Přednášky

neděle 2. března 2025 17:01

Přednáška #1

- OpenGL - low-level development (aerospace, embedded systems etc.)
- WebGL - Web development (games, Google Maps/Earth)
- Unreal Engine - Contemporary Multiplatform Games Development

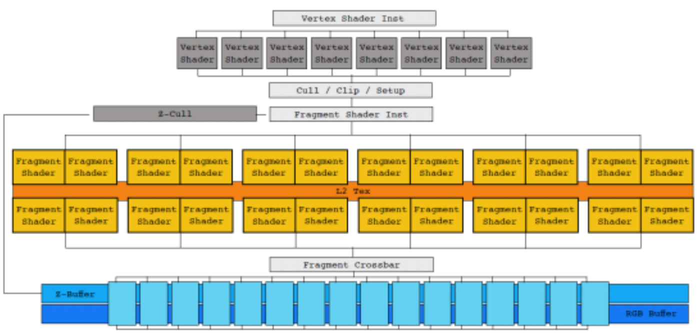
RENDERING PIPELINE

- The rendering pipeline outlines the stages involved in transforming 3D objects into a 2D image on the screen
  - **Vertex shader** - the first programmable stage in the graphics pipeline
    - Positioning - Transforms vertex positions from object space to world space, view space, and clip space.
    - Observer - Adjusts vertex positions based on the camera's position and orientation.
    - Camera FOV - Applies perspective projection based on the camera's field of view (FOV).
    - Connection - Links vertex data to transformations (e.g., model, view, and projection matrices).
    - Camera space - Converts vertices to camera (view) space.
    - Clip Space - Converts vertices to clip space, where they are ready for clipping.
    - **Output:** Transformed vertices
  - **Geometry shader** - Optional stage that processses entire primitives and can generate new geometry
    - Modifies or generates new primitives based on input vertices
    - Useful for effects like fur, grass or dynamic tessellation

      Output: New or Modified vertices
  - **Clipping** - Removes the vertices that are outside the visible area of the 3D world
  - **Screen mapping** - Converts clipped vertices from clip space to screen space
    - Maps clip space coordinates to the screen pixel coordinates
    - Adjusts for screen aspect ratio and resolution
  - **Primitive setup** - Prepares primitives for rasterization
    - Determines which pixels are covered by each primitive
    - Calculates interpolation parameters for attributes like color, texture coordinates and normals
  - **Rasterization** - Converts primitives into fragments for rendering
  - **Pixel shader** - Processes each fragment to determine its final color and other properties
    - Applies lighting, shading and texturing based on the scenes materials and lights
  - **Frame Buffer Blend** - Combines the fragment colors with the existing frame buffer

GPU IIN MODERN ERA

- Before nVidia Tesla



- This inage represents the architecture of modern GPUs. Specifically how the vertex and fragment shaders process data before being written to the frame buffer

HARDWARE TRACING: GTX VS RTX

-

Radiosity

- Global illumination method that stimulates the way light interacts with surfaces in a secene
- Focuses primarily on diffuse light reflection - where light is scattered evenly in all directions
- Often used in applications where soft, realistic lighting is important such as architectural visualization
- 

Ray tracing

- Ray tracing - RTX
- NVIDIA has introduced "RT cores" in their RTX series GPUs to accelarate ray tracing calculations
- RT Cores are specialized hardware units designed specifically for ray tracing

DEVELOPMENT

- Low-level development:
  - OpenQL + Qt
  - Metal + SwiftUI

- Frameworks and libraries
    - SpriteKit + SwiftUI
    - WebGL + Three.js
- Complex game engines:
    - Unreal Engine (mostly)

FEATURES OF OPENGL

- It is API for 2D and 3D graphics
- It is not complex SDK (Software Development Kit) like DirectX - meaning it focuses primarily on graphcis rendering rather than providing additional tools for input, audio and networking

Platform and language-independent

- OpenGL is designed to work across multiple platforms (Windows, Linux, macOS) etc.
- This contrasts APIs like DirectX (Windows and Xbox), Vulkan (Windows and Linux) & Metal (Apple platforms)

Purely procedural

- OpenGL is a procedural API meaning it uses functions to perform tasks - glDrawArrays()
- However, the code is written using openGL can be organized in object -oriented manner if desired

Versions of OpenGL

- OpenGL has many version and varients tailored for different use cases:
    - OpenGL ES -Lightweight version for embedded systems (mobile, consoles)
    - OpenGL SC - Safety-critical version for industries like aviation and automotive
    - WebGL - Web-based version that allows OpenGL rendering in web browsers using JavaScript

Current use of OpenGL

- OpenGL is still used in some gaming and education contexts - in situations where cross-platform compatibility as well as stability are more important than cutting -edge technology.

EXTRA QUESTIONS

1. **Difference between API and library/engine:**

    API
    - Set of rules, protocols and tools for building software applications
    - It defines how different SW components should interact
    - In graphics, API like OpenGL or Vulkan provide functions to interact with the GPU for rendering
    - Example: OpenGL is an API that allows you to send commands to the GPU to draw shapes, apply textures and handle lighting

    Library
    - Collection of pre-written code that you can use to perform specific tasks

    Engine
    - Comprehensive framework that provides tools, libraries and APIs to build applications
    - It often includes rendering physcis, audio and input handling
    - Examples: Unity and Unreal Engine are game engines that provide a complete suite of tools for game development

2. **What APIs are currently used**
    - Graphic APIs
        - OpenGL - Cross-platform API for 2D and 3D graphics. Still used in education and industry, however largely replaced by Vulkan for high-performance applications
        - Vulkan - Modern, high-performance API for 3D graphics and compute. It is cross-platform and designed for efficiency
        - DirectX - COLLECTION of APIs for Windows and XboX including Direct3D for 3D graphics
        - Metal - Apples API for graphcis and compute on macOS and iOS devices
        - WebGL - Web-based API for rendering 3D graphics in browsers using JS
    - Compute APIs
        - OpenCL - For parallel computing across CPUs, GPUs
    - Other APIs
        - OpenXR - for VR and AR applications
        - OpenAL - for audio rendering

3. **What is rendering pipeline and how does it work?**
    - It is a sequence of steps a GPU follows to transform 3D objects into a 2D image on the screen
    - It involves processing vertices, rasterizing primitives and shading pixels

4. **What is a shader and what is its role in the rendering pipeline?**
    - Shader is a small program that runs on the GPU to perform specific tasks in the rendering pipeline
    - Shaders are written in shading languages like GLSL (OpenGL), HLSL (DirectX) or SPIR -V (Vulkan)

Přednáška #2

- All objects are stored as sets of vertices
- The final shape is just representation of these vertices

GEOMETRIC PRIMITIVES

PONTS

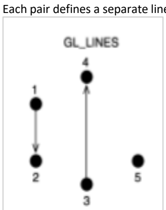
GL_POINTS

- A point is the simples geometric primitive
- It is represented by a single vertex in 3D space
- Points are often used for particle effects, stars or other small visual elements
- We can adjust the size of the point by using glPointSize()

LINES

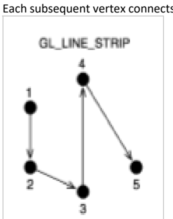
GL_LINES

- Draws a series of unconnected line segments
- Each pair defines a separate line



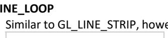
GL_LINE_STRIP

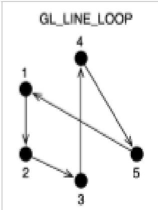
- Draws a connected series of line segments
- Each subsequent vertex connects to the next, forming a continuous line



GL_LINE_LOOP

- Similar to GL_LINE_STRIP, however, the last vertex connects to the first, creating a loop





GL_LINES_ADJACENCY

- This object is used to draw lines with adjacency information
- Each line segment is represented by 4 vertices
- This primitive is used to draw lines with adjacency information.
- Each line segment is represented by 4 vertices:
  - The first vertex is the previous vertex (adjacent to the start of the line).
  - The second vertex is the start of the line.
  - The third vertex is the end of the line.
  - The fourth vertex is the next vertex (adjacent to the end of the line).
- The geometry shader receives all 4 vertices and can use the adjacency information to perform advanced operations.

- Use Case: Detecting sharp edges or generating additional geometry along a line.

GL_LINE_STRIP_ADJACENCY

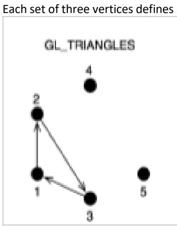
- This primitive is used to draw a connected strip of lines with adjacency information.
- For each line segment in the strip, the geometry shader receives 4 vertices:
  - The first vertex is the previous vertex (adjacent to the start of the line).
  - The second vertex is the start of the line.
  - The third vertex is the end of the line.
  - The fourth vertex is the next vertex (adjacent to the end of the line).
- Unlike GL_LINES_ADJACENCY, this primitive connects the lines into a strip, so the adjacency information is shared between consecutive lines.
- Use Case: Rendering smooth curves or detecting discontinuities in a line strip.

TRIANGLES

- The most primitive type in 3D graphics - they are simple, planar (2D) and can form complex surfaces

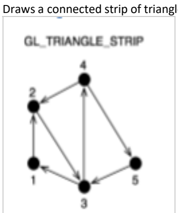
GL_TRIANGLES

- Draws a series of separate triangles
- Each set of three vertices defines a triangle (if we define just two vertices using this object, no object is formed)



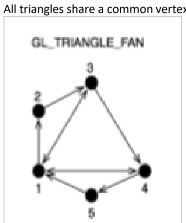
GL_TRIANGLE_STRIP

- Draws a connected strip of triangles, each new vertex forms a triangle with the previous two vertices



GL_TRIANGLE_FAN

- Draws a "fan" of triangles
- All triangles share a common vertex (the first one). Each new vertex forms a triangle with the previous vertex and the common vertex (the first vertex)



GL_TRIANGLES_ADJACENCY

- This primitive is used to draw triangles with adjacency information.
- Each triangle is represented by 6 vertices:
  - The first 3 vertices define the triangle itself.
  - The next 3 vertices represent the adjacent vertices (one for each edge of the triangle).
- The geometry shader receives all 6 vertices and can use the adjacency information to perform advanced operations.
- Use Case: Detecting silhouette edges, generating fur or grass, or performing tessellation.

GL_TRIANGLE_STRIP_ADJACENCY

- This primitive is used to draw a connected strip of triangles with adjacency information.
- For each triangle in the strip, the geometry shader receives 6 vertices:
  - The first 3 vertices define the triangle itself.
  - The next 3 vertices represent the adjacent vertices (one for each edge of the triangle).
- Like GL_TRIANGLES_ADJACENCY, this primitive provides adjacency information, but the triangles are connected in a strip.
- Use Case: Rendering smooth surfaces or detecting discontinuities in a triangle strip.

QUADS

- Note: Quads are deprecated in modern OpenGL and are not supported in OpenGL ES. Instead, use triangles to represent quads.

GL_QUADS

- Draws a series of separate four-sided polygons. Each set of four vertices defines a quad

GL_QUAD_STRIP

- Draws a connected strip of quads. Each part of vertices forms a quad with the next pair.

POLYGON

- Like quads, polygons are deprecated in modern OpenGL and should be replaced with triangles.

GL_POLYGON

- Draws a single convex polygon with any number of vertices
- The vertices must be specified in order (clockwise or counterclockwise) to form a valid convex shape.

PATCHES

GL_PATCHES
- Used in tessellation shaders to define patches, which are collections of vertices that can be subdivided into smaller primiti ves (objects)

VERTEX BUFFERS
- Called Vertex Buffer Objects (VBOs)
- Modern and efficient way to handle vertex data in OpenGL
- They provide an alternative to defining verticies using separate commands (glVertex3f() or glColor3f())
- Instead, vertex buffers allow you to store all vertex data in GPU - making rendering more efficient and faster

- For using vertex buffers, we must implement the following:
  1. **Enabling of the arrays**
     - Before using vertex arrays, you must enable the specific types of arrays you want to use (vertex coordinates, colors, normalsetc.)
     - This is done using the command:

       glEnableClientState(GL_....ARRAY)
     - The following arrays are available to be enabled/disabled:

       GL_VERTEX_ARRAY - coordinates
       GL_COLOR_ARRAY - colors
       GL_SECONDARY_COLOR_ARRAY - secondary colors
       GL_INDEX_ARRAY - deprecated
       GL_NORMAL_ARRAY - normals for lighting
       GL_FOG_COORDINATE_ARRAY - fog
       GL_TEXTURE_COORD_ARRAY - texture coordinates
       GL_EDGE_FLAG_ARRAY - is edge visible?
  - Setting pointers on the data in the arrays
  - Drawing of the geometric primitives
  - Disabling of the arrays - glDisableClientState()

2. SETTING POINTERS FOR READING
- Let us assume that the triangle is given by vertices with two GLint coordinates (x,y). The array will be: $\{x_1, y_1, x_2, y_2, x_3, y_3\}$ where x_a and y_a are Glint values
- In C++, we can define this as:
- Glint vertices[] = {10, 10, 100, 300, 200, 10};
- We must also specify the array structure:
  - glVertexPointer() - Coordinate definition; specifies the location and data format of the vertex coordinates - it tells OpenGL how to intereprett the array of vertex data
  - glCOlorPointer() - Color definition; specifies the color data of each vertex (the colors are usually specified using RGBA values)
- Keep in mind these functions are part of the older OpenGL fixed-function pipeline. In Modern OpenGL you would use shaders and VBO (Vertex Buffer Objects)

VERTEX POINTERS

glVertexPointer(Glint size, Glenum type, Glsiyeistride, const Glvoid *pointers)
- Size - specified the number of coordinates per vertex (it can be 2, 3, 4)
- Type - specified the data type of each coordinate in the array; could be:
  - GL_SHORT
  - GL_INT
  - GL_FLOAT
  - GL_DOUBLE
- Stride - specified the byte offset between consecutive vertices; if there is no space between vertices it is 0
- Pointer - pointer to the array containing the vertex data. It points to the first coordinate of the first vertex

Example 1:

```cpp
glVertexPointer(2, GL_INT, 0, vertices);
```

- **size**: 2 (2D coordinates)
- **type**: GL_INT (integer type)
- **stride**: 0 (data is tightly packed)
- **pointer**: vertices (pointer to the vertex array)

This example specifies that the vertex data consists of 2D integer coordinates with no extra data between vertices.

Example 2:

```cpp
glVertexPointer(2, GL_FLOAT, 5*sizeof(GL_FLOAT), &triangle[0]);
```

- **size**: 2 (2D coordinates)
- **type**: GL_FLOAT (floating-point type)
- **stride**: 5*sizeof(GL_FLOAT) (each vertex is separated by 5 floating-point values)
- **pointer**: &triangle[0] (pointer to the first element of the triangle array)

This example specifies that the vertex data consists of 2D floating-point coordinates, with each vertex separated by 5 floating-point values (possibly including additional data like color or texture coordinates).

4. **Usage Context:**
   - glVertexPointer is typically used in conjunction with glEnableClientState(GL_VERTEX_ARRAY) to enable the use of vertex arrays.
   - After setting up the vertex pointer, you can use functions like glDrawArrays or glDrawElements to render the vertices.

COLOR POINTERS

glColorPointer(GLint size, GLenum type, GLsizei stride, const GLvoid *pointer);
- Size - specified the number of color channels per vertex. It can be 3 (for RGB) or 4 (for RGBA which includes the alpha channel for transparency)
  - For the alpha channel, if it is 1 (255) it is full color
- Type - specified the data type of each coordinate in the array; could be:
  - GL_BYTE
  - GL_UNSIGNED_BYTE
  - GL_UNSIGNED_SHORT
  - GL_UNSIGNED_INT
- Stride - specified the byte offset between consecutive vertices; if there is no space between vertices it is 0
- Pointer - pointer to the array containing the vertex data. It points to the first color channel of the first vertex

4. **Example:**

Suppose you have an array of colors defined as follows:

```cpp
```

4. **Example**:

Suppose you have an array of colors defined as follows:

```cpp
GLubyte colors[] = {
    255, 0, 0, 255,   // Red
    0, 255, 0, 255,   // Green
    0, 0, 255, 255    // Blue
};
```

You can specify the color pointer like this:

```cpp
glColorPointer(4, GL_UNSIGNED_BYTE, 0, colors);
```

- `size`: 4 (RGBA)
- `type`: `GL_UNSIGNED_BYTE` (8-bit unsigned byte)
- `stride`: 0 (data is tightly packed)
- `pointer`: `colors` (pointer to the color array)

EXAMPLE CODE

- This code renders a triangle with vertices at (10, 10), (100, 300), and (200, 10).
- Each vertex has a different color: red, blue, and green.
- OpenGL interpolates the colors across the triangle, creating a gradient effect.

```
glClear(GL_COLOR_BUFFER_BIT);

GLint vertices[] = {10, 10, 100, 300, 200, 10}; // Vertex coordinates
GLfloat colors[] = {1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0}; // Vertex colors

glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);

glVertexPointer(2, GL_INT, 0, vertices);
glColorPointer(3, GL_FLOAT, 0, colors);

// Draw the triangle
glDrawArrays(GL_TRIANGLES, 0, 3);

glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_COLOR_ARRAY);

glFlush();
```

The provided code is an example of how to render a colored triangle using OpenGL's fixed-function pipeline. Here's a step-by-step explanation of each part of the code:

1. Clear the Color Buffer

```cpp
glClear(GL_COLOR_BUFFER_BIT);
```

- This clears the color buffer (the screen) to the default background color (usually black or set by `glClearColor`).

2. Define Vertex Coordinates

```cpp
GLint vertices[] = {10, 10, 100, 300, 200, 10};
```

- This defines the vertex coordinates for a triangle. Each pair of values represents the (x, y) coordinates of a vertex:
  - Vertex 1: (10, 10)
  - Vertex 2: (100, 300)
  - Vertex 3: (200, 10)

3. Define Color Data

```cpp
GLfloat colors[] = {1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0};
```

- This defines the colors for each vertex. Each triplet of values represents the (R, G, B) color for a vertex:
  - Vertex 1: (1.0, 0.0, 0.0) → Red

- This defines the colors for each vertex. Each triplet of values represents the (R, G, B) color for a vertex:
  - Vertex 1: (1.0, 0.0, 0.0) → Red
  - Vertex 2: (0.0, 0.0, 1.0) → Blue
  - Vertex 3: (0.0, 1.0, 0.0) → Green

---

4. Enable Vertex and Color Arrays

cpp Copy

```cpp
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);
```

- These functions enable the use of vertex and color arrays. OpenGL will use the data provided in `vertices` and `colors` for rendering.

5. Specify Vertex and Color Pointers

cpp Copy

```cpp
glVertexPointer(2, GL_INT, 0, vertices);
glColorPointer(3, GL_FLOAT, 0, colors);
```

- `glVertexPointer`:
  - `2`: Number of coordinates per vertex (x, y).
  - `GL_INT`: Data type of the coordinates (integer).
  - `0`: Stride (0 means the data is tightly packed).
  - `vertices`: Pointer to the vertex array.
- `glColorPointer`:
  - `3`: Number of color components per vertex (R, G, B).
  - `GL_FLOAT`: Data type of the color components (floating-point).
  - `0`: Stride (0 means the data is tightly packed).
  - `colors`: Pointer to the color array.

---

6. Drawing the Triangle

- The actual drawing command is missing in the code. Typically, you would use:

  cpp Copy

  ```cpp
  glDrawArrays(GL_TRIANGLES, 0, 3);
  ```

  - `GL_TRIANGLES`: Specifies that the vertices form a triangle.
  - `0`: Starting index in the vertex array.
  - `3`: Number of vertices to render.

---

7. Disable Vertex and Color Arrays

cpp Copy

```cpp
glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_COLOR_ARRAY);
```

- These functions disable the vertex and color arrays after rendering is complete. This is good practice to clean up the OpenGL state.

8. Flush the Pipeline

cpp Copy

```cpp
glFlush();
```

cpp Copy

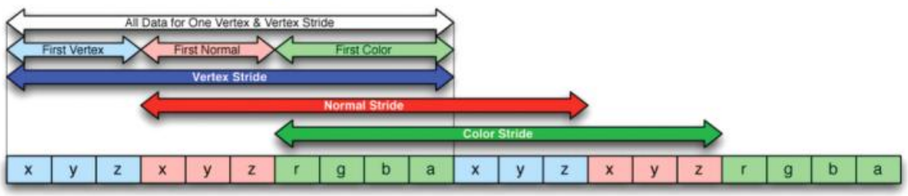
```cpp
glFlush();
```

- This ensures that all OpenGL commands are executed and the triangle is rendered to the screen.

INTERLEAVED ARRAYS

- In case of independent arrays, the stride parameter equals zero
- In case of interleaved arrays, this parameter is equivalent to different values as there are multiple values stored in one array (for example both coordinates as well as colors)
- The value is a distance between first bytes of the values of the same meaning
- Example: both coordinates and colors are GL_FLOAT values. The stride will be 5*sizeof(GL_FLOAT) - because of 2 GL_FLOAT values for coordinates and 3 for colors
- The last parameter is the pointer to where the values begin
- Example: in case of array x1, y1, R1, G1, B1 the pointer will be &data[0] for coords and &data[2] for colors

- The stride parameter is zero in case of independent arrays
- It has non-zero value in case of interleaved arrays where in a single arrays are different values (coords and colors)
- The value is a distance between first bytes of the values of the same meaning (two colors)
- Example: Both coordinates and colors are GL_FLOAT values. The stride will be 5*sizeof(GL_FLOAT) f
- The last parameter is pointer where the values begin
- Example: in case of array x1, y1, R1, G1, B1 - the pointer will be &data[0] for coords and &data[2] for colors




```cpp
glClear (GL_COLOR_BUFFER_BIT);
static GLfloat triangle[] = {
   10.0,  10.0, 1.0, 0.0, 0.0, // 2 coords, 3 colors
  100.0, 300.0, 0.0, 0.0, 1.0,
  200.0,  10.0, 0.0, 1.0, 0.0};
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);
glVertexPointer(2, GL_FLOAT, 5*sizeof(GL_FLOAT), &triangle[0]);
glColorPointer(3, GL_FLOAT, 5*sizeof(GL_FLOAT), &triangle[2]);
// drawing
glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_COLOR_ARRAY);
glFlush();
```

The provided content explains the concept of **interleaved arrays** in OpenGL and demonstrates how to us them with the `glVertexPointer` and `glColorPointer` functions. Interleaved arrays store multiple type of data (e.g., vertex coordinates and colors) in a single array, with a specific **stride** to separate the data fo each vertex. Here's a detailed explanation:

1. Interleaved Arrays

- In interleaved arrays, data for each vertex (e.g., coordinates, colors, normals) is stored together in a single array.
- For example, a vertex might have:
  - 2D coordinates (x, y)
  - RGB color values (r, g, b)
- The data for each vertex is stored sequentially, and the **stride** parameter is used to skip over the data for the next vertex.

2. Stride Parameter

- The **stride** is the number of bytes between the start of one vertex's data and the start of the next vertex's data.
- If the data is tightly packed (e.g., separate arrays for coordinates and colors), the stride is **0**.
- For interleaved arrays, the stride is the total size of one vertex's data (e.g., coordinates + colors).

3. Example: Interleaved Array

The following array stores 2D coordinates and RGB colors for each vertex:

cpp Copy

```cpp
static GLfloat triangle[] = {
    10.0, 10.0, 1.0, 0.0, 0.0, // Vertex 1: (x, y, r, g, b)
    100.0, 300.0, 0.0, 0.0, 1.0, // Vertex 2: (x, y, r, g, b)
    200.0, 10.0, 0.0, 1.0, 0.0 // Vertex 3: (x, y, r, g, b)
```

```cpp
static GLfloat triangle[] = {
    10.0, 10.0, 1.0, 0.0, 0.0, // Vertex 1: (x, y, r, g, b)
    100.0, 300.0, 0.0, 0.0, 1.0, // Vertex 2: (x, y, r, g, b)
    200.0, 10.0, 0.0, 1.0, 0.0 // Vertex 3: (x, y, r, g, b)
};
```

- Each vertex has **5 values**: 2 for coordinates (x, y) and 3 for colors (r, g, b).
- The **stride** is `5 * sizeof(GL_FLOAT)` because each vertex's data occupies 5 floating-point values.

```
static GLfloat triangle[] = {
    10.0, 10.0, 1.0, 0.0, 0.0, // Vertex 1: (x, y, r, g, b)
    100.0, 300.0, 0.0, 0.0, 1.0, // Vertex 2: (x, y, r, g, b)
    200.0, 10.0, 0.0, 1.0, 0.0 // Vertex 3: (x, y, r, g, b)
};
```

4. Setting Up Pointers

To use the interleaved array, you need to specify the starting points for the vertex and color data:

Vertex Pointer

```cpp
glVertexPointer(2, GL_FLOAT, 5 * sizeof(GL_FLOAT), &triangle[0]);
```

- `2`: Number of coordinates per vertex (x, y).
- `GL_FLOAT`: Data type of the coordinates.
- `5 * sizeof(GL_FLOAT)`: Stride (distance between the start of one vertex's coordinates and the next).
- `&triangle[0]`: Pointer to the start of the vertex coordinates in the array.

Color Pointer

```cpp
glColorPointer(3, GL_FLOAT, 5 * sizeof(GL_FLOAT), &triangle[2]);
```

- `3`: Number of color components per vertex (r, g, b).
- `GL_FLOAT`: Data type of the color components.
- `5 * sizeof(GL_FLOAT)`: Stride (distance between the start of one vertex's colors and the next).
- `&triangle[2]`: Pointer to the start of the color data in the array (skips the first 2 values, which are coordinates).

5. Rendering the Triangle

The code sets up the vertex and color pointers and then renders the triangle:

```cpp
glClear(GL_COLOR_BUFFER_BIT); // Clear the screen

glEnableClientState(GL_VERTEX_ARRAY); // Enable vertex array
glEnableClientState(GL_COLOR_ARRAY); // Enable color array

glVertexPointer(2, GL_FLOAT, 5 * sizeof(GL_FLOAT), &triangle[0]); // Set vertex pointer
glColorPointer(3, GL_FLOAT, 5 * sizeof(GL_FLOAT), &triangle[2]); // Set color pointer

// Draw the triangle
glDrawArrays(GL_TRIANGLES, 0, 3);

glDisableClientState(GL_VERTEX_ARRAY); // Disable vertex array
glDisableClientState(GL_COLOR_ARRAY); // Disable color array

glFlush(); // Ensure rendering is complete
```

6. Key Points

- **Interleaved Arrays**: Store multiple types of data (e.g., coordinates, colors) in a single array.
- **Stride**: Specifies the distance between the start of one vertex's data and the next.
- **Pointers**: The `glVertexPointer` and `glColorPointer` functions use the stride and starting pointers to interpret the interleaved data.
- **Rendering**: The `glDrawArrays` function renders the triangle using the specified vertex and color data.

```
glClear(GL_COLOR_BUFFER_BIT); // Clear the screen

glEnableClientState(GL_VERTEX_ARRAY); // Enable vertex array
glEnableClientState(GL_COLOR_ARRAY); // Enable color array
```

```
glClear(GL_COLOR_BUFFER_BIT); // Clear the screen

glEnableClientState(GL_VERTEX_ARRAY); // Enable vertex array
glEnableClientState(GL_COLOR_ARRAY); // Enable color array

glVertexPointer(2, GL_FLOAT, 5 * sizeof(GL_FLOAT), &triangle[0]); // Set vertex pointer
glColorPointer(3, GL_FLOAT, 5 * sizeof(GL_FLOAT), &triangle[2]); // Set color pointer

// Draw the triangle
glDrawArrays(GL_TRIANGLES, 0, 3);

glDisableClientState(GL_VERTEX_ARRAY); // Disable vertex array
glDisableClientState(GL_COLOR_ARRAY); // Disable color array

glFlush(); // Ensure rendering is complete
```

3.

7. Visualization of the Interleaved Array

For the array:

cpp Co

```cpp
static GLfloat triangle[] = {
    10.0, 10.0, 1.0, 0.0, 0.0, // Vertex 1: (x, y, r, g, b)
    100.0, 300.0, 0.0, 0.0, 1.0, // Vertex 2: (x, y, r, g, b)
    200.0, 10.0, 0.0, 1.0, 0.0 // Vertex 3: (x, y, r, g, b)
};
```

- **Vertex 1**: Starts at `&triangle[0]` (coordinates) and `&triangle[2]` (colors).
- **Vertex 2**: Starts at `&triangle[5]` (coordinates) and `&triangle[7]` (colors).
- **Vertex 3**: Starts at `&triangle[10]` (coordinates) and `&triangle[12]` (colors).

8. Summary

- Interleaved arrays combine vertex and color data into a single array.
- The **stride** parameter ensures OpenGL knows how to skip over data for the next vertex.
- This approach is efficient and commonly used in OpenGL for organizing vertex attributes.

Let me know if you need further clarification!

- The function glMultiDrawElements allows you to draw multiple sets of primitives (points, lines, triangles) in a single call. This makes for more efficient rendering performance than if we called glDrawElements multiples times for each set of indices.

glMultiDrawElements(GLenum mode, Glsizei* count, GLenum type, void** indices, GLsizei primcount)
- **mode**: The type of primitive to draw (e.g., GL_LINES for lines, GL_TRIANGLES for triangles, etc.).
- **count**: An array containing the number of indices for each set of primitives.
- **type**: The data type of the indices (e.g., GL_UNSIGNED_BYTE for unsigned bytes, GL_UNSIGNED_INT for unsigned integers, etc.).
- **indices**: An array of pointers, where each pointer points to an array of indices for a set of primitives.
- **primcount**: The number of sets of primitives to draw (i.e., the number of arrays of indices).

1. Define the Index Arrays

c Collapse Wrap Copy

```c
GLubyte firstIndices[] = {0, 1, 2, 3};
GLubyte secondIndices[] = {2, 4, 7, 8};
```

- `firstIndices` and `secondIndices` are two arrays of indices, each containing the vertex indices that define which vertices to use when drawing.
- The indices are of type `GLubyte` (unsigned byte), meaning each index is an 8-bit unsigned integer.
- `firstIndices` contains `{0, 1, 2, 3}`, so it will draw using vertices 0, 1, 2, and 3.
- `secondIndices` contains `{2, 4, 7, 8}`, so it will draw using vertices 2, 4, 7, and 8.

- The command:
- glDrawElements(Glenum mode, Glsizei count, Glenum type, void* indices)
  - Mode - selected primitive (GL_TRIANGLES)
  - Count - amount of indices in an array
  - Type - the type of indices (GL_INT)
  - Indices - pointer on indices array

1. Clear the Color Buffer

cpp Copy

```cpp
glClear(GL_COLOR_BUFFER_BIT);
```

- Clears the screen to the default background color.

2. Define Interleaved Array

cpp Copy

```cpp
GLfloat triangle[] = {
    10.0, 10.0, 1.0, 0.0, 0.0, // Vertex 1: (x, y, r, g, b)
    100.0, 300.0, 0.0, 0.0, 1.0, // Vertex 2: (x, y, r, g, b)
    200.0, 10.0, 0.0, 1.0, 0.0 // Vertex 3: (x, y, r, g, b)
```

```cpp
GLfloat triangle[] = {
    10.0, 10.0, 1.0, 0.0, 0.0, // Vertex 1: (x, y, r, g, b)
    100.0, 300.0, 0.0, 0.0, 1.0, // Vertex 2: (x, y, r, g, b)
    200.0, 10.0, 0.0, 1.0, 0.0 // Vertex 3: (x, y, r, g, b)
};
```

- Stores vertex coordinates and colors in a single interleaved array.
- Each vertex has **5 values**: 2 for coordinates (x, y) and 3 for colors (r, g, b).

3. Enable Vertex and Color Arrays

cpp Copy

```cpp
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);
```

- Enables the use of vertex and color arrays for rendering.

4. Set Up Vertex and Color Pointers

cpp Copy

```cpp
glVertexPointer(2, GL_FLOAT, 5 * sizeof(GL_FLOAT), &triangle[0]);
glColorPointer(3, GL_FLOAT, 5 * sizeof(GL_FLOAT), &triangle[2]);
```

- `glVertexPointer`:
  - `2`: Number of coordinates per vertex (x, y).
  - `GL_FLOAT`: Data type of the coordinates.
  - `5 * sizeof(GL_FLOAT)`: Stride (distance between the start of one vertex's data and the next).
  - `&triangle[0]`: Pointer to the start of the vertex coordinates in the array.
- `glColorPointer`:
  - `3`: Number of color components per vertex (r, g, b).
  - `GL_FLOAT`: Data type of the color components.
  - `5 * sizeof(GL_FLOAT)`: Stride (same as vertex pointer).
  - `&triangle[2]`: Pointer to the start of the color data in the array (skips the first 2 values, which are coordinates).

5. Define Indices

cpp Copy

```cpp
static GLubyte indices[] = {0, 1, 2};
```

- Specifies the order in which vertices are used to draw the triangle.
- `0`, `1`, `2` refer to the first, second, and third vertices in the `triangle` array.

6. Draw Using Indices

cpp Copy

```cpp
glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_BYTE, indices);
```

- `GL_TRIANGLES`: Specifies that the vertices form a triangle.
- `3`: Number of indices to render.
- `GL_UNSIGNED_BYTE`: Data type of the indices (8-bit unsigned byte).
- `indices`: Pointer to the indices array.

7. Disable Vertex and Color Arrays

cpp Copy

```cpp
glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_COLOR_ARRAY);
```

- Disables the vertex and color arrays after rendering is complete.

8. Flush the Pipeline

cpp Copy

```cpp
glFlush();
```

```cpp
glFlush();
```

- Ensures all OpenGL commands are executed and the triangle is rendered to the screen.

FULL CODE:

```
glClear(GL_COLOR_BUFFER_BIT);

GLfloat triangle[] = {
   10.0, 10.0, 1.0, 0.0, 0.0, // Vertex 1: (x, y, r, g, b)
   100.0, 300.0, 0.0, 0.0, 1.0, // Vertex 2: (x, y, r, g, b)
   200.0, 10.0, 0.0, 1.0, 0.0 // Vertex 3: (x, y, r, g, b)
};

glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);

glVertexPointer(2, GL_FLOAT, 5 * sizeof(GL_FLOAT), &triangle[0]);
glColorPointer(3, GL_FLOAT, 5 * sizeof(GL_FLOAT), &triangle[2]);

static GLubyte indices[] = {0, 1, 2}; // Define indices
glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_BYTE, indices);

glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_COLOR_ARRAY);

glFlush();
```

2. Define the Count Array

```c
GLsizei count[] = {4, 4};
```

- The `count` array specifies how many indices are in each index array.
- Here, `count[0] = 4` means the first set (`firstIndices`) has 4 indices.
- `count[1] = 4` means the second set (`secondIndices`) also has 4 indices.

3. Create an Array of Pointers to the Index Arrays

```c
void* indices[2] = {firstIndices, secondIndices};
```

- `indices` is an array of pointers (`void*`), where each pointer points to one of the index arrays.
- `indices[0]` points to `firstIndices`.
- `indices[1]` points to `secondIndices`.

This matches the `indices` parameter in `glMultiDrawElements`, which expects an array of pointers to the index data.

4. Call `glMultiDrawElements`

```c
glMultiDrawElements(GL_LINES, count, GL_UNSIGNED_BYTE, indices, 2);
```

- **`GL_LINES`**: The `mode` parameter specifies that the primitives to draw are lines. In OpenGL, `GL_LINES` means every pair of indices defines a single line segment (e.g., indices 0 and 1 form one line, indices 2 and 3 form another line, etc.).
- **`count`**: The array `{4, 4}`, indicating that each set of indices (`firstIndices` and `secondIndices`) contains 4 indices.
- **`GL_UNSIGNED_BYTE`**: The `type` parameter specifies that the indices are of type `GLubyte` (unsigned byte).
- **`indices`**: The array of pointers `{firstIndices, secondIndices}`, pointing to the index data.
- **`2`**: The `primcount` parameter, which is the number of sets of primitives to draw. Since we have two index arrays (`firstIndices` and `secondIndices`), `primcount` is 2.

1. PRIMITIVES DRAWING

DRAWING USING INDICES

Command for drawing using indices

`glDrawElements(GLenum mode, Glsizei count, GLenum type, void* indices)`

- `mode` – selected primitive (e.g. GL_TRINAGLES),
- `count` – amount of indices in the array,
- `type` – type of indices (e.g. GL_INT),
- `indices` – pointer on **indices array**.

```
glClear (GL_COLOR_BUFFER_BIT);
```

- count – amount of indices in the array,
- type – type of indices (e.g. GL_INT),
- indices – pointer on **indices array**.

```
glClear (GL_COLOR_BUFFER_BIT);
GLfloat triangle[] = {
  10.0, 10.0, 1.0, 0.0, 0.0,
  100.0, 300.0, 0.0, 0.0, 1.0,
  200.0, 10.0, 0.0, 1.0, 0.0};
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);
glVertexPointer(2, GL_FLOAT, 5*sizeof(GL_FLOAT), &triangle[0]);
glColorPointer(3, GL_FLOAT, 5*sizeof(GL_FLOAT), &triangle[2]);
static GLubyte indices[]={0,1,2}; // definice pole indexu
glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_BYTE, indices);
glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_COLOR_ARRAY);
glFlush();
```

EXTENSION OF GLDRAWELEMENTS

DRAWING USING VERTICES

- glDrawArrays(GLenum type, GLint first, GLsizei count),
- glMutiDrawArrays(GLenum type, GLint* first, GLsizei* count, GLsizei primcount)
- It works directly with vertices, not indices
- Draws from the first vertex to first + count vertex.

First Section: `glDrawArrays` and `glMultiDrawArrays`

1. `glDrawArrays(GLenum type, GLint first, GLsizei count)`

- **What it does**: This OpenGL function is used to draw a sequence of geometric primitives (like points, lines, or triangles) using vertex data stored in arrays.
- **Parameters**:
  - `type`: Specifies the type of primitive to draw. Examples include:
    - `GL_POINTS` (draws individual points),
    - `GL_LINES` (draws lines),
    - `GL_TRIANGLES` (draws triangles).
  - `first`: The starting index in the vertex array where drawing begins.
  - `count`: The number of vertices to draw.
- **How it works**:
  - It draws primitives using vertices directly (not indices).
  - It starts at the `first` vertex and draws `count` number of vertices in sequence.
  - For example, if `type` is `GL_TRIANGLES`, `first` is 0, and `count` is 6, it will draw 2 triangles (since each triangle needs 3 vertices: 6 vertices = 2 triangles).

2. `glMultiDrawArrays(GLenum type, GLint* first, GLsizei* count, GLsizei primcount)`

- **What it does**: This is a more advanced version of `glDrawArrays`. It allows you to draw multiple sets of primitives in a single call.
- **Parameters**:
  - `type`: Same as above, the type of primitive (e.g., `GL_TRIANGLES`).
  - `first`: A pointer to an array of starting indices for each set of primitives.
  - `count`: A pointer to an array of vertex counts for each set of primitives.
  - `primcount`: The number of sets (or "draw calls") to perform.
- **How it works**:
  - Instead of drawing one continuous sequence of vertices, it draws multiple sequences in one go.
  - For example, if `primcount` is 2, `first` is `[0, 3]`, and `count` is `[3, 3]`, it will:
    - Draw 3 vertices starting at index 0 (first set),
    - Then draw 3 vertices starting at index 3 (second set).
  - This is useful for rendering multiple objects or batches efficiently.

```
1 ...
2 glVertexPointer(2, GL_FLOAT, 5*sizeof(GL_FLOAT),&triangles[0]);
3 glColorPointer(3, GL_FLOAT, 5*sizeof(GL_FLOAT),&triangles[2]);
4
5 glDrawArrays(GL_TRIANGLES, 0, 6);
6 ...
```

Line 2: `glVertexPointer(2, GL_FLOAT, 5*sizeof(GL_FLOAT), &triangles[0])`

```
4
5 glDrawArrays(GL_TRIANGLES, 0, 6);
6 ...
```

Line 2: `glVertexPointer(2, GL_FLOAT, 5*sizeof(GL_FLOAT), &triangles[0])`

- **What it does**: This tells OpenGL where to find the vertex position data and how to interpret it.
- **Parameters**:
    - `2` : Each vertex has 2 components (x, y coordinates for 2D vertices).
    - `GL_FLOAT` : The data type of each component is a float.
    - `5*sizeof(GL_FLOAT)` : The "stride," or the byte offset between consecutive vertices in the array. Here, each vertex has 5 floats (2 for position + 3 for color, as we'll see next).
    - `&triangles[0]` : A pointer to the start of the vertex data array (`triangles`).
- **Explanation**:
    - The `triangles` array contains vertex data where each vertex is made up of 5 floats: 2 for position (x, y) and 3 for color (r, g, b).
    - The stride is `5*sizeof(GL_FLOAT)` because OpenGL needs to skip 5 floats to get to the next vertex's position.

Line 3: `glColorPointer(3, GL_FLOAT, 5*sizeof(GL_FLOAT), &triangles[2])`

- **What it does**: This tells OpenGL where to find the color data for each vertex.
- **Parameters**:
    - `3` : Each color has 3 components (red, green, blue).
    - `GL_FLOAT` : The data type of each component is a float.
    - `5*sizeof(GL_FLOAT)` : The stride, same as above, since the position and color data are interleaved in the same array.
    - `&triangles[2]` : A pointer to the start of the color data. Since each vertex starts with 2 floats for position, the color data starts at offset 2 (i.e., `triangles[2]` is the red component of the first vertex).
- **Explanation**:
    - The `triangles` array is interleaved: for each vertex, the data is laid out as `[x, y, r, g, b]` .
    - So, for the first vertex, `triangles[0]` and `triangles[1]` are the x and y coordinates, and `triangles[2]` , `triangles[3]` , and `triangles[4]` are the red, green, and blue values.

Line 5: `glDrawArrays(GL_TRIANGLES, 0, 6)`

- **What it does**: This draws the triangles using the vertex data we just set up.
- **Parameters**:
    - `GL_TRIANGLES` : The primitive type is triangles.
    - `0` : Start at the first vertex (index 0).
    - `6` : Draw 6 vertices.
- **Explanation**:
    - Since we're drawing `GL_TRIANGLES` , each triangle needs 3 vertices.
    - With 6 vertices, OpenGL will draw 2 triangles (6 ÷ 3 = 2).
    - The vertices are read in order from the `triangles` array, using the position and color data we specified.

How the Data is Structured:

- The `triangles` array might look like this for 6 vertices (2 triangles):

text Collapse Wrap Copy

```text
triangles = [
  x1, y1, r1, g1, b1,  // Vertex 1
  x2, y2, r2, g2, b2,  // Vertex 2
  x3, y3, r3, g3, b3,  // Vertex 3 (first triangle: v1, v2, v3)
  x4, y4, r4, g4, b4,  // Vertex 4
  x5, y5, r5, g5, b5,  // Vertex 5
  x6, y6, r6, g6, b6   // Vertex 6 (second triangle: v4, v5, v6)
]
```

METHODS FOR DRAWING PRIMITIVES

This lists different OpenGL functions for drawing primitives, each with a specific use case.

1. glDrawElements

- **What it does**: Draws primitives using an array of indices.
- **How it works**:
    - Instead of drawing vertices in sequence, you provide an array of indices that reference vertices in your vertex array.
    - For example, if you have vertices [v0, v1, v2, v3] and indices [0, 1, 2], it draws a triangle using v0, v1, and v2.
    - This is efficient because you can reuse vertices (e.g., for shared edges in a mesh).

2. glMultiDrawElements

- **What it does**: Similar to glDrawElements, but draws multiple sets of primitives using multiple index arrays.
- **How it works**:
    - You provide arrays of indices and counts, allowing you to draw multiple objects in one call.
    - Useful for batching draw calls to improve performance.

- This is efficient because you can reuse vertices (e.g., for shared edges in a mesh).

2. glMultiDrawElements
- **What it does**: Similar to glDrawElements, but draws multiple sets of primitives using multiple index arrays.
- **How it works**:
  - You provide arrays of indices and counts, allowing you to draw multiple objects in one call.
  - Useful for batching draw calls to improve performance.

3. glDrawRangeElements
- **What it does**: A variation of glDrawElements that draws using a range of indices.
- **How it works**:
  - You specify a minimum and maximum index value, and OpenGL only processes indices within that range.
  - This can help the GPU optimize rendering by limiting the vertex data it needs to process.

4. glDrawArrays
- **What it does**: As explained earlier, draws primitives directly using vertex arrays (no indices).
- **How it works**:
  - It processes vertices in sequence, as seen in the code snippet above.

5. glMultiDrawArrays
- **What it does**: Also explained earlier, draws multiple sequences of vertices in one call.
- **How it works**:
  - It's like calling glDrawArrays multiple times, but more efficient because it's a single function call

Přednáška #3

VERTEX SHADER
- Shader is a program that runs on GPU's shader core and processes each vertex of a 3D model individually
- The main purpose of a vertex shader is to take each one individually and figure out where it should appear on the screen
- The input for a vertex shaders are position (x, y, z), color, normal (for lighting), texture coordinates etc.
- The purpose of vertex shader:
  - Geometric transformations - applies the modelview matrix (to position the object in the world and relative to the camera) and the projection matrix (to project the 3D scene onto a 2D scene)
  - Normal transformations - transforms normal vectors (used for lighting) and normalizes them to ensure correct lighting calculations
  - Transformation of texture coordinates
  - Per vertex lighting computation
- Each vertex is processed separately - therefore it does not know its neighbors
- Direct output from the shade is gl_position value

PIXEL/FRAGMENT SHADER
- The fragment shader (also called a pixel shader in some contexts, like Direct3D) operates on individual fragments (potential pixels) after the geometry has been rasterized.
- The fragment shader is used once the vertex shader has processed the vertices and the GPU has rasterized the primitives into fragments
  - A fragment is a piece of data that might become a pixel on the screen, but it's not final until after depth testing, stencil testing, and other pipeline stages.
- The purpose of the fragment shader is to:
  - Calculate the color of the fragment
  - Calculate the texture of from the texture coordinates
  - Calculate the fog - Applies fog effects to simulate atmospheric perspective (e.g., making distant objects fade into a fog color).
  - Per-pixel lightinig - Computes lighting at the fragment level (more precise than per-vertex lighting), using interpolated normals and other
- The input of the fragment shader are values from the previous step of the pipeline like interpolated color, normals, texturecoordinates etc.
- Like the vertex shader, it processes each fragment independently
- The final color of the fragment is at on the output as "gl_FragColor"

GEOMETRY SHADER

- The geometry shader is an optional stage in the OpenGL pipeline
- It generates additional vertices to create for example fur strands which are then rendered
- It operates on entire primitives after the vertex shader

HOW THE SHADERS WORK IN A PIPELINE
- The shaders above fit in an uniform pipeline like this:
  1. **Vertex shader**
     - Processes each vertex individually
     - Transforms vertex positions into clip space - GL_position
     - Passes other data (color, texture coordinates) to the next stage
  2. **Geometry shader - OPTIONAL**
     - Takes entire primitives as input
     - Generates new vertices or modifies the existing ones
  3. **Rasterization**
     - Converts primitives into fragments by interpolating vertex data (colors, texture coordinates) across the surface of the primi tives
  4. **Fragment shader**
     - Processes each fragment individually
     - Computes the final color (and optionally depth or stencil values) for the fragment

EXAMPLE OF SHADERS

Vertex shader

```
#version 330 core
in vec2 position;  // 2D vertex positions
in vec3 color;     // Vertex colors
out vec3 fragColor;  // Pass color to fragment shader
void main() {
   gl_Position = vec4(position, 0.0, 1.0);  // No transformations, directly use NDC
   fragColor = color;
}
```

- Takes the 2D positions and colors as input.
- Outputs gl_Position in clip space (since your coordinates are already in NDC, no transformation is needed).
- Passes the color to the fragment shader.

Fragment shader

```
#version 330 core
in vec3 fragColor;  // Interpolated color from vertex shader
out vec4 finalColor;  // Output color
void main() {
   finalColor = vec4(fragColor, 1.0);  // Set the fragment color (yellow)
}
```

- Takes the interpolated color from the vertex shader.
- Outputs the final color for the fragment (yellow, with full opacity).

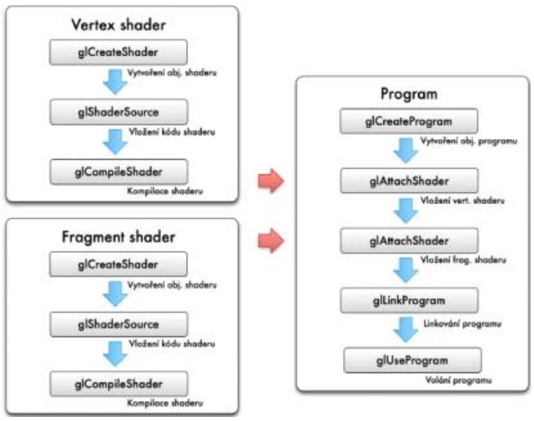
Geometry Shader (Optional)

```
#version 330 core
layout(triangles) in;
layout(triangle_strip, max_vertices=9) out;
in vec3 fragColor[];
out vec3 geomColor;
void main() {
   // Pass through the original triangle
   for (int i = 0; i < 3; i++) {
      gl_Position = gl_in[i].gl_Position;
      geomColor = fragColor[i];
      EmitVertex();
   }
   EndPrimitive();
   // Add a "fur" triangle (simplified example)
   // This would be more complex for real fur
   gl_Position = gl_in[0].gl_Position + vec4(0.1, 0.1, 0.0, 0.0);
   geomColor = fragColor[0];
   EmitVertex();
   gl_Position = gl_in[1].gl_Position + vec4(0.1, 0.1, 0.0, 0.0);
   geomColor = fragColor[1];
   EmitVertex();
   gl_Position = gl_in[2].gl_Position + vec4(0.1, 0.1, 0.0, 0.0);
```

```
gl_Position = gl_in[0].gl_Position + vec4(0.1, 0.1, 0.0, 0.0);
geomColor = fragColor[0];
EmitVertex();
gl_Position = gl_in[1].gl_Position + vec4(0.1, 0.1, 0.0, 0.0);
geomColor = fragColor[1];
EmitVertex();
gl_Position = gl_in[2].gl_Position + vec4(0.1, 0.1, 0.0, 0.0);
geomColor = fragColor[2];
EmitVertex();
EndPrimitive();
}
```

If you wanted to add fur to your polygon (as in the slide's example), you could write a geometry shader like this:
- Takes the original triangle as input.
- Emits the original triangle unchanged.
- Emits a new triangle slightly offset to simulate a fur-like effect.

SHADER INTEGRATION
- The diagram shows two paths—one for the vertex shader and one for the fragment shader—and how they come together to form a program
- On the left side for vertex and fragment shaders:
  - **glCreateShader:** This creates a new vertex/fragment shader object (like a blank piece of paper where you'll write your shader code).
  - **glShaderSource:** This is where you give the vertex shader its code (like writing instructions on the paper).
  - **glCompileShader:** This "compiles" the shader code, which means OpenGL checks the code for errors and prepares it to run on the graphics card.
- On the right side, the process of combining both shaders into a program is shown
  - **glCreateProgram:** Creates a new program object (like a blank recipe book).
  - **glAttachShader:** Attaches the vertex shader and fragment shader to the program
  - **glLinkProgram:** Links the shaders together, making sure they work as a team.
  - **glUseProgram:** Tells OpenGL to use this program when drawing (like telling the chef to follow this recipe).
- To draw something on the screen, you need both the vertex shader (to position your points) and a fragment shader (to color the pixels). They then have to be combined into a program



EXAMPLE OF SHADER INTEGRATION

```
GLuint v = glCreateShader(GL_VERTEX_SHADER);
GLuint f = glCreateShader(GL_FRAGMENT_SHADER);
//Create two shader objects - one for vertex shader and one for fragment shader; this step is like creating two blank notebooks
//the GLuint variables are used to keep track of the shaders
```

```
char* vs = " // here is some vertex shader // ";
char* fs = " // here is some fragment shader // ";
//vs and fs are codes for vertex and fragment shaders
```

```
glShaderSource(v, 1, vs, NULL);
glShaderSource(f, 1, fs, NULL);
//These lines give the shader code to the vertex shader (v) and fragment shader (f); The 1 means you're giving one piece of c ode; NULL is for extra options
```

```
glCompileShader(v);
glCompileShader(f);
//Compile the vertex shader and fragment shader; this is the step where an error will be thrown
```

```
GLuint p = glCreateProgram();
//a new program object is made; this is like making a blank recipe book where vertex and fragment shader instructions will be   combined
```

```
glAttachShader(p, v);
glAttachShader(p, f);
//attaches the vertex and fragment shaders to the program p
```

```
glLinkProgram(p);
//links the program to make sure the vertex and fragment shaders work together properly;
//this step is like making sense that the recipe makes sense as a whole  - like checking that the output of the vertex shader matches the input of the fragment shader
```

```
glUseProgram(p);
//this program tells OpenGL to use this program when drawing
```

- NOTE: In modern OpenGL, there are easier ways to write this code. For example, in Qt, there are helper classed like QOpenGLSh ader and QOPenGLShaderProgram that make this process easier

SIMPLE EXAMPLE
1. Write a Vertex Shader
2. Write a Fragment Shader
3. Set Up the Shaders (Using the Steps in the Slides):
   - Create the shaders with glCreateShader.
   - Give them the code with glShaderSource.
   - Compile them with glCompileShader.
   - Create a program with glCreateProgram.
   - Attach the shaders with glAttachShader.
   - Link the program with glLinkProgram.
   - Use the program with glUseProgram.
4. Draw Your Polygon:
   - Instead of using glVertexPointer and glColorPointer, you'd use modern OpenGL functions (like Vertex Buffer Objects) to send your coords and colors arrays to the shader.
   - Then you'd draw the polygon with glDrawElements, just like you're doing now.

GLSL
- Each API has its shader language
- OpenGL uses GLSL (GL Shader Language) - it is C-like program

Example:

GLSL

- Each API has its shader language
- OpenGL uses GLSL (GL Shader Language) - it is C-like program

Example:

```
void main(void)
//void main(void): The entry point of the shader, similar to main() in C. Every shader must have this function.
{
   gl_Position = vec4(a_Vertex, 1.0);
//gl_Position: A built-in output variable in a vertex shader that specifies the final position of a vertex in clip space (a 4D space used for rendering). It's assigned a vec4 (4D vector) constructed from:
      ○ a_Vertex: An input attribute (likely a vec3 for x, y, z coordinates).
      ○ 1.0: The w component, typically set to 1.0 for positions in 3D space.
   color = vec4(a_Color, 1.0);
//color: A user-defined output variable (likely a vec4) that passes the color to the next stage (e.g., fragment shader). It's constructed from:
      ○ a_Color: An input attribute (likely a vec3 for r, g, b values).
      ○ 1.0: The alpha component (opacity), set to 1.0 (fully opaque).
}
```

DATA TYPES

- GLSL supports various data types for scalars, vectors, matrices and textures

  Scalars - for single values
  - float: A floating-point number (e.g., 3.14).
  - int: An integer (e.g., 42).
  - bool: A boolean value (true or false).

  Vectors - for positions, colors or directions
  - vec2, vec3, vec4: Vectors of 2, 3, or 4 floats (e.g., vec3(1.0, 2.0, 3.0)).
  - ivec2, ivec3, ivec4: Vectors of 2, 3, or 4 integers.
  - uvec2, uvec3, uvec4: Vectors of 2, 3, or 4 unsigned integers.
  - bvec2, bvec3, bvec4: Vectors of 2, 3, or 4 booleans.

  Matrices - for transformations (rotations + scaling)
  - mat2, mat3, mat4: Square matrices (2x2, 3x3, 4x4) of floats.
  - mat2x3, mat2x4, mat3x2, mat3x4, mat4x3: Non-square matrices (e.g., mat2x3 is a 2x3 matrix).

  Textures - for accessing texture data
  - sampler1D, sampler2D, sampler3D: For sampling 1D, 2D, or 3D textures.
  - samplerCube: For cube maps (used in environment mapping).
  - sampler1DShadow, sampler2DShadow: For shadow mapping (used in 1D or 2D shadow textures).

ACCESSING VECTOR COMPONENTS

Accessing components
- For accessing coordinates: use x, y, z, w
- For accessing colors: use r, g, b, a
- Example: For a vec4 color:
  - color.rgba: Accesses all components as r, g, b, a.
  - color.xyzw: Accesses all components as x, y, z, w.
  - color.x, color.r: Accesses a single component.
- Keep in mind that it is not possible to mix notations - color.rz is invalid because it mixes color (r - red) and coordinate (z)

Swizzling:
- Swizzling is useful for extracting or reordering components without extra variables.
  - Vec3 colorRGB = color.rgb //extracts the r, g, b components
  - Coords.xz
  - Vec4 semiTransparent = vec4(someColor.rgb, 0.5) //creates a new vec4 with the rgb components of someColor and alpha of 0.5Con structors:

Constructors
- Constructors allow you to build vectors from smaller vectors or scalars
  vec4 semiTransparent = vec4(someColor, 0.5): Combines a vec3 (someColor) with a scalar (0.5) to make a vec4.

VARIABLE QUALIFIER
- GLSL uses qualifiers to define the role and behavior of variables in the shader pipeline

Qualifier	Meaning
nothing	Local variable (default scope, exists only within the shader).
const	Constant variable (cannot be modified after initialization).
(att)ribute	Input from the previous pipeline stage (e.g., vertex data in a vertex shader). Deprecated in modern OpenGL.
out	Output variable to the next stage (e.g., from vertex to fragment shader).
varying	(Deprecated) Used to pass data from vertex to fragment shader.
in	Input variable (modern replacement for attribute or varying).
in out	Same as in, but with cent (center interpolation, used in geometry shaders).
centroid in	Same as in, but with cent (center interpolation).
centroid out	Same as out, but with cent (center interpolation).

FUNCTION PARAMETER QUALIFIER
- GLSL uses qualifiers for function parameters to specify the input/output behavior

Qualifier	Meaning
nothing	Input parameter (default).
in	Input parameter (explicit).
out	Output parameter.
inout	Input/output parameter (can be modified).

Example:

```
void modifyValue(in float input, out float result, inout float counter) {
   result = input * 2.0;
   counter += 1.0;
}
```

- input is read-only, result is set by the function, and counter is both read and modified.

EMBEDDED FUNCTIONS

- GLSL provides built-in functions for common mathematical and graphics operations.
  - **Angle Conversions**:
    - radians, degrees: Convert between radians and degrees (e.g., radians(180.0) converts 180 degrees to π radians).
  - **Trigonometric Functions**:
    - sin, cos, tan, asin, acos, atan: Standard trigonometric functions (e.g., sin(0.5)).
  - **Exponential Functions**:
    - pow, exp, log, sqrt: Power, exponential, logarithm, and square root (e.g., pow(2.0, 3.0) computes $2^3 = 8$).
  - **Rounding**:
    - abs, floor, ceil: Absolute value, floor (round down), and ceiling (round up) (e.g., floor(3.7) returns 3.0).
  - **Comparison and Distances**:
    - min, max, length, distance: Minimum, maximum, vector length, and distance between vectors (e.g., length(vec3(1.0, 0.0, 0.0)) returns 1.0).
  - **Vector Operations**:
    - dot, cross, normalize: Dot product, cross product, and vector normalization (e.g., normalize(vec3(1.0, 1.0, 0.0)) returns a u nit vector).
  - **Texture Application**:
    - texture: Samples a texture (e.g., texture(sampler2D, coords) retrieves a color from a 2D texture).

VERTEX SHADER - GLSL 1.2 VS 1.3

Vertex Shader (GLSL 1.2)

Simple C-like program

```
#version 120
attribute vec3 a_Vertex; // input from the previous step
attribute vec3 a_Color;
varying vec4 color;      // output for the next step

void main(void)
{
```

Vertex Shader (GLSL 1.3)

Different naming of input/output

```
#version 130

in vec3 a_Vertex;
in vec3 a_Color;
out vec4 color;
```

```
attribute vec3 a_Vertex; // input from the previous step
attribute vec3 a_Color;
varying vec4 color;      // output for the next step

void main(void)
{
  gl_Position = vec4(a_Vertex, 1.0)
  color = vec4(a_Color, 1.0);
}
```

```
#version 130

in vec3 a_Vertex;
in vec3 a_Color;
out vec4 color;

void main(void)
{
  gl_Position = vec4(a_Vertex, 1.0)
  color = vec4(a_Color, 1.0);
}
```

FRAGMENT SHADER - GLSL 1.2 VS 1.3

Fragment shader (GLSL 1.2)

```
#version 120

// interpolated color from the vertex
attribute vec4 color;

void main(void) {
  // output var. up to GLSL 1.2
  gl_FragColor = color;
}
```

Fragment shader (GLSL 1.3)

```
#version 130

in vec4 color;
out vec4 outColor;

void main(void) {
  // gl_FragColor is obsolete
  outColor = color;
}
```

SENDING VARIABLES TO A SHADER

- Shaders need data to process, such as vertex positions or colors.
- Data can be sent as **uniforms** (same for all vertices) or **attributes** (unique per vertex).

  glUniformMatrix4fv(location, 1, transpose, matrix);
  - OpenGL function that sents a 4x4 matrix to the shader as a uniform
  - Location - the location of the uniform variable in the shader
  - 1 - number of matrices to send
  - Transpose - whether to transpose the matrix
  - Matrix - the matrix data
- Uniforms are used for data which do not change per vertex, like transformation matrices or lighting parameters

Registering a variable for sending

```
GLuint coordID; // index/id used for sending into a shader
glBindAttribLocation(m_programID, coordID, coordNameInShader);
```

- glBindAttribLocation: Binds an attribute in the shader to a specific location (index).
- Parameters:
  - m_programID: The ID of the shader program (created with glCreateProgram).
  - coordID: The index to assign to the attribute
  - coordNameInShader: The name of the attribute in the shader (e.g., a_Vertex).

ATTRIBUTE INITIALIZATION

- Before rendering, you need to enable vertex attributes (like positions or colors) so the shader can use them.
- After rendering, you disable them to clean up.
- The code:

```
glEnableVertexAttribArray(coordID); // enable vertex attr.
glEnableVertexAttribArray(colorID); // enable color attr.
someObject->render(); // render the object
glDisableVertexAttribArray(coordID); // disable vertex attr.
glDisableVertexAttribArray(colorID); // disable color attr.
//Enabling attributes tells OpenGL to use the data associated with those indices during rendering. Disabling them prevents unecessary processing after rendering.
```

The code above replaces the previously used code:

```
// 1 glEnableClientState(GL_VERTEX_ARRAY);
// 2 glEnableClientState(GL_COLOR_ARRAY);
```

READING DATA FROM A VBO

- Vertex Buffer Objects store vertex data in GPU memory
- glVertexAttribPointer tells OpenGL how to interpret the data in a VBO for a specific attribute (e.g., positions or colors).
- It's a critical step in the rendering pipeline, as it connects the raw data in the VBO to the shader's input attributes.

```
void glVertexAttribPointer(GLuint index, GLint size, GLenum type,
            GLboolean normalized, GLsizei stride,
            const GLvoid *pointer);
```

- **Parameters**:
  - index: The attribute index (e.g., coordID for positions); matches the attribute index set by glBindAttribLocation.
  - size: Number of components per attribute (e.g., 3 for a vec3 like x, y, z); for a 3D position (vec3), this is 3.
  - type: Data type (e.g., GL_FLOAT for floating-point values); typically GL_FLOAT for vertex data.
  - normalized: Whether to normalize the data (GL_TRUE or GL_FALSE); If GL_TRUE, maps integer data to [-1, 1] or [0, 1] (usually GL_FALSE for floats).
  - stride: Byte offset between consecutive attributes (0 if tightly packed); If the data is interleaved (e.g., position and color together), this specifies the distance between consecutive attributes.
  - pointer: Offset in the buffer where the data starts (e.g., 0 for the beginning).

Example of reading data from a VBO

- // 1 Bind the vertex buffer
  glBindBuffer(GL_ARRAY_BUFFER, m_vertexBuffer);
  // 2 Load data into shader
  glVertexAttribPointer(coordID, 3, GL_FLOAT, GL_FALSE, 0, vertices);
- // 3 Bind the color buffer
  glBindBuffer(GL_ARRAY_BUFFER, m_colorBuffer);
  // 4 Load data into shader
  glVertexAttribPointer(colorID, 3, GL_FLOAT, GL_FALSE, 0, colors);

- **glBindBuffer**: Binds a VBO to a target (GL_ARRAY_BUFFER for vertex data).
  - GL_ARRAY_BUFFER: The target for vertex attributes.
  - m_vertexBuffer, m_colorBuffer: The IDs of the VBOs for vertex positions and colors.
- **glVertexAttribPointer**: Specifies how to read the data from the VBO.
  - coordID, colorID: Attribute indices (set by glBindAttribLocation).
  - 3: 3 components (e.g., x, y, z for positions, r, g, b for colors).
  - GL_FLOAT: Data type.
  - GL_FALSE: No normalization.
  - 0: No stride (data is tightly packed).
  - vertices, colors: Pointers to the data (or offsets in the VBO)

Přednáška #4

VERTEX BUFFER OBJECTS

- Allows us to store data directly to the graphics card memory
- In the first three exercise lessons, the positions of triangles corners were stored in the computer main memory (CPU)
- VBOs let you store this data directly in the graphics card memory (GPU memory)

Přednáška #4

VERTEX BUFFER OBJECTS

- Allows us to store data directly to the graphics card memory
- In the first three exercise lessons, the positions of triangles corners were stored in the computer main memory (CPU)
- VBOs let you store this data directly in the graphics card memory (GPU memory)
- When rendering a scene, GPU can access the data directly from its own memory - this means you don't have to keep sending the data from the CPU to the GPU everytime you draw and you skip the CPU part altogether
- Starting with the OpenGL version of 3.0, VBOs are the only recommended way to handle vertex data.

RENDERING USING VBOS

- Generating name for a buffer - Create an ID (like a label) for the VBO
- Binding the buffer - Tell OpenGL which VBO you are working with
- Storing data in the buffer - Copy the vertex data into the VBO (into GPU memory)
- Drawing the data in the buffer - use the data to render the shape
- Removing buffer from memory - Once the VBO becomes unnecessary, remove from memory

GENERATING THE NAME

```
glGenBuffers(GLsizei n, GLuint *buffers);
```

- This function creates names (IDs) for VBOs.
- OpenGL guarantees that these names are unique (they won't overlap with other VBOs).
- Paremeters:
  - n: How many VBOs you want to create names for.
  - buffers: A pointer to an array where the names (IDs) will be stored.
- Example of generating a single name:
  - GLuint bufferID;
    glGenBuffers(1, &bufferID);
- Creates one VBO name and stores it in bufferID.

Releasing a Buffer:

```
glDeleteBuffers(1, &bufferID);
```

- Deletes the VBO with the given name (bufferID), freeing up the GPU memory.

CREATING A BUFFER

```
void glBindBuffer(GLenum target, GLuint buffer);
```

- Binds (activates) a VBO so you can work with it.
- Binding a buffer is like telling the worker, "Use this box (VBO) for the next steps." The target tells the worker what kind o f data the box will hold (e.g., vertex positions with GL_ARRAY_BUFFER).

- Parameters:
  - target: The type of buffer you're working with. Common types:
    - GL_ARRAY_BUFFER: For vertex data (positions, colors, etc.).
    - GL_ELEMENT_ARRAY_BUFFER: For indices (used in indexed rendering, like drawing with a list of vertex IDs).
  - buffer: The name (ID) of the VBO (bufferID from glGenBuffers).
- Example:
  glBindBuffer(GL_ARRAY_BUFFER, bufferID);
  - Binds bufferID to the GL_ARRAY_BUFFER target, meaning the next operations (like storing data) will affect this VBO.

STORING THE VERTICES DATA

```
void glBufferData(GLenum target, GLsizeiptr size, const GLvoid *data, GLenum usage);
```

- Copies vertex data into the VBO (into GPU memory)
- Calling this function multiple times OVERWRITES the original content
- glBufferData is like filling the storage box with materials (vertex data). You tell the worker:
  - Which box to use (target).
  - How much space the materials need (size).
  - Where the materials are (data).
  - How often you'll change the materials (usage).
- Parameters:
  - target: The type of buffer (e.g., GL_ARRAY_BUFFER).
  - size: The size of the data in bytes (e.g., sizeof(vertices)).
  - data: A pointer to the vertex data (e.g., an array of floats for positions).
  - usage: A hint to OpenGL about how the data will be used for performance optimization. Options:
    1. GL_STREAM_DRAW, GL_STREAM_READ, GL_STREAM_COPY: Data is set once and used a few times.
    2. GL_STATIC_DRAW, GL_STATIC_READ, GL_STATIC_COPY: Data is set once and used many times (most common for static shapes)
       - GL_STATIC_DRAW is often used for vertex data that doesn't change (e.g., a 3D model that stays the same).
    3. GL_DYNAMIC_DRAW, GL_DYNAMIC_READ, GL_DYNAMIC_COPY: Data is changed often and used many times.
    - Using the wrong usage hint (e.g., GL_STREAM_DRAW for data that never changes) will still work, but it won't be as efficient.

EXPLANATION OF THE KEYWORDS IN THE BUFFER USAGE

- When you store data in VBO using glBufferData, you specify the usage hint
- The usage hint is the combination of:
  - How frequently data is accessed (STREAM, STATIC, DYNAMIC)
  - Who creates and uses the data (DRAW, READ, COPY)

How frequently data s accessed

STREAM	Data will be written once, read a few times
STATIC	Data will be written once, read frequently
DYNAMIC	Data will be both written and read frequently

- **STREAM**: Use this if you'll set the data once and use it a few times (e.g., for temporary data).
- **STATIC**: Use this if you'll set the data once and use it many times (e.g., for a 3D model that doesn't change).
- **DYNAMIC**: Use this if you'll change the data often and use it often (e.g., for a shape that moves or deforms every frame).

Who Creates the Buffer Content?

Parameter	Meaning
DRAW	Filled by the app and used by OpenGL for rendering
READ	Filled by OpenGL and read by the app
COPY	Filled by OpenGL and used by OpenGL for rendering

- **DRAW**: The app (your program) provides the data, and OpenGL uses it to draw (most common for vertex data). Essentially used when w e are just providing data to OpenGL - to render a model for example
- **READ**: OpenGL fills the buffer (e.g., with data from the GPU), and the app reads it (e.g., for capturing a screenshot). Read is wh at we need data from OpenGL
- **COPY**: OpenGL fills the buffer and uses it for rendering (e.g., for GPU -generated data like transform feedback). When OpenGL both generated and reuses data

For example:

- GL_STATIC_DRAW: The app sets the data once (STATIC), and OpenGL uses it for rendering (DRAW).
- GL_DYNAMIC_DRAW: The app updates the data often (DYNAMIC), and OpenGL uses it for rendering (DRAW).

STORING DATA INTO A BUFFER

- The slide shows how to store vertex data for a set of triangles in a VBO:
- It's like filling a storage box with materials (the vertex data) and telling the worker (GPU), "You'll use this a lot for drawing, so keep it handy" (GL_STATIC_DRAW).

```
glBufferData(
   GL_ARRAY_BUFFER,      // buffer type
   sizeof(triangles),    // amount of data stored
   triangles,            // where are the data
   GL_STATIC_DRAW        // how it will be used
);
```

- GL_ARRAY_BUFFER: The VBO is for vertex data (like positions or colors).
- sizeof(triangles): The size of the data in bytes (e.g., if triangles is an array of floats).
- triangles: The actual data (e.g., an array of vertex positions).
- GL_STATIC_DRAW: The data will be set once and used many times for rendering.

- If you call glBufferData on a VBO that already has data, the old data will be erased, and the new data will replace it.
- If you try to store more data than the GPU has memory for, OpenGL will raise a GL_OUT_OF_MEMORY error.

STORING DATA INTO A BUFFER

- VBO can be initialized as empty and data can be added later using function glBufferSubData
- This method lets you set up the VBO in two steps:
  1) Reserve space in GPU memory (glBufferData with NULL).
  2) Fill the space later (glBufferSubData).

STORING DATA INTO A BUFFER

- VBO can be initialized as empty and data can be added later using function glBufferSubData
- This method lets you set up the VBO in two steps:
    1) Reserve space in GPU memory (glBufferData with NULL).
    2) Fill the space later (glBufferSubData).

```
void glBufferSubData(GLenum target, GLintptr offset, GLsizeiptr size, const GLvoid *data);
```

- Parameters
    - target: The type of buffer (e.g., GL_ARRAY_BUFFER).
    - offset: Where to start writing in the VBO (in bytes, 0 for the beginning).
    - size: The size of the data to write (in bytes).
    - data: The data to write.
- Essentially, empty VBO is created first used glBufferData

```
glBufferData(GL_ARRAY_BUFFER, sizeof(triangles), NULL, GL_STATIC_DRAW);
```

- NULL means no data is provided yet; it just reserves space.
- Then, you add data with glBufferSubData:

```
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(triangles), triangles);
```

POINTER DEFINITION

- After storing data in a VBO, OpenGL needs to be told how to read it
- This is done with glVertexAttribPointer (or glVertexPointer in older OpenGL), which was introduced in earlier slides.
- You must bind the VBO (with glBindBuffer) before setting the pointer. This tells OpenGL to read the data from the VBO instead of CPU memory.
- The pointer definition is like giving the GPU a map to find the data in the VBO:
    - "The data starts here (offset), has this many pieces per vertex (components), and is this type (e.g., integers or floats)."
- This step connects the VBO data to the shader's inputs (like vertex positions or colors).

- **Example** (without shaders, using the older fixed-function pipeline):

```
glVertexPointer(2, GL_INT, 0, 0);
```

- Tells OpenGL how to read vertex positions: 2 components (x, y), type GL_INT, stride 0, offset 0.

- **Example** (with shaders, using modern OpenGL):

```
glVertexAttribPointer(m_colAttr, 2, GL_INT, GL_FALSE, 0, 0);
```

- Tells OpenGL how to read an attribute (e.g., colors) for a shader: m_colAttr is the attribute index, 2 components, type GL_IN T, no normalization, stride 0, offset 0.

VBO INITILIAZATION

```
GLuint vertexID; // buffer ids'
GLuint colorID;
...
// name generation
glGenBuffers(1, &vertexID);
glGenBuffers(1, &colorID);
...
GLint triangles[] = {
   10, 10,
   320, 470,
   630, 10
};
GLfloat colors[] = {
   1.0, 0.0, 0.0,
   0.0, 1.0, 0.0,
   0.0, 0.0, 1.0
};


glBindBuffer(GL_ARRAY_BUFFER, vertexID);
glBufferData(GL_ARRAY_BUFFER, sizeof(triangles), triangles, GL_STATIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, colorID);
glBufferData(GL_ARRAY_BUFFER, sizeof(colors), colors, GL_STATIC_DRAW);
```

Explanation:

1. **Buffer IDs**:
    - GLuint vertexID;: Creates a variable to store the ID (name) of the VBO for vertex positions.
    - GLuint colorID;: Creates a variable to store the ID of the VBO for vertex colors.
2. **Name Generation**:
    - glGenBuffers(1, &vertexID);: Generates a unique ID for the vertex position VBO and stores it in vertexID.
    - glGenBuffers(1, &colorID);: Generates a unique ID for the color VBO and stores it in colorID.
3. **Vertex Data**:
    - GLint triangles[]: Defines an array of integers for the vertex positions of a triangle:
        - (10, 10): Bottom-left vertex.
        - (320, 470): Top vertex.
        - (630, 10): Bottom-right vertex.
        - These are 2D coordinates (x, y) in pixel space, forming a triangle.
4. **Color Data**:
    - GLfloat colors[]: Defines an array of floats for the colors of the vertices:
        - (1.0, 0.0, 0.0): Red for the first vertex.
        - (0.0, 1.0, 0.0): Green for the second vertex.
        - (0.0, 0.0, 1.0): Blue for the third vertex.
5. **Bind the Vertex VBO**:
    - glBindBuffer(GL_ARRAY_BUFFER, vertexID);: Tells OpenGL to use the VBO with ID vertexID for vertex data (GL_ARRAY_BUFFER).
6. **Store Vertex Data**:
    - glBufferData(GL_ARRAY_BUFFER, sizeof(triangles), triangles, GL_STATIC_DRAW);:
        - Copies the triangles array into the VBO.
        - sizeof(triangles): The size of the data in bytes.
        - triangles: The actual data (the vertex positions).
        - GL_STATIC_DRAW: A hint that the data won't change and will be used many times for rendering.
        - GL_STATIC_DRAW means the data is set once and used repeatedly, which is perfect for a triangle that doesn't change.
7. **Bind the Color VBO**:
    - glBindBuffer(GL_ARRAY_BUFFER, colorID);: Switches to the VBO with ID colorID.
8. **Store Color Data**:
    - glBufferData(GL_ARRAY_BUFFER, sizeof(colors), colors, GL_STATIC_DRAW);:
        - Copies the colors array into the VBO.
        - Same parameters as above, but for the color data.

ACTIVATION AND RENDERING (OLDER METHOD)

```
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);
glBindBuffer(GL_ARRAY_BUFFER, vertexID);
glVertexPointer(2, GL_INT, 0, 0);
glBindBuffer(GL_ARRAY_BUFFER, colorID);
glColorPointer(3, GL_FLOAT, 0, 0);
glDrawArrays(GL_TRIANGLES, 0, 3);
glDisableClientState(GL_COLOR_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);
```

Explanation:

1. **Enable Arrays**:
    - glEnableClientState(GL_VERTEX_ARRAY);: Tells OpenGL to use vertex position data.
    - glEnableClientState(GL_COLOR_ARRAY);: Tells OpenGL to use vertex color data.
      This is part of the older fixed-function pipeline (pre-OpenGL 3.0, before shaders became standard).
2. **Bind and Set Vertex Data**:
    - glBindBuffer(GL_ARRAY_BUFFER, vertexID);: Binds the VBO with vertex positions.
    - glVertexPointer(2, GL_INT, 0, 0);:
        - Tells OpenGL how to read the vertex data:
            - 2: Each vertex has 2 components (x, y).
            - GL_INT: The data type is integers.
            - 0: Stride (distance between consecutive vertices, 0 means tightly packed).
            - 0: Offset (start at the beginning of the VBO).
3. **Bind and Set Color Data**:
    - glBindBuffer(GL_ARRAY_BUFFER, colorID);: Binds the VBO with colors.

- 2: Each vertex has 2 components (x, y).
- GL_INT: The data type is integers.
- 0: Stride (distance between consecutive vertices, 0 means tightly packed).
- 0: Offset (start at the beginning of the VBO).

3. **Bind and Set Color Data**:
   - glBindBuffer(GL_ARRAY_BUFFER, colorID);: Binds the VBO with colors.
   - glColorPointer(3, GL_FLOAT, 0, 0);:
     - Tells OpenGL how to read the color data:
       - 3: Each color has 3 components (r, g, b).
       - GL_FLOAT: The data type is floats.
       - 0: Stride (tightly packed).
       - 0: Offset (start at the beginning).
4. **Draw the Triangle**:
   - glDrawArrays(GL_TRIANGLES, 0, 3);:
     - Draws the triangle:
       - GL_TRIANGLES: Draw as triangles.
       - 0: Start at the first vertex.
       - 3: Draw 3 vertices (one triangle).
5. **Disable Arrays**:
   - glDisableClientState(GL_COLOR_ARRAY);: Disables the color array.
   - glDisableClientState(GL_VERTEX_ARRAY);: Disables the vertex array.
   - This cleans up after rendering.

ACTIVATION AND RENDERING USING SHADERS

```
glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
glBindBuffer(GL_ARRAY_BUFFER, m_vertexBufferId);
glVertexAttribPointer(m_posAttr, 2, GL_FLOAT, GL_FALSE, 0, 0);
glBindBuffer(GL_ARRAY_BUFFER, m_colorBufferId);
glVertexAttribPointer(m_colAttr, 3, GL_FLOAT, GL_FALSE, 0, 0);
glDrawArrays(GL_TRIANGLES, 0, 3);
glDisableVertexAttribArray(1);
glDisableVertexAttribArray(0);
```

Explanation:

1. **Enable Attributes**:
   - glEnableVertexAttribArray(0);: Enables the attribute at index 0 (for positions).
   - glEnableVertexAttribArray(1);: Enables the attribute at index 1 (for colors).
   - These indices (0 and 1) match the locations of the attributes in the shader (e.g., posAttr and colAttr).
     If they are not assigned explicitly, OpenGL assigns them automatically based on the order they appear in shader.
2. **Bind and Set Vertex Data**:
   - glBindBuffer(GL_ARRAY_BUFFER, m_vertexBufferId);: Binds the VBO with vertex positions (assumed to be the same as vertexID).
   - glVertexAttribPointer(m_posAttr, 2, GL_FLOAT, GL_FALSE, 0, 0);:
     - Tells OpenGL how to read the position data for the shader:
       - m_posAttr: The attribute index for positions (set in the C++ code with attributeLocation).
       - 2: Each vertex has 2 components (x, y).
       - GL_FLOAT: The data type is floats (note: the triangles array is GLint, so this might be a mismatch unless the data was converted).
       - GL_FALSE: No normalization.
       - 0: Stride (tightly packed).
       - 0: Offset (start at the beginning).
3. **Bind and Set Color Data**:
   - glBindBuffer(GL_ARRAY_BUFFER, m_colorBufferId);: Binds the VBO with colors (assumed to be the same as colorID).
   - glVertexAttribPointer(m_colAttr, 3, GL_FLOAT, GL_FALSE, 0, 0);:
     - Tells OpenGL how to read the color data:
       - m_colAttr: The attribute index for colors.
       - 3: Each color has 3 components (r, g, b).
       - GL_FLOAT: The data type is floats.
       - GL_FALSE: No normalization.
       - 0: Stride and offset.
4. **Draw the Triangle**:
   - glDrawArrays(GL_TRIANGLES, 0, 3);: Draws the triangle (same as above).
5. **Disable Attributes**:
   - glDisableVertexAttribArray(1);: Disables the color attribute.
   - glDisableVertexAttribArray(0);: Disables the position attribute.

Přednáška #5

RENDERING A SCENE

- Rendering a 3D scene can compared to taking a photo with camera.
- The steps are the following:
  1. Camera - setting the camera into the scene
     - This step is about positioning the camera in the 3D world
     - The view transformation defines where the camera is, where is it looking and which direction is "up".
       This is done with view matrix which transform the whole scene so that the camera appears to be at the origin looking down on negative z-axis
  2. Object positioning – Insert object into the scene – Model transformation:
     - This step is about placing objects in the scene
     - The model transformation moves, rotates and scales the object to position it where you want in the 3D world
     - In OpenGL, this is done with the model matrix which transforms the object's vertices from its local space to the world space (where the object is placed in the scene)
  3. Lenses – Selecting a lens – Projection transformation:
     - This step is about defining how the 3D scene is projected onto a 2D screen - like choosing a lens for your camera
     - The projection transformation determines the field of view (how wide the camera sees), the aspect ration (screen widthxheight), and the near/far clipping planes (what is TOO CLOSE and what is TOO FAR to see)
     - These operations are done with a projection matrix - there are two common types:
       - Perspective projection - Mimics human vision with a vanishing point (objects farther away appear smaller)
       - Orthographic projection - No perspective (objects do not get smaller with distance) - used primarily for 2D and technical drawings
  4. Shot – Mapping the final image into the window – Cropping, etc.:
     - Maps the projected 2D image onto the screen (the window where your program is running)
     - In OpenGL, this is handled by the viewport transformation - which scales and shifts the 2D image to fit the window
     - It also includes things like cropping parts of the scene that are outside the camera view

MATRICES UPDATES

- In 3D graphics, it is required to take a 3D object and show it on a 2D screen. The modelview and projection matrices are tool s that help you achieve that. They are like instructions that tell the computer how to move and shape the 3D object so it loo ks right on the screen.
- These matrices are continuously changed during the rendering process.
- They are continuously sent to the shader (for every single frame)

Modelview Matrix

- Its purpose is to:
  - **Move the object around** (or rotate/scale it) - imagine you have a toy car. The modelview matrix can move it to a new spot, spin it around or make it bigger/smaller
  - **Move the camera** (or rotate it) - it also decides where you are looking from. For example, you are looking at the car from the front, the side or above?
- Think of it like:
  - You place the toy car on a table (that is the model part of the modelview matrix - positioning the object)
  - You decide where to stand and point your camera to take a picture (that is the view part - positioning the camera)
- The modelview matrix combines these two steps into one. It takes object's points (vertices) and adjusts them based on where the object is and where the camera is looking.

Projection matrix

- The projection matrix takes the 3D scene (after the modelview matrix has positioned everything) and squishes it into a 2D picture that can fit on your screen.
- These matrices are sent to the shader as uniforms (variables that are the same for all vertices), so the shader can apply the transformations to each vertex
- It is like taking a photo with camera:
  - Perspective projection - this makes things that are more further away smaller - like in real life. For example, if you are looking down a road, the road will be narrower the further away you are. This is what most 3D games utilize to make things look more realistic.
  - Ortographic projection - This does not make things smaller as they get farther away. It is like drawing a blueprint - everything stays the same size, no matter how far it is. This is often used for 2D games and technical drawings.
- It is the projection matrix that decides how the 3D world gets flattened into a 2D image. IT also sets up viewing area to setup which part of the 3D world will be visible
- The projection matrix is usually the same throughout the program, except in two cases:
  - Change the Appearance of the Scene (CAD) - In a computer-aided design (CAD) program, you might switch between perspective and orthographic projections to view the scene differently.
  - Change the Shape of the Window - If the window is resized (e.g., the user stretches the window), the aspect ratio changes, so the projection matrix needs to be updated to avoid stretching the scene.
- Defines how the 3D scene is projected onto a 2D screen. It usually stays the same unless the window size or projection type changes.
- Like the model-view matrix, the projection matrix must also be sent to the shader, but it's updated less often.

3. Projection & Transform Settings (Note)
- "We do not use any projection/transform command directly, but we use similar commands that generate for us matrices that are provided to the shaders.
- In modern OpenGL (3.0+), you don't use built-in commands like glMatrixMode, glLoadMatrix, or gluPerspective (from older OpenGL) to set up transformations directly.
- Instead, you calculate the matrices (model, view, projection) yourself (or use a library like GLM to do it for you) and send them to the shader as uniforms.

ORTHOGRAPHIC PROJECTION

- Ortographic projection is a way to project a 3D scene onto a 2D screen where the size of the objects does not change with their distance from the camera

- "We do not use any projection/transform command directly, but we use similar commands that generate for us matrices that are provided to the shaders.
- In modern OpenGL (3.0+), you don't use built-in commands like glMatrixMode, glLoadMatrix, or gluPerspective (from older OpenGL) to set up transformations directly.
- Instead, you calculate the matrices (model, view, projection) yourself (or use a library like GLM to do it for you) and send them to the shader as uniforms.

ORTHOGRAPHIC PROJECTION

- Ortographic projection is a way to project a 3D scene onto a 2D screen where the size of the objects does not change with their distance from the camera

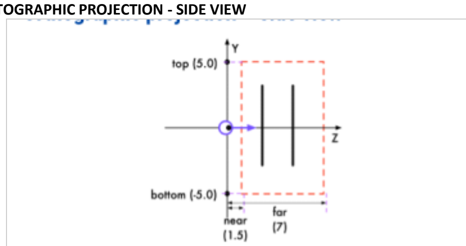
- Ortographic projection is a way to project a 3D scene onto a 2D screen where the size of the objects does NOT change with their distance from the camera
- This is different from the perspective projection where objects that are farther away appear smaller
- Objects are projected using parallel lines - therefore their size remains the same regardless of how far they are from the camera
- This makes ortographic projection useful for applications where you want to preserve the exact size and shape of objects, such as CAD, 2D games and technical drawings

Setting ortographic projection

```
void glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);
```

- This function (from older OpenGL) sets up an orthographic projection by defining a viewing volume (a 3D box) that determines what part of the scene is visible.
- Parameters:
  - left, right: The x-coordinates of the left and right sides of the viewing volume.
  - bottom, top: The y-coordinates of the bottom and top of the viewing volume.
  - near, far: The z-coordinates of the near and far planes of the viewing volume.
    - **near**: The distance from the camera to the near plane (anything closer than this is clipped and not drawn).
    - **far**: The distance from the camera to the far plane (anything farther than this is clipped).
  - These values define the range of the x, y, and z axes that will be visible.
  - The glOrtho function creates a box-shaped viewing volume:
    - The sides of the box are defined by left, right, bottom, and top (for x and y).
    - The depth of the box is defined by near and far (for z).
  - Anything inside this box will be projected onto the screen, and anything outside will be clipped (not drawn)
- In modern OpenGL (3.0+), glOrtho is deprecated. Instead, you calculate the orthographic projection matrix yourself (or use a library like GLM) and send it to the shader as a uniform, as mentioned in earlier slides.

ORTOGRAPHIC PROJECTION - SIDE VIEW



- The diagram shows a side view of the viewing volume for orthographic projection:
  - The camera is at the origin (0, 0, 0), looking along the z-axis.
  - The y-axis goes up, and the z-axis goes into the screen.
  - The viewing volume is a box with:
    - top at y = 5.0.
    - bottom at y = -5.0.
    - near plane at z = 1.5.
    - far plane at z = 7.0.
  - Two vertical lines (at y = 5.0 and y = -5.0) show the boundaries of the viewing volume.
- Any object inside this box will be visible on the screen, and its size won't change based on its z-coordinate (distance from the camera).

MATHEMATICAL BACKGROUND

- The orthographic projection is represented by a **4x4 matrix**, which is applied to each vertex to transform it from 3D space to a 2D screen space.

Matrix:

$$\begin{bmatrix} \frac{2}{\text{right}-\text{left}} & 0 & 0 & -\frac{\text{right}+\text{left}}{\text{right}-\text{left}} \\ 0 & \frac{2}{\text{top}-\text{bottom}} & 0 & -\frac{\text{top}+\text{bottom}}{\text{top}-\text{bottom}} \\ 0 & 0 & -\frac{2}{\text{far}-\text{near}} & -\frac{\text{far}+\text{near}}{\text{far}-\text{near}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Explanation:

- This matrix transforms the 3D coordinates of a vertex (x, y, z) into a normalized device coordinate (NDC) space, where:
  - x, y, and z are mapped to the range [-1, 1].
  - This NDC space is then mapped to the screen by the viewport transformation.
- **How It Works:**
  - The first row scales and shifts the x-coordinate:
    - $\frac{2}{\text{right}-\text{left}}$: Scales the x-coordinate to fit within [-1, 1].
    - $-\frac{\text{right}+\text{left}}{\text{right}-\text{left}}$: Shifts the x-coordinate so the center of the viewing volume maps to 0.
  - The second row does the same for the y-coordinate.
  - The third row scales and shifts the z-coordinate:
    - $-\frac{2}{\text{far}-\text{near}}$: Scales the z-coordinate.
    - $-\frac{\text{far}+\text{near}}{\text{far}-\text{near}}$: Shifts the z-coordinate (the negative sign is because OpenGL's z-axis points into the screen).
  - The fourth row (0, 0, 0, 1) ensures the w-coordinate of the vertex remains 1 (important for homogeneous coordinates in 3D graphics).

Example:

- If `left = -5`, `right = 5`, `bottom = -5`, `top = 5`, `near = 1.5`, `far = 7`:
  - First row: $\frac{2}{5-(-5)} = 0.2, -\frac{5+(-5)}{5-(-5)} = 0.$
  - Second row: $\frac{2}{5-(-5)} = 0.2, -\frac{5+(-5)}{5-(-5)} = 0.$

- If `left = -5`, `right = 5`, `bottom = -5`, `top = 5`, `near = 1.5`, `far = 7`:
  - First row: $\frac{2}{5-(-5)} = 0.2$, $-\frac{5+(-5)}{5-(-5)} = 0$.
  - Second row: $\frac{2}{5-(-5)} = 0.2$, $-\frac{5+(-5)}{5-(-5)} = 0$.
  - Third row: $-\frac{2}{7-1.5} = -0.3636$, $-\frac{7+1.5}{7-1.5} = -1.545$.
  - The matrix becomes:

$$\begin{bmatrix} 0.2 & 0 & 0 & 0 \\ 0 & 0.2 & 0 & 0 \\ 0 & 0 & -0.3636 & -1.545 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Purpose:

- This matrix is applied to each vertex in the vertex shader to project the 3D scene onto a 2D plane without perspective distortion.

- The slide provides a link for more details: http://learnwebgl.brown37.net/08_projections/projections_ortho.html .

PERSPECTIVE PROJECTION

- Perspective projection mimics how human vision works: objects that are farther away from the camera appear smaller.
- The scene is defined by a **frustum** (a pyramid with the top cut off), which represents the camera's field of view.
- The closer an object is to the camera, the larger it appears. This is because a closer object takes up a larger portion of the frustum's cross-section (like a larger slice of the pyramid).
- This creates a realistic sense of depth, commonly used in 3D games, simulations, and visualizations.

glFrustrum

```
void glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);
```

- This function defines a perspective projection by specifying the frustum's dimensions at the near plane (left, right, bottom, top) and the distances to the near and far planes (near, far).
- **Parameters**:
  - left, right: The x-coordinates of the left and right edges of the near plane.
  - bottom, top: The y-coordinates of the bottom and top edges of the near plane.
  - near: The distance from the camera to the near plane (anything closer than this is clipped).
  - far: The distance from the camera to the far plane (anything farther than this is clipped).
- glFrustum sets up the frustum by defining its shape at the near plane (a rectangle defined by left, right, bottom, and top) and its depth (near to far).
- The frustum tapers from the near plane to the camera's position (at the origin), creating the perspective effect.

gluPerspective

```
void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble near, GLdouble far);
```

- This is a helper function from the GLU (OpenGL Utility) library that simplifies setting up a perspective projection.
- **Parameters**:
  - fovy: The field of view angle in the y-direction (in degrees), defining how wide the camera's view is vertically.
  - aspect: The aspect ratio of the viewing window (width/height), which adjusts the horizontal field of view.
  - near: The distance to the near plane.
  - far: The distance to the far plane.
- gluPerspective is easier to use than glFrustum because:
  - Instead of specifying the exact dimensions of the near plane (left, right, bottom, top), you provide a field of view angle (f ovy) and aspect ratio (aspect).
  - The function calculates the left, right, bottom, and top values for you based on fovy, aspect, and near.
    - If fovy = 60 degrees and aspect = 4/3 (a common screen ratio), the function computes the frustum's dimensions to create a 60 -degree vertical field of view and a 4:3 horizontal -to-vertical ratio.
- Both glFrustum and gluPerspective are deprecated in modern OpenGL (3.0+). In modern OpenGL, you calculate the perspective projection matrix yourself (or use a library like GLM) and send it to the shader as a uniform.

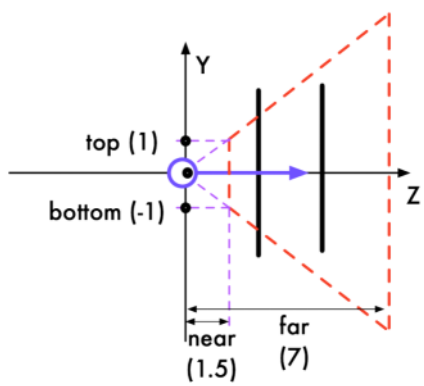


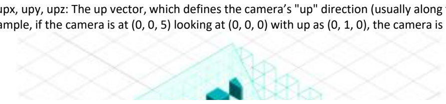
Diagram:
- The diagram shows a side view of the frustum for perspective projection:
  - The camera is at the origin (0, 0, 0), looking along the z -axis.
  - The y-axis goes up, and the z-axis goes into the screen.
  - The frustum is a pyramid with its top cut off:
    - The **near plane** is at z = 1.5, with top = 1 and bottom = -1 (defining the y-boundaries at the near plane).
    - The **far plane** is at z = 7.
  - The frustum tapers from the near plane to the camera's position at the origin, shown by the red dashed lines.
  - The x-boundaries (left and right) are not shown in this side view but would be similar to bottom and top.
- The frustum is the 3D volume that defines what the camera can see:
  - At the near plane (z = 1.5), the frustum's height is from bottom = -1 to top = 1.
  - The frustum extends from the near plane (z = 1.5) to the far plane (z = 7).
  - The tapering shape means that objects closer to the near plane appear larger, while objects closer to the far plane appear sm aller.
- Anything inside the frustum will be visible on the screen; anything outside (e.g., closer than the near plane or farther than the far plane) will be clipped.

VIEW MATRIX

- The **view matrix** defines the position and orientation of the camera (or observer) in the 3D scene.
- It transforms the entire scene so that the camera appears to be at the origin (0, 0, 0) looking down the negative z-axis, with the y-axis pointing up.

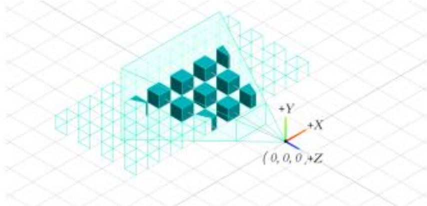
```
- gluLookAt(
    GLdouble eyex, GLdouble eyey, GLdouble eyez,
    GLdouble centerx, GLdouble centery, GLdouble centerz,
    GLdouble upx, GLdouble upy, GLdouble upz
  );
```

- **Parameters**:
  - eyex, eyey, eyez: The position of the camera (eye point).
  - centerx, centery, centerz: The point the camera is looking at (target point).
  - upx, upy, upz: The up vector, which defines the camera's "up" direction (usually along the y-axis, e.g., (0, 1, 0))
- For example, if the camera is at (0, 0, 5) looking at (0, 0, 0) with up as (0, 1, 0), the camera is positioned 5 units along the z-axis, looking toward the origin, with the y-axis as "up."



- upx, upy, upz: The up vector, which defines the camera's "up" direction (usually along the y-axis, e.g., (0, 1, 0))

- For example, if the camera is at (0, 0, 5) looking at (0, 0, 0) with up as (0, 1, 0), the camera is positioned 5 units along the z-axis, looking toward the origin, with the y-axis as "up."

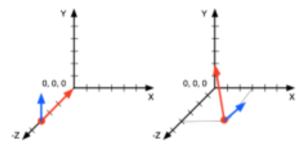


Source: https://jsantell.com/model-view-projection/

COMMON OBSERVER SETTING

```
gluLookAt(
   0.0, 0.0, -5.0,  // Camera position
   0.0, 0.0, 0.0,   // Look-at point
   0.0, 1.0, 0.0    // Up vector
);
```

- What It Means:
  - Camera Position: (0, 0, -5) – The camera is positioned 5 units *behind* the origin along the z-axis (in OpenGL, the positive z-axis points out of the screen, so z = -5 is into the screen); Moving the camera to (0, 0, -5) ensures it's far enough back to see objects placed around the origin.
  - Look-At Point: (0, 0, 0) – The camera looks at the origin; Looking at (0, 0, 0) keeps the focus on the center of the scene.
  - Up Vector: (0, 1, 0) – The y-axis is "up" for the camera; the up vector (0, 1, 0) ensures the camera isn't tilted (the y-axis is up, like in the real world).



Common "step back" and complex orientation.

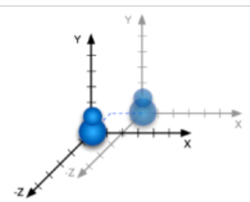
```
// lets create matrix using embedded functions. E.g.:
QMatrix4x4 projectionMatrix;
// make it identity matrix
projectionMatrix.setToIdentity();
// fill it with projection matrix ortho(-2.0, 2.0, -2.0, 2.0, 0.0, 100.0);
projectionMatrix.ortho(-2.0, 2.0, -2.0, 2.0, 0.0, 100.0);
// send it to the shader
program->setUniformValue(m_matrixUniform, projectionMatrix);
// do not forget on depth testing!
```

- This code snippet shows how to set up a projection matrix (not a view matrix, despite the section title) in modern OpenGL using Qt's QMatrix4x4 class.
  - **Create a Matrix** - QMatrix4x4 projectionMatrix;: Creates a 4x4 matrix object to store the projection matrix.
  - **Set to Identity** - projectionMatrix.setToIdentity();: Initializes the matrix as an identity matrix (a matrix that doesn't change the coordinates when multiplied).
  - **Fill with Projection Matrix**:
    - projectionMatrix.ortho(-2.0, 2.0, -2.0, 2.0, 0.0, 100.0);:
      - Sets up an orthographic projection (not perspective, which is a bit inconsistent with the slide's focus on gluLookAt).
      - Parameters: left = -2.0, right = 2.0, bottom = -2.0, top = 2.0, near = 0.0, far = 100.0.
      - This defines a viewing volume that is 4 units wide (x: -2 to 2), 4 units tall (y: -2 to 2), and 100 units deep (z: 0 to 100).
  - **Send to Shader**:
    - program->setUniformValue(m_matrixUniform, projectionMatrix);:
      - Sends the matrix to the shader as a uniform variable (m_matrixUniform).
  - A reminder to enable depth testing (glEnable(GL_DEPTH_TEST))so that OpenGL can properly handle overlapping objects (e.g., closer objects obscure farther ones).

- To set up a view matrix in modern OpenGL, you'd use a function like QMatrix4x4::lookAt (or GLM's equivalent glm::lookAt):

```
QMatrix4x4 viewMatrix;
viewMatrix.setToIdentity();
viewMatrix.lookAt(
   QVector3D(0.0, 0.0, -5.0),  // Camera position
   QVector3D(0.0, 0.0, 0.0),   // Look-at point
   QVector3D(0.0, 1.0, 0.0)    // Up vector
);
```

Object translation




```
glTranslatef(GLfloat x, GLfloat y, GLfloat z)
```

- This function translates (moves) the object by the specified offsets along the x, y, and z axes.
- **Example**: If you call glTranslatef(2.0, 3.0, 1.0), the object will move 2 units along the x-axis, 3 units along the y-axis, and 1 unit along the z-axis.

MATHEMATICAL BACKGROUND

Translation Matrix (T)

- A translation matrix moves an object by some offset (t_x, t_y, t_z) along the x, y, and z axes.
- The matrix is:

```
[ 1  0  0  t_x ]
[ 0  1  0  t_y ]
[ 0  0  1  t_z ]
[ 0  0  0   1  ]
```

- For example, if t_x = 1, t_y = 0, t_z = -1, the matrix translates the object 1 unit along the x-axis and -1 unit along the z-axis.

Applying Translation (X' = T * X)

- If a point X has coordinates (x, y, z, w), the transformed point X' after translation is calculated as:
  - X' = T * X
- Example calculation:
  - Translation matrix T with t_x = 1, t_y = 0, t_z = -1:

text Colla

```
T = ⌈ 1  0  0  1 ⌉
```

text ⌅ Colla

```
T = [ 1  0  0  1 ]
    [ 0  1  0  0 ]
    [ 0  0  1 -1 ]
    [ 0  0  0  1 ]
```

- Point X = (10, 10, 1, 1):

text ⌅ Colla

```
X' = T * X = [ 1  0  0  1 ] * [ 10 ] = [ 10 + 1 ] = [ 11 ]
             [ 0  1  0  0 ]   [ 10 ]   [ 10 + 0 ]   [ 10 ]
             [ 0  0  1 -1 ]   [  1 ]   [  1 - 1 ]   [  0 ]
             [ 0  0  0  1 ]   [  1 ]   [   1    ]   [  1 ]
```

So, the point (10, 10, 1) moves to (11, 10, 0).

Inverse transformation

$$X = T^{-1} X' =$$

$$\begin{bmatrix} 1 & 0 & 0 & -t_x \\ 0 & 1 & 0 & -t_y \\ 0 & 0 & 1 & -t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} 1x + 0y + 0z + -t_x w \\ 0x + 1y + 0z + -t_y w \\ 0x + 0y + 1z + -t_z w \\ 0x + 0y + 0z + 1w \end{bmatrix}$$

$$= \begin{bmatrix} x' - t_x w \\ y' - t_y w \\ z' - t_z w \\ w' \end{bmatrix} = \begin{bmatrix} x' - t_x \\ y' - t_y \\ z' - t_z \\ 1 \end{bmatrix} = \begin{bmatrix} 20 - 10 \\ 10 - 0 \\ 10 - 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 10 \\ 10 \\ 10 \\ 1 \end{bmatrix}$$

Inverse Transformation ($X = T^{-1} * X'$)

- The inverse of a translation matrix undoes the translation. For T with (t_x, t_y, t_z), the inverse T^{-1} has (-t_x, -t_y, -t_z):

text ⌅ Collapse ≡ Wrap ⎘ Copy

```
T⁻¹ = [ 1  0  0  -t_x ]
      [ 0  1  0  -t_y ]
      [ 0  0  1  -t_z ]
      [ 0  0  0    1  ]
```

- Using the same example (t_x = 1, t_y = 0, t_z = -1):

text ⌅ Collapse ≡ Wrap ⎘ Copy

```
T⁻¹ = [ 1  0  0  -1 ]
      [ 0  1  0   0 ]
      [ 0  0  1   1 ]
      [ 0  0  0   1 ]
```

Applying T^{-1} to X' = (11, 10, 0, 1) brings it back to X = (10, 10, 1, 1).

OBJECT ROTATION

```
glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z)
```

- angle: The angle of rotation in degrees.
- (x, y, z): The axis of rotation (a vector). For example:
  - (1, 0, 0) → Rotate around the x-axis.
  - (0, 1, 0) → Rotate around the y-axis.
  - (0, 0, 1) → Rotate around the z-axis.

Rotation matrices

- Rotation around the x-axis by angle θ:

text ⌅ Collapse ≡ Wrap ⎘ Copy

```
[ 1    0       0      0 ]
[ 0  cos(θ)  -sin(θ)  0 ]
[ 0  sin(θ)   cos(θ)  0 ]
[ 0    0       0      1 ]
```

Rotation around the y-axis by angle θ:

```
[ 0  sin(θ)  cos(θ)  0 ]
[ 0    0       0      1 ]
```

- Rotation around the y-axis by angle θ:

text Collapse Wrap Copy

```
[ cos(θ)  0  sin(θ)  0 ]
[   0     1    0     0 ]
[-sin(θ)  0  cos(θ)  0 ]
[   0     0    0     1 ]
```

- Rotation around the z-axis by angle θ:

text Collapse Wrap Copy

```
[ cos(θ) -sin(θ)  0  0 ]
[ sin(θ)  cos(θ)  0  0 ]
[   0       0     1  0 ]
[   0       0     0  1 ]
```

Example: Rotation Around the Z-Axis

- The example uses `glRotatef(45, 0, 0, 1)`, which rotates the object 45 degrees around the z-axis.
- The rotation matrix R for θ = 45° (π/4 radians) is:
  - cos(45°) = √2/2 ≈ 0.707
  - sin(45°) = √2/2 ≈ 0.707

text Collapse Wrap Copy

```
R = [ cos(45°) -sin(45°)  0  0 ]
    [ sin(45°)  cos(45°)  0  0 ]
    [   0         0       1  0 ]
    [   0         0       0  1 ]
  = [ 0.707  -0.707   0  0 ]
    [ 0.707   0.707   0  0 ]
    [  0       0      1  0 ]
    [  0       0      0  1 ]
```

- Apply this to a vertex at (10, 10, 1, 1):

text Collapse Wrap Copy

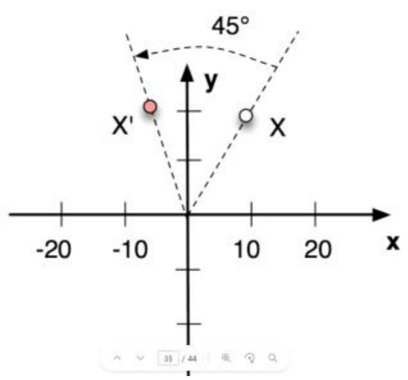
```
X' = R * X = [ 0.707  -0.707   0  0 ] * [ 10 ] = [ 0.707*10 + (-0.707)*10 ] = [ 0 ]
             [ 0.707   0.707   0  0 ]   [ 10 ]   [ 0.707*10 + 0.707*10   ]   [ 14.14 ]
             [  0       0      1  0 ]   [  1 ]   [          1            ]   [  1    ]
             [  0       0      0  1 ]   [  1 ]   [          1            ]   [  1    ]
```

So, the point (10, 10, 1) rotates to approximately (0, 14.14, 1).

So, the point (10, 10, 1) rotates to approximately (0, 14.14, 1).

Sketch of the Result

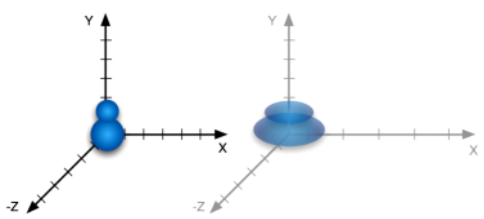
- The diagram shows a 2D view (ignoring the z-axis) of the rotation:
  - The original point (10, 10) is marked with a red X.
  - After a 45° rotation around the z-axis, it moves to (0, 14.14), marked with a red O.
  - The axes are scaled to [-20, 20] for clarity.



SCALE OF AN OBJECT

`glScalef(GLfloat s_x, GLfloat s_y, GLfloat s_z)`

- s_x, s_y, s_z: Scaling factors along the x, y, and z axes.
- For example, glScalef(1.0, 1.0, 1.0) leaves the object unchanged, while glScalef(2.0, 2.0, 2.0) doubles the size in all directions.



SCALING MATRIX

- The scaling matrix is:

text   Collapse   Wrap   Copy

```
S = [ s_x  0    0    0 ]
    [ 0    s_y  0    0 ]
    [ 0    0    s_z  0 ]
    [ 0    0    0    1 ]
```

- Example: If s_x = 1/2, s_y = 1/2, s_z = 1 (scale down by half in x and y, no change in z):

text   Collapse   Wrap   Copy

```
S = [ 1/2  0    0    0 ]
    [ 0    1/2  0    0 ]
    [ 0    0    1    0 ]
    [ 0    0    0    1 ]
```

Applying Scaling

- For a vertex at (10, 10, 1, 1):

text   Collapse   Wrap   Copy

```
X' = S * X = [ 1/2  0    0    0 ] * [ 10 ] = [ 1/2 * 10 ] = [ 5 ]
             [ 0    1/2  0    0 ]   [ 10 ]   [ 1/2 * 10 ]   [ 5 ]
             [ 0    0    1    0 ]   [ 1 ]    [    1     ]   [ 1 ]
             [ 0    0    0    1 ]   [ 1 ]    [    1     ]   [ 1 ]
```

The point (10, 10, 1) scales to (5, 5, 1).

Diagram

- The diagram shows the blue object (two spheres connected by a line) before and after scaling. The scaled object is smaller in the x and y directions but unchanged in the z direction.

COMPOSITION OF TRANSFORMATIONS

The final section shows how to combine multiple transformations.

Example: Two Transformations

- **Left**: Translation by (20, 0, 0) followed by rotation by 45° around the z-axis.
- **Right**: Rotation by 45° around the z-axis followed by translation by (20, 0, 0).

Matrix Multiplication is Not Commutative

- The order of transformations matters because matrix multiplication is not commutative (i.e., A * B ≠ B * A).
- **Translation then Rotation**:
  - First, translate the point (10, 10, 1) by (20, 0, 0) to (30, 10, 1).
  - Then, rotate (30, 10, 1) by 45° around the z-axis:

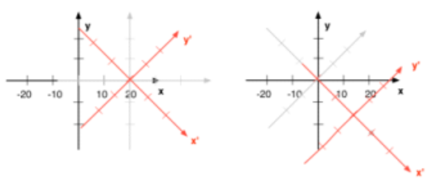
text   Collapse   Wrap   Copy

- First, translate the point (10, 10, 1) by (20, 0, 0) to (30, 10, 1).
- Then, rotate (30, 10, 1) by 45° around the z-axis:

text Collapse Wrap Copy

```
X' = [ 0.707  -0.707   0  0 ] * [ 30 ] = [ 0.707*30 + (-0.707)*10 ] = [ 14.14 ]
     [ 0.707   0.707   0  0 ]   [ 10 ]   [ 0.707*30 + 0.707*10  ]   [ 28.28 ]
     [ 0       0       1  0 ]   [  1 ]   [         1            ]   [  1    ]
     [ 0       0       0  1 ]   [  1 ]   [         1            ]   [  1    ]
```

- **Rotation then Translation:**
  - First, rotate (10, 10, 1) by 45° around the z-axis to (0, 14.14, 1).
  - Then, translate (0, 14.14, 1) by (20, 0, 0) to (20, 14.14, 1).



Left: translate (20, 0, 0) and rotation (45, 0, 0, 1)
Right: rotation (45, 0, 0, 1) and translation (20, 0, 0)

Diagram

- The diagram shows the results:
  - **Left (Translate then Rotate)**: The point ends up at (14.14, 28.28).
  - **Right (Rotate then Translate)**: The point ends up at (20, 14.14).
- The two paths are different because the rotation is applied around the origin (0, 0, 0). Translating first moves the point farther from the origin, so the rotation has a larger effect.

Přednáška #6

Přednáška #7

Přednáška #8

Přednáška #9

Přednáška #10

Přednáška #11

Přednáška #12