COSC 5450 - Computer Graphics Lecture Note 1

Libao Jin (ljin1@uwyo.edu)

November 24, 2018

1 Introduction

Definition 1.1 (Computer graphics)**.** Any use of computers to *create* and *manipulate images* is described by *computer graphics*. Algorithmic and mathematical tools that can be used to create all kinds of images:

- realistic visual effects
- informative technical illustrations
- beautiful computer animations

Graphics can be two- or three-dimensional; images can be completely synthetic or can be produced manipulating photographs.

1.1 Graphics Areas

Major areas of computer graphics:

- *Modelling* deals with the mathematical specification of shape and appearance properties in a way that an be stored on the computer.
- *Rendering* is a term inherited from art and deals with the creation of shaded images from 3D computer tools.
- *Animation* is a technique to create an illusion of motion through sequences of images. Animation uses modeling and rendering but adds the key issue of movement over time, which is not usually dealt with in basic modeling and rendering.

Other areas that involve computer graphics:

- *User interaction* deals with the interface between input devices such as mice and tablets, the application, feedback to the user in imagery, and other sensory feedback.
- *Virtual reality* attempts to *immerse* the user into a 3/d virtual world.
- *Visualization* attempts to give users insight into complex information via visual display.
- *Image processing* deal with the manipulation of 2D images and is used in both the fields of graphics and vision.
- *3D scanning* uses range-finding technology to create measured 3D models.
- *Computational photography* is the use of computer graphics, computer vision, and image processing methods to enable new ways of photographically capturing objects, scenes, and environments.

1.2 Major Applications

- *Video games* increasingly use sophisticated 3D models and rendering algorithms.
- *Cartoons* are often rendered directly from 3D models.
- *Visual effects* use almost all types of computer graphics technology. Almost every modern film uses digital compositing to superimpose backgrounds with separately filmed foregrounds.
- *Animated films* use many of the same techniques that are used for visual effects, but without necessarily aiming for images that look real.

- *CAD/CAM* stands for *computer-aided design* and *computer-aided manufacturing*.
- *Simulation* can be thought of as accurate video gaming.
- *Medical imaging* creates meaningful images of scanned patient data.
- *Information visualization* creates images of data that do not necessarily have a *natural* depiction.

1.3 Graphics APIs

An *application program interface* (API) is a standard collection of functions to perform a set of related operations, and a graphics API is a set of functions that perform basic operations such as drawing images and 3D surfaces into windows on the screen.
Two dominant paradigms for graphics and user-interface APIs:

- Integrated approach, exemplified by Java, where the graphics and user-interface toolkits are intergrated and portable packages that are fully standardized and supported as part of the language.
- Direct3D and OpenGL, where the drawing commands are part of a software library tied to a language such as C++, and the user-interface software is an independent entity that might vary from system to system.

1.4 Graphics Pipeline

3D graphics pipeline is a special software/hardware subsystem that efficiently draws 3D primitives in perspective. Usually these systems are optimized for processing 3D triangles with shared vertices. The basic operations in the pipeline map the 3D vertex locations to 2D screen positions and shade the triangles so that they both look realistic and appear in proper back-to-front order.
Drawing the triangles in valid back-to-front order has been solved using the *z-buffer*, which uses a special memory buffer to solve the problem in a brute-force manner.

1.5 Numerical Issues

Many graphics programs are really just 3D numerical codes. Three "special" values for real numbers in IEEE floating-point:

1. Infinity (∞). This is a valid number that is larger than all other valid numbers.
2. Minus infinity ($-\infty$). This is a valid number that is smaller than all other valid numbers.
3. Not a number (NaN). This is an invalid number that arises from an operation with undefined consequences, such as zero divided by zero.

1.6 Efficiency

Efficiency is achieved through careful tradeoffs, and these tradeoffs are different for different architectures.
Steps to make code fast:

1. Write the code in the most straightforward way possible. Compute intermediate results as needed on the fly rather than storing them.
2. Compile in optimized mode.
3. Use whatever profiling tools exist to find critical bottlenecks.
4. Examine data structures to look for ways to improve locality. If possible, make data unit sizes match the cache/page size on the target architecture.
5. If profiling reveals bottlenecks in numeric computations, examine the assembly code generated by the compiler for missed efficiencies. Rewrite source code to solve any problems you find.

1.7 Designing and Coding Graphics Programs

1.7.1 Class Design

A key part of any graphics program is to have good classes or routines for geometric entities such as vectors and matrices, as wells as graphics entities such as RGB colors and images. These routines should be made as clean and efficient as possible.
Some basic classes to be written include:

- *vector2*. A 2D vector class that stores an x- and y-component. It should store these components in a length-2 array so that an indexing operator can be well supported. You should also include operations for vector addition, vector subtraction, dot product, cross product, scalar multiplication, and scalar division.
- *vector3*. A 3D vector class analogous to vector2.
- *hvector*. A homogeneous vector with four components.
- *rgb*. An RGB color that stores three components. You should also include operations for RGB addition, RGB subtraction, RGB multiplication, scalar multiplication, and scalar division.
- *transform*. A 4×4 matrix for transformations. You should include a matrix multiply and member functions to apply to location, directions, and surface normal vectors.
- *image*. A 2D array of RGB pixels with an output operation.

2 Slides

2.1 What is Computer Graphics?

2.1.1 What is computer graphics?

- Pictures generated by a computer.
- Tools used to make those pictures.
- Deals with all aspects of creating images with a computer
  - Hardware
  - Software
  - Applications
- Creation, storage, and manipulation of models and images
- Models come from diverse and expanding set of fields including physical, mathematical, artistic, biological, and conceptual structures.

2.1.2 What is interactive computer graphics?

- User controls contents, structure, and appearance of objects and their displayed images via rapid visual feedback
- Basic components of an interactive graphics system
  - Input (e.g., mouse, tablet and stylus, multi-touch)
  - Processing (and storage)
  - Display/Output (e.g., screen, paper-based printer, video recorder)

2.1.3 Purpose of computer graphics

- Communication is the purpose
- Human perception is the context
  - Techniques leverage visual perception abilities
- Fidelity is a tool, not (necessarily) the goal
  - Realism is great, but

  - Don't want to be limited to reality
    * Non-photorealistic rendering (NPR) is valuable
  - No apology is required for "approximation"

2.1.4 Ways graphics is output

- Frame-by-frame
- Frame-by-frame under control of user: sequence of frames drawn while user controlling change
- Animation: sequence of frames drawn that proceeds at a particular rate
- Interactive Graphics: sequence of frames under control of user by an input device, sequence of frames is not pre-determined

2.1.5 What drives computer graphics?

- Art, entertainment, and publishing
- Movie production and special effects
- Computer games
- Image processing and compute-aided design (CAD)
- Simulations, virtual environments and visualizations, and scientific analysis

2.2 OpenGL

- Device - Independent Graphics Programming
- Open-source graphics library/application programming interface (API)

2.2.1 Graphics Systems

- Windows-based
- Event-driven
  - Event queue
  - Callback functions
  - Event loop
  - Registering callback functions

2.2.2 Three Main OpenGL Libraries

- GL: basic, fundamental OpenGL library
  - Functions that are a permanent part of OpenGL
- GLUT: the GL Utility Toolkit
  - Managing windows, menus, and events
  - Window system independent toolkit for writing OpenGL programs
- GLU: the GL Utility Library
  - High-level routines to handle matric operations and complex drawings
- GLUI: the GL User Interface Library
  - High-level routines to handle matrix operations and complex drawings

2.2.3 Computer graphics

- Everything in the world is defined in 3 dimensions
  - 2D is a special case
- Objects and Cameras
- Define:
  - Objects and cameras
  - Lighting and shading
  - Final pixel values must be computed

2.2.4 Graphics Pipeline

- An application sends the pipeline a sequence of points $P_1, P_2, \ldots$ using commands such as:

```
glBegin(GL_LINES);
    glVertex3f(...);  // send P1 through the pipeline
    glVertex3f(...);  // send P2 through the pipeline
    ...
glEnd();
```

- World scene/object -> 3D modeling -> Viewing -> 3D Clipping -> Projection -> Rasterization -> 2D Pixelmap display

2.2.5 Graphics Primitives

- Points
- Lines
- Polylines
  - Joints
  - Thickness
- Text
- Filled regions
- Raster images

2.2.6 OpenGL Basic Graphics Primitives

- `GL_POINTS`
- `GL_LINES`
  - `glLineWidth(value);`
- `GL_LINE_STRIP`
- `GL_LINE_LOOP`
- `GL_POLYGON`
- `GL_QUADS`
- `GL_QUAD_STRIP`
- `GL_TRIANGLES`
- `GL_TRIANGLE_STRIP`
- `GL_TRIANGLE_FAN`

2.2.7 Code Sample

- Header

```
#include <GL/glut.h>
#include <GL/glu.h>
#include <GL/gl.h>
```

- OpenGL Basic Graphics Primitives

```
glClear(GL_COLOR_BUFFER_BIT);  // clear the screen
```

- OpenGL Points

```
glBegin(GL_POINTS)
    glVertex2i(100, 50);
    glVertex2i(100, 130);
    glVertex2i(150, 130);
glEnd();
```

- Registering events

```
glutDisplayFunc(displayFunc);
```

- Open a window

```
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize(width, height);  // size of window
glutInitWindowPosition(x, y);  // x = 0, y = 0
glutCreateWindow("Computer Graphics");
```

- Set up a coordinate system

```
void setWindow(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top) {
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(left, right, bottom, top);
    // gluOrtho2D(0.0, 640.0, 0.0, 480.0);
}

void setViewport(GLint left, GLint right, GLint bottom, GLint top) {
    glViewport(left, bottom, right - left, top - bottom);
}
```

- Adding pen properties

```
glClearColor(1.0, 1.0, 1.0, 0.0);  // background color
glColor3f(0.0f, 0.0f, 0.0f);  // set pen color to black
glPointSize(4.0);  // set size of pen 4x4 pixels
```

- Drawing Functions

  - $f(x) = e^{|-x|} \cos 2\pi x$.
  - OpenGL implementaiton:

```
glBegin(GL_POINTS);
for (glDouble x = 0; x < 4.0; x += 0.005) {
    glVertex2d(x, f(x));
}
glEnd();
glFlush();
```

2.2.8 Coordinate System / Viewports

- Coordinate System
  - Screen coordinates
  - Modeling task
    - Coordinates used to describe geometric objects
    - To create and/or use geometric objects
    - Space where geometric objects are described as is called the *world coordinates*
  - Viewing task
    - Coordinates used to size and position the pictures of those geometric objects on the display
    - To size and position the picture of those geometric objects on the display
- World coordinates use the Cartesian xy-coordinate system used in mathematics, based on whatever units are convenient.
- Define a rectangular world window in these world coordinates
  - Specifies which part of the world should be drawn
- Define a rectangular viewport in the screen window on the display
  - Objects specified for drawing in the world window appear automatically at proper sizes and locations within the viewport
  - In screen corrdinates (pixels)
  - A mapping is established by OpenGL
  - Mapping consists of scalings [change of size] and translations [change of positions]
- Windows and Viewports: world coordinates/natural coordinates for world window -> (OpenGL converts: mapping) Screen Coordinates (set up screen window and viewport)
- Screen coordinates
  - `x, y` vary from 0 to x number of pixels
  - Only positive values of x and y
  - Values must have a large range to get a good size drawing
- Window-to-Viewport Mapping
  - Windows are described by their left, top, right, and bottom values, `w.l`, `w.t`, `w.r`, `w.b`.
  - Viewport are described by the same values: `v.l`, `v.t`, `v.r`, `v.b`.
  - We want our mapping to be proportional
  - $sx = Ax + C$, $A = \frac{v.r - v.l}{w.r - w.l}$, $C = v.l - A \cdot W.l$.
  - $sy = By + D$, $B = \frac{v.t - v.b}{w.t - w.b}$, $D = v.b - B \cdot W.b$.
- Resizing the Screen Window

```
glutReshapeFunc(myReshape);
void myReshape(GLsizei W, GLsizei H);  // collects the new width and height for the window
```

  - User are free to alter the size and aspect ratio of the screen window

- Preserving Aspect Ratio
  - We want the largest viewport which preserves the aspect ration R of the world window
  - The screen window has width W and height H:
    - If $R > W/H$, the viewport should be width W and height W/R.
    - If $R < W/H$, the viewport should be width $H * R$ and height H.

2.2.9 Logical Input Graphics Primitives

- String
- Valuator
- Locator
- Pick
- Mouse Events
  - `glutMouseFunc(myMouse);`

```
void myMouse(int button, int state, int x, int y);
```

  - Button
    - `GLUT_LEFT_BUTTON`
    - `GLUT_MIDDLE_BUTTON`
    - `GLUT_RIGHT_BUTTON`
  - State
    - `GLUT_UP`
    - `GLUT_DOWN`
  - `glutMotionFunc(myMovedMouse);`
  - `glutPassiveMotionFunc(myMovedMouse);`
  - `glutKeyboardFunc(myKeyboard);`
  - Drawing Dots with a Mouse

```
void myMouse(int button, int state, int x, int y) {
    if (state == GLUT_DOWN) {
        if (button == GLUT_LEFT_BUTTON) {
            drawDot(x, screenHeight - y);
        }
        else if (button == GLUT_RIGHT_BUTTON) {
            glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
            glClear(GL_COLOR_BUFFER_BIT);
        }
    }
    glFlush();
}
```

- Keyboard Interaction

  - `glutKeyboardFunc(myKeyboard);`
  - `void myKeyboard(unsigned int key, int x, int y);`
  - `void myKeyboard(unsigned char key, int x, int y);`

```
swicth(key) {
    case 'p':
        drawDot(x, y);
        break;
    case 'E':
        exit(-1);
    default:
        break;
}
```

2.2.10 OpenGL Transformations

- The transform is done automatically by OpenGL!
- `modelview matrix`
- Grpahics Pipeline
  - $[x_0, y_0, z_0, w_0]^T \implies$ ModelView Matrix $\implies$ Projection Matrix $\implies$ Perspective Division $\implies$ Viewport Transform $\implies [x_w, y_w, z_w]^T$.
  - ModelView Matrix
- Routines
  - `glScaled(sx, sy, sz)`, `glScaled(sx, sy, 1.0)` for 2D.
  - `glTranslated(tx, ty, tz)`, `glTranslated(tx, ty, 0.0)` for 2D.
  - `glRotated(angle, ux, uy, uz)`, `glTranslated(angle, 0.0, 0.0, 1.0)` for 2D.
- Transformations
  - Transformations change 2D or 3D points and vectors, or change coordindate systems.
    - An *object transformation* alters the coordinates of each point on the object according to the same rule, leaving the underlying coordinate system fixed.
    - A *coordinate transformation* defines a new coordinate system in terms of the old one, then represents all of the object's points in this new system.
  - A corrdinate frame consists of:
    - a point O, called the origin
    - and some mutually perpendicular vectors (called i and j in the 2D case; i, j and k in the 3D case) that serve as the axes of the coordinate frame.
  - In 2D,
$$\tilde{P} = \begin{bmatrix} P_x \\ P_y \\ 1 \end{bmatrix}, \tilde{Q} = \begin{bmatrix} Q_x \\ Q_y \\ 1 \end{bmatrix}.$$
- Affine Transformations
  - Matrix form of the affine transformation in 2D:
$$\begin{bmatrix} Q_x \\ Q_y \\ 1 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ 1 \end{bmatrix} = \begin{bmatrix} m_{11}P_x + m_{12}P_y + m_{13} \\ m_{21}P_x + m_{22}P_y + m_{23} \\ 1 \end{bmatrix}.$$

- Translations
  - To translate a point P by a in the x direction and b in the y direction use the matrix:

$$\begin{bmatrix} Q_x \\ Q_y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ 1 \end{bmatrix} = \begin{bmatrix} P_x + a \\ P_y + b \\ 1 \end{bmatrix}.$$

- Scaling
  - Scaling is about the origin
  - If S_x or $S_y < 0$, the image is reflected across the x or y axis.
  - The matrix form is

$$\begin{bmatrix} Q_x \\ Q_y \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ 1 \end{bmatrix}.$$

  - Types of Scaling
    - Pure reflections, for which each of the cale facotrs is $+1$ or -1.
    - A uniform scaling, or a magnificaiton about the origin: $S_x = S_y$, magnification $|S|$.
    - If the scale factors are not the same, the scaling is called a *differential scaling*.
- Rotation
  - Counterclockwise around origin by angle θ.

$$\begin{bmatrix} Q_x \\ Q_y \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ 1 \end{bmatrix}$$

  - All 2D rotation are R_z. Two roatations combine to make a rotation given by the sume of the rotation angles, and the matrices commute.
  - In 3D the situation is much more complicated, because rotations can be about different axes.
  - The order in which two rotations about different axes are performed does matter: 3D rotation matrices do not commute.
- Shear

$$\begin{bmatrix} Q_x \\ Q_y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & h & 0 \\ g & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ 1 \end{bmatrix}$$

  - Shear H about origin: x depends linearly on y in the figure.
  - Shear along x: $h \neq 0$ and P_x depends on P_y.
  - Shear along y: $g \neq 0$, P_y depends on P_x.

- Summary: OpenGL Transformations
  - Current Transformation Stack
  - `glScaled(sx, sy, sz);`
  - `glTranslated(tx, ty, tz);`
  - `glRotated(angle, ux, uy, uz);`
  - Post-multiply: $CT = CTM^T$.

- Affine Transformations Stack
  - It is also possible to push/pop the current transformation from a stack in OpenGL, using the commands

```
glMatrixMode(GL_MODEVIEW);
glPushMatrix();  // glPopMatrix();
```

  - Caution: Popping a stack that contains only one matrix is an error.
  - `glGet(GL_MODELVIEW_STACK_DEPTH);`

- Wrapper Functions

```
pushCT() {
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();  // push a copy of the top matrix
}

checkStack() {
    if (glGet(GL_MODELVIEW_STACK_DEPTH) <= 1)
        // do something
    else
        popCT();
}

popCT() {
    glMatrixMode(GL_MODELVIEW);
    glPopMatrix();
}
```

2.3 Projection and the Camera

2.3.1 Modelview Matrix

- The graphics pipeline:
  - Modelview matrix is composed of the scene transformations, M, and the camera transformations, V.

2.3.2 Projection Transformation

- View Volume
  - Parallel
  - Perspective
  - Isometric
- Types of Projection
  - Parallel (orthographic)
  - Perspective
- Important to Control
  - Projection type
  - Field of view and image aspect ratio
  - Near and far clipping planes

2.3.3 Parallel (Orthographic) Projection

- No foreshortening effect
  - Distance from camera does not matter
  - Objects are same size no matter what
- The center of projection is at infinity
- Projection caculation (just choose to equal the z coordinates)

2.3.4 Perspective Projection

- Similar to real world
- Foreshortening: Objects appear larger if they are closer to camera
- Need to define
  - Center of Projection
  - Projection (view plane)
  - Connecting the object to the center of projection

2.3.5 Field of View

- Determine how much of the world is going to be "seen"
- Larger field of view = smaller object projection size

2.3.6 The Camera and Perspective Projection

- The camera has an eye (or view reference point VRP) at some point in space.
- Its view volume is a portion of a pyramid, whose apex is at the eye.
- The straight line from a point P to the eye is called the projector of P. (All projectors of a point meet at the eye).
- The axis of the view volume is called the view plane normal, or VPN.
- The opening of the pyramid is set by the view angle θ.
- Three planes are defined perpendicular to the VPN: the near plane, the view plane, and the far plane.
- The windows have an aspect ratio which can be set in a program.
- OpenGL clips points of the scene lying outside the view volume (view frustum).
- Points P inside the view volume are projected onto the view plane to a correpsonding point P'.
- Finally, the image formed on the view plane is mapped into the viewport, and becomes visible on the display device.

2.3.7 Setting the View Volume

- The default camera position has the eye at the origin and the VPN aligned with the z-axis.
- Define a *look point* as a point of particular interest in the scene, and together the two points eye and look define the VPN as *eye-look*.
  - This is later normalized to become the vector $\vec{n}$, which is the central in specifying the camera properly.

2.3.8 Projection Transformation

- Set `glMatrixMode` to `GL_PROJECTION`.
- For perspective projection use `gluPerspective(foxy, aspect, near, far);` or `glFrustrum(left, right, bottom, top, new, far);`
- For orthographic projection, use `glOrtho(left, right, bottom, top, near, far);`

2.3.9 Ascpect Ratio

- Used to calculate window width
- Aspect $= W/H$.

```
void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity():
    gluPerspective(foxy, aspect, near, far);
    glMatrixMode(GL_MODELVIEW);
```

```
    glLoadIdentity();
    gluLookAt(ex, ey, ez, Lx, Ly, Lz, Upx, Upy, Upz);
    Draw();
}
```

2.3.10 3D Viewing

- Similar to your digital camera
- You can control "lens" of camera
- Project 3D world onto a 2D screen

2.3.11 Viewing Transformation

- Setting up the camera: `gluLookAt(eye_x, eye_y, eye_z, look_x, look_y, look_z, up_x, up_y, up_z);`
  - `Up` is usually $[0, 1, 0]$.
  - First set mode to `GL_MODELVIEW`.
- `gluLookAt` fills V part of modelview matrix.
- Modelview Matrix
  - Combination of modeling matrix M and Camera transforms V.

2.3.12 Setting the View Volume

- Set up the camera's position and orientation in exactly the same way we did for the parallel-projection camera.

```
void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity(): // start with a unit matrix
    gluLookAt(eye.x, eye.y, eye.z, look.x, look.y, look.z, up.x, up.y, up.z);
    drawScene();
}
```

2.3.13 Camera with Arbitray Orientation and Position

- $\mathbf{v}$ points vertically upward, $\mathbf{n}$ away from the view volume, and $\mathbf{u}$ at right angles to both $\mathbf{n}$ and $\mathbf{v}$.
- The camera looks toward $-\mathbf{n}$. All are normalized.

2.3.14 How do we construct u, v and n?

- New origin = eye position
- Axes:
  - $\mathbf{n}$ must be parallel to eye - look, so it sets $n = \text{eye} - \text{look}$.
  - $\mathbf{n} = (\text{eye} - \text{look})/|\text{eye} - \text{look}|$.
  - $\mathbf{u}$ points "off to the side", so it makes $\mathbf{u}$ perpendicular to both $\mathbf{u}$ and $\mathbf{up}$: $\mathbf{u} = \mathbf{up} \times \mathbf{n}$.
  - $\mathbf{u} = (\mathbf{up} \times \mathbf{n})/|\mathbf{up} \times \mathbf{n}|$.

2.3.15 World to Eye Transformation

- Use $\mathbf{n}$, $\mathbf{u}$, and $\mathbf{v}$ to compose V for the modelview matrix.

- The view matrix V created by `gluLookAt` is

$$V = \begin{bmatrix} u_x & u_y & u_z & d_x \\ v_x & v_y & v_z & d_y \\ n_x & n_y & n_z & d_z \\ 0 & 0 & 0 & 0 \end{bmatrix},$$

  where $d_x = -\text{eye} \cdot \mathbf{u}$, $d_y = \text{eye} \cdot \mathbf{v}$, $d_z = -\text{eye} \cdot \mathbf{n}$.
- V is postmultiplied by M to form the modelview matrix VM.

2.3.16 `Point3` Class

```
class Point3 {
public:
    float x, y, z;

    void set(float dx, float dy, float dz) {
        x = dx;
        y = dy;
        z = dz;
    }

    void set(Point3& p) {
        x = p.x;
        y = p.y;
        z = p.z;
    }

    Point3(float x, float y, float z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    Point3() {
        x = y = z = 0;
    }

    void build4tuple(float v[]) {
        // load 4-tuple with this color: v[3] = 1 for homogeneous
        v[0] = x;
        v[1] = y;
        v[2] = z;
        v[3] = 1.0f;
    }

}
```

2.3.17 `Vector3` Class

```
class Vector3 {
public:
    float x, y, z;

    void set(float dx, float dy, float dz) {
```

```
    x = dx;
    y = dy;
    z = dz;
}

void set(Vector3& v) {
    x = v.x;
    y = v.y;
    z = v.z;
}

void flip() {
    x = -x;
    y = -y;
    z = -z;
}

void setDiff(Point3& a, Point3& b) {
    x = a.x - b.x;
    y = a.y - b.y;
    z = a.z - b.z;
}

// adjust this vector to unit length
void normalize() {
    double norm = sqrt(pow(x, 2) + pow(y, 2) + pow(z, 2));
    x /= norm;
    y /= norm;
    z /= norm;
}

Vector3 (float x, float y, float z) {
    this.x = x;
    this.y = y;
    this.z = z;
}

Vector3 (Vector3& v) {
    x = v.x;
    y = v.y;
    z = v.z;
}

Vector3 () {
    x = y = z = 0;
}

Vector3 corss(Vector3 b) {
    Vector3 ret;
    ret.x = y * b.z - b.y * z;
    ret.y = z * b.x - x * b.z;
    ret.z = x * b.y - y * b.x;
    return ret;
}
```

```
    float dot(Vector3 b) {
        return a.x * b.x + a.y * b.y + a.z * b.z;
    }
}
```

2.3.18 Camera Class

```
class Camera {
private:
    Point3 eye;
    Vector3 u, v, n;
    double viewAngle, aspect, nearDist, farDist;  // view volume shape
    void setModelviewMatrix();  // tell OpenGL where the camera is

public:
    Camera();  // constructor

    void Camera :: set(Point3 Eye, Point3 look, Vector3 up) {
        // create a modelview matrix and send it to OpenGL
        eye.set(Eye);  // store the given eye position

        // make n
        n.set(eye.x - look.x, eye.y - look.y, eye.z - look.z);
        n.normalize();  // make n unit length
        u.set(up.cross(n));  // make u = up x n
        u.normalize();
        v.set(n.cross(u));  // make v = n x u
        v.normalize();  // make v unit length

        setModelViewMatrix();  // tell OpenGL
    }

    void Camera :: setModelviewMatrix() {
        // load modelview matrix with existing camera values
        float m[16];
        Vector3 eVec(eye.x, eye.y, eye.z);  // a vector version of eye
        m[0] = u.x; m[4] = u.y; m[8] = u.z; m[12] = -eVec.dot(u);
        m[1] = v.x; m[5] = v.y; m[9] = v.z; m[13] = -eVec.dot(v);
        m[2] = n.x; m[6] = n.y; m[10] = n.z; m[14] = -eVec.dot(n);
        m[3] = 0; m[7] = 0; m[11] = 0; m[15] = 1.0;
        glMatrixMode(GL_MODELVIEW);
        glLoadMatrix(m);  // load OpenGL's modelview matrix
    }

    void Camera :: roll(float angle) {
        // roll the camera through angle degress
        double pi = 4 * atan(1);
        float cs = cos(pi / 180 * angle);
        float sn = sin(pi / 180 * angle);

        Vector3 t(u);  // remember old u
        u.set(cs * t.x - sn * v.x, cs * t.y - sn * v.y, cs * t.z - sn * v.z);
        v.set(sn * t.x + cs * v.x, sn * t.y + cs * v.y, sn * t.z + cs * v.z);
```

```
        setModelViewMatrix();
    }

    void pitch(float angle);  // increase pitch
    void yaw(float angle);  // yaw it
    void slide(float delU, float delV, float delN) {
        eye.x += delU * u.x + delV * v.x + delN * n.x;
        eye.y += delU * u.y + delV * v.y + delN * n.y;
        eye.z += delU * u.z + delV * v.z + delN * n.z;
        setModelViewMatrix();
    }
}
```

2.3.19 Flying the Camera through a Scene

- The user can fly the camera through a scene interactively by pressing keys or clicking the mouse.
  - pressing 'u' might slide the camera up some amount.
  - pressing 'y' might yaw it to the left.
  - pressing 'f' might slide it forward.
- There are six degrees of freedom for adjusting a camera:
  - it can fly in three dimensions.
  - it can be rotated about any of three coordinate axes.
- Sliding a camera means to move it along one of its *own* axes, that is, in the **u**, **v** or **n** direction, without rotating it.
- Since the camera is looking along the negative **n**-axis, movement along **n** is forward or back.
- Similarly, movement along **u** is left to right, and along **v** is up or down.
- We want to roll, pitch, or yaw the camera (rotate it around one of its own axes).
- We look at rolling in detail; yaw and pitch are similar.

2.4 Modeling Shapes with Polygonal Meshes

2.4.1 Polygonal Meshes

- A polygonal mesh is a collection of polygons (faces) that approximate the surface of a 3D object.
- Polygons are easy to represent (by a sequence of vertices) and transform.
- They have simple properties (a single normal vector, a well-defined inside and outside, etc.).
- They are easy to draw.
- Meshes are a standard way of representing 3D objects in graphics.
- A mesh can approximate the surface to any degree of accuracy by making the mesh finer or coarser.
- Meshes can model both solid shapes and thin skins.
  - The object is solid if the polygonal faces fit together to enclose space.
  - The faces fit together without enclosing space.
- A polygon mesh is described *by a list of polygons*, along with information about the direction in which each polygon is facing.
- If the mesh represents a *solid*, each face has an *inside and an ouside relative to the rest of the mesh*.
- In such a case, the directional information is often simply the **outward pointing normal vector** to the plane of the face used in the shaing process.
- The normal direction to a face determines its brightness.
- Each vertex V_1, V_2, V_3, and V_4 defining the side wall of the barn has the *same* normal $\mathbf{n}_1$, the normal vector to the side wall.
- But vertices of the front wall, such as V_5, will use normal vector $\mathbf{n}_2$.

- For some objects, we associate a *normal vector to each vertex* of a face rather than one vector to an entire face.
  - We use meshes, which represent objects with smoothly curved faces such as a sphere or cylinder.
  - "Smooth-underlying surface".
- For the smoothly curved surface of the cylinder, both vertex V_1 of face F_1 and V_2 on face F_2 use the same normal $\mathbf{n}$, the vector perpendicular to the underlying smooth surface.
- Defining a Polygonal Mesh
  - A mesh consists of 3 lists: the *vertices* of the mesh, the outside *normal* at each vertex, and the faces of the mesh.
  - The vertex list reports the locations of the distinct vertices in the mesh.
  - The list of normals reports the directions of the distinct normal vectors that occur in the model.
  - Calculating Normals: Take any three non-collinear points on the face, V_1, V_2, V_3, and compute the normal as their cross product $\mathbf{m} = (V_1 - V_2) \times (V_3 - V_2)$ and normalize it to unit length.
    - If the two vectors $V_1 - V_2$ and $V_3 - V_2$ are nearly parallel, the cross product will be very small and numerical inaccuracies may result.
    - The polygon surface represented by the vertices cannot be truly flat.
    - We need to form *some average value* for the normal to the polygon, one that takes into consideration *all of the vertices*.
  - Newell's Method for Normals
    - Given N vertices, define `next(i) = ni = (i + 1) mod n`.
    - Traverse the vertices for the face in counter-clockwise order from the outside.
    - The normal given by the values on the next slide points to the outside (front) of the face.
    - $n_x = \sum_{i=0}^{N-1}(y_i - y_{ni})(z_i + z_{ni})$.
    - $n_y = \sum_{i=0}^{N-1}(z_i - z_{ni})(x_i + x_{ni})$.
    - $n_z = \sum_{i=0}^{N-1}(x_i - x_{ni})(y_i + y_{ni})$.
  - The face list indices into the vertex and normal lists: The list of vertices for a face begins with any vertex in the face, and then proceeds around the face vertex by vertex until a complete circuit has been made.
    - There are two ways to traverse a polygon:
      - clockwise.
      - counterclockwise (convention)
      - Using counterclockwise order, if you traverse around the face by walking from vertex to vertex, the inside of the face is on your left.
      - Using the convention allows algorithms to distinguish with ease the front from the back of a face.
      - If we use an underlying smooth surface, such as a cylinder, normals are computed for that surface.
- Properties of Meshes
  - A mesh is convex if the line connecting any two interior points is entirely inside the mesh.
  - Exterior connecting lines are shown for non-convex objects below (step and torus).
  - A closed mesh represents a solid object (which encloses a volume).
  - A mesh is connected if there is an unbroken path along the edges of the mesh between any two vertices.
  - A mesh is simple if it has not holes, e.g., a sphere is simple; a torus is not.
  - A mesh is planar if every face is a plane polygon. Triangular meshes are frequently used to enforce planarity.
- Platonic Solids
  - All the faces are identical and each is a regular polygon
    - Tetrahedron

    * Hexahedron
    * Octahedron
    * Isosahedron
    * Dodecahedron

- Extruded Shapes
  - A large class of shapes can be generated by *extruding* or *sweeping* a 2D shape through space.
  - In addition, *surfaces of revolution* can also be approximated by *extrusion of a polygon once* we slightly broaden the definition of extrusion.
  - A prism is formed by moving a regular polygon along a straight line, when the line is perpendicular to the polygon, the prism is a right prism.
  - Base has vertices $(x_i, y_i, 0)$ and top has vertices (x_i, y_i, H).
  - Each vertex (x_i, y_i, H) on the top is connected to coorresponding vertex $(x_i, y_i, 0)$ on the base.
  - If the polygon has n sides, then
    * n vertical sides of the prism.
    * a top side (cap).
    * a bottom side (base), or $n + 2$ faces altogether.
  - The normals for the prism are the fae normals.
  - These may be obtained using the Newell method, and the normal list for the prism constructed.
  - Arrays of Extruded Prisms
    * OpenGL can reliably draw only convex polygons. For non-convex prisms, stack the parts.
  - Vertex List for the Prism
    * Suppose the prism's base is a polygon with N vertices (x_i, y_i).
    * We number the vertices of the base $0, \ldots, N-1$ and those of the cap $N, \ldots, 2N-1$, so that an edge joins vertices i and $i + N$.
    * The vertex list is then easily constructed to contain the points $(x_i, y_i, 0)$ and (x_i, y_i, H) for $i = 0, 1, \ldots, N-1$.
  - Face List for the Prism
    * We first make the side faces and then add the cap and base.
    * For the j-th wall ($j = 0, \ldots, N-1$), we create a face with four vertices having indices $j, j + N, next(j) + N$, and $next(j)$ where $next(j)$ is $j + 1\%N$.
    * `if (j < n - 1) next = ++j; else next = 0;` or `j = (++j) % N;`

2.4.2 3D Modeling

- Polygonal meshes capture the shape of complex 3D objects in simple data structures.
  - Platonic solids, the Buckyball, geodesic domes, prisms.
  - Extruded or swept shapes, and surfaces of revolution.
  - Solids with smoothly curved surfaces.
- Animated Particle systems:
  - Each particle responds to conditions.
  - Position, a velocity, and perhaps a color, lifetime, size, degree of transparency, and shape.
  - Randomly generated.
- Physically based systems
- Modeling of Terrain

2.4.3 Using `GLUT`

- `glutSolidSphere`, `glutWireSphere`
  - Render a solid or wireframe sphere respectively.
  - Usage:

    - `void glutSolidSphere(GLdouble radius, GLint slices, GLint stacks);`
    - `void glutWireSphere(GLdouble radius, GLint slices, GLint stacks);`
    - `radius`: The radius of the sphere.
    - `slices`: The number of subdivisions around the z-axis (similar to lines of longitude).
    - `stacks`: The number of subdivisions along the z-axis (similar to lines of latitude).

- `glutSolidCube`, `glutWireCube`
- `glutSolidCone`, `glutWireCone`
- `glutSolidTorus`, `glutWireTorus`
- `glutSolidDodecahedron`, `glutWireDodecahedron`
- `glutSolidOctahedron`, `glutWireOctahedron`
- `glutSolidTetrahedron`, `glutWireTetrahedron`
- `glutSolidIcosahedron`, `glutWireIcosahedron`
- `glutSolidTeapot`, `glutWireTeapot`

2.4.4 Visual Realism Requirements

- Light Sources
- Materials (e.g., plastic, metal)
- Shading Models
- Depth Buffer Hidden Surface Removal
- Textures
- Reflection
- Shadows

2.4.5 Rendering Objects

- We know how to model mesh objects, manipulate a camera, view objects, and make pictures.
- Now we want to make these objects look visually interesting, realistic, or both.
- We want to develop methods of *rendering* a picture of the objects of interest: *computing* how each pixel of a picture should look.
- Much of rendering is based on different **shading models**, which describe how light form light sources interacts with objects in a scene.
    - It is impractical to simulate all of the physical principles of light scattering and reflection.
    - A number of approximate models have been invented that do a good job and produce various levels of realism.

2.4.6 Rendering

- Rendering: deciding how a pixel should look.
- Example: compare wireframe (left) to wire frame with hidden surface removal (right)
- Example: Compare mesh objects drawn using wire-frame, flat shading, smooth (Gouraud) shading.

2.4.7 Basic Illumination, Materials, and Shading

- Problem: Model light/surface point interactions to determine the final color and brightness of your objects.
- Need to apply lighting model at set of points across the entire surface of the mesh.

2.4.8 Illumination Model

- Light Attributes
    - Intensity
    - Color
    - Position

  - Direction
  - Shape
- Object Surface Attributes
  - Color
  - Reflectivity
  - Transparency
- Interaction among lights and objects

2.4.9 Local Illumination

- Light Attributes
- Position/Orientation of the Camera
- Object Material Properties

2.4.10 Global Illumination

- Multiple reflections of light from all surfaces in the scene and from light sources that populate the environment light sources that populate the environment
  - such as light coming through a window, fluorescent lamps, etc.
  - Example: Ray Tracing

2.4.11 Simple Local Illumination - Using OpenGL

- Ambient
- Diffuse
- Specular
- Final Result = ambient + diffuse + specular
- Materials reflect each component differently
- K_a, K_d, K_s -> material reflection coefficients

2.4.12 Ambient Light

- Uniform background glow or lighting
- Soften shadows
- Light scattered by the environment, constant
- Simple approximation of global illumination
- Soft non-directional light, spreads uniformly
  - Independent of light position
  - Independent of object orientation
  - Independent of camera position/orientation Ambient = la x Ka

2.4.13 Diffuse Light

- Reflection of light form surface that a surface received from a light source that reflects equally in all directions
  - Dependent on Light Sources (s)
  - Indpedent of position of camera

2.4.14 Add Ambient Light to Diffuse Reflection

- The diffuse and ambient sources have intensity 1.0, and $K_d = 0.4$.
- $K_a = 0, 0.1, 0.3, 0.5, 0.7$
- Modesta ambient light softens shadows
- Too much ambient ligh washes out shadows

2.4.15 Specular Light

- Bright spot of light on the surface
- Result of total reflection of the light on a concentrated region
  - Dependent on light source
  - Dependent on camera position/orientation
- Example
  - Becuase specular light is mirror-like, the color of the specular component is often the same as that of the light source.
    - E.g., the specular highlighting seen on a glossy red apple when illuminated by a yellow light is yellow rather than red.
  - To create specular highlights for a plastic-like surface
    - set the specular reflection coefficients $\rho_{sr} = \rho_{sg} = \rho_{sb} = \rho_s$.
    - so that the reflection coefficients are 'gray' in nature and
    - do not alter the color of the incident light.
  - THe designer might choose $\rho_s = 0.5$ for a slightly shiny plastic surface, or $\rho_s = 0.9$ for a highly shiny surface.

2.4.16 Materials in OpenGL

- A material is defined by ρ_a, ρ_s, ρ_d, and f for each of the RGB components
- To set your own material properties, creates a `GLfloat` array of reflectivities ρ, one each for RGB.
  - Set the A (alpha) reflectivity to 1.0.
- Then call `glMaterialfv(arg1, arg2, arrayname);`
  - `arg1` can be `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`.
  - `arg2` can be `GL_DIFFUSE`, `GL_AMBIENT`, `GL_SPECULAR`, `GL_AMBIENT_AND_DIFFUSE`, or `GL_EMISSION`.
  - `GL_EMISSION`: A surface may be emissive (glow) like the sun. We give it an intensity I_e, as a `GLfloat` array of 4 values for the emissive color.
  - The `I_e` is added to the other light intensities.
- total $= I_a + I_d + I_s + I_e$.
- OpenGL adds together the contributions of emissive sources, ambient light, and light from sources to find the **total light** in a scene for each of red, green and blue.
- OpenGL does not allow the total light intensity to be more than 1.0; it is set to 1.0 if the sum exceeds 1.0.
- Example: Lighting Colors Mix with Colors of Objects
  - If the color of a sphere is 30% red, 45% green, and 25% blue.
  - It makes senes to set its ambient and diffuse relection coefficients to $(0.3K, 0.45K, 0.25K)$ respectively, where K is some scaling value that determines the overall fraction of incident light that is refelcted from the sphere.
  - Now if it is bathed in white light having equal amount of red, green, and blue ($I_{sr} = I_{sg} = I_{sb} = I$).
  - The individual diffuse components have intensities $I_r = 0.3KI, I_g = 0.45KI, I_n = 0.25KI$, so as expected we see a color that is 30% red, 45$ green, and 25% blue.
  - Suppose a sphere has ambient and diffuse reflection coefficients $(0.8, 0.2, 0.1)$, so it appears mostly red when bathed in white light.
  - We illuminate it with a greenish light $I_s = (0.15, 0.7, 0.15)$.
  - The reflected light is then given $0.12, 0.14, 0.015)$, which is a fairly even mix of red and green, and would appear yellowish.
    - $0.12 = 0.8 \times 0.15$, $0.14 = 0.2 \times 0.7$, $0.015 = 0.1 \times 0.15$.

2.4.17 Lighting in OpenGL

- Basic Light Sources: Dependent or independent on the distance between light source and object.
  - Point light
  - Spot light
  - Directional light
  - Area light
- Creating and using light sources in OpenGL
  - Global ambient light is present even if no lights are created. Its default color is $(0.2, 0.2, 0.2, 1.0)$.
  - To change this value, create a `GLfloat` array of values `newambient` and use the statement `glLightModelfv(GL_LIGHT_M newambient);`
  - Default light values are
    - $(0, 0, 0, 1)$ for ambient
    - $(1, 1, 1, 1)$ for diffuse and specular
    - $(0, 0, 0, 1)$ for all other
  - Light sources are given a number in $[0, 7]$: `GL_LIGHT0`, `GL_LIGHT1`, etc.
  - Lights do not work unless you turn them on.
    - In your main program, add the statement `glEnable(GL_LIGHTING)`.
    - To turn off a light, `glDisable(GL_LIGHT0);`
    - To turn them all off, `glDisable(GL_LIGHTING);`
  - The light color is specified by a 4-component array $[R, G, B, A]$ of `GLfloat`, named `amb0`. The A value can be set to 1.0 for now.
  - The light color is specified by `glLightfv(GL_LIGHT_0, GL_AMBIENT, amb0);` Similar statements specify `GL_DIFFUSE` and `GL_SPECULAR`.
  - Example

```
GLfloat amb0[] = {0.2, 0.4, 0.6, 1.0};
GLfloat diff0[] = {0.8, 0.9, 0.5, 1.0};
GLfloat spec0[] = {1.0, 0.8, 1.0, 1.0};

glLightfv(GL_LIGHT0, GL_AMBIENT, amb0);
glLightfv(GL_LIGHT0, GL_DIFFUSE, diff0);
glLightfv(GL_LIGHT0, GL_SPECULAR, spec0);
```

  - Each light has a position specified in homogeneous coordinates using `GLfloat` array named, for example, `litePos`.
  - A basic point light is created using `glLightfv(GL_LIGHT0, GL_POSITION, litePos);`
  - If the postion is a vector, the 4^{th} component to be 0, the sources is infinitely remote (like the sun).
  - Infinitely remote light sources are often called "directional".
  - Creating and using spotlights in OpenGL
    - To create the spotlight, create a `GLfloat` array for liteDirection.
    - Default values are $\mathbf{d} = (0, 0, 0, 1)$.
    - Then add the statements
      - `glLightfv(GL_LIGHT0, GL_POSITION, litePos);`
      - `glLightfv(GL_LIGHT0, GL_DIRECTION, liteDirection);`
    - A spotligth emits light only in a cone of directions;
    - There is no light outside the cone.

* Inside the cone, $I = I_s(\cos\beta)^\epsilon$, where $\cos\beta$ uses the angle between $\mathbf{d}$ and a line from the source to P.
* To create the spotlight, create a `GLfloat` array for $\mathbf{d}$.
* Default values are $\mathbf{d} = (0, 0, 0, 1), \alpha = 180°, \epsilon = 0$, a point source.
* Then add the statements
  - `glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 45.0);`
  - `glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, 4.0);`
  - `glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, d);`
* Attenuation of Light with Distance
  - OpenGL attenuates the strength of a positional light source by the following attenuation factor:
$$atten = \frac{1}{k_c + k_l D + k_q D^2},$$
where k_c, k_l and k_q are coefficients and D is the distance between the light's position and the vertex in question.
  - These parameters are controlled by function call: `glLightfv(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 2.0);`
  - and similarly for `GL_LINEAR_ATTENUATION`, and `GL_QUADRATIC_ATTENUATION`.
  - The default values are $k_c = 1$, $k_l = 0$ and $k_q = 0$ (no attenuation).

– Moving Light Sources in OpenGL
* To move a light source independently of the camera:
  - set its position array,
  - clear the color and depth buffers,
  - set up the modelview matrix to use for everything except the light source and push it,
  - move the light source and set its position,
  - pop the matrix,
  - set up the camera, and draw the objects.

– Code: Independent Motion Light

```
void display() {
    GLfloat position[] = {2, 1, 3, 1};  // initial light position
    // clear color and depth buffers
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glPushMatrix();
    glRotated(...); glTranslated(...);  // move the light
    glLightfv(GL_LIGHT0, GL_POSITION, position);
    glPopMatrix();
    gluLookAt(...);  // set the camera position
    < draw the object... >
    glutSwapBuffers();
}
```

– Code: Light Moves with Camera

```
void display() {
    GLfloat position[] = {camEye.x, camEye.y, camEye.z, 1};
    // clear color and depth buffers
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glLightfv(GL_LIGHT0, GL_POSITION, position);
    gluLookAt(...);  // move light and camera
    <draw the object...>
    glutSwapBuffers();
}
```

2.4.18 Shading Models

- Rendering
  - Compare to images with specular highlights, shadows, textures.
  - Assume to start with that light has no color, only brightness: $R = G = B$.
  - Assume we also have a point source of light (sun or lamp) and general ambient light, which doesn't come directly from a source, but through windows or scattered by the air.
    - Ambient light comes equally from all directions.
    - Point-source light comes from a single light.
  - When light is reflected from an object, some of the reflected light reaches our eyes, and we see the object.
    - **Diffuse** refelction: some of the light slightly penetrates the surface and is re-radiated uniformly in all directions. The light takes on some fraction of the color of the surface.
    - **Specular** reflection: more mirror-like. Light is reflected directly from the objects outer surface, giving rise to highlights of a approximately the same color as the source. The surface looks shiny.
    - The total light reflected from the surface in a certain direction is the sume of the diffuse componenet and the specular component.
      - For each surface point of interest, we compute the size of each component that reaches the eye.
    - Finding Reflected Light: a model
      - Model is not completely physically correct, but it provides fast and relatively good results on the screen.
      - Intensity of a light is related to its brightness. We will use I_s for intensity, where s is R or G or B.
  - Calculating Reflected Light
    - To compute reflected light at point P, we need 3 vectors: normal $\mathbf{m}$ to the surface at P and vectors $\mathbf{s}$ from P to the source and $\mathbf{v}$ from P to the eye. We use world coordinates.
    - Each face of a mesh objects has an inside and an outside.
    - Normally the eye sees only the outside (front, in OpenGL), and we calculate only light refelcted from the outside
    - If the eye can see inside, we must also compute reflections from the inside (back, in OpenGL).
      - If $\mathbf{v} \cdot \mathbf{m} > 0$, the eye can see the face and lighting must be calculated.
      - `glLightModeli(GL_LIGHT_MODEL_TWO_SIDES, GL_TRUE)` calculates lighting for both front and back faces - in case you have open boxes, for example.
  - Shading and the Graphics Pipeline
    - Shading is applied to a vertex at the point in the pipeline where the projection matrix is applied. We specify a normal and a position for each vertex.
    - `glNormal3f(norm[i].x, norm[i].y, norm[i].z)` specifies a normal for each vertex that follows it.
    - The modelview matrix transforms both vertices and normals ($\mathbf{m}$), the latter by $M^{-T}\mathbf{m}$, where M^{-T} is the transpose of the inverse matrix M^{-1}.
    - The positons of lights are also transformed.
    - Then a color is applied to each vertex, the perpective transformation is applied , and clipping is done.
    - Clipping may create new vertices which need to have colors attached, usually by linear interpolation of initial vertex colors.
    - If the new point a is 40% of the way from v_0 to v_1, the color associated with a is a blend of 60% of (r_0, g_0, b_0) and 40% of (r_1, g_1, b_1): color at point $a = (lerp(r_0, r_1, 0.4), lerp(g_0, g_1, 0.4), lerp(b_0, b_1, 0.4))$.
- The normal vectors determine the shading type.
  - **Flat** shading emphasizes individual polygons (barn).

  - **Smooth** (Gouraud or Phong) shading emphasizes the underlying surface (torus).

- Flat Shading
  - We must calculate the correct color c for each pixel from the vertex colors.
  - Flat Shading: Use the same color for every pixel in a face - usually the color of the first vertex.
  - OpenGL implements this type of coloring using `glShadeModel(GL_FLAT);`
  - Specular hightlights are rendered poorly with flat shading:
    - If there happens to be a large specular component at the representative vertex, that brightness is drawn uniformly over the entire face.
    - If a specular highlight doesn't falll on the representative point, it is missed entirely.

- Smooth (Gouraud) Shading
  - Tires to de-emphasize edges between faces.
  - Smooth shading uses linear interpolation of colors between vertices: top has color c_4 and bottom has color c_1, left has color $c_2 = lerp(c_1, c_4, f)$, where $f = (y_{\text{left}} - y_{\text{bottom}})/(y_{\text{top}} - y_{\text{bottom}})$.
  - OpenGL implements this coloring with `glShadeModel(GL_SMOOTH);`
  - Why do the edges disappear with this technique?
  - When rendering F, the colors c_L and c_R are used, and when rendering F' the color c'_L and c'_R are used.
  - But since $c_R = c'_L$, there is no abrupt change in color at the edge along the scanline. c_L and c_R are calculated using normals that are found by averaging the face normals of all the faces that abut the edge.
  - Because colors are formed by interpolating rather than computing colors at every pixel. Gouraud shading does not render highlights well.
  - Consequently, we usually do not include the specular refelction component in the shading computation.

2.4.19 Color Theory and Perception

- Basics
  - Color is made from three basic primaries:
    - Red
    - Green
    - Blue
  - This model is related to the way the eye operates
  - Consistent with how graphics displays work
    - Mixing amounts of red, green, and blue colors
  - Color depends on subtle interactions between
    - Physics of light radiation
    - Eye-brain system

- Light
  - Light is an electromagnetic phenomenon, like television waves, infared radiation, and x-rays.
  - By light, we mean those waves that lie in a narrow band of wavelengths in the so-called *visible spectrum*.
  - The wavelength is the distance light travels during one cycle of its vibration.
  - Wavelength λ and frequency f are related by $\lambda = v/f$, where v is the speed of light in the medium of interest.
    - In air (or a vacuum) $v = 300,000 km/s$, in glass, it is about 65% as fast.
  - The figure shows the location of the visible specturum (for humans) within the entire electromagnetic spectrum.

- The Human Eye

  - The *retina* of the eye is its its light-sensittive membrane. It lines the rear portion of the eye's wall and contains two kinds of receptor cells: cones and rods.
  - The *cones* are the color-sensitive cells, each of which responds to a particular color, red, green, or blue.
  - The rods cannot distinguish colors, nor can they see fine detail.
  - According to the *tri-stimulus theory*, the color we see is the result of our cones' relative responses to red, green, and blue light.
  - The human eye can distinguish about 200 intensities of red, green, and blue, each.
  - An eye has 6 to 7 million cones, concentrated in a small portion of the retina called *fovea*.
  - each cone has its own nerve cell, allowing the eye to discern tiny details.
    * To see an object in detial, the eye looks directly at it in order to bring the image onto the fovea.

- Color Spaces
  - If one can quantify three descriptors, one can then describe a color by means of a 3-tuple of values.
  - The different choices of coordinates then give rise to different color spaces, and we need ways to convert color descriptions from one color space to another.
  - Often people use *hue, saturation, and brightness* to think of mixing color.
  - Artists use *tints, shades, and tones*.
  - Computer Scienties usually think of mixing color in *red, green and blue*.
    * But, not all colors can be represented by RGB (most can).

- RGB
  - The RGB color model describes colors as *positive combinations* of three primaries: red, green, and blue.
  - If the scalars r, g and b are confined to values between 0 and 1, all definable colors will lie in the cube shown.
  - There is no normalization for the intensity of the color.
  - Points close $(0, 0, 0)$ are dark, and those farther out are ligher.
  - The corner marked magenta properly signifies that red light plus blue light produces magenta light, and similarly for yellow and cyan.

- Hue - Ink Mixing
  - *Substractive color systems* are used when it is natural to think in terms of *removing colors*.
  - When light is reflected (diffusively) from a surface or is transmitted through a partially transparent medium (as when photographic filters are used), centain colors are absorbed by the material and thus removed.
  - A *substractive color system* expresses a color, D by means of a 3-tuple, but each of the three values specifeies how much of a certain color (the complement of the corresponding primary) to *remove from white* in order to produce D.
  - The most common substractive system, the *CMY system*, *uses the subtractive primaries cyan, magenta, and yellow*.
  - Assumption: ink printed on pure white paper.
  - CMY = White - RGB.
    * C = 1 - R, M = 1- G, Y = 1 - B.
  - CMYK from CMY:
    * K means "key" which represents black
    * K = min(C, M, Y)
    * C = C - K, M = M - K, Y = Y - K.

- Hue - light mixing
  - This is what we are doing on a ocmputer monitor - starting as black and adding light.
  - An *additive color system* expresses a color, D, as the sum of certain amounts of primaries, usually red, green, and blue: $D = (r, g, b)$.

  - An additive system can use any three primaries, but because red, green, and blue are situated for apart in the CIE char, they provide a large gamut.
  - Colors are often described by comparing them with a standard color samples or lights and finding the closest mathc.
    * The observer adjusts the intensities (a, b, and c) of the test lights until the test color

$$T(\lambda) = aA(\lambda) + bB(\lambda) + cC(\lambda)$$

      is indistinguishable from the sample color $S(\lambda)$.
    * The two spectra, $S(\lambda)$ and $T(\lambda)$, may be quite different.

- Vector Algebra of Colors
  - Human color perception is three dimensional: any color C can be defined as the superposition of three primary colors, say R, G, and B: $C = rR + gG + bB$.
    * r, g, and b are scalars describing the amounts of each of the primaries contained in C.
  - But $C = rR + gG + bB$ works with any choice of primaries as long as one of them is not just a combination of the other two.
  - Given a set of three primary colors, R, G, and B, any other color, $C = rR + gG + bB$, can be represented in three-dimensinal space by the point (r, g, b).
  - $(0, 1, 0)$ will be perceived as a pure green of unit brightness and $(0.2, 0.3, 0)$ will be perceived as a yellow.
  - If we dboule each component, we will obtain a color that is twice as bright but appears as the same color.

- Color Spaces
  - The International Commission on Illumination (CIE) developed a different specification that CAN be added to form all visible colors.
    * Precise, standard, but not the most natural.
  - The CIE defined three special supersaturated primaries, X, Y, and Z.
  - They don't correspond to real colors, but they do have the property that *all* real colors can be rerpresented as positive combinations of them.
  - The diagram displays the horseshoe-shaped locus of all pure spectral colors, labelled according to wavelength.
  - Inside the horseshoe lie all other visible colors.
  - Points outside the horseshoe region do not correspond to visible light.
  - When two colors are added and their sum is white, we say the colors are complementary.
  - e (blue-green) and f (orange-pink) are complementary colors because proper amounts of them added together form white, w.
  - Other complementary colors:
    * red - cyan
    * green - magenta
    * blue - yellow
  - CIE Gamut Display: what you can usually actually see on a computer monitor.
  - Color Space Conversions
    * A color specified in CIE coordinates (x, y, z) can be transferred into (r, g, b) space, and vice versa.
    * Because the R, G, B primaries are linear combinations of the X, Y, and Z CIE primaries, a linear transformation works.
    * The mapping depends on the definition of the primaries R, G, and B and on the definition of white.
    * The transformation is shown below. Converting from (x, y, z) to (R, G, B) uses the inverse matrix.

$$\begin{bmatrix} r & g & b \end{bmatrix} = \begin{bmatrix} x & y & z \end{bmatrix} \begin{bmatrix} 2.739 & -1.110 & 0.138 \\ -1.145 & 2.029 & -0.333 \\ -0.424 & 0.033 & 1.105 \end{bmatrix}$$

* Indexed Color and the LUT
  - The full range of values that can be viewed on a typical display is about $32K$ different colors. (16 bits)
  - Dealing with all these different colors and transferring the data representing them can be costly.
  - So we often use something called a lookup table.
  - Saves memory, storage space, transmission time.
  - Some systems use a color lookup table (or LUT), which offers a *programmable* association between pixel value and final color.
  - E.g., color depth is six, but the six bits stored in each pixel are used as an index into a table of 64 values, say `LUT[0]`, `LUT[1]`, …, `LUT[63]`.
  - To make a particular pixel this color, say the pixel at location $(x, y) = (479, 532)$, the value 39 is stored in the frame buffer.
  - Each time the frame buffer is "scanned out" to the display, this pixel is read as value 39, so instead uses the value stored in `LUT[39]`.

2.4.20 Blending and Textures

- Alpha Channel and Image Blending
  - Blending allows you to draw a partially transparent image over another image.
  - We use the alpha value, the fourth component in specifying colors.
    * Alpha may have values in the range $[0, 255]$.
    * 0 represents total transparency and 255 represents total opacity.
    * It is usually used as $\alpha/255$, to give a value in $[0.0, 1.0]$.
  - Code: `struct RGBA { public: unsigned char r, g, b, a; };`
  - To blend 2 images, let α be the alpha value in A and use $B = \alpha A + (1 - \alpha)B$ separately for each color (R, G, B).
- Color for Blending
  - *RGBpixmap* must be extended to *RGBApixmap*.
  - Then use `B.draw(); A.blend()`.

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glEnable(GL_BLEND);
draw();  // draw this pixmap blended with the destination
```

- Tools for Setting and Using the Alpha Channel
  - `glBlendFunc` may have arguments other than `GL_SRC_ALPHA` and `GL_ONE_MINUS_SRC_ALPHA`
  - `DST` (destination) can replace `SRC` (source)
  - `GL_ZERO` and `GL_ONE` may also be used
  - `glBlendFunc(GL_DST_COLOR, GL_ZERO);`
  - multiplies each color component of a source pixel by the level of the corresponding component in the destination.
  - `glColor4f(r, g, b, a)`; `a` ranges from 0.0 (total transparency) to 1.0 (total opacity)
    * Fourth value `a` assigns an alpha value to all subsequently defined vertices.
  - Quadruples `refl = [r, g, b, a]` may be used in `glMaterialfv(GL_FRONT, GL_DIFFUSE, refl)` to make materials partially transparent.
- Logical Combinations of Pixmaps

  - Logical operations on pixmaps are used to combine pixmaps in various ways.
  - `glEnable(GL_COLOR_LOGIC_OP);`
  - `glLogic(operator);`

- OpenGL Logical Operators

Parameter Value	Value Written to Destination
GL_CLEAR	0
GL_COPY	S
GL_NOOP	D
GL_SET	1
GL_COPY_INVERTED	NOT S
GL_INVERT	NOT D
GL_AND_REVERSE	S OR NOT D
GL_AND	S AND D
GL_OR	S OR D
GL_NAND	NOT(S AND D)
GL_NOR	NOT(S OR D)
GL_XOR	S XOR D
GL_EQUIV	NOT(S XOR D)
GL_AND_INVERTED	NOT S AND D
GL_OR_INVERTED	NOT S OR D

- Adding Texture to Faces
  - Makes surfaces look more realistic: e.g., brick or wood texture.
  - Texture is painted on or wrapped around surfaces.
  - Texture is a function `texture(s, t)` which sets a color or intensity value for each value of `s` and `t` between 0.0 and 1.0.
  - The value of the function `texture` is between 0.0 and 1.0 (dark and light).
  - Uses of Textures
    - The value `texture(s, t)` can be used in a variety of ways:
      - color of the face itself as if the face were glowing;
      - ambient, diffuse, or specular reflection coefficients;
      - alter the normal vector to the surface to given the object a bumpy appearance.
  - Textures
    - Textures are often formed from bitmap representations of images.
    - Such a texture consists of a 2D array, say `txtr[r][c]`, of color values (often called `texels`).
  - Image Textures
    - Image textures usually have integer coordinates.
    - They are transformed into values between 0 and 1 by dividing by the size of the image: $s = x/x_{\max}$ and $t = y/y_{\max}$.
  - Procedural Textures
    - To produce the procedural texture shown, we implement the following function:

```
float generateTexture(float s, float t) {
    float r = sqrt((s - 0.5) * (s - 0.5) + (t - 0.5) * (t - 0.5));
    if (r > 0.3)
        return 1 - r / 0.3;
    else
        return 0.2;  // 0.2 is the background
}
```

– Mapping Screen Pixel to Texel

  * The rendering process actually goes the other way: for each pixel at (s_x, s_y) there is a sequence of questions:
    (1) What is the closest surface seen at (s_x, s_y)?
    (2) To what point (x, y, z) on this surface does (s_x, s_y) correpsond?
    (3) To which texture coordinate pair (s, t) does this point (x, y, z)$ correspond?
  * So we need inverse transformation:

$$(s, t) = T_{\text{tw}}^{-1}(T_{\text{ws}}^{-1}(s_x, s_y)),$$

    that reports (s, t) coordinates given pixel coordinates.

– Pasting Texture onto a Flat Surface

```
glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0);
    glVertex3f(1.0, 2.5, 1.5);
    glTexCoord2f(0.0, 0.6);
    glVertex3f(1.0, 3.7, 1.5);
    glTexCoord2f(0.8, 0.6);
    glVertex3f(2.0, 3.7, 1.5);
    glTexCoord2f(0.8, 0.0);
    glVertex3f(2.0, 2.5, 1.5);
glEnd();
```

  * The figure shows a common case where the four corners of a texture rectangle are associated with the four corners of a face in the 3D scene.

– Tiling a Texture

  * The figure shows the use of texture coordinates that *tile* the texture, making it repeat.

– Adding Texture Coordinates to Mesh Objects

  * There are several different ways to treat texture for an object.
    (1) Mesh is small number of flat faces, so the data associated with each face would be
      · the number of vertices in the face;
      · the index of the normal vector to the face;
      · a list of indices of the vertices;
      · a list of indices of the texture coordinates;
    (2) The mesh represents a smooth underlying object, and a single texture is to be wrapped around it (or a portion of it). The data associated with each face would then be:
      · the number of vertices in the face;
      · a list of indices of the vertices;

    A single index into the vertex, normal, and texture lists is used for each vertex.

- Rendering the Texture
  - Done scan line by scan line.
  - Affine and projective transformations preserve straightness, so line L_e in eye space projects to line L_s in screen space, and similarly the texels we wish to draw on line L_s lie along the line L_t in texture space that maps to L_e.
  - Complication: calculating s and t by simple linear interpolation will not work.
- Applying Textures
  - In OpenGL, use `glTexEnvf(GL_TEXTURE_ENV, GL_TEX_ENV_MODE, GL_REPLACE);`
  - The intensity I is replaced by the value of the texture. (In color, the replacement is for each of the R, G, B components.), $I = \text{texture}(s, t)$.
  - Modulating Reflection Coefficient
    - Color of an objecct is color of its diffuse light component; vary diffuse refelction coefficient.
    - $I = \text{texture}(s, t) \cdot [I_a\rho_a + I_d\rho_d \times \text{lambert}] + I_{sp}\rho_s \times \text{phong}$.
    - The specular component is the color of the light, not the object.
  - In OpenGL, use `glTexEnvf(GL_TEXTURE_ENV, GL_TEX_ENV_MODE, GL_MODULATE);`
  - Example
    - OpenGL can supply unique names: if we need six unique names we can build an array to hold them: `GLuint name[6];` and then call `glGenTextures(6, name);` OpenGL places six unique integers in `name[0]`, `name[1]`, …, `name[5]` and we subsequently refer to the i^{th} texture using `name[i]`.

```
1  glGenTextures(6, textureName);
2
3  setTexture(GLuint textureName) {
4      glBindTexture(GL_TEXTURE_2D, textureName);
5      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
6      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
7      glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, nCols, nRows, 0, GL_RGB,
           GL_UNSIGNED_BYTE, arrayStoringTexValues);
8  }
```

    - Texture mapping must also be enabled with `glEnable(GL_TEXTURE_2D);`
    - The call `glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST)` is used to request that OpenGL render the texture properly.

2.5 Raycasting and Pixmaps

2.5.1 Goals of Ray Casting/Tracing

- Highly realistic images
- Work with solid 3D texture and bitmapped images.
- Add surface texture to obejcts in the scene
- Study the creation of soli (3D) textures, such as wood grain and marble, in objects.
- To create a much richer class of object shapes based on constructive solid geometry (CSG) and to learn how to ray trace them.

2.5.2 Ray Casting

- Ray Casting:
  - the frame buffer as an array of pixels positioned in space.

  - with the eye looking through it into the scene.

- For each pixel in the frame buffer we ask:
  - What does the eye see through this pixel?
    * We think of a ray of light arriving at the eye through this pixel from some point P in the scene.
    * The color of the pixel is set to that of the light that emanates along the ray from point P in the scene.

- Its path is tested against each object in the scene.
- A parameter t is used to associate the ray's position so…
  - "the ray is here now" or "the ray reaches this point first"

2.5.3 Geometry of Ray Tracing

- In order to trace rays, we need a convenient representation for the ray.
- A parametric representation will be the most serviceable.
- That passes through a particular pixel.

2.5.4 Ray Tracing

- Using a description of light sources in the scene.
- A shading model is applied to the first hit point, and the light components are computed.
- The resulting color is then displayed at the pixel.
- Ray tracing can also work with a richer class of geometric objects than polygon meshes.
- Solid objects are constructed out of various geometric primitives such as spheres, cones, and cylinders.
  - Each shape is represented exactly through a mathematical expression; it is not approximated as a mesh.

- What do you need?
  - Rectangular map of pixels by rows and columns.
    * Block size is set to 4.
    * Each block consists of 16 pixels.

```
glViewport(0, 0, nCols, nRows)
glColor3f(red, green, blue);
glRecti(col, row, col + blockSize, row + blockSize);
// or use glDrawPixels(nCols, nRows, GL_RGB, GL_UNSIGNED_BYTE, pixel);
```

  - A vector or "ray" from the eye through each pixel.
  - Knowledge of Objects in the scene.
  - Description of light sources in the scene.
  - A shading model.

2.5.5 Rectagular Map of Pixels by Rows and Columns

- Scan Conversion (Rasterization)
  - Determines individual pixel values.
  - Rasterization in the GL graphics pipeline.

- Manipulating Pixmaps

  - Pixmaps may be stored in regular memory or in the frame buffer (off-screen or on-screen).
  - Rendering operations that draw into the frame buffer change the paritcular pixmap that is visible on the display.

- Pixmap Operations: Copy
  - `glReadPixels(x, y, nCols, nRows, GL_RGB, GL_UNSIGNED_BYTE, pixel)` reads a portion of the frame buffer into memory.
  - `glCopyPixels(x, y, w, h, GL_COLOR)` copies a region in one part of the frame buffer into another region of the frame buffer.
  - `glDrawPixels(nCols, nRows, GL_RGB, GL_UNSIGNED_BYTE, pixel)` draws a given pixmap into the frame buffer.
  - Scaling Pixmaps `glPixelZoom(float sx, float sy)`
    * Sets scale factors in x and y.
    * Any floating point values are allowed for `sx` and `sy`, even negeative ones. The default values are 1.0.
    * The scaling takes place about the current raster position, pt.
    * The pixel in row r and column c of the pixmap will be drawn as a rectangle of with `sx` and height `sy` screen pixels with lower left corner at screen pixel `(pt.x + sx * r, pt.y + sy * c)`.
    * Scale by s: output has s times aas many pixels in x and in y as input; if s is an integer, scale with pixel replication to enlarge.
  - We may rotate a pixmap.
  - We may compare two pixmaps, e.g., to detect tumors.
  - We can describe regions within a pixmap as circles, squares, and so on.
  - We may fill the interior of a region with a color.
  - Scaling Images
    * To reduce by, for example, 1/3: take every third row and column of the pixmap. This method is usually not satisfactory. Instead, we should averge the values of the 9 pixels in each non-overlapping 3×3 array, and use that value for the pixel.
  - Rotating Images
    * Pixmaps may be roated by any amount.
    * Rotating through 90°, 180°, 270° is simple: create a new pixmap and copy pixels from the original to the appropriate spot in the new pixmap.
    * Other rotations are diffcult. Simplest approach: pixel in transformed image is set to color of pixel it was transformed from in original.

- Pixmap Data Types
  - Bitmap: $pixel = bit$, 0 or 1 (black or white).
  - Gray-scale bitmap: $pixel = byte$, representing gray levels from 0 (black) through 255 (white).
  - Pixel contains an index into a lookup table (LUT); usually index is a byte.
  - RGB pixmap contains 3 bytes, one each for red, green, and blue.
  - RGBA pixmap contains 4 bytes, one each for red, green, blue and alpha (transparency).
  - Code: start by defining a color:

```
struct RGB { public: unsigned char r, g, b; };  // holds one color triple
```

  - OpenGL represents a pixmap as an array `pixel` of pixel values stored row by row from bottom to top and across each row from left to right.

- Examples
  - Window scrolling: blank line replaces bottom, moving all lines up one.
  - Redrawing screen after menu use.

2.5.6 A Vector or "Ray" from the Eye Through Each Pixel

- We use the same camera as in Chapter 7.
- Its eye is at point `eye`.
- The axes of the camera are along the vector **u**, **v** and **n**.
- The near plane lies at distance N in front of the eye, and the frame buffer lies in the near plane.
- Parallel vs. Perspective Projection
  - The camera shape has a viewangle of θ, and the window in the near plane has aspect ratio *aspect*.
  - It extends from $-H$ to H in the **v**-direction.
  - From $-W$ to W in the **u**-direction.
  - $H = N\tan(\theta/2)$.
  - $W = H \times aspectRatio$.
  - Find how far this point from the eye:
    1. Start at `eye`.
    2. Determine how far you msut go in each of the directios **u**, **v**, and **n** to reach the pixel corner. You must go:
       a) Distance N in the negative **n**-direction.
       b) Distance u_c along **u**.
       c) Distance v_r along **v**.
  - The rc-th pixel appears on the **u**, **v** plane at u_c, v_r given by

$$u_c = -W + W\frac{2c}{nCols}, \quad \text{for } c = 0, 1, \ldots, nCols - 1$$
$$v_r = -H + H\frac{2r}{nRows}, \quad \text{for } r = 0, 1, \ldots, nRows - 1$$

  - We will need to express where it lies in 3D: the actual point on the near plane.
    * 3D point (in world coordinate system) is defined by

$$eye - N\mathbf{n} + u_c\mathbf{u} + v_r\mathbf{v}.$$

      where N is the distance that near plane is from the eye.
  - Example: The preview has been drawn, and the lower half of the screen has been ray traced. The important point here is that ray tracing *exactly* overlays the preview scene.

- Initial `Ray` Class

```
class Ray {
public:
    Point start;
    Vector dir;
    void setStart(Point& p) {start.x = p.x; start.y = p.y; start.y = p.y;}
    void setDir(Vector& v) {dir.x = v.x; dir.y = v.y; dir.y = v.y;}
    // other fields and methods
};
```

- `Raytrace()` Skeleton

```
void Camera::raytrace(Scene& scn, int blockSize)
{
    Ray theRay;
```

```
    theRay.setStart(eye);
    // set up OpenGL for simple 2D drawing
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glMatrrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0, nCols, 0, nRows);  // whole screen is the window
    glDisable(GL_LIGHTING);  // so glColor3f() works properly

    // begin raytracing
    for (int row = 0; row < nRows; row += blockSize) {
        for (int col = 0; col < nCols; col += blockSize) {
            // compute the ray's direction
            Vector dir = {...};
            theRay.setDir(dir);  // set the ray's direction
            Color3 clr = scn.shape(theRay);  // find the color
            glColor3f(red, green, blue);
            glRecti(col, row, col + blockSize, row + blockSize);

        }
    }

}
```

- Organizing a Ray Caster
  - The `blockSize` parameter determines the size of the block of pixels being drawn at each step.
  - Displaying pixel blocks is a time-saver for the viewer during the development of a ray tracer.
  - The figure (a) shows how this works for a simple example of a display that has 16 rows and 32 columns of actual pixels, and `blockSize` is set to 4. Each block consists of 16 pixels.
  - The color of the ray through the corner of the block is determined, and the entire block (all 16 pixels) is set the this uniform color.
  - The image would appear as a raster of four by eight blocks, but it would draw very quickly.
  - The figure (b) shows a simple scene ray traced with a block size of 4, 2, and 1.

- How do we compute the ray? Here is the pseudocode for ray tracing.

```
<define objects, light sources, and camera in the scene>
for (int r = 0; r < nRows; ++r) {
    for (int c = 0; c < nCols; ++c) {
        <1. Build the rc-th ray>;
        <2. Find all intersections of rc-th ray with objects>
        <3. Find intersection that lies closest to and in front of the eye>
        <4. Compute the hit point where the ray hits this object, and the normal vector at that point>
        <5. Find the color of the light returning to the eye along the ray from the point of intersection>
        <6. Place the color in the rc-th pixel>
    }
}
```

- Geometry of Ray Tracing/Casting
  - We parameterize the position of a ray along its path using t, which is conveniently taken to be time.
  - As t increases, the ray moves further and further along its path.

- By design, it starts at the eye at $t = 0$ and reaches the lower left corner of the rc-th pixel at $t = 1$: $r(t) = eye \cdot (1 - t) = pixelcorner \cdot t$.
- Substituting for $pixelcorner$, we get the detailed parametric form for the rc-th ray:

$$r(t) = eye \cdot (1 - t) + (eye - N\mathbf{n} + u_c\mathbf{u} + v_r\mathbf{v}) \cdot t.$$

- The *starting point* and *direction* of this ray are

$$r(t) = eye + \mathbf{dir}_{rc} \cdot t = eye + \left[-N\mathbf{n} + W\left(\frac{2c}{nCols} - 1\right)\mathbf{u} + H\left(\frac{2r}{nRows} - 1\right)\mathbf{v}\right] \cdot t.$$

- Note that as t increases from 0, the ray point moves further and further from the eye.
- If the ray strikes two objects in its path, say at time t_a and t_b, the object lying closer to the eye will be the one hit at the lower value of t.
- Representing Lines (parametric form): we have 2 points, B and C, on the line. $P(x, y)$ is on the line when $P = C + \mathbf{b}t$, where $\mathbf{b} = B - C$.
  - $0 \le t \le 1$: line segment.
  - $-\infty \le t \le \infty$: line
  - $-\infty \le t \le 0$ or $0 \le t \le \infty$: ray.

2.5.7 Knowledge of Objects in the Scene

- The object with the smallest hit time is identified.
- The hit spot, P_{hit}, is then easily found from the ray itself by evaluating the ray at the hit time, t_{hit} : $P_{\text{hit}} = eye + \mathbf{dir}_{rc}t_{\text{hit}}\{\text{hit spot}\}$.
- How do we define and store the objects?
  - Define: As a tranformation.
  - Store: As a linked list of matrices.
- Implicit form for generic sphere, cyliner, plane
  - The **generic sphere** is a sphere of radius 1 centered at the origin, which has the implicit form $F(x, y, z) = x^2 + y^2 + z^2 - 1$.
  - The **generic cyliner**, which has radius 1, is aligned along the z-axis with an axis of length 1, and has the implicit form $F(x, y, z) = x^2 + y^2 - 1 = 0$ for $0 < z < 1$.
  - The **generic plane** is the xy-plane, with $z = 0$, so its implicit form is $F(x, y, z) = z$.
- Intersection of a Ray with an Object
  - How do we find the intersection of a ray with a shape whose implicit form is defined as $F(P)$? Suppose the ray has starting point S and direction c. The ray is therefore given by $r(t) = S + ct$. All points on the surface of the shape satify $F(P) = 0$, a condition for $r(t)$ to coincide with a point on the surface is therefore $F(r(t)) = 0$. This will occur at the hit time t_{hit}. To find t_{hit}, we must therefore solve the equation $F(S + ct_{\text{hit}}) = 0$.
  - The intersection of a ray with generic plane is

$$F(x, y, z) = z = 0 \implies F(r(t)) = F(S+\mathbf{c}t) = (S+\mathbf{c}t)_z = 0 \implies S_z+\mathbf{c}_z t_{\text{hit}} = 0 \implies t_{\text{hit}} = -S_z/\mathbf{c}_z.$$

    - If $\mathbf{c}_z = 0$, the ray is moving parallel tto the plane, and there is no intersection.
    - Otherwise, the ray hits the plane at the point $P_{\text{hit}} = S - \mathbf{c}(S_z/\mathbf{c}_z)$.
  - The intersection of a ray with generic sphere is

$$\begin{aligned} &F(x, y, z) = x^2 + y^2 + z^2 - 1 = 0 \\ &\implies F(r(t)) = F(S + \mathbf{c}t) = |S + \mathbf{c}t|^2 - 1 = |\mathbf{c}|^2t^2 + 2(S \cdot \mathbf{c})t + |S|^2 - 1 = 0 \\ &\implies t_{\text{hit}} = \min\left\{\frac{-2(S \cdot \mathbf{c}) \pm \sqrt{4(S \cdot \mathbf{c})^2 - 4|\mathbf{c}|^2(|S|^2 - 1)}}{2|\mathbf{c}|^2}\right\}. \end{aligned}$$

- `Intersection` Class
  - The routine `getFirstHit()` finds the object hit first by the ray and returns the information in the intersection record `best`.
  - We implement intersection records using the class `Intersection` which is given in pseudocode by:

```
class Intersection {
public:
    int numHits;  // number of hits at positive hit times
    HitInfo hit[8];  // list of hits - may need more than 8 later

}
```

- Intersections of Rays and Objects
  - We are particularly interested in the fist hit of the ray with an object.
  - If `inter` is an intersection record and `inter.numHits` is greater than 0, information concerning the first hit is store in `inter.hit[0]`.
  - Normally the eye is outside of all objects and a ray hits just twice: once upon entering the object and once upon exiting it.
  - In such cases `inter.numHits` is 2, `inter.hit[0]` describes where the ray enters the object, and `inter.hit[1]` describes where it exits.
  - `getFirstHit()` code

```
void getFirstHit(Ray &ray, Intersection &best)
{
    Intersection inter;  // make intersection record
    best.numHits = 0;
    for (GeomObj* pObj = obj; pObj != NULL; pObj = pObj->next)
    {
        // test each object in the scene
        if (!pObj->hit(ray.inter))  // does the ray hit pObj?
            continue;               // miss: test the next object
        if (best.numHits = 0 || inter.hit[0].hitTime < best.hit[0].hitTime)
            best.set(inter);        // copy inter into best
    }
}
```

2.5.8 Intro to Phong Shading

- What happends after ray cast?
  - Ray reflections
  - Ray refractions
  - Ray depth
  - Ray Transparencies
  - Shadow Rays
  - Basics of Textures
- Reflections and Transparency
  - One of the great strenghts of th ray tracing method is the ease with which it can handle:
    - reflection of light

    - refraction of light
  - There can be multiple refelctions.
  - Light bounces off several shiny surfaces before reaching the eye, or elaborate combinations of refraction and reflection.
  - Each of these processes requires the spawning and tracing of additional rays.
  - When the surface is mirror-like or transparent, or both, the light I that reaches the eye may have five components:

$$I = I_{\text{ambient}} + I_{\text{diffuse}} + I_{\text{specular}} + I_{\text{reflection}} + I_{\text{transparency}}.$$

  - $I_{\text{reflection}}$ is the reflected light component, arising from the light, I_R, that is incident at P_h along direction $-\mathbf{r}$.
  - $\mathbf{r} = \mathbf{dir} - 2(\mathbf{dir} \cdot \mathbf{n})\mathbf{n}$, where $\mathbf{n}$ and $\mathbf{dir}$ are normalized.
  - To find I_R, we spawn a secondary ray from P_h in the direction $\mathbf{r}$, find the first object it hits, and compute the 5 light components.
  - $I_{\text{transparency}}$ is the transmitted light component, arising from the light, I_T, that is transmitted through the transparent material to P_h along direction $-\mathbf{t}$.
  - I_T is found by casting a ray in direction $\mathbf{t}$ and seeing what surface is hit first, then computing the light contribution there, etc.
  - The number of contributions of light grows at each contract point.
  - The final light that reaches the eye, I is a combination of a large number of contributions, consisting of:
    - reflected light
    - refracted light from various points in the scene in addition to a number of ambient, diffuse, and specular components.
  - I_R and I_T each arise from their own five components: ambient, diffuse, and so on.
  - I_R is the light that would be seen by an eye at P_h along a ray from P' to P_h.
  - To incorporate these visual effects, extend the shade function (or "Phong") so that it can call itself recursively.
  - Under the right conditions, `shade()` calls itself twice to accumulate reflected and transmitted light contributions.
  - If the hit object is shiny enough, a reflected ray is spawned, and `shade()` is used to compute how much light comes back along the reflection direction.
  - The amount of light `shade()` finds is tempered by the reflection coefficient, shininess, of the hit object.
  - This reflection coefficient is stored in one of the fields of the object.
  - If the hit object is transparent enough, a transmitted ray is spawned, and `shade()` is used to compute how much light comes back along the transmitted direction.
  - The amount of light found is scaled by the transmission coefficient, transparency, of the hit object.
  - This coefficient is stored with the object, and it is used for the "if transparent enough" test, as in `if (transparency > 0.5)`.
  - There is the possibility that rays would keep spawning new reflected or transmitted rays forever.
  - We include a `maxRecursionLevel` (as `maxRecursionDepth`) to ensure the recursion stops.
  - Usually a maximum recursion depth of 4 or 5 gives very realistic images.

- Ray tracing Objects with Solid Textures
  - There are various ways that the texture can alter the light coming from a surface point:
    - The light can be set equal to `texture()` itself, as if the object were glowing with that color.
    - The texture `tex` can *modulate* the ambient and diffuse reflection coefficients, so that

$$I = tex(x, y, z)[I_a m_a + I_d m_d(\mathbf{N} \cdot \mathbf{L}) + I_s m_s(\mathbf{H} \cdot \mathbf{N})^e],$$

      where $tex(x, y, z)$ is evaluated at the hit point (x, y, z) of the ray (most common use of texture).

- Adding Shadows
  - The light intesities we have calculated up to now have assumed that the hit point, P_h, of the ray with the first object hit is in fact bathed in light from the various light sources.
  - But this is not the case if some other object happens to lie between P_h and a light source.
  - In that case, P_h is in shadow with respect to that light source, and both the diffuse and specular contributions are absent.
  - This leaves only the ambient light component.
  - Adding Light Contributions
    - The emissive and ambient components are found.
    - For each light source, the *diffuse and specular* contributions arising from that light source found and added into the color.
    - If the light source is shadowed by some object at the hit point, there is no contribution from that source.
  - In order to compute shadows accurately, we need to know when a hit point is in shadow with respect to a light source.
  - So we need a routine, `isInShadow()`, that returns true if any part of *any* object lies between the hit point and a given source, and false otherwise.
  - To do this, we spawn a new ray, often called a **shadow-feeler**, that emanates from P_h at $t = 0$ and reaches L at $t = 1$.
  - The shadow-feeler thus has the parametric representation

$$P_h + (L - P_h)t.$$

  - To see if it hits anything, the entire object list is scanned, and each object is tested for an intersection with this ray.
  - If any intersection is found to lie between $t = 0$ and $t = 1$, `isInShadow()` returns true.
  - Including self-shadowing: if the shadow-feeler really starts at P_h, then there is always an intersection between the feeler ray and the object itself, at $t = 0$. Thus `isInShadow()` would always return `true`, which is clearly wrong.
  - An adjusted shadow feeler is used, $P_h - \varepsilon\mathbf{dir}$, where ε is a small positive number.

- Project 3B
  - If there is no hit, multiply the ambient color by background color.
  - If the ray hits the object, we also need to take the global ambient into consideration,

$$I_a = m_a \times I_{\text{global } a} + \sum_{i=1}^{n} m_a * I_{ia}$$

    where n is the number of light sources that can reach the hit point.
  - Diffuse light

$$I_d = \sum_{i=1}^{n} m_d \times I_{id} \times \max\{0, \mathbf{L} \cdot \mathbf{N}\},$$

    where $\mathbf{L} = \frac{\mathbf{lightPos - hitPoint}}{\|\mathbf{lightPos - hitPoint}\|}$, $\mathbf{N}$ is the normal vector on the surface which can be obtained by $\mathbf{N} = \frac{\mathbf{hitPoint - center}}{\|\mathbf{hitPoint - center}\|}$, **center** is the coordinates for center of the sphere. Here $\mathbf{L}$ and $\mathbf{N}$ should be normalized before doing the dot product.
  - Specular Light

$$I_s = \sum_{i=1}^{n} m_s \times I_{is} \times [\max\{0, \mathbf{H} \cdot \mathbf{N}\}]^s,$$

    where $\mathbf{H} = \frac{\mathbf{V+L}}{\|\mathbf{V+L}\|}$ (both $\mathbf{V}$ and $\mathbf{L}$ should be normalized) is the half-way vector, $\mathbf{V} = \frac{\mathbf{eye - hitPoint}}{\|\mathbf{eye - hitPoint}\|}$, s is the shininess of the object.

  - Spot Light

$$I_i = I_i' \times (\cos\beta)^{\varepsilon} \times \text{attenuation},$$

    where I_i stands for to the local ambient, diffuse and specular illumination for light source i, ε is the expoent, and attenuation $= \frac{1}{k_c + k_l d + k_q d^2}$, d is the distance between the hit point and spot light.

- Project 3C
  - Reflections and Transparency
    * One of the great
    * When the surface is mirror-like or transparent, or both, the light I that reaches the eye may have five components

$$I = I_{\text{amb}} + I_{\text{diff}} + I_{\text{spec}} + I_{\text{ref}} + I_{\text{tran}}.$$

    * The reflected light compent

$$\mathbf{r} = \mathbf{dir} - 2(\mathbf{dir} \cdot \mathbf{n})\mathbf{n}$$

      where **dir** is the view vector, **n** is the normal vector.
    * I_{tran} is the transmitted light component, arising from the light, I_T, that is transmitted through the the transparent materials to
    * The tree of light rays; local components are not shown. I is the sum of three components:
      · the reflected component R_1,
      · the transmitted component T_1 from the refraction,
      · and the local compoenent L_1.
  - Depth
    * Thre is the possibity that rays would keep spawning new refelcted or transmitted rays forever
    * We include

3 Quiz:

- The following [points, lines, polylines, text, filled regions, and raster images] are *Graphics Primitives.*
- The following [GL_LINES, GL_TRIANGLES, GL_POINTS, GL_QUAD_STRIP] are *OpenGL Graphics Primitives.*
- Provide one code example of how to register a callback function. You may choose one of the following: a keyboard event or any type of mouse event.
- World coordinate system: can be negative, no bound.
- World coordinate is defined in the 3D system and screen coordinate is defined in 2D system, that's why we need to have relative …
- OpenGL Tutorial