HOCHSCHULE FURTWANGEN UNIVERSITY | HFU

Rainer Duda

presents

Zertifikat "Digital Games" approved

Fun with Echtzeit-Computergrafik

reloaded

WP-2P | Grafikpipeline Implementierung

Agenda – Grafikpipline Implementierung

- Praktische Vertiefung
  - HTML Boilerplate
    - Einbindung jQuery
    - Einbindung eigener JS Datei
  - JS Strukturaufbau
    - Erzeugung eines WebGL Kontext
    - Definition von Vertices / eines Rechtecks
    - Definition der Geometrie-Buffer
    - Definition von Shader Quellcode
    - Erstellung der Shader Objekte
    - Kompilierung der Shader
    - Erzeugung eines Shader Programms
    - Linking des Shader Programms
    - Verknüpfung der Attribute
    - Rendering

HTML Boilerplate

- Wir benötigen ein HTML Grundgerüst sowie eine geeignete / zugrundeliegende Datenstruktur
  - In einem neuen Ordner werden zunächst erstellt:
    - Eine leere **index.html** Datei.
    - Ein weiterer **Ordner** für die JavaScript Dateien.
      - Darin eine **main.js** Datei
    - Für die Realisierung des Projekts wird in der praktischen Erfahrung Visual Studio Code mit der Extension Live Server verwendet.
      - Durch die Öffnung der leeren index.html Datei in VS Code, kann eine Boilerplate HTML Datei durch die Eingabe eines **!** nebst Bestätigung mit **Enter** erstellt werden.
      - Wir müssen noch jQuery von einer externen Quelle aus einbinden.
      - Eine Seite kann erst dann sicher manipuliert werden, wenn das Dokument "bereit" ist. jQuery erkennt diesen Bereitschaftszustand für uns. Code, der in $(document).ready() enthalten ist, wird nur ausgeführt, wenn das Document Object Model (DOM) der Seite bereit für die Ausführung von JavaScript-Code ist.
    - Die index.html Datei kann geprüft werden mittels rechter Maustaste und Klick auf "Open in Live Server".

HTML Boilerplate

- Wir benötigen ein HTML Grundgerüst sowie eine geeignete / zugrundeliegende Datenstruktur

```
<!DOCTYPE html>
<html lang="de">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>RTCG - WP-02P</title>
</head>

<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js"></script>

<script type="text/javascript">
    $(document).ready(function () {
        console.log( "Document bereit!" );
    });
</script>

<body>

</body>

</html>
```

HTML Boilerplate

- Wir benötigen ein HTML Grundgerüst sowie eine geeignete / zugrundeliegende Datenstruktur
  - Unsere HTML Boilerplate wird nun nachfolgend um zwei Dinge erweitert:
    - Ein Canvas Element nebst ID auf Basis von CSS Stils.
      - Der Einfachheit halber erstellen wir den Stil direkt in der HTML Datei unterhalb des Titels und setzen die Größe auf die Größe, die das Browserfenster maximal zulässt inkl - dynamischer Anpassung.
      - Innerhalb der Body Tags wird dem Canvas eine Auflösung von 1920 x 1080 Pixel zugewiesen.
    - Ein Script Tag, der auf unsere eigens erstellte JavaScript Datei (main.js) zeigt.

HTML Boilerplate

- Wir benötigen ein HTML Grundgerüst sowie eine geeignete / zugrundeliegende Datenstruktur

```
<!DOCTYPE html>
<html lang="de">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>RTCG - WP-02P</title>
    <style>
        my_canvas {
            height: 100vw;
            width: 100vh;
        }
    </style>
</head>

<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js"></script>
<script type="text/javascript">
    $(document).ready(function () {
        console.log( "Document bereit!" );
    });
</script>

<body>
    <canvas id="my_canvas" width=1920 height=1080></canvas>
</body>
<script src="/JS/main.js"> </script>
</html>
```

JS Strukturaufbau

- Nun müssen wir unserer JS Datei (main.js) eine Struktur zuweisen
  - Wir öffnen die Datei und struktuieren sie wie folgt:
  - Ein WebGL Programm arbeitet als State-Machine.
  - Daten die die Position und Farbe der Vertices beschreiben, wird über die Zeit geupdated.
  - Wir werden die Eingabedaten der Shader modifizieren, um den Status des WebGL Kontexts über die Zeit hinweg zu beeinflussen.
  - Wir beziehen uns auf die Programmable-Shader Pipeline des "Modern OpenGL".
  - *Sie sehen, dass über der Deklaration der Funktion bereits der Aufruf stattfindet. Da der Browser den JS Code kompiliert, wenn er vom Webserver empfangen wurde. Der Aufruf kann sowohl davor als auch nach der Deklaration stattfinden, da der Browser bereits die Referenz der Funktion gespeichert hat vor jeglicher Skript-Ausführung.*

```
main();
function main(){

    /* ======== Erzeugung eines WebGL Kontext ======== */

    /* ==== Definition und Speicherung von Geometrie ==== */

    /* ==== Definition von Front-Face Vertices ==== */

    /* ==== Definition von Front-Face Puffern ==== */


    /* ======== Shader ======== */

    /* ==== Definition der Shader-Quelle ==== */

    /* ==== Erzeugung Shader ==== */

    /* ==== Shader Kompilierung ==== */

    /* ==== Erzeugung des Shader-Programms ==== */

    /* ==== Linking des Shader-Programms ==== */

    /* ==== Verknüpfung der Attribute mit dem Vertex Shader ==== */


    /* ======== Rendering ======== */

    /* ==== Zeichnung der Punkte auf den Bildschirm ==== */
}
```

JS Strukturaufbau

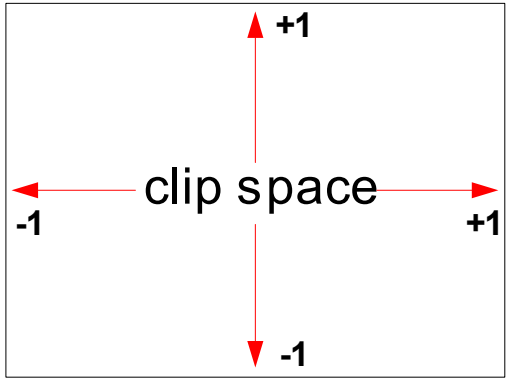
- Erzeugung eines WebGL Kontext
  - Hinter dem WebGLRenderingContext verbirgt sich ein Interface, durch das der Browser Zugang erhält zu der Funktionalität der WebGL Bibliothek.
  - Verständlicherweise können Anwender*Innen den Content unserer Applikation nicht laden, wenn Geräte oder Browser nicht WebGL kompatibel sind.
    - Zur Prüfung, ob ein Client / User Agent WebGL Content laden kann, wird eine Abfrage implementiert.

```
/* ======== Erzeugung eines WebGL Kontext ======== */

const canvas = document.getElementById('my_canvas');
const gl = canvas.getContext('webgl2');
if (!gl) {
    console.log('WebGL nicht verfügbar!');
} else {

    console.log('WebGL verfügbar!');
}
```

JS Strukturaufbau

- Erstellung eines Rechtecks Teil 1
  - Zunächst wird ein Array definiert, dass 6 Vertices zur Darstellung von 2 Dreiecken beinhaltet.
  - Neben X,Y Werten dient die dritte Koordinate als Z-Position.
  - Die Vertice Positionen werden im Raum von -1 und 1 definiert -> Clip Space.
    - Clip Space Koordinaten normalisieren die eigentlichen Positionen der Vertices in den normalisierten Raum -1 bis 1, wobei die Z-Achse sich negativ bewegt ausgehend vom Wert 0.
    - Beim NDC Raum werden hingegen alle 3 Achen in einen normalisierten Wertebereich von -1 und 1 transformiert.

```
/* ==== Definition und Speicherung von Geometrie ==== */

    /* ==== Definition von Front-Face Vertices ==== */

const rechteck_01 = [

    -0.25, -0.25, -0.25,
    0.25, -0.25, -0.25,
    0.25, 0.25, -0.25,

    -0.25, -0.25, -0.25,
    -0.25, 0.25, -0.25,
    0.25, 0.25, -0.25

];
```



JS Strukturaufbau

- Erstellung eines Rechtecks Teil 2
  - Wir haben nun Paar von Koordinaten erzeugt und müssen diese abspeichern.
    - Als erstes wird ein leeres / neues Buffer Objekt erzeugt.
    - Über ein Binding wird das leere Buffer Objekt (Konzept eines Puffers) mit dem eigentlichen Speicherort verknüpft.
    - Buffer sind der Weg, um Vertex- und andere Daten pro Vertex auf die GPU zu bekommen.
      - gl.createBuffer erzeugt einen Buffer.
      - gl.bindBuffer setzt diesen Buffer als den zu bearbeitenden Buffer.
      - gl.bufferData kopiert Daten in den Buffer.
    - Float32Array() castet unser Array, bestehend aus 6 Float Elementen in ein Array aus bestehend aus 32-Bit Float Werten für effizientere mathematische Berechnungen.
    - STATIC_DRAW sagt der GPU, dass die Daten in unserem Puffer nicht mehr als einmal modifiziert werden und deshalb geschrieben werden sollen. Die Speicherung der Pufferdaten in den GPU Speicher und keine weitere Modifizierung ermöglicht Gewinn in der Performance.

```
    /* ==== Definition und Speicherung von Geometrie ==== */

        /* ==== Definition von Front-Face Vertices ==== */

    const rechteck_01 = [

        -0.25, -0.25, -0.25,
        0.25, -0.25, -0.25,
        0.25, 0.25, -0.25,

        -0.25, -0.25, -0.25,
        -0.25, 0.25, -0.25,
        0.25, 0.25, -0.25

    ];

    /* ==== Definition von Front-Face Puffern ==== */

    const rechteck_puffer = gl.createBuffer();

    gl.bindBuffer(gl.ARRAY_BUFFER, rechteck_puffer);

    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(rechteck_01), g
l.STATIC_DRAW);
```

JS Strukturaufbau

- Shader Quellcode Vertex Shader
  - Der Vertex-Shader ist eine Funktion, die wir in GLSL schreiben.
  - Sie wird für jeden Vertex einmal aufgerufen.
  - Es können nun Berechnungen durchgeführt und daraufhin die spezielle Variable gl_Position mit einem Clip Space-Wert für den aktuellen Vertex gesetzt werden.
  - Die GPU nimmt diesen Wert und speichert ihn intern.
  - Vorab definieren wir eine Variable, die die Position unserer Vertices beinhalten soll.
  - *Noch ist die Variable nicht mit unserem Rechteck, bestehend aus zwei Dreiecken verknüpft -> erfolgt später.*

```
/* ======== Shader ======== */

/* ==== Definition der Shader-Quelle ==== */

const vertex_source = `

    attribute vec4 rechteck_position;

    void main(){

        gl_Position = rechteck_position;

    }

`;
```

JS Strukturaufbau

- Shader Quellcode Fragment Shader
  - Im Fragment Shader ist ebenfalls eine spezielle Variable gl_FragColor vorhanden, die RGBA Werte zurückliefert.
  - Der Einfachheit halber legen wir hier nur einen Farbwert fest.

```
/* ======== Shader ======== */

/* ==== Definition der Shader-Quelle ==== */

const vertex_source = `

    attribute vec4 rechteck_position;

    void main(){

        gl_Position = rechteck_position;

    }

`;

const fragment_source = `

    void main() {

        gl_FragColor = vec4(1,0,0,1);

    }

`;
```

JS Strukturaufbau

- Erzeugung der Shader Objekte
  - Zunächst müssen wir ein OpenGL-"Shader-Objekt" erstellen, das angibt, um welche Art von Shader es sich handelt.
    - Vertex-Shader (GL_VERTEX_SHADER)
    - Fragment-Shader (GL_FRAGMENT_SHADER)
  - Daraufhin weisen wir dem jeweiligen Shader-Objekt den passenden Quellcode zu.

```
/* ==== Erzeugung Shader ==== */

const vertex_shader = gl.createShader(gl.VERTEX_SHADER);
const fragment_shader = gl.createShader(gl.FRAGMENT_SHADER);

gl.shaderSource(vertex_shader, vertex_source);
gl.shaderSource(fragment_shader, fragment_source);
```

JS Strukturaufbau

- Kompilierung der Shader Objekte
  - Für das Kompilieren steht uns ebenfalls eine vordefinierte Funktion zur Verfügung namens compileShader().
  - Die Funktion kompiliert die Quellcode-Strings, die im jeweiligen angegebenen Shader-Objekt gespeichert wurden.
  - Wird öfter vorkommen, dass das Kompilieren fehl schlägt, daher eine Nebeninfo:
    - *Der Kompilierungsstatus wird als Teil des Status des Shader-Objekts gespeichert. Der Wert wird auf GL_TRUE gesetzt, wenn der Shader ohne Fehler kompiliert wurde und einsatzbereit ist, andernfalls auf GL_FALSE. Er kann durch den Aufruf von glGetShaderiv mit den Argumenten shader und GL_COMPILE_STATUS abgefragt werden.*

```
/* ==== Shader Kompilierung ==== */

gl.compileShader(vertex_shader);
gl.compileShader(fragment_shader);
```

JS Strukturaufbau

- Erzeugung des Shader Programms
  - gl.CreateProgram erzeugt ein leeres Programmobjekt und gibt einen Wert ungleich Null zurück, mit dem es referenziert werden kann.
  - Ein Programmobjekt ist ein Objekt, an das **Shader-Objekte** angehängt werden können.
  - Das stellt einen Mechanismus zur Verfügung, um die Shader-Objekte anzugeben, die zur Erstellung eines Programms verknüpft werden.
  - Über gl.attachShader wird ein Shader-Objekt an ein Programm-Objekt angehangen.
  - Wir übergeben als Argumente einmal das Shader Programm und den jeweiligen Shader, der angehangen werden soll.

```
/* ==== Erzeugung des Shader-Programms ==== */

const shader_program = gl.createProgram();
gl.attachShader(shader_program, vertex_shader);
gl.attachShader(shader_program, fragment_shader);
```

JS Strukturaufbau

- Linking des Shader Programms
  - Ein WebGL Shader Programm Objekt besteht in unserem Fall aus einer Kombination von zwei kompilierten WebGL Shadern.
  - Bestehend aus einem Vertex-Shader und einem Fragment-Shader (beide in GLSL geschrieben).
  - Diese werden dann zu einem brauchbaren Programm über gl.linkProgram gelinkt.
  - Da theoretisch mehrere Shader Programme exisiteren, muss noch über gl.useProgram() mitgeteilt werden, welches Shader Programm für die Darstellung / Färbung genutzt werden soll.

```
/* ==== Linking des Shader-Programms ==== */

gl.linkProgram(shader_program);
gl.useProgram(shader_program);
```

JS Strukturaufbau

Verknüpfung der Attribute

- Nun folgt der Austausch der Daten aus dem "normalen" JavaScript-Code und den WebGL-Shadern.
- getAttribLocation() ermöglicht anhand des Variablennamens rechteck_position eine Zugriffsmöglichkeit auf eine Eingangsvariable des Vertex-Shaders.
- Datentyp der Vertices ist das WebGL-spezifische Float32Array. Um die Berechnungen auf die GPU zu verlagern, müssen diese Daten in den Grafikspeicher übertragen werden.
- Hierzu wird mit gl.createBuffer zuvor ein neuer Speicherbereich angelegt, der dann mit gl.bindBuffer aktiviert und zuletzt mit bufferData befüllt wurde.
- Der Inhalt von rechteck_position wird hier in den Grafikspeicher kopiert.
  Nun müssen wir WebGL noch mitteilen, wofür der gerade aktivierte und befüllte Puffer verwendet werden soll: gl.vertexAttribPointer() & gl.enableVertexAttribArray()
- Der erste Parameter position_attribut verweist auf die Eingangsvariable des Vertex-Shaders, "3" ist die Anzahl der Werte pro Element (3 Koordinatenwerte pro Punkt) und "gl.Float" ist der Datentyp.
- enableVertexAttribArray() aktiviert die Verwendung des Puffers für folgende Zeichenfunktionen (Rendering).

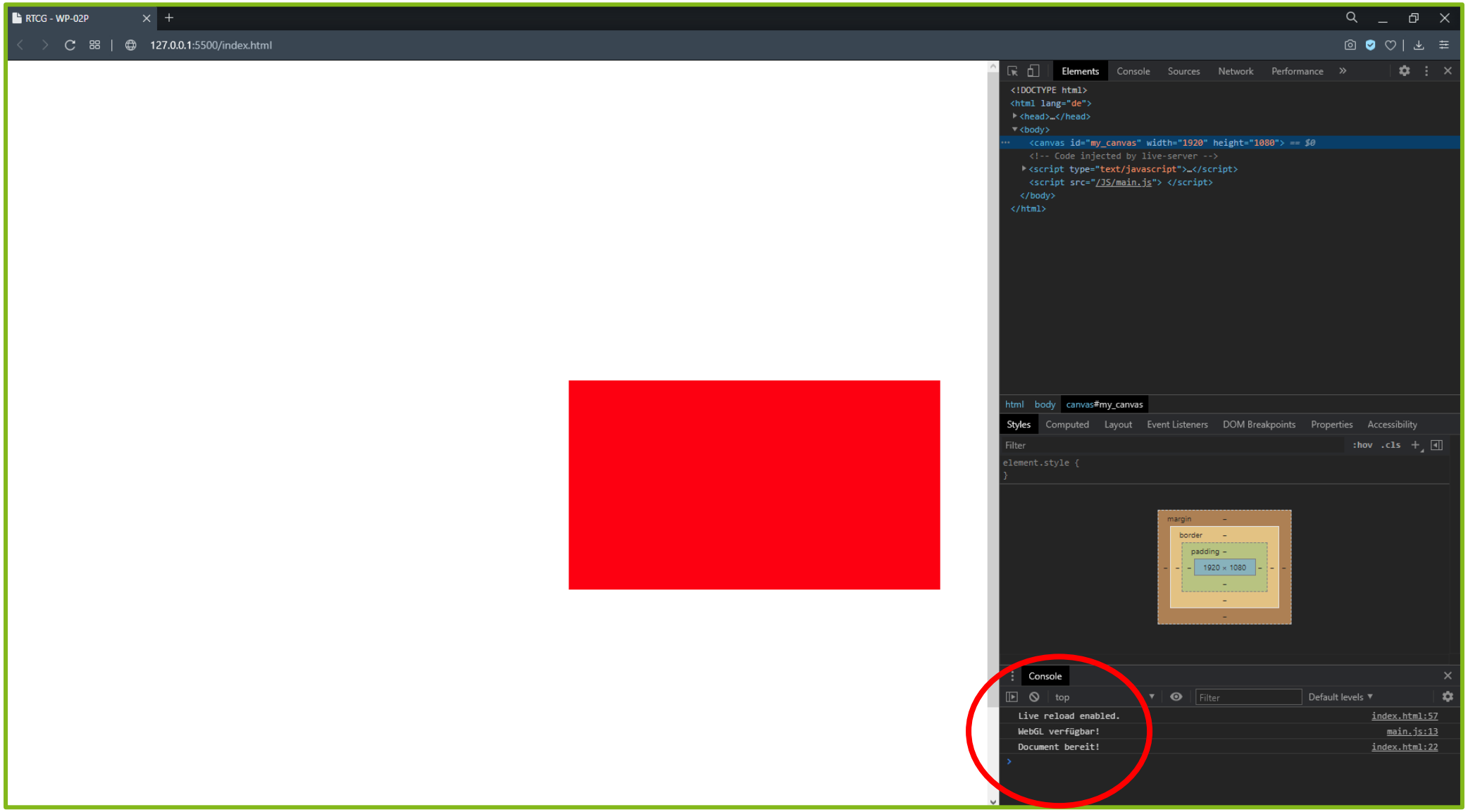
```
    /* ==== Verknüpfung der Attribute mit dem Vertex Shader ==== */

    const position_attribut = gl.getAttribLocation(shader_program,
"rechteck_position");

    gl.bindBuffer(gl.ARRAY_BUFFER, rechteck_puffer);

    gl.vertexAttribPointer(position_attribut, 3, gl.FLOAT, false, 0
,0);

    gl.enableVertexAttribArray(position_attribut);
```

JS Strukturaufbau

- Rendering
  - Die beiden ersten Funktion legen die Hintergrundfarbe der 3D-Szene als RGB-Wert fest und löschen die komplette Szene in dieser Farbe.
  - Mittels gl.drawTriangles() werden aus den zuletzt aktivierten Buffer-Daten Dreiecke generiert (in diesem Fall zwei).
  - Der erste Parameter gl.TRIANGLES kann ebenso in gl.POINTS oder gl.LINE_STRIP geändert werden. Das Resultat sind andere Darstellungen.
  - Der zweite und dritte Parameter geben Start- und Anzahl der Array-Elemente an, die gezeichnet werden sollen.

```
/* ======== Rendering ======== */

/* ==== Zeichnung der Punkte auf den Bildschirm ==== */

gl.clearColor(1,1,1,1);
gl.clear(gl.COLOR_BUFFER_BIT);

const mode = gl.TRIANGLES;
const first = 0;
const count = 6;

gl.drawArrays(mode,first,count);
```

Resultat


RTCG - WP-02P
127.0.0.1:5500/index.html
Elements
Console
Sources
Network
Performance
<!DOCTYPE html>
<html lang="de">
<head>…</head>
<body>
<canvas id="my_canvas" width="1920" height="1080"> == $0
<!-- Code injected by live-server -->
<script type="text/javascript">…</script>
<script src="/JS/main.js"> </script>
</body>
</html>
html
body
canvas#my_canvas
Styles
Computed
Layout
Event Listeners
DOM Breakpoints
Properties
Accessibility
Filter
:hov .cls
element.style {
}
margin
border
padding
1920 × 1080
Console
top
Filter
Default levels
Live reload enabled.
index.html:57
WebGL verfügbar!
main.js:13
Document bereit!
index.html:22