Applicazioni Ipermediali
(Web e Multimedia)

Principi di Grafica 3D
Thimoty Barbieri

Contents

- Graphics Systems and Models
  - Core definitions
  - Physical and Synthetic Images
  - The Synthetic-Camera Model
  - The Modeling-Rendering Paradigm
  - The Graphics Pipeline
    - Clipping
    - Projection
    - Rasterization
Contents (2)

- Graphics Programming
  - Introduction to the OpenGL API
  - Primitives and Attributes
  - Color
  - Viewing
  - Control Functions
  - User Interaction

Contents (3)

- Geometric Objects and Transformations
  - Brushing-up Scalars, Points and Vectors
  - Coordinate Systems and Frames
  - Modeling a Color Cube
  - Rotation, Translation, Scaling
  - Homogeneous Coordinates
  - OpenGL Transformations
Contents (4)

- Viewing
  - Classical and Computer Viewing
  - Positioning of the Camera
  - Simple Projections
  - Projections in OpenGL
  - Hidden-Surface Removal
  - Walking Through a Scene

Contents (5)

- Shading
  - Light and Matter, Light Sources
  - The Phong Reflection Model
  - Computation of Vectors
  - Polygonal Shading
  - Shading in OpenGL
  - Global Rendering: Ray Tracing and Radiosity
Contents (6)

- “Advanced” topics
- VRML
  - Examples and Tools
- Java3D
  - Examples and Tools

Contents (7)

- Networked Virtual Environments
  - Definitions and Examples
  - Introduction to the Java-EAI technique
  - The HOC WebTalk-II System
Bibliography

- Edward Angel - Interactive Computer Graphics, Addison Wesley
- Foley, Van Dam et Al. - Computer Graphics Principles & Practice, Addison Wesley
- Rohel - Late Night VRML with Java
- Petersen - Ready to Run Java 3D
- Woo - OpenGL Programmers’ Reference - Addison Wesley

Internet Bibliography

- Refer to hocboard.elet.polimi.it
- Or to the personal website listed under my page in www.elet.polimi.it
- All information on
  
  http://www.elet.polimi.it/people/barbieri
Fundamentals of 3D graphics

I. Introduction

Applications of Computer Graphics

- Computer Graphics is concerned with all aspects of producing pictures or images by means of a computer system
- Visual representation is used to model and solve problems: beginnings with plotting functions and simulations
- New horizons are opening up with Virtual Reality (VR)
- Graphical User Interfaces (GUIs) have been capital in advances in Computer Human Interfaces (CHI)
Components of a Graphics System

- Almost all graphics systems are raster based. A raster is an array of pixels. The pixels are stored in a region of memory called frame buffer (VRAM/DRAM).
- The depth of a frame buffer is the number of bits used for each pixel. (1-bit = 2 colors; 8-bits = 2^8 = 256 colors). Full color systems have at least true color depth (24 bits).
- The bits are partitioned between the Red, Green and Blue components to form the color.
- The resolution is the number of pixels in the frame buffer - and it determines the detail of the image.

Output Devices

- The process of converting information into pixels, in proper colors, is called rasterization.
- The dominant output device is still the CRT (Cathode Ray Tube).
- The refreshing rate must be at least 50 Hz - 85 Hz.
Output Devices (2)

- The screen can be non-interlaced or interlaced. An interlaced screen at 60 Hz redraws the scene 30 times/sec.
- The phosphors are arranged in triads of RGB - the shadow mask ensures that the electron beam excites only the correct color.

Objects and Viewers

- Computer generated images are artificial, but the process of generation can be compared to traditional physical imaging methods.
- The two basic entities in image formation are the object and the viewer. A set of space positions (vertices) define an object.
- The object cannot be formed until a viewer is defined. The image seen by the viewer is formed in a plane (retina, film). The image is related to the position of the viewer towards the object. This relationship defines a 2D image out of a 3D scene.
Objects and Viewers

It is not possible to see the scene without Light. The way light interacts with the object and reaches the viewer determines the image.

In CG, light is modeled using Geometric Optics (sources and rays), and considering separately the contributions of R, G, B.
Ray Tracing

- Given a single point source, the light rays bounce off the objects of the scene. Some are reflective, some absorb light (object properties). Some rays arrive at the viewer, others don’t.
- *Ray Tracing* is an image formation technique that use this light-ray model with appropriate simplifications.
- At first we will assume a single *monochrome* light with all objects uniformly bright.

The Pinhole Camera

We use the *pinhole camera* as a model to describe image formation. Some analogies with the human eye.

A *pinhole camera* is a box with a small hole, in which a single ray of light enters. The *film plane* is located at distance $d$. 

The Pinhole Camera

The image of point \((x, y, z)\) is on film plane \(z = -d\).

Moreover (similar triangles), \(y_p = \frac{-y}{z/d}\) and \(x_p = \frac{-x}{z/d}\)

The point \(x_p, y_p, -d\) is the projection of the original point on the film plane.

Field of view and Depth of field

The field or angle of view of the camera is the angle made by the largest object that the camera can image on the film plane.

The ideal care has an infinite depth of field, that is all points in the FOV are in focus.

This model is very limited:
- It admits only a single ray
- Its angle of view cannot be adjusted
The Synthetic-Camera Model

By replacing the pinhole with a lens, we can gather more light. The larger the aperture of the lens, the more light.

Picking a lens with a proper focal length (like choosing $d$ in the pinhole camera) we can select our desired FOV. With the lens, however, not all distances are in focus.

This optical system is the basis for the Synthetic Camera Model.

We use this model to describe visualisation with 3D API.

The S/C Model: definitions

The image of a point is found drawing a line, the projector, from the point itself to the center of the lens, the center of projection (COP). All projectors are rays emanating from the COP.

The film backplane is called the projection plane. The image is located where the projector of that point intersects the plane.
The Clipping Plane

The FOV expresses a limitation on the portion of image that can be imaged. In the S/C model, this is represented by putting a viewing window in front, called the clipping plane.

Clipping is one of the early processes in graphics processing.

The Main Elements

The main elements in describing a scene are thus:

- Objects
- Viewer
- Light sources
- Material properties

To specify a viewer (camera)

- Position
- Orientation
- Focal length
- Film Plane
Graphical APIs

An API provides means to specify all these elements

usually user interaction is platform-specific and left out of the graphics API. Api components are typically focused to primitive functions, attribute functions, viewing functions, transformation functions, input functions, control functions

OpenGL, PHIGS, GKS, Java3D are all graphical APIs

The Modeling-Rendering Paradigm

Modeling is the definition of the scene by description of all its main elements. It can be done via an API, or via modeling tools, which generate files containing the descriptions of the models (e.g. Max R3)

Rendering starts from the modeling descriptions to build as accurately as possible a view of the model, according to the given parameters.

Modeling and Rendering can be separated (as in Pixar’s RenderMan)
Pipeline Architectures for Rendering

Rendering is a computer-intensive process, which can be decomposed in stages. These stages are grouped in a processing pipeline for performance.

Transformations from the internal camera/object coordinate system into the screen coordinate system

Clipping to take into account the FOV limitations. To be done early.

Projection of 3D objects into 2D plane of visualisation

Rasterization converts geometric objects in array of pixels to be stored in the frame buffer. This is the end result.

Fundamentals of 3D Graphics

II. Graphics Programming with OpenGL
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.

Related APIs

- AGL, GLX, WGL
  - glue between OpenGL and windowing systems
- GLU (OpenGL Utility Library)
  - part of OpenGL
  - NURBS, tessellators, quadric shapes, etc.
- GLUT (OpenGL Utility Toolkit)
  - portable windowing API
  - not officially part of OpenGL
OpenGL and Related APIs

application program

- OpenGL Motif widget or similar
- GLX, AGL or WGL
- X, Win32, Mac O/S

software and/or hardware

Preliminaries

- Headers Files
  - #include <GL/gl.h>
  - #include <GL/glu.h>
  - #include <GL/glut.h>

- Libraries

- Enumerated Types
  - OpenGL defines numerous types for compatibility
    - GLfloat, GLint, GLenum, etc.
**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**

```c
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

```c
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

```c
    glutDisplayFunc( display );
    glutIdleFunc( idle );
    glutKeyboardFunc( keyboard );
```
**Rendering Callback**

- Do all of your drawing here

```c
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

```c
void idle( void )
{
    t += dt;
    glutPostRedisplay();
}
```
User Input Callbacks

- Process user input

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

OpenGL Geometric Primitives

- All geometric primitives are specified by vertices
Simple Example

```c
void drawRhombus( GLfloat color[] )
{
    glBegin( GL_QUADS );
    glColor3fv( color );
    glVertex2f( 0.0, 0.0 );
    glVertex2f( 1.0, 0.0 );
    glVertex2f( 1.5, 1.118 );
    glVertex2f( 0.5, 1.118 );
    glEnd();
}
```

OpenGL Command Formats

```c
GLfloat glVertex3fv( vv )
```

- **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 )

Principi di Grafica 3D

Thimoty Barbieri
Specifying Geometric Primitives

Primitives are specified using

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

- `primType` determines how vertices are combined

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

The RGB and CMYK color models

- Two of the main models are the subtractive CMYK (for classic printing) and the additive RGB (for visualisation)
- Each color model defines a different color gamut
Additive color matching

- The RGB additive model is linked to light emission. Color is obtained by adding light components \((\text{tristimulus})\)
- \(C = t_1 R + t_2 G + t_3 B\)

OpenGL Color Models

- RGBA or Color Index

  \(\text{color index mode}\)

  - Display
    - RGBA mode
      - Red Green Blue
      - 0 1 2 3
      - 16 8 4 2
      - ***
      - 24 123 219 74
      - 25 26
      - ***
The LookUp Table

- For systems with a limited depth in the frame buffer, a palette of a subset of the viewable color is used.
- The depth pixels are interpreted as indices rather than color values, which correspond to lines in a table (the lookup table).
- If k is the maximum bit depth and m is the depth subset for describing color, we have $2^m$ reds, $2^m$ greens and $2^m$ blues = $2^{3m}$ colors, but with a frame buffer for only $2^k$ colors.
- We can then select the colors needed and put them in a LUT of $2^k \times 3m$. (most common 256 color out of 16M for $k=m=8$).

Controlling Rendering Appearance

- From wireframe to texture mapping
OpenGL’s State Machine

- All rendering attributes are encapsulated in the OpenGL State
  - rendering styles
  - shading
  - lighting
  - texture mapping

Manipulating OpenGL State

- Appearance is controlled by current state
  for each (primitive to render) {
    update OpenGL state
    render primitive
  }

  Manipulating vertex attributes is most common way to manipulate state
  
  `glColor*()` / `glIndex*()`
  `glNormal*()`
  `glTexCoord*()`
Controlling current state

- Setting State
  ```c
  glPointSize( size );
glLineStipple( repeat, pattern );
glShadeModel( GL_SMOOTH );
  ```
- Enabling Features
  ```c
  glEnable( GL_LIGHTING );
glDisable( GL_TEXTURE_2D );
  ```

Hidden Surface Removal: the Z-Buffer

- From the viewers' viewpoint, some objects are visible, but some are occluded and others partially visible.
- Determining occluded surfaces is known as the HSR problem. The Z-Buffer Algorithm is a simple method for this purpose.
- Other algorithms reverse the problem: Visible Surface Determination.
Example 1: The Sierpinski Gasket (Zbuffer)

- The Sierpinski Gasket example shows how the Z-Buffer determination is enabled in OpenGL.
- It also uses several other primitives already covered.

```c
glInitDisplayMode(GLUT_DEPTH);
 glEnable(GL_DEPTH_TEST);
 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Example 2: The Rotating Square (Double Buffering)

- This example shows a simple user interaction and the use of *double buffering*
- The average CRT refresh rate ranges 50-85 Hz. The contents of the frame buffer should be redrawn at this rate. Changing the frame buffer during refreshing creates *artifacts*.
- Even if the object is simple, there is no coupling between completion of drawing in the FB and redisplay. This also creates *artifacts*.
- *Double buffering* provides a *front* and a *back* buffer, which are drawn and redisplayed in turns. One is updated when the other is displayed. Then the buffers are swapped. This is done in OpenGL:

```c
glInitDisplayMode(GLUT_DOUBLE);
 glutSwapBuffers();
```
Immediate and Retained Mode

- API provide two basic operating modes:

- IMMEDIATE MODE
  - as soon as the program executes a statement that defines a primitive, the primitive is sent to the processing server for display, and no memory of it is retained in the system. To redisplay the primitive, it must be redefined and reprocessed.

- RETAINED MODE
  - the object is defined once, then its description is put in a display list. This is stored in the server and can be redisplayed with a simple function call. Reduced data traffic, it takes advantage of special purpose processing capabilities of the processing server. Disadvantage: requires additional memory at framebuffer level, and Display List overhead during definition.

OpenGL Display Lists - Declaration

```c
#define BOX 1 /* or any available n */
gNewList(BOX, GL_COMPILE);

glBegin(GL_POLYGON);
setColor3f(1.0, 0.0, 0.0);
Vertex2f(-1.0, -1.0);
Vertex2f(1.0, -1.0);
Vertex2f(1.0, 1.0);
Vertex2f(-1.0, 1.0);
End();
EndList();
```

OpenGL Display Lists - Calling

```c
glCallList(BOX);
```

- PAY ATTENTION - when modifying the state of the OpenGL State Machine, provide fallback mechanisms to restore original state WITHIN each Display List (Self-Reentrant State), otherwise the result of following calls might be unpredictable!

---

Fundamentals of 3D Graphics

III. Geometric Transformations
Three Dimensional Primitives

- In 3D, there can be a lot of different shapes:

![Image of various 3D shapes]

- However, all objects have three characteristics that go well with graphics hardware and software:
  - Objects are described by their surfaces and can be considered hollow
  - Objects can be specified by a set of vertices in 3D
  - Objects can be approximated by a set of flat convex polygons

Tessellation

- Modern graphics systems are designed to render fast flat triangular polygons (approx. 5 millions per second).
- Most objects can be approximated to flat surfaces. The triangle is used because it can be always obtained using three vertices on the same plan. Quads can be non-planar.
- The process of dividing complex 3D shapes in groups of triangles is called tessellation.
- An exception is Constructive Solid Geometry, in which the system uses a small set of volumetric objects, and defines shapes by composition and difference of the basic set. Rendering CSG is more difficult than surface-based polygonal models.
Vectors

- Any vector can be represented by three linearly independent vectors, which form the basis, and three scalars, which are the components.

\[ v = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 \]

- Thus any vector, given a basis, can be represented by a column matrix:

\[
\begin{bmatrix}
\alpha_1 \\
\alpha_2 \\
\alpha_3
\end{bmatrix}
\]

Reference Point

- This representation still calls for a reference point $P_0$ which marks the origin of the basis. These two make the frame.

\[ P = P_0 + \eta_1 v_1 + \eta_2 v_2 + \eta_3 v_3 \]
Transformation Matrix

- It is possible to change from one basis to another with a transformation matrix:
  \[
  \begin{bmatrix}
  u_1 \\ u_2 \\ u_3 
  \end{bmatrix} = M 
  \begin{bmatrix}
  v_1 \\ v_2 \\ v_3 
  \end{bmatrix}
  \]

- However, such a transformation can represent with a single multiplication only rotation and scaling around the origin.

- Translation, which requires to address also the origin point, is not possible with this 3x3 transformation matrix.

\[
M = \begin{bmatrix}
\alpha_{11} & \alpha_{12} & \alpha_{13} \\
\alpha_{21} & \alpha_{22} & \alpha_{23} \\
\alpha_{31} & \alpha_{32} & \alpha_{33}
\end{bmatrix}
\]

The need for Homogeneous Coordinates

- Homogeneous coordinates allow to represent the entire frame and to process it within a single matrix multiplication.

- Let us consider a point \( P \) in a frame \( P_0, v_1, v_2, v_3 \):

\[
P = P_0 + \eta_1 v_1 + \eta_2 v_2 + \eta_3 v_3
\]

- If we define two new frame operations as:

\[
\begin{align*}
0 \cdot P &= 0 \\
1 \cdot P &= P
\end{align*}
\]

- The entire relation can be expressed formally as a single matrix product between the component and the frame itself:

\[
P = \begin{bmatrix}
\alpha_1 & \alpha_2 & \alpha_3 \\
v_1 \\
v_2 \\
v_3 \\
P_0
\end{bmatrix}
\]
The Homogeneous Matrix $M$

Because it operates on the complete frame, the $4 \times 4$ matrix allows to describe all line-preserving transformation. Even if we add one dimension, we have less arithmetic involved than using two separate $3 \times 3$ transformations.

$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ Q_0 \end{bmatrix} = M \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix}$

Frames in OpenGL

- OpenGL has two frames: the camera frame and the world fixed in regard to the other. In the model-view approach, the camera is fixed and the scene is transformed.
- Initially the two frames are overlapped, but to have the scene at a certain distance from the camera, a translation is necessary.
- For example along the z axis:

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -d \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
Modeling of a Cube: Sides of a Face

There are several ways to describe a cube, for example by listing the coordinates of the vertices of its sides. If we specify arrays of three elements as coordinates, OpenGL treats them internally as homogeneous coordinates.

The order of the points which make a side is important to determine if the face points inwards or outwards: this issue influences rendering. Right Hand Rule: counterclockwise order for outward faces.

```c
GLfloat vertices[4][3] = {{-1.0, 1.0, -1.0}, {1.0, -1.0, -1.0}, {1.0, 1.0, -1.0}, {-1.0, 1.0, -1.0}, {-1.0, -1.0, 1.0}, {1.0, -1.0, 1.0}, {1.0, 1.0, 1.0}, {-1.0, 1.0, 1.0}};
```

Modeling a Cube: Data Structures

An intelligent data structure for models attempts to describe not only geometry but topology, that is preserving the information that the cube has sides. This is accomplished by using the array of vertices as a vertex list, which maps vertices to faces:
**Modeling a Color Cube: the Code**

```c
GLfloat vertices[3][3] = {{1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0},
                           {-1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0},
                           {-1.0,-1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0}};

void polygon(int a, int b, int c, int d)
{
    glBegin(GL_POLYGON);
    glVertex3fv(vertices[a]);
    glVertex3fv(vertices[b]);
    glVertex3fv(vertices[c]);
    glVertex3fv(vertices[d]);
    glEnd();
}

void colorcube()
{
    polygon(0,3,2,1);
    polygon(2,3,7,6);
    polygon(0,4,7,3);
    polygon(1,2,6,5);
    polygon(4,5,6,7);
    polygon(0,1,5,4);
}
```

---

**Modeling a Cube: Color interpolation**

- Colors are assigned to each vertex. The color inside the shape is calculated by *bilinear interpolation*. The coloring in relation to shape, lights, etc. is a complex subject named *Shading*.

- Bilinear interpolation interpolates colors on the edges in relation to the colors in the vertices, and colors in the inside by drawing a connecting line between points on the edges.

- Another array for colors is needed, plus an additional call to glColor3fv(*GLfloat*).

  (see example cube.c)
Transformations: Translation

- Translation is a transformation which displaces points by a fixed distance in a given direction. It only needs a displacement vector $d$.

Equation: $P' = P + d$

Transformation Matrix:

$$
T = \begin{bmatrix}
1 & 0 & 0 & \alpha_x \\
0 & 1 & 0 & \alpha_y \\
0 & 0 & 1 & \alpha_z \\
0 & 0 & 0 & 1
\end{bmatrix}
$$

- $d = \begin{bmatrix} \alpha_x \\ \alpha_y \\ \alpha_z \\ 0 \end{bmatrix}$

Transformations: Rotation

- Rotation involves more parameters. If we consider rotation on a plane:

$$
\begin{align*}
    x &= \rho \cos \varphi \\
y &= \rho \sin \varphi \\
x' &= \rho \cos(\vartheta + \varphi) \\
y' &= \rho \sin(\vartheta + \varphi)
\end{align*}
$$

- Using the trigonometric identities:

$$
\begin{align*}
x' &= \rho \cos \varphi \cos \vartheta - \rho \sin \varphi \sin \vartheta = x \cos \vartheta - y \sin \vartheta \\
y' &= \rho \cos \varphi \sin \vartheta + \rho \sin \varphi \cos \vartheta = x \sin \vartheta + y \cos \vartheta
\end{align*}
$$

- This describes a rotation around the z axis
Transformations: Rotation Matrices

- We have thus three rotation matrices, one for each rotation plane. An arbitrary rotation can be obtained by the composition of these three.

\[
R_x(\theta) = \begin{bmatrix}
\cos\theta & -\sin\theta & 0 \\
\sin\theta & \cos\theta & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

\[
R_y(\theta) = \begin{bmatrix}
\cos\theta & 0 & \sin\theta \\
0 & 1 & 0 \\
-\sin\theta & 0 & \cos\theta
\end{bmatrix}
\]

\[
R_z(\theta) = \begin{bmatrix}
\cos\theta & -\sin\theta & 0 \\
\sin\theta & \cos\theta & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

- Two interesting properties:

\[
R^{-1}(\theta) = R(-\theta)
\]

\[
R^{-1}(\theta) = R^T(\theta)
\]

Transformations: Scaling

- Scaling is a non rigid body transformation which can make objects smaller or bigger uniformly or following a direction.
- Scaling is expressed with a factor. A factor >1 causes an enlargement, a 0-factor<1 causes a reduction, a negative factor mirrors the shape.
- It suffices to put these factors, one per axis in the transformation matrix:

\[
S_{x,y,z}(\beta_x, \beta_y, \beta_z) = \begin{bmatrix}
\beta_x & 0 & 0 \\
0 & \beta_y & 0 \\
0 & 0 & \beta_z
\end{bmatrix}
\]

- The Inverse Scale is obtained using the reciprocals of the factor (1/b)
Concatenation of Transformations

- Is it possible to carry more than one transformation at once, as in \( q = CBp \), but remember that the first transformation to be performed is \( A \), then \( B \), then \( C \): \( q = (C(B(Ap))) \)

- Usually the transformation processors in the pipeline first computes \( M=CB \), then feeds \( M \) to the transformator. In this way for each point there is a single matrix multiplication. This matrix in OpenGL is the Current Transformation Matrix (CTM).

Transformations in OpenGL

- In OpenGL the matrix that it is applied to all primitives is the product of the ModelView matrix \( GL\_MODELVIEW \) and the projection matrix \( GL\_PROJECTION \). Each portion can be manipulated separately by using \( glMatrixMode \).
  - \( glLoadIdentity \) loads the identity in the current mode of the CTM.
  - \( glRotatef(\text{angle}, \text{vx}, \text{vy}, \text{vz}) \), \( glTranslatef(\text{dx}, \text{dy}, \text{dz}) \) and \( glScalef(\text{sx}, \text{sy}, \text{sz}) \) put the correct matrixes in the CTM. \( V^* \) marks the rotation spin vector, \( D^* \) the displacement vector, \( S^* \) the scaling factors.
  - The transformation specified most recently is the one applied first!
Spinning a Cube

```c
static GLfloat theta[] = {0.0, 0.0, 0.0};
static GLint axis = 2;

void display(void)
{
  /* display callback, clear frame buffer and z buffer,
   rotate cube and draw, swap buffers */
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glLoadIdentity();
  glRotatef((theta[0], 1.0, 0.0, 0.0);
  glRotatef((theta[1], 0.0, 1.0, 0.0);
  glRotatef((theta[2], 0.0, 0.0, 1.0);
  colorcube();
  glutSwapBuffers();
}

void spinCube()
{
  /* Idle/* callback, spin cube 2
   degrees about selected axis */
  theta[axis] += 2.0;
  if (theta[axis] > 360.0 )
  theta[axis] -= 360.0;
  /* display(); */
  glutPostRedisplay();
}
```

Fundamentals of 3D Graphics

IV. Viewing
Classical Viewing

- The basic elements in classical and computer-based viewing are the same, as in the synthetic-camera model: objects, viewers, projectors, projecting plane and the COP, which corresponds to the origin of the camera frame.
- The COP can be moved to infinity (the viewer is at an infinite distance from the scene). In this case the COP is replaced with a Direction of Projection (DOP). The size of the scene remains the same once infinity is reached.
- Views with a finite COP are named Perspective Views
- Views with a COP to infinity (DOP) are Parallel Views
- APIs can produce both: Planar Geometric Projections

Parallel and Perspective Views
Ortographic Projections

- In all views named *orthographic* or *orthogonal*, the projectors are perpendicular to the projection plane, which is parallel to one *principal face*. Very often it is necessary to show more of one of these projections to let the viewer understand the features of an object, usually one projection for each *principal face*.

Axonometric Projections

- To see more than one principal face at a time, one can use *axonometric views*.
- The projectors are *still* orthogonal to the projection plane, but the plane can have any orientation in relation to the object.
- If the plane is placed symmetrically on respect to the three principal faces (at the corner): *isometric view*
- If it is symmetrical in respect to two faces: *dimetric view*
- The most general case (any orientation): *trimetric view*
- This kind of views produce distortions (a line segment’s length in the image space is shorter than in the “real” object space)
- The lengths are still measurable provided that we know the fixed ratios between the two spaces (*trimetric* has three of such ratios)
**Examples of Axonometric Projections**

- **Oblique Projections**

  - The most general parallel views are the *Oblique* projections: the projectors are still parallel, but can form an arbitrary angle with the projection plane.
General Considerations on Parallel Prj.

- Creating oblique views is difficult and unnatural. The most widespread imaging devices (human eye, traditional camera) produce perspective views or parallel views with the scene very far from the viewer. This parallel view is always orthogonal because the lens is parallel to the projection plane. The only way to produce an oblique view is by twisting a bellows camera. (!)

- In general, Graphics APIs do not make a fine distinction between the different parallel views, but just between Parallel and Perspective. It is up to the programmer to tune the parameters in order to approximate the kind of parallel view she wants.

Perspective Viewing

- All perspective views exhibit a diminution of the size of the object. If the viewer moves farther, the object gets smaller. It is not possible to take measurements from a perspective view.

- In classical graphics, the viewer is opposed symmetrically to the object in respect to the projection plane.

- Thus the pyramid formed by the COP and the projectors is a right pyramid.
- This models canonical imaging devices such as cameras and the eye (lens and backplane parallel).
- CG includes the general case of non-parallel plane (bellows camera).
Perspective Views

- The views are usually known as one, two and three point perspectives.

![Perspective Views](image)

- The difference is on how many of the three principal directions are parallel to the projection plane. As soon as one of the principal directions is parallel, we need one vanishing point less. There is always at least one vanishing point and two parallel directions.

Positioning of the Camera

- Different APIs have different way to specify the camera position. In general, the first step is positioning the camera, and the second is to specify the type of projection.

- OpenGL uses two concatenated matrix, the ModelView and the Projection, that are applied as one to the scene. One of the uses of the ModelView matrix is to position objects. The other use, conversely, is to position the camera.

- OpenGL places the camera at the origin of the scene frame. If the ModelView matrix is an identity, the camera frame and the world frame are identical.
Imagine to place a point $P$ at the initial state (ViewMatrix = I). We then apply a transformation $C$ to the ViewMatrix.

The two frames are no longer the same, and the matrix contains the information to move from the scene to the camera frame.

If a new point $Q$ is placed after the transformation, the point is at $Q$ in the scene frame, and at $CQ$ in the camera frame. Thinking with a global coordinate system (the camera’s) or with a local one (the scene’s after the translation) leads to the same results.

ModelView transformations are strictly related, whether the programmer is thinking in term of moving the object (Modeling Transformation) or the camera (Viewing Transformation).

A result of this fact is that a modeling transformation can be represented as a viewing transformation in the opposite direction (the model space is a RIGHT HANDED coordinate system, the camera space is a LEFT HANDED coordinate system, but the transformation itself is identical).

Pay attention to the order of transformations - usually since the matrix are applied in “reverse” order - the viewing transformations go first in the code and the model transformations follow.
Generalities on Viewing APIs

- To position the Camera, in general one needs a View Reference Point (VRP) which is the point of view.
- It is then to be defined a View Plane Normal (VPN) to identify the projection plane (the back of the camera) and the ViewUp Vector (VUP) to determine which side is up.
- \( V \) is obtained projecting the VUP on the plane, and \( u \) with the cross product between \( n \) and \( v \).
- The \( u-v-n \) triplet forms the base for the viewing coordinate system.

Specifying the POV in OpenGL

- OpenGL GLU provide an intuitive way to set up the camera coordinate system, the LookAt function.
- The \( e \) coordinates specify the camera position, the \( at \) coordinate the point towards which the camera points, the \( up \) coordinates are the components of the VUP.
- See Nate’s Tutorials.
- The VPN is obtained by point subtracting the eye coordinates and the at coordinates.

\[
VPN = e - a
\]
Viewing Volume

- Once the camera is positioned, there is still to set the FOV (e.g. changing the lens of the camera) to determine how much of the scene is viewable.
- We also have to determine in which way is the scene to be mapped on our viewport (clipped area): as Parallel or Perspective. Objects outside the viewing volume are clipped.
- These two determine an area in space in which the objects can be visualised by the imaging device considered: the viewing volume.

Viewing Frustum

- Generally the viewing volume is a semi-infinite pyramid whose apex is the COP. In Perspective Projection the viewing volume is called a Frustum.
- All Projection Operations in OpenGL must be done on the GL_PROJECTION matrix! (The second half of the CTM!)
### Perspective Projection

- Perspective Projection in OpenGL is obtained by defining the View Frustum. There are two ways:

  - `glFrustum(xmin, xmax, ymin, ymax, near, far)`
  - `gluPerspective(fovy, aspect, near, far)`
    - `aspect = w/h`

### Parallel Projection

- The only parallel projection provided by OpenGL is the Orthogonal Projection. Here the parameters are similar to the frustum, but the projectors are parallel.

  - `glOrtho(xmin, xmax, ymin, ymax, near, far)`
    - `z = z_min = near`; `z = z_max = far`; `far > near`
Example on Viewing and Modeling Transformations

```cpp
void init(void)
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glShadeModel (GL_FLAT);
}

void display(void)
{
    glClearColor (GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0); /* clear the matrix */
    glLoadIdentity (); /* viewing transformation */
    gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
    glScalef (1.0, 2.0, 1.0); /* modeling transformation */
    glutWireCube (1.0);
    glFlush ();
}

void reshape (int w, int h)
{
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    glFrustum (-1.0, 1.0, -1.0, 1.0, 1.5, 20.0);
    glMatrixMode (GL_MODELVIEW);
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize (500, 500);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    init ();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}
```
Walking Through a Scene

- We modify the rotating cube program to allow to move the camera frame via keyboard. However, the viewer will always look at the center of the cube. It will suffice to put parameters in the `LookAt` viewing function in the `display` callback.

```c
void display(void)
{
    glLoadIdentity();
    gluLookAt(viewer[0],viewer[1],viewer[2],
              0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
}

void keys(unsigned char key, int x, int y)
{
    if(key == 'x') viewer[0]--;    
    if(key == 'X') viewer[0]++;    
}
```

Fundamentals of 3D Graphics

V. Shading
Introduction to Shading

- Until now the models looked flat: we have not considered the interactions between objects and light. The different colors the objects receive by the different ways they are lit are calculated with a process called Shading.
- There are two main models: local lighting models and global lighting models.
- The Local models compute shades to assign to a point on a surface, independently from any other surface in the scene.
- The Global models take into account also all the light components incoming from the environment and the other objects, and the interactions between them.

Light and Matter

- A surface can emit light by self-emission, or reflect light from other surfaces, or do both. The color on an object is determined by the sum of all these interactions.
- These interactions can be tracked in a recursive process.
  - The light from A shines on B and then back to A, and viceversa.
  - This recursion is represented in an integral equation named Rendering Equation.
  - This equation cannot be solved but by approximation
**Rendering Methods**

- RayTracing and Radiosity are global methods which try to approximate the Rendering Equation. However, their speed is not comparable to the normal polygon-pipeline operation, and the processing has to be done off-pipeline.
- Rather than considering the global effects, other methods consider rays from light sources to surfaces with only a single interaction.

- The clipping window is mapped to the screen, and the viewer is replaced by the COP. Only the rays from the source to the COP are to be considered.

**Light-Material Interactions**

- Interactions between light and materials can be classified as:

  - **A) Specular Surfaces**: most of the light is reflected and scattered in a narrow angle (mirrors)
  - **B) Diffuse Surfaces**: reflected light is scattered in all directions. Perfectly Diffuse Surfaces scatter light equally in all directions.
  - **C) Translucent Surfaces**: some light penetrates the object and emerges from another location, in a process of *refraction*.
Light Sources

Light can leave a surface through two fundamental processes: self-emission and reflection. In most cases the two contributions are always present simultaneously.

A source can be considered as an object with a surface:

- Each point \( x, y, z \) on the surface can emit light in a direction and with a certain wavelength.
- A light source can thus be characterized by its illumination function

\[
I(x, y, z, \theta, \phi, \lambda)
\]

Classification of Light Sources

- The general contribution of a complex light source is thus the integral of the illumination function over the surface of the source, assuming that each wavelength is emitted independently.
- To avoid this complex computation, complex light sources can be modeled with polygons, each of which is a simple-emitting light source.
- There are mainly four basic types of light sources:
  - ambient lighting
  - point sources
  - spotlights
  - distant light
Color Sources

- Light sources not only can emit at different frequencies, but the direction of emission can be different in relation to the frequency.
- The simplified model for color emission is based on the tristimulus theory, and describe sources with a luminance function.

\[
I = \begin{bmatrix}
I_r \\
I_g \\
I_b
\end{bmatrix}
\]

- The three components of the luminance function can be treated separately, and we will do so in the following.
- Every equation in the following should be applied for each component of the luminance function.

Ambient Light

- Very diffused lights for general illumination of an environment are achieved with large sources which scatter light in all directions.
- The effect is to provide an ambient lighting: we can postulate an ambient intensity at each point of the target environment.
- An ambient illumination is characterized by an intensity \( I_a \) which is uniform in every point of the scene.

\[
I_a = \begin{bmatrix}
I_{ar} \\
I_{ag} \\
I_{ab}
\end{bmatrix}
\]
Point Sources

- An **ideal point source** emits light equally in all directions.
- The intensity of illumination received from such a source depends on the distance to the surface:

\[ I(p, p_0) = \frac{1}{|p - p_0|^2} I(p_0) \]

Characteristics of Point Sources

- Point sources are used more because of their ease of use than because of their resemblance to reality.
- Scenes with only pointsource lights have high contrast, an object appears either light or dark. The inverse quadratic also has an effect on harshness. Often a term \( a+b+d+cd^2 \) is used to soften the lights on the scene.

- In real scenes, appearances are softer because light sources have an extension, and this causes the creation of **umbra** and **penumbra** in the scene.
SpotLights

- Spotlights are characterized by a narrow range of angles through which light is emitted. (A point source with a limited emission angle).

- Real spotlights do not emit uniformly within the cone, but follow an intensity distribution function. Light is usually most concentrated in the center of the cone.

- The distribution within the cone is usually modeled with $d = \cos^e \varphi$

Distant Light Sources

- With a point source, as we move along the surface, the light emission direction has to be recomputed. However, if the source is distant enough, the directions can be considered parallel.

- For describing Distant Sources, it is then needed not a Source Point but a Source Direction. Computing directions instead of points is much easier.

$$p_s = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

From point to direction

$$p_s = \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix}$$
The Phong Reflection Model

- Instead of trying to compute illumination with a physical-based model, Phong invented an approach to get similar results with an eye to reduced computation efforts. While thus the process does not model the physical reality, it is a good approximation and easy to calculate.

- Phong uses 4 vectors to determine the color of any point of an illuminated surface.
- \( n = \) normal at \( p \)
- \( v = \) direction from \( p \) to the COP
- \( l = \) direction from \( p \) to the Point Source
- \( r = \) direction of a perfectly reflected ray from \( l \)

Light-Material Interaction in Phong

- Given a set of point sources, each source can have a separate Ambient, Diffuse and Specular component for each color component. This seems unnatural because it is applied to the source and not to the nature of the material, but allows to simulate locally effects that are in nature global.
- Any given point source is thus represented:

\[
L_i = \begin{bmatrix}
L_{ira} & L_{iga} & L_{iga} \\
L_{ird} & L_{igd} & L_{ibd} \\
L_{irs} & L_{igs} & L_{ibs}
\end{bmatrix}
\]

- It is also possible to compute the reflected light from the surface with a similar matrix \( R \)
### Intensity Balances in Phong

- Intensity of a color component (e.g. Red):

\[
I_{ir} = R_{ira}L_{ira} + R_{ird}L_{ird} + R_{irs}L_{irs} = I_{ira} + I_{ird} + I_{irs}
\]

- Total Intensity:

\[
I_r = \sum_i (I_{ira} + I_{ird} + I_{irs}) + I_{art}
\]

Red component of Ambient Light

### Ambient Reflection

- The intensity of ambient light La is the same at every point of the surface. Some of it is absorbed, and some reflected by the material, according to Ka, the ambient reflection coefficient.
- Because reflected light is positive, it is 0 <= Ka <= 1
- \( I_a = KaL_a \)
- There is a reflection coefficient for every color component of ambient light, so for example a material with low Kab and high Kar and Kag appears yellow with a white ambient light, because the blue is retained, but red and green are re-emitted, thus giving the appearance of yellow.
Diffuse Reflection

- Diffuse Reflection scatters the reflected light in all directions. This happens because of the rough surface of the material. The nature of diffusion depends on the direction of the light, but in general uniformly rough surfaces scatter equally in all directions.

- These surfaces can be modeled with Lambert’s law. This law states that on rough surfaces we only see the vertical component of the light, so as the light source is lowered, the quantity of light is scattered on a bigger surface and its appearance get dimmer.

Lambert’s Law

- Given $\mathbf{n}$ the normal to the point and $\mathbf{l}$ the direction of the source, and $K_d$ the diffusion coefficient:

$$R_d \propto \cos u$$

$$\cos u = \mathbf{l} \cdot \mathbf{n}$$

$$I_d = k_d (\mathbf{l} \cdot \mathbf{n}) L_d$$

attenuation

$$I_d = \frac{k_d}{a + bd + cd^2} (\mathbf{l} \cdot \mathbf{n}) L_d$$
Specular Reflection

- Specular Reflection is responsible for the formation of highlights of different colors on 3D surfaces. This happens on smooth surfaces, where light is reflected in a single direction (or concentrated in a very narrow cone).
- In the Phong model, the surface is considered rough to compute the diffuse term, and smooth to compute the specular term. The amount of light in the specular term depends on the angle between $r$ (direction of perfect reflection) and $v$ direction of viewer.

\[ I_s = k_s L_s \cos^\alpha \phi \]

\[ I_s = k_s L_s (r \cdot v)^\alpha \]

- The Alpha coefficient is the shininess of the material. To the infinite it is a mirror, in range 100-500 metallic, under 100 some highlights.

The Complete Phong Model

- By adding all the terms with the distance term, the complete Phong Model which describes intensities with a Local Illumination model is:

\[
I = \frac{1}{a + bd + cd^2} [k_d (I \cdot n) L_d + k_s L_s (r \cdot v)^\alpha] + k_a L_a
\]

- This has to be computed three times, for each color component.
- As we cannot solve the full rendering equation, a good compromise is to use the Phong Model with an intelligent setup of its parameters to create a good realistic effect, approximating a global effect only with local calculations.
Normals

- A key component to determine shading of a surface is the **Normal** to the surface at a given point. More specifically, **OpenGL** requires to specify normals to the **vertices** that compose the scene.
- Because the calculations take place in a pipeline, in which generally only vertices are passed, not always the system has the geometric information to compute the normals automatically.
- Most renderers, OpenGL included, do not provide with tools to compute normals, task which is usually left to the programmer.
- To associate normal to vertices, OpenGL uses `glNormal3f(nx, ny, nz)` or `glNormal3fv(*normal)`.
- The normal is **modal**: once it is declared, it is used for all the subsequent vertices until a new one is set.

Computing Normals to a Plane

- A normal to a plane can be computed given its equation:
  \[
  ax + by + cz + d = 0
  \]
- By the condition of orthogonality we can write a dot product:
  \[
  \mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0) = 0
  \]
- And by comparing the two forms:
  \[
  \mathbf{n} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}
  \]
- Alternatively, if three complanar points are given, the normal can be computed as:
  \[
  \mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_0) \times (\mathbf{p}_1 - \mathbf{p}_0) 
  \]
Computing Normals to a Parametric Surface

A surface can be represented in a parametric form. For the simple case of a sphere:

\[
\begin{align*}
x(u, v) &= \cos u \sin v \\
y(u, v) &= \cos u \cos v \\
z(u, v) &= \sin u \\
-\frac{\pi}{2} &< u < \frac{\pi}{2}, -\pi < v < \pi
\end{align*}
\]

The normal to the tangent plane in P is computed from the cross product of the two gradients, which are in the tangent plane:

\[
n = \frac{\partial p}{\partial u} \times \frac{\partial p}{\partial v} = p
\]

---

Polygonal Shading

To recap, to start the shading computation we need:

- A set of light sources with their characteristics
- To select the lighting and illumination models
- A set of vertices and their normals

Even with simple equations, shading computation for each point of the surface can be very hard. By substituting shapes with polygonal approximations, we can significantly reduce the work for shading.

OpenGL exploits this by rendering curve surfaces into many small flat polygons (Polygonal Meshes). There are three basic methods for polygonal shading: flat shading, Gouraud Shading, and Phong Shading.
Flat Shading

- The starting point is still the $I$, $v$, and $n$ triplet. For a flat surface, $n$ is constant. Considering a distant viewer, $v$ is constant too. Considering a distant light source, rays hit the surface in parallel, so $I$ is constant over the surface as well.
- With all the three vectors constant, the three vectors are computed only once per surface, and the whole surface will get the same shade: this is called flat shading.

- This mode is activated in OpenGL with `glShadeModel(GL_FLAT)`. The normal to the first vertex of the polygon is taken into account. In a triangle strip the third is used for the first polygon, the fourth for the second, and so on.

Mach Bands
Mach Bands

- In Flat Shading there is a marked shade difference at the edge of one surface with the other. Due to lateral inhibition, the human eye overshoots one side of the intensity and undershoots the other, giving the phenomenon known as Mach Bands.

- There is nothing we can do to prevent this, except devising smoother shading techniques.

Interpolative and Gouraud Shading

- With glShadeModel(GL_SMOOTH), OpenGL interpolates colors assigned to vertices across the polygons.
- It also performs lighting calculation computing I and v for each vertex. For each vertex, the programmer can also specify a different normal.
- Since a normal has a discontinuity in the point where vertices meet in a mesh, Gouraud defines the vertex normal as:

\[
\mathbf{n} = \frac{\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4}{||\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4||}
\]
Phong Shading

- Even smoother shading can be obtained with Phong Shading. Instead of just working on vertices, Phong proposed to interpolate normals across each side of the polygon in a mesh.

- First we interpolate the normals at the sides, starting from the normals at the vertices like this:

\[ n(\alpha) = (1 - \alpha) n_A + (\alpha) n_B \]

- Then we interpolate all the interior points starting from the normals obtained on the edges:

\[ n(\alpha, \beta) = (1 - \beta) n_C + (\beta) n_D \]

OpenGL Examples

- We propose a shading example of a sphere approximated by recursive polygonal subdivision. This will allow to experiment with flat and smooth shading.

- We have the following problems:

  - How do we set lights?
  - Which model of illumination?
  - How do we specify materials?
  - Which shading do we use?
  - How do we compute normals to feed the shader?
Light Sources in OpenGL

- OpenGL supports the 4 basic light types, for a max of 8 sources in the scene (GL_LIGHT0 … 7).
- The structure is: `glLightfv(source, parameter, *value)`

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_POSITION</td>
<td>4 h. coordinates for LS position XYZ1</td>
</tr>
<tr>
<td>GL_AMBIENT</td>
<td>Ambient color component RGBA</td>
</tr>
<tr>
<td>GL_DIFFUSE</td>
<td>Diffuse component RGBA</td>
</tr>
<tr>
<td>GL_SPECULAR</td>
<td>Specular component RGBA</td>
</tr>
<tr>
<td>GL_CONSTANT_ATTENUATION</td>
<td>Attenuation factor in relation to distance</td>
</tr>
<tr>
<td>GL_SPOT_DIRECTION</td>
<td>XYZ0 Direction of Spot</td>
</tr>
<tr>
<td>GL_SPOT_EXPONENT</td>
<td>Attenuation exponent</td>
</tr>
<tr>
<td>GL_SPOT_CUTOFF</td>
<td>Spotlight cone angle</td>
</tr>
<tr>
<td>GL_LIGHT_MODEL_LOCAL_VIEWER</td>
<td>GL_TRUE/GL_FALSE Shade only what's in view</td>
</tr>
<tr>
<td>GL_LIGHT_MODEL_TWO_SIDE</td>
<td>GL_TRUE/GL_FALSE Shade front and back side</td>
</tr>
</tbody>
</table>

Materials in OpenGL

- OpenGL supports material description with the general light sources theory
- The structure is: `glMaterialfv(face, type, *value)`

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_FRONT_AND_BACK</td>
<td>Determines material spec. on both sides</td>
</tr>
<tr>
<td>GL_AMBIENT</td>
<td>Ambient reflection coefficient Ka RGBA</td>
</tr>
<tr>
<td>GL_DIFFUSE</td>
<td>Diffusve reflection coefficient Kd RGBA</td>
</tr>
<tr>
<td>GL_SPECULAR</td>
<td>Specular reflection component Ks RGBA</td>
</tr>
<tr>
<td>GL_SHININESS</td>
<td>Attenuation factor in relation to distance</td>
</tr>
<tr>
<td>GL_EMISSION</td>
<td>Self-emission component RGBA</td>
</tr>
</tbody>
</table>
Implementing Flat Shading

```c
void normal(point p)
{
    double sqrt();
    float d = 0.0;
    int i;
    for (i = 0; i < 3; i++) { d += p[i]*p[i]; }
    d = sqrt(d);
    if (d > 0.0) for (i = 0; i < 3; i++) { p[i] /= d; }
}
```

```c
cross(point a, point b, point c, point d)
{
    d[0] = (b[1]-a[1])*(c[2]-a[2]) - (b[2]-a[2])*(c[1]-a[1]);
    d[1] = (b[2]-a[2])*(c[0]-a[0]) - (b[0]-a[0])*(c[2]-a[2]);
    d[2] = (b[0]-a[0])*(c[1]-a[1]) - (b[1]-a[1])*(c[0]-a[0]);
    normal(d);
}
```

Using True Normals

```c
void triangle(point a, point b, point c)
/* display one triangle using a line loop for wire frame, a single normal for constant shading, three normals for interpolative shading */
{
    if (mode == 0) glBegin(GL_LINE_LOOP);
    else glBegin(GL_POLYGON);
    if (mode == 1) glNormal3fv(a);
    if (mode == 2) glNormal3fv(a);
    glVertex3fv(a);
    if (mode == 2) glNormal3fv(b);
    glVertex3fv(b);
    if (mode == 2) glNormal3fv(c);
    glVertex3fv(c);
    glEnd();
}
```
Global Rendering

- Up to now we have used a Local Lighting Model. All the computations were limited per object, and were independent from other objects on the scene.

- This technique is good for rapid rendering but cannot handle properly several situations that occur from the general composition of the scene (at global level):
  - light occlusion from other objects
  - specular objects which shine light on other objects
  - shadows

- If these effects are important, a Global Lighting Model is to be used. These are slower and more sophisticated techniques: Ray Tracing and Radiosity. The first works well for highly specular surfaces, the second is suited best for scenes with diffuse surfaces.

Global and Local Lighting

- Effect of a Global Lighting Model
- Effect of a Local Lighting Model
Ray Tracing

- Ray Tracing starts from the same assumptions of the local lighting model: light is formed by rays which arrive to the viewer.
- The direction of rays is followed backwards, from the viewer to the source, because it has no sense to trace rays which do not hit the viewer. So the rays are casted from the COP and followed throughout the scene. (Ray Casting) Each cast ray traverses the pixel grid and hits objects until it arrives at a light source.
- Instead of stopping at the first intersection as in the Phong Model, we compute a feeler or shadow ray to check if the point of intersection between object and cast ray is illuminated.
- If the feeler ray intersects a surface before joining the point, that point is in shadow.

Shadow Ray and Cast Ray

- For highly reflective surfaces, the shadow ray is followed as it bounces from surface to surface, until it goes off or reaches a source.
- These calculations are done recursively and take into account absorption.
- For reflective and transmitting surfaces, three tasks:
  - compute the contribution of the light source at the intersection point between surface and cast ray
  - cast a ray in the perfect reflection direction
  - cast a ray in the direction of the transmitted ray
- These three rays are then traced exactly like the original cast ray
Ray Tree and Recursive Ray Tracing

- The main activity of the Ray Tracer is thus to cast rays and follow them recursively within a data structure called a Ray Tree.

- To do so, there are lots of intersection computations. This is difficult with complex polygons, so most are limited to flat surfaces or quadrics.

- This also explains why ray tracers are suited to reflective environments and not to diffuse: in diffuse environments there are too many rays scattered in all directions, and the recursion (the treespan) becomes too deep.

Radiosity

- For diffuse environments Radiosity is a best suited method.

  - Radiosity is a numerical method to describe the global energy balance within lighting in the scene. If the scene is lit by distant light, we can consider the surfaces as uniformly colored, but each interacts with the other by diffusion.

  - The basic method breaks up the scene in patches, each of which is perfectly diffuse and of constant shade. In the first step the patches are considered pairwise to compute form factors that describe how much the light energy of one patch interacts with the other. We then have a set of linear equations that describe mutual reflectivity of the facets.

  - This method is independent from the viewer’s position.
Radiosity Principles

- Radiosity is an approach based on thermal engineering models for the emission and reflection of radiations. This provides a more accurate representation of interobject reflection without the need of a global illumination term.
- All energy emitted or reflected by every surface is accounted for by its reflection from or absorption by other surfaces. The rate at which energy leaves a surface, called its RADIOSITY, is the sum of the rates at which the surface emits energy and reflects or transmits it from that surface to other surfaces.
- Thus, radiosity methods determine surface interactions independently from the point of view.

Radiosity Equation

- In Ray based models, light sources are separate from the model surfaces. In Radiosity, every surface is an emitter, so also sources are accounted in the equation. The whole environment is broken up in \( n \) discrete patches each of which of finite size, emitting and reflecting like uniformly over its entire area. We can then describe the whole patch system as

\[
B_i = E_i + \rho_i \sum_{1 \leq j \leq n} B_j F_{j-1} \frac{A_j}{A_i}
\]
Radiosity Equation

\[ B_i = E_i + \rho_i \sum_{1 \leq j \leq n} B_j F_{j-1} \frac{A_j}{A_i} \]

- \( B_i \) and \( B_j \) are the radiosity patches \( i \) and \( j \) (e.g. W/m²).
- \( E_i \) is the rate of light emission from patch \( i \) (W/m², dep. On wavelength).
- \( \rho_i \) is the patch \( i \) reflectivity (dimensionless, depends on wavelength).
- \( F_{i-j} \) is the form factor which describes the fraction of energy leaving the entirety of patch \( j \) that arrives to the entirety of path \( i \), taking into account space and orientation.
- \( A_i \) and \( A_j \) are the areas of the two patches.

Equation Solution

- This equation leads to a matrix system, that has to be solved for every considerable range of wavelengths, since \( E \) and \( \rho \) are dependent on the wavelength.
- The form factors are dependent only on the geometry of the problem.
- The diagonal of the matrix is NOT made of 1, because some patches can even be reflecting on themselves (curved surfaces).
- Instead of computing radiosity on patches, it is also possible to compute it on the vertices of the mesh, and doing radiosity bilinear interpolation on each surface.
Radiosity Form Factors

- The general representation of the form factor geometry is the following:

\[
dF_{ij} = \frac{\cos \theta_i \cos \theta_j}{\pi r^2} H_{ij} dA_j
\]

- \(dA\) is the infinitesimal patch
- \(r\) is the ray length
- \(\theta\) are the angles between \(r\) and the normal to \(A\)
- \(H=1\) if \(A_j\) is visible from \(A_i\), 0 otherwise