Computer Graphics
– WebGL2*

J.-Prof. Dr. habil. Kai Lawonn

Introduction

- So far, we learned the basics of OpenGL with C++
- Now, we want to transfer our knowledge to the web
- In the following, we will briefly learn how to build an OpenGL context in the browser

WebGL 2

- We will use WebGL 2 for our OpenGL context
- WebGL 2 uses OpenGL 3.0 ES (embedded systems)
- It is supported by Chrome, Edge, Firefox, etc.
- We can immediately start to program, we only need a browser (e.g., chrome) and a text editor (e.g., Notepad++)

Remark

- I will not go too much into detail, this lecture will only show you how to use WebGL and how to transfer your knowledge
- For further reading, I recommend:
  - Real-Time 3D Graphics with WebGL 2
  - WebGL Programming Guide
  - https://webgl2fundamentals.org/
  - Fun with WebGL 2
    https://github.com/sketchpunk/FunWithWebGL2
    https://www.youtube.com/playlist?list=PLMinhigDWz6emRKVkIEAaePW7vtIkaIF

Set Up

Tools

- Chrome
- Notepad++

Start

- First, we start with the html and head tag
- In the head, we define the title and set the style of our context:
  - We define a gray background and a solid black line as outline

```
<html>
        <head>
                <title>WebGL2 Start</title>
                        <style type="text/css">
                                body{background-color:#404040;}
                                canvas{border:3px solid black;}
                        </style>
        </head>
        …
```

Start

- Then, we define the body tag, here we have a script block that's where we put our WebGL code
- The context is drawn inside the canvas later, if not an error message appears:

```
…</head>
        <body>
                <script>

                </script>
                <canvas id="glcanvas" width="800" height="600">
                        HTML5 canvas is not supported by your browser.
                </canvas>
        </body>
</html>
```

Start

```
<html>
        <head>
                <title>WebGL2 Start</title>
                        <style type="text/css">
                                body{background-color:#404040;}
                                canvas{border:3px solid black;}
                        </style>
        </head>
        <body>
                <script>

                </script>
                <canvas id="glcanvas" width="800" height="600">
                        HTML5 canvas is not supported by your browser.
                </canvas>
        </body>
</html>
```

Start

- This is how it looks like

Start – Remark

- Pressing Crtl+Shift+J opens the developer console:
- (Potential error messages can be found here)

WebGL

- Inside the body block, we add WebGL

```
<script>
    var gl;
    function init() {
      const canvas = document.getElementById('glcanvas');
      if (!canvas) {
        console.error('HTML5 Canvas was not found!');
        return;
      }
      gl = canvas.getContext('webgl2');
    }
    window.onload = init;
</script>
<canvas id="glcanvas" width="800" height="600">
      HTML5 canvas is not supported by your browser.
</canvas>
```

WebGL

- Inside the body block, we add WebGL

```
<script>
    var gl;
    function init() {
      const canvas = document.getElementById('glcanvas');
      if (!canvas) {
        console.error('HTML5 Canvas was not found!');
        return;
      }
      gl = canvas.getContext('webgl2');
    }
    window.onload = init;
</script>
<canvas id="glcanvas" width="800" height="600">
      HTML5 canvas is not supported by your browser.
</canvas>
```

Remember?

- OpenGL needed the size of the rendering window
- And we set the background color

```
glViewport(0, 0, 800, 600);
glClearColor(0.2f, 0.3f, 0.4f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);
```

- WebGL:

```
gl.viewport(0, 0, canvas.width, canvas.height);
gl.clearColor(0.2, 0.3, 0.4, 1.0);
gl.clear(gl.COLOR_BUFFER_BIT);
```

WebGL

- Then the init function is changed:

```
function init() {
      const canvas = document.getElementById('glcanvas');

      if (!canvas) {
         console.error('HTML5 Canvas was not found!');
        return;
      }

      gl = canvas.getContext('webgl2');
      gl.viewport(0, 0, canvas.width, canvas.height);
      gl.clearColor(0.2, 0.3, 0.5, 1);
      gl.clear(gl.COLOR_BUFFER_BIT);
    }
```

Start

- Background acquired

Hello Triangle

Introduction

- Again, we follow the content of the second lecture
- But, we organize this better, by using functions:

```
function initBuffers() {}
function getShader() {}
function initProgram() {}
function draw() {}
function init() {}
```

Buffer

- We also need some global variables:

```
var gl, program, VAO, VBO, EBO, indices;
```

Buffer

- First, we define the vertices and the indices of the triangle:

```
function initBuffers() {

    const vertices = [
          -0.5,-0.5,0.0, // left
           0.5,-0.5,0.0, // right
           0.0, 0.5,0.0  // top
    ];

    indices = [0, 1, 2];
```

Buffer

- Create and bind the VAO, VBO and EBO (like many times before)

```
VAO = gl.createVertexArray();
gl.bindVertexArray(VAO);

VBO = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, VBO);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
gl.vertexAttribPointer(0, 3, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(0);

EBO = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, EBO);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices), gl.STATIC_DRAW);

gl.bindVertexArray(null);
gl.bindBuffer(gl.ARRAY_BUFFER, null);
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, null);
```

Buffer

- Instead of using (layout (location=0)):

```
gl.vertexAttribPointer(0, 3, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(0);
```

- We can also use:

```
program.aVertexPosition = gl.getAttribLocation(program, 'aVertexPosition');
gl.vertexAttribPointer(program.aVertexPosition, 3, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(program.aVertexPosition);
```

Shader

- First, we define the vertex shader:

```
<script id="vertex-shader" type="x-shader/x-vertex">
    #version 300 es
    precision mediump float;
    layout (location=0) in vec3 aVertexPosition;

    void main(void) {
      gl_Position = vec4(aVertexPosition, 1.0);
    }
</script>
```

- precision highp, medium, lowp determines the precision → low precision comes with less intensive calculations and higher performance

Shader

- Then, we define the fragment shader:

```
<script id="fragment-shader" type="x-shader/x-fragment">
    #version 300 es
    precision mediump float;
    out vec4 fragColor;

    void main(void) {
      fragColor = vec4(1.0, 0.5, 0.2, 1.0);
    }
</script>
```

Shader

- Create and compile the shader: (Real-Time 3D Graphics with WebGL 2)

```
function getShader(id) {
      const script = document.getElementById(id);
      const shaderString = script.text.trim();

      var shader;
      if (script.type === 'x-shader/x-vertex') {
        shader = gl.createShader(gl.VERTEX_SHADER);
      }
      else if (script.type === 'x-shader/x-fragment') {
        shader = gl.createShader(gl.FRAGMENT_SHADER);
      }
      else {
        return null;
      }
…
```

Shader

- Create and compile the shader: (Real-Time 3D Graphics with WebGL 2)

```
      gl.shaderSource(shader, shaderString);
      gl.compileShader(shader);

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

Initialize the Program

- Now, we initialize the program and call the shaders:

```
function initProgram() {
      const vertexShader = getShader('vertex-shader');
      const fragmentShader = getShader('fragment-shader');

      program = gl.createProgram();
      gl.attachShader(program, vertexShader);
      gl.attachShader(program, fragmentShader);
      gl.linkProgram(program);

      if (!gl.getProgramParameter(program, gl.LINK_STATUS)) {
        console.error('Could not initialize the shaders');
      }
      gl.useProgram(program);
}
```

Draw

- Clear the buffer, bind the VAO, and draw the triangle

```
function draw() {
  gl.viewport(0, 0, gl.canvas.width, gl.canvas.height);
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

  gl.bindVertexArray(VAO);
  gl.drawElements(gl.TRIANGLES, indices.length, gl.UNSIGNED_SHORT, 0);

  gl.bindVertexArray(null);
}
```

- We write `gl.UNSIGNED_SHORT` because we used Uint16Array for the indices

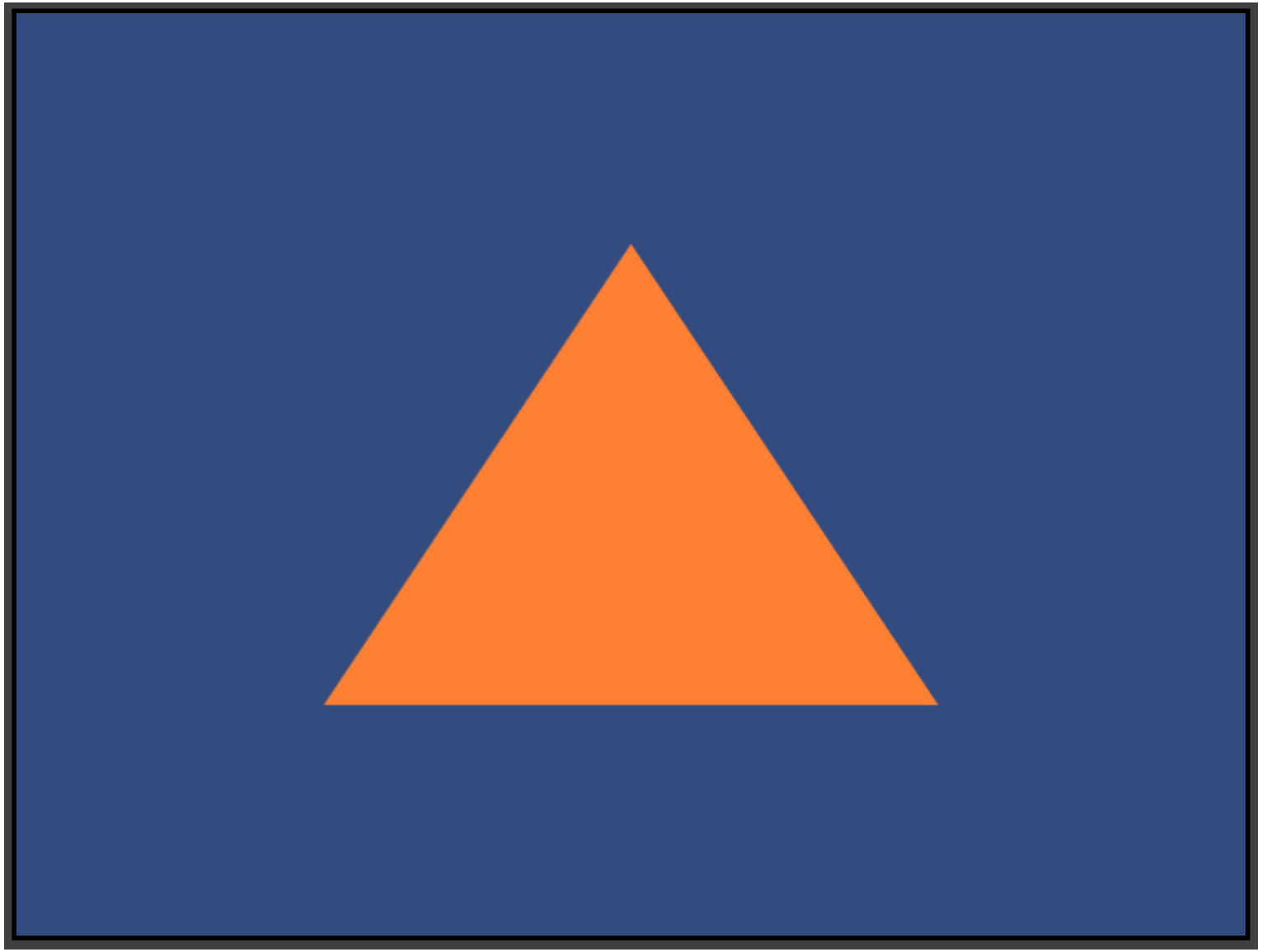
Draw

- Finally, we call everything:

```
function init() {
      const canvas = document.getElementById('glcanvas');
      if (!canvas) {
         console.error('HTML5 Canvas was not found!');
        return;
      }
      //canvas.width = window.innerWidth; // activate for full size
      //canvas.height = window.innerHeight; // activate for full size
      gl = canvas.getContext('webgl2');
      gl.clearColor(0.2, 0.3, 0.5, 1);

      initProgram();
      initBuffers();
      draw();
    }
    window.onload = init;
```

Start

- Our well-known friend



Textures

Introduction

- In the next step, we want to draw a square
- On the square, we would like to have a texture on it

Introduction

- For this, we need to emulate a server in our folder
- We have different options, e.g., XAMPP, EasyPHP, AMPPS, etc.
- The easiest solution, I came across, was to use Python

Introduction

- Open a terminal (I use Anaconda Prompt)
- Then navigate to the directory:

```
cd C:\the\path\of\your\html\file
```

- If you have
  - Python 2: `python -m SimpleHTTPServer 8000`
  - Python 3: `python -m http.server 8000`
- In chrome go to: http://localhost:8000/NameOfYourFile.html

Introduction

- Set new global variables:

```
var gl, program, VAO, VBO, VBO_TC, EBO, indices, texture;
```

Buffer

- Following the lecture on texture, we change the initBuffer function:

```
function initBuffers() {
        vertices = [
           0.5,  0.5, 0.0,
           0.5, -0.5, 0.0,
          -0.5, -0.5, 0.0,
          -0.5,  0.5, 0.0 ];
        texCoords = [
          1.0, 1.0,
          1.0, 0.0,
          0.0, 0.0,
          0.0, 1.0 ];
        indices = [
          0, 1, 3,
          1, 2, 3];
```

Buffer

- Add a VBO for texture coordinates:

```
VAO = gl.createVertexArray();
gl.bindVertexArray(VAO);

VBO = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, VBO);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
gl.vertexAttribPointer(0, 3, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(0);

VBO_TC = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, VBO_TC);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(texCoords), gl.STATIC_DRAW);
gl.vertexAttribPointer(1, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(1);

EBO = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, EBO);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices), gl.STATIC_DRAW);
```

Shader

- The vertex shader uses texture coordinates now:

```
<script id="vertex-shader" type="x-shader/x-vertex">
   #version 300 es
   precision mediump float;

   layout (location=0) in vec3 aVertexPosition;
   layout (location=1) in vec2 aTexCoords;

   out vec2 tc;

   void main(void) {
      tc=aTexCoords;
      gl_Position = vec4(aVertexPosition, 1.0);
    }
  </script>
```

Shader

- The fragment shader uses it:

```
<script id="fragment-shader" type="x-shader/x-fragment">
   #version 300 es
   precision mediump float;

   uniform sampler2D u_texture;

   in vec2 tc;
   out vec4 fragColor;

   void main(void) {
      fragColor =   texture(u_texture,tc);
    }
</script>
```

Texture

- We need to initialize the texture:

```
function initTexture()
{
        texture = gl.createTexture();
        gl.bindTexture(gl.TEXTURE_2D, texture);

        const image = new Image();
        image.onload = function() {
                gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
                gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA,
                gl.UNSIGNED_BYTE, image);
                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 = 'resources/awesomeface2.png';}
```

Draw

- Activate the texture in the draw function:

```
gl.bindVertexArray(VAO);
gl.activeTexture(gl.TEXTURE0);
gl.bindTexture(gl.TEXTURE_2D, texture);

var textureLocation = gl.getUniformLocation(program, "u_texture");
gl.uniform1i(textureLocation, 0);

gl.drawElements(gl.TRIANGLES, indices.length, gl.UNSIGNED_SHORT, 0);

requestAnimationFrame(draw);
```

- RequestAnimationFrame causes a frequently re-call of the draw function (similar to the render loop)

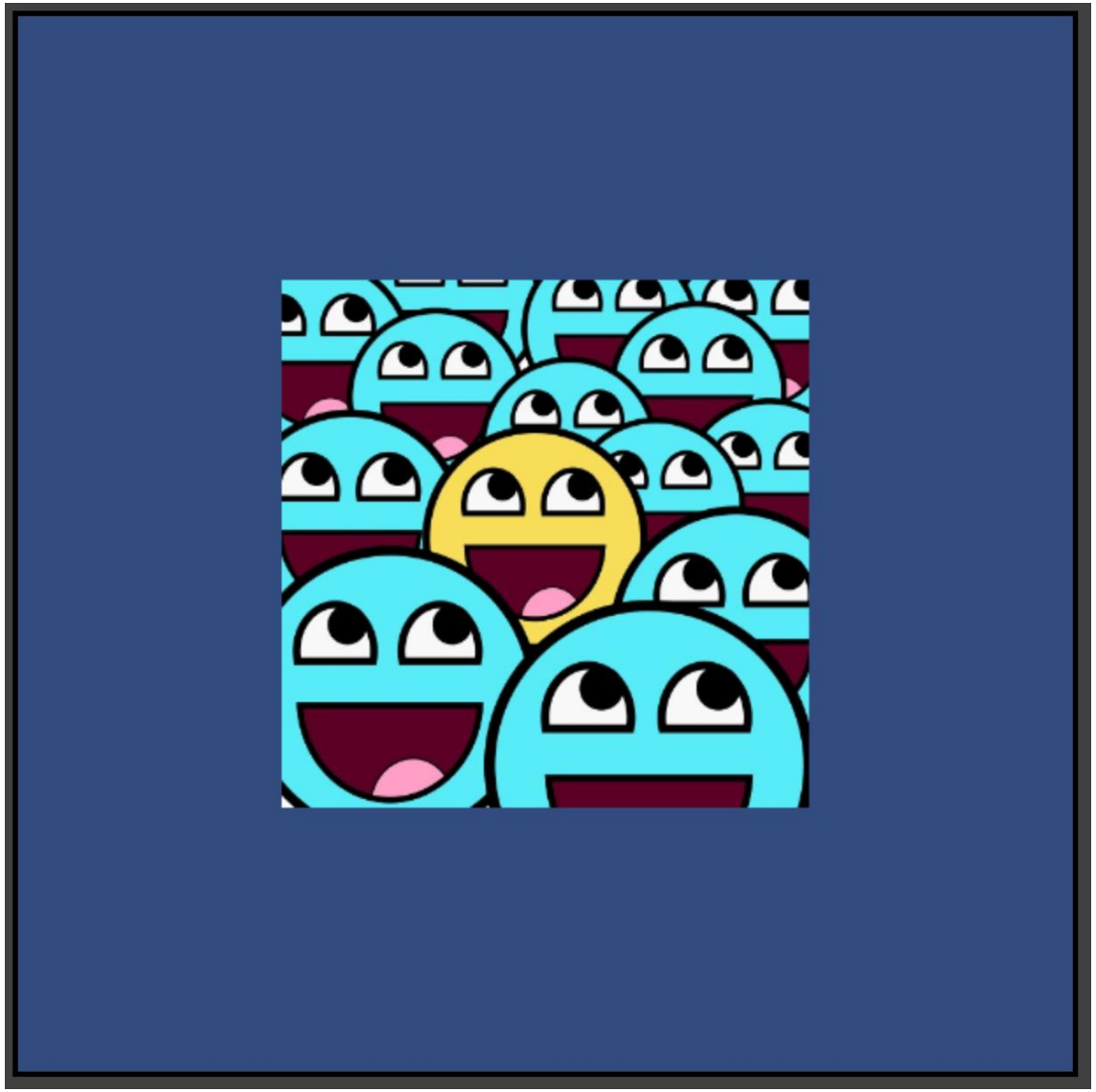
Init

- The init functions calls:

```
initProgram();
initBuffers();
initTexture();
draw();
```

Start

- Looks familiar:



Moving Cube

Introduction

- Now, we combine our knowledge about setting up the camera and the model, view and projection matrix
- Similar to the glm library in C++, we need a library for WebGL
- We use gl-matrix:
  - https://github.com/toji/gl-matrix
  - http://glmatrix.net/

Introduction

- After downloading it, we extract it in:
  - C:\the\path\of\your\html\file\resources\
- Rename the folder to ‘gl-matrix’
- Add the library to the html file:

```
<html>
<head>
<title>WebGL2 Start</title>
<script src="resources/gl-matrix/dist/gl-matrix-min.js"></script>
```

Introduction

- Set new global variables:

```
var gl, program, VAO, VBO, VBO_TC, texture,
      projection, model, view, lastTime, angle=0;
```

Shader

- The vertex shader uses our matrices:

```
<script id="vertex-shader" type="x-shader/x-vertex">
    #version 300 es
    precision mediump float;
    layout (location=0) in vec3 aVertexPosition;
    layout (location=1) in vec2 aTexCoords;

    uniform mat4 model;
    uniform mat4 view;
    uniform mat4 projection;

    out vec2 tc;

    void main(void) {
      tc=aTexCoords;
      gl_Position = projection * view * model * vec4(aVertexPosition, 1.0);
    }
  </script>
```

Buffer

- We create a cube and use the arrays from previous lectures:

```
vertices = [
-0.5, -0.5, -0.5,
 0.5, -0.5, -0.5,
 0.5,  0.5, -0.5,
 0.5,  0.5, -0.5,
-0.5,  0.5, -0.5,
-0.5, -0.5, -0.5,
-0.5, -0.5,  0.5,
 0.5, -0.5,  0.5,
 0.5,  0.5,  0.5,
 0.5,  0.5,  0.5,
-0.5,  0.5,  0.5,
-0.5, -0.5,  0.5,
-0.5,  0.5,  0.5,
-0.5,  0.5, -0.5,
-0.5, -0.5, -0.5,
-0.5, -0.5, -0.5,
-0.5, -0.5,  0.5,
-0.5,  0.5,  0.5,
 0.5,  0.5,  0.5,
 0.5,  0.5, -0.5,
 0.5, -0.5, -0.5,
 0.5, -0.5, -0.5,
 0.5, -0.5,  0.5,
 0.5,  0.5,  0.5,
-0.5, -0.5, -0.5,
 0.5, -0.5, -0.5,
 0.5, -0.5,  0.5,
 0.5, -0.5,  0.5,
-0.5, -0.5,  0.5,
-0.5, -0.5, -0.5,
-0.5,  0.5, -0.5,
 0.5,  0.5, -0.5,
 0.5,  0.5,  0.5,
 0.5,  0.5,  0.5,
-0.5,  0.5,  0.5,
-0.5,  0.5, -0.5];
```

```
texCoords = [
0.0, 0.0,
1.0, 0.0,
1.0, 1.0,
1.0, 1.0,
0.0, 1.0,
0.0, 0.0,
0.0, 0.0,
1.0, 0.0,
1.0, 1.0,
1.0, 1.0,
0.0, 1.0,
0.0, 0.0,
1.0, 0.0,
1.0, 1.0,
0.0, 1.0,
0.0, 1.0,
0.0, 0.0,
1.0, 0.0,
1.0, 0.0,
1.0, 1.0,
0.0, 1.0,
0.0, 1.0,
0.0, 0.0,
1.0, 0.0,
0.0, 1.0,
1.0, 1.0,
1.0, 0.0,
1.0, 0.0,
0.0, 0.0,
0.0, 1.0,
0.0, 1.0,
1.0, 1.0,
1.0, 0.0,
1.0, 0.0,
0.0, 0.0,
0.0, 1.0];
```

Camera

- We create our camera set up in a function:

```
function initCamera()
{
        projection = glMatrix.mat4.create();
        glMatrix.mat4.perspective(projection, 45,
                  gl.canvas.width/gl.canvas.height, 0.01, 100);

        model = glMatrix.mat4.create();
        glMatrix.mat4.identity(model);
        glMatrix.mat4.translate(model, model, [0, 0, 0]);
        glMatrix.mat4.rotate(model, model, 30 * Math.PI / 180, [0, 1, 0]);

        view = glMatrix.mat4.create();
        glMatrix.mat4.identity(view);
}
```

Animation

- Create a changing angle:

```
function animate()
{
      var timeNow = new Date().getTime();
      if (lastTime) {
        const elapsed = timeNow - lastTime;
        angle += elapsed * 0.001;
      }
      lastTime = timeNow;
}
```

Render

- A render function similar to the render loop

```
function render()
{
	requestAnimationFrame(render);
	animate();
	draw();
}
```

Draw

- The draw function has to send the matrices to the vertex shader:

```
var radius = 2.0;
var camX   = Math.sin(angle) * radius;
var camY   = Math.cos(angle) * Math.sin(angle) * radius;
var camZ   = Math.cos(angle) * radius;

var pos = glMatrix.vec3.create();
pos.set([camX,camY,camZ]);

var center = glMatrix.vec3.create();
center.set([0,0,0]);

var up = glMatrix.vec3.create();
up.set([0,1,0]);
```

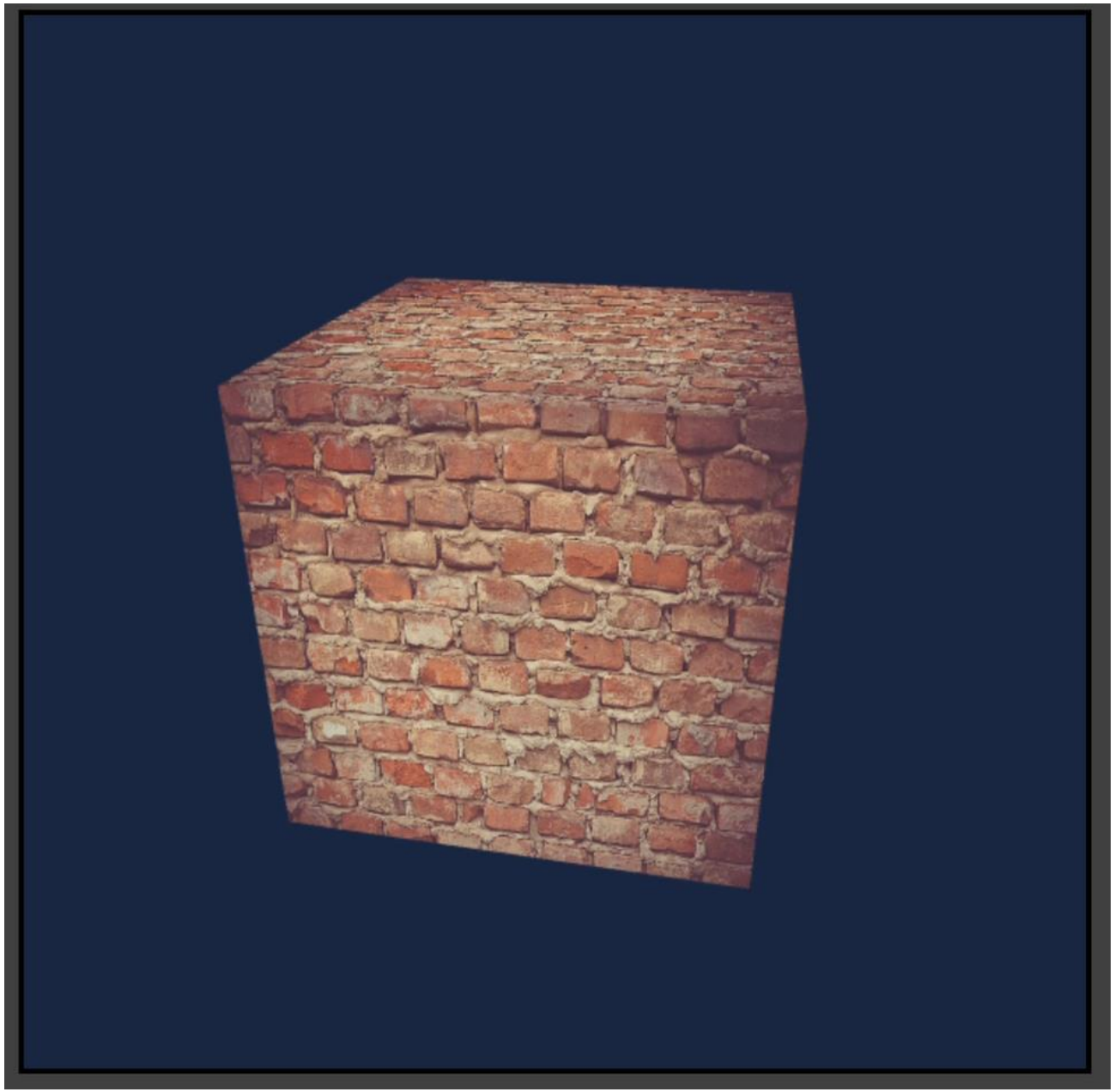
Draw

- The draw function has to send the matrices to the vertex shader:

```
glMatrix.mat4.lookAt(view, pos, center, up);

var viewLocation = gl.getUniformLocation(program, "view");
gl.uniformMatrix4fv(viewLocation, false, view);

var modelLocation = gl.getUniformLocation(program, "model");
gl.uniformMatrix4fv(modelLocation, false, model);

var projLocation = gl.getUniformLocation(program, "projection");
gl.uniformMatrix4fv(projLocation, false, projection);

gl.drawArrays(gl.TRIANGLES, 0, 36);
```

Start

- A rotating cube:



Navigation

Introduction

- Time to get independent
- Now, we want to look around

Introduction

- We need a lot of new global variables (see last lectures):

```
var lastX = 300;
var lastY = 300;
var firstMouse = true;
var yaw   = -90.0;
var pitch =  0.0;

var cameraFront=glMatrix.vec3.create();
cameraFront.set([0,0,-1]);
var cameraPos = glMatrix.vec3.create();
cameraPos.set([0,0,3]);
var cameraUp = glMatrix.vec3.create();
cameraUp.set([0,1,0]);

var click=false;
var deltaTime=0;
```

Draw

- During the navigation, the view matrix changes
- We use the lookAt function and set the new center

```
var center = glMatrix.vec3.create();
glMatrix.vec3.add(center,cameraPos,cameraFront);

glMatrix.mat4.lookAt(view, cameraPos, center, cameraUp);

var viewLocation = gl.getUniformLocation(program, "view");
gl.uniformMatrix4fv(viewLocation, false, view);
```

Animate

- Then, we need deltaTime to adjust the sensitivity of the movement

```
function animate() {
    var timeNow = new Date().getTime();
    if (lastTime) {
        deltaTime = timeNow - lastTime;
    }
    lastTime = timeNow;
}
```

Init

- To query if keys are pressed (or the mouse moves), we add an event listener (similar to the callback functions):

```
function init() {
  const canvas = document.getElementById('glcanvas');

  document.addEventListener('keydown', keyDownHandler, false);
  document.addEventListener('mousedown', mouseDown, false);
  document.addEventListener('mouseup', mouseUp, false);
  document.addEventListener('mousemove', mouseMove, false);
```

Mouse Down/Up

- These functions are called, when the mouse is pressed/released:

```
function mouseDown(event) {
        click=true;
}

function mouseUp(event) {
        click=false;
        firstMouse=true;
}
```

- We only want to move, if the mouse is pressed

Key Pressed

- We want to move, if the arrow keys are pressed:

```
function keyDownHandler(event) {
   var cameraSpeed=0.01*deltaTime;
   if(event.keyCode == 39) { // rightPressed = true;
       var crossVec=glMatrix.vec3.create();
       glMatrix.vec3.cross(crossVec,cameraFront,cameraUp);
       glMatrix.vec3.scaleAndAdd(cameraPos,cameraPos,crossVec,cameraSpeed);
    }
    else if(event.keyCode == 37) { // leftPressed = true;
       var crossVec=glMatrix.vec3.create();
       glMatrix.vec3.cross(crossVec,cameraFront,cameraUp);
       glMatrix.vec3.scaleAndAdd(cameraPos,cameraPos,crossVec,-cameraSpeed);
    }
    if(event.keyCode == 40) { // downPressed = true;
       glMatrix.vec3.scaleAndAdd(cameraPos,cameraPos,cameraFront,-cameraSpeed);
    }
    else if(event.keyCode == 38) { // upPressed = true;
       glMatrix.vec3.scaleAndAdd(cameraPos,cameraPos,cameraFront,cameraSpeed);
    }}
```

Key Pressed

- Look around if the mouse is clicked

```
function mouseMove(event) {
if(click)
{
        if (firstMouse)
                {
                        lastX = event.pageX;
                        lastY = event.pageY;
                        firstMouse = false;
                }

                var xoffset = event.pageX - lastX;
                var yoffset = lastY - event.pageY;

                lastX = event.pageX;
                lastY = event.pageY;
```

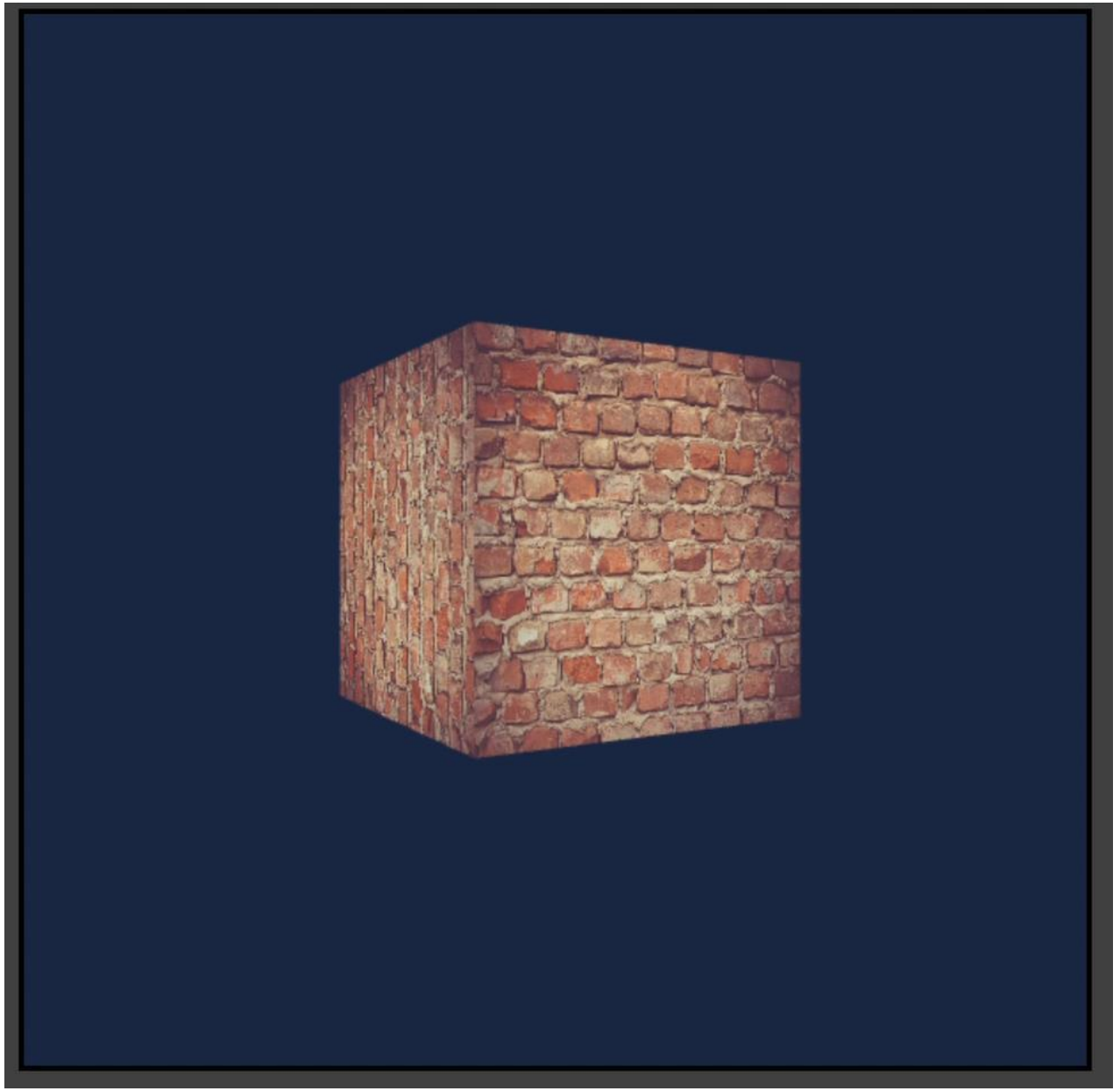
Key Pressed

- We want to move, if the arrow keys are pressed:

```
var sensitivity = 0.1;
xoffset *= sensitivity;
yoffset *= sensitivity;

yaw += xoffset;
pitch += yoffset;

if (pitch > 89.0)
        pitch = 89.0;
if (pitch < -89.0)
        pitch = -89.0;

var front=glMatrix.vec3.create();
front[0] = Math.cos(yaw * Math.PI / 180) * Math.cos(pitch * Math.PI / 180);
front[1] = Math.sin(pitch * Math.PI / 180);
front[2] = Math.sin(yaw * Math.PI / 180) * Math.cos(pitch * Math.PI / 180);
glMatrix.vec3.normalize(cameraFront,front);
}}
```

Start

- Finally, we can move



Load an OBJ Model

Introduction

- A cube may be boring after a while
- Now, we want to load arbitrary models
- For simplicity and educational purposes, we write our own simple OBJ loader

Introduction

- We need (of course) new global variables:

```
var vertices;
vertices=[];
var faces;
faces=[];
var normals;
normals=[];
```

Async – Await

- It may happen that our code is finished even before we completely loaded the OBJ file
- This means, the vertices, faces, and normal array are empty, and nothing is drawn
- To overcome this, we can use async and await to ensure to continue after the model is loaded

Init

- Write async before the function and add await before we call our new loadOBJ() function:

```
async function init() {
        …
          initProgram();
          await loadOBJ('resources/dog.obj');
          initBuffers();
          initCamera();
          render();
    }
```

Load OBJ

- With fetch, we get the content of the file, get the text, and iterate over every line

```
async function loadOBJ(location) {

var numberOfTriangles=0;
const response = await fetch(location);
const text = await response.text();
var lines = text.split('\n');

for (var lineNo = 0; lineNo < lines.length; ++lineNo) {
const line = lines[lineNo].trim();
      if (line === '' || line.startsWith('#'))
      {
            continue;
      }
```

Load OBJ

- If we identify a vertex (face), we split the line into the upcoming numbers (get rid of the v (f)) and push them to the arrays:

```
        if(line[0]=='v' || line[0]==' ')
        {
                var vertexPos=line.split(/\s+/).slice(1);
                vertices.push(vertexPos[0],vertexPos[1],vertexPos[2]);
                normals.push(0,0,0);
        }
        if(line[0]=='f' || line[0]==' ')
        {
                var faceInd=line.split(/\s+/).slice(1);
                faces.push(faceInd[0]-1,faceInd[1]-1,faceInd[2]-1);
                numberOfTriangles++;
        }

}
```

Load OBJ

- Now, we calculate the normals for every vertex:

```
for (var i = 0; i < numberOfTriangles; i++)
{
        var ind1 = faces[3 * i];
        var ind2 = faces[3 * i + 1];
        var ind3 = faces[3 * i + 2];

        var coord1 = [ vertices[3 * ind1 + 0], vertices[3 * ind1 + 1],
              vertices[3 * ind1 + 2] ];
        var coord2 = [ vertices[3 * ind2 + 0], vertices[3 * ind2 + 1],
              vertices[3 * ind2 + 2] ];
        var coord3 = [ vertices[3 * ind3 + 0], vertices[3 * ind3 + 1],
              vertices[3 * ind3 + 2] ];
```

Load OBJ

- Now, we calculate the normals for every vertex:

```
        var norm=calculateNormal(coord1, coord2, coord3);

        normals[3 * ind1 + 0] += norm[0] * norm[3];
        normals[3 * ind1 + 1] += norm[1] * norm[3];
        normals[3 * ind1 + 2] += norm[2] * norm[3];

        normals[3 * ind2 + 0] += norm[0] * norm[3];
        normals[3 * ind2 + 1] += norm[1] * norm[3];
        normals[3 * ind2 + 2] += norm[2] * norm[3];

        normals[3 * ind3 + 0] += norm[0] * norm[3];
        normals[3 * ind3 + 1] += norm[1] * norm[3];
        normals[3 * ind3 + 2] += norm[2] * norm[3];
}
```

Calculate the Normal

- Define a function that calculates the cross product (without using gl-matrix)

```
function calculateNormal(coord1, coord2, coord3)
{
        var va=[0,0,0], vb=[0,0,0], vr=[0,0,0], val;
        va[0] = coord1[0] - coord2[0];
        va[1] = coord1[1] - coord2[1];
        va[2] = coord1[2] - coord2[2];
        vb[0] = coord1[0] - coord3[0];
        vb[1] = coord1[1] - coord3[1];
        vb[2] = coord1[2] - coord3[2];
        /* cross product */
        vr[0] = va[1] * vb[2] - vb[1] * va[2];
        vr[1] = vb[0] * va[2] - va[0] * vb[2];
        vr[2] = va[0] * vb[1] - vb[0] * va[1];
```

Calculate the Normal

- Define a function that calculates the cross product (without using gl-matrix)

```
        /* normalization factor */
        val = Math.sqrt(vr[0] * vr[0] + vr[1] * vr[1] + vr[2] * vr[2]);

        var normals=[0,0,0,0];
        normals[0] = vr[0] / val;
        normals[1] = vr[1] / val;
        normals[2] = vr[2] / val;
        normals[3] = val / 2;

        return normals;
}
```

Buffer

- Do not forget to bind the buffer with the vertices, normal, and faces arrays:

```
function initBuffers() {

        VAO = gl.createVertexArray();
        gl.bindVertexArray(VAO);

        VBO = gl.createBuffer();
        gl.bindBuffer(gl.ARRAY_BUFFER, VBO);
        gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices),
                gl.STATIC_DRAW);
        gl.vertexAttribPointer(0, 3, gl.FLOAT, false, 0, 0);
        gl.enableVertexAttribArray(0);
```

Buffer

- Do not forget to bind the buffer with the vertices, normal, and faces arrays:

```
        VBO_NORMALS = gl.createBuffer();
        gl.bindBuffer(gl.ARRAY_BUFFER, VBO_NORMALS);
        gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(normals),
                gl.STATIC_DRAW);
        gl.vertexAttribPointer(1, 3, gl.FLOAT, false, 0, 0);
        gl.enableVertexAttribArray(1);

        EBO = gl.createBuffer();
        gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, EBO);
        gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(faces),
                gl.STATIC_DRAW);
        gl.bindVertexArray(null);
        gl.bindBuffer(gl.ARRAY_BUFFER, null);
        gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, null);     }
```

Draw

- The correct draw call:

```
function draw() {
        …
        gl.drawElements(gl.TRIANGLES, faces.length, gl.UNSIGNED_SHORT, 0);
        …
}
```

Shader

- Last but not least the shaders:

```
<script id="vertex-shader" type="x-shader/x-vertex">
    #version 300 es
    precision mediump float;

    // Supplied vertex position attribute
    layout (location=0) in vec3 aVertexPosition;
    layout (location=1) in vec3 aNormals;

    uniform mat4 model;
    uniform mat4 view;
    uniform mat4 projection;
    out vec3 normals;

    void main(void) {
      normals=mat3(model)*normalize(aNormals);
      gl_Position = projection * view * model * vec4(aVertexPosition, 1.0);
    } </script>
```

Shader

- Last but not least the shaders:

```
<script id="fragment-shader" type="x-shader/x-fragment">
    #version 300 es
    precision mediump float;

    in vec3 normals;
    out vec4 fragColor;

    void main(void) {
      fragColor =   vec4(vec3(dot(normalize(normals),vec3(0,0,1))),1);
    }
  </script>
```

Start

- We did it!

Remark

- Before we end the lectures about computer graphics, we must do one last example that cannot be missed in a computer graphics class

Mandelbrot Set

Introduction

- The Mandelbrot set is the set of complex numbers c, for which

$$f_c(z) = z^2 + c$$

does not diverge, when we iterate with the sequence

$$f_c(0), f(f(0)), f(f(f(c))), \ldots$$

Introduction

- Let $z = a + bi$ be a complex number, the multiplication with itself gives:

$$\begin{aligned} z \cdot z &= (a + bi)(a + bi) \\ &= a^2 + abi + abi + b^2 i^2 \\ &= (a^2 - b^2) + (2ab)i \end{aligned}$$

Introduction

- Let $z = 0.5 + 0.5i$, is this complex number in the Mandelbrot set?

Introduction

$$f_c(z) = z^2 + c$$
$$z \cdot z = (a^2 - b^2) + (2ab)i$$

- Let c $= \frac{1}{2} + \frac{1}{2}i$, is this point in the Mandelbrot set?

$$f_c(0) = 0^2 + \frac{1}{2} + \frac{1}{2}i = \frac{1}{2} + \frac{1}{2}i$$

$$f_c(\frac{1}{2} + \frac{1}{2}i) = (\frac{1}{2^2} - \frac{1}{2^2}) + (2 \cdot \frac{1}{2} \cdot \frac{1}{2})i + \frac{1}{2} + \frac{1}{2}i = \frac{1}{2} + 1i$$

$$f_c(\frac{1}{2} + 1i) = (\frac{1}{2^2} - 1^2) + (2 \cdot \frac{1}{2} \cdot 1)i + \frac{1}{2} + \frac{1}{2}i = -\frac{1}{4} + \frac{3}{2}i$$

$$f_c(-\frac{1}{4} + \frac{3}{2}i) = (\frac{1}{4^2} - \frac{3^2}{2^2}) + (2 \cdot \frac{-1}{4} \cdot \frac{3}{2})i + \frac{1}{2} + \frac{1}{2}i = -\frac{27}{16} - \frac{1}{4}i$$

Introduction

$$f_c(z) = z^2 + c$$
$$z \cdot z = (a^2 - b^2) + (2ab)i$$

- Let c $= \frac{1}{2} + \frac{1}{2}i$, is this point in the Mandelbrot set?

$$f_c^4(0) = 3.2852 + 1.3438i$$
$$f_c^5(0) = 9.4866 + 9.3289i$$
$$f_c^6(0) = 3.4678 + 177.5i$$
$$\vdots$$

- This point leads to a divergent sequence → it is not in the Mandelbrot set

Introduction

- We have an equivalent formulation:
- A point c belongs to the Mandelbrot set if and only if

$$|f_c^n(0)| \leq 2 \text{ for all } n$$

- This means, during the iteration, we can check if the complex value exceeds 2 and if so, we can break the iteration

$|f_c^n(0)| \leq 2 \text{ for all } n$

Introduction

- Let‘s check:

$$f_c^4(0) = 3.2852 + 1.3438i$$

$$|f_c^4(0)| \approx \sqrt{3.2852^2 + 1.3438^2} \approx 3.5494$$

- Length exceeds 2 so it is not in the Mandelbrot set

Set Up

- To visualize this set, we need to create a plane with texture coordinates first:

```
function initBuffers() {

    vertices = [
         1.0,  1.0, 0.0,  // top right
         1.0, -1.0, 0.0,  // bottom right
        -1.0, -1.0, 0.0,  // bottom left
        -1.0,  1.0, 0.0   // top left
    ];

       indices = [  // note that we start from 0!
         0, 1, 3,  // first Triangle
         1, 2, 3   // second Triangle
    ];
…}
```

Shader

- The vertex shader is not interesting:

```
<script id="vertex-shader" type="x-shader/x-vertex">
    #version 300 es
    precision mediump float;

    layout (location=0) in vec3 aVertexPosition;

    void main(void) {
       gl_Position = vec4(aVertexPosition, 1.0);
    }
 </script>
```

Shader

- In the fragment shader, we use the well-known colormap from last lecture

```
float colormap_red(float x) {
if (x < 0.09752005946586478) {
return 5.63203907203907E+02 * x + 1.57952380952381E+02;
} else if (x < 0.2005235116443438) {
return 3.02650769230760E+02 * x + 1.83361538461540E+02;
} else if (x < 0.2974133397506856) {
return 9.21045429665647E+01 * x + 2.25581007115501E+02;
} else if (x < 0.5003919130598823) {
return 9.84288115246108E+00 * x + 2.50046722689075E+02;
} else if (x < 0.5989021956920624) {
return -2.48619704433547E+02 * x + 3.79379310344861E+02;
} else if (x < 0.902860552072525) {
return ((2.76764884219295E+03 * x - 6.08393126459837E+03) * x + 3.80008072407485E+03) * x - 4.57725185424742E+02;
} else {
return 4.27603478260530E+02 * x - 3.35293188405479E+02;
}
}

float colormap_green(float x) {
if (x < 0.09785836420571035) {
return 6.23754529914529E+02 * x + 7.26495726495790E-01;
} else if (x < 0.2034012006283468) {
return 4.60453201970444E+02 * x + 1.67068965517242E+01;
} else if (x < 0.302409765476316) {
return 6.61789401709441E+02 * x - 2.42451282051364E+01;
} else if (x < 0.4005965758690823) {
return 4.82379130434784E+02 * x + 3.00102898550747E+01;
} else if (x < 0.4981907026473237) {
return 3.24710622710631E+02 * x + 9.31717541717582E+01;
} else if (x < 0.6064345916502067) {
return -9.64699507389807E+01 * x + 3.03000000000023E+02;
} else if (x < 0.7987472620841592) {
return -2.54022986425337E+02 * x + 3.98545610859729E+02;
} else {
return -5.71281628959223E+02 * x + 6.51955082956207E+02;
}
}

float colormap_blue(float x) {
if (x < 0.0997359608740309) {
return 1.26522393162393E+02 * x + 6.65042735042735E+01;
} else if (x < 0.1983790695667267) {
return -1.22037851037851E+02 * x + 9.12946682946686E+01;
} else if (x < 0.4997643530368805) {
return (5.39336225400169E+02 * x + 3.55461986381562E+01) * x + 3.88081126069087E+01;
} else if (x < 0.6025972254407099) {
return -3.79294261294313E+02 * x + 3.80837606837633E+02;
} else if (x < 0.6990141388105746) {
return 1.15990231990252E+02 * x + 8.23805453805459E+01;
} else if (x < 0.8032653181119567) {
return 1.68464957265204E+01 * x + 1.51683418803401E+02;
} else if (x < 0.9035796343050095) {
return 2.40199023199020E+02 * x - 2.77279202279061E+01;
} else {
return -2.78813846153774E+02 * x + 4.41241538461485E+02;
}
}

vec4 colormap(float x) {
float r = clamp(colormap_red(x) / 255.0, 0.0, 1.0);
float g = clamp(colormap_green(x) / 255.0, 0.0, 1.0);
float b = clamp(colormap_blue(x) / 255.0, 0.0, 1.0);
return vec4(r, g, b, 1.0);
}
```

Shader

- Set the precision high and then copy the colormap functions:

```
<script id="fragment-shader" type="x-shader/x-fragment">
    #version 300 es
    precision highp float;

    out vec4 fragColor;
    …
```

Shader

- Set the precision high and then copy the colormap functions:

```
vec2 center=vec2(0.3333,0.0);
float scale=3.0;

void main()
{
        int iter=50;
        vec2 z, c;

        c.x = 4.0/3.0 * (gl_FragCoord.x/800.0 - 0.5) * scale - center.x;
        //[-1/3;1/3]
        c.y = (gl_FragCoord.y/600.0 - 0.5) * scale - center.y;
        //[-0.5;0.5]
        int i;
        z = c;
```

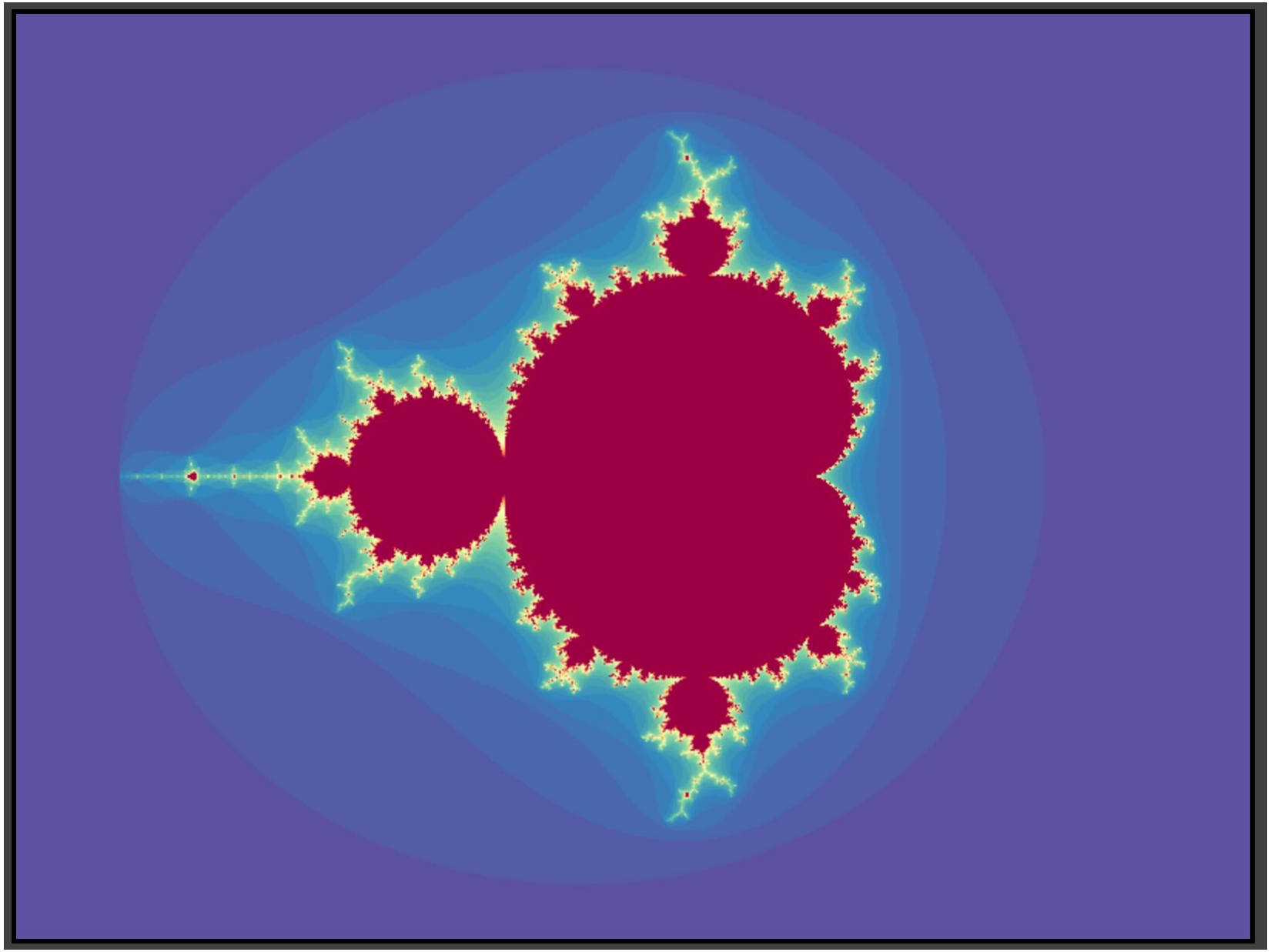
Shader

- Iterate over the function and check if the lengths exceeds 2:

```
for(i=0; i<iter; i++) {
        float x = (z.x * z.x - z.y * z.y) + c.x;
        float y = (2.0 * z.x * z.y) + c.y;

         z.x = x;
         z.y = y;

        if(length(z) > 2.0)
                 break;
}
fragColor = colormap( 1.0-float(i) / float(iter));
}
 </script>
```

Start

- Beautiful!



Improvements

- Let’s get a bit fancier

Draw

- Define a global variable timeNow and define it before the calling the draw function:

```
timeNow = new Date().getTime();
draw();
```

- In the draw function, we use a timer as a uniform (starts after 1000ms) and call the draw function again:

```
timeCurrent = Math.max(new Date().getTime()-timeNow-1000.0,0.0);
var timeLoc = gl.getUniformLocation(program, "time");
gl.uniform1f(timeLoc, timeCurrent);
…
requestAnimationFrame(draw);
```

Shader

- Change the fragment shader:

```
<script id="fragment-shader" type="x-shader/x-fragment">
    #version 300 es
    precision highp float;

    uniform float time;
    …
    vec2 center=vec2(0.7701,0.1);
    float scale=3.0;
```

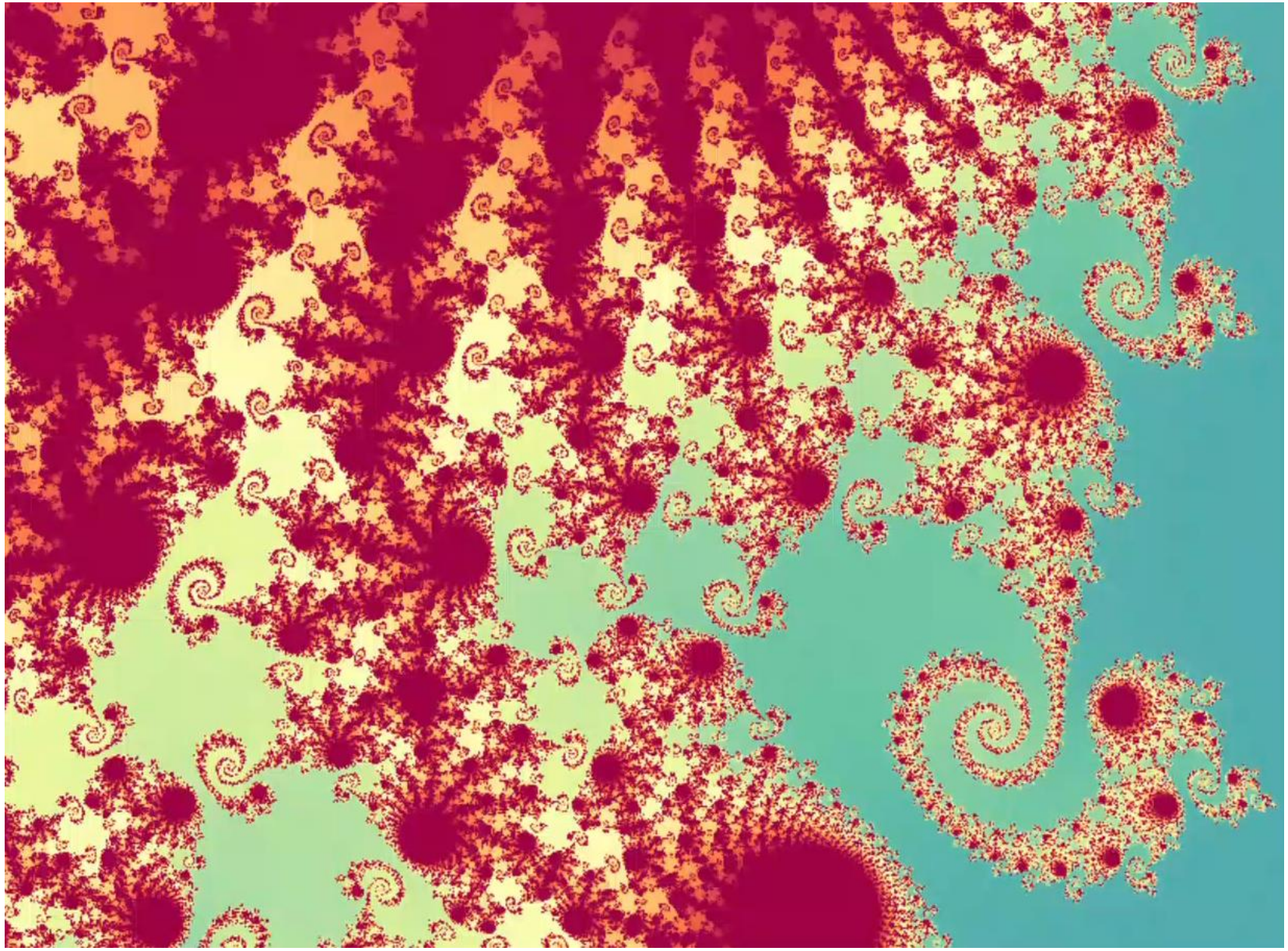
Shader

- Change the fragment shader:

```
void main(){
int iter=1;

if(int(time/40.0)<1000)
iter=int(time/40.0);

vec2 z, c;
float animSpeed=((time*time*time/100000.0))*0.0001;

c.x = 4.0/3.0 * (gl_FragCoord.x/800.0 - 0.5) * scale/animSpeed - center.x;
c.y = (gl_FragCoord.y/600.0 - 0.5) * scale/animSpeed - center.y;
…
```

Start

- Beautiful!



Questions???