Go back. Print Page.

Beschreibung zu Labor 7

main()

Die main() Funktion stellt den initialen Einstieg in das Programm dar. Diese wird wie folgt aufgerufen:

```
// initial start point
window.onload = main;
```

Innerhalb der main() Funktion wird zunächst eine Referenz auf das Canvas-Element erzeugt und der WebGL-Kontext initialisiert.

```
const canvas = document.querySelector("#canvasElement");
// Initialize the GL context
const gl = canvas.getContext("webgl2");
```

Vertex- und Fragment-Shader

Nun werden die Shader initialisiert. Der Vertex-Shader hält dabei die Vertexkoordinaten, die Farbwerte und die Texturkoordinaten bereit. Die Positionen der Vertex ergeben sich dabei zusätzlich aus der ModelView- und Projection-Matrix (Kameraposition).

Der Fragment-Shader erhält die Farbwerte und Textur-koordinaten. Dabei wird die Textur als 2D-Textur angelegt und anschließend mit dem jeweiligen Farbwert (vColor) multipliziert. Somit entsteht die Überlagerung von Textur und Farbe auf dem Würfel.

```
// Vertex shader program
const vsSource = `
  attribute vec4 aVertexPosition;
  attribute vec4 aVertexColor;
  attribute vec2 aTextureCoord;

  uniform mat4 uModelViewMatrix;
  uniform mat4 uProjectionMatrix;

  varying lowp vec4 vColor;
  varying highp vec2 vTextureCoord;

  void main() {
    gl_Position = uProjectionMatrix * uModelViewMatrix * aVertexPosition;
    vColor = aVertexColor;
```

```
    vTextureCoord = aTextureCoord;
  }
`;

// Fragment shader program
const fsSource = `
  varying lowp vec4 vColor;
  varying highp vec2 vTextureCoord;

  uniform sampler2D uSampler;

  void main() {
    gl_FragColor = texture2D(uSampler, vTextureCoord) * vColor;
  }
`;
```

Shader-Programm (Initialisierung)

Nun wird das Shader-Programm initialisiert. Dazu werden die beiden zuvor definierten Shader geladen und in ein Programm zusammengefügt. Die Funktion wird unter initShaderProgram() genauer erklärt. Anschließend werden die Positionen der in den Shadern definierten Attribute und Uniformen für das Program definiert.

```
const shaderProgram = initShaderProgram(gl, vsSource, fsSource);
const programInfo = {
  program: shaderProgram,
  attribLocations: {
    vertexPosition: gl.getAttribLocation(shaderProgram, 'aVertexPosition'),
    vertexColor: gl.getAttribLocation(shaderProgram, 'aVertexColor'),
    textureCoord: gl.getAttribLocation(shaderProgram, 'aTextureCoord'),
  },
  uniformLocations: {
    projectionMatrix: gl.getUniformLocation(shaderProgram, 'uProjectionMatrix'),
    modelViewMatrix: gl.getUniformLocation(shaderProgram, 'uModelViewMatrix'),
    uSampler: gl.getUniformLocation(shaderProgram, 'uSampler'),
  },
};
```

Texturen laden

Bevor die Animation gestartet werden kann, müssen die Texturen initial geladen werden. Dazu wird die Funktion loadTexture() verwendet.

```
// load textures
const backgroundTexture = loadTexture(gl, 'background.png', true);
const cubeTexture = loadTexture(gl, 'bricks.png');
```

```
// flip the image
gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
```

Animation starten

Zum Schluss wird die Animation gestartet. Dazu wird die render() Funktion definiert. Diese wird mittels der Funktion requestAnimationFrame(render) wiederholt aufgerufen. Innerhalb der render() Funktion wird neben der Berechnung der Rotation des Würfels auch die Szene gezeichnet. Dafür wird die Funktion drawScene() verwendet.

```
// Draw the scene repeatedly
function render(now) {
  now *= 0.1;
  const deltaTime = now - then;
  then = now;

  drawScene(gl, programInfo, [cubeBuffers(gl), backgroundBuffers(gl)], [cubeTexture, backgroundTextur
  // apply rotation
  xRotation = 90;
  // xRotation += deltaTime;
  // yRotation += deltaTime * .7;
  requestAnimationFrame(render);
}
requestAnimationFrame(render);
```

initShaderProgram()

Zunächst werden die beiden Shader angelegt und kompiliert. Dies geschieht mit der Funktion loadShader().

```
const vertexShader = loadShader(gl, gl.VERTEX_SHADER, vsSource);
const fragmentShader = loadShader(gl, gl.FRAGMENT_SHADER, fsSource);
```

Anschließend werden die beiden Shader in ein Shader-Programm zusammengefügt, welches anschließend zurückgegeben wird.

```
const shaderProgram = gl.createProgram();
gl.attachShader(shaderProgram, vertexShader);
gl.attachShader(shaderProgram, fragmentShader);
gl.linkProgram(shaderProgram);

return shaderProgram;
```

loadShader(gl, type, source)

Die Funktion loadShader() erstellt einen neuen Shader mit dem übergebenen Typ (VERTEX_SHADER oder FRAGMENT_SHADER), kompiliert diesen und gibt ihn zurück.

```
const shader = gl.createShader(type);
// Send the source to the shader object
gl.shaderSource(shader, source);
// Compile the shader program
gl.compileShader(shader);
return shader
```

loadTexture(gl, url, useColor)

Für das Laden der Textur wird zunächst eine neue WebGL-Textur als 2D-Textur angelegt.

```
const texture = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, texture);
```

Damit die Textur geladen werden kann müssen mehrere Parameter festgelegt werden. Ein besonderer Fokus liegt dabei auf den Parametern internalFormat und srcFormat. Diese müssen zunächst übereinstimmen. Zusätzlich kann mittels gl.RGBA ein Bild inklusive Farben, mittels gl.LUMINANCE_ALPHA ein Bild ohne Farben geladen werden. Für den Hintergrund wird somit gl.RGBA verwendet, für den Würfel gl.LUMINANCE_ALPHA.

```
const internalFormat = useColor ? gl.RGBA : gl.LUMINANCE_ALPHA;
const srcFormat = useColor ? gl.RGBA : gl.LUMINANCE_ALPHA;
```

Im ladeprozess der Textur wird die Textur in den WebGL Kontext geladen (gl.texImage2D). Um Probleme mit dem Seitenverhältnis und der Auflösung der Textur zu vermeiden, wird hier eine Unterscheidung eingeführt und je nach dem eine Mipmap erstellt oder die Textur so skaliert, dass diese sich an das zu füllende Rechteck anpasst.

```
const image = new Image();
image.onload = () => {
  gl.bindTexture(gl.TEXTURE_2D, texture);
  gl.texImage2D(gl.TEXTURE_2D, level, internalFormat,
                srcFormat, srcType, image);

  // WebGL1 has different requirements for power of 2 images
  // vs non power of 2 images so check if the image is a
  // power of 2 in both dimensions.
  if (isPowerOf2(image.width) && isPowerOf2(image.height)) {
```

```
        // Yes, it's a power of 2. Generate mips.
        gl.generateMipmap(gl.TEXTURE_2D);
    } else {
        // No, it's not a power of 2. Turn off mips and set
        // wrapping to clamp to edge
        gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
        gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
        gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
    }
  };
  image.src = url;
```

Die Funktion isPowerOf2() prüft mittels bitweise AND ob die übergebene Zahl eine Zweierpotenz ist. Da im Binären Zweierpotenzen über ein einzelnes gesetztes Bit dargestellt werden, funktioniert die untenstehende Rechnung. Bsp: 0x100 & 0x011 = 0x000.

```
function isPowerOf2(value) {
  return (value & (value - 1)) == 0;
}
```

drawScene(gl, programInfo, buffers, texture)

Die drawScene() Funktion bringt die definierten Meshes mit ihrer Farbe und Textur in die Szene. Dafür muss zunächst ein Blick auf die Übergabeparameter buffers (ein Array mit den Meshes) geworfen werden. Die Erstellung der Meshes wird beispielsweise an der Funktion cubeBuffers() erklärt.

Kamera

Zunächst wird die Position der Kamera inklusive field of view, near und far definiert. Die Kamera wird dabei als Matrix definiert.

```
const fieldOfView = 45 * Math.PI / 180;   // in radians
const aspect = gl.canvas.clientWidth / gl.canvas.clientHeight;
const zNear = 0.1;
const zFar = 100.0;
const projectionMatrix = mat4.create();

// note: glmatrix.js always has the first argument
// as the destination to receive the result.
mat4.perspective(projectionMatrix,
                 fieldOfView,
                 aspect,
                 zNear,
                 zFar);
```

Objekt-Ebene

Die Position, von der aus die Objekte in die Szene gezeichnet werden, wird ebenfalls als Matrix definiert.

```
const modelViewMatrix = mat4.create();

mat4.translate(modelViewMatrix,     // destination matrix
               modelViewMatrix,     // matrix to translate
               [-0.0, 0.0, -6.0]);  // amount to translate
```

Darstellung der Objekte

Da mehrere Objekte dargestellt werden sollen, wird über die übergebene Buffer-Liste iteriert. Der folgende Ablauf ist somit für jedes Objekt identisch:

Laden der Werte aus der übergebenen Buffer-Liste

Die Werte werden aus der übergebenen Buffer-Liste geladen und an die Buffer der programInfo übergeben (somit die Shader). Beispiel für die Positionen:

```
const numComponents = 3;
const type = gl.FLOAT;
const normalize = false;
const stride = 0;
const offset = 0;
gl.bindBuffer(gl.ARRAY_BUFFER, buffers.position);
gl.vertexAttribPointer(
    programInfo.attribLocations.vertexPosition,
    numComponents,
    type,
    normalize,
    stride,
    offset);
gl.enableVertexAttribArray(
    programInfo.attribLocations.vertexPosition);
```

Abschließend wird WebGL angewiesen, die soeben beschriebenen Buffer (das program) zu verwenden:

```
gl.useProgram(programInfo.program);
```

Setzen der uniform attribute

Die projectionMatrix und modelViewMatrix werden im Shader als uniform Attribut übergeben. Die Zuweisung sieht wie folgt aus:

```
gl.uniformMatrix4fv(
    programInfo.uniformLocations.projectionMatrix,
    false,
    projectionMatrix);
gl.uniformMatrix4fv(
    programInfo.uniformLocations.modelViewMatrix,
    false,
    modelViewMatrix);
```

Anwenden der Textur

Die Textur wird nun an den Shader übergeben.

```
// Tell WebGL we want to affect texture unit 0
gl.activeTexture(gl.TEXTURE0);

// Bind the texture to texture unit 0
gl.bindTexture(gl.TEXTURE_2D, texture[i]);

// Tell the shader we bound the texture to texture unit 0
gl.uniform1i(programInfo.uniformLocations.uSampler, 0);
```

Zeichnen der Objekte

Die Objekte werden nun gezeichnet. Dazu muss die Anzahl der zu lesenden vertex-Punkte definiert werden. Da dieses Labor nicht modular ist, wird hier die Zuweisung der Vertex-Anzahl abhängig von der durchlaufenen Iteration über die übergebenen Buffer festgelegt (statische Zuweisung). Mit dem Befehl drawElements() werden die Objekte final in die Szene gezeichnet.

```
{
  const offset = 0;
  const vertexCount = i === 0 ? 36 : 6;
  const type = gl.UNSIGNED_SHORT;
  gl.drawElements(gl.TRIANGLES, vertexCount, type, offset);
}
```

cubeBuffers(gl)

Die Funktion cubeBuffers() definiert die Vertices, Farben und Textur-positionen für den Würfel.

Vertices

Zunächst werden die Eckpunkte der Würfel-Seiten definiert (positions).

```
// set positions
const positionBuffer = gl.createBuffer();

// Select the positionBuffer as the one to apply buffer
// operations to from here out.
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);

// Now create an array of positions for the cube.
const positions = [
  // Front face
  -0.5, -0.5,  0.5,
   0.5, -0.5,  0.5,
   0.5,  0.5,  0.5,
  -0.5,  0.5,  0.5,

  // Back face
  -0.5, -0.5, -0.5,
  -0.5,  0.5, -0.5,
   0.5,  0.5, -0.5,
   0.5, -0.5, -0.5,

  // Top face
  -0.5,  0.5, -0.5,
  -0.5,  0.5,  0.5,
   0.5,  0.5,  0.5,
   0.5,  0.5, -0.5,

  // Bottom face
  -0.5, -0.5, -0.5,
   0.5, -0.5, -0.5,
   0.5, -0.5,  0.5,
  -0.5, -0.5,  0.5,

  // Right face
   0.5, -0.5, -0.5,
   0.5,  0.5, -0.5,
   0.5,  0.5,  0.5,
   0.5, -0.5,  0.5,

  // Left face
  -0.5, -0.5, -0.5,
  -0.5, -0.5,  0.5,
  -0.5,  0.5,  0.5,
  -0.5,  0.5, -0.5,
];
```

Da der Würfel später rotiert werden soll, werden nun drei Transformationen um die jeweiligen Achsen definiert, welche auf die definierten Werte angewandt werden:

```
// rotate positions by phi degrees around the x axis
// deg to rad
const phi = xRotation * Math.PI / 180.0;
const c = Math.cos(phi);
const s = Math.sin(phi);
for (var i = 0; i < positions.length; i += 3) {
  const y = positions[i + 1];
  const z = positions[i + 2];
  positions[i + 1] = y * c - z * s;
  positions[i + 2] = y * s + z * c;
}
// rotate positions by theta degrees around the local y axis
// deg to rad
const theta = yRotation * Math.PI / 180.0;
const c2 = Math.cos(theta);
const s2 = Math.sin(theta);
for (var i = 0; i < positions.length; i += 3) {
  const x = positions[i];
  const z = positions[i + 2];
  positions[i] = x * c2 - z * s2;
  positions[i + 2] = x * s2 + z * c2;
}
// rotate positions by psi degrees around the z axis
// deg to rad
const psi = zRotation * Math.PI / 180.0;
const c3 = Math.cos(psi);
const s3 = Math.sin(psi);
for (var i = 0; i < positions.length; i += 3) {
  const x = positions[i];
  const y = positions[i + 1];
  positions[i] = x * c3 - y * s3;
  positions[i + 1] = x * s3 + y * c3;
}
```

Abschließen werden die Eckpunkte in den Buffer geschrieben:

```
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(positions), gl.STATIC_DRAW);
```

Farben

Anschließend werden die Farben der Vertices definiert (colors) und in einen neu erstellten Buffer geschrieben (colorBuffer):

```
const faceColors = [
  [0.1,  0.1,  0.1,  1.0],    // Front face: almost black
  [1.0,  0.0,  0.0,  1.0],    // Back face: red
  [0.0,  1.0,  0.0,  1.0],    // Top face: green
  [0.0,  0.0,  1.0,  1.0],    // Bottom face: blue
  [1.0,  1.0,  0.0,  1.0],    // Right face: yellow
  [1.0,  0.0,  1.0,  1.0],    // Left face: purple
];

// Convert the array of colors into a table for all the vertices.

var colors = [];

for (var j = 0; j < faceColors.length; ++j) {
  const c = faceColors[j];
  // Repeat each color four times for the four vertices of the face
  colors = colors.concat(c, c, c, c);
}

const colorBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors), gl.STATIC_DRAW);
```

Textur-Positionen

Die Textur-Positionen werden ebenfalls in einen Buffer geschrieben (textureCoordBuffer):

```
const textureCoordBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, textureCoordBuffer);

const textureCoordinates = [
  // Front
  0.0,  0.0,
  1.0,  0.0,
  1.0,  1.0,
  0.0,  1.0,
  // Back
  0.0,  0.0,
  1.0,  0.0,
  1.0,  1.0,
  0.0,  1.0,
  // Top
  0.0,  0.0,
  1.0,  0.0,
  1.0,  1.0,
  0.0,  1.0,
  // Bottom
  0.0,  0.0,
```

```
    1.0,  0.0,
    1.0,  1.0,
    0.0,  1.0,
    // Right
    0.0,  0.0,
    1.0,  0.0,
    1.0,  1.0,
    0.0,  1.0,
    // Left
    0.0,  0.0,
    1.0,  0.0,
    1.0,  1.0,
    0.0,  1.0,
  ];

  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(textureCoordinates),
                gl.STATIC_DRAW);
```

Indizes

Die Indizes der Dreiecke werden ebenfalls in einen Buffer geschrieben (indexBuffer). Damit kann der WebGL-Renderer die Dreiecke basierend auf den Vertices Koordinaten zeichnen:

```
  const indexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);

  // This array defines each face as two triangles, using the
  // indices into the vertex array to specify each triangle's
  // position.

  const indices = [
    0,  1,  2,      0,  2,  3,    // front
    4,  5,  6,      4,  6,  7,    // back
    8,  9,  10,     8,  10, 11,   // top
    12, 13, 14,     12, 14, 15,   // bottom
    16, 17, 18,     16, 18, 19,   // right
    20, 21, 22,     20, 22, 23,   // left
  ];

  gl.bufferData(gl.ELEMENT_ARRAY_BUFFER,
      new Uint16Array(indices), gl.STATIC_DRAW);
```

Die Definierten Buffer werden zu einem Objekt zusammengefasst und zurückgegeben:

```
  return {
    position: positionBuffer,
    color: colorBuffer,
```

```
      textureCoord: textureCoordBuffer,
      indices: indexBuffer,
    };
```