基于 three.js 的 web 游戏项目

在布置下期末项目以后，我是打算用 Webgl 来做我们的渲染系统，后来在查找资料的过程中发现了 three.js 这个 webgl 第三方库，发现非常的有意思，在权衡之下，我选择使用了这个第三方库。

选择这个库表理由有以下几点：首先，这个库的功能非常强大，提供了许多的接口、模块和函数，使得我们可以把精力放到渲染出更漂亮的系统上来。

其次，该库配备了丰富的数学函数、数据结构以及开发游戏最重要的一套物理引擎，因为以前我接触过 unity 的游戏开发，所以对于开发小游戏有一定的经验，正巧这个库配备了物理引擎，正适合开发游戏，模拟一些真实的场景和物理环境。

当然，这个库也有一些不太好的地方，比如说没有详细的官方文档，网上可供参考的资料有限以及版本兼容性差（往往相邻的版本都有很多的函数不能兼容使用）

但是，总的来说，这个库的优点远大于缺点，使用它开发一个渲染系统能起到一个事半功倍的效果。

下面是我的这个项目介绍:

项目说明

该项目可以使用 WASD 控制碰撞器移动，用空格来控制跳跃，用鼠标来显示视角方向。接下来我将展示游戏的实现细节。实现游戏细节我将以问题的形式来介绍:

1 谁是主角

2 如何渲染

3 如何移动及切换视角

4 如何判断碰撞

但是还需要等一下，在进入回答问题之前，我们有必要先对一些代码中使用到的模块和方法进行解释说明。

PerspectiveCamera

模型是要在放到一个场景中展示的，这个场景就是处在世界空间中。所有物体的位置，都是以世界空间的坐标系定义的。

模型从模型空间到世界空间，就需要指定一个转换矩阵（这个矩阵可能包括旋转，缩放和平移），这个矩阵就叫做**模型矩阵**（Model Matrix）。

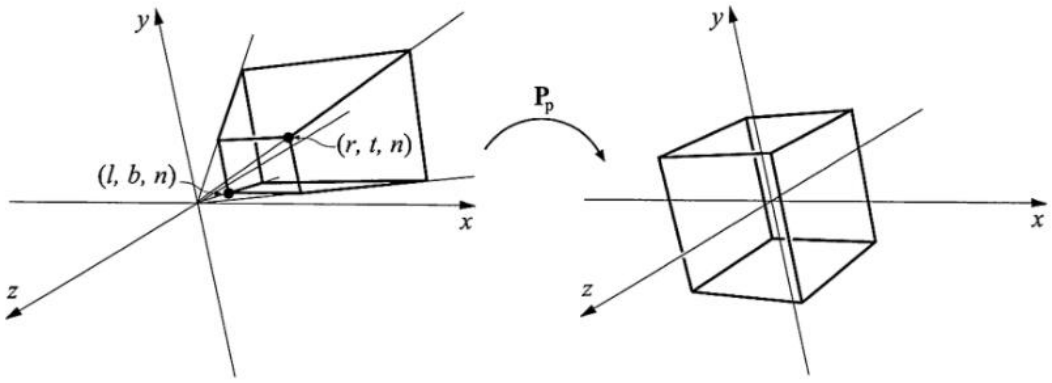
世界空间最后要展示到 Camera 中，需要做一次世界空间坐标向视图空间的转换。这个矩阵就是视图矩阵（View Matrix）。

这里会有一个问题，为什么模型一开始不转换到视图空间中，而要在中间插入一个世界空间呢？这么做的目的就是可以让设计者可以只关心整个世界的构造，而不用关心摄像机的位置。

我们绘制的一切东西，最后都要展示到一个二维屏幕上。而且这个二维空间中物体呈现的效果是带有透视效果的，简单说是近大远小的效果。

那在转换为二维屏幕坐标前，透视投影矩阵会将平截头体转换为一种单元立方体，它的中心在坐标原点，上下左右的边界从-1 到 1。这就是 **CVV**（canonical view volume），规则

观察体。里面的坐标称为 **NDC**，标准设备坐标。有了这个坐标，就不需要考虑具体设备分辨率的影响，之后的一系列操作就基于标准的坐标系。最后再经过视口转换，将-1 到 1 之间的点转换为真正的设备分辨率的点坐标。



PPM 需要四个参数确定：
1）屏幕的高宽比 Aspect
2）相机的垂直视角大小 Fov
3）近剪裁面位置
4） 远剪裁面位置
这几个参数其实就是 View Space 中的平截头体的参数。透视投影就是将平截头体中的点，投影到 CVV。
具体的推导过程可以参考：
http://wiki.jikexueyuan.com/project/modern-opengl-tutorial/tutorial12.html
其中有几个关键点：
1) 公式的推导主要借助了相似三角形原理中，相似三角形的对应边成比例。推导出了由摄像机处发出的穿过点 P 的射线，在投影平面和点 P 所在的 xy 平面之间形成的线段，与 z 轴与这两个平面之间形成的线段的比例关系。
2）为了保留 z 坐标在深度测试中的使用，z 坐标要保留到 w 中。
3）z 坐标在 CVV 中的范围是-1 到 1，所以通过变换近平面的 z 坐标转换到-1，远平面的 z 坐标转换到 1，通过这两个方程，就可以确定 PPM 的第三行的值。
而其中的透视相机就是使用透视矩阵；这个投影模式模拟人类视角。三维空间渲染中最常见的投影模式。

用法：

Var camera = new THREE.PerspectivaCamera(60,width/height,0.1,1000);
Scene.add(camera);

构造函数

PerspectiveCamera(fov, aspect, near, far)
Fov – 相机的视锥体的垂直视野角
Aspect – 相机视锥体的长宽比
Near – 相机视锥体的近平面
Far – 相机视锥体的远平面

属性

.aspect 相机视锥体的长宽比，一般都是画布 canvas 的宽高比。默认是 1（正方形）
.far 相机视锥体远平面。默认是 2000
.fov 默认 50 度，从下到上。相机视野角。
.near 默认 0.1，视锥体近平面值
x-子相机水平偏移值
y-子相机垂直偏移值
width-子相机的显示宽度
height-子相机的显示高度

updateProjectionMatrix ()

如果相机对象与投影矩阵相关的属性发生了变化，就需要手动更新相机的投影矩阵，执行 camera.updateProjectionMatrix();的时候，threejs 会重新计算相机对象的投影矩阵值，如果相机对象的投影矩阵相关属性没有变化，每次执行.render()方法的时候，都重新计算相机投影矩阵值，会浪费不必要的计算资源,一般来说就是首次渲染计算一次，后面不执行.updateProjectionMatrix()方法 threejs 不会读取相机相关属性重新计算投影矩阵值，直接使用原来的值就可以。

无论正投影相机还是投影投影相机对象的.near 和.far 属性变化，都需要手动更新相机对象的投影矩阵。

比如透视投影相机对象的.aspect、.fov 属性变化，会影响相机的投影矩阵属性.projectionMatrix,需要执行.updateProjectionMatrix ()更新相机对象的投影矩阵属性

stats

stats 是 three.js 提供的一个性能监测插件在 Three.js 里面，遇到的最多的问题就是性能问题，所以我们需要时刻检测当前的 Three.js 的性能。现在 Three.js 常使用的一款插件叫 stats。

stats 插件的核心指标：FPS: 画面每秒传输帧数。

那么如何使用它呢，我们需要实例化一个 stats 对象，然后把对象内生成的 dom 添加到页面当中。最后，我们需要在 requestAnimationFrame 的回调里面进行更新每次渲染的时间：

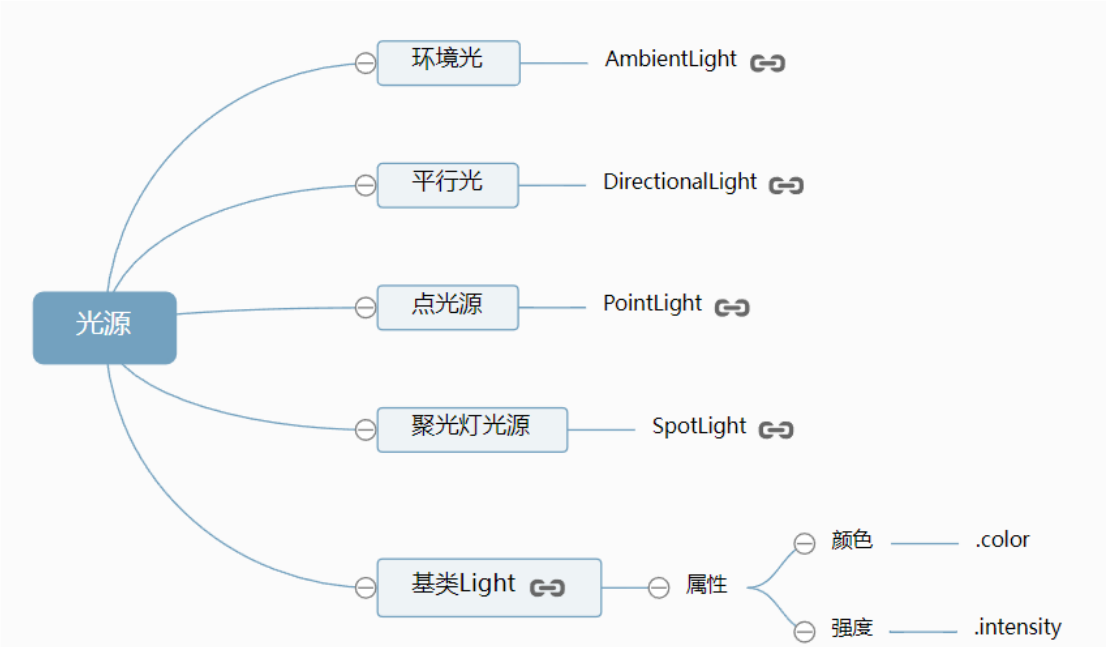
octree

OcTree 原理介绍

八叉树是一种用于描述三维空间的树状数据结构。八叉树的每个节点表示一个正方体的体积元素，每个节点有八个子节点，将八个子节点所表示的体积元素加在一起就等于父节点的体积。八叉树是四叉树在三维空间上的扩展，二维上我们有四个象限，而三维上，我们有 8 个卦限。八叉树主要用于空间划分和最近邻搜索。

我们使用它主要是为了存放我们的 glb 场景模型，这也是 octree 的一个一般用法，在 octree 中，我们可以方便的调用一些封装好的函数，比如碰撞检测函数，获取边界位置函数等

Light



环境光是没有特定方向的光源，主要是均匀整体改变 Threejs 物体表面的明暗效果，这一点和具有方向的光源不同，比如点光源可以让物体表面不同区域明暗程度不同。

平行光顾名思义光线平行，对于一个平面而言，平面不同区域接收到平行光的入射角一样。

点光源因为是向四周发散，所以设置好位置属性.position 就可以确定光线和物体表面的夹角，对于平行光而言,主要是确定光线的方向,光线方向设定好了，光线的与物体表面入射角就确定了，仅仅设置光线位置是不起作用的。

在三维空间中为了确定一条直线的方向只需要确定直线上两个点的坐标即可，所以 Threejs 平行光提供了位置.position 和目标.target 两个属性来一起确定平行光方向。目标.target 的属性值可以是 Threejs 场景中任何一个三维模型对象，比如一个网格模型 Mesh，这样 Threejs 计算平行光照射方向的时候，会通过自身位置属性.position 和.target 表示的物体的位置属性.position 计算出来。

Clock

计时对象，方便距离计算、刷新帧数

WebGLRenderer

Three.js 渲染器 WebGLRenderer 的.domElement 属性是 Three.js 执行渲染方法.render()的渲染结果，本质上就是一个 HTML 元素 Canvas，如果通过原生 WebGL 渲染一个场景，需

要手动创建一个 Cnavas 画布获得 WebGL 上下文，如果使用 Three.js 框架，Threejs 系统会自动创建一个 canvas 对象，然后把渲染结果呈现在该 Canvas 画布上。

渲染的范围就是直接通过 Three.js 的 WebGL 渲染器 WebGLRenderer 的.setSize()方法设置渲染尺寸为浏览器 body 区域宽高度。

pointerLockElement

用一句话说明 Pointer Lock API 的作用就是：Pointer Lock 可以让我们的的鼠标无限移动，脱离浏览器窗体的限制。这对于一些需要鼠标控制的应用非常有用。

现在，我们进入问题解答环节。

1 谁是主角

主角是我们的碰撞器，他有两个关键属性：一个是运动速度，一个是运动方向

这里没有太多需要介绍的，在后面调用这些属性和碰撞器时我们再详细说明。

2 如何渲染

Three.js 自带有渲染器，当然 webgl 自己也有，这里我们先创建一个场景，设置一下场景的背景颜色。

然后创建一个投影相机，并将投影相机的 camera.rotation.order = 'YXZ';保证摄像机正确旋转。当我在相机的局部 Y 轴上旋转时，它也会考虑局部 X 旋转，这是不必要的。
当设置为'ZYX'时，按照外界坐标系进行旋转。当设置为'XYZ'时，按照物体自身坐标系旋转

```
//初始化场景，
scene = new THREE.Scene();
scene.background = new THREE.Color( 0x54abef );

camera = new THREE.PerspectiveCamera( 60, window.innerWidth / w
camera.rotation.order = 'YXZ';

//由于这里才创建了scene，我们把cube在这里加入场景中
scene.add(cube);
```

我们注意到，其实在这里就创建了一个计时器和一些光照，我们这里使用的关照主要是两种，一个是环境光，一个是平行光，其中平行光为它设置一些附加属性，包括阴影，大小，距离等。

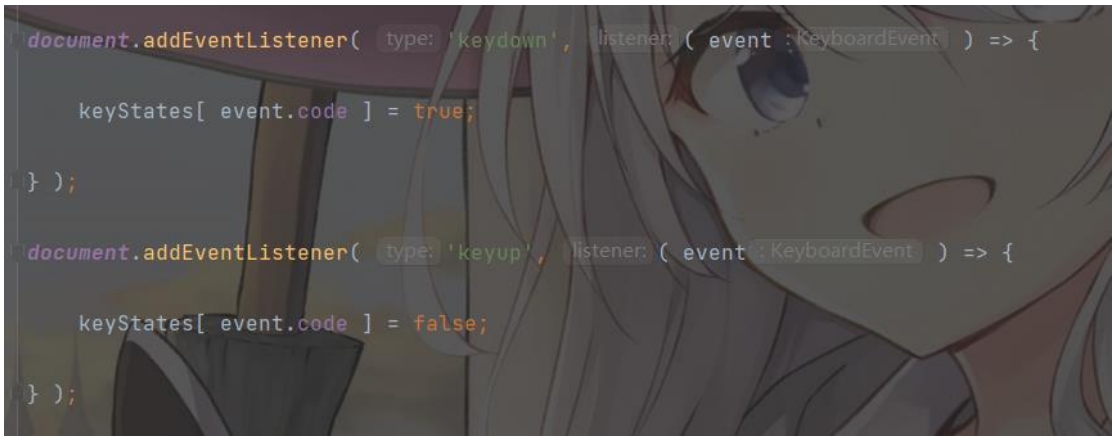
```
clock = new THREE.Clock();

ambientlight = new THREE.AmbientLight( 0x6688cc );

scene.add( ambientlight );

fillLight1 = new THREE.DirectionalLight( 0xff9999, 0.5 );
fillLight1.position.set( - 1, 1, 2 );
scene.add( fillLight1 );

fillLight2 = new THREE.DirectionalLight( 0x8888ff, 0.2 );
fillLight2.position.set( 0, - 1, 0 );
scene.add( fillLight2 );

directionalLight = new THREE.DirectionalLight( 0xffffaa, 1.2
directionalLight.position.set( - 5, 25, - 1 );
directionalLight.castShadow = true;
directionalLight.shadow.camera.near = 0.01;
directionalLight.shadow.camera.far = 500;
directionalLight.shadow.camera.right = 30;
directionalLight.shadow.camera.left = - 30;
directionalLight.shadow.camera.top  = 30;
directionalLight.shadow.camera.bottom = - 30;
directionalLight.shadow.mapSize.width = 1024;
directionalLight.shadow.mapSize.height = 1024;
directionalLight.shadow.radius = 4;
directionalLight.shadow.bias = - 0.00006;
scene.add( directionalLight );
```

最后我们使用 webglrenderer，该场景进行渲染，设置渲染大小为窗口大小，最后将 renderer 加入到容器（这个容器就是一个 dom 的容器元素，存放页面的所有元素）中，这也是 three.js 的一个常见做法。

```
renderer = new THREE.WebGLRenderer( parameters: { antialias: true } );
renderer.setPixelRatio( window.devicePixelRatio );
renderer.setSize( window.innerWidth, window.innerHeight );
renderer.shadowMap.enabled = true;
renderer.shadowMap.type = THREE.VSMShadowMap;

container = document.getElementById( elementId: 'container' );

container.appendChild( renderer.domElement );

//显示性能监测插件stats
stats = new Stats();
stats.domElement.style.position = 'absolute';
stats.domElement.style.top = '0px';

container.appendChild( stats.domElement );
```

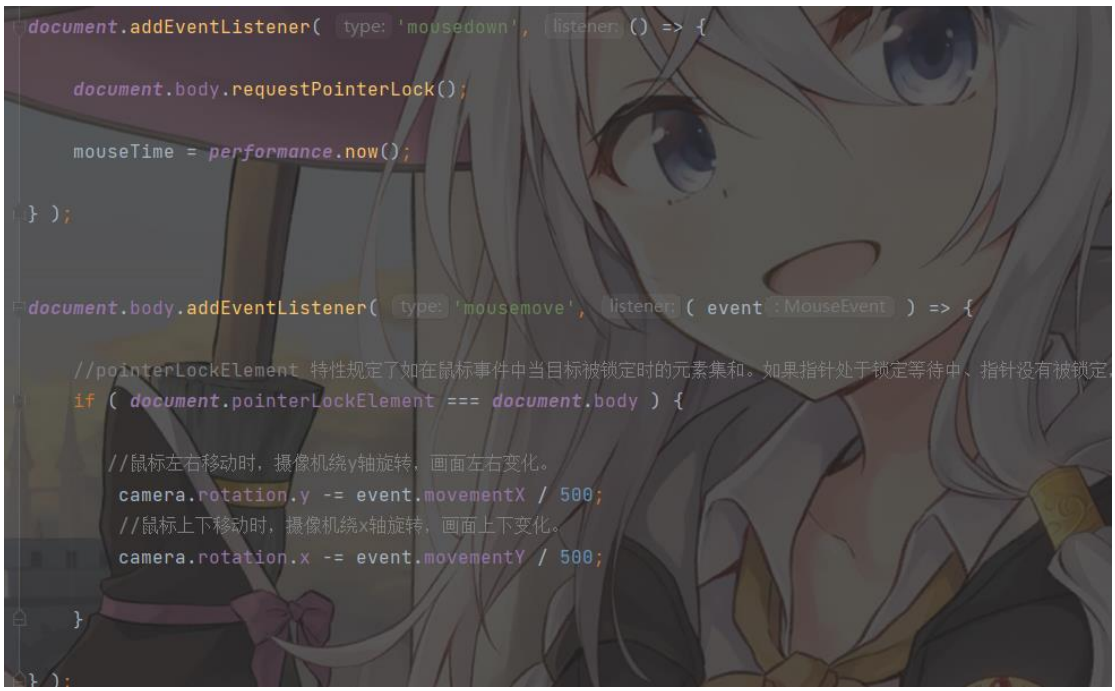
3 如何移动

我们使用 dom 监听方法，对鼠标和键盘敲击事件进行监听，监听的过程如下：
第一个是监听键盘是否被按下，并随时改变 keystate 的键值，为后面的监听判断做铺垫。



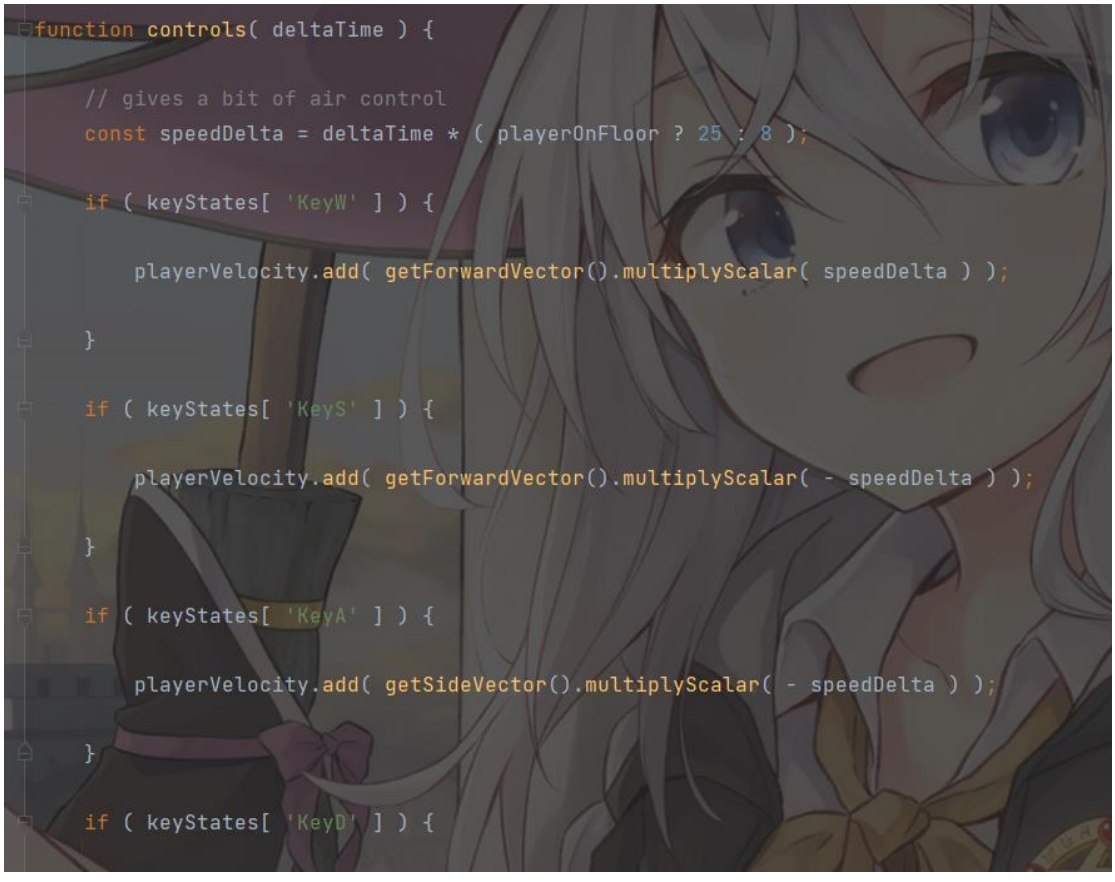
```
document.addEventListener( type: 'keydown', listener: ( event : KeyboardEvent ) => {

    keyStates[ event.code ] = true;

} );

document.addEventListener( type: 'keyup', listener: ( event : KeyboardEvent ) => {

    keyStates[ event.code ] = false;

} );
```

第二个是监听鼠标的点击事件和移动事件，前者是为了进入游戏，后者是为了移动视角
这里着重说明后者，移动视角实际上需要与相机进行交互，我们这里调用了前面说的 document.pointerLockElement，pointerLockElement 特性规定了如在鼠标事件中当目标被锁定时的元素集和。如果指针处于锁定等待中、指针没有被锁定，或者目标在另外一个文档中这几种情况，返回的值 null。这样说可能不太清楚，具体的功能还是参照我前面所说的通俗翻译：Pointer Lock 可以让我们的的鼠标无限移动，脱离浏览器窗体的限制。



```
document.addEventListener( type: 'mousedown', listener: () => {

    document.body.requestPointerLock();

    mouseTime = performance.now();

} );

document.body.addEventListener( type: 'mousemove', listener: ( event : MouseEvent ) => {

    //pointerLockElement 特性规定了如在鼠标事件中当目标被锁定时的元素集和。如果指针处于锁定等待中、指针没有被锁定
    if ( document.pointerLockElement === document.body ) {

        //鼠标左右移动时，摄像机绕y轴旋转，画面左右变化。
        camera.rotation.y -= event.movementX / 500;
        //鼠标上下移动时，摄像机绕x轴旋转，画面上下变化。
        camera.rotation.x -= event.movementY / 500;

    }

} );
```

最后是监听各个键盘按键，为碰撞体速度和方向赋值，这个下面细讲。



```
function controls( deltaTime ) {

    // gives a bit of air control
    const speedDelta = deltaTime * ( playerOnFloor ? 25 : 8 );

    if ( keyStates[ 'KeyW' ] ) {

        playerVelocity.add( getForwardVector().multiplyScalar( speedDelta ) );

    }

    if ( keyStates[ 'KeyS' ] ) {

        playerVelocity.add( getForwardVector().multiplyScalar( - speedDelta ) );

    }

    if ( keyStates[ 'KeyA' ] ) {

        playerVelocity.add( getSideVector().multiplyScalar( - speedDelta ) );

    }

    if ( keyStates[ 'KeyD' ] ) {
```

接着我们定义两个移动方向确认的函数，根据相机的方向，确定 playerDirection 属性的值，如果是前后移动，那就是该向量，如果不是，而是左右移动，还要求出它的法向量，前进后退向量对应的法向量自然就是左右平移。

```
function getForwardVector() {

    camera.getWorldDirection( playerDirection );
    playerDirection.y = 0;
    playerDirection.normalize();
    //复制得到相机方向，向着相机方向移动
    return playerDirection;

}

function getSideVector() {

    camera.getWorldDirection( playerDirection );
    playerDirection.y = 0;
    playerDirection.normalize();
    playerDirection.cross( camera.up );
    //进行一个变换，得到与相机法向量相互垂直的另一个向量，方向向着这个相连
    return playerDirection;

}
```

这里的更新函数就是我们真正实现移动的函数，可以看到，他调用了上面被赋值后的playerVelocity，再根据前面计时器计时的时间，算出移动距离，加入到碰撞体的下降和平移之中，最后用 translate 函数对碰撞体进行移动。

同时在下降中模拟量空气阻力，这里的空气阻力计算和参数设置我参考了网上的资料，使物体的移动不显得突兀和僵硬。

```
function updatePlayer( deltaTime ) {

    let damping = Math.exp( x: - 4 * deltaTime ) - 1;

    if ( ! playerOnFloor ) {

        //下落速度等于加速度*时间
        playerVelocity.y -= GRAVITY * deltaTime;


        // 空气阻力
        damping *= 0.1;

    }

    //该方法是可以平滑的从A逐渐移动到B点，考虑空气阻力，可以控制速度，最常见的用法是相机跟随目标。
    playerVelocity.addScaledVector( playerVelocity, damping );



    //速度乘时间得到物体最终位置
    const deltaPosition = playerVelocity.clone().multiplyScalar( deltaTime );
    //移动到物体最终位置
    playerCollider.translate( deltaPosition );
```

当然在最后别忘了更新相机的位置，使其追随碰撞体

```
    //相机跟随物体
    camera.position.copy( playerCollider.end );
```

4 如何判断碰撞

在这里，我们调用的是前面新建的八叉树，由于已经把场景模型存放到了八叉树当中，我们直接调用八叉树的碰撞检测函数，该函数的返回值是一个对象，该对象可以获得场景的发生碰撞边界坐标，因此我们在这里限制死碰撞器坐标为该边界坐标，这样就不会发生穿模现象。

小结

以上所有就是项目的全部内容，总的来说，three.js 给我的使用体验也是不错的，这次的 webgl 项目也使得我学到了不少的知识。

参考资料

【1】javascript - Three.js OrbitControls how to stop gradually and smoothly, instead of instantly when panning? - Stack Overflow
【2】(12 条消息) three.js 学习笔记之 混乱的矩阵_Calculon 的专栏-CSDN 博客
【3】(12 条消息) Unity3D Player 角色移动控制脚本——转载_失恋王子的博客-CSDN 博客
【4】Three.js 教程 (webgl3d.cn)