Office Hours: Week 5

- Monday: 1-2 (Roscoe)
- Tuesday: 1-2 (Roscoe)
- Wednesday: 1-3 (Roscoe)
- Thursday: 1130-1230 (Hank)
- Friday: 1130-1230 (Roscoe)
Timeline (1/2)

• 1E: assigned Thurs Jan 31\textsuperscript{st}, due Weds Feb 6\textsuperscript{th}
  – → will be extra support with this. Tough project.

• 1F: assigned Feb 7\textsuperscript{th} (Feb 1), due Feb 19\textsuperscript{th}
  – → shading is easier than camera, but: movie

• 2A: posted now, due Feb 21\textsuperscript{st}

• → you need to work on both 1F and 2A during Week 6 (Feb 11-15)

• 2B: posted now, due Feb 27\textsuperscript{th}

• YouTube lectures for Feb 12\textsuperscript{th} and 14\textsuperscript{th}
<table>
<thead>
<tr>
<th>Sun</th>
<th>Mon</th>
<th>Tues</th>
<th>Weds</th>
<th>Thurs</th>
<th>Fri</th>
<th>Sat</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><img src="image1.png" alt="Super Bowl" /> Feb 4</td>
<td>Feb 5</td>
<td>Lec 8</td>
<td>Feb 6 1E due</td>
<td>Feb 7 Begin 1F, begin 2A</td>
<td>Feb 8</td>
<td>Feb 9</td>
</tr>
<tr>
<td>Feb 10</td>
<td>Feb 11</td>
<td>YouTube</td>
<td>Feb 13</td>
<td>YouTube??</td>
<td>Feb 15</td>
<td>Feb 16</td>
</tr>
<tr>
<td>Feb 17</td>
<td>Feb 18</td>
<td>Feb 19 1F due</td>
<td>Feb 20</td>
<td>Feb 21 2A due, begin 2B</td>
<td>Feb 22</td>
<td>Feb 23</td>
</tr>
</tbody>
</table>
Comparing to previous terms (1/3)

• Way ahead on lecture
  – If I complete today’s lecture, we will be 1.5 lectures ahead of the pace from previous term
  – Why?:
    • YouTube videos saving on material repeat
    • Bad materials in previous terms, and then have to waste class time fixing things
  – May only need 1 YouTube lecture from Japan
Comparing to previous terms (2/3)

• A little behind on project pace

<table>
<thead>
<tr>
<th>Project</th>
<th>Due date (F16)</th>
<th>Due date (W19)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1E</td>
<td>Monday of Week 5</td>
<td>Weds of Week 5</td>
</tr>
<tr>
<td>1F</td>
<td>Monday of Week 6</td>
<td>Tuesday of Week 7</td>
</tr>
<tr>
<td>2A</td>
<td>Monday of Week 7</td>
<td>Thursday of Week 7</td>
</tr>
<tr>
<td>2B</td>
<td>Monday of Week 8</td>
<td>Wednesday of Week 8</td>
</tr>
</tbody>
</table>

• The W19 plan only works if you pursue *both* 1F and 2A during Week 6!
Comparing to previous terms (3/3)

• Grading way ahead of previous terms
  – Solved a lot of issues & happy about this
• 1E,1F,2A,2B: not quite as prompt
Midterm

• Date still not set
• Considering different plan: 25 & 5
Asking good questions: my own experience
(start recording)
Phong Model

- Combine three lighting effects: ambient, diffuse, specular
Phong Model

- Simple version: 1 light, with “full intensity” (i.e., don’t add an intensity term)

- Phong model
  - \( \text{Shading\_Amount} = K_a + K_d \times \text{Diffuse} + K_s \times \text{Specular} \)

- Signature:
  - double \( \text{CalculatePhongShading} \) (LightingParameters &,
    double *viewDirection, double *normal)
  - Will have to calculate viewDirection for each pixel!
Specular Term of Phong Model

- Specular part of Phong: $K_s \cdot \text{Specular}$
- and Specular is: $(\text{Shininess strength}) \cdot \cos(\alpha)^\text{(shininess coefficient)}$
- Putting it all together would be:
  - $K_s \cdot (\text{Shininess strength}) \cdot \cos(\alpha)^\text{(shininess coefficient)}$
- But now we have two multipliers, $K_s$ and $(\text{Shininess Strength})$. Not needed.
- So: just use one. Drop Shininess Strength and only use $K_s$
  - $K_s \cdot \cos(\alpha)^\text{(shininess coefficient)}$
struct LightingParameters
{
    LightingParameters(void)
    {
        lightDir[0] = -0.6;
        lightDir[1] = 0;
        lightDir[2] = -0.8;
        Ka = 0.3;
        Kd = 0.7;
        Ks = 2.3;
        alpha = 2.5;
    }
};

double lightDir[3]; // The direction of the light source
double Ka;          // The coefficient for ambient lighting.
double Kd;          // The coefficient for diffuse lighting.
double Ks;          // The coefficient for specular lighting.
double alpha;       // The exponent term for specular lighting.
};

LightingParameters lp;
Goal: add shading, movie

Extend your project1E code

Important:

add #define NORMALS
Changes to data structures

class Triangle
{
    public:
        double X[3], Y[3], Z[3];
        double colors[3][3];
        double normals[3][3];
};

reader1e.cxx::GetTriangles() will not compile (with #define NORMALS) until you make these changes

Now initializes normals at each vertex
More comments (1/3)

- This project in a nutshell:
  - Add method called “CalculateShading”.
    - My version of CalculateShading is about ten lines of code.
  - Call CalculateShading for each vertex
  - This is a new field, which you will LERP.
  - Modify RGB calculation to use shading.
More comments (2/3)

- Data to help debug
  - I will make the shading value for each pixel available.
  - I will also make it available for ambient, diffuse, specular.

- Don’t forget to do two-sided lighting for diffuse, one-sided lighting for specular
I haven’t said anything about movie encoders

ffmpeg
Where Hank spent his debugging time...
Where Hank spent his debugging time…
(end recording)
Some notes about OpenGL

- OpenGL has evolved a lot over 25+ years
- The slides that follow and the homeworks will detail an early version of OpenGL (OpenGL V1.0)
- This is the easiest version to understand and implement
  - It is also inefficient
- Since efficiency is important, newer versions are more complex and also faster
  - Optional final projects (developed by Roscoe) will let you play with this
Problem! (big one)

- VTK8 does not work with OpenGL1
- And learning OpenGL3 would require weeks of time
- So we have to roll back to VTK6
- I am very sorry for this

IMPORTANT: I installed VTK6 in Room 100. You can use that for 2A/2B if you don’t want to install again.
Models and Architectures

Ed Angel
Professor of Computer Science, Electrical and Computer Engineering, and Media Arts
University of New Mexico
Objectives

• Learn the basic design of a graphics system
• Introduce pipeline architecture
• Examine software components for an interactive graphics system
Image Formation Revisited

• Can we mimic the synthetic camera model to design graphics hardware software?

• Application Programmer Interface (API)
  - Need only specify
    • Objects
    • Materials
    • Viewer
    • Lights

• But how is the API implemented?
Physical Approaches

• **Ray tracing**: follow rays of light from center of projection until they either are absorbed by objects or go off to infinity
  - Can handle global effects
    • Multiple reflections
    • Translucent objects
  - Slow
  - Must have whole data base available at all times

• **Radiosity**: Energy based approach
  - Very slow
Practical Approach

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

Vertices must be collected into geometric objects before clipping and rasterization can take place
- Line segments
- Polygons
- Curves and surfaces
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.
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
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
The Programmer’s Interface

- Programmer sees the graphics system through a software interface: the Application Programmer Interface (API)
API Contents

• Functions that specify what we need to form an image
  - Objects
  - Viewer
  - Light Source(s)
  - Materials

• Other information
  - Input from devices such as mouse and keyboard
  - Capabilities of system
Object Specification

• Most APIs support a limited set of primitives including
  - Points (0D object)
  - Line segments (1D objects)
  - Polygons (2D objects)
  - Some curves and surfaces
    • Quadrics
    • Parametric polynomials

• All are defined through locations in space or vertices
Example

glBegin(GL_POLYGON);
glVertex3f(0.0, 0.0, 0.0);
glVertex3f(0.0, 1.0, 0.0);
glVertex3f(0.0, 0.0, 1.0);
glEnd();
Lights and Materials

• Types of lights
  - Point sources vs distributed sources
  - Spot lights
  - Near and far sources
  - Color properties

• Material properties
  - Absorption: color properties
  - Scattering
    • Diffuse
    • Specular
Programming with OpenGL
Part 1: Background

Ed Angel
Professor of Computer Science, Electrical and Computer Engineering, and Media Arts
University of New Mexico
Objectives

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

- IFIPS
- FKS
- PHIGS
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
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 Evolution

• Originally controlled by an Architectural Review Board (ARB)
  - Members included SGI, Microsoft, Nvidia, HP, 3DLabs, IBM, ……
  - Relatively stable
    • Evolution reflects new hardware capabilities
      – 3D texture mapping and texture objects
      – Vertex programs
  - Allows for platform specific features through extensions
  - ARB replaced by Kronos
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
- Links with window system
  - GLX for X window systems
  - WGL for Windows
  - AGL for Macintosh
• 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 slide bars

• <GLUT no longer well maintained, we will use VTK>
OpenGL Functions

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

VTK
OpenGL State

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

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

/glVertex3f(x,y,z)/

- function name
- x,y,z are floats
- belongs to GL library

/glVertex3fv(p)/

- p is a pointer to an array
OpenGL #defines

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

Generate a square on a solid background
#include <GL/glut.h>

void mydisplay(){
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_POLYGON);
        glVertex2f(-0.5, -0.5);
        glVertex2f(-0.5, 0.5);
        glVertex2f(0.5, 0.5);
        glVertex2f(0.5, -0.5);
    glEnd();
    glFlush();
}

int main(int argc, char** argv){
    glutCreateWindow("simple");
    glutDisplayFunc(mydisplay);
    glutMainLoop();
}

Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009
Event Loop

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

VTK will be similar ... callback issued to render geometry
Defaults

- `simple.c` is too simple
- Makes heavy use of state variable default values for
  - Viewing
  - Colors
  - Window parameters
- Next version will make the defaults more explicit
How to make a graphics program?

• Need to create a window
  - This window contains a “context” for OpenGL to render in.

• Need to be able to deal with events/interactions

• Need to render graphics primitives
  - OpenGL!
Windows and Events

- Creating windows and dealing with events varies from platform to platform.
“Hello World” with X-Windows.

```c
#include <X11/Xlib.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
    Display *d;
    Window w;
    XEvent e;
    char *msg = "Hello, World!";
    int s;

    d = XOpenDisplay(NULL);
    if (d == NULL) {
        fprintf(stderr, "Cannot open display\n");
        exit(1);
    }

    s = DefaultScreen(d);
    w = XCreateSimpleWindow(d, RootWindow(d, s), 10, 10, 100, 100, 1,
                            BlackPixel(d, s), WhitePixel(d, s));
    XSelectInput(d, w, ExposureMask | KeyPressMask);
    XMapWindow(d, w);

    while (1) {
        XNextEvent(d, &e);
        if (e.type == Expose) {
            XFillRectangle(d, w, DefaultGC(d, s), 20, 20, 10, 10);
            XDrawString(d, w, DefaultGC(d, s), 10, 50, msg, strlen(msg));
        } else if (e.type == KeyPress)
            break;
    }

    XCloseDisplay(d);
    return 0;
}
```
Windows and Events

- Creating windows and dealing with events varies from platform to platform.
- Some packages provide implementations for key platforms (Windows, Unix, Mac) and abstractions for dealing with windows and events.
- GLUT: library for cross-platform windowing & events.
  - My experiments: doesn’t work as well as it used to.
- VTK: library for visualization
  - But also contains cross-platform windowing & events.
Visualization with VTK

Content from: Erik Vidholm, Univ of Uppsula, Sweden
David Gobbi, Robarts Research Institute, London, Ontario, Canada
• Open source, freely available software for 3D computer graphics, image processing, and visualization
• Managed by Kitware Inc.
• Use C++, Tcl/Tk, Python, Java
The visualization pipeline

DATA

FILTER

Visualization algorithms

MAPPING

Interactive feedback

DISPLAY
We will replace these and write our own GL calls.

from vtkpython import *
cone = vtkConeSource()
cone.SetResolution(10)
coneMapper = vtkPolyDataMapper()
coneMapper.SetInput(cone.GetOutput())
coneActor = vtkActor()
coneActor.SetMapper(coneMapper)
ren = vtkRenderer()
ren.AddActor(coneActor)
renWin = vtkRenderWindow()
renWin.SetWindowName("Cone")
renWin.SetSize(300,300)
renWin.AddRenderer(ren)
iren = vtkRenderWindowInteractor()
iren.SetRenderWindow(renWin)
iren.Initialize()
iren.Start()
How to make a graphics program?

• Need to create a window
  - This window contains a “context” for OpenGL to render in.

• Need to be able to deal with events/interactions

• Need to render graphics primitives
  - OpenGL!

Borrow Build
OpenGL Functions

- Primitives
  - Points
  - Line Segments
  - Polygons
- Attributes
- Transformations
  - Viewing
  - Modeling
- Control (VTK)
- Input (VTK)
- Query
First OpenGL programs

• Remember: none of these programs have windowing or events
• They contain just the code to put primitives on the screen, with lighting and colors.
class vtk441PolyDataMapper : public vtkOpenGLPolyDataMapper
{
public:
    static vtk441PolyDataMapper *New();
    virtual void RenderPiece(vtkRenderer *ren, vtkActor *act)
    {
        float ambient[3] = { 1, 1, 1 };{
            glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient);
            glBegin(GL_TRIANGLES);
            glVertex3f(0,0,0);
            glVertex3f(0,1,0);
            glVertex3f(1,1,0);
            glEnd();
    }
};
class vtk441PolyDataMapper : public vtkOpenGLPolyDataMapper
{
    public:
        static vtk441PolyDataMapper * New();
    virtual void RenderPiece(vtkRenderer * ren, vtkActor * act)
    {
        glEnable(GL_COLOR_MATERIAL);
        float ambient[3] = { 1, 1, 1 };
        glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient);
        glBegin(GL_TRIANGLES);
        glColor3ub(0, 0, 255);
        glVertex3f(0,0,0);
        glVertex3f(0,1,0);
        glVertex3f(1,1,0);
        glEnd();
    }
};
glEnable/glDisable: important functions

Both glEnable and glDisable take a single argument, `cap`, which can assume one of the following values:

- **GL_BLEND**
  - If enabled, blend the computed fragment color values with the values in the color buffers. See `glBlendFunc`.

- **GL_CULL_FACE**
  - If enabled, cull polygons based on their winding in window coordinates. See `glCullFace`.

- **GL_DEPTH_TEST**
  - If enabled, do depth comparisons and update the depth buffer. Note that even if the depth buffer exists and the depth mask is non-zero, the depth buffer is not updated if the depth test is disabled. See `glDepthFunc` and `glDepthRange`.

- **GL_DITHER**
  - If enabled, dither color components or indices before they are written to the color buffer.

- **GL_POLYGON_OFFSET_FILL**
  - If enabled, an offset is added to depth values of a polygon's fragments produced by rasterization. See `glPolygonOffset`.

- **GL_SAMPLE_ALPHA_TO_COVERAGE**
  - If enabled, compute a temporary coverage value where each bit is determined by the alpha value at the corresponding sample location. The temporary coverage value is then ANDEd with the fragment coverage value.

- **GL_SAMPLE_COVERAGE**
  - If enabled, the fragment's coverage is ANDEd with the temporary coverage value. If `GL_SAMPLE_COVERAGE_INVERT` is set to `GL_TRUE`, invert the coverage value. See `glSampleCoverage`.

- **GL_SCISSOR_TEST**
  - If enabled, discard fragments that are outside the scissor rectangle. See `glScissor`.

- **GL_STENCIL_TEST**
  - If enabled, do stencil testing and update the stencil buffer. See `glStencilFunc` and `glStencilOp`.

### C Specification

#### void glEnable(GLenum `cap`);

- **Parameters**
  - `cap` Specifies a symbolic constant indicating a GL capability.

#### void glDisable(GLenum `cap`);

- **Parameters**
  - `cap` Specifies a symbolic constant indicating a GL capability.
class vtk441PolyDataMapper : public vtkOpenGLPolyDataMapper
{
    public:
        static vtk441PolyDataMapper *New();
        virtual void RenderPiece(vtkRenderer *ren, vtkActor *act)
        {
            glEnable(GL_COLOR_MATERIAL);
            float ambient[3] = { 1, 1, 1 };
            glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient);
            glBegin(GL_TRIANGLES);
            glColor3ub(0, 0, 255);
            glVertex3f(0, 0, 0);
            glColor3ub(0, 255, 0);
            glVertex3f(0, 1, 0);
            glColor3ub(255, 0, 0);
            glVertex3f(1, 1, 0);
            glEnd();
        }
};
Why is there purple in this picture?
class vtk441PolyDataMapper : public vtkOpenGLPolyDataMapper
{
    public:
        static vtk441PolyDataMapper *New();
        virtual void RenderPiece(vtkRenderer *ren, vtkActor *act)
        {
            glEnable(GL_COLOR_MATERIAL);
            float ambient[3] = { 1, 1, 1 };
            glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient);
            glBegin(GL_TRIANGLES);
            glColor3ub(0, 0, 255);
            glVertex3f(0, 0, 0);
            glColor3ub(0, 255, 0);
            glVertex3f(0, 1, 0);
            glColor3ub(255, 0, 0);
            glVertex3f(1, 1, 0);
            glEnd();
        }
};