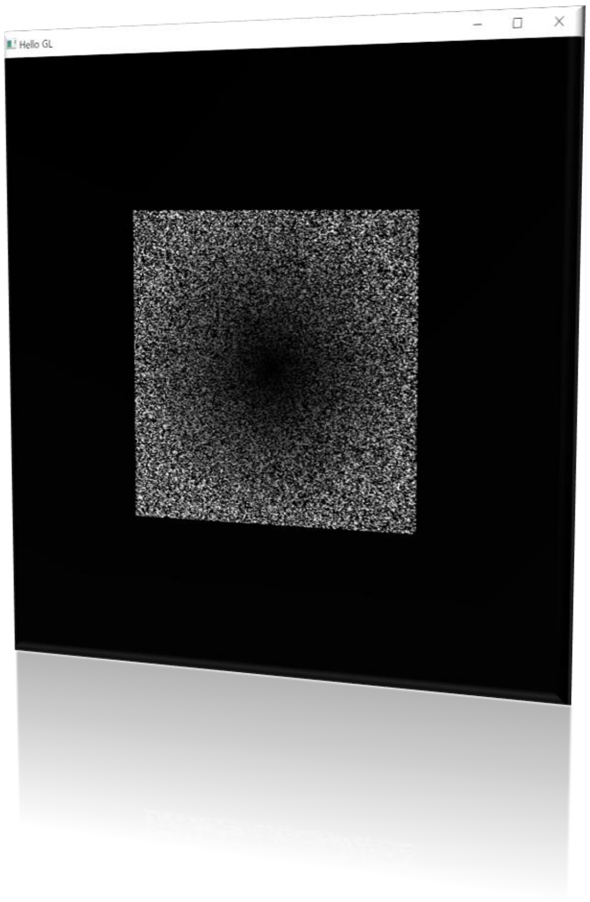
A Color Cube Example and OpenGL Shading Language (GLSL)

Sang Il Park

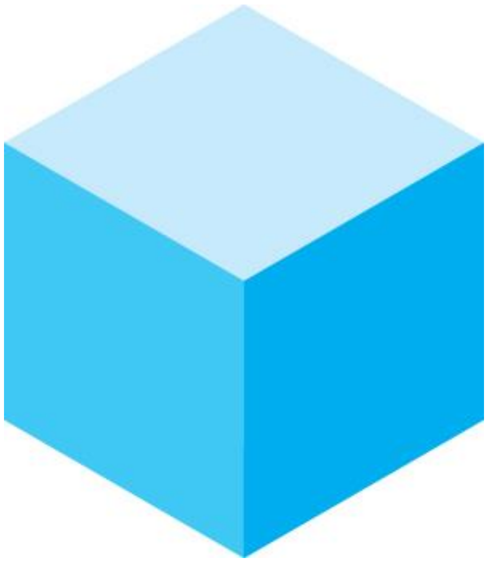
Dept. of Software

Review:

- Setting for libraries
  - Set include/lib folder
  - `#include <vgl.h>`
  - `#include <initshader.h>`
- Creating data(in an array form)
- Sending the data
  - VAO – vertex array object
  - VBO – vertex buffer object
- Loading the shaders
- Draw it with `glDrawArrays(...)`



A Color Cube example



Rendering 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

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

```
#include <vec.h>
```

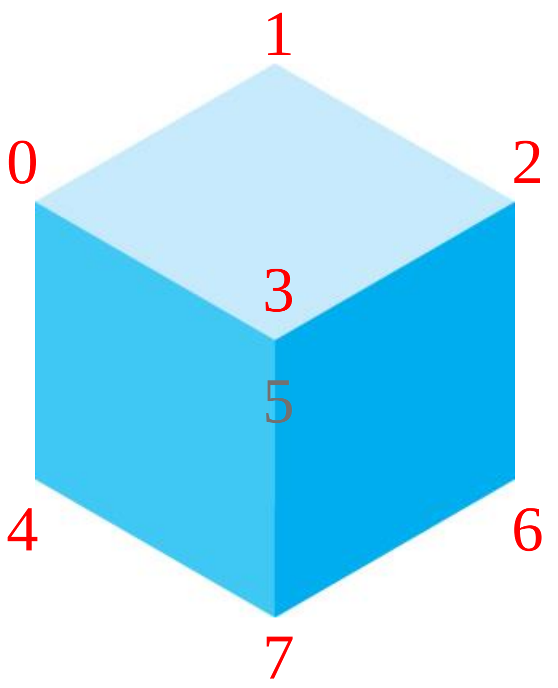
Initializing the Cube's Data (cont'd)

- Before we can initialize our VBO, we need to stage the data
- Our cube has two attributes per vertex
  - position
  - color
- We create two arrays to hold the VBO data

```
vec4  points[NumVertices];
vec4  colors[NumVertices];
```

Cube Data

```
// Vertices of a unit cube centered at origin, sides aligned
  with axes
  vec4 vertex_pos [8] = {
        vec4 ( -0.5, -0.5,  0.5, 1.0 ),
        vec4 ( -0.5,  0.5,  0.5, 1.0 ),
        vec4 (  0.5,  0.5,  0.5, 1.0 ),
        vec4 (  0.5, -0.5,  0.5, 1.0 ),
        vec4 ( -0.5, -0.5, -0.5, 1.0 ),
        vec4 ( -0.5,  0.5, -0.5, 1.0 ),
        vec4 (  0.5,  0.5, -0.5, 1.0 ),
        vec4 (  0.5, -0.5, -0.5, 1.0 )
};
```

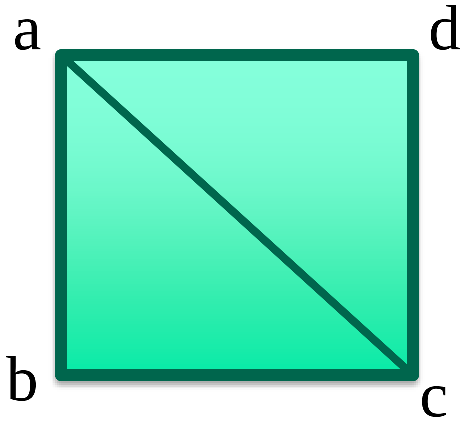


Cube Data

```
// RGBA colors
 vec4 vertex_colors[8] = {
        vec4 ( 0.0, 0.0, 0.0, 1.0 ),  // black
        vec4 ( 1.0, 0.0, 0.0, 1.0 ),  // red
        vec4 ( 1.0, 1.0, 0.0, 1.0 ),  // yellow
        vec4 ( 0.0, 1.0, 0.0, 1.0 ),  // green
        vec4 ( 0.0, 0.0, 1.0, 1.0 ),  // blue
        vec4 ( 1.0, 0.0, 1.0, 1.0 ),  // magenta
        vec4 ( 1.0, 1.0, 1.0, 1.0 ),  // white
        vec4 ( 0.0, 1.0, 1.0, 1.0 )   // cyan
};
```

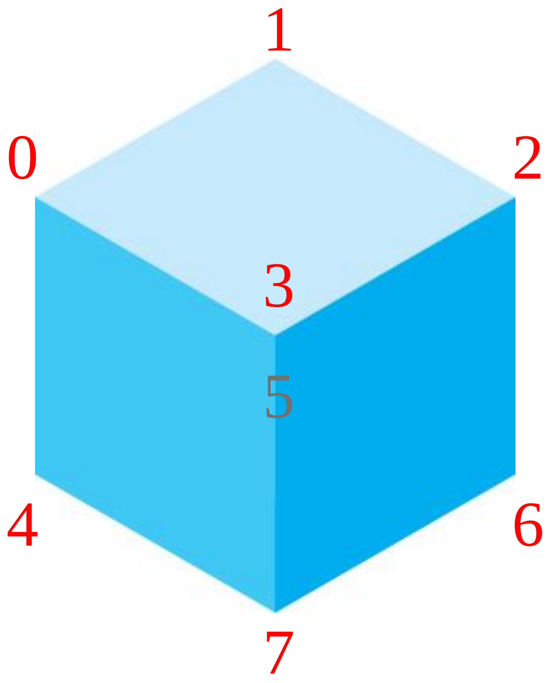
Generating a Cube Face from Vertices

```
// quad() generates two triangles for each face and assigns colors to the
  vertices
int Index = 0;  // global variable indexing into VBO arrays

void quad(int a, int b, int c, int d) {
    colors[Index] = vertex_colors[a]; points[Index] = vertex_pos[a]; Index++;
    colors[Index] = vertex_colors[b]; points[Index] = vertex_pos[b]; Index++;
    colors[Index] = vertex_colors[c]; points[Index] = vertex_pos[c]; Index++;

    colors[Index] = vertex_colors[a]; points[Index] = vertex_pos[a]; Index++;
    colors[Index] = vertex_colors[c]; points[Index] = vertex_pos[c]; Index++;
    colors[Index] = vertex_colors[d]; points[Index] = vertex_pos[d]; Index++;
}
```



Generating the Cube from Faces

```
// generate 12 triangles: 36 vertices and 36
 colors
void
colorcube() {
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}
```



Vertex Array Objects (VAOs)

- VAOs store the data of a geometric object
- Steps in using a VAO
  - generate VAO names by calling `glGenVertexArrays()`
  - bind a specific VAO for initialization by calling `glBindVertexArray()`
  - update VBOs associated with this VAO
  - bind VAO for use in rendering
- This approach allows a single function call to specify all the data for an objects
  - previously, you might have needed to make many calls to make all the data current

VAOs in Code

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

Storing Vertex Attributes

- Vertex data must be stored in a VBO, and associated with a VAO
- The code-flow is similar to configuring a VAO
  - generate VBO names by calling `glGenBuffers()`
  - bind a specific VBO for initialization by calling `glBindBuffer(GL_ARRAY_BUFFER, …)`
  - load data into VBO using `glBufferData(GL_ARRAY_BUFFER, …)`
  - bind VAO for use in rendering `glBindVertexArray()`

VBOs in Code

```
// Create and initialize a buffer object
GLuint buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(points) +
             sizeof(colors), NULL, GL_STATIC_DRAW);

glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(points), points);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(points),
             sizeof(colors), colors);
```

Loading Shaders

```
#include <InitShader.h>
```

```
// Load and use shaders
GLuint program
     = InitShader( "vshader.glsl", "fshader.glsl" );
glUseProgram( program );
```

glsl : opengl shader language
Those are provided in our homepage.

Connecting Vertex Shaders with Geometry

- Application vertex data enters the OpenGL pipeline through the vertex shader
- Need to connect vertex data to shader variables
  - requires knowing the attribute location
- Attribute location can either be queried by calling `glGetVertexAttribLocation()`

Vertex Array Code

```
// set up vertex arrays (after shaders are loaded)
GLuint vPosition = glGetAttribLocation(program, "vPosition”);
glEnableVertexAttribArray(vPosition);
glVertexAttribPointer(vPosition, 4, GL_FLOAT, GL_FALSE, 0,
  BUFFER_OFFSET(0));


GLuint vColor = glGetAttribLocation(program, "vColor”);
glEnableVertexAttribArray(vColor);
glVertexAttribPointer(vColor, 4, GL_FLOAT, GL_FALSE, 0,
  BUFFER_OFFSET(sizeof(points)));
```

Drawing Geometric Primitives

- For contiguous groups of vertices

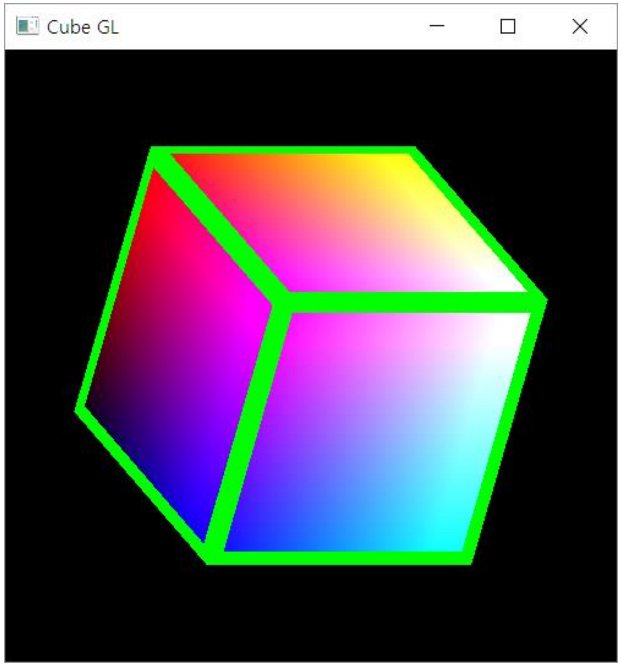
```
glDrawArrays(GL_TRIANGLES, 0, NumVertices);
```

- Usually invoked in display callback
- Initiates vertex shader

Summary

- We learnt the very basic of defining geometry and use it on GPU
- Today, we will learn how to code the shaders

- ***Vertex Shader*** :
  - Determining the positions of vertex
  - Useful for scaling, rotating, deforming and so on.
  - Also important for preparing some information sending to Fragment Shader
- ***Fragment Shader*** :
  - Determining the color of each fragment



GLSL Syntax Overview

- GLSL is like C without
  - Pointers
  - Recursion
  - Dynamic memory allocation
- GLSL is like C with
  - Built-in vector, matrix and sampler types
  - Constructors
  - A great math library
  - Input and output qualifiers

Allow us to write concise, efficient shaders.

GLSL Syntax

- GLSL has a preprocesssor

```
#version 430

#ifdef FAST_EXACT_METHOD
  FastExact();
#else
  SlowApproximate();
#endif

// ... many others
```

- All Shaders have main()

```
void main(void)
{
}
```

GLSL Data Types

Scalar types: float, int, bool

Vector types: vec2, vec3, vec4
ivec2, ivec3, ivec4
bvec2, bvec3, bvec4

Matrix types: mat2, mat3, mat4

Texture sampling: sampler1D, sampler2D, sampler3D, samplerCube

C++ style constructors: `vec3 a = vec3(1.0, 2.0, 3.0);`

Operators

- Standard C/C++ arithmetic and logic operators
- Operators overloaded for matrix and vector operations

```
mat4 m;
vec4 a, b, c;

b = a*m;
c = m*a;
```

Components and Swizzling

For vectors can use [ ], xyzw, rgba or stpq

Example:

```
vec3 v;
v[1], v.y, v.g, v.t
```
all refer to the same element

Swizzling:

```
vec3 a, b;
a.xy = b.yx;
```

GLSL Syntax: Vectors

Constructors

```
vec3 xyz = vec3(1.0, 2.0, 3.0);

vec3 xyz = vec3(1.0); // [1.0, 1.0, 1.0]

vec3 xyz = vec3(vec2(1.0, 2.0), 3.0);
```

~~vec3 xyz (1.0, 2.0, 3.0);~~   // ← Error!

GLSL Syntax: Vectors

```
vec4 c = vec4(0.5, 1.0, 0.8, 1.0);

vec3 rgb = c.rgb;  // [0.5, 1.0, 0.8]

vec3 bgr = c.bgr;  // [0.8, 1.0, 0.5]

vec3 rrr = c.rrr;  // [0.5, 0.5, 0.5]

c.a = 0.5;         // [0.5, 1.0, 0.8, 0.5]
c.rb = 0.0;        // [0.0, 1.0, 0.0, 0.5]

float g = rgb[1];  // 0.5, indexing, not swizzling
```

GLSL Syntax: Matrices

- Matrices are built-in types:
  - Square: `mat2`, `mat3`, `mat4`
  - Rectangular: `mat`*`m`*`x`*`n`* (`m` columns, `n` rows)
    `mat2x3`, `mat3x4`, `mat4x4`
- Stored <u>*column major*</u>

GLSL Syntax: Matrices

- Constructors

```
mat3 i = mat3(1.0); // 3x3 identity matrix

mat2 m = mat2(1.0, 2.0,   // [1.0 3.0]
              3.0, 4.0); // [2.0 4.0]
```

Column major!

- Accessing elements

```
float f = m[column][row];

float x = m[0].x; // x component of first column

vec2 yz = m[1].yz; // yz components of second column
```

Treat matrix as array of column vectors

Can swizzle too!

GLSL Syntax: Vectors and Matrices

- Matrix and vector operations are easy and fast:

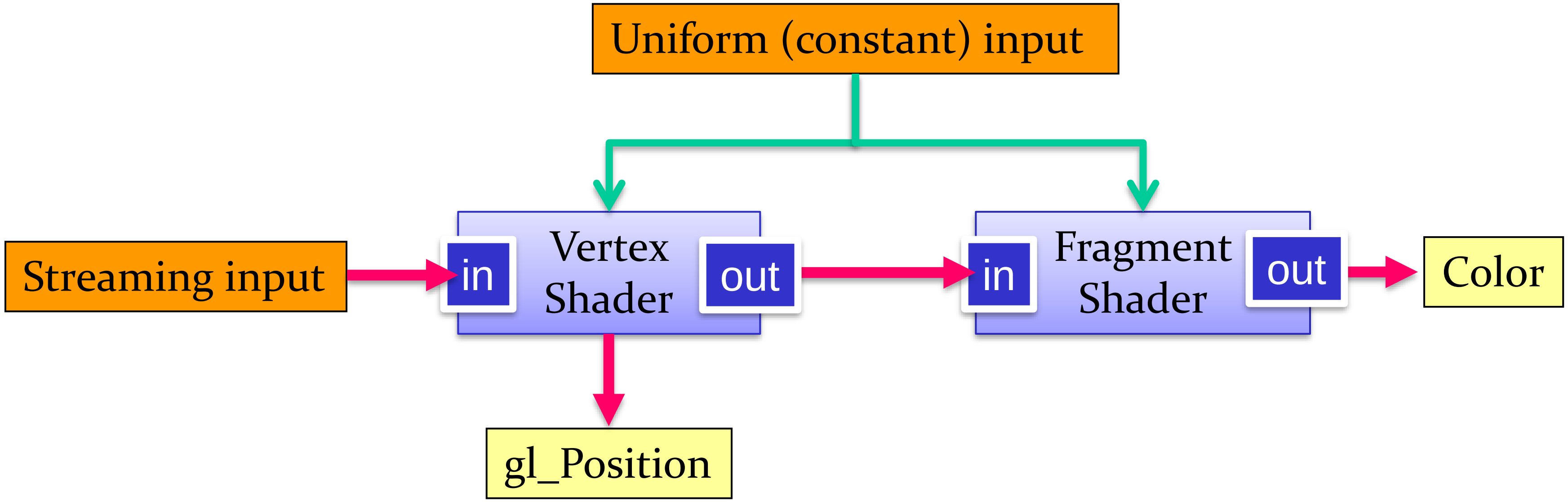
```
vec3 xyz = // ...

vec3 v0 = 2.0 * xyz; // scale
vec3 v1 = v0 + xyz;  // component-wise
vec3 v2 = v0 * xyz;  // component-wise

mat3 m = // ...
mat3 t = // ...

mat3 mt = t * m;       // matrix * matrix
vec3 xyz2 = mt * xyz; // matrix * vector
vec3 xyz3 = xyz * mt; // vector * matrix  ( = transposed_matrix * vector )
```

- For more information:
  - http://en.wikibooks.org/wiki/GLSL_Programming/Vector_and_Matrix_Operations
  - http://en.wikibooks.org/wiki/GLSL_Programming/Applying_Matrix_Transformations

Qualifiers

- in, out
  - Copy vertex attributes and other variables to/from shaders
  - `in vec2 tex_coord;`
  - `out vec4 color;`
- Uniform: variable from application
  - `uniform float time;`
  - `uniform vec4 rotation;`

GLSL Syntax: `in`/ `out`/ `uniform`



GLSL Syntax: `in`/ `out`/ `uniform`

- Example

```
#version 430

uniform mat4 u_ModelView;
in vec3 Position;
in vec3 Color;
out vec3 fs_Color;

void main(void)
{
  fs_Color = Color;
  gl_Position = u_ModelView * vec4(Position, 1.0);
}
```

`uniform`: shader input constant across `glDraw`

`in`: shader input varies per vertex attribute

`out`: shader output

Flow Control

- if
- if else
- expression ? true-expression : false-expression
- while, do while
- for

Functions

- Built in
  - Arithmetic: sqrt, power, abs
  - Trigonometric: sin, asin
  - Graphical: length, reflect
- User defined

Built-in Variables

- `gl_Position`: output position from vertex shader

- `gl_FragColor`: output color from fragment shader
    - Only for ES, WebGL and older versions of GLSL
    - Present version use an out variable

Simple Vertex Shader for Cube

```
#version 330

in vec4 vPosition;
in vec4 vColor;
out vec4 color;

void main() {
    color = vColor;
    gl_Position = vPosition;
}
```

The Simplest Fragment Shader

```
#version 330

in vec4 color;
out vec4 FragColor;

void main() {
    FragColor = color;
}
```

Using Shaders: A Simpler Way

- We've created a routine for this course to make it easier to load your shaders
  - available at course website:

```
GLuint InitShader (char* vFile, char* fFile);
```

- `InitShaders` takes two filenames
  - `vFile` for the vertex shader
  - `fFile` for the fragment shader
- Fails if shaders don't compile, or program doesn't link

Determining Locations After Linking

Assumes you already know the variables' name

```
GLint  idx =
 glGetAttribLocation(program, "name");

GLint  idx =
 glGetUniformLocation(program, "name");
```

Vertex Attributes Setting

```
// set up vertex arrays (after shaders are loaded)
GLuint vPosition = glGetAttribLocation(program, "vPosition”);
glEnableVertexAttribArray(vPosition);
glVertexAttribPointer(vPosition, 4, GL_FLOAT, GL_FALSE, 0,
  BUFFER_OFFSET(0));


GLuint vColor = glGetAttribLocation(program, "vColor”);
glEnableVertexAttribArray(vColor);
glVertexAttribPointer(vColor, 4, GL_FLOAT, GL_FALSE, 0,
  BUFFER_OFFSET(sizeof(points)));
```

Initializing Uniform Variable Values

Uniform Variables

```
glUniform1f(index, value);

glUniform4f(index, x, y, z, w);

Glboolean transpose = GL_TRUE;
    // Since we're C programmers
Glfloat mat[3][4][4] = { … };
```

row column

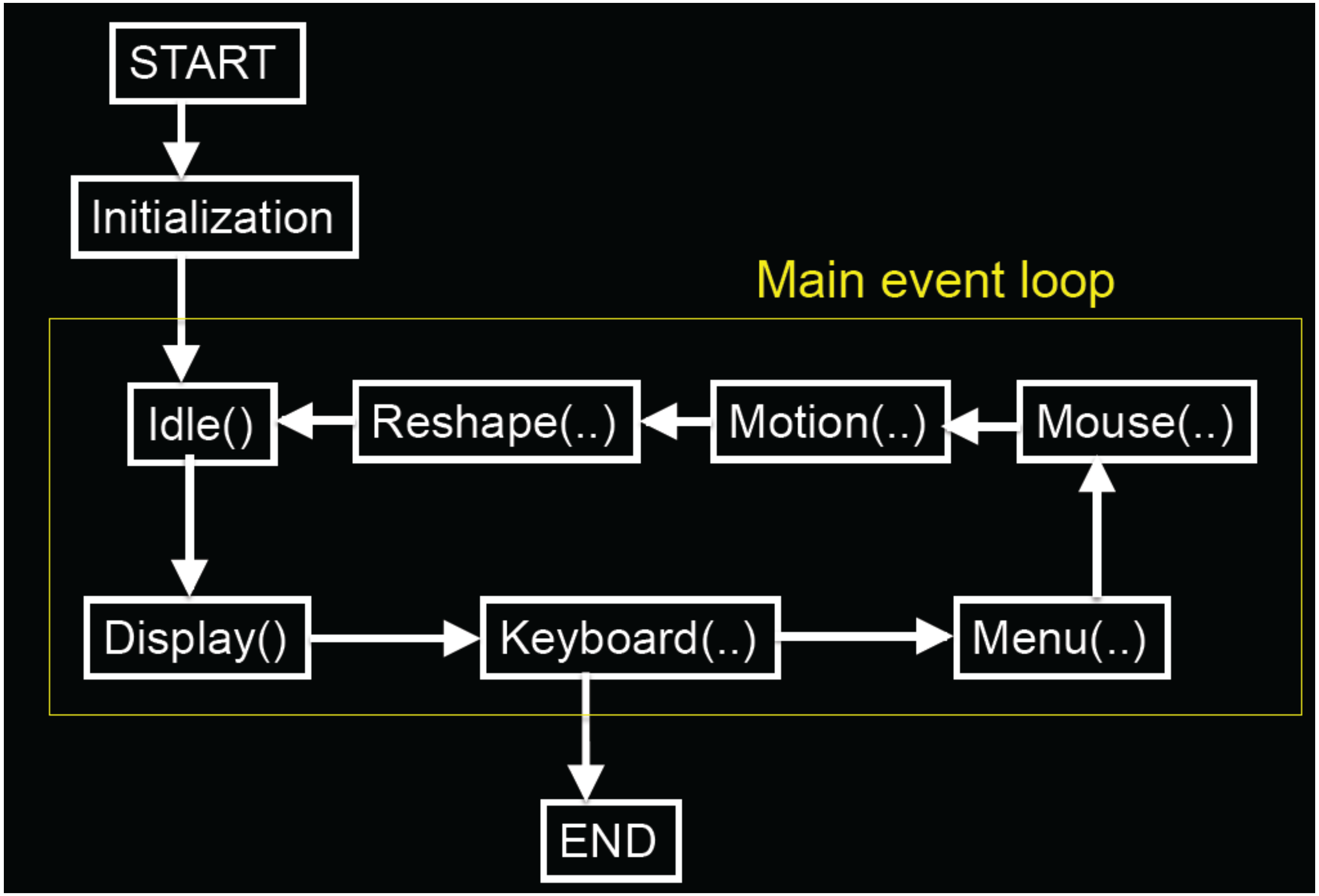
```
glUniformMatrix4fv(index, 3, transpose, mat);
```

Shader Applications

- Moving vertices
  - Morphing
  - Wave motion
  - Fractals
- Lighting
  - More realistic models
  - Cartoon shaders
- And more and more!

Interaction: Callbacks

GLUT Program with Callbacks



Event Types

- Window: resize, expose, iconify
- Mouse: click one or more buttons
- Motion: move mouse
- Keyboard: press or release a key
- Idle: non-event
  - Define what should be done if no other event is in queue

Callbacks

- Programming interface for event-driven input
- Define a *callback function* for each type of event the graphics system recognizes
- This user-supplied function is executed when the event occurs
- GLUT example: **`glutMouseFunc(mymouse)`**

mouse callback function

GLUT callbacks

GLUT recognizes a subset of the events recognized by any particular window system (Windows, X, Macintosh)

- **`glutDisplayFunc`**
- **`glutMouseFunc`**
- **`glutReshapeFunc`**
- **`glutKeyboardFunc`**
- **`glutIdleFunc`**
- **`glutMotionFunc`, `glutPassiveMotionFunc`**

Types of Callbacks

- Display ( ) : when window must be drawn
- Idle ( ) : when no other events to be handled
- Keyboard (unsigned char key, int x, int y) : key pressed
- Menu (...) : after selection from menu
- Mouse (int button, int state, int x, int y) : mouse button
- Motion (...) : mouse movement
- Reshape (int w, int h) : window resize
- Any callback can be NULL

GLUT Event Loop

- Recall that the last line in **`main.c`** for a program using GLUT must be

  `glutMainLoop();`

which puts the program in an infinite event loop

- In each pass through the event loop, GLUT
  - looks at the events in the queue
  - for each event in the queue, GLUT executes the appropriate callback function if one is defined
  - if no callback is defined for the event, the event is ignored

Example : Idling Callback

- Idling Callback is useful for defining a periodic work
  - Ex.) making an animation (rotating a cube and so on)

- How to use:
  1. Registering your function as an Idling Callback Function

```
glutIdleFunc ( myIdle );
```

  2. Implement what you want to do repeatedly

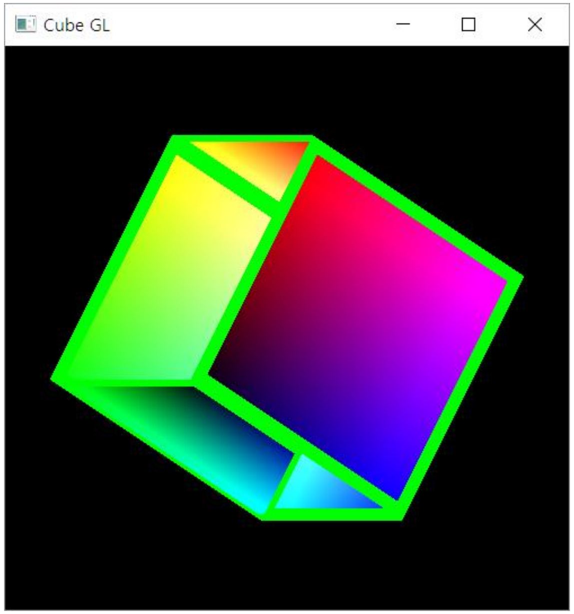
```
void myIdle()
{
        // do some periodic job
        Sleep(16);
        glutPostRedisplay();
}
```

Wait for 16 mille-sec.
(Ensuring 60 FPS)

Invoking redrawing

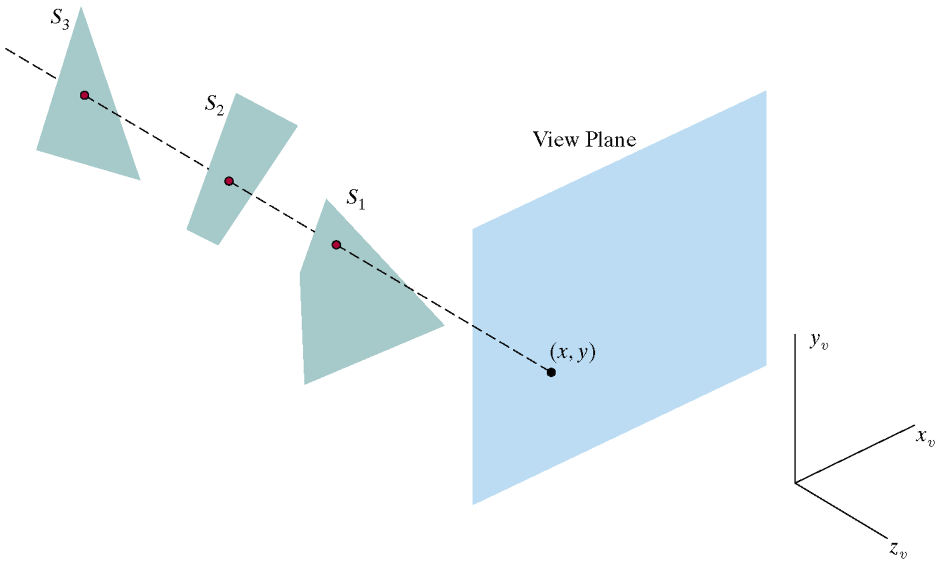
One little problem

- Incorrect Depth Handling



Depth-Buffer (Z-Buffer)

- ***Z-Buffer*** has memory corresponding to each pixel location for storing the current depth value (distance from view plane)
  - Useful for determining whether a new drawing pixel is visible or not
    - Is ***visible*** when the its distance is closer than the previously stored one
    - Is ***not visible*** when it is farer



Enabling the Depth Buffer in OPENGL

- On Initialization :

```
glutInitDisplayMode(GLUT_DOUBLE |
                    GLUT_RGBA |
                    GLUT_DEPTH);
```

- On Drawing :

```
glClear (GL_COLOR_BUFFER_BIT |
         GL_DEPTH_BUFFER_BIT);
glEnable(GL_DEPTH_TEST);
```