Speeding up your game

- The scene graph
- Culling techniques
- Level-of-detail rendering (LODs)
- Collision detection
- Resources and pointers

(adapted by Marc Levoy from a lecture by Tomas Möller, using material from Real-Time Rendering)
The scene graph

- DAG – directed acyclic graph
  - Simply an $n$-ary tree without loops
- leaves contains geometry
- each node holds a
  - bounding volume (BV)
  - pointers to children
  - possibly a transform
- examples of BVs: spheres, boxes
- the BV in a node encloses all the geometry of the nodes in its subtree
Scene graph example

circles=BVs

scene graph

root
Using transforms for instancing...

- put transform in internal node(s)

- Move up
- Move right, Rotate 45°
…or hierarchical animations

No hierarchy:  
one transform

Hierarchy: 3 transforms

Tomas Möller © 2000
Types of culling

- backface culling
- hierarchical view-frustum culling
- portal culling
- detail culling
- occlusion culling
Backface culling

- often implemented for you in the API
- OpenGL: `glCullFace(GL_BACK);`
- requires consistently oriented polygons
(Hierarchical) view frustum culling
Variants

- octree
- BSP tree
  - axis-aligned
  - polygon-aligned (like Fuchs’s algorithm)
- if a splitting plane is outside the frustum, one of its two subtrees can be culled
Portal culling

- plan view of architectural environment
- circles are objects to be rendered
Simple algorithm (Luebke and Georges ‘95)

- create graph of environment (e.g. building)
  - nodes represent cells (e.g. rooms)
  - edges represent portals between cells (doors)

- for each frame:
  - V cell containing viewer, P screen bbox
  - * render V’s contents, culling to frustum through P
  - V a neighbor of V (through a portal)
  - project portal onto screen, intersect bbox with P
    - if empty intersection, then V is invisible from viewer, return
    - if non-empty, P intersection, recursively call *
Example

Images courtesy of David P. Luebke and Chris Georges

typical speedups: 2x - 100x

Tomas Möller © 2000
Variants

- stop recursion when cell is too far away
- stop recursion when out of time
- compute potentially visible set (PVS)
  - viewpoint-independent pre-process
  - which objects in V2 might be visible from V1?
  - only meaningful if V1 and V2 are not adjacent
  - easy to be conservative; hard to be optimal
Detail culling

- cull object if projected BV occupies less than N pixels
- not much visible difference here, but 1x - 4x faster
- especially useful when moving
Estimating projected area

- distance in direction $d$ is $d \cdot (c-v)$
- projected radius $p$ is roughly $(n \cdot r) / (d \cdot (c-v))$
- projected area is $p^2$
Occlusion culling

- main idea: objects that lie completely "behind" another set of objects can be culled
- "portal culling" is a special case of occlusion culling
Sample occlusion culling algorithm

- draw scene from front to back
- maintain an “occlusion horizon” (yellow)
Sample occlusion culling algorithm

- to process tetrahedron (which is behind grey objects):
  - find axis-aligned box of projection
  - compare against occlusion horizon
Sample occlusion culling algorithm

- when an object is partially visible:
  - add its bounding box to the occlusion horizon
Hierarchical Z-buffer algorithm
(Greene, Kass, and Miller 1993)

- octree in object space
  + multiresolution Z-buffer in screen space
- used in both NVIDIA and ATI chips
Object-space octree
(shown using quadtree)
Object-space octree
(shown using quadtree)
Object-space octree
(shown using quadtree)
Object-space octree
(shown using quadtree)
Object-space octree
(shown using quadtree)
Object-space octree
(shown using quadtree)
Object-space octree
(shown using quadtree)
Hierarchical Z-buffer

- reduce cost of Z-testing large polygons
- maintain low-res versions of Z-Buffer
Level-of-detail rendering

• use different levels of detail at different distances from the viewer
Level-of-detail rendering

- Not much visual difference, but a lot faster

- Use area of projection of BV to select appropriate LOD
Collision detection

- cannot test every pair of triangles: $O(n^2)$
- use BVs because these are cheap to test
- better: use a hierarchical scene graph
Testing for collision between two scene graphs

- start with the roots of the two scene graphs
- testing for collision between the bounding volumes of two internal nodes
  - if no overlap, then exit
  - if overlap, then descend into the children of the internal node with largest volume
- an internal node against a triangle
  - descend into the internal node
- a triangle against a triangle
  - test for interpenetration
Triangle - triangle collision test

- compute the line of intersection between the supporting planes of the two triangles

- compute the intersection interval between this line and the two triangles
  - gives two intervals

- if the two intervals overlap, then the two triangles interpenetrate!
Simpler collision detection

- only shoot rays to find collisions, i.e., approximate an object with a set of rays
- cheaper, but less accurate
Can you compute the time of a collision?

- move ball, test for hit, move ball, test for hit... can get “quantum effects”!
- in some cases it’s possible to find closed-form expression: $t = s / v$
Resources and pointers

● Real Time Rendering (the book)
  – http://www.realtimerendering.com

● Journal of Graphics Tools
  – http://www.acm.org/jgt/