Scene graph

Computer Graphics

CSE 167

Lecture 10

Scene graph

- Data structure for intuitive construction of 3D scenes
- So far, our GLFW-based projects store a linear list of objects
- This approach does not scale to large numbers of objects in complex, dynamic scenes

Data structure

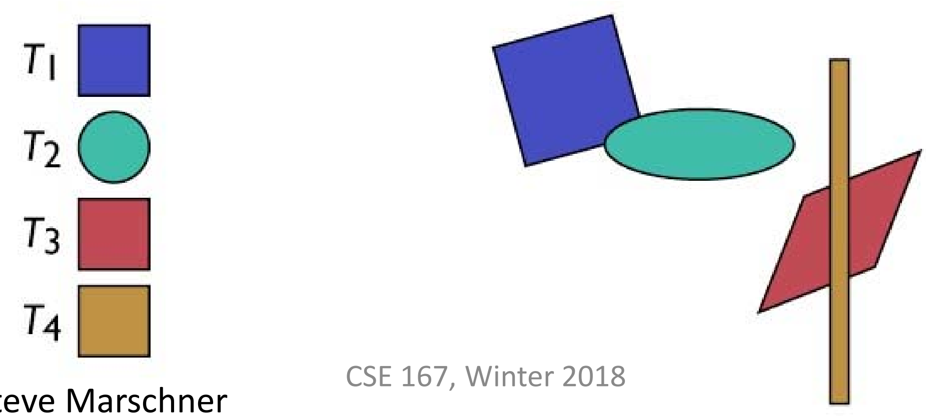
- Requirements
  - Collection of separable geometry models
  - Organized in groups
  - Related via hierarchical transformations
- Use a tree structure
- Nodes have associated local coordinates
- Different types of nodes
  - Geometry
  - Transformations
  - Lights
  - Many more

Class hierarchy

- Many designs possible
- Design driven by intended application
  - Games
    - Optimized for speed
  - Large-scale visualization
    - Optimized for memory requirements
  - Modeling system
    - Optimized for editing flexibility

Data structures with transforms

- Representing a drawing ("scene")
- List of objects
- Transform for each object
  - Can use minimal primitives: ellipse is transformed circle
  - Transform applies to points of object



Example

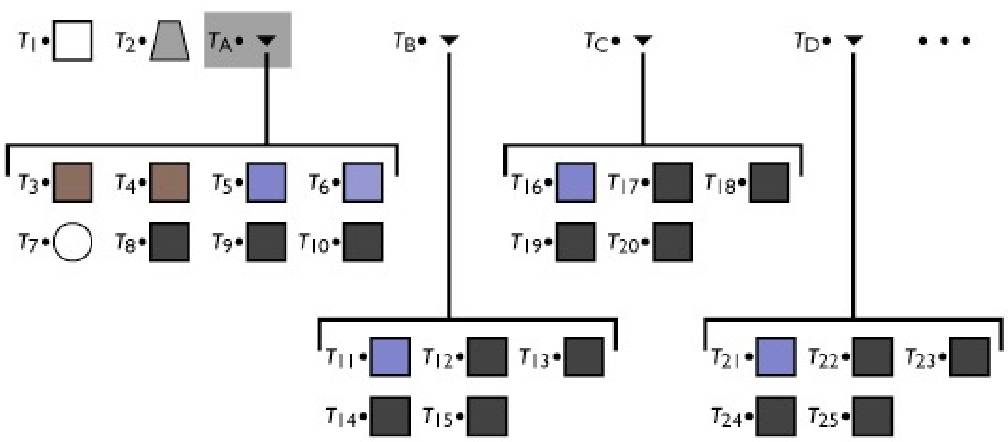
- Can represent drawing with flat list
  - But editing operations require updating many transforms

Groups of objects

- Treat a set of objects as one
- Introduce new object type: group
  – Contains list of references to member objects
- This makes the model into a tree
  – Interior nodes = groups
  – Leaf nodes = objects
  – Edges = membership of object in group

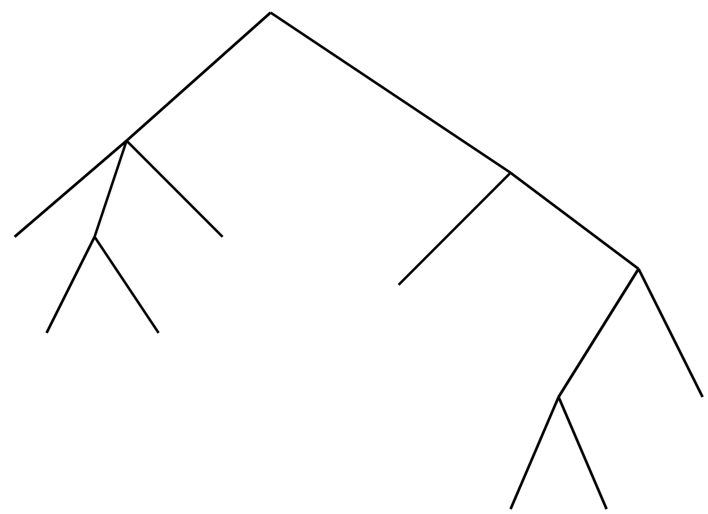
Example

- Add group as a new object type
  - Lets the data structure reflect the drawing structure
  - Enables high-level editing by changing just one node



The scene graph (tree)

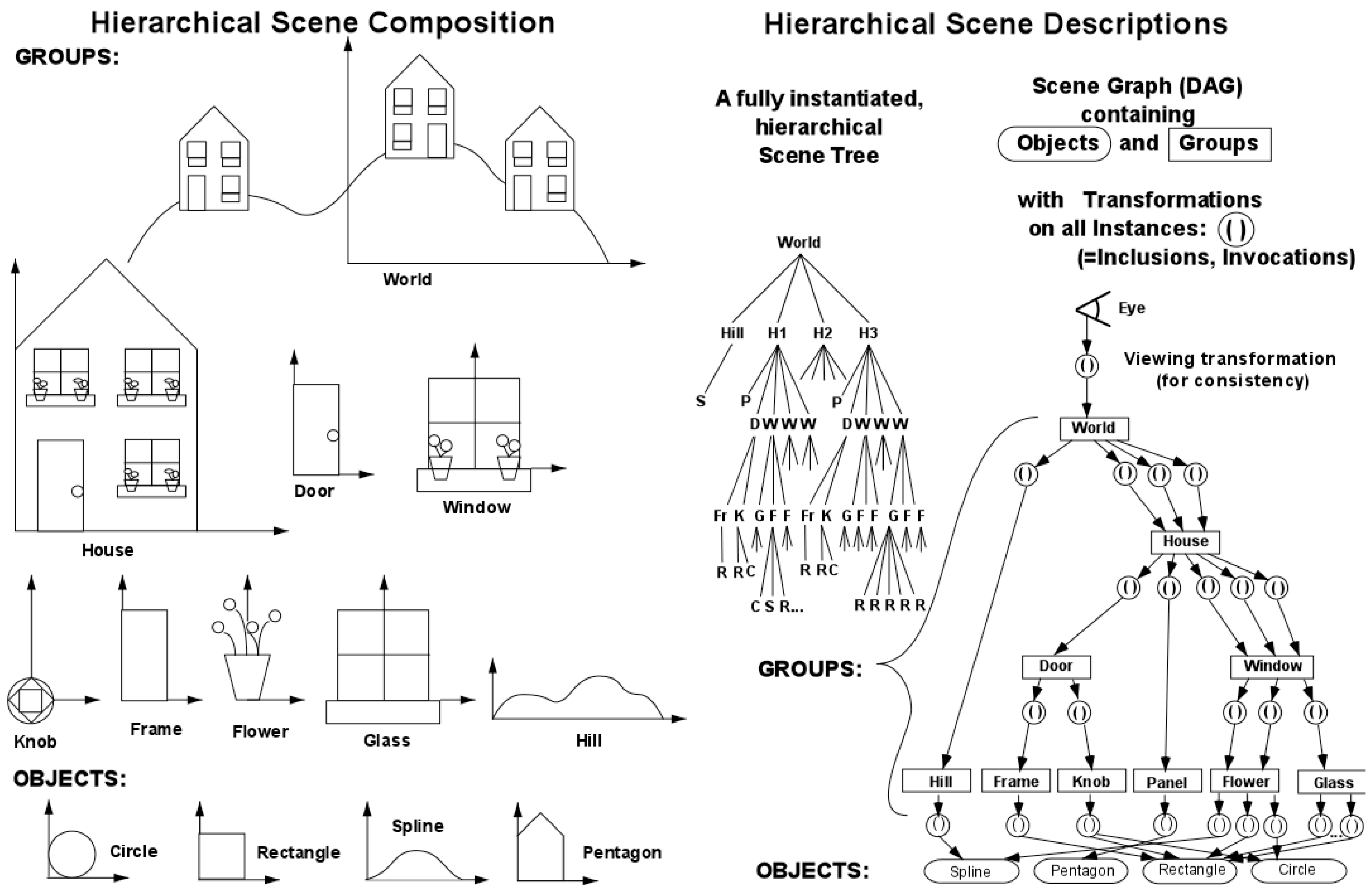
- A name given to various kinds of graph structures (nodes connected together) used to represent scenes
- Simplest form: tree
  - Just saw this
  - Every node has one parent
  - Leaf nodes are identified with objects in the scene



Concatenation and hierarchy

- Transforms associated with nodes or edges
- Each transform applies to all geometry below it
  - Want group transform to transform each member
  - Members already transformed—concatenate
- Frame transform for object is product of all matrices along path from root
  - Each object's transform describes relationship between its local coordinates and its group's coordinates
  - Frame-to-world transform is the result of repeatedly changing coordinates from group to containing group

Hierarchical scene



Variants of the scene graph

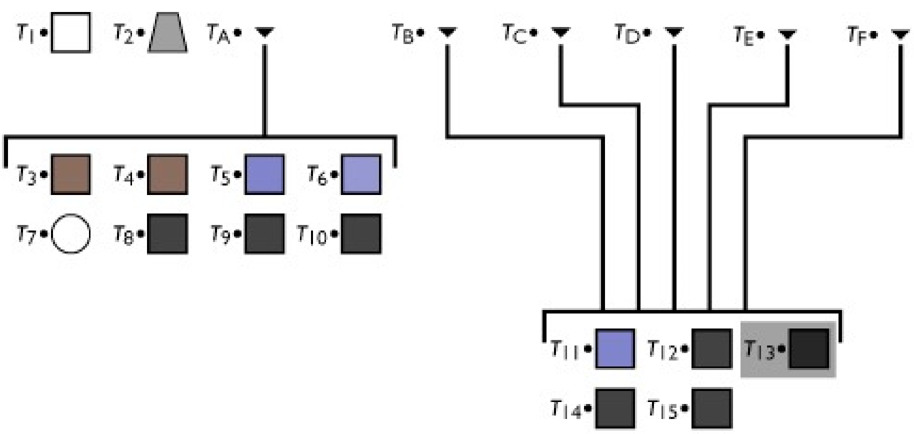
- Parenting
  - Allow any object to have child objects
  - Every object is effectively also a group
  - Common in 3D modeling packages
- Instancing
  - Allow objects to belong to multiple parents/groups
  - Creates multiple copies of geometry

Instances

- Simple idea: allow an object to be a member of more than one group at once
  - Transform different in each case
  - Leads to linked copies
  - Single editing operation changes all instances

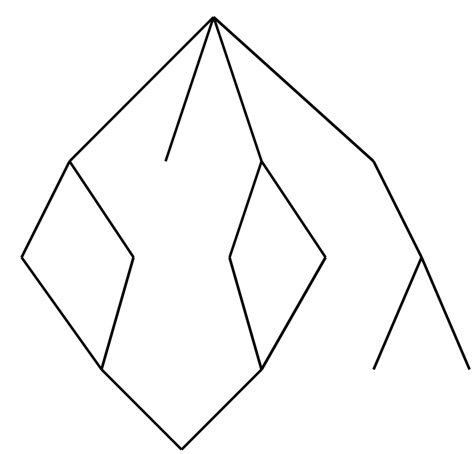
Example

- Allow multiple references to nodes
  - Reflects more of drawing structure
  - Allows editing of repeated parts in one operation



The scene graph (with instances)

- With instances, there is no more tree
  - An object that is instanced multiple times has more than one parent
- Transform tree becomes a directed acyclic graph (DAG)
  - Group is not allowed to contain itself, even indirectly
- Transforms still accumulate along path from root
  - Now paths from root to leaves are identified with scene objects



Implementing a hierarchy

- Define shapes and groups as derived from single class

```
abstract class Shape {
    void draw();
}

class Square extends Shape {
    void draw() {
        // draw unit square
    }
}

class Circle extends Shape {
    void draw() {
        // draw unit circle
    }
}
```

Implementing traversal

- Pass a transform down the hierarchy
  - Before drawing, concatenate

```
abstract class Shape {
    void draw(Transform t_c);
}

class Square extends Shape {
    void draw(Transform t_c) {
        // draw t_c * unit square
    }
}

class Circle extends Shape {
    void draw(Transform t_c) {
        // draw t_c * unit circle
    }
}
```

```
class Group extends Shape {
    Transform t;
    ShapeList members;
    void draw(Transform t_c) {
        for (m in members) {
            m.draw(t_c * t);
        }
    }
}
```

Basic scene graph operations

- Editing a transformation
  - Good to present usable user interface
- Getting transform of object in world frame
  - Traverse path from root to leaf
- Grouping and ungrouping
  - Can do these operations without moving anything
  - Group: insert identity node
  - Ungroup: remove node, push transform to children
- Reparenting
  - Move node from one parent to another
  - Can do without altering position

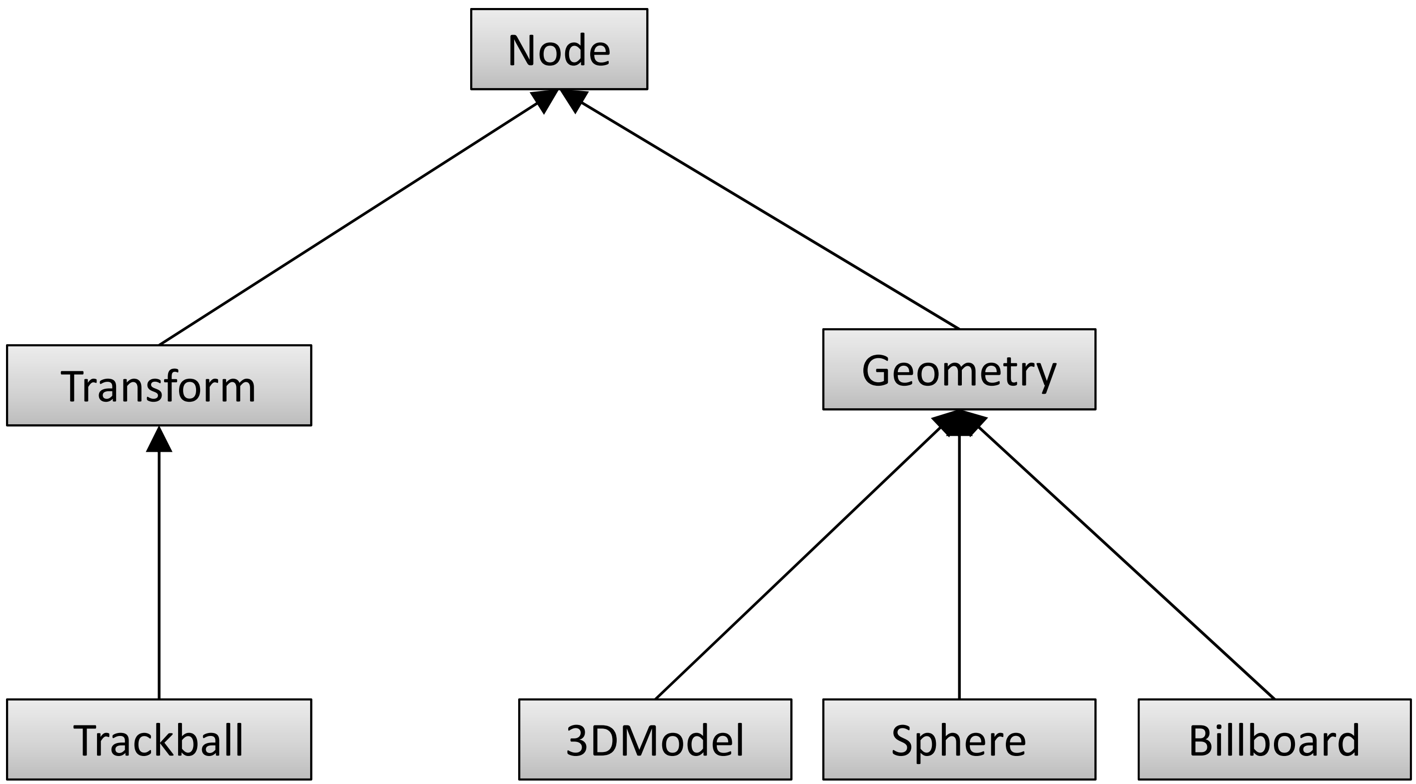
Adding more than geometry

- Objects have properties besides shape
  – Color, shading parameters
  – Approximation parameters (e.g., precision of subdividing curved surfaces into triangles)
  – Behavior in response to user input
  – Etc.
- Setting properties for entire groups is useful
  – Paint entire window green
- Many systems include some kind of property nodes
  – In traversal they are read as, e.g., “set current color”

Scene graph variations

- Where transforms go
  - In every node
  - On edges
  - In group nodes only
  - In special Transform nodes
- Tree vs. directed acyclic graph (DAG)
- Nodes for cameras and lights?

Example class hierarchy



Class hierarchy

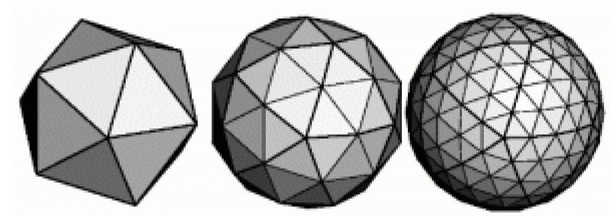
- Node
  - Common base class for all node types
  - Stores node name, pointer to parent, bounding box
- Geometry

  - Sets the modelview matrix to the current C matrix
  - Has a class method which draws its associated geometry
- Transform

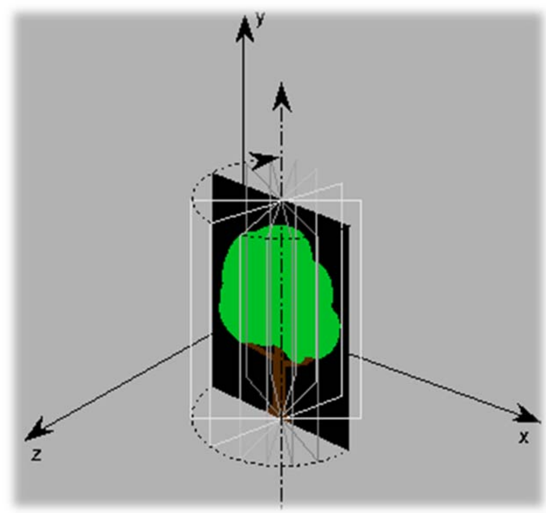
  - Stores list of children
  - Stores 4x4 matrix for affine transformation

Class hierarchy

- Sphere
  - Derived from Geometry node
  - Pre-defined geometry with parameters, e.g., for tesselation level (number of triangles), solid/wireframe, etc.

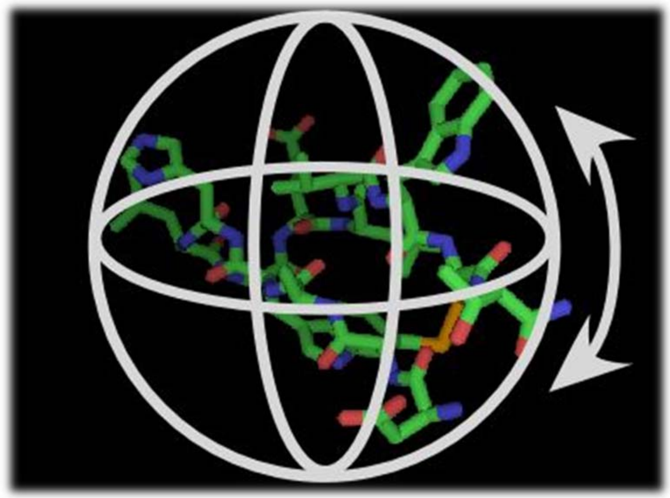


- Billboard
  - Special geometry node to display an image always facing the viewer

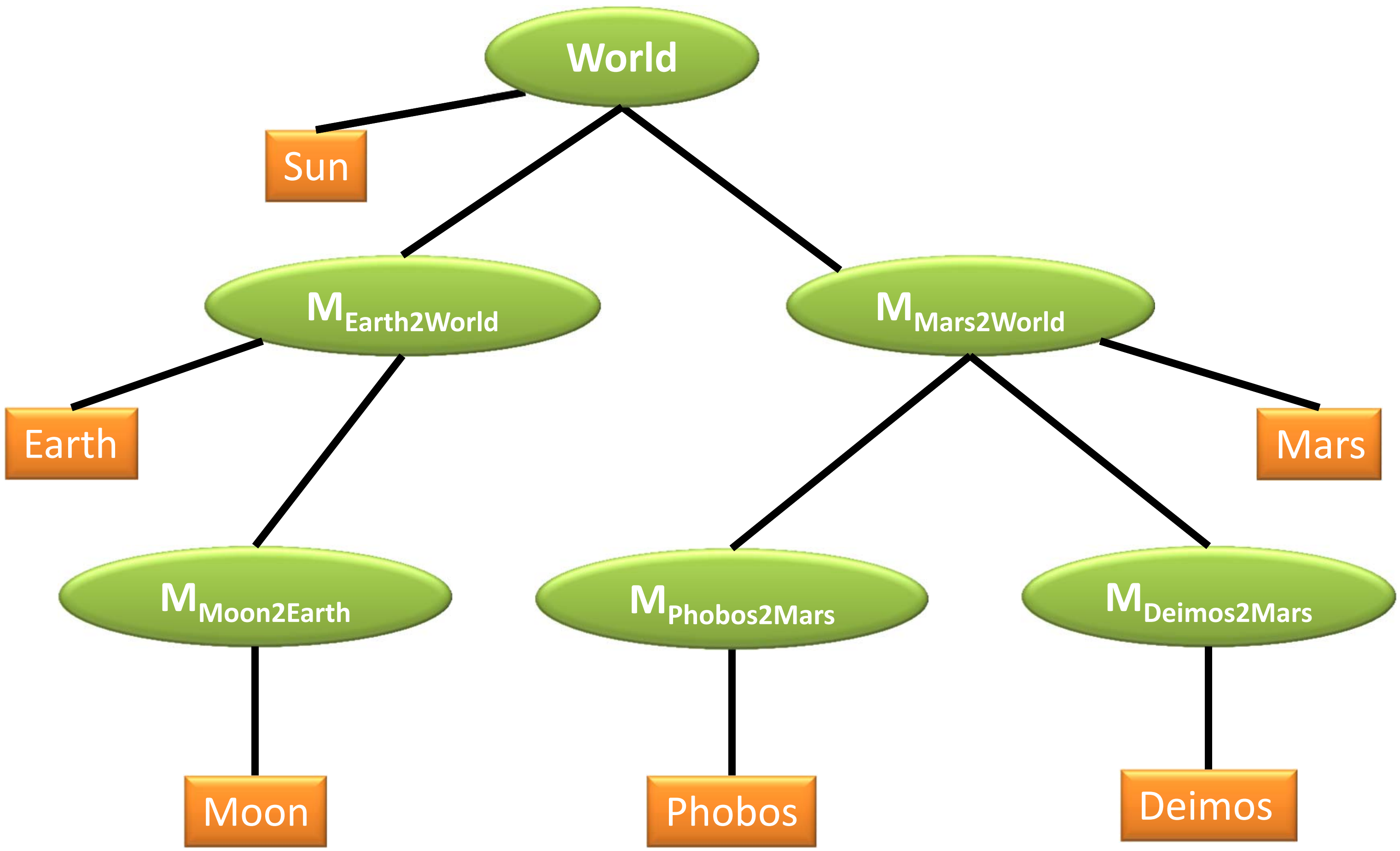


Class hierarchy

- 3DModel
  - Takes file name to load 3D model file
- Trackball
  - Creates the matrix transformation based on a virtual trackball controlled with the mouse



Scene graph for solar system



Building the solar system

```
// create sun:
world = new Transform();
world.addChild(new Model(“Sun.obj”));

// create planets:
earth2world = new Transform(…);
mars2world = new Transform(…);
earth2world.addChild(new Model(“Earth.obj”));
mars2world.addChild(new Model(“Mars.obj”));
world.addChild(earth2world);
world.addChild(mars2world);

// create moons:
moon2earth = new Transform(…);
phobos2mars = new Transform(…);
deimos2mars = new Transform(…);
moon2earth.addChild(new Model(“Moon.obj”));
phobos2mars.addChild(new Model(“Phobos.obj”));
deimos2mars.addChild(new Model(“Deimos.obj”));
earth2world.addChild(moon2earth);
mars2world.addChild(phobos2mars);
mars2world.addChild(deimos2mars);
```

Transformation calculations

- moon2world = moon2earth * earth2world;
- phobos2world = phobos2mars * mars2world;
- deimos2world = deimos2mars * mars2world;

Scene Rendering

- Recursive draw calls

```
Transform::draw(Matrix4 M)
{
  M_new = M * MT;   // MT is a class member
  for all children
    draw(M_new);
}

Geometry::draw(Matrix4 M)
{
  setModelMatrix(M);
  render(myObject);
}
```

Initiate rendering with
`world->draw(IDENTITY);`

Modifying the scene

- Change tree structure
  - Add, delete, rearrange nodes
- Change node parameters
  - Transformation matrices
  - Shape of geometry data
  - Materials
- Create new node subclasses
  - Animation, triggered by timer events
  - Dynamic “helicopter-mounted” camera
  - Light source
- Create application dependent nodes
  - Video node
  - Web browser node
  - Video conferencing node
  - Terrain rendering node

Drawing a scene graph

- Draw scene with pre-and-post-order traversal
  - Apply node, draw children, undo node if applicable
- Nodes can carry out any function
  - Geometry, transforms, groups, color, etc.
- Requires stack to "undo" post children
  - Transform stacks in OpenGL
- Caching and instancing possible
- Remember: instances make it a directed acyclic graph (DAG), not strictly a tree

Benefits of a scene graph

- Can speed up rendering by efficiently using low-level API
  - Avoid state changes in rendering pipeline
  - Render objects with similar properties in batches (geometry, shaders, materials)
- Change parameter once to affect all instances of an object
- Abstraction from low level graphics API
  - Easier to write code
  - Code is more compact
- Can display complex objects with simple APIs
  - Example: osgEarth class provides scene graph node which renders a Google Earth-style planet surface with progressive refinement and data streaming from server

Graphics system architecture

- Interactive Applications
  - Video games, scientific visualization, virtual reality
- Rendering Engine, Scene Graph API
  - Implement functionality commonly required in applications
  - Back-ends for different low-level APIs
  - No broadly accepted standards
  - Examples: OpenSceneGraph, SceniX, Torque, Ogre
- Low-level graphics API
  - Interface to graphics hardware
  - Highly standardized: OpenGL, Direct3D

Scene graph APIs

- OpenSceneGraph (www.openscenegraph.org)
  - For scientific visualization, virtual reality, GIS (geographic information systems)
- NVIDIA SceniX
  - Optimized for shader support
  - Support for interactive ray tracing
  - http://www.nvidia.com/object/scenix-home.html
- Torque 3D
  - Open source game engine
  - For Windows and browser-based games
  - http://www.garagegames.com/products/torque-3d
- Ogre3D
  - Open source rendering engine
  - For Windows, Linux, OSX, Android, iOS, Javascript
  - http://www.ogre3d.org/

Commonly offered functionality

- Resource management
  - Content I/O (geometry, textures, materials, animation sequences)
  - Memory management
- High-level scene representation
  - Graph data structure
- Rendering
  - Optimized for efficiency (e.g., minimize OpenGL state changes)