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

- Generate points & store vertices into an array

```c
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

```c
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)

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

- Make created VBO currently active one

```c
glBindBuffer(GL_ARRAY_BUFFER, buffer); // data is array
```
Recall: 4. Move points GPU memory

3. Move points generated earlier to VBO

```c
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)

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

- Display function using `glDrawArrays`:

- Other possible arguments to `glDrawArrays` instead of `GL_POINTS`?
glDrawArrays() Parameters

\texttt{glDrawArrays(GL\_POINTS, ...)}

- draws dots

\texttt{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

\texttt{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`
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

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

Main Program  initShader( )
Homegrown, connects main Program to shader files
More on this later!!
1. Vertex data Moved to GPU (glBufferData)

2. glDrawArrays

3. Vertex shader invoked on each vertex on GPU

Graphics Hardware (not programmable)
Vertex Shader

- We write a simple “pass-through” shader (does nothing)
- Simply sets output vertex position to received input position
- \texttt{gl\_Position} is built-in variable (already declared)

```cpp
in vec4 vPosition

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

output vertex position

input vertex position
Execution Model

1. Fragments corresponding to Rendered vertices
2. Fragment shader invoked on each fragment on GPU
3. Rendered Fragment Color

OpenGL Program
Fragment Shader

- We write a simple fragment shader (sets color to red)
- \texttt{gl\_FragColor} is built in variable (already declared)

```cpp
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`.

```cpp
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 \((x_1, y_1, 0)\), \((x_2, y_2, 0)\) and \((x_3, y_3, 0)\)

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

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

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

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

$\text{glUniform3f}(x, y, z)$

- **function name**: $\text{glUniform3f}$
- **belongs to GL library**: Yes
- **$x, y, z$ are floats**: Yes
- **$\text{glUniform3fv}(p)$**: Argument is array of values
- **$p$ is a pointer to array**: Yes
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()`

```c
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()

```c
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

<table>
<thead>
<tr>
<th>C++</th>
<th>OpenGL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed char</td>
<td>GLByte</td>
</tr>
<tr>
<td>Short</td>
<td>GLShort</td>
</tr>
<tr>
<td>Int</td>
<td>GLInt</td>
</tr>
<tr>
<td>Float</td>
<td>GLFloat</td>
</tr>
<tr>
<td>Double</td>
<td>GLDouble</td>
</tr>
<tr>
<td>Unsigned char</td>
<td>GLubyte</td>
</tr>
<tr>
<td>Unsigned short</td>
<td>GLushort</td>
</tr>
<tr>
<td>Unsigned int</td>
<td>GLuint</td>
</tr>
</tbody>
</table>

**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
   
   ```c
   glutKeyboardFunc( myKeyboard );
   ```

2. implement keyboard function
   
   ```c
   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

  \[ \text{glutSpecialFunc (specialKeyFcn)}; \]

  ...

  \[ \text{Void specialKeyFcn (Glint specialKey, GLint, xMouse,}
  \]

  \[ \text{Glint yMouse)}] \]

- 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 \texttt{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

```c
def 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*
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