Week 2

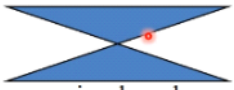
- Triangulation
- Colour
- Pipeline Architecture
- Graphics Modes
- More Complex OpenGL Programs
- The Initialization Function
- Camera Placement and Viewports

Triangulation

Wednesday, November 20, 2019 3:19 PM

Advantages of Triangles

OpenGL will **only display triangles**, as they have three advantages:

1. **Simple**
   - Edges cannot cross
   - 
     nonsimple polygon
2. **Convex**
   - All points on a line segment between two points in a polygon are also in the polygon
   - 
     nonconvex polygon
3. **Flat**
   - All vertices are in the same plane

Tessellation/Triangulation

If polygons are used, then the application program must ***tessellate each polygon into triangles***
It breaks down a given polygon into triangles

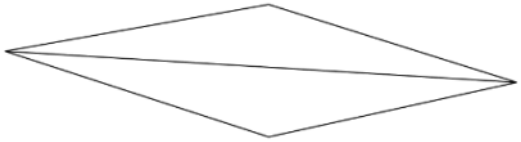
OpenGL 4.1 contains a tessellator to help you do the job
There are preexisting functions that can perform this

Polygon Testing

- **Polygon testing** refers to *testing a polygon for its simplicity and convexity*
- Conceptually it is a simple procedure, however, it is time consuming
- Earlier versions of OpenGL assumed that both properties existed and left the polygon testing to the application
- The present version of OpenGL **only renders triangles**
- We need an algorithm to triangulate an arbitrary polygon

Triangle Quality

Long Thin Triangles = Bad Triangles = Render badly



Equilateral Triangles = Good Triangles = Render well

- To get good triangles, *maximize the minimum interior angle*
- Delaunay triangulation (very expensive) can be used for unstructured points

Recursive Triangulation of a Convex Polygon

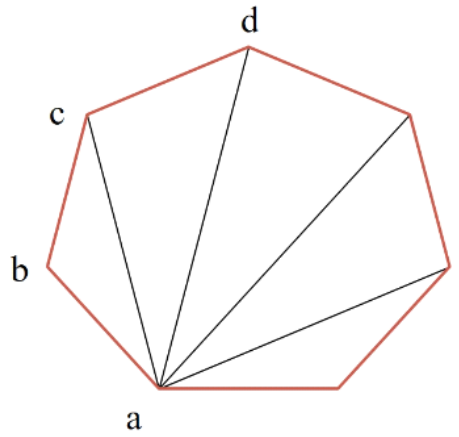
If the polygon is convex, then we can recursively triangulate it to form triangles

Algorithm:

1. Start with abc to form the 1st triangle, then
2. Remove b (the resultant polygon has one fewer vertex
3. (Recursion) Go to Step 1 to form the 2nd triangle

Start with abc, then acd.

Does not guarantee all triangles are good - better algorithms are used



Colour

Wednesday, November 20, 2019 3:19 PM

RGB Colour

Each color component is stored separately in the frame buffer

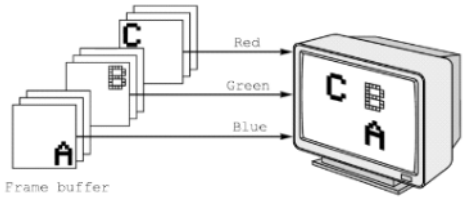
Usually occupies 8 bits per component in the buffer

Colour values in each component can be in one of two ranges:

- From 0.0 (none) to 1.0 (all) using floats
- Over the range 0 to 255 using unsigned bytes

Colors are stored in vec3 or vec4:

```
vec3 red = vec3(1.0, 0.0, 0.0); // R, G, B
vec4 cyan = vec4(0.0, 1.0, 1.0, 1.0); // 4th = opacity
```



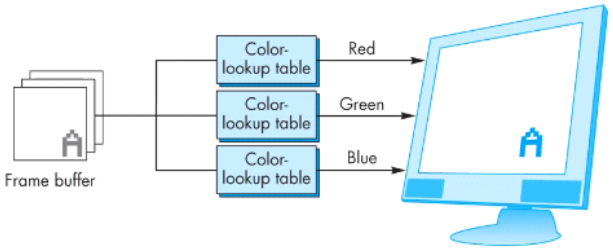
Indexed Colour

In **Indexed Colours**,
colors are indices into tables of **RGB values**
Requires less memory

- Indices are usually 8 bits

But it is not really used now:

- Memory for buffers is inexpensive
- Need more colors for shading



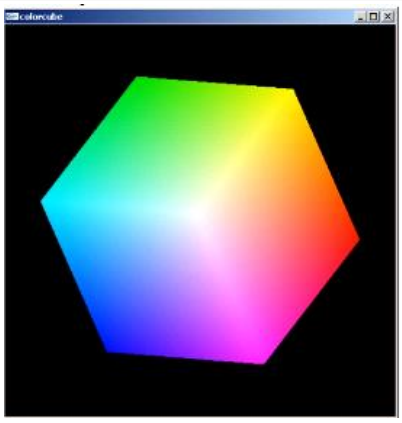
Smooth Colour

We can tell the **rasterizer** in the pipeline how to interpolate the vertex colors across the primitives

Default is **smooth shading** - *OpenGL interpolates vertex colors across visible polygons*.

Alternative is **flat shading** - *Color of the first vertex determines the fill color*.

Shading is handled in the **fragment shader**

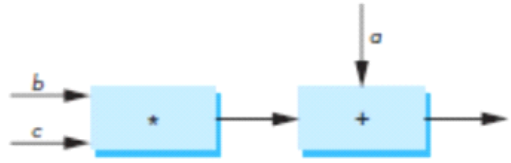


Each vertex has a color here, and it has been interpolated between the vertices

Pipeline Architecture

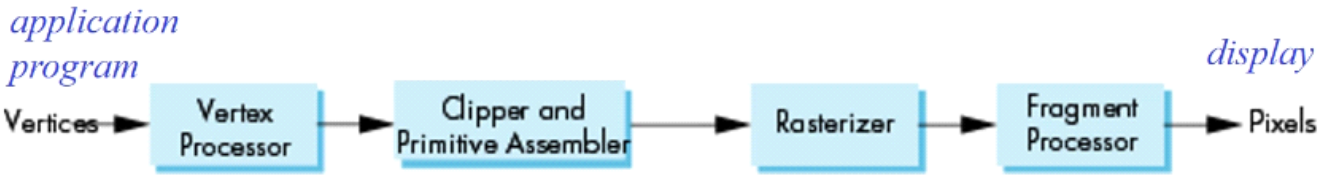
Pipeline Architectures are very common and can be found in many application domains.

Example: *An arithmetic pipeline*

- 

- When two sets of a, b, and c values are passed to the system, the multiplier can carry out the 2nd multiplication without waiting for the adder to finish the calculation time is shortened!
- **When values move to the next part of the pipe, the previous section is freed up for use**

The OpenGL Graphics Pipeline



- Objects passed to the pipeline are processed one at a time in the order they are generated by the application program
  - This means only local lighting effects can be generated - we cannot render shadows with this pipeline
- All steps can be implemented in hardware on the graphics card

Vertex Processor

The Vertex Processor **converts object representations** *from one coordinate system to another*

- Object coordinates
- Camera (eye) coordinates
- Screen coordinates

Every change of vertex coordinates is the result of a matrix transformation being applied to the vertices

Vertex processor also computes vertex colors

Projection

Projection is the process that combines the 3D viewer with the 3D objects to *produce the 2D image,* that occurs during vertex processing (we project the 3D image onto a 2D plane)

- **Perspective projections**
  - All projected rays meet at the center of projection
  - The rays have a common meeting point
- **Parallel/Orthographic projection**
  - Projected rays are parallel - they never meet
  - The center of projection is at infinity.
  - In OpenGL, we can specify the direction of projection instead of the center of projection

Primitive Assembler

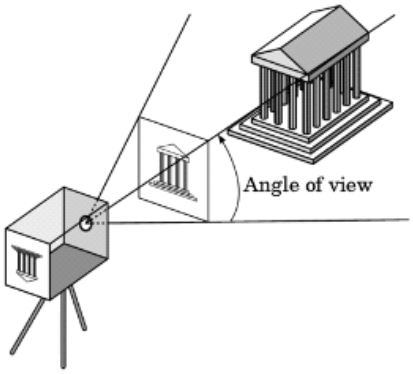
The **Primitive Assembler** collects vertices into geometric objects before clipping and rasterization can take place.

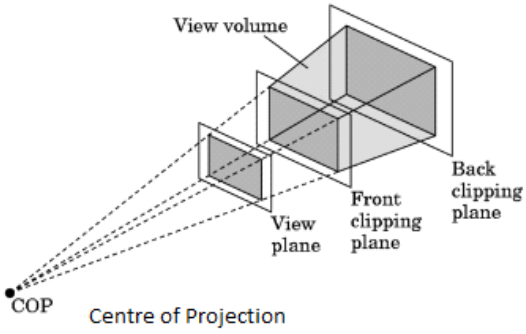
Objects are often:

- Line segments
- Polygons
- Curves and surfaces

Clipping is the *removing of objects in a scene* outside the view volume.

Just as a real camera cannot "see" the whole world, **the virtual camera can only see part of the world** or object space.





Rasterization

The **rasterizer** *produces a set of fragments* (potential pixels) for each object which:

- Have a location in the frame buffer
- Have colour, depth, and alpha attributes
- Alpha is the last attribute in RGBA color which is basically transparency/opacity

The **rasterizer** also interpolates vertex attributes (colour, transparency) over the objects, filling the frame buffer

Fragment Processing is determining the color of the corresponding pixel in the frame buffer

The color of a fragment can be determined by:

- texture mapping (pre assigning of color values)
- interpolation of vertex colors

Fragments may be blocked by other fragments closer to the camera

- **Hidden-surface removal**

Graphics Modes

The two **rendering strategy** modes in Computer Graphics:

- Immediate Mode Graphics
- Retained Mode Graphics

In both, objects are specified by vertices

- Locations in space (2 or 3 dimensional)
- Geometric entities, e.g. points, lines, circles, polygons, curves, surfaces

Immediate Mode Graphics

In **Immediate Mode Graphics,** each time a vertex is specified in application, its location is sent ***immediately*** to the GPU

- Old style programming that uses the glVertex function
- Was removed in OpenGl 3.1
- **Advantages:**
  - Memory efficient as no memory is required to store geometric data
- **Disadvantages:**
  - As the vertices are not stored, if they need to be displayed again, the entire vertex creation and the display process must be repeated.
  - Creates bottleneck between CPU and GPU

Retained Mode Graphics

In **Retained Mode Graphics**, all vertex and attribute data is in an array, which we *send over to GPU each time/frame* we need another render of the scene

- Even better if we store the array on the GPU for multiple renderings - but now we need GPU memory
- Overcomes the vertex recreation problem
- Used in OpenGL 3.1 onward

Comparison using Sierpinski Pseudocode

Immediate Mode Graphics	Retained Mode Graphics

Every time we generate a point, we display it (send to the GPU)
`main()`
`{`
`  initialize_the_system();`
`  p = find_initial_point();`
`  for (some_no_of_points) {`
`  q = generate_a_point(p);`
`  display(q);`
`  p = q;`
`  }`
`  cleanup();`
`}`
	Each time we generate a point, we store it, and at the end we display all of them. This is *more efficient as we send data to the GPU once*
	`main()`
	`{`
	`  initialize_the_system();`
	`  p = find_initial_point();`
	`  for (some_no_of_points) {`
	`  q = generate_a_point(p);`
	`  store_the_point(q);`
	`  p = q;`
	`  }`
	`  display_all_points();`
	`  cleanup();`
	`}`

More Complex OpenGL Programs

Wednesday, 26 February 2020 4:07 PM

Defining Objects in OpenGL Programs

In Retained Graphics Mode in OpenGL programs, we define any object by:

1. Putting all its vertices geometric data in an array
   -
```
vec3 points[3];
points[0] = vec3(0.0, 0.0, 0.0);
points[1] = vec3(0.0, 1.0, 0.0);
points[2] = vec3(0.0, 0.0, 1.0);
```
2. We send the array to the GPU
3. We tell the GPU to render the points as a particular shape

The Main Function

```
// Includes gl.h, glext.h, freeglut.h, vec.h, mat.h
#include "Angel.h"

int main (int argc, char** argv) {

    // Initialize GLUT
    glutInit (&argc , argv);

    // Request "double buffering" and a "depth buffer"
    // - Double buffering helps to create smoother graphics
    // - The depth buffer helps in removing hidden surfaces
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_DEPTH);

    // Specify window size and position
    glutInitWindowSize (500, 500);
    glutInitWindowPosition (0, 0);

    // Require OpenGL 3.2 Core profile
    glutInitContextVersion(3, 2 );
    glutInitContextProfile(GLUT_CORE_PROFILE);

    // Create a window with the title "simple"
    glutCreateWindow("simple");

    // Set mydisplay (our function) as the display callback function
    // (called when the window needs redrawing)
    glutDisplayFunc(mydisplay);

    // Initialize GLEW
    glewInit();

    // Run our Init function = Setup OpenGL state and initialize shaders
    init();

    // Enter event loop
    glutMainLoop();

    // Never reached
    return 0;
}
```

GLUT Functions

Function Name	Info
glutInit	Initializes the GLUT system and allows it to receive command line arguments Should always be included
glutInitDisplayMode	Requests properties for the window (the rendering context) • RGBA color (default) or indexed colour (rare now) • **Double buffering** (usually, smoother) or Single buffering (redraw flickers) • **Depth buffer** (usually in 3D) stores pixel depths with respect to camera to find visible parts ○ Often used with glEnable(GL_DEPTH_TEST) ; • Others: GLUT ALPHA, ... generally for special additional window buffers • Multiple properties are bitwise OR'd with | (vertical bar)
glutWindowSize	Defines the window size in pixels
glutWindowPosition	Positions the window (relative to top-left corner of display)
glutCreateWindow	Creates window with a certain title Certain functions must be called before this Other functions must be called after this
glutDisplayFunc	Sets the display callback function
glutKeyboardFunc	Sets the keyboard callback function (Runs when any key pressed)
glutReshapeFunc	Sets the reshape callback function (Called when window resized)
glutTimerFunc	Sets the timer callback function
glutIdleFunc	Sets the idle callback function (Called when system idle)
glutMainLoop	Enters an infinite event loop (nothing returned, but may exit)

Once we get data to GPU, we can initiate the rendering with a simple display

Prior to this, the vertex buffer objects should contain the vertex data.

A display call back function:

```
void mydisplay() {
    // Clear the buffer
    glClear(GL_COLOR_BUFFER_BIT);
    // Draw arrays as GL_TRIANGLES (first index, 3 vertices)
    glDrawArrays(GL_TRIANGLES, 0, 3);
    // Single buffering: glFlush();
    // Double buffering:
    // - While OpenGL is filling the back buffer, the front buffer is being displayed
    // - When the back one is full, we swap it with the front one
    // - This removes flickering
    glutSwapBuffers();
}
```

The Initialization Function

Initialization tasks include:

1. Clearing window's background and other OpenGL parameters
2. Setting up the vertex array objects and vertex buffer objects
3. Setting up vertex and fragment shaders

Vertex Array Objects (VAOs) bundle **attributes of vertices**

- Position
- Color
- Texture Coordinates

To define an empty VAO:

Linux/Windows	Mac

`// Make object ID holder`
`GLuint vao;`
`// Define type as vertex array`
`glGenVertexArrays(1, &vao);`
`// Bind the VAO ID`
`glBindVertexArray(vao);`
	`GLuint abuffer;`
	`glGenVertexArraysAPPLE(1, &abuffer);`
	`glBindVertexArrayAPPLE(abuffer);`

glBindVertexArray

- Makes the inputted VAO the active object, affecting the Vertex Buffer Object (VBO)
  - A VBO may hold more than one VAO
- Causes a switch of VBOs

Vertex Buffer Objects

Vertex Buffer Objects (VBOs) allow us to transfer

- Large amounts of data to the GPU
- Data in the current vertex array to the GPU

We need to create and bind the VBO like the VAO:

```
Gluint buffer;
glGenBuffers(I, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
```

We need to **copy the vertices to the buffer:**

```
glBufferData(GL_ARRAY_BUFFER, sizeof(points), points);
```

If we have a points array and a colours array, then we need to specify a larger buffer for the VBO and we need to call glBufferSubData for each array

The buffer size is the total size of both

```
glBufferData(GL_ARRAY_BUFFER, sizeof(points) + sizeof(colours), NULL,
GL_STATIC_DRAW);
```

We load the data separately, and adjust the offset accordingly

```
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(points), points);
glBufferSubData(GL_ARRAY BUFFER, sizeof(points), sizeof(colours), colours);
```

Multiple VBOs

Points = Array of vec2/vec3/vec4 = Passed to GPU using buffer[0]
Colours = Array of vec3/vec4 = Passed to GPU using buffer[1]

```
// Create IDs and turn into buffers
GLuint buffer [2];
glGenBuffers(2, buffer); // No address-of needed as it is already a pointer

// Bind 1st buffer and send data to GPU
glBindBuffer(GL_ARRAY_BUFFER, buffer[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(points), points) ;

// Bind 2nd buffer and send data to GPU
glBindBuffer(GL_ARRAY_BUFFER, buffer[1]);
glBufferData(GL_ARRAY_BUFFER, sizeof(colours), colours) ;
```

We **read, compile and link** the shaders to create a program object
The InitShader program (InitShader.cpp) does this for us and crashes if there are any issues.

```
GLuint program = InitShader("vshader. glsl" , "fshader.glsl");
glUseProgram(program);
```

Camera Placement and Viewports

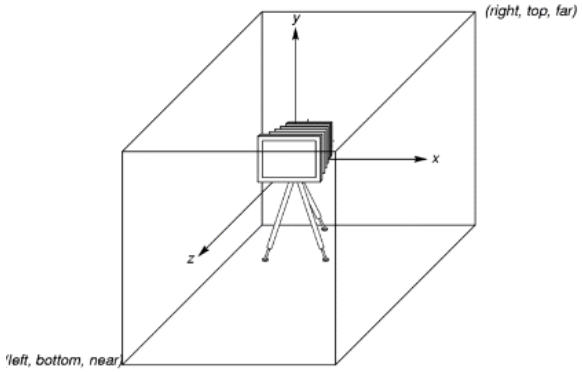
Wednesday, 26 February 2020 5:22 PM

Coordinate Frames

In OpenGL there are 6 coordinate frames specified in the pipeline:

X Coordinates	Info
Object	The units of the points are determined in object/model coordinates
World	The virtual worlds created in OpenGL are in World Coordinates. Each virtual world may contain 100's of objects. The application program applies a sequence of transformations to orient and scale each object before placing them in the virtual world.
Camera	Similar to World Coordinates All graphics systems use coordinate frames that are aligned with the camera coordinates i.e. the camera is at the origin and looking into the negative z-direction. However, this can be altered in the program
Clip	Objects that are not inside the view volume are clipped out. All projection transformations are carried out before clipping. 3D information is still retained here.
Normalized Device	After perspective division gives 3D representation in normalized device coordinates.
Window/Screen	Taking into account the viewport, creates 3D representation in window coordinates. Removes the depth value to produce 2D window coordinates.

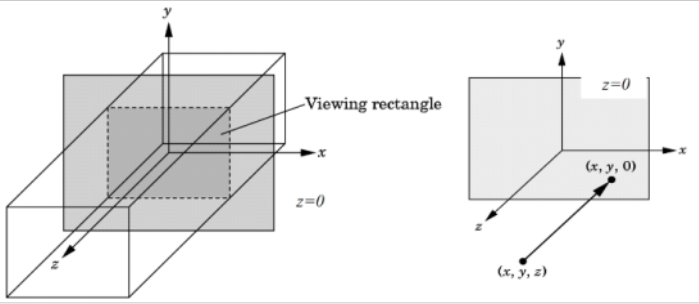
The OpenGL Camera



OpenGL *places a camera* at the **origin in the object coordinate space** pointing in the negative z direction

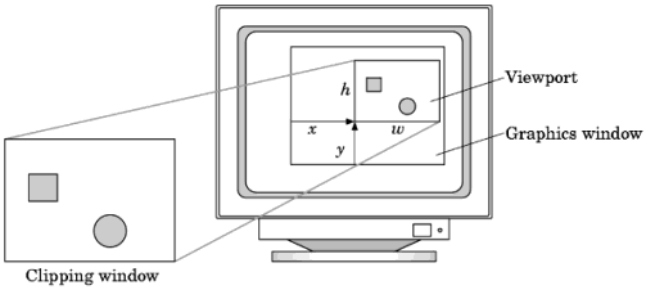
The **default viewing volume** is a box centered at the origin with sides of length 2 (±1 in each direction)

Orthographic Viewing



In the default orthographic view, points are projected forward along the z axis onto the plane z = 0

Viewports



We do not have to use the entire window to render the scene.

We can set a viewport like this:

```
glViewport(x , y , w , h);
```

Values passed to this function should be in pixels (window coordinates)

Transformation is when projection is carried out by a projection matrix

Transformations are also used for **changes in coordinate systems**
(e.g. bringing objects from object coordinates to world coordinates through rotation and translation)

Transformations can be applied in

- Application code (C++)
- Shader (GLSL)