Final Exam

Time: 12:30pm – 3:00 pm, Friday, December 9

Closed book and closed notes

Note:

- closed-book and closed-note
- one letter-size cheat sheet, you can use both sides
- coverage: all materials covered discussed in the class

Quiz 2

Take-home

5pm Wednesday, Nov. 30 – 2am Thursday, Dec. 1

Topics

Hierarchical modeling

Procedural methods

Generalizations

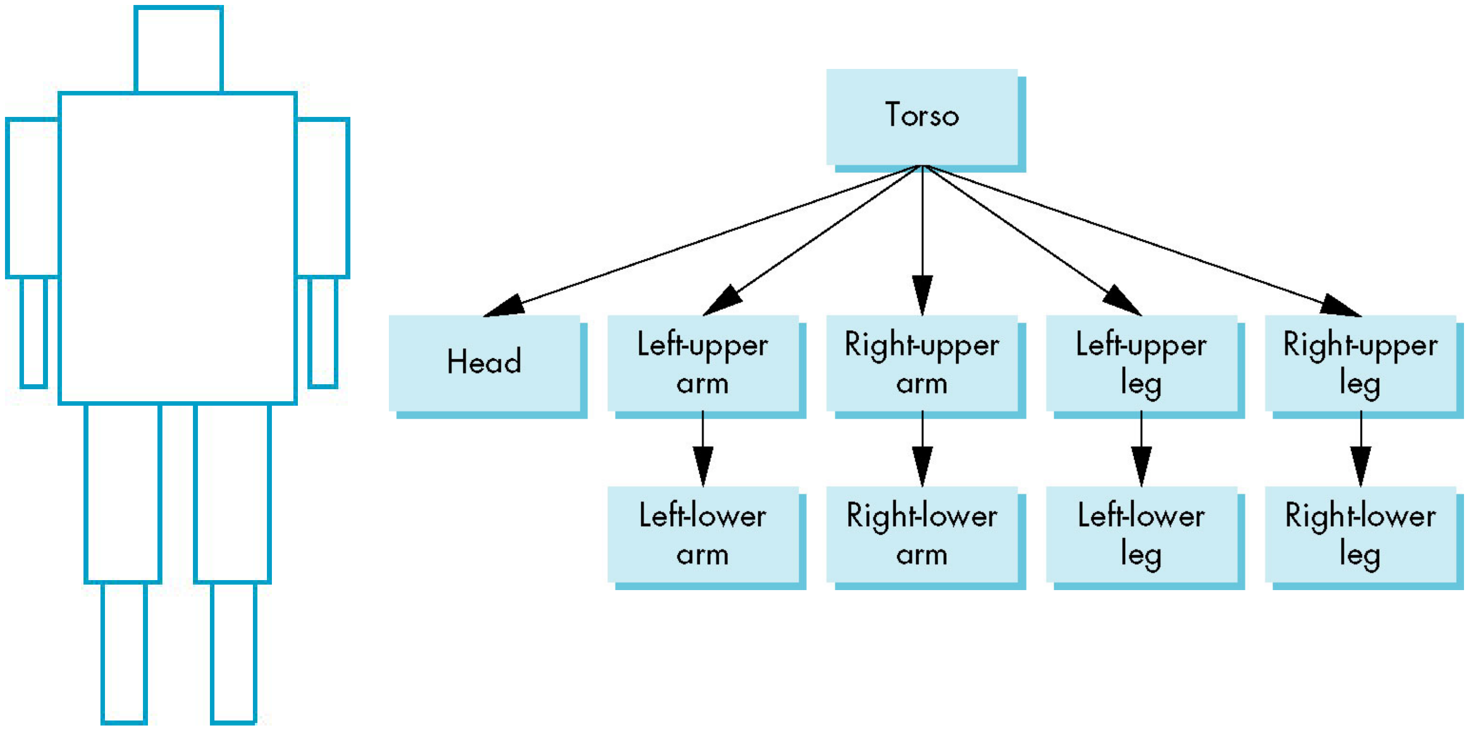
Need to deal with multiple children

- How do we represent a more general tree?
- How do we traverse such a data structure?

Animation

- How to use dynamically?
- Can we create and delete nodes during execution?

Humanoid Figure



Building the Model

Can build a simple implementation using quadrics:

- ellipsoids and cylinders

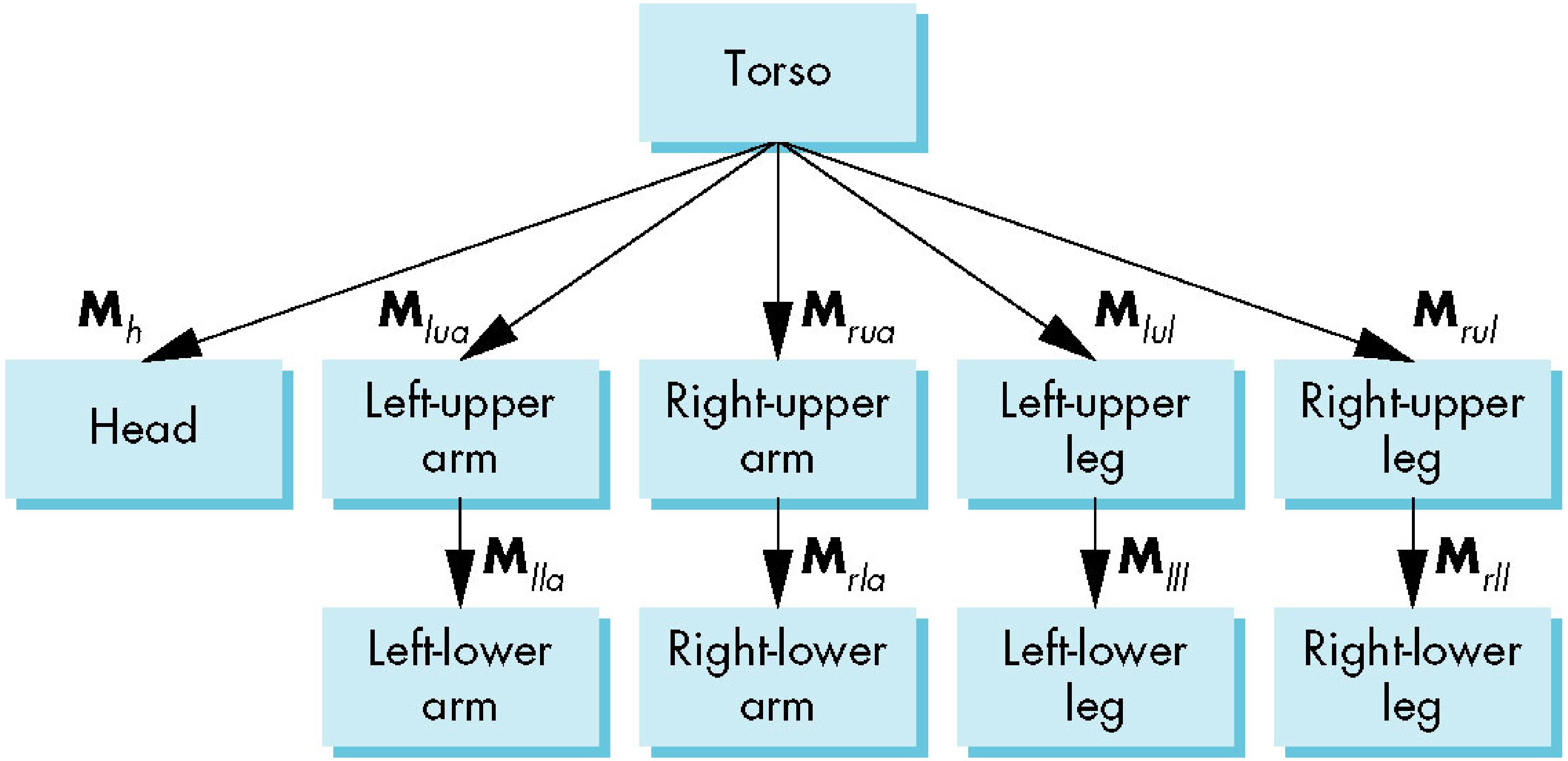
Access parts through functions drawing individual parts in their own frames

- `torso()`
- `left_upper_arm()`

Matrices describe position of node with respect to its parent

- $\mathbf{M}_{lla}$ positions left lower arm with respect to left upper arm

Tree with Matrices



Display and Traversal

The position of the figure is determined by 11 joint angles (two for the head and one for each other part)

Display of the tree requires a *graph traversal*

- Visit each node once
- Display function at each node that describes the part associated with the node, applying the correct transformation matrix for position and orientation

Transformation Matrices

There are 10 relevant matrices

- $\mathbf{M}$ positions and orients entire figure through the torso which is the root node
- $\mathbf{M}_{\mathrm{h}}$ positions head with respect to torso
- $\mathbf{M}_{\mathrm{lua}}$, $\mathbf{M}_{\mathrm{rua}}$, $\mathbf{M}_{\mathrm{lul}}$, $\mathbf{M}_{\mathrm{rul}}$ position arms and legs with respect to torso
- $\mathbf{M}_{\mathrm{lla}}$, $\mathbf{M}_{\mathrm{rla}}$, $\mathbf{M}_{\mathrm{lll}}$, $\mathbf{M}_{\mathrm{rll}}$ position lower parts of limbs with respect to corresponding upper limbs

Stack-based Traversal

Set model-view matrix to $\mathbf{M}$ 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 $\mathbf{M}$ and other matrices as we traverse the tree

Stack-based Traversal

```
class MatrixStack {
    int    _index;    int    _size;    mat4*  _matrices;
   public:
    MatrixStack( int numMatrices = 32 ):_index(0), _size(numMatrices)
        { _matrices = new mat4[numMatrices]; }
    ~MatrixStack(){ delete[]_matrices; }
    mat4& push( const mat4& m ) {
        assert( _index + 1 < _size );        _matrices[_index++] = m;
    }
    mat4& pop( void ) {
        assert( _index - 1 >= 0 );        _index--;
        return _matrices[_index];
    }
};
```

Notes

The position of figure is determined by 11 joint angles stored in `theta[11]`

Animate by changing the angles and redisplaying

We form the required matrices using `Rotate` and `Translate`

- Because the matrix is formed using the model-view matrix, we may want to first push original model-view matrix on matrix stack

Traversal Code

```
mat4 model_view;
matrix_stack mvstack;
figure() {
 //save present model-view matrix
   mvstack.push(model_view);
   torso();
 //update model-view matrix for head
   model_view = model_view*Translate()*Rotate();
   head();
 //recover original model-view matrix
   model_view = mvstack.pop();
 //save it again
   mvstack.push(model_view);
```

Traversal Code

…

```
//update model-view matrix for left upper arm
   model_view = model_view*Translate()*Rotate();
   left_upper_arm();

//recover and save original model-view matrix again
   model_view = mvstack.pop();
   mvstack.push(model_view);

…//rest of code
}
```

Code for Individual Parts

```
void torso(){
mvstack.push(model_view);
instance = Translate(0.0, 0.5*TORSO_HEIGHT,0.0)
*Scale(TORSO_WIDTH, TORSO_HEIGHT, TORSO_WIDTH);
glUniformMatrix4fv(model_view_loc, 16, GL_TRUE, model_view*instance);
colorcube();
glDrawArrays(GL_TRIANGLES, 0, N);
model_view = mvstack.pop();
}
```

Model view matrix linked to the shader

Note: need a push at the beginning and a pop at the end to isolate this function and protect the other parts

Code for Individual Parts

Functions for other parts can be defined similarly as for Torso.

Appendix A.9 provides the full program for the figure with tree traversal

Analysis

The code describes a particular tree and a particular traversal strategy

- Can we develop a more general approach?

Note that the sample code does not include state changes, such as changes to colors

- May also want to use a **`PushAttrib`** and **`PopAttrib`** to protect against unexpected state changes affecting later parts of the code

General Tree Data Structure

The code describes a particular tree and a particular traversal strategy

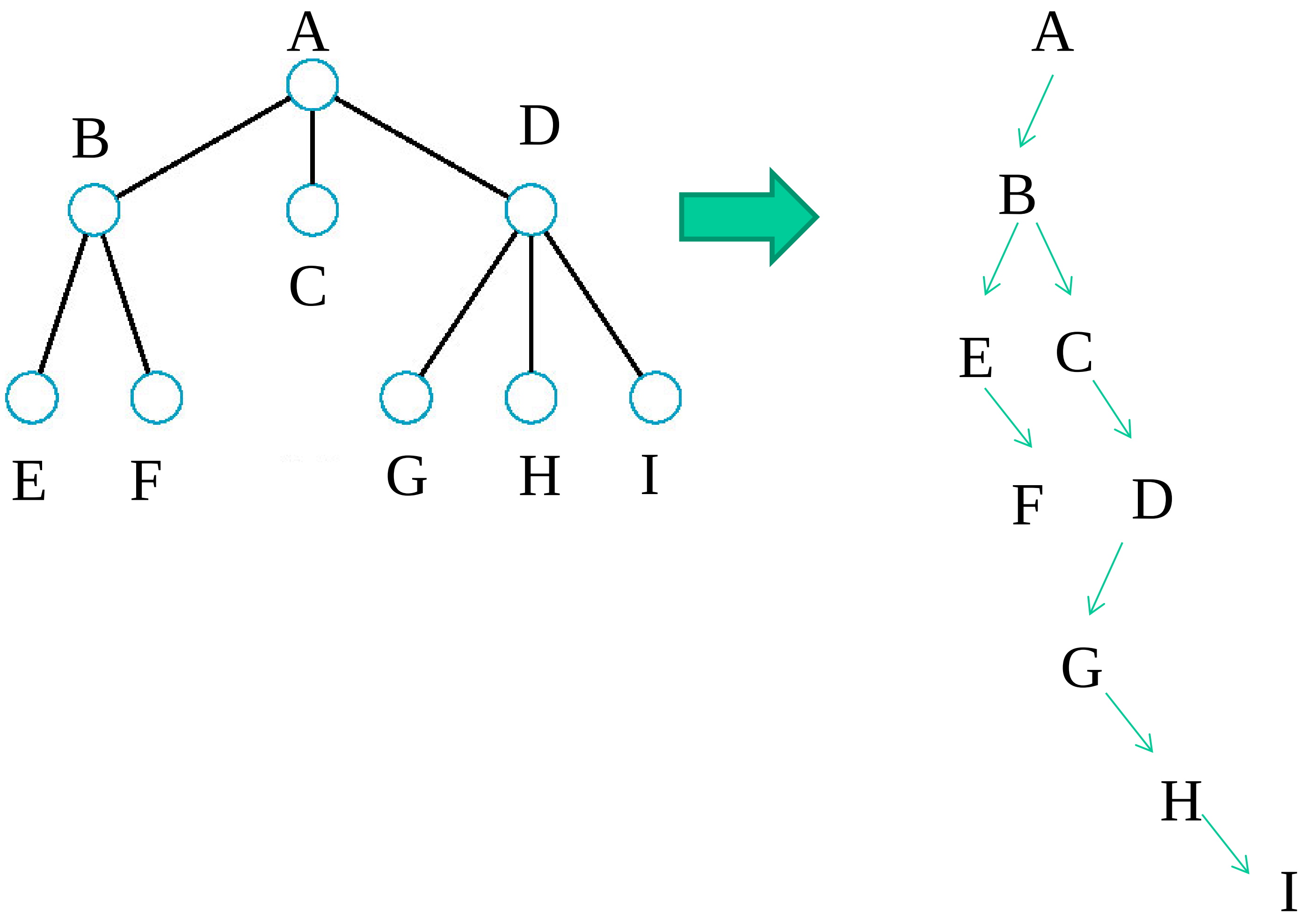
- Can we develop a more general approach?

Need a data structure to represent tree and an algorithm to traverse the tree

We will use a *left-child right sibling* binary tree

- Uses linked lists
- Each node in data structure has two pointers
- Left: linked list of children
- Right: sibling

Left-Child Right-Sibling Binary Tree



Tree node Structure

At each node we need to store

- Pointer to sibling
- Pointer to child
- Pointer to a function that draws the object represented by the node
- Homogeneous coordinate matrix to multiply on the right of the current model-view matrix
  - Represents changes going from parent to node
  - In OpenGL this matrix is a 1D array storing matrix by columns

Definition of Treenode

```
typedef struct treenode
{
    mat4 m;        → Transformation matrix
    void (*f)();   → Drawing function
    struct treenode *sibling;
    struct treenode *child;
} treenode;
```

Example of Torso and Head Nodes

```
treenode torso_node, head_node, lua_node, … ;

torso_node.m = RotateY(theta[0]);

torso_node.f = torso;

torso_node.sibling = NULL;

torso_node.child =  &head_node;

head_node.m = translate(0.0,TORSO_HEIGHT+0.5*HEAD_HEIGHT,
0.0)*RotateX(theta[1])*RotateY(theta[2]);

head_node.f = head;

head_node.sibling = &lua_node;

head_node.child = NULL;
```

Preorder Traversal

```
void traverse(treenode* root)
{
   if(root==NULL) return;
   mvstack.push(model_view);
   model_view = model_view*root->m;
   root->f();
   if(root->child!=NULL) traverse(root->child);
   model_view = mvstack.pop();
   if(root->sibling!=NULL) traverse(root->sibling);
}
```

Notes

We must save model-view matrix before multiplying it by node matrix

- Updated matrix applies to children of node but not to siblings which contain their own matrices

The traversal program applies to any left-child right-sibling tree

- The particular tree is encoded in the definition of the individual nodes

The order of traversal matters because of possible state changes in the functions

Dynamic Trees

If we use pointers, the structure can be dynamic

```
typedef treenode *tree_ptr;

tree_ptr torso_ptr;

torso_ptr = malloc(sizeof(treenode));
```

Definition of nodes and traversal are essentially the same as before but we can add and delete nodes during execution

Animation

Animation is realized by changing the model view matrix of each part as a function of time

Reading Assignments

Chapter 8.6 – 8.11 in Angel & Shreiner

Procedural Methods

How can we model

- Natural phenomena
  - Clouds
  - Terrain
  - Plants
- Crowd Scenes
- Real physical processes

Procedural methods:

Describe objects in an algorithmic way and generate polygons when needed during rendering

Procedural Approaches

- **Physically-based models and particle system**
  - Describing dynamic behaviors
    - Fireworks
    - Flocking behavior of birds
    - Wave action
- **Language-based models**
  - Describing trees or terrain
  - Representing relationships
- **Fractal geometry**

Newtonian Particle

Particle system is a set of particles

Each particle is an ideal point mass

- Gives the positions of particles
- At each location, we can show an object

Six degrees of freedom

- Position
- Velocity

Each particle obeys Newtons' law

$$\mathbf{f} = m\mathbf{a}$$

Vectors in 3D

Particle Equations

The state of the i^th^ particle is defined by its position $\mathbf{p}_i = \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix}$

Then, we have 6 ordinary differential equations:

Velocity $\mathbf{v}_i = \frac{d\mathbf{p}_i}{dt} = \begin{bmatrix} \frac{dx_i}{dt} \\ \frac{dy_i}{dt} \\ \frac{dz_i}{dt} \end{bmatrix}$

Acceleration $\mathbf{a}_i = \frac{d\mathbf{v}_i}{dt} = \frac{1}{m_i}\mathbf{f}_i(t)$

The question is how we get the force vector

Solution of Particle Systems

//For a system with *n* particles

```
float time, delta, state[6n], force[3n];
state = initial_state();
for(time = t0; time<final_time, time+=delta) {
        //compute forces
        force =  force_function(state, time);
        // solve the differential equation
        state = ode(force, state, time, delta);
        render(state, time)
}
```

Force Vector

Depending on how particles interact with each other

- Independent Particles O(n)
  - Gravity
  - Drag
- Coupled Particles O(n)
  - Spring-Mass Systems
  - Meshes
- Coupled Particles O(n^2)
  - Attractive and repulsive forces

Simple Forces

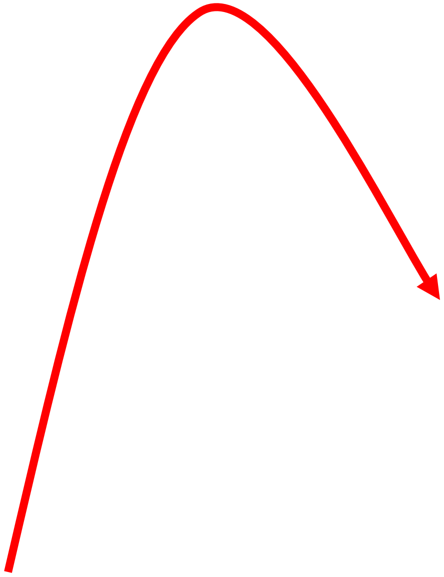
Consider force on a particle i

$$\mathbf{f}_i = \mathbf{f}_i(\mathbf{p}_i, \mathbf{v}_i)$$

Gravity $\mathbf{f}_{gi} = m_i\, \mathbf{g}$

$$\mathbf{g}_i = (0, -g, 0)$$

Drag $\mathbf{f}_{di} = \mu_i\, \mathbf{f}_{normi}$



$\mathbf{p}_i(t_0)$, $\mathbf{v}_i(t_0)$

Spring Forces

Assume each particle has unit mass and is connected to its neighbor(s) by a spring

Keep particles together

Hooke's law: force proportional to distance ($d = ||p - q||$) between the points