CSc 165
Computer Game Architecture

06 - Scenegraphs

Overview

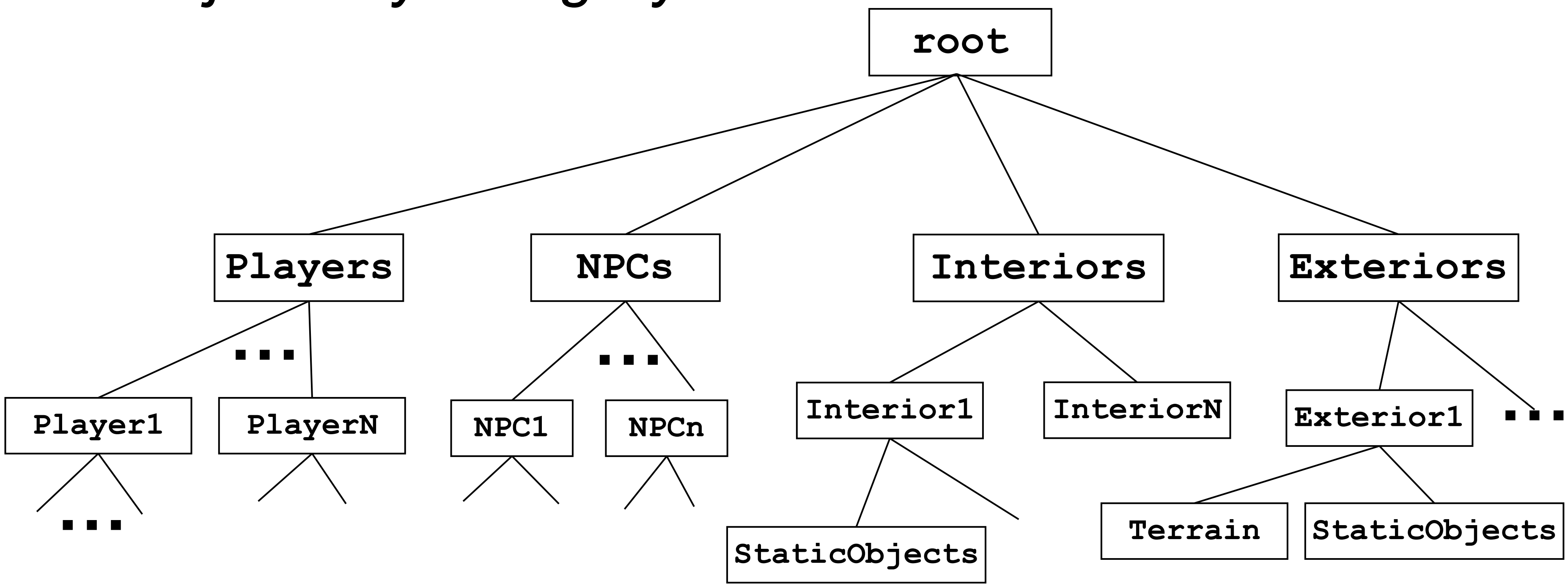
- The Scenegraph Concept
- Common Scenegraph APIs
- Scenegraph Node Hierarchy
- Scenegraph Traversal
- Node Controllers
- Render Queues

Scenegraph

- Allows game programmer to organize game objects into hierarchies
- How to make use of it is up to the programmer
- There are many uses

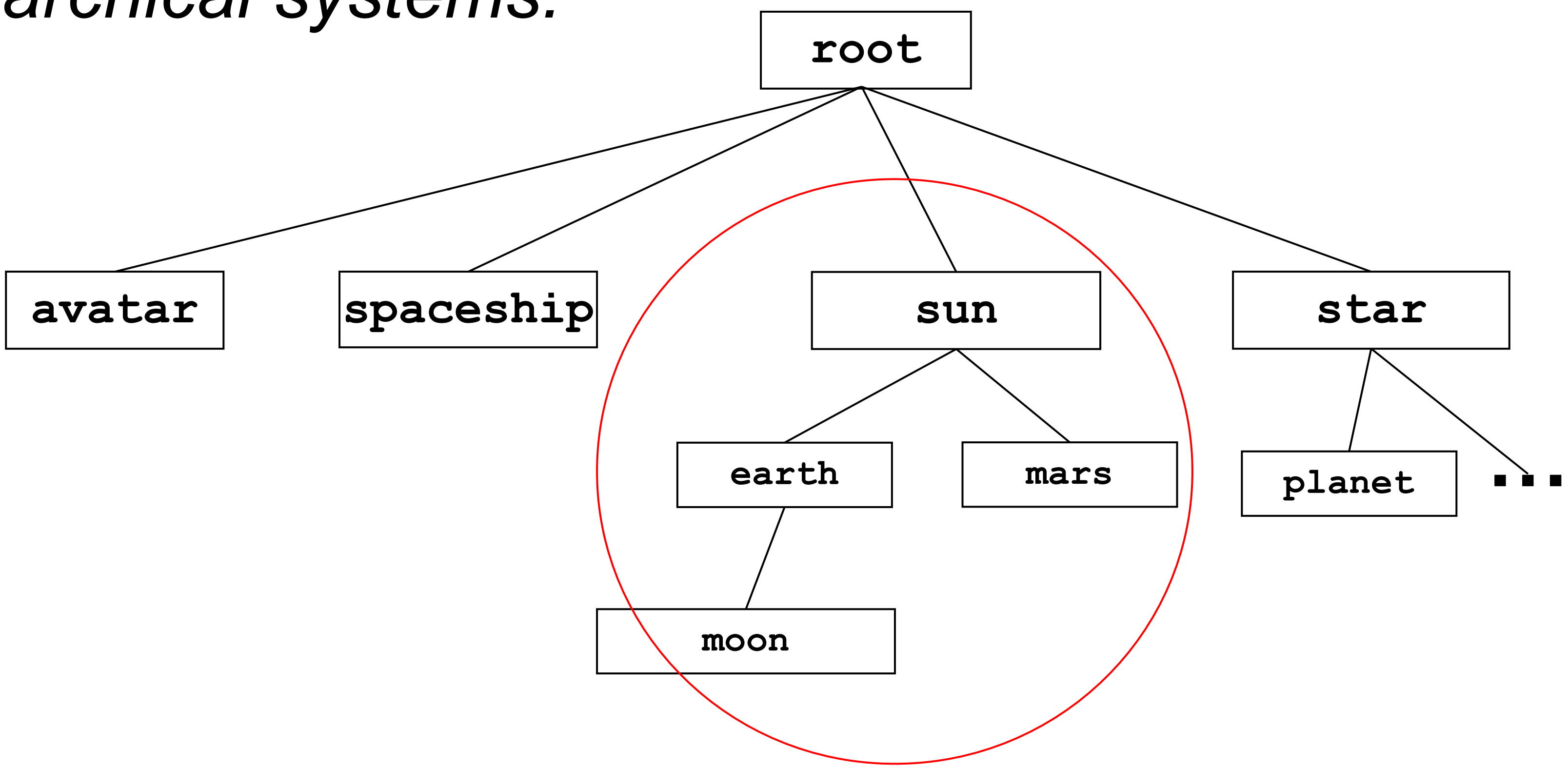
Hierarchical Scenegraphs

Can be used to organize objects by category:



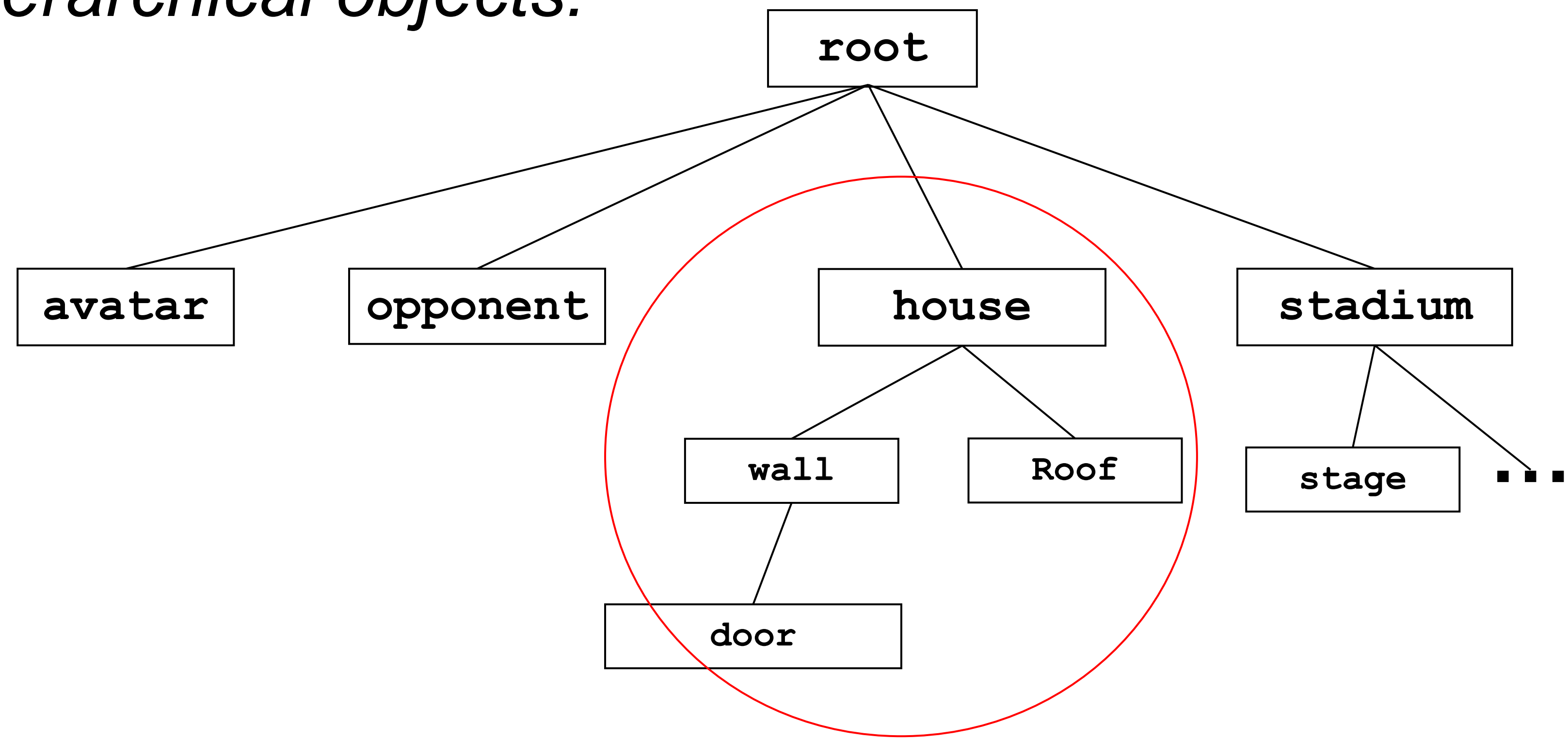
Hierarchical Scenegraphs

can be used for building hierarchical systems:



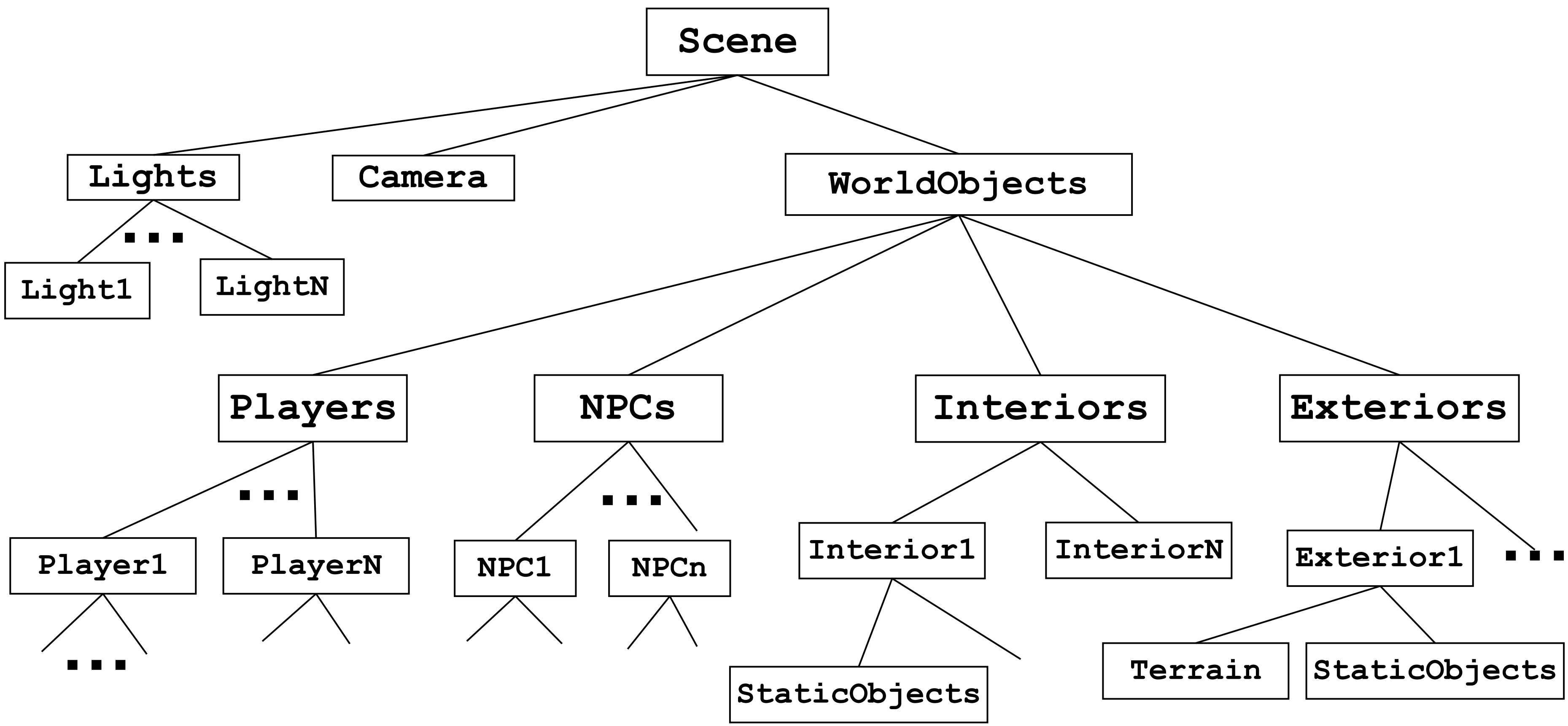
Hierarchical Scenegraphs

can be used for building hierarchical objects:



Hierarchical Scenegraphs

Some scenegraphs include the lights and cameras (TAGE does not):

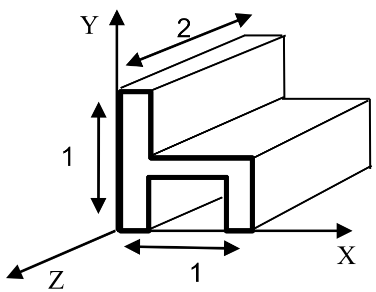


Common Scenegraph APIs

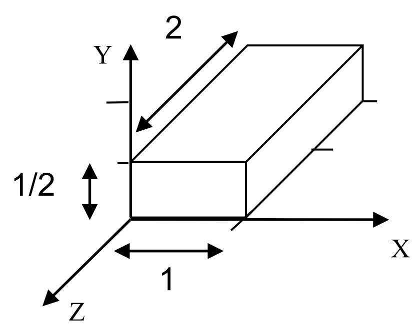
- OpenSceneGraph
  `www.openscenegraph.org`
- Java3D (Sun/Java Community)
  `https://java3d.dev.java.net`
- X3D (Web3D Consortium)
  `www.web3d.org/x3d`
- OpenSG
  `http://www.opensg.org`
- OpenInventor (SGI)
  `http://oss.sgi.com/projects/inventor`
- Xith3D
  `https://xith3d.dev.java.net`

Building a Scenegraph (1)

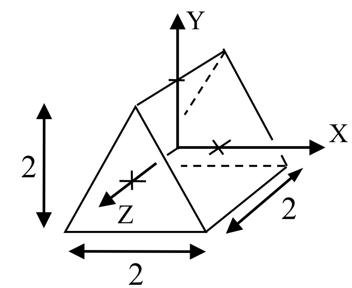
Primitive objects ("models") defined in "local space":



"Bench"



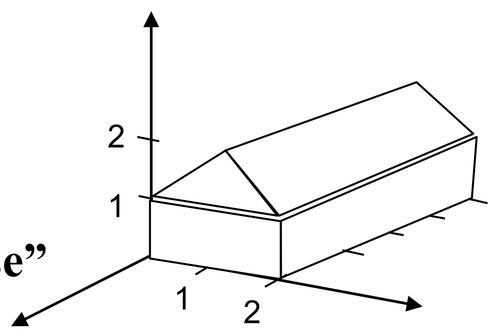
"Wall"



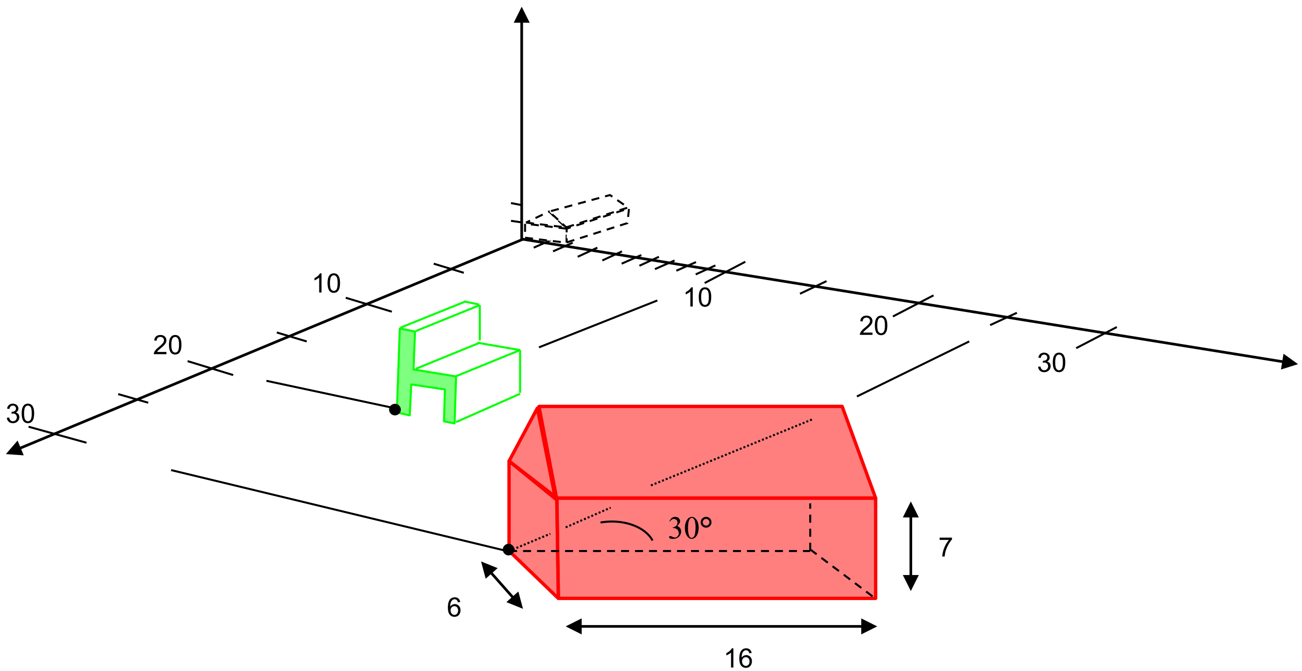
"Roof"

Hierarchical object: House = Wall + Roof:

"House"

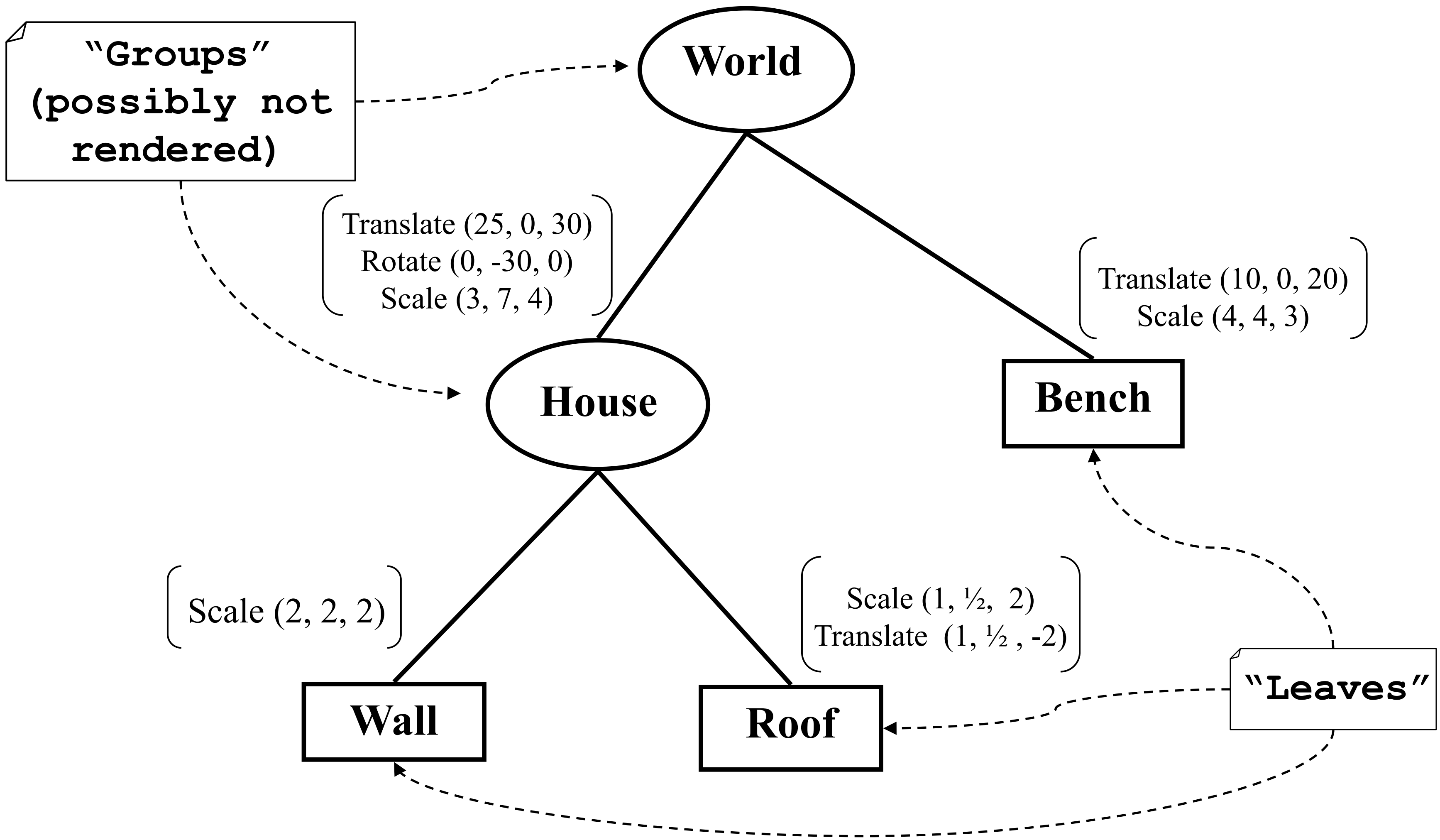


Building a Scenegraph (2)

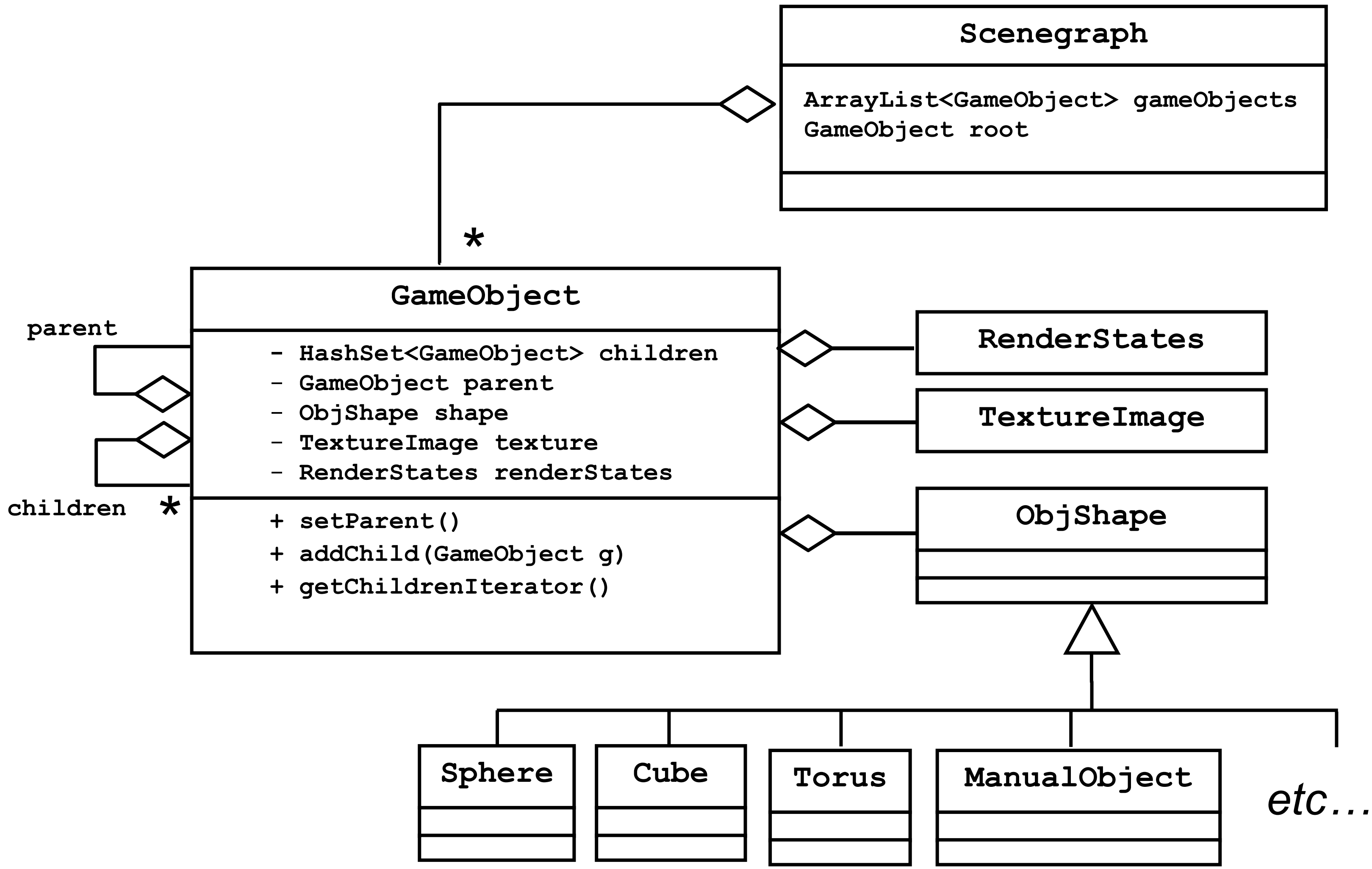



```
World = House ( Transform = ([Translate(25,0,30)] x [Rotate(0,-30,0)]
                x [Scale(3,7,4)]),   Color = Red )
      +
      Bench ( Transform = ([Translate(10,0,20)] x [Scale(4,4,3)]),
              Color = Green)
```

Building a Scenegraph (3)



Scenegraph & Game Objects



(Naïve) Scenegraph Traversal

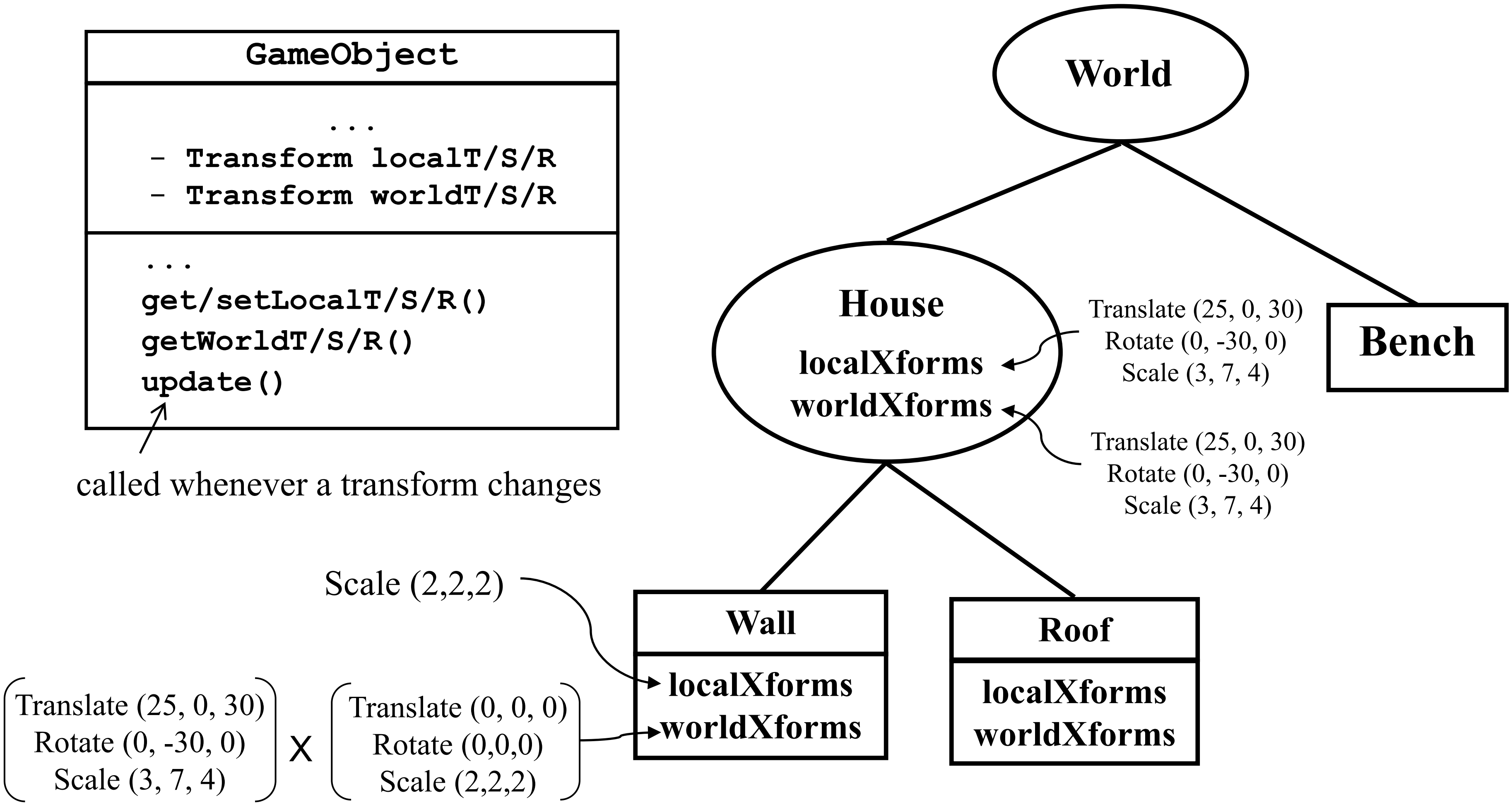
Renderer :

```
displayScenegraph()
{   ...
    save current xform matrix
    sceneGraphRoot.draw()
    restore xform matrix
    ...
}
```

SceneNode :

```
draw()
{   save r.xform
    concatenate transforms onto r.xform
    if visible, render(this)

    for each child:
       child.draw()

    restore r.xform
}
```

Improved Scenegraph Structure



Example Game Application

```
public class SolarSystem extends VariableFrameRateGame
{   public void buildObjects()
    {   ...
        // ----- the "earth" object is a child of the "sun" object
        earth = new GameObject(sun, pyrS);
        earth.propagateTranslation(true);
        earth.propagateRotation(false);

        // ----- the "moon" object is a child of the "earth" object
        moon = new GameObject(earth, torS, brick);
        moon.propagateTranslation(true);
        moon.propagateRotation(false);
        ...
    }

    ...continued...
```

Updating GameObject's Positions

```
protected void update()
{   if (this != root)
    {   if (propagateTranslation)
        {   Vector4f loc = (new Vector4f(0,0,0,1)).mul(localTranslation);
            if (applyParentRotationToPosition) loc.mul(parent.getWorldRotation());
            if (applyParentScaleToPosition)   loc.mul(parent.getWorldScale());
            loc.mul(parent.getWorldTranslation());
            worldTranslation.translation(loc.x(), loc.y(), loc.z());
        }
        else
        {   worldTranslation = new Matrix4f(localTranslation);
        }
        ... (same for rotation and scale)
    }
    Iterator<GameObject> i = children.iterator();
    while (i.hasNext()) (i.next()).update();
}
```

ENGINE
(GameObject)

Specifying Node Behavior

Certain operations on nodes occur often:

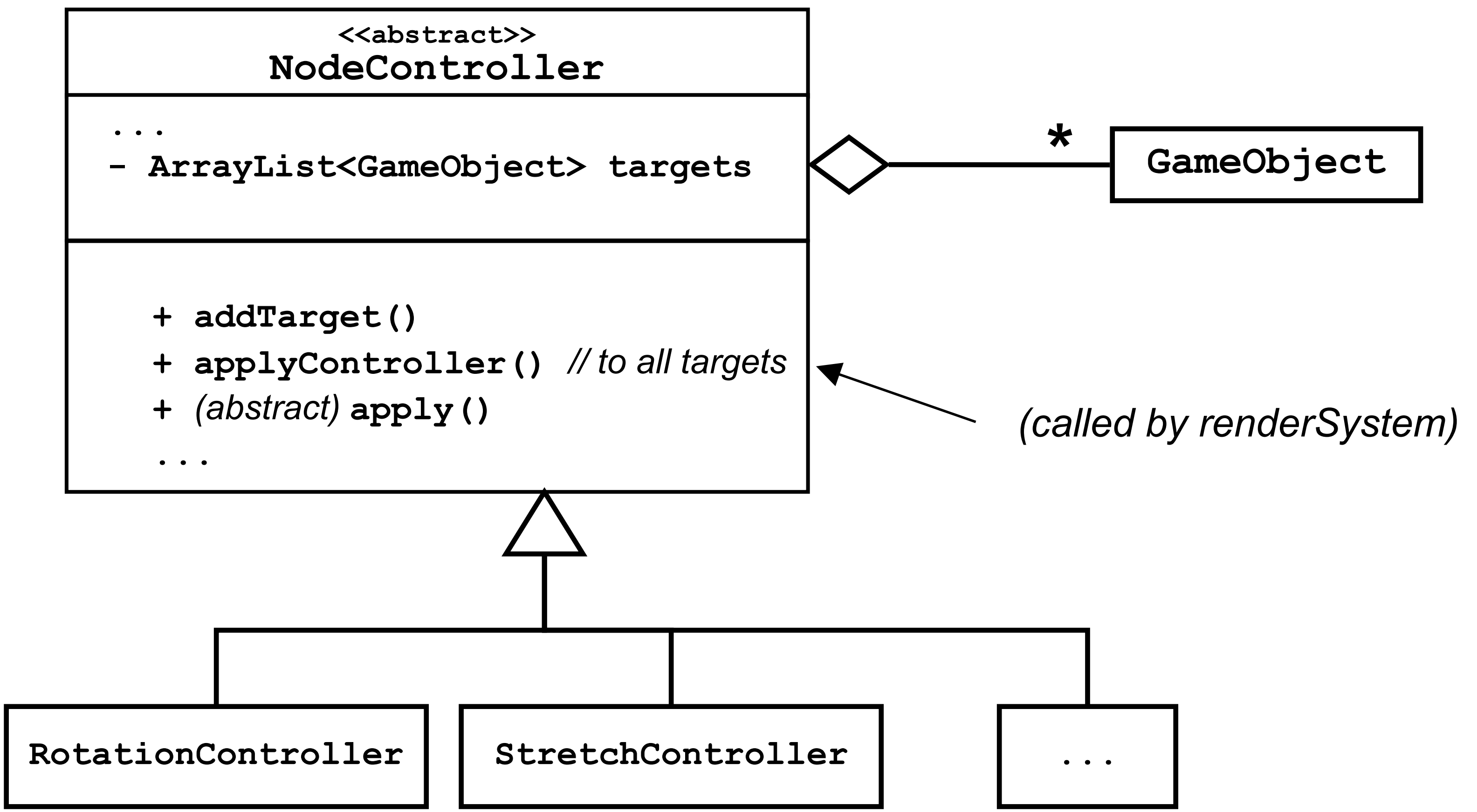
- Spatial transforms (rotate, scale, translate)
- Lifetime expiration
- Change in appearance (transparency, etc.)
- … many others …

Two approaches:

- Require <u>game application</u> to support such changes
- Provide <u>game engine classes</u> to manage changes

Node Controllers

Controllers are attached to scene nodes



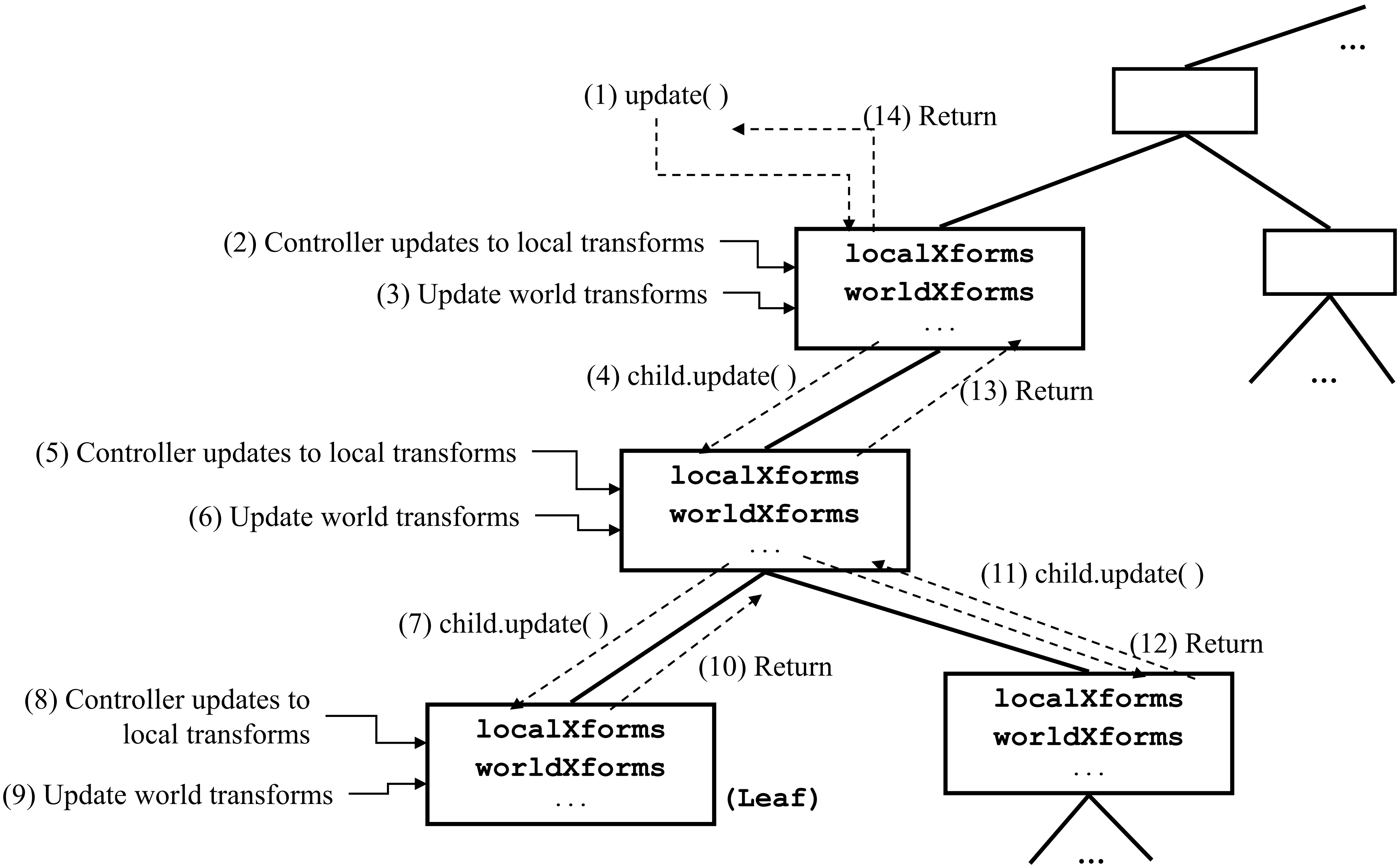
Example: StretchController

```
public class StretchController extends NodeController
{
    private float scaleRate = .0003f;
    private float cycleTime = 2000.0f;
    private float totalTime = 0.0f;
    private float direction = 1.0f;
    private Matrix4f curScale, newScale;

    public StretchController(Engine e, float ctime)
    {   super();
        cycleTime = ctime;
        newScale = new Matrix4f();
    }

    public void apply(GameObject go)
    {   float elapsedTime = super.getElapsedTime();
        totalTime += elapsedTime/1000.0f;

        if (totalTime > cycleTime)
        {   direction = -direction;
            totalTime = 0.0f;
        }
        curScale = go.getLocalScale();
        float scaleAmt = 1.0f + direction * scaleRate * elapsedTime;
        newScale.scaling(curScale.m00()*scaleAmt, curScale.m11(), curScale.m22());
        go.setLocalScale(newScale);
    }
}
```

Example Game (revisited)

```
public class HelloDolphin extends VariableFrameRateGame
{   ...
    public void initializeGame()
    {   ...
        sc = new StretchController(engine, 2.0f);
        sc.addTarget(dolphinObject);
        engine.getSceneGraph().addNodeController(sc);

        ...
```

Hierarchical Update Sequence



Rendering the SceneGraph

- **`render()`** == 'draw the scene'
- Standard approach: recursive tree-walk
  calling **`draw()`** at each **`SceneNode`**
- Problem: Scenegraph traversal order doesn't
  account for differences in nodes:
  - ***Opaque*** nodes should be rendered front-to-back for speed
  - ***Transparent*** nodes must be rendered after opaque ones,
    and in back-to-front order
- Solution: place nodes in a RenderQueue, then sort the queue based on the above factors.
- TAGE uses a **`RenderQueue`** (but not yet sorting)