Homework2

Ketbjano Vocaj
1652134

June 2020

1 Introduction

In this report I will describe how I managed to implement the tasks assigned in the second homework, starting from the creation of the objects in the environment, then specifying the relation between them, then applying different textures for each independent structure and finally animating the crucial parts of the bear's body (or the so called **"joints"** of the body).

2 Objects allocation and hierarchies

All the objects were created by following a simple geometry structure based only on cubes, it means that the main figure is a cube which is istantiated every time we want to draw some specific element on the canvas. In fact we have only one call of the function `cube()` in the main program, but each time we call the render function of a certain object it draws a cube with its own peculiarities.
Since we have to draw and animate a Grizzly bear[1] we need a structure that lets us to create dependencies between the parts of the body, for example if the entire body moves then the legs, arms, head etc. moves too depending on the movement of the torso, or also if the upper part of a limb rotates then the respective lower part follows its rotation; we can abstract this procedure with a **tree based structure** in which the nodes represent a part of the body and the edges represent the transformation matrices that will be applied to the children.
In the node structure we have:

- the transformation matrix
- pointer to the function that draws the cube
- pointer to the the next sibling
- pointer to the first child

Initially we create empty nodes in the function `createNode()` and store them in the array `figure[]` with a for-loop, then we initialize them in the function `initNodes(Id)`.
This function takes as parameter the **Id** concerning the body section we are considering and for each of them we define the transformation matrix (that will be applied to the cube's istantiation) and the link between the sibling and children.
As we know, the body parts are connected to a central element which is the torso and all their movements or rotations depend on it, so we choose it as the root of our tree from which begins all the *hierarchical* structure of the bear: it starts with the **torso**, then proceeds to the first child which is the **head**, then in the same "level" we have all the **upper** parts of the limbs and in the end we have the **tail**; instead in the next level we have the **lower** part of the limbs which are the children of the upper parts of the body.
After the declaration of the body sections we define the **joint** points in which the rotations will take place, it means that the lower part of the limbs must rotate around a fixed point that connects both parts and also the upper limbs must rotate around a point that connects them to the torso; this is done in every nodes render function where we specify these rotation

[1] The tree is implemented with the same rule of drawing but without hierarchy specification

points before the effective drawing of the cube.
The last thing to do now is to make our bear concrete and for doing so we have a function called `traverse(Id)` which basically traverses all the children and siblings of a certain node, it is constructed as follows:

```
function traverse(Id) {

   if(Id == null) return;
   stack.push(modelViewMatrix); //Keeps in memory the modelView applied to the parent. Before rendering the sibling, we pop the matrix.
                                //This is done because sibling don't get influenced by the modelView which we apply to the children
   modelViewMatrix = mult(modelViewMatrix, figure[Id].transform); //Update the modelViewMatrix
   figure[Id].render();
   if(figure[Id].child != null) traverse(figure[Id].child); //If it has a child we call this function recursively
    modelViewMatrix = stack.pop(); // Restoring the previous value of the modelView
   if(figure[Id].sibling != null) traverse(figure[Id].sibling);
}
```

This function is called in the `render` function (so it's updated every time with `requestAnimationFrame(render)`) and since the torso is the root of the tree, we give it as parameter on the traverse function, this means that every children is bounded to the same ***modelViewMatrix*** of the father and each "grandchild" is bounded to both modelView Matrices of the father and the "grandpa". Note that the a child is **NOT** influenced by the modelViewMatrix of its sibling but only by the father's one, because we restore the previous value of the matrix before calling a sibling.

3 Texture mapping

The texture mapping was done using the same approach for the first homework, the only difference is that I had to manage multiple textures in this case.
There are 5 images that are used generally: one for the bear's fur, one for the bear's head, one for the tree's trunk, one for the leaves and the last one for the background. In the **.html** file we load all of them and they are passed afterwards to the **.js** file where the textures will be configured in the `configureTexture()` function; after allocating the 5 different textures the shader has to know which one is used for a specific sampler, in order to do so each texture is bound to its own texture unit through the function `gl.activeTexture(gl.TEXTUREI)` and then it's passed to the shader through `gl.getUniformLocation`.
After allocating and binding the texture units I managed to apply each texture to a certain object using some boolean controls in each node's render function i.e. every time we reach a certain part of the body the respective control is set to `true` and the controls of the remaining parts are set to `false` (uniforming them each time for the shader); therefore in the fragment shader we check which of these controls are true and apply the texture concerning that object.

4 Animation

The tricky part on implementing the animation was to define the values of rotation/translation and the complex interaction of movement between the bear's body parts.
First of all I created three arrays in which the transformations had to take place i.e. `bearPosArray[]` is the bear's positioning array (along x,y,z) which is affected by the translating transformation, `theta[]` is the array in where each body part is subordinate to the rotating transformation, `yValue[]` instead contains the positioning of the objects along the Y axis.
After checking the pressure of the button, the animation starts in the function `animation()` inside the render and here we can see that there are some `if...else` branches in order to check if the bear reaches some checkpoints; there are four important checkpoints:

1. Bear reaches a defined position near the tree along X axis by moving its paws.
2. Bear turns backwards rotating around Y axis.
3. Bear stands up rotating around Z axis.
4. Bear scratches itself back and forth along the tree for 6 times.

The first movement was made by increasing the X value of the bear and the angles of the paws, checking if their rotation does not exceed a certain degree (45° in this case) in order to simulate some real anatomical behaviour; note that for every modified parameter we should "update" the node by calling the function `initNodes(Id)`.
The second movement was made by increasing the `theta[torsoId]` angle of the torso and the `yValue[]` of the paws, so the bear can turn itself doing a 170° angle while moving its legs.
In the third movement the bear is rotated around the Z axis using `theta[torsoId2]` angle and keeps its leg flexed in order to prepare itself to scratch.
For the fourth movement, both the Y position of the torso and the upper legs angles are updated each time the animal reaches a certain height; the leg's rotation is done for underlining the fact that the bear is pushing itself upwards for scratching its back and for returning in the initial position.
When the bear completes all these movements it returns in the walking position and walks toward the starting point by decreasing the X value of the torso.

5 References

https://www.khronos.org
https://www.khronos.org/registry/webgl/specs/latest/2.0
http://interactivecomputergraphics.com/8E/