Scene Management

- The scene management system is responsible for efficiently rendering complex scenes
- We will mainly focus on real time scene management, but many of these techniques are also useful for offline rendering
- The system maintains a world full of objects and determines what gets drawn and in what order
- Some of the primary components include:
  - Scene graph
  - Culling
  - Level of detail (LOD)
  - Draw order
  - Instancing
  - Paging
Layers

- The scene management layer deals primarily with objects.
- The rendering layer deals primarily with triangles and graphics state.
Scene Graph

- Computer graphics often makes use of a scene graph of some sort
- At a minimum, graphics scene graphs would probably have a tree structure where each node in the tree stored some geometry and a transformation (matrix or some other form)
- The hand from project 2, for example, can be represented as a scene graph. Each node in the tree would have a box model and a transform that does one or two rotations
Hand Scene Graph

Palm

Digit00
Digit01
Digit02

Digit10
Digit11
Digit12

Digit20
Digit21
Digit22

Digit30
Digit31
Digit32
Real Time Scene Graph

- Combining the transformation and geometry into a single node makes sense, but for more complex scenes, it is more common to separate the two into different nodes.

- As always, there are different ways to do this, but today, we will consider a scene graph with a base node class and various derived nodes that perform specific functions.

- Our base Node will not really do anything, but will have a generic ‘Draw()’ function that can be overridden by higher level nodes.

```cpp
class Node {
public:
    virtual void Draw();
};
```
The Transform node is just a local transformation in the scene graph (exactly like the local joint rotations we did in project 2)

class Transform : public Node {
  Matrix44 Mtx;
  Node *Child;
public:
  void Draw() {
    if(Child == 0) return;
    glPushMatrix();
    glLoadIdentity();
    glMultMatrix(&Mtx); // need to convert Mtx* to float[]…
    Child->Draw();
    glPopMatrix();
  }
};
The ‘Instance’ node is an instance of some piece of geometry.

It stores a pointer to a Model class, so that multiple instances could potentially share the same model data.

Notice that Instance doesn’t have any ‘child’ nodes, and can therefore only be a leaf node in the scene tree.

class Instance: public Node {
    Model *Mod;
public:
    void Draw() {if(Mod) Model->Draw();}  
};
We define a ‘Group’ node which stores an array of child nodes, but doesn’t perform any real computation.

Groups are used when one needs to parent several nodes to a single node (like the fingers on the hand).

class Group:public Node {
    int NumChildren;
    Node *Child;
public:
    void Draw() {
        for(int i=0;i<NumChildren;i++)
            Child[i]->Draw();
    }
};
Hand Scene Graph (version 2)

Group
  Instance (palm)
    Transform
      Group
        Transform
          Group
            Transform
              Instance (digit00)
                Group
                  Transform
                    Instance (digit00)
                      Group
                        Transform
                          Instance (digit00)
                            Group
                              Transform
                                Instance (digit00)
                                  Instance (digit02)

Transform
  Instance (digit01)
    Group
      Transform
        Instance (digit01)
          Group
            Transform
              Instance (digit01)
                Instance (digit01)

Transform
  Instance (digit10)
    Group
      Transform
        Instance (digit10)
          Group
            Transform
              Instance (digit10)
                Instance (digit11)

Transform
  Instance (digit12)
    Group
      Transform
        Instance (digit12)
          Group
            Transform
              Instance (digit12)
                Instance (digit12)

etc… etc…
The second version of the scene graph might look a bit more complicated, but the extra flexibility offered by the technique usually justifies it.
Sub-Tree Instancing

- We can also have nodes parented to more than one node, as long as we avoid any cyclic loops in our tree graph.
- Having nodes parented to more than one parent changes the tree graph into a directed acyclic graph or DAG.
- For example, if we wanted several copies of the hand at different locations:
Bounding Volume Culling
Culling

- The term *cull* means to remove from a group.
- In graphics, it means to determine which objects in the scene are *not* visible.
- We looked at culling of individual triangles. Today, we will consider culling of objects.
- There are many approaches to object culling (and they can usually be combined easily):
  - Bounding volumes (spheres, boxes…)
  - Cull-planes, face clustering
  - Occlusion surfaces
  - Portals
  - Potentially Visible Sets (PVS)
Camera View Volume

The main data that defines a real-time perspective camera includes:

- World space camera matrix
- Field of view (FOV)
- Aspect ratio
- Near & far clipping plane distances
Backface Culling

- Backface culling refers to the removal of individual triangles that face away from the camera.
- This is usually built into the lower level renderer (OpenGL / Direct3D…) and is not really a part of scene management.
Bounding Volumes

- Objects are contained within simple bounding volumes (sphere, cylinder, box…)
- Before drawing an object, its bounding volume is tested against the camera’s viewing volume. There are 3 possible outcomes:
  - Totally visible
  - Totally invisible
  - Partially visible (may require clipping)
Bounding Volume Types

- Sphere
- Cylinder
- Hot dog / capsule / lozenge
- AABB: axis-aligned bounding box
- OBB: oriented bounding box
- Convex polyhedron
Generating Bounding Spheres

- Method 1: Average vertex positions
  - Step 1: Compute average of all vertex positions and place the center of the bounding sphere there.
  - Step 2: Find the vertex farthest from the center and set the radius to that distance
- Will rarely, if ever, generate optimal results
Optimal Enclosing Sphere

Sphere ComputeSphere(int N,Point P[]) {
    randomly mix up points P[0]...P[N-1];
    Sphere sphere(P[0],0);
    i=1;
    while (i<N) {
        if(P[i] not in support) {
            if(P[i] not in sphere) {
                add P[i] to support and remove any unnecessary points;
                compute sphere from current support;
                i=0; // start over when support changes
                continue;
            }
        }
        i++;
    }
    return sphere;
}
Testing Bounding Spheres

- Transform bounding sphere to view space
- Compare against all 6 view volume faces
- Object is totally invisible if it is completely outside any one plane
- Object is totally visible if it is completely inside all planes
- Otherwise, the object may intersect the view volume and may require clipping
Transforming to Camera Space

- Object’s world matrix: $W$
- Camera’s world matrix: $C$
- Sphere position in object space: $s$
- Sphere position in view space: $s' = C^{-1} \cdot W \cdot s$
if(s.z-radius > -NearClip) then outside
else if(s.z+radius<-FarClip) then outside
else if(s.z+radius>-NearClip) then intersect
else if(s.z-radius<-FarClip) then intersect
else inside
Testing Right, Left, Top, & Bottom

dist = n \cdot s'
if(dist>radius) then outside
if(dist<-radius) then inside
else intersect

Right plane:
n = [\cos(\text{FOV}/2), 0, \sin(\text{FOV}/2)]

Top plane:
n = ~[0, \cos(\text{FOV}/2)/\text{Aspect}, \sin(\text{FOV}/2)]
Performance

- Transform sphere center to view space: 9*, 9+
- Compare to near & far plane: 6+
- Compare to left & right: 2*, 6+
- Compare to top & bottom: 2*, 6+
- Total: 13*, 27+
Special Case for Corners
Cull Node

- We can easily integrate bounding volume culling into our scene graph
- We could make various types of culling volumes such as:

```cpp
class CullSphere: public Node {
    Node *Child;
    Vector3 Position;
    float Radius;

    public:
    bool IsVisible();
    void Draw() {
        if(IsVisible() && Child!=0) Child->Draw();
    }
};
```
Bounding volumes can be grouped hierarchically (i.e., bounding volumes can contain other bounding volumes). This allows a method for complex objects with many parts to be culled. If a bounding volume is totally visible, then volumes contained within it do not need to be cull tested. Only if an outer volume intersects the border of the view volume do the inner volumes need to be tested. If the individual objects move around, it may be necessary to dynamically re-compute the outer bounding volumes.
Clipping Issues

- Ideally, clipping and even clip testing of individual triangles should be avoided, especially on the PS2.
- If an object is totally visible, it should not be clip tested.
- If an object intersects the far clipping plane, we might just want to reject it entirely.
- If an object intersects the top, bottom, or side clipping planes, we might be OK using hardware scissoring.
- If an object intersects the front clipping plane, it probably needs to be clipped (or maybe we can reject individual triangles).
- Consider differences in requirements between large objects (terrain) and small objects (characters, props).
Culling Planes

- Objects with detailed surfaces that are more or less flat can take advantage of culling planes.
- A good example is the exterior walls of a building.
- This technique of backface culling can be extended to more general objects by clustering polygons with similar normals.
- We can also make a CullPlane node in our tree that will only draw its sub-tree if the camera is on the ‘front’ side of the plane.
Culling

- Here's an example of a building with 4 detailed walls and a roof

```
CullSphere
  Group
    CullPlane Instance (north wall)
    CullPlane Instance (west wall)
    CullPlane Instance (south wall)
    CullPlane Instance (east wall)
    CullPlane Instance (roof)
```
Level of Detail
Level of Detail

- Level of Detail (LOD) refers to the various techniques of reducing detail for objects as they get further away.
- There are various techniques:
  - Discrete LODs
  - Progressive meshes
  - Patches, NURBS, tessellated surfaces
  - Terrain techniques
Discrete LOD

- Several discrete LODs are prepared off line and dynamically switched based on simple distance comparisons (ideally it should take the camera FOV and resolution into account too)
- Tried and true method used for many years in flight simulation
- Basically no CPU overhead for algorithm itself
- Hardware is well adapted to rendering static display lists
- Additional memory required for storing LODs, but all low LODs together are usually smaller than the high LOD
- Can use sophisticated mesh decimation tools off line for preparing models
- If desired, techniques exist for cross dissolving (fading) between LODs
- Bottom line: very fast and very simple (quick & dirty)
Progressive Meshes

- Automated mesh decimation and dynamic reconstruction technique
- Can take any model and then render it dynamically with any number of polygons (less than or equal to the original number)
- Requires about 1/3 extra memory (or more, depending on details of the implementation)
Progressive Meshes
Progressive Mesh Performance

- Rendering requires custom processing at the per-vertex and per-triangle level, and so the algorithm, while pretty efficient, has trouble competing with the simpler discrete LOD technique. In other words, even though it may be able to render fewer triangles, the per-triangle cost is higher, usually making the overall approach slower.

- Neat technology and useful research, but hasn’t been too successful in video games just yet. Future graphics hardware may have better support for progressive meshes, and so they may be more important later. Also, mesh decimation technology is still very useful in off line modeling.
Patches & Subdivision Surfaces

- Geometry is defined by a control mesh that explicitly describes a high order surface
- Surface is dynamically tessellated into polygons, based on camera distance and other visual quality controls
- Patches & subdivision surfaces allow for smooth, rounded surfaces to be described without using tons of memory
- Overall approach suffers from similar drawbacks to progressive meshes as far as its usefulness to games
Terrain Rendering
LOD Issues

- Color & lighting (normals) pops are more visible than geometry or silhouette pops.
- Camera is often moving and there are often many dynamic objects in the scene. This can help mask LOD transitions.
- Ideally, LOD should be pixel error based.
Here is a simple example of a two-level LOD node.
One could also make a LOD node that switches between several different levels.
Instead of just switching between different levels of geometry, the LOD node can switch entire sub-trees. In this way, the low detail object could just be a single Instance, but the high detail version could contain a whole sub-tree of stuff…

```cpp
class LOD:public Node {
    Node *LowDetail;
    Node *HighDetail;
    float SwitchDist;

public:
    float DistToCamera();
    void Draw() {
        if(DistToCamera() < SwitchDist) HighDetail->Draw();
        else LowDetail->Draw();
    }
};
```
Let’s add level of detail switching to our building

- LOD
  - CullSphere
    - LOD
      - Instance (low detail)
        - CullPlane Instance (north wall)
        - CullPlane Instance (west wall)
        - CullPlane Instance (south wall)
        - CullPlane Instance (east wall)
        - CullPlane Instance (roof)
Now, let’s make a bunch of copies of the building at different locations.

- Instances
  - Group
    - Transform
      - CullSphere
        - LOD
          - Instance (low detail)
            - Group
              - CullPlane Instance (north wall)
              - CullPlane Instance (west wall)
              - CullPlane Instance (south wall)
              - CullPlane Instance (east wall)
              - CullPlane Instance (roof)
More Scene Management
Off-Screen Rendering

- Imposters
- Shadows
- Environment maps
- Full-screen effects (image distortion, blurs, color processing…)
- Other visual effects…
Imposters

- Imposters are an LOD technique where an polygonal object is replaced by a simple picture (usually a texture map on a rectangle)
- Complex geometric objects such as trees, are often rendered with imposters
- Imposters can be generated offline or on the fly by using off-screen rendering
- There are a variety of ways in which one can use imposters...
There are several reasons why one would want to control the order in which objects are drawn:

- Transparency & some effects often require back-to-front order
- Minimize graphics state changes (keeps pipeline running full speed & minimizes cache thrashing)
- Expensive pixel operations (bump mapping…) can run faster if rendered front-to-back because if a pixel is occluded (by the zbuffer), it will detect this early and skip the expensive operation
- Some shadowing effects require objects rendered in order from closest to furthest from the light
- Some textures or other data must be ‘rendered’ off screen before being used in the scene
- Some special effects require manipulating the entire rendered image, and must run last

It is possible to have conflicting draw order requirements…
Scene Draw Order Example

1. Render any off-screen textures
2. Draw sky (clears framebuffer and z-buffer)
3. Draw all opaque objects
4. Draw all transparent objects and localized effects, sorted back to front at the object level. Individual objects and effects could sort at the polygon level if necessary
5. Full-screen image processing effects (color processing, image warping, camera fade…)
6. Heads-up display (HUD)
Paging

- Modern 3D worlds are too big to fit in game machine memory (32 megs on PS2, 64 on XBox, 24+8 on GameCube)
- Data can be dynamically paged off of the DVD ROM drive
- Must compete with audio & other streaming data
Procedural Modeling

- In addition to paging, data can also be generated on the fly using procedural modeling.
- As one gets closer and closer to a particular object, it will be constructed with more and more detail.
Load Balancing

- LOD tolerances can be dynamically adjusted to provide a stable framerate.
- Uses timers to analyze how long the last frame took to render, then adjusts LODs so that current frame makes the most of available CPU time.
Portal Culling
Portals

- Culling algorithm designed specifically to handle interior environments (buildings, dungeons, mazes…)
- Can work in conjunction with bounding volume hierarchies and other culling algorithms
- World is made up of *rooms* connected by rectangular *portals*
Portals
Portals
Portals
Portal Issues

- Imposters
- Portal clipping
- Camera location
- Combining with bounding volume culling
- Moving objects
- Dynamic portals (opening & closing doors)
- Procedurally generating portals
PVS: Potentially Visible Sets
PVS Algorithm

- Static world is broken up into individual renderable objects
- The space of all legal camera positions is broken up into zones
- A precomputed table is generated that lists for each zone, all objects that might be visible
- Potentially visible objects are then further tested with bounding volume culling
PVS
PVS
PVS Table Generation

for (zone=1 to num camera zone) {
    for (i=1 to NUM_SAMPLES) {
        Vector3 eye=GenerateLegalEyePosition();
        for (j=1 to 6) {
            image img=RenderBoxView(eye, j);
            for (k=1 to num pixels in img) {
                AddToPVS(zone, img.pixel[k]);
            }
        }
    }
}
PVS Issues

- Overall, the algorithm can work quite well and handle very complex culling situations efficiently.
- However, the PVS table can get quite large and require a lot of memory.
- Also, defining viewing zones can be tricky.