Lecture 5
Basic Rendering

Dr. Lin ZHANG
School of Software Engineering
Tongji University
Spring 2012
At the end of this lecture, you will have enough weapons to generate the above cool figures!
Let’s start this amazing and thorny journey!
Outline

• Overview
• Vector operations
• Ray-casting framework
• Camera coordinate system
• Ray representation
• Ray-surface intersection
• Shading models
• Surface rendering
Realism in computer graphics

- Goal of many graphics algorithms is to create results that are as *physically* realistic as possible
- Realism in graphics can be subdivided into different categories:
  - geometric modeling, rendering, behavior, and interaction
- Techniques and algorithms used to achieve realism in category are dependent on several factors
  - application or content (movies, scientific visualization, etc.)
  - user type (novice vs. expert)
  - resources (time, money, processing power, etc.)
- Degree of realism that can be achieved usually depends on amount of resources
  - when resources are short, use approximations
Realism and media

- Early computer graphics focused on still images
  - typically meant photorealism – goal was to accurately reconstruct a scene at a specific time
  - emphasized good approximations of geometry and light reflection properties of surfaces
Realism and media

- Early computer graphics focused on still images
- With the increased production of animated graphics (commercials, movies, special effects, cartoons) comes new standard of realism
  - character animation
  - natural phenomena: cloth, fur, hair, skin, grass, trees, smoke, water, clouds, wind
  - Newtonian physics: objects which collide, fall, scatter, bend, shatter, etc.
Realism and media

- Always a tradeoff between realism and rendering time
- Applications that require real-time input/output could handle severely limited amount of realism
- Applications which do not require real-time interaction (films, static imagery), have unlimited time to achieve realism

Real-time interaction during modeling (3DS Max)                      Rendered image
Tradeoff

- Consider a movie vs. scientific visualizations and CAD models
  - movie will want to achieve as many special effects as possible (motion blur, depth of field, etc.)
  - CAD (engineering) models will skip these special effects for clarity
  - scientific visualizations will display the geometric model and/or simulation data as is, without any approximations, curve smoothing, etc., to avoid hiding artifacts, approximations, or errors
Realism in geometry

- Polygonalization
  - represent objects as sets of polygons
  - finite level of detail – if we zoom in it doesn’t look good
  - easily hardware accelerated
  - often want simplest mesh possible while preserving as much detail as possible
    - fewer vertices needed at flat, simple portions of mesh
    - more vertices needed at complex varying regions of mesh
Realism in geometry

- Mesh simplification / decimation
  - take detailed model, reduce number of polygons while keeping as much shape as possible
  - various algorithms exist – active area of research

- Meshes can also be smoothed, sharpened, manipulated, etc.
  - same concepts learned in filtering can be applied to 3d models instead of 2d images
  - E.g., smoothing a mesh requires removing high frequencies

Lin ZHANG, SSE, 2012
Realism in geometry

- **Subdivision surfaces**
  - two-step process to add more detail to mesh:
    - Topology refinement (add new vertices)
    - Geometry refinement
  - several schemes for subdividing
  - continuously subdivide triangles into more triangles for more details as needed
  - allows for editing of multi-resolution models (editing at various levels of detail)
Realism in geometry

- Dynamic LOD
  - objects far from the camera don’t need as many polygons, while objects closer should have more
  - replace farther objects with low resolution meshes
  - multiple versions of model in scene graph (differing only in level of detail)

Terrain and water rendered at different LODs
Realism in geometry

- **Bump mapping**
  - Achieved by perturbing the surface normals of the object and using the perturbed normal during lighting calculations
  - Does not modify the underlying geometry
Realism in geometry

- Displacement mapping
  - Displace vertices in a mesh by values stored in a 2D texture
  - Actually modifies the geometry
Realism in rendering

- Multiple light bounces
  - add more lighting to the scene by calculating lighting coming directly from light sources, as well as light reflected off of other objects
  - appears more photorealistic, but much more expensive to compute

- Often “hacked” by using a constant ambient term instead

Left: single ray cast via non-recursive ray tracing (3 sec. render time)

Right: multiple light bounces via photon mapping (57 sec.)
Realism in rendering

- Polygonal rendering and Gouraud shading
  - fast, easily implemented in hardware (default lighting in OpenGL)
  - per vertex lighting interpolates color between vertices
  - lighting is only calculated at vertices from direct light sources
  - no global illumination
  - does not look very realistic (hard shadows, no light reflecting off other objects)
Realism in rendering

- Raytracing
  - shoot rays from the eye into the scene
    - at each intersection calculate amount of light hitting the object
    - if the surface is reflective or refractive, reflect/refract the ray and continue tracing
  - good for shiny, reflective surfaces
  - expensive for global illumination
    - hard to compute multi-bounce lighting
Realism in rendering

- Radiosity
  - subdivide scene into patches, iteratively compute how much light each patch contributes to each other patch
  - good for diffuse global illumination
  - view independent, once solved, can move camera around

A virtual scene rendered by radiosity algorithm
Realism in rendering

- Other rendering techniques:
  - spherical harmonics
  - path tracing
  - point based rendering
  - image based lighting (IBL)
Realism in behavior

- Behavior realism is important
  - we are very distracted by unrealistic behavior even if the rendering is realistic
  - good behavior is very convincing even if the rendering is not realistic

- Keyframing
  - most animations are created through keyframes
  - animators spend hours specifying one key pose after another; in between frames interpolated based on the keyframes
Realism in behavior

- Motion capture (mocap)
  - sample positions and orientations of real world makers over time
  - once captured, apply these motions to a computer model
  - usually better than keyframed animations – extremely realistic
Realism in behavior

- Physical simulation
  - can look extremely realistic
  - usually very expensive to calculate (such as fluid dynamics)
  - realism vs. time tradeoff

Egg explosion simulation

Simulating fracturing of stone
Outline

• Overview
• Vector operations
• Ray-casting framework
• Camera coordinate system
• Ray representation
• Ray-surface intersection
• Shading models
• Surface rendering
Vector operations

Vector representation
\[ \vec{a} = x\vec{i} + y\vec{j} + z\vec{k} = \{x, y, z\} \]

Length (or norm) of a vector
\[ |\vec{a}| = \sqrt{x^2 + y^2 + z^2} \]

Normalized vector (unit vector)
\[ \vec{\hat{a}} = \left\{ \frac{x}{|\vec{a}|}, \frac{y}{|\vec{a}|}, \frac{z}{|\vec{a}|} \right\} \]

We say \( \vec{\hat{a}} = \vec{0} \), if and only if \( x = 0, y = 0, z = 0 \)
Vector operations

if \( \vec{a} = (x_1, y_1, z_1) \), \( \vec{b} = (x_2, y_2, z_2) \),
then \( \vec{a} \pm \vec{b} = (x_1 \pm x_2, y_1 \pm y_2, z_1 \pm z_2) \),

Dot product (inner product)

\[
\vec{a} \cdot \vec{b} = |\vec{a}| |\vec{b}| \cos \theta = x_1 x_2 + y_1 y_2 + z_1 z_2
\]

Laws of dot product:

\[
\vec{a} \cdot \vec{b} = \vec{b} \cdot \vec{a}, \vec{a} \cdot (\vec{b} + \vec{c}) = \vec{a} \cdot \vec{b} + \vec{a} \cdot \vec{c}
\]

Theorem

\[
\vec{a} \cdot \vec{b} = 0 \iff \vec{a} \perp \vec{b} \quad \text{(why?)}
\]
Vector operations

Cross product

\[ \vec{a} \times \vec{b} = \begin{vmatrix} i & j & k \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{vmatrix} = \begin{vmatrix} y_1 & z_1 \\ y_2 & z_2 \end{vmatrix} \hat{i} + \begin{vmatrix} z_1 & x_1 \\ z_2 & x_2 \end{vmatrix} \hat{j} + \begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix} \hat{k} \]
Vector operations

Cross product

\[ \vec{c} = \vec{a} \times \vec{b} \] is also a vector, whose direction is determined by the right-hand law and

\[ \vec{c} \perp \vec{a}, \vec{c} \perp \vec{b} \]

\[ |\vec{c}| = |\vec{a}||\vec{b}| \sin \theta \]

\( \vec{c} \) represents the oriented area of the parallelogram taking \( \vec{a} \) and \( \vec{c} \) as two sides \( \vec{b} \) (easy to prove)

\[ \vec{r}_1 \times \vec{r}_2 = -\vec{r}_2 \times \vec{r}_1 \] (why?)
Vector operations

Cross product

Theorem

\[ \vec{a} \parallel \vec{b} \iff \vec{a} \times \vec{b} = \mathbf{0} \]  
(why?)

Theorem

\[ \vec{a} \parallel \vec{b} \iff \exists \lambda, \mu, \text{ they are not equal to zero at the same time, and } \lambda \vec{a} + \mu \vec{b} = \mathbf{0} \]  
(easy to understand)

Property

\[ \vec{r}_1 \times (\vec{r}_2 + \vec{r}_3) = \vec{r}_1 \times \vec{r}_2 + \vec{r}_1 \times \vec{r}_3 \]
Vector operations

Mixed product (scalar triple product or box product)

\[(a, b, c) = (a \times b) \cdot c = \begin{vmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{vmatrix} \]

Geometric Interpretation: it is the (signed) volume of the parallelepiped defined by the three vectors given
Vector operations

Mixed product (scalar triple product or box product)

$$(a, b, c) = (a \times b) \cdot c = \begin{vmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{vmatrix}$$

$$(a \times b) \cdot c = |a \times b| |c| \cos \alpha$$

$$= |a| |b| \sin \theta \cdot |c| \cos \alpha$$

Base $h$

Lin ZHANG, SSE, 2012
Vector operations

Mixed product (scalar triple product or box product)

\[(a, b, c) = (a \times b) \cdot c = \begin{vmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{vmatrix} \]

Property:

\[(a, b, c) = (b, c, a) = (c, a, b)\]

\[(a, b, c) = -(b, a, c) = -(a, c, b)\]

why?
Vector operations

Mixed product (scalar triple product or box product)

Theorem

\[ \mathbf{a}, \mathbf{b}, \mathbf{c} \text{ are coplanar} \iff (\mathbf{a}, \mathbf{b}, \mathbf{c}) = 0 \]

\[ \exists \lambda, \mu, \nu, \text{ they are not equal to zero at the same time, and} \]
\[ \lambda \mathbf{a} + \mu \mathbf{b} + \nu \mathbf{c} = \mathbf{0} \]
Vector operations

A question often met in applications

Given two vectors $\vec{v}_1$, $\vec{v}_2$, how to construct another vector $\vec{v}_3$, making $\vec{v}_3 \perp \vec{v}_1$

Solution:

$$\vec{v}_p = \left( \frac{\vec{v}_2}{\|\vec{v}_2\|} \cdot \cos \theta \right) \cdot \text{normalized}(\vec{v}_1)$$

$$= \left( \frac{\vec{v}_1 \cdot \vec{v}_2}{\|\vec{v}_1\| \|\vec{v}_2\|} \right) \frac{\vec{v}_1}{\|\vec{v}_1\|} = \left( \frac{\vec{v}_1 \cdot \vec{v}_2}{\|\vec{v}_1\|^2} \right) \vec{v}_1$$

then: $\vec{v}_3 = \vec{v}_2 - \vec{v}_p$
Introduction to vector functions

\[ \mathbf{r}(t) = (x(t), y(t), z(t)) \]

where \( t \) is a variable, is a vector function

Derivative of a vector function

\[ \mathbf{r}'(t) = \left( \frac{dx}{dt}, \frac{dy}{dt}, \frac{dz}{dt} \right) \]

Properties

(1) \[ \left( \lambda(t) \mathbf{r}(t) \right)' = \lambda'(t) \mathbf{r}(t) + \lambda(t) \mathbf{r}'(t) \]

(2) \[ \left( \mathbf{r}_1(t) \cdot \mathbf{r}_2(t) \right)' = \mathbf{r}_1'(t) \cdot \mathbf{r}_2(t) + \mathbf{r}_1(t) \cdot \mathbf{r}_2'(t) \]

(3) \[ \left( \mathbf{r}_1(t) \times \mathbf{r}_2(t) \right)' = \mathbf{r}_1'(t) \times \mathbf{r}_2(t) + \mathbf{r}_1(t) \times \mathbf{r}_2'(t) \]
Introduction to vector functions

\[ \mathbf{r}(t) = (x(t), y(t), z(t)) \]

where \( t \) is a variable, is a vector function

Derivative of a vector function

\[ \mathbf{r}'(t) = \left( \frac{dx}{dt}, \frac{dy}{dt}, \frac{dz}{dt} \right) \]

Properties

(4)

\[ \left( \mathbf{r}_1(t), \mathbf{r}_2(t), \mathbf{r}_3(t) \right)' \]

\[ = \left( \mathbf{r}_1'(t), \mathbf{r}_2(t), \mathbf{r}_3(t) \right) + \left( \mathbf{r}_1(t), \mathbf{r}_2'(t), \mathbf{r}_3(t) \right) + \left( \mathbf{r}_1(t), \mathbf{r}_2(t), \mathbf{r}_3'(t) \right) \]
Introduction to vector functions

Theorem

Length of \( \mathbf{r}(t) \) is const \( \iff \mathbf{r}(t) \cdot \mathbf{r}'(t) = 0 \)

Proof:

\[
|\mathbf{r}(t)| = \text{const} \iff \mathbf{r}(t) \cdot \mathbf{r}(t) = \text{const}^2 \\
\iff \mathbf{r}'(t) \cdot \mathbf{r}(t) + \mathbf{r}(t) \cdot \mathbf{r}'(t) = 0 \\
\iff 2\mathbf{r}'(t) \cdot \mathbf{r}(t) = 0 \\
\iff \mathbf{r}'(t) \cdot \mathbf{r}(t) = 0
\]

Geometric interpretation: \( \mathbf{r}(t) \) is the trace of a (hyper) sphere curve
Theorem

\[ \mathbf{r}(t) \neq 0, \text{ and the direction of } \mathbf{r}(t) \text{ is fixed} \]

\[ \iff \mathbf{r}(t) \times \mathbf{r}'(t) = 0 \]

Geometric interpretation: the trace of \( \mathbf{r}(t) \) is a straight line passing the origin of the coordinate system.
Introduction to vector functions

Theorem
\[ \mathbf{r}(t) \times \mathbf{r}'(t) \neq \mathbf{0}, \text{ and } \mathbf{r}(t) \text{ is parallel to a fixed plane} \]
\[ \iff \left( \mathbf{r}(t), \mathbf{r}'(t), \mathbf{r}''(t) \right) = \mathbf{0} \]

Proof:
Let \( \mathbf{n} \) be the normal of that plane, then \( \mathbf{n} \cdot \mathbf{r}(t) = \mathbf{0} \),
Then,
\[ \mathbf{n} \cdot \mathbf{r}'(t) = \mathbf{0}, \]
\[ \mathbf{n} \cdot \mathbf{r}''(t) = \mathbf{0}, \]
Then, \( \mathbf{n} \) is perpendicular to three vectors \( \mathbf{r}(t), \mathbf{r}'(t), \mathbf{r}''(t) \)
Then, \( \mathbf{r}(t), \mathbf{r}'(t), \mathbf{r}''(t) \) should belong to same plane
Then,
\[ \left( \mathbf{r}(t), \mathbf{r}'(t), \mathbf{r}''(t) \right) = \mathbf{0} \]

Geometric interpretation: \( \mathbf{r}(t) \) locates in a plane passing the origin
Outline

• Overview
• Vector operations
• Ray-casting framework
• Camera coordinate system
• Ray representation
• Ray-surface intersection
• Shading models
• Surface rendering
What is 3D rendering?

- Topics in computer graphics
  - Imaging: representing 2D images
  - Modeling: representing 3D objects
  - **Rendering**: constructing 2D images from 3D models
  - Animation: simulating changes over time
3D rendering scenario I

- Interactive
  - Images generated in fraction of a second (<1/10) as user controls rendering parameters (e.g., camera)
    - Achieve highest quality possible in given time
    - Useful for visualization, games, etc.
3D rendering scenario II

- Batch
  - One image generated with as much quality as possible for a particular set of rendering parameters
  - Take as much time as is needed (minutes)
  - Useful for photorealism, movies, etc.

Avatar
3D rendering pipeline

3D primitives

- Modeling transformation: Transform into 3D world coordinate system
- Lighting: Illuminate according to lighting and reflectance
- Viewing transformation: Transform into 3D camera coordinate system
- Projection transformation: Transform into 2D camera coordinate system
- Clipping: Clip primitives outside camera’s view
- Viewport transformation: Transform into image coordinate system
- Scan conversion: Draw pixels (includes texturing, hidden surface)

2D image
Ray-casting framework

- Ray-casting (a simple version of ray tracing)
  - Generate an image by sending one ray per pixel
  - It is a major rendering algorithm in CG
Ray-casting framework

Image Raycast (Camera cam, Scene scene, int width, int height)
{
    Image image = new Image (width, height) ;
    for (int i = 0 ; i < height ; i++)
        for (int j = 0 ; j < width ; j++)
        {
            Ray ray = RayThruPixel (cam, i, j) ;
            Intersection hit = Intersect (ray, scene) ;
            image[i][j] = FindColor (hit) ;
        }
    return image ;
}
Ray-casting framework illustration

Scene
Ray-casting framework illustration

Scene

Camera
Ray-casting framework illustration
Ray-casting framework illustration
Each pixel corresponds to one ray. We need to figure out which scene point each ray hits.
Ray-casting framework illustration

What’s the color you put in each pixel?
Ray-casting framework illustration

Pixel color determined by shading models
Ray-casting framework illustration

- Following terminologies are equivalent
  - Eye = camera = viewpoint
  - Eye coordinate system = camera coordinate system = viewpoint coordinate system
  - View plane = virtual screen = image plane
Ray-casting framework

Image Raycast (Camera cam, Scene scene, int width, int height)
{
    Image image = new Image (width, height) ;
    for (int i = 0 ; i < height ; i++)
        for (int j = 0 ; j < width ; j++)
        {
            Ray ray = RayThruPixel (cam, i, j) ;
            Intersection hit = Intersect (ray, scene) ;
            image[i][j] = FindColor (hit) ;
        }
    return image ;
}
Outline

• Overview
• Vector operations
• Ray-casting framework
• Camera coordinate system
• Ray representation
• Ray-surface intersection
• Shading models
• Surface rendering
Camera coordinate system

- The scene and the camera are described in WCS
- However, ray-casting can be more conveniently described in the camera coordinate system
Camera coordinate system

- The scene and the camera are described in WCS
- However, ray-casting can be more conveniently described in the camera coordinate system

Here comes the question:

*How to get the coordinates of objects in the camera coordinate system?*

*Answer: we need a transformation matrix to transform the objects defined in WCS to the camera coordinate system*
Camera coordinate system

We assume the camera is at the origin of WCS. To specify the camera coordinate system, two vectors are given, $\mathbf{N}$ and $\mathbf{V}$. ($\mathbf{N}$ and $\mathbf{V}$ are not necessarily perpendicular)

\[
\mathbf{n} = \frac{\mathbf{N}}{|\mathbf{N}|} = (n_x, n_y, n_z)
\]

\[
\mathbf{u} = \frac{\mathbf{V} \times \mathbf{n}}{|\mathbf{V} \times \mathbf{n}|} = (u_x, u_y, u_z)
\]

\[
\mathbf{v} = \mathbf{n} \times \mathbf{u} = (v_x, v_y, v_z)
\]
Camera coordinate system

Then, in the camera coordinate system, the coordinate of the point \((x, y, z, 1)\) defined in the WCS is

\[
\begin{bmatrix}
    x_c \\
    y_c \\
    z_c \\
    1
\end{bmatrix} =
\begin{bmatrix}
    u_x & u_y & u_z & 0 \\
    v_x & v_y & v_z & 0 \\
    n_x & n_y & n_z & 0 \\
    0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
    x \\
    y \\
    z \\
    1
\end{bmatrix}
\]
Camera coordinate system

One step further, if the camera’s position is \((x_0, y_0, z_0, 1)\)

\[
\begin{bmatrix}
  x_c \\
  y_c \\
  z_c \\
  1
\end{bmatrix}
= \begin{bmatrix}
  u_x & u_y & u_z & 0 \\
  v_x & v_y & v_z & 0 \\
  n_x & n_y & n_z & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
  0 & 0 & 0 & -x_0 \\
  0 & 1 & 0 & -y_0 \\
  0 & 0 & 1 & -z_0 \\
  0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
  x \\
  y \\
  z \\
  1
\end{bmatrix}
\]

The final transformation matrix from WCS to camera system is

\[
\begin{bmatrix}
  x_c \\
  y_c \\
  z_c \\
  1
\end{bmatrix}
= \begin{bmatrix}
  u_x & u_y & u_z & -u_x x_0 - u_y y_0 - u_z z_0 \\
  v_x & v_y & v_z & -v_x x_0 - v_y y_0 - v_z z_0 \\
  n_x & n_y & n_z & -n_x x_0 - n_y y_0 - n_z z_0 \\
  0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
  x \\
  y \\
  z \\
  1
\end{bmatrix}
\]

Lin ZHANG, SSE, 2012
Camera coordinate system—summary (1)

- Ray-casting is more convenient expressed in camera coordinate system
  - The scene should be transformed from WCS to the camera coordinate system
- To specify the camera system, you need to provide the camera position $P_0$, the reference point $P_{\text{ref}}$, and the up vector $(V)$ (Note that: $P_0 - P_{\text{ref}}$ determines $N$)
- The center of view-plane resides on the $z_{\text{view}}$; normally the view-plane is perpendicular to $z_{\text{view}}$;
  - So, only parameter needs to be provided to determine the view-plane, that is what?
The camera coordinate system mentioned in our lecture is a right-handed system.

- However, in some existing systems, camera system is left-handed (e.g., OpenGL);

For more details about the 3D pipeline, you can refer to Chapter 7 of the recommended textbook.

In OpenGL, the following three functions are most related:

- `gluLookAt (x0, y0, z0, xref, yref, zref, Vx, Vy, Vz)`
- `glOrtho (xwmin, xwmax, ywmin, ywmax, dnear, dfar)`
- `gluPerspective (theta, aspect, dnear, dfar)`

- Note: in OpenGL, the view-plane coincides with near clipping plane
Ray-casting framework

Image Raycast (Camera cam, Scene scene, int width, int height) {
    Image image = new Image (width, height) ;
    for (int i = 0 ; i < height ; i++)
        for (int j = 0 ; j < width ; j++)
            {
            Ray ray = RayThruPixel (cam, i, j) ;
            Intersection hit = Intersect (ray, scene) ;
            image[i][j] = FindColor (hit) ;
            }
    return image ;
}
Outline

• Overview
• Vector operations
• Ray-casting framework
• Camera coordinate system
• Ray representation
• Ray-surface intersection
• Shading models
• Surface rendering
Ray representation

- Ray representation
  - One point \( P_0 \) (if you select the camera position as \( P_0 \), in the camera coordinate system \( P_0 = (0, 0, 0) \))
  - Direction vector \( \mathbf{d} \) (normalized is better)
  - Parametric line equation: \( \mathbf{r} = \mathbf{P}_0 + \mathbf{d}t \)
Ray representation

- **Note**
  - in most cases, the ray is constructed by linking the eye and each pixel position in the view-plane. In this way, the projection of the scene to the view-plane is actually **perspective** projection (pinhole camera).
  - If each ray takes $z_{view}$ as the direction, you get the **orthographic** (parallel) projection.
  - Both of these two projection modes are supported in OpenGL
Ray representation

Looking different?
Ray-casting framework

Image Raycast (Camera cam, Scene scene, int width, int height) {
    Image image = new Image (width, height) ;
    for (int i = 0 ; i < height ; i++)
        for (int j = 0 ; j < width ; j++)
            {
            Ray ray = RayThruPixel (cam, i, j) ;
            Intersection hit = Intersect (ray, scene) ;
            image[i][j] = FindColor (hit) ;
            }
    return image ;
}
Ray-plane intersection

- Actually, for each ray through, we need to find the closest intersection point

```c
Intersection Intersect (Ray ray, Scene scene)
{
    min_t = infinity; min_primitive = NULL;
    foreach primitive in scene
    {
        t = intersect (ray, primitive) ;
        if (t>0 && t < min_t)
        {
            min_primitive = primitive;
            min_t = t;
        }
    }
    return Intersection(min_t, min_primitive) ;
}
```
Outline

• Overview
• Vector operations
• Ray-casting framework
• Camera coordinate system
• Ray representation
  • Ray-surface intersection
    • Ray-polygon intersection
    • Ray-sphere intersection
    • Ray-cylinder intersection
    • Ray-cone intersection
    • CSG
• Shading models
• Surface rendering
Ray-polygon intersection

- At first, check whether the plane is back-facing; if so, you do not need to calculate intersection since the “eye” cannot see such a face at all!
- Each planar polygon is enclosed by boundary edges in a counterclockwise manner; its normal points outward.

The “eye” can see this face

The “eye” cannot see this face
Ray-polygon intersection

• At first, check whether the plane is back-facing; if so, you do not need to calculate intersection since the “eye” cannot see such a face at all!

• Each planar polygon is enclosed by boundary edges in a counterclockwise manner; its normal points outward

\[ \mathbf{n} \cdot \mathbf{d} < 0 \quad \text{The “eye” can see this face} \]

Then, you can calculate the intersection point of the ray and the planar polygon
Ray-polygon intersection

• The equation of the infinite plane

The plane passes the known point $\mathbf{r}_0$, and its normal is $\mathbf{n}$

Its equation is

$$\mathbf{r} \cdot \mathbf{n} - \mathbf{n} \cdot \mathbf{r}_0 = 0$$

Why?

We let $D = \mathbf{n} \cdot \mathbf{r}_0$, then

$$\mathbf{r} \cdot \mathbf{n} - D = 0$$
Ray-polygon intersection

- Intersection of the infinite plane with the ray

Plane equation \( \mathbf{r} \cdot \mathbf{n} - D = 0 \)
Ray equation \( \mathbf{r} = \mathbf{P}_0 + \mathbf{d} t \)

Then, we have

\[
(P_0 + dt) \cdot \mathbf{n} - D = 0
\]

\[
t = \frac{D - P_0 \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}
\]

The intersection point is

\[
\mathbf{r} = \mathbf{P}_0 + \mathbf{d} \frac{D - \mathbf{P}_0 \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}
\]
Ray-polygon intersection

- Intersection of the infinite plane with the ray

**Plane equation** \( \mathbf{r} \cdot \mathbf{n} - D = 0 \)

**Ray equation** \( \mathbf{r} = \mathbf{P}_0 + \mathbf{d}t \)

The intersection point is \( \mathbf{r} = \mathbf{P}_0 + \frac{D - \mathbf{P}_0 \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}} \)

E.g.

Plane: passing \((1, 0, 0), (0, 1, 0), \) and \((0, 0, 1)\)

Ray: the Z-axis

Calculate their intersection point

Can I work it out?
Ray-polygon intersection

- Intersection of the infinite plane with the ray

Plane equation \( \mathbf{r} \cdot \mathbf{n} - D = 0 \)
Ray equation \( \mathbf{r} = \mathbf{P}_0 + d t \)
The intersection point is \( \mathbf{r} = \mathbf{P}_0 + d \frac{D - \mathbf{P}_0 \cdot \mathbf{n}}{d \cdot \mathbf{n}} \)

It is straightforward to get the distance between the camera (eye) and the intersection point

**How to decide whether the intersection point is within a planar polygon mesh?**
Ray-polygon intersection

- In or out checking

Check the direction of the following cross products for the two cases:

\[ PA \times PB, PB \times PC, PC \times PD \]
\[ PD \times PE, PE \times PA \]

If \( P \) locates within the polygon, all the cross products have the same direction; otherwise, the directions would be different.

Any limitations?
Special case: ray-triangle intersection

- Barycentric definition of a plane
  - A (non-degenerate) triangle \((a, b, c)\) defines a plane
  - Any point \(P\) on this plane can be written as
    \[
    P(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c \quad , \text{where} \quad \alpha + \beta + \gamma = 1
    \]
Special case: ray-triangle intersection

- Barycentric definition of a plane

Since \( \alpha = 1 - \beta - \gamma \), we have

\[
P(\beta, \gamma) = (1 - \beta - \gamma)a + \beta b + \gamma c = a + \beta(b - a) + \gamma(c - a)
\]
Special case: ray-triangle intersection

- Barycentric definition of a plane

Since $\alpha = 1 - \beta - \gamma$, we have

$$P(\beta, \gamma) = (1 - \beta - \gamma)a + \beta b + \gamma c = a + \beta(b - a) + \gamma(c - a)$$

If we require that $\alpha, \beta, \gamma \geq 0$, we get just the triangle!

E.g., if $\beta = 0$, $P$ must lines on the line $c-a$
Special case: ray-triangle intersection

- Barycentric definition of a plane

Given \( \mathbf{a}, \mathbf{b}, \mathbf{c}, \) and \( \mathbf{P} \), how to compute \( \beta, \gamma \)?

\[
\mathbf{P} = \mathbf{a} + \beta (\mathbf{b} - \mathbf{a}) + \gamma (\mathbf{c} - \mathbf{a})
\]

\[
\mathbf{a} + \beta \mathbf{e}_1 + \gamma \mathbf{e}_2 - \mathbf{P} = 0
\]

\[
\begin{align*}
\begin{pmatrix}
\mathbf{e}_1 \cdot \mathbf{e}_1 & \mathbf{e}_1 \cdot \mathbf{e}_2 \\
\mathbf{e}_1 \cdot \mathbf{e}_2 & \mathbf{e}_2 \cdot \mathbf{e}_2
\end{pmatrix}
\begin{pmatrix}
\beta \\
\gamma
\end{pmatrix}
&= 
\begin{pmatrix}
\mathbf{e}_1 \cdot (\mathbf{P} - \mathbf{a}) \\
\mathbf{e}_2 \cdot (\mathbf{P} - \mathbf{a})
\end{pmatrix}
\end{align*}
\]

\[
\begin{pmatrix}
\beta \\
\gamma
\end{pmatrix}
= 
\begin{pmatrix}
\mathbf{e}_1 \cdot \mathbf{e}_1 & \mathbf{e}_1 \cdot \mathbf{e}_2 \\
\mathbf{e}_1 \cdot \mathbf{e}_2 & \mathbf{e}_2 \cdot \mathbf{e}_2
\end{pmatrix}^{-1}
\begin{pmatrix}
\mathbf{e}_1 \cdot (\mathbf{P} - \mathbf{a}) \\
\mathbf{e}_2 \cdot (\mathbf{P} - \mathbf{a})
\end{pmatrix}
\]
Special case: ray-triangle intersection

- Intersection of a ray and barycentric triangle

Ray equation \( \mathbf{r} = \mathbf{P}_0 + \mathbf{d}t \)

Points on the barycentric triangle

\[
\mathbf{r} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a})
\]

The ray intersects with the triangle if

\[
\mathbf{P}_0 + \mathbf{d}t = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a})
\]

with \( \beta + \gamma < 1, \beta > 0, \gamma > 0 \)
Special case: ray-triangle intersection

- Intersection of a ray and barycentric triangle

\[ P_0 + dt = a + \beta(b - a) + \gamma(c - a) \]

\[ \begin{align*}
P_{0x} + d_x t &= a_x + \beta(b_x - a_x) + \gamma(c_x - a_x) \\
P_{0y} + d_y t &= a_y + \beta(b_y - a_y) + \gamma(c_y - a_y) \\
P_{0z} + d_z t &= a_z + \beta(b_z - a_z) + \gamma(c_z - a_z)
\end{align*} \]

Re-group and write in the matrix form \( Ax = b \)

\[ \begin{align*}
\beta(a_x - b_x) + \gamma(a_x - c_x) + d_x t &= a_x - P_{0x} \\
\beta(a_y - b_y) + \gamma(a_y - c_y) + d_y t &= a_y - P_{0y} \\
\beta(a_z - b_z) + \gamma(a_z - c_z) + d_z t &= a_z - P_{0z}
\end{align*} \]
Special case: ray-triangle intersection

- Intersection of a ray and barycentric triangle

\[
\begin{bmatrix}
a_x - b_x & a_x - c_x & d_x \\
a_y - b_y & a_y - c_y & d_y \\
a_z - b_z & a_z - c_z & d_z \\
\end{bmatrix}
\begin{bmatrix}
\beta \\
\gamma \\
t
\end{bmatrix}
= \begin{bmatrix}
a_x - P_{0x} \\
a_y - P_{0y} \\
a_z - P_{0z}
\end{bmatrix}
\]

\[
A
\]

Can be easily solved by Cramer’s rule,

\[
\beta = \frac{\begin{bmatrix}
a_x - P_{0x} & a_x - c_x & d_x \\
a_y - P_{0y} & a_y - c_y & d_y \\
a_z - P_{0z} & a_z - c_z & d_z \\
\end{bmatrix}}{|A|},
\gamma = \frac{\begin{bmatrix}
a_x - b_x & a_x - P_{0x} & d_x \\
a_y - b_y & a_y - P_{0y} & d_y \\
a_z - b_z & a_z - P_{0z} & d_z \\
\end{bmatrix}}{|A|},
\]

\[
t = \frac{\begin{bmatrix}
a_x - b_x & a_x - c_x & a_x - P_{0x} \\
a_y - b_y & a_y - c_y & a_y - P_{0y} \\
a_z - b_z & a_z - c_z & a_z - P_{0z} \\
\end{bmatrix}}{|A|}
\]

Can be copied mechanically into code for implementation!
Special case: ray-triangle intersection

- Pros of barycentric representation for intersection
  - Efficient
  - Stores no plane equation
  - Get the barycentric coordinates for free
    - Useful for interpolation, texture mapping, etc.
Special case: ray-triangle intersection

- Barycentric interpolation
  - Values $v_1, v_2, v_3$ defined at $a, b, c$
  - They can be colors, normals, texture coordinates etc
  - $P(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c$ is the point
  - Then, $v(\alpha, \beta, \gamma) = \alpha v_1 + \beta v_2 + \gamma v_3$ is the barycentric interpolation of $v_1 \sim v_3$ at point $P$
    - i.e., once you know $\alpha, \beta, \gamma$, you can interpolate values using the same weights, very convenient

Lin ZHANG, SSE, 2012
Outline

• Overview
• Vector operations
• Ray-casting framework
• Camera coordinate system
• Ray representation
• Ray-surface intersection
  • Ray-polygon intersection
  • Ray-sphere intersection
  • Ray-cylinder intersection
  • Ray-cone intersection
  • CSG
• Shading models
• Surface rendering

Lin ZHANG, SSE, 2012
Ray-sphere intersection

- Can be used when the object in the scene is described in explicit analytical form of sphere.

Suppose that the center of the sphere is $S_0$ and its radius is $R$. Its equation is

$$\left(\mathbf{r} - S_0\right) \cdot \left(\mathbf{r} - S_0\right) = R^2 \quad (1)$$

The ray equation is $\mathbf{r} = \mathbf{P}_0 + \mathbf{d}t$ (d is a unit vector) (2)
Ray-sphere intersection

- Can be used when the object in the scene is described in explicit analytical form of sphere.

Suppose that the center of the sphere is $S_0$ and its radius is $R$. Its equation is

$$\mathbf{(r - S_0) \cdot (r - S_0)} = R^2 \quad (1)$$

The ray equation is $\mathbf{r = P_0} + \mathbf{d}t$ ($\mathbf{d}$ is a unit vector) \((2)\)

Combine (1) and (2), we can get

$$t^2 + 2(\mathbf{P_0 - S_0}) \cdot \mathbf{d}t + (\mathbf{P_0 - S_0})^2 - R^2 = 0$$

\[b \quad c\]

If $b^2 - 4c < 0$, the ray does not intersect with the sphere.

Otherwise $t = \frac{-b \pm \sqrt{b^2 - 4c}}{2}$
Outline

• Overview
• Vector operations
• Ray-casting framework
• Camera coordinate system
• Ray representation
• Ray-surface intersection
  • Ray-polygon intersection
  • Ray-sphere intersection
  • Ray-cylinder intersection
  • Ray-cone intersection
  • CSG
• Shading models
• Surface rendering
Ray-cylinder intersection

- Can be used when the cylinder is analytically described

Cylinder: the center of the bottom face is \( S_0 \), radius is \( R \), height is \( h \), the central symmetric axis is \( AXIS \),

Such a solid cylinder can be described as

\[
\begin{align*}
(r - S_0) \cdot (r - S_0) - ((r - S_0) \cdot AXIS)^2 &= R^2 \quad (1) \\
(r - S_0) \cdot AXIS &\geq 0 \quad (2) \\
(r - S_1) \cdot AXIS &\leq 0 \\
\text{Since } S_1 &= S_0 + h \cdot AXIS
\end{align*}
\]

\[
(r - S_0) \cdot AXIS - h \leq 0 \quad (3)
\]
Ray-cylinder intersection

- Can be used when the cylinder is analytically described

Cylinder:

\[
\begin{align*}
(r - S_0) \cdot (r - S_0) - ((r - S_0) \cdot AXIS)^2 &= R^2 \\
(r - S_0) \cdot AXIS &\geq 0 \\
(r - S_0) \cdot AXIS - h &\leq 0
\end{align*}
\]  

Ray: \( r = P_0 + dt \)

\[
(P_0 + dt - S_0) \cdot (P_0 + dt - S_0) - ((P_0 + dt - S_0) \cdot AXIS)^2 = R^2
\]

Simplify it, we have:
Ray-cylinder intersection

\[ at^2 + 2bt + c = 0 \]

where,

\[ a = 1 - (AXIS \cdot d)^2 \]

\[ b = d \cdot (P_0 - d_0) - (AXIS \cdot d)(AXIS \cdot (P_0 - d_0)) \]

\[ c = (P_0 - d_0)^2 - [AXIS \cdot (P_0 - d_0)]^2 - R^2 \]

Then,

\[ t_{1,2} = \frac{-b \pm \sqrt{b^2 - ac}}{a} \]

Then, you need to check whether the intersection points really reside on the cylindrical surface bounded by the bottom and top faces. That is, whether it satisfies (2) and (3)

\[
\begin{cases}
(r - S_0) \cdot AXIS \geq 0 & \text{(2)} \\
(r - S_0) \cdot AXIS - h \leq 0 & \text{(3)}
\end{cases}
\]
Outline

• Overview
• Vector operations
• Ray-casting framework
• Camera coordinate system
• Ray representation
• Ray-surface intersection
  • Ray-polygon intersection
  • Ray-sphere intersection
  • Ray-cylinder intersection
  • Ray-cone intersection
  • CSG
• Shading models
• Surface rendering
Ray-cone intersection

- Can be used when the cone is analytically described

Cone: the apex of the cone is $S_0$, half apex angle is $\theta$, height is $h$, the central symmetric axis is $AXIS$,

Try to describe such a cone in a mathematical formula and compute its intersection with the ray $r = P_0 + dt$
Outline

• Overview
• Vector operations
• Ray-casting framework
• Camera coordinate system
• Ray representation
• Ray-surface intersection
  • Ray-polygon intersection
  • Ray-sphere intersection
  • Ray-cylinder intersection
  • Ray-cone intersection
  • CSG
• Shading models
• Surface rendering
Constructive solid geometry

- **CSG**
  - Build a complex object based on a set of primitives
  - Three basic boolean operations: union, intersection, and subtraction
- **CSG can be naturally implemented in the ray-casting framework**

### 4 cases

- **Union**
- **Intersection**
- **Subtraction**
Constructive solid geometry

- CSG implementation in ray-casting framework
  - Step 1: compute the intersection points (actually ray parameters $t_s$) with left child node and right child node
  - Step 2: sort the intersection points according to their corresponding parameters
  - Step 3: classify the line segments in the final results as “in” or “out”
  - Step 4: remove the redundant points for the “in” line segments; after this step, any two adjacent line segments belong to different categories (“in” or “out”)
  - Above 4-steps procedure actually is a recursive process since each child node is also a CSG tree

Lin ZHANG, SSE, 2012
## Constructive solid geometry

- CSG implementation in ray-casting framework

Rules for line segments classification

<table>
<thead>
<tr>
<th>Operator</th>
<th>Left node</th>
<th>Right node</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>union</td>
<td>in</td>
<td>in</td>
<td>in</td>
</tr>
<tr>
<td></td>
<td>in</td>
<td>out</td>
<td>in</td>
</tr>
<tr>
<td></td>
<td>out</td>
<td>in</td>
<td>in</td>
</tr>
<tr>
<td></td>
<td>out</td>
<td>out</td>
<td>in</td>
</tr>
<tr>
<td>intersection</td>
<td>in</td>
<td>in</td>
<td>in</td>
</tr>
<tr>
<td></td>
<td>in</td>
<td>out</td>
<td>out</td>
</tr>
<tr>
<td></td>
<td>out</td>
<td>in</td>
<td>out</td>
</tr>
<tr>
<td></td>
<td>out</td>
<td>out</td>
<td>out</td>
</tr>
<tr>
<td>subtraction</td>
<td>in</td>
<td>in</td>
<td>out</td>
</tr>
<tr>
<td></td>
<td>in</td>
<td>out</td>
<td>in</td>
</tr>
<tr>
<td></td>
<td>out</td>
<td>in</td>
<td>out</td>
</tr>
<tr>
<td></td>
<td>out</td>
<td>out</td>
<td>out</td>
</tr>
</tbody>
</table>
Constructive solid geometry

- CSG implementation in ray-casting framework

E.g., union of A and B

![Diagram of ray casting with A and B intersecting]

Step 1: intersections with $A$: $t_1$, $t_2$; intersections with $B$: $t_3$, $t_4$

Step 2: sort the points, $t_1$, $t_3$, $t_2$, $t_4$

Step 3: classify the line segments

Step 4: remove redundant points
Constructive solid geometry

- CSG implementation in ray-casting framework

E.g., intersection of \( A \) and \( B \)

![Diagram of intersecting shapes]

**Step 1:** intersections with \( A \): \( t_1, t_2 \); intersections with \( B \): \( t_3, t_4 \)

**Step 2:** sort the points, \( t_1, t_3, t_2, t_4 \)

**Step 3:** classify the line segments

**Step 4:** remove redundant points

Lin ZHANG, SSE, 2012
Constructive solid geometry

- CSG implementation in ray-casting framework

E.g., difference of $A$ and $B$

Step 1: intersections with $A$: $t_1, t_2$; intersections with $B$: $t_3, t_4$

Step 2: sort the points, $t_1, t_3, t_2, t_4$

Step 3: classify the line segments

Step 4: remove redundant points
Ray-casting framework

Image Raycast (Camera cam, Scene scene, int width, int height)
{
    Image image = new Image (width, height) ;
    for (int i = 0 ; i < height ; i++)
        for (int j = 0 ; j < width ; j++)
            {
            Ray ray = RayThruPixel (cam, i, j) ;
            Intersection hit = Intersect (ray, scene) ;
            image[i][j] = FindColor (hit) ;
        }
    return image ;
}
Outline

• Overview
• Vector operations
• Ray-casting framework
• Camera coordinate system
• Ray representation
• Ray-surface intersection

• Shading models
  • Light sources
  • Phong shading model
  • Shadow
  • Transparency

• Surface rendering
Shading model

- Shading model, or illumination model, or lighting model is used to mimic the physical world to decide the “color” of a point in the visual scene.
- The object’s “color” is decided by:
  - The light
  - The attributes of the object
Point light source

- The most simple one
- It is omni-directional
- Attributes to specify a point light source
  - Position \((p_x, p_y, p_z)\)
  - Intensity \(I\) (if it is a chromatic light, three values representing R, G, and B are needed)
  - Coefficients \((a_0, a_1, a_2)\) to specify its attenuation property with distance \(d\)

\[
\frac{1}{a_0 + a_1 d + a_2 d^2}
\]

\[
I_d = f_{\text{radatten}} I
\]

Lin ZHANG, SSE, 2012
Point light source

- The most simple one
- It is omni-directional
- Attributes to specify a point light source
  - Position \((p_x, p_y, p_z)\)
  - Intensity \(I\) (if it is a chromatic light, three values representing R, G, and B are needed)
  - Coefficients \((a_0, a_1, a_2)\) to specify its attenuation property with distance \(d\)

In this lecture, except otherwise specified, the light source is considered as monochromatic. For a practical chromatic light, simply repeat the theories to R, G, and B respectively, and combine the results together.
Directional light source

- Besides position and color, the apex angle of the lighting cone needs to be specified.

\[ f_{\text{angatten}} = \begin{cases} 0.0, & \text{when } \mathbf{V}_{\text{obj}} \cdot \mathbf{V}_{\text{light}} < \cos \theta_l \\ \left( \mathbf{V}_{\text{obj}} \cdot \mathbf{V}_{\text{light}} \right)^\alpha, & \text{otherwise} \end{cases} \]

where \( \alpha \) is a constant.

\( f_{\text{radatten}}, f_{\text{angatten}} \) can be combined.
Directional light source

- Besides position and color, the apex angle of the lighting cone needs to be specified.
Outline

• Overview
• Vector operations
• Ray-casting framework
• Camera coordinate system
• Ray representation
• Ray-surface intersection
• Shading models
  • Light sources
  • Phong shading model
  • Shadow
  • Transparency
• Surface rendering
Phong shading model (single light)

- Developed by Bui Tuong Phong (裴祥风)
  - (Vietnamese, 1942-1975)
  - In 1973, he developed the Phong shading model
  - He dies in 1975 for leukaemia

- By Phong model, the light energy emitted from a point on an object to the observer’s eye comprises three components
  - Ambient lighting
  - Diffuse reflection
  - Specular reflection
Phong shading model (single light)

- Ambient lighting
  - Ambient lighting is a constant for a scene $I_a$
  - Different surface can have different ambient reflection coefficient $k_a$ (0 <= $k_a$ <= 1)
  - So, if only consider ambient lighting, the illumination at a point simply is $I = I_a k_a$

Rendering results when only ambient lighting is considered
Phong shading model (single light)

- **Diffuse reflection**
  - Diffuse reflection is the reflection of light from a surface such that an incident ray is reflected at many angles rather than at just one angle as in the case of specular reflection (chalk, clay)
  - An illuminated ideal diffuse reflecting surface will have equal luminance from all directions in the hemisphere surrounding the surface
Phong shading model (single light)

- Diffuse reflection
  - Diffuse reflection is the reflection of light from a surface such that an incident ray is reflected at many angles rather than at just one angle as in the case of specular reflection (chalk, clay)
  - An illuminated ideal diffuse reflecting surface will have equal luminance from all directions in the hemisphere surrounding the surface
  - Lambert reflection law

\[
I_{\text{diff}} = \begin{cases} 
I_L k_d (\mathbf{N} \cdot \mathbf{L}), & \mathbf{N} \cdot \mathbf{L} > 0 \\
0, & \mathbf{N} \cdot \mathbf{L} \leq 0 \end{cases} 
\]

where, \( \mathbf{N} \) is the normal of the surface at the examined position, \( \mathbf{L} \) is the direction of the light, \( k_d \) is the diffuse reflection coefficient of the surface, \( I_L \) is the intensity of the incident light
Phong shading model (single light)

- Diffuse reflection
  - Diffuse reflection is the reflection of light from a surface such that an incident ray is reflected at many angles rather than at just one angle as in the case of specular reflection (chalk, clay)
  - An illuminated ideal diffuse reflecting surface will have equal luminance from all directions in the hemisphere surrounding the surface
  - Lambert reflection law

Or simply write as

\[ I_{\text{diff}} = I_L k_d \max(N \cdot L, 0) \]
Phong shading model (single light)

When considering both the ambient lighting and the diffusion reflection, the observed intensity of a point can be described as

\[ I = I_a k_a + I_L k_d \ \max(N \cdot L, 0) \]

Illuminated by ambient lighting and diffuse reflection

Left: diffuse reflection only
Right: ambient + diffuse reflection
Phong shading model (single light)

- Specular reflection
  - Specular reflection is the mirror-like reflection of light from a surface, in which light from a single incoming direction (a ray) is reflected into a single outgoing direction
  - Used to model mirrors, metals, etc.
Phong shading model (single light)

- Specular reflection
  - Specular reflection is the mirror-like reflection of light from a surface, in which light from a single incoming direction (a ray) is reflected into a single outgoing direction
  - Used to model mirrors, metals, etc.

\[
\begin{align*}
\text{Surface normal} & \quad \text{Direction of reflection} \\
\end{align*}
\]

The observed light is
\[
I_{spec} = I_L k_s \cos^n(\phi) = I_L k_s (\mathbf{V} \cdot \mathbf{R})^n
\]

\[
\mathbf{R} = (2\mathbf{N} \cdot \mathbf{L})\mathbf{N} - \mathbf{L}
\]

\[ k_s = \text{specular reflection coefficient} \]
\[ n = \text{specular exponent} \]
Phong shading model (single light)

- Specular reflection (a simplified version)

Surface normal: $\mathbf{N}$
Direction of reflection: $\mathbf{R} - \mathbf{N}$

Then, $I_{spec} = I_L k_s (\mathbf{V} \cdot \mathbf{R})^n$
More accurately, $I_{spec} = I_L k_s (\mathbf{H} \cdot \mathbf{N})^n$

$$\mathbf{H} = \frac{\mathbf{L} + \mathbf{V}}{\|\mathbf{L} + \mathbf{V}\|}$$

Lin ZHANG, SSE, 2012
Phong shading model (single light)

When considering the ambient lighting, the diffusion reflection, and the specular reflection at the same time, the observed intensity of a point can be described as

\[ I = I_a k_a + I_L \left( k_d \max(N \cdot L, 0) + k_s \left( \max(N \cdot H, 0) \right)^n \right) \]

This is called as Phong shading model!!
Phong shading model (single light)

When considering the ambient lighting, the diffusion reflection, and the specular reflection at the same time, the observed intensity of a point can be described as

\[ I = I_a k_a + I_L \left( k_d \max(N \cdot L, 0) + k_s \left( \max(N \cdot H, 0) \right)^n \right) \]

(two light sources) with increased \( n \)
Phong shading model (multi-lights)

- Multi-lights
  - Single light case can be easily extended to multi-light case
  - Suppose there are $m$ lights in the scene,

$$I = I_a k_a + \sum_{i=1}^{m} I_{L_i} \left( k_d \max(N \cdot L_i, 0) + k_s \left( \max(N \cdot H_i, 0) \right)^n \right)$$
Outline

• Overview
• Vector operations
• Ray-casting framework
• Camera coordinate system
• Ray representation
• Ray-surface intersection
• Shading models
  • Light sources
  • Phong shading model
  • Shadow
  • Transparency
• Surface rendering
Shadows

- In previous discussions, we always assume that the light source can “see” the object.
- Shadow term tells if a light source is blocked.
  - Cast a ray towards the light source $L_i$.
  - $s_i = 0$, if the ray is blocked; $s_i = 1$, otherwise.

The light cannot illuminate the sphere; the light ray is blocked by the triangle.
Shadows

- In previous discussions, we always assume that the light source can “see” the object
- Shadow term tells if a light source is blocked
  - Cast a ray towards the light source $L_i$
  - $s_i = 0$, if the ray is blocked; $s_i = 1$, otherwise
  - Then, the Phong shading model becomes

\[
I = I_a k_a + \sum_{i=1}^{m} I_{L_i} s_i \left( k_d \max(N \cdot L_i, 0) + k_s \left( \max(N \cdot H_i, 0) \right)^n \right)
\]

Scene rendering results when considering shadow terms
Outline

- Overview
- Vector operations
- Ray-casting framework
- Camera coordinate system
- Ray representation
- Ray-surface intersection
- Shading models
  - Light sources
  - Phong shading model
  - Shadow
  - Transparency
- Polygonal rendering
Transparency

- If an object is transparent to some extent, the observer can see the objects behind it.
- Accurate transparency modeling needs the ray-tracing algorithm, which is quite time-consuming.
Transparency

- A simple transparency model

\[ I = (1 - k_t)I_{refl} + k_t I_{trans} \]

Where \( I_{refl} \) is the intensity of the examined point by using the Phong shading mode; \( I_{trans} \) is the intensity of the intersection point (with the viewpoint-ray) behind the transparent object; \( 0 \leq k_t \leq 1 \) is transparency factor.
Outline

• Overview
• Vector operations
• Ray-casting framework
• Camera coordinate system
• Ray representation
• Ray-surface intersection
• Shading models
• Polygon rendering
  • Flat surface rendering
  • Gouraud surface rendering
  • Phong surface rendering
Polygon rendering

- How do we choose a color for each filled pixel if the object is represented in a set of polygonal meshes?
- It is called as “polygon shading” problem
- Simplest shading approach is to perform independent lighting calculation for every pixel

\[
I = I_a k_a + \sum_{i=1}^{m} I_{L_i} s_i \left( k_d \max(N \cdot L_i, 0) + k_s \left( \max(N \cdot H_i, 0) \right)^n \right)
\]
Polygon rendering

- Actually, we can take advantage of spatial coherence
  - Illumination calculations for pixels covered by the same primitive are related to each other
- Common used polygon algorithms, flat shading, Gouraud shading, and Phong shading
Flat surface rendering

- The most simplest one, also called as “constant intensity surface rendering”
- One illumination calculation per polygon
  - Assign all pixels inside each polygon the same color
  - No interpolation, pick a single representative intensity and propagate it over entire primitive
  - Loses almost all depth cues
- Objects look like they are composed of polygons
  - OK for polyhedral objects
  - Not so good for smooth surfaces
Flat surface rendering

- Some examples with flat surface rendering
Flat surface rendering

- Some examples with flat surface rendering
Flat surface rendering

- Some examples with flat surface rendering
Outline

- Overview
- Vector operations
- Ray-casting framework
- Camera coordinate system
- Ray representation
- Ray-surface intersection
- Shading models
- Polygon rendering
  - Flat surface rendering
  - Gouraud surface rendering
  - Phong surface rendering
Gouraud surface rendering

- Was proposed by Henri Gouraud in 1971


- Properties
  - produce continuous shading of surfaces represented by polygon meshes
  - It is often used to achieve continuous lighting on triangle surfaces by computing the lighting at the corners of each triangle and linearly interpolating the resulting colors for each pixel covered by the triangle.
Gouraud surface rendering

- Working principles
  - Step 1: estimate the normal of each vertex by averaging the surface normals of the polygons that meet at each vertex

\[ \overrightarrow{N_v} = \frac{\sum_{k=1}^{n} \overrightarrow{N_k}}{\sum_{k=1}^{n} \left| \overrightarrow{N_k} \right|} \]
Gouraud surface rendering

- Working principles
  - Step 1: estimate the normal of each vertex by averaging the surface normals of the polygons that meet at each vertex
  - Step 2: compute the color for each vertex based on a shading model (e.g., Phong model)
  - Step 3: for each screen pixel that is covered by the polygonal mesh, color intensities can then be interpolated from the color values calculated at the vertices (more on next page)
Gouraud surface rendering

- Working principles
  - Bilinear interpolation
    - At first, interpolate illumination along polygon edges \((I_a, I_b)\)
    - Then, interpolate illumination along scan lines \((I_p)\)

\[
I_a = I_1 \frac{y_s - y_2}{y_1 - y_2} + I_2 \frac{y_1 - y_s}{y_1 - y_2} \\
I_b = I_1 \frac{y_s - y_3}{y_1 - y_3} + I_3 \frac{y_1 - y_s}{y_1 - y_3} \\
I_p = I_a \frac{x_b - x_p}{x_b - x_a} + I_b \frac{x_p - x_a}{x_b - x_a}
\]
Gouraud surface rendering

- **Working principles**
  - For interpolation of triangle, we can use barycentric coordinates, still remember? Very convenient and straightforward!! (Page 87 of this slide)

Special case: ray-triangle intersection

- Barycentric interpolation
  - Values $\nu_1, \nu_2, \nu_3$ defined at $a, b, c$
    - They can be colors, normals, texture coordinates etc
  - $P(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c$ is the point
  - Then, $\nu(\alpha, \beta, \gamma) = \alpha \nu_1 + \beta \nu_2 + \gamma \nu_3$ is the barycentric interpolation of $\nu_1 \sim \nu_3$ at point $P$
    - i.e., once you know $\alpha, \beta, \gamma$, you can interpolate values using the same weights, very convenient
Gouraud surface rendering

- Cons
  - Gouraud shading can miss specular highlights because it interpolates vertex colors instead of calculating intensity directly at each point, or interpolating vertex normals
Gouraud surface rendering

- Some examples with Gouraud surface rendering

Flat (left) vs Gouraud (right)
Gouraud surface rendering

- Some examples with Gouraud surface rendering

Flat (left) vs Gouraud (right)
Gouraud surface rendering

- Some examples with Gouraud surface rendering

Flat (left) vs Gouraud (right)
Gouraud surface rendering

- Some examples with Gouraud surface rendering

Flat (left) vs Gouraud (right)
Outline

- Overview
- Vector operations
- Ray-casting framework
- Camera coordinate system
- Ray representation
- Ray-surface intersection
- Shading models
- Polygon rendering
  - Flat surface rendering
  - Gouraud surface rendering
  - Phong surface rendering
Phong surface rendering

- Phong surface rendering refers to an interpolation technique for surface rendering in 3D computer graphics
- It is also called Phong interpolation or normal-vector interpolation surface rendering
- Specifically, it interpolates surface normals across rasterized polygons and computes pixel colors based on the interpolated normals and a reflection model
Phong surface rendering

- Phong model: normal vector interpolation
  - Similar interpolation process as Gouraud shading; however, it interpolates $N$ rather than $I$
  - Always captures specular highlights, but computationally expensive
    - At each pixel, $N$ is recomputed and normalized
    - Then $I$ is computed at each pixel (lighting model is more expensive than interpolation algorithms)
    - This is now implemented in hardware, very fast
Phong surface rendering

- Some examples with Phong surface rendering

Flat (left), Gouraud (middle) and Phong shading models
Some examples with Phong surface rendering

Flat (left), Gouraud (middle) and Phong shading models
Phong surface rendering

- Some examples with Phong surface rendering

Flat (left), Gouraud (middle) and Phong shading models
Thanks for your attention