MIT EECS 6.837 Computer Graphics

Ray Casting II

Courtesy of Henrik Wann Jensen. Used with permission.

MIT EECS 6.837 – Matusik
• 3 ways to pass arguments to a function
  – by value, e.g. float f(float x)
  – by reference, e.g. float f(float &x)
    • f can modify the value of x
  – by pointer, e.g. float f(float *x)
    • x here is just a memory address
    • motivations:
      less memory than a full data structure if x has a complex type
      dirty hacks (pointer arithmetic), but just do not do it
    • clean languages do not use pointers
    • kind of redundant with reference
    • arrays are pointers
Pointers

• Can get it from a variable using &
  – often a BAD idea. see next slide

• Can be dereferenced with *
  – float *px=new float; // px is a memory address to a float
  – *px=5.0; //modify the value at the address px

• Should be instantiated with new. See next slide
Pointers, Heap, Stack

- Two ways to create objects
  - The BAD way, on the stack
    - myObject *f() {
      - myObject x;
      - ...
      - return &x
    }
    - will crash because x is defined only locally and the memory gets de-allocated when you leave function f
  - The GOOD way, on the heap
    - myObject *f() {
      - myObject *x=new myObject;
      - ...
      - return x
    }
    - but then you will probably eventually need to delete it
Segmentation Fault

• When you read or, worse, write at an invalid address

• Easiest segmentation fault:
  – float *px; // px is a memory address to a float
  – *px=5.0; // modify the value at the address px
  – Not 100% guaranteed, but you haven’t instantiated px, it could have any random memory address.

• 2nd easiest seg fault
  – Vector<float> vx(3);
  – vx[9]=0;
Segmentation Fault

- TERRIBLE thing about segfault: the program does not necessarily crash where you caused the problem
- You might write at an address that is inappropriate but that exists
- You corrupt data or code at that location
- Next time you get there, crash

- When a segmentation fault occurs, always look for pointer or array operations before the crash, but not necessarily at the crash
Debugging

• Display as much information as you can
  – image maps (e.g. per-pixel depth, normal)
  – OpenGL 3D display (e.g. vectors, etc.)
  – cerr<< or cout<< (with intermediate values, a message when you hit a given if statement, etc.)

• Doubt everything
  – Yes, you are sure this part of the code works, but test it nonetheless

• Use simple cases
  – e.g. plane z=0, ray with direction (1, 0, 0)
  – and display all intermediate computation
Questions?
Thursday Recap

• Intro to rendering
  – Producing a picture based on scene description
  – Main variants: Ray casting/tracing vs. rasterization
  – Ray casting vs. ray tracing (secondary rays)

• Ray Casting basics
  – Camera definitions
    • Orthographic, perspective
  – Ray representation
    • \( P(t) = \text{origin} + t \times \text{direction} \)
  – Ray generation
  – Ray/plane intersection
  – Ray-sphere intersection
Questions?
Ray-Triangle Intersection

- Use ray-plane intersection followed by in-triangle test
- Or try to be smarter
  - Use barycentric coordinates
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
  \[ P(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c, \]
  with \(\alpha + \beta + \gamma = 1\)

[Moebius, 1827]
Barycentric Coordinates

- Since $\alpha + \beta + \gamma = 1$, we can write $\alpha = 1 - \beta - \gamma$

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

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

$$= a + \beta(b - a) + \gamma(c - a)$$
Barycentric Definition of a Plane

- $P(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c$
  with $\alpha + \beta + \gamma = 1$
- Is it explicit or implicit?

Fun to know:
P is the barycenter, the single point upon which the triangle would balance if weights of size $\alpha$, $\beta$, & $\gamma$ are placed on points $a$, $b$ & $c$. 
Barycentric Definition of a Triangle

- \( P(\alpha,\beta,\gamma) = \alpha a + \beta b + \gamma c \)
  with \( \alpha + \beta + \gamma = 1 \) parameterizes the entire plane
Barycentric Definition of a Triangle

- \( P(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c \)
  with \( \alpha + \beta + \gamma = 1 \) parameterizes the entire plane

- If we require in addition that \( \alpha, \beta, \gamma \geq 0 \), we get just the triangle!
  - Note that with \( \alpha + \beta + \gamma = 1 \) this implies
    \[ 0 \leq \alpha \leq 1 \quad \& \quad 0 \leq \beta \leq 1 \quad \& \quad 0 \leq \gamma \leq 1 \]
  - Verify:
    - \( \alpha = 0 \) \( \Rightarrow \) \( P \) lies on line \( b-c \)
    - \( \alpha, \beta = 0 \) \( \Rightarrow \) \( P = c \)
    - etc.
Barycentric Definition of a Triangle

\[
P(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c
\]

• Condition to be barycentric coordinates:
  \[\alpha + \beta + \gamma = 1\]

• Condition to be inside the triangle:
  \[\alpha, \beta, \gamma \geq 0\]
How Do We Compute $\alpha$, $\beta$, $\gamma$?

- Ratio of opposite sub-triangle area to total area
  \[ \alpha = \frac{A_a}{A}, \quad \beta = \frac{A_b}{A}, \quad \gamma = \frac{A_c}{A} \]
- Use signed areas for points outside the triangle

![Diagram of a triangle with points A, a, b, P, and sub-triangle areas A_a, A_b, A_c marked.](image-url)
How Do We Compute $\alpha, \beta, \gamma$?

- Or write it as a $2 \times 2$ linear system
- $P(\beta, \gamma) = a + \beta e_1 + \gamma e_2$
  
  $e_1 = (b-a), \ e_2 = (c-a)$

This should be zero
How Do We Compute $\alpha$, $\beta$, $\gamma$?

- Or write it as a $2 \times 2$ linear system
- \[ \mathbf{P}(\beta, \gamma) = a + \beta \mathbf{e}_1 + \gamma \mathbf{e}_2 \]
  \[ \mathbf{e}_1 = (b-a), \quad \mathbf{e}_2 = (c-a) \]

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

This should be zero

Something’s wrong... This is a linear system of 3 equations and 2 unknowns!
How Do We Compute $\alpha, \beta, \gamma$?

- Or write it as a $2 \times 2$ linear system
- $P(\beta, \gamma) = a + \beta e_1 + \gamma e_2$
  
  $e_1 = (b - a), e_2 = (c - a)$

  $\langle e_1, a + \beta e_1 + \gamma e_2 - P \rangle = 0$

  $\langle e_2, a + \beta e_1 + \gamma e_2 - P \rangle = 0$

  These should be zero

Ha! We’ll take inner products of this equation with $e_1 \& e_2$
How Do We Compute $\alpha, \beta, \gamma$?

• Or write it as a $2\times2$ linear system

\[ \mathbf{P}(\beta, \gamma) = \mathbf{a} + \beta \mathbf{e}_1 + \gamma \mathbf{e}_2 \]

\[ \mathbf{e}_1 = (\mathbf{b} - \mathbf{a}), \quad \mathbf{e}_2 = (\mathbf{c} - \mathbf{a}) \]

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

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

\[ \begin{pmatrix} \langle \mathbf{e}_1, \mathbf{e}_1 \rangle & \langle \mathbf{e}_1, \mathbf{e}_2 \rangle \\ \langle \mathbf{e}_2, \mathbf{e}_1 \rangle & \langle \mathbf{e}_2, \mathbf{e}_2 \rangle \end{pmatrix} \begin{pmatrix} \beta \\ \gamma \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} \]

where

\[ \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} \langle (\mathbf{P} - \mathbf{a}), \mathbf{e}_1 \rangle \\ \langle (\mathbf{P} - \mathbf{a}), \mathbf{e}_2 \rangle \end{pmatrix} \]

and $\langle \mathbf{a}, \mathbf{b} \rangle$ is the dot product.
How Do We Compute $\alpha, \beta, \gamma$?

- Or write it as a $2 \times 2$ linear system

$$\mathbf{P}(\beta, \gamma) = \mathbf{a} + \beta \mathbf{e}_1 + \gamma \mathbf{e}_2$$

$$\mathbf{e}_1 = (\mathbf{b} - \mathbf{a}), \mathbf{e}_2 = (\mathbf{c} - \mathbf{a})$$

\[
\begin{align*}
\langle \mathbf{e}_1, \mathbf{a} + \beta \mathbf{e}_1 + \gamma \mathbf{e}_2 - \mathbf{P} \rangle &= 0 \\
\langle \mathbf{e}_2, \mathbf{a} + \beta \mathbf{e}_1 + \gamma \mathbf{e}_2 - \mathbf{P} \rangle &= 0
\end{align*}
\]

$$\begin{pmatrix}
\langle \mathbf{e}_1, \mathbf{e}_1 \rangle & \langle \mathbf{e}_1, \mathbf{e}_2 \rangle \\
\langle \mathbf{e}_2, \mathbf{e}_1 \rangle & \langle \mathbf{e}_2, \mathbf{e}_2 \rangle
\end{pmatrix}
\begin{pmatrix}
\beta \\
\gamma
\end{pmatrix} =
\begin{pmatrix}
c_1 \\
c_2
\end{pmatrix}$$

where

$$\begin{pmatrix}
c_1 \\
c_2
\end{pmatrix} = \begin{pmatrix}
\langle (\mathbf{P} - \mathbf{a}), \mathbf{e}_1 \rangle \\
\langle (\mathbf{P} - \mathbf{a}), \mathbf{e}_2 \rangle
\end{pmatrix}$$

and $\langle \mathbf{a}, \mathbf{b} \rangle$ is the dot product.
Intersection with Barycentric Triangle

- Again, set ray equation equal to barycentric equation
  \[ P(t) = P(\beta, \gamma) \]
  \[ R_o + t \cdot R_d = a + \beta(b-a) + \gamma(c-a) \]
- Intersection if \( \beta + \gamma \leq 1 \) \& \( \beta \geq 0 \) \& \( \gamma \geq 0 \)
  \( \text{(and } t > t_{\text{min}} \cdots ) \)
Intersection with Barycentric Triangle

- \( \mathbf{R}_o + t \mathbf{R}_d = \mathbf{a} + \beta (\mathbf{b} - \mathbf{a}) + \gamma (\mathbf{c} - \mathbf{a}) \)

\[
\begin{align*}
\mathbf{R}_{ox} + t\mathbf{R}_{dx} &= a_x + \beta (b_x - a_x) + \gamma (c_x - a_x) \\
\mathbf{R}_{oy} + t\mathbf{R}_{dy} &= a_y + \beta (b_y - a_y) + \gamma (c_y - a_y) \\
\mathbf{R}_{oz} + t\mathbf{R}_{dz} &= a_z + \beta (b_z - a_z) + \gamma (c_z - a_z)
\end{align*}
\]

- Regroup & write in matrix form \( \mathbf{A}\mathbf{x} = \mathbf{b} \)

\[
\begin{bmatrix}
a_x - b_x & a_x - c_x & R_{dx} \\
a_y - b_y & a_y - c_y & R_{dy} \\
a_z - b_z & a_z - c_z & R_{dz}
\end{bmatrix}
\begin{bmatrix}
\beta \\
\gamma \\
t
\end{bmatrix}
= \begin{bmatrix}
a_x - R_{ox} \\
a_y - R_{oy} \\
a_z - R_{oz}
\end{bmatrix}
\]
Cramer’s Rule

- Used to solve for one variable at a time in system of equations

\[
\beta = \frac{\begin{vmatrix} a_x - R_{ox} & a_x - c_x & R_{dx} \\ a_y - R_{oy} & a_y - c_y & R_{dy} \\ a_z - R_{oz} & a_z - c_z & R_{dz} \end{vmatrix}}{|A|}
\]

\[
\gamma = \frac{\begin{vmatrix} a_x - b_x & a_x - R_{ox} & R_{dx} \\ a_y - b_y & a_y - R_{oy} & R_{dy} \\ a_z - b_z & a_z - R_{oz} & R_{dz} \end{vmatrix}}{|A|}
\]

\[
t = \frac{\begin{vmatrix} a_x - b_x & a_x - c_x & a_x - R_{ox} \\ a_y - b_y & a_y - c_y & a_y - R_{oy} \\ a_z - b_z & a_z - c_z & a_z - R_{oz} \end{vmatrix}}{|A|}
\]

\[
| \text{denotes the determinant}
\]

Can be copied mechanically into code
Barycentric Intersection Pros

- Efficient
- Stores no plane equation
- Get the barycentric coordinates for free
  - Useful for interpolation, texture mapping
Barycentric Interpolation

- Values $v_1, v_2, v_3$ defined at $a, b, c$
  - Colors, normal, texture coordinates, etc.
- $P(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c$ is the point...
- $v(\alpha, \beta, \gamma) = \alpha v_1 + \beta v_2 + \gamma v_3$ is the barycentric interpolation of $v_1, v_2, v_3$ at point $P$
  - Sanity check: $v(1,0,0) = v_1$, etc.
- I.e., once you know $\alpha, \beta, \gamma$ you can interpolate values using the same weights.
  - Convenient!
Questions?

- Image computed using the RADIANCE system by Greg Ward
Ray Casting: Object Oriented Design

For every pixel

Construct a ray from the eye

For every object in the scene

Find intersection with the ray

Keep if closest
Object-Oriented Design

- We want to be able to add primitives easily
  - Inheritance and virtual methods
- Even the scene is derived from Object3D!

Object3D

bool intersect(Ray, Hit, \( t_{\text{min}} \))

- Plane
  - bool intersect(Ray, Hit, \( t_{\text{min}} \))
- Sphere
  - bool intersect(Ray, Hit, \( t_{\text{min}} \))
- Triangle Mesh
  - bool intersect(Ray, Hit, \( t_{\text{min}} \))
- Group
  - bool intersect(Ray, Hit, \( t_{\text{min}} \))

- Also cameras are abstracted (perspective/ortho)
  - Methods for generating rays for given image coordinates
Assignment 4 & 5: Ray Casting/Tracing

• Write a basic ray caster
  – Orthographic and perspective cameras
  – Spheres and triangles
  – 2 Display modes: color and distance

• We provide classes for
  – Ray: origin, direction
  – Hit: t, Material, (normal)
  – Scene Parsing

• You write ray generation, hit testing, simple shading
Books

- Peter Shirley et al.: *Fundamentals of Computer Graphics*
  AK Peters

- Ray Tracing
  - Jensen
  - Shirley
  - Glassner

Remember the ones at books24x7 mentioned in the beginning!

Images of three book covers have been removed due to copyright restrictions. Please see the following books for more details:
- Shirley P. and R.K. Morley, *Realistic Ray Tracing*
Constructive Solid Geometry (CSG)

• A neat way to build complex objects from simple parts using Boolean operations
  – Very easy when ray tracing

• Remedy used this in the Max Payne games for modeling the environments
  – Not so easy when not ray tracing :)

© Rockstar Games. All rights reserved. This content is excluded from our Creative Commons license. For more information, see [http://ocw.mit.edu/help/faq-fair-use/](http://ocw.mit.edu/help/faq-fair-use/).
CSG Examples
Constructive Solid Geometry (CSG)

Given overlapping shapes A and B:

Union

Intersection

Subtraction

Should only “count” overlap region once!
How Can We Implement CSG?

4 cases

- Points on A, Outside of B
- Points on B, Inside of A
- Points on B, Outside of A
- Points on A, Inside of B

Union

Intersection

Subtraction
Collect Intersections

Each ray processed separately!
Implementing CSG

1. Test "inside" intersections:
   • Find intersections with A, test if they are inside/outside B
   • Find intersections with B, test if they are inside/outside A

This would certainly work, but would need to determine if points are inside solids...
Implementing CSG

1. Test "inside" intersections:
   • Find intersections with A, test if they are inside/outside B
   • Find intersections with B, test if they are inside/outside A

2. Overlapping intervals:
   • Find the intervals of "inside" along the ray for A and B
   • How? Just keep an “entry” / “exit” bit for each intersection
     • Easy to determine from intersection normal and ray direction
   • Compute union/intersection/subtraction of the intervals
Implementing CSG

Problem reduces to 1D for each ray

2. Overlapping intervals:
   - Find the intervals of "inside" along the ray for A and B
   - How? Just keep an “entry” / “exit” bit for each intersection
     - Easy to determine from intersection normal and ray direction
   - Compute union/intersection/subtraction of the intervals
CSG is Easy with Ray Casting...

- ...but **very hard** if you actually try to compute an explicit representation of the resulting surface as a triangle mesh
- In principle very simple, 
  
  *but floating point numbers are not exact*
  
  - E.g., points do not lie exactly on planes...
  - Computing the intersection A vs B is not necessarily the same as B vs A...
  - The line that results from intersecting two planes does not necessarily lie on either plane...
  - etc., etc.
What is a Visual Hull?
Why Use a Visual Hull?

- Can be computed robustly
- Can be computed efficiently
Rendering Visual Hulls
CSG then Ray Casting

Reference 1

Desired

Reference 2
CSG then Ray Casting

Reference 1  Desired  Reference 2
CSG then Ray Casting
CSG then Ray Casting
CSG then Ray Casting

Reference 1

Desired

Reference 2
Ray Casting then Intersection

- Reference 1
- Desired
- Reference 2
Ray Casting then Intersection

Reference 1  Desired  Reference 2
Ray Casting then Intersection
Ray Casting then Intersection

Reference 1  Desired  Reference 2
Ray Casting then Intersection
Ray Casting then Intersection
Ray Casting then Intersection
Ray Casting then Intersection

Reference 1

Desired

Reference 2
Image Based (2D) Intersection
Image Based Visual Hulls
Questions?
Precision

• What happens when
  – Ray Origin lies on an object?
  – Grazing rays?

• Problem with floating-point approximation
The Evil $\varepsilon$

- In ray tracing, do NOT report intersection for rays starting on surfaces
  - Secondary rays start on surfaces
  - Requires epsilons
  - Best to nudge the starting point off the surface e.g., along normal
The Evil $\varepsilon$

- Edges in triangle meshes
  - Must report intersection (otherwise not watertight)
  - Hard to get right
Questions?

Image by Henrik Wann Jensen

Courtesy of Henrik Wann Jensen. Used with permission.
Transformations and Ray Casting

• We have seen that transformations such as affine transforms are useful for modeling & animation
• How do we incorporate them into ray casting?
1. Make each primitive handle any applied transformations and produce a camera space description of its geometry

```
Transform {
    Translate { 1 0.5 0 }
    Scale { 2 2 2 }
    Sphere {
        center 0 0 0
        radius 1
    }
}
```

2. ...Or Transform the Rays
Primitives Handle Transforms

Sphere {
    center 3 2 0
    z_rotation 30
    r_major 2
    r_minor 1
}

- Complicated for many primitives
Transform Ray

• Move the ray from *World Space* to *Object Space*

\[ p_{WS} = M \quad p_{OS} \]

\[ p_{OS} = M^{-1} \quad p_{WS} \]
Transform Ray

- New origin:
  \[ \text{origin}_{OS} = M^{-1} \text{origin}_{WS} \]

- New direction:
  \[ \text{direction}_{OS} = M^{-1} (\text{origin}_{WS} + 1 \ast \text{direction}_{WS}) - M^{-1} \text{origin}_{WS} \]
  \[ \text{direction}_{OS} = M^{-1} \text{direction}_{WS} \]

Note that the w component of direction is 0

World Space

Object Space
What About $t$?

• If $M$ includes scaling, $direction_{OS}$ ends up NOT be normalized after transformation

• Two solutions
  – Normalize the direction
  – Do not normalize the direction
1. Normalize Direction

- $t_{OS} \neq t_{WS}$ and must be rescaled after intersection
  $\Rightarrow$ One more possible failure case...
2. Do Not Normalize Direction

- \( t_{OS} = t_{WS} \) \( \rightarrow \) convenient!
- But you should not rely on \( t_{OS} \) being true distance in intersection routines (e.g. \( a \neq 1 \) in ray-sphere test)
Transforming Points & Directions

• Transform point

\[
\begin{bmatrix}
x' \\
y' \\
z' \\
1
\end{bmatrix}
=
\begin{bmatrix}
a & b & c & d \\
e & f & g & h \\
i & j & k & l \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix}
=
\begin{bmatrix}
ax+by+cz+d \\
ex+fy+gz+h \\
ix+jy+kz+l \\
1
\end{bmatrix}
\]

• Transform direction

\[
\begin{bmatrix}
x' \\
y' \\
z' \\
0
\end{bmatrix}
=
\begin{bmatrix}
a & b & c & d \\
e & f & g & h \\
i & j & k & l \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
0
\end{bmatrix}
=
\begin{bmatrix}
ax+by+cz \\
ex+fy+gz \\
ix+jy+kz \\
0
\end{bmatrix}
\]

Homogeneous Coordinates:
\((x,y,z,w)\)

\(w = 0\) is a point at infinity (direction)

• If you do not store \(w\) you need different routines to apply \(M\) to a point and to a direction ==> Store everything in 4D!
Recap: How to Transform Normals?

Object Space

World Space

World Space Normal: $n_{WS}$

Object Space Normal: $n_{OS}$
Transformation for Shear and Scale

Incorrect Normal Transformation

Correct Normal Transformation
So How Do We Do It Right?

- Think about transforming the tangent plane to the normal, not the normal vector.

Pick any vector \( \mathbf{v}_{OS} \) in the tangent plane, how is it transformed by matrix \( \mathbf{M} \)?

\[
\mathbf{v}_{WS} = \mathbf{M} \mathbf{v}_{OS}
\]
Transform Tangent Vector \( \mathbf{v} \)

\( \mathbf{v} \) is perpendicular to normal \( \mathbf{n} \):

\[
\begin{align*}
\mathbf{n}_{OS}^T \mathbf{v}_{OS} &= 0 \\
\mathbf{n}_{OS}^T \left( \mathbf{M}^{-1} \mathbf{M} \right) \mathbf{v}_{OS} &= 0 \\
\left( \mathbf{n}_{OS}^T \mathbf{M}^{-1} \right) \left( \mathbf{M} \mathbf{v}_{OS} \right) &= 0 \\
\left( \mathbf{n}_{OS}^T \mathbf{M}^{-1} \right) \mathbf{v}_{WS} &= 0
\end{align*}
\]

\( \mathbf{v}_{WS} \) is perpendicular to normal \( \mathbf{n}_{WS} \):

\[
\begin{align*}
\mathbf{n}_{WS}^T \mathbf{v}_{WS} &= 0 \\
\mathbf{n}_{WS}^T &= \mathbf{n}_{OS}^T \left( \mathbf{M}^{-1} \right) \\
\mathbf{n}_{WS} &= \left( \mathbf{M}^{-1} \right)^T \mathbf{n}_{OS}
\end{align*}
\]
Position, Direction, Normal

- **Position**
  - transformed by the full homogeneous matrix $\mathbf{M}$

- **Direction**
  - transformed by $\mathbf{M}$ except the translation component

- **Normal**
  - transformed by $\mathbf{M}^{-T}$, no translation component
That’s All for Today!

• Further reading
  – Realistic Ray Tracing, 2nd ed.
    (Shirley, Morley)