(3) Buffer Objects

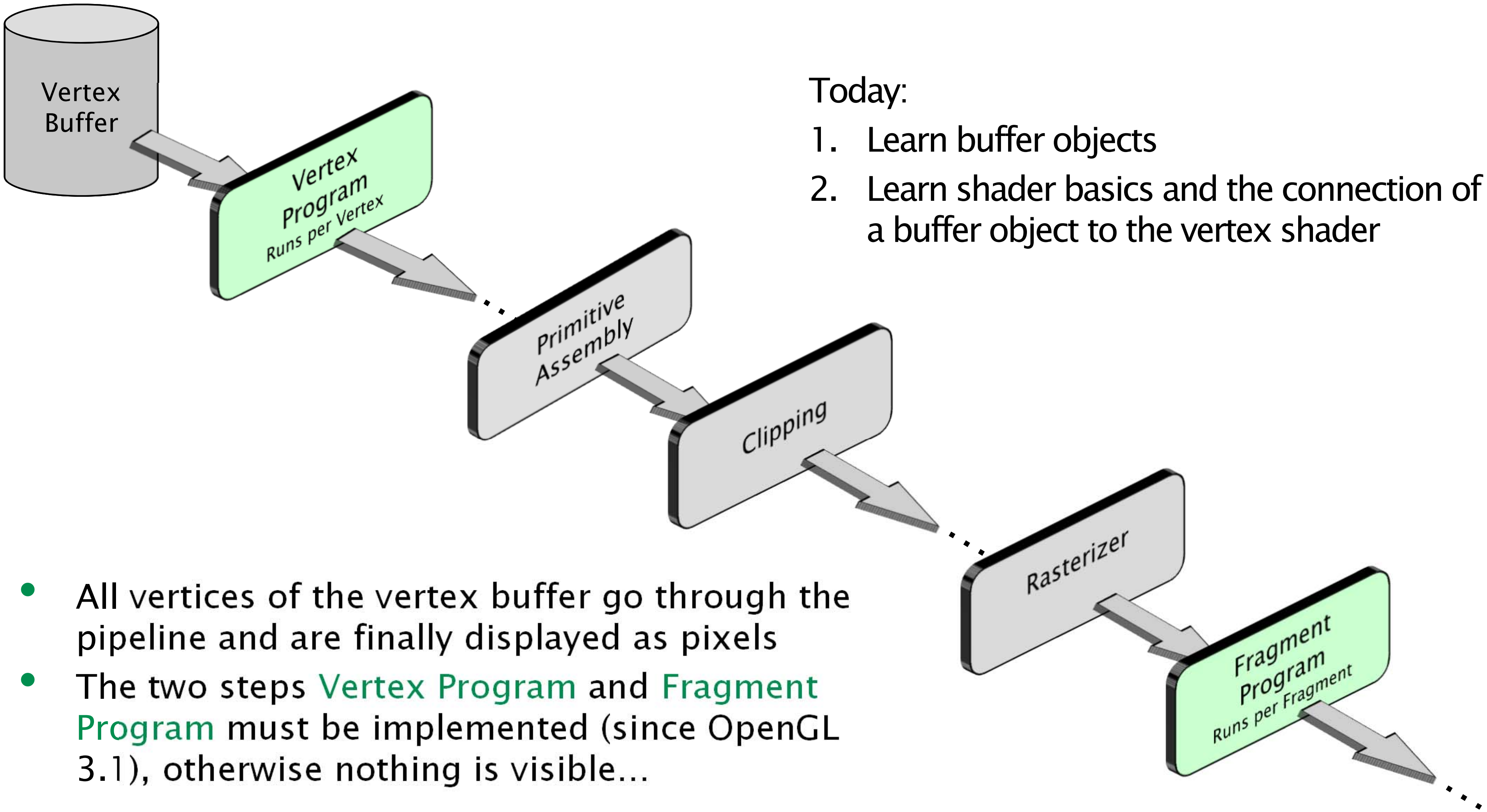
GPU Programming
T. Grosch

TU Clausthal

Today

- Buffer Objects
- Repeat Vertex Buffer Objects and Vertex Array Objects (but with a new semantic)
- Uniform Buffer Objects
- Vertex and Fragment Shader Intro

Simplified OpenGL Pipeline

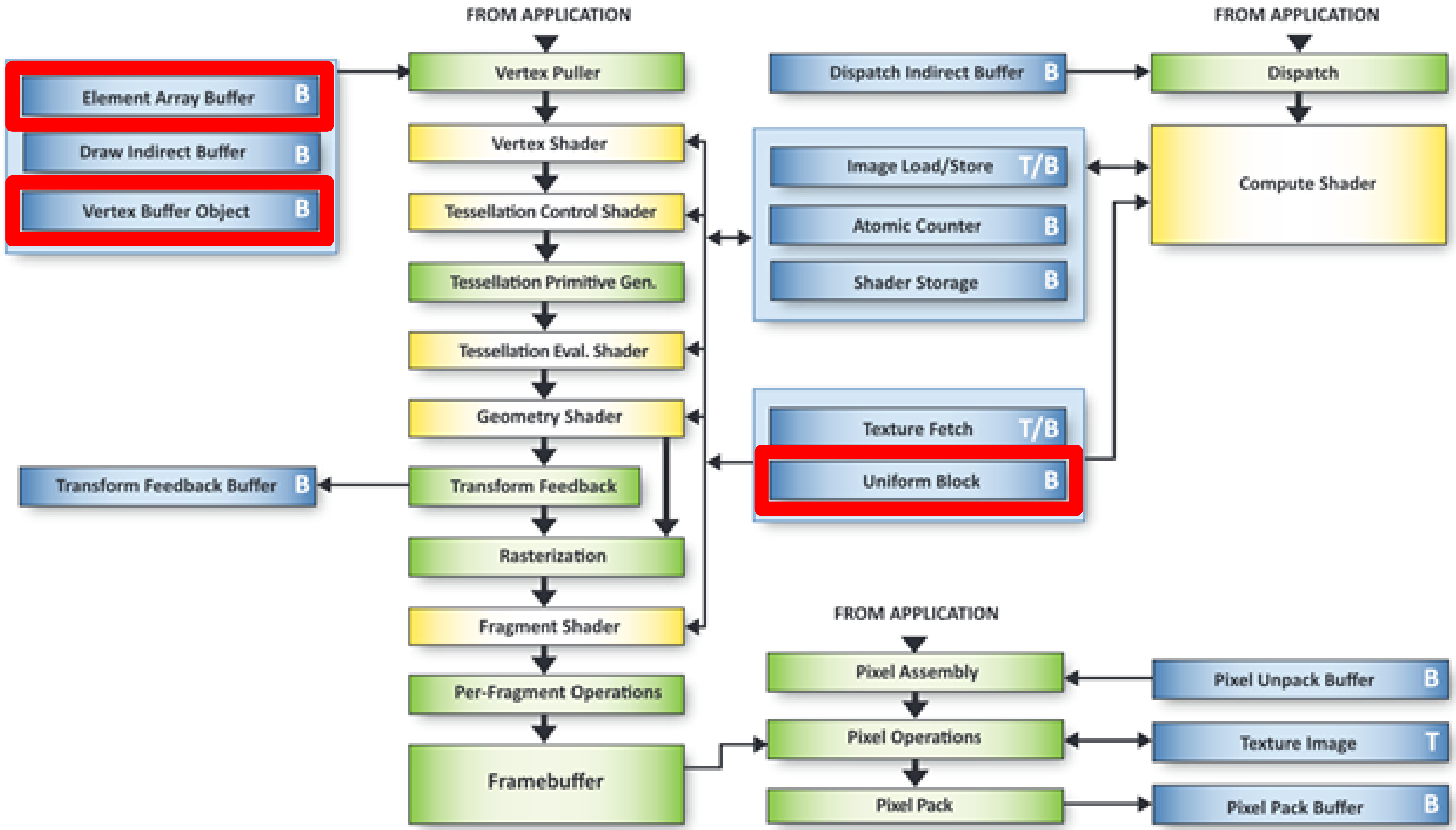


Today:

1. Learn buffer objects
2. Learn shader basics and the connection of a buffer object to the vertex shader

- All vertices of the vertex buffer go through the pipeline and are finally displayed as pixels
- The two steps Vertex Program and Fragment Program must be implemented (since OpenGL 3.1), otherwise nothing is visible…

OpenGL 4.4


FROM APPLICATION
Element Array Buffer B
Draw Indirect Buffer B
Vertex Buffer Object B
Vertex Puller
Vertex Shader
Tessellation Control Shader
Tessellation Primitive Gen.
Tessellation Eval. Shader
Geometry Shader
Transform Feedback Buffer B
Transform Feedback
Rasterization
Fragment Shader
Per-Fragment Operations
Framebuffer
Dispatch Indirect Buffer B
Image Load/Store T/B
Atomic Counter B
Shader Storage B
Texture Fetch T/B
Uniform Block B
FROM APPLICATION
Dispatch
Compute Shader
FROM APPLICATION
Pixel Assembly
Pixel Operations
Pixel Pack
Pixel Unpack Buffer B
Texture Image T
Pixel Pack Buffer B

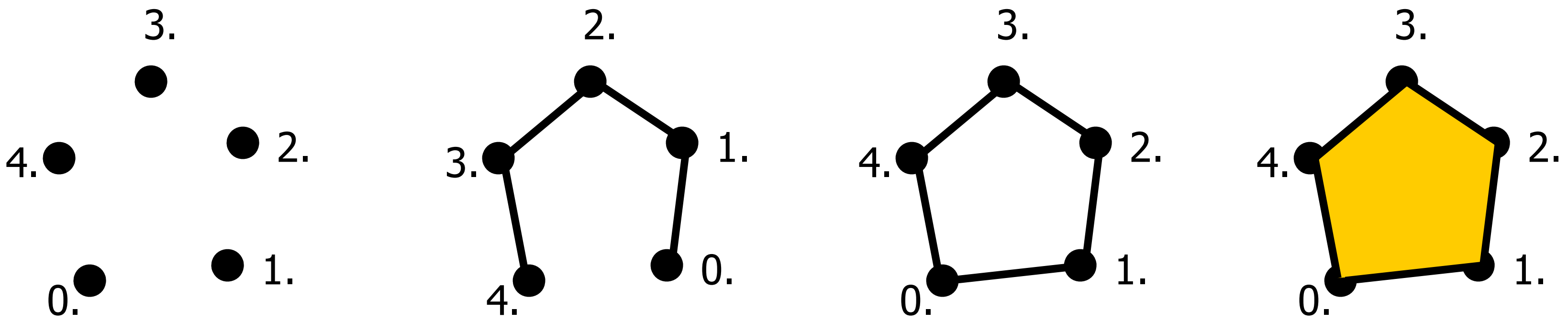
OpenGL Buffers

- Buffer = Block of data in (GPU) memory with additional description
  - OpenGL uses buffers at several places, not only as vertex data
  - Examples:
    - GL_ARRAY_BUFFER : Vertex Data
    - GL_ELEMENT_ARRAY_BUFFER : Index Data
    - PIXEL_(UN)PACK_BUFFER : RGB Color Data
    - GL_TRANSFORM_FEEDBACK_BUFFER : Vertex Data
    - GL_SHADER_STORAGE_BUFFER : Shader Input/Output Data
    - GL_UNIFORM_BUFFER : Shader Input Data
    - …

Vertex Buffer Object

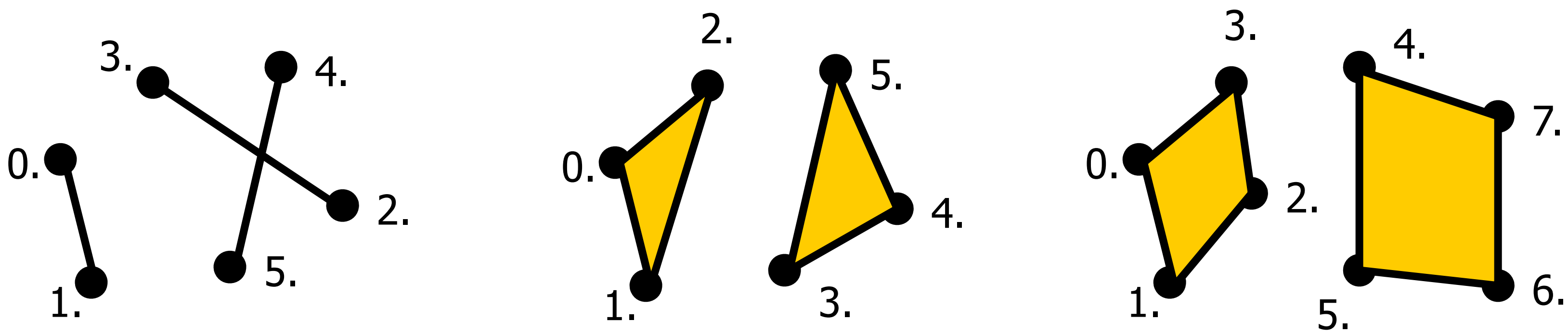
- We start with the Vertex Buffer Object
- Data = xyz positions of the vertices
- Goal: Draw a mesh consisting of graphical primitives
  - Triangles, Lines, Points, …
- Optional: Additional vertex information
  - Vertex Normals
  - Vertex Colors
  - Vertex Texture Coordinates
  - …

Primitive Types in OpenGL (Excerpt)



GL_POINTS GL_LINE_STRIP GL_LINE_LOOP GL_POLYGON

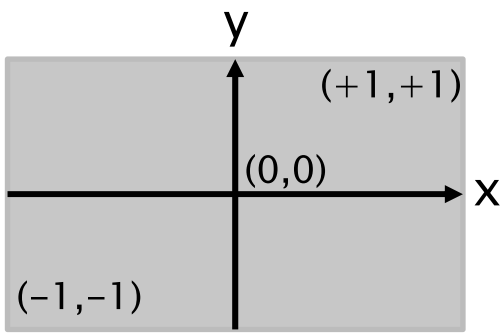
Multiple Primitives



GL_LINES GL_TRIANGLES GL_QUADS

Rasterization using OpenGL

- The default coordinates for vertices are not pixel coordinates !
  - The standard coordinate system in OpenGL ranges from −1 to +1 in both directions



- Definition:
  - The x axis points to the right
  - The y axis points upward
    - Attention: In GLFW, the y axis points down !
  - The origin is in the center of the image

- All objects with coordinates outside the range [−1,+1] are not visible !

Drawing with OpenGL

1. The complete vertex data is transferred (as a block) from CPU to GPU and remains there (usually).
2. A description for the vertex data is added, which describes the memory layout (start address, number of vertices, how many coordinates 2D/3D, float or int as base type, …)
3. Using a simple draw-command, the whole block can be drawn (without any further data transmission)

This type of geometry data is called a **Vertex Buffer Object (VBO)**

Vertex Buffer Checklist

- Define the vertex coordinates, colors, normals… (as C arrays)
- Generate Buffer Object(s) `glGenBuffer, glBindBuffer`
- Allocate memory `glBufferStorage`
- Copy the vertex coordinates data `glBufferSubData`
- Generate a Vertex Array Object `glGenVertexArrays` and `glBindVertexArray`
- Describe and enable the vertex attributes `glVertexAttribFormat, glEnableVertexAttribArray`
- Set the bindings for the attributes `glVertexAttribBinding, glBindVertexBuffer`
- Define the attribute numbers in the vertex shader `(layout(location = …))`
- Draw the geometry `glDrawArrays`

Description of the Commands

- We will go through all these commands and their parameters (which is quite a lot…)
- Some of them are related to the input of the vertex shader (we assume that there is a vertex shader and explain this later)

- Afterwards we will have a look at a small example to understand the connection between the commands
  - The commands are slightly different to those used in Computer Graphics 1 for an older OpenGL version

Vertex Buffer Object Init

- OpenGL reserves IDs for different VBOs, such an ID can be generated with:

```
GLint vboID;
glGenBuffer( 1 , &vboID );
```

- Afterwards we can use

```
glBindBuffer( GL_ARRAY_BUFFER, vboID );
```

to bind the buffer for upcoming buffer related commands

Allocating Memory for the VBO

- Assigning vertex data to a VBO is implemented with

```
glBufferStorage(target, size, data, flags);
```

- `target` is set to `GL_ARRAY_BUFFER` (for now)
- `size` is the size in bytes
- `data` is the address in main memory
- `flags` is set to `GL_DYNAMIC_STORAGE_BIT` or none (0) (for now).

- Using this command, the vertex data is transferred from CPU to GPU. In case of `data == NULL,` only the GPU memory is allocated.

Assigning Partial Vertex Data to a VBO

- (Re-)assigning (partial) vertex data to an existing VBO is implemented with

```
glBufferSubData(target, offset, size, data);
```

- **`target`** is set to **`GL_ARRAY_BUFFER`** (for now)
- **`offset`** is added to the buffer start address and decides where the **`data`** should be copied to (measured in bytes). Can be set to 0 if the whole image should be copied.
- **`size`** is the size in bytes
- **`data`** is the address in main memory

Vertex Array Object

- In addition to the Vertex Buffer Object(s), we need a **Vertex Array Object (VAO)** to define how the buffer data is **streamed** into the vertex shader

- VAOs come in two different flavors:
  - Aggregated buffer bindings (old)
    One VAO per object required
  - Separated: only specifies the format
    One VAO per different vertex shader input

- To create and use a VAO call **`glGenVertexArrays`** and **`glBindVertexArray`**

Vertex Array Object Init

- OpenGL reserves IDs for different VAOs, such an ID can be generated with:

```
GLint vaoID;
glGenVertexArrays( 1 , &vaoID );
```

- Afterward we can use

```
glBindVertexArray( vaoID );
```

to define (bind) the current VAO. The following commands are then related to this VAO.

Detailed Data Description

- Up to now, we only know the total size of the geometry data, but we do not know the **memory layout** of the vertex data. Therefore, we need an additional format description:

```
glVertexAttribFormat(attribIndex, size, type,
                     normalized, offset);
```

- **`attribIndex`**: Attribute number in the vertex shader (later)
- **`size`**: number of components per vertex (2 for xy, 3 for xyz, ...)
- **`type`**: Base type, typically **`GL_FLOAT`** or **`GL_UNSIGNED_BYTE`**
- **`normalized`**: Integer types are converted to [-1,1] or [0,1] values (we set this to **`GL_FALSE`** for now)
- **`offset`**: byte offset within each vertex (data can be interleaved: e.g. positions and normals in a single buffer). Can be set to 0 for simple (non-interleaved) buffers.

Activate Vertex Attribute

- Each vertex attribute must be activated:

```
glEnableVertexAttribArray(attribIndex);
```

  otherwise is does not work (!)

- Typical vertex attributes are
  - Position, normal, color, texture coordinate, …

Bind a Vertex Buffer

- To use a buffer as vertex buffer for the draw call we need to bind it to a so-called **binding index**:

```
glBindVertexBuffer( bindingIndex, vboID,
                    offset , stride );
```

- **`bindingIndex`**: Binding index of the vertex buffer (this is not the attribute index!)
- **`vboID`**: buffer object ID to be assigned to the binding index
- **`stride`**: Gap between two successive elements in memory (in bytes)
- **`offset`**: is added to the address given by **`glBufferStorage`**. Can be set to zero in most cases.

Note: In Computer Graphics 1, we use stride = 0 for tightly packed vertices, because the effective stride is computed automatically. This does no longer work here !

Vertex Attribute Binding

- Connect a VBO (given by the binding index) to the input of the vertex shader (given by the attribute index)
- Which attribute is taken from which buffer:

```
glVertexAttribBinding( attribIndex,
                       bindingIndex );
```

- **`attribIndex`**: The attribute location in the shader
- **`bindingIndex`**: The vertex buffer from which the attribute is taken

- Must be called after **`glVertexAttribFormat`**

Draw the VBO (finally…)

- After defining all the VBO data, we can draw the VBO using

```
glDrawArrays(mode, first, count);
```

- Here **`mode`** is the geometry type, e.g. **`GL_TRIANGLES`** or **`GL_LINES`**.
- Draw **`count`** vertices starting from index **`first`**. The vertex coordinates are not neccessary here.

Clean up

- VAO and VBO can be deleted with

```
glDeleteBuffers(…);

glDeleteVertexArrays(…);
```

Example

- To explain all these commands, we use geometry with **positions**, **normals** and **texture coordinates** (3 attributes)
- We assume that we have two triangles = 6 vertices
- For positions and normals, we use 3 floats for xyz coordinates (=12 bytes each)
- For texture coordinates we use 2 floats for st coordinates (= 8 bytes each)

- We store positions and normals **interleaved** in one buffer object
  - Like this: position1 normal1 position2 normal2 …
- Texture coordinates are stored in a second buffer object
  - This is an arbitrary design decision, we could also use three individual buffers or only a single buffer with three attributes interleaved

Define Vertex Coordinates

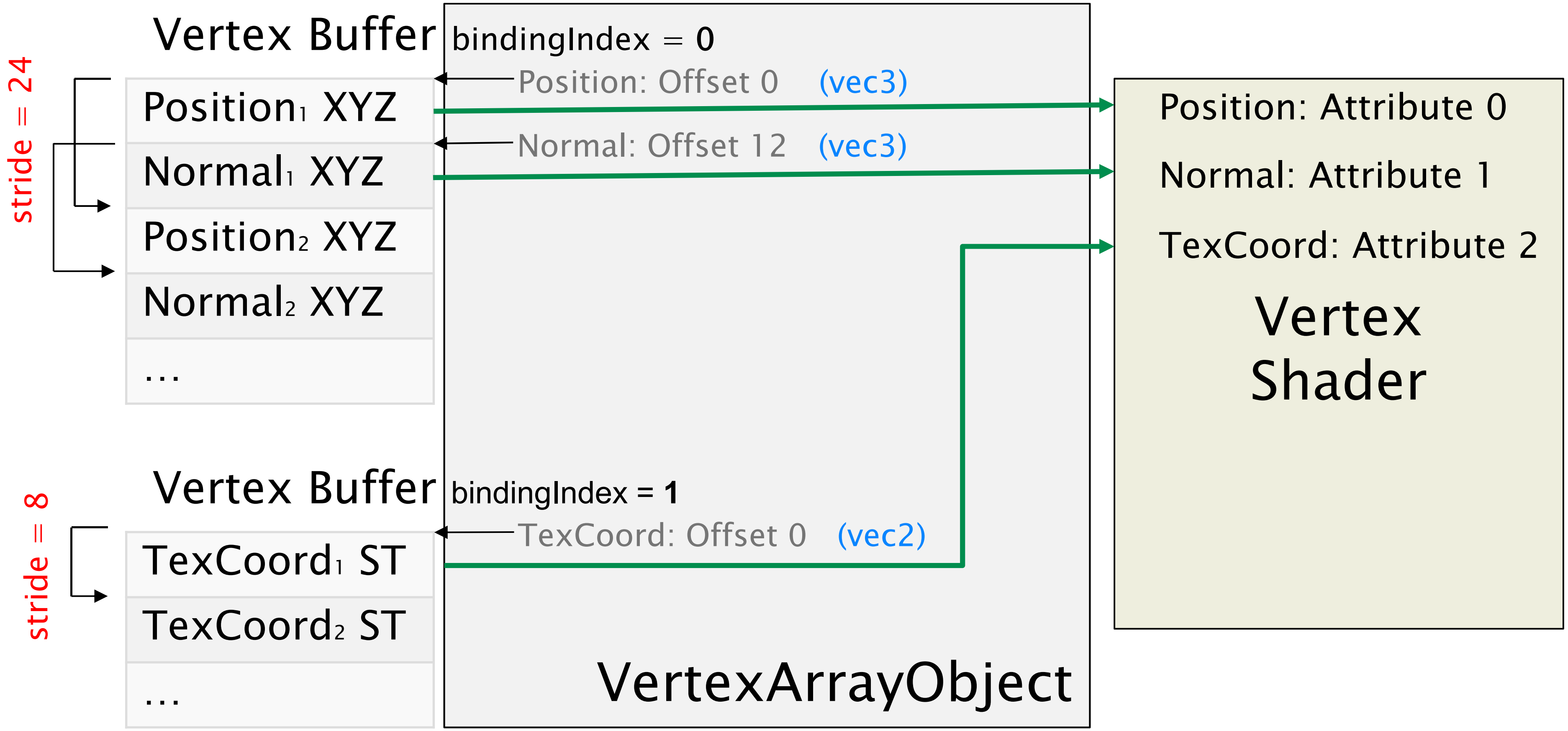
- The vertex data is defined completely, e.g. as a C array:

```
GLfloat positions_normals[6][6] = {   // interleaved
    {0.0, 0.0, 0.0,   0.0, 0.0, 1.0}, // vertex 1 normal 1
    {1.0, 0.0, 0.0,   0.0, 0.0, 1.0}, // vertex 2 normal 2
    {1.0, 1.0, 0.0,   0.0, 0.0, 1.0}, // vertex 3 normal 3
    {0.0, 0.0, 0.0,   1.0, 0.0, 0.0}, // vertex 4 normal 4
    {0.0, 1.0, 0.0,   1.0, 0.0, 0.0}, // vertex 5 normal 5
    {0.0, 1.0, 1.0,   1.0, 0.0, 0.0}  // vertex 6 normal 6
};
GLfloat texcoords[6][2] = { // 6 texcoords
    {0.0, 0.0}, {0.0, 1.0}, {1.0, 0.0},
    {0.0, 1.0}, {0.0, 0.0}, {1.0, 0.0}
};
```

Example: Create two VBOs

```
GLuint vboPosNormal, vboTexCoord;
GLfloat positions_normals[6][6] = { … };
GLfloat texcoords[6][2] = { … };

glGenBuffers(1, &vboPosNormal);
glBindBuffer(GL_ARRAY_BUFFER, vboPosNormal);
glBufferStorage(GL_ARRAY_BUFFER, 6 * 6 * sizeof(float),
                positions_normals, 0);

glGenBuffers(1, &vboTexCoord);
glBindBuffer(GL_ARRAY_BUFFER, vboTexCoord);
glBufferStorage(GL_ARRAY_BUFFER, 6 * 2 * sizeof(float),
                texcoords, 0);
```

Example: VAO Overview



Example: Create a VAO and set Connections

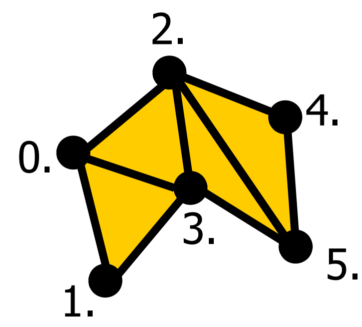
```
GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

glEnableVertexAttribArray(0);        // position
glVertexAttribFormat(0, 3, GL_FLOAT, GL_FALSE, 0);
glVertexAttribBinding(0, 0);

glEnableVertexAttribArray(1);        // normal
glVertexAttribFormat(1, 3, GL_FLOAT, GL_FALSE, 12);
glVertexAttribBinding(1, 0);

glEnableVertexAttribArray(2);        // texture coords
glVertexAttribFormat(2, 2, GL_FLOAT, GL_FALSE, 0);
glVertexAttribBinding(2, 1);
```

Example: Draw the VBO

```
void display()
{
  glClear(…);
  glBindVertexArray(vao);
  glBindVertexBuffer(0, vboPosNormal, 0, 24);
  glBindVertexBuffer(1, vboTexCoord, 0, 8);
  glDrawArrays(GL_TRIANGLES, 0, 6);
}
```

Note that there are redundancies in OpenGL:
For example, we assign the binding index 0 to the VBO vboPosNormal,
although vboPosNormal is already a unique ID for this VBO…
But for the connection with the shader (glVertexAttribBinding) we need the binding index.

Mesh Variants: Indexed VBOs



- Geometry is often described as a connected **Mesh**, where one vertex belongs to multiple triangles
- Up to now, we repeat the vertices → increased memory requirement for multiple points
- Therefore, we can use indexing for VAOs:
  - First we use non-connected list of all vertices
  - Furthermore we use an index-list which describes which vertices should be connected to triangles

Vertex List
x_0, y_0
x_1, y_1
x_2, y_2
x_3, y_3
x_4, y_4
x_5, y_5

Index List
0, 1, 3
0, 3, 2
2, 3, 5
2, 5, 4

- In the example, we have 6 vertices with indices instead of 4*3=12 vertices → less memory
  - One vertex needs 2*4 = 8 bytes
  - 12 vertices need 12*8 = **96 bytes**
  - three indices need 3*2 = 6 bytes (short index 16 bit)
  - 6 vertices with 4 triangle index list need only
    6*8 + 4*6 = **72 bytes**

Assign Index Data

- In addition to the VBO for the vertex data, another buffer for the index data is created using

```
glGenBuffers(1, &ibo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);
glBufferStorage(GL_ELEMENT_ARRAY_BUFFER, size, data, 0);
```

- the index data is marked with `GL_ELEMENT_ARRAY_BUFFER` (instead of `GL_ARRAY_BUFFER`)
  - (no good names here in OpenGL: GL_VERTEX_BUFFER and GL_INDEX_BUFFER would be better…)

- Again, `data` is the address in main memory and `size` is the size in bytes.

Draw VBO with Indices

- After defining vertex and index data, we can draw with

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboIndex);
glDrawElements(mode, indicesSize, indicesType, offset);
```

- `mode` defines the type of geometry, e.g. `GL_TRIANGLES` or `GL_LINES`.
- `indicesSize` is the number of indices
- `indicesType` defines the base type of the index data
  - Note that only unsigned integer types can be used here e.g. `GL_UNSIGNED_INT`.
  - Attention: The command does not work if a signed type is used, e.g. `GL_INT`
- `offset`: is added to the index address given by `glBufferStorage`. Can be set to zero in most cases.

Remark: The indices (elements) are **not** a vertex attribute and therefore not used in a VAO

Example: Create VBO with Indices

```
void vboIndexedInit()
{
   GLuint vbo, ibo;
   GLfloat vertices[6][2] = { … };
   GLuint indices[3*4] = { 0,1,3, 0,3,2, … };

   // generate VBO for vertices…

   glGenBuffers(1, &ibo);
   glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);
   glBufferStorage(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices),
   indices, 0);

   // generate VAO, define formats and set all connections…
}
```

Example: Draw VBO with Index List

```
void display()
{
   glClear(…);
   glBindVertexArray(…);
   glBindVertexBuffer(…);
   glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);
   glDrawElements(GL_TRIANGLES, 3*4, GL_UNSIGNED_INT, NULL);
}
```

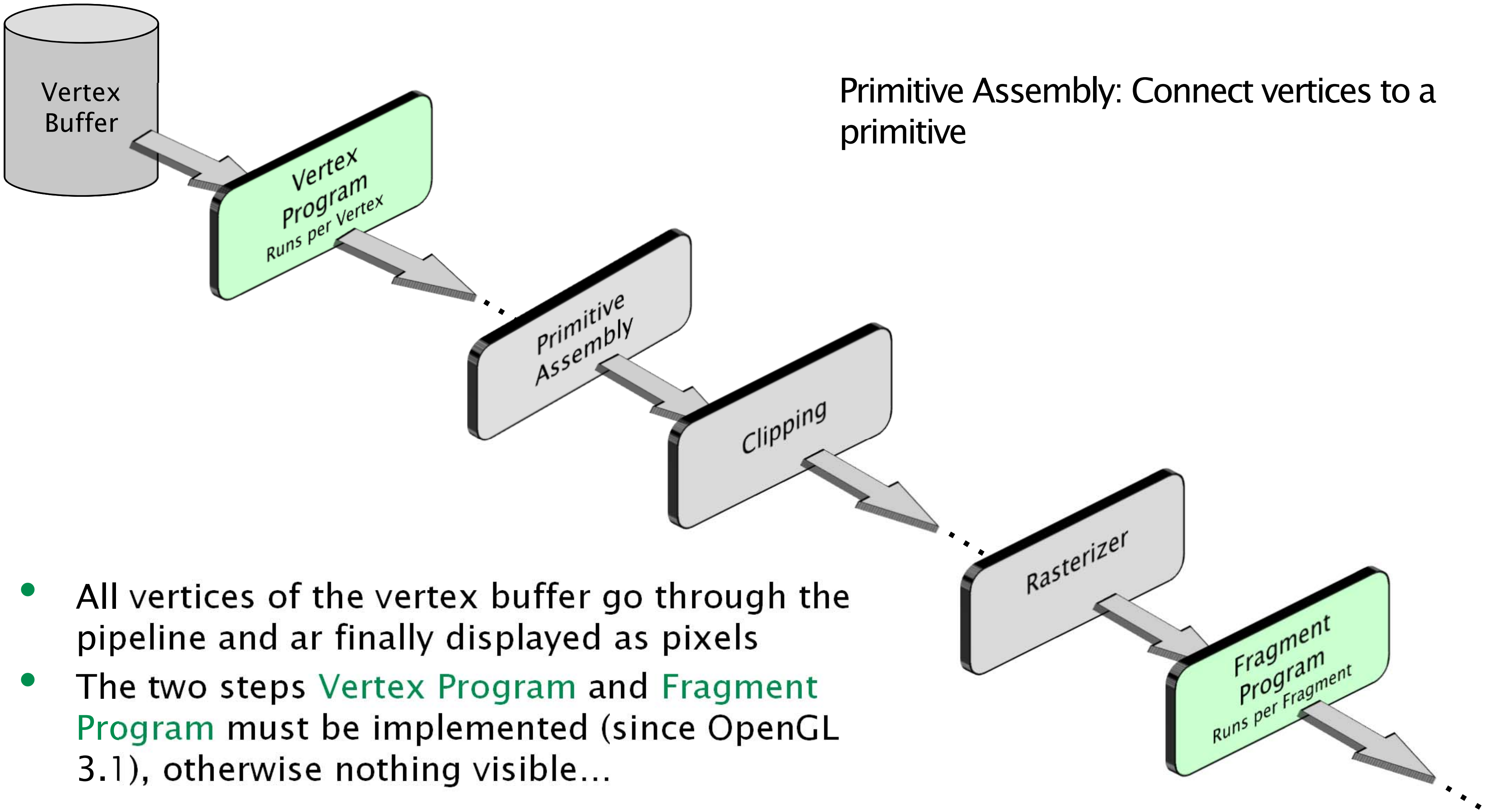
Drawing with modern OpenGL

- What happens if we draw the buffer objects using the shown commands→ black screen ?
- Although buffer objects are quite complicated we still need more to be able to display an image
- The reason: each vertex goes through a **pipeline** before it is displayed at a pixel coordinate
- Parts of the pipeline must be implemented

Introduction to
Vertex and Fragment Shaders

TU Clausthal

Simplified OpenGL Pipeline



Primitive Assembly: Connect vertices to a primitive

- All vertices of the vertex buffer go through the pipeline and ar finally displayed as pixels
- The two steps Vertex Program and Fragment Program must be implemented (since OpenGL 3.1), otherwise nothing visible...

Vertex Shader (or Vertex Program)

- …is called per vertex
  - One vertex is passed to the next pipeline stage
  - The vertex can be modified in the Vertex Shader
    - Transformation, Projectionen, set color → later
  - All other vertices of the geometry are unknown
  - It is not possible to generate new vertices, there is also no (direct) way to remove to vertex
- Simple Vertex Shader
  - pass through

```
#version 440 core
layout (location = 0) in vec4 vPosition; // vertex attribute 0
void main()
{
     gl_Position = vPosition;
}
```

Short Explanation: Vertex Shader

- First Line
  - OpenGL Version number (here: 4.4)
  - `core`: New OpenGL without backwards compatibility
    - (different models have different input/output syntax…)
- `gl_Position`: Output, this vertex is passed to the next stage
- `vPosition`: Input (Vertex position from VBO)

- This is the **connection with the VBOs**:

```
layout (location = 0) in vec4 vPosition
```

This must fit to the description of the **vertex attributes**:

```
glVertexAttribFormat(0, 4, GL_FLOAT, GL_FALSE, 0);
glVertexAttribBinding(0, 0);
```

Fixed Pipeline Stages

- Primitive Assembly
  - After the Vertex Shader, the vertices are assembled to primitives (triangles, lines, …)

- Clipping
  - The primitives are clipped at the visible range [-1,1] ^2
    - e.g. Cohen-Sutherland for lines
    - Sutherland-Hodgman for polygons

- Rasterization
  - The visible range of a primitive is subdivided in pixels (fragments). The difference between pixel and fragment will be explained later.

Fragment Shader (or Pixel Shader, Fragment Program)

- The Fragment Shader is called during the rasterization for each generated pixel (fragment)
  - The main task of the Fragment Shader is the definition of the pixel color
  - The Fragment Shader only knows the current pixel, directly reading/writing from/to other pixels is impossible

- Simple example: draw all primitives in red

```
#version 440 core
out vec3 fColor;
void main()
{
    fColor = vec3(1.0, 0.0, 0.0);
}
```

Fragment Shader (or Pixel Shader, Fragment Program)

- Again, the first line sets the version number
- Typically, the Fragment Shader has a single output (the pixel color)
  - the name `fColor` is arbitrary, using `out` defines it as output
  - In case of multiple outputs, we need to describe the output using `layout(location = …)` (we will need this later)

Uniform Buffer Objects

Uniform Blocks

- Transmit parameters from OpenGL to shader program: **Uniform Variables** (Constants)
- In case of multiple shader programs the number of uniform parameters grows…
- Many of them are often used in multiple shader programs
  - → optimize the access and common usage of uniform variables

- Grouping of multiple uniform Variables in a Block, for example for transformation and projection matrices in a vertex shader:

```
uniform Matrices {
    mat4 model;
    mat4 view;
    mat4 projection;
};
```

Insert data in Uniform Block

- Create a Buffer Object for the Uniform block using **`glGenBuffers`**

- To set a Uniform Variable corresponds to the insertion of data in a Uniform Buffer Object using

```
glBindBuffer(…)
glBufferStorage(…)
glBufferSubData(…)
```

UBO Generation

- Create Uniform Buffer Object (UBO), bind as current UBO, reserve memory

```
glGenBuffers(1, &myUBO);
glBindBuffer(GL_UNIFORM_BUFFER, myUBO);
glBufferStorage(GL_UNIFORM_BUFFER, sizeInBytes, NULL, 0);
```

- Insert Data in Buffer (= set values in Uniform Variables in Block)

```
glBufferSubData(GL_UNIFORM_BUFFER, offset, sizeInBytes,
                dataPointer);
```

- It is possible to update the entire buffer or a small range using `offset` and `sizeInBytes` (e.g. replacing all matrices or only one)

UBO Connection to Shader

```
glBindBufferBase(GL_UNIFORM_BUFFER,
                 bindingPoint,
                 myUBO);
```

- This connects the UBO **`myUBO`** to the binding point **`bindingPoint`** in the shader
- A binding point in the shader can be defined using

```
layout (binding = …) uniform … { };
```

Note that OpenGL is slightly confusing here in comparison to the VAO connection:
For Uniform Buffers, we directly assign the UBO ID myUBO to a binding point in the shader.
In the VAO case, we have to create a binding index for each VBO (which is not in the shader!) and then connect this to an attribute index in the shader.

Uniform Block Example

- Let us look at the matrix example again
- We use the binding point 0 for the uniform block in the vertex shader

```
layout (binding = 0) uniform Matrices {
    mat4 model;
    mat4 view;
    mat4 projection;
};
```

Uniform Block Example

- Generate a uniform buffer object and reserve memory for three matrices

```
glGenBuffers(1, &myUBO);
glBindBuffer(GL_UNIFORM_BUFFER, myUBO)
glBufferStorage(GL_UNIFORM_BUFFER, 3 * sizeof(glm::mat4),
                NULL, GL_DYNAMIC_STORAGE_BIT);
```

- Fill in the three matrices

```
glm::mat4 modelMatrix, viewMatrix, projMatrix;
…
glBufferSubData(GL_UNIFORM_BUFFER, 0 * sizeof(glm::mat4),
                sizeof(glm::mat4), glm::value_ptr(modelMatrix));
glBufferSubData(GL_UNIFORM_BUFFER, 1 * sizeof(glm::mat4),
                sizeof(glm::mat4), glm::value_ptr(viewMatrix));
glBufferSubData(GL_UNIFORM_BUFFER, 2 * sizeof(glm::mat4),
                sizeof(glm::mat4), glm::value_ptr(projMatrix));
```

Uniform Block Example

- Connect the UBO with the shader at binding point 0

```
glBindBufferBase(GL_UNIFORM_BUFFER, 0, myUBO);
```

- In the shader we have

```
layout (binding = 0) uniform Matrices {
    mat4 model;
    mat4 view;
    mat4 projection;
};
```

- Now we can compute things like this in the shader:

```
vec4 viewPos = view * worldPos;
```

Note that the name „Matrices“ is never used,
but the GLSL compiler needs a name at this point…

Uniform Block Layout

- Memory layout

Layout qualifier	Description
`shared`	Used in multiple programs
`packed`	Optimize memory
`std140`	Standard layout
`row_major`	Matrix row by row
`column_major`	Matrix column by column

- `e.g. layout(shared, row_major) uniform Matrices { … };`

That´s all for today

- Next Week: Textures