Termin 9:
Scene Graphs

Virtuelle Realität
Wintersemester 2006/07

Prof. Bernhard Jung

Overview

- Motivation
- Scene Graph Concepts: Node & Traversal
- Issues in Scene Graph Design
- Examples

Further information:
http://www.realityprime.com/scenegraph.php
OpenSG Documentation.
http://www.opensg.org/documentation.EN.html
Open Scene Graph Documentation.
http://www.openscenegraph.org/

Prof. B. Jung
Virtuelle Realität, WS 2006/07
Image Generation: Levels of Abstraction

- **Low-Level Graphic APIs**
  - DrawTriangle, SetLineColor, SetLightColor, ReadFrameBuffer, SetShadingModel
  - e.g. OpenGL, DirectX
  - very fine-grained and flexible
  - Every object has to be specified vertex by vertex including all its data

- **Scene Graph**
  - LoadFile, CreateNode, MoveObject, SetPickable
  - e.g. OpenInventor, Performer, Open Scene Graph, OpenSG, Java3D
  - The idea of a scene graph is to provide a higher level of abstraction. A scene graph stores the whole scene in the form of a graph of connected objects
  - declarative, not imperative
    - Thinking objects … not rendering processes

Overview: Image Generation in VR Systems

- Scene Graph
- Render
- Graphics Subsystem
- Display devices
- Scene Management
  - Java
  - Python
  - C/C++
- Low Level Graphics APIs
  - C/C++
- Rendering Hardware
- Monitor, Goggles, etc
What is a Scene Graph?

- Scene graph
  - a representation of all the visual data for a 3D scene (and often, other types of non-visual data, such as sound)
  - The specification for a scene graph is a mathematical graph
    - DAG (directed acyclic graph) or tree

- A scene graph includes data that describes:
  - 3D shape objects: their geometry and their appearance,
  - geometric structure relationships for 3D shape objects: geometric transformations, ordering, and grouping,
  - global objects that affect how all other objects are viewed: viewpoints, lights, backgrounds, environment effects,
  - (sometimes) behaviors: procedures for modifying information stored in the scene graph or the graph structure itself.

Example – VRML Scene Graph

```vrml
#VRML V2.0 utf8
DEF airplane Transform {
  position x y z
  orientation x y z angle
  children [
    Shape {
      material Material {
        diffuseColor r g b }
    }
  ]
}
```

---

Prof. B. Jung          Virtuelle Realität, WS 2006/07
Why Scene Graphs?

- The purpose of a scene graph is:
  - to provide an **application developer** (either a content author or a programmer) with a conceptual means to **define and organize** the static contents of a 3D scene and the dynamic events and behaviors that can modify the 3D scene, and
  - to provide the **graphics system** with a data structure that contains all the information needed to **render** one or more images of the 3D scene.

- I.e. a scene graph is an **abstraction** that provides an **interface** between an **application developer (a human)** and a **rendering system (a computer)**.

[Lewis E. Hitchner]
Why Scene Graphs?

- Generalized frustum culling
  - Culling = removing parts of the scene that do not contribute to final image
  - OpenGL supports frustum culling at polygon level
  - Culling at object level improves rendering performance (store bounding volumes in scene graph hierarchy)

- Performer, OpenSceneGraph
  - app (update scene) – cull – draw

Why Scene Graphs?
Immediate Mode vs Retained Mode Graphics (OpenGL)

- Immediate Mode Graphics
  - Primitives (vertices, attributes, ...) are sent to the pipeline and displayed right away
  - No memory of graphical entities
  - New images are created by regenerating & resending the primitives

- Display Listed / Retained Mode Graphics
  - Primitives placed in display lists
  - Display lists are kept on graphics server (in "compiled" form)
  - Images can be recreated by “executing” the display list
  - Can be redisplayed with different state
  - Can be shared among OpenGL graphics contexts

- Scene Graphs: All about retained mode
Immediate Mode versus Display Lists

- Immediate Mode
  - Polynomial Evaluator
  - Per Vertex Operations & Primitive Assembly
  - Rasterization
  - Texture Memory
  - Per Fragment Operations
  - Frame Buffer

- Display Lists
  - CPU
  - Display Listed
  - Pixel Operations

Node Types

- 2 Hierarchies
  - scene graph hierarchy
  - class hierarchy (OOP)
- Some classes are not in the scene graph but still part of the scene

Nodes

- Inner Nodes
  - LOD
  - Group
  - Transform
- Leaves
  - Geometry
  - Sound
  - Particle System

Non-Nodes

- Material
- Texture
- (Light Sources)
Scene Graph Semantics

- Semantic of nodes
  - root = "universe"
  - leaves = "content" (can be rendered; geometry, sound, …)
  - inner nodes:
    - grouping
    - state changes (transformation, materials, light, …)
    - non-geometric functionality (sensors, interpolation nodes, …)

- Semantic of edges
  - inheritance of "state" (transformation, materials, light, …)

Rendering via Scene Graph Traversal

1. At the start (root node) all graphics state variables are set to the system default values.
2. If a node is a group type node that has more than one child, the algorithm selects one child and continues the path’s traversal through that child node.
3. If a node contains a geometric transformation, the algorithm updates the current modeling transformation, i.e., the node’s transform is composed or concatenated (via matrix multiplication) with the transforms from all previous transformation nodes in the path.
4. If a node contains appearance data (surface material, texture, drawing style, etc.), the algorithm updates the appropriate graphics state variable by setting (replacing) the current state values to the values stored in the node.
5. If a node contains geometry data, the algorithm renders that geometry.
6. If a node is a leaf, the traversal backtracks or terminates.

* Idealized description - Actual implementations will include speed-ups, e.g. culling.
Rendering via Scene Graph Traversal

Scene graph traversals

E.g. Performer – not only rendering (draw) traversals!

- Stages (sometimes processes) traverse from the root node
- Traversal Order: Scene graphs are traversed in a depth-first, left-to-right order.
History of Scene Graphs

Issues: Inheritance

Parent-Child Inheritance (most scene graphs)

Parent-Child Inheritance + Left-to-Right Inheritance (e.g. OpenInventor) bad idea!

Issues: DAG or Tree?
(sharing geometry and other state info)

Example: Sharing State in VRML
DEF / USE

Shape { appearance
  DEF common Appearance {
    material Material {diffuseColor 1 0 0}
  }
  geometry Sphere { }
}

Transform {
  translation -2 0 0
  children [
    Shape {
      appearance USE common
      geometry Cone { }
    }
  ]
}
Issues: DAG or Tree

Directed Acyclic Graph (DAG)
- a node may have more than one parent
- i.e. multiple paths to root node possible
- e.g. OpenInventor, VRML
- advantage: can share e.g. geometry between nodes
- problems:
  - multiple inheritance issues
  - VRML uniqueness of node names/pointers?

Tree
- at most one parent per node
- i.e. unique path to root node
- most scene graphs
- how to share geometry? e.g. copy?

Geometry Sharing in OpenSG

- node is split in 2 parts
  - Node
    - only general information, e.g. parent-child, bounding volume
    - Node hierarchy is a tree!
  - NodeCore
    - attached to nodes
    - detailed information e.g. transformation matrices, geometry, materials, …
    - nodecores can be shared!
### Issue: Scene Graph Grouping

- **Possible Criteria**
  - spatial proximity
    - supports culling
  - same / similar material
    - minimize state changes
    - improve rendering performance
  - logical structure
    - e.g. same object types
  - ...

- Generally, scene graph grouping is left to the application
  - i.e. human modeler, application programmer decides

- Some scene graphs graphics libraries reorder the scene graph automatically to improve rendering performance
  - works best for static scenes

---

### Culling

Organizing a database for efficient culling: Organizing this database spatially, rather than by object type or other attributes, promotes efficient culling

Performer: can reorder scene for culling (`pfChannel()`)
State Sorting

- State = totality of all attributes
  - e.g. color, lighting parameters, texture, transformations, …

- State changes are one of the performance killers
  - costs:

```
<table>
<thead>
<tr>
<th>transformation</th>
<th>lighting</th>
<th>texture</th>
<th>shader program</th>
</tr>
</thead>
<tbody>
<tr>
<td>changes</td>
<td>changes</td>
<td>changes</td>
<td>changes</td>
</tr>
</tbody>
</table>
```

- Goal: Render scene graph with minimal number of state changes
- "Solution": Pre-Sorting
  - does not work well with dynamic scenes, occlusion culling
- In some scene graphs APIs, user can mark static and dynamic aspects
  - e.g. Performer: DCS vs SCS nodes (dynamic / static coordinate system)
  - e.g. Java3D: immediate, retained, and compiled-retained rendering modes

Common Node Types

- Hierarchy / inner nodes
  - TransformGroup, Group, Billboard, LOD, Switch, …

- Drawables / leaf nodes
  - Shape / Geometry, Sound, Particle System, …

- Less common:
  - Event & action nodes
    - for user interaction and animation
    - e.g. OpenInventor, VRML
      - dataflow programming in the scene graph
      - problem: interplay with culling/rendering?
  - Special nodes
    - Body & facial animation, e.g. X3D, MPEG-4
VRML – Events & Routing (dataflow prog.)

GROUP
  DEF PI PositionInterpolator
    key [ 0.0, .1, .4, .7, .9, 1.0 ]
    keyValue [ -3 0 0, 0 0 0, 0 20 -50, 0 0 -100, 0 0 0, -3 0 0 ]
  DEF T Transform
    translation -3 0 0
    children Shape { geometry Cone ()
      appearance Appearance {
        material Material { diffuseColor 1 0 0 } } }
  DEF TOS TouchSensor
  DEF TS TimeSensor { cycleInterval 3.0 } # 3 sec loop

ROUTE
  PI.value_changed TO T.translation
  TOS.touchTime TO TS.startTime
  TS.fraction_changed TO PI.set_fraction

Scene graph nodes in Performer

http://www.sgi.com/products/software/performer/

pNode Abstract Basic node type.
pGroup Branch Groups zero or more children.
pScene Branch Parent of the visual database.
pSCS Branch Static coordinate system.
pDCS Branch Dynamic coordinate system.
pFCS Branch Flux coordinate system.
pDoubleSCS Branch Double-precision static coordinate system.
pDoubleDCS Branch Double-precision dynamic coordinate system.
pDoubleFCS Branch Double-precision flux coordinate system.
pSequence Branch Sequences through its children.
plOD Branch Level-of-detail node.
plLayer Branch Renders coplanar geometry.
plLightSource Branch Contains specifications for a light source.
plGeode Leaf Contains geometric specifications.
plBillboard Leaf Rotates geometry to face the eyepoint.
plPartition Leaf Partitions geometry for efficient intersections.
plText Leaf Renders 2D and 3D text.
plASD Leaf Controls transition between LOD levels.
Scene Graph Node Types in OpenSG

http://www.opensg.org/

Groups
- osg::Transform - base for class nodes which transform their subgraph
- osg::Geode - leaf node from which drawable leaves are hung
  - osg::Billboard - orients associated drawables to face the viewer
- osg::LOD - level of detail node
- osg::Impostor - adds support for hierarchical image caching
- osg::Switch - controls which of the children are on or off
- osg::Sequence - automatically steps through children
- osg::LightSource - positions an osg::Light in the scene
- osg::ClipNode - positions osg::ClipPlane planes in the scene
- osg::Projection - overrides the projection matrix (useful for HUD’s)
- osg::OccluderNode - positions convex planer occluders and convex planer portals in the scene

Drawables (leaf nodes)
- osg::Geometry - adds real geometry and ability to draw it
- osg::ShapeDrawable - adds the ability to render the shape primitives
- osg::ImpostorSprite - used internally by the ImpostorNode as a billboard
  – …

Scene Graph Node Types in OpenSceneGraph

http://www.openscenegraph.org/

Groups
- osg::Transform - base for class nodes which transform their subgraph
- osg::Geode - leaf node from which drawable leaves are hung
  - osg::Billboard - orients associated drawables to face the viewer
- osg::LOD - level of detail node
- osg::Impostor - adds support for hierarchical image caching
- osg::Switch - controls which of the children are on or off
- osg::Sequence - automatically steps through children
- osg::LightSource - positions an osg::Light in the scene
- osg::ClipNode - positions osg::ClipPlane planes in the scene
- osg::Projection - overrides the projection matrix (useful for HUD’s)
- osg::OccluderNode - positions convex planer occluders and convex planer portals in the scene

Drawables (leaf nodes)
- osg::Geometry - adds real geometry and ability to draw it
- osg::ShapeDrawable - adds the ability to render the shape primitives
- osg::ImpostorSprite - used internally by the ImpostorNode as a billboard
  – …
Billboard Nodes

- Billboards always orient themselves towards the viewer.

![Billboard Diagram](image)

LOD Nodes

- Levels of Detail are a simple way of increasing rendering performance. The basic idea is to have a number of differently detailed versions of an object and use low-res versions for objects that are far away.

![LOD Diagram](image)
Switch Nodes

- A Switch node allows to select one of its children for traversal instead of all of them (as for the other nodes).

Particles

- The main idea of particles is to give a way of easily rendering large numbers of simple geometric objects. Particles are mostly used together with partially transparent textures to simulate fuzzy objects, but other uses are possible, e.g. molecules as stick-and-sphere models, stars or arrows for flow-field visualizations.
Example – Constructing a Scene Graph Programatically in OpenSceneGraph

```cpp
go::ref_ptr<go::Group> group = new go::Group; // create a Group node
go::Geode* geode = new go::Geode; // create a Geode node
geode->addDrawable(new go::ShapeDrawable(new go::Cone(go::Vec3(0,0,0), 0.5f, 1.0f )));
group->addChild( geode); // add the geode node to the group node
geode = new go::Geode; // create another Geode node
geode->addDrawable(new go::ShapeDrawable(new go::Box( go::Vec3(0,0,0), 3.0f, 3.0f, 0.05f )));
group->addChild( geode );
goProducer::Viewer viewer(); // create the viewer window & set its root node
viewer.setSceneData( group.get() );
viewer.realize();
```

Example – OpenSceneGraph Main Loop

```cpp
goProducer::Viewer viewer(); // create the viewer window & set its root node
viewer.setSceneData( group.get() );
viewer.realize();

while( !viewer.done() )
{
    // wait for all cull and draw threads to complete.
    viewer.sync();

    // update the scene by traversing it with the update visitor which will
    // call all node update callbacks and animations.
    viewer.update();

    // fire off the cull and draw traversals of the scene.
    viewer.frame();
}
```
Scene Graphs in VR Systems

- Virtual Reality System
- Scene Graph
- OpenGL
- Operating System (UNIX, Windows)

Scene Graphs
- useful not only for rendering

- Input
- Network
- Physical Simulation
- Visual Rendering
- Collision Detection
- Haptic Rendering
Further Issues

- Distributed scene graphs
  - distributed rendering, e.g. cluster support (CAVE)
  - WAN
  - e.g. OpenSG, Avango
- Thread-Safety (multi-process environments)
- Shading languages support

- is les mor?
  - Instead of using one scene graph for many tasks (rendering, animation, physics, application logic, networking, …)
  - …
  - better to use several hierarchies? → "Multi-view scene graphs"