UCL

Speeding Up Ray Tracing

UCL

Optimisations

- Limit the number of rays
- Make the ray test faster
  - for shadow rays
    - the main drain on resources if there are several lights
  - for primary rays
  - for all rays

Over 90% of the cost of ray tracing is in ray-object intersection tests

UCL

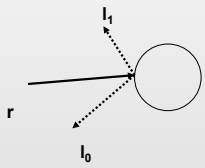
Ray Tracing Acceleration

- Intersect ray with all objects
  - Way too expensive
- Faster intersection algorithms
  - Little effect
- Less intersection computations
  - Space partitioning (often hierarchical)
    - Grid, octree, BSP or kd-tree, bounding volume hierarchy (BVH)

UCL

Optimisation for Shadow Rays

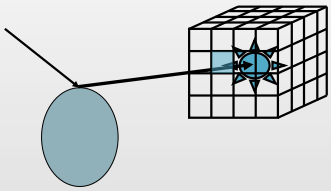
- Problem with shadow rays is that for every intersection we trace an additional N rays for each light



UCL

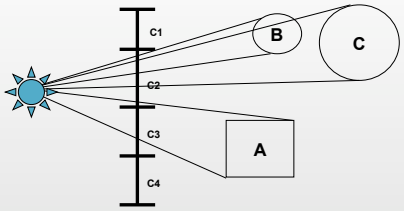
Optimisation for Shadow Rays: Light Buffer

- Enclose light in a box.
- Cells of box faces store objects



UCL

Construction (2D Analogue)

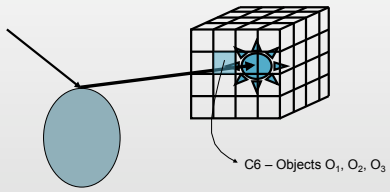


C1 - <NULL> C2 - A, B, C

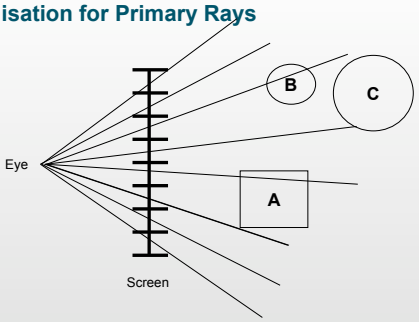
C3 - A C4 - <NULL>

Using the Light Buffer

- Check intersection of ray with polygons stored in cell
- Case with object itself…



Optimisation for Primary Rays

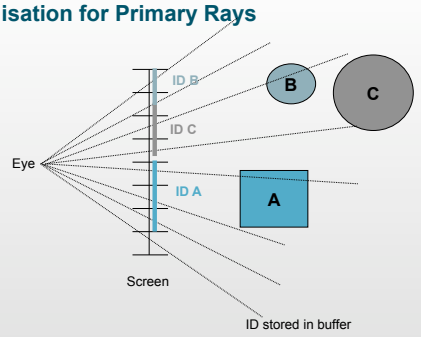


Optimisation for Primary Rays

- Use a z-buffer!
- Instead of writing colour write an object identifier
  - Easy to support in OpenGL - turn off lighting, do flat shading and encode object id within 24bit colour
- Difficult technique to use elsewhere because rays are no longer spatially coherent and evenly spaced

UCL

Optimisation for Primary Rays



UCL

Optimisation for General Rays

- Techniques to use
  - bounding volumes
  - hierarchical bounding volumes
  - space subdivision
    - regular
    - adaptive
  - ray coherence

UCL

Bounding Volume

- Find a tight bounding volume and use it for a first reject test
- If hit volume then test full object

Axis Aligned Bounding Box (AABB)

Bounding Box

Bounding Sphere

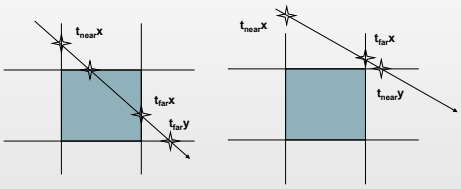
UCL

Fast BV Tests (AABB)

- Box-Ray test (when box planes parallel to axes)
  - a box is three sets of parallel planes, each set orthogonal to the other two,
  - ray defined by $q(t) = q_0 + t.dq$
    - Calculate t_{near} for each of the three plane pairs
    - find max of the 3 t_{near}
    - Calculate t_{far} for each of the three plane pairs
    - find min of the 3 t_{far}
  - If max t_{near} is greater that min t_{far}, then the box is not intersected

UCL

Fast BV Tests (AABB, 2D case)



UCL

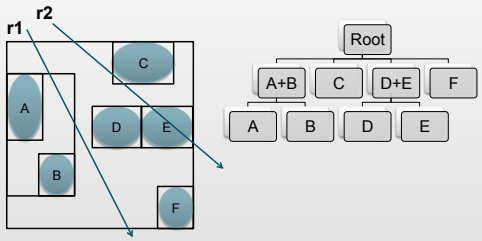
Choosing a Volume

- Choice depends on the cost of the test and the fit of the shape
  - The “void” area can be very large
- More efficient fitting shapes are possible (this is still a research area e.g. k-dops)

UCL

Bounding Volume Hierarchy

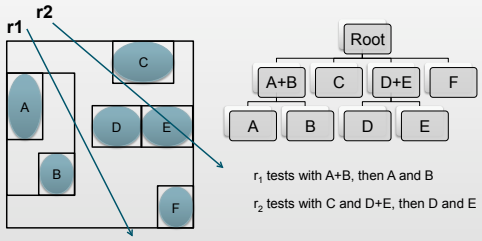
- Organise a hierarchy of bounding volumes
  - Bounding volumes of bounding volumes



UCL

Bounding Volume Hierarchy

- Organise a hierarchy of bounding volumes
  - Bounding volumes of bounding volumes



UCL

Choosing a BVH

- Scene graph might not map to a decent space partitioning
  - Group BVs based on actual proximity rather than scene graph position
- You could e.g., sort the BVH using a BSP tree ...

UCL

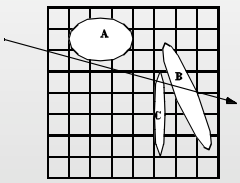
Bounding Volume Hierarchy

- Advantages:
  - Very good adaptivity
  - Efficient traversal O(log N)
- Problems
  - How to arrange BVs?

UCL

Regular Spatial Subdivision (Grid)

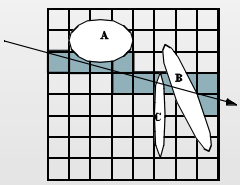
- Regular 3D grid of “voxels”
- In each voxel, store the list of objects that intersects with the voxel



UCL

Regular Spatial Subdivision

- Ray traverses the regular grid
- For each traversed voxel, intersection test with stored objects



UCL

RSS: Issues

- Grid traversal
  - Requires enumeration of voxel along ray → 3D-DDA
  - Simple and hardware-friendly
- Grid resolution
  - Strongly scene dependent
  - Cannot adapt to local density of objects
    - Problem: „Teapot in a stadium“
  - Possible solution: hierarchical grids

UCL

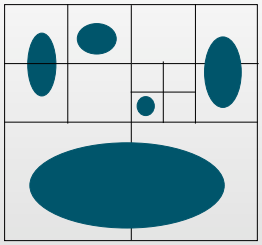
RSS: Issues

- Objects in multiple voxels
  - Store only references
  - Use mailboxing to avoid multiple intersection computations
    - Store (ray, object)-tuple in small cache (e.g. with hashing)
    - Do not intersect if found in cache
  - Original mailbox uses ray-id stored with each triangle
    - Simple, but likely to destroy CPU caches

UCL

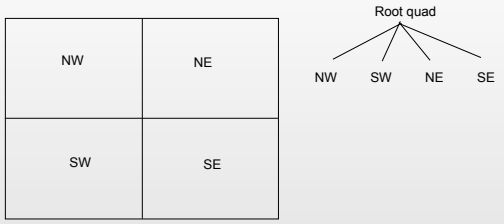
Adaptive Spatial Subdivision

- The octree idea (illustrated with a quadtree!)



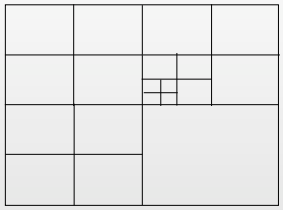
UCL

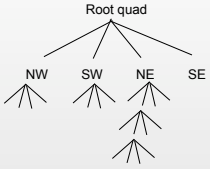
Quadtree Definition



UCL

Building a Quadtree



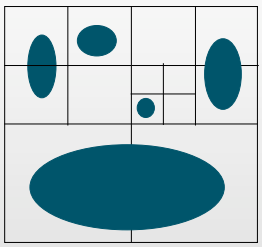


- Split recursively each quad into 4

UCL

Building the Octree: 3D representation

- Recursively split the cells into eight, until each cell meets some criteria
  - e.g. “only one object intersects each cell”



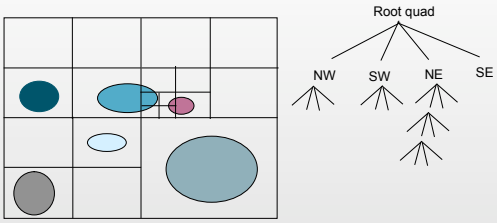
UCL

Octree Representation

- Advantage is clear - cells are not wasted on void areas
- Disadvantage:
  - Doesn't build on object shape
    - Object can belong to different branches of the tree
    - Split to isolate the object can be too fine
  - Cost to traverse can be high

UCL

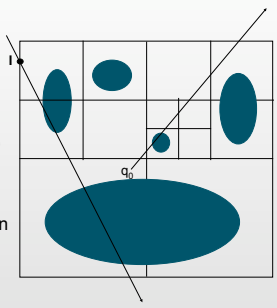
Octree Drawbacks: Example



UCL

Tracing an Octree

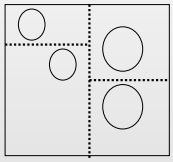
- Find octree voxel containing start point (q_0) or find intersection (I) of ray with cube that surround the octree
- ***R***: Find (I) in the octree
  - it is moved a little along the ray to fall inside a cube
- Intersect ray with faces of the cube it is in,
- Find the intersecting point when the ray exit the voxel
- Repeat at ***R*** until ray out of the whole volume



UCL

KD-Tree

- Similar to octree, but ***orthogonal*** splitting planes are chosen to provide a good search tree



Split based on balancing object numbers and minimising volume disparity

UCL

BSP Tree

- Generalisation of the kd-tree (which is a generalisation of an octree)
- Choosing your BSP tree is hard!
- Can be done by finding a plane that separates the objects in two equal sets, and applying the subdivision recursively

UCL

Traversing a BSP tree

- BSP-RayIntersect(Ray, Node)
  - Test if ray interval empty or no more node
  - If node is a leaf, intersect ray with objects
  - Else
    - Clip ray to near side of the plane (RayNear)
    - BSP-RayIntersect(RayNear, Node->near)
    - If (no intersection)
      - Clip ray to far side of the plane (RayFar)
      - BSPIntersect (RayFar, Node->far)

UCL

Conclusion

- Several can be applied for accelerating the ray intersection tests with the scene
- Some are specific to rays from light source or viewpoints
- Some are more general
- Choosing a good scene partitioning is crucial, but depends on the scene structure
- Need a good trade off between scene partitioning and traversal efficiency (and memory cost)