Hierarchical Data Structures, Scene Graph and Quaternion

Jian Huang
Spatial Data Structure

- Octree, Quadtree
- BSP tree
- K-D tree
Spatial Data Structures

• Data structures for efficiently storing geometric information. They are useful for
  – Collision detection (will the spaceships collide?)
  – Location queries (which is the nearest post office?)
  – Chemical simulations (which protein will this drug molecule interact with?)
  – Rendering (is this aircraft carrier on-screen?), and more

• Good data structures can give speed up rendering by 10x, 100x, or more
Bounding Volume

- Simple notion: wrap things that are hard to check for ray 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.
- Can build bounding volume hierarchies.
Bounding Volumes

• Choose Bounding Volume(s)
  – Spheres
  – Boxes
  – Parallelepipeds
  – Oriented boxes
  – Ellipsoids
  – Convex hulls
Quad-trees

- Quad-tree 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

- Octree is the 3-D generalization of quad-tree
- Node (cell) is a cube, recursively split into eight equal sub-cubes
  - stop splitting when the number of objects intersecting the cell gets “small enough” or the tree depth exceeds a limit
  - internal nodes store pointers to children, leaves store list of surfaces
- More expensive to traverse than a grid
- Adapts to non-homogeneous, clumpy scenes better
K-D tree

- The K-D approach is to make the problem space a rectangular parallelepiped whose sides are, in general, of unequal length.
- The length of the sides is the maximum spatial extent of the particles in each spatial dimension.
K-D tree
K-D Tree in 3-D

- Similarly, the problem space in three dimensions is a parallelepiped whose sides are the greatest particle separation in each of the three spatial dimensions.
Motivation for Scene Graph

• Three-fold
  – Performance
  – Generality
  – Ease of use

• How to model a scene?
  – Java3D, Open Inventor, Open Performer, VRML, etc.
Scene Graph Example
Scene Graph Example
Scene Graph Example
Scene Graph Example
Scene Description

- Set of Primitives
- Specify for each primitive
  - Transformation
  - Lighting attributes
  - Surface attributes
    - Material (BRDF)
    - Texture
    - Texture transformation
Scene Graphs

• Scene Elements
  – Interior Nodes
    • Have children that inherit state
    • transform, lights, fog, color, …
  – Leaf nodes
    • Terminal
    • geometry, text
  – Attributes
    • Additional sharable state (textures)
Scene Element Class Hierarchy
Scene Graph

• Graph Representation
  – What do edges mean?
  – Inherit state along edges
    • group all red object instances together
    • group logical entities together
      – parts of a car
  – Capture intent with the structure
Scene Graph

• Inheritance -- Overloaded Term
  – Behavior inheritance (subclassing)
    • Benefit of OO design
  – Implementation inheritance
    • Perhaps provided by implementation language
    • *Not essential* for a good API design
  – Implied inheritance
    • Designed into the API
Scene Graph (VRML 2.0)

Interior Nodes
- Groups
- Transformations
- Levels of Detail
- Lights
Example Scene Graph
Scene Graph Traversal

- Simulation
  - Animation
- Intersection
  - Collision detection
  - Picking
- Image Generation
  - Culling
  - Detail elision
  - Attributes
Scene Graph Considerations

• Functional Organization
  – Semantics
• Bounding Volumes
  – Culling
  – Intersection
• Levels of Detail
  – Detail elision
  – Intersection
• Attribute Management
  – Eliminate redundancies
Functional Organization

- Semantics:
  - Logical parts
  - Named parts
Functional Organization

- Articulated Transformations
  - Animation
  - Difficult to optimize animated objects
Bounding Volume Hierarchies
View Frustum Culling
Level Of Detail (LOD)

- Each LOD nodes have distance ranges
Attribute Management

- Minimize transformations
  - Each transformation is expensive during rendering, intersection, etc. Need automatic algorithms to collapse/adjust transform hierarchy.
Attribute Management

• Minimize attribute changes
  – Each state change is expensive during rendering
Question: How do you manage your light sources?

• OpenGL supports only 8 lights. What if there are 200 lights? The modeler must ‘scope’ the lights in the scene graph?
Sample Scene Graph
Think!

• How to handle optimization of scene graphs with multiple competing goals
  – Function
  – Bounding volumes
  – Levels of Detail
  – Attributes
Scene Graphs Traversal

• Perform operations on graph with traversal
  – Like STL iterator
  – Visit all nodes
  – Collect inherited state while traversing edges
• Also works on a sub-graph
Typical Traversal Operations

• Typical operations
  – Render
  – Search (pick, find by name)
  – View-frustum cull
  – Tessellate
  – Preprocess (optimize)
Scene Graphs Organization

• Tree structure best
  – No cycles for simple traversal
  – Implied depth-first traversal (not essential)
  – Includes lists, single node, etc as degenerate trees

• If allow multiple references (instancing)
  – Directed acyclic graph (DAG)

• Difficult to represent cell/portal structures
State Inheritance

• General (left to right, top to bottom, all state)
  – Open Inventor
  – Need Separator node to break inheritance
  – Need to visit all children to determine final state

• Top to bottom only
  – IRIS Performer, Java3D, …
  – State can be determined by traversing path to node
Scene Graphs Appearance Overrides

• One attempt to solve the “highlighting” problem
  – After picking an object, want to display it differently
  – Don’t want to explicitly edit and restore its appearance
  – Use override node in the scene graph to override appearance of children

• Only works if graph organization matches model organization
Appearance Override
Multiple Referencing (Instancing)

• Convenient for representing multiple instances of an object
  – rivet in a large assembly

• Save memory

• Need life-time management
  – is the object still in use
  – garbage collection, reference counts
Multiple Referencing

• Changes trees into DAGs
• Instance of an object represented by its *path*, (path is like a mini-scene)
• Difficult to attach instance specific properties
  – e.g., caching transform at leaf node
Other Scene Graph Organizations

- Logical structure (part, assembly, etc.)
  - Used by modeling applications
- Topology structure, e.g., boundary
  - surfaces, faces, edges, vertices
  - Useful for CAD applications
- Behaviors, e.g., engine graph
- Environment graph (fog, lights, etc.)
- Scene graph is not just for rendering!!
Specifying Rotation

• How to parameterize rotation
  – Traditional way: use Euler angles, rotation is specified by using angles with respect to three mutually perpendicular axes
    • Roll, pitch and yaw angles (one matrix for each Euler angle)
    • Difficult for an animator to control all the angles (practically unworkable)
      – With a sequence of key frames, how to interpolate??
      – Separating motion from path

• Better to use parameterized interpolation of quaternions
Quatetion

- A way to specify rotation
- As an extension of complex numbers
- Quaternion:
  \[ u = (u_0, u_1, u_2, u_3) = u_0 + iu_1 + ju_2 + ku_3 = u_0 + \textbf{u} \]
  
  - Pure quaternion: \( u_0 = 0 \)
  
  - Conjugate: \( u^* = u_0 - \textbf{u} \)
  
  - Addition: \( u + v = (u_0 + v_0, u_1 + v_1, u_2 + v_2, u_3 + v_3) \)
  
  - Scalar multiplication: \( c \cdot u = (cu_0, cu_1, cu_2, cu_3) \)
Quaternion multiplication

• $u \times v$
  
  $= (u_0 + iu_1 + ju_2 + ku_3)(v_0 + iv_1 + jv_2 + kv_3)$
  
  $= [u_0 v_0 - (u \cdot v)] + (u \times v) + u_0 v + v_0 u$

• The result is still a quaternion, this operation is not commutative, but is associative

• $u \times u = - (u \cdot u)$

• $u \times u^* = u_0^2 + u_1^2 + u_2^2 + u_3^2 = |u|^2$

• $\text{Norm}(u) = u/|u|$

• Inverse quaternion:

  $u^{-1} = u^*/|u|^2$, $u \times u^{-1} = u^{-1} \times u = 1$
Polar Representation of Quaternion

- Unit quaternion: \(|u|^2 = 1\), normalize with \(\text{norm}(u)\)
- For some theta, \(-\pi < \theta < \pi\), unit quaternion, \(u\):
  \[
  |u|^2 = \cos^2(\theta) + \sin^2(\theta)
  \]
  \[
  u = u_0 + |u|s, \quad s = u/|u|
  \]
  \[
  u = \cos(\theta) + ss\sin(\theta)
  \]
Quaternion Rotation

• Suppose \( p \) is a vector \((x,y,z)\), \( p \) is the corresponding quaternion: \( p = 0 + p \)

• To rotate \( p \) about axis \( s \) (unit quaternion: \( u = \cos(\theta) + s\sin(\theta) \)), by an angle of \( 2\theta \), all we need is: \( upu^* (u \times p \times u^*) \)

• A sequence of rotations:
  – Just do: \( u_n u_{n-1} \ldots u_1 pu_1^* \ldots u_{n-1}^* u_n^* = 0 + p' \)
  – Accordingly just concatenate all rotations together: \( u_n u_{n-1} \ldots u_1 \)
Quaternion Interpolation

- Quaternion and rotation matrix has a strict one-to-one mapping (pp. 489, 3D Computer Graphics, Watt, 3rd Ed)
- To achieve smooth interpolation of quaternion, need spherical linear interpolation (slerp), (on pp. 489-490, 3D Computer Graphics, Watt, 3rd Ed)
  - Unit quaternion form a hyper-sphere in 4D space
  - Play with the hyper-angles in 4D
- Gotcha: you still have to figure out your up vector correctly
More

- If you just need to consistently rotate an object on the screen (like in your lab assignments), can do without quaternion
  - Only deal with a single rotation that essentially corresponds to an orientation change
  - Maps to a ‘hyper-line’ in a ‘transformed 4D space’
  - Be careful about the UP vector
  - Use the Arcball algorithm proposed by Ken Shoemaker in 1985