# Homework 2

Pignata Giovanni 1913547

## 1. The Hierarchical Model

In order to create the **Kangaroo** model, I used a **left-child, right-sibling tree structure**. The Scene is composed of 20 nodes, 16 for the Kangaroo and the others for the land and the hill. 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)**: in particular, with this function the cube corresponding to the node is rotated along the rotation axis and around the respective joint. The node function **render()** is the function used to draw a node (a cube). In this function, some **uniform parameters** (as the *modelview* matrix and the colored texture) are updated for the specific node before calling **gl.drawArrays()**, that draw the final (scaled) cube. The drawing of the entire tree structure is possible due to the function **traverse()** called in the script **render()** function:

```
       function traverse(Id) {
// Traverse the tree structure.
          if(Id == null) return;
          stack.push(modelViewMatrix);
          modelViewMatrix = mult(modelViewMatrix, figure[Id].transform);
          figure[Id].render();
          if(figure[Id].child != null) traverse(figure[Id].child);
          modelViewMatrix = stack.pop();
          if(figure[Id].sibling != null) traverse(figure[Id].sibling); }
```

## Textures

To generate the **bump texture**, I used the following code:

```javascript
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 rough texture to the whole kangaroo body (face excluded) and for the land and the hill, but in this last case with a different **texSize**.

For the face, I loaded a very simple texture '**face.png**' and configured in a similar way as in the first homework. In the shaders, I controlled which style of texture (bump or simple colored) to use with the uniform bool **uBump** variable.

## Hop Animation

I divided the animation in two main part: the model translation around the hill, and the hop animation. Both the implementation are in the same function **hop()**, called in the render function if the animation in enabled by a button.

For the first part of the animation, I wrote the following code:

```
//  Movement Around the Hill:
      var t = 0;
      const T = 1000.0          // Number of frames for a single round.
      const f = t/T
      const wt = 2 * Math.PI * f

      // Compute the center of the XZ coord of the body.
      roots[BODY][0] = 4*Math.sin(wt - Math.PI/2)
      roots[BODY][2] = 4*Math.cos(wt - Math.PI/2)
      // Compute the Y angle of the body
      theta[BODY] = - 360 * f + 90
      if(f>=1){
        animation = false;
        t = 0;
      }
      else {
        t++;
      }
```

I implemented a **keyframes based animation** for the "hop": In particular, I divided the animation in 5 main frames, repeated until the animation stop.

For each keyFrame there is associated a time in which it is reached by the animation (starting from the beginning) and a particular keyValue for each node which represents the position (generally rotation angle or height for the body) to be assumed.

Starting the animation, the script calculates the increments to be added to the position of each node so that they can reach the value of the first keyframe in the fixed time through the **deltas**() function. Then, each time the render function is called, all the transformation matrices are updated with the respective increments. Once a keyframe is reached, deltas() is called again to recalculate the increments for the transition to the next keyframe.

## 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. 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);
```