CSE167

- Computer Graphics
- Instructor: Ronen Barzel (ronen@graphics.ucsd.edu)
- TAs:
  - Alex Kozlowski (kman@graphics.ucsd.edu)
  - Cameron Chrisman (cchrisman@cs.ucsd.edu)
- Lecture: H&SS 1330 (TTh 8:00am-9:20am)
- Section: TBD
- Office: EBU3B 2104 (T 9:45-10:45am; Th 10:30-11:30am)
- Web page:
  - http://graphics.ucsd.edu/courses/cse167_w06
What is computer graphics?

- Large field...
  - 3D Modeling
  - Image Synthesis (AKA Rendering)
  - Animation
  - Simulation
  - Interaction
  - Image Processing
  - Vision
  - Perception
  - Hardware design
  - Etc...

- Way too much to cover in one quarter
What you’ll learn in this class

- The Basics…
  - How 3D models are defined
    - Polygonal models
    - Smooth surfaces
  - How to render 3D models
    - Camera simulation
    - Lighting, shading
- (Won’t make pretty pictures – not an art class)
- Experience with Linear Algebra, C++, OpenGL
- Basis for going on to advanced topics, game development, etc.
Prerequisites

- Basic Familiarity with:
  - Linear Algebra
    - Vectors: dot products, cross products…
    - Matrices: matrix multiplication
  - C++
    - (if you know Java you’ll be able to adapt)
  - Object oriented programming

- You’ll get more comfortable with them. (And I’ll review Vectors & Matrices today)
Reading

- 3D Computer Graphics: A Mathematical Introduction with OpenGL (Buss)
- Required pages are listed on the class web page
Programming Projects

NOTE: Details of this may change. Check the web page for updates
- Projects due every Tuesday
- Project 1: Matrices and basic graphics program
- Project 2: Matrix stack and hierarchical modeling
- Project 3: Scene graph & traversal
- Project 4: Bézier Curves & Surfaces
- Project 5: Camera Perspective & Clipping
- Project 6: Scan Conversion & Z-Buffer
- Project 7: Ray Casting & Lighting
- Final Project (due last day of class): Choose one of the following
  - Water fountain (particle system)
  - Ray tracer
  - Procedural tree
  - Choose your own project (but talk to me first)
Tests

- Midterm (in class)
  - Thurs, Feb 9, 8:00am-9:20am
  - H&SS 1330
- Final
  - Date & Time: TBD
  - Location: TBD
Grading

- 5% Project 1
- 5% Project 2
- 8% Project 3
- 8% Project 4
- 8% Project 5
- 8% Project 6
- 8% Project 7
- 15% Final Project
- 15% Midterm
- 20% Final
About me

- Pixar animation studios (12 years)
  - Toy Story
    - modeling, lighting, software
  - Software tools development

- Research
  - Physically-based modeling
  - Non-photorealistic rendering
Course Outline

- Introduction / Linear Algebra Review
- Geometry and Homogeneous Coordinates
- Hierarchical Transforms / Geometrical Calculations
- TA Lecture: C++, OpenGL, project implementation
- Hierarchical Modeling
- Cubic Curves
- Curved Surfaces
- Perspective & Viewing
- Clipping & Scan Conversion
- Lighting
- Ray Tracing
- Texture Mapping
- Antialiasing
- Advanced Topics, Guest Lecturers TBD
- Review
Outline for Today

1. Overview of the class
2. Fundamental concepts of graphics
3. Linear Algebra Review
Fundamental concepts of CG

- Modeling
- Rendering
Modeling

- Creating 3D geometric data, AKA the “model” or the “scene”
  - By hand, often with an interactive editing program
    - Maya, Blender, AutoCAD, LightWave 3D, …
  - Procedurally, i.e. by writing programs
  - Scanning real-world objects

- Manipulating the data
  - Deforming or editing the data
    - Change this over time: you have animation!
  - Converting one form into another.
    - Convert higher order descriptions into things we know how to render
    - Convert complex objects into simpler ones that we can render more quickly
  - Makes heavy use of differential geometry and computational geometry

- Ultimately, typically defined as or reduced to polygonal models:
  - Triangles
  - Quadrilaterals
  - Meshes
Modeling
Modeling Primitives

- Complex *scenes*:  
  - Usually built up from simpler *objects*
  - Objects are built from individual *primitives*

- Most fundamental and useful:
  - *3D triangle*
  - Points and lines are also useful primitives
  - May have more complex “primitives”:
    - Polygons, spheres, curves, curved surfaces, …
    - Often automatically *tessellated* into triangles before rendering.
Rendering

- Fancier term: *Image Synthesis*
- Synthesis of a 2D image from a 3D scene description
- Result is a 2D array of pixels
  - Red, Green, Blue values (range 0-255 or 0.0-1.0)
  - Can also have: opacity ("alpha"), depth ("Z"), …
- Rendering style often classified as
  - "Photorealistic"
    - simulate light & camera
  - "Nonphotorealistic" AKA *stylized*
    - aesthetics, communication
Photoreal Rendering
CSE167 Rendering :)

[Images of rendering projects]
Rendering a scene

- Define a “virtual camera”
  - Tells renderer where to look in the scene
Rendering a Scene, cont’d.

- Specify “virtual lights” (and shadows):

- Specify “shaders” for model surfaces:
Hardware vs. Software Rendering

- Highest quality rendering is done by software
  - Algorithms such as “ray tracing”, “photon maps”, etc…
  - Fanciest lighting, shadowing, surface shading, smoke & fog, etc.
  - Can take minutes or hours to compute an image
  - RenderMan (Pixar), Dali (Henrik Wann Jensen), RADIANCE, POVRay, …

- Modern computers often have special-purpose 3D rendering hardware.
  - “GPU” == Graphics Processing Unit. (Nvidia, ATI)
  - Hardware implements the traditional 3D graphics rendering pipeline
  - Very fast, but relatively simple algorithm:
    - Limits ability to get subtle shadows, reflections, etc.
    - Limits on complexity of surface shading description, etc.
  - Continually improving, driven by games industry.

- (Modern graphics hardware is programmable, blurring the distinction between hardware & software rendering.)
OpenGL

- An API for rendering
  - Widely supported on many platforms
  - Provides a standard interface to (platform-specific) hardware.
  - Other APIs: Java3D, Direct3D, …

- Example—render a red triangle (this doesn’t include the ‘setup’ code, or camera or lights):
  ```
  glBegin(GL_TRIANGLES);
  glColor3f(1.0, 0.0, 0.0); // red
  glVertex3f(-4.0, -2.0, 0.0);
  glVertex3f(4.0, -2.0, 0.0);
  glVertex3f(0.0, 5.0, 0.0);
  glEnd();
  ```

- Programming projects will use OpenGL.
  - Lectures won’t focus on OpenGL
  - Bulk of material is about the theory and algorithms
  - (See Buss, ch. 1 for an intro on rendering triangles in OpenGL.)
3-D Graphics Rendering Pipeline

Primitives

Modeling Transformation

Viewing Transformation

Lighting & Shading

Clipping

Projection

Scan conversion, Hiding

Image

Object coordinates

World space

Eye space

Normalized view space

Device coordinates
Outline for Today

1. Overview of the class
2. Fundamental concepts of graphics
3. Linear Algebra Review
   - Vectors
   - Matrices
Coordinate Systems

- Right handed coordinate systems

- (more on coordinate systems next class)
Vector Arithmetic

\[
\mathbf{a} = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix}
\]

\[
\mathbf{a} + \mathbf{b} = \begin{bmatrix} a_x + b_x \\ a_y + b_y \\ a_z + b_z \end{bmatrix} \quad \mathbf{a} - \mathbf{b} = \begin{bmatrix} a_x - b_x \\ a_y - b_y \\ a_z - b_z \end{bmatrix}
\]

\[
-\mathbf{a} = \begin{bmatrix} -a_x \\ -a_y \\ -a_z \end{bmatrix} \quad s\mathbf{a} = \begin{bmatrix} sa_x \\ sa_y \\ sa_z \end{bmatrix}
\]
Vector Magnitude

- The magnitude (length) of a vector is:
  \[ |\mathbf{v}|^2 = v_x^2 + v_y^2 + v_z^2 \]
  \[ |\mathbf{v}| = \sqrt{v_x^2 + v_y^2 + v_z^2} \]

- A vector with length=1.0 is called a *unit vector*
- We can also *normalize* a vector to make it a unit vector:
  \[ \frac{\mathbf{v}}{|\mathbf{v}|} \]
Dot Product

\[ \mathbf{a} \cdot \mathbf{b} = \sum a_i b_i \]

\[ \mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z \]

\[ \mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| |\mathbf{b}| \cos \theta \]
Dot Product

\[ \mathbf{a} \cdot \mathbf{b} = \sum a_i b_i \]

\[ \mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z \]

\[ \mathbf{a} \cdot \mathbf{b} = \begin{bmatrix} a_x & a_y & a_z \end{bmatrix} \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} \]

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

\[ \mathbf{a} \cdot \mathbf{b} = |a||b| \cos \theta \]
Example: Angle Between Vectors

- How do you find the angle $\theta$ between vectors $\mathbf{a}$ and $\mathbf{b}$?
Example: Angle Between Vectors

\[ \mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| |\mathbf{b}| \cos \theta \]

\[
\cos \theta = \left( \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}| |\mathbf{b}|} \right) 
\]

\[
\theta = \cos^{-1} \left( \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}| |\mathbf{b}|} \right) 
\]
Dot Product Properties

The dot product is a scalar value that tells us something about the relationship between two vectors:

- If $\mathbf{a} \cdot \mathbf{b} > 0$ then $\theta < 90^\circ$
  - Vectors point in the same general direction
- If $\mathbf{a} \cdot \mathbf{b} < 0$ then $\theta > 90^\circ$
  - Vectors point in opposite direction
- If $\mathbf{a} \cdot \mathbf{b} = 0$ then $\theta = 90^\circ$
  - Vectors are perpendicular
  - (or one or both of the vectors is degenerate (0,0,0))
Dot Products with One Unit Vector

- If $|\mathbf{u}|=1.0$ then $\mathbf{a} \cdot \mathbf{u}$ is the length of the projection of $\mathbf{a}$ onto $\mathbf{u}$
Dot Products with Unit Vectors

\[ a \cdot b = 0 \]

\[ 0 < a \cdot b < 1 \]

\[ a \cdot b = 1 \]

\[ -1 < a \cdot b < 0 \]

\[ a \cdot b = -1 \]

\[ |a| = |b| = 1.0 \]

\[ a \cdot b = \cos(\theta) \]
Cross Product

\[
\mathbf{a} \times \mathbf{b} = \begin{vmatrix}
i & j & k \\
a_x & a_y & a_z \\
b_x & b_y & b_z
\end{vmatrix}
\]

\[
\mathbf{a} \times \mathbf{b} = \begin{bmatrix}
  a_y b_z - a_z b_y \\
a_z b_x - a_x b_z \\
a_x b_y - a_y b_x
\end{bmatrix}
\]
Properties of the Cross Product

\( \mathbf{a} \times \mathbf{b} \) is a vector perpendicular to both \( \mathbf{a} \) and \( \mathbf{b} \), in the direction defined by the right hand rule

\[
|\mathbf{a} \times \mathbf{b}| = |\mathbf{a}||\mathbf{b}||\sin \theta|
\]

\[
|\mathbf{a} \times \mathbf{b}| = \text{area of parallelogram } ab
\]

\[
|\mathbf{a} \times \mathbf{b}| = 0 \text{ if } \mathbf{a} \text{ and } \mathbf{b} \text{ are parallel (or one or both degenerate)}
\]
Example: Align two vectors

- We are heading in direction \( h \). We want to rotate so that we will align with a different direction \( d \). Find a unit axis \( a \) and an angle \( \theta \) to rotate around.
Example: Align two vectors

\[ a = \frac{h \times d}{|h \times d|} \]

\[ \theta = \sin^{-1}\left(\frac{|h \times d|}{|h||d|}\right) \]

\[ \theta = \cos^{-1}\left(\frac{h \cdot d}{|h||d|}\right) \]

\[ \theta = \tan^{-1}\left(\frac{|h \times d|}{h \cdot d}\right) \]

\[ \text{theta} = \text{atan2}\left(|h \times d|, h \cdot d\right) \]
Vector Class

class Vector3 {
public:
    Vector3() {x=0.0; y=0.0; z=0.0;}
    Vector3(float x0, float y0, float z0) {x=x0; y=y0; z=z0;}
    void Set(float x0, float y0, float z0) {x=x0; y=y0; z=z0;}
    void Add(Vector3 &a) {x+=a.x; y+=a.y; z+=a.z;}
    void Add(Vector3 &a, Vector3 &b) {x=a.x+b.x; y=a.y+b.y; z=a.z+b.z;}
    void Subtract(Vector3 &a) {x-=a.x; y-=a.y; z-=a.z;}
    void Subtract(Vector3 &a, Vector3 &b) {x=a.x-b.x; y=a.y-b.y; z=a.z-b.z;}
    void Negate() {x=-x; y=-y; z=-z;}
    void Negate(Vector3 &a) {x=-a.x; y=-a.y; z=-a.z;}
    void Scale(float s) {x*=s; y*=s; z*=s;}
    void Scale(float s, Vector3 &a) {x=s*a.x; y=s*a.y; z=s*a.z;}
    float Dot(Vector3 &a) {return x*a.x+y*a.y+z*a.z;}
    void Cross(Vector3 &a, Vector3 &b) {
        x=a.y*b.z-a.z*b.y; y=a.z*b.x-a.x*b.z; z=a.x*b.y-a.y*b.x;
    }
    float Magnitude() {return sqrtf(x*x+y*y+z*z);}
    void Normalize() {Scale(1.0f/Magnitude());}

    float x, y, z;
};
Matrix Arithmetic

- Multiply a matrix by a vector:

\[
M = \begin{bmatrix}
  m_{xx} & m_{xy} & m_{xz} \\
  m_{yx} & m_{yy} & m_{yz} \\
  m_{zx} & m_{zy} & m_{zz}
\end{bmatrix}, \quad v = \begin{bmatrix}
  v_x \\
  v_y \\
  v_z
\end{bmatrix}
\]

\[
(Mv)_i = \sum m_{ij}v_j
\]

\[
Mv = \begin{bmatrix}
  \sum m_{xx}v_x + m_{xy}v_y + m_{xz}v_z \\
  \sum m_{yx}v_x + m_{yy}v_y + m_{yz}v_z \\
  \sum m_{zx}v_x + m_{zy}v_y + m_{zz}v_z
\end{bmatrix}
\]

- Each entry is dot product of row of \( M \) with \( v \)
Identity

- Multiplication by the *identity matrix* does not affect the vector

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

\[
\mathbf{I} \mathbf{v} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \begin{bmatrix} 1 \cdot v_x + 0 \cdot v_y + 0 \cdot v_z \\ 0 \cdot v_x + 1 \cdot v_y + 0 \cdot v_z \\ 0 \cdot v_x + 0 \cdot v_y + 1 \cdot v_z \end{bmatrix} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \mathbf{v}
\]

\[
\mathbf{v} = \mathbf{I} \mathbf{v}
\]
Uniform Scaling

- A diagonal matrix scales a vector

\[
S = \begin{bmatrix}
  s & 0 & 0 \\
  0 & s & 0 \\
  0 & 0 & s \\
\end{bmatrix}
\]

\[
Sv = \begin{bmatrix}
  s & 0 & 0 \\
  0 & s & 0 \\
  0 & 0 & s \\
\end{bmatrix} \begin{bmatrix}
  v_x \\
  v_y \\
  v_z \\
\end{bmatrix} = \begin{bmatrix}
  s v_x + 0 v_y + 0 v_z \\
  0 v_x + s v_y + 0 v_z \\
  0 v_x + 0 v_y + s v_z \\
\end{bmatrix} = \begin{bmatrix}
  sv_x \\
  sv_y \\
  sv_z \\
\end{bmatrix} = s \begin{bmatrix}
  v_x \\
  v_y \\
  v_z \\
\end{bmatrix}
\]

\[sv = Sv\]

- If \(s>1\), the vector will grow by a factor of \(s\)
- If \(s=1\), the vector won’t change
- If \(0<s<1\), the vector will shrink
- If \(s<0\), the vector will point in the opposite direction
Non-Uniform Scaling

- Each dimension has its own scale factor

\[
\begin{bmatrix}
    v'_x \\
    v'_y \\
    v'_z
\end{bmatrix} =
\begin{bmatrix}
    s_x & 0 & 0 \\
    0 & s_y & 0 \\
    0 & 0 & s_z
\end{bmatrix}
\begin{bmatrix}
    v_x \\
    v_y \\
    v_z
\end{bmatrix}
\]

which leads to the equations:

- \( v'_x = s_x v_x \)
- \( v'_y = s_y v_y \)
- \( v'_z = s_z v_z \)

- Scaling by 0 in a dimension projects onto that plane
- Scaling by -1 in a dimension reflects across that plane
Rotation

- Rotate in the \(xy\) plane by an angle \(\theta\), spinning around the \(z\) axis:

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

\[
v' = R_z(\theta) \cdot v
\]

\[
v' = \begin{bmatrix}
\cos(\theta)v_x - \sin(\theta)v_y \\
\sin(\theta)v_x + \cos(\theta)v_y \\
v_z
\end{bmatrix}
\]

- A positive angle will rotate counterclockwise when the rotation axis is pointing towards the observer.
  - Right-hand rule

- Food for thought:
  - What is the matrix when \(\theta\) is 0? 90 degrees? 180 degrees?
  - How does a rotation by \(-\theta\) compare to rotation by \(\theta\)?
Rotation about coordinate axes

We can define rotation matrices for each axis:

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

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

\[
\mathbf{R}_z (\theta) = \begin{bmatrix}
\cos \theta & -\sin \theta & 0 \\
\sin \theta & \cos \theta & 0 \\
0 & 0 & 1
\end{bmatrix}
\]
Rotation about arbitrary axis

- This can be derived…

\[
R(a, \theta) = 
\begin{bmatrix}
1 + (1 - \cos(\theta))(a_x^2 - 1) & -a_z \sin(\theta) + (1 - \cos(\theta))a_x a_y & a_y \sin(\theta) + (1 - \cos(\theta))a_x a_z \\
- a_z \sin(\theta) + (1 - \cos(\theta))a_y a_x & 1 + (1 - \cos(\theta))(a_y^2 - 1) & -a_x \sin(\theta) + (1 - \cos(\theta))a_y a_z \\
-a_y \sin(\theta) + (1 - \cos(\theta))a_z a_x & a_x \sin(\theta) + (1 - \cos(\theta))a_z a_y & 1 + (1 - \cos(\theta))(a_z^2 - 1)
\end{bmatrix}
\]

- Note: \(a\) must be a unit vector!

\[|a| = 1\]

- Same right-hand rule applies for direction of rotation
Next class

- Finish up linear algebra review: Matrix Composition
- Points
- Moving points around
- “Homogeneous Coordinates”