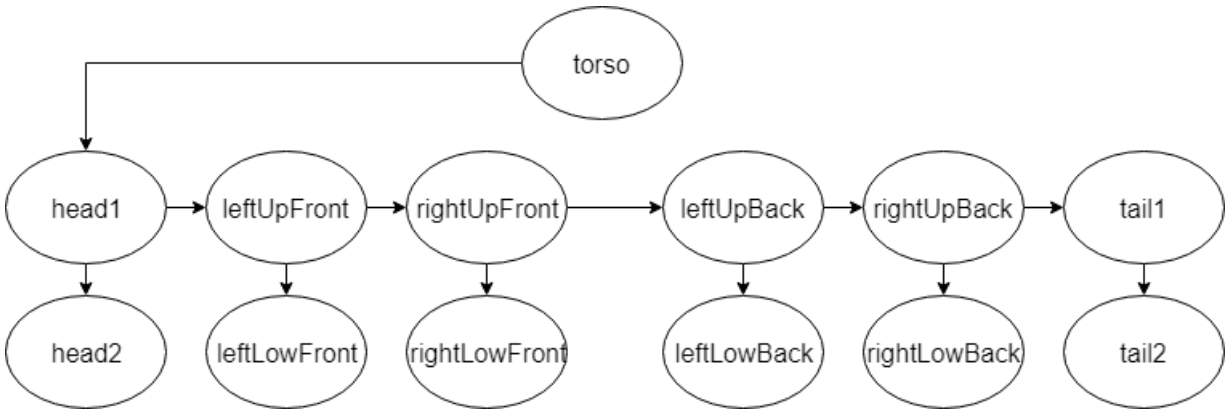
Interactive Graphics HomeWork#2 Documentation

by Herson John Nolasco Ruiz

June 2, 2019

1 Hierarchical model



In order to create the horse model, I implemented the **child-sibling structure** shown in the figure above to represent the hierarchy.
Each part of the horse is represented through a node in the structure and for each node is assigned an index.
In the initialization step of the nodes (the function "initNodes"), we have a switch-case statement controlled by the "Id" of the parts that compose the horse.
For each part of the model, basically we construct a matrix and we call the transformation functions "translate" and "rotate". In the translate function we place each object in their initial positions, and we apply the rotation function to the z axis by taking the theta angle value in the array **var theta = [0, -25, -10 , 0 , 0, 30, 20, -20, 25, -30, 75, -90, 90];**.
The "createNode" function used to create the node for each object, takes in input as a second parameter a render function. The render function is defined for every object.
Notice that the "traverse1" function starts from "torso" that is the root node, in fact we placed the components of the horse according to the torso position that is translated to the (x_horse, y_horse) position where x_horse = -15 and y_horse = 5 are fixed a priori.
To implement the head of the horse, I used two objects "head1" and "head2". The first object is directly connected to the torso in the scene and the second object is connected to the first part of the head and in the hierarchy is the child of "head1" node.
Also for the tail of the horse I used two objects "tail1" directly connected to the torso and "tail2" connected to the previous object.

2 Procedural texture

The "checkerboard pattern" is created through two variables "image1" and "image2" following the example of the textured cube. The linear decrease of intensity from the front to the back of the body of the "torso" is obtained through this part of code:

```
var image2 = new Uint8Array(4 * texSize * texSize);
for (var i = 0; i < texSize; i++) {
for (var j = 0; j < texSize; j++) {
image2[4 * i * texSize + 4 * j] = texSize + j;
image2[4 * i * texSize + 4 * j + 1] = texSize + j;
image2[4 * i * texSize + 4 * j + 2] = texSize + j;
image2[4 * i * texSize + 4 * j + 3] = 255;
}}
```

In order to apply the texture only to the "torso" part of the horse, I introduced two boolean variables "applyTexture" and "horse_part". The first variable is used in the index file to distinguish between the "torso" and the rest of the horse, more precisely in the fragment shader we have this if-else statement:

```
if(applyTexture){
gl_FragColor = fColor * (texture2D(Tex1, fTexCoord) * texture2D(Tex0, fTexCoord));}
else gl_FragColor = fColor;
```

The "horse_part" variable instead, is used to pass a different fColor to the horse to distinguish it from the colors of the obstacle that will be introduced later.

Notice that those boolean variables are rewritten in every render functions. The fColor for the horse is set to **fColor = vec4(0.5, 0.3, 0.1, 1.0)** that corresponds to the "brown" color.

3 The obstacle

The obstacle is created through another hierarchical model shown in this diagram:

The root node this time is the horizontalPole1 that is the upper one so the "traverse2" function defined for the obstacle starts from it.

The position of the obstacle is (-0.5, 6.1). To manage the color of the obstacle, I introduced a boolean variable "vertical_pole" to distinguish the horizontal and the vertical poles of the obstacle.

This variable is used in the index file, more precisely in the vertex shader:

```
else if (vertical_pole){fColor = vec4(0.2, 0.2, 1, 1);}
else fColor = vec4(0.5, 1.0, 0.5, 1);
```

For the vertical_pole I used "blue" color and for the horizontal poles I used "lime green" color.

4 The animation

Before implementing the animation I modified the "canvas" dimensions to 1200 x 1024, in order to make the scene bigger to contain the full animation. Also I introduced a camera, to change the point of view of the scene. The default values are theta = 60/180; phi = 70/180.
In the main index there is a "Start animation" button in the upper-left corner of the page to play the full animation.
The animation is implemented using the **"requestAnimationFrame"** function.
I divided the animation in three parts: before the obstacle, the jump of the obstacle and the final part.
I introduced some auxiliary functions used during the entire animation.
The first function **"drawHorse"** recalls the "initNodesHorse" and "traverse1" to redraw the horse at every frame.
The second function **"rotate_leg(id)"** manages the legs of the horse during the "before-obstacle" and "final" parts of the animation. More precisely uses an auxiliary array
var flag = [null, null, false, null , false, null, false, null, true, null, true, null, null];
where a boolean value "true" or "false" is assigned for every up-leg.
If the flag[legId] is "true" we mean that the leg should increase and move counterclockwise, if is "false" the leg moves clockwise.
So in the rotate_leg function we have a switch-case statement controlled by the id of every leg, in the case of flag equal to "true" we decrease the theta angle of the leg, if "false" we increase the theta angle. Notice that the leg can move in the range [-40, 40], so when it reaches these limits the flag is negated.
The first part of the animation is defined through **"start()"**, that while the x_horse is < -7.5 increases the x_horse and recalls the rotate_leg, traverse2, drawHorse functions.
For the jumping action I decided to create three functions.
The first is **"jump_up()"** that manages the horse while the x position is < -2. This function increases the x and y position of the horse, increases the theta angle of the torso and fixes the theta angles for the upper and lower legs. With this function we mean to simulate the "takeoff" action.
The second function is **"jump_flat()"** manages the horse while its x position is < 3. This function also increases the x position, the y position, decreases the theta angle of the torso and fixes the theta angles of the legs. With this function we mean to simulate the moment when the horse is above the obstacle.
The third function is **"jump_down()"** that manages the horse while its x position is < 7.5. It simulates the "landing" action. The function increases the x position, decreases the y position, decreases the theta angles of the torso and fixes the theta angles of the legs. If the control condition does not hold anymore, it fixes the flag values and theta angles of the upper legs for the last part of the animation.
The last part of the animation is managed by the **"end()"** function that continues to increase the x position of the horse and brings again the theta angles of the torso and the y position to the default one.