17. Hidden Surface Removal

Drawing the objects that are closer to the viewing position and eliminating objects which are obscured by other “nearer” objects

Two General Categories of Algorithms:

- Object Space
  
  Compares objects and parts of object to each other to determine which surfaces and lines should be labeled as invisible

  Generally used for *hidden line removal*

- Image Space

  Visibility is determined point by point at each pixel position on the projection plane

  Generally used for *hidden surface removal*

*Back face culling* is also a form of hidden surface removal
• **Painter’s Algorithm**

The painter’s algorithm is the simplest hidden surface removal algorithm

- Start by drawing the objects which are furthest from the viewpoint
- Continue drawing objects working from far to near
- Draw the closest objects last

The nearer objects will obscure the farther objects

♦ Problems:
- Objects must be drawn in a particular order based upon their distance from the viewpoint
- If the viewing position is changed, the drawing order must be changed

• **Z Buffering**

Commonly used image-space algorithm which uses a *depth* or *Z buffer* to keep track of the distance from the projection plane to each point on the object

- For each pixel position, the surface with the smallest z coordinate is visible
- Depth or Z values are usually normalized to values between zero and one
Z Buffer Algorithm

1. Clear the color buffer to the background color
2. Initialize all xy coordinates in the Z buffer to one
3. For each fragment of each surface, compare depth values to those already stored in the Z buffer
   - Calculate the distance from the projection plane for each xy position on the surface
   - If the distance is less than the value currently stored in the Z buffer:
     Set the corresponding position in the color buffer to the color of the fragment
     Set the value in the Z buffer to the distance to that object
   - Otherwise:
     Leave the color and Z buffers unchanged

♦ Comments
- Z-buffer testing can increase application performance
- Software buffers are much slower than specialized hardware depth buffers
- The number of bitplanes associated with the Z buffer determine its precision or resolution
• Z Buffering under OpenGL

OpenGL uses Z buffering for hidden surface removal

♦ The depth of the Z buffer in a win32 application is specified as part of the PIXELFORMATDESCRIPTOR used to set up the rendering context

Example 17.1

Setting the Z buffer depth

```c
static PIXELFORMATDESCRIPTOR pfd = {
    sizeof( PIXELFORMATDESCRIPTOR ), // size of pfd
    1,                               // version number
    PFD_DRAW_TO_WINDOW |             // support window
    PFD_SUPPORT_OPENGL |            // support OpenGL
    PFD_DOUBLEBUFFER,               // double buffered
    PFD_TYPE_RGBA,                  // RGBA type
    24,                             // 24-bit color
    0, 0, 0, 0, 0, 0,               // color bits ignored
    0,                              // no alpha buffer
    0,                              // shift bit ignored
    0,                              // no accumulation buffer
    0, 0, 0, 0,                     // accum bits ignored
    // 32, // 32-bit z-buffer
    16,                             // better performance with 16-bit z-buffer
    0,                              // no stencil buffer
    0,                              // no auxiliary buffer
    PFD_MAIN_PLANE,                 // main layer
    0,                              // reserved
    0, 0, 0                         // layer masks ignored
};
```
Setting the comparison function for the depth test

The depth test a fragment must pass in order to alter the frame buffer can be set using

```c
void glDepthFunc (GLenum func);
```

where `func` may be

- GL_NEVER, GL_ALWAYS, GL_LESS, GL_LEQUAL, GL_EQUAL, GL_GEQUAL, GL_GREATER, or GL_NOTEQUAL
- default is GL_LESS

To turn on depth buffering

```c
glEnable( GL_DEPTH_TEST );
```

To turn off depth buffering

```c
glDisable( GL_DEPTH_TEST );
```

Prior to drawing the scene each time clear both the color and the depth buffers:

```c
glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | ... );
```

- Use a **single** `glClear` command to clear all buffers
♦ Changing the depth range

z values may be scaled to lie in a range other than 0.0 to 1.0 using

```c
void glDepthRange( GLclampd near, GLclampd far );
```

where

- `near` and `far` values represent adjustments to the minimum and maximum values that can be stored in the depth buffer
- by default, `near` and `far` are 0.0 and 1.0 respectively

♦ Enabling and Disabling writing to the Depth buffer

```c
void glDepthMask( GLboolean flag );
```

where

- `flags` specifies if the depth buffer is enabled for writing
- `GL_TRUE` enables the writing
- `GL_FALSE` disables writing
- By default, writing is enabled
Decals allow composition of two coplanar figures A and B where B should always appear on top of A.

Depth buffers have a finite precision:

- Setting the depth-buffering function to “replace on greater” and rendering A following B will not have the desired effect.

Decal Algorithm

1. Disable the depth buffer for writing, render A
2. Render B
3. Enable the depth buffer for writing
4. Disable the color buffer for writing, render A
5. Enable the color buffer for writing
Example 17.2

Drawing a decal

// Turn on z-buffering
glEnable( GL_DEPTH_TEST );

// Disable writing to the z-buffer
// Leave z-buffering on.
glDepthMask( GL_FALSE );

// Draw the underlying polygon.
drawSquare( );

// Draw the decal in its appropriate color.
drawTriangle( );

// Re-enable the z-buffer for writing.
glDepthMask( GL_TRUE );

// Disable the RGBA planes for writing. RGBA order.
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);

// Draw the underlying polygon again.
drawSquare( );

// Re-enable RGBA planes for writing. RGBA order.
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);