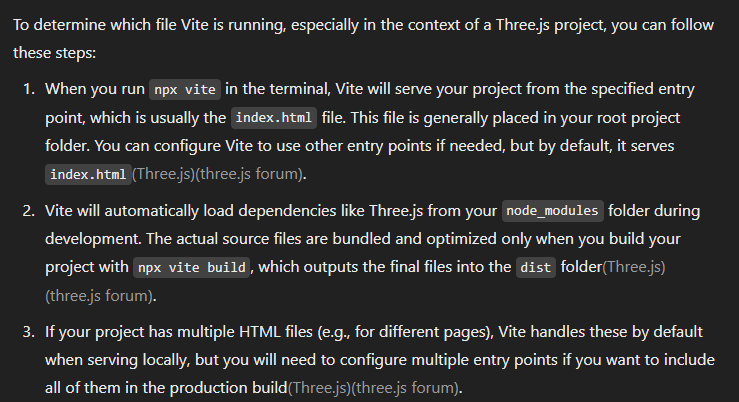
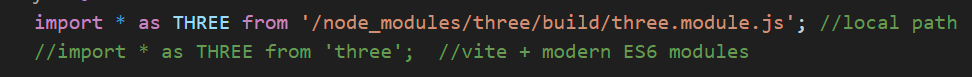
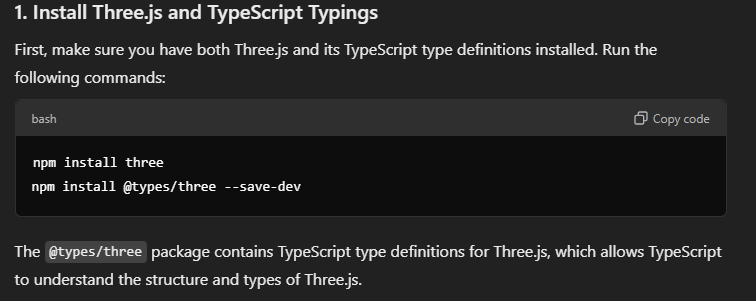
默认.index.html也比较麻烦



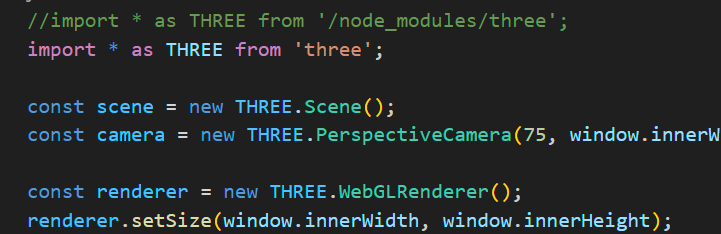


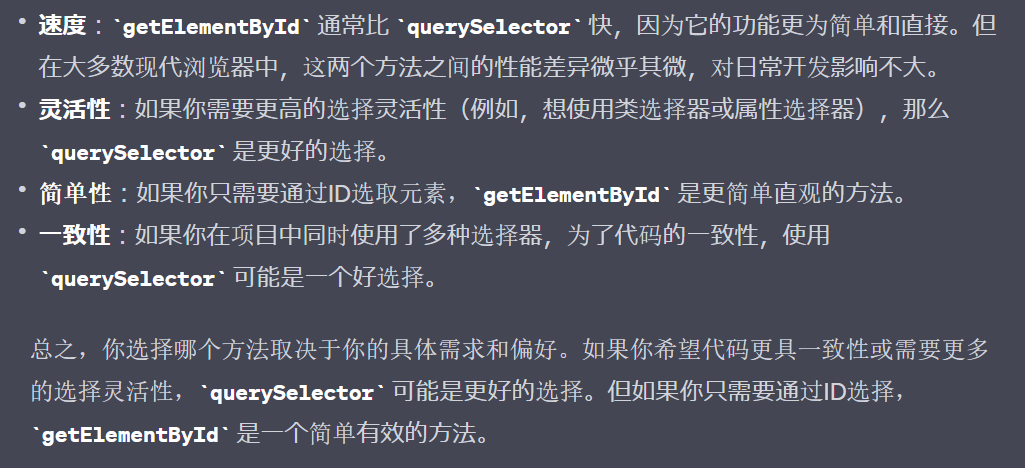
@types/three 似乎是community在手动维护。。。not so reliable?





vite会用绿色？





=======script

扩展

script element的

src=

<script src="https://threejs.org/build/three.js"></script>

=======2D canvas

webgl 的3D context自己管理，

但也是接入<canvas>提供的rendertarget,

the renderer的attribute中的 <canvas> element的引用加入body;

either

◆将自己的canvas append to <body>

document.body.appendChild(renderer.domElement);

or

◆在<body>定义好，指向，



//renderer , context

var renderer = new THREE.WebGLRenderer();

renderer.setSize(window.innerWidth, window.innerHeight);

UI设计基本concepts

嵌套矩形

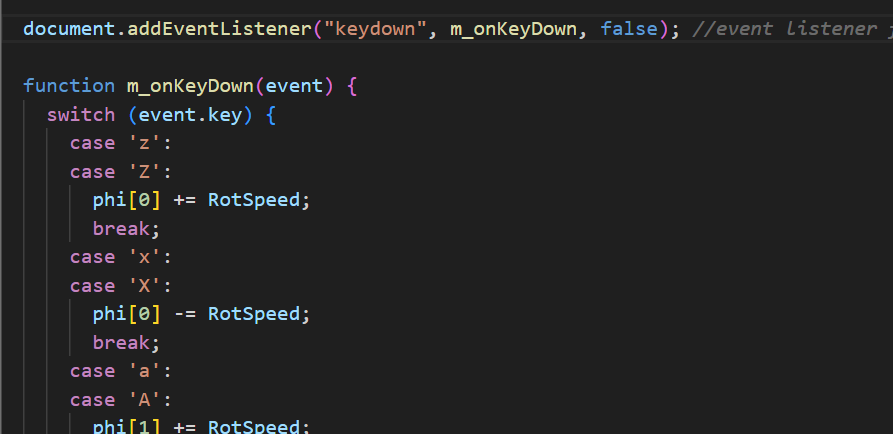
screen

/ window

/viewport

/ canvas

/ elements



------3D API

概念上的hierarchy

◆ context, window

管理在renderer下面；

狭义canvas , by定义camera,

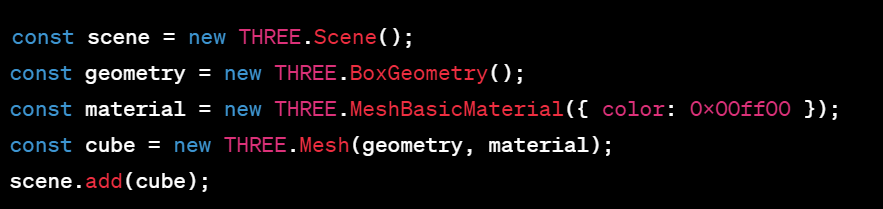
◆Scene

◆lighting setup

围绕

scene

render scene



API应该尽可能地实现出抽象底层实现的

drawing api

最终接口

renderer.render(scene, camera);

render(scene)无争议；

scene客观存在，the objects, the lighting,

scene.Add(object) ; 如何管理is abstracted;

camera决定canvas,

//不过也能将camera设计为renderer一部分罢；

a mesh object takes two

◆geometry

◆material

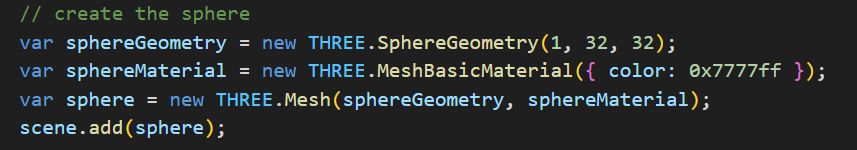
geometry & material, 这两部分是固有概念，

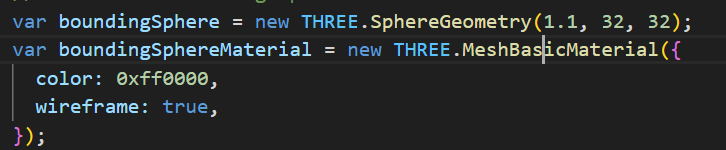
其中我当初稍微觉得有点暧昧的就是顶点数据很自由，远不限于geometry;

实际上object的确不仅是mesh

所以强调是mesh object

这里的mesh, 就区分于geometry本身；





js描述性接口名 本身就说明了是什么东西；

基本就是 basic material继承material的设计

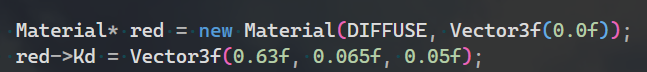
也有以enum /mode 形式的universal constructor

这么做的问题是其他参数可能 不统一 通用性不好

我这里觉得，作为参数的只有非常统一的，

其他的set public property也没问题，

C++的参数 如果有 name = value的机制就好了



处理scene

a renderer take

scene(objects集合) + camera

renderer 不需要 own scene,

scene可以客观地存在，without being drawed;

camera适合从属于renderer，

考虑到camera也是唯一地用于drawcall

var scene = new THREE.Scene();

scene.add(sphere);

scene.remove(obj);

遍历scene

scene.traverse(function(object) {

// do sth for mesh objects

if (object.isMesh) {

object.material.needsUpdate = true;

}

});

光源

const light = new THREE.PointLight(0xffffff);

light.position.set(0, 0, 10);

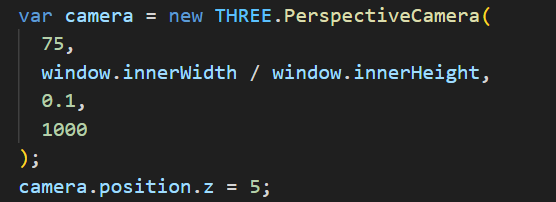
scene.add(light);

渲染器

var renderer = new THREE.WebGLRenderer();

renderer.setSize(window.innerWidth, window.innerHeight);

相机



const camera =

new THREE.PerspectiveCamera(75, window.innerWidth / window.innerHeight, 0.1, 1000);

材质

var sphereMaterial = new THREE.MeshBasicMaterial({ color: 0x7777ff });

几何

var sphereGeometry = new THREE.SphereGeometry(1, 32, 32);

mesh objects

var bounding = new THREE.Mesh(boundingSphere, boundingSphereMaterial);

mesh.position.set(1, 2, 3);

mesh.rotation.set(Math.PI / 4, 0, 0);

mesh.scale.set(1, 2, 3);

depth map workflow

定义"render target"

结果is a member of the target,

const depthTexture = depthTarget.texture;

结论上就是一个texture,



depthmap creation本质上取决于shader 和 framebuffer实现

但接口而言称 set as depth material.

built-in material基本命名为 **Mesh + shader + Material**.

在不同pass分别设置material;



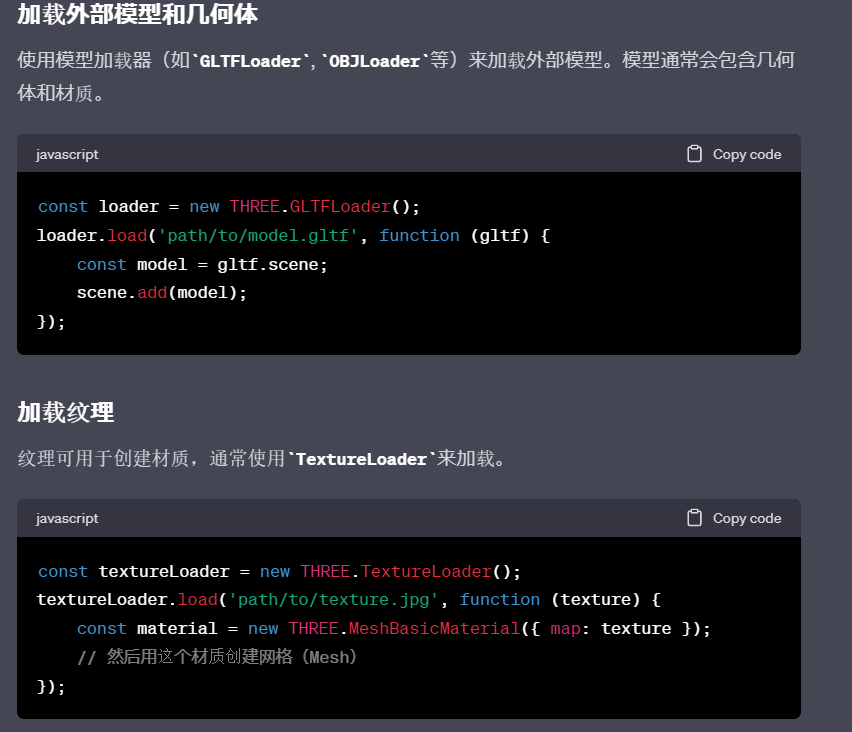
shader为material的一部分

.vertexShader

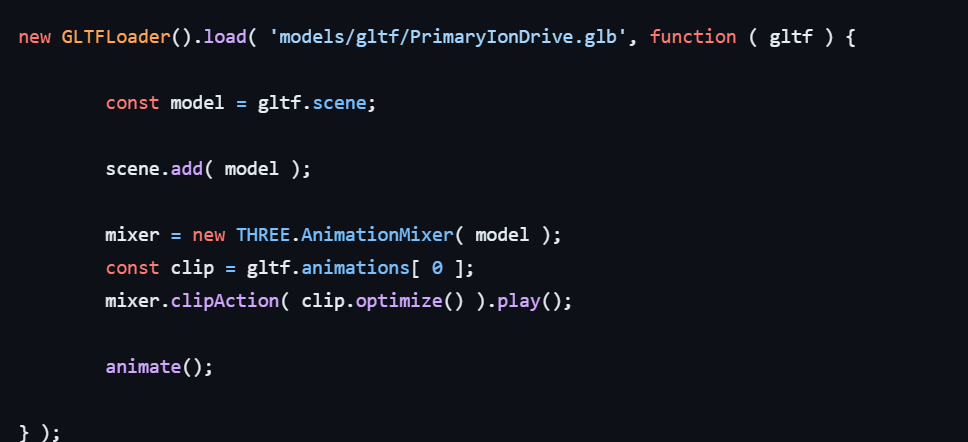
.fragmentShader

.uniforms





model animator



mixer(deltaTime)



-------3js

看起来不错的设置

Buffer.ToAttribute

InstancedBufferAttribute

可能is shader node不能复刻

但处理buffer思路感觉很好

instanceIndex,

Invocation ID

是一回事，

问题是两者的自如转换，

This kind of chaining is common in JavaScript, especially in functional programming and in scenarios involving callbacks or asynchronous operations.

=

() => {...}

an arrow function named createBuffer

InstancedBufferAttribute

所有传给shader的数据

Float32

count \* 4 , 4 byte

应该是4 component,

既然是attributes也就还有interpret,

vec3，

count个



tslfn是预设的，from shader nodes

看来是不直接写shader，

就要有类似的转化封装，就是shader node了吧；

ま、能不能实现Niagara那种上限表示怀疑；

const computeHit = tslFn( () => { ... } )().compute( particleCount );

tslFn(...) takes another function as rgument,

tslFn(...)()

可以推测其返回另一个函数,

然后后() immediately invoke it,

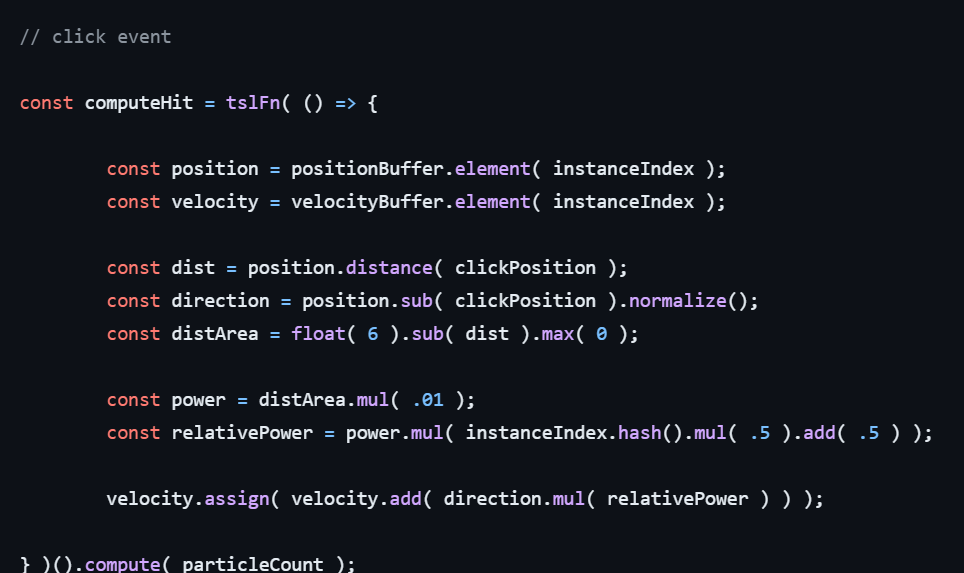
and return a object,

.compute() 是方法

最后返回到computeHit=

被调用在onMove( event )

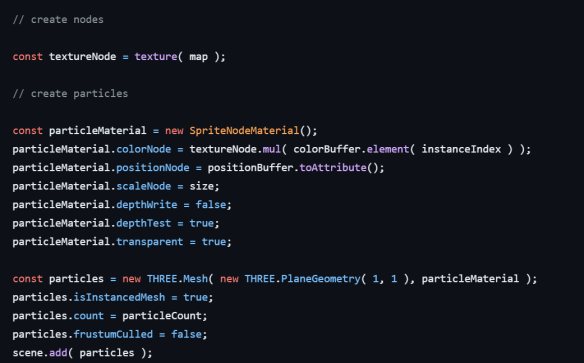
renderer.compute( computeHit );



Buffer.toAttribute()

isInstanced

.count



定义

exported, meaning it can be used in other modules that import it.

传入js函数, create new Shader Node,

return fn()

返回shaderNode.call() 看着像某种封装

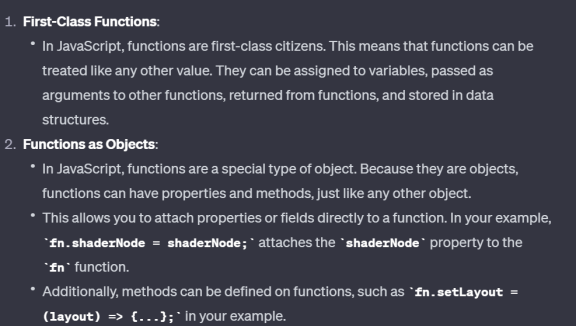
js feature

"First-Class Functions", "Functions as Objects."

fn还有两个shaderNode相关field

那就是shaderNode.call().compute(particleCount)





import{} from ' '



交互envent

window.addEventListener()



examples

源码看法

基本flow

import

定义参数

init()为入口



add built-in pass in 3js

class EffectComposer()

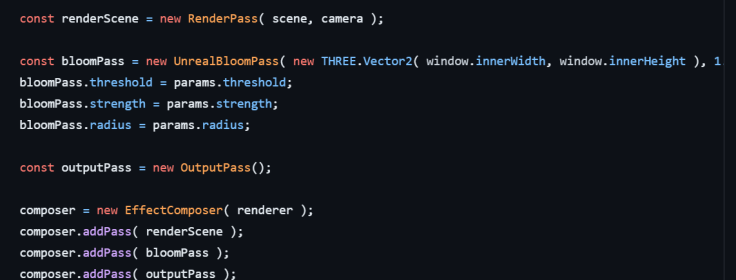
class Pass()

composer.

addPass()

renderpass

postProc pass



custom rendertarget



high pass filter

class ShaderMaterials;

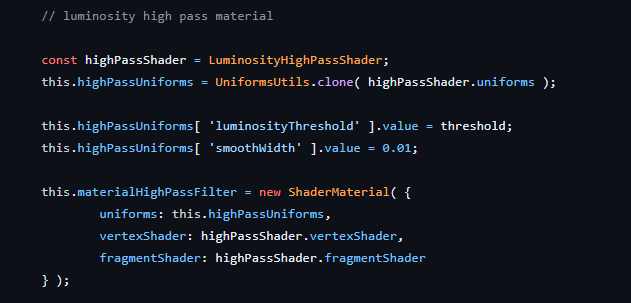
(

{

uniforms:

vs:

fs:



uniform 像是dictionary,

弱类型这手真方便啊

fs, vs就是string





custom pass

class UnrealBloomPass extends Pass {

constructor基本参数

super()

new WebGLRenderTarget(

两个lists .push()

// render targets

this.renderTargetsHorizontal = [];

this.renderTargetsVertical = [];

material list

渲染流程

uniform

setRenderTarget

clear

Quad render



参数renderer. 负责

clear

rendertarget state， 不直接用FBO

rendertarget和管线联动

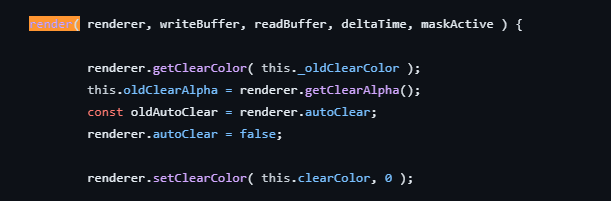
input

basic.map

看样子是交替的读取 readbuffer内容

render() 似乎不是在app里调用 pass里的

提供了 IO和系统参数

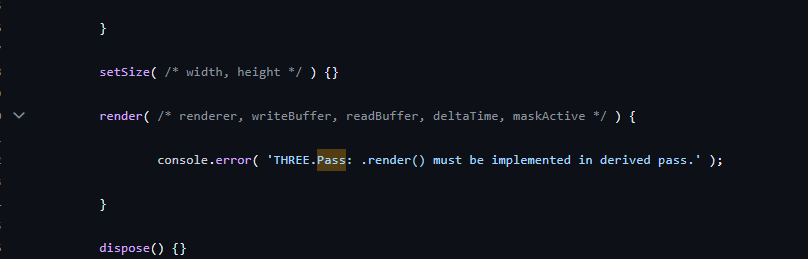


= readBuffer.texture



class Pass

render() 唯一的问题是 如果是multiple呢？

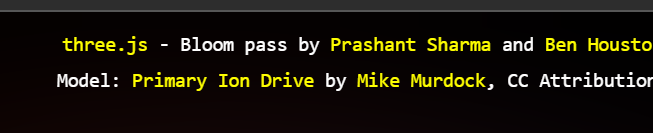


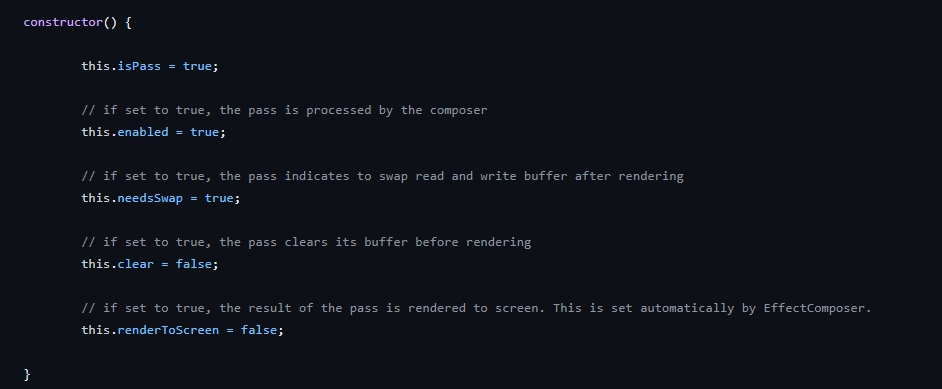
<div id="info">

</div>

<br/>分行

<a href="https://threejs.org" target="\_blank" rel="noopener">three.js</a>





将pass提出的清晰性

这个class我要了

各种pass堆叠久已

rendertarget的切换 也在局部即可

将context state放到material里

日后再说