TYPO

Interactive Graphics Project

Leonardo Di Paolantonio 1654563

Contents

1 Introduction **2**

2 Dragon model **3**

3 Animations **6**

3.1 Walking 7

3.2 WalkToRun 8

3.3 Running 11

3.4 RunToFly 12

3.5 Flight 13

3.6 FlyToRun 13

4 Words **14**

5 Floor **18**

6 Lights **19**

7 Game management **20**

8 Rendering **22**

1 Introduction

The purpose of this project is to create a Typing game.
It is written using **ThreeJS** library, without any additional libraries or tools. There are no imported model.
The game is composed by two main part: the words appearing at the top and the dragon at the bottom.
The words that appear are created using *TextGeometry*, while the dragon is a hierarchical model of polygons.
Once the game is started, the words start to appear on the top of the screen and the user has to write them as fast as possible, trying not to make **typos** (typing error) possibly.
The progress of the game influences the animations of the dragon:
The dragon can **walk**, **run** and **fly**; the speed of the dragon increases going from walking to running to flying. Moreover the dragon can have an **happy** facial expression or an **angry** one.
At the start of the game the dragon is walking very slowly; every time the user insert a word correctly, the speed of the current state of the dragon increases (e.g. if the dragon is running, it starts to run faster). After a certain level of speed, the dragon change state (e.g. when max speed of running is reached, the dragon starts to fly).
On the contrary, every time the user type the wrong letter of a word, the spped of the current state of the dragon decreases and if such speed reach the lowest bound, the dragon change state (e.g. the dragon passes from flying state to running state).
Moreover at the start of the game the dragon is happy, if the user types the wrong letter the dragon become angry. It will return happy when the user type correctly a word.
From the point of view of the word on the screen, such word is **blue**. Every time the next letter to be typed is typed correctly, such letter turns **green**, otherwise turns **red** (until the user will type it correctly). In the lower right corner there are the **meters** done by the dragon.
Such meters basically reflects how many errors has been done by the user, because the dragon will be slower at each error.
This is useful since a player who wants to typed as fast as possible is more likely to commit typos.
The game lasts **30 seconds**, at the end of it the player receives the data:
Number of words typed, numbers of errors, numbers of words per second and the meters done by the dragon.

2 Dragon model

The dragon is a hierarchical model, the root of such model is the **Body**:

```
geometry = new THREE.CylinderGeometry( 23, 23, 50, 12 );
material = new THREE.MeshStandardMaterial( { color: 0xf4ac66} );
var body = new THREE.Mesh( geometry, material );
scene.add(body);
```

The body is a cylinder, with a Standard orange material. So once we create the mesh combining the geometry and the material, we can add it to our scene.
Now we focus on the **Head** which is a child of the Body:

```
geometry = new THREE.BoxGeometry( 100, 65, 50);
material = new THREE.MeshStandardMaterial( {  color: 0xfdb900 } );
var head = new THREE.Mesh( geometry, material );
head.position.x = 40;
head.position.y = 70;
body.add(head);
```

This is a rectangle with a slightly different orange color with respect to the Body. We need to adjust the position of the Head in order to give the appearance of a dragon.
Finally we can add the Head as a child of the Body.
The Head in turn has several children, the **Eye** for example:

```
var l = new THREE.TextureLoader();
var texture1 = l.load("file1.jpg");
var texture2 = l.load("file2.jpg");
geometry = new THREE.BoxGeometry( 30, 30, 4);
material = new THREE.MeshStandardMaterial( {map: texture1} );
var eye = new THREE.Mesh( geometry, material );
eye.position.z = head.geometry.parameters.depth/2 + eye.geometry.parameters.depth/2;
eye.position.x = -25;
eye.position.y = 10;
head.add(eye);
```

In this case we want to use a texture for the eye, in particular we load two different textures. This is because we want that the dragon passes from happy mood to angry mood during the game, so we will switch the textures during the game (we will see it in ”Word” section).
So we pass the texture **texture1** in the material, positionate the eye on the Head and add it as a child of the Head.

Another child of the Head is the **Mouth**:

```
material = new THREE.LineBasicMaterial( { color: 0x000000,linewidth: 3} );
geometry = new THREE.Geometry();
geometry.vertices.push(new THREE.Vector3( -45, 15, 0) );
geometry.vertices.push(new THREE.Vector3( -30, 0, 0) );
geometry.vertices.push(new THREE.Vector3( 0, 0, 0) );
geometry.vertices.push(new THREE.Vector3( 0, 0, -head.geometry.parameters.depth) );
var mouth1 = new THREE.Line( geometry, material );
...
```

In this case we need to create a line by pushing in the geometry the points in which such line must pass.
But this is the Mouth used when the dragon is happy, is a smiling mouth. So we need to create a second Mouth in which basically the only thing that change is the points in the geometry (we want a sad mouth). Thus there is also the code for the second mouth (**mouth2**), notably only the smiling mouth (**mouth1**) is added to the Head, then during the game the mouths are removed/added to the head according to the mood of the dragon.

Then the Head has other (decorative) children like the **nose**, that is simply a small black circle, and three **sphere** of different dimensions positionated on the top of the head.

Other children of the Body are the Paws. I call **Paw** the posterior paws:

```
geometry = new THREE.BoxGeometry( 40, 10, 25);
material = new THREE.MeshStandardMaterial( {  color: 0xffce44} );
pawR = new THREE.Mesh( geometry, material );
pawR.position.y =-40;
pawR.position.x = 0;
pawR.position.z =40;
body.add(pawR);

pawL = pawR.clone();
pawL.position.z = -40;
body.add(pawL);
```

I create the right paw, positionate it and add it to the body.
Then I simply clone the right paw to create the light paw, changing only the position of z.

I call **Hand** the anterior paws:

```
handR = pawR.clone();
handR.scale.x = 2/3;
handR.position.x =0;
handR.position.y = 0;
handR.position.z = 35;
handR.rotation.x = Math.PI / 2;
handR.rotation.y = Math.PI / 2;
body.add(handR);

handL = handR.clone();
handL.position.z = -35;
body.add(handL);
```

Again I clone the right paw to generate the right hand, but first of all I want that the hands have a smaller width with respect to the paws, so I scale the x. Then I also need to rotate it on x and y, in order to have the hands along the body let's say.

The last children of the Body are the **Wings**.
In particular, to simplify the management of the rotation of the wings, I have use auxiliar pivots:

```
geometry = new THREE.BoxGeometry( 5, 5, 5);
material = new THREE.MeshBasicMaterial( {  color: 0xffb347 } );
pivotR = new THREE.Mesh( geometry, material );
pivotR.position.x = -15;
pivotR.position.z = 8;
pivotR.rotation.z = -0.4;
body.add(pivotR);
```

This is the code for the pivot of the right wing. Such pivot is a very small cube that is positionated inside the body of the dragon, so it is not visible. The idea is to define the real wing as the child of the pivot, in this way the rotation of the pivot will make a realistic wing rotation:

```
geometry = new THREE.BoxGeometry( 30, 40, 7);
material = new THREE.MeshStandardMaterial( { color:colorWings} );
var wingR = new THREE.Mesh( geometry, material );
pivotR.add(wingR);
```

Moreover there is an additional part of the wing, that emerges only when the dragon starts to fly:

```
geometry = new THREE.BoxGeometry( 30, 30, 7);
material = new THREE.MeshStandardMaterial( {  color:colorWings} );
var wingR_2 = new THREE.Mesh( geometry, material );

wingR_2.geometry.vertices[0].x -= 12;
wingR_2.geometry.vertices[1].x -= 12;
wingR_2.geometry.vertices[4].x += 12;
wingR_2.geometry.vertices[5].x += 12;

wingR.add(wingR_2);
```

Such additional part is pointed, so we need to modify the coordinates of some vertices.

3 Animations

As we said we have three main animation for the dragon: walk, run and fly. But there are also animations for the transition from a state to another. First of all we define some variables:

```
var t =0;
var state = "";
var speedWalk = 1;
var speedRun = 1;
var speedFly = 1;
var needWalkToRun = false;
var needRunToFly = false;
var needFlyToRun = false;
```

We have **t** that is used for the periodic movements in the animations, **state** that indicates in which state the dragon is and so what animation must be performed (the values of state are "walk", "run", "fly" and "trans", that indicates that the dragon is performing a transition animation).
Then we have the **speeds** of the various states and the boolean variable which indicates the **need** to perform a transition animation.

3.1 Walking

Now we focus on the first animation, ***walk()***:

```
t += 0.08 * speedWalk;
t = t % (2* Math.PI);
```

This pattern is used in every animation, basically you use **t** to have the period from 0 to 2π. The increment of **t** depends on the value of the speed of the walking.
Let's see how to move the **paws** (posterior paws):

```
pawR.position.x =  Math.cos(t) * 15;
pawR.position.y = -40 - Math.sin(t) * 15;

pawL.position.x =  Math.cos(t + Math.PI) * 15;
pawL.position.y = -40 - Math.sin(t + Math.PI) * 15;

pawR.position.y = Math.max(-40,pawR.position.y);
pawL.position.y = Math.max(-40,pawL.position.y);

if (t>Math.PI){
    pawR.rotation.z = -Math.sin(2*t)*0.5;
    pawL.rotation.z = 0;
}
else{
    pawL.rotation.z = -Math.sin(2*t)*0.5;
    pawR.rotation.z = 0;
}
```

The first four lines of code generate a circular rotation of the paws.
So the x position of the right paw goes from 15 to 0 to -15 to 0 to 15 again, while the y position goes from -40 to -55 to -40 to -35 to -40 again.
Notably, since we want that the movements of the two paws are **alternated**, the movements of the left paw are out of phase of π.
But since the paws touch the ground at y= -40, what we do is to limit the lowest value of y to -40, using the **max** function from Math.
The last two condition are to make the walking more realistic:
When **t** is greater that π it means that the right paw is starting to **come off the ground** and we want that the paw make a rotation on z axis.
With such rotations first the paw is inclined on one side, then returns to the original position, then is inclined on the other side and finally return to the original position. This movement must be done within half of the period (that is the period in which the paw is off the ground) so we need to consider the sin of **2*t**.

The same approach is used for the left paw, the assignments to zero are for adjusting eventual misalignments.
Then we manage the **hands**:

```
handR.position.x = -Math.cos(t)*10;
handR.rotation.y = -Math.cos(t)*0.5;

handL.position.x = -Math.cos(t+ Math.PI)*10;
handL.rotation.y = -Math.cos(t+ Math.PI)*0.5;
```

In this case we want to create the typical movements of the arm when a person is walking, that is like a semicircular rotation.
To do this we have to make a periodic movement along the x axis with a periodic rotation on the y axis (this is because the hands are generated as paws and then are rotated of $\pi/2$ on the x axis, so the rotation is on the y axis instead of z axis).
Moreover we make all the assignments negative because we want that the movements of the hands are **opposite** to the movements of the paws.
Then we have the **real** movements of the object on the x axis:

```
body.position.x += 0.8 * speedWalk;
camera.position.x = body.position.x;
light.position.x = 150 + body.position.x;
if(!rotated)text.position.x = body.position.x -text_space;
else text.position.x = body.position.x + text_space;
```

So we have the movements of the body (and so of all the dragon) on the x axis, which depends on the speed of the walking (such variable goes from 1 to 3, so the translation can be 0.8, 1.6 or 2.4).
The **camera** follows the dragon and also the **light** (which maintains its distance of 150 from the dragon).
The last two conditions is for the position of the **text** on the top of the screen, that depends if it is mirrored or not (a word can be rotated only in the hard mode).

3.2 WalkToRun

After the speed of the walk reach the upper bound, the first transition animation is performed, ***walkToRun()***:

```
if(body.rotation.z > -Math.PI/2) body.rotation.z -= 0.05;
```

First of all, we want that the dragon to run uses both hands and paws (run on four legs), so we need to rotate the body on the z axis of 90 degree, making the dragon **"fall forward"**.
Then we have to manage the **paws**:

```
if(pawR.position.x < 40){
    pawR.position.x += 0.9;
    pawL.position.x = pawR.position.x;
}
if(pawR.position.y <-25){
    pawR.position.y += 0.9;
    pawL.position.y = pawR.position.y;
}
if(pawR.rotation.z < Math.PI/2){
    pawR.rotation.z += 0.03;
    pawL.rotation.z = pawR.rotation.z;
}
pawR.scale.x = 2/3;
pawL.scale.x = 2/3;
```

Because of the rotation of the body, which is parent of the paws, we need to adjust the x and y position of the paws and also the rotation on the z axis. Note that, since at the end of the transition animation, the pasw must reach the same positions/rotations, both right and left paws are included in the same IF condition.
I decided, for estetical reason, that when the dragon is walking, the width of the paws is greater that the width of the hand. But when the dragon start to run, I want that both paws and hand have the same dimensions, so I **scale** the paws on the x.

Then there is the part for the **hands**:

```
handR.rotation.y = 0
handL.rotation.y = 0
if(handR.rotation.z < Math.PI/2){
    handR.rotation.z += 0.06;
    handL.rotation.z = handR.rotation.z;
}
if(handR.position.x < 40){
    handR.position.x += 1.2;
    handL.position.x = handR.position.x;
}
if(handR.position.y < 45){
    handR.position.y += 1.2;
    handL.position.y = handR.position.y;
body.position.x += 2;
}
else{
    state = "run";
    needWalkToRun = false;
}
```

Firt of all we directly adjust the rotation on the y of the hands, this is because such rotation was altered by the previous state of walking and since is a minimal alteration we can do a **direct assignments** without any bad visual effect.
Then like in the paws case, we need to perform some translations and rotations to bring the hands in the right position.
To decide when the transition animantion is ended I refers to the y position of the hands. In particular **when the hands touch the ground** it means that the animation can be stopped and the dragon can start to run.
Also the **head** must be positionated:

```
if(head.position.x > -30) head.position.x -= 2;
if(head.position.y < 85) head.position.y += 2/5;
if(head.rotation.z < Math.PI/2) head.rotation.z += 0.05;
```

Like the other parts, the head suffers the rotation of the body, so it must be adjusted by translations and rotations.
The last parts to be managed are the **wings**:

```
if(pivotR.position.x < -12) pivotR.position.x += 0.5;
if(pivotR.position.y > -9) pivotR.position.y -= 0.5;
if(pivotR.position.z < 14) pivotR.position.z += 0.5;
pivotR.rotation.x = -0.1;
pivotR.rotation.y = 0.4;
pivotR.rotation.z = 1.9;
if(wingR.position.y < 30) wingR.position.y += 0.9;
```

Basically we adjust the position of the internal pivots as always.
But when the dragon is walking, most of its real wings is hided in its body.
We want that when starts to run, its wings **emerge** from its body, so we need to translate them on the y.
This is the code for the right wing, the code for the left one is identical.

Finally there is the same pattern code of the walking, for moving **camera**, **light** and **word**.

3.3 Running

Now the dragon is able to run:

```
pawR.position.x =  40 - Math.cos(t) * 30;
pawR.position.y = -25 + Math.sin(t) * 15;
pawL.position.x =  40 - Math.cos(t) * 30;
pawL.position.y = -25 + Math.sin(t) * 15;

pawR.position.x = Math.min(40,pawR.position.x);
pawL.position.x = Math.min(40,pawL.position.x);

handR.position.x =  40 - Math.cos(t + Math.PI) * 15;
handR.position.y = 45 + Math.sin(t + Math.PI) * 15;
handL.position.x =  40 - Math.cos(t+ Math.PI) * 15;
handL.position.y = 45 + Math.sin(t+ Math.PI) * 15;

handR.position.x = Math.min(40,handR.position.x);
handL.position.x = Math.min(40,handL.position.x);
```

The basic idea is that, to create a realistic running, the circular movement of the paws (so the posterior paws) is **wider** than the movement of the hands, in particular we a factor of 30 in the x translation of the paws with respect to a factor of 15 in the hands.
Note that, because of the rotation on the z axis of the body, we now have that x and y are switched and so the code is adpated to this.
Finally note that we have to maintain the position defined in the previous transition animation (-25, 40 ...).
Then we have some adjustments for making the running realistic:

```
body.rotation.z = -Math.PI/2 - Math.sin(t)*0.05;
head.rotation.y = Math.sin(t)*0.08;

if(t > Math.PI*3/2 && t < Math.PI*2) pawR.rotation.z = Math.PI/2 + Math.cos(t)*0.6;
if(t < Math.PI/2) pawR.rotation.z  = Math.PI/2 - Math.sin(t)*0.6 +0.6;
```

The first two line are two minimal rotation of the head and the body to give dynamicity to the running.
The circular movement of the paws has **four phases**:
1. the paw goes on the maximum height and falls down going to the left;
2. the paw goes from the extreme right to centre;
3. the paw goes from the centre to the extreme left;
4. the paw goes up to the initial position.

The first IF condition represents the **phase 4** and here I want that the paw starts to tilt upwards.
In the second IF condition we are in the **phase 1** and I want that the paw returns to its original position.
The last part is about the **wings**:

```
pivotR.rotation.y = 0.8 + Math.cos(t+Math.PI)*0.3;
pivotL.rotation.y = -0.8 - Math.cos(t+Math.PI)*0.3;
```

What I want is that during the running the dragon moves its wings slowly, as if to prepare for the flight.
So I simply make a rotation of the pivot, that will generate the desired movement for the wings.

As usal the camera, the light and the word follows the movement of the Dragon.

3.4 RunToFly

When the speed of the running reach the upper bound, the dragon start the transition animation ***runToFly()***:

```
pawR.position.x = pawL.position.x = handR.position.x = handL.position.x = 40;
pawR.position.y = pawL.position.y = -20;
handR.position.y = handL.position.y = 20;
if(pawR.rotation.z > Math.PI/2 -2.5){
    pawR.rotation.z -= 0.05;
    pawL.rotation.z  -=  0.05;
}
if(handR.rotation.y > -2.5){
    handR.rotation.y -= 0.05;
    handL.rotation.y -= 0.05;
}
```

First of all I reset the position x and y of the paws and the hands.
Then I want that paws and hands perform a rotation in order to have them as if they were hanging in the air.
Then we obviusly must manage the wings:

```
pivotR.rotation.y = 0.8 + Math.cos(t+Math.PI)*0.8;
pivotL.rotation.y = -0.8 - Math.cos(t+Math.PI)*0.8;
```

```
if(wingR_2.position.y < 35){
    wingR_2.position.y += 1;
    wingL_2.position.y += 1;
}
```

The movements of the pivot is the same of the running animation but are **wider**.
Moreover we want to make emerging the second parts of the wings which are hidden inside the wings, so we make the translation for making emerge them.
The last foundamental part is the following:

```
if(body.position.y < 250){
    body.position.y += 3 + Math.sin(t)*2;
    body.position.x += 16;
}
else{
    state = "fly";
    needRunToFly = false;
}
```

In this case we want that the transition animation is stopped when the dragon reach the desired **altitude**.
While going up, the body suffers also a periodic movement, according to the flapping of the wings.
Moreover we note that while going up, the Dragon goes on, with speed of 16, that corresponds to the minimum speed of flight (that is 16*speedFly with speedFly = 1).

3.5 Flight

This animation is basically identical to the transition animation **runToFly()**. The only difference is that now the altitude is fixed to 250 and the body suffers only the minimal periodic translation on the y axis.

3.6 FlyToRun

This transaction animation is basically the reversed version of **runToFly()** in which we need to reach the position for the running.

4 Words

The idea to create the words that appear on the screen is to use the object **TextGeometry** provided by ThreeJs.
This leads to some problem for the implementation of the changing of the color of the letters during the game because changing the color by accessing the geometry (and so the faces) of the text is tricky.
To overcame this problem my idea is to create the word as a **Group** of ThreeJs, where each letter is a TextGeometry belonging to such group.
In this way is very simple to change the color of each letter by simply access its material.
First of all I have created a small dictionary (**dict**) where the function ***write()*** takes randomly the next word to display on the screen:

```
var loader = new THREE.FontLoader();
var write = function(){
    loader.load( 'optimer_regular.typeface.json', function ( font ) {
        text = new THREE.Group();
        next = Math.floor(Math.random()*dict.length);
        word = dict[next];

        for(var i = 0; i<word.length; i++){
            geo = new THREE.TextGeometry( word.charAt(i), {
                font: font,
                size: 150,
                height: 0.5,
                curveSegments: 0.6,
                bevelEnabled: true,
                bevelThickness: 20,
                bevelSize: 10,
                bevelSegments: 5
                } );
            mat = new THREE.MeshStandardMaterial({color: 0x0000ff});
            mesh1 = new THREE.Mesh(geo,mat);
            text.add(mesh1);
        }

        ...

        scene.add(text);
}
```

So after take randomly the next word, for each letter composing the word a new mesh is generated with the TextGeometry of the corresponding letter and it is added to the group.

Before adding the **text** to the **scene** there are some computations to adjust the position of the **text** with respect to the character and also to adjust the distance between among the letter of the word to obtain an acceptable result in term of appearence.
In addition there is the piece of code for managing the **hard mode**:

```
if(hardMode){
        if(Math.random() > 0.5){
            text.rotation.y = -Math.PI;
        ...
}
```

If the hard mode is active, the code decides randomly if the word must appears mirrored or not.
Then we need a **listener** to manage the interaction with the user:

```
document.addEventListener('keydown', function(event) {
    if(startTime != 0){
        if(event.keyCode == 13 && step == text.children.length){
                feed.style.color = "green";
                feed.innerHTML = "Good!";
                count ++;
                processCorrect();
                step = 0;
                scene.remove(text);
                write();
        }
        else if(step < text.children.length) {
            if(String.fromCharCode(event.keyCode) ==
            text.children[step].geometry.parameters.text){
                text.children[step].material.color.setHex(0x00ff00);
                step++;
                feed.innerHTML = "";
            }
            else if(text.children[step].material.color.getHex()!=0xff0000){
                text.children[step].material.color.setHex(0xff0000);
                errors++;
                feed.style.color = "red";
                feed.innerHTML = "Fix typo!";
                processError();
            }
        }
    }
});
```

First of all, the listener works only if the game is started and so if **startTime** is different from zero.
Then we have different events to manage.
The variable **step** says what is the position of the current letter to be typed, so if the user presses **Enter** (keyCode 13) and the word is all typed, what happens is the following:
A green "Good" appears on the screen (feed is the id of an element present in HTML file), the counter **count** of the word typed is increased, the **step** return to zero, the current word is removed from the scene, an aux function ***processCorrect()*** is called and finally ***write()*** is called again to generate the next word.
Else if the user presses some key and the word isn't already all typed, we have different scenarios:
If the key pressed is the correct one, i.e. corresponds to the next letter to be typed, suche letter become green and **step** is incremented.
If the key is not correct, the letter become red, the counter **errors** of the errors done is incremented, a red "Fix typo!" appears on the screen and the function **processError()** is called.
If the user inserts the wrong letter more than one times nothing happens.

We now see the ***processCorrect()*** function:

```
var processCorrect = function(){
    head.remove(mouth2);
    head.add(mouth1);

    eye.material.map = texture1;

    head.material.color.setHex(0xffb347);

    if(state == "walk") speedWalk +=1;
    if(speedWalk == 4){
        speedWalk = 1;
        needWalkToRun = true;
        state = "trans";
    }
    if(state == "run") speedRun +=1;
    if(speedRun == 4){
        speedRun = 3;
        needRunToFly = true;
        state = "trans";
    }
    if(state == "fly" && speedFly < 3) speedFly +=1;
}
```

This function basically manage the connection between the typing part with the animation of the dragon part.
The first part of the code is devoted to change the facial expression of the dragon: when the user type a word correctly, the dragon has the happy expression, so we remove the (eventual) angry expression in which we have **mouth2** and the **eye** has the texture **texture2** and we add the happy face in which we have **mouth1** and the **eye** has the texture **texture1**.
Moreover we change the color of the face to pastel orange (0xffb347).
The second part of the code is devoted to change the movements of the dragon, in particular:
If the dragon is walking, we increment the speed of the walk.
If the speed of the walk reaches the limit, we indicate that we need to perform the transition animation from walk to run using the variable **needWalkToRun**, we reset the speed of the walk and we change the state of the dragon to "trans" to indicate that a transition animation is performing.
The same approach is used from run to fly.
The flying is the ultimate animation possible, so after reaching the limit of speed of flying nothing happens.

The ***processError()*** function works in a specular way:
The facial expression become angry and the head change color to Red.
According to what is the state of the dragon, the speed is decremented and, if the speed of flying reach zero, the transition animation from flying to running is activated.

5 Floor

In the **side-scrolling games** two possible approaches can be used for the floor where the character moves.
In the first approach the character seems to move, but the only thing that actually is moving is the floor, in the opposite direction with respect to the character movement. So the character remains in the same position, while the movement of the floor create the effect that the character is moving.
In the second approach the character is really moving forward, so we need a dynamic floor that is generated continously in front of the character.
I have used the second one, in particular I have created 4 different floors:

```
geometry = new THREE.BoxGeometry(window.innerWidth , 300, 600);
material = new THREE.MeshStandardMaterial({color:0x234d20});
floor1 = new THREE.Mesh(geometry, material);

... //creation of the other 3 floors

floor1.position.x = -window.innerWidth;
floor2.position.x = 0;
floor3.position.x = window.innerWidth;
floor4.position.x = window.innerWidth*2;
```

The 4 floors together create a single continuos floor with a width equal to 4*window.innerWidth.
Now we need a function ***updateFloor()*** that manages the position of the floors in a way that when the first floor (the one on the extreme left) goes out of the screen, it is translated in front of the last current floor (the one one the extreme right):

```
var floorIndex = 0;
var updateFloor = function(){
    if(body.position.x - floors[floorIndex%4].position.x > window.innerWidth*2){
        floors[floorIndex%4].position.x += window.innerWidth *4,
        floorIndex++;
    }
}
```

Notably, it would we enough to have only 2 floors. I used 4 floors of different colors to better give the impression of the movement of the character.

6 Lights

First of all I create a white **ambient light**:

```
var light1 = new THREE.AmbientLight(0xffffff);
scene.add(light1);
```

Then I create a **directional light**:

```
var light = new THREE.DirectionalLight( 0xffffff, 1);
light.position.set(150, 150, 30 );
light.castShadow = true;
light.target = body;
scene.add( light );
```

Such light is white and has intensity 1.
It is positionated in (150,150,30) so it is at the top right of the body which is positionated in (0,0,0).
We have also to specify that such light cast the shadow and that its **target** is the body of the Dragon.
Then we have to define other parameters to create the shadows:

```
body.castShadow = true;
head.castShadow = true;
pawR.castShadow = true;
pawL.castShadow = true;
handR.castShadow = true;
handL.castShadow = true;
wingR.castShadow = true;
wingL.castShadow = true;
wingR_2.castShadow = true;
wingL_2.castShadow = true;
circle1.castShadow = true;
circle2.castShadow = true;
circle3.castShadow = true;

floor1.receiveShadow = true;
floor2.receiveShadow = true;
floor3.receiveShadow = true;
floor4.receiveShadow = true;
```

Basically we specify that all the components of the Dragon will **cast** the shadow and that the floors will **receive** the shadow.
In this way we obtain the shadow of the Dragon on the floor.

7 Game management

There are some functions devoted to the game mechanism management.
In the starting page there is a menu in which are displayed the general rule of the game and the buttons to play the classical version or the hard version of the game.
Basically, the listener of the button ”Play” basically sets to True the variable **needMoveCamera**, such variable setted to true indicates that there is the need to move the camera in the right position to start the game.
The listener of the button ”Hard mode” does the same thing and in addition sets to True the variable **hardMode** which indicates that the player wants to play the hard version of the game.
So after clicking one of the two button, the function ***moveCamera()*** is called (by the render function):

```
var moveCamera = function(){
    if(camera.position.z > 800){
        camera.position.z -= 3;
        camera.position.y -= 6;
        time.innerHTML =  gameTime;
    }
    else{
        needMoveCamera = false;
        document.getElementById("play").blur();
        document.getElementById("hard").blur();
        document.getElementById('world').focus();
        state = "walk";
        write();
        startTime = Date.now();
    }
}
```

The first part of the function moves the camera in the right position and display the time remaining to the player.
After the position of the camera is setted, we indicate that there is no more need of moving the camera by setting the variable to False.
Then we delete the focus from the buttons and we put the focus on the element **world**, that is the element in which there is the scene, in this way after clicking the button the user can immediately starting to type.

Then there is the function ***checkTime()*** that basically checks the state of the game:

```
var checkTime = function(){
    if(startTime !=0){
        time.innerHTML = gameTime - Math.floor((Date.now() - startTime) /1000);
        var meters = Math.floor(body.position.x)/100;
        mt.innerHTML = Math.floor(meters) + " Mt";

        if(Date.now() - startTime > gameTime* 1000){
            menu.style.opacity = "1";
            menu.style.top = "0px";
            document.getElementById("play").disabled = false;
            document.getElementById("hard").disabled = false;
            document.getElementById("rule").innerHTML = "<p>" +
                "Typed words: " + count + "<br><br>" +
                "Words per second: " + Math.floor(count/gameTime*100)/100 + "<br><br>"+
                "Errors: " + errors + "<br><br>" +
                "Meters: " + meters + "<br><br>" +
                "</p>"
            ;

         ...

}
```

If **starTime** is non-zero, it means that the game is started and so the function updates the clock with the seconds remaining and updates the meters done by the creature writing them in the proper HTML element.
If the time is over, the game ends and so the function has to display the results on the screen. To do that we reuse the same element used for the **menu**, writing the results in place of the rules.
The results are: the number of typed words, the words per second (approximated), the numbers of errors done and the meters done by the dragon.

The rest of the function (that I have omitted) basically **RESETS** all the variables to come back to the initial situation.

8 Rendering

The render funcion works as follow:

```
var render = function () {
    if(needMoveCamera) moveCamera();
    if(state == "walk") walk();
    if(needWalkToRun) walkToRun();
    if(state == "run") run();
    if(needRunToFly) runToFly();
    if(state == "fly") fly();
    if(needFlyToRun) flyToRun();

    updateFloor();
    checkTime();

    requestAnimationFrame( render );
    renderer.render(scene, camera);
}
```

Thus, according to the various needs the corresponding transition animation is called, according to the various states of the Dragon the corresponding animation is called.
Moreover are continously called the functions for updating the floor and for checking the time.