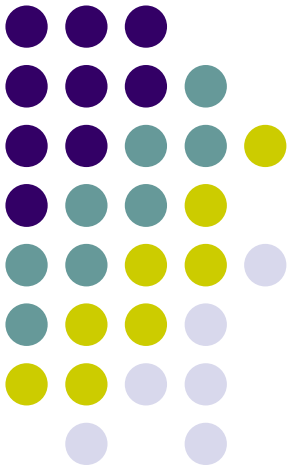
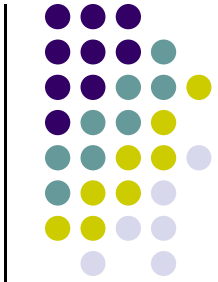
# Computer Graphics
# CS 543 – Lecture 2 (Part 1)
# Intro to GLSL (Part 2)

Prof Emmanuel Agu

*Computer Science Dept.*

*Worcester Polytechnic Institute (WPI)*





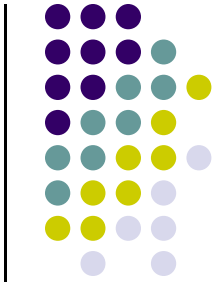
# Recall: Generated Points

- Generated points & stored vertices into an array

```
point3 points[3] = { point2(100,50),
                     point2(100,130),
                     point2(150, 130); }
```

- Draw points from array using **`glDrawArrays`**

# Display Callback

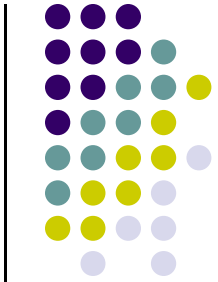


- Once we get data to GPU, initiate rendering with simple callback

```
void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawArrays(GL_TRIANGLES, 0, 3);
    glFlush();
}
```

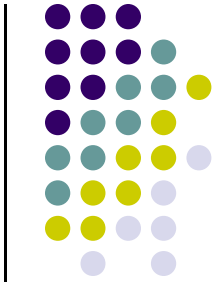
- Arrays are buffer objects that contain vertex arrays

# Vertex Arrays



- Vertices can have many attributes
  - Position (x, y, z)
  - Color (R,G,B)
  - Texture Coordinates
  - Application data
- A vertex array holds these data
- Using types in **`vec.h`**

```
point2 vertices[3] = {point2(0.0, 0.0),
      point2( 0.0, 1.0), point2(1.0, 1.0)};
```



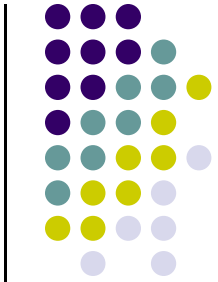
# Recall: Storing Vertices

- Generate points & store vertices into an array

```
point3 points[3] = { point2(100,50),
                     point2(100,130),
                     point2(150, 130); }
```

- Draw points from array using **`glDrawArrays`**

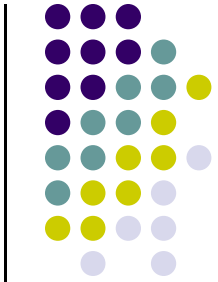
# Vertex Array Object



- Bundles all vertex data (positions, colors, ..,)
- Get name for buffer then bind

```
Glunit abuffer;
glGenVertexArrays(1, &abuffer);
glBindVertexArray(abuffer);
```

- At this point we have an ***empty*** current vertex array
- We can bind VAO to different buffers
- glBindVertexArray lets us switch between VBOs



# Recall: Move points GPU memory

- Rendering from GPU memory significantly faster. Move data there
- Fast GPU memory for data called ***Buffer Objects***
- Three steps:
  1. Create VBO and give it name (unique ID number)

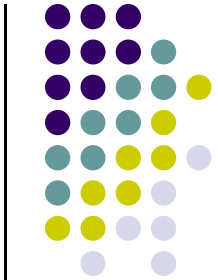
```
GLuint buffer;
glGenBuffers(1, &buffer); // create one buffer object
```

**Number of Buffer Objects to return**

  2. Make created VBO currently active one

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

# Recall: Move points GPU memory

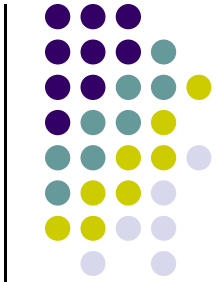


3. Move **`points`** generated earlier to VBO (Data in current vertex array is sent to GPU)

```
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: Draw points

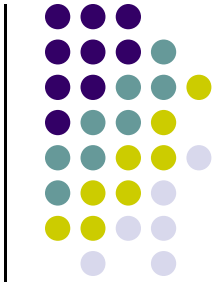
```
glDrawArrays(GL_POINTS, 0, N);
```

**Render buffered data as points**

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

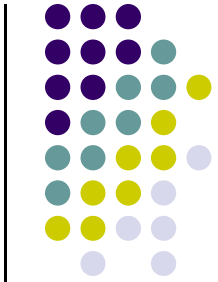
```
void mydisplay(void){
      glClear(GL_COLOR_BUFFER_BIT); // clear screen
      glDrawArrays(GL_POINTS, 0, N);
      glFlush( );         // force rendering to show
}
```

# Initialization

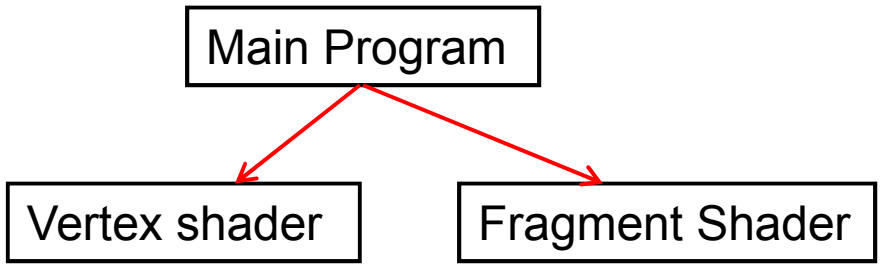


- Vertex array objects and buffer objects can be set in **init()**
- Also set clear color and other OpenGL parameters
- Also set up shaders as part of initialization
  - Read
  - Compile
  - Link

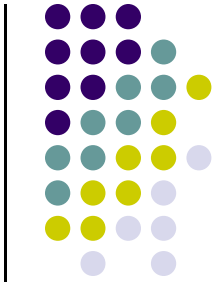
# Vertex Shader



- Remember: every OpenGL program must now write shaders that our OpenGL program will read in
- Need two shaders:
  - Vertex shader: program that is run **per vertex**
  - Fragment shader: program that is run **per pixel**
- OpenGL programs now have 3 parts:
  - Main program, vertex shader, fragment shader



# Vertex Shader



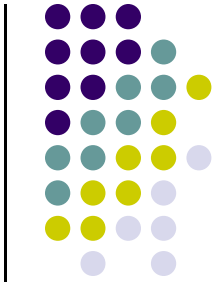
- We write a simple “pass-through” shader (does nothing)
- Save to file on disk called **vsource.glsl**

```
in vec4 vPosition

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

input vertex position

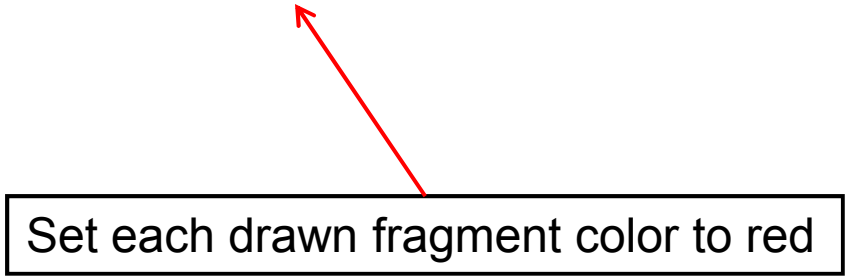
output vertex position

# Fragment Shader

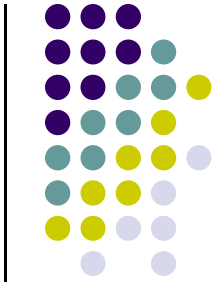


- We write a simple fragment shader (sets color to red)
- Save to file on disk called **fsource.glsl**

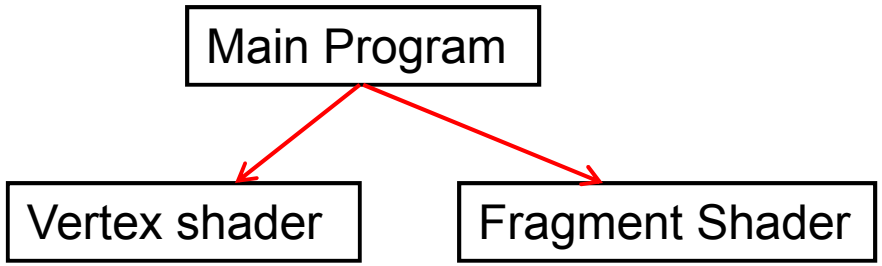
```
void main( )
{
      gl_FragColor = vec(1.0, 0.0, 0.0, 1.0);
}
```



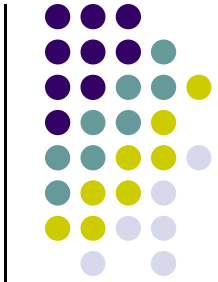
# Putting it all together



- Vertex shader and Fragment shader in same directory as main program
- Main program reads in vertex shader and fragment shader (as strings) and uses them for rendering



# Putting it all Together

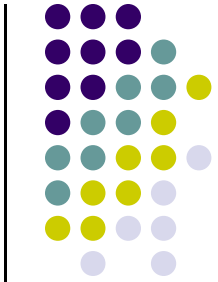


- First, we create container called **program object**

```
Gluint = program;

program = InitShader("vsource.glsl", "fsource.glsl");
glUseProgram(program);
```

- Shader sources are read in, compiled and linked
- During linking, names of all shader variables are bound to indices in tables
- We can retrieve internal index assigned to each variable using function **glGetAttribLocation** command

# Putting it all Together



- Example: To retrieve index of vertex shader variable **vPosition**

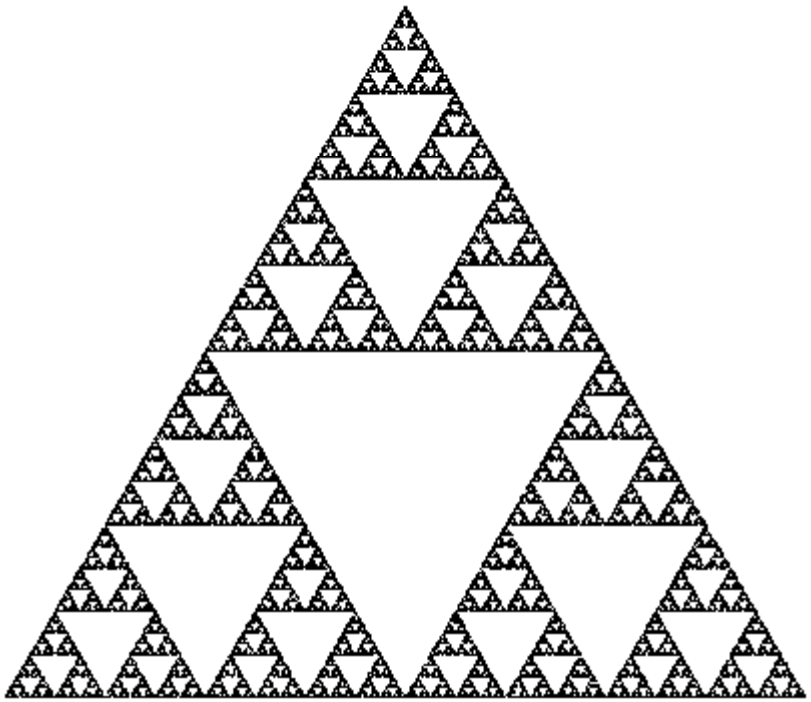
```
Gluint loc;

loc =  glGetAttribLocation(program, "vPosition");
```

- We then have to enable vertex attibutes in the shaders and describe the form of the data in the vertex array

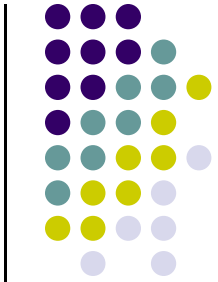
```
glEnableVertexAttribArray(loc);
glVertexAttribPointer(loc, 2, GL_FLOAT, GL_FALSE, 0
                              BUFFER_OFFSET(0));
```

# Sierpinski Gasket Program

- Popular fractal



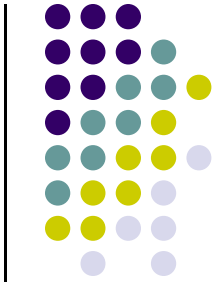
# Sierpinski Gasket



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

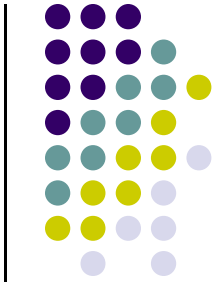
1. Pick initial point **p** = (*x, y, 0*) at random inside a triangle
2. Select on of 3 vertices at random
3. Find **q,** halfway between **p** and randomly selected vertex
4. Draw dot at **q**
5. Replace **p** with **q**
6. Return to step 2

# Actual Sierpinski Code

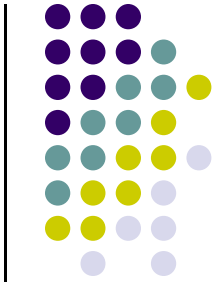


```
#include "vec.h"     // include point types and operations
#include <stdlib.h> // includes random number generator

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



```
// 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;
}
```



# References

- Angel and Shreiner, Chapter 2
- Hill, chapter 2