JOHANNES KEPLER
UNIVERSITY LINZ
INSTITUTE OF
COMPUTER GRAPHICS

Computer Graphics

Lab 1: Introduction to WebGL

CG Lab Team





Philipp
Kern
(Tutor)

Jonathan
Kudlich
(Tutor)

Marc
Streit

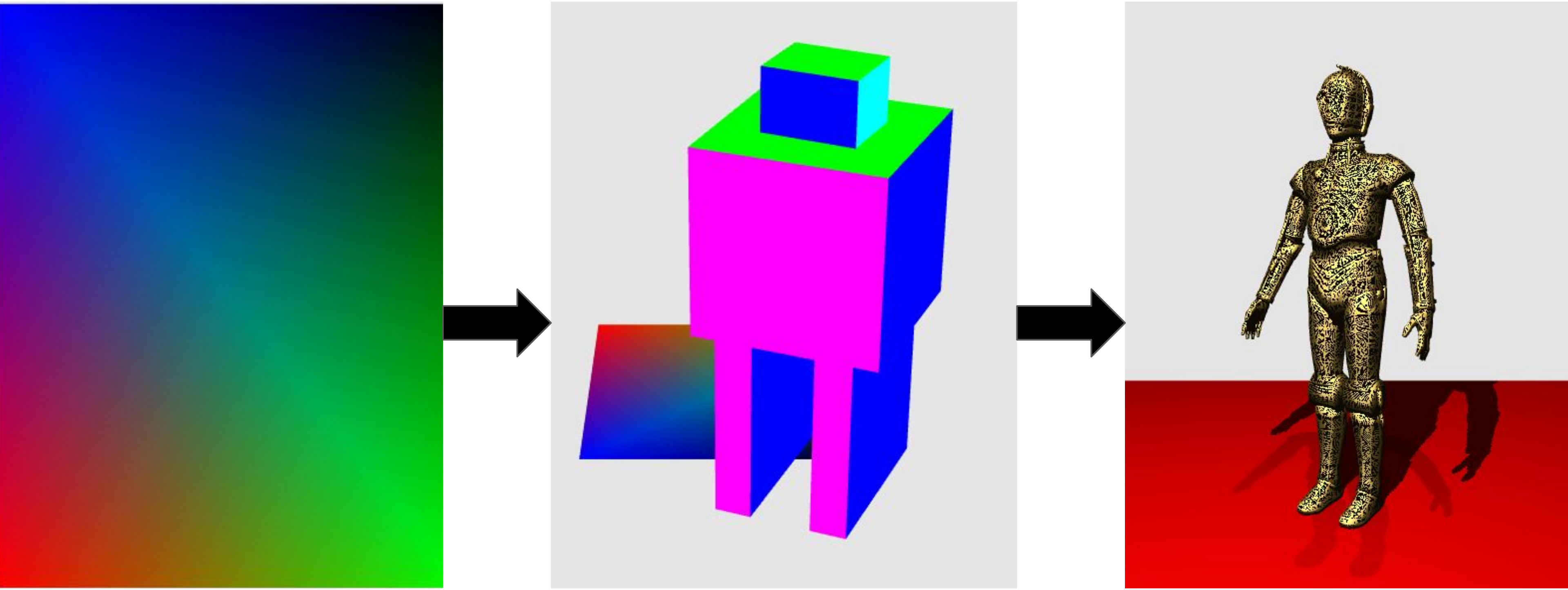
Günter
Wallner

Indrajit
Kurmi

Patrick
Adelberger

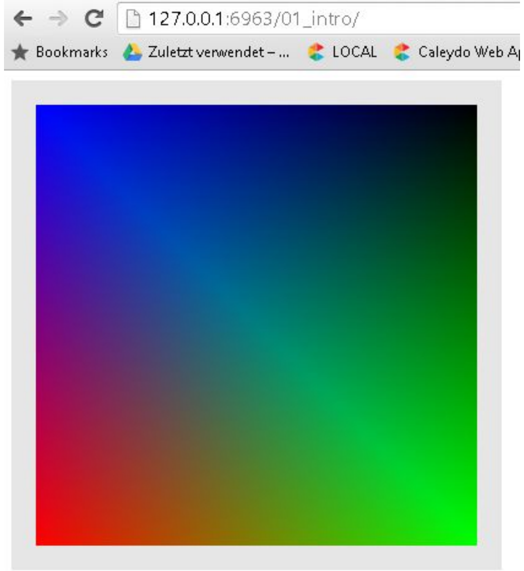
Klaus
Eckelt

What you can expect...



Today’s Lab

- Lab organization
- What is OpenGL?
  - Programmable Pipeline
  - Definitions
- Development environment
  - Our lab package
  - JavaScript, HTML5, CSS
  - Visual Studio Code + Live-Server
  - Chrome|Firefox Developer Tools



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

General Overview

Lecture: 2h, 3 ECTS

Labs: 1h, 1.5 ECTS

9 KUSSS groups assigned to 5 time slots (A-E)

Contact

- Use the Moodle discussion forum
- General point of contact for individual questions: cg-lab@jku.at

Tutorial slides and lab material available in Moodle

More information on other courses & invited talks

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

Lab Organization

There are 7 mandatory lab sessions

- For lab 7 you can pick one of the two topics.

Labs will take place online via Zoom only

Five time slots (A,B,C,D,E) for each lab session

- A and B will be held on Tuesdays
- C, D, and E will be held on Fridays (with one exception)

Lab Schedule

Lab 1	Introduction to WebGL	Week from March 8
Lab 2	Transformations and Projections	Week from March 15
Lab 3	Scene Graphs	Week from March 22
Lab 4	Illumination and Shading	Week from April 12
Lab 5	Texturing	Week from April 19
Lab 6	Advanced Texture Mapping	Week from April 26
Lab 7a	Project Q&A	1.6. 15:30-17:00, 11.6. 08:30-10:00
Lab 7b	Introduction to CUDA	11.6. 10:00-11:30 & 12:15-13:45

Lab Overview

Use your own laptop

- First labs: WebGL compatible browser
- Last (optional) CUDA lab: shader model 4, Nvidia only

Lab sessions

- Assignment to time slots (A-E) is less strict due to remote setup
- If you cannot join a lab session to any of the 5 time slots contact us: cg-lab@jku.at

Lab Overview

Prerequisites

- Labs require basic knowledge of JavaScript programming
- We do not teach JavaScript programming!

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

Modus Operandi in Lab Sessions

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

Grading

- **Attendance to each lab session is mandatory!**
  - Inform us when you cannot attend due to sickness or other important reasons
  - If you miss >1 lab sessions without a legitimate reason you will be unregistered
- Lab project grade = course grade
  - No lab exam at the end
- Lab project in groups of two students
  - Independent of KUSSS groups and time slots A-E
- Interviews at the end
  - Theoretical concept questions + code walkthrough

CG Lab Project: Create a Movie

30 seconds WebGL movie

Use our framework

Implementation tasks

- Requirements & basic effects (e.g. scenegraph, lighting, …)
- Special effects of your choice

Detailed specification will soon be available in Moodle

Example from 2017

CG Lab Project: Create a Movie

Group project in teams of 2 students

Submissions via Github

- **26.03.2021 23:59:** Movie concept submission (incl. team announcement)
- **23.04.2021 23:59:** Intermediate submission
- **22.06.2021 23:59:** Hand-in final package

Individual interviews (alone): **24.-30.06.2021**

Let’s start…

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

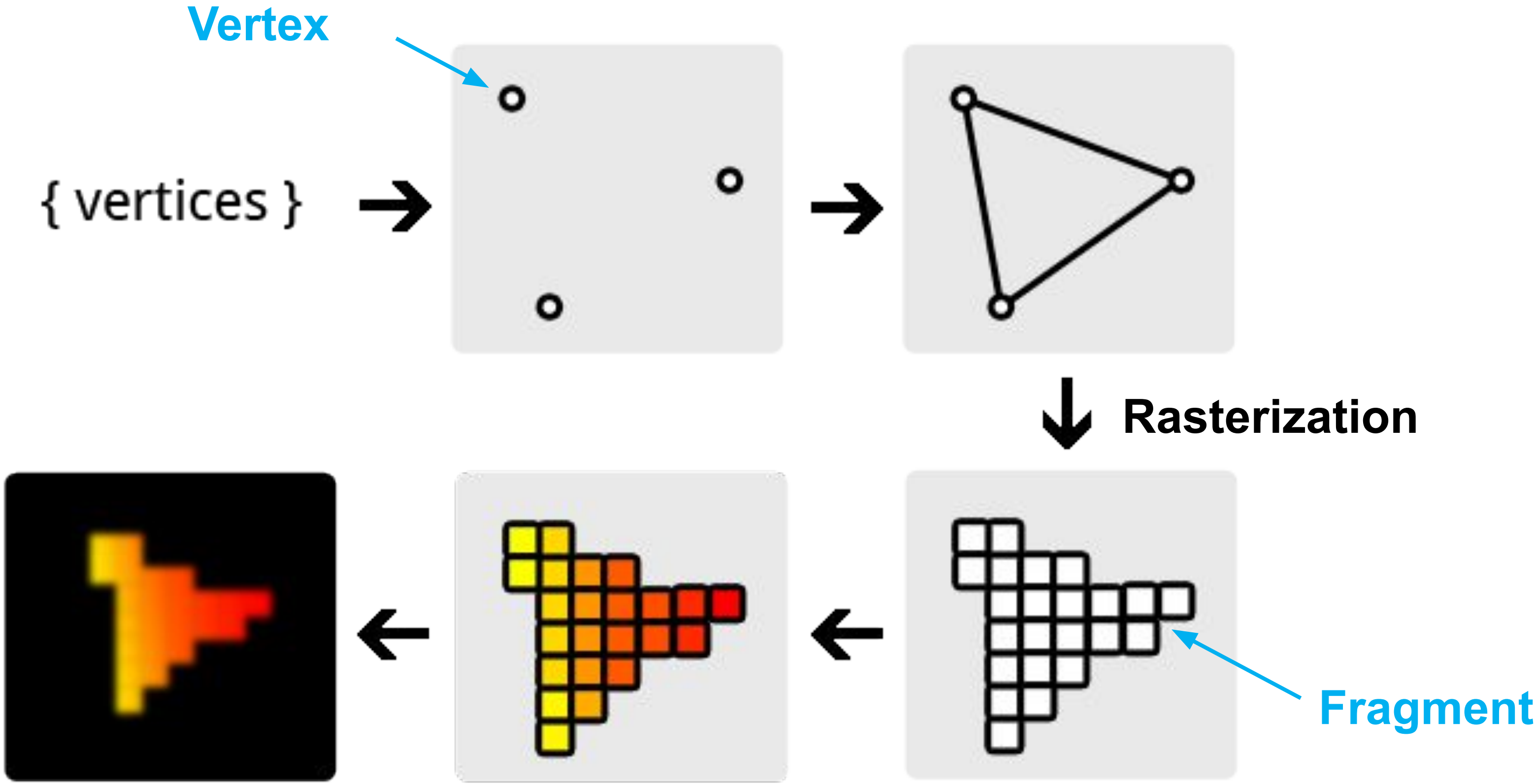
… is platform independent

… is a procedural graphics API, i.e., state machine

… provides no window management, user interaction, or file I/O

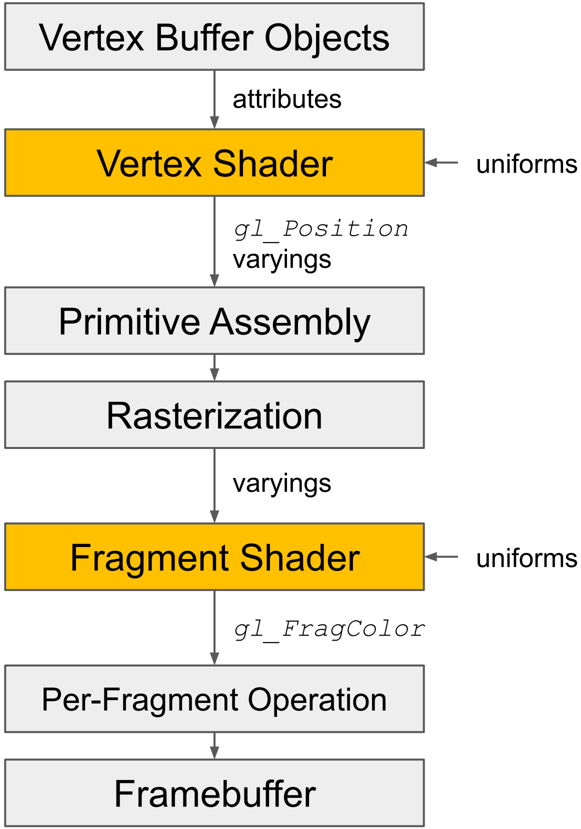
… special versions for different platforms: OpenGL ES, WebGL

Rendering Pipeline



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

Programmable Pipeline



Vertex Buffer Objects

attributes

Vertex Shader ← uniforms

gl_Position
varyings

Primitive Assembly

Rasterization

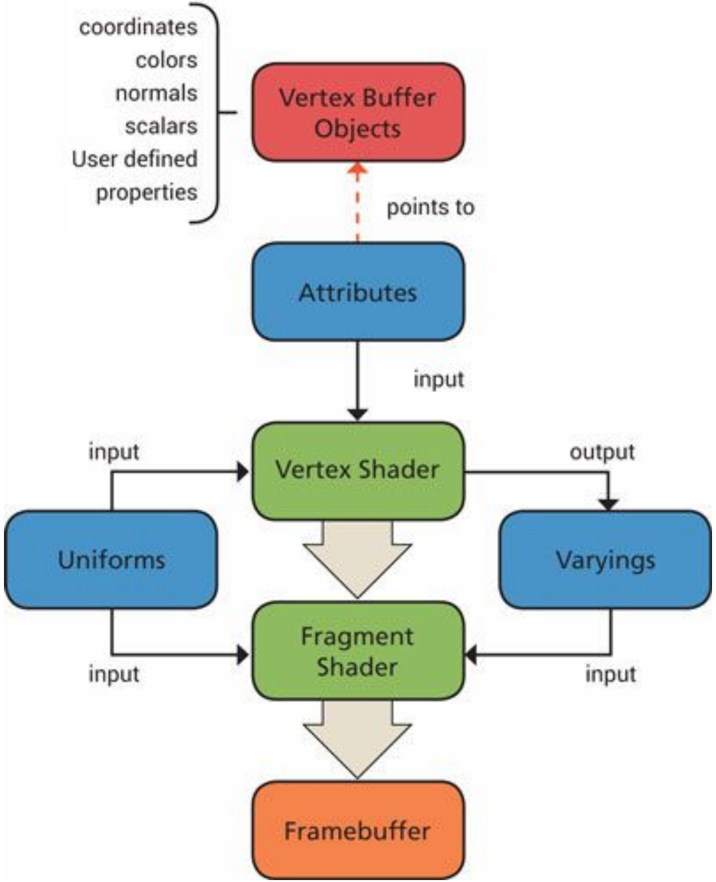
varyings

Fragment Shader ← uniforms

gl_FragColor

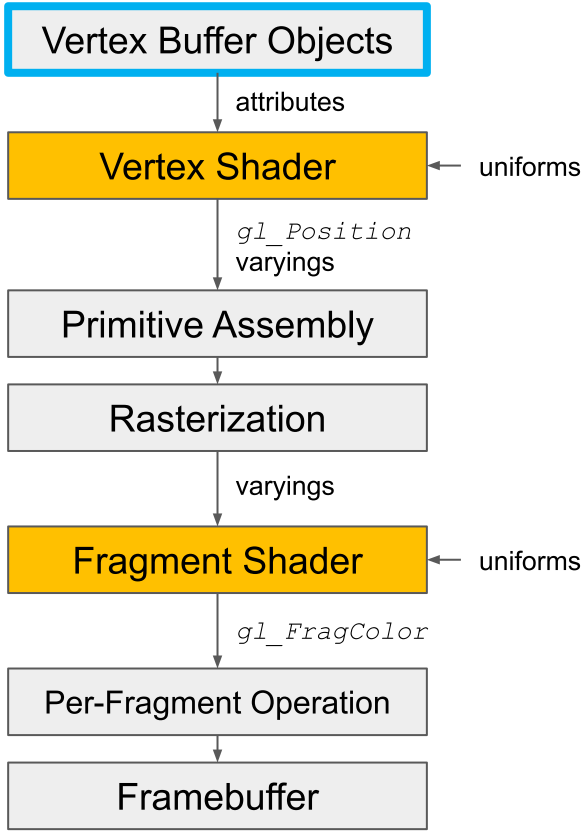
Per-Fragment Operation

Framebuffer



Simple overview of WebGL programmable pipeline

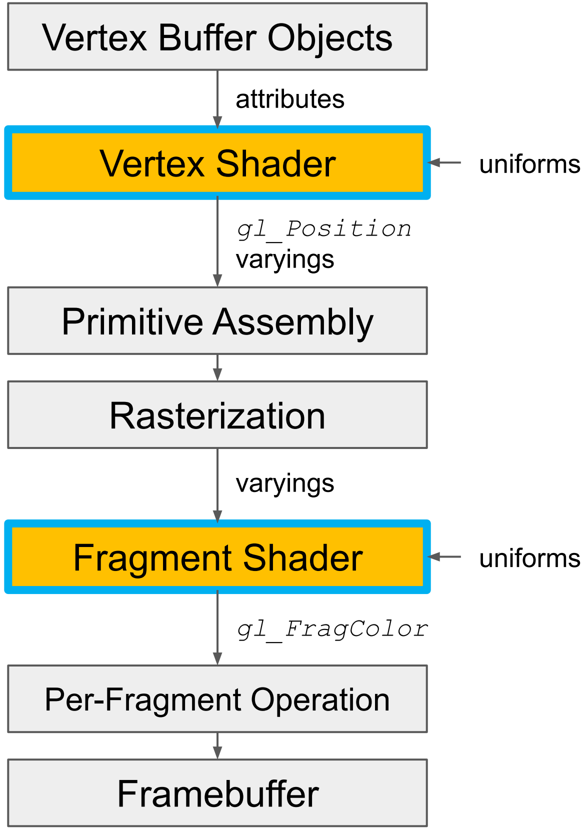
Programmable Pipeline



Vertex Buffer Objects

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

Programmable Pipeline



Vertex Buffer Objects

attributes

Vertex Shader ← uniforms

gl_Position
varyings

Primitive Assembly

Rasterization

varyings

Fragment Shader ← uniforms

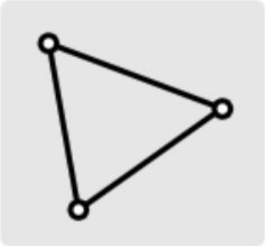
gl_FragColor

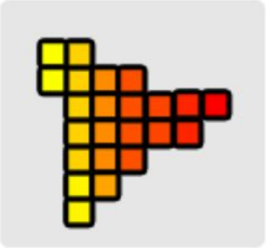
Per-Fragment Operation

Framebuffer

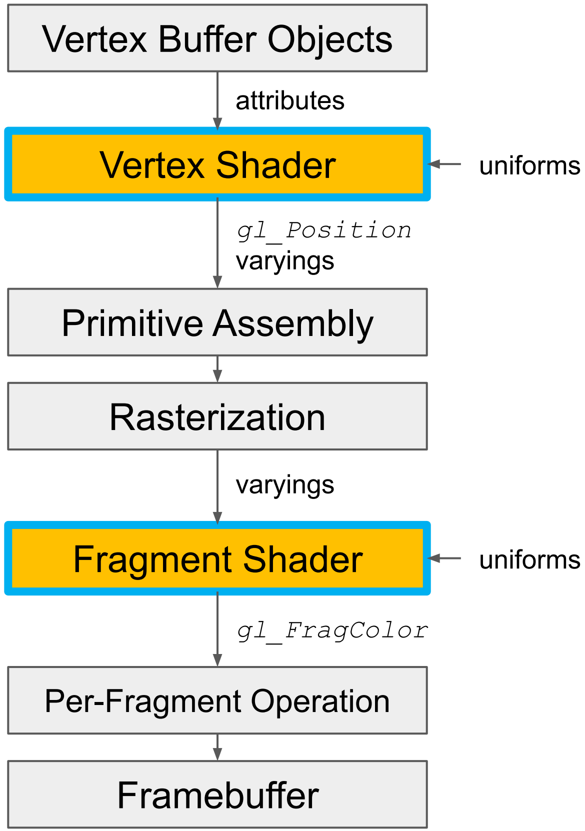
Shader

- 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



Vertex Buffer Objects

attributes

Vertex Shader ← uniforms

`gl_Position`
varyings

Primitive Assembly

Rasterization

varyings

Fragment Shader ← uniforms

`gl_FragColor`

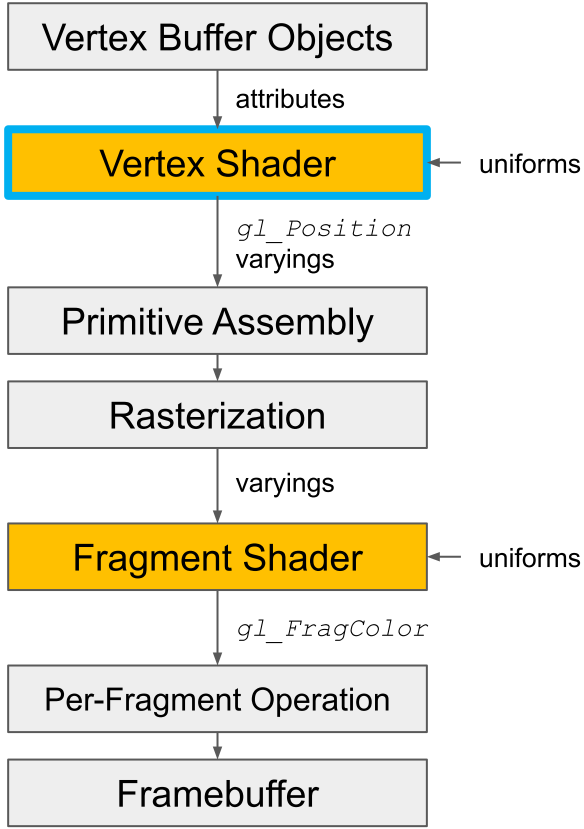
Per-Fragment Operation

Framebuffer

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

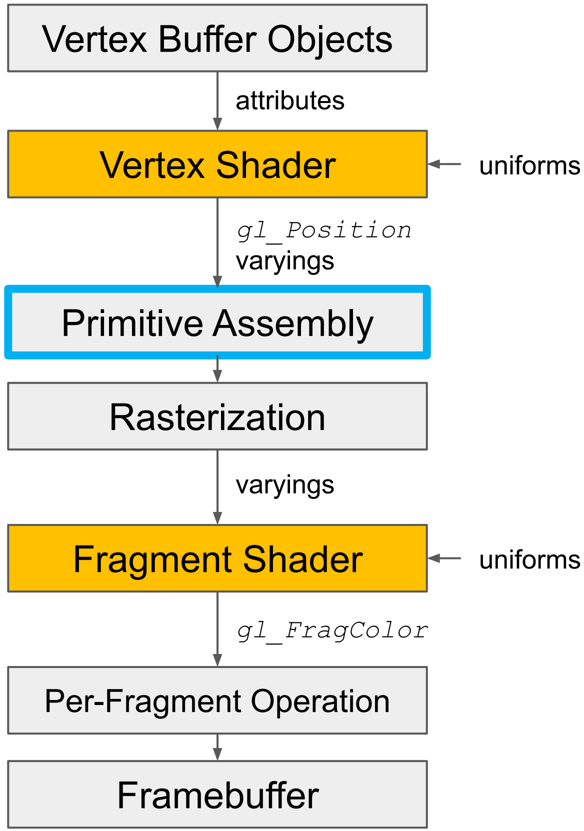
Programmable Pipeline



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



Vertex Buffer Objects

attributes

Vertex Shader

uniforms

gl_Position
varyings

Primitive Assembly

Rasterization

varyings

Fragment Shader

uniforms

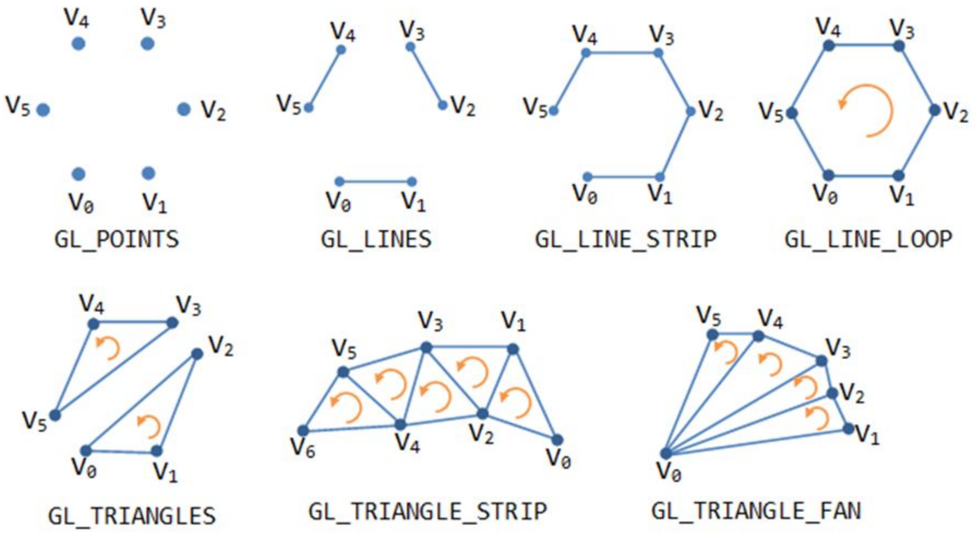
gl_FragColor

Per-Fragment Operation

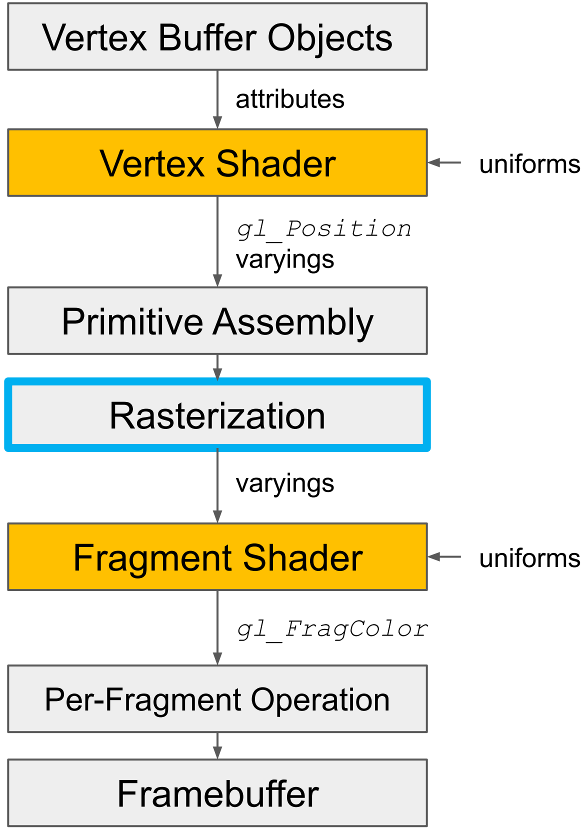
Framebuffer

Primitive Assembly

takes vertices and build primitives out of it for rasterization



Programmable Pipeline



Vertex Buffer Objects

attributes

Vertex Shader ← uniforms

gl_Position
varyings

Primitive Assembly

Rasterization

varyings

Fragment Shader ← uniforms

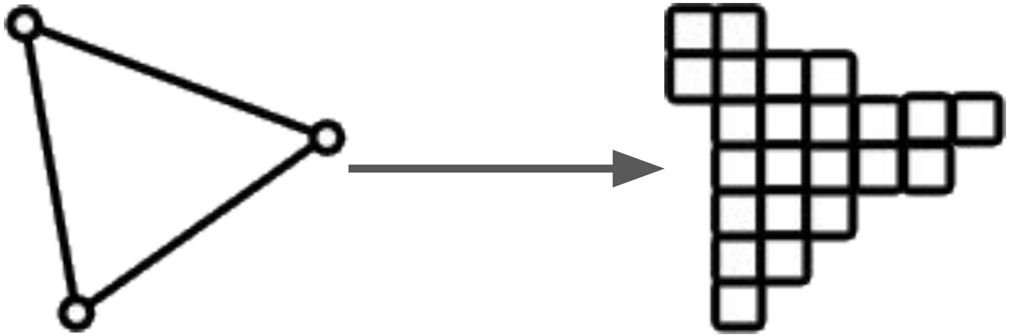
gl_FragColor

Per-Fragment Operation

Framebuffer

Rasterization

converts abstract primitives to set of fragments



Programmable Pipeline

Vertex Buffer Objects

attributes

Vertex Shader ← uniforms

gl_Position
varyings

Primitive Assembly

Rasterization

varyings

Fragment Shader ← uniforms

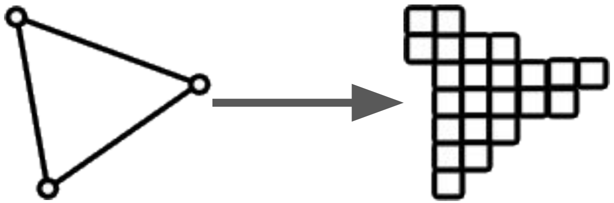
gl_FragColor

Per-Fragment Operation

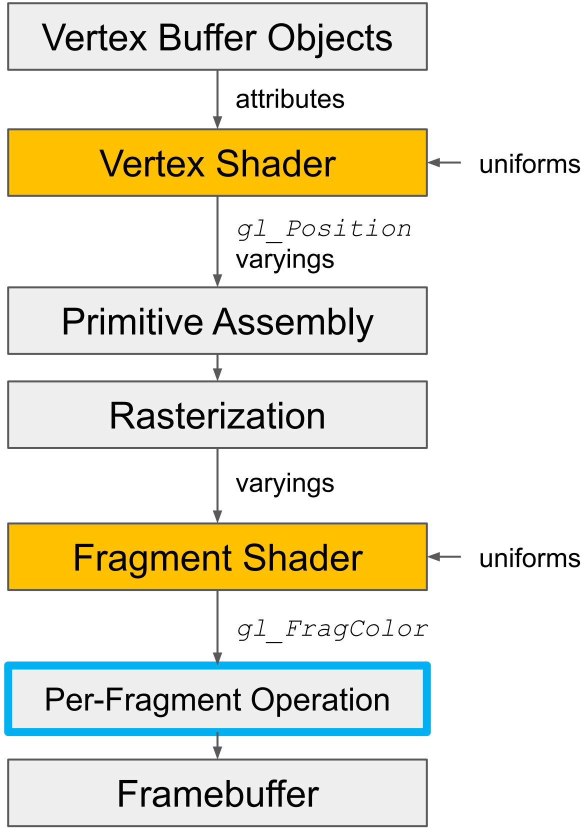
Framebuffer

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



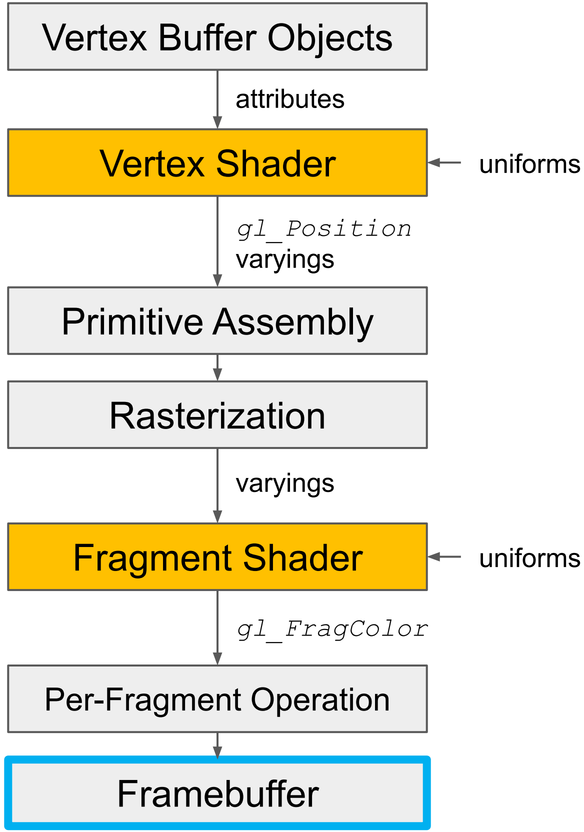
Programmable Pipeline



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


Vertex Buffer Objects

attributes

Vertex Shader

uniforms

`gl_Position`

varyings

Primitive Assembly

Rasterization

varyings

Fragment Shader

uniforms

`gl_FragColor`

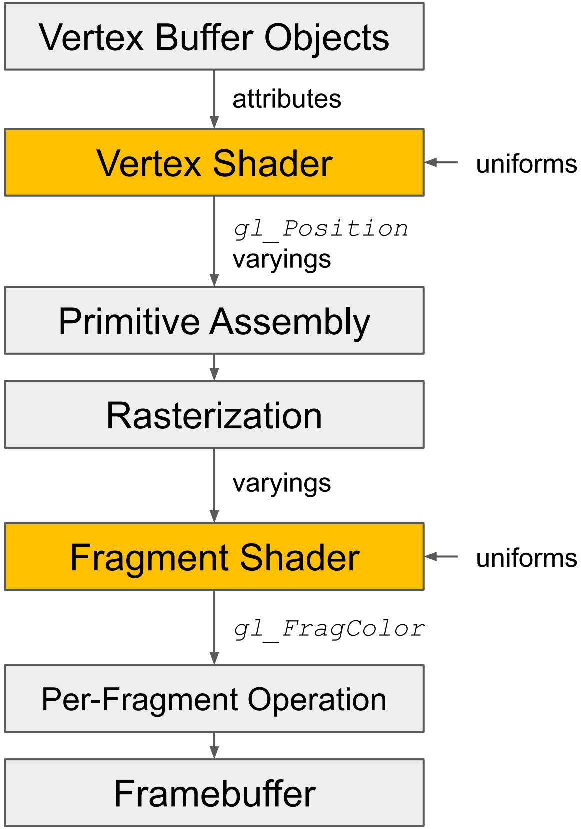
Per-Fragment Operation

Framebuffer

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


Vertex Buffer Objects
attributes
Vertex Shader
uniforms
gl_Position
varyings
Primitive Assembly
Rasterization
varyings
Fragment Shader
uniforms
gl_FragColor
Per-Fragment Operation
Framebuffer

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: Visual Studio Code

A good editor / IDE simplifies your life!

We use Visual Studio Code but you can use anyone you like

If you use your own machine

- **Install Live-Server for VS Code**
- You need to access your WebGL website through a local web server.
- Reason? Security! Browsers don’t allow to load local files asynchronously
- **Install Shader languages support for VS Code**

Fork me on GitHub

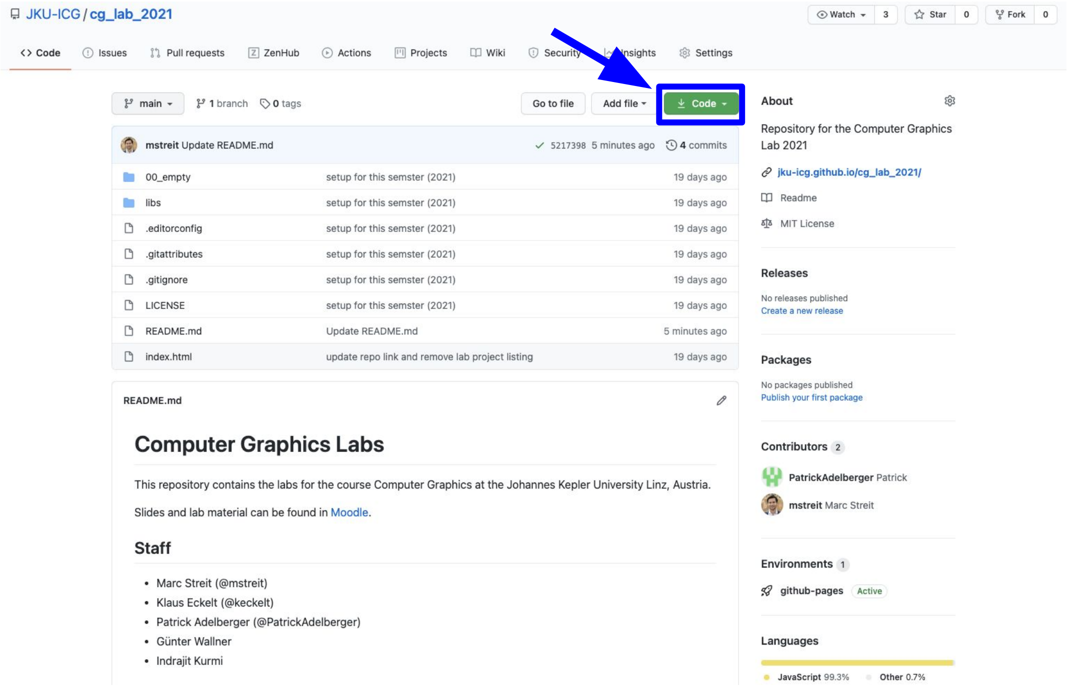
Dev Environment: Lab Package

Hosted on GitHub: https://github.com/jku-icg/cg_lab_2021

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 Visual Studio Code
4. Open `cg_lab_2021` folder (*File → Open*)
5. Click on **Go Live** button in lower right corner



mstreit Update README.md ✓ 5217398 5 minutes ago 4 commits

00_empty	setup for this semster (2021)	19 days ago
libs	setup for this semster (2021)	19 days ago
.editorconfig	setup for this semster (2021)	19 days ago
.gitattributes	setup for this semster (2021)	19 days ago
.gitignore	setup for this semster (2021)	19 days ago
LICENSE	setup for this semster (2021)	19 days ago
README.md	Update README.md	5 minutes ago
index.html	update repo link and remove lab project listing	19 days ago

README.md

Computer Graphics Labs

This repository contains the labs for the course Computer Graphics at the Johannes Kepler University Linz, Austria.

Slides and lab material can be found in Moodle.

Staff

- Marc Streit (@mstreit)
- Klaus Eckelt (@keckelt)
- Patrick Adelberger (@PatrickAdelberger)
- Günter Wallner
- Indrajit Kurmi

About

Repository for the Computer Graphics Lab 2021

jku-icg.github.io/cg_lab_2021/

Readme

MIT License

Releases

No releases published
Create a new release

Packages

No packages published
Publish your first package

Contributors 2

PatrickAdelberger Patrick
mstreit Marc Streit

Environments 1

github-pages Active

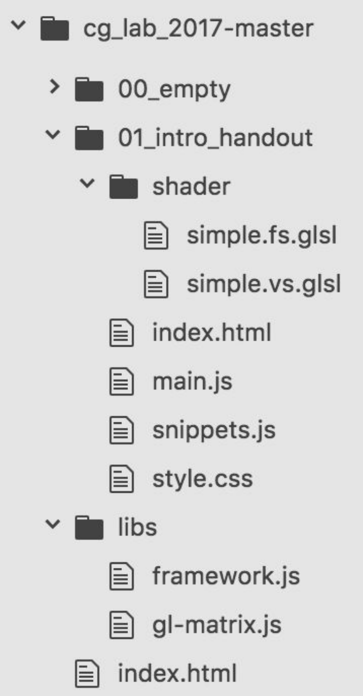
Languages

JavaScript 99.3% Other 0.7%

Dev Environment: HTML5, JS, CSS

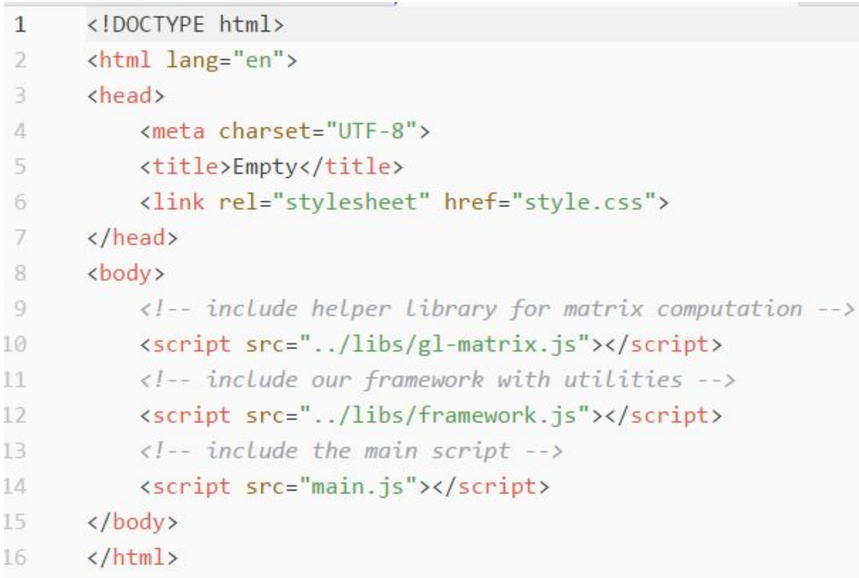
WebGL → OpenGL in the web-browser based on OpenGL ES 2.0

Basic project structure:

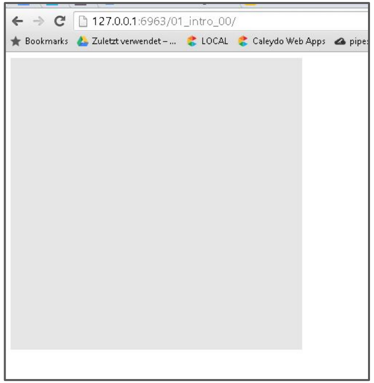


- cg_lab_2017-master
  - 00_empty
  - 01_intro_handout
    - shader
      - simple.fs.glsl
      - simple.vs.glsl
    - index.html
    - main.js
    - snippets.js
    - style.css
  - libs
    - framework.js
    - gl-matrix.js
  - index.html

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>
```

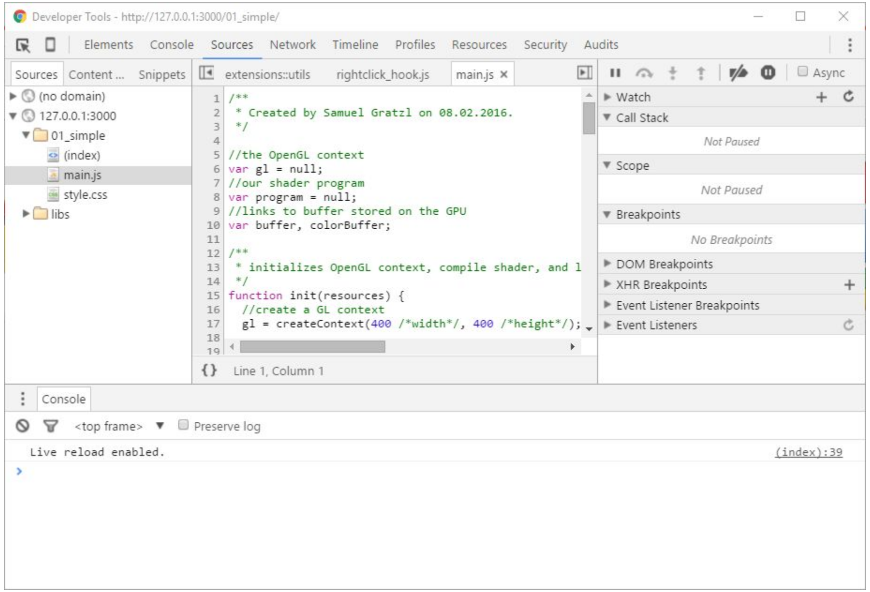


Dev Environment: Developer Tools

Know the Web Developer Tools of your favorite browser

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

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

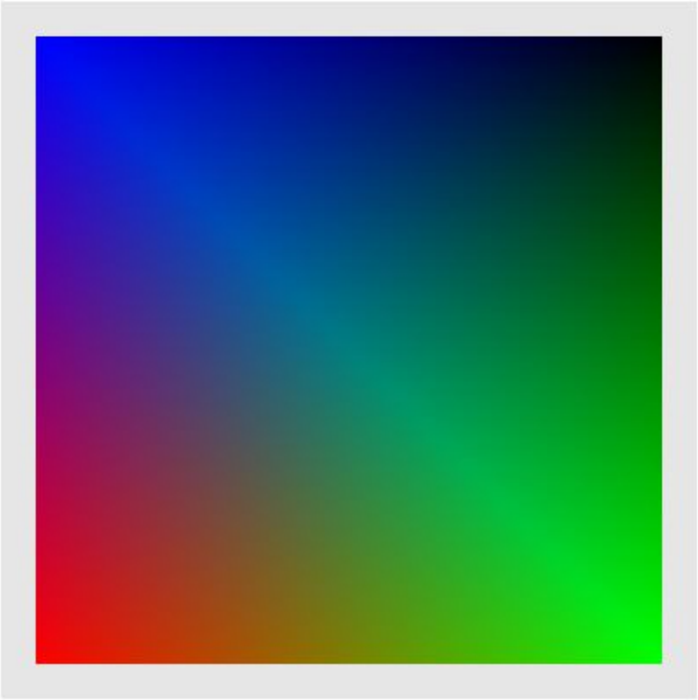


```
/**
 * Created by Samuel Gratzl on 08.02.2016.
 */

//the OpenGL context
var gl = null;
//our shader program
var program = null;
//links to buffer stored on the GPU
var buffer, colorBuffer;

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

```

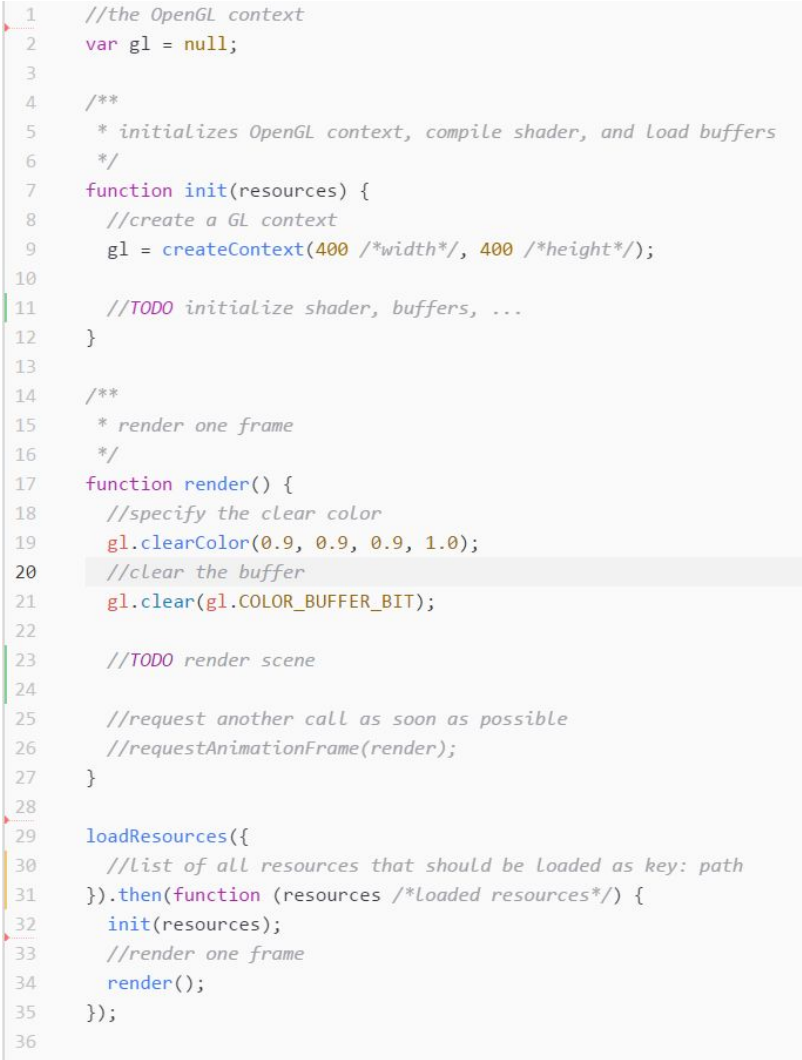
```
Live reload enabled.
>
```

First Application

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



main.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();
});

```

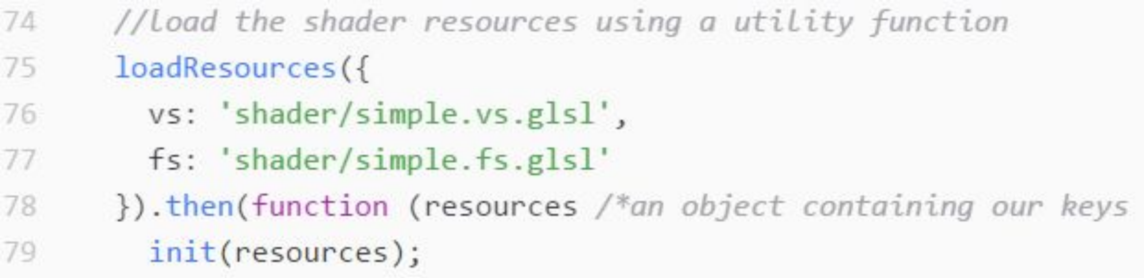
← 2. Initialize OpenGL

← 3. Render frame

← 1. Load external resources

1. Load External Resources

main.js → load external resources



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

2. Initialization

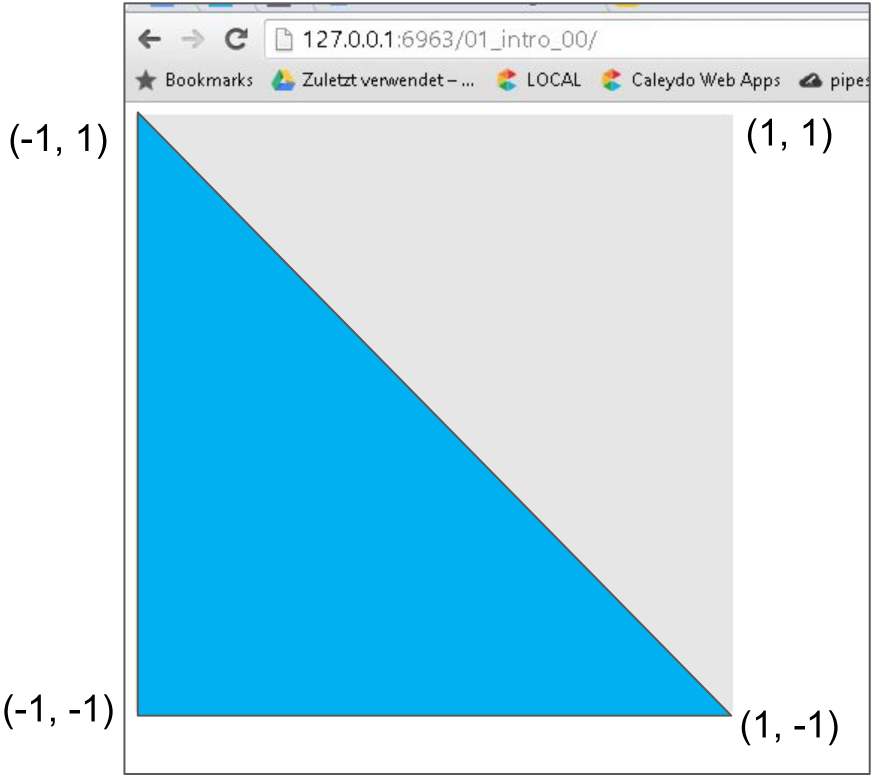
1. **`gl = createContext(width, height);`**
   Utility framework function to create canvas and context to access WebGL
2. **`createProgram(gl,resources.vs,resources.fs);`**
   Utility framework function to compile shader (implemented in libs/framework.js):

```
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

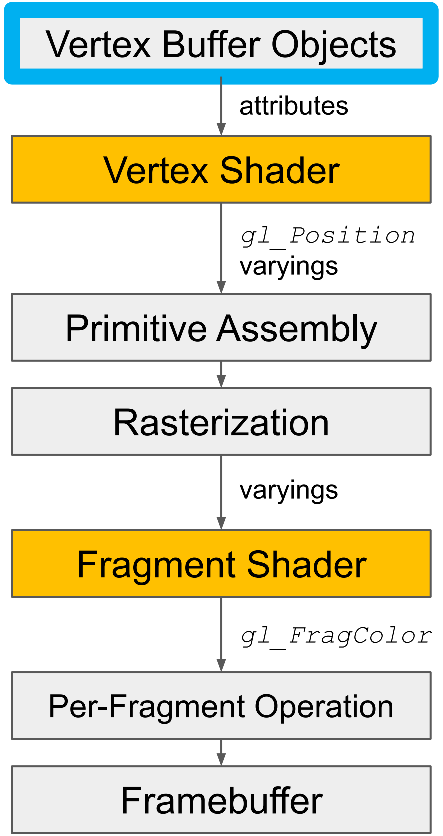
```
var buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
gl.bufferData(gl.ARRAY_BUFFER, arr, gl.STATIC_DRAW);
```

Vertices for Drawing a Rectangle

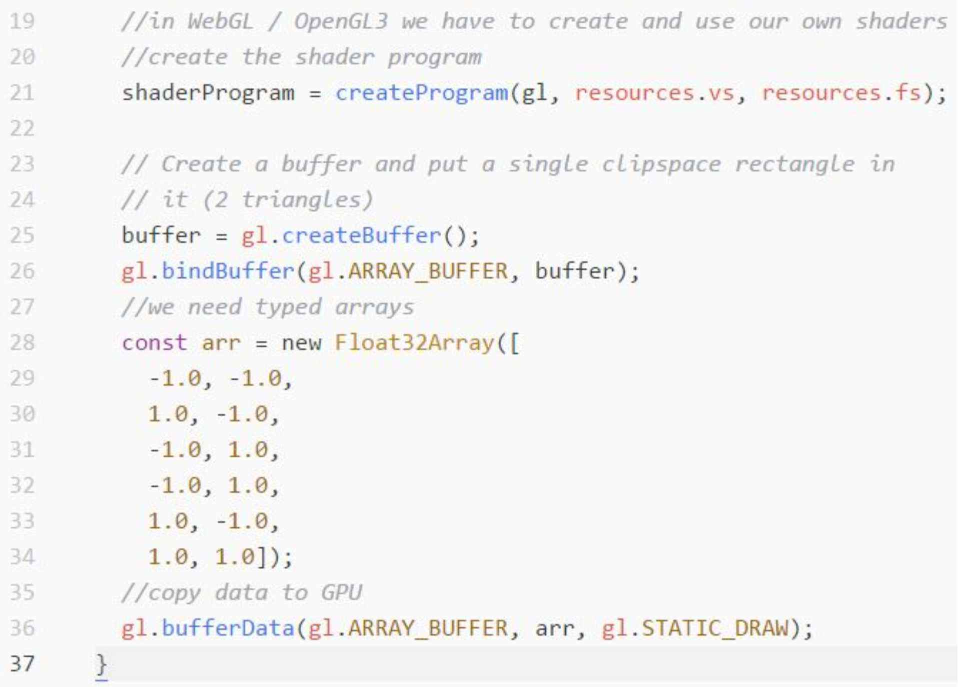


```
const arr = new Float32Array([
     -1.0, -1.0,
      1.0, -1.0,
     -1.0,  1.0,
     -1.0,  1.0,
      1.0, -1.0,
      1.0,  1.0
]);
```

2. Initialization: Buffer



main.js → init(): compile and upload buffer



```
    //in WebGL / OpenGL3 we have to create and use our own shaders
    //create the shader program
    shaderProgram = createProgram(gl, resources.vs, resources.fs);

    // Create a buffer and put a single clipspace rectangle in
    // it (2 triangles)
    buffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
    //we need typed arrays
    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]);
    //copy data to GPU
    gl.bufferData(gl.ARRAY_BUFFER, arr, gl.STATIC_DRAW);
}
```

3. Render Frame

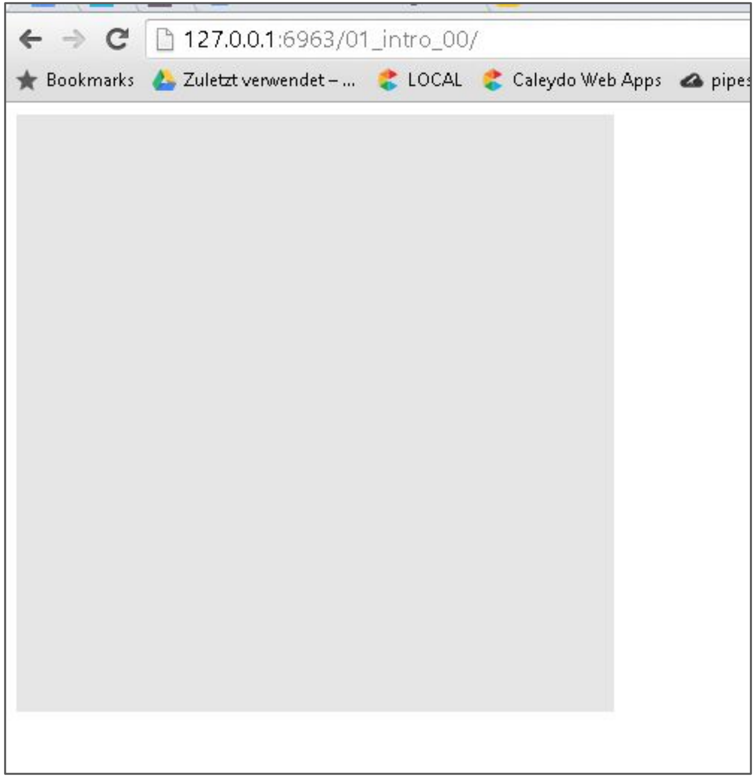
1. Clear existing frame buffer

```
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

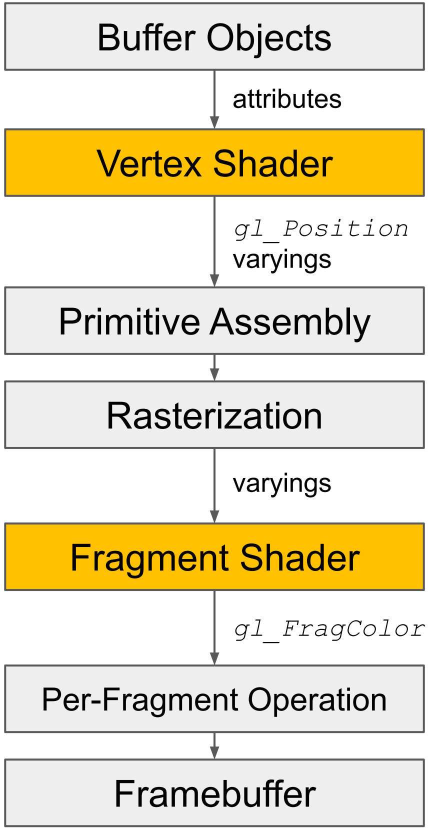


3. Render Frame

main.js → render()

```
//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



Buffer Objects

attributes

Vertex Shader

gl_Position
varyings

Primitive Assembly

Rasterization

varyings

Fragment Shader

gl_FragColor

Per-Fragment Operation

Framebuffer

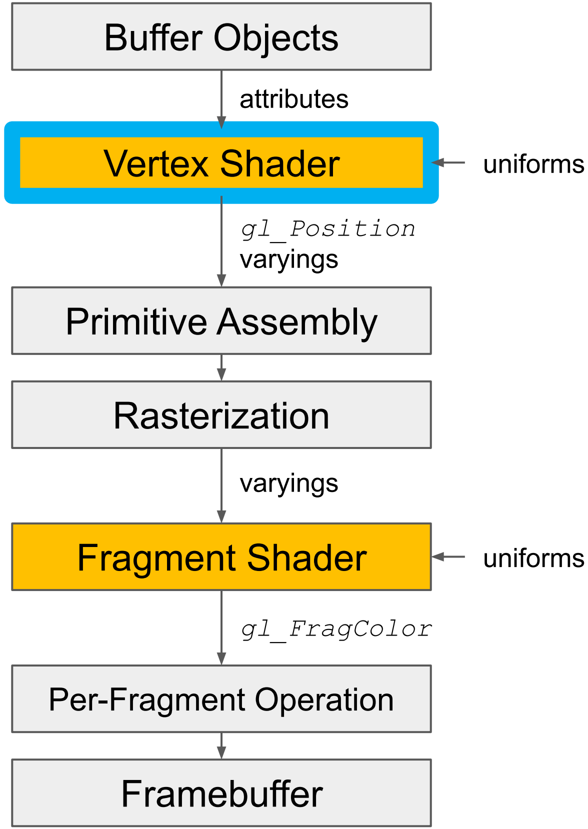
OpenGL Shader Language (GLSL)

C-like syntax

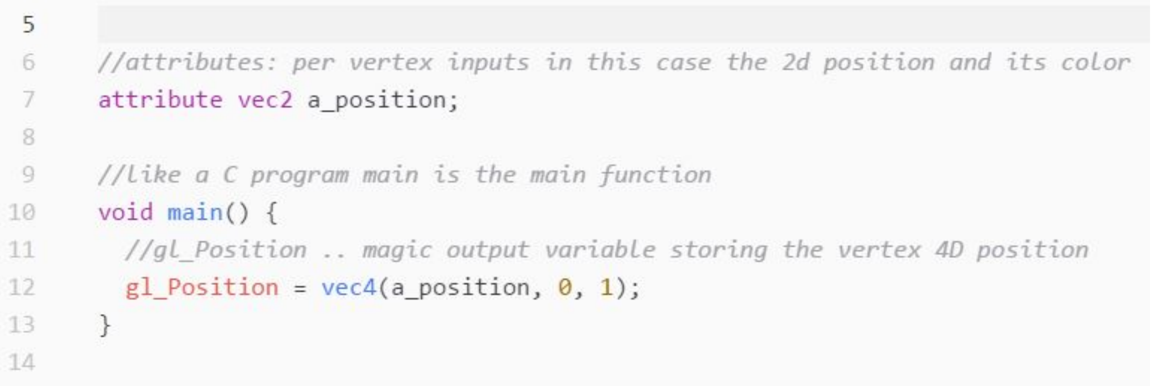
Customized data types: mat4, vec2, vec3, vec4

```
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

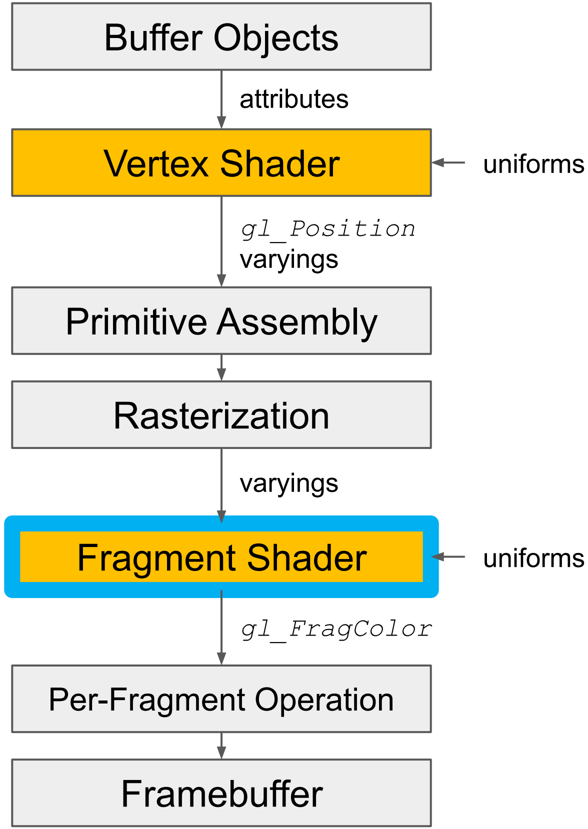


simple.vs.glsl



```

//attributes: per vertex inputs in this case the 2d position and its color
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, 1);
}

```

Programmable Pipeline: Fragment Shader


Buffer Objects

attributes

Vertex Shader ← uniforms

gl_Position
varyings

Primitive Assembly

Rasterization

varyings

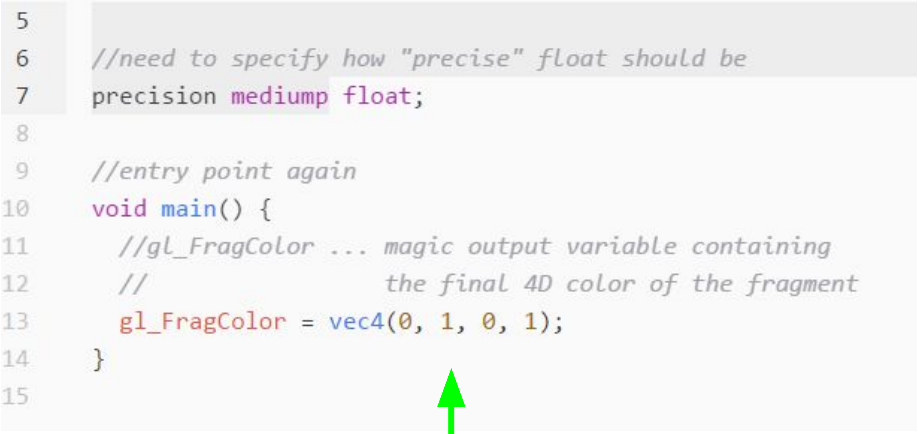
Fragment Shader ← uniforms

gl_FragColor

Per-Fragment Operation

Framebuffer

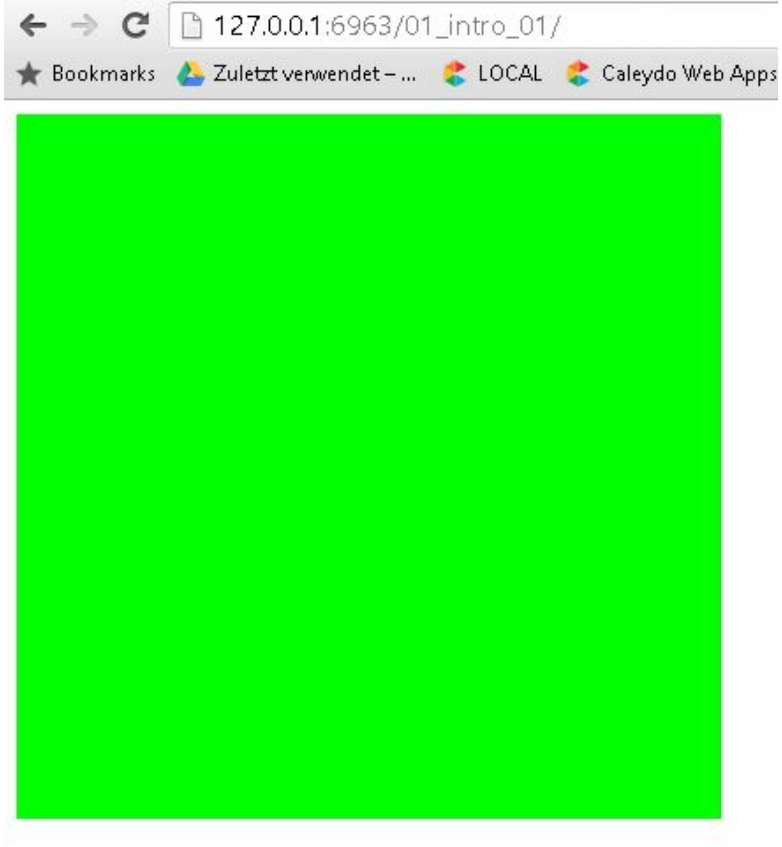
simple.fs.glsl


```glsl

//need to specify how "precise" float should be
precision mediump float;

//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);
}

```

color: green

Render Rectangle

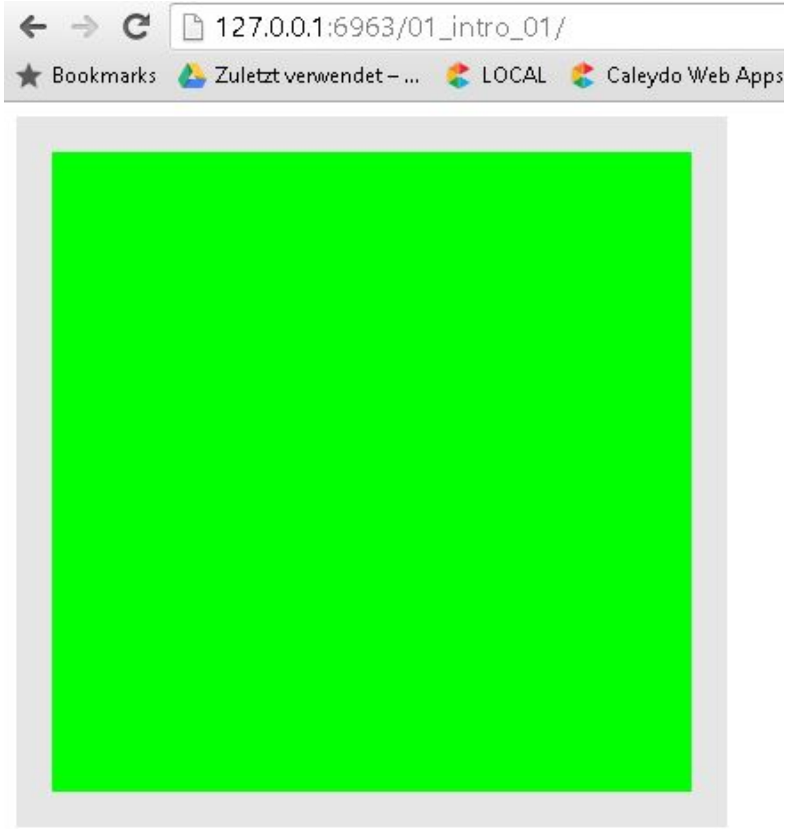
Render Rectangle



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

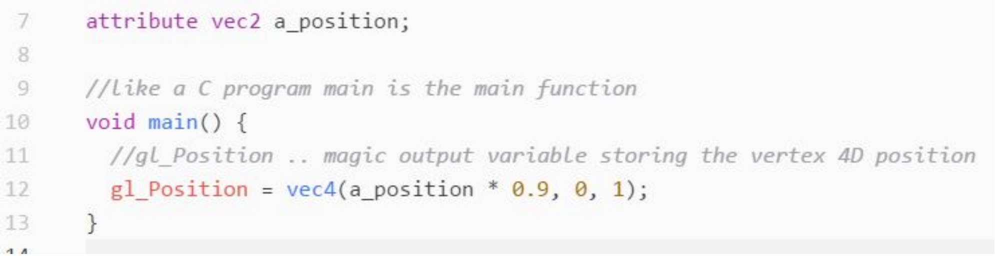
Let’s shrink the rectangle to 90%

Render Rectangle



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

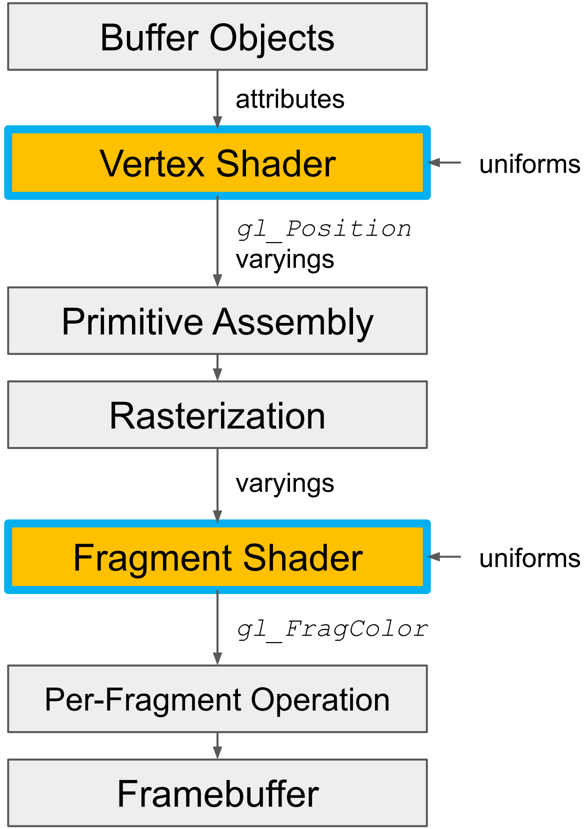
Let’s shrink the rectangle to 90%



```
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, so let’s parameterize them

Programmable Pipeline



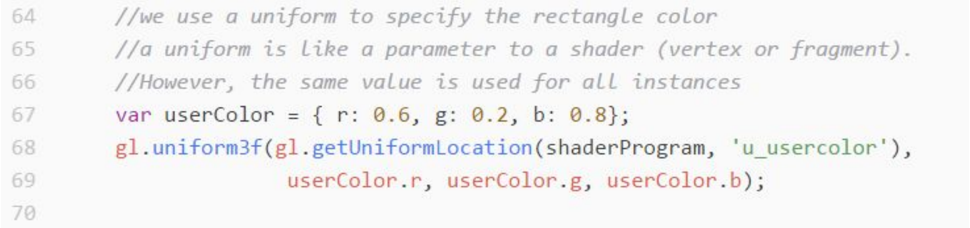
Shader Parameter

- 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

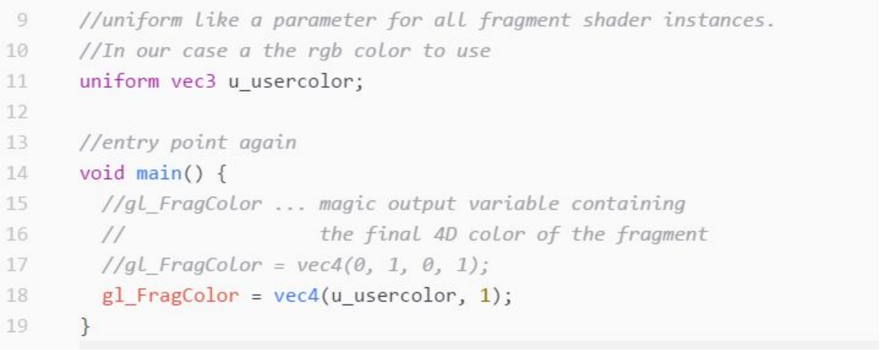
Recap: uniforms are parameters for a shader.
In contrast to attributes, they remain the same for each vertex.

main.js → render() after gl.useProgram



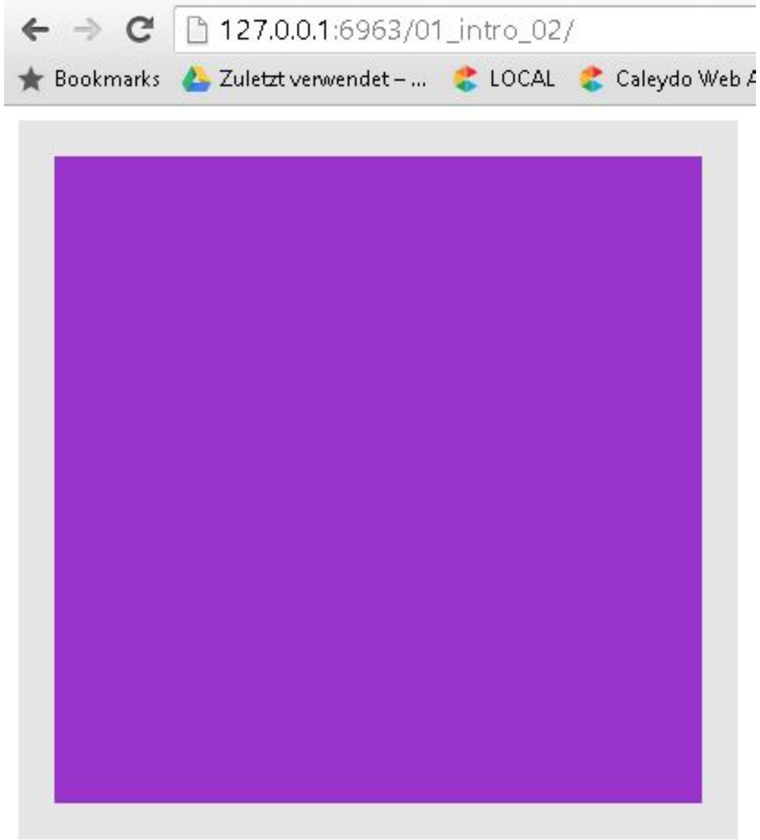
```
//we use a uniform to specify the rectangle color
//a uniform is like a parameter to a shader (vertex or fragment).
//However, the same value is used for all instances
var userColor = { r: 0.6, g: 0.2, b: 0.8};
gl.uniform3f(gl.getUniformLocation(shaderProgram, 'u_usercolor'),
          userColor.r,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

```

simple.fs.glsl



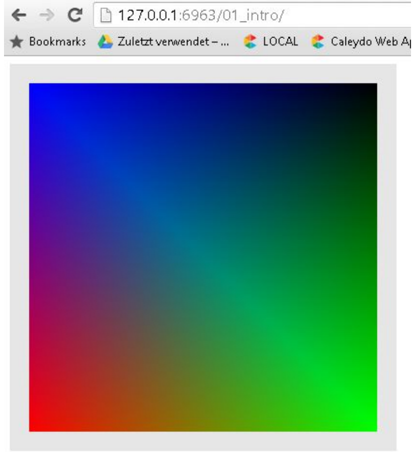
```
//uniform like a parameter for all fragment shader instances.
//In our case a the rgb color to use
uniform vec3 u_usercolor;

//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);
}
```

4. Specifying Color via Uniform



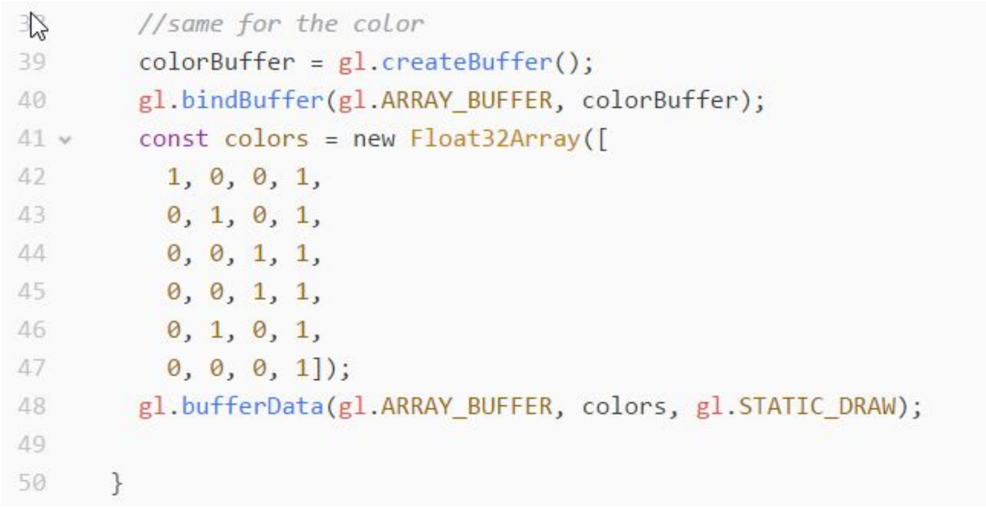
What about a color gradient, i.e., a color per vertex?

→ another buffer for the colors instead of a uniform

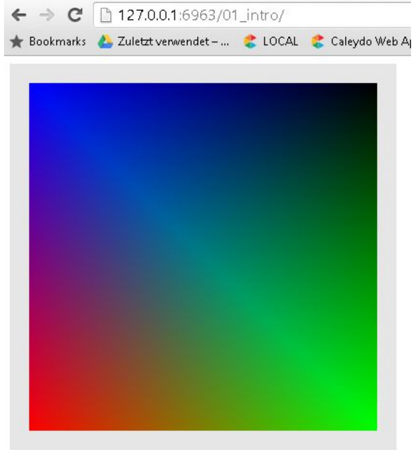


5. Specifying Color per Vertex

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);

  }
```



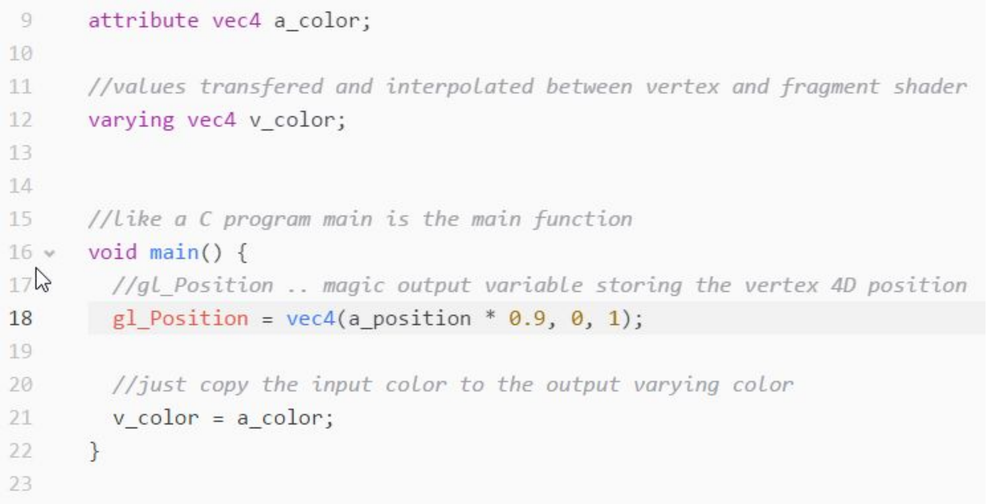
main.js → render()

```

    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

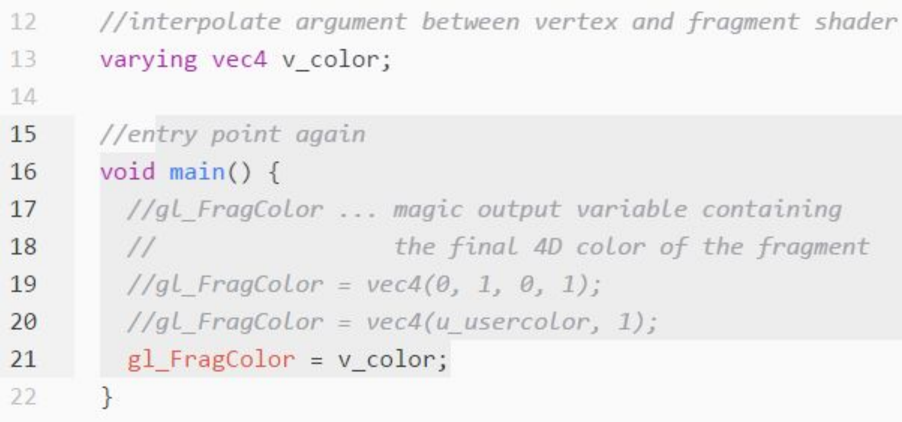
Recap: attributes are data for each vertex.
varyings are parameters between shader stages that are interpolated during rasterization

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;
}

```

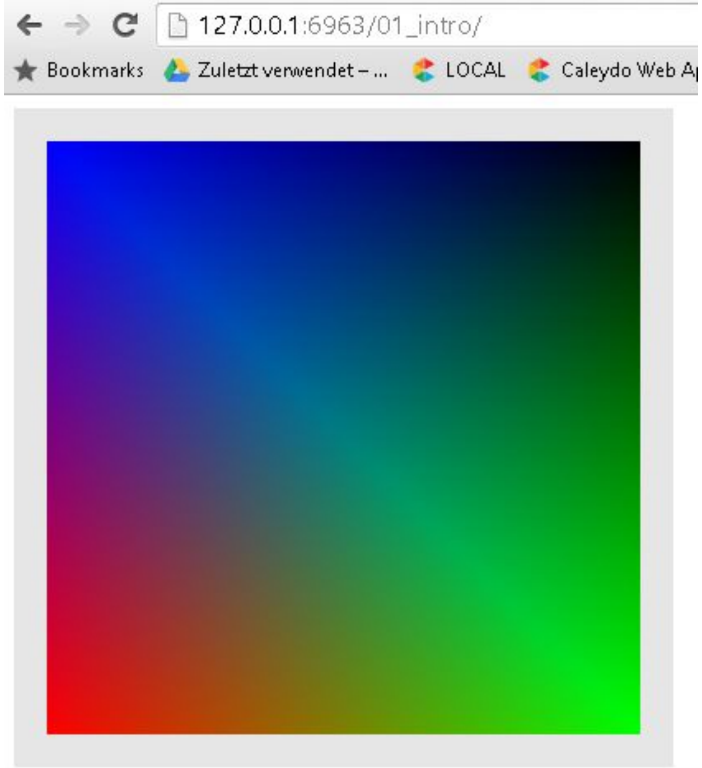
simple.fs.glsl



```
//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