Rainer Duda

presents

Zertifikat "Digital Games" approved

Fun with Echtzeit-Computergrafik reloaded

WP-3P | Die dritte Dimension

Agenda – Die dritte Dimension

- Praktische Vertiefung
  - JS Strukturerweiterung **(main.js)**
    - Fehlerbehandlung (Vertex / Fragment Shader, Shader Programm)
    - Culling und Tiefentest
    - Würfeldefinition
    - Anpassung Draw-Befehl
    - Vorbereitung Animation
    - Transformationen
      - Definition einer perspektivischen Projektionsmatrix
      - Definition der Modelltransformationsmatrix
  - Helfer-Funktionen **(utilities.js)**
    - FPS Zähler

Strukturerweiterung (main.js) - Fehlerbehandlung

- Diskrepanzen bei der Shader Programmierung
  - Da syntaktische Fehler auftreten können, ist es ratsam, dass wir eine Fehlerabfrage bei der Shader Kompilierung positionieren.
    - Das WebGLRenderingContext.getShaderInfoLog liefert das Informationsprotokoll für das angegebene WebGLShader-Objekt zurück. Es enthält Warnungen, Debugging- und Kompilierinformationen.
    - Die Methode WebGLRenderingContext.getShaderParameter() der WebGL-API gibt Informationen über den angegebenen Shader zurück.
    - gl.COMPILE_STATUS: Gibt ein GLboolean zurück, das anzeigt, ob die letzte Shaderkompilierung erfolgreich war.
    - gl.SHADER_TYPE: Gibt eine GLenum zurück, die anzeigt, ob der Shader ein Vertex-Shader (gl.VERTEX_SHADER) oder ein Fragment-Shader (gl.FRAGMENT_SHADER) Objekt ist.
  - Ebenso kann es vorkommen, dass beim Linking des Shader Programms Fehler auftreten und auch da eine Fehlerabfrage positioniert werden sollte.

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

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

if (!gl.getShaderParameter(vertex_shader, gl.COMPILE_STATUS)) {
    console.error(gl.getShaderInfoLog(vertex_shader));
    throw new Error('Fehler beim Kompilieren des Vertex Shader');
  }

if (!gl.getShaderParameter(fragment_shader, gl.COMPILE_STATUS)) {
    console.error(gl.getShaderInfoLog(fragment_shader));
    throw new Error('Fehler beim Kompilieren des Fragment Shader');
  }
```

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

gl.linkProgram(shader_program);

if (!gl.getProgramParameter(shader_program, gl.LINK_STATUS)) {
    console.error(gl.getProgramInfoLog(shader_program));
    throw new Error('Fehler beim Linken des Shader Programms');
  }

gl.useProgram(shader_program);
```

Strukturerweiterung (main.js) - Fehlerbehandlung

- Diskrepanzen bei der Shader Programmierung
  - Ebenso kann es vorkommen, dass beim Linking des Shader Programms Fehler auftreten und auch da eine Fehlerabfrage positioniert werden sollte.
    - WebGLRenderingContext.getProgramParameter() der WebGL-API gibt Informationen über das angegebene Programm zurück.
    - gl.LINK_STATUS: Gibt ein GLboolean zurück, das angibt, ob der letzte Link-Vorgang erfolgreich war oder nicht.
    - gl.VALIDATE_STATUS: Gibt ein GLboolean zurück, das angibt, ob die letzte Validierungsoperation erfolgreich war oder nicht.
    - gl.ATTACHED_SHADERS: Gibt einen GLint zurück, der die Anzahl der angehängten Shader eines Programms angibt.
    - gl.ACTIVE_ATTRIBUTES: Gibt einen GLint zurück, der die Anzahl der aktiven Attributvariablen zu einem Programm anzeigt.
    - gl.ACTIVE_UNIFORMS: Gibt einen GLint zurück, der die Anzahl der aktiven Uniform-Variablen zu einem Programm anzeigt.

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

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

if (!gl.getShaderParameter(vertex_shader, gl.COMPILE_STATUS)) {
    console.error(gl.getShaderInfoLog(vertex_shader));
    throw new Error('Fehler beim Kompilieren des Vertex Shader');
  }

if (!gl.getShaderParameter(fragment_shader, gl.COMPILE_STATUS)) {
    console.error(gl.getShaderInfoLog(fragment_shader));
    throw new Error('Fehler beim Kompilieren des Fragment Shader');
  }
```

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

gl.linkProgram(shader_program);

if (!gl.getProgramParameter(shader_program, gl.LINK_STATUS)) {
    console.error(gl.getProgramInfoLog(shader_program));
    throw new Error('Fehler beim Linken des Shader Programms');
  }

gl.useProgram(shader_program);
```

Culling und Tiefentest

- Optimierungsbausteine
  - **Tiefenprüfung** (Depth Test). Damit wird sichergestellt, dass Dreiecke, die näher am Betrachter sind, die weiter entfernten blockieren.
  - **Rückseitenauslese** (Back-Face Culling). Eine gute Optimierung, um zu vermeiden, dass Dreiecke gezeichnet werden, die vom Betrachter abgewandt sind.
    - Ebenfalls nützlich, um Dreiecke nicht zu rastern, die bei einem vollständig geschlossenen Modell nie zu sehen sein werden.
    - Hinweis: Sie sollten dies nicht für offene Netze oder solche mit Transparenz verwenden.
    - Werden bei uns aktiviert, sobald wir den GL-Rendering-Kontext erhalten.
  - Wenn die Tiefenprüfung aktiviert ist, müssen wir auch den Tiefenpuffer vor dem Rendern löschen.

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

const canvas = document.getElementById('my_canvas');
const gl = canvas.getContext('webgl2');

gl.enable(gl.DEPTH_TEST);
gl.enable(gl.CULL_FACE);

if (!gl) {
    console.log('WebGL nicht verfügbar!');
} else {

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

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

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

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

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

Würfeldefinition

- Aktualisierung des Shaders zur Behandlung von Farben pro Vertex
  - Für unser Würfelobjekt benötigen wir abermals 3D Positionen der jeweiligen Vertices.
  - Des Weiteren geben wir zusätzlich noch Farbwerte per Vertex zu Visualisierungszwecken an.
  - Also aktualisieren wir unseren Vertex-Shader, um die Eingabefarbe einzubeziehen und leiten diese auch gleich an an den Fragment-Shader weiter.

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

const vertex_source = `

    attribute vec4 wuerfel_position;
    attribute vec3 wuerfel_eingabefarbwerte;
    varying vec4 wuerfel_farbwerte;

    void main(){

        gl_Position = wuerfel_position;
        wuerfel_farbwerte = vec4(wuerfel_eingabefarbwerte, 1);

    }

`;
```

Würfeldefinition

- Aktualisierung des Shaders zur Behandlung von Farben pro Vertex
  - Da wir im Vertex Shader Farbwerte definiert haben, müssen diese nun in den Fragment Shader gespeist und angwandt werden.
  - Hierzu wird zunächst eine Präzision festgelegt.
  - *Diese legt fest, wie viel Präzision die GPU bei der Berechnung von Fließkommazahlen verwendet. highp ist hohe Präzision und natürlich intensiver als mediump (mittlere Präzision) und lowp (niedrige Präzision).*
  - *Einige Systeme unterstützen highp überhaupt nicht, was dazu führt, dass Code auf diesen Systemen nicht ausgeführt werden kann.*
  - *Eine gute Faustregel ist: highp für Vertex-Positionen, mediump für Texturkoordinaten, lowp für Farben.*
  - *Nun müssen nur noch die Farbwerte an gl_FragColor übergeben werden.*

```
const fragment_source = `

        precision lowp float;
        varying vec4 wuerfel_farbwerte;

        void main() {

            gl_FragColor = wuerfel_farbwerte;

        }

    `;
```

Würfeldefinition

- Definition der Würfel-Mesh Daten (Position & Farbe) #01
  - Um zwei Eingänge einzuspeisen, haben wir zwei (von vielen) Möglichkeiten:
  1. Erstellung eines separaten Vertex Buffer Objects (VBOs) pro Eingang.
     - Das ist nützlich, wenn sich die Eingänge während der Programmausführung mit unterschiedlicher Häufigkeit ändern, z. B. wenn einer statisch ist und der andere sich bei jedem Frame ändert.
  2. Erstellung eines einzelnen VBO mit beiden Informationen ineinander verschachtelt.
     - Wir werden sowohl die Basisposition als auch die Farbe während des gesamten Programms konstant halten, daher ist die zweite Option sinnvoll.

```
const wuerfel = new Float32Array([

    // FRONT
    /* pos = */ -1, -1,  1, /* color = */ 1, 0, 0,
    /* pos = */  1, -1,  1, /* color = */ 1, 0, 0,
    /* pos = */  1,  1,  1, /* color = */ 1, 0, 0,

    /* pos = */ -1, -1,  1, /* color = */ 1, 0, 0,
    /* pos = */  1,  1,  1, /* color = */ 1, 0, 0,
    /* pos = */ -1,  1,  1, /* color = */ 1, 0, 0,

    // RECHTS
    /* pos = */  1, -1, -1, /* color = */ 0, 1, 0,
    /* pos = */  1,  1, -1, /* color = */ 0, 1, 0,
    /* pos = */  1,  1,  1, /* color = */ 0, 1, 0,

    /* pos = */  1, -1, -1, /* color = */ 0, 1, 0,
    /* pos = */  1,  1,  1, /* color = */ 0, 1, 0,
    /* pos = */  1, -1,  1, /* color = */ 0, 1, 0,

    // RÜCKSEITE
    /* pos = */  1, -1, -1, /* color = */ 0, 0, 1,
    /* pos = */ -1, -1, -1, /* color = */ 0, 0, 1,
    /* pos = */ -1,  1, -1, /* color = */ 0, 0, 1,

    /* pos = */  1, -1, -1, /* color = */ 0, 0, 1,
    /* pos = */ -1,  1, -1, /* color = */ 0, 0, 1,
    /* pos = */  1,  1, -1, /* color = */ 0, 0, 1,

    // LINKS
    /* pos = */ -1, -1, -1, /* color = */ 1, 1, 0,
    /* pos = */ -1, -1,  1, /* color = */ 1, 1, 0,
    /* pos = */ -1,  1,  1, /* color = */ 1, 1, 0,
```

Würfeldefinition

- Definition der Würfel-Mesh Daten (Position & Farbe) #02
  - Unser Vertex-Shader ist derjenige, der die Punkte erzeugt. Das bedeutet, dass wir die Eingabepunkte dort platzieren können, wo sie für Modellierungszwecke nützlich sind (z. B. um den Ursprung herum, wie wir es getan haben.)
  - Der Vertex-Shader kann sie an eine Stelle verschieben, die der OpenGL-Rasterizer erkennt.

```
    /* pos = */ -1, -1, -1, /* color = */ 1, 1, 0,
    /* pos = */ -1,  1,  1, /* color = */ 1, 1, 0,
    /* pos = */ -1,  1, -1, /* color = */ 1, 1, 0,

    // OBEN
    /* pos = */  1,  1, -1, /* color = */ 1, 0, 1,
    /* pos = */ -1,  1, -1, /* color = */ 1, 0, 1,
    /* pos = */ -1,  1,  1, /* color = */ 1, 0, 1,

    /* pos = */  1,  1, -1, /* color = */ 1, 0, 1,
    /* pos = */ -1,  1,  1, /* color = */ 1, 0, 1,
    /* pos = */  1,  1,  1, /* color = */ 1, 0, 1,

    // UNTEN
    /* pos = */ -1, -1, -1, /* color = */ 0, 1, 1,
    /* pos = */  1, -1, -1, /* color = */ 0, 1, 1,
    /* pos = */  1, -1,  1, /* color = */ 0, 1, 1,

    /* pos = */ -1, -1, -1, /* color = */ 0, 1, 1,
    /* pos = */  1, -1,  1, /* color = */ 0, 1, 1,
    /* pos = */ -1, -1,  1, /* color = */ 0, 1, 1,
]);
```

Würfeldefinition

- ## Verbinden der Daten mit den Shader-Eingängen

- Schließlich können wir nun beide Eingabedaten, die in einer einzigen Bytefolge verschachtelt sind, mit ihren entsprechenden Shader-Eingängen verbinden.
- Wir legen wie zuvor erwähnt nur einen Buffer auf der GPU an.
- Das Binding findet mit dem GL_ARRAY_BUFFER statt.
- Für die Positionsdaten zeigen wir wie gehabt auf den Buffer, nur diesmal mit 24 Schritten und einem Offset von 0.
- Das bedeutet, dass die relevanten Daten für die Position alle 24 Bytes (Floats sind 4 Bytes) zur Verfügung stehen, beginnend mit dem ersten Byte.
- Wir wiederholen den Vorgang für die Farbwerte. Wir verwenden dieses Mal den gleichen Schritt, aber diesmal einen Offset von 12, da die Farbwerte nach drei aufeinanderfolgenden Vertice-Positionsdaten auftreten.

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

    const vertex_buffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, vertex_buffer);
    gl.bufferData(gl.ARRAY_BUFFER, wuerfel, gl.STATIC_DRAW);
```

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

    const positions_attribut = gl.getAttribLocation(shader_program, 'wu
erfel_position');
    gl.vertexAttribPointer(positions_attribut, 3, gl.FLOAT, false, 24,
0);
    gl.enableVertexAttribArray(positions_attribut);

    const farb_attribut = gl.getAttribLocation(shader_program, 'wuerfel
_eingabefarbwerte');
    gl.vertexAttribPointer(farb_attribut, 3, gl.FLOAT, false, 24, 12);
    gl.enableVertexAttribArray(farb_attribut);
```

Anpassung Draw-Befehl

- Korrekte Anzahl an Dreiecken?
- Zusätzlich müssen wir nun dem **glDrawArrays**-Aufruf mitteilen, dass er auch tatsächlich die richtige Anzahl von Dreiecken rendern soll!
- An diesem Punkt zeigt das Programm nichts an, da die Art und Weise, wie wir den 3D-Eingabepunkt in eine 4D gl_Position umwandeln, bedeutet, dass wir im Wesentlichen **innerhalb** des Würfels sind und nach außen schauen.
- Back-face culling bedeutet, dass die Innenseiten der Würfelflächen nicht sichtbar sind.
- Wenn Sie das Culling ausschalten, sehen Sie ein einzelnes eingefärbtes Quadrat.

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

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

    const mode = gl.TRIANGLES;
    const first = 0;
    const count = (3 * 2 * 6);

    gl.drawArrays(mode,first,count);

}
```

Transformationen

- Rückblick
- Der letzte Schritt besteht darin, die Eingabepositionen, die in einer für die Modellierung geeigneten Weise spezifiziert wurden, entgegen zu nehmen und die Positionen auszugeben, die OpenGL korrekt rastern kann.
- Das bedeutet, dass endgültige Positionen erzeugt werden, die sich im Clip-Raum befinden.
- Werfen Sie bitte nochmals einen Blick in die Präsentationen der Impulsvorträge.

- **Vertex-Shader-Eingabedaten**. Die Daten können hier in jeder Form vorliegen und müssen nicht einmal Koordinaten sein. In der Praxis bestehen die Eingabedaten jedoch typischerweise aus 3D-Punkten in einigen Koordinaten, die sich leicht modellieren lassen, in diesem Fall werden die Punkte als im Modellraum liegend betrachtet.
- **Clip-Raum**. Dies ist die Ausgabe des Vertex-Shaders, in homogenen Koordinaten.
- **Normalisierte Gerätekoordinaten (NDC)**. Dieselben Punkte im Clip-Raum, aber von homogenen Koordinaten in kartesische 3D-Koordinaten umgewandelt, indem eine w-Division durchgeführt wird. Nur Punkte innerhalb eines 2×2×2-Würfels, der um den Ursprung zentriert ist (zwischen -1 und 1 auf allen Achsen), werden gerendert.
- **Bildschirmbereich (Screen Space)**. Hierbei handelt es sich um 2D-Punkte, bei denen der Ursprung, der früher in der Mitte des Raums lag, jetzt unten links auf der Leinwand liegt. Die z-Koordinate wird nur für die Tiefensortierung verwendet. Dies sind die Koordinaten, die am Ende für die Rasterung verwendet werden.

Transformationen

- Tipp
- Der Grund für unsere Aufschlüsselung ist, dass in vielen Tutorials neben Clip-Space, NDC und Screen-Space auch von Model-, World- und View-Space die Rede ist.
- Aber es ist wichtig zu erkennen, dass **alles vor dem Clip-Raum von uns abhängt**. Wir entscheiden, wie die Eingabedaten aussehen und wie sie in den Clipbereich konvertiert werden.
- Das bedeutet, dass wir die Eingaben im Modellraum spezifizieren und dann nacheinander die lokalen und Kameratransformationen anwenden.
- Für ein besseres Verständnis der OpenGL-APIs, ist es wichtig, zuerst zu verinnerlichen, was unter unserer Kontrolle ist und was OpenGL für uns bereitstellt.

Transformationen

Aktualisieren des Shaders zur Aufnahme von Transformationen

- Um von den im VBO angegebenen Modell-Raum-Koordinaten in den Clip-Raum zu konvertieren, müssen wir die Transformation in zwei Teile zerlegen:
  1. Eine Modell-zu-Welt-Transformation.
  2. Eine perspektivische Projektionstransformation.
- Diese Aufteilung funktioniert für unsere Zwecke, da wir den Würfel im Laufe der Zeit drehen und die perspektivische Projektion unverändert lassen.
- In realen Anwendungen würden wir zwischen den beiden Schritten eine Kameratransformation (oder Ansichtstransformation) einfügen, um eine bewegte Kamera zu ermöglichen, ohne die gesamte Welt verändern zu müssen.
- Wir fügen dem Vertex Shader zwei Uniform-Variablen hinzu.
- Eine Uniform Variable ist ein Teil der Eingabedaten, der für einen gesamten Zeichenaufruf festgelegt wird und über alle Scheitelpunkte in diesem Zeichenaufruf konstant bleibt.

- In unserem Fall sind die Transformationen zu jedem Zeitpunkt für alle Scheitelpunkte der Dreiecke, aus denen unser Würfel besteht, dieselben.
- Die beiden Variablen repräsentieren die beiden Transformationen und sind daher 4×4-Matrizen. Wir können die Transformationen verwenden, indem wir die Eingabeposition in homogene Koordinaten umwandeln und dann mit den Transformationsmatrizen vormultiplizieren.

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

const vertex_source = `

    attribute vec4 wuerfel_position;
    attribute vec3 wuerfel_eingabefarbwerte;
    varying vec4 wuerfel_farbwerte;
    uniform mat4 transformation;
    uniform mat4 projektion;

    void main(){

        gl_Position = projektion * transformation * wuerfel_position;
        wuerfel_farbwerte = vec4(wuerfel_eingabefarbwerte, 1);

    }

`;
```

Transformationen

Definition einer perspektivischen Projektionsmatrix

- Denken Sie daran, dass die Projektionsmatrix aufgrund der Art und Weise, wie unser Vertex-Shader eingerichtet ist, Punkte im Weltraum (wo die Eingabepunkte nach der Modelltransformation landen) nimmt und sie in den Clip-Raum setzt. Das bedeutet, dass die Projektionsmatrix zwei Aufgaben hat:
    1. Sie stellt sicher, dass alles, was gerendert werden soll, in den 2×2×2-Würfel gesetzt wird, der um den Ursprung zentriert ist. Das bedeutet, dass jeder Vertex, der gerendert werden soll, Koordinaten im Bereich von -1 bis 1 haben sollte.
    2. Wenn wir eine perspektivische Darstellung wünschen, stellen Sie die w-Koordinate so ein, dass die Division durch sie eine perspektivische Teilung bewirkt.
- Wichtig ist, dass wir die Matrix in Bezug auf einige Parameter definieren, z. B. den Abstand zu den nahen und fernen Clipping Planes und den Betrachtungswinkel.

```
/* ==== Definition einer perspektivischen Projektionsmatrix ==== */

const f = 100;
const n = 0.1;
const t = n * Math.tan(Math.PI / 8);
const b = -t;
const r = t * gl.canvas.clientWidth / gl.canvas.clientHeight;
const l = -r;

const projektion = new Float32Array([
2 * n / (r - l), 0, 0,  0,
0, 2 * n / (t - b), 0,  0,
(r + l) / (r - l), (t + b) / (t - b), -(f + n) / (f - n), -1,
0, 0, -2 * f * n / (f - n), 0,
]);
```

Transformationen

- ## Definition einer perspektivischen Projektionsmatrix

- Am wichtigsten zu Wissen ist, dass Matrizen in OpenGL in Spalten-Hauptreihenfolge definiert sind. Das heißt, die ersten vier Werte oben sind eigentlich die erste Spalte. In ähnlicher Weise trägt jeder vierte Wert zu einer einzigen Zeile bei, was bedeutet, dass die unterste Zeile tatsächlich (0, 0, -1, 0) darüber ist.
- Die ersten drei Zeilen verwenden die Begrenzungen, die die perspektivische Transformation definieren, um quasi die X-, Y- und Z-Koordinaten in den richtigen Bereich zu bringen.
- Die letzte Zeile multipliziert die Z-Koordinate mit -1 und setzt sie in die ausgegebene W-Koordinate ein. Dies bewirkt, dass das abschließende "Teilen durch w" Objekte mit einer negativeren Z-Koordinate verkleinert. *(Denken Sie daran, dass negative Z-Werte von uns weg zeigen)*
- Wenn wir eine orthografische Projektion durchführen würden, würden wir die W-Koordinate als 1 belassen.

```
/* ==== Definition einer perspektivischen Projektionsmatrix ==== */

const f = 100;
const n = 0.1;
const t = n * Math.tan(Math.PI / 8);
const b = -t;
const r = t * gl.canvas.clientWidth / gl.canvas.clientHeight;
const l = -r;

const projektion = new Float32Array([
2 * n / (r - l), 0, 0,  0,
0, 2 * n / (t - b), 0,  0,
(r + l) / (r - l), (t + b) / (t - b), -(f + n) / (f - n), -1,
0, 0, -2 * f * n / (f - n), 0,
]);
```

Transformationen

- ## Definition einer perspektivischen Projektionsmatrix

- Da wir nun eine Matrix im richtigen Format haben, können wir sie mit der Variablen im Shader verknüpfen.
- Dies funktioniert in etwa so, wie ein Array mit einer Attributvariablen verknüpft wird.
- Da eine Uniform jedoch im Wesentlichen ein Stück Daten enthält, das von allen Vertices gemeinsam genutzt wird (im Gegensatz zu einem Stück Daten pro Vertex), gibt es keinen Zwischenpuffer.
- Wir greifen symbolisch nach der Variable im Shader und senden die Daten.
- Wir müssen sicherstellen, dass dies außerhalb der Schleifenfunktion getan wird. Wir werden die Projektion während der gesamten Programmausführung nicht ändern, daher müssen wir diese Matrix nur einmal senden!

```
    /* ==== Definition einer perspektivischen Projektionsmatrix ==== */

    const f = 100;
    const n = 0.1;
    const t = n * Math.tan(Math.PI / 8);
    const b = -t;
    const r = t * gl.canvas.clientWidth / gl.canvas.clientHeight;
    const l = -r;

    const projektion = new Float32Array([
    2 * n / (r - l), 0, 0,  0,
    0, 2 * n / (t - b), 0,  0,
    (r + l) / (r - l), (t + b) / (t - b), -(f + n) / (f - n), -1,
    0, 0, -2 * f * n / (f - n), 0,
    ]);

    const uniform_projection = gl.getUniformLocation(shader_program, 'proj
ektion');
    gl.uniformMatrix4fv(uniform_projection, false, projektion);
```

Transformationen

- ## Definition der Modelltransformationsmatrix

- Eine herkömmliche Projektionsmatrix geht von Punkten im "Ansichtsraum" aus, d. h. in der Ansicht einer Kamera.
- Der Einfachheit halber nehmen wir an, dass sich die Kamera im Ursprung befindet und in die negative z-Richtung blickt (was im Wesentlichen der Ansichtsraum ist).
- Das Ziel besteht nun darin, die Shader-Eingabepositionen in Punkte zu transformieren, die innerhalb des sichtbaren Bereichs liegen, der durch das Sichtfeld und die Clipping-Ebenen der perspektivischen Projektion definiert ist.
- Wir wählen die folgenden Transformationen, in der Reihenfolge:
  - Skalieren um den Faktor 2.
  - Drehen um die Y-Achse um einen bestimmten Winkel.
  - Drehen um die X-Achse um denselben Winkel.
  - Verschieben entlang der Z-Achse um -9 Einheiten.
- Genutzt werden übliche Transformationsmatrizen, die in umgekehrter Reihenfolge multipliziert und schließlich transponiert wurden, um sie in die Spalten-Hauptreihenfolge zu bringen.
- *Beachten Sie, dass die Modelltransformationsmatrix innerhalb der Schleifenfunktion definiert wird, was die Verwendung des Zeitparameters zur Definition des Drehwinkels ermöglicht.*

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

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

    const uniform_transformation = gl.getUniformLocation(shader_program, '
transformation');

    requestAnimationFrame(t => loop(gl, t));

    function loop(gl, t) {

        gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

        /* == Transformationsmatrix == */
        const a = t * Math.PI / 4000;
        const c = Math.cos(a);
        const s = Math.sin(a);
        const x = 2;
        const tz = -9;

        const transformation = new Float32Array([
           c * x,  s * s * x, -c * s * x, 0,
               0,      c * x,      s * x, 0,
           s * x, -c * s * x,  c * c * x, 0,
               0,          0,         tz, 1,
        ]);

        gl.uniformMatrix4fv(uniform_transformation, false, transformation)
;

        /* == - == */

        const mode = gl.TRIANGLES;
        const first = 0;
        const count = (3 * 2 * 6);

        gl.drawArrays(mode,first,count);

      requestAnimationFrame(t => loop(gl, t));
    }
```

Helfer-Funktionen

- FPS Zähler
  - Zunächst wird eine neue Datei im JS Verzeichnis angelegt mit dem Namen: utilities.js
    - Der Aufbau der neuen Datei folgt der Konvention der main.js Datei.
  - Wie bereits bekannt, muss die neue Script Datei als Quelle angegeben werden.
  - Innerhalb der index.html Datei wird im Body ein FPS Div angelegt.

```
<body>
    <div>FPS: <span id="fps_counter"></span></div>

    <canvas id="my_canvas" width=1920 height=1080></canvas>

</body>

<script src="/JS/utilities.js"> </script>
```

```
utilities()

function utilities(){

/* ======== Erzeugung eines FPS Zählers ======== */

const fps_element = document.querySelector("#fps_counter");

let vergangen = 0;

function render(aktuell) {
  aktuell *= 0.001;
  const deltaTime = aktuell - vergangen;
  vergangen = aktuell;
  const fps = 1 / deltaTime;
  fps_element.textContent = fps.toFixed(1);

  requestAnimationFrame(render);
}
requestAnimationFrame(render);

}
```

Resultat