Announcements

- OH
  - Hank: Weds 1-2, Thursday 11-12
  - Dan: Weds 4-530, Thursday 930-11

- Emails / Piazza
  - I declare email bankruptcy on all 441 correspondences over the weekend
    - Please re-send or come to OH
Outline

- Review
- Project 1F
- OpenGL Basics
The viewing transformation is not a combination of simple translations, rotations, scales or shears: it is more complex.
Derivation of Viewing Transformation

- I personally don’t think it is a good use of class time to derive this matrix.

- Well derived at:
The View Transformation

- **Input parameters:** \((\alpha, n, f)\)
- **Transforms view frustum to image space cube**
  - **View frustum:** bounded by viewing pyramid and planes \(z=-n\) and \(z=-f\)
  - **Image space cube:** \(-1 \leq u,v,w \leq 1\)
    
    |  cot(\(\alpha/2\)) | 0  | 0  | 0  |
    |-------------------|----|----|----|
    | 0                 | cot(\(\alpha/2\)) | 0  | 0  |
    | 0                 | 0  | \((f+n)/(f-n)\) | -1 |
    | 0                 | 0  | \(2fn/(f-n)\)  | 0  |

- **Cotangent = 1/tangent**
Let’s do an example

- **Input parameters:** \((\alpha, n, f) = (90, 5, 10)\)

\[
\begin{bmatrix}
\cot(\alpha/2) & 0 & 0 & 0 & 0 \\
0 & \cot(\alpha/2) & 0 & 0 & 0 \\
0 & 0 & (f+n)/(f-n) & -1 & 0 \\
0 & 0 & 2fn/(f-n) & 0 & 0
\end{bmatrix}
\]
Let’s do an example

- **Input parameters**: \((\alpha, n, f) = (90, 5, 10)\)

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 3 & -1 & \ \\
0 & 0 & 20 & 0 & \end{bmatrix}
\]
Let’s do an example

- **Input parameters**: $(\alpha, n, f) = (90, 5, 10)$

Let’s multiply some points:

- $(0,7,-6,1)$
- $(0,7,-8,1)$
Let's do an example

- **Input parameters:** \( (\alpha, n, f) = (90, 5, 10) \)

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 3 & -1 \\
0 & 0 & 20 & 0 \\
\end{bmatrix}
\]

Let's multiply some points:

- \((0,7,-6,1) = (0,7,-2,6) = (0, 1.16, -0.33)\)
- \((0,7,-8,1) = (0,7,4,8) = (0, 0.88, 0.5)\)
Let’s do an example

- **Input parameters:** $(\alpha, n, f) = (90, 5, 10)$

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 3 & -1 \\
0 & 0 & 20 & 0 \\
\end{bmatrix}
\]

More points:
- $(0, 7, -4, 1) = (0, 7, -12, 4) = (0, 1.75, -3)$
- $(0, 7, -5, 1) = (0, 7, -15, 3) = (0, 2.33, -1)$
- $(0, 7, -6, 1) = (0, 7, -2, 6) = (0, 1.16, -0.33)$
- $(0, 7, -8, 1) = (0, 7, 4, 8) = (0, 0.88, 0.5)$
- $(0, 7, -10, 1) = (0, 7, 10, 10) = (0, 0.7, 1)$
- $(0, 7, -11, 1) = (0, 7, 13, 11) = (0, .63, 1.18)$
The viewing transformation is not a combination of simple translations, rotations, scales or shears: it is more complex.
View Transformation

More points:
(0,7,-4,1) = (0,7,-12,4) = (0, 1.75, -3)
(0,7,-5,1) = (0,7,-5,5) = (0, 1.4, -1)
(0,7,-6,1) = (0,7,-2,6) = (0, 1.16, -0.33)
(0,7,-8,1) = (0,7,4,8) = (0, 0.88, -0.5)
(0,7,-10,1) = (0,7,10,10) = (0, 0.7, 1)
(0,7,-11,1) = (0,7,13,11) = (0, .63, 1.18)

Note there is a non-linear relationship in Z.
How do we transform?

- For a camera C,
  - Calculate Camera Frame
  - From Camera Frame, calculate Camera Transform
  - Calculate View Transform
  - Calculate Device Transform
  - Compose 3 Matrices into 1 Matrix (M)

- For each triangle T, apply M to each vertex of T, then apply rasterization/zbuffer/Phong shading

```cpp
class Camera {
    public:
        double near, far;
        double angle;
        double position[3];
        double focus[3];
        double up[3];
};
```
Outline

- Review
- Project 1F
- OpenGL Basics
Floating point precision
(not review)

C02LN00GFD58:Downloads hank$ cat tmp.C
#include <iostream>
#include <float.h>

using std::cerr;
using std::endl;

#include <limits>
#include <math.h>

int main()
{
    float X = FLT_MAX;
    float Y = -X;
    float Z = -1;
    float W1 = (X+Y)+Z;
    float W2 = X+(Y+Z);
    printf(stderr, "W1 = %f, W2 = %f\n", W1, W2);
}

C02LN00GFD58:Downloads hank$ g++ tmp.C
C02LN00GFD58:Downloads hank$ ./a.out
W1 = -1.000000, W2 = 0.000000
New policy for pixel differences

Hi Everyone,

It is becoming clear that floating point differences are leading to small differences in output.

So, new policy:
1D: 0 pixels different for full credit (no change)
1E: <= 20 pixels different for full credit. These 20 pixels can be from either of the two posted baselines.
1F: <= 100 pixels different for full credit. (There is an increase because the transformations will involve more floating point math, and thus more differences.)

I apologize for the confusion, and greatly appreciate the folks who have been providing evidence for 1E that we cannot continue to all get the same answer.

Best,
Hank
Goal: add arbitrary camera positions

Extend your project1E code

Re-use:
- `proj1e_geometry.vtk` available on web (9MB), "reader1e.cxx",
- "shading.cxx".

No Cmake, `project1F.cxx`

New: `Matrix.cxx`, `Camera.cxx`
Project #1F, expanded

- Matrix.cxx: complete
- Methods:

```cpp
class Matrix {
public:
    double A[4][4];

    void TransformPoint(const double *ptIn, double *ptOut);
    static Matrix ComposeMatrices(const Matrix &, const Matrix &);
    void Print(ostream &o);
};
```
Camera.cxx: you work on this

class Camera
{
    public:
        double    near, far;
        double    angle;
        double    position[3];
        double    focus[3];
        double    up[3];

        Matrix ViewTransform(void) {;};
        Matrix CameraTransform(void) {;};
        Matrix DeviceTransform(void) {;};
        // Will probably need something for calculating Camera Frame as well

Also: GetCamera(int frame, int nFrames)
Project #1F, deliverables

- Same as usual, but times 4
  - 4 images, corresponding to
    - GetCamera(0, 1000)
    - GetCamera(250, 1000)
    - GetCamera(500, 1000)
    - GetCamera(750, 1000)

- If you want:
  - Generate all thousand images, make a movie
    - Can discuss how to make a movie if there is time
vector<Triangle> t = GetTriangles();
AllocateScreen();
for (int i = 0 ; i < 1000 ; i++)
{
    InitializeScreen();
    Camera c = GetCamera(i, 1000);
    TransformTrianglesToDeviceSpace(); // involves setting up and applying matrices
        //… if you modify vector<Triangle> t,
        // remember to undo it later
    RenderTriangles()
    SaveImage();
}
Correct answers given for `GetCamera(0, 1000)`

Camera Frame: \( U = 0, 0.707107, -0.707107 \)
Camera Frame: \( V = -0.816497, 0.408248, 0.408248 \)
Camera Frame: \( W = 0.57735, 0.57735, 0.57735 \)
Camera Frame: \( O = 40, 40, 40 \)

Camera Transform
\[
\begin{pmatrix}
0.0000000 & -0.8164966 & 0.5773503 & 0.0000000 \\
0.7071068 & 0.4082483 & 0.5773503 & 0.0000000 \\
-0.7071068 & 0.4082483 & 0.5773503 & 0.0000000 \\
0.0000000 & 0.0000000 & -69.2820323 & 1.0000000
\end{pmatrix}
\]

View Transform
\[
\begin{pmatrix}
3.7320508 & 0.0000000 & 0.0000000 & 0.0000000 \\
0.0000000 & 3.7320508 & 0.0000000 & 0.0000000 \\
0.0000000 & 0.0000000 & 1.0512821 & -1.0000000 \\
0.0000000 & 0.0000000 & 10.2564103 & 0.0000000
\end{pmatrix}
\]

Transformed 37.1132, 37.1132, 37.1132, 1 to 0, 0, 1
Transformed -75.4701, -75.4701, -75.4701, 1 to 0, 0, -1
All vertex multiplications use 4D points. Make sure you send in 4D points for input and output, or you will get weird memory errors.

Make sure you divide by w.

Your Phong lighting assumed a view of (0,0,-1). The view will now be changing with each render and you will need to incorporate that view direction in your rendering.
People often get a matrix confused with its transpose. Use the method Matrix::Print() to make sure the matrix you are setting up is what you think it should be. Also, remember the points are left multiplied, not right multiplied.

Regarding multiple renderings:
- Don’t forget to initialize the screen between each render
- If you modify the triangle in place to render, don’t forget to switch it back at the end of the render
Goal: add arbitrary camera positions
You can also generate 1000 images and use a movie encoder to make a movie (i.e., ffmpeg to make a mpeg)

Could someone post a how to?
Project 1F: Shading

- We used viewDirection = (0, 0, -1) for 1E
- For 1F, we will do it right:
  - Prior to transforming, you can calculate viewDirection
    - triangle vertex minus camera position
    - then call CalculateShading with correct viewDir
    - then associate shading value as a scale on the triangle
    - and then LERP that shading value across scanline
Outline

- Review
- Project 1F
- OpenGL Basics
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

  Vertices → Vertex Processor → Clipper and Primitive Assembler → Rasterizer → Fragment Processor → Pixels

• 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

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

type of object
location of vertex

end of object definition
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
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 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 Architecture

Immediate Mode

- Polynomial Evaluator
- Per Vertex Operations & Primitive Assembly
- Rasterization
- Texture Memory
- Frame Buffer

CPU

Display List

Pixel Operations

Per Fragment Operations

geometry pipeline
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

belongs to GL library

function name

vertices belong to

x, y, z are floats

dimensions

$p$ is a pointer to an array

$glVertex3f(x, y, z)$

$glVertex3fv(p)$
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
```c
#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();
}
```
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