dafür, dass die Grafik im passenden Format in den Grafikspeicher geladen wird. Mit der Funktion *texParameteri* werden die Parameter für die Vergrößerung oder Verkleinerung der Textur festgelegt. Diese werden später im Abschnitt 3.7, Texturfilter, näher erleutert.

Bisher wurde in der Funktion *initBuffer* die Farben an den jeweiligen Vertecies definiert. Für Texturen muss eine ähnliche Definition gemacht werden, da festgelegt werden muss, wie die Textur auf der Fläche ausgerichtet sein soll. Dies ähnelt der bisherigen Definition stark und wird im Listing 3.29 beschrieben. Der größte Unterschied ist, dass die Anzahl der Elemente sich reduziert hat, da nur noch zwei für X- und Y-Koordinate der Textur benötigt werden. Dabei wird angenommen, dass die Textur 1.0 Einheiten breit und hoch ist, woraus resultiert dass die Koordinate

Listing 3.28: handleLoadedTexture

```
1 function handleLoadedTexture(texture){
2         gl.bindTexture(gl.TEXTURE_2D, texture);
3         gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
4         gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.
              UNSIGNED_BYTE, texture.image);
5         gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER,
              gl.NEAREST);
6         gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
              gl.NEAREST);
7         gl.bindTexture(gl.TEXTURE_2D, null);
8 }
```

0.0 / 0.0 links unten und 1.0 / 1.0 rechts oben darstellt. Eine Modifikation des Codes kann für eine Kachelung der Textur genutzt werden: Wird als Maximum z.b. 2.0 statt 1.0 angegeben, wird die Textur in diese Richtung zweimal nebeneinander dargestellt. Dies findet in einem späteren Projekt Verwendung.

Ähnlich Änderungen müssen auch in der Funktion *drawScene* (Code in Listing 3.30) vorgenommen werden, in der der Puffer für Farbattribute durch den für die Textur-Koordinaten ersetzt wird. Zusätzlich muss noch die aktuelle Textur und der zugehörige Textursampler definiert werden. WebGL kann bis zu 32 Texturen gleichzeitig handhaben, die mit *TEXTURE0* bis *TEXTURE31* benannt sind. Zu beachten ist, dass eine Textur immer eine Kantenlängen hat, die einer Zweierpotenz entspricht, andernfalls wird die Textur schlicht ignoriert.

Der Sampler repräsentiert die Textur für die Shader, dementsprechend müssen auch Änderungen an den Shadern durchgeführt werden.

Im Vertex-Shader wird die Farbvariable durch eine Texture-Variable ersetzt, sonst gibt es keine Änderungen, siehe Listing 3.32.

Im Fragment Shader, abgebildet in Listing 3.31 wird der Sampler definiert. Die Farbe eines Fragments wird nun aus den Daten der Textur ermittelt. Die Koordinatenbezeichnung s und t sind hier üblich, sie repräsentieren die Koordinaten x und y der Textur. Die unterschiedliche Benennung dient zur Unterscheidung vom Globalen Koordinaten System (mit X-,Y- und Z-Koordinaten) zum lokalen der Textur (mit S und T). Die Koordinaten im globalen Koordinatensystem werden ausgehend von den Vertecies linear interpoliert.

Im Canvas-Element sollte nun ein rotierender Würfel mit dem Logo der HTW Aalen sein, siehe dazu Beispiel 3.5. Auffällig ist, dass die Textur in der Bewegung

Listing 3.29: Festlegung der Texturkoordinaten in initBuffer

```
1  cubeVertexTextureCoordBuffer = gl.createBuffer();
2  gl.bindBuffer(gl.ARRAY_BUFFER, cubeVertexTextureCoordBuffer);
3
4  var textureCoords = [
5          // Vorderseite
6          0.0, 0.0,
7          1.0, 0.0,
8          1.0, 1.0,
9          0.0, 1.0,
10
11         // Rueckseite
12         1.0, 0.0,
13         1.0, 1.0,
14         0.0, 1.0,
15         0.0, 0.0,
16
17         // Obere Seite / Deckel
18         0.0, 1.0,
19         0.0, 0.0,
20         1.0, 0.0,
21         1.0, 1.0,
22
23         // Untere Seite / Boden
24         1.0, 1.0,
25         0.0, 1.0,
26         0.0, 0.0,
27         1.0, 0.0,
28
29         // Rechte Seite
30         1.0, 0.0,
31         1.0, 1.0,
32         0.0, 1.0,
33         0.0, 0.0,
34
35         // Linke Seite
36         0.0, 0.0,
37         1.0, 0.0,
38         1.0, 1.0,
39         0.0, 1.0
40 ];
41 gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(textureCoords),
       gl.STATIC_DRAW);
42 cubeVertexTextureCoordBuffer.itemSize = 2;
43 cubeVertexTextureCoordBuffer.numItems = 24;
```

Listing 3.30: Ausschnitt drawScene mit Angaben zu Texturen

```
1  gl.bindBuffer(gl.ARRAY_BUFFER, cubeVertexTextureCoordBuffer);
2  gl.vertexAttribPointer(shaderProgram.textureCoordAttribute,
       cubeVertexTextureCoordBuffer.itemSize, gl.FLOAT, false, 0,
       0);
3
4  gl.activeTexture(gl.TEXTURE0);
5  gl.bindTexture(gl.TEXTURE_2D, htwTexture);
6  gl.uniform1i(shaderProgram.samplerUniform, 0);
```

Listing 3.31: Fragement-Shader für Texturen

```
1  #ifdef GL_ES
2  precision highp float;
3  #endif
4
5  varying vec2 vTextureCoord;
6
7  uniform sampler2D uSampler;
8
9  void main(void){
10     gl_FragColor = texture2D(uSampler, vec2(vTextureCoord.s,
           vTextureCoord.t));
11 }
```

Listing 3.32: Vertex-Shader für Texturen

```
1  attribute vec3 aVertexPosition;
2  attribute vec2 aTextureCoord;
3
4  uniform mat4 uMVMatrix;
5  uniform mat4 uPMatrix;
6
7  varying vec2 vTextureCoord;
8
9  void main(void){
10     gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition,
           1.0);
11          vTextureCoord = aTextureCoord;
12 }
```

flimmert, gut zu sehen an den Kanten des Würfels. Mögliche Lösungen werden später im Abschnitt 3.7 behandelt.

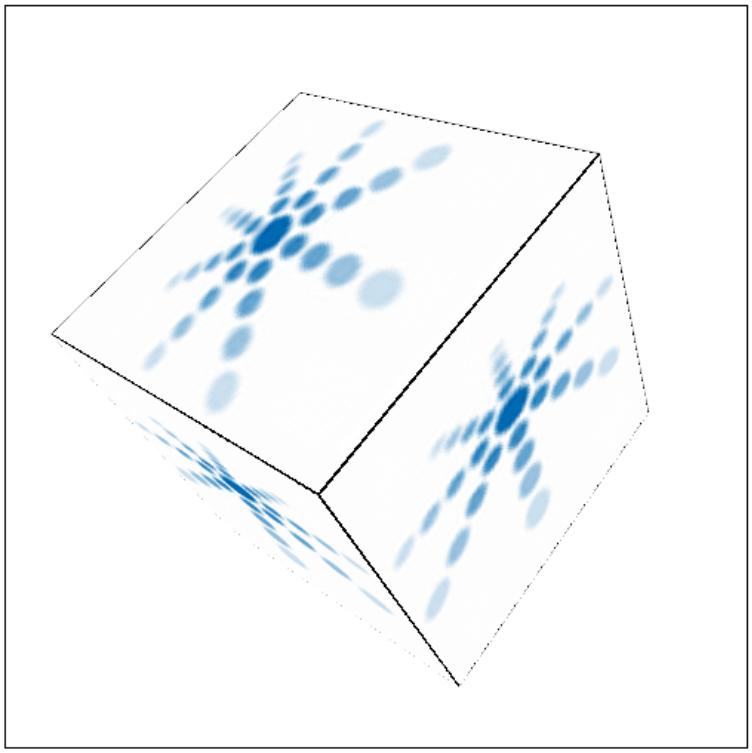


Abbildung 3.5: Texturierte Würfel

3.6 Tastatur und Maus Eingaben

Ein Merkmal von WebGL gegenüber anderen Technologie zum 3D-Rendering im Webbrowser sind die Möglichkeiten zur Interaktion. Statt nur ein vorgefertiges Programm ablaufen zu lassen, kann der User über Maus und Tastatur in des Programm eingreifen, die Position der Kamera oder andere Parameter ändern. In WebGL werden dafür JavaScript-Events verwendet. Das bisherige Beispiel soll dahingehend verändert werden, dass der User mit der Tastatur die Rotations-Richtung und -Geschwindigkeit steuern kann, sowie die Entfernung der Kamera zum Würfel verändern. Zusätzlich soll der User mit der Maus den Würfel drehen können.

3.6.1 Tastatur

Zuerst werden weitere Variablen benötigt, die die aktuelle Rotation in X- und Y-Richtung speichern sowie die Geschwindigkeitsänderung. Zudem wird eine Variable für den aktuellen Kameraabstand benötigt. Dabei genügt eine einzige, da die Kamera sich auf der Z-Achse befindet und zum Schnittpunkt der Achsen blickt. Tasten werden in JavaScript anhand von festgelegten Identifiern bestimmt, die aktuell gedrückten Tasten werden in einen Hashtable geschrieben und ausgewertet. Die Definitionen sind in Listing 3.33.

Listing 3.33: Globale Variablen für Tastatursteuerung

```
1 var xRot = 0;
2 var xSpeed = 0;
3 var yRot = 0;
4 var ySpeed = 0;
5
6 var z = -5;
7
8 var currentlyPressedKeys = {};
```

Zur Erkennung eines Tastendrucks sind in JavaScript zwei Events definiert, *onkeydown* für das runter drücken und *onkeyup* für das loslassen einer Taste. Grundsätzlich müssen zwei Arten von Verhalten bei einem Tastendruck möglich sein: Ein einzelner Druck soll eine einzelne Aktion auslösen. Wenn eine Taste länger gedrückt wird, z.b. um die Kamera zu verschieben, erwartet der User, dass die Kamera um so weiter verschoben wird, je länger er die Taste gedrückt hält. Es werden zwei Funktion benötigt, die jeweils den Status bestimmen, ob eine Taste gedrückt ist oder nicht. Zusätzlich wird noch eine Taste abgefragt, die den Würfen in seinen Ursprungszustand zurück versetzt. 3.34

Der Hashtable kommt in einer weitere Funktion zum Einsatz, in der die Richtungs- und Geschwindigkeitsparameter ausgewertet werden. Es kommt ein Hashtable zum Einsatz, da es auch möglich sein soll, mehrere Tasten gleichzeitig drücken zu können, um eine Rotation um X- und Y-Achse zu Ermöglichen, was sich z.b. von einer einfachen Texteingabe erheblich unterscheidet. Die Auswertung besteht aus einer Reihe von Abfrage, abgebildet in Listing 3.35. Die Funktion wird in der Funktion *tick* aufgerufen, damit die Tastendrücke zeitnah ausgewertet werden.

Damit die Aktion auch auf dem Bildschirm umgesetzt werden, müssen die Funktionen *animate* und *drawScene* angepasst werden.

Listing 3.34: Funktion zur Bestimmt einer gedrückten Taste

```
1  function handleKeyDown(event){
2          currentlyPressedKeys[event.keyCode] = true;
3
4          if(String.fromCharCode(event.keyCode) == "R"){
5                  xRot = 0;
6                  xSpeed = 0;
7                  yRot = 0;
8                  ySpeed = 0;
9                  // Reset der Drehung per Maus tut noch nicht
10         }
11 }
12
13 function handleKeyUp(event){
14         currentlyPressedKeys[event.keyCode] = false;
15 }
```

In *animate* (siehe Listing 3.36) wird die Art der Berechnung der aktuellen Rotation geändert, so dass der Würfel schneller rotiert, je länger eine Taste in eine Richtung gedrückt wird. Abbremsen wird durch das Drücken der Taste in die Gegenrichtung realisiert.

In *drawScene*, Code-Ausschnitt dazu in Listing 3.37, wird die Rotation in zwei Operation geteilt, eine für X- und eine für Y-Achse, bei der Anhand der aktuellen Rotation und der Model-View-Matrix die Rotation umgesetzt wird. Des weiteren wird hier der Abstand der Kamera zum Würfel festgelegt.

3.6.2 Maus

Die Umsetzung der Drehung mit der Maus erfolgt, ähnlich wie bei den Tastatureingaben, über JavaScript-Events, die bereits definiert sind. Auch hier ist entscheidend, ob die primäre Maustaste gedrückt ist oder nicht. Das vereinfacht die Handhabung und gibt dem User mehr Gefühl dafür, dass er den Würfel dreht.

Wieder werden globale Variablen definiert, die den Status der Maustaste sowie die Position der letzten Aktion festhalten. Da die Drehung des Würfels erhalten bleiben soll, wird eine weiter Rotations-Matrix definiert, siehe dazu Listing 3.38.

Die Handler-Funktion für das gedrück halten der Taste sind weitestgehend identisch zur denen für die Tastatureingabe, abgebildet in Listing 3.39. Als einziger Unterschied wird die Position im Canvas, in der das Maus-Event ausgelöst wurde,

Listing 3.35: Auswertung der Tastendrücke über den Hashtable

```
function handleKeys(){
        if(currentlyPressedKeys[33]){
                // Bild auf
                z -= 0.05;
        }

        if(currentlyPressedKeys[34]){
                // Bild ab
                z += 0.05;
        }

        if(currentlyPressedKeys[37]){
                // Pfeil links
                ySpeed -= 1;
        }

        if(currentlyPressedKeys[39]){
                // Pfeil rechts
                ySpeed += 1;
        }

        if(currentlyPressedKeys[38]){
                // Pfeil hoch
                xSpeed -= 1;
        }

        if(currentlyPressedKeys[40]){
                // Pfeil runter
                xSpeed += 1;
        }
}
```

Listing 3.36: Animate mit Änderungen für Tastatureingaben

```
function animate(){
        var timeNow = new Date().getTime();
        if(lastTime != 0 ){
                var elapsed = timeNow - lastTime;
                xRot += (xSpeed * elapsed) / 1000.0;
                yRot += (ySpeed * elapsed) / 1000.0;
        }
        lastTime = timeNow;
}
```

Listing 3.37: Ausschnitt drawScene mit Änderungen für Tastatureingaben

```
mat4.translate(mvMatrix, [0.0, 0.0, z]);

mat4.rotate(mvMatrix, degToRad(xRot), [1, 0, 0]);
mat4.rotate(mvMatrix, degToRad(yRot), [0, 1, 0]);
```

Listing 3.38: Globale Variablen für Mausinteraktion

```
var mouseDown = false;
var lastMouseX = null;
var lastMouseY = null;

var cubeRotationMatrix = mat4.create();
mat4.identity(cubeRotationMatrix);
```

in die globalen Variablen gespeichert.

Listing 3.39: Handlerfunktionen für Maus

```
1 function handleMouseDown(event) {
2     mouseDown = true;
3     lastMouseX = event.clientX;
4     lastMouseY = event.clientY;
5 }
6
7 function handleMouseUp(event) {
8     mouseDown = false;
9 }
```

Die Funktionalität steckt in der Funktion *handleMouseMove*, Code siehe Listing 3.40. Um den Grad der Drehung zu bestimmen, wird das Delta zwischen der letzten und der aktuellen Position eines Maus-Events bestimmt, um welches dann der Würfel rotiert wird. Dies geschieht unabhängig von der Rotation per Tastatur und wird in der *cubeRotationMatrix* gespeichert, damit die Position nach dem Loslassen der Maustaste erhalten bleibt.

Da hier bereits die Drehung stattfindet, muss die Funktion *animate* nicht abgeändert werden. In der Funktion *drawScene* wird die Model-View-Matrix mit der aktuellen *cubeRotationMatrix* multipliziert, damit die Änderung auch auf dem Bildschirm umgesetzt wird.

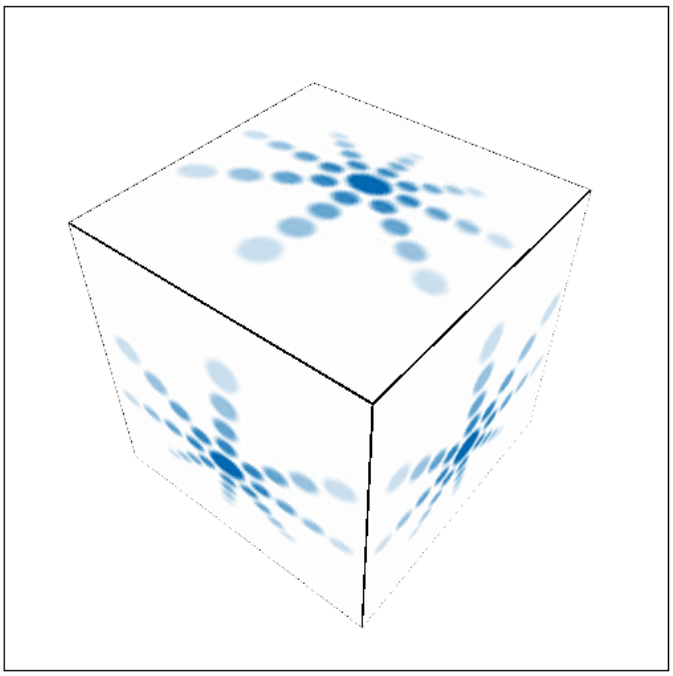
3.7 Texturfilterung

Wie bereits am Ende von Abschnitt 3.5 gezeigt, flimmern Texturen, wenn sich der Sichtwinkel auf die ändert. Um dies zu vermeiden werden Texturfilter eingesetzt. Zur bessere Verständnis der Effekte können die drei behandelten Filter per Tastendruck durchgeschaltet werden.

Die Fehler an den Texturen entstehen, wenn die Texturen auf den Fläche skaliert werden, was im Prinzip immer vorkommt. In diesem Beispiel werden die Textur auf drei unterschiedliche Arten gefiltert und die drei Varianten in ein globales Array *htwTextures* geschrieben. Die Funktion *handleLoadedTextures* ändert sich im Kern kaum, allerdings werden die meisten Operationen dreimal ausgeführt, einmal für jede Variante. Dabei wird der Filter-Modus unterschieden.

Listing 3.40: Funktion zur Umsetzung der Mausaktionen

```
function handleMouseMove(event){
    if(!mouseDown){
        return;
    }

    var newX = event.clientX;
    var newY = event.clientY;

    var deltaX = newX - lastMouseX
        var deltaY = newY - lastMouseY;

    var newRotationMatrix = mat4.create();
    mat4.identity(newRotationMatrix);

    mat4.rotate(newRotationMatrix, degToRad(deltaX / 10), [0,
        1, 0]);
    mat4.rotate(newRotationMatrix, degToRad(deltaY / 10), [1,
        0, 0]);

    mat4.multiply(newRotationMatrix, cubeRotationMatrix,
        cubeRotationMatrix);

    lastMouseX = newX
    lastMouseY = newY;
}
```



Steuerung:

Bild auf / Bild ab für Zoom
Pfeiltaten um Würfel zu rotieren - je länger die Taste gedrückt um so schneller die Rotation.
R um den Würfen in seine Ursprungsposition zu setzten.

Auch mit der Maus drehbar.

Abbildung 3.6: Rotierenden Würfen mit Steuerung durch den User

- **NEAREST** wählt beim Skalieren der Textur den naheliegensten Punkt des Original-Bildes.
- **LINEAR** wählt einen linear interpolierten Wert. Bei hochskalierten Texturen liefert dieser Filter bessere Ergebnisse als *NEAREST*, allerdings werden harte Kanten im Bild verwischt.
- **MIPMAP** wird verwendet, da lineare Filterung zwar beim hochskalieren besser Ergebnisse liefert, beim herunter Skalieren aber genauso gut oder schlechte wie die *NEAREST*-Methode liefert. Dabei werden mehrere Varianten der Texture für unterschiedliche Verkleinerungsstufen erstellt, die dann zur Größe passend gewählt werden. Der Aufwand für Mipmaps ist sehr viel höher als für *NEAREST* oder *LINEAR*, liefert aber auch bedeutend bessere Ergebnisse.

Die neue Funktion *handleLoadedTexture* ist in Listing 3.41 beschrieben. Für die drei Textur-Varianten werden jeweils unterschiedliche Filter definiert. Da das Erzeugen von Mipmaps sehr aufwändig ist und bei der Hochskalierung keine Vorteile gegenüber der Linearen Filterung bringt, wird sie nur bei der Verkleinerung eingesetzt. der Funktion werden die Mipmaps erzeugt, was explizit angegeben werden muss.

Auch die Funktion *iniTexture* muss angepasst werden. De meisten Operationen müssen für alle drei Textur-Varianten ausgeführt werden, was in einer Schleife geschieht und anschließend die Texturen in das Textur-Array schreibt. Code in Listing 3.42.

3.8 Einfache Beleuchtung

Beleuchtung ist ein wesentliches Element, wenn man eine annährend realistische Grafik anstrebt. Bislang waren die Szene alle gleichmäßig ausgeleuchtet, jede Seite des Würfels erschien gleich hell. Dies ist allerdings wenig realistisch.

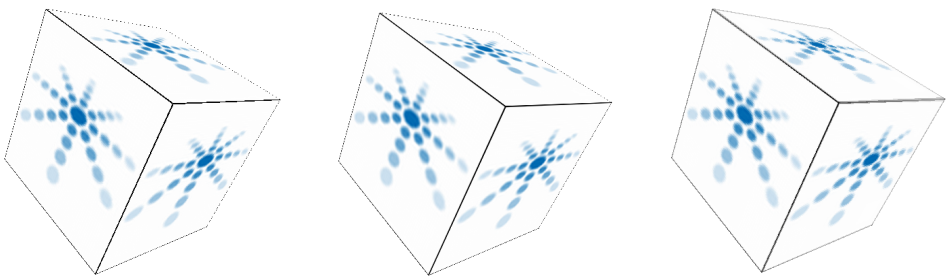
WebGL bietet, im Gegensatz zu OpenGL, keine vorgegebenen Möglichkeiten für die Beleuchtung. In OpenGL können bis zu acht Lichtquellen spezifiziert werden, alles weitere übernimmt OpenGL. Allerdings sind diese Lichter stark eingeschränkt, sie erlauben z.b. keinen Schattenwurf. Deshalb sind sie nur für sehr simple Anforderungen geeignet und werden in der Praxis kaum eingesetzt. Das bisherige Beispiel soll um zwei Arten der Beleuchtung erweitert werden: *Ambient Light* und *Directional Light*.

Listing 3.41: handleLoadedTexture mit unterschiedlichen Texturfilter

```
1  function handleLoadedTexture(textures){
2          gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
3
4          gl.bindTexture(gl.TEXTURE_2D, textures[0]);
5          gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.
               UNSIGNED_BYTE, textures[0].image);
6          gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER,
               gl.NEAREST);
7          gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
               gl.NEAREST);
8
9          gl.bindTexture(gl.TEXTURE_2D, textures[1]);
10         gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.
               UNSIGNED_BYTE, textures[1].image);
11         gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER,
               gl.LINEAR);
12         gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
               gl.LINEAR);
13
14         gl.bindTexture(gl.TEXTURE_2D, textures[2]);
15         gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.
               UNSIGNED_BYTE, textures[2].image);
16         gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER,
               gl.LINEAR);
17         gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
               gl.LINEAR_MIMAP_NEAREST);
18         gl.generateMipmap(gl.TEXTURE_2D);
19
20         gl.bindTexture(gl.TEXTURE_2D, null);
21 }
```

Listing 3.42: initTexture für Texturfilter

```
1  function initTexture(){
2          var htwImage = new Image();
3
4          for(var i=0; i < 3; i++){
5                  var texture = gl.createTexture();
6                  texture.image = htwImage;
7                  htwTextures.push(texture);
8          }
9
10         htwImage.onload = function(){
11                 handleLoadedTexture(htwTextures)
12         }
13         htwImage.src = "htwlogo256_border.png";
14 }
```



(a) Nearest (b) Linear (c) Mipmap

Abbildung 3.7: Varianten der Texturfilter

Ambient Light bezeichnet eine Art der Beleuchtung, bei der alle Objekte in der Szene gleichmäßig ausgeleuchtet werden. Dies ähnelt stark der bisherigen, impliziten Beleuchtung, die in den Beispielen zum Einsatz kam.

Directional Light beschreibt eine gerichtete Lichtquelle, die in einem bestimmten Winkel auf die Objekte strahlt. Für die Ausleuchtung ist der Winkel zur Lichtquelle entscheident.

Um die Auswirkungen besser Sichtbar zu machen, wird ein einfaches Interface in die HTML-Seite eingebaut, in der der User die Ausrichtung und Färbung des gerichteten Lichtes sowie die Farbanteile des *Ambient Light* steuern kann. Aus den Eingabefeldern werden in der Funktion *drawScene* die Werte ausgelesen. Über eine Checkbox kann zudem die Beleuchtung Ein und Aus geschaltet werden, um die Unterschiede noch weiter zu verdeutlichen. Der entsprechende Code-Ausschnitt ist in Listing 3.43 abgebildet.

Eine entscheidende Bedeutung bei der Beleuchtungsrechnung kommt den Normalenvektoren der Flächen zu. Mit den Normalenvektoren der Vertecies kann die Ausrichtung der Fläche im Raum mit einem einzigen Vektor beschrieben werden. Da in diesem Beispiel das Licht immer aus der selben Richtung kommt, genügt es, aus dem Normalenvektor und der Richtung des Lichtes das Skalarprodukt zu bilden, was im Vertex-Shader gemacht wird. Die Normalenvektor jedes Vertecies werden in einen Puffer geschrieben, der Code dafür ist in Listing 3.44 abgebildet.

Auch die Normalenvektoren müssen transformiert und dann an die Shader weitergegeben werden. Damit es zu keinen Fehler bei der Tranformation kommt, wird eine neue Matrix erzeugt, die aus den oberen linken 3x3 Teil der Mode-View-Matrix besteht. Der erweiterte Code ist in Listing 3.45 angegeben.

Da die Texturierung letztlich immer über die Shader geht, müssen auch diese erweitert werden.

Der Fragment-Shader wird nur geringfügig erweitert. Bisher wurde die Darstellung eines Pixels nur durch die Textur vorgegeben, jetzt wird auch die Färbung des Lichtes und seine Gewichtung, als wie stark die Ausleuchtung an diesem Punkt ist, mit ein bezogen. Code in Listing 3.46.

Der Vertex-Shader erfordert mehr Änderungen. Da das Objekt ein Würfel ist genügt es, die Beleuchtungsintensität an jedem Vertex zu ermitteln und für alle anderen Punkte der Fläche zu übertragen. Bei einer gekrümmten Fläche muss für jeden Punkt die Beleuchtung neu berechnet werden, weshalb auch der Fragment-Shader nur wenig Änderungen erfahren hat. Im Vertex-Shader (Listing 3.47) wird die Normale des Vertex mit der Transformationsmatrix für Normalen transformiert. An-

Listing 3.43: Ausschnitt drawScene mit Beleuchtungsparametern

```
var lighting = document.getElementById("lighting").checked;
gl.uniform1i(shaderProgram.useLightingUniform, lighting);
if(lighting){
    gl.uniform3f(
        shaderProgram.ambientColorUniform,
        parseFloat(document.getElementById("ambientR").value),
        parseFloat(document.getElementById("ambientG").value),
        parseFloat(document.getElementById("ambientB").value)
    );

    var lightingDirection = [
        parseFloat(document.getElementById("lightDirectionX").
            value),
        parseFloat(document.getElementById("lightDirectionY").
            value),
        parseFloat(document.getElementById("lightDirectionZ").
            value)
    ];
    var adjustedLD = vec3.create();
    vec3.normalize(lightingDirection, adjustedLD);
    vec3.scale(adjustedLD, -1);
    gl.uniform3fv(shaderProgram.lightingDirectionUniform,
        adjustedLD);

    gl.uniform3f(
        shaderProgram.directionalColorUniform,
        parseFloat(document.getElementById("directionalR").
            value),
        parseFloat(document.getElementById("directionalG").
            value),
        parseFloat(document.getElementById("directionalB").
            value)
    );
}
```

Listing 3.44: Puffer für Normalen des Würfels in Funktion initBuffer

```
1  cubeVertexNormalBuffer = gl.createBuffer();
2  gl.bindBuffer(gl.ARRAY_BUFFER, cubeVertexNormalBuffer);
3  var vertexNormals = [
4      // Vorderseite
5       0.0,  0.0,  1.0,
6       0.0,  0.0,  1.0,
7       0.0,  0.0,  1.0,
8       0.0,  0.0,  1.0,
9
10     // Rueckseite
11      0.0,  0.0, -1.0,
12      0.0,  0.0, -1.0,
13      0.0,  0.0, -1.0,
14      0.0,  0.0, -1.0,
15
16     // Obere Seite / Deckel
17      0.0,  1.0,  0.0,
18      0.0,  1.0,  0.0,
19      0.0,  1.0,  0.0,
20      0.0,  1.0,  0.0,
21
22     // Untere Seite / Boden
23      0.0, -1.0,  0.0,
24      0.0, -1.0,  0.0,
25      0.0, -1.0,  0.0,
26      0.0, -1.0,  0.0,
27
28     // Rechte Seite
29      1.0,  0.0,  0.0,
30      1.0,  0.0,  0.0,
31      1.0,  0.0,  0.0,
32      1.0,  0.0,  0.0,
33
34     // Linke Seite
35     -1.0,  0.0,  0.0,
36     -1.0,  0.0,  0.0,
37     -1.0,  0.0,  0.0,
38     -1.0,  0.0,  0.0
39 ];
40 gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertexNormals),
       gl.STATIC_DRAW);
41 cubeVertexNormalBuffer.itemSize = 3;
42 cubeVertexNormalBuffer.numItems = 24;
```

Listing 3.45: Transformierung der Normalenvektoren in Funktion setMatrixUniforms

```
1 function setMatrixUniforms(){
2     gl.uniformMatrix4fv(shaderProgram.pMatrixUniform, false,
          pMatrix);
3     gl.uniformMatrix4fv(shaderProgram.mvMatrixUniform, false,
          mvMatrix);
4
5         var normalMatrix = mat3.create();
6     mat4.toInverseMat3(mvMatrix, normalMatrix);
7     mat3.transpose(normalMatrix);
8     gl.uniformMatrix3fv(shaderProgram.nMatrixUniform, false,
          normalMatrix);
9 }
```

Listing 3.46: Fragment-Shader mit Beleuchtung

```
1  #ifdef GL_ES
2  precision highp float;
3  #endif
4
5  varying vec2 vTextureCoord;
6  varying vec3 vLightWeighting;
7
8  uniform sampler2D uSampler;
9
10 void main(void){
11     vec4 textureColor = texture2D(uSampler, vec2(vTextureCoord.
           s, vTextureCoord.t));
12         gl_FragColor = vec4(textureColor.rgb * vLightWeighting,
               textureColor.a);
13 }
```

schließend wird die Gewichtung oder Intensität des des Lichtes an dieser Stelle berechnet, die aus dem Skalarprodukt (Funktion *dot()*) der transformierten Normalen und der allgemeinen Richtung des Lichtes errechnet werden. Da sich hierbei auch negative Werte ergeben können, werden diese mit einem Maximum-Vergleich mit 0 eliminiert. Negative Werte können entstehen, da hier die Menge des diffus reflektierten Lichts der Oberfläche berechnet wird, was sich proportional aus dem Cosinus zwischen den Winklen des Normalenvektors und der Richtung des Lichtes ergibt. Zur Intensität des Lichtes kommen noch die Färbung des *Ambient Light* und *Directional Light*.

Im Abbildung 3.8 wird ein Würfel gezeigt, der die unterschiedliche Beleuchtung der Seite verdeutlicht. Zudem wurde der Würfel durch die Parameter des *Directional Lights* grün gefärbt. Mit entsprechend hohen Werten kann so die Textur mit dem Licht "überschrieben"werden.

3.9 Geometrie aus einer Datei laden

Die Geometrie der Objekte in der Szene nehmen bislang im Programmcode einen sehr großen Teil ein. Dabei wurden nur die Puffer für den WebGL Kontext mit den Positionsdaten der Vertices. Um den Programmcode übersichtlicher zu gestalten, werden die Daten in eine externe Datei ausgelagert, die zur Laufzeit nachgeladen wird.

In Lektion 10 [Thob] stellt Giles Thomas ein sehr einfaches Format vor, das im wesentlichen die gleichen Informationen enthält, mit denen bisher die Puffer gefüllt wurden: Für jeden Vertex die Positionsdaten, die Texturkoordinaten und der Normalenvektor. Dieses Format besticht zwar durch seine Einfachheit und dass es leicht zu parsen ist, aber bei größeren Objekten wird es schnell unübersichtlich und enthält viel redudante Information. Ein Ausschnitt des Formates ist in Listing 3.48, der Parser dazu in Listing 3.49. Im Parser wird zuerst über die Funktion *loadWorld* die Datei über ein *XMLHttpRequest* asynchron geladen und an die Funktion *handleLoadedWorld* weitergegeben. In dieser Funktion wird die geladene Datei in ihre Zeilen zerlegt und die Werte in die zugehörigen Arrays geschrieben wird, wobei davon ausgegangen wird, dass sie in der korrekten Reihenfolge sind.

Für den normalen Gebrauch ist diese Format kaum geeignet, da es sehr viele redundante Daten enthält. Da in der Praxis auch kaum Geometrie von Hand erzeugt wird, sondern mit Hilfe eines Modellierungswerkzeuges, werden für den Import auch entsprechenden Formate wie z.b. COLLADA eingesetzt. Einige dieser Forma-

Listing 3.47: Vertex-Shader mit Beleuchtung

```
1  attribute vec3 aVertexPosition;
2  attribute vec3 aVertexNormal;
3  attribute vec2 aTextureCoord;
4
5  uniform mat4 uMVMatrix;
6  uniform mat4 uPMatrix;
7  uniform mat3 uNMatrix;
8
9  uniform vec3 uAmbientColor;
10
11 uniform vec3 uLightingDirection;
12 uniform vec3 uDirectionalColor;
13
14 uniform bool uUseLighting;
15
16 varying vec2 vTextureCoord;
17 varying vec3 vLightWeighting;
18
19 void main(void){
20     gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition,
       1.0);
21         vTextureCoord = aTextureCoord;
22
23         if(!uUseLighting){
24                 vLightWeighting = vec3(1.0, 1.0, 1.0);
25         } else {
26                 vec3 transformedNormal = uNMatrix *
                   aVertexNormal;
27                 float directionalLightWeighting = max(dot(
                   transformedNormal, uLightingDirection), 0.0)
                   ;
28                 vLightWeighting = uAmbientColor +
                   uDirectionalColor *
                   directionalLightWeighting;
29         }
30 }
```

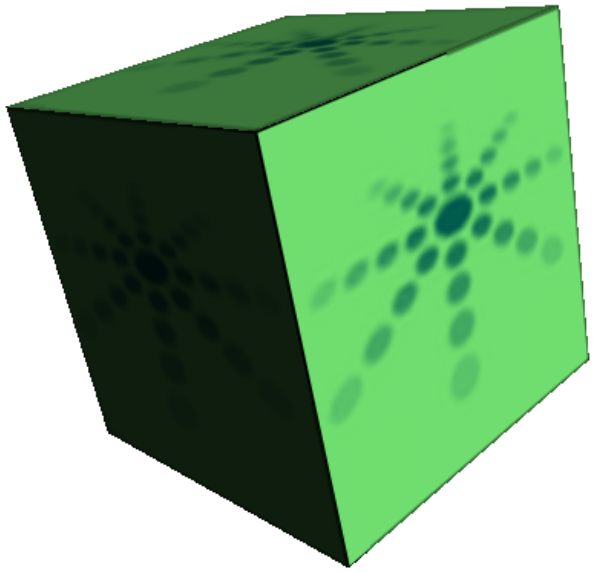


Abbildung 3.8: Grün beleuchteter Würfel

Listing 3.48: primitives Datenformat

```
1 // Floor
2 //   X     Y     Z     S     T     Nx    Ny    Nz
3    -5.0   0.0  -5.0   0.0  10.0   0.0   1.0   0.0
4    -5.0   0.0   5.0   0.0   0.0   0.0   1.0   0.0
5     5.0   0.0   5.0  10.0   0.0   0.0   1.0   0.0
6
7    -5.0   0.0  -5.0   0.0  10.0   0.0   1.0   0.0
8     5.0   0.0  -5.0  10.0  10.0   0.0   1.0   0.0
9     5.0   0.0   5.0  10.0   0.0   0.0   1.0   0.0
```

Listing 3.49: Ausschnitt Parser für primitives Datenformat

```
function loadWorld(object){
    var request = new XMLHttpRequest();
    request.open("GET", object.filename);
    request.onreadystatechange = function(){
        if(request.readyState == 4){
            handleLoadedWorld(object, request.responseText);
        }
    }
    request.send();
}

function handleLoadedWorld(object, data){
    lines = data.split("\n");
    vertexCount = 0;

    object.vertexPositions = [];
    object.vertexTextureCoords = [];
    for(var i in lines){
        var vals = lines[i].replace(/^\s+/, "").split(/\s+/);
        if(vals.length == 8 && vals[0] != "//"){
            // Beschreibung eines Vertex: X, Y, Z
            object.vertexPositions.push(parseFloat(vals[0]));
            object.vertexPositions.push(parseFloat(vals[1]));
            object.vertexPositions.push(parseFloat(vals[2]));

            // Texturkoordinaten S und T
                        [...]
                        // Normalenvektoren
                        [...]
            vertexCount += 1;
        }
    }

    object.vertexPositionBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, object.vertexPositionBuffer)
        ;
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(object.
        vertexPositions), gl.STATIC_DRAW);
    object.vertexPositionBuffer.itemSize = 3;
    object.vertexPositionBuffer.numItems = vertexCount;
```

te können mit Hilfe von Bibliotheken und Frameworks eingelesen werden, mehr dazu im Abschnitt 5.1.

3.10 Simulation einer einfachen Kamera

Bislang wurde in allen Szenen einen implizite, feste Kamera eingesetzt, die sich bei den Koordinaten *0/0/0* befindet und entlang der Z-Achse gerichtet ist. Da auch jedes mal ein einzelnes Objekte betrachtet wurde, das ohne jeden Kontext, quasi im "Nichts", dargestellt wurde. Die mit der in Abschnitt 3.9 gezeigten Methoden geladene Geometrie soll nun begehbar werden, eine Art Kamera soll vom Benutzer kontrolliert werden.

Dies lässt sich realisieren, in dem die bisherige Model-View-Matrix aufgespalten wird in eine Model- und eine View-Matrix. Erstere behandelt die Informationen der Geometrie, letztere die der Kamera, die getrennt voneinander geändert werden können, wodurch eine richtige Kamera möglich wird. Da WebGL über keine eingebauten Möglichkeiten für eine Kamera bietet, muss diese vom Programmierer selbst entwickelt werden.

Eine andere, einfachere Möglichkeit besteht darin, eine Kamera zu simulieren. Dabei bleibt die Kamera fest am Punkt *0/0/0* und es wird die bereits vorhandene Model-View-Matrix verwendet. Bei dieser Methode wird nicht die Kamera bewegt, sondern die Geometrie entsprechenden den simulierten Bewegungen verändert und neu gezeichnet. Um die Kamera passend zu simulieren, muss jede Bewegung negiert werden, beschrieben in Listung 3.51. Um der Bewegung das Gefühl des Laufens zu geben, wird in der Funktion *animate* zusätzlich noch der Faktor der Auf- und Abbewegung des Kopfes simuliert, ausgedrückt durch die Variable *headbob* (siehe dazu Listing 3.50.

Natürlich hat diese Methode auch einige Nachteile: durch die komplette Neuberechnung der Koordinaten der Geometrie entsteht besonders in Aufwändigen Szenen ein enormer Rechenaufwand. Auch muss darauf geachtet werden, dass alle Objekte, insbesondere die nicht immer sichtbaren Beleuchtungsobjekte, mit bewegt werden müssen. Auch muss für die Beleuchtungsrechnung die mögliche Änderung der Normalvektoren berücksichtigt werden. Für eine komplexere Applikation sollte deswegen eine eigene Kamera-Matrix eingesetzt werden. Dies war auch für diese Arbeit angedacht, konnte aber bis zum Abgabe Termin nicht mehr umgesetzt werden.

Listing 3.50: Ausschnitt animate zur Simulation der Kamera

```
if(speed != 0){
    xPos -= Math.sin(degToRad(yaw)) * speed * elapsed;
    zPos -= Math.cos(degToRad(yaw)) * speed * elapsed;

    headbob += elapsed * 0.6;
    yPos = Math.sin(degToRad(headbob)) / 20 + 0.4
}

yaw += yawRate * elapsed;
pitch += pitchRate * elapsed;
```

Listing 3.51: Ausschnitt drawScene zur Simulation der Kamera

```
mat4.rotate(mvMatrix, degToRad(-pitch), [1, 0, 0]);
mat4.rotate(mvMatrix, degToRad(-yaw), [0, 1, 0]);
mat4.translate(mvMatrix, [-xPos, -yPos, -zPos]);
```

3.11 Punktlicht und Per-Fragment-Beleuchtung

In Abschnitt 3.8 wurde ein einfaches Beleuchtungsmodell vorgestellt, dass eine Lichtquelle beschreibt, die von außerhalb der Szene auf diese scheint. Eine Lichtquelle kann aber auch innerhalb der Szene, bei der nicht nur der Winkel, mit der das Licht auf die Objekte fällt entscheidet, sondern auch die Position. Zudem ist der Winkel zu den Objekten nicht überall gleich, er muss also für jeden Vertex neu berechnet werden. Beleuchtung ausgehend von der Vertecies hat den Nachteil, dass sie speziell bei runden Objekten unrealistisch aussieht. Aber auch bei flachen Objekten kann dies zu einer unrealistischen Ausleuchtung führen, wenn z.b. die Lichtquelle und das beleuchtete Objekte auf derselbe Höhe sind. Die Fläche erscheint dann gleichmäßig ausgeleuchtet, obwohl sie auf Höhe der Lichtquelle stärker und zu den Rändern hin schwächer ausgeleuchtet sein sollte. Um dieses Problem zu beheben muss die Beleuchtung für jeden Punkt berechnet werden, was zwar einen höheren Rechenaufwand bedeutet, aber eine realistischere Beleuchtung ermöglicht.

Für beides müssen die Shader angepasst werden. Der Code für Vertex- und Fragment-Shader sind im Vergleich zu den vorherigen Beispielen in weiten Teilen vertauscht, da die Berechnungen nun nicht mehr nur für einen Vertex, sondern für jedes Frag-

ment ausgeführt werden müssen. Im Vertex-Shader (Listing 3.52) werden nur noch die Textur-Koordinaten sowie seine Position bestimmt. Im Fragment-Shader (Listing 3.53) wird die Lichtstärke sowie der Winkel zur Punktquelle berechnet, um die Ausleuchtung des Punktes zu bestimmen. Da bei einer texturierten Oberfläche der Unterschied nur schlecht sichtbar wird, kann über einen Schalter zudem die Texturierung durch eine Färbung der Fläche ersetzt werden.

Auch die Berechnung der Richtung des Lichtes muss für jedes Fragment einzeln gemacht werden. Bei der einfachen Lichquelle in Abschnitt 3.8 reichte es, in der Funktion *drawScene* die Berechnung des Lichtes einmalig durch zu führen. Es biete sich hier an, die einfache Berechnung auch in die Shader zu verlagern, genauer gesagt in den Fragment-Shader (vgl. Listing 3.53). Die Richtung wird durch einen normalisierten Vektor angegeben, der sich aus dem Punkt der Lichtquelle und der Koordinaten des aktuellen Fragments ergibt. An Berechnung der Lichtintensität ändert sich nichts.

Listing 3.52: Per-Fragment-Beleuchtung Vertex-Shader

```
1  attribute vec3 aVertexPosition;
2  attribute vec3 aVertexNormal;
3  attribute vec2 aTextureCoord;
4
5  uniform mat4 uMVMatrix;
6  uniform mat4 uPMatrix;
7  uniform mat3 uNMatrix;
8
9  varying vec2 vTextureCoord;
10 varying vec3 vTransformedNormal;
11 varying vec4 vPosition;
12
13 void main(void){
14     vPosition = uMVMatrix * vec4(aVertexPosition, 1.0);
15     gl_Position = uPMatrix * vPosition;
16     vTextureCoord = aTextureCoord;
17     vTransformedNormal = uNMatrix * aVertexNormal;
18 }
```

Listing 3.53: Per-Fragment-Beleuchtung Fragment-Shader

```
1  #ifdef GL_ES
2  precision highp float;
3  #endif
4
5  varying vec2 vTextureCoord;
6  varying vec3 vTransformedNormal;
7  varying vec4 vPosition;
8
9  uniform bool uUseLighting;
10 uniform bool uUseTextures;
11
12 uniform vec3 uAmbientColor;
13 uniform vec3 uPointLightingLocation;
14 uniform vec3 uPointLightingColor;
15
16 uniform sampler2D uSampler;
17
18 void main(void){
19     vec3 lightWeighting;
20     if(!uUseLighting){
21         lightWeighting = vec3(1.0, 1.0, 1.0);
22     } else {
23         vec3 lightDirection = normalize(uPointLightingLocation
               - vPosition.xyz);
24
25         float directionalLightWeighting = max(dot(normalize(
               vTransformedNormal), lightDirection), 0.0);
26         lightWeighting = uAmbientColor + uPointLightingColor *
               directionalLightWeighting;
27     }
28
29     vec4 fragmentColor;
30         if(uUseTextures){
31                 fragmentColor = texture2D(uSampler, vec2(
                       vTextureCoord.s, vTextureCoord.t));
32         } else {
33                 fragmentColor = vec4(1.0, 1.0, 1.0, 1.0);
34         }
35
36     gl_FragColor = vec4(fragmentColor.rgb * lightWeighting,
           fragmentColor.a);
37 }
```

4 Virtuelle Hochschule

4.1 Konzept

Das Ziel dieser Projektarbeit ist, ein Programm zu entwickeln, dass es einem User ermöglicht, einen virtuellen Rundgang durch das Hochschulgebäude der Fakultät Elektronik und Informatik zu ermöglichen. Dies beschränkt sich zunächst auf das Erdgeschoss. Es soll keine Kamerafahrt werden, sondern der User soll selbst die Kamera steuern können.

4.2 Programmierung

Das Programm basiert im Wesentlichen auf den in Abschnitten 3.10 und 3.9 vorgestellten Lösungen. Der Programmcode wurde entsprechend angepasst, um eine beliebige Anzahl von Objekten von Datein aus dem Dateisystem laden zu können. Zuerst wird ein kleines PHP Skript aufgerufen (Code in Listing 4.1), dass in einem Unterordner des Dateisystems nach Datei mit einem festgelegten Prefix und Endung sucht (in diesem Fall als Prefix *htw_* und als Endung *.txt*. Die Zeichen dazwischen werden als regulärer Ausdruck auf Ziffern, Groß- und Kleinbuchstaben sowie den Unterstrich beschränkt. Die Dateipfade werden in ein JavaScript-Array geschrieben, welches vom späteren Programm aus erreichbar ist. Es wird ein PHP-Skript verwendet, da das suchen in Datein auf dem Server ohne den genauen Pfad in JavaScript schwierig ist, da es clientseitig ausgeführt wird. PHP dagegen läuft serverseitig, dazu wird die kleine Suche schneller ausgeführt. Das PHP-Skript wird ähnlich wie ein JavaScript in die HTML-Datei eingebunden.

Im Hauptprogramm wird ein Array *geom* angelegt, für welches ein Objekt für jede gefundene Datei in *objectArray*, welches vom PHP-Skript erzeugt wird, erzeugt. Dies wird in der Funktion *initObjectsToDraw* (siehe Listing 4.2) erledigt sowie die Pfade zu den Geometrie-Informatinen sowie der Textur festgelegt. Der Pfad zur Textur ist fest vorgegeben und der Dateiname derselbe wie zur Datei mit den Geometrie-Informationen. Versuche, die Texturen in der Datei selber fest zu legen scheiterten an den Browser-Optimierungen, die dazu führten dass bereits gezeichnet wurde, obwohl der Pfad zur Textur noch nicht bekannt war, was in schwarzen Oberflächen resultierte, die je nach Ladezeit variierten. Dieses Fehlen wird von WebGL nicht nachträglich korrigiert, da die Textur bereits initialisiert ist, dafür müsste weiterer Code eine Standard-Textur laden und nachträglich mit der richtigen überschreiben, wenn sie geladen und initialisiert ist. Diese Problematik trat nur im normalen Ablauf auf, nicht wenn mit der JavaScript-Code per Firefox-Addon

Listing 4.1: PHP-Skript um Pfade zu Datein mit Geometrieinformationen zu erhalten

```
1  <?
2  Header("content-type: application/x-javascript");
3  function returnObjects($dirname="./geom") {
4          $pattern="(htw_[0-9a-zA-Z_]*.txt$)";
5          $files = array();
6          $curObject=0;
7          if($handle = opendir($dirname)) {
8                  while(false !== ($file = readdir($handle))){
9                          if(preg_match($pattern, $file)){
10                         echo 'objectArray['.$curObject.']="'.
                               $file .'";';
11                         $curObject++;
12                         }
13                 }
14                 closedir($handle);
15         }
16         return($files);
17 }
18
19 echo 'var objectArray = new Array();';
20 returnObjects();
21 ?>
```

Firebug debugged wurde.

Listing 4.2: Festlegen der Objekte

```
1  function initObjectsToDraw(){
2          for ( var i in objectArray ){
3                  objectArray[i] = objectArray[i].substr(0,
                       objectArray[i].length-3);
4
5                  geom[i] = new Object();
6                  geom[i].filename = "geom/" + objectArray[i] + "
                       txt";
7                  geom[i].textureFilename = "tex/" + objectArray[
                       i] + "png";
8          }
9
10 }
```

4.3 Kontruktion

Als Grundlage für die Konstruktion dient der Übersichtsplan des Erdgeschosses von Gebäude 2 auf dem Campus Burren. Die Geometriedaten werden, wie im [Thob] beschriebene Format, als einzelne Dreiecke definiert. Dies bedeutete einen große Aufwand im Vergleich zur Modellierung mit einem Werkzeug, allerdings wurde darauf geachtet, die Größenverhältnisse bei zu behalten, was auch bei einer Modellierung auf die manuelle Eingabe und Errechnung der Koordinaten bedeutet hätte. Im Modell wurde aus Zeitgründen nur die Wände verwendet, keine Details der Einrichtung.

Die Längenangaben wurden 1 zu 1 übernommen, aus einem Meter in der realen Welt wurde eine Längeneinheit in der Virtuellen. Die Größenverhältnisse wurden Subjektiv geprüft und für gut befunden.

4.3.1 Texturierung

Da keine Originaltexturen vorlagen, wurden Texturen der Webseite CG Textures [Vij] verwendet, die die realen nächstmöglich kommen. Die Texturen wurden in ihrer Größe angepasst, da WebGL nur Texturen mit Kantenlängen, die Zweierpotenzen entsprechen akzeptiert. Zudem wurden einige in ihrer Farbgebung, Helligkeit

und Kontrast angepasst, damit sie den realen Vorbildern mehr entsprechen.

4.4 Erweiterungen

4.4.1 Maus-Steuerung

Die Steuerung der Kamera mit der Maus, wie sie aus Computerspielen des Genres der First-Person Shooter bekannt ist, ist prinzipiell denkbar, aber nicht unproblematisch in JavaScript und WebGL. JavaScript arbeitet mit Events, die bei WebGL innerhalb des Canvas-Elements erfasst werden. Bewegt man aber den Mauszeiger aus dem Canvas-Element heraus, bleibt der Punkt des Mouse-Events auf dem letzten Pixel innerhalb des Canvas hängen, wodurch die Ansicht weiter verändert wird, der Mauszeiger aber bereits den Bereich des Canvas verlassen hat. Da sich bisher der Mauszeiger nicht “fangen“ und gezwungen im Canvas-Element halten lässt, wie es bei Desktop-Applikationen möglich ist, ist diese Methode der Steuerung nicht sehr praktikabel und deshalb wurde hier bisher darauf verzichtet.

4.4.2 Kamera

Im Programm wird eine simulierte Kamera, wie beschrieben in Abschnitt 3.10, eingesetzt. Dies hat aber zu Problemen, insbesondere bei der Beleuchtung, geführt, weshalb diese deaktiviert ist. Mit einer richtigen Kamera ließe sich dieses Problem beheben. Diese Funktionalität war angedacht, konnte aber nicht mehr bis zum Abgabetermin umgesetzt werden.

4.4.3 Kollissions-Abfrage

Im Programm ist es bisher möglich, die Wände zu ignorieren und einfach durch sie hindurch zu gehen, da es keine Kollisionsabfrage gibt. Die Geometrie wird zwar gezeichnet, hat aber auf die Bewegung der Kamera keinen Einfluss. Eine Kollissionsabfrage wäre hierzu nötigt, allerdings wurde in dieser Arbeit darauf verzichtet, da der Aufwand für die Implementierung die Anfoderungen übersteigen würde. Das einbinden einer Bibliothek eines Drittanbieters ist ebenfalls schwierig, da die meisten Lösungen komplette Frameworks und keine einfach ein zu bauenden Bibliotheken sind, nähers dazu im Abschnit 5.1.

5 Weiterführendes

5.1 Frameworks

Obwohl WebGL noch eine sehr junge Technologie ist, sind bereits einige Frameworksentstanden, die die Programmierung vereinfachen sollen. Die meisten Frameworks maskieren den Einsatz von WebGL stark, wodurch der Programmcode mit den Frameworks sich von jenem mit reinem WebGL stark unterscheidet, da viele Definitionen und Routine Code dem Programmierer abgenommen wird und sind generell eine gute Möglichkeit, schnell und ohne großen Aufwand und spezielle Vorkenntnisse im Bereich der Computergraphik, sichtbare Ergebnisse zu erzeugen. Allerdings begibt man sich dadurch auch eine starke Abhängigkeit zum jeweilige Framework und da hinter den meisten keine große Firmen stehen ist eine Weiterentwicklung nicht gewährleistet.

5.1.1 c3DL

Ursprünglich für den WebGL-Vorläufer *Canvas-3D* entwickelt, nimmt dieses Framework des Seneca College dem Programmierer vor allem Routineaufgabe ab. Die Dokumentation ist ordentlich, obwohl die Entwickler oft wechseln. Allerdings fehlen die heraus stechenden Besonderheiten gegenüber anderen Frameworks, die teilweise sehr ähnlich aufgebaut sind.

5.1.2 Copperlicht

Copperlicht beschreibt sich selbst nicht als Framework, sondern als 3D-Engine. Die Dokumentation ist gut und bietet neben den Standardfunktionen auch einige Fortgeschrittene wie eine Kollisionsabfrage. Die Engine wird unter einer geteilten Lizenz verteilt, die kostenlose ist komprimiert und nur schwer lesbar, was nur bei der kommerziellen Varianten möglich ist. Neben der eigentlichen Engine gibt es einen komerziellen, graphischen Editor für 3D-Szenen, die direkt in ein Web-taugliches Format exportiert werden können und Copperlicht zur Darstellung nutzt, hierbei reicht die kostenfreie Version. Der Editor ist zwar sehr einfach zu bedienen, kann aber mit einer professionellen Modellierungssoftware oder den Editoren, die teilweise kommerziellen Videospielen beiliegen, vom Funktionsumfang her nicht mithalten.

5.1.3 GLGE

GLGE ist eines der ältesten Frameworks für WebGL, das bereits seit Dezember 2009 in Entwicklung. Der Fokus liegt auf der Maskierung der Eigenarten von WebGL als Low-Level-API, damit sich der Programmierer mehr auf das eigentliche Programm konzentrieren kann. Daneben ist die Performance ein wichtiger Punkt für die Entwickler.

5.1.4 PhiloGL

PhiloGL ist ein Framework der Sencha Labs, bekannt für das JavaScript-Framework ExtJS. PhiloGL biete viele Funktionen und nimmt dem Programmierer vieles ab. Das Framework ist gut strukturiert und dokumentiert, es hält sich an viele Best-Praticeses der JavaScript-Entwicklung. Für einen leichten Einstieg wurden alle WebGL-Tutorials von Giles Thomas für PhiloGL portiert, was vor allem den Unterschied in der reinen Anzahl der Lines of Code zeigt und verdeutlicht, wie viel Arbeit Frameworks dem Entwickler abnehmen können.

5.1.5 SceneJS

Wie der Name bereits vermuten lässt, kommt **SceneJS** eine Scene-Graph-API auf Basis von WebGL sehr nahe, wobei die Syntax an VRML angelehnt ist, einer Beschreibungssprache für 3D-Szenen.

5.1.6 SpiderGL

SpiderGL ist ein sehr komplexes, aber auch gut dokumentiertes Framework. Eine Besonderheit ist die einfache Möglichkeit, Shader in einer eigenen Testumgebung testen zu können.

5.1.7 X3DOM

X3DOM wird am Frauenhofer IDG, Darmstadt entwickelt und stellt eine Schnittstelle zwischen X3D und WebGL dar. X3D ist eine Beschreibungsspache für 3D-Szenen, die als Nachfolger für VRML entwickelt wurde, sich aber nie durchgesetzt hat.

Abbildungsverzeichnis

Listings

Literaturverzeichnis

[Inca] Google Inc. angleproject - angle: Almost native graphics layer engine - google project hosting. http://code.google.com/p/angleproject/. Last accessed: 2011/08/19.

[Incb] Google Inc. webgl-utils.js. https://cvs.khronos.org/svn/repos/registry/trunk/public/webgl/sdk/demos/common/webgl-utils.js. Last accessed: 2011/08/03.

[Jon] Brandon Jones. glmatrix - high performance matrix and vector operations for webgl. http://code.google.com/p/glmatrix/. Last accessed: 2011/08/02.

[Ltd] Context Information Security Ltd. Webgl - a new dimension for browser exploitation. http://www.contextis.com/resources/blog/webgl/. Last accessed: 2011/08/19.

[Pet] Jan Petzold. Entwickeln in javascript | heise development. http://heise.de/-1318509. Last accessed: 2011/08/19.

[Pro] Mesa Project. Mesa home page. http://mesa3d.org/. Last accessed: 2011/08/19.

[Str11] Peter Strohm. Quellcode-Matroschka. *iX*, pages 117–121, 5 2011.

[Thoa] Giles Thomas. Webgl lesson 1 – a triangle and a square. http://learningwebgl.com/blog/?p=28. Last accessed: 2011/08/02.

[Thob] Giles Thomas. Webgl lesson 10 - loading a world, and the most basic kind of camera. http://learningwebgl.com/blog/?p=1067. Last accessed: 2011/08/17.

[Vij] Marcel Vijfwinkel. [cg textures] - textures for 3d, graphic design and photoshop! http://www.cgtextures.com/. Last accessed: 2011/09/06.

[Wika] Wikipedia. 3d computer graphics - wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/3D_computer_graphics`. Last accessed: 2011/08/19.

[Wikb] Wikipedia. Webgl - wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/WebGL`. Last accessed: 2011/08/19.