



WebGL&Threejs

唐婷 2020-9-23
医学技术前端



图形基础

WebGL

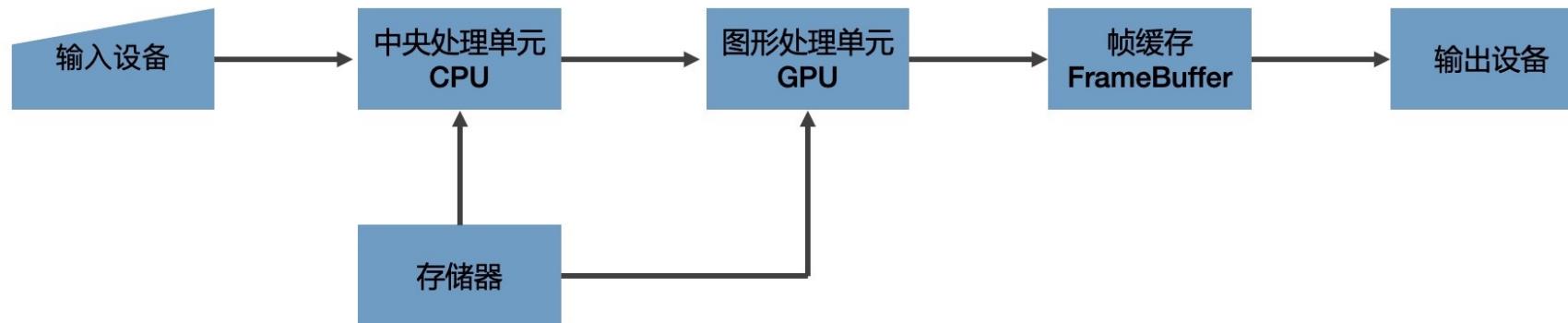
Threejs

Part

1 图形基础



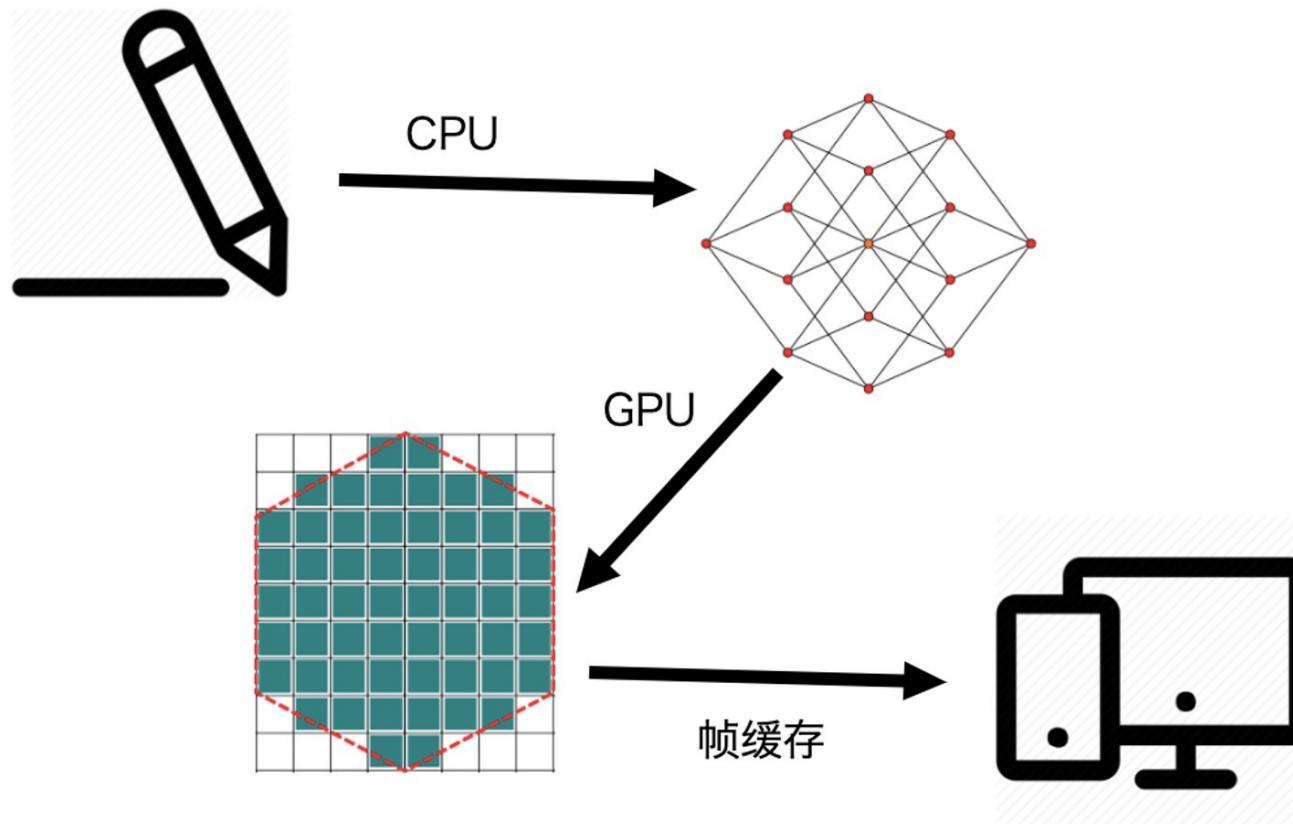
一个通用计算机图形系统主要包括 6 个部分，分别是输入设备、中央处理单元、图形处理单元、存储器、帧缓存和输出设备。



- ✓ 输入：人机交互及数据文件等。
- ✓ CPU (Central Processing Unit)：中央处理单元，负责逻辑计算。
- ✓ GPU (Graphics Processing Unit)：图形处理单元，负责图形计算。
- ✓ 存储器：远程和本机。
- ✓ 帧缓存：在绘图过程中，像素信息被存放于帧缓存中，帧缓存是一块内存地址。
- ✓ 输出：显示器、打印机等。

图形系统是如何绘图的

数据经过 CPU 处理，成为具有特定结构的几何信息。然后，这些信息会被送到 GPU 中进行处理。在 GPU 中要经过两个步骤生成光栅信息。这些光栅信息会输出到帧缓存中，最后渲染到屏幕上。

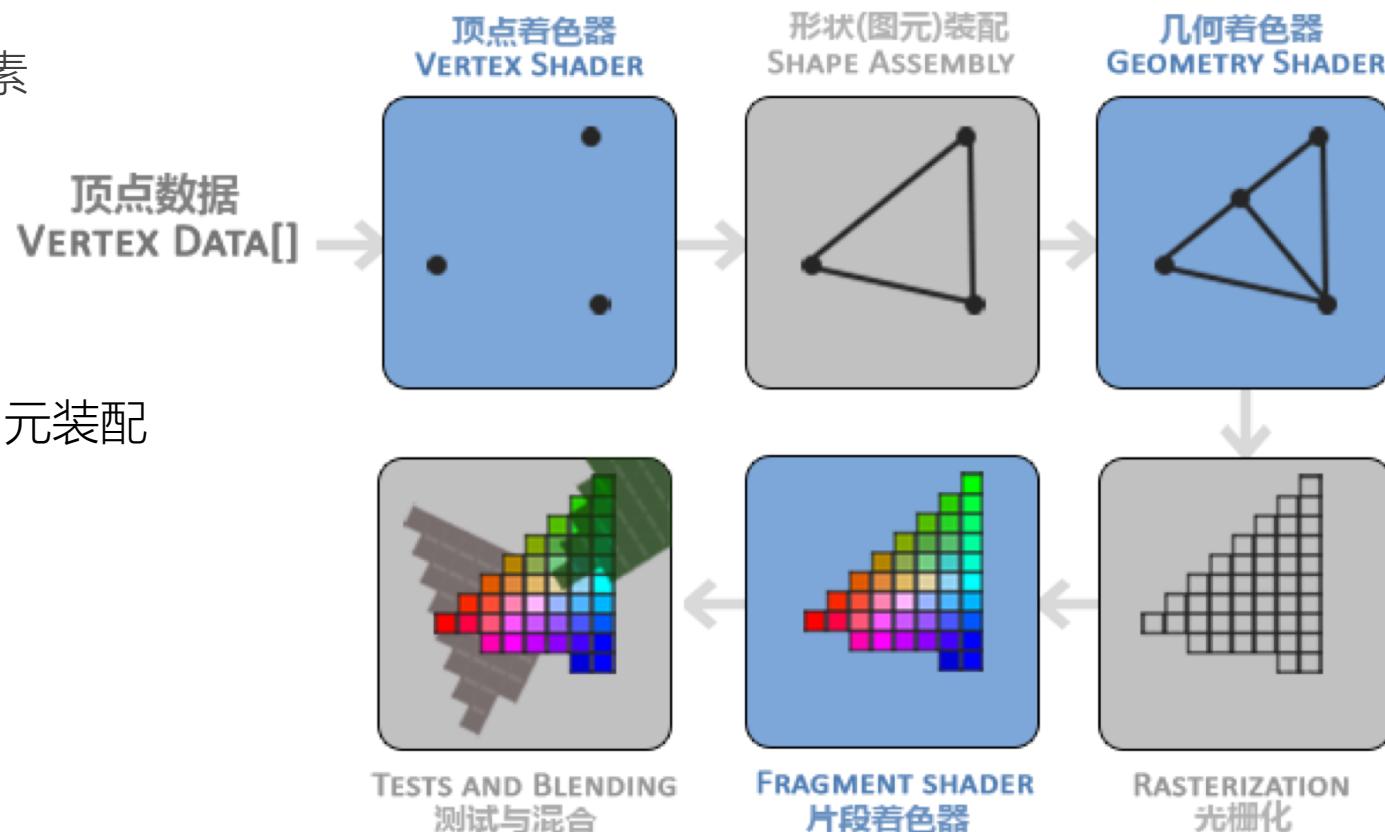


主要工作可以被划分为两个部分：

- ✓ 把 3D 坐标转换为 2D 坐标
- ✓ 把 2D 坐标转变为实际的有颜色的像素

具体实现可分为六个阶段：

- 顶点着色器 (Vertex Shader)
- 形状装配 (Shape Assembly) 又称图元装配
- 几何着色器 (Geometry Shader)
- 光栅化 (Rasterization)
- 片段着色器 (Fragment Shader)
- 测试与混合 (Tests and Blending)



Part

2 WebGL



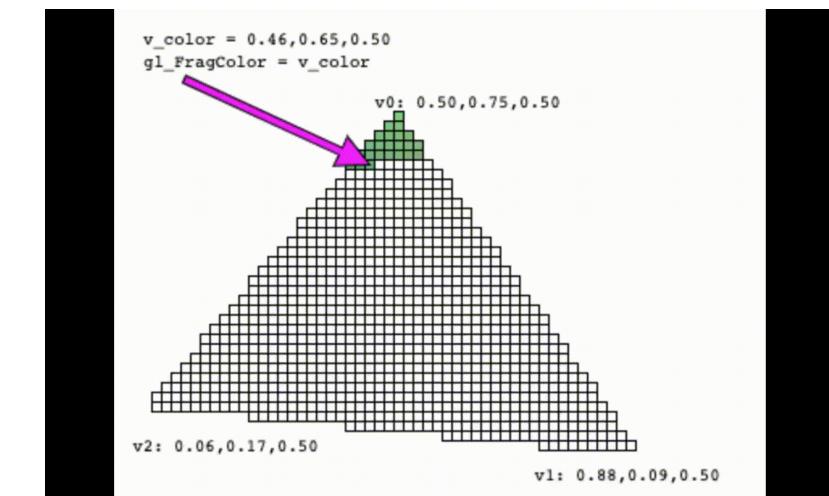
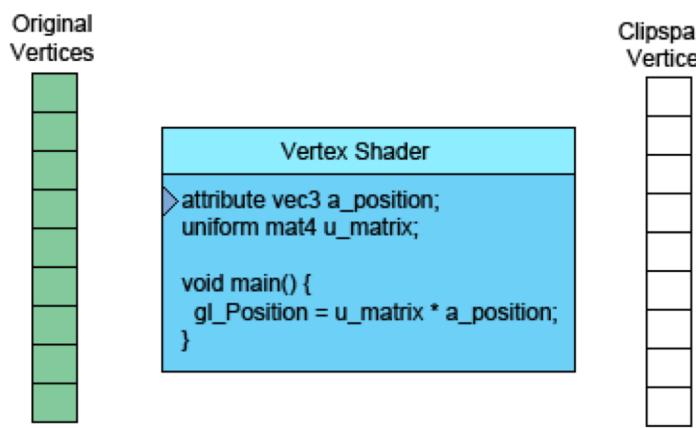
- 传统的 **HTML+CSS** , 通常用来呈现普通的 Web 网页。
- **SVG** (Scalable Vector Graphics , 可缩放矢量图) , SVG 是一种基于 XML 语法的图像格式 , 可以用图片 (img 元素) 的 src 属性加载。
- **Canvas2D** , 浏览器提供的 Canvas API 中的一种上下文 , 使用它可以非常方便地绘制出基础的几何图形。
- **WebGL** , Canvas API 中的另一种上下文 , 它是 OpenGL ES 规范在 Web 端的实现。通过它 , 用 GPU 渲染各种复杂的 2D 和 3D 图形。

WebGL仅仅是一个光栅化引擎，它可以根据你的代码绘制出点，线和三角形。

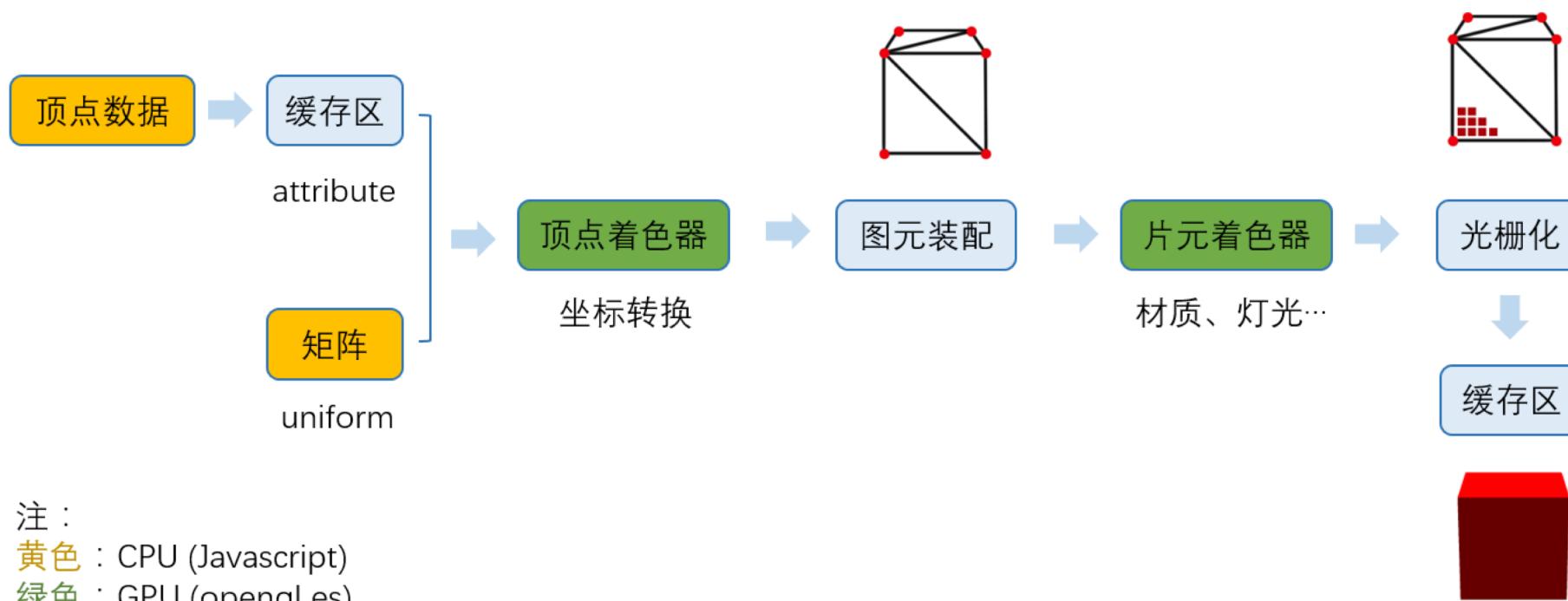
在GPU中运行。使用一种和C或C++类似的强类型的语言 GLSL (GL着色语言)。这样的代码需要提供成对的方法。每对方法中一个叫顶点着色器，另一个叫片断着色器，每一对组合起来称作一个 program (着色程序) 。

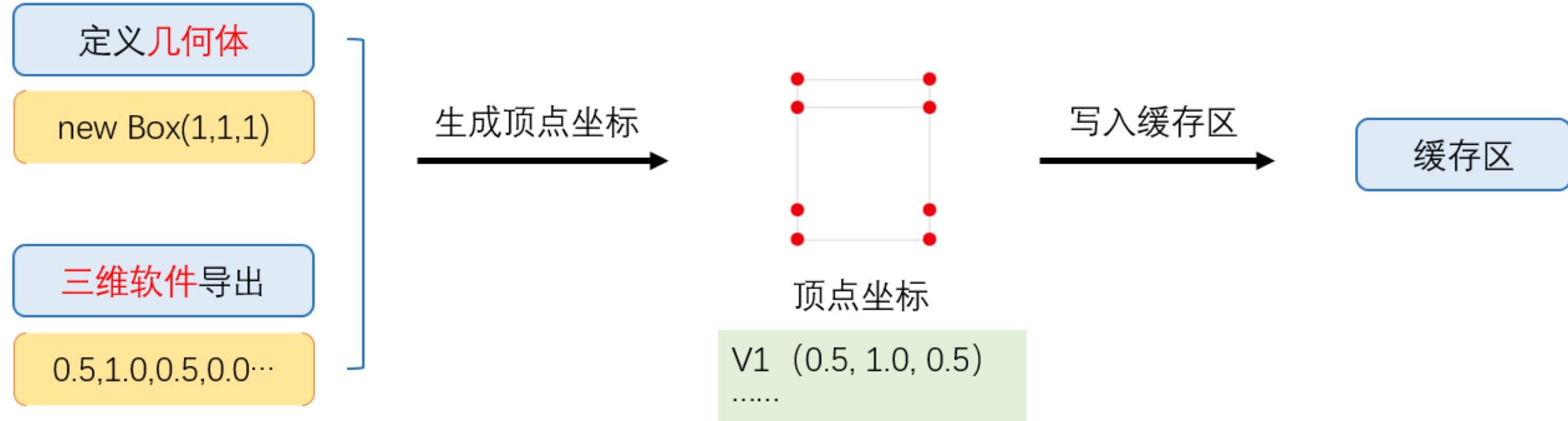
顶点着色器的作用是计算顶点的位置。根据计算出的一系列顶点位置，WebGL可以对点，线和三角形在内的一些图元进行光栅化处理。

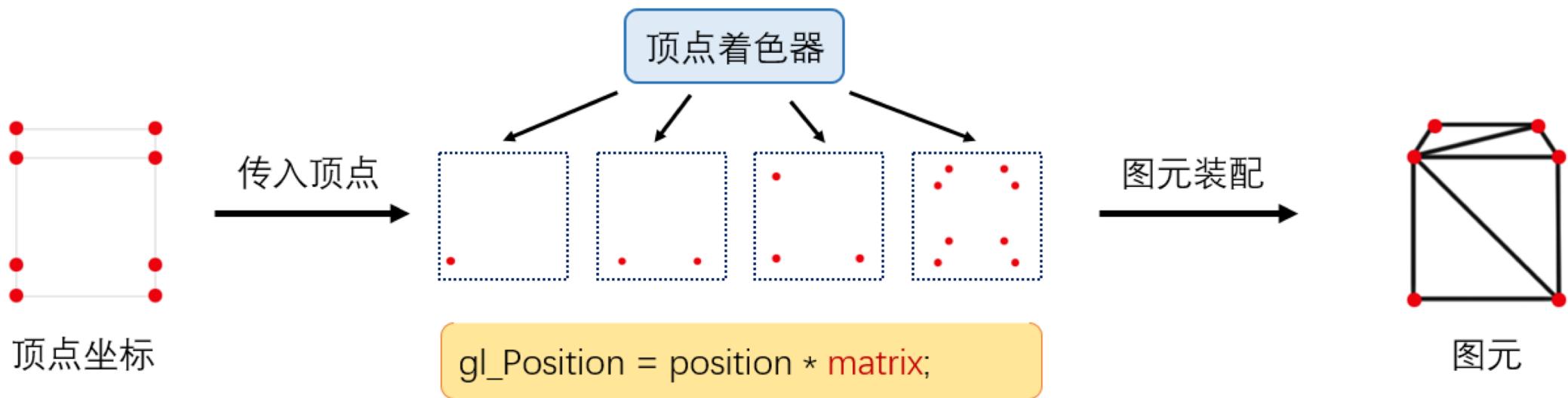
当对这些图元进行光栅化处理时需要使用片断着色器方法。片断着色器的作用是计算出当前绘制图元中每个像素的颜色值。

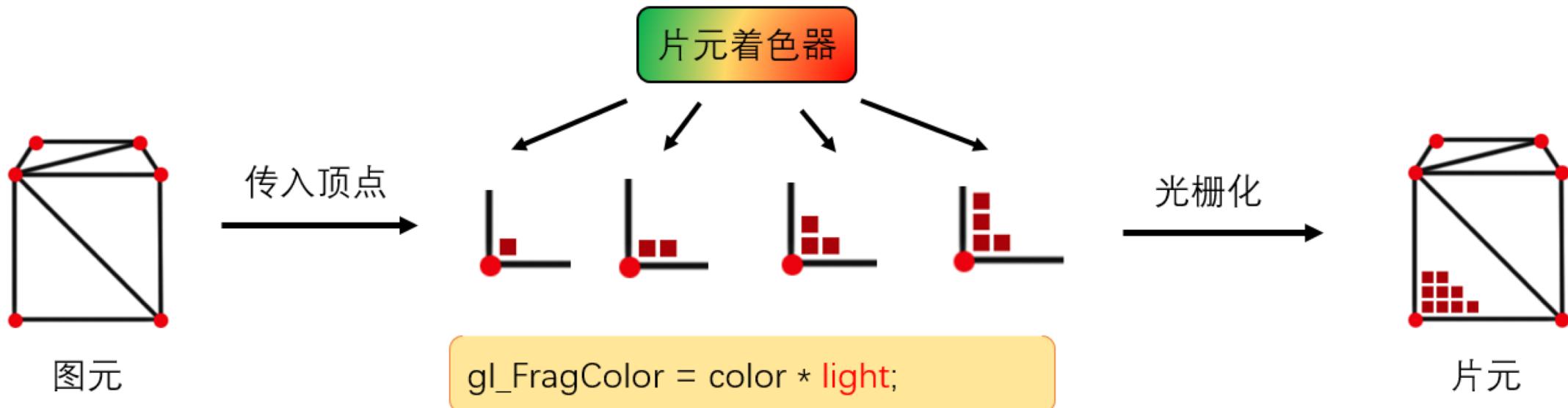


1. 获取顶点坐标
2. 图元装配（即画出一个个三角形）
3. 光栅化（生成片元，即一个个像素点）

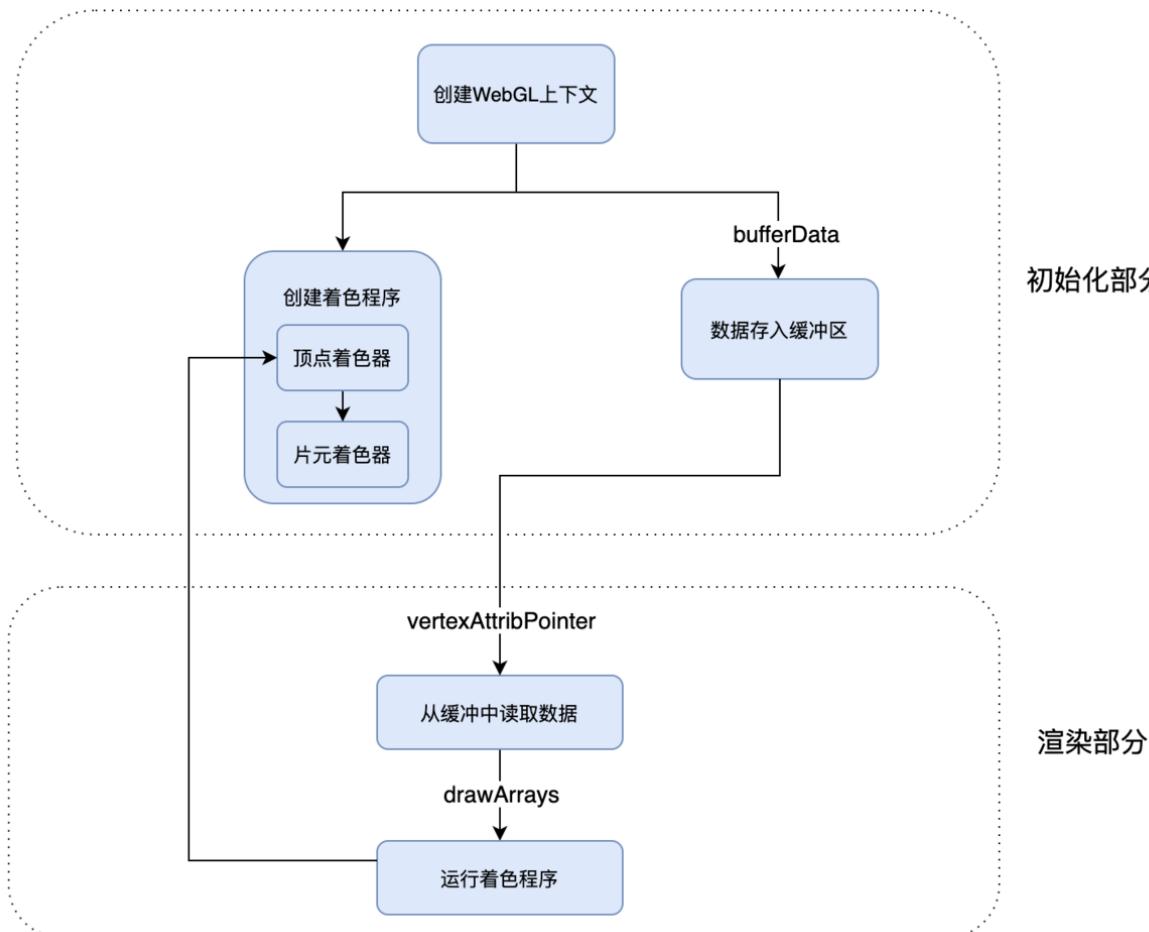








如何用 WebGL 绘制三角形



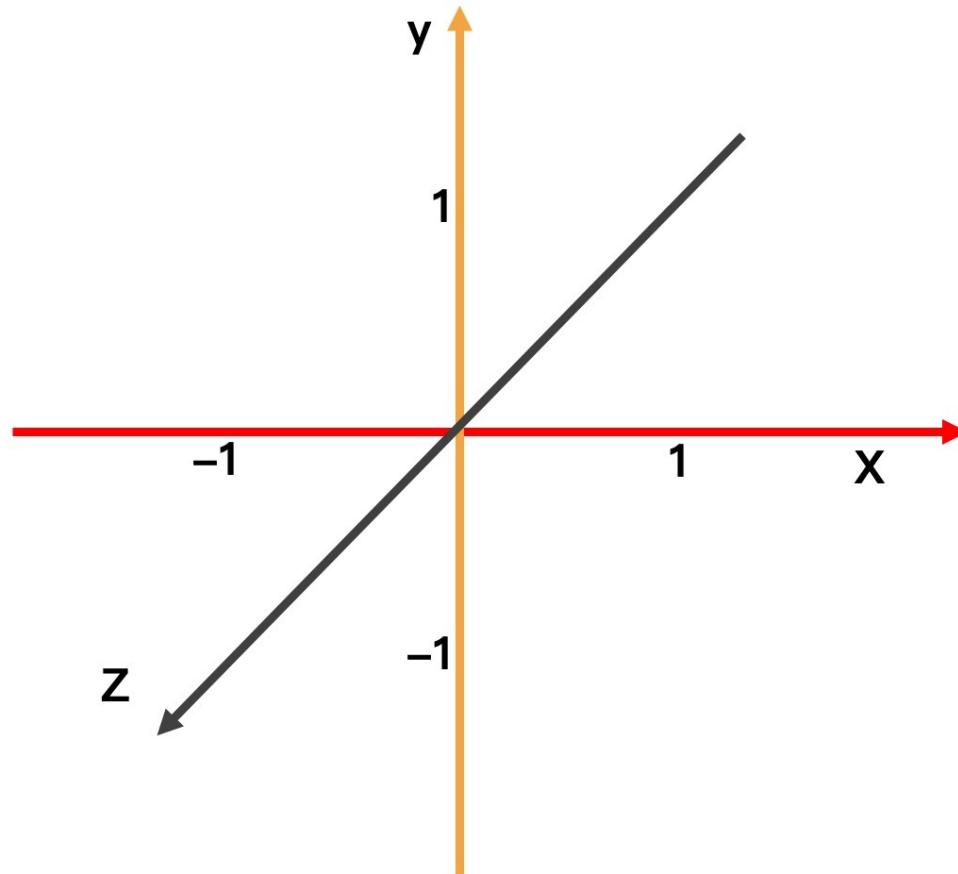
```

const canvas = document.querySelector('canvas');
const gl = canvas.getContext('webgl');
const vertex =
`attribute vec2 position;

void main() {
    gl_PointSize = 1.0;
    gl_Position = vec4(position * 0.5, 1.0, 1.0);
}
`;
const fragment =
`precision mediump float;

void main()
{
    gl_FragColor = vec4(1, 0, 0, 1);
}
`;
const vertexShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertexShader, vertex);
gl.compileShader(vertexShader);
const fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragmentShader, fragment);
gl.compileShader(fragmentShader);
const program = gl.createProgram();
gl.attachShader(program, vertexShader);
gl.attachShader(program, fragmentShader);
gl.linkProgram(program);
gl.useProgram(program);
const points = new Float32Array([
    -1, -1,
    0, 1,
    1, -1,
]);
const bufferId = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, bufferId);
gl.bufferData(gl.ARRAY_BUFFER, points, gl.STATIC_DRAW);
const vPosition = gl.getAttribLocation(program, 'position');
gl.vertexAttribPointer(vPosition, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(vPosition);
gl.clear(gl.COLOR_BUFFER_BIT);
gl.drawArrays(gl.TRIANGLES, 0, points.length / 2);

```



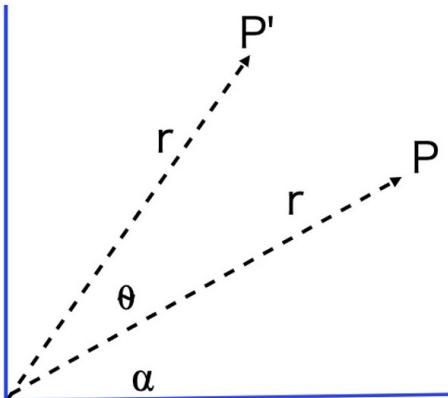
WebGL 的坐标系是一个三维空间坐标系，坐标原点是 (0,0,0)。其中，x 轴朝右，y 轴朝上，z 轴朝外。这是一个右手坐标系。

仿射变换简单来说就是“线性变换 + 平移”。

平移：让向量 $P(x_0, y_0)$ 沿着向量 $Q(x_1, y_1)$ 平移，只要将 P 和 Q 相加

$$\begin{cases} x = x_0 + x_1 \\ y = y_0 + y_1 \end{cases}$$

旋转：假设向量 P 的长度为 r ，角度是 α ，将它逆时针旋转 θ 角，此时新的向量 P' 的参数方程为：

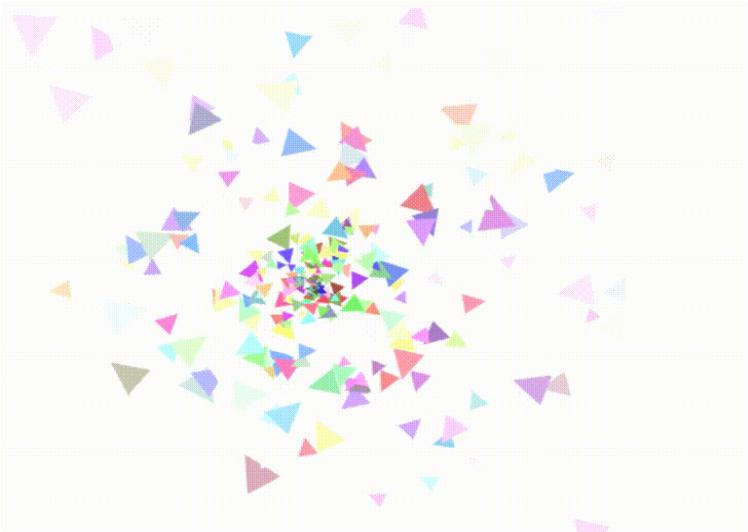


$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \times \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$$

缩放：直接让向量与标量（标量只有大小、没有方向）相乘。

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \times \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$$

仿射变换的应用：实现粒子动画



translateMatrix 是偏移矩阵，
rotateMatrix 是旋转矩阵，
scaleMatrix 是缩放矩阵。

将 pos 的值设置为这三个矩阵与 position 的乘积，这样就完成对顶点的线性变换。

```
attribute vec2 position;
uniform float u_rotation;
uniform float u_time;
uniform float u_duration;
uniform float u_scale;
uniform vec2 u_dir;
varying float vP;

void main() {
    float p = min(1.0, u_time / u_duration);
    float rad = u_rotation + 3.14 * 10.0 * p;
    float scale = u_scale * p * (2.0 - p);
    vec2 offset = 2.0 * u_dir * p * p;
    mat3 translateMatrix = mat3(
        1.0, 0.0, 0.0,
        0.0, 1.0, 0.0,
        offset.x, offset.y, 1.0
    );
    mat3 rotateMatrix = mat3(
        cos(rad), sin(rad), 0.0,
        -sin(rad), cos(rad), 0.0,
        0.0, 0.0, 1.0
    );
    mat3 scaleMatrix = mat3(
        scale, 0.0, 0.0,
        0.0, scale, 0.0,
        0.0, 0.0, 1.0
    );
    gl_PointSize = 1.0;
    vec3 pos = translateMatrix * rotateMatrix * scaleMatrix * vec3(position, 1.0);
    gl_Position = vec4(pos, 1.0);
    vP = p;
}
```

例如绘制3000个小球。性能排序：WebGL(60fps)>Canvas(30fps)>SVG(15fps)

用 WebGL 渲染，不需要一个一个小球去渲染，利用 GPU 的并行处理能力，可以一次完成渲染。因为要渲染的小球形状相同，所以它们的顶点数据是可以共享的。WebGL 支持的批量绘制技术，叫做 **InstancedDrawing（实例化渲染）**。

对于 Canvas 和 SVG 来说，影响渲染性能的主要因素是绘制元素的数量和元素的大小。相比较而言，Canvas 的整体性能要优于 SVG，尤其是图形越多，二者的性能差异越大。SVG 是浏览器 DOM 来渲染的，元素个数越多，消耗就越大。

WebGL 要复杂一些，它的渲染性能主要取决于三点。

1. 渲染次数，渲染次数越多，性能损耗就越大。需注意，要绘制的元素个数多，不一定渲染次数就多，因为 WebGL 支持批量渲染。
2. 着色器执行的次数，这里包括顶点着色器和片元着色器，前者的执行次数和几何图形的顶点数有关，后者的执行次数和图形的大小有关。
3. 着色器运算的复杂度，复杂度和 glsl 代码的具体实现有关，越复杂的处理逻辑，性能的消耗就会越大。

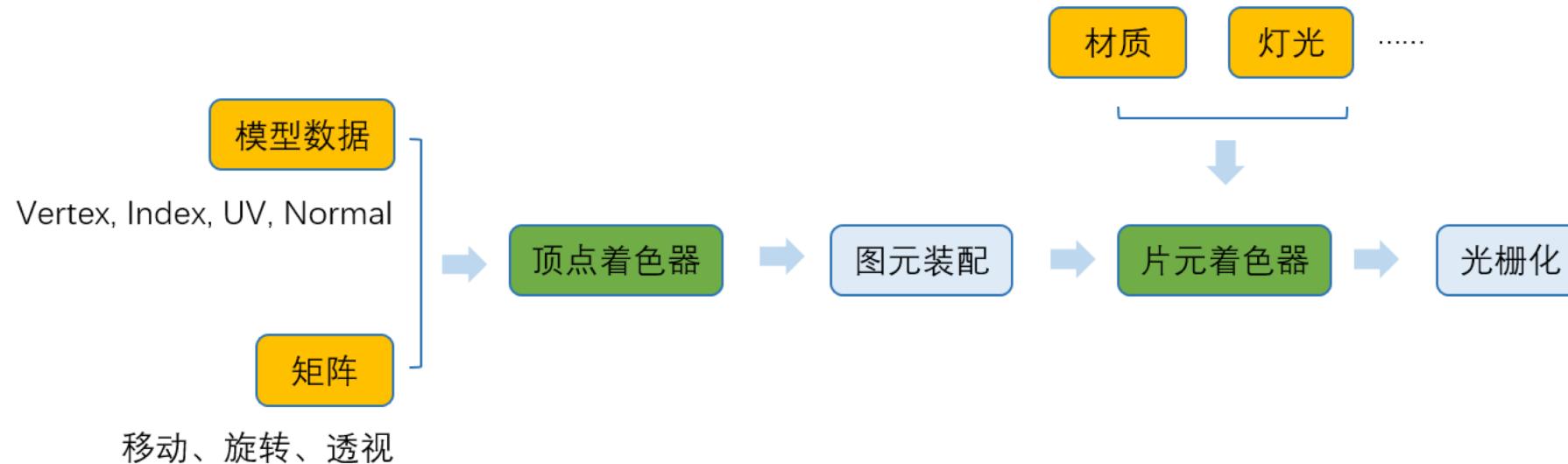
Part

3 Threejs

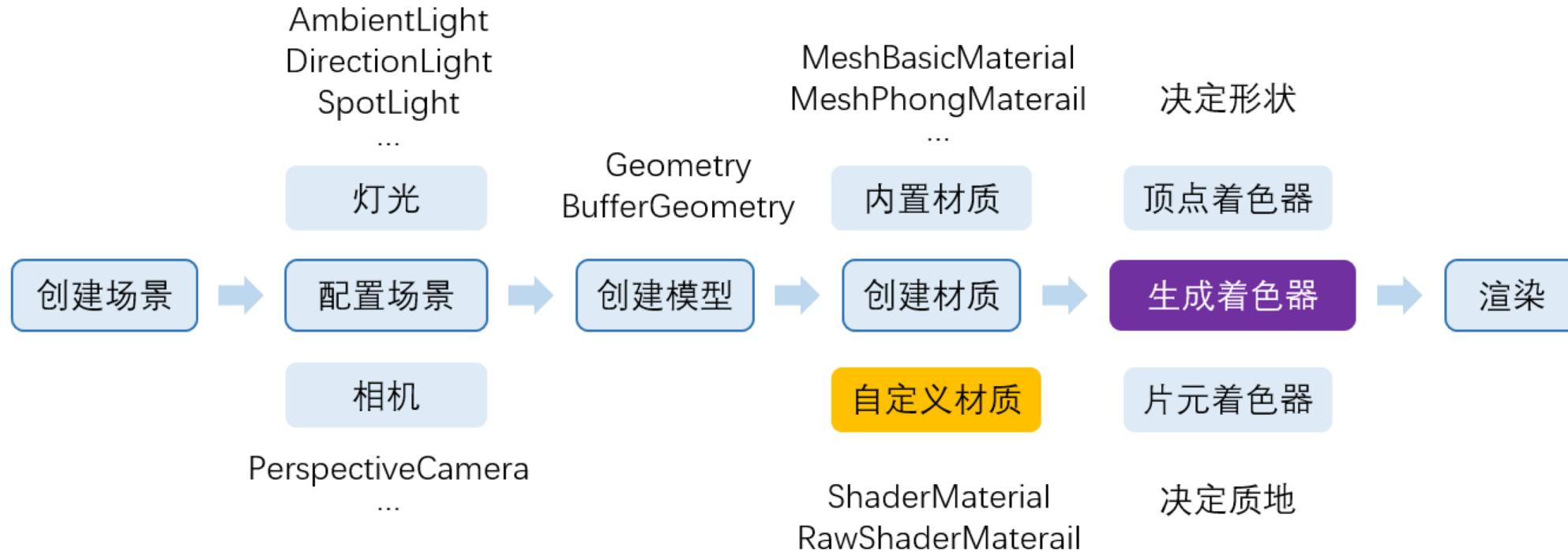


Three.js是基于原生WebGL封装运行的三维引擎。

1. 辅助我们导出了模型数据；
2. 自动生成了各种矩阵；
3. 生成了顶点着色器；
4. 辅助我们生成材质，配置灯光；
5. 根据我们设置的材质生成了片元着色器。



黄色和绿色部分，都是Three.js参与的部分。



Three.js完整运行流程



```
const scene = new THREE.Scene();
const geometry = new THREE.BoxGeometry(100, 100, 100); //创建一个立方体几何对象Geometry
const material = new THREE.MeshLambertMaterial({
    color: 0x0000ff
});
const mesh = new THREE.Mesh(geometry, material); //网格模型对象Mesh
scene.add(mesh); //网格模型添加到场景中
const point = new THREE.PointLight(0xffffff);
point.position.set(400, 200, 300); //点光源位置
scene.add(point); //点光源添加到场景中
const ambient = new THREE.AmbientLight(0x444444);
scene.add(ambient);
const width = window.innerWidth; //窗口宽度
const height = window.innerHeight; //窗口高度
const k = width / height; //窗口宽高比
const s = 200; //三维场景显示范围控制系数，系数越大，显示的范围越大
const camera = new THREE.OrthographicCamera(-s * k, s * k, s, -s, 1, 1000);
camera.position.set(200, 300, 200); //设置相机位置
camera.lookAt(scene.position); //设置相机方向(指向的场景对象)
const renderer = new THREE.WebGLRenderer(); // 创建渲染器对象
renderer.setSize(width, height); //设置渲染区域尺寸
renderer.setClearColor(0xb9d3ff, 1); //设置背景颜色
document.body.appendChild(renderer.domElement); //body元素中插入canvas对象
renderer.render(scene, camera);
```

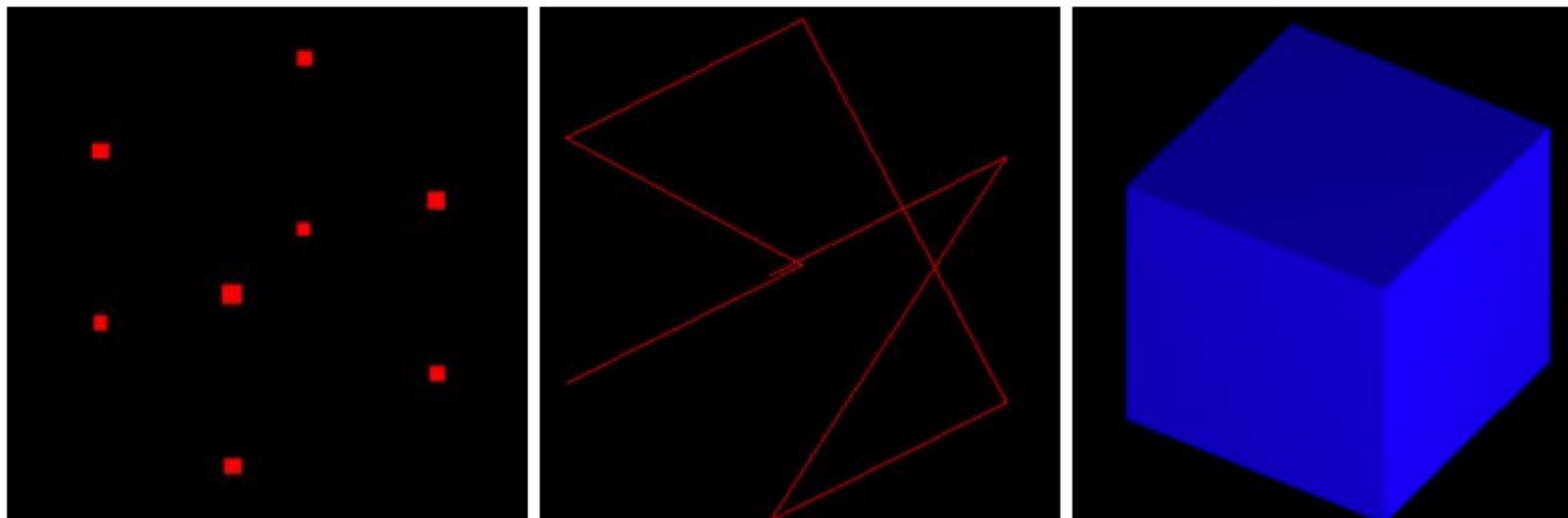
Three.js常用模块—点线面模型对象

点模型Points、线模型Line、网格网格模型Mesh都是由几何体Geometry和材质Material构成，这三种模型的区别在于对几何体顶点数据的渲染方式不同。

```
// 点渲染模式  
const material = new THREE.PointsMaterial({ color: 0xff0000, size: 5.0});
```

```
// 线条渲染模式  
const material=new THREE.LineBasicMaterial({ color:0xff0000});
```

```
// 三角形面渲染模式  
const material = new THREE.MeshLambertMaterial({ color: 0x0000ff});
```



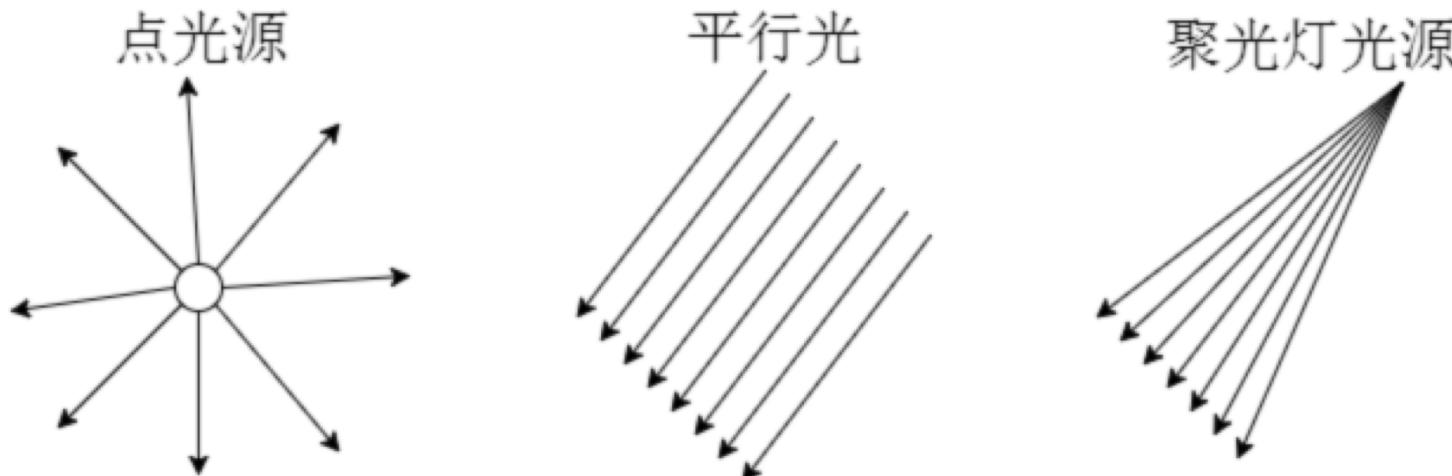
Three.js虚拟光源是对自然界光照的模拟，three.js搭建虚拟场景的时候，为了更好的渲染场景，往往需要设置不同的光源，设置不同的光照强度，就像摄影师给你拍照要设置各种辅助灯光一样。

环境光AmbientLight：没有特定方向的光源，主要是均匀整体改变Three.js物体表面的明暗效果。

点光源PointLight：就像生活中的白炽灯，光线沿着发光核心向外发散，同一平面的不同位置与点光源光线入射角是不同的，点光源照射下，同一个平面不同区域是呈现出不同的明暗效果。

平行光DirectionalLight：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。

聚光源SpotLight：一个沿着特定方向会逐渐发散的光源，照射范围在三维空间中构成一个圆锥体。



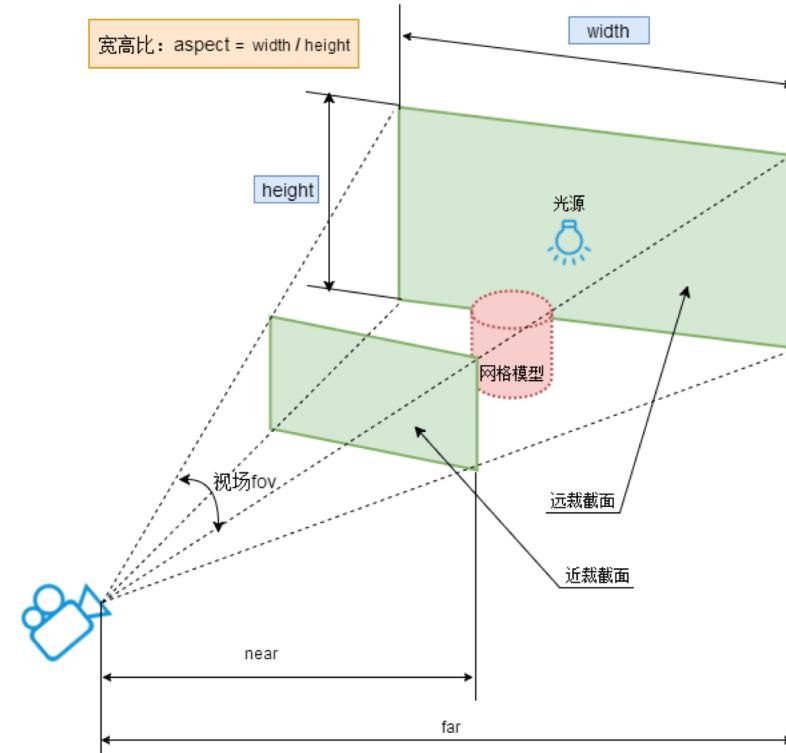
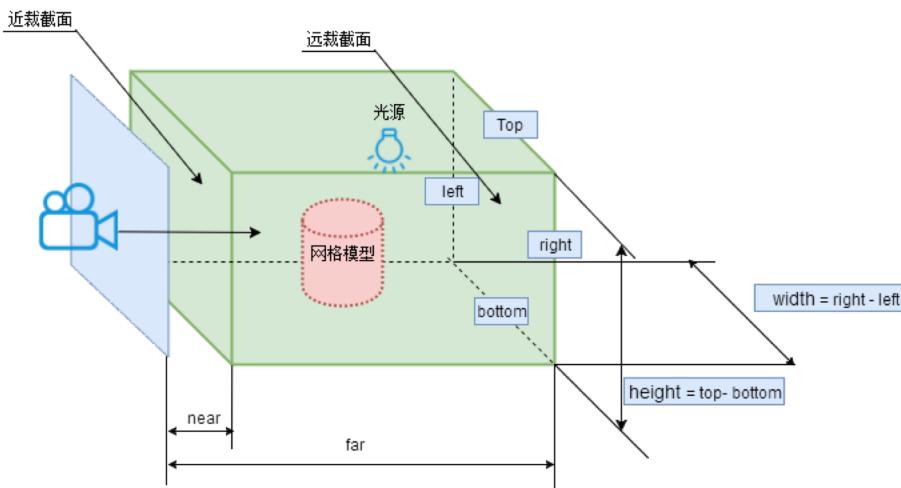
Three.js常用模块—相机对象

Three.js相机对象的最简单的方式就是参考生活中真实的相机拍照过程。

正投影相机: OrthographicCamera(left, right, top, bottom, near, far)

透视投影相机: PerspectiveCamera(fov, aspect, near, far)

相机位置放置 : 如果是观察一个产品外观效果 , 相机就位于几何体的外面 , 如果是室内漫游预览 , 就把相机放在房间三维模型的内部

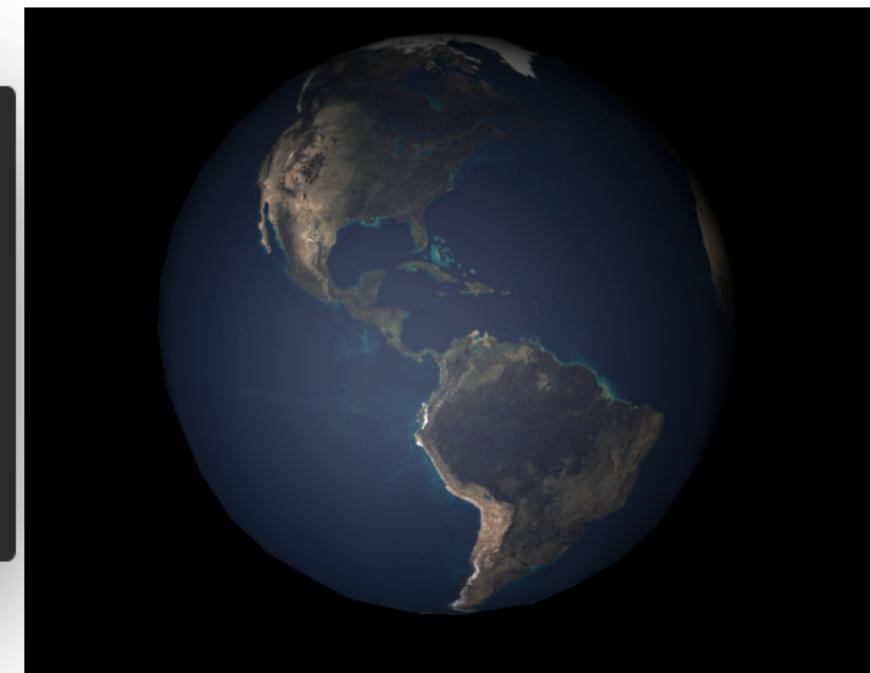


Three.js常用模块——纹理贴图

通过纹理贴图加载器[TextureLoader](#)的load()方法加载一张图片可以返回一个纹理对象[Texture](#)，纹理对象Texture可以作为模型材质颜色贴图.map属性的值。

材质的颜色贴图属性.map设置后，模型会从纹理贴图上采集像素值，这时候一般来说不需要再设置材质颜色.color。.map贴图之所以称之为颜色贴图就是因为网格模型会获得颜色贴图的颜色值RGB。

```
const earthGeometry = new THREE.SphereBufferGeometry( EARTH_RADIUS, 16, 16 );
const earthMaterial = new THREE.MeshPhongMaterial({
    specular: 0x333333,
    shininess: 5,
    map: textureLoader.load( './earth_atmos_2048.jpg' ),
    normalScale: new THREE.Vector2( 0.85, 0.85 )
});
earth = new THREE.Mesh( earthGeometry, earthMaterial );
scene.add( earth );
```



Threejs提供了一系列用户编辑和播放关键帧动画的API,例如关键帧[KeyframeTrack](#)、剪辑[AnimationClip](#)、操作[AnimationAction](#)、混合器[AnimationMixer](#)。



```
// 创建名为Box对象的关键帧数据
const times = [0, 10]; //关键帧时间数组，离散的时间点序列
const values = [0, 0, 0, 150, 0, 0]; //与时间点对应的值组成的数组
// 创建位置关键帧对象: 0时刻对应位置0, 0, 0    10时刻对应位置150, 0, 0
const posTrack = new THREE.KeyframeTrack('Box.position', times, values);
// 创建颜色关键帧对象: 10时刻对应颜色1, 0, 0    20时刻对应颜色0, 0, 1
const colorKF = new THREE.KeyframeTrack('Box.material.color', [10, 20], [1, 0, 0, 0, 0, 1]);
// 创建名为Sphere对象的关键帧数据 从0~20时间段，尺寸scale缩放3倍
const scaleTrack = new THREE.KeyframeTrack('Sphere.scale', [0, 20], [1, 1, 1, 3, 3, 3]);
```

所谓骨骼动画，以人体为例简单地说，人体的骨骼运动，骨骼运动会带动肌肉和人体皮肤的空间移动和表面变化。

Threejs骨骼动画需要通过骨骼网格模型类[SkinnedMesh](#)来实现，一般来说骨骼动画模型都是3D美术创建，然后程序员通过threejs引擎加载解析。

骨骼动画主要骨头关节[Bone](#)、骨骼网格模型[SkinnedMesh](#)、骨架对象[Skeleton](#)这三个骨骼相关的类。

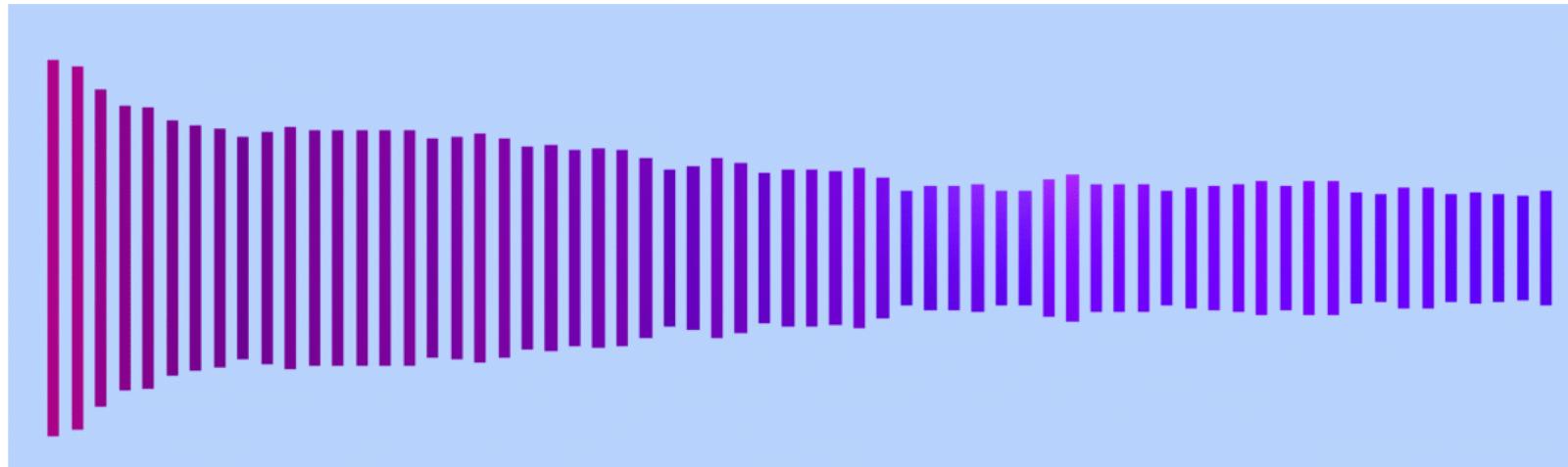
加载外部模型骨骼动画：

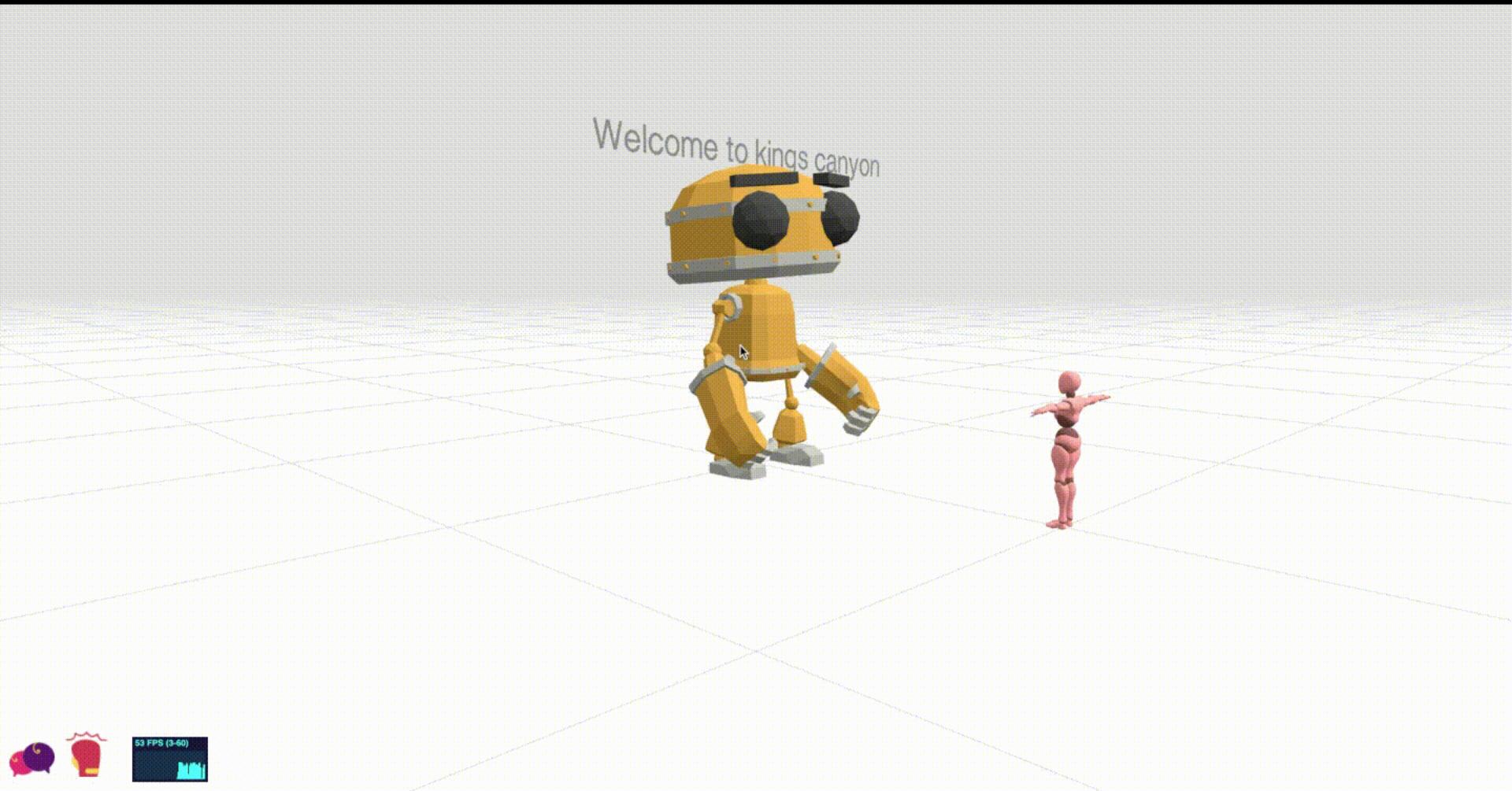


```
const loader = new THREE.ObjectLoader(); // 创建一个加载器
let mixer = null; // 声明一个混合器变量
loader.load("./marine_anims_core.json", function(obj) {
    scene.add(obj); // 添加到场景中
    // 从返回对象获得骨骼网格模型
    const SkinnedMesh = obj.children[0];
    // 骨骼网格模型作为参数创建一个混合器
    mixer = new THREE.AnimationMixer(SkinnedMesh);
    // 查看骨骼网格模型的帧动画数据
    console.log(SkinnedMesh.geometry.animations)
    // 解析跑步状态对应剪辑对象clip中的关键帧数据
    const AnimationAction = mixer.clipAction(SkinnedMesh.geometry.animations[1]);
    // 解析步行状态对应剪辑对象clip中的关键帧数据
    const AnimationAction = mixer.clipAction(SkinnedMesh.geometry.animations[3]);
    AnimationAction.play();
})
```

Threejs对原生[Web Audio API](#)的封装提供了一些方便大家使用的语音模块。比如一个声音和一个网格模型绑定，这样网格模型的位置就是音源位置。

音频相关的API：音频[Audio](#)、位置音频[PositionalAudio](#)、监听者[AudioListener](#)、音频分析器[AudioAnalyser](#)、音频加载器[AudioLoader](#)。







强烈推荐



计算机图形学基础

Three.js入门指南 (<https://www.ituring.com.cn/book/1272>)



<https://github.com/mrdoob/three.js>

<https://github.com/Ovilia/ThreeExample.js>



<https://www.shadertoy.com/>

https://webglfundamentals.org/webgl/lessons/zh_cn/

<http://www.webgl3d.cn/Three.js/>

T H A N K S

MAY YOU HAVE A COLOURFUL DAY !