# Programming with OpenGL: PART I

Serena Morigi

A.A.2018/2019


http://www.opengl.org

# What is OpenGL

"A software interface to graphics hardware"

# Why OpenGL?

Use graphics hardware, via OpenGL or DirectX

- OpenGL is multi-platform,

  Regularly released by the Khronos Group

- DirectX is MS only

# What Is OpenGL?

- OpenGL (for “Open Graphics Library”) is a multi-platform graphics API.
- OpenGL is a computer graphics *rendering* API
  - With it, you can generate high-quality color images by rendering with geometric and image primitives
  - It forms the basis of many interactive applications that include 3D graphics
  - By using OpenGL, the graphics part of your application can be
    - operating system independent
    - window system independent

# API (Application Programming Interface)

For each windowing system there's a *binding library* that lets interfaces between OpenGL and the native windowing system.

**– GLX for Xwindow System**

**– AGL for Apple Macintosh**

**– WGL for Microsoft Windows**

**- EGL for OpenGL ES on mobile and embedded devices**

**GLU (OpenGL Utility Library)**

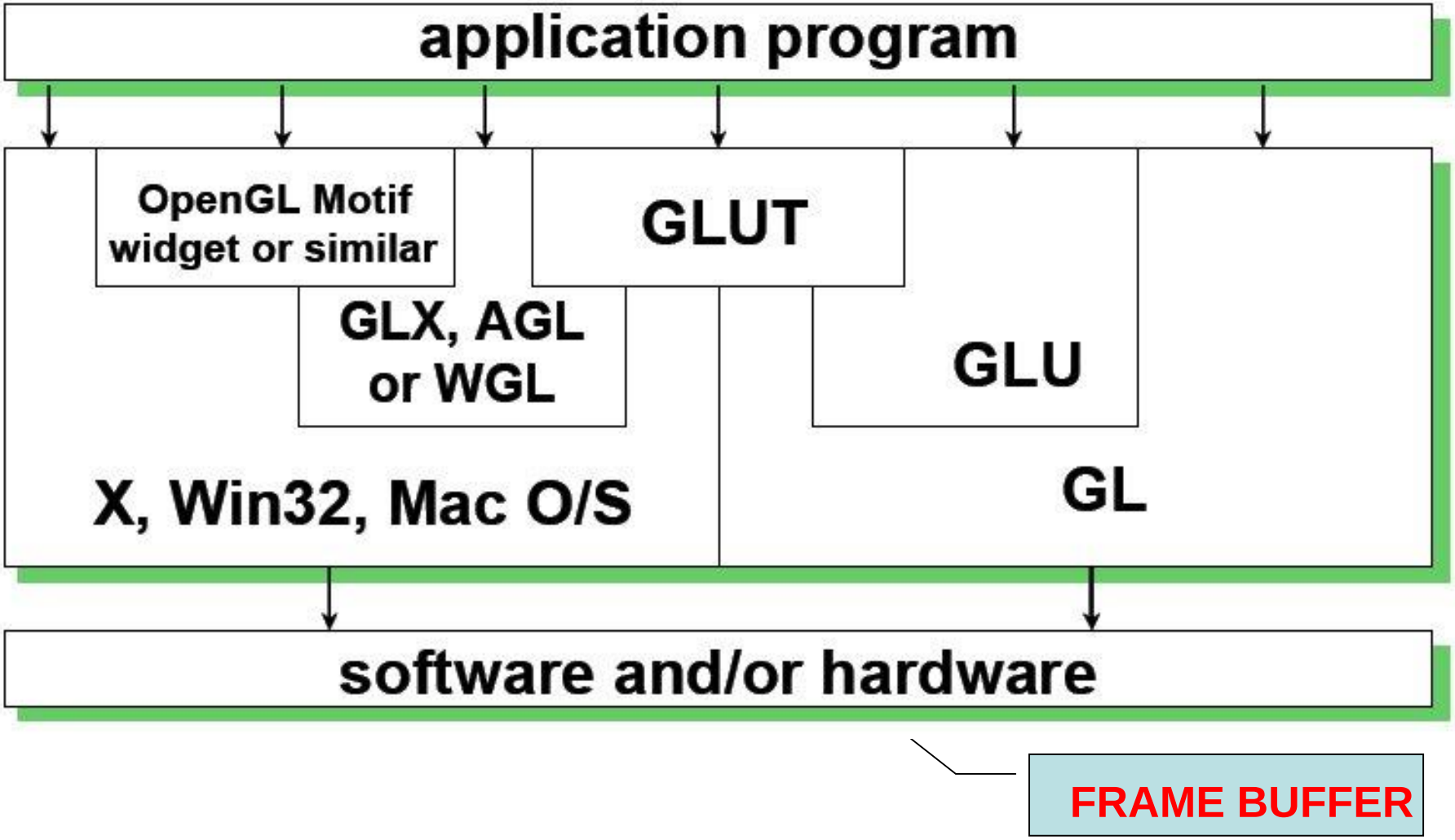
**-** Library embedded in OpenGL, utility functions and other graphics primitives (curves and surfaces NURBS)

**GLUT (OpenGL Utility Toolkit)** (more specifically, **Freeglut**)

**-** Simple open source library that will help us in creating windows, dealing with user input and input devices, and other window-system activities. **(windowing system independent interface)**

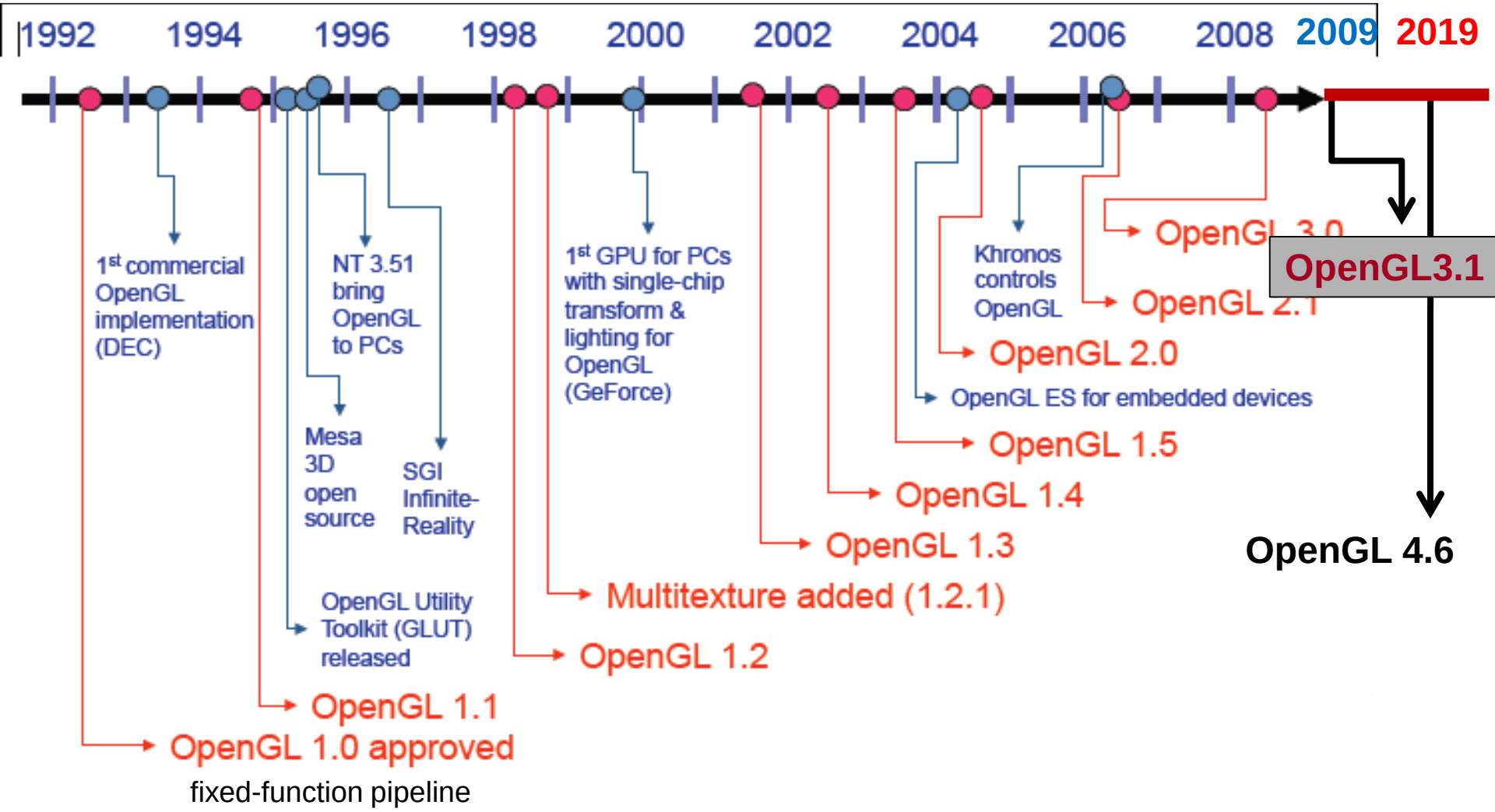
**GLEW, (OpenGL Extension Wrangler library)**

- Open-source library which removes all the complexity of accessing OpenGL functions, and working with OpenGL extensions.


application program
OpenGL Motif widget or similar
GLUT
GLX, AGL or WGL
GLU
X, Win32, Mac O/S
GL
software and/or hardware
FRAME BUFFER

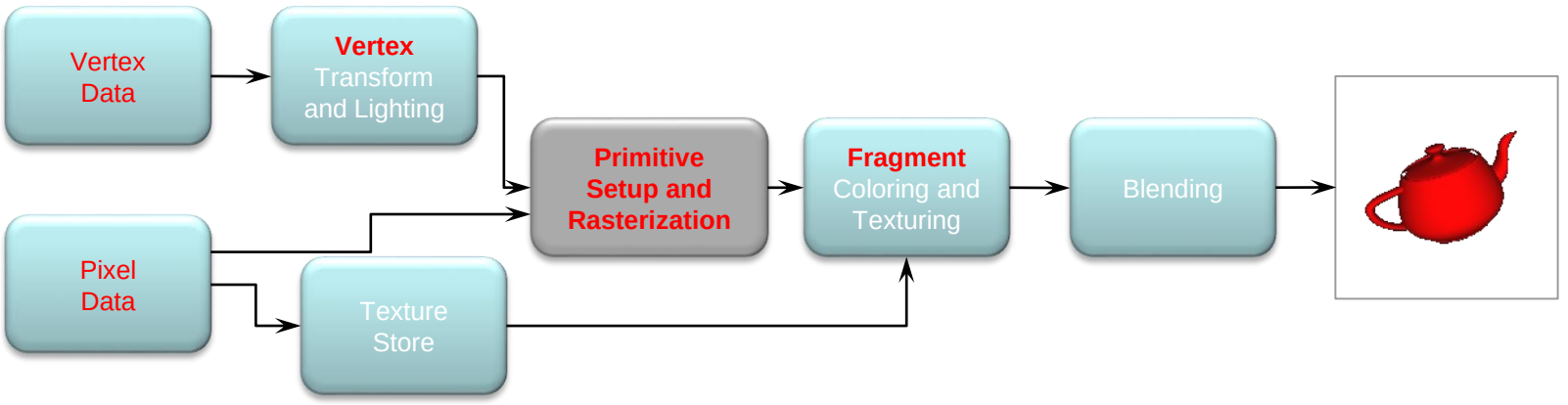
# The Evolution of the OpenGL Pipeline

# Timeline of OpenGL's Development



# In the Beginning …

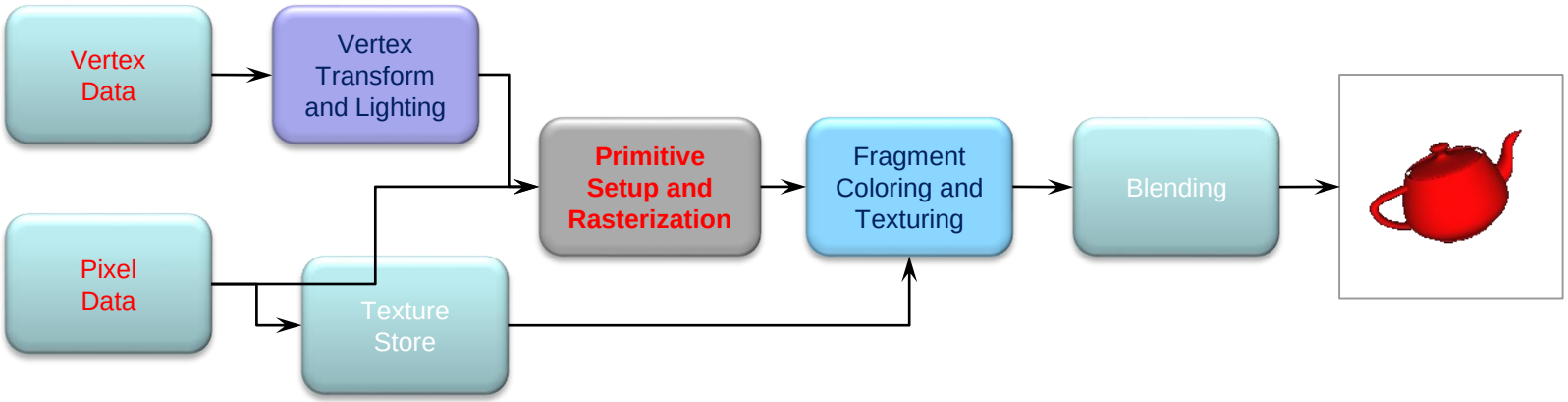
- OpenGL 1.0 was released on July 1st, 1994
- Its pipeline was entirely *fixed-function*
  - the only operations available were fixed by the implementation




- The pipeline evolved, but remained fixed-function through OpenGL versions 1.1 through 2.0 (Sept. 2004)

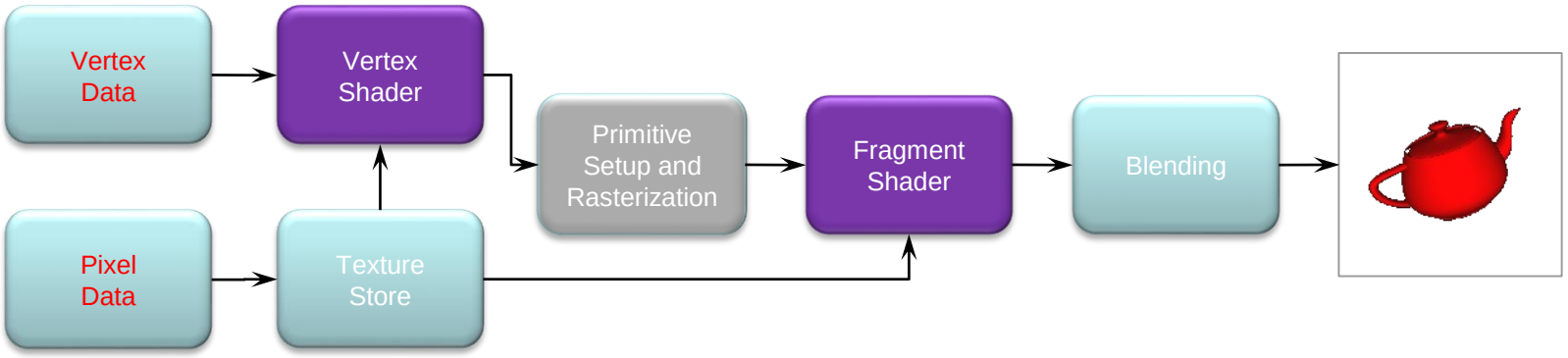
# The Start of the Programmable Pipeline

- OpenGL 2.0 (officially) added programmable shaders
  - *vertex shading* augmented the fixed-function transform and lighting stage
  - *fragment shading* augmented the fragment coloring stage
- However, the fixed-function pipeline was still available



# The Exclusively Programmable Pipeline

- OpenGL 3.1 removed the fixed-function pipeline
  - programs were required to use only shaders

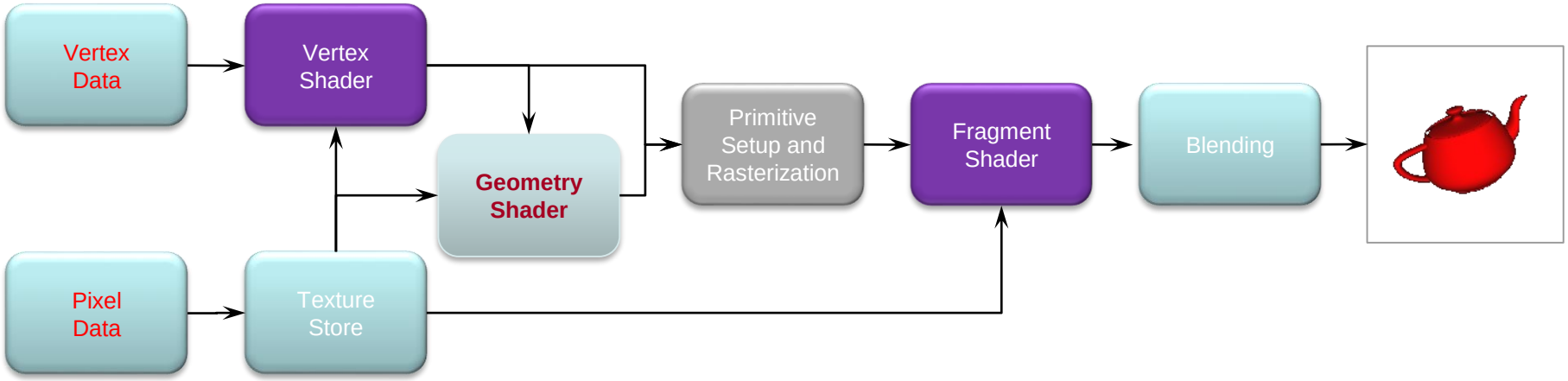



- Almost all data is *GPU-resident*
  - all vertex data sent using buffer objects
- Introduced an OpenGL extensions,
  GL_ARB_compatibility

# More Programability

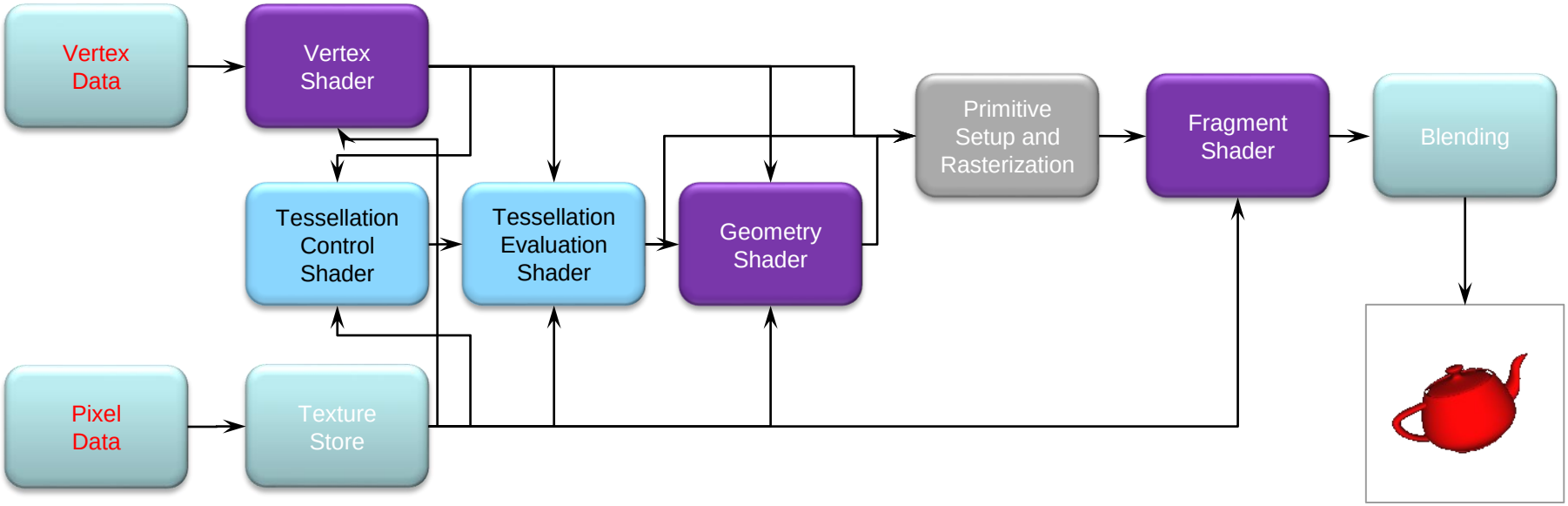
OpenGL 3.2 (released August 3rd, 2009) added an additional shading stage – *geometry shaders*

- modify geometric primitives within the graphics pipeline



# The Latest Pipelines

- OpenGL 4.1 (released July 25th, 2010) included additional shading stages – *tessellation-control and tessellation-evaluation shaders*
- Latest version is 4.6 (July 31, 2017)



# OpenGL ES , Vulkan and WebGL

- OpenGL ES 3.2
  - Designed for embedded and hand-held devices such as cell phones
  - Based on OpenGL 3.1
  - Shader based
- Vulkan, the "Next Generation OpenGL Initiative" (glNext), effort to unify OpenGL and OpenGL ES into one common API
- WebGL (Web-based Graphics Library)
  - JavaScript implementation of ES 2.0
  - 3D graphics for the browsers web

# Two OpenGL approaches..

- Fixed-Function Pipeline

  ok with OpenGL < 3.2

  OpenGL3.0 retains backward compatibility.

  With OpenGL > 3.2 **, use the "Compatibility" profile since the "Core" profile** does not support any deprecated features

- Programmable Pipeline

  pixel and fragment shaders,

  needs OpenGL >= 2.0

  OpenGL Shading Language (GLSL)

# OpenGL Application Development

# Application Framework Requirements: freeGLUT and GLEW

- OpenGL applications need a place to render into
  - usually an on-screen window
- Need to communicate with native windowing system (Each windowing system interface is different)
- We use **GLUT** (more specifically, freeglut)
  - simple, open-source library that works everywhere
  - handles all windowing operations:
    - opening windows
    - input processing

# freeGLUT
# (openGL Utility Toolkit)

A window system independent toolkit for writing OpenGL programs.

The freeGLUT source code distribution is portable to nearly all OpenGL implementations and platforms. The current version is 3.0

The toolkit supports:

- Multiple windows for OpenGL rendering
- Callback driven event processing
- Sophisticated input devices
- An 'idle' routine and timers
- A simple, cascading pop-up menu facility
- Utility routines to generate various solid and wire frame objects
- Support for bitmap and stroke fonts
- Miscellaneous window management functions

freeglut updates GLUT

# GLEW: OpenGL Extension Wrangler Library
## Simplifying Working with OpenGL

- OS deal with library functions differently
- Additionally, OpenGL has many versions and profiles which expose different sets of functions
  - managing function access is cumbersome, and window-system dependent
- We use a cross-platform open-source library, GLEW, to hide those details http://glew.sourceforge.net
- **GLEW** provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform.

# OpenGL #include

- OpenGL #include <GL/gl.h>
  - the “core” library that is platform independent
- GLU #include <GL/glu.h>
  - an auxiliary library that handles a variety of graphics accessory functions
- GLUT #include <GL/freeglut.h>
  - an auxiliary library that handles window creation, OS system calls (mouse buttons, movement, keyboard, etc), callbacks
  - Note **`#include <GL/freeglut.h>`** should automatically include the others

# Windows OS using OpenGL/FreeGLUT/GLEW with Visual Studio 2017

**Option 1) dummy**

New Project

Strumenti

Gestione Pacchetti NuGet

Console di Gestion Pacchetti

>> Install-Package nupengl.core

**Option 2) advanced**

**-** Downloading freeGLUT and GLEW

- Creating a Visual Studio Project

- Installing freeGLUT and GLEW on a Project

# 1) Windows using OpenGL/FreeGLUT/GLEW with Visual Studio 2017

In Visual Studio a Solution is a set of projects (programs).

To create a new project:

1. Click on the "New Project".
2. Select "Visual C++".
3. Select the category “Empty Project".
4. Specify the **nameProject** and **directory** of the project.
5. Click "Ok"
6. Copy your **file.cpp** in the **nameProject** folder created.
7. menu '"Project - Add Existing Item ...") and select **file.cpp**
8. **menu “Tools – NuGet Packages – Console ”**

   >> Install-Package nupengl.core

   >> Install Package glm % libreria matematica

Compile (Ctrl-F7) and run (Ctrl-F5) your program.

# 2) Windows using OpenGL/FreeGLUT/GLEW with Visual Studio 2017

Run or write an opengl program:

First thing you need to do when the project opens up is to click on the "**Project**" menu item from the top.

Select "**Properties**" (a window will come up)

Select **Configuration/All Configuration**

On the left side of the window there are tabs

**C/C++ /General** → Additional Include Directories

Select folder for include directory for freeglut/include

**Linker/Input** → Additional Dependencies

Select folder for library opengl32.lib;glu32.lib;freeglut32.lib

**Linker/General** → Additional Library Directories

Select folder for include directory for freeglut/lib

"OK" and that will include all the opengl libraries that you need

Now all you need to do is include <gl\freeglut.h> and you are ready to go

# Program Structure

- Event Driven Programming
- Most OpenGL programs have a similar structure that consists of the following functions
  - **`main()`**:
    - opens one or more windows with the required properties
    - specifies the callback functions
    - enters event loop (last executable statement)
  - **`init()`**: sets the state variables
    - Viewing
    - Attributes
  - **`initShader():`** read, compile and link shaders
  - callbacks
    - Display function
    - Input and window functions

# main.c

```
#include <GL/freeglut.h>

int main(int argc, char** argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_SINGLE|GLUT_RGBA|GLUT_DEPTH);
   glutInitWindowSize(500,500);
   glutInitWindowPosition(0,0);
   glutCreateWindow("simple");

   glutDisplayFunc(mydisplay);
   glutReshapeFunc( resize );
   glutKeyboardFunc( keyb );
   init();
   glutMainLoop();
}
```

includes **gl.h**

specify window properties

display callback

set OpenGL state (and initialize shaders)

enter event loop

# GLUT functions

- **`glutInit`** allows application to get command line arguments and initializes system
- **`gluInitDisplayMode`** requests properties for the window (the *rendering context*)
  - RGB color
  - Single buffering
  - Properties logically ORed together
- **`glutWindowSize`** in pixels
- **`glutWindowPosition`** from top-left corner of display
- **`glutCreateWindow`** create window with title “simple”
- **`glutDisplayFunc`** display callback
- **`glutMainLoop`** enter infinite event loop

# Init()

Initialization of the states that will be used throughout the application execution

```
void init( void ) {
  glClearColor( 0.0, 0.0, 0.0, 1.0 );
  glClearDepth( 1.0 );
  glEnable( GL_LIGHT0 );
  glEnable( GL_LIGHTING );
  glEnable( GL_DEPTH_TEST );
 }
```

# Rendering Callback

```
glutDisplayFunc( display );

void display( void )
{
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

glDrawArrays( GL_TRIANGLES, 0, NumVertices );

glutSwapBuffers();
}
```

# User Input Callback

```
glutKeyboardFunc( keyboard );

void keyboard( char key, int x, int y )
{
  switch( key ) {
  case 'q' : case 'Q' :
     exit( EXIT_SUCCESS );
  break;
  case 'r' : case 'R' :
     rotate = GL_TRUE;
  break;
  }
}
```

# Mouse callback

**glutMouseFunc(mymouse)**

```
void mymouse(GLint button, GLint
                    state, GLint x, GLint y)
```

- Handles
  - which button caused event (**GLUT_LEFT_BUTTON**, **GLUT_MIDDLE_BUTTON**, **GLUT_RIGHT_BUTTON**)
  - state of that button (**GLUT_UP**, **GLUT_DOWN**)
  - Position in window

**glutMouseFunc(), glutMotionFunc(), glutPassiveMouseFunc()**

# Events in OpenGL

| Event | Example | OpenGL Callback Function |
|---|---|---|
| Keypress | KeyDown<br>KeyUp | `glutKeyboardFunc` |
| Mouse | leftButtonDown<br>leftButtonUp | `glutMouseFunc` |
| Motion | With mouse press<br>Without | `glutMotionFunc`<br>`glutPassiveMotionFunc` |
| Window | Moving<br>Resizing | `glutReshapeFunc` |
| System | Idle<br>Timer | `glutIdleFunc`<br>`glutTimerFunc` |
| Software | What to draw | `glutDisplayFunc` |

# OpenGL Functions

- Primitives
  - Points
  - Line Segments
  - Triangles
- Attributes
- Transformations
  - Viewing
  - Modeling
- Control (GLUT)
- Input (GLUT)
- Query

# OpenGL State

- OpenGL is a state machine

- You give it orders to set the current state of any one of its internal variables, or to query for its current status
- The current state won't change until you specify otherwise
- Ex.: if you set the current color to Red, everything you draw will be painted Red until you change the color explicitly
- Each of the system's state variables has a default value

# State changing: Examples

**glPointSize( *size* );**

**glLineStipple( *repeat*, *pattern* ); *//*patterns**

**glShadeModel( *GL_SMOOTH* );**

**glEnable( *GL_LIGHTING* );**

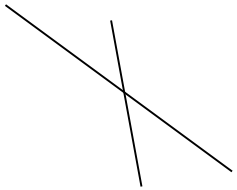
**glDisable( *GL_TEXTURE_2D* );**

# OpenGL's Geometric Primitives

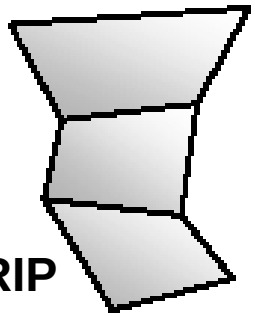
All primitives are specified by vertices

GL_POINTS

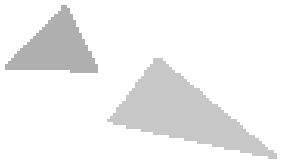
GL_LINES



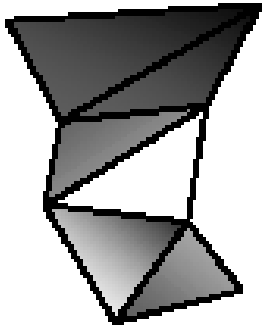
GL_QUAD_STRIP



GL_TRIANGLES



GL_TRIANGLE_STRIP



GL_TRIANGLE_FAN

GL_QUADS

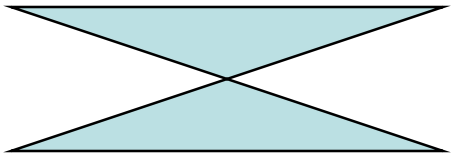
GL_LINE_STRIP

GL_LINE_LOOP

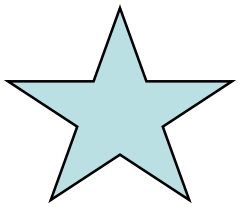
GL_POLYGON

# Polygon Issues

- OpenGL will only display triangles
  - Simple: edges cannot cross
  - Convex: All points on line segment between two points in a polygon are also in the polygon
  - Flat: all vertices are in the same plane
- Application program must tessellate a polygon into triangles (triangulation)
- OpenGL 4.1 contains a tessellator



nonsimple polygon



nonconvex polygon

# The rendering techniques are:

- 1) Immediate mode
  - Each time a vertex is specified in application, its location is sent to the GPU
  - Uses **`glVertex, glBegin/glEnd`**
  - Creates bottleneck between CPU and GPU
  - To redraw the same data needs to resend the data

- 2) Display list mode
  - Better to send array over and store on **GPU** for multiple renderings
  - Each display list can contain commands and data and is associated with  a name (id)

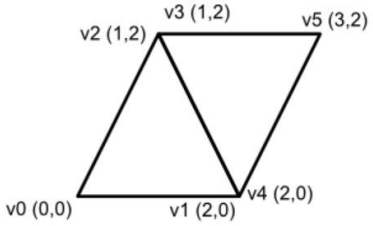
# The four rendering techniques are:

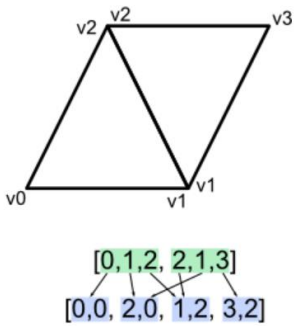
- 3) DrawArrays/DrawElements with Arrays — The data is stored in a vertex array, and each primitive is drawn using a call to *glDrawArrays*. The data has to be sent to the GPU every time a primitive is drawn.

- *4) DrawArrays/DrawElements with VBOs* — Again, *glDrawArrays* is used to draw the primitive, but this time the data is stored in a VBO instead of in an array, so the data only has to be transmitted to the **GPU** once (*Later*).

*DrawArrays (without indexing)*

*DrawElements (with indexing)*



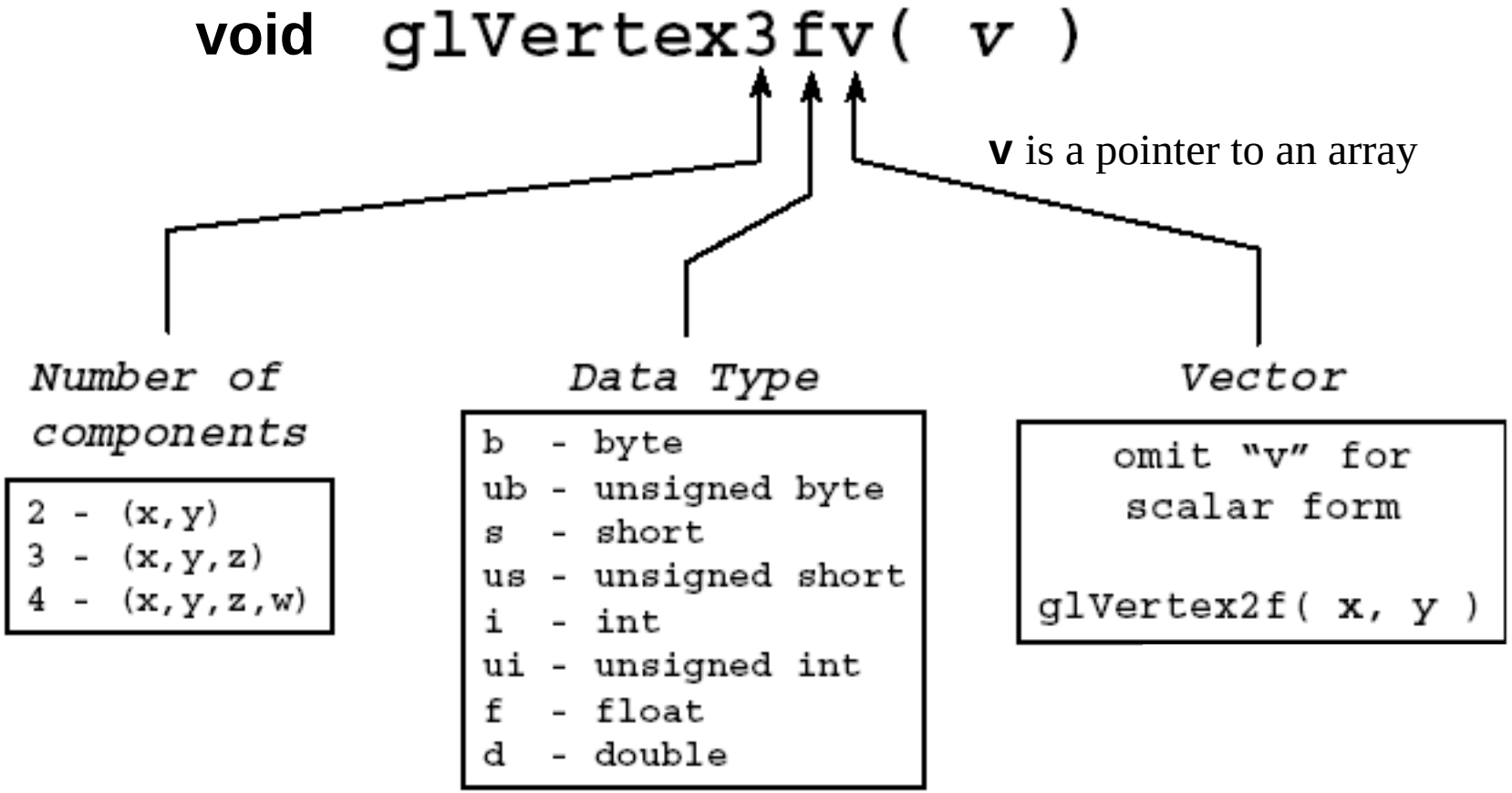



# Immediate Mode

## OpenGL function format

rtype glNAME{1234}{b s i f d ub us ui}[v]

belongs to GL library, function name, dimensions,



# Immediate Mode

## Primitive generating

```
glBegin( primType );
..
..
glEnd();
```

```
GLfloat red, greed, blue;
Glfloat coords[3];
glBegin( primType );
for ( i = 0; i < nVerts; ++i )
{
glColor3f( red, green, blue );
glVertex3fv( coords );
}
glEnd();
```

%*primType* geometric primitive type

# Display list

Display list is a group of OpenGL commands and the data used by those commands that have been stored (compiled) for later execution in GPU.

Once a display list is created, all vertex and pixel data are evaluated and copied into the display list memory on the server machine. It is only one time process.

After the display list has been prepared (compiled), you can reuse it repeatedly without re-evaluating and re-transmitting data over and over again to draw each frame.
Disadvantage : it’s STATIC, while Vertex Buffer Object can handle both static and dynamic dataset.

# Display list

- Create a display list

```
GLuint id;
void init( void )
{
        id = glGenLists( 1 );
        glNewList( id, GL_COMPILE );
        /* other OpenGL routines */
        glEndList();
}
```

- Call the created display list

```
void display( void )
{
        glCallList( id );
}
```

Replace GL_COMPILE with GL_COMPILE_AND_EXECUTE to store and visualize directly

# Example: display list

- Consider drawing a car model:
  - Create a display list for chassis
  - Create a display list for a wheel

```
glNewList( CAR, GL_COMPILE );
glCallList( CHASSIS );
glTranslatef( … );
glCallList( WHEEL );
glTranslatef( … );
glCallList( WHEEL );
…
glEndList();
```

# A First Program

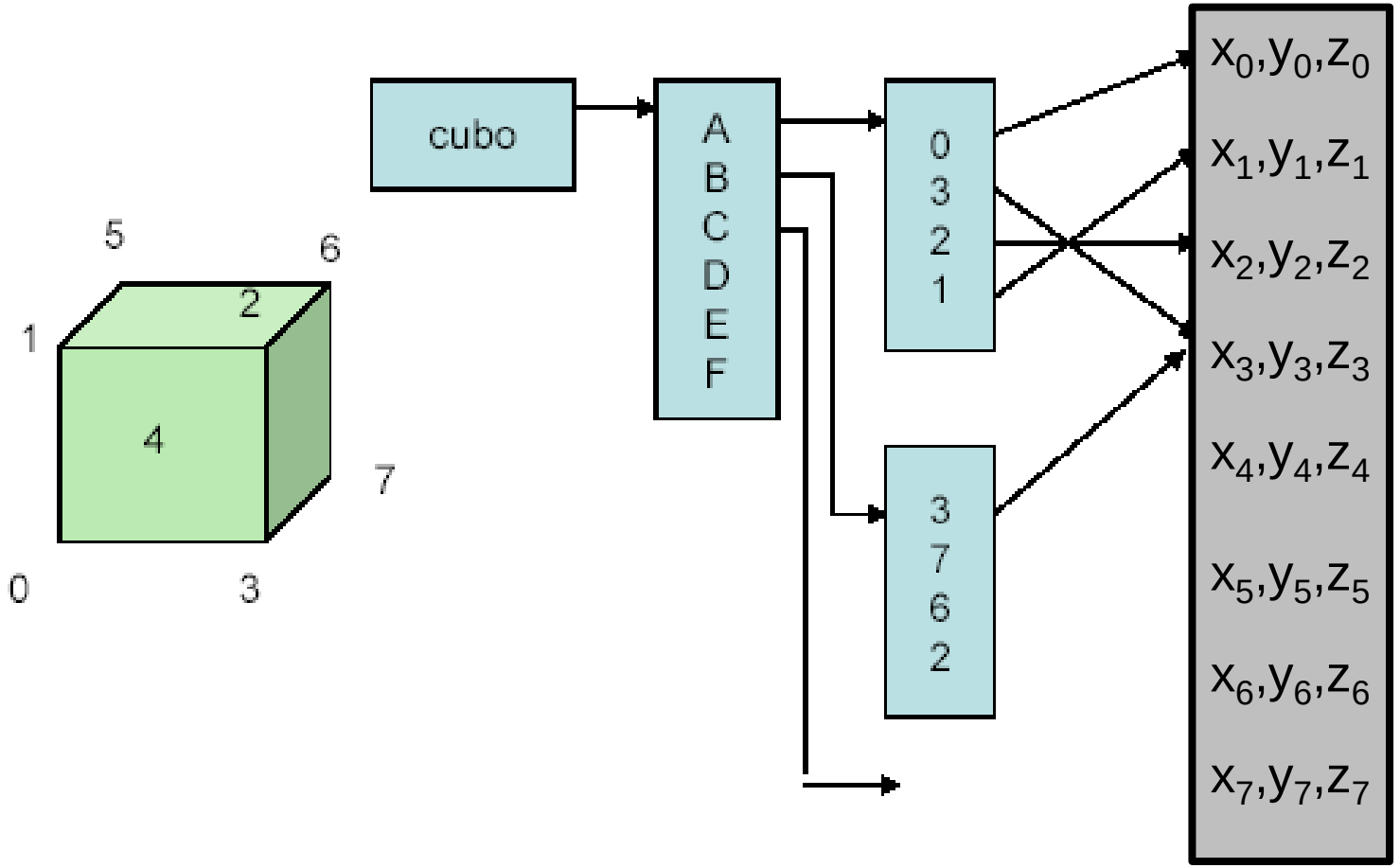
# Draw a cube

- We'll render a cube with colors at each vertex
- Our example demonstrates:
  - initializing vertex data
  - organizing data for rendering
  - simple object modeling
    - building up 3D objects from geometric primitives
    - building geometric primitives from vertices

# Draw a cube

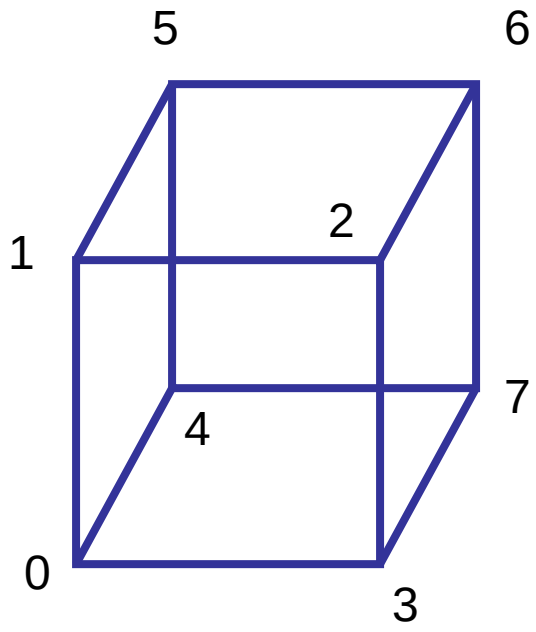
**Cube is an object with 6 faces and 8 vertices**

We'll render a cube with colors at each vertex



# 1) Draw a cube: Immediate mode

```
void colorcube( )
{
    polygon(0,3,2,1);
    polygon(2,3,7,6);
    polygon(0,4,7,3);
    polygon(1,2,6,5);
    polygon(4,5,6,7);
    polygon(0,1,5,4);
}
```




**Called by display()**

**Each polygon has 2 faces INTERNAL and EXTERNAL: how do we define it?**

**Right-Hand Rule!!**

The vertices are ordered so that we obtain correct outward facing normals

# Define vertices

```
GLfloat vertices[][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
{1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
{1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};
```

**Optional:**

```
GLfloat normals[][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
{1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
{1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};
```

```
GLfloat colors[][3] = {{0.0,0.0,0.0},{1.0,0.0,0.0},
{1.0,1.0,0.0}, {0.0,1.0,0.0}, {0.0,0.0,1.0},
{1.0,0.0,1.0}, {1.0,1.0,1.0}, {0.0,1.0,1.0}};
```

# Draw a cube

**Define faces:**

```
void polygon(int a, int b, int c , int d)
{

glBegin(GL_POLYGON);
    glVertex3fv(vertices[a]);
    glVertex3fv(vertices[b]);
    glVertex3fv(vertices[c]);
    glVertex3fv(vertices[d]);
glEnd( );
}
```

```
glBegin(GL_POLYGON);
        glColor3fv(colors[a]);
        glVertex3fv(vertices[a]);
        glColor3fv(colors[b]);
        glVertex3fv(vertices[b]);
        glColor3fv(colors[c]);
        glVertex3fv(vertices[c]);
        glColor3fv(colors[d]);
        glVertex3fv(vertices[d]);
glEnd();
```

// draw in immediate mode
// 72 calls = 6x6 glVertex*() calls + 6x6 glColor*() calls

# 3) Draw a cube: Use vertex array

- Instead you specify individual vertex data in immediate mode (between glBegin() and glEnd() pairs), you can store vertex data in a set of arrays including vertex coordinates, normals, texture coordinates and color information.
- Once that is done, the primitive can be drawn with a single call to
  - *glDrawArrays()* % reduces the number of function calls
  - *glDrawElements()* %reduces the number of function calls and redundant usage of shared vertices

# Define vertices

```
GLOBALS:
float cubeCoords[72] = {
       1,1,1,    -1,1,1,   -1,-1,1,   1,-1,1,       // face #1
       1,1,1,     1,-1,1,   1,-1,-1,  1,1,-1,       // face #2
       1,1,1,     1,1,-1,  -1,1,-1,  -1,1,1,        // face #3
       -1,-1,-1, -1,1,-1,   1,1,-1,   1,-1,-1,      // face #4
       -1,-1,-1, -1,-1,1,  -1,1,1,   -1,1,-1,       // face #5
       -1,-1,-1,  1,-1,-1,  1,-1,1,   -1,-1,1  };  // face #6

float cubeFaceColors[72] = {
        1,0,0,  1,0,0,  1,0,0,  1,0,0,       // face #1 is red
        0,1,0,  0,1,0,  0,1,0,  0,1,0,       // face #2 is green
        0,0,1,  0,0,1,  0,0,1,  0,0,1,       // face #3 is blue
        1,1,0,  1,1,0,  1,1,0,  1,1,0,       // face #4 is yellow
        0,1,1,  0,1,1,  0,1,1,  0,1,1,       // face #5 is cyan
        1,0,1,  1,0,1,  1,0,1,  1,0,1,   }; // face #6 is red
```

# Use vertex array: `glDrawArrays`

1. functions to activate and deactivate 6 different types of arrays:

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

2. specify the exact positions(addresses) of arrays

```
glVertexPointer(3, GL_FLOAT, 0, cubeCoords);
glColorPointer(3,GL_FLOAT, 0, cubeFaceColors);
```

- **glVertexPointer()**: specify pointer to vertex coords array
- **glNormalPointer()**: specify pointer to normal array
- **glColorPointer()**: specify pointer to RGB color array
- **glIndexPointer()**: specify pointer to indexed color array
- **glTexCoordPointer()**: specify pointer to texture cords array
- **glEdgeFlagPointer()**: specify pointer to edge flag array

3. draw the cube (in display() routine)

```
glDrawArrays( GL_QUADS, 0, 24 );
```

# Define vertices

```
GLOBALS:
GLfloat vertices[] = {-1.0,-1.0,-1.0,1.0,-1.0,-1.0,
    1.0,1.0,-1.0, -1.0,1.0,-1.0, -1.0,-1.0,1.0,
    1.0,-1.0,1.0, 1.0,1.0,1.0, -1.0,1.0,1.0};

GLfloat colors[] = {0.0,0.0,0.0,1.0,0.0,0.0,
    1.0,1.0,0.0, 0.0,1.0,0.0, 0.0,0.0,1.0,
    1.0,0.0,1.0, 1.0,1.0,1.0, 0.0,1.0,1.0};
Glubyte
cubeIndices[] =
    {0,3,2,1,2,3,7,6,0,4,7,3,1,2,6,5,4,5,6,7,0,1,5,4};
```

# Use vertex array: `glDrawElements`

1. functions to activate and deactivate 6 different types of arrays:

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

2. specify the exact positions(addresses) of arrays

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

3. draw the cube (in display() routine)

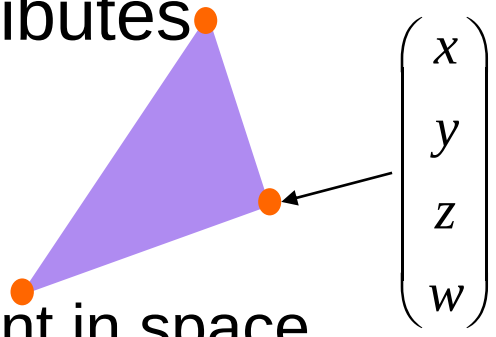
```
glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, cubeIndices);
```

# 4) Draw a cube: Use VAO and VBO

- With VBOs, we copy the whole lot into a buffer before drawing starts, and this sits on the graphics hardware memory instead.
- This is much more efficient for drawing because, although the bus between the CPU and the GPU is very wide, a bottleneck for drawing performance is created when drawing operations stall to send OpenGL commands from the CPU to the GPU.
- To avoid this we try to keep as much data and processing on graphics card's high-performance memory as we can.
- Drawing VBO using OpenGL fixed pipeline is almost identical to Vertex Array. The only difference is to specify the memory offsets where the data are stored, instead of the pointers to the arrays. For OpenGL programmable pipeline using vertex/fragment shaders, please refer to the SHADER slides.

# Vertex Buffer Object (VBO)

- A vertex is a collection of generic attributes
  - positional coordinates
  - colors
  - texture coordinates
  - any other data associated with that point in space




- Position stored in 4 dimensional homogeneous coordinates
- Vertex data must be stored in

  **vertex buffer objects** (VBOs)
- VBOs must be stored in

  **vertex array objects** (VAOs)

# Vertex Array Objects (VAOs)

- VAOs store the data of an geometric object
- Steps in using a VAO
  1) generate VAO names by calling
     `glGenVertexArrays()`
  2) sets the VAO as the active one
     `glBindVertexArray()`

  3) update VBOs associated with this VAO
  4) bind VAO for use in rendering
- This approach allows a single function call to specify all the data for an objects

# VAOs in Code

- Create a vertex array object

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

# VBO: Storing Vertex Attributes

- Vertex data must be stored in a VBO, this needs to be done only once , and then associated with a VAO,

- The code-flow is similar to configuring a VAO
  - generate VBO names by calling
    `glGenBuffers()`
  - sets a buffer as the current buffer. We use it to tell `OpenGL` that this is the buffer we are working on now.
    `glBindBuffer( GL_ARRAY_BUFFER, … )`

  - load data into VBO using

    `glBufferData( GL_ARRAY_BUFFER, … )`

- This at rendering time
  - bind VAO for use in rendering
    `glBindVertexArray()`

```
glDrawArrays( GL_QUADS, 0, NumVertices );
```

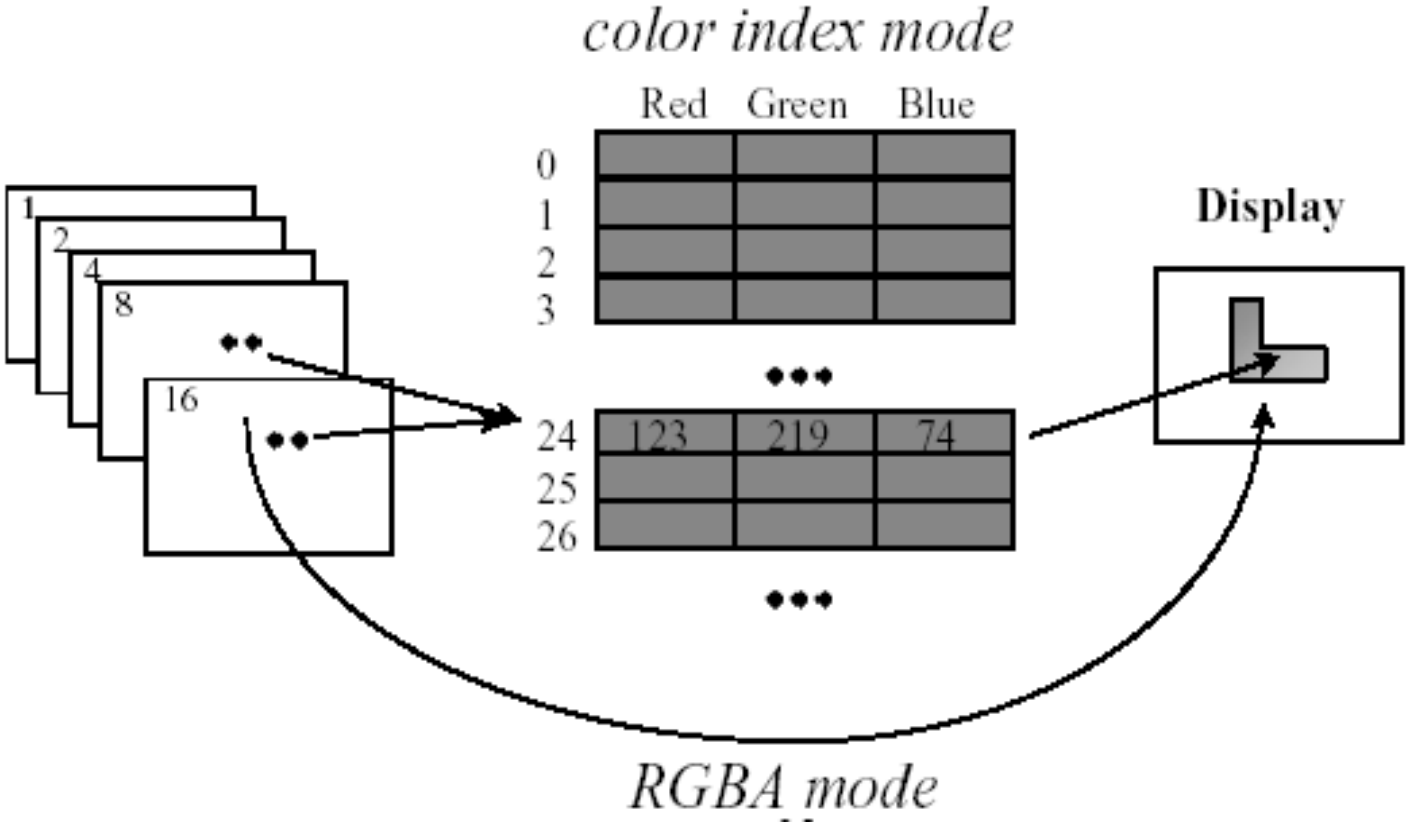
# VBOs in Code

- Create and initialize a buffer object

```
GLuint buffer;
glGenBuffers( 1, &buffer );
glBindBuffer( GL_ARRAY_BUFFER, buffer );

glBufferData( GL_ARRAY_BUFFER,
              sizeof(vPositions) + sizeof(vColors),
              NULL, GL_STATIC_DRAW );
glBufferSubData( GL_ARRAY_BUFFER, 0,
                 sizeof(vPositions), vPositions );
glBufferSubData( GL_ARRAY_BUFFER, sizeof(vPositions),
                 sizeof(vColors), vColors );
```

# RGB vs Indexed COLOR




**RGBA (TRUECOLOR) or COLOR INDEX (COLORMAP)**
**glColor*() glIndex*()**

***glutInitDisplayMode()*** **specifies a RGB window RGBA (with GLUT_RGBA), or a color indexed window (with GLUT_INDEX).**

# RGBA color in OpenGL

- Each color component (8 bits) is stored separately in the frame buffer
- Color values can range from 0.0 (none) to 1.0 (all) using floats or over the range from 0 to 255 using unsigned bytes
- Example, define the background color for a window:

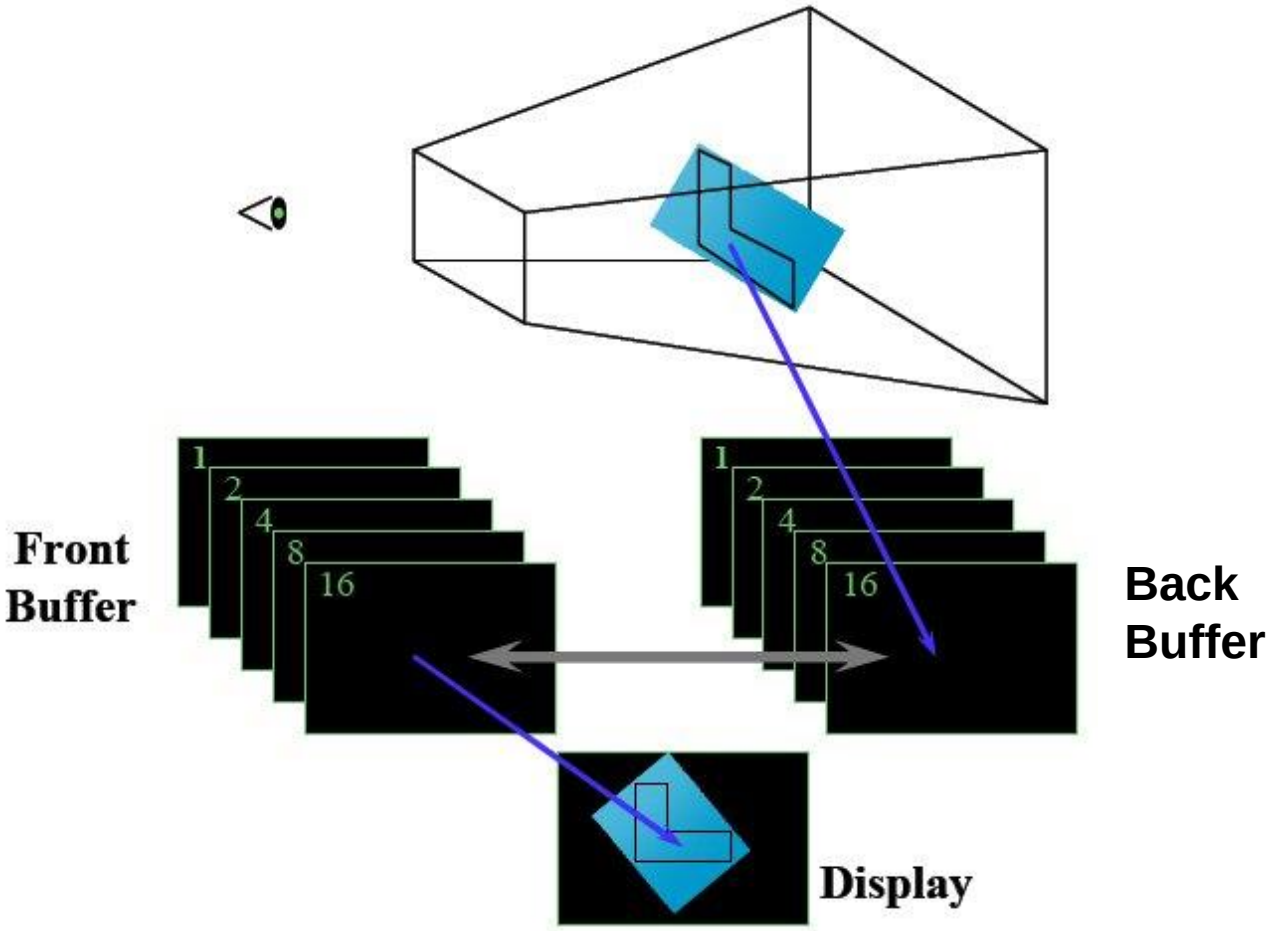
```
glClearColor(1.0,0.3,0.6,1.0)
             R   G   B   A
```

- RGBA…..Alpha Transparency values from 0 (transparent) to 1 (solid)
- GLUT_RGB and GLUT_RGBA with alpha channel
  - glColor3f (1.0, 1.0, 1.0);
  - glColor4f (1.0, 1.0, 1.0, 0.0);

# Double buffering

Front buffer and Back buffer

The application displays the contents of the front buffer while it is being made the next frame in the back buffer then an appropriate signal allows the exchange of roles of the two buffers.



# Animation using double buffering

1. Specify the use of double buffer FB

```
glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
```

2. Clear FB

```
glClear ( GL_COLOR_BUFFER_BIT );
```

3. Scene Rendering

4. The contents of the back buffer of the *current window* becomes the contents of the front buffer.

```
glutSwapBuffers( );
```

An implicit glFlush is done by glutSwapBuffers before it returns.

5. Repeat 2 - 4 to produce animation

# Idle Callback

```
glutIdleFunc( idle );

void idle( void )
{
t += dt;
glutPostRedisplay();
}
```

Trigger an automatic redraw for animation

The window's display callback will be called to redisplay the window

# References

- www.opengl.org
  - Standards documents
  - Sample code
- *OpenGL Programming Guide: The Official Guide to Learning OpenGL, (9th Edition)*
  John Kessenich, Graham Sellers, and Dave Shreiner.
- *OpenGL 4 Shading Language Cookbook: Build high-quality, real-time 3D graphics with OpenGL 4.6, GLSL 4.6 and C++17, 3rd Edition, 2018,*
  *David Wolff*
- OpenGL Shading Language.