Interactive Graphics,
Final Project:
"Almost Pacman"
a.y. 2018/2019

Dario Benvenuti
1562938

January 14, 2019

Abstract

"Almost Pacman" is a simple project, built to show how some of the most important features in the interactive graphics' field can be handled with WebGl. The goal of the project was to build something similar to a simple video game, in which the user can move its character in a maze, trying to collect objectives. The final result is much more like a prototype than a fully developed video game, but in its simplicity it accurately shows the challenges that developers would encounter if graphic engines would not have existed. Everything was built in native Javascript and HTML, with the aid of WebGl only, from scene organization, to collisions detection, movement, animations and cameras. In its roughness, the project can easily be improved, adding real 3d models, shadows, various levels and game modes, in order to make it a real video game.

1 Introduction

In this paper, we are going to analyze, step by step, the main challenges that was faced in the development of the project, and the solutions taken.
Those can be categorized as:

- Designing the maze and how the users can move the character in it.
- Building the hierarchical structure.
- Moving and animating objects.
- Handling collisions.
- Texturing the maze.
- Lighting the scene.
- Implementing two different viewing modes.

2 Design

Designing the project wasn't really a challenge, but, considering that the main goal was to build something very close to a video game, while coding everything with just Javascript and WebGl, clearly it needed to be designed to be as simple as possible.

- The design choice that had the biggest impact on the project, was to limit the user's freedom while moving the character. The whole scene is designed as a virtual grid, in which the character can move only from one tile to an adjacent one. This choice led to an incredible decrease in the complexity of the project, especially in animating the model and handling collisions with the maze's walls.

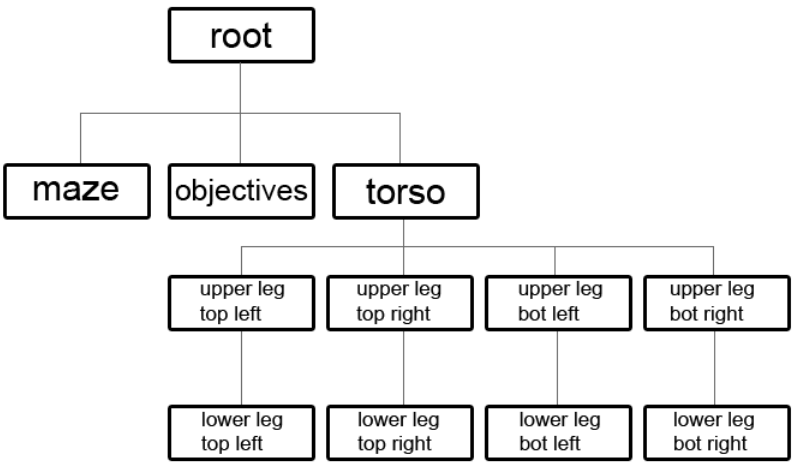
- Choosing a 3d model for our character that could be easy to build just through code, while not being as simple as the stylized human, was not so easy. At the end, considering that the project was born with a top-down view, choosing a stylized spider as our character, seemed a good option. The model was quite easy to build through code, and it shows the animations tailored for it even in the top-down view.
  Even later, when a more common over-the-shoulder view was added to the project, the model performs really well.

- Having two viewing modes, one top down and one over the shoulder of the character, following it, allows the user to have a clear idea of how the maze is done, and of how to navigate it. Obviously this leads to a straight forward completion of this "game", but at the same time lets the user fully experience the project. In the future, this project could evolve into a quite enjoyable game just by deleting the top down view, or restricting the time in that view at the user's disposal.

- In the over the shoulder viewing mode, we are using a perspective projection. This decision was made because we are using really basic lighting and texturing models, so, with an orthogonal projection, the user would have had a really hard time trying to navigate the maze. In the top down view, instead, the orthogonal projection fits perfectly, giving to the user the classical vibe given by old 2D games like PacMan. Notice that, in that view, anyway, the scene is slightly rotated, to give to the user a little feeling of 3D.

3 The hierarchical structure

Considering that we are not using any scene management tool, building the hierarchical structure was a core part of this project.
We have various objects in the scene that needs to be able to be transformed independently, but that are all at the same level. So, as the root of our tree, we will keep an empty node, representing the scene. The maze, the objectives and the spider will all be children of this empty node.
The spider can be considered as a sub tree, with the torso as its root node. Its 4 legs will be divided into upper parts and lower parts. All 4 upper parts are torso's children, and each lower part is the children of the corresponding upper one.



The objectives will be all inside one node, just because we don't care if each objective doesn't transform independently. They will just rotate around their center, for the whole time, so it's definitely not a problem. To handle their differences in position, instead of giving them a relative transformation matrix wrt to the scene, we will directly draw them from hard coded vertices.
In order to code this structure, we used the classic "left child, right sibling" linked list, in which from one node we can reach its first children, and then, from that children, we can reach all the others, that can be considered as siblings of the first one.

```
function createNode(transform, render, sibling, child){
    var node = {
    transform: transform,
    render: render,
```

```
    sibling: sibling,
    child: child,
    }
    return node;
}
```

In order to traverse this structure, we used the classic function based on a matrices stack. This model combines the relative matrix of each node with the transformation matrix of its parent, to get the right transformation matrix for each node.

```
function traverse(Id) {
  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);
}
```

It is important to point out that some components of our structure, like the objectives and each one of the spider's components, need to be able to rotate around their center.
To accomplish this, we simply apply two translations in the "initNodes" function: one before applying to the component's relative matrix the required rotation, and one after. Our goal is to have the center of the component that we need to rotate at the origin, before applying the rotation. Since rotations are always computed around the origin, in this way, we are rotating the component around it's center. Next, we translate everything back to it's original position.
Since WebGL applies matrices in the inverse order wrt the one that we specify in our code, we will first translate everything by componentCenter, then apply our rotation, and finally translate it by componentCenter*(-1).
We can see it in the code:

```
function initNodes(Id) {

    var m = mat4();

    switch(Id) {

    case sceneId:
      //rotations
      // x
      m = mult( m, rotate(nodesAngle[sceneId],  1, 0, 0 ) );
      // y
      m = mult( m, rotate(nodesAngle[sceneId+1],  0, 0, 1 ) );
      // z
      m = mult( m, rotate(nodesAngle[sceneId+2],  0, 1, 0 ) );
      figure[sceneId] = createNode( m, scene, null, mazeId );
      break;
    .
    .
    .
    case botUpSLegId:
      // rotation around its center
      // first, translate it by +center
      m = mult ( m, translate( uSLegsTransX-(uLegWidth/2) , torsoTransY+(uLegHeight
          /2), uBLegsTransZ ) );
      // next, apply the rotation
      m = mult ( m, rotate( nodesAngle[botUpSLegId], 1, 0, 0 ) );
      m = mult ( m, rotate( nodesAngle[botUpSLegId+1], 0, 0, 1 ) );
      m = mult ( m, rotate( nodesAngle[botUpSLegId+2], 0, 1, 0 ) );
      // finally, translate it back by -center
      m = mult ( m, translate( -1*(uSLegsTransX-(uLegWidth/2)) , -1*(torsoTransY+(
          uLegHeight/2)), -1*uBLegsTransZ ) );
      figure[botUpSLegId] = createNode ( m, botUpSLeg, botUpRLegId, botLowSLegId );
      break;
    .
    .
    .
}
```

4 Moving and animating the elements

To animate objects in our scene, we will call an update function, inside the render function, if animations are enabled. This will lead to frame rate dependent animations, meaning that we update the transformation matrix at each frame, so objects will move faster if we open our project on a machine with a monitor that can run with higher frames per second. Since 60 frames per second is still considered the standard between machines, we can do this without worrying too much of great distortions in our animations.
Our code will look like this:

```
function render()
{
    .
    .
    .
  if ( animationFlag )
      update();
  .
  .
  .
    requestAnimFrame( render );
}

function update() {

    // -------------------------------------------------------- animate objectives

    nodesAngle[objective1Id]  += 0.5;
      nodesAngle[objective1Id+1]  += 0.5;
      nodesAngle[objective1Id+2]  += 0.5;
        .
        .
        .
      // --------------------------------------- avoid overflow for objectives angles

    if ( nodesAngle[objective1Id]   >= 360 )    nodesAngle[objective1Id]  = 0;
        .
        .
        .

    // -------------------------- ROTATE THE CHARACTER ------------------------

    if ( characterRotation == 1){
      rotateCharacter();
      rotationCounter++;
      if ( rotationCounter == 20 ){
        rotationCounter = 0;
        characterRotation = 0;
        // ---------------------------- update direction
        if ( rotationDirection == 0 ){
          // -------------------------  we rotated left
          direction--;
          // ------ check correctness of the new direction
          if ( direction < 0 )
            direction = 3;
        }
        else{
          // -------------------------- we rotated right
          direction++;
          // ----- check correctness of the new direction
          if ( direction > 3 )
            direction = 0;
        }
      }
    }

    // -------------------------- MOVE THE CHARACTER --------------------------

    if ( characterMovement == 1 ){

```

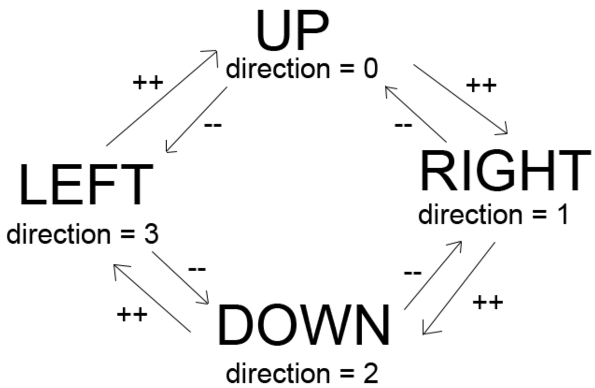
```
   // ---------- if there will be no collision in the current movement direction, move the character

   if ( checkCollision(torsoX, torsoZ, direction) == 0 ){
    if (cameraMode > 0)
     topDownMove(direction);
    else
     firstPersonMove(direction);
   }

   // ---------------------------- update the movement frame counter and the maze texture

   movementCounter++;
   updateTexture();

   // ------------------------------ after 20 movement frames reset movement parameters

   if ( movementCounter == 20 ){
    characterMovement = 0;
    movementCounter = 0;
    torsoX = Math.round(torsoX);
    torsoZ = Math.round(torsoZ);

    // -----------------------------------------------------  check if the user has collected an objective

    if ( torsoX == 4 && torsoZ == -5 && objective1Captured == 0 ){
     objective1Captured = 1;
     objectivesCaptured++;
     // ----------------------- update the text in the html
     document.getElementById("objectivesDiv").innerHTML = "Objectives collected: " + objectivesCaptured;
     document.getElementById("objectivesDiv").style.fontSize = "20px";
    }
        .
        .
        .
   }
  }

  // ---------------------------- update hierarchical structure components involved in the animation

  initNodes(torsoId);
    .
    .
    .
}
```

From the code we can notice these things:

- To animate objectives we are just increasing their rotation angles in the "nodesAngle" structure, that holds the rotation angles of all the components of our hierarchical structure, to make them constantly spin around their center. After increasing those values, we check if we have gone over 360, just to avoid overflow.
- In order to animate the spider we have divided its movement in 20 frames. Those 20 frames will be divided again in 2 sections, each one of 10 frames. So, in the first half of the movement, we move 2 opposite legs of the spider forward and up, while the other 2 are moved backward, to simulate the fact that in real life those would not move at all, being the movement pin. In the second half of the movement, we just move each leg in the opposite way, to make everything return to its original position.
- We actually have 3 different functions to animate the spider, "rotateCharacter", "topDownMove" and "firstPersonMove". The one that will be called between the 3 depends from the camera mode that the user is using. If the user is in the top down camera mode, the spider can be moved up, down, left or right, pressing the arrow keys on the keyboard. So, in this case, "topDownMove" will be called, with the right direction to move the character. If the

user is in the over the shoulder camera mode, the spider can be moved only forward, and rotated clockwise or anticlockwise. In this case, if the user moves the spider forward, pressing the up arrow key on the keyboard, "firstPersonMove" will be called. Instead, if the user presses the left or right arrow key on the keyboard, "rotateCharacter" will be called.
The direction in which we have to move the spider while in top down view, and the one in which we have to rotate it while in over the shoulder view, are just integers set by handling the "onclick" events and passed to "rotateCharacter" and "firstPersonMove".

- The direction in which we have to move the spider while in over the shoulder view is a bit more tricky to handle, because even if the spider can be moved only forward, it doesn't mean that we move always in the same direction. So we will keep track of the direction that corresponds to the right "forward" in our grid, using this scheme:



  When a rotation key is pressed, we just increase or decrease the direction value, and the we check if we have gone lower than 0 or greater than 3.

- To animate the spider we need to take care of the rotations of each of its components, and of its translation:
  To rotate the involved components, we just increase or decrease their rotation angles in the "nodesAngle" structure, as we did with objectives.
  To actually move the spider, we just increase or decrease it's X and Z translation values, so that, after updating all the nodes and traversing again the hierarchical structure, it will be drawn in the right position of the maze.

- Those values are stored in two variables, "charX" and "charZ", that keeps track of the current top right vertex of the tile in which the spider is.

- The clockwise rotation animation is the same as the animation to move right in the top down view.

- The anticlockwise rotation animation is the same as the animation to move left in the top down view.

5 Collisions

From the animation's code that we saw before, we can notice that actually we are checking if there will be a collision before moving the spider. So, we can say that, instead of handling collisions, we are actually preventing them to happen at all.
In order to do this, we just need 2 things:

- A data structure in which we hold the information needed to know if a given tile is part of a wall.
  For each tile with vertices (x1,z1),(x1,z2),(x2,z1),(x2,z2) that is part of a wall, we will have an entry in our structure like this: (x1, x2, z1, z2).

- A function to check if the tile in which we are going to move the spider is part of a wall.

```
function isAWall( x, z ) {

  var i = 0;
  var result = 0;

  for ( i; i<walls.length; i+=4 ){
    if ( walls[i] <= x && x <= walls[i+1] && walls[i+2] <= z && z <= walls
        [i+3] ){
      result = 1;
      break;
    }
  }

  return result;

}
```

So, before moving the spider, we will simply call the "checkCollision" function, that will just check if applying the translation that is needed to perform the movement in the current direction will lead in a tile that is part of a wall:

```
function checkCollision( charX, charZ, direction){
  // function used to check for a collision before moving the character.
  // returns 1 if there will be the collision, 0 if there will be not
  if ( direction == 0 ){
    // -------------------------------------- moving up
    if ( charZ > -5 && isAWall( charX+0.1, charZ ) == 0 )
      return 0;
    else
      return 1;
  }
  else if ( direction == 1 ){
    // ---------------------------------------- moving right
    if ( charX > -5   && isAWall( charX, charZ+0.1 ) == 0 )
      return 0;
    else
      return 1;
  }
  else if ( direction == 2 ){
    // ------------------------------------------ moving down
    if ( charZ < 4  && isAWall( charX+0.1, charZ+1 ) == 0 )
      return 0;
    else
      return 1;
  }
  else if ( direction == 3 ){
    // ------------------------------------------- moving left
    if ( charX < 4  && isAWall( charX+1, charZ+0.1 ) == 0 )
      return 0;
    else
      return 1;
  }
  else{
    // --------------------------------------------- wrong direction
    return -1;
  }
}
```

6 Texturing

The texture used in this project is a procedural color texture, applied only to the maze. We will keep track, at each frame, of the distances between the spider and the 4 objectives; each objective has a different color, so, we will add to the texture an amount, proportional to the distance between the spider and it, of its color.
For example, the objective 1 is red, so, the nearest the spider is to it, the more red the maze will be.
In order to implement this, we need to:

- Obviously, keep track of all distances.
  This can be easily done, taking advantage of the fact that we know in advance the top right vertex of tiles with an objectives in it, so, we will just subtract those X and Z values, to the X and Z values of the spider.

- Determine how to transform the distance from the spider to a certain objective to the relative amount of color, in RGB components.
  This could be a bit more tricky, since we have a yellow objective. This leads to the R and G components to be produced by two different sources. In order to avoid interference, we will convert distances to an exponential function, and then use a proportion to end up with a value between 0 and 255. This is done because, in our scene, the maze has a width of 10 units, so distances would fall in an interval too little to be directly converted to RGB components avoiding interference between objectives.

```
    function mapDistance( positionX, positionZ, objective ){
  var distanceX = 0;
  var distanceZ = 0;

  var distance = 0;
  var maxDistance = 1;

  if ( objective == 1 ){
    distanceX      = positionX - objective1X;
    if (distanceX < 0 )
      distanceX    = distanceX * (-1);
    distanceZ      = positionZ - objective1Z;
    if (distanceZ < 0 )
      distanceZ    = distanceZ * (-1);
    distance     = distanceX + distanceZ;
    maxDistance    = maxDistance1;
  }
  else if ( objective == 2 ){
    .
    .
    .
  }
  .
  .
  .
  distance = Math.pow(distance, 5);
  maxDistance = Math.pow(maxDistance, 5);
  return 255*distance/maxDistance;
}
```

- Update the texture after every movement of the spider:

```
    function updateTexture(){
  // compute rgb components from distance
  red    = 100 + mapDistance( torsoX, torsoZ, 1) + mapDistance( torsoX, torsoZ,
      2);
  green    = 100 + mapDistance( torsoX, torsoZ, 3) + mapDistance( torsoX,
      torsoZ, 2);
  blue   = 100 + mapDistance( torsoX, torsoZ, 4);
  // avoid to have components greater than 255 ( since R and G components have
      2 sources )
  if ( green   > 255 ) green   = 255;
  if ( blue    > 255 ) blue   = 255;
  if ( red   > 255 ) red    = 255;
  // compute the texture
```

```
  computeTexture();
}
```

- Finally, compute the texture.

```
    function computeTexture(){
  for(  var   i = 0;   i < texSize ;   i++ ){
    for(  var   j = 0;   j <texSize ;   j++ ){
      image[4*i*texSize+4*j ]   = red;
      image[4*i*texSize+4*j+1] = green;
      image[4*i*texSize+4*j+2] = blue;
      image[4*i*texSize+4*j+3] = 255;
    }
  }
  configureTexture(image);
}
```

In order to apply the texture only to the maze, we will use a flag that will be sent to the shader. There, if the flag is 1, it means that we are drawing the maze, and so we will multiply the fragment color by the corresponding texel color.
Since we are using a color texture, mapping from texture coordinates to object coordinates is trivial.

7 Lighting

Nothing special is done with respect to lightning in this project.
We have a light source, standing above the spider, that will follow it in its movements.
In order to compute the light, we compute the normal for each triangle that we draw and send it to the vertex shader. There, we will do the classic computation:

```
void main()
{
  // ----------------------- LIGHT COMPUTATION -----------------------
  pos = -(modelViewMatrix*vPosition).xyz;
  light = lightPosition.xyz;
  L = normalize(light-pos);
  E = normalize(-pos);
  NN = vec4(vNormal,0);
  H = normalize(L+E);
  N = normalize((modelViewMatrix*NN).xyz);
  // -----------------------------------------------------------------
  fColor = vColor;
  fUsingTexture = usingTexture;
  fTexCoord = vTexCoord;
  fPosition = vPosition;
  gl_Position = projectionMatrix * modelViewMatrix * vPosition;
```

To have a smoother lightning, we will let the fragment shader interpolate this result accross vertices:

```
void
main()
{
  // ----------------------- LIGHT COMPUTATION -----------------------
  vec4 fColorPhong;
  vec4 ambient = ambientProduct;
  float Kd = max(dot(L,N), 0.0);
  vec4 diffuse = Kd*diffuseProduct;
  float Ks = pow(max(dot(N,H), 0.0), shininess);
  vec4 specular = Ks*specularProduct;
  if (dot(L,N)<0.0){
    specular = vec4(0.0, 0.0, 0.0, 1.0);
  }
  fColorPhong = ambient+diffuse+specular;
  fColorPhong.a = 1.0;
  if ( fUsingTexture == 0.0 )
    gl_FragColor = fColor*fColorPhong;
  else
    gl_FragColor = fColor*fColorPhong*texture2D(Tex0, fTexCoord);
}
```

One last important thing to point out about the lighting, is that, switching between the 2 viewing modes, we are rotating the maze of 90 degrees; but the model view matrix doesn't apply to the light position. This means that while using the top down view, the light needs to move on the X and Y axis, while in the other viewing mode it needs to move on the X and Z axis.
So, in order to keep the light on top of the spider, we need to switch the light position Y and Z values when switching viewing mode, and, while handling the spider's movements, take care about moving the light position along the right axis.

8 Viewing

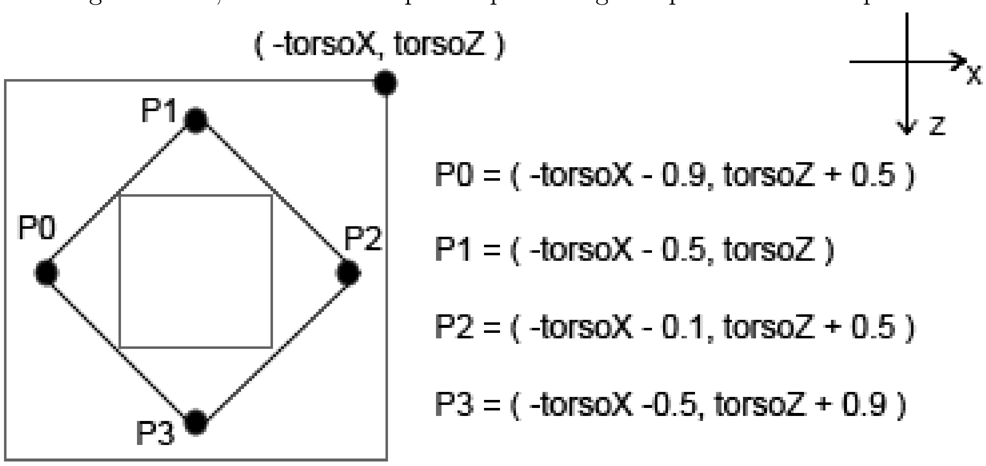
Implementing the top down view was trivial. We simply rotate the scene to make the align with the Y axis.
In order to let the user see the whole maze, we will use a projection matrix to extend the field of view. Since the maze goes from -5 to 5 on the X and on the Y, we will increase the field of view to the interval [-6, 6] on those axis by calling:

```
modelViewMatrix    = mat4();
projectionMatrix   = ortho(-6, 6, -6, 6, 15, -15);
```

To implement the second view mode, things get a bit more complex, because we need to build a moving camera that follows the spider in its rotations and translations, while keeping it at the center of the view.
To achieve this, we will use the "lookAt" function, specifying a camera position and target vectors.

```
var cameraEye       =   [(-torsoX-0.9)-factorZ,1,-(torsoZ+0.5)+factorX];
var cameraTarget    =   [-torsoX-0.5,0.1,-(torsoZ+0.5)];
var cameraUp        =   [0,1,0];
//-----------------------------------------------------------------------------
modelViewMatrix     =   lookAt(cameraEye, cameraTarget, cameraUp);
//-----------------------------------------------------------------------------
projectionMatrix    =   perspective(110, -1, 0.1, -20);
```

As we can see from the code, "cameraEye" and "cameraTarget" will change X and Z values as the spider moves or rotates. To follow it in its translations, we will directly use "torsoX" and "torsoZ", that, as we saw before, contains the X and Z values of the top right vertex of the tile in which the spider is. To those values we will subtract or add a constant value, and a varying value, "factorX" and "factorZ", to make the eye vector translate correctly when the spider rotates.
The camera position vector will follow this scheme, in which we have the outer square representing a tile of the maze, with edge of 1 unit, and the inner square representing the spider's torso component:



In each rotation, the position vector's X and Z values will gradually go from their original values to the one of the point in which it will land after the rotation. So, for example, if we are in P0, as we are when the user choose to switch viewing mode, and we rotate clockwise, the camera position will translate along a line from P0 to P1, meaning that after the rotation the X will be 0.4 greater and the Z will be 0.4 less, meaning that in each one of the 20 rotation frames the X will increase by 0.02 and the Z will decrease by 0.02.

In order to implement this, we will use, as said, 2 variables, namely "factorX" and "factorZ", that will start equal to 0. During rotations, we will increment or decrease those variables by 0.02, depending on the direction of the rotation.
The camera target vector will remain at the center of the spider, so it will change only after translations, thanks to the "torsoX" and "torsoZ" values in it.
From this scheme we must point out some things:

- We need to use "(-1)*torsoX" because the coordinate system that we used to define vertices of our objects and the one used to render the scene have the X axis with opposite directions.
- In theory, it would be better to make the eye vector move along a circumference, but that would add a lot of complexity without adding much in terms of quality of the camera movement itself. Even moving the camera along a line doesn't lead us to clipping, so we can be fine with that.
- As we can see from the code, we are using "factorX" to modify the Z value of the eye vector, and "factorZ" to modify the X value. This is required because through the first call to the "perspective" function, we start at P0, so we will have the X and Z axis inverted.

9 Switching between viewing mode

Since in the 2 viewing modes we have different types of movement for our spider, we need to take care of something when the user switches viewing mode.
In particular, we need to think that in the top down view, the spider can't rotate, while in the second view it can. The user is able to switch between viewing mode whenever he or she wants, if the spider is not moving or rotating, so it means that he or she can go to the following camera mode, rotate the spider once, and go back to the top down view. If we don't do nothing in regard, we will end up with a rotated spider in the top down view, something that will break all our animations.
To avoid this we will simply make sure to reset everything related to components rotation, directions and camera eye vector's factorX and factorZ.
So, we will end up with this:

```
// render function
function render(){
    .
    .
    .
    // if the camera is in first person
  if ( cameraMode < 0 ){
    if ( changedCamera == 1 ) {
      // ------------ remove the X rotation from the scene that is useful only in top down view
      nodesAngle[sceneId] = 0;
      initNodes(sceneId);
      // adjust lightPosition
      var tempLight = lightPosition[3];
      lightPosition[3] = -lightPosition[2];
      lightPosition[2] = lightPosition[3];
      gl.uniform4fv (gl.getUniformLocation(program,"lightPosition"),  flatten(lightPosition));
      // reset camera factors
      factorX = 0;
      factorZ = 0;
      // reset camera position
      cameraPosition = 0;
      // reset direction
      direction = 1;
      changedCamera = 0;
    }
    // ----------------------------------------------------------------------------

    var cameraEye =   [(-torsoX-0.9)-factorZ,1,-(torsoZ+0.5)+factorX];
    var cameraTarget =  [-torsoX-0.5,0.1,-(torsoZ+0.5)];
    var cameraUp =    [0,1,0];
```

```
    // ---------------------------------------------------------------------------
    modelViewMatrix = lookAt(cameraEye, cameraTarget, cameraUp);
    // ---------------------------------------------------------------------------
    projectionMatrix  = perspective(110, -1, 0.1, -20);
  }
  else {
    if ( changedCamera == 1 ) {
      // add the X rotation from the scene that is useful only in top down view
      nodesAngle[sceneId] = -75;
      initNodes(sceneId);
      // adjust lightposition
      var tempLight = lightPosition[3];
      lightPosition[3] = -lightPosition[2];
      lightPosition[2] = lightPosition[3];
      gl.uniform4fv (gl.getUniformLocation(program,"lightPosition"),  flatten(lightPosition));
      changedCamera = 0;
    }
    // reset rotations from first person mode
    nodesAngle[torsoId+1] = 0;
    initNodes(torsoId);
    modelViewMatrix   = mat4();
    projectionMatrix  = ortho(-6, 6, -6, 6, 15, -15);
  }
  .
  .
  .
}
```

10 Problems that could have been fixed differently

The final version of this project has one problem that could have been handled in a better way. While defining the maze's vertices we have used a coordinate system with an inverse X and Z axis wrt to the one used by WebGL.
Even if this does not lead to any decrease in performance, since it can be easily fixed with a simple rotation of the scene, there are circumstances in which it will lead to a tremendous decrease in the readability of the code.
For example, when we need to rotate objects around their center, it can be very hard to tell the signs of the X and Z values needed to compute the right translations to apply before and after the rotation; when we use the perspective view, we need to think that we are actually dividing the X and Y values of the position of each point of our scene by its Z component, and, if there are problems with coordinate systems, we could get a mirrored image, because of signs.
The longer this problem is just corrected using signs and not solved at its root, changing the original coordinate system used to define vertices, the more difficult it will be to fix it, because there will be a lot of signs to change through all the code.
That is exactly what happened with this project.
Of course, in a future version, this could be fixed by carefully going backward and change everything to make coordinate systems match.