Ray tracing and geometric representations
Course announcements

• Programming assignment 1 will be posted on Friday 1/27 and will be due two weeks later.

• Take-home quiz 1 will be posted on Tuesday 1/24 and will be due a week later.

• Office hours for this week only (will finalize starting next week based on survey results):
  - Yannis–Thursday 3-4 pm, Smith Hall (EDSH) 225.
Course announcements

• Is anyone not on Piazza?
  
  https://piazza.com/class/lctj7gng8wql4/

• Is anyone not on Canvas?
  
  https://canvas.cmu.edu/courses/33678

• Is anyone not on Slack?
Hanyu Chen

15-468 TA, Senior in Computer Science & Math

- Research interest broadly in computer vision/computer graphics/rendering
- Currently working with Yannis in neural rendering & surface reconstruction related research
Jeff Tan (jefftan@andrew.cmu.edu)
15-468 TA and Senior in Computer Science

Research: Neural rendering for real-time dynamic 3D reconstruction
Advised by Prof. Deva Ramanan

Yang et al. (CVPR 2022)  
Tan et al. (in submission)
Overview of today’s lecture

• Introduction to ray tracing.
• Intersections with geometric primitives.
• Triangular meshes.
Most of these slides were directly adapted from:

- Wojciech Jarosz (Dartmouth).
Two forms of 3D rendering

Rasterization: object point to image plane
- start with a 3D object point
- apply transforms
- determine the 2D image plane point it projects to

Ray tracing: image plane to object point
- start with a 2D image point
- generate a ray
- determine the visible 3D object point

Inverse processes
Two forms of 3D rendering

**Rasterization**
- for (each triangle)
  - for (each pixel)
    - if (triangle covers pixel)
      - keep closest hit
  
  **Triangle-centric**

**Ray tracing**
- for (each pixel or ray)
  - for (each triangle)
    - if (ray hits triangle)
      - keep closest hit

  **Ray-centric**
Rasterization advantages

Modern scenes are more complicated than images
- A 1920x1080 frame (1080p) at 64-bit color and 32-bit depth per pixel is 24 MB (not that much)
  • of course, if we have more than one sample per pixel this gets larger, but e.g. 4x supersampling is still a relatively comfortable ~100 MB
- Our scenes are routinely larger than this
  • This wasn’t always true

A rasterization-based renderer can *stream* over the triangles, no need to keep entire dataset around
- Allows parallelism and optimizations of memory systems
Rasterization limitations

Restricted to scan-convertible primitives
- Pretty much: triangles

Faceting, shading artifacts
- This is largely going away with programmable per-pixel shading, though

No unified handling of shadows, reflection, transparency
Ray/path tracing

Advantages
- Generality: can render anything that can be intersected with a ray
- Easily allows recursion (shadows, reflections, etc.)

Disadvantages
- Hard to implement in hardware (lacks computation coherence, must fit entire scene in memory, bad memory behavior)
  • Not such a big point anymore given general purpose GPUs
- Has traditionally been too slow for interactive applications
- Both of the above are changing rather rapidly right now!
A ray-traced image
Ray tracing today
Rapid change in film industry

2008:
- Most CGI in films rendered using micro-polygon rasterization.
- “You’d be crazy to render a full-feature film with ray/path tracing.”
- Ray/path tracing mostly interesting to academics

2018:
- Most major films now rendered using ray/path tracing.
- “You’d be crazy not to render a full-feature film using path tracing.”
Albrecht Dürer (1525)
René Descartes (1650)
Isaac Newton (1670)
Appel (1968)

Ray casting
- Generate an image by sending one ray per pixel
- Check for shadows by sending a ray towards the light
Whitted (1979)

recursive ray tracing (reflection & refraction)
Light Transport - Assumptions

Geometric optics:
- no diffraction, no polarization, no interference

Light travels in a straight line in a vacuum
- no atmospheric scattering or refraction
- no gravity effects

Color can be represented as three numbers: (R,G,B)
Emission theory of vision

Eyes send out “feeling rays” into the world

Supported by:
- Ancient greeks
- 50% of US college students*

Ray Tracing - Overview

“light tracing”

eye point

image plane

light source
Basic Ray Tracing Pipeline
Basic Ray Tracing Pipeline

Ray Generation

Intersection
Basic Ray Tracing Pipeline

- Ray Generation
- Intersection
- Shading
Basic Ray Tracing Pipeline

Ray Generation → Intersection → Shading
Basic Ray Tracing Pipeline

- Ray Generation
- Intersection
- Shading
Basic Ray Tracing Pipeline

Ray Generation → Intersection → Shading
Basic Ray Tracing Pipeline

Ray Generation → Intersection → Shading
Ray Tracing Pseudocode

```plaintext
rayTraceImage()
{
    parse scene description

    for each pixel
        ray = generateCameraRay(pixel)
        pixelColor = trace(ray)
}
```
Ray Tracing Pseudocode

trace(ray)
{
    hit = find first intersection with scene objects

    color = shade(hit)

    return color
}

might trace more rays (recursive)
Ray Tracing Pseudocode

```
rayTraceImage()
{
    parse scene description

    for each pixel
        ray = generateCameraRay(pixel)

        pixelColor = trace(ray)

    }

what is a ray? how do we generate a camera ray?
```
**Ray: a half line**

Standard representation: origin (point) $\mathbf{0}$ and direction $\mathbf{d}$

$$\mathbf{r}(t) = \mathbf{0} + t\mathbf{d}$$

- this is a parametric equation for the line
- lets us directly generate the points on the line
- if we restrict to $t > 0$ then we have a ray
- note replacing $\mathbf{d}$ with $a\mathbf{d}$ does not change ray (for $a > 0$)
Generating eye rays

Orthographic

Perspective

view rect

pixel position

viewing ray

viewpoint

pixel position

viewing ray
Pinhole Camera (Camera Obscura)
Pinhole Camera

film / physical image plane

virtual image plane

viewing volume

pinhole
Pinhole Camera

virtual image plane

viewing volume

eye
Generating eye rays—perspective

Establish view rectangle in X–Y plane, specified by, e.g.
- l, r, t, b

Place rectangle at $z = -d$

$$s = [u, v, -d]^T$$

$$d = s$$

$$r(t) = o + td$$

Does distance $d$ matter?
Placing the camera in the scene
Generating eye rays—orthographic

How do you generate a ray for an orthographic camera?
Ray-Surface Intersections

Surface primitives
- spheres
- planes
- triangles
- general implicits
- etc.
Ray-Sphere Intersection

Algebraic approach:

- Condition 1: point is on ray: \( \mathbf{r}(t) = \mathbf{o} + t\mathbf{d} \)

- Condition 2: point is on sphere: \( \|\mathbf{x} - \mathbf{c}\|^2 - r^2 = 0 \)

- substitute and solve for \( t \):

\[ \|\mathbf{o} + t\mathbf{d} - \mathbf{c}\|^2 - r^2 = 0 \]
Ray-Sphere Intersection

substitute and solve for \( t \)

\[
\| \mathbf{o} + t \mathbf{d} - \mathbf{c} \|^2 - r^2 = 0 \quad \rightarrow \quad (\mathbf{o}_x + t \mathbf{d}_x - \mathbf{c}_x)^2 + (\mathbf{o}_y + t \mathbf{d}_y - \mathbf{c}_y)^2 + (\mathbf{o}_z + t \mathbf{d}_z - \mathbf{c}_z)^2 - r^2 = 0
\]

which reduces to: \( At^2 + Bt + C = 0 \)

Solve for \( t \) using quadratic equation:

\[
t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}
\]

What happens when square root is zero or negative?
Ray-Surface Intersections

Surface primitives
- spheres
- planes
- triangles
- general implicits
- etc.
Ray-Plane Intersection

Plane equation (implicit)

Algebraic form:

\[ ax + by + cz + d = 0 \]
Ray-Plane Intersection

Plane equation (implicit)

\[(x - p) \cdot n = 0\]

Point of interest \hspace{1cm} Point on plane \hspace{1cm} Plane normal

Substitute ray equation for \(x\) and solve for \(t\)

\[(o + td - p) \cdot n = 0\]

\[td \cdot n + (o - p) \cdot n = 0\]

\[t = - \frac{(o - p) \cdot n}{d \cdot n}\]
Ray-Surface Intersections

Surface primitives

- spheres
- planes
- triangles
- general implicits
- etc.
Ray-Triangle intersection

Condition 1: point is on ray: \( \mathbf{r}(t) = \mathbf{o} + td \)

Condition 2: point is on plane: \( (\mathbf{x} - \mathbf{p}) \cdot \mathbf{n} = 0 \)

Condition 3: point is on the inside of all three edges

First solve 1&2 (ray–plane intersection) for \( t \):

\[
(\mathbf{o} + td - \mathbf{p}) \cdot \mathbf{n} = 0
\]

\[
t = -\frac{(\mathbf{o} - \mathbf{p}) \cdot \mathbf{n}}{d \cdot \mathbf{n}}
\]

Several options for 3
Ray-Triangle intersection (Approach 1)

In plane, triangle is the intersection of 3 half spaces
Ray-Triangle intersection (Approach 1)

\[ \mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1) \]

Which way does \( \mathbf{n} \) point?
Ray-Triangle intersection (Approach 1)

\[ \mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1) \]

\[ \mathbf{n}_{x13} = (\mathbf{x} - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1) \]

Which way does \( \mathbf{n} \) point?

What about \( \mathbf{n}_{x13} \)?
Ray-Triangle intersection (Approach 1)

\[ \mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1) \]
\[ \mathbf{n}_{x13} = (\mathbf{x} - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1) \]

Which way does \( \mathbf{n} \) point?

What about \( \mathbf{n}_{x13} \)?

- How about now?
Ray-Triangle intersection (Approach 1)

\[ \mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1) \]

\[ \mathbf{n}_{x13} = (\mathbf{x} - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1) \]

Which way does \( \mathbf{n} \) point?

What about \( \mathbf{n}_{x13} \)?

- How about now?
- Edge test: \( (\mathbf{n}_{x13} \cdot \mathbf{n}) < 0 \)
Ray-Triangle intersection (Approach 1)

\[ n = (p_2 - p_1) \times (p_3 - p_1) \]
\[ n_{x13} = (x - p_1) \times (p_3 - p_1) \]

Which way does \( n \) point?

What about \( n_{x13} \)?

- How about now?

- Edge test: \( (n_{x13} \cdot n) < 0 \)
Ray-Triangle Intersection (Approach 2)

Intersect ray with triangle’s plane

Test whether hit-point is within triangle

- compute sub-triangle areas $\alpha$, $\beta$, $\gamma$

- test inside triangle conditions
Barycentric coordinates

Barycentric coordinates: \[ \mathbf{x}(\alpha, \beta, \gamma) = \alpha \mathbf{p}_1 + \beta \mathbf{p}_2 + \gamma \mathbf{p}_3 \]

Inside triangle conditions:

\[ \alpha + \beta + \gamma = 1 \quad 0 \leq \alpha \leq 1 \]
\[ \gamma = 1 - \alpha - \beta \quad 0 \leq \beta \leq 1 \]
\[ 0 \leq \gamma \leq 1 \]
Interpretations of barycentric coords

Sub-triangle areas

\[ \alpha = \frac{|\Delta p_2 p_3 x|}{|\Delta p_1 p_2 p_3|} \]
\[ \beta = \frac{|\Delta p_1 p_3 x|}{|\Delta p_1 p_2 p_3|} \]
\[ \gamma = \frac{|\Delta p_1 p_2 x|}{|\Delta p_1 p_2 p_3|} \]

\[ x = \alpha p_1 + \beta p_2 + \gamma p_3 \]
Ray-Triangle Intersection (Approach 3)

Insert ray equation: 
\[ \alpha p_1 + \beta p_2 + (1 - \alpha - \beta)p_3 = o + td \]

\[ \alpha(p_1 - p_3) + \beta(p_2 - p_3) + p_3 = o + td \]

\[ \alpha(p_1 - p_3) + \beta(p_2 - p_3) - td = o - p_3 \]

\[ \alpha a + \beta b - td = e \]

Solve directly

\[
\begin{bmatrix}
-d & a & b
\end{bmatrix}
\begin{bmatrix}
t \\
\alpha \\
\beta
\end{bmatrix} = e
\]

Can be much faster!
Ray-Surface Intersections

Other primitives

- cylinder
- cone, paraboloid, hyperboloid
- torus
- disk
- general polygons, meshes
- etc.
Intersecting transformed primitive?

Option 1: Transform the primitive
- simple for triangles, since they transform to triangles
- other primitives get more complicated (e.g. sphere $\rightarrow$ ellipsoid)

Option 2: Transform the ray (by the inverse transform)
- more elegant; works on any primitive
- allows simpler intersection tests
  (e.g., just use existing sphere-intersection routine)
Intersection and coordinate systems

World space

Local space
Intersection and coordinate systems

World space

Local space
Intersection and coordinate systems

We have a sphere now

But with a different ray
Transformations in homogeneous coords

A 3D transformation matrix:

\[
M = \begin{pmatrix}
  m_{11} & m_{12} & m_{13} & m_{14} \\
  m_{21} & m_{22} & m_{23} & m_{24} \\
  m_{31} & m_{32} & m_{33} & m_{34} \\
  m_{41} & m_{42} & m_{43} & m_{44}
\end{pmatrix}
\]

A 3D homogenous vector:

\[
v = \begin{pmatrix}
  x \\
  y \\
  z \\
  w
\end{pmatrix}
\]

A position has \( w \neq 0 \), and a direction has \( w = 0 \)
Transformations

Matrix-vector multiplication, $M\mathbf{v}$, transforms the vector

A translation matrix:

$$M_t = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

A scaling matrix:

$$M_s = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
Intersection and coordinate systems

Have a transform $M$, a ray $r(t)$, and a surface $S$

To intersect:
1. Transform ray to local coords (by inverse of $M$)
2. Call surface intersection
3. Transform hit data back to global coords (by $M$)

How to transform a ray $r(t) = p + td$ by $M^{-1}$?
- $r'(t) = M^{-1}p + tM^{-1}d$
- Remember: $p$ forms as a point, $d$ as a direction!
Ray-Surface Intersections

Other primitives

- cylinder
- cone, paraboloid, hyperboloid
- torus
- disk
- general polygons, meshes
- etc.
Image so far

With eye ray generation and sphere intersection

parse scene description

for each pixel:
  ray = camera.getRay(pixel);
  hit = s.intersect(ray, 0, +inf);
  if hit:
    image.set(pixel, white);
Intersecting many shapes

Intersect each primitive

Pick closest intersection
- Only within considered range $[t_{\text{min}}, t_{\text{max}}]$
- After each valid intersection, update $t_{\text{max}}$

Essentially a line search
Intersection against many shapes

The basic idea is:

```c
Surfaces::intersect(ray, tMin, tMax):
    tBest = +inf; firstHit = null;
    for s in surfaces:
        hit = s.intersect(ray, tMin, tBest);
        if hit:
            tBest = hit.t;
            firstHit = hit;
    return firstHit;
```

- this is linear in number of surfaces but there are sublinear methods (acceleration structures)
Image so far

With eye ray generation and scene intersection

for each pixel:
    ray = camera.getRay(pixel);
    c = scene.trace(ray, 0, +inf);
    image.set(pixel, c);

Scene::trace(ray, tMin, tMax):
    hit = surfaces.intersect(ray, tMin, tMax);
    if (hit)
        return hit.color();
    else
        return backgroundColor;
Ray-Surface Intersections

Other primitives

- cylinder
- cone, paraboloid, hyperboloid
- torus
- disk
- general polygons, meshes
- etc.
How should we represent complex geometry?

How are they obtained?
- modeled by hand
- scanned

What operations must we support?
- modeling/editing
- animating
- texturing
- rendering
# Surface representation zoo!

### Parametric
- Splines, tensor-product surfaces
- Subdivision surfaces

### Implicit
- Metaballs/blobs
- Distance fields
- Procedural, CSG
- Neural nets

### Discrete/Sampled
- Meshes
- Point set surfaces

---

After a slide by Olga Sorkine-Hornung
Polygonal Meshes

Boundary representations of objects

- Piecewise linear

After a slide by Olga Sorkine-Hornung
A small triangle mesh

12 triangles, 8 vertices
A large mesh

10 million triangles from a high-resolution 3D scan
After a slide by Steve Marschner
After a slide by Steve Marschner
spheres approximate sphere
Meshes as Approx. of Smooth Surfaces

Piecewise linear approximation

- Error is $O(h^2)$
Meshes as Approx. of Smooth Surfaces

Piecewise linear approximation

- Error is $O(h^2)$

After a slide by Olga Sorkine-Hornung.
Polygonal Meshes

Polygonal meshes are a good representation

- approximation $O(h^2)$
- arbitrary topology
- piecewise smooth surfaces
- adaptive refinement
- efficient rendering
Data Structures: What should be stored?

Geometry: 3D coordinates

Attributes
- Normal, color, texture coordinates
- Per vertex, face, edge

Connectivity
- Adjacency relationships
Separate Triangle List or Face Set (STL)

Face: 3 vertex positions

Storage:
- 4 Bytes/coordinate (using 32-bit floats)
- 36 Bytes/face

Wastes space
Indexed Face Set (OBJ, OFF, WRL)

**Vertex: position**

**Face: vertex indices**

**Storage:**
- 12 Bytes/vertex
- 12 Bytes/face

**Reduces wasted space**

**Even better with per-vertex attributes**
Data on meshes

Often need to store additional information besides just the geometry

Can store additional data at faces, vertices, or edges

Examples
- colors stored on faces, for faceted objects
- information about sharp creases stored at edges
- any quantity that varies continuously (without sudden changes, or discontinuities) gets stored at vertices
Key types of vertex data

Surface normals
- when a mesh is approximating a curved surface, store normals at vertices

Texture coordinates
- 2D coordinates that tell you how to paste images on the surface

Positions
- at some level this is just another piece of data
Defining normals

Face normals: same normal for all points in face
- geometrically correct, but faceted look
Problems with face normals

Piecewise planar approximation converges pretty quickly to the smooth geometry as the number of triangles increases
- error is $O(h^2)$

But the surface normals don’t converge so well
- normal is constant over each triangle, with discontinuous jumps across edges
- error is only $O(h)$
Problems with face normals—2D example

Approximating circle with increasingly many segments

Max error in position error drops by factor of 4 each step

Max error in normal only drops by factor of 2
Problems with face normals—solution

Piecewise planar approximation converges pretty quickly to the smooth geometry as the number of triangles increases
- for mathematicians: error is $O(h^2)$

But the surface normals don’t converge so well
- normal is constant over each triangle, with discontinuous jumps across edges
- for mathematicians: error is only $O(h)$

Better: store the “real” normal at each vertex, and interpolate to get normals that vary gradually across triangles
Defining normals

Vertex normals: store normal at vertices, interpolate in face
- geometrically “inconsistent”, but smooth look
Barycentric coordinates

Barycentric interpolation: \( p(\alpha, \beta, \gamma) = \alpha p_1 + \beta p_2 + \gamma p_3 \)

Can use this eqn. to interpolate any vertex quantity across triangle!
Barycentric coordinates

Barycentric interpolation:

\[ p(\alpha, \beta, \gamma) = \alpha p_1 + \beta p_2 + \gamma p_3 \]

\[ c(\alpha, \beta, \gamma) = \alpha c_1 + \beta c_2 + \gamma c_3 \]

Can use this eqn. to interpolate any vertex quantity across triangle!
Barycentric coordinates

Barycentric interpolation:

\[
\mathbf{p}(\alpha, \beta, \gamma) = \alpha \mathbf{p}_1 + \beta \mathbf{p}_2 + \gamma \mathbf{p}_3
\]

\[
\mathbf{c}(\alpha, \beta, \gamma) = \alpha \mathbf{c}_1 + \beta \mathbf{c}_2 + \gamma \mathbf{c}_3
\]

\[
\mathbf{n}(\alpha, \beta, \gamma) = \alpha \mathbf{n}_1 + \beta \mathbf{n}_2 + \gamma \mathbf{n}_3
\]

not guaranteed to be unit length

Can use this eqn. to interpolate any vertex quantity across triangle!
Realism through geometric complexity
Ray Tracing Acceleration

Ray-surface intersection is at the core of every ray tracing algorithm

Brute force approach:
- intersect every ray with every primitive
- many unnecessary ray-surface intersection tests
Ray Tracing Cost

“the time required to compute the intersections of rays and surfaces is over 95 percent” [Whitted 1980]

Cost = $O(n_x \cdot n_y \cdot n_o)$
- (number of pixels) $\cdot$ (number of objects)
- Assumes 1 ray per pixel

Example: 1024 x 1024 image of a scene with 1000 triangles
- Cost is (at least) $10^9$ ray-triangle intersections

Typically measured per ray:
- Naive: $O(n_o)$ - linear with number of objects
$O(n_o)$ Ray Tracing (The Problem)

8 primitives $\rightarrow$ 3 seconds

50K trees each with 1M polygons = 50B polygons
$\rightarrow$ 594 years!
Sub-linear Ray Tracing

50K trees each with 1M polygons = 50B polygons → **11 minutes**

300,000,000x speedup!

Andreas Byström
The solution

Improve efficiency of ray-surface intersections by constructing acceleration structures.

- A spatial organization of objects in a scene to minimize the necessary number of ray-object intersection tests.

Spatial sorting/subdivision (e.g. grid, kd-tree, ochre)

- Decompose space into disjoint regions & assign objects to regions

Object sorting/subdivision (bounding volume hierarchy)

- Decompose objects into disjoint sets & bound using simple volumes for fast rejection
Bounding Volumes

Spheres
Bounding Volumes

Axis-aligned bounding boxes (most common)
Bounding Volumes Hierarchies

Now do this hierarchically!
BVH Traversal

```c++
void BVHNode::intersectBVH(ray, &hit):
    if (bound.hit(ray)):
        if (leaf):
            leaf.intersect(ray, hit);
        else:
            leftChild.intersectBVH(ray, hit);
            rightChild.intersectBVH(ray, hit); 
```
Constructing BVHs

Top-down:
- partition objects along an axis and create two sub-sets

Bottom-up:
- recursively group nearby objects together
Divisive (top-down) BBH construction

1. Choose split axis
2. Choose split plane location
3. Choose whether to create leaf or split + repeat

Many strategies for each of these steps
Choosing axis based on centroid extents

PBRe2 fig. 4.8
Object-median splitting

1. Sort bbox centroids along split axis
2. Take first half as left child, second half as right