CSE 167 (FA21)
Computer Graphics:
Hierarchical Modeling
Albert Chern
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.*
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.
SideFX Houdini software

- Houdini Apprentice license is free
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
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

```glsl
#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

```c
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
  
  \[ VM_1 M_2 M_3 M_5 \quad VM_1 M_2 M_3 M_6 \quad VM_1 M_2 M_3 M_5 M_7 \quad \text{etc.} \]

  - 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 $\mathbb{T}$

\[
\text{std::stack}<\mathbb{T} > \ a
\]

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

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

together with the following three operations:

- push
- pop
- top
Stack

• push

\[
PUSH: \mathbb{T}^k \times \mathbb{T} \rightarrow \mathbb{T}^{k+1}
\]
\[
(a_1, \ldots, a_k).PUSH(b) = (a_1, \ldots, a_k, b)
\]

• pop

• top
Stack

• push

\[ \text{PUSH: } \mathbb{T}^k \times \mathbb{T} \rightarrow \mathbb{T}^{k+1} \]
\[ (a_1, \ldots, a_k).\text{PUSH}(b) = (a_1, \ldots, a_k, b) \]

• pop

\[ \text{POP: } \mathbb{T}^k \rightarrow \mathbb{T}^{k-1} \]
\[ (a_1, \ldots, a_{k-1}, a_k).\text{POP}() = (a_1, \ldots, a_{k-1}) \]

• top
• push
  \[ \text{PUSH: } \mathbb{T}^k \times \mathbb{T} \rightarrow \mathbb{T}^{k+1} \]
  \[ (a_1, \ldots, a_k).\text{PUSH}(b) = (a_1, \ldots, a_k, b) \]

• pop
  \[ \text{POP: } \mathbb{T}^k \rightarrow \mathbb{T}^{k-1} \]
  \[ (a_1, \ldots, a_{k-1}, a_k).\text{POP}() = (a_1, \ldots, a_{k-1}) \]

• top
  \[ \text{TOP: } \mathbb{T}^k \rightarrow \mathbb{T} \]
  \[ (a_1, \ldots, a_k).\text{TOP}() = a_k \]
Stack

- **push**
  \[ \text{PUSH}: \mathbb{T}^k \times \mathbb{T} \rightarrow \mathbb{T}^{k+1} \]
  \[(a_1, \ldots, a_k).\text{PUSH}(b) = (a_1, \ldots, a_k, b)\]

- **pop**
  \[ \text{POP}: \mathbb{T}^k \rightarrow \mathbb{T}^{k-1} \]
  \[(a_1, \ldots, a_{k-1}, a_k).\text{POP}() = (a_1, \ldots, a_{k-1})\]

- **top**
  \[ \text{TOP}: \mathbb{T}^k \rightarrow \mathbb{T} \]
  \[(a_1, \ldots, a_k).\text{TOP}() = a_k\]

*We only have access to the top of 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
Matrix Stack

- Define a **modelviewStack** $\text{STACK} \in \text{std::stack<glm::mat4>}$
- Let $\text{VM}$ be the current modelview matrix
- Initially $\text{VM} = V$
» Define a **modelviewStack** \( S \)

» Let \( VM \) be the current modelview matrix

» Initially \( VM = V \)

» Push \( S \). \( STACK.PUSH(VM) \)
Matrix Stack

- Define a `modelviewStack` STACK
- Let VM be the current modelview matrix
- Initially \( VM = V \)
- Push `STACK.push(VM)`
Matrix Stack

- Initially $\text{VM} = V$
- Push $\text{STACK}\cdot\text{PUSH}(\text{VM})$
- Update $\text{VM} = \text{VM} \star M_1$
Matrix Stack

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

\[ \begin{array}{c|c}
V & VM_1 \\
\hline
\text{STACK} & \text{VM} \\
\end{array} \]
Matrix Stack

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

\[
\text{VM} \Rightarrow \text{VM}_1
\]

- \( M_1 \): World to Room
- \( M_2 \): Room to Chair
- \( M_3 \): Room to Table
- \( M_4 \): Room to Book
- \( M_5 \): Room to Laptop

- Initially \( \text{VM} = V \)

**Stack**

- \( V \)
- \( \text{VM}_1 \)

**VM**
Matrix Stack

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

![Diagram showing the matrix stack with operations and transformations]
Matrix Stack

- Initially $VM = V$
- Push $STACK.PUSH(VM)$
- Update $VM = VM * M_1$
- drawRoom
- Push $STACK.PUSH(VM)$
Matrix Stack

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

Initially $\text{VM} = V$

Push $\text{STACK}.\text{push}($VM$)$

Update $\text{VM} = \text{VM} \ast M_1$

Push $\text{STACK}.\text{push}($VM$)$

Update $\text{VM} = \text{VM} \ast M_2$

$\text{VM}_1$

$V$

$\text{VM}_1$

$\text{STACK}$

$\text{VM}$

$\mathbb{P}^3_{\text{world}}$

$\mathbb{P}^3_{\text{room}}$

$\mathbb{P}^3_{\text{chair}}$

$\mathbb{P}^3_{\text{table}}$

$\mathbb{P}^3_{\text{book}}$

$\mathbb{P}^3_{\text{laptop}}$

$\mathbb{P}^3_{\text{camera}}$
Matrix Stack

- Initially \( VM = V \)
- Push \( \text{STACK}.\text{PUSH}(VM) \)
- Update \( VM = VM \times M_1 \)
- drawRoom
- Push \( \text{STACK}.\text{PUSH}(VM) \)
- Update \( VM = VM \times M_2 \)
Matrix Stack

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

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

\[
\begin{array}{c|c}
VM_1 & VM_1 M_2 \\
V & \\
\end{array}
\]

Stack VM
Matrix Stack

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

- Initially  $\text{VM} = V$
- Push  $\text{STACK.push}(\text{VM})$
- Update  $\text{VM} = \text{VM} \ast M_1$
- drawRoom
- Push  $\text{STACK.push}(\text{VM})$
- Update  $\text{VM} = \text{VM} \ast M_2$
- drawChair
- Pop  $\text{VM} = \text{STACK.pop}$
- Push  $\text{STACK.push}(\text{VM})$
- Push  $\text{STACK.push}(\text{VM})$
- Update  $\text{VM} = \text{VM} \ast M_3$
- drawChair
- Pop  $\text{VM} = \text{STACK.pop}$
- Push  $\text{STACK.push}(\text{VM})$
- Push  $\text{STACK.push}(\text{VM})$
- Update  $\text{VM} = \text{VM} \ast M_4$
- drawChair
- Push  $\text{STACK.push}(\text{VM})$
- Push  $\text{STACK.push}(\text{VM})$
- Update  $\text{VM} = \text{VM} \ast M_5$
- drawChair
Matrix Stack

- Initially: \( VM = V \)
- Push: \( \text{STACK}.\text{push}(VM) \)
- Update: \( VM = VM \times M_1 \)
- drawRoom
- Push: \( \text{STACK}.\text{push}(VM) \)
- Update: \( VM = VM \times M_2 \)
- drawChair
- Pop: \( VM = \text{STACK}.\text{pop} \)
- Push: \( \text{STACK}.\text{push}(VM) \)

![Diagram]

\( VM_1 \)

\( V \)

\( VM_1 \)

\( \text{STACK} \)

\( \text{VM} \)

\( \mathbb{P}^3_{\text{world}} \)

\( \mathbb{P}^3_{\text{room}} \)

\( \mathbb{P}^3_{\text{chair}} \)

\( \mathbb{P}^3_{\text{table}} \)

\( \mathbb{P}^3_{\text{book}} \)

\( \mathbb{P}^3_{\text{laptop}} \)

\( \mathbb{P}^3_{\text{camera}} \)
Matrix Stack

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

\[
\text{VM}_1
\]

\[
V
\]

\[
\text{STACK}
\]

\[
\text{VM}
\]

\[
M_1
\] → \( \mathbb{P}^3 \) _world_

\[
M_2
\] → \( \mathbb{P}^3 \) _room_

\[
M_3
\] → \( \mathbb{P}^3 \) _chair_

\[
M_4
\] → \( \mathbb{P}^3 \) _table_

\[
M_5
\] → \( \mathbb{P}^3 \) _book_

\[
\mathbb{P}^3 \] _laptop_

\[
V
\]
Matrix Stack

- Initially \( VM = V \)
- Push \( \text{STACK.push}(VM) \)
- Update \( VM = VM \times M_1 \)
- drawRoom
- Push \( \text{STACK.push}(VM) \)
- Update \( VM = VM \times M_2 \)
- drawChair
- Pop \( VM = \text{STACK.pop} \)
- Push \( \text{STACK.push}(VM) \)

- Update \( VM = VM \times M_3 \)

\[
\begin{array}{c|c|c}
\text{Stack} & VM & VM_1 M_3 \\
\hline
V & & \\
\end{array}
\]

- Initially \( VM = V \)
- Push \( \text{STACK.push}(VM) \)
- Update \( VM = VM \times M_1 \)
- drawRoom
- Push \( \text{STACK.push}(VM) \)
- Update \( VM = VM \times M_2 \)
- drawChair
- Pop \( VM = \text{STACK.pop} \)
- Push \( \text{STACK.push}(VM) \)

- Update \( VM = VM \times M_3 \)

\[
\begin{array}{c|c|c}
\text{Stack} & VM & VM_1 M_3 \\
\hline
V & & \\
\end{array}
\]

- Initially \( VM = V \)
- Push \( \text{STACK.push}(VM) \)
- Update \( VM = VM \times M_1 \)
- drawRoom
- Push \( \text{STACK.push}(VM) \)
- Update \( VM = VM \times M_2 \)
- drawChair
- Pop \( VM = \text{STACK.pop} \)
- Push \( \text{STACK.push}(VM) \)

- Update \( VM = VM \times M_3 \)
Matrix Stack

- Initially: $VM = V$
- Push: $STACK.push(VM)$
- Update: $VM = VM \times M_1$
- drawRoom
- Push: $STACK.push(VM)$
- Update: $VM = VM \times M_2$
- drawChair
- Pop: $VM = STACK.pop$
- Push: $STACK.push(VM)$

$VM_1 \quad VM_1M_3$

STACK VM

$M_1 \rightarrow P^3_{world}$
$M_2 \rightarrow P^3_{room}$
$M_3 \rightarrow P^3_{table}$
$M_4 \rightarrow P^3_{chair}$
$M_5 \rightarrow P^3_{book}$
$V \rightarrow P^3_{laptop}$
Matrix Stack

- **Initially** $VM = V$
- **Push** $STACK.push(VM)$
- **Update** $VM = VM * M_1$
- **drawRoom**
- **Push** $STACK.push(VM)$
- **Update** $VM = VM * M_2$
- **drawChair**
- **Pop** $VM = STACK.pop$
- **Push** $STACK.push(VM)$

$VM_1M_3$

$VM_1$

$V$

$VM_1M_3$

**STACK**

**VM**

$M_1$

$M_2$

$M_3$

$M_4$

$M_5$

$P^3_{world}$

$P^3_{camera}$

$P^3_{room}$

$P^3_{chair}$

$P^3_{table}$

$P^3_{book}$

$P^3_{laptop}$
Matrix Stack

- Initially $VM = V$
- Push $STACK.push(VM)$
- Update $VM = VM \cdot M_1$
- drawRoom
- Push $STACK.push(VM)$
- Update $VM = VM \cdot M_2$
- drawChair
- Pop $VM = STACK.pop$
- Push $STACK.push(VM)$
- Update $VM = VM \cdot M_3$
- Push $STACK.push(VM)$
- Update $VM = VM \cdot M_4$

![Diagram of Matrix Stack](image.png)
Matrix Stack

- Initially \( VM = V \)
- Push \( \text{STACK.push}(VM) \)
- Update \( VM = VM \times M_1 \)
- drawRoom
- Push \( \text{STACK.push}(VM) \)
- Update \( VM = VM \times M_2 \)
- drawChair
- Pop \( VM = \text{STACK.pop} \)
- Push \( \text{STACK.push}(VM) \)
- Update \( VM = VM \times M_3 \)
- Push \( \text{STACK.push}(VM) \)
- Update \( VM = VM \times M_4 \)

\[
\begin{align*}
VM_1M_3 & \\
VM_1 & \\
V & \\
VM_1M_3M_4
\end{align*}
\]

\[\text{STACK} \quad \text{VM}\]
Matrix Stack

- Initially: $VM = V$
- Push: STACK.push($VM$)
- Update: $VM = VM \times M_1$
- drawRoom
- Push: STACK.push($VM$)
- Update: $VM = VM \times M_2$
- drawChair
- Pop: $VM = STACK.pop$
- Push: STACK.push($VM$)
- Update: $VM = VM \times M_3$
- Push: STACK.push($VM$)
- Update: $VM = VM \times M_4$
- drawBook

Stack

<table>
<thead>
<tr>
<th>VM</th>
</tr>
</thead>
<tbody>
<tr>
<td>$VM_1 M_3$</td>
</tr>
<tr>
<td>$VM_1$</td>
</tr>
<tr>
<td>$V$</td>
</tr>
</tbody>
</table>

VM
Matrix Stack

- Initially \( VM = V \)
- Push \( \text{STACK.push}(VM) \)
- Update \( VM = VM \times M_1 \)
- drawRoom
- Push \( \text{STACK.push}(VM) \)
- Update \( VM = VM \times M_2 \)
- drawChair
- Pop \( VM = \text{STACK.pop} \)
- Push \( \text{STACK.push}(VM) \)
- Update \( VM = VM \times M_3 \)
- Push \( \text{STACK.push}(VM) \)
- Update \( VM = VM \times M_4 \)
- drawBook
- Pop \( VM = \text{STACK.pop} \)

\[
\begin{align*}
\text{STACK} & \quad VM \\
V & \quad VM_1 M_3 \\
VM_1 & \quad VM_1 M_3 \\
V & \quad VM_1 M_3 M_4
\end{align*}
\]
Matrix Stack

- Initially \( \text{VM} = V \)
- Push \( \text{STACK}.\text{push(VM)} \)
- Update \( \text{VM} = \text{VM} \times M_1 \)
- drawRoom
  - Push \( \text{STACK}.\text{push(VM)} \)
  - Update \( \text{VM} = \text{VM} \times M_2 \)
- drawChair
  - Pop \( \text{VM} = \text{STACK}.\text{pop} \)
  - Push \( \text{STACK}.\text{push(VM)} \)
- Update \( \text{VM} = \text{VM} \times M_3 \)
  - Push \( \text{STACK}.\text{push(VM)} \)
  - Update \( \text{VM} = \text{VM} \times M_4 \)
  - drawBook
  - Pop \( \text{VM} = \text{STACK}.\text{pop} \)

\[
\begin{array}{c|c|c}
\text{VM}_1 & V & \text{VM}_1M_3 \\
\hline
\text{STACK} & \text{VM} & \\
\end{array}
\]
Matrix Stack

- Initially \( VM = V \)
- Push \( \text{STACK.push}(VM) \)
- Update \( VM = VM \cdot M_1 \)
- drawRoom
- Push \( \text{STACK.push}(VM) \)
- Update \( VM = VM \cdot M_2 \)
- drawChair
- Pop \( VM = \text{STACK.pop} \)
- Push \( \text{STACK.push}(VM) \)

- Update \( VM = VM \cdot M_3 \)
- Push \( \text{STACK.push}(VM) \)
- Update \( VM = VM \cdot M_4 \)
- drawBook
- Pop \( VM = \text{STACK.pop} \)
- drawTable

Matrix Stack

<table>
<thead>
<tr>
<th>VM ( M_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
</tr>
<tr>
<td>VM ( M_1 M_3 )</td>
</tr>
</tbody>
</table>

Stack | VM
Matrix Stack

- Initially $\text{VM} = V$
- Push $\text{STACK.push}($VM$)$
- Update $\text{VM} = \text{VM} \times M_1$
- Draw Room
- Push $\text{STACK.push}($VM$)$
- Update $\text{VM} = \text{VM} \times M_2$
- Draw Chair
- Pop $\text{VM} = \text{STACK.pop}$
- Push $\text{STACK.push}($VM$)$
- Update $\text{VM} = \text{VM} \times M_3$
- Push $\text{STACK.push}($VM$)$
- Update $\text{VM} = \text{VM} \times M_4$
- Draw Book
- Pop $\text{VM} = \text{STACK.pop}$
- Draw Table
- Push $\text{STACK.push}($VM$)$
Matrix Stack

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

\[
\begin{array}{|c|}
\hline
VM_1 M_3 \\
VM_1 \\
V \\
\hline
\end{array}
\]

\[
\begin{array}{|c|}
\hline
\text{STACK} \\
VM \\
\hline
\end{array}
\]
Matrix Stack

- Initially $VM = V$
- Push $STACK.push(VM)$
- Update $VM = VM \times M_1$
- drawRoom
- Push $STACK.push(VM)$
- Update $VM = VM \times M_2$
- drawChair
- Pop $VM = STACK.pop$
- Push $STACK.push(VM)$
- Update $VM = VM \times M_3$
- Push $STACK.push(VM)$
- Update $VM = VM \times M_4$
- drawBook
- Push $STACK.push(VM)$
- Pop $VM = STACK.pop$
- drawTable
- Push $STACK.push(VM)$
- Update $VM = VM \times M_5$
Matrix Stack

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

\[ \begin{array}{c|c}
VM_1M_3 & V \\
VM_1 & \hline
V & VM_1M_3M_5
\end{array} \]
Matrix Stack

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

\[
\begin{array}{c|c|c}
\text{STACK} & \text{VM} \\
\hline
VM_1M_3 & V \\
VM_1 & VM_1M_3M_5 \\
V & \\
\end{array}
\]
Matrix Stack

- Initially: $VM = V$
- Push: $STACK.push(VM)$
- Update: $VM = VM \times M_1$
- drawRoom
- Push: $STACK.push(VM)$
- Update: $VM = VM \times M_2$
- drawChair
- Pop: $VM = STACK.pop$
- Push: $STACK.push(VM)$
- Update: $VM = VM \times M_3$
- drawLaptop
- Push: $STACK.push(VM)$
- Update: $VM = VM \times M_4$
- drawTable
- Push: $STACK.push(VM)$
- Update: $VM = VM \times M_5$
- drawBook
- Pop: $VM = STACK.pop$
- Push: $STACK.push(VM)$
- Pop: $VM = STACK.pop$

$$VM_1M_3$$

$$VM_1$$

$$V$$

$$VM_1M_3M_5$$

STACK

VM

**Diagram:**
- $V$
- $M_1$
- $M_2$
- $M_3$
- $M_4$
- $M_5$
- $P_3^{world}$
- $P_3^{camera}$
- $P_3^{book}$
- $P_3^{laptop}$
- $P_3^{table}$
- $P_3^{chair}$
- $P_3^{room}$

The diagram shows the process of updating the matrix stack with various operations.
Matrix Stack

- Initially $VM = V$
- Push $STACK.push(VM)$
- Update $VM = VM \times M_1$
- drawRoom
- Push $STACK.push(VM)$
- Update $VM = VM \times M_2$
- drawChair
- Pop $VM = STACK.pop$
- Push $STACK.push(VM)$
- Update $VM = VM \times M_3$
- drawRoom
- Pop $VM = STACK.pop$
- Push $STACK.push(VM)$
- Update $VM = VM \times M_4$
- drawBook
- Push $STACK.push(VM)$
- Update $VM = VM \times M_5$
- drawLaptop
- Pop $VM = STACK.pop$

$VM_1$

$V$

$VM_1M_3$

Stack

VM
Matrix Stack

Initially $\text{VM} = V$
- Push $\text{STACK}.\text{push} (\text{VM})$
- Update $\text{VM} = \text{VM} \times M_1$
- $\text{drawRoom}$
- Push $\text{STACK}.\text{push} (\text{VM})$
- Update $\text{VM} = \text{VM} \times M_2$
- $\text{drawChair}$
- Pop $\text{VM} = \text{STACK}.\text{pop}$
- Push $\text{STACK}.\text{push} (\text{VM})$
- Update $\text{VM} = \text{VM} \times M_3$
- Update $\text{VM} = \text{VM} \times M_4$
- Pop $\text{VM} = \text{STACK}.\text{pop}$
- drawTable
- Push $\text{STACK}.\text{push} (\text{VM})$
- Update $\text{VM} = \text{VM} \times M_5$
- drawLaptop
- Pop $\text{VM} = \text{STACK}.\text{pop}$
- Pop $\text{VM} = \text{STACK}.\text{pop}$

STACK

VM

$\text{VM}_1$

$V$

$\text{VM}_1 M_3$
Matrix Stack

- Initially \( \text{VM} = V \)
- Push \( \text{STACK}.\text{push}(\text{VM}) \)
- Update \( \text{VM} = \text{VM} \times M_1 \)
- drawRoom
- Push \( \text{STACK}.\text{push}(\text{VM}) \)
- Update \( \text{VM} = \text{VM} \times M_2 \)
- drawChair
- Pop \( \text{VM} = \text{STACK}.\text{pop} \)
- Push \( \text{STACK}.\text{push}(\text{VM}) \)
- Update \( \text{VM} = \text{VM} \times M_3 \)
- Update \( \text{VM} = \text{VM} \times M_4 \)
- drawBook
- Push \( \text{STACK}.\text{push}(\text{VM}) \)
- drawTable
- Push \( \text{STACK}.\text{push}(\text{VM}) \)
- Update \( \text{VM} = \text{VM} \times M_5 \)
- drawLaptop
- Pop \( \text{VM} = \text{STACK}.\text{pop} \)
- Pop \( \text{VM} = \text{STACK}.\text{pop} \)

\[
\begin{array}{c|c}
\text{STACK} & \text{VM} \\
\hline
V & VM_1 \\
\end{array}
\]
Matrix Stack

- Initially, $\text{VM} = V$
- Push $\text{STACK.push(VM)}$
- Update $\text{VM} = \text{VM} \times M_1$
- drawRoom
- Push $\text{STACK.push(VM)}$
- Update $\text{VM} = \text{VM} \times M_2$
- drawChair
- Pop $\text{VM} = \text{STACK.pop}$
- Push $\text{STACK.push(VM)}$
- Update $\text{VM} = \text{VM} \times M_3$
- drawLaptop
- Push $\text{STACK.push(VM)}$
- Update $\text{VM} = \text{VM} \times M_4$
- drawBook
- Pop $\text{VM} = \text{STACK.pop}$
- drawTable
- Push $\text{STACK.push(VM)}$
- Update $\text{VM} = \text{VM} \times M_5$
- drawBook
- Pop $\text{VM} = \text{STACK.pop}$
- Pop $\text{VM} = \text{STACK.pop}$

$V$

<table>
<thead>
<tr>
<th>VM</th>
<th>VM_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>STACK</td>
<td>VM</td>
</tr>
</tbody>
</table>
Matrix Stack

- Initially: $\text{VM} = \text{V}$
- Push: $\text{STACK}.\text{push}($VM$)$
- Update: $\text{VM} = \text{VM} \times M_1$
- drawRoom
- Push: $\text{STACK}.\text{push}($VM$)$
- Update: $\text{VM} = \text{VM} \times M_2$
- drawChair
- Pop: $\text{VM} = \text{STACK}.\text{pop}$
- Push: $\text{STACK}.\text{push}($VM$)$
- Update: $\text{VM} = \text{VM} \times M_3$
- drawLaptop
- Push: $\text{STACK}.\text{push}($VM$)$
- Update: $\text{VM} = \text{VM} \times M_4$
- drawTable
- Push: $\text{STACK}.\text{push}($VM$)$
- Update: $\text{VM} = \text{VM} \times M_5$
- drawBook
- Pop: $\text{VM} = \text{STACK}.\text{pop}$
- Pop: $\text{VM} = \text{STACK}.\text{pop}$
- Pop: $\text{VM} = \text{STACK}.\text{pop}$
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
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”)

  \[
  \text{A B E C D F}
  \]

  Put A into a queue Q.
  \textbf{While} (Q is nonempty)
  \[
  x = \text{Q.dequeue()};
  \]
  process/visit x;
  Put neighbors of x into Q;
  \textbf{EndWhile}

- Depth first search (DFS)
  (Prioritize “exploring” over “overviewing”)

  \[
  \text{A B C D F E}
  \]
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
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**
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
Any directed graph (such as shown on the right) has a **reachability relation** on the node set:

\[ \text{node}_1 \leq \text{node}_2 \]

if there exists a path traveling from \( \text{node}_1 \) to \( \text{node}_2 \).

\[
\begin{align*}
B \leq D \\
D \leq B & \quad C \not\leq E \\
B \not\leq E & \quad E \not\leq C
\end{align*}
\]
A directed graph is acyclic if the reachability relation $\leq$ becomes a partial ordering. That is,

$$X \leq Y \quad \& \quad Y \leq X \quad \implies \quad X = Y$$

If you like this sort of stuff, check out the lecture note.
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 \( i=1,\ldots,k \).

- Whenever we draw models in \( x \), we have the correct modelview matrix ready.
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.
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 Material{
    vec4 ambient;
    vec4 diffuse;
    vec4 specular;
    vec4 emission;
    float shininess;
};

class Geometry{
    Geometry* geometry;
    Material* material;
};

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

struct Node{
    std::vector<Node*> childnodes;
    std::vector<mat4> childtransforms;
    std::vector<Models*> models;
    std::vector<mat4> modeltransforms;
};
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>
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
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

```cpp
void Scene::init(){
}
```
void Scene::init(){

    geometry[“cube”] = new Cube; // Cube and Teapot are subclasses of Geometry
    geometry[“cube”] -> init();
    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 = …;
    …

    // Cube and Teapot are subclasses of Geometry
}

Example for setting up a scene graph
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

```javascript
node["world"] = new Node;
node["table"] = new Node;
node["table top"] = new Node;
node["table leg"] = new Node;

node["world"] -> childnodes[0] = node["table"];  //A
node["world"] -> childtransforms[0] = A;
```
Example for setting up a scene graph

```javascript
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"];  
```
Example for setting up a scene graph

```plaintext

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
void Scene::draw()
{
}

void Scene::draw(){
    std::stack<Node*> node_stack;
    std::stack<mat4> matrix_stack;
    Node* x = node[“World”];
    mat4 vm = camera’s view matrix;
}
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.
    }
}

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