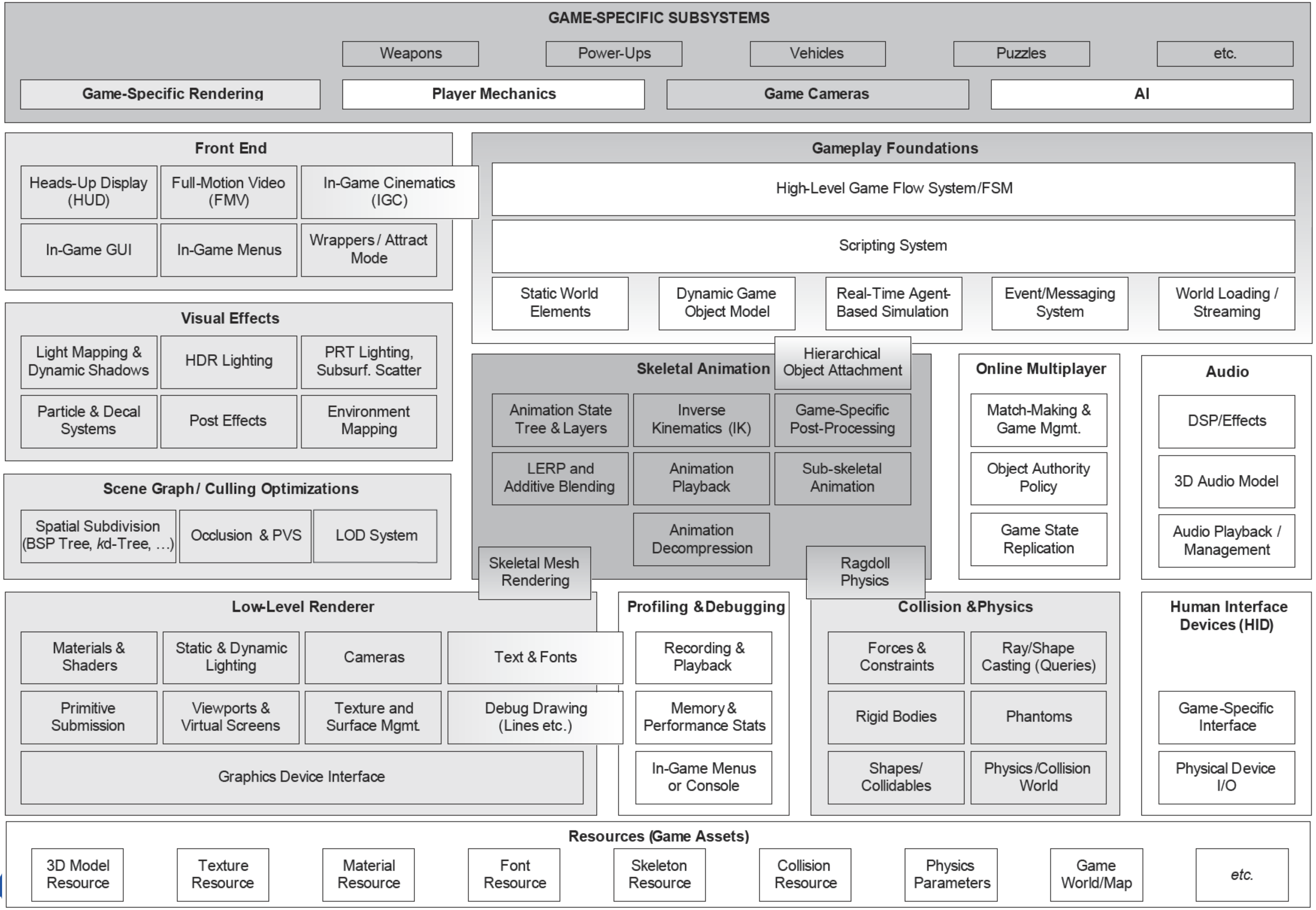
Design of Game Software

- OpenGL
- Transformations and Viewing
- Illuminations

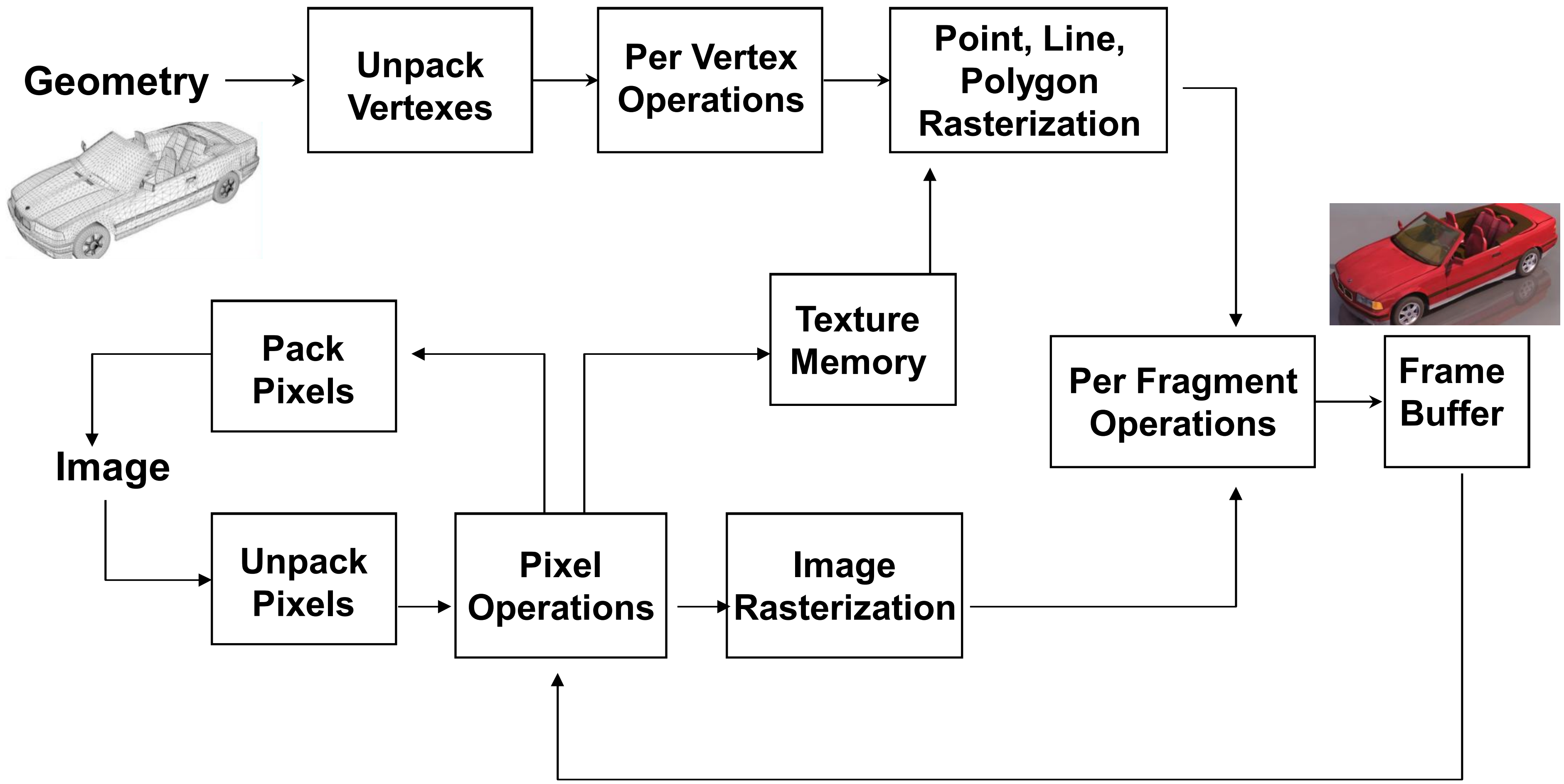
Game Architecture



What Is OpenGL?

- Graphics rendering API
  - high-quality color images composed of geometric and image primitives
  - window system independent
  - operating system independent

OpenGL Architecture



OpenGL as a Renderer

- Geometric primitives
  - points, lines and polygons
- Image Primitives
  - images and bitmaps
  - separate pipeline for images and geometry
    - linked through texture mapping
- Rendering depends on state
  - colors, materials, light sources, etc.

Preliminaries

- Headers Files
  - #include <GL/gl.h>
  - #include <GL/glu.h>
  - #include <GL/glut.h>
- Libraries
  - ex) opengl32.lib (Windows), libGL.so (Unix)
- Enumerated Types
  - OpenGL defines numerous types for compatibility
    - GLfloat, GLint, GLenum, etc.

Preliminaries

- Naming Conventions
  - Functions: begin with gl
  - Constants: begin with GL_
  - Types: begin with GL

Suffix	OpenGL Datatype	C/C++ Datatype
b s i	GLbyte GLshort GLint	signed char short int
ub us ui	GLubyte GLushort GLuint	unsigned char unsigned short unsigned int
f d	GLfloat GLdouble	float double

OpenGL Data Types

GLUT Basics

- Application Structure
  - Configure and open window
  - Initialize OpenGL state
  - Register input callback functions
    - render
    - resize
    - input: keyboard, mouse, etc.
  - Enter event processing loop

Sample Program

```
void main( int argc, char** argv )
{
  int mode = GLUT_RGB|GLUT_DOUBLE;
  glutInitDisplayMode( mode );
  glutCreateWindow( argv[0] );
  init();
  glutDisplayFunc( display );
  glutReshapeFunc( resize );
  glutKeyboardFunc( key );
  glutIdleFunc( idle );
  glutMainLoop();
}
```

OpenGL Initialization

- Set up whatever state you're going to use

```
void init( void )
{
  glClearColor( 0.0, 0.0, 0.0, 1.0 );
  glClearDepth( 1.0 );

  glEnable( GL_LIGHT0 );
  glEnable( GL_LIGHTING );
  glEnable( GL_DEPTH_TEST );
}
```

GLUT Callback Functions

- Routine to call when something happens
  - window resize or redraw
  - user input
  - animation
- "Register" callbacks with GLUT

```
glutDisplayFunc( display );
glutIdleFunc( idle );
glutKeyboardFunc( keyboard );
```

Rendering Callback

- Do all of your drawing here

```
glutDisplayFunc( display );
```

```
void display( void )
{
  glClear( GL_COLOR_BUFFER_BIT );
  glBegin( GL_TRIANGLE_STRIP );
    glVertex3fv( v[0] );
    glVertex3fv( v[1] );
    glVertex3fv( v[2] );
    glVertex3fv( v[3] );
  glEnd();
  glutSwapBuffers();
}
```

Idle Callbacks

- Use for animation and continuous update

```
glutIdleFunc( idle );
```

```
void idle( void )
{
  t += dt;
  glutPostRedisplay();
}
```

User Input Callbacks

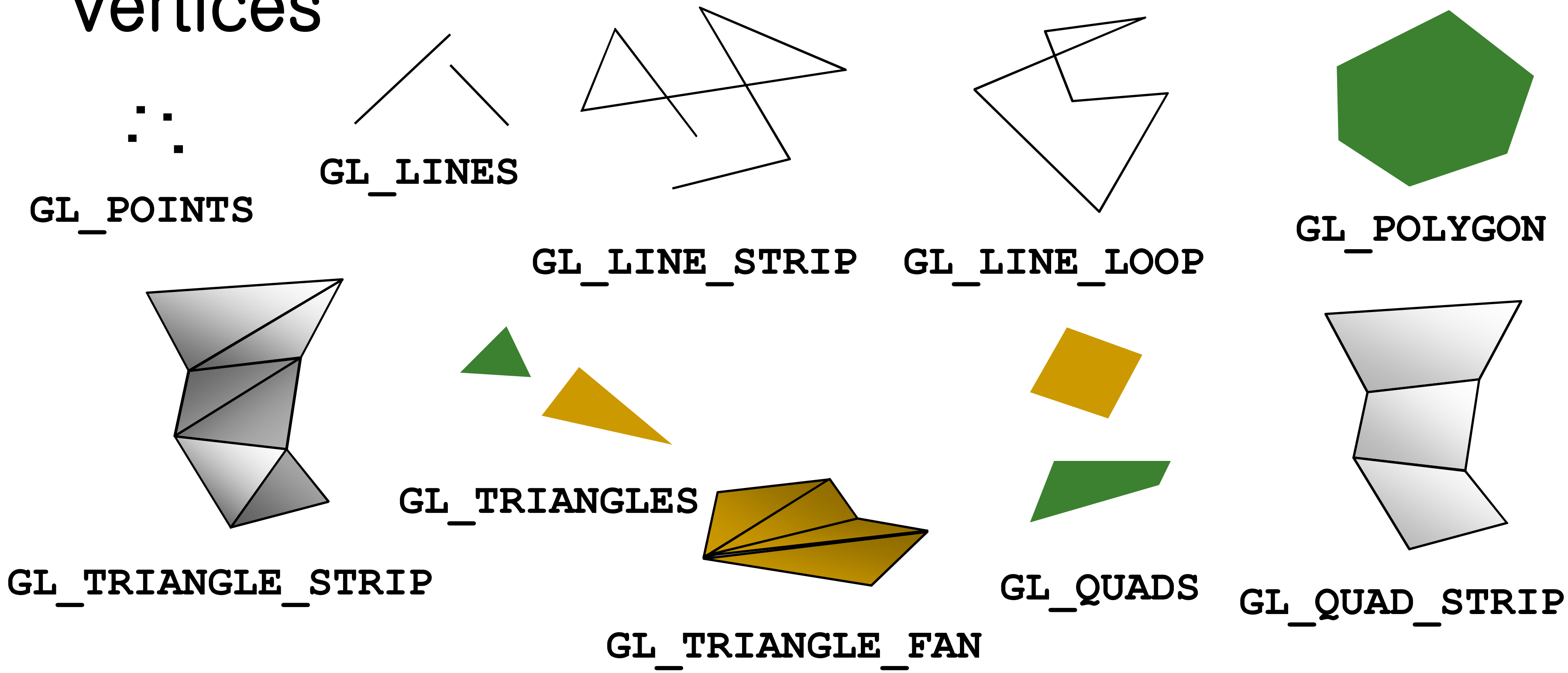
- Process user input

```
glutKeyboardFunc( keyboard );
```

```
void keyboard( unsigned char key, int x, int y )
{
  switch( key ) {
    case 'q' : case 'Q' :
      exit( EXIT_SUCCESS );
      break;
    case 'r' : case 'R' :
      rotate = GL_TRUE;
      glutPostRedisplay();
      break;
  }
}
```

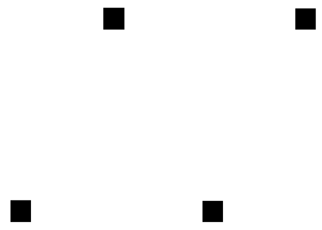
OpenGL Geometric Primitives

- All geometric primitives are specified by vertices



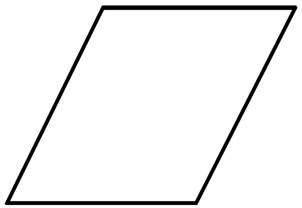
Simple Example

```
void drawPoints( GLfloat color[] )
{
    glBegin( GL_POINTS );
    glColor3fv( color );
    glVertex2f( 0.0, 0.0 );
    glVertex2f( 1.0, 0.0 );
    glVertex2f( 1.5, 1.0 );
    glVertex2f( 0.5, 1.0 );
    glEnd();
}
```



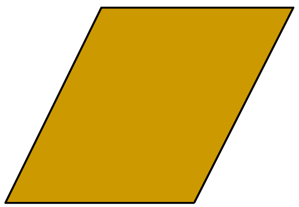
Simple Example

```
void drawLines( GLfloat color[] )
{
    glBegin( GL_LINE_LOOP );
    glColor3fv( color );
    glVertex2f( 0.0, 0.0 );
    glVertex2f( 1.0, 0.0 );
    glVertex2f( 1.5, 1.0 );
    glVertex2f( 0.5, 1.0 );
    glEnd();
}
```



Simple Example

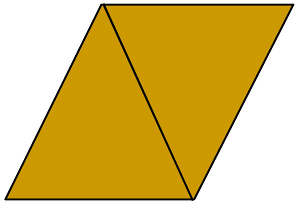
```
void drawRhombus( GLfloat color[] )
{
    glBegin( GL_QUADS );
    glColor3fv( color );
    glVertex2f( 0.0, 0.0 );
    glVertex2f( 1.0, 0.0 );
    glVertex2f( 1.5, 1.0 );
    glVertex2f( 0.5, 1.0 );
    glEnd();
}
```



Simple Example

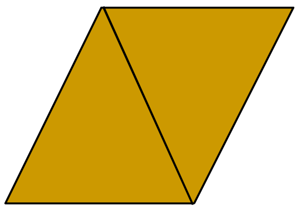
```
void drawTriangles( GLfloat color[] )
{
  glBegin( GL_TRIANGLES );
   glColor3fv( color );
   glVertex2f( 0.0, 0.0 );
   glVertex2f( 1.0, 0.0 );
   glVertex2f( 0.5, 1.0 );

   glVertex2f( 0.5, 1.0 );
   glVertex2f( 1.0, 0.0 );
   glVertex2f( 1.5, 1.0 );
  glEnd();
}
```



Simple Example

```
void drawTriangleStrips( GLfloat color[] )
{
  glBegin( GL_TRIANGLE_STRIP );
   glColor3fv( color );
   glVertex2f( 0.0, 0.0 );
   glVertex2f( 1.0, 0.0 );
   glVertex2f( 0.5, 1.0 );
   glVertex2f( 1.5, 1.0 );
  glEnd();
}
```



OpenGL Command Formats

```
glVertex3fv( v )
```

Number of components

```
2 - (x,y)
3 - (x,y,z)
4 - (x,y,z,w)
```

Data Type

```
b  - byte
ub - unsigned byte
s  - short
us - unsigned short
i  - int
ui - unsigned int
f  - float
d  - double
```

Vector

```
omit "v" for
scalar form

glVertex2f( x, y )
```

Specifying Geometric Primitives

- Primitives are specified using

```
glBegin( primType );
glEnd();
```

  - *primType* determines how vertices are combined

```
GLfloat red, green, blue;
Glfloat coords[3];
glBegin( primType );
for ( i = 0; i < nVerts; ++i ) {
  glColor3f( red, green, blue );
  glVertex3fv( coords );
}
glEnd();
```

primType can be one of:

- GL_POINTS
- GL_LINES
- GL_POLYGON
- GL_LINE_STRIP
- GL_TRIANGLE_STRIP
- GL_TRIANGLES
- GL_QUADS
- GL_LINE_LOOP
- GL_QUAD_STRIP
- GL_TRIANGLE_FAN

OpenGL Color Models

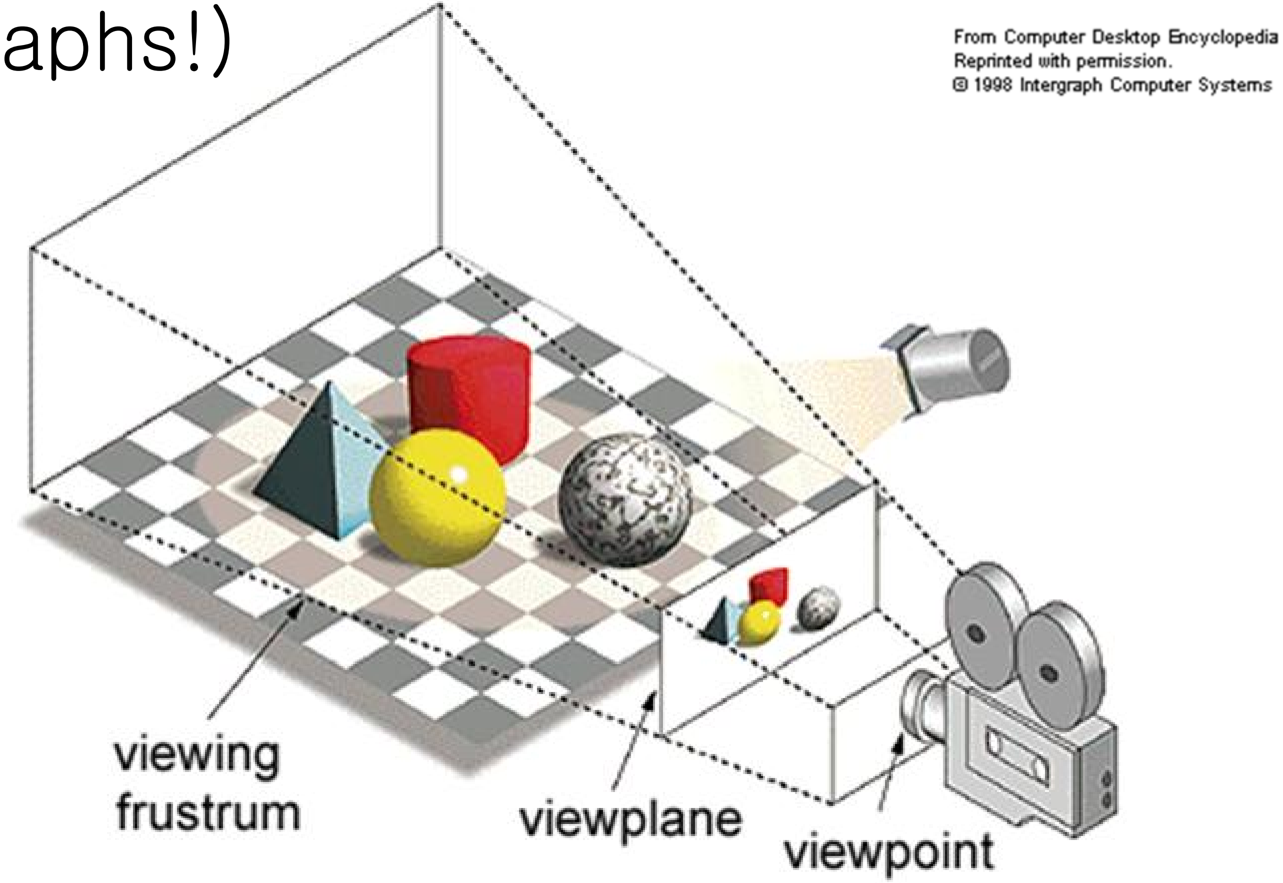
- RGBA or Color Index
  - `float color[] = {1.0, 0.0, 0.0}; glColor3fv( color );` ➡ **Red**
  - `glColor3f( 1.0, 0.0, 1.0 );` ➡ **Purple**

Red **Green** **Blue**

Viewing Systems

Camera Analogy

- 3D is just like taking a photograph (lots of photographs!)



Camera Analogy and Transformations

- Projection transformations
  - adjust the lens of the camera
- Viewing transformations
  - tripod-define position and orientation of the viewing volume in the world
- Modeling transformations
  - moving the model
- Viewport transformations
  - enlarge or reduce the physical photograph

3D Transformations

- A vertex is transformed by 4 x 4 matrices
  - all affine operations are matrix multiplications
  - all matrices are stored column-major in OpenGL
  - matrices are always post-multiplied
  - product of matrix and vector is $\mathbf{M}\vec{v}$

$$\mathbf{M} = \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

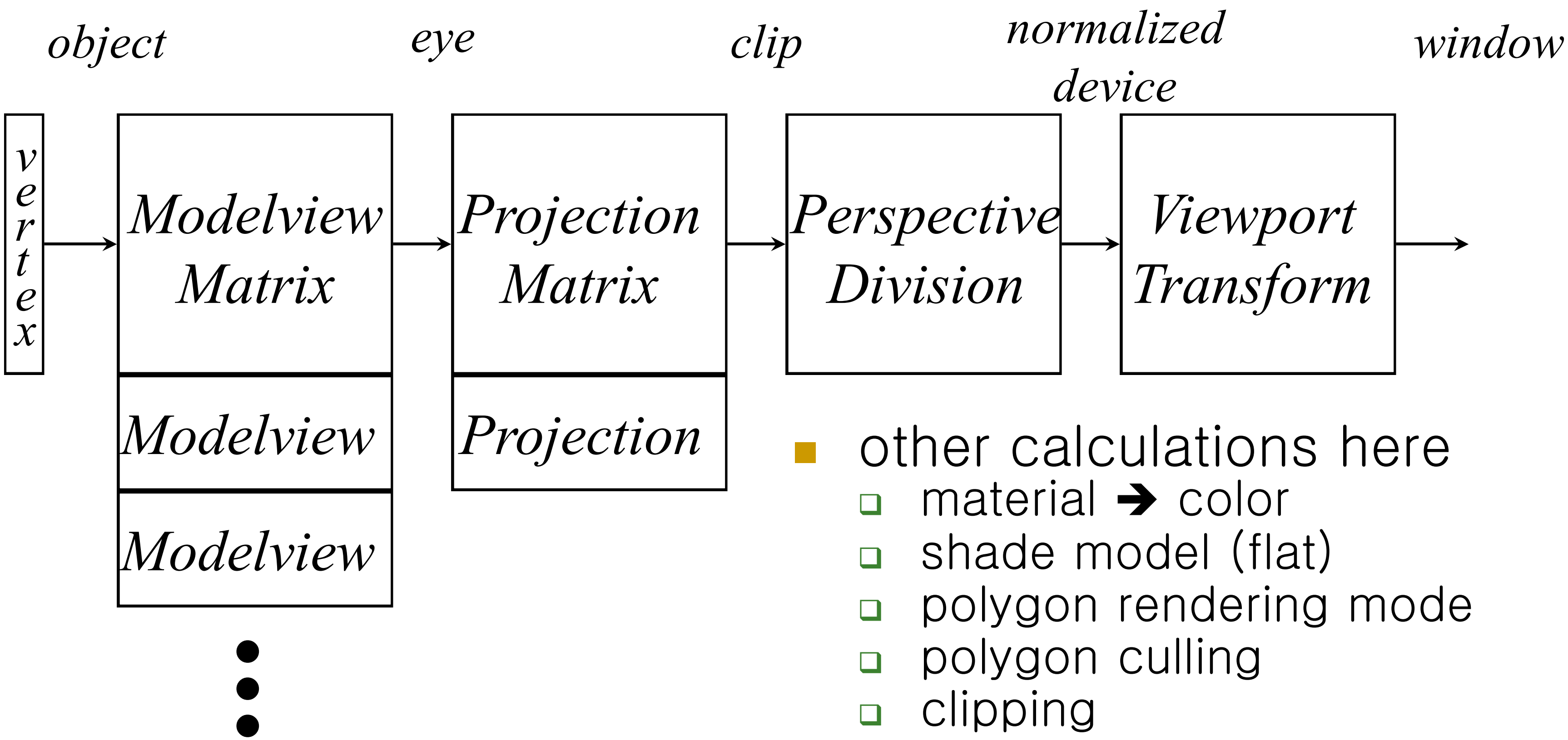
Specifying Transformations

- Programmer has two styles of specifying transformations
  - specify matrices (**`glLoadMatrix, glMultMatrix`**)
  - specify operation (**`glRotate, glOrtho`**)

Programming Transformations

- Prior to rendering, view, locate, and orient:
  - eye/camera position
  - 3D geometry
- Manage the matrices
  - including matrix stack
- Combine (composite) transformations

Transformation Pipeline



- other calculations here
  - material ➔ color
  - shade model (flat)
  - polygon rendering mode
  - polygon culling
  - clipping

Matrix Operations

- Specify Current Matrix Stack

  `glMatrixMode(` ***`GL_MODELVIEW`*** **`or`** ***`GL_PROJECTION`*** **`)`**

- Other Matrix or Stack Operations

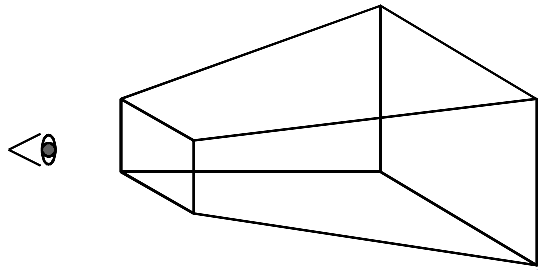
  `glLoadIdentity()`　　**`glPushMatrix()`**

  `glPopMatrix()`

- Viewport
  - usually same as window size
  - viewport aspect ratio should be same as projection transformation or resulting image may be distorted

  `glViewport(` ***`x, y, width, height`*** **`)`**

Projection Transformation



- Shape of viewing frustum
- Perspective projection

  gluPerspective(fovy, aspect, zNear, zFar)

  glFrustum(left, right, bottom, top, zNear, zFar)
- Orthographic parallel projection

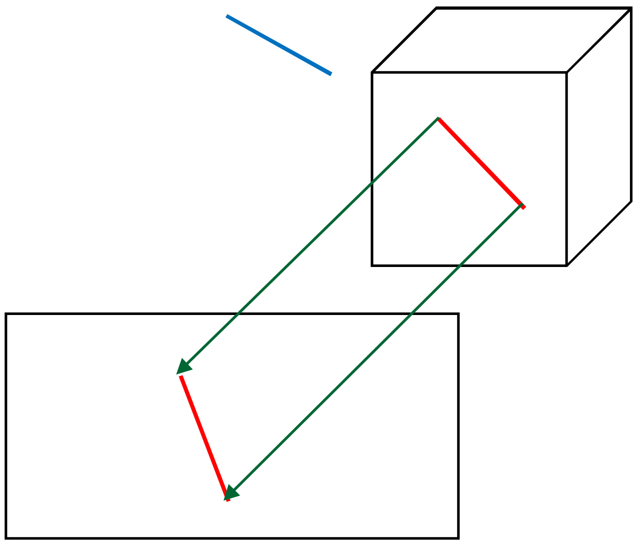
  glOrtho(left, right, bottom, top, zNear, zFar)

  gluOrtho2D(left, right, bottom, top)
  - calls **glOrtho** with z values near zero

Applying Projection Transformations

- Typical use (orthographic projection)

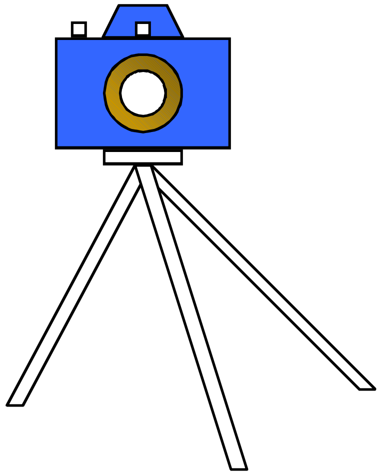
```
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
glOrtho( left, right, bottom, top, zNear, zFar );
```



Viewing Transformations

- Position the camera/eye in the scene
  - place the tripod down; aim camera
- To “fly through” a scene
  - change viewing transformation and redraw scene
- $\texttt{gluLookAt(eye}_x\texttt{, eye}_y\texttt{, eye}_z\texttt{,}$
  $\texttt{aim}_x\texttt{, aim}_y\texttt{, aim}_z\texttt{,}$
  $\texttt{up}_x\texttt{, up}_y\texttt{, up}_z\texttt{)}$
  - up vector determines unique orientation
  - careful of degenerate positions

tripod



Modeling Transformations

- Move object

  `glTranslate{fd}(` ***`x, y, z`*** **`)`**

- Rotate object around arbitrary axis $\begin{pmatrix} x & y & z \end{pmatrix}$

  `glRotate{fd}(` ***`angle, x, y, z`*** **`)`**

  - angle is in degrees

- Dilate (stretch or shrink) or mirror object

  `glScale{fd}(` ***`x, y, z`*** **`)`**

- Transformation

  `glMultMatrix{fd}(` ***`m`*** **`)`**

  `glLoadMatrix{fd} (` ***`m`*** **`)`**

Common Transformation Usage

- 3 examples of `resize()` routine
  - restate projection & viewing transformations
- Usually called when window resized
- Registered as callback for `glutReshapeFunc()`

resize(): Perspective & LookAt

```
void resize( int w, int h )
{
    glViewport( 0, 0, (GLsizei) w, (GLsizei) h );
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluPerspective( 65.0, (GLdouble) w / h,
                    1.0, 100.0 );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    gluLookAt( 0.0, 0.0, 5.0,
               0.0, 0.0, 0.0,
               0.0, 1.0, 0.0 );
}
```

`resize()`: Perspective & Translate

- Same effect as previous LookAt

```
void resize( int w, int h )
{
   glViewport( 0, 0, (GLsizei) w, (GLsizei) h );
   glMatrixMode( GL_PROJECTION );
   glLoadIdentity();
   gluPerspective( 65.0, (GLdouble) w/h,
                   1.0, 100.0 );
   glMatrixMode( GL_MODELVIEW );
   glLoadIdentity();
   glTranslatef( 0.0, 0.0, -5.0 );
}
```

`resize()`: Ortho (part 1)

```
void resize( int width, int height )
{
   GLdouble aspect = (GLdouble) width / height;
   GLdouble left = -2.5, right = 2.5;
   GLdouble bottom = -2.5, top = 2.5;
   glViewport( 0, 0, (GLsizei) w, (GLsizei) h );
   glMatrixMode( GL_PROJECTION );
   glLoadIdentity();
```

… continued …

`resize()`: Ortho (part 2)

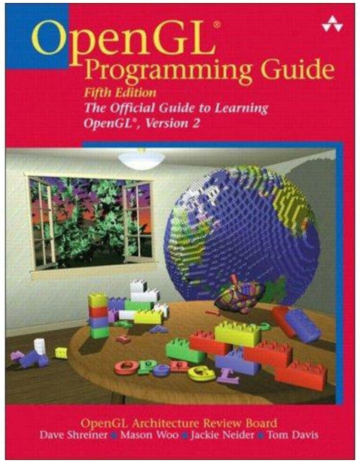
```
   if ( aspect < 1.0 ) {
      bottom /= aspect;
      top /= aspect;
   } else {
      left *= aspect;
      right *= aspect;
   }
   glOrtho( left, right, bottom, top, near, far );
   glMatrixMode( GL_MODELVIEW );
   glLoadIdentity();
}
```

On-Line Resources

- **http://www.opengl.org**
- **http://nehe.gamedev.net/**
- **http://www.mesa3d.org/**

Books

- OpenGL Programming Guide
  - Ver 1.1: http://www.glprogramming.com/red/



- OpenGL Reference Manual

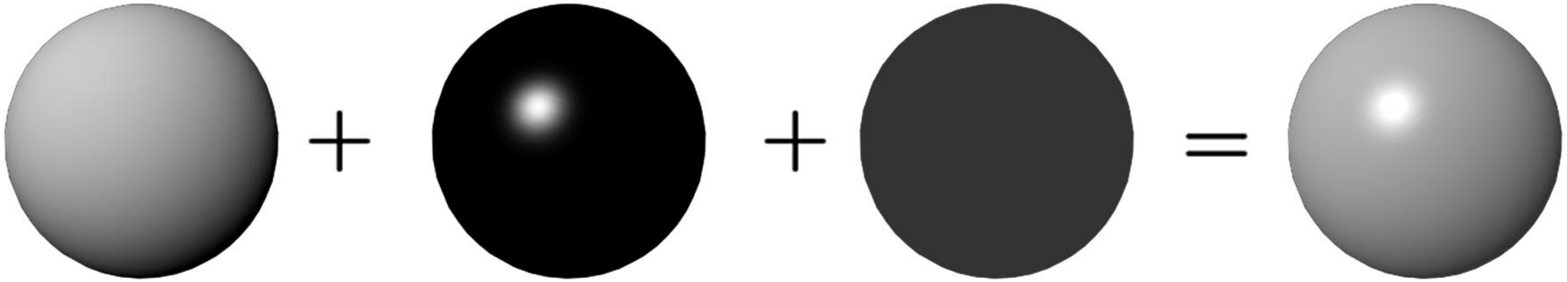
Illumination

Illumination

- Local Illumination Models
  - Point light sources and direct interaction with light
  - Simple diffuse and specular approximations
- Shading
  - Determining the intensity of illumination incident at a surface point
  - Compute it at vertices and interpolate in between

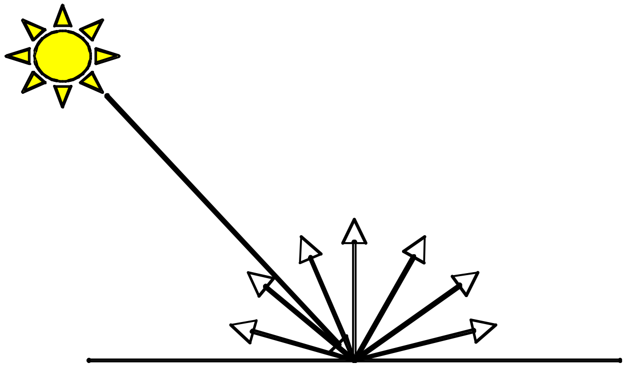
A Simple Model

- Approximate local illumination as sum of
  - A diffuse component
  - A specular component
  - A "ambient" term



Diffuse Component

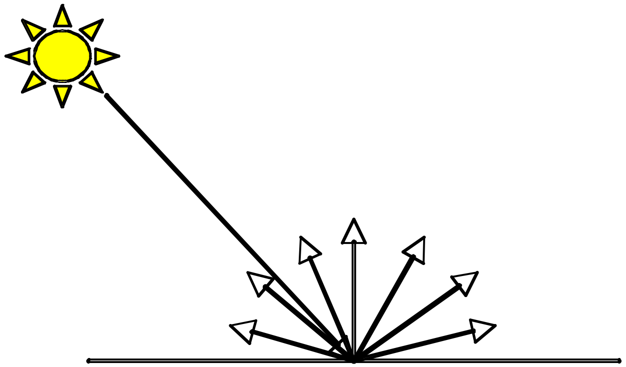
- Lambert's Law
  - Intensity of reflected light proportional to cosine of angle between surface and incoming light direction
  - Applies to "diffuse" or "Lambertian" surfaces
  - Independent of viewing angle

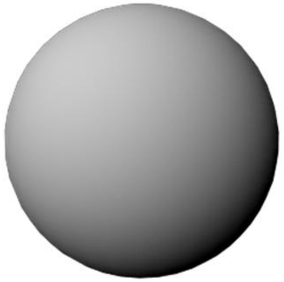


Diffuse Component

$$k_d I(\hat{\mathbf{l}} \cdot \hat{\mathbf{n}})$$

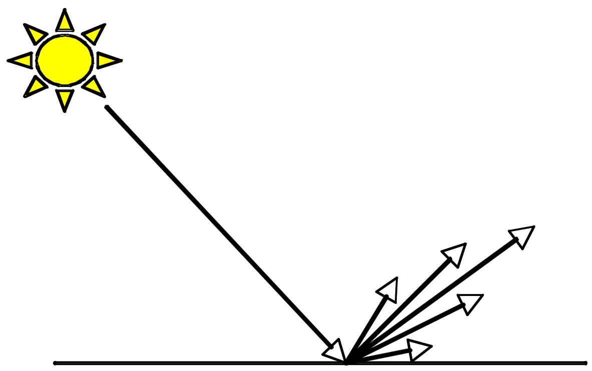
$$\max(k_d I(\hat{\mathbf{l}} \cdot \hat{\mathbf{n}}), 0)$$





Specular Component

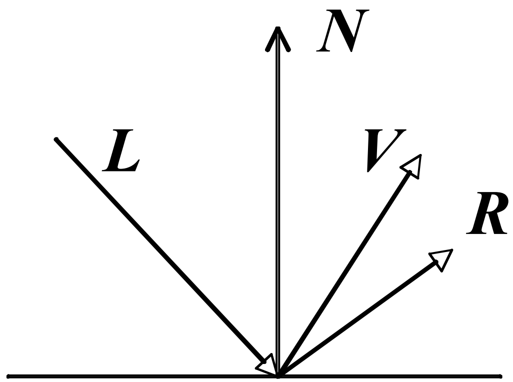
- A mirror-like reflection
- Phong Illumination Model
  - A reasonable approximation for some surfaces
  - Fairly cheap to compute
- Depends on view direction

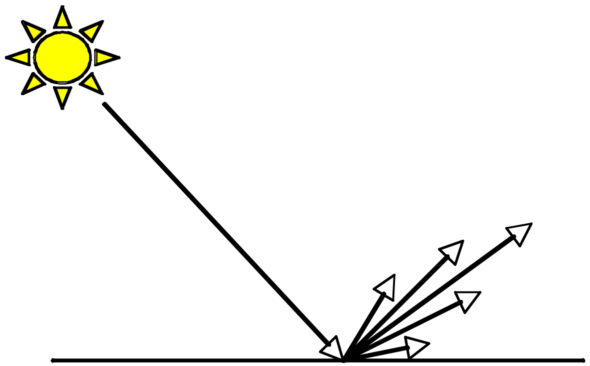


Specular Component

$$k_s I \left(\hat{\mathbf{r}} \cdot \hat{\mathbf{v}}\right)^p$$

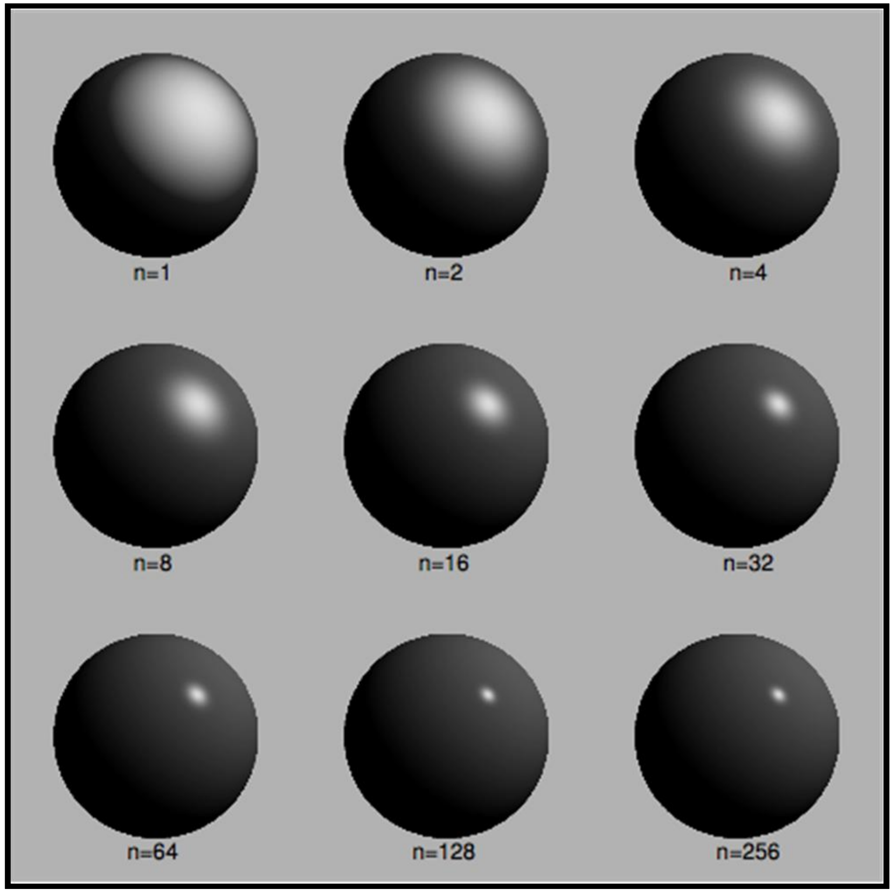
$$k_s I \max\left(\hat{\mathbf{r}} \cdot \hat{\mathbf{v}}, 0\right)^p$$





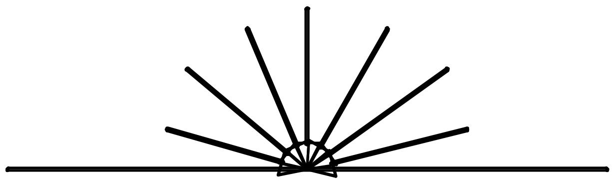
Specular Component

- Specular exponent sometimes called "roughness"



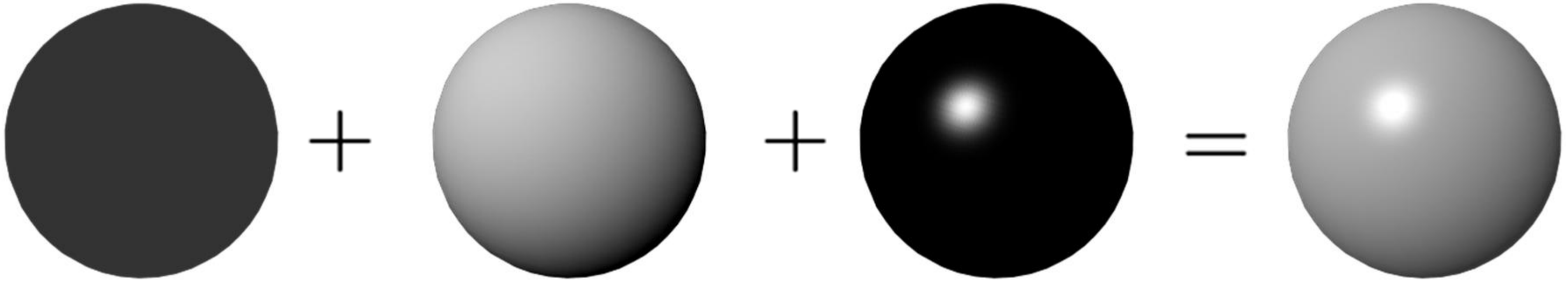
Ambient Term

- A background glow that illuminates all objects, irrespective of light source location
- Accounts for “ambient, omnidirectional light”



Summing the Parts

- $R = k_a I + k_d I \max(\hat{\mathbf{l}} \cdot \hat{\mathbf{n}}, 0) + k_s I \max(\hat{\mathbf{r}} \cdot \hat{\mathbf{v}}, 0)^p$

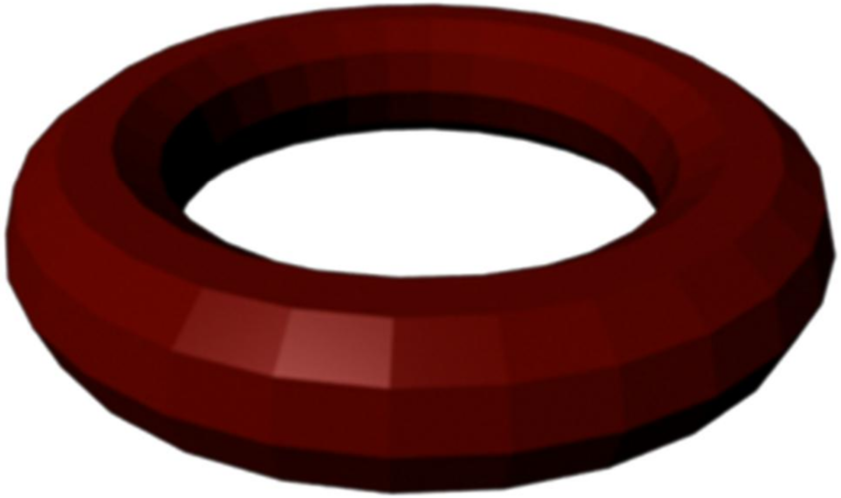


Shading

- Flat shading
- Gouraud shading
- Phong shading

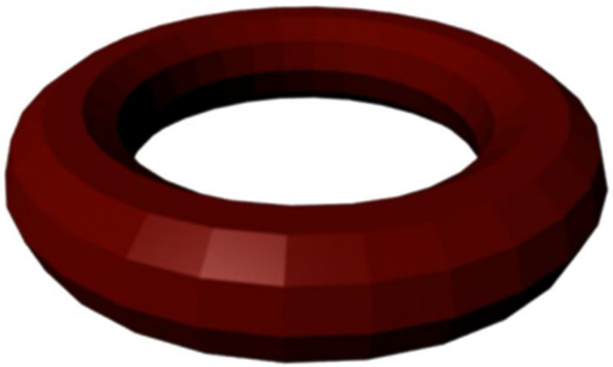
Flat Shading

- A single normal for each triangle (polygon)
  - A faceted appearance

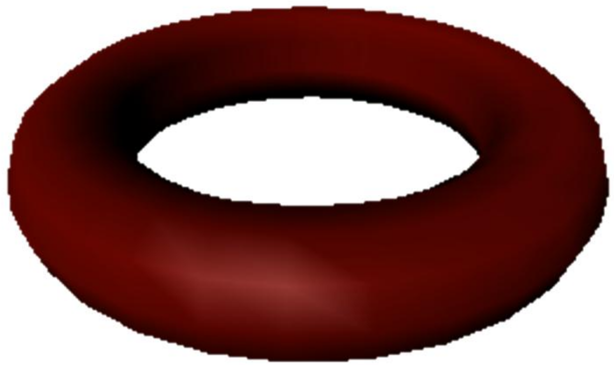


Gouraud Shading

- Compute shading at each vertex
  - Interpolate colors from vertices
  - Pros: fast and easy, looks smooth
  - Cons: terrible for specular reflections



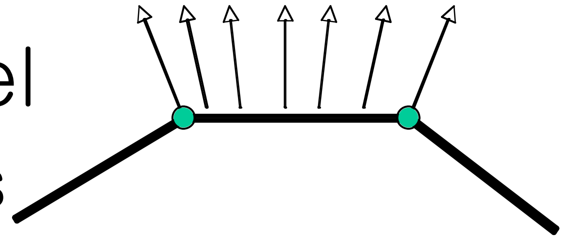
Flat

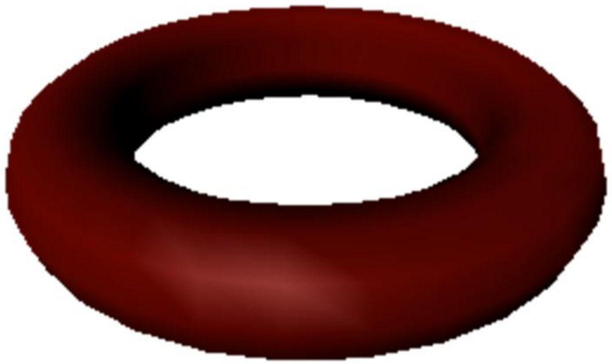


Gouraud

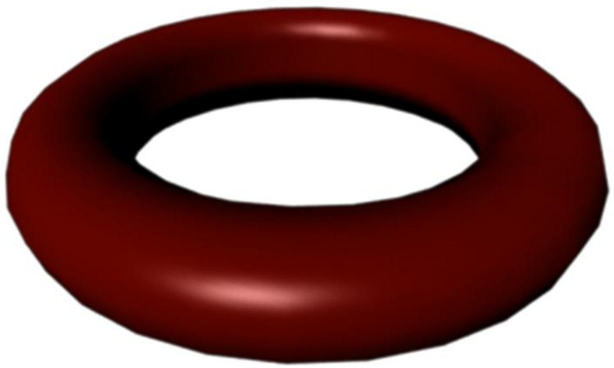
Phong Shading

- Compute shading at each pixel
  - Interpolate *normals* from vertices
  - Pros: looks smooth, better speculars
  - Cons: expensive





Gouraud



Phong

Lighting in OpenGL

- **Ambient, diffuse, specular** illuminations are supported
- Users define:
  - Shading model: flat or smooth
  - Light sources: position, color, type
  - Object materials: color, shininess, …
- Enabling (turn on/off)
  - glEnable (GL_LIGHTING);
  - glDisable (GL_LIGHTING);

OpenGL Shading

- OpenGL supports flat and Gouraud shading.
  No support for Phong shading yet.
- glShadeModel(GL_FLAT)
  - Flat shading
- glShadeModel(GL_SMOOTH)
  - Gouraud shading
- Remember to supply normals with triangles or vertices to get correct lighting and shading

Lights

- At least 8 lights: **GL_LIGHT0, ..., GL_LIGHT7**
- To get maximum lights in your program:

  glGetIntegerv(GL_MAX_LIGHTS, GLint *num_lights);
- You can turn on/off each light (disabled by default)

  glEnable(GL_LIGHT0);

  glEnable(GL_LIGHT1); ...

  glEnable(GL_LIGHTING);

glLight*()

- glLight{if}(GLenum *light*, GLenum *pname*, TYPE *param*)

  glLight{if}v(GLenum *light*, GLenum *pname*, TYPE **param*)
- *light* can be: GL_LIGHT0, ..., GL_LIGHT7
- *pname* can be one of following:
  - GL_POSITION: light position
  - GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR : light colors
  - GL_SPOT_DIRECTION, GL_SPOT_EXPONENT, GL_SPOT_CUTOFF: spotlight parameters
  - GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION, GL_QUADRATIC_ATTENUATION: parameters for attenuation

Light Position

```
GLfloat lightA_position[ ] = {1.0, 1.0, 1.0, 0.0};
GLfloat lightB_position[ ] = {1.0, 2.0, 3.0, 1.0};
```

- A *directional* light source coming from the direction (1, 1, 1)

```
glLightfv(GL_LIGHT0, GL_POSITION, lightA_position);
```

- A *positional* light source located at the point (1, 2, 3) in the world coordinates.

```
glLightfv(GL_LIGHT1, GL_POSITION, lightB_position);
```

Example

```
GLfloat ambientIntensity[4] = { 0.9, 0.0, 0.0, 1.0 };  // red
GLfloat diffSpecIntensity[4] = { 1.0, 1.0, 1.0, 1.0 };  // white
GLfloat position[4] = { 2.0, 4.0, 5.0, 1.0 };

glShadeModel ( GL_SMOOTH );                     // (or GL_FLAT)
glEnable ( GL_LIGHTING );                       // enable lighting
glEnable ( GL_LIGHT0 );                         // enable light 0
   // set up light 0 properties
glLightfv ( GL_LIGHT0, GL_AMBIENT, ambientIntensity );
glLightfv ( GL_LIGHT0, GL_DIFFUSE, diffSpecIntensity );
glLightfv ( GL_LIGHT0, GL_SPECULAR, diffSpecIntensity );
glLightfv ( GL_LIGHT0, GL_POSITION, position );
```

Object Materials

- Object colors under illumination are computed as a component-wise multiplication of the light colors and material colors
- Material colors are specified for each of ambient, diffuse, and specular illuminations
- In addition to this emissive material color is also defined:
  - Lights don't influence emissive material
  - Emissive objects don't add further light to environment

glMaterial*()

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

  glMaterial{if}v(GLenum *face*, GLenum *pname*, TYPE **param*)
- *face* can be: GL_FRONT, GL_BACK, GL_FRONT_AND_BACK
- *pname* can be:
  - GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_EMISSION: material colors
  - GL_SHININESS: Specular (Phong) illumination exponent

glMaterial*()

```
GLfloat mat0_ambient[ ] = {0.2, 0.2, 0.2, 1.0};
GLfloat mat0_diffuse[ ] = {0.7, 0.0, 0.0, 1.0};
GLfloat mat0_specular[ ] = {1.0, 1.0, 1.0, 1.0};
GLfloat mat0_shininess[ ] = {5.0};

glMaterialfv(GL_FRONT, GL_AMBIENT, mat0_ambient);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat0_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat0_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, mat0_shininess);
```

Example

```
GLfloat red[4] = {1.0, 0.0, 0.0, 1.0};   // RGBA object color (red)

glMaterialfv ( GL_FRONT_AND_BACK,  // you can assign different
    GL_AMBIENT_AND_DIFFUSE, red ); // colors to different vertices

glBegin ( GL_POLYGON );              // draw polygon
    glNormal3f ( ... ); glVertex3f ( ... );
    glNormal3f ( ... ); glVertex3f ( ... );
    glNormal3f ( ... ); glVertex3f ( ... );
glEnd ( );
```

glColorMaterial()

- If only one material property is to be changed, it is more efficient to use glColorMaterial( )
- glColorMaterial( ) causes material to track glColor*( )
- Ex)

```
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT, GL_DIFFUSE);
glColor3f(0.2, 0.5, 0.8); // changes the diffuse material color
   <Draw objects here>
glColorMaterial(GL_FRONT, GL_SPECULAR);
glColor3f(0.9, 0.0, 0.2); // changes the specular material color
   <Draw objects here>
glDisable(GL_COLOR_MATERIAL);
```