Homework2

Pignata Giovanni 1913547

The Hierarchical Model

In order to create a Hierarchical Model of a **Sheep**, I used a l**eft-child, right-sibling tree structure**. This structure was composed of 11 Nodes for the sheep plus one for the grass and one for the fence, for a total of **13 nodes**.

Each node presents the following structure and it was initialized in the script using the function `CreateNode(Id)`:

```
var node = {
            id:
            transform:
            child:
            sibling:
            render: function(){...}
```

The `transform` parameter corresponds to the current transformation matrix that loaded during the tree **traversal**. It is set for each node using the function `transformNode(Id)`:

```
function transformNode(Id) {
    // Translate to the relative center..
    var m = translate(roots[Id][0],roots[Id][1],roots[Id][2]);
    // Rotate..
    m = mult(m,rotate(theta[Id], vec3(0,0,1)));
    figure[Id].transform = m;
}
```

The function `render()` is the function used to draw a node (a cube), and its code is the following:

```
function render(){
//  Scale ModelViewMatrix..
```

```
    var m = scale(l[Id],h[Id],w[Id]);
    m = mult(modelViewMatrix, m);
    gl.uniformMatrix4fv(modelViewMatrixLoc, false, flatten(m));

//  Link the color..
    gl.uniform4fv(colorLoc, textureColors[colors[Id]]);
//  Draw..
    if (Id == HEAD){
        gl.drawArrays(gl.TRIANGLES, 0, 6);
        gl.uniform4fv(colorLoc, textureColors[1]);
        gl.activeTexture(gl.TEXTURE1);
        gl.uniform1i( gl.getUniformLocation(program,
"uTextureMap"), 1);
        gl.uniform1i( gl.getUniformLocation(program, "uBump"),
false);
        gl.drawArrays(gl.TRIANGLES, 6, 6);
        gl.uniform4fv(colorLoc, textureColors[colors[Id]]);
        gl.activeTexture(gl.TEXTURE0);
        gl.uniform1i( gl.getUniformLocation(program,
"uTextureMap"), 0);
        gl.uniform1i( gl.getUniformLocation(program, "uBump"),
true);
        gl.drawArrays(gl.TRIANGLES, 12, 24);
    }
    if (Id == GRASS){
        gl.activeTexture(gl.TEXTURE2);
        gl.uniform1i( gl.getUniformLocation(program,
"uTextureMap"), 2);
        gl.drawArrays(gl.TRIANGLES, 0, nVertices);
        gl.activeTexture(gl.TEXTURE0);
        gl.uniform1i( gl.getUniformLocation(program,
"uTextureMap"), 0);
    }
    else gl.drawArrays(gl.TRIANGLES, 0, nVertices);
}
```

As it is possible to read, in the render() function some **uniform** parameters (as the modelview matrix and the colored texture) are updated for the specific node before calling the `gl.drawArrays()` function, that draw the final (scaled) cube.

Respect to the original code, I slightly modified the function `quad()` used to compute the coordinates of each face of the cube, in order to add additional information as the texture coordinates, the tangents and the normals (for bumping) :

```
        function quad(a, b, c, d) {
            var vertices = [...];
            var texCoord = [...];
            var quadIndex = [a,b,c, a,c,d];
            var texIndex = [0,1,2, 0,2,3];

            var t1 = subtract(vertices[b], vertices[a]);
            var t2 = subtract(vertices[c], vertices[b]);
            var normal = cross(t1, t2);
            normal = vec3(normal[0],normal[1],normal[2]);
            var tangent = vec3(t1[0],t1[1],t1[2]);

            for(var i = 0; i<6; i++){
                symbol.Points.push(vertices[quadIndex[i]]);
                symbol.Texture.push(texCoord[texIndex[i]]);
                symbol.Normals.push(normal);
                symbol.Tangents.push(tangent);
            }
        }
```

I considered the grass field and the fence as cubes, then I inserted them in the node structure as new siblings.

Textures

For the **bump** texture I used the same function as in Homework1:

```
function roughTextureMap(texSize){
    //  Bump Data:
        var data = new Array()
        for (var i = 0; i<= texSize; i++)  data[i] = new Array();
        for (var i = 0; i<= texSize; i++) for (var j=0; j<=texSize;
j++)
            data[i][j] = Math.random();
    //  Bump Map Normals:
        var normalst = new Array()
        for (var i=0; i<texSize; i++)  normalst[i] = new Array();
```

```
        for (var i=0; i<texSize; i++) for ( var j = 0; j < texSize;
j++)
            normalst[i][j] = new Array();
        for (var i=0; i<texSize; i++) for ( var j = 0; j < texSize;
j++) {
            normalst[i][j][0] = data[i][j]-data[i+1][j];
            normalst[i][j][1] = data[i][j]-data[i][j+1];
            normalst[i][j][2] = 1;
        }
    //  Scale to Texture Coordinates..
        for (var i=0; i<texSize; i++) for (var j=0; j<texSize; j++)
{
            var d = 0;
            for(k=0;k<3;k++) d+=normalst[i][j][k]*normalst[i][j]
[k];
            d = Math.sqrt(d);
            for(k=0;k<3;k++) normalst[i][j][k]= 0.5*normalst[i][j]
[k]/d + 0.5;
        }
        var normals = new Uint8Array(3*texSize*texSize);
        for ( var i = 0; i < texSize; i++ ){
            for ( var j = 0; j < texSize; j++ ) {
                  for(var k =0; k<3; k++){
                      normals[3*texSize*i+3*j+k] = 255*normalst[i]
[j][k];
                  }
            }
        }
        return normals;
    }
```

I applied the bump texture to the whole body of the sheep (face excluded), the fence and the grass field, but in this last case with a different `texSize`.

For the face, I computed a very simple texture:

```
function faceTexture(texSize){
        var texels = new Uint8Array(3*texSize*texSize);
        for(var i= 0; i<3*texSize*texSize; i+=3){
            var c = 255;
            texels[i] = c;
            texels[i+1] = c;
            texels[i+1] = c;
        }
        return texels;
    }
```

In the shaders, I controlled which style of texture (bump or simple colored) to use with the `uniform bool uBump` variable and a code very similar to the one used in the first Homework, with the only **diffuse** parameter for the color.

```
// Vertex Shader
void main() {
    gl_Position = uProjectionMatrix * uModelViewMatrix * aPosition;
    vColor = uColor;
    vTexCoord = aTexCoord;
    vec3 pos = (uModelViewMatrix * aPosition).xyz;
    vec3 light = uLightPosition.xyz;
    vec4 NN = vec4(aNormal,0);
    vec3 T  = normalize(uNormalMatrix*aTangent);
    vec3 M = normalize(uNormalMatrix*aNormal);
    vec3 B = cross(M, T);
    if (uBump){
    L.x = dot(T, light-pos);
    L.y = dot(B, light-pos);
    L.z = dot(M, light-pos);
    L = normalize(L);
    }
    else L = normalize(light - pos);
}
```

```
// Fragment Shader
void main() {
    vec4 MM = texture(uTexMap, vTexCoord);
    vec3 M = (uBump) ? normalize(2.0*MM.xyz-1.0) :
normalize(MM.xyz);
    vec3 LL = normalize(L);
    float Kd = max(dot(M,LL), 0.0);
    fColor = (uBump) ? (Kd*vColor) :(Kd*vColor);
    fColor.a = 1.0;
}
```

Camera Motion

I fixed the camera distance from the origin and I let the user to rotate the camera around the origin. I used the function `eye()` to compute (dynamically) the position of the camera, in polar coordinates and the `lookAt()` function for set the correct `ModelviewMatrix`. In particular, I used `var phi = [15,15];` to manage the angle between the Y and the (X,Z) plane and the (Z,X) Angle.

I implemented two different methods to rotate the camera:

1. Using 4 different buttons for the four main rotation directions (UP, DOWN, LEFT AND RIGHT).
2. Using the mouse, by clicking and holding the left button and moving the cursor, with a consequent movement of the camera.

The following code show hoe I implemented them in the script:

```
//  Camera Rotarion Parameters:
var flagCam = false;            //  Flag to control the camera
motion.
var camPos = [0,0];             //  State variable to animate the
camera with the mouse.
var dPhi = [0,0];               //  Camera angles increments.

function initInteractions() {
    document.getElementById("Animation").onclick = function()
{animation = !animation;};
    document.getElementById("buttonL").onmouseup =  function() {
flagCam = false;};
    // All the button have nearly the same code, so only one button
is  shown.
```

```
    document.getElementById("buttonL").onmousedown = function() {
        flagCam = true;
        dPhi = [0,-1];
    }
    canvas.addEventListener("mousedown", function(event) {
        var x = 2*event.clientX/canvas.width-1;
        var y = 2*(canvas.height-event.clientY)/canvas.height-1;
        flagCam = true;
        camPos = [x,y];
    });
    canvas.addEventListener("mouseup", function(event){flagCam =
false;});
    canvas.addEventListener("mousemove", function(event){
        var x = 2*event.clientX/canvas.width-1;
        var y = 2*(canvas.height-event.clientY)/canvas.height-1;
        if (flagCam) {
            dPhi[1] = 45*(x - camPos[0]);
            dPhi[0] = 45*(y - camPos[1]);
            camPos = [x,y];
        }
    });
}
//  Meanwhile, in render()..
//  Update Camera Position..
    if (flagCam){
        phi[0] += dPhi[0];
        phi[1] += dPhi[1];
    }
    modelViewMatrix = lookAt(eye(),at,up);
```

Keyframes Animation

For this point of the Homework, I tried to implement **keyframes based animation**.

I initialized a `keyframes[]` array and put inside a set of different positions of the center of the sheep body during the animation. The distance between two consecutive keyframes correspond to an **animated step** of the sheep. In the code, the animated step is computed through **interpolation** of different intermediate **frames** . The number of frames for each animated step is defined by the variable `nFrames`.

In the script, when the animation of a step (or of the jump) start, a function `delta()` is called to set the different increments `dx, dy, da` , the first two used to implement **translation** (through interpolation), the last one for the interpolation of the angles animation.

```
function delta(nFrames) {
 // Updates the step parameters (dx,dy,da)..
    dx = (keyFrames[currentFrameID+1][0] -
keyFrames[currentFrameID][0])/nFrames;
    dy = (keyFrames[currentFrameID+1][1] -
keyFrames[currentFrameID][1])/nFrames;
    da = (theta[LEG_LAU] < alfa/2) ? +alfa/nFrames: -alfa/nFrames;
}
```

In the `render()` function, if the animation is allowed (by using a button in the browser page), a function `step()` is called to perform the **per-frame animation**:

```
function step(){
//  Translate the Sheep Root Vertex..
    roots[0][0] += dx;
    roots[0][1] += dy;

//  Rotate Nodes..
    theta[BODY] -= da*0.1;
    for(var leg = LEG_LAU; leg<LEG_RPD; leg++) {
        theta[leg] += da;
    }
//  Update Nodes transformation matrices..
    for(var i=0; i<nNodes; i++) {
        transformNode(i);
    }
 // Check for the nex keyFrame..
    if (roots[0][0]>keyFrames[currentFrameID+1][0]){
        currentFrameID = (currentFrameID < nKeys-2) ?
 currentFrameID+1 : 0;
        roots[0][0] = keyFrames[currentFrameID][0];
        roots[0][1] = keyFrames[currentFrameID][1];
        delta(nFrames);
    }
}
```