FINAL PROJECT REPORT, INTERACTIVE GRAPHICS

Accademic year 2019/2020

MADE BY

MATTEO EMANUELE

MATRICULA N° 1912588

THE IDEA

For this final project,was a very good idea to build a game. As a big fan of Valve Corporation products, I decided to attempt to reproduce the so famous ***Portal 2 game***, released in 2011.

The signature feature of this game is the possibility to *spawn two portals*, an orange and a blue one, and pass through them in a very smooth way,without even noticing the teleportation effect.

Here I reproduced the portals in their integrity, plus few other minus features like the possibility to grab the Companion Cubes and use them to press some pressure buttons. Everything was implemented using three.js as base library.

In this report we will start with a brief look to the collision system implemented with physijs, a physics based pugin for three.js.

We will go through the mechanics implemented, giving a specific attention to the portals, in both the visual effect and the teleportation implementation.

In the last part we will give a look on to the models used, to the hierarchical model of the character and its animation togheter with the other minor animations in the scene.

We will skip everything related to the hierarchical aspect linked to the mechanic of three.js itself, like the parent-child approch used for the bounding box on the objects, or the elements inserted in the scene which are considered children of the scene.

PHYSICS AND COLLISION SYSTEM

To implement the physics we used a plugin named Physi.js, a physics based plugin for three.js

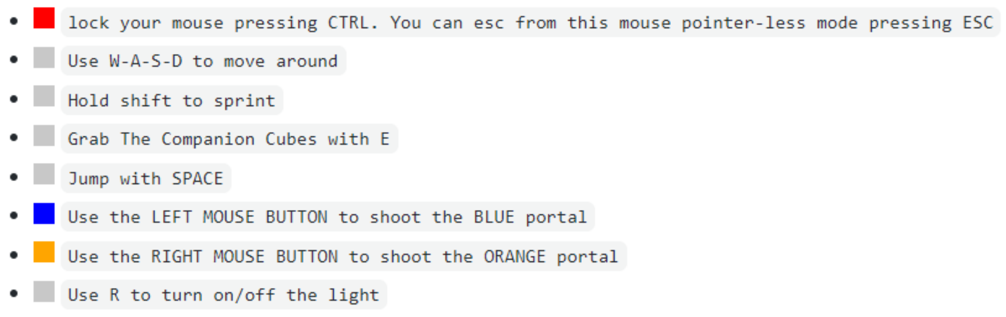
Basically the usage of such plugin allows to create meshes that are not THREE.Mesh, but Physijs.Mesh, and these new kind of mesh is completely compatible with everything from three.js, with the add-on of few parameters, which are friction and the mass.
At the start of the init() function, we will create a physijs scene with new Physijs.Scene. This new scene has the property to allows gravity to everything that is physijs-made,and that is inside it.
Adding a mass to a mesh, will bring precise physic simulated on the objects, allowing also collision between object.

This last part allowed us to introduce a precise collision system between objects. The only work was to add a physijs mesh to every object with which the collision was needed.

I Underline here how the collision detection of the playing Character has not been implemented with this system, but through raycast.

MECHANICS IMPLEMENTED

We report here a snippet from the README file from the github project:



- 🟥 `lock your mouse pressing CTRL. You can esc from this mouse pointer-less mode pressing ESC`
- ⬜ `Use W-A-S-D to move around`
- ⬜ `Hold shift to sprint`
- ⬜ `Grab The Companion Cubes with E`
- ⬜ `Jump with SPACE`
- 🟦 `Use the LEFT MOUSE BUTTON to shoot the BLUE portal`
- 🟧 `Use the RIGHT MOUSE BUTTON to shoot the ORANGE portal`
- ⬜ `Use R to turn on/off the light`

These are the command implemented in the project. We will take a look at them, leaving the portals for last,being them filled of technical notions.

Movement, grab, jump & flashlight

Everything related to the character, and the movement too, is included inside a file named FirstPersonCharacter.js.
This was thought to be a homemade library for the first person character that we will play in the game. The movement was simply implemented through key buttons to add a direction information to the character object. This changes were checked at runtime to compute the movement in the x-y plane. The orientation of the movement was lead by the camera. The camera is implemented using PointLockerControl.js. Is basically a library for the first person camera. It is less accurate for such job than firstPersonControls, the most famous library for first-person-visualization, but PointLocker had a specific feature that was key for our job: the possibility to lock the mouse and hide it.

Being both the right&left mouse button needed for shooting,we really needed to hide the arrow, and the only way to do it was through such library.

What keep track of the collision with walls and floors for the character,being him not physijs-based, is the function MovementTrace(). Basically, this function “trace” the movement through a complex system of raycasters starting from the character and going towards 6 different directions. This system implement a less accurate system of collision with respect to the physijs method, but was needed because at the start of the project, the implementation of the body capsule with physijs led to different problems related to the jump, and the compenetration of the body in the ground. Plus, also the movement was needed to be built through SeLinearVelocity() method from physijs, seeing as that a computation of the position at runtime used to override the calculus of the physic engine, make it not work for the character. This choice was needed. Though,it works still quite well.

Last mention to the shift button, that was implemented to increase the velocity feature of the character object, and accordingly increase the movement speed, simulating a “sprint” of the character.

The grab was implemented with the simple function inside the case 69 of the OnKeyDown() function called as a callback for the key events.

The updateGrab() function then is called at render time to keep track grabbing event and handle it accordingly.

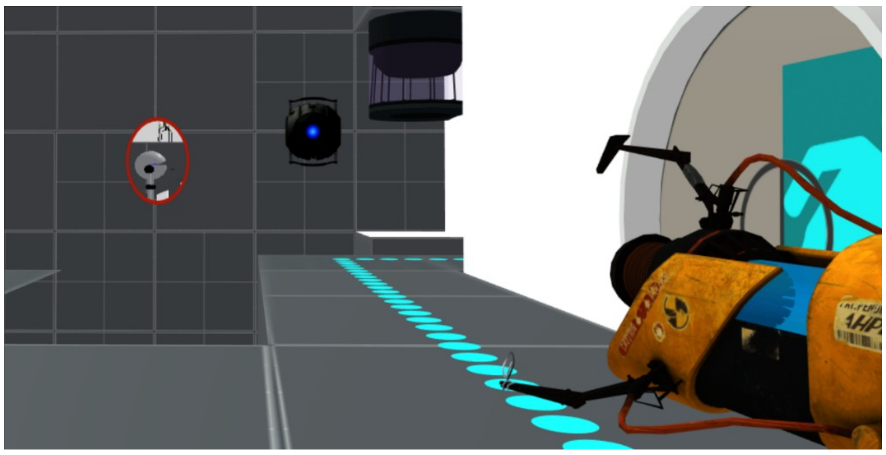
It is nice to underline how the effect of the cube that follow the character once is grabbed***, is not made by attaching as a child the cube to the camera***, but through a fix of the y position of the cube on the plane in front of the camera. Then, it is used the physijs to compute at runtime the linear velocity needed for the cube to follow the camera. *This leads to a very pleasent effect of smoothness in the movement while grabbing the cube.*

The jump action was implemented simply applying a force to the character upwards, and then applying a simulated physics each frame using the delta(from the getDelta() function).

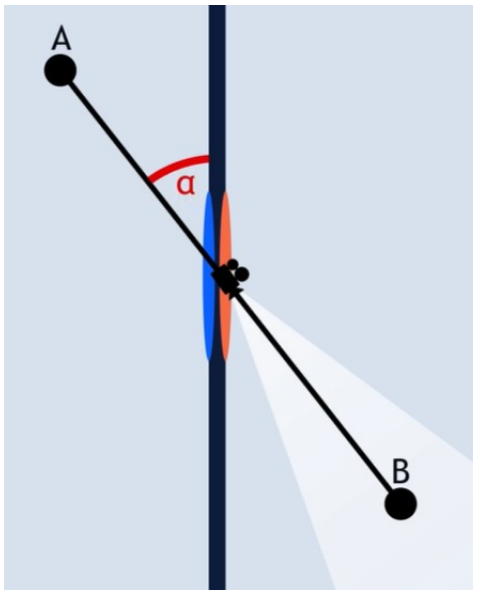
All the forces applied to the character are taken into account and handled into the UpdateForce() function.

Last shoutout to the R button and its functionality that is to turn on or off a flashlight directed toward the with respect where the player is looking. This simple feature was implemented because for those who are able to complete this short level, the final event will trigger some dark all around the room, and the flashlight can help to keep playing after this happened.

THE PORTALS: the visual effect



As it is showed in the pictures above,the visual effect was one of the signature features that I wanted to implement perfectly for the project. This took quite a lot of work, being a very complex visual effect that I will try to explain now.



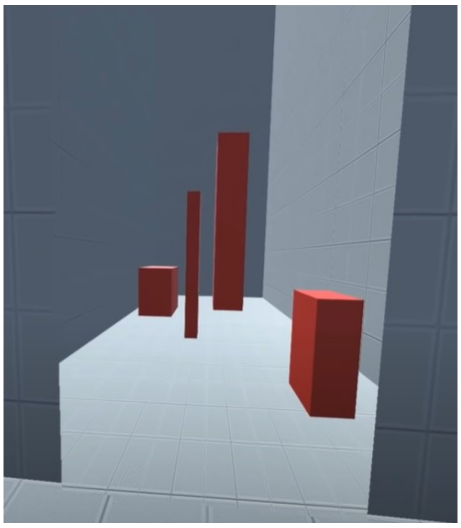
We will start saying that the effect was obtained through a render texture. Onto this render texture, we rendered what a camera, placed behind the portal, was looking at.

For each portal we have a camera, so we have two camera plus the one of the character.

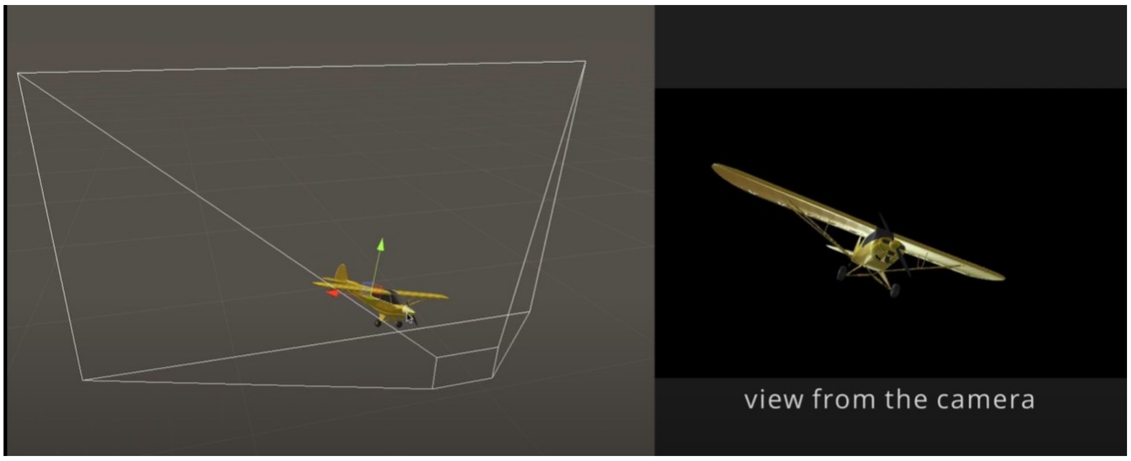
The intuition behind it, is that if we are placed in a point, lets say B(figure) close to the camera, with a portal in front of us, we would be able to see what the camera of the other portal is seeing.

Also, what we see has to be at the same distance that we are from the first portal, so we have to take into account the FOV of the camera to keep it as realistic as possible.

If we implement “simply” this concept, we will obtain this:

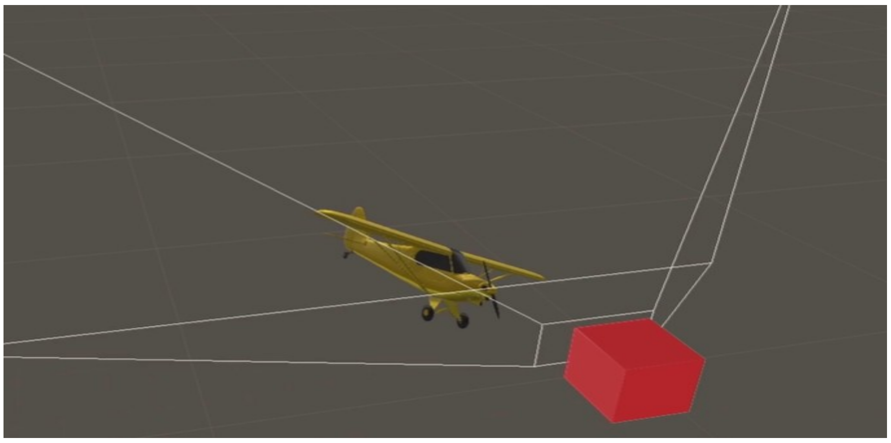


It is what we wanted, or no? There is something off about what we are seeing. Everything is put into perspect, and is placed “correctly”, but the problem is that what we are looking at, is just “too much 2D.” This is because we are not taking into account the depth of the scene, that when rendering on a texture lose all its depth component.



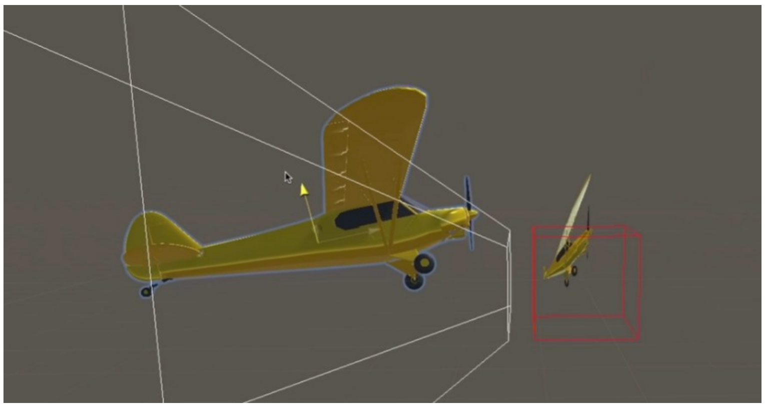
To explain the technic used for the effect, let’s look at the picture above. As long as the airplane in the figure is inside the delined area, which represent the camera frustum, the camera will be able to see it.

Taken the projection matrix of the camera, if we apply such projection matrix to the object, we will obtain this:

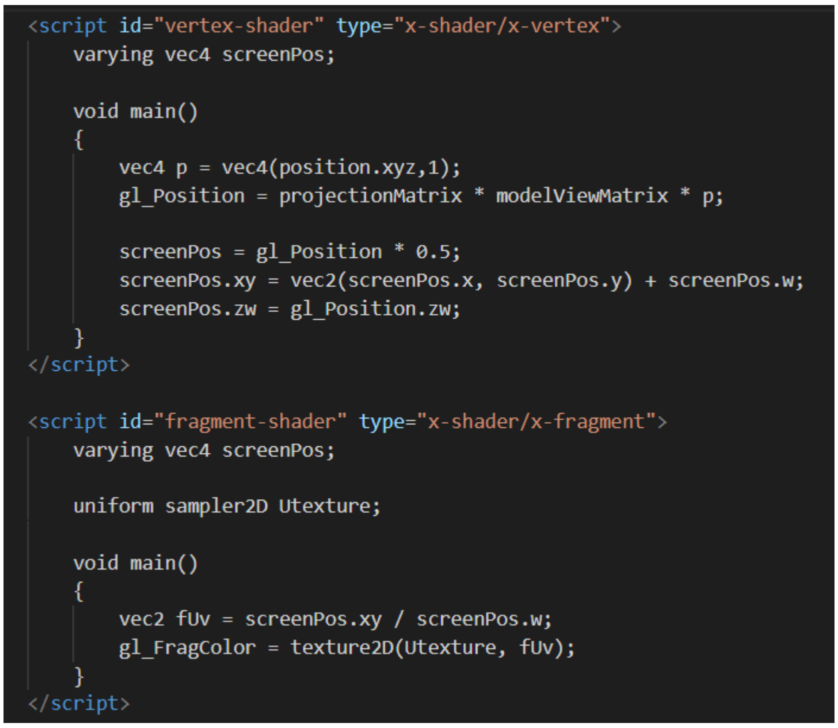


The result of the projection matrix application though, is a 4 dimensional vector vec4(x,y,z,w), where the last component is ***proportional to the distance from the camera.***

If we are able to define the position of each vertex of the object with respect this last component, we are able to represent the object inside that red box drawn in the picture. That box represent the 3D environment we see when looking inside the camera.

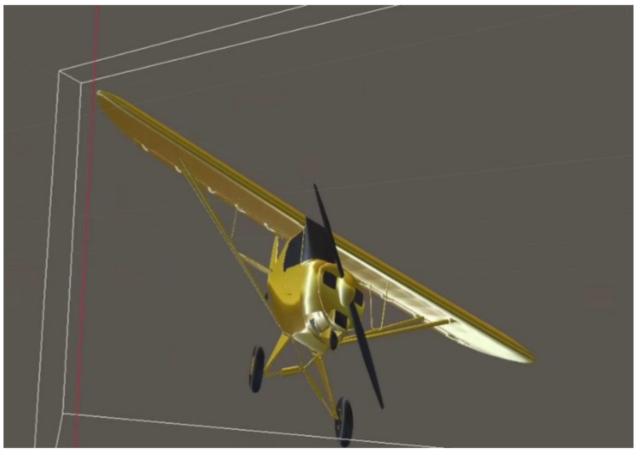


In the picture here is shown the result if we redifine the x,y component of this vector adding to the original one fourth component w in the vertex shader, and then we divide for the component w the result in the fragment shader:



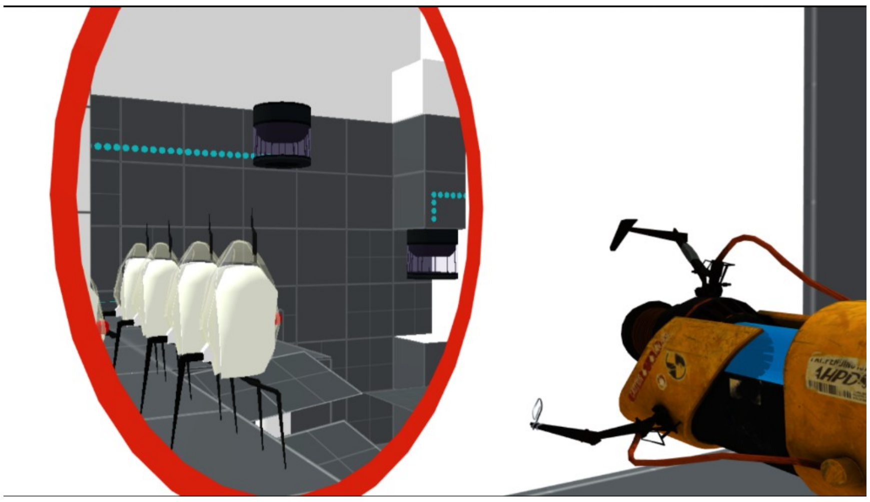
```
<script id="vertex-shader" type="x-shader/x-vertex">
    varying vec4 screenPos;

    void main()
    {
        vec4 p = vec4(position.xyz,1);
        gl_Position = projectionMatrix * modelViewMatrix * p;

        screenPos = gl_Position * 0.5;
        screenPos.xy = vec2(screenPos.x, screenPos.y) + screenPos.w;
        screenPos.zw = gl_Position.zw;
    }
</script>

<script id="fragment-shader" type="x-shader/x-fragment">
    varying vec4 screenPos;

    uniform sampler2D Utexture;

    void main()
    {
        vec2 fUv = screenPos.xy / screenPos.w;
        gl_FragColor = texture2D(Utexture, fUv);
    }
</script>
```

If now we redifine the depth and the w component of the object seen by the camera onto the "screen" (our portal, basically) as the basic third and fourth component of the object, we are able to obtain a perfectly flat image that is still helding the information on the depth depending on the distance of the object from the camera:

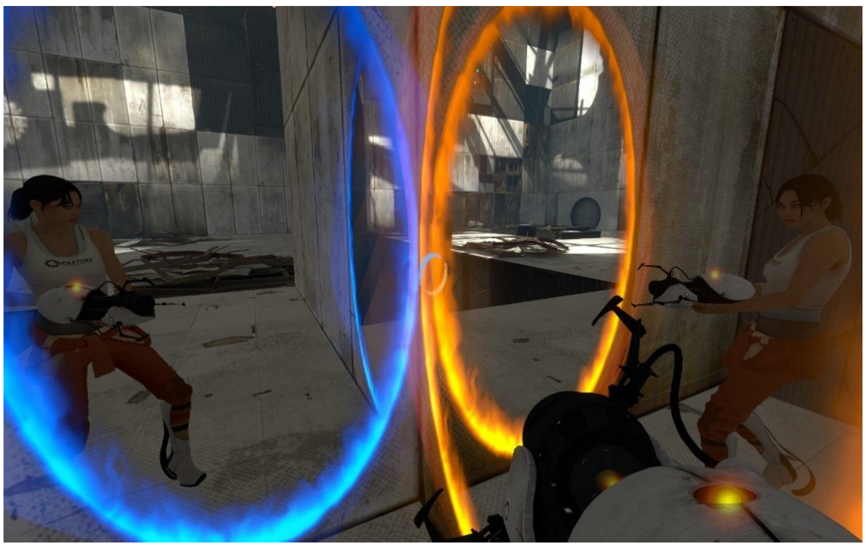


Here is reported the “flattenized-without-depth-loss-information” picture of the airplane.

Here we show an ingame picture to realize better what we are seeing after this explaination:

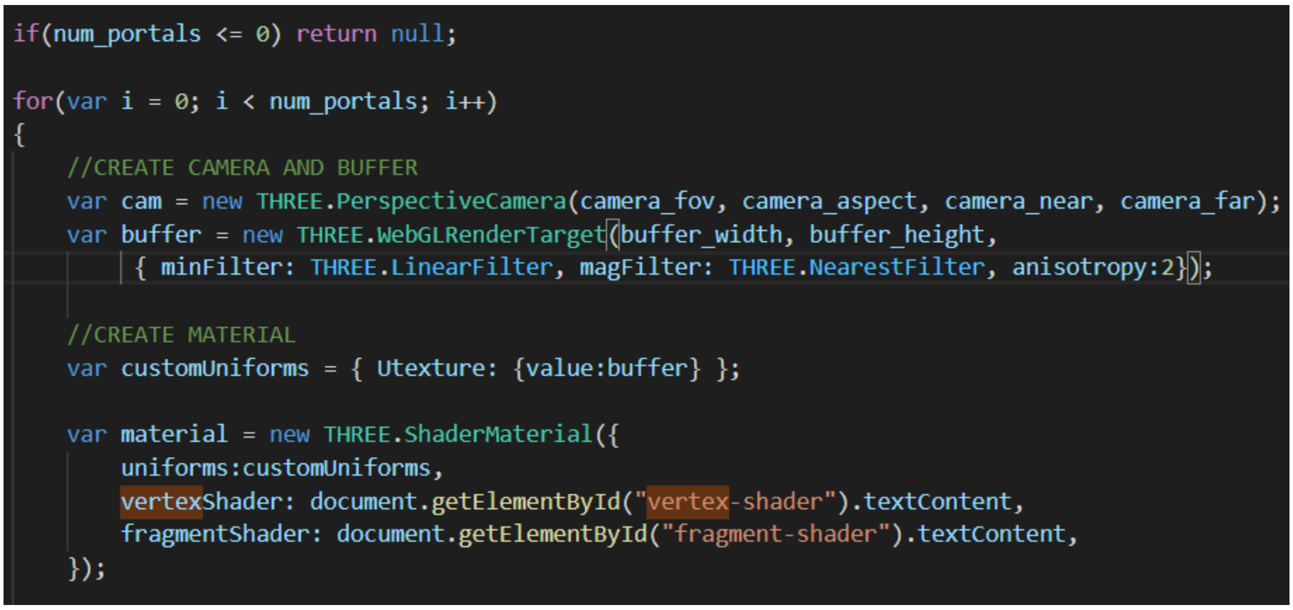


We can say proudly that the result was beautifully obtained. We report here a picture from the original game to show the comparison:



Last technical thing that we can say, is that the cameras of the portals, when they are rendering, need to change their position with respect the distance of the player to the portal, and also ***we have to take into account the znear of the camera and change it dinamically with the distance***, or else we will render part of the wall where the portal is attached to.

To explain as all this visual effect was implemented practically, I wrote a vertex and a fragment shader to manually make it render on material of the portals:



```
if(num_portals <= 0) return null;

for(var i = 0; i < num_portals; i++)
{
    //CREATE CAMERA AND BUFFER
    var cam = new THREE.PerspectiveCamera(camera_fov, camera_aspect, camera_near, camera_far);
    var buffer = new THREE.WebGLRenderTarget(buffer_width, buffer_height,
        { minFilter: THREE.LinearFilter, magFilter: THREE.NearestFilter, anisotropy:2});

    //CREATE MATERIAL
    var customUniforms = { Utexture: {value:buffer} };

    var material = new THREE.ShaderMaterial({
        uniforms:customUniforms,
        vertexShader: document.getElementById("vertex-shader").textContent,
        fragmentShader: document.getElementById("fragment-shader").textContent,
    });
```

The math applied in the shader can be readed in the upper picture posted.

<u>Bottom Pharagraph Note</u>: I want to state here that even though the effect obtained is exactly as the one from portal the game, the technic used in this project was different. The effect of the portal rendering from the original game was obtained through the usage of Stencil Buffers. At the start the idea was to use them, but they appeared to be hard to handle, and almost no material was found online to study this alternative implementation so I decided to pivot onto this path of approach.

THE PORTALS: the teleportation effect

This feature was probably the hardest one to implement in the way I had it in mind. If the visual effect was complex and sophisticated, that was known and the formula to obtained was all it was needed to make it work. But for the teleportation, there is no magic formula that solves all the problems, and as I went deeper into this implementation, the more I realized how many problems could pop up.

We will start explaining the basic intuition behind it.
When the character is "close enough" to the portal, we want the character to change his position and move to the other portal's location. ***A simple position.set can intuitively do the job, but the problem is way more problematic than that.***

We have to think about the portals as if they are windows. If I want to jump from a window going straight forward, I expect to land "straight-forwardly" on the other side. But if I want to jump with a direction that is not straight, like in an inclined way, then I expect to land respectevely on the other side.

Here we can already imagine the criticality of such implementation: we have to realize what is the direction of the movement with respect where the camera of the portal is looking at,and then we have to compute the new direction respectevely. Initially, the implementation of this took time because I stucked myself into some unpleasant matrix computation, having some trouble passing from local reference to world reference when extracting the direction/position of the cameras.
The idea that allowed me to solve this problem was indeed simple though. How can I know the exact direction in which I have to be looking at when passing through the portal, without feeling any dizzy aftereffect or artifacts? If the basic implementation I attempt first was made of rotation, euler and quaternion conversion, this other solution was simple and extremely effective. The camera of a given portal, is already pointed in the exact direction I want to be directed to when passing through the portal**, so the trick was simply to copy the transformation matrix of the camera** that is rendering on the portal I'm passing through to the camera of the character.

Another problem was related to the position of spawning. If I pass through the portal centered in the vertical axis of the portal itself, then I just need to spawn on the center of the other portal. But the implementation was thought to solve all the problems in general about teleportation, and not only to make it work for our case. If the character is approaching the portal from his left or right most edge,we expect to spawn on the specular position on the other portal. So we had to take into account also this problem when implementing it. This was easily done using the .reflect() function with respect to the vertical axis.

There is a statement to make: ***the teleportation was not implemented using the distance with respect the portal.***

Even though this may seem counterintuitive, ***there are quite few reasons behind this choice***. The first one, is just for coding purposes: if we want to implement the teleportation of the cubes also, the computation of the distance from each portal to any other object at each frame would be very incovienent to implement.

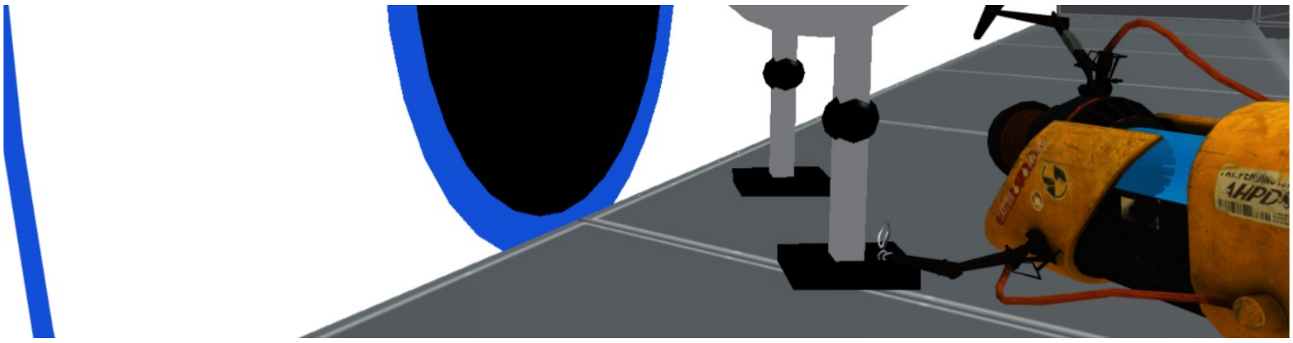
Thanks to physijs we had another great tool in our shelf to use: ***the collision callback***, and that is exactly as we implemented the teleportation. This is also why every object has a mesh in the code, even the character that compute the movement tracing using the raycast system. It is needed to catch on the collision callback.

So basically the implementation idea is the following explained here, but with the difference that each time a collision is registered between a body and the portal mesh, the onCollision() callback is called. This function will take into account the teleportation system of both the character and the cube.

Another tricky part to solve was the feeling of the teleportation. When moving through the portal, even though everything was working, it happened for most of the time of the implementation that when the character was teleporting ***there was a small flicker of the camera, given by the repositioning of tbe body center***. ***The game of Portal has a very smooth feeling***, allowing the player to literally stand between two portals. I wanted to achive such effect too, and I found a very strategic solution.
The flicker of the camera is given by the fact that what the portal is rendering,is what the camera is seeing,and the camera see everything from the portal position to everything that is inside the frustrum.
Most of the teleportation flicker was given by the fact that even though the camera was rendering from the very near edge of the portal,the mesh of the character was of a certain width.
The cleaver trick here was produced by this thought***: if I force the rendering of the camera to render everything that the camera see from <u>half the width of the character</u>,*** then I will be able to teleport the center of the body of the character without feeling any flicker effect,seeing as that what I'm seeing inside the portal is exactly what I will see once I spawn on the other side.

So the portal collision take into account the width of the character,and use that to computer what to render,in order to make this teleportation as smooth as possible.

Here a picture is reported:



As we can see from this picture, we are almost inside in the middle of the portal(0.54 of distance from the center to be precise), and still havent teleported.

In the real game of portal, there was the possibility to stand exactly in the middle of the portal. That was obtained with another trick for sure, but I wasn't able to find any explanation anywhere. It probably disable the character collider during certain event related to the distance of the portal(a calculus that really has to be precise) and "cutting a hole" into the wall of the same shape of the portal,allows to pass through it,and stand almost in the middle of the portal, being the body without the collider mesh reduced to a single point. This was not implemented,being very sophisticated and looked like a kind of a long shot.

The smoothness of the portal passage togheter to the visual effect made this experiment a huge success for my expectation, being the portals the main feature of this final project of mine.

The last criticality that will be treated just superficially here, being the one who causes most of the current bugs, was the force handling. The basic idea behind it is that if we have an object passing the portal, ***we would like that the forces are kept during the passage as much as possible.*** This was more difficult, given the teleportation. The fact that the position of the object is recomputed for a single istant, makes the forces a little unstable to be kept. For the character, with some trial and error approch almost all the known problems were solved,while for the objects(the cubes) there are some unbalances in the forces distribution during the teleoport. When the cubes are entering the portal in a specific combination of direction-linear velocity, the results will not be smooth as I would wanted to.

The last impracticality,that was also the hardest to deal with, was the ***camera reposition during teleportation***. If we have a portal in front of us,and the other one is placed on another surface with the same normal, everything works just fine.

But if we place the portals in a way that their norms are not parallel, the trasformation matrix applied to the character camera,will makes the character to be oriented in a bad way, loosing the balance and fell on the ground while the visual camera seem to be pointing upwards. There has been some debug process that took different days to fix all the particular cases of all the portal positioning. Everything was solved though, making the portal applicable to every couple of surfaces.

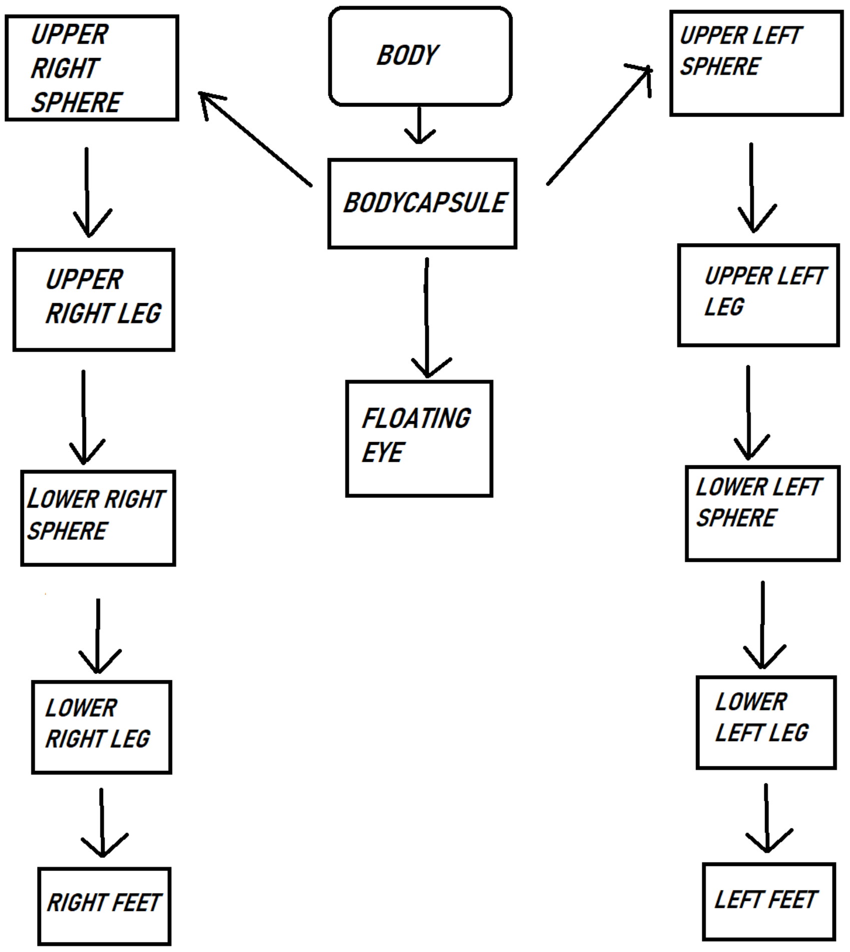
HIERARCHICAL MODEL OF THE CHARACTER & ANIMATIONS

The body model

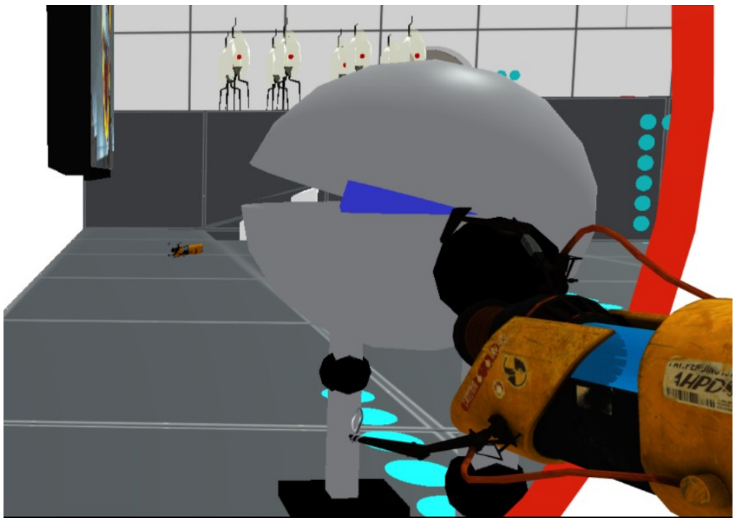
After all the talk about the character collider meshes, I will point out here istantly that the character that can be seen moving as the character body was not a collidable model. It was placed only for esthetic,and was built inside the code, so no model for it was imported whatsover.

The decision to not import anything but just straight code the body was needed being the seen kind of heavy in terms of polygons, and this was a decision of computational economy.

The hierarchical structure of the body is the following:



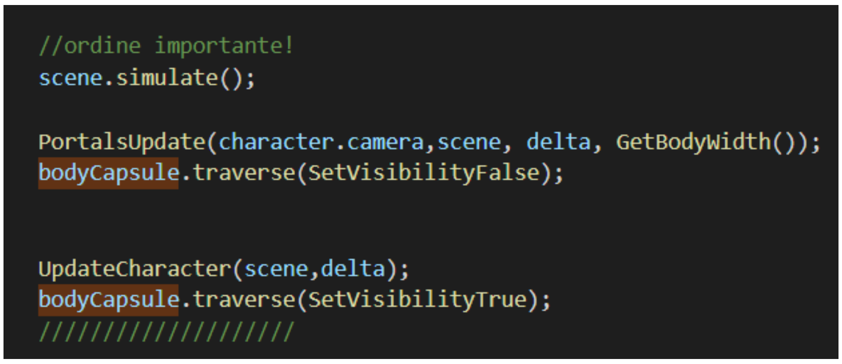
The body appear to look like this:



I suppose the graph showed is quite self explanatory about the hierarchical model. A thing that I can say is that thanks to the three.js infrastructure was way easier to implement a model like this,using the .add() function of the object3D() from three.js.

The interesting thing about it was how not to make it compenetrate the camera. Basically the camera is at the center of the body,and thus we should be able to see it. The truth is that the body is added to the camera,but is not rendered by the player camera,but only by the portals camera. So it can be seen,but is not showed blocking the view of the player.

This was implemented with this simple trick:



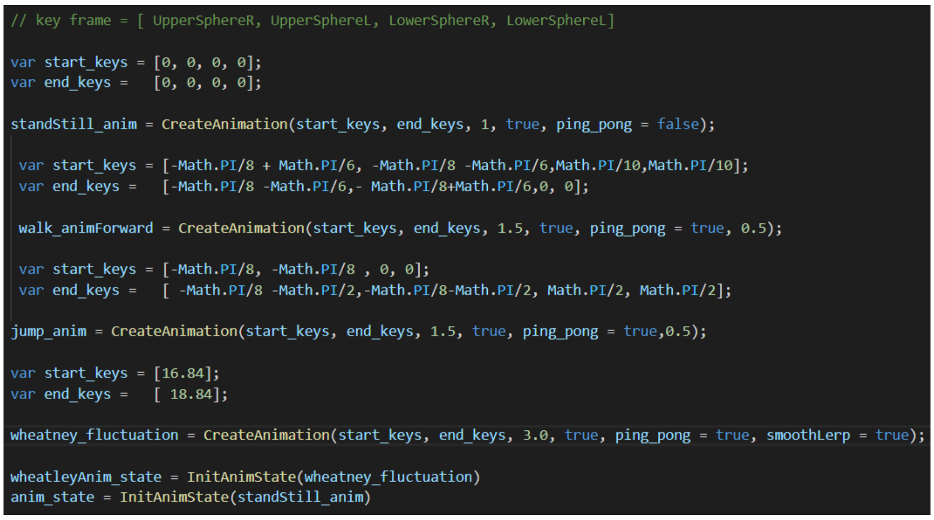
```
//ordine importante!
scene.simulate();

PortalsUpdate(character.camera,scene, delta, GetBodyWidth());
bodyCapsule.traverse(SetVisibilityFalse);


UpdateCharacter(scene,delta);
bodyCapsule.traverse(SetVisibilityTrue);
////////////////////
```

Basically we update the camera of the portals, then we set the visibility of the body to false,then we update the character camera,and then we update the visibility of the body back to true. Is quite a simple trick,and do the job perfectly.

The animations

There are three main animation in the scene. Two of these reguard the body of the character,and are the “jump animation” and the “walk animation”.

The other animation in the scene is the “floating animation” from wheatley, robot head implemented to look at the character all the time while floating up and done in a slow and smooth way.



```
// key frame = [ UpperSphereR, UpperSphereL, LowerSphereR, LowerSphereL]

var start_keys = [0, 0, 0, 0];
var end_keys =   [0, 0, 0, 0];

standStill_anim = CreateAnimation(start_keys, end_keys, 1, true, ping_pong = false);

 var start_keys = [-Math.PI/8 + Math.PI/6, -Math.PI/8 -Math.PI/6,Math.PI/10,Math.PI/10];
 var end_keys =   [-Math.PI/8 -Math.PI/6,- Math.PI/8+Math.PI/6,0, 0];

 walk_animForward = CreateAnimation(start_keys, end_keys, 1.5, true, ping_pong = true, 0.5);

 var start_keys = [-Math.PI/8, -Math.PI/8 , 0, 0];
 var end_keys =   [ -Math.PI/8 -Math.PI/2,-Math.PI/8-Math.PI/2, Math.PI/2, Math.PI/2];

jump_anim = CreateAnimation(start_keys, end_keys, 1.5, true, ping_pong = true,0.5);

var start_keys = [16.84];
var end_keys =   [ 18.84];

wheatney_fluctuation = CreateAnimation(start_keys, end_keys, 3.0, true, ping_pong = true, smoothLerp = true);

wheatleyAnim_state = InitAnimState(wheatney_fluctuation)
anim_state = InitAnimState(standStill_anim)
```

We will focus on the character animation first,and then we will briefly see the floating animation.

All the animation were implemented *bare hands*, using the infrastructure code that was build in the second homework, composed of the following component:

-an *AnimationPlayer*;

-a set of *key frames*, divided in starting and finishing frames;

-an *updateAnimation* function;

-an *updateBlending* function;

-a *createAnimation* Function;

a *initAnimState* function;

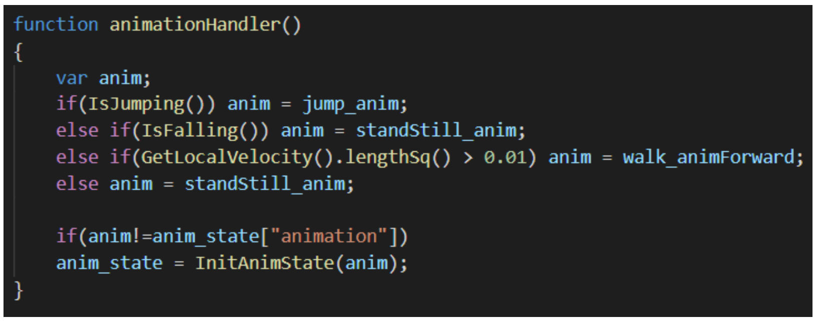
The function of such system will not be repeated here for the seek of verbosity, but was explained throughly in the second homework report, so we will just explain the animation itself.

As we can see from the last picture, the animation were built with only two key frame,seeing as the animations for how they were conceived were implementable with just two keyframe.

The code is self explanatory. Once the animation are created, the animation player will handle the execution of both the blending and the animation itself.

The jump and walk animation, obviously, has to be done only when, respectevely, jump and walk. Also another dummy animation were implemented,which is the "standstill animation" that is executed everytime the character is not moving,nor jumping.

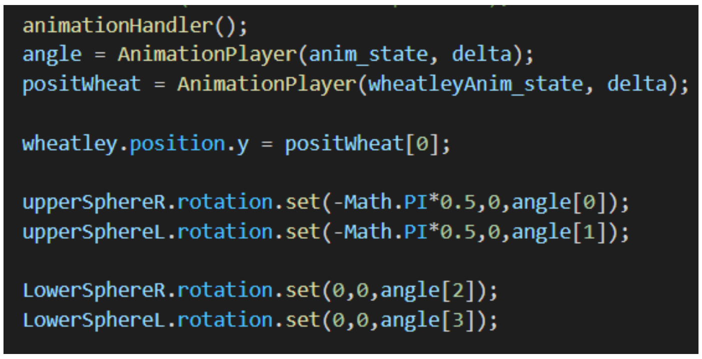
The logic of the animation was implemented through the function AnimationHandler(), that is reported here:



```
function animationHandler()
{
    var anim;
    if(IsJumping()) anim = jump_anim;
    else if(IsFalling()) anim = standStill_anim;
    else if(GetLocalVelocity().lengthSq() > 0.01) anim = walk_animForward;
    else anim = standStill_anim;

    if(anim!=anim_state["animation"])
    anim_state = InitAnimState(anim);
}
```

About the animation of Wheatley, the creation was the same,the only difference is that the animation needed to run regardless of any logic,so the animation player execute the floating animation at render time directly.

Here there is the render part of the animations:

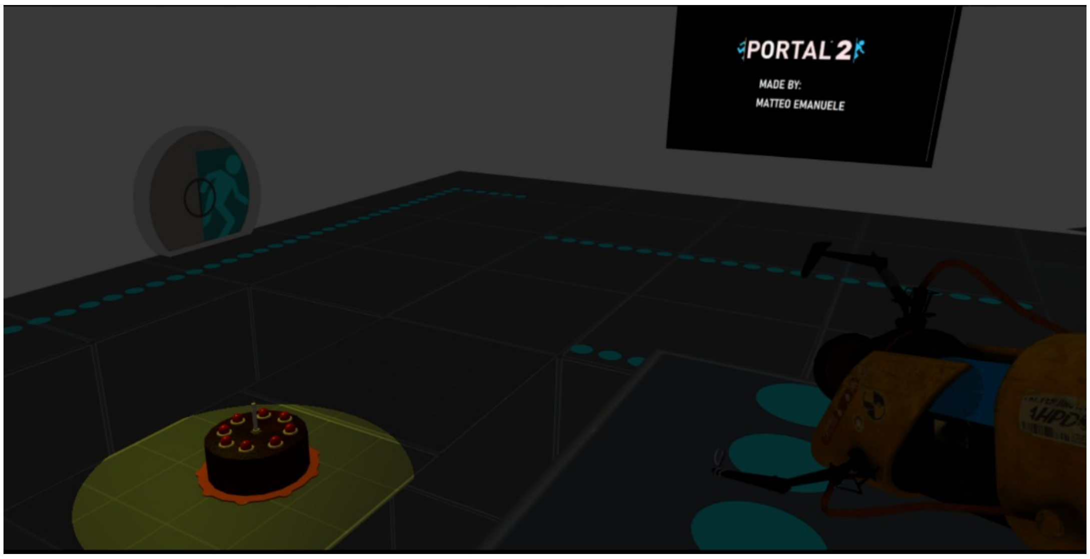


```
animationHandler();
angle = AnimationPlayer(anim_state, delta);
positWheat = AnimationPlayer(wheatleyAnim_state, delta);

wheatley.position.y = positWheat[0];

upperSphereR.rotation.set(-Math.PI*0.5,0,angle[0]);
upperSphereL.rotation.set(-Math.PI*0.5,0,angle[1]);

LowerSphereR.rotation.set(0,0,angle[2]);
LowerSphereL.rotation.set(0,0,angle[3]);
```

LEVEL DESIGN

Even though it was not part of the evaluation,I want to dedicate this last half of paragraph to the level design. The idea is simple: there are buttons around the room,some of the partially or totally hided.

Activate the buttons pressing them using the cubes, and if u press the right button with the right cube, you will make another cube spawn. If you succeed at pressing all the three buttons, you will unlock the ending, which is a quote to one of the signature scene of the original game Portal:



Where the infamous cake from the game spawns,illuminated by a cone of light in a dark room,while the ending song of Portal 1 plays in the background.

The idea was quite simple,but seeing as all the basic mechanics of the game were implemented,basically is it possible to implement even more complex level from the level-design point of view,and enjoy such work even more. This was not done seeing as I was lacking of models and also because it is a very time-consuming thing to do,and it was not the purpose of this course.

REFERENCES

- *Coding adventures: Portals – the video from Sebastian Lague. The picture for the visual effect explanation were picked from this source. All right reserved.*

- *How were portals in Portal created | Bitwise*

- *https://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix/projection-matrices-what-you-need-to-know-first*

- *https://www.turiyaware.com/blog/a-solution-to-unitys-camera-worldtoscreenpoint-causing-ui-elements-to-display-when-object-is-behind-the-camera*

- *http://www.terathon.com/lengyel/Lengyel-Oblique.pdf*

- *http://tomhulton.blogspot.com/2015/08/portal-rendering-with-offscreen-render.html*

- *https://discourse.threejs.org*

- *https://threejs.org/docs/*

-*Test room model used from Far0s, sketchfab : https://sketchfab.com/3d-models/portal-2-test-room-d2bc3fdf30ff486d9932def6f6b89ac7*

- *Portal radio model for the spatial sound from nrebei2,sketchfab : https://sketchfab.com/3d-models/portal-gun-companion-cube-radio-3ada09157e3941a8b6fad405611fe08b*

- *portal gun model from donnie7102006, sketchfab :*

https://sketchfab.com/3d-models/mel-portal-gunwith-download-c2c834f4a8d54e26accb2182714d523c

- *Musics picked from youtube videos*