Lecture #4

GPU Rendering

Computer Graphics
Winter Term 2016/17

Marc Stamminger

GPUs

- Rendering is very compute intensive
  → Full-HD, 60 frames per second → 120 mio pixels to be set per second !
  → Today's scene have easily 10-100 millions of triangles
- But it can also be easily parallelized
  → per triangle, per vertex, per pixel, …

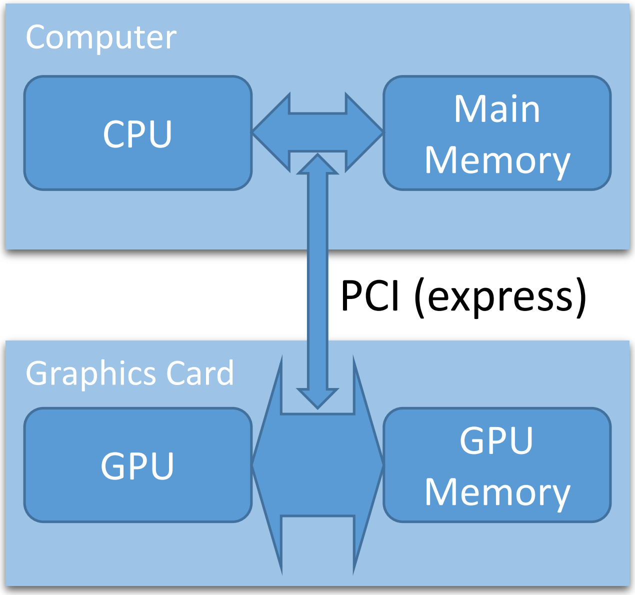
- GPUs: Graphics Processing Units
  - a processor like a CPU
  - but with many cores (hundreds or thousands)
  - and with its own memory

- APIs:
  - OpenGL, DirectX, …
  - command-based
  - In this lecture: OpenGL / WebGL

NVIDIA Titan GTX
3.000 cores

GPUs

- Data Transfer
  - CPU – Main Memory: fast
  - GPU – GPU Memory: very fast
  - CPU – GPU: slow



- **We first have to submit the scene data to GPU memory**
- **And then we can start the render process**

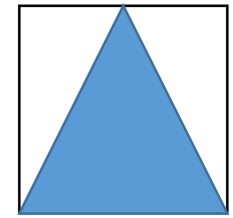
Minimal WebGL program

```
// get WebGL context
var gl = canvas.getContext(“webgl”);

// triangle coordinates
var v = [-1,-1, 1,-1, 0,0];          1

-> upload v to GPU Memory
gl.???                               2

-> setup shaders
gl.???                               3

-> render
gl.???                               4
```

Triangle Data 1

- One single triangle:

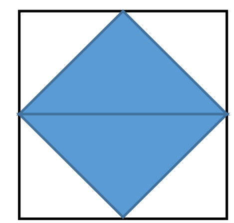
```
var v = [-1,-1, 1,-1, 0,1];
```



→ OpenGL coordinates go from -1 to 1 ! (for now)

- Two triangles:

```
var v = [-1,0, 1,0, 0,1, -1,0, 1,0, 0,-1];
```



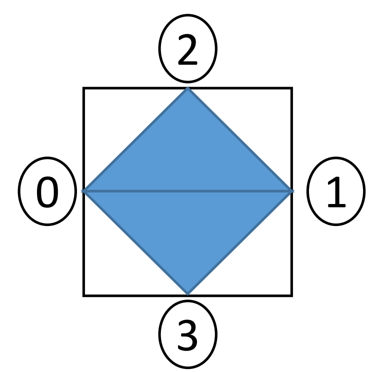
→ inefficient, because two vertices are used twice

- Indexed Face Set data structure:

```
var v = [-1,0, 1,0, 0,1, 0,-1];
var i = [0,1,2, 0,3,1];
```

vertex coordinates

vertex indices per triangle



- → Needed for large scenes with many triangles (millions)

Upload Data to GPU Memory

2

- Upload vertex array to ARRAY_BUFFER

```
var vbo = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vbo);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(v), gl.STATIC_DRAW);
```

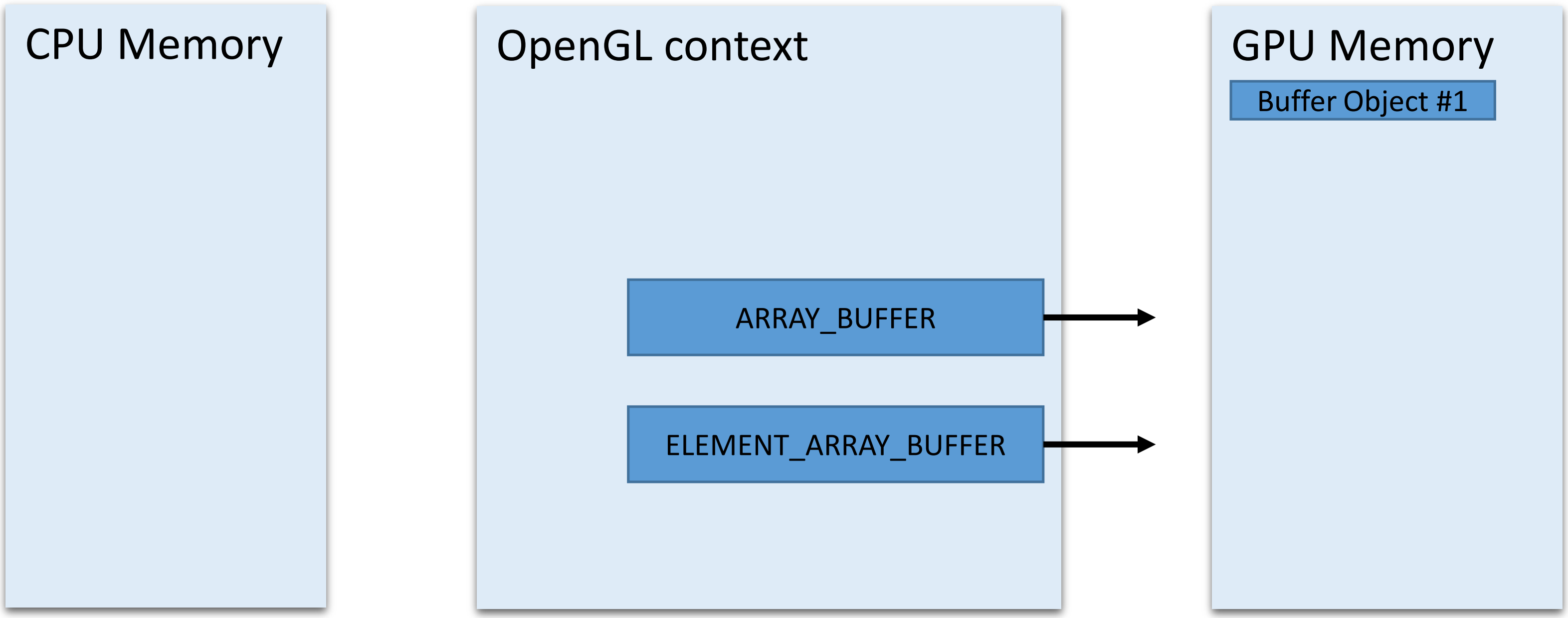
- memory chunks on the GPU are referred to as “buffer objects”
- `gl.createBuffer()` creates such an object
- one such buffer object contains the current vertex array data
- setting this `ARRAY_BUFFER` is called “binding the buffer”: `gl.bindBuffer(…)`
- data can be copied into the currently bound buffer using `gl.bufferData(…)`
- JavaScript arrays first have to be converted to a `TypedArray`, in this example `Float32Array`

Upload Data to GPU Memory

2

```
var vbo = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vbo);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(v), gl.STATIC_DRAW);
```

- `createBuffer()` creates a GPU memory object
- first, this is only a handle, no memory is allocated yet

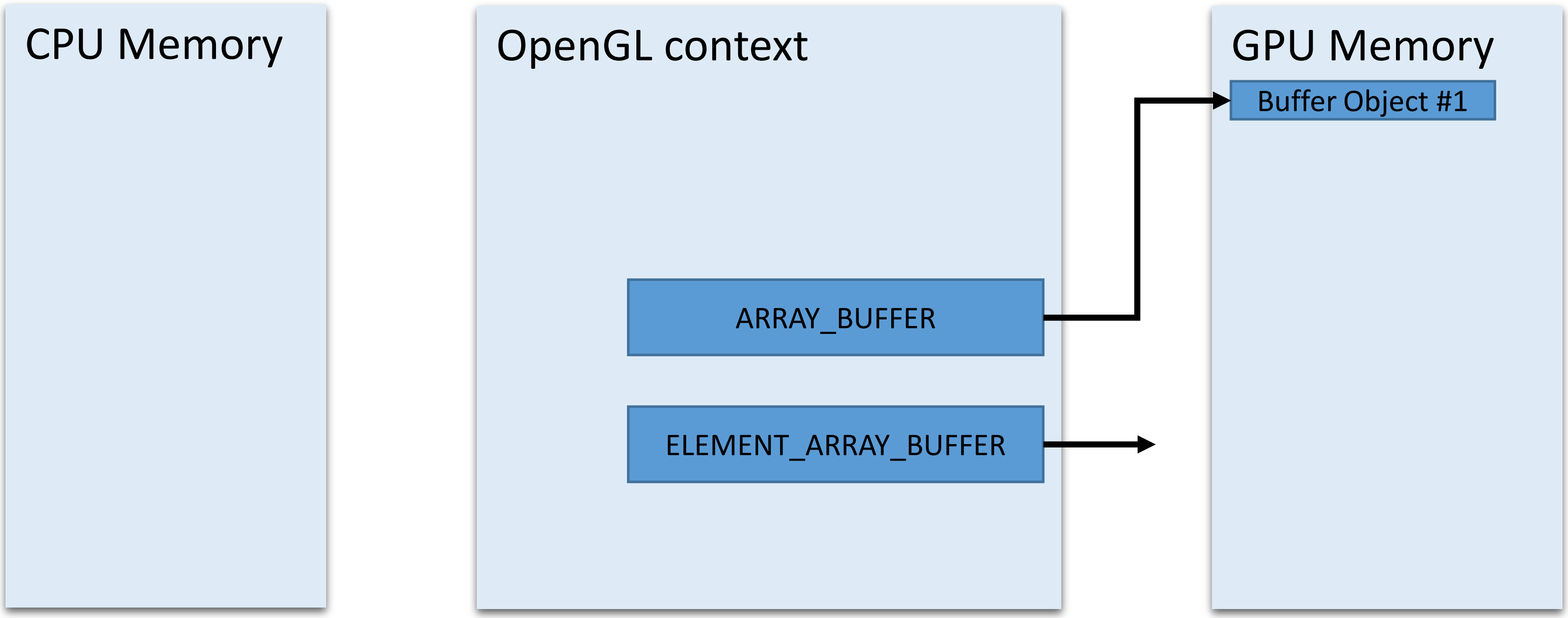


Upload Data to GPU Memory

2

```
var vbo = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vbo);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(v), gl.STATIC_DRAW);
```

- **`bindBuffer(…)`** connects this buffer with OpenGL's **array buffer**
- later render commands get their data from the currently bound array buffer

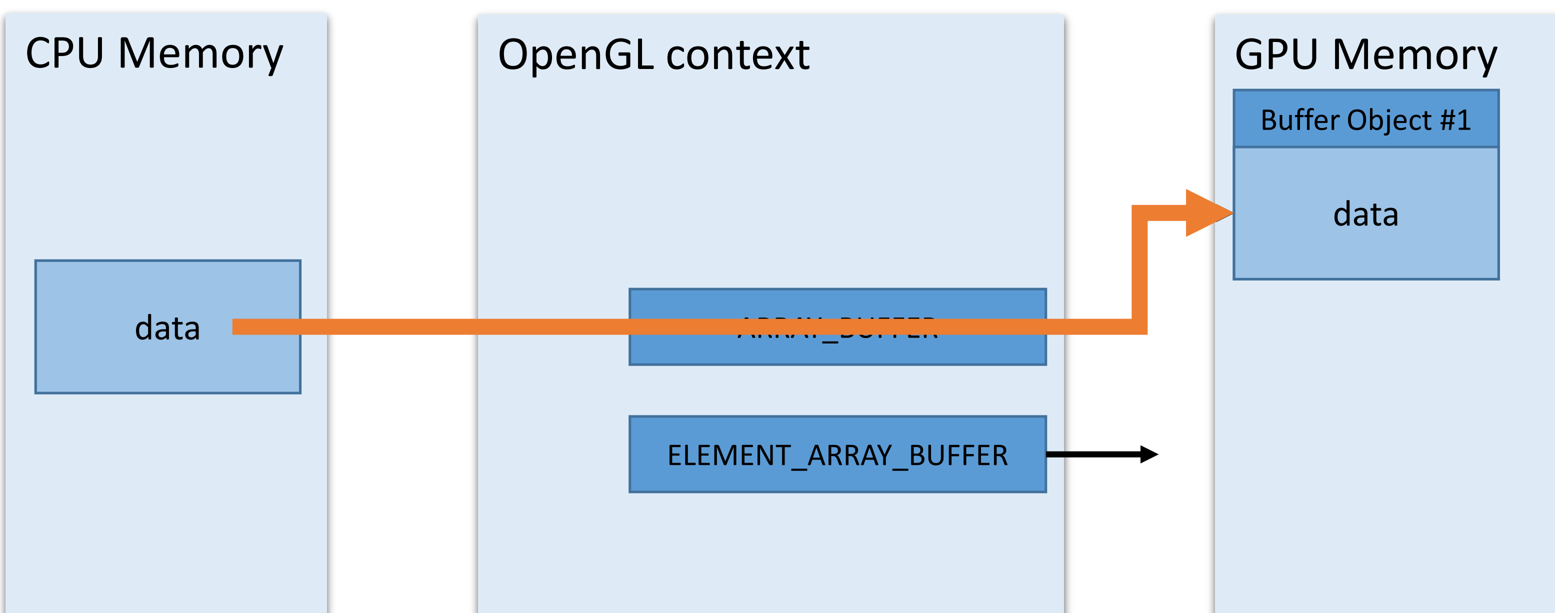


Upload Data to GPU Memory

2

```
var vbo = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vbo);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(v), gl.STATIC_DRAW);
```

- **`bufferData(…)`** loads data to GPU memory
- in this step, memory is allocated in GPU memory



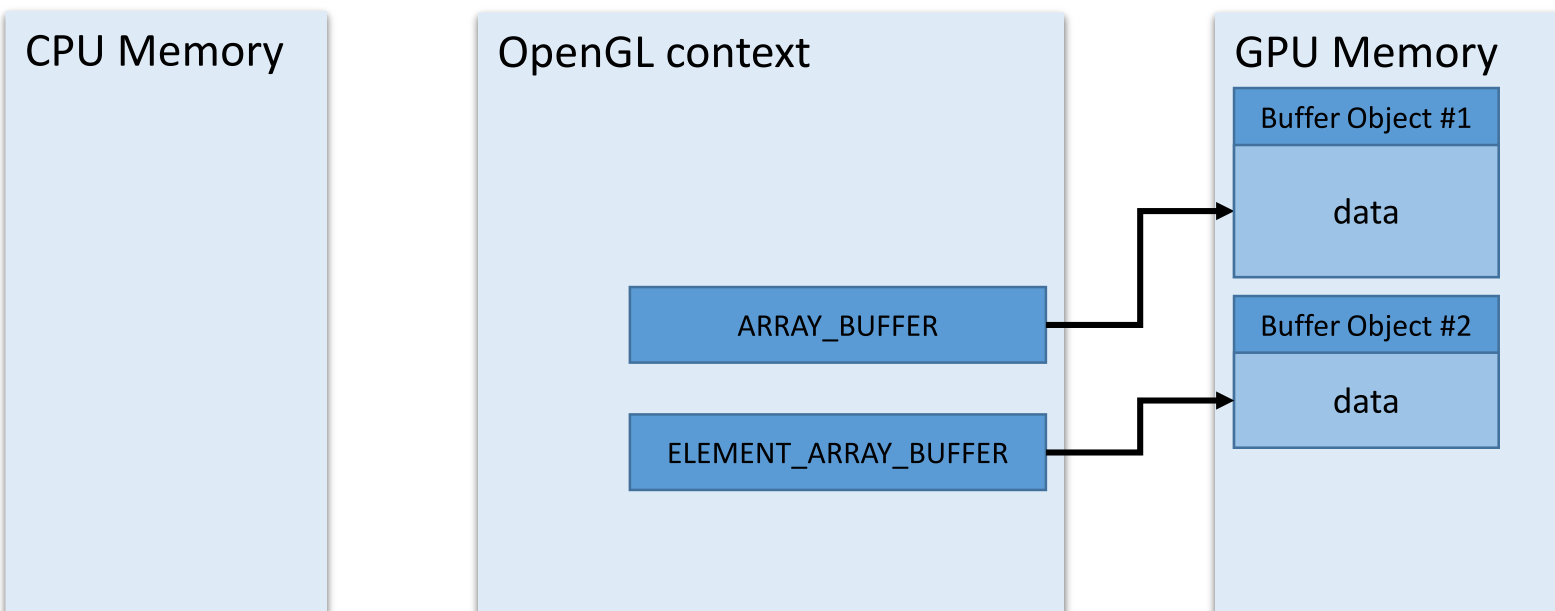
Upload Data to GPU Memory

2

- We do the same with the **ELEMENT_ARRAY_BUFFER**

```
var ibo = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, ibo);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(i), gl.STATIC_DRAW);
```

- index data needs to be bound to **ELEMENT_ARRAY_BUFFER**
- index data must be an unsigned integer array

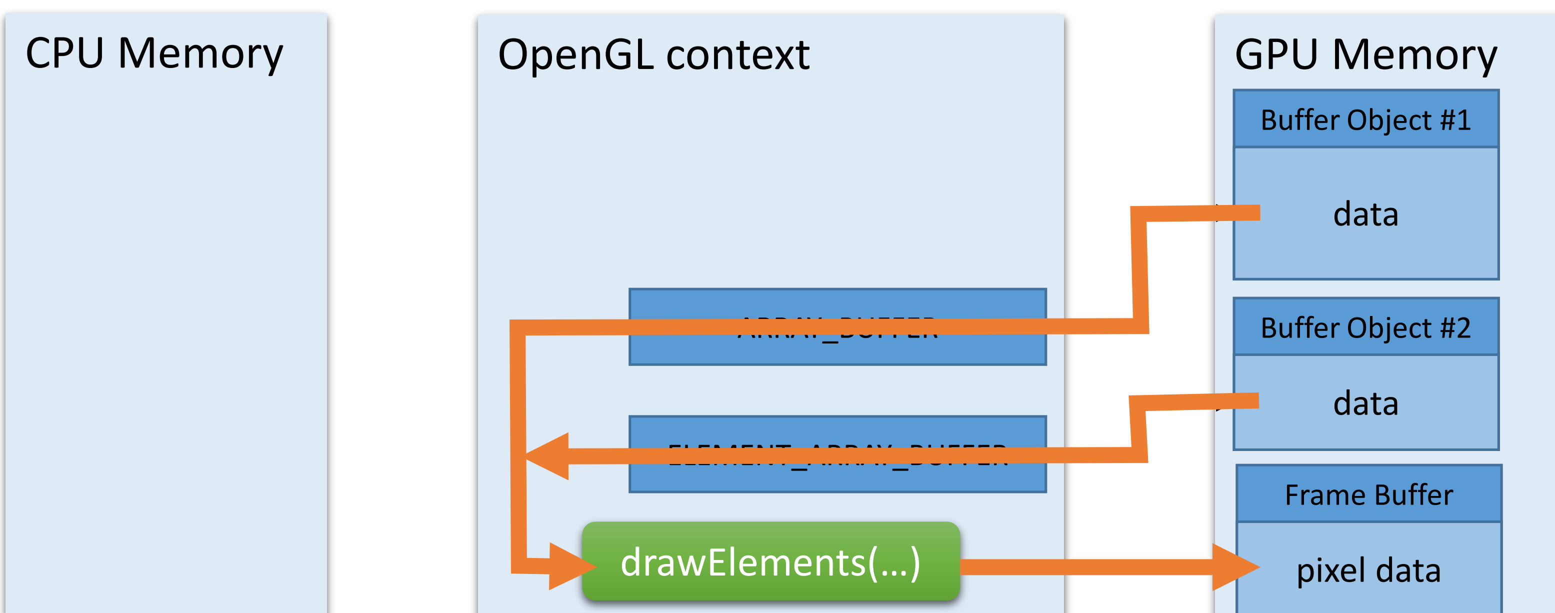


Upload Data to GPU Memory

- Finally, we will render the triangles

```
gl.drawElements(...)
```

- Vertex data will be fetched from the ARRAY_BUFFER
- and index data from the ELEMENT_ARRAY_BUFFER



Setup Shaders 3

- Last, we have to define a **Vertex Shader** and a **Fragment Shader**
- A shader is a little program written in a C-like language **GLSL**
- The vertex shader is called for every vertex before rasterization
  - here we can transform the vertices (e.g. to animate objects, to zoom, …)
    → next lecture “Transformations”
  - and we can add attributes to the vertices that are interpolated during rasterization
    → Gouraud Shading from previous lecture
- The fragment shader is called for every rasterized pixel
  - here we compute the color of a pixel
    → later lectures “Lighting”, “Texturing”, …

Setup Shaders

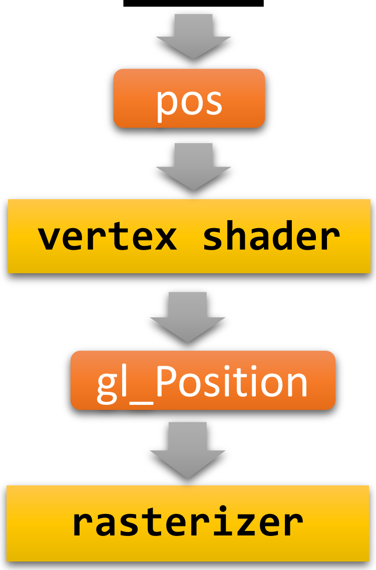
3

- Vertex Shader

```
attribute vec2 pos;

void main(void) {
    gl_Position = vec4(pos, 0.0, 1.0);
}
```

- executed for every vertex

```
var v = [-1,0, 1,0, 0,1, -1,0, 1,0, 0,-1];
```



Setup Shaders

3

- Vertex Shader

```
attribute vec2 pos;

void main(void) {
    gl_Position = vec4(pos, 0.0, 1.0);
}
```

- Bind attributes from **`ARRAY_BUFFER`** to vertex shader attributes:

```
var attrPos = gl.getAttribLocation(shaderProgram, “pos");
gl.enableVertexAttribArray(attrPos);
gl.vertexAttribPointer(attrPos,2,gl.FLOAT,false,8,0);
```

2D-attribute

Offset between successive vertices

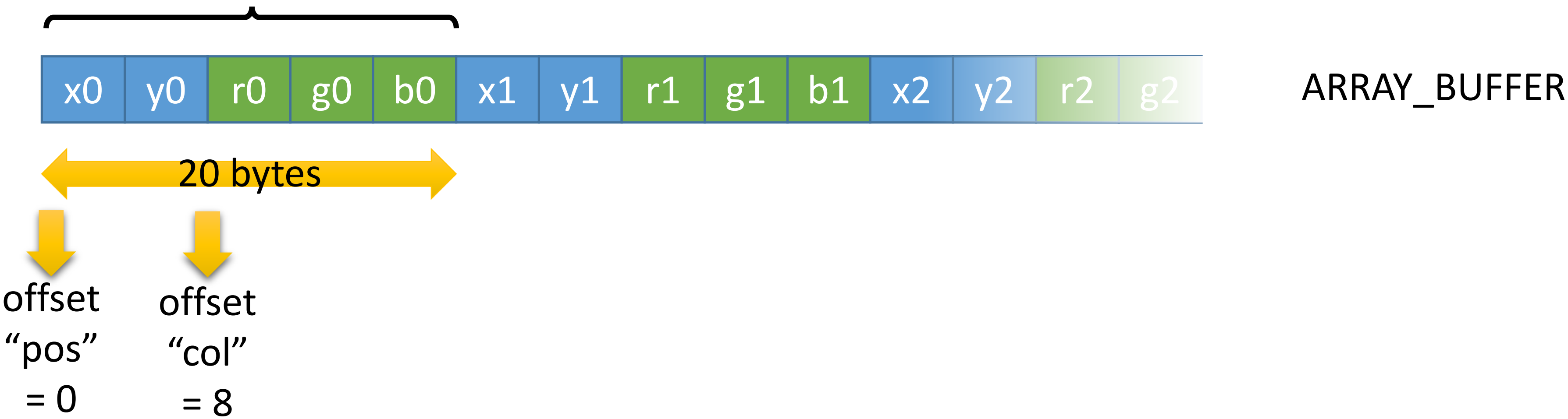
Start offset of first vertex

Setup Shaders

3

- Interleaved attributes

vertex 0
attribute 1 “pos”: (x0,y0)
attribute 2 “col”: (r0,g0,b0)




```
var attrPos = gl.getAttribLocation(shaderProgram, “pos");
gl.enableVertexAttribArray(attrPos);
gl.vertexAttribPointer(attrPos,2,gl.FLOAT,false,20,0);

var attrCol = gl.getAttribLocation(shaderProgram, “col");
gl.enableVertexAttribArray(attrCol);
gl.vertexAttribPointer(attrCol,3,gl.FLOAT,false,20,8);
```

Setup Shaders 3

- A more complicated shader

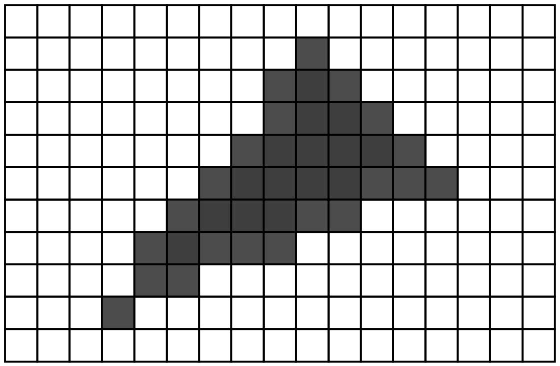
```
uniform vec2 offset;
uniform float zoom;
attribute vec2 pos;

void main(void) {
    gl_Position = vec4((pos+offset)*zoom, 0.0, 1.0);
}
```

- **`uniform`**: variables that can be set globally
- **`vec2,vec3,vec4,mat2,mat3,mat4`**:
  - special types for graphics computing
  - includes many functions, e.g. for dot product, matrix multiplication etc.
  - see [documentation of GLSL]()

Setup Shaders

3

- Next, the triangles is rasterized



- for every pixel, a fragment shader is called (aka pixel shader):

```
precision highp float; // set precision of float computations

void main(void){
    gl_FragColor = vec4(1,0,0,1);
}
```

- finally, pixel is set to **`gl_FragColor`**

Setup Shaders 3

- vertex shaders can output additional values, which are then interpolated during rasterization (→ Gouraud Shading)
- these become an input of the fragment shader:

```
// vertex shader
attribute vec2 pos;        input from ARRAY_BUFFER
attribute vec3 col;
varying vec3 c;            output sent to rasterizer

void main(void) {
    gl_Position = vec4((pos+offset)*zoom, 0.0, 1.0);
    c = col;
}
```

```
// fragment shader
precision highp float;
varying vec3 c;            input coming from rasterizer

void main(void) {
    gl_FragColor = vec4(c,1);
}
```

Setup Shaders

3

- Vertex shader and Fragment shaders come in pairs, interacting with varyings
- Shaders are compiled using GL commands:

```
var vertexShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertexShader, “shaderCodeAsString”);
gl.compileShader(vertexShader);
...
var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragmentShader, “shaderCodeAsString”);
gl.compileShader(fragmentShader);
```

- and must then be linked to a **shader program:**

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

- and finally be activated:

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

Render

4

- Finally, we can make a render call:

```
gl.drawElements(gl.TRIANGLES, 6, gl.UNSIGNED_SHORT, 0);
```

three successive vertices in the index buffer form a triangle

process 6 vertices = 2 triangles

index type

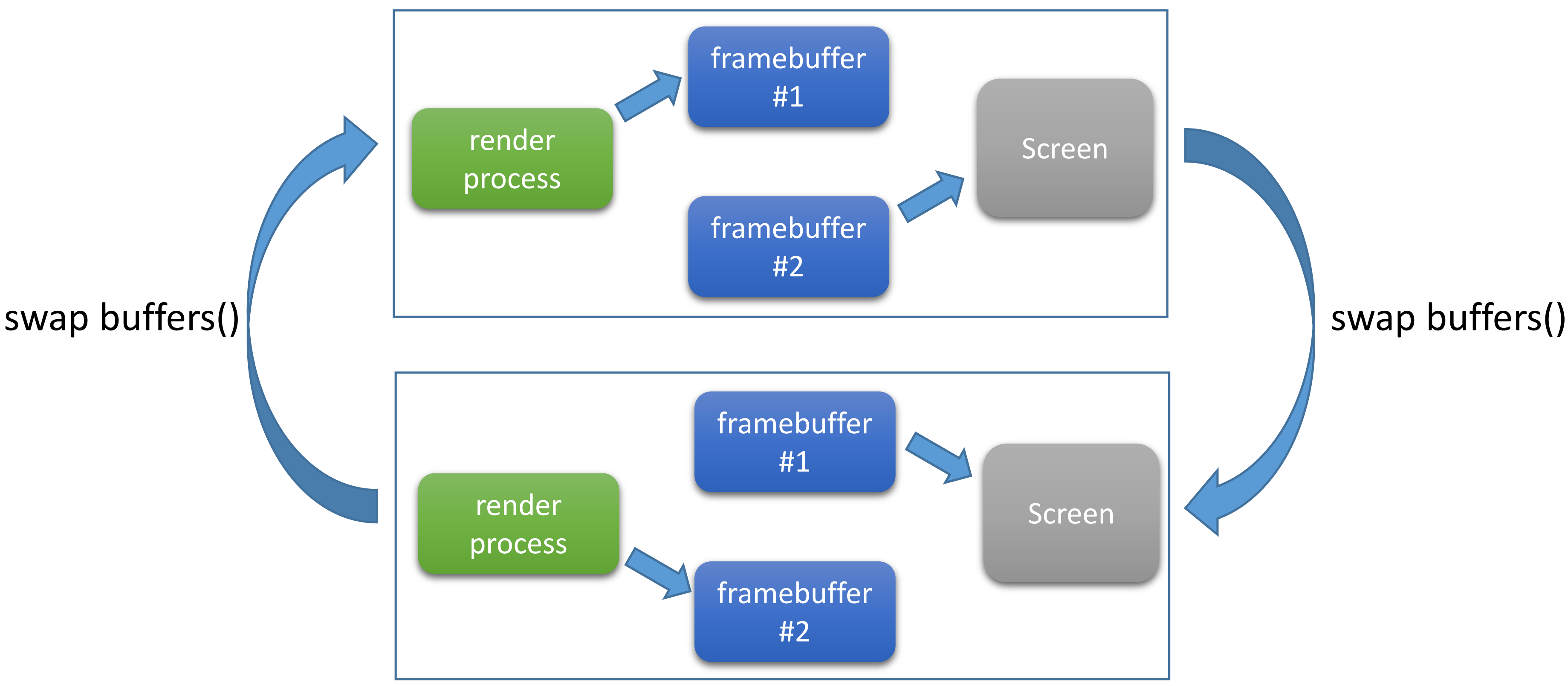
start position in index buffer

Let’s play

- WebGL

Render Loop

- To render an animation, we continuously re-render the scene
- Usually two frame buffers available (**double buffering**)
  - One is displayed
  - The other one is being rendered into
  - When new frame has been rendered, buffers are **swapped**



Render Loop

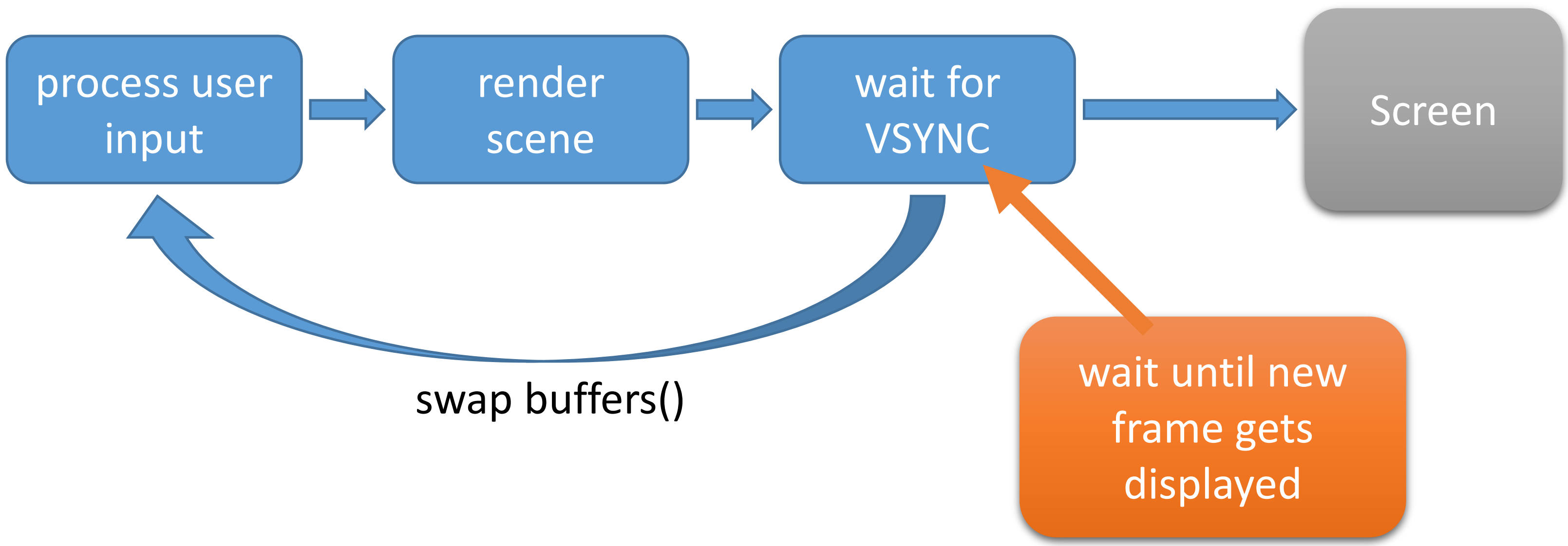
- Typically the swap is delayed until the next **VSYNC**, i.e. the moment the monitor starts a new frame (usually every 1/60th second)
- Otherwise: waste of rendering power, images are torn → screen tearing

Wikipedia

rendering at 120 Hz,
display with 60 Hz

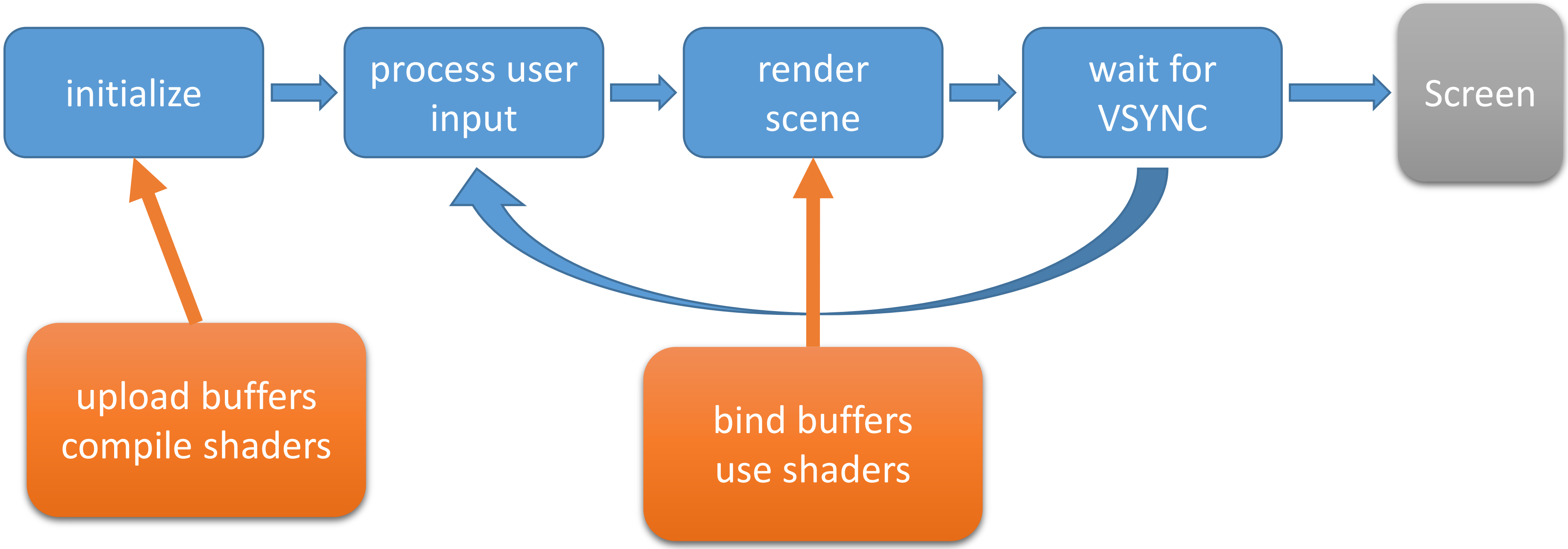
Render Loop

- To render an animation, we continuously re-render the scene
- The screen displays a new image every 1/60th second
- → **Render Loop**



Render Loop

- Memory upload, shader compilation and linking etc. are **time expensive**
- **before the render loop**
  - upload buffers (multiple buffers possible)
  - compile and link shaders (multiple shaders / shader programs possible)
- **In the render loop**
  - buffers can be quickly activated by **`bindBuffer()`**
  - shader programs are activated using **`useProgram()`**



Render Loop

- Multiple buffers can be set, as well as multiple shader programs
- **`initialize()`** (before render loop)

```
var vboObject1 = gl.createBuffer();  // Object 1
gl.bindBuffer(gl.ARRAY_BUFFER, vboObject1);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vObject1), gl.STATIC_DRAW);

var vboObject2 = gl.createBuffer(); // Object 2
gl.bindBuffer(gl.ARRAY_BUFFER, vboObject2);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vObject2), gl.STATIC_DRAW);
```

- **`render()`** (within render loop)

```
gl.bindBuffer(gl.ARRAY_BUFFER, vboObject1);
gl.drawElements(gl.TRIANGLES, nVert1, gl.UNSIGNED_SHORT, 0);

gl.bindBuffer(gl.ARRAY_BUFFER, vboObject2);
gl.drawElements(gl.TRIANGLES, nVert2, gl.UNSIGNED_SHORT, 0);
```

Render Loop

- Same with shaders
- **`initialize()`**

```
var vertexShader1 = gl.createShader(gl.VERTEX_SHADER);
var fragmentShader1 = gl.createShader(gl.FRAGMENT_SHADER);
...
gl.linkProgram(shaderProgram1);
...
var vertexShader2 = gl.createShader(gl.VERTEX_SHADER);
var fragmentShader2 = gl.createShader(gl.FRAGMENT_SHADER);
...
gl.linkProgram(shaderProgram2);
```

- **`render():`** object #1 with shader #1, object #2 with shader #2

```
gl.useProgram(shaderProgram1)
gl.bindBuffer(gl.ARRAY_BUFFER, vboObject1);
gl.drawElements(gl.TRIANGLES, nVert1, gl.UNSIGNED_SHORT, 0);

gl.useProgram(shaderProgram2);
gl.bindBuffer(gl.ARRAY_BUFFER, vboObject2);
gl.drawElements(gl.TRIANGLES, nVert2, gl.UNSIGNED_SHORT, 0);
```

Next Week

- Transformations
  - Translations, Rotations, Scalings, …
  - usually happen in the vertex shader