Introduction to Computer Graphics (CS360A)

Instructor: Soumya Dutta

Department of Computer Science and Engineering

Indian Institute of Technology Kanpur (IITK)

email: soumyad@cse.iitk.ac.in

Graphics API

- Goal of Graphics APIs are to perform graphics rendering and enable developers to use Graphics Processing Units (GPU)
- There are several popular graphics APIs
  - OpenGL
  - DirectX (Microsoft)
  - Vulkan
  - Metal (Apple)
- Web-based graphics APIs
  - WebGL (This course)
  - WebGPU (Relatively new, under development)

Graphics API

- **OpenGL:**
  - Cross-language, cross-platform application programming interface (API)
  - Used for rendering 2D and 3D graphics in a graphics processing unit (GPU), to achieve hardware-acceleration

- **OpenGL ES:**
  - OpenGL for Embedded Systems (OpenGL ES or GLES)
  - A subset of the OpenGL API
  - Used in mobile devices

Wikipedia.org

Graphics API

- **Direct3D (DirectX):**
  - A graphics API for Microsoft Windows
  - Often used to render three-dimensional graphics in applications where performance is important, such as games
  - Direct3D uses hardware acceleration if it is available on the graphics card
  - DirectXBox → Xbox (game console)

Wikipedia.org

Graphics API

- **Vulkan:**
  - A low-overhead, cross-platform API, open standard for 3D graphics and computing
  - Targets high-performance real-time 3D-graphics applications
  - Intended to offer higher performance and more efficient CPU and GPU usage compared to the OpenGL and DirectX

Wikipedia.org

Graphics API

- **Metal:**
  - Metal powers hardware-accelerated graphics on Apple platforms
  - Provides a low-overhead API, rich shading language, tight integration between graphics and compute
  - Offers an unparalleled suite of GPU profiling and debugging tools

https://developer.apple.com/metal/

Web-based Graphics API

- **WebGL:**
  - A JavaScript API for rendering interactive 2D and 3D graphics within any compatible web browser
  - Fully integrated with other web standards, allowing GPU-accelerated usage of physics and image processing and effects as part of the web page canvas.
  - WebGL 1.0 is based on OpenGL ES 2.0
  - WebGL 2.0 is based on OpenGL ES 3.0
  - https://www.khronos.org/webgl/

- **WebGPU:**
  - A potential web standard and JavaScript API for accelerated graphics and compute, aiming to provide "modern 3D graphics and computation capabilities"

Wikipedia.org

WebGL v2.0

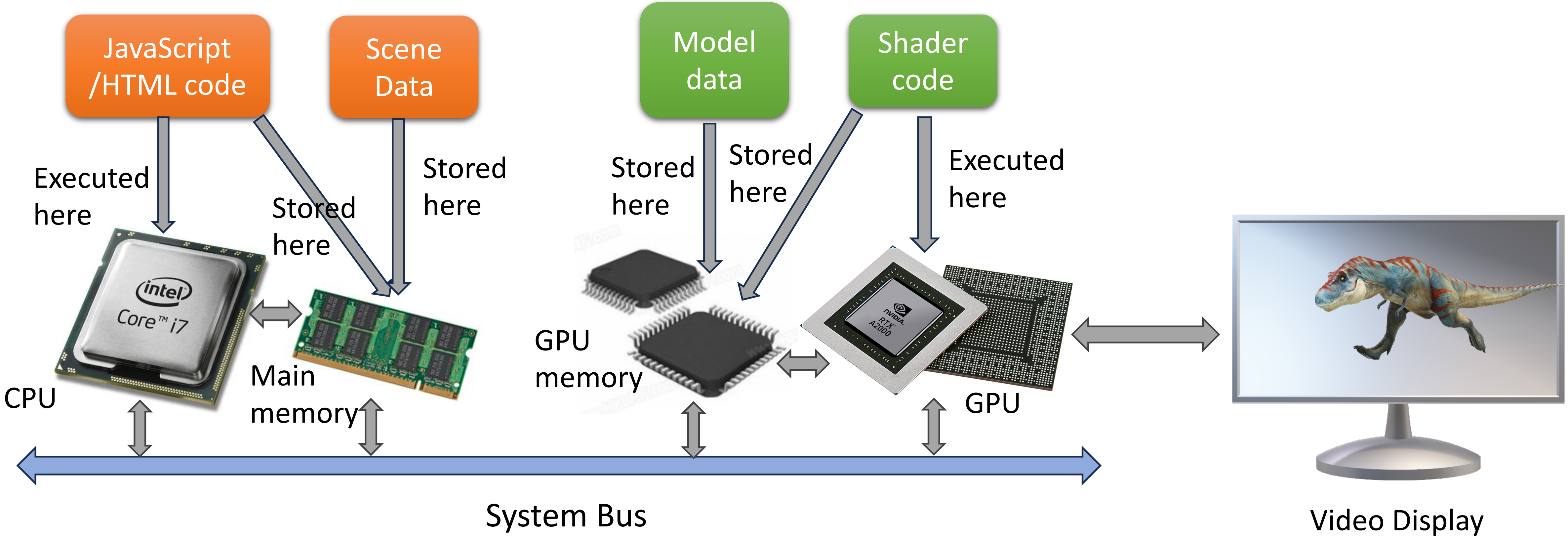
WebGL: Web Graphics Library

- We are going to use WebGL in this course
  - Works seamlessly in modern browsers
  - Easy code building
  - No compilation issues
  - Cross platform development environment

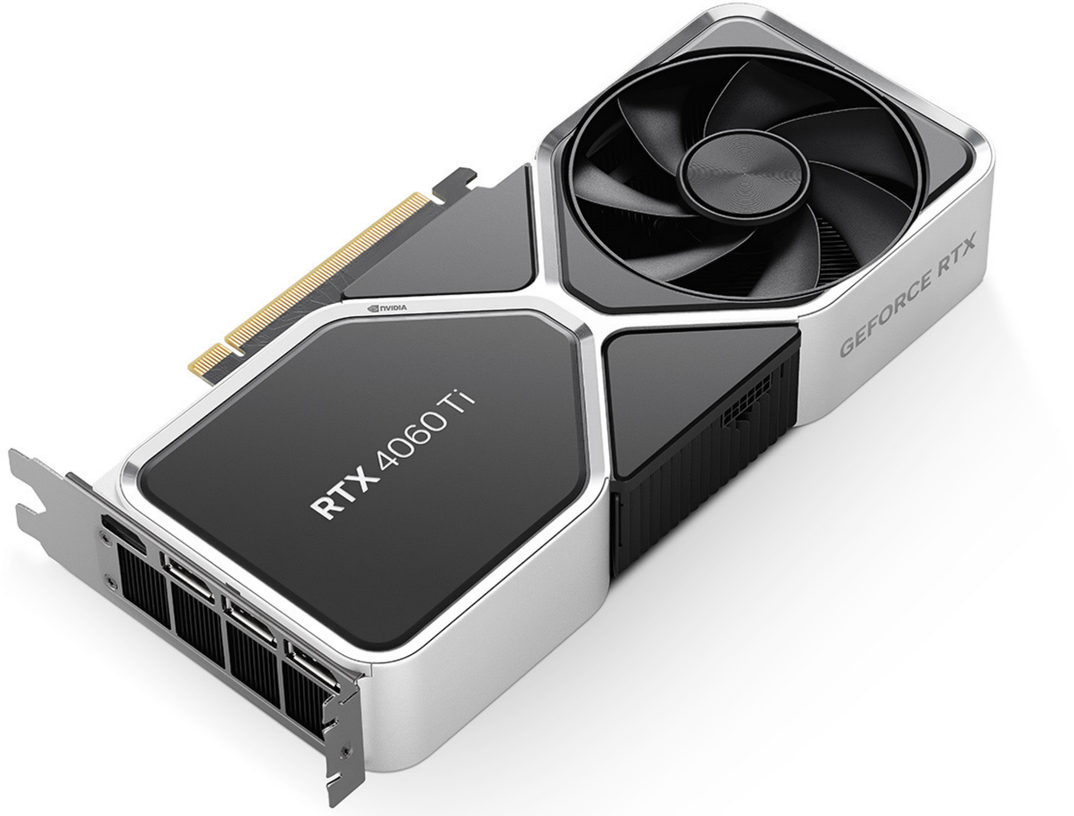
A Typical WebGL Program

- A HTML (Hypertext Markup Language) description of the web page
- A CSS (Cascading Style Sheet) that describes how each element of the HTML description is formatted (Optional)
- A HTML canvas element in the web page that provides a rectangular area in which 3D computer graphics can be rendered
- Graphical data that defines the 3D objects to be rendered
- JavaScript programs that load your graphical data, configure your graphical data, render your graphical data, and code that responds to user events
- OpenGL Shader programs (GLSL) that perform critical parts of the graphical rendering

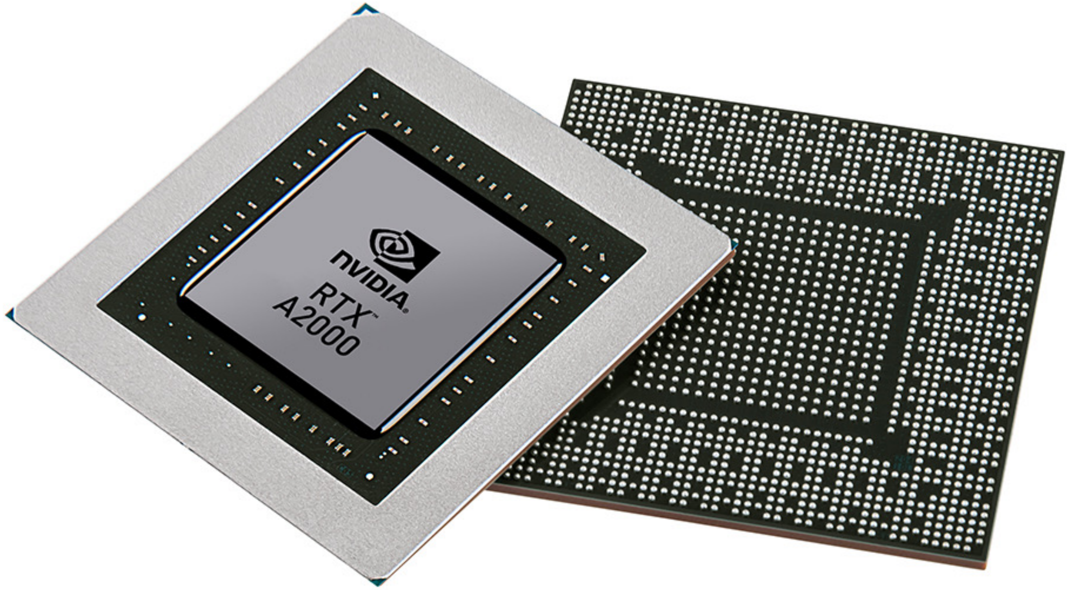
A Typical WebGL Program



GPU Pipeline

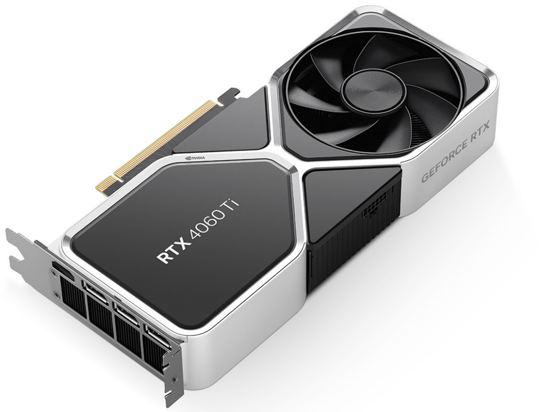


Graphics/Video Card



Graphics Processing Unit (GPU) Chip

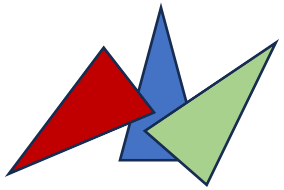
GPU Pipeline



- One of the primary tasks of a GPU is to produce raster images (graphics)
  - Videos, games, visualization, animation, etc.
- GPUs can be used for general purpose computing, known as **GPGPU**
  - Consider taking Parallel Programming (CS433) and Parallel Computing (CS633)
- In this course, we focus on Graphics APIs (WebGL) to perform real-time 2D/3D rendering using GPUs

GPU Pipeline

CPU memory



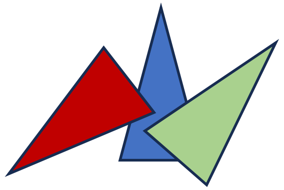
Source: Google

GPU Pipeline

GPU memory

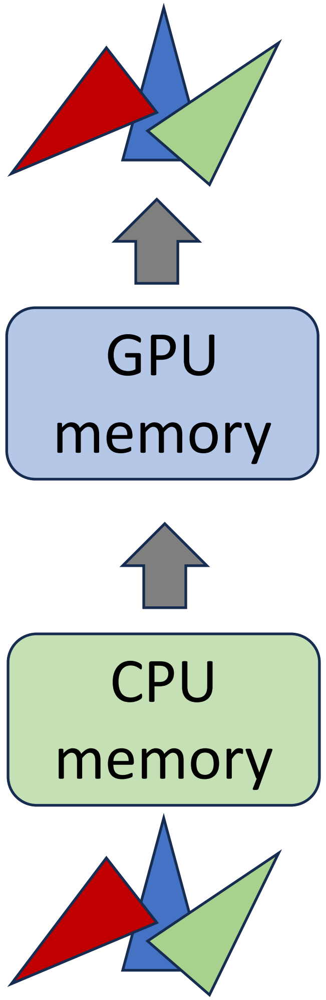
↑

CPU memory



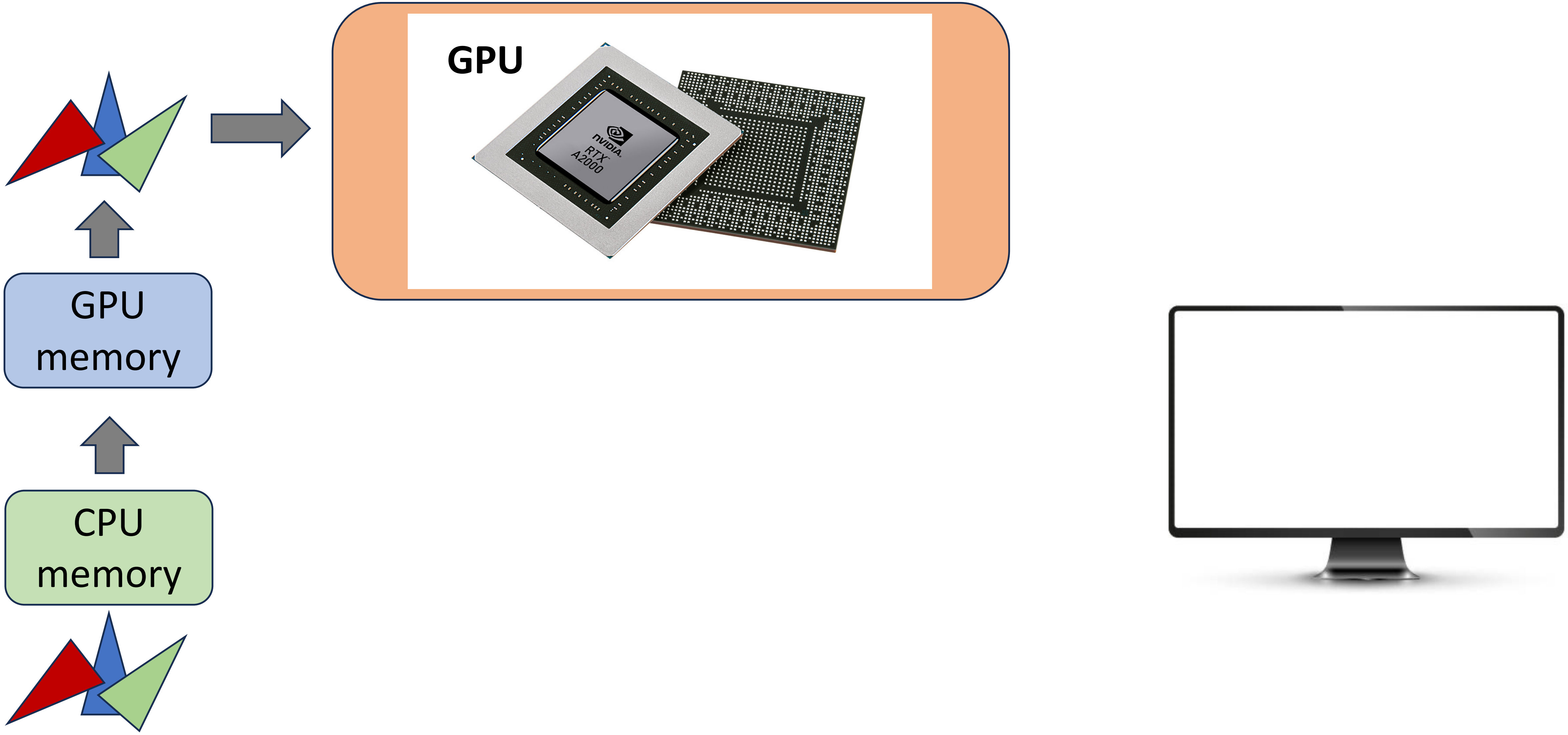
Source: Google

GPU Pipeline

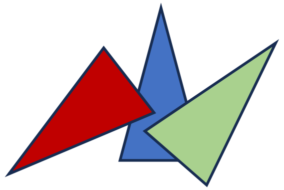


Source: Google

GPU Pipeline

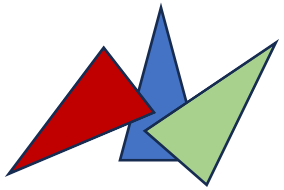


GPU Pipeline

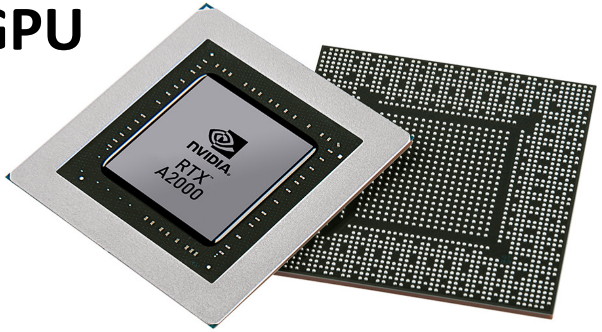


GPU memory

CPU memory

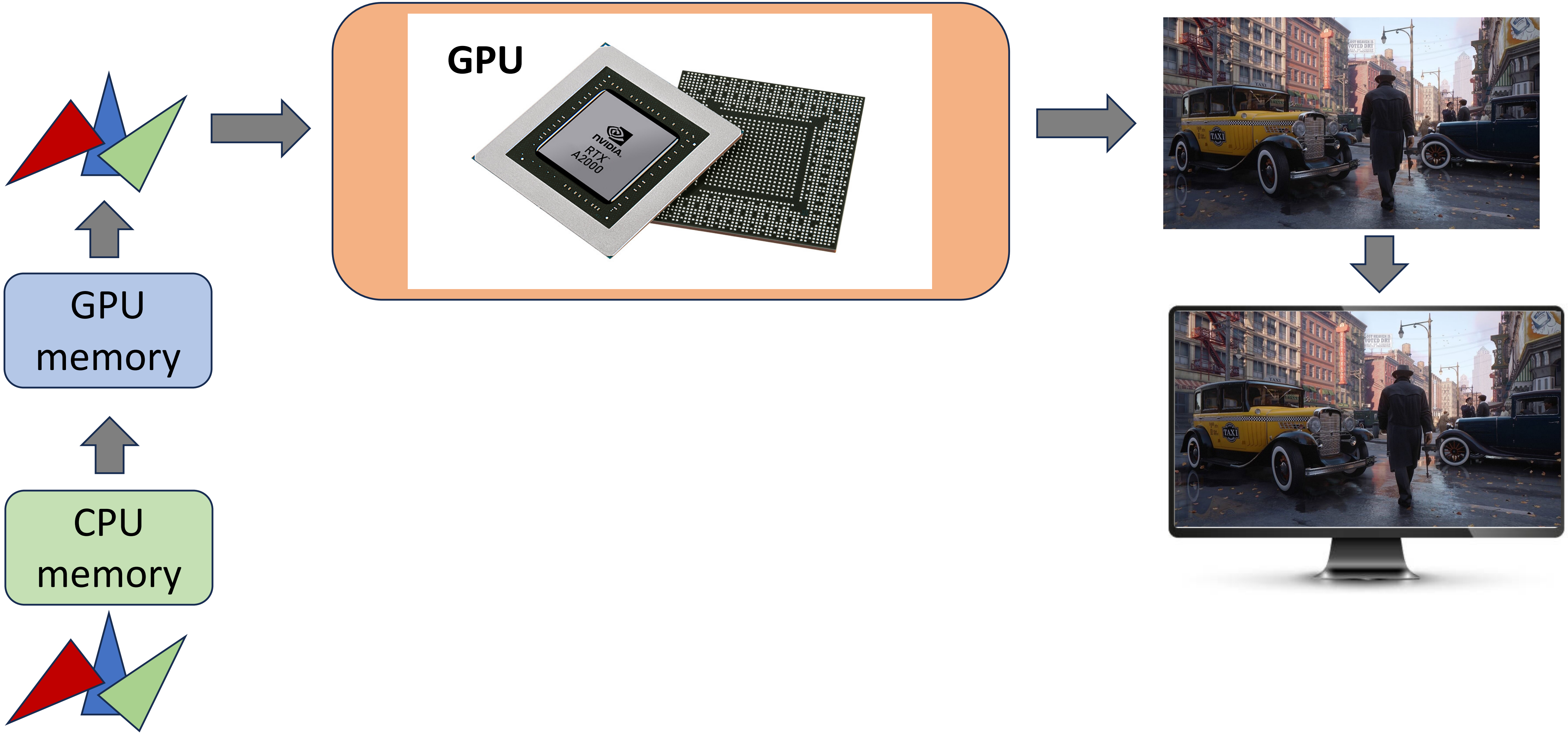


GPU

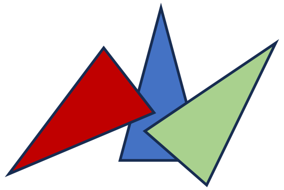


Source: Google

GPU Pipeline

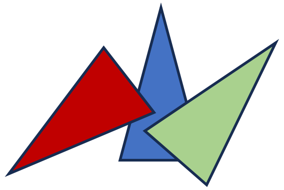


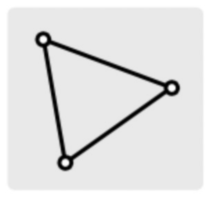
GPU Pipeline



GPU memory

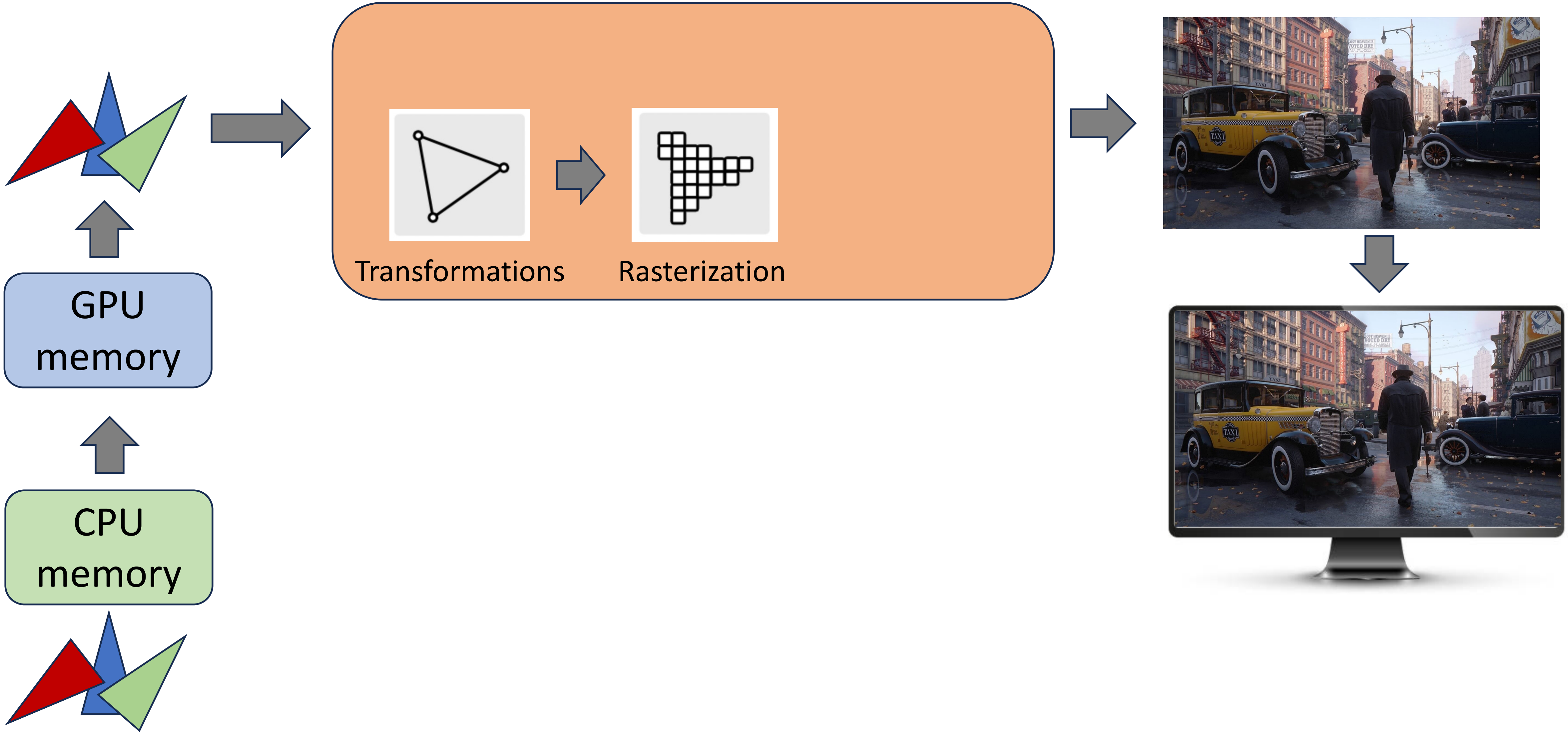
CPU memory



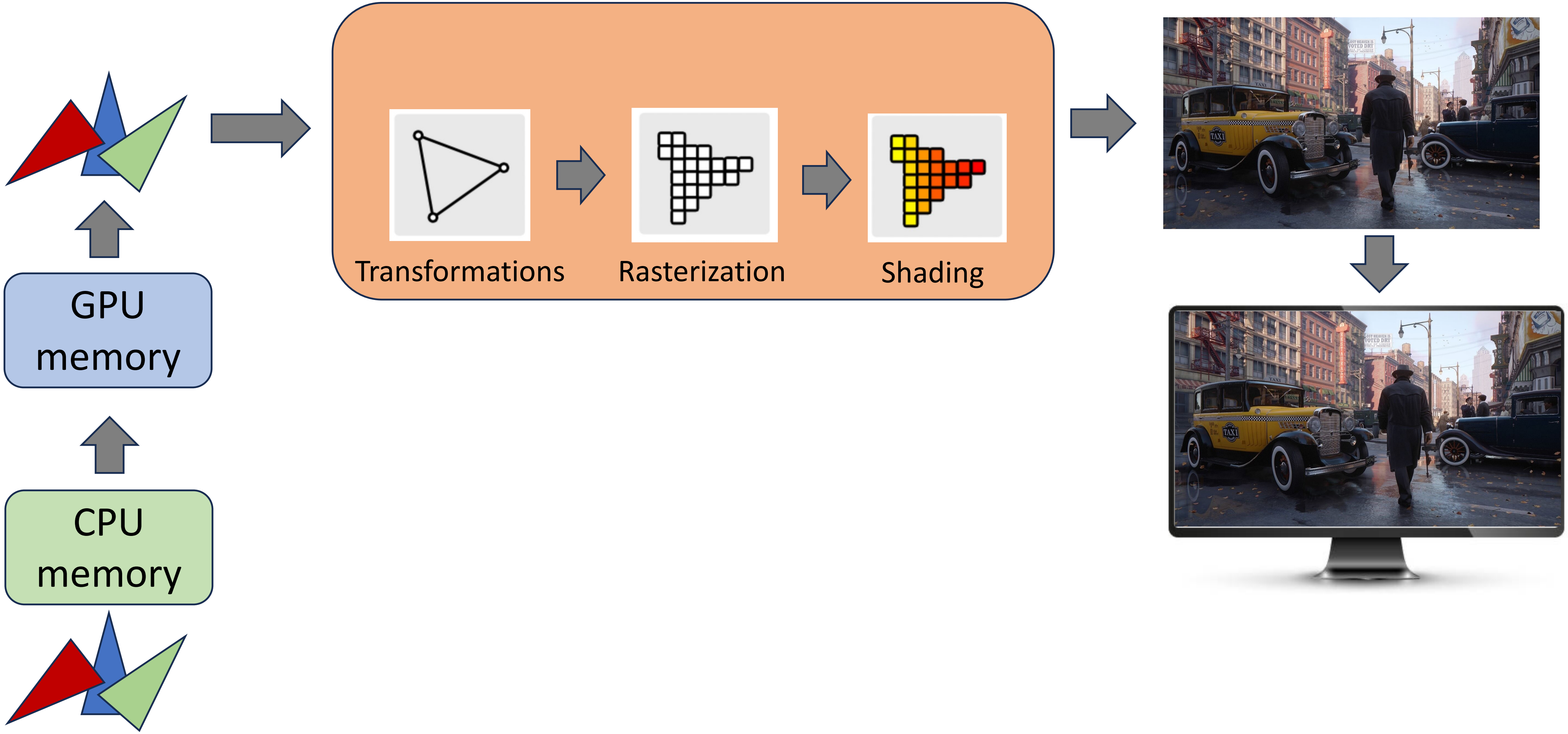


Transformations

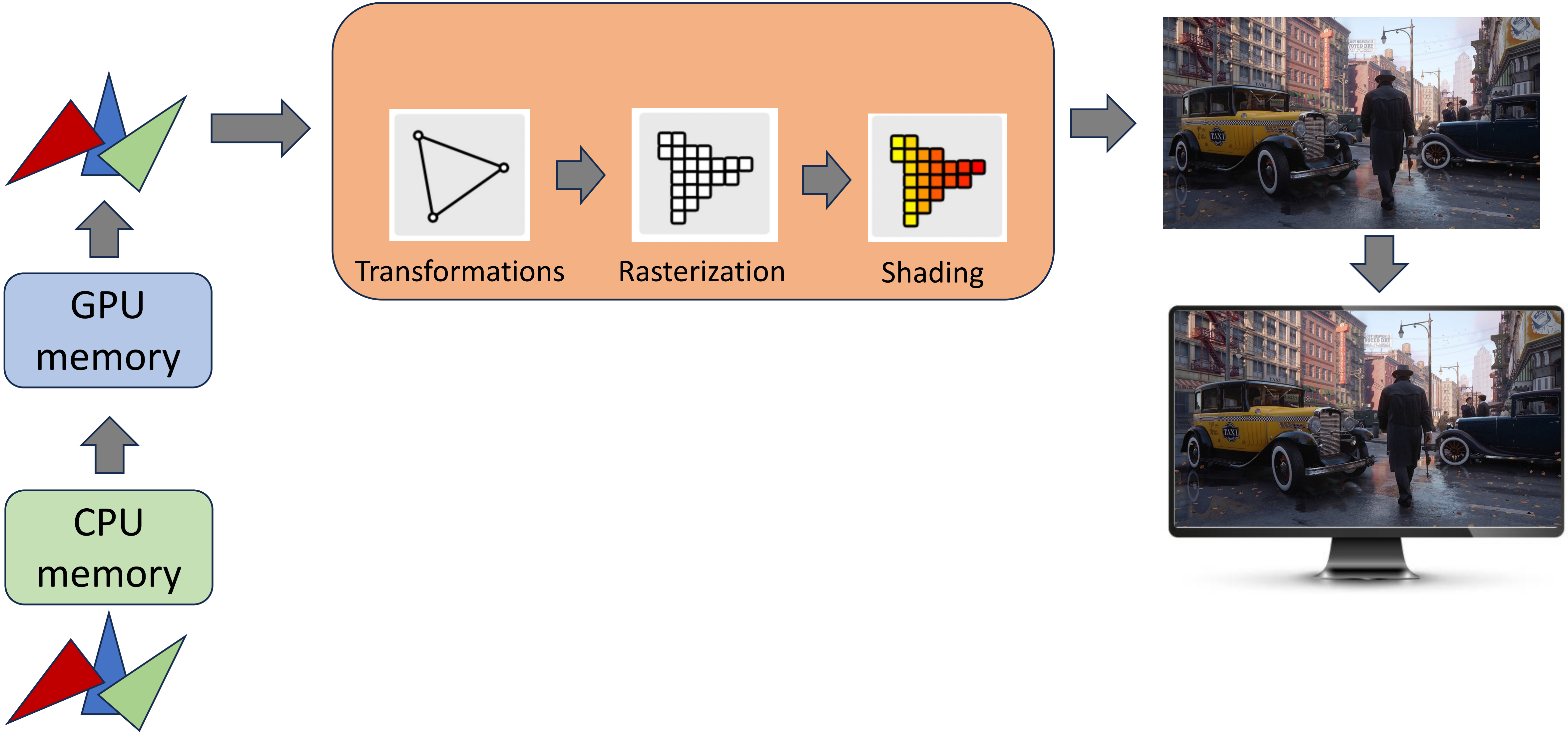
GPU Pipeline



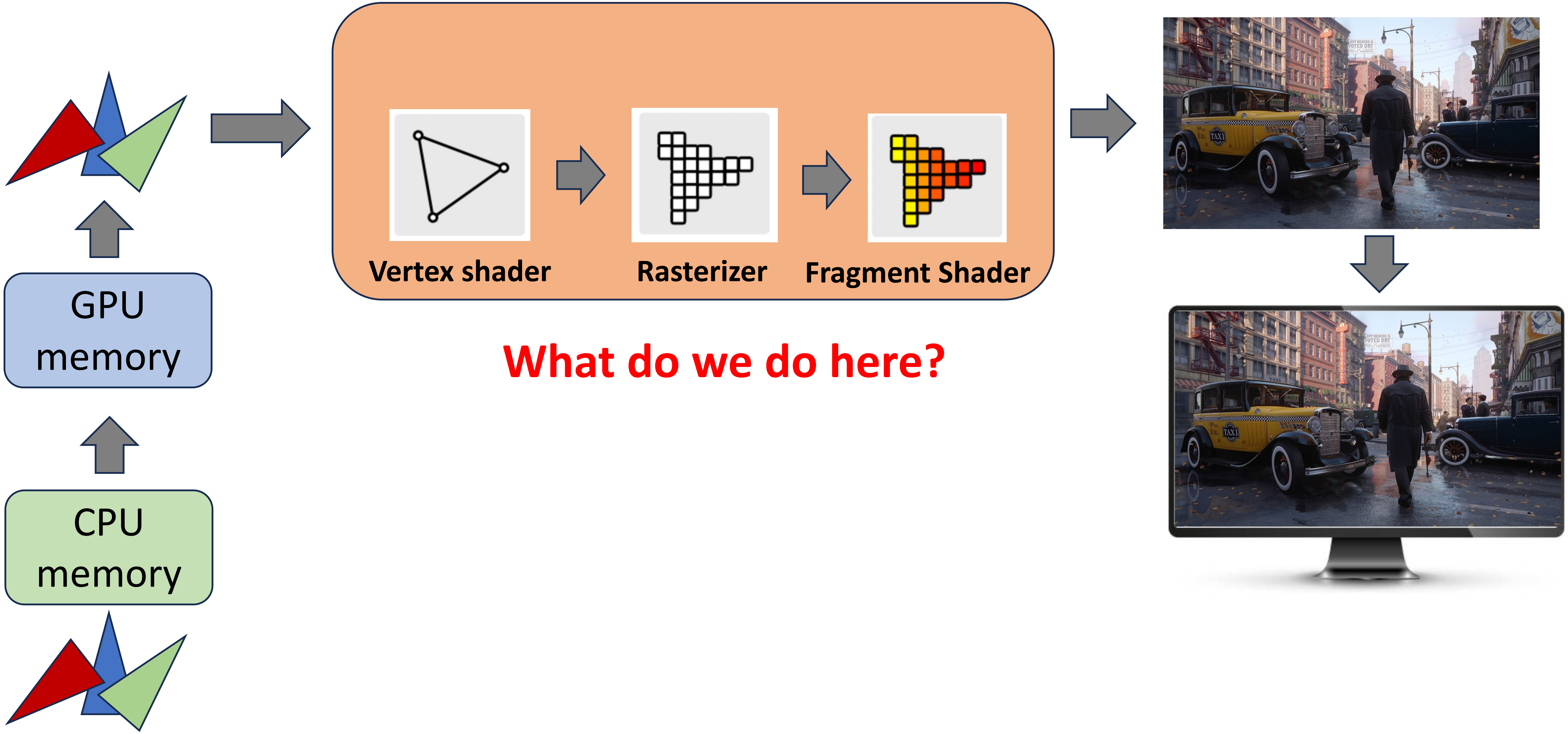
GPU Pipeline



GPU Pipeline

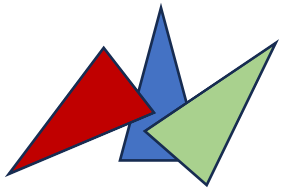


GPU Pipeline



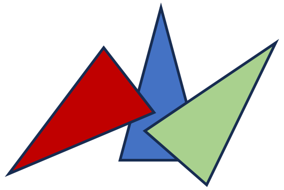
GPU Pipeline

IITK CS360A



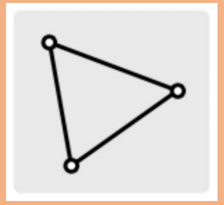
GPU memory

CPU memory



Programmable

Programmable



Vertex shader

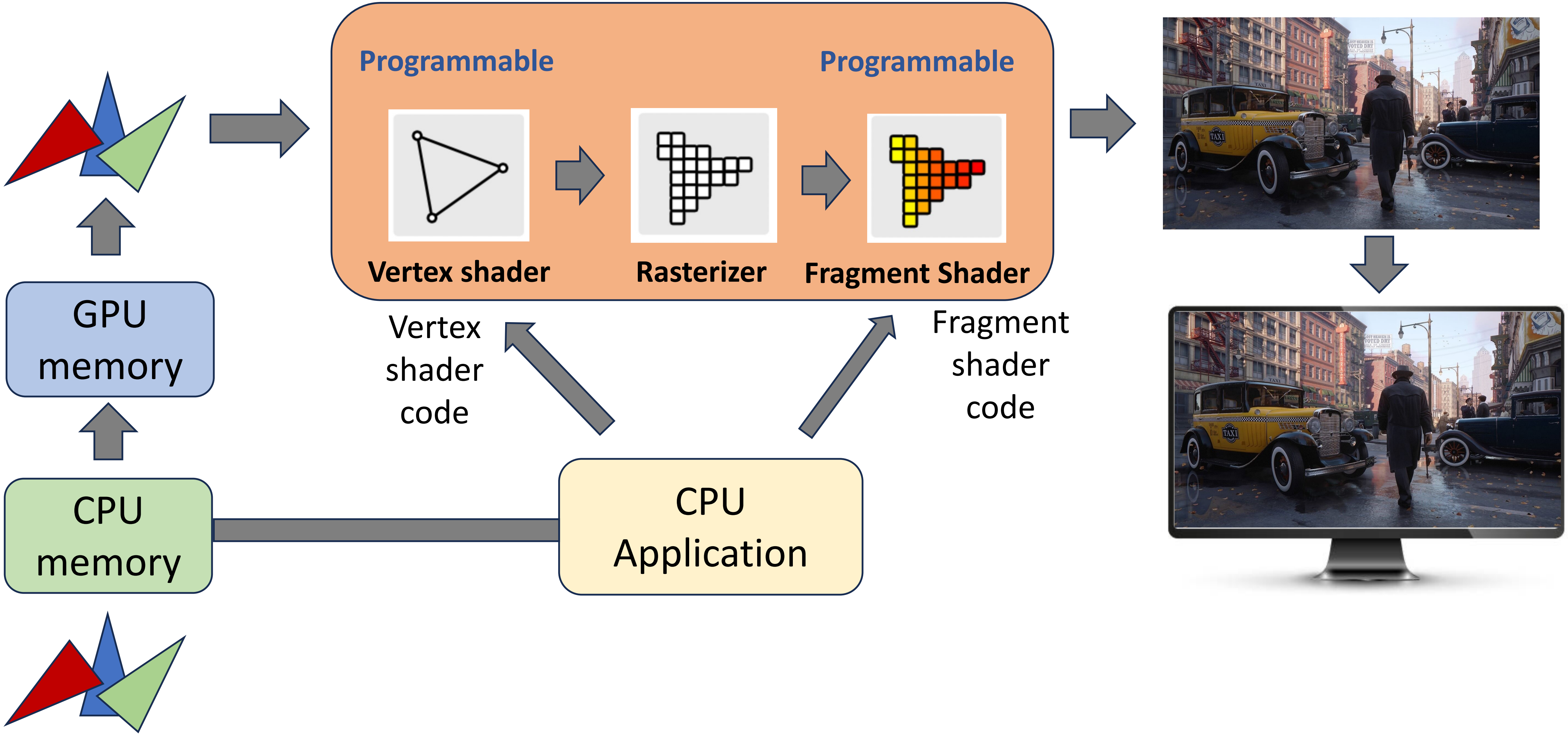
Rasterizer

Fragment Shader

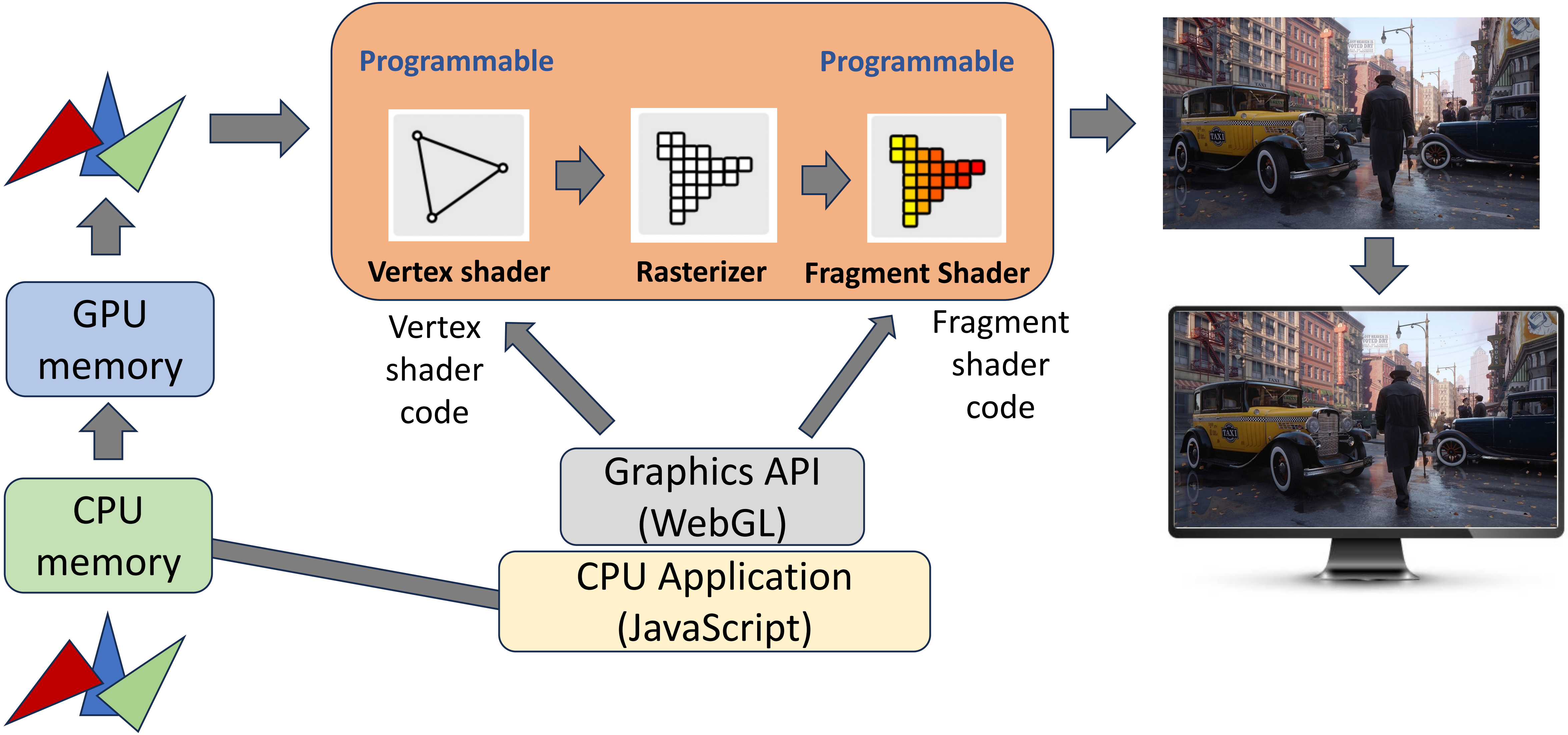
What do we do here?

Source: Google

GPU Pipeline



GPU Pipeline



WebGL

- https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API

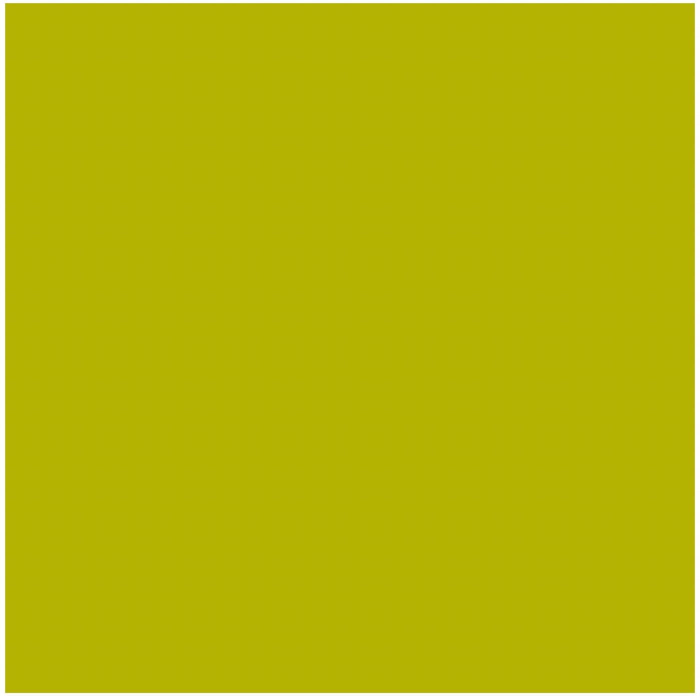
WebGL: Simple Canvas (HTML)

IITK CS360A

```html
<!DOCTYPE html>
<html>

  <head>
    <title>WebGL SimpleCanvas</title>
    <script type="text/javascript"
            src="simpleCanvas.js"></script>
  </head>

  <body onload="webGLStart();">
    <canvas
      id="simpleCanvas"
      width="500"
      height="500"
    ></canvas>
  </body>

</html>
```

WebGL: Simple Canvas (JavaScript)

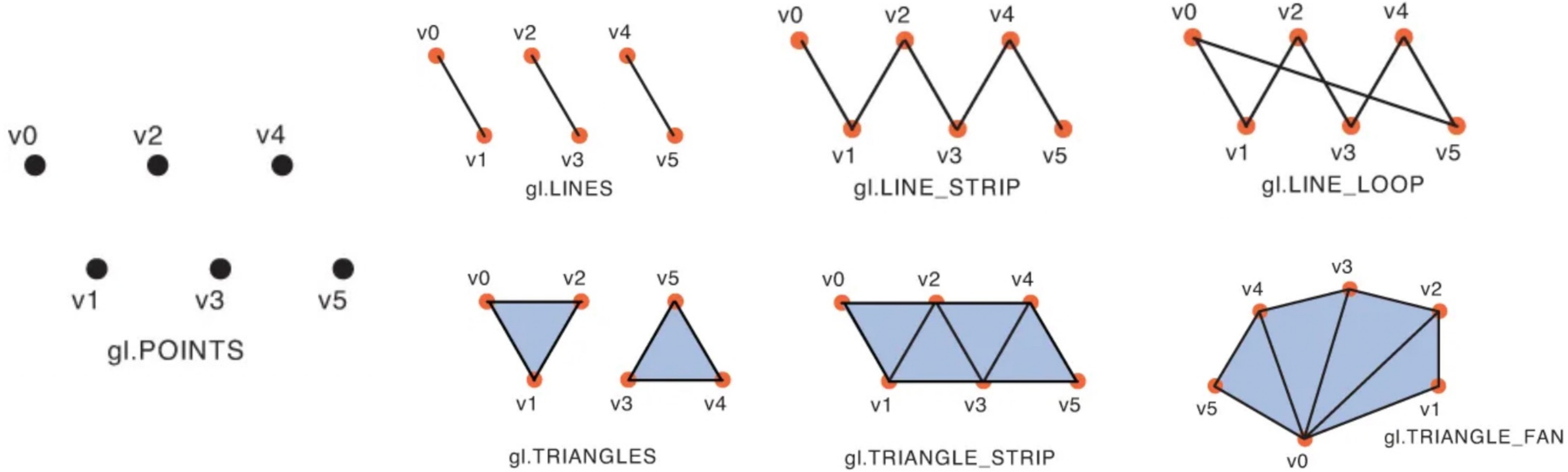
```javascript
var gl;

function initGL(canvas) {
  try {
    gl = canvas.getContext("webgl2"); // the webgl2 graphics context
    gl.viewportWidth = canvas.width; // the width
    gl.viewportHeight = canvas.height; // the height
  } catch (e) {}
  if (!gl) {
    alert("WebGL initialization failed");
  }
}

////////////////////////////////////////////////////////////////////////
// The main drawing routine, but does nothing except clearing the canvas
//
function drawScene() {
  gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
  gl.clearColor(0.7, 0.7, 0.0, 1.0);
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
}

// This is the entry point from the html
function webGLStart() {
  var canvas = document.getElementById("simpleCanvas");
  initGL(canvas);
  drawScene();
}
```



Output

WebGL: Scene Data

- Scene data will be primarily consisted of vertices (points)
  - Vertices will form primitives (triangles)
  - Primitives will form objects
  - Objects will form a scene
- WebGL Primitives:



WebGL: Draw A Triangle: webGLStart()

IITK CS360A

```
// This is the entry point from the html
function webGLStart() {
  var canvas = document.getElementById("triangleRender");
  initGL(canvas);
  shaderProgram = initShaders();

  // Print how many vertex attributes are supported in your device
  console.log(gl.getParameter(gl.MAX_VERTEX_ATTRIBS));

  drawScene();
}
```

WebGL: Draw A Triangle: initGL(canvas)


```
function initGL(canvas) {
  try {
    gl = canvas.getContext("webgl2"); // the graphics webgl2 context
    gl.viewportWidth = canvas.width; // the width of the canvas
    gl.viewportHeight = canvas.height; // the height
  } catch (e) {}
  if (!gl) {
    alert("WebGL initialization failed");
  }
}
```

Vertex Buffer Object (VBO)

- A Vertex Buffer Object (VBO) is a memory buffer in the high-speed memory of graphics card
  - Hold information about vertices and its properties
- We can create just one VBO for a model and then render that model multiple times using the same VBO by instancing it
  - Apply transformations to translate/rotate/scale the model before rendering

Vertex Buffer Object (VBO)

IITK CS360A

```
// buffer for the three points and their color
const bufData = new Float32Array([
0.0, 0.5, 1.0, 0.0, 0.0, -0.5, -0.5, 0.0, 1.0, 0.0, 0.5, -0.5,
0.0, 0.0, 1.0,]);

// create VBO
const buf = gl.createBuffer();

// decide where to copy the data in GPU memory by binding
gl.bindBuffer(gl.ARRAY_BUFFER, buf);

// copy data from CPU buffer to GPU memory
gl.bufferData(gl.ARRAY_BUFFER, bufData, gl.STATIC_DRAW);
```

WebGL: Draw A Triangle: drawScene()

IITK CS360A

```
function drawScene() {
  gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
  gl.clearColor(0.9, 0.9, 0.9, 1.0);
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

  //get locations of attributes declared in the vertex shader
  const aPositionLocation = gl.getAttribLocation(shaderProgram, "aPosition");
  const aColorLocation = gl.getAttribLocation(shaderProgram, "aColor");

  // buffer for the three points
  const bufData = new Float32Array([
    0.0, 0.5, 1.0, 0.0, 0.0,
    -0.5, -0.5, 0.0, 1.0, 0.0,
    0.5, -0.5, 0.0, 0.0, 1.0,
  ]);
  const buf = gl.createBuffer();
  // decide where to copy the data in GPU memory by binding to it
  gl.bindBuffer(gl.ARRAY_BUFFER, buf);
  // copy data from CPU buffer to GPU memory
  gl.bufferData(gl.ARRAY_BUFFER, bufData, gl.STATIC_DRAW);
```

```
  gl.vertexAttribPointer(aPositionLocation, 2, gl.FLOAT, false, 5 * 4, 0);
  gl.vertexAttribPointer(aColorLocation, 3, gl.FLOAT, false, 5 * 4, 2 * 4);

  //enable the attribute arrays
  gl.enableVertexAttribArray(aPositionLocation);
  gl.enableVertexAttribArray(aColorLocation);

  // It says how many points are being drawn.
  // try: LINE_LOOP/TRIANGLES
  gl.drawArrays(gl.TRIANGLES, 0, 3); // 3 = 3 points are part of drawing
}
```

WebGL: Draw A Triangle: initShaders()


```
function initShaders() {
  shaderProgram = gl.createProgram();

  var vertexShader = vertexShaderSetup(vertexShaderCode);
  var fragmentShader = fragmentShaderSetup(fragShaderCode);

  // attach the shaders
  gl.attachShader(shaderProgram, vertexShader);
  gl.attachShader(shaderProgram, fragmentShader);
  //link the shader program
  gl.linkProgram(shaderProgram);

  // check for compilation and linking status
  if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
    console.log(gl.getShaderInfoLog(vertexShader));
    console.log(gl.getShaderInfoLog(fragmentShader));
  }

  //finally use the program.
  gl.useProgram(shaderProgram);

  return shaderProgram;
}
```

WebGL: Draw A Triangle: ShaderSetUps


```
function vertexShaderSetup(vertexShaderCode) {
  shader = gl.createShader(gl.VERTEX_SHADER);
  gl.shaderSource(shader, vertexShaderCode);
  gl.compileShader(shader);
  // Error check whether the shader is compiled correctly
  if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
    alert(gl.getShaderInfoLog(shader));
    return null;
  }
  return shader;
}
```

```
function fragmentShaderSetup(fragShaderCode) {
  shader = gl.createShader(gl.FRAGMENT_SHADER);
  gl.shaderSource(shader, fragShaderCode);
  gl.compileShader(shader);
  // Error check whether the shader is compiled correctly
  if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
    alert(gl.getShaderInfoLog(shader));
    return null;
  }
  return shader;
}
```

WebGL: Draw A Triangle: Shaders

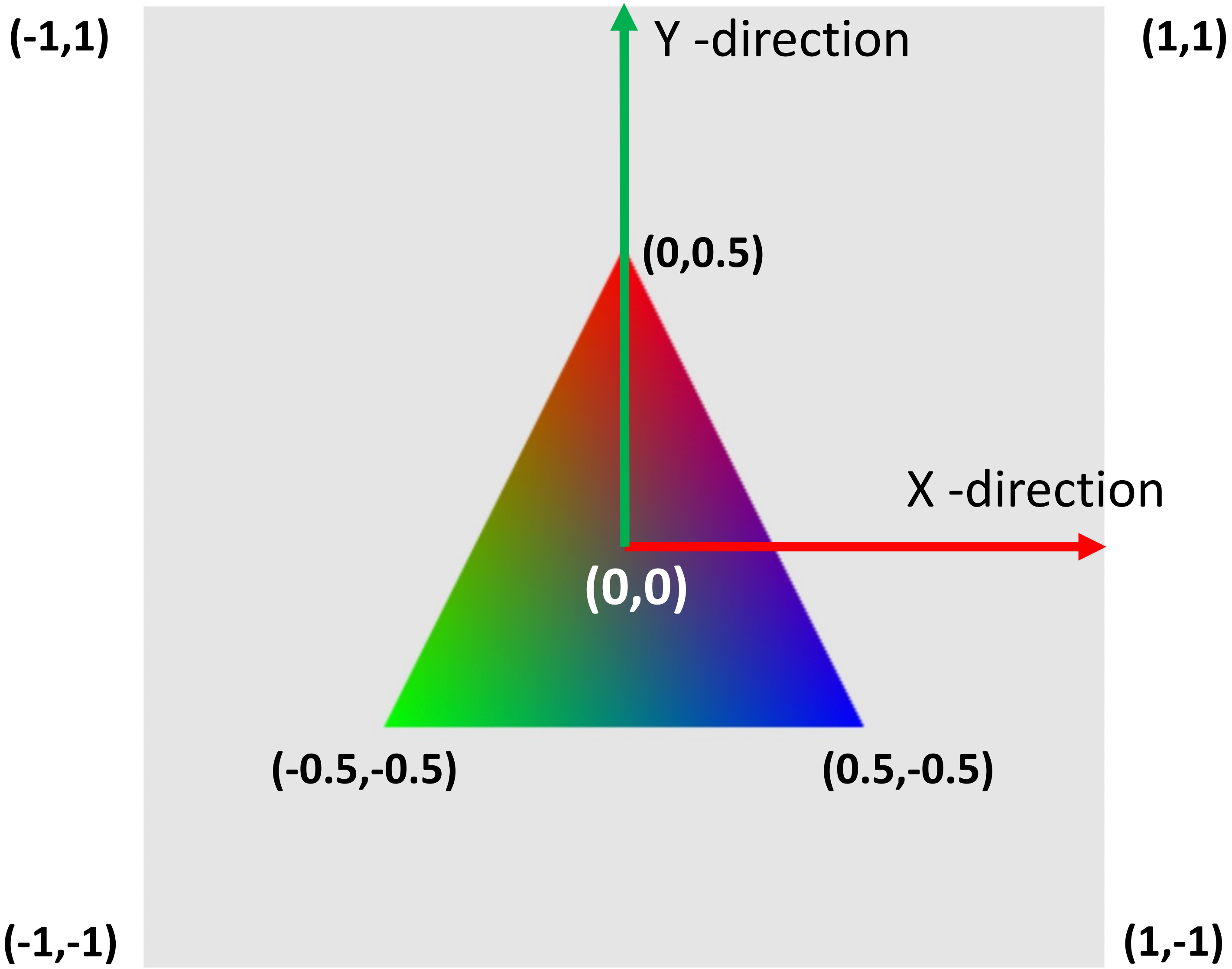

```
const vertexShaderCode = `#version 300 es
in vec2 aPosition;
in vec3 aColor;
out vec3 fColor;

void main() {
  fColor = aColor;
  gl_Position = vec4(aPosition,0.0,1.0);
}`;
```

Vertex Shader Code

```
const fragShaderCode = `#version 300 es
precision mediump float;
out vec4 fragColor;
in vec3 fColor;

void main() {
  fragColor = vec4(fColor, 1.0);
}`;
```

Fragment Shader Code

WebGL: Draw A Triangle: Output



OpenGL Shading Language (GLSL)

- A high-level shading language with a syntax based on the C programming language
- Created by the OpenGL ARB (OpenGL Architecture Review Board) to give developers more direct control of the graphics pipeline without having to use ARB assembly language or hardware-specific languages

- WebGL2 supports GLSL ES 3.0 Spec: https://registry.khronos.org/OpenGL/specs/es/3.0/GLSL_ES_Specification_3.00.pdf
- https://www.khronos.org/files/opengl42-quick-reference-card.pdf

Wikipedia

OpenGL Shading Language (GLSL)

- Data Types:
    - bool
    - int
    - float
    - bvec2, bvec3, bvec4: 2, 3, and 4-component Boolean vectors
    - ivec2, ivec3, ivec4: 2, 3, and 4-component integer vectors
    - vec2, vec3, vec4: 2, 3, and 4-component floating point vectors
    - mat2, mat3, mat4: 2x2, 3x3, and 4x4 floating point matrices
- Special Data types
    - sampler2D: a reference to a TEXTURE_2D *texture unit* (which has an attached *texture object*)
    - samplerCube: a reference to a SAMPLER_CUBE *texture unit*

Vector Components

- The individual element of a vector can be accessed using array notation, 0^{th} element of vector a is = a[0]
- 'dot' notation can also be used such as 0^{th} element of vector a is = a.x
  - The names of the vector components are x,y,z,w, or r,g,b,a, or s,t,p,q
  - You can use any of these names on a vector, regardless of the actual data in the vector
    - But, the intent is to use x,y,z,w when you are accessing geometric data
    - r,g,b,a when you are accessing color data
    - s,t,p,q when you are accessing texture data

Vector Components


```
vec3 alpha = vec3(1.0, 2.0, 3.0);
vec4 a;
vec3 b;
vec2 c;
float d;

b = alpha.xyz;  // b is now (1.0, 2.0, 3.0)
d = alpha[2];   // d is now 3.0
a = alpha.xxxx; // a is now (1.0, 1.0, 1.0, 1.0)
c = alpha.zx;   // c is now (3.0, 1.0)
b = alpha.rgb;  // b is now (1.0, 2.0, 3.0)
b = alpha.stp;  // b is now (1.0, 2.0, 3.0)
a = alpha.yy;   // compiler error; the right hand side is a 2-component vector,
                // while "a" is a 4-component vector.
```

Overall Execution of GLSL Program

- A shader program is composed of one or more functions.
- Execution always begins with the main function which receives no parameters and returns no value:

```
void main(void) {
  // statement(s)
}
```

Main function

```
vec3 example(float x, bool beta) {
  // statement(s)
}
```

A different function declaration

Function Parameter Qualifiers

- **const**: for function parameters that cannot be written to
- **in**: for function parameters passed into function
- **out**: for function parameters passed back out of function, but not initialized when passed in
- **inout**: for function parameters passed both into and out of a function

Storage Qualifiers

- **none**: (default) local read/write memory, or input parameter
- **const**: global compile-time constant, or read-only function parameter, or read-only local variable
- **in**: linkage into shader from previous stage
- **out**: linkage out of a shader to next stage

Storage Qualifiers

- How can we pass information from JavaScript to vertex/fragment shader code?
    - Use uniforms
- uniform: linkage between a shader, OpenGL, and the application
    - uniform1i, uniform 1f
    - uniform2fv, uniform3fv, uniform4fv, ….
    - uniformMatrix2fv, uniformMatrix3fv , uniformMatrix4fv, …

Some Useful Built-in Shader Functions

IITK CS360A

- abs() = absolute value
- sign() = returns -1.0, 0.0, or 1.0
- min(), max()
- floor(), ceil(), round(), trunc()
- mod() = modulus
- Length() = length of a vector
- distance() = distance between two points
- dot() = dot product between two vectors
- cross() = cross product between two vectors
- normalize() = normalize a vector
- reflect() = compute reflection vector
- refract() = compute refraction vector

- transpose() = matrix transpose
- inverse() = matrix inverse
- determinant() = matrix determinant
- matrixCompMult() = component-wise multiply