**基于WebGL及three.js库的推球游戏demo**

目录

[**一、** **环境基础** 2](#_Toc28621743)

[**二、** **实现功能** 2](#_Toc28621744)

[**1.** **生成场景和相机** 2](#_Toc28621745)

[**2.** **基本体素的建模表达** 2](#_Toc28621746)

[**3.** **三维模型导入功能** 2](#_Toc28621747)

[**4.** **基本材质、纹理的显示和编辑** 3](#_Toc28621748)

[**6.** **基本光照模型** 5](#_Toc28621749)

[**7.** **场景漫游** 5](#_Toc28621750)

[**8.** **动画播放** 5](#_Toc28621751)

[**9.** **屏幕截取** 11](#_Toc28621752)

[**10.** **物理引擎初始化** 11](#_Toc28621753)

[**11.** **物理模型建立** 12](#_Toc28621754)

[**12.** **碰撞检测** 15](#_Toc28621755)

[**三、** **操作说明和效果展示** 17](#_Toc28621756)

[**四、** **组员与分工** 19](#_Toc28621757)

[**1.** **组员信息** 19](#_Toc28621758)

[**2.** **分工** 20](#_Toc28621759)

[**五、** **附件** 20](#_Toc28621760)

1. **环境基础**
2. 实现平台

采用HTML5的移动平台实现。

1. 编程环境

采用WebGL进行场景的构建，使用了three.js这一利用JavaScript编写的WebGL第三方库，其中提供了非常多的3D显示功能。同时使用ammo.js作为物理引擎。

1. **实现功能**
2. **生成场景和相机**

新声明场景变量scene调用Scene()函数，再利用CubeTextureLoader()函数读取场景的背景。

***scene*** = new THREE.Scene();

//CubeTexture  
var cubeLoader = new THREE.CubeTextureLoader();  
cubeLoader.setPath( 'textures/cube/skyboxsun25deg/' );  
***scene***.background = cubeLoader.load( [  
 'px.jpg', 'nx.jpg',  
 'py.jpg', 'ny.jpg',  
 'pz.jpg', 'nz.jpg'  
] );

用PerspectiveCamera()函数构建相机，通过参数确定相机视野张角，屏幕长宽，视野远近距离，再设置相机位置和相机正方向。

***camera*** = new THREE.PerspectiveCamera( 30, ***window***.innerWidth / ***window***.innerHeight,

1, 10000 );  
***camera***.position.set( 0, 6, 12 );  
***camera***.lookAt( 0, 1, 0 );

1. **基本体素的建模表达**

新声明变量网格模型mesh以及材质material，根据所需要的几何体调用类型不同的几何体构建函数，如球体使用SphereBufferGeometry()，参数分别为ballRadius球的半径，球的纬度和经度（关系到球的面数量），最后使用Mesh(geometry,material)构成网格模型：

var ball = new THREE.Mesh( new THREE.SphereBufferGeometry( ballRadius, 20, 20 ),

ballMaterial );

1. **三维模型导入功能**

新声明模型读取器loader和模型model，按类型可以分为OBJLoader()和GLTFLoader()等，读取器调用load函数，从url中读取，设定位置，再添加到scene中。以下为GLTF模型的导入：

var loader = new GLTFLoader();

loader.load( 'models/gltf/Soldier.glb', function ( gltf ) {  
 ***model*** = gltf.scene;  
 ***model***.position.set(0,0,0);  
 ***scene***.add( ***model*** );

1. **基本材质、纹理的显示和编辑**

新声明材质变量material，使用函数THREE.MeshPhongMaterial编辑颜色或者添加纹理。

var ballMaterial = new THREE.MeshPhongMaterial( { color: 0x666666 } );

纹理需要声明纹理读取器textureLoader为THREE.TextureLoader()类型，利用函数.load(“url”,function(map))读取纹理，map参数设置XY方向上的重复,添加到材质中。

var material1 = new THREE.MeshStandardMaterial( {

roughness: 0.7,  
 color: 0xffffff,  
 bumpScale: 0.002,  
 metalness: 0.2  
} );

//加载法线贴图

***textureLoader***.load( "textures/brick\_diffuse.jpg", function ( map ) {  
 map.wrapS = THREE.***RepeatWrapping***;  
 map.wrapT = THREE.***RepeatWrapping***;  
 map.anisotropy = 4;  
 map.repeat.set( 19.5, 1.5 );  
 material1.map = map;  
 material1.needsUpdate = true;  
} );

//加载凹凸贴图  
***textureLoader***.load( "textures/brick\_bump.jpg", function ( map ) {  
 map.wrapS = THREE.***RepeatWrapping***;  
 map.wrapT = THREE.***RepeatWrapping***;  
 map.anisotropy = 4;  
 map.repeat.set( 19.5, 1.5 );  
 material1.bumpMap = map;  
 material1.needsUpdate = true;  
} );

也可以加入镜子实现反射效果，使用PlaneBufferGeometry()新建平面几何体，在利用Reflector()做出镜面效果，需要Reflector.js：

var geometry0 = new THREE.PlaneBufferGeometry(30,40);

var groundMirror = new Reflector( geometry0, {  
 clipBias: 0.003,  
 textureWidth: ***window***.innerWidth \* ***window***.devicePixelRatio,  
 textureHeight: ***window***.innerHeight \* ***window***.devicePixelRatio,  
 color: 0x777777,  
 recursion: 1  
} );  
groundMirror.position.y = 0.001;  
groundMirror.position.x = 15;  
groundMirror.rotateX( - ***Math***.PI / 2 );  
***scene***.add( groundMirror );

1. **基本几何变化功能**

主要体现在人物的运动中,人物前进是平移与动画的结合，转向是平移，旋转与动画三者的结合，几何变化主要采用坐标的变换，根据所需情况利用三角函数计算坐标的变化量，同时相机保持跟随。

//步长与运动状态权重相结合，以确保动画不停止，人物移动也不停止

*step*=0.02\*(*walkWeight*+*runWeight*\*3);

//如果按住W或者走路或奔跑的动作还没有停止，并且没有按S就让人物模型往前移动  
if ((*moveForward*||*walkWeight*!==0||*runWeight*!==0)&&!*moveBackward*) {  
 //边界检测，检测下一步会不会越界，如果越界就不往前走  
 if(*Math*.abs(*model*.position.z - *step* \* *Math*.cos(*model*.rotation.y))<=(*groundwidth*-3)/2){  
 *model*.position.z -= *step* \* *Math*.cos(*model*.rotation.y);  
 *camera*.position.z -= *step* \* *Math*.cos(*model*.rotation.y);  
 }  
  
 //边界检测，同上  
 if(*Math*.abs(*model*.position.x - *step* \* *Math*.cos(*model*.rotation.y))<=(*groundlong*-3)/2){  
 *model*.position.x -= *step* \* *Math*.sin(*model*.rotation.y);  
 *camera*.position.x -= *step* \* *Math*.sin(*model*.rotation.y);  
 }  
}  
  
//如果进行转向，改变模型的朝向，即按Y轴旋转模型  
if (*turnLeft*||*turnRight*) {  
 *model*.rotation.y += (*turnLeft*-*turnRight*)\*0.01;  
 *camera*.position.z = 12 \* *Math*.cos(*model*.rotation.y) + *model*.position.z;  
 *camera*.position.x = 12 \* *Math*.sin(*model*.rotation.y) + *model*.position.x;  
 *camera*.lookAt(*model*.position);  
}  
  
//只有按住S才能触发往后走的操作，区别于往前走，以保证动画的连续不卡顿  
if(*moveBackward*){  
 //边界检测  
 if(*Math*.abs(*model*.position.z + *step* \* *Math*.cos(*model*.rotation.y))<=(*groundwidth*-3)/2){  
 *model*.position.z += *step* \* *Math*.cos(*model*.rotation.y);  
 *camera*.position.z += *step* \* *Math*.cos(*model*.rotation.y);  
 }  
  
 if(*Math*.abs(*model*.position.x + *step* \* *Math*.cos(*model*.rotation.y))<=(*groundlong*-3)/2){  
 *model*.position.x += *step* \* *Math*.sin(*model*.rotation.y);  
 *camera*.position.x += *step* \* *Math*.sin(*model*.rotation.y);  
 }  
}

1. **基本光照模型**

环境光使用AmbientLight(color)函数实现，可以直接添加到场景中：

***scene***.add( new THREE.AmbientLight( 0x666666 ) );

可以添加半球光，光源位于场景上方，两种颜色从天空到地面渐变，使用HemisphereLight(skyColor, groundColor)函数：

var hemiLight = new THREE.HemisphereLight( 0xffffff, 0x444444 );

hemiLight.position.set( 0, 20, 0 );  
***scene***.add( hemiLight );

方向光用于产生阴影，使用DirectionalLight(color)函数，并指定光源位置和朝向，开启产生阴影：

var dirLight = new THREE.DirectionalLight( 0xffffff );

dirLight.position.set( - 100, 100, - 100 );  
dirLight.castShadow = true;

***scene***.add( dirLight );

1. **场景漫游**

漫游用到了控件OrbitControls(camera, renderer.domElement)，它自带有鼠标左键的定点旋转相机和右键的平移相机，以及滚轮的缩放功能。我们给该控件添加target，绑定模型的位置，实时更新，使其右键实现同时平移人物和相机的位置。

***controls*** = new OrbitControls( ***camera***, ***renderer***.domElement );

***controls***.update();

***controls***.target=***model***.position;  
***controls***.update();

1. **动画播放**

Three.js提供了THREE.AnimationMixer这一函数进行动画混合，gltf格式

的三维模型自带gltf.animations的骨骼动画。

在创建动作前，首先定义几个变量：

var *idleAction*, *walkAction*, *runAction*;//三个动作状态，分别表示原地、走路、

跑步  
var *idleWeight*, *walkWeight*, *runWeight*; //三个动作的权重，默认为（10 0.01）代表原地停留  
 //（0，1，0）代表走路，（0，0，1）代表跑步  
var *actions*, *step*; //分别为模型的动作（数组）、前进的步长（控制移动速度）  
var *moveForward* = false; //四种移动的状态

var *moveBackward* = false;  
var *turnLeft* = false;  
var *turnRight* = false;  
var *quickMove* = false;

首先初始化人物模型的动作：

//模型建立

var loader = new *GLTFLoader*();  
loader.load( 'models/gltf/Soldier.glb', function ( *gltf* ) {  
  
 *model* = *gltf*.scene;  
 *model*.position.set(0,0,0);  
 *model*.traverse(function ( *child* ) {  
  
 *child*.castShadow = true;  
  
 } );  
 *scene*.add( *model* );  
 dirLight.target=*model*; //平行光跟随人物模型移动，以保证阴影不会消失  
 var animations = *gltf*.animations; //读取模型的动画骨骼  
  
 *mixer* = new THREE.AnimationMixer( *model* ); //动画混合器  
  
 *idleAction* = *mixer*.clipAction( animations[ 0 ] ); //返回传递的剪辑的动画混合器  
 *walkAction* = *mixer*.clipAction( animations[ 3 ] );  
 *runAction* = *mixer*.clipAction( animations[ 1 ] );  
  
 *actions* = [ *idleAction*, *walkAction*, *runAction* ];  
  
 activateAllActions(); //让所有动作都动起来  
 animate();  
});

创建动作控制的相关函数：

///动作控制

//启用所有动作

function activateAllActions() {  
  
 setWeight( *idleAction*, 1 ); //初始化三种动作的权重  
 setWeight( *walkAction*, 0 );  
 setWeight( *runAction*, 0.01 );  
  
 *actions*.forEach( function ( *action* ) {  
  
 *action*.play(); //播放动画  
  
 } );  
  
}  
  
//改变动作的权重  
function setWeight( *action*, *weight* ) {  
 // 这个函数是必需的，因为animationAction.crossFadeTo（）默认禁用了它的开始操作和设置  
 *action*.enabled = true;  
 //开始动作的时间刻度为（（开始动画的持续时间）/（结束动画的持续时间））  
 *action*.setEffectiveTimeScale( 1 );  
 *action*.setEffectiveWeight( *weight* );  
  
}  
  
//动作转换判断函数，包含三个参数（开始的动作，结束的动作，中间经过的时长）  
function prepareCrossFade( *startAction*, *endAction*, *defaultDuration* ) {  
  
 // 确保所有动作都不停止  
 unPauseAllActions();  
  
 // 如果当前操作为“空闲”（持续4秒），则立即执行动作转换；  
 //否则，等待当前操作完成其当前循环  
 if ( *startAction* === *idleAction* ) {  
 executeCrossFade( *startAction*, *endAction*, *defaultDuration* );  
 } else {  
 synchronizeCrossFade( *startAction*, *endAction*, *defaultDuration* );  
 }  
}  
  
//不停止所有动作  
function unPauseAllActions() {  
 *actions*.forEach( function ( *action* ) {  
 *action*.paused = false;  
 } );  
}  
  
//动作淡入淡出  
function executeCrossFade( *startAction*, *endAction*, *duration* ) {  
 // 不仅开始动作，而且结束动作在衰减前必须获得1的权重  
 //（关于开始动作，这里已经保证了它的逻辑正确）  
 setWeight( *endAction*, 1 );  
 *endAction*.time = 0;  
  
 // 淡入操作来开启动作，也可以通过将第三个参数设置为false来尝试不进行变形  
 *startAction*.crossFadeTo( *endAction*, *duration*, true );  
}  
  
//动作转换  
function synchronizeCrossFade( *startAction*, *endAction*, *duration* ) {  
 //在当前动作结束后，将新的动作转换加入队列  
 *mixer*.addEventListener( 'loop', onLoopFinished );  
   
 function onLoopFinished( *event* ) {  
 if ( *event*.action === *startAction* ) {  
 //移除先前的动作，加入新的动作并执行动作淡入淡出操作  
 *mixer*.removeEventListener( 'loop', onLoopFinished );  
 executeCrossFade( *startAction*, *endAction*, *duration* );  
 }  
 }  
}

按键响应事件：

//鼠标按下的事件

var onKeyDown = function ( *event* ) {  
  
 switch ( *event*.keyCode ) {  
  
 case 87: // w  
 if(*moveForward*===false){ //只执行一次赋值以及动作播放  
 *moveForward* = true;  
 //将作动从原地转换为走路状态，中间的事件由当前的走路权重决定，以防止快速连续按键下的动作抽搐  
 prepareCrossFade( *idleAction*, *walkAction*, 1.0-*walkWeight* );  
 }  
 break;  
  
 case 65: // a  
 if(!*turnLeft*){  
 *turnLeft* = true;  
 if(!*moveForward*)prepareCrossFade( *idleAction*, *walkAction*, 0.2\*(1.0-*walkWeight*));  
 }  
  
 break;  
  
 case 83: // s  
 if(*moveBackward*===false){  
 *moveBackward* = true;  
 prepareCrossFade( *idleAction*, *walkAction*, 1.0-*walkWeight* );  
 }  
 break;  
  
 case 68: // d  
 if(*turnRight*===false){  
 *turnRight* = true;  
 if(!*moveForward*)prepareCrossFade( *idleAction*, *walkAction*, 0.2\*(1.0-*walkWeight*));  
 }  
 break;  
  
 case 32: //quickMove  
 //space  
 //在按了前进键的同时，按冲刺键才是有效的  
 if(*walkWeight* && *quickMove* ===false){  
 *quickMove* = true;  
 prepareCrossFade( *walkAction*, *runAction*, 1.0-*runWeight* );  
 }  
 break;  
 }  
  
};  
  
//鼠标松开事件  
var onKeyUp = function ( *event* ) {  
  
 switch ( *event*.keyCode ) {  
  
 case 87: // w  
 *moveForward* = false;  
 if(*quickMove*){  
 prepareCrossFade( *runAction*, *walkAction*, 0.8\**walkWeight*);  
 prepareCrossFade( *walkAction*, *idleAction*, 0.8\**walkWeight* );  
  
 }  
 else prepareCrossFade( *walkAction*, *idleAction*, 0.8\**walkWeight* );  
 break;  
  
 case 65: // a  
 *turnLeft* = false;  
 if(!*moveForward*)prepareCrossFade( *walkAction*, *idleAction*, 0.2 );  
 break;  
  
 case 83: // s  
 *moveBackward* = false;  
 //为防止后退与前进动画的冲突，在此直接设置权重，让人物快速停下来。  
 //移动控制中认为只要walkAction这一变量值不为零，就继续向前移动，以匹配动画效果  
 setWeight(*idleAction*,1);  
 setWeight(*walkAction*,0);  
 break;  
  
 case 68: // d  
 *turnRight* = false;  
 if(!*moveForward*)prepareCrossFade( *walkAction*, *idleAction*, 0.2\**walkWeight* );  
 break;  
  
 case 32: //quickMove  
 //space  
 *quickMove* = false;  
 prepareCrossFade( *runAction*, *walkAction*, 0.8\**runWeight* );  
 break;  
  
 case 80:  
 *renderer*.render(*scene*, *camera*);  
 let imgData = *renderer*.domElement.toDataURL("image/jpeg");//这里可以选择png格式jpeg格式  
 debugBase64(imgData);  
 break;  
 }  
  
};  
  
//将鼠标响应事件加入事件列表  
*document*.addEventListener( 'keydown', onKeyDown, false );  
*document*.addEventListener( 'keyup', onKeyUp, false );

在animate()函数中更新动画：

//更新动画

if ( *mixer* ) *mixer*.update( delta );

1. **屏幕截取**

键盘P键按下添加事件响应，渲染一帧的画面，从渲染器中读取图片的url，调用函数将url转换成64位并新打开页面将url中图片展示。

case 80:

***renderer***.render(***scene***, ***camera***);  
 let imgData = ***renderer***.domElement.toDataURL("image/jpeg");//这里可以选择png格式jpeg格式  
 debugBase64(imgData);  
 break;

function debugBase64(base64URL){  
 let win = ***window***.open();  
 win.document.write('<image id="IframeReportImg" src="' + base64URL + '" frameborder="0" style="border:0; top:0px; left:0px; bottom:0px; right:0px; width:100%; height:100%;" allowfullscreen ></image>');  
}

1. **物理引擎初始化**

对于物理引擎需要的全局变量进行初始化，

var ***gravityConstant*** = 9.8;

var ***collisionConfiguration***;  
var ***dispatcher***;  
var ***broadphase***;  
var ***solver***;  
var ***physicsWorld***;  
var ***margin*** = 0.05;

var ***rigidBodies*** = [];

其中***gravityConstant***变量表示加速度，***margin***表示的是物理模型边缘的容错参数，

物理引擎环境的配置，包括生成碰撞配置、碰撞解决参数、物理世界等变量，以及物理世界的加速度等参数配置

function initPhysics() {

// Physics configuration  
 ***collisionConfiguration*** = new ***Ammo***.btDefaultCollisionConfiguration();  
 ***dispatcher*** = new ***Ammo***.btCollisionDispatcher( ***collisionConfiguration*** );  
 ***broadphase*** = new ***Ammo***.btDbvtBroadphase();  
 ***solver*** = new ***Ammo***.btSequentialImpulseConstraintSolver();  
 ***physicsWorld*** = new ***Ammo***.btDiscreteDynamicsWorld( ***dispatcher***, ***broadphase***, ***solver***, ***collisionConfiguration*** );  
 ***physicsWorld***.setGravity( new ***Ammo***.btVector3( 0, - ***gravityConstant***, 0 ) );  
  
 ***transformAux1*** = new ***Ammo***.btTransform();  
 ***tempBtVec3\_1*** = new ***Ammo***.btVector3( 0, 0, 0 );  
  
}

1. **物理模型建立**

首先，编写一个建立物理刚性模型以及将物理模型和显示模型相关联的函数createRigidBody()，

function createRigidBody( object, physicsShape, mass, pos, quat, vel,

angVel ) {

通过输入物体的显示模型、物理模型形状、质量、位置（position）、四元数（quatertion，表示方向）来建立物理空间的物理模型和显示空间的显示模型的相关性，

接下来调用并确认位置和方向信息，

if ( pos ) {

object.position.copy( pos );  
} else {  
 pos = object.position;  
}  
if ( quat ) {  
 object.quaternion.copy( quat );  
} else {  
 quat = object.quaternion;  
}

之后调用初始参数先进行转换，

将位置和方向信息加入transform变量，生成物体在物理世界的位置方向的运动状态

var transform = new ***Ammo***.btTransform();

transform.setIdentity();  
transform.setOrigin( new ***Ammo***.btVector3( pos.x, pos.y, pos.z ) );  
transform.setRotation( new ***Ammo***.btQuaternion( quat.x, quat.y, quat.z, quat.w ) );  
var motionState = new ***Ammo***.btDefaultMotionState( transform );

对物体的形状和质量进行关联，

var localInertia = new ***Ammo***.btVector3( 0, 0, 0 );

physicsShape.calculateLocalInertia( mass, localInertia );

对转换过的运动状态参数和物体本身的参数进行结合，并生成变量body

var rbInfo = new ***Ammo***.btRigidBodyConstructionInfo( mass, motionState,

physicsShape, localInertia );  
var body = new ***Ammo***.btRigidBody( rbInfo );

关联显示模型object和物理模型body

object.userData.physicsBody = body;

在显示世界总加载显示模型object，

***scene***.add( object );

当物体有的质量mass大于0时，将其物理模型设置为可以移动变化的单位，

同时，当质量为0时，物体的物理模型在物理空间为固定位置方向，

if ( mass > 0 ) {

***rigidBodies***.push( object );  
  
 // Disable deactivation  
 body.setActivationState( 4 );  
}

在物理空间加载物理模型，

***physicsWorld***.addRigidBody( body );

之后调用之前的函数createRigidBody()编写一个生成简单立方体的物理模型和显示模型生成函数createParalellepipedWithPhysics()，

function createParalellepipedWithPhysics( sx, sy, sz, mass, pos, quat,

material ) {  
  
 var object = new THREE.Mesh( new THREE.BoxBufferGeometry( sx, sy, sz, 1, 1, 1 ), material );  
 var shape = new ***Ammo***.btBoxShape( new ***Ammo***.btVector3( sx \* 0.5, sy \* 0.5, sz \* 0.5 ) );  
 shape.setMargin( ***margin*** );  
  
 createRigidBody( object, shape, mass, pos, quat );  
  
 return object;  
  
}

通过函数Three.Mesh建立显示长方体，通过函数Ammo.btBoxShape建立从中心开始的立方体物理形状，调用函数createParalellepipedWithPhysics()建立长方体的物理模型以及和显示模型的相关联，

建立基本物体以及它们的物理模型，

设置地面的初始位置以及方位，

***pos***.set( 0, - 0.5, 0 );

***quat***.set( 0, 0, 0, 1 );

pos为position，为位置函数，quat为quaternion四元数，用来表示方向，

之后调用函数createParalellepipedWithPhysics()去建立显示模型和物理模型，

var ground = createParalellepipedWithPhysics( ***groundlong***, 1, ***groundwidth***,

0, ***pos***, ***quat***, new THREE.MeshPhongMaterial( { color: 0x666666 } ) );

三面墙的模型的建立，

设定初始值墙高wallheight，

var wallheight = 3;

使用关于地面长宽的全局变量来设定墙面的位置、方向以及形状，同时调用函数createParalellepipedWithPhysics()去建立墙面的显示模型和物理模型，

***pos***.set( 0, wallheight/2, (***groundwidth***-1)/2 );

***quat***.set( 0, 0, 0, 1 );  
var wall1 = createParalellepipedWithPhysics( ***groundlong***, wallheight, 1, 0, ***pos***, ***quat***, material1 );

设定墙面可以接收其他物体的阴影，

wall1.receiveShadow = true;

同理来制作其他两面墙，

球体模型的建立，

由于要在场景中生成多个球体，使用for循环函数设置20个球体，球体的位置使用函数random随机生成，

设置球体的质量、半径、材质颜色，

var ballMass = 0.02;

var ballRadius = 0.4;  
var ballMaterial = new THREE.MeshPhongMaterial( { color: 0x666666 } );

生成模型，

var ball = new THREE.Mesh( new THREE.SphereBufferGeometry( ballRadius, 20,

20 ), ballMaterial );

设置球体的产生阴影和接收阴影，

ball.castShadow = true;

ball.receiveShadow = true;

生成物理形状，

var ballShape = new ***Ammo***.btSphereShape( ballRadius );

通过随机函数生成各个球体位置以及建立物理模型并与显示模型相关联，

***pos***.set( (***Math***.random()-0.5)\*(***groundlong***-2), (***Math***.random())\*100,

(***Math***.random()-0.5)\*(***groundwidth***-2) );  
***quat***.set( 0, 0, 0, 1 );  
createRigidBody( ball, ballShape, ballMass, ***pos***, ***quat*** );

由于人物模型较为复杂，在物理世界使用圆柱体来代表物理模型，这就需要生成一个新的物理模型，并与人物模型同步，

具体代码类似上述的立方体、球体的物理模型生成，

其中，使用函数CylinderGeometry()来生成圆柱体，

var manMass = 0;

***man*** = new THREE.Mesh( new THREE.CylinderGeometry(0.5, 0.5 ,2 ,40 ,40), new THREE.MeshPhongMaterial( { color: 0x202020 } ) );  
***man***.visible=false;  
***pos***.set( 0, 1, 0 );  
***quat***.set( 0, 0, 0, 1);  
***man***.castShadow = true;  
***man***.receiveShadow = true;  
var manShape = new ***Ammo***.btSphereShape( 0.5 );  
manShape.setMargin( ***margin***);  
createRigidBody( ***man***, manShape, manMass, ***pos***, ***quat*** );  
***man***.userData.physicsBody.setFriction( 0.2 );

1. **碰撞检测**

物理碰撞的实现就是实时更新物理模型的位置并根据它们的运动情况和物理模型进行检测是否碰撞，

需要编写一个新的更新物理世界的函数updatePhysics( deltaTime )，根据设定的计时器deltaTime来运行，

function updatePhysics( deltaTime ) {

首先，加载物理世界，

***physicsWorld***.stepSimulation( deltaTime, 10 );

更新人物的物理模型（圆柱体），

var softBody = ***man***.userData.physicsBody;

var ns=softBody.getMotionState();  
ns.getWorldTransform( ***transformAux1*** );  
***transformAux1***.setOrigin( new ***Ammo***.btVector3( ***model***.position.x, ***model***.position.y, ***model***.position.z ) );  
ns.setWorldTransform(***transformAux1***);  
softBody.setMotionState(ns);

通过数组更新rigidBodies调用并其他物理世界中的刚性物理模型（实验中只有20个刚性球体），

for ( var i = 0, il = ***rigidBodies***.length; i < il; i ++ ) {

var objThree = ***rigidBodies***[ i ];  
 var objPhys = objThree.userData.physicsBody;  
 var ms = objPhys.getMotionState();  
  
 if ( ms ) {  
  
 ms.getWorldTransform( ***transformAux1*** );  
 var p = ***transformAux1***.getOrigin();  
 var q = ***transformAux1***.getRotation();  
 objThree.position.set( p.x(), p.y(), p.z() );  
 objThree.quaternion.set( q.x(), q.y(), q.z(), q.w() );  
  
 objThree.userData.collided = false;  
 }  
}

通过dispatcher来确认所以物理模型，

for ( var i = 0, il = ***dispatcher***.getNumManifolds(); i < il; i ++ ) {

通过检验所遍历的物理模型的状态来确认需不需要对该物理模型进行碰撞改变（碰撞检测的前置条件），

var contactManifold = ***dispatcher***.getManifoldByIndexInternal( i );

var rb0 = ***Ammo***.castObject( contactManifold.getBody0(), ***Ammo***.btRigidBody );  
var rb1 = ***Ammo***.castObject( contactManifold.getBody1(), ***Ammo***.btRigidBody );  
  
var threeObject0 = ***Ammo***.castObject( rb0.getUserPointer(), ***Ammo***.btVector3 ).threeObject;  
var threeObject1 = ***Ammo***.castObject( rb1.getUserPointer(), ***Ammo***.btVector3 ).threeObject;  
  
if ( ! threeObject0 && ! threeObject1 ) {  
  
 continue;  
  
}  
  
var userData0 = threeObject0 ? threeObject0.userData : null;  
var userData1 = threeObject1 ? threeObject1.userData : null;  
  
var collided0 = userData0 ? userData0.collided : false;  
var collided1 = userData1 ? userData1.collided : false;  
  
if ( collided0 && collided1 ) {  
  
 continue;  
  
}

若通过检测则进行碰撞变化，

var contact = false;

var maxImpulse = 0;  
for ( var j = 0, jl = contactManifold.getNumContacts(); j < jl; j ++ ) {  
  
 var contactPoint = contactManifold.getContactPoint( j );  
  
 if ( contactPoint.getDistance() < 0 ) {  
  
 contact = true;  
 var impulse = contactPoint.getAppliedImpulse();  
  
 if ( impulse > maxImpulse ) {  
  
 maxImpulse = impulse;  
 var pos = contactPoint.get\_m\_positionWorldOnB();  
 var normal = contactPoint.get\_m\_normalWorldOnB();  
 ***impactPoint***.set( pos.x(), pos.y(), pos.z() );  
 ***impactNormal***.set( normal.x(), normal.y(), normal.z() );  
  
 }  
 break;  
 }  
}

在碰撞之后对于改变运动状态的物理模型以及其显示模型，

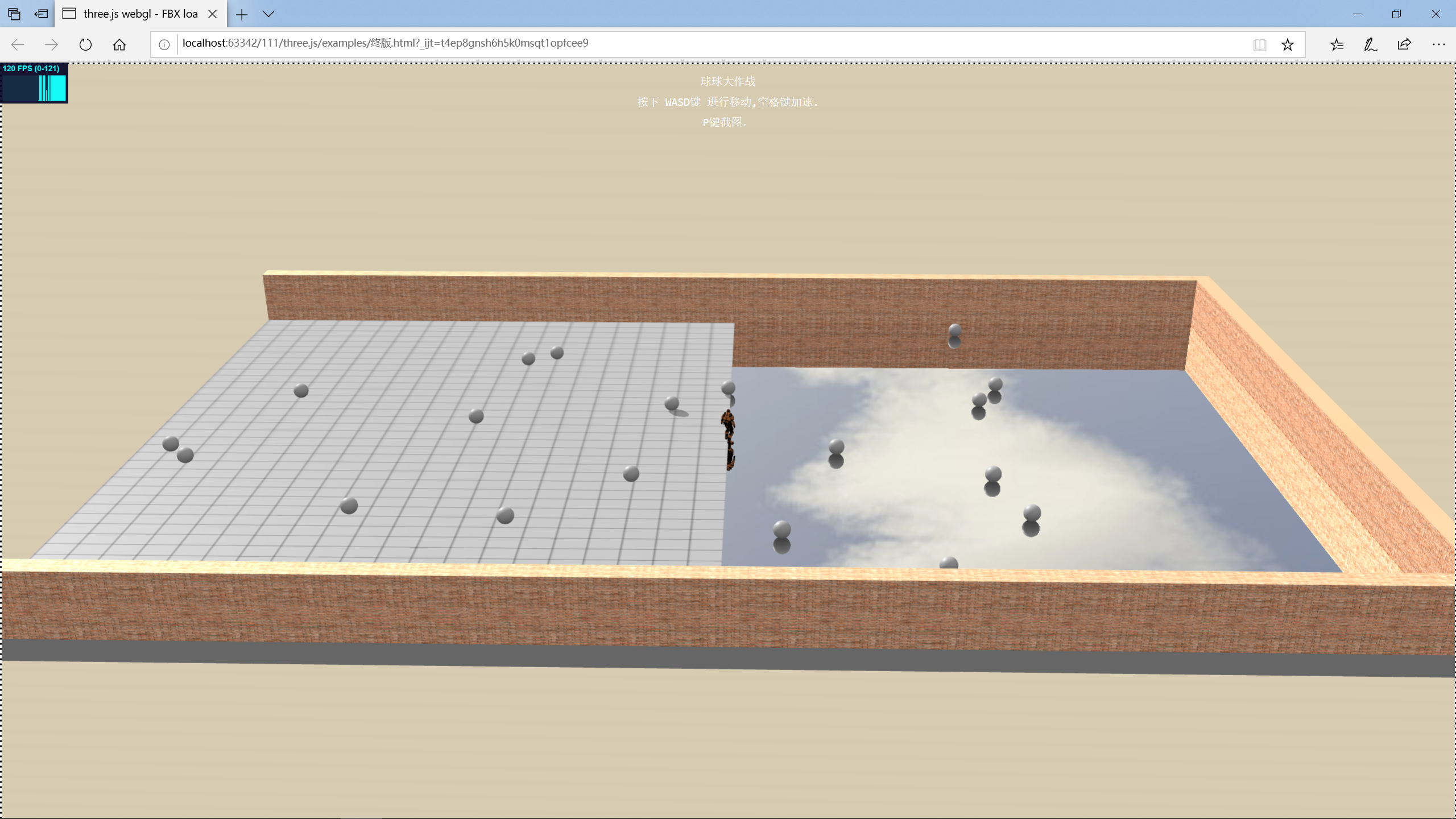
for ( var i = 0; i < ***numObjectsToRemove***; i ++ ) {

removeDebris( ***objectsToRemove***[ i ] );  
  
}

1. **操作说明和效果展示**

WASD控制人物移动；空格键加速移动，P键截屏，鼠标左键漫游定点改变相机朝向，右键平移人物以及相机位置，滚轮进行缩放。

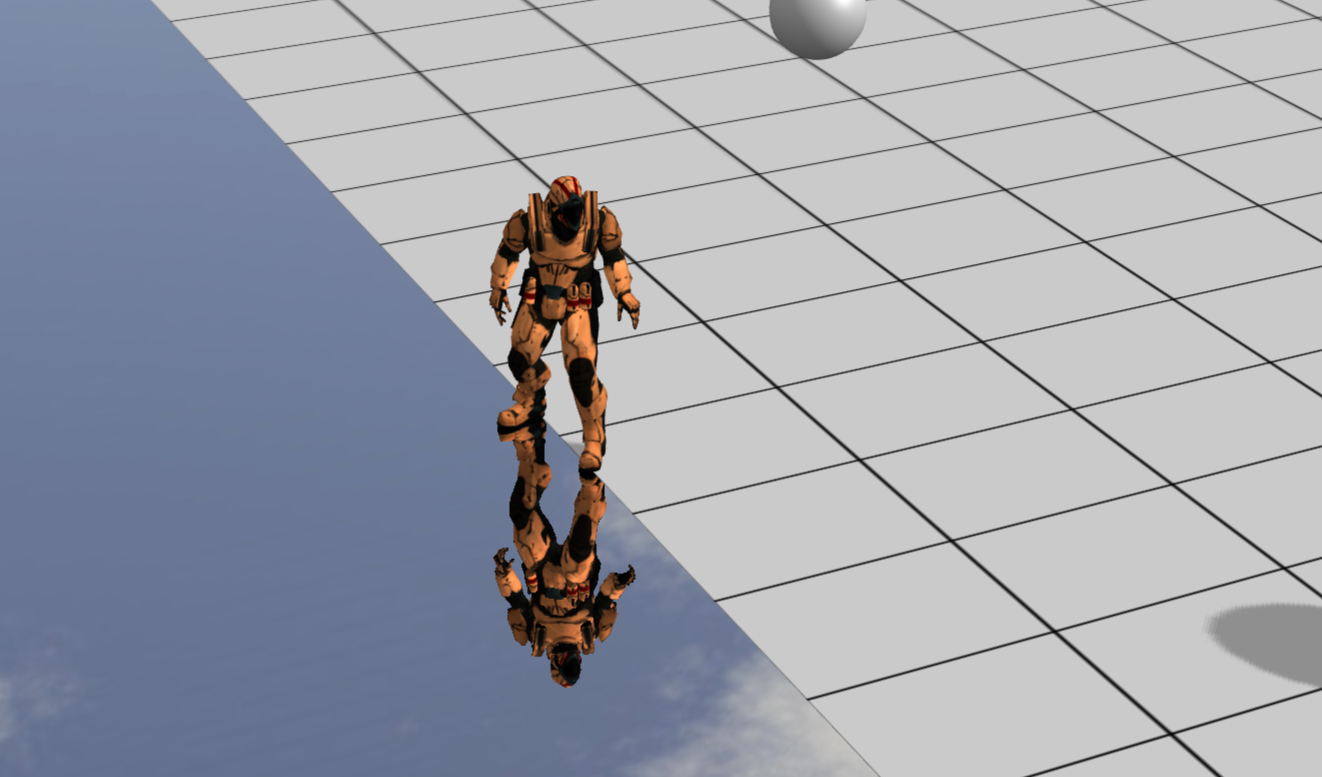
场景总览：



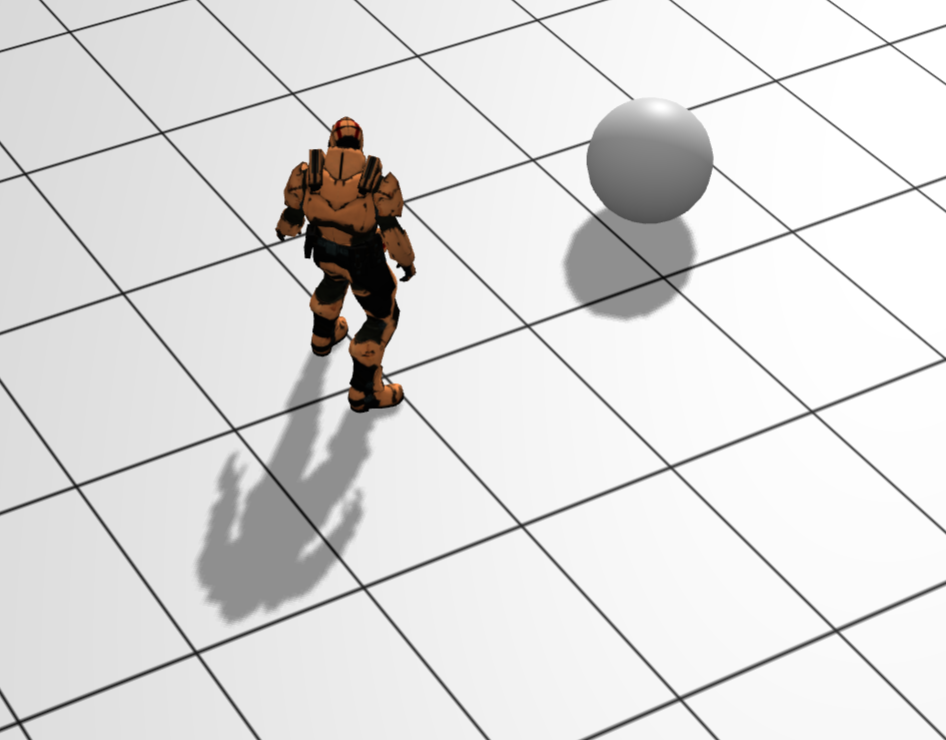
墙面的纹理展示：



人物模型：



阴影效果：



1. **组员与分工**
2. **组员信息**

|  |  |  |
| --- | --- | --- |
| **姓名** | **学号** | **专业** |
| **邱洪发** | **3170103107** | **GIS1701** |
| **杨安邦** | **3170102862** | **GIS1701** |
| **王嘉奇** | **3170101571** | **GIS1701** |

1. **分工**

**邱洪发**：场景搭建、光照、动画移动控制等

**杨安邦：**纹理、移动控制、屏幕截图等

**王嘉奇：**物理引擎、建模等

1. **附件**

源代码详见附件，将“工程与源代码”文件夹作为工程项目打开，可以使用Tomcat、Webstorm等软件搭建本地服务器，否则会遇到浏览器跨域问题，最后运行/examples/index.html文件。