Computer Graphics (CS 4731)
Lecture 3: Introduction to OpenGL/GLUT (Part 2)

Prof Emmanuel Agu

Computer Science Dept.

Worcester Polytechnic Institute (WPI)

Recall: 1. Generate Points to be Drawn
2. Store in an array

- Generate points & store vertices into an array

```
point2 points[NumPoints];

points[0] = point2( -0.5, -0.5 );
points[1] = point2( 0.0, 0.5 );
points[2] = point2( 0.5, -0.5 );
```

Recall: 3. Create GPU Buffer for Vertices

- Rendering from GPU memory significantly faster. Move data there
- Fast GPU (off-screen) memory for data called ***Buffer Objects***
- An array of buffer objects (called ***vertex array object***) are usually created
- So, first create the vertex array object

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

Recall: 3. Create GPU Buffer for Vertices

- Next, create a buffer object in two steps
  1. Create VBO and give it name (unique ID number)

```
GLuint buffer;
glGenBuffers(1, &buffer);  // create one buffer object
```

Number of Buffer Objects to return

  2. Make created VBO currently active one

```
glBindBuffer(GL_ARRAY_BUFFER, buffer);  //data is array
```

Recall: 4. Move points GPU memory

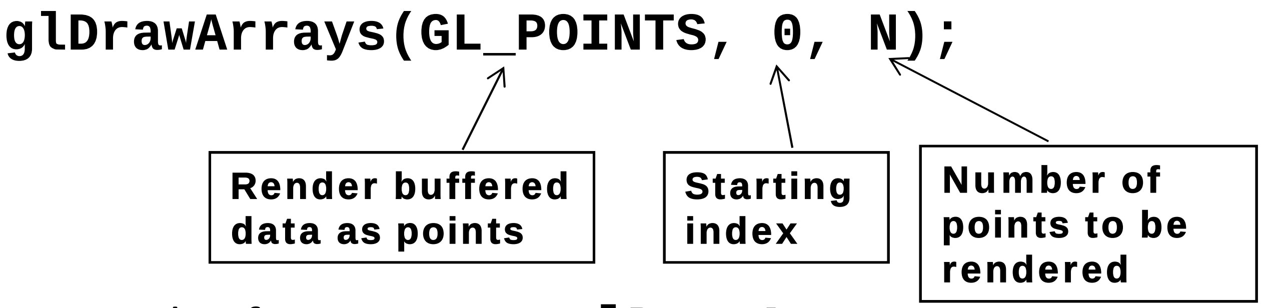
3. Move **`points`** generated earlier to VBO

```
glBufferData(GL_ARRAY_BUFFER, buffer, sizeof(points),
points, GL_STATIC_DRAW ); //data is array
```

Data to be transferred to GPU memory (generated earlier)

- **`GL_STATIC_DRAW:`** buffer object data will be specified once by application and used many times to draw
- **`GL_DYNAMIC_DRAW:`** buffer object data will be specified repeatedly and used many times to draw

Recall: 5. Draw points (from VBO)



```
glDrawArrays(GL_POINTS, 0, N);
```

Render buffered data as points

Starting index

Number of points to be rendered

- Display function using **`glDrawArrays:`**

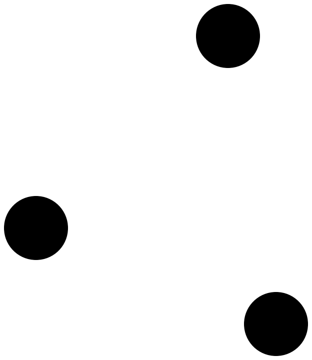
```
void mydisplay(void){
      glClear(GL_COLOR_BUFFER_BIT); // clear screen
      glDrawArrays(GL_POINTS, 0, N);
      glFlush( );         // force rendering to show
}
```

- Other possible arguments to **`glDrawArrays`** instead of **`GL_POINTS?`**

glDrawArrays() Parameters

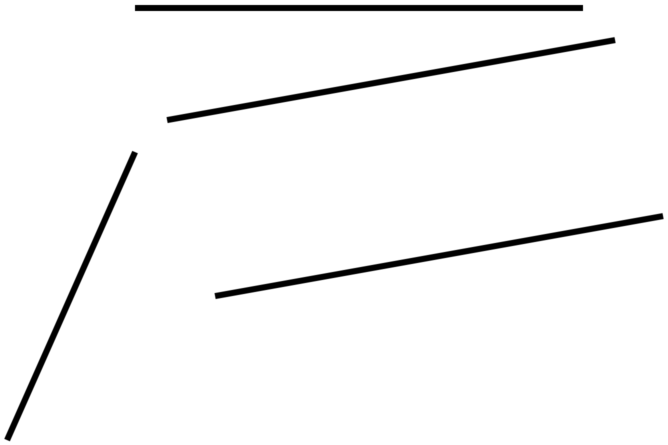
glDrawArrays(GL_POINTS, ….)

– draws dots



glDrawArrays((GL_LINES, …)

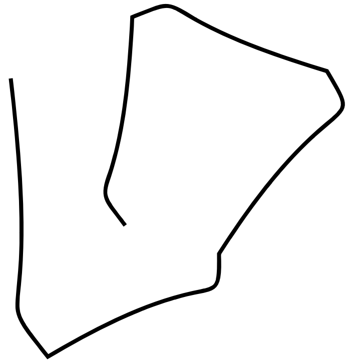
– Connect vertex pairs to draw lines



`glDrawArrays()` Parameters

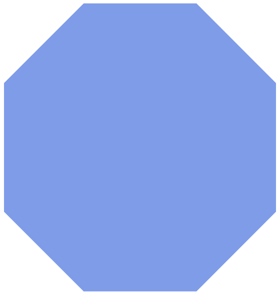
`glDrawArrays`(GL_LINE_STRIP,..)

– polylines



`glDrawArrays`(GL_POLYGON,..)

– convex filled polygon

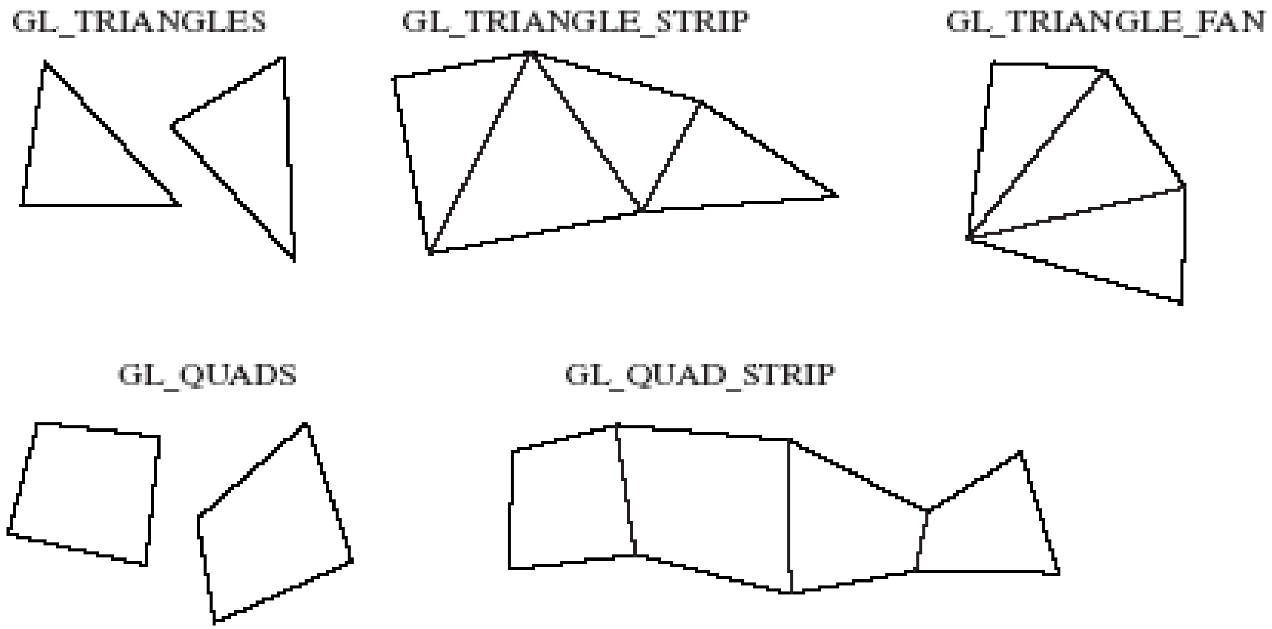


`glDrawArrays`(GL_LINE_LOOP)

– Close loop of polylines
(Like GL_LINE_STRIP but closed)

`glDrawArrays()` Parameters

- Triangles: Connect 3 vertices
  - GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN
- Quad: Connect 4 vertices
  - GL_QUADS, GL_QUAD_STRIP

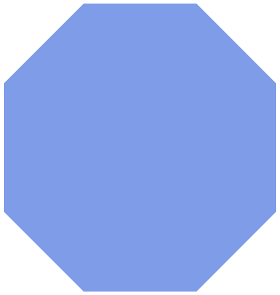


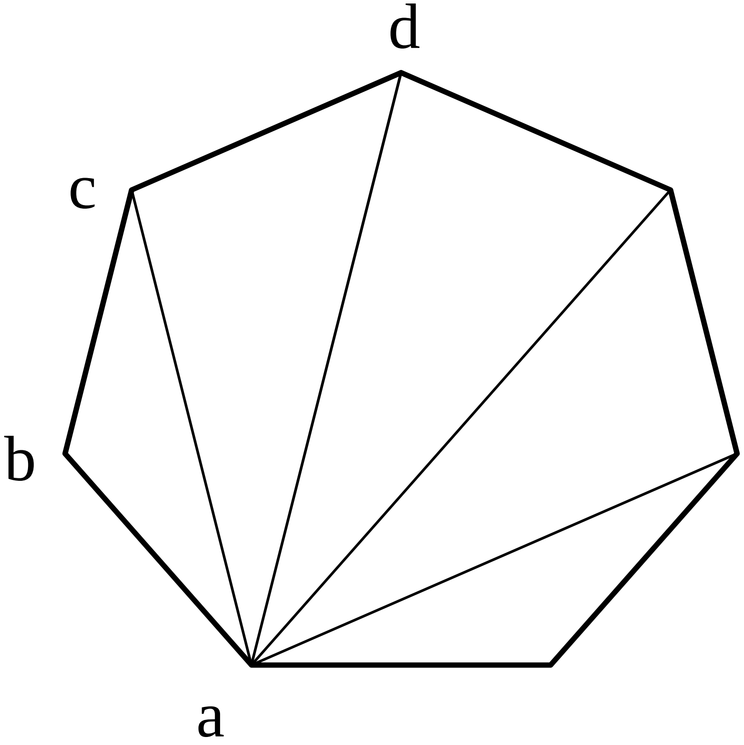
Triangulation

- Generally OpenGL breaks polygons down into triangles which are then rendered. Example

`glDrawArrays`(GL_POLYGON,..)

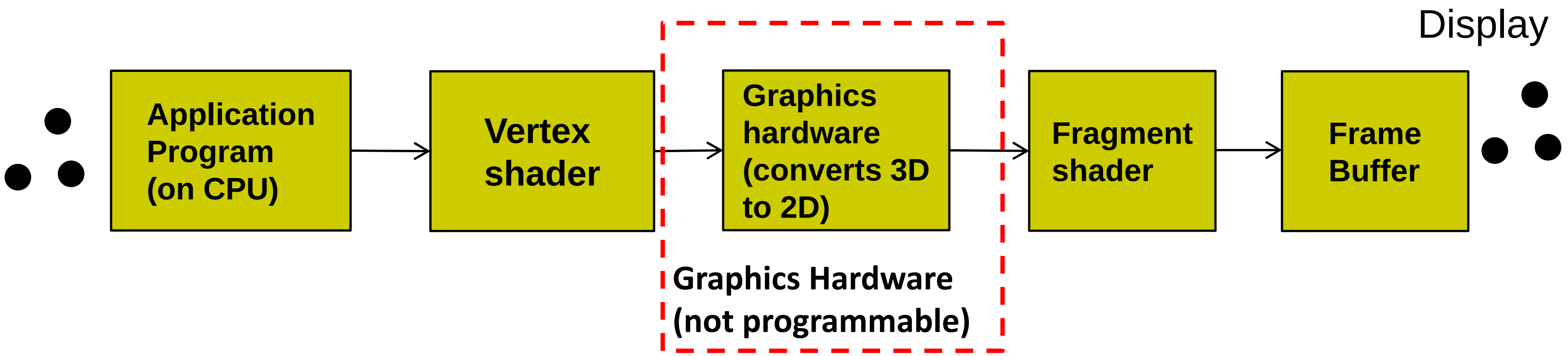
– convex filled polygon





What other Initialization do we Need?

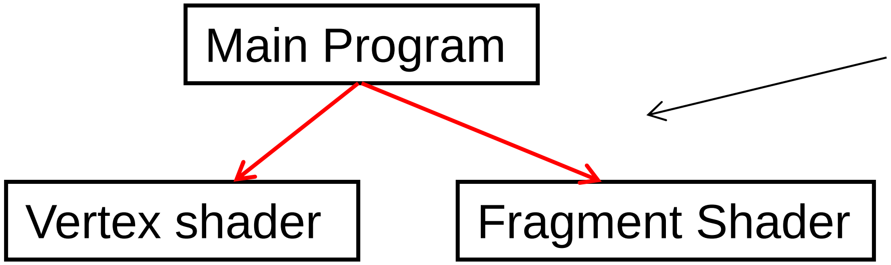
- Also set clear color and other OpenGL parameters
- Also set up shaders as part of initialization
  - Read, compile, link
- Also need to specify two shaders:
  - **Vertex shader:** program that is run once on **each vertex**
  - **Fragment shader:** program that is run once on **each pixel**
- Need to connect **.cpp file** to **vertex shader** and **fragment shader**


Display
Application Program (on CPU)
Vertex shader
Graphics hardware (converts 3D to 2D)
Fragment shader
Frame Buffer
Graphics Hardware (not programmable)

OpenGL Program: Shader Setup

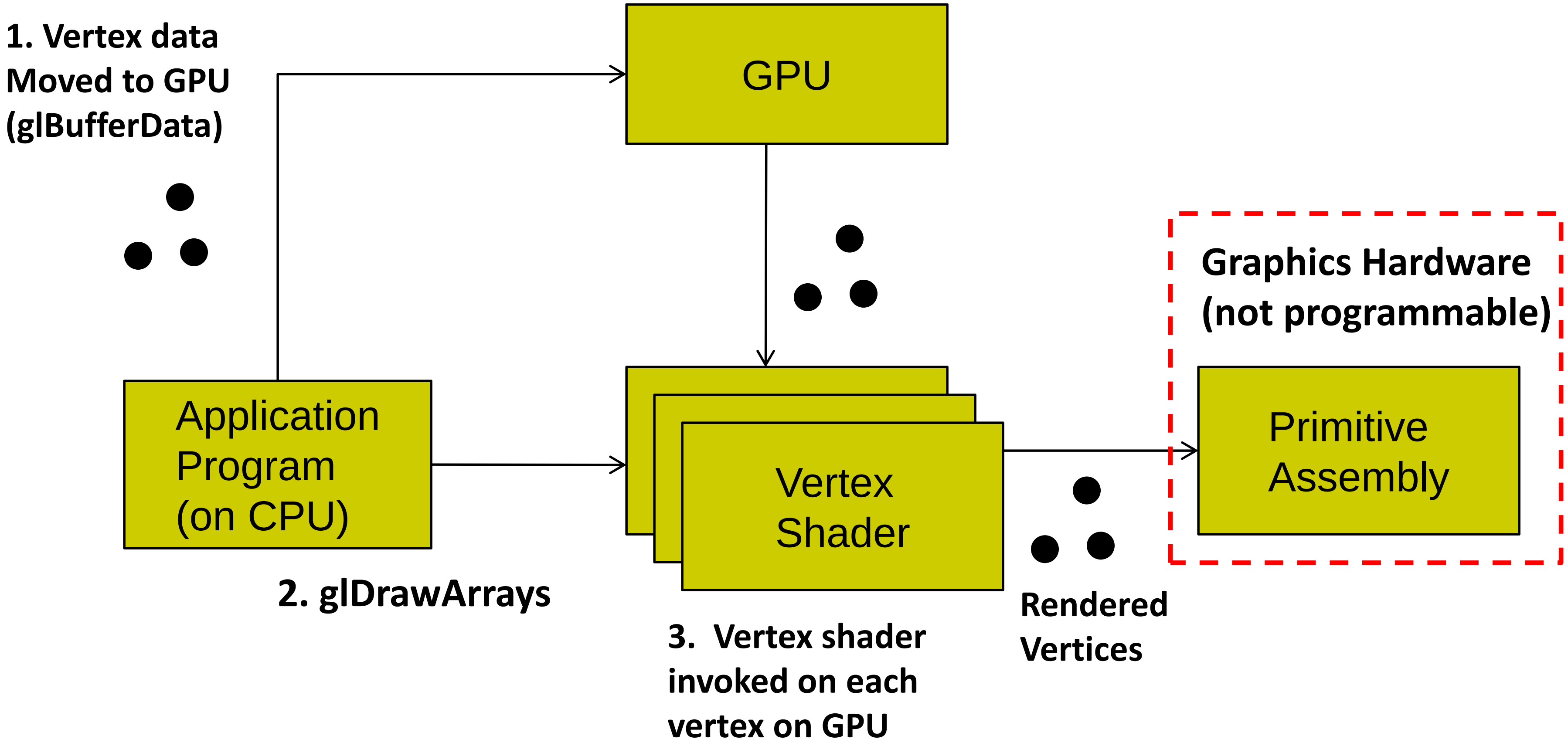
- OpenGL programs now have 3 parts:
  - Main **OpenGL program** (.cpp file), **vertex shader** (e.g. vshader1.glsl), and **fragment shader** (e.g. fshader1.glsl) in same Windows directory
  - In main program, need to link names of vertex, fragment shader
  - **initShader( )** is homegrown shader initialization function. More later

```
GLuint = program;
GLuint program = InitShader( "vshader1.glsl", "fshader1.glsl" );
glUseProgram(program);
```



initShader( )
Homegrown, connects main
Program to shader files
More on this later!!

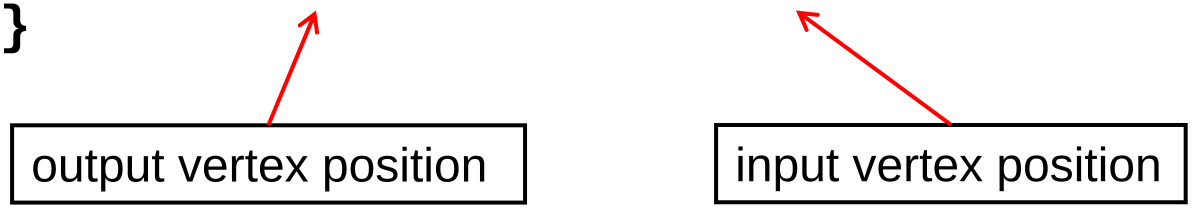
Execution Model



Vertex Shader

- We write a simple "pass-through" shader (does nothing)
- Simply sets output vertex position to received input position
- **`gl_Position`** is built in variable (already declared)

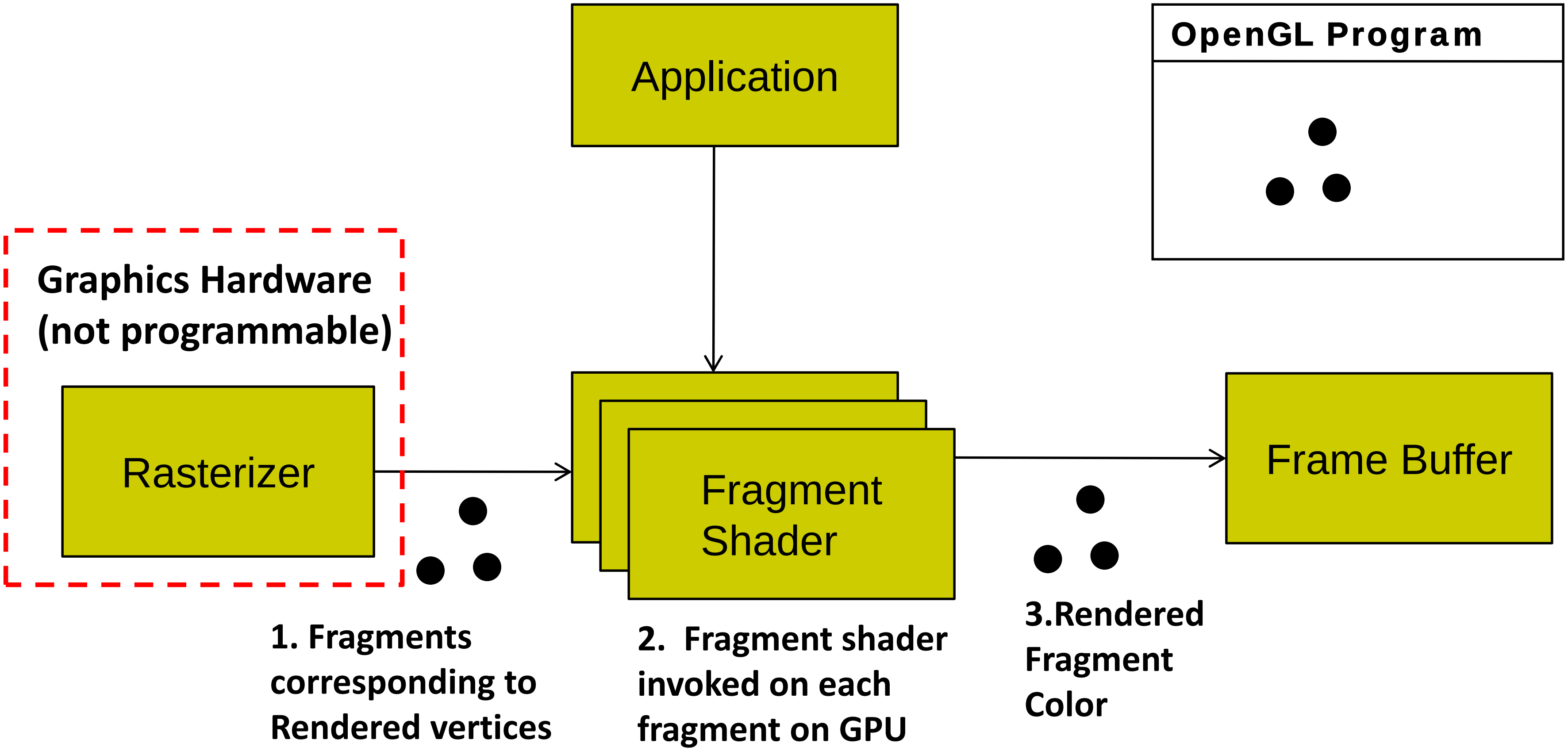
```
in vec4 vPosition

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



output vertex position

input vertex position

Execution Model



Application

OpenGL Program

Graphics Hardware (not programmable)

Rasterizer

Fragment Shader

Frame Buffer

1. Fragments corresponding to Rendered vertices

2. Fragment shader invoked on each fragment on GPU

3.Rendered Fragment Color

Fragment Shader

- We write a simple fragment shader (sets color to red)
- **`gl_FragColor`** is built in variable (already declared)

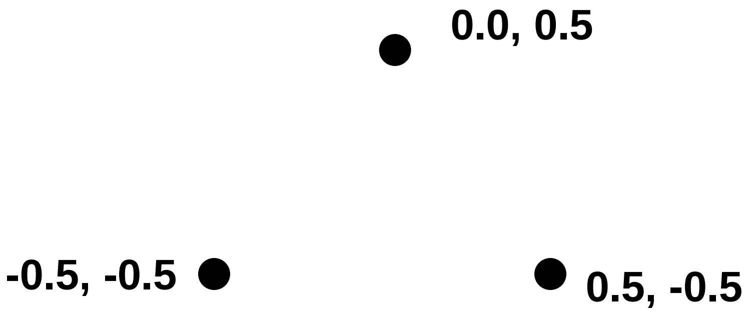
```
void main( )
{
        gl_FragColor = vec(1.0, 0.0, 0.0, 1.0);
}
```

Set each drawn fragment color to red

Previously: Generated 3 Points to be Drawn

- Stored points in array **points[**], moved to GPU, draw using **`glDrawArray`**

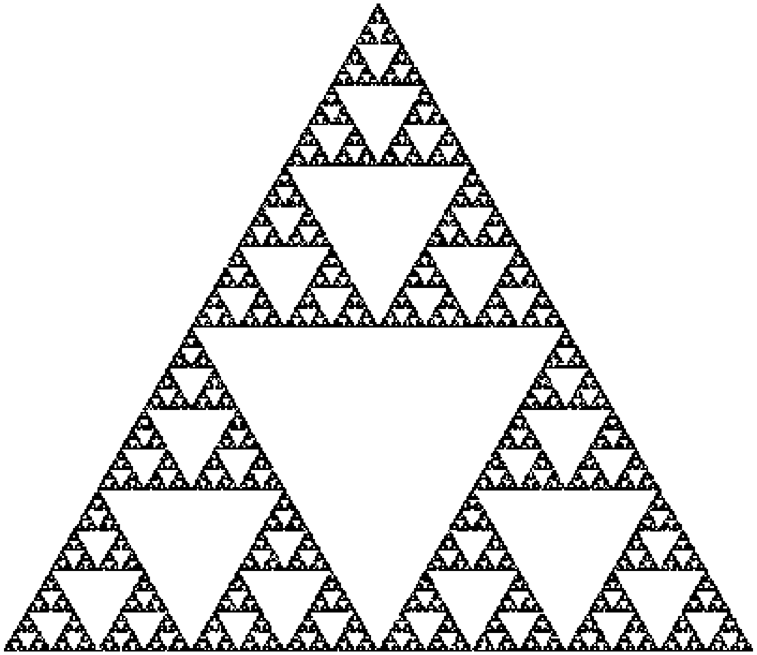
```
point2 points[NumPoints];

points[0] = point2( -0.5, -0.5 );
points[1] = point2( 0.0, 0.5 );
points[2] = point2( 0.5, -0.5 );
```



- Once drawing steps are set up, can generate more complex sequence of points algorithmically, drawing steps don’t change
- Next: example of more algorithm to generate more complex point sequences

Sierpinski Gasket Program

- Any sequence of points put into array points[ ] will be drawn
- Can generate interesting sequence of points
  - Put in array points[ ], draw!!
- Sierpinski Gasket: Popular fractal



Sierpinski Gasket

Start with initial triangle with corners (*x1, y1, 0*), (*x2, y2, 0*) and (*x3, y3, 0*)

1. Pick initial point **p** = (*x, y, 0*) at random inside a triangle
2. Select on of 3 vertices at random
3. Find **q,** halfway between **p** and randomly selected vertex
4. Draw dot at **q**
5. Replace **p** with **q**
6. Return to step 2

Actual Sierpinski Code

```
#include “vec.h”    // include point types and operations
#include <stdlib.h> // includes random number generator

void Sierpinksi( )
{
    const int NumPoints = 5000;
    vec2 points[NumPoints];

    // Specifiy the vertices for a triangle
    vec2 vertices[3] = {
        vec2( -1.0, -1.0 ), vec2( 0.0, 1.0 ), vec2( 1.0, -1.0 )
    };
```

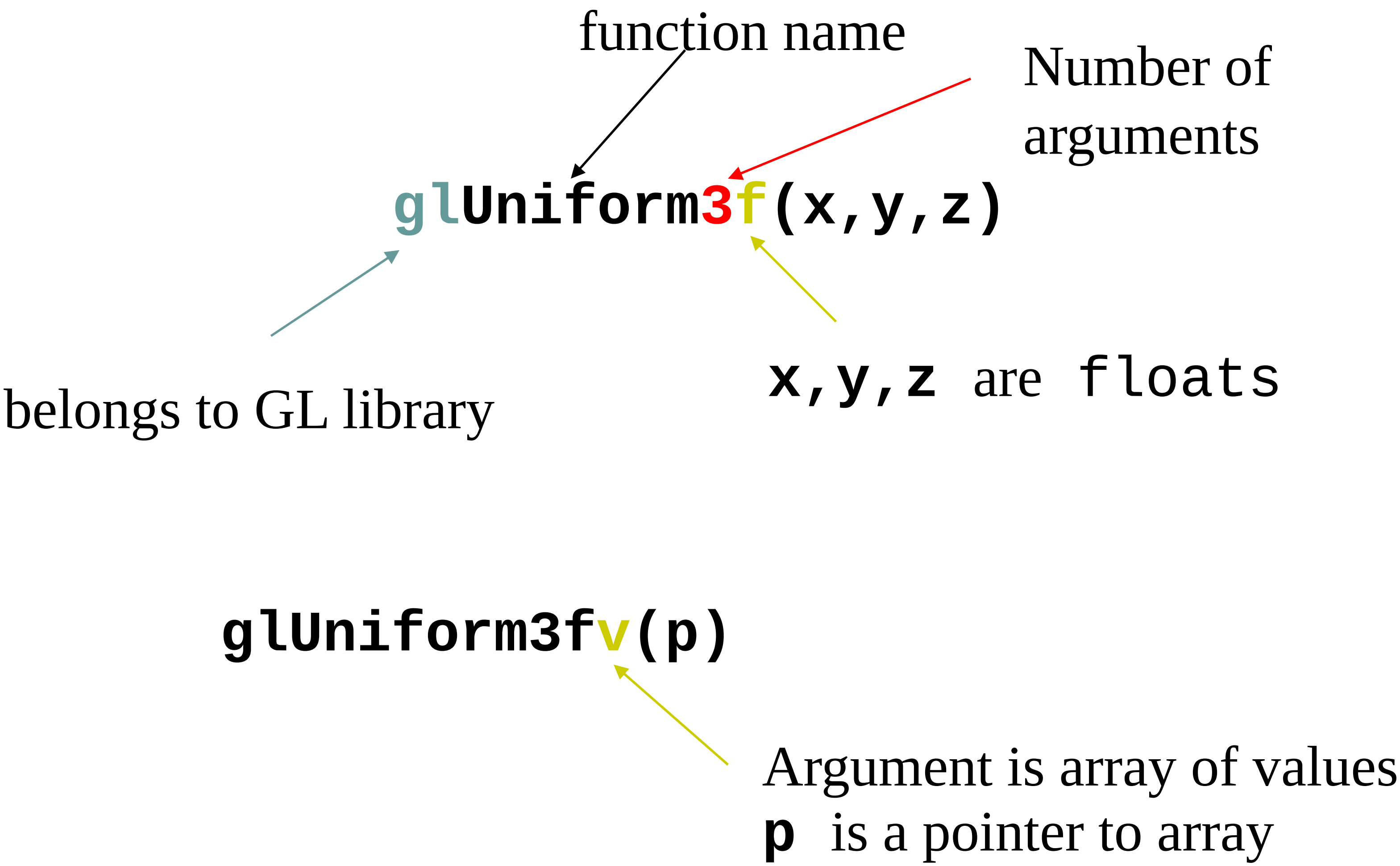
Actual Sierpinski Code

```
// An arbitrary initial point inside the triangle
points[0] = point2(0.25, 0.50);

// compute and store N-1 new points
for ( int i = 1; i < NumPoints; ++i ) {
    int j = rand() % 3;   // pick a vertex at random

    // Compute the point halfway between the selected vertex
    //   and the previous point
    points[i] = ( points[i - 1] + vertices[j] ) / 2.0;
}
```

Lack of Object Orientation

- OpenGL is not object oriented
- Multiple versions for each command
  - `glUniform3f`
  - `glUniform2i`
  - `glUniform3dv`

OpenGL function format



function name

Number of arguments

`glUniform3f(x,y,z)`

belongs to GL library

`x,y,z` are `floats`

`glUniform3fv(p)`

Argument is array of values
`p` is a pointer to array

Recall: Single Buffering

- If display mode set to single framebuffers
- Any drawing into framebuffer is seen by user. How?
  - `glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);`
    - Single buffering with RGB colors
- Drawing may not be drawn to screen until call to `glFlush( )`

```
void mydisplay(void){
      glClear(GL_COLOR_BUFFER_BIT); // clear screen
      glDrawArrays(GL_POINTS, 0, N);
      glFlush( );
}
```

Drawing sent to screen

Double Buffering

- Set display mode to double buffering (create front and back framebuffers)
  - `glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);`
    - Double buffering with RGB colors

- Front buffer displayed on screen, back buffers not displayed
- Drawing into back buffers (not displayed) until swapped in using `glutSwapBuffers( )`

```
void mydisplay(void){
      glClear(GL_COLOR_BUFFER_BIT); // clear screen
      glDrawArrays(GL_POINTS, 0, N);
      glutSwapBuffers( );
}
```

Back buffer drawing swapped in, becomes visible here

OpenGL Data Types

C++	**OpenGL**
Signed char	GLByte
Short	GLShort
Int	GLInt
Float	GLFloat
Double	GLDouble
Unsigned char	GLubyte
Unsigned short	GLushort
Unsigned int	GLuint

Example: Integer is 32-bits on 32-bit machine
but 64-bits on a 64-bit machine

Adding Interaction

- So far, OpenGL programs just render images
- Can add user interaction
- Examples:
  - User hits ‘h’ on keyboard -> Program draws house
  - User clicks mouse left button -> Program draws table

Types of Input Devices

- **String:** produces string of characters e.g. keyboard

- **Locator:** User points to position on display. E.g mouse

Types of Input Devices

- **Valuator:** generates number between 0 and 1.0
- **Pick:** User selects location on screen (e.g. touch screen in restaurant, ATM)

Using Keyboard Callback for Interaction

- 1. register callback in main( ) function

```
glutKeyboardFunc( myKeyboard );
```

ASCII character
of pressed key

x,y location
of mouse

- 2. implement keyboard function

```
void myKeyboard(char key, int x, int y )
{    // put keyboard stuff here
    ……….
     switch(key){    // check which key
         case ‘f’:
            // do stuff
         break;

          case ‘k’:
             // do other stuff
         break;

     }
…………..
}
```

Note: Backspace, delete, escape keys checked using their ASCII codes

Keyboard Interaction

- For function, arrow and other special-purpose keys, use

```
glutSpecialFunc (specialKeyFcn);
…
Void specialKeyFcn (Glint specialKey, GLint, xMouse,
                                    Glint yMouse)
```

- Example: if (`specialKey == GLUT_KEY_F1)`// F1 key pressed
  - `GLUT_KEY_F1, GLUT_KEY_F12, ….` for function keys
  - `GLUT_KEY_UP, GLUT_KEY_RIGHT, ….` for arrow keys keys
  - `GLUT_KEY_PAGE_DOWN, GLUT_KEY_HOME, ….` for page up, home keys
- Complete list of special keys designated in `glut.h`

Mouse Interaction

- Declare prototype
  - `myMouse(int button, int state, int x, int y)`
  - `myMovedMouse`
- Register callbacks:
  - `glutMouseFunc(myMouse):` mouse button pressed
  - `glutMotionFunc(myMovedMouse):` mouse moves with button pressed
  - `glutPassiveMotionFunc(myMovedMouse):` mouse moves with no buttons pressed
- Button returned values:
  - GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, GLUT_RIGHT_BUTTON
- State returned values:
  - GLUT_UP, GLUT_DOWN
- X,Y returned values:
  - x,y coordinates of mouse location

Mouse Interaction Example

- Each mouse click generates separate events
- Store click points in **global** or **static** variable in mouse function
- **Example:** draw (or select) rectangle on screen
- Mouse y returned assumes y=0 at top of window
- OpenGL assumes y=0 at bottom of window. Solution? Flip mouse y

```
void myMouse(int button, int state, int x, int y)
{
   static GLintPoint corner[2];
   static int numCorners = 0;   // initial value is 0
   if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
   {
        corner[numCorners].x = x;
        corner[numCorners].y = screenHeight – y; //flip y coord
        numCorners++;
```

Screenheight is height of drawing window

Mouse Interaction Example (continued)

```
   if(numCorners == 2)
   {
        // draw rectangle or do whatever you planned to do
        Point3 points[4] = corner[0].x, corner[0].y,
                           corner[1].x, corner[0].y,
                           corner[1].x, corner[1].y,
                            corner[0].x, corner[1].y);

         glDrawArrays(GL_QUADS, 0, 4);

        numCorners == 0;
   }
   else if(button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
        glClear(GL_COLOR_BUFFER_BIT); // clear the window
   glFlush( );
}
```

References

- Angel and Shreiner, Interactive Computer Graphics, 6th edition, Chapter 2
- Hill and Kelley, Computer Graphics using OpenGL, 3rd edition, Chapter 2