Tutorial 08 Animation and User Input

1. Animation

This tutorial implements a WebGL animation application, based upon the work done in the previous tutorials. The animation will implement two types of motions: the motion of the box on top of the table and real-time viewport control (or scene navigation). The motion of the box will involve a upward motion of 5 units (along y axis) in approximately 3 seconds and then followed by a circular motion along a path within the horizontal plane at a speed of one revolution per 2-second. The navigation control includes moving and rotating the scene along or around the coordinate axes at user's command via mouse and key actions. These actions are achieved by modifying the modelview transformation of the entire scene (e.g., for viewpoint control) or of the individual object (e.g., the flying box). Modifications to the modelview transformation have to be implemented in `draw ()` function so that the modelview transformation will be updated in each animation frame.

The animation loop in WebGL can be accessed by calling **`requestAnimFrame(`**`draw`**`)`** at the beginning of the draw function. The call requests for the next frame to be drawn by invoking the `draw` function when the current call to `draw` function returns, therefore a rendering loop is formed. The returned value of the function call **`requestAnimFrame(`**`draw`**`)`** is a non-zero integer ID number (We use it to call **`cancelRequestAnimFrame (ID)`**, see last tutorial). It is assigned to a variable, `requested`, in the program. At the beginning, the current system time is also taken. The time will be used to estimate the duration of the animation and the time to render a frame, which allows us to calculate the current position of an object from its speed specification.

```
requestId = requestAnimFrame(draw);
currentTime = Date.now();

. . .
if (animationStartTime === undefined) {
   animationStartTime = currentTime;
}

. . .

// update the modelview transformation for the entire scene
mat4.translate(pwgl.modelViewMatrix, [0.0, transY, transZ],
                        pwgl.modelViewMatrix);
mat4.rotateX(pwgl.modelViewMatrix, xRot/50,
                        pwgl.modelViewMatrix);
mat4.rotateY(pwgl.modelViewMatrix, yRot/50,
                        pwgl.modelViewMatrix);
yRot = xRot =transY=transZ=0;
. . .

drawFloor();
. . .
drawTable();


//Draw box.
if (pwgl.y < 5) {
   // First move the box vertically from its original position on
   // top of the table (where y = 2.7) to 5 units above the
   // floor (y = 5). Let this movement take 3 seconds
   pwgl.y = 2.7+(currentTime-pwgl.animationStartTime)/3000 * (5.0-
                                                    2.7);
} else {
   // Then move the box in a circle where one revolution takes 2 // seconds
   pwgl.angle = (currentTime - pwgl.animationStartTime)
                /2000*2*Math.PI % (2*Math.PI);
   pwgl.x = Math.cos(pwgl.angle) * pwgl.circleRadius;
   pwgl.z = Math.sin(pwgl.angle) * pwgl.circleRadius;
}

mat4.translate(pwgl.modelViewMatrix, [pwgl.x, pwgl.y, pwgl.z],
                        pwgl.modelViewMatrix);
mat4.scale(pwgl.modelViewMatrix, [0.5, 0.5, 0.5],
                        pwgl.modelViewMatrix);
```

```
uploadModelViewMatrixToShader();
drawCube(pwgl.boxTexture);
   .  .  .
```

2. User input

Interactive control is an important aspect of 3D graphics applications. For standard PCs, user-input devices are limited to mouse and keyboard and interactive control must be realised through such devices by monitoring key/mouse actions such as clicking and dragging of the mouse or pressing of the keys. These actions are monitored and handled by JavaScript *event handling*. Event handling is not a concept special for WebGL API or JavaScript. It is a part of most languages that support interactive applications. If you have some experiences in JavaScript, Java, C++, etc, you may have used it before. Here we briefly introduce the techniques of JavaScript key- and mouse-event handling that you will need in developing interactive WebGL applications.

2.1 Key event handling

In general, three keyboard events are generated when an alphanumeric key is pressed:

- a *keydown* event is first generated.
- a *keypress* event follows immediately.
- a *keyup* event is generated when the key is released, .

The *keydown* and *keyup* events are actually different from the *keypress* event. The *keydown* and *keyup* events represent physical keys that are pressed down or released, while the *keypress* event represents which character is typed. A key event has two properties:

- *keyCode* contains the ASCII code for the uppercase version of the key. E.g., the key labelled "A" has a keycode 65, regardless of whether or not Caps lock is enabled.
- *charCode* gives you the ASCII value for the resulting character, i.e., "A" or "a"

The following example demonstrates the differences:

```
document.addEventListener('keydown', handleKeyDown, false);
document.addEventListener('keyup', handleKeyUp, false);
document.addEventListener('keypress', handleKeyPress, false);
   .  .  .
function handleKeyDown(event) {
      console.log('keydown - keyCode=%d, charCode=%d',
                  event.keyCode, event.charCode);
}

function handleKeyUp(event) {
      console.log('keyup - keyCode=%d, charCode=%d',
                  event.keyCode, event.charCode);
}

function handleKeyPress(event) {
      console.log('keypress - keyCode=%d, charCode=%d',
                  event.keyCode, event.charCode);
}
```

In Firefox, pressing the key "A" generates the following result in the console:

```
keydown - keyCode=65, charCode=0
keypress - keyCode=0, charCode=97
keyup - keyCode=65, charCode=0
```

2.2 Handling multiple keys

For some applications, you need to handle multiple keys at the same time, e.g., in games. To monitor which keys are being pressed at any time, a list of the keys that have been pressed needs to be maintained.

Tracking Pressed Keys

```
var listOfPressedKeys={};//object for keeping the list of pressed keys

//keydown event handler
function handleKeyDown(event) {
      // On a keydown event, any immediate actions are handled first
      if (String.fromCharCode(event.keyCode) == "S") {
```

```
                // Fire a weapon
                fireMissile();
        } else if (String.fromCharCode(event.keyCode) == "W") {
                doSomething();
        } else if()
                . . .
        }
        // Then store the information about which key has been pressed
        listOfPressedKeys[event.keyCode] = true;
}

//Keyup event handler
function handleKeyUp(event) {
        // Update list of keys that are pressed down
        listOfPressedKeys[event.keyCode] = false;
}
...
```

Process Pressed Keys
This function is called each frame in the animation loop. Suppose we want to monitor the arrow keys:

```
function monitorPressedKeys() {
        if (listOfPressedKeys[38]) {
                // Arrow up, acceleration
                speed += 0.5;
        }
        if (listOfPressedKeys[40]) {
                // Arrow down, deceleration
                speed -= 0.5;
        }
        if (listOfPressedKeys[37]) {
                // Arrow left, the user wants to turn left.
                turnLeft();//function to handle turn left
        }
        if (listOfPressedKeys[39]) {
                // Arrow right, the user wants to turn right.
                turnRight();//function to handle turn right
        }
}
```

2.3 Mouse Events

HTML 5 specification defines many mouse events. Some are simple events, e.g., *mousedown, mouseup, mousemove, mouseover, and mouseout*. Most browsers support them. Some are complex events, e.g., click, double click, mousedrag, etc. Such complex events might be useful for graphics applications, e.g., use mousedrag to rotate the viewport. Unfortunately, not all browsers support complex event (e.g., Firefox might not support the mousedrag event).

In this tutorial, we will use simple mouse events **`mousedown`**, **`mouseup`** and **`mousemove`** to implement the viewport control. We utilise the properties *clientX* and *clientY*, which contain the distance from the mouse pointer to the upper-left corner of the browser's viewport, to specify the amount of translations or rotations. The *mousedown* and *mouseup* events also contain a property called button index indicating which mouse button is pressed or released.

- 0 – the left mouse button
- 1 – the middle mouse button/wheel; and
- 2 – the right mouse button.

Exercise:
Complete the provided incomplete program by providing the functions **`draw()`** and **`startup()`** and other missing statements or functions.

User input:

- up & down arrows: change the diameter of the path of the cube
- mouse drag: rotate the scene
- Alt+mouse drag (or alt+mouse whel): translation in y
- Shift+ mouse drag (mouse wheel): translation in z

Note: If you start from your own program from the last session, pay attention to the part of the *lost context event handling* function and variable for the increment control of the background colour. In this tutorial, the lost context control is still in place, but the simulator has been removed.

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<title>Animation with Key and Mouse Input</title>
<script src="webgl-debug.js"></script>
<script type="text/javascript" src="glMatrix.js"></script>
<script src="webgl-utils.js"></script>
<meta charset="utf-8">

<script id="shader-vs" type="x-shader/x-vertex">
   Vertex shader
    ...
</script>

<script id="shader-fs" type="x-shader/x-fragment">

</script>
   Fragment shader
    ...
</script>


<script type="text/javascript">
// globals
var gl;
var pwgl = {};
pwgl.ongoingImageLoads = [];
var canvas;

// variables for translations and rotations
var transY = 0, transZ=0;
var xRot =yRot =zRot =xOffs = yOffs = drag = 0;
// Keep track of pressed down keys in a list
pwgl.listOfPressedKeys = [];

function createGLContext(canvas) {
      ...
      return context;
}

function loadShaderFromDOM(id) {
    ...
    return shader;
}

function setupShaders() {
```

```
    ...
}

function pushModelViewMatrix() {
    ...
}

function popModelViewMatrix() {
    ...
}

function setupFloorBuffers() {
    pwgl.floorVertexPositionBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, pwgl.floorVertexPositionBuffer);

    var floorVertexPosition = [
        // Plane in y=0
          5.0,   0.0,  5.0,  //v0
          5.0,   0.0, -5.0,  //v1
         -5.0,   0.0, -5.0,  //v2
         -5.0,   0.0,  5.0]; //v3

    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(floorVertexPosition),
    gl.STATIC_DRAW);
    pwgl.FLOOR_VERTEX_POS_BUF_ITEM_SIZE = 3;
    pwgl.FLOOR_VERTEX_POS_BUF_NUM_ITEMS = 4;
    pwgl.floorVertexTextureCoordinateBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER,
    pwgl.floorVertexTextureCoordinateBuffer);

    var floorVertexTextureCoordinates = [
      2.0, 0.0,
      2.0, 2.0,
      0.0, 2.0,
      0.0, 0.0
    ];

    gl.bufferData(gl.ARRAY_BUFFER, new
    Float32Array(floorVertexTextureCoordinates),
                              gl.STATIC_DRAW);
    pwgl.FLOOR_VERTEX_TEX_COORD_BUF_ITEM_SIZE = 2;
    pwgl.FLOOR_VERTEX_TEX_COORD_BUF_NUM_ITEMS = 4;

    pwgl.floorVertexIndexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, pwgl.floorVertexIndexBuffer);
    var floorVertexIndices = [0, 1, 2, 3];
    gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new
    Uint16Array(floorVertexIndices), gl.STATIC_DRAW);
    pwgl.FLOOR_VERTEX_INDEX_BUF_ITEM_SIZE = 1;
    pwgl.FLOOR_VERTEX_INDEX_BUF_NUM_ITEMS = 4;
}

function setupCubeBuffers() {
    pwgl.cubeVertexPositionBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, pwgl.cubeVertexPositionBuffer);

    var cubeVertexPosition = [

        vertex coordinates
        ...

      ];

    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(cubeVertexPosition),
```

```
                        gl.STATIC_DRAW);
    pwgl.CUBE_VERTEX_POS_BUF_ITEM_SIZE = 3;
    pwgl.CUBE_VERTEX_POS_BUF_NUM_ITEMS = 24;

    // Setup buffer with texture coordinates
    pwgl.cubeVertexTextureCoordinateBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER,
    pwgl.cubeVertexTextureCoordinateBuffer);

    var textureCoordinates = [

        Texture coordinates
          ...

    ];

    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(textureCoordinates),
                            gl.STATIC_DRAW);
    pwgl.CUBE_VERTEX_TEX_COORD_BUF_ITEM_SIZE = 2;
    pwgl.CUBE_VERTEX_TEX_COORD_BUF_NUM_ITEMS = 24;



    pwgl.cubeVertexIndexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, pwgl.cubeVertexIndexBuffer);

    var cubeVertexIndices = [

      vertex indices
      ...

        ];

    gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new
    Uint16Array(cubeVertexIndices),
                          gl.STATIC_DRAW);
    pwgl.CUBE_VERTEX_INDEX_BUF_ITEM_SIZE = 1;
    pwgl.CUBE_VERTEX_INDEX_BUF_NUM_ITEMS = 36;
}

function textureFinishedLoading(image, texture) {
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA,
                              gl.UNSIGNED_BYTE, image);
    gl.generateMipmap(gl.TEXTURE_2D);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER,
                              gl.LINEAR);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
                              gl.LINEAR);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S,
                              gl.MIRRORED_REPEAT);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T,
                              gl.MIRRORED_REPEAT);
    gl.bindTexture(gl.TEXTURE_2D, null);
}

function loadImageForTexture(url, texture) {
    var image = new Image();
    image.onload = function() {
      pwgl.ongoingImageLoads.splice(
                  pwgl.ongoingImageLoads.indexOf(image), 1);
      textureFinishedLoading(image, texture);
    }
```

```
    pwgl.ongoingImageLoads.push(image);
    image.src = url;
}

function setupTextures() {
    // Texture for the table
    pwgl.woodTexture = gl.createTexture();
    loadImageForTexture("wood_128x128.jpg", pwgl.woodTexture);

    // Texture for the floor
    pwgl.groundTexture = gl.createTexture();
    loadImageForTexture("wood_floor_256.jpg",pwgl.groundTexture);

    // Texture for the box on the table
    pwgl.boxTexture = gl.createTexture();
    loadImageForTexture("wicker_256.jpg", pwgl.boxTexture);
}

function setupBuffers() {
   setupFloorBuffers();
   setupCubeBuffers();
}

function uploadModelViewMatrixToShader() {
      gl.uniformMatrix4fv(pwgl.uniformMVMatrixLoc, false,
                                    pwgl.modelViewMatrix);
}

function uploadProjectionMatrixToShader() {
    gl.uniformMatrix4fv(pwgl.uniformProjMatrixLoc, false,
                                    pwgl.projectionMatrix);
}

function drawFloor() {
    // Bind position buffer
    gl.bindBuffer(gl.ARRAY_BUFFER, pwgl.floorVertexPositionBuffer);
    gl.vertexAttribPointer(pwgl.vertexPositionAttributeLoc,
                  pwgl.FLOOR_VERTEX_POS_BUF_ITEM_SIZE,
                  gl.FLOAT, false, 0, 0);

    // Bind texture coordinate buffer
    gl.bindBuffer(gl.ARRAY_BUFFER,
    pwgl.floorVertexTextureCoordinateBuffer);
    gl.vertexAttribPointer(pwgl.vertexTextureAttributeLoc,
                  pwgl.FLOOR_VERTEX_TEX_COORD_BUF_ITEM_SIZE,
                  gl.FLOAT, false, 0, 0);
    gl.activeTexture(gl.TEXTURE0);
    gl.bindTexture(gl.TEXTURE_2D, pwgl.groundTexture);

    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, pwgl.floorVertexIndexBuffer);
    gl.drawElements(gl.TRIANGLE_FAN,
                  pwgl.FLOOR_VERTEX_INDEX_BUF_NUM_ITEMS,
                  gl.UNSIGNED_SHORT, 0);
}

function drawCube(texture) {

    // Bind position buffer
    gl.bindBuffer(gl.ARRAY_BUFFER, pwgl.cubeVertexPositionBuffer);
    gl.vertexAttribPointer(pwgl.vertexPositionAttributeLoc,
                  pwgl.CUBE_VERTEX_POS_BUF_ITEM_SIZE,
                  gl.FLOAT, false, 0, 0);

    // bind texture coordinate buffer
```

```
    gl.bindBuffer(gl.ARRAY_BUFFER,
    pwgl.cubeVertexTextureCoordinateBuffer);
    gl.vertexAttribPointer(pwgl.vertexTextureAttributeLoc,

    pwgl.CUBE_VERTEX_TEX_COORD_BUF_ITEM_SIZE,gl.FLOAT,false, 0, 0);
    gl.activeTexture(gl.TEXTURE0);
    gl.bindTexture(gl.TEXTURE_2D, texture);

    // Bind index buffer and draw cube
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, pwgl.cubeVertexIndexBuffer);
    gl.drawElements(gl.TRIANGLES, pwgl.CUBE_VERTEX_INDEX_BUF_NUM_ITEMS,
                                gl.UNSIGNED_SHORT, 0);
}

function drawTable(){

    // setup transformations for table top
    pushModelViewMatrix();
    mat4.translate(pwgl.modelViewMatrix, [0.0, 1.0, 0.0],
    pwgl.modelViewMatrix);
    mat4.scale(pwgl.modelViewMatrix, [2.0, 0.1, 2.0],
    pwgl.modelViewMatrix);
    uploadModelViewMatrixToShader();

    // Draw the table top with woodTexture
    drawCube(pwgl.woodTexture);
    popModelViewMatrix();

    // Draw the table legs
    for (var i=-1; i<=1; i+=2) {
       for (var j= -1; j<=1; j+=2) {
          pushModelViewMatrix();
          mat4.translate(pwgl.modelViewMatrix, [i*1.9, -0.1, j*1.9],
          pwgl.modelViewMatrix);
          mat4.scale(pwgl.modelViewMatrix, [0.1, 1.0, 0.1],
          pwgl.modelViewMatrix);
          uploadModelViewMatrixToShader();
          drawCube(pwgl.woodTexture);
          popModelViewMatrix();
       }
    }
}

function handleContextLost(event) {
    event.preventDefault();
    cancelRequestAnimFrame(pwgl.requestId);
    // Ignore all ongoing image loads by removing their onload handler
    for (var i = 0; i < pwgl.ongoingImageLoads.length; i++) {
       pwgl.ongoingImageLoads[i].onload = undefined;
    }
    pwgl.ongoingImageLoads = [];
}

function handleContextRestored(event) {
    init();
    pwgl.requestId = requestAnimFrame(draw,canvas);
}

function init() {
    // Initialization that is performed during first startup and when the
    // event webglcontextrestored is received is included in this
    function.
    setupShaders();
    setupBuffers();
```

```
    setupTextures();
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    gl.enable(gl.DEPTH_TEST);

    // Initialize some varibles for the moving box
    pwgl.x = 0.0;
    pwgl.y = 2.7;
    pwgl.z = 0.0;
    pwgl.circleRadius = 4.0;
    pwgl.angle = 0;

    // Initialize some variables related to the animation
    pwgl.animationStartTime = undefined;
    pwgl.nbrOfFramesForFPS = 0;
    pwgl.previousFrameTimeStamp = Date.now();

    gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

    mat4.perspective(60, gl.viewportWidth / gl.viewportHeight,
          1, 100.0, pwgl.projectionMatrix);
    mat4.identity(pwgl.modelViewMatrix);
    mat4.lookAt([8, 12, 8],[0, 0, 0], [0, 1,0], pwgl.modelViewMatrix);
}



function draw() {
    pwgl.requestId = requestAnimFrame(draw);

    var currentTime = Date.now();


    handlePressedDownKeys();

    // Update FPS if a second or more has passed since last FPS update
    if(currentTime - pwgl.previousFrameTimeStamp >= 1000) {
       pwgl.fpsCounter.innerHTML = pwgl.nbrOfFramesForFPS;
       pwgl.nbrOfFramesForFPS = 0;
       pwgl.previousFrameTimeStamp = currentTime;
    }

    //console.log("1   xRot= "+xRot+"    yRot="+yRot+"  t= "+transl);
    mat4.translate(pwgl.modelViewMatrix, [0.0, transY, transZ],
                                          pwgl.modelViewMatrix);
    mat4.rotateX(pwgl.modelViewMatrix, xRot/50, pwgl.modelViewMatrix);
    mat4.rotateY(pwgl.modelViewMatrix, yRot/50, pwgl.modelViewMatrix);
    //mat4.rotateZ(pwgl.modelViewMatrix, zRot/50, pwgl.modelViewMatrix);
    yRot = xRot = zRot =transY=transZ=0;

    uploadModelViewMatrixToShader();
    uploadProjectionMatrixToShader();
    //Note: in uniform1i next line "1" is "one" not "L"!! Check WebGL for
    //uniform2i, 2v,3i, 3v
    gl.uniform1i(pwgl.uniformSamplerLoc, 0);
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

    drawFloor();

    // Draw table
    pushModelViewMatrix();
    mat4.translate(pwgl.modelViewMatrix, [0.0, 1.1, 0.0],
    pwgl.modelViewMatrix);
    uploadModelViewMatrixToShader();
```

```
    drawTable();
    popModelViewMatrix();

    //Draw box.
    // Calculate the position for the box that is initially
    // on top of the table but will then be moved during animation
    pushModelViewMatrix();
    if (currentTime === undefined) {
      currentTime = Date.now();
    }
    if (pwgl.animationStartTime === undefined) {
      pwgl.animationStartTime = currentTime;
    }

    // Update the position of the box
    if (pwgl.y < 5) {
      // First move the box vertically from its original position on
      // top of the table (where y = 2.7) to 5 units above the
      // floor (y = 5). Let this movement take 3 seconds
      pwgl.y = 2.7+(currentTime-pwgl.animationStartTime)/3000 * (5.0-
                                                    2.7);
    } else {
      // Then move the box in a circle where one revolution takes 2 //
      seconds
      pwgl.angle = (currentTime - pwgl.animationStartTime)
                  /2000*2*Math.PI % (2*Math.PI);
      pwgl.x = Math.cos(pwgl.angle) * pwgl.circleRadius;
      pwgl.z = Math.sin(pwgl.angle) * pwgl.circleRadius;
    }

    mat4.translate(pwgl.modelViewMatrix, [pwgl.x, pwgl.y, pwgl.z],
                        pwgl.modelViewMatrix);
    mat4.scale(pwgl.modelViewMatrix, [0.5, 0.5, 0.5],
    pwgl.modelViewMatrix);
    uploadModelViewMatrixToShader();
    drawCube(pwgl.boxTexture);
    popModelViewMatrix();

    // Update number of drawn frames to be able to count fps
    pwgl.nbrOfFramesForFPS++;
}

function handleKeyDown(event) {
    pwgl.listOfPressedKeys[event.keyCode] = true;
}

function handleKeyUp(event) {
    pwgl.listOfPressedKeys[event.keyCode] = false;
}

function handlePressedDownKeys() {
    if (pwgl.listOfPressedKeys[38]) {
      // Arrow up, increase radius of circle
      pwgl.circleRadius += 0.1;
    }

    if (pwgl.listOfPressedKeys[40]) {
      // Arrow down, decrease radius of circle
      pwgl.circleRadius -= 0.1;
      if (pwgl.circleRadius < 0) {
         pwgl.circleRadius = 0;
      }
    }
}
```

```
function mymousedown( ev ){
    drag  = 1;
    xOffs = ev.clientX;
    yOffs = ev.clientY;
}

function mymouseup( ev ){
    drag  = 0;
}

function mymousemove( ev ){
    if ( drag == 0 ) return;
    if ( ev.shiftKey ) {
       transZ = (ev.clientY - yOffs)/10;
       //zRot = (xOffs - ev.clientX)*.3;
    } else if (ev.altKey) {
       transY = -(ev.clientY - yOffs)/10;
    } else {
       yRot = - xOffs + ev.clientX;
       xRot = - yOffs + ev.clientY;
    }
    xOffs = ev.clientX;
    yOffs = ev.clientY;
    //console.log("xOff= "+xOffs+"    yOff="+yOffs);
}

function wheelHandler(ev) {
    if (ev.altKey) transY = -ev.detail/10;
    else transZ =ev.detail/10;
    //console.log("delta ="+ev.detail);
    ev.preventDefault();
}

function startup() {
    canvas = document.getElementById("myGLCanvas");
    canvas = WebGLDebugUtils.makeLostContextSimulatingCanvas(canvas);
    canvas.addEventListener('webglcontextlost', handleContextLost, false);
    canvas.addEventListener('webglcontextrestored', handleContextRestored,
    false);
    document.addEventListener('keydown', handleKeyDown, false);
    document.addEventListener('keyup', handleKeyUp, false);
    canvas.addEventListener('mousemove', mymousemove, false);
    canvas.addEventListener('mousedown', mymousedown, false);
    canvas.addEventListener('mouseup', mymouseup, false);
    canvas.addEventListener('mousewheel', wheelHandler, false);
    canvas.addEventListener('DOMMouseScroll', wheelHandler, false);

    gl = createGLContext(canvas);

    init();

    pwgl.fpsCounter = document.getElementById("fps");

    // Draw the complete scene
    draw();
}

</script>

</head>
<body onload="startup();">
<canvas id="myGLCanvas" width="500" height="500"></canvas>
<div id="fps-counter"> FPS: <span id="fps">--</span></div>
```

```
</body>
</html>
```