Computer Graphics - WebGL, Teil 1

Thomas Koller

1 JavaScript

Die Programmier-Übungen in diesem Modul werden mit JavaScript durchgeführt. Dabei werden nur die einfachsten Sprachelemente benutzt, trotzdem empfiehlt es sich ein JavaScript Tutorial durchzuführen, oder ein entsprechendes Buch zu verwenden, zum Beispiel:

Head First JavaScript Programming. Eric. T. Freeman, Elisabeth Robson; O'Reilly Media; 1 edition (April 10, 2014); ISBN-13: 978-1449340131

JavaScript: The Definitive Guide; David Flanagan. O'Reilly & Associates; Auflage: 6 (1. Juni 2011); ISBN-13: 978-0596805524

Es wird empfohlen eine Entwicklungsumgebung für JavaScript zu verwenden. Für die Produkte von jetbrains (www.jetbrains.com) erhalten sie mit ihrer hslu Email kostenfreie Lizenzen, WebStorm ist eine Entwicklungsumgebung von jetbrains für JavaScript. Natürlich dürfen sie auch gerne andere Umgebungen wie zum Beispiel eclipse oder netbeans verwenden.

2 Einführung

In dieser Übung sollen sie erste Objekte mit WebGl zeichnen. Um den Einstieg einfach zu halten, werden wir zuerst die Konzepte anhand eines 2D Beispiels anschauen und danach das Ganze zu 3D erweitern.

WebGL verfügt über eine programmierbare Grafik Pipeline. Dies bedingt, dass bereits zum Zeichnen einfacher Objekte einiges programmiert werden muss, dafür können dann später interessante Grafikeffekte relativ einfach integriert werden.

Schritt 1: Aufsetzen von WebGL, Zeichnen des Hintergrunds

In einem ersten Schritt soll WebGL aufgesetzt werden und mit den ersten WebGL Befehlen der Hintergrund gezeichnet werden. Das Programm besteht dabei aus aus einem HTML Dokument, dass die Zeichenfläche (Canvas) aufsetzt und die JavaScript Files definiert in denen das eigentliche Program steht. Der folgende Code definiert die Zeichenfläche und lädt die JavaScript Files, die für das Projekt benötigt werden. Diese werden im folgenden genauer vorgestellt.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Computer Graphics</title>
    <script type="text/javascript" src="webgl-debug.js"></script>
    <script type="text/javascript" src="shaderUtils.js"></script>
    <script type="text/javascript" src="gl-matrix.js"></script>
    <script type="text/javascript" src="Exercise1.js"></script>
</head>
<body>
<h1>Computer Graphics</h1>
<canvas id="myCanvas" width="800" height="600"></canvas>
</body>
</html>
```

Im Javascript Code muss nun zuerst das Canvas Objekt aus dem Dokument gefunden werden und dann die `getContext()` Methode aufgerufen werden um das Kontext Objekt zu bekommen. Dafür wird der Parameter `"webgl"` benötigt, damit ein WebGL Kontext zurück gegeben wird und nicht ein 2D Kontext, der über ein anderes Interface angesprochen wird. Sämtliche WebGl Funktionen werden nun als Methoden des Kontext Objekts aufgerufen.

Der folgenden Code zeigt die Basisstruktur des Haupt Javascript Files.

```
//
// Computer Graphics
//
// WebGL Exercises
//

// Register function to call after document has loaded
window.onload = startup;

// the gl object is saved globally
var gl;

// we keep all local parameters for the program in a single object
var ctx = {
    shaderProgram: -1
};

/**
 * Startup function to be called when the body is loaded
```

```
 */
function startup() {
    "use strict";
    var canvas = document.getElementById("myCanvas");
    gl = createGLContext(canvas);
    initGL();
    draw();
}

/**
 * InitGL should contain the functionality that needs to be executed only once
 */
function initGL() {
    "use strict";
    ctx.shaderProgram = loadAndCompileShaders(gl, 'VertexShader.glsl', '
    FragmentShader.glsl');
    setUpAttributesAndUniforms();
    setUpBuffers();
    gl.clearColor(1,0,0,1);

    // add more necessary commands here
}

/**
 * Setup all the attribute and uniform variables
 */
function setUpAttributesAndUniforms(){
    "use strict";
}

/**
 * Setup the buffers to use. If more objects are needed this should be split in
     a file per object.
 */
function setUpBuffers(){
    "use strict";
}

/**
 * Draw the scene.
 */
function draw() {
    "use strict";
    console.log("Drawing");
    gl.clear(gl.COLOR_BUFFER_BIT);
    // add drawing routines here
}
```

In den Funktionen `initGL()` und `draw()` kann mittels `gl.someFunction()` auf Befehle von WebGL zugegriffen werden.

Debug Version der WebGL Library

WebGL gibt keine Fehlermeldungen zurück, jedoch kann jeweils mit dem Befehl `glError` abgefragt werden, ob ein Fehler existiert. Dies ist jedoch zum debuggen mühsam. Daher gibt es eine Bibliothek `webgl-debug.js`, die nach jedem gl Befehl automatisch diese Funktion aufruft und Fehler auf die Console schreibt, siehe auch `http://www.khronos.org/webgl/wiki/Debugging`.

Im obigen Code ist dies bereits eingebaut. Die aufgerufene Funktion `createGLContext` ist im file `shaderUtils.js` definiert. Im JavaScript code:

```
function createGLContext(canvas) {
  // get the gl drawing context
  var context = canvas.getContext("webgl");
  if (!context) {
    alert("Failed to create GL context");
  }
  // wrap the context to a debug context to get error messages
  return WebGLDebugUtils.makeDebugContext(context);
}
```

Aufgabe 1: Im Code oben wird der Hintergrund des Canvas rot eingefärbt, ändern sie dies auf einen hellgrauen Hintergrund.

Frage: Wieso steht `gl.clearColor(...)` in der Funktion `initGL()`, `gl.clear(...)` aber in `draw()`?

Schritt 2: Zeichnen eines Rechtecks

Shader Programme

Als nächstes soll ein Rechteck gezeichnet werden. Dazu müssen nun der Vertex Shader und der Fragment Shader implementiert werden, da WebGL eine vollständig programmierbare Grafik Pipeline beinhaltet. Am einfachsten werden die Shader von externen Files geladen. Im JavaScript File `shaderUtils.js` steht eine Funktion `loadAndCompileShaders()` zur Verfügung, die die Shaders lädt, kompiliert und zusammenfügt (linking), dies ist im Code der Funktion `loadAndCompileShaders()` ersichtlich, die auch im file `shaderUtils.js` definiert ist. Falls es beim Kompilieren der Shader zu einem Problem kommt, wird ein Fenster mit dem Fehler angezeigt.

```
function loadAndCompileShaders(gl, vertexShaderFileName, fragmentShaderFileName
    ) {
  "use strict";
  var vertexShaderSource = loadResource(vertexShaderFileName);
  var fragmentShaderSource = loadResource(fragmentShaderFileName);
```

```
  if (vertexShaderSource === null || fragmentShaderSource === null) {
    console.log("Could not load shader files");
    return false;
  }
  var vertexShader = gl.createShader(gl.VERTEX_SHADER);
  gl.shaderSource(vertexShader, vertexShaderSource);
  gl.compileShader(vertexShader);
  if (!gl.getShaderParameter(vertexShader, gl.COMPILE_STATUS)) {
    alert("Vertex Shader Error: " + gl.getShaderInfoLog(vertexShader));
    return false;
  }

  var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
  gl.shaderSource(fragmentShader, fragmentShaderSource);
  gl.compileShader(fragmentShader);

  if (!gl.getShaderParameter(fragmentShader, gl.COMPILE_STATUS)) {
    alert("Fragment Shader Error: " + gl.getShaderInfoLog(fragmentShader));
    return false;
  }
  return setupProgram(gl, vertexShader, fragmentShader);
}
```

Der Vertex und der Fragment Shader werden in der OpenGL Shading Language (GLSL) implementiert. Diese benutzt eine C ähnliche Syntax.

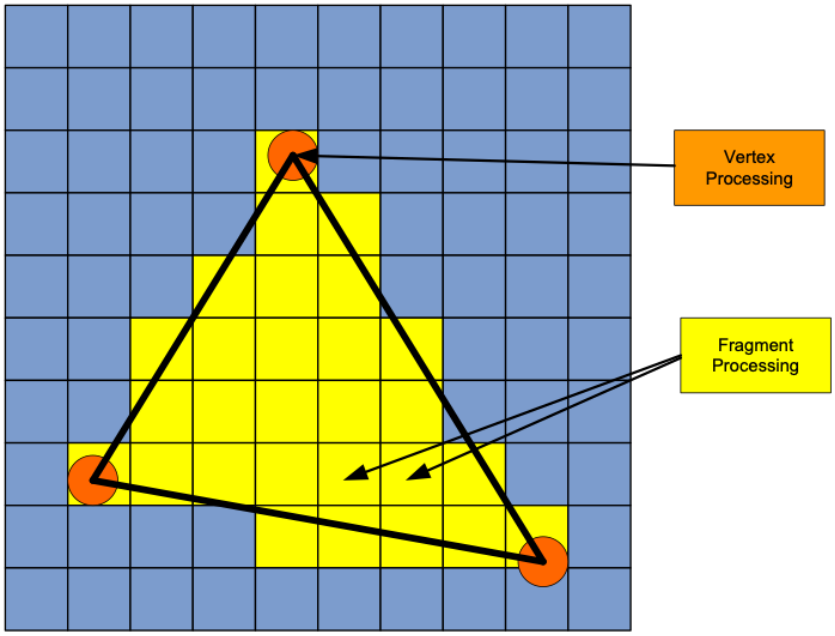

Abbildung 1: Funktionsweise der Shader. Der Vertex Shader wird für jeden Vertex aufgerufen, der Fragment Shader für jeden Pixel.

Der Vertex Shader wird für jeden Vertex aufgerufen. Seine Hauptaufgabe besteht darin die Position des Vertices auf dem Canvas zu berechnen, diese Position wird in sogenannten *normalized screen coordinates* angegeben, die von -1 zu +1 gehen.

```
attribute vec2 aVertexPosition;

void main() {
    gl_Position = ...;
}
```

Der Fragment Shader wird für jeden zu zeichnenden Pixel (Fragment) aufgerufen. Seine Hauptaufgabe besteht darin die Farbe des Pixels zu setzen.

```
precision mediump float;

void main() {
    gl_FragColor = ...;
}
```

Zeichnen von Objekten

Zum Zeichnen einer Figur muss zuerst spezifiziert werden, wie die Information vom JavaScript Program zu den Shader Programmen gelangt. Für diese Übung möchten wir Vertex Positionen übergeben, die dann als Linien oder Dreiecke gezeichnet werden. Wir benutzen daher eine Variante, in der ein ganzer Buffer von Werten übergeben werden kann. In einer anderen Variante könnten wir jeweils einzelne Werte übergeben, diese eignet sich jedoch nicht für Positionen.

Es ist das folgende Vorgehen notwendig:

1. Spezifikation des entsprechenden Attributes im Vertex Shader Program
2. Abfragen des Index dieses Attributs im JavaScript Program
3. Erzeugen der Vertex Daten (als `Float32Array`)
4. Erzeugend des WebGL Buffers
5. Verbinden des Buffers mit dem Attribut Index
6. Zeichnen des Arrays

Im Beispiel des Vertex Shaders oben ist bereits eine Attribut Variable `aVertexPosition` definiert (Schritt 1). Um dieser Variablen Werte zuordnen zu können, muss sie im JavaScript Program abgefragt werden (Schritt 2):

```
// we keep all local parameters for the program in a single object
var ctx = {
    shaderProgram: -1,
    aVertexPositionId: -1
};
...
function setupAttributes() {
  // finds the index of the variable in the program
  ctx.aVertexPositionId = gl.getAttribLocation(ctx.shaderProgram, "
    aVertexPosition");
}
```

Der Vertex Buffer muss erzeugt und mit einem Float32Array Objekt gefüllt werden (Schritte 3 und 4):

```
// we keep all the parameters for drawing a specific object together
var rectangleObject = {
    buffer: -1
};
...
function setupBuffers() {
  rectangleObject.buffer = gl.createBuffer();

  var vertices = [
    0,0,
    ...
  ];
  gl.bindBuffer(gl.ARRAY_BUFFER, rectangleObject.buffer);
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
}
```

Nun muss noch der Buffer mit dem Attribut verbunden werden und es muss in WebGL gesetzt werden, dass die Werte für das Attribut vom Buffer übernommen werden soll (Schritt 5)

```
function draw() {
  gl.clear(gl.COLOR_BUFFER_BIT);

  gl.bindBuffer(gl.ARRAY_BUFFER, rectangleObject.buffer);
  gl.vertexAttribPointer(ctx.aVertexPositionId, 2, gl.FLOAT, false, 0, 0);
  gl.enableVertexAttribArray(ctx.aVertexPositionId);

  ...
}
```

Schlussendlich kann das Array gezeichnet werden (Schritt 6). WebGl wird nun für jeden Wert des Arrays den Vertex Shader aufrufen.

```
function draw() {
  ...
  gl.drawArrays(gl.LINE_LOOP,0,4);
}
```

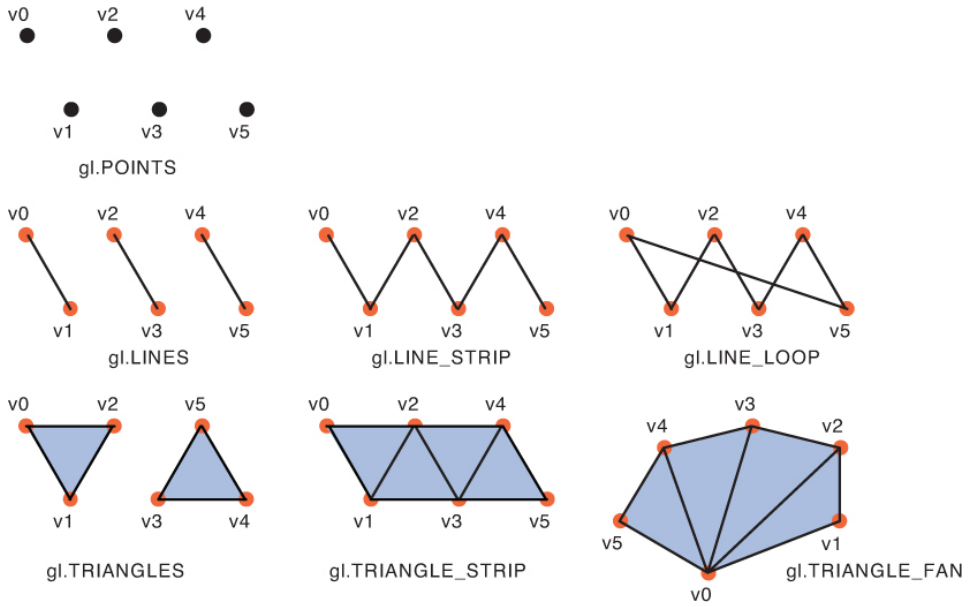

Abbildung 2: Die verschiedenen Parameter Werte von drawArrays werden verwendet um Punkte, Linien und Dreiecke darzustellen.

Aufgabe 2: Fügen sie die vorgegebenen Programmteile zusammen und ergänzen Sie sie, sodass ein weisses Rechteck (ungefüllt) auf dem Hintergrund gezeichnet wird.

Aufgabe 3: Verändern sie das Program, sodass ein gefülltes Rechteck gezeichnet wird. Dazu müssen sie `gl.drawArrays()` eine andere Konstante übergeben. Diese sind in Abbildung 2 spezifiziert.

Falls Sie bereits fertig sind, können Sie mit den folgenden Aufgaben fortfahren, die dann auch Teil der nächsten Übung sind. Dort werden die dazu benötigten Schritte dann genauer erklärt.

Aufgabe 4*: Ergänzen sie das Program, sodass sie die Farbe des Rechtecks übergeben können.

Aufgabe 5*: Zeichnen sie mehrere Rechtecke in verschiedenen Farben.