INTRODUCTION TO
COMPUTER GRAPHICS AND
INTERACTION

SCENE MANAGEMENT

Christopher Peters
CST, KTH Royal Institute of Technology, Sweden
chpeters@kth.se
http://kth.academia.edu/ChristopherEdwardPeters
what is scene management?
• the reduction of all scene data to a subset of only the data that could possibly be visible from the position of the viewer (i.e. anything out of sight is not even considered for rendering)

why is scene management necessary?
• most graphics (rendering) calculations are complex and can be very time-consuming
• even visibility calculations (clipping against viewing volume & back-face culling) can take a long time
A model contains lots of polygons:

Let’s say we want to figure out if a line is intersecting with the tank.
There are many reasons we may want to do this, e.g. the movement of a projectile fired at the tank.
One option: compare the line with every polygon on the tank...
In this case:

```cpp
for (every polygon in the tank) {
    Intersect the line with the polygon
    If the line intersects then collision = true
}
```
Speed of the algorithm is dependent on the number of polygons in the object.

If a projectile is nowhere near the tank (e.g. is on the other side of the map), the algorithm will still have to check every single polygon in the tank to see if there is an intersection.

Need a quick way to decide whether something is near the tank. And to be able to reject it very quickly if it is not.
A Solution

Bounding volumes offer (one part of) the solution
As an aside, spatial partitioning and other techniques can also be used to speed up

- We will consider them slightly later

For the tank object, we may have chosen a number of different bounding volumes:
Bounding Volumes

BV$s chosen to enclose all the vertices in a mesh

Ensure every triangle or polygon is also contained

The bounding volume should be made as small as possible

Different bounding volumes may be more appropriate depending on the layout of the object being fitted
Popular bounding volumes
Notice that each volume is very simple
Can do very quick calculations in each case to figure out if e.g. a line is intersecting a sphere or a box
Tests using these bounding volumes are also popular for determining if something is within the view volume when doing visibility testing
Spheres are a common volume type chosen for bounding objects.
Simple representation = extremely fast calculations
Object rotates in the game world = with proper positioning, it is usually not necessary to update the sphere to match the objects new orientation.
Another popular type of bounding volume is unlike spheres, depending on the type of object, they may provide a better fit:

However, it may be slower to do tests against bounding boxes than spheres.
As the tank changes orientation, update the bounding box to ensure it still encapsulates object.
OBB’s

Here, the box is oriented with respect to the tank

Extra calculations are needed when doing tests against the volume
Here, the box is remains orientated with respect to the main axes
Bounding Ellipsoid

Useful for meshes of some other shapes
Other bounding volumes also possible
E.g. cylinder
In practice, bounding boxes and spheres are the most commonly used
We can also calculate bounding spheres that encapsulate other bounding spheres and bounding volumes for subparts of objects.

In this example, a separate bounding sphere is created for the turret, gun, body and antennae of the tank:
Say we want to quickly test a point to see if it is in a danger zone for our tank
Take three scenarios:

Let’s see what happens in each case...
We do a quick test to see if the point is inside of the outer sphere. In this case, the point is outside. We can therefore reject it very quickly and do not need to do any more calculations.
We do the same test as before with the outer sphere. This time we find the point is inside the outer sphere. We therefore compare it with the lower level bounding spheres, and end up testing the point with the main gun mesh.
Case 3.

We do the same test as before with the outer sphere
Again, we find the point is inside the outer sphere
We compare it with the lower level bounding spheres...
Test the point with the all the meshes apart from the main gun mesh
These tests, which do quick high-level tests to see if objects are potentially intersecting, form what is called the **Broad Phase** collision detection. Quickly find the sets of objects that may be colliding with each other.

What happens if the objects are found to be potentially colliding? Then we need to do further tests to see if it is the case and, if so, find out where the objects are colliding. This is referred to as the **Narrow Phase** of collision detection.
Gilbert-Johnson-Keerthi algorithm
Solves proximity queries between two complex convex polyhedra
Given the two polyhedra:
Computes the distance $d$ between them
Can also return the closest pair of points on each polygons
Scene Management Techniques

trees are the data structure of choice for many applications

scene management methods

• BSP (Binary Space Partitioning) trees
• Quadtrees & Octrees
• Portals (rendering the PVS – Potential Visibility Set)

scene management methods can be combined

(example: portals with BSP trees)
Binary Space Partitioning Trees

- BSP trees use a binary tree structure to store the geometry of a scene [Fuchs et al. 1980]
- BSP trees are a very efficient scene management method that allows for very fast rendering of complex scenes
- the creation of the BSP tree can take a long time as many complicated operations can be involved in the insertion of data into the tree
  - creating a BSP tree structure from the geometric data of a scene is called “compilation” of the BSP tree
  - compiling the BSP tree is an off-line task (compilation should not be attempted in real-time)
BSP trees in games

• BSP trees have been proven to be highly successful for real-time rendering in computer games
  – the rise of the FPS games genre would not have been possible without BSP trees
  – examples: Doom (2D-space partitioning only), Quake etc.
Polygon-Aligned BSP tree compilation [Akenine-Möller and Haines 2002]

• starting from an arbitrarily selected polygon (usually from the geometric centre of a scene), all polygons of the scene are inserted into the tree

• the position of a polygon in relation to polygons that are already inside the tree decides into which branch of the binary tree (left or right) a polygon is entered

• if a polygon of the scene that has not yet been inserted into the BSP tree intersects with the plane defined by another polygon which is already inside of the tree, that polygon may have to be split into two polygons

this method can be simplified (splitting of polygons can be disallowed)

note: simplification of this method may lead to less accurate rendering

Christopher Peters  
Scene Management  
chpeters@kth.se
BSP tree rendering

- the BSP tree is traversed in-order and the position of the polygon in each tree-node is tested against the virtual camera position & alignment
- if the polygon in a BSP tree node is found to be outside of the view of the camera, the whole branch of the BSP tree (the node and all its children) can be discarded (i.e. it does not have to be traversed)
- this method can considerably reduce the amount of data that will have to be sent to the renderer
construction of a simple BSP tree

simple example (2D BSP tree, no polygon splitting)

Ordering of BSP tree:
- objects in front → right node
- objects to the back → left node
construction of a BSP tree (cont)

select an arbitrary polygon

(here: wall 0)

enter into the root of the tree
construction of a BSP tree (cont)

select next polygon

(here: wall 1)

1 is in front of 0
→ traverse right
→ enter into leaf
select next polygon

*(here: wall 2)*

2 is in front of 0
→ traverse right
2 is in front of 1
→ traverse right
→ enter into leaf
construction of a BSP tree (cont)

select next polygon
(here: wall 3)

3 is in front of 0
→ traverse right
3 is in front of 1
→ traverse right
3 is to the back of 2
→ traverse left
→ enter into leaf
using a BSP tree for rendering

Solution:

1 is not in view
→ right branch of 0 (1 and all its child nodes) can be discarded

9 is not in view
→ right branch of 8 (9 and all its child nodes) can be discarded
Quadtrees

A tree structure in which every tree node holds four child nodes

- quadtrees divide a scene up into rectangular areas that contain objects (or polygons)
- objects are usually stored in the greatest depth of the tree (exception: use of quadtree for CLOD rendering [Ulrich 2000])
- if a quad (or part of a quad) is visible (in front of the virtual camera) then the child nodes of the quad need to be tested for visibility
- if a quad is not visible then none of its child nodes needs to be traversed and consequently none of the objects (or polygons) contained within the quad need to be rendered

Christopher Peters  
Scene Management  
chpeters@kth.se
quadtree example (cont)

- each of the quads of the scene is split up again into four even smaller quads
- each quad now holds a sixteenth of the scene

An octree works similar to quadtrees (with an expansion into the 3rd dimension, i.e. using cubes instead of squares)
Portals provide a simple scene-management method

[Akenine-Möller and Haines 2002]

- environment is divided into cells that are connected through portals

Portal Rendering

1. test which neighbouring cells are visible from camera position
2. recursively test visible cells for visibility of their neighbours
3. render (draw) all visible cells
Portal Rendering Examples

Visibility:
- cell C (camera position)
  - cell B (from C)
  - cell F (from C)
    - cell D (from F)
      - cell B (from D)

Visibility:
- cell B (camera position)
  - cell A (from B)
  - cell D (from B)
    - cell E (from D)


Next lecture

- You should be working on Lab 3

- Next lab help session:
  13:00-15:00, Visualisation Studio,
  Thursday 12th May

- Animation and image based rendering
  - 11th May
  - 15:00–17:00