Midterm Review

15-426: Computer Graphics
Midterm Course Evaluations

15-426: Computer Graphics
Course Evaluation
Adrien Treuille

1. Please rate the content of the course on a scale of 1 (why am I learning this) to 10 (this is rad!). Comments appreciated...

2. Please rate the speed of the course from 1 (is my grandma teaching this course?) to 5 (just right) to 10 (please stop the train, I’m getting dizzy). Comments appreciated...

3. Please rate how challenging you find this course on a scale from 1 (my dog solves the programming assignments) to 10 (what is happening?!?) Comments appreciated...

4. How would you improve the course?

Please recommend your TAs on the back!
P2 Plagiarism

CHEATING

- Cheating: Please don’t! Using code from the web is ok as long as it is a SMALL percentage of the code for written the assignment.

You CANNOT solve an project requirements with:
- code from the web
- copied from a friend
Our Definition of Plagiarism

• Compared to code on the web or other submitted solutions, your code has:
  • very similar code structure
  • few original comments demonstrating understanding
  • no reference to borrowed code
Name: ______________________________________

Andrew ID: __________________________________

On project 2, I

        DID

        DID NOT

copy code from the web or another student to answer the questions.

Source of code:

Used for which parts of the project:

Signature: _____________________________________
Midterm Topics:

- **Basic Mathematical Concepts**
- **The OpenGL Pipeline**
  - how it works
  - basic passes
- **GPUs**
  - why some computations happen at the vertex level and some at the pixel level
- **Barycentric Coordinates**
- **Viewing transforms**
- **Basic understanding of...**
  - texture mapping
  - bump mapping
  - displacement mapping
- **Implicit and Explicit Surfaces**
  - pluses and minus
  - how to compute implicit functions for basic shapes
  - their union
  - their intersection
  - basic understanding of marching cubes algorithm
  - basic understanding of marching squares algorithm
- **Splines**
  - why we use them
  - how they work
  - how to derive the basis matrices
  - (you don’t need to memorize all the different spline types)
- **Ray Tracing**
  - how it works
  - advantages / disadvantages over OpenGL
  - advanced techniques such as jittering
- **BRDFs and shading models**
  - what is a BRDF
  - how the Phong model works
  - the pluses and minuses of Phong in OpenGL vs. a ray tracer
Midterm Topics:

- **Basic Mathematical Concepts**
- **The OpenGL Pipeline**
  - how it works
  - basic passes
- **GPUs**
  - why some computations happen at the vertex level and some at the pixel level
- **Barycentric Coordinates**
- **Viewing transforms**
- **Basic understanding of...**
  - texture mapping
  - bump mapping
  - displacement mapping
- **Implicit and Explicit Surfaces**
  - pluses and minus
  - how to compute implicit functions for basic shapes
  - their union
  - their intersection
- **Splines**
  - why we use them
  - how they work
  - how to derive the basis matrices
  - (you don't need to memorize all the different spline types)
- **Ray Tracing**
  - how it works
  - advantages / disadvantages over OpenGL
  - advanced techniques such as jittering
- **BRDFs and shading models**
  - what is a BRDF
  - how the Phong model works
  - the pluses and minuses of Phong in OpenGL vs. a ray tracer
Vector arithmetic

• We define the *unary minus* (negative) such that
  \[-a + a = 0\]
• We can then define *subtraction* as
  \[a - b \equiv -b + a\]
• This gives the vector from the end of \(b\) to the end of \(a\) if both have the same origin.
Coordinate systems

- A vector can be multiplied by a scalar to scale the vector’s magnitude without changing its direction:
  \[ \|ka\| = k\|a\| \]
- In 2D, we can represent any vector as a unique linear combination, or weighted sum, of any two non-parallel basis vectors.
- 3D requires three non-parallel, non-coplanar basis vectors.
Coordinate systems

• Basis vectors that are unit vectors at right angles to each other are called **orthonormal**.
• The **x-y Cartesian** coordinate system is a special orthonormal system.
• Vectors are commonly represented in terms of their Cartesian coordinates:

\[
\mathbf{a} = (x_a, y_a) \quad \mathbf{a} = \begin{bmatrix} x_a \\ y_a \end{bmatrix} \quad \mathbf{a}^T = \begin{bmatrix} x_a & y_a \end{bmatrix}
\]
Coordinate systems

• Vectors expressed by orthonormal coordinates
  \[ \mathbf{a} = (x_a, y_a) \]
  have the very useful property that their magnitudes can be calculated according to the Pythagorean Theorem:
  \[ ||\mathbf{a}|| = \sqrt{x_a^2 + y_a^2} \]
Dot product

- We can multiply two vectors by taking the *dot product*.
- The dot product is defined as
  \[ \mathbf{a} \cdot \mathbf{b} = ||\mathbf{a}|| \ ||\mathbf{b}|| \cos \varphi \]
  where \( \varphi \) is the angle between the two vectors.
- Note that the dot product takes two vectors as arguments, but it is often called the *scalar product* because its result is a scalar.
Dot product

Some cool properties:

- It’s often useful in graphics to know the cosine of the angle between two vectors, and we can find it with the dot product:
  \[ \cos \varphi = \mathbf{a} \cdot \mathbf{b} / (||\mathbf{a}|| \, ||\mathbf{b}||) \]

- We can use the dot product to find the projection of one vector onto another. The scalar \( \mathbf{a} \rightarrow \mathbf{b} \) is the magnitude of the vector \( \mathbf{a} \) projected at a right angle onto vector \( \mathbf{b} \), and
  \[ \mathbf{a} \rightarrow \mathbf{b} = ||\mathbf{a}|| \cos \varphi = \mathbf{a} \cdot \mathbf{b} / ||\mathbf{b}|| \]

- Dot products are commutative and distributive:
  \[ \mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a} \]
  \[ \mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) = \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \cdot \mathbf{c} \]
  \[ (k\mathbf{a}) \cdot \mathbf{b} = \mathbf{a} \cdot (k\mathbf{b}) = k(\mathbf{a} \cdot \mathbf{b}) \]
Cross product

- The *cross product* is another vector multiplication operation, usually used only for 3D vectors.
- The direction of \( \mathbf{a} \times \mathbf{b} \) is orthogonal to both \( \mathbf{a} \) and \( \mathbf{b} \).
- The magnitude is equal to the area of the parallelogram formed by the two vectors. It is given by
  \[
  ||\mathbf{a} \times \mathbf{b}|| = ||\mathbf{a}|| \ ||\mathbf{b}|| \ \sin \varphi
  \]
Cross product

Some cool properties:
- Cross products are distributive:
  \[ \mathbf{a} \times (\mathbf{b} + \mathbf{c}) = \mathbf{a} \times \mathbf{b} + \mathbf{a} \times \mathbf{c} \]
  \[ (k\mathbf{a}) \times \mathbf{b} = \mathbf{a} \times (k\mathbf{b}) = k(\mathbf{a} \times \mathbf{b}) \]
- Cross products are intransitive; in fact,
  \[ \mathbf{a} \times \mathbf{b} = -\mathbf{b} \times \mathbf{a} \]
- Because of the sine in the magnitude calculation, for all \( \mathbf{a} \),
  \[ \mathbf{a} \times \mathbf{a} = 0 \]
- In x-y-z Cartesian space,
  \[ \mathbf{x} \times \mathbf{y} = \mathbf{z} \quad \mathbf{y} \times \mathbf{z} = \mathbf{x} \quad \mathbf{z} \times \mathbf{x} = \mathbf{y} \]
Cross product

- As defined on previous slides, the direction of the cross product is ambiguous.
- The *left-hand rule* and the *right-hand rule* distinguish the two choices.
- If \( \mathbf{a} \) points in the direction of your thumb and \( \mathbf{b} \) points in the direction of your index finger, \( \mathbf{a} \times \mathbf{b} \) points in the direction of your middle finger.
- Of the two, the right-hand rule is the predominant convention.
Normal vectors

• A *normal vector* is a vector perpendicular to a surface. A *unit normal* is a normal vector of magnitude one.
• Normal vectors are important to many graphics calculations.
• If the surface is a polygon containing the points $a$, $b$, and $c$, one normal vector

$$n = (b - a) \times (c - a)$$

• This vector points *into* the polygon if $a$, $b$, and $c$ are arranged clockwise; it points outward if they are arranged counterclockwise.
Midterm Topics:

- **Basic Mathematical Concepts**
- **The OpenGL Pipeline**
  - how it works
  - basic passes
- **GPUs**
  - why some computations happen at the vertex level and some at the pixel level
- **Barycentric Coordinates**
- **Viewing transforms**
- **Basic understanding of**...
  - texture mapping
  - bump mapping
  - displacement mapping
- **Implicit and Explicit Surfaces**
  - pluses and minus
  - how to compute implicit functions for basic shapes
  - their union
  - their intersection
  - basic understanding of marching cubes algorithm
  - basic understanding of marching squares algorithm
- **Splines**
  - why we use them
  - how they work
  - how to derive the basis matrices
  - (you don't need to memorize all the different spline types)
- **Ray Tracing**
  - how it works
  - advantages / disadvantages over OpenGL
  - advanced techniques such as jittering
- **BRDFs and shading models**
  - what is a BRDF
  - how the Phong model works
  - the pluses and minuses of Phong in OpenGL vs. a ray tracer
The OpenGL Pipeline

How does it work?
From the implementor’s perspective:

geometric objects
properties: color...
moves camera and objects around

pixels
Frame buffer

graphics pipeline

Primitives
Transformee
Clipper
Projector
Rasterizer
Pixels

Primitives + material properties
Rotate Translate Scale
Is it visible? 3D to 2D
Convert to pixels

* Featuring slides shamelessly stolen from a previous teaching of 15-462!
The OpenGL Pipeline

Primitives: drawing a polygon

Build models in appropriate units (microns, meters, etc.). From simple shapes: triangles, polygons,…

* Featuring slides shamelessly stolen from a previous teaching of 15-462!
The OpenGL Pipeline

Transforms

- Rotate
- Translate
- Scale

- `glRotate(x, y, z);`
- `glTranslate(x, y, z);`
- draw geometry

* Featuring slides shamelessly stolen from a previous teaching of 15-462!
The OpenGL Pipeline

Clipping

Not everything should be visible on the screen

any vertices that lie outside of the viewing volume are clipped

* Featuring slides shamelessly stolen from a previous teaching of 15-462!
The OpenGL Pipeline

Position it relative to the camera

Perspective projection

glFrustum (left, right, bottom, top, near, far);

* Featuring slides shamelessly stolen from a previous teaching of 15-462!
The OpenGL Pipeline

Rasterizer
Go from pixel value in world coordinates to pixel value in screen coordinates

* Featuring slides shamelessly stolen from a previous teaching of 15-462!
Midterm Topics:

- Basic Mathematical Concepts
- The OpenGL Pipeline
  - how it works
  - basic passes
- GPUs
  - why some computations happen at the vertex level and some at the pixel level
- Barycentric Coordinates
- Viewing transforms
- Basic understanding of...
  - texture mapping
  - bump mapping
  - displacement mapping
- Implicit and Explicit Surfaces
  - pluses and minus
  - how to compute implicit functions for basic shapes
    - their union
    - their intersection
  - basic understanding of marching cubes algorithm
  - basic understanding of marching squares algorithm
- Splines
  - why we use them
  - how they work
  - how to derive the basis matrices
  - (you don't need to memorize all the different spline types)
- Ray Tracing
  - how it works
  - advantages / disadvantages over OpenGL
  - advanced techniques such as jittering
- BRDFs and shading models
  - what is a BRDF
  - how the Phong model works
  - the pluses and minuses of Phong in OpenGL vs. a ray tracer
OpenGL Pipeline
More Detail

Vertex Operation

Positions
Normals
Colors

Eye Positions
Eye Normals
Colors

Rasterization (Interpolation)

Position
Normal
Color

(Interpolated)

Fragment Operation

Framebuffer

Eye Positions
Eye Normals
Colors

(Interpolated)

Framebuffer
Programmability

Vertex Operation

Rasterization (Interpolation)

Fragment Operation

Framebuffer

Positions
Normals
Colors

Eye Positions
Eye Normals
Colors

Position
Normal
Color

(Interpolated)

Programmable

Programmable

Framebuffer
Variable Types

- **const** - The declaration is of a compile time constant

- **attribute** - Global variables that may change per vertex, that are passed from the OpenGL application to vertex shaders. This qualifier can only be used in vertex shaders. For the shader this is a read-only variable.

- **varying** - used for interpolated data between a vertex shader and a fragment shader. Available for writing in the vertex shader, and read-only in a fragment shader.

- **uniform** - Global variables that may change per primitive (may not be set inside glBegin,/glEnd), that are passed from the OpenGL application to the shaders. This qualifier can be used in both vertex and fragment shaders. For the shaders this is a read-only variable.
Attribute Variables

Positions, Normals, Colors → Attributes

Vertex Operation
- Programmable
  - Eye Positions
  - Eye Normals
  - Colors

Rasterization (Interpolation)
- Position
- Normal
- Color (Interpolated)

Fragment Operation
- Programmable
  - Framebuffer
Varying Variables

- Positions
- Normals
- Colors
- Attributes

Vertex Operation

- Eye Positions
- Eye Normals
- Colors
- Varying

Programmable

Rasterization (Interpolation)

- Position
- Normal
- Color
- Varying
  (Interpolated)

Fragment Operation

Programmable

Framebuffer
Uniform Variables

**Vertex Operation**
- Programmable
- Positions
- Normals
- Colors
- Attributes

**Rasterization (Interpolation)**
- Eye Positions
- Eye Normals
- Colors
- Varying

**Fragment Operation**
- Programmable
- Position
- Normal
- Color
- Varying
- (Interpolated)

**Framebuffer**
Why So Fast?

Vertex Operation
- Positions
- Normals
- Colors
- Attributes

Rasterization (Interpolation)
- Eye Positions
- Eye Normals
- Colors
- Varying

Uniform
- Position
- Normal
- Color
- Varying (Interpolated)

Fragment Operation
- ~16 Fragment Shaders

Framebuffer

~4 Vertex Shaders

Programmable
Type Properties

• Matrix / vector / integer / floating point types.

• *Strict* casting requirements.

• Const / attribute / varying / uniform variables.
Control Loops in GLSL

```glsl
if (condition) {
    statements;
} else {
    statements;
}

Beware: If Statement Implementation

for (int i=0 ; i<10 ; i++) {
    statements;
}

while (condition) {
    statements;
}
```
Midterm Topics:

- Basic Mathematical Concepts
- The OpenGL Pipeline
  - how it works
  - basic passes
- GPUs
  - why some computations happen at the vertex level and some at the pixel level
- Barycentric Coordinates
- Viewing transforms
- Basic understanding of...
  - texture mapping
  - bump mapping
  - displacement mapping
- Implicit and Explicit Surfaces
  - pluses and minus
  - how to compute implicit functions for basic shapes
  - their union
  - their intersection
  - basic understanding of marching cubes algorithm
  - basic understanding of marching squares algorithm
- Splines
  - why we use them
  - how they work
  - how to derive the basis matrices
  - (you don’t need to memorize all the different spline types)
- Ray Tracing
  - how it works
  - advantages / disadvantages over OpenGL
  - advanced techniques such as jittering
- BRDFs and shading models
  - what is a BRDF
  - how the Phong model works
  - the pluses and minuses of Phong in OpenGL vs. a ray tracer
Why barycentric coordinates?

- Triangles are the fundamental primitive used in 3D modeling programs.
- Triangles are stored as a sequence of three vectors, each defining a vertex.
- Often, we know information about the vertices, such as color, that we’d like to interpolate over the whole triangle.
What are barycentric coordinates?

- The simplest way to do this interpolation is *barycentric coordinates*.
- The name comes from the Greek word *barus* (heavy) because the coordinates are weights assigned to the vertices.
- Point $a$ on the triangle is the origin of the non-orthogonal coordinate system.
- The vectors from $a$ to $b$ and from $a$ to $c$ are taken as basis vectors.
What are barycentric coordinates?

- We can express any point \( p \) coplanar to the triangle as:
  \[ p = a + \beta(b - a) + \gamma(c - a) \]
- Typically, we rewrite this as:
  \[ p(\alpha,\beta,\gamma) = \alpha a + \beta b + \gamma c \]
where \( \alpha \equiv 1 - \beta - \gamma \)
- \( a = p(1,0,0), \ b = p(0,1,0), \ c = p(0,0,1) \)
What are barycentric coordinates?

Some cool properties:
• Point $p$ is inside the triangle if and only if
  \[ 0 < \alpha < 1, \]
  \[ 0 < \beta < 1, \]
  \[ 0 < \gamma < 1 \]
• If one component is zero, $p$ is on an edge.
• If two components are zero, $p$ is on a vertex.
• The coordinates can be used as weighting factors for properties of the vertices, like color.
Barycentric coordinates

Chalkboard examples:
• Conversion from 2D Cartesian
• Conversion from 3D Cartesian
Midterm Topics:

- Basic Mathematical Concepts
- The OpenGL Pipeline
  - how it works
  - basic passes
- GPUs
  - why some computations happen at the vertex level and some at the pixel level
- Barycentric Coordinates
- Viewing transforms
- Basic understanding of...
  - texture mapping
  - bump mapping
  - displacement mapping
- Implicit and Explicit Surfaces
  - pluses and minus
  - how to compute implicit functions for basic shapes
  - their union
  - their intersection
  - basic understanding of marching cubes algorithm
  - basic understanding of marching squares algorithm
- Splines
  - why we use them
  - how they work
  - how to derive the basis matrices
  - (you don't need to memorize all the different spline types)
- Ray Tracing
  - how it works
  - advantages / disadvantages over OpenGL
  - advanced techniques such as jittering
- BRDFs and shading models
  - what is a BRDF
  - how the Phong model works
  - the pluses and minuses of Phong in OpenGL vs. a ray tracer
Perspective and Orthographic Projection
Viewing and Projection

Build this up in stages

• Canonical view volume to screen
• Orthographic projection to canonical view volume
• Perspective projection to orthographic space
Orthographic Projection

the focal point is at infinity, the rays are parallel, and orthogonal to the image plane

good model for telephoto lens. No perspective effects.

when xy-plane is the image plane \((x,y,z) \rightarrow (x,y,0)\)

front orthographic view
Canonical View Volume

Why this shape?
- Easy to clip to
- Trivial to project from 3D to 2D image plane
Orthographic Projection

- X=l  left plane
- X=r  right plane
- Y=b  bottom plane
- Y=t  top plane
- Z=n  near plane
- Z=f  far plane

Why near plane? Prevent points behind the camera being seen
Why far plane? Allows $z$ to be scaled to a limited fixed-point value ($z$-buffering)
Arbitrary View Positions

Eye position: e
Gaze direction: g
View-up vector: t
Perspective Projection

324. The *radio astronomico* used to measure the width of a façade from Gemma’s Frisius’s *De Radio astronomico*. . . . , Antwerp, 1545.

source: http://www.dartmouth.edu/~matc/math5.geometry/unit15/Frisius.gif
The simplest way to look at perspective projection is as painting on a window...

Paint on the window whatever color you see there.
**Simple Perspective Camera**

**Canonical case:**

- camera looks along the z-axis
- focal point is the origin
- image plane is parallel to the xy-plane at distance $d$
- (We call $d$ the focal length, mainly for historical reasons)

![Image Plane Diagram](image.png)
Perspective Projection of a Point

\[ y_s = d \frac{y}{z} \]
Clipping

Something is missing between projection and viewing...
Before projecting, we need to eliminate the portion of scene that is outside the viewing frustum

Need to clip objects to the frustum (truncated pyramid)
Now in a canonical position but it still seems kind of tricky...
Normalizing the Viewing Frustum

Solution: transform frustum to a cube before clipping

Converts perspective frustum to orthographic frustum
Yet another homogeneous transform!
Warping a perspective projection into an orthographic one

Lines for the two projections intersect at the view plane

How can we put this in matrix form?

Need to divide by $z$—haven’t seen a divide in our matrices so far…

Requires our $w$ from last time (or $h$ in the book)
Vanishing Points

What Causes Vanishing Points?
Midterm Topics:

- **Basic Mathematical Concepts**
- **The OpenGL Pipeline**
  - how it works
  - basic passes
- **GPUs**
  - why some computations happen at the vertex level and some at the pixel level
- **Barycentric Coordinates**
- **Viewing transforms**
- **Basic understanding of...**
  - texture mapping
  - bump mapping
  - displacement mapping
- **Implicit and Explicit Surfaces**
  - pluses and minus
  - how to compute implicit functions for basic shapes
  - their union
  - their intersection
- **Splines**
  - why we use them
  - how they work
  - how to derive the basis matrices
  - (you don't need to memorize all the different spline types)
- **Ray Tracing**
  - how it works
  - advantages / disadvantages over OpenGL
  - advanced techniques such as jittering
- **BRDFs and shading models**
  - what is a BRDF
  - how the Phong model works
  - the pluses and minuses of Phong in OpenGL vs. a ray tracer
Midterm Topics:

- Basic Mathematical Concepts
- The OpenGL Pipeline
  - how it works
  - basic passes
- GPUs
  - why some computations happen at the vertex level and some at the pixel level
- Barycentric Coordinates
- Viewing transforms
- Basic understanding of...
  - texture mapping
  - bump mapping
  - displacement mapping
- Implicit and Explicit Surfaces
  - pluses and minus
  - how to compute implicit functions for basic shapes
  - their union
  - their intersection
  - basic understanding of marching cubes algorithm
  - basic understanding of marching squares algorithm
- Splines
  - why we use them
  - how they work
  - how to derive the basis matrices
  - (you don’t need to memorize all the different spline types)
- Ray Tracing
  - how it works
  - advantages / disadvantages over OpenGL
  - advanced techniques such as jittering
- BRDFs and shading models
  - what is a BRDF
  - how the Phong model works
  - the pluses and minuses of Phong in OpenGL vs. a ray tracer
Texture Mapping

- Map reflectance over a piece of geometry
- 2D texture mapping steps:
  - $f(x, y, z)$ mapping function: 3D points to $u, v$ coordinates
  - $g(u, v)$ sampling function: $u, v$ coordinates to color.
Texture Mapping

- The mapping function
  - Easy for simple geometries: cubes, spheres...
  - Not so easy for human body, plant, alien...
    - So it’s usually done manually
Texture Mapping

Figure 5

Spheres...
alien...
Texture Mapping

- The mapping function
  - Easy for simple geometries: cubes, spheres...
  - Not so easy for human body, plant, alien...
    - So it’s usually done manually
- You will texture map spheres in project 2(P247)
Texture Mapping

- The sampling function (P242)
  - Nearest-neighbor
  - Bilinear
    - Linear interpolation on two directions
  - Hermite
    - Similar to bilinear interpolation, weighting neighbor points differently.
Midterm Topics:

- **Basic Mathematical Concepts**
- **The OpenGL Pipeline**
  - how it works
  - basic passes
- **GPUs**
  - why some computations happen at the vertex level and some at the pixel level
- **Barycentric Coordinates**
- **Viewing transforms**
- **Basic understanding of...**
  - texture mapping
  - bump mapping
  - displacement mapping
- **Implicit and Explicit Surfaces**
  - pluses and minus
  - how to compute implicit functions for basic shapes
  - their union
  - their intersection
- basic understanding of marching cubes algorithm
- basic understanding of marching squares algorithm

- **Splines**
  - why we use them
  - how they work
  - how to derive the basis matrices
  - (you don’t need to memorize all the different spline types)

- **Ray Tracing**
  - how it works
  - advantages / disadvantages over OpenGL
  - advanced techniques such as jittering

- **BRDFs and shading models**
  - what is a BRDF
  - how the Phong model works
  - the pluses and minuses of Phong in OpenGL vs. a ray tracer
Bump Mapping

- Texture mapping by itself does not produce very satisfying result.
- What can we do to fix it?
Bump Mapping

- Texture mapping by itself does not produce very satisfying result.
- What can we do to fix it?
  - Normal mapping
Bump Mapping

- Texture mapping by itself does not produce very satisfying result.
- What can we do to fix it?
  - perturb the normals
Displacement Mapping

- Bump mapping is not good enough
  - Bumps do not cast shadow or affect silhouette (they don’t officially exist...)
- The hard way to do it, add more geometric details.
  - heightmap: displacement in the direction of normals.
  - vertex displacement – possible in vertex shader
  - subvertex displacement
    - Shader model 4.0 (DirectX 10), supported only on epic graphics cards (geforce 8800 and above)
    - Requires subdivision, need to generate new vertices
Midterm Topics:

- Basic Mathematical Concepts
- The OpenGL Pipeline
  - how it works
  - basic passes
- GPUs
  - why some computations happen at the vertex level and some at the pixel level
- Barycentric Coordinates
- Viewing transforms
- Basic understanding of...
  - texture mapping
  - bump mapping
  - displacement mapping
- Implicit and Explicit Surfaces
  - pluses and minus
  - how to compute implicit functions for basic shapes
  - their union
  - their intersection
- basic understanding of marching cubes algorithm
- basic understanding of marching squares algorithm
- Splines
  - why we use them
  - how they work
  - how to derive the basis matrices
  - (you don't need to memorize all the different spline types)
- Ray Tracing
  - how it works
  - advantages / disadvantages over OpenGL
  - advanced techniques such as jittering
- BRDFs and shading models
  - what is a BRDF
  - how the Phong model works
  - the pluses and minuses of Phong in OpenGL vs. a ray tracer
Displacement Mapping

- Bump mapping is not good enough
  - Bumps do not cast shadow or affect silhouette (they don’t officially exist...)
- The hard way to do it, add more geometric details.
  - heightmap: displacement in the direction of normals.
  - vertex displacement – possible in vertex shader
  - subvertex displacement
    - Shader model 4.0 (DirectX 10), supported only on epic graphics cards (geforce 8800 and above)
    - Requires subdivision, need to generate new vertices
Displacement Mapping

- How?
  - \( p' = p + f(p) \times n \)
    - \( f(p) \): height value from height map
    - \( p \): point position
    - \( n \): normal
Midterm Topics:

- Basic Mathematical Concepts
- The OpenGL Pipeline
  - how it works
  - basic passes
- GPUs
  - why some computations happen at the vertex level and some at the pixel level
- Barycentric Coordinates
- Viewing transforms
- Basic understanding of...
  - texture mapping
  - bump mapping
  - displacement mapping
- Implicit and Explicit Surfaces
  - pluses and minus
  - how to compute implicit functions for basic shapes
    - their union
    - their intersection
  - basic understanding of marching cubes algorithm
  - basic understanding of marching squares algorithm
- Splines
  - why we use them
  - how they work
  - how to derive the basis matrices
  - (you don't need to memorize all the different spline types)
- Ray Tracing
  - how it works
  - advantages / disadvantages over OpenGL
  - advanced techniques such as jittering
- BRDFs and shading models
  - what is a BRDF
  - how the Phong model works
  - the pluses and minuses of Phong in OpenGL vs. a ray tracer
Outline

• Homework I
• Height Fields
• Normals
• Implicit and Explicit Surfaces
  • **Numerical** vs. Analytic
  • **Implicit** vs. **Explicit**
• Conversions
• Subdivision Surfaces
• Mesh Editing
Implicit $\rightarrow$ Explicit 2D
(Marching Squares Algorithm)
Marching Squares

- Sample function $f$ at every grid point $x_i$, $y_j$
- For every point $f_{ij} = f(x_i, y_j)$ either $f_{ij} \leq c$ or $f_{ij} > c$
Cases for Vertex Labels

16 cases for vertex labels

4 unique mod. symmetries
Ambiguities of Labelings

Ambiguous labels

Different resulting contours

Resolution by subdivision (where possible)
Marching Squares Examples

Can you do better?
Interpolating Intersections

• Approximate intersection
  – Midpoint between $x_i$, $x_{i+1}$ and $y_j$, $y_{j+1}$
  – Better: interpolate

• If $f_{ij} = a$ is closer to $c$ than $b = f_{i+1j}$ then intersection is closer to $(x_i, y_j)$:

\[
\frac{x - x_i}{x_{i+1} - x} = \frac{c - a}{b - c}
\]

• Analogous calculation for y direction

$f_{ij} = a < c \quad c < b = f_{i+1j}$
Marching Squares Examples
Marching Squares Examples

Adaptive Subdivision
Implicit $\rightarrow$ Explicit 3D
(Marching Cubes Algorithm)
3D Scalar Fields

- Volumetric data sets
- Example: tissue density
- Assume again regularly sampled
  
  \[
  \begin{align*}
  x_i &= x_0 + i \Delta x \\
  y_j &= y_0 + j \Delta y \\
  z_k &= z_0 + k \Delta z
  \end{align*}
  \]

- Represent as voxels

- Two rendering methods
  - Isosurface rendering
  - Direct volume rendering (use all values [next])
Isosurfaces

• Generalize contour curves to 3D

• Isosurface given by $f(x,y,z) = c$
  - $f(x, y, z) < c$ inside
  - $f(x, y, z) = c$ surface
  - $f(x, y, z) > c$ outside
Marching Cubes

- Display technique for isosurfaces
- 3D version of marching squares
- How many possible cases?

$2^8 = 256$
Marching Cubes

- 14 cube labelings (after elimination symmetries)
Marching Cube Tessellations

- Generalize marching squares, just more cases
- Interpolate as in 2D
- Ambiguities similar to 2D
Marching Squares Examples

12 Subdivisions

20 Subdivisions

50 Subdivisions
Marching Squares Examples

- 12 Subdivisions
- 20 Subdivisions
- 50 Subdivisions

- 12 Adaptive Subdivisions
- 20 Adaptive Subdivisions
Explicit $\rightarrow$ Implicit 3D
(Fast Marching Algorithm)
Signed Distance Field
Fast Marching
Fast Marching Algorithm
Start Along the Boundary
Push Adjacent Cells
Pop Nearest Cell
Add It’s Neighbors to the Heap
Continue Propagating...
Further and Further...
Until All Cells Have Been Processed
Reminder: Why would we want to do this?
Midterm Topics:

- Basic Mathematical Concepts
  - The OpenGL Pipeline
    - how it works
    - basic passes
  - GPUs
    - why some computations happen at the vertex level and some at the pixel level
- Barycentric Coordinates
- Viewing transforms
- Basic understanding of...
  - texture mapping
  - bump mapping
  - displacement mapping
- Implicit and Explicit Surfaces
  - pluses and minus
  - how to compute implicit functions for basic shapes
    - their union
    - their intersection
  - basic understanding of marching cubes algorithm
  - basic understanding of marching squares algorithm
- Splines
  - why we use them
  - how they work
  - how to derive the basis matrices
    - (you don't need to memorize all the different spline types)
- Ray Tracing
  - how it works
  - advantages / disadvantages over OpenGL
  - advanced techniques such as jittering
- BRDFs and shading models
  - what is a BRDF
  - how the Phong model works
  - the pluses and minuses of Phong in OpenGL vs. a ray tracer
Splines: Piecewise Polynomials

- A spline is a *piecewise polynomial* - many low degree polynomials are used to interpolate (pass through) the control points

- *Cubic piecewise* polynomials are the most common:
  - piecewise definition gives local control
Piecewise Polynomials

- Spline: lots of little polynomials pieced together
- Want to make sure they fit together nicely

- $C_0$ continuity: continuous in position
- $C_0$ & $C_1$ continuity: continuous in position and tangent vector
- $C_0$ & $C_1$ & $C_2$ continuity: continuous in position, tangent, and curvature
Splines

• Types of splines:
  – Hermite Splines
  – Catmull-Rom Splines
  – Bezier Splines
  – Natural Cubic Splines
  – B-Splines
  – NURBS
Hermite Curves

- Cubic Hermite Splines

That is, we want a way to specify the end points and the slope at the end points!
Splines

chalkboard
The Cubic Hermite Spline Equation

• Using some algebra, we obtain:

\[ p(u) = \begin{bmatrix} u^3 & u^2 & u \end{bmatrix} \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ \nabla p_1 \\ \nabla p_2 \end{bmatrix} \]

- point that gets drawn
- basis
- control matrix (what the user gets to pick)

• This form typical for splines
  – basis matrix and meaning of control matrix change with the spline type
The Cubic Hermite Spline Equation

- Using some algebra, we obtain:

\[
p(u) = \begin{bmatrix} u^3 & u^2 & u \end{bmatrix} \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ \nabla p_1 \\ \nabla p_2 \end{bmatrix}
\]

The point that gets drawn

basis

control matrix

(what the user gets to pick)

\[
p(u) = \begin{bmatrix} 2u^3 - 3u^2 + 1 \\ -2u^3 + 3u^2 \\ u^3 - 2u^2 + u \\ u^3 - u^2 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ \nabla p_1 \\ \nabla p_2 \end{bmatrix}
\]

4 Basis Functions
Four Basis Functions for Hermite splines

\[ p(u) = \begin{bmatrix} 2u^3 - 3u^2 + 1 \\ -2u^3 + 3u^2 \\ u^3 - 2u^2 + u \\ u^3 - u^2 \end{bmatrix}^T \begin{bmatrix} p_1 \\ p_2 \\ \nabla p_1 \\ \nabla p_2 \end{bmatrix} \]

4 Basis Functions

Every cubic Hermite spline is a linear combination (blend) of these 4 functions
Piecing together Hermite Curves

- It's easy to make a multi-segment Hermite spline
  - each piece is specified by a cubic Hermite curve
  - just specify the position and tangent at each “joint”
  - the pieces fit together with matched positions and first derivatives
  - gives C1 continuity

- The points that the curve has to pass through are called *knots or knot points*
Problem with Hermite Splines?

- Must explicitly specify derivatives at each endpoint!
- How can we solve this?
Catmull-Rom Splines

- With Hermite splines, the designer must specify all the tangent vectors.
- Catmull-Rom: an interpolating cubic spline with built-in $C^1$ continuity.

\[ \text{tangent at } p_i = s(p_{i+1} - p_{i-1}) \]
Catmull-Rom Splines

- With Hermite splines, the designer must arrange for consecutive tangents to be collinear, to get $C^1$ continuity. This gets tedious.
- Catmull-Rom: an interpolating cubic spline with built-in $C^1$ continuity.
**Catmull-Rom Spline Matrix**

\[ p(u) = \begin{bmatrix} u^3 & u^2 & u \end{bmatrix} \begin{bmatrix} -s & 2-s & s-2 & s \\ 2s & s-3 & 3-2s & -s \\ -s & 0 & s & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix} \]

- spline coefficients
- CR basis
- control vector

- Derived similarly to Hermite
- Parameter \( s \) is typically set to \( s = 1/2 \).
Catmull-Rom Spline Matrix

\[
\begin{bmatrix}
x & y & z \\
x_1 & y_1 & z_1 \\
x_2 & y_2 & z_2 \\
x_3 & y_3 & z_3 \\
x_4 & y_4 & z_4 \\
\end{bmatrix} =
\begin{bmatrix}
-u^3 & u^2 & u & 1 \\
-2s & 2s & s-2 & s \\
2s & s-3 & 3-2s & -s \\
-s & 0 & s & 0 \\
0 & 1 & 0 & 0 \\
\end{bmatrix}
\]

spline coefficients \quad \text{CR basis} \quad \text{control vector}
Catmull-Rom Splines

- With Hermite splines, the designer must specify all the tangent vectors
- Catmull-Rom: an interpolating cubic spline with \textit{built-in} \( C^1 \) continuity.
Catmull-Rom Spline Matrix

\[ p(u) = \begin{bmatrix} u^3 & u^2 & u \end{bmatrix} 1 \begin{bmatrix} -s & 2-s & s-2 & s \\ 2s & s-3 & 3-2s & -s \\ -s & 0 & s & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix} \]

- Derived similarly to Hermite
- Parameter s is typically set to s=1/2.
Cubic Curves in 3D

- **Three cubic polynomials, one for each coordinate**
  - \( x(u) = a_x u^3 + b_x u^2 + c_x u + d_x \)
  - \( y(u) = a_y u^3 + b_y u^2 + c_y u + d_y \)
  - \( z(u) = a_z u^3 + b_z u^2 + c_z u + d_z \)

- **In matrix notation**

\[
\begin{bmatrix}
  x(u) \\
  y(u) \\
  z(u)
\end{bmatrix} =
\begin{bmatrix}
  u^3 & u^2 & u & 1
\end{bmatrix}
\begin{bmatrix}
  a_x & a_y & a_z \\
  b_x & b_y & b_z \\
  c_x & c_y & c_z \\
  d_x & d_y & d_z
\end{bmatrix}
\]
Catmull-Rom Spline Matrix in 3D

\[
\begin{bmatrix}
x(u) & y(u) & z(u)
\end{bmatrix} =
\begin{bmatrix}
u^3 & u^2 & u & 1
\end{bmatrix}
\begin{bmatrix}
-s & 2-s & s-2 & s \\
2s & s-3 & 3-2s & -s \\
-s & 0 & s & 0 \\
0 & 1 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
x_1 & y_1 & z_1 \\
x_2 & y_2 & z_2 \\
x_3 & y_3 & z_3 \\
x_4 & y_4 & z_4
\end{bmatrix}
\]

CR basis  control vector
Midterm Topics:

- Basic Mathematical Concepts
- The OpenGL Pipeline
  - how it works
  - basic passes
- GPUs
  - why some computations happen at the vertex level and some at the pixel level
- Barycentric Coordinates
- Viewing transforms
- Basic understanding of...
  - texture mapping
  - bump mapping
  - displacement mapping
- Implicit and Explicit Surfaces
  - pluses and minus
  - how to compute implicit functions for basic shapes
  - their union
  - their intersection
- basic understanding of marching cubes algorithm
- basic understanding of marching squares algorithm
- Splines
  - why we use them
  - how they work
  - how to derive the basis matrices
  - (you don't need to memorize all the different spline types)
- Ray Tracing
  - how it works
  - advantages / disadvantages over OpenGL
  - advanced techniques such as jittering
- BRDFs and shading models
  - what is a BRDF
  - how the Phong model works
  - the pluses and minuses of Phong in OpenGL vs. a ray tracer
Global vs. Local Rendering Models

Local rendering models: the color of one object is independent of its neighbors (except for shadows)

Missing scattering of light between objects, real shadowing

Global Rendering Models
- Raytracing—specular highlights
- Radiosity—diffuse surfaces, closed environments
Light is Bouncing Photons

Light sources send off photons in all directions
- Model these as particles that bounce off objects in the scene
- Each photon has a wavelength and energy (color and intensity)
- When photons bounce, some energy is absorbed, some reflected, some transmitted

If we can model photon bounces we can generate images

Technique: follow each photon from the light source until:
- All of its energy is absorbed (after too many bounces)
- It departs the known universe (not just the part of the world that is within the viewing volume!)
- It strikes the image and its contribution is added to appropriate pixel
Backward Ray Tracing

Basic idea:

Each pixel gets light from just one direction—the line through the image point and focal point.

Any photon contributing to that pixel’s color has to come from this direction.

So head in that direction and see what is sending light:

- If we hit a light source—done
- If we find nothing—done
- If we hit a surface—see where that surface is lit from

At the end we’ve done forward ray tracing, but ONLY for the rays that contribute to the image.
Ray Tracing

The basic algorithm is

compute u, v, w basis vectors

for each pixel do

shoot ray from eye point through pixel (x,y) into scene

intersect with all surfaces, find first one the ray hits

shade that point to compute pixel (x,y)’s color
Ray Tracing
Computing Rays

\[ p(t) = e + t(s - e) \]

\( t = 0 \) origin of the ray
\( t > 0 \) in positive direction of ray
\( t < 0 \Rightarrow \) then \( p(t) \) is behind the eye
\( t_1 < t_2 \Rightarrow p(t_1) \) is closer to the eye than \( p(t_2) \)
Computing Rays

Where is $s$? ($x,y$ of image)
Intersection of ray with image plane

Details in book. Derived using viewing transformations
Ray Object Intersection

Sphere
Triangle
Polygon

blackboard

Computer Graphics 15-462
Ray Object Intersection

Sphere
Triangle
Polygon

blackboard
Ray Object Intersection

Sphere
Triangle
Polygon

Ray-polygon—In book
Intersection with plane of polygon
in/outside of polygon determination

Ray-triangle—3D models composed of triangles

Ray-sphere—Early models for raytracing, and now bounding volumes
Recursive Ray Tracing

Four ray types:
- Eye rays: originate at the eye
- Shadow rays: from surface point toward light source
- Reflection rays: from surface point in mirror direction
- Transmission rays: from surface point in refracted direction
Writing a Simple Ray Caster (no bounces)

Raycast()  // generate a picture
    for each pixel x,y
        color(pixel) = Trace(ray_through_pixel(x,y))

Trace(ray)  // fire a ray, return RGB radiance
    // of light traveling backward along it
    object_point = Closest_intersection(ray)
    if object_point return Shade(object_point, ray)
    else return Background_Color

Closest_intersection(ray)
    for each surface in scene
        calc_intersection(ray, surface)
    return the closest point of intersection to viewer
    (also return other info about that point, e.g., surface
     normal, material properties, etc.)

Shade(point, ray)  // return radiance of light leaving
    // point in opposite of ray direction
    calculate surface normal vector
    use Phong illumination formula (or something similar)
    to calculate contributions of each light source
Shadow Rays

$p + t \mathbf{l}$ does not hit any objects
$q + t \mathbf{l}$ does hit an object and is shadowed

The same for both points because this is a directional light (infinitely far away)
Four ray types:
- Eye rays: originate at the eye
- Shadow rays: from surface point toward light source
- Reflection rays: from surface point in mirror direction
- Transmission rays: from surface point in refracted direction
Specular Reflection Rays

\[ \mathbf{r} \]

\[ \mathbf{n} \]

\[ \theta \]

\[ \theta \]

\[ \mathbf{d} \]
Ray Tracing Algorithm

- Send ray from eye through each pixel.
- Compute point of closest intersection with a scene surface.
- Shade that point by computing shadow rays.
- Spawn reflected and refracted rays, repeat.
Ray Genealogy

RAY PATHS (BACKWARD)

RAY TREE
When to stop?

When a ray leaves the scene

When its contribution becomes small—at each step the contribution is attenuated by the $K$’s in the illumination model.

\[ I = k_a I_a + f_{att} I_{light} \left[ kd \cos \theta + k_s (\cos \phi)^{n_{shiny}} \right] \]
Ray Casting vs. Ray Tracing

Ray Casting -- 1 bounce

Ray Tracing -- 2 bounces

Ray Tracing -- 3 bounces
Writing a Simple Ray Tracer

Raytrace() // top level function
    for each pixel x,y
        color(pixel) = Trace(ray_through_pixel(x,y))

Trace(ray) // fire a ray, return RGB radiance
    object_point = closest_intersection(ray)
    if object_point return Shade(object_point, ray)
    else return Background_Color
Shade(point, ray) /* return radiance along ray */
    radiance = black; /* initialize color vector */
    for each light source
        shadow_ray = calc_shadow_ray(point, light)
        if !in_shadow(shadow_ray, light)
            radiance += phong_illumination(point, ray, light)
    if material is specularly reflective
        radiance += spec_reflectance * Trace(reflected_ray(point, ray))
    if material is specularly transmissive
        radiance += spec_transmittance * Trace(refracted_ray(point, ray))
    return radiance

Closest_intersection(ray)
    for each surface in scene
        calc_intersection(ray, surface)
    return the closest point of intersection to viewer
    (also return other info about that point, e.g., surface normal, material properties, etc.)
Problem with Simple Ray Tracing: Aliasing
Aliasing

Ray tracing gives a color for every possible point in the image
But a square pixel contains an infinite number of points
These points may not all have the same color
Sampling: choose the color of one point (center of pixel)
Regular sampling leads to aliasing
jaggies
moire patterns
aliasing means one frequency (high) masquerading as another (low)
e.g. wagon wheel effect
Anti-aliasing

Supersampling

Fire more than one ray for each pixel (e.g., a 4x4 grid of rays)

Average the results (perhaps using a filter)
Anti-aliasing: Supersampling

Can be done *adaptively*

- divide pixel into 2x2 grid, trace 5 rays (4 at corners, 1 at center)
- if the colors are similar then just use their average
- otherwise recursively subdivide each cell of grid
- keep going until each 2x2 grid is close to uniform or limit is reached
- filter the result
Supersampling

No antialiasing

3x3 supersampling
3x3 unweighted filter
Temporal Aliasing: Motion Blur

Aliasing happens in time as well as space
- the sampling rate is the frame rate, 30Hz for NTSC video, 24Hz for film
- fast moving objects move large distances between frames
- if we point-sample time, objects have a jerky, strobed look

Real media (film and video) automatically do temporal anti-aliasing
- photographic film integrates over the exposure time
- video cameras have persistence (memory)
- this shows up as motion blur in the photographs

To avoid temporal aliasing we need to filter in time too
- so compute frames at 120Hz and average them together (with appropriate weights)?
- a bit expensive
Motion Blur

Apply stochastic sampling to time as well as space
Assign a time as well as an image position to each ray
The result is still-frame motion blur and smooth animation
Jitter time: \( T = T_0 + \delta(T_1 - T_0) \) for each ray
Example of Motion Blur

From Foley et al. Plate III.16

Rendered using distribution ray tracing at 4096x3550 pixels, 16 samples per pixel.

Note motion-blurred reflections and shadows with penumbrae cast by extended light sources.
Glossy Reflection

Simple ray tracing spawns only one reflected ray—perfect reflection
But Phong illumination models a cone of rays
  Produces fuzzy highlights
  Change fuzziness (cone width) by varying the shininess parameter
Can we generate fuzzy highlights?
  Yes
  But there’s a catch
    we can’t do light reflected from the fuzzy highlight onto other objects
A more accurate model is possible using stochastic sampling
  Stochastically sample rays within the cone
  Sampling probability drops off sharply away from the specular angle
  Highlights can be soft, blurred reflections of other objects
Soft Shadows

Point light sources produce sharp shadow edges
   – the point is either shadowed or not
   – only one ray is required
With an extended light source the surface point may be partially visible to it
   – only part of the light from the sources reaches the point
   – the shadow edges are softer
   – the transition region is the *penumbra*
Accomplish this by
   – firing shadow rays to random points on the light source
   – weighting them by the brightness
   – the resulting shading depends on the fraction of the obstructed shadow rays
Soft Shadows

firing shadow rays to random points on the light source
weighting them by the brightness
the resulting shading depends on the fraction of the obstructed shadow rays
Soft Shadows

fewer rays, more noise

more rays, less noise
Midterm Topics:

- **Basic Mathematical Concepts**
- **The OpenGL Pipeline**
  - how it works
  - basic passes
- **GPUs**
  - why some computations happen at the vertex level and some at the pixel level
- **Barycentric Coordinates**
- **Viewing transforms**
- **Basic understanding of...**
  - texture mapping
  - bump mapping
  - displacement mapping
- **Implicit and Explicit Surfaces**
  - pluses and minus
  - how to compute implicit functions for basic shapes
  - their union
  - their intersection
  - basic understanding of marching cubes algorithm
  - basic understanding of marching squares algorithm
- **Splines**
  - why we use them
  - how they work
  - how to derive the basis matrices
  - (you don't need to memorize all the different spline types)
- **Ray Tracing**
  - how it works
  - advantages / disadvantages over OpenGL
  - advanced techniques such as jittering
- **BRDFs and shading models**
  - what is a BRDF
  - how the Phong model works
  - the pluses and minuses of Phong in OpenGL vs. a ray tracer
Light Transport

Shading (today)
Ray Tracing
Radiosity
Texture Mapping
Reflection Models
Scan Conversion

Later:
Non-photorealistic rendering
Image-based Rendering
Shading: Illumination

Light Sources emit light
   EM spectrum
   Position and direction

Surfaces reflect light
   Reflectance
   Geometry (position, orientation, micro-structure)
   Absorption
   Transmission

Illumination determined by the interactions between light sources and surfaces
Surface Reflection

- When light hits an opaque surface some is absorbed, the rest is reflected (some can be transmitted too—but ignore that for now)
- The reflected light is what we see
- Reflection is not simple and varies with material
  - the surface’s micro structure define the details of reflection
  - variations produce anything from bright specular reflection (mirrors) to dull matte finish (chalk)
What we will learn about today

Light (with color)
Specular highlights
Shadows
No transmission of light through surfaces
No reflections from other surfaces
Diffuse reflectors

Diffuse reflection occurs from dull, matte surfaces, like latex paint, or chalk. These **diffuse** or **Lambertian** reflectors reradiate light equally in all directions. Picture a rough surface with lots of tiny **microfacets**.
Diffuse reflectors

...or picture a surface with little pigment particles embedded beneath the surface (neglect reflection at the surface for the moment):

The microfacets and pigments distribute light rays in all directions.
Embedded pigments are responsible for the coloration of diffusely reflected light in plastics and paints.

Note: the figures above are intuitive, but not strictly (physically) correct.
Diffuse reflectors, cont.

The reflected intensity from a diffuse surface does not depend on the direction of the viewer. The incoming light, though, does depend on the direction of the light source:
Diffuse Reflection

- Simplest kind of reflector (also known as *Lambertian Reflection*)
- Models a matte surface -- rough at the microscopic level
- Ideal diffuse reflector
  - incoming light is scattered equally in all directions
  - viewed brightness does not depend on viewing direction
  - brightness *does* depend on direction of illumination
Lambertian Shading Model

\[ c \propto \cos \theta \]

\[ c \propto n \cdot l \]

\( n \) : surface normal

\( l \) : direction to light

\( \theta \) : Light/Normal angle

\[
\cos \theta = \frac{n \cdot l}{\|n\|\|l\|}
\]
Lambert’s Law

\[ I_{\text{diffuse}} = k_d I_{\text{light}} \cos \theta \]
\[ = k_d I_{\text{light}} (n \cdot l) \]

- \( I_{\text{light}} \): Light Source Intensity
- \( k_d \): Surface reflectance coefficient in [0, 1]
- \( \theta \): Light/Normal angle
What happens for surfaces facing away from the light?

\[ c = c_r c_l \max(0, n \cdot l) \]

\[ c = c_r c_l |n \cdot l| \quad \text{Two-sided light} \]
Wavelength dependence

Really, \(k_e, k_a\), and \(L_a\) are functions over all wavelengths \(\lambda\).

Ideally, we would do the calculation on these functions. For the ambient shading equation, we would start with:

\[
I(\lambda) = k_a(\lambda) L_a(\lambda)
\]

then we would find good RGB values to represent the spectrum \(I(\lambda)\).

Traditionally, though, \(k_a\) and \(I_a\) are represented as RGB triples, and the computation is performed on each color channel separately:

\[
\begin{align*}
I_R &= k_{a,R} L_{a,R} \\
I_G &= k_{a,G} L_{a,G} \\
I_B &= k_{a,B} L_{a,B}
\end{align*}
\]
Examples of Diffuse Illumination

Same sphere lit diffusely from different lighting angles

What happens with surfaces facing away from the light?
Pitch black—not exactly realistic

How to solve?
Several light sources—dim light source at eye, for example
Ambient light
Ambient + Diffuse Reflection

\[ I_{d+a} = k_a I_a + k_d I_{\text{light}} \left( n \cdot l \right) \]
\[ c = c_r \left( c_a + c_l \max(0, n \cdot l) \right) \]

\( I_a \) : Ambient light intensity (global)
\( k_a \) : Ambient reflectance (local)

Diffuse illumination plus a simple ambient light term

a TRICK to account for a background light level caused by multiple reflections from all objects in the scene (less harsh appearance)
Further Simple Illumination Effects

• Light attenuation:
  – light intensity falls off with the square of the distance from the source - so we add an extra term for this

\[ I_{d+a} = k_a I_a + f_{\text{att}} k_d I_{\text{light}} (n \cdot l) \]

where \( f_{\text{att}} = \frac{1}{d^2} \)

with \( d \) the light source to surface distance—more complicated formulae are possible

• Colored lights and surfaces:
  – just have three separate equations for RGB

• Atmospheric attenuation:
  – use viewer-to-surface distance to give extra effects
  – the distance is used to blend the object’s radiant color with a “far” color (e.g., a nice hazy gray)
Specular reflection

Specular reflection accounts for the highlight that you see on some objects. It is particularly important for smooth, shiny surfaces, such as:

- metal
- polished stone
- plastics
- apples
- skin

Properties:

- Specular reflection depends on the viewing direction $V$.
- For non-metals, the color is determined solely by the color of the light.
- For metals, the color may be altered (e.g., brass)
Specular Reflection (Phong Shading)

- Shiny surfaces change appearance when viewpoint is varied
  - specularities (highlights) are view-dependent
  - caused by surfaces that are microscopically smooth (tile floors, gloss paint, whiteboards)
- For shiny surfaces part of the incident light reflects coherently
  - an incoming ray is reflected in a single direction (or narrow beam)
  - direction is defined by the incoming direction and the surface normal
  - when $\sigma$ is near zero, viewer sees reflection
Phong Illumination

- One function that approximates specular falloff is called the *Phong Illumination* model

  \[ c = c_l (e \cdot r) \]

  \[ c = c_l \max(0, e \cdot r)^p \]

  \[ I_{\text{specular}} = k_s I_{\text{light}} (e \cdot r)^p \]

  \[ k_s \quad \text{: Specular reflectance} \]
  
  \[ p \quad \text{: Rate of specular falloff (phong exponent)} \]

\[ ^{30} \text{Greater } p \text{ , more focused beam} \]
Computing the Reflected Ray

blackboard
Or an approximation to the Reflected Ray

blackboard
Phong Illumination

No real physical basis but provides approximately the right answer
Phong Illumination

Moving the light source

Changing p
The Phong illumination model is really a function that maps light from incoming (light) directions $\omega_{in}$ to outgoing (viewing) directions $\omega_{out}$:

$$f_r(\omega_{in}, \omega_{out})$$

This function is called the **Bi-directional Reflectance Distribution Function** (BRDF).

Here’s a plot with $\omega_{in}$ held constant:

BRDF’s can be quite sophisticated…
The Phong illumination model is really a function that maps light from incoming (light) directions $\omega_{\text{in}}$ to outgoing (viewing) directions $\omega_{\text{out}}$:

$$f_r(\omega_{\text{in}}, \omega_{\text{out}})$$

This function is called the Bi-directional Reflectance Distribution Function (BRDF).

Here's a plot with $\omega_{\text{in}}$ held constant:

BRDF's can be quite sophisticated…
BRDFs (Continued)

BRDF Model:

\[ L_o(\omega_o) = \int_{\Omega} L_i(\omega_i) f(\omega_i, \omega_o) d\omega \]

Constraint:

\[ \int_{\Omega} f(\omega_i, \omega_o) d\omega \leq 1 \]