Threejs可视化技术总结

threejs 可视化效果实现

- 热力图实现
  - 实现步骤
    1. 热力范围信息获取： 根据热力点分布矩形范围、热力点等级及热力半径大小构建热力矩形网格信息（左上右下角点坐标、中心点坐标和矩形宽高等信息）；
    2. 热力画布纹理构建： 参照上一步生成的矩形范围信息，将所有热力点坐标从世界坐标系转化为以矩形左上角点为原点的局部坐标系，然后使用heatmap.js将这些局部坐标系点集生成一张canvas画布，作为纹理贴图备用；
    3. 热力网格构建： 借助第一步生成的矩形信息生成一直同尺寸的矩形平面几何对象即可；
    4. 热力图生成： 用第2步生成的canvas作为网格材质及第三步生成的几何对象生成最终的热力图；
  - 注意事项： 热力范围要在原有热力点的基础上扩宽以防范围边界的热力显示不完整;
- 城市扫光实现
  - 实现原理： 使用动态材质的手段，在动画中更新材质属性，在顶点着色器中根据扫光形状计算扫光区域的颜色分布；
- 屏幕暗角实现
  - 实现原理: 使用离屏渲染的手段（指定WebGLRenderTarget）,对屏幕画面进行模糊处理再输出；
- 墙体生长特效
  - 实现原理:
    1. 墙几何：根据地面线顶点设置墙体的顶点和面，从而生成墙几何对象；
    2. 生长动画：将墙体的贴图从底部逐渐偏移搭配顶部位置，在requestAnimationFrame的回调方法里修改墙体高度，只贴墙体当前高度到地面的部分（等比），高度上方部分透明度设为0即可。
- 精灵图
  - 精灵图失真问题
  - 设置纹理的minFilter
  - .minFilter : number 当一个纹素覆盖小于一个像素时，贴图将如何采样。默认值为 THREE.LinearMipmapLinearFilter， 它将使用mipmapping以及三次线性滤镜。
  - .magFilter : number 当一个纹素覆盖大于一个像素时，贴图将如何采样。默认值为 THREE.LinearFilter， 它将获取四个最接近的纹素，并在他们之间进行双线性插值。 另一个选项是 THREE.NearestFilter，它将使用最接近的纹素的值。

    ```
    texture.minFilter = THREE.LinearFilter
    ```

- 过滤器参考

1. THREE.NearestFilter — 临近过滤：放大会导致方块化，缩小会丢失细节
2. THREE.LinearFilter — 线性过滤：放大会平滑很多
3. THREE.NearestMipMapNearestFilter — 选择最贴近目标解析度的纹理图片，然后使用最邻近过滤原则
4. THREE.NearestMipMapLinearFilter — 选择层次最近的两个纹理图片，在这两层之间使用最邻近过滤原则获取两个中间值，将这两个中间值传递给线性过滤器，以获得最终效果
5. THREE.LinearMipMapNearestFilter — 选择最贴近目标解析度的纹理图片，然后使用线性过滤原则
6. THREE.LinearMipMapLinearFilter — 选择层次最近的两个纹理图片，在这两层之间使用线性过滤原则获得两个中间值，将这两个中间值传递给线性过滤器，以获得最终结果

- 主建筑标注
  - 采集方法
    1. 主建筑绑定了双击进入的方法
    2. 方法中调用`getObject3DRangeInfo`(ThreeBussiness.js中的方法)获取模型中心坐标
    3. 双击点击建筑顶部获取的中心点即可作为标注的坐标点

- 透明物体的渲染
  - 绘制透明物体时，关掉深度测试才能保证正确的遮挡关系， depthTest = false
  - 透明渲染次序，three的渲染器是基于webGL的。它的渲染机制是根据物体离照相机的距离来控制和进行渲染的。对于透明的物体，则是按照从最远到最近的顺序进行渲染。也就是说，它根据物体的空间位置进行排序，然后根据这个顺序来渲染物体 1.renderer.sortObjects = false; 这样就是按照其添加到场景的先后顺序渲染。 2.renderer.sortObjects = true;并指明需要先渲染的物体的渲染顺序（object.renderOrder）。 3.不透明的物体会优先渲染。

- `z-flighting`问题,例如模型重叠面不停闪烁
  - 概念:当场景中的两个模型在同一个像素生成的渲染结果对应到一个相同的深度值时，渲染器就不知道哪个面在前，哪个面在后，于是便开始任意而为，这次让这个面在前面，下次让那个面在前面，于是模型的重叠部位便不停的闪烁起来.
  - 解决方案
  - 相机设置合适的near值以及far值, 用来设置相机的近平面和远平面,这两个参数与z-buffer密切相关,深度缓冲其实是非线性的，靠近相机的地方精度更高,例如深度缓冲有10个级别,near值为1,far值为100.则23.3484到99.9999为同一级别,那两个面对应到同一个深度级别则可能出现f-flighting现象.选择一个较大的near值可以较为明显的发现场景重叠闪烁问题少了.例如从0.01改为1.

- 相机环绕动画
  - 相机环绕使用tweeenjs做补件动画使得环绕更加顺滑
  - 相机与聚焦物体的角度和距离影响相机捕获的画面
  - 难点
    1. 相机环绕动画增加偏移角度后如何使得平移跟环绕衔接顺滑
    2. 偏移角度分为**俯视角度**和**环绕偏移角度**

3. 俯视角度: 相机与物体所在直线和物体所在平面的角度所夹角度
4. 环绕偏移角度: 相机做环绕运动开始时, 相机与物体所在直线与坐标轴所夹角度
5. 偏移角度, 环绕偏移角度在计算平移和环绕运动衔接点时相当重要, 在对应的轴上都需要加上对应的偏移值

- 待优化的地方
  1. 目前环绕分成固定分成4段环绕, 需要考虑抽离段数作为参数, 但是目前抽离后会有相关的计算结果出现**NaN**问题导致在某些场景下相机捕获的画面为黑屏
  2. 已经优化为自定义段数环绕, **NaN**的问题为三角函数计算传入不正确的参数导致

- `POI搜索`
  - 难点 火星坐标 =》 WGS84 => 世界坐标
  - 用 coordtransform完成火星坐标到wgs84的转换，用PRO4完成WGS84到世界坐标的转换。
  - 期间出现了以为是坐标转换导致的偏移问题，实际是由于道路和矢量面都做了X轴的翻转，但是MARKER并没有翻转导致的。

threejs 与vue集成

- `非数据的不挂载在data下，减少不必要的性能开销`
  - 例如scene,renderer等
  - threejs 中关于场景的数据尽量避开使用Vue的响应式数据处理, 如data/prop等
- `在页面销毁事件前销毁以及释放资源`
- beforeDestroyed() { this.scene.dispose(); this.controls.dispose(); const gl = this.renderer.domElement.getContext('webgl'); gl && gl.getExtension('WEBGL_lose_context').loseContext(); }

数据资源文件加载策略（例如json）

- 一开始就需要使用的数据
  - 数据量很大的文件（> 10M）
    1. 考虑**文件拆分**, 浏览器走网络请求加载一个大文件的效率不如加载几个小文件
    2. 使用浏览器的**缓存机制**, 使用协商缓存从硬盘中读取缓存文件, 减少网络数据传输
  - 数据量不算大的文件（< 1M）
    1. 可以采用**本地引入**数据的方式, 如webpack中的json-loader解析数据
- 根据功能需要请求回的数据
  1. 通过网络请求一些不经常改变的数据可以在首次请求成功之后使用持久化存储的方式缓存数据, 如浏览器的`localStorage`(注: Storage存储大小约为5MB, 考虑是否可以使用这种方式缓存以及需要合理规划缓存数据的大小)
  2. 数据大小如果不大, 也可以考虑使用本地引入数据的方式
- 矢量数据过于庞大，读取市区json太耗时，考虑后台传回
  1. 传显示范围给后台，后台根据需要显示的范围以及显示级别对点/线/面优化，并返回需要加载的数据给前端。（想法，暂未落地）

threejs 数据优化

- 按照数据要求合并obj数据，例如分层展示的则按照层合并obj数据，减少loader
- 使用draco压缩后的gltf格式的模型代替obj格式的模型
- threejs官方推荐gltf格式。
- ***模型出现部分面缺失问题，未开双面渲染导致

threejs 性能优化

- 用group加载mesh
    - 加载同类型大量mesh的时候.同时添加一group, 其性能比分别添加同样的mesh要更高, 另外可以避免mesh过多时, 浏览器会因为webgl context数量过多, 而显示警告:WARNING: Too many active WebGL contexts. Oldest context will be lost
- 使用buffergeometry代替geometry
    - BufferGeometry 会缓存网格模型，可以有效减少向 GPU 传输顶点位置，面片索引、法相量、颜色值、UV 坐标和自定义缓存属性值等数据所需的开销
    - 详细原理如下：1、Geometry 生成模型-> (CPU 进行数据处理，转化成虚拟3D数据) -> (GPU 进行数据组装，转化成像素点，准备渲染) -> 显示器 , 第二次操作时重复走这些流程。 2、BufferGeometry 生成模型 -> (CPU 进行数据处理，转化成虚拟3D数据) -> (GPU 进行数据组装，转化成像素点，准备渲染) -> (丢入缓存区) -> 显示器, 第二次修改时，通过API直接修改缓存区数据，流程就变成了这样 -> (CPU 进行数据处理，转化成虚拟3D数据) -> (修改缓存区数据) -> 显示器, 节约GPU运算性能。
    - 实际遇到的案例：矢量拉伸模型用extrudegeometry加载三十平方公里的矢量数据并拉伸需要20s, 而应用extrudeBuffergeometry只需要8s.
- 大量mesh绘制导致速度变慢和内存暴增，通常的解决问题做法是使用多个实例化渲染和合并几何图形。
    - 内存换效率，合并几何图形
    - 原理：
    - 实际遇到的案例：甘肃武威白膜未采取合并几何图形前加载三十平方公里立方体帧率为17fps,加载耗时为7.55s, BufferGeometryUtils.mergeBufferGeometries,采取合并几何图形，帧率为56fps。
- 多个实例化渲染
- 参考文章：https://learnopengl-cn.readthedocs.io/zh/latest/04%20Advanced%20OpenGL/10%20Instancing/
- 原理：实例化是一种只调用一次渲染函数却能绘制出很多物体的技术，它节省渲染物体时从CPU到GPU的通信时间，而且只需做一次即可，适用于场景大量重复的物体，比如树，粒子等。
- glDrawArrays或glDrawElements这样的函数(Draw call)过多。这样渲染顶点数据，会明显降低执行效率，这是因为OpenGL在它可以绘制顶点数据之前必须做一些准备工作(比如告诉GPU从哪个缓冲读取数据，以及在哪里找到顶点属性，所有这些都会使CPU到GPU的总线变慢)。所以即使渲染顶点超快，而多次给GPU下达这样的渲染命令却未必。
- 渲染的对象时side属性尽量用单面渲染，双面渲染会导致更多的渲染
- 类似于地图或者地板类的mesh只需要单面渲染即可，毕竟背面也看不到。

- 谨慎地在`render()`中操作，`fps`为`60`则代表一秒执行`60`次，如果在此有赋值或者实例化的操作容易导致崩溃，不需要持续在整个场景中都执行的动画则增加一个布尔值判断
- 共享材质
- 相同类型的精灵图对url到texture这一步骤做缓存, 共用texture
- 纹理图片尽可能是2的幂次方，尽可能小。
    - 非2的幂次方贴图在WebGL的支持是有限的。显卡渲染的要求也是2^n, 图片的纹理像素在threejs、Unity3D等3d引擎中需要遵循2的N次方。非2的N次方的图片会转化为2的N次方图片（500 x 500 → 512 x 512）,是因为转化过程比较慢，由运行程序转换十分耗时，所以threejs、Unity3D等引擎提前将资源转化为符合标准的图片。故尽量加载2的幂次方的纹理图片。
- 纹理支持SVG格式
- `THREE.TextureLoader().load(url);` url 可以直接是一个svg后缀的图片地址; SVG的优势在于能够很好的处理图形大小的改变。
- 每一次动画里的`requestAnimationFrame`都要把`id`保存下来，在组件销毁时`cancelAnimationFrame`掉， 尚未开展待优化

threejs 浅谈object3d的visible属性

- 可以用group的visible控制组内所有object3d的显示；
- 组内所有object3d的visible属性并不会根据group的visible改变而进行调整，推测可能是防止组内对象过多，遍历设置子对象属性会引起性能问题；
- 使用射线法对mesh对象进行拾取时，即使mesh对象不可视（其group的visible为false时），intersectObjects方法也能检测到这些mesh对象；
- 注：遇见一个奇怪的问题，group设为true没法显示所有子对象，必须traverse每一个子对象才行；group设为false却能关闭所有子对象。