Computergrafik

Matthias Zwicker
Universität Bern
Herbst 2016
Scene graphs & hierarchies

• Introduction
• Scene graph data structures
• Rendering scene graphs
• Level-of-detail
• Culling
So far: rendering pipeline

Scene data

- Vertex processing, modeling and viewing transformation
- Projection
- Rasterization, fragment processing, visibility

Image
Interactive applications
- Games, virtual reality, visualization

Rendering engine, scene graph API
- Implement functionality commonly required in applications
- Back-ends for different low-level APIs

Low-level graphics API
- Interface to graphics hardware
System architecture

Interactive applications
• Thousands

Rendering engine, scene graph API
• No broadly accepted standards
• Java3D, Ogre3D, OpenSceneGraph, jrtr, ...

Low-level graphics API
• Highly standardized
• OpenGL (jogl), Direct3D
Scene graph APIs

Common functionality

• Resource management
  - Content I/O (geometry, textures, materials, animation sequences)
  - Memory management

• High level scene representation
  - Scene graph

• Rendering
  - Efficiency
  - Advanced shading (materials, shadows, etc.)

Game engines

• Networking, physics, AI, etc.
  http://en.wikipedia.org/wiki/Game_engine
Scene graph APIs

- APIs focus on different clients/applications

- **Java3D** ([https://java3d.dev.java.net/](https://java3d.dev.java.net/))
  - Simple, easy to use, web-based applications

- **OpenSceneGraph** ([www.openscenegraph.org](http://www.openscenegraph.org))
  - Scientific visualization, virtual reality, GIS (geographic information systems)

- **Ogre3D** ([http://www.ogre3d.org/](http://www.ogre3d.org/))
  - Games, high-performance rendering

- **jrtr**
  - Under development...
Common functionality: Ogre3D
Scene graphs & hierarchies

• Introduction
• Scene graph data structures
• Rendering scene graphs
• Level-of-detail
• Culling
Scene graphs

• Data structure for intuitive construction of 3D scenes

• So far, jtr scene manager just stores a linear list of objects

• Ideas for improvement?
  - Disadvantages of list structure?
  - What functionality could an improved data structure support?
Sample scene

KK 5045
1500x450x760mm

KK 5060
1500x600x760mm
Top view
Top view with coordinates
Hierarchical organization
Data structure

- Requirements
  - Collections of individual models/objects
  - Organized in groups
  - Related via *hierarchical transformations* (transformations apply to groups of objects)
- Use a tree or graph structure
- Each node has associated local coordinates
  - Its own coordinate system
- Different types of graph nodes
  - Geometry (shapes, objects)
  - Transformations
  - Lights
  - ...
Class hierarchy

- Use a class hierarchy for graph nodes
  - All derived from same base class
  - Classes provide different functionality based on their intended use

- Many designs possible
  - Concepts are the same, details may differ

- Design driven by intended application
  - Games: optimize for speed
  - Large-scale visualization: optimize for memory requirements
  - Modeling system: optimize for editing flexibility
Class hierarchy

- Inspired by Java3D

```
Node
  /   \
Group Leaf
  /     \
TransformGroup LightNode ShapeNode
```

Abstract classes
Class hierarchy

Node

• Access to local-to-world coordinate transform

Group

• List of children
• Get, add, remove child

Leaf

• Node with no children
Class hierarchy

TransformGroup

- Stores additional transformation $M$
- $M$ applies to complete subtree below node
- Keyboard-to-world transform $M_0 M_1 M_2$
Class hierarchy

Subclasses of Leaf

Light

• Stores (or references) a light source (position, direction, strength, etc.)

ShapeNode

• References a geometric object and associated properties, e.g., material
Scene graph for sample scene

TransformGroup

ShapeNode
Building sample scene

```java
WORLD = new Group();
table1Trafo = new TransformGroup(...); WORLD.addChild(table1Trafo);
table1 = makeTable(); table1Trafo.addChild(table1);
top1Trafo = new TransformGroup(...); table1Trafo.addChild(top1Trafo);
lampTrafo = new TransformGroup(...); top1Trafo.addChild(lampTrafo);
lamp = makeLamp(); lampTrafo.addChild(lamp);

book1Trafo = new TransformGroup(...); top1Trafo.addChild(book1Trafo);
book1 = makeBook(); book1Trafo.addChild(book1);
```

- More convenient to construct scenes than using linear list of objects
- Easier to manipulate
  - Transformations automatically apply to whole subtree
Modifying the scene

• Change tree structure
  - Add, delete, rearrange nodes

• Change node parameters
  - Transformation matrices, create animations
  - Shape of geometry data
  - Materials

• Define specific subclasses
  - Animation, triggered by timer events...
Modifying the scene

- Change a transform in the tree
  
  ```
  table1Trafo.setRotationZ(23);
  ```

- Table rotates, everything on the table moves with it

- Allows easy animation
  - Build scene once at start of program
  - Update parameters to draw each frame

- Allows interactive model manipulation tools
  - Add objects relative to parent objects
  - E.g., book on table
Articulated character

- Separate rigid parts
- Joint angles define transformation matrices
- Hierarchy
  - Rooted at pelvis
  - Neck, head subtree
  - Arms subtree
  - Legs subtree
Parameteric models

• Scene graph can implement parametric model
  - Encapsulate functionality of a model in separate class

• Parameters for
  - Position in world
  - Relationship between parts (e.g., joint angles)
  - Shape of individual parts (e.g., length of limbs)

• Degrees of freedom (DOFs)
  - Total number of float parameters in the model
More node types

• Shape nodes
  - Cube, sphere, curved surface, etc...

• Nodes that control graph structure
  - Switch/Select: parameters choose whether or which children to enable, etc...

• Nodes that define other properties
  - Camera
  - Advantage: easy to specify location/orientation relative to some other scene part
Multiple instantiation

- A scene may have many copies of a model
- A model might use several copies of a part

- **Multiple instantiation**
  - One copy of node or subtree in memory
  - Reference (pointer) inserted as child of many parents
  - Object appears in scene multiple times, with different coordinates

- Not the same as instantiation in C++/Java terminology

- Scene is a **directed acyclic graph (DAG)**, not a tree
Multiple instantiation

Table-trafo

Tabletop

Leg-trafo 1

Leg-trafo 2

Leg-trafo 3

Leg-trafo 4

Leg

Transform group

Shape node
Multiple instantiation

Advantages

• Saves memory
• May save rendering time, depending on caching/optimization

Possible disadvantage

• Change parameter once, affects all instances
  - Can be good or bad, depending on what you want
• Solution: let children inherit properties from parent
  - Different instances have different properties
Fancier operations

Given articulated character, i.e., skeleton, compute skin

- Shape nodes that compute surface across multiple joint nodes
- Nodes that change shape of geometry
- Extremely popular in games
Scene graphs & hierarchies

- Introduction
- Scene graph data structures
- Rendering scene graphs
- Level-of-detail
- Culling
Basic rendering

- Traverse the tree recursively

```
TransformGroup::draw(Matrix4 C) {
    C_new = C*M;   // matrix M is a class member
    for all children
        draw(C_new);
}
```

```
ShapeNode::draw(Matrix4 C) {
    setModelView(C);
    setMaterial(myMaterial);
    render(myObject);
}
```

Initiate rendering with root node, here called \texttt{world}

```
world->draw(IDENTITY);
```
Rendering

• How about rendering huge scenes?
• For example, Google Earth

http://earth.google.com
Problems

- Too much data to store in main memory
- Too slow to render all geometry in each frame
Performance optimization

- Culling
  - Quickly discard invisible parts of the scene
- Level-of-detail (LOD) techniques
  - Use lower quality for distant (small) objects
  - Lower quality of geometry
  - Lower quality of shading
Performance optimization

• Scene graph compilation
  - Efficient use of low-level API
  - Minimize state changes (switching shaders, etc.) in rendering pipeline
  - Render objects with similar properties (geometry, shaders, materials) in batches (groups)

• Data management
  - Manage memory hierarchy (network, disk, CPU RAM, GPU RAM)
  - Load data into desired level of memory hierarchy on-demand
Today

Scene graphs & hierarchies

• Introduction
• Scene graph data structures
• Rendering scene graphs
• Level-of-detail
• Culling
Level-of-detail techniques

• “Adapt rendering technique to level of detail visible in the objects that are rendered”
  - Level of detail depends on distance of objects from camera

• Simplest approach: don’t draw small objects

• Use threshold
  - E.g., size in pixels

• Problem: popping artifacts
Impostors

- Replace objects by impostors
  - Textured planes representing the objects
  - Faster to render than original geometry

- Impostor generation
  - Exploits frame-to-frame coherence
  - Dynamic: generate periodically, reuse for a few frames
Geometric LOD

- Use fewer triangles for objects that are further away from the viewer
Shading LOD

• Use simpler shader for objects that are further away

With normal maps/bump mapping

Without normal maps/bump mapping
Today

Scene graphs & hierarchies

• Introduction
• Scene graph data structures
• Rendering scene graphs
• Level-of-detail
• Culling
Culling

- “Don’t attempt to draw objects that are not visible”
- Essential for interactive performance with large scenes
  - View frustum culling
    - Discard objects outside view frustum
  - Occlusion culling
    - Discard objects that are within view frustum, but hidden behind other objects
View frustum culling

- Frustum defined by 6 planes
- Each plane divides space into “outside”, “inside”
- Check each object against each plane
  - Outside, inside, intersecting
- If “outside” at least one plane
  - Outside the frustum
- If “inside” all planes
  - Inside the frustum
- Else partly inside and partly out
- Challenge: compute intersections efficiently
Bounding volumes

• Simple shape that completely encloses an object
  http://en.wikipedia.org/wiki/Bounding_volume

• Generally a box or sphere
  - Spheres easiest to work with, but hard to get tight fits
Culling with bounding volumes

- Intersect bounding volume with view frustum, instead of full geometry
  - Intersection computation much simpler
- Culling is **conservative**
  - Sometimes, bounding volume intersects view frustum, but actual geometry does not
  - No artifacts, but some performance loss
Culling with bounding spheres

Precomputation

• Simple computation of bounding spheres
  - Compute object „center“, e.g., average of all vertices
  - Sphere radius is largest distance from center to any vertex

• Tightest possible bounding sphere is harder to find, http://en.wikipedia.org/wiki/Bounding_sphere

Rendering

• Intersection of sphere with view frustum
Distance to plane

- A plane is described by a point $\mathbf{p}$ on the plane and a unit normal $\mathbf{n}$
- Find the (perpendicular) distance from point $\mathbf{x}$ to the plane
Distance to plane

- The distance is the length of the projection of $\overrightarrow{x-p}$ onto $\hat{n}$

$$dist = (\overrightarrow{x-p}) \cdot \hat{n}$$
Distance to plane

- The distance has a sign
  - positive on the side of the plane the normal points to
  - negative on the opposite side
  - 0 exactly on the plane
- Divides all of space into two infinite half-spaces

\[ \text{dist}(x) = (x - p) \cdot \hat{n} \]
Distance to plane

- Simplification

\[
\text{dist}(x) = (x - p) \cdot n = x \cdot n - p \cdot n
\]

\[
\text{dist}(x) = x \cdot n - d, \quad d = pn
\]

- \(d\) is independent of \(p\)
- \(d\) is distance from the origin to the plane
- We can represent a plane with just \(d\) and \(\vec{n}\)
Frustum with signed planes

- Normal of each plane points outside
  - “outside” means positive distance
  - “inside” means negative distance
Test sphere and plane

• For sphere with radius \( r \) and origin \( \mathbf{x} \), test the distance to the origin, and see if it’s beyond the radius

• Three cases
  
  – \( \text{dist}(\mathbf{x}) > r \)
    • completely above
  
  – \( \text{dist}(\mathbf{x}) < -r \)
    • completely below
  
  – \( -r < \text{dist}(\mathbf{x}) < r \)
    • intersects
Summary

• Precompute the normal \( \mathbf{n} \) and value \( d \) for each of the six planes

• Given a sphere with center \( \mathbf{x} \) and radius \( r \)

• For each plane:
  - if \( \text{dist}(\mathbf{x}) > r \): sphere is outside! (no need to continue loop)
  - add 1 to count if \( \text{dist}(\mathbf{x}) < -r \)

• If we made it through the loop, check the count:
  - if the count is 6, the sphere is completely inside
  - otherwise the sphere intersects the frustum
  - *(can use a flag instead of a count)*
Culling groups of objects

• If whole group of objects outside view frustum, want to be able to cull the whole group quickly

• But if the group is partly in and partly out, need to be able to cull individual objects

• Should we use scene graph hierarchy for culling?
Bounding volume hierarchies (BVH)

- Construct hierarchy of objects in a tree
- Bounding volume of each parent node encloses the bounding volumes of all its children
- Top-down or bottom-up construction

[Diagram of BVH construction]

Recursively split groups
Recursively merge groups
Hierarchical view frustum culling

- Start by testing the **outermost** bounding volume
  - If it’s entirely out, don’t draw the group at all
  - If it’s entirely in, draw the whole group
Hierarchical view frustum culling

- If bounding volume is partly inside and partly outside
  - Test each child’s bounding volume individually
  - If the child is in, draw it; if it’s out cull it; if it’s partly in and partly out, recurse.
  - If recursion reaches a leaf node, draw it normally
Occlusion culling

- „Don‘t attempt to draw objects that are occluded behind others“

- Related to the hidden surface determination problem


Occlusion culling

- Cell-based occlusion culling using potentially visible sets
  http://en.wikipedia.org/wiki/Potentially_visible_set
  http://www.gamedev.net/reference/articles/article1212.asp

1. Preprocessing
   - Divide scene into cells (3D cubes)
   - Each cell stores list of objects that intersect it
   - Determine potentially visible set (PVS) for each cell
   - PVS: For each cell, set of other cells potentially visible from any viewpoint within first cell

2. Rendering
   - When camera is in one cell, do not need to render any cells (i.e., any objects associated with that cell) not in PVS
PVS computation

- **Conservative**
  - Never exclude a potentially visible cell from the PVS, but some cells in PVS may actually not be visible from certain viewpoints
  - Problem: performance degradation

- **Aggressive**
  - No invisible cells are in PVS, but some visible cells may be omitted
  - Problem: visibility errors

- **Exact**
  - No visibility errors and no redundancy
  - Challenge: tractable algorithms to compute

http://en.wikipedia.org/wiki/Potentially_visible_set
Occlusion culling

• Example video using “Unity” game engine
  http://unity3d.com/
  http://www.youtube.com/watch?v=S5l3unhW4e0

• Specialized algorithms for different types of geometry
  - Indoor scenes (scenes with portals)
    http://en.wikipedia.org/wiki/Portal_rendering
  - Terrain (height field, 2.5D)
  - Urban scenes
Dynamic scenes

- Challenge: objects move or change shape
- Hierarchical culling with bounding volume hierarchies (BVH)
  - Need to update BVH
- Occlusion culling with potentially visible sets (PVS)
  - Need to update PVS
- Require efficient algorithms!
Next time

- Modeling: 3D curves