Introduction to Computer Graphics

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

• Intense introduction to computer graphics.
• Intended for advanced undergraduate and graduate students.
• Topics include:
  - OpenGL API, GLSL shading language
  - Geometric transformations
  - 3D viewing
  - Geometric modeling, curves and surfaces
  - Shading, texture mapping, compositing
## Syllabus

<table>
<thead>
<tr>
<th>Week</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction, history, vector/raster graphics</td>
</tr>
<tr>
<td>2-4</td>
<td>OpenGL, GLSL, Qt</td>
</tr>
<tr>
<td>5-6</td>
<td>Geometry, 2D/3D transformations</td>
</tr>
<tr>
<td>7</td>
<td>Texture mapping</td>
</tr>
<tr>
<td>8</td>
<td>Projections, perspective</td>
</tr>
<tr>
<td>9</td>
<td>3D viewing</td>
</tr>
<tr>
<td>10</td>
<td>Midterm</td>
</tr>
<tr>
<td>11-12</td>
<td>Shading</td>
</tr>
<tr>
<td>13-14</td>
<td>Curves and surfaces</td>
</tr>
</tbody>
</table>
Required Text

Supplementary Texts

Supplementary Texts:


  - The definitive OpenGL programming reference
Grading

• The final grade is computed as follows:
  - Midterm exam: 25%
  - Final exam: 25%
  - Homework programming assignments: 50%

• Substantial programming assignments are due every three weeks.

• Proficiency in C/C++ is expected.

• Prereqs: CSc 221
Contact Information

• Prof. Wolberg
  - Office hours:
    • Wednesdays 3:30-4:30pm, NAC 8/202N
  - Email: wolberg@cs.ccny.cuny.edu

• Teaching Assistant (TA): Siavash Zokai
  - Email: ccny.cs472@gmail.com

• See class web page for all class info such as homework and sample source code:
  www-cs.ccny.cuny.edu/~wolberg/cs472
Objectives

• Broad introduction to Computer Graphics
  - Software
  - Hardware
  - Applications
• Top-down approach
• Shader-Based
• Programs in C/C++ will be assigned to reinforce understanding of the material
Prerequisites

• Good programming skills in C (or C++)
• Basic data structures
• Geometry
• Simple linear algebra
OpenGL Resources

• Can run OpenGL on any system
  - Desktop OpenGL on Windows, Mac, Linux
  - OpenGL ES on mobile platforms: iOS, Android

• Get Qt from www.qt.io/download-open-source
  - Graphical user interface toolkit for all platforms
  - Adds sliders, pushbuttons, advanced widgets to GUI

• www.opengl.org
  - Standards documents and sample code

• www.opengl-tutorial.org
  - Informative tutorials on basic and intermediate topics

• www.khronos.org
Outline: Part 1

• Part 1: Introduction
  - What is Computer Graphics?
  - Applications Areas
  - History
  - Image formation
  - Basic Architecture
Outline: Part 2

• Part 2: Modern OpenGL (shader-based)
  - Architecture
  - Qt for advanced GUIs
  - Simple programs in two and three dimensions
  - Basic shaders and GLSL
  - Interaction
Outline: Part 3

• Part 3: Texture Mapping
  - Buffers
  - Shader Applications
  - Compositing and Transparency
Outline: Part 4

• Part 4: Three-Dimensional Graphics
  - Geometry
  - Transformations
  - Homogeneous Coordinates
  - Viewing
  - Lighting and Shading
Outline: Part 5

• Part 5: Curves and Surfaces
  - Bezier Curves
  - Hermite Curves
  - B-Splines
  - Cubic Splines
  - Coons Patches
What is Computer Graphics?

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

• In this lecture, we explore what computer graphics is about and survey some application areas
• But we start with a historical introduction
Computer Graphics

• *Computer graphics* deals with all aspects of creating images with a computer
  - Hardware
  - Software
  - Applications
Related Fields

Image Processing

Image

Scene Description

Computer Graphics

Computer Vision
Example

• Where did this image come from?

• What hardware/software did we need to produce it?
Preliminary Answer

• **Application**: The object is an artist’s rendition of the sun for an animation to be shown in a domed environment (planetarium)

• **Software**: Maya for modeling and rendering but Maya is built on top of OpenGL

• **Hardware**: PC with discrete graphics (GPU) for modeling and rendering
Basic Graphics System

Input devices

Processor (CPU)

Graphics processor

Frame buffer

Output device

CPU Memory

GPU Memory

Image formed in frame buffer
CRT

Can be used as a line-drawing device (vector graphics) or to display contents of frame buffer (raster graphics)

• Computer graphics goes back to the earliest days of computing
  - Strip charts
  - Pen plotters
  - Simple displays using A/D converters to go from computer to calligraphic CRT

• Cost of refresh for CRT too high
  - Computers slow, expensive, unreliable

- **Wireframe** graphics
  - Draw only lines
- **Sketchpad**
- **Display Processors**
- **Storage tube**

wireframe representation of sun object
Project Sketchpad

- Ivan Sutherland’s PhD thesis at MIT
  - Recognized the potential of man-machine interaction
  - Loop
    - Display something
    - User moves light pen
    - Computer generates new display
  - Sutherland also created many of the now common algorithms for computer graphics
Display Processor

- Rather than have host computer try to refresh display use a special purpose computer called a *display processor* (DPU)

- Graphics stored in display list (display file) on display processor
- Host *compiles* display list and sends to DPU
Direct View Storage Tube

• Created by Tektronix
  - Did not require constant refresh
  - Standard interface to computers
    • Allowed for standard software
    • Plot3D in Fortran
  - Relatively inexpensive
    • Opened door to use of computer graphics for CAD community

• Raster Graphics
• Beginning of graphics standards
  - IFIPS
    • GKS: European effort
      – Becomes ISO 2D standard
    • Core: North American effort
      – 3D but fails to become ISO standard

• Workstations and PCs
Raster Graphics

• Image produced as an array (the *raster*) of picture elements (*pixels*) in the *frame buffer*
Raster Graphics

• Allow us to go from lines and wireframes to filled polygons
PCs and Workstations

• Although we no longer make the distinction between workstations and PCs, historically they evolved from different roots
  - Early workstations characterized by
    • Networked connection: client-server model
    • High-level of interactivity
  - Early PCs included frame buffer as part of user memory
    • Easy to change contents and create images

Realism comes to computer graphics

smooth shading  environment mapping  bump mapping

• Special purpose hardware
  - Silicon Graphics geometry engine
    • VLSI implementation of graphics pipeline

• Industry-based standards
  - PHIGS
  - RenderMan

• Networked graphics: X Window System

• Human-Computer Interface (HCI)

• OpenGL API
• Completely computer-generated feature-length movies (Toy Story) are successful
• New hardware capabilities
  - Texture mapping
  - Blending
  - Accumulation, stencil buffers

• Photorealism

• Graphics cards for PCs dominate market
  - Nvidia, ATI

• Game boxes and game players determine direction of market

• Computer graphics routine in movie industry: Maya, Lightwave

• Programmable pipelines
Computer Graphics: 2010-

• Xbox, Playstation
  - Realistic rendering, animation
• Kinect sensor
  - Gesture recognition
• Touchscreen interfaces
  - Phones, tablets, Windows 10
• 3D scanning and printing
  - Editing tools for rapid prototyping 3D models
Image Formation

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

• Fundamental imaging notions
• Physical basis for image formation
  - Light
  - Color
  - Perception
• Synthetic camera model
• Other models
Image Formation

• In computer graphics, we form images which are generally two dimensional using a process analogous to how images are formed by physical imaging systems
  - Cameras
  - Microscopes
  - Telescopes
  - Human visual system
Elements of Image Formation

- Objects
- Viewer
- Light source(s)

- Attributes that govern how light interacts with the materials in the scene
- Note the independence of the objects, viewer, and light source(s)
Light

- *Light* is the part of the electromagnetic spectrum that causes a reaction in our visual systems.
- Generally these are wavelengths in the range of about 350-780 nm (nanometers).
- Long wavelengths appear as reds and short wavelengths as blues.
Ray Tracing and Geometric Optics

One way to form an image is to follow rays of light from a point source determine which rays enter the lens of the camera. However, each ray of light may have multiple interactions with objects before being absorbed or going to infinity.
Luminance and Color Images

• Luminance
  - Monochromatic
  - Values are gray levels
  - Analogous to working with black and white film or television

• Color
  - Has perceptual attributes of hue, saturation, and lightness
  - Use three primary colors: red, green, blue (RGB)
Shadow Mask CRT
Additive and Subtractive Color

• Additive color
  - Form a color by adding amounts of three primaries
    • CRTs, projection systems, positive film
  - Primaries are Red (R), Green (G), Blue (B)

• Subtractive color
  - Form a color by filtering white light with cyan (C), Magenta (M), and Yellow (Y) filters
    • Light-material interactions
    • Printing
    • Negative film
Pinhole Camera

Use trigonometry to find projection of a point

\[ x_p = -d(x/z) \quad y_p = -d(y/z) \quad z_p = -d \]

These are equations of simple perspective
Synthetic Camera Model

- projector
- image plane
- projection of $p$
- center of projection
Advantages

- Separation of objects, viewer, light sources
- Two-dimensional graphics is a special case of three-dimensional graphics
- Leads to simple software API
  - Specify objects, lights, camera, attributes
  - Let implementation determine image
- Leads to fast hardware implementation
Global vs Local Lighting

• Cannot compute color or shade of each object independently
  - Some objects are blocked from light
  - Light can reflect from object to object
  - Some objects might be translucent
Physical Approach

• 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
  - Must have whole data base available at all times
Why not ray tracing?

• Ray tracing is more physically based so why don’t we use it to design a graphics system?
• Possible and is actually simple for simple objects such as polygons and quadrics with simple point sources
• In principle, can produce global lighting effects such as shadows and multiple reflections but ray tracing is slow and not well-suited for interactive applications
• Ray tracing with GPUs is now close to real time
Graphics Pipeline

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

• Learn the basic design of a graphics system
• Introduce pipeline architecture
• Examine software components for an interactive graphics system
Practical Approach

- Process 3D objects one at a time in the order they are generated by the application
  - Can consider only local lighting
- Pipeline architecture

  
  ![Pipeline Diagram]

  - All steps can be implemented in hardware on the graphics card
Following the Graphics Pipeline: Vertex Processing

• Much of the work in the pipeline is in converting object representations from one coordinate system to another
  - Object coordinates
  - Camera (eye) coordinates
  - Screen coordinates

• Every change of coordinates is equivalent to a matrix transformation

• Vertex processor also computes vertex colors
Projection

- *Projection* is 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
Perspective Projection

Projectors converge at center of projection (COP)
Perspective Projection
Parallel Projection
Primitive Assembly

• The fundamental unit of rendering in OpenGL is known as the *primitive*.
• The three basic primitive types are points, lines, and triangles.
• Vertices must be collected into primitives before clipping and rasterization can take place.
Clipping

Just as a real camera cannot “see” the whole world, the virtual camera can only see part of the world or object space.

- Objects that are not within this volume are said to be clipped out of the scene.
Specification of Virtual Camera

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

• If an object is not clipped out, the appropriate pixels in the frame buffer must be assigned colors
• Rasterizer produces a set of fragments for each object
• Fragments are “potential pixels”
  - Have a location in frame buffer
  - Color and depth attributes
• Vertex attributes are interpolated over objects by the rasterizer
Putting It All Together

• Vertices stream into vertex processor and are transformed into new vertices
• These vertices are collected to form primitives
• Primitives are rasterized to form fragments
• Fragments are colored by fragment processor
Fragment Processing

• Fragments are processed to determine the color of the corresponding pixel in the frame buffer
• Colors can be determined by texture mapping or interpolation of vertex colors
• Fragments may be blocked by other fragments closer to the camera
  - Hidden-surface removal
API Background

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

• Graphics API history
• OpenGL API
• OpenGL function format
• Immediate Mode vs Retained Mode
• Examples
The Programmer’s Interface

• Programmer sees the graphics system through a software interface: the Application Programmer Interface (API)
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
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 access 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 (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 API

• We shall use the OpenGL API

• **Benefits:** widespread, portable across all platforms

• **Includes:** functions needed to form an image
  - Objects
  - Viewer
  - Light Source(s)
  - Materials

• **Goal:** provides an abstraction layer between the application and the underlying graphics subsystem (GPU).
Example (old style)

```c
glBegin(GL_POLYGON)
    glVertex2f(0.0, 0.0);
    glVertex2f(0.0, 1.0);
    glVertex2f(1.0, 0.0);
    glVertex2f(1.0, 1.0);
    glVertex2f(0.0, 1.0);
    glVertex2f(0.0, 0.0);
 glEnd();
```

type of object

location of vertex

end of object definition
A Simple Program

Generate a square on a solid background
```c
#include <GL/glut.h>
void mydisplay(){
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_QUAD);
        glVertex2f(-0.5,-0.5);
        glVertex2f(-0.5, 0.5);
        glVertex2f( 0.5, 0.5);
        glVertex2f( 0.5,-0.5);
    glEnd();
}
int main(int argc, char** argv){
    glutCreateWindow("simple");
    glutDisplayFunc(mydisplay);
    glutMainLoop();
}
```
OpenGL Functions

- Primitives
  - Points
  - Line Segments
  - Triangles
- Attributes
- Transformations
  - Viewing
  - Modeling
- Control (GLUT)
- Input (GLUT)
- Query
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
    - In modern OpenGL (3.1+) most state variables are defined by the application and sent to the shaders
Lack of Object Orientation

- OpenGL is not object oriented so there are multiple functions for a given logical function
  - `glUniform3f`
  - `glUniform2i`
  - `glUniform3dv`
OpenGL Function Format

- **function name**: `glUniform3f(x,y,z)`
- **belongs to GL library**
- **dimensions**: `x,y,z` are floats
- **glUniform3fv(p)**
  - `p` is a pointer to an array (vector)
Modern OpenGL (3.1+)

- Performance achieved by using GPU rather than CPU
- Control GPU through programs called shaders
- Application’s job is to send data to GPU
- GPU does all rendering
Immediate Mode Graphics

• Geometry specified by vertices
  - Locations in space (2 or 3 dimensional)
  - Points, lines, circles, polygons, curves, surfaces

• Immediate mode
  - Each time a vertex is specified in application, its location is sent to the GPU
  - Old style used `glVertex`
  - Creates bottleneck between CPU and GPU
  - Removed from OpenGL 3.1 and OpenGL ES 2.0
Retained Mode Graphics

• Put all vertex attribute data in array
• Send array to GPU to be rendered immediately
• Almost OK but problem is we would have to send array over each time we need another render of it
• Better to send array over and store on GPU for multiple renderings
OpenGL and GLSL

• Shader-based OpenGL is based less on a state machine model than a data flow model
• Most state variables, attributes and related pre 3.1 OpenGL functions have been deprecated
• Action happens in shaders
• Job in application is to get data to GPU
Execution Example (WebGL)
OpenGL Libraries

• OpenGL core library
  - OpenGL32 on Windows
  - GL on most unix/linux systems (libGL.a)

• OpenGL Utility Library (GLU)
  - Provides functionality in OpenGL core but avoids having to rewrite code
  - Will only work with legacy code (pre 3.1)

• 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
    • No sliders, spinboxes, combo boxes, radio buttons, …
    • We will use Qt instead
Evolution of OpenGL

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

- Evolution of the OpenGL pipeline
- A prototype application in OpenGL
- OpenGL shading language (GLSL)
- Vertex shaders
- Fragment shaders
- Examples
What Is OpenGL?

- OpenGL is a computer graphics rendering application programming interface, (API)
  - With it, you can generate high-quality color images by rendering with geometric and image primitives
  - It forms the basis of many interactive applications that include 3D graphics
  - By using OpenGL, the graphics part of your application can be
    - operating system independent
    - window system independent
Course Ground Rules

• We’ll concentrate on latest versions of OpenGL
• Enforces a new way to program with OpenGL
  - Allows more efficient use of GPU resources
• Modern OpenGL (3.1+) doesn’t support many of the “classic” ways of doing things, such as
  - Fixed-function graphics operations, like vertex lighting and transformations
• All applications must use shaders for their graphics processing
## OpenGL Versions

<table>
<thead>
<tr>
<th>Version</th>
<th>Publication Date</th>
<th>Version</th>
<th>Publication Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenGL 1.0</td>
<td>January 1992</td>
<td>OpenGL 3.0</td>
<td>August 2008</td>
</tr>
<tr>
<td>OpenGL 1.1</td>
<td>January 1997</td>
<td>OpenGL 3.1</td>
<td>March 2009</td>
</tr>
<tr>
<td>OpenGL 1.2</td>
<td>March 1998</td>
<td>OpenGL 3.3</td>
<td>March 2010</td>
</tr>
<tr>
<td>OpenGL 1.2.1</td>
<td>October 1998</td>
<td>OpenGL 4.0</td>
<td>March 2010</td>
</tr>
<tr>
<td>OpenGL 1.3</td>
<td>August 2001</td>
<td>OpenGL 4.1</td>
<td>July 2010</td>
</tr>
<tr>
<td>OpenGL 1.4</td>
<td>July 2002</td>
<td>OpenGL 4.2</td>
<td>August 2011</td>
</tr>
<tr>
<td>OpenGL 1.5</td>
<td>July 2003</td>
<td>OpenGL 4.3</td>
<td>August 2012</td>
</tr>
<tr>
<td>OpenGL 2.0</td>
<td>September 2004</td>
<td>OpenGL 4.4</td>
<td>July 2013</td>
</tr>
<tr>
<td>OpenGL 2.1</td>
<td>July 2006</td>
<td>OpenGL 4.5</td>
<td>August 2014</td>
</tr>
</tbody>
</table>
Evolution of the OpenGL Pipeline

• OpenGL 1.0 was released on July 1\textsuperscript{st}, 1994
• Its pipeline was entirely \textit{fixed-function}
  - the only operations available were fixed by the implementation

• The pipeline evolved
  - but remained based on fixed-function operation through OpenGL versions 1.1 through 2.0 (Sept. 2004)
Example (old style)

```c
glBegin(GL_POLYGON)
    glVertex2f(0.0, 0.0);
    glVertex2f(0.0, 1.0);
    glVertex2f(1.0, 0.0);
    glVertex2f(1.0, 1.0);
    glVertex2f(0.0, 1.0);
    glVertex2f(0.0, 0.0);
 glEnd();
```

**type of object**

**location of vertex**

**end of object definition**
A Simple Program:
It Used to be Easy

```c
#include <GL/glut.h>
void mydisplay(){
    glClearColor(GL_COLOR_BUFFER_BIT);
    glBegin(GL_QUAD);
        glVertex2f(-0.5,-0.5);
        glVertex2f(-0.5,0.5);
        glVertex2f(0.5, 0.5);
        glVertex2f(0.5,-0.5);
    glEnd();
}
int main(int argc, char** argv){
    glutCreateWindow("simple");
    glutDisplayFunc(mydisplay);
    glutMainLoop();
}
```
Event Loop

• Note that the program specifies 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
Beginnings of The Programmable Pipeline

• OpenGL 2.0 (officially) added programmable shaders
  - *vertex shading* augmented the fixed-function transform and lighting stage
  - *fragment shading* augmented the fragment coloring stage

• However, the fixed-function pipeline was still available
Example (GPU based)

• Put geometric data in an array

```cpp
std::vector<vec2> points;
points.push_back(vec2(0.0, 0.0));
points.push_back(vec2(0.0, 1.0));
points.push_back(vec2(1.0, 0.0));
numPoints = 3;
```

• Send array to GPU
  - See next slide

• Tell GPU to render geometry as triangles

```cpp
glClear(GL_COLOR_BUFFER_BIT);
glDrawArrays(GL_TRIANGLES, 0, 3);
```
enum {
    ATTRIB_VERTEX, 
    ATTRIB_COLOR, 
    ATTRIB_TEXTURE_POSITION
};

// create vertex buffer and get handle to it
GLuint vertexBuffer;
glGenBuffers(1, &vertexBuffer);

// bind buffer to the GPU; all future drawing calls gets data from this buffer
glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);

// copy the vertices from CPU to GPU
glBufferData(GL_ARRAY_BUFFER, numPoints*sizeof(vec2), &points[0], GL_STATIC_DRAW);

// enable the assignment of attribute vertex variable
glEnableVertexAttribArray(ATTRIB_VERTEX);

// assign the buffer object to the attribute vertex variable
glVertexAttribPointer(ATTRIB_VERTEX, 2, GL_FLOAT, false, 0, NULL);
An Evolutionary Change

• OpenGL 3.0 introduced the *deprecation model* - the method used to remove features from OpenGL
• Pipeline remained the same until OpenGL 3.1 (released March 24\textsuperscript{th}, 2009)
• Introduced change in how OpenGL contexts are used

<table>
<thead>
<tr>
<th>Context Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full</td>
<td>Includes all features (including those marked deprecated) available in the current version of OpenGL</td>
</tr>
<tr>
<td>Forward Compatible</td>
<td>Includes all non-deprecated features (i.e., creates a context that would be similar to the next version of OpenGL)</td>
</tr>
</tbody>
</table>
The Exclusively Programmable Pipeline

- OpenGL 3.1 removed the fixed-function pipeline
  - programs were required to use only shaders
  - no default shaders
  - app must provide both a vertex and a fragment shader

- Additionally, almost all data is GPU-resident
  - all vertex data sent using buffer objects
OpenGL 3.2 (released August 3rd, 2009) added an additional shading stage: geometry shaders - modify geometric primitives within the graphics pipeline.
More Evolution: Context Profiles

• OpenGL 3.2 also introduced *context profiles*
  - profiles control which features are exposed
  - currently two types of profiles: *core* and *compatible*
  - *Core* profile deprecates legacy features (trim version)
  - *Compatibility* profile maintains backwards compatibility with all versions back to version 1.0.

<table>
<thead>
<tr>
<th>Context Type</th>
<th>Profile</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full</td>
<td>core</td>
<td>All features of the current release</td>
</tr>
<tr>
<td></td>
<td>compatible</td>
<td>All features ever in OpenGL</td>
</tr>
<tr>
<td>Forward Compatible</td>
<td>Core</td>
<td>All non-deprecated features</td>
</tr>
<tr>
<td></td>
<td>Compatible</td>
<td>Not supported</td>
</tr>
</tbody>
</table>
The Latest Pipelines

- OpenGL 4.1 (released July 25\textsuperscript{th}, 2010) included additional shading stages: \textit{tessellation-control} and \textit{tessellation-evaluation} shaders.
- Latest version is 4.5 (August 2014).
Other Shader-Based Versions

• OpenGL ES
  - Designed for embedded and hand-held devices such as cell phones and tablets
  - Version 1.0 (2003) based on OpenGL 2.1
  - Version 2.0 (2007) based on OpenGL 3.1
  - Version 3.0 (2012) based on OpenGL 4.3
  - Version 3.2 (August 2015)

• WebGL
  - Javascript implementation of ES 2.0
  - Runs on most recent browsers
Programming with Qt

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

• Introduce Qt for our GUI
• Write our first “Hello World” program
• Leverage OpenGL for drawing
• Leverage Qt for user interface
• Prototype of homework submissions
Application Framework Requirements

• OpenGL applications need a place to render into
  - usually an on-screen window
• Need to communicate with native windowing system
• Each windowing system interface is different
• We use Qt
  - Advanced C++ GUI library that works everywhere
  - handles widgets and all windowing operations:
    • opening windows
    • input processing
    • widgets such as menus, sliders, spinboxes, combo boxes
Qt

- Qt is ‘cute’

- Cross-platform GUI library (C++)
  - Write once, compile on different platforms for cross-platform use
  - Works on Windows, Mac, Linux, iOS, Android

- A complete interface environment with support for file system browser, OpenGL, WebKit API, media streamer, etc.

- Download from: http://www.qt.io/download-open-source
Homework Interface

• Homework submissions will use Qt
  - Renderings based on OpenGL
  - Parameter selection using advanced GUI
• The screen will be divided into two parts:
  - Tabbed widget containing OpenGL canvas
  - Control panel containing widgets to select parameters
• Each homework problem will be installed in a tabbed widget
  - Pressing a tab brings up associated homework solution and control panel
Interface (1)
Interface (2)
Interface (3)
Files

- main.cpp
  - Create and display UI window
  - Enter infinite loop to process events
- MainWindow.h
  - Header file for MainWindow class
- MainWindow.cpp
  - MainWindow class sets up GUI
- HW.h
  - Header file for HW class
- HW.cpp
  - Base class for homework solutions
  - Virtual functions: initializeGL(), resizeGL(), and paintGL()
```cpp
#include "MainWindow.h"

int main(int argc, char **argv)
{
    QApplication app(argc, argv);       // create application
    MainWindow window;                  // create UI window
    window.showMaximized();             // display window
    return app.exec();                  // infinite processing loop
}
```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

// standard include files

#include <QtWidgets>
#include <iostream>
#include <fstream>
#include <cstdio>
#include <cmath>
#include <cstring>
#include <cstdlib>
#include <cstdarg>
#include <cassert>
#include <vector>
#include <map>
#include <algorithm>
#include "HW.h"

typedef std::map<QString, HW*> hw_type;
class MainWindow : public QWidget {
    Q_OBJECT

public:
    MainWindow (QWidget *parent = 0);

public slots:
    void createWidgets  ();
    void changeTab     (int);
    void reset         ();
    void quit          ();

protected:
    QGroupBox*      createGroupView  ();
    QGroupBox*      createGroupInput  ();
    QHBoxLayout*   createExitButtons ();

private:
    // homework objects
    QStringList            m_hwName;
    hw_type                m_hw;

    // widgets
    QTabWidget         *m_tabWidget;
    QStackedWidget  *m_stackWidget;
};

extern MainWindow *MainWindowP;
#endif // MAINWINDOW_H
MainWindow::MainWindow(QWidget *parent)
    : QWidget(parent)
{
    setWindowTitle("Computer Graphics Homework");

    // set global variable
    MainWindowP = this

    // create a stacked widget to hold multiple control panels
    m_stackWidget = new QStackedWidget;

    // INSERT YOUR OBJECTS IN THIS FUNCTION
    createWidgets();

    // add stacked widget to vertical box layout
    QVBoxLayout *vbox = new QVBoxLayout;
    vbox->addWidget(m_stackWidget);
    vbox->addStretch(1);
    vbox->addLayout(createExitButtons());

    // add all widgets to grid layout
    QHBoxLayout *hbox = new QHBoxLayout;
    hbox->addWidget(createGroupView());
    hbox->setStretch(0, 1);
    hbox->addLayout(vbox);
    setLayout(hbox);

    // set stacked widget to latest solution
    m_tabWidget->setCurrentIndex(m_hwName.size() - 1);
    m_stackWidget->setCurrentIndex(m_hwName.size() - 1);
}
// MainWindow::createWidgets:

// Create user-defined widgets for display and control panel.
// INSERT YOUR LINES HERE TO CREATE A TAB PER HOMEWORK OBJECT.

void MainWindow::createWidgets()
{
    // create list of hw names; m_hwName name will be used for
    // tab name and as key for class in m_hw container
    m_hwName << "1a: P"
        << "1b: Triangle"
        << "2a: P (GLSL)"
        << "2b: Triangle (GLSL)"
        << "3a: Triangle (Texture/Wire)"
        << "3b: Wave";

    // instantiate homework solution classes
    m_hw[m_hwName[0]] = new HW1a;
    m_hw[m_hwName[1]] = new HW1b;
    m_hw[m_hwName[2]] = new HW2a;
    m_hw[m_hwName[3]] = new HW2b;
    m_hw[m_hwName[4]] = new HW3a;
    m_hw[m_hwName[5]] = new HW3b;

    // add control panels to stacked widget
    for(int i = 0; i < (int) m_hwName.size(); i++)
    {
        m_stackWidget->addWidget(m_hw[m_hwName[i]]->controlPanel());
    }
}
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// MainWindow::createGroupView:
//
// Create preview window groupbox.
//
QGroupBox*
MainWindow::createGroupView()
{
    // init group box
    QGroupBox *groupBox = new QGroupBox();
    groupBox->setStyleSheet(GroupBoxStyle);

    // create a tab widget to handle multiple displays
    m_tabWidget = new QTabWidget;

    // add one tab per homework problem
    for(int i = 0; i < (int) m_hwName.size(); i++)
    {
        m_tabWidget->addTab(m_hw[m_hwName[i]], m_hwName[i]);
    }

    // assemble stacked widget in vertical layout
    QVBoxLayout *vbox = new QVBoxLayout;
    vbox->addWidget(m_tabWidget);
    groupBox->setLayout(vbox);

    // init signal/slot connections
    connect(m_tabWidget, SIGNAL(currentChanged(int)), this, SLOT(changeTab(int)));

    return groupBox;
}
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// MainWindow::changeTab:
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void MainWindow::changeTab(int index)
{
    m_tabWidget->setCurrentIndex(index);  // change OpenGL widget
    m_stackWidget->setCurrentIndex(index);  // change control panel
}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// MainWindow::createExitButtons:
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
QHBoxLayout *MainWindow::createExitButtons()
{
    // create pushbuttons
    QPushButton *buttonReset = new QPushButton("Reset");
    QPushButton *buttonQuit    = new QPushButton("Quit");

    // init signal/slot connections
    connect(buttonReset, SIGNAL(clicked()), this, SLOT(reset()));
    connect(buttonQuit,    SIGNAL(clicked()), this, SLOT(quit ()));

    // assemble pushbuttons in horizontal layout
    QHBoxLayout *buttonLayout = new QHBoxLayout;
    buttonLayout->addWidget(buttonReset);
    buttonLayout->addWidget(buttonQuit);

    return buttonLayout;
}
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// MainWindow::reset:
//
// Reset application.
//
void MainWindow::reset()
{
    int index = m_tabWidget->currentIndex(); // current OpenGL widget
    m_hw[m_hwName[index]]->reset();
}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// MainWindow::quit:
//
// Quit application.
//
void MainWindow::quit()
{
    // close the dialog window
    close();
}
#ifndef  HW_H
#define HW_H

#define MXPROGRAMS    32
#define MXUNIFORMS      32

#include <QtWidgets>
#include <QGLWidget>
#include <QGLFunctions>
#include <QGLShaderProgram>
#include <QtOpenGL>

typedef QVector2D vec2;
typedef QVector3D vec3;

enum {
    ATTRIB_VERTEX,
    ATTRIB_COLOR,
    ATTRIB_TEXCOORD,
    ATTRIB_NORMAL
};

typedef std::map<QString, GLuint> UniformMap;
// standard include files

class HW : public QGLWidget, protected QGLFunctions {

public:
    HW(QWidget *parent = 0);
    virtual QGroupBox* controlPanel (); // create control panel
    virtual void reset (); // reset parameters
    void initShader (int, QString, QString, UniformMap);

protected:
    QGLShaderProgram m_program[MXPROGRAMS]; // GLSL programs
    GLint m_uniform [MXPROGRAMS][MXUNIFORMS]; // uniform vars for each program

};

# endif // HW_H
#include "HW.h"

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// HW::HW:
//
// HW constructor.
// This is base class for homework solutions that will replace the control panel, reset function, and add homework solution.
//
HW::HW(QWidget *parent)
 : QGLWidget (parent) {}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// HW::controlPanel:
//
// Create a control panel of widgets for homework solution.
//
QGroupBox* HW::controlPanel() { return NULL; }

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// HW::reset:
//
// Reset parameters in control panel.
//
void HW::reset() {}
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// HW::initShader:
//
// Initialize vertex and fragment shaders.
//
void HW::initShader(int shaderID, QString vshaderName, QString fshaderName, UniformMap uniforms) {

    // compile vertex shader
    if(!m_program[shaderID].addShaderFromSourceFile(QGLShader::Vertex, vshaderName)) {
        QMessageBox::critical(0, "Error", "Vertex shader error: " + vshaderName + "\n" +
                               m_program[shaderID].log(), QMessageBox::Ok);
        exit(-1);
    }

    // compile fragment shader
    if(!m_program[shaderID].addShaderFromSourceFile(QGLShader::Fragment, fshaderName)) {
        QMessageBox::critical(0, "Error", "Fragment shader error: " + fshaderName + "\n" +
                               m_program[shaderID].log(), QMessageBox::Ok);
        exit(-1);
    }

    // bind the attribute variable in the glsl program with a generic vertex attribute index;
    // values provided via ATTRIB_VERTEX will modify the value of "a_position")
    glBindAttribLocation(m_program[shaderID].programId(), ATTRIB_VERTEX, "a_Position");
    glBindAttribLocation(m_program[shaderID].programId(), ATTRIB_COLOR, "a_Color");
    glBindAttribLocation(m_program[shaderID].programId(), ATTRIB_TEXCOORD, "a_TexCoord");
    glBindAttribLocation(m_program[shaderID].programId(), ATTRIB_NORMAL, "a_Normal");
}
// link shader pipeline; attribute bindings go into effect at this point
if(!m_program[shaderID].link()) {
    QMessageBox::critical(0, "Error", "Could not link shader: " + vshaderName + "\n" + m_program[shaderID].log(), QMessageBox::Ok);
    exit(-1);
}

// iterate over all uniform variables; map each uniform name to shader location ID
for(std::map<QString, GLuint>::iterator iter = uniforms.begin(); iter != uniforms.end(); ++iter) {
    QString uniformName = iter->first;
    GLuint uniformID   = iter->second;
    // get storage location
    m_uniform[shaderID][uniformID]=glGetUniformLocation(m_program[shaderID].programId(),
             uniformName.toStdString().c_str());
    if(m_uniform[shaderID][uniformID] < 0) {
        qDebug() << "Failed to get the storage location of " + uniformName;
        exit(-1);
    }
}

}
HW0a:
White square on black background using Legacy OpenGL (pre 3.1)

Aspect ratio is not preserved
#include "HW.h"

// ---------------------------------------------------------------------------
// standard include files
//

class HW0a : public HW {
    Q_OBJECT
public:
    HW0a (QWidget *parent = 0);    // constructor
    QGroupBox*  controlPanel();    // create control panel

protected:
    void initializeGL ();          // init GL state
    void resizeGL    (int, int);   // resize GL widget
    void paintGL     ();           // render GL scene
}
#include "HW0a.h"

// HW0a::HW0a:
// HW0a constructor.
HW0a::HW0a(QWidget *parent)
    : HW (parent) {}

// HW0a::initializeGL:
// Initialization routine before display loop.
// Gets called once before the first time resizeGL() or paintGL() is called.
void HW0a::initializeGL()
{
    // init state variables
    glClearColor(0.0, 0.0, 0.0, 0.0); // set background color
    glColor3f (1.0, 1.0, 1.0); // set foreground color
}
// Resize event handler. The input parameters are the window width (w) and height (h).
//
void HW0a::resizeGL(int w, int h)
{
    // set viewport to occupy full canvas
    glViewport(0, 0, w, h);

    // init viewing coordinates for orthographic projection
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
}

// Update GL scene.
//
void HW0a::paintGL()
{
    // clear canvas with background values
    glClear(GL_COLOR_BUFFER_BIT);

    // define polygon
    glBegin(GL_POLYGON);
        glVertex2f(0.25, 0.25);
        glVertex2f(0.75, 0.25);
        glVertex2f(0.75, 0.75);
        glVertex2f(0.25, 0.75);
    glEnd();
}
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// HW0a::controlPanel:
//
// Create control panel groupbox.
//
QGroupBox*
HW0a::controlPanel()
{
    // init group box
    QGroupBox *groupBox = new QGroupBox("Homework 0a");

    return(groupBox);
}
HW0b:
Rotating trapezoid using Legacy OpenGL (pre 3.1)

Aspect ratio is preserved
#include "HW.h"

// ----------------------------------------------------------------------------------
// standard include files
//

class HW0b : public HW {
    Q_OBJECT

public:
    HW0b (QWidget *parent = 0);    // constructor
    QGroupBox*  controlPanel();    // create control panel
    void                reset           ();    // reset parameters

public slots:
    void                flipY            (int);    // flip y-coordinate
    void                aspect         (int);    // maintain aspect ratio
    void                rotate          (int);    // rotate data

protected:
    void                initializeGL  ();    // init GL state
    void                resizeGL     (int, int);    // resize GL widget
    void                paintGL       ();    // render GL scene

private:
    int                   m_winW;                        // window width
    int                   m_winH;                         // window height
    double            m_angle;                        // rotation angle
    QCheckBox  *m_checkBoxFlip;           // checkbox to flip y-coordinate
    QCheckBox  *m_checkBoxAR;            // checkbox to maintain aspect ratio
    QSlider         *m_slider;                        // rotation slider
    QSpinBox     *m_spinBox;                    // rotation spinbox
};
#include "HW0b.h"

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// HW0b::HW0b:
// // HW0b constructor.
// HW0b::HW0b(QWidget *parent)
//   : HW (parent), m_angle(0.0f)
//{}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// HW0b::initializeGL:
// // Initialization routine before display loop.
// // Gets called once before the first time resizeGL() or paintGL() is called.
// void
HW0b::initializeGL()
{
    // init state variables
    glClearColor(0.0, 0.0, 0.0, 0.0); // set background color
    glColor3f (1.0, 1.0, 1.0);         // set foreground color
}
void
HW0b::resizeGL(int w, int h)
{
    // save window dimensions
    m_winW = w;
    m_winH = h;

    // compute aspect ratio
    float xmax, ymax;
    if(m_checkBoxAR->isChecked()) {
        float ar = (float) w / h;
        if(ar > 1.0) {          // wide screen
            xmax = ar;
            ymax = 1.;
        } else {                // tall screen
            xmax = 1.;
            ymax = 1/ar;
        }
    } else {
        xmax = 1.0;
        ymax = 1.0;
    }

    // set viewport to occupy full canvas
    glViewport(0, 0, w, h);

    // init viewing coordinates for orthographic projection
    glLoadIdentity();
    if(m_checkBoxFlip->isChecked())
        glOrtho(-xmax, xmax, ymax, -ymax, -1.0, 1.0);
    else   glOrtho(-xmax, xmax, -ymax, ymax, -1.0, 1.0);
}
void HW0b::paintGL()
{
    // initial data
    QVector2D v[] = {
        QVector2D(-.25, -.25),
        QVector2D(.25, -.25),
        QVector2D(.15, .25),
        QVector2D(-.15, .25)
    };

    // clear canvas with background values
    glClear(GL_COLOR_BUFFER_BIT);

    // init cosine and sine variables
    double c = cos(m_angle);
    double s = sin(m_angle);

    // define polygon
    glBegin(GL_POLYGON);
        for(int i=0; i<4; i++) {
            glVertex2f(c*v[i].x() - s*v[i].y(), s*v[i].x() + c*v[i].y());
        }
    glEnd();
    glFlush();
}
QGroupBox *
HW0b::controlPanel()
{
   // init group box
   QGroupBox *groupBox = new QGroupBox("Homework 0b");
   groupBox->setMinimumWidth(300);

   // layout for assembling widgets
   QGridLayout *layout = new QGridLayout;

   // create checkboxes
   m_checkBoxFlip = new QCheckBox("Flip y-coordinates");
   m_checkBoxAR   = new QCheckBox("Maintain aspect ratio");
   m_checkBoxFlip->setChecked(false);
   m_checkBoxAR   ->setChecked(true);

   // create slider to rotate data
   m_slider = new QSlider(Qt::Horizontal);
   m_slider->setRange(0, 360);
   m_slider->setValue(0);

   // create spinBox
   m_spinBox = new QSpinBox;
   m_spinBox->setRange(0, 360);
   m_spinBox->setValue(0);

   // slider label to display name
   QLabel *label = new QLabel("Rotation");
// assemble widgets into layout
layout->addWidget(m_checkBoxFlip, 0, 0, 1, 3);
layout->addWidget(m_checkBoxAR  , 1, 0, 1, 3);
layout->addWidget(label,                     2, 0);
layout->addWidget(m_slider,               2, 1);
layout->addWidget(m_spinBox,           2, 2);

// assign layout to group box
groupBox->setLayout(layout);

// init signal/slot connections
connect(m_checkBoxFlip, SIGNAL(stateChanged(int)), this, SLOT(flipY(int)));
connect(m_checkBoxAR  , SIGNAL(stateChanged(int)), this, SLOT(aspect(int)));
connect(m_slider,              SIGNAL(valueChanged(int)), this, SLOT(rotate(int)));
connect(m_spinBox,         SIGNAL(valueChanged(int)), this, SLOT(rotate(int)));

return(groupBox);
// Slot function to flip y-coordinates.
void
HW0b::flipY(int state)
{
    // update checkbox
    m_checkBoxFlip->setChecked(state);

    // call resizeGL() to reset coordinate system
    resizeGL(m_winW, m_winH);

    // redraw
    updateGL();
}

// Slot function to maintain aspect ratio.
void
HW0b::aspect(int state)
{
    // update checkbox
    m_checkBoxAR->setChecked(state);

    // call resizeGL() to reset coordinate system
    resizeGL(m_winW, m_winH);

    // redraw
    updateGL();
}
// Slot function to rotate data.
void HW0b::rotate(int angle) {
  // update slider and spinbox
  m_slider->blockSignals(true);
  m_slider->setValue(angle);
  m_slider->blockSignals(false);

  m_spinBox->blockSignals(true);
  m_spinBox->setValue(angle);
  m_spinBox->blockSignals(false);

  // convert angle to radians
  m_angle = angle * (M_PI / 180.);

  // redraw
  updateGL();
}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Reset parameters.
void HW0b::reset() {
  // reset checkboxes
  m_checkBoxFlip->setChecked(false);
  m_checkBoxAR->setChecked(true);
  resizeGL(m_winW, m_winH);

  // reset angle and slider/spinbox settings
  m_angle = 0;
  m_slider->setValue(m_angle);
  m_spinBox->setValue(m_angle);

  // redraw
  updateGL();
}
Programming with Legacy OpenGL
(Pre OpenGL 3.1)

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

• Build a complete first program
  - Introduce a standard program structure using basic OpenGL (pre 3.1)
  - Why start with pre OpenGL 3.1 (legacy code)?
    • Easier learning curve
    • Familiarity with lots of existing code already written in it.

• Simple viewing
  - Two-dimensional viewing as a special case of three-dimensional viewing

• Initialization steps and program structure
OpenGL Camera

• OpenGL places a camera at the origin in object space 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 forward along the $z$ axis onto the plane $z=0$.
Viewports

• Do not have to use the entire window for the image: `glViewport(x, y, w, h)`
• Values in pixels (window coordinates)
Program Structure

- Most OpenGL programs have a similar structure that consists of the following functions
  - `main()`:
    - Opens main window with control panel and OpenGL canvas
    - Enters event loop (last executable statement)
  - `initializeGL()`: sets the state variables
    - Viewing
    - Attributes
  - `resizeGL()`: handles window resizing event
    - Sets viewport
    - Sets viewing coordinates for orthographic or perspective projection
  - `paintGL()`: render scene
    - Clear framebuffer
    - Call `glVertex*()` to draw primitives (triangles, polygons)
void initializeGL()
{
    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);
}

black clear color
opaque window
fill with white
viewing volume
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
  \begin{verbatim}
  glMatrixMode (GL_PROJECTION);
  \end{verbatim}
• Transformation functions are incremental so we start with an identity matrix and alter it with a projection matrix that gives the view volume
  \begin{verbatim}
  glLoadIdentity ();
  glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
  \end{verbatim}
2D and 3D Viewing

- In `glOrtho(left, right, bottom, top, near, far)` the near and far distances are measured from the camera.
- 2D vertex commands place all vertices in the plane $z=0$.
- In 2D, the view or clipping volume becomes a *clipping window*.
void paintGL()
{
    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();
}
Pre-OpenGL 3.1 Primitives

- **GL_POINTS**
- **GL_LINES**
- **GL_LINES**
- **GL_LINE_STRIP**
- **GL_LINE_LOOP**
- **GL_TRIANGLES**
- **GL_TRIANGLES**
- **GL_TRIANGLES**
- **GL_TRIANGLES**
- **GL_TRIANGLE_STRIP**
- **GL_TRIANGLE_FAN**
- **GL_POLYGON**
- **GL_QUAD_STRIP**
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 $0 \leq \theta \leq 2\pi$. 
Example Using 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();
}
```
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
  glVertex
  ```
First Assignment: Tessellation and Twist

• Consider rotating a 2D point about the origin

\[ x' = x \cos \theta - y \sin \theta \]

\[ y' = x \sin \theta + y \cos \theta \]

• Now let amount of rotation depend on distance from origin giving us **twist**

\[ x' = x \cos(d\theta) - y \sin(d\theta) \]

\[ y' = x \sin(d\theta) + y \cos(d\theta) \]

\[ d \propto \sqrt{x^2 + y^2} \]
Example

triangle

triangle

tessellated triangle

twist without tessellation

twist after tessellation
void initializeGL()
{
    // init vertex and color buffers
    initBuffers();

    // init state variables
    glClearColor (0.0, 0.0, 0.0, 1.0);
    glColor3f(1.0, 1.0, 1.0);
}
void resizeGL(int w, int h)
{
    // compute aspect ratio
    float ar = (float) w / h;

    // set xmax, ymax
    float xmax, ymax;
    if(ar > 1.0) {          // wide screen
        xmax = ar;
        ymax = 1.;
    } else {                // tall screen
        xmax = 1.;
        ymax = 1/ar;
    }

    // set viewport to occupy full canvas
    glViewport(0, 0, w, h);

    // init viewing coordinates for orthographic projection
    glLoadIdentity();
    glOrtho(-xmax, xmax, -ymax, ymax, -1.0, 1.0);
}
typedef QVector2D vec2;
typedef QVector3D vec3;
std::vector<vec2> m_points;
std::vector<vec3> m_colors;

void paintGL()
{
    // clear canvas with background values
    glClear(GL_COLOR_BUFFER_BIT);

    // draw all points in m_points
    for(uint i=0, j=0; i<m_colors.size(); ++i) {
        // set color
        glColor3f(m_colors[i][0], m_colors[i][1], m_colors[i][2]);

        glBegin(GL_TRIANGLES);
        glVertex2f(m_points[j][0], m_points[j][1]); j++;
        glVertex2f(m_points[j][0], m_points[j][1]); j++;
        glVertex2f(m_points[j][0], m_points[j][1]); j++;
        glEnd();
    }
}
void initBuffers()
{
 // init triangle vertices
 const vec2 v[] = {
   vec2( 0.0 , 0.75),
   vec2( 0.65, -0.375),
   vec2(-0.65, -0.375)
 };

 // recursively subdivide triangle;
 // store vertices and colors in m_points[] and m_colors[]
 divideTriangle(v[0], v[1], v[2], m_subdivisions);
}"
void divideTriangle(vec2 a, vec2 b, vec2 c, int count)
{
    if(count > 0) {
        vec2 ab = vec2((a[0]+b[0]) / 2.0, (a[1]+b[1]) / 2.0);
        vec2 ac = vec2((a[0]+c[0]) / 2.0, (a[1]+c[1]) / 2.0);
        vec2 bc = vec2((b[0]+c[0]) / 2.0, (b[1]+c[1]) / 2.0);
        divideTriangle( a, ab, ac, count-1);
        divideTriangle( b, bc, ab, count-1);
        divideTriangle( c, ac, bc, count-1);
        divideTriangle(ab, ac, bc, count-1);
    } else triangle(a, b, c);
}
void triangle(vec2 a, vec2 b, vec2 c) {
    if (m_updateColor) {
        m_colors.push_back(vec3((float) rand()/RAND_MAX,
               (float) rand()/RAND_MAX,
               (float) rand()/RAND_MAX));
    }

    // init geometry
    m_points.push_back(rotTwist(a));
    m_points.push_back(rotTwist(b));
    m_points.push_back(rotTwist(c));
}
void rotTwist(vec2 p) {
    float d = m_twist ? sqrt(p[0][0]*p[0][0] + p[1]*p[1]) : 1;
    float sinTheta = sin(d*m_theta);
    float cosTheta = cos(d*m_theta);
    return vec2(p[0]*cosTheta - p[1]*sinTheta,
                 p[0]*sinTheta + p[1]*cosTheta);
}
Programming with OpenGL: Sierpinski Gasket Example (3D)

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

• Develop a more sophisticated 3D 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
• Going to 3D
  - Not much changes
  - Use `vec3`, `glUniform3f`
  - Have to worry about the order in which primitives are rendered or use hidden-surface removal
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 is a fractal (fractional dimension) object
// initial triangle
vec2 v[3] = {vec2(-1.0, -0.58),
             vec2( 1.0, -0.58),
             vec2( 0.0,  1.15)};

int n; // number of recursive steps
Draw one triangle

// display one triangle
void triangle(vec2 a, vec2 b, vec2 c)
{
    static int i = 0;

    points[i] = a;
    points[i+1] = b;
    points[i+2] = c;
    i += 3;
}

// triangle subdivision using vertex numbers
void divide_triangle(vec2 a, vec2 b, vec2 c, int m)
{
    vec2 ab, ac, bc;

    if(m > 0){
        ab = (a + b)/2;
        ac = (a + c)/2;
        bc = (b + c)/2;
        divide_triangle(a, ab, ac, m-1);
        divide_triangle(c, ac, bc, m-1);
        divide_triangle(b, bc, ab, m-1);
    }
    // else, draw triangle at end of recursion
    else triangle(a,b,c);
}
void paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawArrays(GL_TRIANGLES, 0, NumVertices);
}

void initializeGL()
{
    ...
    // v: initial triangle vertices
    // n: number of recursive steps
    divide_triangle(v[0], v[1], v[2], n);
    ...
}
Moving to 3D

• We can easily make the program three-dimensional by using
  
  vec3 v[3]

and starting 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

• Code almost identical to 2D example
Triangle code

// display one triangle
void triangle(vec3 a, vec3 b, vec3 c) 
{
    static int i = 0;

    points[ i ] = a;
    points[ i+1 ] = b;
    points[ i+2 ] = c;
    i += 3
}

Subdivision code

// triangle subdivision using vertex numbers
void divide_triangle(vec3 a, vec3 b, vec3 c, int m) {
    vec3 ab, ac, bc;

    if(m > 0) {
        ab = (a + b)/2;
        ac = (a + c)/2;
        bc = (b + c)/2;
        divide_triangle(a, ab, ac, m-1);
        divide_triangle(c, ac, bc, m-1);
        divide_triangle(b, bc, ab, m-1);
    }
    // else, draw triangle at end of recursion
    else triangle(a,b,c);
}
void tetrahedron(int m)
{
    glColor3f(1.0,0.0,0.0,0.0);
    divide_triangle(v[0], v[1], v[2], m);

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

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

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

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

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

get this

want this
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
  - Enabled in `initializeGL()`
    - `glEnable(GL_DEPTH_TEST)`
  - Cleared in the `paintGL()`
    - `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)`
Surface vs. Volume Subdivision

• In our example, we divided the surface of each face
• We could also divide the volume using the same midpoints
• The midpoints define four smaller tetrahedrons, one for each vertex
• Keeping only these tetrahedrons removes a volume in the middle
Volume Subdivision
Input and Interaction

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

• Event-driven input
• Callback functions / slot functions
• Window resize functions
  - Alter aspect ratio
  - Preserve aspect ratio
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.

Diagram:
- Trigger process
- Measure process
- Event queue
- Program
- Arrows indicate flow: Trigger → Measure → Event queue → Program
- Arrows labeled: Trigger, Measure, Event, Await
Event Types

- Window: resize, expose, iconify
- Mouse: click one or more buttons
- Motion: move mouse
- Keyboard: press or release a key
Callbacks

• Programming interface for event-driven input
• Define a *callback function* for each type of event the graphics system recognizes
• In Qt, this function is known as the *slot function*
• This user-supplied function is executed when the event occurs
• Qt example:

```cpp
QSlider m_sliderTheta;
QSpinBox m_spinBoxTheta;
connect(m_sliderTheta, SIGNAL(valueChanged(int)), this, SLOT(changeTheta(int)));
connect(m_spinBoxTheta, SIGNAL(valueChanged(int)), this, SLOT(changeTheta(int)));
```
void changeTheta(int angle)
{
    // update slider and spinbox
    m_sliderTheta->blockSignals(true);
    m_sliderTheta->setValue(angle);
    m_sliderTheta->blockSignals(false);
    
    m_spinBoxTheta->blockSignals(true);
    m_spinBoxTheta->setValue(angle);
    m_spinBoxTheta->blockSignals(false);
    
    m_theta = angle * (M_PI/180.);   // convert to radians
    m_points.clear();    // clears points vector
    initBuffers();   // recalculates points
    updateGL();     // redraw: invokes paintGL()
}
Qt Event Loop

- Remember that the last line in main.c for a program using Qt must be return app.exec();

```c
#include "MainWindow.h" // UI window header

int main(int argc, char **argv)
{
    QApplication app(argc, argv); // create application
    MainWindow window; // create UI window
    window.showMaximized(); // display window
    return app.exec(); // infinite processing loop
}
```
Infinite Event Loop

• In each pass through the event loop, Qt
  - looks at the events in the queue
  - for each event in the queue, Qt executes the appropriate slot function if one is defined
  - if no slot is defined for the event, the event is ignored
The paintGL() function is executed whenever Qt 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

Every Qt/OpenGL program must have a paintGL()
Posting Displays

• Many events may invoke `paintGL()`
  - 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
  
  ```cpp
  updateGL(); // if using QGLWidget
  update(); // if using QOpenGLWidget
  ```

  which sets a flag.

• Qt checks to see if the flag is set at the end of the event loop

• If set, then the `paintGL()` function is executed
Animating a Display

• When we redraw the display through the display callback, we usually start by clearing the window
  - `glClearColor(GL_COLOR_BUFFER_BIT)`
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
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
- Handled automatically by QGLWidget() in Qt.
Timer

In `initializeGL()`:

```cpp
QBasicTimer timer; // faster than QTimer
timer.start(12, this); // countdown 12 msec
```

In `MainWidget` class:

```cpp
void MainWidget::timerEvent(QTimerEvent *)
{
    angularSpeed *= 0.99; // decrease angular speed (friction)
    // stop rotation when speed goes below threshold
    if(angularSpeed < 0.01) angularSpeed = 0.0;
    else {
        // update rotation
        rotation = QQuaternion::fromAxisAndAngle(rotationAxis,
                                                   angularSpeed) * rotation;
        update(); // request an update
    }
}
```
Positioning

• Positions in the screen window are 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
  • \( y = h - y; \)

\( (0,0) \)
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 `resizeGL()`
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

original

reshaped
resizeGL()

- The `resizeGL()` function is a good place to put camera functions because it is invoked when the window is first opened

```cpp
void resizeGL(int w, int h)
```
Pre-OpenGL 3.0: Reshape Example #1

• This reshape fct *does not preserve* shapes; it ignores the aspect ratio between the viewport and world window

```c
void resizeGL(int w, int h)
{
    // set viewport to occupy full window
    glViewport(0, 0, w, h);

    // init viewing coordinates for orthographic projection
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-1., 1., -1., 1., -1., 1.);
}
```
Pre-OpenGL 3.0: Reshape Example #2

- This reshape fct *preserves* shapes by making the viewport and world window have the same aspect ratio.

```c
void resizeGL(int w, int h)
{
    // compute aspect ratio
    float ar = (float) w / h;

    // set xmax, ymax
    float xmax, ymax;
    if(ar > 1.0) {  // wide screen
        xmax = ar;
        ymax = 1;
    } else {        // tall screen
        xmax = 1;
        ymax = 1 / ar;
    }

    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-xmax, xmax, -ymax, ymax, 1., 1.);
}
```
Modern OpenGL: Reshape Example (with Qt)

- This reshape fct *preserves* shapes by making the viewport and world window have the same aspect ratio. Uses Qt.

```cpp
Qmatrix4x4 m_projection;
void resizeGL(int w, int h)
{
    // compute aspect ratio
    float ar = (float) w / h;

    // set xmax, ymax
    float xmax, ymax;
    if(ar > 1.0) {
        xmax = ar;
        ymax = 1;
    } else {
        xmax = 1;
        ymax = 1 / ar;
    }

    glViewport(0, 0, w, h);
    m_projection.setToIdentity();
    m_projection.ortho(-xmax, xmax, -ymax, ymax, 1., 1.);
}
```
Shaders and GLSL

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

• Introduce shaders
  • Vertex shaders
  • Fragment shaders
    - Introduce a standard program structure
• Initialization steps and program structure
• Review sample shaders
Graphics Pipeline

- Vertices stream into vertex processor and are transformed into new vertices
- These vertices are collected to form primitives
- Primitives are rasterized to form fragments
- Fragments are colored by fragment processor

```
gl_FragColor = v_Color;
```
Simplified Pipeline Model

Application → GPU Data Flow → Framebuffer

Vertices → Vertex Processing → Rasterizer → Fragments → Fragment Processing → Pixels

- Vertex Shader
- Fragment Shader
Execution Model

Vertex data
Shader Program

Application Program (C++)

Vertex Shader (GLSL)

GPU

Vertex

Primitive Assembly
to Rasterizer

glDrawArrays

Angel/Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012
Execution Model

Rasterizer → Fragment Shader (GLSL) → Frame Buffer

Shader Program

Application Program (C++)

Fragment

Fragment Color
Writing Shaders

• As of OpenGL 3.1, application programs must provide shaders
  - Application programs reside on CPU
  - Shader programs reside on GPU

• OpenGL extensions added for vertex and fragment shaders

• Shaders are written with the OpenGL Shading Language (GLSL)
GLSL: OpenGL Shading Language

• Part of OpenGL 2.0 and up
• High level C-like language
• New data types
  - Matrices (mat2, mat3, mat4)
  - Vectors (vec2, vec3, vec4, ...)
  - Samplers (sampler1D, sampler2D, ...)
• New qualifiers: in, out, uniform
• Similar to Nvidia’s Cg and Microsoft HLSL
• New OpenGL functions to compile, link, and get information to shaders
Differences between GLSL and C

• Matrix and vector types are built into GLSL
  - they can be passed into and output from GLSL functions, e.g. mat3 func(mat3 a)

• GLSL is designed to be run on massively parallel implementations
  - Recursion is not allowed in GLSL
  - No pointers in GLSL
  - Precision requirements for floats are not as strict as IEEE standards that govern C implementations
GLSL Data Types

- Scalar types: `float`, `int`, `bool`
- Vector types: `vec2`, `vec3`, `vec4`  
  `ivec2`, `ivec3`, `ivec4`  
  `bvec2`, `bvec3`, `bvec4`  
- Matrix types: `mat2`, `mat3`, `mat4`  
- Texture sampling: `sampler1D`, `sampler2D`, `sampler3D`, `samplerCube`  
- C++ Style Constructors
  ```cpp
  vec3 a = vec3(1.0, 2.0, 3.0);
  ```
Qualifiers (1)

- GLSL has many of the same qualifiers as C/C++
- Need others due to the nature of the execution model
- Variables can change
  - Once per primitive
  - Once per vertex
  - Once per fragment
  - At any time in the application
- Vertex attributes are interpolated by the rasterizer into fragment attributes
Qualifiers (2)

- **in, out**
  - Copy vertex attributes and other variable into and out of shaders

  ```
in  vec2 texCoord;
out  vec4 color;
  ```

- **uniform**
  - shader-constant variable from application

  ```
uniform float time;
uniform vec4 rotation;
  ```
Simple Vertex Shader

```glsl
in vec4 vPosition;
void main(void)
{
    gl_Position = vPosition;
}
```

- **input from application; may use `attribute` instead of `in`**
- **must link to variable in application**
- **built-in variable**
Simple Fragment Program

```c
void main(void)
{
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```
Attribute Qualifier

• Attribute-qualified variables can change at most once per vertex

• There are a few built-in variables such as `gl_Position` but most have been deprecated

• User defined (in application program)
  - Use `in` or `attribute` qualifier to get to shader
    - `in float temperature`
    - `attribute vec3 velocity`
Varying Qualified

• Variables that are passed from vertex shader to fragment shader
• Automatically interpolated by the rasterizer
• Old style used the varying qualifier
   
   varying vec4 color;

• Now use **out** in vertex shader and **in** in the fragment shader
  
  out vec4 color;
Attribute and Varying Qualifiers

• Starting with GLSL 1.5 attribute and varying qualifiers have been replaced by in and out qualifiers
• No changes needed in application
• Vertex shader example:

```glsl
#version 1.4
attribute vec3 vPosition;
varying vec3 color;

#version 1.5
in vec3 vPosition;
out vec3 color;
```
Uniform Qualified

• Variables that are constant for an entire primitive
• Can be changed in application and sent to shaders
• Cannot be changed in shader
• Used to pass information to shader such as the bounding box of a primitive
Built-in Variables

- **gl_Position**
  - (required) output position of current vertex
- **gl_PointSize**
  - pixel width/height of the point being rasterized
- **gl_FragCoord**
  - input fragment position
- **gl_FragDepth**
  - input depth value in fragment shader
#version 450

in  vec4 a_Position;
in  vec4 a_Color;
out vec4 color;

void main()
{
    color = a_Color;
    gl_Position = a_Position;
}
#version 450

in vec4 color;
out vec4 fColor; // fragment’s final color

void main()
{
    fColor = color;
    // OR: gl_FragColor = color_out;
}
Operators and Functions

• Standard C functions
  - Arithmetic: sqrt, power, abs
  - Trigonometric: sin, asin
  - Graphical: length, reflect

• Overloading of vector and matrix types

```c
mat4 a;
vec4 b, c, d;
c = b*a; // a row vector stored as a 1D array
d = a*b; // a column vector stored as a 1D array
```
Swizzling and Selection

- Can refer to array elements by element using [ ] or selection (.) operator with
  - x, y, z, w
  - r, g, b, a
  - s, t, p, q
  - a[2], a.b, a.z, a.p are the same

- **Swizzling** operator lets us manipulate components

```cpp
vec4 a;
a.yz = vec2(1.0, 2.0);
```
Programming with OpenGL: More GLSL

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

• Coupling shaders to applications
  - Reading
  - Compiling
  - Linking
• Vertex Attributes
• Setting up uniform variables
• Example applications
Getting Your Shaders into OpenGL

• Shaders need to be compiled and linked to form an executable shader program
• OpenGL provides the compiler and linker
• A program must contain
  - vertex and fragment shaders
  - other shaders are optional

These steps need to be repeated for each type of shader in the shader program
Linking Shaders with Application

• Read shaders
• Compile shaders
• Create a program object
• Link everything together
• Link variables in application with variables in shaders
  - Vertex attributes
  - Uniform variables
Program Object

• Container for shaders
  - Can contain multiple shaders
  - Other GLSL functions

```c
GLuint program = glCreateProgram();

// define shader objects here
glUseProgram (program);
glLinkProgram (program);
```
Reading a Shader

• Shaders are added to the program object and compiled
• Usual method of passing a shader is as a null-terminated string using the function `glShaderSource`
• If the shader is in a file, we can write a reader to convert the file to a string
Adding a Vertex Shader (1)

GLint LoadShaders(const char * vertex_file_path, const char * fragment_file_path) {

    // Create the shaders
    GLuint VertexShaderID   = glCreateShader(GL_VERTEX_SHADER);
    GLuint FragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER);

    // Read the Vertex Shader code from the file
    std::string VertexShaderCode;
    std::ifstream VertexShaderStream(vertex_file_path, std::ios::in);
    if(VertexShaderStream.is_open())
    {
        std::string Line = "";
        while(getline(VertexShaderStream, Line))
            VertexShaderCode += \n + Line;
        VertexShaderStream.close();
    }

    // Read the Fragment Shader code from the file
    std::string FragmentShaderCode;
    std::ifstream FragmentShaderStream(fragment_file_path, std::ios::in);
    if(FragmentShaderStream.is_open()){
        std::string Line = "";
        while(getline(FragmentShaderStream, Line))
            FragmentShaderCode += \n + Line;
        FragmentShaderStream.close();
    }
}
Adding a Vertex Shader (2)

GLint Result = GL_FALSE;
int InfoLogLength;

// Compile Vertex Shader
printf("Compiling shader : %s\n", vertex_file_path);
char const * VertexSourcePointer = VertexShaderCode.c_str();
glShaderSource (VertexShaderID, 1, &VertexSourcePointer , NULL);
glCompileShader(VertexShaderID);

// Check Vertex Shader
glGetShaderiv(VertexShaderID, GL_COMPILE_STATUS, &Result);
glGetShaderiv(VertexShaderID, GL_INFO_LOG_LENGTH, &InfoLogLength);
std::vector<char> VertexShaderErrorMessage(InfoLogLength);
glGetShaderInfoLog(VertexShaderID, InfoLogLength, NULL, &VertexShaderErrorMessage[0]);
fprintf(stdout, "%s\n", &VertexShaderErrorMessage[0]);

// Compile Fragment Shader
printf("Compiling shader : %s\n", fragment_file_path);
char const * FragmentSourcePointer = FragmentShaderCode.c_str();
glShaderSource (FragmentShaderID, 1, &FragmentSourcePointer , NULL);
glCompileShader(FragmentShaderID);

// Check Fragment Shader
glGetShaderiv(FragmentShaderID, GL_COMPILE_STATUS, &Result);
glGetShaderiv(FragmentShaderID, GL_INFO_LOG_LENGTH, &InfoLogLength);
std::vector<char> FragmentShaderErrorMessage(InfoLogLength);
glGetShaderInfoLog(FragmentShaderID, InfoLogLength, NULL, &FragmentShaderErrorMessage[0]);
fprintf(stdout, "%s\n", &FragmentShaderErrorMessage[0]);
// Link the program
fprintf(stdout, "Linking program\n");
GLuint ProgramID = glCreateProgram();
glAttachShader(ProgramID, VertexShaderID);
glAttachShader(ProgramID, FragmentShaderID);
glLinkProgram(ProgramID);

// Check the program
glGetProgramiv(ProgramID, GL_LINK_STATUS, &Result);
glGetProgramiv(ProgramID, GL_INFO_LOG_LENGTH, &InfoLogLength);
std::vector<char> ProgramErrorMessage( max(InfoLogLength, int(1)) );
gGetProgramInfoLog(ProgramID, InfoLogLength, NULL, &ProgramErrorMessage[0]);
fprintf(stdout, "%s\n", &ProgramErrorMessage[0]);

gDeleteShader(VertexShaderID);
gDeleteShader(FragmentShaderID);

return ProgramID;
A Simpler Way

• Qt created a routine to make it easy to load shaders

```cpp
#include <QGLShaderProgram>
QGLShaderProgram program;
program.addShaderFromSourceFile(QGLShader::Vertex, "./vshader.glsl");
program.addShaderFromSourceFile(QGLShader::Fragment, "./fshader.glsl");
```

• Fails if shaders don’t compile, or program doesn’t link

• Add shader programs in qrc file:

```xml
<RCC>
    <qresource prefix="/">
        <file>vshader.glsl</file>
        <file>fshader.glsl</file>
    </qresource>
</RCC>
```
Associating Shader Variables and Data

- Vertex attributes are named in the shaders
- Linker forms a table
- Application can get index from table and tie it to an application variable
- Similar process for uniform variables
Vertex Attribute Example

GLuint positionID = glGetAttribLocation( program, "a_Position" );
glEnableVertexAttribArray( positionID );
glVertexAttribPointer(positionID,
  2, // size
  GL_FLOAT, // type
  GL_FALSE, // normalized?
  0, // stride
  (void *) 0 // array buffer offset
);
Uniform Variable Example

GLint angleID; // location of angle defined in shader
angleID = glGetUniformLocation(program, "angle");

// my_angle set in application
GLfloat my_angle = 5.0 // or some other value

glUniform1f(angleID, my_angle);
Adding Color

• If we set a color in the application, we can send it to the shaders as a vertex attribute or as a uniform variable depending on how often it changes

• Let’s associate a color with each vertex

• Set up an array of same size as positions

• Send to GPU as a vertex buffer object
vec3  base_colors[4] = {vec3(1.0, 0.0, 0.0), ....
vec3  colors[NumVertices];
vec3  points[NumVertices];

// in loop setting positions

colors[i] = base_colors[color_index]
points[i] = .......
Setting Up Buffer Object

void setupBuffer
{
    GLuint vertexbuffer;

    glGenBuffers(1, &vertexbuffer);

    glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);

    glBufferData(GL_ARRAY_BUFFER, sizeof(points) + sizeof(colors), NULL, GL_STATIC_DRAW);

    glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(points), points);
    glBufferSubData(GL_ARRAY_BUFFER, sizeof(points), sizeof(colors), colors);
}
// vPosition and vColor identifiers in vertex shader

loc1 = glGetUniformLocation(program, "vPosition");
glEnableVertexAttribArray(loc1);
glVertexAttribPointer(loc1, 3, GL_FLOAT, GL_FALSE, 0, (void *) 0);

loc2 = glGetUniformLocation(program, "vColor");
glEnableVertexAttribArray(loc2);
glVertexAttribPointer(loc2, 3, GL_FLOAT, GL_FALSE, 0, (void *) sizeof(points));

// draw the triangles
glDrawArrays(GL_TRIANGLES, 0, NumVertices);
Vertex Shader Examples

- A vertex shader is initiated by each vertex output by `glDrawArrays()`.
- A vertex shader must output a position in clip coordinates to the rasterizer.
- Basic uses of vertex shaders:
  - Transformations
  - Lighting
  - Moving vertex positions
Wave Motion Vertex Shader

in vec4 vPosition;
uniform float xs, zs, // frequencies
uniform float h; // height scale
void main()
{
    vec4 t = vPosition;
    t.y = vPosition.y
    + h*sin(time + xs*vPosition.x)
    + h*sin(time + zs*vPosition.z);
    gl_Position = t;
}
Particle System

in vec3 vPosition;
uniform mat4 ModelViewProjectionMatrix;
uniform vec3 init_vel;
uniform float g, m, t;

void main(){
    vec3 object_pos;
    object_pos.x = vPosition.x + vel.x*t;
    object_pos.y = vPosition.y + vel.y*t + g/(2.0*m)*t*t;
    object_pos.z = vPosition.z + vel.z*t;
    gl_Position  = ModelViewProjectionMatrix*vec4(object_pos,1);
}
Pass Through Fragment Shader

// pass-through fragment shader
in vec4 color;
void main(void)
{
    gl_FragColor = color;
}

Angel/Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012
Fragment Shader Applications (1)

Per fragment lighting calculations

per vertex lighting

per fragment lighting
Fragment Shader Applications (2)

Texture mapping

- smooth shading
- environment mapping
- bump mapping
Programming with OpenGL: A Complete Program with Shaders

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

- Build a complete shader-based program
  - Application program (C++)
  - Shaders (GLSL)
    - Vertex shaders
    - Fragment shaders
  - Introduce a standard program structure
- Simple viewing
  - Two-dimensional viewing as a special case of three-dimensional viewing
- Initialization steps and program structure
Modern OpenGL programs essentially do the following steps:

- Create shader programs
- Create buffer objects and load data into them
- “Connect” data locations with shader variables
- Render
Application Program Structure

• Most OpenGL programs have a similar structure that consists of the following functions

  – main():
    • Opens main window with control panel and OpenGL canvas
    • Enters event loop (last executable statement)

  – initializeGL(): sets the state variables
    • Viewing
    • Attributes

  – resizeGL(): handles window resizing event
    • Sets viewport
    • Sets viewing coordinates for orthographic or perspective projection

  – paintGL(): render scene
    • Clear framebuffer
    • Call glVertexArrays() to pump vertices to vertex shader
Key issue is that we must form a data array to send to GPU and then render it.

Once we get data to GPU, we can initiate the rendering with a call to `paintGL()`.

```c
void paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawArrays(GL_TRIANGLES, 0, 3);
}
```

Arrays are buffer objects that contain vertex arrays.
Representing Geometric Objects

• Geometric objects are represented using *vertices*
• A vertex is a collection of generic attributes
  - positional coordinates
  - colors
  - texture coordinates
  - any other data associated with that point in space
• Position stored in 4 dimensional homogeneous coordinates
• Vertex data must be stored in vertex buffer objects (VBOs)
• VBOs may be stored in vertex array objects (VAOs)
OpenGL’s Geometric Primitives

- All primitives are specified by vertices

- GL_POINTS
- GL_LINES
- GL_LINE_STRIP
- GL_LINE_LOOP
- GL_TRIANGLES
- GL_TRIANGLE_STRIP
- GL_TRIANGLE_FAN
Vertex Arrays

- Vertices can have many attributes
  - Position
  - Color
  - Texture Coordinates
  - Application data
- A vertex array holds these data
- Using QVector2D types

```cpp
typedef QVector2D vec2;
vec2 points[3] = {
    vec2(0.0, 0.0), vec2(0.0, 1.0), vec2(1.0, 1.0)
};
```
Vertex Array Object

• Bundles all vertex data (positions, colors, ..,)
• Get name for buffer then bind

   Gluint abuffer;
   glGenVertexArrays(1, &abuffer);
   glBindVertexArray(abuffer);

• At this point we have a current vertex array but no contents
• Use of glBindVertexArray lets us switch between VBOs
Buffer Object

• Buffer objects allow us to transfer large amounts of data to the GPU
• Need to create, bind and identify data

Gluint buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(points), points);

• Data in current vertex array is sent to GPU
Our First Program

• We’ll render a cube with colors at each vertex
• Our example demonstrates:
  - initializing vertex data
  - organizing data for rendering
  - simple object modeling
    • building up 3D objects from geometric primitives
    • building geometric primitives from vertices
Initializing the Cube’s Data (1)

- We’ll build each cube face from individual triangles
- Need to determine how much storage is required
  - (6 faces)(2 triangles/face)(3 vertices/triangle)

```c
const int NumVertices = 36;
```

- To simplify communicating with GLSL, we’ll use a `vec4` class (implemented in C++) similar to GLSL’s `vec4` type
Before we can initialize our VBO, we need to stage the data.

Our cube has two attributes per vertex:
- position
- color

We create two arrays to hold the VBO data:

```cpp
vec4 positions[NumVertices];
vec4 colors    [NumVertices];
```
Cube Data (1)

• Vertices of a unit cube centered at origin
  - sides aligned with axes

```cpp
vec4 xyz[8] = {
  vec4( -0.5, -0.5, 0.5, 1.0 ),
  vec4( -0.5, 0.5, 0.5, 1.0 ),
  vec4( 0.5, 0.5, 0.5, 1.0 ),
  vec4( 0.5, -0.5, 0.5, 1.0 ),
  vec4( -0.5, -0.5, -0.5, 1.0 ),
  vec4( -0.5, 0.5, -0.5, 1.0 ),
  vec4( 0.5, 0.5, -0.5, 1.0 ),
  vec4( 0.5, -0.5, -0.5, 1.0 )
};
```
We’ll also set up an array of RGBA colors

```cpp
vec4 rgba[8] = {
    vec4( 0.0, 0.0, 0.0, 1.0 ),  // black
    vec4( 1.0, 0.0, 0.0, 1.0 ),  // red
    vec4( 1.0, 1.0, 0.0, 1.0 ),  // yellow
    vec4( 0.0, 1.0, 0.0, 1.0 ),  // green
    vec4( 0.0, 0.0, 1.0, 1.0 ),  // blue
    vec4( 1.0, 0.0, 1.0, 1.0 ),  // magenta
    vec4( 1.0, 1.0, 1.0, 1.0 ),  // white
    vec4( 0.0, 1.0, 1.0, 1.0 )   // cyan
};
```
Generating the Cube from Faces

• Generate 12 triangles for the cube
  - 36 vertices with 36 colors

```c
void colorcube()
{
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}
```
Generating a Cube Face from Vertices

To simplify generating the geometry, we use a convenience function quad()
- create two triangles for each face and assign colors to the vertices

```c
int Index = 0;  // global variable indexing into VBO arrays
void quad( int a, int b, int c, int d )
{
    colors[Index] = rgba[a]; positions[Index] = xyz[a]; Index++;
    colors[Index] = rgba[b]; positions[Index] = xyz[b]; Index++;
    colors[Index] = rgba[c]; positions[Index] = xyz[c]; Index++;
    colors[Index] = rgba[a]; positions[Index] = xyz[a]; Index++;
    colors[Index] = rgba[c]; positions[Index] = xyz[c]; Index++;
    colors[Index] = rgba[d]; positions[Index] = xyz[d]; Index++;
}
```

![Diagram of a cube face generated from vertices a, b, c, d]
Storing Vertex Attributes

- Vertex data must be stored in a VBO
- Generate VBO names by calling `glGenBuffers()`
- Bind a specific VBO for initialization by calling

```
glBindBuffer( GL_ARRAY_BUFFER, ... )
```

- load data into VBO using

```
glBufferData( GL_ARRAY_BUFFER, ... )
```
VBOs in Code

• Create and initialize a buffer object

```c
GLuint buffer;
glGenBuffers( 1, &buffer );
glBindBuffer( GL_ARRAY_BUFFER, buffer );
glBufferData( GL_ARRAY_BUFFER,
            sizeof(positions) + sizeof(colors),
            NULL, GL_STATIC_DRAW );
glBufferSubData( GL_ARRAY_BUFFER,
                0,
                sizeof(positions), positions );
glBufferSubData( GL_ARRAY_BUFFER, sizeof(vPositions),
                sizeof(colors), colors );
```
Connecting Vertex Shaders with Geometric Data

- Application vertex data enters the OpenGL pipeline through the vertex shader.
- Need to connect vertex data to shader variables.
  - Requires knowing the attribute location.
- Attribute location can either be queried by calling `glGetVertexAttribLocation()`.
• Associate shader variables with vertex arrays
  - do this after shaders are loaded

```c
GLuint a_Position =
    glGetAttribLocation( program, "a_Position" );
glEnableVertexAttribArray( a_Position );
glVertexAttribPointer( a_Position, 4, GL_FLOAT, GL_FALSE, 0, (void *) 0);

GLuint a_Color =
    glGetAttribLocation( program,"a_Color" );
glEnableVertexAttribArray( a_Color );
glVertexAttribPointer( a_Color, 4, GL_FLOAT, GL_FALSE, 0, (void *) sizeof(a_Positions));
```
A Better Approach

- Associate shader variables with attribute variables
  - do this after shaders are loaded

```c
enum { ATTRIB_VERTEX, ATTRIB_COLOR, ATTRIB_TEXTURE_POSITION };
glBindAttribLocation(program, ATTRIB_VERTEX, "a_Position");
glBindAttribLocation(program, ATTRIB_COLOR, "a_Color");

 glEnableVertexAttribArray(ATTRIB_VERTEX);
 glVertexAttribPointer(ATTRIB_VERTEX, 4, GL_FLOAT, false, 0, NULL);
 glEnableVertexAttribArray(ATTRIB_COLOR);
 glVertexAttribPointer(ATTRIB_COLOR, 4, GL_FLOAT, GL_FALSE, 0, (void *)sizeof(a_Positions));
```
Drawing Geometric Primitives

• For contiguous groups of vertices

```c
glDrawArrays( GL_TRIANGLES, 0, NumVertices );
```

• Usually invoked in display callback
• Initiates vertex shader
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
  - Scalar: number representing magnitude
  - Vector: quantity representing magnitude and direction
  - Point: location in space
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
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

\[ v = P - Q \]
\[ P = v + Q \]
Planes

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

\[ P(\alpha, \beta) = R + \alpha u + \beta v \]

\[ P(\alpha, \beta) = R + \alpha(Q-R) + \beta(P-Q) \]
Triangles

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 triangle
Coordinate System

• Consider a basis \( v_1, v_2, \ldots, v_n \) (such as the x, y, and z axes)

• A vector is written \( v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n \)

• 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: \( a = [\alpha_1 \ \alpha_2 \ \ldots \ \alpha_n]^T = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \cdot \\ \cdot \\ \alpha_n \end{bmatrix} \)

Ex: \( v = 2v_1 + 3v_2 - 4v_3 \rightarrow a = [2 \ 3 \ -4]^T \)

This representation is with respect to a particular basis
To form a coordinate frame, we must add a single point, the origin $P_0$, to the basis vectors.
Representation of Vectors and Points

• Frame determined by \((P_0, v_1, v_2, v_3)\)
• Within this frame, every vector can be written as
  \[ v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n \]
  Vector is just direction
• Every point can be written as
  \[ P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \ldots + \beta_n v_n \]
  Point is anchored: displaced from origin
Confusing Points and Vectors

• Points and vectors appear to have similar representations
  \[ \mathbf{p} = [\beta_1 \beta_2 \beta_3] \]
  \[ \mathbf{v} = [\alpha_1 \alpha_2 \alpha_3] \]

• A vector has no position.

Vector can be placed anywhere

point: fixed
A Single Representation

If we define $0 \cdot P = 0$ and $1 \cdot P = P$ then we can write

$$v = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 = [\alpha_1 \ \alpha_2 \ \alpha_3 \ \ 0] \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix}^T$$

$$P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \beta_3 v_3 = [\beta_1 \ \beta_2 \ \beta_3 \ \ 1] \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix}^T$$

Thus we obtain the 4D homogeneous coordinate representation

$$v = [\alpha_1 \ \alpha_2 \ \alpha_3 \ \ 0]^T$$

$$p = [\beta_1 \ \beta_2 \ \beta_3 \ \ 1]^T$$
Homogeneous Coordinates

The homogeneous coordinates form for a three dimensional point \([x \ y \ z]\) is given as
\[
p = [x' \ y' \ z' \ w]^T = [wx \ wy \ wz \ w]^T
\]
We return to a three dimensional point (for \(w \neq 0\)) by
\[
\begin{align*}
x &\leftarrow x'/w \\
y &\leftarrow y'/w \\
z &\leftarrow z'/w
\end{align*}
\]
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.
For \(w=1\), the representation of a point is \([x \ y \ z \ 1]\)
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 $4 \times 4$ matrices
  - Hardware pipeline works with 4D representations
  - For orthographic viewing, we can maintain $w = 0$ for vectors and $w = 1$ for points
  - For perspective, we need a *perspective division*
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

\[ Q = T(P) \]

\[ v = T(u) \]
Pipeline Implementation

The pipeline process involves transforming vertices from the application program into pixels on the frame buffer. The process includes:

1. **Transformation**: This stage applies transformations to the vertices, including scaling, rotation, and translation, using matrices \( T \) provided by the application program.

2. **Rasterizer**: This stage rasterizes the transformed vertices into pixels on the frame buffer. The transformed vertices \( T(u) \) and \( T(v) \) are input to the rasterizer, resulting in output pixels.

The diagram illustrates the flow of data from vertices to pixels, with intermediate steps for transformation and rasterization.
Translation

• Move (translate, displace) a point to a new location

• Displacement determined by a vector $d$
  - Three degrees of freedom
  - $P' = P + d$
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
Translation Matrix

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

\[
p' = Tp \text{ where} \]

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

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 $\theta$

\[
x = r \cos (\phi + \theta)
\]
\[
y = r \sin (\phi + \theta)
\]

\[
x' = x \cos \theta - y \sin \theta
\]
\[
y' = x \sin \theta + y \cos \theta
\]

\[
x = r \cos \phi
\]
\[
y = r \sin \phi
\]
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
    \[
    x' = x \cos \theta - y \sin \theta \\
    y' = x \sin \theta + y \cos \theta \\
    z' = z
    \]
  - or in homogeneous coordinates
    \[p' = R_z(\theta)p\]
Rotation Matrix

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

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

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

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

\[ x' = s_x x \]
\[ y' = s_y y \]
\[ z' = s_z z \]

\[ p' = Sp \]

\[
S = S(s_x, s_y, s_z) = \begin{bmatrix}
    s_x & 0 & 0 & 0 \\
    0 & s_y & 0 & 0 \\
    0 & 0 & s_z & 0 \\
    0 & 0 & 0 & 1
\end{bmatrix}
\]
Reflection
corresponds to negative scale factors

\[ s_x = -1 \quad s_y = 1 \]

original

\[ s_x = -1 \quad s_y = -1 \]

\[ s_x = 1 \quad s_y = -1 \]
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 \( \sin(-\theta) = -\sin(\theta) \)
      \( R^{-1}(\theta) = R^T(\theta) \)
  - 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
  \[ p'^T = p^T C^T B^T A^T \]
A rotation by $\theta$ about an arbitrary axis can be decomposed into the concatenation of rotations about the $x$, $y$, and $z$ axes.

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

$\theta_x \ \theta_y \ \theta_z$ 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
\[ M = T(p_f) \ R(\theta) \ T(-p_f) \]
Shear

• Helpful to add one more basic transformation
• Equivalent to pulling faces in opposite directions
Shear Matrix

Consider simple shear along the $x$ axis

\[ x' = x + y \cot \theta \]
\[ y' = y \]
\[ z' = z \]

\[ H(\theta) = \begin{bmatrix}
1 & \cot \theta & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix} \]
3D Transformations

• A vertex is transformed by $4 \times 4$ matrices
• All matrices are stored column-major in OpenGL
  - this is opposite of what “C” programmers expect
• Matrices are always post-multiplied
  - product of matrix and vector is $M\vec{v}$

$$M = \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix} \quad v = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix}$$
Affine Transformations

\[
\begin{bmatrix}
x' \\
y' \\
z' \\
1
\end{bmatrix} =
\begin{bmatrix}
m_0 & m_4 & m_8 & m_{12} \\
m_1 & m_5 & m_9 & m_{13} \\
m_2 & m_6 & m_{10} & m_{14} \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix}
\]

• Characteristic of many important transformations
  - Translation
  - Rotation
  - Scaling
  - Shear

• Line preserving
OpenGL Transformations

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

- Learn how to carry out transformations in OpenGL
  - Rotation
  - Translation
  - Scaling
- Introduce QMatrix4x4 and QVector3D transformations
  - Model-view
  - 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: \( \mathbf{C} \leftarrow \mathbf{I} \)
  Load an arbitrary matrix: \( \mathbf{C} \leftarrow \mathbf{M} \)

  Load a translation matrix: \( \mathbf{C} \leftarrow \mathbf{T} \)
  Load a rotation matrix: \( \mathbf{C} \leftarrow \mathbf{R} \)
  Load a scaling matrix: \( \mathbf{C} \leftarrow \mathbf{S} \)

  Postmultiply by an arbitrary matrix: \( \mathbf{C} \leftarrow \mathbf{CM} \)
  Postmultiply by a translation matrix: \( \mathbf{C} \leftarrow \mathbf{CT} \)
  Postmultiply by a rotation matrix: \( \mathbf{C} \leftarrow \mathbf{CR} \)
  Postmultiply by a scaling matrix: \( \mathbf{C} \leftarrow \mathbf{CS} \)
Rotation about a Fixed Point

Start with identity matrix: \( C \leftarrow I \)
Move fixed point to origin: \( C \leftarrow CT \)
Rotate: \( C \leftarrow CR \)
Move fixed point back: \( C \leftarrow CT^{-1} \)

Result: \( C = TR T^{-1} \) which is **backwards**.

This result is a consequence of doing postmultiplications. Let’s try again.
Reversing the Order

We want $C = T^{-1} R T$ so we must do the operations in the following order:

1. $C \leftarrow I$
2. $C \leftarrow CT^{-1}$
3. $C \leftarrow CR$
4. $C \leftarrow CT$

Each operation corresponds to one function call in the program.

The last operation specified is the first executed in the program!
Rotation, Translation, Scaling

Create an identity matrix:

\[
\begin{align*}
\text{QMatrix4x4 } m; \\
m.\text{setToIdentity}();
\end{align*}
\]

Multiply on right by rotation matrix of \textit{theta} in degrees where \((v_x, v_y, v_z)\) define axis of rotation

\[
\begin{align*}
m.\text{rotate}(\text{theta}, \text{QVector3D}(v_x, v_y, v_z));
\end{align*}
\]

Do same with translation and scaling:

\[
\begin{align*}
m.\text{scale}(s_x, s_y, s_z); \\
m.\text{translate}(d_x, d_y, d_z);
\end{align*}
\]
Example

- Rotation about z axis by 30 degrees with a fixed point of (1.0, 2.0, 3.0)

```cpp
QMatrix4x4 m;
m.setToIdentity();
m.translate(1.0, 2.0, 3.0);
m.rotate (30.0, QVector3D(0.0, 0.0, 1.0));
m.translate(-1.0,-2.0,-3.0);
```

- Remember that the last matrix specified is the first applied
Arbitrary Matrices

• Can load and multiply by matrices defined in the application program
• Matrices are stored as one dimensional array of 16 elements which are the components of the desired 4 x 4 matrix stored by columns
• OpenGL functions that have matrices as parameters allow the application to send the matrix or its transpose
in vec4 vPosition;
in vec4 vColor;
out vec4 color;
uniform vec3 theta;

void main()
{
    // Compute the sines and cosines of theta for
    // each of the three axes in one computation.
    vec3 angles = radians( theta );
    vec3 c = cos( angles );
    vec3 s = sin( angles );
// Remember: these matrices are column-major

mat4 rx = mat4( 1.0,  0.0,  0.0, 0.0,
                0.0,  c.x,  s.x, 0.0,
                0.0, -s.x,  c.x, 0.0,
                0.0,  0.0,  0.0, 1.0 );

mat4 ry = mat4( c.y,  0.0, -s.y, 0.0,
                0.0,  1.0,  0.0, 0.0,
                s.y,  0.0,  c.y, 0.0,
                0.0,  0.0,  0.0, 1.0 );
Vertex Shader for Rotation of Cube (3)

```c
mat4 rz = mat4( c.z, -s.z, 0.0, 0.0,
               s.z,  c.z, 0.0, 0.0,
               0.0,  0.0, 1.0, 0.0,
               0.0,  0.0, 0.0, 1.0 );

color = vColor;
gl_Position = rz * ry * rx * vPosition;
```

Sending Angles from Application

GLuint thetaID;  // theta uniform location
vec3 theta;     // axis angles

void display( void )
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glUniform3fv( thetaID, 1, theta );
    glDrawArrays( GL_TRIANGLES, 0, NumVertices );
}
Computer Viewing

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

• Introduce the mathematics of projection
• Introduce OpenGL viewing functions
• Look at alternate viewing APIs
Computer Viewing

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

• Transformations take us from one “space” to another
  - All of our transforms are $4 \times 4$ matrices
Transformations

• Modeling transformations
  - move models into world coordinate system

• Viewing transformations
  - define position and orientation of the camera

• Projection transformations
  - adjust the lens of the camera; define view volume

• Viewport transformations
  - enlarge or reduce the physical photograph
The Model Coordinate System

The X,Y,Z coordinates of the model’s vertices are defined relative to the object’s center, where (0,0,0) is the center of the object.
The World Coordinate System

The model is moved to a new position, and possibly included with other models, in the world coordinate system.
The View Coordinate System

The vertices expressed in the world coordinate system must be transformed into the view coordinate system since they are now relative to the camera.
Model-View Transformation

- A 4x4 matrix transforms vertices from the model to the world coordinate system.
- A second 4x4 matrix maps the world to the view coordinate system.
- The product of these two matrices is called the model-view matrix.
- It maps the object from the original model coordinate system directly to the camera’s (viewer’s) coordinate system.
The World and Camera Frames

- Changes in frame are defined by 4 x 4 matrices
- In OpenGL, we start with the world frame
- We move models from the world frame to the camera frame by using the model-view matrix \( M \)
- Initially these frames are the same \((M=I)\)
- If you want to move the camera three units to the right \((+x)\), this is achieved by moving the objects three units to the left \((-x)\).
- Camera always stays at the origin and points in the negative z direction
Moving the Objects

Move objects back (along –z direction) to view it in front of camera, which is at origin.

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

- Position the camera/eye in the scene
  - place the tripod down; aim camera
- To “fly through” a scene
  - change viewing transformation and redraw scene

- Simple interface:
  \( \text{LookAt}(\text{eyex}, \text{eyey}, \text{eyez}, \text{atx}, \text{aty}, \text{atz}, \text{upx}, \text{upy}, \text{upz}) \)
  - up vector determines unique orientation
  - careful of degenerate positions
LookAt

LookAt(eye, at, up)
Creating the LookAt Matrix

\[ \hat{n} = \frac{\vec{at} - \vec{eye}}{||\vec{at} - \vec{eye}||} \]

\[ \hat{u} = \frac{\hat{n} \times \vec{up}}{||\hat{n} \times \vec{up}||} \]

\[ \hat{v} = \hat{u} \times \hat{n} \]

\[ \begin{pmatrix} u_x & u_y & u_z & - (\vec{eye} \cdot \hat{u}) \\ v_x & v_y & v_z & - (\vec{eye} \cdot \hat{v}) \\ -n_x & -n_y & -n_z & - (\vec{eye} \cdot \hat{n}) \\ 0 & 0 & 0 & 1 \end{pmatrix} \]
Specifying What You Can See (1)

• Once camera is positioned in scene, we must set up a viewing frustum (view volume) to specify how much of the world we can see.

• Done in two steps
  - specify the size of the frustum (projection transform)
  - specify its location in space (model-view transform)

• Anything outside of viewing frustum is clipped
  - primitive is either modified or discarded (if entirely outside frustum)
Specifying What You Can See (2)

- OpenGL projection model uses eye coordinates
  - the “eye” is located at the origin
  - looking down the -z axis
- Projection matrices use a six-plane model:
  - near (image) plane and far (infinite) plane
    - both are distances from the eye (positive values)
  - enclosing planes
    - top & bottom, left & right
Specifying What You Can See (3)

\[ O = \begin{pmatrix} \frac{2}{r \cdot l} & 0 & 0 & \frac{r+1}{r \cdot l} \\ 0 & \frac{2}{t \cdot b} & 0 & \frac{t+b}{t \cdot b} \\ 0 & 0 & \frac{2}{f \cdot n} & \frac{f+n}{f \cdot n} \\ 0 & 0 & 0 & 1 \end{pmatrix} \]

\[ P = \begin{pmatrix} \frac{2n}{r \cdot l} & 0 & \frac{r+1}{r \cdot l} & 0 \\ 0 & \frac{2n}{t \cdot b} & \frac{t+b}{t \cdot b} & 0 \\ 0 & 0 & \frac{(f+n)}{f \cdot n} & \frac{2fn}{f \cdot n} \\ 0 & 0 & 1 & 0 \end{pmatrix} \]
Perspective Projection
Parallel Projection

Object

Projector

Projection plane

DOP
Orthographic Projection

- Projectors are orthogonal to projection surface.
- Special (and most common) case of parallel projections
Default OpenGL Viewing

- Default view volume is a cube with sides of length 2 centered at the origin (from -1 to 1)
- Default projection is orthographic
- For points within the default view volume:

  \[ x_p = x \]
  \[ y_p = y \]
  \[ z_p = 0 \]
Orthogonal Normalization

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

normalization \(\Rightarrow\) find transformation to convert specified clipping volume to default cube

\(\text{near}\) and \(\text{far}\) measured from camera
Orthogonal Matrix

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

\[
P = ST = \begin{bmatrix}
\frac{2}{\text{right} - \text{left}} & 0 & 0 & -\frac{\text{right} - \text{left}}{\text{right} - \text{left}} \\
0 & \frac{2}{\text{top} - \text{bottom}} & 0 & -\frac{\text{top + bottom}}{\text{top} - \text{bottom}} \\
0 & 0 & \frac{2}{\text{near} - \text{far}} & -\frac{\text{far + near}}{\text{far} - \text{near}} \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
Final Projection

• Set \( z = 0 \)

• Equivalent to the homogeneous coordinate transformation

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

\[ M_{\text{orth}} = \]

• Hence, general orthogonal projection in 4D is

\[ P = M_{\text{orth}}ST \]
OpenGL Perspective

frustum(left, right, bottom, top, near, far)
Using Field of View

- It is often difficult to get desired view with `frustum()`
- `perspective(fovy, aspect, near, far)` often provides a better interface

\[ \text{aspect} = \frac{w}{h} \]
QMatrix4x4 Projection;
Projection.perspective(
    45.0f, // vertical field of view
    4.0f/3.0f, // aspect ratio
    0.1f, // near clipping plane
    100.0f // far clipping plane
)
Model-view and Projection Matrices

• In OpenGL the model-view matrix is used to
  - Position the camera
    • Easily done by using a LookAt function
  - Build models of objects
    • Positioning model elements together in world coordinates

• The projection matrix is used to define the view volume and to select a camera lens

• We create the model-view and projection matrices in our own applications and pass them to the vertex shader
Putting It All Together (1)

```cpp
QMatrix4x4 Projection, View, Model, MVP;
Projection.perspective(
    45.0f, // vertical field of view
    4.0f/3.0f, // aspect ratio
    0.1f, // near clipping plane
    100.0f // far clipping plane
);

View.lookAt(
    vec3(4, 3, 3); // camera in world space
    vec3(0, 0, 0); // and looks at the origin
    vec3(0, 1, 0); // up direction
);

Model.setToIdentity(); // model matrix is identity

MVP = Projection * View * Model; // composite matrix
```
// get a handle for our "u_MVP" uniform at initialization time
GLuint MatrixID = glGetUniformLocation(programID, "u_MVP");

// send our transformation to the currently bound shader
// in the "u_MVP" uniform
glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP[0][0]);

In vertex shader:

    in  vec4  a_Position;  // vertex position
    uniform  mat4  u_MVP;  // Projection * Modelview
    void  main()
    {
        gl_Position = u_MVP * a_Position;
    }
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 projection normalization
Simple Perspective

- Center of projection at the origin
- Projection plane $z = d, d < 0$
Perspective Equations

Consider top and side views

\[
x_p = \frac{x}{z/d} \quad \quad y_p = \frac{y}{z/d} \quad \quad z_p = d
\]
Consider $p = Mq$ where:

\[
\begin{bmatrix}
  x \\
  y \\
  z \\
  z/d
\end{bmatrix} =
\begin{bmatrix}
  1 & 0 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 & 0 \\
  0 & 0 & 1 & 0 & 0 \\
  0 & 0 & 1/d & 0 & 0
\end{bmatrix}
\begin{bmatrix}
  x \\
  y \\
  z \\
  1
\end{bmatrix}
\]
Perspective Division

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

• This *perspective division* yields

\[
\begin{align*}
x_p &= \frac{x}{z/d} \\
y_p &= \frac{y}{z/d} \\
z_p &= d
\end{align*}
\]

the desired perspective equations
Pipeline View

modelview transformation → projection transformation → perspective division

4D → 3D

nonsingular

clipping → projection

against default cube

3D → 2D
Model-view and Projection Matrices

• In OpenGL the model-view matrix is used to
  - Position the camera
    • Easily done by using the LookAt function
  - Build models of objects
    • Positioning model elements together in world coordinates

• The projection matrix is used to define the view volume and to select a camera lens
  - ortho(left, right, bottom, top, near, far)
  - perspective(fovy, aspect, near, far)
View Normalization

• Rather than derive a different projection matrix for orthographic and perspective projections, 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.
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 $x = \pm z$, $y = \pm z$
Perspective Matrices

Simple projection matrix in homogeneous coordinates

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

Note that \(-1 = 1/d\) where \(d = -1\) and that \(M\) is independent of the far clipping plane.
Generalization

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

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

\[
x'' = \frac{x}{z}
\]

\[
y'' = \frac{y}{z}
\]

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

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

If we pick

$$\alpha = \frac{\text{near} + \text{far}}{\text{far} - \text{near}}$$

$$\beta = \frac{2\text{near} \times \text{far}}{\text{near} - \text{far}}$$

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

original clipping volume

original object

new clipping volume

distorted object projects correctly

$z = -x$

$z = x$

$z = -\text{far}$

$z = -\text{near}$

$z = 1$

$x = -1$

$x = 1$

$z = -1$
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 Matrix

• The normalization in \texttt{frustum()} 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

\[
P = NSH
\]

our previously defined perspective matrix  
shear and scale
Why do we do it this way?

• Normalization allows for a single pipeline for both perspective and orthogonal viewing
• We stay in four dimensional homogeneous coordinates as long as possible to retain three-dimensional information needed for hidden-surface removal and shading
• We simplify clipping
A perspective projection of the scene is generated as the rays that connect the vertices to the center of projection intersect the viewplane. The view volume consists of a frustum (truncated pyramid) extending from the camera.
Before Projection

Before projection, we have the blue objects in camera space and the red camera frustum.
After Projection

Multiplying everything by the projection matrix has the following effect: the frustum is now a unit cube and the blue objects have been deformed.
View from Behind Frustum
Resized to Window
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
• Deprecated 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

• Define each polygon by the geometric locations of its vertices

• Leads to OpenGL code such as

```c
vertex[i] = vec3(x1, x1, x1);
vertex[i+1] = vec3(x6, x6, x6);
vertex[i+2] = vec3(x7, x7, x7);
i+=3;
```

• Inefficient and unstructured
  - Consider moving a vertex to a new location
  - Must search for all occurrences
Inward and Outward Facing Polygons

• The order \( \{v_1, v_6, v_7\} \) and \( \{v_6, v_7, v_1\} \) are equivalent in that the same polygon will be rendered by OpenGL but the order \( \{v_1, v_7, v_6\} \) is different.

• The first two describe *outwardly facing* polygons.

• Use the *right-hand rule* = counter-clockwise encirclement of outward-pointing normal.

• OpenGL can treat 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*
Edge List

Note polygons are not represented
Modeling a Cube

Define global arrays for vertices and colors

typedef vec3 point3;
point3 vertices[] = {
    point3(-1.0,-1.0,-1.0), point3( 1.0,-1.0,-1.0),
    point3( 1.0, 1.0,-1.0), point3(-1.0, 1.0,-1.0),
    point3(-1.0,-1.0, 1.0), point3( 1.0,-1.0, 1.0),
    point3( 1.0, 1.0, 1.0), point3(-1.0, 1.0, 1.0)
};

typedef vec3 color3;
color3 colors[] = {
    color3(0.0,0.0,0.0), color3(1.0,0.0,0.0),
    color3(1.0,1.0,0.0), color3(0.0,1.0,0.0),
    color3(0.0,0.0,1.0), color3(1.0,0.0,1.0),
    color3(1.0,1.0,1.0), color3(0.0,1.0,1.0)
};
Draw a triangle from a list of indices into the array vertices and assign a color to each index.

```c
void triangle(int a, int b, int c, int d) {
    vcolors[i]   = colors[d];
    position[i]  = vertices[a];
    vcolors[i+1] = colors[d];
    position[i+1] = vertices[a];
    vcolors[i+2] = colors[d];
    position[i+2] = vertices[a];
    i+=3;
}
```
void colorcube( )
{
    quad(0,3,2,1);
    quad(2,3,7,6);
    quad(0,4,7,3);
    quad(1,2,6,5);
    quad(4,5,6,7);
    quad(0,1,5,4);
}

Note that vertices are ordered so that we obtain correct outward facing normals
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 straightforward way used to require:
  - 6 glBegin, 6 glEnd
  - 6 glColor
  - 24 glVertex
  - More if we use texture and lighting
Mapping indices to faces

- Form an array of face indices

```c
GLubyte 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

• Old Method:

```c
glDrawElements(GL_QUADS, 24,
               GL_UNSIGNED_BYTE, cubeIndices);
```

[Draws cube with 1 function call!!]

• Problem is that although we avoid many function calls, data are still on client side

• Solution:
  - no immediate mode
  - Vertex buffer object
  - Use glDrawArrays
Rotating Cube

• Full example
• Model Colored Cube
• Use 3 button mouse to change direction of rotation
• Use idle function to increment angle of rotation
Cube Vertices

// Vertices of a unit cube centered at origin
// sides aligned with axes
point4 vertices[8] = {
    point4( -0.5, -0.5, 0.5, 1.0 ),
    point4( -0.5, 0.5, 0.5, 1.0 ),
    point4( 0.5, 0.5, 0.5, 1.0 ),
    point4( 0.5, -0.5, 0.5, 1.0 ),
    point4( -0.5, -0.5, -0.5, 1.0 ),
    point4( -0.5, 0.5, -0.5, 1.0 ),
    point4( 0.5, 0.5, -0.5, 1.0 ),
    point4( 0.5, -0.5, -0.5, 1.0 )
};
// RGBA colors

color4 vertex_colors[8] = {
    color4( 0.0, 0.0, 0.0, 1.0 ),  // black
    color4( 1.0, 0.0, 0.0, 1.0 ),  // red
    color4( 1.0, 1.0, 0.0, 1.0 ),  // yellow
    color4( 0.0, 1.0, 0.0, 1.0 ),  // green
    color4( 0.0, 0.0, 1.0, 1.0 ),  // blue
    color4( 1.0, 0.0, 1.0, 1.0 ),  // magenta
    color4( 1.0, 1.0, 1.0, 1.0 ),  // white
    color4( 0.0, 1.0, 1.0, 1.0 )   // cyan
};
// quad generates two triangles for each face
// and assigns colors to the vertices
int Index = 0;
void quad( int a, int b, int c, int d )
{
    colors[Index] = vertex_colors[a]; points[Index] = vertices[a]; Index++;
    colors[Index] = vertex_colors[b]; points[Index] = vertices[b]; Index++;
    colors[Index] = vertex_colors[c]; points[Index] = vertices[c]; Index++;
    colors[Index] = vertex_colors[a]; points[Index] = vertices[a]; Index++;
    colors[Index] = vertex_colors[c]; points[Index] = vertices[c]; Index++;
    colors[Index] = vertex_colors[d]; points[Index] = vertices[d]; Index++;
}
// generate 12 triangles: 36 vertices and 36 colors
void colorcube()
{
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}
void init()
{
    colorcube();

    // create a vertex array object
    GLuint vao;
    glGenVertexArrays ( 1, &vao );
    glBindVertexArray ( vao );

    // create and initialize a buffer object
    GLuint buffer;
    glGenBuffers( 1, &buffer );
    glBindBuffer( GL_ARRAY_BUFFER, buffer );
    glBufferData( GL_ARRAY_BUFFER, sizeof(points) + sizeof(colors), NULL,
                 GL_STATIC_DRAW );
    glBufferSubData( GL_ARRAY_BUFFER, 0, sizeof(points), points );
    glBufferSubData( GL_ARRAY_BUFFER, sizeof(points), sizeof(colors), colors );

    // load shaders and use the resulting shader program
    GLuint program = InitShader( "vshader36.glsl", "fshader36.glsl" );
    glUseProgram( program );
// set up vertex arrays
GLuint vPosition = glGetUniformLocation(program, "vPosition");
glEnableVertexAttribArray(vPosition);
glVertexAttribPointer(vPosition, 4, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(0));

GLuint vColor = glGetUniformLocation(program, "vColor");
glEnableVertexAttribArray(vColor);
glVertexAttribPointer(vColor, 4, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(sizeof(points)));

theta = glGetUniformLocation(program, "theta");
}
void paintGL( void )
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    glUniform3fv( theta, 1, theta );
    glDrawArrays( GL_TRIANGLES, 0, NumVertices );
}
enum {Xaxis, Yaxis, Zaxis};

void mousePressEvent(QMouseEvent *e)
{
    m_mousePressPosition = QVector2D(e->pos());
    switch( e->button() ) {
        case Qt::LeftButton:  m_axis = Xaxis;  break;
        case Qt::MidButton:   m_axis = Yaxis;  break;
        case Qt::RightButton: m_axis = Zaxis;  break;
    }
}

// Virtual function called when timer times out.
void timerEvent(QTimerEvent *e)
{
    // avoid compiler warning for unused event e
    Q_UNUSED(e);

    // update appropriate theta based on m_axis
    theta[m_axis] += 0.01;

    if( theta[m_axis] > 360.0 )
        theta[m_axis] -= 360.0;

    updateGL();

    // restart animation
    m_timer->start(10, this);
}
Programming with OpenGL: Color and Attributes

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

• Expanding primitive set
• Adding color
• Vertex attributes
• Uniform variables
OpenGL Primitives

GL_POINTS

GL_LINES

GL_LINE_STRIP

GL_LINE_LOOP

GL_TRIANGLES

GL_TRIANGLE_STRIP

GL_TRIANGLE_FAN
Polygon Issues

• OpenGL will only display triangles
  - 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
• Application program must tessellate a polygon into triangles (triangulation)
• OpenGL 4.1 contains a tessellator

[Diagram of nonsimple polygon and nonconvex polygon]
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.
Good and Bad Triangles

• Long thin triangles render badly

• Equilateral triangles render well
• Maximize minimum angle
• Delaunay triangulation for unstructured points
Triangularization

- Convex polygon

- Start with abc, remove b, then acd, ....
Non-convex (concave)
Recursive Division

- Find leftmost vertex and split
Attributes

- Attributes 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
    - Display vertices
- Only a few (gl_PointSize) are supported by OpenGL functions
RGB color

- Each color component is stored separately in the frame buffer.
- Usually 8 bits per component in buffer.
- Color values can range from 0.0 (none) to 1.0 (all) using floats or over the range from 0 to 255 using unsigned bytes.
Smooth Color

- Default is *smooth* shading
  - OpenGL interpolates vertex colors across visible polygons
- Alternative is *flat shading*
  - Color of first vertex determines fill color
  - Handle in shader
Setting Colors

• Colors are ultimately set in the fragment shader but can be determined in either shader or in the application
• Application color: pass to vertex shader as a uniform variable or as a vertex attribute
• Vertex shader color: pass to fragment shader as varying variable
• Fragment color: can alter via shader code
Shading

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

• Learn to shade objects so their images appear three-dimensional
• Introduce the types of light-material interactions
• Build a simple reflection model (Phong model) that can be used with real-time graphics hardware
Why We Need Shading

• Suppose we build a model of a sphere using many polygons and color it with `glColor`. We get something like

• But we want
Shading

• Why does the image of a real sphere look like

• Light-material interactions cause each point to have a different color or shade

• Need to consider
  - Light sources
  - Material properties
  - Location of viewer
  - Surface orientation
Scattering

• Light strikes A
  - Some scattered
  - Some absorbed

• Some of scattered light strikes B
  - Some scattered
  - Some absorbed

• Some of this scattered light strikes A and so on
Rendering Equation

• The infinite scattering and absorption of light can be described by the rendering equation
  - Cannot be solved in general
  - Ray tracing is a special case for perfectly reflecting surfaces

• Rendering equation is global and includes
  - Shadows
  - Multiple scattering from object to object
Global Effects

- translucent surface
- shadow
- multiple reflection
Local vs Global Rendering

• Correct shading requires a global calculation involving all objects and light sources
  - Incompatible with pipeline model which shades each polygon independently (local rendering)

• However, in computer graphics, especially real time graphics, we are happy if things “look right”
  - Exist many techniques for approximating global effects
Light-Material Interaction

• Light that strikes an object is partially absorbed and partially scattered (reflected)
• The amount reflected determines the color and brightness of the object
  - A surface appears red under white light because the red component of the light is reflected and the rest is absorbed
• The reflected light is scattered in a manner that depends on the smoothness and orientation of the surface
Light Sources

General light sources are difficult to work with because we must integrate light coming from all points on the source.
Simple Light Sources

• Point source
  - Model with position and color
  - Distant source = infinite distance away (parallel)

• Spotlight
  - Restrict light from ideal point source

• Ambient light
  - Same amount of light everywhere in scene
  - Can model contribution of many sources and reflecting surfaces
Surface Types

• The smoother a surface, the more reflected light is concentrated in the direction a perfect mirror would reflected the light

• A very rough surface scatters light in all directions

smooth surface  rough surface
Phong Model

- A simple model that can be computed rapidly
- Has three components
  - Diffuse
  - Specular
  - Ambient
- Uses four vectors
  - To light source
  - To viewer
  - Normal
  - Perfect reflector
**Ideal Reflector**

- Normal is determined by local orientation
- Angle of incidence = angle of reflection
- The three vectors must be coplanar

\[
\mathbf{r} = 2 (\mathbf{l} \cdot \mathbf{n}) \mathbf{n} - \mathbf{l}
\]
Lambertian Surface

- Perfectly diffuse reflector
- Light scattered equally in all directions
- Amount of light reflected is proportional to the vertical component of incoming light
  - reflected light $\sim \cos \theta_i$
  - $\cos \theta_i = \mathbf{l} \cdot \mathbf{n}$ if vectors normalized
  - There are also three coefficients, $k_r$, $k_b$, $k_g$ that show how much of each color component is reflected
Specular Surfaces

- Most surfaces are neither ideal diffusers nor perfectly specular (ideal reflectors)
- Smooth surfaces show specular highlights due to incoming light being reflected in directions concentrated close to the direction of a perfect reflection
Modeling Specular Reflections

- Phong proposed using a term that dropped off as the angle between the viewer and the ideal reflection increased.

\[ I_r \sim k_s I \cos^\alpha \phi \]

- reflected intensity
- shininess coef
- incoming intensity
- absorption coef
The Shininess Coefficient

- Values of $\alpha$ between 100 and 200 correspond to metals
- Values between 5 and 10 give surface that look like plastic
Ambient Light

• Ambient light is the result of multiple interactions between (large) light sources and the objects in the environment.
• Amount and color depend on both the color of the light(s) and the material properties of the object.
• Add $k_a I_a$ to diffuse and specular terms.

reflection coef  intensity of ambient light
Distance Terms

- The light from a point source that reaches a surface is inversely proportional to the square of the distance between them.
- We can add a factor of the form $\frac{1}{a + bd + cd^2}$ to the diffuse and specular terms.
- The constant and linear terms soften the effect of the point source.
Light Sources

• In the Phong Model, we add the results from each light source
• Each light source has separate diffuse, specular, and ambient terms to allow for maximum flexibility even though this form does not have a physical justification
• Separate red, green and blue components
• Hence, 9 coefficients for each point source
  - $I_{dr}$, $I_{dg}$, $I_{db}$, $I_{sr}$, $I_{sg}$, $I_{sb}$, $I_{ar}$, $I_{ag}$, $I_{ab}$
Material Properties

• Material properties match light source properties
  - Nine absorption coefficients
    • $k_{dr}$, $k_{dg}$, $k_{db}$, $k_{sr}$, $k_{sg}$, $k_{sb}$, $k_{ar}$, $k_{ag}$, $k_{ab}$
  - Shininess coefficient $\alpha$
Adding up the Components

For each light source and each color component, the Phong model can be written (without the distance terms) as

\[ I = k_d I_d \cdot n + k_s I_s (v \cdot r)^\alpha + k_a I_a \]

For each color component we add contributions from all sources.
Modified Phong Model

• The specular term in the Phong model is problematic because it requires the calculation of a new reflection vector and view vector for each vertex

• Blinn suggested an approximation using the halfway vector that is more efficient
The Halfway Vector

- $h$ is normalized vector halfway between $l$ and $v$

$$h = \frac{l + v}{|l + v|}$$
Using the halfway vector

• Replace \((\mathbf{v} \cdot \mathbf{r})^\alpha\) by \((\mathbf{n} \cdot \mathbf{h})^\beta\)

• \(\beta\) is chosen to match shininess

• Note that halfway angle is half of angle between \(\mathbf{r}\) and \(\mathbf{v}\) if vectors are coplanar

• Resulting model is known as the modified Phong or Blinn lighting model
  - Specified in OpenGL standard
Example

Only differences in these teapots are the parameters in the Phong model.
Computation of Vectors

- $\mathbf{l}$ and $\mathbf{v}$ are specified by the application.
- Can compute $\mathbf{r}$ from $\mathbf{l}$ and $\mathbf{n}$.
- Problem is determining $\mathbf{n}$, which depends on underlying representation of surface.
Computing Reflection Direction

- Angle of incidence = angle of reflection
- Normal, light direction and reflection direction are coplaner
- Want all three to be unit length

\[ r = 2(l \cdot n)n - l \]
Plane Normals

• Equation of plane: \( ax + by + cz + d = 0 \)
• We know that a plane is determined by three points \( p_0, p_2, p_3 \) or normal \( \mathbf{n} \) and \( p_0 \)
• Normal can be obtained by

\[
\mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_0) \times (\mathbf{p}_1 - \mathbf{p}_0)
\]
Parametric Form

• For sphere
  
  \[x = x(u,v) = \cos u \sin v\]
  \[y = y(u,v) = \cos u \cos v\]
  \[z = z(u,v) = \sin u\]

• Tangent plane determined by vectors

\[
\frac{\partial p}{\partial u} = [\frac{\partial x}{\partial u}, \frac{\partial y}{\partial u}, \frac{\partial z}{\partial u}]^T
\]
\[
\frac{\partial p}{\partial v} = [\frac{\partial x}{\partial v}, \frac{\partial y}{\partial v}, \frac{\partial z}{\partial v}]^T
\]

• Normal given by cross product

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

• Introduce the OpenGL shading methods
  - per vertex vs per fragment shading
  - Where to carry out

• Discuss polygonal shading
  - Flat
  - Smooth
    • Gouraud shading
    • Phong shading
Shading Principles

• Shading simulates how objects reflect light
  - material composition of object
  - light’s color and position
  - global lighting parameters

• Usually implemented in
  - vertex shader for faster speed
    • Gouraud shading
  - fragment shader for nicer shading
    • Phong shading
OpenGL shading

- Need to specify:
  - Normals
  - Material properties
  - Lights
- Get computed values in application or send attributes to shaders
Surface Normals

- Normals define how a surface reflects light
  - Application usually provides normals as a vertex attribute
  - Current normal is used to compute vertex’s color
  - Use unit normals for proper lighting
    - scaling affects a normal’s length
Normal for Triangle

plane \quad \mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0) = 0

\mathbf{n} = (\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_0)

normalize \quad \mathbf{n} \leftarrow \mathbf{n} / |\mathbf{n}|

Note that right-hand rule determines outward face
Specifying a Point Light Source

• For each light source, we can set an RGBA for the diffuse, specular, and ambient components, and XYZW for the position

\[
\begin{align*}
\text{vec4} & \quad \text{diffuse0} = \text{vec4}(1.0, 0.0, 0.0, 1.0); \\
\text{vec4} & \quad \text{ambient0} = \text{vec4}(1.0, 0.0, 0.0, 1.0); \\
\text{vec4} & \quad \text{specular0} = \text{vec4}(1.0, 0.0, 0.0, 1.0); \\
\text{vec4} & \quad \text{light0\_pos} = \text{vec4}(1.0, 2.0, 3.0, 1.0);
\end{align*}
\]

• 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
Spotlights

• Derive from point source
  - Direction
  - Cutoff
  - Attenuation Proportional to $\cos^\alpha \phi$
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 (1)

- Define the surface properties of a primitive

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diffuse</td>
<td>Base object color</td>
</tr>
<tr>
<td>Specular</td>
<td>Highlight color</td>
</tr>
<tr>
<td>Ambient</td>
<td>Low-light color</td>
</tr>
<tr>
<td>Emission</td>
<td>Glow color</td>
</tr>
<tr>
<td>Shininess</td>
<td>Surface smoothness</td>
</tr>
</tbody>
</table>

- you can have separate materials for front and back
Material Properties (2)

- Material properties should match the terms in the light model.
- Specifies amount of reflected light:
  - An object appears red because it reflects the red component of light.
- $w$ component gives opacity.

```
vec4 ambient = vec4(0.2, 0.2, 0.2, 1.0);
vec4 diffuse = vec4(1.0, 1.0, 1.0, 1.0);
GLfloat shine = 100.0
```
Flat Shading

- Use triangle normal across all fragments of triangle
- One color per triangular facet
Smooth Shading

- Set a new normal at each vertex
- Easy for sphere model
  - If centered at origin $\mathbf{n} = \mathbf{p}$
- Note *silhouette edges*
Gouraud Shading

• Computes a color for each vertex using
  - Surface normals
  - Diffuse and specular reflections
  - Viewer’s position and viewing direction
  - Ambient light
  - Emission

• Vertex colors are interpolated across polygons by the rasterizer
  - *Phong shading* does the same computation per pixel, interpolating the normal across the polygon
    • more accurate results
Polygonal Shading

- In per vertex shading, shading calculations are done for each vertex
  - Vertex colors become vertex shades and can be sent to the vertex shader as a vertex attribute
  - Alternately, we can send the parameters to the vertex shader and have it compute the shade
- By default, vertex shades are interpolated across an object if passed to the fragment shader as a varying variable (smooth shading)
- We can also use uniform variables to shade with a single shade (flat shading)
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 the normals around a mesh vertex.

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

• Gouraud Shading
  - Find average normal at each vertex (vertex normals)
  - Apply Phong illumination equation at each vertex
  - Interpolate vertex shades across each polygon

• Phong shading
  - Find vertex normals
  - Interpolate vertex normals across edges
  - Interpolate edge normals across polygon
  - Apply modified Phong model at each fragment
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
  - Until recently not available in real time systems
  - Now can be done using fragment shaders

• Both need data structures to represent meshes so we can obtain vertex normals
Normalization

- Cosine terms in lighting calculations can be computed using dot product
- Unit length vectors simplify calculation
- Usually we want to set the magnitudes to have unit length but
  - Length can be affected by transformations
  - Note that scaling does not preserve length
- GLSL has a normalization function
Gouraud Shading: Vertex Shader(1)

// Vertex Shader

in vec4 vPosition;
in vec3 vNormal;
out vec4 color; // vertex shade

// light and material properties
uniform vec4 AmbientProduct, DiffuseProduct, SpecularProduct;
uniform mat4 ModelView;
uniform mat4 Projection;
uniform vec4 LightPosition;
uniform float Shininess;
void
main()
{

    // transform vertex position into eye coordinates
    vec3 pos = (ModelView * vPosition).xyz;

    vec3 L = normalize( LightPosition.xyz - pos );
    vec3 E = normalize( -pos );
    vec3 H = normalize( L + E );

    // transform vertex normal into eye coordinates
    vec3 N = normalize( ModelView*vec4(vNormal, 0.0) ).xyz;
}
// compute terms in the illumination equation
// ambient lighting term
vec4 ambient = AmbientProduct;

// diffuse lighting term
float Kd = max(dot(L, N), 0.0);
vec4 diffuse = Kd*DiffuseProduct;

// specular lighting term
float Ks = pow(max(dot(N, H), 0.0), Shininess);
vec4 specular = Ks * SpecularProduct;
if( dot(L, N) < 0.0 ) specular = vec4(0.0, 0.0, 0.0, 1.0);

// add lighting terms to form color
color = ambient + diffuse + specular;
color.a = 1.0;

gl_Position = Projection * ModelView * vPosition;
}
// Fragment Shader
in vec4 color;

void main()
{
    gl_FragColor = color;
}
// Vertex Shader
in vec4 vPosition;
in vec3 vNormal;

// output values that will be interpolated per-fragment
out vec3 fN;
out vec3 fE;
out vec3 fL;

uniform mat4 ModelView;
uniform vec4 LightPosition;
uniform mat4 Projection;
void main()
{
    fN = vNormal;
    fE = vPosition.xyz;
    fL = LightPosition.xyz;

    if( LightPosition.w != 0.0 ) {
        fL = LightPosition.xyz - vPosition.xyz;
    }

    gl_Position = Projection*ModelView*vPosition;
}
Phong Shading: 
Fragment Shader (1)

// Fragment Shader

// per-fragment interpolated values from the vertex shader
in vec3 fN;
in vec3 fL;
in vec3 fE;

uniform vec4 AmbientProduct, DiffuseProduct, SpecularProduct;
uniform mat4 ModelView;
uniform vec4 LightPosition;
uniform float Shininess;

void main()
{
    // normalize the input lighting vectors
    vec3 N = normalize(fN);
    vec3 E = normalize(fE);
    vec3 L = normalize(fL);
}
Phong Shading: Fragment Shader (2)

```glsl
vec3 H = normalize( L + E );
vec4 ambient = AmbientProduct;

float Kd = max(dot(L, N), 0.0);
vec4 diffuse = Kd*DiffuseProduct;

float Ks = pow(max(dot(N, H), 0.0), Shininess);
vec4 specular = Ks*SpecularProduct;

// discard the specular highlight if the light
// is behind the vertex
if( dot(L, N) < 0.0 )
    specular = vec4(0.0, 0.0, 0.0, 1.0);

gl_FragColor = ambient + diffuse + specular;
gl_FragColor.a = 1.0;
```
Fragment Shaders

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

• A shader that’s executed for each “potential” pixel
  - fragments still need to pass several tests before making it to the framebuffer

• There are lots of effects we can do in fragment shaders
  - Per-fragment lighting
  - Texture and bump Mapping
  - Environment (Reflection) Maps
Shader Examples

• **Vertex Shaders**
  - Moving vertices: height fields
  - Per vertex lighting: height fields
  - Per vertex lighting: cartoon shading

• **Fragment Shaders**
  - Per vertex vs. per fragment lighting: cartoon shader
  - Samplers: reflection Map
  - Bump mapping
Height Fields

• A height field is a function \( y = f(x, z) \)
  - \( y \) represents height of point for a location in the \( x-z \) plane

• Heights fields are usually rendered as a rectangular mesh of triangles or rectangles sampled from a grid
  - samples \( y_{ij} = f(x_i, z_j) \)
Displaying a Height Field

• First, generate a mesh data and use it to initialize data for a VBO

```cpp
float dx = 1.0/N, dz = 1.0/N;
for( int i = 0; i < N; i++ ) {
    float x = i*dx;
    for( int j = 0; j < N; j++ ) {
        float z = j*dz;
        float y = f( x, z );

        vertex[Index++] = vec3( x, y, z );
        vertex[Index++] = vec3( x, y, z + dz );
        vertex[Index++] = vec3( x + dx, y, z + dz );
        vertex[Index++] = vec3( x + dx, y, z );
    }
}
```

• Finally, display each quad using

```cpp
for( int i = 0; i < NumVertices ; i += 4 )
glDrawArrays( GL_LINE_LOOP, 4*i, 4 );
```
Time Varying Vertex Shader

```glsl
in vec4 vPosition;
in vec4 vColor;

uniform float time; // in milliseconds
uniform mat4 ModelViewProjectionMatrix;

void main()
{
    vec4 v = vPosition;
    vec4 u = sin( time + 5*v );

    v.y = 0.1 * u.x * u.z;

    gl_Position = ModelViewProjectionMatrix * v;
}
```
Mesh Display
Adding Lighting

• Solid Mesh: create two triangles for each quad
• Display with
  
  ```glDrawArrays( GL_TRIANGLES, 0, NumVertices );```
• For better looking results, add lighting
• We’ll do per-vertex lighting
  - leverage the vertex shader since we’ll also use it to vary the mesh in a time-varying way
uniform float time, shininess;
uniform vec4 vPosition, lightPosition, diffuseLight, specularLight;
uniform mat4 ModelViewMatrix, ModelViewProjectionMatrix, NormalMatrix;

void main()
{
    vec4 v = vPosition;
    vec4 u = sin(time + 5*v);
    v.y = 0.1 * u.x * u.z;

    gl_Position = ModelViewProjectionMatrix * v;

    vec4 diffuse, specular;
    vec4 eyePosition = ModelViewMatrix * vPosition;
    vec4 eyeLightPos = lightPosition;
}
Mesh Shader (2)

```cpp
vec3 N = normalize(NormalMatrix * Normal);
vec3 L = normalize(vec3(eyeLightPos - eyePosition));
vec3 E = -normalize(eyePosition.xyz);
vec3 H = normalize(L + E);

float Kd = max(dot(L, N), 0.0);
float Ks = pow(max(dot(N, H), 0.0), shininess);
diffuse = Kd*diffuseLight;
specular = Ks*specularLight;
color = diffuse + specular;
}
```
Shaded Mesh
Shadows

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

• Introduce Shadow Algorithms
• Expand to projective textures
Flashlight in the Eye Graphics

• When do we not see shadows in a real scene?
• When the only light source is a point source at the eye or center of projection
  - Shadows are behind objects and not visible
• Shadows are a global rendering issue
  - Is a surface visible from a source
  - May be obscured by other objects
Projective Shadows

• Oldest methods
  - Used in flight simulators to provide visual clues

• Projection of a polygon is a polygon called a **shadow polygon**

• Given a point light source and a polygon, the vertices of the shadow polygon are the projections of the original polygon’s vertices from a point source onto a surface
Shadow Polygon
Computing Shadow Vertex

1. Source at \((x_l, y_l, z_l)\)
2. Vertex at \((x, y, z)\)
3. Consider simple case of shadow projected onto ground at \((x_p, 0, z_p)\)
4. Translate source to origin with \(T(-x_l, -y_l, -z_l)\)
5. Perspective projection

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

6. Translate back
Shadow Process

1. Put two identical triangles and their colors on GPU (black for shadow triangle)
2. Compute two model-view matrices as uniforms
3. Send model-view matrix for original triangle
4. Render original triangle
5. Send second model-view matrix
6. Render shadow triangle
   - Note shadow triangle undergoes two transformations
   - Note hidden surface removal takes care of depth issues
Generalized Shadows

• Approach was OK for shadows on a single flat surface
• Note with geometry shader we can have the shader create the second triangle
• Cannot handle shadows on general objects
• Exist a variety of other methods based on same basic idea
• We’ll pursue methods based on projective textures
Image Based Lighting

• We can project a texture onto the surface in which case we are treating the texture as a “slide projector”
• This technique is the basis of projective textures and image based lighting
• Supported in OpenGL and GLSL through four-dimensional texture coordinates
4D Textures Coordinates

- Texture coordinates \((s, t, r, q)\) are affected by a perspective division so the actual coordinates used are \((s/q, t/q, r/q)\) or \((s/q, t/q)\) for a two dimensional texture.
- GLSL has a variant of the function texture \textit{textureProj} which will use the two- or three-dimensional texture coordinate obtained by a perspective division of a 4D texture coordinate a texture value from a sampler:

  \[
  \text{color} = \text{textureProj}(\text{my\_sampler}, \text{tex\_coord})
  \]
Shadow Maps

- If we render a scene from a light source, the depth buffer will contain the distances from the source to each fragment.
- We can store these depths in a texture called a depth map or shadow map.
- Note that although we don’t care about the image in the shadow map, if we render with some light, anything lit is not in shadow.
- Form a shadow map for each source.
Final Rendering

• During the final rendering we compare the distance from the fragment to the light source with the distance in the shadow map.

• If the depth in the shadow map is less than the distance from the fragment to the source, the fragment is in shadow (from this source).

• Otherwise we use rendered color.
Application’s Side

• Start with vertex in object coordinates
• Want to convert representation to texture coordinates
• Form LookAt matrix from light source to origin in object coordinates (MVL)
• From projection matrix for light source (PL)
• From a matrix to convert from \([-1, 1]\) clip coordinates to \([0, 1]\) texture coordinates
• Concatenate to form object to texture coordinate matrix (OTC)
uniform mat4 modelview;
uniform mat4 projection;
uniform normalmatrix; // for diffuse lighting
uniform mat4 otc; // object to texture coordinate
uniform vec4 diffuseproduct; // diffuse light*diffuse reflectivity

in vec4 vPosition;
in vec4 normal;

out vec4 color;
out vec4 shadowCoord;

void main()
{
// compute diffuse color as usual
// using normal, normal matrix, diffuse product
  color = ...

  gl_Position = projection*modelview*vPosition;
  shadowCoord = OTC*vPosition;
}
textureProj function

- Application provides the shadow map as a texture object.
- The GLSL function `textureProj` compares the third value of the texture coordinate with the third value of the texture image.
- For nearest filtering of the texture object, `textureProj` returns 0.0 if the shadow map value is less than the third coordinate and 1.0 otherwise.
- For other filtering options, `textureProj` returns values between 0.0 and 1.0.
uniform sampler2DShadow ShadowMap;

in vec4 shadowCoord;
in vec4 Color;

main()
{
    
    // assume nearest sampling in ShadowMap
    float shadeFactor = textureProj(ShadowMap, ShadowCoord);
    gl_FragColor = vec4(shadeFactor*Color.rgb, Color.a)
}
Texture Mapping

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

• Introduce Mapping Methods
  - Texture Mapping
  - Environment 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 (1)

- 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 known as 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

• Environment (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

generic model  texture mapped
Environment Mapping
Bump Mapping
Texture Mapping

game

geometry

image

screen

x

y

z

s

t
Basic Idea

- Map an image to a surface
- There are 3 or 4 coordinate systems involved
Coordinate Systems

• Parametric coordinates
  - May be used to model curves and surfaces

• Texture coordinates
  - Used to identify points in the image to be mapped

• Object or World Coordinates
  - Conceptually, where the mapping takes place

• Window Coordinates
  - Where the final image is really produced
Texture Mapping

- Parametric coordinates
- Texture coordinates
- World coordinates
- Window coordinates
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
- Simple examples: cylinder, sphere, box
Cylindrical Mapping

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

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 Mapping

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 used in environmental maps
Box Mapping

• Easy to use with simple orthographic projection
• Also used in environment maps
Second Mapping

- Map from intermediate object (e.g., sphere) to actual object
- Three variations:
  - Normals from intermediate to actual
  - Normals from actual to intermediate
  - Vectors from center of intermediate
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
Applying Textures

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 Objects (1)

- Have OpenGL store your images
  - one image per texture object
  - may be shared by several graphics contexts
- Generate texture names

```c
glGenTextures( n, *texIds );
```
Texture Objects (2)

- Create texture objects with texture data and state
  
  ```c
  glBindTexture( target, id );
  ```

- Bind textures before using
  
  ```c
  glBindTexture( target, id );
  ```
Specifying a Texture Image

\texttt{glTexImage2D( target, level, components, w, h, border, format, type, texels );}

- \texttt{target}: type of texture, e.g. \texttt{GL\_TEXTURE\_2D}
- \texttt{level}: used for mipmapping (discussed later)
- \texttt{components}: elements per texel
- \texttt{w, h}: width and height of \texttt{texels} in pixels
- \texttt{border}: used for smoothing (discussed later)
- \texttt{format and type}: describe \texttt{texels}
- \texttt{texels}: pointer to texel array

\texttt{glTexImage2D(GL\_TEXTURE\_2D, 0, 3, 512, 512, 0, GL\_RGB, GL\_UNSIGNED\_BYTE, my\_texels);}
Mapping a Texture

- Based on parametric texture coordinates
- Coordinates need to be specified at each vertex

\[(s, t) = (0.2, 0.8)\]
Applying Texture to Cube

// add texture coordinate attribute to quad function
quad( int a, int b, int c, int d )
{
    vColors[Index] = colors[a];
    vPositions[Index] = positions[a];
    vTexCoords[Index] = vec2( 0.0, 0.0 );
    Index++;

    vColors[Index] = colors[b];
    vPositions[Index] = positions[b];
    vTexCoords[Index] = vec2( 1.0, 0.0 );
    Index++;

    ... // rest of vertices
}
// Create a checkerboard pattern
for ( int i = 0; i < 64; i++ ) {
    for ( int j = 0; j < 64; j++ ) {
        GLubyte c;
        c = ((i & 0x8 == 0) ^ (j & 0x8 == 0)) * 255;
        image[i][j][0]  = c;
        image[i][j][1]  = c;
        image[i][j][2]  = c;
    }
}

Texture Object

GLuint textures[1];
glGenTextures( 1, textures );

glActiveTexture( GL_TEXTURE0 );
glBindTexture( GL_TEXTURE_2D, textures[0] );

glTexImage2D( GL_TEXTURE_2D, 0, GL_RGB, TextureSize, TextureSize, GL_RGB, GL_UNSIGNED_BYTE, image );

glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST );
offset = 0;
GLuint vPosition = glGetUniformLocation( program, "vPosition" );
glEnableVertexAttribArray( vPosition );
glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE, 0,(void*) offset );

offset += sizeof(points);
GLuint vTexCoord = glGetUniformLocation( program, "vTexCoord" );
glEnableVertexAttribArray( vTexCoord );
glVertexAttribPointer( vTexCoord, 2,GL_FLOAT, GL_FALSE, 0,(void*) offset );
Vertex Shader

in vec4 vPosition;
in vec4 vColor;
in vec2 vTexCoord;

out vec4 color;
out vec2 texCoord;

void main()
{
    color       = vColor;
    texCoord    = vTexCoord;
    gl_Position = vPosition;
}
in vec4 color;
in vec2 texCoord;
out vec4 fColor;
uniform sampler texture;

void main()
{
    fColor = color * texture( texture, texCoord);
}
Fragment Shader for Modulating Intensity with Texture

in vec4 texCoord;

// Declare the sampler
uniform float intensity;
uniform sampler2D diffuseMaterialTexture;

// Apply the material color
vec3 diffuse = intensity * texture(diffuseMaterialTexture, texCoord).rgb;
Interpolation

OpenGL uses interpolation (in rasterizer) 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 has a variety of parameters 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

\[
\begin{align*}
\text{glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP)} \\
\text{glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT)}
\end{align*}
\]

t

\[
\begin{align*}
texture & \quad \text{GL\_REPEAT wrapping} & \quad \text{GL\_CLAMP wrapping}
\end{align*}
\]
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
Filter Modes

Modes determined by

- `glTexParameteri(target, type, mode)`

```c
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

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

point sampling

mipmapped point sampling

linear filtering

mipmapped linear filtering
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`
Using Texture Objects

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

• Environment 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
  - Alternative is to use a cube map

• Multitexturing
  - Apply a sequence of textures through cascaded texture units
Applying Textures

• Textures are applied during fragments shading by a sampler
• Samplers return a texture color from a texture object

in vec4 color; //color from rasterizer
in vec2 texCoord; //texture coordinate from rasterizer
uniform sampler2D texture; //texture object from application

void main() {
    gl_FragColor = color * texture2D( texture, texCoord );
}
Vertex Shader

- Usually vertex shader will output texture coordinates to be rasterized
- Must do all other standard tasks too
  - Compute vertex position
  - Compute vertex color if needed

```glsl
in vec4 vPosition; // vertex position in object coordinates
in vec4 vColor;    // vertex color from application
in vec2 vTexCoord; // texture coordinate from application

out vec4 color;     // output color to be interpolated
out vec2 texCoord;  // output tex coordinate to be interpolated
```
GLubyte image[64][64][3];

// Create a 64 x 64 checkerboard pattern
for ( int i = 0; i < 64; i++ ) {
    for ( int j = 0; j < 64; j++ ) {
        GLubyte c = (((i & 0x8)==0)^((j & 0x8)==0)) * 255;
        image[i][j][0]  = c;
        image[i][j][1]  = c;
        image[i][j][2]  = c;
    }
}
Adding Texture Coordinates

```c
void quad( int a, int b, int c, int d )
{
    quad_colors[Index] = colors[a];
    points[Index] = vertices[a];
    tex_coords[Index] = vec2( 0.0, 0.0 );
    Index++;
    quad_colors[Index] = colors[a];
    points[Index] = vertices[b];
    tex_coords[Index] = vec2( 0.0, 1.0 );
    Index++;

    // other vertices
}
```
GLuint textures[1];
glGenTextures( 1, textures );

glBindTexture( GL_TEXTURE_2D, textures[0] );
glTexImage2D( GL_TEXTURE_2D, 0, GL_RGB, TextureSize,
             TextureSize, 0, GL_RGB, GL_UNSIGNED_BYTE, image );
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                 GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                 GL_NEAREST);
glActiveTexture(GL_TEXTURE0);
Linking with Shaders

GLuint vTexCoord = glGetUniformLocation( program, "vTexCoord" );
glEnableVertexAttribArray( vTexCoord );
glVertexAttribPointer( vTexCoord, 2, GL_FLOAT, GL_FALSE, 0,
    BUFFER_OFFSET(offset) );

// Set the value of the fragment shader texture sampler variable
// ("texture") to the the appropriate texture unit. In this case,
// zero, for GL_TEXTURE0 which was previously set by calling
// glActiveTexture().
glUniform1i( glGetUniformLocation(program, "texture"), 0 );
Curves and Surfaces

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

• Introduce types of curves and surfaces
  - Explicit
  - Implicit
  - Parametric
  - Strengths and weaknesses

• Discuss Modeling and Approximations
  - Conditions
  - Stability
Escaping Flatland

• Until now we have worked with flat entities such as lines and flat polygons
  - Fit well with graphics hardware
  - Mathematically simple

• But the world is not composed of flat entities
  - Need curves and curved surfaces
  - May only have need at the application level
  - Implementation can render them approximately with flat primitives
Modeling with Curves

- data points
- approximating curve
- interpolating data point
What Makes a Good Representation?

• There are many ways to represent curves and surfaces

• Want a representation that is
  - Stable
  - Smooth
  - Easy to evaluate
  - Must we interpolate or can we just come close to data?
  - Do we need derivatives?
Representation of Curves & Surfaces

• Three types of object representation:
  - explicit: \( y = f(x) \).
  - implicit: \( f(x, y) = 0 \).
  - parametric: \( \mathbf{p}(u) = [x(u) \ y(u) \ z(u)]^T \).
Explicit Representation

• Most familiar form of curve in 2D
  \[ y = f(x) \]

• Cannot represent all curves
  - Vertical lines
  - Circles

• Extension to 3D
  - \[ y = f(x), \quad z = g(x) \]
  - The form \[ z = f(x,y) \] defines a surface
Explicit Representation of Lines

- The explicit form of a curve in 2D gives the value of one dependent variable in terms of the other independent variable.

\[
y = f(x).
\]

- An explicit form may or may not exist. We write

\[
y = mx + h
\]

for the line even though the equation does not hold for vertical lines.
Explicit Representation of Circles

• A circle has constant \textit{curvature}.
  - An explicit form exists only for half of the curve:
    \[ y = \sqrt{r^2 - x^2}. \]
  - The other half requires a second equation:
    \[ y = -\sqrt{r^2 - x^2}. \]
  - In addition, we must restrict the range of \textit{x}.
    • \textit{f} is a function, so there must be exactly one value of \textit{y} for every \textit{x}. 
Explicit Surfaces

• A surface requires two independent variables and two equations:

\[ y = ax + b, \]
\[ z = cx + d. \]

- The line cannot be in a plane of constant \( x \).
- We cannot represent a sphere with only one equation of the form

\[ z = f(x, y). \]
Implicit Representation

• An implicit curve has the form
  \[ f(x, y) = 0. \]

• Much more robust
  - A line: \( ax + by + c = 0. \)
  - A circle: \( x^2 + y^2 - r^2 = 0. \)

• Implicit functions test membership.
  - Does the point \((x, y)\) lie on the curve determined by \(f\)?

• In general, there is no analytic way to find the \(y\) value for a given \(x\).
Implicit Surfaces

• In three dimensions, a surface is described by the implicit form

\[ f(x, y, z) = 0. \]

  - A plane: \( ax + by + cz + d = 0. \)
  - A sphere: \( x^2 + y^2 + z^2 - r^2 = 0. \)

• Intersect two 3D surfaces to get a 3D curve.
• Implicit curve representations are difficult to use in 3D.
Algebraic Surfaces

• One class of useful implicit surfaces is the **quadric** surface.
  - **Algebraic surfaces** are those for which the function \( f(x, y, z) \) is the sum of polynomials.
    \[
    \sum \sum \sum x^i y^j z^k = 0
    \]
  - Quadric surfaces contain polynomials that have degree at most two: \( 2 \geq i+j+k \)
    This yields at most 10 terms
Parametric Form

• Expresses the value of each spatial component in terms of an independent variable $u$, the parameter:

$$x = x(u), \quad y = y(u), \quad z = z(u).$$

- 3 explicit functions, 1 independent variable.
- Same form in 2D and 3D.

• The most flexible and robust form for computer graphics.
Parametric Form of Line

- 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:
    \[
    \begin{align*}
    x(\alpha) &= \alpha x_0 + (1-\alpha)x_1 \\
    y(\alpha) &= \alpha y_0 + (1-\alpha)y_1
    \end{align*}
    \]
Parametric Curves

- Separate equation for each spatial variable
  \[x = x(u)\]
  \[y = y(u)\]
  \[z = z(u)\]

- The parametric form describes the locus of points being drawn as \(u\) varies:
  \[u_{\text{min}} \leq u \leq u_{\text{max}}\]

Matrix notation:
\[\mathbf{p}(u) = [x(u), y(u), z(u)]^T\]
Derivative of the Curve

- The derivative is the velocity with which the curve is traced out:

\[
\frac{dp(u)}{du} = \begin{bmatrix}
\frac{dx(u)}{du} \\
\frac{dy(u)}{du} \\
\frac{dz(u)}{du}
\end{bmatrix}.
\]

- It points in the direction tangent to the curve.
Parametric Lines

We can normalize $u$ to be over the interval $(0,1)$

Line connecting two points $p_0$ and $p_1$

$$p(u) = (1-u)p_0 + up_1$$

Ray from $p_0$ in the direction $d$

$$p(u) = p_0 + ud$$
Parametric Surfaces

• Surfaces require 2 parameters
  \[ x = x(u,v) \]
  \[ y = y(u,v) \]
  \[ z = z(u,v) \]
  \[ \mathbf{p}(u,v) = [x(u,v), y(u,v), z(u,v)]^T \]

• Want same properties as curves:
  - Smoothness
  - Differentiability
  - Ease of evaluation
Normals

We can differentiate with respect to $u$ and $v$ to obtain the normal at any point $p$

$$\frac{\partial p(u, v)}{\partial u} = \begin{bmatrix} \frac{\partial x(u, v)}{\partial u} \\ \frac{\partial y(u, v)}{\partial u} \\ \frac{\partial z(u, v)}{\partial u} \end{bmatrix}$$

$$\frac{\partial p(u, v)}{\partial v} = \begin{bmatrix} \frac{\partial x(u, v)}{\partial v} \\ \frac{\partial y(u, v)}{\partial v} \\ \frac{\partial z(u, v)}{\partial v} \end{bmatrix}$$

$$n = \frac{\partial p(u, v)}{\partial u} \times \frac{\partial p(u, v)}{\partial v}$$
Parametric Planes

Point-Vector form

\[
p(u,v) = p_0 + uq + vr
\]

\[
n = q \times r
\]

Three-point form

\[
q = p_1 - p_0
\]

\[
r = p_2 - p_0
\]
Parametric Sphere

\[
x(u,v) = r \cos q \sin f \\
y(u,v) = r \sin q \sin f \\
z(u,v) = r \cos f
\]

\[360 \geq q \geq 0\]
\[180 \geq f \geq 0\]

\(\theta\) constant: circles of constant longitude
\(f\) constant: circles of constant latitude

differentiate to show \(\mathbf{n} = \mathbf{p}\)
Curve Segments

• After normalizing $u$, each curve is written
  \[ p(u) = [x(u), y(u), z(u)]^T, \quad 1 \geq u \geq 0 \]
• In classical numerical methods, we design a single global curve
• In computer graphics and CAD, it is better to design small connected curve segments

![Diagram of curve segments with points $p(0)$, $p(u)$, $q(u)$, and $q(1)$ joined at $p(1) = q(0)$]
Parametric Polynomial Curves

\[
x(u) = \sum_{i=0}^{N} c_{xi} u^i \quad y(u) = \sum_{j=0}^{M} c_{yj} u^j \quad z(u) = \sum_{k=0}^{L} c_{zk} u^k
\]

- If N=M=K, we need to determine 3(N+1) coefficients.
- Equivalently we need 3(N+1) independent conditions.
- Noting that the curves for x, y and z are independent, we can define each independently in an identical manner.
- We will use the form where p can be any of x, y, z.

\[
p(u) = \sum_{k=0}^{L} c_k u^k
\]
Why Polynomials

- Easy to evaluate
- Continuous and differentiable everywhere
  - Must worry about continuity at join points including continuity of derivatives

\[ p(u) \]

\[ q(u) \]

Join point \( p(1) = q(0) \)
But \( p'(1) \neq q'(0) \)
Cubic Parametric Polynomials

- \( N=M=L=3 \), gives balance between ease of evaluation and flexibility in design

\[
p(u) = \sum_{k=0}^{3} c_k u^k
\]

- Four coefficients to determine for each of \( x, y \) and \( z \)
- Seek four independent conditions for various values of \( u \) resulting in 4 equations in 4 unknowns for each of \( x, y \) and \( z \)
  - Conditions are a mixture of continuity requirements at the join points and conditions for fitting the data
Cubic Polynomial Surfaces

\[ p(u,v) = [x(u,v), y(u,v), z(u,v)]^T \]

where

\[ p(u, v) = \sum_{i=0}^{3} \sum_{j=0}^{3} c_{ij} u^i v^j \]

\( p \) is any of \( x, y \) or \( z \)

Need 48 coefficients (3 independent sets of 16) to determine a surface patch
Parametric Polynomial Surfaces

In general,

\[ p(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} c_{ij} u^i v^j \]

• A surface patch:
  - Specify \(3(n+1)(m+1)\) coefficients.
  - Let \(n = m\), and let \(u\) and \(v\) vary over the rectangle \(0 \leq u, v \leq 1\).
Designing Parametric Cubic Curves

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

• Introduce the types of curves
  - Interpolating
  - Hermite
  - Bezier
  - B-Spline

• Analyze their performance
Design Criteria

• Why we prefer parametric polynomials of low degree:
  - Local control of shape,
  - Smoothness and continuity,
  - Ability to evaluate derivatives,
  - Stability,
  - Ease of rendering.
Smoothness

- Smoothness guaranteed because our polynomial equations are differentiable.
- Difficulties arise at the join points.
Control Points

• We prefer local control for stability.
  - The most common interface is a group of **control points**.
    - In this example, the curve passes through, or **interpolates**, some of the control points, but only comes close to, or **approximates**, others.
Parametric Cubic Polynomial Curves

• Choosing the degree:
  - High degree allows many control points, but computation is expensive.
  - Low degree may mean low level of control.

• The compromise: use low-degree curves over short intervals.
  - Most designers work with cubic polynomial curves.
Matrix Notation

\[ p(u) = \sum_{k=0}^{3} c_k u^k = u^T c, \]

where

\[ c = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}, \quad u = \begin{bmatrix} 1 \\ u \\ u^2 \\ u^3 \end{bmatrix}, \quad c_k = \begin{bmatrix} c_{kx} \\ c_{ky} \\ c_{kz} \end{bmatrix}. \]
Interpolation

• An interpolating polynomial passes through its control points.

  - Suppose we have four controls points
    \[ \mathbf{p}_k = \begin{bmatrix} x_k \\ y_k \\ z_k \end{bmatrix}, \text{ for } 0 \leq k \leq 3. \]

  - We let \( u \) vary over the interval \([0,1]\), giving us four equally spaced values: 0, 1/3, 2/3, 1.
Evaluating the Control Points

• We seek coefficients $c_0, c_1, c_2, c_3$ satisfying the four conditions:

\[
\begin{align*}
p_0 &= p(0) = c_0, \\
p_1 &= p(1/3) = c_0 + \frac{1}{3}c_1 + \left(\frac{1}{3}\right)^2c_2 + \left(\frac{1}{3}\right)^3c_3, \\
p_2 &= p(2/3) = c_0 + \frac{2}{3}c_1 + \left(\frac{2}{3}\right)^2c_2 + \left(\frac{2}{3}\right)^3c_3, \\
p_3 &= p(1) = c_0 + c_1 + c_2 + c_3.
\end{align*}
\]
Matrix Notation

• In matrix notation \( \mathbf{p} = \mathbf{A}\mathbf{c} \), where

\[
\mathbf{p} = \begin{bmatrix}
\mathbf{p}_0 \\
\mathbf{p}_1 \\
\mathbf{p}_2 \\
\mathbf{p}_3
\end{bmatrix}
\quad \text{and} \quad
\mathbf{A} = \begin{bmatrix}
1 & 0 & 0 & 0 \\
1 & 1/3 & (1/3)^2 & (1/3)^3 \\
1 & 2/3 & (2/3)^2 & (2/3)^3 \\
1 & 1 & 1 & 1
\end{bmatrix}.
\]

\( \mathbf{p} \) is a column vector of row vectors, and \( \mathbf{A} \) is nonsingular: we will use its inverse.
Interpolating Geometry Matrix

\[ M_I = A^{-1} = \begin{bmatrix}
1 & 0 & 0 & 0 \\
-5.5 & 9 & -4.5 & 1 \\
9 & -22.5 & 18 & -4.5 \\
-4.5 & 13.5 & -13.5 & 4.5 \\
\end{bmatrix} \]

- The desired coefficients are

\[ c = M_I p. \]
Interpolating Multiple Segments

• Use the last control point of one segment as the first control point of the next segment.

- To achieve smoothness in addition to continuity, we will need additional constraints on the derivatives.
Blending Functions

• Substituting the interpolating coefficients into our polynomial:

\[ p(u) = u^T c = u^T M_I p. \]

• Let

\[ p(u) = b(u)^T p, \text{ where } b(u) = M_I^T u. \]

• The \( b(u) \) are the blending polynomials.
Visualizing the Curve Using Blending Functions

- The effect on the curve of an individual control point is easier to see by studying its blending function.
The Cubic Interpolating Patch

• A bicubic surface patch:

\[ p(u, v) = \sum_{i=0}^{3} \sum_{j=0}^{3} u^i v^j c_{ij}. \]
Matrix Notation

• In matrix form, the patch is defined by

\[ p(u, v) = u^T C v, \]
- The column vector \( v = [1 \ v \ v^2 \ v^3]^T. \)
- \( C \) is a 4 x 4 matrix of column vectors.

• 16 equations in 16 unknowns.
By setting \( v = 0, 1/3, 2/3, 1 \) we can sample the surface using curves in \( u \):

\[
\mathbf{u}^T \mathbf{M}_I \mathbf{P} = \mathbf{u}^T \mathbf{C} \mathbf{A}^T.
\]

- The coefficient matrix \( \mathbf{C} \) is computed by

\[
\mathbf{C} = \mathbf{M}_I \mathbf{P} \mathbf{M}_I^T.
\]

- The equation for the surface becomes

\[
\mathbf{p}(\mathbf{u}, \mathbf{v}) = \mathbf{u}^T \mathbf{M}_I \mathbf{P} \mathbf{M}_I^T \mathbf{v}.
\]
Blending Patches

• Extending our use of blending polynomials to surfaces:

\[ p(u, v) = \sum_{i=0}^{3} \sum_{j=0}^{3} b_i(u)b_j(v)p_{ij}. \]

- 16 simple patches form a surface.
- Also known as tensor-product surfaces.
- These surfaces are not very smooth.
  • But they are separable, meaning they allow us to work with functions in \( u \) and \( v \) independently.
Other Types of Curves and Surfaces

• How can we get around the limitations of the interpolating form
  - Lack of smoothness
  - Discontinuous derivatives at join points

• We have four conditions (for cubics) that we can apply to each segment
  - Use them other than for interpolation
  - Need only come close to the data
Hermite Form

Use two interpolating conditions and two derivative conditions per segment.

Ensures continuity and first derivative continuity between segments.
Hermite Curves and Surfaces

- Use the data at control points differently in an attempt to get smoother results.
  - We insist that the curve interpolate the control points only at the two ends, $p_0$ and $p_3$.

$$p(0) = p_0 = c_0,$$
$$p(1) = p_3 = c_0 + c_1 + c_2 + c_3.$$
Additional Conditions

• The derivative is a quadratic polynomial:

\[ p'(u) = \begin{bmatrix} \frac{dx}{du} \\ \frac{dy}{du} \\ \frac{dz}{du} \end{bmatrix} = c_1 + 2uc_2 + 3u^2c_3. \]

- We now can derive two additional conditions:

\[ p'_0 = p'(0) = c_1, \]
\[ p'_3 = p'(1) = c_1 + 2c_2 + 3c_3. \]
Matrix Form

\[
\begin{bmatrix}
  p_0 \\
  p_3 \\
  p'_0 \\
  p'_3 \\
\end{bmatrix}
= \begin{bmatrix}
  1 & 0 & 0 & 0 \\
  1 & 1 & 1 & 1 \\
  0 & 1 & 0 & 0 \\
  0 & 1 & 2 & 2 \\
\end{bmatrix}
\begin{bmatrix}
  c \\
\end{bmatrix}
\]

• The desired coefficient matrix is

\[
c = M_H q.
\]

- \(M_H\) is the Hermite geometry matrix.
The Hermite Geometry Matrix

\[
M_H = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
-3 & 3 & -2 & -1 & 1 \\
2 & -2 & 1 & 1 & 1
\end{bmatrix}.
\]

• The resulting polynomial is

\[
p(u) = u^T M_H q.
\]
Blending polynomials

- Using blending functions $p(u) = b(u)^T q$,

$$b(u) = M_H^T u = \begin{bmatrix} 2u^3 - 3u^2 + 1 \\ -2u^3 + 3u^2 \\ u^3 - 2u^2 + u \\ u^3 - u^2 \end{bmatrix}.$$

- Although these functions are smooth, the Hermite form is not used directly in Computer Graphics and CAD because we usually have control points but not derivatives.
- However, the Hermite form is the basis of the Bezier form.
Parametric and Geometric Continuity

- We can require the derivatives of $x$, $y$, and $z$ to each be continuous at join points (parametric continuity)
- Alternately, we can only require that the tangents of the resulting curve be continuous (geometry continuity)
- The latter gives more flexibility as we have need satisfy only two conditions rather than three at each join point
Example

• Here the $p$ and $q$ have the same tangents at the ends of the segment but different derivatives
• Generate different Hermite curves
• This techniques is used in drawing applications
Parametric Continuity

- Continuity is enforced by matching polynomials at join points.

- $C^0$ parametric continuity:

\[
p(1) = \begin{bmatrix} p_x(1) \\ p_y(1) \\ p_z(1) \end{bmatrix} = q(0) = \begin{bmatrix} q_x(0) \\ q_y(0) \\ q_z(0) \end{bmatrix}.
\]
C\(^1\) Parametric Continuity

• Matching derivatives at the join points gives us \(C^1\) continuity:

\[
\mathbf{p}'(1) = \begin{bmatrix} p'_x(1) \\ p'_y(1) \\ p'_z(1) \end{bmatrix} = \mathbf{q}'(0) = \begin{bmatrix} q'_x(0) \\ q'_y(0) \\ q'_z(0) \end{bmatrix}.
\]
Another Approach: Geometric Continuity

• If the derivatives are proportional, then we have geometric continuity.

- One extra degree of freedom.
- Extends to higher dimensions.
Beziers Curves: Basic Idea

• In graphics and CAD, we usually don’t have derivative data
• Bezier suggested using the same 4 data points as with the cubic interpolating curve to approximate the derivatives in the Hermite form
Beziers Curves and Surfaces

- Bezier added control points to manipulate derivatives.

- The two derivative conditions become

\[ 3p_1 - 3p_0 = c_1, \]
\[ 3p_3 - 3p_2 = c_1 + 2c_2 + 3c_3. \]
Bezisr Geometry Matrix

• We solve \( \mathbf{c} = \mathbf{M}_B \mathbf{p} \), where

\[
\mathbf{M}_B = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
-3 & 3 & 0 & 0 & 0 \\
3 & -6 & 3 & 0 & 0 \\
-1 & 3 & 3 & 1 & 0
\end{bmatrix}.
\]

• The cubic Bezier polynomial is thus

\[
\mathbf{p}(u) = \mathbf{u}^T \mathbf{M}_B \mathbf{p}.
\]
Beziers Blending Functions

- These functions are Bernstein polynomials:

\[ b_{kd}(u) = \frac{d!}{k!(d-k)!} u^k (1-u)^{d-k}. \]
Properties of Bernstein Polynomials

- All zeros are either at $u = 0$ or $u = 1$.
  - Therefore, the curve must be smooth over (0,1)
- The value of $u$ never exceeds 1.
  - $p(u)$ is a convex sum, so the curve lies inside the convex hull of the control points.
Bezier Surface Patches

• Using a 4 x 4 array of control points \( P \),

\[
p(u, v) = \sum_{i=0}^{3} \sum_{j=0}^{3} b_i(u)b_j(u)p_{ij}
= u^T M_B P M_B^T v.
\]
Convex Hull Property in 3D

• The patch is inside the convex hull of the control points and interpolates the four corner points $p_{00}, p_{03}, p_{30}, p_{33}$. 
BeziersPatch Edges

• Partial derivatives in the $u$ and $v$ directions treat the edges of the patch as 1D curves.

\[
\hat{\frac{\partial \mathbf{p}}{\partial u}}(0,0) = 3(\mathbf{p}_{10} - \mathbf{p}_{00}),
\]

\[
\hat{\frac{\partial \mathbf{p}}{\partial v}}(0,0) = 3(\mathbf{p}_{01} - \mathbf{p}_{00}).
\]
Beziers Patch Corners

• The **twist** vector draws the center of the patch away from the plane.

\[
\frac{\partial^2 p}{\partial u \partial v}(0,0) = 9(p_{00} - p_{01} + p_{10} - p_{11}).
\]
Cubic B-Splines

• Bezier curves and surfaces are widely used.
  - One limitation: $C^0$ continuity at the join points.

• **B-Splines** are not required to interpolate any control points.
  - Relaxing this requirement makes it possible to enforce greater smoothness at join points.
The Cubic B-Spline Curve

• The control points now reside in the middle of a sequence:

$$\{p_{i-2}, p_{i-1}, p_i, p_{i+1}\}.$$

- The curve spans only the distance between the middle two control points.
Formulating the Geometry Matrix

• We are looking for a polynomial

\[ p(u) = u^T M p, \]

where \( p \) is the matrix of control points.

- \( M \) can be made to enforce a number of conditions.
- In particular, we can impose continuity requirements at the join points.
Join Point Continuity

• Construct \( q \) from the same matrix as \( p \):

\[
p = \begin{bmatrix}
p_{i-2} \\
p_{i-1} \\
p_i \\
p_{i+1}
\end{bmatrix}
\]

and

\[
q = \begin{bmatrix}
p_{i-3} \\
p_{i-2} \\
p_{i-1} \\
p_i
\end{bmatrix}.
\]

- Now let \( q(u) = u^T M q \).
- Constraints on derivates allow us to control smoothness.
Symmetric Approximations

- Enforcing symmetry at the join points is a popular choice for $M$.
- Two conditions that satisfy symmetry are:

$$p(0) = q(1) = \frac{1}{6}(p_{i-2} + 4p_{i-1} + p_i),$$

$$p'(0) = q'(1) = \frac{1}{2}(p_i - p_{i-2}).$$
Additional Conditions

• We apply the same symmetry conditions to $p(1)$, the other endpoint.
  - We now have four equations in the four unknowns $c_0, c_1, c_2, c_3$:

$$p(u) = u^T c.$$
The B-Spline Geometry Matrix

Once we have the coefficient matrix, we can solve for the geometry matrix:

\[
M_s = \frac{1}{6} \begin{bmatrix}
1 & 4 & 1 & 0 \\
-3 & 0 & 3 & 0 \\
3 & -6 & 3 & 0 \\
-1 & 3 & -3 & 1
\end{bmatrix}.
\]
B-Spline Blending Functions

• The blending functions are

\[
\begin{bmatrix}
\frac{1}{6} \\
4 - 6u^2 + 3u^3 \\
1 + 3u + 3u^2 - 3u^3 \\
u^3
\end{bmatrix}
\]
Advantages of B-spline Curves

• In sequence, B-spline curve segments have $C^2$ continuity at the join points.
  - They are also confined to their convex hulls.

• On the other hand, we need more control points than we did for Bezier curves.
B-Splines and Bases

- Each control point affects four adjacent intervals.

\[ B_i(u) = \begin{cases} 
0 & u < i - 2, \\
b_0(u + 2) & i - 2 \leq u < i - 1, \\
b_1(u + 1) & i - 1 \leq u < i, \\
b_2(u) & i \leq u < i + 1, \\
b_3(u - 1) & i + 1 \leq u < i + 2, \\
0 & u \geq i + 2. 
\]
Spline Basis Function

- A single expression for the spline curve using basis functions:

\[ p(u) = \sum_{i=1}^{m-1} B_i(u) p_i. \]
Approximating Splines

• Each $B_i$ is a shifted version of a single function.
  - Linear combinations of the $B_i$ form a piecewise polynomial curve over the whole interval.
Spline Surfaces

• The same form as Bezier surfaces:

\[ p(u, v) = \sum_{i=0}^{3} \sum_{j=0}^{3} b_i(u) b_j(v) p_{ij}. \]

- But one segment per patch, instead of nine!

- However, they are also much smoother.
General B-Splines

- Polynomials of degree $d$ between $n$ knots $u_0,...,u_n$:

\[
p(u) = \sum_{j=0}^{d} c_{jk} u^j, \quad u_k < u < u_{k+1}
\]

- If $d = 3$, then each interval contains a cubic polynomial: $4n$ equations in $4n$ unknowns.
- A global solution that is not well-suited to computer graphics.
The Cox-deBoor Recursion

• A particular set of basis splines is defined by the Cox-deBoor recursion:

\[ B_{k0} = \begin{cases} 1 & u_k \leq u \leq u_{k+1}, \\ 0 & \text{otherwise}; \end{cases} \]

\[ B_{kd} = \frac{u - u_k}{u_{k+d} - u_k} B_{k,d-1}(u) + \frac{u_{k+d} - u}{u_{k+d+1} - u_{k+1}} B_{k+1,d-1}(u). \]
Recursively Defined B-Splines

- Linear interpolation of polynomials of degree $k$ produces polynomials of degree $k + 1$. 
Uniform Splines

- Equally spaced knots.
Nonuniform B-Splines

• Repeated knots pull the spline closer to the control point.
  - Open splines extend the curve by repeating the endpoints.
  - Knot sequences:
    \{0,0,0,0,1,2,\ldots,n-1,n,n,n,n\} \quad \text{often used}
    \{0,0,0,0,1,1,1,1,1\}. \quad \text{cubic Bezier curve}
  - Any spacing between the knots is allowed in the general case.
NURBS

• Use weights to increase or decrease the importance of a particular point.
  - The weighted homogeneous-coordinate representation of a control point \( p_i = [x_i, y_i, z_i] \) is

\[
q_i = w_i \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix}.
\]
The NURBS Basis Functions

- A 4D B-spline

\[
q(u) = \begin{bmatrix} x(u) \\ y(u) \\ z(u) \end{bmatrix} = \sum_{i=0}^{n} B_{i,d}(u)w_i p_i.
\]

- Derive the \( w \) component from the weights:

\[
w(u) = \sum_{i=0}^{n} B_{i,d}(u)w_i.
\]
Nonuniform Rational B-Splines

• Each component of \( p(u) \) is a rational function in \( u \).
  - We use perspective division to recover the 3D points:

\[
p(u) = \frac{1}{w(u)} q(u) = \sum_{i=0}^{n} \frac{B_{i,d}(u) w_i p_i}{\sum_{i=0}^{n} B_{i,d}(u) w_i}.
\]

- These curves are invariant under perspective transformations.
- They can approximate quadrics—one representation for all types of curves.
Rendering Curves and Surfaces

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

• Introduce methods to draw curves
  - Approximate with lines
  - Finite Differences
• Derive the recursive method for evaluation of Bezier curves and surfaces
• Learn how to convert all polynomial data to data for Bezier polynomials
Evaluating Polynomials

- Simplest method to render a polynomial curve is to evaluate the polynomial at many points and form an approximating polyline.
- For surfaces we can form an approximating mesh of triangles or quadrilaterals.
- Use Horner’s method to evaluate polynomials

\[ p(u) = c_0 + u(c_1 + u(c_2 + uc_3)) \]

- 3 multiplications/evaluation for cubic
Polynomial Evaluation Methods

- Our standard representation:

\[ p(u) = \sum_{i=0}^{n} c_i u^i, \quad 0 \leq u \leq 1 \]

- Horner's method:

\[ p(u) = c_0 + u(c_1 + u(c_2 + u(\ldots + c_n u))). \]

- If the points \{u_i\} are spaced uniformly, we can use the method of **forward differences**.
The Method of Forward Differences

- Forward differences defined iteratively:
  \[ \Delta^{(0)} p(u_k) = p(u_k), \]
  \[ \Delta^{(1)} p(u_k) = p(u_{k+1}) - p(u_k), \]
  \[ \Delta^{(m+1)} p(u_k) = \Delta^{(m)} p(u_{k+1}) - \Delta^{(m)} p(u_k). \]

- If \( u_{k+1} - u_k = h \) is constant, then \( \Delta^{(n)} p(u_k) \) is constant for all \( k \).
Computing The Forward-Difference Table

• For the cubic polynomial

\[ p(u) = 1 + 3u + 2u^2 + u^3, \]

we construct the table as follows:

<table>
<thead>
<tr>
<th>( t )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p )</td>
<td>1</td>
<td>7</td>
<td>23</td>
<td>55</td>
<td>109</td>
<td>191</td>
</tr>
<tr>
<td>( D^{(1)}_p )</td>
<td>6</td>
<td>16</td>
<td>32</td>
<td>54</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td>( D^{(2)}_p )</td>
<td>10</td>
<td>16</td>
<td>22</td>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( D^{(3)}_p )</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

compute these
Using the Table

- Compute successive values of $p(u_k)$ starting from the bottom:

<table>
<thead>
<tr>
<th>$t$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>1</td>
<td>7</td>
<td>23</td>
<td>55</td>
<td>109</td>
<td>191</td>
</tr>
<tr>
<td>$D^{(1)}_p$</td>
<td>6</td>
<td>16</td>
<td>32</td>
<td>54</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td>$D^{(2)}_p$</td>
<td>10</td>
<td>16</td>
<td>22</td>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$D^{(3)}_p$</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$$
\Delta^{(m-1)}(p_{k+1}) = \Delta^{(m)}p(u_k) + \Delta^{(m-1)}p(u_k).
$$
Subdivision Curves and Surfaces

• A process of iterative refinement that produces smooth curves and surfaces.
Recursive Subdivision of Bezier Polynomials: deCasteljau Algorithm

1. Break the curve into two separate polynomials, \( I(u) \) and \( R(u) \).

- The convex hulls for \( I \) and \( R \) must lie inside the convex hull for \( p \): the variation-diminishing property:
Efficient Computation of the Subdivision

\[ l_0 = p_0, \]
\[ l_1 = \frac{1}{2} (p_0 + p_1), \]
\[ l_2 = \frac{1}{2} \left( l_1 + \frac{1}{2} (p_1 + p_2) \right), \]
\[ l_3 = r_0 = \frac{1}{2} (l_2 + r_1). \]

Requires only shifts and adds!
Every Curve is a Bezier Curve

• We can render a given polynomial using the recursive method if we find control points for its representation as a Bezier curve

• Suppose that \( p(u) \) is given as an interpolating curve with control points \( q \)

\[
p(u) = \mathbf{u}^T \mathbf{M}_I \mathbf{q}
\]

• There exist Bezier control points \( p \) such that

\[
p(u) = \mathbf{u}^T \mathbf{M}_B \mathbf{p}
\]

• Equating and solving, we find \( \mathbf{p} = \mathbf{M}_B^{-1} \mathbf{M}_I \)
Interpolating to Bezier

\[ \mathbf{M}_B^{-1} \mathbf{M}_I = \begin{bmatrix}
1 & 0 & 0 & 0 \\
-\frac{5}{6} & 3 & -\frac{3}{2} & 1 \\
\frac{1}{3} & -\frac{3}{2} & 3 & -\frac{5}{6} \\
0 & 0 & 0 & 1
\end{bmatrix} \]

B-Spline to Bezier

\[ \mathbf{M}_B^{-1} \mathbf{M}_S = \begin{bmatrix}
1 & 4 & 1 & 0 \\
0 & 4 & 2 & 0 \\
0 & 2 & 4 & 0 \\
0 & 1 & 4 & 1
\end{bmatrix} \]
Example

These three curves were all generated from the same original data using Bezier recursion by converting all control point data to Bezier control points.

Bezier

Interpolating

B Spline
Surfaces

- Can apply the recursive method to surfaces if we recall that for a Bezier patch curves of constant $u$ (or $v$) are Bezier curves in $u$ (or $v$)
- First subdivide in $u$
  - Process creates new points
  - Some of the original points are discarded
Second Subdivision

- New points created by subdivision
- Old points discarded after subdivision
- Old points retained after subdivision

16 final points for 1 of 4 patches created
Normals

• For rendering we need the normals if we want to shade
  - Can compute from parametric equations
    \[ n = \frac{\partial p(u, v)}{\partial u} \times \frac{\partial p(u, v)}{\partial v} \]
  - Can use vertices of corner points to determine
  - OpenGL can compute automatically
Utah Teapot

• Most famous data set in computer graphics
• Widely available as a list of 306 3D vertices and the indices that define 32 Bezier patches
Algebraic Surfaces

- **Quadric surfaces** are described by implicit equations of the form
  \[ \mathbf{p}^T \mathbf{A} \mathbf{p} + \mathbf{b}^T \mathbf{p} + c = 0. \]
  - 10 independent coefficients \( \mathbf{A}, \mathbf{b}, \) and \( c \) determine the quadric.
  - Ellipsoids, paraboloids, and hyperboloids can be created by different groups of coefficients.
  - Equations for quadric surfaces can be reduced to standard form by affine transformation.
Finding the intersection of a quadric with a ray involves solving a scalar quadratic equation.

- We substitute ray $\mathbf{p} = \mathbf{p}_0 + \alpha \mathbf{d}$ and use the quadratic formula.
- Derivatives determine the normal at a given point.
Quadric Objects in OpenGL

- OpenGL supports disks, cylinders and spheres with quadric objects.

```
GLUquadricObj *qobj;
qobj = gluNewQuadric();

- Choose wire frame rendering with
  gluQuadricDrawStyle(qobj, GLU_LINE);
- To draw an object, pass the reference:
  gluSphere(qobj, RADIUS, SLICES, STACKS);
```
Beziers Curves in OpenGL

Creating a 1D evaluator:

```c
glMap1f(type, u_min, u_max, stride, order, point_array);
```

- `type`: points, colors, normals, textures, etc.
- `u_min, u_max`: range.
- `stride`: points per curve segment.
- `order`: degree + 1.
- `point_array`: control points.
Drawing the Curve

• One evaluator call takes the place of vertex, color, and normal calls.
  - The user enables them with `glEnable`.

```c
typedef float point[3];
point data[] = {...};
glMap1f(GL_MAP_VERTEX_3, 0.0, 1.0, 3, 4, data);
glEnable(GL_MAP_VERTEX_3);

 glBegin(GL_LINE_STRIP)
 for(i=0; i<100; i++) glEvalCoord1f(i/100.);
 glEnd();
```
Beziers Surfaces in OpenGL

• Using a 2D evaluator:

```c
glMap2f(GL_MAP_VERTEX_3, 0, 1, 3, 4, 0, 1, 12, 4, data);
...
for (j = 0; j < 99; j++) {
    glBegin(GL_QUAD_STRIP);
    for (i = 0; i <= 100; i++) {
        glEvalCoord2f(i/100., j/100.);
        glEvalCoord2f((i+1)/100., j/100.);
    }
    glEnd();
}
```
Example: Bezier Teapot

Vertex information goes in an array:

```c
GLfloat data[32][4][4];
```

Initialize the grid for wireframe rendering:

```c
void myInit() {
    glEnable(GL_MAP2_VERTEX_3);
    glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
}
```
Drawing the Teapot

for(k=0; k<32; k++) {
    glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4,
            0, 1, 12, 4, &data[k][0][0][0]);
    for (j=0; j<=8; j++) {
        glBegin(GL_LINE_STRIP);
        for (i=0; i<=30; i++)
            glEvalCoord2f((GLfloat)i/30.0,
                           (GLfloat)j/8.0);
        glEnd();
        glBegin(GL_LINE_STRIP);
        for (i=0; i<=30; i++)
            glEvalCoord2f((GLfloat)j/8.0,
                           (GLfloat)i/30.0);
        glEnd();
    }
}