Graphics & Visualization

Chapter 9

Scene Management

Introduction

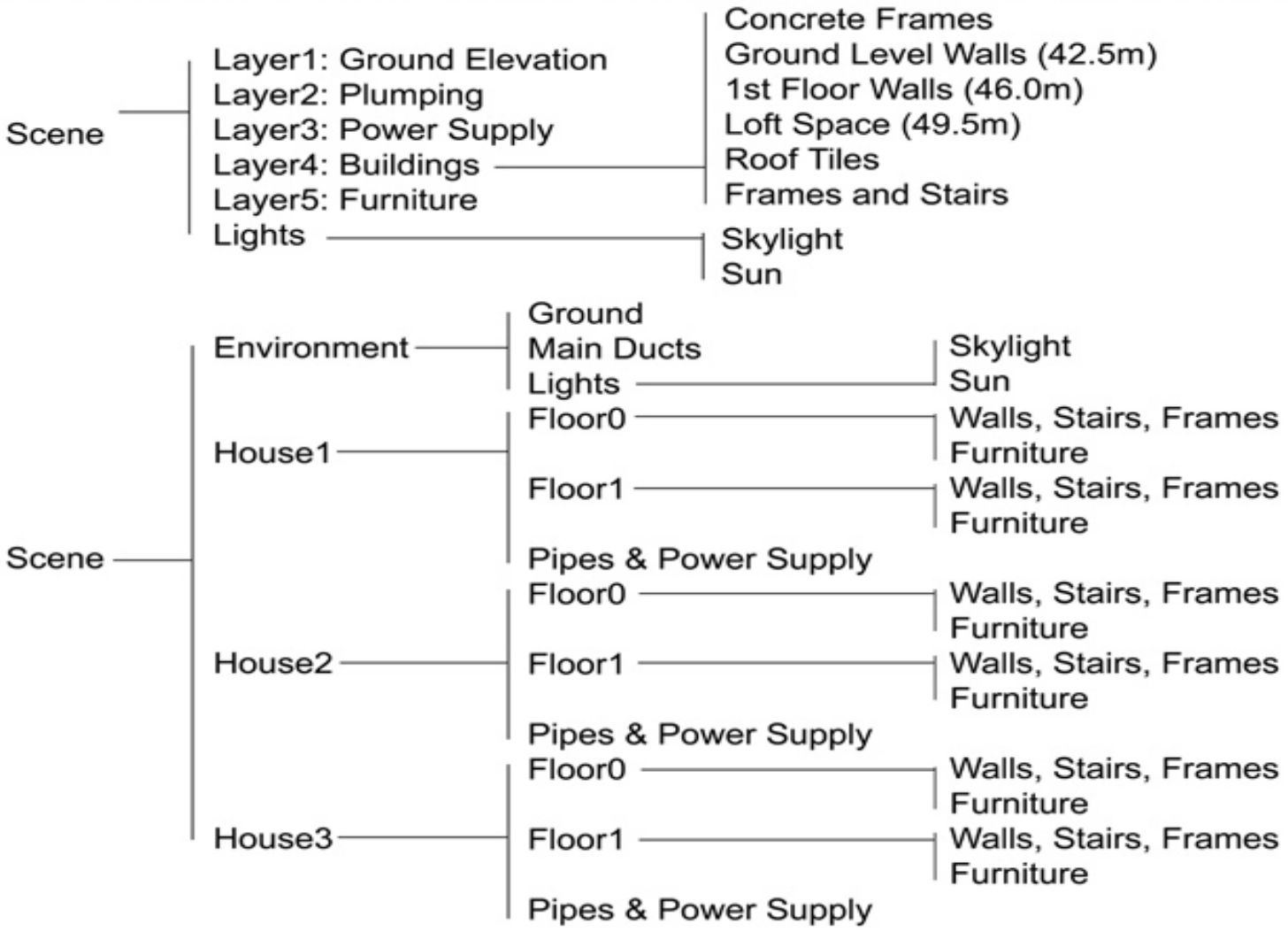
- Scene management:
  - Primitives of a scene are gathered in spatially coherent clusters
  - The clusters can be grouped in larger spatial aggregations
- Why is scene management useful:
  - All primitives are arranged in a hierarchical manner and can be efficiently accessed, removed early from operations such as viewport frustum culling, and easily managed as memory objects (dynamic loading, caching, etc.)
- When designing a virtual world we think in ontological terms and group entities according to logical relationships but:
- We can organize data in spatially coherent manner instead (*spatial partitioning)*

Introduction (2)

- Breaking the scene into spatially coherent hierarchies provides a significant performance → invisible geometry can be culled early in the process at a high hierarchy level.
- Ontological hierarchies are not necessarily optimized for spatial partitioning
- For real-time graphics applications, the common practice is to build scenes as ontological hierarchies with spatial-coherency priority.
- Grouping and hierarchical world construction also offers better asset management and memory conservation
- Decomposition of a scene can slow down rendering:
  - The application spend too much time in the traversal of the scene hierarchy
  - In hardware-accelerated graphics, a partitioning can lead to poor geometry streams and frequent state changes

Examples

Examples (2)


Scene
Layer1: Ground Elevation
Layer2: Plumping
Layer3: Power Supply
Layer4: Buildings
Layer5: Furniture
Lights
Concrete Frames
Ground Level Walls (42.5m)
1st Floor Walls (46.0m)
Loft Space (49.5m)
Roof Tiles
Frames and Stairs
Skylight
Sun
Scene
Environment
Ground
Main Ducts
Lights
Skylight
Sun
House1
Floor0
Walls, Stairs, Frames
Furniture
Floor1
Walls, Stairs, Frames
Furniture
Pipes & Power Supply
House2
Floor0
Walls, Stairs, Frames
Furniture
Floor1
Walls, Stairs, Frames
Furniture
Pipes & Power Supply
House3
Floor0
Walls, Stairs, Frames
Furniture
Floor1
Walls, Stairs, Frames
Furniture
Pipes & Power Supply

Examples (3)

- Indoor environments are constructed with portal culling and BSP trees
- Outdoor scenes provide hierarchies with a large branching factor for efficient frustum culling

Scene Graphs

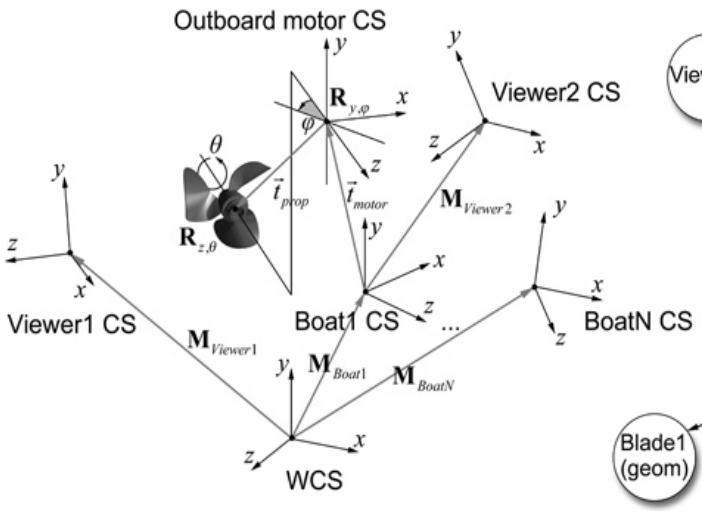
- *Scene graph (SG)*: a hierarchy of geometric elements that are related in an ontological manner and are spatially dependent
- It consists of nodes that can represent:
  - Geometric elements (static or animated)
  - Aggregations of nodes
  - Transformations
  - Conditional selections
  - Other renderable entities (e.g. sounds)
  - Pure simulation task nodes
  - Other scene graphs
- SG is a directed, non-cyclic graph of nodes, whose arcs define geometrical or functional dependence of a child node to its parent
- The nodes encapsulate all the functionality required to define a behavior → they adhere to the object – oriented programming model

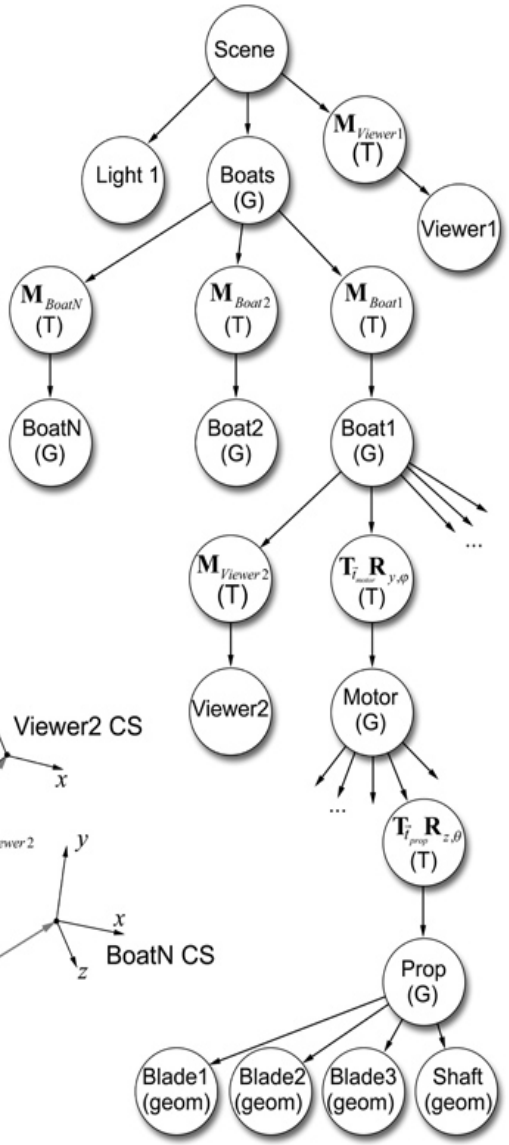
Scene Graphs (2)

- The root node is the abstract scene node:
  - Provides a single entry traversal point
  - Propagates to the hierarchy any operations needed to be performed on the elements
- An operation performed on a node affects all of its children
- Groups (node aggregations) are treated as single abstract objects:
  - Makes modeling of complex environments and their animation easy
  - Provides the means for the construction of self-contained and reusable elements
- Complex animations and behaviors are broken down into simpler ones by providing local behavior to hierarchically organized elements

Example

- The movement of each individual mechanical part of a speedboat relative to the global coordinate system is difficult to derive directly
- E.g., what is the apparent motion of the speedboat propeller relative to the person on the dock or to the driver?



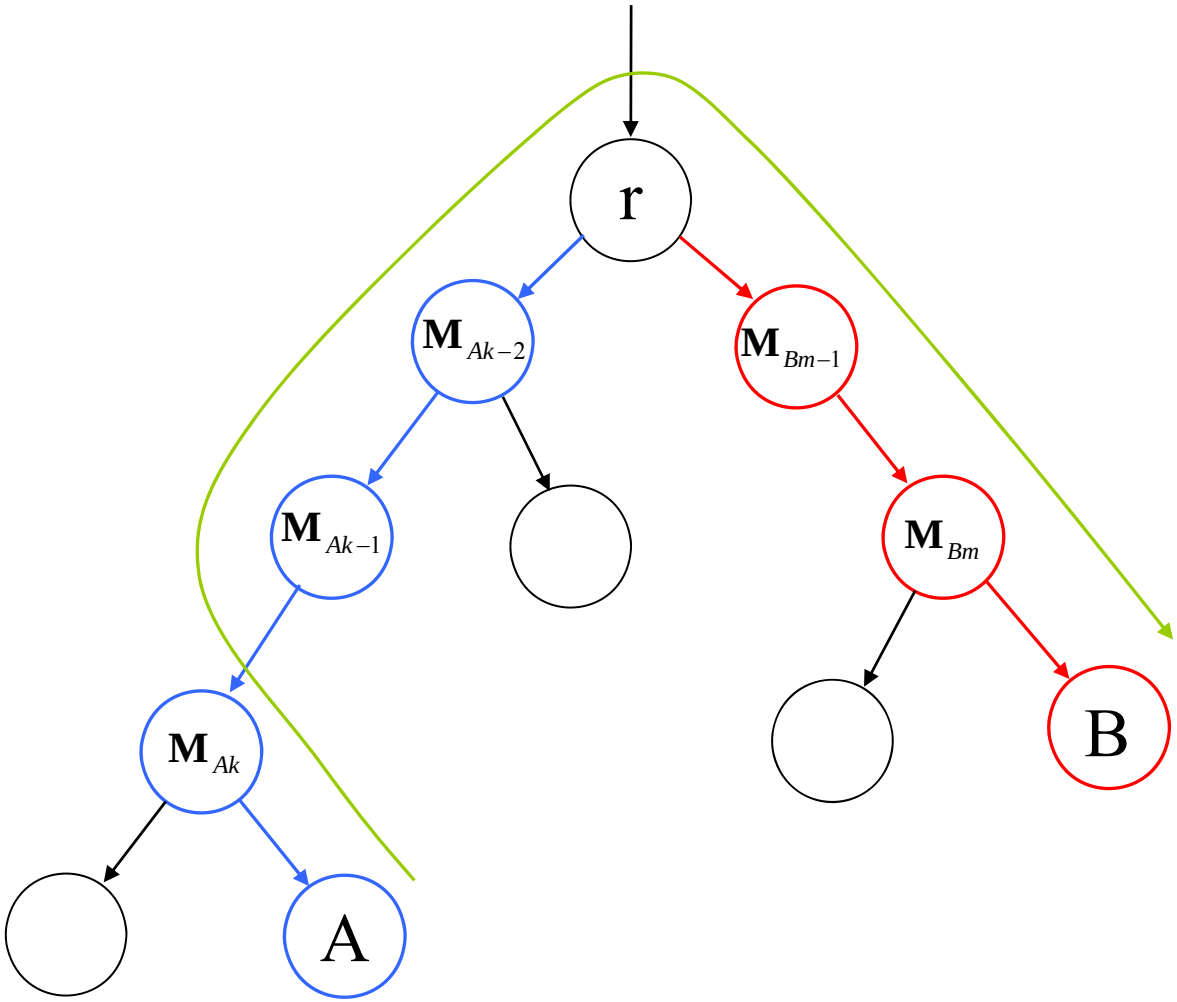


Node Relations

- In general, we express one node of the scene graph (target) relative to another (reference node) by a change of reference frame according to the intermediate transformations:
  - Perform an upward traversal of the tree from the target to the common parent node of the target and reference nodes
  - Descend to the reference node by inversely applying all transformations of this path
- The transformation of a node at level k on a branch A relative to another node at level m on a branch B with a common root at level r is:

$$\mathbf{M}_{A\to B} = \prod_{j=m}^{r+1} \mathbf{M}_{Bj}^{-1} \prod_{i=r+1}^{k} \mathbf{M}_{Ai}$$

Node Relations (2)



Node Relations (3)

- The above equation is a direct consequence of the duality between transformations and change of reference frame:
  - Move the reference frame from the common node at level *r* to that of reference node
  - As the reference node is transformed by $\mathbf{M}_{Br+1} \cdot \mathbf{M}_{Br+2} \cdot \ldots \cdot \mathbf{M}_{Bm}$ with regard to the common root at level *r*, node *r* and everything depending on it are inversely transformed to reflect this change of basis

Data Organization

- The majority of a scene graph describes relations between:
  - Ontological entities
  - Aggregations of nodes
- Geometric data are essentially leaves of the hierarchical structure
- Geometry nodes may contain:
  - Data
  - Other primitive data information (NURBS surfaces, volume data)
- The data of the nodes may be:
  - Unordered (raw)
  - Indexed

Hybrid Organizations – Non R-T Rendering

- Geometry nodes may be part of (or self-contained) space-partitioning structures (AABB trees)
- This scheme is frequently used in applications using large datasets

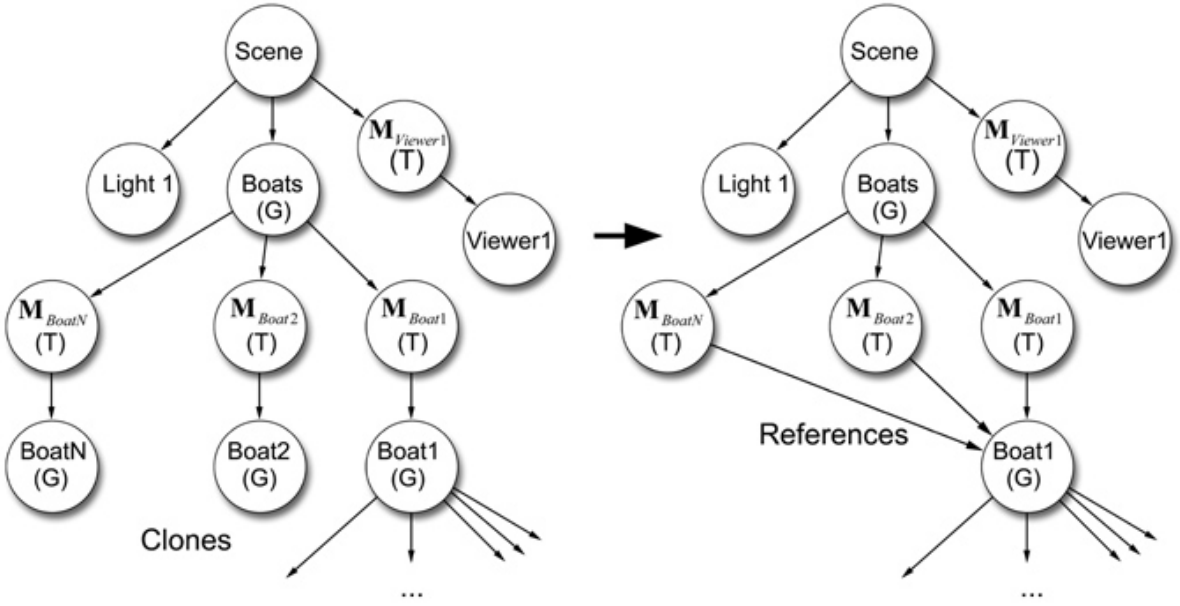
- Scene graph:
  - Represents the ontological hierarchy of the data
  - Encapsulates the functionality of each entity
- A space-partitioning system operates on the leaves accelerating:
  - Rendering
  - Search operations
- Since efficient space partitioning requires that data are pre-processed and stored in appropriate structure → scene graph - space partitioning approach is effective for static data

Hybrid Organizations – R-T Rendering

- A scene graph is decoupled from the space-partitioning scheme and used for animated objects only
- All static environment information is isolated in a high-performance spatial-partitioning system (BSP tree)

Node Instancing

- Why one should actually replicate the node to create separate copies of the same entity when the latter are identical?
- A reference is created to the original node, wherever an identical node needs to be inserted into the scene graph
- No copy of the data attached to the new node is made
- When instancing is used, the tree-like structure of the scene graph is transformed into a directed cyclic graph



Node Instancing (2)

- Creating instances of nodes instead of copies saves storage space
- It also speeds up the calculations if certain simulation steps; processing is performed once for the original node and all instances reuse the new data
- Traversal order is not important in instancing:
  - When the node's data are accessed for the first time in frame *k+1*, the node is marked as "processed‘" for the current time stamp
  - Subsequent visits to the node (directly or via reference) check the frame counter and skip all calculations
- Node instancing on an aggregate or animated node means that internal behavior is identical for all clones, which may not be desired (e.g. for crowd simulation)

Scene Graph Traversal

- Main advantage of a scene-graph representation of a 3D world:
  - Every operation can be applied to the root node of the hierarchy and propagated to the rest of the entities via scene-graph traversal
- 4 major operations performed on a scene graph:
  - ***Initialization***
  - ***Simulation***
  - ***Culling***
  - ***Drawing***
- Each operation is applied hierarchically on the nodes
- **Simulation** procedure:
  - Is responsible for determining all internal node parameters
  - Is responsible for performing variable updates according to a node-specific behavior
  - Is referred to as the *animation* or *application* stage :
    - Emphasizes either the visual impact of the node's change of state or the fact that this procedure is decoupled from the rendering algorithms

Scene Graph Culling

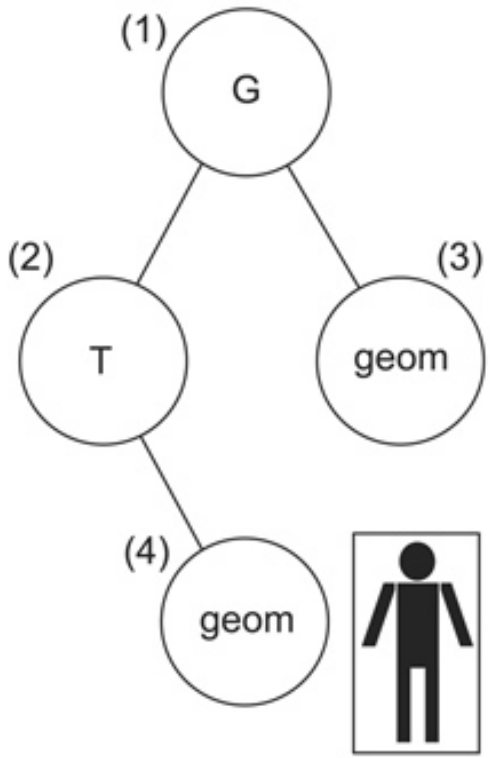
- **Culling** in scene graphs is tightly coupled with the node dependencies
  - Each node is assumed to "contain" all of its children:
    - If a subtree root node is marked as completely hidden → every child node is also invisible → the whole subtree is pruned
  - If an aggregate node passes the visibility test → its children are individually tested → in the case of partial parent-node occlusion, some children may be invisible
  - Result of the culling process is determined by testing the bounding volume of a node with the chosen criterion
  - For an aggregate node, the bounding volume reflects the collective extents of its children → **Bounding volume hierarchies**

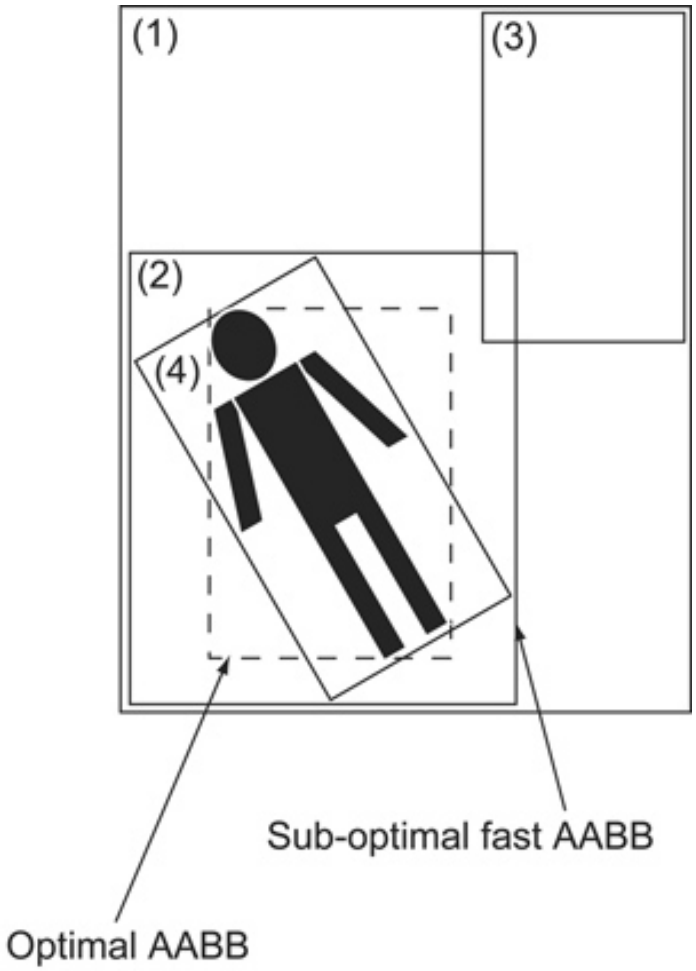
Bounding Volume Hierarchies

- If a node contains an animated branch of the hierarchy, its extents need to be dynamically adjusted each time the extents of one of its children change
- Recalculation of a geometry node‘s exact extents requires iteration through all its vertices to determine min & max values, or moments & principal axes
- For animated subtrees that need to iterate through the raw data anyway (i.e. skeletal animation), this imposes no additional overhead
- For rigid-body animation, the reevaluation involves substantial processing time and may be prohibitive for large models.

Bounding Volume Hierarchies (2)

- A practice adopted when speed is a more important factor than exact visibility is to adjust the bounding volume of an aggregate node based on the transformed, object-aligned bounding volumes of its children





Bounding Volume Hierarchies (3)

- The above solution is suboptimal as the extents of the transformed bounding volumes are in general larger than the extents of the contained geometry
- Culling at a fine level can be handled by the spatial subdivision scheme of the geometry nodes (if present)

Scene Graph Drawing

- The **drawing** operation:
  - Recursively moves down the hierarchy and applies the rendering algorithms to each visible renderable node
  - In direct rendering and first-level ray-casting the pruned subtrees are not traversed

Programming with Scene Graphs

- In a scene graph, most nodes tend to be self-managed → provide their own drawing functionality & behavior
- An aggregate node (not renderable) needs to propagate operations such as drawing and culling its children
- This distributed operation of a scene graph:
  - Facilitates an object-oriented design of the nodes
  - Takes advantage of polymorphism and abstraction

Programming with Scene Graphs (2)

- All nodes of a scene graph are derived from a common generic node class & mutate their behavior as a node inherits common attributes and overrides the behavior of another node:

```
class Node
{
    protected:
        bool active;
        bool culled;
    public:
        Node();
        virtual void init();
        virtual void simulate();
        virtual void cull();
        virtual void draw();
        virtual void reset();
};
```

Programming with Scene Graphs (3)

```
class Group : Node
{
    protected:
        vector<Node*> children;
        Bvolume extents;
    public:
        Group();
        void add(Node *n);
        void remove(int i);
        Node * getChild(int i);
        int  getNumChildren();
        virtual void init();
        virtual void simulate();
      virtual void cull();
      virtual void draw();
};
```

Programming with Scene Graphs (4)

- Nodes extending abstract class `Node` inherit the four basic operations
- Aggregate nodes are derived from the `Group` class & share a common functionality:
  - They provide a list of children and basic operations on them
- More elaborate `Group` subclasses may need to extend this behavior by adding more specific methods, i.e. *Transformation node* or a *Selector node* (activates one child at a time)
- A common feature of all `Group` nodes is the traversal of their children:
  - Is manifested as an iterative call to all available `Node` objects in their internal list
- Due to polymorphism, a `Group` object can invoke the method `init()`, `draw()`, `cull()`, or `simulate()` without caring what subclass the particular object is instantiated from

Programming with Scene Graphs (5)

- All scene-graph node type classes share a common interface:

```
void Group::draw()
{
    vector <Node>::size_type i, sz;
    sz = children.size();
    for (i=0; i<sz; i++)
        children[i] -> draw();
}
```

```
void Geometry::draw() // Geometry is a subclass of Node
{
    if (!enabled || culled)
        return;
    // ... render the geometry
}
```

- Culling, initialization, and simulation steps are similarly defined

Programming with Scene Graphs (6)

- Traversal pattern of scene graph defines order of invocation of each method at runtime
- Initialization and the 3 repetitive steps are executed in a scene-graph basis
- After initialization the typical duty cycle of a scene graph involves:
  - The invocation of the simulation
  - The culling
  - The drawing methods of the root node
- Therefore, 3 traversals occur before the scene graph is visualized

Programming with Scene Graphs (7)

```
// Scene is a subclass of Group
Scene *myScene = new Scene();
myScene -> load("village.scn");
myScene -> init();
while (notTerminating)
{
    // ... other operations such as user input
    myScene -> simulate() ;
    myScene -> cull() ;
    myScene -> draw() ;
}
```

- This keep all nodes in pace with each other in terms of status and parameter values and leads to the more efficient *deferred update* strategy.

Deferred Data Update

- In this strategy, changes in the state of an entity do not produce immediate result:
  - Instead, all attributes are evaluated just once to produce a simulation, culling, or drawing result
- Deferred updates are useful when a computationally heavy operation needs to be repeated whenever a variable changes
- Practical examples in the scene-graph paradigm:
  - Various geometry-dependent culling techniques (occlusion culling)
  - Physically based animation
  - Estimation of shadow volumes

Message Passing

- Complex scene graphs → we need to exceed the limitations of strict hierarchical control.
- Direct communication: Nodes may communicate with each other
  - By direct method invocation
  - By *message passing*
- In the latter (more elegant and easy to synchronize) case, each node primitive needs to be extended to support a message queue and possibly an event map:

```
class NodeMessage
{
    Node *from, *to;
    int ID;
    void * params;
};
typedef EventID int;
```

Message Passing (2)

```
class Node
{
    protected:
        ...
        vector<NodeMessage *> msgQueue;
        multimap<EventID,NodeMessage *> eventMap;
        //Remove all pending messages and invoke appropriate methods
        void processMessages();
        //Notify other nodes according to events registered in the event map
        void dispatchMessages();
    public:
        ...
        message( NodeMessage *msg ); //add msg to the queue
        registerEvent( EventID evt, Node* target,
                       int msgID, void* params );
};
```

Message Passing (3)

- Message queue is necessary → a node may receive multiple command messages from an unknown number of other nodes
- An event map:
  - Creates an interface for user-defined responses to node state changes node (useful for trigger nodes)

Message Passing Example

- Consider a room full of furniture.
  - The light is initially turned off → no need for the furniture to cast shadows → initially disabled for these geometry nodes
  - When the light is turned on → furniture geometry needs to start casting shadows
- We make a halo object visible around the bulb to make the scene realistic

```
Light * bulb; //Light extends Node
Geometry *furniture, *halo;
...
bulb -> registerEvent( EVENT_ON, furniture, MSG_SHADOWS_ON, NULL);
bulb -> registerEvent( EVENT_ON, halo, MSG_ENABLE, NULL);
```

Message Passing (4)

- In the extended Node class 2 new protected member functions are added:
  - processMessages()
  - dispatchMessages()
- These operations are executed before & after simulation step respectively
- Need to introduce additional *pre/post*-simulation methods → will be invoked via a corresponding pre/post simulation step for the whole scene:

```
void Scene::simulate()
{
    preSimulate();          Group::simulate();          postSimulate();
}
...
void Group::preSimulate()
{
    vector <Node>::size_type i,sz;     sz = children.size();
    for (i=0; i<sz; i++)
        children[i]->preSimulate();
}
```

Pre/Post Node Processing Steps

- We need to introduce additional *pre/post*-simulation methods → will be invoked via a corresponding pre/post simulation step for the whole scene:

```
void Scene::simulate()
{
    preSimulate();
    Group::simulate();
    postSimulate();
}
...
void Group::preSimulate()
{
    vector <Node>::size_type i,sz;     sz = children.size();
    for (i=0; i<sz; i++)
        children[i]->preSimulate();
}
```

Pre/Post Node Processing Steps (2)

- Similar pre/post operation function calls implemented for the draw and culling stages:
  - Either locally for each node (invoked before & after the corresponding operation on each node)
  - Or globally (as in pre- and post-simulation stages)

Examples:

- When rendering with OpenGL, a *Transformation node* needs to implement:
  - a pre-draw function to push the current matrix state in the stack
  - a post-draw function to pop the current matrix state in the stack
- A global post-draw function may trigger a buffer swap

Distributed Scene Rendering

- Demanding or multi-user/multi-display applications require distributing the scene-graph data and rendering to multiple processing units
- A processing unit can be:
  - A separate computer
  - A specialized co-processor
  - One or more parallel system processors
  - A scalable graphics subsystem
- The scene is not necessarily resident in a common space in memory

Distributed Scene Rendering (2)

- Data transfers between processing units occur at a wide range of bandwidth limitations
- A drawing operation of a scene can be split in 3 ways
  - In the spatial domain
  - In the time domain
  - In the image domain
- The procedure for rendering a single frame of the synthetic imagery consists of four stages:
  - Splitting
  - Distribution
  - Rendering
  - Compositing

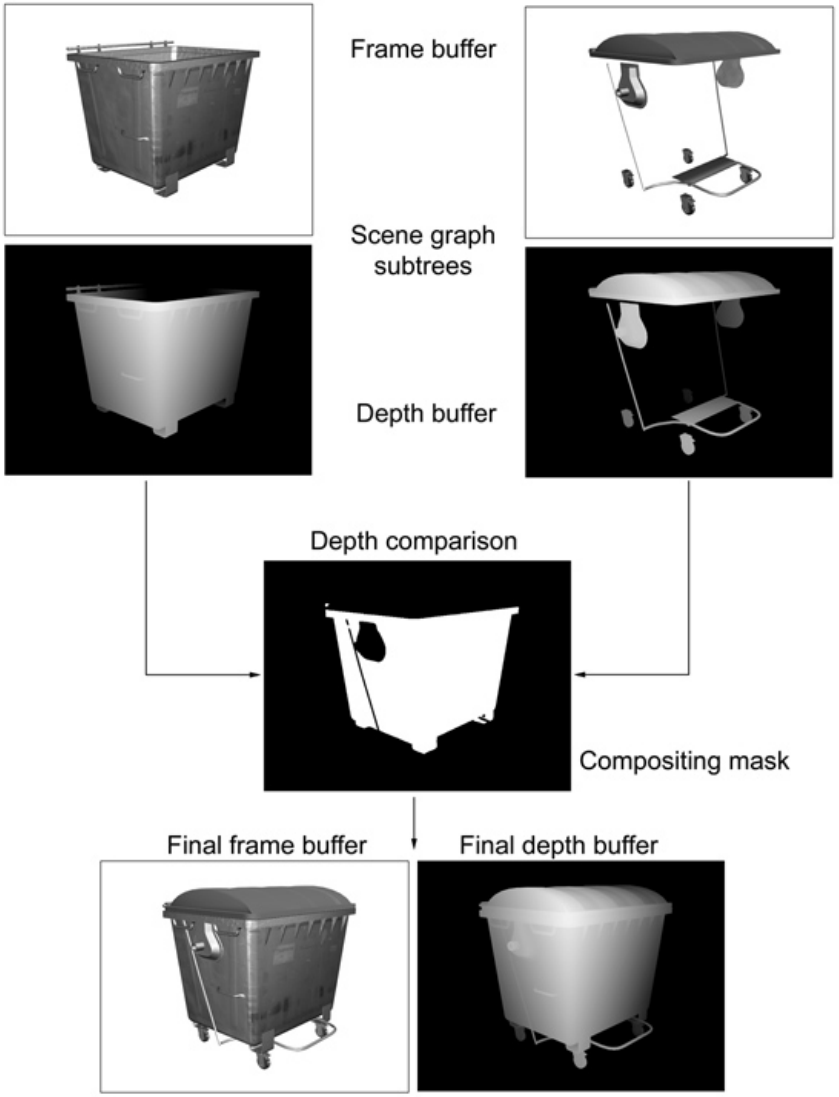
Distributed Rendering – Sort-last

- When distributing the rendering of a scene among processing units in the spatial domain
  - a portion of the scene is transferred to each unit
  - it is rendered independently
  - then composited to form a unified, final result
- The scene is divided according to the hierarchy of a scene graph or a spatial subdivision scheme
  - the tokens are distributed among the available units
  - each unit renders a partial result
  - Then the result needs to be combined with the output from its siblings.
- In the case of direct rendering:
  - The resulting partial images are unordered
  - They overlap in the image domain
  - Resulting frame buffers cannot be combined
  - Usual practice is to maintain and transmit the depth buffer of each partial rendering as well

Distributed Rendering – Sort-last (2)

- A unit plays the role of the *compositing engine* → gathers the results and combines the partial color based on the fragment depth-buffer comparison of the incoming images and the transparency

- Distributed rendering schemes like this, are called *sort-last* because the decision for which part of the image is attributed to which node occurs at the end of the frame generation



Distributed Rendering – Sort-last (2)

- A *data-parallel* rendering approach is also possible for ray tracing:
  - The scene database is distributed among the units
  - A server node1 casts rays
  - The rays are then redirected to the appropriate rendering node(s) to calculate the ray-geometry intersection.
- This is also a *sort-last* approach:
  - The resulting intersection tests from all rendering units need to be sorted according to distance from the starting point of a ray
- The distribution of rays among rendering units is a parallel task → this architecture is suitable for tightly-coupled parallel systems.

Distributed Rendering – Sort-first

- *Sort-first* schemes:
  - Perform a pre-partitioning of the target output space
  - Each rendering unit is assigned one or more chunks
- Composition of the rendered pieces is trivial in these algorithms → the gathered image chunks have no overlap
- In the case of offline rendering of animations:
  - Each processing unit maintains a full copy of the scene database as well as external assets (textures), and independently draws a complete frame image.
  - Trivially parallel: No communication apart from initialization and gathering of resulting frames.

Distributed Rendering – Sort-first (2)

- Image-domain sort-first strategies are very common:
  - The scene database is replicated among the rendering units, or shared by multiple processes that perform the rendering.
  - Each unit is assigned one or more “windows” of the final image
  - The results are composited by copying the prepared image segments into a common buffer
  - Direct distributed rendering in multiple graphics systems on the same machine is handled by the hardware of the graphics display boards

Sort-first Partitioning

- Partitioning strategies in image domain play a significant role in efficient load balancing:
  - Common split methods are:
    - interlaced scan-line
    - Tiled
    - offset full-image

Sort-first Partitioning (2)

- Larger segments → higher probability that the workload won't be balanced evenly among the rendering units
- A partitioning that splits the image in even and odd scan-lines would ensure the best load balancing.

- For real-time rendering:
  - an important factor for choosing a partitioning scheme is the incremental nature of the rasterization process.
  - Spatial coherence and sampling in a regular pattern is beneficial for the rendering stages of the graphics subsystem
- Using the post-filtering antialiasing technique to render an image:
  - interesting strategy: distribute the sampling kernel among the rendering units.
  - done by rendering the full frame on each unit but with a fragment center offset that corresponds to the sample offset of the multisampling matrix
  - Resulting fragments are then weighted to produce the final image.