AUEB COMPUTER GRAPHICS GROUP

COMPUTER GRAPHICS COURSE

Scene Organization



Andreas A. Vasilakis

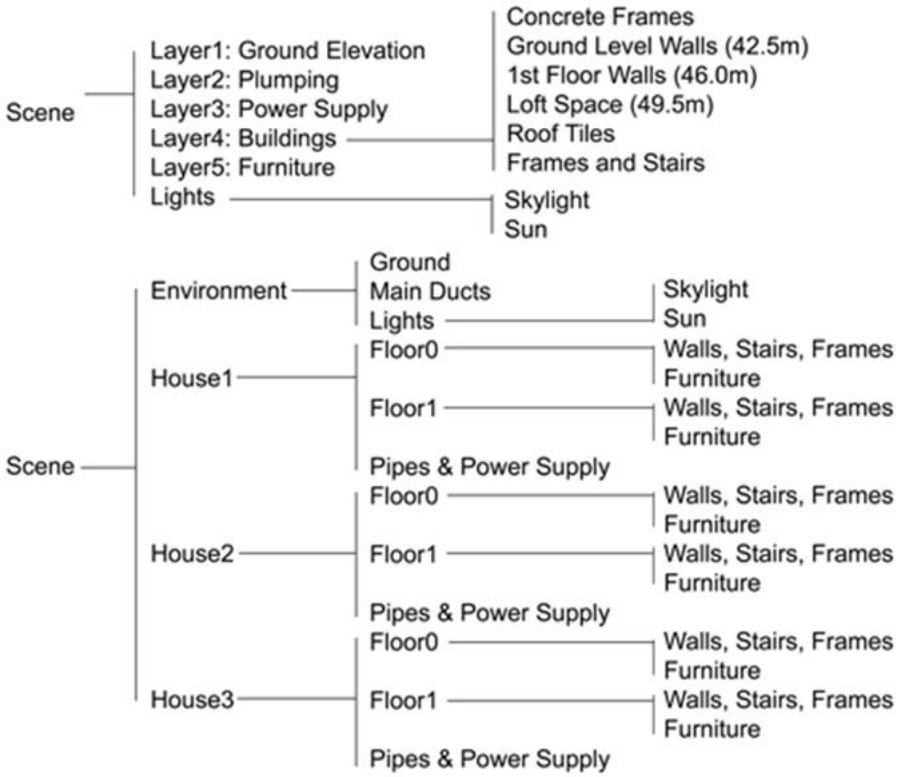
Scene organization

- Primitives (building blocks) of a scene are gathered in spatially coherent clusters
- The clusters can be grouped in larger spatial aggregations based on:
  - Function
  - Relational clustering
  - Spatial coherence
- Why is this useful?
  - All primitives can be efficiently accessed, removed early from operations such as viewport frustum culling, and easily managed as memory objects (dynamic loading, caching, etc.)

Types of Arrangement: Relational (1)

- When designing a virtual world we think in ontological terms and group entities according to logical relationships





Types of Arrangement: Relational (2)

- This type of arrangement is a user-centric approach to scene management
  - Useful for modelling a virtual environment (scene)
  - Helps as synthesize larger constructions from small parts
  - Promotes reusability of the building blocks
- The same environment can be modelled with different approaches (see example in previous page)
- This object hierarchy is the basis for the scene graph

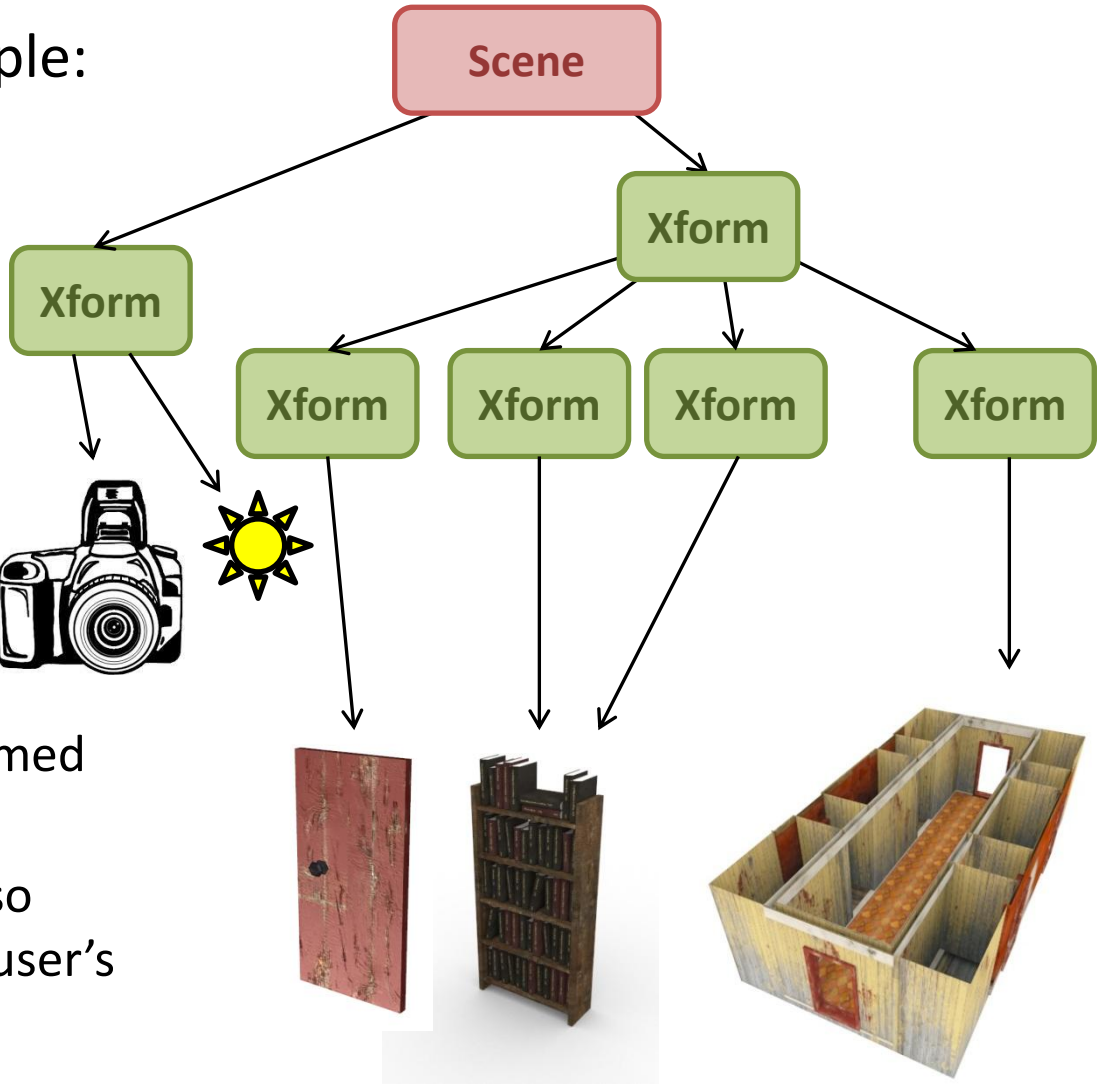
Types of Arrangement: Spatial

- We can organize data in spatially coherent manner instead (spatial partitioning)
- Provides a significant performance improvement → invisible geometry can be culled early in the process at a high hierarchy level
  - Ontological hierarchies are not necessarily optimized for spatial partitioning
  - Static geometry is represented ontologically and subsequently organized in spatial hierarchies for speed

Transformation Hierarchies (1)

- Consider this simple example:

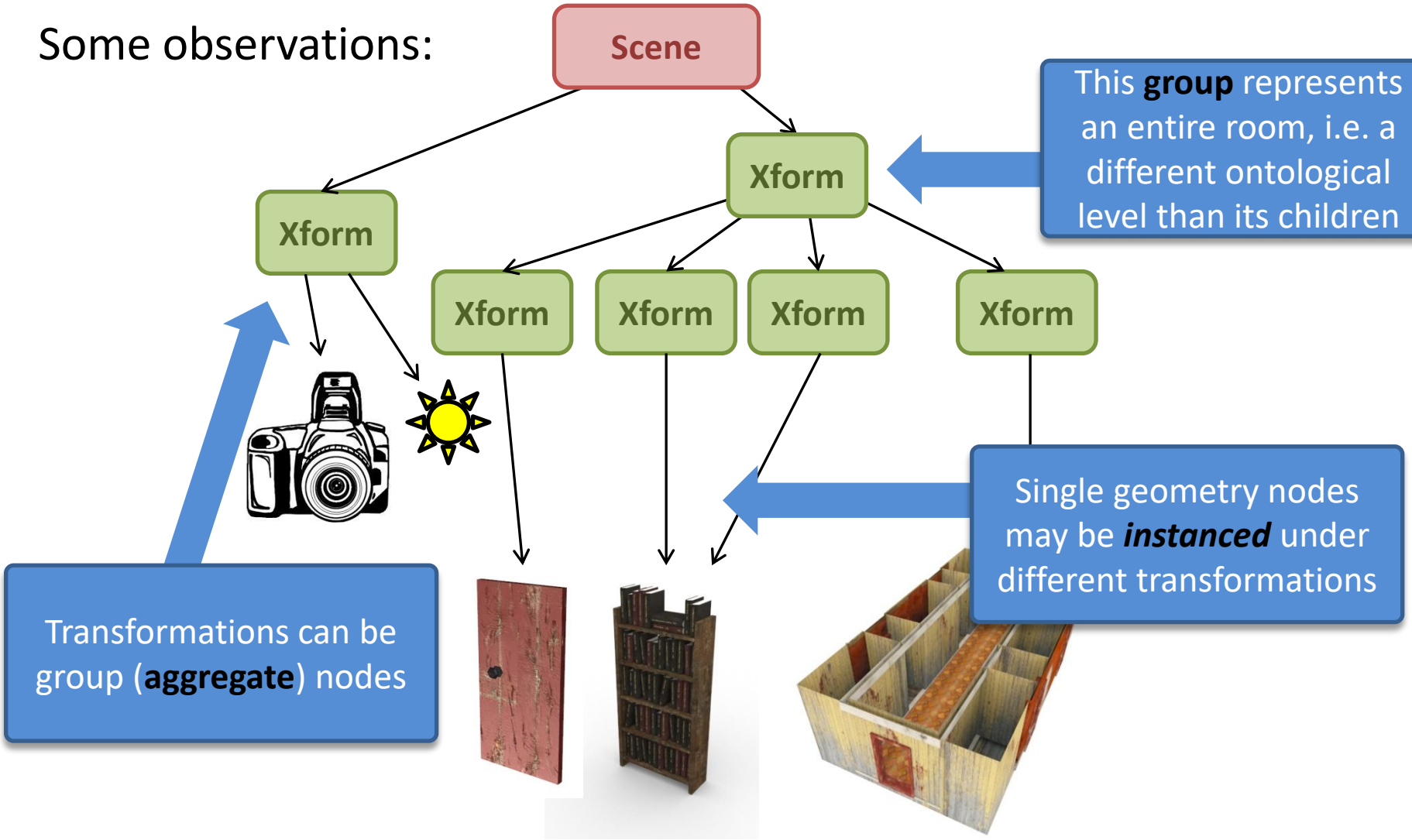




  - Uses appropriately transformed objects
  - The camera and light are also transformed, following the user's movement

Transformation Hierarchies (2)

Some observations:



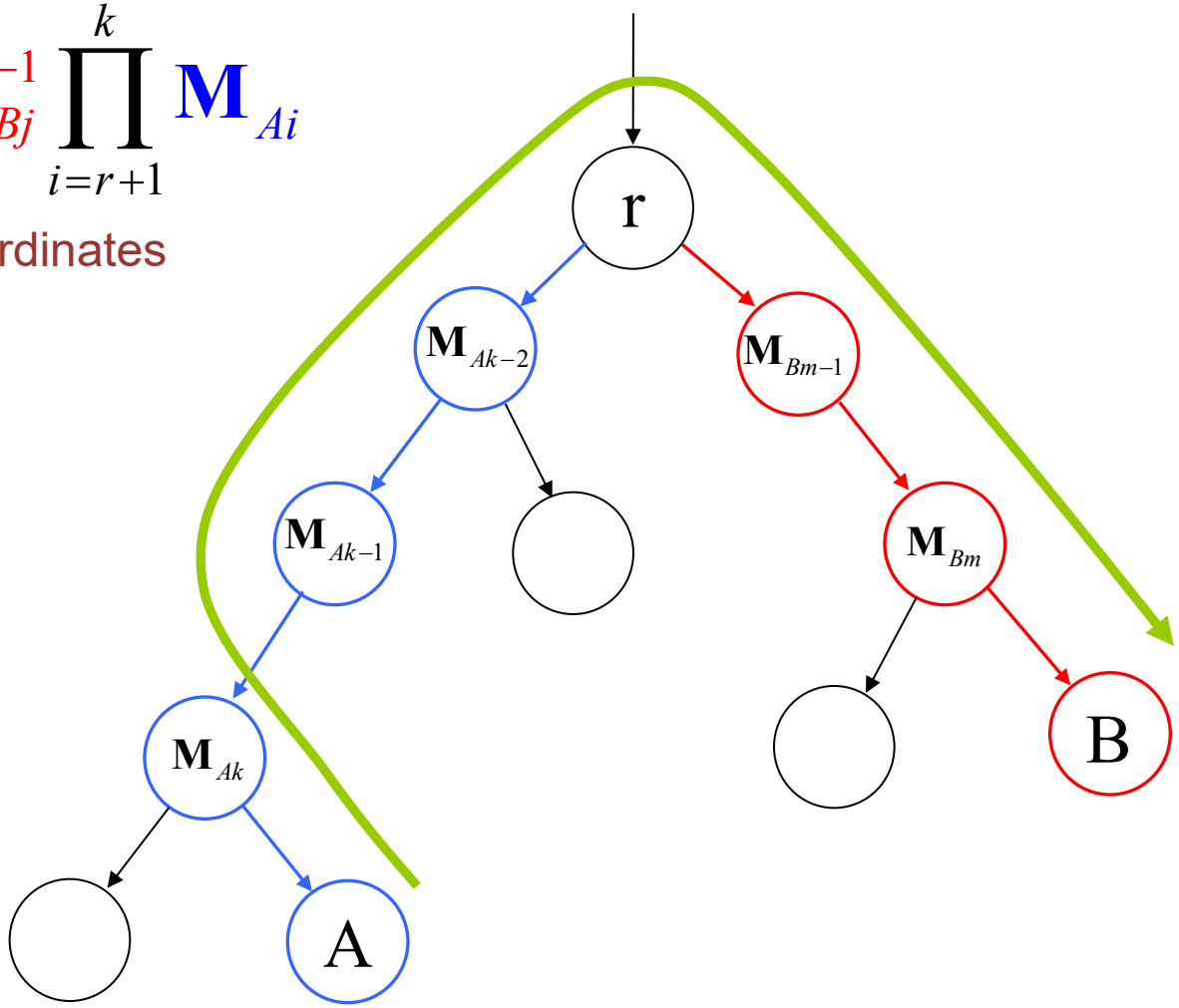
Traversing the Hierarchy (1)

- We express one node (target) relative to another (reference) 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

Traversing the Hierarchy (2)

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

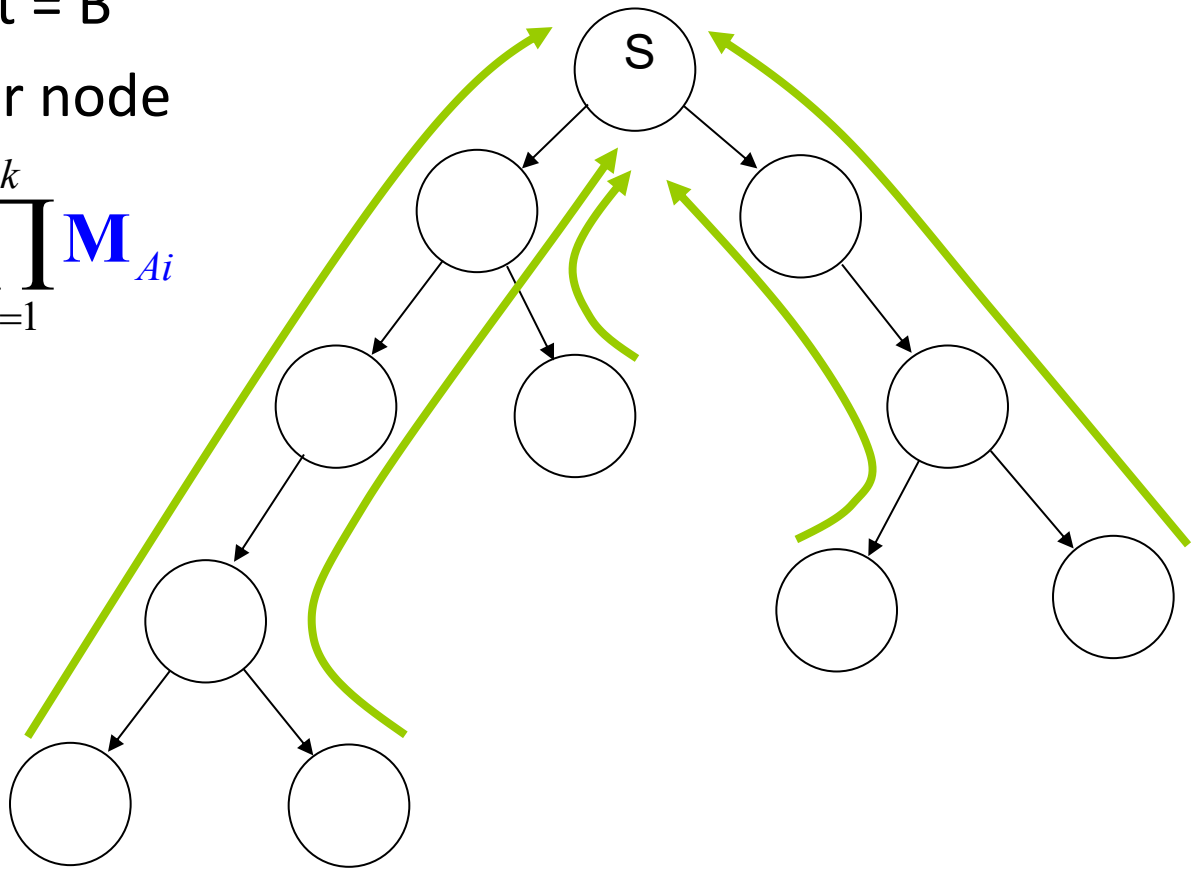
Express A w.r.t B coordinates



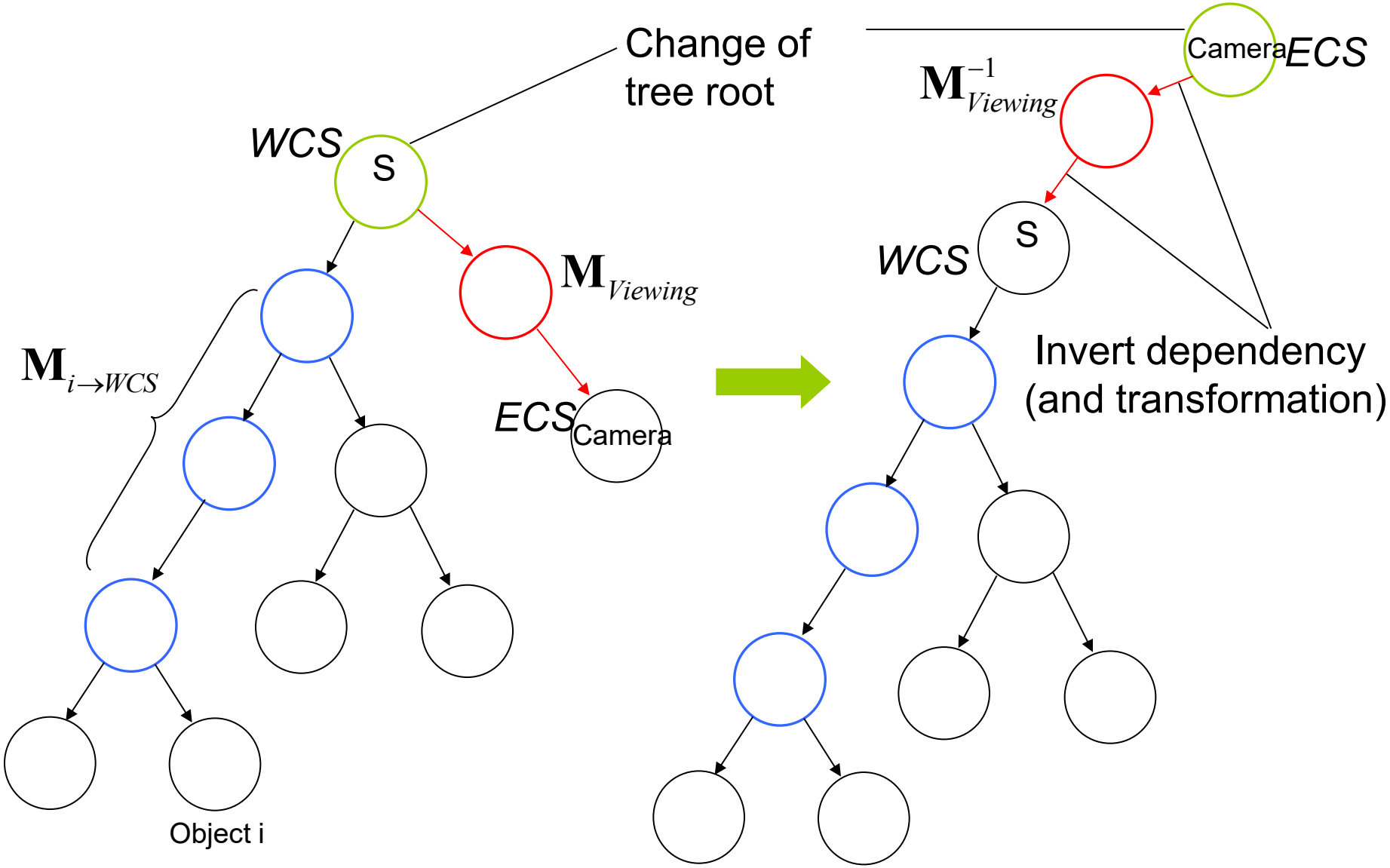
Traversing the Hierarchy: Example 1

- Expressing all nodes w.r.t the WCS
- r = Scene root = B
- A: every other node

$$\mathbf{M}_{A\to WCS} = \prod_{i=1}^{k} \mathbf{M}_{Ai}$$



Traversing the Hierarchy: Example 2



The Scene Graph

- A DAG representing the scene entities and their relationships
- Entities:
  - Geometry instances
  - Groups
  - Transformations
  - Virtual cameras and Lights
  - Dynamic nodes (triggers, iterators, etc)
  - Other “renderable” entities, such as sounds
- Content of the scene graph may be indirectly indexed in other data structures (e.g. spatial hierarchies)

Scene Graphs: Functionality

- Usually scene graphs perform the following tasks:
  - Maintain and manage the relationships between the scene entities
  - Perform all per-frame state calculations, i.e. advance the current values of all node variables
  - Perform any culling operations (prune invisible node sub-trees)
  - Trigger events and dispatch messages for node interaction
  - Provide a traversal mechanism for drawing the entire scene

Scene Graphs: Basic Node Examples (1)

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

```
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();
};
```

Scene Graphs: Basic Node Examples (2)

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

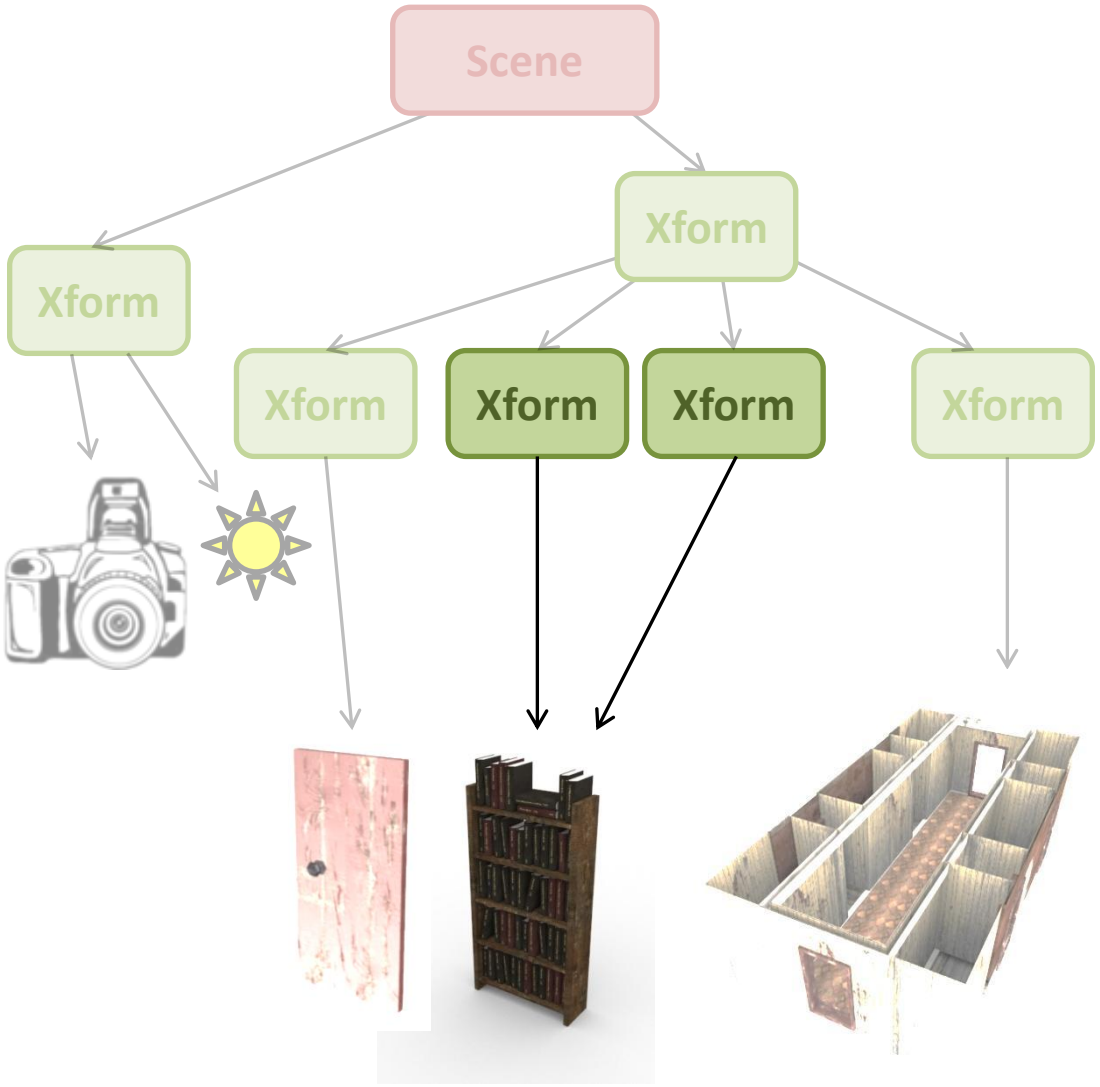
```
void Geometry::draw() // Geometry is a subclass of Node
{
    if (!enabled || culled)
        return;
    // ... render the geometry
}
```

Scene Graphs: Life Cycle

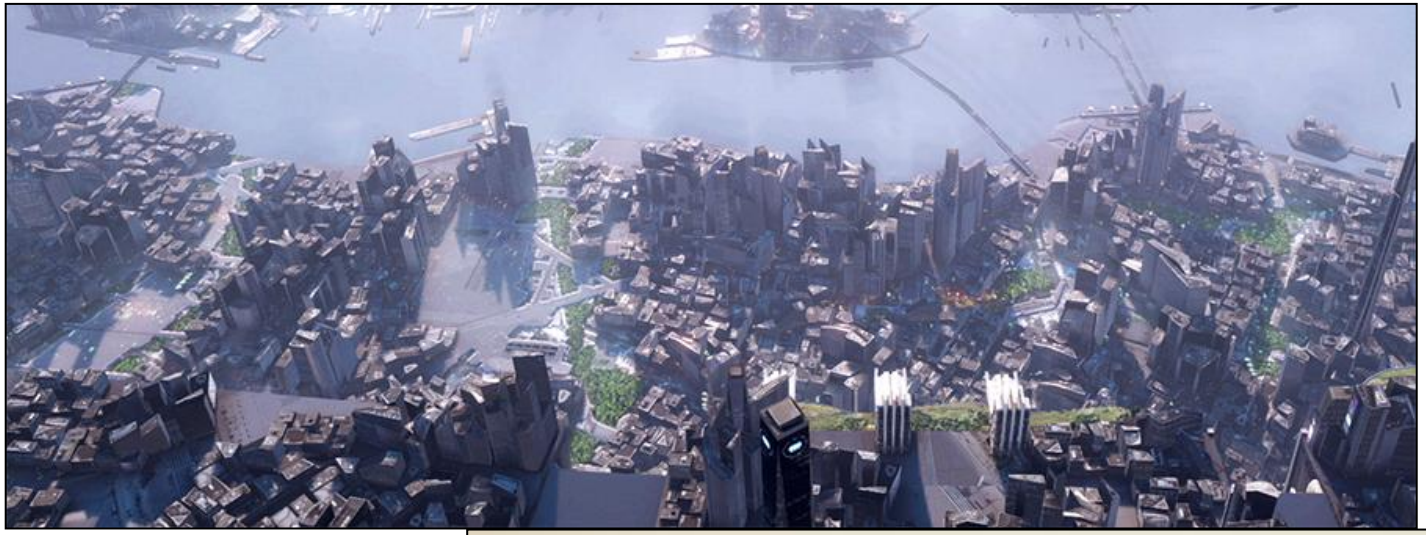
```
// Scene is a subclass of Group
Scene *myScene = new Scene();
myScene->load("village.scn");
myScene->init();

while (notTerminating)
{
    // ... other operations such as user input
    myScene->simulate() ;
    myScene->cull() ;
    // ... Rendering engine frame setup
    myScene->draw() ;
    // ... Rendering engine frame post processing
    // GUI, etc.
}
```

Node Instancing

- Node instancing helps reduce the storage and bandwidth requirements for the geometry representation by avoiding the replication of geometry data



Node Instancing: Use Cases



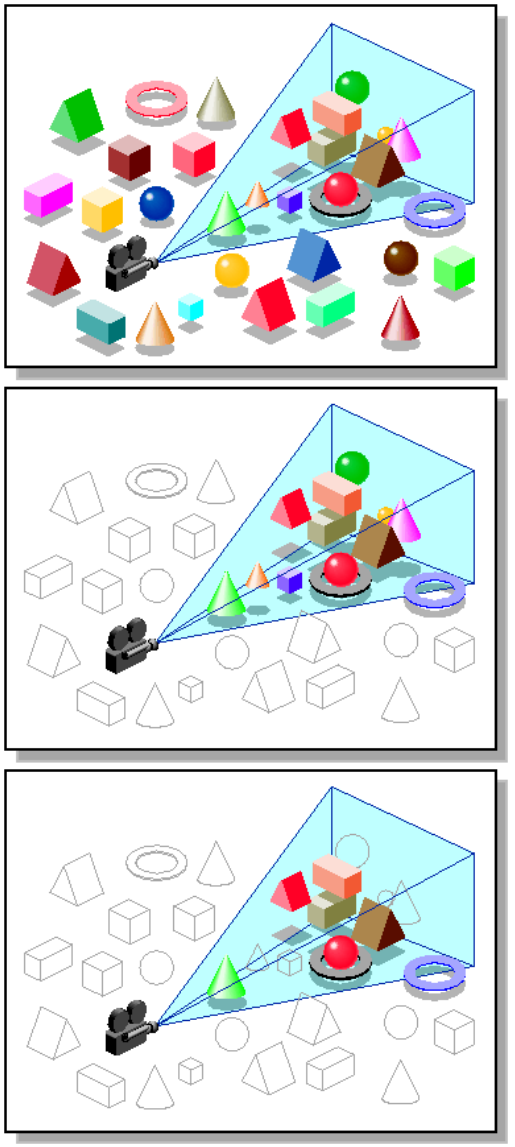
Killzone: Shadow Fall

Octane renderer



Accelerating Visibility Queries (1)

- In cluttered or large environments, it is important to efficiently discard objects that are not going to appear in the view frustum either because:
  - They are not contained in the frustum (frustum culling)
  - They are hidden behind other objects (occlusion culling)



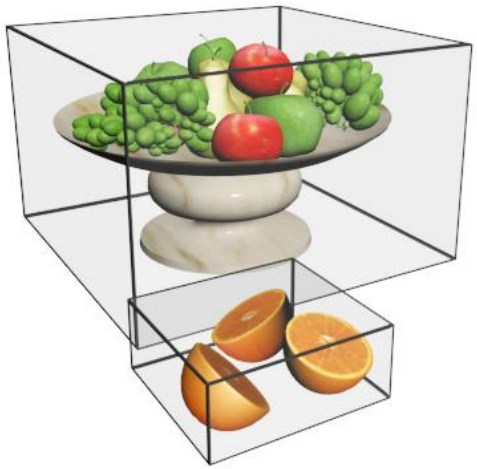
Accelerating Visibility Queries (2)

- To perform such queries, we must do so efficiently, i.e. not at the detailed geometry level, but rather using the objects' bounding volumes or better, the bounding volume of aggregations of objects

- Many more opportunities arise for culling, especially in indoor environments (portal-based culling)

Bounding the World (1)

- It is much more efficient to perform visibility (and other) queries on object bounds instead of the objects themselves
  - Provided that bounds are efficiently represented: boxes, spheres or other easy to calculate and transform shapes



Bounding the World (2)

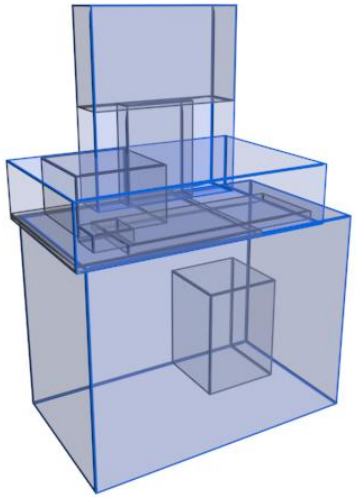
- Bounding Volumes provide a conservative query mechanism
  - They definitely enclose and contain the geometry
  - But… may leave too much void space

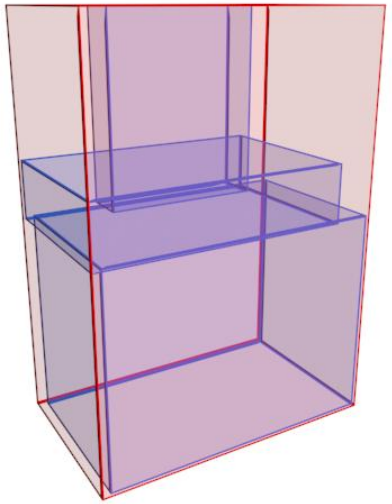


Bounding Volume Hierarchies

- A Bounding Volume Hierarchy, commonly abbreviated as BVH, can organize several bounding volumes into a hierarchy

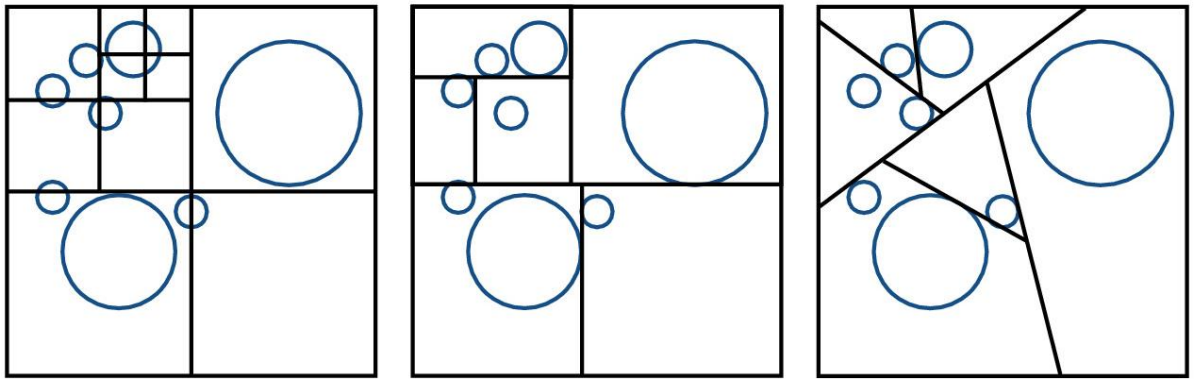






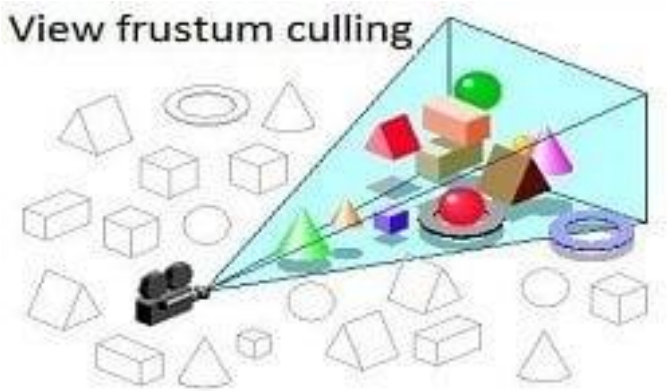
Spatial Subdivision Structures

- Octrees
- BSP Trees
- K-d Trees



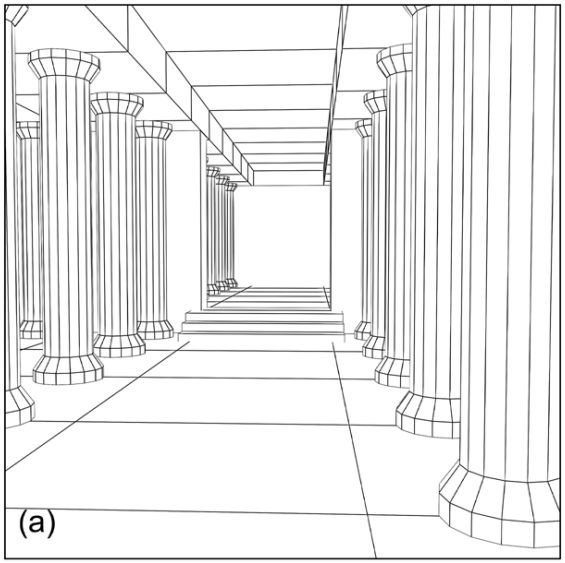
Frustum Culling

- Objects and object hierarchies whose bounds are outside the frustum can be eliminated prior to a drawing traversal

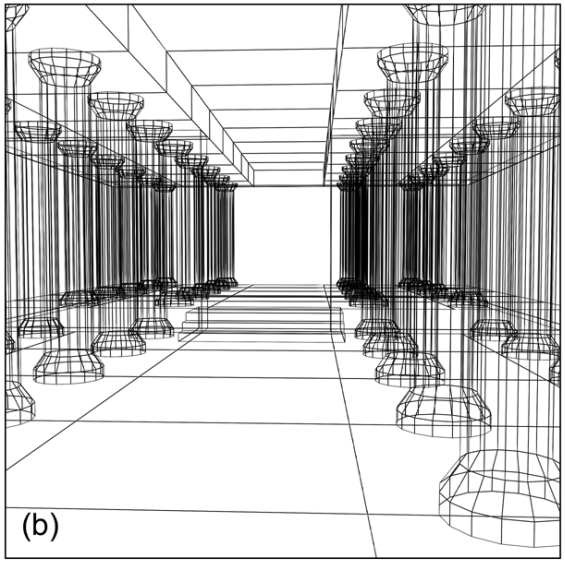


Occlusion Culling (1)

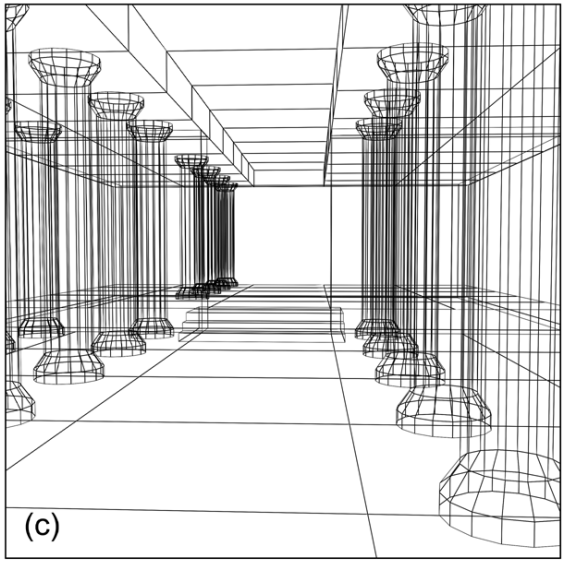
- In occlusion culling, we try to conservatively eliminate objects that are hidden from view behind other geometry
  - We try to guess a potentially visible set of objects



Visible geometry



All geometry in frustum



Potentially visible set

Occlusion Culling (2)

- To perform occlusion culling we:
  - Determine geometry to act as a set of occluders (obstacles behind which other geometry can be potentially hidden)
  - Provide a mechanism to mark and cull hidden geometry
- Occlusion determination can be done:
  - Analytically, via frustum culling
    - Create a frustum using each occluder geometry and the eye position
    - Perform frustum/object bounding box containment test
  - In screen space (see next)

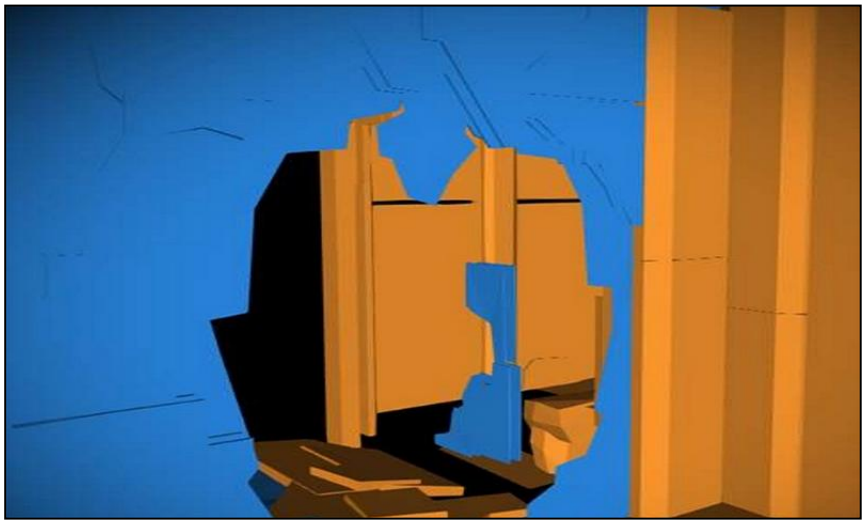
Occluder Determination (1)

- Automatic occluder determination is hard for typical scenes:
  - Objects are not convex
  - Geometry may be thin or filled with holes
  - Geometry may be textured with transparent maps

Rendered geometry

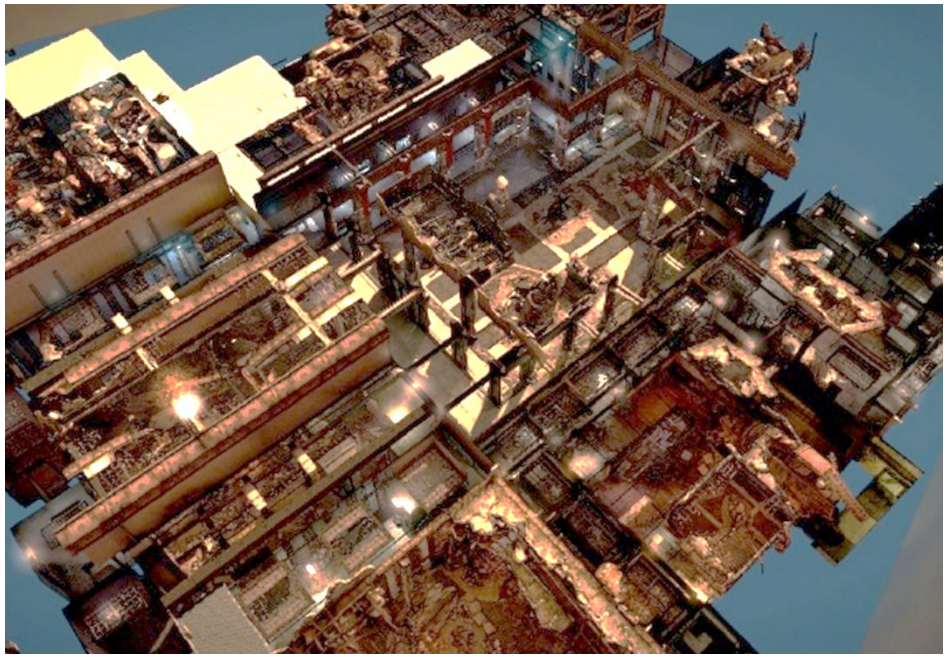


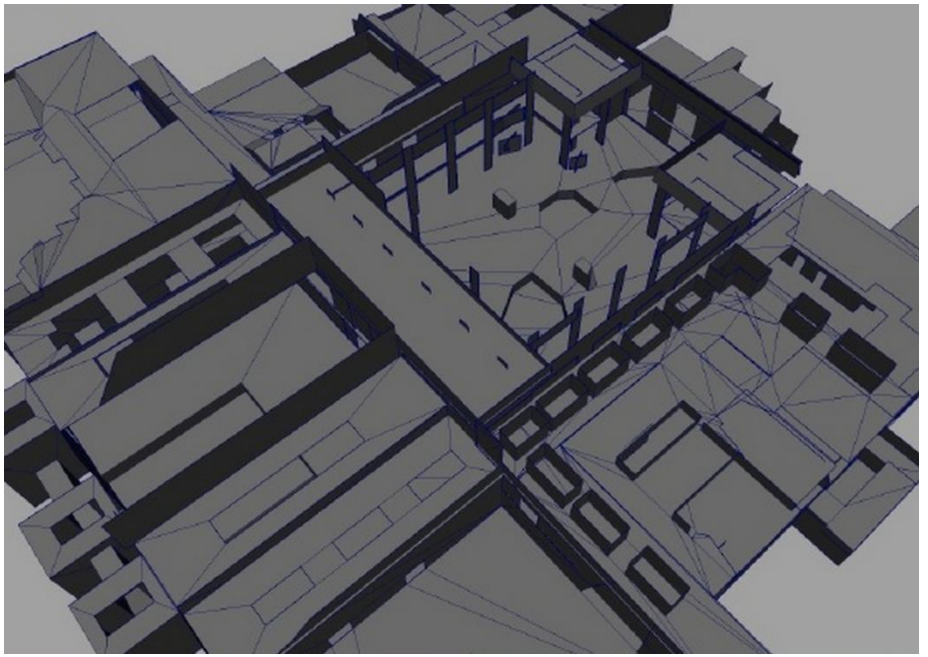
Occluder geometry



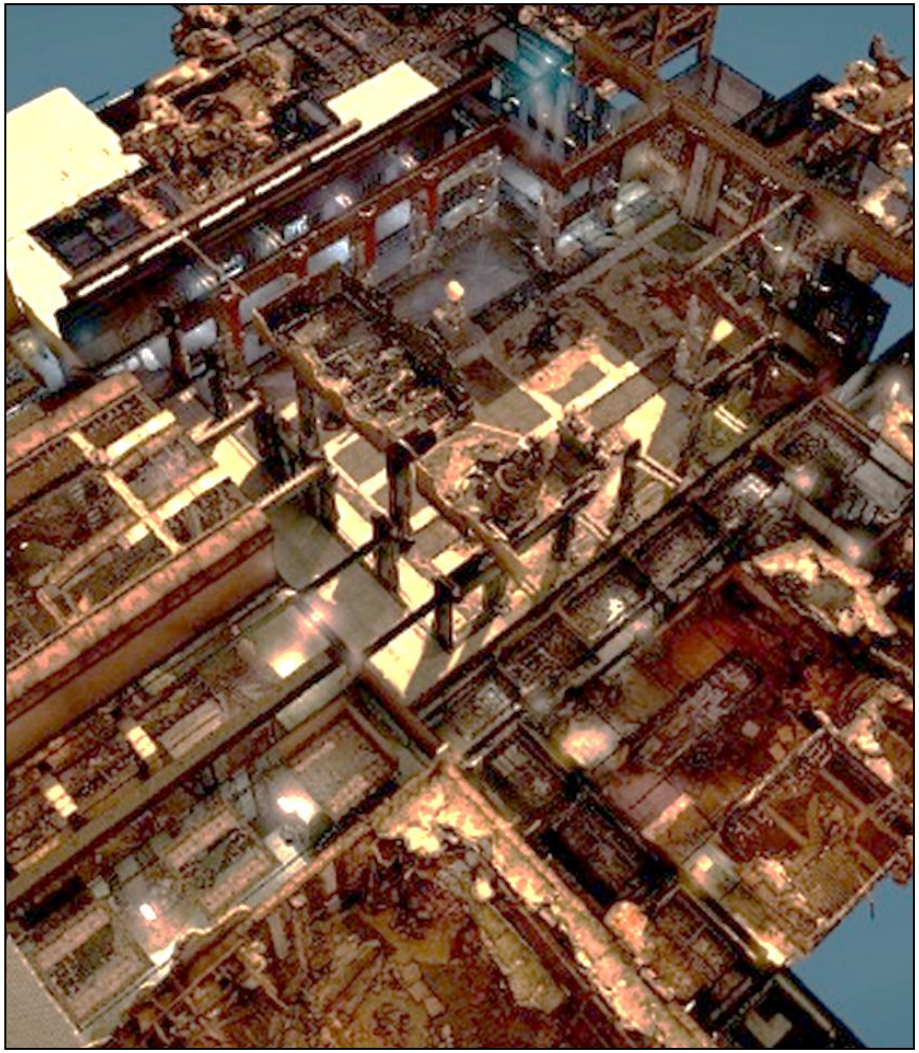
Occluder Determination (2)

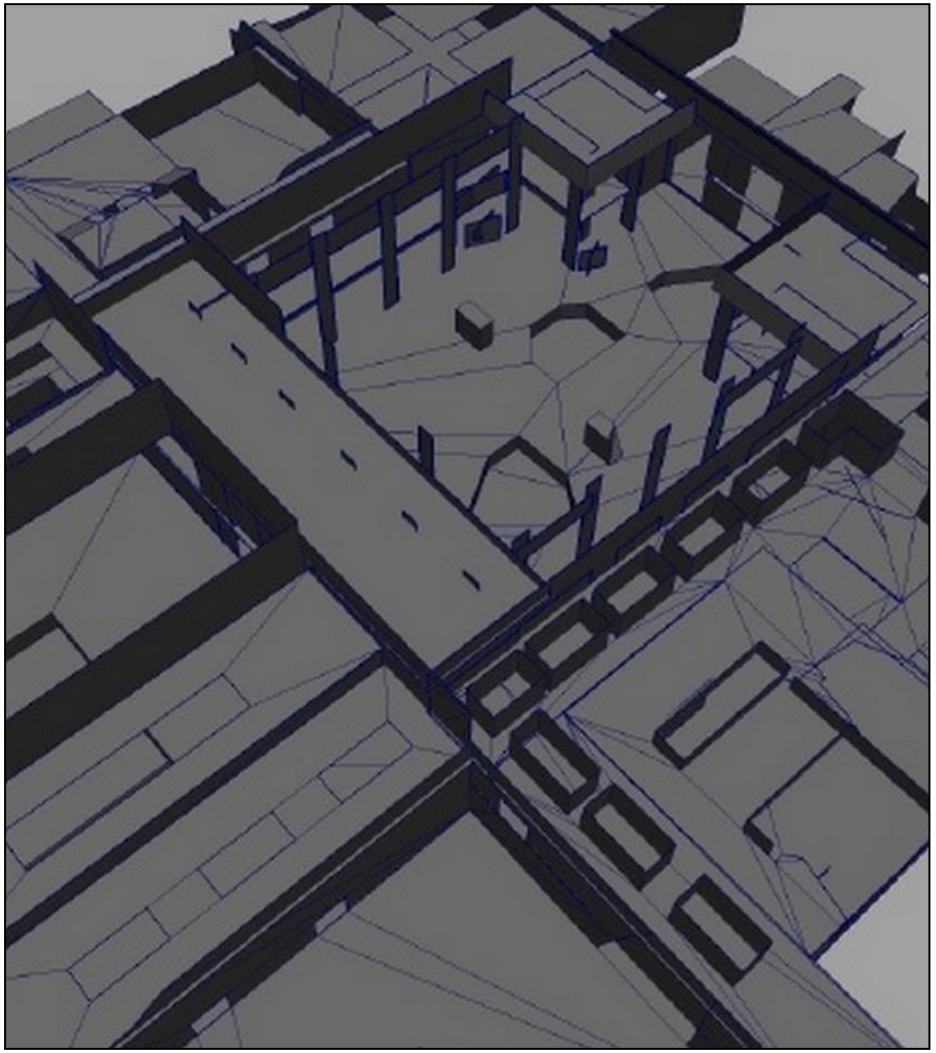
- Most occluders are either:
  - Manually built by artists or
  - Automatically pre-selected based on certain features and meta-data and approved by artists





Occluder Determination (3)





Practical Occlusion Culling in GPUs

- Typically, the occluders' geometry is large enough to make the generation of individual culling frusta prohibitive
- Another solution is to create a rough depth buffer based on the occlusion geometry and
- Test the bounding boxes of the objects against this buffer:
  - Fully hidden bounding boxes signify hidden objects

AUEB COMPUTER GRAPHICS GROUP

Contributors

- Georgios Papaioannou