

是的，可以在当前代码的基础上实现让 Chunk 壳的组织类保持不动，而通过相机在场景内实际移动来实现无限循环的城市效果。以下是对问题的分析以及具体的实现步骤和代码修改建议。

---

## 当前系统的工作原理

当前的无限循环城市效果通过以下机制实现：

### <sup>1</sup> 场景划分与 Chunk 壳管理：

- 场景被划分为多个 Chunk，每个 Chunk 包含城市的一部分。
- 这些 Chunk 被组织在一个网格中，通过 Chunk 壳（例如 Application 类中的 chunks 数组）管理。
- Chunk 壳会根据相机的“虚拟移动”动态调整位置，加载或卸载相应的 Chunk。

### <sup>2</sup> 相机固定，场景移动：

- 相机位置固定（例如通过 Zk 类中的 this.\_camera）。
- 用户拖动时，通过调整整个场景（this.\_scene.position）的位置来模拟相机的移动。
- 在 Zk 类的 update 函数中，\_worldOffset 和 point.lerp 用于平滑调整场景位置。

### <sup>3</sup> 无限循环机制：

- 当场景移动到网格边缘时，系统重新排列 Chunk 壳，将新的 Chunk 加载到视野内，旧的 Chunk 移出视野。
- 这通过 raycast 和 move 事件触发，确保场景始终围绕相机更新。

这种方法在视觉上等同于相机移动，但实际是场景在动。

---

## 修改目标：相机移动，Chunk 壳不动

要改为相机实际移动，同时保持 Chunk 壳的组织类不动，需要以下调整：

### <sup>1</sup> 相机控制：

- 让相机根据用户输入在场景中移动，而不是固定。

- 用户拖动时更新相机位置，而不是场景位置。

## 2 Chunk 管理:

- Chunk 壳保持固定位置（即 Application 类中的 chunks 网格位置不变）。
- 根据相机的新位置动态加载和卸载 Chunk，确保相机视野内始终有连续的城市景观。

## 3 无限循环:

- 通过计算相机在固定 Chunk 网格中的位置，确定需要加载哪些 Chunk，卸载超出视野的 Chunk。
- 利用现有网格结构，实现无缝的循环效果。

这种方法在当前代码基础上是可行的，但需要对相机控制和 Chunk 管理逻辑进行修改。

---

## 实现步骤

以下是具体的实现步骤：

### 1. 修改相机控制逻辑

当前在 Zk 类（模块 57）中，用户拖动通过 \_onDrag 更新 \_offset，最终调整 this.\_scene.position。需要改为更新相机位置：

- 修改 \_onDrag 函数：
  - 将拖动偏移量应用到 this.\_camera.position，而不是 this.\_scene.position。
  - 确保相机沿水平平面（x-z 平面）移动，高度（y 轴）保持不变或由其他逻辑控制。
- 移除场景移动逻辑：
  - 删除 this.\_scene.position.addVectors(this.\_sceneOffset, point)，因为场景不再移动。

代码修改示例：

---

javascript

```
// 在 Zk 类（模块 57）中
```



```

_onDrag: function() {
    var vector = new THREE.Vector2;
    return function(planeOrigin) {
        if (this.enabled && this._panning) {
            vector.subVectors(planeOrigin, this._startCoords);
            this._offset.addVectors(this._lastOffset, vector);
            // 计算相机的新位置
            var offset = this._offset.clone();
            offset.rotateAround(new THREE.Vector2(), -Math.PI / 4); // 保持原有旋转
            var worldOffset = new THREE.Vector3(offset.x, 0, offset.y).multiply(
                this._worldOffset.lerp(worldOffset, 0.05)); // 平滑移动
            this._camera.position.x += this._worldOffset.x;
            this._camera.position.z += this._worldOffset.z;
        }
    };
}(),
update: function() {
    // 移除 raycast 和场景移动逻辑，只保留相机更新
    this._camera.update(); // 调用相机自身的 update (如有)
}

```

## 2. 调整 Chunk 加载逻辑

由于 Chunk 壳不再移动，需要根据相机位置动态管理 Chunk：

- 计算相机所在的 Chunk：

使用相机的世界坐标（`this._camera.position`），计算其在 Chunk 网格中的索引。

- 例如，`chunkX = Math.floor(camera.position.x / GConfig.CHUNK_SIZE)`。

- 加载周围 Chunk：

- 根据相机位置，加载当前 Chunk 及其周围的 Chunk（例如 3x3 网格）。
- 使用类似 `_forEachNeighboringChunk` 的逻辑，但基于相机位置而不是场景偏移。

- 卸载超出视野的 Chunk：

- 维护一个可见 Chunk 列表，移除超出相机视野范围的 Chunk。

### 代码修改示例：

在 Application 类（模块 47）中添加动态加载逻辑：

---

javascript

✖️ ⌂ ▶️ ↻

```
updateChunks: function(camera) {
```

```

var chunkX = Math.floor(camera.position.x / GConfig.CHUNK_SIZE);
var chunkZ = Math.floor(camera.position.z / GConfig.CHUNK_SIZE);
var visibleChunks = [];

// 加载相机周围的 Chunk (例如 3x3 网格)
for (var i = chunkX - 1; i <= chunkX + 1; i++) {
    for (var j = chunkZ - 1; j <= chunkZ + 1; j++) {
        var chunk = this.getOrCreateChunkAt(i, j);
        visibleChunks.push(chunk);
        if (!this.children.includes(chunk)) {
            this.add(chunk);
        }
    }
}

// 卸载不可见 Chunk
this.children.forEach(child => {
    if (child.name === "chunk" && !visibleChunks.includes(child)) {
        this.remove(child);
    }
});
}

getOrCreateChunkAt: function(x, z) {
    // 标准化索引到 TABLE_SIZE 内
    var tableX = THREE.Math.euclideanModulo(x, GConfig.TABLE_SIZE);
    var tableZ = THREE.Math.euclideanModulo(z, GConfig.TABLE_SIZE);
    if (!this.chunks[tableX]) this.chunks[tableX] = [];
    if (!this.chunks[tableX][tableZ]) {
        this.chunks[tableX][tableZ] = this._createChunkAt(tableX, tableZ);
    }
    // 调整 Chunk 位置到世界坐标
    var chunk = this.chunks[tableX][tableZ];
    chunk.position.x = x * GConfig.CHUNK_SIZE;
    chunk.position.z = z * GConfig.CHUNK_SIZE;
    return chunk;
}

```

### 3. 集成到主循环

在主场景类（例如 Scene 或 Application）的 update 函数中调用 updateChunks：

---

```
update: function(data) {  
    this.updateChunks(this.camera); // 假设 camera 是场景属性  
    // 其他更新逻辑  
}
```

## 4. 确保无缝过渡

- 预加载：
  - 在相机接近 Chunk 边界时预加载相邻 Chunk，避免加载延迟。
- 平滑效果：
  - 可选地在加载新 Chunk 时添加淡入效果（修改材质透明度）。

## 5. 性能优化

- Chunk 重用：
  - 不直接销毁超出视野的 Chunk，而是将它们移动到新位置并更新内容。
  - 修改 getOrCreateChunkAt 以重用现有 Chunk 对象。
- 高效管理：
  - 使用对象池或缓存机制，减少频繁的内存分配和释放。

代码示例（重用机制）：

```
getOrCreateChunkAt: function(x, z) {
```

```
var tableX = THREE.Math.euclideanModulo(x, GConfig.TABLE_SIZE);
var tableZ = THREE.Math.euclideanModulo(z, GConfig.TABLE_SIZE);
var chunk = this.chunks[tableX][tableZ];
if (!chunk) {
    chunk = this._createChunkAt(tableX, tableZ); // 新建或从池中获取
    this.chunks[tableX][tableZ] = chunk;
}
chunk.position.x = x * GConfig.CHUNK_SIZE;
chunk.position.z = z * GConfig.CHUNK_SIZE;
return chunk;
}
```

## 具体代码修改建议

### Zk 类 (模块 57)

- 修改 `_onDrag` 和 `update`，如步骤 1 所示。
- 删除 `raycast` 和 `move` 事件触发，因为 `Chunk` 更新交由 `Application` 处理。

### Application 类 (模块 47)

- 添加 `updateChunks` 和 `getOrCreateChunkAt` 函数。
- 修改 `_initChunks` 为初始化固定网格，但不预先添加所有 `Chunk` 到场景：

javascript



```
_initChunks: function() {
    this.chunks = [];
    for (var i = 0; i < GConfig.TABLE_SIZE; i++) {
        this.chunks[i] = [];
    }
}
```

## 主场景类

- 确保 `update` 中调用 `updateChunks`，传入当前相机对象。

## 总结

通过以上修改，可以实现相机在场景内移动而 `Chunk` 壳组织类保持不动的无限循环城市效果。主

要变化包括：

- 将拖动逻辑从场景移动改为相机移动。
- 根据相机位置动态加载和卸载固定网格中的 Chunk。
- 利用现有网格结构和加载机制，确保无缝体验。

这种方法保留了代码的核心架构，只需调整控制和加载逻辑即可实现目标，同时保持性能和用户体验的平滑性。