Scene Management

Introduction

• Graphics cards can render a lot, and fast
  — But never as much or as fast as we’d like!
• Intelligent scene management squeezes more graphics performance out of limited resources
  — Scene graphs
  — Scene partitioning
  — Visibility calculations
  — Level of detail control

Outline

• Scene Graphs

Scene Graphs

• Specification of object and attribute relationships
  — Spatial
  — Hierarchical
  — Material properties
• Easy to “attach” objects together
  — E.g. Riding in a vehicle
• Implementation does not need to be objects in tree
  — Can use pointers (e.g. to textures, sprites) instead
• Logical and possibly spatial relationships
  — Often goal is to make it easy to discard large swaths so do not need to render
→ Spatial data structures (next)

Motivation for Scene Graphs

• Consider game with people, in a car, on a road
• People move around the car, don’t affect the position of car
• But car moving affects position of people
• If massive hand picks up road → affects location of car and people!
• Exists beyond positions, too
  — Consider animations or textures tied to skeletons
• To make movement/drawing more efficient, structure that supports such relationships → Scene graphs

Spatial Data Structures

• Spatial data structures store data indexed by location
  — E.g. Store according to Position ...
  — Without graphics, used for queries like “Where is the nearest hotel?” or “Which stars are near enough to influence the sun?”
• Multitude of uses in computer games
  — Visibility - What can player see?
  — Collision detection - Did bullet just hit wall?
  — Proximity queries - Where is nearest health-pack?
• Can reduce “cost” with fast, approximate queries that eliminate most irrelevant objects quickly
  — Trees with containment property enable this
  — Cell of parent completely contains all cells of children
  — If query fails for cell, it will fail for all children
  — If query succeeds, try it for children
  — Cost? → Depends on object distribution, but roughly O(log n)
Spatial Data Structures

- For games, focus on spatial data structures that partition space into regions, or cells, of some type
  - Generally, cut up space with planes that separate regions
- Uniform Grids
  - Split space up into equal sized / number of cells
- Quad (or Oct) Trees
  - Recursively split space into 4 (or 8) equal-sized regions
  - Can do with a sphere, too
- Binary-Space Partitioning (BSP) trees
  - Recursively divide space along a single, arbitrary plane
- k-dimensional trees (k-d trees)
  - Recursively partition in k dimensions until termination condition
    (e.g. 1 object per cell)

Uniform Grid

- Cells can be approximately size of view distance
- Only need consider objects in cell and neighbor
- Pro: Easy to find, compute
- Con: Not effective if many objects in one cell

Quad Tree

- Each node has exactly 4 children
- For 2-d space, subdivide into 4 regions
- Split until (max-1) objects in each cell
  - E.g. 1 object in each

Binary Space Partitioning (BSP) Tree

- Recursively sub-divide space into convex sets
- For 3-d polygon scenes, can apply painter’s algorithm
  - Draw leaves of tree up (back polygons written first)
  - (Originally used in Doom before zbuffer to get fast rendering)

K-D tree

- Instead of 2 dimensions (binary) can use k-dimensions

Cell-Portal Structures

- Cell-Portal data structures dispense with hierarchy → just store neighbor information
  - Makes them graphs, not trees
- Cells described by bounding polygons
- Portals polygonal openings between cells
- Good for visibility culling algorithms, OK for collision detection and ray-casting
- Several ways to construct
  - By hand, as part of authoring process
  - Automatically, starting with BSP or k-d tree and extracting cells and portals
  - Explicitly, as part of automated modeling process
Cell-Portal Visibility

- Keep track of which cell viewer is in
- Enumerate all visible regions
- Preprocess to identify potentially visible set (PVS) for each cell

Potentially Visible Set (PVS)

- PVS: Set of cells/regions/objects/polygons that may be seen from a particular cell
  - Want to identify objects that can be seen
  - Trade-off is memory consumption vs. accurate visibility
- Computed as pre-process
  - Easy for static objects (e.g., cells)
  - Need strategy to manage dynamic objects
- Used in various ways:
  - As only visibility computation - render everything in PVS for viewer’s current cell
  - As first step - identify regions of interest, then apply more accurate run-time algorithms

Cell-to-Cell PVS

- Cell A in cell B’s PVS if stabbing line from portal of B to portal of A
  - Stabbing line → line segment intersecting only portals
  - Neighbor cells are trivially in PVS

Putting it All Together

- The “best” solution will be a combination
  - Static things
    - E.g., quad-tree for terrain
    - E.g., cells and portals for interior structures
  - Dynamic things
    - E.g., quick reject using bounding spheres
  - Balance between pre-computation and run-time computation

Group Exercise (1)

- Assume you want to design SceneManager for Dragonfly
- Support: Altitude
  - Keep current levels, but have more efficient data structure
- Design SceneManager
  - Attributes (data structures)?
  - Methods?
- What existing code need refactoring?

Group Exercise (2)

- Consider additional Scene Management functionality
  - More efficient collision detection
- Consider simple first (list), then advanced (grid)
- To support, what is needed ...
  - Attributes (data structures)?
  - Methods?
- What existing code need refactoring?
Group Exercise (3)

• Consider views with SceneManager grid
  – How can they be used for more efficient drawing with views?
• Sketch out algorithm