Programming with OpenGL: PART I

http://www.opengl.org

Serena Morigi

A.A.2022/2023

An incremental Learning …… Have fun!!

What is OpenGL? API

"A software interface to graphics hardware"

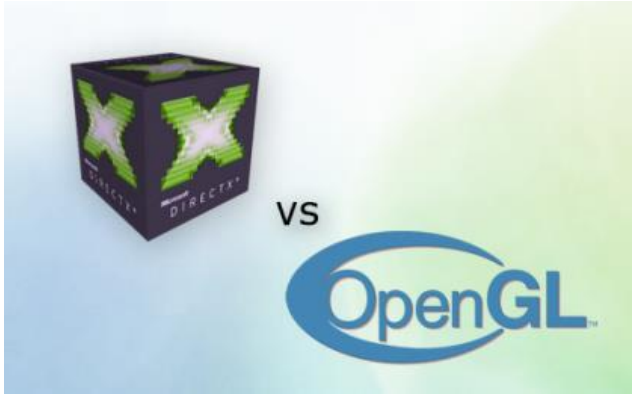
Why OpenGL?

Use graphics hardware, via OpenGL or DirectX

- OpenGL is multi-platform

  (compatible with Linux, macOS, and Windows),

- DirectX is MS only



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 (from 2016..), low-level API that provides high-efficiency, cross-platform access to GPUs

In OpenGL getting something on the screen is by far easier. Vulkan's level of verbosity to get to the first pixel on the screen is far higher. Especially for people new to graphics, it may be better to use OpenGL or rendering middleware that hides this complexity and focus on the actual task.

- WebGL (Web-based Graphics Library)
  - JavaScript implementation of ES 2.0
  - 3D graphics for the browsers web

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 (Unix-like, Linux)

– CGL for MacOS

– WGL for Microsoft Windows

– EGL for OpenGL ES on mobile and embedded devices (Android)

GLU (OpenGL Utility Library)

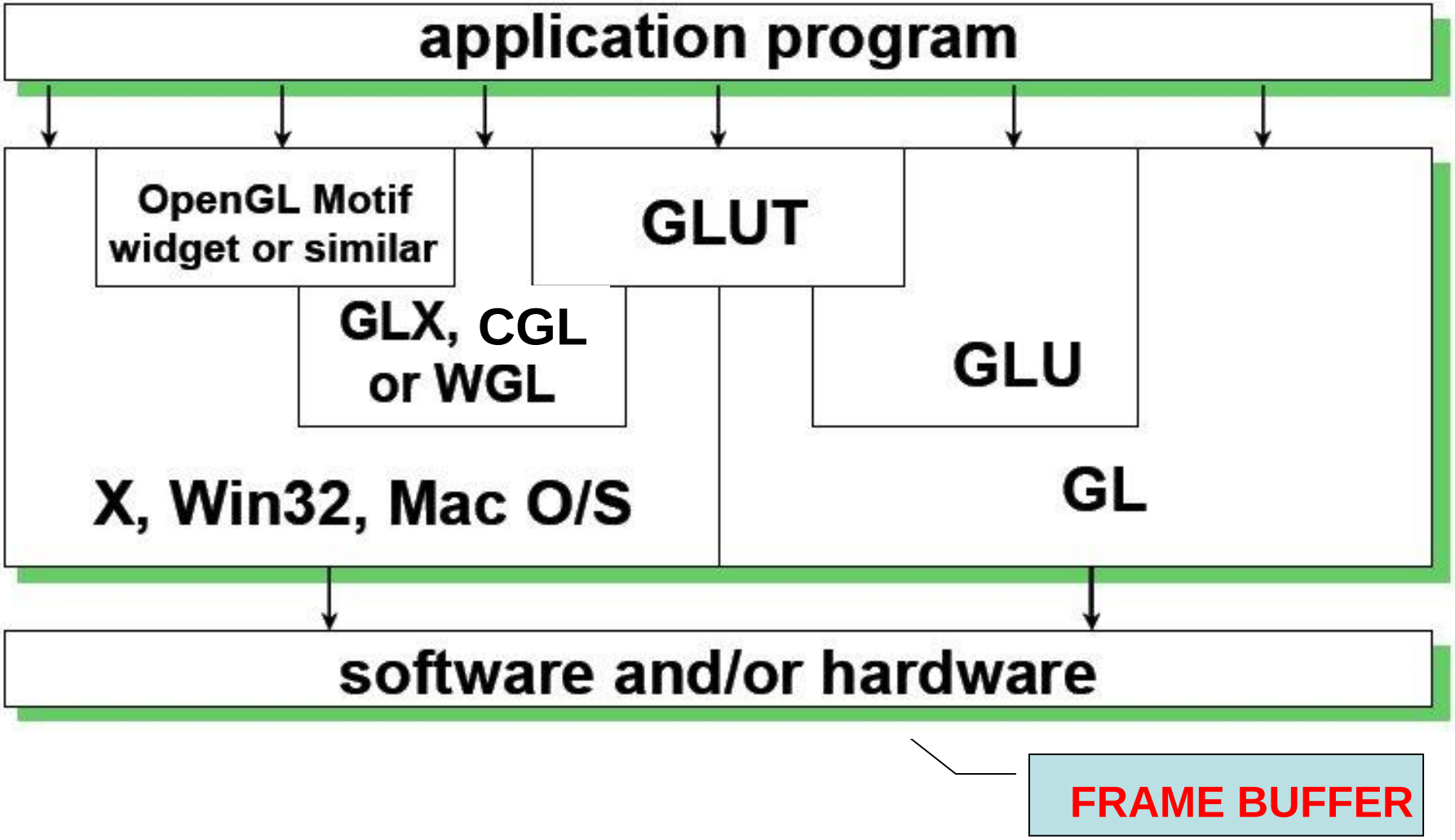
- Library embedded in OpenGL, utility functions and other graphics primitives (curves and surfaces NURBS)

GLUT (OpenGL Utility Toolkit) (alternative: **Freeglut,GLFW**)

- 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)**

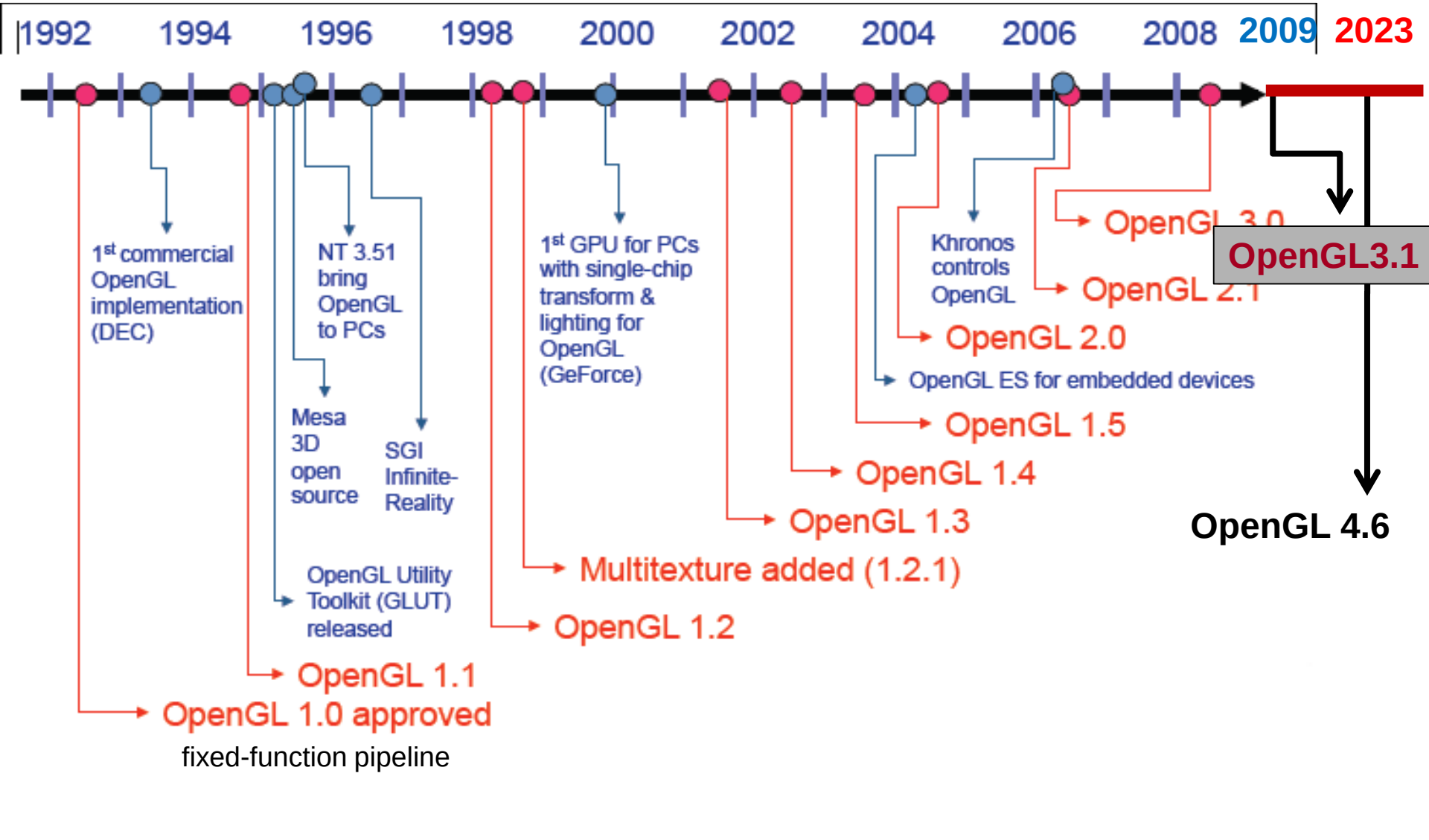
GLEW, (OpenGL Extension Wrangler library)

- 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, CGL or WGL
GLU
X, Win32, Mac O/S
GL
software and/or hardware
FRAME BUFFER

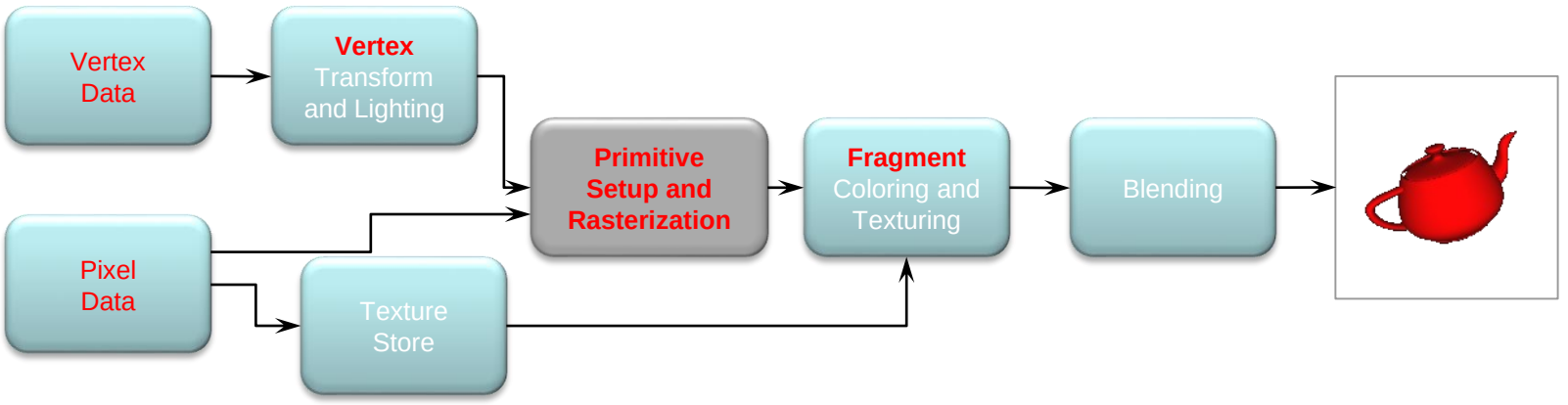
The Evolution of the OpenGL Pipeline

Timeline of OpenGL's Development



In the Beginning …

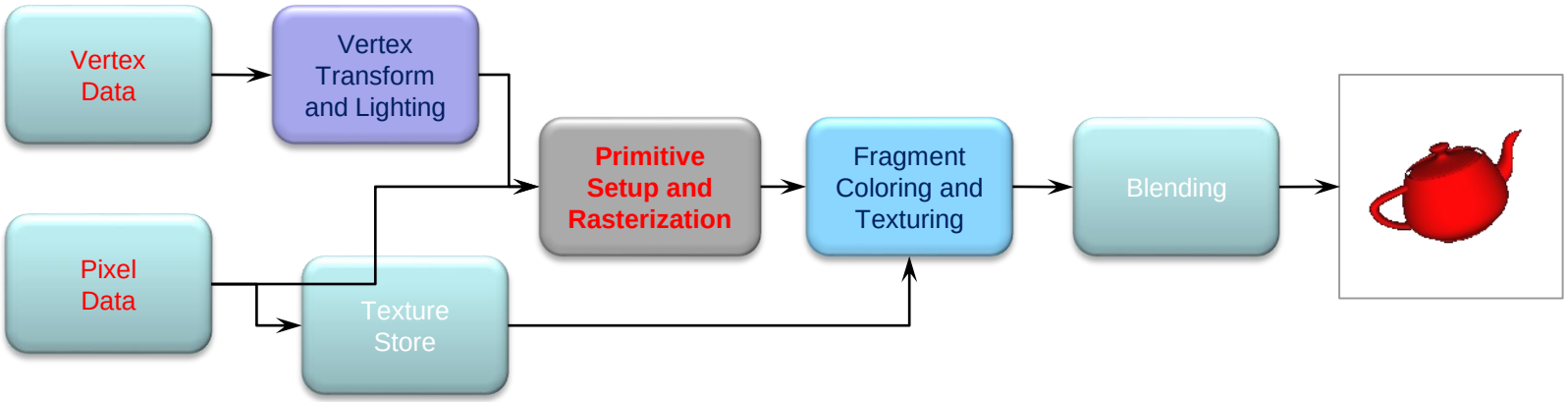
- OpenGL 1.0 was released on July 1st, 1994
- Its pipeline was entirely *fixed-function*
  - the only operations available were fixed by the implementation



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

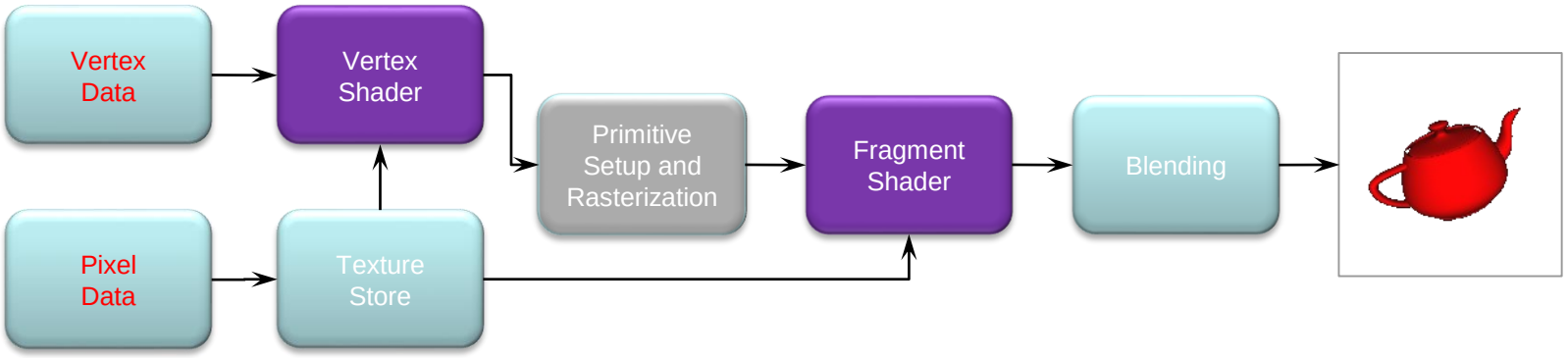
The Start of the Programmable Pipeline

- 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

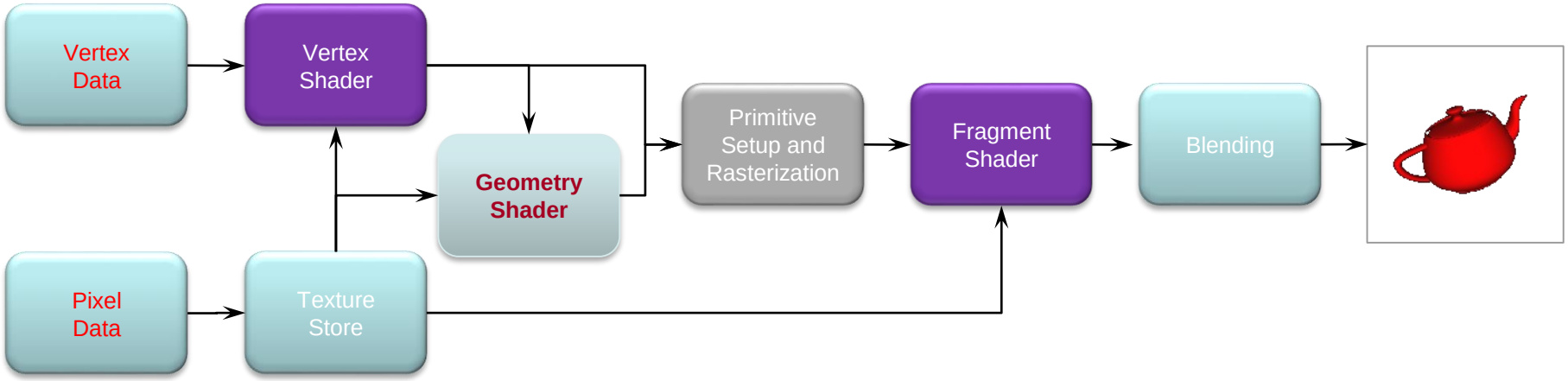


- Almost all data is *GPU-resident*
  - all vertex data sent using buffer objects
- Introduced an OpenGL extensions,
  GL_ARB_compatibility

More Programability

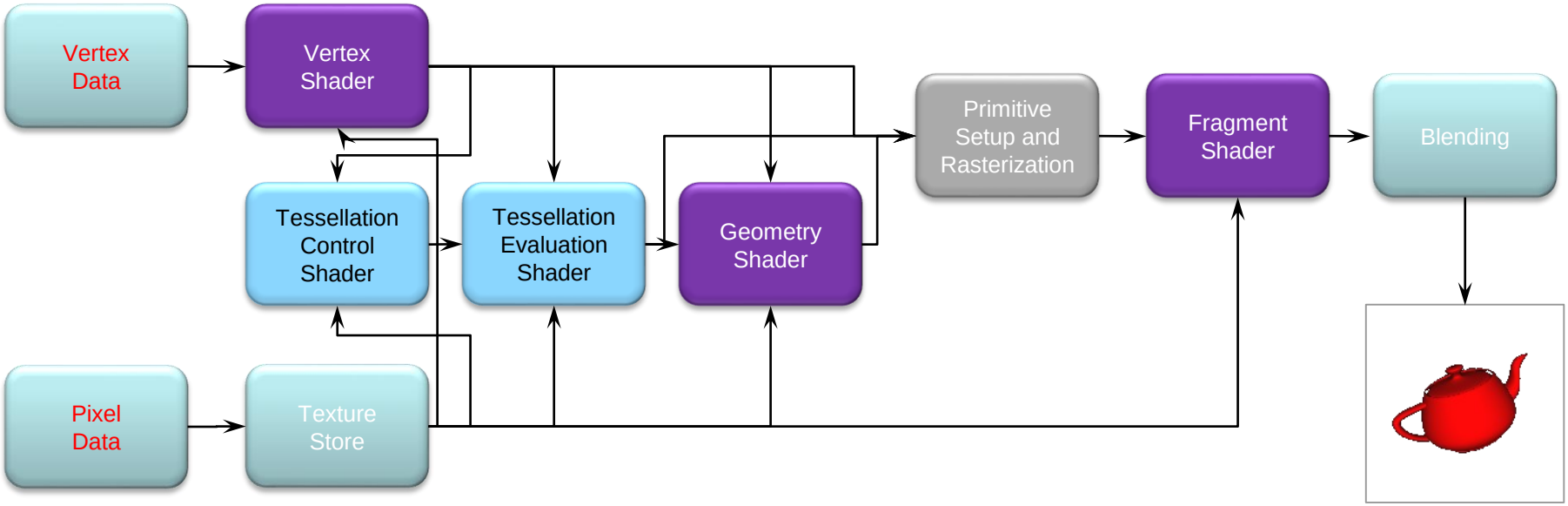
OpenGL 3.2 (released August 3rd, 2009) added an additional shading stage – *geometry shaders*

- modify geometric primitives within the graphics pipeline

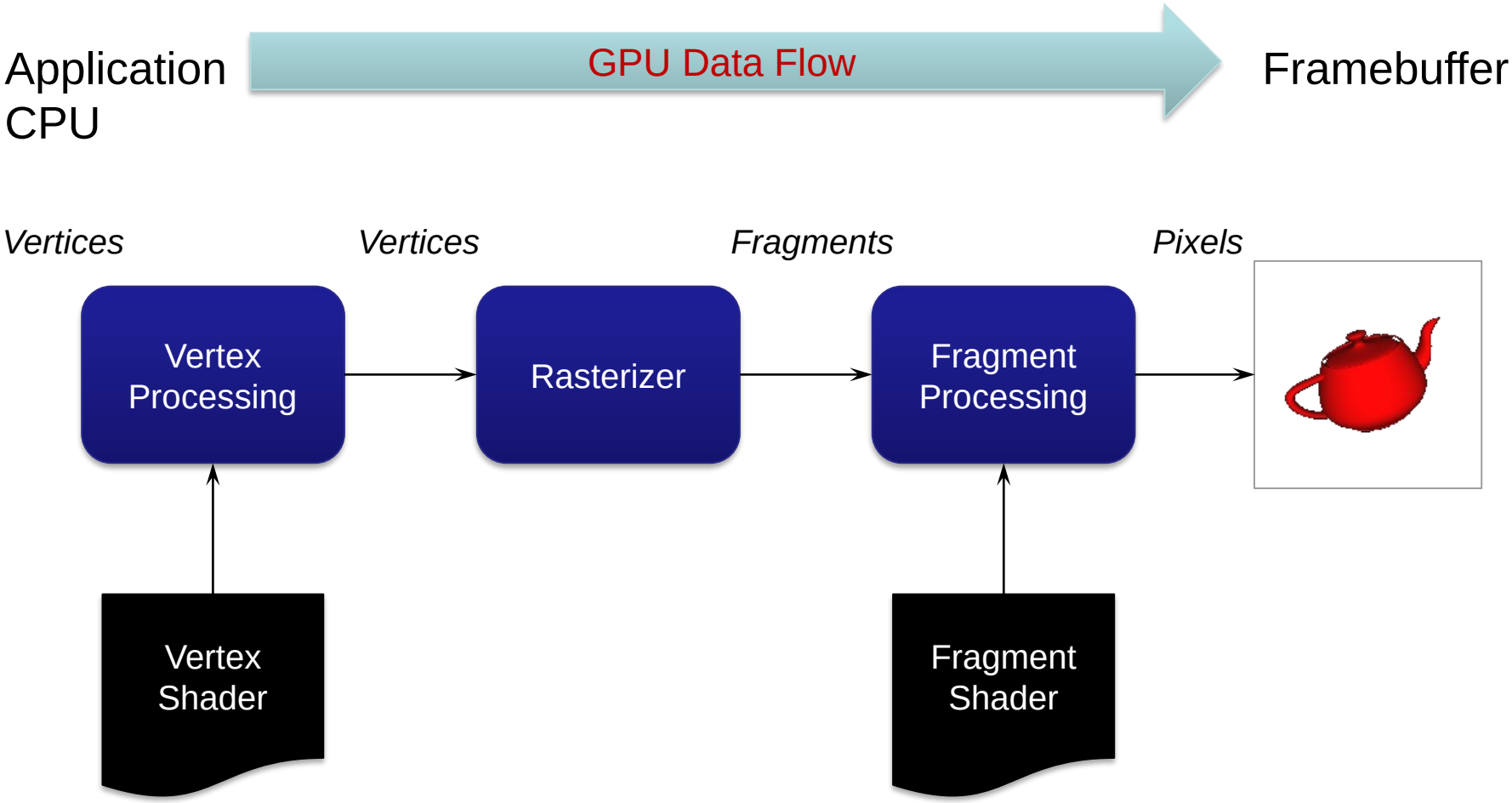


The Latest Pipelines

- 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)



A Simplified Pipeline Model



OpenGL Programming in a Nutshell

- Programmable pipeline OpenGL programs essentially do the following steps:
  1. Create shader programs

     OpenGL Shading Language (GLSL)
  1. Create buffer objects and load data into them
  2. “Connect” data locations with shader variables
  3. Render

OpenGL Application Development

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 communicates with native windowing system and 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 , one alternative is **GLFW** which runs on Windows, Linux and Mac OS X.

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
- GLEW **#include <GL/glew.h>**

Windows OS using OpenGL/FreeGLUT/GLEW with Visual Studio

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

To create a new project:

1. Click on the "New Project"
2. Select “App Console"
3. 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. (right click) header file **ShaderMaker.h**

   (right click) source file **ShaderMaker.cpp**

menu “Tools – NuGet Packages – Console ”

>> install-package nupengl.core

>> install-package glm % math library

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

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)
  - **`initShader():`** read, compile and link shaders
  - **`init()`**: sets the state variables
    - Viewing
    - Attributes
  - callbacks
    - Display function
    - Input and window functions

main.c

```
#include <GL/glew.h>
#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(drawScene);
   glutReshapeFunc( resize );
   glutKeyboardFunc( keyb );
   initShader()
   init();
   glutMainLoop();
}
```

includes **gl.h**

specify window properties

display title

display callback

initialize shaders

set OpenGL state

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*)
  - RGBA color
  - Single/Double buffering
- **`glutWindowSize`** in pixels
- **`glutWindowPosition`** from top-left corner of display
- **`glutCreateWindow`** create window with title “simple”
- **`glutDisplayFunc`** display callback
- **`glutMainLoop`** enter infinite event loop

Rendering Callback

```
glutDisplayFunc(drawScene);

void drawScene( void )
{
glClear(GL_COLOR_BUFFER_BIT);


// draw the objects
glBindVertexArray(VAO);
glDrawArrays(GL_TRIANGLES, 0, 3);

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

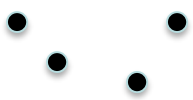
glutMouseFunc(), glutMotionFunc(), glutPassiveMouseFunc()

Events in OpenGL

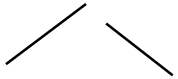
Event	Example	OpenGL Callback Function
Keypress	KeyDown KeyUp	`glutKeyboardFunc`
Mouse	leftButtonDown leftButtonUp	`glutMouseFunc`
Motion	With mouse press Without	`glutMotionFunc` `glutPassiveMotionFunc`
Window	Moving Resizing	`glutReshapeFunc`
System	Idle Timer	`glutIdleFunc` `glutTimerFunc`
Software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

OpenGL's Geometric Primitives

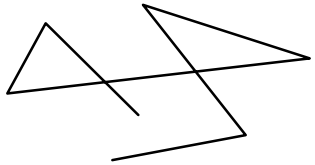
All primitives are specified by vertices



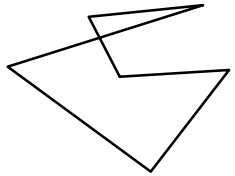
GL_POINTS



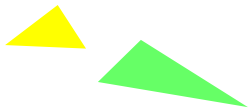
GL_LINES



GL_LINE_STRIP



GL_LINE_LOOP



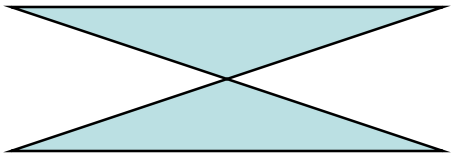
GL_TRIANGLES

GL_TRIANGLE_STRIP

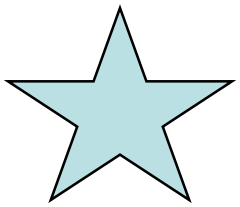
GL_TRIANGLE_FAN

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 4.6 contains a tessellator



nonsimple polygon



nonconvex polygon

The rendering techniques are:

1) Immediate mode

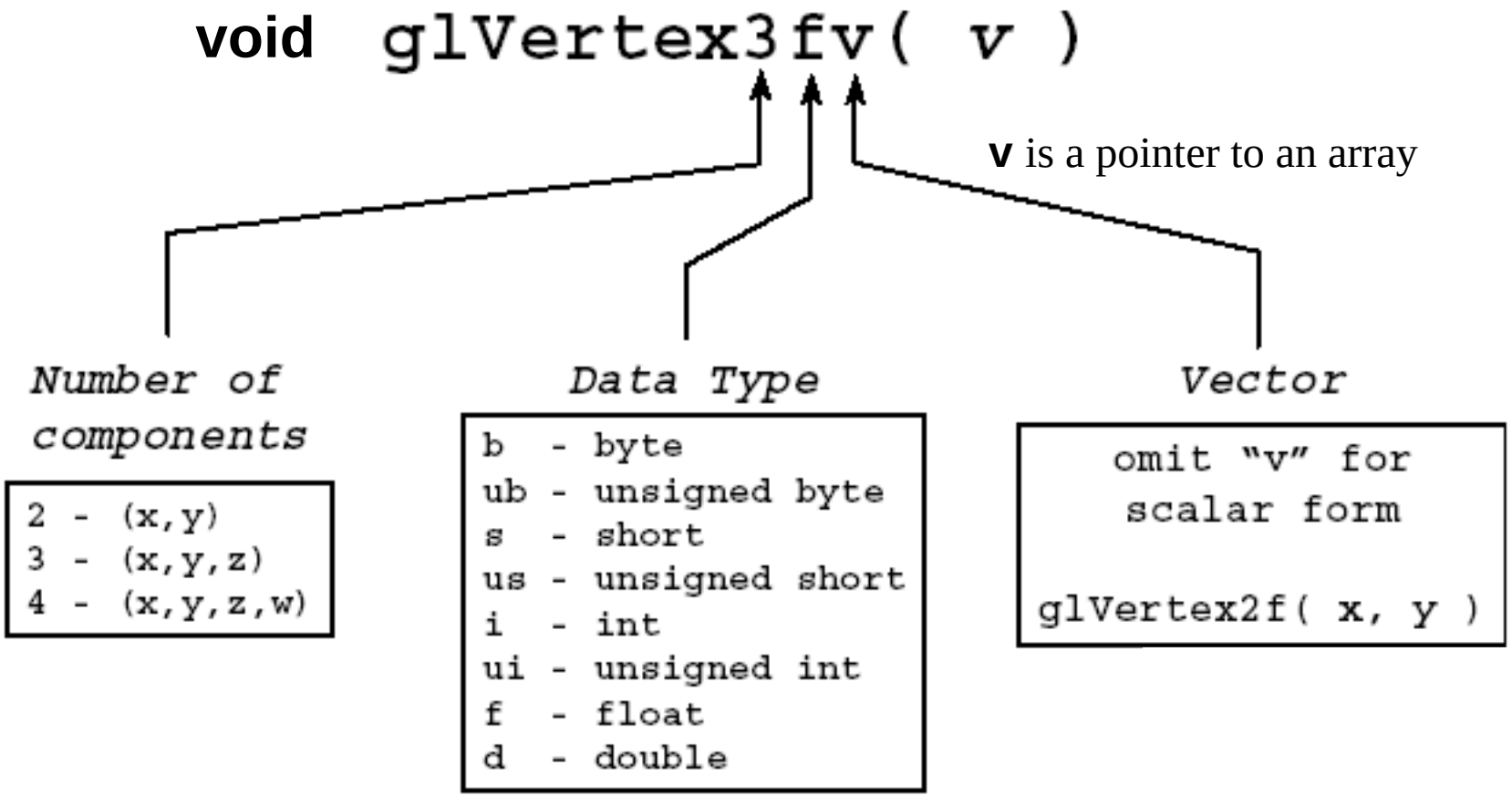
- Each time a vertex is specified in application, its location is sent to the GPU
- Uses **`glVertex, glBegin/glEnd`**
- Creates bottleneck between CPU and GPU
- To redraw the same data needs to resend the data

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

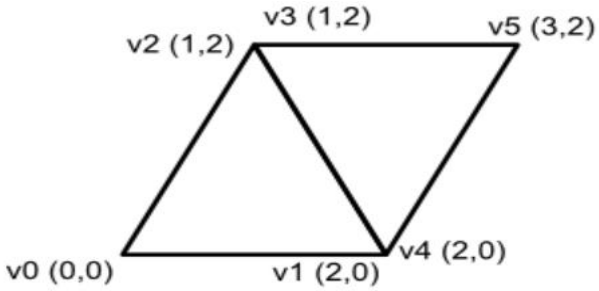
2) DrawArrays/DrawElements with VBOs

Vertex buffer objects store arrays of vertex data--positional or descriptive. With a vertex buffer object ("VBO") you can compute all vertices at once, pack them into a VBO, and pass them to OpenGL en masse to let the GPU processes all the vertices together.

`glDrawArrays()` *or* **`glDrawElements()`**

DrawArrays (without indexing)

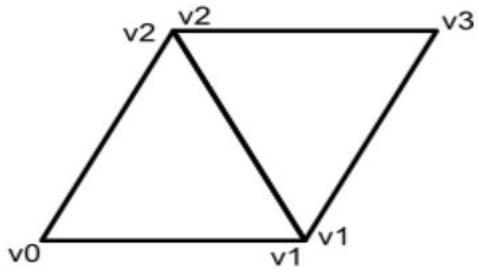
% reduces the number of function calls



[0,0, 2,0, 1,2, 1,2, 2,0, 3,2]

DrawElements (with indexing)

%reduces the number of function calls and redundant usage of shared vertices

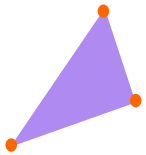


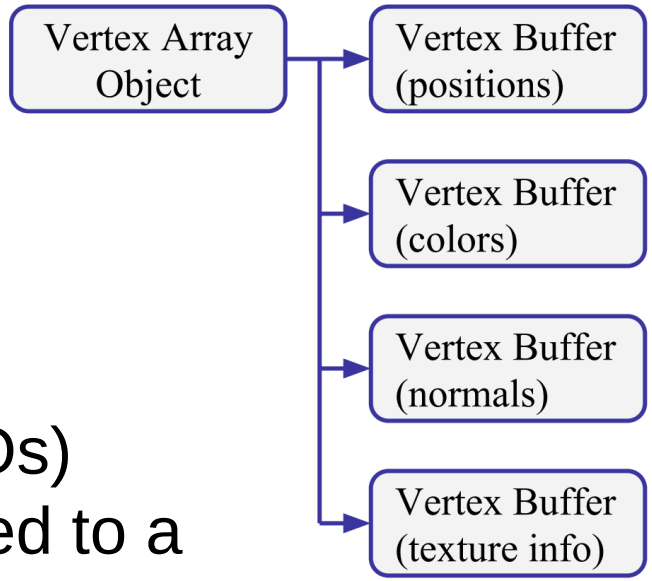
[0,1,2, 2,1,3]

[0,0, 2,0, 1,2, 3,2]

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





- Vertex **data** must be stored in **vertex buffer objects** (VBOs)
- One or multiple VBOs must be assigned to a **vertex array objects** (VAOs)

We can create as many buffer objects VBO as we want for different types of per-vertex data. This lets us bind vertices with normals, colors, texture coordinates, etc...

Vertex Array Objects (VAOs)

- VAOs store the data of an geometric object
- Steps in using a VAO

```
unsigned int vao;
```

1) generate VAO names by calling

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

2) sets the VAO as the active one

```
glBindVertexArray(vao);
```

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

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 time (in **`init()`**)
  - generate VBO names by calling

```
unsigned int VBO_1;
glGenBuffers(1, &VBO_1);
```

  - sets a buffer as the current buffer, i.e. the buffer we are working on now

```
glBindBuffer(GL_ARRAY_BUFFER, VBO_1);
```

  - load vector data (vData) into VBO using (into the GPU's memory)

```
glBufferData(GL_ARRAY_BUFFER,sizeof(vData),
                                vData,GL_STATIC_DRAW);
```

- At rendering time (in **`drawScene()`**)
  - bind VAO for use in rendering

```
glBindVertexArray(vao)
glDrawArrays( GL_TRIANGLES, 0, NumVertices );
```

A First Program

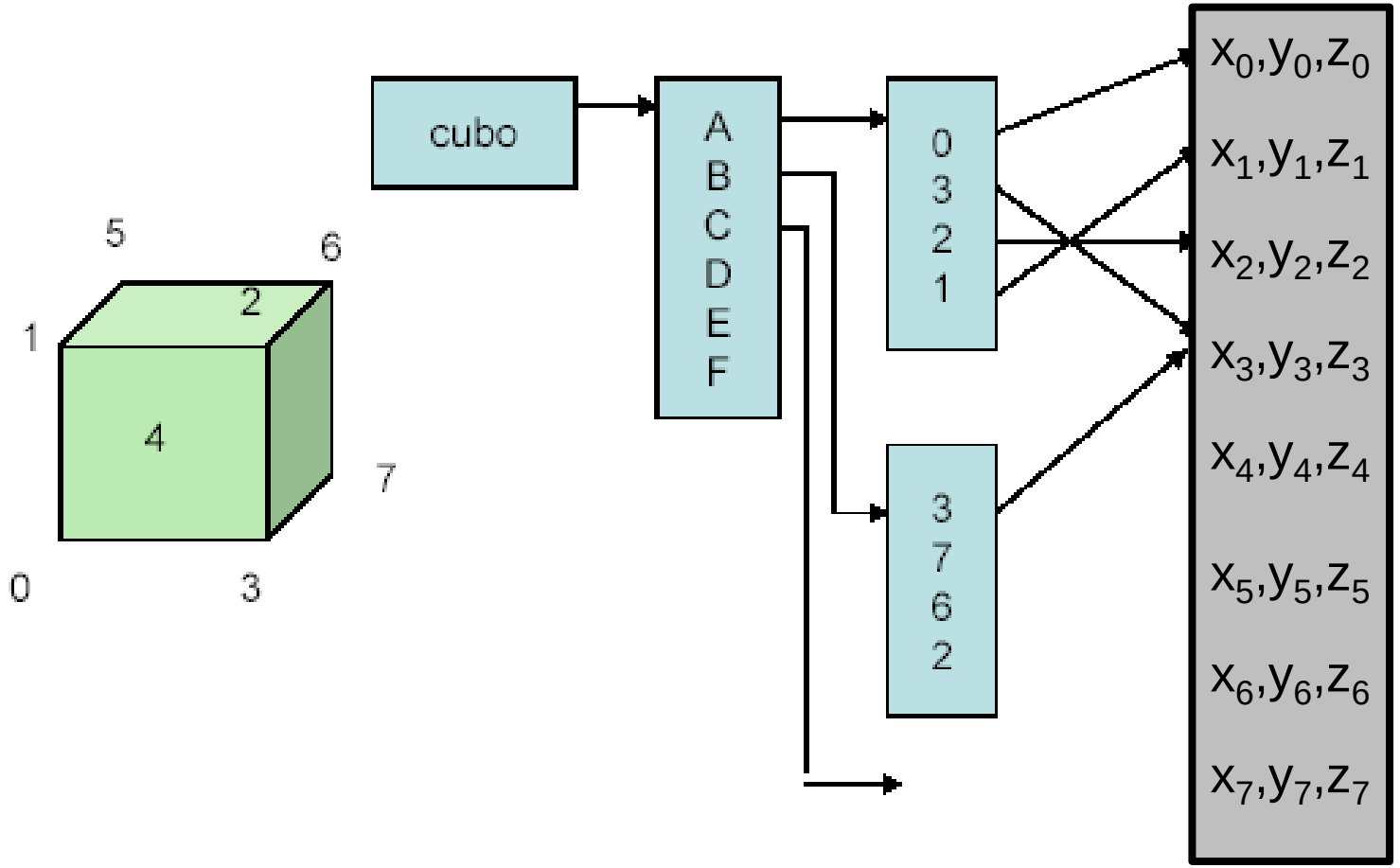
render a cube

with colors at each vertex

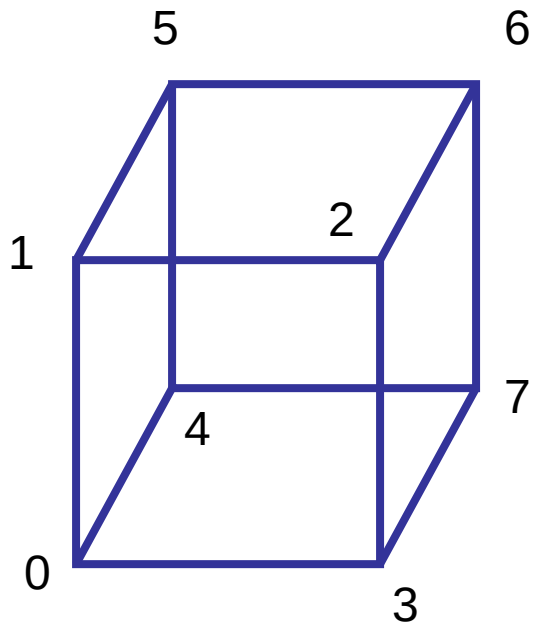
Draw a cube

Cube is an object with 6 faces and 8 vertices

We'll render a cube with colors at each vertex



```
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);
}
```



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

1) Draw a cube: Immediate mode

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}};
```

1) Draw a cube: Immediate mode

Define faces and draw

```
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

2) 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.

Initializing the Cube's Data

- We'll build each cube face from individual triangles
- Need to determine how much storage is required
  - (6 faces)(2 triangles/face)(3 vertices/triangle)

```
const int NumVertices = 36;
```

- To simplify communicating with GLSL, we'll use a `vec4` class (implemented in C++) similar to GLSL's `vec4` type
  - we'll also typedef it to add logical meaning

```
typedef  vec4  point4;
typedef  vec4  color4;
```

- We create two arrays to hold the VBO data

```
point4  vPositions[NumVertices];
color4  vColors[NumVertices];
```

Cube Data

- Vertices of a unit cube centered at origin
  - sides aligned with axes

```
point4 positions[8] = {
    point4( -0.5, -0.5,  0.5, 1.0 ),
    point4( -0.5,  0.5,  0.5, 1.0 ),
    point4(  0.5,  0.5,  0.5, 1.0 ),
    point4(  0.5, -0.5,  0.5, 1.0 ),
    point4( -0.5, -0.5, -0.5, 1.0 ),
    point4( -0.5,  0.5, -0.5, 1.0 ),
    point4(  0.5,  0.5, -0.5, 1.0 ),
    point4(  0.5, -0.5, -0.5, 1.0 )
};
```

Cube Data (cont'd)

- We'll also set up an array of RGBA colors

```
color4 colors[8] = {
    color4( 0.0, 0.0, 0.0, 1.0 ),  // black
    color4( 1.0, 0.0, 0.0, 1.0 ),  // red
    color4( 1.0, 1.0, 0.0, 1.0 ),  // yellow
    color4( 0.0, 1.0, 0.0, 1.0 ),  // green
    color4( 0.0, 0.0, 1.0, 1.0 ),  // blue
    color4( 1.0, 0.0, 1.0, 1.0 ),  // magenta
    color4( 1.0, 1.0, 1.0, 1.0 ),  // white
    color4( 0.0, 1.0, 1.0, 1.0 )   // cyan
};
```

Generating a Cube Face from Vertices

- To simplify generating the geometry, we use a convenience function polygon( )
  - create two triangles for each face and assigns colors to the vertices

```
int Index = 0;  // global variable indexing into VBO arrays

void polygon( int a, int b, int c, int d )
{
    vColors[Index] = colors[a]; vPositions[Index] = positions[a]; Index++;
    vColors[Index] = colors[b]; vPositions[Index] = positions[b]; Index++;
    vColors[Index] = colors[c]; vPositions[Index] = positions[c]; Index++;
    vColors[Index] = colors[a]; vPositions[Index] = positions[a]; Index++;
    vColors[Index] = colors[c]; vPositions[Index] = positions[c]; Index++;
    vColors[Index] = colors[d]; vPositions[Index] = positions[d]; Index++;
}
```

Generating the Cube from Faces

- Generate 12 triangles for the cube
  - 36 vertices with 36 colors

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

VAO & VBOs in Code

```
In init()
colorcube() //Pack the data positions and colors in vPositions and vColors array

// Create a vertex array object
GLuint vao;
glGenVertexArrays( 1, &vao );
glBindVertexArray( vao );

// 1st attribute VBO : positions
Gluint vpositionID;
glGenBuffers(1, &vpositionID);
glBindBuffer(GL_ARRAY_BUFFER, vpositionID);
glBufferData(GL_ARRAY_BUFFER,sizeof(vPositions),vPositions, GL_STATIC_DRAW);
glEnableVertexAttribArray(0);      // attribute index to be enabled
glVertexAttribPointer( //how Opengl should interpret the vertex data
        0,                 // attribute index in the shader
        3,                 //number of components per vertex (must be 1,2,3, or 4)
        GL_FLOAT,          // data type of each component
        GL_FALSE,          // data should be normalized?
        0,    // stride : byte offset between two consecutive vertex attributes
        (void*)0           // specify an offset of the first component of the array
 );
```

Caution: all VBOs in a VAO must describe the same number of vertices!

VAO & VBOs in Code

```
// 2nd attribute VBO : colors
Gluint vcolorID;
glGenBuffers(1, &vcolorID);
glBindBuffer(GL_ARRAY_BUFFER, vcolorID);
glBufferData(GL_ARRAY_BUFFER,sizeof(vColors), vColors, GL_STATIC_DRAW);
glEnableVertexAttribArray(1);

glVertexAttribPointer(
                1, // attribute index. No particular reason for 1,
                       but must match layout in the shader.
                3,                  // size
                GL_FLOAT,           // type
                GL_FALSE,           // normalized?
                0,                  // stride
                (void*)0            // array buffer offset
    );
```

Drawing Geometric Primitives

In drawScene()

```
void drawScene( void )
{
glBindVertexArray(vao);
glDrawArrays( GL_TRIANGLES, 0, NumVertices );

glutSwapBuffers();
}
```

See code 3D_CUBE_TRANSMAT.cpp

if you call glDrawArrays() specifiying 100 vertices, your vertex shader will be called 100 times, each time with a different vertex.

OpenGL Mathematics (GLM)

The GLM Library

A C++ mathematics library for graphics programming

GLM Setup

```
// include basic vector and matrix types
      #include <glm/glm.hpp>

// include matrix transform
      #include <glm/gtc/matrix_transform.hpp>
      #include <glm/gtc/type_ptr.hpp>
```

GLM vectors

- Integer type vectors: ivec{2,3,4}
- Float type vectors: vec{2,3,4}

```
ivec2   mivec2;                    // mivec2 = (0, 0)
ivec3   mivec3(mivec2, 0);         // mivec3 = (0, 0, 0)
ivec4   mivec4(mivec3, 0);         // mivec4 = (0, 0, 0, 0)
vec2    mvec2;                     // mvec2 = (0.0f, 0.0f)
vec3    mvec3(1.0f, 2.0f, 3.0f); // mvec3 = (1.0f, 2.0f, 3.0f)
vec3    V(3.0f, 4.0f, 5.0f);   // a non-unit vector V
float   V_len = V.length();    // find the length of V by .length()
                                        member
vec3    V_unit = V / V_len;    // find the unit vector by division
V_unit = normalize(V);         // or by glm::normalize function
```

Vector Dot Product

```
vec3 A(x1, y1, z1), B(x2, y2, z2);
// calculate dot product by definition
float    dotProduct = A.x * B.x + A.y * B.y + A.z * B.z;
dotProduct = dot(A, B);
// calculate angle by definition (note that 0 <= angleRad <= PI)
float angleRad = acos(dotProduct / A.length() / B.length());
float angleDegree = angleRad / PI * 180.0f;
```

Vector Cross Product

```
vec3 A(x1, y1, z1), B(x2, y2, z2);
// calculate dot product by definition
vec3 crossProduct = vec3(
A.y * B.z - A.z * B.y,
A.z * B.x - A.x * B.z,
A.x * B.y - A.y * B.x
);
crossProduct = cross(A, B);
```

Matrix

- Declare a matrix in glm

```
mat4 M1;          // create an identity matrix
mat3 M2(2.0f);    // create a 3x3 matrix with diagonal components set to 2.0f
// create a matrix by 4D vectors
mat4 M3(vec4(1, 2, 3, 4), vec4(1, 2, 3, 4),vec4(1, 2, 3, 4),vec4(1, 2, 3, 4));
float c00 = M3[0][0];                  // access the matrix component
```

- Multiply an identity matrix to a vector

```
vec4 V(1, 2, 3, 1);
mat4 M4(1.0f);              // create a 4x4 identity matrix
vec4 V2 = M4 * V;           // apply the identity matrix
if(V == V2)
        printf("V is the same as V2");
```

Translation Matrix

- Construct a translation matrix by (T_x, T_y, T_z)

```
#include <glm/gtx/transform.hpp>

vec4 V(1, 2, 3, 1);
mat4 M4(1.0);
M4 = translate(M4, vec3(Tx, Ty, Tz));
vec4 V2 = M4 * V; // apply the translation matrix
```

Scaling Matrix

- Construct a scaling matrix by (S_x, S_y, S_z)

```
#include <glm/gtx/transform.hpp>

vec4 V(1, 2, 3, 1);
mat4 M4(1.0);
M4 = scale(M4, vec3(Sx, Sy, Sz));
vec4 V2 = M4 * V; // apply the scaling matrix
```

Rotation around Given Axis

- Construct a rotation matrix by given axis (R_x, R_y, R_z) and angle θ

```
#include <glm/gtx/transform.hpp>
#define PI 3.14159265358979323846
#define deg2rad(x) ((x)*((PI)/(180.0)))

vec4 V(1, 2, 3, 1);
mat4 M4(1.0);
M4 = rotate(M4, deg2rad(theta), vec3(Rx, Ry, Rz));
vec4 V2 = M4 * V; // apply the rotation matrix
```

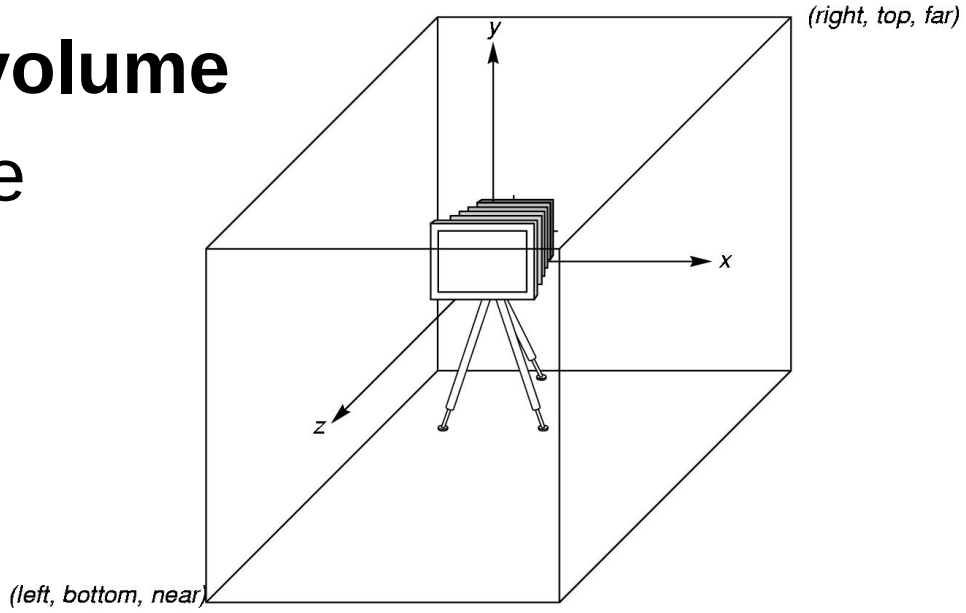
Transformations

- Modeling
- Viewing
- Projection

OpenGL Camera

- OpenGL places a camera at the origin in object space pointing in the negative z direction

- The default **viewing volume** is a box centered at the origin with sides of length 2



Transformations and camera analogy

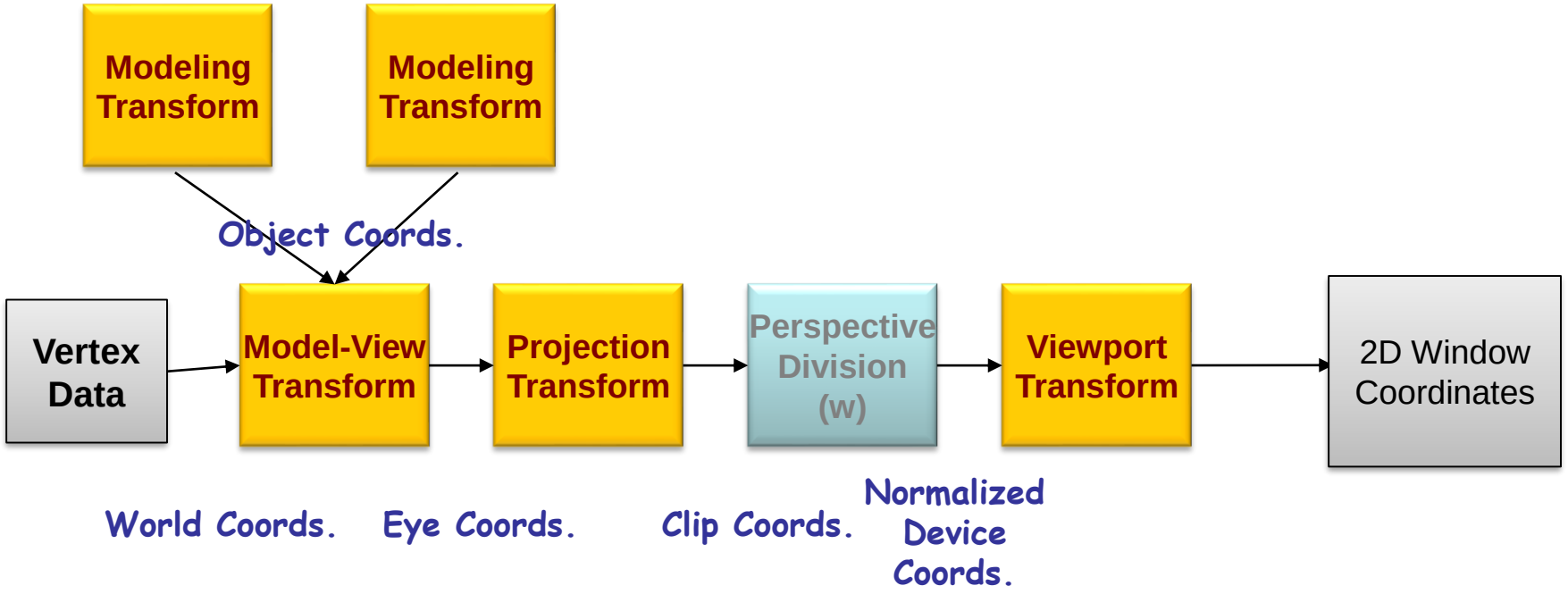
- Modeling transformations
  - moving the model
- Viewing transformations
  - camera–define position and orientation of the viewing volume in the world
- Projection transformations
  - adjust the lens of the camera
- Viewport transformations
  - enlarge or reduce the physical photograph

Moving camera is equivalent to moving every object in the world towards a stationary camera

Transfomations in OPENGL

Transformations take us from one “space” to another, used to realize the 3D scene and the rendering pipeline

We use transformations represented by 4×4 matrices



Model Transformations

First define the transformations on the geometric model then draw the geometric model itself.

```
//initialize identity matrix and transformations
mat4 trans = mat4(1.0f);
trans = translate( trans, dx, dy, dz );   //matrix C
trans = scale(trans, sx, sy, sz );        //matrix B
trans = rotate(trans, angle, rx,ry,rz );  //matrix A

// Draw  the object
glDrawArrays(GL_TRIANGLES, 0, nvertices);
```

p'=(C(B(Ap)))

Apply the tranformations in inverse order w.r.t. their definitions

Rotation about a Fixed Point

We want $\mathbf{C} = \mathbf{T}^{-1}\,\mathbf{R}\,\mathbf{T}$
so we must do the operations in the following order

$\mathbf{C} \leftarrow \mathbf{I}$	Start with identity matrix
$\mathbf{C} \leftarrow \mathbf{C}\mathbf{T}^{-1}$	Move fixed point back
$\mathbf{C} \leftarrow \mathbf{C}\mathbf{R}$	Rotate
$\mathbf{C} \leftarrow \mathbf{C}\mathbf{T}$	Move fixed point to origin

Each operation corresponds to one function call in the program. Note that the last operation specified is the first executed in the program

Example

First we scale the object by 0.5 on each axis and then rotate the object 90 degrees around the z axis.

```
mat4 trans = mat4(1.0f);
mat4 trans = rotate(trans, radians(90.0f), vec3(0.0f, 0.0f, 1.0f));
mat4 trans = scale(trans, vec3(0.5f, 0.5f, 0.5f));
draw_object()
```

Each vertex p that is draw in draw_object() will be multiplied by trans matrix , thus forming the new vertex q = trans * p

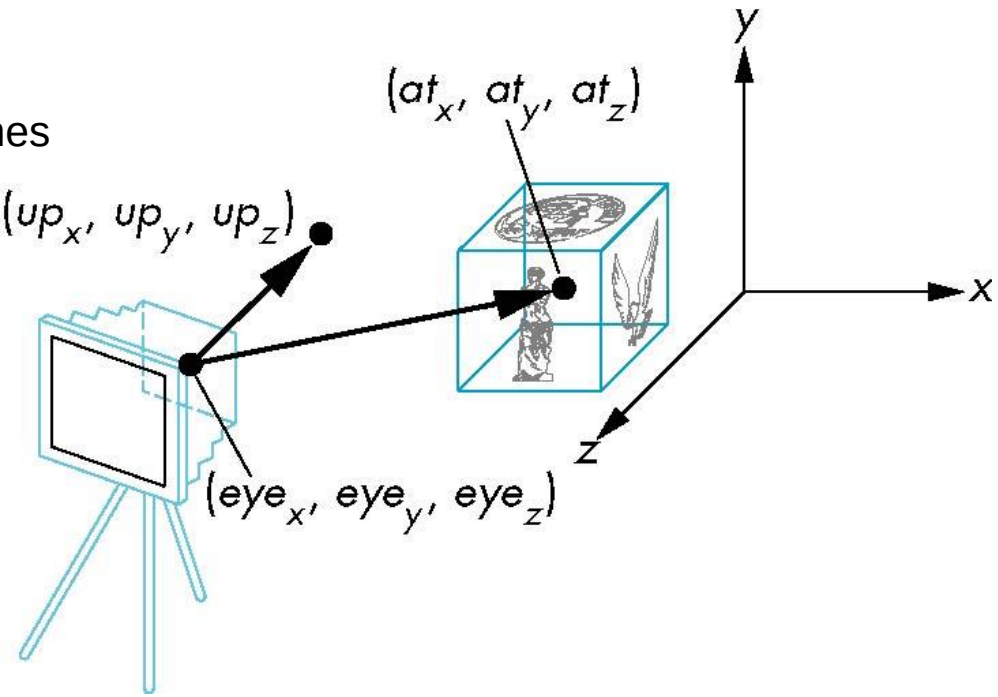
$$q = \mathbf{R}_{90}(0,0,1)\mathbf{S}(0.5,0.5,0.5)\ p$$

View Transformation

- Construct a viewing matrix

```
#include <glm/gtx/transform.hpp>
vec4 V(1, 2, 3, 1);
mat4 M4 = lookAt(vec3(Px, Py, Pz), vec3(Ex, Ey, Ez),vec3(Ux, Uy, Uz));
vec4 V2 = M4 * V; // apply the viewing matrix
```

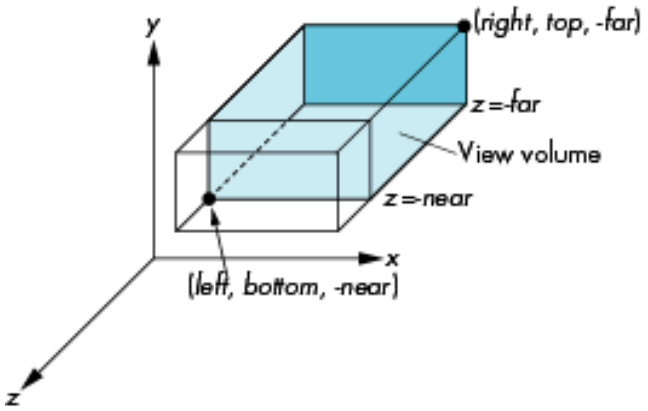
up vector determines unique orientation



Orthographic Projection

- Construct an orthographic projection matrix

```
#include <glm/gtx/transform.hpp>

vec4 V(1, 2, 3, 1);
mat4 M4 = ortho(left, right, bottom, top);
M4 = ortho(left, right, bottom, top, 0, 1); // this is the same as above

M4 = ortho(left, right, bottom, top, nearVal, farVal);
vec4 V2 = M4 * V; // apply the projection matrix
```



near and **far**
measured from camera

2D orthographic

In 2D, the clipping **volume** becomes a clipping **rectangle**

Equivalent to calling **glOrtho** with *near* = *-1* and *far* = *1*

```
gluOrtho2D(left, right, bottom, top);
```

define a 2D orthographic projection matrix used to set up the world window.

Example: If we wanted a world window with

x varying from -1.0 to 1.0 and

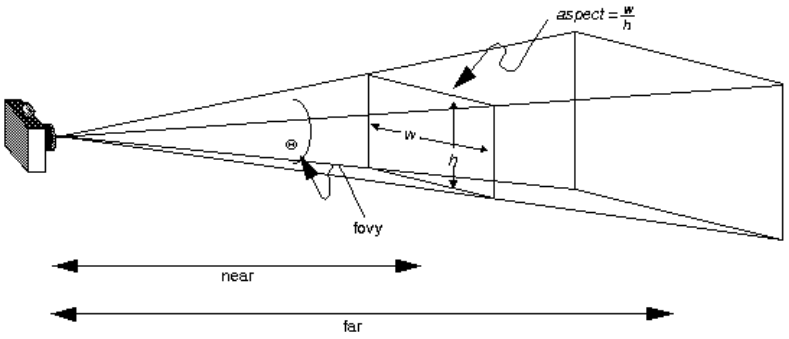
y varying from 3.0 to 5.0,

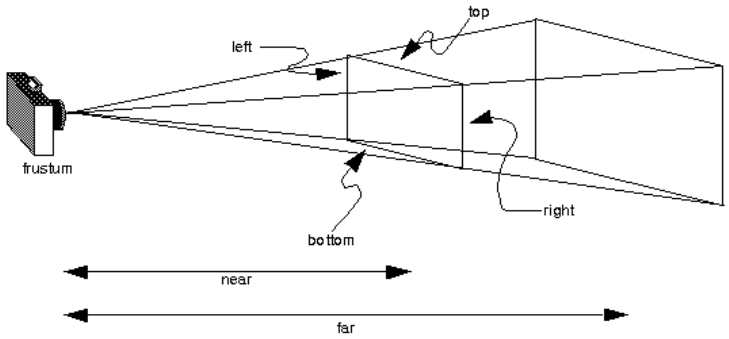
```
gluOrtho2D(-1.0, 1.0, 3.0, 5.0);
```

Perspective Projection

- Construct a perspective projection matrix

```
#include <glm/gtx/transform.hpp>
#define PI 3.14159265358979323846
#define deg2rad(x) ((x)*((PI)/(180.0)))

vec4 V(1, 2, 3, 1);
mat4 M4 = frustum(left, right, bottom, top, nearVal, farVal);
// use frustum
M4 = perspective(deg2rad(fovy), aspect, nearVal, farVal);
// or perspective
vec4 V2 = M4 * V; // apply the projection matrix
```





FOVY is the angle between the top and bottom planes (between 0.0 and 180.0)

Matrix Transformation

- V' = Projection * Viewing * Modeling * V

drawScene()

```
mat4 model;
mat4 view;
mat4 proj;
…
mat4 mvp = proj * view * model;
glUniformMatrix4fv(0, 1, GL_FALSE, value_ptr(mvp));
```

Send the matrix data to the vertex shader

How many matrices? 1
Transposed? FALSE

vertex_shader.glsl

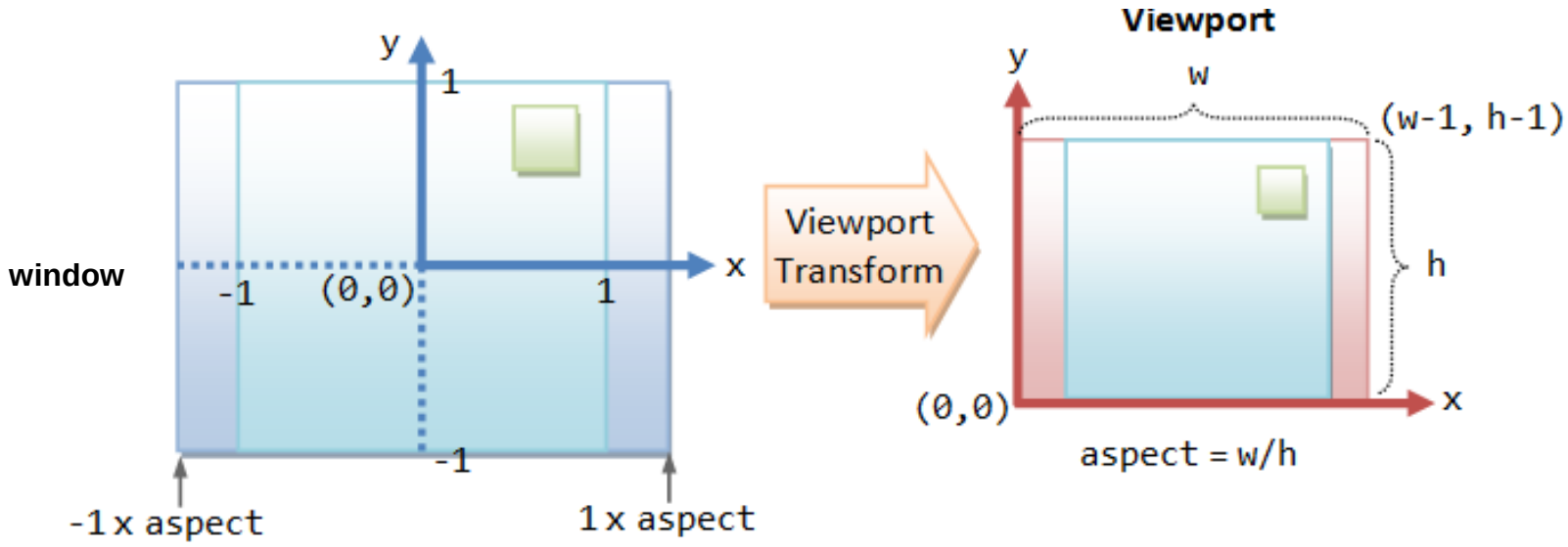
```
#version 330 core
layout (location = 0) in vec3 aPos;

uniform mat4 Umvp;
in vec4 vertex;

void main()
{
gl_Position = Umvp * vertex;
}
```

Transform the vertex coords.

Viewport Transformations

- Viewport = rectangular region of the screen where the image is projected
- Default : viewport is the window open in the screen
- Modify the viewport with the function

```
glViewport(x,y,width,height)
```



x y specify the lower left corner of the viewport rectangle, in pixels, default (0,0)

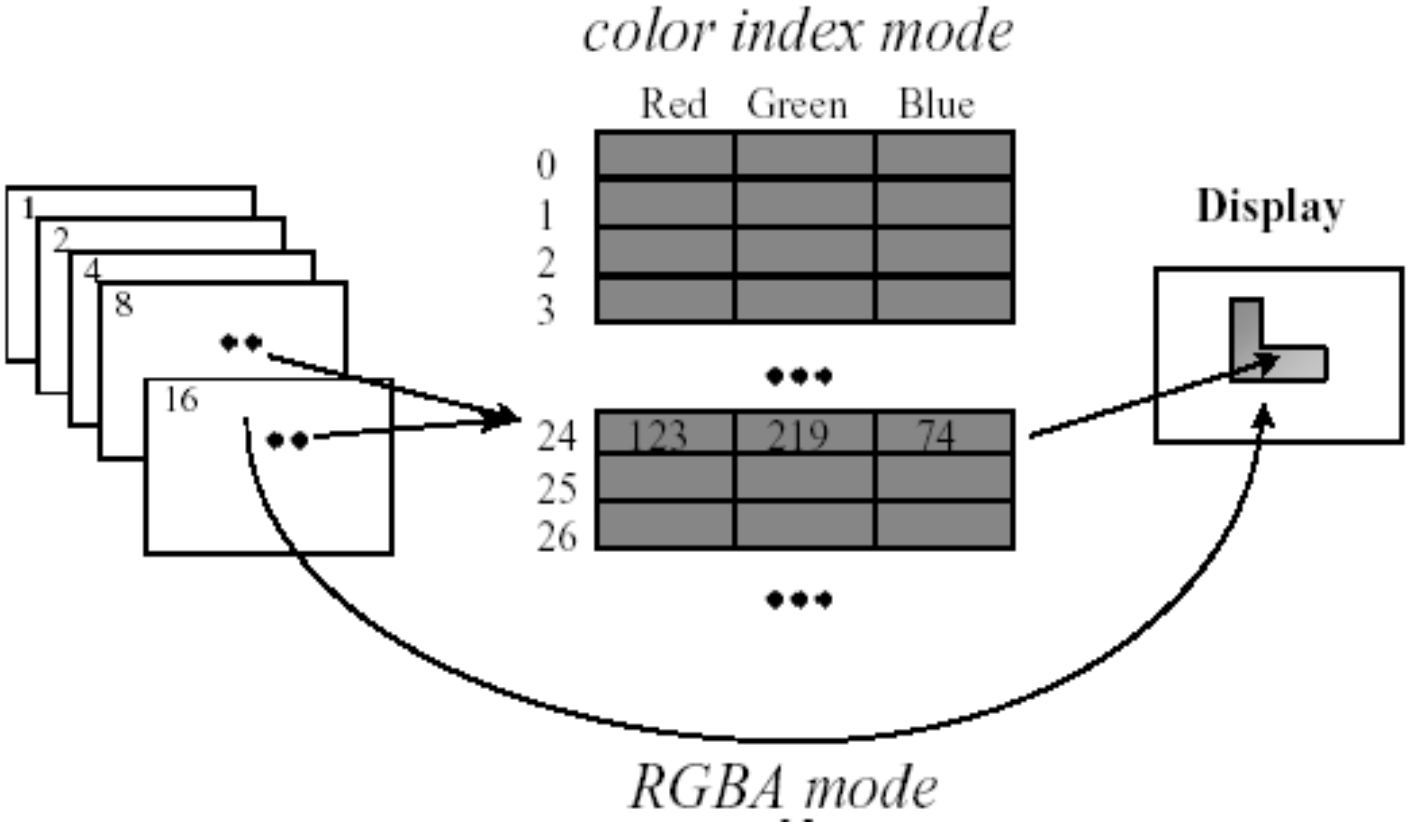
Reshape callback

Registered as callback for **glutReshapeFunc()**

```
/* called for window resize */
void resize( int w, int h )
{
glViewport( 0, 0, w, h );
perspective( 65.0, (GLfloat) w / h,1.0, 100.0 );
lookAt( 0.0, 0.0, 5.0,0.0, 0.0, 0.0,0.0, 1.0, 0.0);
}
```

resize() uses the viewport's width and height values as the aspect ratio for perspective() which eliminates distortion.

RGB vs Indexed COLOR



RGBA (TRUECOLOR) or COLOR INDEX (COLORMAP)

```
glutInitDisplayMode(GLUT_RGBA);
```

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)

Multiple **(Double)** buffering

Front buffer and Back buffer

Graphics cards constantly redraw the content of the frame buffer (front,back,front,back..) on the display.

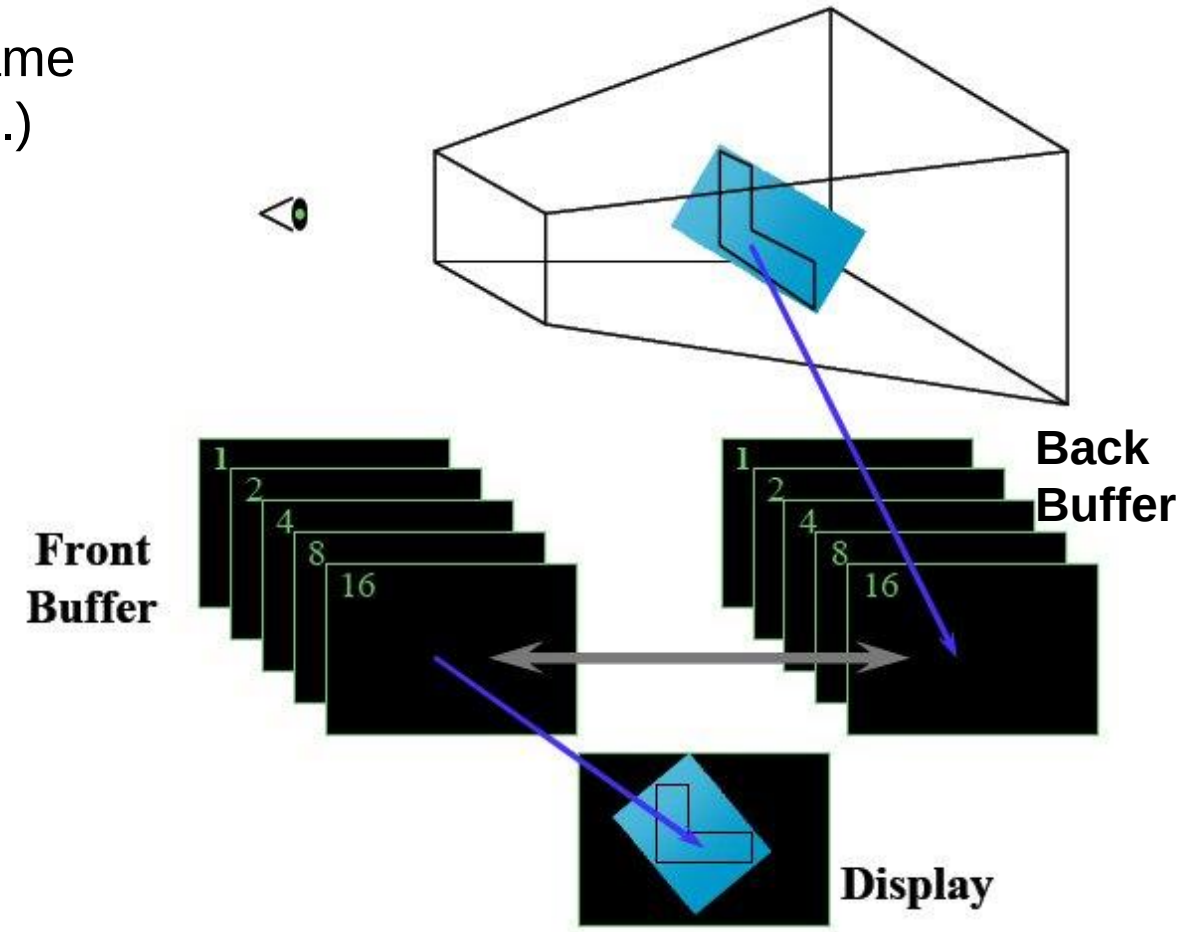
The program draws in the back buffer (frame buffer)

WHILE

the front buffer is read by the graphics card to display the image on the monitor

SWAP (exchange of roles of the two buffers)

The program starts the next frame drawing in the buffer….



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 draw

Idle Callback

it's called after all the current events are processed.

```
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

glutTimerFunc() registers a timer callback to be triggered in a specified number of milliseconds.

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++*
- OpenGL Shading Language.