Chapter 3
OpenGL 1.1: Geometry

Dr. Terence van Zyl

University of the Witwatersrand

10th March 2016

Outline

1. Shapes and Colors in OpenGL 1.1
   - OpenGL Primitives
   - OpenGL Colour
   - glColor and glVertex with Arrays
   - The Depth Test

Introduction

- Core features of OpenGL 1.1
- Limited to 2D
- From a C programming language perspective

The Basic Building Blocks Of OpenGL

- OpenGL primitives are
  - points;
  - lines; and
  - triangles
- OpenGL has no Support for
  - curves
- Primitives are defined by there vertices

A Basic Example

OpenGL C code for a triangle

```
glBegin( GL_TRIANGLES );
glVertex2f( -0.7, -0.5 );
glVertex2f( 0.7, -0.5 );
glVertex2f( 0, 0.7 );
glEnd();
```

Definitions (OpenGL functions)

glVertex2f(x,y) the vertex function with 2 dimensions and type float

glBegin(type) start a primitive, this time of type triangle

glEnd() end the primitive

The Other Primitives

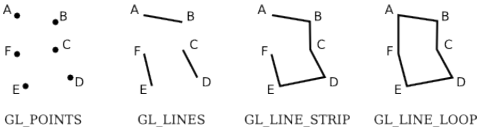


Figure: Visuals of primitives.

Definitions (OpenGL functions)

glPointSize(size) set the size of points in pixels

glEnable(GL_POINT_SMOOTH) round rather than square points

glLineWidth(width) line width in pixels, **doesn't** scale

How The Code Looks

OpenGL C code for a circle

```
glBegin( GL_LINE_LOOP );
for (i = 0; i < 64; i++) {
    angle = 6.2832 * i / 64;  // 6.2832 represents 2*PI
    x = 0.5 * cos(angle);
    y = 0.5 * sin(angle);
    glVertex2f( x, y );
}
glEnd();
```

Notes

Draws 64 line segments to form a circle.

The Other Primitives Cont.

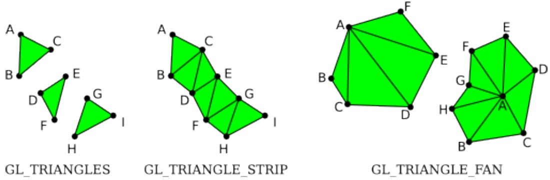


Figure: Triangle primitives.

Definitions (OpenGL constants)

GL_TRIANGLES the first three vertices are the first triangle

The Other Primitives Cont.

Definitions (OpenGL constants)

GL_TRIANGLE_* the first three vertices are the first triangle

GL_TRIANGLE_STRIP each additional vertex adds a triangle using last two vertices

GL_TRIANGLE_FAN each addition vertex adds a triangle between last vertex and first vertex

The Dead Primitives

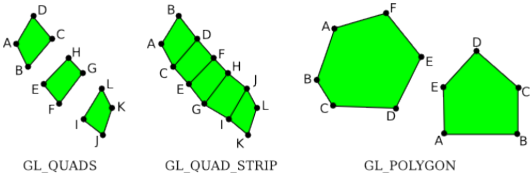


Figure: Quads and Polygons primitives.

Notes

- were buggy, since had to be convex
- don't need them if we have triangles

glColor*

Definitions (OpenGL functions)

glColor3f(r,g,b) red green blue colour as floats in the range 0.0 – 1.0

glColor4f(r,g,b,a) adds alpha

glEnable(GL_BLEND); enables alpha components

glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA) how alpha should be implemented

glColor3ub(r,g,b) as unsigned bytes in range 0 – 255

glColor* In Action

OpenGL C Code for colour

```
glColor3f(0,0,0);     // Draw in black.
glColor3f(1,1,1);     // Draw in white.
glColor3f(1,0,0);     // Draw in full-intensity red.
glColor3ub(1,0,0);    // Draw colour a tiny bit different from black.
glColor3ub(255,0,0); // Draw in full-intensity red.
glColor4f(1, 0, 0, 0.5); // Draw in transparent red.
```

Notes

The drawing in transparent red will only happen if OpenGL has been configured to do transparency.

glColor* and glVertex* working together

OpenGL C code for a coloured triangle

```
glBegin(GL_TRIANGLES);
glColor3f( 1, 0, 0 ); // red
glVertex2f( -0.8, -0.8 );
glColor3f( 0, 1, 0 ); // green
glVertex2f( 0.8, -0.8 );
glColor3f( 0, 0, 1 ); // blue
glVertex2f( 0, 0.9 );
glEnd();
```

Notes

- The colour in OpenGL is stored with the vertex
- The resulting primitive is the interpolation of the colours at its vertices

What A glColor* And glVertex* Look Like

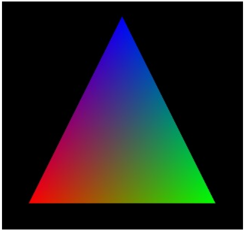


Figure: Triangle with vertices different colours.

glColor* and glVertex* still working together

OpenGL C code for a coloured triangle

```
glColor3ub(255,255,0);   // yellow
glBegin(GL_TRIANGLES);
glVertex2f( -0.5, -0.5 );
glVertex2f( 0.5, -0.5 );
glVertex2f( 0, 0.5 );
glEnd();
```

Notes

- Set the colour once and applies to all vertices

Let's See It In Action

Demo

The Basic OpenGL Colour Triangle

Clearing The Screen

Definitions (OpenGL functions)

glClearColor(r,g,b,a); sets the clear colour

glClear(GL_COLOR_BUFFER_BIT); OpenGl uses a **colour buffer**

glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT); Can also clear the **depth buffer**

Definition

Buffer a region of memory

glColor*v

OpenGL C code for array

```
float coords[] = { -0.5, -0.5,  0.5, -0.5,  0.5, 0.5,  -0.5, 0.5 };

glBegin(GL_TRIANGLE_FAN);
glVertex2fv(coords);        // Uses coords[0] and coords[1].
glVertex2fv(coords + 2);    // Uses coords[2] and coords[3].
glVertex2fv(coords + 4);    // Uses coords[4] and coords[5].
glVertex2fv(coords + 6);    // Uses coords[6] and coords[7].
glEnd();
```

Notes

- Operates over a “vector” or a one dimensional array of values
- Uses pointer arithmetic

The most useful set

OpenGL C code

```
glVertex2f( x, y );              glVertex2fv( xyArray );
glVertex2d( x, y );              glVertex2dv( xyArray );
glVertex2i( x, y );              glVertex2iv( xyArray );
glVertex3f( x, y, z );           glVertex3fv( xyzArray );
glVertex3d( x, y, z );           glVertex3dv( xyzArray );
glVertex3i( x, y, z );           glVertex3iv( xyzArray );

glColor3f( r, g, b );            glColor3f( rgbArray );
glColor3d( r, g, b );            glColor3d( rgbArray );
glColor3ub( r, g, b );           glColor3ub( rgbArray );
glColor4f( r, g, b, a);          glColor4f( rgbaArray );
glColor4d( r, g, b, a);          glColor4d( rgbaArray );
glColor4ub( r, g, b, a);         glColor4ub( rgbaArray );
```

Hidden Surface Problem

The Problem

In 3D things in front should hide the things behind them unless they are transparent.

The Solution

- Painters Algorithm. (Doesn't work in 3D)
  - Just draw from back to front.
  - Front things Hide Back things.

Depth Test

- Depth is the distance from the viewer to the object
- Objects with smaller depth hide objects with larger depth
- Depth value for each pixel stored in **depth buffer**
- When new object drawn **depth buffer** checked
  - If value is in front then update **colour buffer**

Definition (OpenGL function)

glEnable(GL_DEPTH_TEST) turns depth test on, default is off

Let's See It In Action

Demo

Depth Test Demo with Cube

Some Issues With The Depth buffer

Figure: Objects at same depth.

Outline

Let's See It In Action

Demo

3D Axes

Right Handed Coordinate System

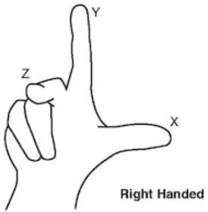


Figure: The right hand rule.

Eye Versus World Coordinates

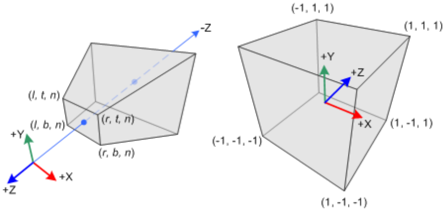
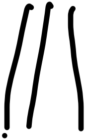


Figure: http://www.songho.ca/opengl/gl_projectionmatrix.html

Where are we now?



Holography

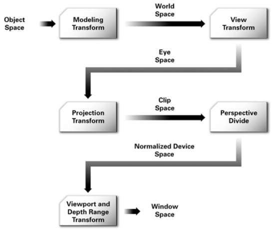


Figure: From object Space to window space.

3D Rotation, Scaling and Translation

Definitions (OpenGL functions: Translate)

glTranslatef(dx,dy,dz) translate from (x, y, z) to $(x + dx, y + dy, z + dz)$

glTranslated(dx,dy,dz) does the same as above but for doubles instead of floats

Definitions (OpenGL functions: Scale)

glScalef(sx,sy,sz) scale (x, y, z) to $(x * sx, y * sy, z * sz)$

glScalef(1,1,-1) does reflection about the z axis

3D Rotation, Scaling and Translation

- Rotation is about a **line** called the **axis of rotation**
- Usually the line is the x, y, z -axis
- But we still don't know which way we go around the axis
- Clockwise or Counter-clockwise?

3D Rotation, Scaling and Translation

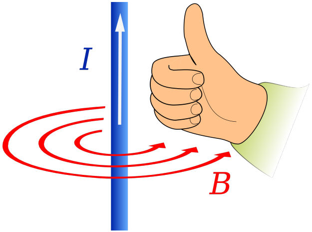


Figure: Rotation from the right hand rule.

Let's See It In Action

Demo

3D Rotation Demo

Some Examples

Examples

glScalef(2,2,2) Uniform scaling by a factor of 2.

glScalef(0.5,1,1) Shrink by half in the x-direction only.

glScalef(-1,1,1) Reflect through the yz-plane. Reflects the positive x-axis onto negative x.

glTranslatef(5,0,0) Move 5 units in the positive x-direction.

glTranslatef(3,5,-7.5) Move each point (x, y, z) to $(x + 3, y + 5, z - 7.5)$.

Some Examples

Examples

glRotatef(90,1,0,0) Rotate 90 degrees about the x-axis. Moves the $+y$ axis onto the $+z$ axis and the $+z$ axis onto the $-y$ axis.

glRotatef(-90,-1,0,0) Has the same effect as the previous rotation.

glRotatef(90,0,1,0) Rotate 90 degrees about the y-axis. Moves the $+z$ axis onto the $+x$ axis and the $+x$ axis onto the $-z$ axis.

glRotatef(90,0,0,1) Rotate 90 degrees about the z-axis. Moves the $+x$ axis onto the $+y$ axis and the $+y$ axis onto the $-x$ axis.

glRotatef(30,1.5,2,-3) Rotate 30 degrees about the line through the points $(0, 0, 0)$ and $(1.5, 2, -3)$.

Order Counts

NB!!!!!

Remember that transforms are applied to objects that are drawn after the transformation function is called, and that transformations apply to objects in the opposite order of the order in which they appear in the code.

Let's See It In Action

Demo

3D Rotation Demo

Hierarchy In 3D Like 2D

Definitions (OpenGL functions)

glLoadIdentity() restore the identity matrix

glPushMatrix() push the current transform to the stack

glPopMatrix() pop transform from the stack and replace current

OpenGL C code for drawing a square

```
void square( float r, float g, float b ) {
    glColor3f(r,g,b);
    glBegin(GL_TRIANGLE_FAN);
    glVertex3f(-0.5, -0.5, 0.5);
    glVertex3f(0.5, -0.5, 0.5);
    glVertex3f(0.5, 0.5, 0.5);
    glVertex3f(-0.5, 0.5, 0.5);
    glEnd();
}
```

Hierarchy In 3D Like 2D

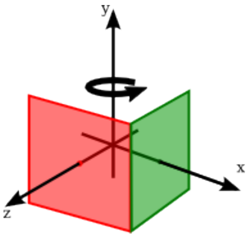


Figure: Using hierarchy and stack to draw a cube.

OpenGL C code for starting a cube

```
glPushMatrix();
glRotatef(90, 0, 1, 0);
square(0, 1, 0);
glPopMatrix();
```

Hierarchy In 3D Like 2D

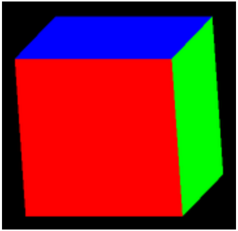


Figure: The completed cube.

OpenGL C code for whole cube

```
void cube(float size) {  // draws a cube with side length = size
    glPushMatrix();       // Save a copy of the current matrix.
    glScalef(size,size,size); // scale unit cube to desired size

    square(1, 0, 0); // red front face
```

Hierarchy In 3D Like 2D cont.

OpenGL C code for whole cube

```
glPushMatrix();
glRotatef(90, 0, 1, 0);
square(0, 1, 0); // green right face
glPopMatrix();

glPushMatrix();
glRotatef(-90, 1, 0, 0);
square(0, 0, 1); // blue top face
glPopMatrix();

glPushMatrix();
glRotatef(180, 0, 1, 0);
square(0, 1, 1); // cyan back face
glPopMatrix();

glPushMatrix();
glRotatef(-90, 0, 1, 0);
square(1, 0, 1); // magenta left face
glPopMatrix();
```

Hierarchy In 3D Like 2D cont.

OpenGL C code for whole cube

```
    glPushMatrix();
    glRotatef(90, 1, 0, 0);
    square(1, 1, 0); // yellow bottom face
    glPopMatrix();

    glPopMatrix(); // Restore matrix to its state before cube() was called.

}
```

Let's See It In Action

Demo

glism/unlit-cube.html

Outline

Many Coordinate Systems

Introduction

When working with OpenGL you need to work in a number of coordinate systems. This part of the lecture will review some of the coordinate systems we have already encountered and will introduce some new coordinate systems.

The Coordinates

The coordinates we have so far are:

Object Coordinates which can be transformed with a **modelling transform** into

World Coordinates which can be transformed with a **viewing transform** into

Eye Coordinates ...

The Modelview Transform

Note

OpenGL doesn't keep track of your modelling transform or your viewing transform. It has one transform the **modelview transform.**

The Coordinates

The coordinates we have so far are:

Object Coordinates which can be transformed with a **modelling transform** into

World Coordinates which can be transformed with a **viewing transform** into

Eye Coordinates which can be transformed with a **projection transform** into

Clip Coordinates ...

The Clip Coordinates and Projection Transform

When working within the constrains of computer generated world

- we are clipped by our **viewport** in the (x, y)-directions;
- we are clipped by the **depth test** in the z-direction;
- the (x, y, z)-clipping gives rise to a **view volume**;
- the view volume is scaled to a **cube** with origin $(0, 0, 0)$ and extends from $(-1, 1)$ along each axis; and
- the cube gives rise to the **clip coordinates**.

The Coordinates

The coordinates we have so far are:

Object Coordinates which can be transformed with a **modelling transform** into

World Coordinates which can be transformed with a **viewing transform** into

Eye Coordinates which can be transformed with a **projection transform** into

Clip Coordinates which can be transformed with a **viewport transform** into

Device Coordinates which can be transformed with a **brain transform** into

... ...

The Device Coordinates

A device is the surface to which the scene will be rendered. As such it typically

- is in 2D coordinates;
- has units of size pixel; and
- is the size of the screen or window

The Whole Picture

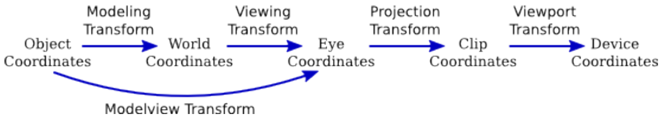


Figure: The Transformation Pipeline[1]

Notes

The Viewport Transformation

Introduction

The viewport transform is a very simple transform which takes the clip coordinates to device coordinates. We just need to specify the size of the device in pixels.

Viewport In OpenGL

Definition (OpenGL functions)

glViewport(x,y,width,height) Everything from the view volume will be shown in the viewport using $0, 0$ at the lower left corner, width and height specify the size of the viewport.

OpenGL C code example of multiple viewports

```
glViewport(0,0,300,400);  // Draw to left half of the drawing surface.
    .
    .   // Draw the first scene.
    .
glViewport(300,0,300,400);  // Draw to right half of the drawing surface.
    .
    .   // Draw the second scene.
    .
```

The Projection Transformation

Introduction

- OpenGL keeps track of the projection transform as a separate matrix
- All transforms can be applied to both the modelview and the projection matrices
- Matrix mode tells OpenGL which matrix it is applying the transform to

Which Transformation

Definitions (OpenGL functions)

glMatrixMode(GL_PROJECTION) perform transforms on the projection matrix

glMartrixMode(GL_MODELVIEW) perform transforms on the modelview matrix

Projections

Definitions

Perspective Projection Physically realistic projection, scale size with distance when projecting z onto the x, y-plane

Orthographic Projection Just discard z and project scene onto the xy-plane

Perspective Projection

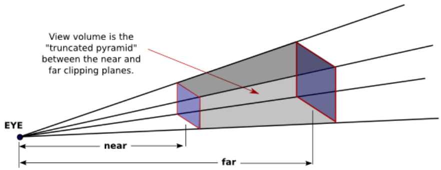


Figure: The view volume for a perspective projection transform[1].

Notes

- Must clip due to depth test between **near** < **far**.
- Anything closer then near or further than far is discarded.

The Frustrum

Definition (OpenGL function)

glFrustum(xmin,xmax,ymin,ymax,near,far) In order to set up a perspective projection transformation in OpenGL we use a thing called a **frustrum**. xmin, xmax, ymin, and ymax specify the horizontal and vertical limits of the view volume at the near clipping plane.

OpenGL C code for setting up a perspective viewing volume

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum( xmin, xmax, ymin, ymax, near, far );
glMatrixMode(GL_MODELVIEW);
```

Orthographic Projection

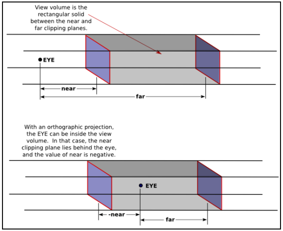


Figure: The view volume for a orthographic projection transform[1].

Notes

- Near < far.

The Orthographic Projection

Definition (OpenGL function)

glOrtho(xmin,xmax,ymin,ymax,near,far); xmin, xmax, ymin, and ymax specify the horizontal and vertical limits of the view volume at the clipping plane.

OpenGL C code for setting up a perspective viewing volume

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho( -10, 10, -10, 10, -10, 10 );
glMatrixMode(GL_MODELVIEW);
```

GLU

Definition (OpenGL function)

gluPerspective(fieldOfViewAngle,aspect,near,far) since glFrustrum is not very intuitive, there is a library call **GLU** which provides this utility function. Here aspect should be set to the aspect of the viewport. The angle says how much of the scene you would see. Typical values 30 to 60 degrees.

The Modelview Transformation

Introduction

OpenGL combines modelling and viewing into a single transform since although they seem distinct they are in fact very difficult to model separately. The reason for this is that a view transform can be achieved by a model transform and vice versa.

Model And View Transforms Are Not The Same

Typically when we draw a scene we do the following:

1. Load the identity matrix, for a well-defined starting point;
2. apply the viewing transformation; and
3. draw the objects in the scene, each with its own modeling transformation.

Lets See It In Action

Demo

Modelling Transform vs. Viewing Transform in 3D

GLU

The GLU library provides the following convenient method for setting up a viewing transformation

Definition (GLU function)

gluLookAt(eyeX,eyeY,eyeZ,refX,refY,refZ,upX,upY,upZ) places the viewer at the point (eyeX,eyeY,eyeZ), looking towards the point (refX,refY,refZ) oriented so that the vector (upX,upY,upZ) points upward.

OpenGL C Code for a look

```
gluLookAt( 10,0,0,  0,0,0,  0,1,0 );
```

Setting It All Up

OpenGL C Code for typical setup

```
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

// possibly set up the projection here, if not done elsewhere

glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
gluLookAt( eyeX,eyeY,eyeZ, refX,refY,refZ, upX,upY,upZ );
glPushMatrix();
  .
  .   // apply modeling transform and draw an object
  .
glPopMatrix();
glPushMatrix();
  .
  .   // apply another modeling transform and draw another object
  .
glPopMatrix();
  .
  .
```

Outline

4. Polygonal Meshes and glDrawArrays
   - Indexed Face Sets
   - glDrawArrays and glDrawElements
   - Display Lists and VBOs

Indexed Face Sets

Introduction

We need to loook at how we can generate more complex surfaces in OpenGL. We do this through the use of a **polygonal mesh**. Polygons in a polygonal mesh are also referred to as "faces". The way we represent the polygonal mesh is through a **index face set (IFS)**.

Index Face Sets

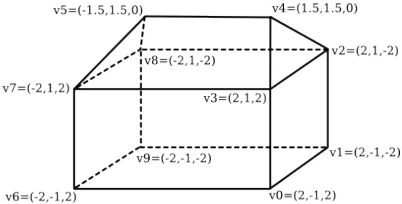


Figure: A "house," as a polyhedron with 10 vertices and 9 faces[1].

```
Vertex #0.  (2, −1, 2)
Vertex #1.  (2, −1, −2)
Vertex #2.  (2, 1, −2)
Vertex #3.  (2, 1, 2)
Vertex #4.  (1.5, 1.5, 0)
Vertex #5.  (−1.5, 1.5, 0)
Vertex #6.  (−2, −1, 2)
Vertex #7.  (−2, 1, 2)
Vertex #8.  (−2, 1, −2)
Vertex #9.  (−2, −1, −2)
```

```
Face #0:  (0, 1, 2, 3)
Face #1:  (3, 2, 4)
Face #2:  (7, 3, 4, 5)
Face #3:  (2, 8, 5, 4)
Face #4:  (5, 8, 7)
Face #5:  (0, 3, 7, 6)
Face #6:  (0, 6, 9, 1)
Face #7:  (2, 1, 9, 8)
Face #8:  (6, 7, 8, 9)
```

Vertex Order

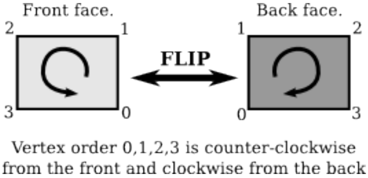


Figure: Vertex Order Counts, Follow the right Hand Rule.

Pseudo code example IFS

```
vertexList = {  {2,-1,2}, {2,-1,-2}, {2,1,-2}, {2,1,2}, {1.5,1.5,0},
       {-1.5,1.5,0}, {-2,-1,2}, {-2,1,2}, {-2,1,-2}, {-2,-1,-2}  };

faceList = {  {0,1,2,3}, {3,2,4}, {7,3,4,5}, {2,8,5,4}, {5,8,7},
       {0,3,7,6}, {0,6,9,1}, {2,1,9,8}, {6,7,8,9}  };
```

Example house in OpenGL

OpenGL C code for a house

```
int vertexCount = 10;  // Number of vertices.
double vertexData[] = { 2,-1,2, 2,-1,-2, 2,1,-2, 2,1,2, 1.5,1.5,0,
                        -1.5,1.5,0, -2,-1,2, -2,1,2, -2,1,-2, -2,-1,-2 };

int faceCount = 9;  // Number of faces.
int[] faceData = { 0,1,2,3,-1, 3,2,4,-1, 7,3,4,5,-1, 2,8,5,4,-1,
                   5,8,7,-1, 0,3,7,6,-1, 0,6,9,1,-1, 2,1,9,8,-1,
                   6,7,8,9,-1 };

int i,j=0; // index into the faceData array
for (i = 0; i < faceCount; i++) {
    glColor3dv( &faceColors[ i*3 ] );  // Color for face number i.
    glBegin(GL_TRIANGLE_FAN);
    while ( faceData[j] != -1) { // Generate vertices for face number i.
        int vertexNum = faceData[j]; // Vertex number in vertexData array.
        glVertex3dv( &vertexData[ vertexNum*3 ] );
        j++;
    }
    j++;  // increment j past the -1 that ended the data for this face.
    glEnd();
}
```

Lines And Faces At The Same Depth

Definition (OpenGL function)

glPolygonOffset(1,1) adjust the depth, in clip coordinates, of a polygon, in order to avoid having two objects exactly at the same depth.

OpenGL C code for line offset

```
glPolygonOffset(1,1);
glEnable( GL_POLYGON_OFFSET_FILL );
    .
    .   // Draw the faces.
    .
glDisable( GL_POLYGON_OFFSET_FILL );
    .
    .   // Draw the edges.
    .
```

Lets See It In Action

Demo

IFS Polyhedron Viewer

glDrawArrays and glDrawElements

Introduction

Up till now we have been focused on drawing objects using individual function calls. This is not very efficient. Enter glDrawArrays and glDrawElements.

glDrawArrays

Definitions (OpenGL functions)

glVertexPointer(size,type,stride,array) where to find the data; type is one of GL_FLOAT, GL_INT, and GL_DOUBLE

glDrawArrays(primitiveType,firstVertex,vertexCount) equivalent to glBegin/glEnd where primitiveType is: on of GL_TRIANGLE_STRIP, etc.

OpenGL C code setting up a vertex array

```c
float coords[8] = { -0.5,-0.5, 0.5,-0.5, 0.5,0.5, -0.5,0.5 };
glEnableClientState( GL_VERTEX_ARRAY );
glVertexPointer( 2, GL_FLOAT, 0, coords );
glDrawArrays( GL_TRIANGLE_FAN, 0, 4 );
```

Colour At Each Vertex Of Array

Definitions (OpenGL functions)

glColorPointer(size,type,stride,array) one dimensional array of colours for each vertex

glEnableClientState(GL_COLOR_ARRAY) on top of glEnableClientState above we must also tell OpenGL we using colour

Putting It All Together

OpenGL C code drawing a colour array

```
// two coords per vertex, three RGB values per vertex.
float coords[6] = { -0.9,-0.9,  0.9,-0.9,  0,0.7 };
float colors[9] = { 1,0,0,  0,1,0,  1,0,0 };

// Set data type and location.
glVertexPointer( 2, GL_FLOAT, 0, coords );
glColorPointer( 3, GL_FLOAT, 0, colors );

// Enable use of arrays.
glEnableClientState( GL_VERTEX_ARRAY );
glEnableClientState( GL_COLOR_ARRAY );

// Use 3 vertices, starting with vertex 0.
glDrawArrays( GL_TRIANGLES, 0, 3 );
```

glDrawElements

Definition (OpenGL function)

glDrawElements(primitiveType,vertexCount,dataType,array) here array is the list vertex coordinates

Note

glDrawElements works like glDrawArrray except it works with an IFS like structure. The main advantage of IFS being reuse of vertices.

glDrawElements

OpenGL C code using glDrawElements

```
// Coordinates for the vertices of a cube.
float vertexCoords[24] =
{ 1,1,1, 1,1,-1, 1,-1,-1, 1,-1,1, -1,1,1, -1,1,-1, -1,-1,-1, -1,-1,1 };

// An RGB color value for each vertex
float vertexColors[24] =
{ 1,1,1, 1,0,0, 1,1,0, 0,1,0, 0,0,1, 1,0,1, 0,0,0, 0,1,1 };

// Vertex numbers for the six faces.
int elementArray[24] =
{ 0,1,2,3, 0,3,7,4, 0,4,5,1, 6,2,1,5, 6,5,4,7, 6,7,3,2 };

glVertexPointer( 3, GL_FLOAT, 0, vertexCoords );
glColorPointer( 3, GL_FLOAT, 0, vertexColors );

glEnableClientState( GL_VERTEX_ARRAY );
glEnableClientState( GL_COLOR_ARRAY );

glDrawElements( GL_QUADS, 24, GL_UNSIGNED_INT, elementArray );
```

Lets See It In Action

Demo

glsim/cubes-with-vertex-arrays.html.

Vertext Buffer Objects (VBOs)

Introduction

We need a more efficient mechanism to keep data at the GPU. Enter VBOs. Basically we send the data to the GPU as an array using VBOs and then we can get glDrawArray and glDrawElements to use the VBOs instead of local arrays.

Outline

5. Some Linear Algebra
    - Vectors and Vector Math
    - Matrices and Transformations
    - Homogeneous Coordinates

Vectors and Vector Math

Make Sure You Know

- Dot product
- Scalar product
- Cross product
- Norm and Normalize

The Unit Normal For Lighting

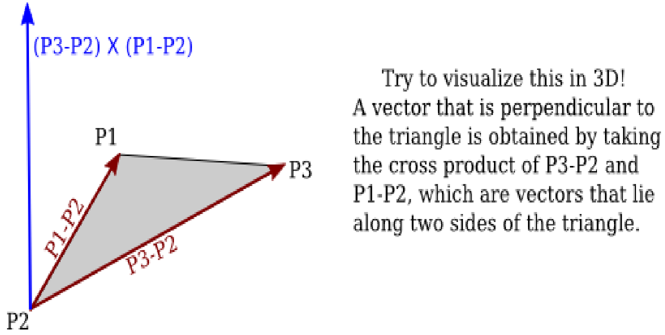


Figure: Cross product of the vertices P_1,$P_2$and P_3.

Matrices and Transformations

Introduction

The mathematics that underpins the fields of graphics is that of linear algebra.

Matrix Multiplication

$$\begin{pmatrix} a1 & a2 & a3 \\ b1 & b2 & b3 \\ c1 & c2 & c3 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a1{*}x + a2{*}y + a3{*}z \\ b1{*}x + b2{*}y + b3{*}z \\ c1{*}x + c2{*}y + c3{*}z \end{pmatrix}$$

Figure: Matrix time vector.

Linear Transformation In 3D

$$\begin{aligned} x_2 &= a_1x_1 + a_2y_1 + a_3z_1 + t_1 \\ y_2 &= b_1x_1 + b_2y_1 + b_3z_1 + t_2 \\ z_2 &= c_1x_1 + c_2y_1 + c_3z_1 + t_3 \end{aligned}$$

$$\begin{pmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{pmatrix}$$

Affine Transform In 3D

$$\begin{pmatrix} a_1 & a_2 & a_3 & t_1 \\ b_1 & b_2 & b_3 & t_2 \\ c_1 & c_2 & c_3 & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Transform Matrices

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Identity Matrix

$$\begin{pmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

glTranslatef(tx,ty,tz)

$$\begin{pmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

glScalef(sx,sy,sz)

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(d) & -\sin(d) & 0 \\ 0 & \sin(d) & \cos(d) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

glRotatef(d,1,0,0)

$$\begin{pmatrix} \cos(d) & 0 & \sin(d) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(d) & 0 & \cos(d) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

glRotatef(d,0,1,0)

$$\begin{pmatrix} \cos(d) & -\sin(d) & 0 & 0 \\ \sin(d) & \cos(d) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

glRotatef(d,0,0,1)

Figure: Transformations.

Homogeneous Coordinates

Introduction

...

Definition

Homogeneous coordinates A way of representing n-dimensional vectors as $(n+1)$-dimensional vectors where two $(n+1)$ vectors represent the same n-dimensional vector if they differ by a scalar multiple. In $3D$, for example, if w is not zero, then the homogeneous coordinates (x, y, z, w) are equivalent to homogeneous coordinates $(x/w, y/w, z/w, 1)$, since they differ by multiplication by the scalar w. Both sets of coordinates represent the $3D$ vector $(x/w, y/w, z/w)$.

Outline

6. Using GLUT
   - Using Glut

Using Glut

Introduction

OpenGL is an API for graphics only, with no support for things like windows or events. In order to add this support we use OpenGL Utility Toolkit (GLUT).

Usage

```
$ gcc -o glutprog glutprog.c -lGL -lglut
```

Definition (GLUT function)

glutDisplayFunc(display) The display function should contain OpenGL drawing code that can completely redraw the scene

GLUT C code basic example

```c
#include <GL/gl.h>
#include <GL/glut.h>
void display() {    .    .   // OpenGL drawing code goes here!    . }
glutDisplayFunc(display);
```

Larger Example

GLUT C code example

```
int main(int argc, char** argv) {
    glutInit(&argc, argv);  // Required initialization!
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_DEPTH);
    glutInitWindowSize(500,500);       // size of display area, in pixels
    glutInitWindowPosition(100,100);   // location in screen coordinates
    glutCreateWindow("Title");         // parameter is window title

    glutDisplayFunc(display);          // called when window needs to be redrawn
    glutReshapeFunc(reshape);          // called when size of the window changes
    glutKeyboardFunc(keyFunc);         // called when user types a character
    glutSpecialFunc(specialKeyFunc);   // called when user presses a special key
    glutMouseFunc(mouseFunc);          // called for mousedown and mouseup events
    glutMotionFunc(mouseDragFunc);     // called when mouse is dragged
    glutIdleFunc(idleFun);             // called when there are no other events

    glutMainLoop(); // Run the event loop!  This function never returns.
    return 0;       // This line will never actually be reached.
}
```

Double Buffering

Definition

Double Buffering A graphics technique in which an image is drawn off-screen, in a region of memory called an off-screen buffer or "back buffer." When the image is drawn, it can be copied to the buffer that represents the contents of the screen, which is also known as the "front buffer." In true double buffering, the image doesn't have to be copied; instead, the buffers can be "swapped" so that the back buffer becomes the front buffer, and the front buffer becomes the back buffer.

Double Buffering

GLUT code for double buffering

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_DEPTH);
            .
            .
glutSwapBuffers();
```

Note

GLUT also supports basic shapes such as spheres and cones. Using for instance glutSolidSphere(...), glutSolidCone(...), glutSolidTorus(...), glutSolidCylinder(...) and glutSolidCube(...)

Solid Tea Pot

glutSolidTeapot(size) draws a tea pot

Figure: Teapot.

David J. Eck; Introduction to Computer Graphics; 2016;
http://math.hws.edu/graphicsbook/