Lecture 08: Hierarchical Modeling with Scene Graphs

CSE 40166 Computer Graphics

Peter Bui

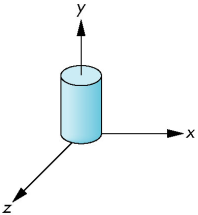
University of Notre Dame, IN, USA

November 2, 2010

Symbols and Instances

Objects as Symbols

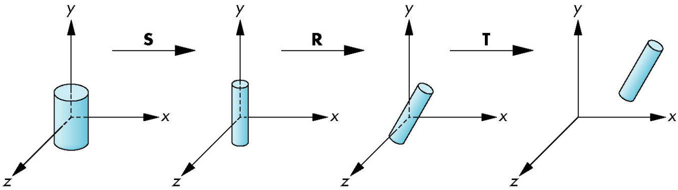
Model world as a collection of object symbols.



Instance Transformation

Place instances of each symbol in model using $M = TRS$.

```
glTranslatef(dx, dy, dz);
glRotatef(angle, rx, ry, rz);
glScalef(sx, sy, sz);
gluCylinder(quadric, base, top, height, slices, stacks);
```



Problem: No Relationship Among Objects

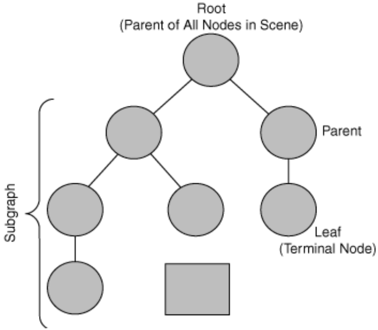
Symbols and instances modeling technique contains no information about relationships among objects.

- Cannot separate movement of Wall-E from individual parts.
- Hard to take advantage of the re-usable objects.

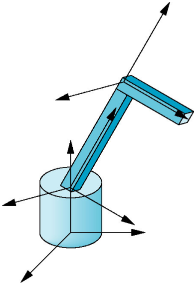
Solution: Use Graph to Model Objects

Represent the visual and abstract relationships among the parts of the models with a graph.

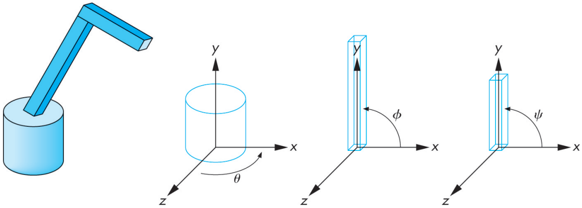
- Graph consists of a set of **nodes** and a set of **edges**.
- In **directed graph**, edges have a particular direction.



Example: Robot Arm

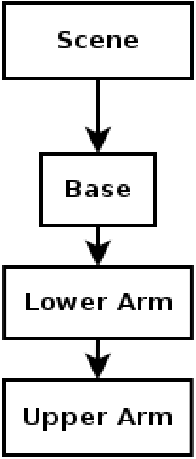


```
display()
{
    glRotatef(theta, 0.0, 1.0, 0.0);
    base();
    glTranslatef(0.0, h1, 0.0);
    glRotatef(phi, 0.0, 0.0, 1.0);
    lower_arm();
    glTranslatef(0.0, h2, 0.0);
    glRotatef(psi, 0.0, 0.0, 1.0);
    upper_arm();
}
```



(a) (b)

Example: Robot Arm (Graph)




```
ConstructRobotArm(Root)
{
    Create Base;
    Create LowerArm;
    Create UpperArm;

    Connect Base to Root;
    Connect LowerArm to Base;
    Connect UpperArm to LowerArm;
}

RenderNode(Node)
{
    Store Environment;

    Node.Render();

    For Each child in Node.children:
        RenderNode(child);

    Restore Environment;
}
```

Scene Graph: Node

Node in a scene graph requires the following components:

- **Render:** A pointer to a function that **draws** the object represented by the node.
- **Children:** Pointers to the children of the node.

It may also contain the following:

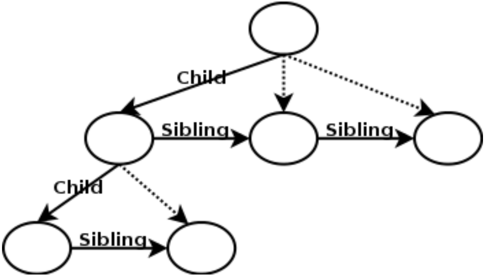
- **Transformation:** Homogeneous-coordinate matrix that positions, scales, and orients node and children relative to parent.
- **Material properties:** Values that define the color or materials of object.
- **Drawing Style:** Settings that determine the drawing style for the object.

Scene Graph: Node (C)

Simple implementation using a **left-child, right-sibling** structure.

```c
typedef void render_func_t(SSG_Node *n);

struct SSG_Node {
    render_func_t    *render;
    struct SSG_Node *child;
    struct SSG_Node *sibling;
    void *data;
};
```



Scene Graph: Node (C++)

Simple implementation using STL lists and C++ classes:

```
class SSG_Node {
public:
         SSG_Node();
virtual ~SSG_Node();
virtual  render();
    void add_child(SSG_Node *n);
protected:
    std::list<SSG_Node *> mChildren;
};
```

Scene Graph: Traversal

To render or process a graph, we have to *traverse* it. The most common method is recursively using a **depth-first** and **preorder** approach:

```
void
ssg_node_render(SSG_Node *n)
{
    if (n == NULL) return;

    glPushMatrix();
    if (n->render)
        n->render(n);

    ssg_node_render(n->child);
    glPopMatrix();

    ssg_node_render(n->sibling);
}
```

- Why the check if $n == NULL$?
- Why push and pop the matrix?

Scene Graph: Viewer/Engine

Once we have a scene setup in a graph, we need another class or object that will view or process the scene graph. This object is normally a wrapper for the traditional OpenGL, GLUT functions we have been using and will contain the global variables we have been using:

```c
/* Data Structure */
struct SSG_Viewer {
    const char *title;
    size_t      width;
    size_t      height;
    int         frame;
    double      eye_x;
    double      eye_y;
    double      eye_z;
    double      camera_distance;
    double      camera_longitude;
    double      camera_latitude;
    int         mouse_x;
    int         mouse_y;
};

/* Methods */
SSG_Viewer *ssg_viewer_create(const char *title, size_t width, size_t height);
void        ssg_viewer_initialize(SSG_Viewer *v, int *argc, char *argv[]);
void        ssg_viewer_show(SSG_Viewer *v, SSG_Node *n);
```

Scene Graph: Robot Arm

```
SSG_Node *root = NULL;
SSG_Node *base = NULL;
SSG_Node *lower_arm = NULL;
SSG_Node *upper_arm = NULL;
SSG_Viewer *viewer = NULL;

/* Create nodes */
root      = ssg_node_create(NULL, NULL);
base      = ssg_node_create(cylinder_render, NULL);
lower_arm = ssg_node_create(cube_render, NULL);
upper_arm = ssg_node_create(cube_render, NULL);

/* Connect nodes */
ssg_node_connect(root, base);
ssg_node_connect(base, lower_arm);
ssg_node_connect(lower_arm, upper_arm);

/* Setup viewer and render scene */
viewer = ssg_viewer_create("robot arm", 640, 480);
ssg_viewer_initialize(viewer, &argc, argv);
ssg_viewer_show(viewer, root);
```

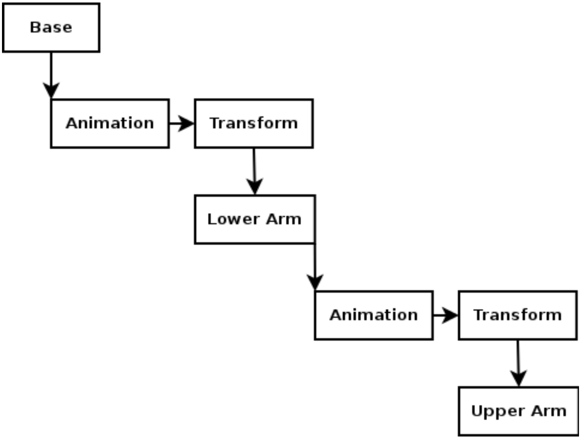
Scene Graph: Transformations and Animation

In a scene graph, nodes are not restricted to drawing objects. They could also be:

- **Lighting:** Adjust lighting for different sets of objects.
- **Camera:** Move camera around per object or group of objects.
- **Transformation:** Apply transformation to children in a hierarchical fashion.
- **Animation:** Control animation of different objects based on time of key-frames.
- **Switch:** Activate or deactivate child nodes based on some parameter.
- **Event:** Have keyboard, mouse, or timer events trigger changes in animation or objects.

Scene Graph: Robot Arm Animation

Suppose we want to add animation to our robot arm. We would need to create a graph like so:

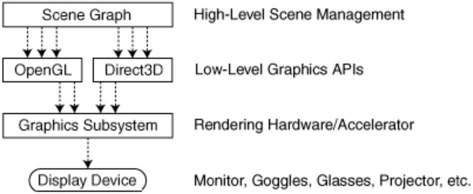


Demo example 23.

Scene Graph: Summary

A scene graph is basically a *n-tree* or *DAG* where we order our data into a hierarchy such that parent nodes affect the child nodes.

Normally, each node contains a transformation matrix and a renderable object. As tree is traversed during rendering, the matrices are concatenated and objects are rendered wth the resulting transformation.



Scene Graph: Applications

Used in may real-world applications such as:

- **Graphics Editing Tools:** AutoCAD, Adobe Illustrator, Acrobat 3D.
- **Modeling Systems:** VRML97, OpenSceneGraph.
- **Multimedia:** MPEG-4.
- **Games:** Quake.

Scene Graph: Why use them?

- **Transform Graph:** Model hierarchical objects such that child objects are defined relative to parents.
- **Abstraction:** Only concern yourself with what's in the scene and any associated behavior or interaction among objects.
- **Culling:**Allow for high-level culling or object removal.
- **Easy of manipulation:** Break object down into individual nodes and we can animate pieces separately.
- **State Sorting:** All objects rendered are sorted by similarities in state.

Focus on **what** to render, rather than **how** to render.

Scene Graph: Spatial Partitioning

Besides simplifying and possibly speeding up rendering, scene graphs can also be used to help in collision detection by adapting them into a set of **bounding volume hierarchies**.

1. Check if probe object is in gallery object's volume.
2. If in volume, then check repeat for childre of gallery object.
3. If not in volume, then skip the rest of gallery object's children.