Scene Management and Matrix Stacks

CS 385 - Class 8
17 February 2022

Scene Management

Managing Objects in a Scene

- Scenes of any complexity require a lot of bookkeeping
  - tracking the position and orientation of objects
  - maintaining the relationships between objects
    - *dependent transformations*
- We'll see multiple approaches to managing this complexity
  - *matrix stacks*
  - *scene graphs*

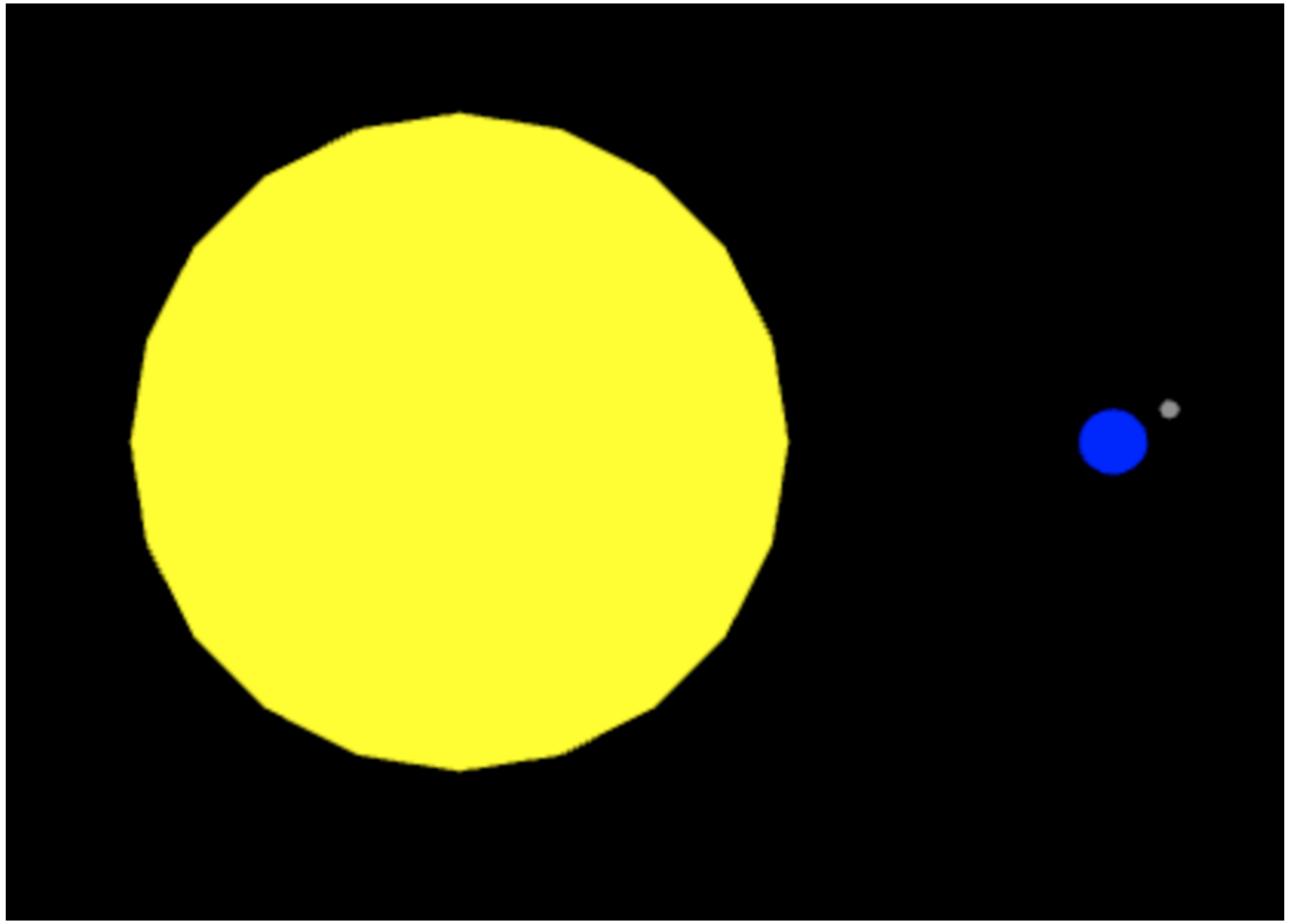
Managing Transformations

- Recall transformations are used to *modify the coordinate system*
  - we don't transform objects!
  - we transform the coordinate system, and then render relative to it
- All of those modifications are done using a 4x4 matrix transform
- Often you'll composite transformations for an effect
  - e.g., apply a translation, then a rotation, then a scale
- Sometimes you'll use one transformation for multiple objects
  - perhaps adding unique transformations to process the rendering for a single object

A Simple Example

An Example to Consider

- Let's simulate a simple Solar System
- We need to render three objects
  - Sun, Earth, and Moon
- The Sun is the center of the solar system, so everyone's dependent on its position
- The Moon's dependent on the position of the Earth
- Also, everyone's dependent on the viewer
  - which we specifying using the *viewing transformation*
- We'll use *time* as our motion variable
  - hours, days, and years control the positions of the various bodies



Implementation Considerations

- We need to keep track of lots of transforms
  - some of whom depend on other previous transforms
- We'll generate transforms based on various physical properties, as well as time

```
var sun = undefined;
var earth = undefined;
var moon = undefined;
var t = 0.0;

function init() {
  ... // initialize WebGL as usual


  sun = new Sphere(...);
  earth = new Sphere(...);
  moon = new Sphere(...);

  // Add add'l properties for our planets
  sun.radius = 696340;
  earth.radius = 6371;
  earth.distance = 147820000;
  moon.radius = 1737.4;
  ...

  reqeustAnimationFrame(render);
}
```

Tracking Transforms

- We need to keep track of lots of transforms
  - some of whom depend on other previous transforms

Variable	Description
V	Viewing transform
S	misc. scales
earthPos	Earth's position (relative to Sun)
earthDay	Earth's rotation around Sun
earthHour	Earth's rotation around its axis
moonPos	Moon's position (relative to Earth)

```
function render() {
  gl.clear(...);

  // various other initialization
  var near = ...;
  var far = ...;

  t += dt; // in hours

  // Simplest viewing transform - affects everyone
  var V = translate(0, 0, -0.5*(near + far));
  var P = perspective(...);

  // Render the sun first, so compute its
  //   transforms first
  var S = scale(sun.radius);

  sun.P = P;
  sun.MV = mult(V, S);
  sun.render();
```

Tracking Transforms (cont'd)

- Set up the transforms and render planet Earth
  1. Define constants related to converting time to useful values
  2. Determine the appropriate day and hour (which control Earth's position relative to the Sun)
  3. Generate a number of transforms to help with rendering
  4. Specify the transforms for Earth's shader `uniforms`

```
❶ const HoursPerDay = 24;
  const HoursPerYear = 365.25 /* days */ * HoursPerDay;

❷ // Render the Earth next
  var day = t / HoursPerYear * 360; // in degrees
  var hour = t % HoursPerDay;

❸ var earthPos = translate(0.0, 0.0, earth.distance);
  var earthDay = rotate(day, [0, 1, 0]);
  var earthHour = rotate(hour, [0, 1, 0]);
  S = scale(earth.radius);

❹ earth.P = P;
  // Earth's transforms: V * R * T * R * S
  earth.MV = mult(
               mult(
                 mult(
                   // rotate to day of year
                   mult(V, earthDay),
                 earthPos), // move to orbit
               earthHour), // rotate planet
             S); // scale to size
  earth.render();
```

Tracking Transforms (cont'd)

- Set up transforms and render the Moon

```
var moonPos = translate(0.0, 0.0, moon.distance);
S = scale(moon.radius);

moon.P = P;
// Moon's transforms: V * R * T * R * T * S
moon.MV = mult(
            mult(
              mult(
                mult(
                  // rotate to day of year
                  mult(V, earthDay),
                earthPos), // move to orbit
              earthHour), // rotate planet
            moonPos), // move to Moon's orbit
          S); // scale to size
moon.render();
```

Tracking Transforms (cont'd)

- The highlighted code is identical for the Moon and Earth
  - Couldn't we reuse that?
- Yes, however ...
  - this situation doesn't really scale to much larger applications

```
var moonPos = translate(0.0, 0.0, moon.distance);
S = scale(moon.radius);

moon.P = P;
// Moon's transforms: V * R * T * R * T * S
moon.MV = mult(
            mult(
              mult(
                mult(
                  // rotate to day of year
                  mult(V, earthDay),
                earthPos), // move to orbit
              earthHour), // rotate planet
            moonPos), // move to Moon's orbit
          S); // scale to size
moon.render();
```

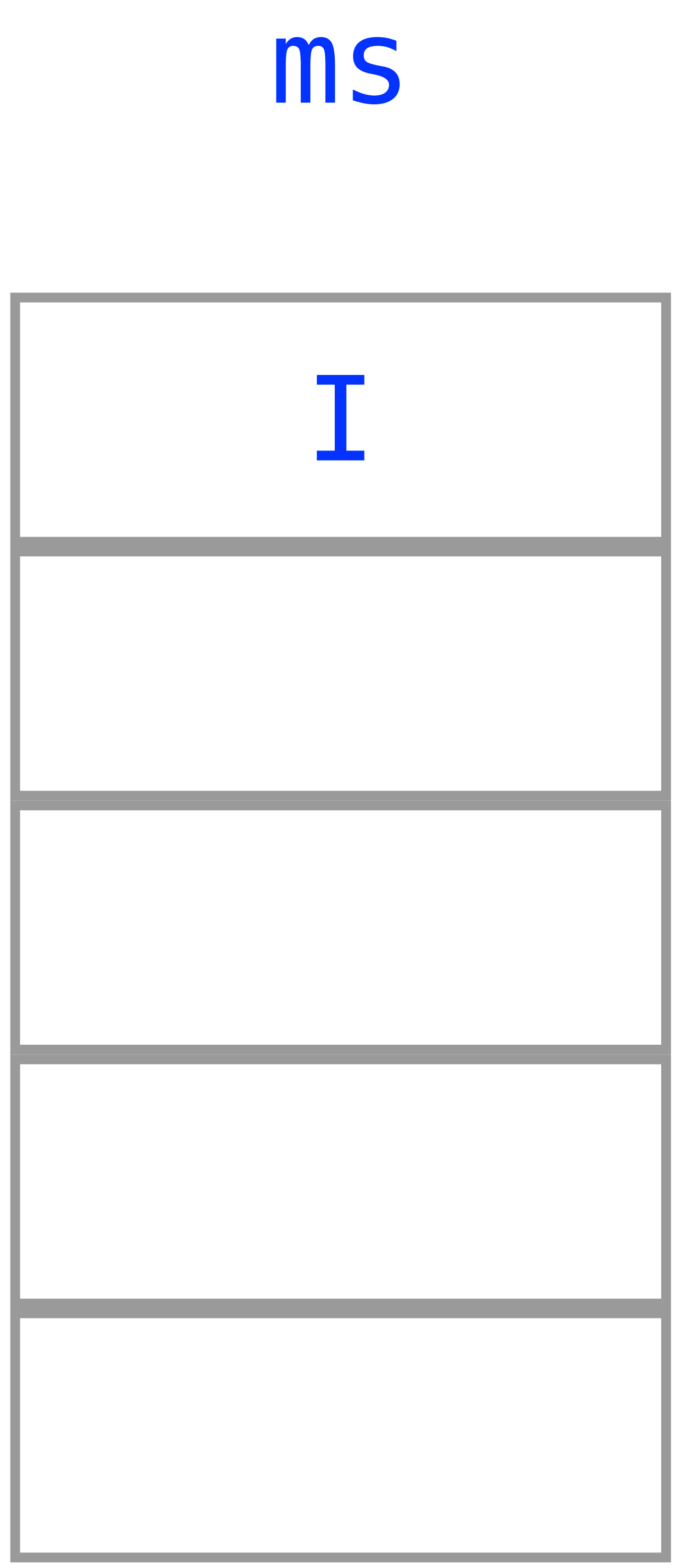
MatrixStack Functions

Matrix Stack

- A convenient way to manage transformations is to use a *matrix stack*
- It's a stack data structure that:
  - stores 4x4 matrices
  - supports standard stack operations: push, pop, etc.
- We'll encapsulate JavaScript's native list structure in an object
  - add some methods to make the job simpler, and the code more descriptive

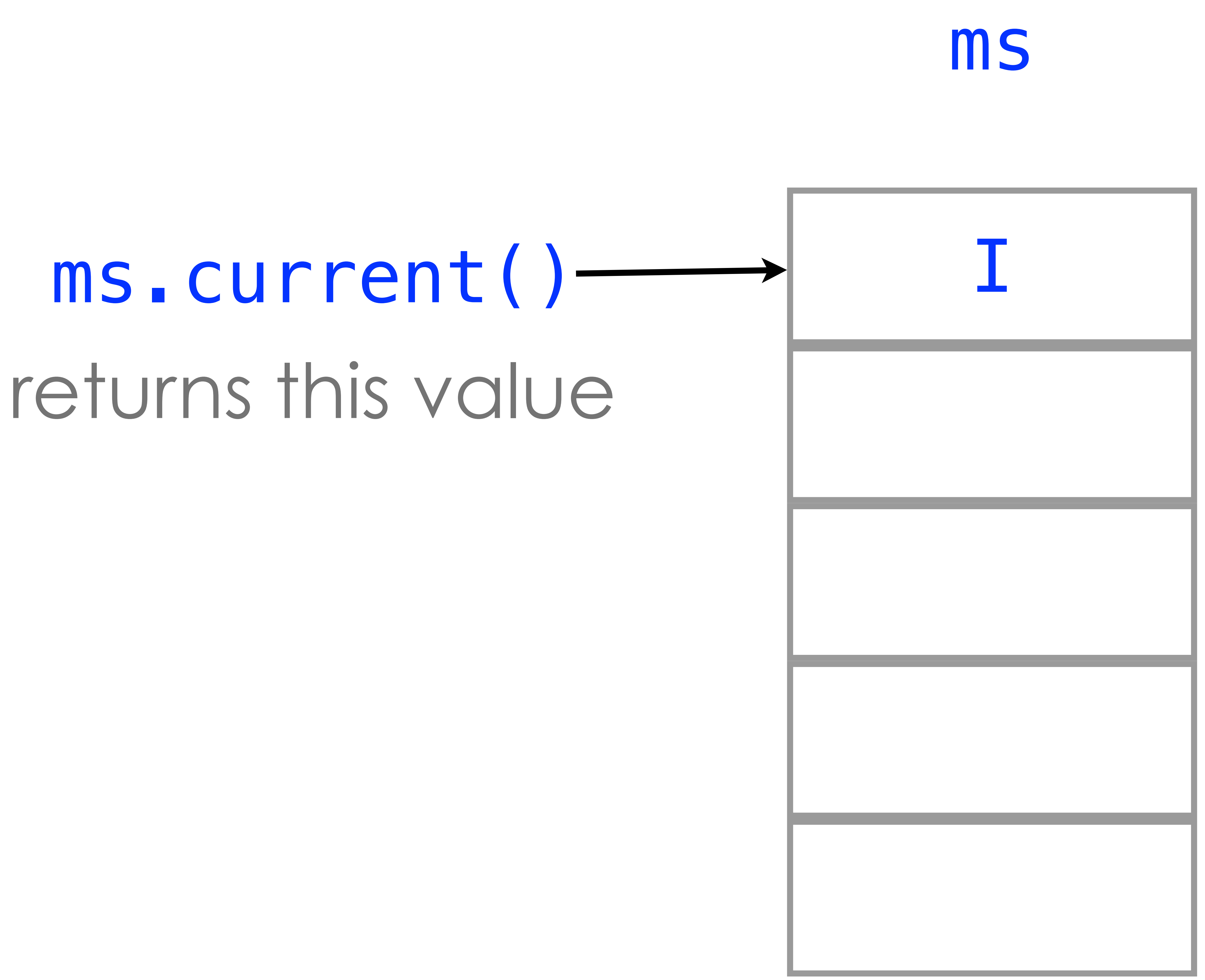
`new MatrixStack()`

- When a new MatrixStack is created, an object with a JavaScript list is returned
- The first element in the list is initialized to an identity matrix



ms.current()

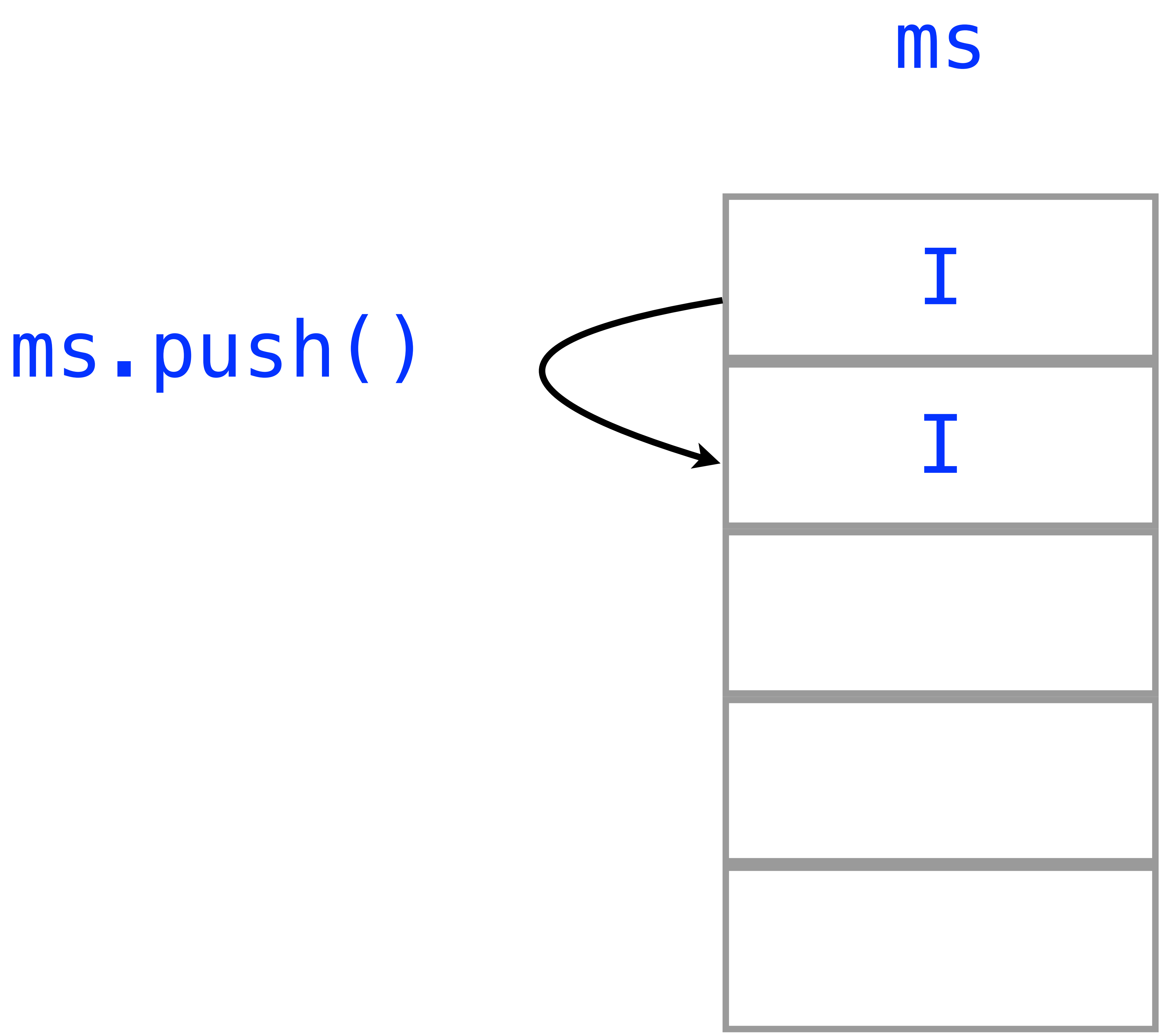
- Returns the *current matrix*, which is the element at the top of the stack
- You'll use this when you want to apply the current transformation in the scene
- Most often, this value will be passed into a *shader uniform variable*




```
gl.uniformMatrix4fv(uniformVariableLocation, false, flatten(ms.current()));
```

ms.push()

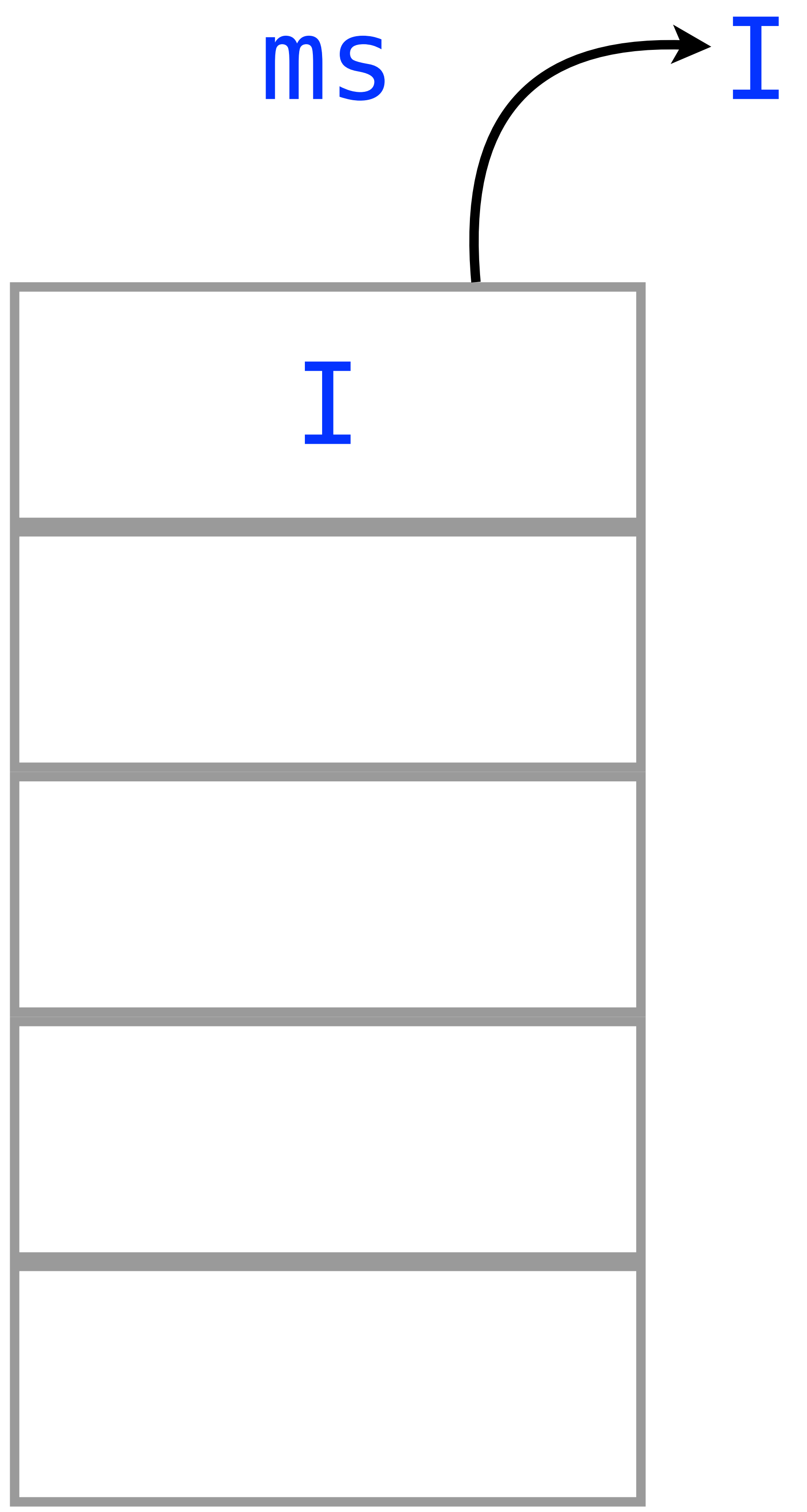
- Duplicates the top element (i.e., the current matrix), and pushes it onto the stack



ms.pop()

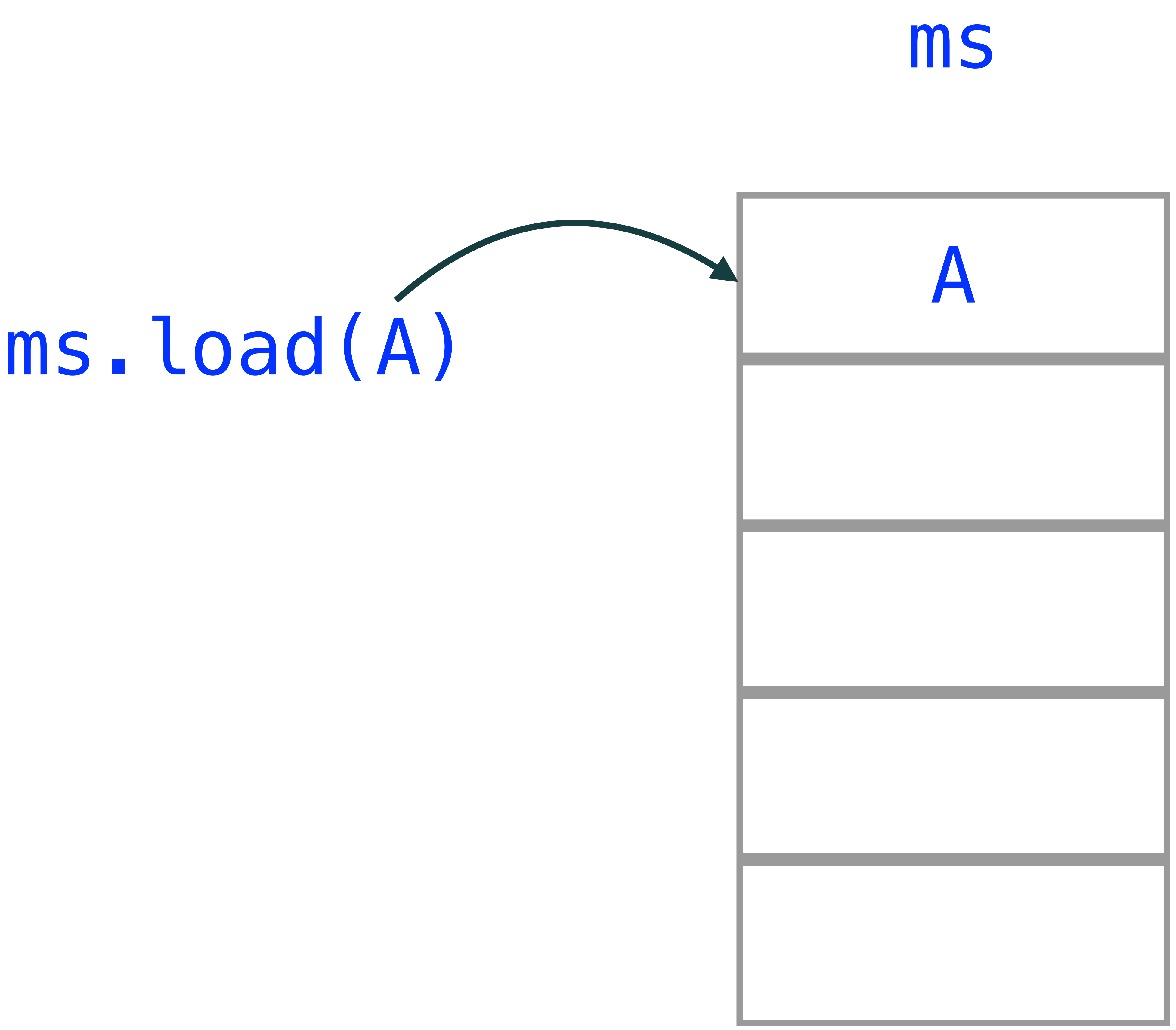
- Discards the top element of the stack
- All other elements move up one position

ms.pop()



ms.load(M)

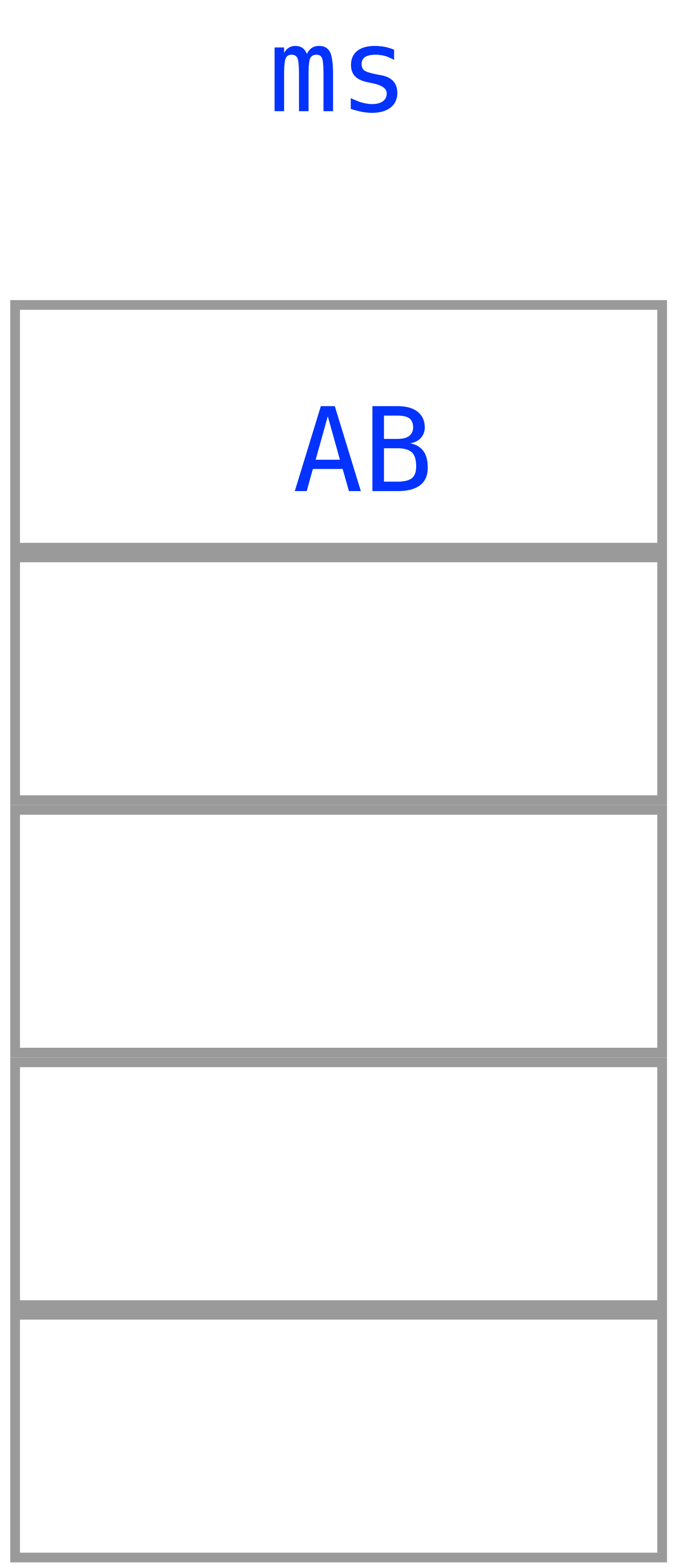
- Replaces the top element of the stack with passed matrix value



`ms.mult(M)`

- Multiplies the top element of the matrix stack *on the right side* with `M`

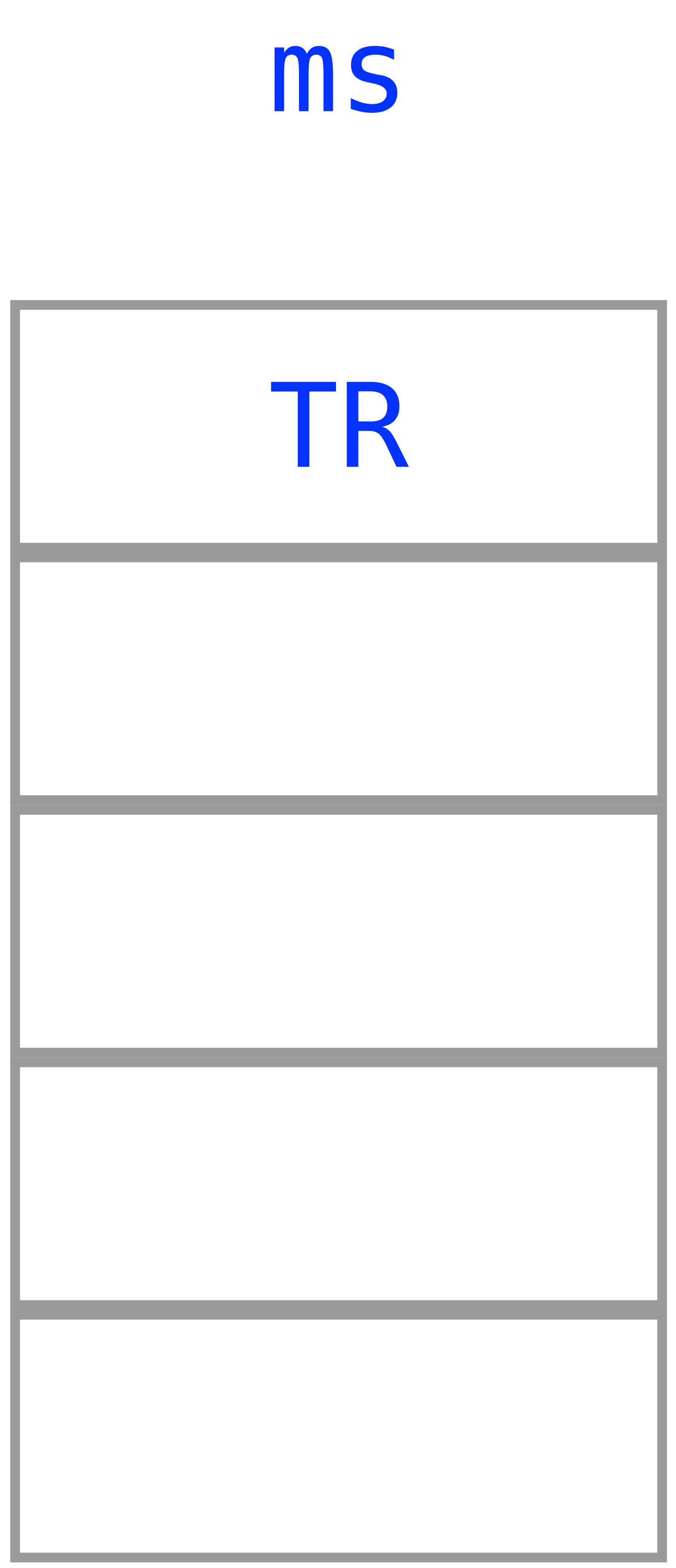
`ms.mult(B)`



ms.rotate(angle, axis)

- Generates a rotation matrix, R, using the provided angle and axis of rotation
- Multiples the current matrix by R on the right

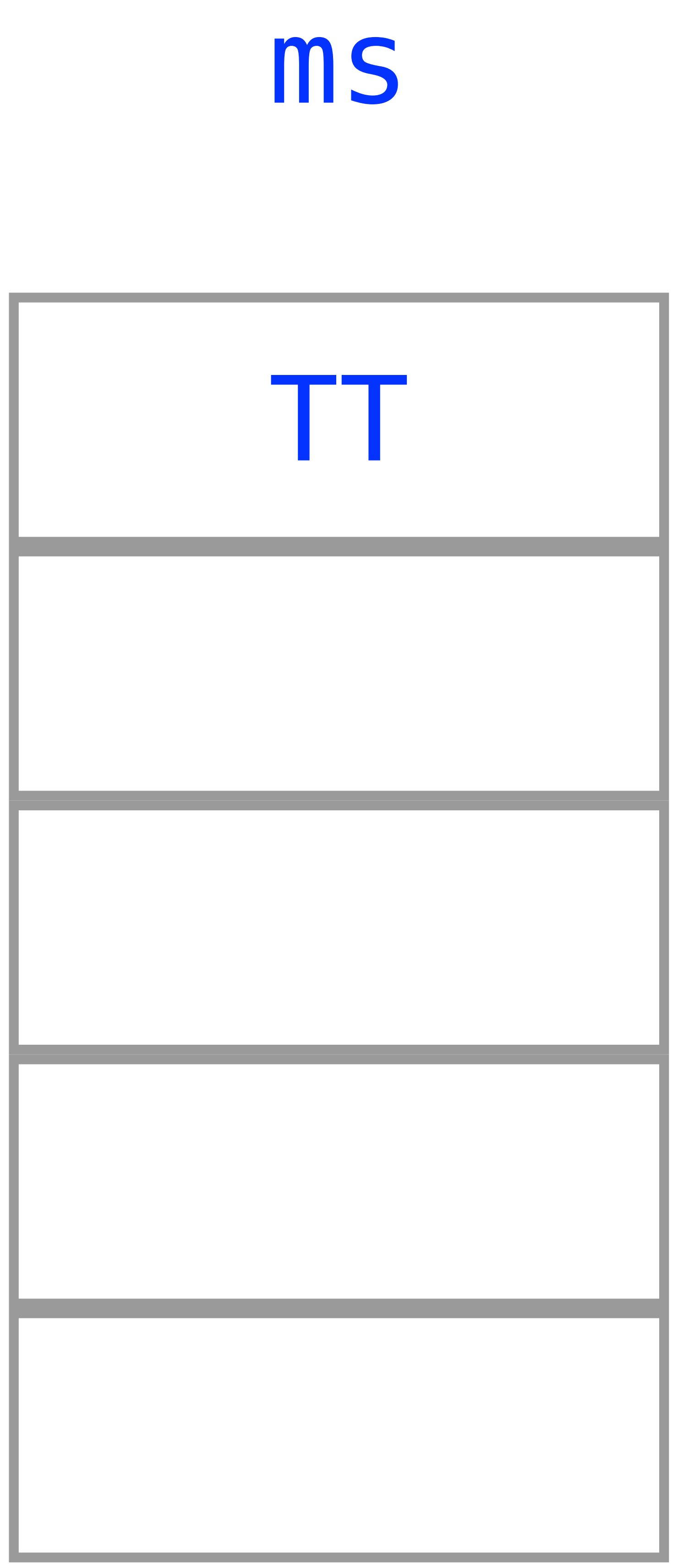
```
ms.rotate(theta,[0, 1, 0])
```



`ms.translate(x, y, z)`

- Generates a translation matrix, T, using the provided distances
- Multiples the current matrix by T on the right

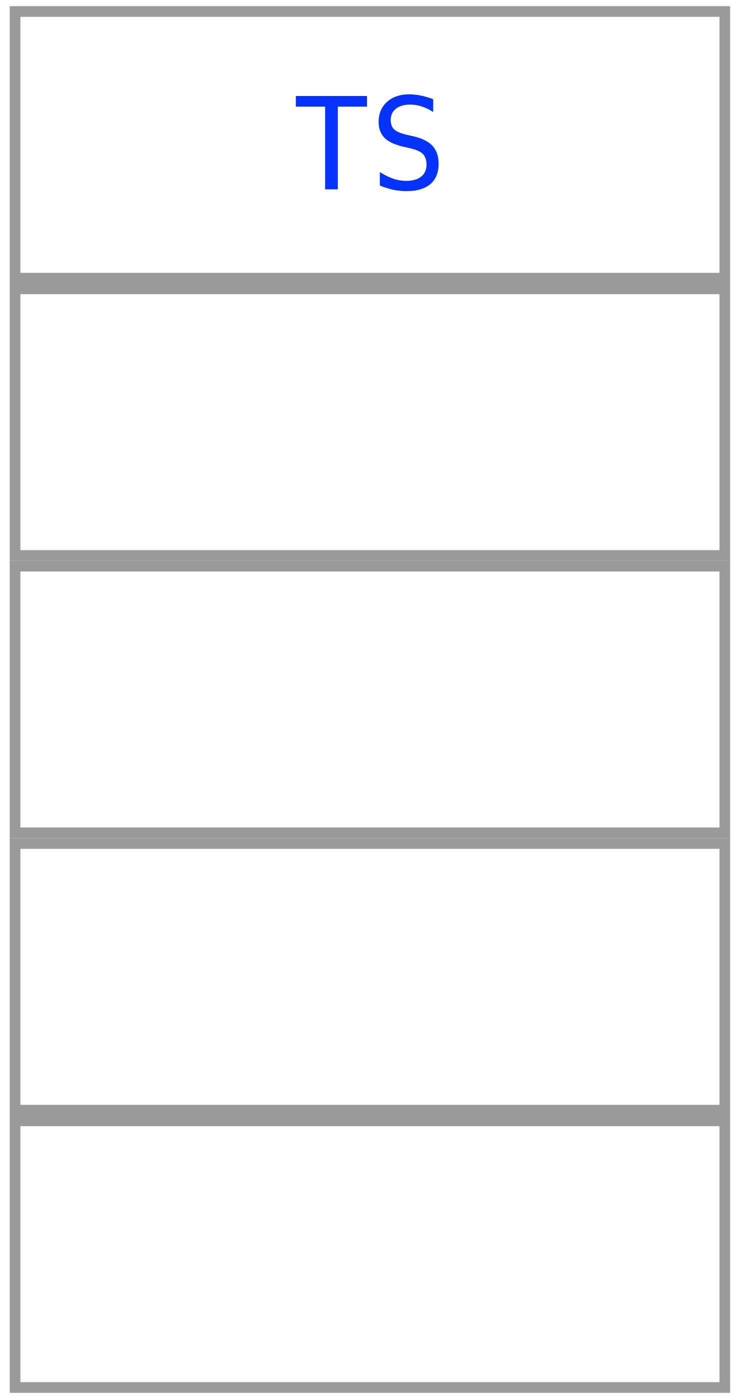
`ms.translate(x, y, z)`



`ms.scale(x, y, z)`

- Generates a translation matrix, S, using the provided scale factors
- Multiples the current matrix by S on the right

ms

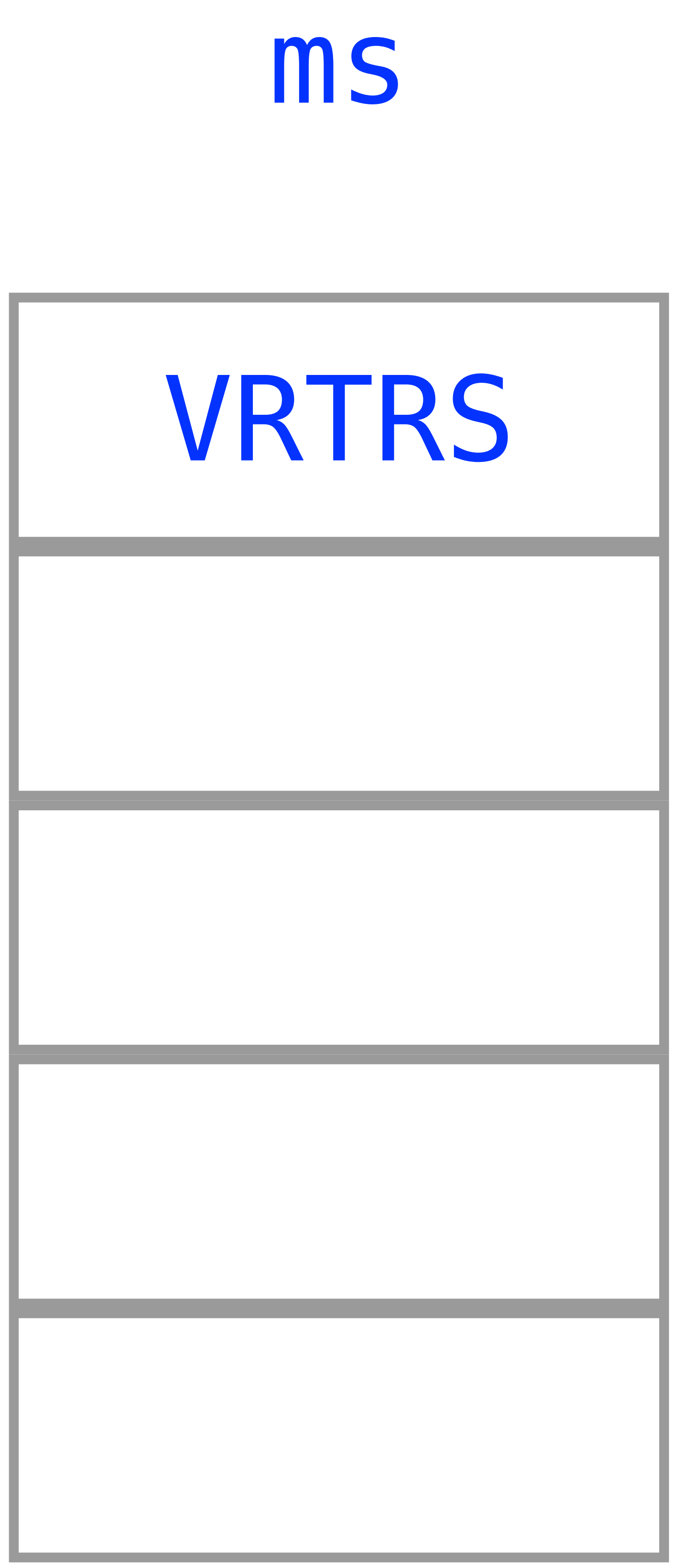



```
ms.scale(x, y, z)
```

Multiple Transformations

- Often, you'll have multiple transformations that affect the current matrix
- Recall that matrices multiply on the *right* of the current matrix
- For example, the following sequence would yield the illustrate current matrix

```
ms.load(V); // Viewing transform
ms.rotate(theta, [0, 1, 0];
ms.translate(x, y, z);
ms.rotate(phi, [0, 1, 0]);
ms.scale(x, y, z)
```



The MatrixStack

MatrixStack.js

- Our JavaScript matrix stack implementation
  - stored in **MatrixStack.js**
  - put it in the Common directory
- In your HTML file, include a reference to the JavaScript file
  - make sure to use the path correct
- And since **MatrixStack.js** relies on **MV.js**, don't forget to include it as well

```
<!DOCTYPE html>
<html>
<head>
  <script src="MV.js"></script>
  <script src="MatrixStack.js"></script>
  ...
</head>
<body>
  ...
</body>
</html>
```

Creating a MatrixStack

- In your JavaScript application code
- Instance a new matrix stack
- To instance a JavaScript object, you use the `new` operator

  ```
  ms = new MatrixStack();
  ```

- In this example, we create the matrix stack inside of the `render()` function
  - since that's the only place we'll use it
  - you may find times to allocate it other places

```
var canvas;   // our HTML5 canvas
var gl;       // our WebGL context

function init() {
  ...
}

function render() {
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

  var ms = new MatrixStack();

  ...
}
```

Matrix Stack Implementation

- MatrixStack is implemented as a JavaScript object
  - encapsulates the important data
- JavaScript instances are just classes created with a *constructor*
- Constructors in JavaScript are just functions that return an object
- Members of the object are initialized in the constructor using the `this.property` construct
  - `property` is just a member of the object
  - For example, `this.stack` creates an array initialized with a 4x4 identity matrix

```
//
// MatrixStack.js
//

function MatrixStack() {
  this.stack = [ mat4() ];

  this.current = function () { return this.stack[0]; }
  this.pop = function () { this.stack.shift(); }
  ...
}
```