Week 6

- The Mouse Callback
- The Keyboard Callback
- Reshaping the Window
- Menus
- 3D Graphics
- Hidden Surface Removal
- Double Buffering

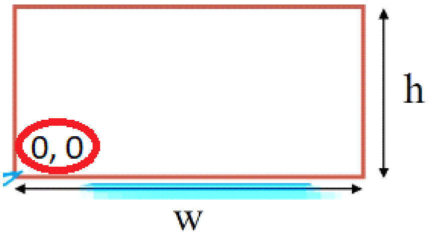
The Mouse Callback

```
glutMouseFunc(mymouse);
void mymouse(GLint button, GLint state, GLint x, GLint y)
```

Parameters passed to the function are:

- *button* - Mouse button (GLUT LEFT BUTTON, GLUT MIDDLE BUTTON, GLUT RIGHT BUTTON)
- *state* - The event (GLUT UP, GLUT DOWN)
- *x,y* - Mouse click position (in pixels) in the window

Positioning

	OpenGL uses a world coordinate system with the **origin at the bottom left corner.** The position on the screen window is usually measured in pixels with the origin at the top-left corner Thus, you must invert the y coordinate passed to your callback function by the height of the window $\boldsymbol{Callback\,Y = h - MouseY}$

Obtain the Window Height

To invert the y position we need to know the window height

- The window height value can change during program execution
- We can use a global variable to keep track of the window height value

Program Termination

A mouse callback where the right mouse button terminates our program.

```
void mouse(int btn, int state, int x, int y)
{
    if (btn == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
        exit(0);
}
```

We have ignored the x and y parameters

Example: *Drawing a small square at the location of the mouse each time the left mouse button is clicked*

We do not use the display callback
Since one is required by GLUT we will use the empty one:

```
mydisplay ( ) { }
```

Code

<pre>void mymouse(int btn, int state, int x, int y)
{
    if (btn == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
        exit(0);
    if (btn == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
        drawSquare(x, y);
}
 	
void drawSquare(int x, int y)
{
    y = h-y; /* invert y position */
    points[i] = vec2(x+size, y+size);
    points[i+1] = vec2(x-size, y+size);
    points[i+2] = vec2(x-size, y-size);
    points[i+3] = vec2(x+size, y-size);
    i += 4;
}</pre>
	Right click = Exit
	
	Left click = Draw square at mouse
	
	Global variables:
	`h, size` and `i`
	
	h stores the height (in pixels) of the window

Motion Callback

We can draw squares continuously as long as a mouse button is depressed by using the ***motion callback***

```
glutMotionFunc(drawSquare);
```

We can also draw squares without depressing a button using the ***passive motion callback***

```
glutPassiveMotionFunc(drawSquare);
```

The Keyboard Callback

```
glutKeyboardFunc(mykey);
void mykey(unsigned char key, int x, int y)
```

Parameters passed to the function are:

- `key` - The ASCII code of the key depressed
- *`x,y`* - Mouse click position (in pixels) in the window, when key was pressed

Example

```
void mykey(unsigned char key, int x,
int y)
{
   if (key == 'Q' || key == 'q') {
      exit (0);
   }
}
```

Special and Modifier Keys

GLUT defines the special keys in `glut.h`

- Function key 1: GLUT_KEY_F1
- Up arrow key: GLUT_KEY_UP

We can check whether modifiers is depressed by `glutGetModifiers()`

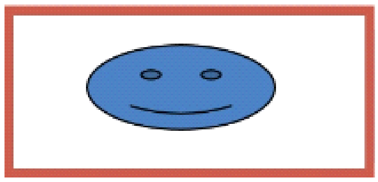
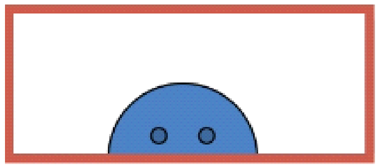
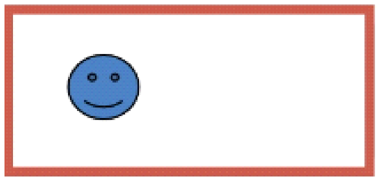
- `GLUT_ACTIVE_SHIFT`
- `GLUT_ACTIVE_CTRL`
- `GLUT_ACTIVE_ALT`

This allows emulation of a three-button mouse with one- or two-button mice

Reshaping the Window

We can reshape/resize the display window by pulling at the corners

The window in the application **can be redrawn in three ways**:

We can display the whole world but force it to fit in the new window (this can alter the aspect ratio). **Default setting**	
We can display part of the world.	
We can scale the whole world to fit in the window while keeping the aspect ratio and then display it	

The Reshape Callback

```
glutReshapeFunc(myreshape);
void myreshape(int w, int h)
```

Parameters: `w,h` - The width and height of new window (pixels)

What happens when the window is resized:

- *A redisplay is posted automatically at end of execution of the callback*
- GLUT has a default reshape callback but you probably want to define your own

The reshape callback is a **good place to put viewing functions** because it is also invoked when the window is first opened

Example 1

<table>
<tr>
<td>

```
void reshape(int w, int h)
{
  glViewport(0, 0, w, h);

  //glOrtho(left,right,bottom,top,near,far)
  glOrtho(-0.2*(float)w/(float)h,
           0.2*(float)w/(float)h,
          -0.2, 0.2, 2.0, 20.0);
}
```

No need to call glutPostRedisplay() here, it is automatically called for reshapes
</td>
<td>
Suppose that the our original window is 500 (width) x 500 (height) pixels and the clipping volume is: left=-0.2, right= 0.2, bottom=-0.2, top=0.2, near=2.0, far=20.0.

glViewport call

- Makes the image still occupy the complete window

glOrtho call

- Redefines clipping variables
- Note that **near and far clipping planes near should be positive.** Otherwise the clipping volume would be taken as behind the camera.
</td>
</tr>
</table>

Example 2

<pre>void reshape(int w, int h) {
  glViewport(0, 0, w, h);
  //glOrtho(left,right,bottom,top,near,far)
  if (w > h)
    glOrtho(-0.2*(float)w/(float)h,
             0.2*(float)w/(float)h,
            -0.2, 0.2, 2.0, 20.0);
  else
    glOrtho(-0.2, 0.2, -0.2*(float)h/(float)w,
            0.2*(float)h/(float)w, 2.0, 20.0);
}</pre>
	This callback preserves the aspect ratio
	
	If width is greater than height, redefine left and right clipping planes
	
	Else, redefine top and bottom clipping planes

Menus

Wednesday, November 20, 2019 3:19 PM

Toolkits and Widgets

Most window systems provide a **platform dependent toolkit** (library of functions) for **building user interfaces** that use <u>special types of windows called widgets</u>

Widget sets include tools such as

- Menus
- Slide bars
- Dials
- Input boxes

GLUT provides a few widgets including menus that are **platform *independent***

GLUT Menus

GLUT supports **pop-up menus**, which may have submenus

Three steps for setting up a menu:

1. Define entries for the menu
2. Define actions to be carried out when each menu item is selected
3. Attach menu to a mouse button

Simple Menu Example

In the init function:

```
// We give our callback function to glutCreateMenu and receive a unique ID
menu id = glutCreateMenu(mymenu);

// We add menu entries. Label first, then what will be returned upon
selection
glutAddMenuEntry("Clear Screen", 1);
glutAddMenuEntry("Exit", 2) ;

// Attach the current menu to the right mouse button
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

Simple menu callback function:

```
void mymenu(int id)
{
    if (id == 1) glClear();
    if (id == 2) exit(0);
}
```

Complex Menu Example

<pre>// submenu for two light sources
int lightMenuId = glutCreateMenu(lightMenu);
glutAddMenuEntry("Move Light 1", 11);
glutAddMenuEntry("Change RGB of Light 1", 12);
glutAddMenuEntry("Move Light 2", 21);
glutAddMenuEntry("Change RGB of Light 2", 22);
 	
// submenu for camera
int cameraMenuId = glutCreateMenu(cameraMenu);
 	
// add these submenus to the main menu
glutCreateMenu(mainMenu);
glutAddSubMenu("Light sources", lightMenuId);
glutAddSubMenu("Camera", cameraMenuId);
glutAttachMenu(GLUT_RIGHT_BUTTON);</pre>
	We <u>create submenus first</u> and **save their IDs**
	
	We then create the main menu,
	and add the submenus using their ID and glutAddSubMenu()

Complex Menu Callbacks

<pre>// callback function for the light menu
void lightMenu(int id) {
    switch (id) {
        case 11: // action for moving light 1
        case 12: // action for changing RGB of light 1
        ...
    }
}
 	
// callback function for the camera menu
void cameraMenu(int id) {
    ...
}</pre>
	Use a switch statement for many possibilities

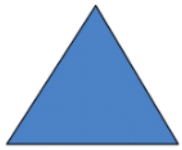
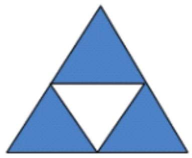
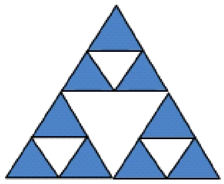
In OpenGL, 2D applications are a special case of 3D graphics

Going from 2D to 3D:

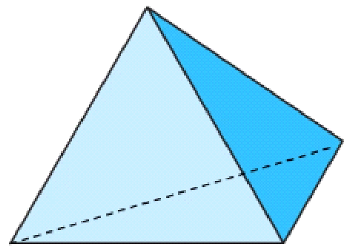
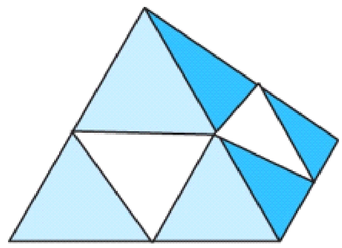
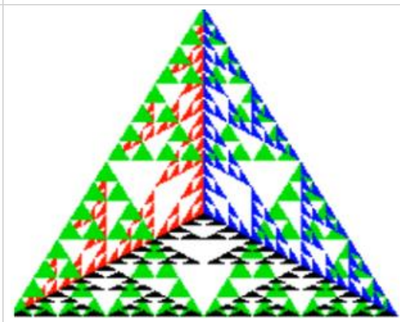
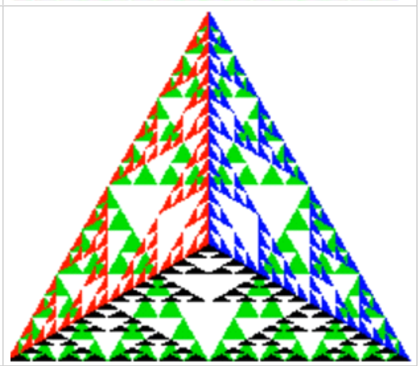
- Not many changes
- Use `vec3 and glUniform3f` (instead of `vec2 and glUniform2f`)
- Need to worry about the order in which primitives are rendered
  OR
- Need to do **hidden-surface removal** (as objects in front can cover objects behind)

Sierpinski Gasket in 2D

The Sierpinski Gasket is a **fractal** that can be produced by a **recursive algorithm**

Start with a triangle	
Connect bisectors of sides and remove central triangle	
Repeat on smaller triangles	
After five subdivisions The area is approaching 0, but the perimeter is approaching infinity	

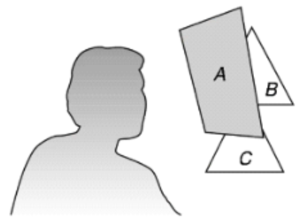
Sierpinski Gasket in 3D

We define a tetrahedron with four triangular faces. We draw each of the four faces using a different color.	
We can subdivide each of the four faces into triangles It appears as if we remove a solid tetrahedron from the center, leaving four smaller tetrahedra	
Because the triangles are drawn in the order they are specified in the program, the front triangles are not always rendered in front of triangles behind them.	
This is more realistic as the faces that are close to us occlude the faces away from us. **Hidden surface removal** is used to do this.	

Hidden Surface Removal

To display 3D graphics, we must only show the visible surfaces (those surfaces that are in front of other surfaces)

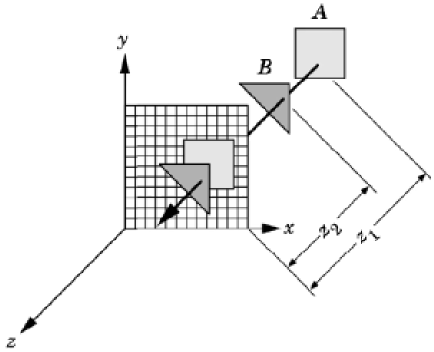
OpenGL uses a **hidden-surface removal method** called the **Z-buffer algorithm**, which saves the depth information of fragments as they are rendered so that *only the front fragments appear in the image.*

	A should be visible but not the parts of B and C that are covered by A

The Z Buffer Algorithm

- Most widely-used hidden-surface-removal algorithm
- Has the advantages of being easy to implement, in either hardware or software
- Compatible with the pipeline architectures, where the algorithm can be executed at the speed at which fragments are passed through the pipeline
- Works in **the image space.**
- It loops over the polygons rather than over pixels of the frame buffer.

Operation



Process of rasterizing one of the two polygons:

We can compute a colour for each point of intersection (**point p** say) between a ray from the camera's center of projection and a pixel

Point p is a point on an object in the object space

We must check whether p is visible. It is visible if it is the closest point of intersection to the camera

- If we are rasterizing polygon B, then its shade will appear at that pixel on the screen as **it has a lower z value** ($z_2 < z_1$)
- If we are rasterizing polygon A, then its shade won't appear at that pixel on the screen

Pseudocode

for each pixel (i,j) **do** *Z-buffer*$[i,j] \leftarrow$ *FAR* *Framebuffer[i,j]* $\leftarrow$ *<background color>* **end for** **for** each polygon A **do** **for** each pixel (i,j) occupied by A **do** Compute depth z and shade s of A at (i,j) **if** $z >$ *Z-buffer*$[i,j]$ **then** Z-buffer$[i,j] \leftarrow z$ *Framebuffer[i,j]* $\leftarrow s$ **end if** **end for** **end for**	We initialize the same sized: • Z buffer with the furthest values • Frame buffer with the background colour For every polygon: • **For each pixel** ○ We compute its depth (z value) and shade ○ If its z value is greater than the Z buffer ▪ We save depth and shade The shades left will be the ones closer to the camera

Using the Z Buffer in OpenGL

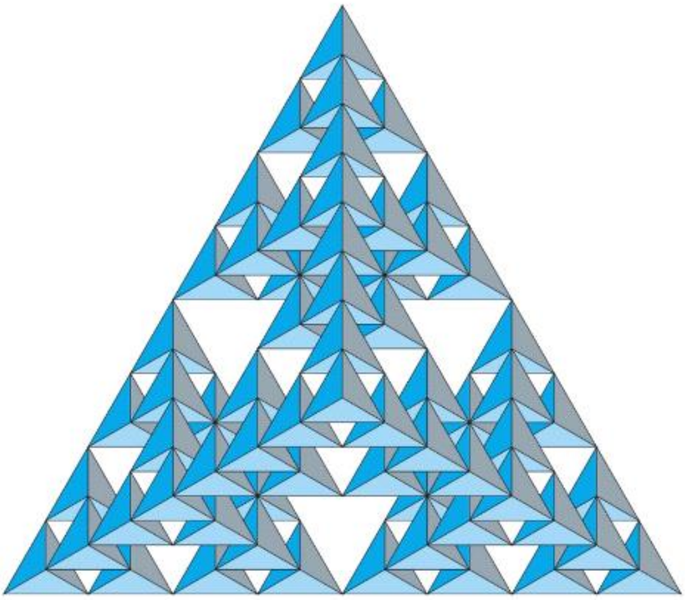
The algorithm uses an extra buffer, the z-buffer, to store depth information as geometry travels down the pipeline

In OpenGL,

- HSR is called "depth testing"
- **The z-buffer must be:**
-

Requested in `main()`	`glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH)`
Enabled in `init()`	`glEnable(GL_DEPTH_TEST)`
Cleared in the display callback	`glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)`

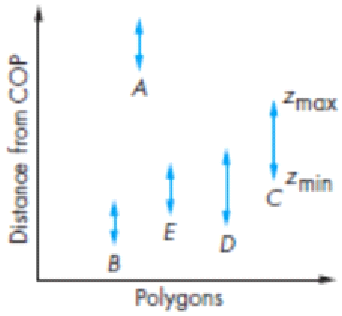
Sierpinski Gasket with HSR



The Painter's Algorithm

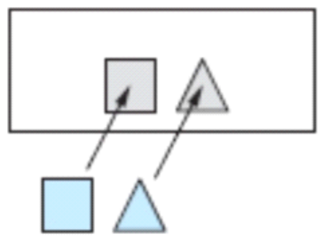
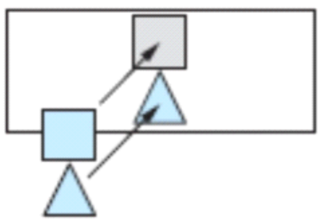
- Although **image-space methods** are dominant in hardware due to the efficiency and ease of implementation of the z-buffer algorithm, often **object-space methods** are used in combination.
- The painter's algorithm is an object-space approach to **hidden surface removal.** It is one of the simplest solutions to the visibility problem in 3D computer graphics
- The name refers to the technique employed by many painters of painting distant parts of a scene before parts which are closer, thereby covering some areas of the distant parts.
- The algorithm sorts all the polygons in a scene by their depths.
- Polygons are painted from the furthest to the closest depth. The **closer polygons are painted over** the further ones.
- Because of how the algorithm works, it is also known as a **depth- sort algorithm.**

Operation

<table>
<tr>
<td>


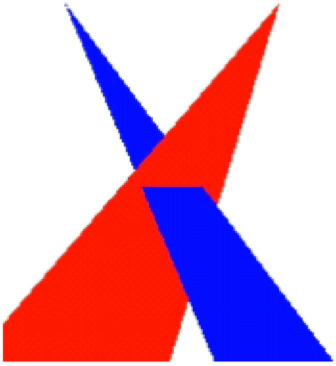
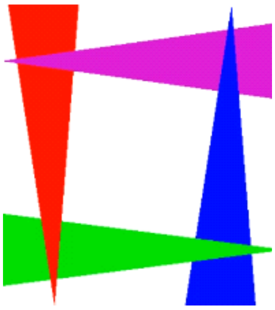
(COP = center of projection)
</td>
<td>
Suppose that we have the z-extent of 5 polygons:

Polygon A can be painted first
However, we can't determine the order for painting the other polygons
The algorithm needs to run a number of increasingly more difficult tests in order to find the painting ordering.

</td>
</tr>
<tr>
<td>

Test for overlap in the x-extents

Test for overlap in the y-extents
</td>
<td>
The simplest test is to check the x- and y- extents of the polygons.
If either of the x- or the y-extents do not overlap, neither polygon can obscure the other. So they can be painted in any order.
If the above test fails, can still determine the order of painting by testing if one polygon lies completely on one side of the other.
</td>
</tr>
</table>

Limitations

The Painter's Algorithm doesn't work in certain cases

	Triangles that pierce/intersect each other Drawing either first does not result in correct image
	Triangles that form a cycle of depth overlap If you draw one triangle first, you cannot produce the correct image

Double Buffering is used for **smooth animation**

- Using a uniform variable opens the door to animation:
  - We can call `glUniform` in the display callback
  - We can then force a redraw through `glutPostRedisplay()`
- To animate a scene smoothly, we need to prevent a partially redrawn frame buffer from being displayed.
- A way to prevent the above issue from happening is to use double buffering — we request two buffers
  - While drawing is performed on the back buffer, the front buffer is being displayed
  - After we are finished drawing, we swap the two buffers

Enabling in OpenGL

1. Request a double buffer:

```
glutInitDisplayMode(GLUT_DOUBLE | … );
```

2. Swapping buffers in display callback:

```
void mydisplay ( ) {
    glClear();
    glDrawArrays(); // Drawing on back buffer
    glutSwapBuffers();
}
```

3. The Idle Callback to makes the scene be redrawn more than once:

Registering:

```
glutIdleFunc(myIdle);
```

Actual:

```
// The Idle Callback is called when no other actions are pending
void myIdle( ) {
    // Explicitly calls the display
    // Recomputes display
    glutPostRedisplay( );
}
```

Element Buffers

Complex 3D models have thousands of triangles

We can *re-use the vertices while defining triangles for efficiency*

`GL_ELEMENT_ARRAY_BUFFER` is used for this

Example:

- Lab 5, q4aIndex.cpp draws a cube by specifying only 8 vertices
- 8 vertices -> 6 squares -> 12 triangles
- A vertex can be part of multiple triangles

3D Model File Format

3D model file formats follow a similar convention:

- A list of vertices as floats (x, y, z)
- A list of elements as integers specifying which vertices connect to form a triangle

Sometimes the vertex normals are also provided as floats