JOHANNES KEPLER UNIVERSITY LINZ

Institute of Computer Graphics

Computer Graphics

Lab 1: Introduction to WebGL

CG Lab Team

David Schedl, MSc

Prof. Dr. Marc Streit

Katharina Prinz

Pieter-Jan Hoedt

What you can expect...

Schedule

JOHANNES KEPLER UNIVERSITY LINZ

Institute of Computer Graphics

Lab 1	Introduction to WebGL	10.3.2017 / 14.3/2017	-
Lab 2	Transformations and Projections	21.3.2017 / 24.3.2017	Lecture: 14.3.2017
Lab 3	Scene Graphs	28.3.2017 / 31.3.2017	Lecture: 21.3.2017
Lab 4	Illumination and Shading	4.4.2017 / 7.4.2017	Lecture: 28.3.2017
Lab 5	Texturing	25.4.2017 / 28.4.2017	Lecture: 4.4.2017
Lab 6	Advanced Texture Mapping	2.5.2017 / 5.5.2017	Lecture: 26.4.2017
Lab 7a	CUDA	9.5.2017 / 12.5.2017	
Lab 7b	VTK	12.5.2017	

Today's Lab

JOHANNES KEPLER UNIVERSITY LINZ

Institute of Computer Graphics

- Administrative issues
- What is OpenGL?
  - Programmable Pipeline
  - Definitions
- Development environment
  - Our lab package
  - JavaScript, HTML5, CSS
  - Atom + Live-Server
  - Chrome|Firefox Developer Tools

Baby's first WebGL program aka *Let's draw a colored 2D quad*

Administrative Issues

JOHANNES KEPLER UNIVERSITY LINZ

Institute of Computer Graphics

Lectures: 2h weekly, 3 ECTS

Labs: 2h ~ bi-weekly, 1.5 ECTS, 6 groups

Contact

- General point of contact: cg-lab@jku.at
- Also use the CG discussion forum in KUSSS

Tutorial slides and lab material

- www.cg.jku.at/teaching/computergraphics/lab

More information on other courses & invited talks

- www.cg.jku.at

Interested in BSc./MSc. thesis or practical course?

- www.cg.jku.at/teaching/theses

Administrative Issues

JOHANNES KEPLER UNIVERSITY LINZ

Institute of Computer Graphics

- We only have 12 PCs in the pool
- Work in pairs on 1 PC or bring your own laptop
  - First labs: WebGL compatible browser
  - Last CUDA lab: shader model 4, Nvidia only
- Group assignment
  - If you have a conflict with an obligatory course (not electoral course, private reasons, etc.) then contact us: cg-lab@jku.at
  - You can use the CG forum in KUSSS to find someone who wants to swap groups
  - If you found someone, send an email to cg-lab@jku.at
  - In all other cases switching groups is not possible

PC Pool (S3 0063)

JOHANNES KEPLER UNIVERSITY LINZ

Institute of Computer Graphics

Usage

- Use the PC pool if you don‘t have an appropriate computer
- Access 24/7
- Do not use the PCs and pool for anything else than your CG exercises
- The computer will reset the C drive during restart, so backup your data!

Registration to get access using Kepler Card

- http://www.cg.jku.at/ → [Internal] → [Requesting Access to PC-Pool S3 0063]

Entering the PC Pool

- By holding Kepler card in front of lock
- Lock the door every time you leave the pool last!
- Any problems with the room or equipment can be traced back

Lab Overview

JOHANNES KEPLER UNIVERSITY LINZ

Institute of Computer Graphics

7 lab sessions with different topics related to the lectures

What you should learn in the labs

- How the theory learned in the lectures can be applied
- Basic knowledge about the OpenGL/WebGL computer graphics API
- Understand what the API commands do and how they are used

What we expect you to learn by yourself

- Apply the learned skills to new tasks!
- Experiment and get practice! → You will need it for the lab project
- JavaScript basics

Prerequisites

JOHANNES KEPLER UNIVERSITY LINZ

Institute of Computer Graphics

Labs require basic knowledge of JavaScript programming

We do not teach JavaScript programming!

Resources to online books, software and libraries

http://140.78.90.140/projects/index.php/Coding_Resources

Or go to www.jku.at/cg, then go to ‚internal' (menu left) and select ‚coding resources'

Modus Operandi in Lab Sessions

JOHANNES KEPLER UNIVERSITY LINZ

Institute of Computer Graphics

Multiple alternating short blocks of theory and hands-on parts

- We will briefly explain the needed commands and theory …
- … then you have to do practical tasks on your machine …
- … finally, we discuss the solution.

Slides

- The slides you get before the lab sessions sometimes miss solutions
- You will get the complete slides by the end of each lab week

Practice at home!

- Take a look at tutorials and hand-books
- Practice between the lab sessions!
- We will give you hints what you should practice

Grading

JOHANNES KEPLER UNIVERSITY LINZ

Institute of Computer Graphics

No lab exam at the end

Lab project grade = course grade

Lab project in groups of two students

- Independent of groups in tutorials

Interviews at the end

- Theoretical concept questions + code walkthrough

Details on the lab project will be announced soon

Let’s start...

Do you know...

JOHANNES KEPLER UNIVERSITY LINZ

Institute of Computer Graphics

Java

CSS

C++

Bower

OpenGL®

WebGL™

OpenGL|ES™

OpenGL …

… is an abbreviation for Open Graphics Library
… is an interface to graphics hardware (GPU)
… is a compact and platform independent open standard API
… arose from SGI's efforts to evolve IRIS GL in 1992
… was developed by the ARB (Architecture Review Board)
… is now a part of the Khronos working group (since 2006)
… is platform independant
… is a procedural graphics API, i.e., state machine
… provides no window management, user interaction or file I/O
… special versions for different platforms: OpenGL ES, WebGL

Rendering Pipeline

https://open.gl/drawing (adapted)

Programmable Pipeline
(as for OpenGL ES 2.0 / WebGL)

Programmable Pipeline

Programmable Pipeline

Buffer Objects

- array stored on the GPU
- contain vertex data
  - position
  - color
  - normal
  - textureCoordinates
  - …
- bound to attributes in the vertex shader

Programmable Pipeline

Shader

- small program executed on the GPU
- written in GLSL (C-like Syntax)
- different types:
  - Vertex Shader
    compute the vertex position
  - Fragment Shader
    compute the fragment color
  - more types in plain OpenGL,
    e.g. Geometry Shader, Tesselation Shader
- vertex + fragment → linked ⇒ program

Programmable Pipeline

Shader Parameters

- attributes
  streams for the vertex shader stored in buffer,
  e.g., vertex position
- uniforms
  parameter from the program to the shader,
  e.g., light position, texture reference
- varyings
  out/input between shader stages with interpolation
  e.g., vertex color → fragment color

Programmable Pipeline

Vertex Shader

- executed per vertex
- purpose:
  compute the vertex position in ClipSpace (-1,+1) coordinates → stored in gl_Position
- inputs
  - attributes, e.g., vertex position in world coordinates → change per vertex
  - uniforms ~ program parameters
- outputs
  - varyings, e.g., vertex color
  - gl_Position

Programmable Pipeline

Primitive Assembly

takes vertices and build primitives out of it for rasterization

Programmable Pipeline

Rasterization

converts abstract primitives to set of fragments

Programmable Pipeline

Fragment Shader

- executed per fragment
- purpose:
  compute the fragment RGBA color
  → stored in gl_FragColor
- inputs
  - varyings e.g., interpolated varyings from vertex shader, e.g. vertex/fragment color
  - uniforms ~ program parameters
- outputs
  - gl_FragColor

Programmable Pipeline

Per-Fragment Operation

- additional tests/operations per fragment
- including
  - depth test
    the closer fragment should be drawn
  - blending
    in case of semi transparent fragments
  - stencil test
    check whether the fragment is masked in the stencil mask
  - ...

Programmable Pipeline

Framebuffer

- output canvas i.e., the screen
- 2D array of RGB pixels
- could also just be a memory position on the GPU (Framebuffer Object) that can be used in a different run as texture

Programmable Pipeline

Summary

- vertex … point in 2D/3D space
- fragment … pixel + additional properties
- shader … tiny program on the GPU
- shader program = vertex + fragment shader
- buffer … array on GPU
- attribute … accessing the current buffer element in shader
- uniform … parameter from program to shader
- varying … parameter between shader
- gl_Position, gl_FragColor … magic variables
- rasterization: 3 vertices → N fragments

Dev Environment: Atom

JOHANNES KEPLER UNIVERSITY LINZ

Institute of Computer Graphics

A good editor / IDE will simplify your life!

We use Atom (https://atom.io) but you can use anyone you like

ATOM

Important Note!

You need to access your WebGL website through a local web server.

Reason? Security! Chrome doesn't allow to load local files asynchronously

→ Big list of one-liners to run a local web server

→ Atom: Packages → Live Server (also launch via CTRL-SHIFT-3)

Fork me on GitHub

Dev Environment: Lab Package

JOHANNES KEPLER UNIVERSITY LINZ

Institute of Computer Graphics

Hosted on GitHub: https://github.com/jku-icg/cg_lab_2017

The repository will be updated during the lab with the new projects.

To get started (now):

1. Download the zip
2. Extract the folder
3. Open Atom editor
4. Use “Open Folder” in Atom
5. Start server on any port (Packages -> Live Server)

Dev Environment: HTML5, JS, CSS

JOHANNES KEPLER UNIVERSITY LINZ

Institute of Computer Graphics

WebGL → OpenGL in the Web-browser based on OpenGL ES 2.0

Basic project structure:

- cg_lab_2017-master
  - 00_empty
  - 01_intro_handout
    - shader
      - simple.fs.glsl
      - simple.vs.glsl
    - index.html
    - main.js
    - snippets.js
    - style.css
    - libs
      - framework.js
      - gl-matrix.js
  - index.html

index.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Empty</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <!-- include helper library for matrix computation -->
    <script src="../libs/gl-matrix.js"></script>
    <!-- include our framework with utilities -->
    <script src="../libs/framework.js"></script>
    <!-- include the main script -->
    <script src="main.js"></script>
</body>
</html>
```

Dev Environment: Developer Tools

JOHANNES KEPLER UNIVERSITY LINZ

Institute of Computer Graphics

Know the Web Developer Tools of your favorite browser

Chrome, Firefox, Edge, Safara, … → usually F12

Great for debugging JavaScript code, manipulating CSS & DOM, …

First Application

1. Load Resources
2. Initialization
3. Render a rectangle
4. Specify color as uniform
5. Specify colors as additional attribute

main.js

```js
//the OpenGL context
var gl = null;

/**
 * initializes OpenGL context, compile shader, and load buffers
 */
function init(resources) {
  //create a GL context
  gl = createContext(400 /*width*/, 400 /*height*/);

  //TODO initialize shader, buffers, ...
}

/**
 * render one frame
 */
function render() {
  //specify the clear color
  gl.clearColor(0.9, 0.9, 0.9, 1.0);
  //clear the buffer
  gl.clear(gl.COLOR_BUFFER_BIT);

  //TODO render scene

  //request another call as soon as possible
  //requestAnimationFrame(render);
}

loadResources({
  //list of all resources that should be loaded as key: path
}).then(function (resources /*loaded resources*/) {
  init(resources);
  //render one frame
  render();
});
```

← 2. Initialize OpenGL

← 3. Render frame

← 1. Load external resources

1. Load External Resources

JOHANNES KEPLER UNIVERSITY LINZ

Institute of Computer Graphics

main.js → load external resources

```
74    //load the shader resources using a utility function
75    loadResources({
76      vs: 'shader/simple.vs.glsl',
77      fs: 'shader/simple.fs.glsl'
78    }).then(function (resources /*an object containing our keys
79      init(resources);
```

2. Initialization

1. `gl = createContext(width, height);`

   Utility function to create canvas and context to access WebGL

2. `createProgram(gl,resources.vs,resources.fs);`

   Utility function to compile shader

```
var vshader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vshader, code);
gl.compileShader(vshader);
var program = gl.createProgram();
gl.attachShader(program, vshader); //and fragment shader, too
gl.linkProgram(program);
```

3. Upload buffer data

```
var buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
gl.bufferData(gl.ARRAY_BUFFER, arr, gl.STATIC_DRAW);
```

Vertices for Drawing a Rectangle

```
const arr = new Float32Array([
    -1.0, -1.0,
     1.0, -1.0,
    -1.0,  1.0,
    -1.0,  1.0,
     1.0, -1.0,
     1.0,  1.0
]);
```

2. Initialization: Buffer

main.js → init: compile and upload buffer

```
19    //in WebGL / OpenGL3 we have to create and use our own shaders
20    //create the shader program
21    shaderProgram = createProgram(gl, resources.vs, resources.fs);
22
23    // Create a buffer and put a single clipspace rectangle in
24    // it (2 triangles)
25    buffer = gl.createBuffer();
26    gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
27    //we need typed arrays
28    const arr = new Float32Array([
29      -1.0, -1.0,
30      1.0, -1.0,
31      -1.0, 1.0,
32      -1.0, 1.0,
33      1.0, -1.0,
34      1.0, 1.0]);
35    //copy data to GPU
36    gl.bufferData(gl.ARRAY_BUFFER, arr, gl.STATIC_DRAW);
37  }
```

3. Render Frame

1. Clear existing frame buffer

```
gl.clearColor(0.9, 0.9, 0.9, 1);
gl.clear(gl.COLOR_BUFFER_BIT);
```

2. Render the scene
   a. Activate shader program

   ```
   gl.useProgram(shaderProgram);
   ```

   b. Activate and set attributes
   c. Draw elements

3. Render Frame

JOHANNES KEPLER UNIVERSITY LINZ

Institute of Computer Graphics

```
//activate this shader program
gl.useProgram(shaderProgram);

//we look up the internal location after compilation of the shader program given the name of the attribute
const positionLocation = gl.getAttribLocation(shaderProgram, 'a_position');

//enable this vertex attribute
gl.enableVertexAttribArray(positionLocation);
//use the currently bound buffer for this location
//each element is a FLOAT with 2 components
//2 .. number of components
//float ... type
//false ... the array should not be normalized
//stride / offset ... in case you are interleaving different attribute
gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
gl.vertexAttribPointer(positionLocation, 2, gl.FLOAT, false, 0, 0);

// draw the bound data as 6 vertices = 2 triangles starting at index 0
gl.drawArrays(gl.TRIANGLES, 0, 6);
```

Vertex Shader and Fragment Shader

OpenGL Shader Language (GLSL)

C-like syntax

Customized data types: mat4, vec2, vec3, vec4

```
vec4 v = vec4(0, 1, 2, 3);
v.x, v.y, v.xy, v.xyzw, v.rgba, v.zx, 5*v
mat4 m =
mat4(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15)
m[0][0], m[0], v * m
entry point: main function
main() {
    gl_FragColor = vec4(1,0,0,1);
}
```

Programmable Pipeline: Vertex Shader

simple.vs.glsl

```
5
6   //attributes: per vertex inputs in this case the 2d position and its color
7   attribute vec2 a_position;
8
9   //like a C program main is the main function
10  void main() {
11    //gl_Position .. magic output variable storing the vertex 4D position
12    gl_Position = vec4(a_position, 0, 1);
13  }
14
```

Programmable Pipeline: Fragment Shader

simple.fs.glsl

```
5
6   //need to specify how "precise" float should be
7   precision mediump float;
8
9   //entry point again
10  void main() {
11    //gl_FragColor ... magic output variable containing
12    //                 the final 4D color of the fragment
13    gl_FragColor = vec4(0, 1, 0, 1);
14  }
15
```

color: green

Render Rectangle

Render Rectangle

Is there really a rectangle or just a strange cleared framebuffer…

Let’s shrink the rectangle to 90%

Render Rectangle

Is there really a rectangle or just a strange cleared framebuffer…

Let's shrink the rectangle to 90%

```
attribute vec2 a_position;

//like a C program main is the main function
void main() {
  //gl_Position .. magic output variable storing the vertex 4D position
  gl_Position = vec4(a_position * 0.9, 0, 1);
}
```

Hard coded colors are bad practice, let's parameterize it

Programmable Pipeline

Shader Parameter

- attributes
  streams for the vertex shader stored in buffer,
  e.g., vertex position
- uniforms
  parameter from the program to the shader,
  e.g., light position, texture reference
- varyings
  out/input between shader stages with interpolation
  e.g., vertex color → fragment color

4. Specifying Color via Uniform

JOHANNES KEPLER UNIVERSITY LINZ

Institute of Computer Graphics

Recap: uniforms are parameters for a shader.
In contrast to attributes, they remain the same for each vertex.

simple.fs.glsl

```
//uniform like a parameter for all fragment shader instances.
//In our case a the rgb color to use
uniform vec3 u_usercolor;

//entry point again
void main() {
  //gl_FragColor ... magic output variable containing
  //                 the final 4D color of the fragment
  //gl_FragColor = vec4(0, 1, 0, 1);
  gl_FragColor = vec4(u_usercolor, 1);
}
```

main.js → render() after gl.useProgram

```
//we use a uniform to specify the rectangle color
//a uniform is like a parameter to a shader (vertex or fragment).
//However, the same value is used for all instances
var userColor = { r: 0.6, g: 0.2, b: 0.8};
gl.uniform3f(gl.getUniformLocation(shaderProgram, 'u_usercolor'),
             userColor.r, userColor.g, userColor.b);
```

4. Specifying Color via Uniform

What about a color gradient, i.e., a color per vertex?

→ another buffer for the colors instead of a uniform

5. Specifying Color per Vertex

JOHANNES KEPLER UNIVERSITY LINZ

Institute of Computer Graphics

main.js → init()

```
    //same for the color
    colorBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
    const colors = new Float32Array([
      1, 0, 0, 1,
      0, 1, 0, 1,
      0, 0, 1, 1,
      0, 0, 1, 1,
      0, 1, 0, 1,
      0, 0, 0, 1]);
    gl.bufferData(gl.ARRAY_BUFFER, colors, gl.STATIC_DRAW);

  }
```

main.js → render()

```
    const colorLocation = gl.getAttribLocation(shaderProgram, 'a_color');
    gl.enableVertexAttribArray(colorLocation);
    gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
    gl.vertexAttribPointer(colorLocation, 4, gl.FLOAT, false, 0, 0);

    // draw the bound data as 6 vertices = 2 triangles starting at index 0
    gl.drawArrays(gl.TRIANGLES, 0, 6);
```

5. Specifying Color per Vertex

JOHANNES KEPLER UNIVERSITY LINZ

Institute of Computer Graphics

Recap: attributes are data for each vertex. varyings are parameters between shader stages that are interpolated during rasterization

simple.vs.glsl

```
attribute vec4 a_color;

//values transfered and interpolated between vertex and fragment shader
varying vec4 v_color;


//like a C program main is the main function
void main() {
  //gl_Position .. magic output variable storing the vertex 4D position
  gl_Position = vec4(a_position * 0.9, 0, 1);

  //just copy the input color to the output varying color
  v_color = a_color;
}
```

simple.fs.glsl

```
//interpolate argument between vertex and fragment shader
varying vec4 v_color;

//entry point again
void main() {
  //gl_FragColor ... magic output variable containing
  //                 the final 4D color of the fragment
  //gl_FragColor = vec4(0, 1, 0, 1);
  //gl_FragColor = vec4(u_usercolor, 1);
  gl_FragColor = v_color;
}
```

Recap

1. Administrative Issues
2. What is OpenGL / WebGL
3. Programmable Rendering Pipeline
4. First Application: Colored rectangle
   a. initialize context
   b. define buffer, compile shader
   c. draw rectangle using two triangles
   d. specify uniforms
   e. specify color per vertex

Next Time

Going to 3D

Transformations

- Translate
- Rotate
- Scale
- Projections (3D → 2D): perspective vs. orthographic
- Combined transformations

Object → world → camera → normalized coordinates

References

The Official Guide to Learning OpenGL, Dave Shreiner, Mason Woo, Jackie Neider, Addison-Wesley Longman,
ISBN: 0321335732, 2005

The Official Reference Document to OpenGL, OpenGL Architecture Review Board, Dave Shreiner, Addison-Wesley Professional, 032117383X, 2004

Also available online:

http://glprogramming.com/red/

http://glprogramming.com/blue/

http://opengl.org

http://www.khronos.org/opengl/

http://www.ntu.edu.sg/home/ehchua/programming/opengl/cg_basicstheory.html

(some images in slides taken from this source)

There are many, many tutorials on the internet!

Thank you