Assignment 3

Michael Kazhdan

(601.457/657)

[Adapted from Diego Salume and Peizhao Li]

Outline

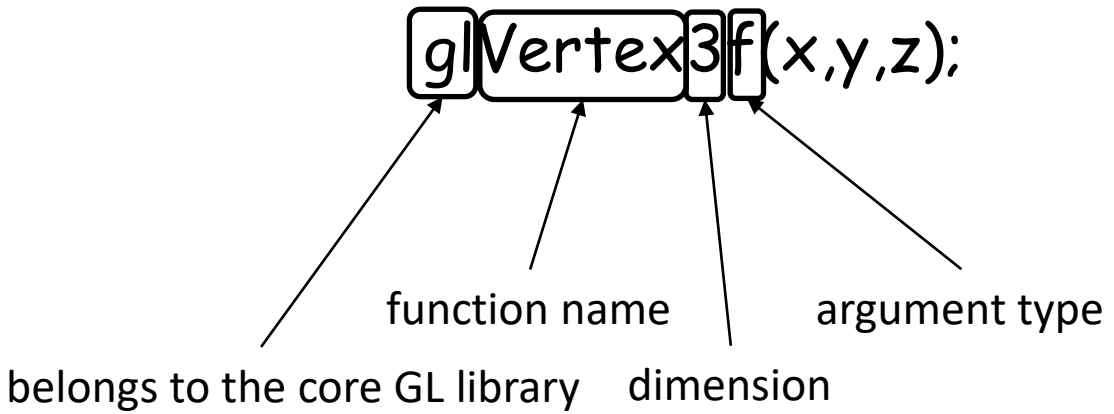
- OpenGL in General
  - Getting Started
  - Matrix Manipulation
  - Drawing
  - Lighting
  - Materials
  - Textures
- Assignment 3

What is OpenGL

- OpenGL is a platform-independent API
- Related utility libraries
  - OpenGL core
  - OpenGL Utility Library (GLU):
    higher-level drawing routines.
  - OpenGL Utility Toolkit (GLUT):
    I/O with host OS.

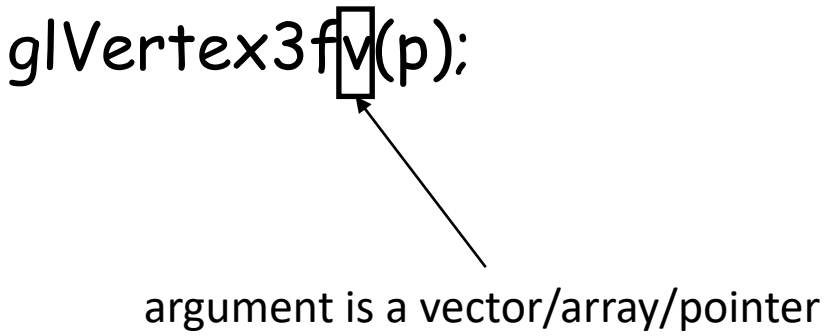
What is **OpenGL**

- OpenGL Core (GL: Graphics Library)
  - Mostly low-level graphics functions.
  - Create vertices, manipulate matrices, position cameras, etc.
  - Prefixed with "*gl*", e.g. *glVertex()*, *glBegin()*



What is **OpenGL**

- OpenGL Core (GL: Graphics Library)
  - Mostly low-level graphics functions.
  - Create vertices, manipulate matrices, position cameras, etc.
  - Prefixed with "*gl*", e.g. *glVertex()*, *glBegin()*



What is **OpenGL**

- OpenGL Core (GL: Graphics Library)
  - Mostly low-level graphics functions.
  - Create vertices, manipulate matrices, position cameras, etc.
  - Prefixed with "*gl*", e.g. *glVertex()*, *glBegin()*

```
glVertex3fv(p);
```

Note:
OpenGL was developed in C, before overloading:

- Function names reflect input types.
- May use **void *** pointers and require the user to specify the type.

What is OpenGL

- GLU (OpenGL Utility Library)
  - Slightly higher level.
  - Prefixed with "glu", e.g. gluSphere(), gluLookAt()
  - Draws more complex shapes, e.g. cylinders, spheres, cones.
  - Manages tessellation
- GLUT (OpenGL Utility Toolkit)
  - Handles windowing API
  - OS-independent
  - Functions prefixed with "glut", e.g. glutCreateWindow(), glutPostRedisplay()

Getting Started

Structure of a basic OpenGL program:

- Create a window
- Initialize OpenGL states:
  - Lighting, camera, etc.
- Per frame display function
  - e.g. set local transformations, clear buffers, draw, swap buffers, etc.
- Handle UI
  - Keyboard, mouse input
- Loop: keep calling display function

Getting Started

```
void main( int argc , char *argv[] )
{
        glutInit( &argc , argv );
        glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGB );
        glutInitWindowSize( 500 , 500 );
        glutInitWindowPosition( 100 , 100 );
        glutCreateWindow( argv[0] );
        glutIdleFunc( myIdle );
        glutDisplayFunc( myDisplay );
        glutReshapeFunc( myReshape );
        glutKeyboardFunc( myKeyboard );
        glutMouseFunc( myMouse );
        // do set-up here
        glutMainLoop();
}
```

1. Initialize GLUT
2. Set up display window
3. Register drawing/UI call-back functions
4. Do whatever needs to be done
5. Start the rendering loop

Getting Started

- OpenGL is a *state machine*:
  Once the value of a property is set, the value persists until a new value is set.
- Set properties via function calls, e.g. position/types of lights, the current material, the current modeling transformation, etc.
- When you make a call to draw some geometry, you enter a draw state, draw some geometry and then exit the draw state.

```
glBegin( GL_TRIANGLES );
...
glVertex3d( ... );
glEnd();
```

Matrix Manipulation

We need matrices to define transformations

- Matrices are stored on one of three stacks:
  - GL_PROJECTION (for camera-to-window transformations)
  - GL_MODELVIEW (for model-to-world / world-to-camera transformations)
  - GL_TEXTURE (won't be using)
- The current matrix is the top of the stack
- Initialized to the identity matrix

Matrix Manipulation (Modifying the Stack)

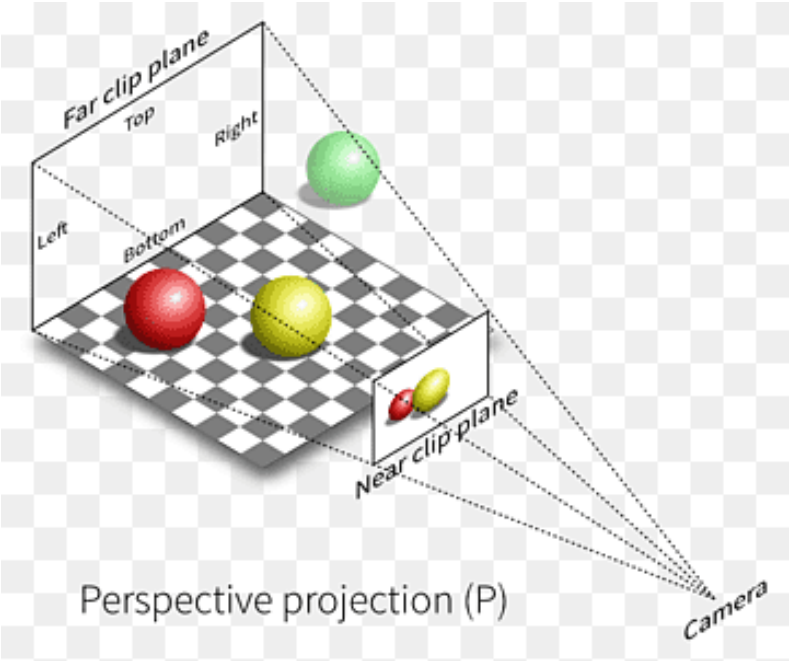
- Select matrix stack:
  - glMatrixMode(GL_PROJECTION | GL_MODELVIEW | GL TEXTURE);
- Add to/remove from the stack
  - glPushMatrix();
  - glPopMatrix();
- Overwrite top of the stack
  - glLoadIdentity();
  - glLoadMatrix(…);
- Modify top of the stack
  - glMultMatrix(…);
  - glTranslate(…), glRotate(…), glScale(…);

If S is the matrix on the stack and M is the argument of these functions, the new matrix on the stack will be $S \cdot M$.

Matrix Manipulation

glMatrixMode(GL_PROJECTION);

- Describes the camera-to-window transformation
- Usually only set it once.
- Helper functions:
  - gluPerspective(…);
  - glOrtho(…);



This set-up of the projection using gluPerspective is handled for you in `window.cpp`.

Image courtesy of: https://www.pngwing.com/en/free-png-ipsph

Matrix Manipulation

glMatrixMode(GL_MODELVIEW);

- Describes positions of objects relative to the camera
  - Camera:
    - gluLookAt(eyeXYZ , centerXYZ , upXYZ);
  - Object transformations:
    - glTranslatef(…)
    - glScalef(…)
    - glRotatef(…)

Note:
The matrix defined by gluLookAt multiplies the current matrix on the top of the stack. It does not replace it.
(If you want to replace it, you need to set it to the identity first.)

Recall:
Operations like glTranslatef, act on the matrix at the top of the stack by multiplication on the right.

Drawing

Primitives are drawn by declaring vertices:

```
glBegin( <primitive type> );
...
glVertex3f(...);
glEnd();
```

Can also declare normal, color, and texture info for each vertex.

- glNormal(…), glColor(…), glTexCoords(…)
- This has to be done **within** the glBegin(…)/glEnd() context, **before** specifying the vertex position with glVertex__(…);

Drawing

Primitive types for glBegin(…):

- GL_POINTS
- GL_LINES
- GL_TRIANGLES
- GL_TRIANGLE_STRIP
- GL_QUADS
- GL_POLYGON

Drawing more complex primitives using GLU:

- gluCylinder(…)
- gluDisk(…)
- gluSphere(…)

You are not allowed to use these for your assignment

Lighting

glLightfv(GL_LIGHT<N>, <param>, <val>);

- At least eight lights, GL_LIGHT0, GL_LIGHT1…
- Parameters include:
  - GL_POSITION (in homogenuous coordinates)
  - GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR
  - GL_CONSANT_ATTENUATION, GL_LINEAR_ATTENUATION, GL_QUADRATIC_ATTENUATION
  - GL_SPOT_EXPONENT, GL_SPOT_CUTOFF

Materials

glMaterialfv(<side>, <param> , <val>);

- Can specify materials separately for front and back: GL_FRONT , GL_BACK
- Parameters include:
  - GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_EMISSION, GL_SHININESS

- Set the material parameters then draw.

Textures

To use a texture you need to:

- Have a description of the texture and its properties on the GPU
- Have a handle (like a pointer, but represented as an unsigned integer) to the texture information on the GPU
- Enable texture mapping
- Specify which texture you will be using

Textures

To get a texture handle, ask OpenGL to generate one for you:

```
glGenTextures( GLsizei num , GLuint *tHandlePointer );
```

- The first argument gives the number of handles you want OpenGL to generate.
- The second argument is the memory address of the array where OpenGL should write the handles. (Needs to be pre-allocated.)

Typically, you will generate one texture at a time.

Textures

To specify that a particular texture handle should be active:

```
glBindTexture( GLenum target , GLuint tHandle );
```

- The first argument is an enumerator describing the texture type (e.g. GL_TEXTURE_1D, GL_TEXTURE_2D, or GL_TEXTURE_3D)
- The second is the handle itself.

Note that you need to do this both when rendering the texture and (initially) when specifying the texture properties (so OpenGL know which texture's properties you're setting).

Textures

To specify the texture information you need to specify how the texture gets mapped and what the texture values are:

```
glTexParameteri( <target> , <parmeter name> , <parameter value> );
glTexEnvi( <texture> , <parameter name> , <parameter value> );
glTexImage2D( ... , <texture data pointer> );
```

Textures

To specify the texture information you need to

- Specify the parameters of the texture:

  `glTexParameteri( GLenum target , GLenum pName , GLint pValue );`

  - Is the texture periodic?
  - How should the texture be filtered?

Textures

To specify the texture information you need to

- Describe how the texture gets applied:

  glTexEnvi(GLenum target , GLenum pName , GLint pValue);

  - Should it be applied as a decal or as a modulation?

Textures

To specify the texture information you need to

- Provide the texture how the texture gets applied:

```
glTexImage2D( ... , const void *data );
```

  - What types of texture are you using?
    - How many channels
    - How are the channels being stored
  - What is the texture size?
  - What is the texture data?

Textures

To enable texture mapping you need to tell OpenGL that you will be using (2D) texture mapping, and which handle you will be using:

```
glEnable( GL_TEXTURE_2D );
glBindTexture( GL_TEXTURE_2D , tHandle )
```

Overview

- OpenGL in General
- Assignment 3
  - General Structure
  - The Matrix Stack
  - Drawing Primitives
  - Textures

Assignment 3 (General Structure)

Window::DisplayFunction:

At every frame, the code redraws the display window by invoking Window::DisplayFunction.
This function:

1. Clears the buffers
2. Sets up the projection matrix
3. Invokes Scene::drawOpenGL

This is taken care of for you.

Assignment 3 (General Structure)

Scene::drawOpenGL:

When invoked, this function

1. Draws the camera (Camera::drawOpenGL)
2. Draws the lights (Light::drawOpenGL)
3. Draws the geometry (SceneGeometry::drawOpenGL)
    - Which draws the shapes (ShapeList::drawOpenGL)

The calls to these functions are made for you, but you have to implement them.

Assignment 3 (Matrix Stack)

In OpenGL, the model-to-camera transformation are stored in the *GL_MODELVIEW* matrix stack.

The matrix at the top of the stack describes the transformation applied to subsequently drawn geometry (vertex positions and normal) to bring it into the camera's coordinate system.

Assignment 3 (Matrix Stack)

In OpenGL, the model-to-camera transformation are stored in the *GL_MODELVIEW* matrix stack.

You can push/pop matrices onto the stack:

- *glPushMatrix*: Pushes the matrix at the top of the stack onto the matrix stack (increasing the stack size by one)
- *glPopMatrix*: Pops the top matrix off of the stack

Assignment 3 (Matrix Stack)

In OpenGL, the model-to-camera transformation are stored in the *GL_MODELVIEW* matrix stack.

You can set the matrix at the top of the stack by:

- *glLoadMatrix**: sets it to the prescribed matrix
- *glLoadIdentity*: sets it to the identity matrix

Assignment 3 (Matrix Stack)

In OpenGL, the model-to-camera transformation are stored in the *GL_MODELVIEW* matrix stack.

You can multiply the matrix at the top of the stack **on the right** by:

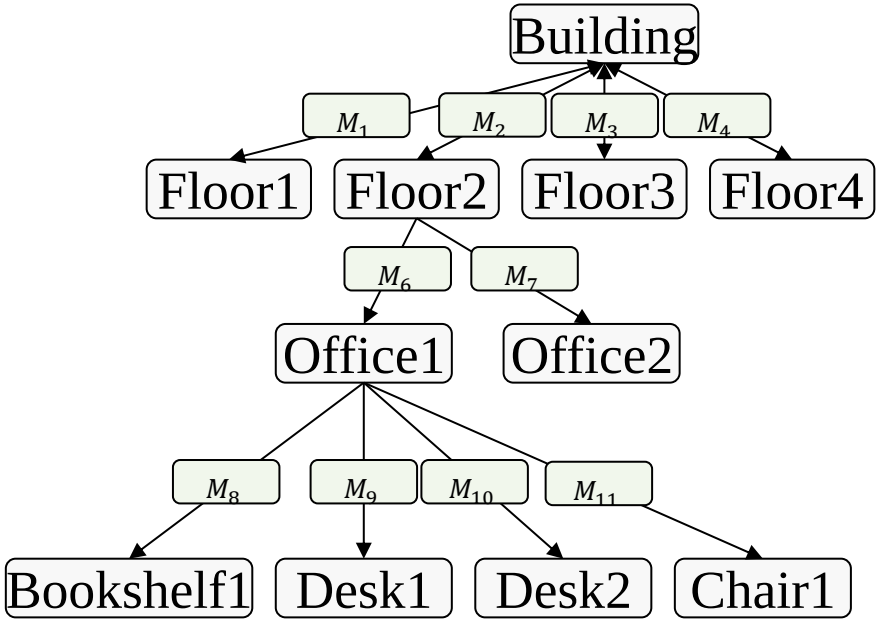
- *glMultMatrix**: multiplies by the prescribed matrix
- *glTranslate*/glRotate**: multiplies by the matrix describing the translation/rotation
- *gluLookAt*: multiplies by the matrix describing a camera with the prescribed orientation

Assignment 3 (Matrix Stack)

Scene Graphs (Recall):

In a scene graph, edges are tagged with a local-to-global transformations.

The transformation taking a node in a scene graph into global coordinates is the composition of the transformations from the root to the node.



Assignment 3 (Matrix Stack)

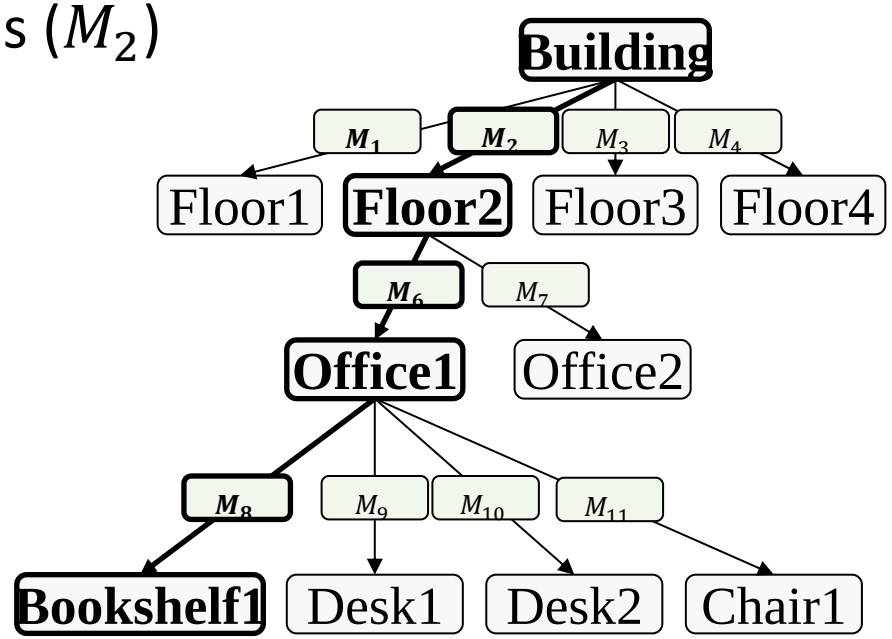
Scene Graphs (Recall):

To transform *Bookshelf1* into the *Building*'s coordinate system we:

- Transform it into *Office*1's coordinates (M_8), then
- Transform it into *Floor2*'s coordinates (M_6), then
- Transform it into *Building*'s coordinates (M_2)

This gives:

$$M_2 \circ M_6 \circ M_8$$



Assignment 3 (Matrix Stack)

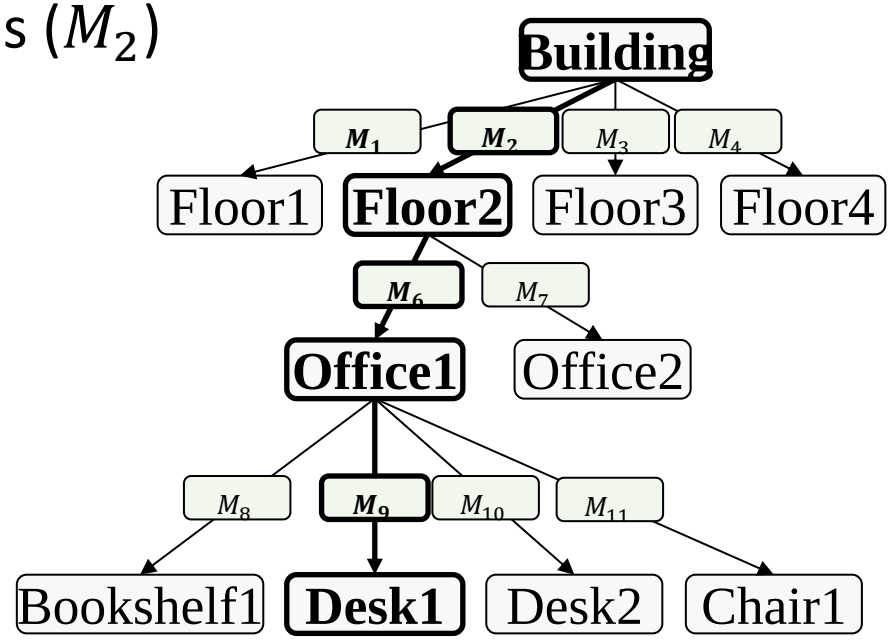
Scene Graphs (Recall):

To transform *Desk1* into the *Building*'s coordinate system we:

- Transform it into *Office*1's coordinates (M_9), then
- Transform it into *Floor2*'s coordinates (M_6), then
- Transform it into *Building*'s coordinates (M_2)

This gives:

$$M_2 \circ M_6 \circ M_9$$



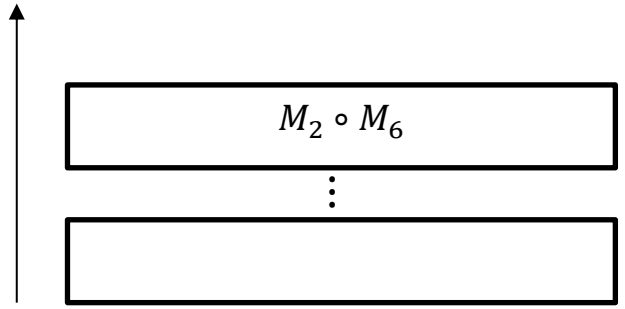
Assignment 3 (Matrix Stack)

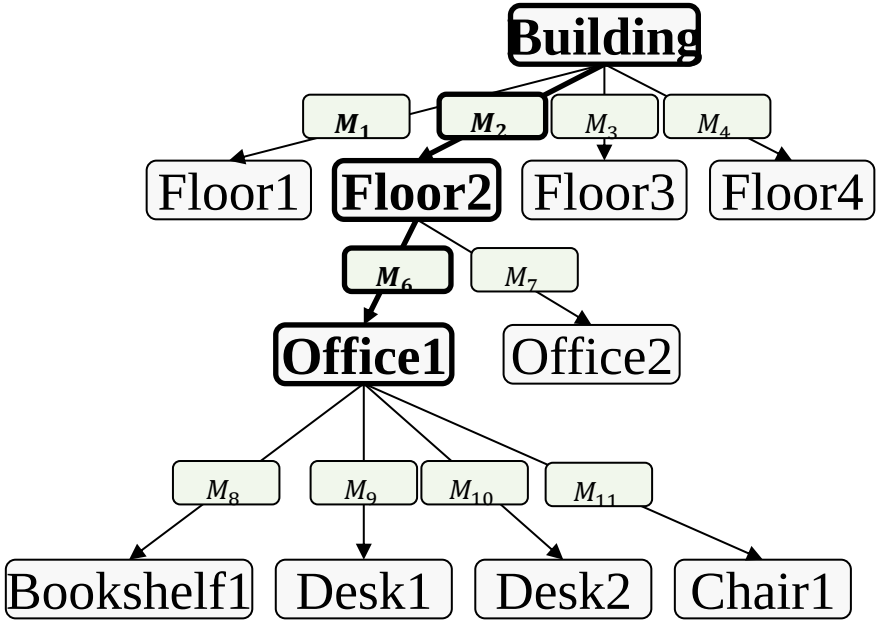
Scene Graphs (Recall):

When drawing *Office1*, we have $M_2 \circ M_6$ at the top of the stack.

Before drawing *Bookshelf1*, we want $M_2 \circ M_6 \circ M_8$ at the top of the stack.

Before drawing *Desk1*, we want $M_2 \circ M_6 \circ M_9$ at the top of the stack.



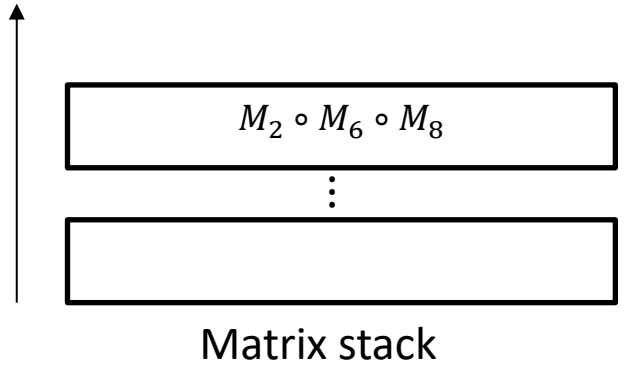


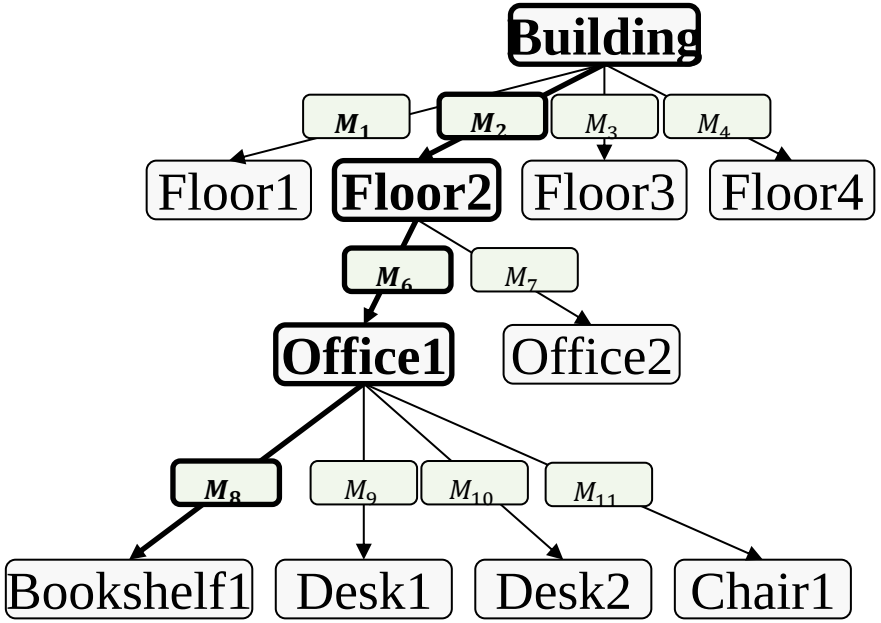
Assignment 3 (Matrix Stack)

Scene Graphs (Recall):

We could:

- Multiply on the right by M_8



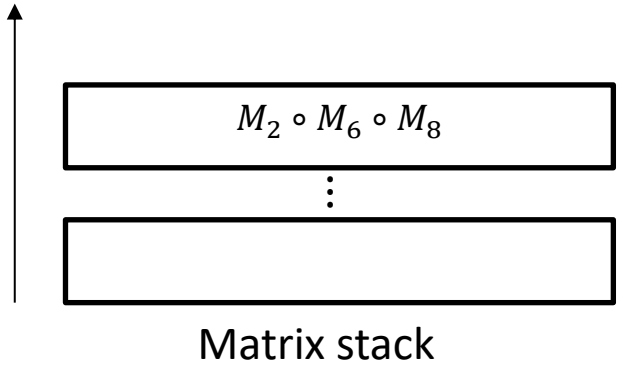


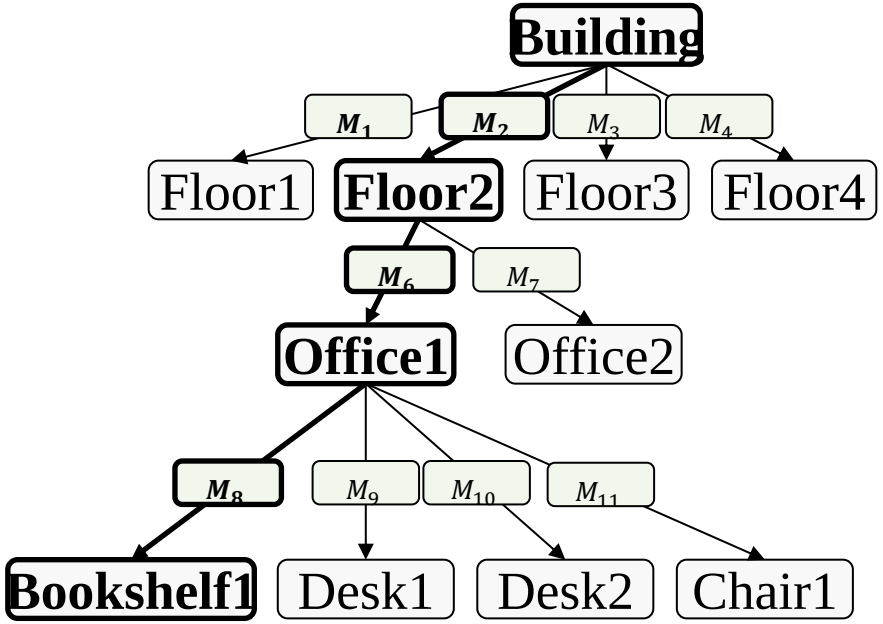
Assignment 3 (Matrix Stack)

Scene Graphs (Recall):

We could:

- Multiply on the right by M_8
- Draw *Bookshelf1*



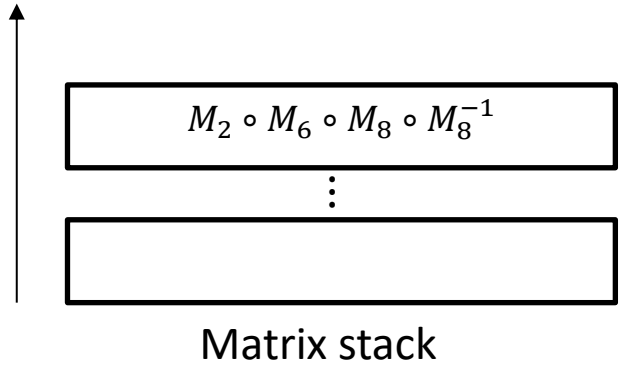


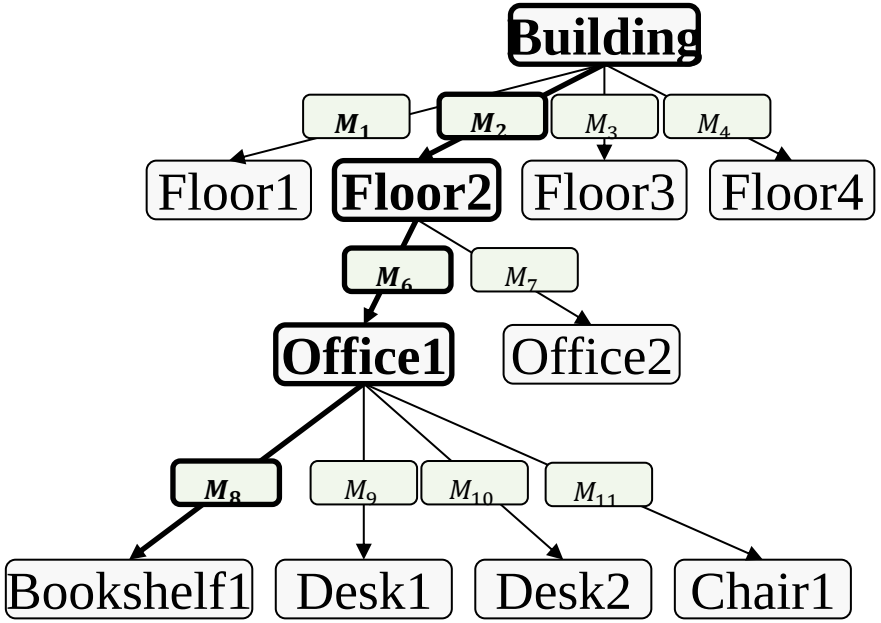
Assignment 3 (Matrix Stack)

Scene Graphs (Recall):

We could:

- Multiply on the right by M_8
- Draw *Bookshelf1*
- Multiply on the right by M_8^{-1}



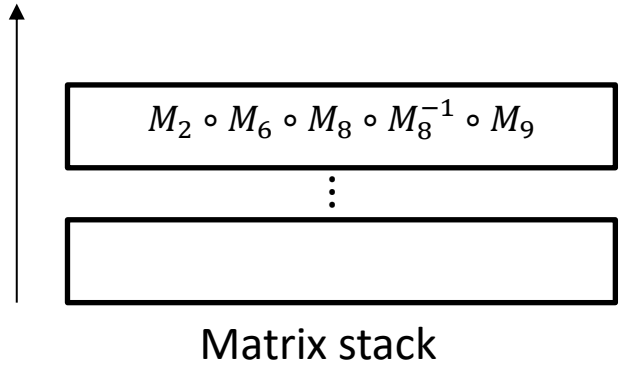


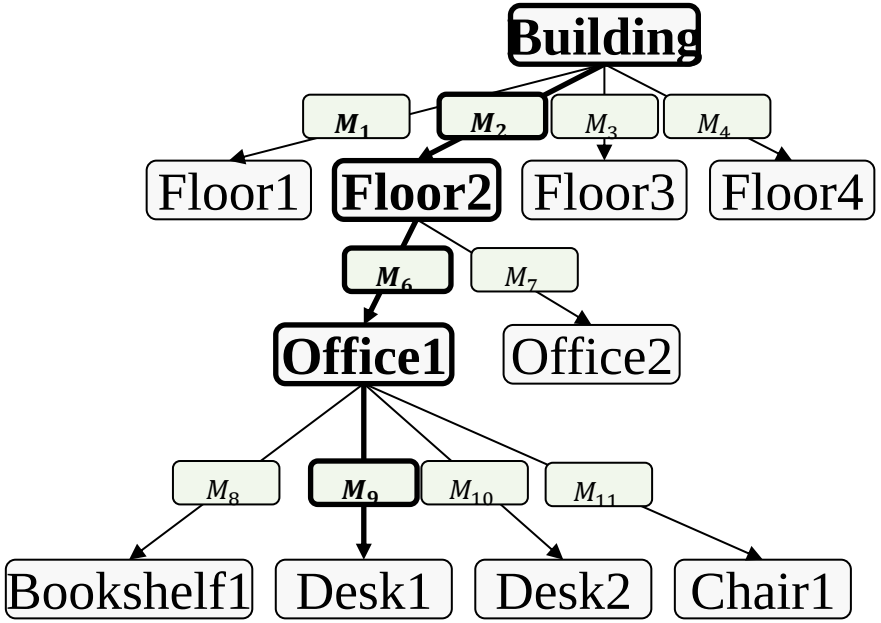
Assignment 3 (Matrix Stack)

Scene Graphs (Recall):

We could:

- Multiply on the right by M_8
- Draw *Bookshelf1*
- Multiply on the right by M_8^{-1}
- Multiply on the right by M_9



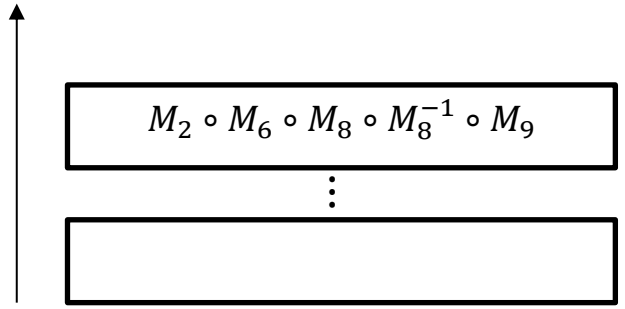


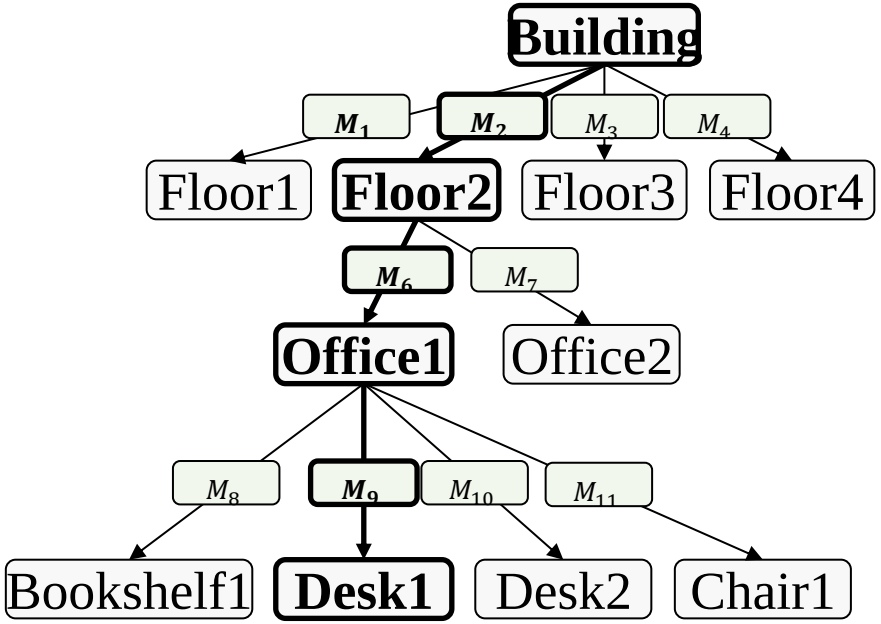
Assignment 3 (Matrix Stack)

Scene Graphs (Recall):

We could:

- Multiply on the right by M_8
- Draw *Bookshelf1*
- Multiply on the right by M_8^{-1}
- Multiply on the right by M_9
- Draw *Desk1*



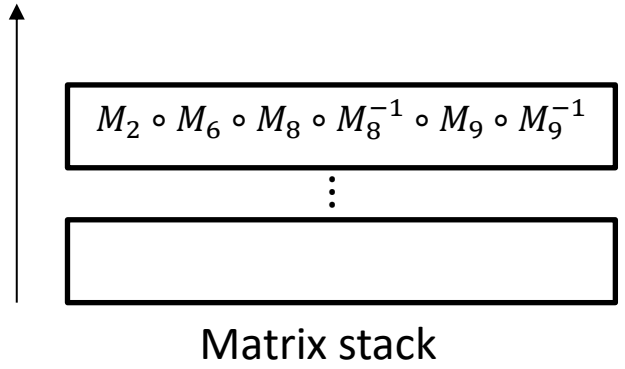


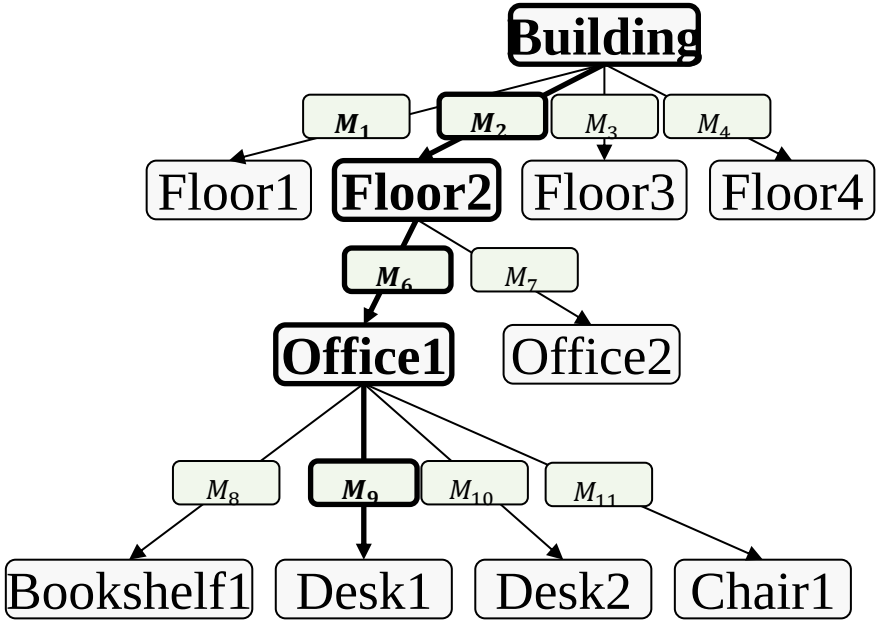
Assignment 3 (Matrix Stack)

Scene Graphs (Recall):

We could:

- Multiply on the right by M_8
- Draw *Bookshelf1*
- Multiply on the right by M_8^{-1}
- Multiply on the right by M_9
- Draw *Desk1*
- Multiply on the right by M_9^{-1}





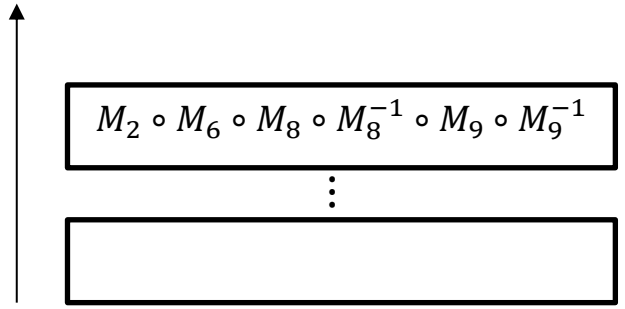
Assignment 3 (Matrix Stack)

<u>Scene Graphs (Recall)</u>:

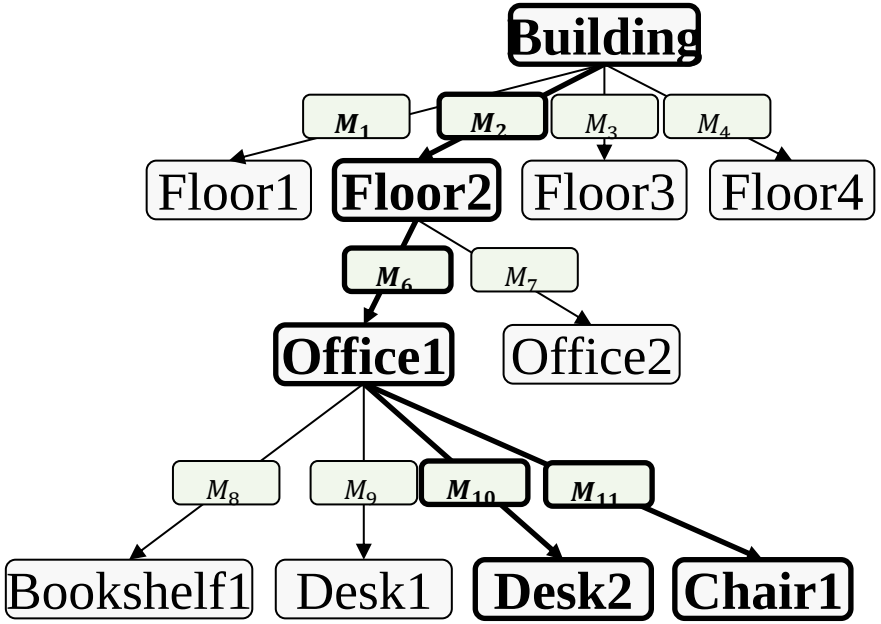
We could:

- Multiply on the right by M_8
- Draw *Bookshelf1*
- Multiply on the right by M_8^{-1}
- Multiply on the right by M_9
- Draw *Desk1*
- Multiply on the right by M_9^{-1}

The accumulation of matrix products, combined with numerical imprecision, could produce inaccurate results.



Matrix stack

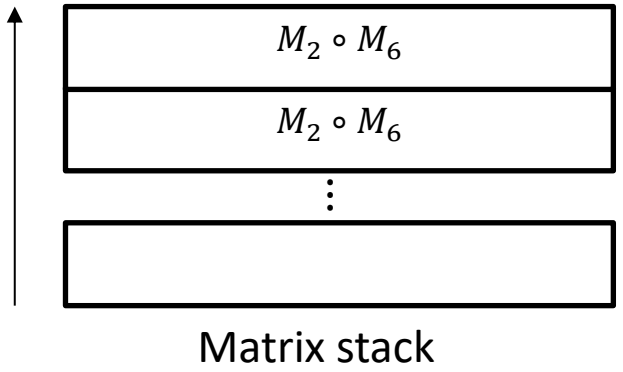


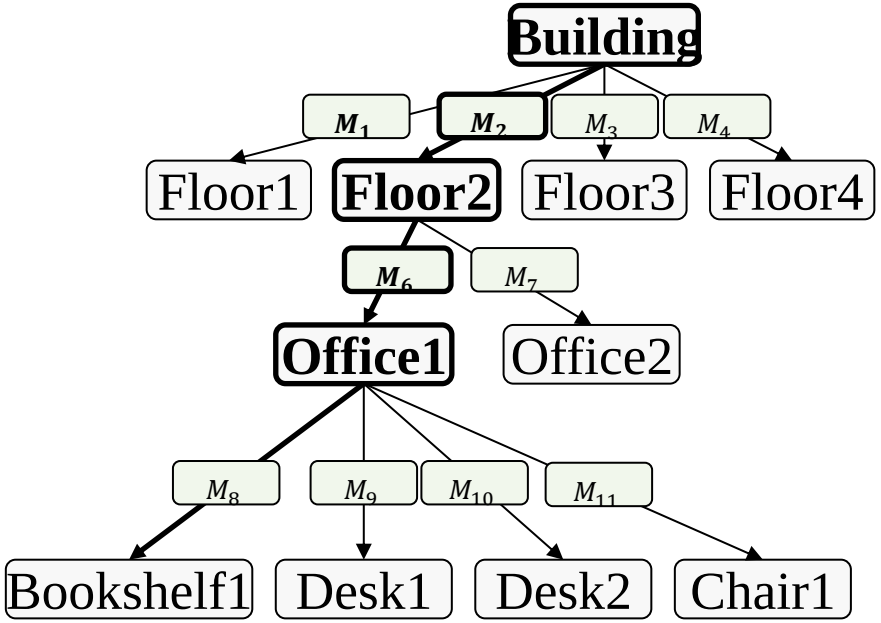
Assignment 3 (Matrix Stack)

<u>Scene Graphs (Recall)</u>:

We could also:

- Push ($M_2 \circ M_6$) onto the stack



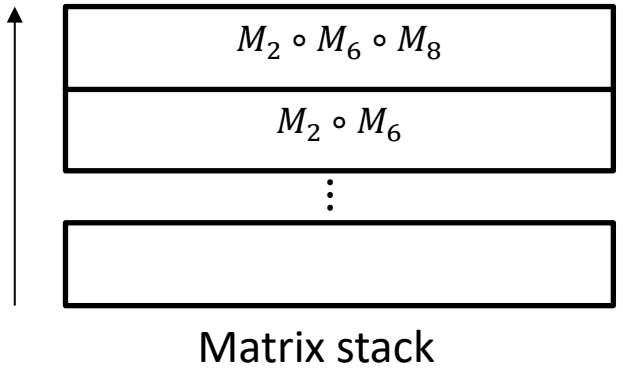


Assignment 3 (Matrix Stack)

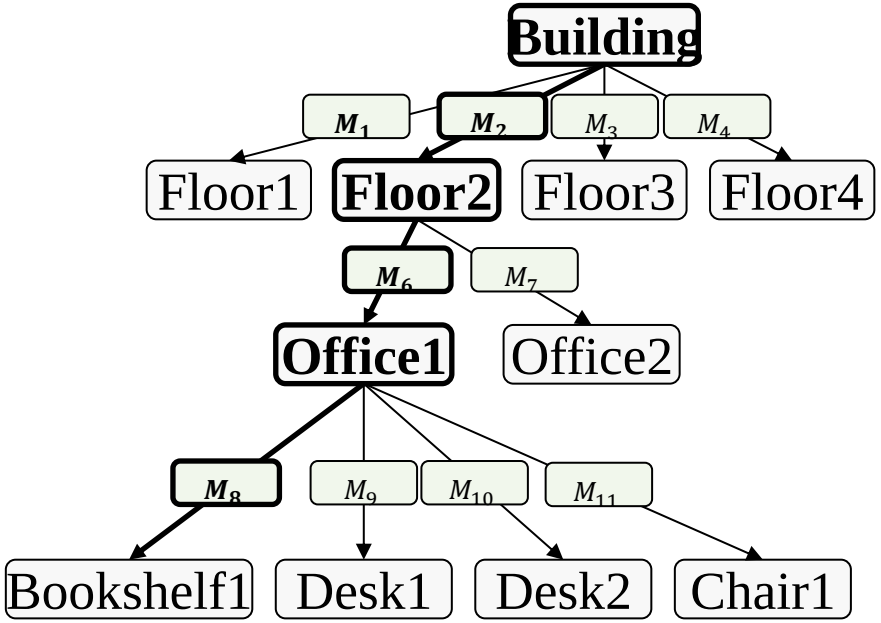
<u>Scene Graphs (Recall)</u>:

We could also:

- Push ($M_2 \circ M_6$) onto the stack
- Multiply on the right by M_8



Matrix stack

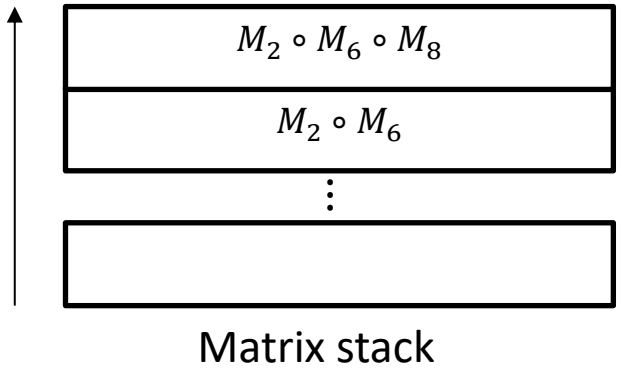


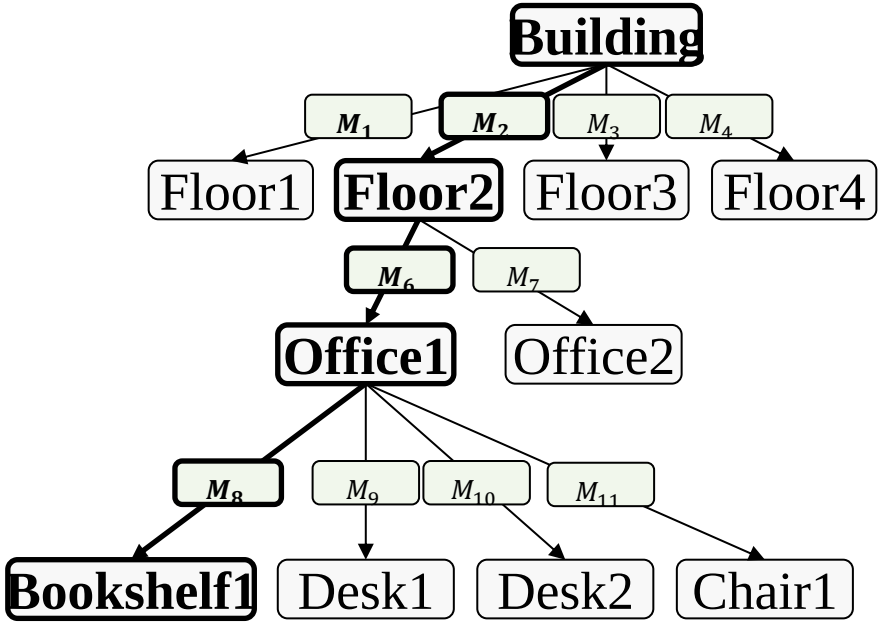
Assignment 3 (Matrix Stack)

<u>Scene Graphs (Recall)</u>:

We could also:

- Push ($M_2 \circ M_6$) onto the stack
- Multiply on the right by M_8
- Draw *Bookshelf1*



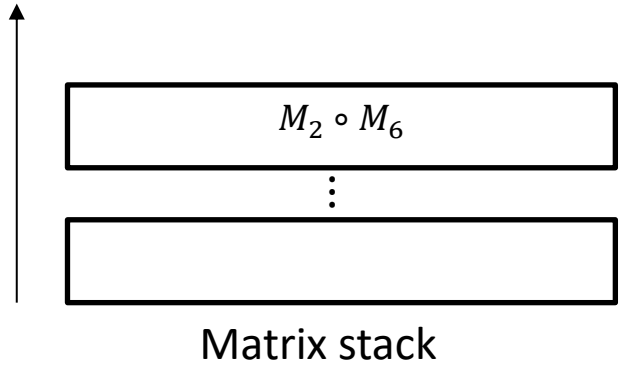


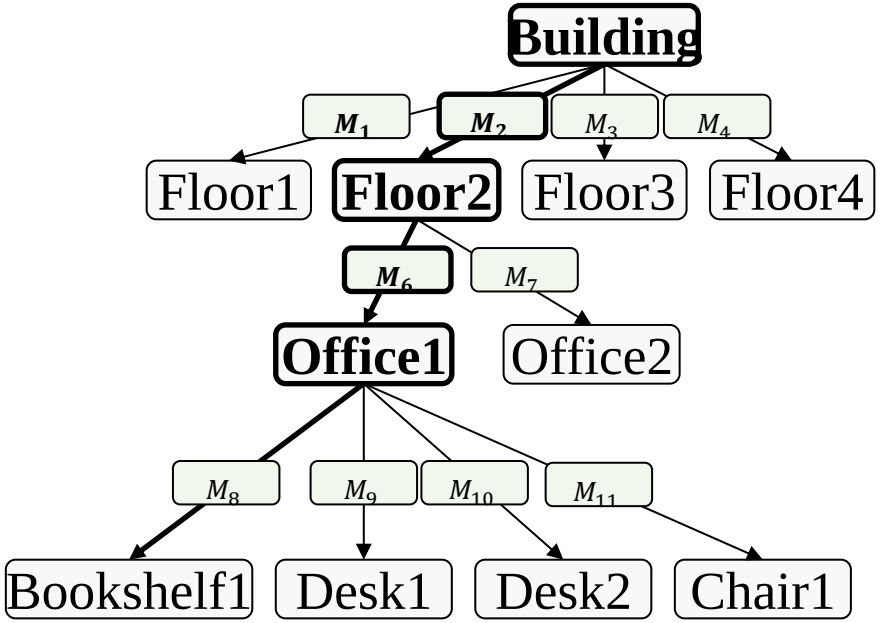
Assignment 3 (Matrix Stack)

Scene Graphs (Recall):

We could also:

- Push ($M_2 \circ M_6$) onto the stack
- Multiply on the right by M_8
- Draw *Bookshelf1*
- Pop off the top of the stack



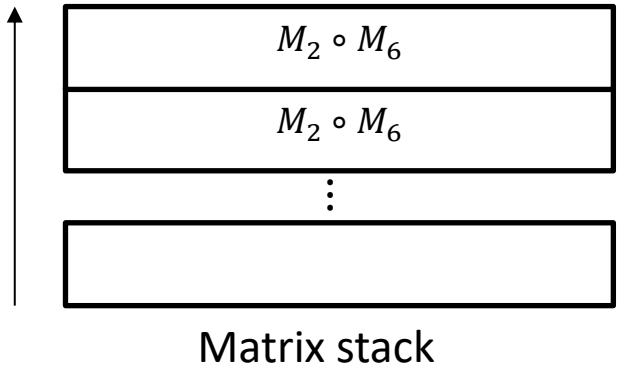


Assignment 3 (Matrix Stack)

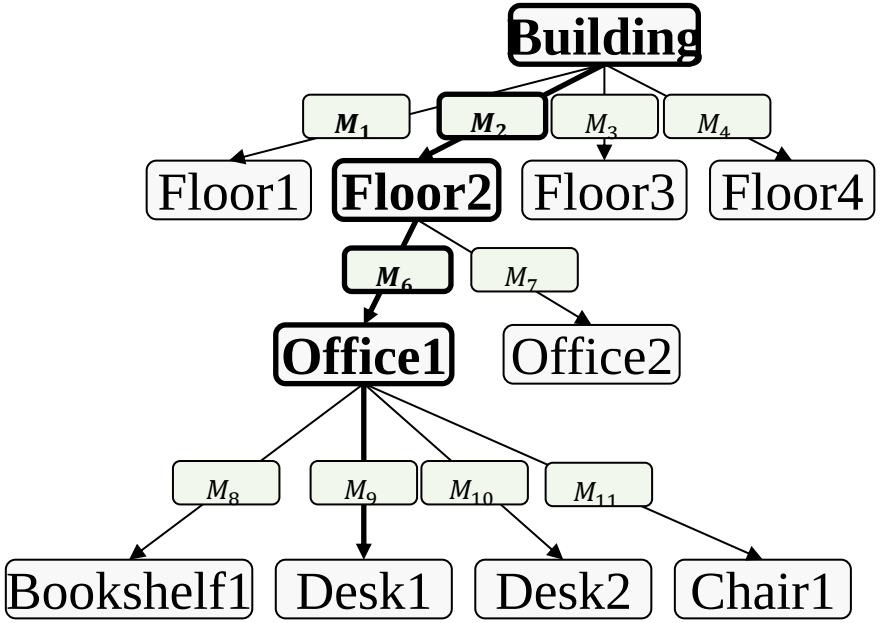
<u>Scene Graphs (Recall)</u>:

We could also:

- Push ($M_2 \circ M_6$) onto the stack
- Multiply on the right by M_8
- Draw *Bookshelf1*
- Pop off the top of the stack
- Push ($M_2 \circ M_6$) onto the stack



Matrix stack

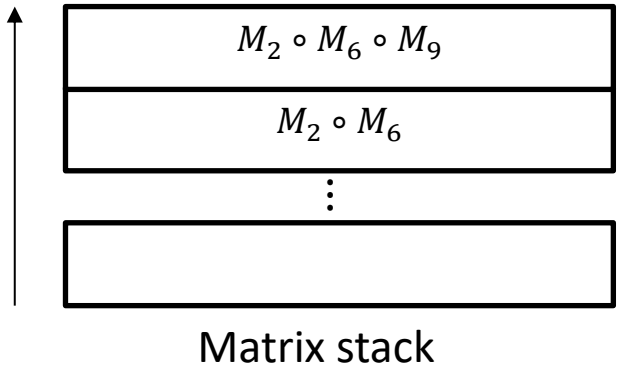


Assignment 3 (Matrix Stack)

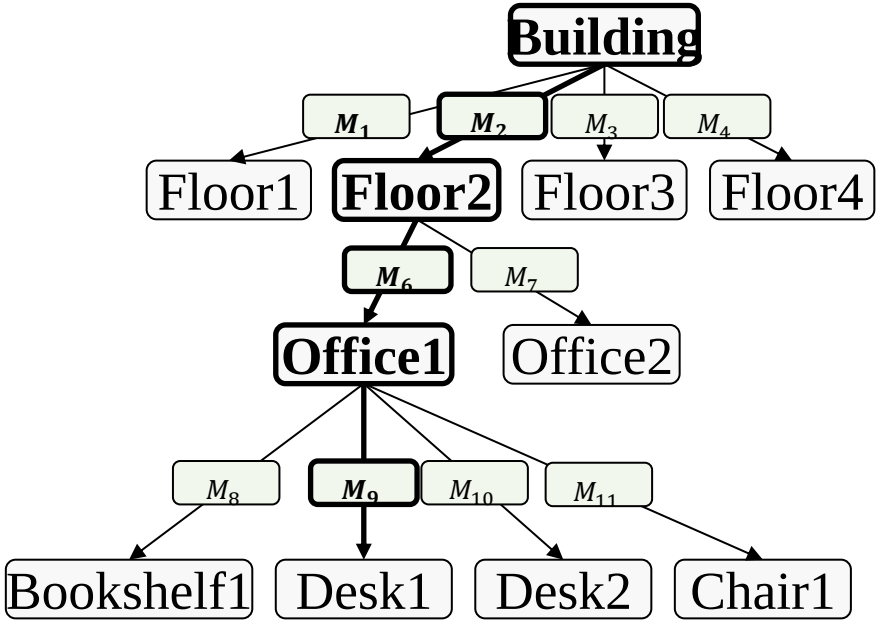
<u>Scene Graphs (Recall)</u>:

We could also:

- Push ($M_2 \circ M_6$) onto the stack
- Multiply on the right by M_8
- Draw *Bookshelf1*
- Pop off the top of the stack
- Push ($M_2 \circ M_6$) onto the stack
- Multiply on the right by M_9



Matrix stack

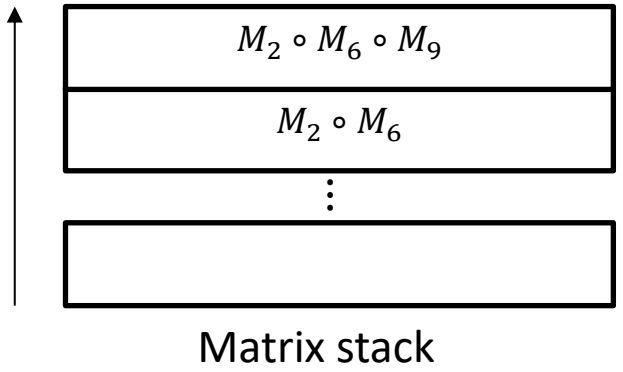


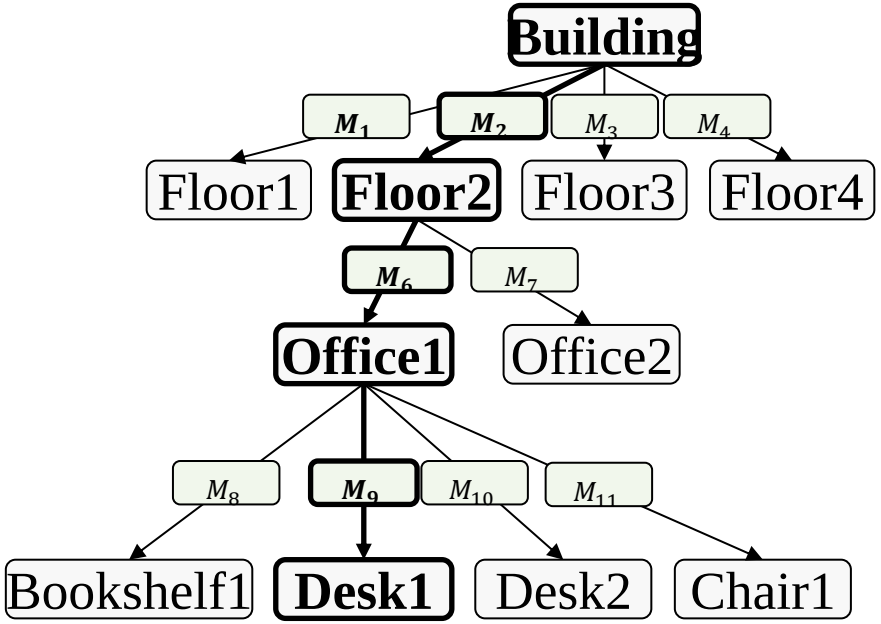
Assignment 3 (Matrix Stack)

<u>Scene Graphs (Recall)</u>:

We could also:

- Push ($M_2 \circ M_6$) onto the stack
- Multiply on the right by M_8
- Draw *Bookshelf1*
- Pop off the top of the stack
- Push ($M_2 \circ M_6$) onto the stack
- Multiply on the right by M_9
- Draw *Desk1*



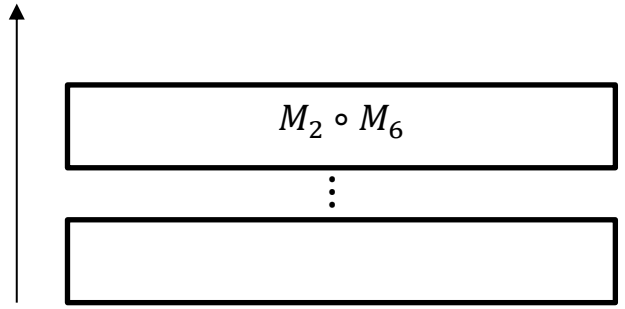


Assignment 3 (Matrix Stack)

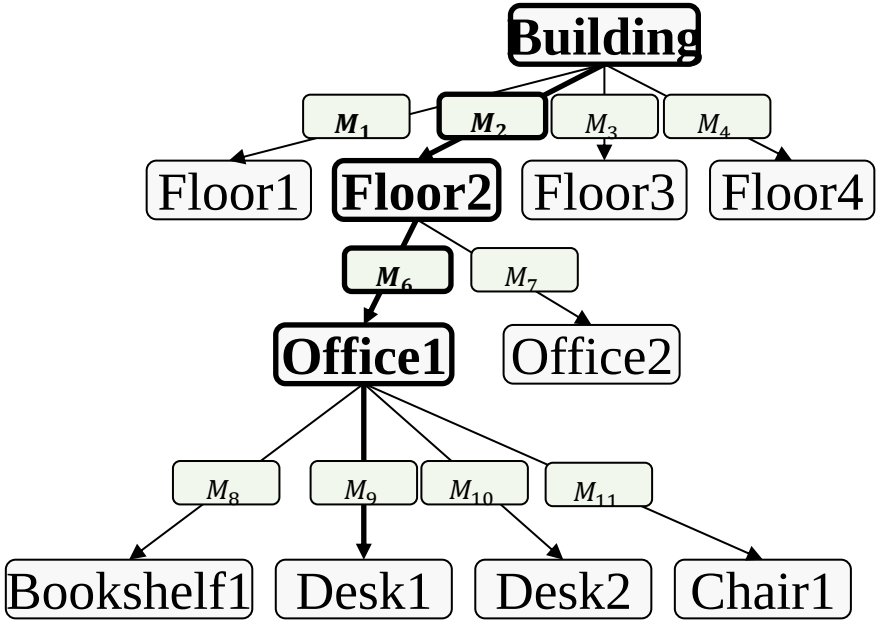
Scene Graphs (Recall):

We could also:

- Push ($M_2 \circ M_6$) onto the stack
- Multiply on the right by M_8
- Draw *Bookshelf1*
- Pop off the top of the stack
- Push ($M_2 \circ M_6$) onto the stack
- Multiply on the right by M_9
- Draw *Desk1*
- Pop off the top of the stack



Matrix stack

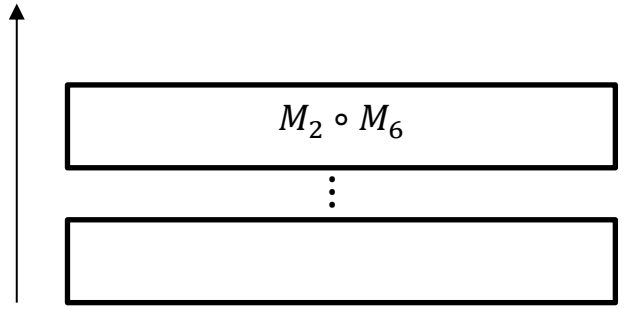


Assignment 3 (Matrix Stack)

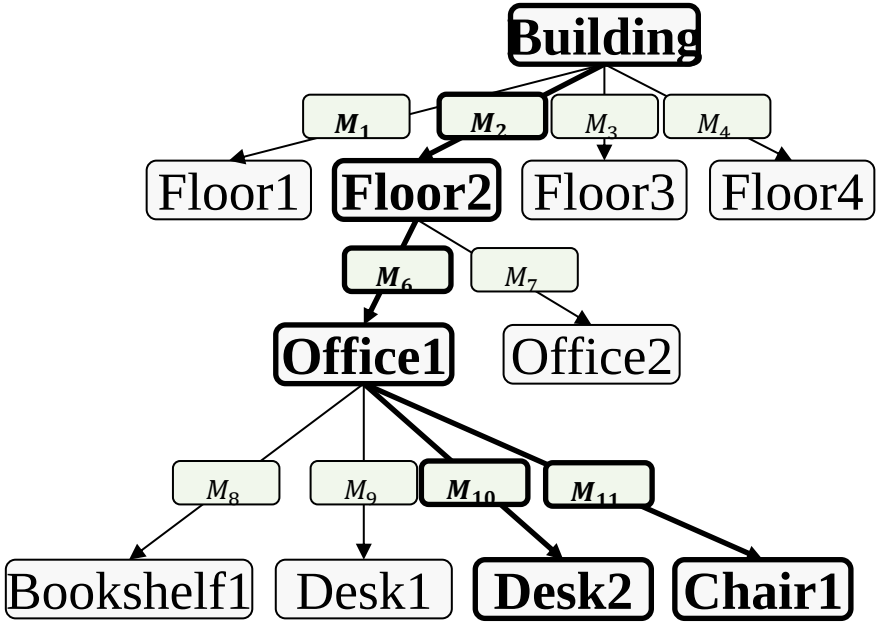
Scene Graphs (Recall):

We could also:

- Push ($M_2 \circ M_6$) onto the stack
- Multiply on the right by M_8
- Draw *Bookshelf1*
- Pop off the top of the stack
- Push ($M_2 \circ M_6$) onto the stack
- Multiply on the right by M_9
- Draw *Desk1*
- Pop off the top of the stack
- Etc.



Matrix stack



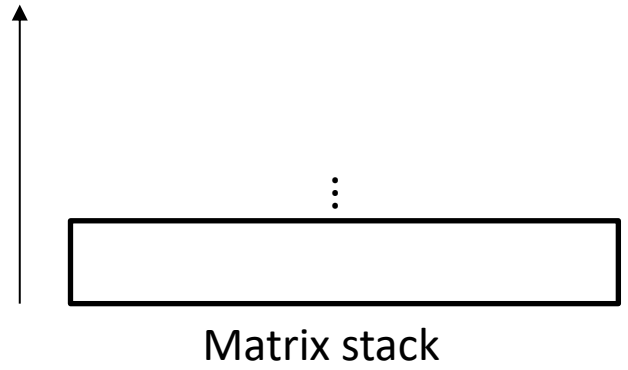
Assignment 3 (Matrix Stack)

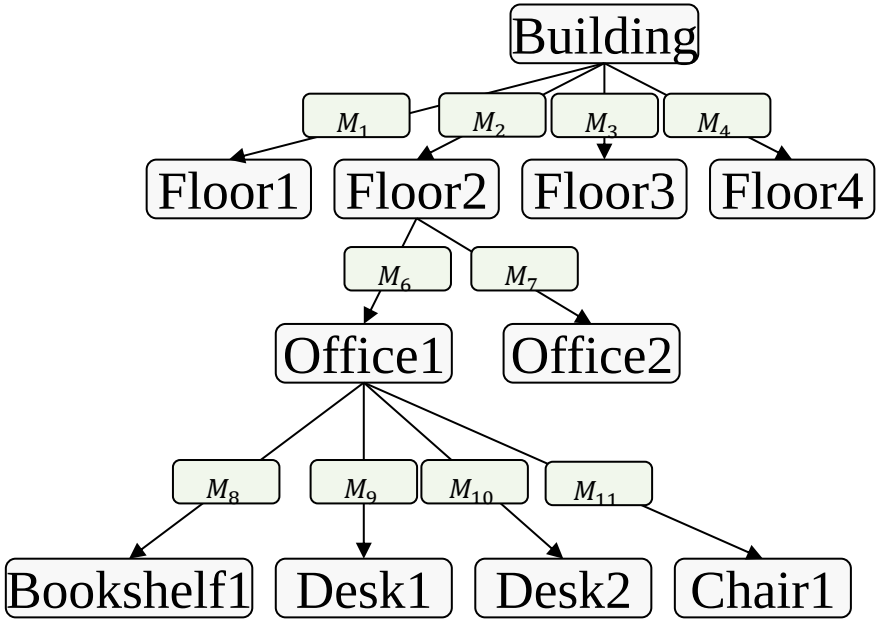
Scene Graphs (Recall):

We could also:

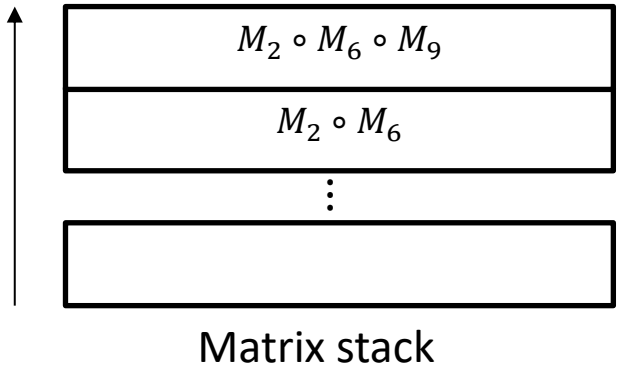
- Push ($M_2 \circ M_6$) onto the stack
- Multiply on the right by M_8
- Draw *Bookshelf1*
- Pop off the top of the stack
- Push ($M_2 \circ M_6$) onto the stack
- Multiply on the right by M_9
- Dr
- Po
- Et

Note:
The matrix stack has to be able to grow to (at least) the depth of the scene graph.





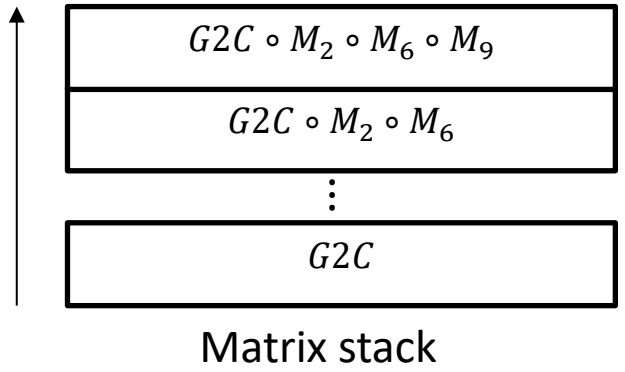
Assignment 3 (Matrix Stack)



Scene Graphs (Recall):

While all of this can be used to tell OpenGL how to transform the geometry from local coordinates to *global coordinates*, you actually want to tell OpenGL how to transform the geometry from local coordinates to *camera coordinates*.

Assignment 3 (Matrix Stack)



Scene Graphs (Recall):

To do this, you want to multiply the local-to-global transformation **on the left** by the global-to-camera transformation.

⇔ Make the global-to-camera transformation the first thing on the stack.

- Make sure you are **setting** the matrix stack with the global-to-camera transformation, **not multiplying** by it.
- Make sure you are working with the `GL_MODELVIEW` matrix stack.

When you're done rendering, the matrix stack should have the same depth as when you started.

Assignment 3 (Drawing Primitives)

glVertex*:

When you invoke this function, OpenGL sends the vertex into the rendering pipeline with:

- Position obtained by applying the current GL_MODELVIEW transform
- Color computed using the current lights and materials
- Texture coordinates as specified

OpenGL is a state machine:
OpenGL will always use the last specified normals, lights, material properties, etc. even if you did not specify them explicitly.

Assignment 3 (Drawing Primitives)

glVertex*:

When you invoke this function, OpenGL sends the vertex into the rendering pipeline with:

- Position obtained by applying the current GL_MODELVIEW transform
- Color computed using the current lights and materials
- Texture coordinates as specified

OpenGL is a state machine:
OpenGL will always use the last specified normals, lights, material properties, etc. even if you did not specif

Make sure to set the vertex's properties before specifying its position!

Assignment 3 (Textures)

```
glTexImage2D( ... , const void *data );
```

You do not have to copy the texture values into a separate array. You can directly use the memory address of the first image pixel:

```
&_image(0,0)
```

Recall:

The operator `Image32::operator()( int , int )` returns a reference to the pixel.

Assignment 3 (Textures)

If the material does not have a texture associated with it, you should disable texture mapping, so that OpenGL doesn't try using the texture from the previously specified material.

Recall that OpenGL is a state machine.