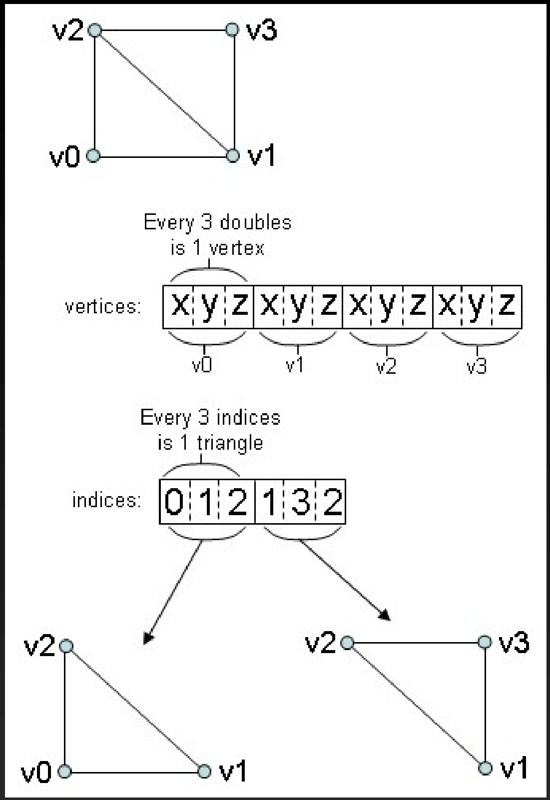
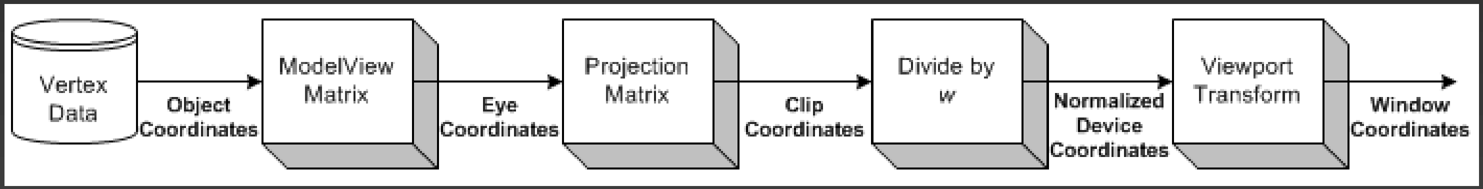
Overview

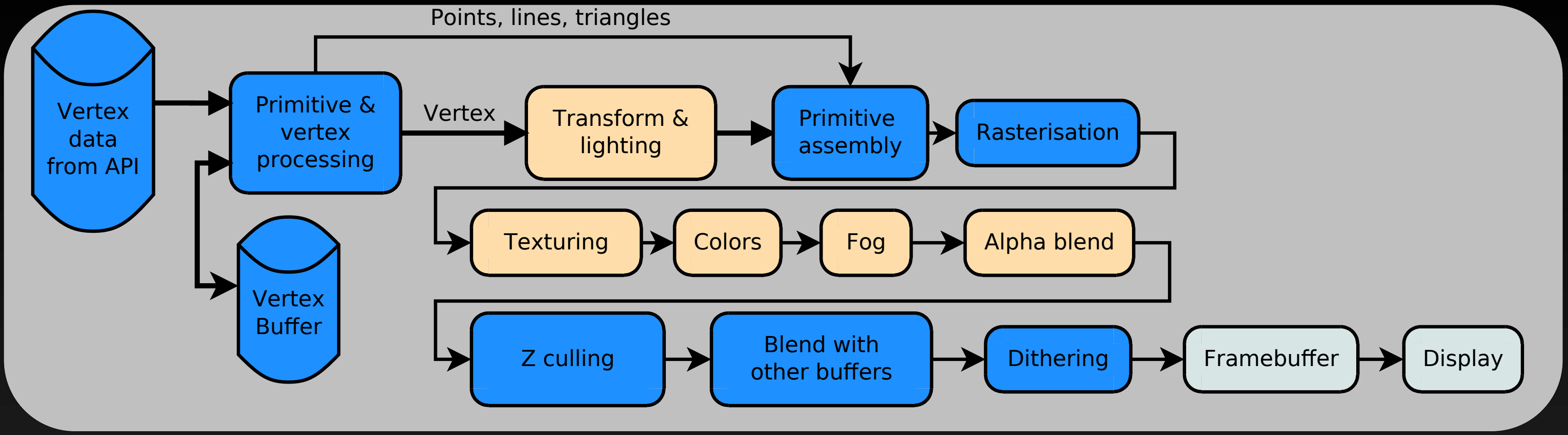


- Representation
  - 3D boundary
    - vertex
    - edge
    - face
- 3D render
  - load data
  - transform vertex coordinades



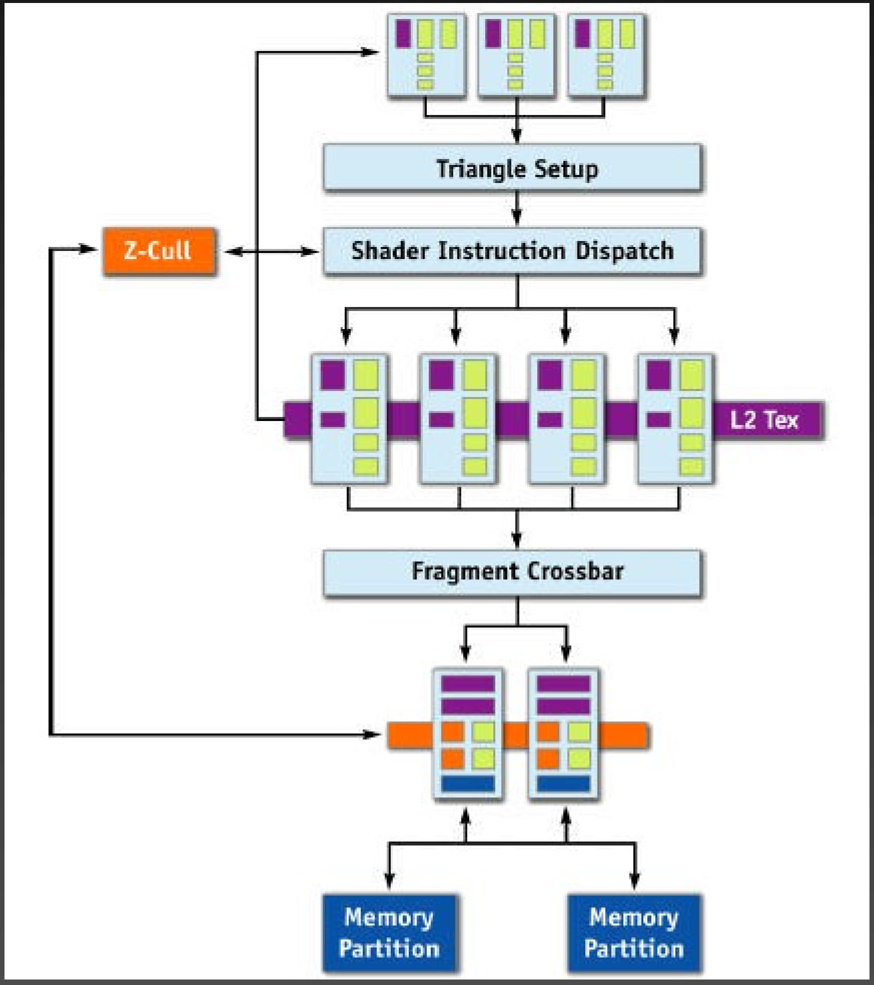
  - rasterisation (vector → fragments)
  - compute color of the fragment
  - resolve Z-occlusion and transparency

OpenGL pipeline

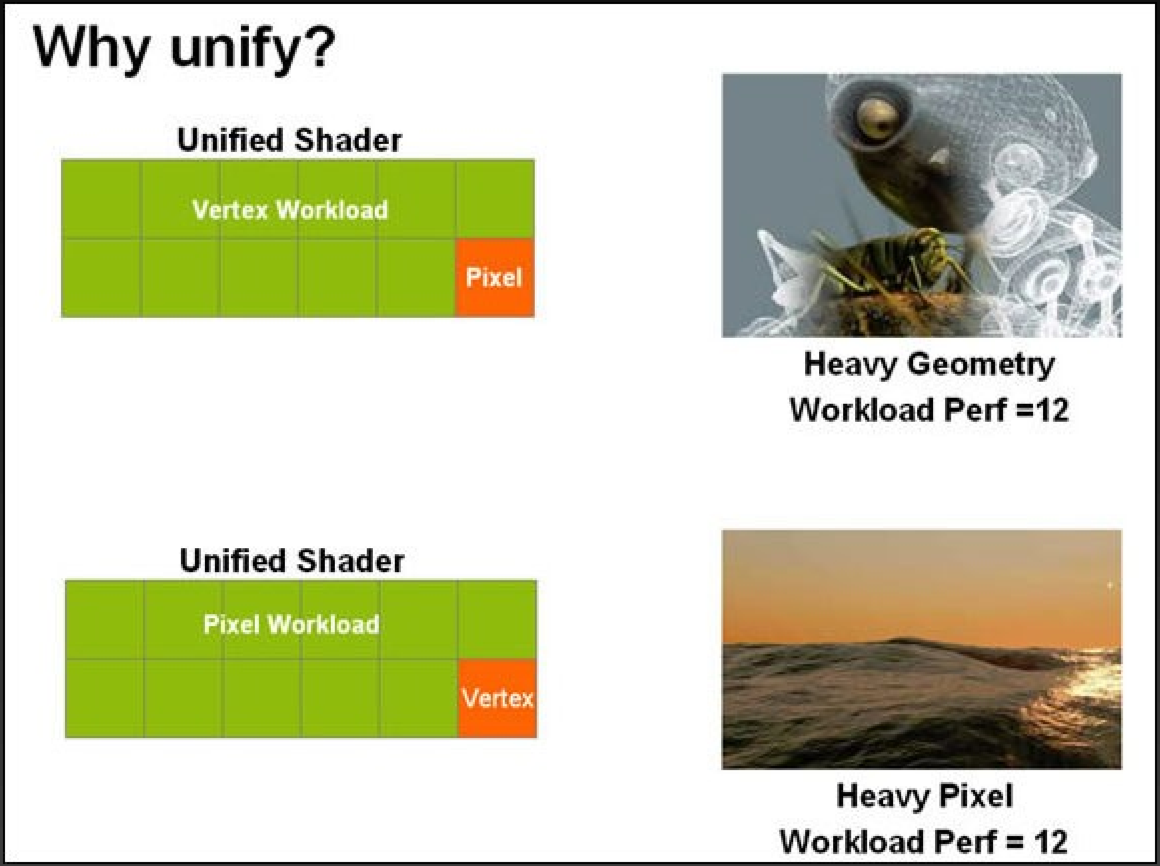


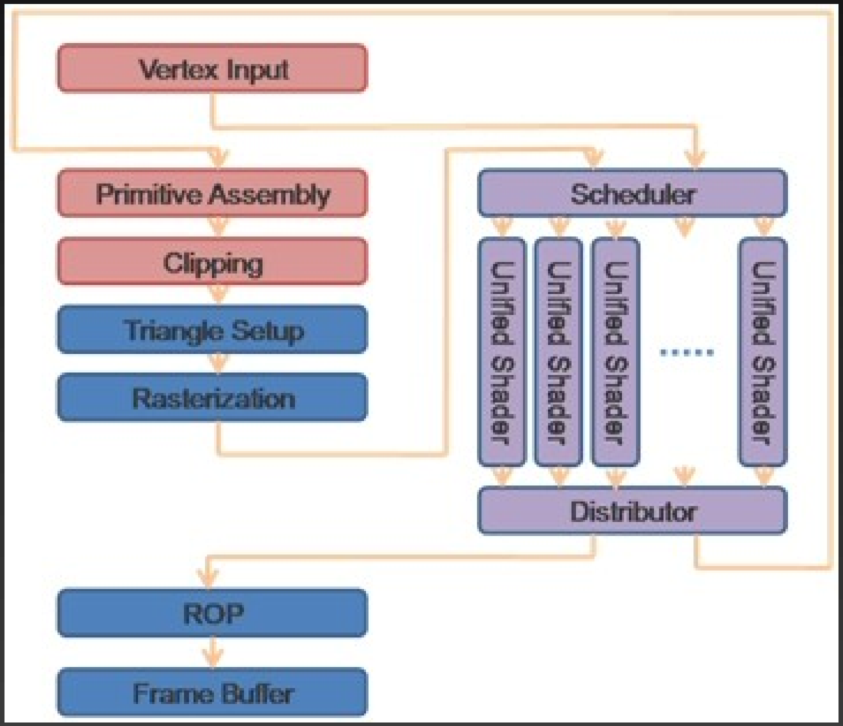
OpenGL Pipeline In Hardware

(top: fixed, bottom: non-unified)

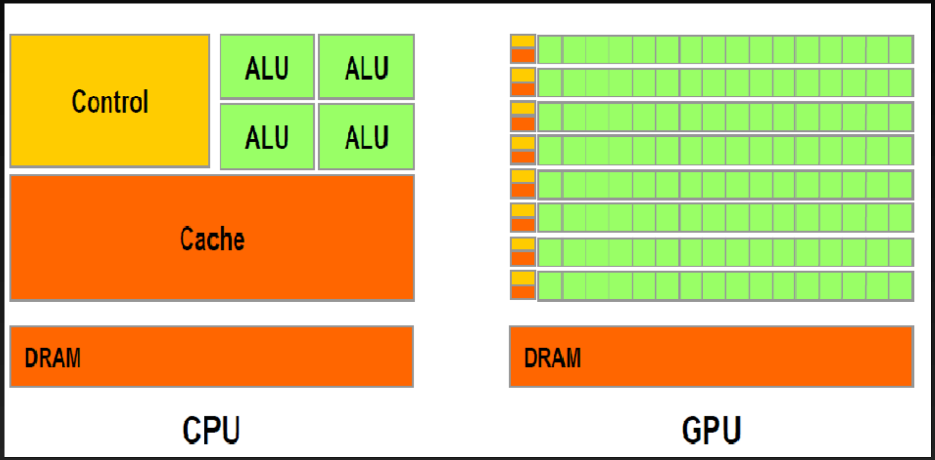


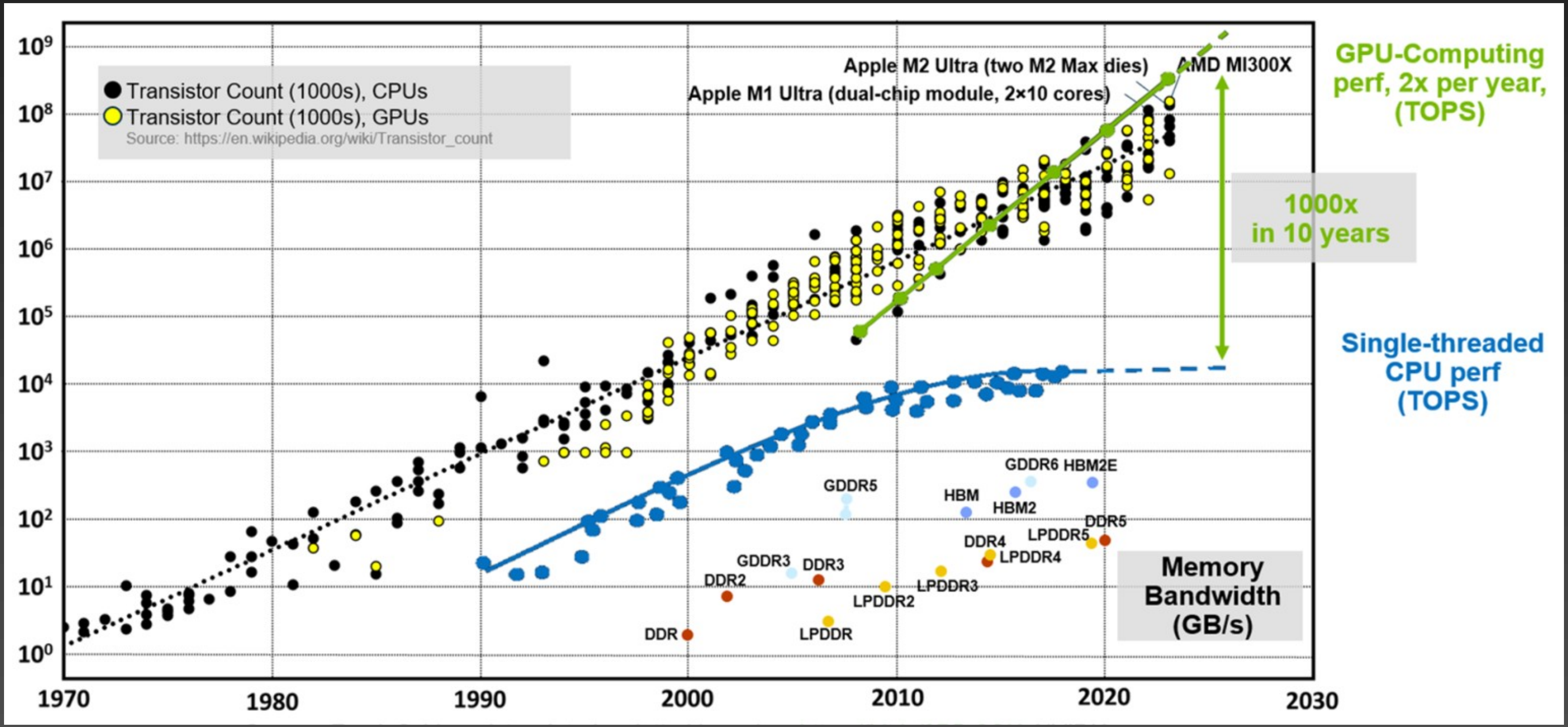
Unified shaders

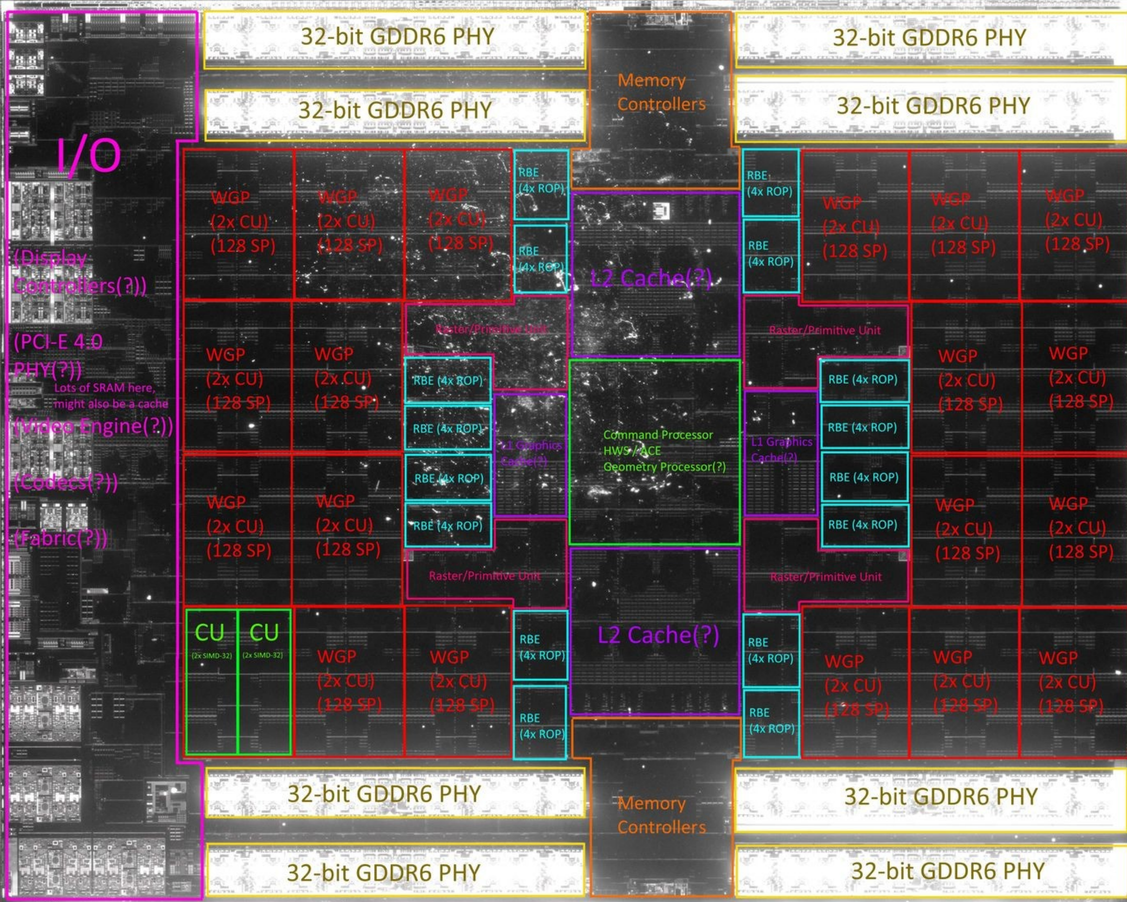


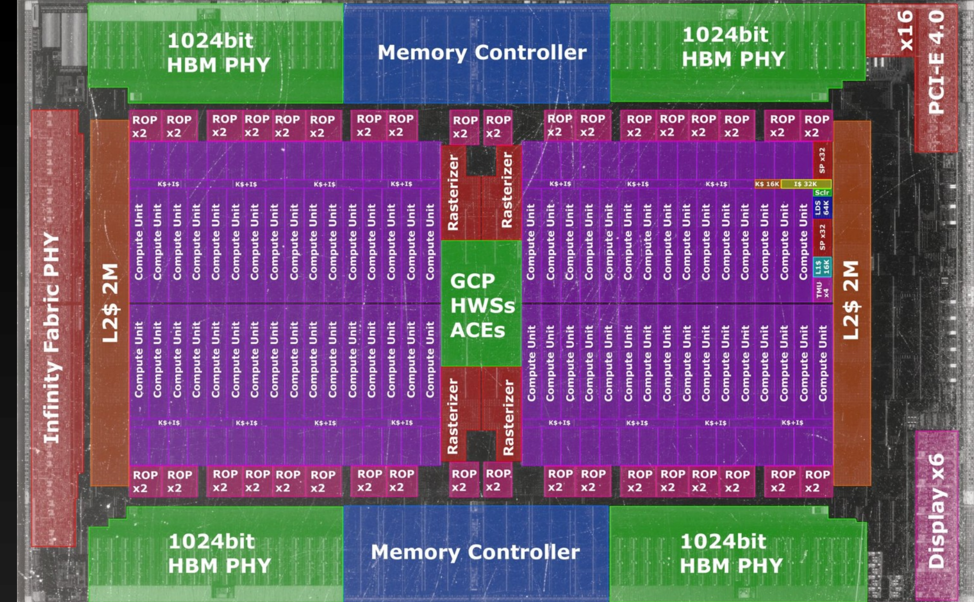


CPU vs. GPU performance







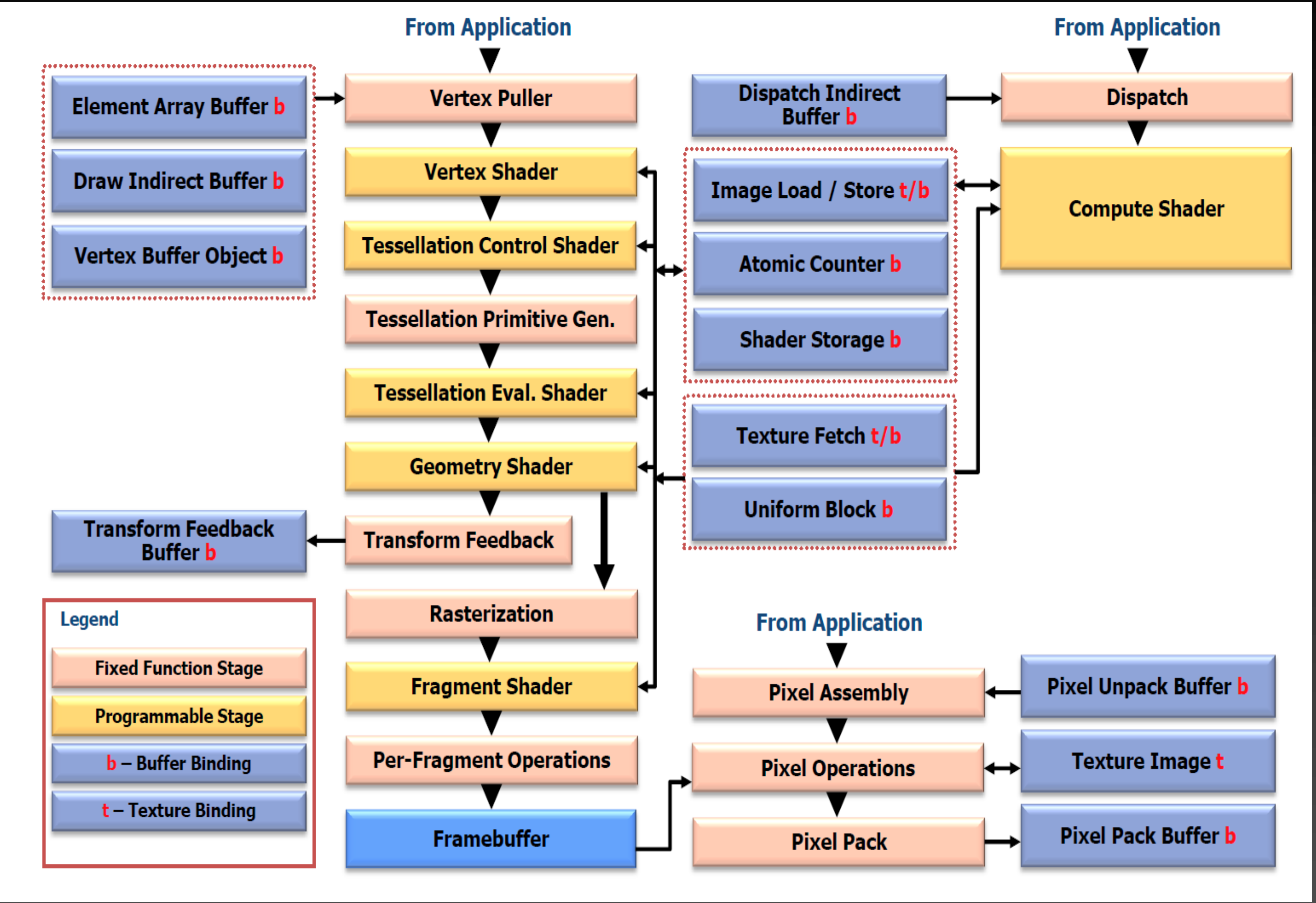


AMD VegaII (Radeon 7)
7nm, 331 mm^2, 3480 core

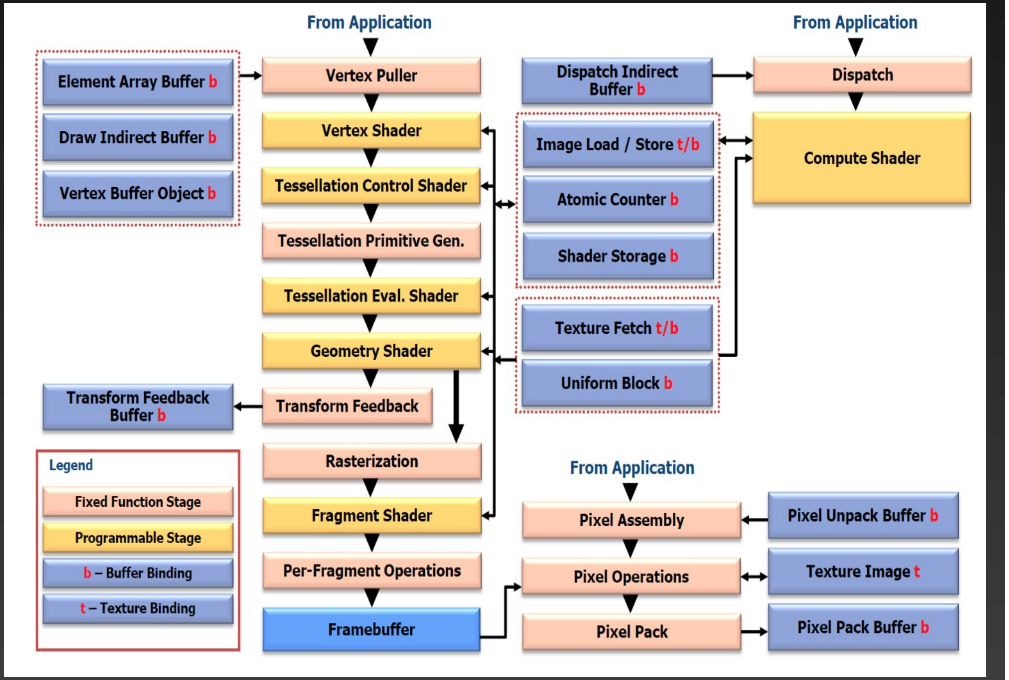
AMD Navi10 (Radeon 5700XT)
7nm, 251 mm^2, 2560 core

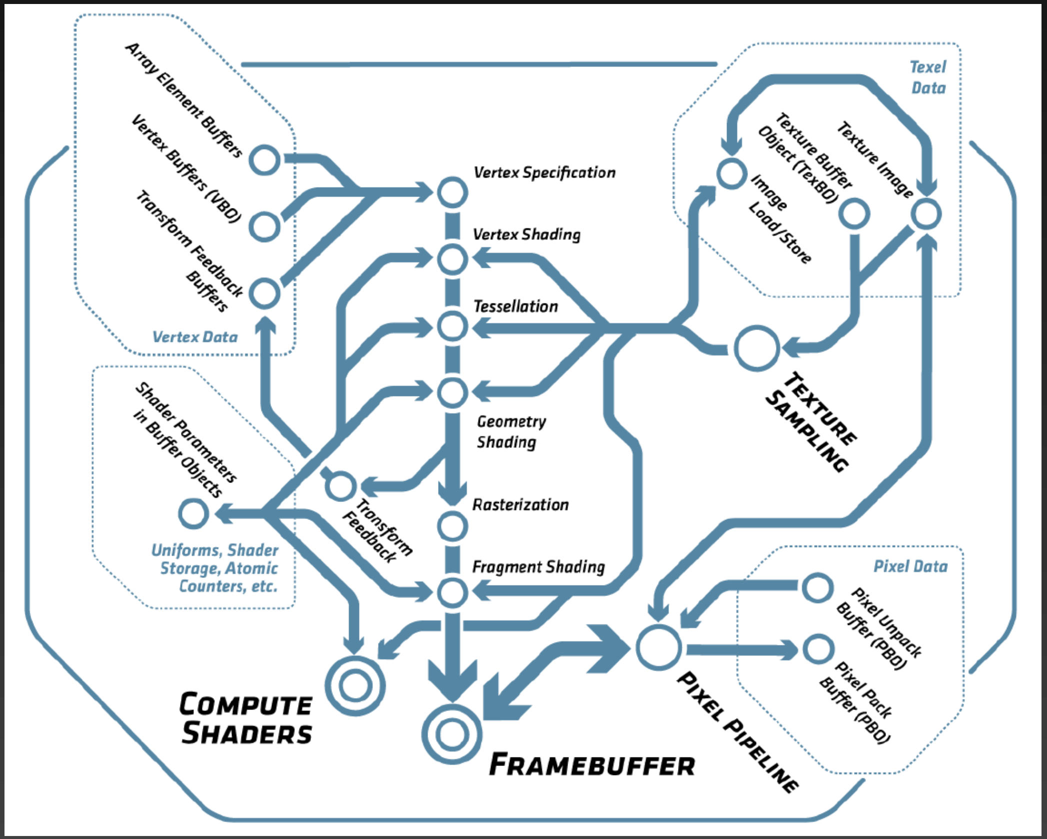
Intel Skylake (Xeon Platinum)
14nm, 694 mm^2, 28 core + SMT2

OpenGL 4.6 (Core Profile) - May 5, 2022



OpenGL 4.6 (Core Profile) - May 5, 2022





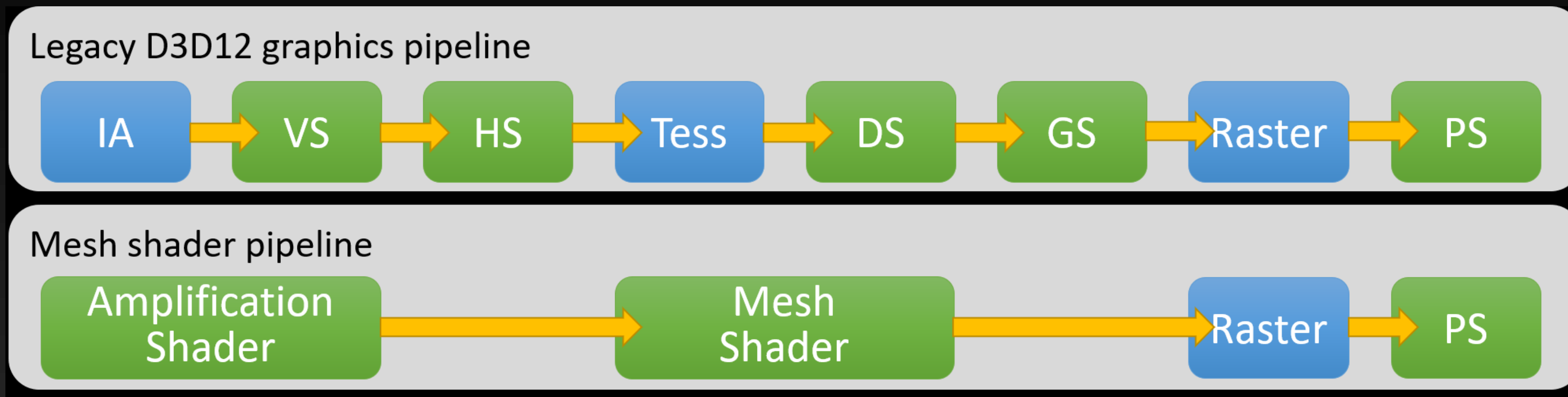
OpenGL pipeline functions

- Vertex processing
  - coordinates transformation, calculation of normals, colors, UV textures, …
    - directly or from lighting model
- Geometry processing
  - clipping, culling
  - perspective projection (larger distance → smaller object)
  - enabling of vertex/edges/faces rasterization
- Rasterisation = conversion to fragments
  - first: cull back side of polygons, clip by 6 planes, w division (perspective)
  - viewport, antialiasing
  - fragment = complex entity, set of information
    - similar to pixel, but not stored yet
    - each fragment has [x,y,z] coordinates and color
  - all information taken into account
    - line width, point size, lights, materials, antialiasing, …
  - drawing of edges, filling polygons
- Pixel and textures operations
  - decompression, format conversion, filtration
  - math operations (+,*, saturation, …)

OpenGL pipeline functions (cont.)

- Fragment operations
  - texturing, fog
  - clipping by stencil, depth
  - blending with already existing fragment (alfa blending)
  - dithering
  - math ops

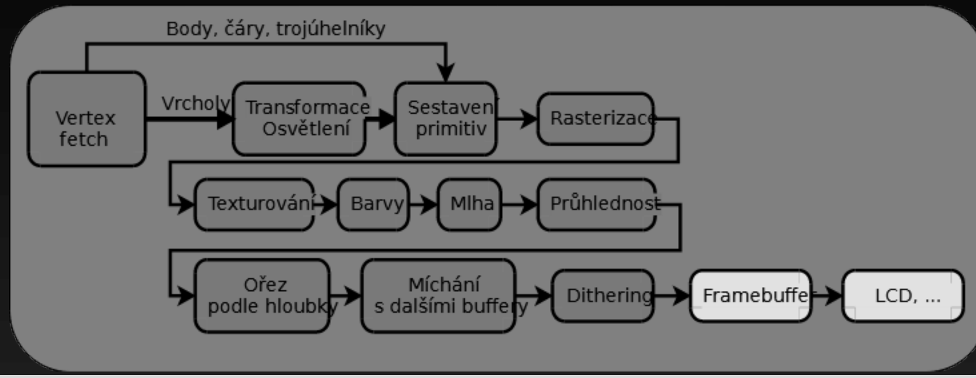
2020+ trend: fully programmable

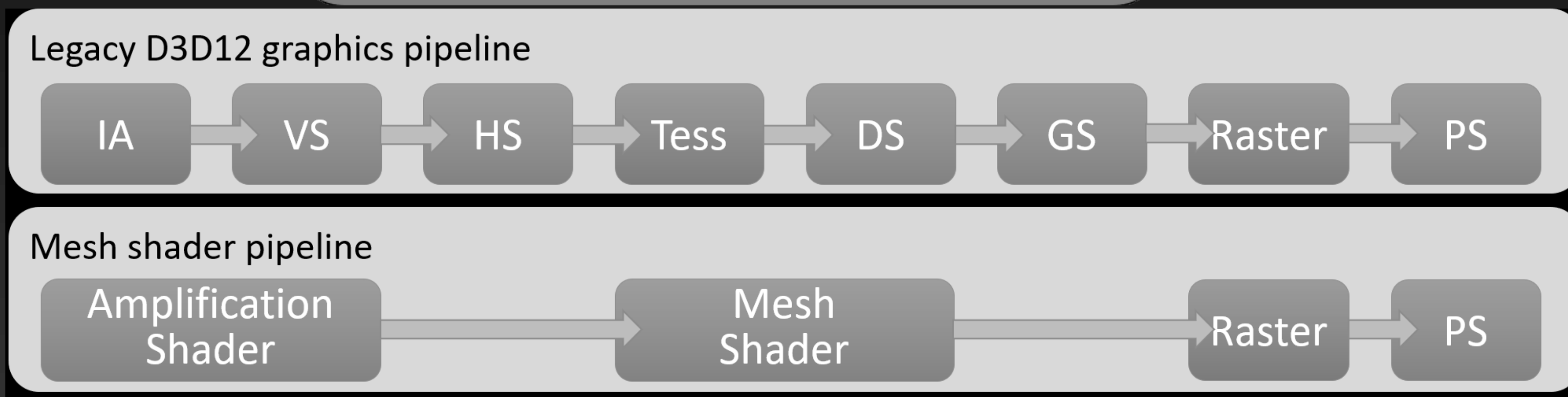


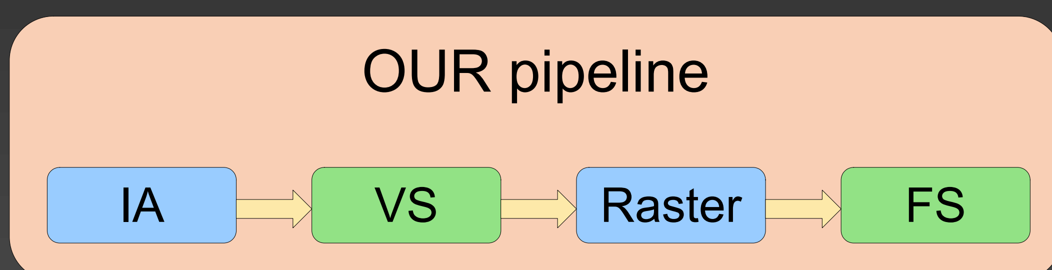
- API
  - OpenGL, DX12 Ultimate, Vulkan
- HW
  - PS5, Xbox series X
  - NVidia RTX 3000, AMD Radeon 6000
- Roots in
  - AMD Next-Generation Geometry (NGG), cca 2017

Our pipeline

Fixed pipeline (legacy)







Modern OpenGL

- relativelly new
  - check your graphics card for support
- create OpenGL context with latest version
  - no version specification → select latest
    - ⚠ profile not guaranteed
  - latest version: 4.6
- create OpenGL context with specific version
  - e.g. OpenGL Core version **4.6+**

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 6);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
```

- version specification in shaders

```
#version 460 core
```

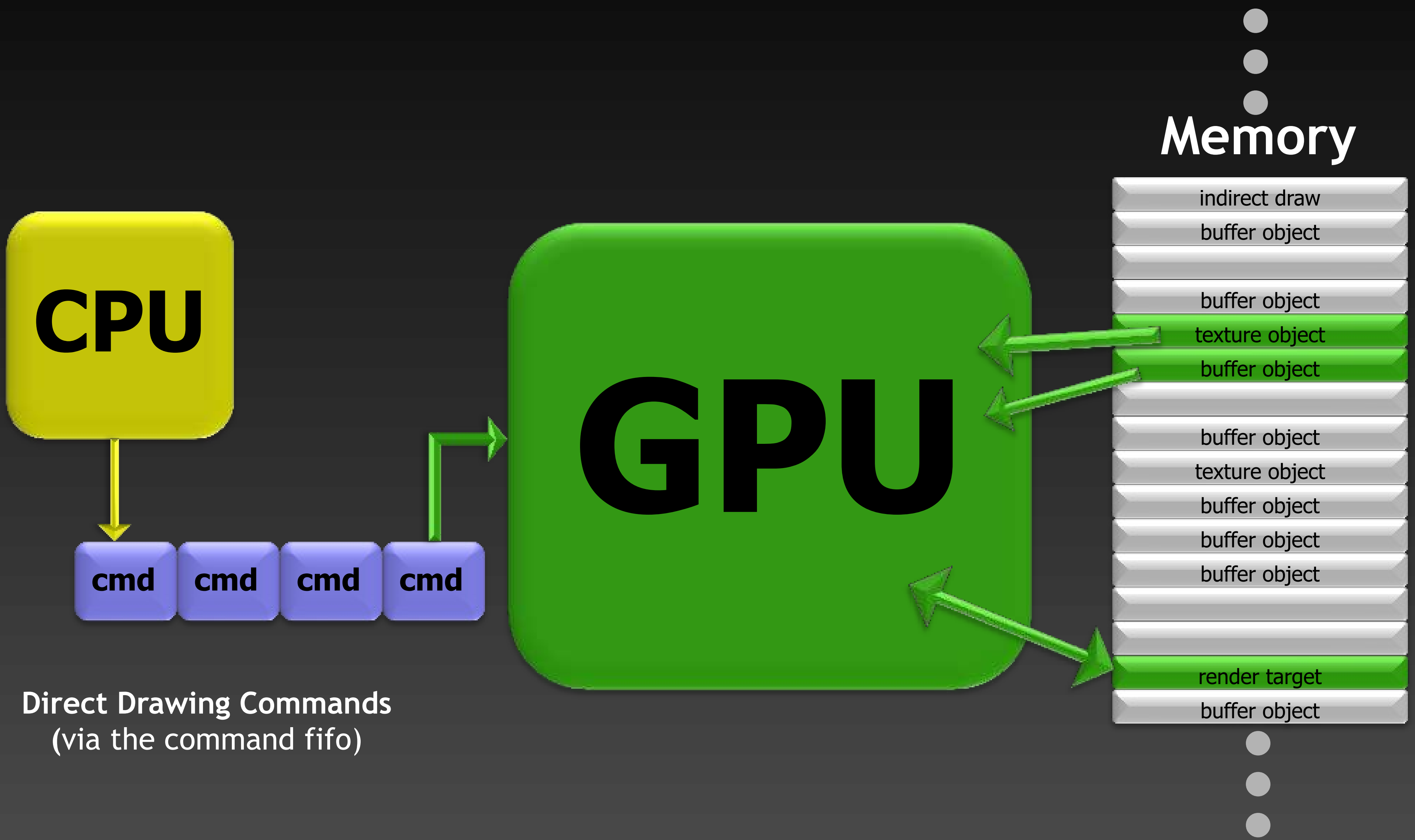
Modern OpenGL
DSA - Direct State Access

- OpenGL = state machine
  - to modify data, we need to bind it first
    - bind(res_A), modify_prop_foo(), modify_prop_bar(),
    - bind(res_B), modify_prop_foo(),…
  - slower (more calls)
  - error prone (modifying different object by accident)
  - prevents multithreading
- Explicit binding is expensive → get rid of it!
- Bindless acces = Direct State Access = MODIFY without binding
- AZDO → Approaching Zero Driver Overhead
  - reduce API calls to minimum
- Support
  - OpenGL and shaders – Core, version **4.5+**
    - first line of the shader must be (at least)

      `#version 450 core`

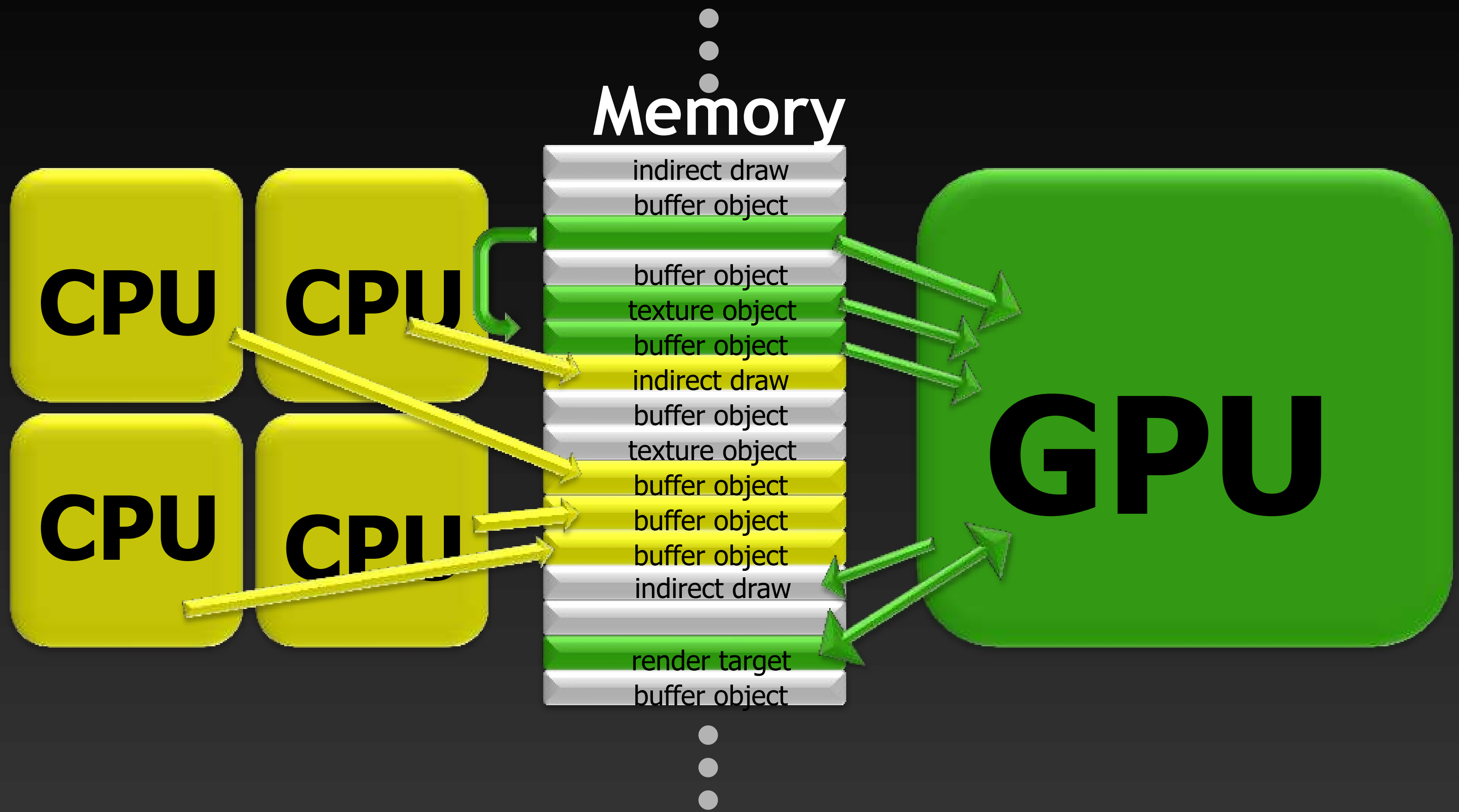
```
if (!GLEW_ARB_direct_state_access)
     throw std::runtime_error("No DSA :-(");
```

Classic OpenGL Model



Direct Drawing Commands
(via the command fifo)

Efficient OpenGL Model



- CPU and GPU decoupled
- CPU writes to memory – multithreaded (no API calls!)
- GPU writes to memory – still no API
- GPU reads from memory – minimal API
- SPEEDUP 5x-15x

One Textured Triangle: Compatibility vs. Core profile

```
glBegin(GL_TRIANGLES);
  glTexCoord2f(0.0f, 0.0f);
  glVertex2i(200, 50);

  glTexCoord2f(0.0f, 1.0f);
  glVertex2i(50, 250);

  glTexCoord2f(1.0f, 1.0f);
  glVertex2i(350, 250);
glEnd();
```

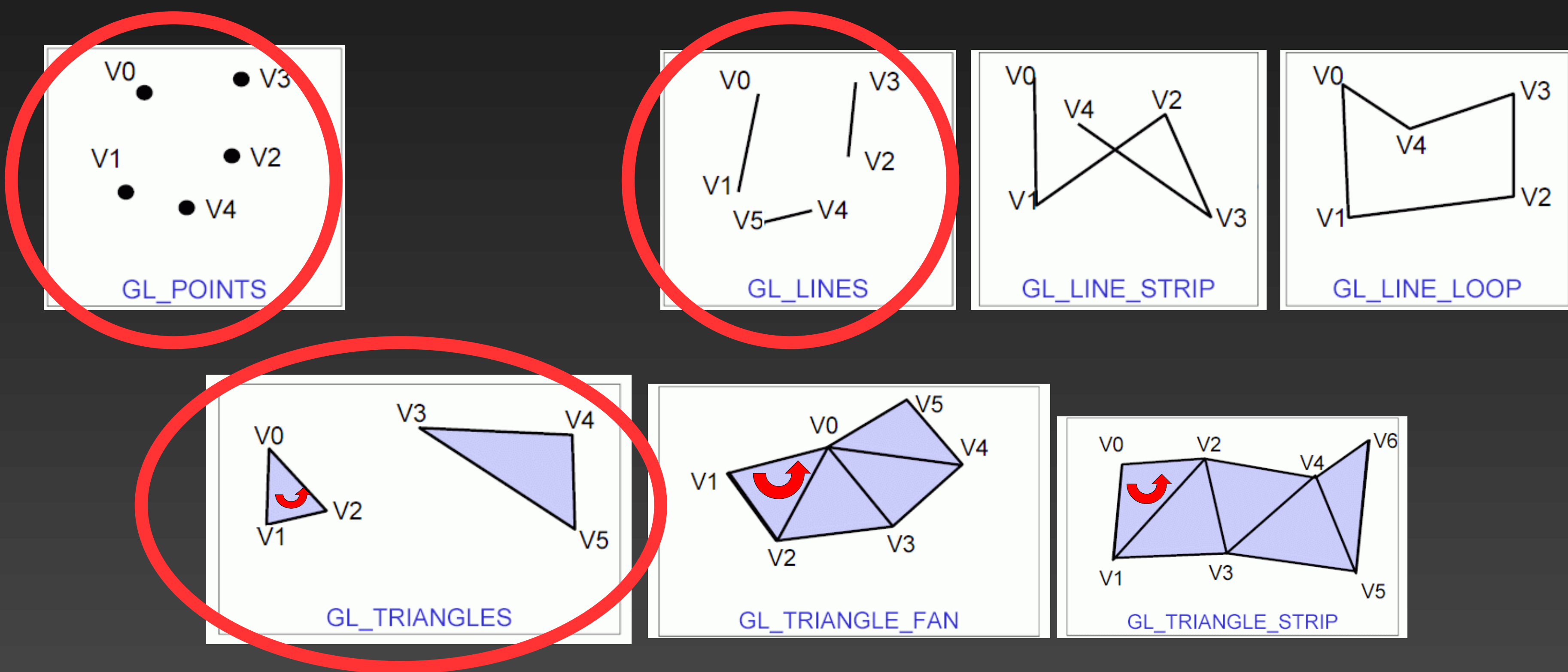
```
#version 460 core
in vec3 aPos; // Positions/Coordinates
in vec2 aTex; // Texture Coordinates

uniform mat4 uProj_m,uV_m,uM_m;

out VS_OUT {
    vec3 color; // Outputs color for FS
    vec2 texCoord; // Outputs texture coordinates for FS
} vs_out;

void main() {
    // Outputs coordinates of all vertices
    gl_Position = uProj_m * uV_m * uM_m * vec4(aPos,1.0f);

    // Assigns the colors somehow
    vs_out.color = vec3(1.0); //white

    // Pass the texture coordinates to "texCoord" for FS
    vs_out.texCoord = aTex;
}
```

```
#version 460 core
in VS_OUT {
    vec3 color; // color for FS
    vec2 texCoord; // texture coordinates for FS
} fs_in;

uniform sampler2D tex0; // texture unit from C++

out vec4  FragColor; // Final output

void main() {

   FragColor = fs_in.color * texture(tex0, fs_in.texcoord);
}
```

```
        //existing data
        struct my_vertex {
                glm::vec3 position;  // Vertex
                glm::vec2 texcoord;  // Texcoord0
        };
        std::vector<my_vertex> vertices = {
                        { {200,50,0}, {0,0} },
                        { {50,250,0}, {0,1} },
                        { {350,250,0}, {1,1} } };
        std::vector<GLuint> indices = {0,1,2};

        //GL names for Array and Buffers Objects
        GLuint VAO, VBO, EBO;
//-----------------------------------------------------------------------------------
    // create and use shaders
    GLuint VS_h, FS_h, prog_h;
    VS_h = glCreateShader(GL_VERTEX_SHADER);
    FS_h = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(VS_h, 1, &VS_string, NULL);
    glShaderSource(FS_h, 1, &FS_string, NULL);
    glCompileShader(VS_h);
    glCompileShader(FS_h);
    prog_h = glCreateProgram();
    glAttachShader(prog_h, VS_h);
    glAttachShader(prog_h, FS_h);
    glLinkProgram(prog_h);
    glUseProgram(prog_h);
//-----------------------------------------------------------------------------------
    // Create the VAO and VBO
    glCreateVertexArrays(1, &VAO);

    // Set Vertex Attribute to explain OpenGL how to interpret the data
    GLint position_attrib_location = glGetAttribLocation(prog_h, "aPos");
    glVertexArrayAttribFormat(VAO, position_attrib_location, 3, GL_FLOAT, GL_FALSE,
                              offsetof(my_vertex, position)));
    glVertexArrayAttribBinding(VAO, position_attrib_location, 0);
    glEnableVertexArrayAttrib(VAO, position_attrib_location);

    // Set and enable Vertex Attribute for Texture Coordinates
    GLint texture_attrib_location = glGetAttribLocation(prog_h, "aTex");
    glVertexArrayAttribFormat(VAO,texture_attrib_location,2,GL_FLOAT,GL_FALSE,
                              offsetof(my_vertex, texcoord)));
    glVertexArrayAttribBinding(VAO, texture_attrib_location, 0);
    glEnableVertexArrayAttrib(VAO, texture_attrib_location);

    // Create and fill data
    glCreateBuffers(1, &VBO); // Vertex  Buffer Object
    glCreateBuffers(1, &EBO); // Element Buffer Object

    glNamedBufferData(VBO, vertices.size()*sizeof(vertex), vertices.data(), GL_STATIC_DRAW);
    glNamedBufferData(EBO, indices.size()*sizeof(GLuint), indices.data(), GL_STATIC_DRAW);

    //Connect together
    glVertexArrayVertexBuffer(VAO, 0, VBO, 0, sizeof(vertex));
    glVertexArrayElementBuffer(VAO, EBO);
//-----------------------------------------------------------------------------------


    // USE buffers
    glBindVertexArray(VAO);

    glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0)
```

Geometrická primitiva

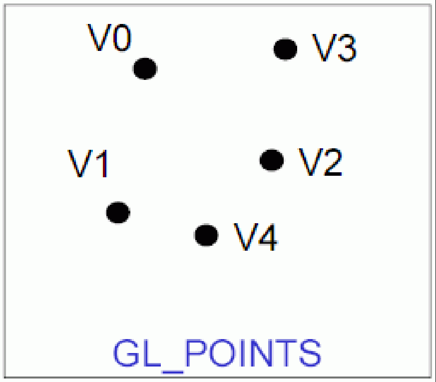
- 7 primitiv, zadávány pomocí vertexů [x,y,z,w]
- jen 3 primitiva skutečně v HW → překlad



Grafická primitiva

- Zadávány pomocí série vertexů
- Při nedostatečném počtu vertexů
  - nedefinované chování
    - nic se nevykreslí
    - nevykreslí se jen poslední část
    - (jakékoliv chybné chování)…

Vlastnosti bodů



- Teoreticky nekonečně malý, zadán jako *float*
  - několik bodů může vyústit v jeden pixel
    glPointSize(GLfloat)
  - standardně 1.0 (jeden pixel)
- Podle nastavení antialiasingu
  - čtverec
    glDisable(GL_POINT_SMOOTH)
  - kruh s rozmazaným okrajem (ne vždy podporováno)
    glEnable(GL_POINT_SMOOTH)
- Místo velkých (složitých) bodů – POINT SPRITE

Vlastnosti úseček

- Určené koncovými body
- Šířka čáry

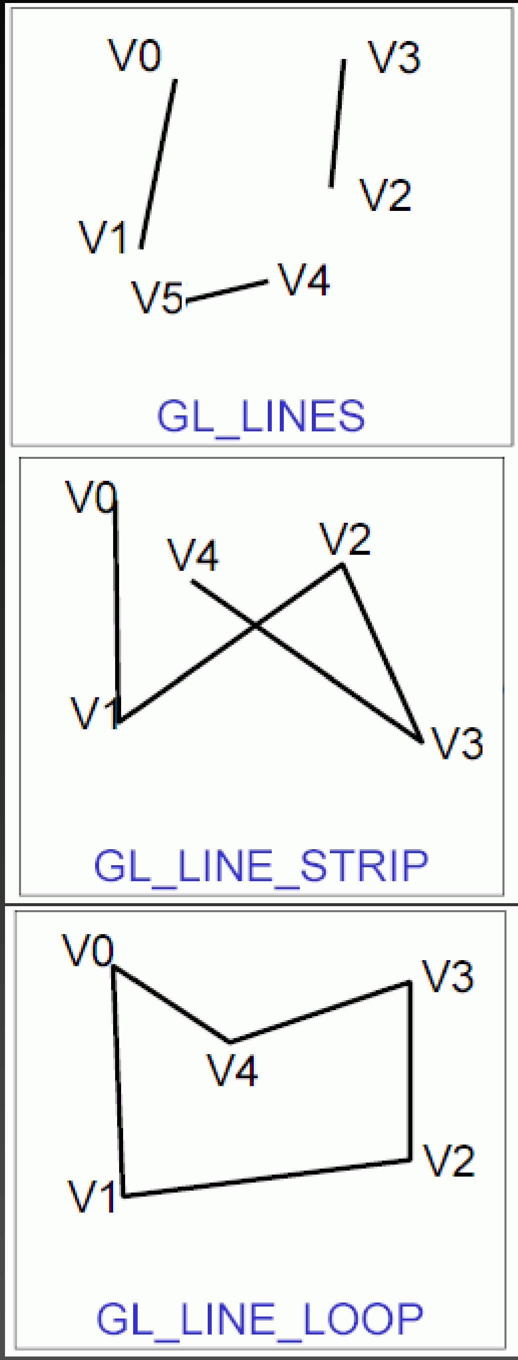
    glLineWidth(GLfloat)

    - standardně 1.0 (jeden pixel)

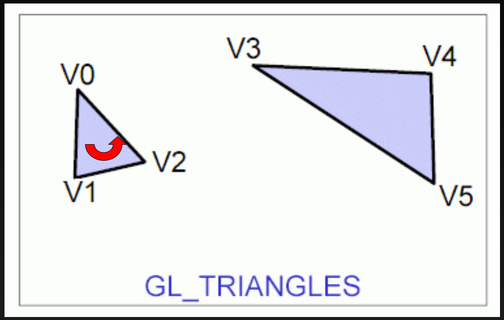
- Antialiasing určuje i zakončení
    - vertikální nebo horizontální konec

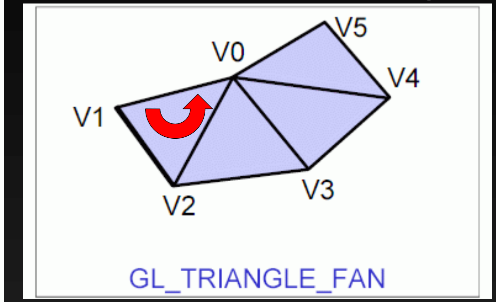
      glDisable(GL_LINE_SMOOTH)
    - jako natočený obdélník

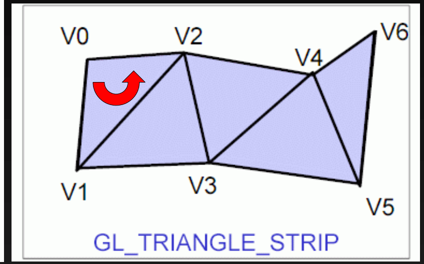
      glEnable(GL_LINE_SMOOTH)



Vlastnosti polygonů







- bez průsečíků, konvexní, v jedné rovině
  - případně teselace
  - nejlépe trojúhelník
- Čelní a zadní strana
  - určené pořadím zadávání vertexů (prav. pravé ruky)

Vlastnosti polygonů

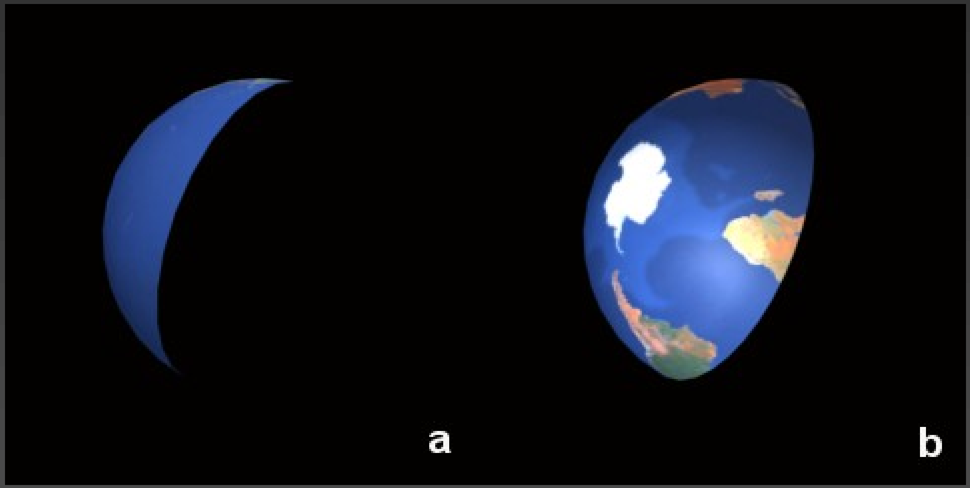
- ATRIBUTY pro jednotlivé vertexy
  - poloha, barva, normála, texturovací souřadnice...
- Čelní a zadní strana různé vlastnosti vykreslování
  - body, hrany, vyplněná plocha

    glPolygonMode(face, mode)

    face: GL_FRONT_AND_BACK, GL_FRONT, GL_BACK
    mode: GL_POINT, GL_LINE, GL_FILL
  - ořez

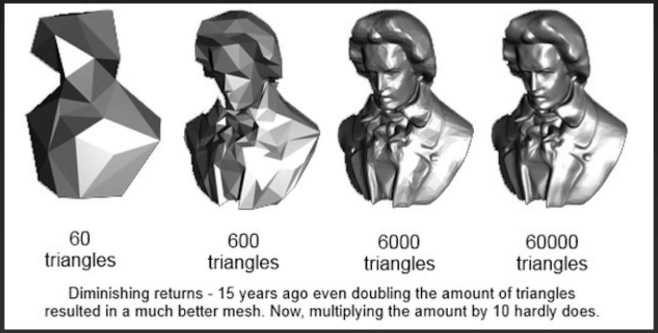
    glCullFace(mode)

    GL_FRONT_AND_BACK, GL_FRONT, GL_BACK



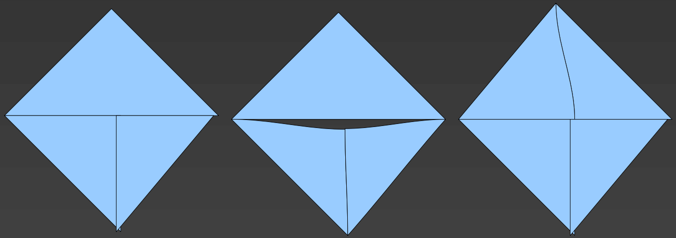
Doporučení

- Shodná orientace, CCW
- Trojúhelníky (konvexní, vždy v rovině)
- Kompromis kvalita X množství polygonů
  - adaptivní dělení
    - podle křivosti
    - podle vzdálenosti
    - podle hrany
      - tečna - skalární součin se blíží nule



Diminishing returns - 15 years ago even doubling the amount of triangles resulted in a much better mesh. Now, multiplying the amount by 10 hardly does.

- Nepoužívat T křížení!



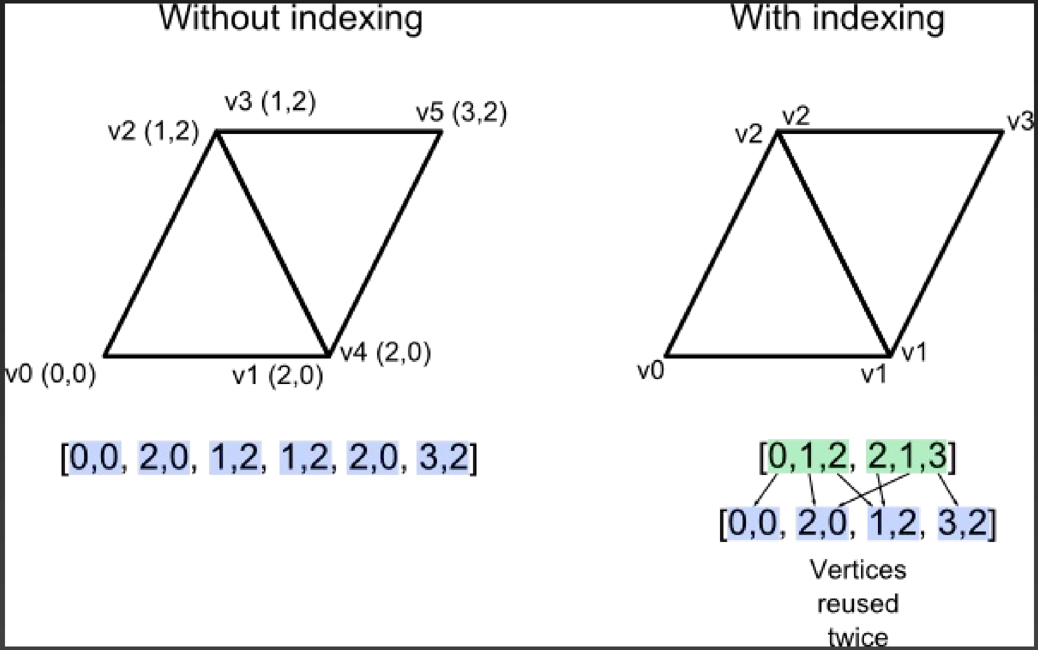
- Pro napojení použít přesně stejná čísla

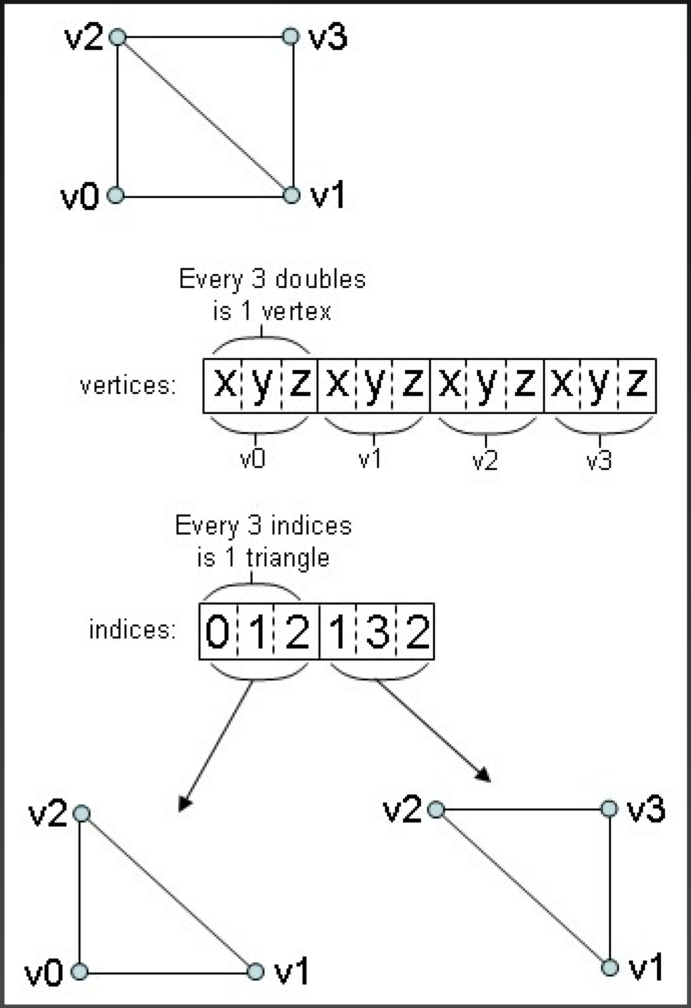
$$(a + b) + c \neq a + (b + c)$$

Pole souřadnic vs. pole indexů souřadnic

3D

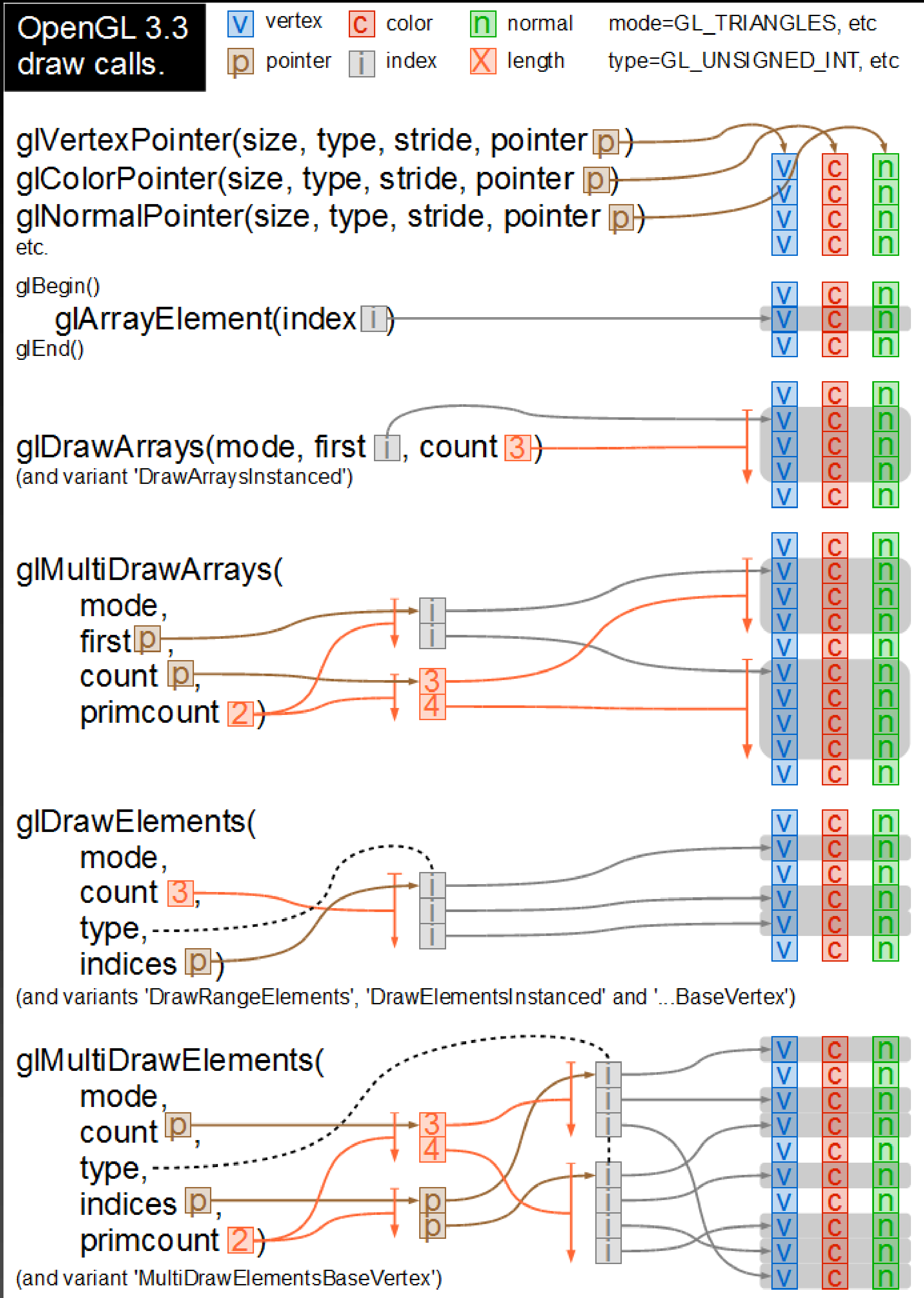
2D





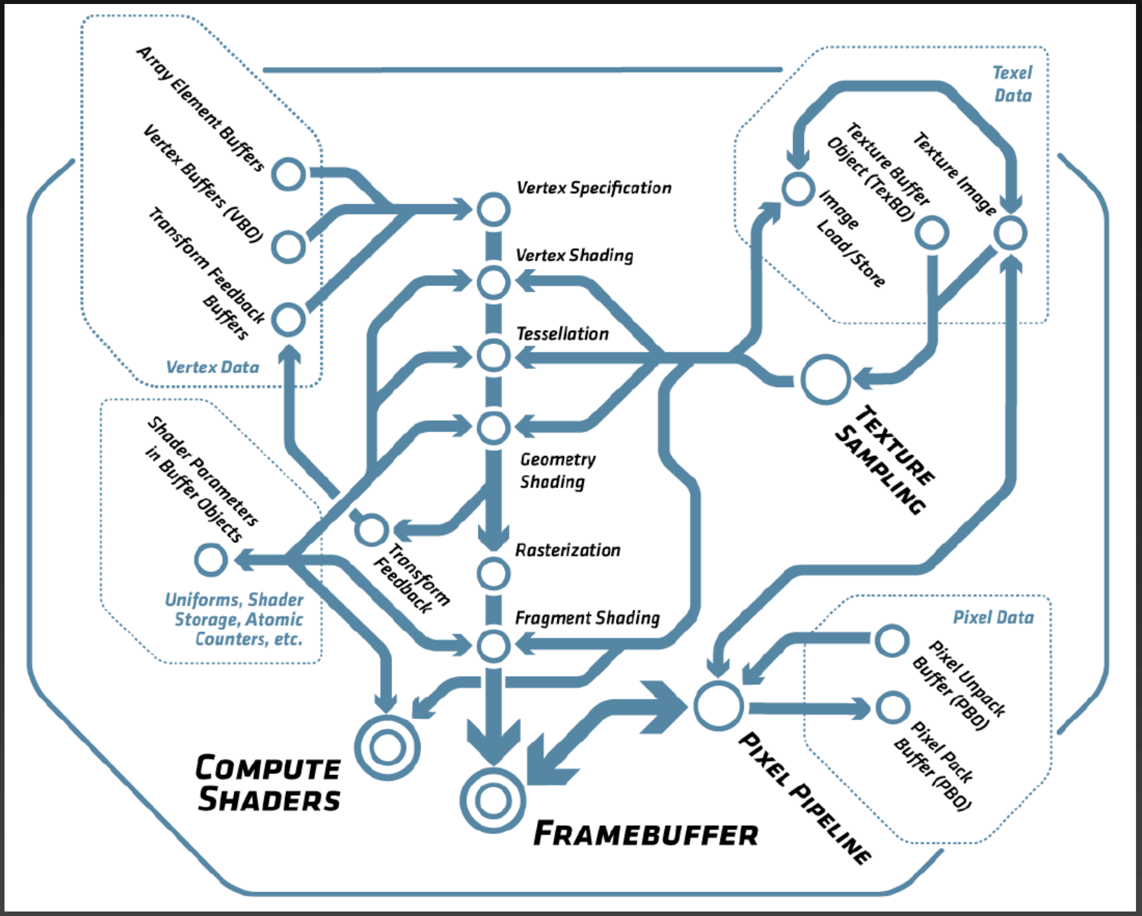
Některá vykreslovací volání

- zkus: https://docs.gl/
  - GL 4.5
  - inkrementální hledání
    - glDraw



Using buffers for vertex data

- Used with shaders
- Linear memory in GPU
- Identified by ID
  - allocate
    glCreateBuffers()
  - obtain data
    - fill to GPU
      glNamedBufferData()
    - map CPU → GPU
      glMapNamedBuffer()
  - draw
    glDrawArrays(), glDrawElements()
- Buffers are in GPU mem
  (unlike arrays)
  - fast
  - allocation can fail
    (no GPU mem paging)
  - changing data is not straightforward



Vertex data

- Vertex Array Object = VAO
  - Container for grouping of attribute settings, placement etc.
  - Single rebinding by *glBindVertexArray(VAO2)* prepares vertex data of other object for draw
- Generic array
  - any data, YOU must specify how to interpret
    glVertexArrayAttribFormat()
    - define meaning of specific attribute, data types etc.
    - attribute on slot (position) 0 ≈ vertex [xyz] position

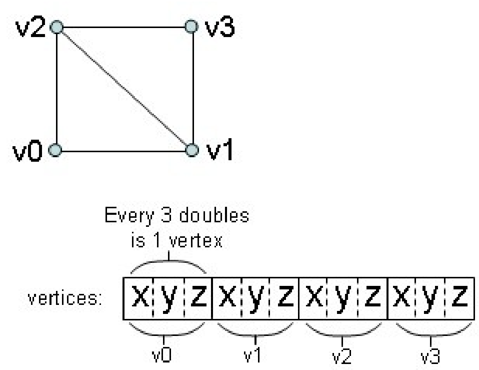
    glVertexArrayAttribBinding()
    - associate attribute and vertex buffer binding for VAO

    glEnableVertexArrayAttrib()
    - enable usage of the attribute at specified slot

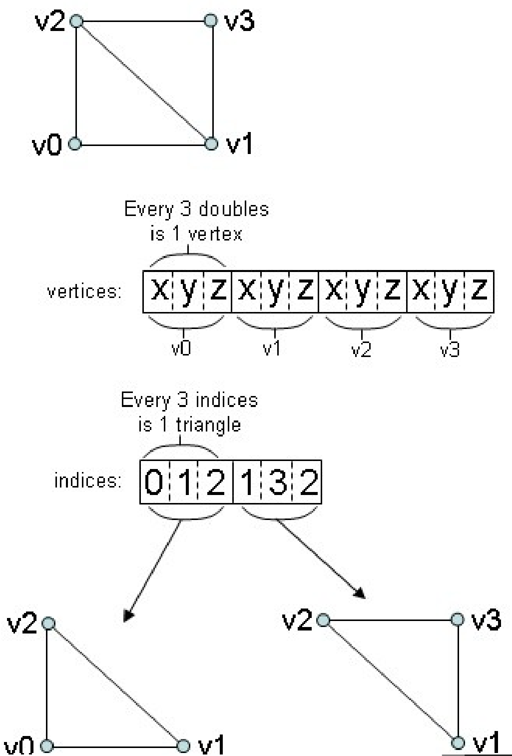
VAO – direct coordinates

- Only vertices (as glm::vec3) = attribute „position“ (VAO pointer slot = 0)

```
//existing data
std::vector<glm::vec3> vertices = { {…,…,…}, {…,…,…}, … };

//GL names for Array and Buffers Objects
GLuint VAO, VBO;

// Generate the VAO
glCreateVertexArrays(1, &VAO);

// Set Vertex Attribute to explain OpenGL how to interpret the data
GLint position_attrib_location = glGetAttribLocation(prog_h, "aPos");
glVertexArrayAttribFormat(VAO, position_attrib_location, 3, GL_FLOAT, GL_FALSE, 0);
glVertexArrayAttribBinding(VAO, position_attrib_location, 0);
glEnableVertexArrayAttrib(VAO, position_attrib_location);

// Create and fill data
glCreateBuffers(1, &VBO); // Vertex  Buffer Object
glNamedBufferData(VBO, vertices.size()*sizeof(sizeof(glm::vec3)), vertices.data(), GL_STATIC_DRAW);

// Connect together
glVertexArrayVertexBuffer(VAO, 0, VBO, 0, sizeof(vertex));


  // USE
  glUseProgram(shaderProgram);
  glBindVertexArray(VAO);

  glDrawArrays(GL_TRIANGLES, 0, vertices.size());
```



VAO
indirect vertex access

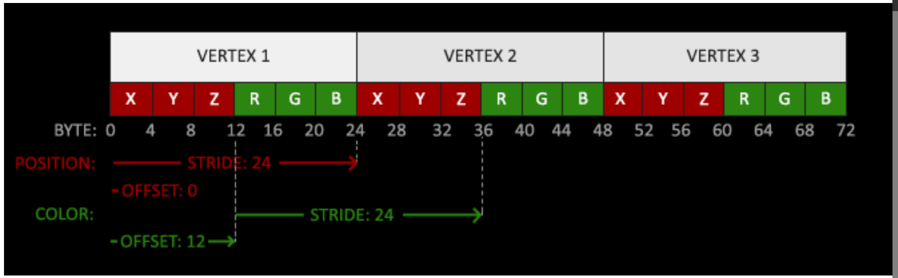



```
//existing data
std::vector<glm::vec3> vertices = {{…,…,…},{…,…,…},… };
std::vector<GLuint> indices = {…,…,… };

//GL names for Array and Buffers Objects
GLuint VAO, VBO, EBO;

// Generate the VAO
glCreateVertexArrays(1, &VAO);

// Set Vertex Attribute to explain OpenGL how to interpret the data
GLint position_attrib_location = glGetAttribLocation(prog_h, "aPos");
glVertexArrayAttribFormat(VAO, position_attrib_location, 3, GL_FLOAT, GL_FALSE, 0);
glVertexArrayAttribBinding(VAO, position_attrib_location, 0);
glEnableVertexArrayAttrib(VAO, position_attrib_location);

// Create and fill data
glCreateBuffers(1, &VBO); // Vertex  Buffer Object
glNamedBufferData(VBO, vertices.size()*sizeof(sizeof(glm::vec3)), vertices.data(), GL_STATIC_DRAW);

glCreateBuffers(1, &EBO); // Element Buffer Object
glNamedBufferData(EBO, indices.size() * sizeof(GLuint), indices.data(), GL_STATIC_DRAW);

// Connect together
glVertexArrayVertexBuffer(VAO, 0, VBO, 0, sizeof(vertex));
glVertexArrayElementBuffer(VAO, EBO);

// USE
glUseProgram(shaderProgram);
glBindVertexArray(VAO);

glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
```

VAO – additional vertex attributes (normals, etc.)

```
        //existing data
        struct my_vertex {
                glm::vec3 position;  // Vertex
                glm::vec3 normal;    // Normal
                glm::vec2 texcoord;  // Texcoord0
        };
        std::vector<my_vertex> vertices = { {{x,y,z}, {nx,ny,nz}, {s,t}}, … };
        …
    // Set Vertex Attribute to explain OpenGL how to interpret the data
    GLint position_attrib_location = glGetAttribLocation(prog_h, "aPos");
    glVertexArrayAttribFormat(VAO, position_attrib_location, 3, GL_FLOAT, GL_FALSE, offsetof(my_vertex, position));
    glVertexArrayAttribBinding(VAO, position_attrib_location, 0);
    glEnableVertexArrayAttrib(VAO, position_attrib_location);

    // Set end enable Vertex Attribute for Normal
    GLint normal_attrib_location = glGetAttribLocation(prog_h, "aNormal");
    glVertexArrayAttribFormat(VAO, normal_attrib_location, 3, GL_FLOAT, GL_FALSE, offsetof(my_vertex, normal));
    glVertexArrayAttribBinding(VAO, normal_attrib_location, 0);
    glEnableVertexArrayAttrib(VAO, normal_attrib_location);

    // Set end enable Vertex Attribute for Texture Coordinates
    GLint tex_attrib_location = glGetAttribLocation(shader_prog_ID, "aTex"); //name in shader src
    glVertexArrayAttribFormat(VAO, tex_attrib_location, 2, GL_FLOAT, GL_FALSE, offsetof(my_vertex, texcoord));
    glVertexArrayAttribBinding(VAO, tex_attrib_location, 0);
    glEnableVertexArrayAttrib(VAO, tex_attrib_location);
        …
```

```
#version 460 core
in vec3 aPos;    // Positions/Coordinates
in vec3 aNormal; // Normals
in vec2 aTex;    // Texture Coordinates

uniform mat4 uProj_m,uV_m,uM_m;

out VS_OUT {
       vec3 color; // Outputs color for FS
       out vec2 texCoord; // Outputs texture coordinates for FS
} vs_out;

void main() {
       // Outputs coordinates of all vertices
       gl_Position = uProj_m * uV_m * uM_m * vec4(aPos,1.0f);

       // Assigns the colors somehow
       vs_out.color = vec3(1.0); //white

       // Pass the texture coordinates to "texCoord" for FS
       vs_out.texCoord = aTex;
}
```



Single textured triangle: Core profile

```
        //existing data
        struct my_vertex {
                glm::vec3 position;  // Vertex
                glm::vec2 texcoord;  // Texcoord0
        };
        std::vector<my_vertex> vertices = {
                        { {200,50,0}, {0,0} },
                        { {50,250,0}, {0,1} },
                        { {350,250,0}, {1,1} } };
        std::vector<GLuint> indices = {0,1,2};

        //GL names for Array and Buffers Objects
        GLuint VAO, VBO, EBO;
//-----------------------------------------------------------------------------------
    // create and use shaders
    GLuint VS_h, FS_h, prog_h;
    VS_h = glCreateShader(GL_VERTEX_SHADER);
    FS_h = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(VS_h, 1, &VS_string, NULL);
    glShaderSource(FS_h, 1, &FS_string, NULL);
    glCompileShader(VS_h);
    glCompileShader(FS_h);
    prog_h = glCreateProgram();
    glAttachShader(prog_h, VS_h);
    glAttachShader(prog_h, FS_h);
    glLinkProgram(prog_h);
    glUseProgram(prog_h);
//-----------------------------------------------------------------------------------
    // Create the VAO and VBO
    glCreateVertexArrays(1, &VAO);
    // Set Vertex Attribute to explain OpenGL how to interpret the data
    GLint position_attrib_location = glGetAttribLocation(prog_h, "aPos");
    glVertexArrayAttribFormat(VAO, position_attrib_location, 3, GL_FLOAT, GL_FALSE,
                              offsetof(my_vertex, position)));
    glVertexArrayAttribBinding(VAO, position_attrib_location, 0);
    glEnableVertexArrayAttrib(VAO, position_attrib_location);
    // Set and enable Vertex Attribute for Texture Coordinates
    GLint texture_attrib_location = glGetAttribLocation(prog_h, "aTex");
    glVertexArrayAttribFormat(VAO,texture_attrib_location,2,GL_FLOAT,GL_FALSE,
                              offsetof(my_vertex, texcoord)));
    glVertexArrayAttribBinding(VAO, texture_attrib_location, 0);
    glEnableVertexArrayAttrib(VAO, texture_attrib_location);
    // Create and fill data
    glCreateBuffers(1, &VBO); // Vertex  Buffer Object
    glCreateBuffers(1, &EBO); // Element Buffer Object
    glNamedBufferData(VBO, vertices.size()*sizeof(vertex),
                      vertices.data(),GL_STATIC_DRAW);
    glNamedBufferData(EBO, indices.size()*sizeof(GLuint),
                      indices.data(),GL_STATIC_DRAW);
    //Connect together
    glVertexArrayVertexBuffer(VAO, 0, VBO, 0, sizeof(vertex));
    glVertexArrayElementBuffer(VAO, EBO);
//-----------------------------------------------------------------------------------
    // USE buffers
    glBindVertexArray(VAO);
    glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0)
```

```
#version 460 core
in vec3 aPos; // Positions/Coordinates
in vec2 aTex; // Texture Coordinates

uniform mat4 uProj_m,uV_m,uM_m;

out VS_OUT {
        vec3 color; // Outputs color for FS
        vec2 texCoord; // Outputs texture coordinates for FS
} vs_out;

void main() {
        // Outputs coordinates of all vertices
        gl_Position = uProj_m * uV_m * uM_m * vec4(aPos,1.0f);

        // Assigns the colors somehow
        vs_out.color = vec3(1.0); //white

        // Pass the texture coordinates to "texCoord" for FS
        vs_out.texCoord = aTex;
}
```

```
#version 460 core
in VS_OUT {
        vec3 color; // color for FS
        vec2 texCoord; // texture coordinates for FS
} fs_in;

uniform sampler2D tex0; // texture unit from C++

out vec4  FragColor; // Final output

void main() {

        FragColor = fs_in.color * texture(tex0, fs_in.texcoord);
}
```