Introduction to Scene Graphs

Kévin Boulanger
November 2006

Scene Graph

- Hierarchical data structure
- Easy to build
- Easy to extend
- Allows work by many developers at the same time

Some examples

- Scene graph is a commonly used data structure (web, video games)

- Examples:
  - OpenSceneGraph (www.openscenegraph.org)
  - OpenSG (www.opensg.org)
  - nVSG (developer.nvidia.com/object/nvsg_home.html)
  - Java3D (java.sun.com/products/java-media/3D)
  - VRML, X3D (www.x3d.org)
  - SGI Open Inventor (oss.sgi.com/projects/inventor)
  - OpenMASK (www.irisa.fr/siames/OpenMASK)
  - etc.

- Many scene graph designs exist, each adapted to its respective use

Definition

- Objects, items in a 3D scene are rarely independent:
  - physical connection
  - interaction
  - influence (light)
  - requirement (data)

A scene graph is a hierarchical data structure, based on a graph:
- the leaf nodes represent objects to render,
- the internal nodes are groups, transformations, properties

- Two types:
  - tree
  - DAG (Directed Acyclic Graph)
Tree

A tree contains no loop and each node has a single parent node

- paint
- translation
- translation
- translation
- translation
- body
- tire material
- tire material
- tire material
- tire material
- wheel
- wheel
- wheel
- wheel

DAG (Directed Acyclic Graph)

A DAG contains no loop and each node can have several parent nodes

- paint
- tire material
- body
- translation
- translation
- translation
- translation
- translation
- wheel

Comparison

- Tree (one parent, several children per node):
  - very simple to build and manage,
  - simple to traverse,
  - however, duplication of data in memory is required

- DAG (several parents, several children per node):
  - more complicated to build,
  - slightly more complicated to traverse,
  - great advantage: no duplication of data in memory, sharing of resources

Rendering

- Rendering consists of traversing the graph:
  - a geometry node has to be rendered (primitives, meshes),
  - a transform node has to modify the transformation matrices,
  - a state node has to change a render state (color, material, shader, ...)

- Depth-first traversal:
  - simple for a tree,
  - for a DAG, do as if it is a tree, nodes will be traversed several times if necessary
Traversal of the graph (example 1)

Execution equivalent to the depth-first traversal

```cpp
void RenderCar()
{
    setMaterial(paint);
    renderBody();
    renderBody();
    setMaterial(tire);
    // Wheel 1
    glPushMatrix();
    glTranslatefv(transWheel1);
    renderWheel();
    glPopMatrix();

    // Wheel 2
    glPushMatrix();
    glTranslatefv(transWheel2);
    renderWheel();
    glPopMatrix();

    // Wheel 3
    glPushMatrix();
    glTranslatefv(transWheel3);
    renderWheel();
    glPopMatrix();

    // Wheel 4
    glPushMatrix();
    glTranslatefv(transWheel4);
    renderWheel();
    renderWheel();
    glPopMatrix();
}
```

Transform node

1. Save the current modelview matrix: `glPushMatrix()`
2. Apply the transformation: `glTranslatef()`, `glRotatef()`, `glMultMatrixf()`
3. Render every children
4. Restore the modelview matrix: `glPopMatrix()`

Second example

Hand-coded function

```cpp
render(some); glTranslatef(some); render(some);
render(some); glTranslatef(some); render(some);
render(some); glTranslatef(some); render(some);
render(some); glTranslatef(some); render(some);
render(some);
```

Direct traversal of the graph will render the same object but not optimally

Traversal of the graph (example 2)

Execution equivalent to the depth-first traversal

```cpp
render(some); glTranslatef(some); render(some);
render(some); glTranslatef(some); render(some);
render(some); glTranslatef(some); render(some);
render(some); glTranslatef(some); render(some);
render(some);
```
Comparison

Hand-coded function

```c
glPushMatrix();
glRotatef(-angle1);
render(arm1);
glPopMatrix();
```

Graph traversal

```c
glPushMatrix();
glRotatef(-angle1);
render(arm1);
glPushMatrix();
glTranslatef(transArm);
glRotatef(-angle2);
render(arm2);
glPopMatrix();
glPushMatrix();
glRotatef(angle1);
render(arm1);
glPopMatrix();
glPushMatrix();
glRotatef(angle2);
render(arm2);
glPopMatrix();
glPopMatrix();
glPopMatrix();
glPushMatrix();
glTranslatef(transArm);
glRotatef(angle2);
render(arm2);
glPushMatrix();
glRotatef(angle1);
render(arm1);
glPopMatrix();
glPushMatrix();
glTranslatef(transArm);
glPushMatrix();
glRotatef(angle2);
render(arm2);
```

Optimization

- Direct traversal of the graph for rendering is not optimal when there are many transform and render state nodes:
  - useless `glPushMatrix()` and `glPopMatrix()` calls,
  - useless changes of render states (two times the same color)

- Usage of a structure in parallel, dedicated to rendering:
  - the scene graph is traversed,
  - each encountered node is stored in a queue or a tree,
  - sorting is done to group render state changes,
  - if possible, matrix modifications are reduced,
  - finally, the queue or tree is traversed for rendering

  Sometimes quite difficult to do the sorting because of constraints, as for transparency (polygons have to be rendered from back to front)

Culling

- Culling consists in rendering parts of the 3D scene for optimization purposes
  - frustum culling allows to render only what is in the viewer field-of-view,
  - space partitioning methods minimize the size of the spaces to manage (BSP trees, octrees, etc.).

- Scene graphs allow efficient culling by removing full branches when traversing. If a node is considered out of view, it is not rendered, neither the children

  Bounding boxes or spheres are commonly associated with the nodes
  - the leaf nodes have associated bounding volumes
  - the internal node bounding volumes enclose every bounding volume of the children
  - the closer to the root node, the bigger are the bounding volumes

Collision detection

- Scene graphs are well adapted to collision detection

- Approach similar to culling:
  - intersection points of rays with the scene have to be computed,
  - if a ray does not collide the bounding volume of a node, there is no need to test the intersection with the children, a whole branch of the graph is eliminated
  - if a ray collides the bounding volume of a node, the intersection has to be found by testing the child nodes
**Base node: group**

- Connects a parent to several children
- Used in graphs where every nodes have a single child
- When traversing for rendering, simply traverse every children for rendering

**Base node: transform**

- Transform the child or every children (translation, rotation, scale, projection)
- Must save the current transformation matrix before rendering the children and restore it after

**Base node: render state**

- Change one of the current render states (color, material, lighting, transparency, shader, etc.)
- Implemented different ways depending on the graph design
- Must save the current render state before rendering the children and restore it after

**Base node: switch**

- Does a selection by rendering only the desired child
- Often used for levels-of-detail management (selection of the mesh quality depending on the distance from the camera for example)
Scene graph design

• There are at least many different scene graph design approaches as the number of existing libraries

• Common examples:
  • usage of nodes for everything, with only one child
    (group nodes have to be used)
  • usage of nodes for everything, with several children per node
  • usage of only transform nodes (several children) and geometry nodes (leaves). Additional data are associated with each node.