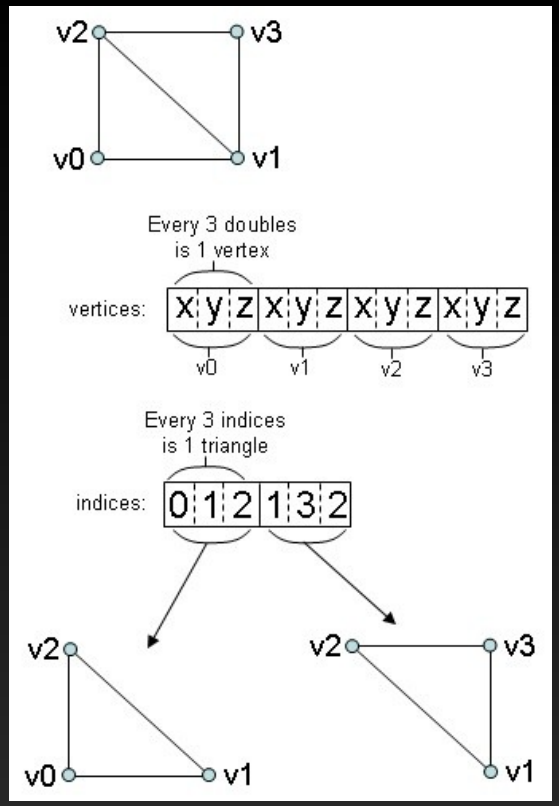
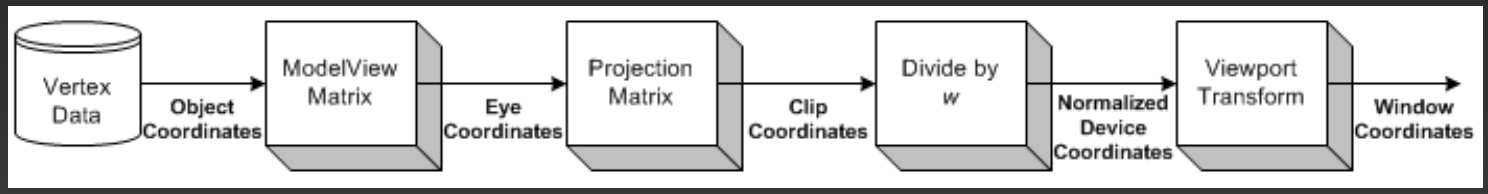
Opakování



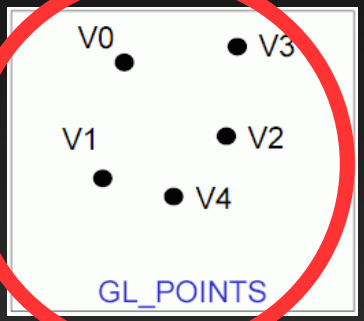
- Reprezentace
  - 3D rastr
  - obálka
    - vrchol (vertex)
    - hrana (edge)
    - ploška (face)
- 3D zobrazování

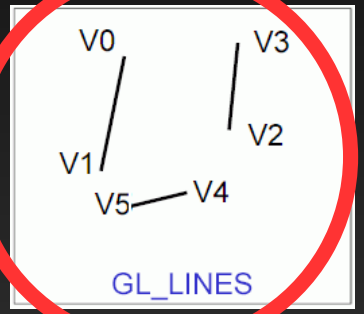


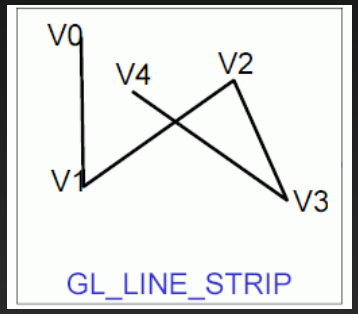
  - načtení a transformace souřadnic zadaných vertexů
  - rasterizace
  - výpočet barvy fragmentu
  - průhlednost a zakrývání podle Z

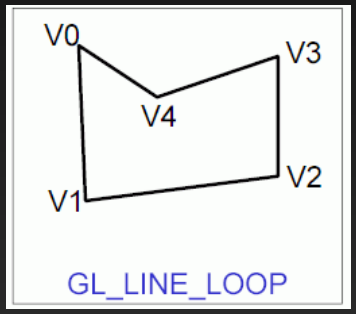
Geometrická primitiva

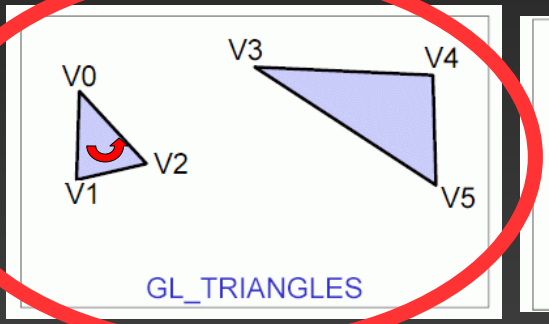
- 10 primitiv, zadávány pomocí vertexů [x,y,z,w]
- jen 3 primitiva skutečně v HW → překlad

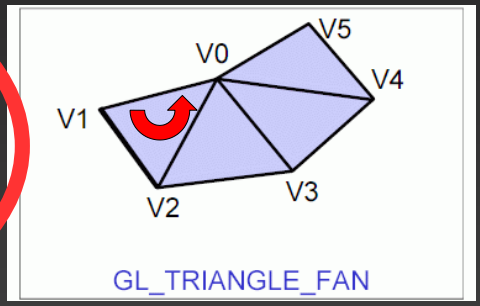


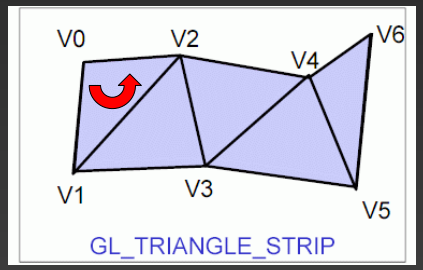








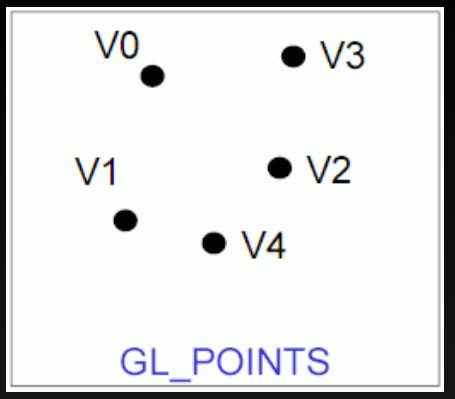




Grafická primitiva

- 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

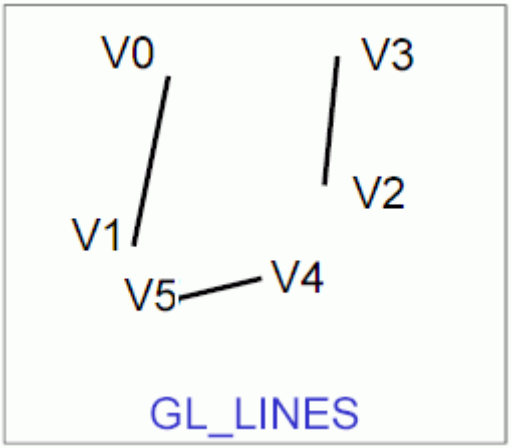
- Určené koncovými body
- Šířka čáry

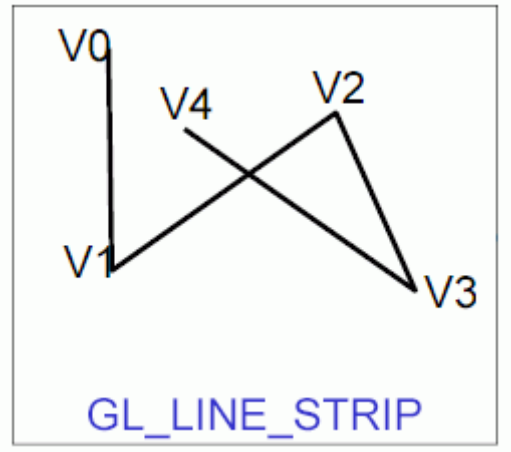
  glLineWidth(GLfloat)
  - standardně 1.0 (jeden pixel)

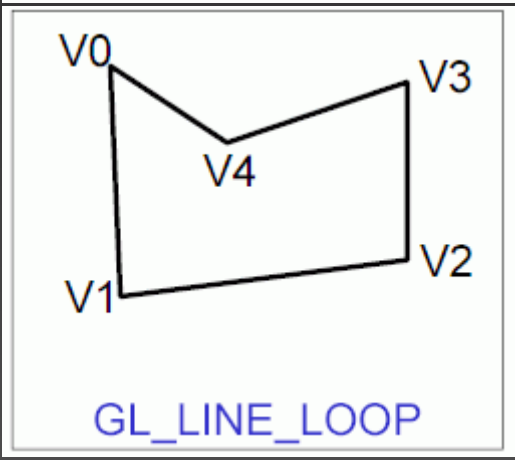
- Antialiasing určuje i zakončení
  - vertikální nebo horizontální konec

    glDisable(GL_LINE_SMOOTH)
  - jako natočený obdélník

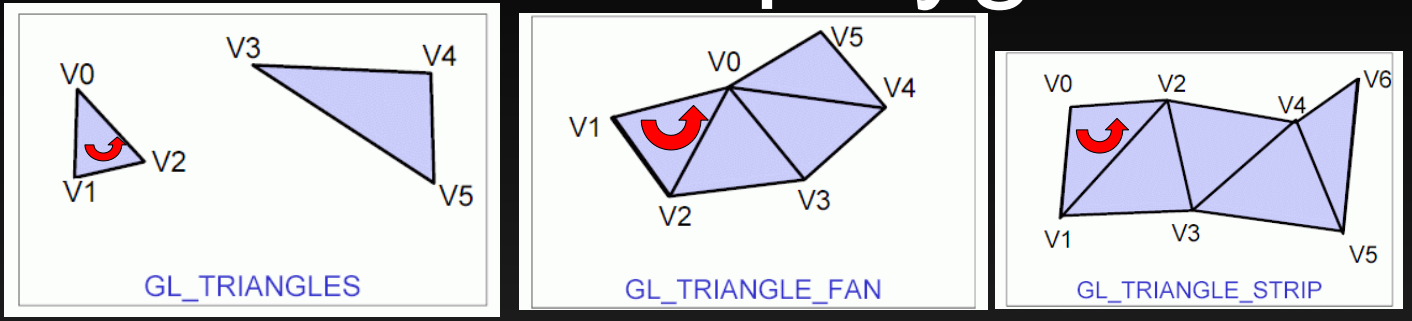
    glEnable(GL_LINE_SMOOTH)







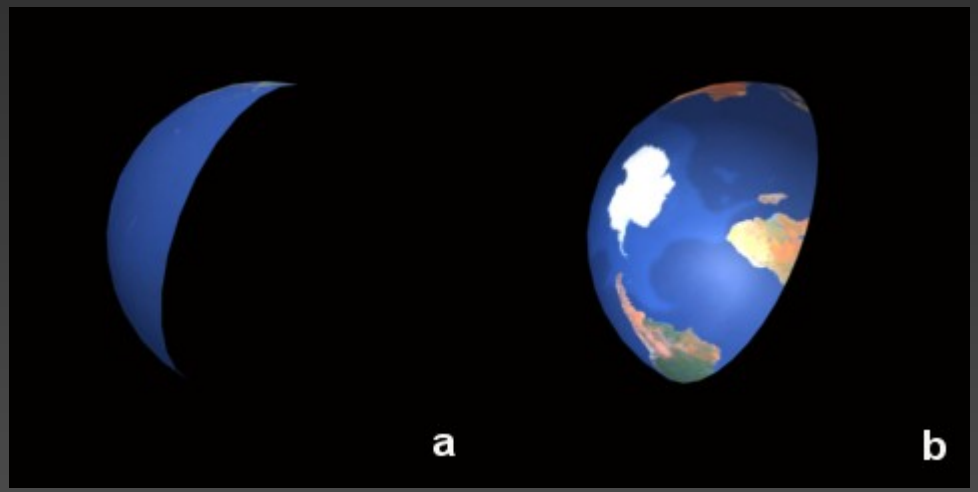
Vlastnosti polygonů



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

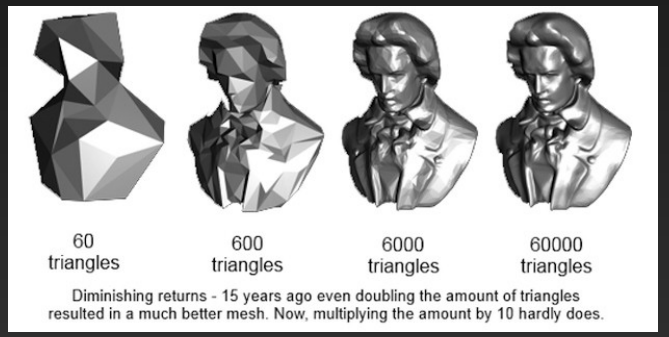
Vlastnosti polygonů

- 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

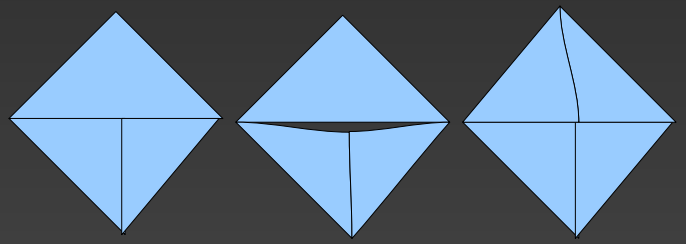


Doporučení

- 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



- Nepoužívat T křížení!

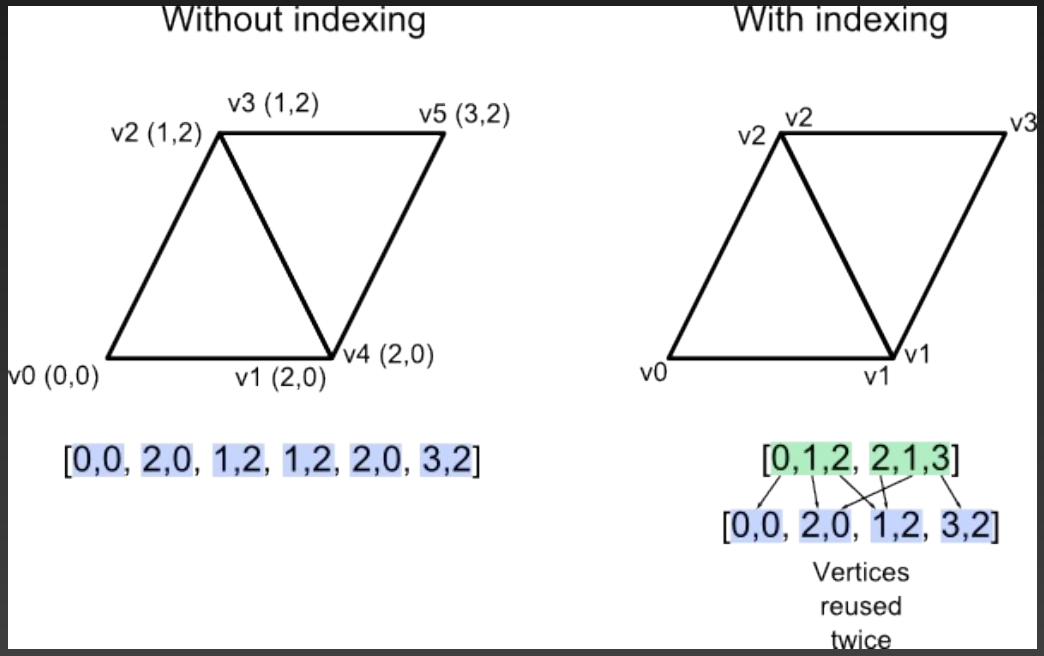


- Pro napojení použít přesně stejná čísla

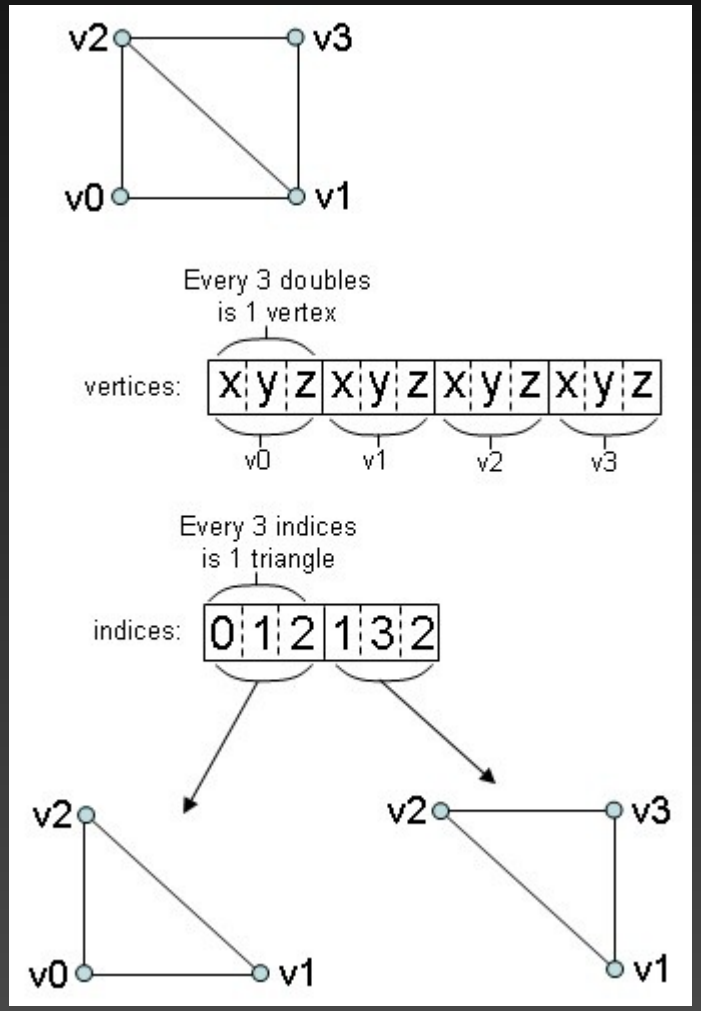
  $(a + b) + c \neq a + (b + c)$

Pole souřadnic vs. pole indexů souřadnic

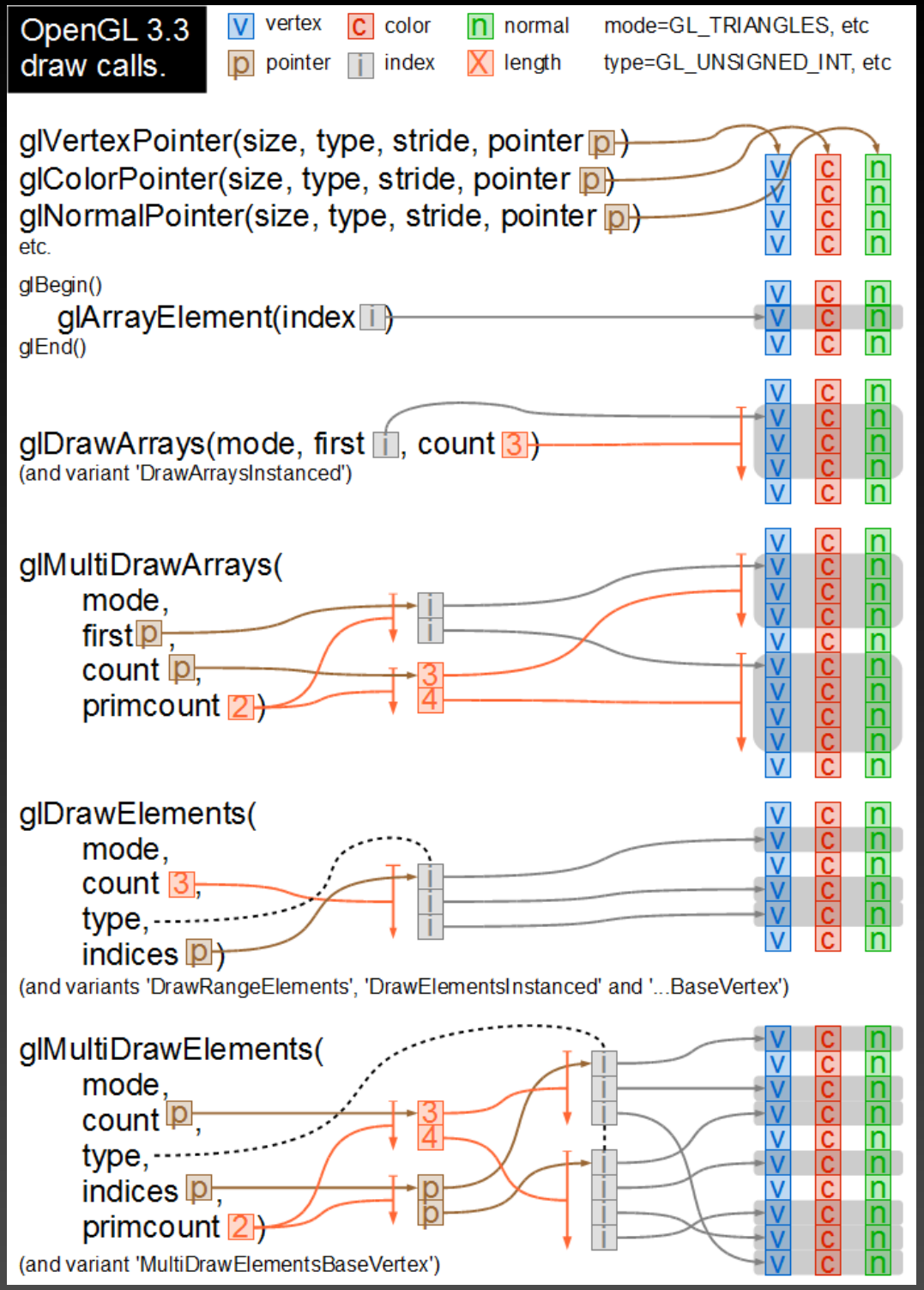
2D



3D

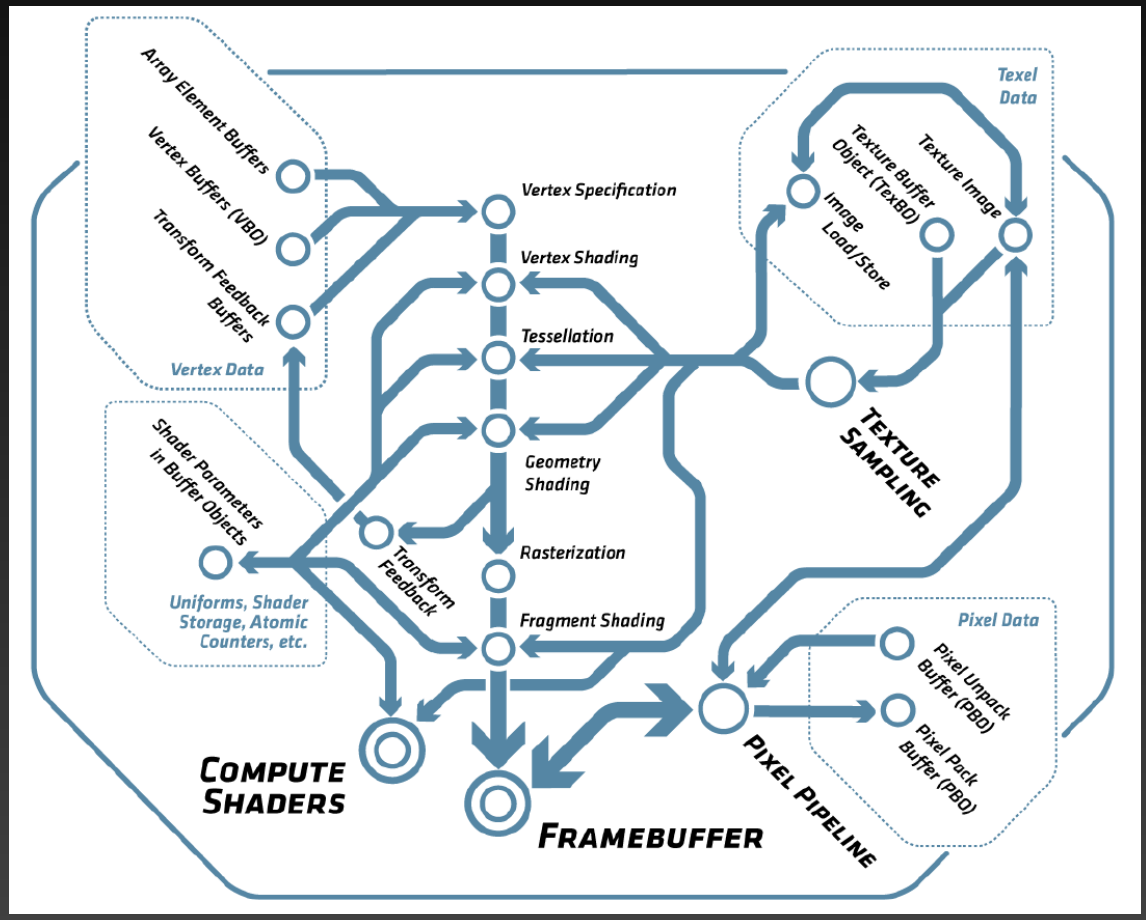


Některá vykreslovací volání



Using buffers for vertex data

- Used with shaders
- Linear memory in GPU
- Identified by ID
  - allocate
    glCreateBuffers(), glGenBuffers()
  - activate buffer
    glBindBuffer()
  - obtain data
    - fill to GPU
      glBufferData()
    - map CPU → GPU
      glMapBuffer()
  - draw
    glDrawArrays(), glDrawElements(),…
- Buffers are in GPU mem
  - fast
  - allocation can fail
    (no GPU mem paging)
  - changing data is not straightforward



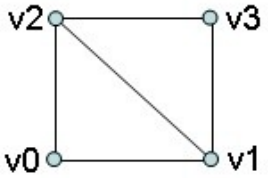
Vertex data

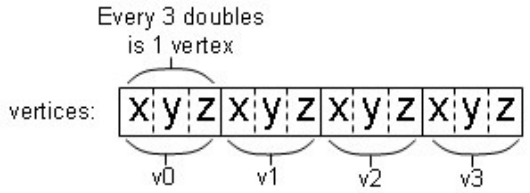
- Vertex Array Object = VAO
  - Container for grouping of attribute settings, placement etc.
  - Single rebinding by glBindVertexArray(VAO2) prepares vertex data of other object for draw
- Generic array
  - any data, YOU must specify how to interpret
    glVertexAttrib()
      - define meaning of specific attribute, data types etc.
      - attribute on position 0 ≈ position ≈ glVertex()
    glVertexAttribPointer()
      - array of attributes; vertices and others (interleaved)
    glEnableVertexAttribArray()
      - enable usage of the attribute at specified slot

VAO – direct coordinates

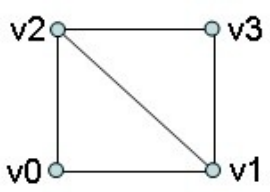
- Only vertices as glm::vec3 (VAO pointer slot = 0)

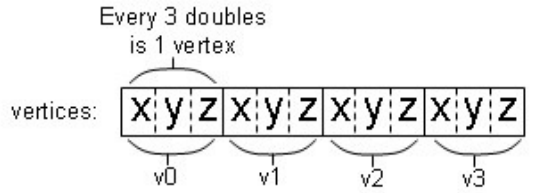
```
//existing data
std::vector<glm::vec3> vertices = { };
std::vector<GLuint> indices = { };

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

// Generate the VAO and VBO
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);
// Bind VAO (set as the current)
glBindVertexArray(VAO);
// Bind the VBO, set type as GL_ARRAY_BUFFER
glBindBuffer(GL_ARRAY_BUFFER, VBO);
// Fill-in data into the VBO
glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(glm::vec3), vertices.data(), GL_STATIC_DRAW);
// Set Vertex Attribute to explain OpenGL how to interpret the VBO
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(glm::vec3), reinterpret_cast<void*>(0));
// Enable the Vertex Attribute 0 = position
glEnableVertexAttribArray(0);
// Bind VBO and VAO to 0 to prevent unintended modification of VAO,VBO
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);

// USE
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
glDrawArrays(GL_TRIANGLES, 0, vertices.size());
```

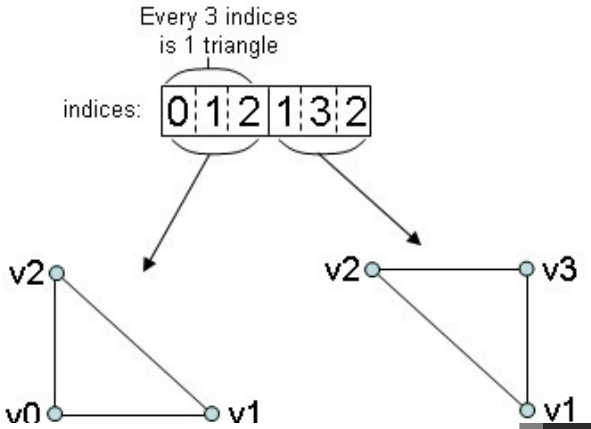




VAO – indirect








```
//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 and VBO
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);
glGenBuffers(1, &EBO);
// Bind VAO (set as the current)
glBindVertexArray(VAO);
// Bind the VBO, set type as GL_ARRAY_BUFFER
glBindBuffer(GL_ARRAY_BUFFER, VBO);
// Fill-in data into the VBO
glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(vertex), vertices.data(), GL_STATIC_DRAW);
// Bind EBO, set type GL_ELEMENT_ARRAY_BUFFER
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
// Fill-in data into the EBO
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(GLuint), indices.data(), GL_STATIC_DRAW);
// Set Vertex Attribute to explain OpenGL how to interpret the VBO
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(vertex), reinterpret_cast<void*>(0 + offsetof(vertex, position)));
// Enable the Vertex Attribute 0 = position
glEnableVertexAttribArray(0);
// Bind VBO and VAO to 0 to prevent unintended modification of VAO,VBO
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);

// USE
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
```

VAO – additional vertex attributes (colors, normals, etc.)

```
	//existing data
	struct my_vertex {
		glm::vec3 position;  // Vertex
		glm::vec3 normal;    // Normal
		glm::vec2 texcoord;  // Texcoord0
	};
	std::vector<my_vertex> vertices = { };
	…
	// Set Vertex Attribute to explain OpenGL how to interpret the VBO
	glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(my_vertex), reinterpret_cast<void*>(0 + offsetof(my_vertex, position)));
	// Enable the Vertex Attribute 0 = position
	glEnableVertexAttribArray(0);

	// Set end enable Vertex Attribute 1 = Normal
	glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(my_vertex), reinterpret_cast<void*>(0 + offsetof(my_vertex, normal)));
	glEnableVertexAttribArray(1);

	// Set end enable Vertex Attribute 2 = Texture Coordinates
	glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(my_vertex), reinterpret_cast<void*>(0 + offsetof(my_vertex, texcoord)));
	glEnableVertexAttribArray(2);
	…
```

```
#version 330 core
layout (location = 0) in vec3 aPos; // Positions/Coordinates
layout (location = 1) in vec3 aNormal; // Normals
layout (location = 2) 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;
}
```

One textured triangle: Core profile

```
#version 330 core
layout (location = 0) in vec3 aPos; // Positions/Coordinates
layout (location = 1) 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 330 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;

	// Generate the VAO and VBO
	glGenVertexArrays(1, &VAO);
	glGenBuffers(1, &VBO);
	glGenBuffers(1, &EBO);
	// Bind VAO (set as the current)
	glBindVertexArray(VAO);
	// Bind the VBO, set type as GL_ARRAY_BUFFER
	glBindBuffer(GL_ARRAY_BUFFER, VBO);
	// Fill-in data into the VBO
	glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(vertex),
		vertices.data(), GL_STATIC_DRAW);
	// Bind EBO, set type GL_ELEMENT_ARRAY_BUFFER
	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
	// Fill-in data into the EBO
	glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(GLuint),
		indices.data(), GL_STATIC_DRAW);
	// Set Vertex Attribute to explain OpenGL how to interpret the VBO
	glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(vertex),
		(void*)(0 + offsetof(vertex, position)));
	// Enable the Vertex Attribute 0 = position
	glEnableVertexAttribArray(0);

	// Set end enable Vertex Attribute 1 = Texture Coordinates
	glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, sizeof(my_vertex),
	(void*)(0 + offsetof(my_vertex, normal)));
	glEnableVertexAttribArray(1);

	// Bind VBO and VAO to 0 to prevent unintended modification
	glBindBuffer(GL_ARRAY_BUFFER, 0);
	glBindVertexArray(0);
	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);

// 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);

	// USE buffers
	glBindVertexArray(VAO);

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