ARLAB

ME/CprE/ComS 557

Computer Graphics and Geometric Modeling

Geometric Modeling

September 6, 2015

Rafael Radkowski

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

You should know by now

ARLAB

- Create and set up a Visual Studio project (manual and with cmake)
- Compile shader code.

Topics

- Types of Models
- Representation of a 2D/3D Model
- Rendering objects with OpenGL primitives

Next:

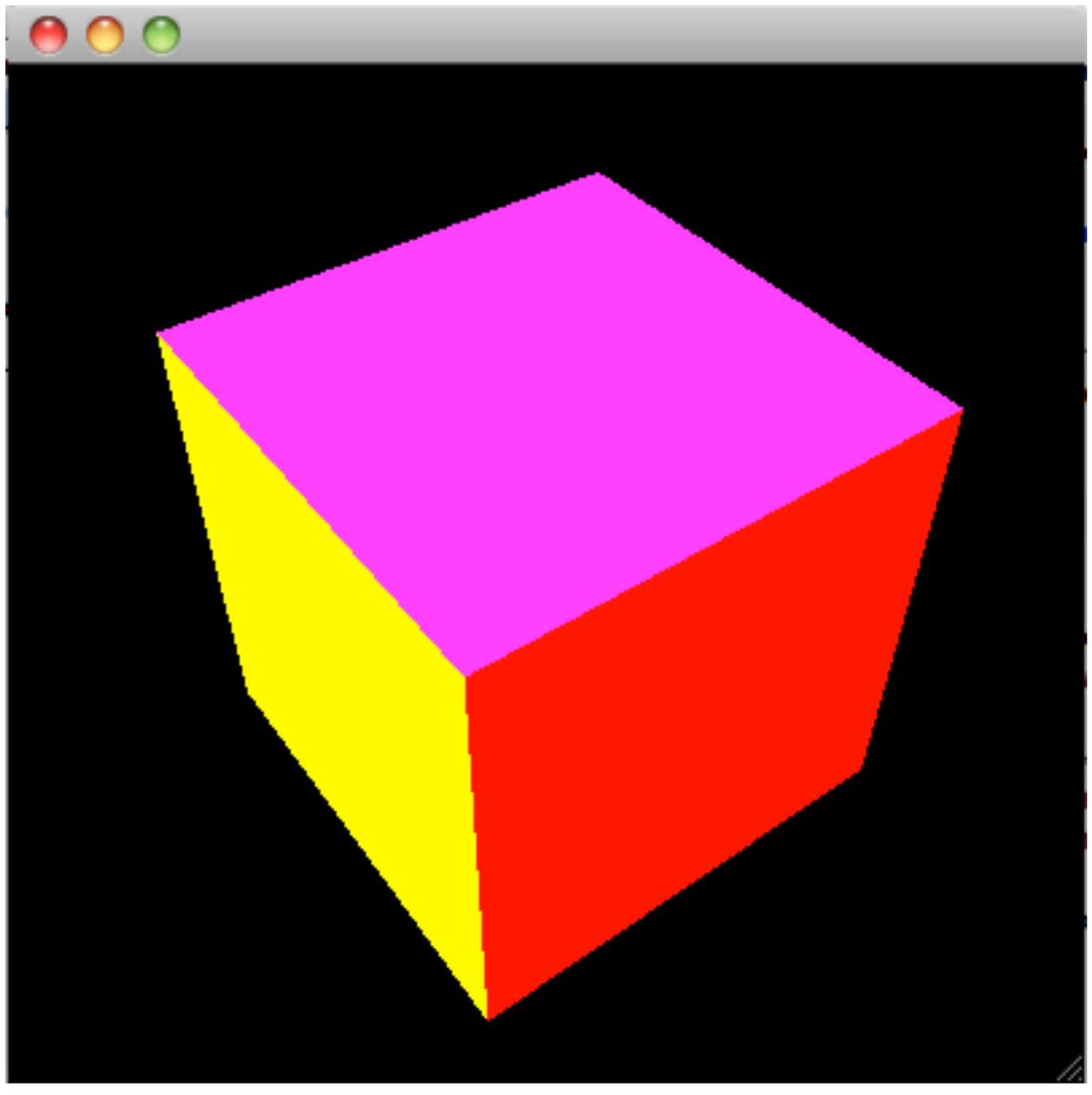
- Transformations in 2D/3D
- Hierarchical representation of models
- Approximation and Filtering of Parametric Models

Superbible Chapter 3 & 7

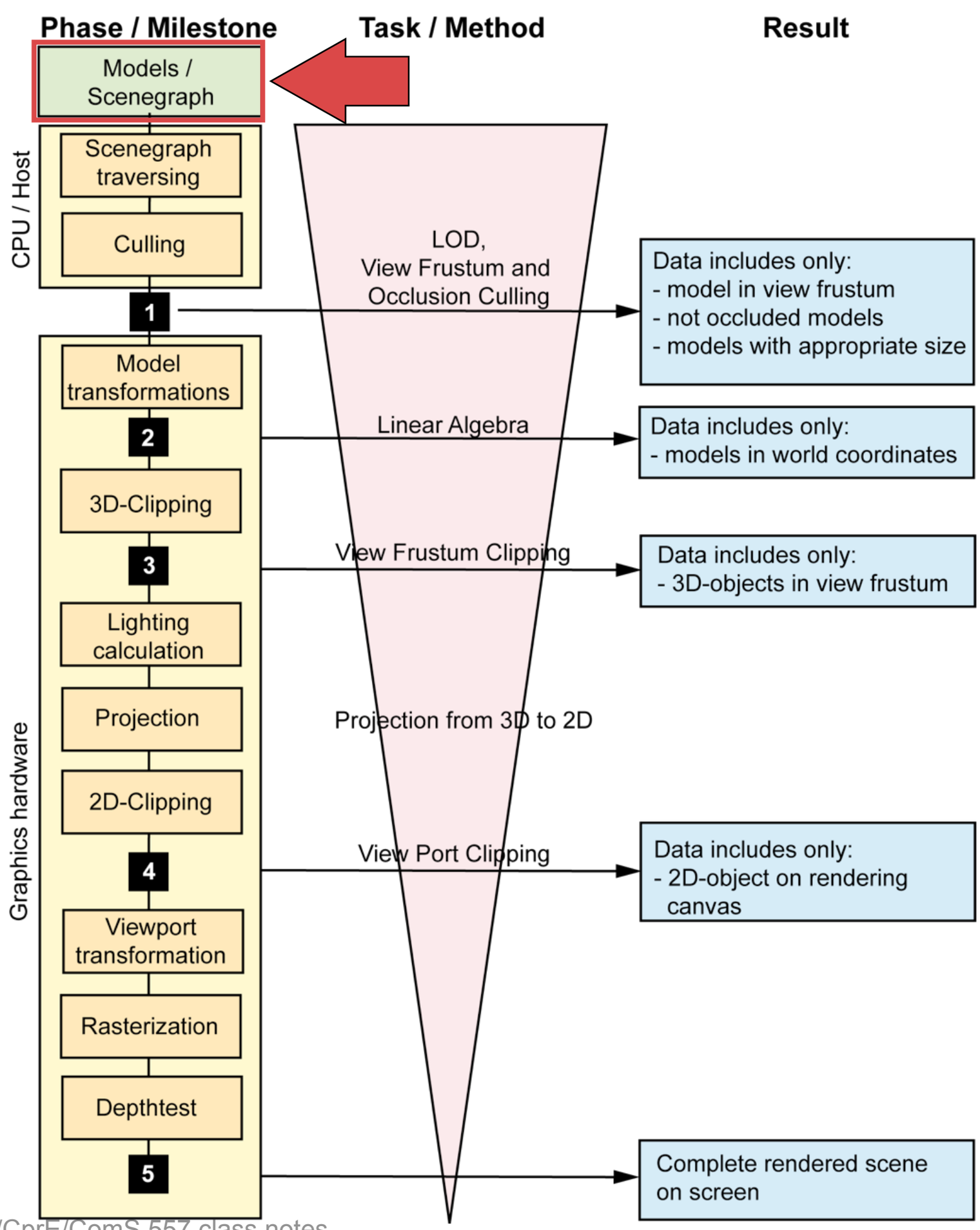
Computer Graphics 8

3D Model Example

ARLAB



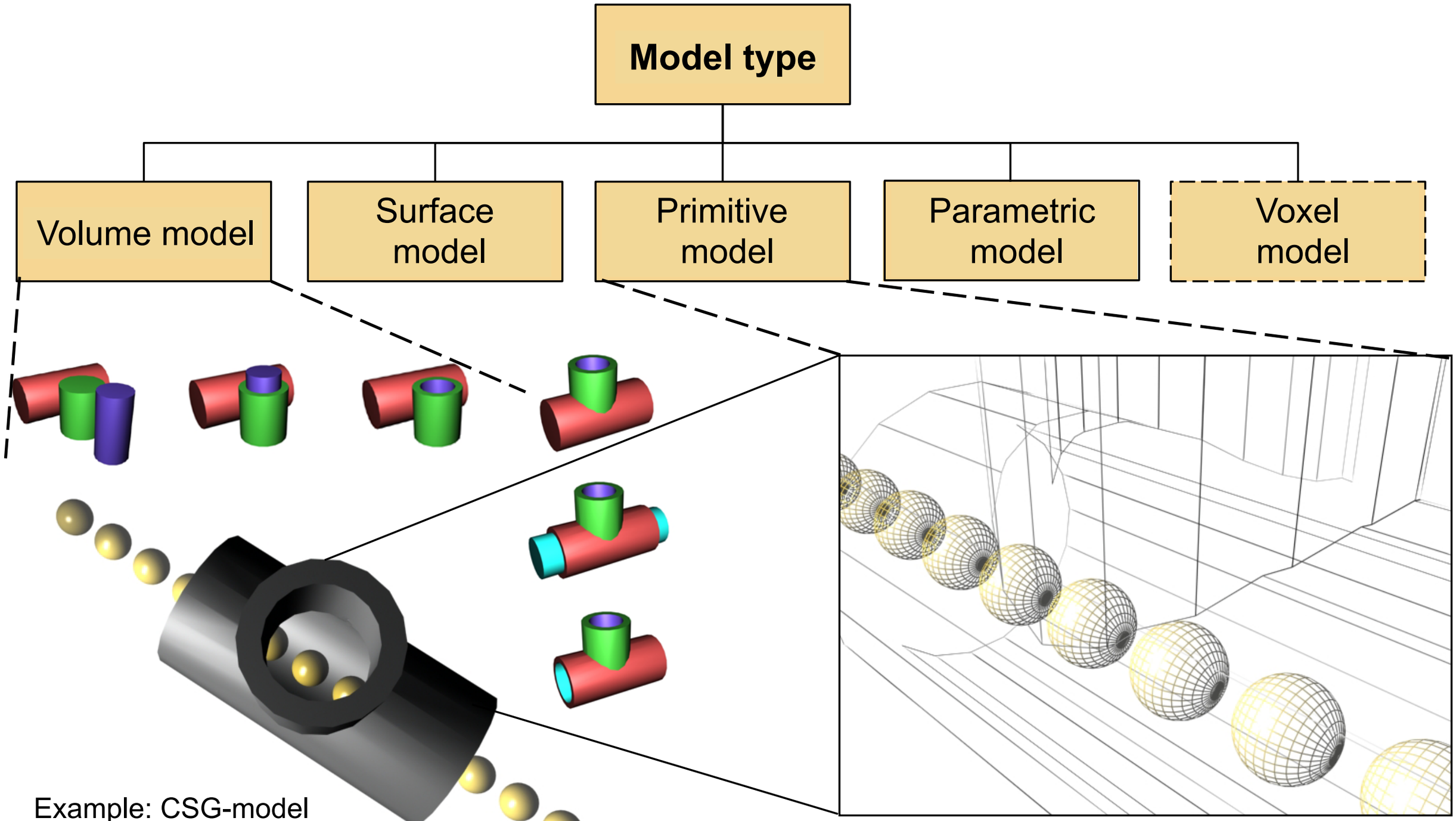
A 3D model



Rendering Pipeline

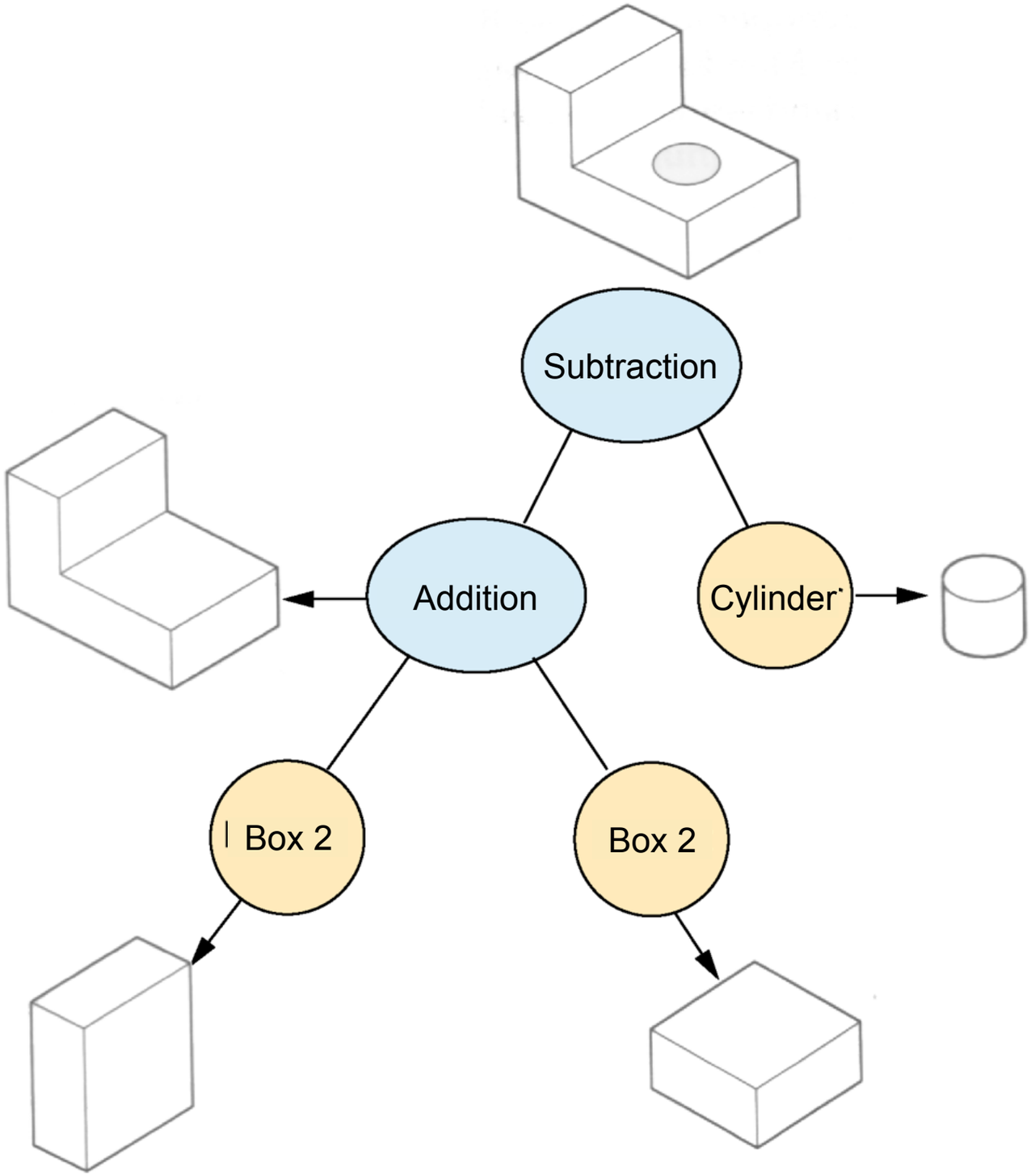
Types of Models

ARLAB

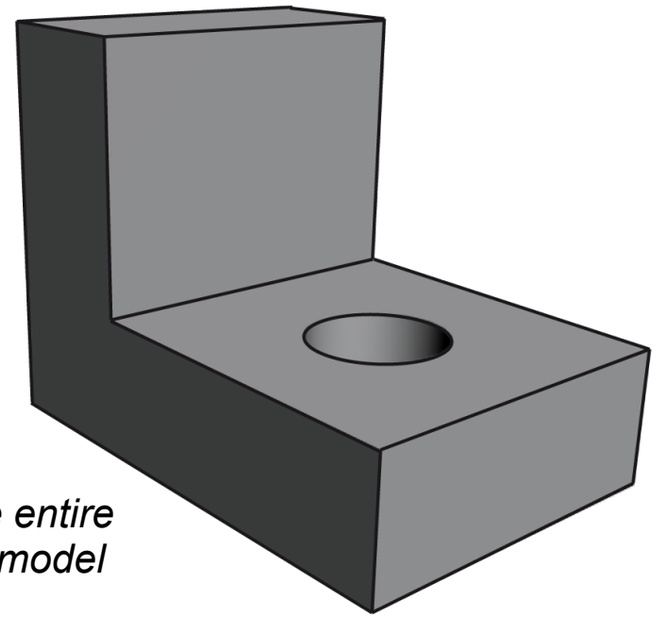


CSG Volume Model (Constructive Solid Geometry)

ARLAB



CSG-model represents 3D models as aggregation of multiple primitive models.



The entire 3D model

CSG-models are the result of an aggregation (boolean composition) of basic volumes (primitives) that represent the 3D object. These primitives are provided by the graphics sub-system of the software program. The logical association between these primitives is represented as tree data model.

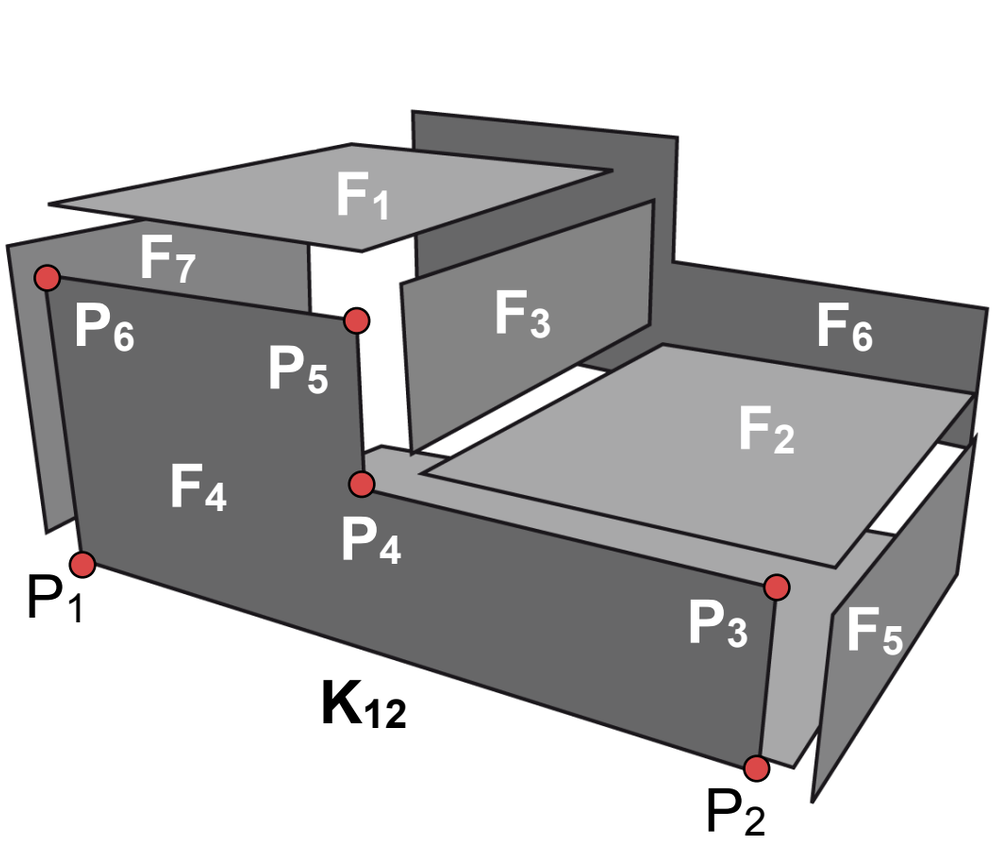
graphic primitives

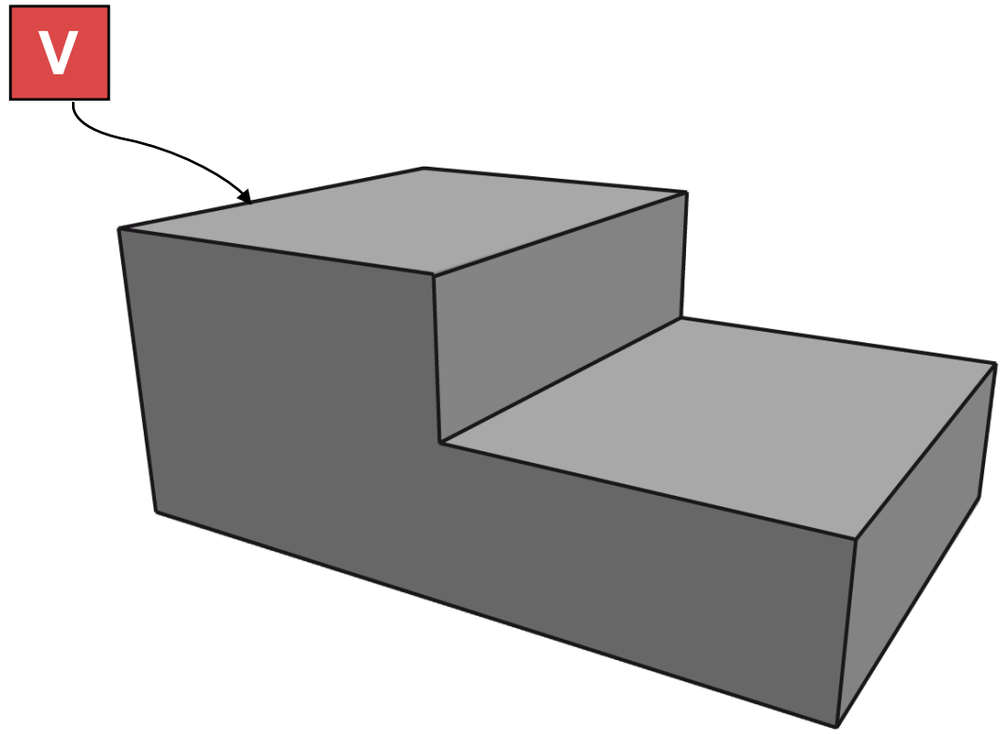
- cube
- cylinder
- sphere
- pyramid
- ring

operations

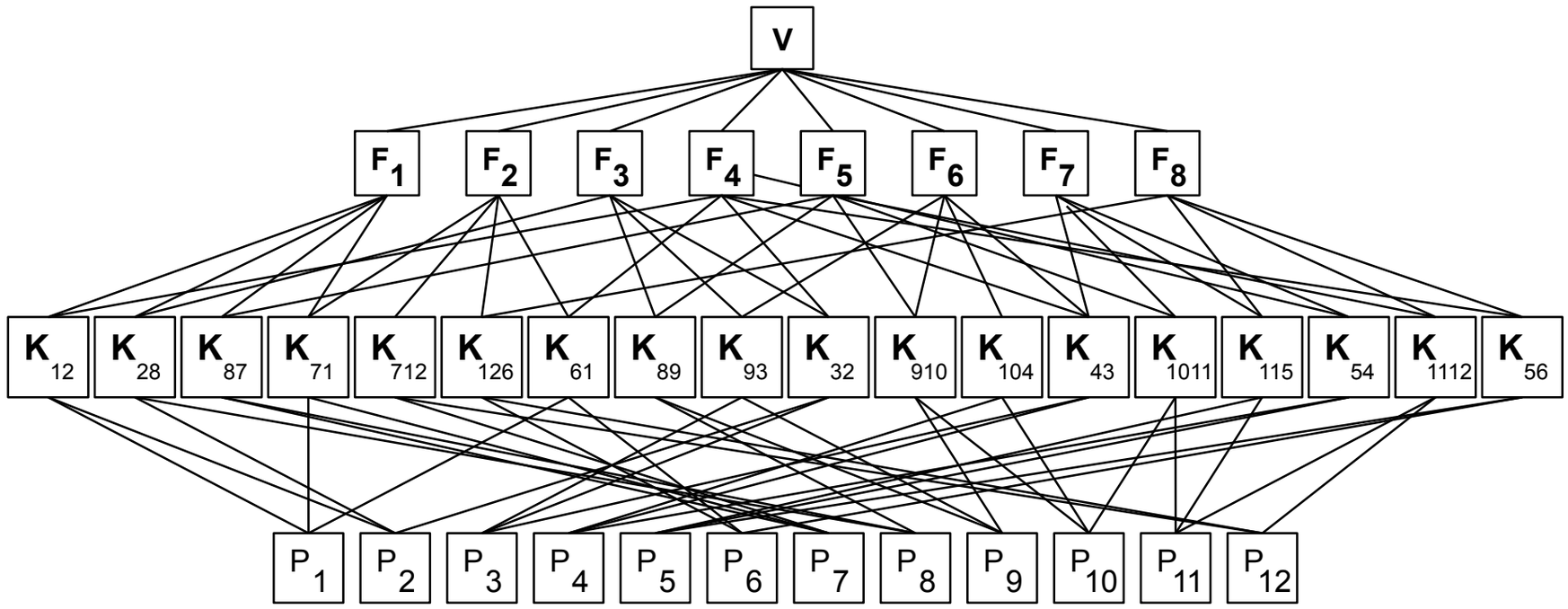
- subtraction
- addition
- crop
- slice

Boundary Representation Model (BRep)





The BRep-model represents a volume with the volume limiting boarders. These boarders are represented by edges, which are described by points. The entire model of a volume is kept in a tree structure data model.



Essentials

ARLAB

Hughes, van Dam et al.: *"If you understand the mathematical process well enough, you can write a program that executes it."*

Please make sure that you know all the material

Notation:

$\mathbf{v}$	vector: bold, lowercase
$\mathbf{M}$	matrix: bold, uppercase
$\mathbf{R},\mathbf{C}$	predefined as real and complex numbers
$\mathbf{v}_i$	index: subscript, italic
A	set: uppercase, italic
	set of vectors $A = \{\mathbf{a}_0, \mathbf{a}_1, ..., \mathbf{a}_N\}$

Function

$$f : \mathbf{R} \rightarrow \mathbf{R} : x \rightarrow x^2$$

function name — domain — co-domain — rule

Further reading: Hughes, van Dam, Chapter 7: Essential Mathematics

Essentials

ARLAB

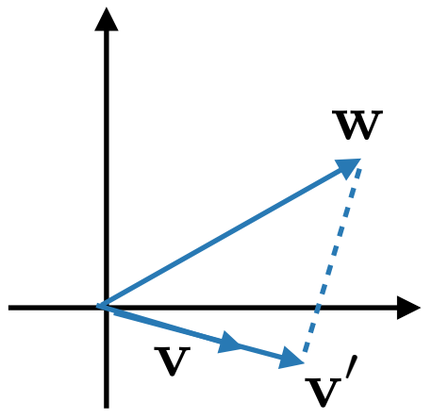
Hughes, van Dam et al.: *"If you understand the mathematical process well enough, you can write a program that executes it."*

Please make sure that you know all the material

- Coordinates systems (Cartesian coordinates / Euclidian Geometry)
- Vectors (displacement, difference between points, Section 7.6.2)
- Vector normalization, vector operations
- Cross product, dot product
- Vector projection

$$\mathbf{v}' = \frac{\mathbf{v} \cdot \mathbf{w}}{\mathbf{v} \cdot \mathbf{v}} \mathbf{v}$$

- Matrices
- Matrix transposition
- Matrix multiplication
- (Anti-symmetric matrix or skew matrix)
- (Matrix rank)

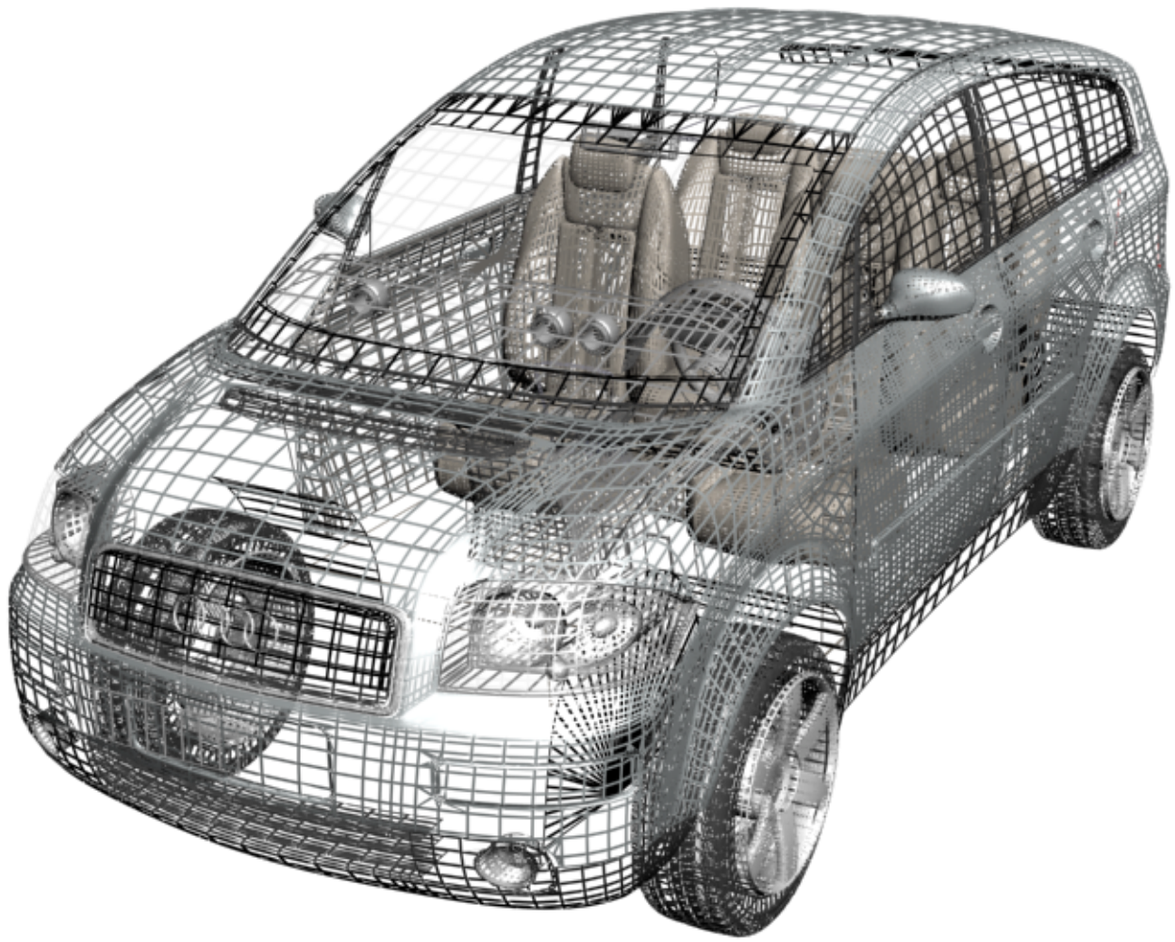


Further reading: Hughes, van Dam, Chapter 7: Essential Mathematics

3D models in OpenGL

ARLAB

Surface rendering of a vehicle



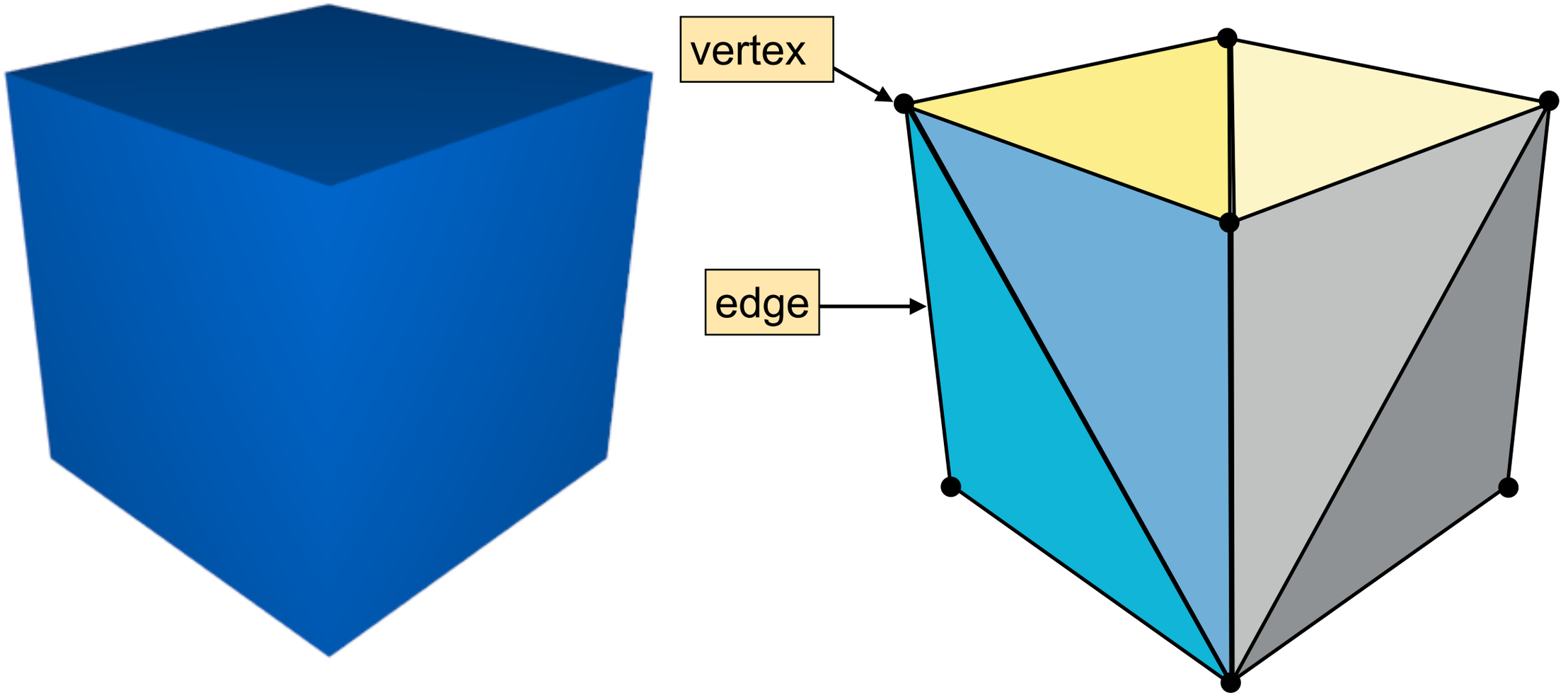
Wireframe rendering of a vehicle

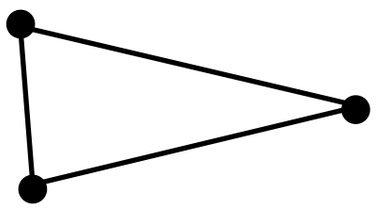
Further reading: Hughes, van Dam, Chapter 8

Mesh Approximation

ARLAB

We have to approximate every surface with triangles by identifying



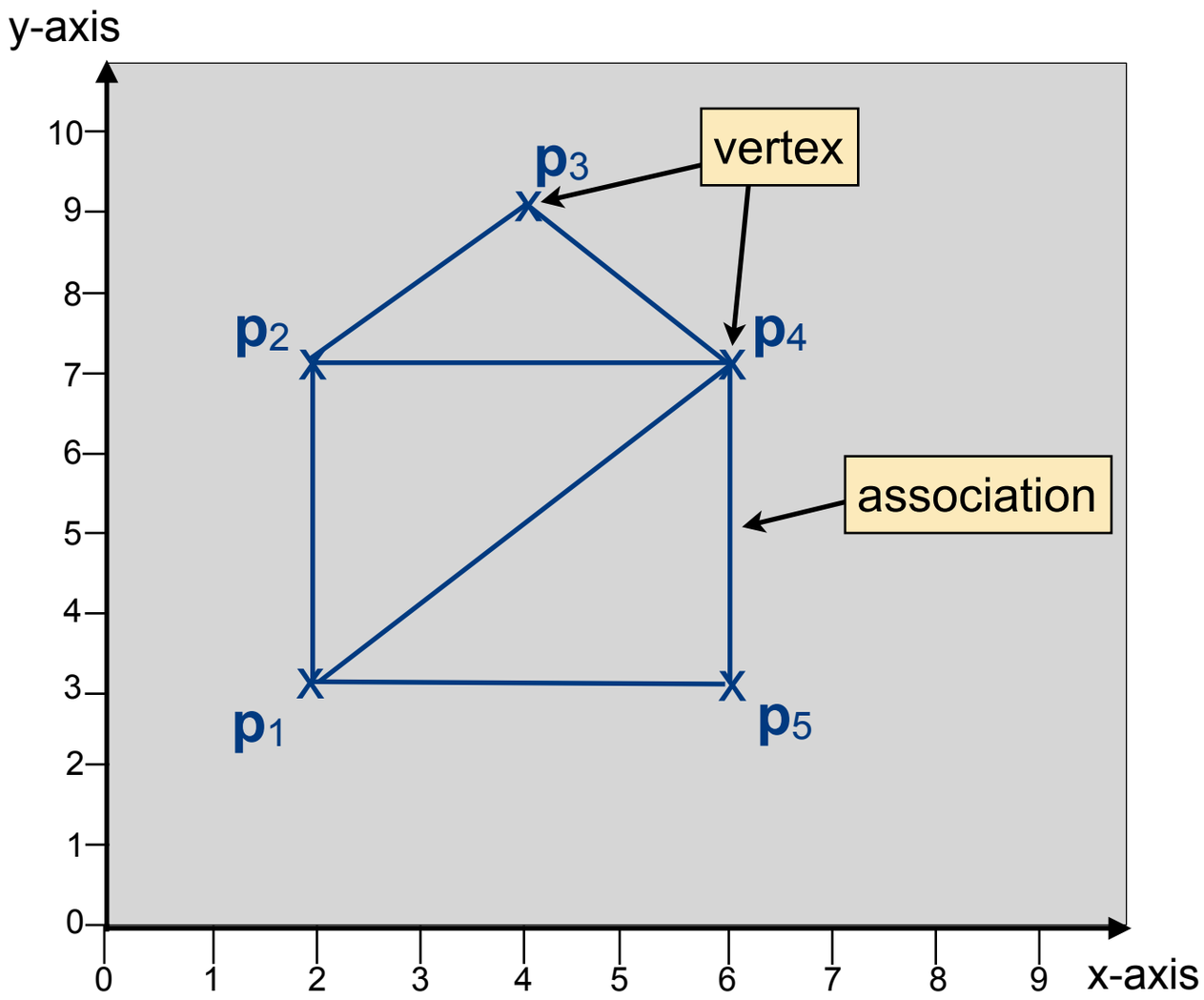


The triangle is one of the simplest object we have to approximate a surface.

Advantages: piecewise constant, simple, etc.

Model represented by primitives

All models are represented as a set of vertices (vector 2D or 3D) and the point relations. This results in to lists:



vertex list

vertex	x	y
p_1	2	3
p_2	7	2
p_3	4	9
p_4	6	7
p_5	6	3

edges

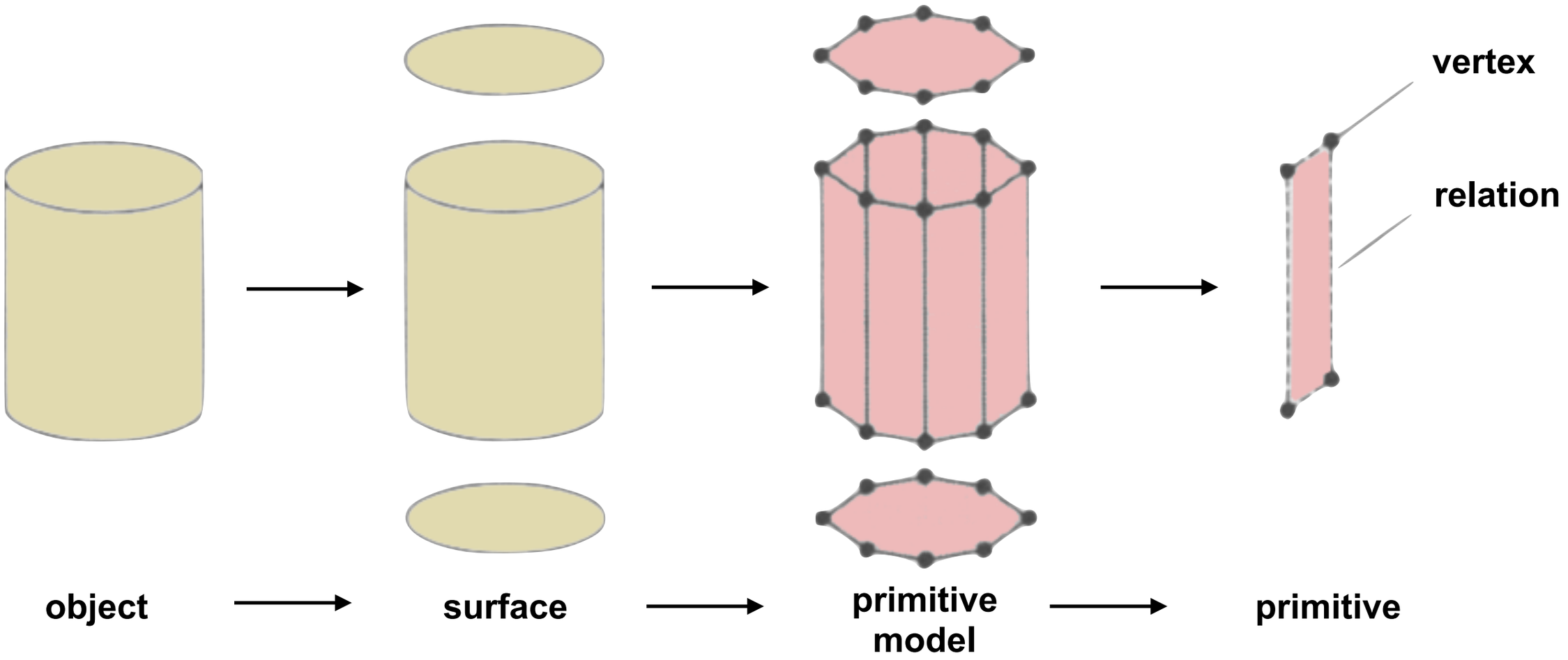
start	end
p_1	p_2
p_2	p_3
p_3	p_4
p_4	p_5
p_5	p_1
p_2	p_4
p_1	p_4

In computer graphics, every model is represented by two lists.

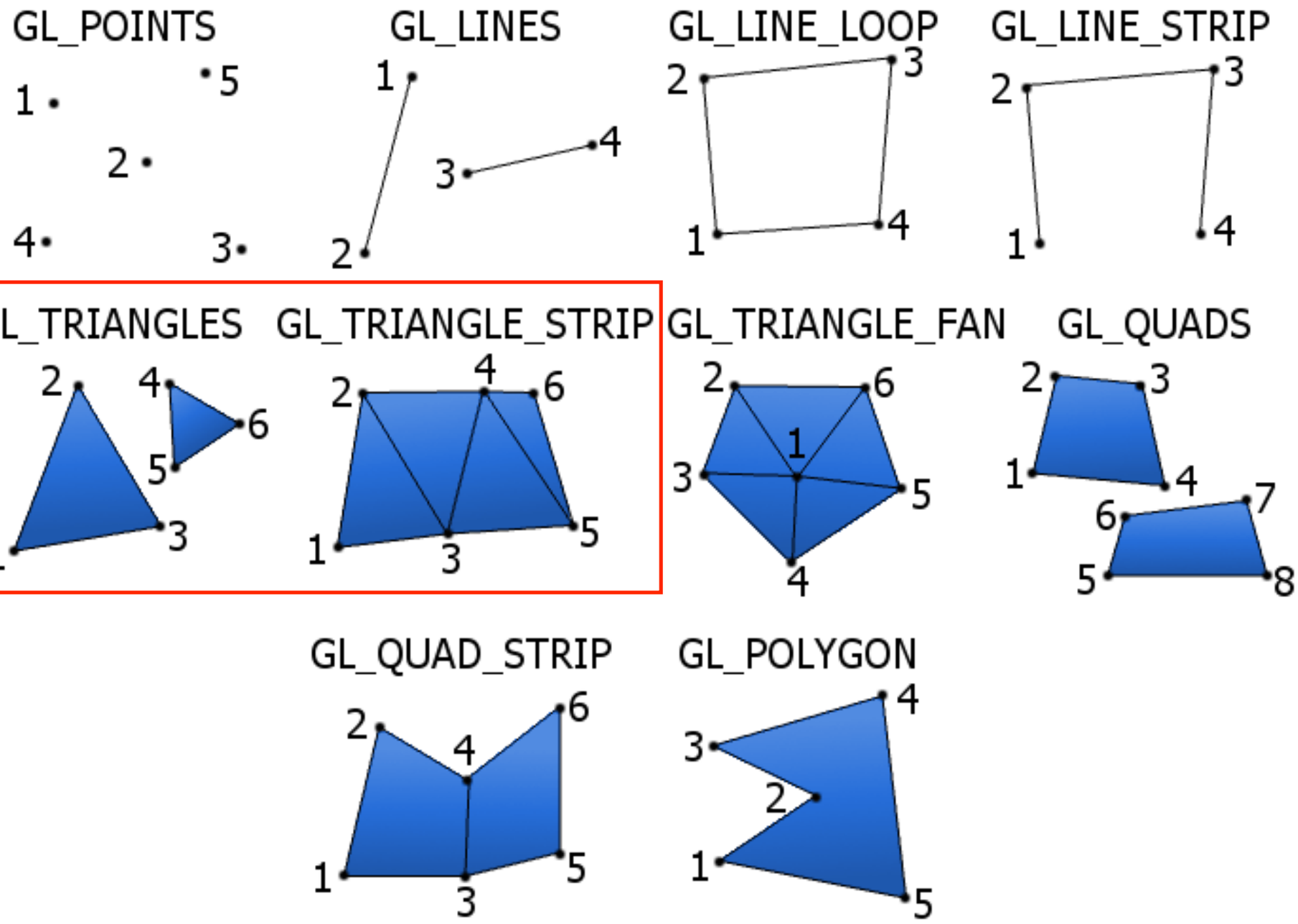
- One list keeps the vertices,
- a second list the associations.

Topology of models

The surface of every 3D model is
approximated by multiple small surfaces



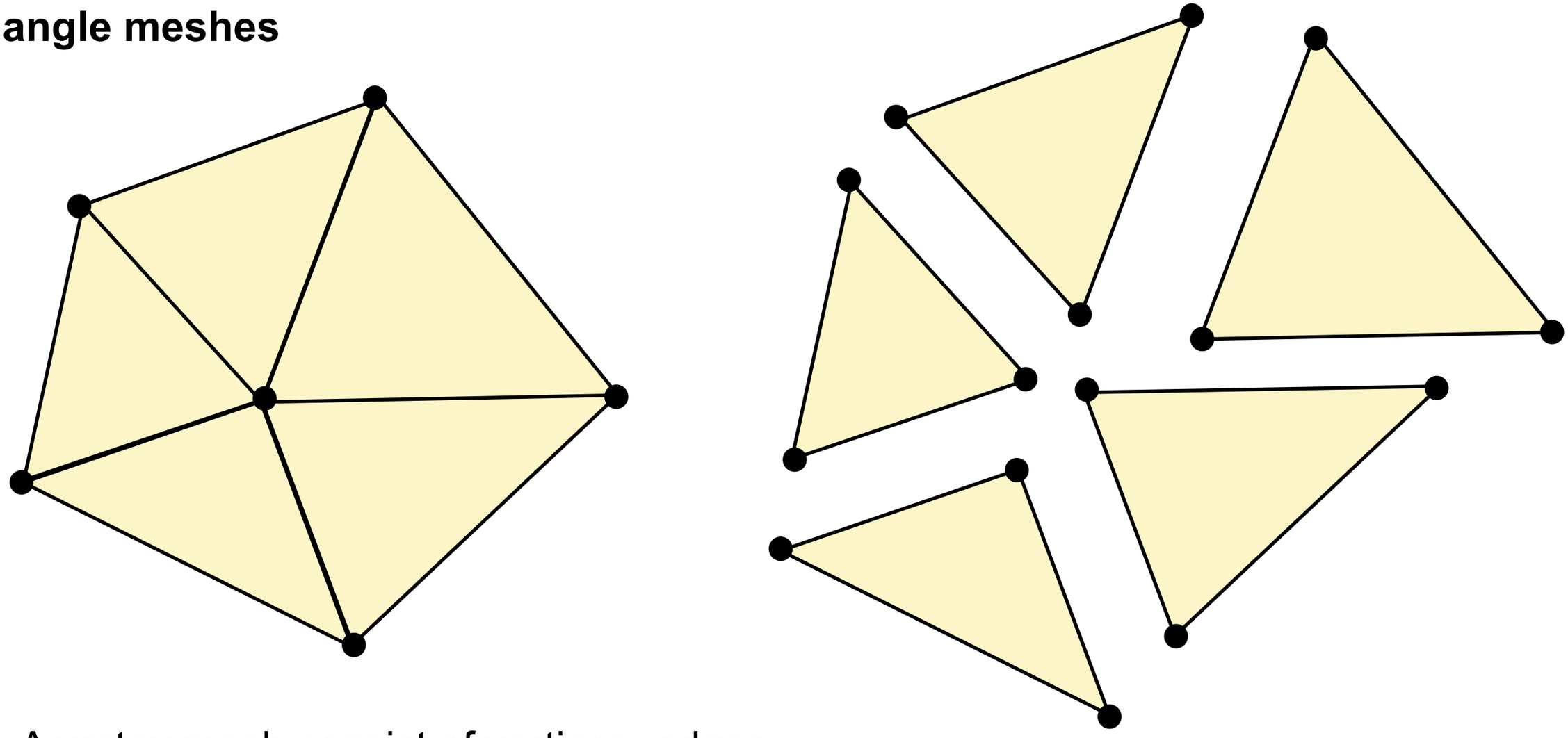
OpenGL Primitives



Shapes in 3D

ARLAB

Triangle meshes

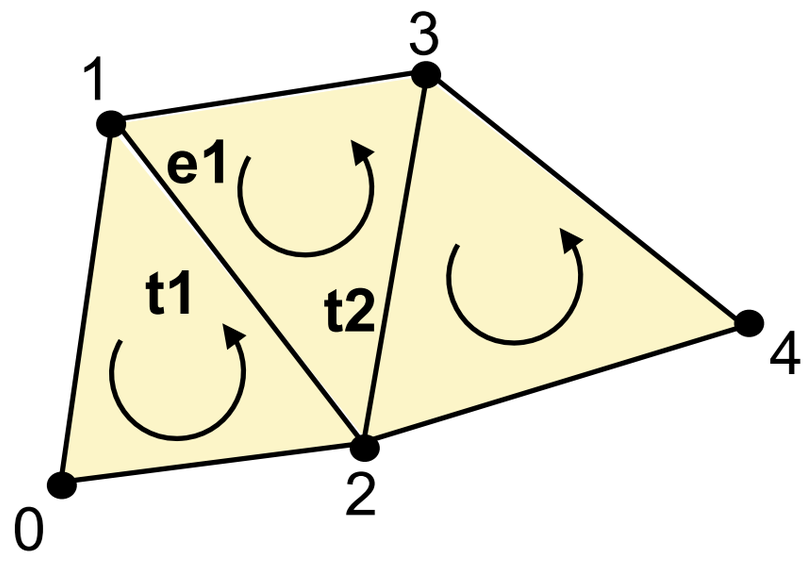


- A vertex mesh consist of vertices, edges, which result in triangle faces
- Almost every surfaces can be split into triangles.
- To create a mesh, we list the vertices and the triangles of the mesh. BUT modern graphics cards infer the edges.

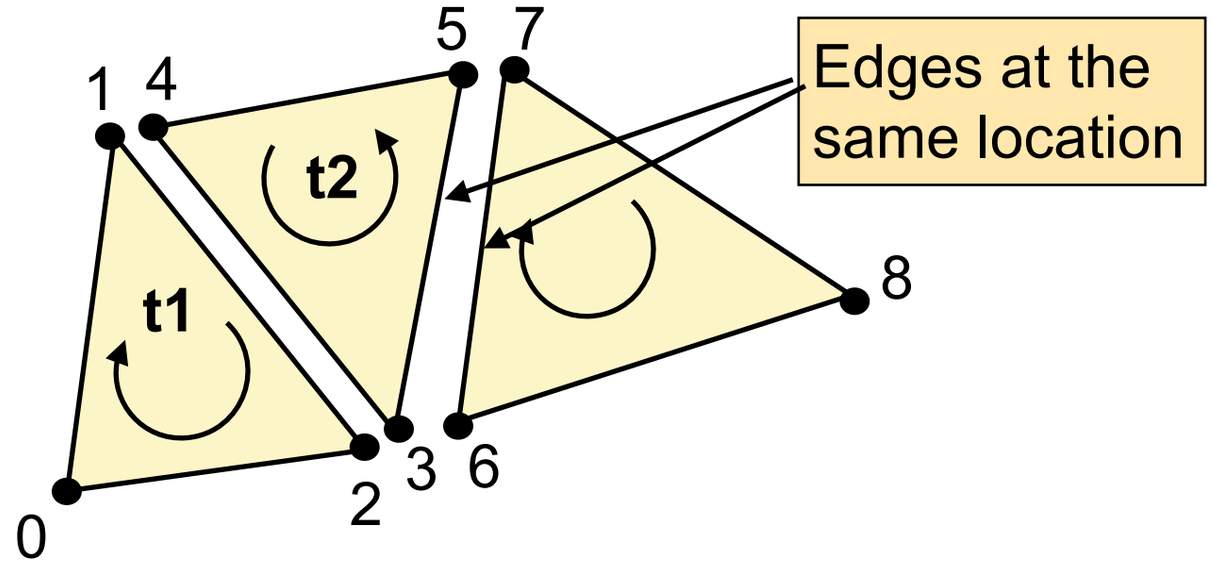
Manifold Meshes

ARLAB

Manifold Meshes can be arranged in a cyclic order without repetition such as edge **e1** is part of triangle **t1** and **t2**. A cyclic order is required.



Manifold mesh (triangle strip)



Non-Manifold mesh

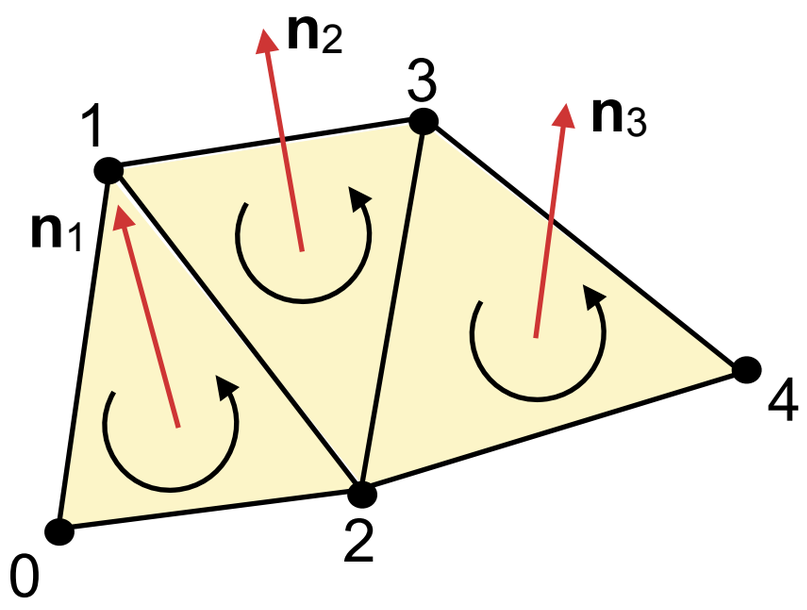
Manifold meshes:

- **are oriented.**
- faster to compute (if your implementation works well)
- do not cause graphics artifacts
- behavior meets our understanding / expectation.
- they do allow to easily add or remove vertices and edges

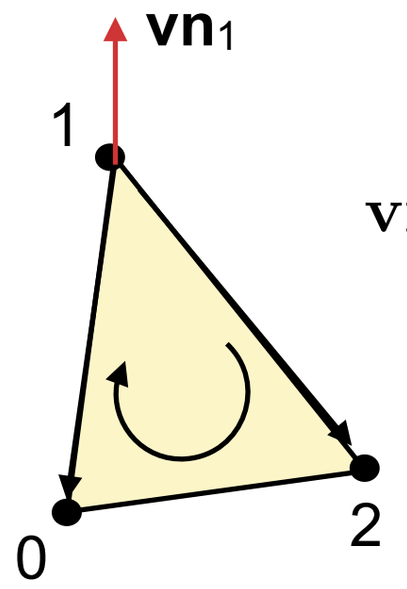
Orientation

The orientation of each triangle is defined by its **normal vector n**.

The surface normal vectors are typically derived using vertex normal vectors $\mathbf{vn}_i$



Surface normal vectors



Non-Manifold mesh

$$\mathbf{vn}_i = (\mathbf{p}_0 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_1)$$

$$norm(\mathbf{vn}_i) = \frac{\mathbf{vn}_i}{||\mathbf{vn}_i||}$$

Vertex normal vectors

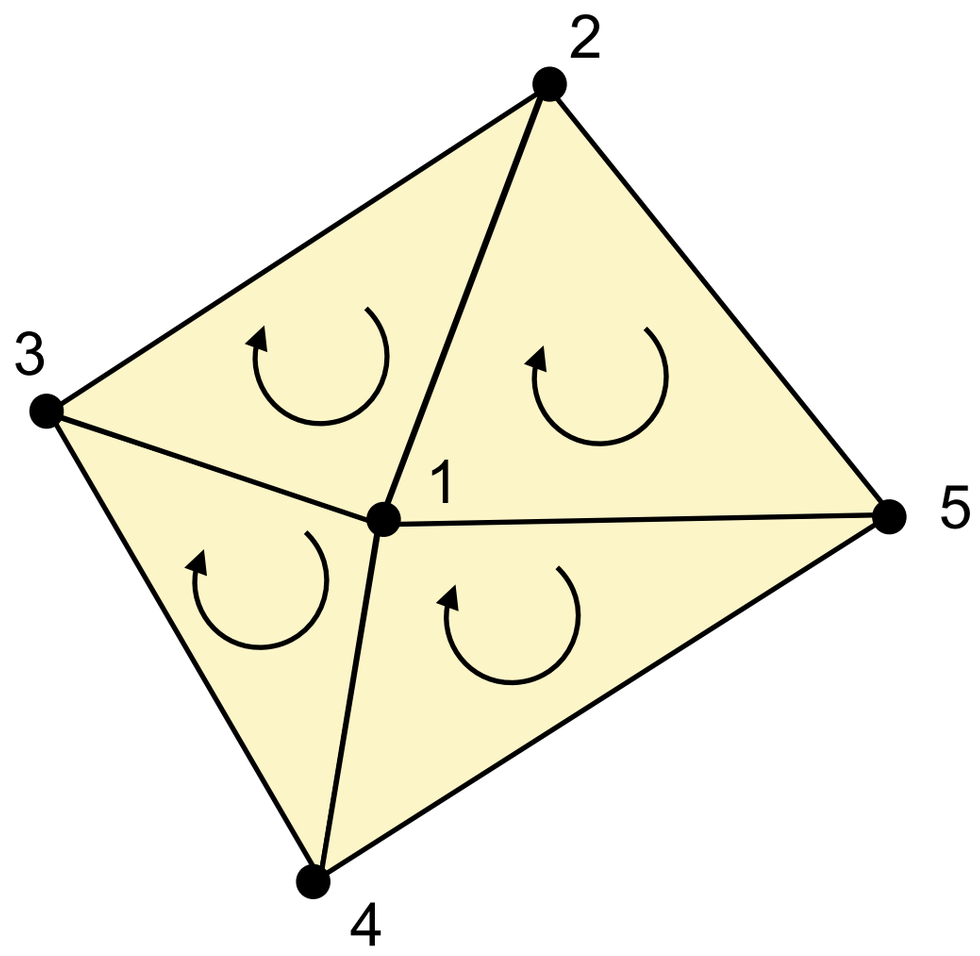
Further, we can

- select one normal vector or interpolate
- Normal vectors are **piecewise constant,** they do not change when the mesh changes

Orientation

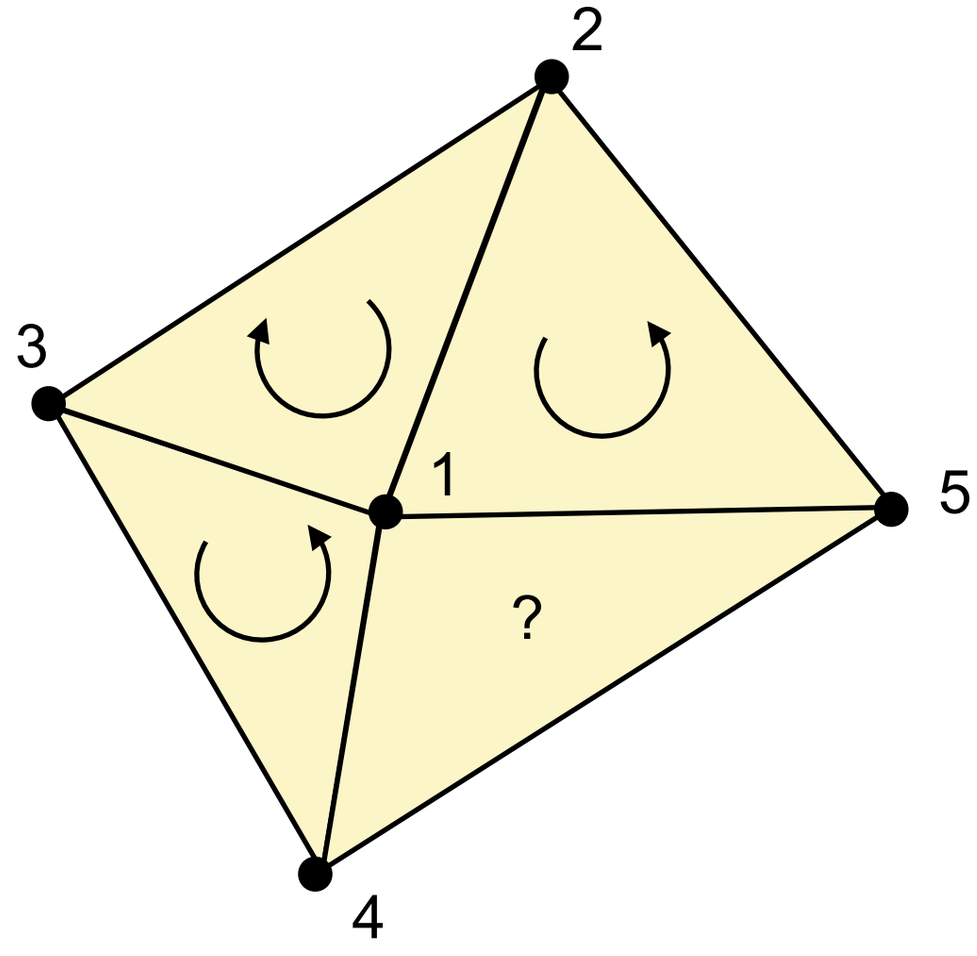
ARLAB

Non-manifold representation



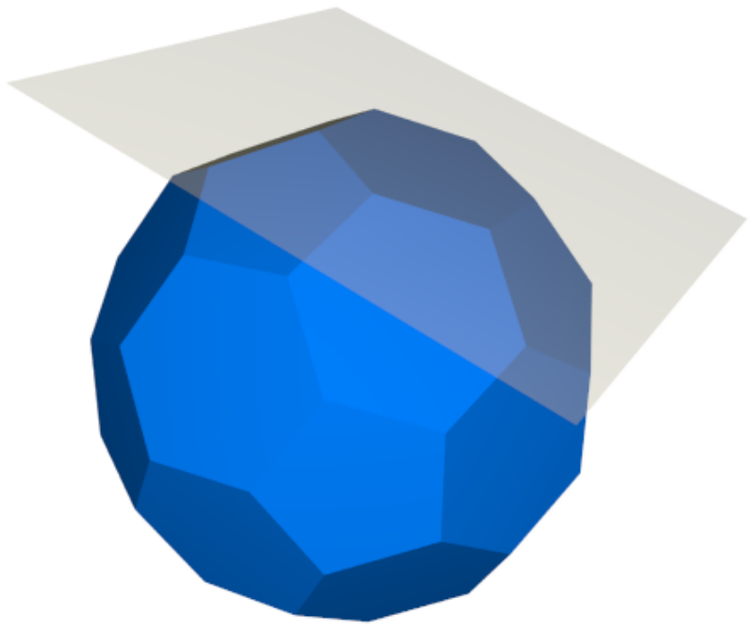
(5, 1, 2), (4, 3, 1), (1, 5, 4), (1, 3, 2)

Manifold representation



(5, 2, 1), (2, 1, 3), (3, 1, 4), (?, ?, ?)

Primitive shape

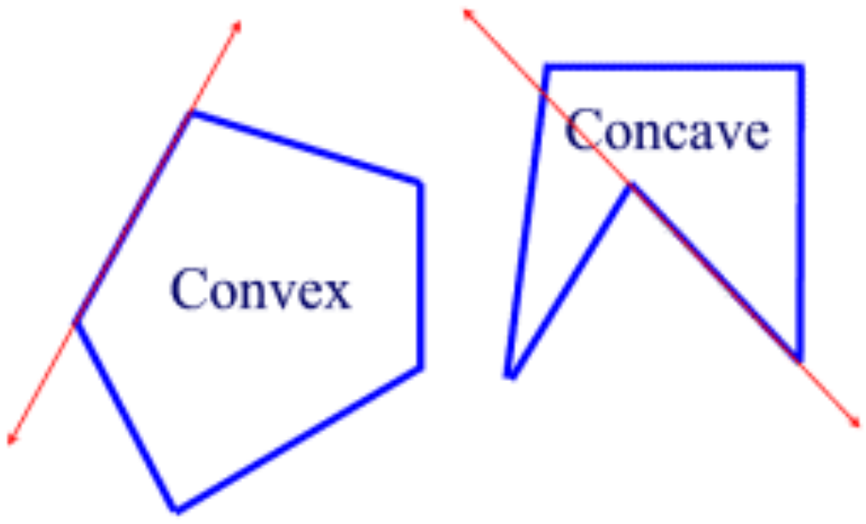


Primitive must fit into a plane

Each primitive must **fit into a plane.**

This is simple for triangles.

However, it is easy to design a quad that does not lie inside a plane. This causes rendering artifacts.



Create convex primitives

A primitive (polygon in this case) is **convex** if no line that contains a side of the polygon contains a point in the interior of the polygon. In a convex polygon, each interior angle measures less than 180 degrees.

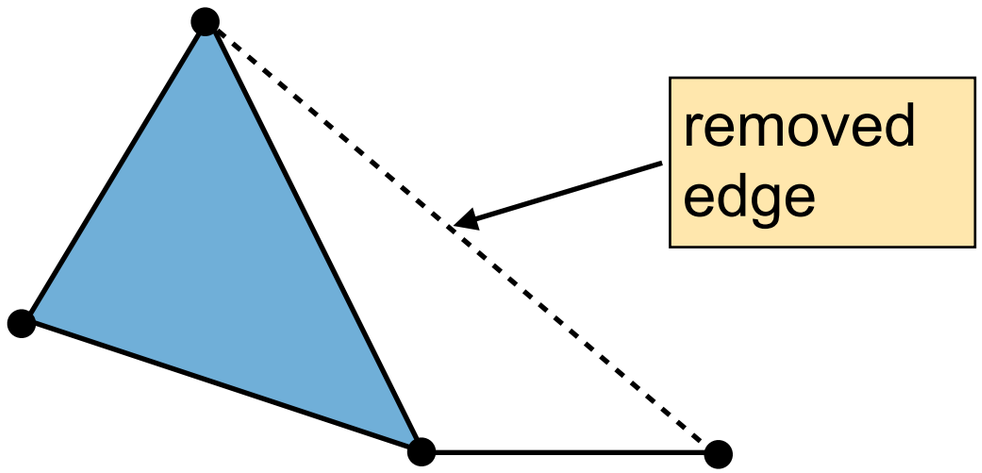
Concave primitives intersect with the interior, creating at least one interior angle greater than 180 degrees (a reflex angle).

Work with convex objects

Primitive shape

ARLAB

Pay attention



Never leave a primitive with a dangling edge!

Advantage:

- models are prepared for fast processing. Our graphics hardware has been developed for this type of models.
- Addition of new vertices is fast

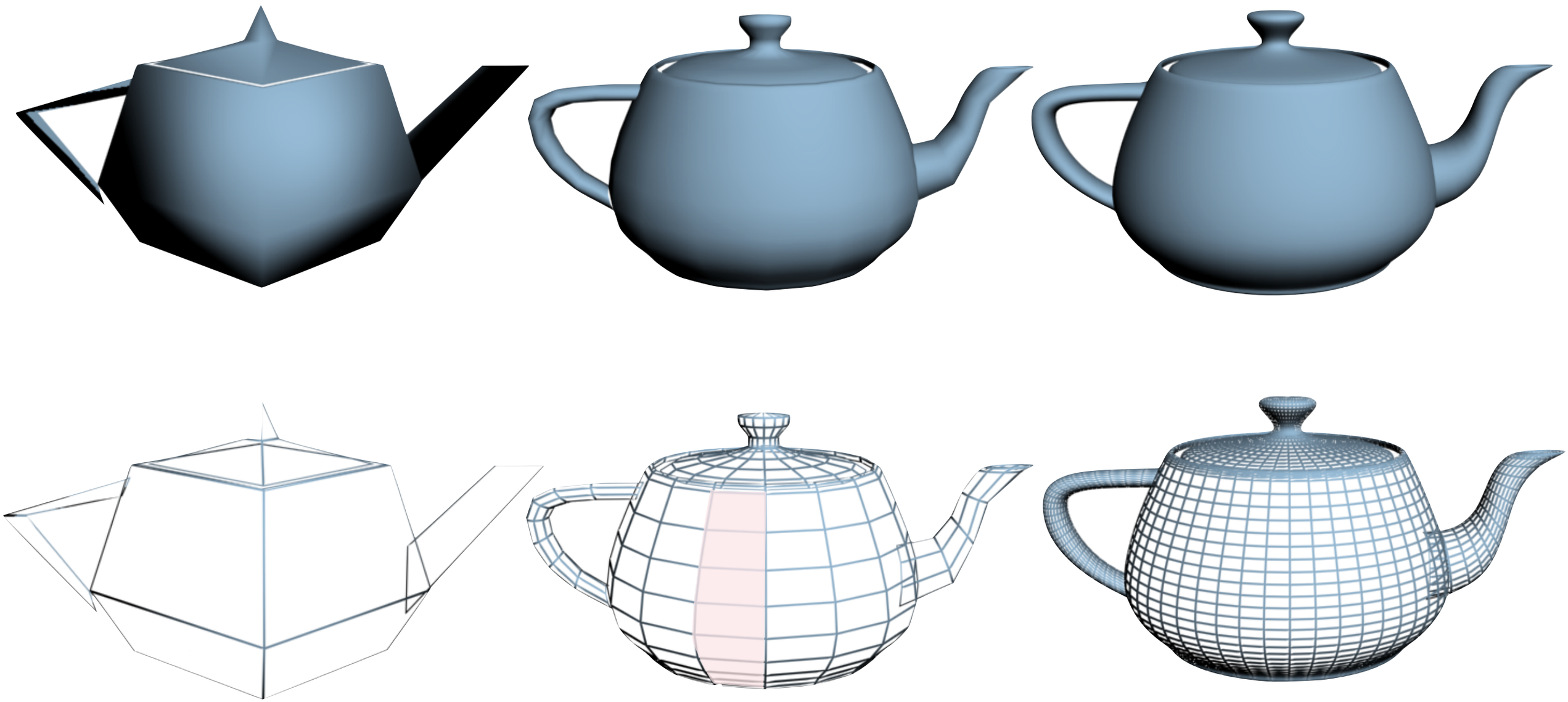
Disadvantage:

- the structure of the entire 3D model is not part of the data structure. Is the triangle A neighbor of triangle B?
- deletion of vertices and triangles is slow

Number of Primitives

ARLAB

Mesh models only **approximate** the surface. The more primitives, the better the approximation but also more processing power is required



34 primitives

160 primitives

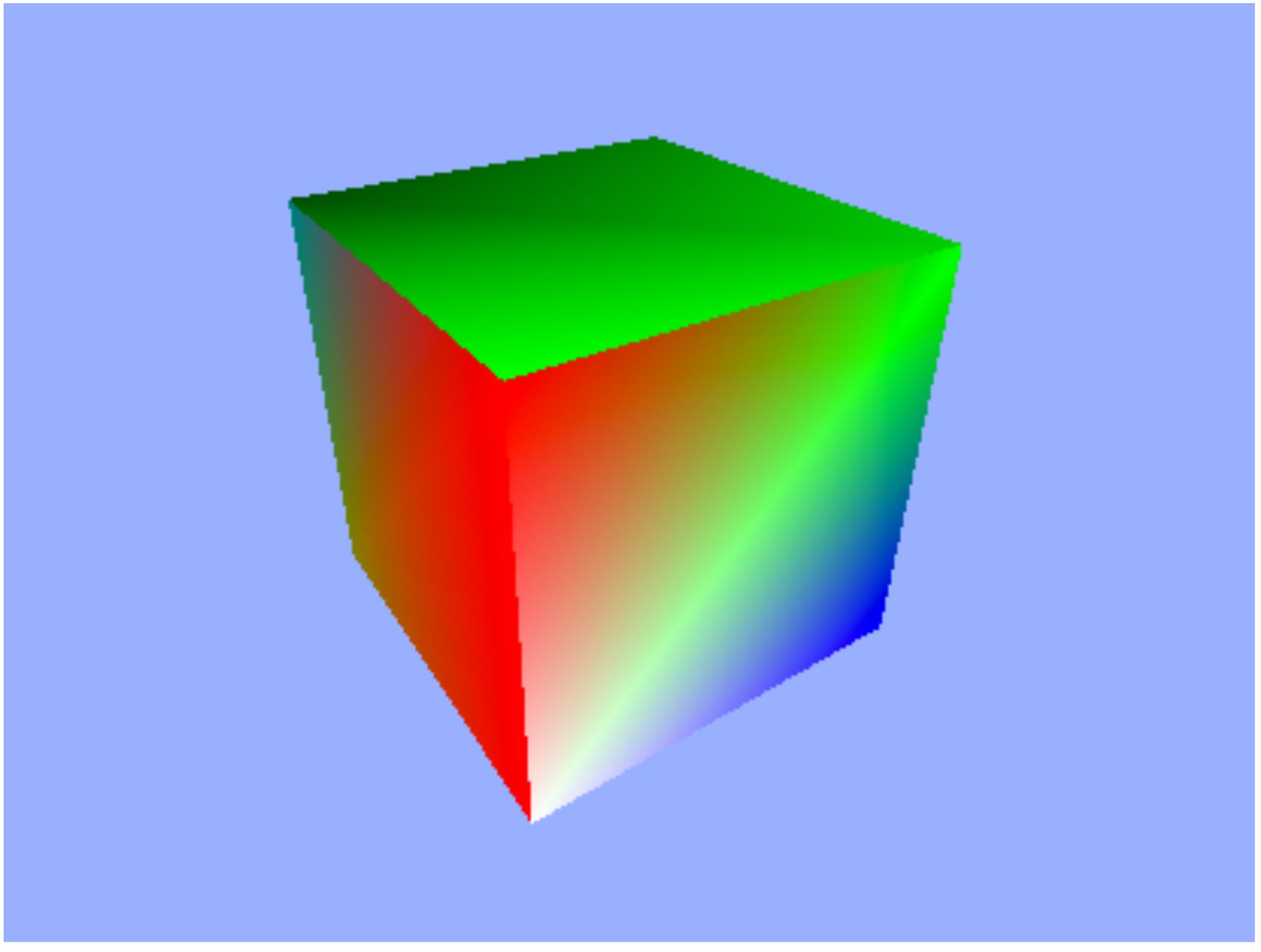
1048 primitives

ARLAB

Model representation in OpenGL

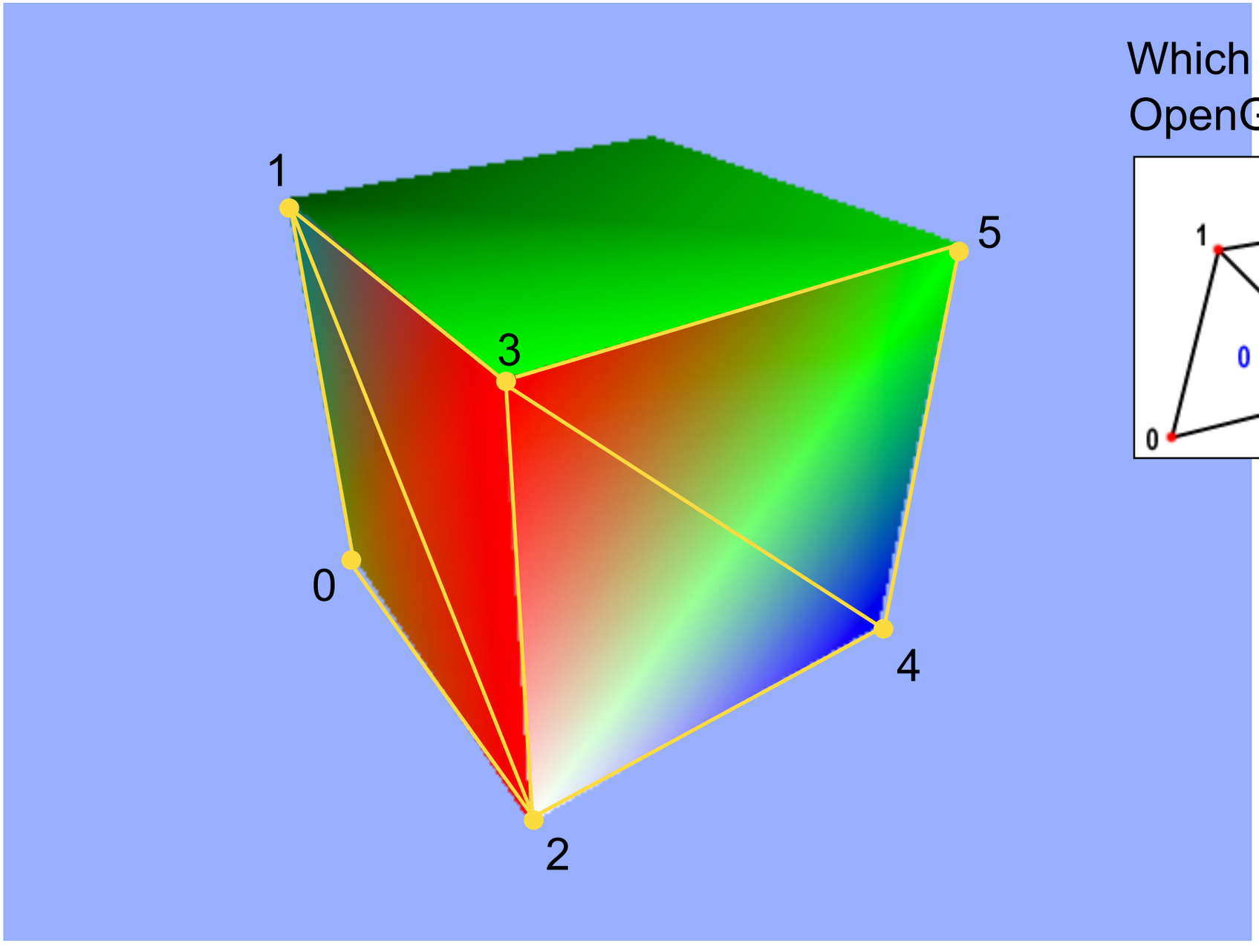
Simple 3D Cube

ARLAB

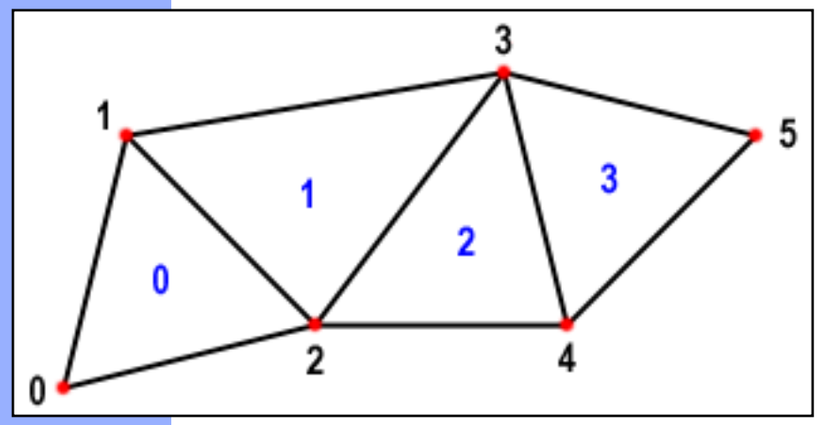


Simple 3D Cube

ARLAB



Which primitive should we use OpenGL Triangle Strip



We need to specify:

- vertices
- edges
- color per vertex

Program Structure

ARLAB

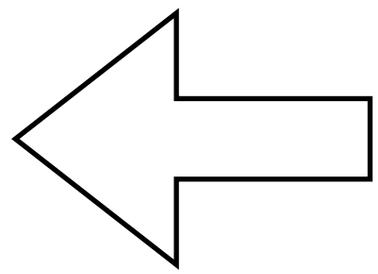
Header

```
int main(int argc, const char * argv[])
{
```

Program init

[....]

Model Definition / Declaration



we are here

[....]

```
while
```

main loop

Clear the window

[....]

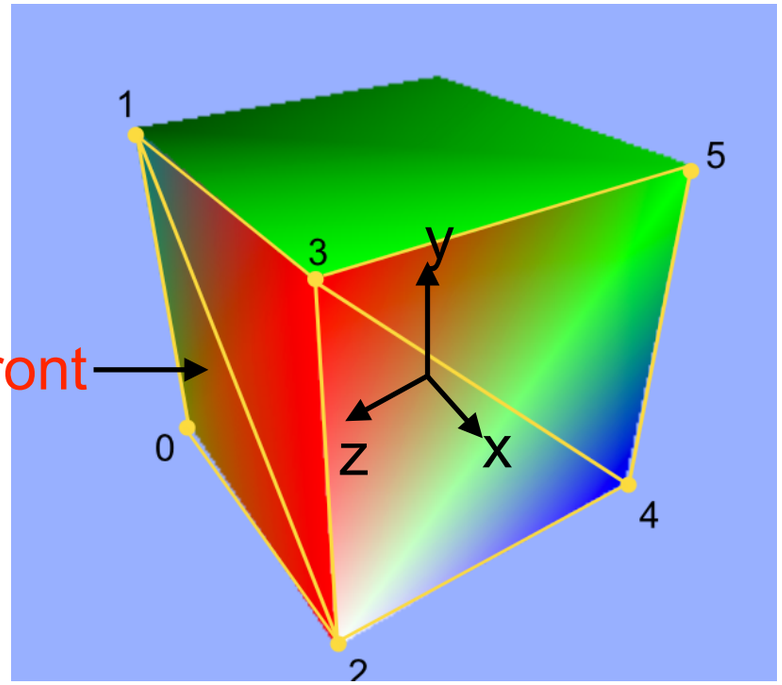
Render the Content

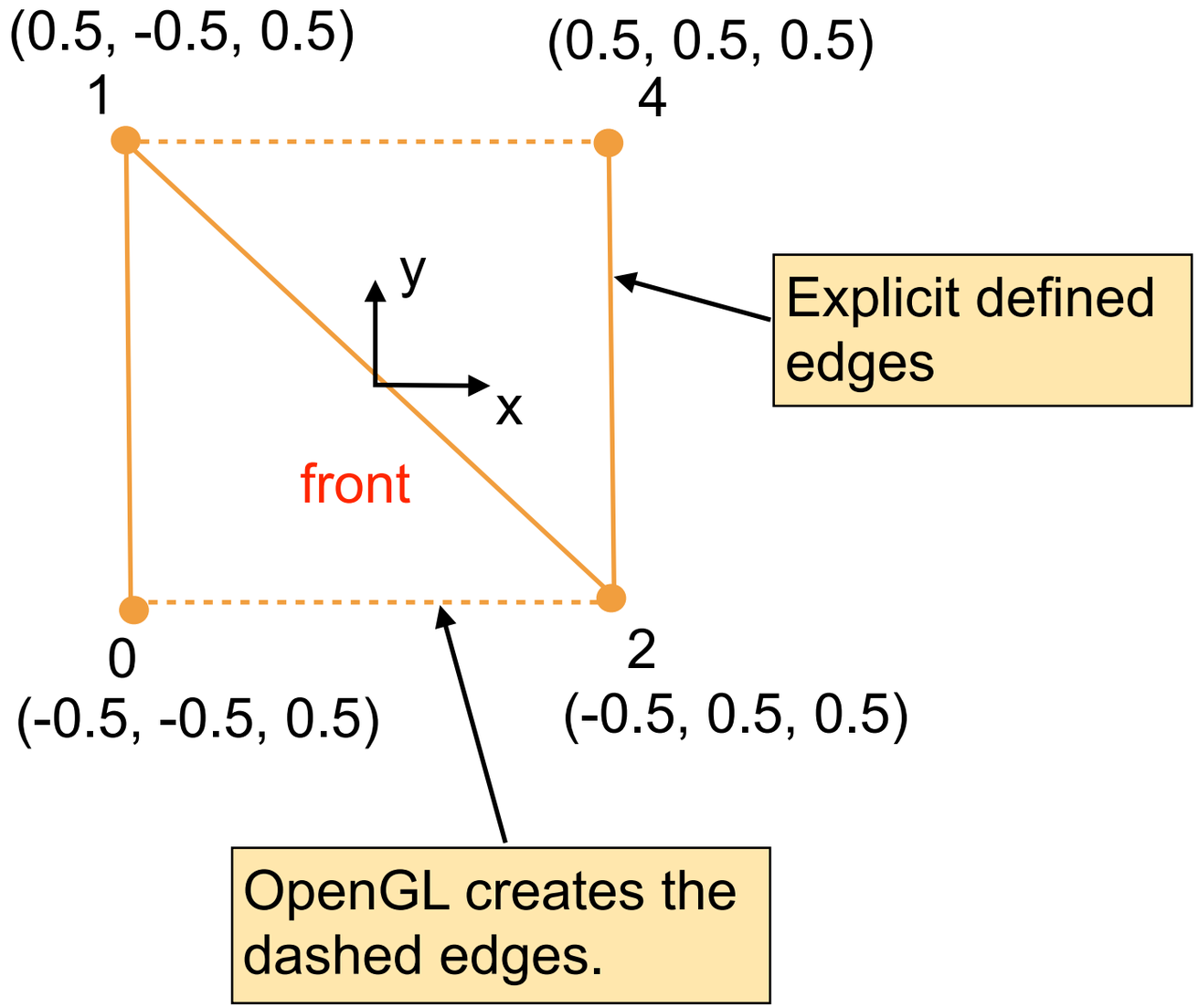
Swap buffers

[....]

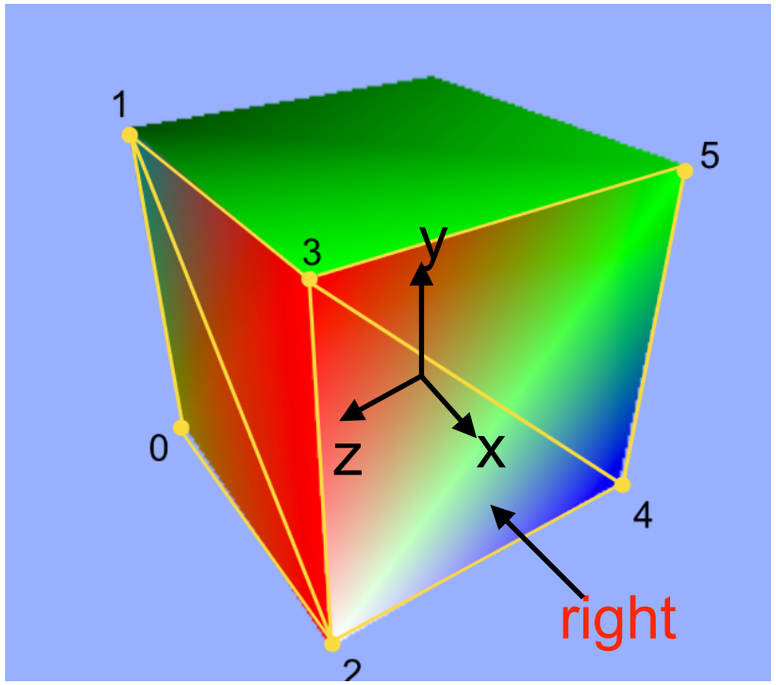
}

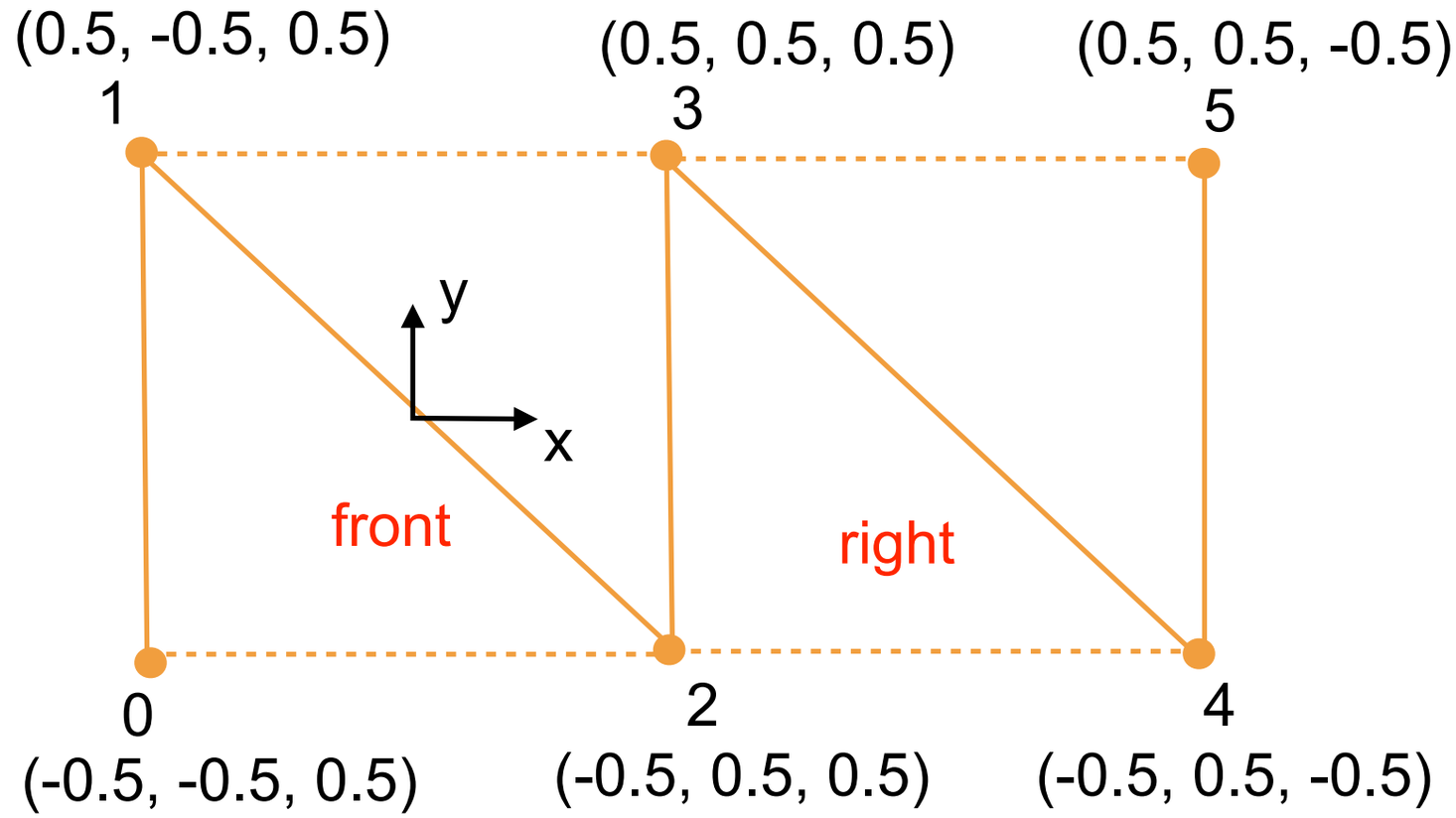
Triangle Data



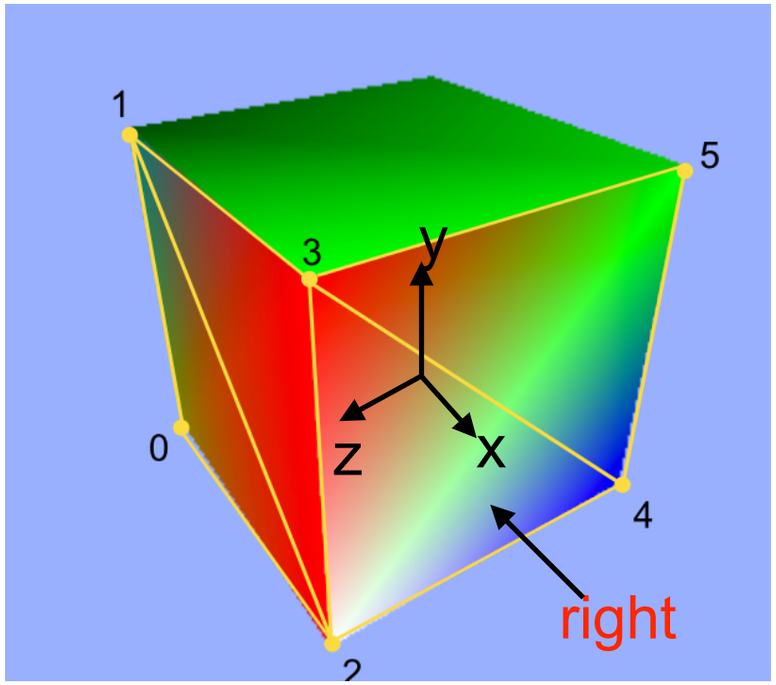


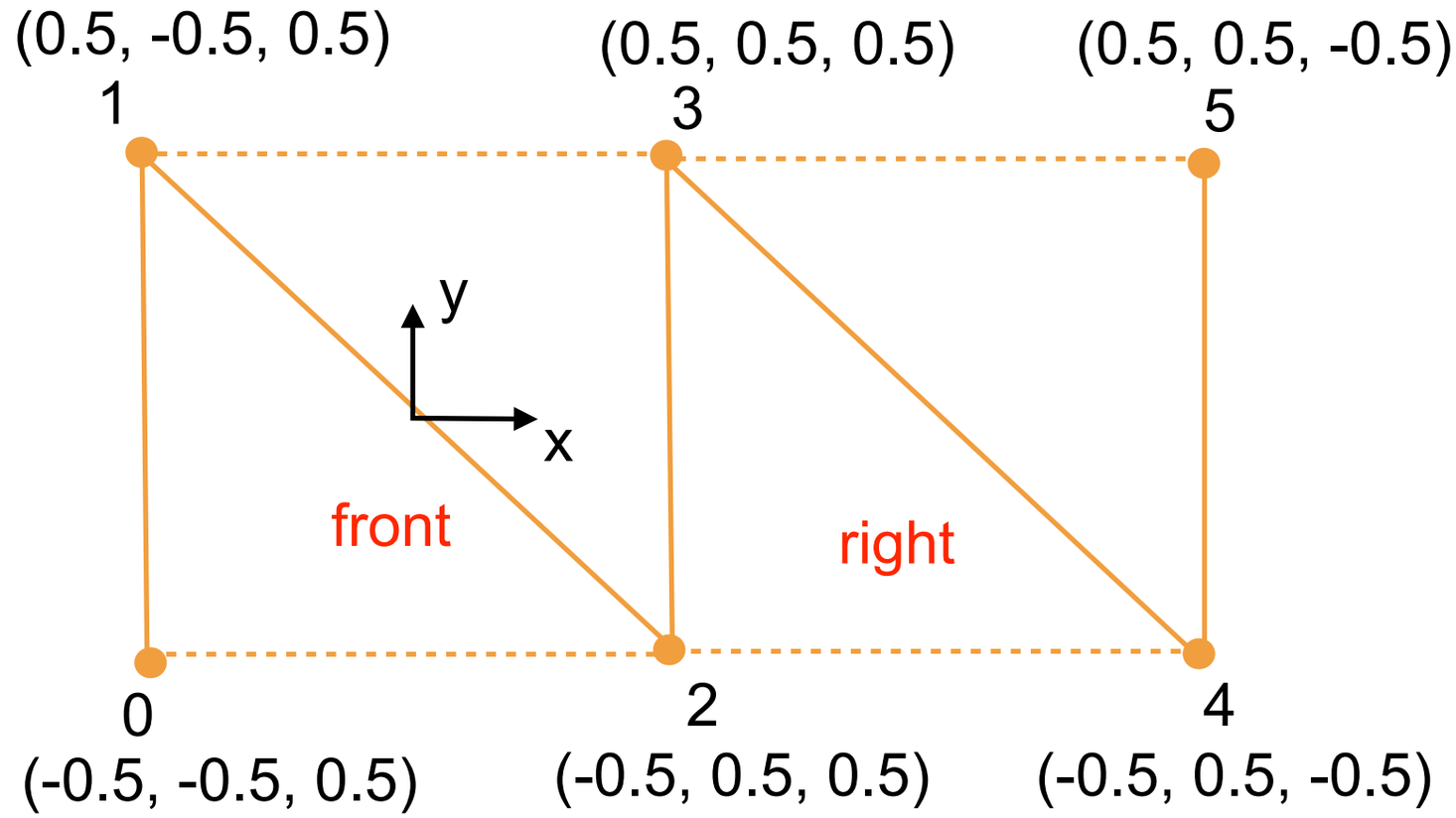
Triangle Data





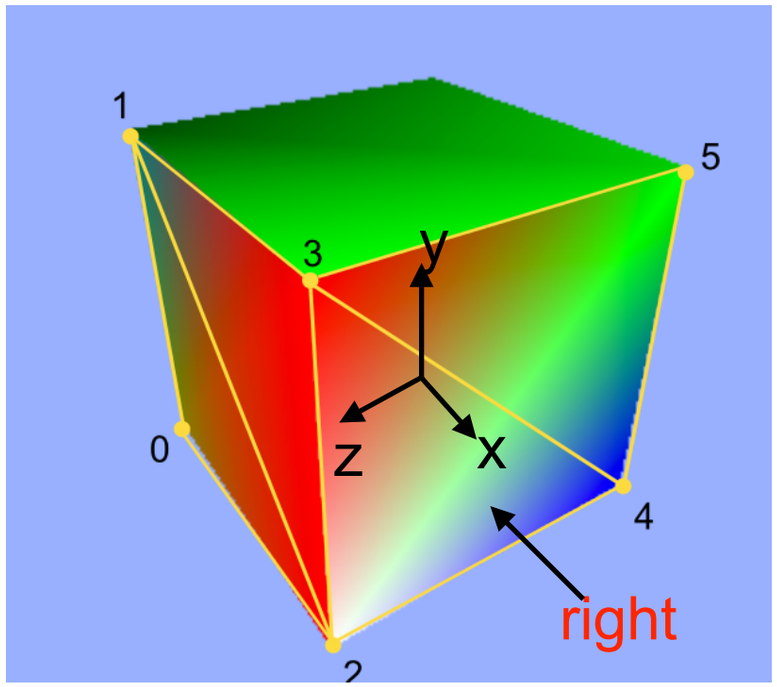
Triangle Data

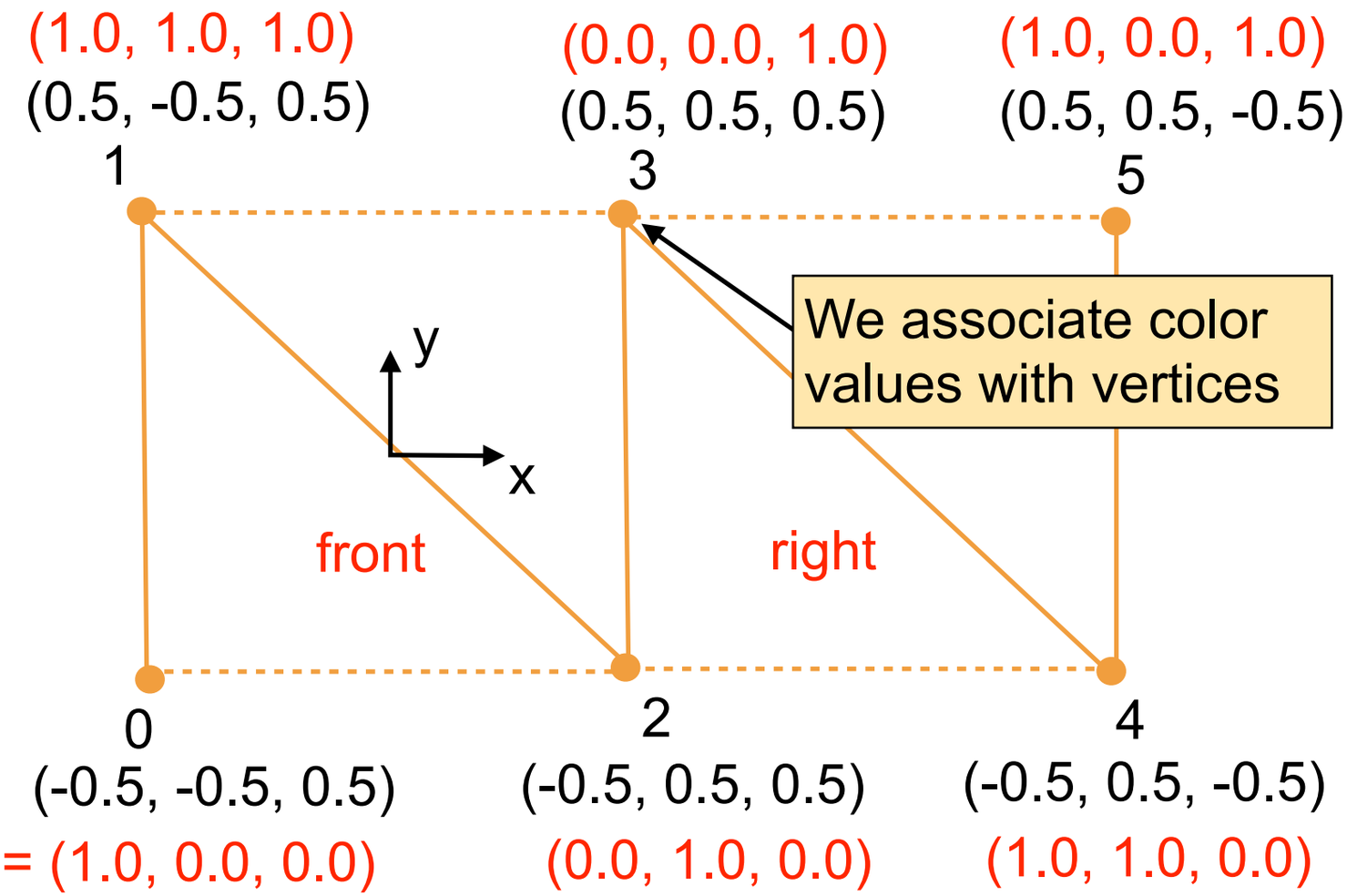


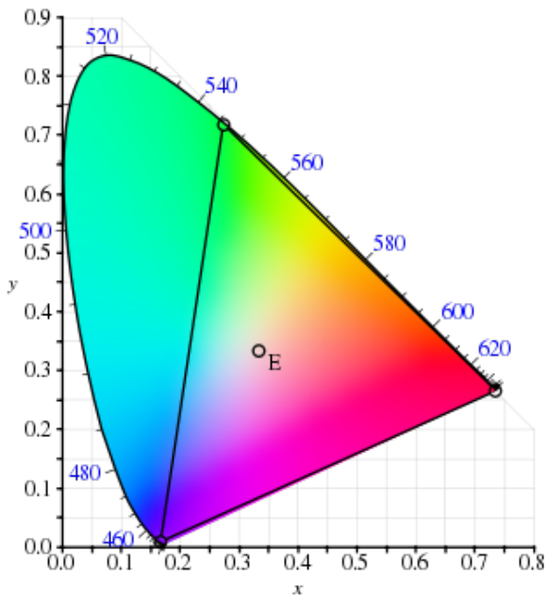


- Select the primitive you want to use
- Define the primitives, the location of each primitive (coordinates)
- Be aware of your coordinate system
- Every primitive should fit into a plane

Triangle Data







Color values are, if not specified in any other way, defined by a linear combinations of the color values

Red	Green	Blue

with each value in a range [0.0, 1.0]

Note, here, the color values are random

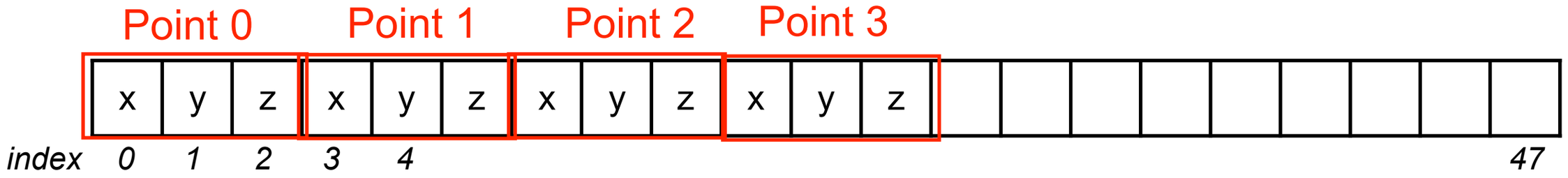
C++ Object

Create an array of float objects and add all values to the object

Float pointer

Size, we need to know the number of elements

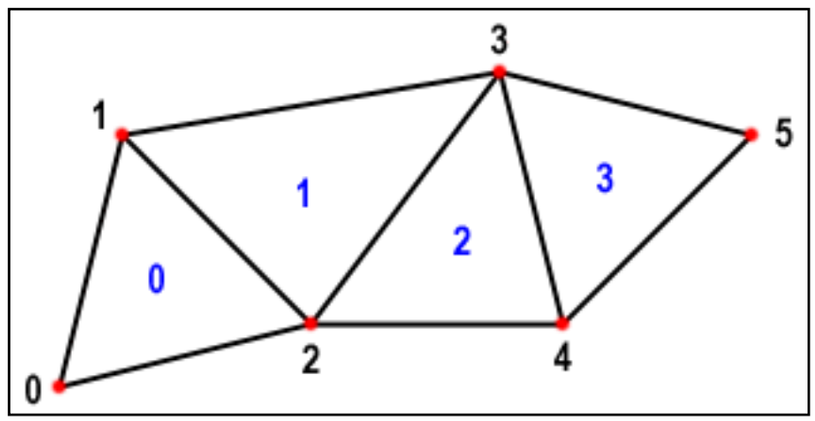
```
float* vertices = new float[48];  // Vertices for our square
float *colors = new float[48]; // Colors for our vertices
```



Index starts with 0

last index is 47

The sequence of points defines the triangles



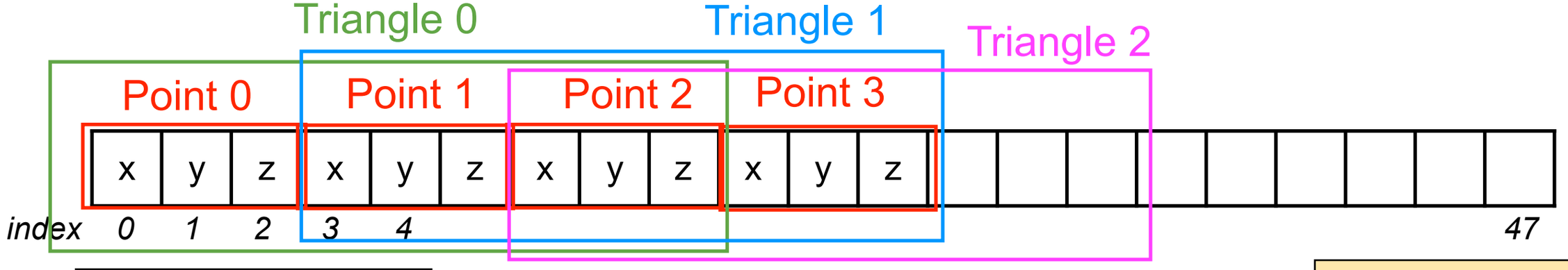
C++ Object

Create an array of float objects and add all values to the object

Float pointer

Size, we need to know the number of elements

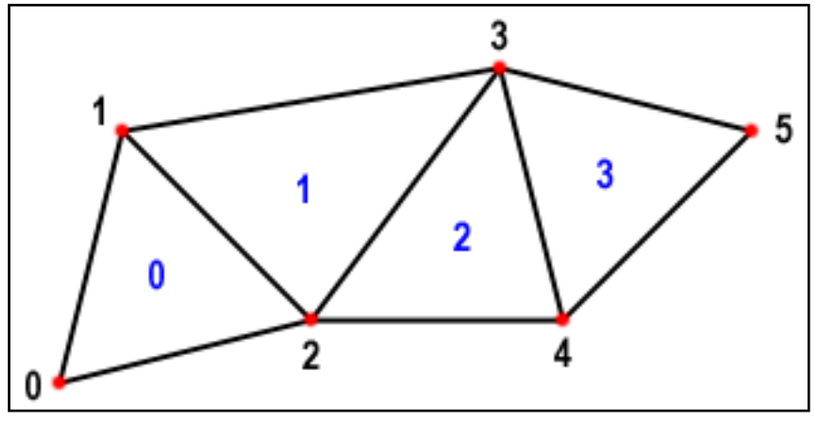
```
float* vertices = new float[48];  // Vertices for our square
float *colors = new float[48]; // Colors for our vertices
```



Index starts with 0

last index is 47

The sequence of points defines the triangles



Code Example

ARLAB

```
void createBox(void)
{
    float* vertices = new float[48];  // Vertices for our square
    float *colors = new float[48]; // Colors for our vertices

    // FRONT
    vertices[0] = -0.5; vertices[1] = 0.5; vertices[2] = 0.5; // Top left corner
    colors[0] = 1.0; colors[1] = 0.0; colors[2] = 0.0; // Top left corner

    vertices[3] = -0.5; vertices[4] = -0.5; vertices[5] = 0.5; // Bottom left corner
    colors[3] = 1.0; colors[4] = 1.0; colors[5] = 1.0; // Bottom left corner

    vertices[6] = 0.5; vertices[7] = 0.5; vertices[8] = 0.5; // Top Right corner
    colors[6] = 0.0; colors[7] = 1.0; colors[8] = 0.0; // Top Right corner

    vertices[9] = 0.5; vertices[10] = -0.5; vertices[11] = 0.5; // Bottom right corner
    colors[9] = 0.0; colors[10] = 0.0; colors[11] = 1.0; // Bottom right corner

// RIGHT SIDE
vertices[12] = 0.5; vertices[13] = 0.5; vertices[14] = -0.5; // Top right corner
colors[12] = 1.0; colors[13] = 1.0; colors[14] = 0.0; // Top right corner

vertices[15] = 0.5; vertices[16] = -0.5; vertices[17] = -0.5; // Bottom right corner
colors[15] = 1.0; colors[16] = 0.0; colors[17] = 1.0; // Bottom right corner

// BACK
vertices[18] = -0.5; vertices[19] = 0.5; vertices[20] = -0.5; // Top left  corner
colors[18] = 0.0; colors[19] = 0.5; colors[20] = 0.5; // Top left corner

vertices[21] = -0.5; vertices[22] = -0.5; vertices[23] = -0.5; // Bottom left  corner
colors[21] = 0.5; colors[22] = 0.5; colors[23] = 0.0; // Bottom left corner

// LEFT
vertices[24] = -0.5; vertices[25] = 0.5; vertices[26] = 0.5; // Top left  corner
colors[24] = 1.0; colors[25] = 0.0; colors[26] = 0.0; // Top left corner

vertices[27] = -0.5; vertices[28] = -0.5; vertices[29] = 0.5; // Bottom left  corner
colors[27] = 1.0; colors[28] = 0.0; colors[29] = 0.0; // Bottom left corner

 // BOTTOM
vertices[30] = 0.5; vertices[31] = -0.5; vertices[32] = -0.5; // Top left  corner
colors[30] = 1.0; colors[31] = 0.0; colors[32] = 0.0; // Top left corner
```

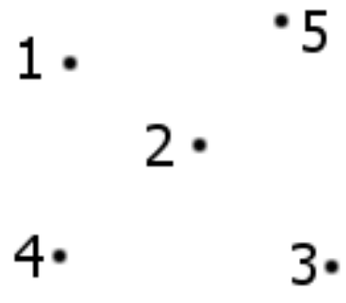
ARLAB

OpenGL objects for the primitive data

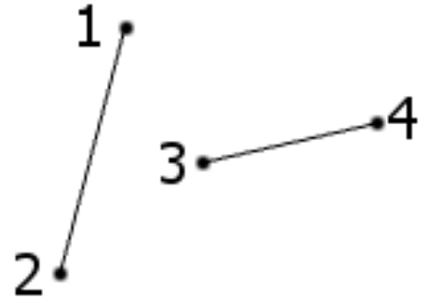
OpenGL Primitives

ARLAB

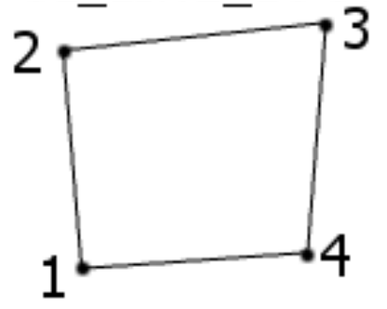
GL_POINTS



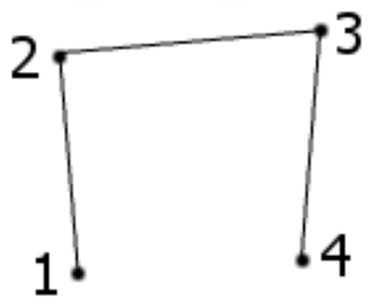
GL_LINES



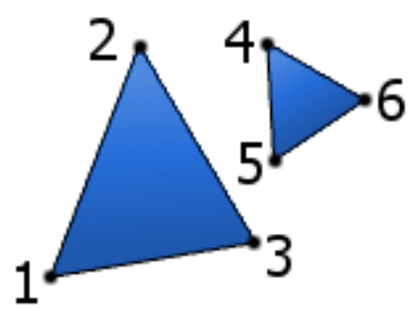
GL_LINE_LOOP



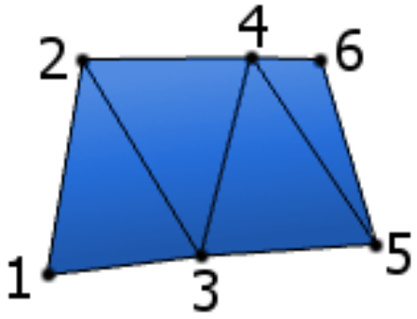
GL_LINE_STRIP



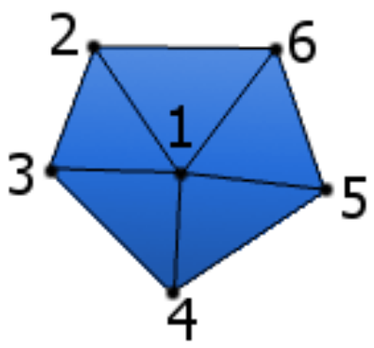
GL_TRIANGLES



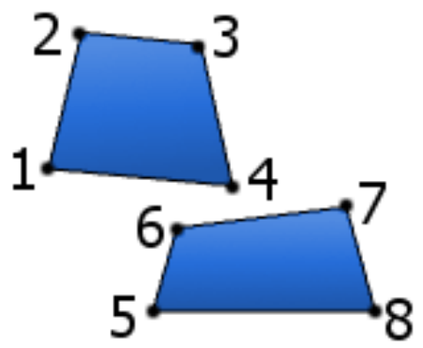
GL_TRIANGLE_STRIP



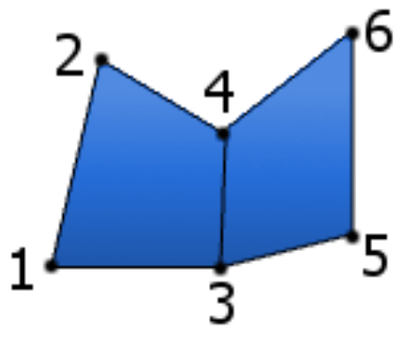
GL_TRIANGLE_FAN



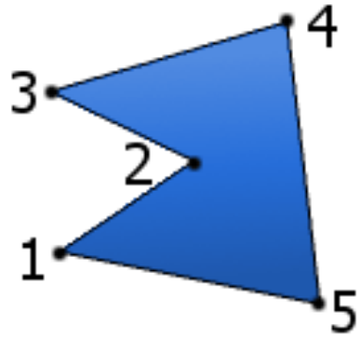
GL_QUADS



GL_QUAD_STRIP



GL_POLYGON



Vertex Buffer Object

ARLAB

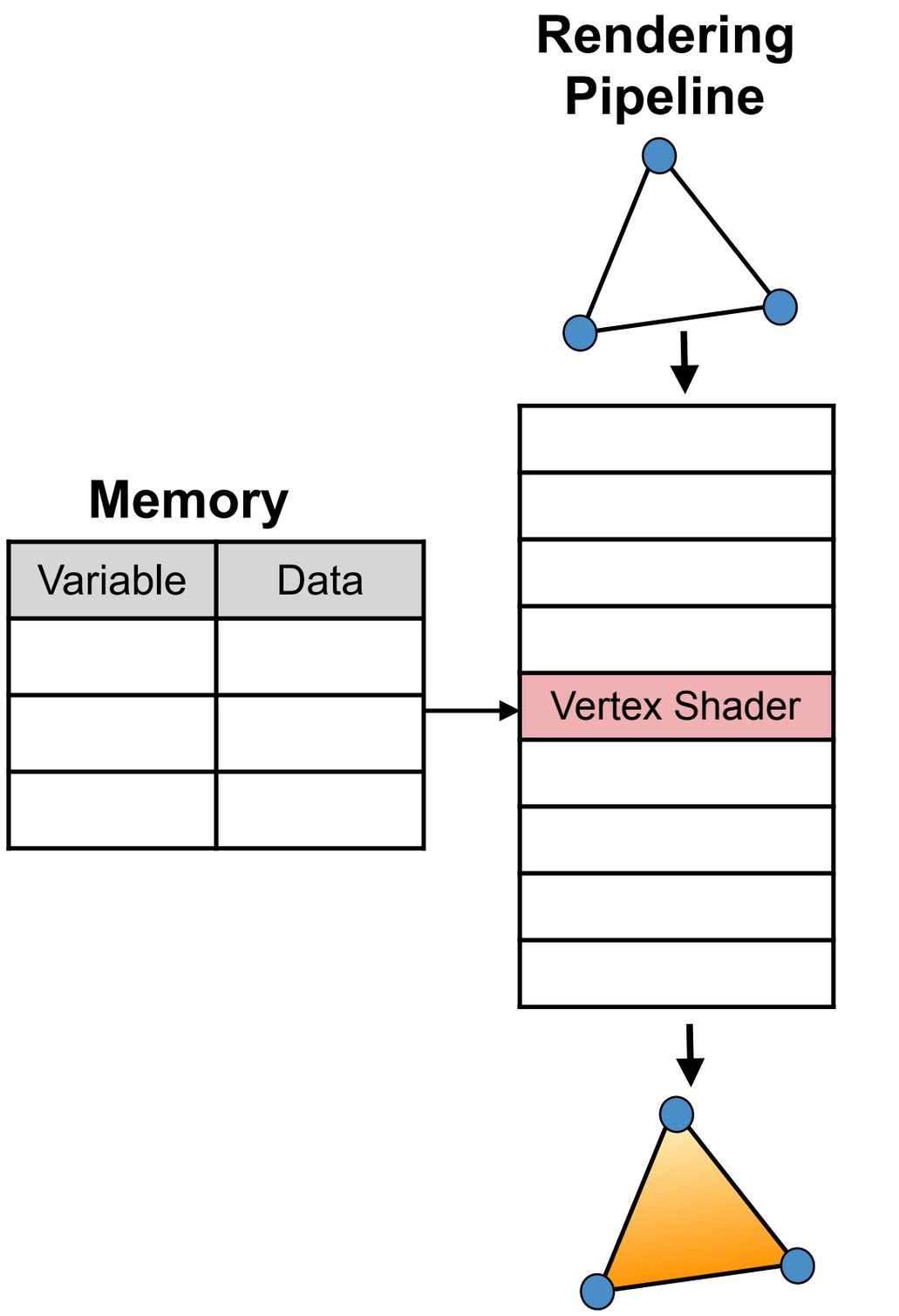
```
void glGenBuffers(GLsizei n, GLuint *buffers);
```

Generate buffer object names

Parameters:

- n: Specifies the number of buffer object names to be generated.
- buffers: Specifies an array in which the generated buffer object names are stored.

A **Vertex Buffer Object (VBO)** is an OpenGL capability to upload vertex data (position, normal vector, color, etc.) to the video device's memory for non-immediate rendering. VBOs cause substantial performance gains over immediate mode rendering (fixed function pipeline rendering) because we copy the data in the video device memory before we render where it resides and so it can be rendered directly by the video device.
The fixed function rendering pipeline required us to copy the vertex data every frame to the graphics memory.



Vertex Buffer Object

ARLAB

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

Generate buffer object names

Parameters:

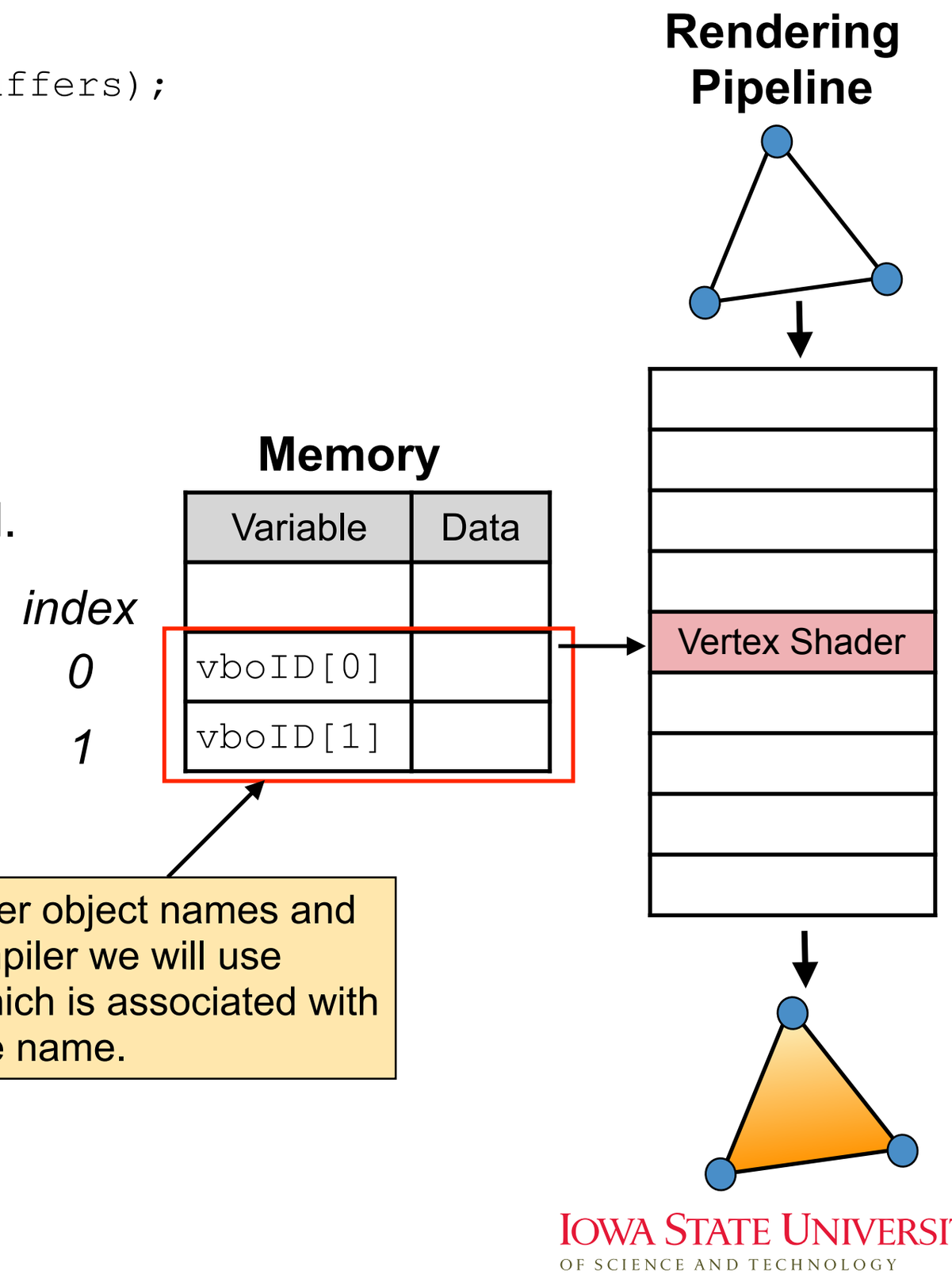
- n: Specifies the number of buffer object names to be generated.
- buffers: Specifies an array in which the generated buffer object names are stored.

Example:

```
unsigned int vboID[2];

    [...]

glGenBuffers(2, vboID);
```



Vertex Array Object

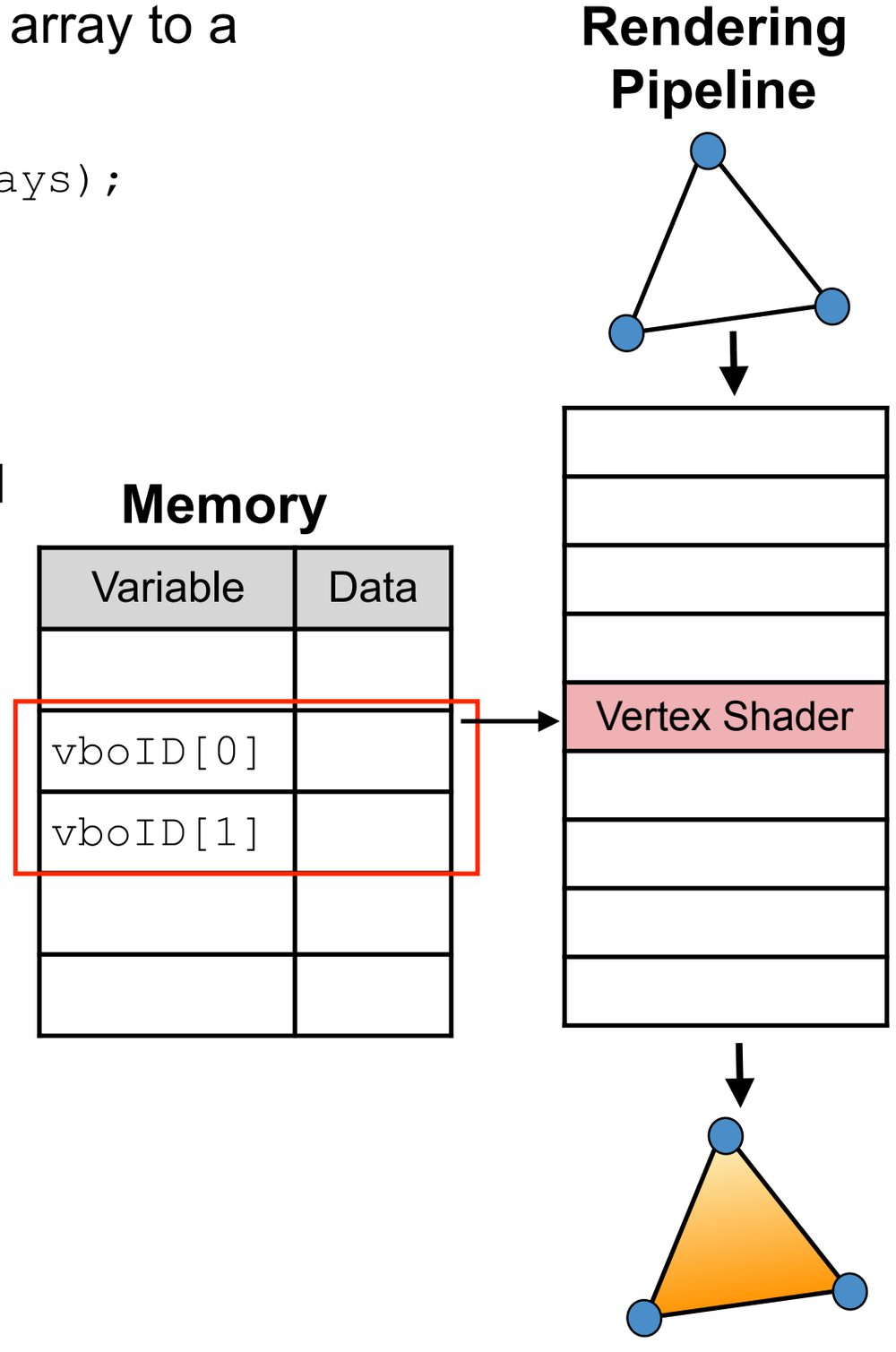
Generate vertex array objects. The function ties the array to a variable name.

```
void glGenVertexArrays(GLsizei n, GLuint *arrays);
```

Parameters:

- n: Specifies the number of vertex array object names to generate.
- arrays: Specifies an array in which the generated vertex array object names are stored.

A Vertex Array Object (VAO) is an OpenGL object that stores all of the state data needed to supply primitive data (vertices, color, normals, etc.). It can be considered as an envelop or box object that allows us to access and handle all data at once.



Vertex Array Object

Generate vertex array objects. The function ties the array to a variable name.

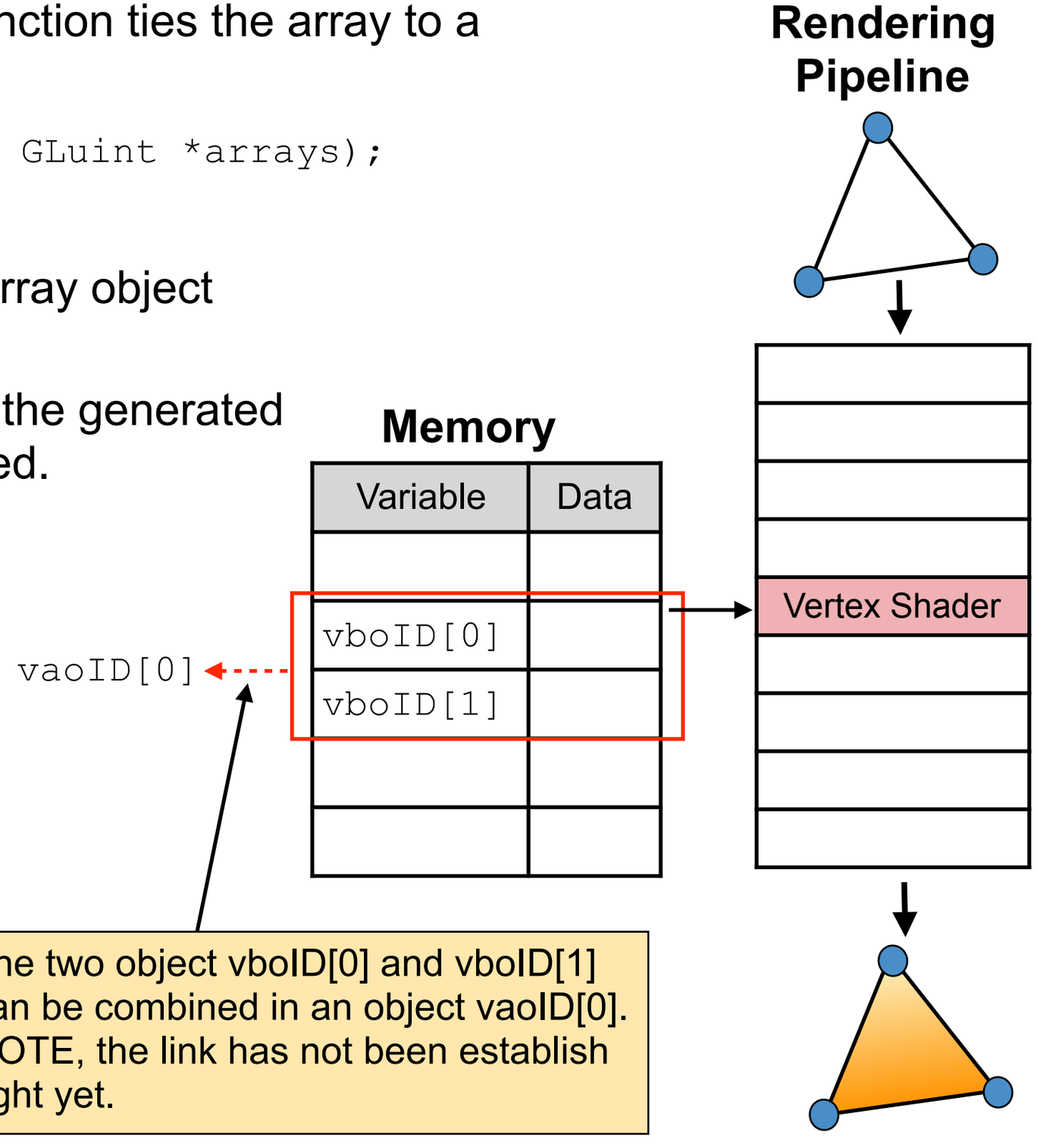
```
void glGenVertexArrays(GLsizei n, GLuint *arrays);
```

Parameters:

- n: Specifies the number of vertex array object names to generate.
- arrays: Specifies an array in which the generated vertex array object names are stored.

Example:

```
// the vertex array object
unsigned int vaoID[1];
glGenVertexArrays(2, &vaoID[0]);
```



Bind Vertex Array

ARLAB

```
void glBindVertexArray(GLuint array);
```

Bind a vertex array object

Parameters:

- array: Specifies the name of the vertex array to bind.

Example:

```
glGenVertexArrays(1, &vaoID[0]); // Create our Vertex Array Object
glBindVertexArray(vaoID[0]); //
```

All OpenGL bind operations bind the called memory, which means, the switch it to an active memory. All function calls / operations that are invoked after this call are applied to the active memory.

Bind Vertex Array

```
int main(int argc, const char * argv[])
{
        [....]
        glGenVertexArrays(2, &vaoID[0]);
        glBindVertexArray(vaoID[0]); //
```

All operations affect vaoID[0]

[....]

```
        glBindVertexArray(vaoID[1]); //
```

All operations affect vaoID[1]

[....]

Program init

```
    while
        glBindVertexArray(vaoID[0]); //
```

All operations affect vaoID[0]

[....]

main loop

```
}
```

All operations that are related to a vertex array object (vertices, color, normals).
Later, when we will bind textures, they do not influence vertex arrays.

Memory

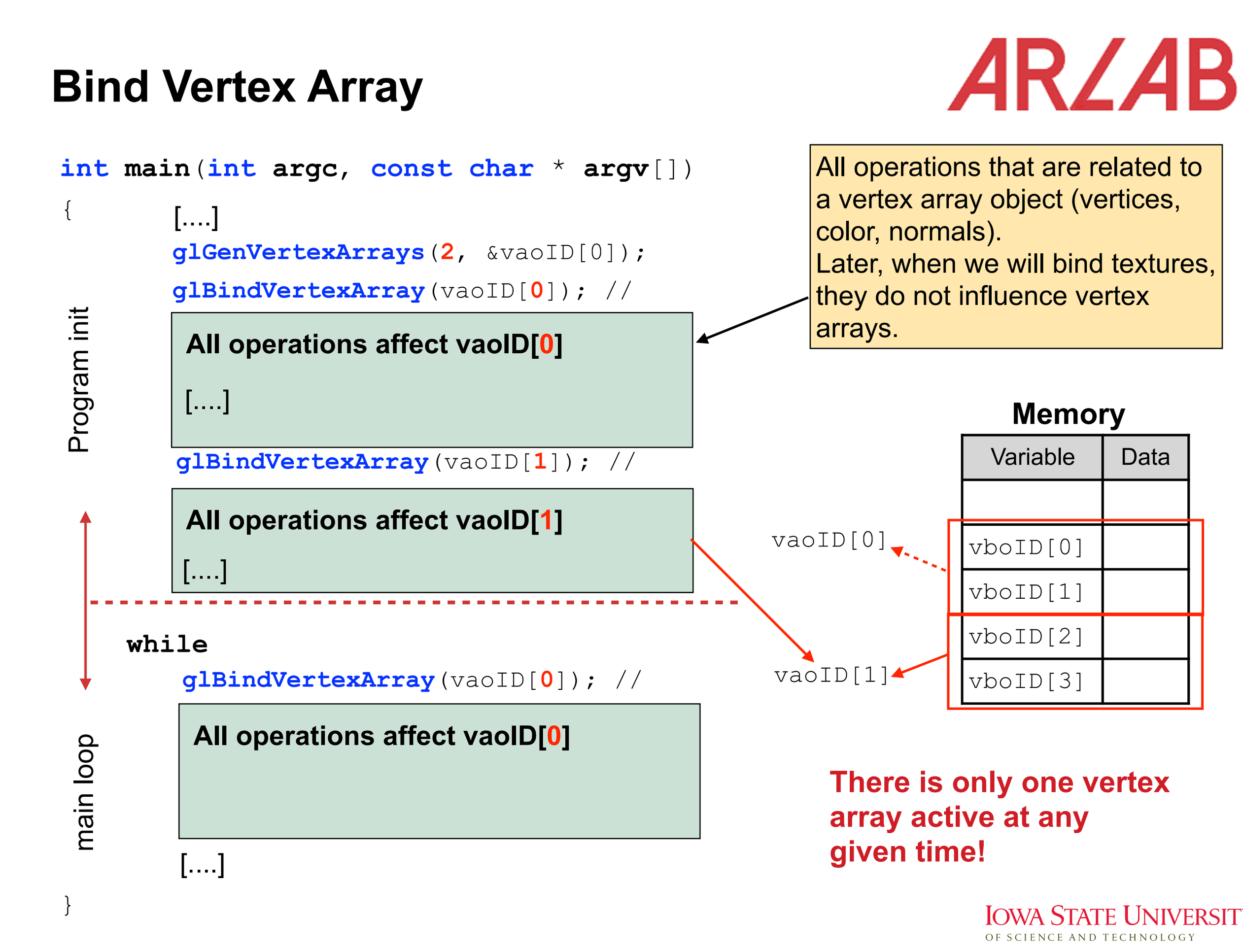
Variable	Data
vboID[0]	
vboID[1]	
vboID[2]	
vboID[3]	

vaoID[0]

vaoID[1]

There is only one vertex array active at any given time!

Bind Vertex Array



Sequence

```
int main(int argc, const char * argv[])
{
        [....]
        glGenVertexArrays(2, &vaoID[0]);
        glBindVertexArray(vaoID[0]); //
```

All operations affect vaoID[0]

[....]

```
        glBindVertexArray(vaoID[1]); //
```

All operations affect vaoID[1]

[....]

```
        while
            glBindVertexArray(vaoID[0]); //
```

All operations affect vaoID[0]

[....]

```
}
```

Program init

main loop

Sequence in our code!

1. Generate the vertex array
2. Bind one vertex array; even if you only generated one, switch it to active.

Bind Buffer

ARLAB

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

Parameters:

- target: Specifies the target to which the buffer object is bound, which must be one of the buffer binding targets in the following table:

Buffer Binding Target	Purpose
GL_ARRAY_BUFFER	Vertex attributes
GL_ATOMIC_COUNTER_BUFFER	Atomic counter storage
GL_COPY_READ_BUFFER	Buffer copy source
GL_COPY_WRITE_BUFFER	Buffer copy destination
GL_DISPATCH_INDIRECT_BUFFER	Indirect compute dispatch commands
GL_DRAW_INDIRECT_BUFFER	Indirect command arguments
GL_ELEMENT_ARRAY_BUFFER	Vertex array indices
GL_PIXEL_PACK_BUFFER	Pixel read target
GL_PIXEL_UNPACK_BUFFER	Texture data source
GL_QUERY_BUFFER	Query result buffer
GL_SHADER_STORAGE_BUFFER	Read-write storage for shaders
GL_TEXTURE_BUFFER	Texture data buffer
GL_TRANSFORM_FEEDBACK_BUFFER	Transform feedback buffer
GL_UNIFORM_BUFFER	Uniform block storage

Bind Vertex Array

```
int main(int argc, const char * argv[])
{
        [....]
        glGenVertexArrays(2, &vaoID[0]);
        glBindVertexArray(vaoID[0]); //
            glGenBuffers(2, vboID);
            glBindBuffer(GL_ARRAY_BUFFER, vboID[0]);
        glBindVertexArray(vaoID[1]); //
        while
            glBindVertexArray(vaoID[0]); //
            All operations affect vaoID[0]
        [....]
}
```

Program init

main loop

All operations that are related to the vertex buffer object vboID[0]

Memory

Variable	Data
vboID[0]	
vboID[1]	
vboID[2]	
vboID[3]	

vaoID[0]

There is only one vertex buffer active at any given time!

Bind Vertex Array

```
int main(int argc, const char * argv[])
{
        [....]
        glGenVertexArrays(2, &vaoID[0]);
        glBindVertexArray(vaoID[0]); //
            glGenBuffers(2, vboID);
            glBindBuffer(GL_ARRAY_BUFFER, vboID[0]);
                What is missing?
        glBindVertexArray(vaoID[1]); //
while
        glBindVertexArray(vaoID[0]); //
            All operations affect vaoID[0]
        [....]
}
```

Program init

main loop

All operations that are related to the vertex buffer object vboID[0]

Memory

Variable	Data
vboID[0]	
vboID[1]	
vboID[2]	
vboID[3]	

vaoID[0]

There is only one vertex buffer active at any given time!

Copy Buffer Data

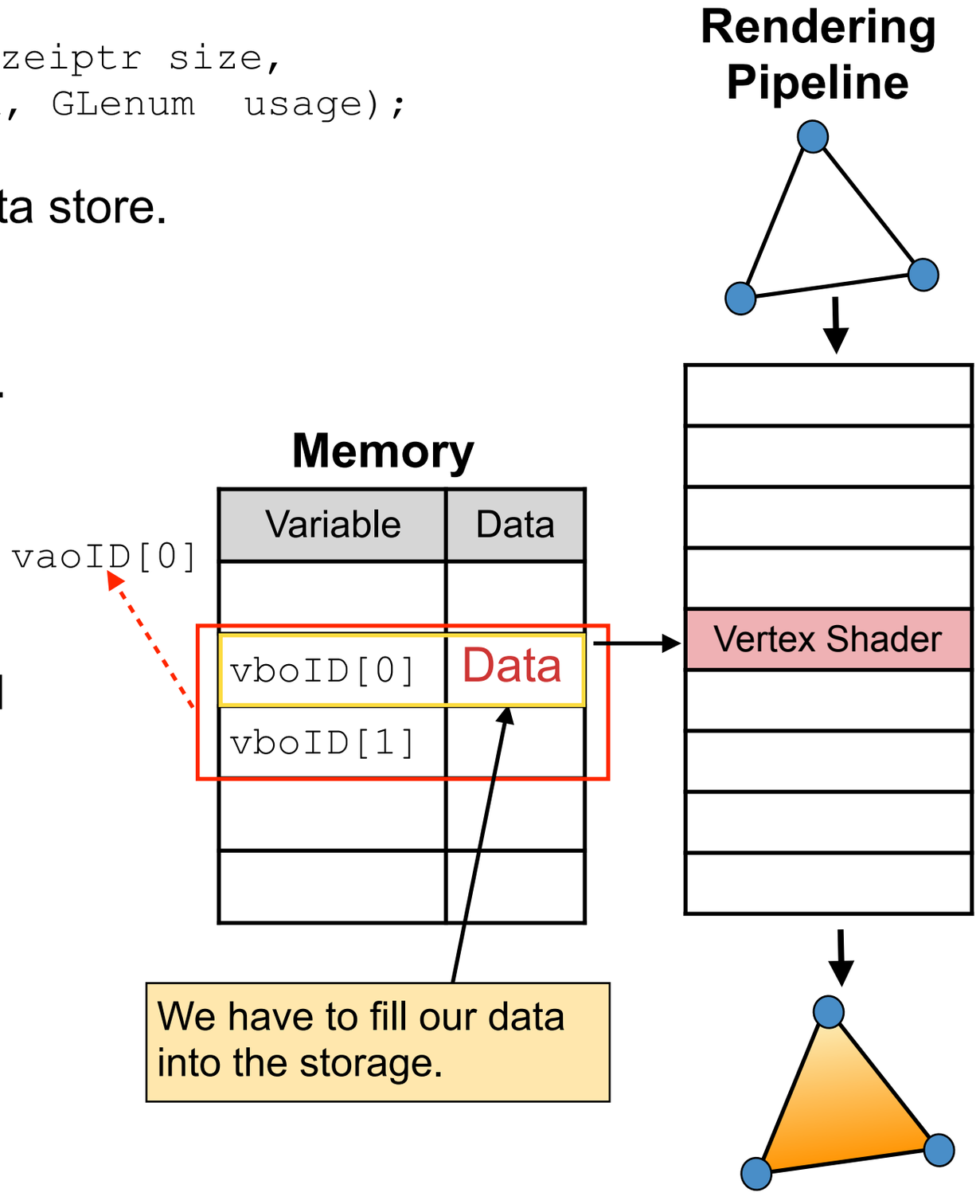
ARLAB

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

Creates and initializes a buffer object's data store.

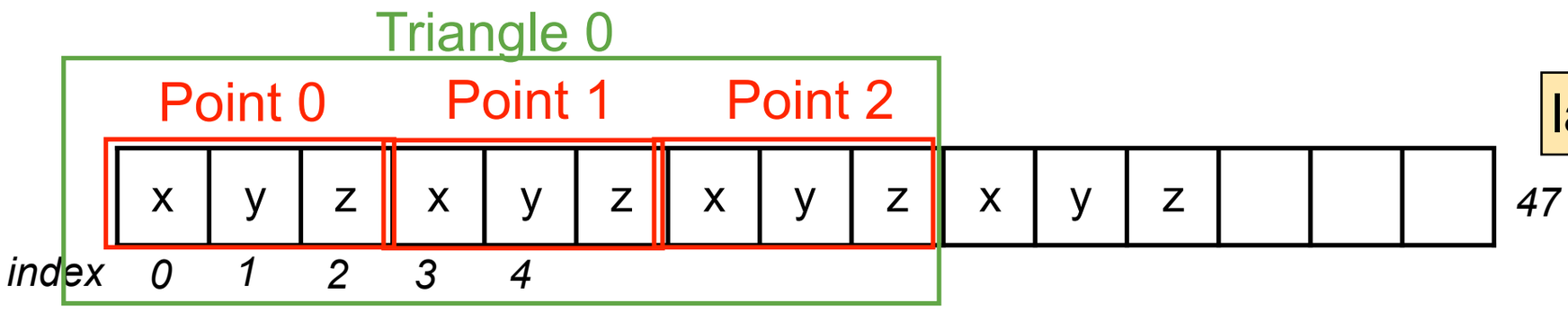
Parameters:

- target: Specifies the target buffer object. The symbolic constant must be GL_ARRAY_BUFFER
- size: Specifies the size in bytes of the buffer object's new data store.
- data: Specifies a pointer to data that will be copied into the data store for initialization, or NULL if no data is to be copied.
- usage: Specifies the expected usage pattern of the data store. The symbolic constant must be GL_STATIC_DRAW, GL_DYNAMIC_DRAW



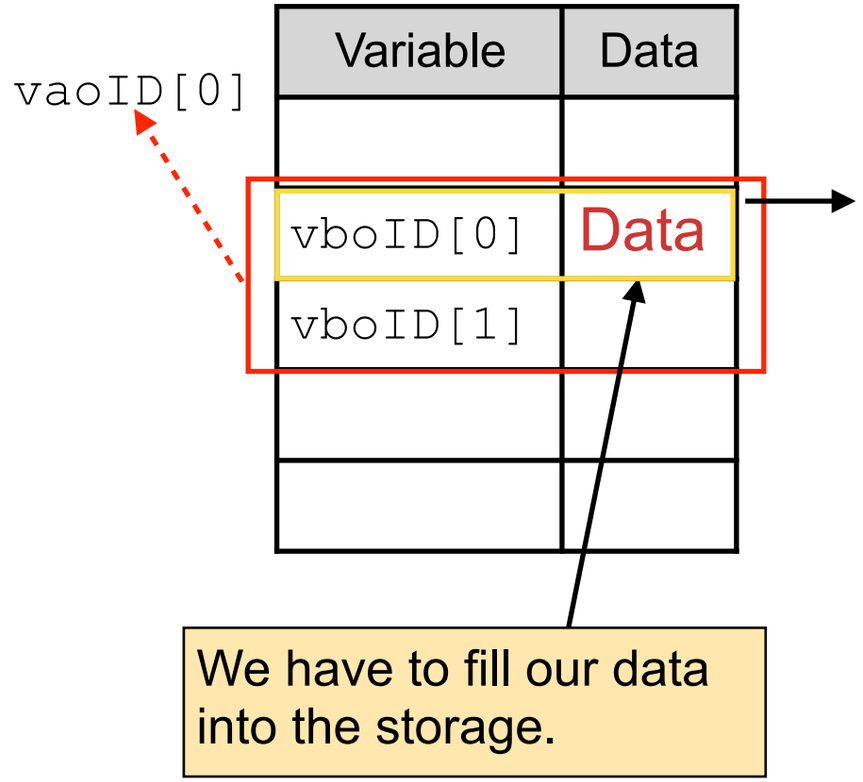
Copy Buffer Data

```
float* vertices = new float[48];  // Vertices for our square
float *colors = new float[48]; // Colors for our vertices
```



last index is 47

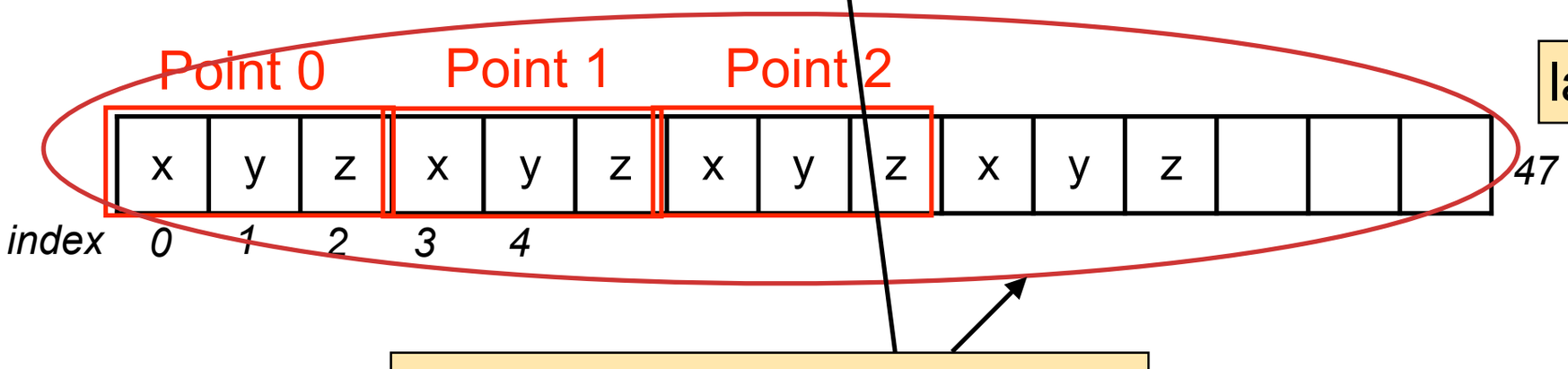
```
glBufferData(   GL_ARRAY_BUFFER,
                48 * sizeof(GLfloat),
                vertices, GL_STATIC_DRAW);
```



Copy Buffer Data

ARLAB

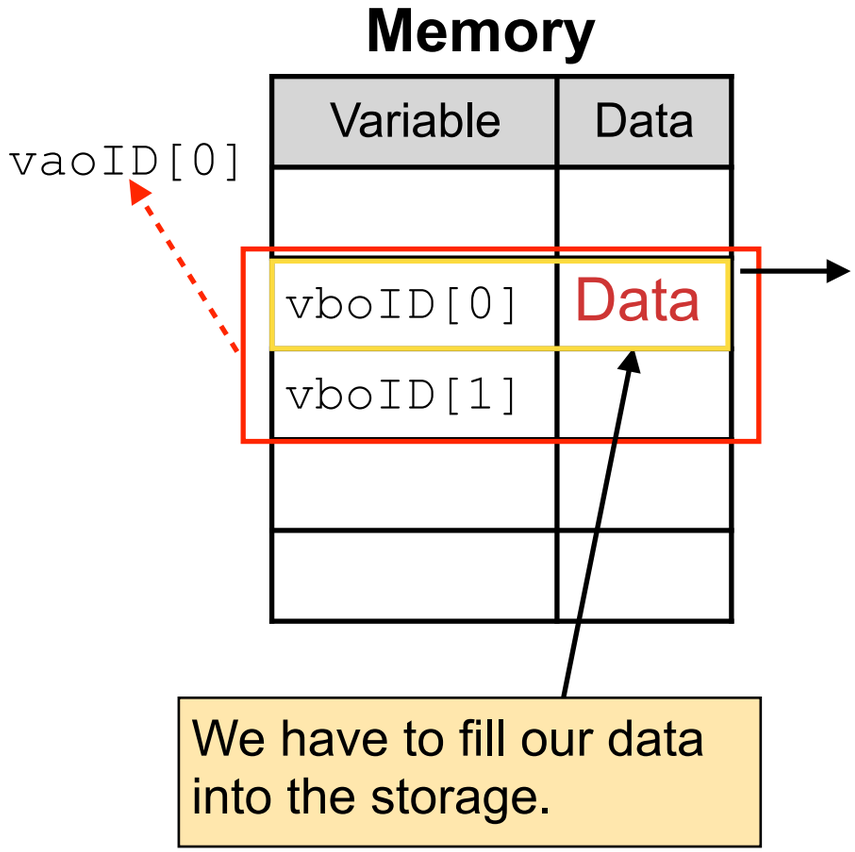
```
float* vertices = new float[48];  // Vertices for our square
float *colors = new float[48]; // Colors for our vertices
```



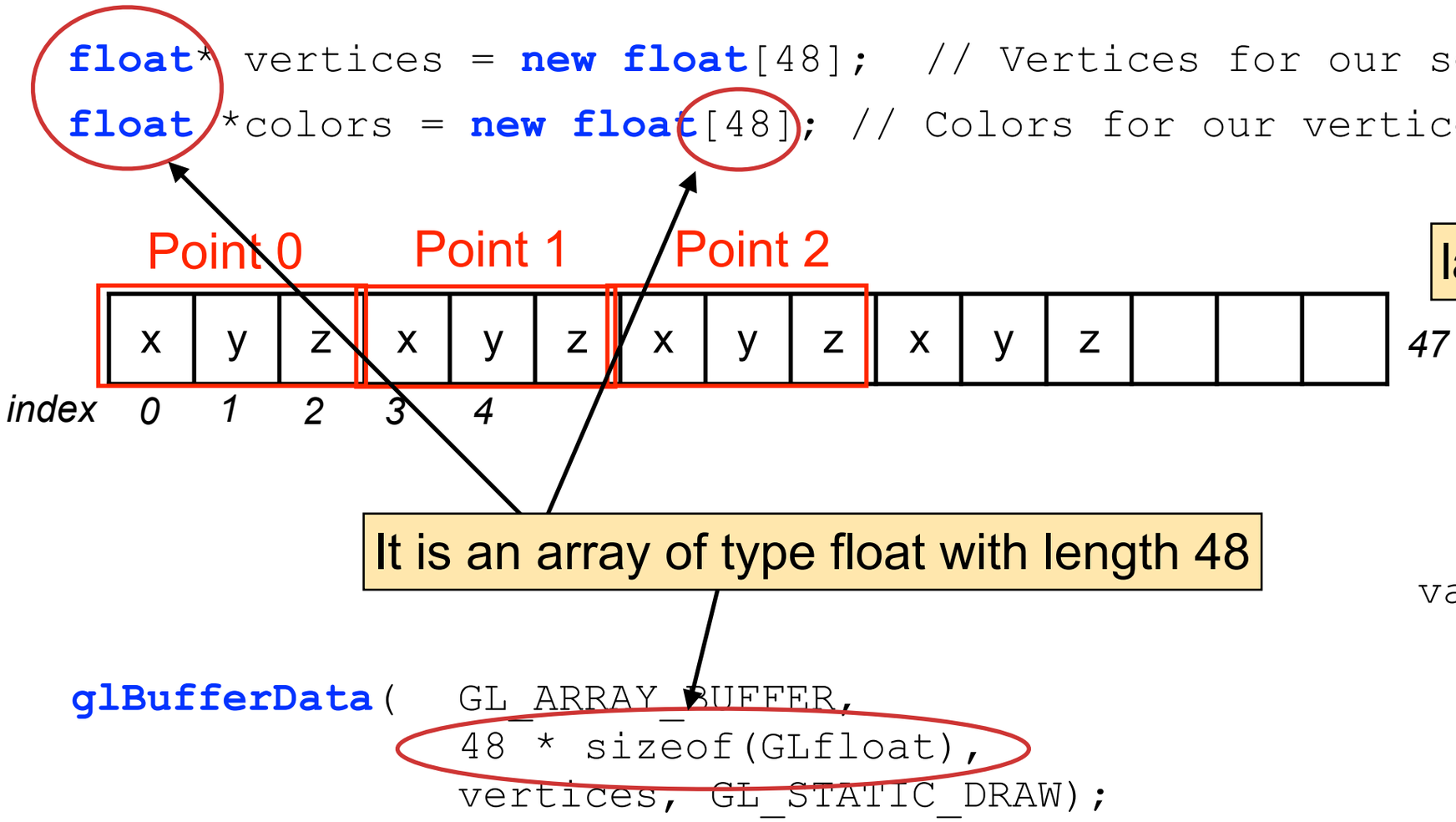
last index is 47

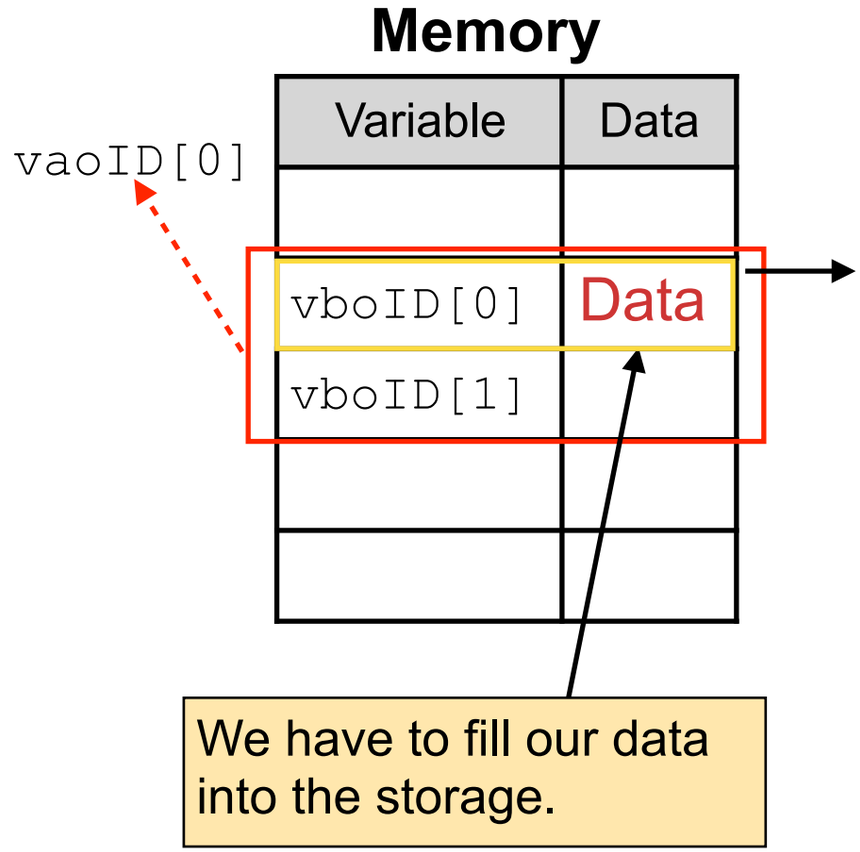
The entire data set is an array

```
glBufferData( GL_ARRAY_BUFFER,
              48 * sizeof(GLfloat),
              vertices, GL_STATIC_DRAW);
```



Copy Buffer Data



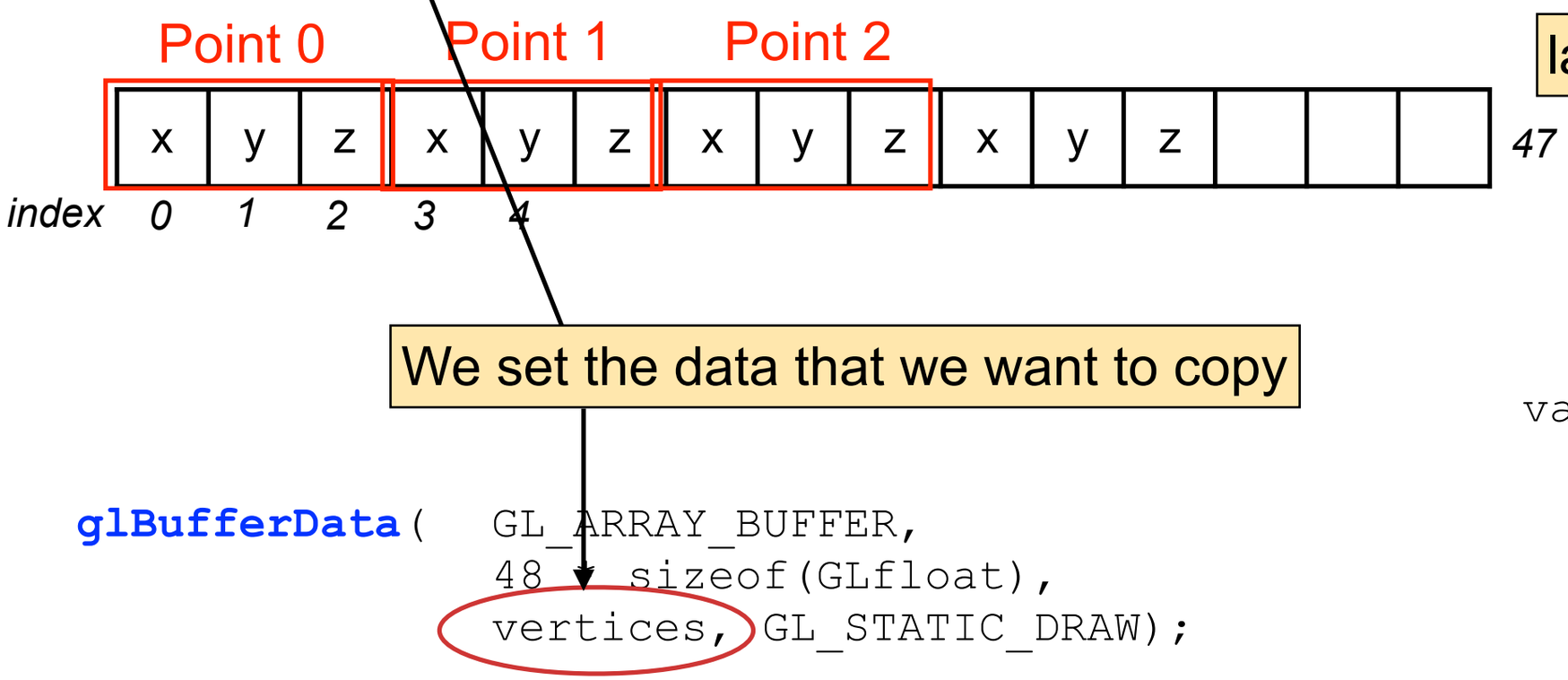


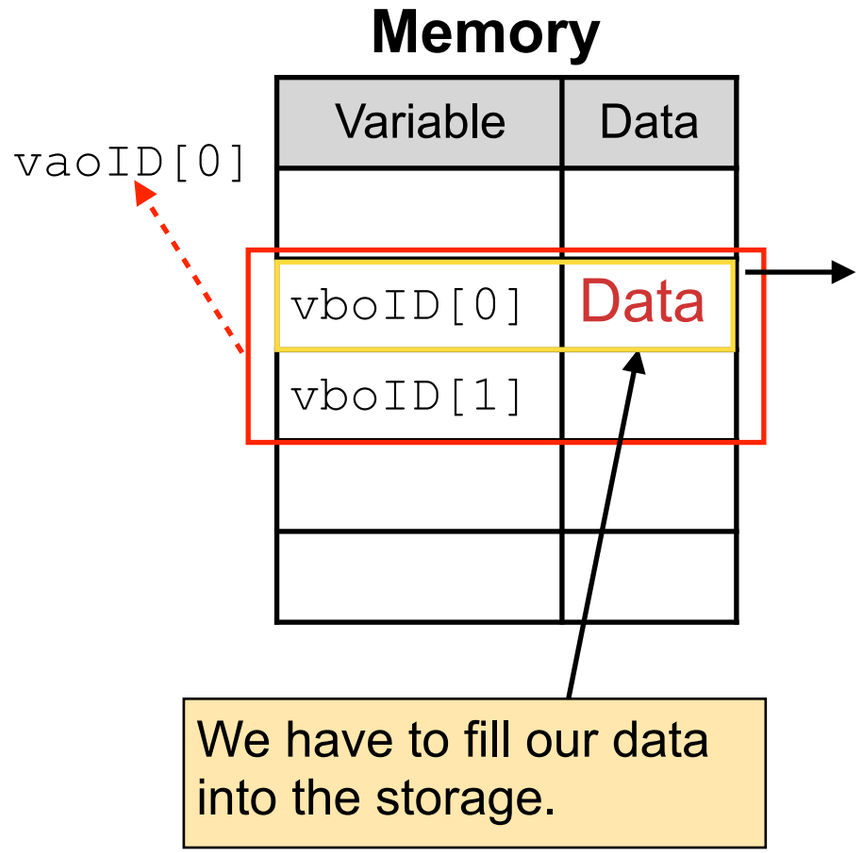
The second variable specifies how much memory we want create.

Copy Buffer Data

ARLAB

```
float* vertices = new float[48];  // Vertices for our square
float *colors = new float[48]; // Colors for our vertices
```

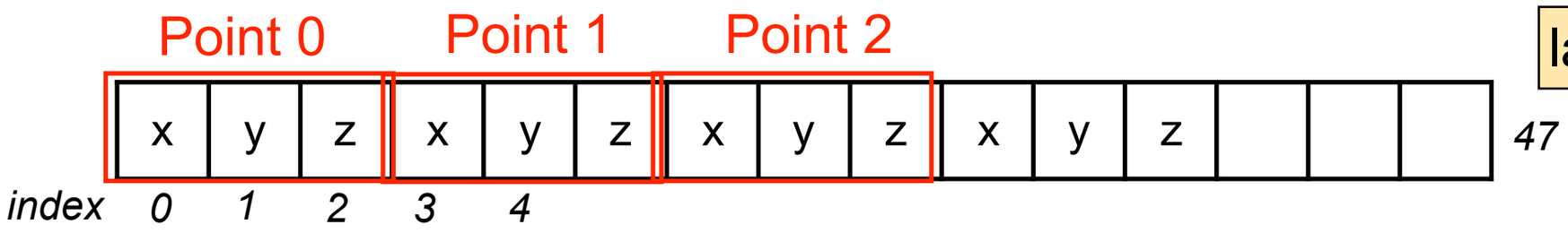




The second variable specifies how much memory we want create.

Copy Buffer Data

```
float* vertices = new float[48];  // Vertices for our square
float *colors = new float[48]; // Colors for our vertices
```

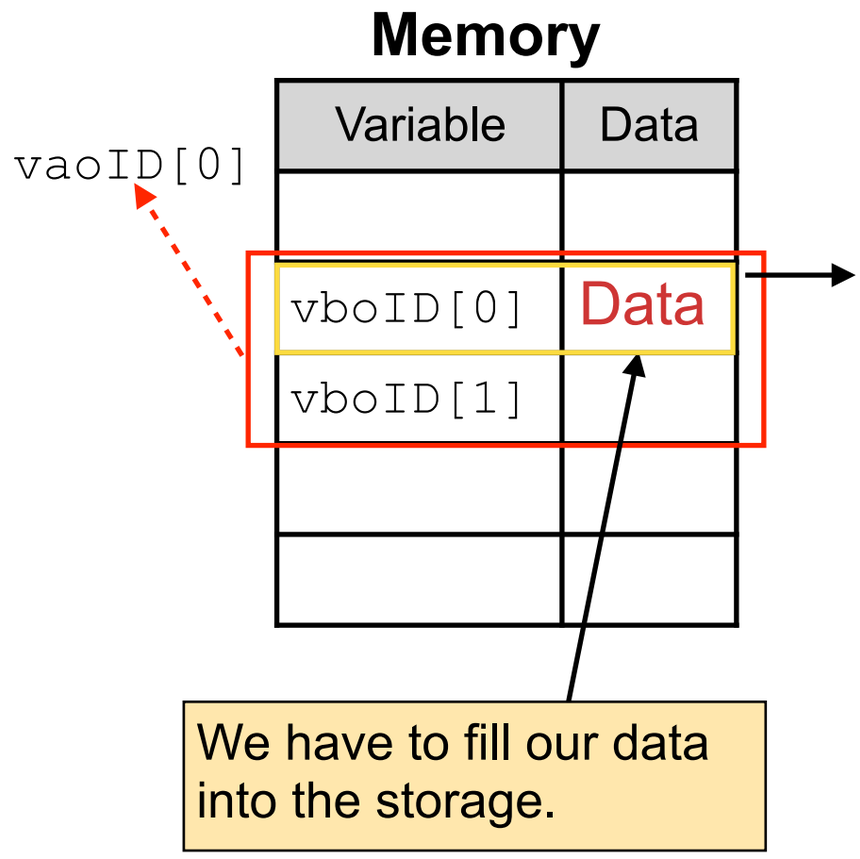


last index is 47

And set a specifier that says in this case,
we do promise not to change the values in the array.

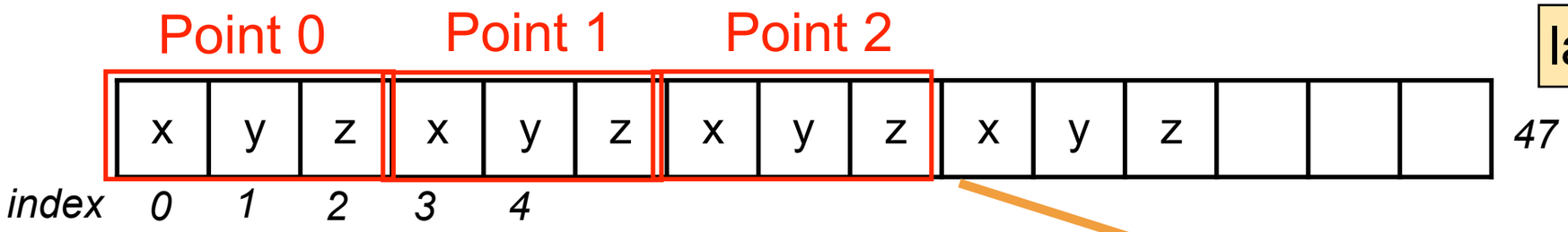
```
glBufferData(   GL_ARRAY_BUFFER,
                48 * sizeof(GLfloat),
                vertices, GL_STATIC_DRAW);
```

We need to know what we are going to do with this data



Copy Buffer Data

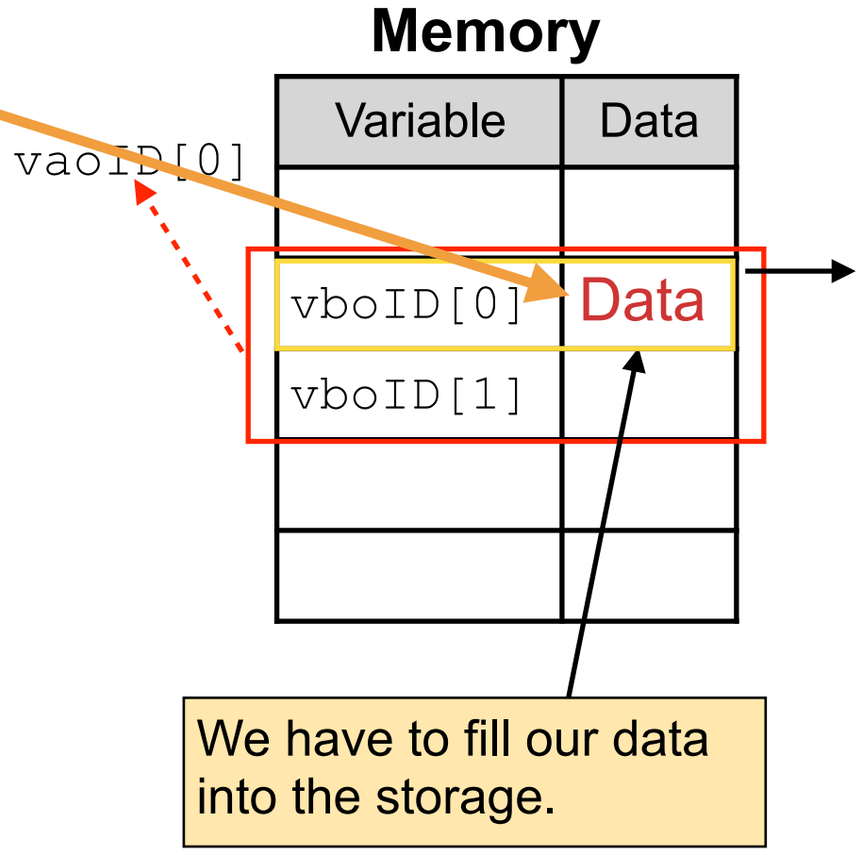
```
float* vertices = new float[48];  // Vertices for our square
float *colors = new float[48]; // Colors for our vertices
```



last index is 47

The entire function copies our data into the graphics card memory.

```
glBufferData(   GL_ARRAY_BUFFER,
                48 * sizeof(GLfloat),
                vertices, GL_STATIC_DRAW);
```



Bind Vertex Array

ARLAB

```
int main(int argc, const char * argv[])
{
        [....]
        glGenVertexArrays(2, &vaoID[0]);
```

Program init

```
        glBindVertexArray(vaoID[0]); //
            glGenBuffers(2, vboID);
            glBindBuffer(GL_ARRAY_BUFFER, vboID[0]);
            glBufferData(GL_ARRAY_BUFFER, ...
        glBindVertexArray(vaoID[1]); //
```

```
    while
        glBindVertexArray(vaoID[0]); //
```

All operations affect vaoID[0]

main loop

```
        [....]
}
```

Memory

Variable	Data
vboID[0]	Data
vboID[1]	
vboID[2]	
vboID[3]	

vaoID[0]

Define the Buffer Data

ARLAB

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

Define an array of generic vertex attribute data or in other words, explain the meaning of your variables.

Parameters:

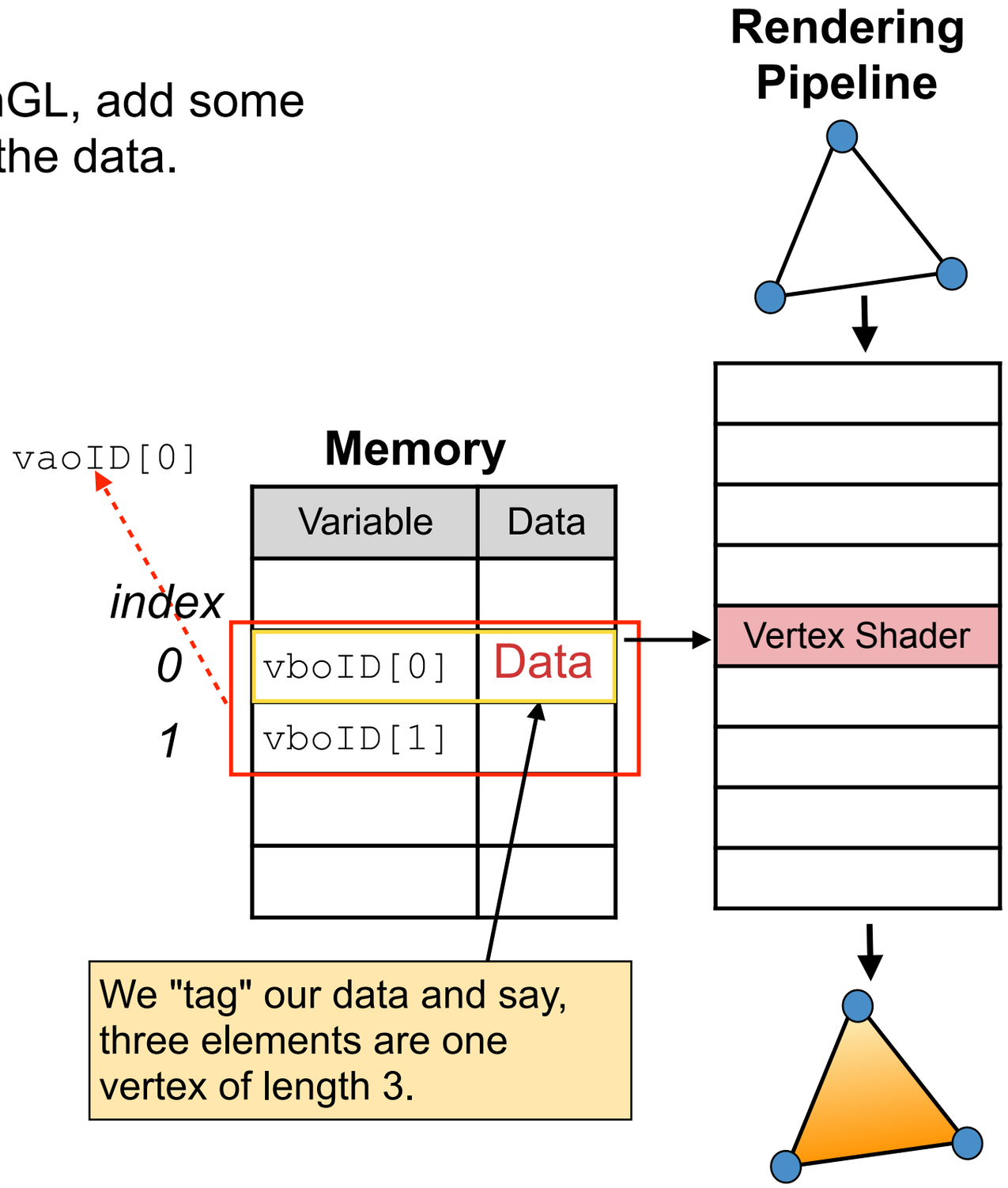
- index: Specifies the index of the generic vertex attribute to be modified.
- size: Specifies the number of components per generic vertex attribute. Must be 1, 2, 3, 4. Additionally, the symbolic constant GL_BGRA is accepted by glVertexAttribPointer. The initial value is 4.
- type: Specifies the data type of each component in the array. The symbolic constants GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, and GL_UNSIGNED_INT, GL_FLOAT, GL_DOUBLE, GL_FIXED
- normalized: values should be normalized (GL_TRUE) or not (GL_FALSE)
- stride: Specifies the byte offset between consecutive generic vertex attributes.
- pointer: Specifies an offset of the first component of the first generic vertex attribute in the array in the data store of the buffer currently bound to the GL_ARRAY_BUFFER target.

Define the Buffer Data

Explain OpenGL what the data is about.

Note, these are only 0,1 values for OpenGL, add some meaning that allow OpenGL to interpret the data.

```
glVertexAttribPointer((GLuint)0, 3,
                      GL_FLOAT,
                      GL_FALSE,
                      0, 0);
```

Enable the Array

ARLAB

void **`glEnableVertexAttribArray`**(GLuint index);

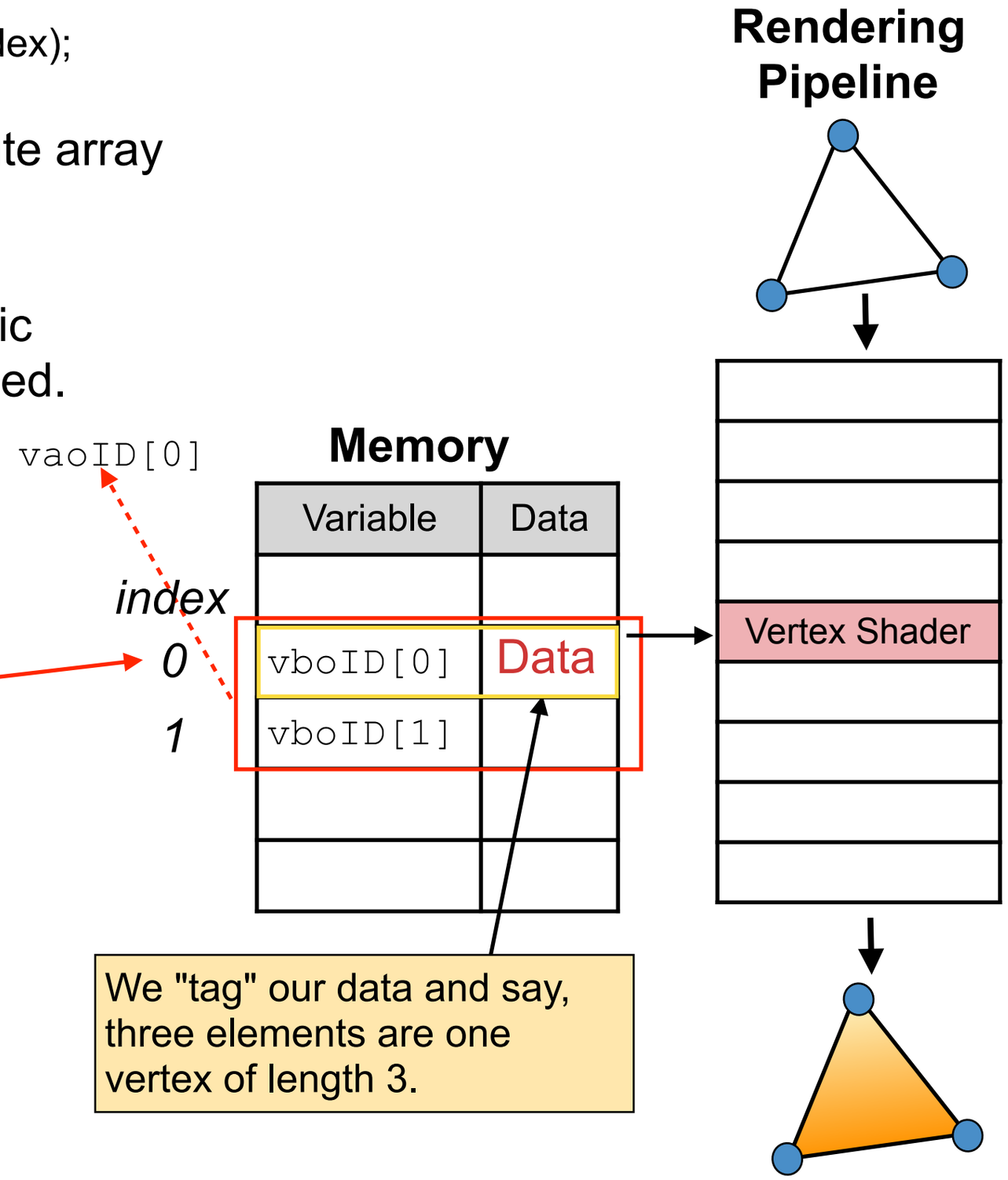
Enable or disable a generic vertex attribute array

Parameters:

- index: Specifies the index of the generic vertex attribute to be enabled or disabled.

Example:

```
glEnableVertexAttribArray(0);
```



Bind Vertex Array

ARLAB

```
int main(int argc, const char * argv[])
{
        [....]
        glGenVertexArrays(2, &vaoID[0]);
```

Program init

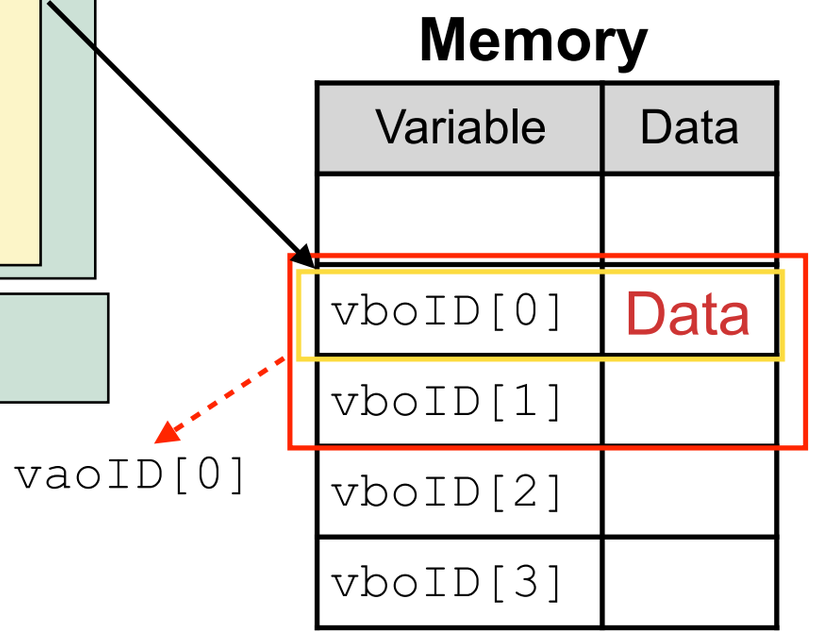
```
        glBindVertexArray(vaoID[0]); //
            glGenBuffers(2, vboID);
            glBindBuffer(GL_ARRAY_BUFFER, vboID[0]);
            glBufferData(GL_ARRAY_BUFFER, ...
            glVertexAttribPointer ( ..... );
            glEnableVertexAttribArray(0);
        glBindVertexArray(vaoID[1]); //
```

```
    while
        glBindVertexArray(vaoID[0]); //
```

main loop

All operations affect vaoID[0]

```
        [....]
}
```



Bind Vertex Array

ARLAB

```
int main(int argc, const char * argv[])
{
        [....]
        glGenVertexArrays(2, &vaoID[0]);                    1
        glBindVertexArray(vaoID[0]); //                     2
            glGenBuffers(2, vboID);                         3
            glBindBuffer(GL_ARRAY_BUFFER, vboID[0]);        4
            glBufferData(GL_ARRAY_BUFFER, ...               5
            glVertexAttribPointer ( ... );                  5
            glEnableVertexAttribArray(0);                   7
        glBindVertexArray(vaoID[1]); //
    while
        glBindVertexArray(vaoID[0]); //
            All operations affect vaoID[0]
        [....]
}
```

Program init

main loop

Sequence in our code!

1. Generate the vertex array
2. Bind one vertex array; even if you only generated one, switch it to active.
3. Generate vertex buffer object
4. Switch one object active
5. Fill it with data
6. Define the data
7. Enable the object

Color

We have to do the same procedure for our color values:

```
vertices[39] = -0.5; vertices[40] = 0.5; vertices[41] = -0.5; // Bottom left  corner
colors[39] = 0.0; colors[40] = 0.3; colors[41] = 0.0; // Bottom left corner

vertices[42] = 0.5; vertices[43] = 0.5; vertices[44] = 0.5; // Top left  corner
colors[42] = 0.0; colors[43] = 0.8; colors[44] = 0.0; // Top left corner

vertices[45] = 0.5; vertices[46] = 0.5; vertices[47] = -0.5; // Bottom left  corner
colors[45] = 0.0; colors[46] = 0.5; colors[47] = 0.0; // Bottom left corner
```

Vertex and color value definition

```
glGenVertexArrays(2, &vaoID[0]); // Create our Vertex Array Object
glBindVertexArray(vaoID[0]); // Bind our Vertex Array Object so we can use it
```

Vertex array definition

```
glGenBuffers(2, vboID); // Generate our Vertex Buffer Object

// vertices
glBindBuffer(GL_ARRAY_BUFFER, vboID[0]); // Bind our Vertex Buffer Object
glBufferData(GL_ARRAY_BUFFER, 48 * sizeof(GLfloat), vertices, GL_STATIC_DRAW); // Set the                ur VBO and se

glVertexAttribPointer((GLuint)0, 3, GL_FLOAT, GL_FALSE, 0, 0); // Set up our vertex attributes pointer
glEnableVertexAttribArray(0); // Disable our Vertex Array Object
```

VBO for the vertices

```
//Color
glBindBuffer(GL_ARRAY_BUFFER, vboID[1]); // Bind our second Vertex Buffer Object
glBufferData(GL_ARRAY_BUFFER, 48 * sizeof(GLfloat), colors, GL_STATIC_DRAW); // Set the size and data of our VBO and set

glVertexAttribPointer((GLuint)1, 3, GL_FLOAT, GL_FALSE, 0, 0); // Set up our vertex attrib
glEnableVertexAttribArray(1); // Enable the second vertex attribute array

glBindVertexArray(0); // Disable our Vertex Buffer Object
```

VBO for the color values

```
delete [] vertices; // Delete our vertices from memory
```

ARLAB

Link between shader program and data

What do we have?

ARLAB

CPU / host computer program

```
int main(int argc, const char * argv[])
{
.....
```

We defined everything from the host program side

Graphic Memory

Variable	Data
vboID[0]	Data
vboID[1]	
vboID[2]	
vboID[3]	

vaoID[0]

GPU shader program

```
static const string vs_string =
    "#version 410 core                                                 \n"
    "                                                                   \n"
    "uniform mat4 projectionMatrix;                                    \n"
    "uniform mat4 viewMatrix;                                          \n"
    "uniform mat4 modelMatrix;                                         \n"
    "in vec3 in_Position;                                              \n"
    "                                                                  \n"
    "in vec3 in_Color;                                                 \n"
    "out vec3 pass_Color;                                              \n"
```

What do we have?

ARLAB

CPU / host computer program

```
int main(int argc, const char * argv[])
{
.....
```

We defined everything from the host program side

Graphic Memory

Variable	Data
vboID[0]	Data
vboID[1]	
vboID[2]	
vboID[3]	

vaoID[0]

GPU shader program

```
static const string vs_string =
    "#version 410 core
    "
    "uniform mat4 projectionMatrix;
    "uniform mat4 viewMatrix;
    "uniform mat4 modelMatrix;
    "in vec3 in_Position;
    "
    "in vec3 in_Color;
    "out vec3 pass_Color;
```

```
\n"
 \n"
\n"
 \n"
 \n"
 \n"
 \n"
 \n"
 \n"
```

Now we have to give our GPU program access to the data

Assign a GPU name to your data

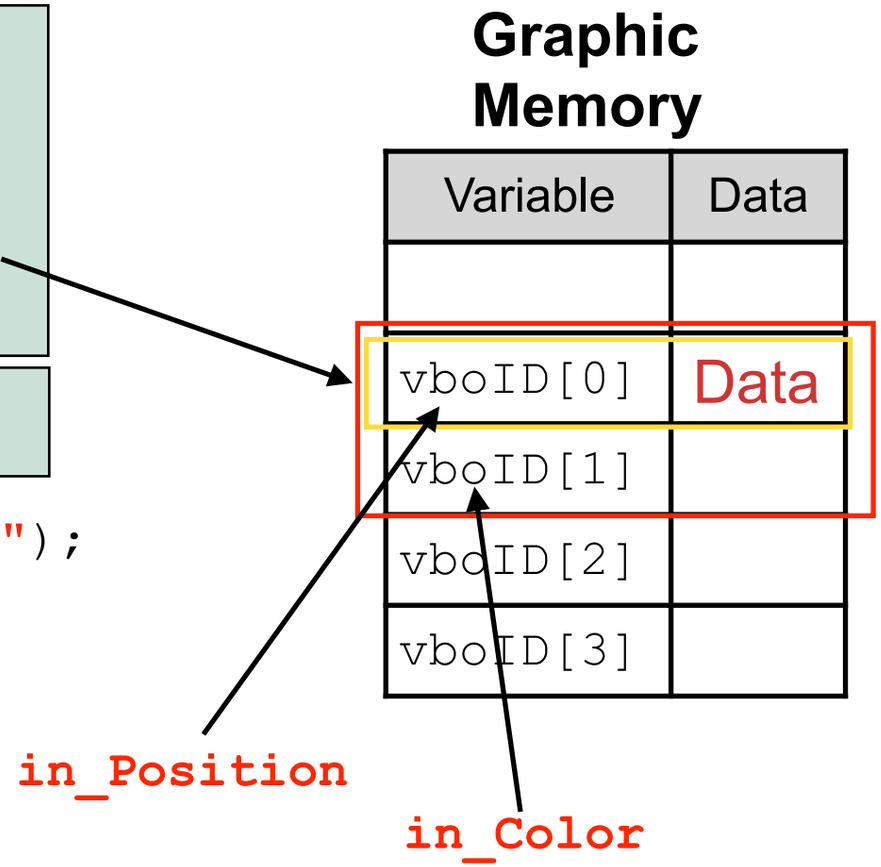
```
int main(int argc, const char * argv[])
{
        [....]
        glGenVertexArrays(2, &vaoID[0]);
        glBindVertexArray(vaoID[0]); //
            glGenBuffers(2, vboID);
        glBindVertexArray(vaoID[1]); //
        glBindAttribLocation(program, 0, "in_Position");
        glBindAttribLocation(program, 1, "in_Color");
        ----------------------------------------
        while
            glBindVertexArray(vaoID[0]); //
```

Program init

main loop

All operations affect vaoID[0]

```
        [....]
}
```



We label our data with variable names that we can use in our shader program. This must be strings.

Connect data and shader program

ARLAB

```
void glBindAttribLocation( GLuint program, GLuint index, const GLchar *name);
```

associate a generic vertex attribute index with a named attribute variable

Parameters:

- program: Specifies the handle of the program object in which the association is to be made.
- index: Specifies the index of the generic vertex attribute to be bound.
- name: Specifies a null terminated string containing the name of the vertex shader attribute variable to which index is to be bound.

Example:

```
glBindAttribLocation(program, 0, "in_Position");
glBindAttribLocation(program, 1, "in_Color");
```

Shader Program

ARLAB

```
static const string vs_string =
    "#version 410 core                                                  \n"
    "                                                                   \n"
    "uniform mat4 projectionMatrix;                                     n"
    "uniform mat4 viewMatrix;                                           \n"
    "uniform mat4 modelMatrix;                                          \n"
    "in vec3 in_Position;                                               \n"
    "                                                                   \n"
    "in vec3 in_Color;                                                  \n"
    "out vec3 pass_Color;                                               \n"
    "                                                                   \n"
    "void main(void)                                                    \n"
    "{                                                                  \n"
    "    gl_Position = projectionMatrix * viewMatrix * modelMatrix *
                  vec4(in_Position, 1.0);  \n"
    "    pass_Color = in_Color;                                         \n"
    "}                                                                  \n";
```

Keep in mind, we have to know that this is a vector of length 3

`glVertexAttribPointer ( ..... );`

We can now access our position data and our color data in our shader program. We use the specifier **in** to define the data.

Same for the fragment shader program!

ARLAB

Rendering

Prerequisites

ARLAB

We assume, our shader program is already available and prepared to work on the GPU

```
GLuint program = glCreateProgram();

GLuint fs = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fs, 1, &fs_source, NULL);
glCompileShader(fs);

GLuint vs = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vs, 1, &vs_source, NULL);
glCompileShader(vs);

// We'll attach our two compiled shaders to the OpenGL program.
glAttachShader(program, vs);
glAttachShader(program, fs);
```

See class notes: 04_ME557_GPU_Programming_Introduction

Program Structure

ARLAB

```
int main(int argc, const char * argv[])
{
        [....]
        glGenVertexArrays(2, &vaoID[0]);
```

Program init

```
glBindVertexArray(vaoID[0]); //
    glGenBuffers(2, vboID);
    glBindBuffer(GL_ARRAY_BUFFER, vboID[0]);
    glBufferData(GL_ARRAY_BUFFER, ...
    glVertexAttribPointer ( ..... );
    glEnableVertexAttribArray(0);
glBindVertexArray(vaoID[1]); //
```

Memory

Variable	Data
vboID[0]	Data
vboID[1]	
vboID[2]	
vboID[3]	

vaoID[0]

```
while
```

Clear the window

main loop

```
glBindVertexArray(vaoID[0]); //
```

All operations affect vaoID[0]

We are finally here!

Swap buffers

```
}
```

Draw a Primitive

ARLAB

```
void glDrawArrays( GLenum mode, GLint first, GLsizei count);
```

Draw an OpenGL primitive

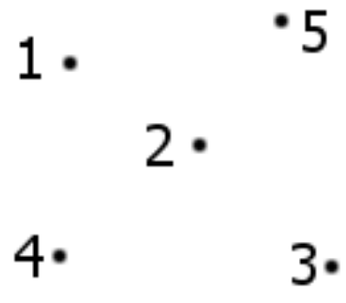
Parameters:

- mode: Specifies what kind of primitives to render. Symbolic constants GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_LINE_STRIP_ADJACENCY, GL_LINES_ADJACENCY, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES, GL_TRIANGLE_STRIP_ADJACENCY and GL_TRIANGLES_ADJACENCY are accepted.
- first: Specifies the starting index in the enabled arrays.
- count: Specifies the number of indices to be rendered.

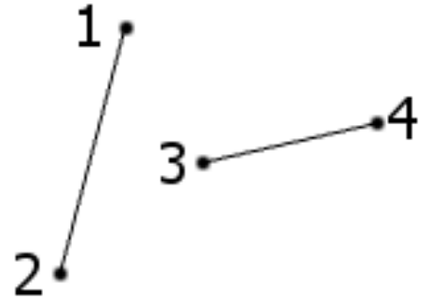
OpenGL Primitives

ARLAB

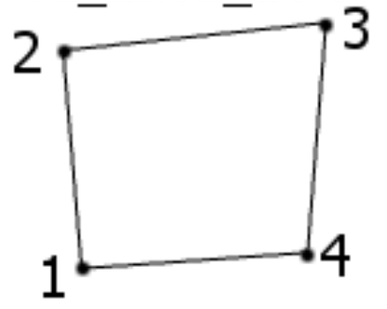
GL_POINTS



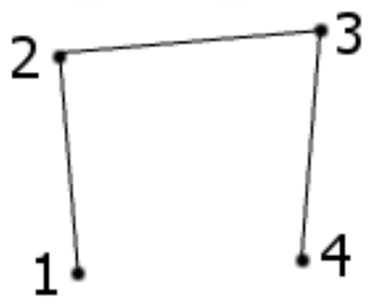
GL_LINES



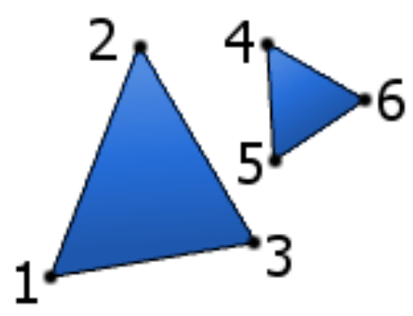
GL_LINE_LOOP



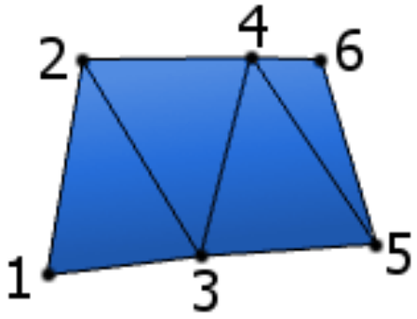
GL_LINE_STRIP



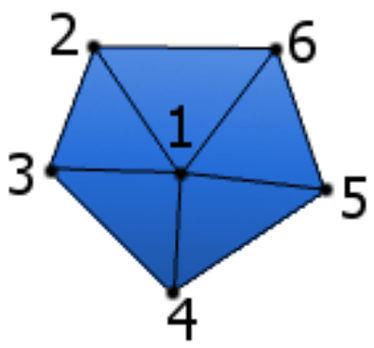
GL_TRIANGLES



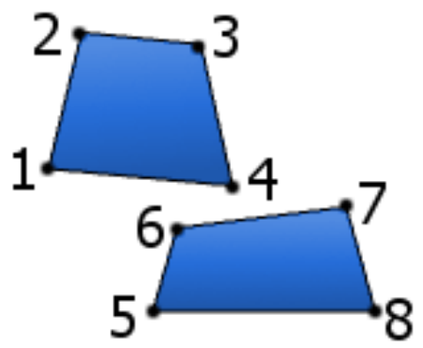
GL_TRIANGLE_STRIP



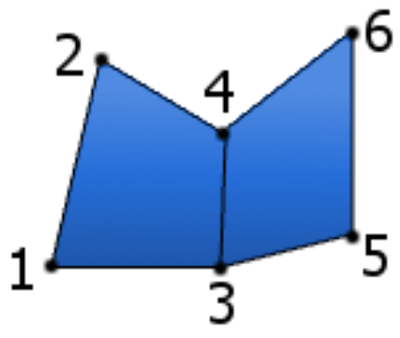
GL_TRIANGLE_FAN



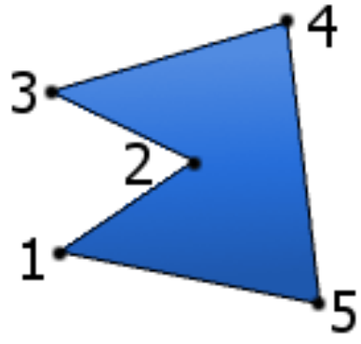
GL_QUADS



GL_QUAD_STRIP



GL_POLYGON



Program Structure

```
int main(int argc, const char * argv[])
{
    [....]

    while
    {
        Clear the window

        // Enable the shader program
        glUseProgram(program);

        glBindVertexArray(vaoID[0]);

        glDrawArrays(GL_TRIANGLE_STRIP, 0, 16);

        glBindVertexArray(0);

        // Disable the shader program
        glUseProgram(0);

        Swap buffers
    }
}
```

Program init

main loop

Memory

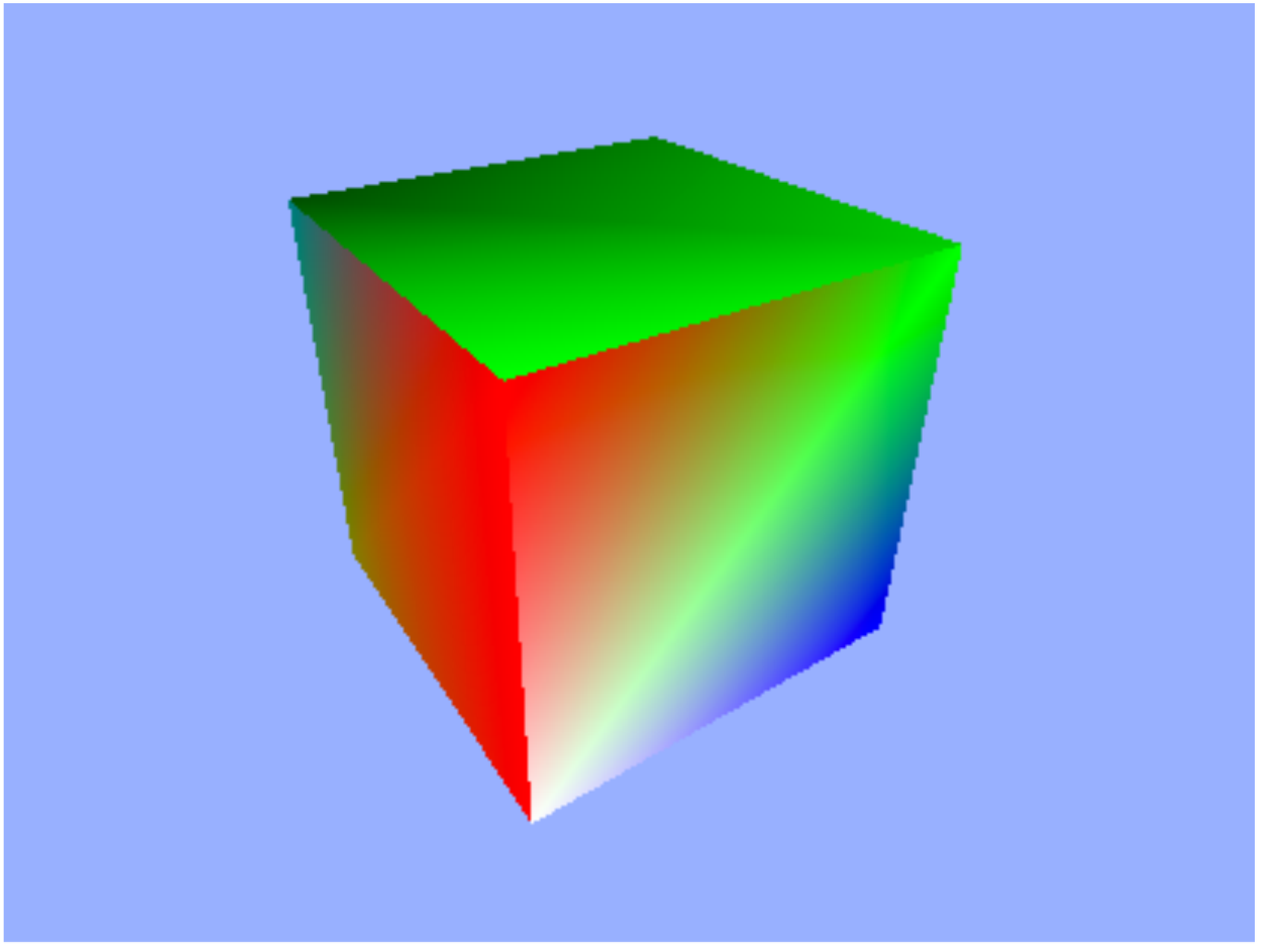
Variable	Data
vboID[0]	Data
vboID[1]	Data
vboID[2]	
vboID[3]	

vaoID[0]

The value 0 unbind the buffer / array or shader program

Simple 3D Cube

ARLAB



Program Structure

```
int main(int argc, const char * argv[])
{
    [....]

    while
    {
        Clear the window

        // Enable the shader program
        glUseProgram(program);

        glBindVertexArray(vaoID[0]);
        glDrawArrays(GL_TRIANGLE_STRIP, 0, 16);

        glBindVertexArray(vaoID[1]);
        glDrawArrays(GL_TRIANGLE, 0, 16);

        glUseProgram(0);

        Swap buffers
    }
    // Disable the shader program
    glUseProgram(0);
}
```

Program init

main loop

Memory

Variable	Data
vboID[0]	Data
vboID[1]	Data
vboID[2]	
vboID[3]	

vaoID[0]

vaoID[1]

This would switch to a new data set. Note, it must exist.

Program Structure

```
int main(int argc, const char * argv[])
{
    [....]

    while
        Clear the window
        // Enable the shader program
        glUseProgram(program);
            glBindVertexArray(vaoID[0]);
            glDrawArrays(GL_TRIANGLE_STRIP, 0, 16);
            glUseProgram(program_number2);
            glBindVertexArray(vaoID[1]);
            glDrawArrays(GL_TRIANGLE, 0, 16);
        glUseProgram(0);
        Swap buffers
                    // Disable the shader program
                    glUseProgram(0);
}
```

Program init

main loop

Memory

Variable	Data
vboID[0]	Data
vboID[1]	Data
vboID[2]	
vboID[3]	

vaoID[0]

vaoID[1]

This changes the program, but we continue to work with the same dataset

Thank you!

Questions

Rafael Radkowski, Ph.D.
Iowa State University
Virtual Reality Applications Center
1620 Howe Hall
Ames, Iowa 5001, USA

+1 515.294.5580
+1 515.294.5530(fax)

rafael@iastate.edu

ARLAB

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY