Hierarchical Modeling

Geometric Primitives

Remember that most graphics APIs have only a few geometric primitives

- Spheres, cubes, triangles, etc

These primitives are *instanced* in order to apply transforms

Instance Transforms

We start with a prototype object (symbol)

Each appearance of the object is an instance

Instance defines transforms:

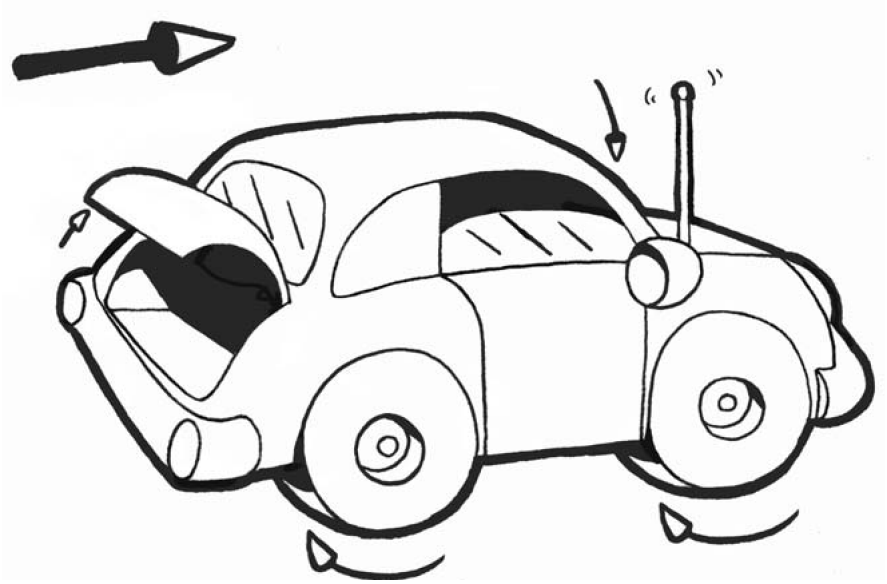
- Scale, rotate, translate, etc

Relationships between Instances

Objects often consist of sub-objects

Consider: A car chassis has four wheels

- 2 symbols
- 5 instances



Rotational wheel speed determines forward motion

How to represent this?

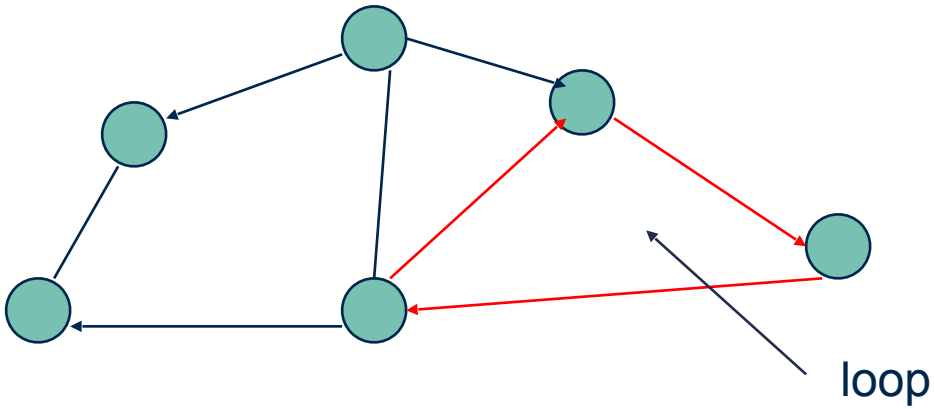
Graphs

- Set of nodes and edges
- Edge connects a pair of nodes
- Direction reflects relationship

Any problems with this?

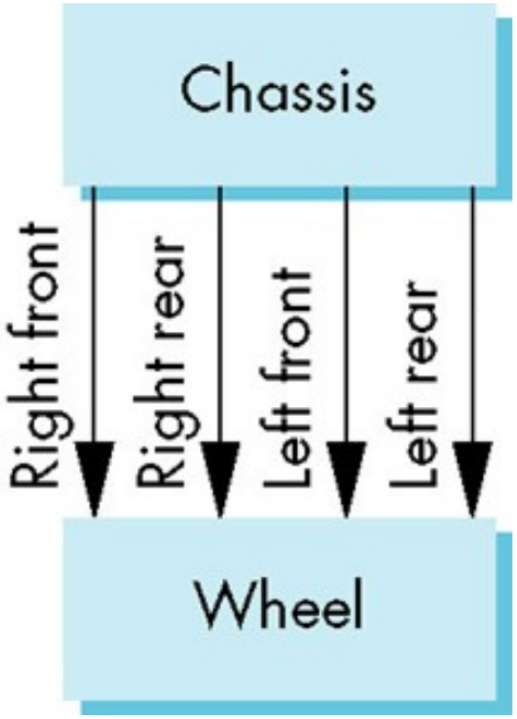
Graphs

Graph should be acyclic!

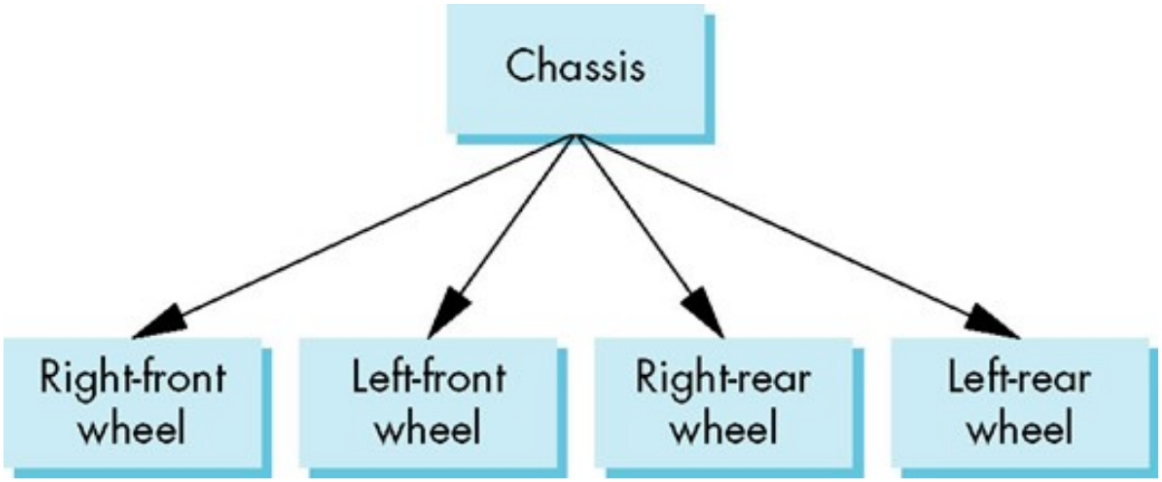


What might our car representation look like?

Directed Acyclic Graph



Trees



(Same idea as the DAG)

What information is encoded in the nodes and edges?

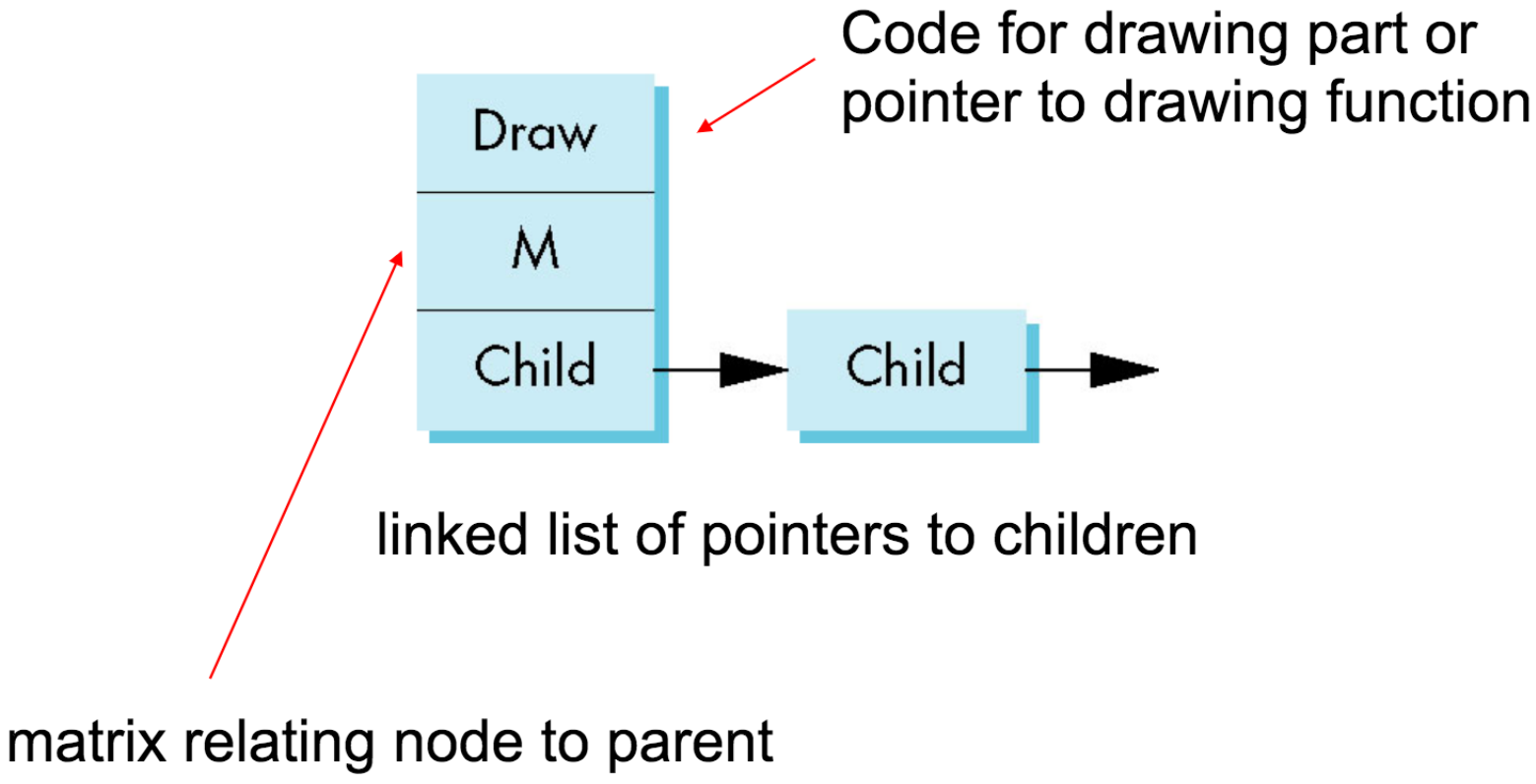
Modeling with Trees

Nodes contain:

- What to draw
- Drawing attributes
- Pointers to children

Edges can contain:

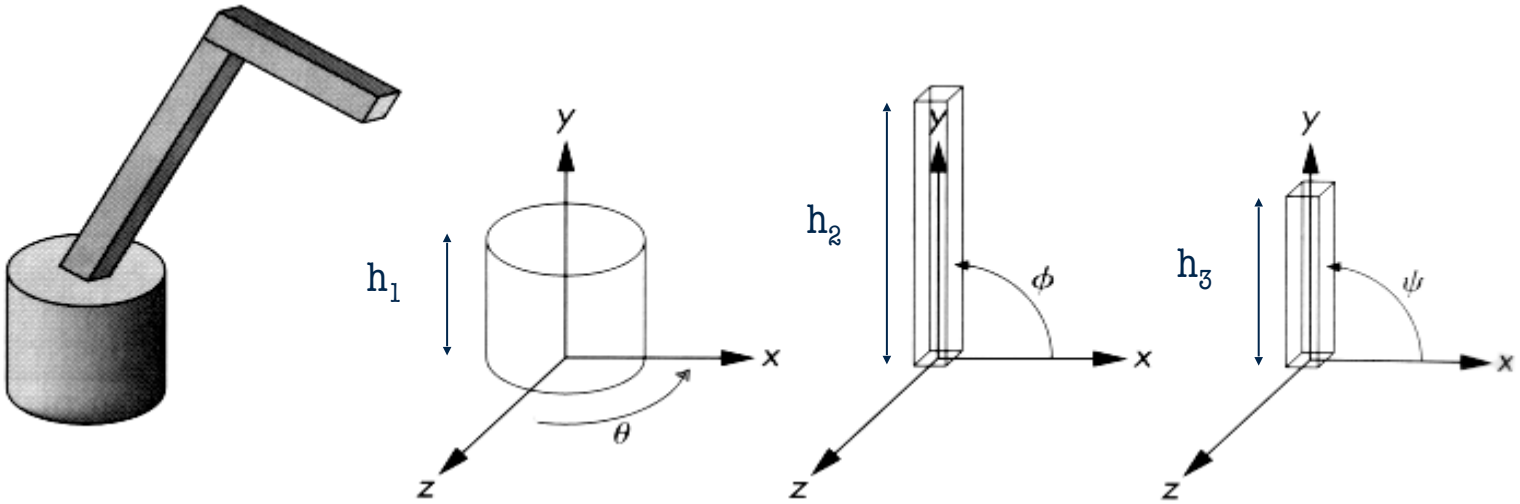
- Geometric transformations (or can be stored in node)


Code for drawing part or
pointer to drawing function
Draw
M
Child
Child
linked list of pointers to children
matrix relating node to parent

Consider this Robot Arm

3 degrees of freedom:

- Base rotates about its vertical axis by θ
- Lower arm rotates in its *xy*-plane by ϕ
- Upper arm rotates in its *xy*-plane by ψ



Relationships in Robot Arm

Base rotates independently

- Position depends on θ

Lower arm attached to base

- Position depends on θ
- Position depends on ϕ

Upper arm attached to lower arm

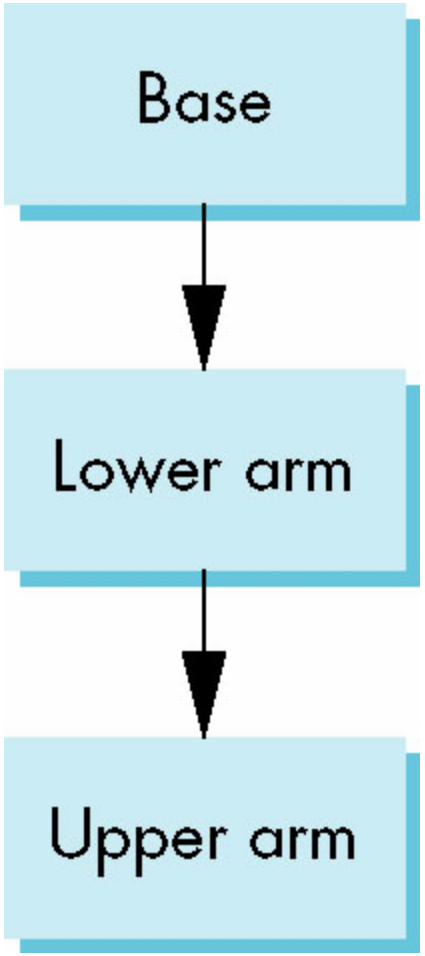
- Position depends on θ
- Position depends on ϕ
- Position depends on ψ

In-Class Activity

Draw a tree structure that represents this robot arm

Define the matrices at each joint (edge) that will position the robot parts correctly

Robot Tree Model Example



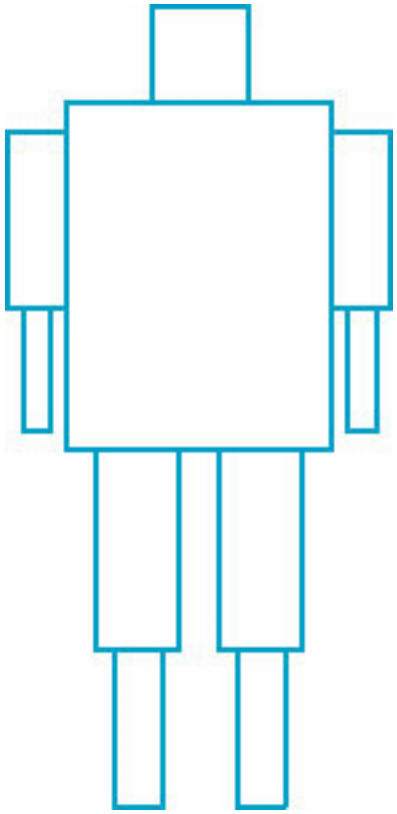
Robot Transform Matrices Example

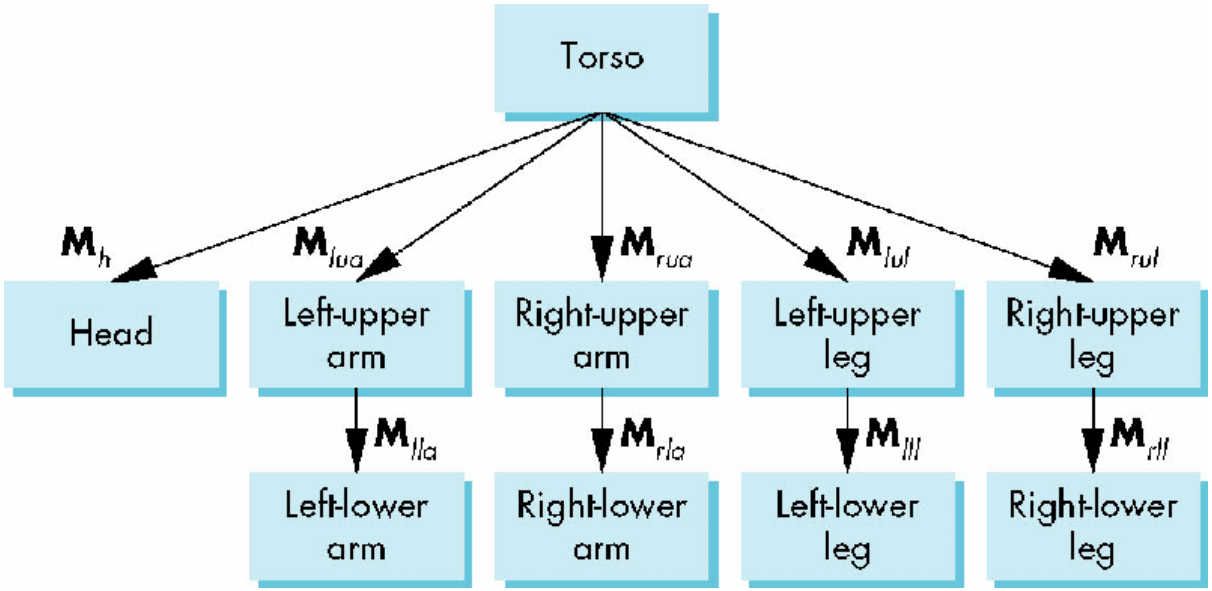
- Rotation of base: $\mathbf{R}_b$
  - Apply $\mathbf{M} = \mathbf{R}_b$ to base
- Translate lower arm relative to base: $\mathbf{T}_{lu}$
- Rotate lower arm around joint: $\mathbf{R}_{lu}$
  - Apply $\mathbf{M} = \mathbf{R}_b\ \mathbf{T}_{lu}\ \mathbf{R}_{lu}$ to lower arm
- Translate upper arm relative to lower arm: $\mathbf{T}_{uu}$
- Rotate upper arm around joint: $\mathbf{R}_{uu}$
  - Apply $\mathbf{M} = \mathbf{R}_b\ \mathbf{T}_{lu}\ \mathbf{R}_{lu}\ \mathbf{T}_{uu}\mathbf{R}_{uu}$ to upper arm

OpenGL Code

```
robot_arm(){
    glm::rotate(theta, 0.0, 1.0, 0.0);
    base();
    glm::translate(0.0, h1, 0.0);
    glm::rotate(phi, 0.0, 0.0, 1.0);
    lower_arm();
    glm::translate(0.0, h2, 0.0);
    glm::rotate(psi, 0.0, 0.0, 1.0);
    upper_arm();
}
```

Humanoid Example





Transformation Matrices

There are 10 relevant matrices:

- $\mathbf{M}$ positions and orients entire figure through torso (root node)
- $\mathbf{M}_h$ positions head wrt torso
- $\mathbf{M}_{lua}$, $\mathbf{M}_{rua}$, $\mathbf{M}_{lul}$, $\mathbf{M}_{rul}$ position arms and legs wrt torso
- $\mathbf{M}_{lla}$, $\mathbf{M}_{rla}$, $\mathbf{M}_{lll}$, $\mathbf{M}_{rll}$ position lower parts of limbs wrt corresponding upper limbs

Display and Traversal

Display a tree via graph traversal:

- Visits each node once
- Applies correct transformation matrix for position and orientation

Stack-based Traversal

- Set model-view matrix to **M** and draw torso
- Set model-view matrix to $\mathbf{MM}_h$ and draw head
- For left-upper arm need $\mathbf{MM}_{lua}$ and so on
- Rather than recomputing $\mathbf{MM}_{lua}$ from scratch or using an inverse matrix, we can use the matrix stack to store **M** and other matrices as we traverse the tree

Fixed Function Example

```
glPushMatrix()
 torso();
 glRotate3f(…);
 head();
 glPopMatrix();
 glPushMatrix();
 glTranslate3f(…);
 glRotate3f(…);
 left_upper_arm();
 glPopMatrix();
 glPushMatrix();
```

glPushMatrix and glPopMatrix

Now deprecated functions handled matrix stacks

- `glPushMatrix()` duplicates current matrix and adds to stack
- `glPopMatrix()` removes current matrix and replaces it with one below

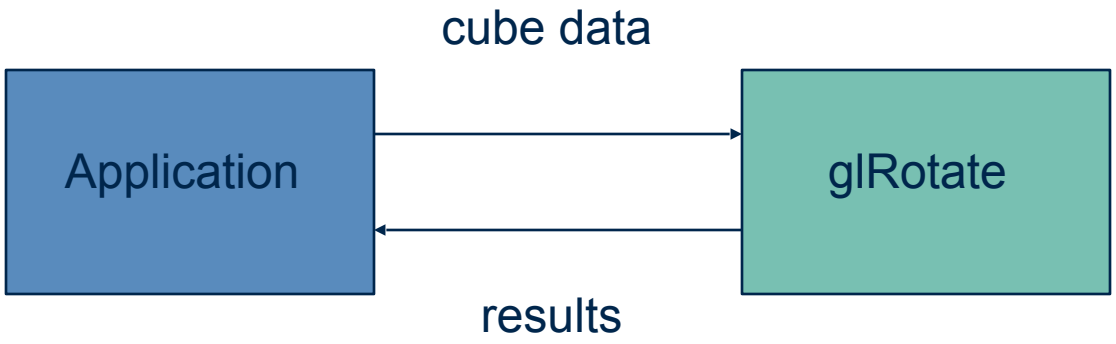
Note that there is no direct “replacement” in Modern OpenGL — you must manage your own matrix stacks!

OpenGL and Objects

OpenGL lacks object orientation

- Properties determined by OpenGL state rather than object characteristics

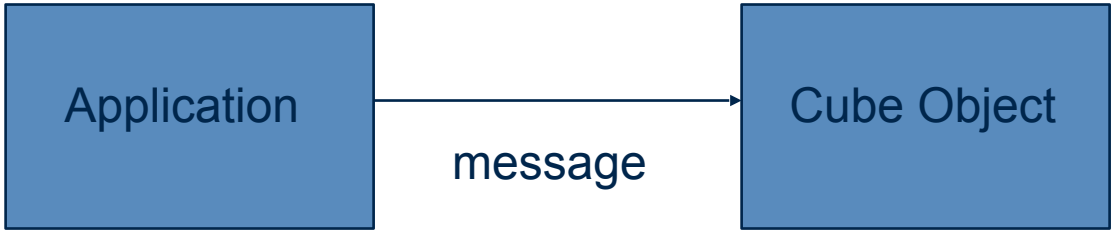
Imperative Model



Representation stored within application

Representation provides information (e.g vertex list, edge list) for functions to calculate results

Object-Oriented Model



Representation stored within object

Application passes message to object

Object has necessary information to transform itself

C/C++

C allows structs for building objects

C++ provides more comprehensive support

- Support for public, private, protected members
- Polymorphism
- Other OOP concepts

Cube Object Example

Creating functionality for scaling, orienting, positioning and coloring a cube object:

```
cube mycube;
mycube.color[0]=1.0;
mycube.color[1]=
  mycube.color[2]=0.0;
mycube.matrix[0][0]= …
```

Cube Object Functions

Use functions to apply changes within the class:

```
mycube.translate(1.0, 0.0,0.0);
mycube.rotate(theta, 1.0, 0.0,
0.0);
setcolor(mycube, 1.0, 0.0,
0.0);
mycube.render();
```

Cube Class Example

```
class cube {
    public:
        float color[3];
        float matrix[4][4];
    // public methods

    private:
   // implementation

}
```

Private Implementation

Implementation of object (e.g. vertex lists etc) should generally be kept private

Using an object should not require understanding the implementation

(Basic OOP concepts still apply!)

Other Objects

Objects can have geometric aspects:

- Cameras
- Light sources

Objects can also be non-geometric:

- Materials
- Colors
- Transformations

Application Code Example

```
cube mycube;
material plastic;
mycube.setMaterial(plastic);

camera frontView;
frontView.position(x ,y, z);
```

Light Object Example

```
class light {     // match Phong model
  public:
   boolean type; //ortho or perspective
   boolean near;
   float position[3];
   float orientation[3];
   float specular[3];
   float diffuse[3];
   float ambient[3];
}
```

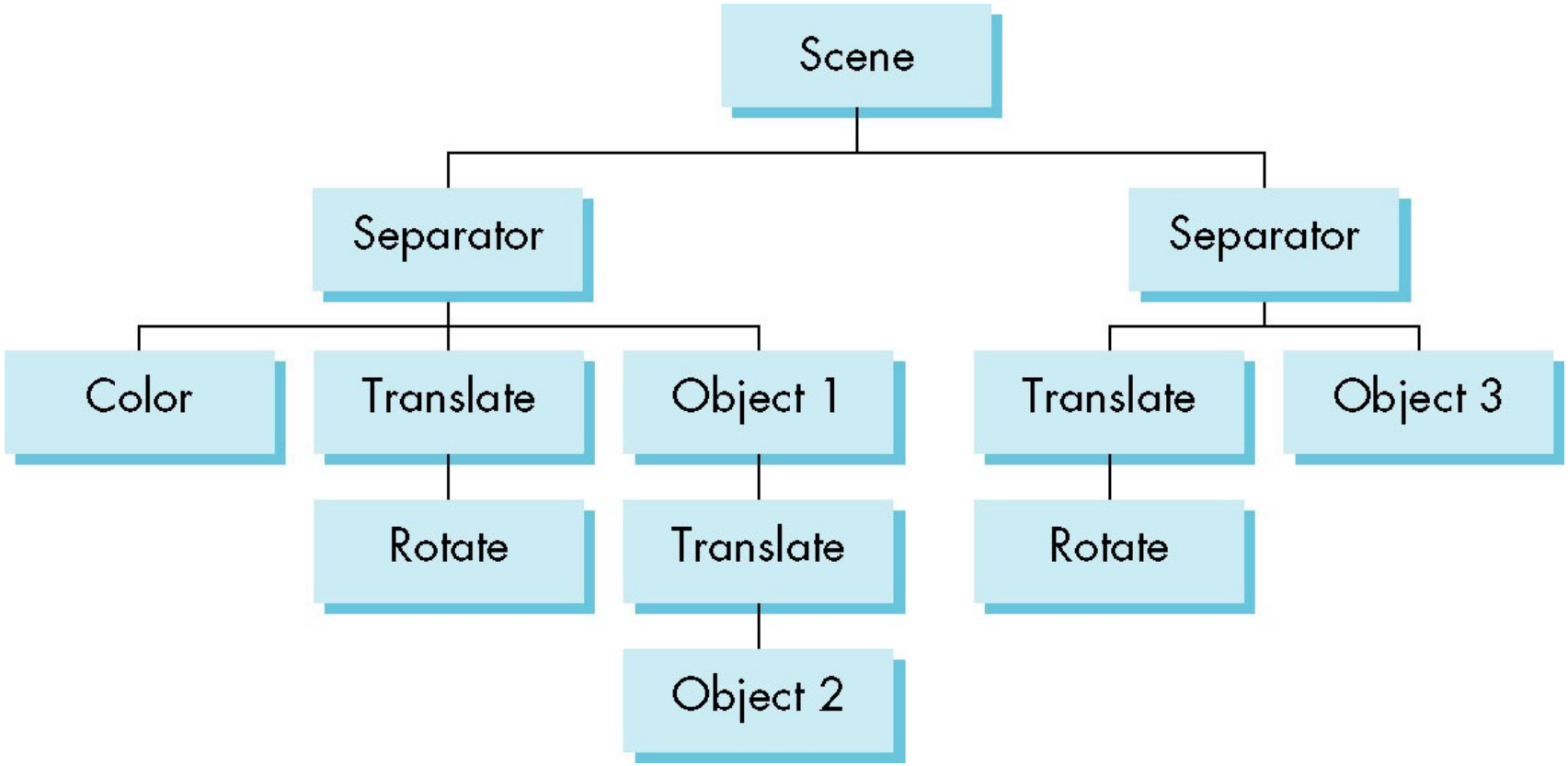
Scene Descriptions

Hierarchical modeling used to structure and manipulate object's underlying form

This idea can be applied to greater scene and its elements (cameras, lights, materials, geometry etc)

Render this scene via tree traversal

Scene Graphs



Objects in Graphics Engines

Goals:

- Provide access to functionality in an intuitive way
- Hide low level details
- Encourage modularity and reuse

Example: Unreal Engine 4

AActor is base class for anything placed or spawned in a level

- Handles collisions, network replication, and graphics transforms

- Examples of its graphics functionality:
  - GetActorLocation() -> FVector
  - GetActorQuat() -> FQuat
  - GetActorRotation() -> FRotator
  - GetActorForwardVector() -> FVector
  - ActorToWorld() -> FTransform

UE4: FVector

Basic storage of 3 (x, y, z) components

Provides various point/vector functionalities:

- Normalize()
- Orthogonal()
- DotProduct()
- Dist()
- ProjectOnTo()
- ToOrientationRotator()
- etc…

UE4: FRotator

Basic storage of pitch, yaw, and roll

Provides various rotation functionality:

- GetInverse()
- Normalize()
- IsNearlyZero()
- Quaternion()
- Serialize()
- Vector()
- etc…

UE4: FTransform

Composed of scale, translate and rotate (quaterion)

Provides various transform functionalities:

- GetDeterminant()
- Inverse()
- Multiply()
- SetLocation/Rotation/Scale3D()
- ToHumanReadableString()
- etc…