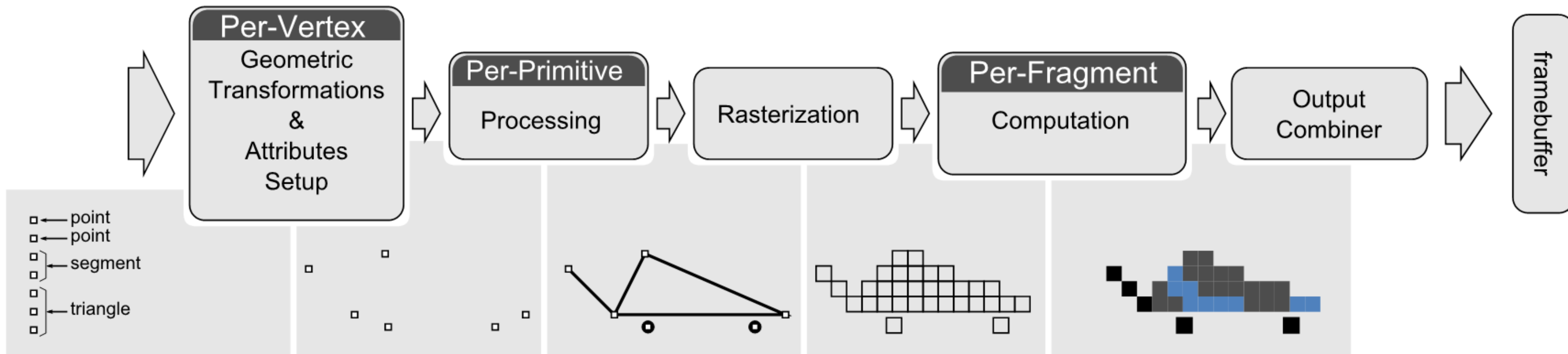
Rasterization-based pipeline



Rasterization-based rendering

- Input: set of vertices and its associated attributes
- Algorithm goes through several phases:

1. Per-vertex transformations and attributes setup
2. Primitive processing
3. Rasterization
4. Per-fragment computation

Advantages of ray tracing

- Conceptually simple algorithm
- Takes into account GLOBAL effects
- More realistic rendering of lighting effects

Advantages of rasterization

- More easily parallelizable
- More controllable complexity
- Better-suited for graphics cards
- Better treatment of anti-aliasing (more later on)

First Steps

Interactive Graphics Course

Prof. Marco Schaerf

Dept. of Computer, Systems and Management Science (DIAG)

Sapienza University of Rome

marco.schaerf@uniroma1.it

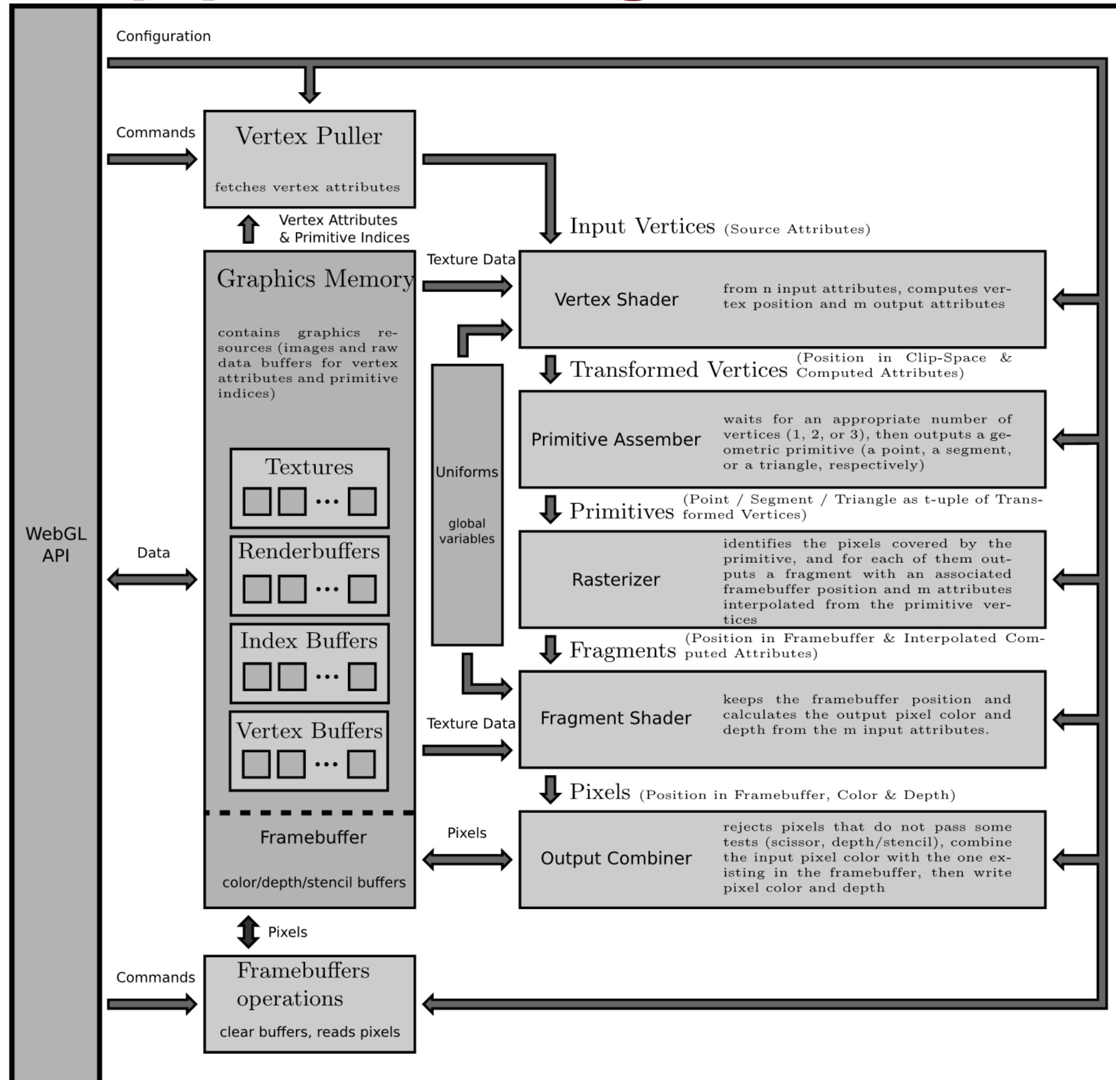
Outline

- Goal: Create an HTML page with WebGL JavaScript code in it
- To accomplish it we need to (briefly) introduce:
  - OpenGL and WebGL API
  - WebGL rasterization-based pipeline
  - Programming the rendering pipeline
  - WebGL Supporting Libraries (in particular, SpiderGL)
  - eNVyMyCar framework and Class NVMC
-

Application Programming Interface

- In order to write 3D graphic programs we will use a predefined and standardized 3D API WebGL
- WebGL is based on OpenGL, an open standard widely used for real-time (interactive) graphics and now mantained by the Khronos group.
- Almost all of the graphic cards support and accelerate the OpenGL API (and indirectly also WebGL)
- WebGL runs in a browser using JavaScript, it does not require ad-hoc programming tools.
- Most browsers support WebGL, including Firefox, Chrome, Safari and Opera

WebGL pipeline: Image



WebGL pipeline

- Vertex Puller (VP)
- Vertex Shader (VS)
  - Written in GLSL (OpenGL Shading Language)
- Primitive Assembler (PA)
- Rasterizer (RS)
- Fragment Shader (FS)
  - Written in GLSL (OpenGL Shading Language)
- Output Combiner (OC)
- FrameBuffer Operations (FO)

First Rendering

- We need to:

1. Define the HTML page containing our program
2. Initialize WebGL
3. Define what to draw
4. Define how to draw
5. Perform the actual drawing

HTML Page

```
<html>
 <head>
  <script type="text/javascript">
  </script>
 </head>
 <body>
  <canvas
         id     = "OUTPUT-CANVAS"
         width  = "500px"
         height = "500px"
         style  = "border: 1px solid black"
  ></canvas>
 </body>
</html>
```

Skeleton Code

```
<script type="text/javascript">
        // global variables
        function setupWebGL() {}
        function setupWhatToDraw() {}
        function setupHowToDraw() {}
        function draw() {}
        function helloDraw() {
                setupWebGL();
                setupWhatToDraw();
                setupHowToDraw();
                draw();
        }
        window.onload = helloDraw;
</script>
```

Initialize WebGL 1/2

- WebGL is a **state machine** where the outcome of every operation depends on the internal state of the **machine.**
- In OpenGL the state (called the rendering context) is implicit, while in WebGL is stored in a JavaScript object **WebGLRenderingContext**, which is tied to an **HTMLCanvasElement** used as an output

```
var gl = null;
function setupWebGL() {
        var canvas = document.getElementById("OUTPUT-CANVAS");
        gl = canvas.getContext("experimental-webgl");
        }
```

Initialize WebGL 2/2

- Associate the canvas variable to the HTML canvas defined in the html source of the page
- Canvases contain a 4-channels RGBA framebuffer and a depth buffer. The framebuffer Alpha channel is used as a transparency value
- The method **getContex**t returns the WebGL context, **webgl** is the string denoting it in most browsers (in some cases you need to use **experimental-webg**l
- Note that the name used for all (and only) the WebGL variables will start with **g**l

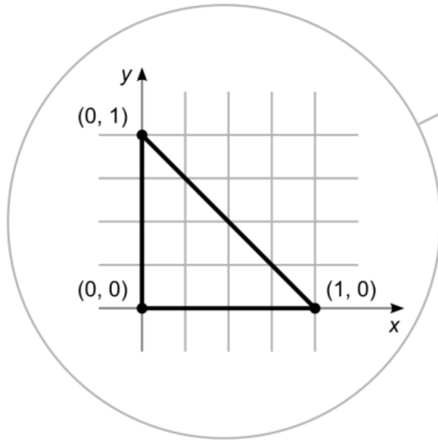
Define what to draw 1/4

- WebGL is designed to take advantage of graphic cards
- Most data structures MUST be mirrored with corresponding ones provided by the API
- As an example, we define a 2D triangle by using real coordinatse from -1 to 1

```
1 var triangle = {
2   vertexPositions : [
3     [0.0, 0.0],  // 1st vertex
4     [1.0, 0.0],  // 2nd vertex
5     [0.0, 1.0]   // 3rd vertex
6   ]
7 };
```

LISTING 2.4: A triangle in JavaScript.

Define what to draw 2/4



① Native Array - System Memory

```
var positions = [ 0, 0,  1, 0,  0, 1 ];
```

Create a typed copy of the native JavaScript array

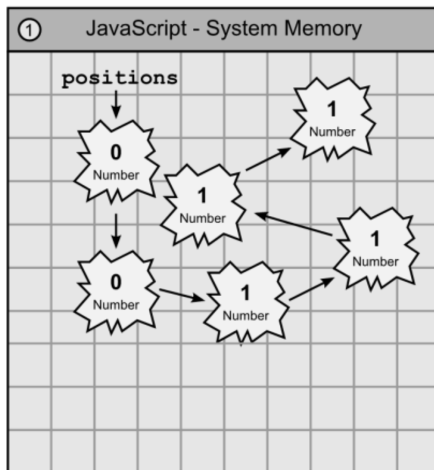
② Typed Array - System Memory

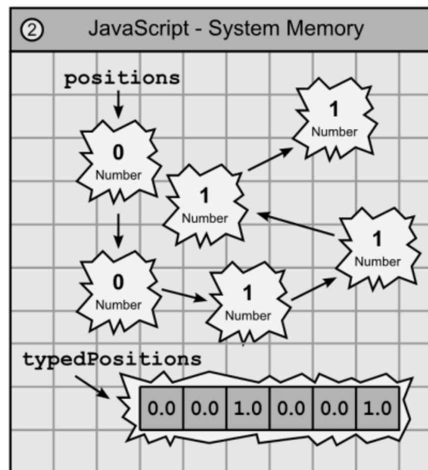
```
var typedPositions = new Float32Array(positions);
```

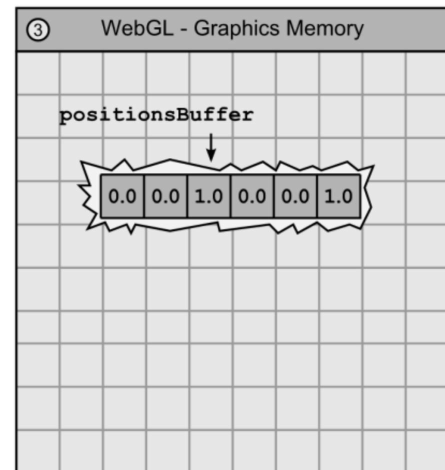
Create and fill a WebGLBuffer transferring typed data to graphics memory

③ WebGLBuffer - Graphics Memory

```
var positionsBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, positionsBuffer);
gl.bufferData(gl.ARRAY_BUFFER, typedPositions, gl.STATIC_DRAW);
```







Define what to draw 3/4

- More efficient form allows for less redundacy, but is less intuitive and structured

```
var positions = [
        0.0, 0.0,  // 1st vertex
        1.0, 0.0,  // 2nd vertex
        0.0, 1.0   // 3rd vertex
];
```

Define what to draw 4/4

```
var typedPositions = new Float32Array(positions);
```

- Or alternatively we could have filled the native array directly

```
var typedPositions = new Float32Array(6); // 6 floats
typedPositions[0] = 0.0; typedPositions[1] = 0.0;
typedPositions[2] = 1.0; typedPositions[3] = 0.0;
typedPositions[4] = 0.0; typedPositions[5] = 1.0;
```

- We mirror this data and encapsulate it in WebGL object

```
var positionsBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, positionsBuffer);
gl.bufferData(gl.ARRAY_BUFFER, typedPositions, gl.STATIC_DRAW);
```

- STATIC_DRAW specifies that we expect this object to be accessed many times but rarely changed