Introduction to OpenGL

Prof. George Wolberg
Dept. of Computer Science
City College of New York

The Programmer's Interface

- Programmer sees the graphics system through an interface: the Application Programmer Interface (API)

API Contents

- Functions that specify what we need to form an image
  - Objects
  - Viewer
  - Light Source(s)
  - Materials
- Other information
  - Input from devices such as mouse and keyboard
  - Capabilities of system

Object Specification

- Most APIs support a limited set of primitives including
  - Points (1D object)
  - Line segments (2D objects)
  - Polygons (3D objects)
  - Some curves and surfaces
    - Quadratics
    - Parametric polynomial
- All are defined through locations in space or vertices

Example

```c
glBegin(GL_POLYGON)
glVertex3f(0.0, 0.0, 0.0);
glVertex3f(0.0, 1.0, 0.0);
glVertex3f(0.0, 0.0, 1.0);
glEnd();
```

Camera Specification

- Six degrees of freedom
  - Position of center of lens
  - Orientation
- Lens
- Film size
- Orientation of film plane


Lights and Materials

- Types of lights
  - Point sources vs distributed sources
  - Spot lights
  - Near and far sources
  - Color properties
- Material properties
  - Absorption: color properties
  - Scattering
    - Diffuse
    - Specular

Following the Pipeline: Transformations

- Much of the work in the pipeline is in converting object representations from one coordinate system to another
  - World coordinates
  - Camera coordinates
  - Screen coordinates
- Every change of coordinates is equivalent to a matrix transformation

Clipping

- Just as a real camera cannot "see" the whole world, the virtual camera can only see part of the world space
  - Objects that are not within this volume are said to be clipped out of the scene

Projection

- Must carry out the process that combines the 3D viewer with the 3D objects to produce the 2D image
  - Perspective projections: all projectors meet at the center of projection
  - Parallel projection: projectors are parallel, center of projection is replaced by a direction of projection

Rasterization

- If an object is visible in the image, the appropriate pixels in the frame buffer must be assigned colors
  - Vertices assembled into objects
  - Effects of lights and materials must be determined
  - Polygons filled with interior colors/shades
  - Must have also determined which objects are in front (hidden surface removal)

Programming with OpenGL Part 1: Background

Prof. George Wolberg
Dept. of Computer Science
City College of New York
Objectives

• Development of the OpenGL API
• OpenGL Architecture
  - OpenGL as a state machine
• Functions
  - Types
  - Formats
• Simple program

Early History of APIs

• IFIPS (1973) formed two committees to come up with a standard graphics API
  - Graphical Kernel System (GKS)
    - 2D but contained good workstation model
    - Core
      - Both 2D and 3D
    - GKS adopted as IS0 and later ANSI standard (1980s)
  - GKS not easily extended to 3D (GKS-3D)
  - Far behind hardware development

PHIGS and X

• Programmers Hierarchical Graphics System (PHIGS)
  - Arose from CAD community
  - Database model with retained graphics (structures)
• X Window System
  - DEC/MIT effort
  - Client-server architecture with graphics
• PEX combined the two
  - Not easy to use (all the defects of each)

SGI and GL

• Silicon Graphics (SGI) revolutionized the graphics workstation by implementing the pipeline in hardware (1982)
• To use the system, application programmers used a library called GL
• With GL, it was relatively simple to program three dimensional interactive applications

OpenGL

• The success of GL lead to OpenGL in 1992, a platform-independent API that was
  - Easy to use
  - Close enough to the hardware to get excellent performance
  - Focused on rendering
  - Omitted windowing and input to avoid window system dependencies

OpenGL Evolution

• Controlled by an Architectural Review Board (ARB)
  - Members include SGI, Microsoft, Nvidia, HP, 3DLabs,IBM,…….
  - Relatively stable (present version 1.4)
    - Evolution reflects new hardware capabilities
      - 3D texture mapping and texture objects
      - Vertex programs
  - Allows for platform specific features through extensions
  - See www.opengl.org for up-to-date info
**OpenGL Libraries**

- **OpenGL core library**
  - OpenGL32 on Windows
  - GL on most Unix/Linux systems
- **OpenGL Utility Library (GLU)**
  - Provides functionality in OpenGL core but avoids having to rewrite code
- **Links with window system**
  - GLX for X window systems
  - WGL for Windows
  - AGL for Macintosh

**GLUT**

- **OpenGL Utility Toolkit (GLUT)**
  - Provides functionality common to all window systems
    - Open a window
    - Get input from mouse and keyboard
    - Menus
    - Event-driven
  - Code is portable but GLUT lacks the functionality of a good toolkit for a specific platform
    - Slide bars

**Software Organization**

```
application program
  OpenGL, Motif
gluit, or similar
  GLX, AGL
  or WGL
  X, Win32, Mac OS
  GL
software and/or hardware
```

**OpenGL Architecture**

```
CPU
  Immediate Mode
  Geometric pipeline
    Polygonal
    Operations &
    Primitive Assembly
  Per Vertex
  Display
  List
  Rasterization
  Frame Buffer
  Pixel Memory
  Textures
  Operations
```

**OpenGL Functions**

- **Primitives**
  - Points
  - Line Segments
  - Polygons
- **Attributes**
- **Transformations**
  - Viewing
  - Modeling
- **Control**
- **Input (GLUT)**

**OpenGL State**

- **OpenGL is a state machine**
- **OpenGL functions are of two types**
  - Primitive generating
    - Can cause output if primitive is visible
    - How vertices are processed and appearance of primitive are controlled by the state
  - State changing
    - Transformation functions
    - Attribute functions
Lack of Object Orientation

- OpenGL is not object oriented so that there are multiple functions for a given logical function, e.g., `glVertex3f`, `glVertex2i`, `glVertex3dv`, etc.
- Underlying storage mode is the same
- Easy to create overloaded functions in C++ but issue is efficiency

OpenGL function format

- Function name
  - `glVertex3f(x,y,z)`
    - `x,y,z` are floats
- `glVertex3fv(p)`
  - `p` is a pointer to an array

OpenGL #defines

- Most constants are defined in the include files `gl.h`, `glu.h`, and `glut.h`
  - Note `#include <glut.h>` should automatically include the others
  - Examples
    - `glBegin(GL_POLYGON)`
    - `glClear(GL_COLOR_BUFFER_BIT)`
- Include files also define OpenGL data types: `GLfloat`, `GLdouble`, etc.

A Simple Program

Generate a square on a solid background

`simple.c`

```c
#include <glut.h>
void mydisplay(){
  glClear(GL_COLOR_BUFFER_BIT);
  glBegin(GL_POLYGON);
  glVertex2f(-0.5,-0.5);
  glVertex2f(-0.5, 0.5);
  glVertex2f( 0.5, 0.5);
  glVertex2f( 0.5,-0.5);
  glEnd();
  glFlush();
}
int main(int argc, char** argv){
  glutCreateWindow("simple");
  glutDisplayFunc(mydisplay);
  glutMainLoop();
}
```

Event Loop

- Note that the program defines a display callback function named `mydisplay`
  - Every glut program must have a display callback
  - The display callback is executed whenever OpenGL decides the display must be refreshed, for example when the window is opened
  - The `main` function ends with the program entering an event loop
Defaults

- simple.c is too simple
- Makes heavy use of state variable default values for
  - Viewing
  - Colors
  - Window parameters
- Next version will make the defaults more explicit

Notes on compilation

- See website and ftp for examples
- Unix/Linux
  - Include files usually in …/include/GL
  - Compile with –lglut –lglu –lglt loader flags
  - May have to add –L flag for X libraries
  - Mesa implementation included with most linux distributions
  - Check web for latest versions of Mesa and glut

Compilation on Windows

- Visual C++
  - Get glut.h, glut32.lib and glut32.dll from web
  - Create a console application
  - Add opengl32.lib, glut32.lib, glut32.lib to project settings (under link tab)
- Borland C similar
- Cygwin (linux under Windows)
  - Can use gcc and similar makefile to linux
  - Use –lopengl32 –lglu –lglut flags

Programming with OpenGL
Part 2: Complete Programs

Prof. George Wolberg
Dept. of Computer Science
City College of New York

Objectives

- Refine the first program
  - Alter the default values
  - Introduce a standard program structure
- Simple viewing
  - Two-dimensional viewing as a special case of three-dimensional viewing
- Fundamental OpenGL primitives
- Attributes

Program Structure

- Most OpenGL programs have a similar structure that consists of the following functions
  - main():
    - defines the callback functions
    - opens one or more windows with the required properties
    - enters event loop (last executable statement)
  - init(): sets the state variables
    - viewing
    - Attributes
    - callbacks
    - Display function
    - Input and window functions
Simple.c revisited

- In this version, we will see the same output but have defined all the relevant state values through function calls with the default values
- In particular, we set
  - Colors
  - Viewing conditions
  - Window properties

main.c

```c
#include <GL/glut.h> // includes gl.h
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(0,0);
    glutCreateWindow("simple");
    glutDisplayFunc(mydisplay);
    init(); // define window properties
    glutMainLoop(); // display callback
}
```

includes

- gl.h

GLUT functions

- glutInit allows application to get command line arguments and initializes system
- glutInitDisplayMode requests properties of the window (the rendering context)
  - RGB color
  - Single buffering
  - Properties logically ORed together
- glutWindowSize in pixels
- glutWindowPosition from top-left corner of display
- glutCreateWindow create window with title "simple"
- glutDisplayFunc display callback
- glutMainLoop enter infinite event loop

init.c

```c
void init() {
    glClearColor (0.0, 0.0, 0.0, 1.0); // black clear color
    glColor3f(1.0, 1.0, 1.0); // opaque window
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0); // fill with white
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity();
}
```

Coordinate Systems

- The units of in glVertex are determined by the application and are called world or problem coordinates
- The viewing specifications are also in world coordinates and it is the size of the viewing volume that determines what will appear in the image
- Internally, OpenGL will convert to camera coordinates and later to screen coordinates

OpenGL Camera

- OpenGL places a camera at the origin pointing in the negative z direction
- The default viewing volume is a box centered at the origin with a side of length 2
Orthographic Viewing

In the default orthographic view, points are projected forward along the z axis onto the plane z=0.

Transformations and Viewing

• In OpenGL, the projection is carried out by a projection matrix (transformation)
• There is only one set of transformation functions so we must set the matrix mode first
  glMatrixMode (GL_PROJECTION)
• Transformation functions are incremental so we start with an identity matrix and alter it with a projection matrix that gives the view volume
  glLoadIdentity();
  glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);

Two- and three-dimensional viewing

• In glOrtho(left, right, bottom, top, near, far) the near and far distances are measured from the camera
• Two-dimensional vertex commands place all vertices in the plane z=0
• If the application is in two dimensions, we can use the function
  gluOrtho2D(left, right, bottom, top)
• In two dimensions, the view or clipping volume becomes a clipping window

OpenGL Primitives

Example: Drawing an Arc

• Given a circle with radius r, centered at (x,y), draw an arc of the circle that sweeps out an angle \( \theta \).

\[
(x, y) = (x_0 + r \cos \theta, y_0 + r \sin \theta),
\]

for \( 0 \leq \theta \leq 2\pi \).
The Line Strip Primitive

```c
void drawArc(float x, float y, float r, 
float t0, float sweep)
{
    float t, dt; /* angle */
    int n = 30; /* # of segments */
    int i;
    t = t0 * PI/180.0; /* radians */
    dt = sweep * PI/(180*n); /* increment */
    glBegin(GL_LINE_STRIP);
    for(i=0; i<=n; i++, t += dt)
        glVertex2f(x + r*cos(t), y + r*sin(t));
    glEnd();
}
```

Polygon Issues

- OpenGL will only display polygons correctly that are
  - Simple: edges cannot cross
  - Convex: All points on line segment between two
    points in a polygon are also in the polygon
  - Flat: all vertices are in the same plane
- User program must check if above true
- Triangles satisfy all conditions

Attributes

- Attributes are part of the OpenGL and determine the appearance of objects
  - Color (points, lines, polygons)
  - Size and width (points, lines)
  - Stipple pattern (lines, polygons)
  - Polygon mode
    - Display as filled: solid color or stipple pattern
    - Display edges

RGB color

- Each color component stored separately in the frame buffer
- Usually 8 bits per component in buffer
- Note in `glColor3f` the color values range from 0.0 (none) to 1.0 (all), while in `glColor3ub` the values range from 0 to 255

Indexed Color

- Colors are indices into tables of RGB values
- Requires less memory
  - indices usually 8 bits
  - not as important now
    - Memory inexpensive
    - Need more colors for shading

Color and State

- The color as set by `glColor` becomes part of the state and will be used until changed
  - Colors and other attributes are not part of the object but are assigned when the object is rendered
- We can create conceptual vertex colors by code such as
  ```c
  glColor
  glVertex
  ```
Smooth Color

- Default is smooth shading
  - OpenGL interpolates vertex colors across visible polygons
- Alternative is flat shading
  - Color of first vertex determines fill color
  
  • `glShadeModel(GL_SMOOTH)` or `GL_FLAT`

Viewports

- Do not have use the entire window for the image: `glViewport(x, y, w, h)`
- Values in pixels (screen coordinates)

Objectives

- Develop a more sophisticated three-dimensional example
  - Sierpinski gasket: a fractal
- Introduce hidden-surface removal

Three-dimensional Applications

- In OpenGL, two-dimensional applications are a special case of three-dimensional graphics
  - Not much changes
  - Use `glVertex3*( )`
  - Have to worry about the order in which polygons are drawn or use hidden-surface removal
  - Polygons should be simple, convex, flat

Sierpinski Gasket (2D)

- Start with a triangle
- Connect bisectors of sides and remove central triangle
- Repeat
Example

• Five subdivisions

The gasket as a fractal

• Consider the filled area (black) and the perimeter (the length of all the lines around the filled triangles)
  • As we continue subdividing
    - the area goes to zero
    - but the perimeter goes to infinity
• This is not an ordinary geometric object
  • It is neither two- nor three-dimensional
• It has a fractal (fractional dimension) object

Gasket Program

```c
#include <GL/glut.h>
/* a point data type
typedef GLfloat point2[2];
/* initial triangle */
point2 v[]={{-1.0, -0.58}, {1.0, -0.58}, {0.0, 1.15}};
int n; /* number of recursive steps */
```

Draw a triangle

```c
/* display one triangle */
void triangle(point2 a, point2 b, point2 c)
{
    glBegin(GL_TRIANGLES);
    glVertex2fv(a);
    glVertex2fv(b);
    glVertex2fv(c);
    glEnd();
}
```

Triangle Subdivision

```c
/* triangle subdivision using vertex numbers */
void divide_triangle(point2 a, point2 b, point2 c, int m)
{
    point2 v0, v1, v2;
    int j;
    if(m > 0) {
        for(j=0; j<2; j++) v0[j]=(a[j]+b[j])/2;
        for(j=0; j<2; j++) v1[j]=(a[j]+c[j])/2;
        for(j=0; j<2; j++) v2[j]=(b[j]+c[j])/2;
        divide_triangle(a, v0, v1, m-1);
        divide_triangle(c, v1, v2, m-1);
        divide_triangle(b, v2, v0, m-1);
    } /* else, draw triangle at end of recursion */
    else triangle(a,b,c);
}
```

Display and Init Functions

```c
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    divide_triangle(v[0], v[1], v[2], n);
    glFlush();
}
void myinit()
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-2.0, 2.0, -2.0, 2.0);
    glMatrixMode(GL_MODELVIEW);
    glClearColor (1.0, 1.0, 1.0,1.0);  
    glColor3f(0.0,0.0,0.0);  
}
```
main Function

```c
int main(int argc, char **argv)
{
  n=4;
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
  glutInitWindowSize(500, 500);
  glutCreateWindow("2D Gasket");
  glutDisplayFunc(display);
  myinit();
  glutMainLoop();
}
```

Moving to 3D

- We can easily make the program three-dimensional by using
  ```c
typedef GLfloat point3[3]
glVertex3f
glOrtho
```

- Instead, we can start with a tetrahedron

3D Gasket

- We can subdivide each of the four faces

  - Appears as if we remove a solid tetrahedron from the center leaving four smaller tetrahedra

Example

```markdown
after 5 iterations
```

triangle code

```c
void triangle(point3 a, point3 b, point3 c)
{
  glBegin(GL_POLYGON);
  glVertex3fv(a);
  glVertex3fv(b);
  glVertex3fv(c);
  glEnd();
}
```

subdivision code

```c
void divide_triangle(point3 a, point3 b, point3 c, int m)
{
  point3 v1, v2, v3;
  int j;
  if(m > 0) {
    for(j=0; j<3; j++) v1[j]=(a[j]+b[j])/2;
    for(j=0; j<3; j++) v2[j]=(a[j]+c[j])/2;
    for(j=0; j<3; j++) v3[j]=(b[j]+c[j])/2;
    divide_triangle(a, v1, v2, m-1);
    divide_triangle(c, v2, v3, m-1);
    divide_triangle(b, v3, v1, m-1);
  }
  else triangle(a,b,c);
}
```
tetrahedron code

```c
void tetrahedron(int m)
{
    glColor3f(1.0, 0.0, 0.0);
    divide_triangle(v[0], v[1], v[2], m);

    glColor3f(0.0, 1.0, 0.0);
    divide_triangle(v[3], v[2], v[1], m);

    glColor3f(0.0, 0.0, 1.0);
    divide_triangle(v[0], v[3], v[1], m);

    glColor3f(0.0, 0.0, 0.0);
    divide_triangle(v[0], v[2], v[3], m);
}
```

Almost Correct

- Because the triangles are drawn in the order they are defined in the program, the front triangles are not always rendered in front of triangles behind them.

Hidden-Surface Removal

- We want to see only those surfaces in front of other surfaces.
- OpenGL uses a hidden-surface method called the z-buffer algorithm that saves depth information as objects are rendered so that only the front objects appear in the image.

Using the z-buffer algorithm

- The algorithm uses an extra buffer, the z-buffer, to store depth information as geometry travels down the pipeline.
- It must be:
  - Requested in main.c:
    - glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH)
  - Enabled in init.c:
    - glEnable(GL_DEPTH_TEST)
  - Cleared in the display callback:
    - glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

Example: Drawing a Sphere

- In this example, we draw a sphere using a combination of OpenGL primitives.
- Locate points on the sphere by varying two parameters: longitude and latitude.

\[
x(\theta, \phi) = \sin \theta \cos \phi \\
y(\theta, \phi) = \cos \theta \cos \phi \\
z(\theta, \phi) = \cos \phi
\]

Longitude and Latitude Using Quad Strips

```c
c = M_PI / 180.0; /* convert degrees to radians */
for(p = -80.0; p <= 80.0; p += 20.0) {
    glBegin(GL_QUAD_STRIP);
    for(t = -180.0; t <= 180.0; t += 20.0) {
        x = sin(c*t) * cos(c*p);
        y = cos(c*t) * cos(c*p);
        z = cos(c*p);
        glVertex3d(x, y, z);
        x = sin(c*(t + 20.0)) * cos(c*(p + 20.0));
        y = cos(c*t) * cos(c*(p + 20.0));
        z = cos(c*(p + 20.0));
        glVertex3d(x, y, z);
    }
    glEnd();
}
```
Covering the Poles with Triangle Fans

/* north pole */
z = 1.0;
glBegin(GL_TRIANGLE_FAN);
glVertex3d(x, y, z);
for (t = -180.0; t <= 180.0; t += 20.0) {
  x = sin(c*t) * cos(c*80.0);
  y = cos(c*t) * cos(c*80.0);
  glVertex3d(x, y, z);
}
glEnd();

• The south pole is a reflection in the xy-plane.

Subdivision Sphere

• We can approximate a sphere to any desired resolution by recursive subdivision on a tetrahedron.

typedef float point[4];
/* initial tetrahedron */
point v[] = {
  { 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0 },
  { -0.816497, -0.471405, -0.333333 },
  { -0.816497, -0.471405, -0.333333 },
  { 0.816497, -0.471405, -0.333333 }
};

Subdividing a Tetrahedron

• Subdividing the triangular faces of the tetrahedron gives us the desired approximation.

void tetrahedron(int m) {
  /* subdivide the tetrahedron faces */
  divide_triangle(v[0], v[1], v[2], m);
  divide_triangle(v[3], v[2], v[1], m);
  divide_triangle(v[0], v[3], v[1], m);
  divide_triangle(v[0], v[2], v[3], m);
}

Input and Interaction

Prof. George Wolberg
Dept. of Computer Science
City College of New York

Objectives

• Introduce the basic input devices
  - Physical Devices
  - Logical Devices
  - Input Modes
• Event-driven input
• Introduce double buffering for smooth animations
• Programming event input with GLUT

Project Sketchpad

• Ivan Sutherland (MIT 1963) established the basic interactive paradigm that characterizes interactive computer graphics:
  - User sees an object on the display
  - User points to (picks) the object with an input device (light pen, mouse, trackball)
  - Object changes (moves, rotates, morphs)
  - Repeat
Graphical Input

- Devices can be described either by
  - Physical properties
    - Mouse
    - Keyboard
    - Trackball
  - Logical Properties
    - What is returned to program via API
      - A position
        - An object identifier
- Modes
  - How and when input is obtained
    - Request or event

Physical Devices

- Mouse
- Trackball
- Light pen
- Data tablet
- Joy stick
- Space ball

Incremental (Relative) Devices

- Devices such as the data tablet return a position directly to the operating system
- Devices such as the mouse, trackball, and joy stick return incremental inputs (or velocities) to the operating system
  - Must integrate these inputs to obtain an absolute position
    - Rotation of wheels in mouse
    - Roll of trackball
    - Difficult to obtain absolute position
    - Can get variable sensitivity

Logical Devices

- Consider the C and C++ code
  - C++: cin >> x;
  - C: scanf ("%d", &x);
- What is the input device?
  - Can’t tell from the code
  - Could be keyboard, file, output from another program
- The code provides logical input
  - A number (an int) is returned to the program regardless of the physical device

Graphical Logical Devices

- Graphical input is more varied than input to standard programs which is usually numbers, characters, or bits
- Two older APIs (GKS, PHIGS) defined six types of logical input
  - Locator: return a position
  - Pick: return ID of an object
  - Keyboard: return strings of characters
  - Stroke: return array of positions
  - Valuator: return floating point number
  - Choice: return one of n items

X Window Input

- The X Window System introduced a client-server model for a network of workstations
  - Client: OpenGL program
  - Graphics Server: bitmap display with a pointing device and a keyboard

Input Modes

- Input devices contain a trigger which can be used to send a signal to the operating system
  - Button on mouse
  - Pressing or releasing a key
- When triggered, input devices return information (their measure) to the system
  - Mouse returns position information
  - Keyboard returns ASCII code

Request Mode

- Input provided to program only when user triggers the device
- Typical of keyboard input
  - Can erase (backspace), edit, correct until enter (return) key (the trigger) is depressed

Event Mode

- Most systems have more than one input device, each of which can be triggered at an arbitrary time by a user
- Each trigger generates an event whose measure is put in an event queue which can be examined by the user program

Event Types

- Window: resize, expose, iconify
- Mouse: click one or more buttons
- Motion: move mouse
- Keyboard: press or release a key
- Idle: nonevent
  - Define what should be done if no other event is in queue

Callbacks

- Programming interface for event-driven input
- Define a callback function for each type of event the graphics system recognizes
- This user-supplied function is executed when the event occurs
- GLUT example:
  `glutMouseFunc(mymouse)`

GLUT callbacks

GLUT recognizes a subset of the events recognized by any particular window system (Windows, X, Macintosh)
- `glutDisplayFunc`
- `glutMouseFunc`
- `glutReshapeFunc`
- `glutKeyFunc`
- `glutIdleFunc`
- `glutMotionFunc`, `glutPassiveMotionFunc`
GLUT Event Loop

- Remember that the last line in main.c for a program using GLUT must be glutMainLoop(); which puts the program in an infinite event loop
- In each pass through the event loop, GLUT
  - looks at the events in the queue
  - for each event in the queue, GLUT executes the appropriate callback function if one is defined
  - if no callback is defined for the event, the event is ignored

The display callback

- The display callback is executed whenever GLUT determines that the window should be refreshed, for example
  - When the window is first opened
  - When the window is reshaped
  - When a window is exposed
  - When the user program decides it wants to change the display
- In main.c
  - glutDisplayFunc(mydisplay) identifies the function to be executed
  - Every GLUT program must have a display callback

Posting redisplays

- Many events may invoke the display callback function
  - Can lead to multiple executions of the display callback on a single pass through the event loop
- We can avoid this problem by instead using glutPostRedisplay(); which sets a flag.
- GLUT checks to see if the flag is set at the end of the event loop
- If set then the display callback function is executed

Animating a Display

- When we redraw the display through the display callback, we usually start by clearing the window
  - glClear()
- then draw the altered display
- Problem: the drawing of information in the frame buffer is decoupled from the display of its contents
  - Graphics systems use dual ported memory
  - Hence we can see partially drawn display
  - See the program single_double.c for an example with a rotating cube

Double Buffering

- Instead of one color buffer, we use two
  - Front Buffer: one that is displayed but not written to
  - Back Buffer: one that is written to but not displayed
- Program then requests a double buffer in main.c
  - glutInitDisplayMode(GL_RGB | GL_DOUBLE)
  - At the end of the display callback buffers are swapped
    void mydisplay()
    {
      glClear();
      /* draw graphics here */
      glutSwapBuffers();
    }

Using the idle callback

- The idle callback is executed whenever there are no events in the event queue
  - glutIdleFunc(myidle)
  - Useful for animations
  - void myidle()
  
  void mydisplay()
  { 
    /* change something */
    t += dt
    glutPostRedisplay();
  }

  void mydisplay()
  { 
    glClear();
    /* draw something that depends on t */
    glutSwapBuffers();
  }
Using globals

- The form of all GLUT callbacks is fixed:
  void mydisplay()
  void mymouse(GLint button, GLint state, GLint x, GLint y)
- Must use globals to pass information to callbacks:
  float t; /* global */
  void mydisplay()
  { /* draw something that depends on t */

Working with Callbacks

Prof. George Wolberg
Dept. of Computer Science
City College of New York

Objectives

- Learn to build interactive programs using GLUT callbacks
  - Mouse
  - Keyboard
  - Reshape
- Introduce menus in GLUT

The mouse callback

- glutMouseFunc(mymouse)
- void mymouse(GLint button, GLint state, GLint x, GLint y)
- Returns:
  - which button (GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, GLUT_RIGHT_BUTTON) caused event
  - state of that button (GLUT_UP, GLUT_DOWN)
  - Position in window

Positioning

- The position in the screen window is usually measured in pixels with the origin at the top-left corner
- Consequence of refresh done from top to bottom
- OpenGL uses a world coordinate system with origin at the bottom left
- Must invert y coordinate returned by callback by height of window:
  \[ y = h - y \]

Obtaining the window size

- To invert the y position we need the window height:
  - Height can change during program execution
  - Track with a global variable
  - New height returned to reshape callback that we will look at in detail soon
  - Can also use enquiry functions
  &gt; glutGet()
  &gt; glutGetFloat()
  to obtain any value that is part of the state
Terminating a program

• In our original programs, there was no way to terminate them through OpenGL
• We can use the simple mouse callback

```c
void mouse(int btn, int state, int x, int y)
{
    if(btn==GLUT_RIGHT_BUTTON && state==GLUT_DOWN)
        exit(0);
}
```

Using the mouse position

• In the next example, we draw a small square at the location of the mouse each time the left mouse button is clicked
• This example does not use the display callback but one is required by GLUT; We can use the empty display callback function

```c
void mydisplay()
{
}
```

Using the motion callback

• We can draw squares (or anything else) continuously as long as a mouse button is depressed by using the motion callback
  - glutMotionFunc(drawSquare)
• We can draw squares without depressing a button using the passive motion callback
  - glutPassiveMotionFunc(drawSquare)

Using the keyboard

• glutKeyboardFunc(mykey)
• void mykey(unsigned char key, int x, int y)
  - Returns ASCII code of key depressed and mouse location
  - Note GLUT does not recognize key release as an event

```c
void mykey()
{
    if(key == 'Q' || key == 'q')
        exit(0);
}
```

Special and Modifier Keys

• GLUT defines the special keys in glut.h
  - Function key 1: GLUT_KEY_F1
  - Up arrow key: GLUT_KEY_UP
    - if(key == 'GLUT_KEY_UP')
  • Can also check if one of the modifiers
    - GLUT_ACTIVE_SHIFT
    - GLUT_ACTIVE_CTRL
    - GLUT_ACTIVE_ALT
    - is depressed by
      - glutGetModifiers()
    - Allows emulation of three-button mouse with one- or two-button mice
Reshaping the window

• We can reshape and resize the OpenGL display window by pulling the corner of the window
• What happens to the display?
  - Must redraw from application
  - Two possibilities
    - Display part of world
    - Display whole world but force to fit in new window
      - Can alter aspect ratio

Reshape possibilities

We can reshape and resize the OpenGL display window by pulling the corner of the window. What happens to the display?

- Must redraw from application
- Two possibilities:
  - Display part of world
  - Display whole world but force to fit in new window
    - Can alter aspect ratio

The Reshape callback

• glutReshapeFunc(myreshape)
• void myreshape( int w, int h)
  - Returns width and height of new window (in pixels)
  - A redisplay is posted automatically at end of execution of the callback
  - GLUT has a default reshape callback but you probably want to define your own
• The reshape callback is good place to put camera functions because it is invoked when the window is first opened

Example Reshape

This reshape preserves shapes by making the viewport and world window have the same aspect ratio

```
void myReshape(int w, int h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION); /* switch matrix mode */
    glLoadIdentity();
    if (w <= h)
        gluOrtho2D(-2.0, 2.0, -2.0 * (GLfloat) h / (GLfloat) w, 2.0 * (GLfloat) h / (GLfloat) w);
    else  gluOrtho2D(-2.0 * (GLfloat) w / (GLfloat) h, 2.0 * (GLfloat) w / (GLfloat) h, -2.0, 2.0);
    glMatrixMode(GL_MODELVIEW); /* return to modelview mode */
}
```

Toolkits and Widgets

• Most window systems provide a toolkit or library of functions for building user interfaces that use special types of windows called widgets
• Widget sets include tools such as
  - Menus
  - Slidebars
  - Dials
  - Input boxes
• But toolkits tend to be platform dependent
• GLUT provides a few widgets including menus

Menus

• GLUT supports pop-up menus
  - A menu can have submenus
• Three steps
  - Define entries for the menu
  - Define action for each menu item
    - Action carried out if entry selected
  - Attach menu to a mouse button
Defining a simple menu

- In `main.c`
  ```c
  menu_id = glutCreateMenu(mymenu);
  glutAddmenuEntry("clear Screen", 1);
  glutAddMenuEntry("exit", 2);
  glutAttachMenu(GLUT_RIGHT_BUTTON);
  ```

Menu actions

- Menu callback
  ```c
  void mymenu(int id)
  {
    if(id == 1) glClear();
    if(id == 2) exit(0);
  }
  ```

  - Note each menu has an id that is returned when it is created
  - Add submenus by
    ```c
    glutAddSubmenu(char *submenu_name, submenu id)
    ```

Other functions in GLUT

- Dynamic Windows
  - Create and destroy during execution
- Subwindows
- Multiple Windows
- Changing callbacks during execution
- Timers
- Portable fonts
  - `glutBitmapCharacter`
  - `glutStrokeCharacter`

Better Interactive Programs

Prof. George Wolberg
Dept. of Computer Science
City College of New York

Objectives

- Learn to build more sophisticated interactive programs using
  - Picking
    - Select objects from the display
    - Three methods
  - Rubberbanding
    - Interactive drawing of lines and rectangles
  - Display Lists
    - Retained mode graphics

Picking

- Identify a user-defined object on the display
- In principle, it should be simple because the mouse gives the position and we should be able to determine to which object(s) a position corresponds
- Practical difficulties
  - Pipeline architecture is feed forward, hard to go from screen back to world
  - Complicated by screen being 2D, world is 3D
  - How close do we have to come to object to say we selected it?
Three Approaches

- **Hit list**
  - Most general approach but most difficult to implement
  - Use back or some other buffer to store object ids as the objects are rendered
- **Rectangular maps**
  - Easy to implement for many applications
  - See paint program in text

Rendering Modes

- OpenGL can render in one of three modes selected by `glRenderMode(mode)`
  - `GL_RENDER`: normal rendering to the frame buffer (default)
  - `GL_FEEDBACK`: provides list of primitives rendered but no output to the frame buffer
  - `GL_SELECTION`: Each primitive in the view volume generates a hit record that is placed in a name stack which can be examined later

Selection Mode Functions

- `glSelectBuffer()`: specifies name buffer
- `glInitNames()`: initializes name buffer
- `glPushName(id)`: push id on name buffer
- `glPopName()`: pop top of name buffer
- `glLoadName(id)`: replace top name on buffer

- id is set by application program to identify objects

Using Selection Mode

- Initialize name buffer
- Enter selection mode (using mouse)
- Render scene with user-defined identifiers
- Reenter normal render mode
  - This operation returns number of hits
- Examine contents of name buffer (hit records)
  - Hit records include id and depth information

Selection Mode and Picking

- As we just described it, selection mode won’t work for picking because every primitive in the view volume will generate a hit
- Change the viewing parameters so that only those primitives near the cursor are in the altered view volume
  - Use `gluPickMatrix` (see text for details)

Example: Picking

- In this example, we use picking to select rectangles using the mouse.
Assigning Object Names

```c
void drawSquares(GLenum mode) {
    GLuint i, j;
    for(i=0; i<3; i++) {
        if(mode == GL_SELECT) glLoadName(i);
        for(j=0; j<3; j++) {
            if(mode == GL_SELECT) glPushName(j);
            glColor3fv((GLfloat) colors[i][j]);
            glRecti(i, j, i+1, j+1);
            if(mode == GL_SELECT) glPopName();
        }
    }
}
```

Initializing Select Mode

• This code segment belongs in the mouse handler (or in a function called by the mouse handler).

```c
GLuint buf[BUFSIZE]; /* selection buffer */
GLint viewport[4]; /* window dimensions */
int hits; /* hit count */

glGetIntegerv(GL_VIEWPORT, viewport);
glSelectBuffer(BUFSIZE, buf);
(void) glRenderMode(GL_SELECT);
glInitNames();
glPushName(0);
```

Drawing in Select Mode

```c
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();
/* 5x5 picking region near cursor */
gluPickMatrix((GLdouble) x,
    (GLdouble) (viewport[3] - y),
    5.0, 5.0, viewport);
gluOrtho2D(0.0, 3.0, 0.0, 3.0);
drawSquares(GL_SELECT);
glMatrixMode(GL_PROJECTION);
glPopMatrix();
glFlush();
```

Processing Hits

```c
hits = glRenderMode(GL_RENDER);
ptr = (GLuint *) buf;
for(i=0; i<hits; i++) { /* for each hit */
    names = *ptr; ptr++; /* how many names? */
    z1 = (float) *ptr/0x7fffffff; ptr++;
    z2 = (float) *ptr/0x7fffffff; ptr++;
    for(j=0; j<names; j++) { /* for each name */
        if(j == 0) ii = *ptr; /* set row */
        else if(j == 1) jj = *ptr; /* set column */
        ptr++;
    }
}
```

• Choose the smallest z.

Using Regions of the Screen

• Many applications use a simple rectangular arrangement of the screen
  - Example: paint/CAD program

<table>
<thead>
<tr>
<th>tools</th>
<th>drawing area</th>
<th>menus</th>
</tr>
</thead>
</table>

• Easier to look at mouse position and determine which area of screen it is in that using selection mode picking

Using another buffer and colors for picking

• For a small number of objects, we can assign a unique color (often in color index mode) to each object
• We then render the scene to a color buffer other than the front buffer so the results of the rendering are not visible
• We then get the mouse position and use glReadPixels() to read the color in the buffer we just wrote at the position of the mouse
• The returned color gives the id of the object
Writing Modes

bitwise logical operation

application

Source

s

d' = s

frame buffer

read_pixel

write_pixel

Destination

XOR write

• Usual (default) mode: source replaces destination (d' = s)
  - Cannot write temporary lines this way because we cannot recover what was “under” the line in a fast simple way
• Exclusive OR mode (XOR) (d' = d ⊕ s)
  - x ⊕ y ⊕ x = x
  - Hence, if we use XOR mode to write a line, we can draw it a second time and line is erased!

Rubberbanding

• Switch to XOR write mode
• Draw object
  - For line can use first mouse click to fix one endpoint and then use motion callback to continuously update the second endpoint
  - Each time mouse is moved, redraw line which erases it and then draw line from fixed first position to new second position
  - At end, switch back to normal drawing mode and draw line
  - Works for other objects: rectangles, circles

Rubberband Lines

initial display

draw line with mouse

in XOR mode

mouse moved to original line redrawn

new position

with XOR

new line drawn with XOR

XOR in OpenGL

• There are 16 possible logical operations between two bits
• All are supported by OpenGL
  - Must first enable logical operations
    - glEnable(GL_COLOR_LOGIC_OP)
  - Choose logical operation
    - glLogicOp(GL_XOR)
    - glLogicOp(GL_COPY) (default)

Example: Erasing Lines

• In this example, we use the XOR drawing mode to draw erasable line segments.
• The user selects the first endpoint using the mouse.

```c
/* global variables */
float xm, ym, xmm, ymm;

/* in mouse handler */
xm = x/500.;
ym = (500 - y)/500.;
```
XOR Drawing Mode

• When the user selects a second point, draw the line in XOR mode.

\[
\begin{align*}
\text{xmm} &= x/500.; \\
\text{ymm} &= (500 - y)/500.; \\
glLogicOp(GL\_XOR); \\
glBegin(GL\_LINES); \\
glVertex2f(xm, ym); \\
glVertex2f(xmm, ymm); \\
glEnd(); \\
glLogicOp(GL\_COPY); \\
glFlush();
\end{align*}
\]

Erasing the Line

• Redraw the old line in XOR mode before retrieving new coordinates.

\[
\begin{align*}
glLogicOp(GL\_XOR); \\
glBegin(GL\_LINES); \\
glVertex2f(xm, ym); \\
glVertex2f(xmm, ymm); \\
glEnd(); \\
glFlush(); \\
\text{xmm} &= x/500.; \\
\text{ymm} &= (500 - y)/500.; \\
glBegin(GL\_LINES); \\
\ldots \\
glLogicOp(GL\_COPY); \\
\end{align*}
\]

Immediate and Retained Modes

• Recall that in a standard OpenGL program, once an object is rendered there is no memory of it and to redisplay it, we must re-execute the code for it.
  - Known as immediate mode graphics
  - Can be especially slow if the objects are complex and must be sent over a network
• Alternative is define objects and keep them in some form that can be redisplayed easily
  - Retained mode graphics
  - Accomplished in OpenGL via display lists

Display Lists

• Conceptually similar to a graphics file
  - Must define (name, create)
  - Add contents
  - Close
• In client-server environment, display list is placed on server
  - Can be redisplayed without sending primitives over network each time

Display List Functions

• Creating a display list

\[
\text{GLuint id;}
\]

\[
\begin{align*}
\text{void init( void )}
\{ \\
\quad \text{id = glGenLists( 1 );}
\quad \text{glNewList( id, GL\_COMPILE );}
\quad /* other OpenGL routines */
\quad \text{glEndList();}
\}
\]

• Call a created list

\[
\text{void display( void )}
\]

\[
\text{glCallList( id );}
\]

Display Lists and State

• Most OpenGL functions can be put in display lists
• State changes made inside a display list persist after the display list is executed
• Can avoid unexpected results by using glPushAttrib and glPushMatrix upon entering a display list and glPopAttrib and glPopMatrix before exiting
Hierarchy and Display Lists

- Consider model of a car
  - Create display list for chassis
  - Create display list for wheel
  
  ```c
  glNewList(CAR, GL_COMPILE);
  glCallList(CHASSIS);
  glTranslatef(...);
  glCallList(WHEEL);
  glTranslatef(...);
  glCallList(WHEEL);
  glEndList();
  ```

Geometry

Prof. George Wolberg
Dept. of Computer Science
City College of New York

Objectives

- Introduce the elements of geometry
  - Scalars
  - Vectors
  - Points
- Develop mathematical operations among them in a coordinate-free manner
- Define basic primitives
  - Line segments
  - Polygons

Basic Elements

- Geometry is the study of the relationships among objects in an n-dimensional space
  - In computer graphics, we are interested in objects that exist in three dimensions
- Want a minimum set of primitives from which we can build more sophisticated objects
- We will need three basic elements
  - Scalars
  - Vectors
  - Points

Coordinate-Free Geometry

- When we learned simple geometry, most of us started with a Cartesian approach
  - Points were at locations in space \( p=(x,y,z) \)
  - We derived results by algebraic manipulations involving these coordinates
- This approach was nonphysical
  - Physically, points exist regardless of the location of an arbitrary coordinate system
  - Most geometric results are independent of the coordinate system
  - Euclidean geometry: two triangles are identical if two corresponding sides and the angle between them are identical

Scalars

- Need three basic elements in geometry
  - Scalars, Vectors, Points
- Scalars can be defined as members of sets which can be combined by two operations (addition and multiplication) obeying some fundamental axioms (associativity, commutativity, inverses)
- Examples include the real and complex number under the ordinary rules with which we are familiar
- Scalars alone have no geometric properties
Vectors

- Physical definition: a vector is a quantity with two attributes
  - Direction
  - Magnitude
- Examples include
  - Force
  - Velocity
  - Directed line segments
    - Most important example for graphics
    - Can map to other types

Vector Operations

- Every vector has an inverse
  - Same magnitude but points in opposite direction
- Every vector can be multiplied by a scalar
- There is a zero vector
  - Zero magnitude, undefined orientation
- The sum of any two vectors is a vector
  - Use head-to-tail axiom

Linear Vector Spaces

- Mathematical system for manipulating vectors
- Operations
  - Scalar-vector multiplication $u = \alpha \cdot v$
  - Vector-vector addition: $v = u + w$
- Expressions such as $v = u + 2w - 3r$
  Make sense in a vector space

Vectors Lack Position

- These vectors are identical
  - Same length and magnitude
- Vectors spaces insufficient for geometry
  - Need points

Points

- Location in space
- Operations allowed between points and vectors
  - Point-point subtraction yields a vector
  - Equivalent to point-vector addition

Affine Spaces

- Point + a vector space
- Operations
  - Vector-vector addition
  - Scalar-vector multiplication
  - Point-vector addition
  - Scalar-scalar operations
- For any point define
  - $1 \cdot P = P$
  - $0 \cdot P = 0$ (zero vector)
Lines

- Consider all points of the form
  \[ P(\alpha) = P_0 + \alpha \mathbf{d} \]
  - Set of all points that pass through \( P_0 \) in the direction of the vector \( \mathbf{d} \)

Parametric Form

- This form is known as the parametric form of the line
  - More robust and general than other forms
  - Extends to curves and surfaces

- Two-dimensional forms
  - Explicit: \( y = mx + b \)
  - Implicit: \( ax + by + c = 0 \)
  - Parametric:
    \[ x(\alpha) = \alpha x_0 + (1-\alpha)x_1 \]
    \[ y(\alpha) = \alpha y_0 + (1-\alpha)y_1 \]

Rays and Line Segments

- If \( \alpha \geq 0 \), then \( P(\alpha) \) is the ray leaving \( P_0 \) in the direction \( \mathbf{d} \)
  - If we use two points to define \( \mathbf{v} \), then
    \[ P(\alpha) = Q + \alpha (R-Q) = Q + \alpha \mathbf{v} \]
    \[ = \alpha R + (1-\alpha)Q \]
  - For \( 0 \leq \alpha \leq 1 \) we get all the points on the line segment joining \( R \) and \( Q \)

Convexity

- An object is convex iff for any two points in the object all points on the line segment between these points are also in the object

Affine Sums

- Consider the “sum”
  \[ P = \alpha_1 P_1 + \alpha_2 P_2 + \ldots + \alpha_n P_n \]
  - Can show by induction that this sum makes sense iff
    \[ \alpha_1 + \alpha_2 + \ldots + \alpha_n = 1 \]
    - in which case we have the affine sum of the points \( P_1, P_2, \ldots, P_n \)
  - If, in addition, \( \alpha_i \geq 0 \), we have the convex hull of \( P_1, P_2, \ldots, P_n \)

Convex Hull

- Smallest convex object containing \( P_1, P_2, \ldots, P_n \)
  - Formed by “shrink wrapping” points
Curves and Surfaces

• Curves are one parameter entities of the form $P(\alpha)$ where the function is nonlinear.
• Surfaces are formed from two-parameter functions $P(\alpha, \beta)$.
  - Linear functions give planes and polygons.

Planes

• A plane be determined by a point and two vectors or by three points.

Triangles

• A triangle can be formed by convex sums of points.
  - Convex sum of $P$ and $Q$.
  - Convex sum of $S(\alpha)$ and $R$.

For $0 \leq \alpha, \beta \leq 1$, we get all points in the triangle.

Normals

• Every plane has a vector $n$ normal (perpendicular, orthogonal) to it.
• From point-two vector form $P(\alpha, \beta) = R + \alpha u + \beta v$, we know we can use the cross product to find $n = u \times v$ and the equivalent form $(P(\alpha)-P) \times n = 0$.

Objectives

• Introduce concepts such as dimension and basis.
• Introduce coordinate systems for representing vectors spaces and frames for representing affine spaces.
• Discuss change of frames and basis.
• Introduce homogeneous coordinates.
Linear Independence

A set of vectors \( v_1, v_2, \ldots, v_n \) is *linearly independent* if
\[ v_1 + v_2 + \ldots + v_n = 0 \text{ iff } \alpha_1 = \alpha_2 = \ldots = 0 \]

- If a set of vectors is linearly independent, we cannot represent one in terms of the others.
- If a set of vectors is linearly dependent, at least one can be written in terms of the others.

Dimension

- In a vector space, the maximum number of linearly independent vectors is fixed and is called the *dimension* of the space.
- In an \( n \)-dimensional space, any set of \( n \) linearly independent vectors form a *basis* for the space.
- Given a basis \( v_1, v_2, \ldots, v_n \), any vector \( v \) can be written as
  \[ v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n \]
  where the \( \{ \alpha_j \} \) are unique.

Representation

- Until now we have been able to work with geometric entities without using any frame of reference, such a coordinate system.
- Need a frame of reference to relate points and objects to our physical world.
  - For example, where is a point? Can't answer without a reference system.
  - World coordinates.
  - Camera coordinates.

Coordinate Systems

- Consider a basis \( v_1, v_2, \ldots, v_n \).
- A vector is written as \( v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n \).
- The list of scalars \( \{ \alpha_1, \alpha_2, \ldots, \alpha_n \} \) is the representation of \( v \) with respect to the given basis.
- We can write the representation as a row or column array of scalars
  \[ a = [\begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix}] = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix} \]

Example

- \( v = 2v_1 + 3v_2 - 4v_3 \)
- \( A = \begin{bmatrix} 2 & 3 & -4 \end{bmatrix} \)
- Note that this representation is with respect to a particular basis.
- For example, in OpenGL we start by representing vectors using the world basis but later the system needs a representation in terms of the camera or eye basis.

Coordinate Systems

- Which is correct?

  - Both are correct because vectors have no fixed location.
Frames

• Coordinate System is insufficient to present points
• If we work in an affine space we can add a single point, the origin, to the basis vectors to form a frame

Frames II

• Frame determined by \((P_0, v_1, v_2, v_3)\)
• Within this frame, every vector can be written as \(v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n\)
• Every point can be written as \(P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \ldots + \beta_n v_n\)

Confusing Points and Vectors

Consider the point and the vector
\(P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \ldots + \beta_n v_n\)
\(v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n\)

They appear to have the similar representations
\(p = [\beta_1 \beta_2 \beta_3] \quad v = [\alpha_1 \alpha_2 \alpha_3]\)
which confuse the point with the vector
A vector has no position
A point can place anywhere

Homogeneous Coordinates

The general form of four dimensional homogeneous coordinates is
\(p = [x \ y \ z \ w]^T\)

We return to a three dimensional point (for \(w \neq 0\)) by
\(x \leftarrow x/w\)
\(y \leftarrow y/w\)
\(z \leftarrow z/w\)

If \(w = 0\), the representation is that of a vector
Note that homogeneous coordinates replaces points in three dimensions by lines through the origin in four dimensions

Homogeneous Coordinates and Computer Graphics

• Homogeneous coordinates are key to all computer graphics systems
  - All standard transformations (rotation, translation, scaling) can be implemented by matrix multiplications with 4 x 4 matrices
  - Hardware pipeline works with 4 dimensional representations
  - For orthographic viewing, we can maintain \(w = 0\) for vectors and \(w = 1\) for points
  - For perspective we need a perspective division
**Change of Coordinate Systems**

- Consider two representations of a the same vector with respect to two different bases. The representations are

\[
\mathbf{a} = \left[ \alpha_1 \alpha_2 \alpha_3 \right] \\
\mathbf{b} = \left[ \beta_1 \beta_2 \beta_3 \right]
\]

where

\[
v = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 = \left[ \alpha_1 \alpha_2 \alpha_3 \right] \left[ v_1 \ v_2 \ v_3 \right]^T
\]

\[
\mathbf{a} = \left[ \gamma_{11} \gamma_{12} \gamma_{13} \right] \\
\mathbf{b} = \left[ \gamma_{21} \gamma_{22} \gamma_{23} \right] \\
\mathbf{c} = \left[ \gamma_{31} \gamma_{32} \gamma_{33} \right] \\
\mathbf{d} = \left[ \gamma_{41} \gamma_{42} \gamma_{43} \right]
\]

and the basis can be related by

\[
\mathbf{a} = \mathbf{M}^T \mathbf{b}
\]

see text for numerical examples.

**Representing second basis in terms of first**

Each of the basis vectors, \( u_1, u_2, u_3 \), are vectors that can be represented in terms of the first basis

\[
u_1 = \gamma_{11} v_1 + \gamma_{12} v_2 + \gamma_{13} v_3 \\
u_2 = \gamma_{21} v_1 + \gamma_{22} v_2 + \gamma_{23} v_3 \\
u_3 = \gamma_{31} v_1 + \gamma_{32} v_2 + \gamma_{33} v_3
\]

**Change of Frames**

- We can apply a similar process in homogeneous coordinates to the representations of both points and vectors
- Consider two frames

\[
(P_0, v_1, v_2, v_3) \\
(Q_0, u_1, u_2, u_3)
\]

\[
\mathbf{a} = \left[ \alpha_1 \alpha_2 \alpha_3 \alpha_4 \right]
\]

in the first frame

\[
\mathbf{b} = \left[ \beta_1 \beta_2 \beta_3 \beta_4 \right]
\]

in the second frame

where \( \alpha_4 = \beta_4 = 1 \) for points and \( \alpha_4 = \beta_4 = 0 \) for vectors and

\[
\mathbf{a} = \mathbf{M}^T \mathbf{b}
\]

The matrix \( \mathbf{M} \) is 4 x 4 and specifies an affine transformation in homogeneous coordinates.

**Matrix Form**

The coefficients define a 3 x 3 matrix

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

**Representing One Frame in Terms of the Other**

Extending what we did with change of bases

\[
u_1 = \gamma_{11} v_1 + \gamma_{12} v_2 + \gamma_{13} v_3 \\
u_2 = \gamma_{21} v_1 + \gamma_{22} v_2 + \gamma_{23} v_3 \\
u_3 = \gamma_{31} v_1 + \gamma_{32} v_2 + \gamma_{33} v_3
\]

\[
P_0 = \gamma_{41} v_1 + \gamma_{42} v_2 + \gamma_{43} v_3 + \gamma_{44} P_0
\]

defining a 4 x 4 matrix

\[
\mathbf{M} = \begin{bmatrix}
\gamma_{11} & \gamma_{12} & \gamma_{13} & 0 \\
\gamma_{21} & \gamma_{22} & \gamma_{23} & 0 \\
\gamma_{31} & \gamma_{32} & \gamma_{33} & 0 \\
\gamma_{41} & \gamma_{42} & \gamma_{43} & 1
\end{bmatrix}
\]

**Working with Representations**

Within the two frames any point or vector has a representation of the same form

\[
\mathbf{a} = \left[ \alpha_1 \alpha_2 \alpha_3 \alpha_4 \right]
\]

in the first frame

\[
\mathbf{b} = \left[ \beta_1 \beta_2 \beta_3 \beta_4 \right]
\]

in the second frame

where \( \alpha_4 = \beta_4 = 1 \) for points and \( \alpha_4 = \beta_4 = 0 \) for vectors and

\[
\mathbf{a} = \mathbf{M}^T \mathbf{b}
\]
Affine Transformations

• Every linear transformation is equivalent to a change in frames
• Every affine transformation preserves lines
• However, an affine transformation has only 12 degrees of freedom because 4 of the elements in the matrix are fixed and are a subset of all possible 4 x 4 linear transformations

The World and Camera Frames

• When we work with representations, we work with n-tuples or arrays of scalars
• Changes in frame are then defined by 4 x 4 matrices
• In OpenGL, the base frame that we start with is the world frame
• Eventually we represent entities in the camera frame by changing the world representation using the model-view matrix
• Initially these frames are the same (M=I)

Moving the Camera

If objects are on both sides of z=0, we must move camera frame.

\[
M = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & -d \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

Transformations

Prof. George Wolberg
Dept. of Computer Science
City College of New York

Objectives

• Introduce standard transformations
  - Rotations
  - Translation
  - Scaling
  - Shear
• Derive homogeneous coordinate transformation matrices
• Learn to build arbitrary transformation matrices from simple transformations

General Transformations

• A transformation maps points to other points and/or vectors to other vectors

\[
P \rightarrow \overrightarrow{v} = T(u)
\]
Affine Transformations

- Line preserving
- Characteristic of many physically important transformations
  - Rigid body transformations: rotation, translation
  - Scaling, shear
- Importance in graphics is that we need only transform endpoints of line segments and let implementation draw line segment between the transformed endpoints

Pipeline Implementation

Translation

- Move (translate, displace) a point to a new location
- Displacement determined by a vector $\mathbf{d}$
  - Three degrees of freedom
  - $\mathbf{P}' = \mathbf{P} + \mathbf{d}$

Object Translation

Every point in object is displaced by same vector

Translation Using Representations

Using the homogeneous coordinate representation in some frame

$$\mathbf{p}=[x\ y\ z\ 1]^T$$
$$\mathbf{p'}=[x'\ y'\ z'\ 1]^T$$
$$\mathbf{d}=[dx\ dy\ dz\ 0]^T$$

Hence $\mathbf{p'} = \mathbf{p} + \mathbf{d}$ or

$$x' = x + dx$$
$$y' = y + dy$$
$$z' = z + dz$$

Note that this expression is in four dimensions and expresses that point = vector + point

Notation

We will be working with both coordinate-free representations of transformations and representations within a particular frame

$\mathbf{P, Q, R}$: points in an affine space
$\mathbf{u, v, w}$: vectors in an affine space
$\alpha, \beta, \gamma$: scalars

$\mathbf{p, q, r}$: representations of points
- array of 4 scalars in homogeneous coordinates

$\mathbf{u, v, w}$: representations of points
- array of 4 scalars in homogeneous coordinates
### Translation Matrix

We can also express translation using a 4 x 4 matrix \( T \) in homogeneous coordinates:

\[
p' = Tp
\]

\[
T = T(d_x, d_y, d_z) = \begin{bmatrix}
1 & 0 & 0 & d_x \\
0 & 1 & 0 & d_y \\
0 & 0 & 1 & d_z \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

This form is better for implementation because all affine transformations can be expressed this way and multiple transformations can be concatenated together.

---

### Rotation (2D)

- Consider rotation about the origin by \( \theta \) degrees:
  - Radius stays the same, angle increases by \( \theta \)

\[
\begin{align*}
x' &= r \cos (\phi + \theta) \\
y' &= r \sin (\phi + \theta)
\end{align*}
\]

---

### Rotation about the z-axis

- Rotation about z-axis in three dimensions leaves all points with the same z.
  - Equivalent to rotation in two dimensions in planes of constant z:
    \[
    \begin{align*}
x' &= x \cos \theta - y \sin \theta \\
y' &= x \sin \theta + y \cos \theta \\
z' &= z
\end{align*}
    \]
  - Or in homogeneous coordinates:
    \[
p' = R_z(\theta)p
    \]

---

### Rotation about x and y axes

- Same argument as for rotation about z-axis:
  - For rotation about x-axis, \( x \) is unchanged.
  - For rotation about y-axis, \( y \) is unchanged.

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

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

---

### Scaling

Expand or contract along each axis (fixed point of origin):

\[
\begin{align*}
x' &= s_x x \\
y' &= s_y y \\
z' &= s_z z
\end{align*}
\]

\[
p' = Sp
\]

\[
S = \begin{bmatrix}
s_x & 0 & 0 & 0 \\
0 & s_y & 0 & 0 \\
0 & 0 & s_z & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
Reflection

corresponds to negative scale factors

\[ \begin{align*}
  s_x &= -1 & s_y &= 1 \\
  s_x &= -1 & s_y &= -1 \\
  s_x &= 1 & s_y &= -1
\end{align*} \]

Inverses

• Although we could compute inverse matrices by general formulas, we can use simple geometric observations
  - Translation: \( T^{-1}(d_x, d_y, d_z) = T(-d_x, -d_y, -d_z) \)
  - Rotation: \( R^{-1}(\theta) = R(\theta) \)
    - Holds for any rotation matrix
    - Note that since \( \cos(-\theta) = \cos(\theta) \) and \( \sin(-\theta) = -\sin(\theta) \)
  - Scaling: \( S^{-1}(s_x, s_y, s_z) = S(1/s_x, 1/s_y, 1/s_z) \)

Concatenation

• We can form arbitrary affine transformation matrices by multiplying together rotation, translation, and scaling matrices
• Because the same transformation is applied to many vertices, the cost of forming a matrix \( M = ABCD \) is not significant compared to the cost of computing \( Mp \) for many vertices \( p \)
• The difficult part is how to form a desired transformation from the specifications in the application

Order of Transformations

• Note that matrix on the right is the first applied
• Mathematically, the following are equivalent
  \( p' = ABCp = A(B(Cp)) \)
• Note many references use column matrices to present points. In terms of column matrices
  \( p^T' = p^T C^T B^T A^T \)

General Rotation About the Origin

A rotation by \( \theta \) about an arbitrary axis can be decomposed into the concatenation of rotations about the \( x, y, \) and \( z \) axes

\[ R(\theta) = R_x(\theta_y) R_y(\theta_x) R_z(\theta) \]

\( \theta, \theta_x, \theta_y, \theta_z \) are called the Euler angles

Note that rotations do not commute
We can use rotations in another order but with different angles

Rotation About a Fixed Point other than the Origin

Move fixed point to origin
Rotate
Move fixed point back

\( M = T(-p_f) R(\theta) T(p_f) \)
Instancing

- In modeling, we often start with a simple object centered at the origin, oriented with the axis, and at a standard size.
- We apply an instance transformation to its vertices to:
  - Scale
  - Orient
  - Locate

Shear

- Helpful to add one more basic transformation.
- Equivalent to pulling faces in opposite directions.

Shear Matrix

Consider simple shear along x axis:

\[
\begin{bmatrix}
1 & \cot \theta & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Example: Plotting a Function

- Draw a dot plot for the following function:

\[ f(x) = e^{-x} \cos(2\pi x) \text{ for } 0 \leq x \leq 4. \]

Viewport Transformation

- Use a linear transformation to fit the function in the window:

\[
\begin{bmatrix}
A & B \\
C & D
\end{bmatrix}
\begin{bmatrix}
x \\
y
\end{bmatrix}
= \begin{bmatrix}
x' \\
y'
\end{bmatrix}
\]

- Translates the origin to the center of the window.
- Scales the drawing to fit.

Scale and Translation

\[
\begin{bmatrix}
A & B \\
C & D
\end{bmatrix}
= \begin{bmatrix}
\frac{w}{4} & 0 \\
h/2 & h/2
\end{bmatrix}
\]

```cpp
A = screen_width / 4.0;
B = 0.0;
C = screen_height / 2.0;
D = C;

glBegin(GL_POINTS);
for(x = 0.0; x <= 4.0; x += 0.005)
glVertex2f(A*x + B, C*f(x) + D);
glEnd();
```
Objectives

- Learn how to carry out transformations in OpenGL
  - Rotation
  - Translation
  - Scaling
- Introduce OpenGL matrix modes
  - Model-view
  - Projection

OpenGL Matrices

- In OpenGL matrices are part of the state
- Three types:
  - Model-View (GL_MODEL_VIEW)
  - Projection (GL_PROJECTION)
  - Texture (GL_TEXTURE) (ignore for now)
- Single set of functions for manipulation
- Select which to manipulated by
  - glMatrixMode(GL_MODEL_VIEW);
  - glMatrixMode(GL_PROJECTION);

Current Transformation Matrix (CTM)

- Conceptually there is a 4 x 4 homogeneous coordinate matrix, the current transformation matrix (CTM) that is part of the state and is applied to all vertices that pass down the pipeline
- The CTM is defined in the user program and loaded into a transformation unit

CTM operations

- The CTM can be altered either by loading a new CTM or by postmultiplication
  - Load an identity matrix: C ← I
  - Load an arbitrary matrix: C ← M
  - Load a translation matrix: C ← T
  - Load a rotation matrix: C ← R
  - Load a scaling matrix: C ← S
  - Postmultiply by an arbitrary matrix: C ← CM
  - Postmultiply by a translation matrix: C ← CT
  - Postmultiply by a rotation matrix: C ← CR
  - Postmultiply by a scaling matrix: C ← CS

Rotation about a Fixed Point

- Start with identity matrix: C ← I
- Move fixed point to origin: C ← CT^{-1}
- Rotate: C ← CR
- Move fixed point back: C ← CT
- Result: C = T^{-1}RT

Each operation corresponds to one function call in the program.

Note that the last operation specified is the first executed in the program
CTM in OpenGL

- OpenGL has a model-view and a projection matrix in the pipeline which are concatenated together to form the CTM.
- Can manipulate each by first setting the matrix mode.

Rotation, Translation, Scaling

- Load an identity matrix: `glLoadIdentity()`
- Multiply on right:
  
  ```
  glRotatef(theta, vx, vy, vz)
  glTranslatef(dx, dy, dz)
  glScalef(sx, sy, sz)
  ```

  Each has a float (f) and double (d) format (`glScaled`).

Example

- Rotation about z-axis by 30 degrees with a fixed point of (1.0, 2.0, 3.0)
  
  ```
  glmMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glTranslatef(1.0, 2.0, 3.0);
  glRotatef(30.0, 0.0, 0.0, .10);
  glTranslatef(-1.0, -2.0, -3.0);
  ```

Arbitrary Matrices

- Can load and multiply by matrices defined in the application program
  
  ```
  glmLoadMatrixf(m);
  glmMultMatrixf(m);
  ```

  The matrix `m` is a one dimension array of 16 elements which are the components of the desired 4 x 4 matrix stored by columns.

  - In `glmMultMatrixf`, `m` multiplies the existing matrix on the right.

Matrix Stacks

- In many situations we want to save transformation matrices for use later.
  - Traversing hierarchical data structures (Chapter 9)
  - Avoiding state changes when executing display lists

  - OpenGL maintains stacks for each type of matrix.
  - Access present type (as set by `glmMatrixMode`) by
    
    ```
    glmPushMatrix()
    glmPopMatrix()
    ```

Reading Back Matrices

- Can also access matrices (and other parts of the state) by `enquiry` (query) functions
  
  ```
  glmGetIntegerv
  glmGetFloatv
  glmGetBooleanv
  glmGetDoublev
  glmIsEnabled
  ```

  - For matrices, we use as
    
    ```
    double m[16];
    glmGetFloatv(GL_MODELVIEW, m);
    ```
Using Transformations

- Example: use idle function to rotate a cube and mouse function to change direction of rotation
- Start with a program that draws a cube (colorcube.c) in a standard way
  - Centered at origin
  - Sides aligned with axes
  - Will discuss modeling in next lecture

Idle and Mouse callbacks

```c
void spinCube()
{
    theta[axis] += 2.0;
    if( theta[axis] > 360.0 ) theta[axis] -= 360.0;
    glutPostRedisplay();
}

void mouse(int btn, int state, int x, int y)
{
    if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN)
        axis = 0;
    if(btn==GLUT_MIDDLE_BUTTON && state == GLUT_DOWN)
        axis = 1;
    if(btn==GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
        axis = 2;
}
```

Display callback

```c
void display()
{
    glClearColor(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();
}
```

Note that because of fixed form of callbacks, variables such as theta and axis must be defined as globals

Camera information is in standard reshape callback

Using the Model-View Matrix

- In OpenGL the model-view matrix is used to
  - Position the camera
    - Can be done by rotations and translations but is often easier to use gluLookAt (Chapter 5)
  - Build models of objects
- The projection matrix is used to define the view volume and to select a camera lens
- Although both are manipulated by the same functions, we have to be careful because incremental changes are always made by postmultiplication
  - For example, rotating model-view and projection matrices by the same matrix are not equivalent operations.
  - Postmultiplication of the model-view matrix is equivalent to premultiplication of the projection matrix

Example: Solar System

- This example demonstrates how to combine transformations.
- Planets rotate on their axes and also orbit the sun.
- When issuing transformation commands, the order is significant.
Motion Control

• Use global variables to keep track of the planet’s orbit.
  ```
  static int year = 0, day = 0;
  ```
• Update the orbital state in response to events such as keyboard input.
  ```
  switch(key) {
    case 'd': day = (day + 10) % 360;
    glutPostRedisplay();
    break;
    case 'y': year = (year + 5) % 360;
    glutPostRedisplay();
    break;
    case 'Y': year = (year - 5) % 360;
    glutPostRedisplay();
    break;
    default: break;
  }
  ```

The Keyboard Handler

```
void keyboard(unsigned char key, int x, int y) {
switch(key) {
  case 'd': day = (day + 10) % 360;
  glutPostRedisplay();
  break;
  case 'D': day = (day - 10) % 360;
  glutPostRedisplay();
  break;
  case 'y': year = (year + 5) % 360;
  glutPostRedisplay();
  break;
  case 'Y': year = (year - 5) % 360;
  glutPostRedisplay();
  break;
  default: break;
}
```

Viewing Parameters

• Set up a perspective projection in the reshape handler.
  ```
glViewport(0, 0, w, h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(60.0, (float) w/h, 1.0, 20.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(0., 0., 5., 0., 0., 0., 0., 1., 0.);
```  
• Since the camera is stationary, we can set up the model-view matrix here as well.
  ```
glMatrixMode(GL_MODELVIEW);
gluLookAt(0., 0., 5., 0., 0., 0., 0., 1., 0.);
```  

The Reshape Handler

```
void reshape() {
  glViewport(0, 0, w, h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(60.0, (float) w/h, 1.0, 20.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(0., 0., 5., 0., 0., 0., 0., 1., 0.);
}
```  

Ordering the Transformations

• An initial rotation around the sun is determined by the time of year.  
• Translation along the x-axis moves the planet to the correct spot along its orbit.  
• A second rotation based on the time of day rotates the local coordinate axis.

The Display Handler

```
void display(void) {
  glClear (GL_COLOR_BUFFER_BIT);
  glColor3f(1.0, 1.0, 1.0);
  glPushMatrix();
  glutWireSphere(1.0, 20, 16);
  glRotatef ((GLfloat) year, 0.0, 1.0, 0.0);
  glTranslatef (2.0, 0.0, 0.0);
  glutWireSphere(0.2, 10, 8);
  glPopMatrix();
  glutSwapBuffers();
}
```
void init(void) {
    glClearColor(0., 0., 0., 0.);
    glShadeModel(GL_FLAT);
}

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

• From a practical standpoint, we are often want to use transformations to move and reorient an object smoothly.
  - Problem: find a sequence of model-view matrices $M_0, M_1, \ldots, M_n$ so that when they are applied successively to one or more objects we see a smooth transition.
  - For orientating an object, we can use the fact that every rotation corresponds to part of a great circle on a sphere.
    - Find the axis of rotation and angle.
    - Virtual trackball (see text).

• Consider the two approaches:
  - For a sequence of rotation matrices $R_0, R_1, \ldots, R_n$, find the Euler angles for each and use $R_i = R_n R_i R_n$.
    - Not very efficient.
  - Use the final positions to determine the axis and angle of rotation, then increment only the angle.
  - Quaternions can be more efficient than either.

• Extension of imaginary numbers from two to three dimensions.
  • Requires one real and three imaginary components $i, j, k$.
    $$ q = q_0 + q_1 i + q_2 j + q_3 k $$
  • Quaternions can express rotations on sphere smoothly and efficiently. Process:
    - Model-view matrix $\rightarrow$ quaternion.
    - Carry out operations with quaternions.
    - Quaternion $\rightarrow$ Model-view matrix.

• One of the major problems in interactive computer graphics is how to use two-dimensional devices such as a mouse to interface with three dimensional objects.
  • Example: how to form an instance matrix?
  • Some alternatives:
    - Virtual trackball.
    - 3D input devices such as the spaceball.
    - Use areas of the screen.
      - Distance from center controls angle, position, scale depending on mouse button depressed.

Building Models

Prof. George Wolberg
Dept. of Computer Science
City College of New York
Objectives

• Introduce simple data structures for building polygonal models
  - Vertex lists
  - Edge lists
• OpenGL vertex arrays

Representing a Mesh

• Consider a mesh

• There are 8 nodes and 12 edges
  - 5 interior polygons
  - 6 interior (shared) edges
• Each vertex has a location \( v_i = (x_i, y_i, z_i) \)

Simple Representation

• List all polygons by their geometric locations
• Leads to OpenGL code such as
  ```
  glBegin(GL_POLYGON);
  glVertex3f(x1, y1, z1);
  glVertex3f(x6, y6, z6);
  glVertex3f(x7, y7, z7);
  glEnd();
  ```
• Inefficient and unstructured
  - Consider moving a vertex to a new locations

Inward and Outward Facing Polygons

• The order \( \{v_0, v_1, v_2, v_3\} \) and \( \{v_1, v_2, v_3, v_0\} \) are equivalent in that the same polygon will be rendered by OpenGL but the order \( \{v_3, v_2, v_1, v_0\} \) is different
• The first two describe inwardly facing polygons
• Use the right-hand rule = counter-clockwise encirclement of outward-pointing normal
• OpenGL treats inward and outward facing polygons differently

Geometry vs Topology

• Generally it is a good idea to look for data structures that separate the geometry from the topology
  - Geometry: locations of the vertices
  - Topology: organization of the vertices and edges
  - Example: a polygon is an ordered list of vertices with an edge connecting successive pairs of vertices and the last to the first
  - Topology holds even if geometry changes

Vertex Lists

• Put the geometry in an array
• Use pointers from the vertices into this array
• Introduce a polygon list
Shared Edges

- Vertex lists will draw filled polygons correctly but if we draw the polygon by its edges, shared edges are drawn twice
- Can store mesh by edge list

Modeling a Cube

Model a color cube for rotating cube program

Define global arrays for vertices and colors

```c
GLfloat vertices[][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}};
```

```c
GLfloat colors[][3] = {{0.0,0.0,0.0},{1.0,0.0,0.0},
{1.0,1.0,0.0}, {0.0,1.0,0.0}, {0.0,0.0,1.0},
{1.0,0.0,1.0}, {1.0,1.0,1.0}, {0.0,1.0,1.0}};
```

Draw cube from faces

```c
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);
}
```

Efficiency

- The weakness of our approach is that we are building the model in the application and must do many function calls to draw the cube
- Drawing a cube by its faces in the most straight forward way requires
  - `6 glBegin, 6 glEnd`
  - `6 glColor`
  - `24 glVertex`
  - More if we use texture and lighting
**Vertex Arrays**

- OpenGL provides a facility called *vertex arrays* that allow us to store array data in the implementation.
- Six types of arrays supported:
  - Vertices
  - Colors
  - Color indices
  - Normals
  - Texture coordinates
  - Edge flags
- We will need only colors and vertices.

**Initialization**

- Using the same color and vertex data, first we enable:
  ```c
  glEnableClientState(GL_COLOR_ARRAY);
  glEnableClientState(GL_VERTEX_ARRAY);
  ```
- Identify location of arrays:
  ```c
  glVertexPointer(3, GL_FLOAT, 0, vertices);
  glColorPointer(3, GL_FLOAT, 0, colors);
  ```

**Mapping indices to faces**

- Form an array of face indices:
  ```c
  GLuint cubeIndices[24] = {0, 3, 2, 1, 2, 3, 7, 6, 0, 4, 7, 3, 1, 2, 6, 5, 4, 5, 6, 7, 0, 1, 5, 4};
  ```
- Each successive four indices describe a face of the cube.
- Draw through `glDrawElements` which replaces all `glVertex` and `glColor` calls in the display callback.

**Drawing the cube**

- Method 1:
  ```c
  for(i = 0; i < 6; i++)
   glDrawElements(GL_POLYGON, 4, GL_UNSIGNED_BYTE, &cubeIndices[4*i]);
  ```

**Example: Cube Using Quad Primitives**

- OpenGL fills the polygon by interpolating among the vertex colors.

```c
  glBegin(GL_QUADS);
  for(i = 0; i < nfacs; i++) {
    for(j = 0; j < 4; j++) {
      glColor3fv(colors[faces[i][j]]);
      glVertex3fv(vertices[faces[i][j]]);
    }
  }
  glEnd();
```

**Cube Using Vertex Arrays**

- Flatten the data to one-dimensional arrays:
  ```c
  float verts[24] = {-1.0, -1.0, -1.0, 1.0, -1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
  float faces[24] = {0, 3, 2, 1, 2, 3, 7, 6, ...
  ```
- Looping over the face list is replaced by single function call:
  ```c
  glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, faces);
  ```
Objectives

- Introduce the mathematics of projection
- Introduce OpenGL viewing functions
- Look at alternate viewing APIs

There are three aspects of the viewing process, all of which are implemented in the pipeline,
- Positioning the camera
  - Setting the model-view matrix
- Selecting a lens
  - Setting the projection matrix
- Clipping
  - Setting the view volume

In OpenGL, initially the world and camera frames are the same
- Default model-view matrix is an identity
- The camera is located at origin and points in the negative z direction
- OpenGL also specifies a default view volume that is a cube with sides of length 2 centered at the origin
  - Default projection matrix is an identity

Default projection is orthogonal
- Move the camera in the positive z direction
  - Translate the camera frame
- Move the objects in the negative z direction
  - Translate the world frame
- Both of these views are equivalent and are determined by the model-view matrix
  - Want a translation (\texttt{glTranslatef(0.0,0.0,-d);})
  - \(d > 0\)
Moving Camera back from Origin

frames after translation by \(-d\)

\(d > 0\)

default frames

Moving the Camera

• We can move the camera to any desired position by a sequence of rotations and translations
• Example: side view
  - Rotate the camera
  - Move it away from origin
  - Model-view matrix \(C = TR\)

OpenGL code

• Remember that last transformation specified is first to be applied

```glMatrixMode(GL_MODELVIEW)
glLoadIdentity();
glTranslatef(0.0, 0.0, -d);
glRotatef(90.0, 0.0, 1.0, 0.0);```

The LookAt Function

• The GLU library contains the function glLookAt to from the required modelview matrix through a simple interface
• Note the need for setting an up direction
• Still need to initialize
  - Can concatenate with modeling transformations
• Example: isometric view of cube aligned with axes

```glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0. 0.0);```

Other Viewing APIs

• The LookAt function is only one possible API for positioning the camera
• Others include
  - View reference point, view plane normal, view up (PHIGS, GKS-3D)
  - Yaw, pitch, roll
  - Elevation, azimuth, twist
  - Direction angles
Projections and Normalization

- The default projection in the eye (camera) frame is orthogonal.
- For points within the default view volume:
  \[ x' = x \]
  \[ y' = y \]
  \[ z' = 0 \]
- Most graphics systems use view normalization:
  - All other views are converted to the default view by transformations that determine the projection matrix.
  - Allows use of the same pipeline for all views.

Homogeneous Coordinate Representation

- For points within the default view volume:
  \[ x' = x \]
  \[ y' = y \]
  \[ z' = 0 \]
  \[ w' = 1 \]

\[ M = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix} \]

In practice, we can let \( M = I \) and set the \( z \) term to zero later.

Simple Perspective

- Center of projection at the origin.
- Projection plane \( z = d, d < 0 \).

Perspective Equations

Consider top and side views:

\[ x' = \frac{x}{z/d} \]
\[ y' = \frac{y}{z/d} \]
\[ z' = \frac{d}{z/d} \]

Homogeneous Coordinate Form

- Consider \( q = Mp \):

\[ M = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1/d & 0 \\
\end{bmatrix} \]

- \[ p = \begin{bmatrix}
x \\
y \\
z \\
1 \\
\end{bmatrix} \Rightarrow q = \begin{bmatrix}
x \\
y \\
z \\
1 \\
\end{bmatrix} \]

Perspective Division

- However \( w \neq 1 \), so we must divide by \( w \) to return from homogeneous coordinates.

- This perspective division yields

\[ x' = \frac{x}{z/d} \]
\[ y' = \frac{y}{z/d} \]
\[ z' = \frac{d}{z/d} \]

the desired perspective equations.

- We will consider the corresponding clipping volume with the OpenGL functions.

- In practice, we can let \( M = I \) and set the \( z \) term to zero later.
OpenGL Orthogonal Viewing

\texttt{glOrtho(xmin, xmax, ymin, ymax, near, far)}
\texttt{glOrtho(left, right, bottom, top, near, far)}

near and far measured from camera

OpenGL Perspective

\texttt{glFrustum(xmin, xmax, ymin, ymax, near, far)}

Using Field of View

- With \texttt{glFrustum} it is often difficult to get the desired view
- \texttt{gluPerspective(fovy, aspect, near, far)} often provides a better interface

Projection Matrices

Prof. George Wolberg
Dept. of Computer Science
City College of New York

Objectives

- Derive the projection matrices used for standard OpenGL projections
- Introduce oblique projections
- Introduce projection normalization

Normalization

- Rather than derive a different projection matrix for each type of projection, we can convert all projections to orthogonal projections with the default view volume
- This strategy allows us to use standard transformations in the pipeline and makes for efficient clipping
Pipeline View

- **modelview transformation**
- **projection transformation**
- **perspective division**
- 4D \( \rightarrow \) 3D
- **clipping**
- **projection**
- 3D \( \rightarrow \) 2D
- against default cube

Notes

- We stay in four-dimensional homogeneous coordinates through both the modelview and projection transformations
  - Both these transformations are nonsingular
  - Default to identity matrices (orthogonal view)
- Normalization lets us clip against simple cube regardless of type of projection
- Delay final projection until end
  - Important for hidden-surface removal to retain depth information as long as possible

Orthogonal Normalization

\[ \text{glOrtho}(\text{left}, \text{right}, \text{bottom}, \text{top}, \text{near}, \text{far}) \]

- Normalization \( \Rightarrow \) find transformation to convert specified clipping volume to default
- \( \text{normalization} \Rightarrow [1,1,1] \)
- \( \text{left}, \text{bottom}, \text{near} \) \( \Rightarrow [-1,-1,1] \)
- \( \text{right}, \text{top}, \text{far} \) \( \Rightarrow [1,1,1] \)

Orthogonal Matrix

- Two steps
  - Move center to origin
    \( Ti((\text{left}+\text{right})/2, (\text{bottom}+\text{top})/2, (\text{near}+\text{far})/2) \)
  - Scale to have sides of length 2
    \( S(2/(\text{left}-\text{right}), 2/(\text{top}-\text{bottom}), 2/(\text{near}-\text{far})) \)

\[ \begin{bmatrix}
  1 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 \\
  0 & 0 & 1 & 0 \\
  0 & 0 & 0 & 1 \\
\end{bmatrix} \]

\[ P = ST \]

Final Projection

- Set \( z = 0 \)
- Equivalent to the homogeneous coordinate transformation
  \[ M_{\text{final}} = \begin{bmatrix}
  1 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 \\
  0 & 0 & 1 & 0 \\
  0 & 0 & 0 & 1 \\
\end{bmatrix} \]
- Hence, general orthogonal projection in \( 4\mathbb{R}^3 \)

Oblique Projections

- The OpenGL projection functions cannot produce general parallel projections such as

- However if we look at the example of the cube it appears that the cube has been sheared
- Oblique Projection = Shear + Orthogonal Projection
General Shear

![Shear Diagram](Image)

**Shear Matrix**

\[ H(\theta, \phi) = \begin{bmatrix} 1 & 0 & -\cot \theta & 0 \\ 0 & 1 & -\cot \phi & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

Shear matrix with \( z \) values unchanged.

Projection matrix \( P = M_{\text{orth}} H(\theta, \phi) \)

General case: \( P = M_{\text{orth}} STH(\theta, \phi) \)

Equivalency

![Equivalency Diagram](Image)

**Effect on Clipping**

- The projection matrix \( P = STH \) transforms the original clipping volume to the default clipping volume.

![Clipping Diagram](Image)

Simple Perspective

Consider a simple perspective with the COP at the origin, the near clipping plane at \( z = -1 \), and a 90 degree field of view determined by the planes \( x = \pm \pi, y = \pm \pi \).

![Perspective Diagram](Image)

**Perspective Matrices**

Simple projection matrix in homogeneous coordinates

\[ M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix} \]

Note that this matrix is independent of the far clipping plane.
Generalization

\[
N = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & \beta \\
0 & 0 & -1 & 0
\end{bmatrix}
\]

after perspective division, the point \((x, y, z, 1)\) goes to

\[
x'' = x/z \\
y'' = y/z \\
z'' = -(\alpha + \beta/z)
\]

which projects orthogonally to the desired point regardless of \(\alpha\) and \(\beta\)

Picking \(\alpha\) and \(\beta\)

If we pick

\[
\alpha = \frac{\text{near} + \text{far}}{\text{far} - \text{near}} \\
\beta = \frac{2\text{near} \ast \text{far}}{\text{near} - \text{far}}
\]

the near plane is mapped to \(z = -1\)
the far plane is mapped to \(z = 1\)
and the sides are mapped to \(x = \pm 1, y = \pm 1\)

Hence the new clipping volume is the default clipping volume

Normalization Transformation

Normalization and Hidden-Surface Removal

• Although our selection of the form of the perspective matrices may appear somewhat arbitrary, it was chosen so that if \(z_1 > z_2\) in the original clipping volume then the for the transformed points \(z_1' > z_2'\)
• Thus hidden surface removal works if we first apply the normalization transformation
• However, the formula \(z'' = -(\alpha + \beta/z)\) implies that the distances are distorted by the normalization which can cause numerical problems especially if the near distance is small

OpenGL Perspective

• \texttt{glFrustum} allows for an asymmetric viewing frustum (although \texttt{gluPerspective} does not)

OpenGL Perspective Matrix

• The normalization in \texttt{glFrustum} requires an initial shear to form a right viewing pyramid, followed by a scaling to get the normalized perspective volume. Finally, the perspective matrix results in needing only a final orthogonal transformation
Why do we do it this way?

- Normalization allows for a single pipeline for both perspective and orthogonal viewing
- We keep in four dimensional homogeneous coordinates as long as possible to retain three-dimensional information needed for hidden-surface removal and shading
- We simplify clipping

Example: Shadows

- In this example, we model a light casting shadows on a cube.

\[
M = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 1/y_l & 0 & 0
\end{bmatrix}
\]

- \(M\) is a perspective projection matrix.

Specifying the Light Position

- Compute the light position in the display handler.
- Update global \(\theta\) in an animation loop to make the light rotate around the origin.

```c
GLfloat light[3]; /* light position */
light[0] = 10.0 * sin((PI2/180.0) * theta);
light[1] = 10.0;
light[2] = 10.0 * cos((PI2/180.0) * theta);
```

The Shadow Transformation

- Compute the shadow transformation based on the light position.

```c
/* start with the identity matrix */
GLfloat m[16] = {
    1.0, 0.0, 0.0, 0.0,
    0.0, 1.0, 0.0, 0.0,
    0.0, 0.0, 1.0, 0.0,
    0.0, 0.0, 0.0, 1.0,
};
m[7] = -1.0/light[1]; /* y_l */
m[15] = 0.0;
```

Drawing Shadows

- After drawing the scene, move the viewpoint to the light position and apply the shadow transformation.

```c
glPushMatrix();
glTranslatef(light[0], light[1], light[2]);
glMultMatrixf(m);
glTranslatef(-light[0], -light[1], -light[2]);
glColor3f(0.0, 0.0, 0.0); /* shadow color */
drawPolygon();
glPopMatrix();
```

Shading in OpenGL

Prof. George Wolberg
Dept. of Computer Science
City College of New York
Objectives

- Introduce the OpenGL shading functions
- Discuss polygonal shading
  - Flat
  - Smooth
  - Gouraud

Steps in OpenGL shading

1. Enable shading and select model
2. Specify normals
3. Specify material properties
4. Specify lights

Normals

- In OpenGL the normal vector is part of the state
- Set by `glNormal*()`
  - `glNormal3f(x, y, z);`
  - `glNormal3fv(p);`
- Usually we want to set the normal to have unit length so cosine calculations are correct
  - Length can be affected by transformations
  - Note the scale does not preserve length
  - `glEnable(GL_NORMALIZE)` allows for autonormalization at a performance penalty

Normal for Triangle

\[
\text{plane } n \cdot (p - p_0) = 0
\]
\[
n = (p_1 - p_0) \times (p_2 - p_0)
\]

normalize \( n \leftarrow \frac{n}{|n|} \)

Note that right-hand rule determines outward face

Enabling Shading

- Shading calculations are enabled by
  - `glEnable(GL_LIGHTING)`
  - Once lighting is enabled, `glColor()` ignored
- Must enable each light source individually
  - `glEnable(GL_LIGHTi) i=0,1,...`
- Can choose light model parameters
  - `glLightModeli(parameter, GL_TRUE)`
    - `GL_LIGHT_MODEL_LOCAL_VIEWER` do not use simplifying distant viewer assumption in calculation
    - `GL_LIGHT_MODEL_TWO_SIDED` shades both sides of polygons independently

Defining a Point Light Source

- For each light source, we can set an RGB for the diffuse, specular, and ambient parts, and the position
  - `GL float diffuse0[]={1.0, 0.0, 0.0, 1.0};`
  - `GL float ambient0[]={1.0, 0.0, 0.0, 1.0};`
  - `GL float specular0[]={1.0, 0.0, 0.0, 1.0};`
  - `GLfloat light0_pos0[]={1.0, 2.0, 3.0, 1.0};`
  - `glEnable(GL_LIGHTING);`
  - `glEnable(GL_LIGHT0);`
  - `glLightv(GL_LIGHT0, GL_POSITION, light0_pos0);`
  - `gLightr(GL_LIGHT0, GL_DIFUSE, diffuse0);`
  - `gLightr(GL_LIGHT0, GL_DIFFUSE, diffuse0);`
  - `gLightr(GL_LIGHT0, GL_SPECULAR, specular0);`
Distance and Direction

- The source colors are specified in RGBA
- The position is given in homogeneous coordinates
  - If \( w = 1.0 \), we are specifying a finite location
  - If \( w = 0.0 \), we are specifying a parallel source with the given direction vector
- The coefficients in the distance terms are by default \( a = 1.0 \) (constant terms), \( b = c = 0.0 \) (linear and quadratic terms). Change by

  \[
  a = 0.80;
  \text{glLightf(GL_LIGHT0, GLCONSTANT_ATTENUATION, a)};
  \]

Spotlights

- Use `glLightv` to set
  - Direction `GL_SPOT_DIRECTION`
  - Cutoff `GL_SPOT_CUTOFF`
  - Attenuation `GL_SPOT_EXPONENT`
    - Proportional to \( \cos \alpha \)

Global Ambient Light

- Ambient light depends on color of light sources
  - A red light in a white room will cause a red ambient term that disappears when the light is turned off
- OpenGL allows a global ambient term that is often helpful
  - `glLightModelfv(GL_LIGHT_MODEL_AMBIENT, global_ambient)`

Moving Light Sources

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

Material Properties

- Material properties are also part of the OpenGL state and match the terms in the Phong model
  - Set by `glMaterialv()`

  ```c
  GLfloat ambient[] = {0.2, 0.2, 0.2, 1.0};
  GLfloat diffuse[] = {1.0, 0.8, 0.0, 1.0};
  GLfloat specular[] = {1.0, 1.0, 1.0, 1.0};
  GLfloat shine = 100.0
  glMaterialf(GL_FRONT, GL_AMBIENT, ambient);
  glMaterialf(GL_FRONT, GL_DIFFUSE, diffuse);
  glMaterialf(GL_FRONT, GL_SPECULAR, specular);
  glMaterialf(GL_FRONT, GL_SHININESS, shine);
  ```

Front and Back Faces

- The default is shade only front faces which works correct for convex objects
  - If we set two sided lighting, OpenGL will shade both sides of a surface
  - Each side can have its own properties which are set by using `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK` in `glMaterialf`
Emissive Term

- We can simulate a light source in OpenGL by giving a material an emissive component.
- This color is unaffected by any sources or transformations.

```c
GLfloat emission[] = 0.0, 0.3, 0.3, 1.0;
glMaterialf(GL_FRONT, GL_EMISSION, emission);
```

Transparency

- Material properties are specified as RGBA values.
- The A value can be used to make the surface translucent.
- The default is that all surfaces are opaque regardless of A.
- Later we will enable blending and use this feature.

Efficiency

- Because material properties are part of the state, if we change materials for many surfaces, we can affect performance.
- We can make the code cleaner by defining a material structure and setting all materials during initialization.

```c
typedef struct materialStruct {
    GLfloat ambient[4];
    GLfloat diffuse[4];
    GLfloat specular[4];
    GLfloat shininess;
} MaterialStruct;
```

- We can then select a material by a pointer.

Polygonal Shading

- Shading calculations are done for each vertex.
  - Vertex colors become vertex shades.
- By default, vertex colors are interpolated across the polygon.
  ```c
  glShadeModel(GL_SMOOTH);
  ```
- If we use `glShadeModel(GL_FLAT);` the color at the first vertex will determine the color of the whole polygon.

Polygon Normals

- Polygons have a single normal.
  - Shades at the vertices as computed by the Phong model can be almost same.
  - Identical for a distant viewer (default) or if there is no specular component.
- Consider model of sphere.
- Want different normals at each vertex even though this concept is not quite correct mathematically.

Smooth Shading

- We can set a new normal at each vertex.
- Easy for sphere model.
  - If centered at origin \( n = p \).
- Now smooth shading works.
- Note silhouette edge.
Mesh Shading

- The previous example is not general because we knew the normal at each vertex analytically.
- For polygonal models, Gouraud proposed we use the average of normals around a mesh vertex.

\[ n = \frac{n_1 + n_2 + n_3 + n_4}{|n_1| + |n_2| + |n_3| + |n_4|} \]

Gouraud and Phong Shading

- Gouraud Shading
  - Find average normal at each vertex (vertex normals)
  - Apply Phong model at each vertex
  - Interpolate vertex shades across each polygon
- Phong shading
  - Find vertex normals
  - Interpolate vertex normals across edges
  - Find shades along edges
  - Interpolate edge shades across polygons

Comparison

- If the polygon mesh approximates surfaces with a high curvatures, Phong shading may look smooth while Gouraud shading may show edges.
- Phong shading requires much more work than Gouraud shading.
  - Usually not available in real time systems.
- Both need data structures to represent meshes so we can obtain vertex normals.

Objectives

- Introduce additional OpenGL buffers
- Learn to read and write buffers
- Learn to use blending

Buffers

Prof. George Wolberg
Dept. of Computer Science
City College of New York
OpenGL Buffers

- Color buffers can be displayed
  - Front
  - Back
  - Auxiliary
  - Overlay
- Depth
- Accumulation
  - High resolution buffer
- Stencil
  - Holds masks

OpenGL Frame Buffer

Writing in Buffers

- Conceptually, we can consider all of memory as a large two-dimensional array of pixels
- We read and write rectangular block of pixels
  - Bit block transfer (bitblt) operations
- The frame buffer is part of this memory

Writing Modes

- Source and destination bits are combined bitwise
- 16 possible functions (one per column in table)

XOR mode

- Recall from Chapter 3 that we can use XOR by enabling logic operations and selecting the XOR write mode
- XOR is especially useful for swapping blocks of memory such as menus that are stored off screen
  - If S represents screen and M represents a menu
  - The sequence
    
    S ← S ⊕ M
    M ← S ⊕ M
    S ← S ⊕ M

  - Swaps the S and M
The Pixel Pipeline

- OpenGL has a separate pipeline for pixels
  - Writing pixels involves
    - Moving pixels from processor memory to the frame buffer
    - Format conversions
    - Mapping, Lookups, Tests
  - Reading pixels
    - Format conversion

Raster Position

- OpenGL maintains a raster position as part of the state
- Set by `glRasterPos*()`
  - `glRasterPos3f(x, y, z);`
- The raster position is a geometric entity
  - Passes through geometric pipeline
  - Eventually yields a 2D position in screen coordinates
  - This position in the frame buffer is where the next raster primitive is drawn

Buffer Selection

- OpenGL can draw into or read from any of the color buffers (front, back, auxiliary)
- Default to the back buffer
- Change with `glDrawBuffer` and `glReadBuffer`
- Note that format of the pixels in the frame buffer is different from that of processor memory and these two types of memory reside in different places
  - Need packing and unpacking
  - Drawing and reading can be slow

Bitmaps

- OpenGL treats 1-bit pixels (bitmaps) differently than multi-bit pixels (pixelmaps)
- Bitmaps are masks which determine if the corresponding pixel in the frame buffer is drawn with the present raster color
  - 0 ⇒ color unchanged
  - 1 ⇒ color changed based on writing mode
- Bitmaps are useful for raster text
  - `GLUT_BITMAP_8_BY_13`

Raster Color

- Same as drawing color set by `glColor*()`
- Fixed by last call to `glRasterPos*()`
  - `glColor3f(1.0, 0.0, 0.0);`
  - `glRasterPos3f(x, y, z);`
  - `glColor3f(0.0, 0.0, 1.0);`
  - `glBitmap(...)`
  - `glBegin(GL_LINES);`
  - `glVertex3f(....)`
- Geometry drawn in blue
- Ones in bitmap use a drawing color of red

Drawing Bitmaps

- `glBitmap(width, height, x0, y0, xi, yi, bitmap)`
Example: Checker Board

```c
GLubyte wb[2] = {0x00, 0x07};
GLubyte check[512];
int i, j;
for(i=0; i<64; i++) for (j=0; j<64; j++)
  check[i*8+j] = wb[(i/8+j)%2];
glBitmap( 64, 64, 0.0, 0.0, 0.0, 0.0, check);
```

Pixel Maps

- OpenGL works with rectangular arrays of pixels called pixel maps or images
- Pixels are in one byte (8 bit) chunks
  - Luminance (gray scale) images 1 byte/pixel
  - RGB 3 bytes/pixel
- Three functions
  - Draw pixels: processor memory to frame buffer
  - Read pixels: frame buffer to processor memory
  - Copy pixels: frame buffer to frame buffer

OpenGL Pixel Functions

```c
glReadPixels(x, y, width, height, format, type, myimage)
```

start pixel in frame buffer size type of pixels type of image pointer to processor memory

```c
GLubyte myimage[512][512][3];
glReadPixels(0, 0, 512, 512, GL_RGB, GL_UNSIGNED_BYTE, myimage);
glDrawPixels(width, height, format, type, myimage)
```

Image Formats

- We often work with images in a standard format (JPEG, TIFF, GIF)
- How do we read/write such images with OpenGL?
- No support in OpenGL
  - OpenGL knows nothing of image formats
  - Some code available on Web
  - Can write readers/writers for some simple formats in OpenGL

Displaying a PPM Image

- PPM is a very simple format
- Each image file consists of a header followed by all the pixel data
- Header

```
P3
# comment 1
# comment 2
#comment n
rows columns maxvalue
pixels
```

Reading the Header

```c
FILE *fd;
int k, nm;
char c;
int i;
char b[100];
float s;
int red, green, blue;
printf("enter file name
");
scanf("%s", b);
fd = fopen(b, "r");
 fscanf(fd,"%[^n] ",b);
if(b[0]!="P" || b[1] != '3') {
  printf("%s is not a PPM file\n", b);
  exit(0);
}
printf("%s is a PPM file\n", b);
```
Reading the Header (cont)

fscanf(fd, "%c", &c);
while (c == '#')
{
    fscanf(fd, "%[^\n]", b);
    printf("%s\n", b);
    fscanf(fd, "%c", &c);
}
ungetc(c, fd);

skip over comments by looking for # in first column

Reading the Data

fscanf(fd, "%d %d %d", &n, &m, &k);
printf("%d rows  %d columns  max value= %d\n", n, m, k);
nm = n*m;
image = malloc(3*sizeof(GLuint)*nm);
s = 255./k;
for (i=0; i<nm; i++)
{
    fscanf(fd, "%d %d %d", &red, &green, &blue);
    image[3*nm-3*i-3] = red;
    image[3*nm-3*i-2] = green;
    image[3*nm-3*i-1] = blue;
}

Scaling the Image Data

We can scale the image in the pipeline

    glPixelTransferf(GL_RED_SCALE, s);
    glPixelTransferf(GL_GREEN_SCALE, s);
    glPixelTransferf(GL_BLUE_SCALE, s);

We may have to swap bytes when we go from
processor memory to the frame buffer depending on
the processor. If so we need can use

    glPixelStorei(GL_UNPACK_SWAP_BYTES, GL_TRUE);

The display callback

void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glRasterPos2i(0, 0);
    glDrawPixels(n, m, GL_RGB,
                 GL_UNSIGNED_INT, image);
    glFlush();
}

Objectives

- Introduce Mapping Methods
  - Texture Mapping
  - Environmental Mapping
  - Bump Mapping
- Consider basic strategies
  - Forward vs backward mapping
  - Point sampling vs area averaging
The Limits of Geometric Modeling

• Although graphics cards can render over 10 million polygons per second, that number is insufficient for many phenomena
  - Clouds
  - Grass
  - Terrain
  - Skin

Modeling an Orange

• Consider the problem of modeling an orange (the fruit)
  - Start with an orange-colored sphere
    - Too simple
  - Replace sphere with a more complex shape
    - Does not capture surface characteristics (small dimples)
    - Takes too many polygons to model all the dimples

Modeling an Orange (2)

• Take a picture of a real orange, scan it, and “paste” onto simple geometric model
  - This process is texture mapping
• Still might not be sufficient because resulting surface will be smooth
  - Need to change local shape
  - Bump mapping

Three Types of Mapping

• Texture Mapping
  - Uses images to fill inside of polygons
• Environmental (reflection mapping)
  - Uses a picture of the environment for texture maps
  - Allows simulation of highly specular surfaces
• Bump mapping
  - Emulates altering normal vectors during the rendering process

Texture Mapping

geometric model  texture mapped

Environment Mapping
**Bump Mapping**

Where does mapping take place?

- Mapping techniques are implemented at the end of the rendering pipeline
  - Very efficient because few polygons pass down the geometric pipeline

**Where does mapping take place?**

- Mapping techniques are implemented at the end of the rendering pipeline
  - Very efficient because few polygons pass down the geometric pipeline

**Is it simple?**

- Although the idea is simple—map an image to a surface—there are 3 or 4 coordinate systems involved

**Coordinate Systems**

- Parametric coordinates
  - May be used to model curved surfaces
- Texture coordinates
  - Used to identify points in the image to be mapped
- World Coordinates
  - Conceptually, where the mapping takes place
- Screen Coordinates
  - Where the final image is really produced

**Texture Mapping**

Mapping Functions

- Basic problem is how to find the maps
- Consider mapping from texture coordinates to a point a surface
- Appear to need three functions
  
  \[
  x = x(s,t) \\
  y = y(s,t) \\
  z = z(s,t)
  \]

- But we really want to go the other way
Backward Mapping

- We really want to go backwards
  - Given a pixel, we want to know to which point on an object it corresponds
  - Given a point on an object, we want to know to which point in the texture it corresponds
- Need a map of the form
  \[ s = s(x,y,z) \]
  \[ t = t(x,y,z) \]
- Such functions are difficult to find in general

Two-part mapping

- One solution to the mapping problem is to first map the texture to a simple intermediate surface
- Example: map to cylinder

Cylindrical Mapping

parametric cylinder

\[ x = r \cos 2\pi u \]
\[ y = r \sin 2\pi u \]
\[ z = v/h \]

maps rectangle in \( u,v \) space to cylinder of radius \( r \) and height \( h \) in world coordinates

\[ s = u \]
\[ t = v \]

maps from texture space

Spherical Map

We can use a parametric sphere

\[ x = r \cos 2\pi u \]
\[ y = r \sin 2\pi u \cos 2\pi v \]
\[ z = r \sin 2\pi u \sin 2\pi v \]

in a similar manner to the cylinder but have to decide where to put the distortion

Spheres are use in environmental maps

Box Mapping

- Easy to use with simple orthographic projection
- Also used in environmental maps

Second Mapping

- Map from intermediate object to actual object
  - Normals from intermediate to actual
  - Normals from actual to intermediate
  - Vectors from center of intermediate
Aliasing

Point sampling of the texture can lead to aliasing errors

Point samples in u,v (or x,y,z) space

Point samples in texture space

Area Averaging

A better but slower option is to use area averaging

Note that preimage of pixel is curved

OpenGL Texture Mapping

Prof. George Wolberg
Dept. of Computer Science
City College of New York

Objectives

Introduce the OpenGL texture functions and options

Basic Strategy

Three steps to applying a texture
1. specify the texture
   - read or generate image
   - assign to texture
   - enable texturing
2. assign texture coordinates to vertices
   - Proper mapping function is left to application
3. specify texture parameters
   - wrapping, filtering

Texture Mapping
Texture Example

- The texture (below) is a 256 x 256 image that has been mapped to a rectangular polygon which is viewed in perspective.

Texture Mapping and the OpenGL Pipeline

- Images and geometry flow through separate pipelines that join at the rasterizer.
  - "Complex" textures do not affect geometric complexity.

Specify Texture Image

- Define a texture image from an array of texels (texture elements) in CPU memory:
  ```c
  Glubyte my_texels[512][512];
  ```
- Define as any other pixel map:
  - Scan
  - Via application code
- Enable texture mapping:
  ```c
  glEnable(GL_TEXTURE_2D);
  ```
- OpenGL supports 1-4 dimensional texture maps.

Define Image as a Texture

```c
glTexImage2D( target, level, components, w, h, border, format, type, texels );
```
- **target**: type of texture, e.g. GL_TEXTURE_2D
- **level**: used for mipmapping (discussed later)
- **components**: elements per texel
- **w, h**: width and height of texels in pixels
- **border**: used for smoothing (discussed later)
- **format and type**: describe texels
- **texels**: pointer to texel array

```c
glTexImage2D(GL_TEXTURE_2D, 0, 3, 512, 512, 0, GL_RGB, GL_UNSIGNED_BYTE, my_texels);
```

Converting A Texture Image

- OpenGL requires texture dimensions to be powers of 2.
- If dimensions of image are not powers of 2:
  ```c
  gluScaleImage( format, w_in, h_in, type_in, *data_in, w_out, h_out, type_out, *data_out );
  ```
  - **data_in** is source image
  - **data_out** is destination image
- Image interpolated and filtered during scaling.

Mapping a Texture

- Based on parametric texture coordinates
  - `glTexCoord*()` specified at each vertex.

```c
0, 1 1, 1 0, 0
```

Texture Space vs. Object Space

- Texture coordinates to object coordinates:
  ```c
  (s, t) = (0.2, 0.8)
  ```
Typical Code

```c
glBegin(GL_POLYGON);
glColor3f(r0, g0, b0);
glNormal3f(u0, v0, w0);
glTexCoord2f(s0, t0);
glVertex3f(x0, y0, z0);
glColor3f(r1, g1, b1);
glNormal3f(u1, v1, w1);
glTexCoord2f(s1, t1);
glVertex3f(x1, y1, z1);
...
glEnd();
```

Note that we can use vertex arrays to increase efficiency.

Interpolation

OpenGL uses bilinear interpolation to find proper texels from specified texture coordinates. Can be distortions:
- good selection of tex coordinates
- poor selection of tex coordinates
- texture stretched over trapezoid showing effects of bilinear interpolation.

Texture Parameters

- OpenGL variety of parameter that determine how texture is applied
  - Wrapping parameters determine what happens if s and t are outside the (0,1) range
  - Filter modes allow us to use area averaging instead of point samples
  - Mipmapping allows us to use textures at multiple resolutions
  - Environment parameters determine how texture mapping interacts with shading.

Wrapping Mode

Clamping: if \( s,t > 1 \) use 1, if \( s,t < 0 \) use 0
Wrapping: use \( s,t \) modulo 1

```c
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

Filter Modes

Modes determined by

```c
- glTexParameteri(target, type, mode );
```

```c
- glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);,
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIP_FILTER, GL_LINEAR);
```

Note that linear filtering requires a border of an extra texel for filtering at edges (border = 1).

Magnification and Minification

More than one texel can cover a pixel (minification) or more than one pixel can cover a texel (magnification)

Can use point sampling (nearest texel) or linear filtering (2 x 2 filter) to obtain texture values.
Mipmapped Textures

- Mipmapping allows for prefiltered texture maps of decreasing resolutions
- Lessens interpolation errors for smaller textured objects
- Declare mipmap level during texture definition
  ```
glTexImage2D( GL_TEXTURE_*D, level, … )
  ```
- GLU mipmap builder routines will build all the textures from a given image
  ```
gluBuild*DMipmaps( … )
  ```

Example

<table>
<thead>
<tr>
<th>point sampling</th>
<th>linear filtering</th>
</tr>
</thead>
<tbody>
<tr>
<td>mipmapped point sampling</td>
<td>mipmapped linear filtering</td>
</tr>
</tbody>
</table>

Texture Functions

- Controls how texture is applied
  ```
  glTexEnv{fi}[v]( GL_TEXTURE_ENV, prop, param )
  ```
- GL_TEXTURE_ENV_MODE modes
  - GL_MODULATE: modulates with computed shade
  - GL_BLEND: blends with an environmental color
  - GL_REPLACE: use only texture color
  ```
  gl(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE); 
  ```
- Set blend color with GL_TEXTURE_ENV_COLOR

Perspective Correction Hint

- Texture coordinate and color interpolation
  - either linearly in screen space
  - or using depth/perspective values (slower)
- Noticeable for polygons "on edge"
  ```
glHint( GL_PERSPECTIVE_CORRECTION_HINT, hint )
  ```
  where hint is one of
  - GL_DONT_CARE
  - GL_NICEST
  - GL_FASTEST

Generating Texture Coordinates

- OpenGL can generate texture coordinates automatically
  ```
glTexGen{ifd}[v]();
  ```
- specify a plane
  - generate texture coordinates based upon distance from the plane
- generation modes
  - GL_OBJECT_LINEAR
  - GL_EYE_LINEAR
  - GL_SPHERE_MAP (used for environmental maps)

Texture Objects

- Texture is part of the OpenGL state
  - If we have different textures for different objects, OpenGL will be moving large amounts data from processor memory to texture memory
- Recent versions of OpenGL have texture objects
  - one image per texture object
  - Texture memory can hold multiple texture objects
Applying Textures II

1. specify textures in texture objects
2. set texture filter
3. set texture function
4. set texture wrap mode
5. set optional perspective correction hint
6. bind texture object
7. enable texturing
8. supply texture coordinates for vertex
   - coordinates can also be generated

Other Texture Features

- Environmental Maps
  - Start with image of environment through a wide angle lens
  - Can be either a real scanned image or an image created in OpenGL
  - Use this texture to generate a spherical map
  - Use automatic texture coordinate generation
- Multitexturing
  - Apply a sequence of textures through cascaded texture units