Computer Graphics - Scene Graphs

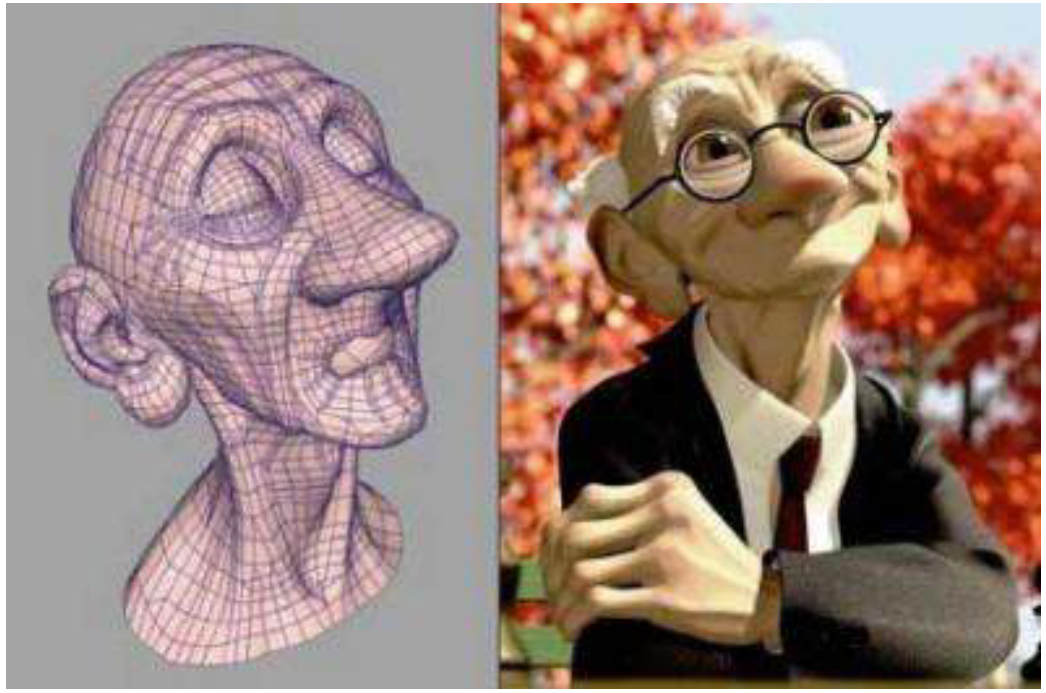
Junjie Cao @ DLUT

Spring 2019

http://jjcao.github.io/ComputerGraphics/

The representations of Mesh && Scene

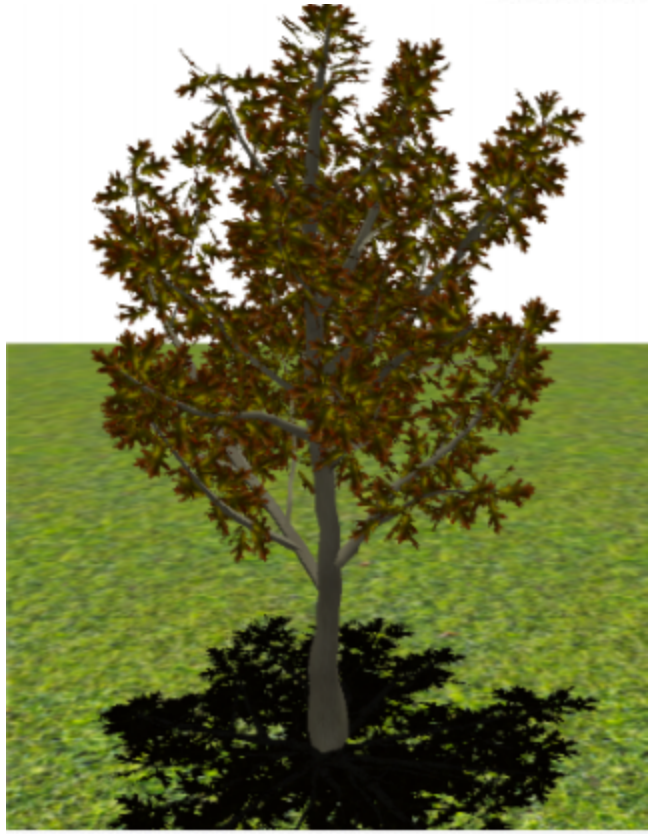
- Mesh: {triangles}, an object in a scene
- Scene: {objects} in desired positions => a great many transformations
- What => easier scene manipulation?
- Most scenes admit to a hierarchical organization =>
- Scene graph



Hierarchical Models

- Many graphical objects are structured
- Structure often naturally hierarchical
  - Wheels of a car
  - Arms or legs of a figure
  - Chess pieces

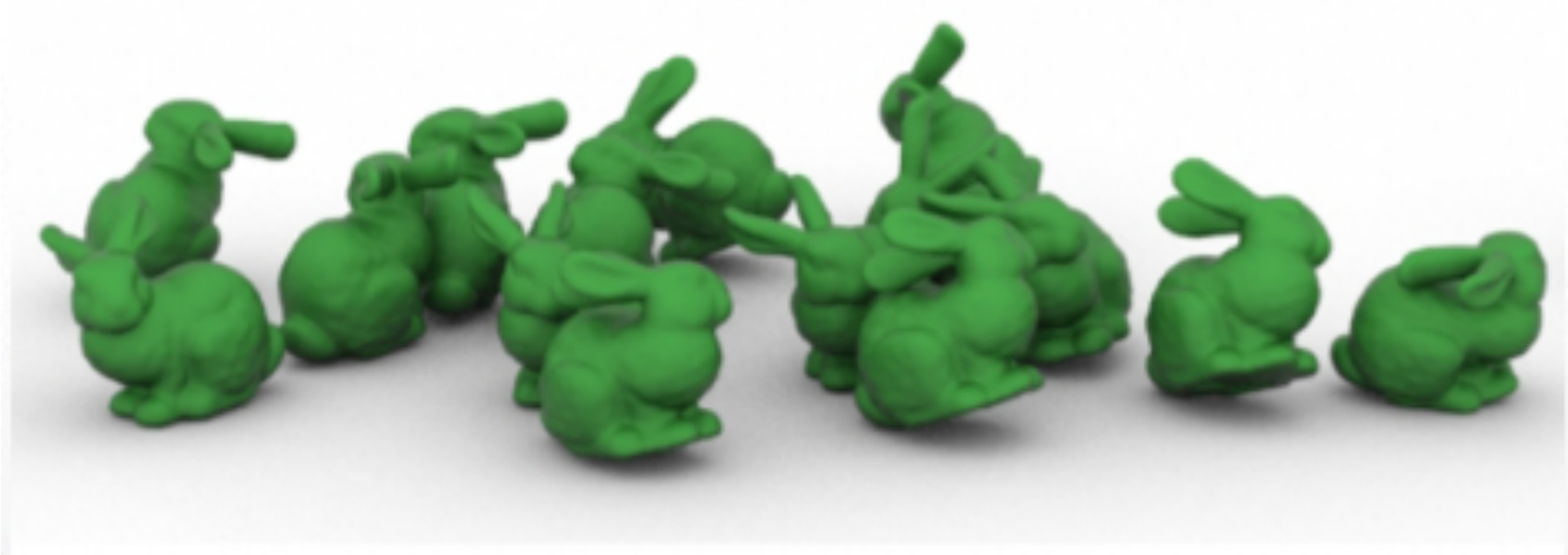
- Exploit structure for
  - Efficient rendering
    - Example: tree leaves
  - Concise specification of model parameters
    - Example: joint angles



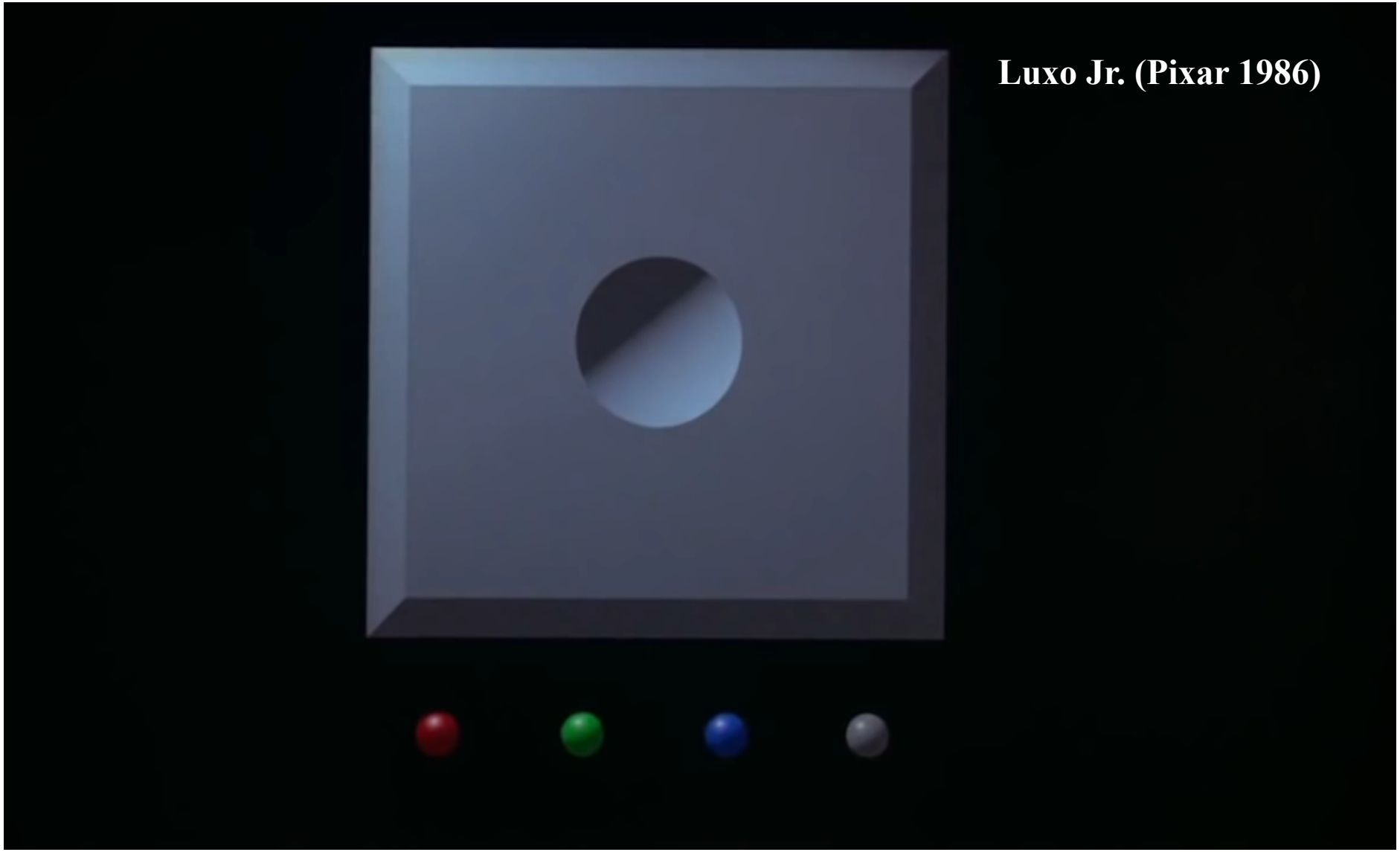
Instance Transformation

- Instances can be shared across space or time

- Write a function that renders the object in “standard” configuration
- Apply transformations to different instances
- Typical order: scaling, rotation, translation

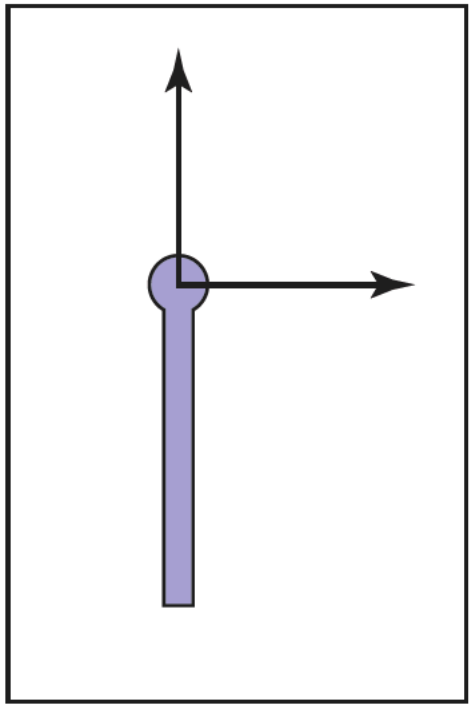


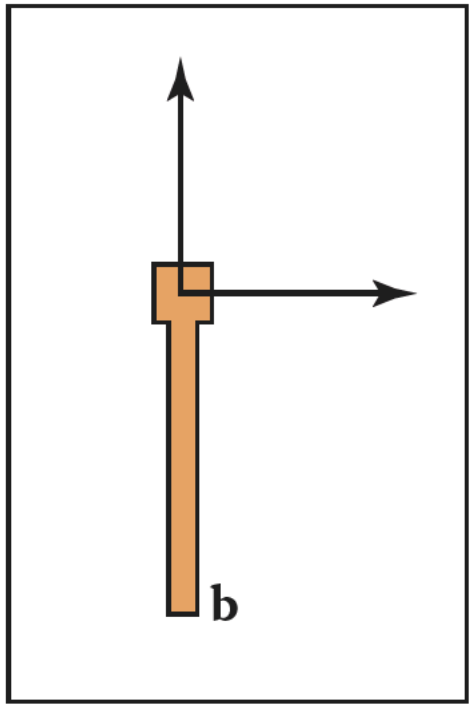
Animation: modeling motion

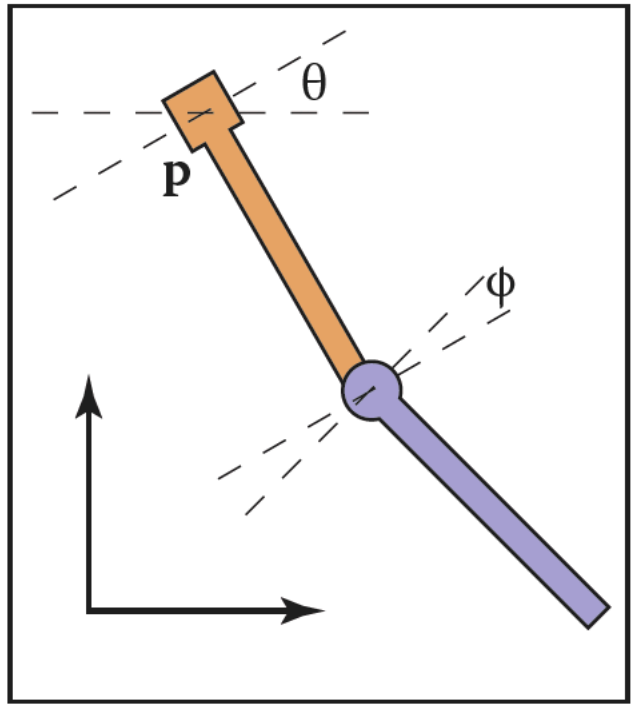


Pixar Shorts Collection Luxo Jr 1986

1st example







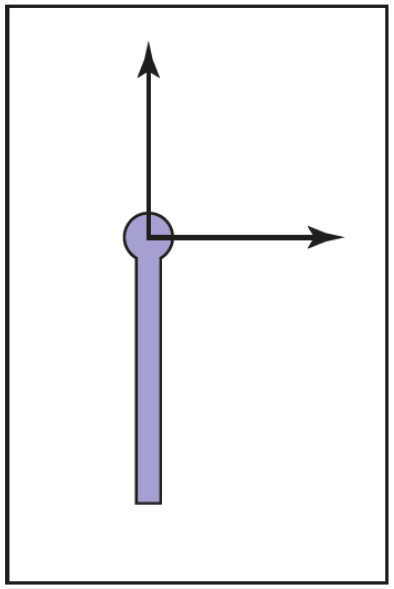
$$\mathbf{M}_1 = \text{rotate}(\theta)$$

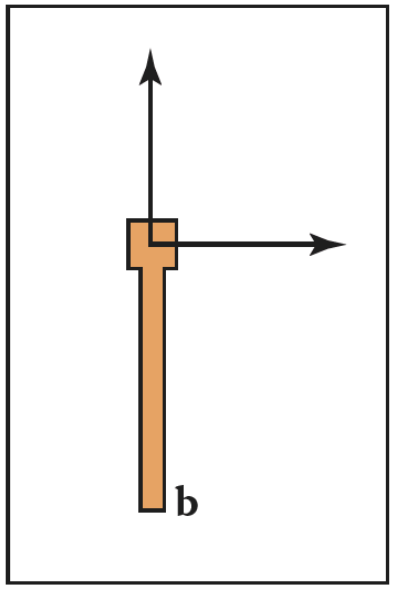
$$\mathbf{M}_2 = \text{translate}(\mathbf{p})$$

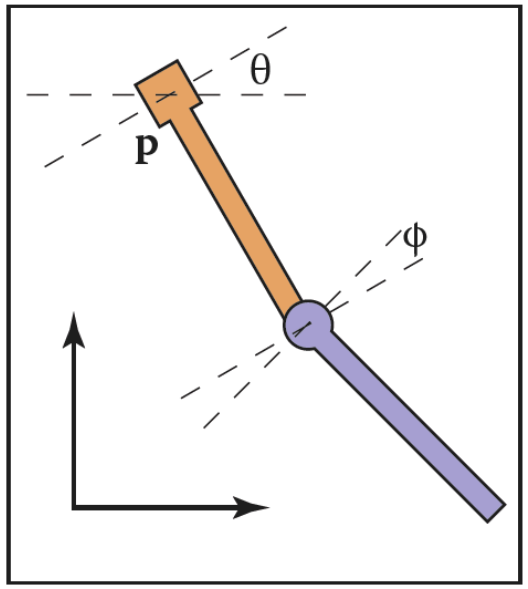
$$\mathbf{M}_3 = \mathbf{M}_2\mathbf{M}_1$$

Apply $\mathbf{M}_3$ to all points in upper pendulum

1st example







$$\mathbf{M}_a = \text{rotate}(\phi)$$
$$\mathbf{M}_b = \text{translate}(\mathbf{b})$$
$$\mathbf{M}_c = \mathbf{M}_b\mathbf{M}_a$$

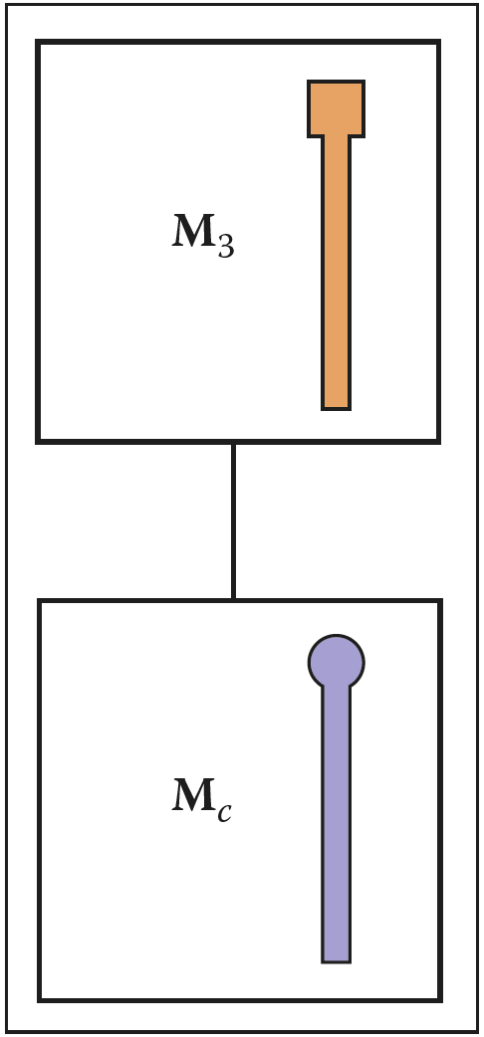
$$\mathbf{M}_1 = \text{rotate}(\theta)$$
$$\mathbf{M}_2 = \text{translate}(\mathbf{p})$$
$$\mathbf{M}_3 = \mathbf{M}_2\mathbf{M}_1$$

Apply $\mathbf{M}_3$ to all points in upper pendulum

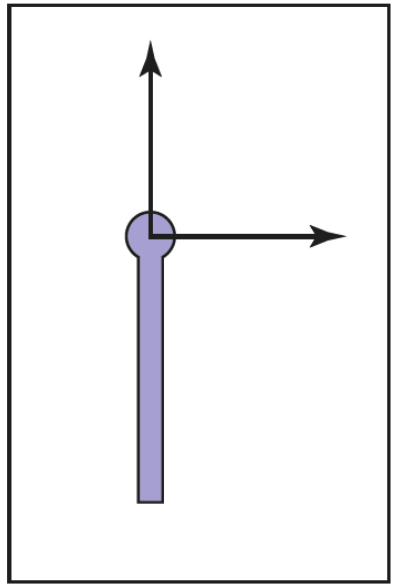
$$\mathbf{M}_d = \mathbf{M}_3\mathbf{M}_c$$

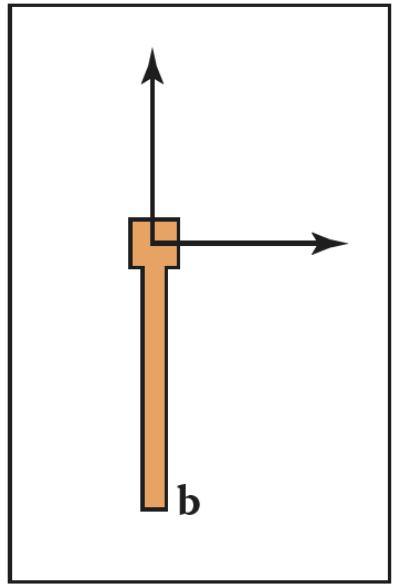
Apply $\mathbf{M}_d$ to all points in lower pendulum

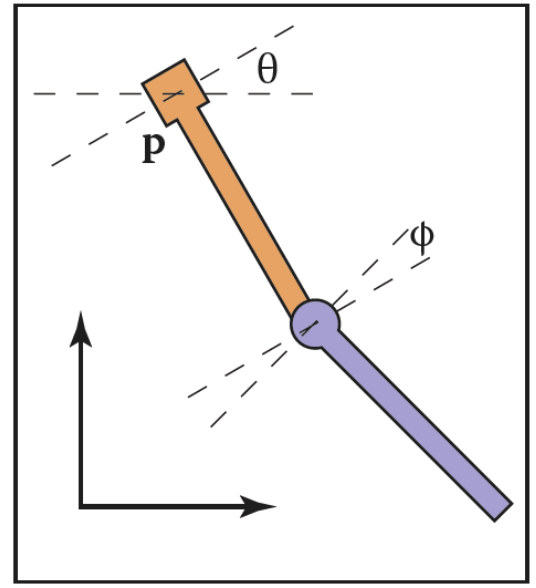
1st example



Scene graph







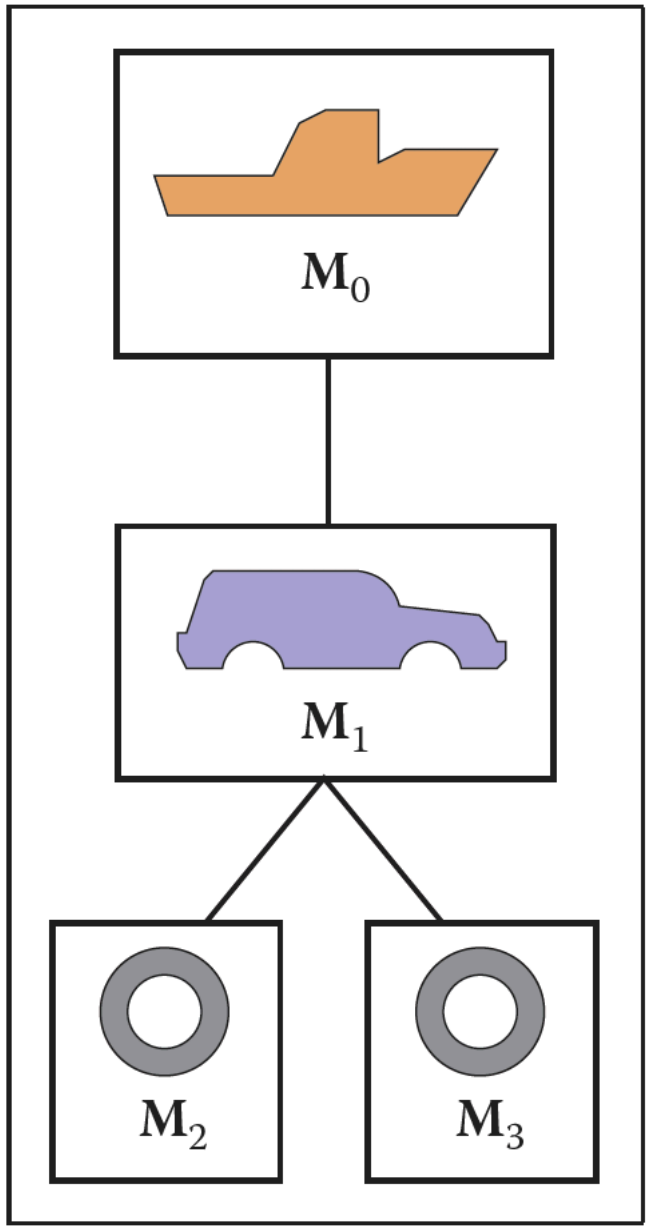
$$M_3 = M_2 M_1$$

Apply M_3 to all points in upper pendulum

$$M_d = M_3 M_c$$

Apply M_d to all points in lower pendulum

2nd example – How to draw a scene?

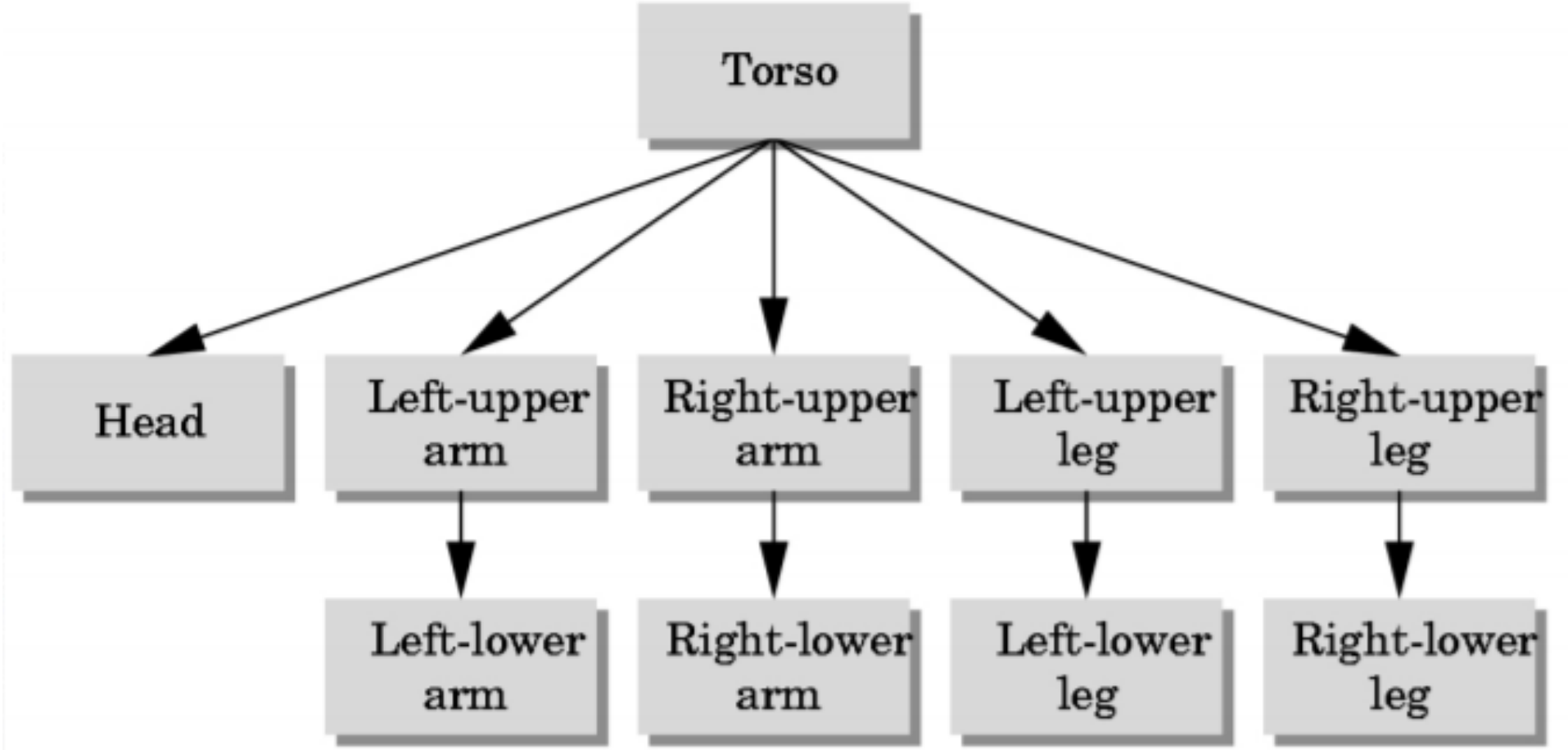


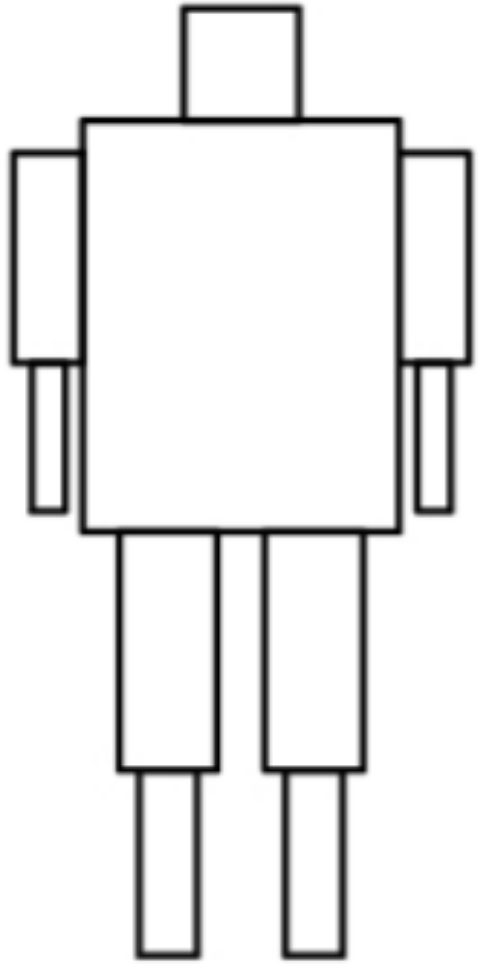
Interleave Drawing

- ferry transform using M_0;
- car body transform using $M_0 M_1$;
- left wheel transform using $M_0 M_1 M_2$;
- left wheel transform using $M_0 M_1 M_3$.

More Complex Objects

- **Tree rather than linear structure**
- **Interleave** along each branch
- Use push and pop to save state





3rd example

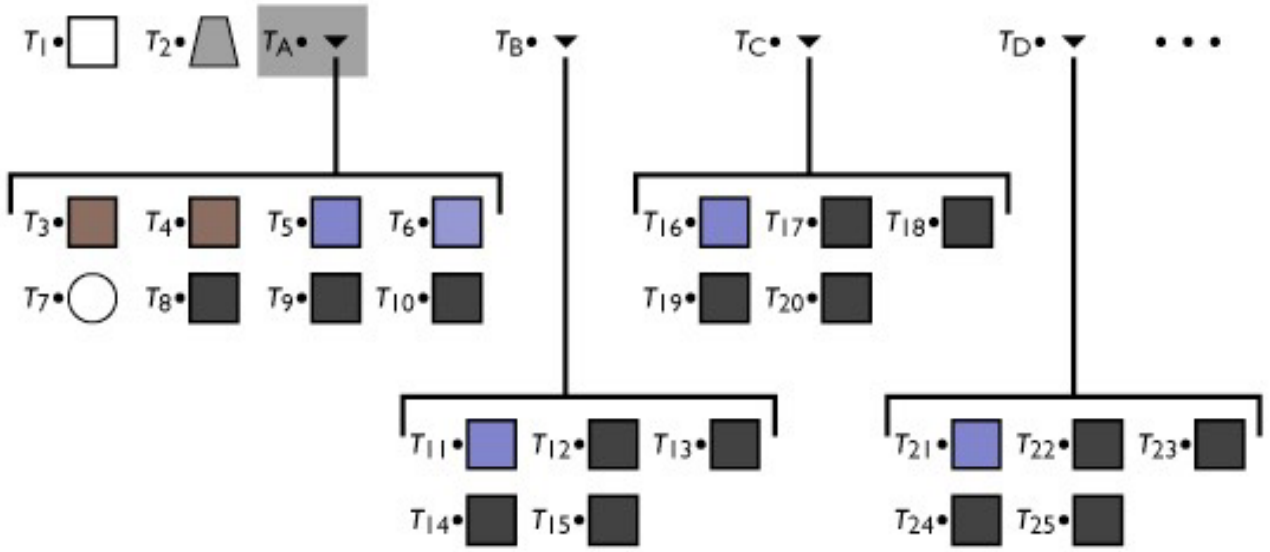
- **Can represent drawing with flat list**
  - but editing operations require updating many transforms

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

3rd 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

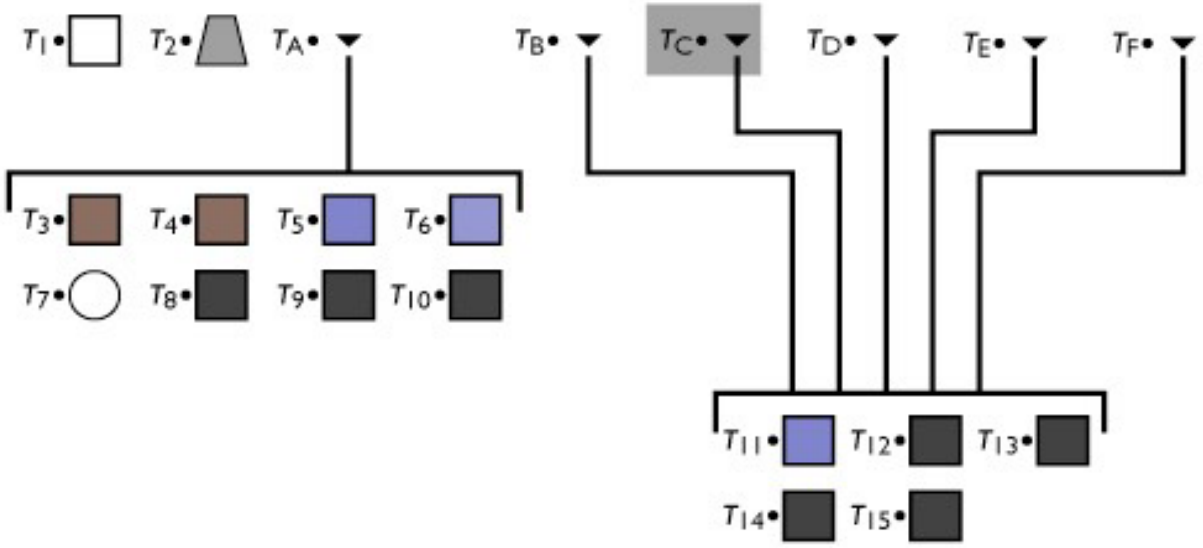


Variants of the Scene Graph - **Instancing**

- **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

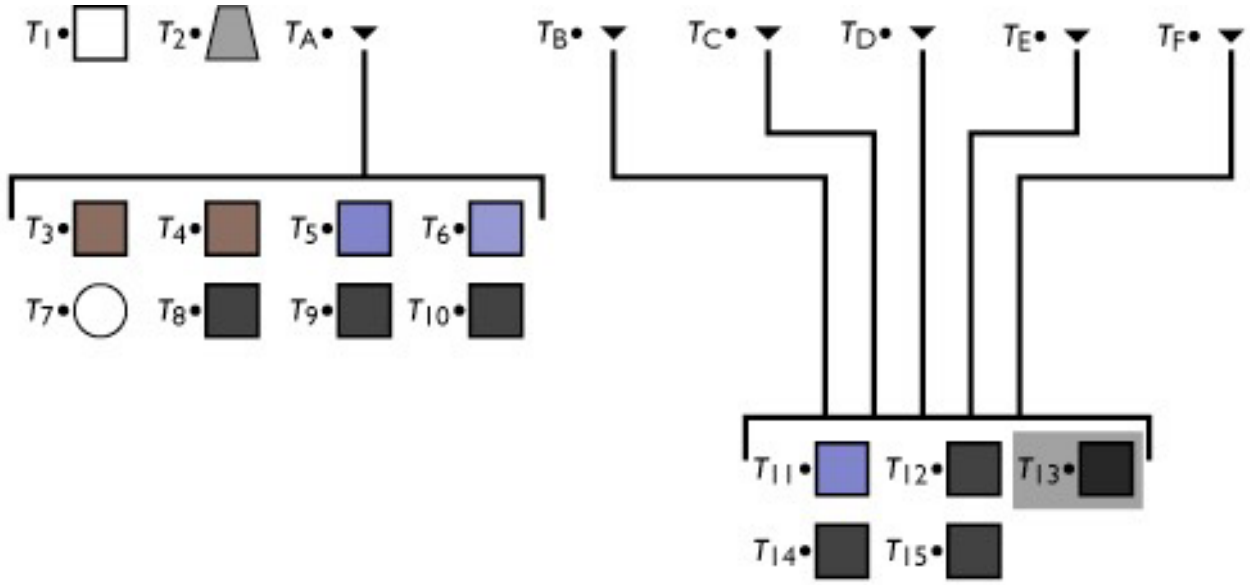
3rd example

- **Allow multiple references to nodes**
  - reflects more of drawing structure
  - allows editing of repeated parts in one operation



3rd example

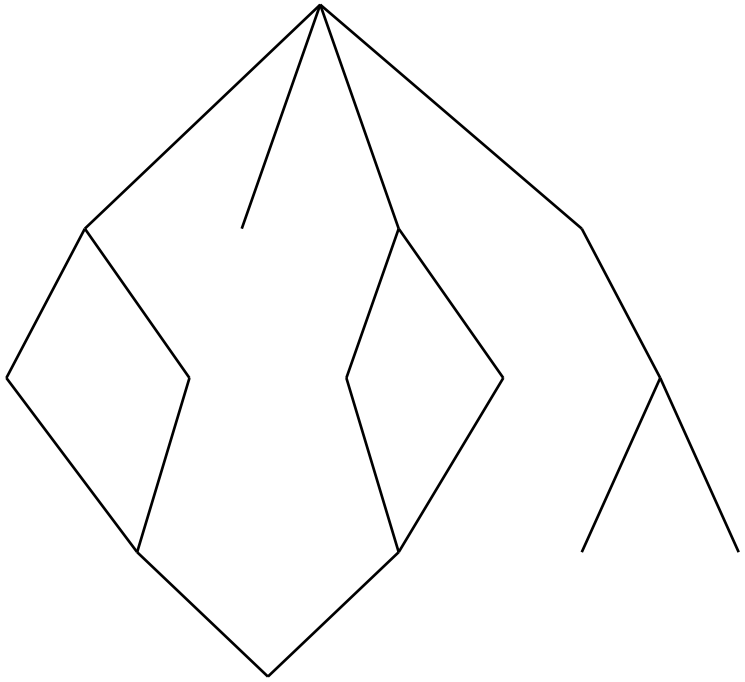
- **Allow multiple references to nodes**
  - reflects more of drawing structure
  - allows editing of repeated parts in one operation



Jan-Walter Schliep, Burak Kahraman, Timm Dapper | Laubwerk via PBRT gallery

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 DAG**
  - **d**irected **a**cyclic **g**raph
  - 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



Scene Graph & matrix stack

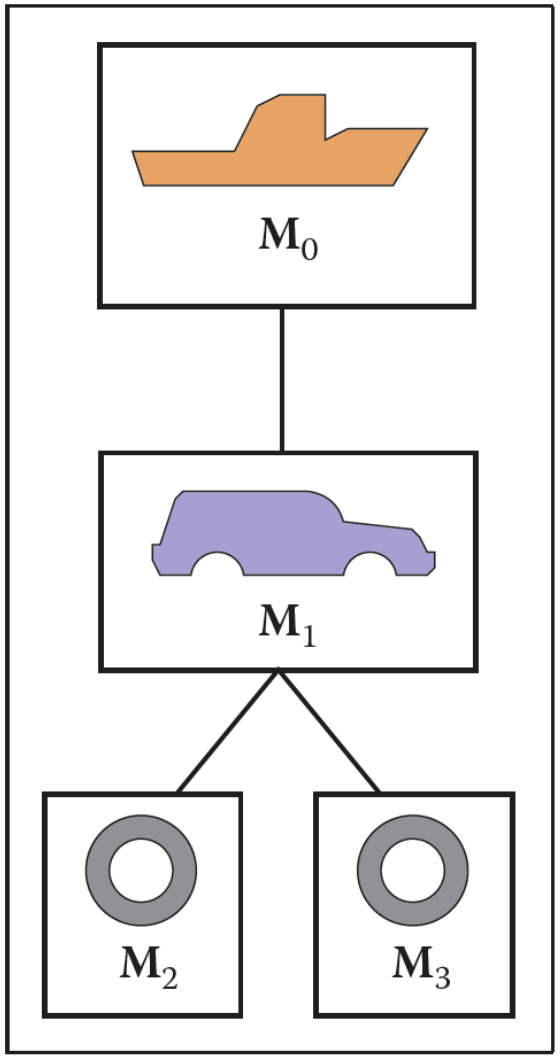
Active matrix
M = I

$\text{push}(\mathbf{M}_0)$
$\text{push}(\mathbf{M}_1)$
$\text{push}(\mathbf{M}_2)$

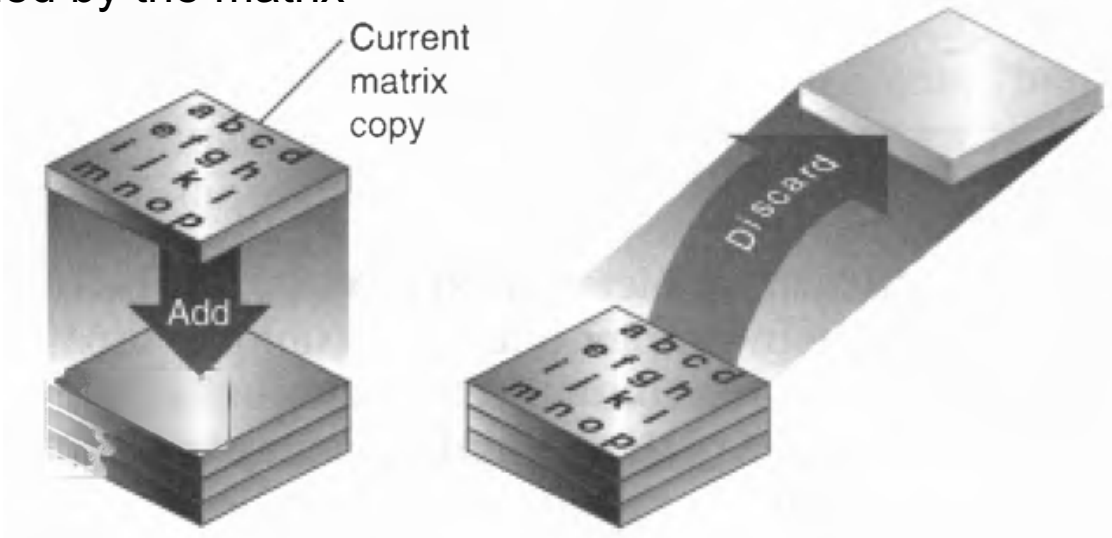
Active matrix
$\mathbf{M} = \mathbf{M}_0\mathbf{M}_1\mathbf{M}_2$

pop( )

Active matrix
$\mathbf{M} = \mathbf{M}_0\mathbf{M}_1$



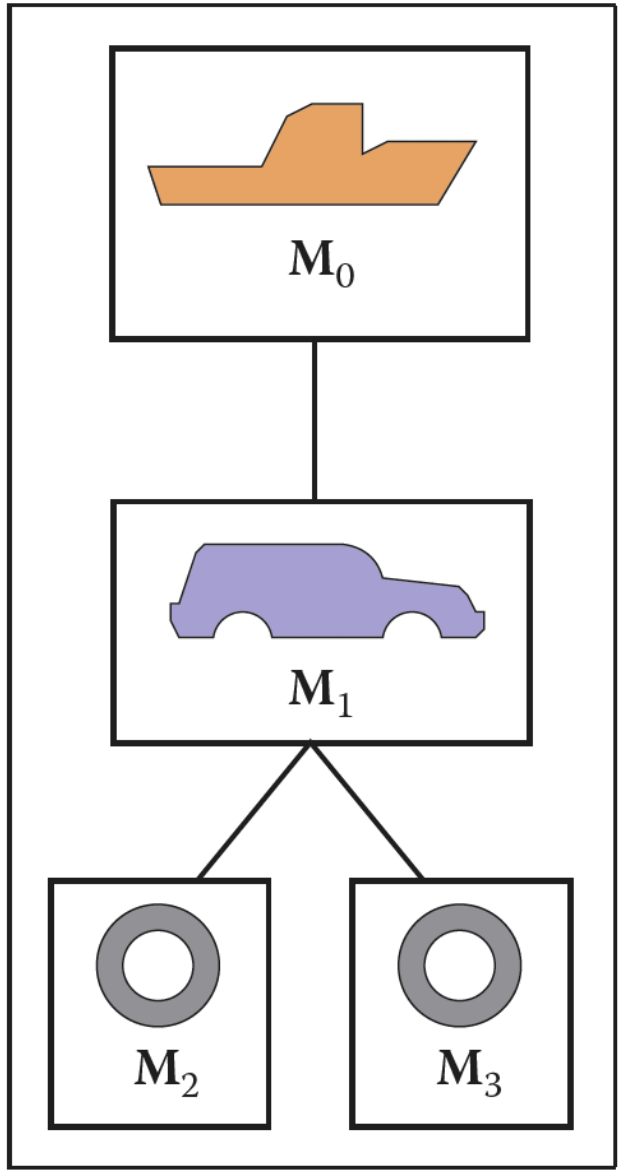
The current matrix is postmultiplied by the matrix



A recursive traversal of a scene graph

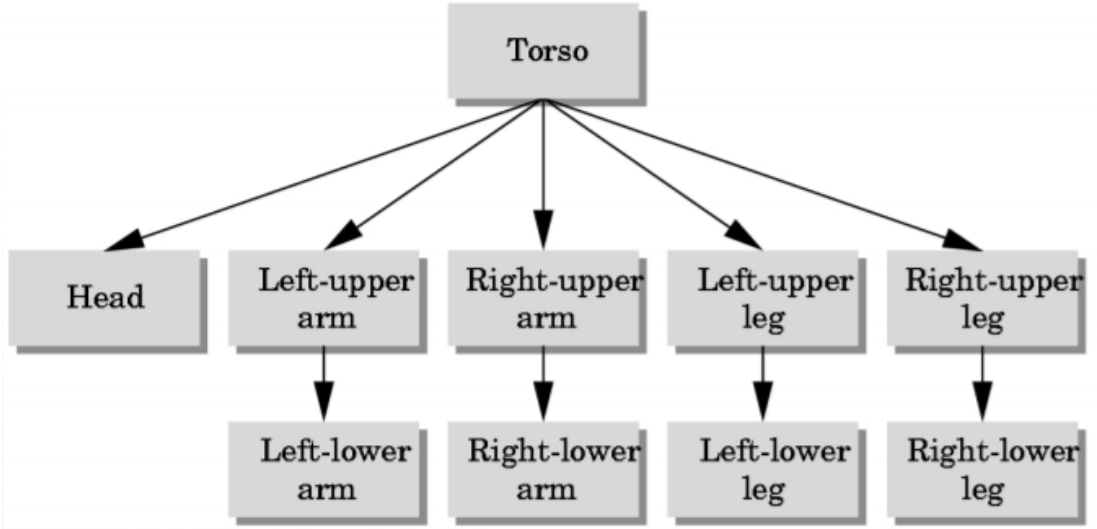
function traverse(node)
push($\mathbf{M}_{\text{local}}$)
draw object using composite matrix from stack
traverse(left child)
traverse(right child)
pop()

- ferry transform using M_0;
- car body transform using $M_0 M_1$;
- left wheel transform using $M_0 M_1 M_2$;
- left wheel transform using $M_0 M_1 M_3$.



Hierarchical Tree Traversal

- Order not necessarily fixed
- Example:



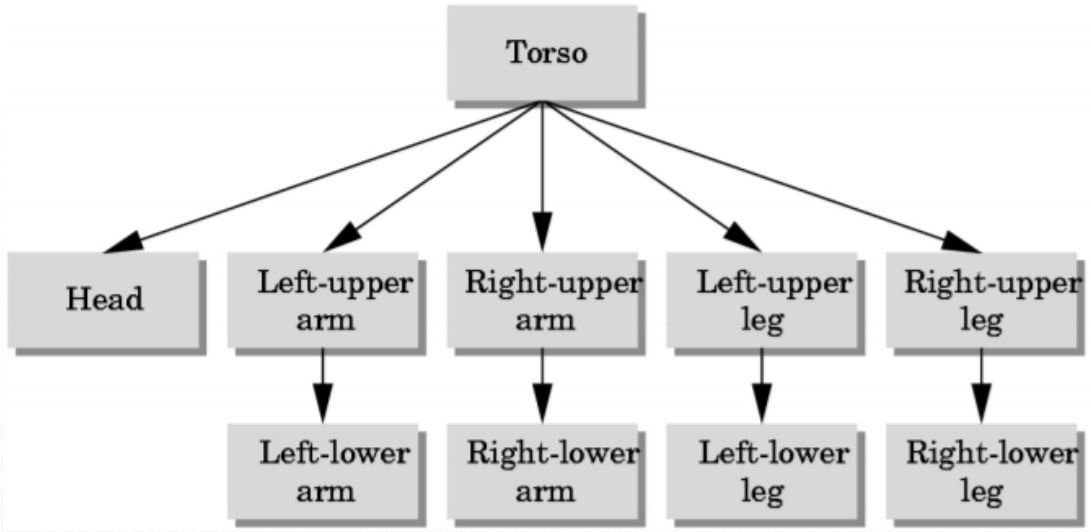

```
void drawFigure()
{
  glPushMatrix();  /* save */
  drawTorso();
  glTranslatef(...); /* move head */
  glRotatef(...);     /* rotate head */
  drawHead();
  glPopMatrix();   /* restore */
```

```
  glPushMatrix();
  glTranslatef(...);
  glRotatef(...);
  drawLeftUpperArm();
  glTranslatef(...)
  glRotatef(...)
  drawLeftLowerArm();
  glPopMatrix();
  ... }
```

Using Tree Data Structures

- Can make tree form explicit in data structure

```
typedef struct treenode
{
  GLfloat m[16];
  void (*f) ( );
  struct treenode *sibling;
  struct treenode *child;
} treenode;
```



Initializing Tree Data Structure

- Initializing transformation matrix for node

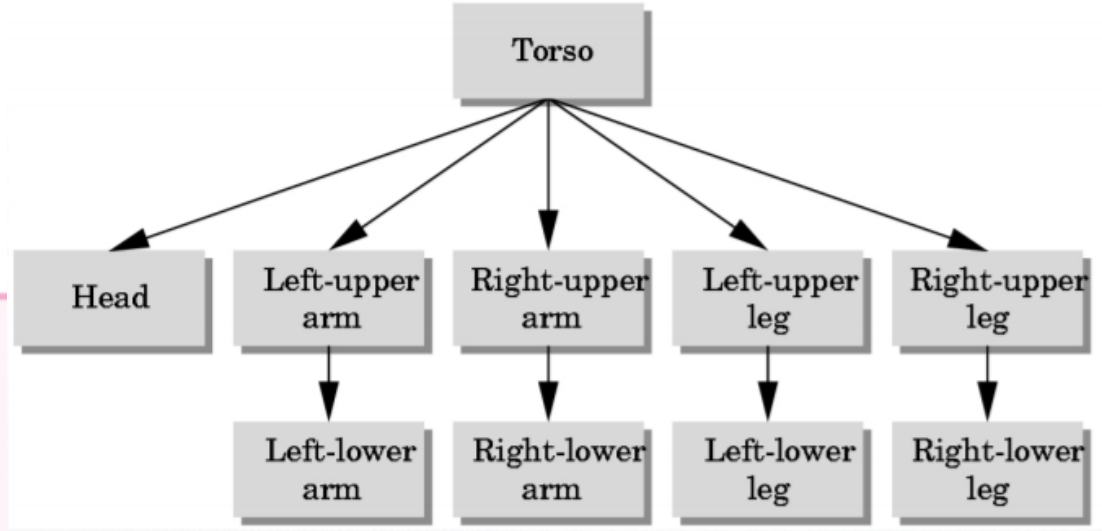
```
treenode torso, head, ...;
/* in init function */
glLoadIdentity();
glRotatef(...);
glGetFloatv(GL_MODELVIEW_MATRIX, torso.m);
```

- Initializing pointers

```
torso.f = drawTorso;
torso.sibling = NULL;
torso.child = &head;
```

Generic Traversal

- Recursive definition




```
void traverse (treenode *root)
{
  if (root == NULL) return;
  glPushMatrix();
  glMultMatrixf(root->m);
  root->f();
  if (root->child != NULL) traverse(root->child);
  glPopMatrix();
  if (root->sibling != NULL) traverse(root->sibling);
}
```

Implementing a hierarchy

- **Object-oriented language is convenient**
  - 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

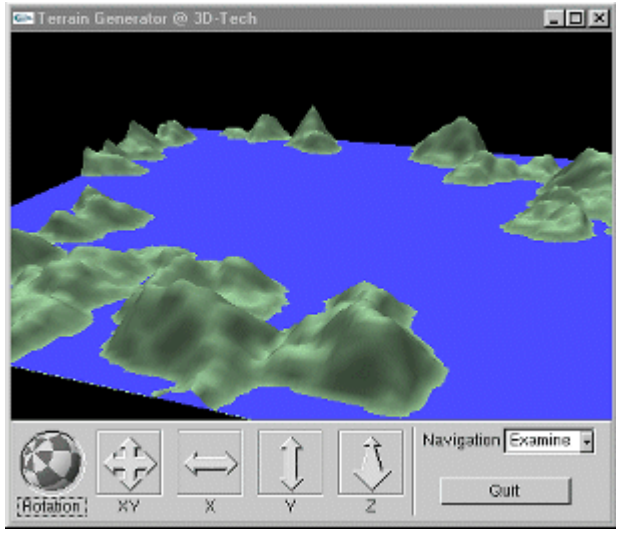
- **Editing a transformation**
  – good to present usable UI
- **Getting transform of object in canonical (world) frame**
  – traverse path from root to leaf
- **Grouping and ungrouping**
  – can do these operations without moving anything
  – group: insert identity node
  – ungroup: remove node, push transform to children
- **Reparenting**
  – move node from one parent to another
  – can do without altering position

Scene Graph variations

- **Where transforms go**
  - in every node
  - on edges
  - in group nodes only
  - in special Transform nodes
- **Tree vs. DAG**
- **Nodes for cameras and lights**
- **Nodes that set attributes**
  - e.g. "make everything in my subtree green"

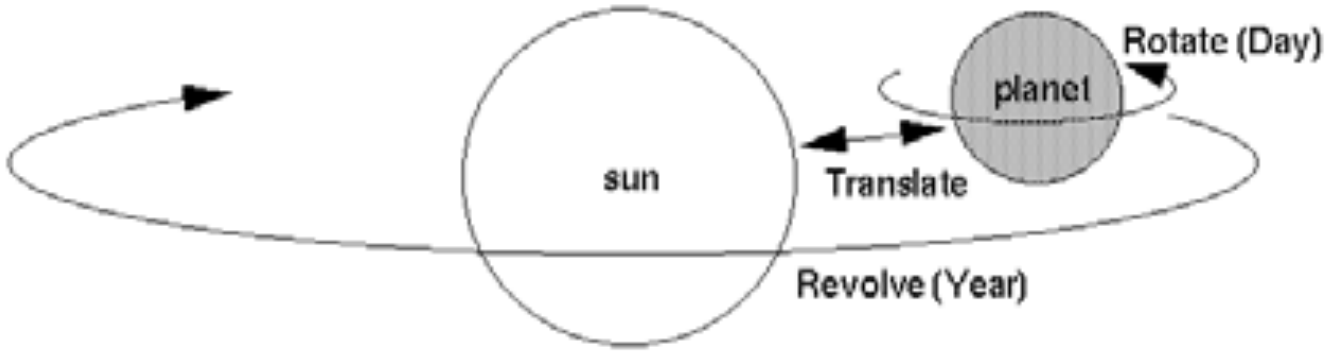
OpenGL Terrain Generator

- An example of OpenGL terrain generator developed by António Ramires Fernandes can be found in: http://www.lighthouse3d.com/opengl/appstools/tg/



- Terrain generation from an image, computing normals and simulating both directional and positional lights

Assignment 1： Building the solar system



- You will need to write from scratch a complete OpenGL programme that renders a Sun with an orbiting planet and a moon orbiting the planet

Assignment Basic Implementation

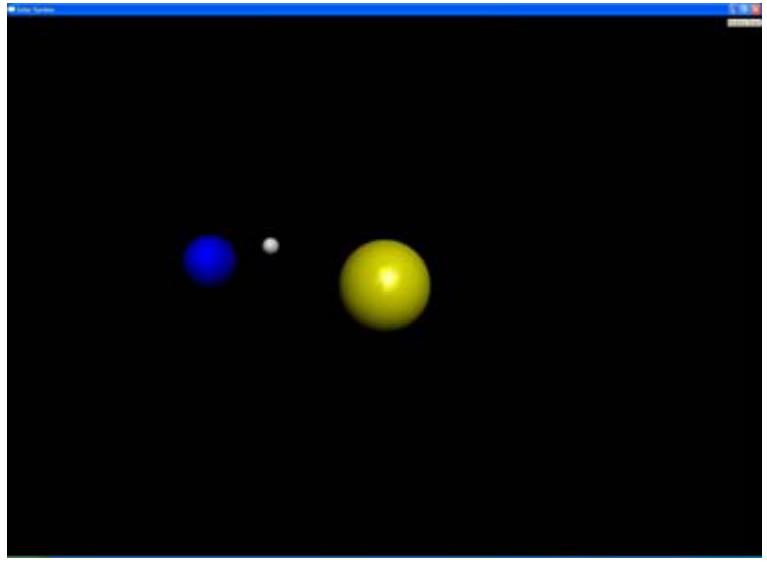
The basic implementation includes the following:

- Add a sphere representing the sun planet
- Make the sun planet to rotate around itself
- Add another sphere representing the earth
- Make the earth planet to rotate around itself
- Make the earth planet to rotate around sun
- Add another sphere representing the moon
- Make the moon planet to rotate around itself
- Make the moon planet to rotate around the earth
- Control the camera position using the keyboard
- Control the camera position using widget menus
- Add a light source
- Add shading to the planets
- Add material properties to the planets (you have to check this out
- yourselves)

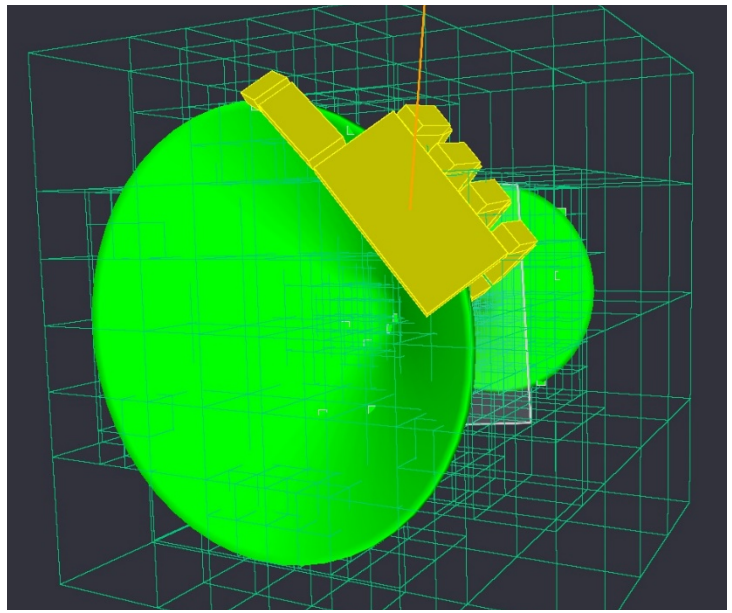
Assignment Advanced Implementation

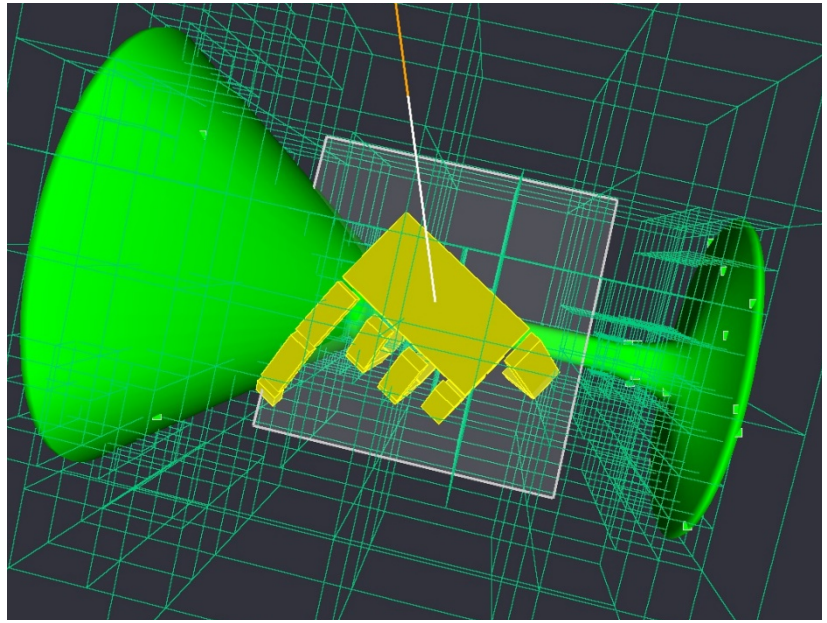
Recommended Implementation

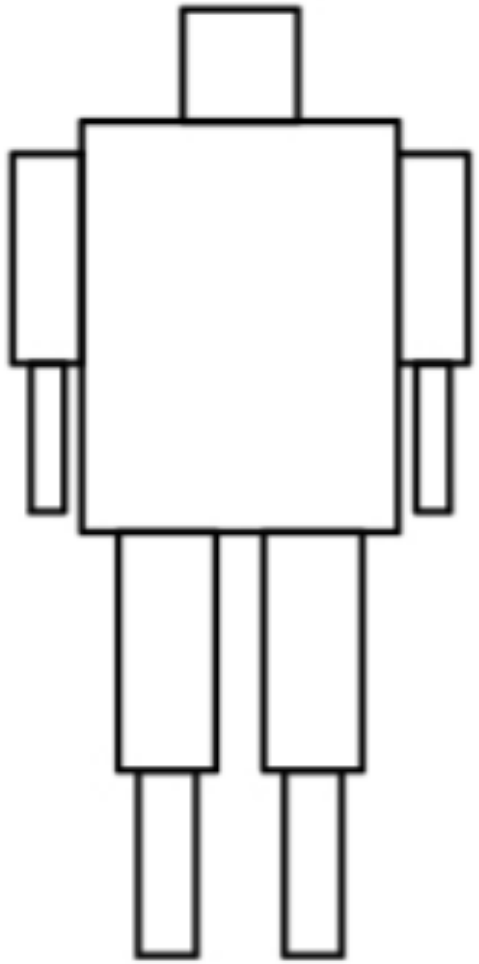
- Add more planets, e.g. if you are quick enough you could create the complete solar system
- Add more light sources (OpenGL supports up to 8 lights)
- Have planets counter rotating
- Add more moons to planets
- Add stars to the planetary system
- Add spaceships



Assignment 2: Building a robot arm or a robot







Thanks