Global Illumination

CPSC 314

Ray-Tracing

Basic Algorithm (Whithead):

for every pixel \( p_i \) {
    Generate ray \( r \) from camera position through pixel \( p_i \)
    for every object \( o \) in scene {
        if( \( r \) intersects \( o \) )
            Compute lighting at intersection point, using local
            normal and material properties; store result in \( p_i \)
        else
            \( p_i = \) background color
    }
}
Ray-Tracking

**Issues:**
- Generation of rays
- Intersection of rays with geometric primitives
- Geometric transformations
- Lighting and shading
- Efficient data structures so we don’t have to test intersection with every object

Ray Intersections

**Spheres at origin:**
- Implicit function:
  \[ S(x, y, z) : x^2 + y^2 + z^2 = r^2 \]
- Ray equation:
  \[
  R_{i,j}(t) = C + t \cdot v_{i,j} = \begin{pmatrix} c_x \\ c_y \\ c_z \end{pmatrix} + t \cdot \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = \begin{pmatrix} c_x + t \cdot v_x \\ c_y + t \cdot v_y \\ c_z + t \cdot v_z \end{pmatrix}
  \]
Ray Intersections

To determine intersection:

• Insert ray $\mathbf{R}_{ij}(t)$ into $S(x,y,z)$:

\[
(c_x + t \cdot v_x)^2 + (c_y + t \cdot v_y)^2 + (c_z + t \cdot v_z)^2 = r^2
\]

• Solve for $t$ (find roots)
  – Simple quadratic equation

Ray Intersections

Other Primitives:

• Implicit functions:
  – Spheres at arbitrary positions
    ▪ Same thing
  – Conic sections (hyperboloids, ellipsoids, paraboloids, cones, cylinders)
    ▪ Same thing (all are quadratic functions!)
  – Higher order functions (e.g. tori and other quartic functions)
    ▪ In principle the same
    ▪ But root-finding difficult
    ▪ Net to resolve to numerical methods
Ray-Tracing
Shadows

**Approach:**
- To test whether point is in shadow, send out *shadow rays* to all light sources
  - *If ray hits another object, the point lies in shadow*

Ray-Tracing
Reflections/Refractions

**Approach:**
- Send rays out in reflected and refracted direction to gather incoming light
- That light is multiplied by local surface color and Fresnel term, and added to result of local shading
Recursive Ray-Tracing

Recursive Ray-Tracing Algorithm

\[ \text{RayTrace}(r, \text{scene}) \]
\[
\text{obj} := \text{FirstIntersection}(r, \text{scene})
\]
\[
\text{if (no obj)} \text{ return BackgroundColor;}
\]
\[
\text{else begin}
\]
\[
\text{if ( Reflect(obj) ) then}
\]
\[
\text{reflect_color} := \text{RayTrace(ReflectRay}(r, \text{obj}));
\]
\[
\text{else}
\]
\[
\text{reflect_color} := \text{Black;}
\]
\[
\text{if ( Transparent(obj) ) then}
\]
\[
\text{refract_color} := \text{RayTrace(RefractionRay}(r, \text{obj}));
\]
\[
\text{else}
\]
\[
\text{refract_color} := \text{Black;}
\]
\[
\text{return Shade(reflect_color, refract_color, obj);}
\]
\[
\text{end;}
\]

Whitted, 1980
Algorithm Termination Criteria

**Termination criteria**

- No intersection
- Reach maximal depth
  - *Number of bounces*
- Contribution of secondary ray attenuated below threshold
  - *Each reflection/refraction attenuates ray*

Ray-Tracing Terminology

**Terminology:**

- Primary ray: ray starting at camera
- Shadow ray
- Reflected/refracted ray
- Ray tree: all rays directly or indirectly spawned off by a single primary ray

**Note:**

- Need to limit maximum depth of ray tree to ensure termination of ray-tracing process!
Ray Tracing

Data Structures

- Goal: reduce number of intersection tests per ray
- Lots of different approaches:
  - (Hierarchical) bounding volumes
  - Hierarchical space subdivision
    - Oct-tree, k-D tree, BSP tree

Bounding Volumes

Idea:

- Rather than testing every ray against a potentially very complex object (e.g. triangle mesh), do a quick conservative test first which eliminates most of the rays
  - Surround complex object by very simple, easy to test geometry (typically sphere or axis-aligned box)
  - Want to make bounding volume as tight as possible!
Hierarchical Bounding Volumes

*Extension of previous idea:*
- Use bounding volumes for groups of objects

Regular Grid

*Subdivide space into rectangular grid:*
- Associate every object with the cell(s) that it overlaps with
- Find intersection: traverse grid

In 3D: regular grid of cubes (voxels):
Area Light Sources

So far:
- All lights were either point-shaped or directional
  - Both for ray-tracing and the rendering pipeline
- Thus, at every point, we only need to compute lighting formula and shadowing for ONE light direction

In reality:
- All lights have a finite area
- Instead of just dealing with one direction, we now have to integrate over all directions that go to the light source

Area Light Sources

Area lights produce soft shadows:

Area light

Occluding surface

Receiving surface

Umbra (core shadow)

Penumbra (partial shadow)
Area Light Sources

**Point lights:**
- Only one light direction:
  \[ I_{\text{reflected}} = \rho \cdot V \cdot I_{\text{light}} \]
- \( V \) is visibility of light (0 or 1)
- \( \rho \) is lighting model (e.g. diffuse or Phong)

![Point light diagram]

Area Light Sources

**Area Lights:**
- Infinitely many light rays
- Need to integrate over all of them:
  \[ I_{\text{reflected}} = \int_{\omega_{\text{light}}} \rho(\omega) \cdot V(\omega) \cdot I_{\text{light}}(\omega) \cdot d\omega \]
- Lighting model visibility and light intensity can now be different for every ray!

![Area light diagram]
Integrating over Light Source

**Rewrite the integration**

- Instead of integrating over directions

\[ I_{\text{reflected}} = \int_{\text{light directions}} \rho(\omega) \cdot V(\omega) \cdot I_{\text{light}}(\omega) \cdot d\omega \]

we can integrate over points on the light source

\[ I_{\text{reflected}}(q) = \int_{s,t} \frac{\rho(p-q) \cdot V(p-q) \cdot I_{\text{light}}(p)}{|p-q|^2} \cdot ds \cdot dt \]

where \( q \): point on reflecting surface, \( p = F(s,t) \) is a point on the area light
- We are integrating over \( p \)
- Denominator: quadratic falloff!

Integration

**Problem:**

- Except for the simplest of scenes, either integral is **not solvable analytically**!
- This is mostly due to the visibility term, which could be arbitrarily complex depending on the scene

**So:**

- Use numerical integration
- Effectively: approximate the light with a whole number of point lights
**Numerical Integration**

*Regular grid of point lights*

- Problem: will see 4 hard shadows rather than as soft shadow
- Need LOTS of points to avoid this problem

**Monte Carlo Integration**

*Better:*

- Randomly choose the points
- Use different points on light for computing the lighting in different points on reflecting surface
  - This produces random noise
  - Visually preferable to structured artifacts
Monte Carlo Integration

Formally:

- Approximate integral with finite sum
  \[
  I_{\text{reflected}}(q) = \int_{s,t} \rho(p-q) \cdot V(p-q) \cdot I_{\text{light}}(p) \cdot ds \cdot dt \\
  \approx \frac{A}{N} \sum_{i=1}^{N} \rho(p_i-q) \cdot V(p_i-q) \cdot I_{\text{light}}(p_i)
  \]

  where
  - The \( p_i \) are randomly chosen on the light source
    - With equal probability!
  - \( A \) is the total area of the light
  - \( N \) is the number of samples (rays)
Sampling

Sample directions vs. sample light source

- Most directions do not correspond to points on the light source
  - Thus, variance will be higher than sampling light directly

Images by Matt Pharr

Global Illumination

So far:

- Have considered only light directly coming from the light sources
  - As well as mirror reflections, refraction

In reality:

- Light bouncing off diffuse and/or glossy surfaces also illuminates other surfaces
  - This is called global illumination
Rendering Equation

**Equation guiding global illumination:**

\[ L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{\Omega} \rho(x, \omega_i, \omega_0)L_i(\omega_i)d\omega_i \]

\[ = L_e(x, \omega_o) + \int_{\Omega} \rho(x, \omega_i, \omega_0)L_i(R(x, \omega_i),-\omega_i)d\omega_i \]

**Where**

- \( \rho \) is the reflectance from \( \omega_i \) to \( \omega_o \) at point \( x \)
- \( L_e \) is the outgoing (i.e. reflected) *radiance* at point \( x \) in direction \( \omega_i \)
  - *Radiance* is a specific physical quantity describing the amount of light along a ray
  - Radiance is constant along a ray
- \( L_o \) is the emitted radiance (=0 unless point \( x \) is on a light source)
- \( R \) is the “ray-tracing function”. It describes what point is visible from \( x \) in direction \( \omega_i \)

---

**Note:**

- The rendering equation is an **integral equation**
- This equation cannot be solved directly
  - Ray-tracing function is complicated!
  - Similar to the problem we had computing illumination from area light sources!
Ray Casting

- Cast a ray from the eye through each pixel
- The following few slides are from Fred Durand (MIT)

Ray Tracing

- Cast a ray from the eye through each pixel
- Trace secondary rays (light, reflection, refraction)
Monte Carlo Ray Tracing

- Cast a ray from the eye through each pixel
- Cast random rays from the visible point
  - Accumulate radiance contribution

Monte Carlo Ray Tracing

- Cast a ray from the eye through each pixel
- Cast random rays from the visible point
- Recurse
Monte Carlo

- Cast a ray from the eye through each pixel
- Cast random rays from the visible point
- Recurse

Monte Carlo

- Systematically sample primary light
Monte Carlo Path Tracing

**In practice:**
- Do not branch at every intersection point
  - This would have exponential complexity in the ray depth!
- Instead:
  - Shoot some number of primary rays through the pixel (10s-1000s, depending on scene!)
  - For each pixel and each intersection point, make a **single, random** decision in which direction to go next

Monte Carlo Path Tracing

- Trace only one secondary ray per recursion
- But send many primary rays per pixel
- (performs antialiasing as well)
How to Sample?

**Simple sampling strategy:**
- At every point, choose between all possible reflection directions with equal probability
- This will produce very high variance/noise if the materials are specular or glossy
- Lots of rays are required to reduce noise!

**Better strategy: importance sampling**
- Focus rays in areas where most of the reflected light contribution will be found
- For example: if the surface is a mirror, then only light from the mirror direction will contribute!
- Glossy materials: prefer rays near the mirror direction

---

How to Sample?

- Images by Veach & Guibas

Naive sampling strategy

Multiple importance sampling
How to Sample?

Sampling strategies are still an active research area!

- Recent years have seen drastic advances in performance
- Lots of excellent sampling strategies have been developed in statistics and machine learning
  - Many are useful for graphics

More on Global Illumination

This was a (very) quick overview

- More details in CPSC 514 (Computer Graphics: Rendering)
- Not taught next year, but the year after
Curves & Surfaces

Motivation

Geometric representations so far:
- Discrete geometry
  - Triangles, line segments
  - Rendering pipeline, ray-tracing
- Specific objects
  - Spheres
  - Ray-tracing

Want more general representations:
- Flexible like triangles
- But smooth!
Curves & Surfaces as Parametric Functions

Curves & surfaces in arbitrary dimensions

- Curves:
  \[ x = F(t); F : \mathbb{R} \mapsto \mathbb{R}^d \]
- Surfaces:
  \[ x = F(s,t); F : \mathbb{R}^2 \mapsto \mathbb{R}^d \]

In practice:

- Restrict to specific class of functions
  - e.g. polynomials of certain degree

\[ x = \sum_{i=0}^{m} b_i t^i \]

In 2D:

\[ \begin{pmatrix} x \\ y \end{pmatrix} = \sum_{i=0}^{m} \begin{pmatrix} b_{x,i} \\ b_{y,i} \end{pmatrix} t^i \]

Polynomial Curves

Advantages:

- Computationally easy to handle
  - \( b_0 \ldots b_m \) uniquely describe curve (finite storage, easy to represent)

Disadvantages:

- Not all shapes representable
  - Partially fix with piecewise functions (splines)
- Still not very intuitive
  - Fix: represent polynomials in different basis
  - For example: Bernstein polynomials
  - This is what is called a Bézier curve
**Polynomial Bases**

**Reminder**
- The set of all polynomials of degree \( \leq m \) over \( \mathbb{R} \) forms a vector space with the common polynomial operations
  - *What are those operations?*
  - *Dimension of this space is \( m+1 \)*
- One common basis for this space are the monomials
  \[ \{1, x, x^2, \ldots, x^m \} \]
- Problem: the relationship between this basis and a geometric shape is quite unintuitive
- Thus: use another!

**Other Bases for Polynomials**

**Example: Lagrange Polynomials**
- Given: \( m+1 \) parameter values \( t_0 \ldots t_m \)
- Define
  \[ L_i^m(t) := \prod_{j=0 \ldots m, j \neq i} \frac{t-t_j}{t_i-t_j}; i = 0 \ldots m \]
- Clear from definition:
  - All \( L_i^m \) are polynomials of degree \( m \)
  - \( L_i^m(t_j) = \begin{cases} 1; & i = j \\ 0; & \text{else} \end{cases} \)
  - *In particular, all \( L_i^m \) are linearly independent!*
Other Bases for Polynomials

**Lagrange Polynomials (cont):**

- The $L_i^m$ are linearly independent and there are $m+1$ of them, therefore they are a basis for the polynomials of degree up to $m$.
- Therefore can write any of polynomial of degree up to $m$ as

$$F(t) = \sum_{i=0}^{m} L_i^m(t_j) \cdot b_i$$

- In addition, we have for all $i$: $F(t_i) = b_i$
  - *In other words, the polynomial interpolates the points $(t_i, b_i)$*

---

**Example**

- Basis function

---
Lagrange Polynomials

Example:
- Interpolation (explicit function)

Note:
- Same works in parametric setting
- The coefficients then become points to be interpolated!

Other Bases for Polynomials

Bernstein Polynomials

\[ B_i^m(t) := \binom{m}{i} t^i (1-t)^{m-i}, \quad i = 0..m; t \in [0,1] \]

- Graph for degree m=1:
Bernstein Polynomials

- Graph for m=2:

- Graph for m=3:

Bernstein Polynomials

\[ B_i^m(t) := \binom{m}{i} t^i (1-t)^{m-i}; \quad i = 0..m; t \in [0,1] \]

Properties:

- \( B_i^m(t) \) is a polynomial of degree m
- \( B_i^m(t) \geq 0 \) for \( t \in [0,1] \); \( B_0^m(0) = 1; B_i^m(0) = 0 \) for \( i \neq 0 \)
- \( B_i^m(t) = B_{m-i}^m(1-t) \)
- \( B_i^m(t) \) has exactly one maximum in the interval 0..1. It is at \( t=i/m \) (proof: compute derivative…)
- W/o proof: all \((m+1)\) functions \( B_i^m \) are linearly independent
  - **Thus they form a basis for all polynomials of degree \( \leq m \)**
Bernstein Polynomials

*More properties*

- \[ \sum_{i=0}^{m} B_i^m(t) = (t + (1 - t))^m = 1 \]

- \[ B_i^m(t) = t \cdot B_{i-1}^{m-1}(t) + (1 - t) \cdot B_i^{m-1}(t) \]

- Both are quite important a fast evaluation algorithm of Bézier curves (de Casteljau algorithm)

Bézier Curves

*Definition:*

- A Bézier curve is a polynomial curve that uses the Bernstein polynomials as a basis

\[ F(t) = \sum_{i=0}^{m} b_i B_i^m(t) \]

- The \( b_i \) are called control points of the Bézier curve
- The control polygon is obtained by connecting the control points with line segments

*Advantage of Bézier curves:*

- The control points and control polygon have clear geometric meaning and are intuitive to use
Properties of Bézier Curves
(Pierre Bézier, Renault, about 1960)

**Easy to see:**
- The endpoints $b_0$ and $b_m$ of the control polygon are interpolated and the corresponding parameter values are $t=0$ and $t=1$

**More properties:**
- The Bézier curve is tangential to the control polygon in the endpoints
- The curve completely lies within the convex hull of the control points
- The curve is affine invariant
- There is a fast, recursive evaluation algorithm

Bézier Curve Properties

$$F(t) = \sum_{i=0}^{m} b_i B_i^m(t)$$

**Recall:**
- Bernstein polynomials have values between 0 and 1 for $t\in[0,1]$, and
  $$\sum_{i=0}^{m} B_i^m(t) = 1$$
  - Therefore: every point on Bézier curve is convex combination of control points
  - Therefore: Bézier curve lies completely within convex hull of control points
De Casteljau Algorithm

Also recall:
• Recursive formula for Bernstein polynomials:
  \[ B_i^m(t) = t \cdot B_{i-1}^{m-1}(t) + (1 - t) \cdot B_i^{m-1}(t) \]

Plug into Bézier curve definition:
\[
F(t) = \sum_{i=0}^{m} b_i \left( t \cdot B_{i-1}^{m-1}(t) + (1 - t) \cdot B_i^{m-1}(t) \right) \\
= t \cdot \sum_{i=1}^{m} b_i B_{i-1}^{m-1}(t) + (1 - t) \cdot \sum_{i=0}^{m-1} b_i B_i^{m-1}(t)
\]

De Casteljau Algorithm

Consequence:
• Every point \( F(t_0) \) on a Bézier curve of degree \( m \) is the convex combination of two points \( G(t_0) \) and \( H(t_0) \) that lie on Bézier curves of degree \( m-1 \).
• The control points of \( G(t) \) are the first \( m \) control points of \( F(t) \)
• The control points of \( H(t) \) are the last \( m \) control points of \( F(t) \)
De Casteljau Algorithm

**Recursion:**

- Every point on a Bézier curve can be generated through successive convex combinations of the degree 0 Bézier curves.
- Degree 0 Bézier curves are the control points!

\[ F(t) = \sum_{i=0}^{0} b_i B_i^0(t) = b_i \cdot 1 \equiv b_i \]

---

**De Casteljau Algorithm**

*After working out the math we get:*

\[ F(t) = b_0^m(t) \text{; where} \]

\[ b_i^0(t) := b_i(t); \quad i = 0 \ldots m \]

\[ b_i^l(t) := (1-t) \cdot b_{i-1}^l(t) + t \cdot b_{i+1}^l(t) \]
De Casteljau Algorithm

Graphical Interpretation:

- Determine point \( F(1/2) \) for the cubic Bézier curve given by the following four points:

```
\begin{align*}
&b_0, b_1, b_2, b_3 \\
&b_0^1, b_1^1, b_2^1, b_3^1 \\
&b_0^2, b_1^2, b_2^2, b_3^2 \\
&b_0^3 = F(1/2)
\end{align*}
```

De Casteljau Algorithm

Evaluation scheme (cubic case):

```
\begin{align*}
&b_0^0, b_1^0, b_2^0, b_3^0 \\
&b_0^1, b_1^1, b_2^1, b_3^1 \\
&b_0^2, b_1^2, b_2^2, b_3^2 \\
&b_0^3, b_1^3, b_2^3, b_3^3
\end{align*}
```
Tensor Product Surfaces

What about surfaces?

- Use basis functions as in the case of curves
- Apply them independently to the parametric directions s and t
- Works for arbitrary basis

Example:

- Bézier curve: \( F(t) = \sum_{i=0}^{m} B_i^m(t) \cdot b_i \)

- Tensor product Bézier patch:
  \[
  F(s,t) = \sum_{i=0}^{m_i} \sum_{j=0}^{m_j} B_i^{m_i}(s) \cdot B_j^{m_j}(t) \cdot b_{i,j}
  \]
Tensor Product Surfaces

Notes:

• The surface is polynomial in $s$ and $t$, depending on basis
  - The degree in $s$ is $m_i$
  - The degree in $t$ is $m_i$
  - The total degree is $m_s + m_t$

• The algorithms from the curves transfer directly to tensor product surfaces
• The properties of these surfaces are directly related to the properties of the corresponding curves

Tensor Product Surfaces

Properties:

• Convex hull
• Affine invariance
• The control points of the edge curves are the boundary points of the control mesh
• A Bézier patch interpolates the corner vertices of its control mesh
More on Curves & Surfaces

This was a (very) quick overview
- More details in CPSC 424 (Geometric Modeling)
- Taught by Alla Sheffer in term 2 next year

Upcoming Lectures

Wednesday:
- Research topics in graphics
- Tour of graphics labs

Friday:
- Final