
Computer Graphics

7 - Lab - Hierarchical Modeling, Mesh

Yoonsang Lee
Hanyang University

Spring 2026

Outline

- Hierarchical Model Example
- Cube (Separate Triangles – `glDrawArrays`)
- Cube (Indexed Triangle Set – `glDrawElements`)

Hierarchical Model Example

Rendering Hierarchical Models - Method 1 (Precomputed, Stored per Node)

- **Method 1:** Store global transform G_i in each node and use it directly for rendering.
 - Global transforms are **computed each frame** and **stored per node before rendering**

Node i	Global Transform $G_i = \dots$
Hips	M_{hips}
Spine	$M_{hips} M_{spine}$
Head	$M_{hips} M_{spine} M_{head}$
RightArm	$M_{hips} M_{spine} M_{ra}$
RightForeArm	$M_{hips} M_{spine} M_{ra} M_{rfa}$
RightHand	$M_{hips} M_{spine} M_{ra} M_{rfa} M_{rh}$
LeftArm	$M_{hips} M_{spine} M_{la}$
...	

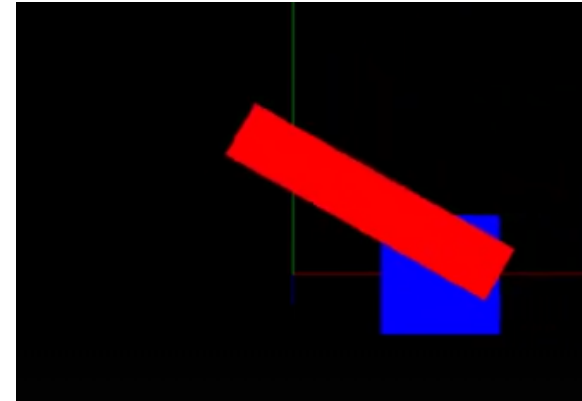
Rendering Hierarchical Models – Method 2

(On-the-fly, Depth-first Traversal with Stack)

- **Method 2:** Compute global transform G_i during hierarchical traversal for rendering and use it for rendering.
 - Global transforms are **computed on the fly** during **depth-first traversal** using a **stack**
- G_i is obtained by accumulating transforms along the path
- Commonly used in legacy OpenGL (matrix stack)
- In modern OpenGL, transforms are typically computed on the CPU → Method 1 is more common
- See supplemental slides for step-by-step example

[Code] 1-hierarchical

- This example follows Method 1. It stores the global transform G_i in each node object.
- Translating "base" + rotating "arm"
- Three things to render, each moving differently:
 - World frame
 - Blue base (parent)
 - Red arm (child of base)



[Code] 1-hierarchical

- Two VAOs
 - Frame VAO
 - Box VAO (for base and arm)
- Two shader programs
 - Shader for **frame**: use vertex color specified as a vertex attribute
 - Shader for **box**: use vertex color passed as a uniform variable
- "Node" class that represents a node and builds a tree structure
 - Two Node instances: base, arm
- Each node instance has:
 - parent (parent node), children (child nodes)
 - transform (local transform)
 - global_transform
 - shape_transform
 - color

[Code] 1-hierarchical

Vertex shader for **frame**

```
#version 330 core

layout (location = 0) in vec3 vin_pos;
layout (location = 1) in vec3 vin_color;

out vec4 vout_color;

uniform mat4 MVP;

void main()
{
    // 3D points in homogeneous
    coordinates
    vec4 p3D_in_hcoord =
    vec4(vin_pos.xyz, 1.0);

    gl_Position = MVP * p3D_in_hcoord;

    vout_color = vec4(vin_color, 1.);
}
```

Vertex shader for **box**

```
#version 330 core

layout (location = 0) in vec3 vin_pos;

out vec4 vout_color;

uniform mat4 MVP;
uniform vec3 color;

void main()
{
    // 3D points in homogeneous
    coordinates
    vec4 p3D_in_hcoord =
    vec4(vin_pos.xyz, 1.0);

    gl_Position = MVP * p3D_in_hcoord;

    vout_color = vec4(color, 1.);
}
```

- Fragment shaders are the same for both frame and box.

[Code] 1-hierarchical

```
def prepare_vao_box():
    # prepare vertex data (in main
memory)
    # 6 vertices for 2 triangles
    vertices = glm.array(glm.float32,
        # position
        -1 , 1 , 0 , # v0
        1 , -1 , 0 , # v2
        1 , 1 , 0 , # v1

        -1 , 1 , 0 , # v0
        -1 , -1 , 0 , # v3
        1 , -1 , 0 , # v2
    )
    ss ...

    # configure vertex positions
    glVertexAttribPointer(0, 3, GL_FLOAT,
GL_FALSE, 3 * glm.sizeof(glm.float32),
None)
    glEnableVertexAttribArray(0)

    return VAO
```

```
def prepare_vao_frame():
    # prepare vertex data (in main
memory)
    vertices = glm.array(glm.float32,
        # position # color
        0, 0, 0, 1, 0, 0, # x-axis start
        1, 0, 0, 1, 0, 0, # x-axis end
        0, 0, 0, 0, 1, 0, # y-axis start
        0, 1, 0, 0, 1, 0, # y-axis end
        0, 0, 0, 0, 0, 1, # z-axis start
        0, 0, 1, 0, 0, 1, # z-axis end
    )
    ...
    # configure vertex positions
    glVertexAttribPointer(0, 3, GL_FLOAT,
GL_FALSE, 6 * glm.sizeof(glm.float32),
None)
    glEnableVertexAttribArray(0)

    # configure vertex colors
    glVertexAttribPointer(1, 3, GL_FLOAT,
GL_FALSE, 6 * glm.sizeof(glm.float32),
ctypes.c_void_p(3*glm.sizeof(glm.float32))
    )
    glEnableVertexAttribArray(1)

    return VAO
```

[Code] 1-hierarchical

```
class Node:
    def __init__(self, parent, shape_transform, color):
        # hierarchy
        self.parent = parent
        self.children = []
        if parent is not None:
            parent.children.append(self)

        # transform
        self.transform = glm.mat4()
        self.global_transform = glm.mat4()

        # shape
        self.shape_transform = shape_transform
        self.color = color

    def set_transform(self, transform):
        self.transform = transform

    def update_tree_global_transform(self):
        if self.parent is not None:
            self.global_transform = self.parent.get_global_transform() * self.transform
        else:
            self.global_transform = self.transform
        for child in self.children:
            child.update_tree_global_transform()

    def get_global_transform(self):
        return self.global_transform

    def get_shape_transform(self):
        return self.shape_transform

    def get_color(self):
        return self.color
```

Global transformation of node i (G_i)

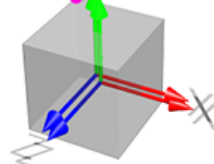
= product of **local transforms** along the path from root to node i

= parent's **global transform** * node i 's **local transform**

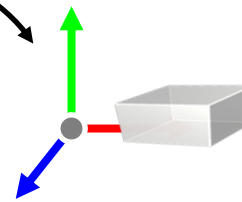
[Code] 1-hierarchical

```
def draw_frame(vao, MVP, loc_MVP):  
    glBindVertexArray(vao)  
    glUniformMatrix4fv(loc_MVP, 1, GL_FALSE, glm.value_ptr(MVP))  
    glDrawArrays(GL_LINES, 0, 6)  
  
def draw_node(vao, node, VP, loc_MVP, loc_color):  
    MVP = VP * node.get_global_transform() * node.get_shape_transform()  
    color = node.get_color()  
  
    glBindVertexArray(vao)  
    glUniformMatrix4fv(loc_MVP, 1, GL_FALSE, glm.value_ptr(MVP))  
    glUniform3f(loc_color, color.r, color.g, color.b)  
    glDrawArrays(GL_TRIANGLES, 0, 6)
```

$p_0 = [-0.5, 0.5, -0.5]$



S_i



$$p_i' = G_i S_i p_0$$

[Code] 1-hierarchical

```
def main():
    ...
    # load shaders
    shader_for_frame = load_shaders(g_vertex_shader_src_color_attribute,
g_fragment_shader_src)
    shader_for_box = load_shaders(g_vertex_shader_src_color_uniform,
g_fragment_shader_src)

    # get uniform locations
    loc_MVP_frame = glGetUniformLocation(shader_for_frame, 'MVP')
    loc_MVP_box = glGetUniformLocation(shader_for_box, 'MVP')
    loc_color_box = glGetUniformLocation(shader_for_box, 'color')

    # prepare vaos
    vao_box = prepare_vao_box()
    vao_frame = prepare_vao_frame()

    # create a hierarchical model - Node(parent, shape_transform, color)
    base = Node(None, glm.scale((.2,.2,0.)), glm.vec3(0,0,1))
    arm = Node(base, glm.translate((.5,0,.01)) * glm.scale((.5,.1,0.)),
glm.vec3(1,0,0))
```

[Code] 1-hierarchical

```
...
while not glfwWindowShouldClose(window):
    # projection matrix
    P = glm.ortho(-1,1,-1,1,-10,10)

    # view matrix
    V = glm.lookAt(...)

    # draw world frame
    glUseProgram(shader_for_frame)
    draw_frame(vao_frame, P*V*glm.mat4(), loc_MVP_frame)

    t = glfwGetTime()

    # set local transformations of each node
    base.set_transform(glm.translate((glm.sin(t),0,0)))
    arm.set_transform(glm.translate((.2, 0, 0)) * glm.rotate(t, (0,0,1)))

    # recursively update global transformations of all nodes
    base.update_tree_global_transform()

    # draw nodes
    glUseProgram(shader_for_box)
    draw_node(vao_box, base, P*V, loc_MVP_box, loc_color_box)
    draw_node(vao_box, arm, P*V, loc_MVP_box, loc_color_box)
```

Quiz 3

- Go to <https://www.slido.com/>
- Join #cg-ys
- Click "Polls"
- Submit your answer in the following format:
 - **Student ID: Your answer**
 - e.g. **2024123456: 4.0**
- Note that your quiz answer must be submitted **in the above format** to receive a quiz score!

Cube

(Separate Triangles – glDrawArrays)

Using glDrawArrays() (Separate Triangles)

- This is how we drew a cube in the previous lab using glDrawArrays().
 - Vertices are duplicated for each triangle (36 vertices total)

```
def prepare_vao_cube():
    # 36 vertices for 12 triangles
    vertices = glm.array(glm.float32,
        # position          color
        -0.5 ,  0.5 ,  0.5 ,  1, 1, 1, # v0
         0.5 , -0.5 ,  0.5 ,  1, 1, 1, # v2
         0.5 ,  0.5 ,  0.5 ,  1, 1, 1, # v1

        -0.5 ,  0.5 ,  0.5 ,  1, 1, 1, # v0
        -0.5 , -0.5 ,  0.5 ,  1, 1, 1, # v3
         0.5 , -0.5 ,  0.5 ,  1, 1, 1, # v2

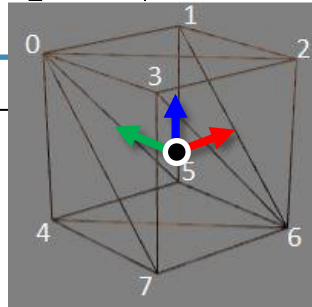
        ...
    )
    ...

def draw_cube(vao, MVP, loc_MVP):
    glBindVertexArray(vao)
    glUniformMatrix4fv(loc_MVP, 1, GL_FALSE, glm.value_ptr(MVP))
    glDrawArrays(GL_TRIANGLES, 0, 36)
```

[Code] 2-cube-separate

- Same as "2-frustum-perspective.py" from the previous lab, except for the cube size.
 - Cube edge length: 1 $\rightarrow$ 2
- Still using `glDrawArrays()` (separate triangles)

[Code] 2-cube-separate



```
def prepare_vao_cube():
    # prepare vertex data (in main
memory)
    # 36 vertices for 12 triangles
    vertices = glm.array(glm.float32,
        # position      color
        -1 ,  1 ,  1 ,  1, 1, 1, # v0
         1 , -1 ,  1 ,  1, 1, 1, # v2
         1 ,  1 ,  1 ,  1, 1, 1, # v1

        -1 ,  1 ,  1 ,  1, 1, 1, # v0
        -1 , -1 ,  1 ,  1, 1, 1, # v3
         1 , -1 ,  1 ,  1, 1, 1, # v2

        -1 ,  1 , -1 ,  1, 1, 1, # v4
         1 ,  1 , -1 ,  1, 1, 1, # v5
         1 , -1 , -1 ,  1, 1, 1, # v6

        -1 ,  1 , -1 ,  1, 1, 1, # v4
         1 , -1 , -1 ,  1, 1, 1, # v6
        -1 , -1 , -1 ,  1, 1, 1, # v7

        -1 ,  1 ,  1 ,  1, 1, 1, # v0
         1 ,  1 ,  1 ,  1, 1, 1, # v1
         1 ,  1 , -1 ,  1, 1, 1, # v5
```

```
-1 ,  1 ,  1 ,  1, 1, 1, # v0
 1 ,  1 , -1 ,  1, 1, 1, # v5
-1 ,  1 , -1 ,  1, 1, 1, # v4

-1 , -1 ,  1 ,  1, 1, 1, # v3
 1 , -1 , -1 ,  1, 1, 1, # v6
 1 , -1 ,  1 ,  1, 1, 1, # v2

-1 , -1 ,  1 ,  1, 1, 1, # v3
-1 , -1 , -1 ,  1, 1, 1, # v7
 1 , -1 , -1 ,  1, 1, 1, # v6

 1 ,  1 ,  1 ,  1, 1, 1, # v1
 1 , -1 ,  1 ,  1, 1, 1, # v2
 1 , -1 , -1 ,  1, 1, 1, # v6

 1 ,  1 ,  1 ,  1, 1, 1, # v1
 1 , -1 , -1 ,  1, 1, 1, # v6
 1 ,  1 , -1 ,  1, 1, 1, # v5

-1 ,  1 ,  1 ,  1, 1, 1, # v0
-1 , -1 , -1 ,  1, 1, 1, # v7
-1 , -1 ,  1 ,  1, 1, 1, # v3

-1 ,  1 ,  1 ,  1, 1, 1, # v0
-1 ,  1 , -1 ,  1, 1, 1, # v4
-1 , -1 , -1 ,  1, 1, 1, # v7

)
```

Cube

(Indexed Triangle Set – glDrawElements)

Using `glDrawElements()`

- Using `glDrawElements()`, we can specify triangles using **indices**.
 - Indexed Triangle Set
- Indices are provided using an **EBO** (element buffer object).
 - EBO stores indices to reuse vertices, avoiding duplication.

Using glDrawElements()

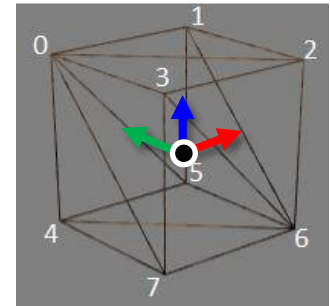
- Preparation (at initialization):
 - 1. Create vertex data **and index data** (in main memory)
 - 2. Create and activate VAO
 - 3. Create and activate VBO
 - **3.5 Create and activate EBO**
 - 4. Copy vertex data to VBO
 - **4.5 Copy index data to EBO**
 - 5. Configure vertex attributes
- Drawing (at every rendering frame):
 - 1. Activate VAO
 - **2. Call glDrawElements()**

P1. Create vertex data and index data

- In this method, the vertex data **should not contain duplicate vertices.**

```
# prepare vertex data
# 8 vertices
vertices = glm.array(glm.float32,
    # position      color
    -1 ,  1 ,  1 ,  1, 1, 1, # v0
     1 ,  1 ,  1 ,  1, 1, 1, # v1
     1 , -1 ,  1 ,  1, 1, 1, # v2
    -1 , -1 ,  1 ,  1, 1, 1, # v3
    -1 ,  1 , -1 ,  1, 1, 1, # v4
     1 ,  1 , -1 ,  1, 1, 1, # v5
     1 , -1 , -1 ,  1, 1, 1, # v6
    -1 , -1 , -1 ,  1, 1, 1, # v7
)
```

```
# prepare index data
# 12 triangles
indices = glm.array(glm.uint32,
    0, 2, 1,
    0, 3, 2,
    4, 5, 6,
    4, 6, 7,
    0, 1, 5,
    0, 5, 4,
    3, 6, 2,
    3, 7, 6,
    1, 2, 6,
    1, 6, 5,
    0, 7, 3,
    0, 4, 7,
```



P3.5 Create and activate EBO

- Element Buffer Object (EBO):
 - A buffer object used to **store vertex indices** that define how vertices form geometric primitives such as triangles.
- Creation and activation are similar to VBOs except the `target` parameter.

```
EBO = glGenBuffers(1)    # create a buffer object
                             ID and store it to EBO variable
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO)  #
activate EBO as an element buffer object
```

P4.5 Copy index data to EBO

- Copying data to EBO is also similar to VBOs except the `target` parameter.

```
glBufferData(GL_ELEMENT_ARRAY_BUFFER,  
indices.nbytes, indices.ptr, GL_STATIC_DRAW) #  
allocate GPU memory for and copy index data to the  
currently bound element buffer
```

D2. Call `glDrawElements()`

- `glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, None)`
- `glDrawElements(mode, count, type, indices)`
 - Render primitives from vertex & index data in VBO & EBO referred by currently bounded VAO.
 - `mode`: Primitive type to render.
 - **`count`**: Number of **elements (indices)** to be rendered.
 - `type`: Type of the values in `indices`.
 - `indices`: Byte offset of the beginning index ('None' for 0)

[Code] 3-cube-indexed

```
def prepare_vao_cube():
    # prepare vertex data (in main memory)
    # 8 vertices
    vertices = glm.array(glm.float32,
        # position      color
        -1, 1, 1, 1, 1, 1, # v0
        1, 1, 1, 1, 1, 1, # v1
        1, -1, 1, 1, 1, 1, # v2
        -1, -1, 1, 1, 1, 1, # v3
        -1, 1, -1, 1, 1, 1, # v4
        1, 1, -1, 1, 1, 1, # v5
        1, -1, -1, 1, 1, 1, # v6
        -1, -1, -1, 1, 1, 1, # v7
    )

    # prepare index data
    # 12 triangles
    indices = glm.array(glm.uint32,
        0, 2, 1,
        0, 3, 2,
        4, 5, 6,
        4, 6, 7,
        0, 1, 5,
        0, 5, 4,
        3, 6, 2,
        3, 7, 6,
        1, 2, 6,
        1, 6, 5,
        0, 7, 3,
        0, 4, 7,
    )
)
```

```
    # create and activate VAO & VBO
    VAO = glGenVertexArrays(1)
    glBindVertexArray(VAO)
    VBO = glGenBuffers(1)
    glBindBuffer(GL_ARRAY_BUFFER, VBO)

    # create and activate EBO
    EBO = glGenBuffers(1)
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO)

    # copy vertex & index data to VBO & EBO
    glBufferData(GL_ARRAY_BUFFER, vertices.nbytes,
vertices.ptr, GL_STATIC_DRAW)
    glBufferData(GL_ELEMENT_ARRAY_BUFFER,
indices.nbytes, indices.ptr, GL_STATIC_DRAW)

    # configure vertex positions
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
6 * glm.sizeof(glm.float32), None)
    glEnableVertexAttribArray(0)

    # configure vertex colors
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,
6 * glm.sizeof(glm.float32),
ctypes.c_void_p(3*glm.sizeof(glm.float32)))
    glEnableVertexAttribArray(1)

    return VAO
```

[Code] 3-cube-indexed

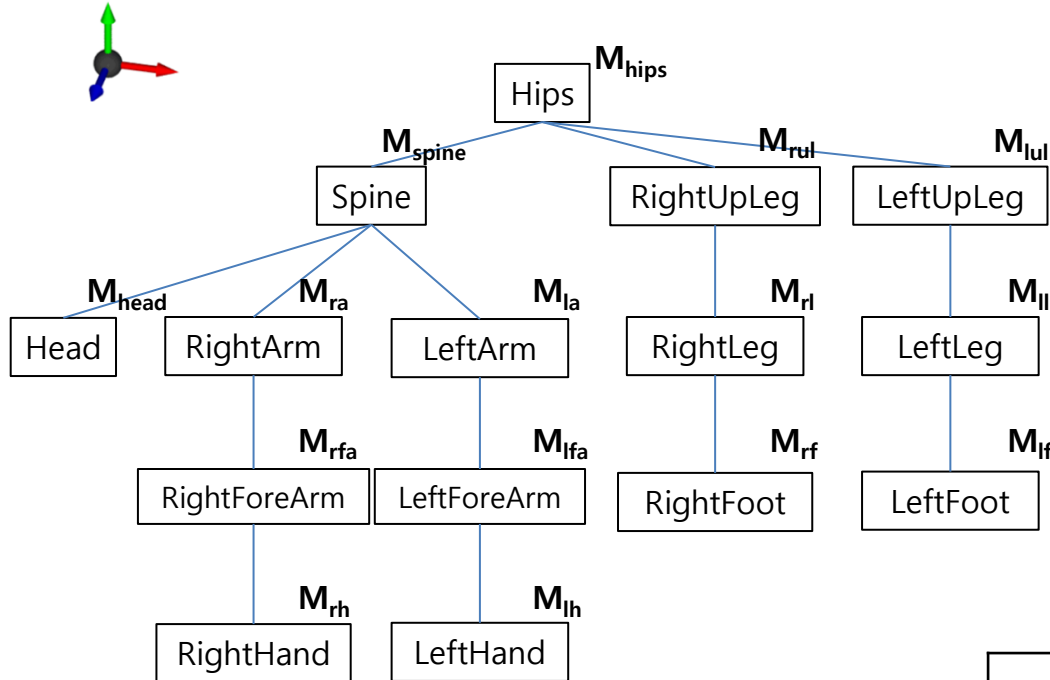
```
def draw_cube(vao, MVP, loc_MVP):  
    glBindVertexArray(vao)  
    glUniformMatrix4fv(loc_MVP, 1, GL_FALSE, glm.value_ptr(MVP))  
    # glDrawArrays(GL_TRIANGLES, 0, 36)  
    glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, None)
```

Time for Lab Check

- Project 2
 - Due: 23:59, May 17, 2026 (NO SCORE for late submissions!)
- Let's start today's lab check.
- TA will guide you.

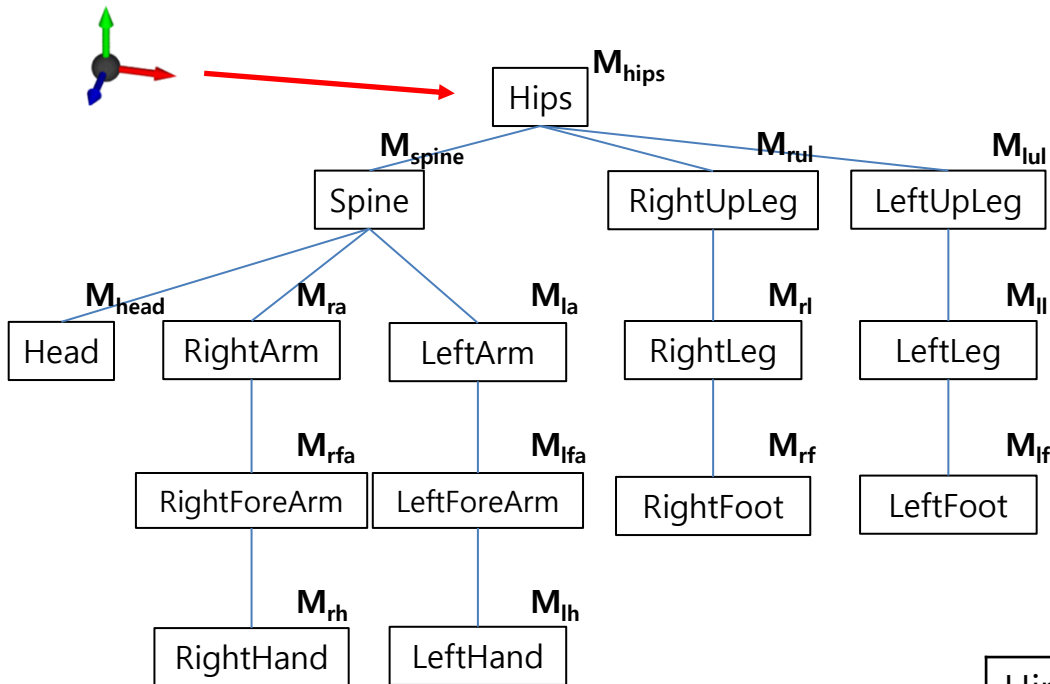
Supplemental Slides

Rendering Hierarchical Models - Method 2

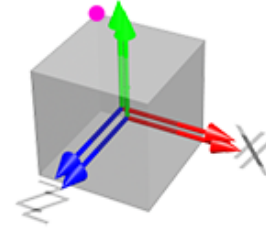


- **Method 2:**
Use *stack* for *depth-first traversal* to compute global transform G_i when rendering.

Rendering Hierarchical Models - Method 2



$\mathbf{p} = [-0.5, 0.5, -0.5]$



Let's say each part is rendered as a unit cube above (without scaling), its vertex position $\mathbf{p}'$ w.r.t. world frame is...

- push: save current transform
- pop: restore parent transform

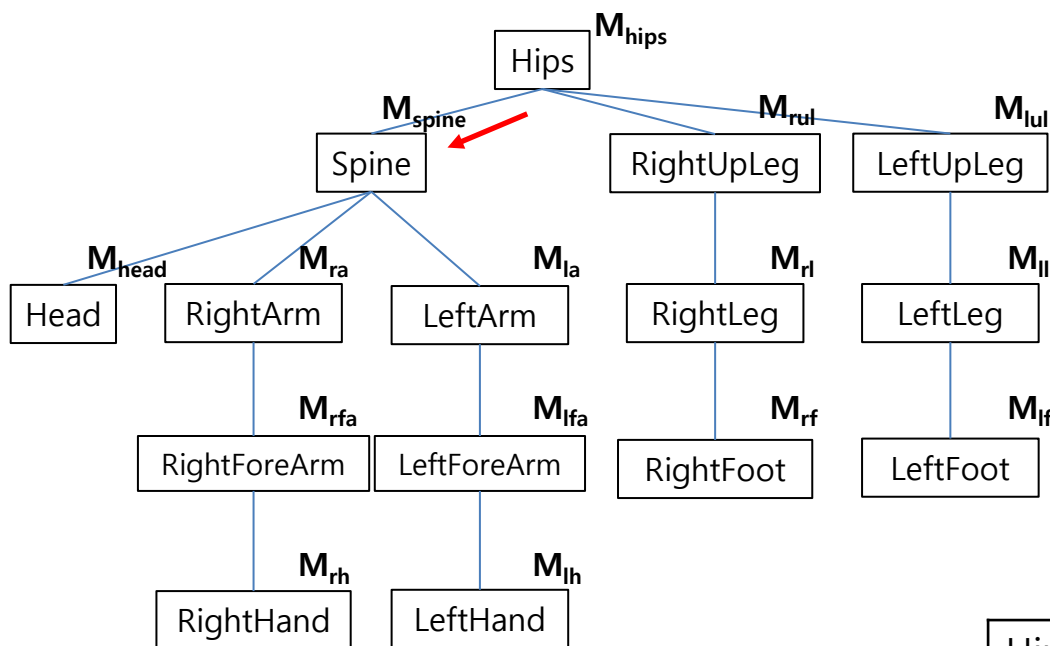
push →

$\mathbf{M}_{\text{hips}}$
$\mathbf{I}$

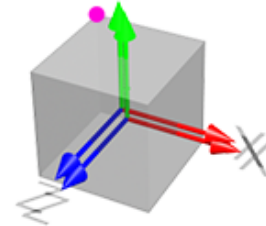
F4020 Yoonsang Lee, 2026

[illegible]

Rendering Hierarchical Models - Method 2



$$\mathbf{p} = [-0.5, 0.5, -0.5]$$



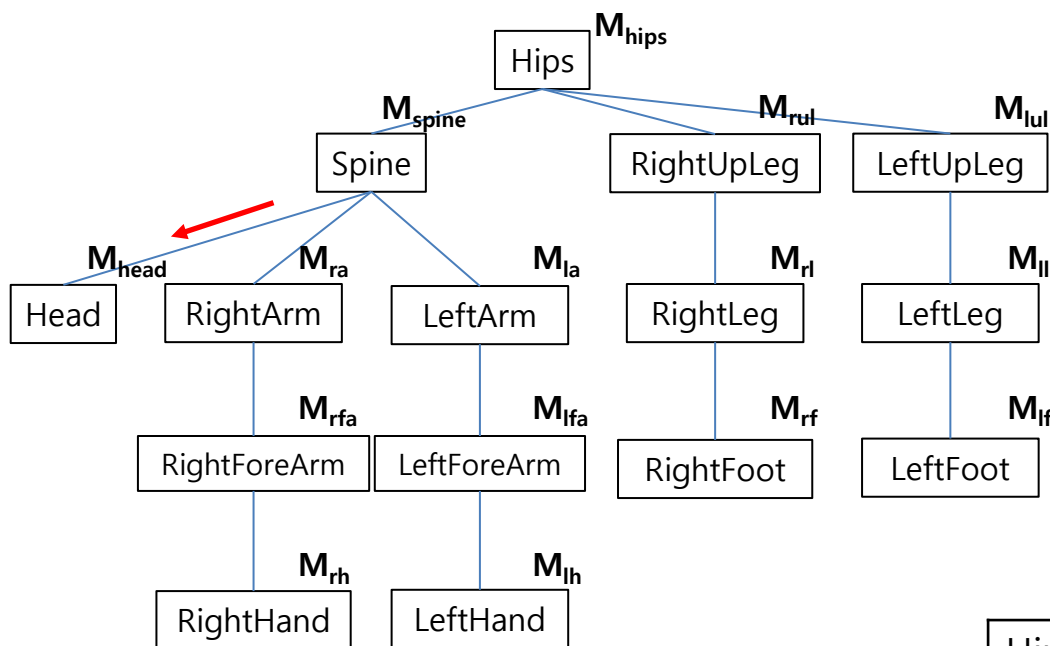
Let's say each part is rendered as a unit cube above (without scaling), its vertex position $\mathbf{p}'$ w.r.t. world frame is...

Hips	$\mathbf{p}' = \mathbf{M}_{\text{hips}} \mathbf{p}$
Spine	$\mathbf{p}' = \mathbf{M}_{\text{hips}} \mathbf{M}_{\text{spine}} \mathbf{p}$

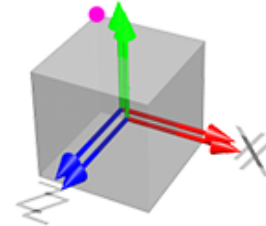
push →

$\mathbf{M}_{\text{hips}} \mathbf{M}_{\text{spine}}$
$\mathbf{M}_{\text{hips}}$
$\mathbf{I}$

Rendering Hierarchical Models - Method 2



$$\mathbf{p} = [-0.5, 0.5, -0.5]$$



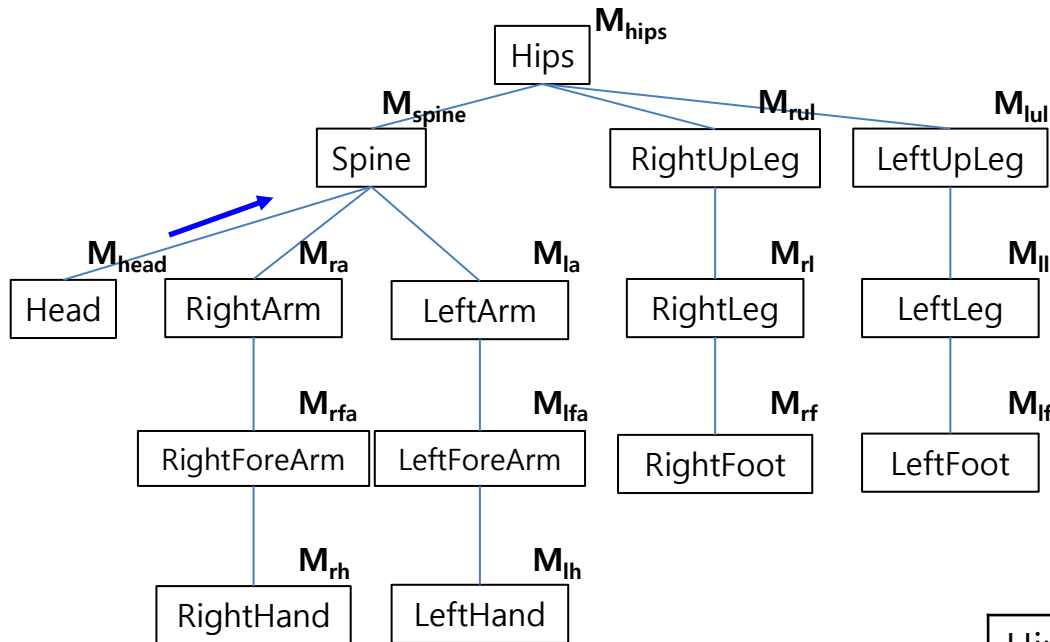
Let's say each part is rendered as a unit cube above (without scaling), its vertex position $\mathbf{p}'$ w.r.t. world frame is...

Hips	$\mathbf{p}' = \mathbf{M}_{\text{hips}} \mathbf{p}$
Spine	$\mathbf{p}' = \mathbf{M}_{\text{hips}} \mathbf{M}_{\text{spine}} \mathbf{p}$
Head	$\mathbf{p}' = \mathbf{M}_{\text{hips}} \mathbf{M}_{\text{spine}} \mathbf{M}_{\text{head}} \mathbf{p}$

push →

$\mathbf{M}_{\text{hips}} \mathbf{M}_{\text{spine}} \mathbf{M}_{\text{head}}$
$\mathbf{M}_{\text{hips}} \mathbf{M}_{\text{spine}}$
$\mathbf{M}_{\text{hips}}$

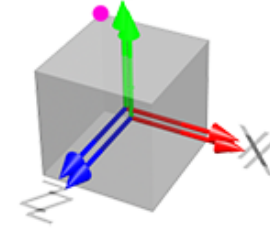
Rendering Hierarchical Models - Method 2



pop ←

$M_{hips} M_{spine}$
M_{hips}
I

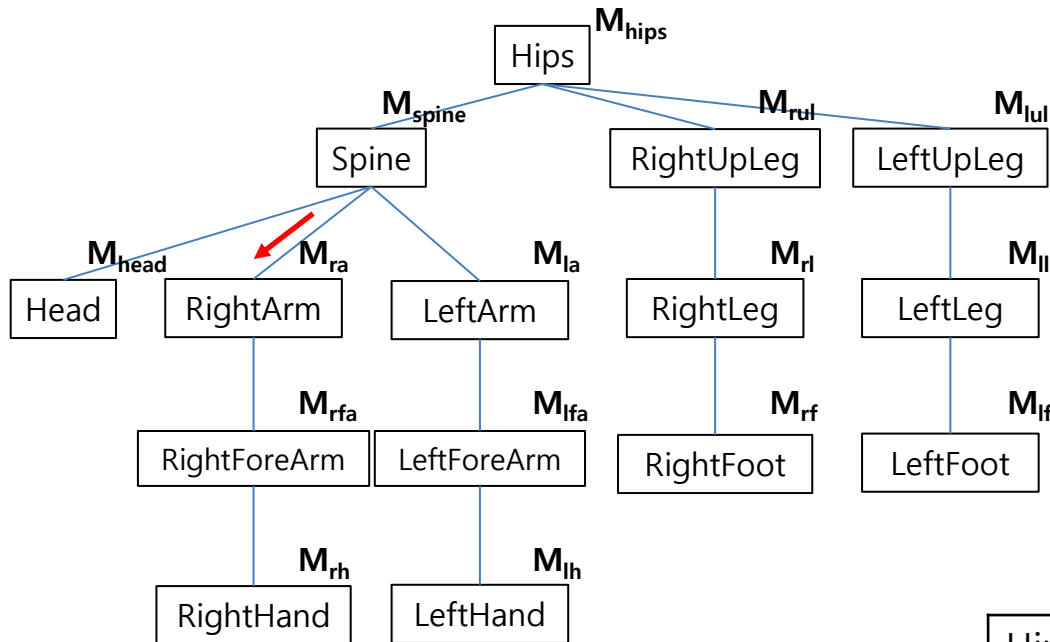
$$\mathbf{p} = [-0.5, 0.5, -0.5]$$



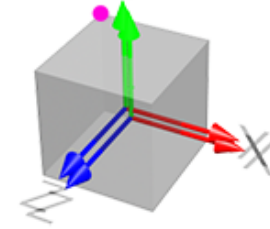
Let's say each part is rendered as a unit cube above (without scaling), its vertex position $\mathbf{p}'$ w.r.t. world frame is...

Hips	$\mathbf{p}' = M_{hips} \mathbf{p}$
Spine	$\mathbf{p}' = M_{hips} M_{spine} \mathbf{p}$
Head	$\mathbf{p}' = M_{hips} M_{spine} M_{head} \mathbf{p}$

Rendering Hierarchical Models - Method 2



$$\mathbf{p} = [-0.5, 0.5, -0.5]$$



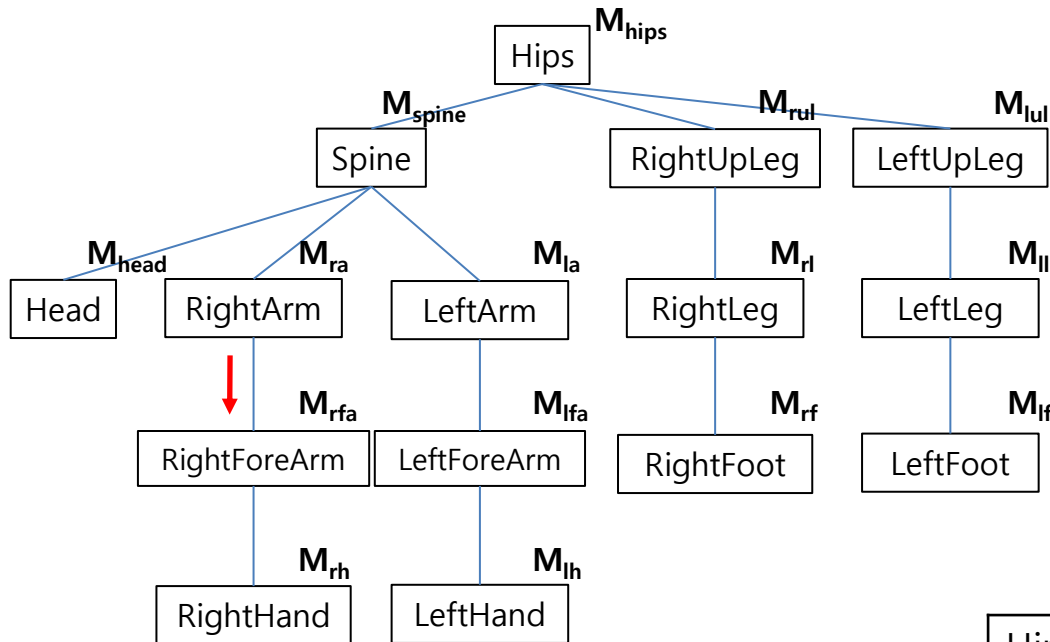
Let's say each part is rendered as a unit cube above (without scaling), its vertex position $\mathbf{p}'$ w.r.t. world frame is...

push →

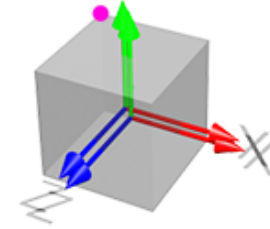
$\mathbf{M}_{hips} \mathbf{M}_{spine} \mathbf{M}_{ra}$
$\mathbf{M}_{hips} \mathbf{M}_{spine}$
$\mathbf{M}_{hips}$

Hips	$\mathbf{p}' = \mathbf{M}_{hips} \mathbf{p}$
Spine	$\mathbf{p}' = \mathbf{M}_{hips} \mathbf{M}_{spine} \mathbf{p}$
Head	$\mathbf{p}' = \mathbf{M}_{hips} \mathbf{M}_{spine} \mathbf{M}_{head} \mathbf{p}$
RightArm	$\mathbf{p}' = \mathbf{M}_{hips} \mathbf{M}_{spine} \mathbf{M}_{ra} \mathbf{p}$

Rendering Hierarchical Models - Method 2



$$\mathbf{p} = [-0.5, 0.5, -0.5]$$



Let's say each part is rendered as a unit cube above (without scaling), its vertex position $\mathbf{p}'$ w.r.t. world frame is...

push →

$\mathbf{M}_{hips} \mathbf{M}_{spine} \mathbf{M}_{ra} \mathbf{M}_{rfa}$	
$\mathbf{M}_{hips} \mathbf{M}_{spine} \mathbf{M}_{ra}$	
$\mathbf{M}_{hips} \mathbf{M}_{spine}$	
$\mathbf{M}_{hips}$	

Hips	$\mathbf{p}' = \mathbf{M}_{hips} \mathbf{p}$
Spine	$\mathbf{p}' = \mathbf{M}_{hips} \mathbf{M}_{spine} \mathbf{p}$
Head	$\mathbf{p}' = \mathbf{M}_{hips} \mathbf{M}_{spine} \mathbf{M}_{head} \mathbf{p}$
RightArm	$\mathbf{p}' = \mathbf{M}_{hips} \mathbf{M}_{spine} \mathbf{M}_{ra} \mathbf{p}$
RightForeArm	$\mathbf{p}' = \mathbf{M}_{hips} \mathbf{M}_{spine} \mathbf{M}_{ra} \mathbf{M}_{rfa} \mathbf{p}$
...	