Recap: Viewing and Projection

⚠ syntax is OpenGL 1.0, not WebGL

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

gluLookAt(eyeX,eyeY,eyeZ,
          refX,refY,refX,
          upX,upY,upZ);
```

viewing transform

```
glPushMatrix();
…
glPopMatrix();
glPushMatrix();
…
glPopMatrix();

…
```

define the scene (modeling transforms and primitives)

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();

…
```

projection transform

```
glFrustum(xmin,xmax,ymin,ymax,
          near,far)
gluPerspective(fieldOfViewAngle,
               aspect,near,far)

glOrtho(xmin,xmax,ymin,ymax,
        near,far)
```

follow `glOrtho` with shear (shear first) for oblique parallel

Transformations in WebGL

- the programmable pipeline does not maintain modelview or projection matrices
  - now up to the programmer!
- three tasks
  - managing viewing pipeline transforms (modeling, viewing, projection) in JavaScript
  - supplying transforms as parameters to the vertex shader
  - applying transforms in the vertex shader

Managing Viewing Pipeline Transforms

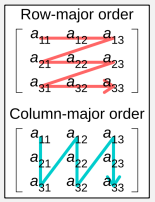
- similar to OpenGL 1.0, we'll keep track of projection and modelview matrices
  - maintained in JavaScript, then passed to the vertex shader

Using `glMatrix`

- `glMatrix` is a free JavaScript library implementing vector and matrix math
  - include in program with `<script src="gl-matrix-min.js"></script>` adjust path as appropriate
    - goes in `<head>` section

Using glMatrix

- defines types `vec2`, `vec3`, `vec4`, `mat3`, `mat4` for vectors and matrices
  - these are JavaScript types rather than GLSL types even though they have the same names
  - really just 1D arrays (regular JavaScript arrays or typed arrays of type `Float32Array`) with right number of elements
    - can pass an array with right number of elements whenever parameter is one of the vector or matrix types
    - matrix types use column-major order (compatible with WebGL)



https://commons.wikimedia.org/wiki/File:Row_and_column_major_order.svg

Using glMatrix

- functions
  - *type*`.create()` creates a typed array of the appropriate length
    - default is 0s for `vec`*i*, identity for `mat`*i*
  - *type*`.clone(`*param*`)` creates a copy of *param*

create projection and modelview matrices to replicate what OpenGL 1.0 maintains –

```
let modelview = mat4.create();     // identity
let projection = mat4.create();    // identity
```


Using glMatrix

- functions – transforms
  - transform functions set the value of the first parameter (which must have been allocated previously) instead of returning result
    - `mat4.multiply(A,B,C)`
    - `mat4.translate(A,B,[tx,ty,tz])`
    - `mat4.scale(A,B,[sx,sy,sz])`
    - `mat4.rotateX(A,B,radians)`
    - `mat4.rotateY(A,B,radians)`
    - `mat4.rotateZ(A,B,radians)`
    - `mat4.rotate(A,B,radians,[px,py,pz])`
      - axis of rotation is vector (0,0,0) → (px,py,pz)
    - `mat4.identity(A)`
      - sets A to the identity matrix

usage example –
to achieve the effect of `glTranslatef(dx,dy,dz)` as a modeling transform, use
`mat4.translate(modelview, modelview,[dx,dy,dz])`

Using glMatrix

- functions – viewing and projection
  - all set A (which must have been allocated previously) to the matrix defined
  - `mat4.lookAt(A,[eyex,eyey,eyez],[refx,refy,refz], [upx,upy,upz])`

    with the modelview matrix as A, this is equivalent to
    `glLoadIdentity();`
    `gluLookAt(eyex,eyey,eyez,refx,refy,refz,upx,upy,upz);`
  - `mat4.ortho(A,left,right,bottom,top,near,far)`
  - `mat4.frustum(A,left,right,bottom,top,near,far)`
  - `mat4.perspective(A,fieldOfView,aspect,near,far)`
    - `fieldOfView` in radians

    with the projection matrix as A, these are equivalent to
    `glLoadIdentity();`
    `glOrtho(left,right,bottom,top,near,far);`
    etc

Managing a Transform Stack

- JavaScript arrays provide push/pop operations

```
let matrixStack = [];
matrixStack.push(mat4.clone(modelview));
…
modelview = matrixStack.pop();
```

push a copy of the transform on the stack or else subsequent operations will continue to modify it

Managing Viewing Pipeline Transforms

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

gluLookAt(eyeX,eyeY,eyeZ,
          refX,refY,refX,
          upX,upY,upZ);

glPushMatrix();
glTranslatef(…);
glRotate(…);

// draw primitive
glPopMatrix();

glPushMatrix();
…
glPopMatrix();
…
```

⚠ syntax is OpenGL 1.0, not WebGL

```
let modelview = mat4.create();
let matrixStack = [];

mat4.lookAt(modelview,[eyex,eyey,eyez],
                      [refx,refy,refz],
                      [upx,upy,upz]);

matrixStack.push(mat4.clone(modelview));
mat4.translate(modelview,modelview,…);
mat4.rotate(modelview,modelview,…);
// setup shader arguments
// draw primitive
modelview = matrixStack.pop();

matrixStack.push(mat4.clone(modelview));
…
// setup shader arguments
// draw primitive
modelview = matrixStack.pop();
```


Managing Viewing Pipeline Transforms

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();

…
```

← projection transform

```
glFrustum(xmin,xmax,ymin,ymax,
          near,far)
gluPerspective(fieldOfViewAngle,
               aspect,near,far)

glOrtho(xmin,xmax,ymin,ymax,
        near,far)
```

follow glOrtho with shear (shear first) for oblique parallel

⚠ syntax is OpenGL 1.0, not WebGL

```
let projection = mat4.create();

mat4.frustum(projection,left,
             right,bottom,top,
             near,far);

mat4.perspective(projection,
                 fieldOfView,
                 aspect,
                 near,far)

mat4.ortho(projection,left,right,
           bottom,top,near,far);
```


Transforms in the Vertex Shader

- vertex shader is responsible for applying the viewing pipeline transformations
- matrices are passed as `uniform` variables – declared in shader
  - same value for whole primitive

```
attribute vec3 a_coords;     // vertex coordinates (object coords)
uniform mat4 u_modelview;    // modelview matrix
uniform mat4 u_projection;   // projection matrix

void main() {
  vec4 coords = vec4(a_coords,1.0);
  vec4 eyeCoords = u_modelview*coords;  // transform to eye coords
  gl_Position = u_projection*eyeCoords; // transform to clip coords
}
```

Passing Transforms to the Vertex Shader

- assigning values – done in JavaScript

```
u_modelview = gl.getUniformLocation(prog,"u_modelview");
gl.uniformMatrix4fv(u_modelview,false,modelview);
```

  - `u_modelview`, `modelview` are JavaScript variables
    - `modelview` is the modelview matrix
    - `u_modelview` is the shader argument location
  - "`u_modelview`" is the name of the shader parameter
  - similar for projection

```
attribute vec3 a_coords;     // vertex coordinates (object coords)
uniform mat4 u_modelview;    // modelview matrix
uniform mat4 u_projection;   // projection matrix
```


Mouse Rotators

- book provides `SimpleRotator` and `TrackballRotator`
- usage
  - create an instance

```
rotator = new SimpleRotator( canvas, callback, viewDistance );
```

    - *canvas* is the DOM canvas element where WebGL is rendering the scene
    - *callback* is the function that renders the scene (draw)
    - *viewDistance* must be between near and far e.g. halfway in between
  - configure rotation center if needed – default is [0,0,0]
    - should be the center of the scene i.e. what would be the reference point in `gluLookAt`

```
rotator.setRotationCenter([refx,refy,refz]);
```

  - initialize the modelview transform

```
let modelview = rotator.getViewMatrix();
```

    - replaces `mat4.lookAt` for defining the viewing transform