Programming with OpenGL: PART III

Serena Morigi

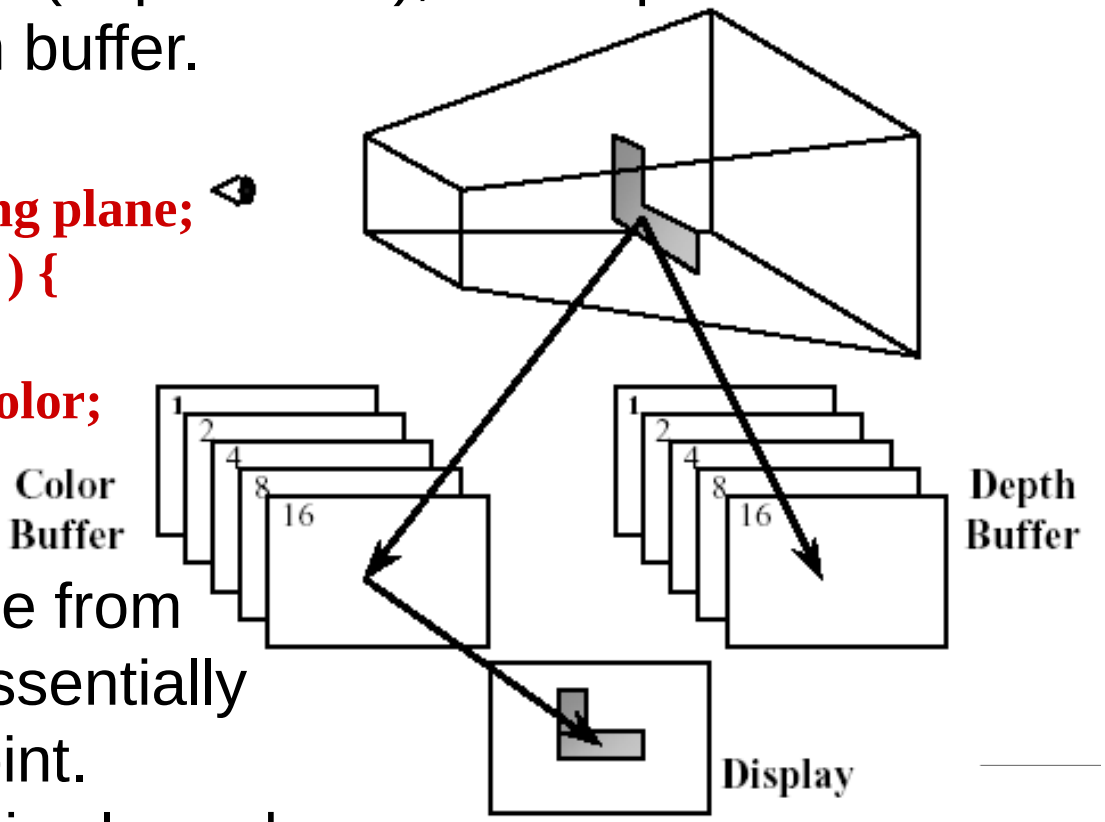
A.A.2018/2019

Depth Buffering and Hidden Surface Removal (z-buffer)

Using depth buffer, as each pixel in a primitive is rasterized, its distance from the eyepoint (depth value), is compared with the values stored in the depth buffer.

Z-buffer algorithm:

```
Set depthBuffer(:,:)->z=far clipping plane;
if ( pixel->z < depthBuffer(x,y)->z ) {
depthBuffer(x,y)->z = pixel->z;
colorBuffer(x,y)->color = pixel->color;
}
```



OpenGL depth values range from [0.0, 1.0], with 1.0 being essentially infinitely far from the eyepoint. Generally, the depth buffer is cleared to 1.0 at the start of a frame.

Depth Buffering Using OpenGL

The current framebuffer, whether an FBO (Framebuffer Object) or the default framebuffer, must have a depth buffer.

- Request a depth buffer

```
glutInitDisplayMode( GLUT_RGB |GLUT_DOUBLE |
                                    GLUT_DEPTH );
```

- Enable depth buffering

```
glEnable( GL_DEPTH_TEST );
```

- Clear Color and depth buffers

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

- Render scene
- Swap color buffers

Lighting

Turning on/off the Lights:

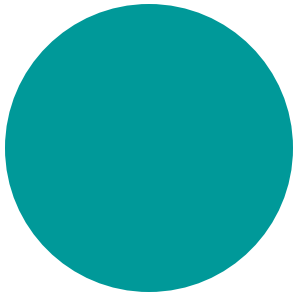
```
glEnable(GL_LIGHTING) // Enable Lighting
glDisable(GL_LIGHTING)// disable is default
```

Rendering techniques

OpenGL can render an object in wireframe mode, shading mode, textured mode

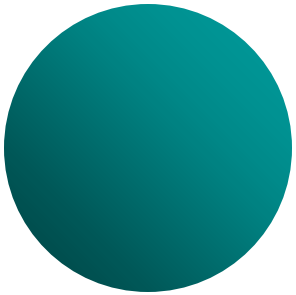
Why we need shading

- Suppose we build a model of a sphere using many polygons and color it with **`glColor`**. We get something like



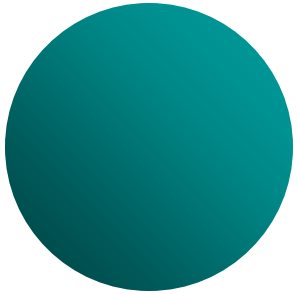
Is Lighting Enabled?
NO!? Final polygon colour is determined only by glColor()

- But we want



Shading

- Why does the image of a real sphere look like



- Light-material interactions cause each point to have a different color or shade
- Lighting contributors:
  - Light sources
  - Material properties
  - Location of viewer
  - Surface orientation
  - Illumination model

Material Properties

```
glMaterial{if}v(GLenum   face,
                GLenum   pname,
                TYPE     *param)
```

– *face*: separate materials for front and back:

GL_FRONT, GL_BACK, GL_FRONT_AND_BACK

- *pname*:

– **GL_DIFFUSE Base color**
– **GL_SPECULAR Highlight Color**
– **GL_AMBIENT Low-light Color**
– **GL_EMISSION Glow Color**
– **GL_SHININESS Surface Smoothness**

Values range from 0 (very rough surface - no highlight) to 128 (very shiny)

– *param*: parameter value

Let's make the sphere shiny.

Add the following material properties before you define the sphere:

```
GLfloat white[] = {0.8f, 0.8f, 0.8f, 1.0f};
GLfloat cyan[] = {0.f, .8f, .8f, 1.f};

glMaterialfv(GL_FRONT, GL_DIFFUSE, cyan);
glMaterialfv(GL_FRONT, GL_SPECULAR, white);

GLfloat shininess[] = {50};
glMaterialfv(GL_FRONT, GL_SHININESS, shininess);
```

More shiny means a smaller highlight.

Turning on the Lights

- Turning on the power (global control over lighting)

```
glEnable(GL_LIGHTING);
```

- Each OpenGL light is controllable separately, using

```
glEnable(GL_LIGHT0);
...
glEnable(GL_LIGHT7);
```

- Turning off the light:

```
glDisable(GL_LIGHT#);
```

At least 8 lights are supported, each identified by a light constant: GL_LIGHT*n*, *n* = 0, 1, …, 7

Types of Lights

- The type of light is determined by the *w coordinate of the* light's position (x,y,z,w).
- Directional (infinite)
  - Infinite light directed along (x,y,z,w) con w=0
- Point light (local)
  - Local Light positioned at (x,y,z,w) con w=1
  - the rays are emitted in all directions.
- Spotlight
  - the light rays are restricted to a cone of light

Specifying a Light Source

Position, Type, Color

```
glLight{if}v(Glenum light,Glenum pname,
                          TYPE *param)
```

- *Light#*: id light number: GL_LIGHT0, ..., GL_LIGHT7

- *pname*: parameter name (GL_POSITION, GL_AMBIENT, GL_DIFFUSE,GL_SPECULAR)
- *param*: parameter value (components RGB for light color, x,y,z for position, etc..)

pname in glLight

GL_POSITION → **Position (x,y,z,w)**

```
float light0Position[4] = {1.0, 0.0, 4.0, 1.0};
glLightfv(GL_LIGHT0, GL_POSITION,light0Position);
```

Type: point , directional, spotlight.

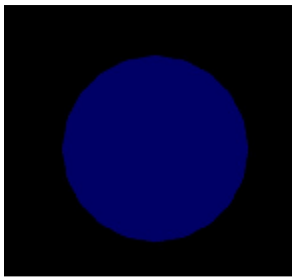
w component determines the type (1.0 in the example)
(w=0 directional lights, w=1 point/spot lights)

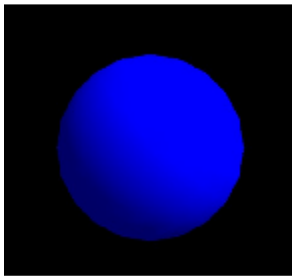
GL_AMBIENT, GL_DIFFUSE,GL_SPECULAR → **Color**

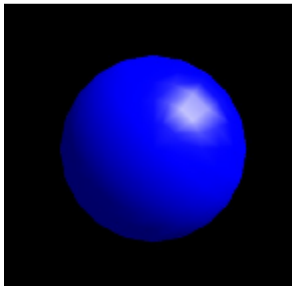
components in the range [0,1]: Ambient, Diffuse, Specular

pname in glLight:

OpenGL lights can emit different colors for each of a materials properties. For example, a light's GL_AMBIENT color is combined with a material's GL_AMBIENT color to produce the ambient contribution to the color -Likewise for the diffuse and specular colors.**(color (RGB))**







- GL_AMBIENT
- GL_DIFFUSE
- GL_SPECULAR

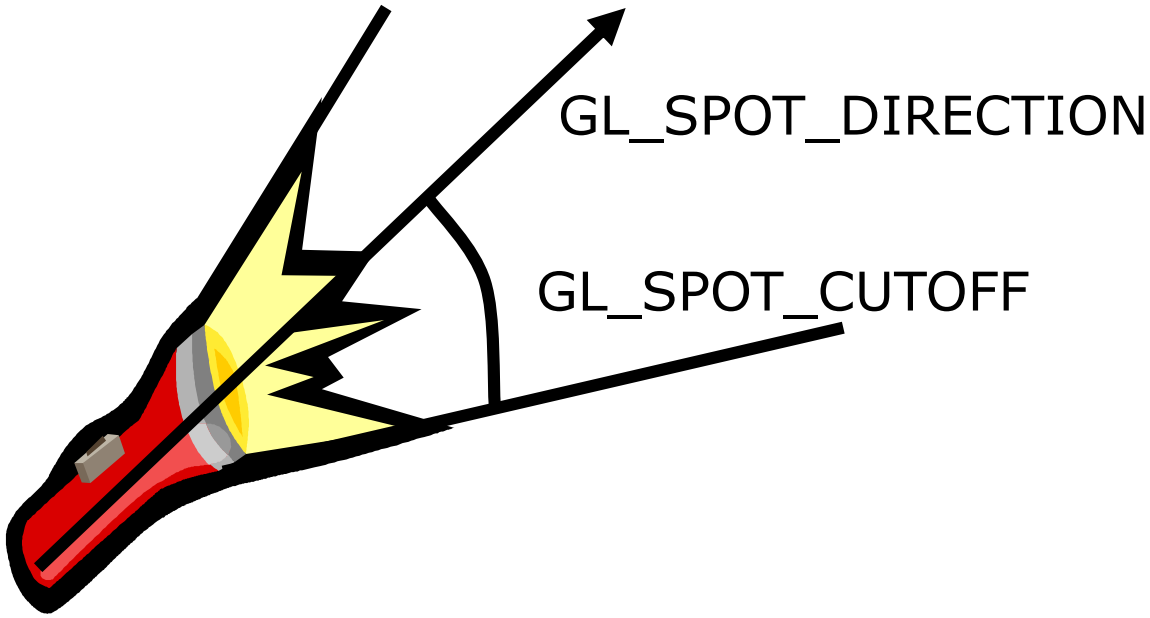
Spotlights

- **`glLight{if} (`*`lightNo`*`, GL_SPOT_CUTOFF, degree)`**
- **`glLight{if}v (`*`lightNo`*`, GL_SPOT_DIRECTION,`** *`spot_direction`*`)`

•The spot light, besides the position has a direction, *spotDirection*, which represents the axis of the cone.

•There is an angle of the cone *spotCosCutoff*.

•Finally we have a rate of decay, *spotExponent*, i.e. a measure of how the light intensity decreases from the center to the walls of the cone.



Example

specifying light source parameters

```
GLfloat position0[] = {1.0, 1.0, 1.0, 0.0};
GLfloat diffuse0[] = {1.0, 0.0, 0.0, 1.0}; // Id term - Red
GLfloat specular0[] = {1.0, 1.0, 1.0, 1.0};// Is term - White
GLfloat ambient0[] = {0.1, 0.1, 0.1, 1.0}; // Ia term - Gray

glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glLightfv(GL_LIGHT0, GL_POSITION, position0);
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse0);
glLightfv(GL_LIGHT0, GL_SPECULAR, specular0);
glLightfv(GL_LIGHT0, GL_AMBIENT, ambient0);
```

Moving Light Sources

- Light sources are geometric objects whose positions or directions are affected by the model-view matrix
- Depending on where we place the position (direction) setting function, we can
  - Move the light source(s) with the object(s)
  - Fix the object(s) and move the light source(s)
  - Fix the light source(s) and move the object(s)
  - Move the light source(s) and object(s) independently

Controlling a Light's Position

- Modelview matrix affects a light's position
- Different effects based on when position is specified
  - light position fixed relative to my eye position:VCS

    CTM=I

    then specify your light position.
  - light stay fixed relative to the scene: WCS

    Set the view transform (with gluLookAt). CTM=Tv

    Set the light position //glLightfv(GL_LIGHT_POSITION,…)
  - light that moves around in a scene: OCS

    Set the view transform CTM=Tv*Tm

    Push the matrix stack

    Set the model transform to update the light's position

    Set the light position //glLightfv(GL_LIGHT_POSITION,…)

    Pop the matrix stack

Distance Terms

- Attenuation controls the natural tendency of light to decay over distance.
- The light from a point source that reaches a surface is inversely proportional to the square of the distance between them

$$f_{att}(d) = \frac{1}{a + bd + cd^2}$$

`glLightf(Glenum light,Glenum pname, float k)`

- **a** = GL_CONSTANT_ATTENUATION (default 1.0)
- **b** = GL_LINEAR_ATTENUATION (default 0.0)
- **c** = GL_QUADRATIC_ATTENUATION (default 0.0)
- Default is no attenuation: a=1, b=0, c=0
- No attenuation for directional lights

Tips for Better Lighting

- Computes a color for each vertex
- Vertex colors are interpolated across polygons by the rasterizer
- Recall lighting computed only at vertices
  - – model tessellation heavily affects lighting results
    - better results but more geometry to process
- Use a single infinite light for fastest lighting
    - minimal computation per vertex

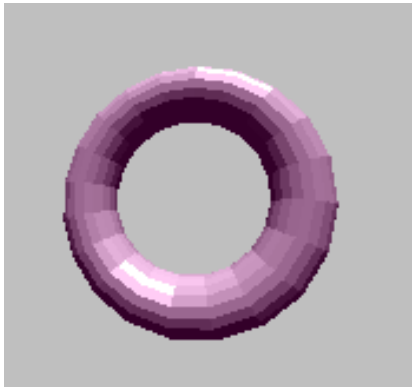
Light Material Tutorial
Light Position Tutorial

OpenGL Shading

OpenGL allows for two shading models:

flat

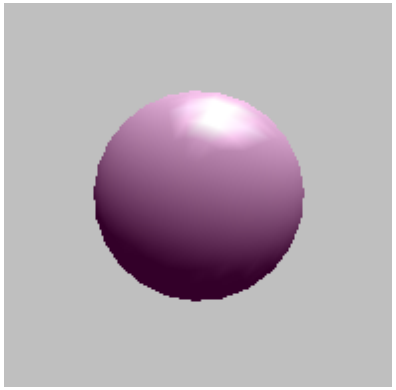
`glShadeModel(GL_FLAT)`

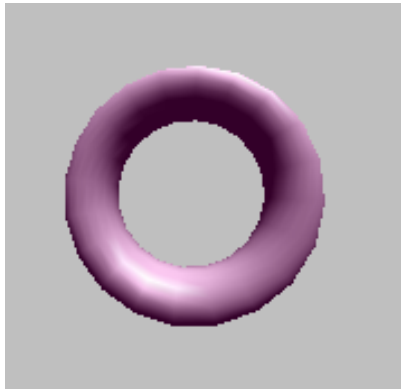


smooth (Gourand shading)

`glShadeModel(GL_SMOOTH)`

Phong shading by shaders





Surface Normals

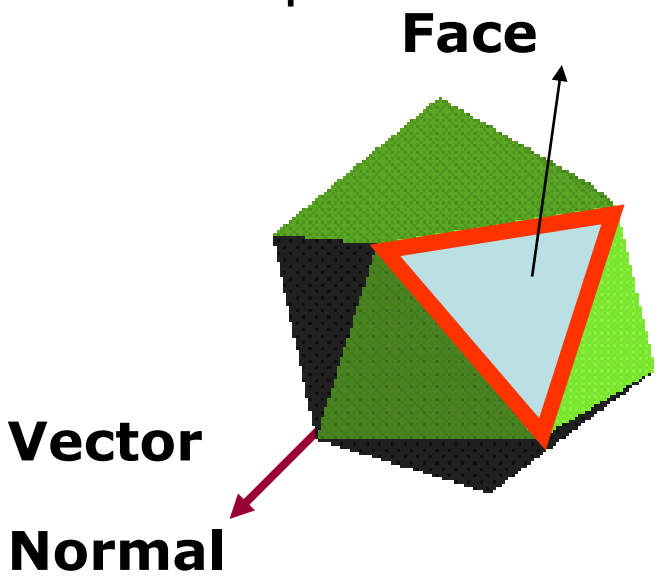
The normal vectors have to be explicitly computed and assigned to the vertices :

```
glNormal3f( x, y, z )
```

Sets the current normal, which is used in the lighting computation for all vertices until a new normal is provided.

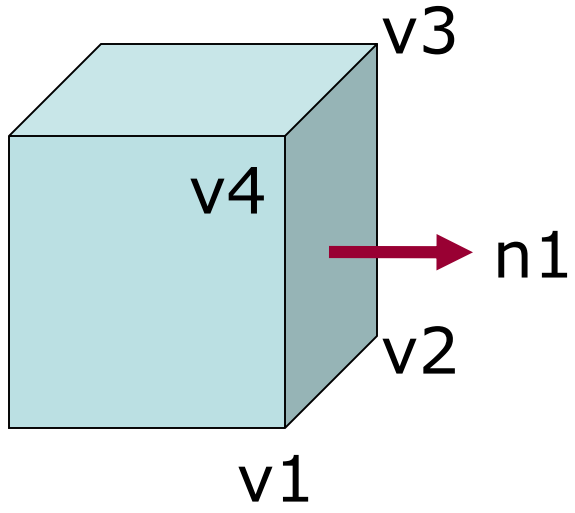
Automatically normalize normals,
by enabling

```
glEnable( GL_NORMALIZE )
```



Rendering example (FLAT)

```
glShadeModel(GL_FLAT);
glBegin(GL_POLYGON)
  glNormal3fv(n1);
  glVertex3fv(v1);
  glVertex3fv(v2);
  glVertex3fv(v3);
  glVertex3fv(v4);
glEnd();
```



Rendering example (Gouraud)

```
glShadeModel(GL_SMOOTH);
glBegin(GL_POLYGON);
   glNormal3fv(n1);
   glVertex3fv(v1);
   glNormal3fv(n2);
   glVertex3fv(v2);
   glNormal3fv(n3);
   glVertex3fv(v3);
   glNormal3fv(n4);
   glVertex3fv(v4);
glEnd( );
```