Objectives

- Learn the basic design of a graphics system
- Introduce graphics pipeline architecture
- Describe software components of an interactive graphics system

Image Formation Revisited

- Can we mimic the synthetic camera model to design graphics hardware and software?
- Application Programmer Interface (API)
  - Need only specify
    - Objects
    - Materials
    - Viewer
    - Lights
  - But how is the API implemented?

Physical Approaches

- Ray tracing: follow rays of light from center of projection until they either are absorbed by objects or go off to infinity
  - Can handle global effects
    - Multiple reflections
    - Translucent objects
  - Slow
  - Need whole data base
- Radiosity: Energy based approach
  - Very slow

Practical Approach

- Process objects one at a time in the order they are generated by the application
  - Can consider only local lighting
- Pipeline architecture
- All steps can be implemented in hardware on the graphics card

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
    - Quadrics
    - 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 determine which objects are in front (hidden surface removal)

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 ISO and later ANSI standard (1980s)
• GKS not easily extended to 3D (GKS-3D)
• Far behind hardware development

Programming with OpenGL Part 1: Background

David Breen
Department of Computer Science
Drexel University
PHIGS and X

- Programmers Hierarchical Graphics System (PHIGS)
  - Arise 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

- GL’s success led to OpenGL (1992), a platform-independent API that was
  - Easy to use
  - Close enough to the hardware to get excellent performance
  - Focus 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, ATI, Apple, Intel, ……
  - Relatively stable (present version 2.1)
    - Evolution reflects new hardware capabilities
      - 3D texture mapping and texture objects
      - Vertex programs
      - Programmable shaders
    - Allows for platform specific features through extensions

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 Library (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
GLX, AGL, or WGL
GL
GLUT
X, Win32, Mac O/S
software and/or hardware

OpenGL Architecture

Immediate Mode
Geometric pipeline

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`......
- Underlying storage mode is the same
- Easy to create overloaded functions in C++ but issue is efficiency

OpenGL function format

```
glVertex3f(x, y, z)
```
function name
belongs to GL library
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, ...

A Simple Program

Generate a square on a solid background

```
#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

• Unix/Linux
  – Include files usually in .../include/GL
  – Compile with -lglut -lglu -lgl loader flags
  – May have to add -L flag for X libraries
  – Mesa implementation included with most linux distributions
Notes on compilation

• On tux
  – #include <GL/glut.h>
  – g++ *.cpp -L/usr/X11R6/lib -lGL -lglut -lGLU

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 -lglu32 -lglut32 flags

Programming with OpenGL
Part 2: Complete Programs

David Breen
Department of Computer Science
Drexel University

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
```c
#include <GL/glut.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();
    glutMainLoop();
}
```

GLUT functions

- `glutInit` allows application to get command line arguments and initializes system
- `glutDisplayMode` 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

```c
void init() {
    glClearColor(0.0, 0.0, 0.0, 1.0);
    glColor3f(1.0, 1.0, 1.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
}
```

Coordinate Systems

- The units used 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 sides of length 2

Orthographic Viewing

In the default orthographic view, points are projected 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 using `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.

```c
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.

```c
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();
}
```

OpenGL Primitives

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

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

Text

- Not an OpenGL primitive
- Fonts provided by windowing system are not portable
- GLUT has some *bitmap* and *stroke* fonts

```c
- glutBitmapCharacter(void *font, int char)
- glutBitmapCharacter(void *font, int char)
```
GLUT Bitmap Text

- `glutBitmapCharacter(void *font, int char)`
  - Displays `char` at current raster position
- `glRasterPos3f(TYPE x, TYPE y, TYPE z)`
  - Sets raster position by transforming world coordinate point into screen coordinates
  - Current raster position is automatically incremented after `char` is displayed
- Example (10 pt times roman)
  - `glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_10, 'a')`

GLUT Stroke Text

- `glStrokeCharacter(void *font, int char)`
  - Displays `char` at current world location mapped to screen
  - Stroke text is geometry that is modified by modeling transformations
  - Each character is approximately 100 x 100 in world coordinates
  - Example (roman)
    - `glutStrokeCharacter(GLUT_STROKE_ROMAN, 'a')`

Attributes

- Attributes are part of the OpenGL state 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
  - `glColor
  - glVertex
  - glColor`
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 to use the entire window for the image: `glViewport(x, y, w, h)`
- Values in pixels (screen coordinates)