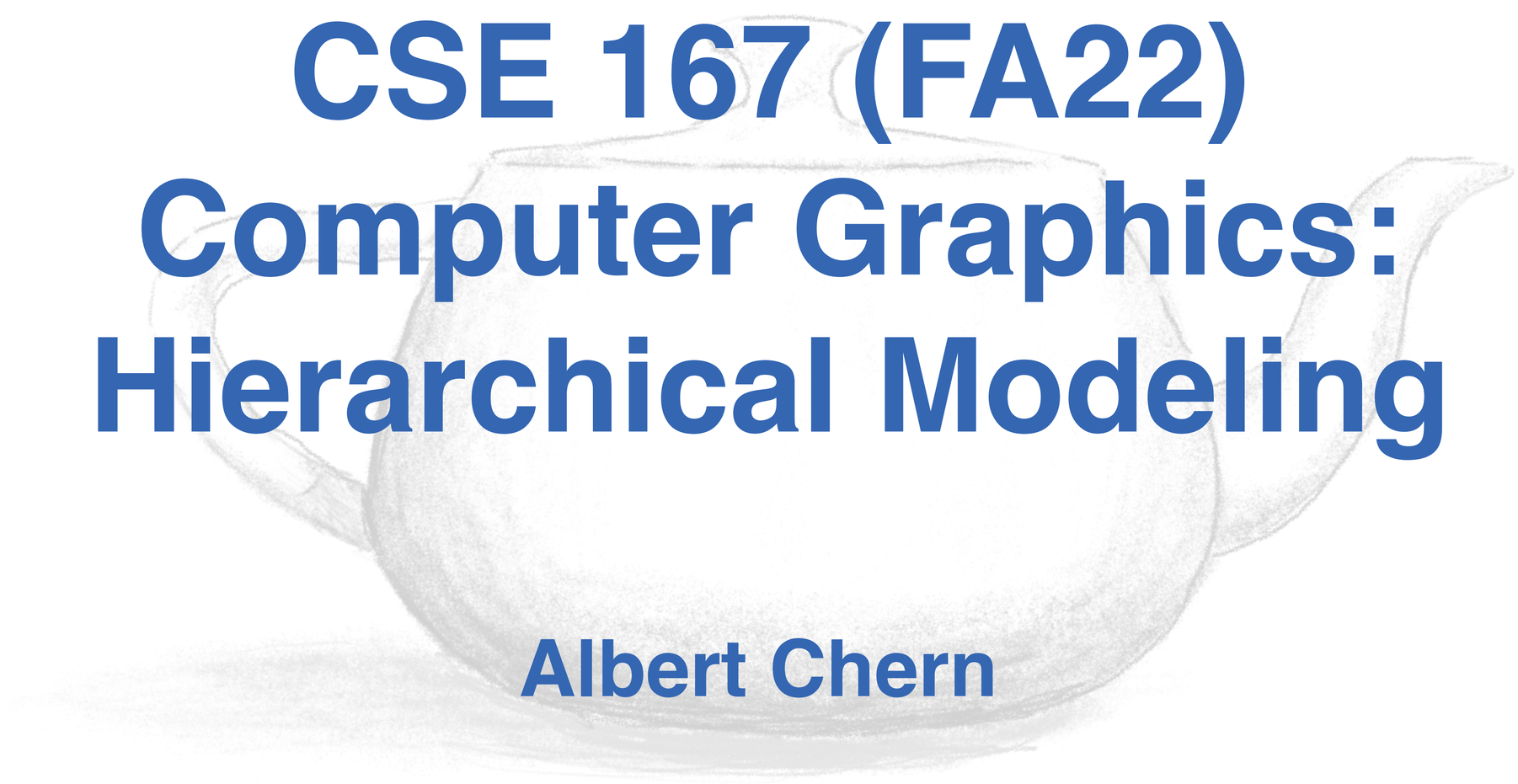
CSE 167 (FA22)
Computer Graphics:
Hierarchical Modeling



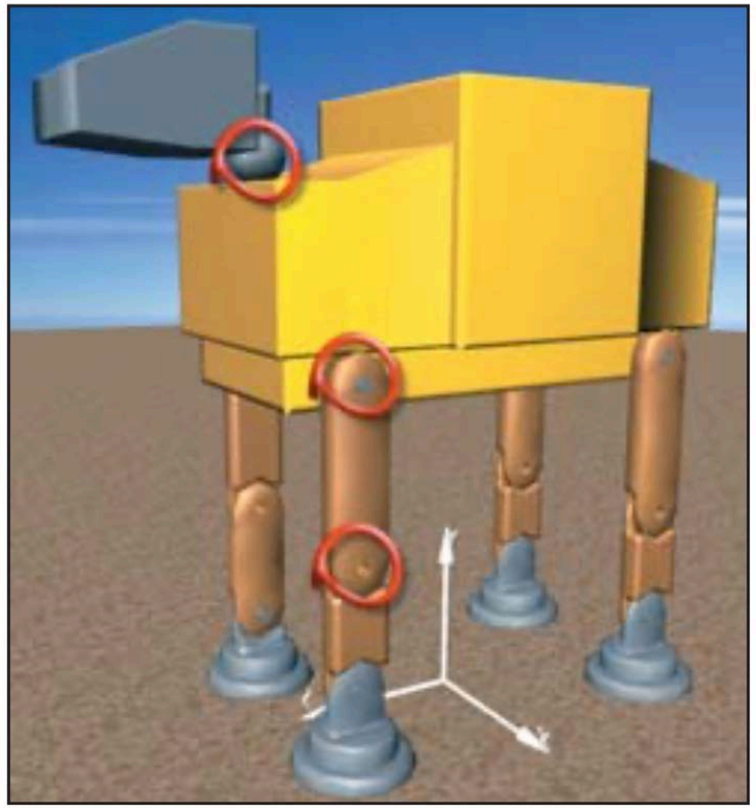
Albert Chern

UC San Diego
JACOBS SCHOOL OF ENGINEERING

Complex Scenes

- Complex scenes
- Matrix stack
- Drawing command sequence
- Graph traversal
- Scene graph data structure

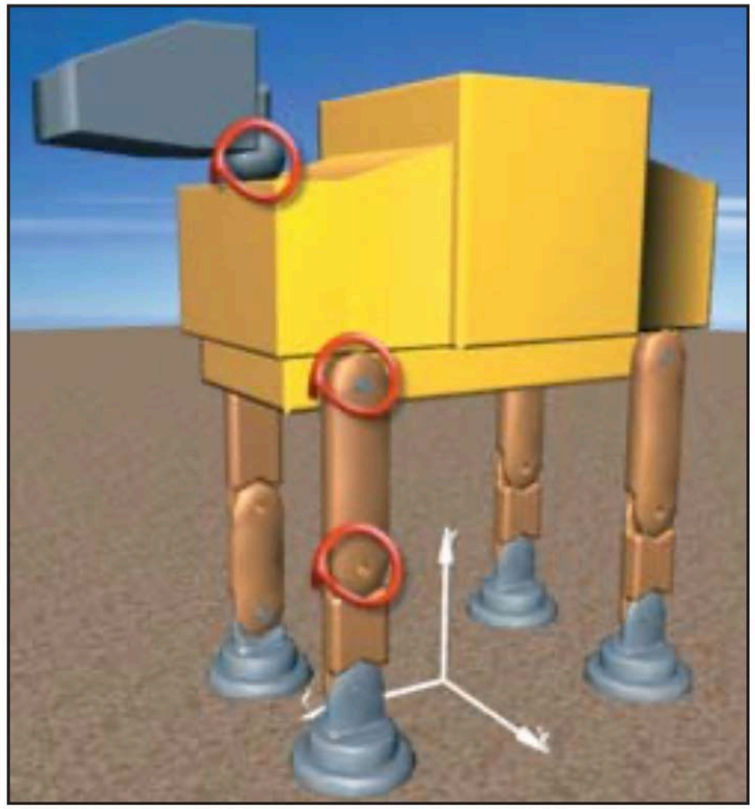
Modeling a complex object



Modeling a complex object

Motivation for **modular modeling**

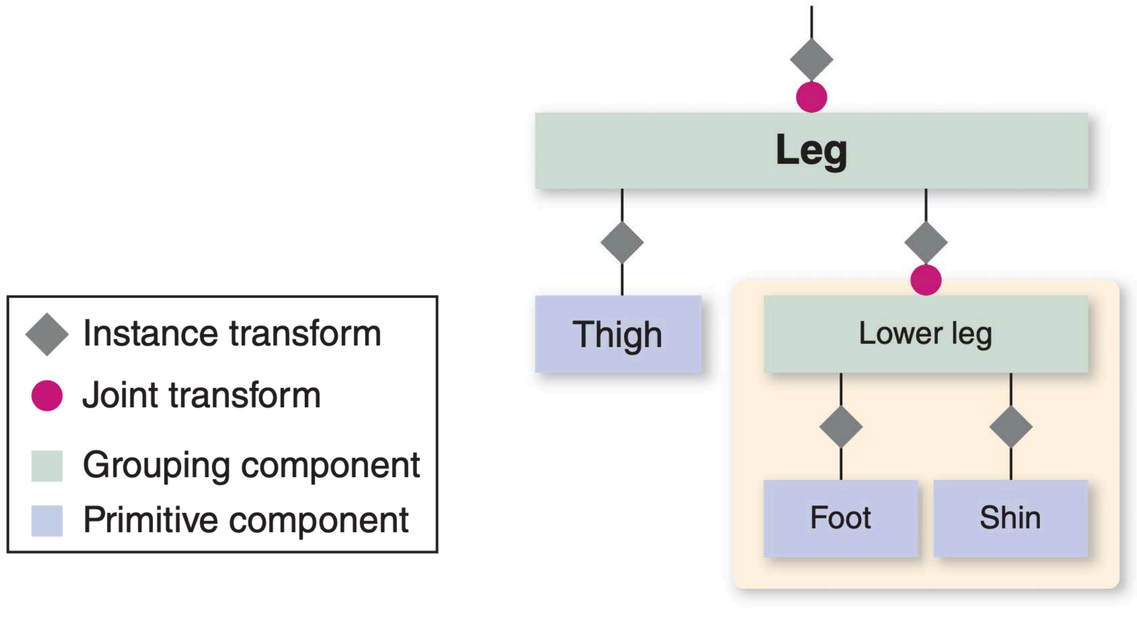
- A model may require many subcomponents.
  - Materials can vary over different subcomponents.
- A component can appear multiple times.
  - Define a component once and instance it multiple times to optimize memory usage for complex scenes.
- Using correlated subcomponents facilitates the animation of subparts.
  - Just apply simple transformations at the joint, instead of editing the whole mesh.
  - When a leg is transformed, the foot beneath it should also be transformed together.

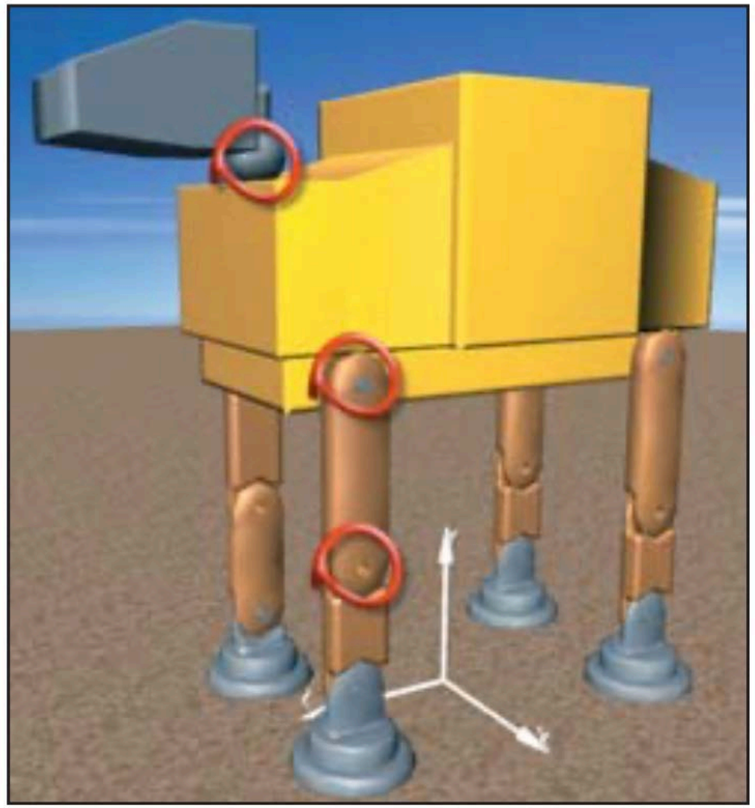


Modeling a complex object

The Hierarchical modeling principle

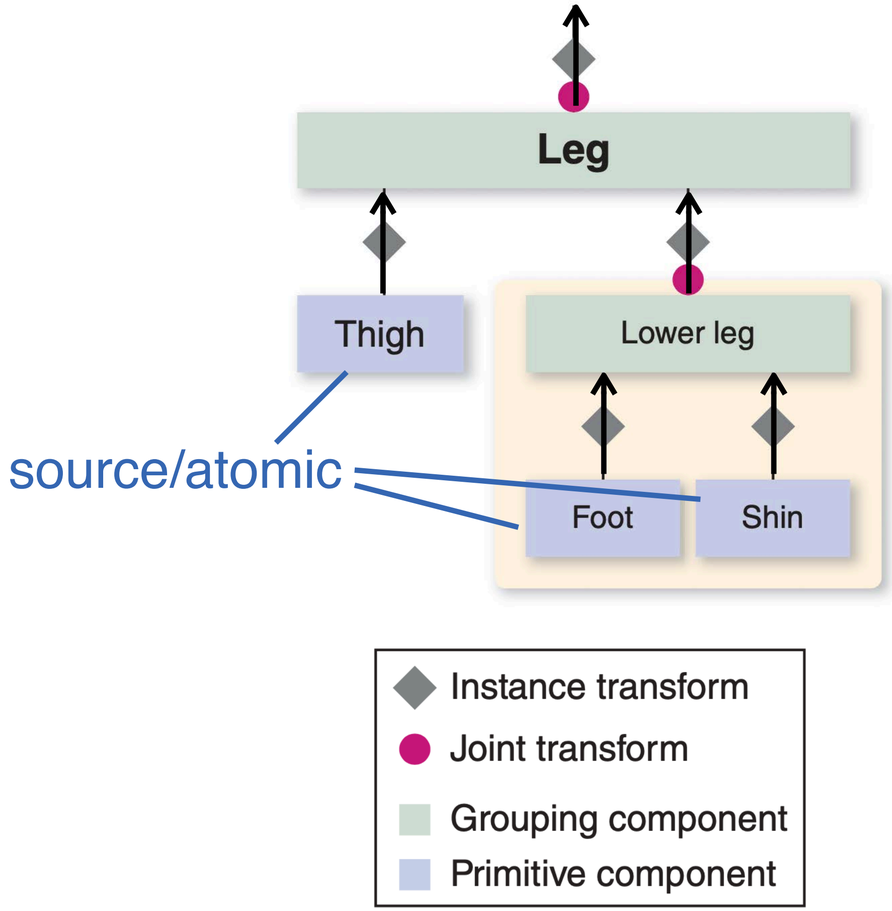
Whenever possible, construct models hierarchically.
Try to make the modeling hierarchy correspond to a functional hierarchy for ease of animation.





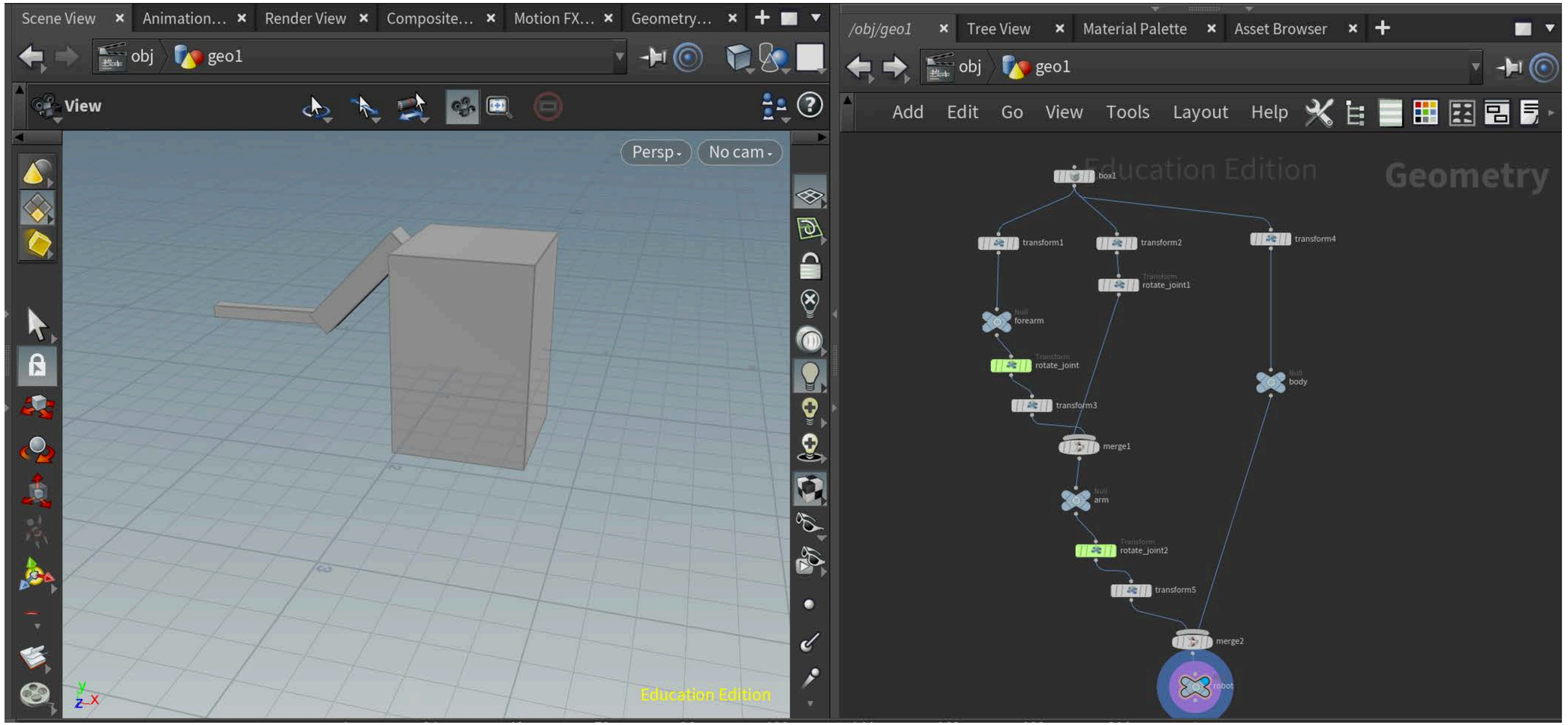
Scene graph

- The diagram showing the hierarchical relationship is the **scene graph**.
  - Nodes are atomic (source/leaf) or group components.
  - The scene graph is an acyclic directed graph.
  - Each edge is annotated with a transformation matrix
- For a graphics software the supports scene graph:
  - There is a scene graph data structure.
  - There is an algorithm that traverses over the graph and renders every component efficiently.



Softwares used in industry

- e.g. SideFX Houdini



Plan

We will separate our discussion into two levels

- First, given a conceptual scene graph, we study what the sequence of commands is to render the scene graph efficiently.

- We design a scene graph data structure, so that the sequence of commands are generated automatically by some graph traversal algorithm.

Matrix stack

- Complex scenes
- Matrix stack
- Drawing command sequence
- Graph traversal
- Scene graph data structure

What are the info for drawing an object?

- Geometry spreadsheet (VAO)
  - Usually static. We don't change the value in VAO if we just linear/affine/projectively transform the object, or move around the camera.
- Modelview matrix (VM)
  - If V is the view matrix, and M is the model matrix, then the modelview matrix is VM.
- Projection matrix (P)
  - The matrix for perspectivity. Usually fixed during the scene.

Basic setup

Vertex shader

```
#version 330 core

// Inputs
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 color;

// Uniforms
uniform mat4 projection;
uniform mat4 modelview;

// Extra outputs, if any
out vec3 color;

void main() {
  gl_Position = projection * modelview
        * vec4(position, 1.0f);
}
```

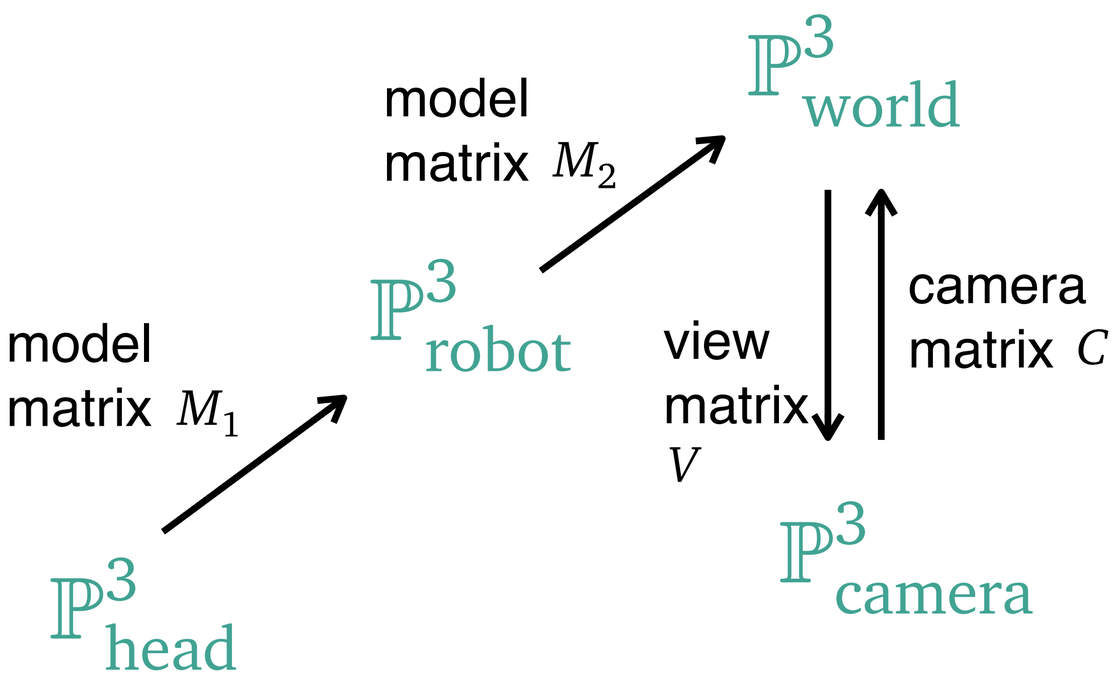
Basic setup

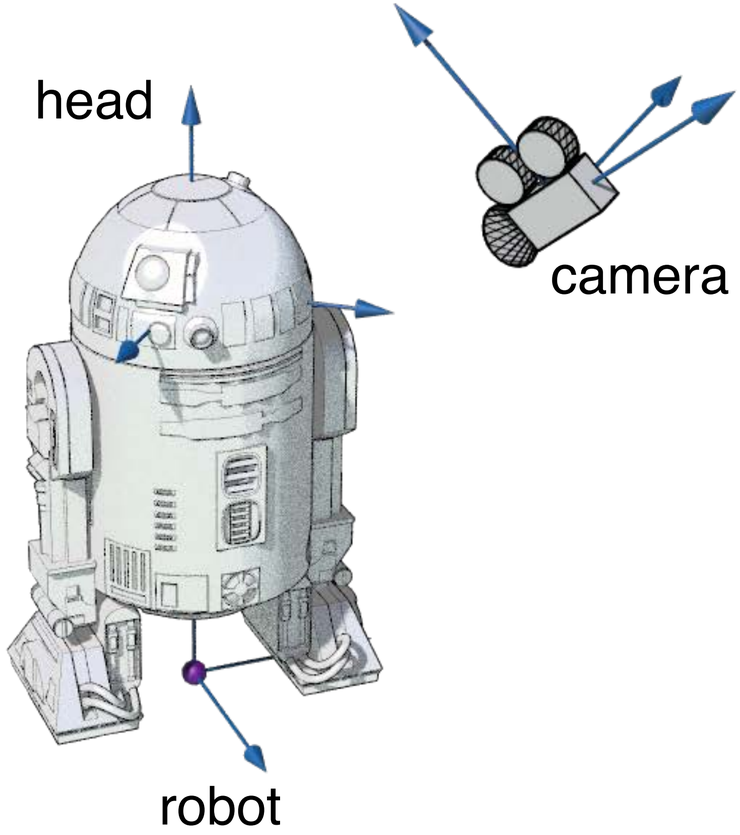
- We often need to draw a lot of objects, each of which need a **modelview** matrix

```
void display(void){

    ... // compute VM1

    glUniformMatrix4fv(modelviewPos, 1, GL_FALSE, &(VM1)[0][0]);
    drawMyObj1;

    ... // compute VM2

    glUniformMatrix4fv(modelviewPos, 1, GL_FALSE, &(VM2)[0][0]);
    drawMyObj2;

    ...

}
```

Scene Hierarchy

- We often need to draw a lot of objects, each of which need a **modelview** matrix





Matrix Stack

- In a large scene with many objects, whose modelview matrices are

$$VM_1M_2M_3M_5 \qquad VM_1M_2M_3M_6 \qquad VM_1M_2M_3M_5M_7 \quad \text{etc.}$$

  - We don’t want to recompute repetitively the same matrix multiplications.
  - We use a **stack** to store the results of matrix multiplication of intermediate stages.

Stack

Definition

A **stack** of type $\mathbb{T}$

`std::stack<`$\mathbb{T}$`>` $\mathbf{a}$

is an array of objects of type $\mathbb{T}$ with an arbitrary length

$$\mathbf{a} = (a_1, \ldots, a_k) \in \mathbb{T}^k$$

together with the following three operations:

- push
- pop
- top

Stack

- push

  $\textsc{Push}: \mathbb{T}^k \times \mathbb{T} \longrightarrow \mathbb{T}^{k+1}$

  $(a_1, \ldots, a_k).\textsc{Push}(b) = (a_1, \ldots, a_k, b)$

- pop
- top

Stack

- push

  $\textsc{Push}\colon \mathbb{T}^k \times \mathbb{T} \to \mathbb{T}^{k+1}$

  $(a_1, \ldots, a_k).\textsc{Push}(b) = (a_1, \ldots, a_k, b)$

- pop

  $\textsc{Pop}\colon \mathbb{T}^k \to \mathbb{T}^{k-1}$

  $(a_1, \ldots, a_{k-1}, a_k).\textsc{Pop}() = (a_1, \ldots, a_{k-1})$

- top

Stack

- push

  $\textsc{Push}: \mathbb{T}^k \times \mathbb{T} \to \mathbb{T}^{k+1}$

  $(a_1, \ldots, a_k).\textsc{Push}(b) = (a_1, \ldots, a_k, b)$

- pop

  $\textsc{Pop}: \mathbb{T}^k \to \mathbb{T}^{k-1}$

  $(a_1, \ldots, a_{k-1}, a_k).\textsc{Pop}() = (a_1, \ldots, a_{k-1})$

- top

  $\textsc{Top}: \mathbb{T}^k \to \mathbb{T}$

  $(a_1, \ldots, a_k).\textsc{Top}() = a_k$

Stack

- push

  $\textsc{Push}: \mathbb{T}^k \times \mathbb{T} \to \mathbb{T}^{k+1}$

  $(a_1, \ldots, a_k).\textsc{Push}(b) = (a_1, \ldots, a_k, b)$

- pop

  $\textsc{Pop}: \mathbb{T}^k \to \mathbb{T}^{k-1}$

  $(a_1, \ldots, a_{k-1}, a_k).\textsc{Pop}() = (a_1, \ldots, a_{k-1})$

- top

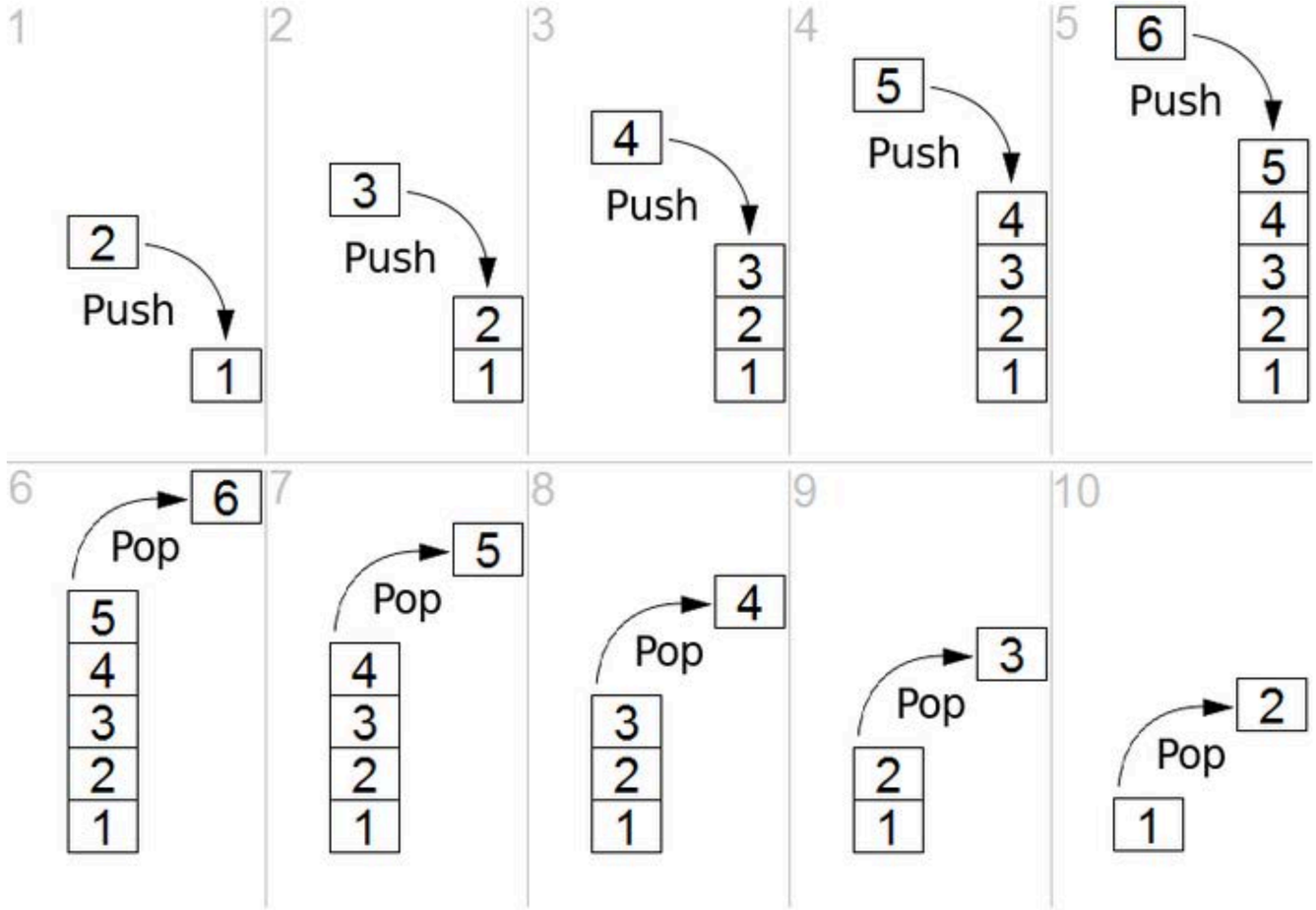
  $\textsc{Top}: \mathbb{T}^k \to \mathbb{T}$

  $(a_1, \ldots, a_k).\textsc{Top}() = a_k$

We only have access to the top of stack

Stack

- Sometimes, pop refers to top-pop combined.



We only have access to the top of stack

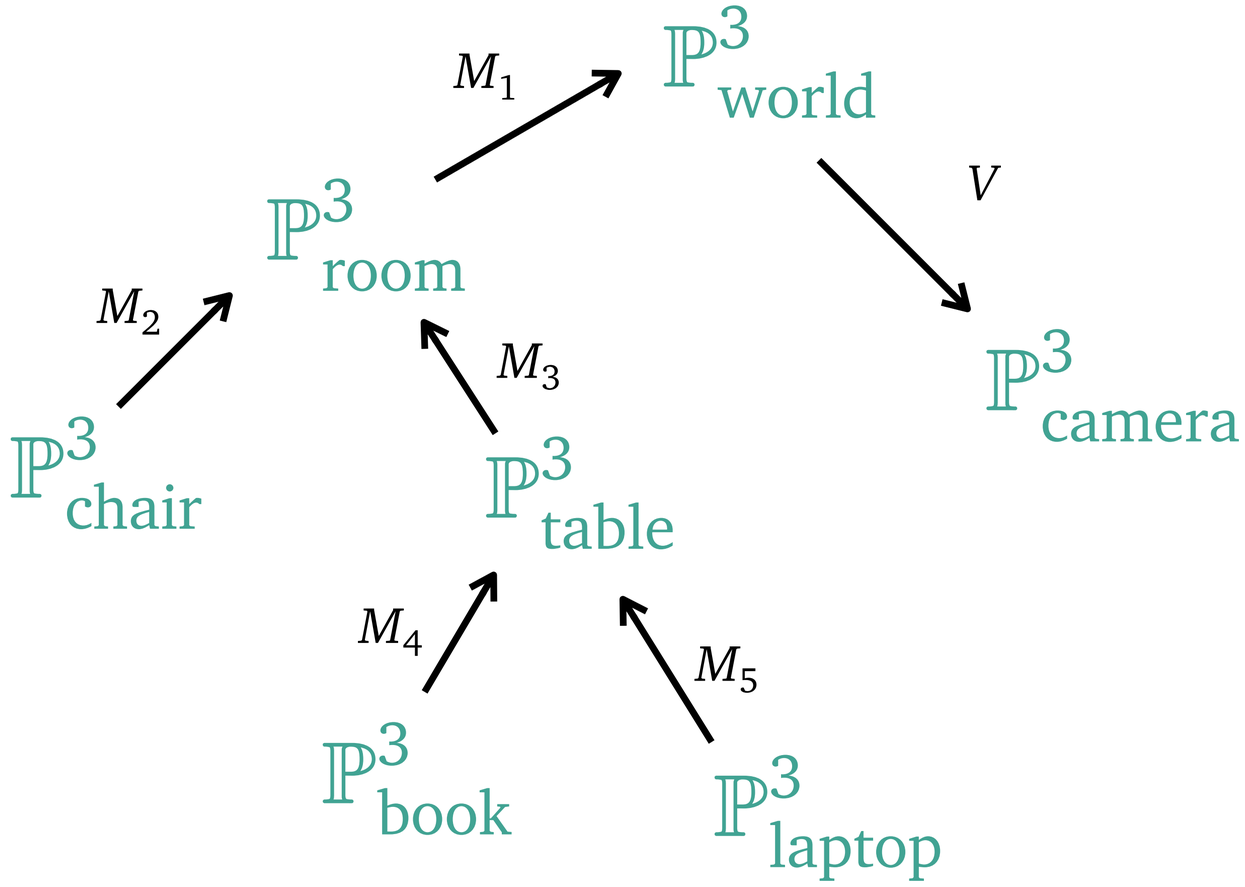
Last-in, first-out

Sequence of commands for rendering a scene using a matrix stack

- Complex scenes
- Matrix stack
- Drawing command sequence
- Graph traversal
- Scene graph data structure

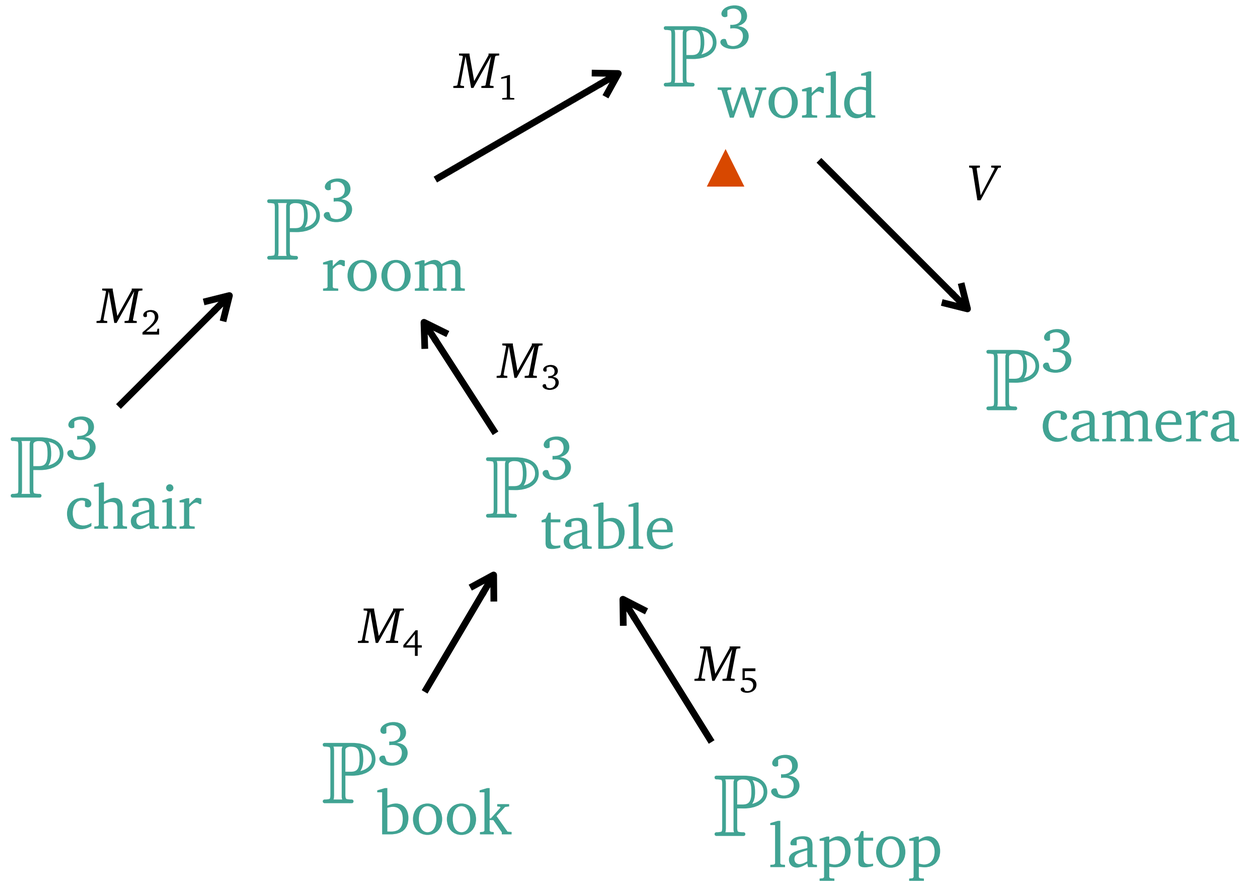
General principle

- Prepare two variables
  - Current modelview matrix
  - A stack of some previously calculated matrices
- When moving down:
  - Push current into stack
  - Update current modelview by multiplying current by the matrix on the edge
- When moving back up:
  - Replace current by pop of stack



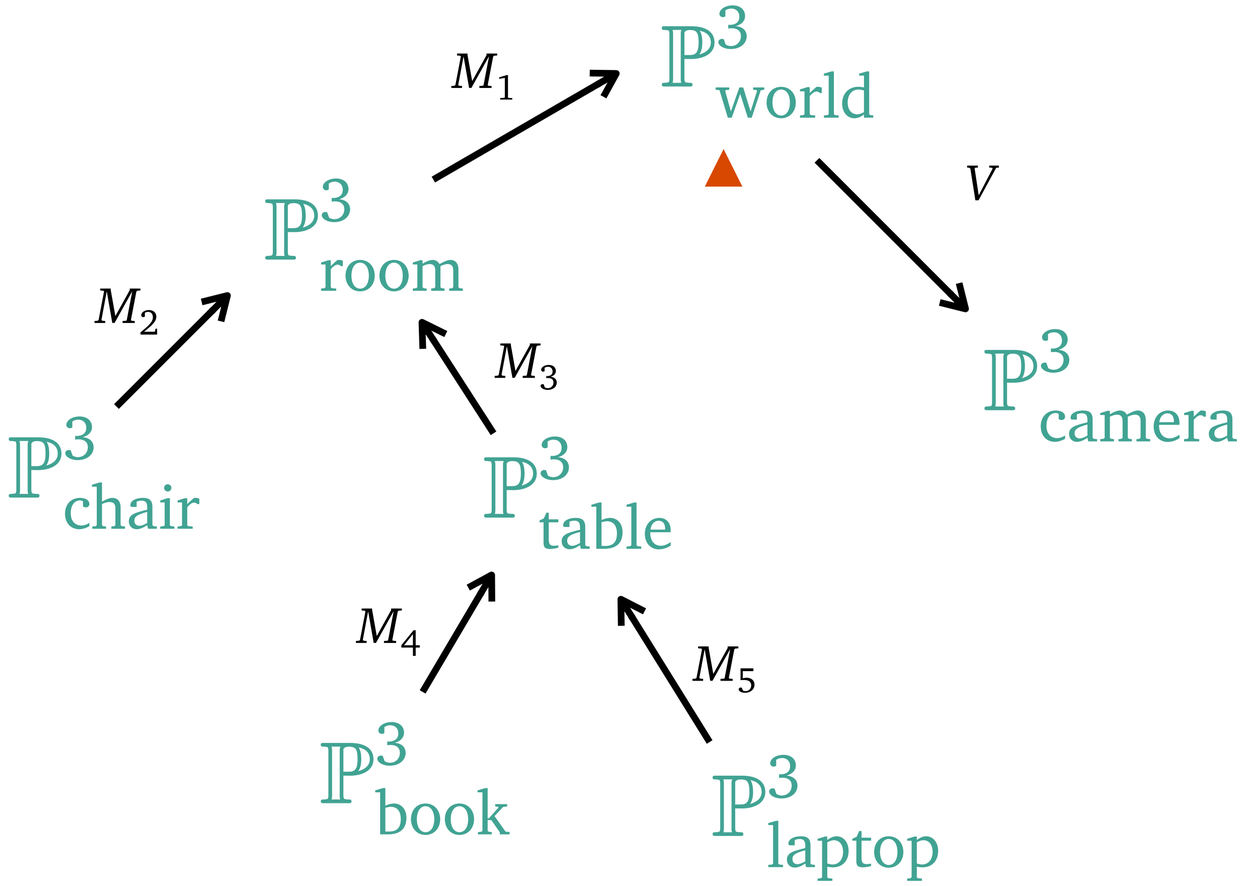
Matrix Stack

- Define a **modelviewStack** $\textsc{Stack} \in$ `std::stack<glm::mat4>`
- Let **VM** be the current modelview matrix
- Initially $\text{VM} = V$



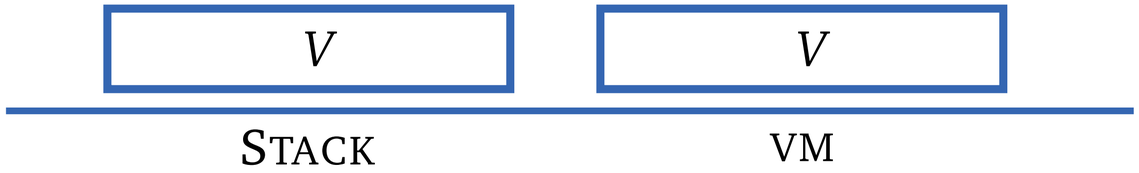
Matrix Stack

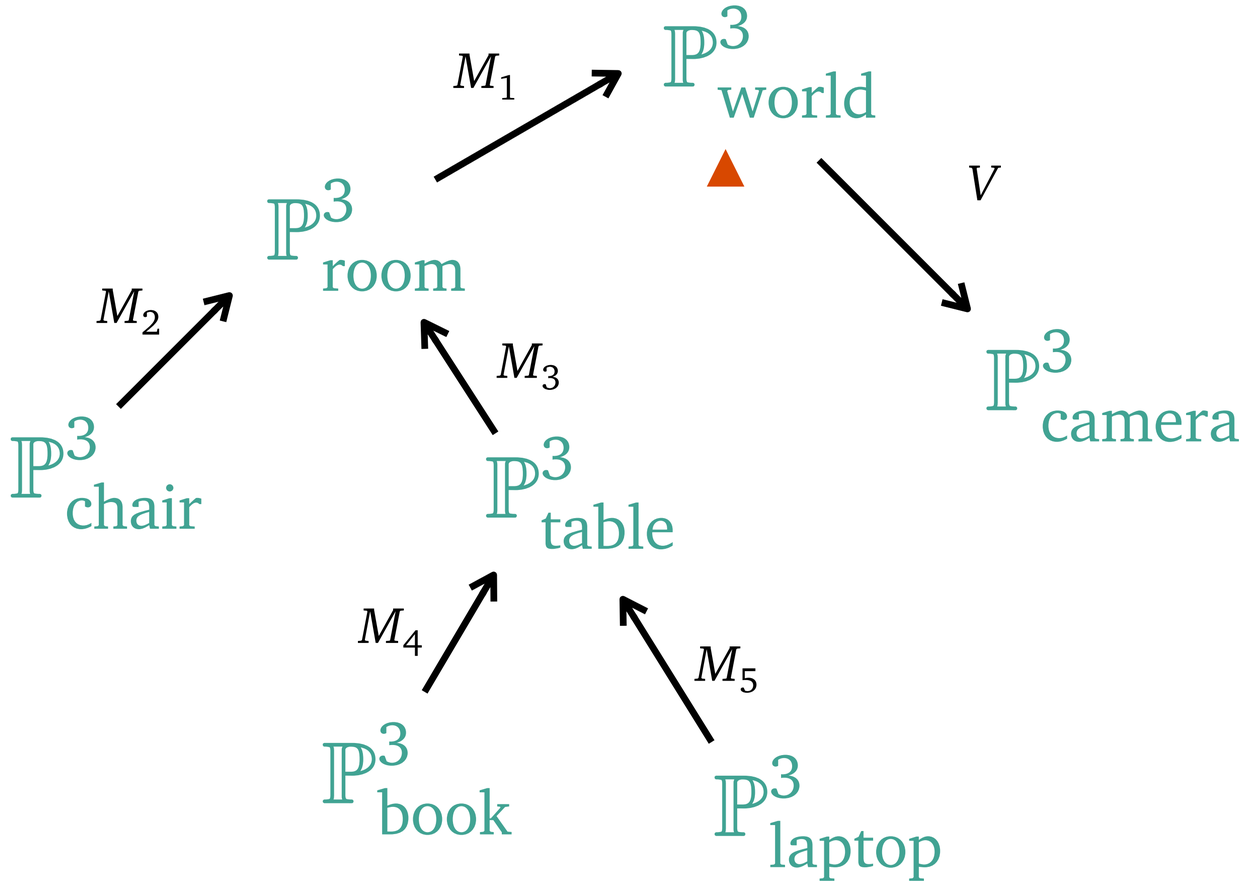
- Define a **modelviewStack** STACK
- Let **VM** be the current modelview matrix
- Initially $\text{VM} = V$
- Push STACK.PUSH(VM)



Matrix Stack

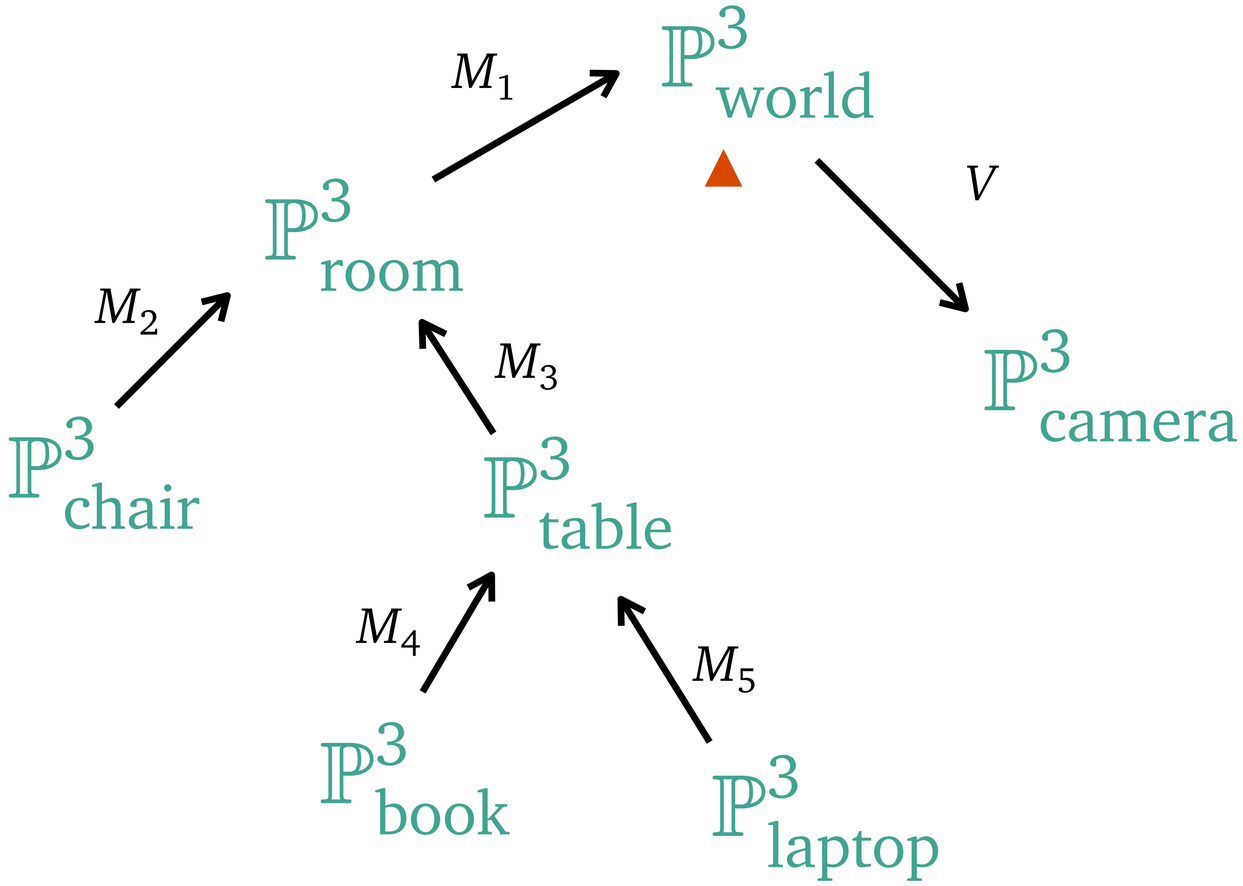
- Define a **modelviewStack** STACK
- Let **VM** be the current modelview matrix
- Initially $\text{VM} = V$
- Push STACK.PUSH(VM)

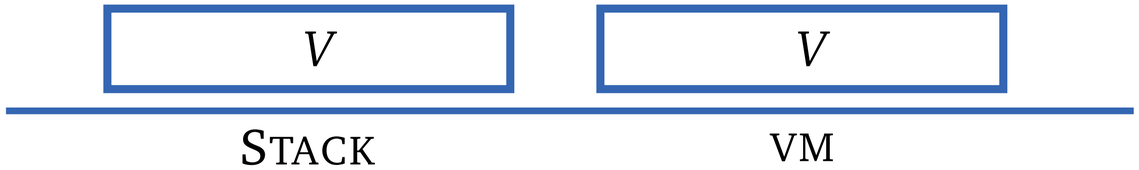




Matrix Stack

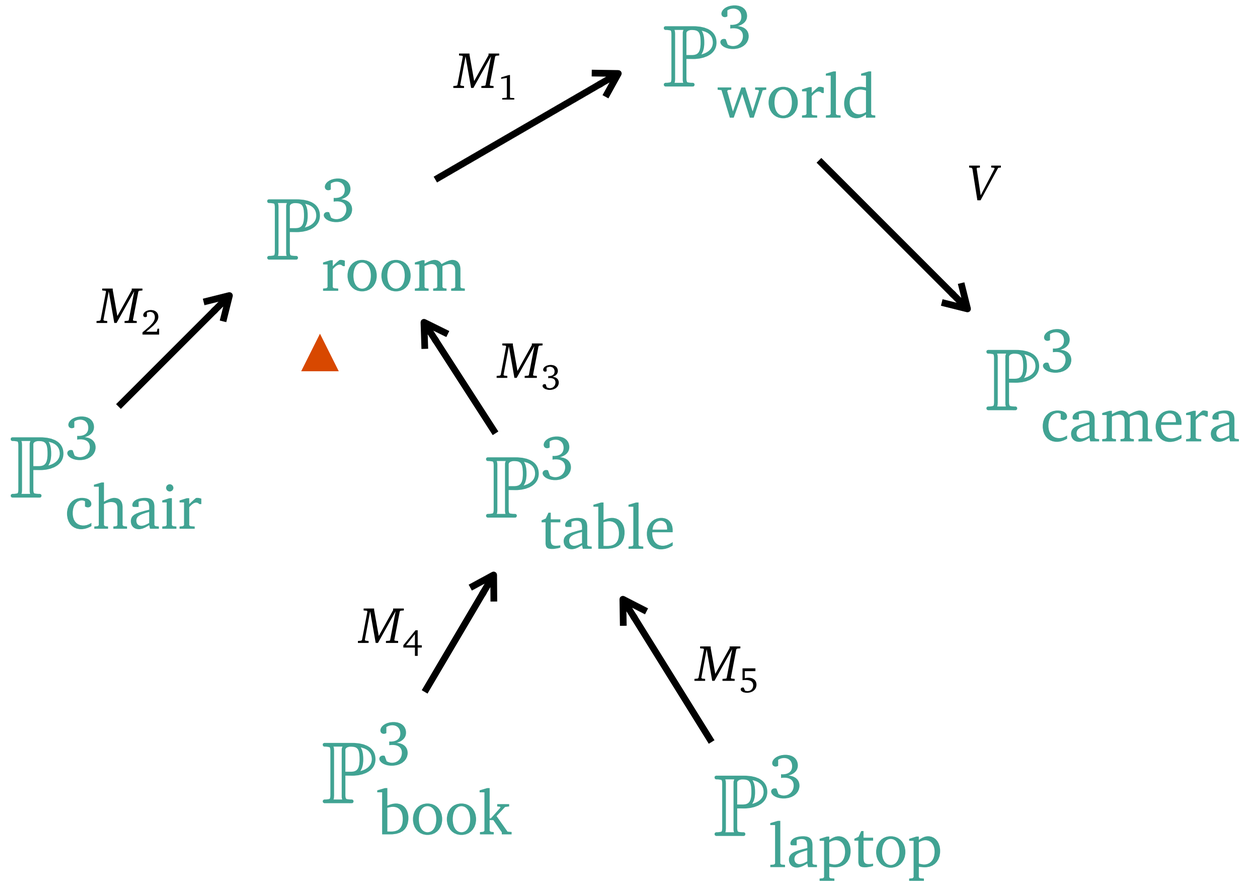
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$

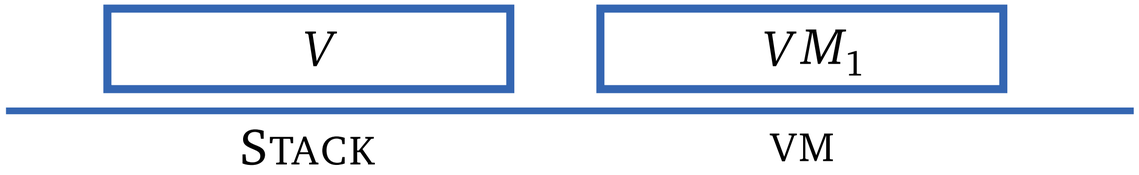




Matrix Stack

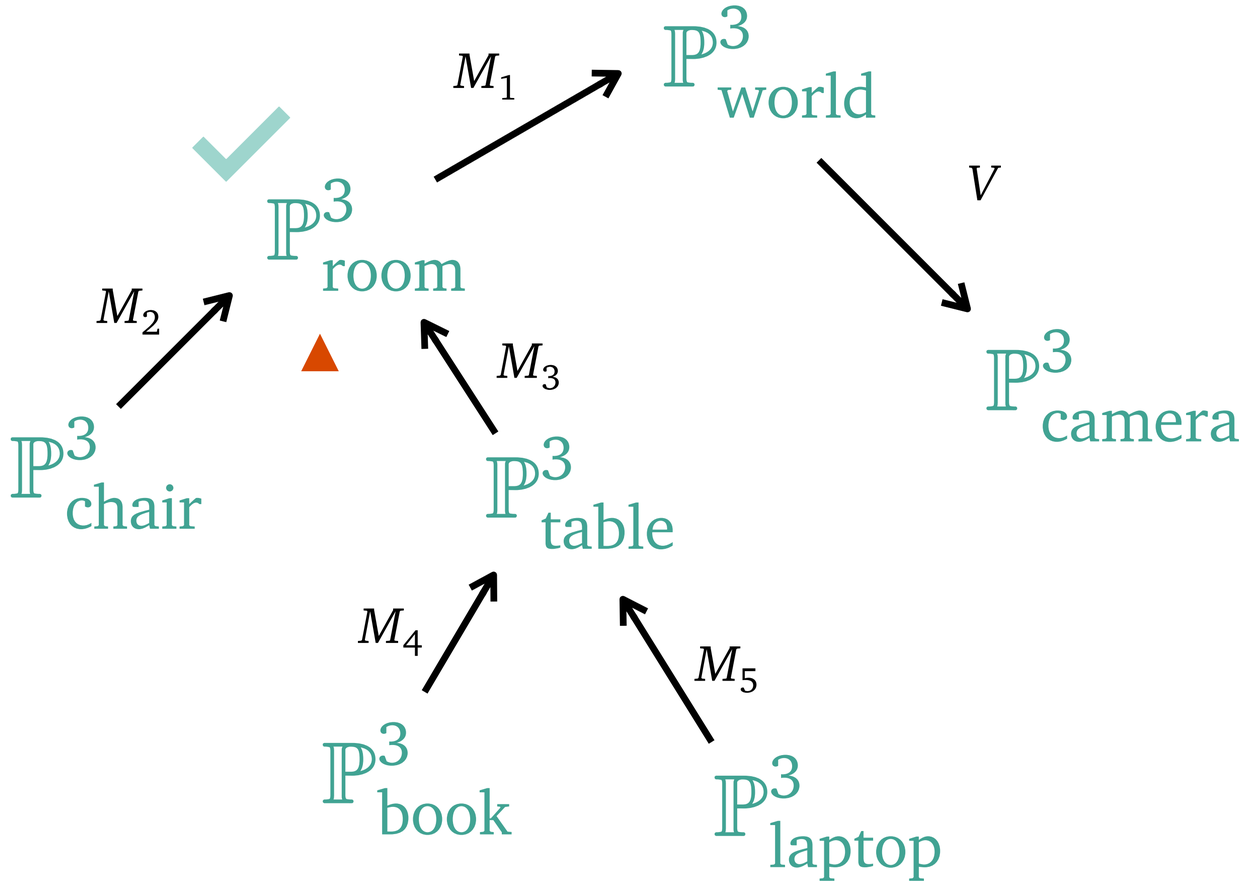
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$

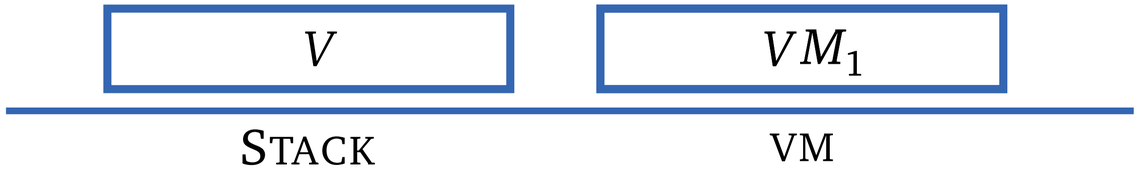




Matrix Stack

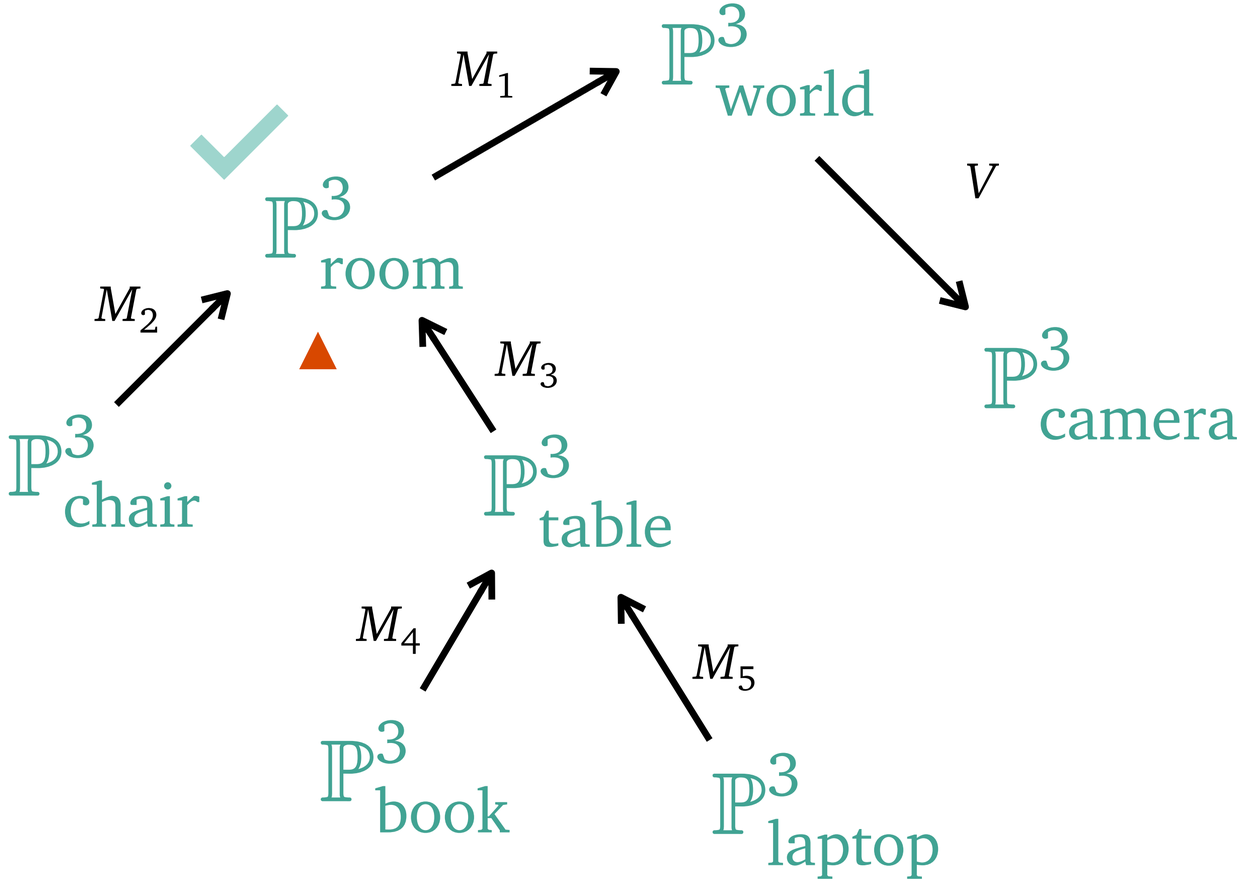
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom

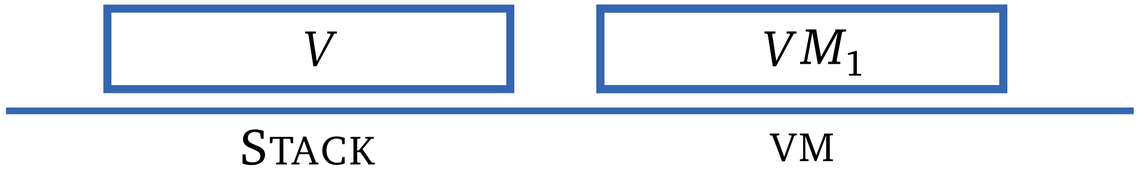




Matrix Stack

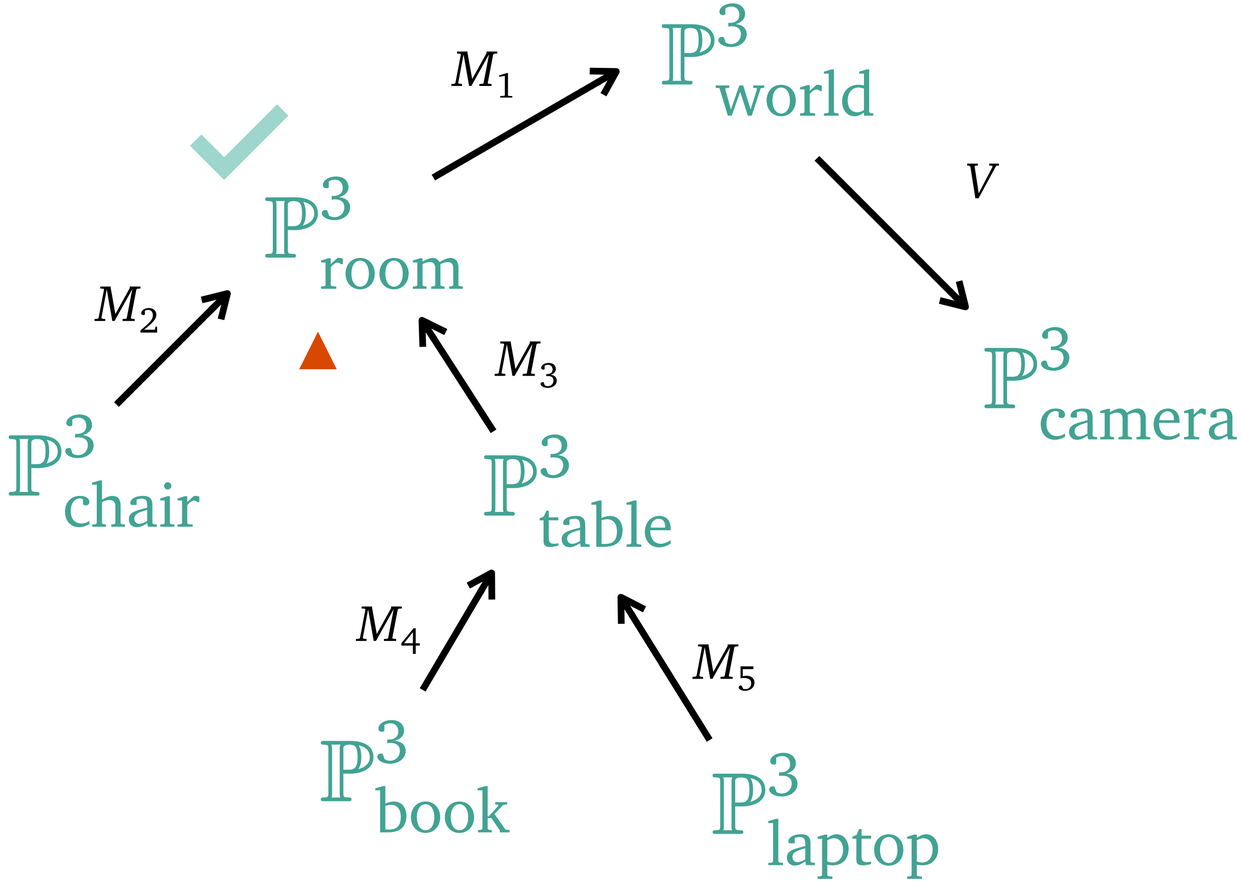
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$

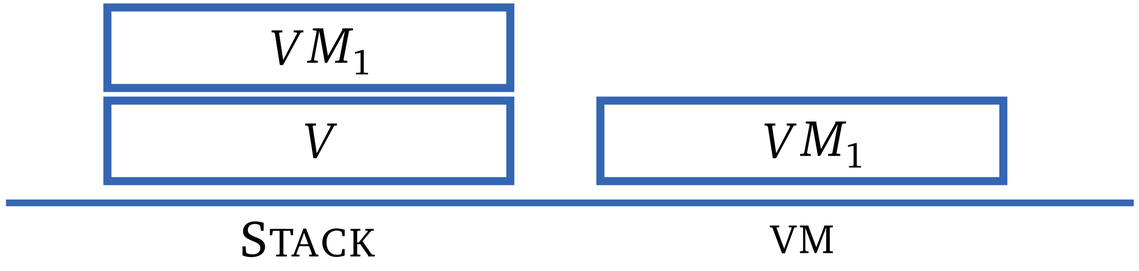




Matrix Stack

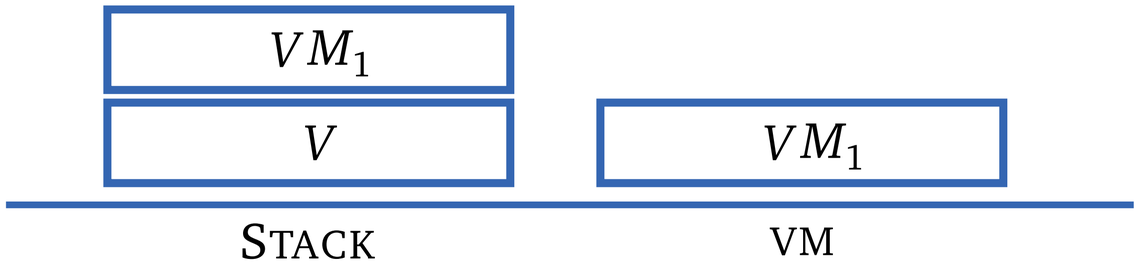
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$

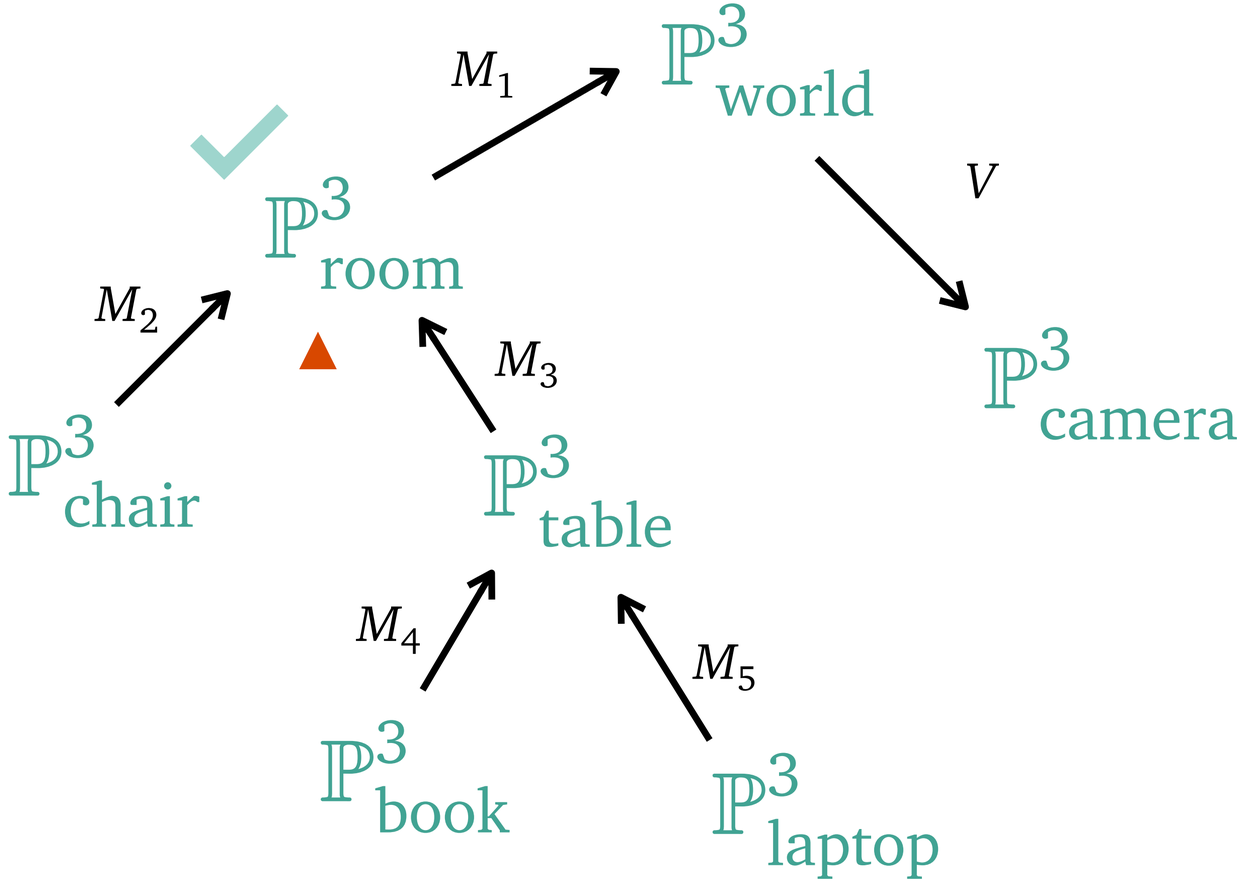




Matrix Stack

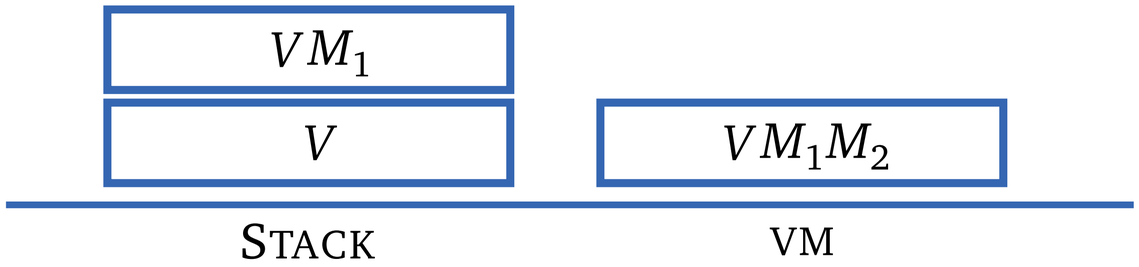
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$

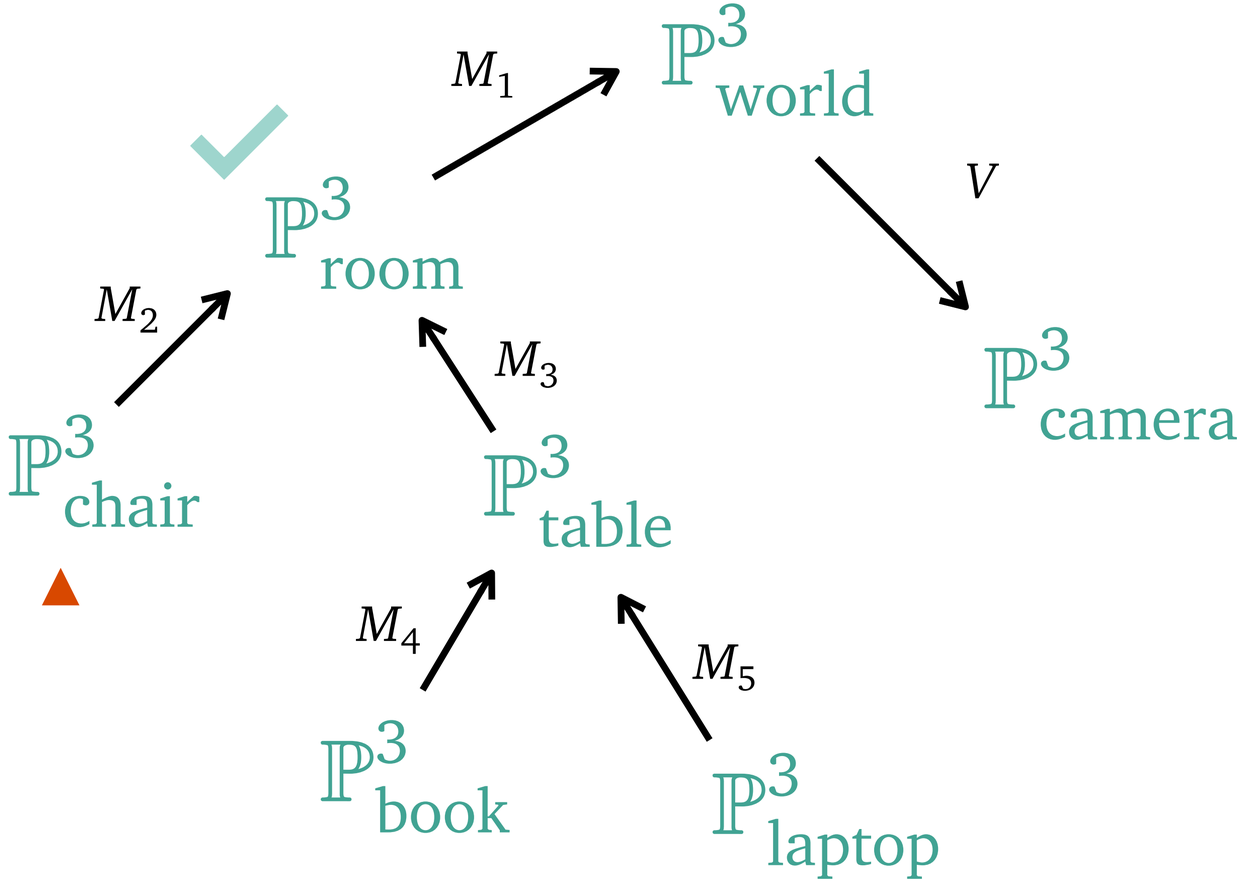




Matrix Stack

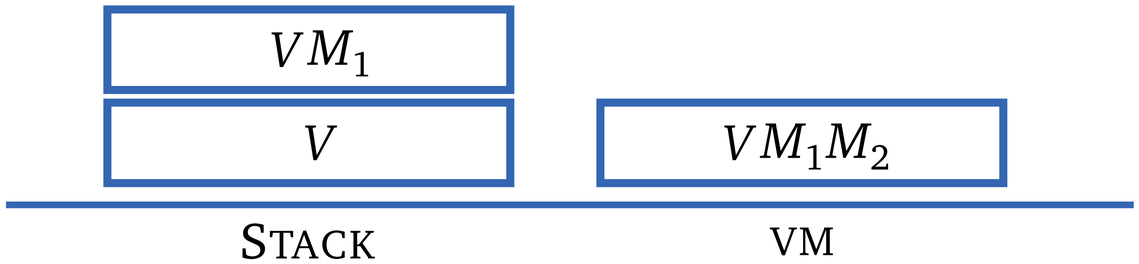
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$

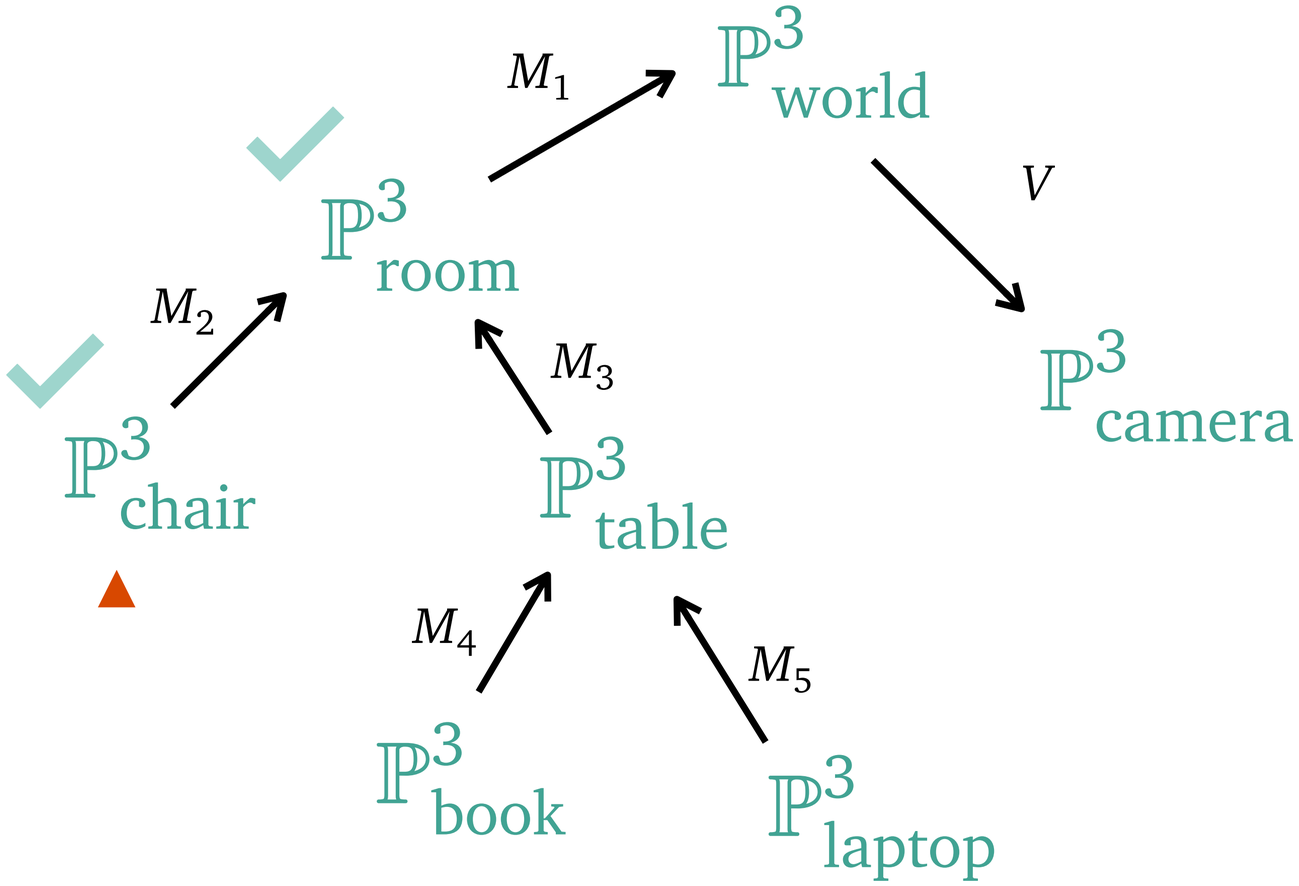




Matrix Stack

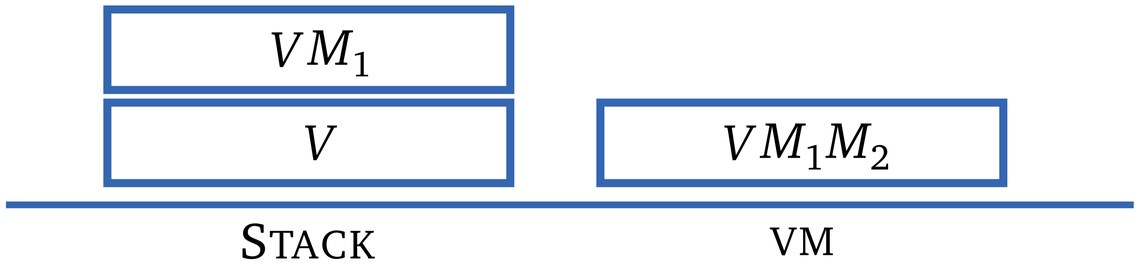
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$
- drawChair

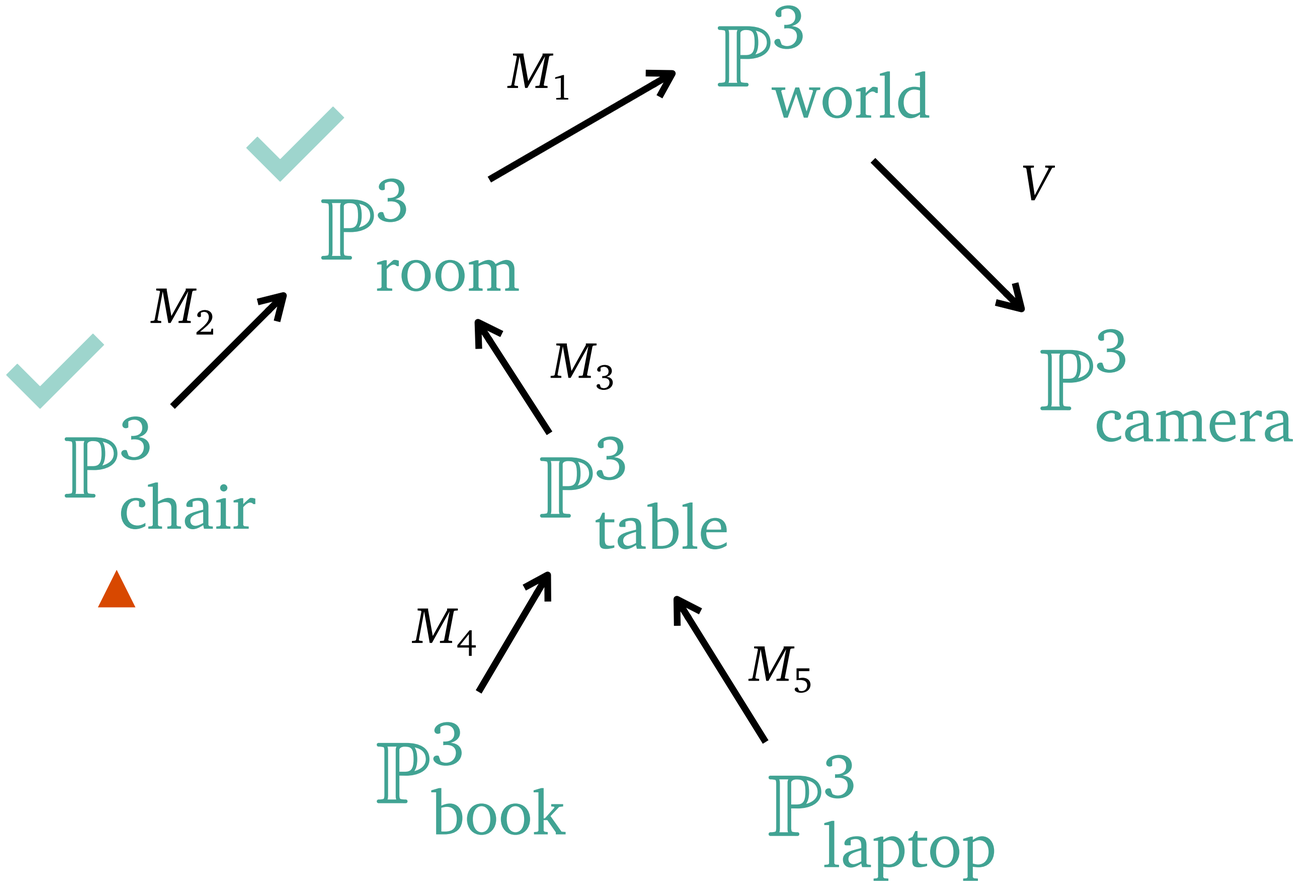




Matrix Stack

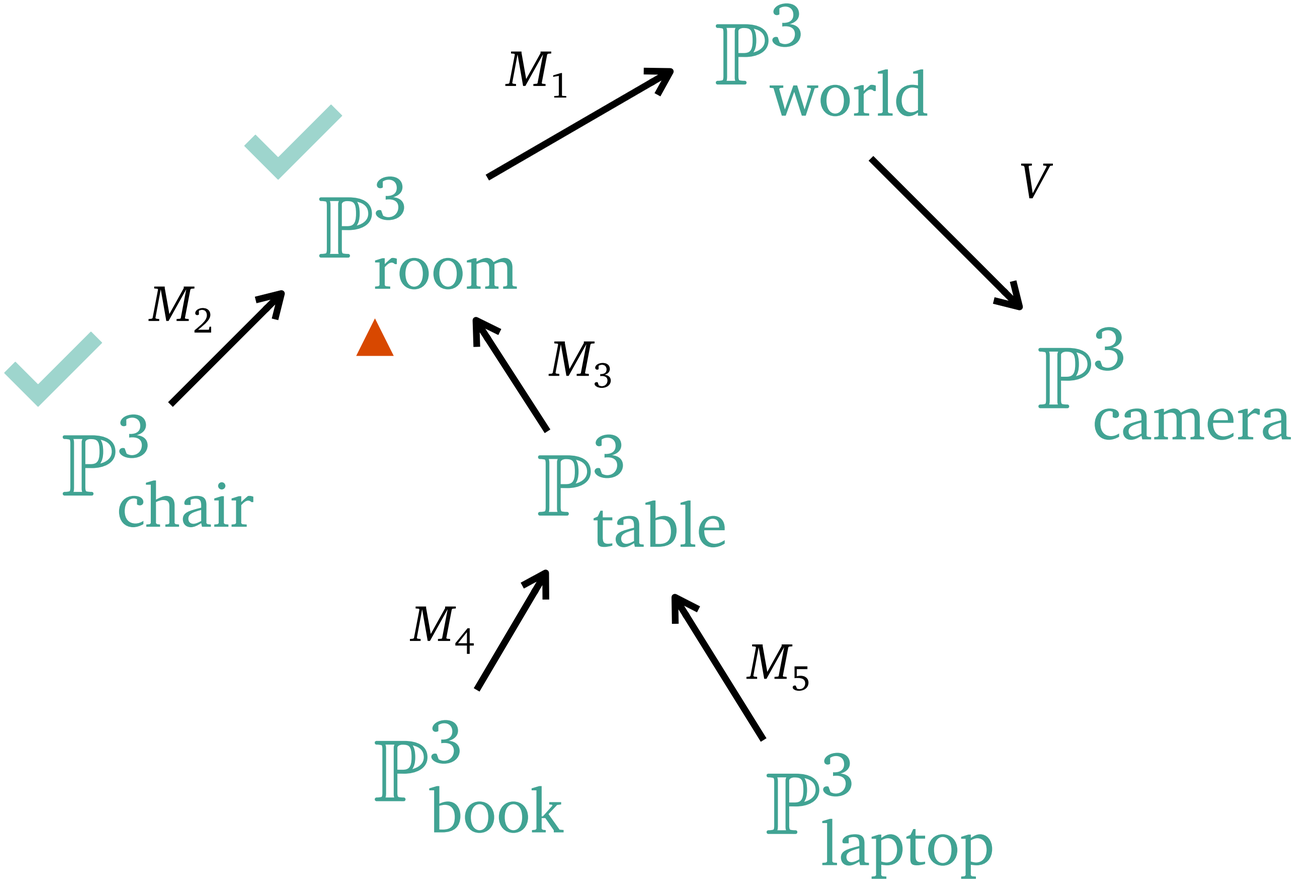
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$
- drawChair
- Pop $\text{VM} = \text{STACK.POP}$

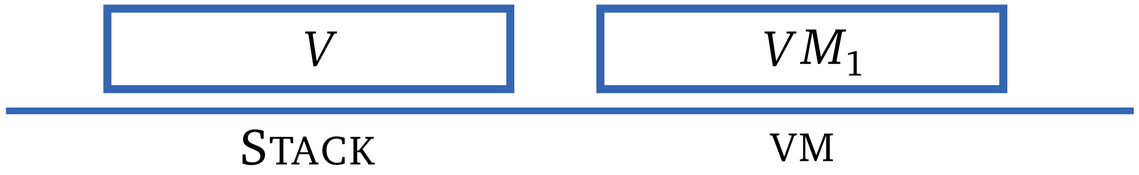




Matrix Stack

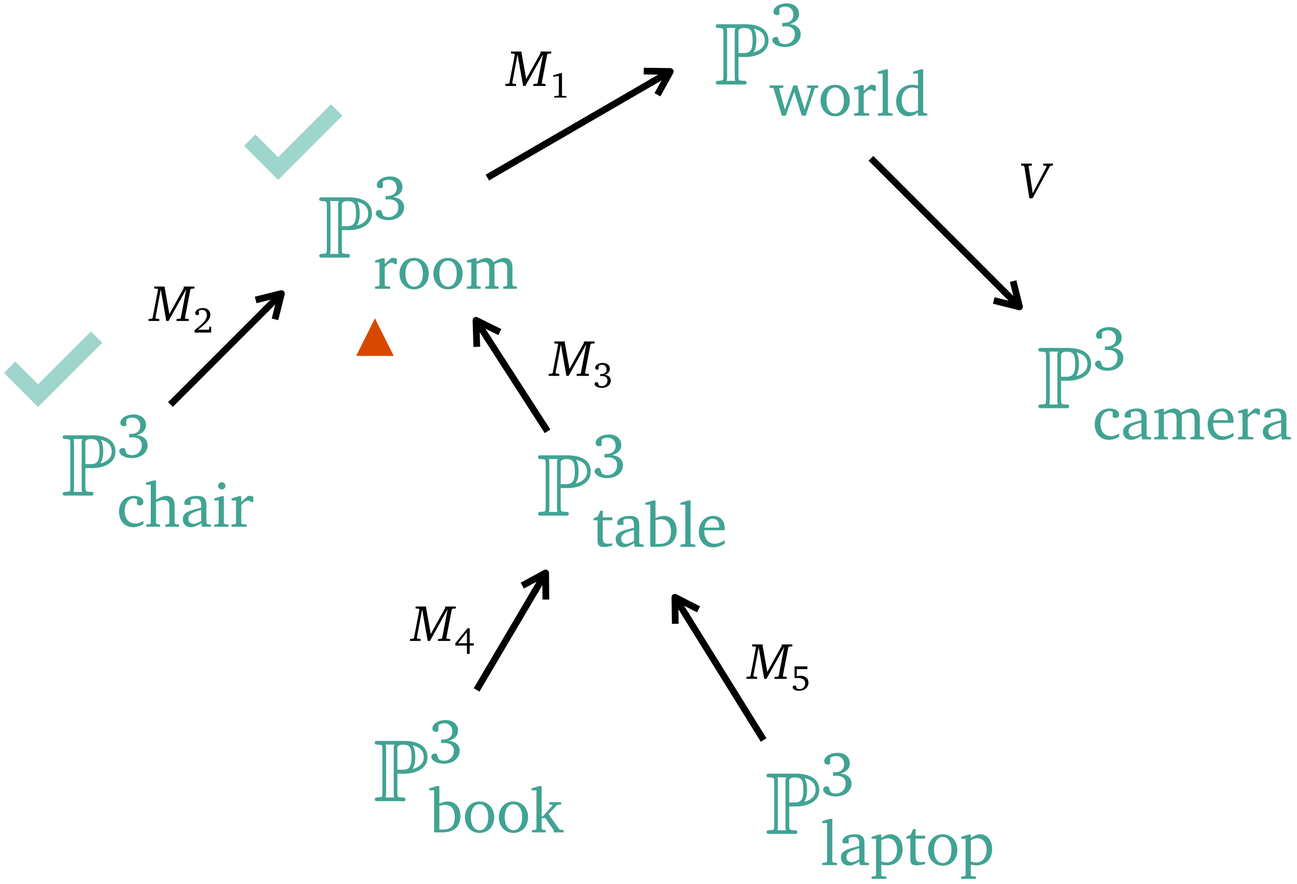
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$
- drawChair
- Pop $\text{VM} = \text{STACK.POP}$

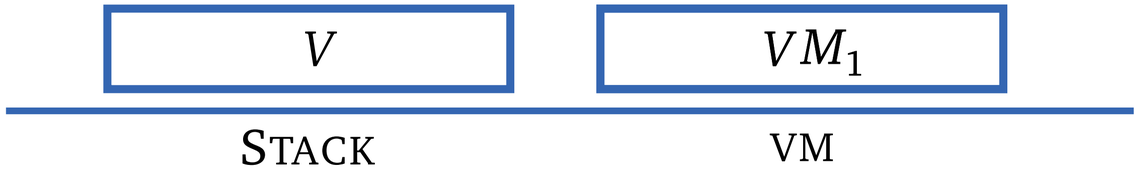




Matrix Stack

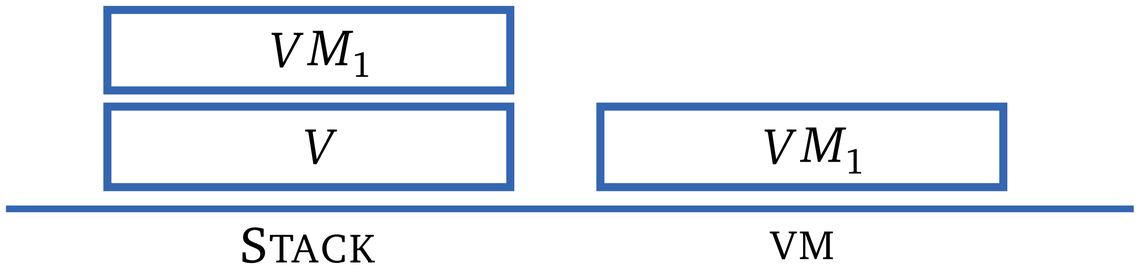
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$
- drawChair
- Pop $\text{VM} = \text{STACK.POP}$
- Push $\text{STACK.PUSH}(\text{VM})$

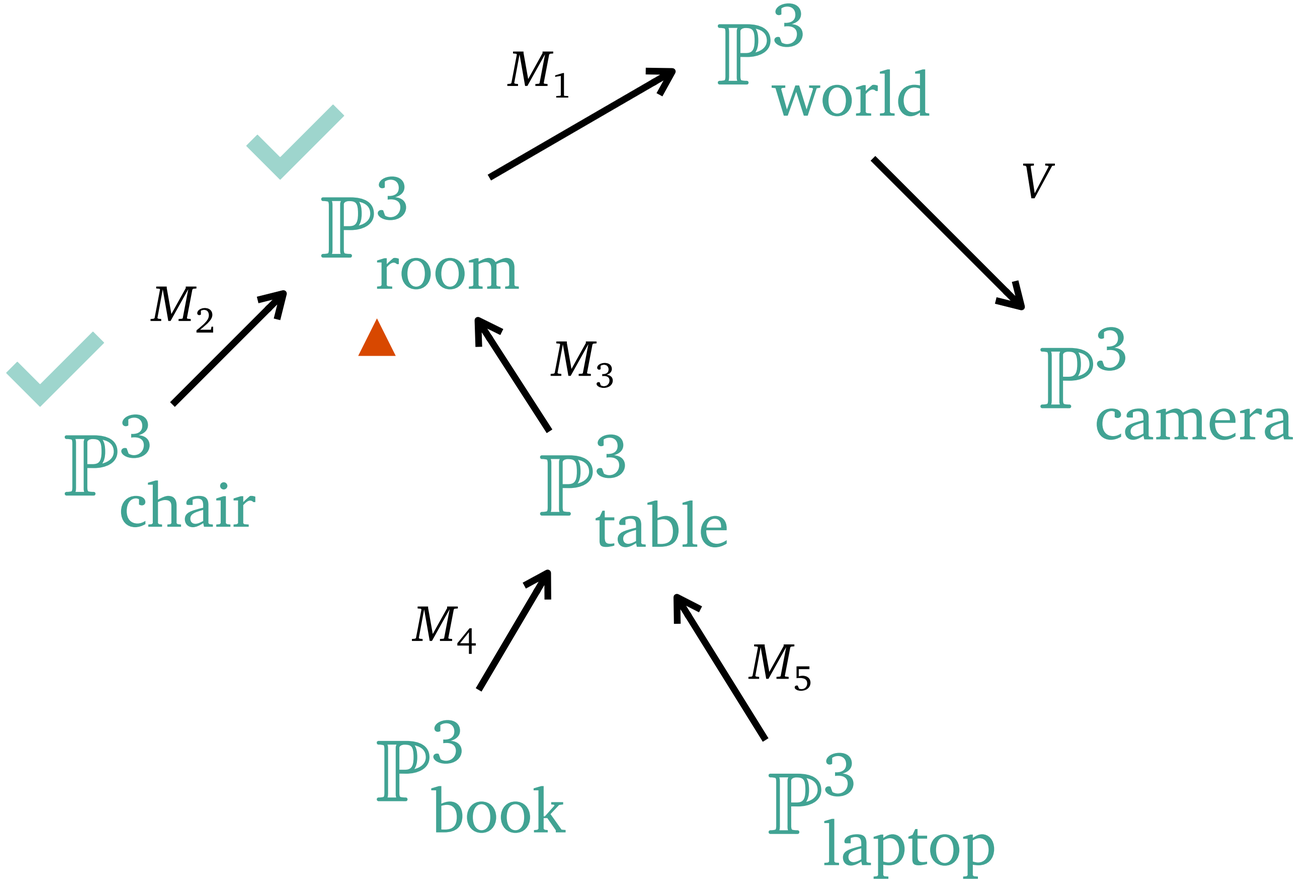




Matrix Stack

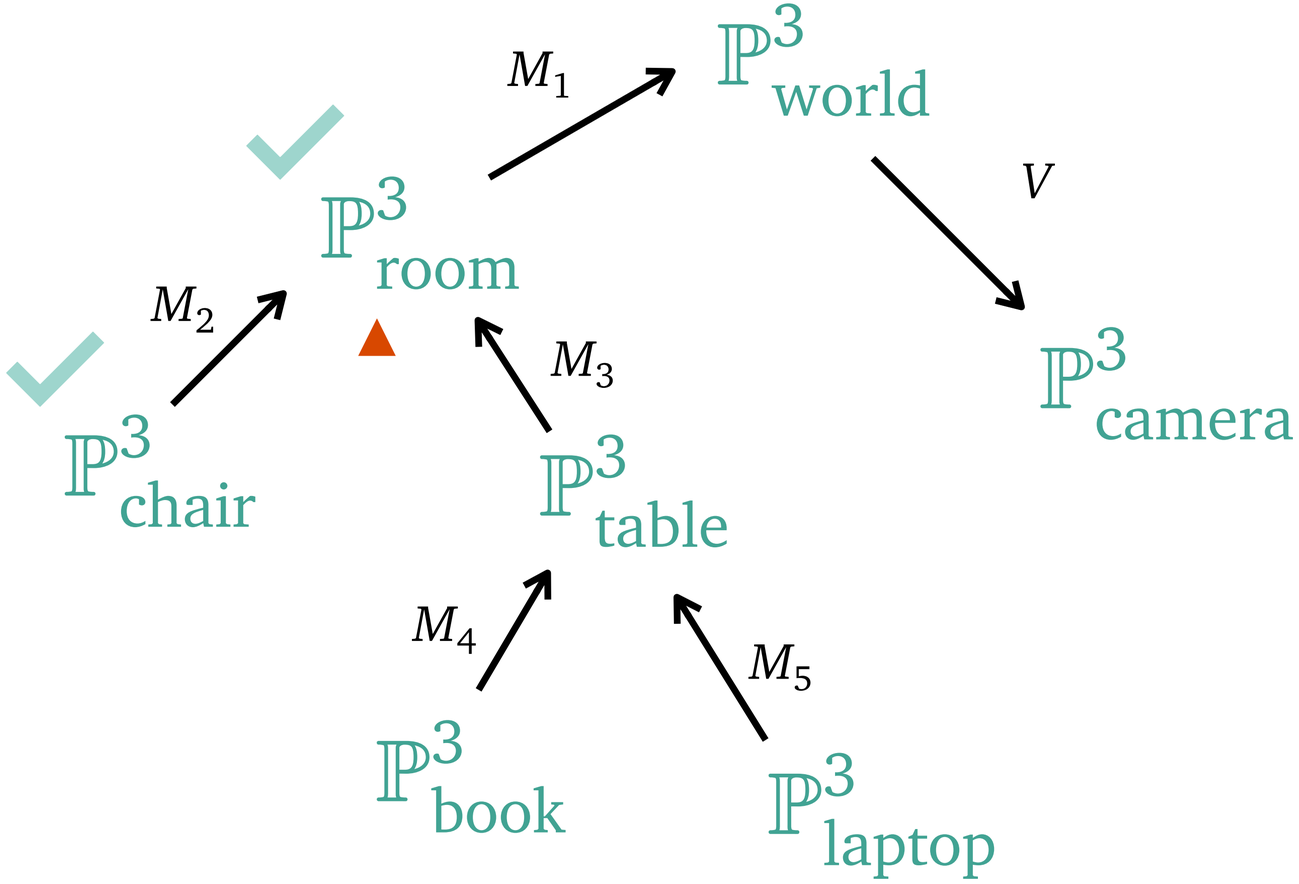
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$
- drawChair
- Pop $\text{VM} = \text{STACK.POP}$
- Push $\text{STACK.PUSH}(\text{VM})$

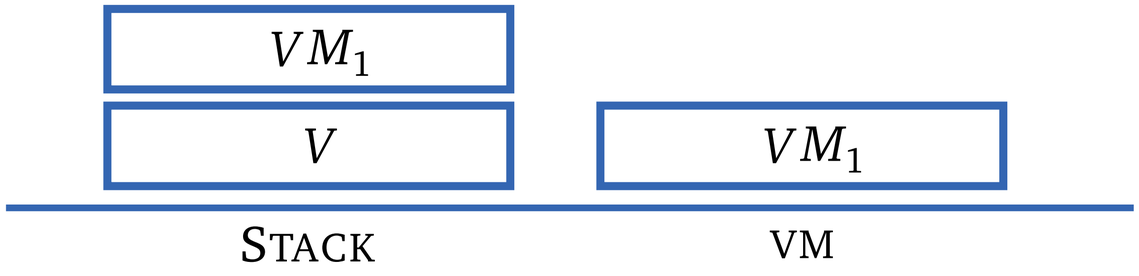




Matrix Stack

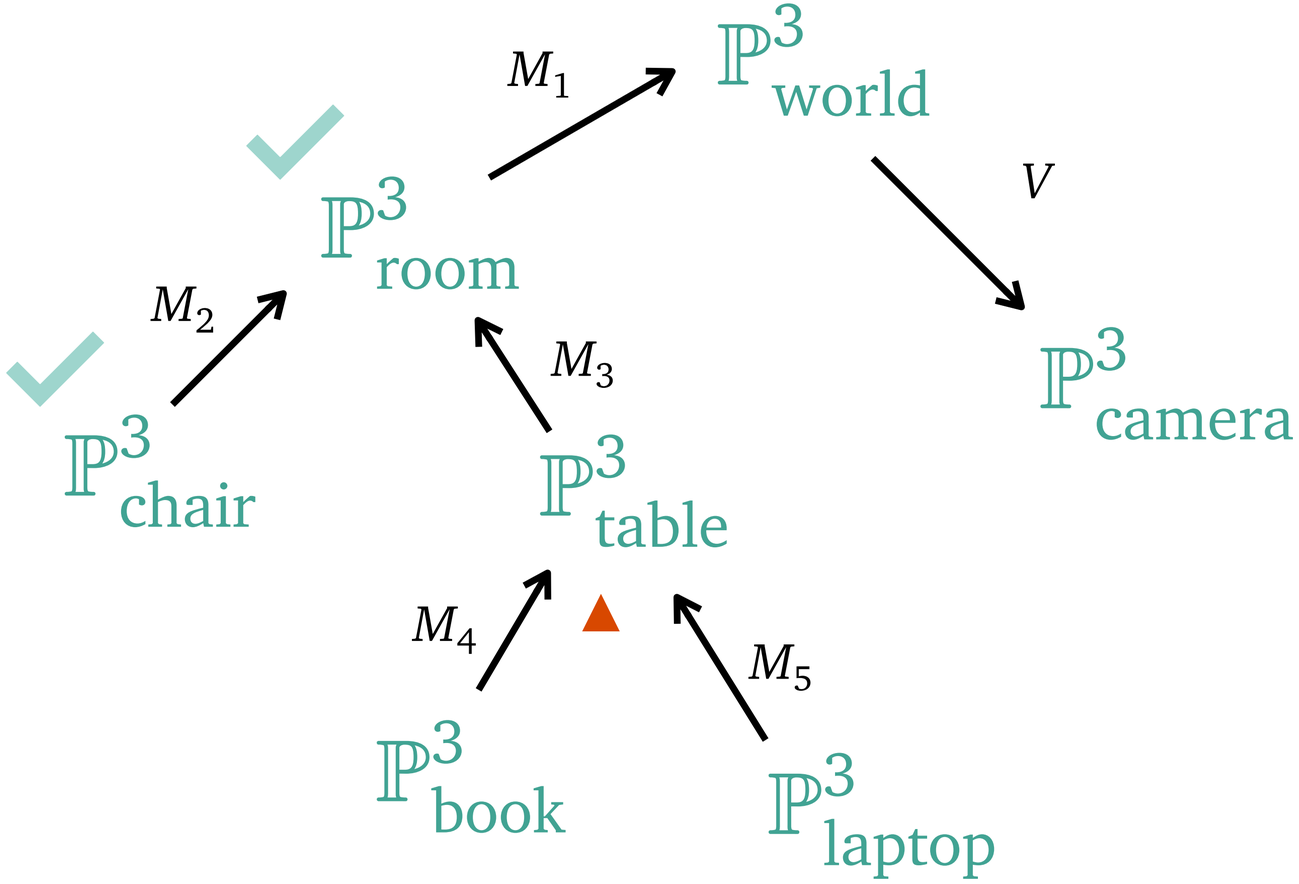
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$
- drawChair
- Pop $\text{VM} = \text{STACK.POP}$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_3$

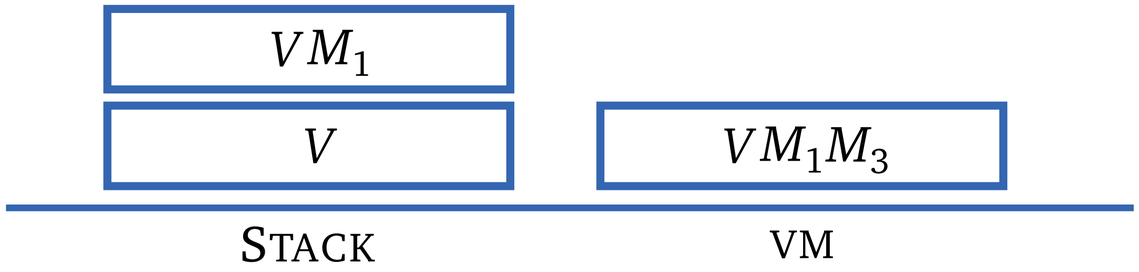




Matrix Stack

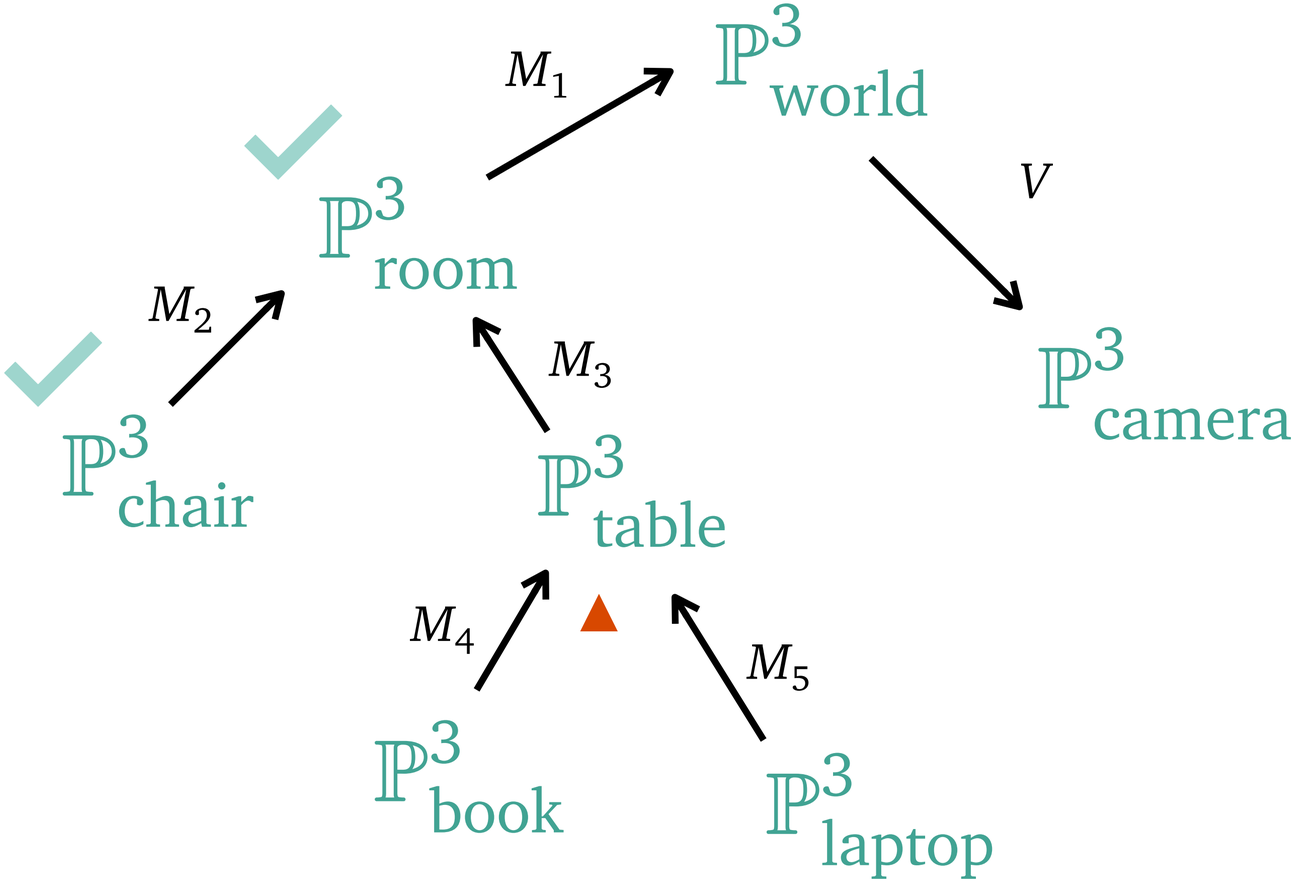
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$
- drawChair
- Pop $\text{VM} = \text{STACK.POP}$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_3$

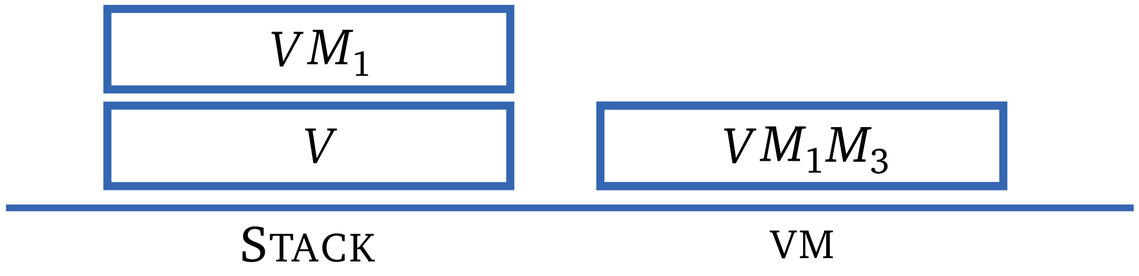




Matrix Stack

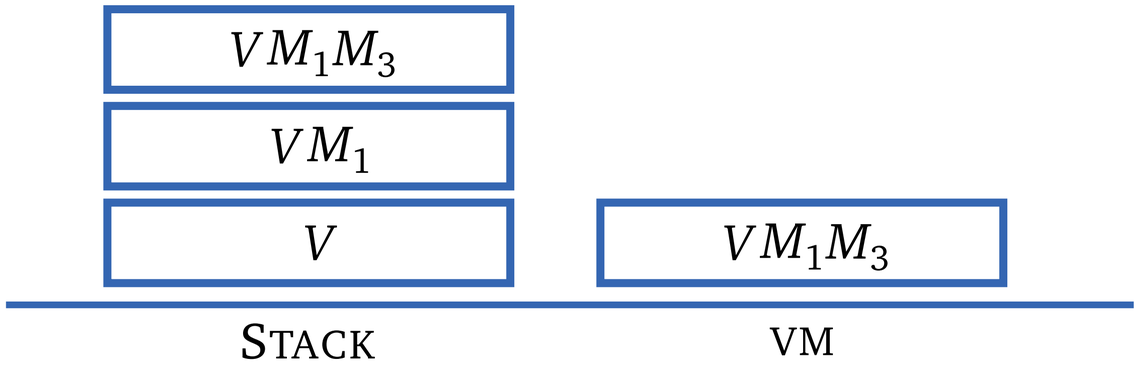
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$
- drawChair
- Pop $\text{VM} = \text{STACK.POP}$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_3$
- Push $\text{STACK.PUSH}(\text{VM})$

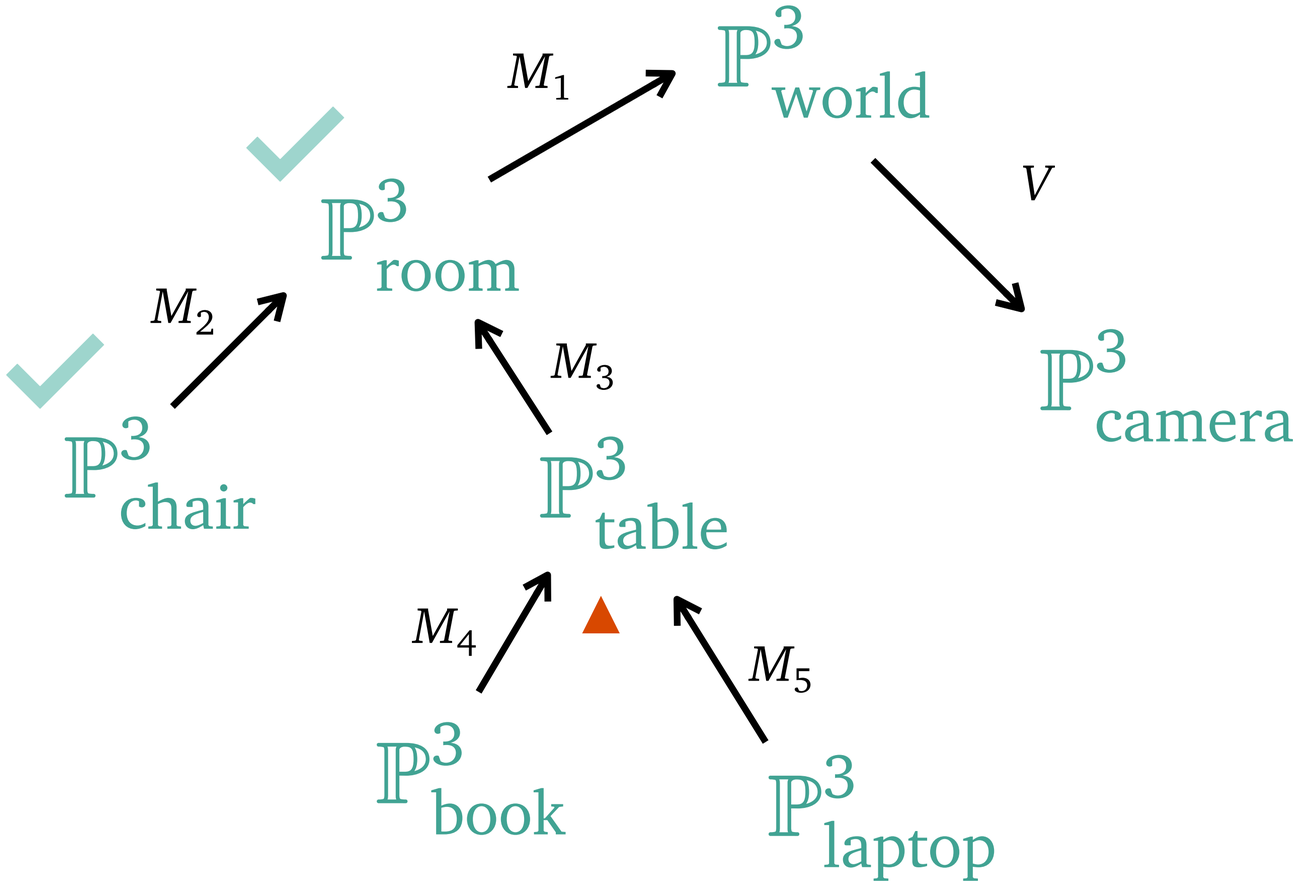




Matrix Stack

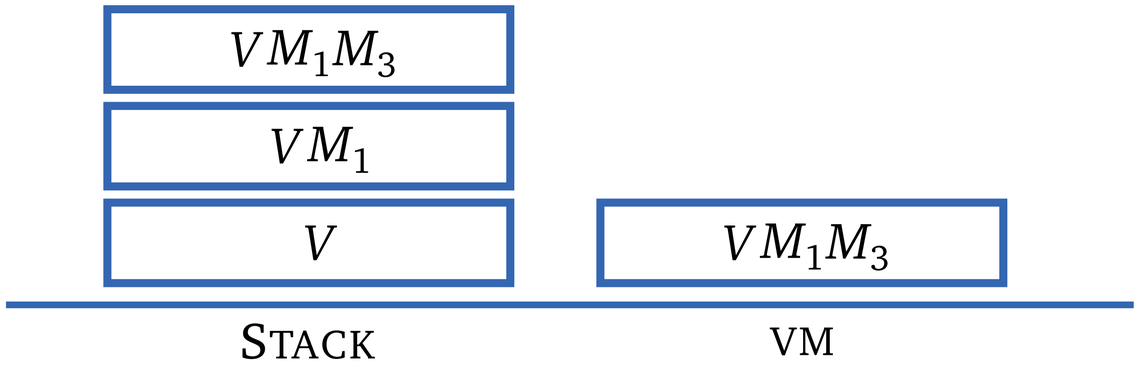
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$
- drawChair
- Pop $\text{VM} = \text{STACK.POP}$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_3$
- Push $\text{STACK.PUSH}(\text{VM})$

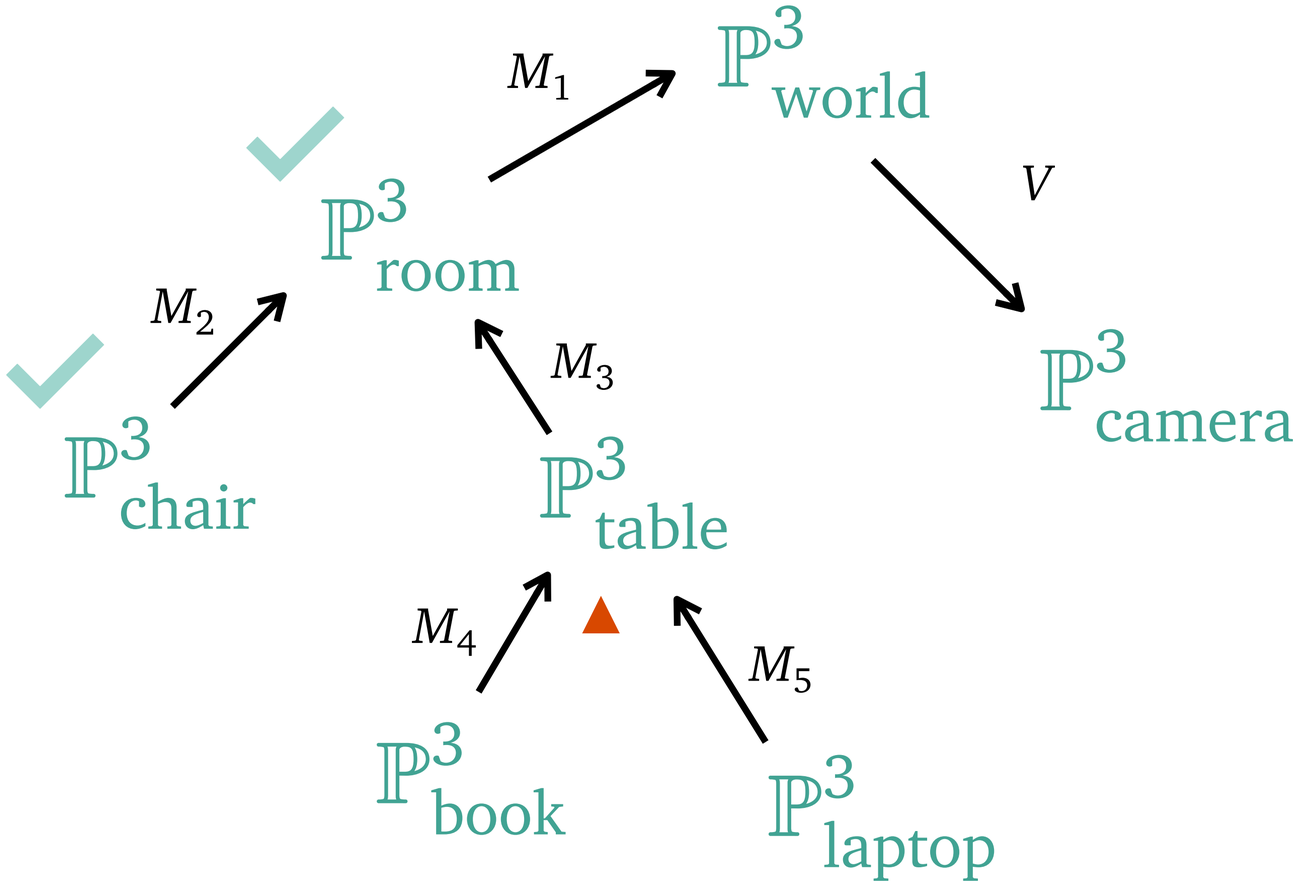




Matrix Stack

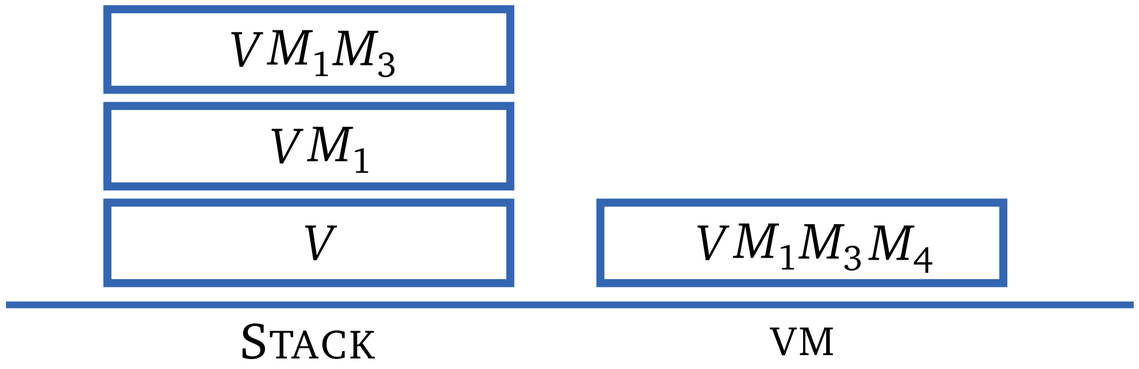
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$
- drawChair
- Pop $\text{VM} = \text{STACK.POP}$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_3$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_4$

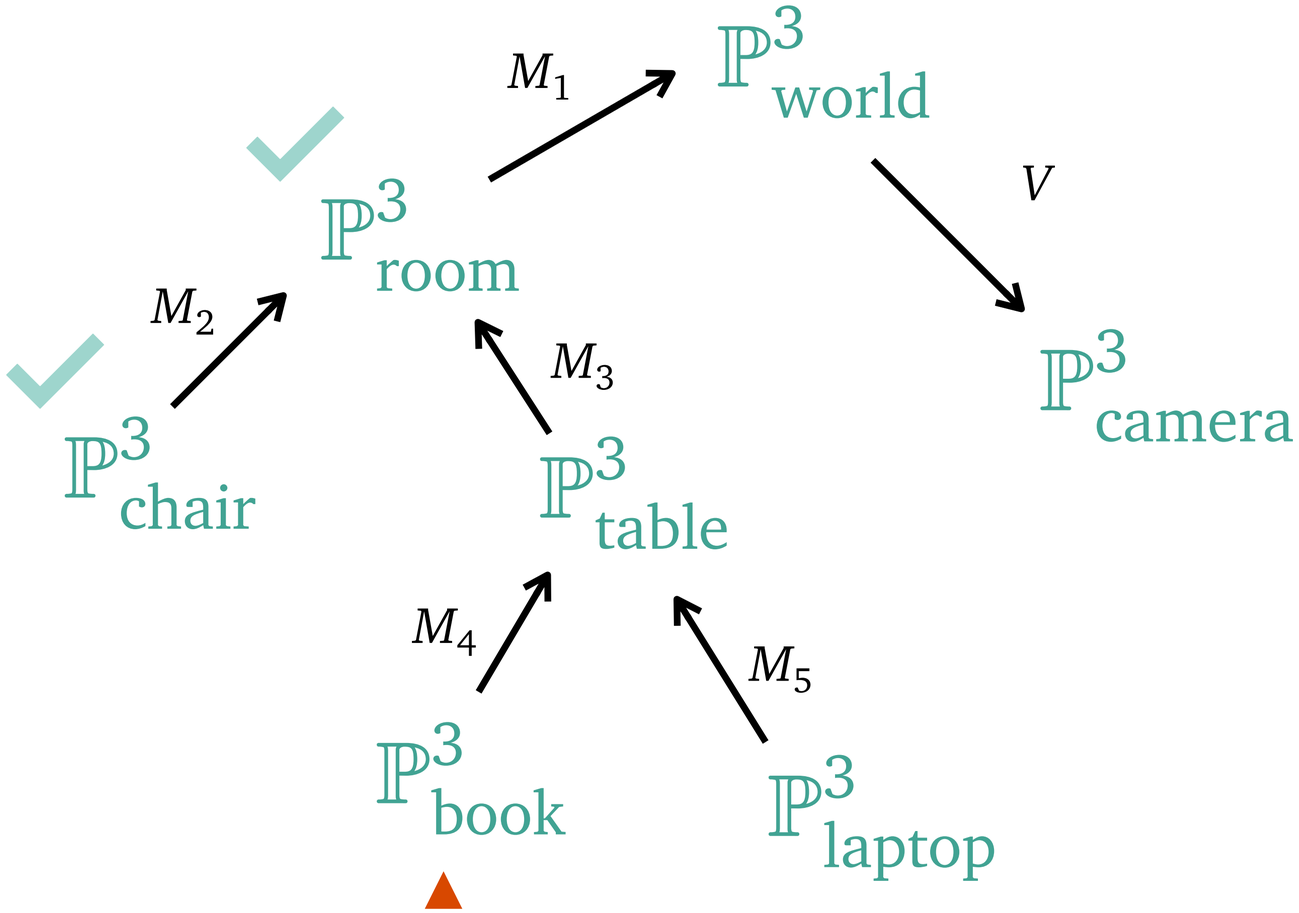




Matrix Stack

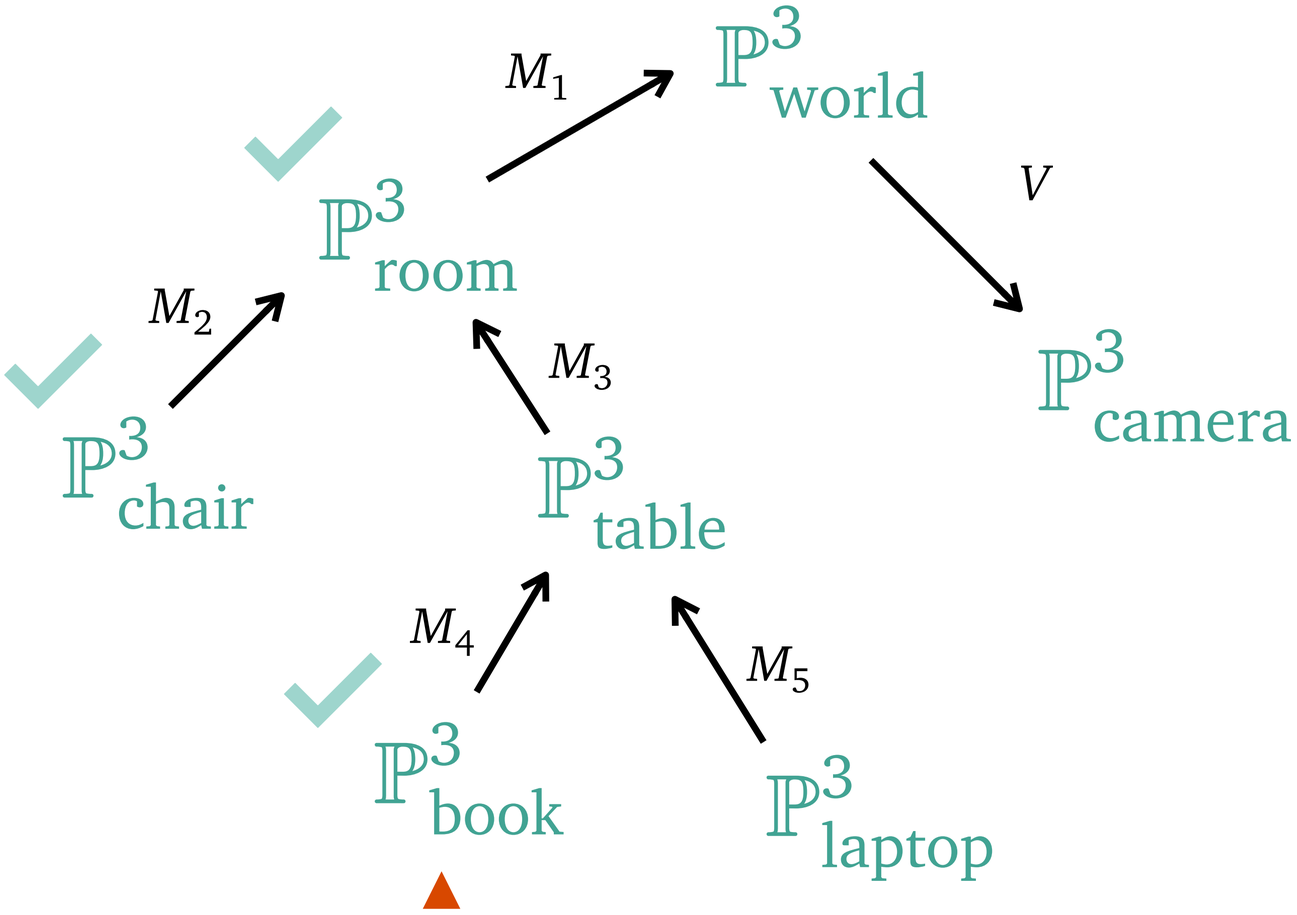
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$
- drawChair
- Pop $\text{VM} = \text{STACK.POP}$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_3$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_4$

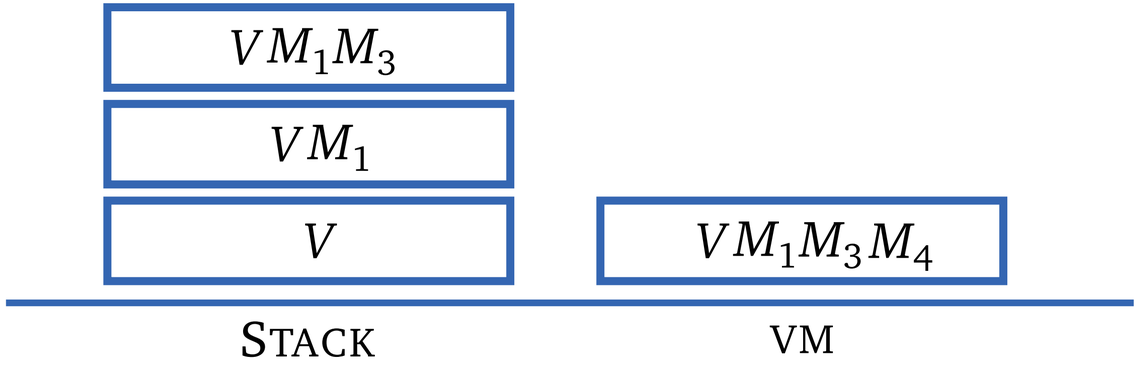




Matrix Stack

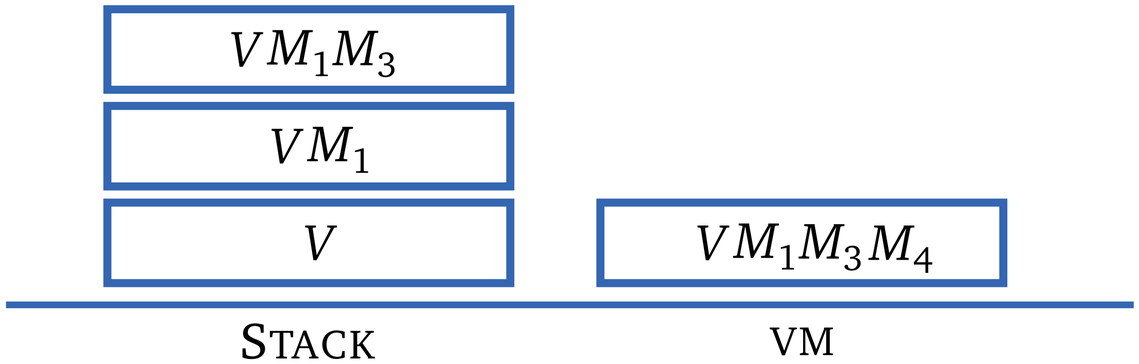
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$
- drawChair
- Pop $\text{VM} = \text{STACK.POP}$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_3$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_4$
- drawBook

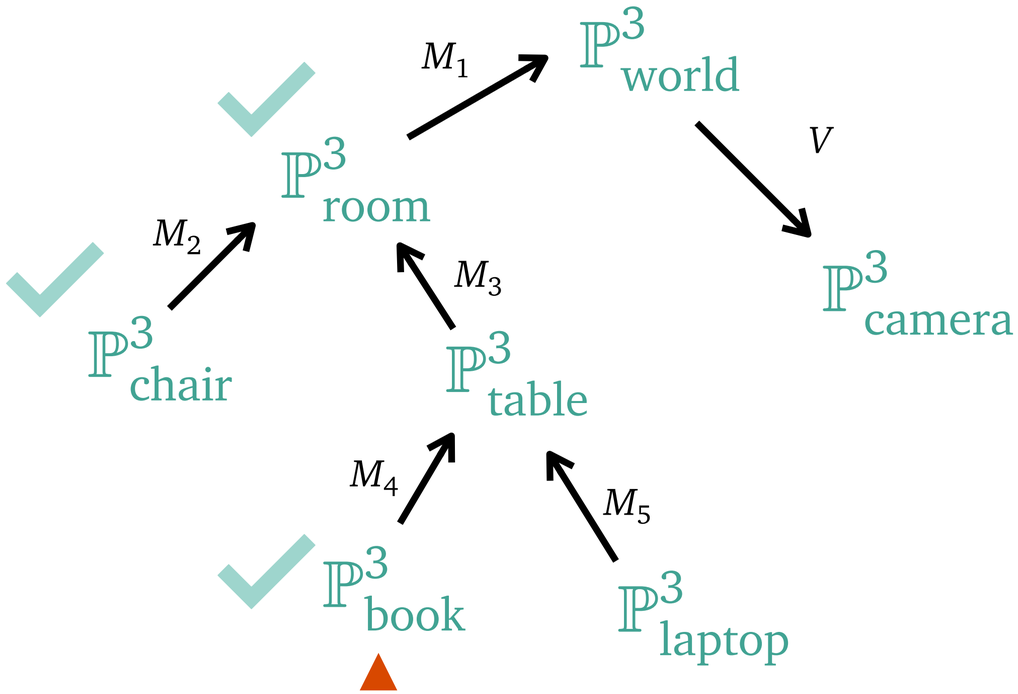




Matrix Stack

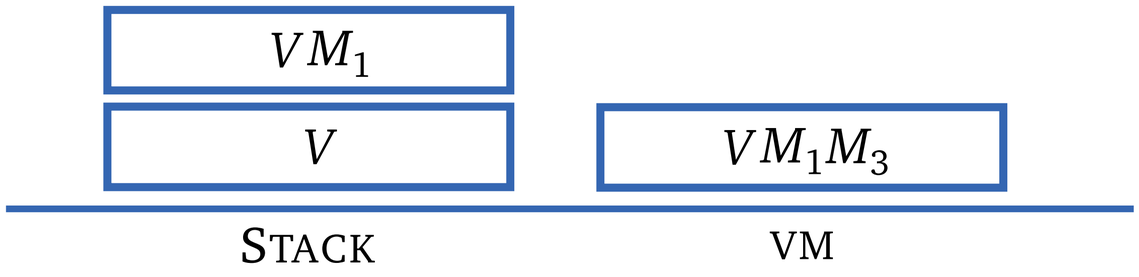
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$
- drawChair
- Pop $\text{VM} = \text{STACK.POP}$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_3$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_4$
- drawBook
- Pop $\text{VM} = \text{STACK.POP}$

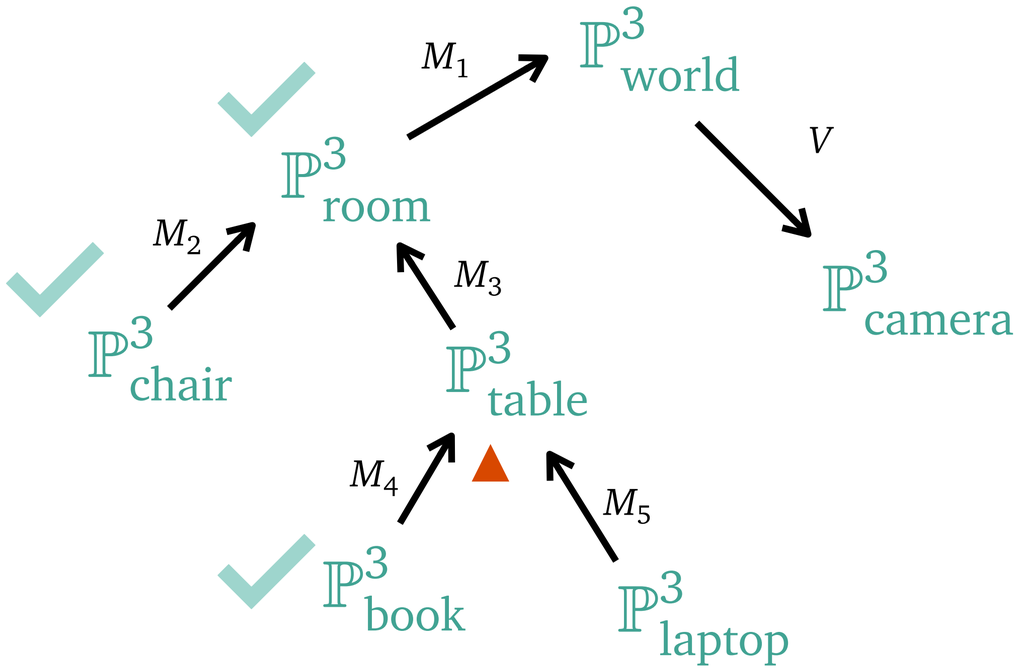




Matrix Stack

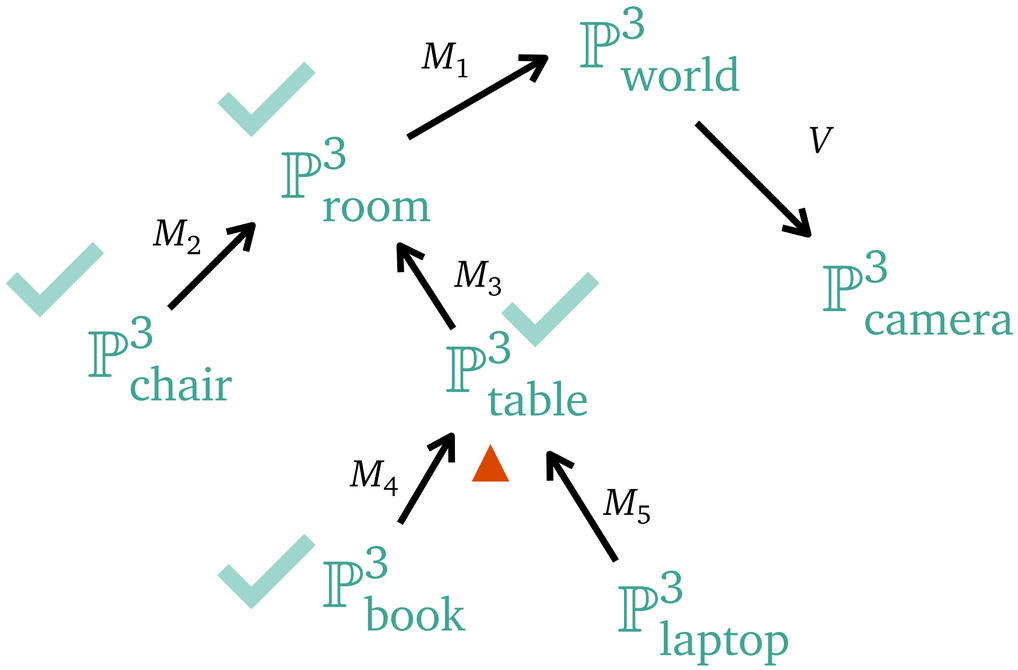
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$
- drawChair
- Pop $\text{VM} = \text{STACK.POP}$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_3$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_4$
- drawBook
- Pop $\text{VM} = \text{STACK.POP}$

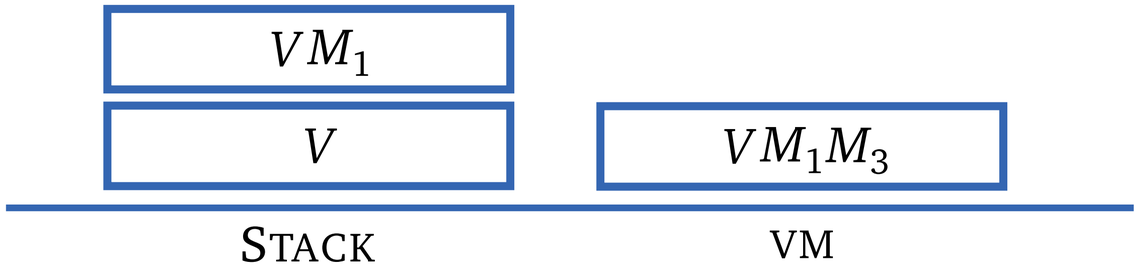




Matrix Stack

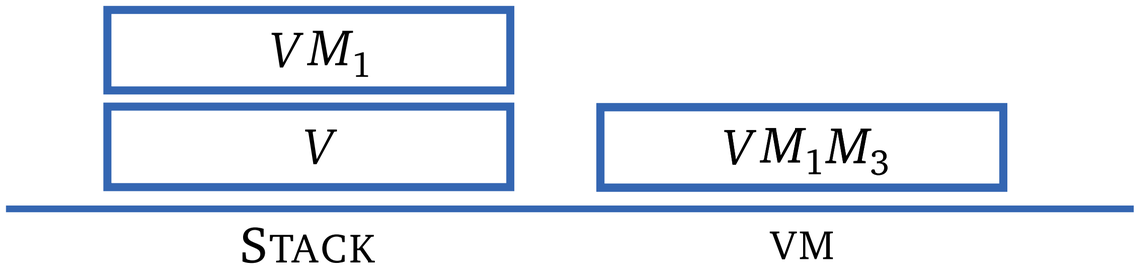
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$
- drawChair
- Pop $\text{VM} = \text{STACK.POP}$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_3$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_4$
- drawBook
- Pop $\text{VM} = \text{STACK.POP}$
- drawTable

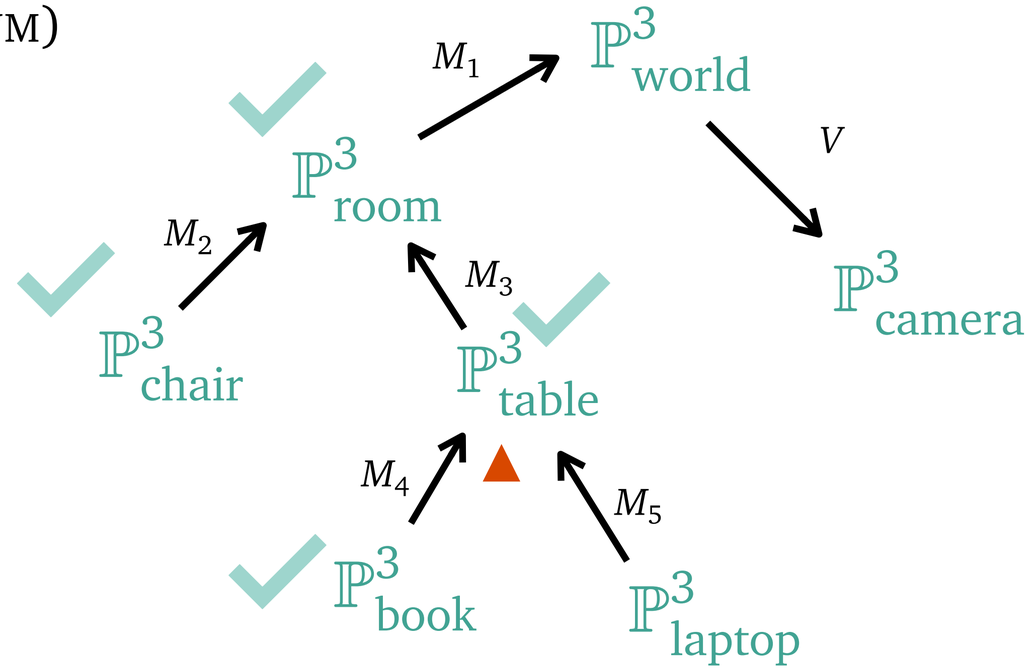




Matrix Stack

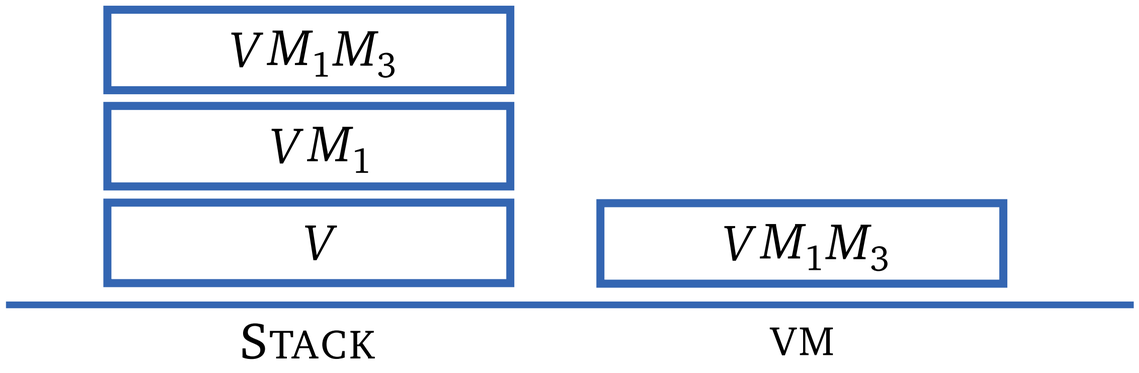
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$
- drawChair
- Pop $\text{VM} = \text{STACK.POP}$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_3$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_4$
- drawBook
- Pop $\text{VM} = \text{STACK.POP}$
- drawTable
- Push $\text{STACK.PUSH}(\text{VM})$

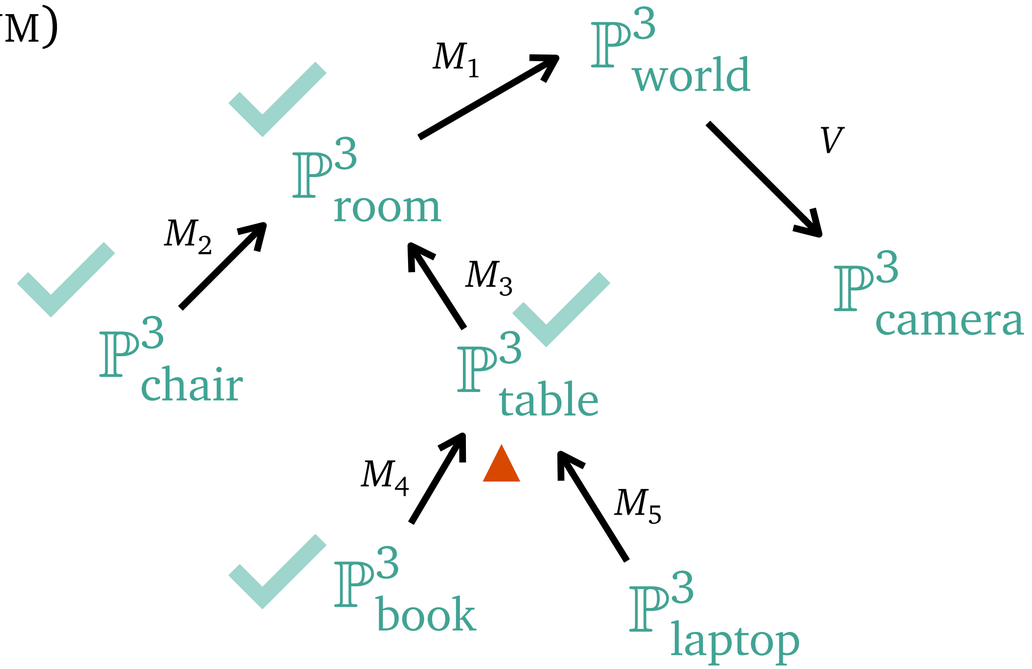




Matrix Stack

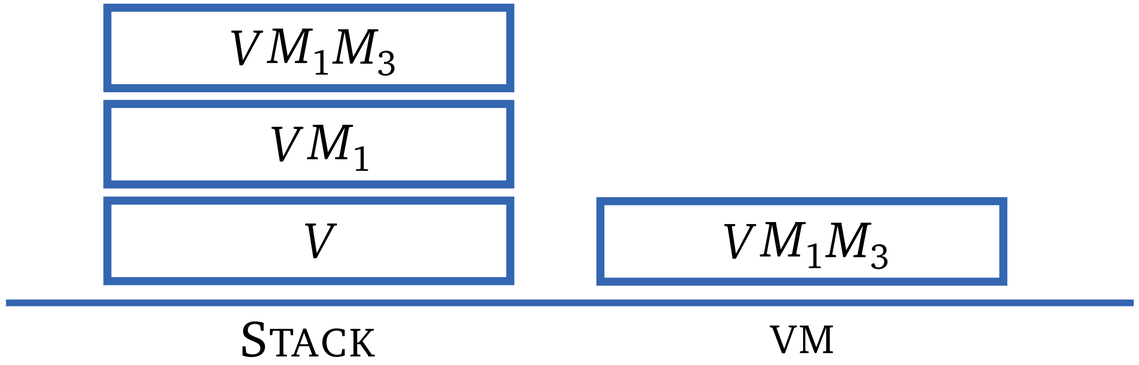
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$
- drawChair
- Pop $\text{VM} = \text{STACK.POP}$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_3$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_4$
- drawBook
- Pop $\text{VM} = \text{STACK.POP}$
- drawTable
- Push $\text{STACK.PUSH}(\text{VM})$

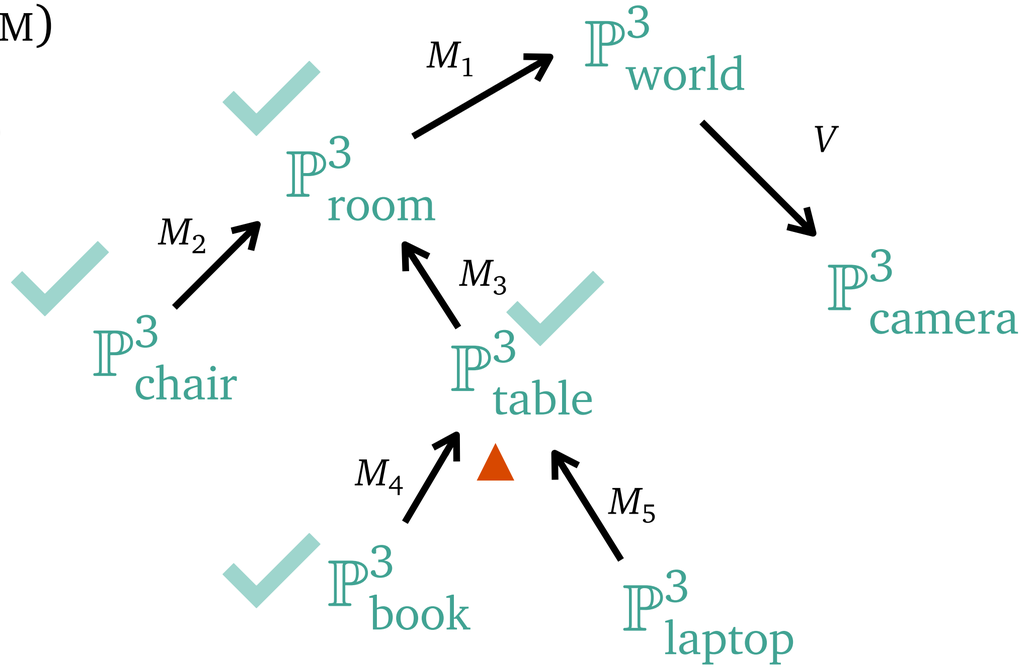




Matrix Stack

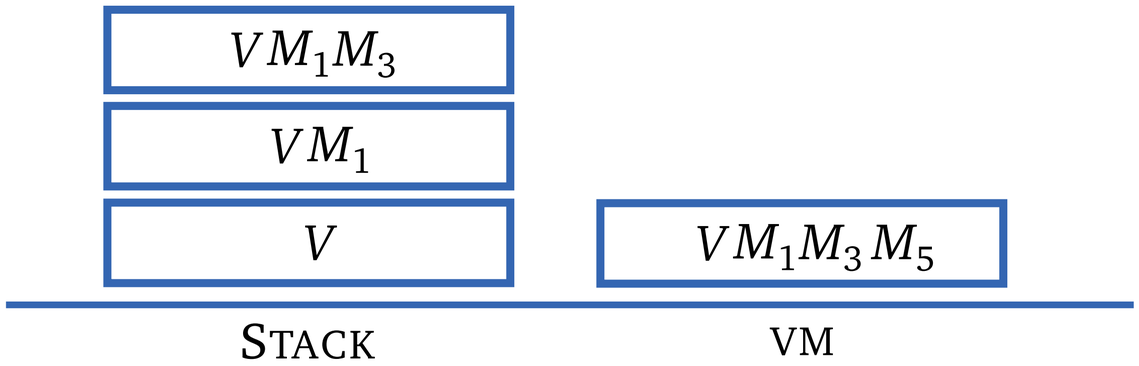
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$
- drawChair
- Pop $\text{VM} = \text{STACK.POP}$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_3$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_4$
- drawBook
- Pop $\text{VM} = \text{STACK.POP}$
- drawTable
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_5$

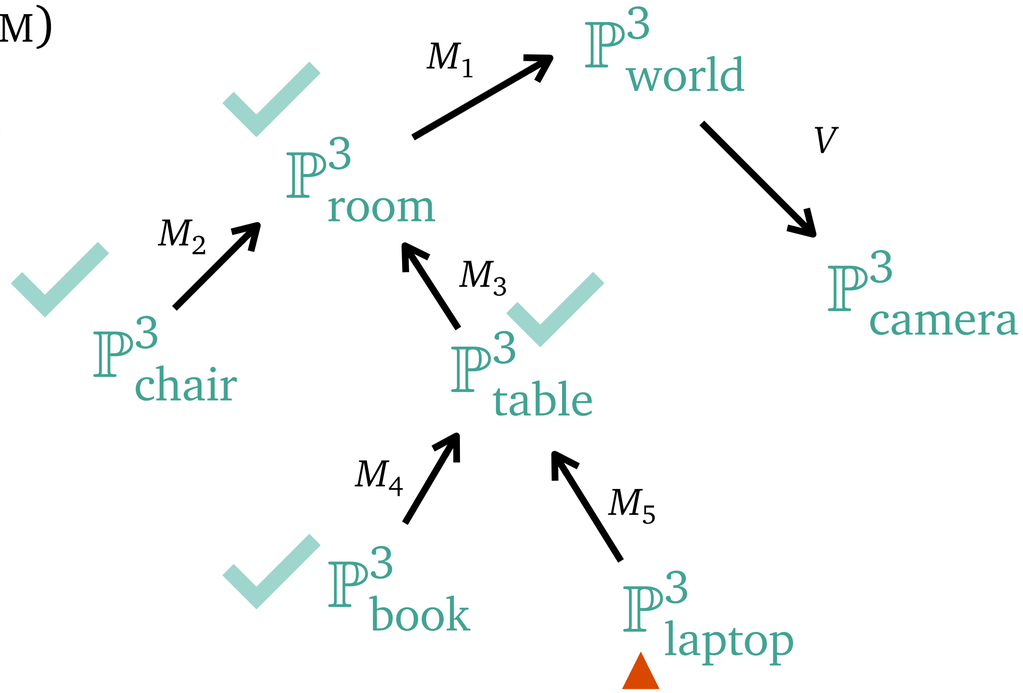




Matrix Stack

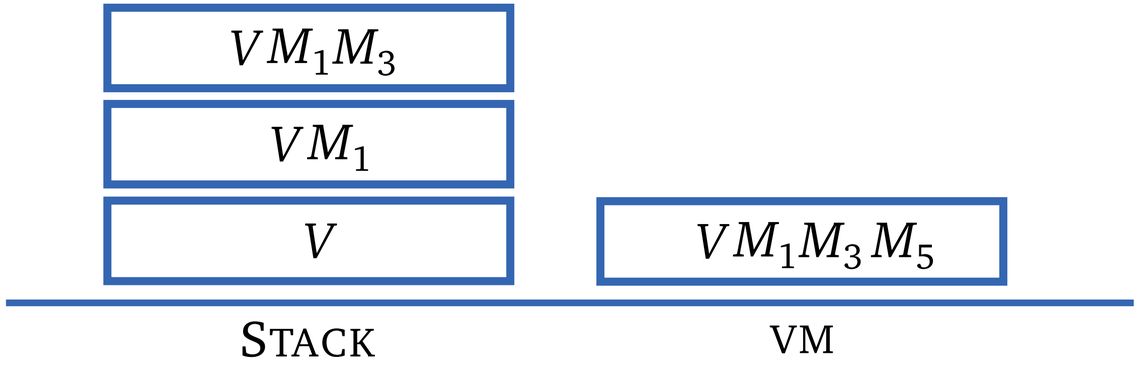
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$
- drawChair
- Pop $\text{VM} = \text{STACK.POP}$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_3$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_4$
- drawBook
- Pop $\text{VM} = \text{STACK.POP}$
- drawTable
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_5$

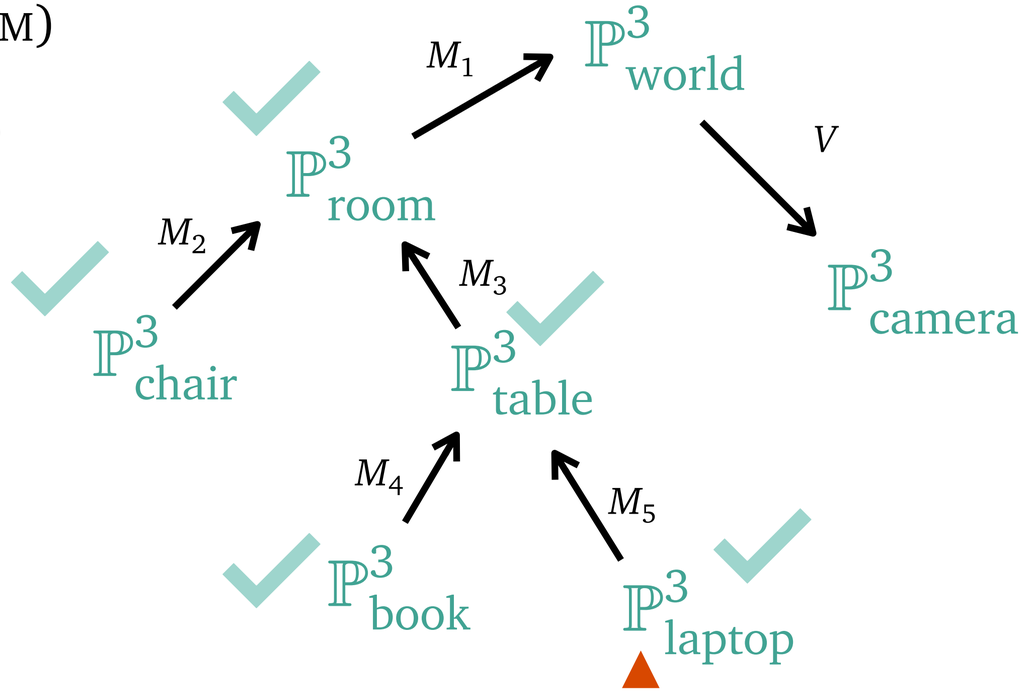




Matrix Stack

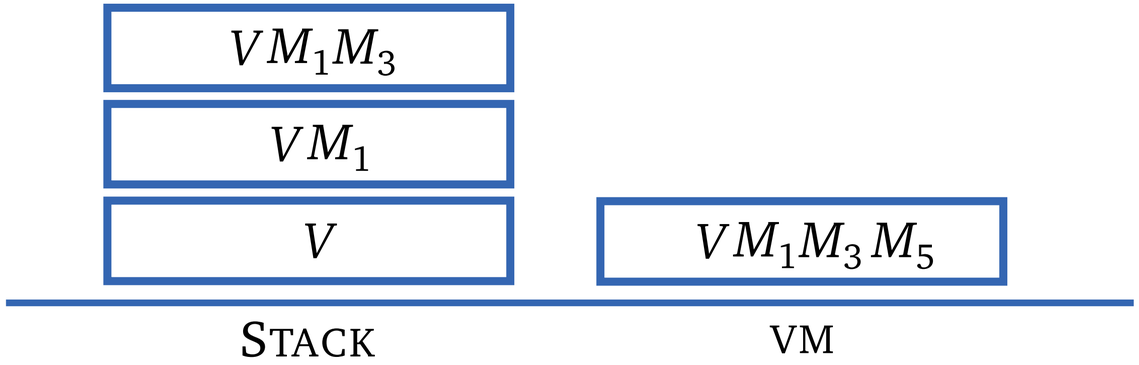
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$
- drawChair
- Pop $\text{VM} = \text{STACK.POP}$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_3$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_4$
- drawBook
- Pop $\text{VM} = \text{STACK.POP}$
- drawTable
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_5$
- drawLaptop

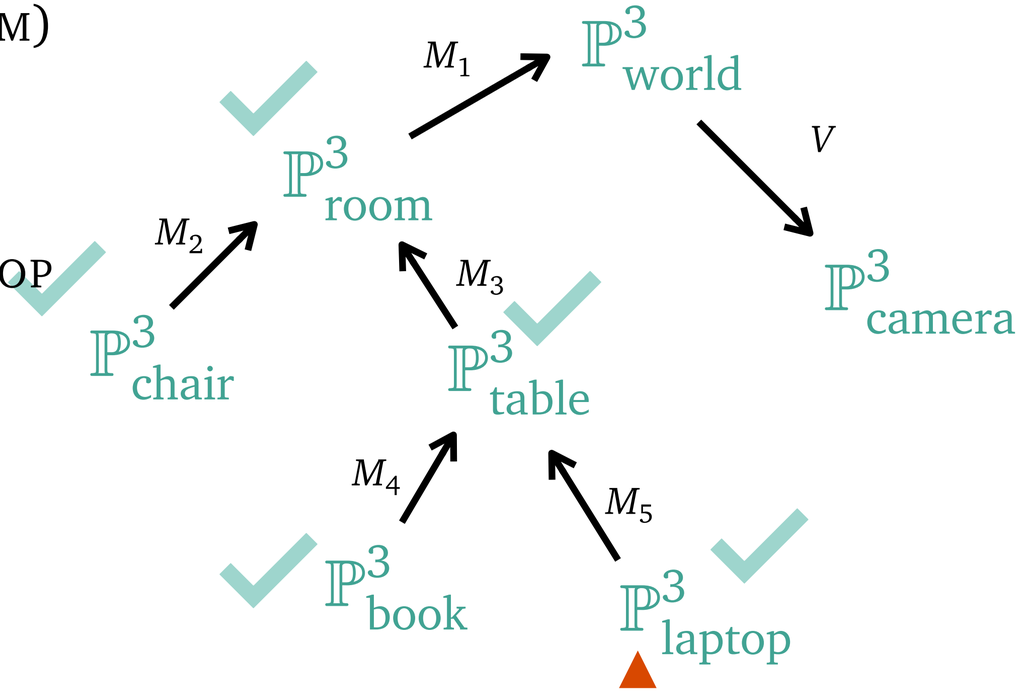




Matrix Stack

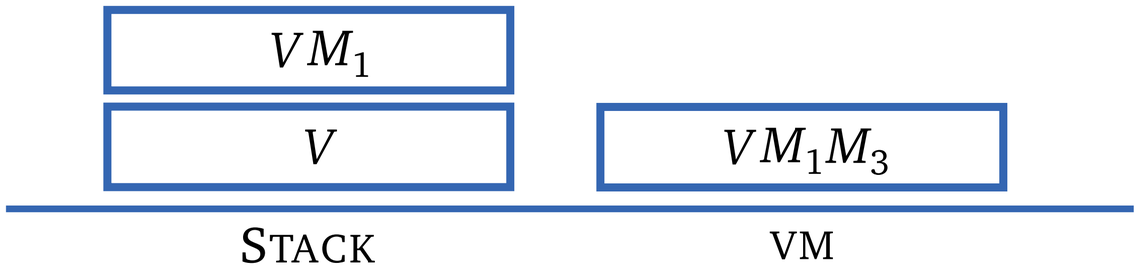
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$
- drawChair
- Pop $\text{VM} = \text{STACK.POP}$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_3$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_4$
- drawBook
- Pop $\text{VM} = \text{STACK.POP}$
- drawTable
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_5$
- drawLaptop
- Pop $\text{VM} = \text{STACK.POP}$

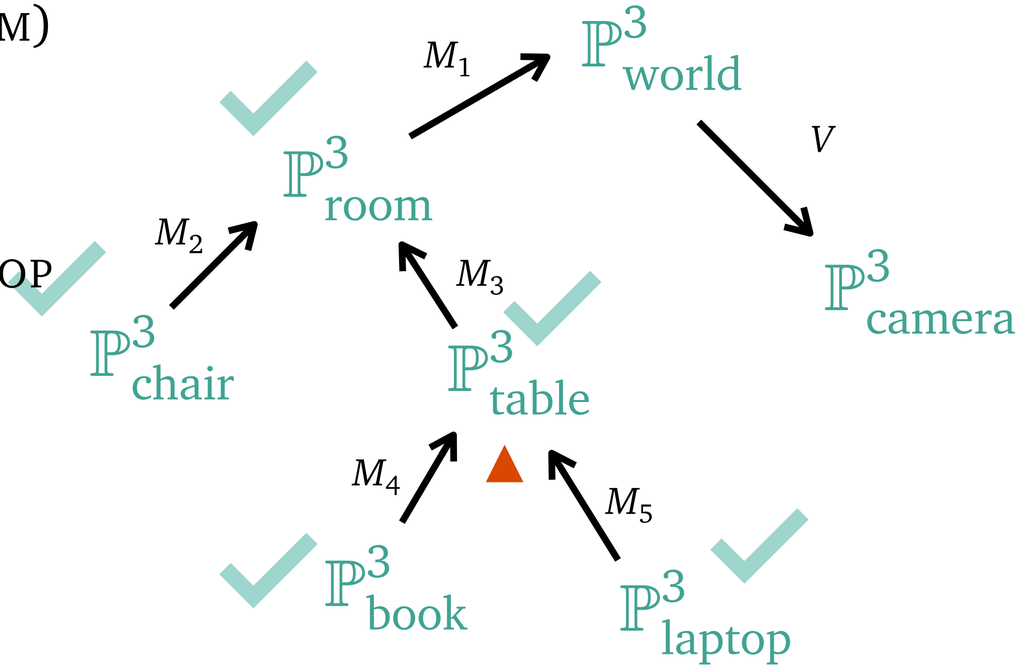




Matrix Stack

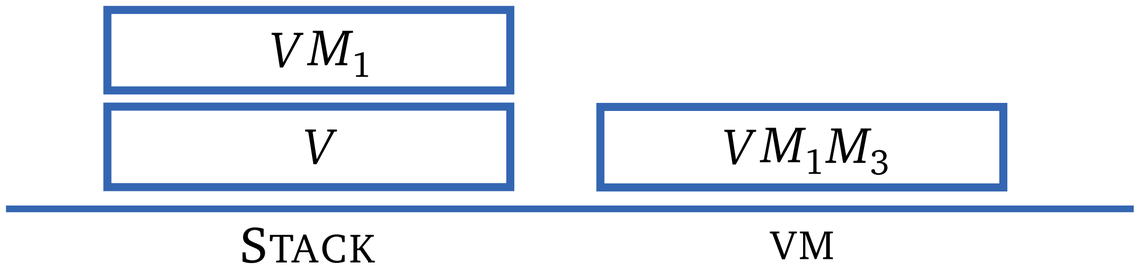
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$
- drawChair
- Pop $\text{VM} = \text{STACK.POP}$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_3$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_4$
- drawBook
- Pop $\text{VM} = \text{STACK.POP}$
- drawTable
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_5$
- drawLaptop
- Pop $\text{VM} = \text{STACK.POP}$

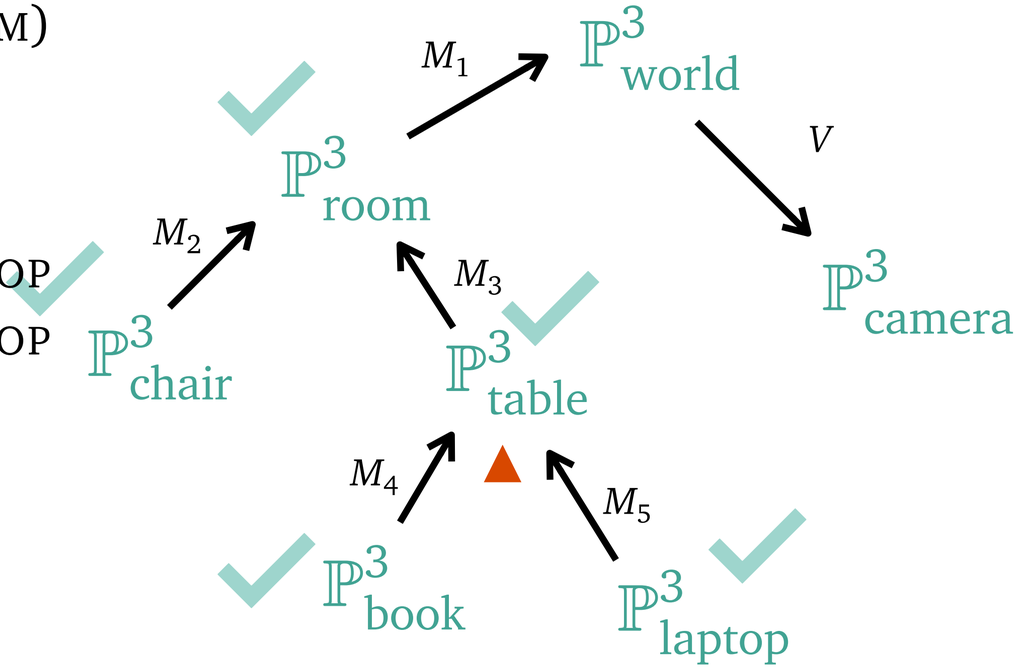




Matrix Stack

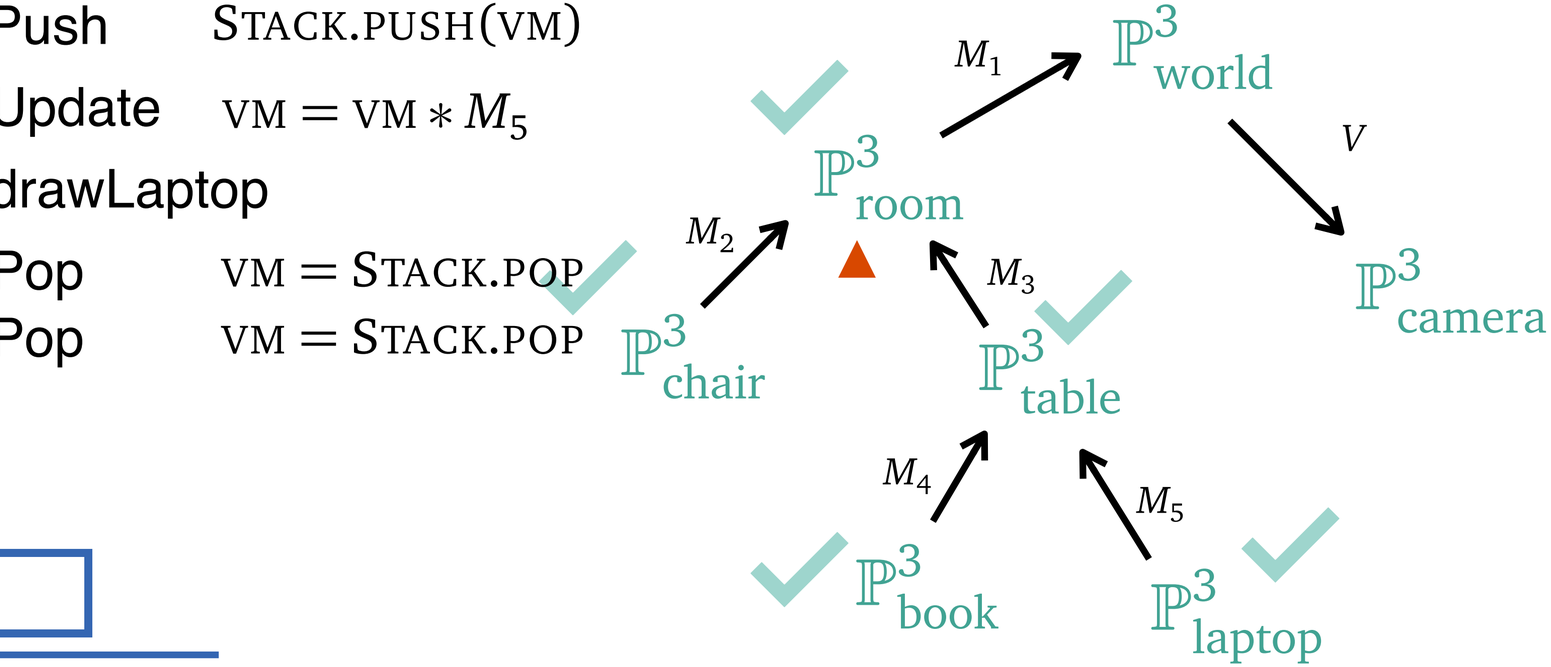
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$
- drawChair
- Pop $\text{VM} = \text{STACK.POP}$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_3$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_4$
- drawBook
- Pop $\text{VM} = \text{STACK.POP}$
- drawTable
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_5$
- drawLaptop
- Pop $\text{VM} = \text{STACK.POP}$
- Pop $\text{VM} = \text{STACK.POP}$

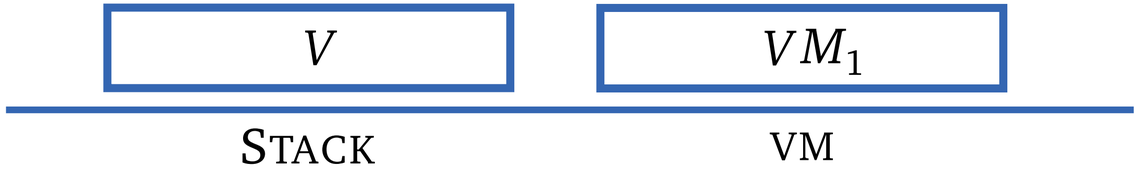




Matrix Stack

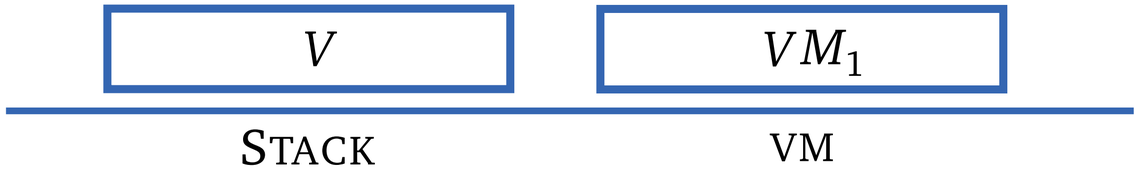
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$
- drawChair
- Pop $\text{VM} = \text{STACK.POP}$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_3$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_4$
- drawBook
- Pop $\text{VM} = \text{STACK.POP}$
- drawTable
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_5$
- drawLaptop
- Pop $\text{VM} = \text{STACK.POP}$
- Pop $\text{VM} = \text{STACK.POP}$

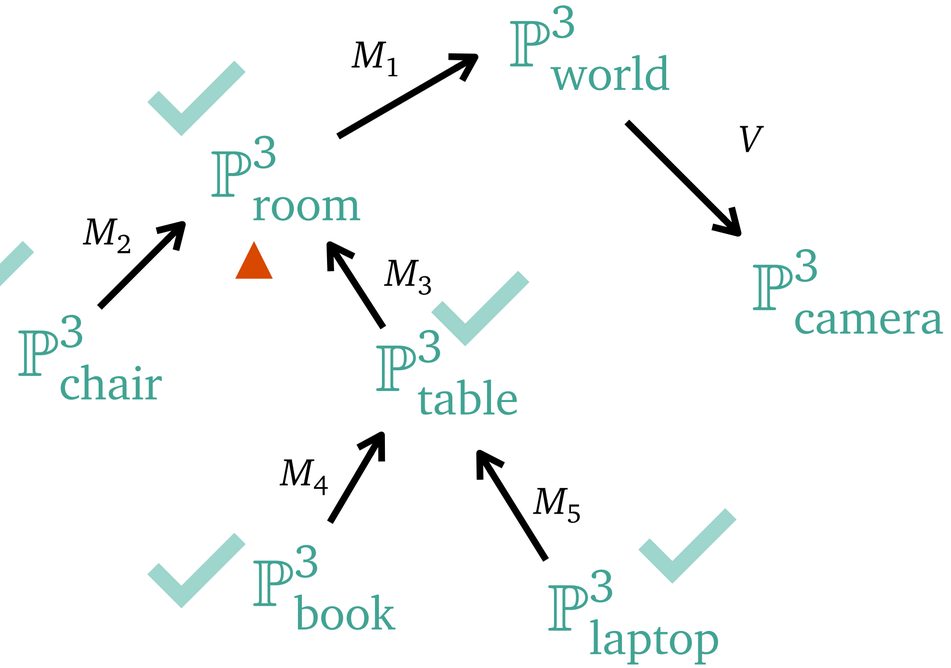




Matrix Stack

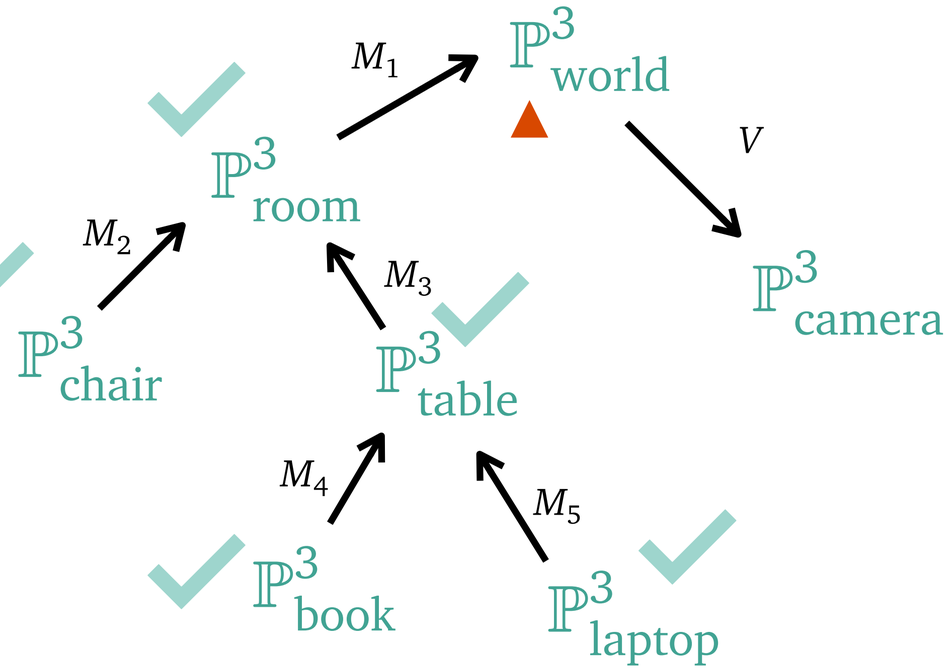
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$
- drawChair
- Pop $\text{VM} = \text{STACK.POP}$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_3$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_4$
- drawBook
- Pop $\text{VM} = \text{STACK.POP}$
- drawTable
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_5$
- drawLaptop
- Pop $\text{VM} = \text{STACK.POP}$
- Pop $\text{VM} = \text{STACK.POP}$
- Pop $\text{VM} = \text{STACK.POP}$





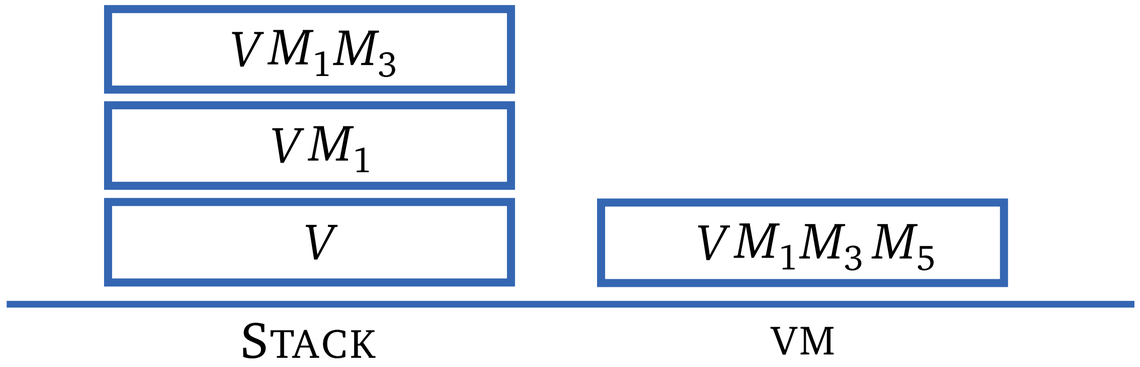
Matrix Stack

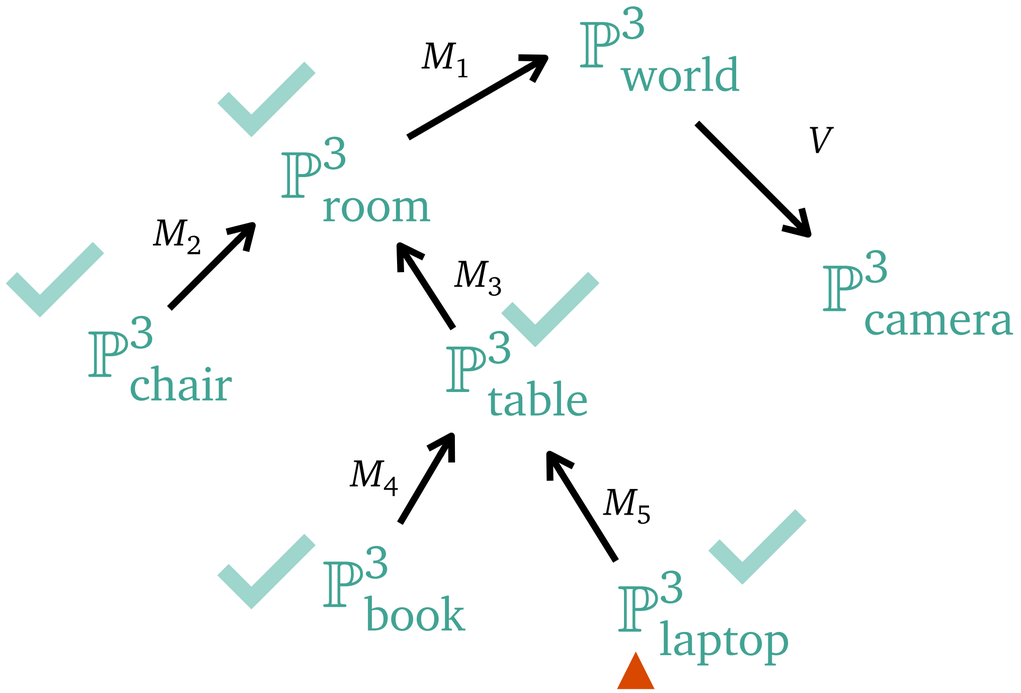
- Initially $\text{VM} = V$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_1$
- drawRoom
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_2$
- drawChair
- Pop $\text{VM} = \text{STACK.POP}$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_3$
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_4$
- drawBook
- Pop $\text{VM} = \text{STACK.POP}$
- drawTable
- Push $\text{STACK.PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} * M_5$
- drawLaptop
- Pop $\text{VM} = \text{STACK.POP}$
- Pop $\text{VM} = \text{STACK.POP}$
- Pop $\text{VM} = \text{STACK.POP}$



Matrix Stack

At any moment, **VM** is the model-view matrix of the "cursor" and the **stack** is always the list of model-view matrices of each node in the path connecting the world to the laptop.

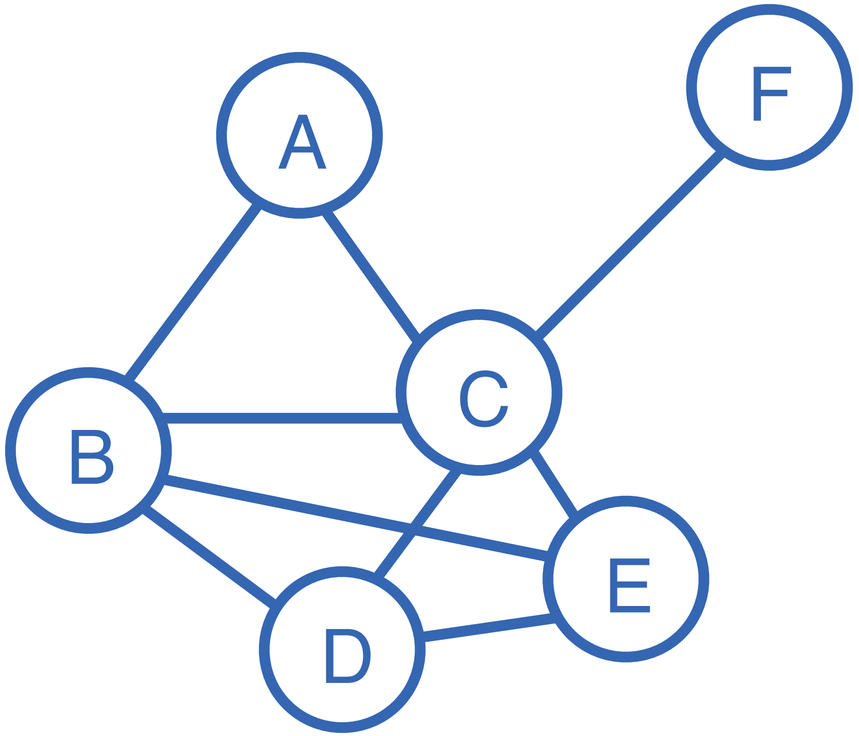




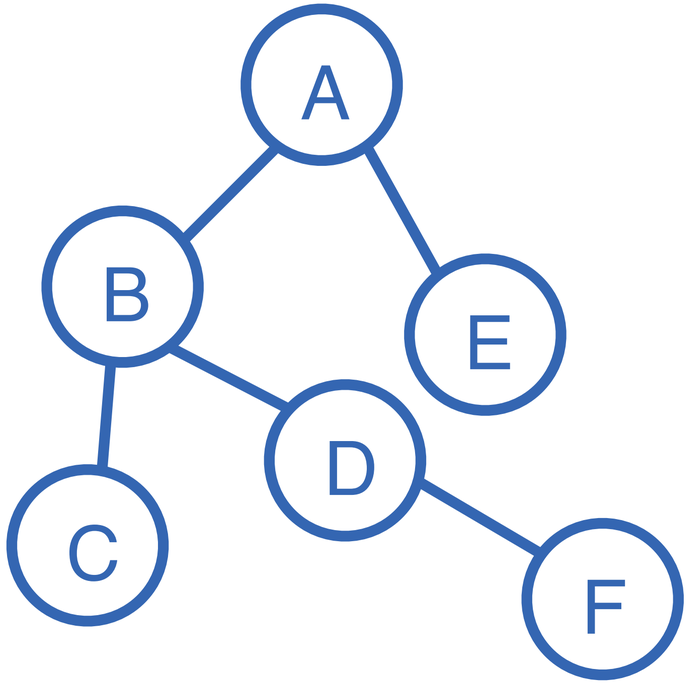
Graph traversal

- Complex scenes
- Matrix stack
- Drawing command sequence
- Graph traversal
- Scene graph data structure

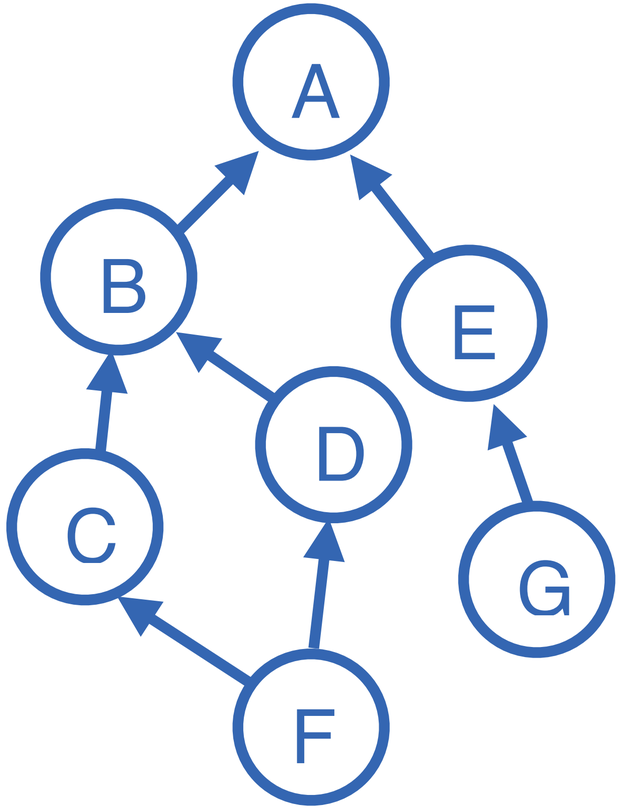
Graph traversal problem



General graph



Tree, acyclic graph



Directed acyclic graph

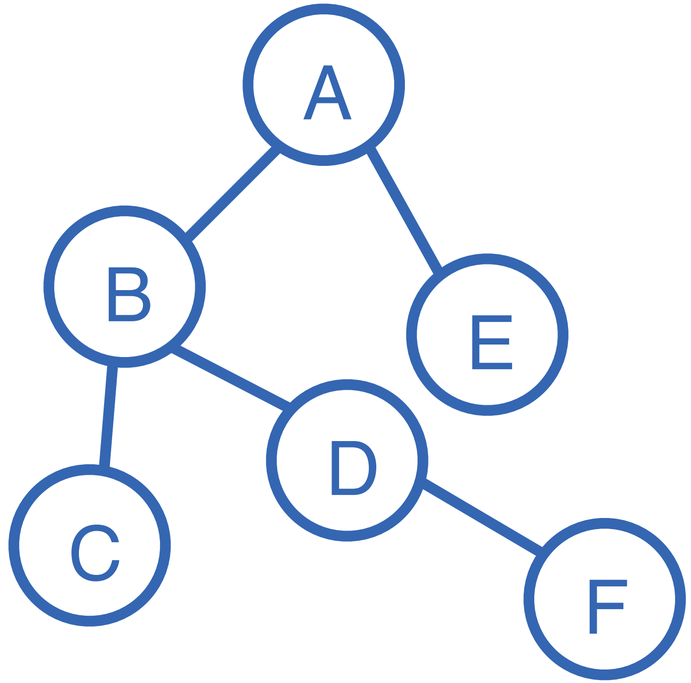
Graph traversal problem

- Breadth first search (BFS)

  (Prioritize “overviewing” over “exploring”)

- Depth first search (DFS)

  (Prioritize “exploring” over “overviewing”)



Tree, acyclic graph

Graph traversal problem

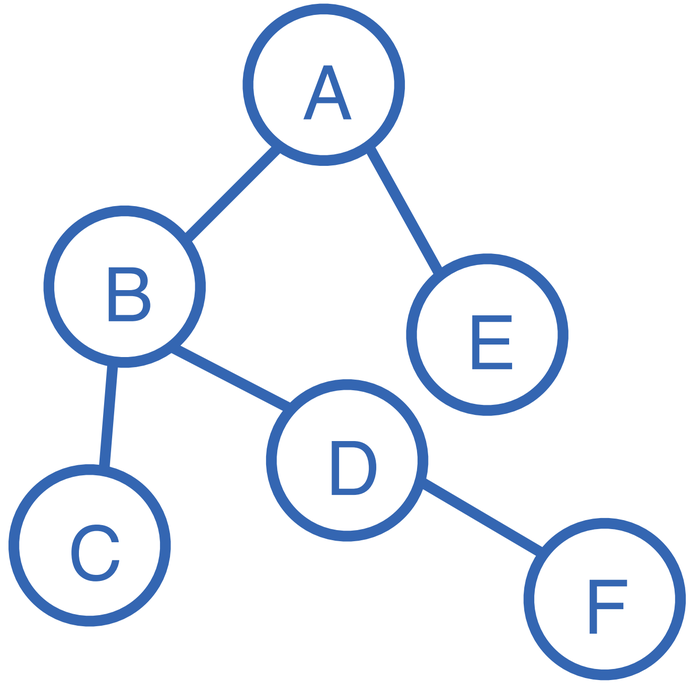
- Breadth first search (BFS)

  (Prioritize “overviewing” over “exploring”)

  A B E C D F

- Depth first search (DFS)

  (Prioritize “exploring” over “overviewing”)



Tree, acyclic graph

Graph traversal problem

- Breadth first search (BFS)

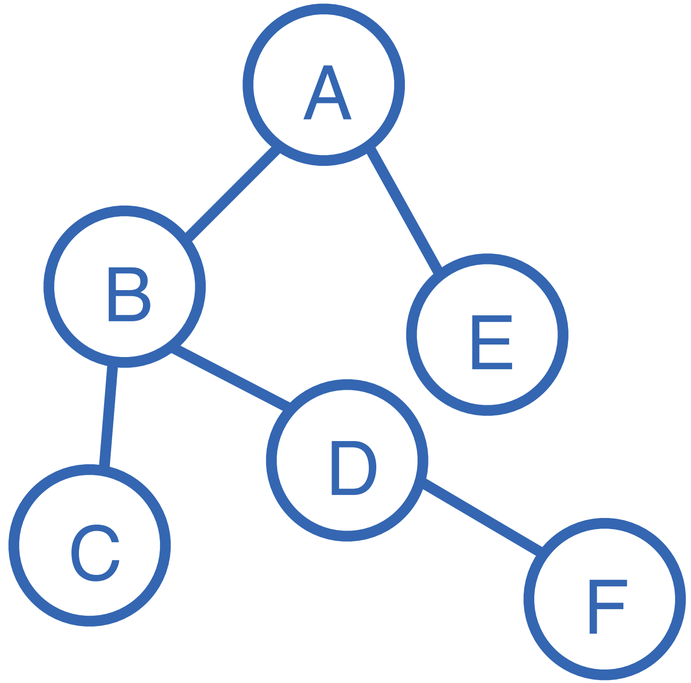
  (Prioritize “overviewing” over “exploring”)

  A B E C D F

- Depth first search (DFS)

  (Prioritize “exploring” over “overviewing”)

  A B C D F E



Tree, acyclic graph

Graph traversal problem

- Breadth first search (BFS)

  (Prioritize “overviewing” over “exploring”)

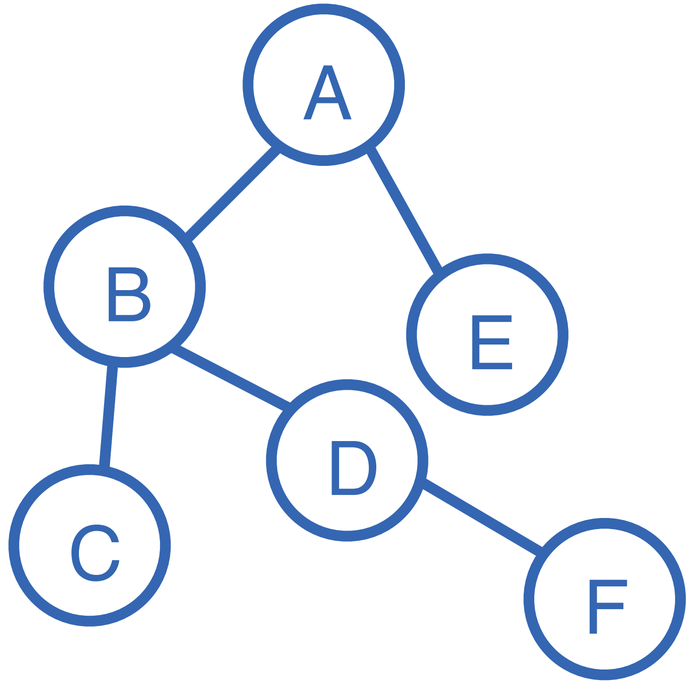
  A B E C D F

```
Put A into a queue Q.
While (Q is nonempty)
   x = Q.dequeue();
   process/visit x;
   Put neighbors of x into Q;
EndWhile
```

- Depth first search (DFS)

  (Prioritize “exploring” over “overviewing”)

  A B C D F E



Tree, acyclic graph

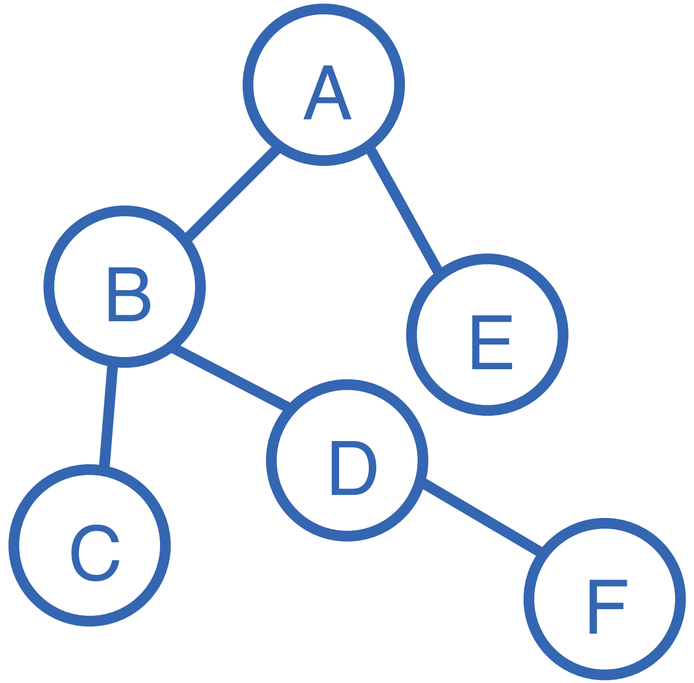
Graph traversal problem

- Breadth first search (BFS)

```
Put A into a queue Q.
While (Q is nonempty)
   x = Q.dequeue();
   process/visit x;
   Put neighbors of x into Q;
EndWhile
```

- Depth first search (DFS)

```
Push A into a stack S.
While (S is nonempty)
   x = S.pop();
   process/visit x;
   Push neighbors of x into S;
EndWhile
```



Tree, acyclic graph

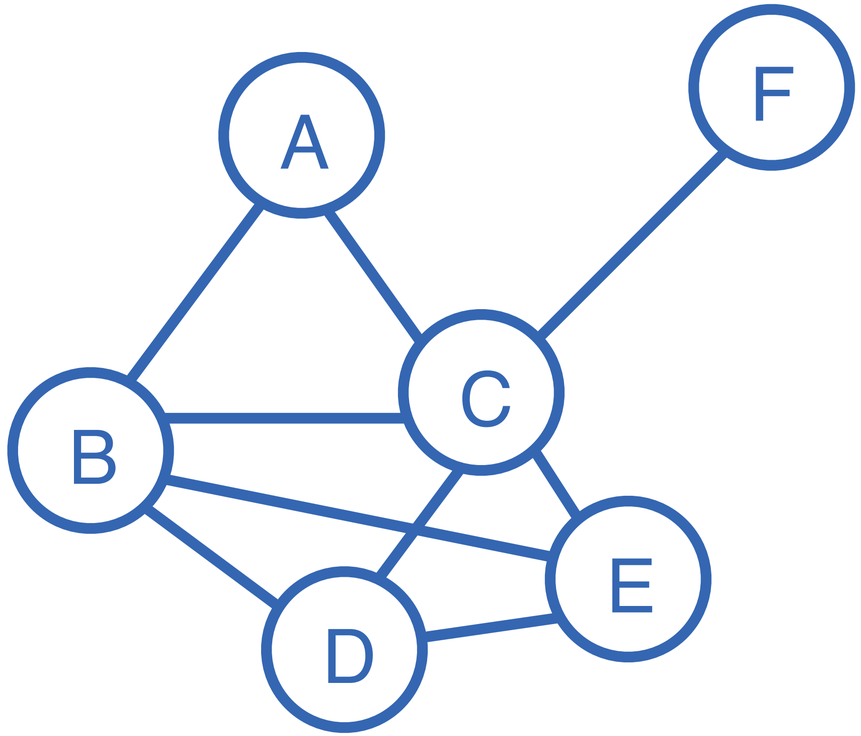
Graph traversal problem

- Breadth first search (BFS)

```
Put A into a queue Q.
While (Q is nonempty)
   x = Q.dequeue();
   process/visit x; Mark x as visited;
   Put unvisited neighbors of x into Q;
EndWhile
```

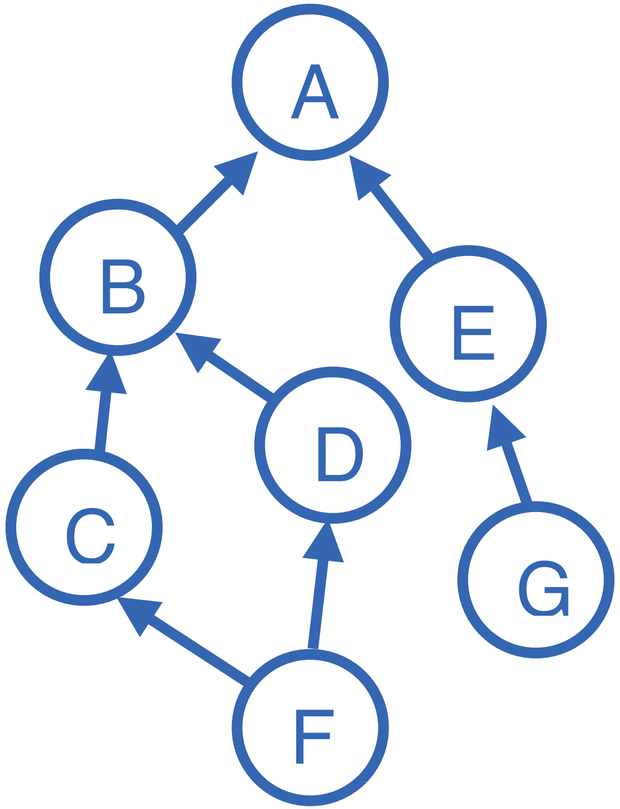
- Depth first search (DFS)

```
Push A into a stack S.
While (S is nonempty)
   x = S.pop();
   process/visit x; Mark x as visited;
   Push unvisited neighbors of x into S;
EndWhile
```



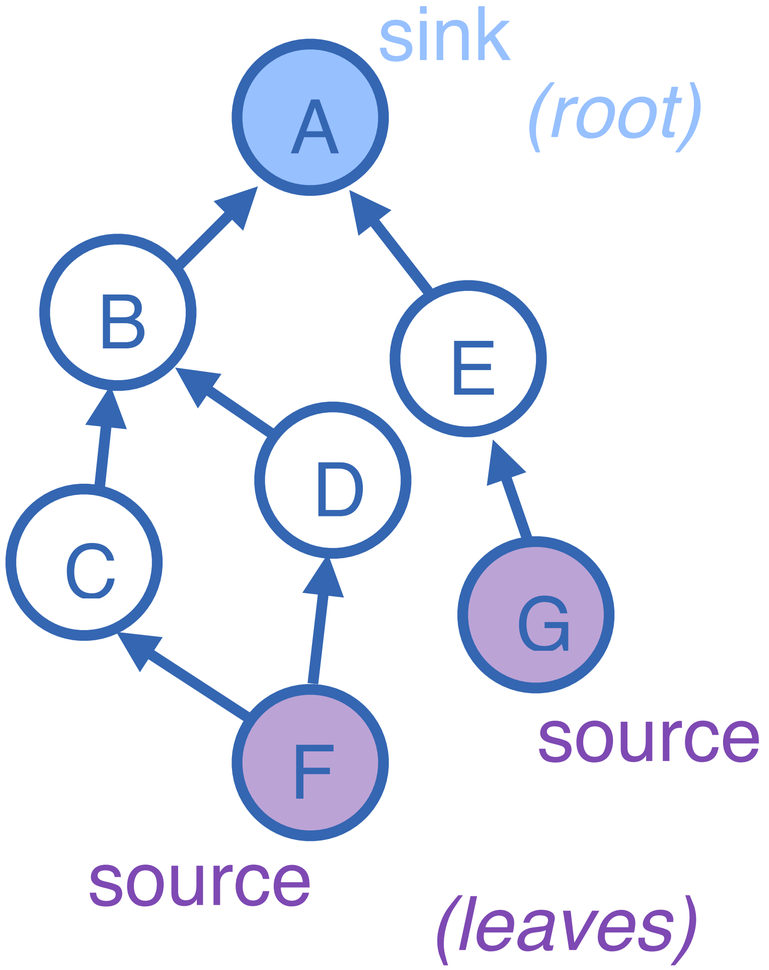
General graph

Directed acyclic graph



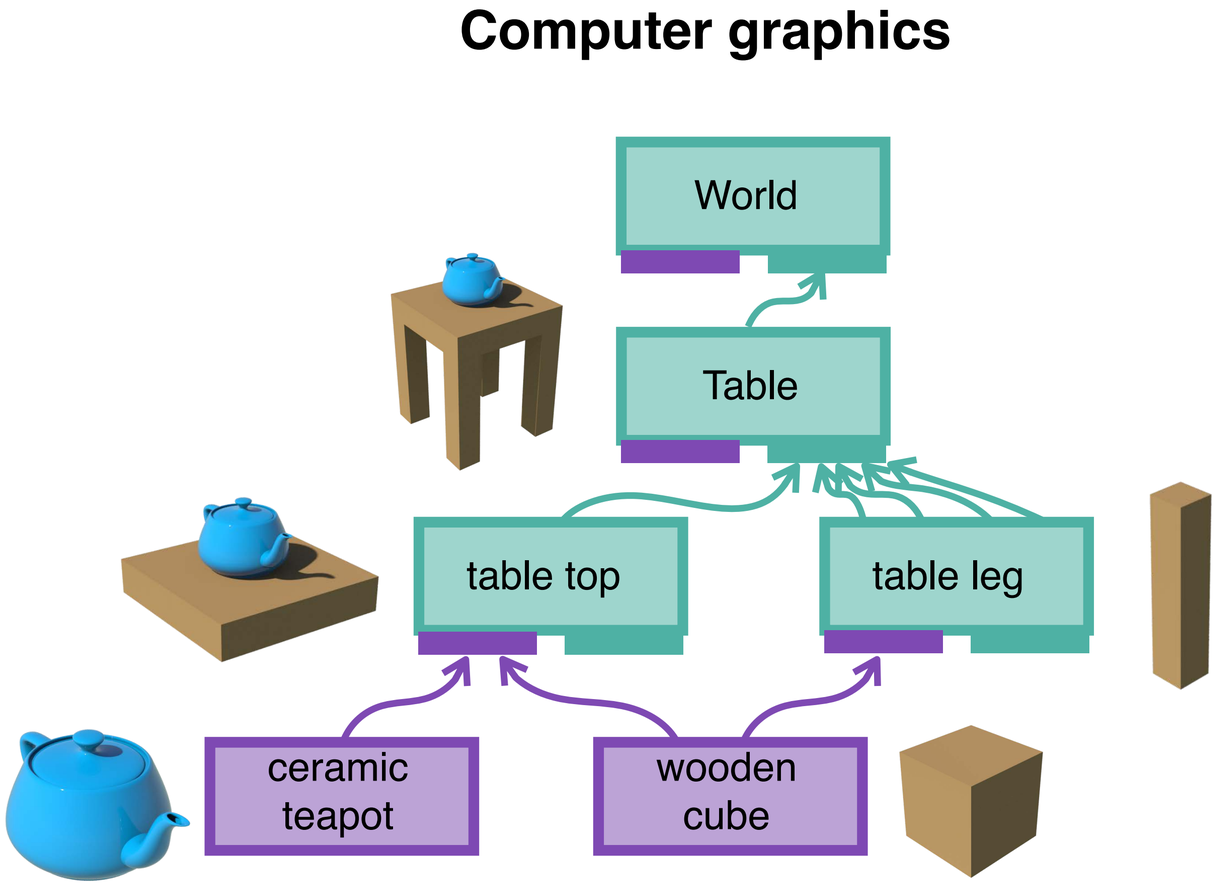
Directed acyclic graph

"Rooted" directed acyclic graph (it has a single sink)



Directed acyclic graph

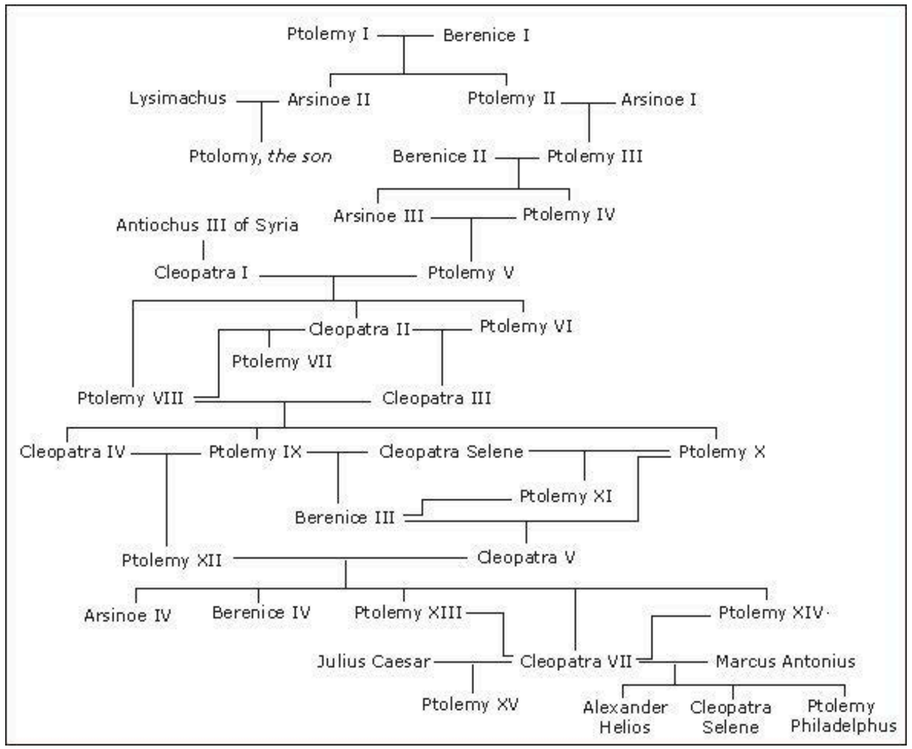
"Rooted" directed acyclic graph (it has a single sink)



Directed acyclic graph

“Rooted” directed acyclic graph (it has a single sink)

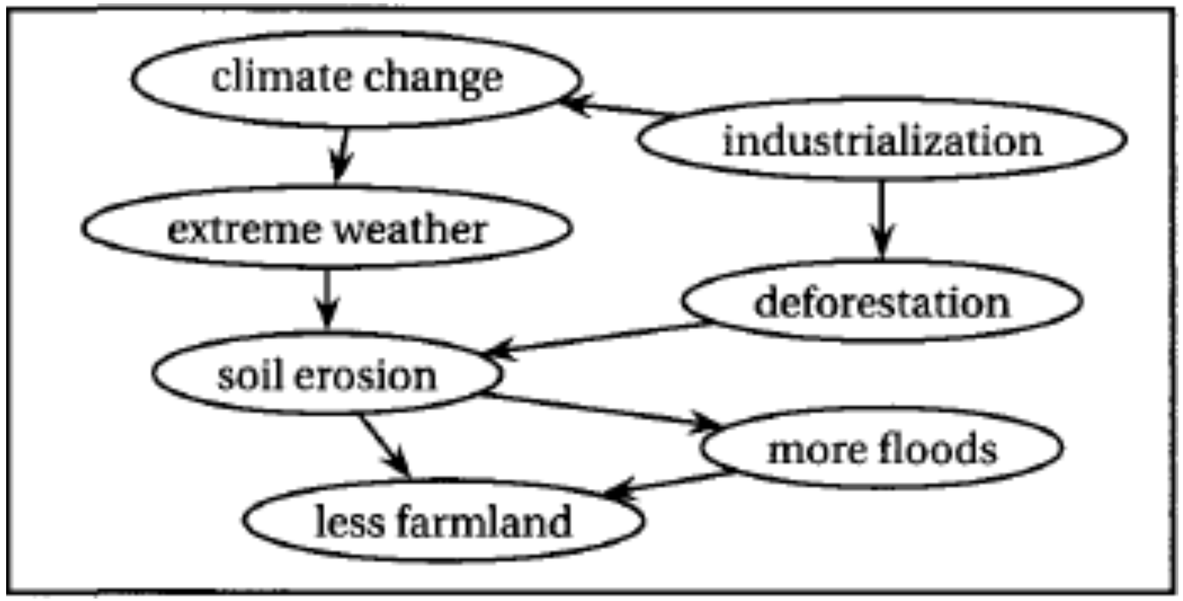
Family “tree”



Directed acyclic graph

"Rooted" directed acyclic graph (it has a single sink)

Causal network

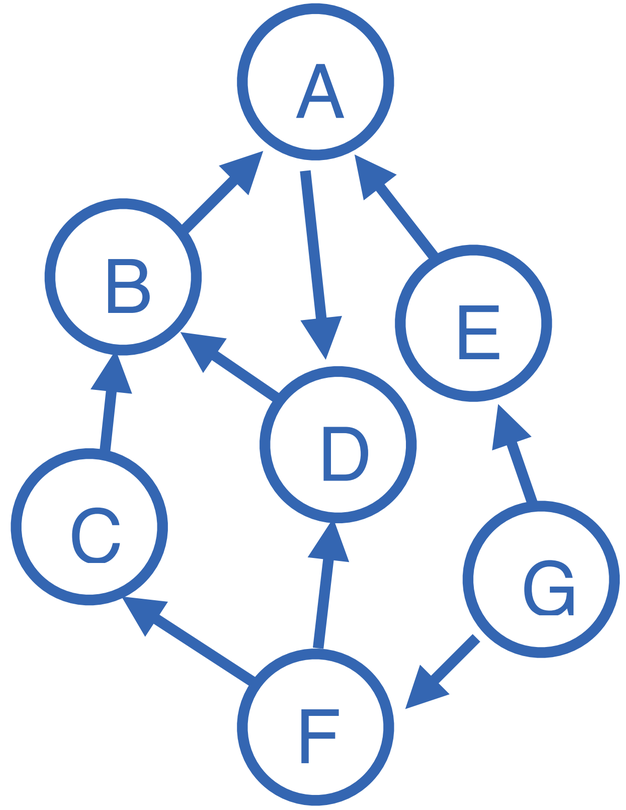


Reachability

Any directed graph (such as shown on the right) has a **reachability relation** $\preceq$ on the node set:

$$\text{node}_1 \preceq \text{node}_2$$

if there exists a path traveling from node_1 to node_2.



$B \preceq D$

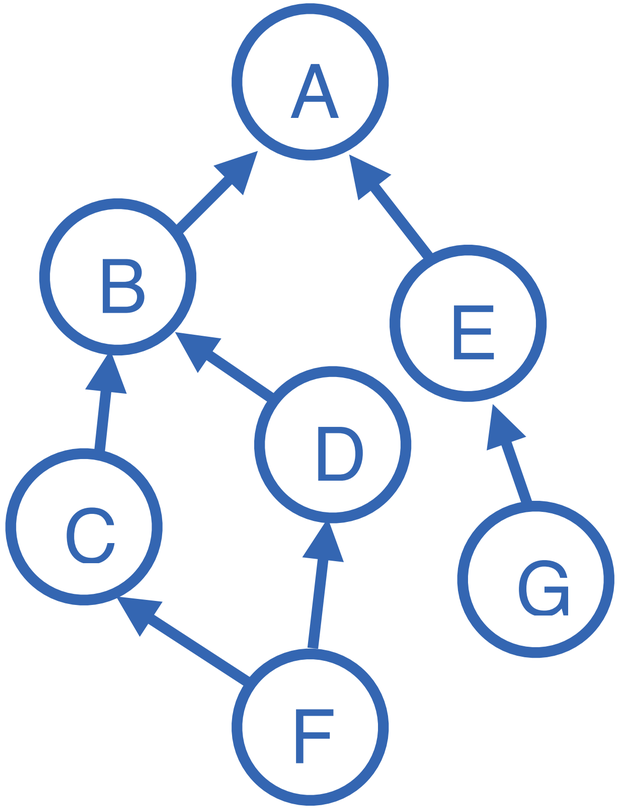
$D \preceq B$ $\quad$ $C \not\preceq E$

$B \not\preceq E$ $\quad$ $E \not\preceq C$

Definition of directed acyclic graph

A directed is acyclic if the reachability relation $\preceq$ becomes a ***partial ordering***. That is,

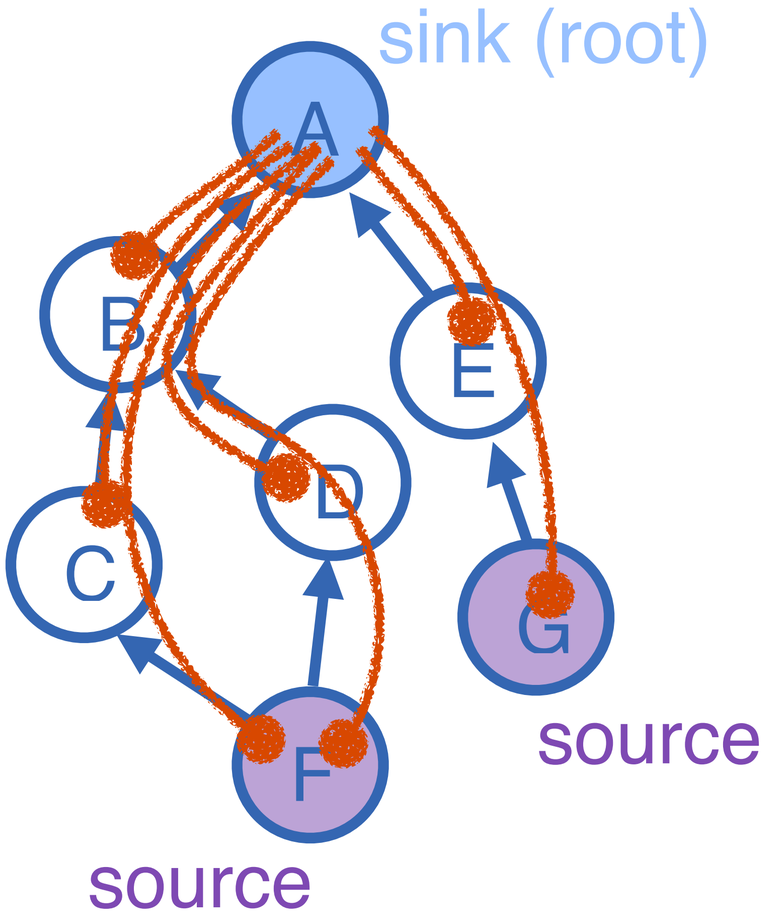
$$X \preceq Y \quad \& \quad Y \preceq X \quad \Longrightarrow \quad X = Y$$



If you like this sort of stuff, check out the lecture note

Traversing over a rooted DAG

We want to traverse over all paths in a rooted directed acyclic graph that ends at the sink (root).



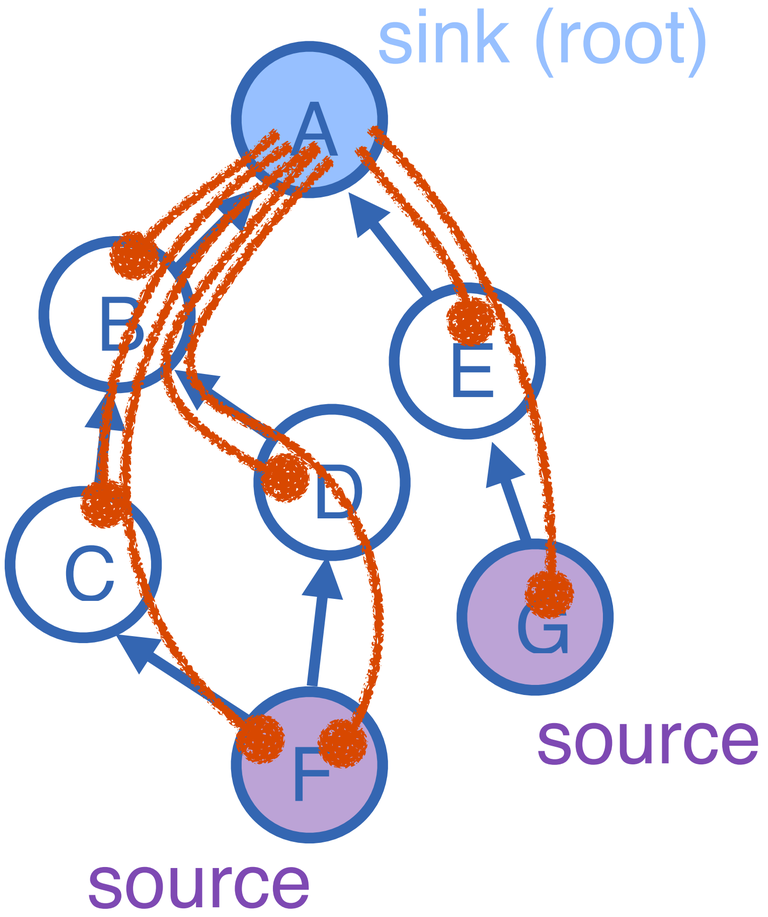
Traversing over a rooted DAG

- Breadth first search (BFS)

  Put A (root) into a queue **Q**.
  While (**Q** is nonempty)
    x = **Q**.dequeue();
    process/visit x; ~~Mark x as visited;~~
    Put ~~unvisited~~ children of x into **Q**;
  EndWhile

- Depth first search (DFS)

  Push A (root) into a stack **S**.
  While (**S** is nonempty)
    x = **S**.pop();
    process/visit x; ~~Mark x as visited;~~
    Push ~~unvisited~~ children of x into **S**;
  EndWhile



Traversing over a rooted DAG

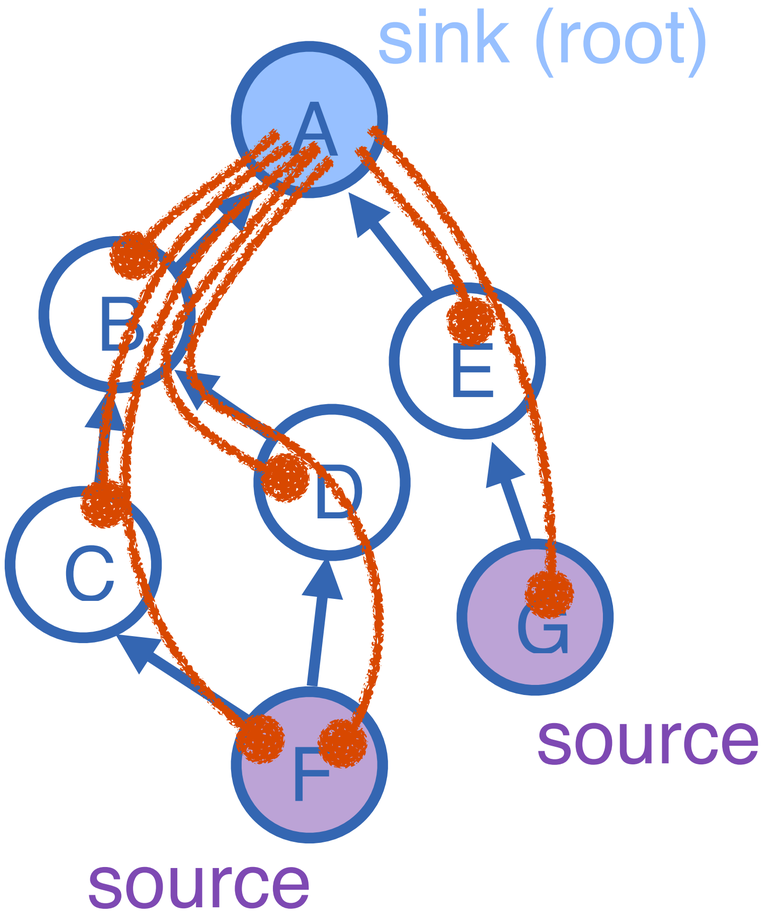
- Breadth first search (BFS)

  Put A (root) into a queue **Q**.
  While (**Q** is nonempty)
    x = **Q**.dequeue();
    process/visit x; ~~Mark x as visited;~~
    Put ~~unvisited~~ children of x into **Q**;
  EndWhile

- Depth first search (DFS)

  Push A (root) into a stack **S**.
  While (**S** is nonempty)
    x = **S**.pop();
    process/visit x; ~~Mark x as visited;~~
    Push ~~unvisited~~ children of x into **S**;
  EndWhile

this resembles our low-level matrix stack procedure



Traversing over a scene graph

Let x denote the current node.

Let vm denote the current modelview matrix.

Let node_stack denote a stack of nodes.

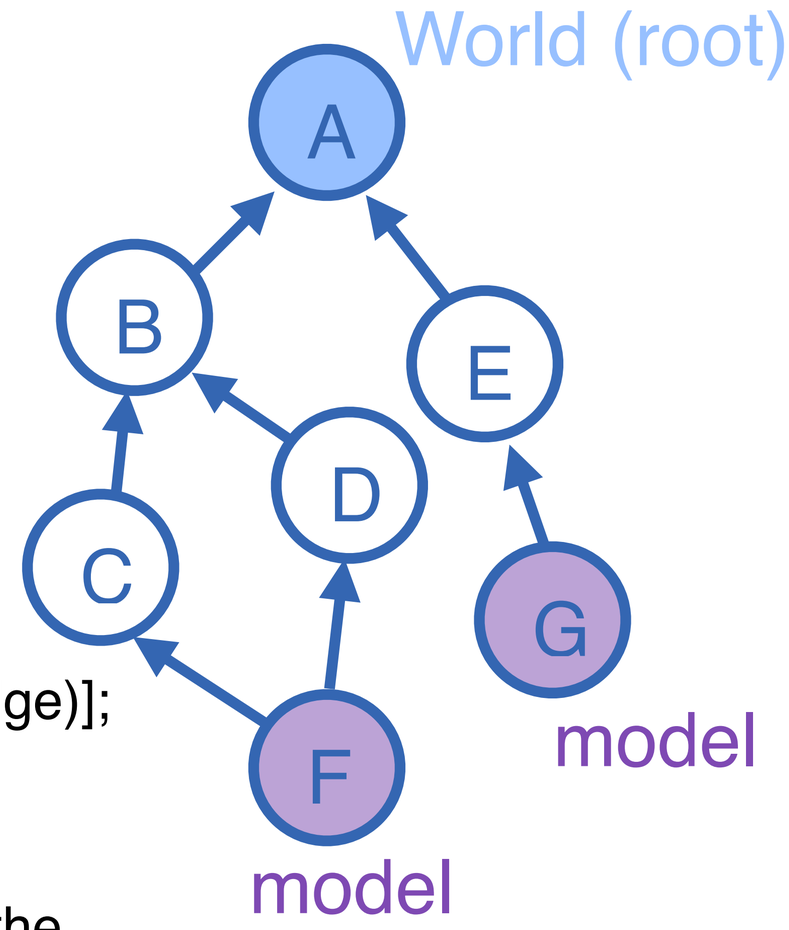
Let matrix_stack denote a stack of nodes.

- Initialize x = the “World” node.
- Initialize vm = camera’s view matrix.
- Push x into the node_stack and vm into the matrix_stack.

While node_stack is nonempty

- x = node_stack.pop(); vm = matrix_stack.pop();
- Draw all models attached to x:
  - Set shader’s modelview to [vm*(matrix associated to the edge)];
  - Draw the model;
- Push each child node into the node_stack
  and correspondingly [vm*(matrix associated to the edge)] into the matrix_stack.

EndWhile



Traversing over a scene graph

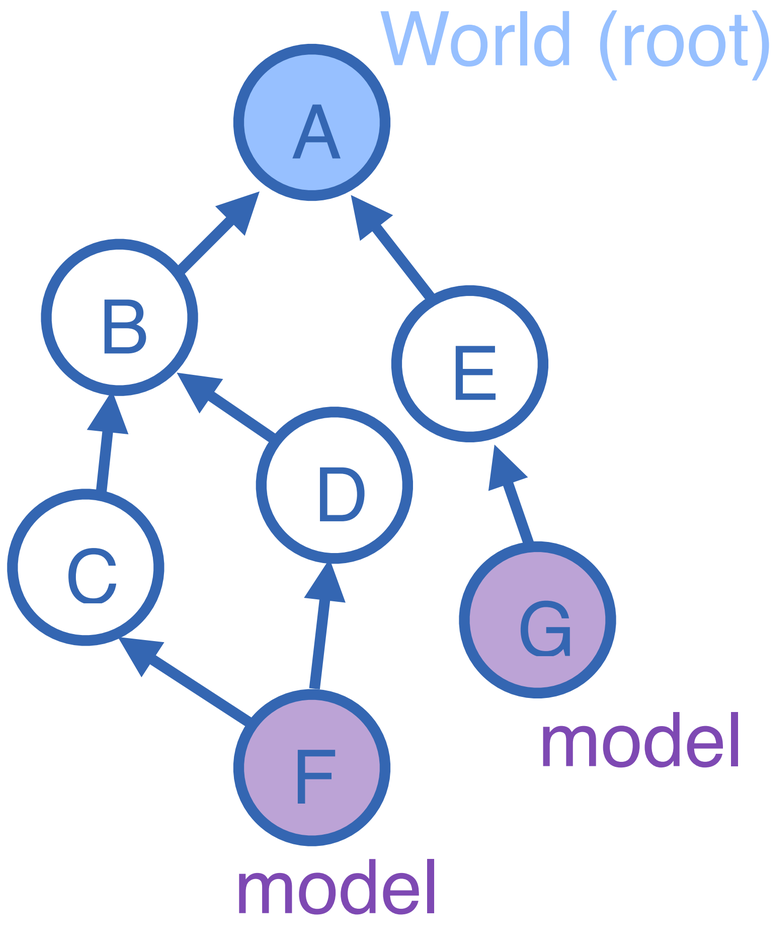
- The current node x and the current modelview matrix vm are always correctly paired.

- At any moment, both stacks

$$\text{node\_stack} = (x_1, \ldots, x_k)$$
$$\text{matrix\_stack} = (m_1, \ldots, m_k)$$

have the same size, and x_i, m_i are correctly paired for all i=1,…,k.

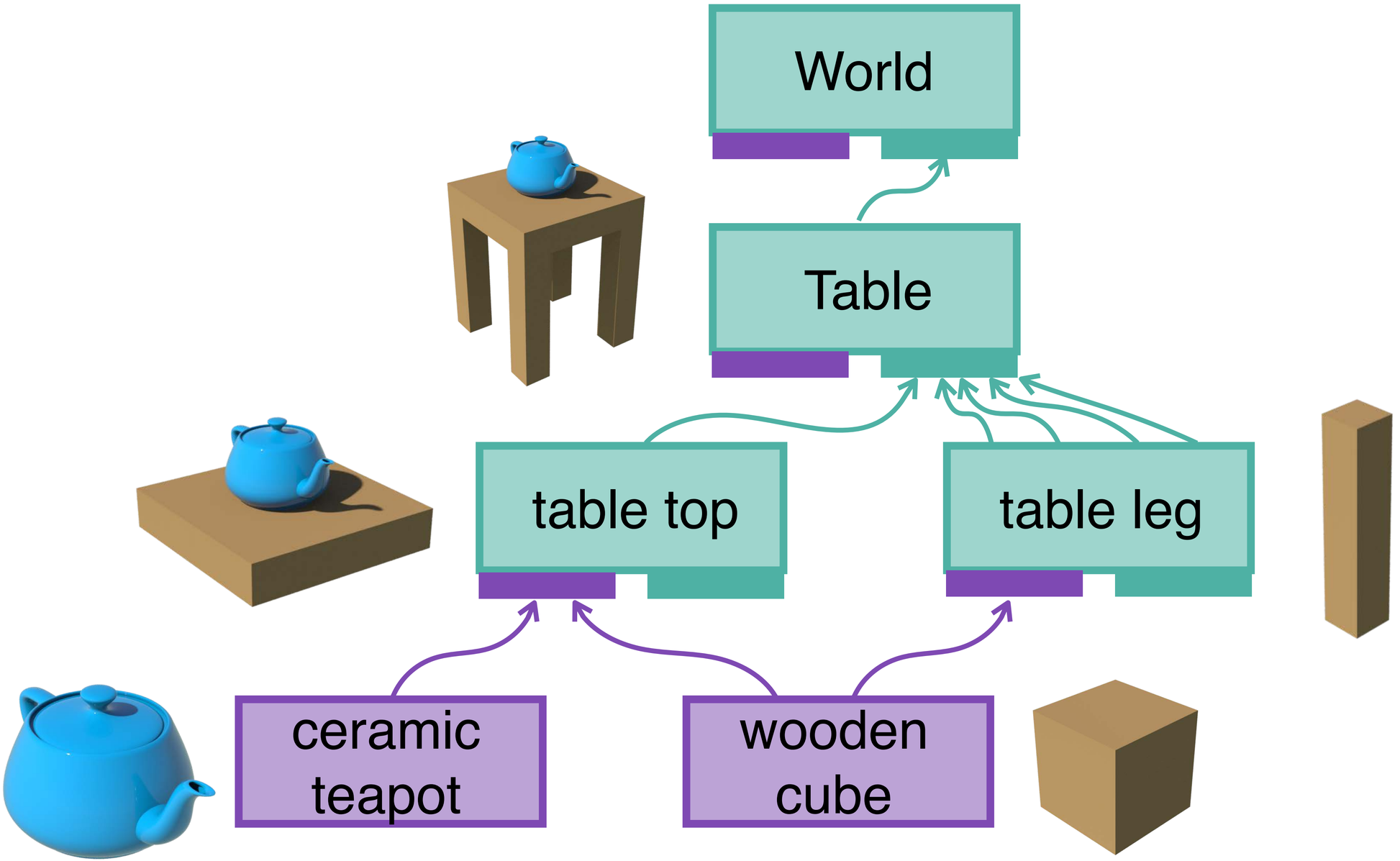
- Whenever we draw models in x, we have the correct modelview matrix ready.



Scene graph data structure

- Complex scenes
- Matrix stack
- Drawing command sequence
- Graph traversal
- Scene graph data structure

Scene graph data structure



Scene graph data structure

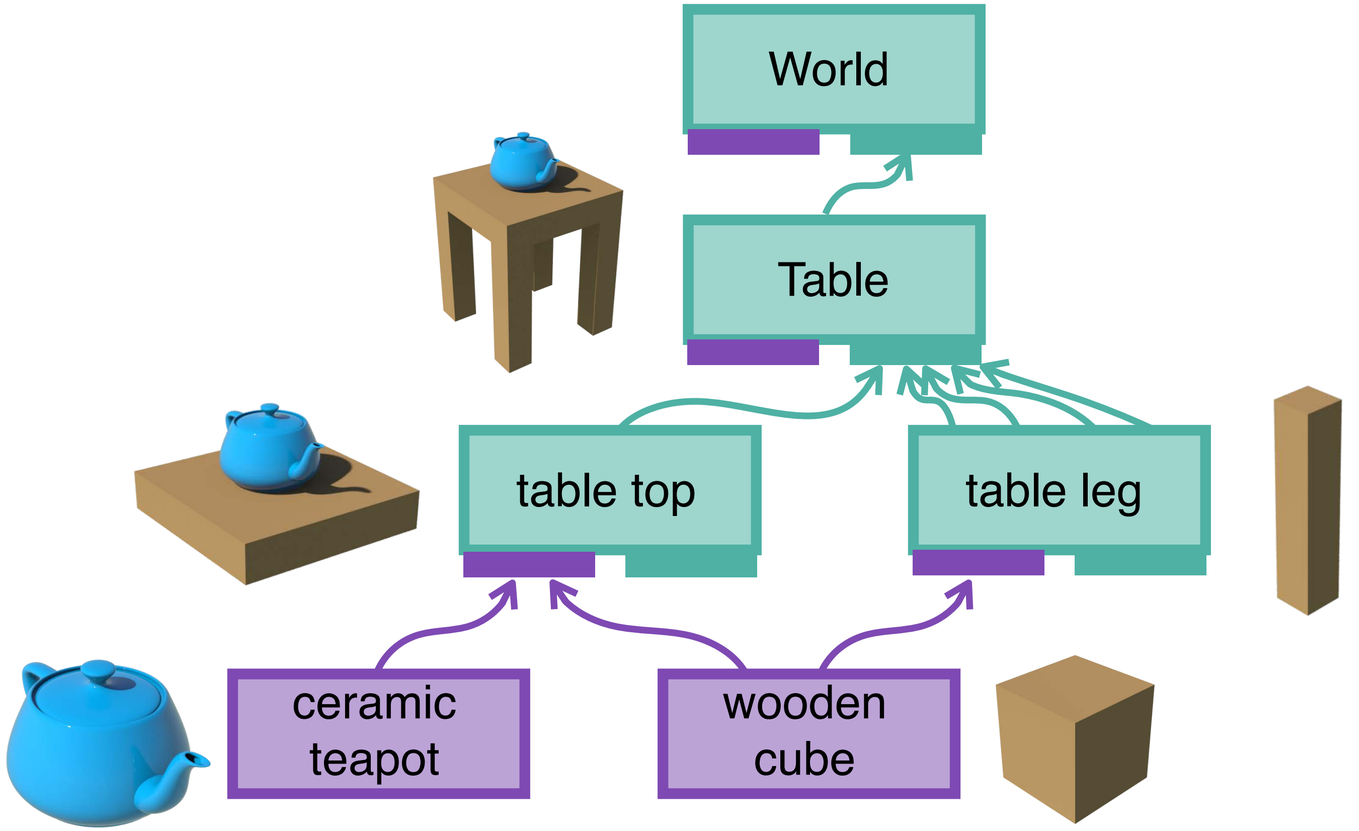
There are two kind of “nodes”:

(regular) node

- Each node records a list of pointers to child nodes and child models.
- Each connection also has the info of transformation.

model (leaf node)

- Each model contains the info for drawing the object; i.e. it has a geometry and a set of shader parameters.



Scene graph data structure

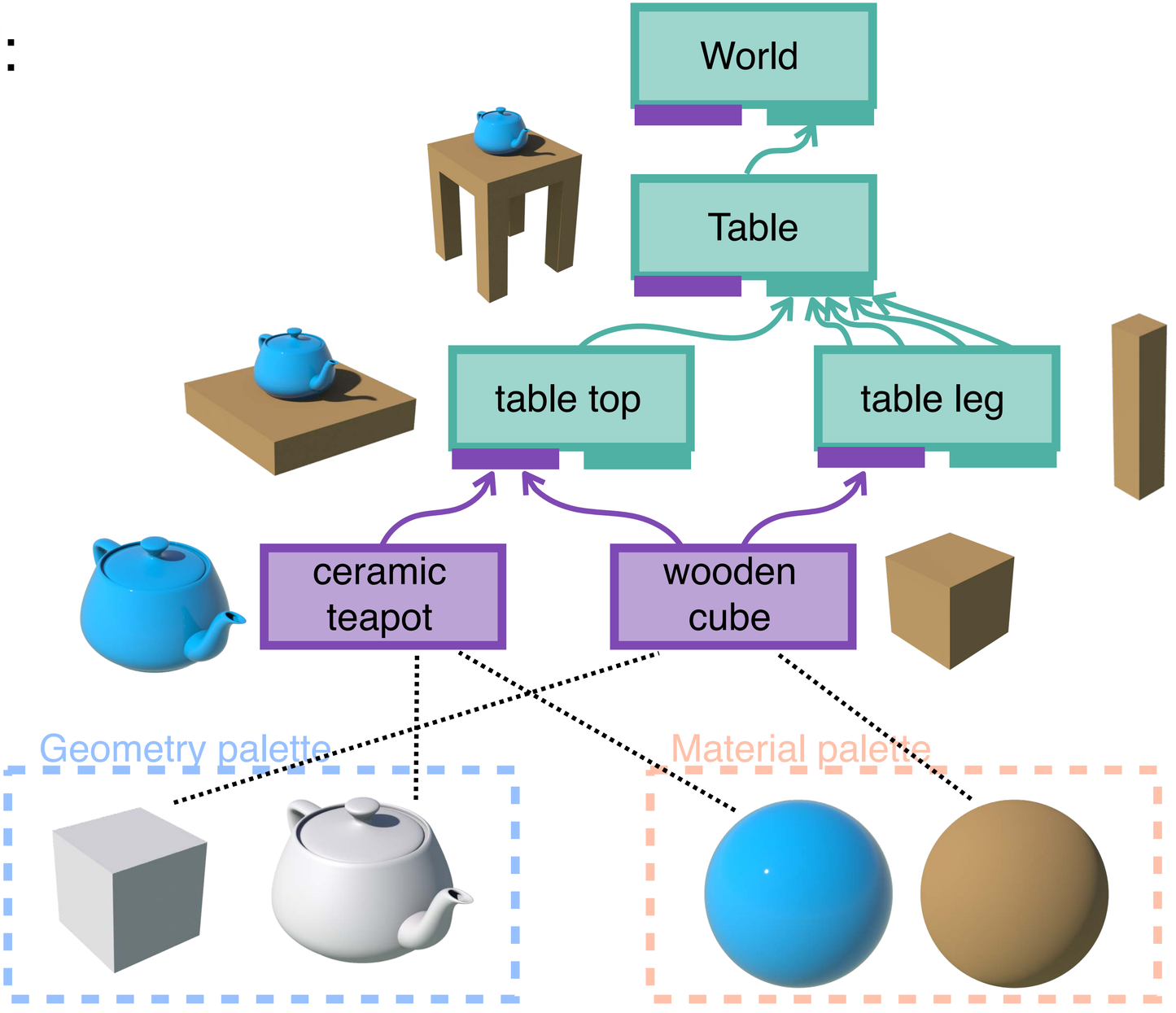
There are two kind of “nodes”:

(regular) node

- Each node records a list of pointers to child nodes and child models.
- Each connection also has the info of transformation.

model (leaf node)

- Each model contains the info for drawing the object; i.e. it has a geometry and a set of shader parameters.

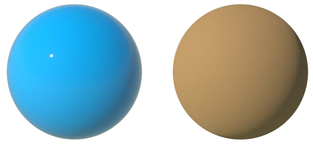


Scene graph data structure

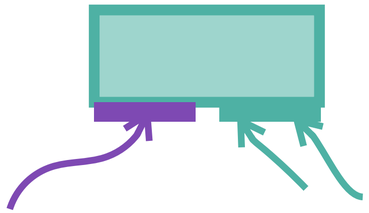
```
class Geometry{
    virtual void init();
    void draw();
};
```

```
struct Model{
    Geometry* geometry;
    Material* material;
};
```

```
struct Material{
    vec4 ambient;
    vec4 diffuse;
    vec4 specular;
    vec4 emission;
    float shininess;
};
```

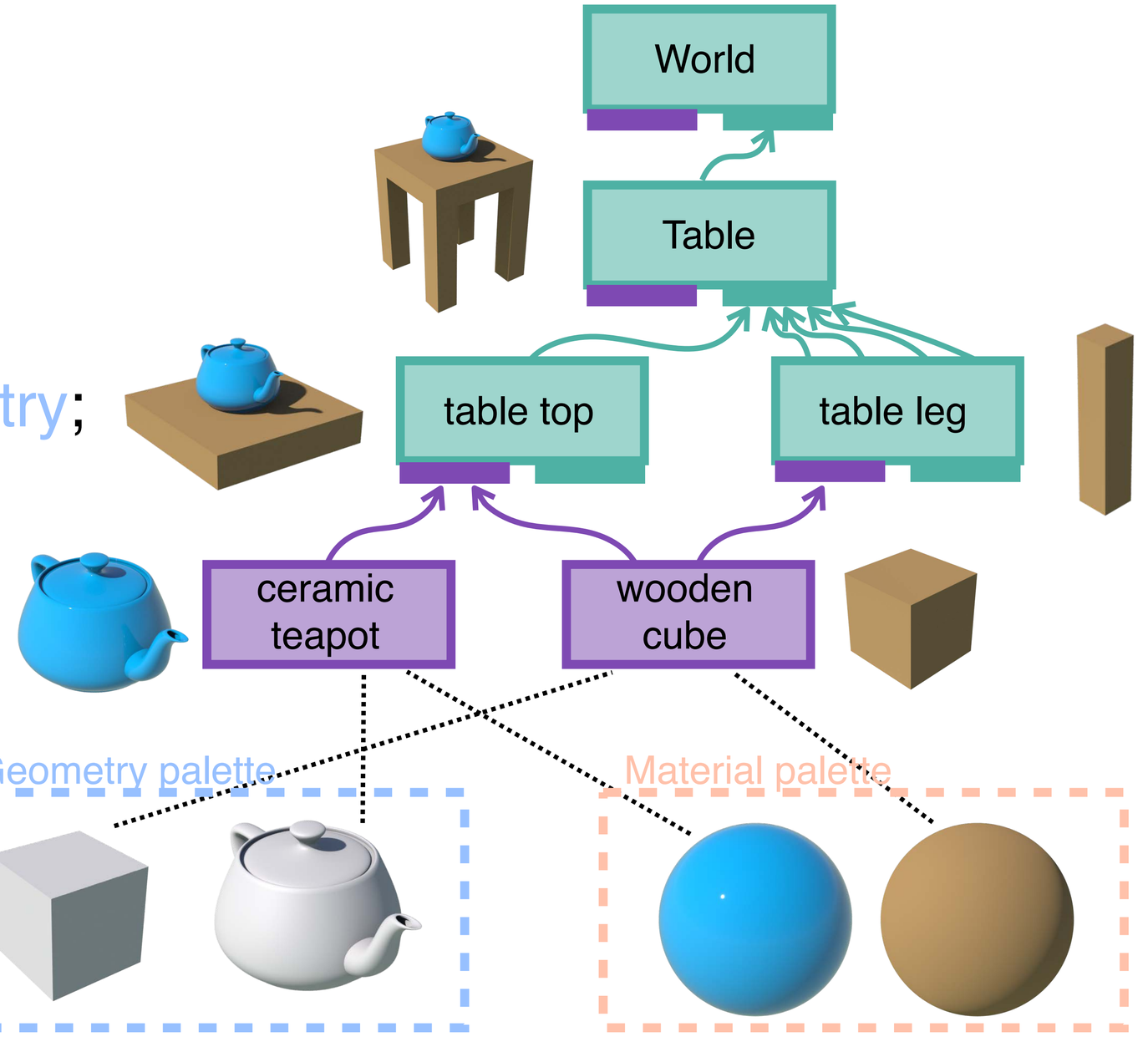


```
struct Node{
    std::vector<Node*> childnodes;
    std::vector<mat4> childtransforms;

    std::vector<Models*> models;
    std::vector<mat4> modeltransforms;
};
```



Scene graph data structure

```
class Scene{

  Container<Node*> node;
  Container<Model*> model;
  Container<Geometry*> geometry;
  Container<Material*> material;


  void init();
  void draw();
};
```



Scene graph data structure

```
class Scene{

  Container<Node*> node;
  Container<Model*> model;
  Container<Geometry*> geometry;
  Container<Material*> material;


  void init();
  void draw();
};
```

Here, *Container* can be either

- std::vector<Type>
- std::map<std::string, Type>

Scene graph data structure

```
class Scene{

  std::vector<Node*> node;
  std::vector<Model*> model;
  std::vector<Geometry*> geometry;
  std::vector<Material*> material;


  void init();
  void draw();
};
```

If we use std::vector, we access each node, model, geometry, material by

node[0], node[1], etc

Scene graph data structure

```
class Scene{

  std::map<std::string, Node*> node;
  std::map<std::string, Model*> model;
  std::map<std::string, Geometry*> geometry;
  std::map<std::string, Material*> material;


  void init();
  void draw();
};
```

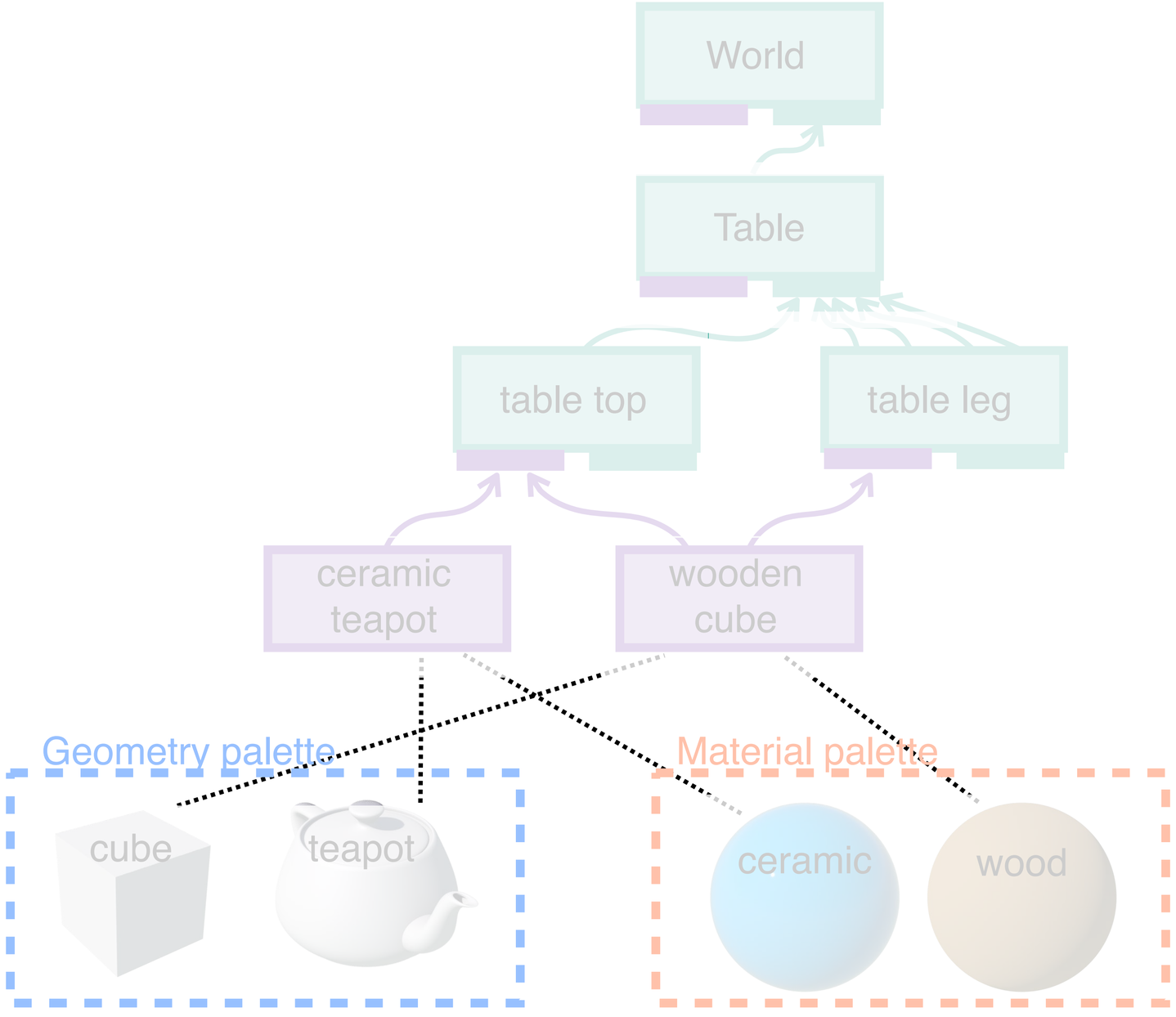
If we use std::map, we access each node, model, geometry, material by

node[“world”], node[“table”], etc.

We will be using map<std::string, . > for our container.

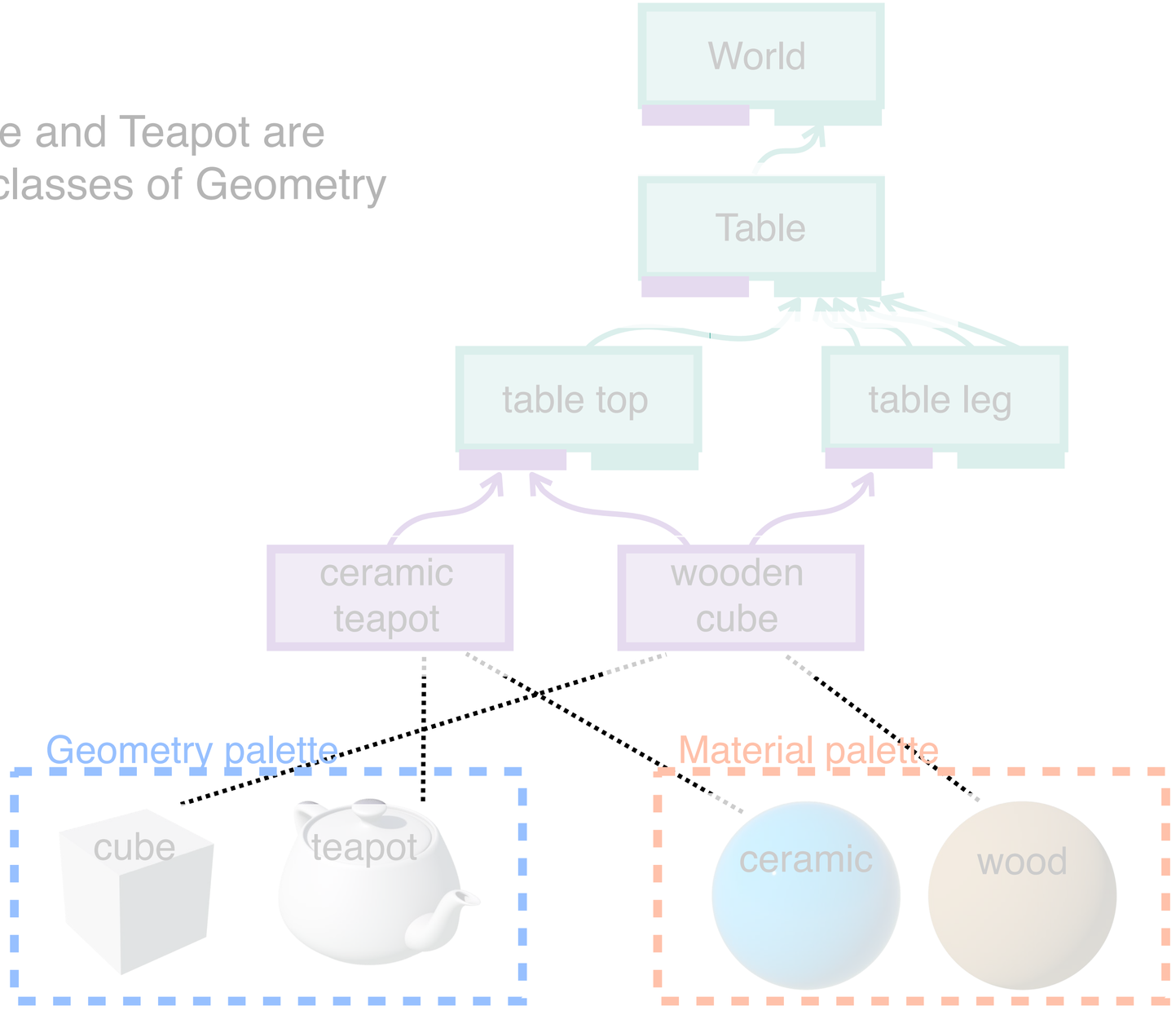
Example for setting up a scene graph

```
void Scene::init(){

};
```



Example for setting up a scene graph

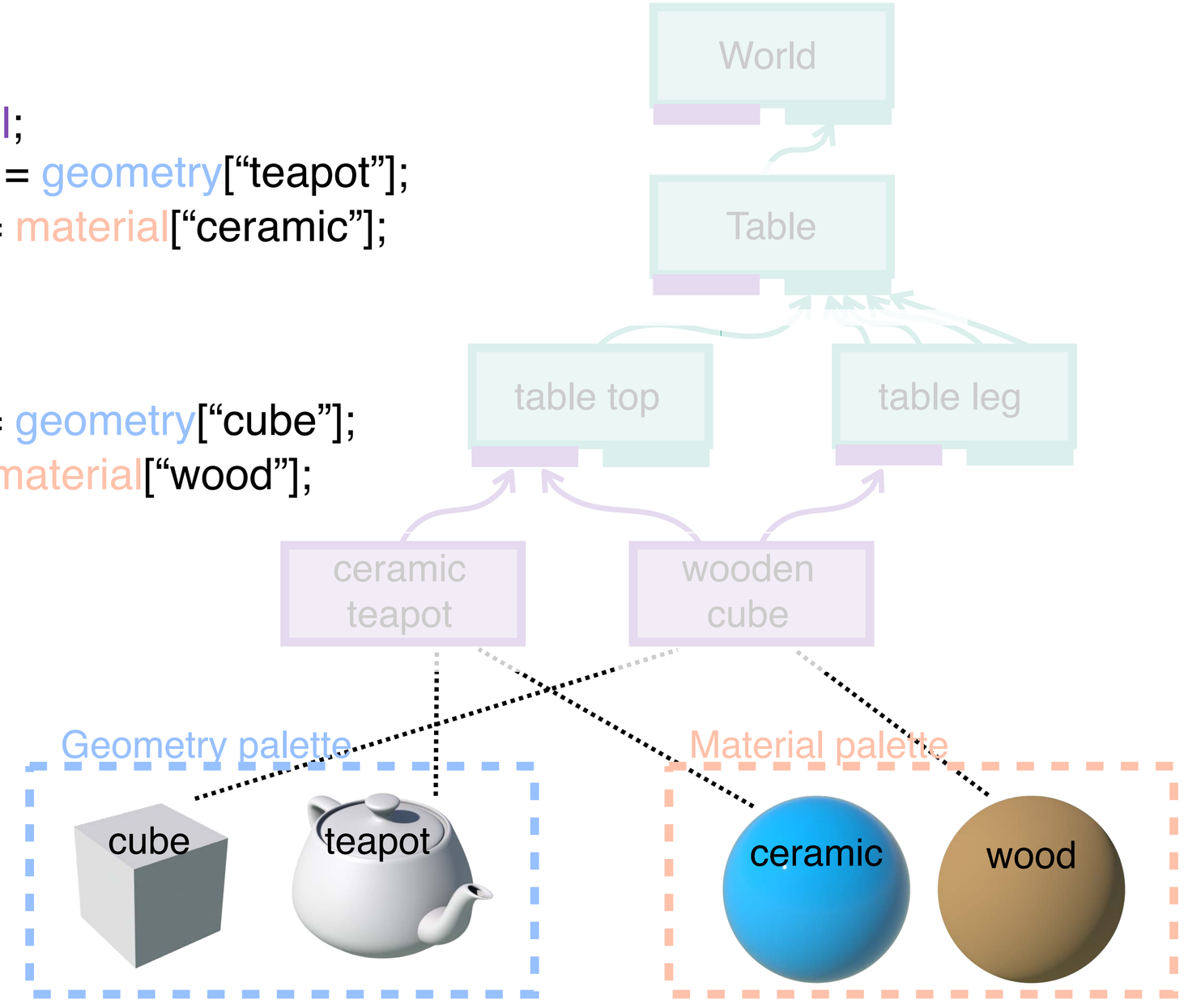
```
void Scene::init(){
    geometry[“cube”] = new Cube; // Cube and Teapot are
    geometry[“cube”] -> init();          // subclasses of Geometry
    geometry[“teapot”] = new Teapot;
    geometry[“teapot”] -> init();

    material[“ceramic”] -> new Material;
    material[“ceramic”] -> ambient = …;
    material[“ceramic”] -> diffuse = …;
    …

    material[“wood”] -> new Material;
    material[“wood”] -> ambient = …;
    material[“wood”] -> diffuse = …;
    …
```



Example for setting up a scene graph

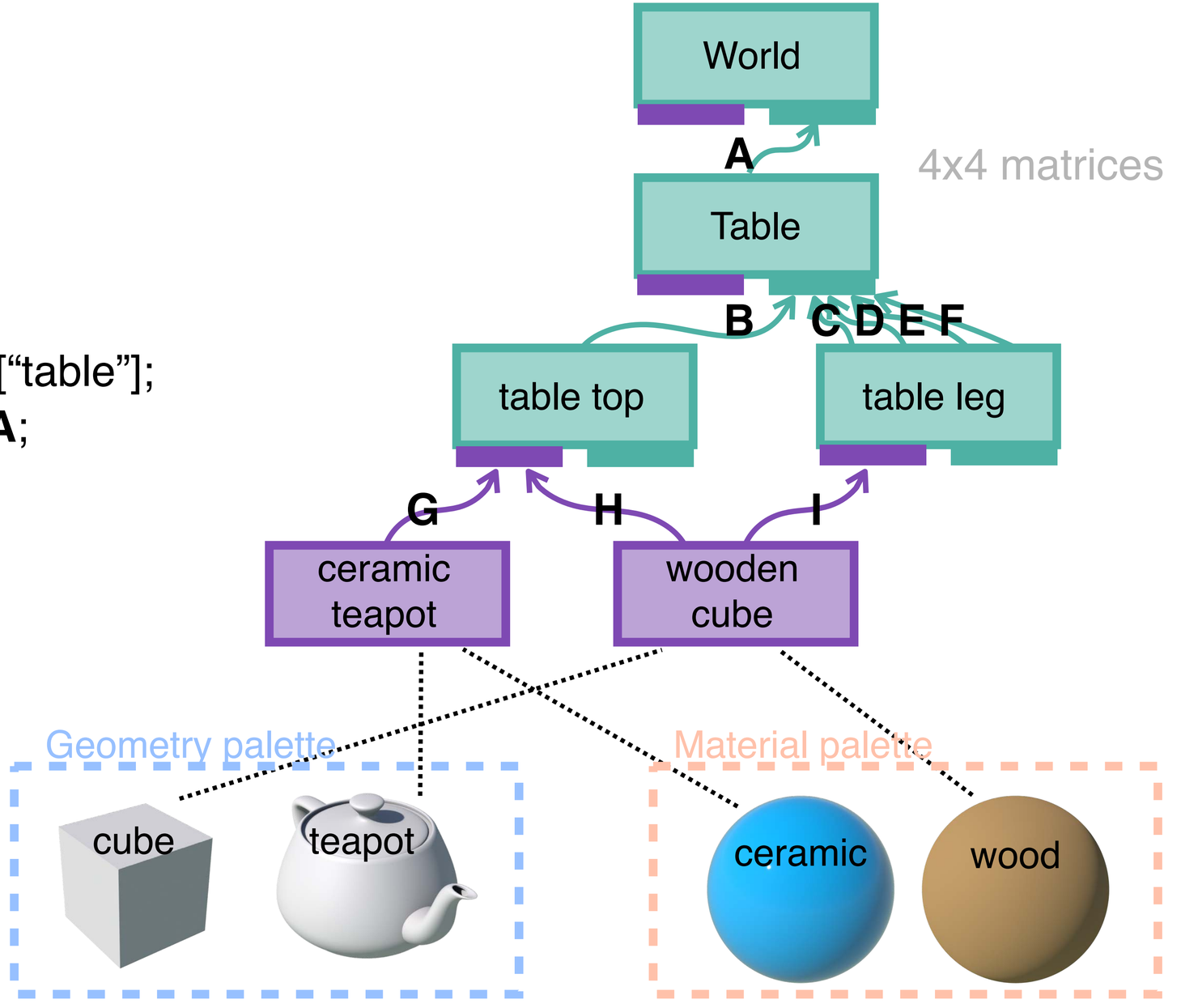
…

```
model[“ceramic teapot”] = new Model;
model[“ceramic teapot”] -> geometry = geometry[“teapot”];
model[“ceramic teapot”] -> material = material[“ceramic”];

model[“wooden cube”] = new Model;
model[“wooden cube”] -> geometry = geometry[“cube”];
model[“wooden cube”] -> material = material[“wood”];
```



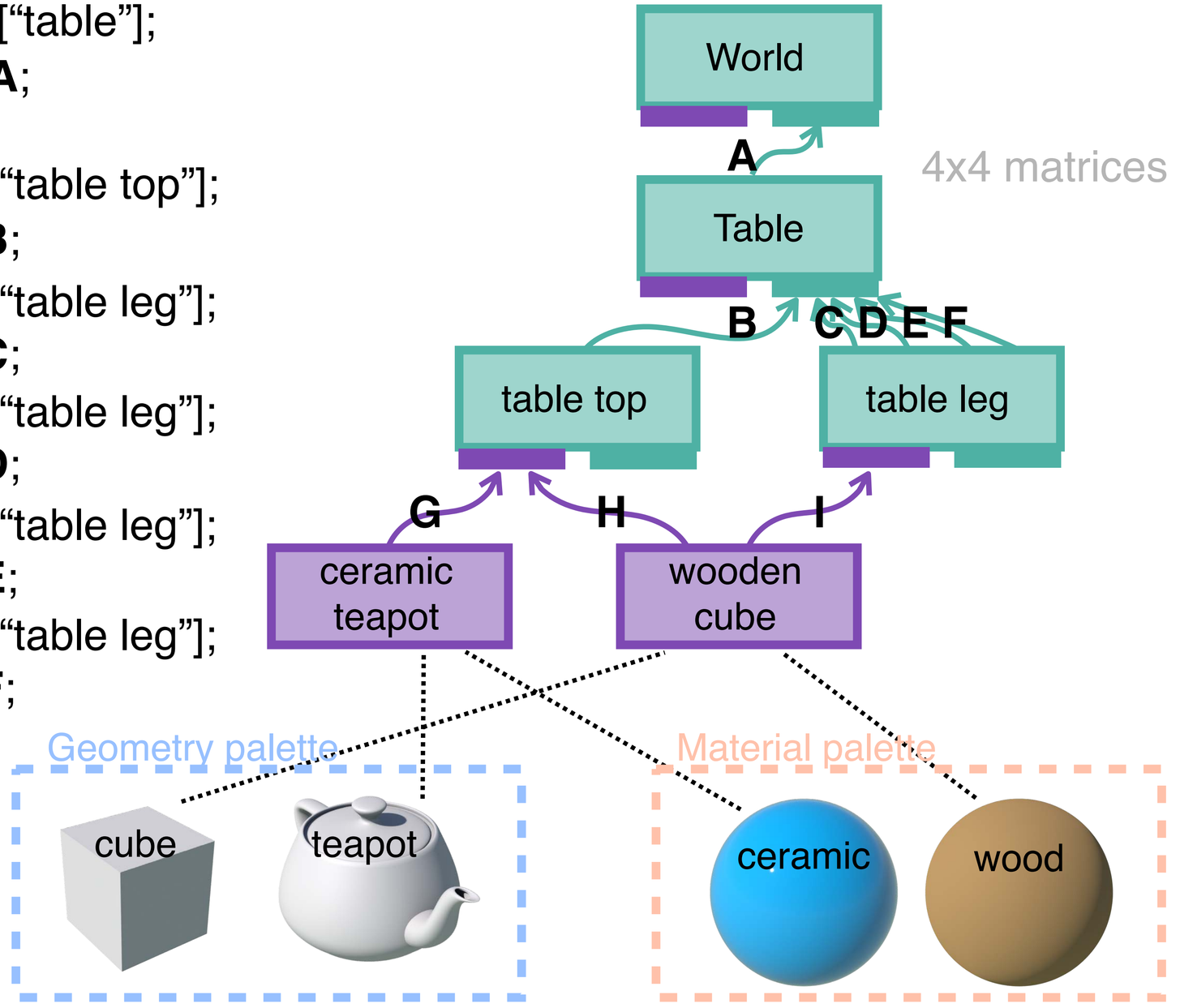
Example for setting up a scene graph

```
node[“world”] = new Node;
node[“table”] = new Node;
node[“table top”] = new Node;
node[“table leg”] = new Node;

node[“world”] -> childnodes[0] = node[“table”];
node[“world”] -> childtransforms[0] = A;
```



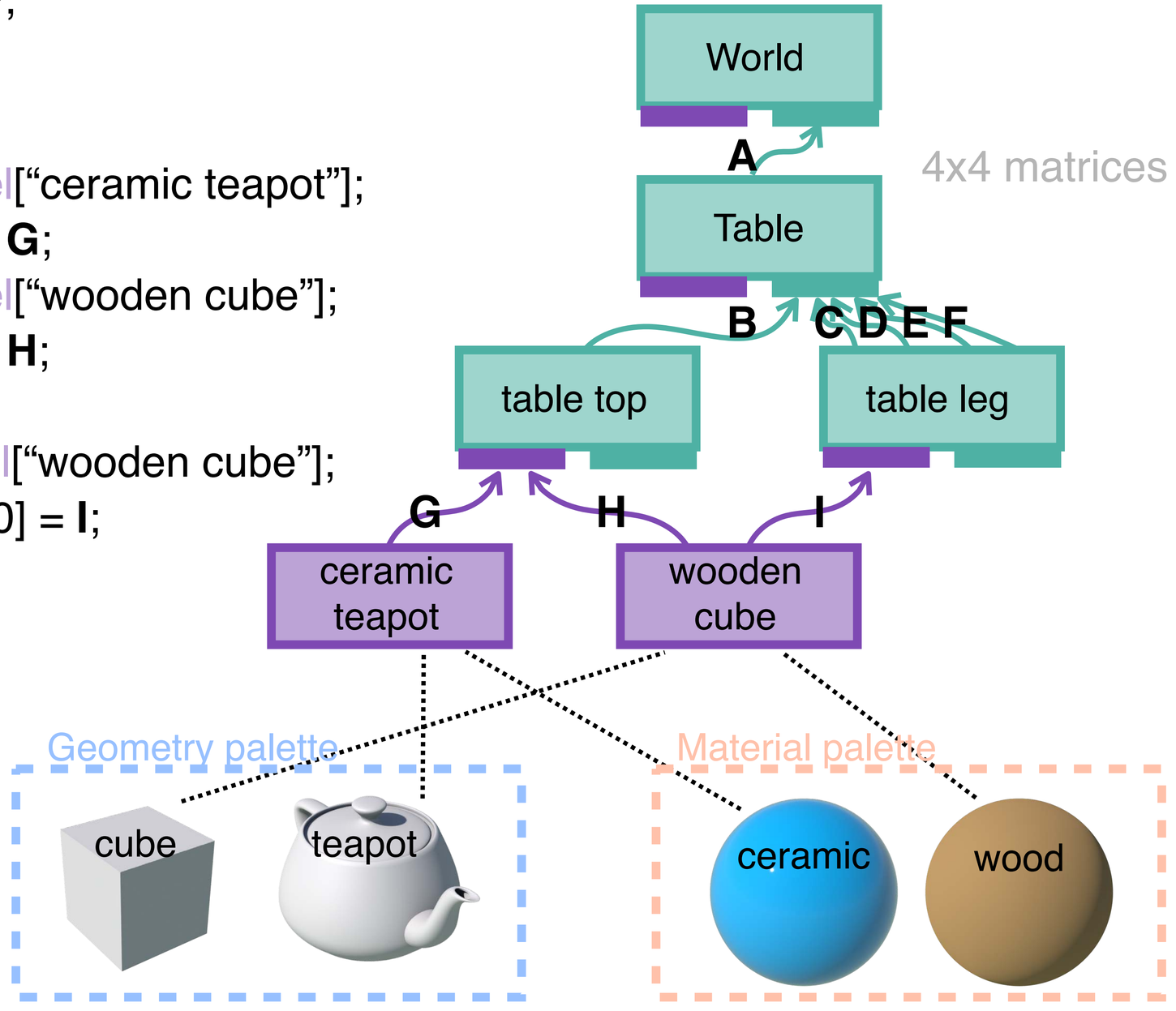
Example for setting up a scene graph

```
node[“world”] -> childnodes[0] = node[“table”];
node[“world”] -> childtransforms[0] = A;

node[“table”] -> childnodes[0] = node[“table top”];
node[“table”] -> childtransforms[0] = B;
node[“table”] -> childnodes[1] = node[“table leg”];
node[“table”] -> childtransforms[1] = C;
node[“table”] -> childnodes[2] = node[“table leg”];
node[“table”] -> childtransforms[2] = D;
node[“table”] -> childnodes[3] = node[“table leg”];
node[“table”] -> childtransforms[3] = E;
node[“table”] -> childnodes[4] = node[“table leg”];
node[“table”] -> childtransforms[4] = F;
```



Example for setting up a scene graph

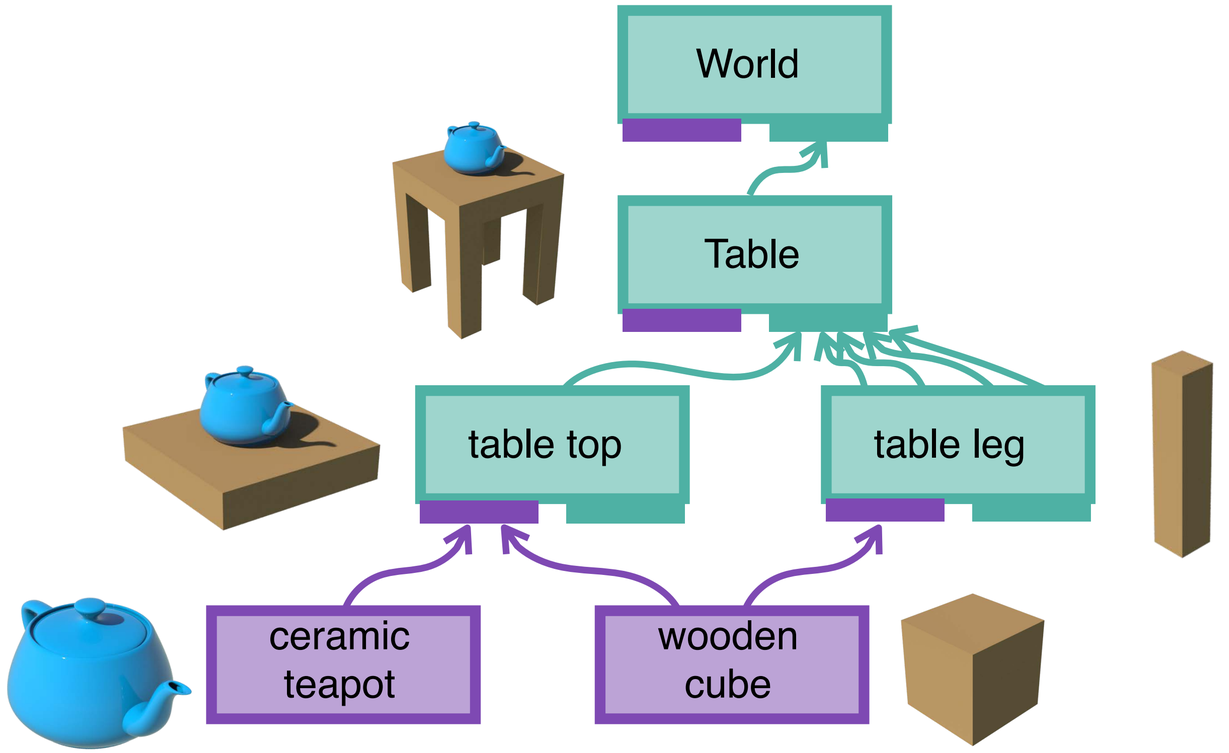
```
node["table"] -> childtransforms[4] = F;


node["table top"] -> models[0] = model["ceramic teapot"];
node["table"] -> modeltransforms[0] = G;
node["table top"] -> models[1] = model["wooden cube"];
node["table"] -> modeltransforms[1] = H;

node["table leg"] -> models[0] = model["wooden cube"];
node["table leg"] -> modeltransforms[0] = I;


};
```



Render the scene

Render the scene

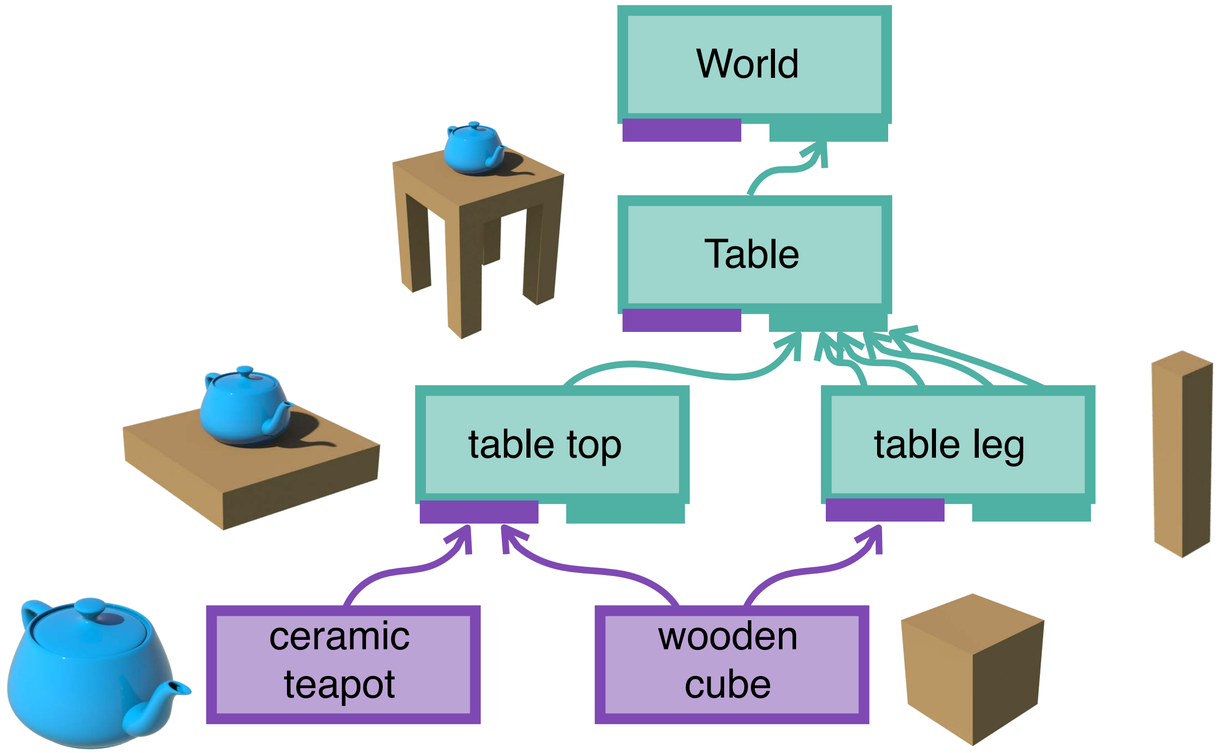
```
void Scene::draw(){

};
```



Render the scene

void Scene::draw(){

- std::stack<Node*> node_stack;
- std::stack<mat4> matrix_stack;
- Node* x = node[“World”];
- mat4 vm = camera’s view matrix;



Render the scene

void Scene::draw(){

- std::stack<Node*> node_stack;
- std::stack<mat4> matrix_stack;
- Node* x = node[“World”];
- mat4 vm = camera’s view matrix;

- Push x into the node_stack and vm into the matrix_stack.

While node_stack is nonempty

- x = node_stack.pop(); vm = matrix_stack.pop();
- Draw all models attached to x:
  - Set shader’s modelview to [vm*(matrix associated to the edge)];
  - Draw the model;
- Push each child node into the node_stack
  and correspondingly [vm*(matrix associated to the edge)] into the matrix_stack.

EndWhile

Part of your HW3 is to turn this into C++ code.