Spatial Data Structures

Hierarchical Bounding Volumes
Grids
Octrees
BSP Trees
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

\[ n^2 \text{ computations} \]
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.

\[ \text{Cost} = n^*B + m^*l \]

- \( n \) - is the number of rays tested against the bounding volume
- \( B \) - is the cost of each test  \((\text{Do not need to compute exact intersection!})\)
- \( m \) - is the number of rays which actually hit the bounding volume
- \( l \) - 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 \( \text{--- } O(n) \)
- 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 \[ \text{--- } O(n) \]
- 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
  ($n=$ # of objects)

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

- Data structure: a 3-D array of cells (voxels) that tile space
  - Each cell points to list of all surfaces intersecting that cell

- Intersection testing:
  - Start tracing at cell where ray begins
  - Step from cell to cell, searching for the first intersection point
  - At each cell, test for intersection with all surfaces pointed to by that cell
  - If there is an intersection, return the closest one
Grids

- Cells are traversed in an incremental fashion
- Hits of sets of parallel lines are very regular
More on Grids

- Be Careful! The fact that a ray passes through a cell and hits an object doesn’t mean the ray hit that object in *that* cell

- Optimization: cache intersection point and ray id in “mailbox” associated with each object

  - Step from cell to cell
  - Get object intersecting cell
  - Find closest intersection
  - If found intersection --- done
More on Grids

• Grids are a poor choice when the world is nonhomogeneous (clumpy)
  – many polygons clustered in a small space

• How many cells to use?
  – too few ⇒ many objects per cell ⇒ slow
  – too many ⇒ many empty cells to step through ⇒ slow

• Non-uniform spatial subdivision is better!
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)
Painters Algorithm

Hidden Surface Elimination
Painters Algorithm

- Need to sort objects back to front
- Order depends on the view point
- Partition objects using BSP tree
- View independent
Building a BSP Tree

- Let’s look at simple example with 3 line segments
- Arrowheads are to show left and right sides of lines.
- Using line 1 or 2 as root is easy.
- (examples from http://www.geocities.com/SiliconValley/2151/bsp.html)
Drawing Objects

- Traverse the tree from the root
- If viewpoint is on the left of the line --- traverse right sub-tree first
- Draw the root
- Traverse left sub-tree

`3 2 1`

*a BSP tree using 2 as root*
Building the Tree 2

Using line 3 for the root requires a split
Drawing Back to Front

- Use Painter’s Algorithm for hidden surface removal

Steps:
- Draw objects on far side of line 3
  » Draw objects on far side of line 2a
    - Draw line 1
  » Draw line 2a
- Draw line 3
- Draw objects on near side of line 3
  » Draw line 2b

Viewpoint
Triangles

Use plane containing triangle $T_1$ to split the space.
If view point is on one side of the plane draw polygons on the other side first.
$T_2$ does not intersect plane of $T_1$. 
Triangles

Use plane containing triangle $T_1$ to split the space
If view point is on one side of the plane draw polygons on the other side first
$T_2$ does not intersect plane of $T_1$
Triangles

Split Triangle
Painter’s Algorithm with BSP trees

• Build the tree
  – Involves splitting some polygons
  – Slow, but done only once for static scene

• Correct traversal lets you draw in back-to-front or front-to-back order for any viewpoint
  – Order is view-dependent
  – Pre-compute tree once
  – Do the “sort” on the fly

• Will not work for changing scenes
Drawing a BSP Tree

- Each polygon has a set of coefficients:
  \[ Ax + By + Cz + D \]
- Plug the coordinates of the viewpoint in and see:
  >0 : front side
  <0 : back facing
  =0 : on plane of polygon
- Back-to-front draw: inorder traversal, do farther child first
- Front-to-back draw: inorder traversal, do near child first

```c
front_to_back(tree, viewpt) {
    if (tree == null) return;
    if (positive_side_of(root(tree), viewpt)) {
        front_to_back(positive_branch(tree, viewpt);
        display_polygon(root(tree));
        front_to_back(negative_branch(tree, viewpt);
    } else { ...draw negative branch first...}
}
```
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 --> larger polygons --> better polygon fill efficiency

• Also, we want a balanced tree.
Demos

BSP Tree construction
http://symbolcraft.com/graphics/bsp/index.html

• KD Tree construction
Real-time and Interactive Ray Tracing

The OpenRT Real-Time Ray-Tracing Project
http://www.openrt.de/index.php

• Interactive ray tracing via space subdivision
  http://www.cs.utah.edu/~reinhard/egwr/

• Interactive ray tracing with good hardware
  http://www.cs.utah.edu/vissim/projects/raytracing/