Non-rigid Body Transformations

Transformations
Implicit Functions
Barycentric Coordinates

Texture Mapping
Spatial Data Structures
Raytracing
Radiosity

Photon Mapping
DiffEQ
Implicit Integration
Cloth
Transformations
Uses of Transformations

• Modeling
  – build complex models by positioning simple components
  – transform from object coordinates to world coordinates

• Viewing
  – placing the virtual camera in the world
  – specifying transformation from world coordinates to camera coordinates

• Animation
  – vary transformations over time to create motion
Rigid Body Transformations

(a) Rotation angle and line about which to rotate

(b) Rotation angle and line about which to rotate
Non-rigid Body Transformations

Distance between points on object do not remain constant
Basic 2D Transformations

Scale

Shear

Rotate

chalkboard
Composition of Transformations

- Created by stringing basic ones together, e.g.
  - "translate \( p \) to the origin, rotate, then translate back"
  
  can also be described as a rotation about \( p \)
- Any sequence of linear transformations can be collapsed into a single matrix formed by multiplying the individual matrices together
- Order matters!
- Can apply a whole sequence of transformations at once
3D Transformations

- 3-D transformations are very similar to the 2-D case
- Scale
- Shear
- Rotation is a bit more complicated in 3-D
  - different rotation axes
Fixed Euler Angles for 3-D Rotations

- Independent rotations about each coordinate axis
  - angle interpolation for animation generates bizarre motions
  - rotations are order-dependent, and there are no conventions about the order to use

- Widely used anyway, because they're “simple”
Euler Angles for 3-D Rotations

Roll
Pitch
Yaw
But what about translation?

• Translation is not linear--how to represent as a matrix?
But what about translation?

• Translation is not linear--how to represent as a matrix?
  
  • Trick: add extra coordinate to each vector
  • This extra coordinate is the *homogeneous* coordinate, or \( w \)
  • When extra coordinate is used, vector is said to be represented in *homogeneous coordinates*
  • We call these matrices *Homogeneous Transformations*
W? Where did that come from?

• Practical answer:
  – W is a clever algebraic trick.
  – Don’t worry about it too much. The w value will be 1.0 for the time being (until we get to perspective viewing transformations)

• More complete answer:
  – (x,y,w) coordinates form a 3D projective space.
  – All nonzero scalar multiples of (x,y,1) form an equivalence class of points that project to the same 2D Cartesian point (x,y).
  – For 3-D graphics, the 4D projective space point (x,y,z,w) maps to the 3D point (x,y,z) in the same way.
Homogeneous 2D Transformations

The basic 2D transformations become

*Translate:*
\[
\begin{bmatrix}
1 & 0 & t_x \\
0 & 1 & t_y \\
0 & 0 & 1
\end{bmatrix}
\]

*Scale:*
\[
\begin{bmatrix}
s_x & 0 & 0 \\
0 & s_y & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

*Rotate:*
\[
\begin{bmatrix}
\cos \theta & -\sin \theta & 0 \\
\sin \theta & \cos \theta & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

Now *any* sequence of translate/scale/rotate operations can be combined into a single homogeneous matrix by multiplication.

3D transforms are modified similarly.
Rigid Body Transformations

Rotation angle and line about which to rotate
Rigid Body Transformations

• A transformation matrix of the form

\[
\begin{bmatrix}
x_x & x_y & t_x \\
y_x & y_y & t_y \\
0 & 0 & 1
\end{bmatrix}
\]

where the upper 2x2 submatrix is a rotation matrix and column 3 is a translation vector, is a rigid body transformation.

• Any series of rotations and translations results in a rotation and translation of this form (and no change in the distance between vertices)
Sequences of Transformations

Often the same transformations are applied to many points.

Calculation time for the matrices and combination is negligible compared to that of transforming the points.

Reduce the sequence to a single matrix, then transform.
Collapsing a Chain of Matrices.

• Consider the composite function ABCD, i.e. \( p' = ABCDp \)
• Matrix multiplication isn’t commutative - the order is important
• But matrix multiplication is associative, so can calculate from right to left or left to right: \( ABCD = (((AB) C) D) = (A (B (CD))) \).
• Iteratively replace either the leading or the trailing pair by its product

\[
\begin{align*}
M & \leftarrow D \\
M & \leftarrow CM \\
M & \leftarrow BM \\
M & \leftarrow AM \\
\end{align*}
\begin{align*}
M & \leftarrow A \\
M & \leftarrow MB \\
M & \leftarrow MC \\
M & \leftarrow MD \\
\end{align*}
\]

Premultiply or Postmultiply both give the same result.
What is a Normal? – refresher

Indication of outward facing direction for lighting and shading

Order of definition of vertices in OpenGL

Right hand rule
Transforming Normals

• It’s tempting to think of normal vectors as being like porcupine quills, so they would transform like points.
• Alas, it’s not so.
• We need a different rule to transform normals.
Examples

• Modeling with primitive shapes
Implicit equations

- *Implicit equations* are a way to define curves and surfaces.
- In 2D, a curve can be defined by
  \[ f(x,y) = 0 \]
  for some scalar function \( f \) of \( x \) and \( y \).
- In 3D, a surface can be defined by
  \[ f(x,y,z) = 0 \]
  for some scalar function \( f \) of \( x \), \( y \), and \( z \).
Implicit equations

• The function $f$ evaluates to 0 at every point on the curve or surface, and it evaluates to a non-zero real number at all other points.
• Multiplying $f$ by a non-zero coefficient preserves this property, so we can rewrite $f(x,y) = 0$ as $kf(x,y) = 0$ for any non-zero $k$.
• The implied curve is unaffected.
$f(x,y)$
Implicit equations

Chalkboard examples:
• Implicit 2D circle
• Implicit 2D line
• Implicit 3D plane
Implicit equations

- We call these equations “implicit” because although they imply a curve or surface, they cannot explicitly generate the points that comprise it.
- In order to generate points, we need another form...
Parametric equations

- *Parametric equations* offer the capability to generate continuous curves and surfaces.
- For curves, parametric equations take the form
  \[ x = f(t) \quad y = g(t) \quad z = h(t) \]
- For 3D surfaces, we have
  \[ x = f(s,t) \quad y = g(s,t) \quad z = h(s,t) \]
Parametric equations

- The *parameters* for these equations are scalars that range over a continuous (possibly infinite) interval.
- Varying the parameters over their entire intervals smoothly generates every point on the curve or surface.
Implicit equations

Chalkboard examples:
- Parametric 3D line
- Parametric sphere
Barycentric Coordinates
Why barycentric coordinates?

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

- The simplest way to do this interpolation is *barycentric coordinates*.
- The name comes from the Greek word *barus* (heavy) because the coordinates are weights assigned to the vertices.

**Goal:** Assign a every point $(x,y)$ or $(x,y,z)$ to barycentric coordinates $(\alpha,\beta,\gamma)$. 
Barycentric Coordinates

Solution: \( \vec{x} = \alpha \vec{a} + \beta \vec{b} + \gamma \vec{c} \)
What are barycentric coordinates?

Some cool properties:

- Point \( p \) is inside the triangle if and only if
  
  \[ 0 < \alpha < 1, \]
  
  \[ 0 < \beta < 1, \]
  
  \[ 0 < \gamma < 1 \]

- If one component is zero, \( p \) is on an edge.
- If two components are zero, \( p \) is on a vertex.
- The coordinates can be used as weighting factors for properties of the vertices, like color.
Barycentric Color Interpolation

If: \( \vec{x} = \alpha \vec{a} + \beta \vec{b} + \gamma \vec{c} \)

Then: \( \text{color}(\vec{x}) = \alpha \text{color}(\vec{a}) + \beta \text{color}(\vec{b}) + \gamma \text{color}(\vec{c}) \)
Barycentric coordinates

Chalkboard examples:

- Conversion from 2D Cartesian
- Conversion from 3D Cartesian
Viewing and Projection

• Our eyes collapse 3-D world to 2-D retinal image (brain then has to reconstruct 3D)
• In CG, this process occurs by projection
• Projection has two parts:
  – Viewing transformations: camera position and direction
  – Perspective/orthographic transformation: reduces 3-D to 2-D
• Use homogeneous transformations (of course…)

Computer Graphics 15-462

Thursday, April 29, 2010
Getting Geometry on the Screen

Given geometry positioned in the world coordinate system, how do we get it onto the display?

- Transform to camera coordinate system
- Transform (warp) into canonical view volume
- Clip
- Project to display coordinates
- Rasterize

Perspective and Orthographic Projection
Orthographic Projection
Viewing and Projection

Build this up in stages

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

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

good model for telephoto lens. No perspective effects.

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

front orthographic view
Telephoto Lenses and Fashion Photography
Canonical View Volume

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

X=I  left plane
X=R  right plane
Y=B  bottom plane
Y=T  top plane
Z=N  near plane
Z=F  far plane

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

Eye position: e
Gaze direction: g
view-up vector: t
324. The *radio astronomico* used to measure the width of a façade from Gemma’s Frisius’s *De Radio astronomico*. , Antwerp. 1545.

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

Paint on the window whatever color you see there.

Simple Perspective Camera

Canonical case:
- camera looks along the z-axis
- focal point is the origin
- image plane is parallel to the xy-plane at distance $d$
  - (We call $d$ the focal length, mainly for historical reasons)
Perspective Projection of a Point

\[ y_s = \frac{d}{z} y \]
History of Perspective Projection


Clipping

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

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

Solution: transform frustum to a cube before clipping

Converts perspective frustum to orthographic frustum
Yet another homogeneous transform!
Warping a perspective projection into and orthographic one
Lines for the two projections intersect at the view plane
How can we put this in matrix form?
   Need to divide by z—haven’t seen a divide in our matrices so far…
Requires our w from last time (or h in the book)
Texture Mapping
Texture Mapping

• A way of adding surface details

• Two ways can achieve the goal:
  – Model the surface with more polygons
    » Slows down rendering speed
    » Hard to model fine features
Texture Mapping

• A way of adding surface details

• Two ways can achieve the goal:
  – Model the surface with more polygons
    » Slows down rendering speed
    » Hard to model fine features
  – Map a texture to the surface
    » This lecture
    » Image complexity does not affect complexity of processing
The texture

- Texture is a bitmap image
- 2D array: texture[height][width][4]
- Pixels of the texture called *texels*
- Texel coordinates (s,t) scaled to [0,1] range
Map textures to surfaces

The polygon can have arbitrary size and shape

7
The drawing itself

• Use GLTexCoord2f(s,t) to specify texture coordinates

• Example:

```c
glEnable(GL_TEXTURE_2D)
glBegin(GL_QUADS);
glTexCoord2f(0.0, 0.0); glVertex3f(0.0, 0.0, 0.0);
glTexCoord2f(0.0, 1.0); glVertex3f(2.0, 1.0, 0.0);
glTexCoord2f(1.0, 0.0); glVertex3f(10.0, 0.0, 0.0);
glTexCoord2f(1.0, 1.0); glVertex3f(12.0, 1.0, 0.0);
glEnd();
glDisable(GL_TEXTURE_2D)
```

• State machine: Texture coordinates remain valid until you change them or exit texture mode via

```c
glDisable (GL_TEXTURE_2D)
```
Color blending

• Final pixel color = f (texture color, object color)

• How to determine the color of the final pixel?
  – GL_REPLACE – use texture color to replace object color
  – GL_BLEND – linear combination of texture and object color
  – GL_MODULATE – multiply texture and object color

• Example:
  – glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
What happens if texture coordinates outside [0,1]?

- **Two choices:**
  - Repeat pattern (GL_REPEAT)
  - Clamp to maximum/minimum value (GL_CLAMP)

- **Example:**
  - `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP)`
  - `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP)`
What happens if texture coordinates outside [0,1]?

clamp

repeat

```
glTexCoord2f(0.0, 0.0); glVertex3f(0.0, 0.0, 0.0);
glTexCoord2f(0.0, 3.0); glVertex3f(0.0, 10.0, 0.0);
glTexCoord2f(3.0, 0.0); glVertex3f(10.0, 0.0, 0.0);
glTexCoord2f(3.0, 3.0); glVertex3f(10.0, 10.0, 0.0);
```
Texture Value Lookup

- For given texture coordinates \((s,t)\), we can find a unique image value, corresponding to the texture image at that location.

Texture (5x5):

\[(0,0) (0.25,0) (0.5,0) (0.75,0) (1,0)\]

3D geometry

\[P(x,y,z)\]
Interpolating colors

- Some \((s,t)\) coordinates not directly at pixel in the texture, but in between
Interpolating colors

- Solutions:
  - Nearest neighbor
    » Use the nearest neighbor to determine color
    » Faster, but worse quality
    » `glTexImage2D(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
  - Linear interpolation
    » Incorporate colors of several neighbors to determine color
    » Slower, better quality
    » `glTexImage2D(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR)`
Other solutions

• Signal processing.
• What is wrong with linear interpolation...

Antialiasing...
Texture Levels

Original Texture

Pre-Filtered Images

1/4

1/16

1/64

etc.

1 pixel

Thursday, April 29, 2010
Spatial Data Structures

Given two bounding boxes at one level of the hierarchy, how do you compute the boxes for the next level?

How about for bounding spheres?

Thursday, April 29, 2010
Speeding Up Computations
Speeding Up Computations

• Ray Tracing
  – Spend a lot of time doing ray object intersection tests
Speeding Up Computations

• Ray Tracing
  – Spend a lot of time doing ray object intersection tests

• Hidden Surface Removal – painters algorithm
  – Sorting polygons front to back
Speeding Up Computations

• Ray Tracing
  – Spend a lot of time doing ray object intersection tests

• Hidden Surface Removal – painters algorithm
  – Sorting polygons front to back

• Collision between objects
  – Quickly determine if two objects collide

\[ n^2 \text{ computations} \]
Speeding Up Computations

• Ray Tracing
  – Spend a lot of time doing ray object intersection tests

• Hidden Surface Removal – painters algorithm
  – Sorting polygons front to back

• Collision between objects
  – Quickly determine if two objects collide

Spatial data-structures

\( n^2 \) computations
Spatial Data Structures

• We’ll look at
  – Hierarchical bounding volumes
  – Grids
  – Octrees
  – K-d trees and BSP trees

• Good data structures can give speed up ray tracing by 10x or 100x
Bounding Volumes

• Wrap things that are hard to check for intersection in things that are easy to check
  – Example: wrap a complicated polygonal mesh in a box
  – Ray can’t hit the real object unless it hits the box
  – Adds some overhead, but generally pays for itself.
Bounding Volumes

• Wrap things that are hard to check for intersection in things that are easy to check
  – Example: wrap a complicated polygonal mesh in a box
  – Ray can’t hit the real object unless it hits the box
  – Adds some overhead, but generally pays for itself.

• Most common bounding volume types: sphere and box
  – box can be axis-aligned or not
Bounding Volumes

• Wrap things that are hard to check for intersection in things that are easy to check
  – Example: wrap a complicated polygonal mesh in a box
  – Ray can’t hit the real object unless it hits the box
  – Adds some overhead, but generally pays for itself.

• Most common bounding volume types: sphere and box
  – box can be axis-aligned or not

• You want a snug fit!
• But you don’t want expensive intersection tests!
Bounding Volumes

• You want a snug fit!
• But you don’t want expensive intersection tests!
• Use the ratio of the object volume to the enclosed volume as a measure of fit.

• Cost = n*B + m*I

  n - is the number of rays tested against the bounding volume
  B - is the cost of each test  (Do not need to compute exact intersection!)
  m - is the number of rays which actually hit the bounding volume
  I - is the cost of intersecting the object within
Bounding Volumes

- You want a snug fit!
- But you don’t want expensive intersection tests!
- Use the ratio of the object volume to the enclosed volume as a measure of fit.

- Cost = n*B + m*I

  n - is the number of rays tested against the bounding volume
  B - is the cost of each test (Do not need to compute exact intersection!)
  m - is the number of rays which actually hit the bounding volume
  I - is the cost of intersecting the object within
Hierarchical Bounding Volumes

• Still need to check ray against every object  --- O(n)
• Use tree data structure
  – Larger bounding volumes contain smaller ones
Hierarchical Bounding Volumes

• Still need to check ray against every object  --- O(n)
• Use tree data structure
  – Larger bounding volumes contain smaller ones
Hierarchical Bounding Volumes

- Still need to check ray against every object --- O(n)
- Use tree data structure
  - Larger bounding volumes contain smaller ones
Hierarchical Bounding Volumes

- Still need to check ray against every object  --- $O(n)$
- Use tree data structure
  - Larger bounding volumes contain smaller ones
Hierarchical Bounding Volumes

- Still need to check ray against every object  --- O(n)
- Use tree data structure
  - Larger bounding volumes contain smaller ones

Check intersect root
If not return no intersections
Hierarchical Bounding Volumes

- Still need to check ray against every object --- $O(n)$
- Use tree data structure
  - Larger bounding volumes contain smaller ones

Check intersect root
If intersect
  check intersect left sub-tree
  check intersect right sub-tree
Hierarchical Bounding Volumes

• Still need to check ray against every object --- O(n)
• Use tree data structure
  – Larger bounding volumes contain smaller ones

Check intersect root
If intersect
  check intersect left sub-tree
  check intersect right sub-tree
Hierarchical Bounding Volumes

• Still need to check ray against every object — O(n)
• Use tree data structure
  – Larger bounding volumes contain smaller ones

Check intersect root
If intersect
  check intersect left sub-tree
  check intersect right sub-tree
Hierarchical Bounding Volumes

- Still need to check ray against every object --- $O(n)$
- Use tree data structure
  - Larger bounding volumes contain smaller ones

Check intersect root
If intersect
  check intersect left sub-tree
  check intersect right sub-tree
Hierarchical Bounding Volumes

• Many ways to build a tree for the hierarchy
• Works well:
  – Binary
  – Roughly balanced
  – Boxes of sibling trees not overlap too much
Hierarchical Bounding Volumes

- Sort the surfaces along the axis before dividing into two boxes
- Carefully choose axis each time
- Choose axis that minimizes sum of volumes
Hierarchical Bounding Volumes

- Sort the surfaces along the axis before dividing into two boxes
- Carefully choose axis each time
- Choose axis that minimizes sum of volumes
Hierarchical Bounding Volumes

• Works well if you use good (appropriate) bounding volumes and hierarchy

• Should give $O(\log n)$ rather than $O(n)$ complexity
  ($n=\# \text{ of objects}$)

• Can have multiple classes of bounding volumes and pick the best for each enclosed object
Questions

Given two bounding boxes at one level of the hierarchy, how do you compute the boxes for the next level?
Hierarchical bounding volumes

Spatial Subdivision

- Grids
- Octrees
- K-d trees and BSP trees
3D Spatial Subdivision

• Bounding volumes enclose the objects (object-centric)

• Instead could divide up the space—the further an object is from the ray the less time we want to spend checking it
  – Grids
  – Octrees
  – K-d trees and BSP trees
Octrees

- Quadtree is the 2-D generalization of binary tree
  - node (cell) is a square
  - recursively split into four equal sub-squares
  - stop when leaves get “simple enough”
Octrees

- Quadtree is the 2-D generalization of binary tree
  - node (cell) is a square
  - recursively split into four equal sub-squares
  - stop when leaves get “simple enough”

- Octree is the 3-D generalization of quadtree
  - node (cell) is a cube, recursively split into eight equal sub-cubes
  - for ray tracing:
    - stop subdivision based on number of objects
    - internal nodes store pointers to children, leaves store list of surfaces
  - more expensive to traverse than a grid
  - but an octree adapts to non-homogeneous scenes better

```c
trace(cell, ray) {       // returns object hit or NONE
  if cell is leaf, return closest(objects_in_cell(cell))
  for child cells pierced by ray, in order     // 1 to 4 of these
    obj = trace(child, ray)
    if obj!=NONE return obj
  return NONE
}
```
Which Data Structure is Best for Ray Tracing?

Grids

- Easy to implement
- Require a lot of memory
- Poor results for inhomogeneous scenes

Octrees

- Better on most scenes (more adaptive)

Spatial subdivision expensive for animations

- Hierarchical bounding volumes
- Better for dynamic scenes
- Natural for hierarchical objects
k-d Trees and BSP Trees

- Relax the rules for quadtrees and octrees:
  - k-dimensional (k-d) tree
    - don’t always split at midpoint
    - split only one dimension at a time (i.e. x or y or z)
  - binary space partitioning (BSP) tree
    - permit splits with any line
    - In 2-D space split with lines (most of our examples)
    - 3-D space split with planes
    - K-D space split with k-1 dimensional hyperplanes

- useful for Painter’s algorithm (hidden surface removal)
Building a Good Tree - the tricky part

• A naïve partitioning of $n$ polygons will yield $O(n^3)$ polygons because of splitting!

• Algorithms exist to find partitionings that produce $O(n^2)$.
  – For example, try all remaining polygons and add the one which causes the fewest splits
  – Fewer splits $\rightarrow$ larger polygons $\rightarrow$ better polygon fill efficiency

• Also, we want a balanced tree.
Raytracing
Global vs. Local Rendering Models

Local rendering models: the color of one object is independent of its neighbors (except for shadows)
Missing scattering of light between objects, real shadowing

Global Rendering Models
Raytracing—specular highlights
Radiosity—diffuse surfaces, closed environments
Object-oriented vs. Pixel-oriented Rendering

OpenGL rendering:
walk through objects, transforming and then drawing each one unless the z buffer says that it is not in front

Ray tracing
walk through each pixel looking for what object (if any) should be shown there
Light is Bouncing Photons

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

If we can model photon bounces we can generate images

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

Rays are the paths of these photons
This method of rendering by following photon paths is called *ray tracing*

*Forward* ray tracing follows the photon in direction that light travels (from the source)

BIG problem with this approach:
  Only a tiny fraction of rays reach the image
  Many, many rays are required to get a value for each pixel

Ideal Scenario:
  We'd like to magically know which rays will eventually contribute to the image, and trace only those
Backward Ray Tracing

The solution is to start from the image and trace backwards—*backward* ray tracing

Start from the image and follow the ray until the ray finds (or fails to find) a light source.
Backward Ray Tracing

Basic idea:

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

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

So head in that direction and see what is sending light

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

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

The basic algorithm is

compute u, v, w basis vectors

for each pixel do

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

intersect with all surfaces, find first one the ray hits

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

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

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

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

Details in book.
Derived using viewing transformations
Ray Object Intersection

Sphere
Triangle
Polygon

blackboard
Ray Object Intersection

Sphere
Triangle
Polygon

blackboard
Ray Object Intersection

Sphere
Triangle
Polygon

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

Ray-triangle—3D models composed of triangles

Ray-sphere—early models for raytracing, and now bounding volumes
Thought Experiment

- Ray tracing an implicit surface...
  - How would you do it?
  - What accelerations are possible?

http://www.cc.gatech.edu/~turk/vimp/scene.jpg
Recursive Ray Tracing

Four ray types:

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

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

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

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

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

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

\( \mathbf{l} \) the same for both points because this is a directional light (infinitely far away)
From Last time: Recursive Ray Tracing

Four ray types:

- **Eye rays**: originate at the eye
- **Shadow rays**: from surface point toward light source
- **Reflection rays**: from surface point in mirror direction
- **Transmission rays**: from surface point in refracted direction
Specular Reflection Rays

\[ r \quad \theta \quad \theta \quad n \quad d \]

blackboard
Transmission Rays

Dielelectrics—transparent material that refracts (and filters) light. Diamonds, glass, water, and air.

Light bends by the physics *principle of least time*

light travels from point A to point B by the fastest path

when passing from a material of one index to another Snell’s law gives the angle of refraction

When traveling into a denser material (larger $n$), light bends to be more perpendicular (eg air to water) and vice versa

<table>
<thead>
<tr>
<th>MATERIAL</th>
<th>INDEX OF REFRACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>air/vacuum</td>
<td>1</td>
</tr>
<tr>
<td>water</td>
<td>1.33</td>
</tr>
<tr>
<td>glass</td>
<td>about 1.5</td>
</tr>
<tr>
<td>diamond</td>
<td>2.4</td>
</tr>
</tbody>
</table>
Transmission Rays

Dielectrics—transparent material that refracts (and filters) light. Diamonds, glass, water, and air.

Snell’s law:
\[ n \sin \theta = n_t \sin \phi \]

\( n \) is the refractive index of the first material.
\( n_t \) is the refractive index of the second material.
Transmission Rays

Dielectrics—transparent material that refracts (and filters) light. Diamonds, glass, water, and air.

Snell’s law:
\[ n \sin \theta = n_t \sin \phi \]

\( n \) is the refractive index of the first material.

\( n_t \) is the refractive index of the second material.
Total Internal Reflection
Transmission Rays
Radiosity
Outline

- A Brief Review/Introduction to Radiosity
- The Radiosity Equation, Form Factors
- Putting it all together, and Improving
- More Realism: A digression, and Two-Pass Rendering
Local Illumination

- fast
- simple
- light → surface → viewer
- ignores many important effects
Review: Local vs. Global Illumination

- **Global illumination:** *Ray tracing*
  - Realistic specular reflection/transmission
  - Simplified diffuse reflection*

* Thursday, April 29, 2010
Beyond Ray Tracing

Ray tracing ignores the diffuse component of incident illumination
  – to achieve this component requires sending out rays from each surface point for the whole visible hemisphere

Even if you could compute such a massive problem there is a conceptual problem—loops:
  – point A gets light from point B
  – point B also gets light from point A
Doing it Right

The real solution is to solve simultaneously for incoming and outgoing light at all surface points. This is a massive integral equation.
Key Idea

• Model **diffuse** interaction only!
Doing it Right

The real solution is to solve simultaneously for incoming and outgoing light at all surface points. This is a massive integral equation.

Radiosity deals with the relatively easy case of purely diffuse scenes.
Doing it Right

The real solution is to solve simultaneously for incoming and outgoing light at all surface points. This is a massive integral equation.

*Radiosity* deals with the relatively easy case of purely diffuse scenes.

Or, you can sample many, many complete paths from light source to camera (photon mapping).
Advantages to diffuse-only model?

Specular interaction depends on viewer position—diffuse does not.

Result: The color seen at any point on any visible surface is independent of viewer position.

Radiosity produces a 3D model of surface patches with colors assigned to each.

Can be rendered in OpenGL.

Useful for architectural fly-throughs.
Review: Local vs. Global Illumination

- **Global illumination: Radiosity**
  - Realistic diffuse reflection
  - Diffuse-only: No specular interaction*
Review: Local vs. Global Illumination

- Global illumination: Ray tracing
  - Realistic specular reflection/transmission
  - Simplified diffuse reflection*

- Global illumination: Radiosity
  - Realistic diffuse reflection
  - Diffuse-only: No specular interaction*

indirect  direct  both
Raytracing Examples

http://www.povray.org/
Raytracing Examples

http://www.povray.org/
Radiosity Examples

Radiosity Examples

Radiosity

Raytracing

Thursday, April 29, 2010
Outline

- A Brief Review/Introduction to Radiosity
- The Radiosity Equation, Form Factors
- Putting it all together, and Improving
- More Realism: A digression, and Two-Pass Rendering
Review: Local vs. Global Illumination

- **Global illumination: Radiosity**
  - Realistic diffuse reflection
  - Diffuse-only: No specular interaction*
• Concentrate on patches instead.
• Want to have as few as possible.
Radiosity

Simple scene with diffuse surfaces

White wall should show effect of being near red wall

Compute light reflected between each pair of patches
Radiosity

Simple scene with diffuse surfaces
White wall should show effect of being near red wall
Compute light reflected between each pair of patches
Radiosity

Closed environment (office, factory)
Compute interaction between all patches (over which intensity is assumed to be constant)
View independent
Difficult to do specular highlights
Classical Radiosity in a Nutshell

Divide all surfaces into patches (squares are typical).
Determine a set of linear equations to model inter-reflection between all patches.
Solve set of simultaneous equations.
Render using standard hardware.
Assumptions of Classical Radiosity

No participating media (no light interaction with air, fog, etc)
Opaque surfaces—no transmission
Radiosity is constant across element
Colors (R, G, B) are independent
Assumptions of Classical Radiosity

Diffuse-only reflection and emission, so outgoing light radiates equally in all directions.

Light radiating from a point on a surface is independent of position on the surface—constant “radiosity” across a single surface.
What *is* radiosity?

- Radiosity $B(x) = \frac{dP}{dA}$
  - $P \rightarrow$ Energy (light “intensity”)
  - $A \rightarrow$ Area

- Integrating radiosity over a patch with respect to $A$ will yield $P$ for the patch

- Thus, radiosity is a representation of a patch’s intensity of light per unit area
What *is* radiosity?

- Radiosity determined by the sum of the emitted and reflected energy:
  \[
  B_i A_i = E_i A_i + R_i \sum_j B_j A_j F_{ji}
  \]

- *i* identifies the patch whose radiosity is being determined
- *j* identifies a single other patch
- *E* is emitted energy (light sources)
- *R* is reflectance (how much incoming light is reflected)
- *F* is the form factor between two patches
What *is* radiosity?

- Radiosity determined by the sum of the emitted and reflected energy:
  \[ B_i A_i = E_i A_i + R_i \sum_j B_j A_j F_{ji} \]

- Outgoing energy =
  
  Emitted energy + Reflected energy
Form Factor?

- $F_{ij}$: Fraction of light leaving patch $i$ arriving at patch $j$
- Determined by properties of $i$ and $j$:
  - Shape
  - Distance
  - Orientation
  - Occlusion by other patches
Form Factor Equation

\[ A_i F_{ij} = \int_{x \in P_i} \int_{y \in P_j} \frac{\cos \theta \cos \theta'}{\pi r^2} v(x, y) \, dy \, dx \]

x and y are points in i and j respectively

r is distance from x to y

Thetas are angles between patch normals and line between x and y

v(x,y) is a visibility function

Can points x and y see each other?
Simplify

Form factors are symmetric:

\[ A_i F_{ij} = A_j F_{ji} \]

Divide radiosity equation by \( A_i \)

\[ B_i A_i = E_i A_i + R_i \sum_j B_j A_j F_{ji} \]

\[ B_i = E_i + R_i \sum_j B_j A_j F_{ji} / A_i \]

\[ B_i = E_i + R_i \sum_j B_j F_{ij} \]
Our new equation gives the radiosity ($B$) of a single patch, so to specify the radiosity of all $n$ patches we need $n$ radiosity equations, one for each patch.

Known values:
- $E$ (given), $R$ (given), $F$ (computable)

Unknown: $B$

$n$ equations, $n$ unknowns
Linear System

Restate as a matrix equation...and solve

\[
\begin{bmatrix}
1 - R_1 F_{11} & - R_1 F_{12} & \cdots & R_1 F_{1n} \\
- R_2 F_{21} & 1 - R_2 F_{22} & \cdots & R_2 F_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
- R_n F_{n1} & R_n F_{n2} & \cdots & 1 - R_n F_{nn}
\end{bmatrix}
\begin{bmatrix}
B_1 \\
B_2 \\
\vdots \\
B_n
\end{bmatrix}
= \begin{bmatrix}
E_1 \\
E_2 \\
\vdots \\
E_n
\end{bmatrix}
\]

Each of our \( n \) linear equations contains \( n \) double integrals, one for each form factor.
The Radiosity “Pipeline”

1. Input Scene Geometry
2. Meshing (division into patches)
3. Form Factor Calculations
4. Input Reflectance/Emission Factors
5. Solve Radiosity Equation

Texture Geometry with Radiosity Solution
Being Smart about Form Factors

Form factors depend only on scene geometry. If geometry is constant, they only need to be calculated once.

Solution of the radiosity system is independent of viewing conditions, so if only the viewer position changes, it only needs to be solved once—can walk around the scene in real-time after it’s initially generated.
Being Smart about Form Factors

Form factors are complicated. Full numeric approximation of these is expensive—many special cases may be solved analytically.

Because we assume that radiosity is constant across a patch, two patches are typically assumed to be fully inter-visible or not at all inter-visible. That means that patches have to be small enough to resolve shadows and other complexities.
How to perform visibility testing?

Two basic methods, both of which have aliasing problems:

- Raycasting (typically slow)
- Hemicube method (z-buffer exploit)

Anti-aliasing may be performed in both cases.
Hemicube Visibility Testing

Render the entire scene from the perspective of the center of the current patch.

Rather than color, store patch identifiers, using the z-buffer to determine visibility.

Takes advantage of graphics hardware.

R. Ramamoorthi
Hemicube in Action

http://www.siggraph.org/education/materials/HyperGraph/radiosity/overview_2.htm
Hemicube in Action

Outline

• A Brief Review/Introduction to Radiosity
• The Radiosity Equation, Form Factors
• Putting it all together, and Improving
• More Realism: A digression, and Two-Pass Rendering
Classical Radiosity in a Nutshell, Revised

• Divide all surfaces into patches.
• Calculate form factors between all patches.
  – Lighting and viewer independent
• Solve the radiosity equation.
  – Viewer independent
• Render using standard 3D hardware.
The Radiosity “Pipeline”

- Input Scene Geometry
- Meshing (division into patches)
- Form Factor Calculations
- Input Reflectance/Emission Factors
- Solve Radiosity Equation

Texture Geometry with Radiosity Solution
Our Result

What is right?
What is wrong?
What went right?

Inter-reflection effects—clearly visible between the box on the right and the wall
What went wrong?

Blocky-looking—patch boundaries extremely obvious

Causes of blockiness

- Aliasing in hemicube method causes significant differences in radiosity between adjacent patches
- Large patch size

Thursday, April 29, 2010
Fixes?

Use antialiasing to clean up hemicube method

Interpolation
  Determine radiosity at each vertex of a patch and use bilinear interpolation to make things look smoother

Increase patch resolution (decrease size)
  Expensive if done uniformly – \( O(n^2) \)
  How can we do this intelligently?
Antialiasing (on hemicube)
Classical, Resolution 300
Classical, Resolution 1200
Classical, Resolution 2500
Supersampling, Resolution 100
Classical, Resolution 2500, Interpolated
Supersampled, Res 100, Interpolated
Adaptive Subdivision

Introduce a patch substructure—divide each patch into smaller *elements*.

Keep distinction between patches and elements in order to avoid efficiency problems.
Adaptive Subdivision

Determine light transport one-way from patches onto elements, not analyzing element-to-element interaction

$O(mn)$ for $m$ elements and $n$ patches. More expensive than the original $n^2$ approach, since $m \gg n$, but much better than $O(m^2)$. 
Adaptive Subdivision

Subdivide elements adaptively:

Begin with elements identical to patches.

Determine radiosity of an element, then compare to neighbors to obtain an error value. If within some error threshold, assign constant radiosity (or optionally interpolate).

Otherwise, subdivide the element and recurse until the error threshold or a minimum element size is reached.
Adaptive Subdivision

Results in very smooth-looking results for a relatively small amount of extra work

Shadows, areas near lights, and edges in general look much better

Not an idea specific to radiosity! Adaptive subdivision is a general tool used in many areas of graphics and other fields as well
Adaptive Subdivision Examples

http://www.acm.org/jgt/papers/TeleaVanOverveld97/
http://aig.cs.man.ac.uk/gallery/vrad.html

15-462 Computer Graphics I
Another example
Outline

• A Brief Review/Introduction to Radiosity
• The Radiosity Equation, Form Factors
• Putting it all together, and Improving
• More Realism
Radiosity Examples

Radiosity

Raytracing
Is raytracing really so bad?
Raytracing Examples

http://www.povray.org/
Raytracing Examples
Raytracing Examples
Can we inject specular effects into radiosity?

Raytracing
Yet More Realism
Wait a minute…

What's THIS?!
Specular Effects in Radiosity?

Keep viewer independence
Light reflected differently in different directions
Calculations for each source and each direction
Impractical
A Better Idea: The Best of Both Worlds

Combine radiosity and raytracing

Goal: Represent four forms of light transport:

- Diffuse -> Diffuse
- Diffuse -> Specular
- Specular -> Diffuse
- Specular -> Specular

Two-pass approach, one for each method
First Pass: Enhanced Radiosity

Diffuse -> Diffuse

Normal diffuse reflection model
Diffuse transmission (translucent objects) – requires modified form factor

Specular -> Diffuse

Specular transmission (transparent objects, e.g. windows) – involves extended form factor
Specular reflection (reflective objects, e.g. mirrors) – create actual “mirror image” environment with copies of all patches. Expensive!
Enhanced Radiosity - Evaluation

• Only accounts for a single specular reflection (try creating “mirror image” environments for two mirrors facing each other)
• Accurate diffuse model
• Equations solved as in the classical method
• Still viewer-independent
Second Pass: Enhanced Raytracing

- **Specular -> Specular**
  - Reflection and transmission as in classical method

- **Diffuse -> Specular**
  - Use the radiosity calculated in the first pass
  - Integrate incoming light over a hemisphere (or hemicube), or approximate with a tiny frustum in the direction of reflection
  - Recurse if visible surface is specular
Second Pass Result
(radiosity info. not yet used, just raytracing)
Two-Pass Global Illumination: Evaluation

Very expensive. Takes the cost of radiosity added to the cost of raytracing and then throws even more calculations into the mix.

Many approximations remain, particularly in specular -> diffuse and diffuse -> specular transport.
Two-Pass Global Illumination: Evaluation

Produces very convincing effects and works very well for scenes with small numbers of reflecting/transmitting objects.

Used in combination with other methods for extremely high-quality images.
More Pretty Pictures
Summary: Classical Radiosity

Divide all surfaces into patches.
Calculate form factors between all patches.
  Lighting and viewer independent
Solve the radiosity equation
  Viewer independent
Render using standard 3D hardware.
Acknowledgements/Resources

• Demo that explores resolution and other parameters
  – http://www.mvpny.com/RadTutMV/RadiosityTut1MV.html

• T. Yeap  (many great radiosity resources)
  – http://www.scs.leeds.ac.uk/cuddles/rover/main.htm

• Cornell graphics group  (many pretty pictures)
  – http://www.graphics.cornell.edu/online/research/
PHOTON MAPPING

15-462: Computer Graphics
March 19, 2009
Global illumination models can be basically partitioned into two sets:

- Point-sampling models
  - Ray tracing
- Integral-based models
  - Radiosity

There are also hybrid models that are a mix of the two.
Questions to Think About

- What are some of the effects that we can’t get using the rendering methods that we’ve gone over?
- More specifically, why can’t we capture these effects with our current models?
- Furthermore, can we come up with a good way to do this efficiently?
- Is there a *perfect* model that can do all of this?
Point-Sampling Models

- Monte Carlo Ray Tracing
  - Path Tracing
  - Bidirectional Path Tracing
  - Metropolis Light Transport
- Photon Mapping
Many of the more complex ray tracing-based algorithms can be defined as “Monte Carlo algorithms,” a class of algorithms that use repeated random sampling to compute results.

Monte Carlo ray tracing is a much more complicated extension of distributed ray tracing, although the two terms tend to overlap.
The biggest problem that these Monte Carlo ray tracing algorithms run into is the amount of noise left behind.

- This is particularly noticeable regarding caustics.

- We can remedy this by increasing the amount of rays we send out, but this is computationally more and more expensive.
We want a global illumination model that can do all of the following:

- All global illumination effects can be simulated
- Arbitrary geometry
- Low memory consumption
- Result is correct (ignoring the variance/noise)

Photon mapping is an alternative to the rendering algorithms presented earlier.
The Basic Idea

- Photon mapping is a two-pass algorithm
  - First pass: Photon tracing
    - Fire photons from all of the light sources in the scene
    - Build a “photon map” that stores their locations after being fired into the scene
  - Second pass: Rendering
    - Render the scene by taking the information stored in the photon map into account
The basic idea is simple: we fire a bunch of photons from all of the lights in the scene.

A photon thought of in a similar way to a ray, except that it also has its own intensity, or power, represented by its color.
Photon Tracing

- We can handle a variety of light sources using this model:
  - Point lights
  - Spherical lights
  - Area lights
  - Complex lights

- Regardless of the shape, the basic idea is the same: for every light, we want to send out photons uniformly in all directions.
  - This can be optimized further by only firing photons in the direction of the geometry in the scene.
Once a photon is fired, it bounces around the scene until it reaches some termination condition.

In the most general case, a photon can either be reflected or absorbed by the object.

- Absorption is the general termination condition upon which we then store it in the photon map.

More accurately, the BRDF of the object with which the photon collides will determine what will happen to it.

- That is, can it be specularly and diffusely reflected, refracted, etc?
For caustics, things are slightly different.

- We will specifically target specular objects in the scene such that we can capture caustics.
- While we normally might allow diffuse reflection for some photons, after a photon has been specularly reflected or refracted for some time, then comes in contact with a diffuse object, we want to terminate it instantly and store it in the photon map.
Russian Roulette

- The technique used to determine what happens to a photon when colliding with an object is a probabilistic technique called Russian Roulette.
- The following is an example of how to use Russian roulette on an object with both diffuse and specular reflection (with probabilities $p_d$ and $p_s$ respectively).
- We generate some random number $\varepsilon \in [0,1]$
  - $\varepsilon \in [0, p_d] \Rightarrow$ diffuse reflection
  - $\varepsilon \in [p_d, p_s + p_d] \Rightarrow$ specular reflection
  - $\varepsilon \in [p_s + p_d, 1] \Rightarrow$ absorption
Why does Russian Roulette work? And also, why do we use it?

Recall that when light hits an object, based on the BRDF, fractions of it are reflected, absorbed, etc...

- One way to deal with this is to spawn additional photons at each collision and reflect, refract, etc them with decreased intensity back into the scene.
- However, this substantially increases the number of photons we fire.
- Russian Roulette allows us to fire less photons at full intensity. We may fire less photons, but the expected value of the intensity remains the same.
The Photon Map

- We can use a spatial data structure to store our photon map. A good spatial data structure will give us the following:
  - Relatively fast insertions
  - Relatively fast lookups, but more importantly, a way of looking up elements “neighboring” a particular element.

- The structure that is most generally used for this is a balanced k-d tree. (This is what you will be asked to implement in project 4.)
Why a balanced k-d tree?

- A 3-D grid is impractical (and subsequently, so is an octree) because our distribution of photons is not uniform.
- A k-d tree is much more tailored to the distribution of the photons and balanced, it gives use O(log\(n\)) insertion and individual lookups. It also gives us a good representation of how to return the \(n\) nearest photons in relation to a given point. This is known more commonly as nearest-neighbor search.
The Photon Map (x2)

- One thing to note is that we actually want to keep two photon maps: one just for caustics and one for global illumination in general.
  - Caustics photon map: Concentrate on firing photons directly at specular objects in the scene.
  - Global photon map: Fire photons uniformly from all light sources into the scene.

- Why?
  - The global photon map will not contain enough photons to accurately represent caustics in our scene since not enough photons will be concentrated at points where caustics occur.
Radiance Estimate

- Once a photon has been fired and absorbed (or terminated), we can look it up again in the future. More specifically, we will need to look it up when calculating the radiance estimate of a point.
- The idea is simple: given a point, we want to find the \( n \) nearest photons and average over them to compute the radiance at that point.
- Getting the \( n \) nearest photons is easy: we run nearest neighbor search on our photon map.
Radiance Estimate

- Once we have these $n$ photons, we want to evaluate the reflected radiance at our point.

$$L_r(x, w) = \int_{\Omega} f_r(x, \omega', \omega') L_i(x, \omega')(\hat{n}_x \cdot \omega'')d\omega'$$

- We can approximate this by drawing a sphere around our $n$ photons with our point as the center. We then project all of our photons onto the circle defined by the intersection of the sphere and our object surface.

- Summing up all of the intensities of the photons and dividing them over the area of the circle will give us an accurate approximation for the radiance.
Rendering

- Photon tracing is a pre-rendering step that should be computed once before the second pass of the algorithm.
- In the rendering pass, we basically want to use the photon map(s) we built to assist us in computing the color of an object at a given point.
Color Computation

- We can now break down the color computation at a point in the following four components:
  - Direct Illumination
  - Specular Reflection
  - Caustics
  - Indirect Illumination

- Thus, solving the rendering equation boils down very simply to: \( L = L_D + L_S + L_C + L_I \)
Direct Illumination

- Direct illumination can be represented by the following integral:

$$\int_{\Omega_x} f_r(x, \omega', \omega)L_{i,l}(x, \omega)(\omega' \cdot \hat{n})d\omega'$$

- There are a variety of ways to evaluate this term, but the simplest way is to take a standard ray tracing algorithm and compute the color at the point of intersection without taking into account the contributions of any specular reflection...
Specular and glossy reflections (along with transmission through dielectrics) can be represented by the following integral:

$$\int_{\Omega_x} f_{\mathbf{r},S}(x, \omega', \omega)(L_{i,c}(x, \omega) + L_{i,d}(x, \omega))(\omega' \cdot \hat{n}) d\omega'$$

The best way to evaluate this term is to take advantage of a ray tracing algorithm and simply evaluate the color returned by reflecting or refracting a ray at the point of intersection.
Caustics can be represented using the following integral:

\[
\int_{\Omega_x} f_{r,D}(x, \omega', \omega)L_{i,c}(x, \omega)(\omega' \cdot \hat{n})d\omega'
\]

Calculating an accurate contribution from caustics can be achieved using our caustics photon map. We simply wish to take the radiance estimate from the map.

(Note: The global photon map will also give an approximate evaluation, but it is not as accurate)
Indirect Illumination

- Indirect illumination can be represented using the following integral:
  \[
  \int_{\Omega_x} f_{r,D}(x, \omega', \bar{\omega}) L_{i,d}(x, \bar{\omega})(\omega' \cdot \bar{n}) d\omega'
  \]

- Taking the radiance estimate from the global photon map will give us a good approximation of this term.
  - A more accurate evaluation can be achieved using Monte Carlo ray tracing, but this requires substantially more computation. The nice thing about our approximation is that caustics are separated out, which is the main cause for noise in Monte Carlo ray tracing.
Thus in summary, for each of the components of the color at a given point:

- $L_D$: best achieved using a general ray tracing algorithm (eye and shadow rays)
- $L_S$: also best achieved by taking the contribution from specular and transmitted rays
- $L_C$: taking the radiance estimate from our caustics photon map
- $L_I$: taking the radiance estimate from our global photon map or using Monte Carlo ray tracing techniques
Other Effects

- Photon mapping is also suitable for picking up other effects (which we aren’t actually going into):
  - Volume caustics
  - Subsurface scattering
What Variables do we Need?

- Position
- Velocity
- Radius
- Mass
- Racquet Info

Static
- Radius
- Mass
- Racquet Info

Dynamic
- Position
- Velocity
- Rotation?
What Happens Next?

- Position
- Velocity

\[
\mathbf{x} = \begin{bmatrix} x \\ y \\ z \\ \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix}
\]

Discrete Time: \[ \mathbf{x}_{t+1} = f(\mathbf{x}_t) \]

Continuous Time: \[ \dot{\mathbf{x}} = f(\mathbf{x}) \]
\[ \dot{x} = f(x) \]
DiffEQ Integration

Differential Equation Basics

Andrew Witkin

Physically Based Modeling
A Canonical Differential Equation

\[ \dot{x} = f(x, t) \]

- \( x(t) \): a moving point.
- \( f(x,t) \): \( x \)'s velocity.
The differential equation

\[ \dot{x} = f(x, t) \]

defines a vector field over x.
Integral Curves

Pick any starting point, and follow the vectors.
Given the starting point, follow the integral curve.
Euler’s Method

- Simplest numerical solution method
- Discrete time steps
- Bigger steps, bigger errors.

\[ x(t + \Delta t) = x(t) + \Delta t f(x, t) \]
Two Problems

• Accuracy
• Instability
Consider the equation:

\[
\dot{x} = \begin{bmatrix}
0 & 1 \\
-1 & 0
\end{bmatrix} x
\]

What do the integral curves look like?
Problem I: Inaccuracy

Error turns $x(t)$ from a circle into the spiral of your choice.
What is this a model for?

http://www.youtube.com/watch?v=3_fLO4xjTqg
Problem 2: Instability

• Consider the following system:

\[
\begin{align*}
\dot{x} &= -x \\
x(0) &= 1
\end{align*}
\]
Problem 2: Instability

To Neptune!
Accuracy of Euler Method

\[ \dot{x} = f(x) \]

Consider Taylor Expansion about \( x(t) \)...

\[ x(t + h) = x(t) + h f(x(t)) + \mathcal{O}(h^2) \]

Therefore, Euler’s method has error \( \mathcal{O}(h^2) \)... it is first order.

How can we get to \( \mathcal{O}(h^3) \) error?
The Midpoint Method

- Also known as second order Runge-Kutta:

\[ k_1 = h(f(x_0, t_0)) \]

\[ k_2 = hf(x_0 + \frac{k_1}{2}, t_0 + \frac{h}{2}) \]

\[ x(t_0 + h) = x_0 + k_2 + O(h^3) \]
The Midpoint Method

a. Compute an Euler step

\[ \Delta x = \Delta t f(x, t) \]

b. Evaluate \( f \) at the midpoint

\[ f_{\text{mid}} = f \left( \frac{x + \Delta x}{2}, \frac{t + \Delta t}{2} \right) \]

c. Take a step using the midpoint value

\[ x(t + \Delta t) = x(t) + \Delta t f_{\text{mid}} \]
q-Stage Runge-Kutta

General Form:

\[ x(t_0 + h) = x_0 + h \sum_{i=1}^{q} w_i k_i \]

where:

\[ k_i = f \left( x_0 + h \sum_{j=1}^{i-1} \beta_{ij} k_j \right) \]

Find the constant that ensure accuracy \( O(h^n) \).
4th-Order Runge-Kutta

\[ k_1 = hf(x_0, t_0) \]

\[ k_2 = hf(x_0 + \frac{k_1}{2}, t_0 + \frac{h}{2}) \]

\[ k_3 = hf(x_0 + \frac{k_2}{2}, t_0 + \frac{h}{2}) \]

\[ k_4 = hf(x_0 + k_3, t_0 + h) \]

\[ x(t_0 + h) = x_0 + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4 + O(h^5) \]

Why so popular?
## Order vs. Stages

<table>
<thead>
<tr>
<th>Order</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stages</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>11</td>
</tr>
</tbody>
</table>
More methods...

- Euler’s method is 1st Order.
- The midpoint method is 2nd Order.
- Just the tip of the iceberg. See *Numerical Recipes* for more.
- Helpful hints:
  - *Don’t* use Euler’s method (you will anyway.)
  - *Do* use adaptive step size.
Modular Implementation

- **Generic operations:**
  - Get dim(x)
  - Get/set x and t
  - Deriv Eval at current (x,t)

- **Write solvers in terms of these.**
  - Re-usable solver code.
  - Simplifies model implementation.
**Solver Interface**

- **System**
- **Solver**

- **Dim(state)**
- **Get/Set State**
- **Deriv Eval**
void eulerStep(Sys sys, float h) {
    float t = getTime(sys);
    vector<float> x0, deltaX;
    t = getTime(sys);
    x0 = getState(sys);
    deltaX = derivEval(sys, x0, t);
    setState(sys, x0 + h*deltaX, t+h);
}
Implicit Integration
Implicit Integration

Implicit Methods

David Baraff
“Give me Stability
or
Give me Death”

— Baraff’s other motto
• If your step size is too big, your simulation blows up. It isn’t pretty.

• Sometimes you have to make the step size so small that you never get anyplace.

• Nasty cases: cloth, constrained systems.
A very simple equation

A 1-D particle governed by \( \dot{x} = -kx \) where \( k \) is a stiffness constant.

\[ E = \frac{1}{2} kx^2 \]
Euler’s method has a speed limit

\[ x = -kx \quad \Delta x = -hkx \]

\[ h \approx 1/k: \text{ oscillate.} \quad h > 2/k: \text{ explode!} \]
Stiff Equations

• In more complex systems, step size is limited by the largest $k$. One stiff spring can screw it up for everyone else.

• Systems that have some big $k$’s mixed in are called stiff systems.
Consider the equation:

\[ \dot{x} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} x \]

What do the integral curves look like?
Explicit Integration

Thursday, April 29, 2010
Explicit Integration
(Explicit) Euler Method

\[ x(t_0 + h) = x(t_0) + h \dot{x}(t_0) \]
Implicit Euler Method

\[ x(t_0 + h) = x(t_0) + h \dot{x}(t_0) \]

\[ x(t_0 + h) = x(t_0) + h \dot{x}(t_0 + \Delta t) \]
Implicit Integration
Implicit Integration
Implicit Euler

\[
\begin{align*}
\dot{x} &= -x \\
x(0) &= 1
\end{align*}
\]
One Step: Implicit vs. Explicit

\[ \dot{x} = -x, \quad x(0) = 1 \]

Correct Solution: \[ x(h) = e^{-hk} \]

Implicit Euler Step: \[ x(h) = \frac{1}{1 + hk} \]

Explicit Euler Step: \[ x(h) = 1 - hk \]
\[
\frac{d}{dt} \dot{X}(t) = \ddot{X}(t) = f(X(t))
\]

\[
\Delta X(t_0) = h \dot{X}(t_0 + \Delta t) = h f(X(t_0 + \Delta t)) = h f(X(t_0) + \Delta X(t_0))
\]
(Linearized) Implicit Integration

\[ \dot{X}(t) = f(X(t)) \]

\[ \Delta X = hf(X_0 + \Delta X) \]

\[ \Delta X = h \left( f(X_0) + \left( \frac{\partial f}{\partial X} \right) \Delta X \right) \]
Single-Step Implicit Euler Method

\[ \Delta X = h \left( f(X_0) + \left( \frac{\partial f}{\partial X} \right) \Delta X \right) \]

\[
\left( \mathbf{I} - h \frac{\partial}{\partial X} \left( \dot{X}(t_0) \right) \right) \Delta X = h \dot{X}(t_0)
\]

\( n \times n \) sparse matrix
Solving Large Systems

- Matrix structure reflects force-coupling: 
  
  \[(i,j)\text{th entry exists iff } f_i \text{ depends on } X_j\]

- Conjugate gradient a good first choice

- Is this a lot of work?
Long Term Effect

Implicit integration causes this rotational system to spiral inwards.
Cloth
What is cloth?
Two basic types...

Woven

Knit
Woven Cloth

Knit Cloth

What is cloth?

- 2 basic types: woven and knit
- We'll restrict to woven
- Warp vs. weft

Figure 1.8. The weaving process.

House, Breen [2000]
Warp and Weft

Cloth and Fur Energy Functions

Michael Kass
Stretch (Continuum Version)

\[ (u, v) \rightarrow \bar{x} \]

\[ S_u = \left| \frac{\partial \bar{x}}{\partial u} \right| - 1 \]

\[ E = \frac{1}{2} k \int (S_u^2 + S_v^2) du dv \]
Shear (Continuum Version)

\[ \theta = \cos^{-1} \left( \frac{\vec{x} \cdot \vec{x}}{\| \vec{x} \| \| \vec{x} \|} \right) \]

\[ E = \frac{1}{2} k \int \theta^2 du dv \]
Bend (Continuum Version)

\begin{align*}
E &= \frac{1}{2} k \int (\kappa_u^2 + \kappa_v^2) du dv
\end{align*}
Resitence To...

- Stretching
- Shearing
- Bending
Discretization
Basic Model
Warp Strings
Weft Springs
Parameters

- Given stretch, shear, and bending constants...
- How would you make a wrinkly t-shirt, thick cloth, or non-uniform cloth?