M30242-Graphics and Computer Vision

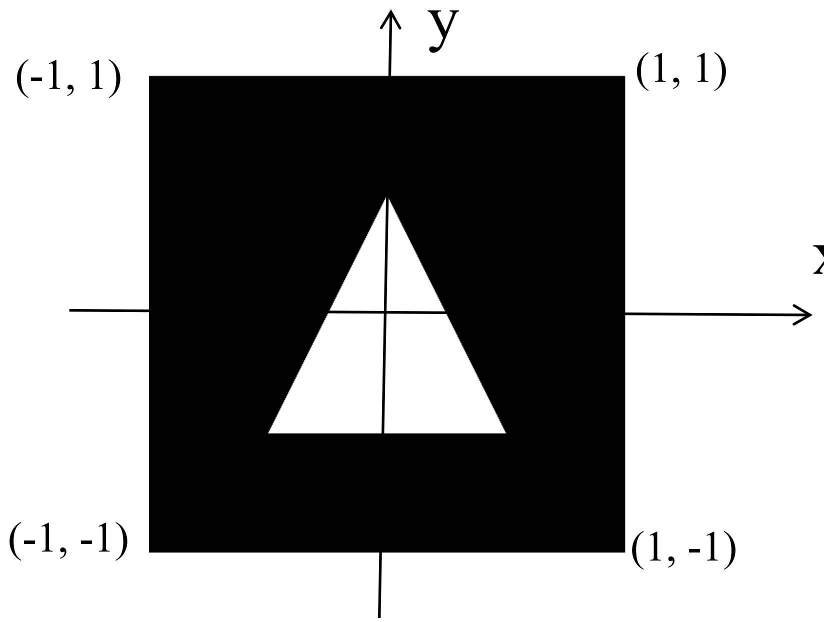
Lecture 05: Transformations in WebGL

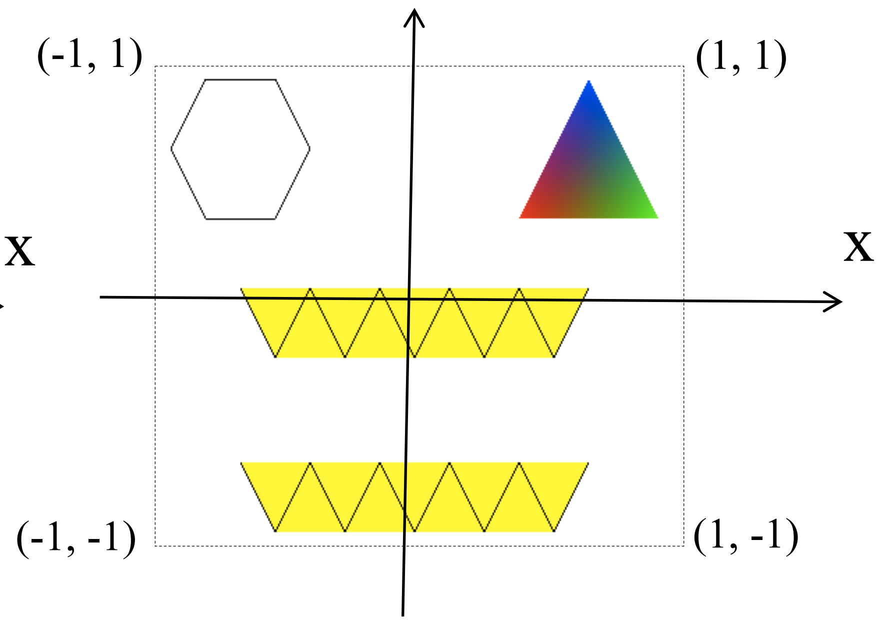
Introduction

- Transformations are the cornerstones of the graphics pipeline.
- They control the position of objects and the way the graphics is drawn.
  - specifying locations of various graphics objects, e.g., geometric objects, lights and the camera.
  - implementing and simulating the properties of physical elements of visual systems, e.g., camera lenses.
- The WebGL pipeline has different transformations and each of them has different use.

Our Approach So Far

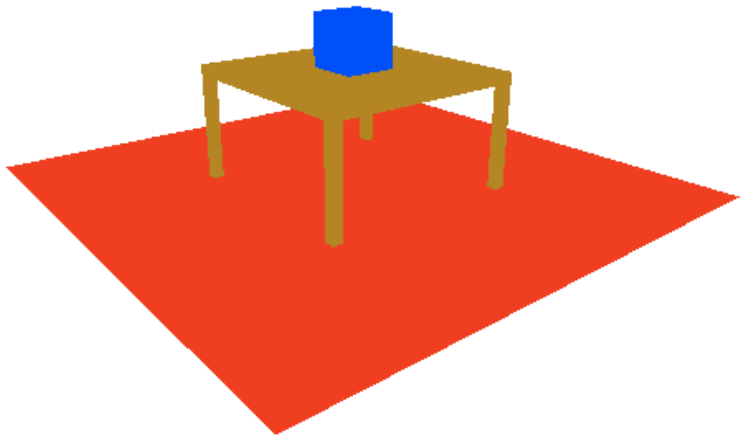
- So far, when we draw a shape, we specify the vertex coordinates in the normalised coordinate system, i.e., in the 3D screen space.





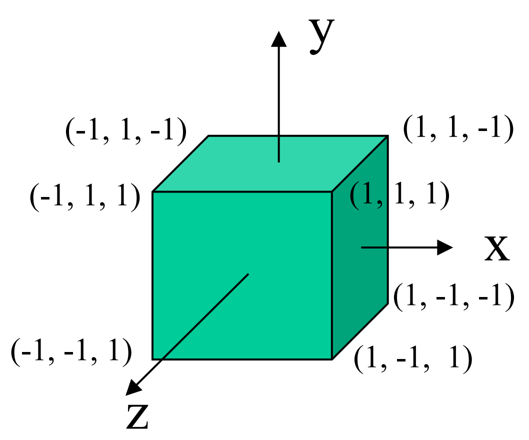
Difficulties in 3D

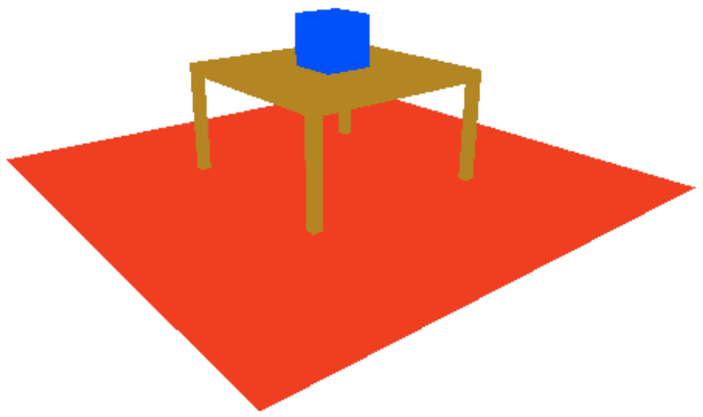
- It is extremely hard, if not impossible to define 3D shapes directly in normalised (or sometimes called clipping window) coordinate system, because *you need to take the perspective effect (i.e., lens and viewpoint) into account*.
- It is hard to reuse objects: you will need to define every shape in full (i.e., all their vertices) to the get the following scene.



Use Transformations

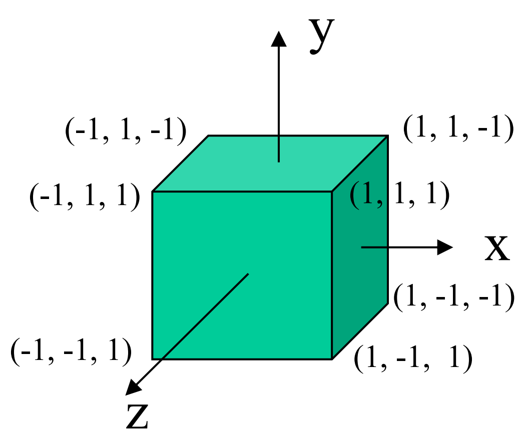
- The use of transformations makes it possible to create complex object by reusing (simpler) objects.
- For example, to draw the scene as below, one needs to define only two objects.
  - A cube and a plane of certain size (e.g., 1-by-1 in size).
  - Both have a default location at the origin of the world coordinate system.

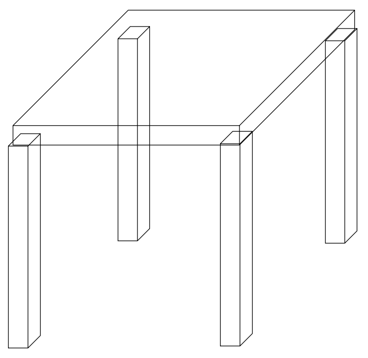




Cont'd

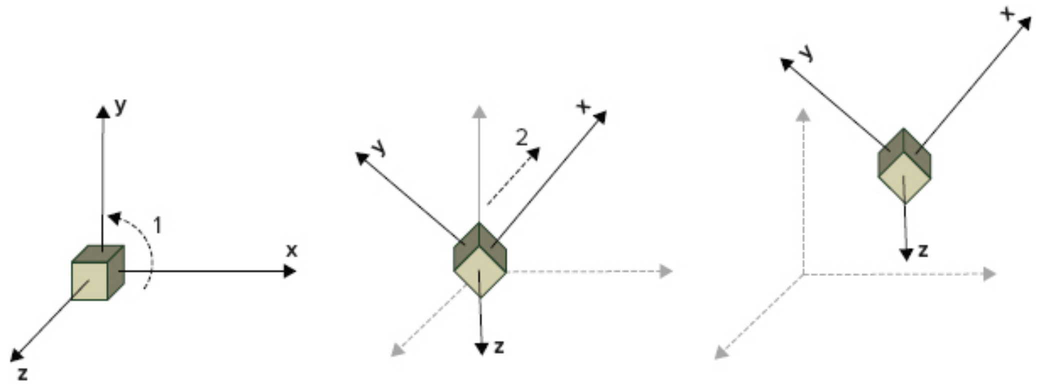
- To draw the table, we need to:
  - Stretch (scale) the cube in x and z dimensions and squeeze it in y dimension to obtain the shape of the tabletop. Place (translate) it at the position of the tabletop and draw.
  - Transform the cube so that it takes the shape of the legs. Position it at the appropriate position, e.g., the front left leg of the table, and draw. Draw the remaining legs in the same way.





Transform Objects

- Once defined, an object can be transformed by the following transformations:
  - translation: change its position
  - rotation: change its orientation
  - Scale (uniform or non-uniform): change its dimension - makes it bigger or smaller, squeeze in one dimension and stretch in others, etc
- These transformations are realised via matrix operations, i.e., translating and rotating the vertices (points) of the object in 3D space.



Transformation & Matrices

- Transforming an object is to transform its vertices one-by-one, which is done by multiplying the transformation matrix (4x4) by the vector (homogeneous coordinates) of a vertex – matrix operations
- For details of transformations and matrices, see the supplement to the lecture.

Transformation & Matrices

- JavaScript does not have any built-in support for matrix and vector operations.
- You can either write your own functions for matrix and vector operations or use one of the available open-source libraries.
- The following are three libraries that are commonly used in WebGL applications:
  - glMatrix
  - Sylvester
  - WebGL-mjs
- These libraries are similar. We use glMatrix (glMatrix.js) (see supplement for details and exercises).

Transformations in WebGL

- In WebGL, we will have following transformations:
  - Model transformations: one for each object. They specify the location of an object
  - A View transformation: it controls the viewpoint.
  - Modelview transformation: the combination of model and view transformations.
  - Perspective transformation: it defines camera properties.
  - Viewport transformation: it controls rendering properties , e.g., aspect ratio.
- The first four transformations are applied to the vertices of objects in the vertex shader.
- The last one is not handled by the vertex shader but by the primitive assembly stage of the WebGL pipeline. We don’t need to worry about it after specifying the viewport.

Model Transformation

- The coordinates stored in the WebGLBuffer should be in the object's *local coordinates*, and,
- Objects are initially positioned at the *origin of the world coordinate system* by default)
- The model transformation is used to *place* and *orient* the model in the world coordinate system. Therefore, *each model in a 3D scene will have its own model transformation*.
- The model transformation is generally a **combination** of translation, rotation and scale operations.

Cont’d

- In the JavaScript library `glMatrix`, these operations/transformations correspond to the following functions:
  - `mat4.translate()`
  - `mat4.rotate()`
  - `mat4.scale()`

View Transformation

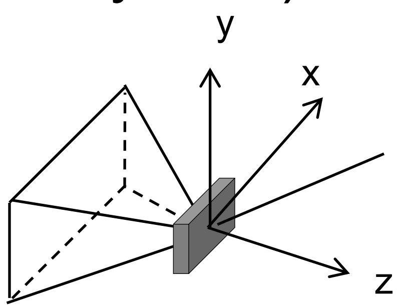
- The view transformation defines the position and orientation of the virtual camera/eye in the world coordinate system.
- In `glMatrix` library, these transformations are defined by setting the position, the upwards pointing axis of camera and where it looks at.

```
mat4.lookAt([camera_position],[point_to_look_at],[upward
pointing axis], destination_matrix)
```

- Once the vertex coordinates are transformed by the view transformation, the coordinate values of objects are defined with respect to the camera coordinate system.

Cont'd

- In WebGL, the default position of all objects, including camera, is at the origin of the world coordinate system.
- The default camera direction is such that it looks down the negative z-axis and its y-axis points upwards in the world coordinate system (the axes of the camera coordinate system are in alignment with those of the world coordinate system)

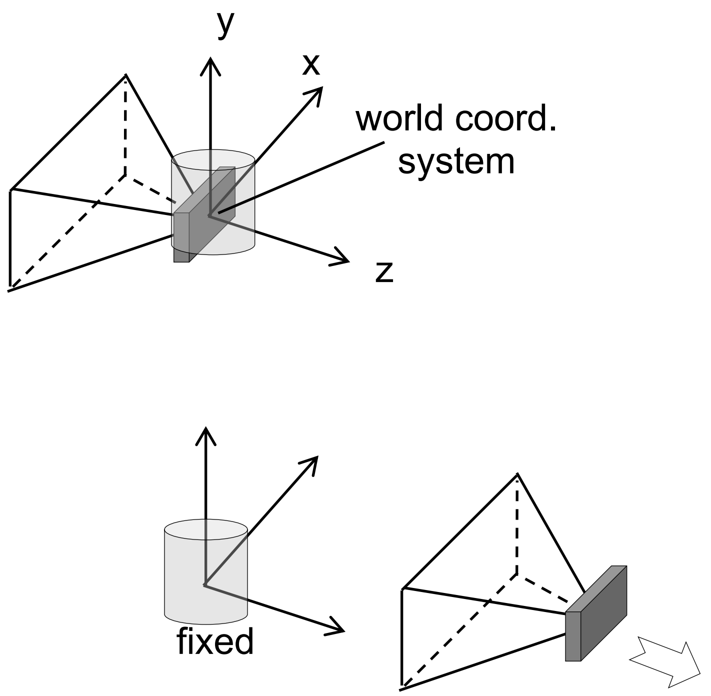


World coord. system and the default camera coord. system

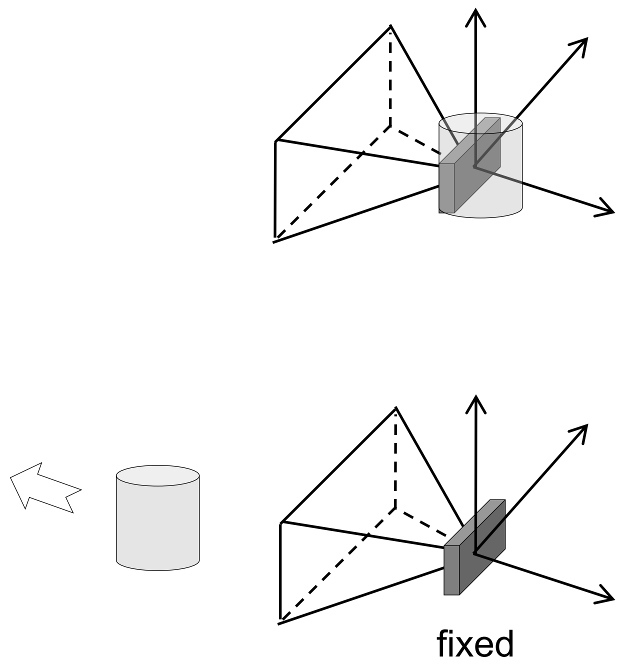
- To achieve the desired views, one must move the camera or the objects (otherwise they are at the same position in the world!).

Two Equivalent Ways

- Two equivalent ways to achieve the desired viewpoint.



Approach 1: move the camera (backwards)



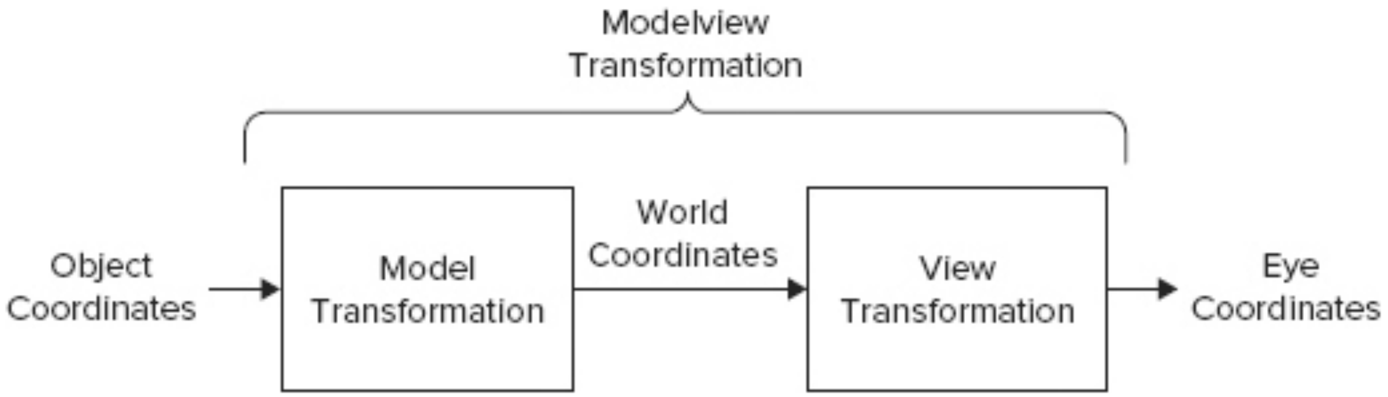
Approach 2: move the object (forwards)

Cont’d

- Imagine that you have both an object and a camera positioned at the origin of the world coordinate system. The camera is looking down the negative z-axis. In order to see the object, you have two options:
- Move the camera backwards — Since the camera is looking down along the negative z-axis by default, a backwards movement means moving the camera in the direction of positive z-axis.
- Move the object forward into the scene — This means to move the object along the negative z-axis.
- Both options produce the same visual effect.

Modelview Transformation

- A model transformation is unique to an object.
- The view transformation associates with the camera and is applicable to the entire scene.
- But they are normally combined as a single modelview transformation (by multiplying them together).
- When multiplied with a combined modelview matrix, the vertices of an object are converted directly from object's local coordinates to the camera/eye coordinates.



Perspective Transformation

- The perspective transformation implements the perspective projection and is applied *after* the modelview transformation.
- It determines the *view frustum* and transforms the view frustum into a cuboid (3D screen space) that ranges from –1 to +1 along x-and y-axis and 0 to 1 in *z* axis. Vertices and objects outside this normalised space will be clipped at later stages.
- The perspective projection transforms the coordinates of the vertices before they are pass on to `gl_Position` variable in the vertex shader (`gl_Position` expects all vertices that are visible on the screen to be defined in this cuboid).

An Analogy

- An analogy of these transformations is to compare them with using a camera and taking a photo of some objects:
  - the model transformation corresponds to positioning your objects in a scene,
  - the view transformation corresponds to positioning and pointing your camera, and
  - the projection transform corresponds to selecting a lens for your camera. The lens of a camera affects the field of view.

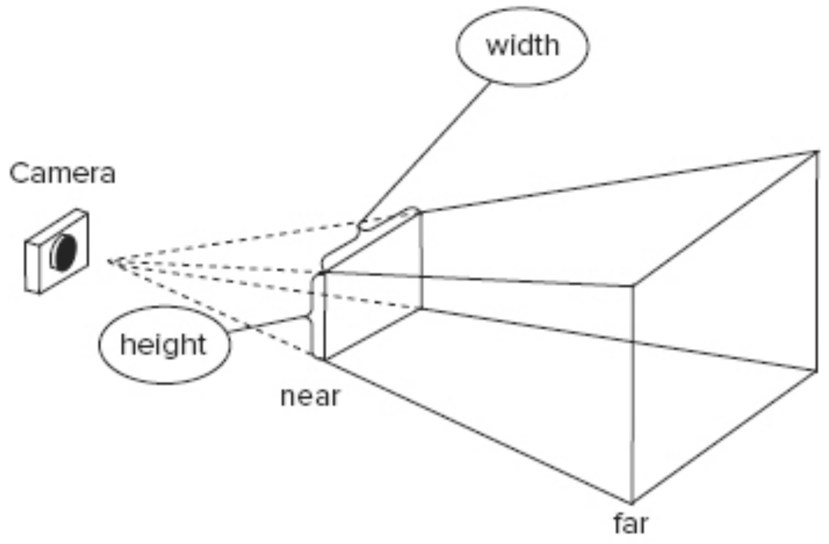
Set up Projection Matrix

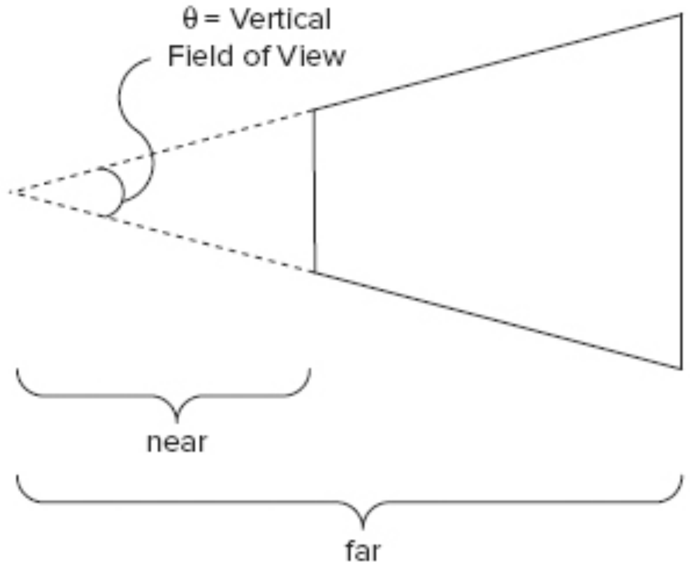
- There are two functions in glMatrix that can be used to set up the perspective projection: `mat4.perspective()` and `mat4.frustum()`.
- The prototype of the first function is as follows:

```
mat4.perspective(fovy, aspect, near, far,
  projectionMatrix)
```

Here

- `fovy` refers to the vertical view angle (in degrees)
- `aspect` is the aspect ratio (which is viewport width/viewport height). The aspect ratio is used to decide the horizontal field of view.
- `near` is the distance to the near plane of the frustum and
- `far` is the distance to the far plane of the frustum.
- `projectionMatrix` is the argument where the result of the method (i.e., the defined projection matrix) is written to .


width
Camera
height
near
far


θ = Vertical Field of View
near
far

Cont'd

- The second function, `mat4.frustum()`, has the prototype:

```
mat4.frustum(left, right, bottom, top, near, far,
  projectionMatrix)
```

where

- `left,right, bottom` and `top` sets the left, right, bottom and top bounds of the **near** clipping plane.
- `near` and `far` set the locations of the near and **far** clipping planes.
- `projectionMatrix` stores the created projection matrix.

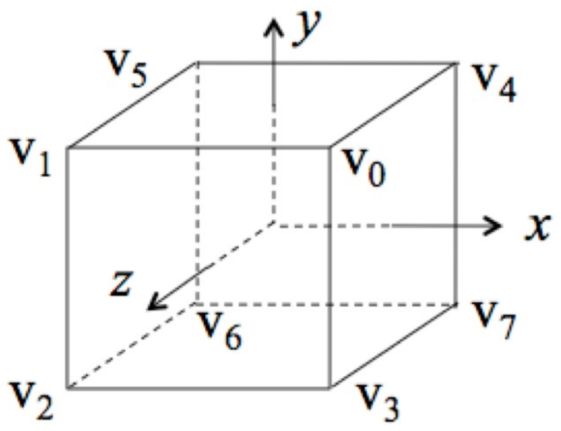
Use Transformations

- The general procedure of using local coordinates (defining objects) and transformations is as the following:
  1. set up WebGLBuffer objects for the local coordinates of the objects (as usual),
  2. set uniform variables in the vertex shader for the **modelview** and **projection matrices**.
  3. before calling any drawing method, create the correct/needed modelview matrix and the projection matrix (this work needs to be done only once if no camera animation is involved).
  4. **upload the modelview and projection matrices** to the vertex shader by the method `gl.uniformMatrix4fv()`.
  5. call drawing method `gl.drawArrays()` or `gl.drawElements()` (as usual).
- When the pipeline is working, the vertex shader performs the transformations by multiplying the matrices with the homogeneous coordinates (a 4d vector, locl coordinated) of a vertex (they are passed from the WebGLBuffer objects).

Define Objects in Local Coordinates

- In the previous lab sessions, we had not used any transformation and the coordinates were given in the normalised coordinates, which means that the coordinate values must be within the ranges from -1 to 1 for x,y and 0 to 1 for z (we had drawn 2D shapes only, so z was set to 0.0).
- By doing so, we had actually skipped the modelview transformation and the perspective transformation in the graphics pipeline.
- By using transformations,, we can now use the local coordinate to define shapes and ignore the restriction on the ranges of x, y and z values, e.g.,

```
var cubeVertexPosition = [
       2.0,  2.0,  2.0, //v0
      -2.0,  2.0,  2.0, //v1
      -2.0, -2.0,  2.0, //v2
       2.0, -2.0,  2.0, //v3
       2.0,  2.0, -2.0, //v4
      -2.0,  2.0, -2.0, //v5
      -2.0, -2.0, -2.0, //v6
       2.0, -2.0, -2.0, //v7
      ];
```



Set up Transformations

- Two 4x4 matrices are first created using standard function `mat4.create()` outside the drawing function (to avoid re-creating them in each draw).
- In function `draw()`, the perspective projection matrix is set by calling `mat4.perspective()` (this can be done outside draw method, if the view frustum remain unchanged at run time).
- The modelview transformation is first set to *identity* matrix, then camera transformation and other transformations are applied

```
modelViewMatrix = mat4.create();
projectionMatrix = mat4.create();

function draw() {

  ...

  mat4.perspective(60, gl.viewportWidth / gl.viewportHeight,
                   0.1, 100.0,   projectionMatrix);
  mat4.identity(modelViewMatrix);
  mat4.lookAt([8, 5, 10],[0, 0, 0], [0, 1, 0], modelViewMatrix);

  mat4.translate(modelViewMatrix, [0.0, 3.0, 0.0], modelViewMatrix);
```

```
mat4.perspective(fovy, aspect, near, far,
  projectionMatrix)
```

← Set view frustum

← Set camera position and orientation

← Move the camera upwards by 3 unit

Cont'd

- Two 4 x 4 matrices, `modelViewMatrix` and `perspectiveMatrix` are created by calling `mat4.create()`.
- A perspective projection matrix is created by calling `mat4.perspective()`:
  - the vertical field of view is 60 degrees,
  - the aspect ratio is aspect ratio of viewport,
  - the near & far planes are at 0.1 and 100.0 units from the viewer.
- The call to `mat4.identity()` sets the modelview matrix to identity. This is needed because the modelview matrix could hold values from the previous frame, which is probably not what you want.

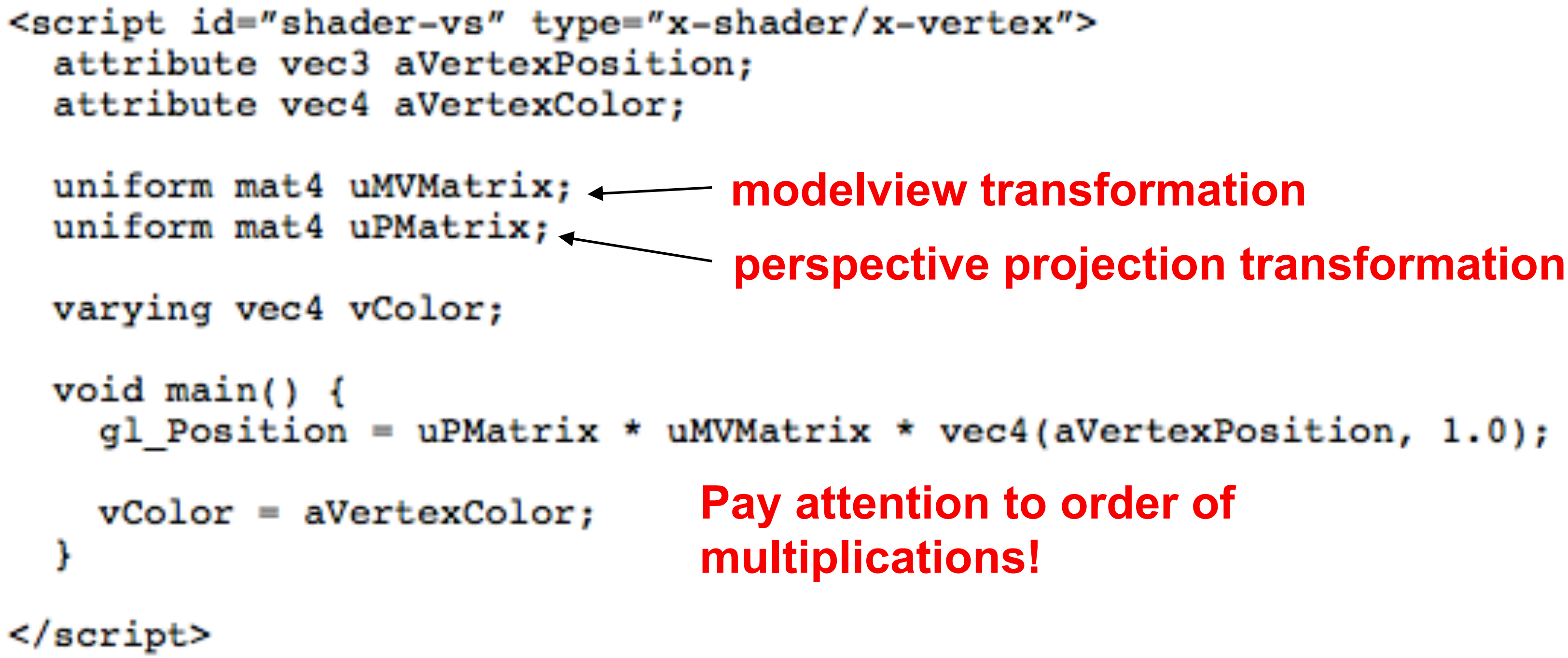
Cont’d

- Camera position & orientation (in world coordinate system) is set using `mat4.lookAt()`:

```
mat4.lookAt([8, 5, 10],[0, 0, 0], [0, 1, 0], modelViewMatrix);
```

  - the camera/viewer is located at the position (8, 5, 10).
  - The second argument specifies that the view direction is towards the origin, i.e., the z axis of camera’s coordinate system passes through the origin of the world coordinate system, (0,0,0).
  - The third argument specifies that the up direction is the positive y-axis.
- The call to `mat4.translate()` adds a translation of 3.0 units in the positive y-direction to the modelview matrix (move the camera upwards by 3 units).

In Vertex Shader…



```
<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;
  attribute vec4 aVertexColor;

  uniform mat4 uMVMatrix;
  uniform mat4 uPMatrix;

  varying vec4 vColor;

  void main() {
    gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);

    vColor = aVertexColor;
  }

</script>
```

modelview transformation

perspective projection transformation

Pay attention to order of multiplications!

- To transform a vertex, `aVertexPosition` (in object's local coordinates), to the normalised coordinates (`gl_Position`), we simply multiply it first with the modelview matrix (**`uMVMatrix`**) and then with the projection matrix (**`uPMatrix`**) and write the result to the built-in variable `gl_Position`:

Upload to Vertex Shader

- Uploading transformation is done in the same way as uploading vertex attribute but using the method `gl.uniformMatrix4fv()`.
- Recall the way of uploading vertex attribute:

Retrieves the pointer to `aVertexPosition` in the shader program

```
shaderProgram.vertexPositionAttribute = gl.getAttribLocation
                    (shaderProgram, "aVertexPosition");
```

Pass data in the buffer to `aVertexPosition` in the shader

```
gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                    . . .);
```

Cont'd

- Retrieve pointers to **uMVMatrix** and **uPMatrix**

```
shaderProgram.uniformMVMatrix = gl.getUniformLocation(
                         shaderProgram, "uMVMatrix");
shaderProgram.uniformProjMatrix = gl.getUniformLocation(
                         shaderProgram, "uPMatrix");
. . . .
```

Uploading transformations

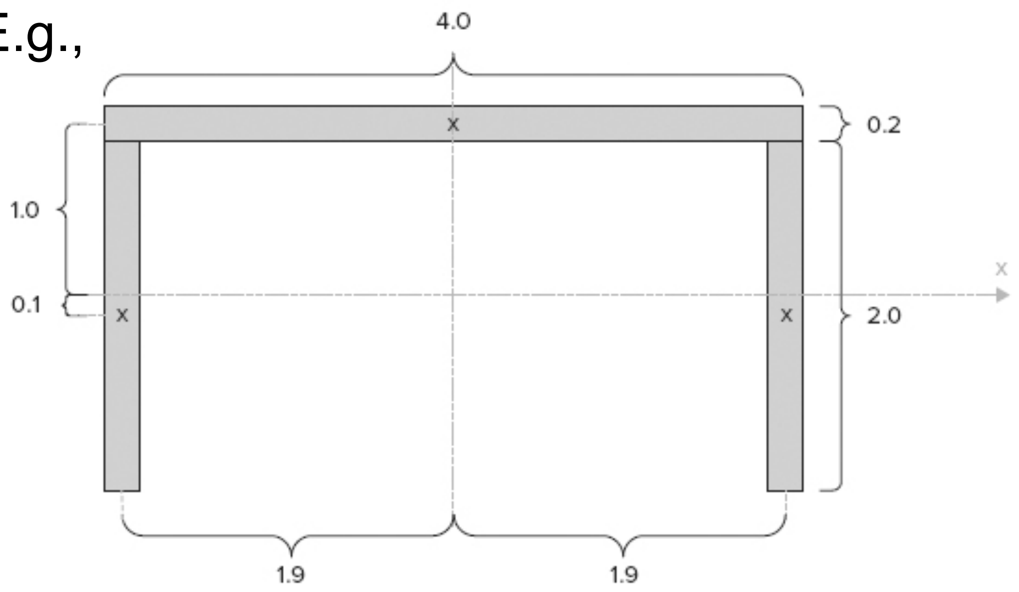
```
gl.uniformMatrix4fv(shaderProgram.uniformMVMatrix,
                         false, modelViewMatrix);
gl.uniformMatrix4fv(shaderProgram.uniformProjMatrix,

                         false, projectionMatrix);
```

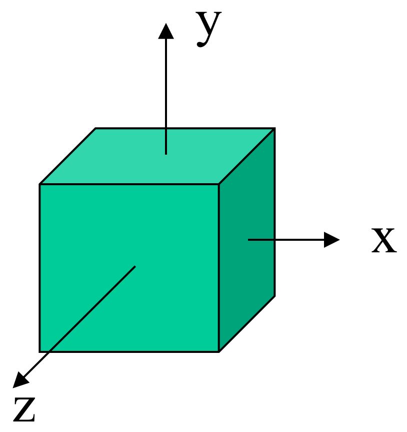
Cont’d

- In `uniformMatrix4fv`
  - the first argument specifies which uniform variable you want to load data into.
  - the second argument specifies whether you want to transpose the columns of the matrix that is uploaded. In WebGL this argument must be set to false.

Tracing Transformations

- Consider drawing a table. As discussed, we can get it by drawing transformed cubes at different positions and sizes:
  - To draw the table top, apply a translation from the cube's origin to where you want to the tabletop to be, then scale the cube to a flat cuboid that looks like a table top. Draw it.
  - To draw the legs, translate the cube to the position where you want the first table leg to be, scale the cube to a cuboid that looks like a table leg, and draw it.
  - ...
- E.g.,





Cont’d

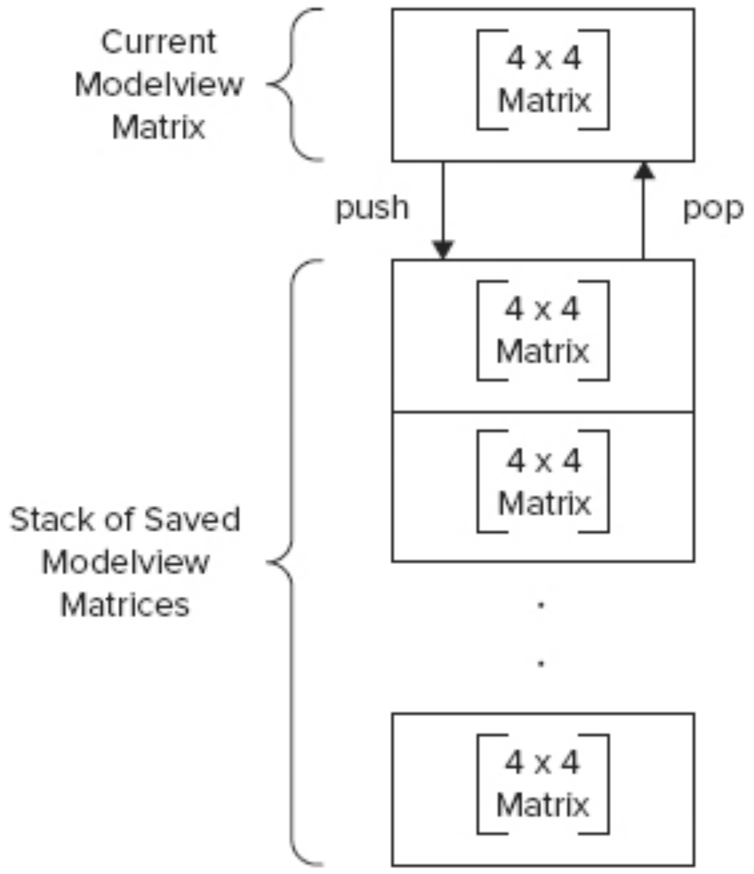
- To draw/create all the elements of the table, i.e., the top and the legs, from the same cube, we need to set up the modelview transformation for each of them, because they are different in dimensions and positions.
- After the tabletop is done, when construct the first leg, we must consider the transformations already applied to the modelview matrix. That is, we need to trace and remember what transformations have been applied to the modelview matrix so far, which is difficult.
- To avoid this difficulty, we often organise compound objects into hierarchies and save the modelview matrix at certain nodes of the hierarchies and recall the matrix when we want to come back to its previous state.

Cont’d

- For instance, we first save the modelview matrix and then transform the cube to have the tabletop.
- When we are done with the tabletop, instead of trying to translate directly from the position of the tabletop to the position of the first leg, we first restore the modelview transformation matrix that we saved before. This means that we are back to the size and position of the original cube.
- From here, you again save a copy of the original modelview matrix, translate and scale the cube to the position and size of a table leg, and draw it. When you are finished with the first table leg, you restore the saved transformation matrix (going back to the original cube). Then you continue with the second, third, and fourth table legs in the same way.
- A stack is an ideal tool for assisting the saving and retrieving of the modelview matrix

Transformation Stack

- A stack can be easily implemented using JavaScript array, which provide `push()` and `pop()` methods



Stack – JavaScript Array

```
modelViewMatrix = mat4.create();
modelViewMatrixStack = [];

function pushModelViewMatrix() {
  var copyToPush = mat4.create(modelViewMatrix);
  modelViewMatrixStack.push(copyToPush);
}

function popModelViewMatrix() {
  if (modelViewMatrixStack.length == 0) {
    throw "Error popModelViewMatrix() - Stack was empty ";
  }
  modelViewMatrix = modelViewMatrixStack.pop();
}
```

Use Stack

```
function drawTable(){
  // draw table top
  pushModelViewMatrix();
  mat4.translate(modelViewMatrix, [0.0, 1.0, 0.0], modelViewMatrix);
  mat4.scale(modelViewMatrix, [2.0, 0.1, 2.0], modelViewMatrix);
  uploadModelViewMatrixToShader();
  // draw the actual cube (now scaled to a cuboid) in brown color
  drawCube(0.72, 0.53, 0.04, 1.0); // argument sets brown color
  popModelViewMatrix();

  // draw table legs
  for (var i=-1; i<=1; i+=2) {
    for (var j= -1; j<=1; j+=2) {
→     pushModelViewMatrix();
      mat4.translate(modelViewMatrix, [i*1.9, -0.1, j*1.9], modelViewMatrix);
      mat4.scale(modelViewMatrix, [0.1, 1.0, 0.1], modelViewMatrix);
      uploadModelViewMatrixToShader();
      drawCube(0.72, 0.53, 0.04, 1.0); // argument sets brown color
→     popModelViewMatrix();
    }
```

Further Reading

- Anyuru, A., WebGL Programming – Develop 3D Graphics for the Web
  - Chapter 4