osgChina.org

# 虚幻引擎特效在OSG场景中的使用方案

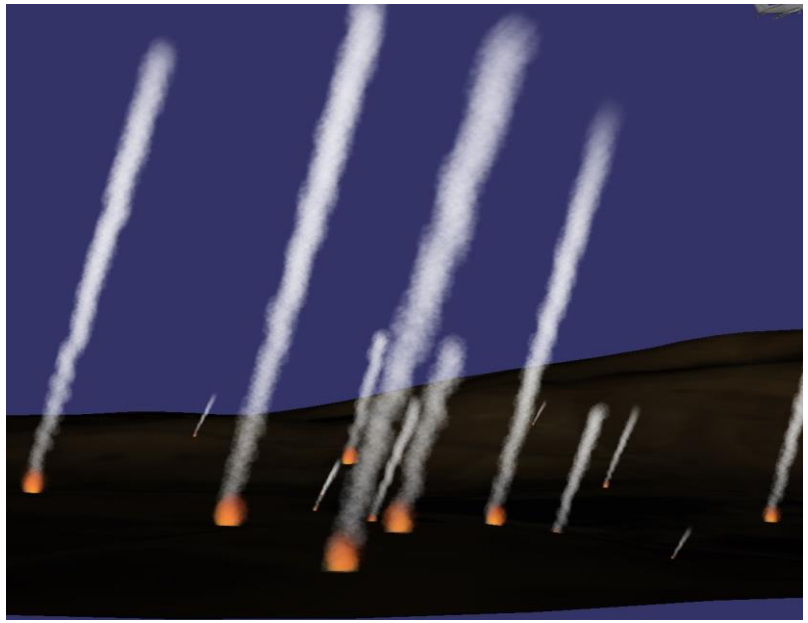
恒歌科技 杨石兴

# 动 机

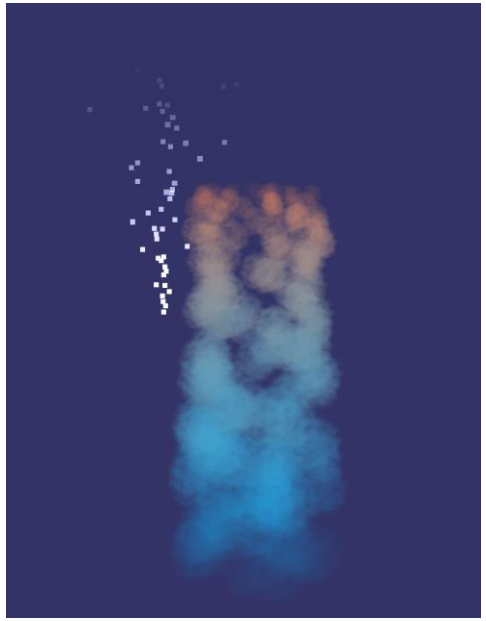
- 在航空航天，军工，安防等业务场景中，类似尾焰，尾烟，武器特效，爆炸，火焰等粒子效果的应用十分广泛。

- 高质量的粒子特效是提升三维场景表现力的重要因素。

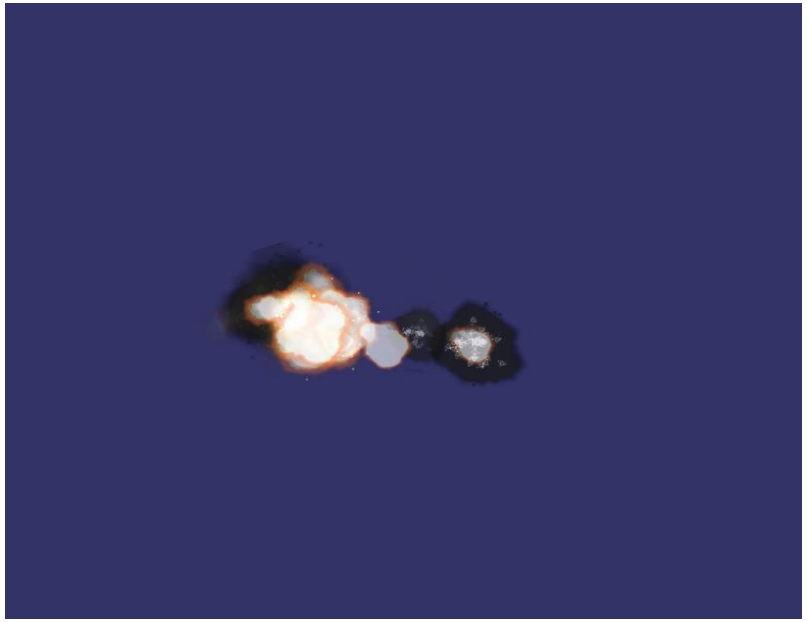
# OSG粒子效果



火焰

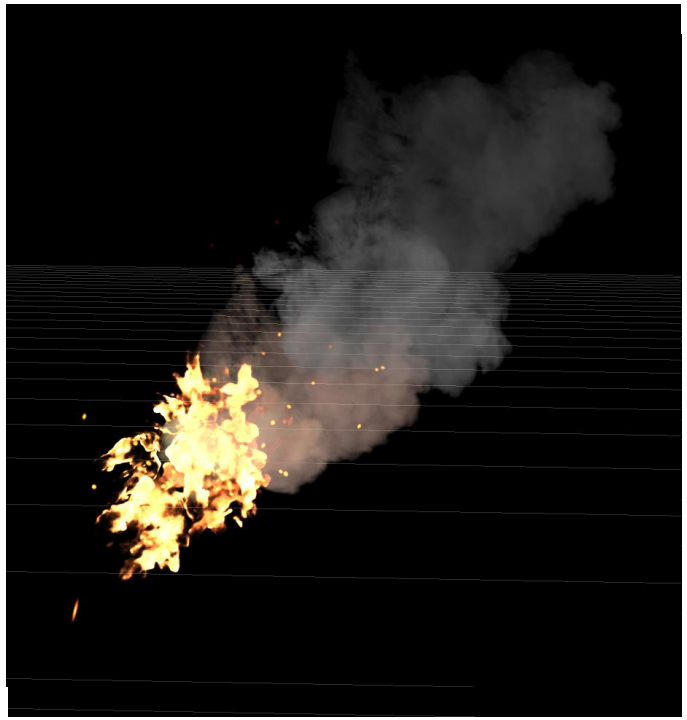


烟花

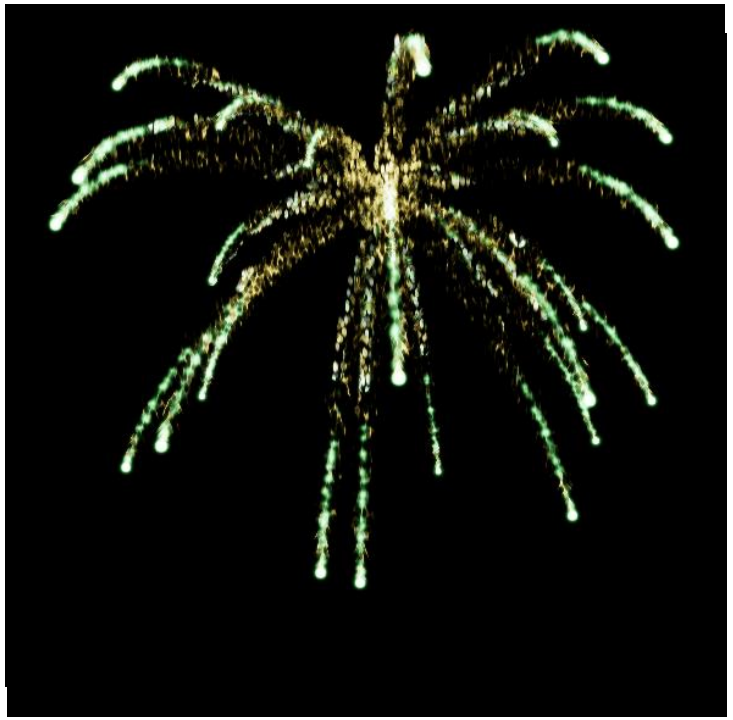


爆炸

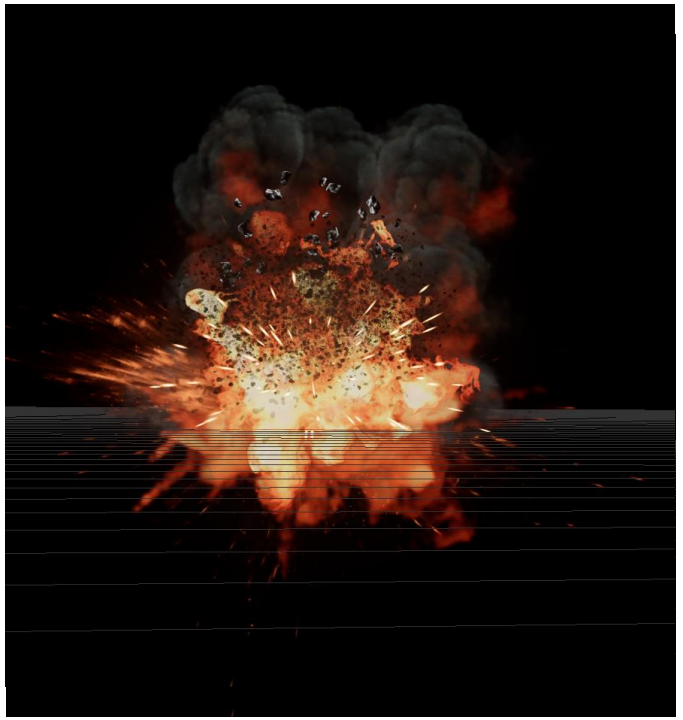
# UE4粒子效果



火焰



烟花



爆炸

# 技术路线

- 直接基于UE4开发

  优点：开发周期短

  缺点：UE4引擎过于庞大，后期维护成本高

- 移植UE4粒子系统到OSG中

  优点：保留了OSG的开发习惯，降低了二次开发人员的学习成本

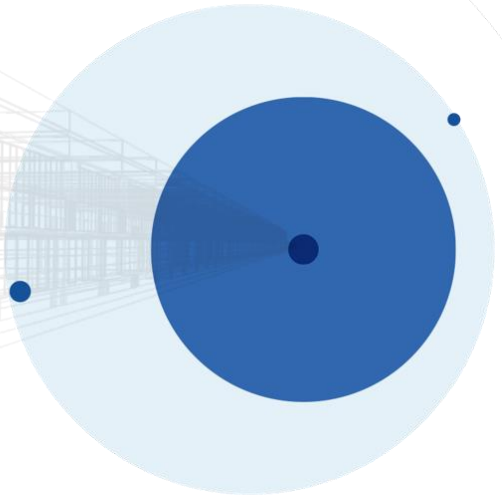
  缺点：前期投入周期长，成本高

# 设计目标

- UE4仅作为特效制作工具，渲染时保持OSG的独立性，不对UE4有任何依赖。
- 需将UE4粒子系统的渲染流程转换为OSG的渲染流程，并在结构上相融合。
- 充分利用OSG现有的扩展机制，使新增的代码都以独立模块的形式添加其中。
- 在不影响效率的前提下，最大限度避免修改OSG源码。

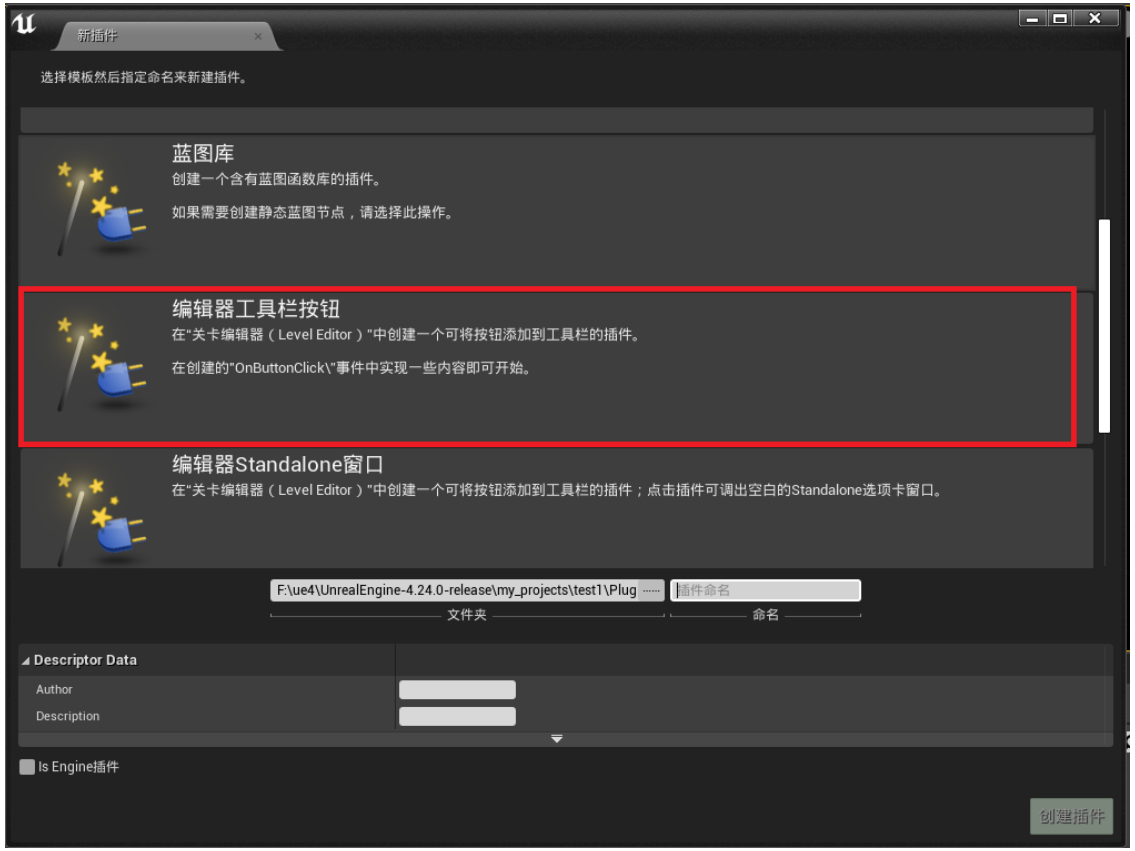
# 结 构

- UE4导出插件
- osgDB读取插件
- FeParticle渲染模块



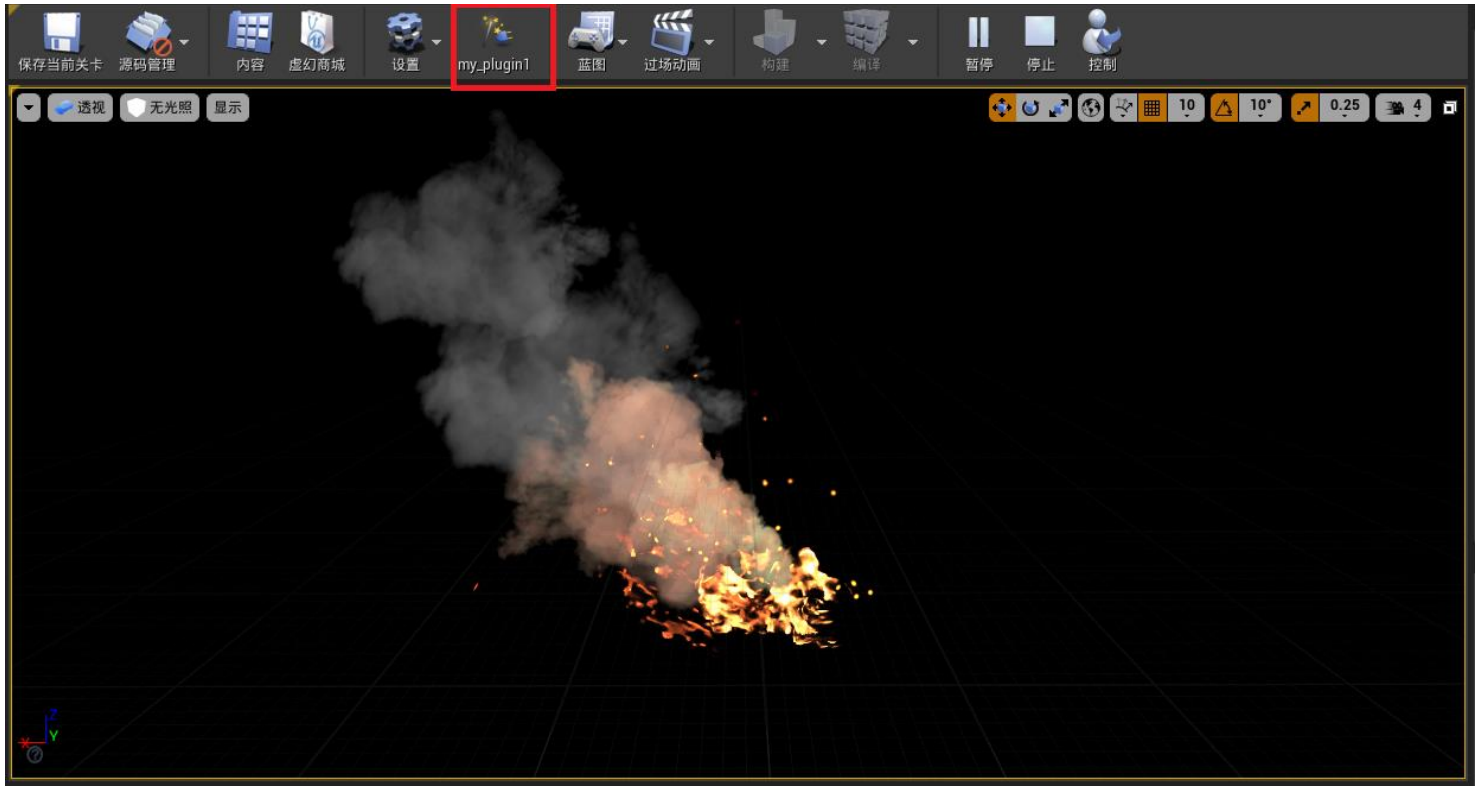
# UE4导出插件

- 新建工具栏插件



# UE4导出插件

- 保存特效



# UE4导出插件

- 写入数据到.hgps文件

```
void Fmy_plugin1Module::PluginButtonClicked()
{
    CWriteParticleSystem writer(&psArr, &levelSet);
    if (writer.WriteParticleSystemToFile())
    {
    }
    else
    {
    }
}
```

# osgDB读取插件

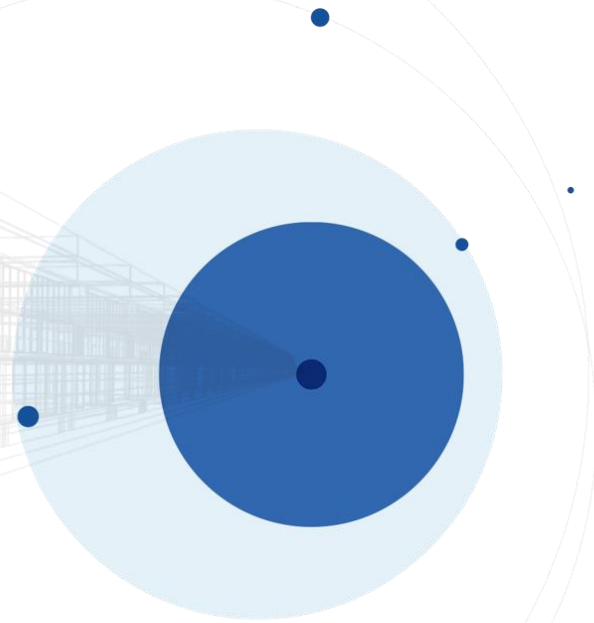
```
class UEReaderWriter : public ReaderWriter
{
    public:
        UEReaderWriter():_wrappersLoaded(false)
        {
            supportsExtension("hgps","hg particleSystem from ue4");
        }

        virtual ReadResult readNode(const std::string& file, const Options* opt)
        const
        {
            //readFile
            ……
        }
}

REGISTER_OSGPLUGIN(osg, UEReaderWriter)
```

# FeParticle渲染模块

- FeParticleManager
- FeEmitterActor

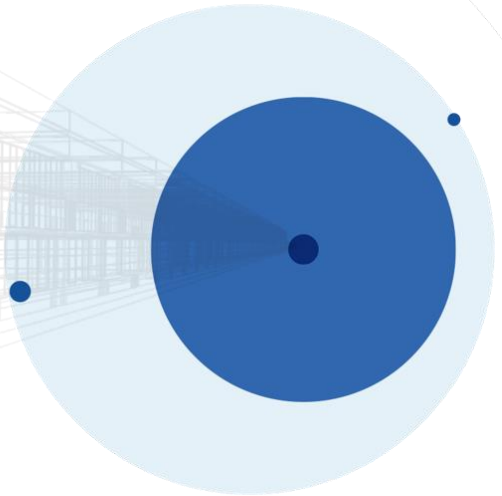
# FeParticleManager

- 公共资源的管理
- GPU粒子类型支持



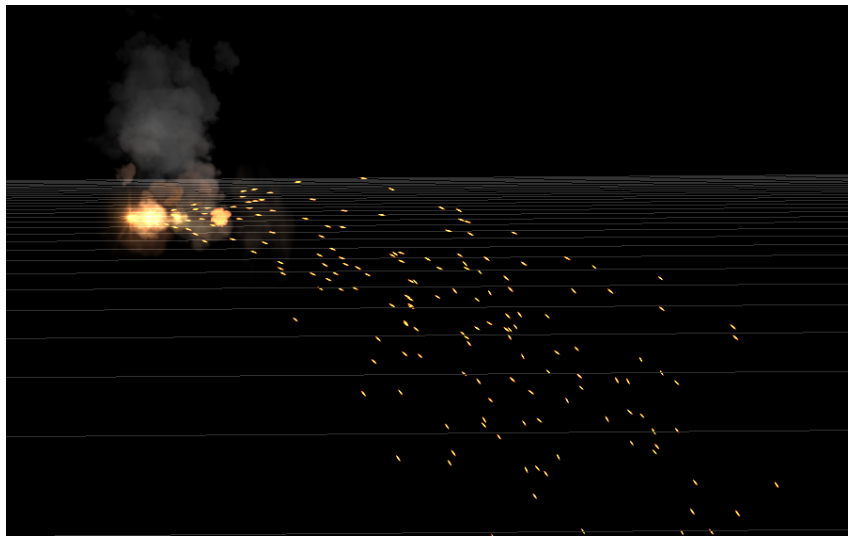
# 公共资源的管理

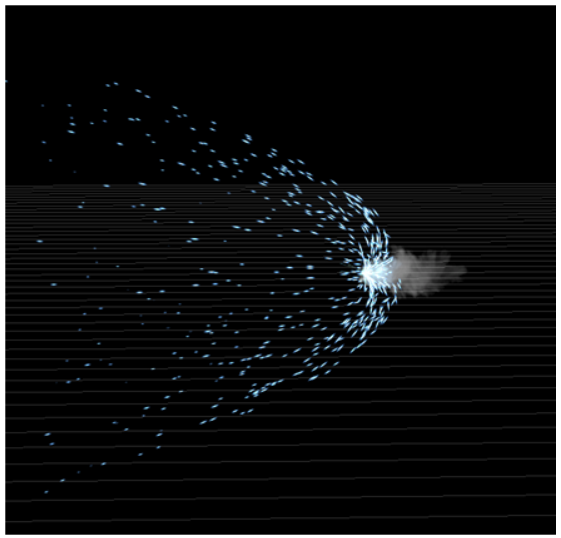
- UE4引擎公共参数，公共材质的保存
- 全局资源池的管理（动态顶点缓存池，动态索引缓存池...）
- 着色器管理



# GPU粒子类型支持

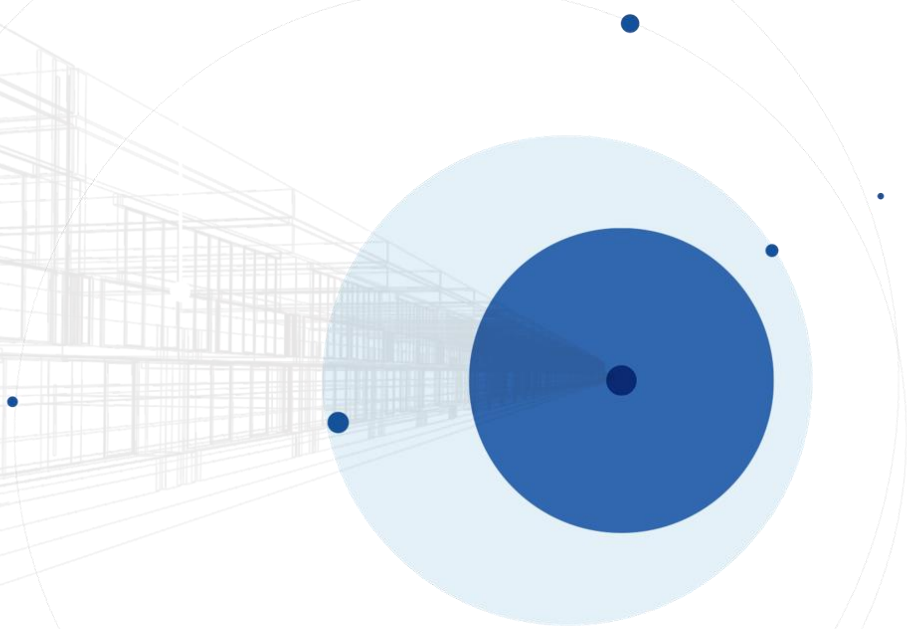
UE4中的GPU粒子类型，官方描述支持上万粒子的高效渲染，在UE4渲染层通过FFXSystem来统一更新粒子属性。在OSG端我们通过FeParticleManager来管理





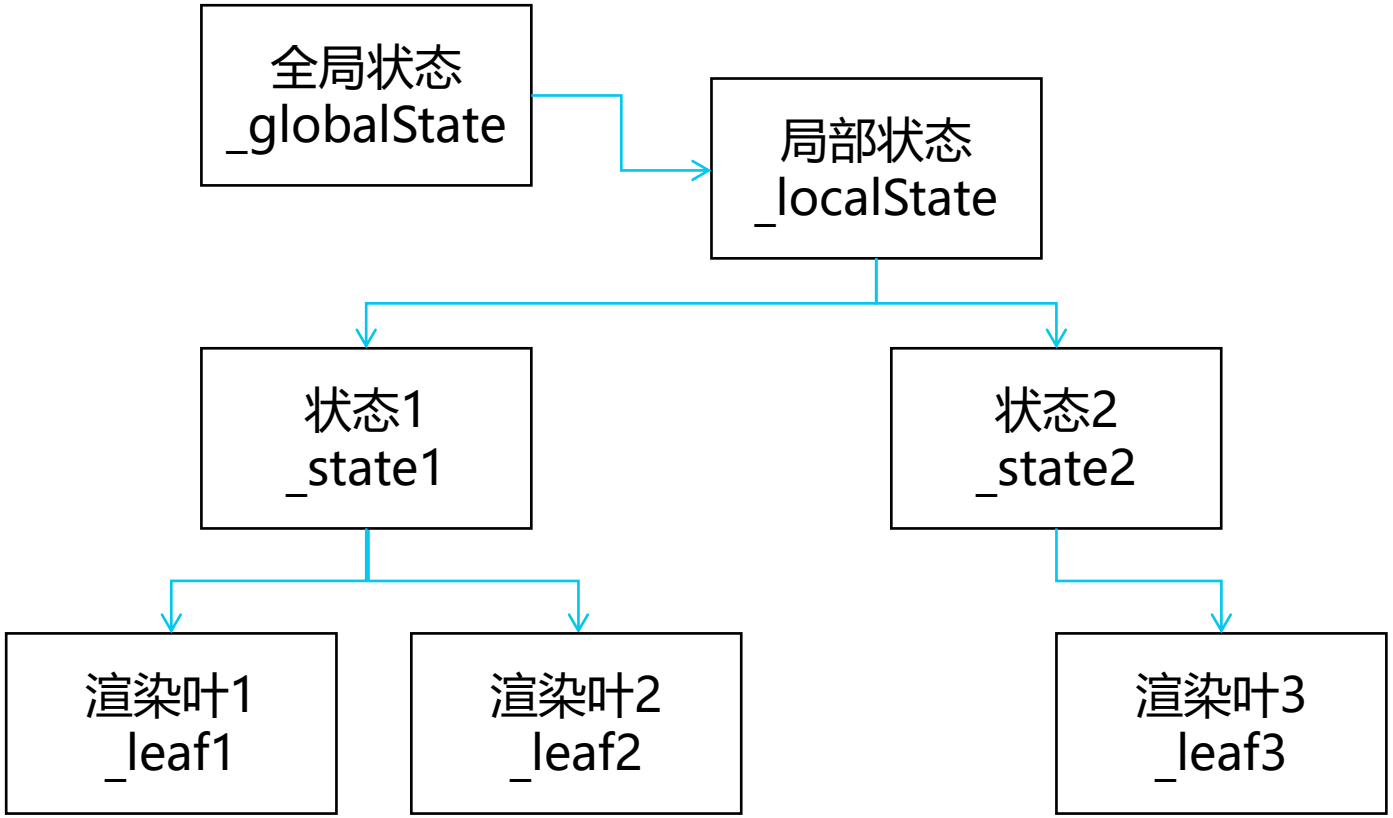
# FeEmitterActor

- OpenGL状态设置
- 渲染粒子特效到纹理
- 后处理
- 将最终图像渲染到主场景



# OpenGL状态设置

- OSG通过状态树来防止OpenGL状态频繁切换



# OpenGL状态设置

在FeEmitterActor内部，由于需要手动调用大量OpenGL接口，所以我们用FOpenGLContextState来管理

```
struct FOpenGLContextState
{
    GLenum activeTexture；//当前纹理
    GLuint  program；          //当前着色器
    FRect    viewport；        //当前视口
    ….                              //其他属性
}

void bindArrayBuffer（FOpenGLContextState & ContextState，Gluint  newBuffer)
{
     if（ContextState .buffer != newBuffer）
     {
          glBindBuffer(GL_ARRAY_BUFFER, newBuffer);
          ContextState .buffer = newBuffer；
     }
}
```

# OpenGL状态设置

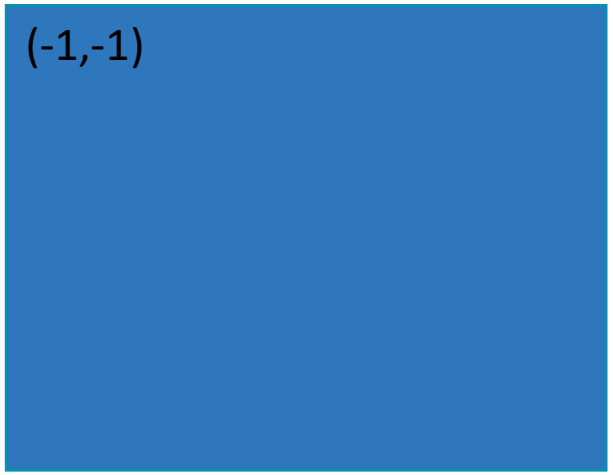
- 渲染前保存OpenGL状态，渲染后恢复
- 渲染前重置粒子特效状态集

```
void CFeEmitterActor::renderThreadUpdate(osg::RenderInfo& renderInfo)
{
	//preRender
	GDynamicRHI->RHIPushAllState();
	GDynamicRHI->RHIResetContextState();

	//render
	.....

	//postRender
	GDynamicRHI->RHIPopAllState();
}
```

# 渲染粒子特效到纹理

- 在裁剪坐标系下，OSG默认原点在左下角，UE4在左上角

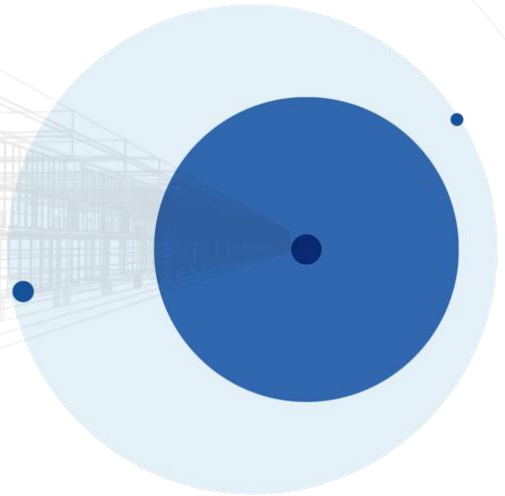

OSG : LOWER_LEFT




UE4 : UPPER_LEFT

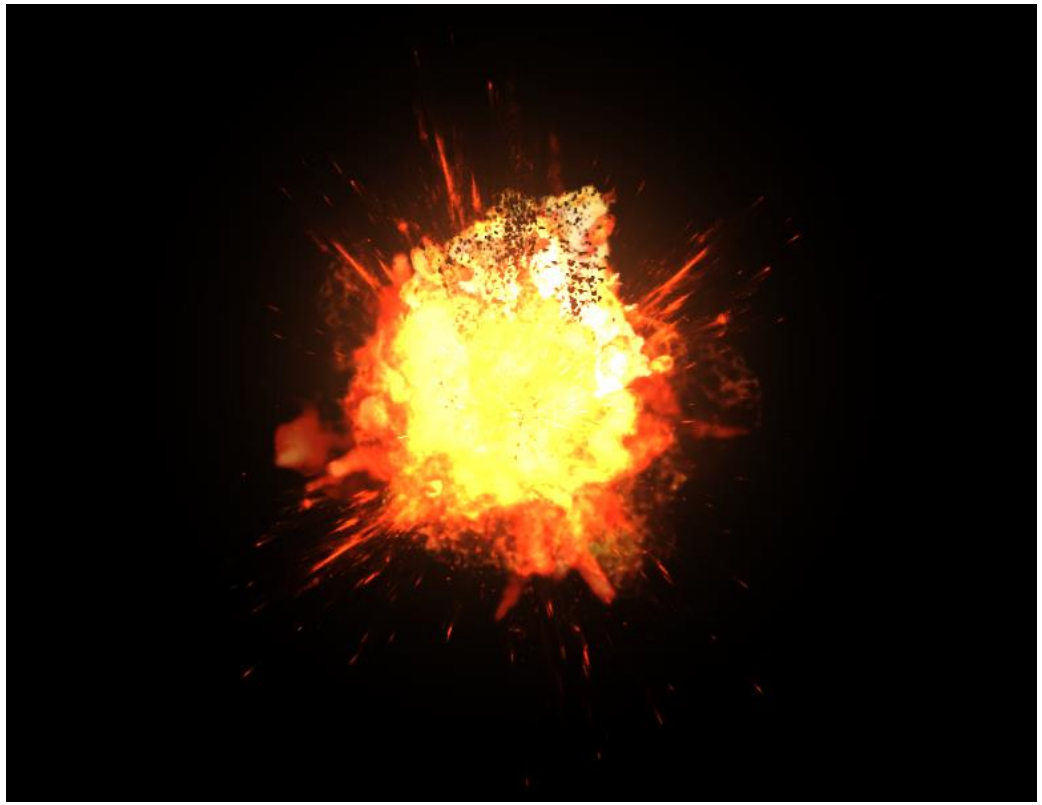
# 后处理

- 原始图像
- EyeAdaptationTexture图像
- Bloom图像
- ColorGradingTexture图像
- 最终图像

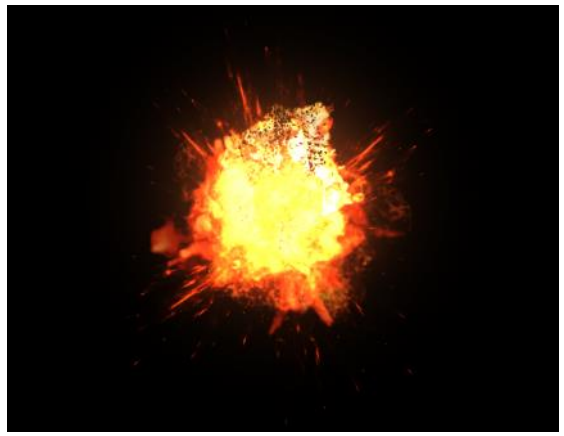


# 原始图像

- ParticleAndSceneColor = AddParticleAddScenePass(contextID, PassInputs);



# EyeAdaptationTexture



HalfSceneColor

+

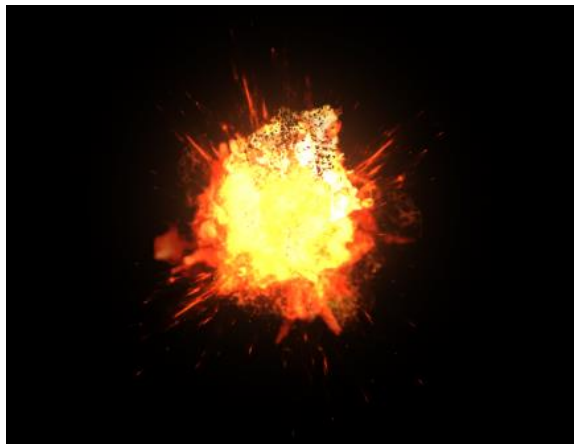
HistogramTexture

=



EyeAdaptationTexture

# Bloom



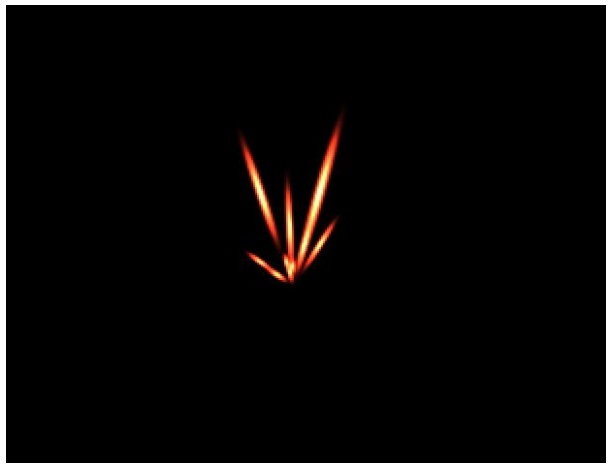
HalfSceneColor

+



EyeAdaptationTexture

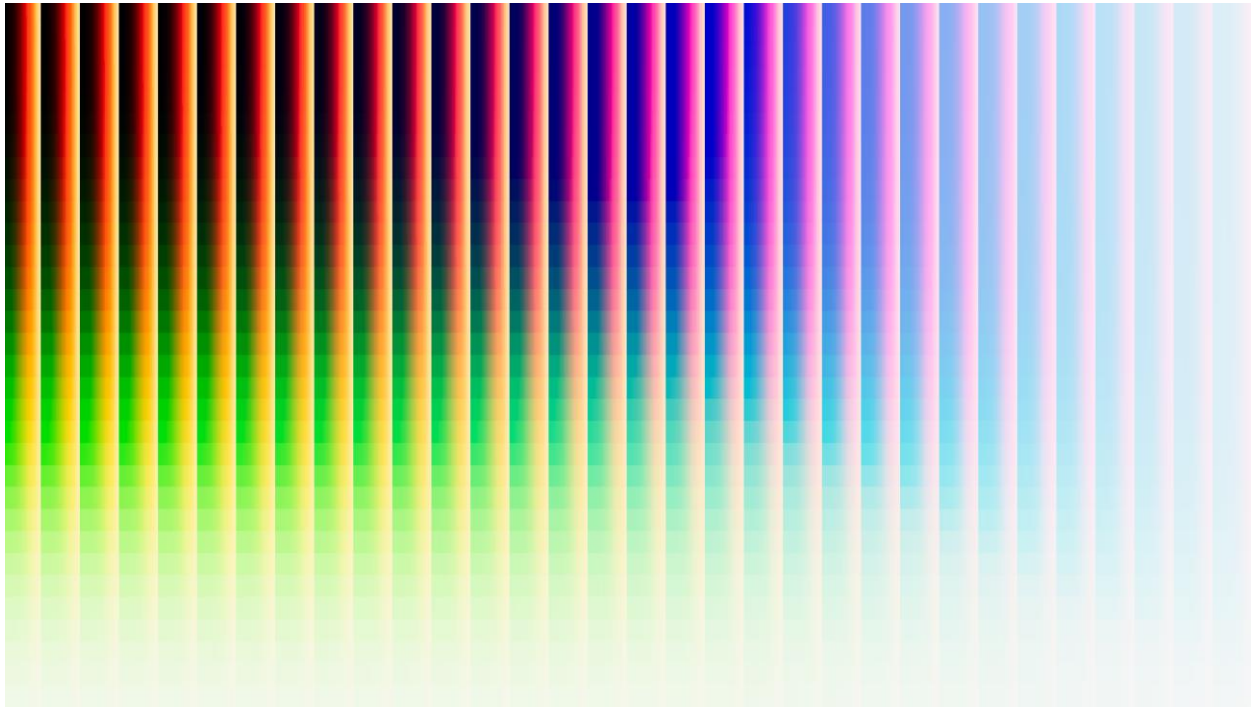
=



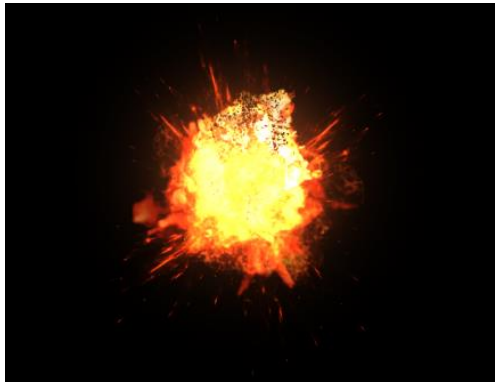
Bloom

# ColorGradingTexture

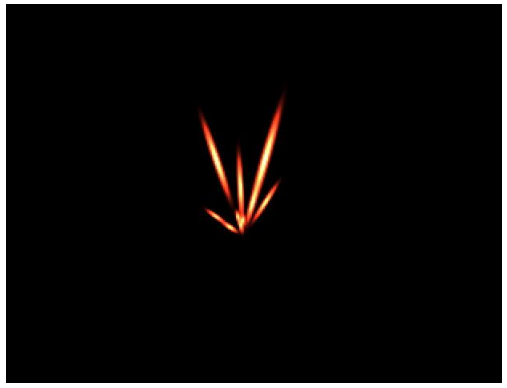
ColorGradingTexture = AddCombineLUTPass(contextID,pActor);



# 最终图像



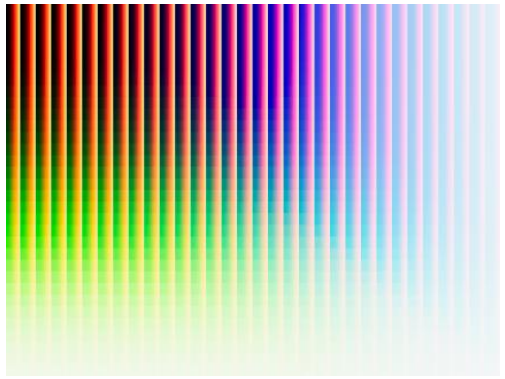
SceneColor



Bloom



EyeAdaptationTexture

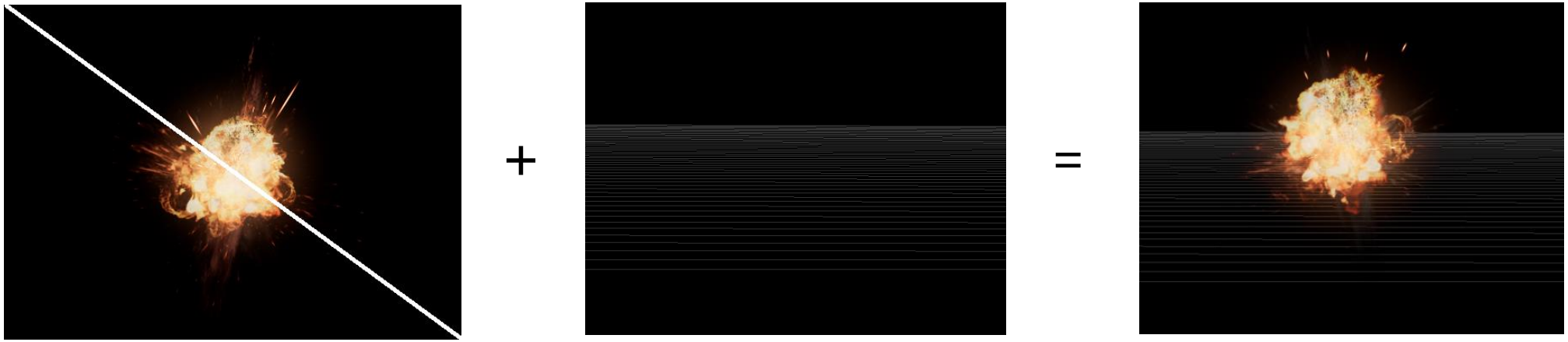


ColorGradingTexture

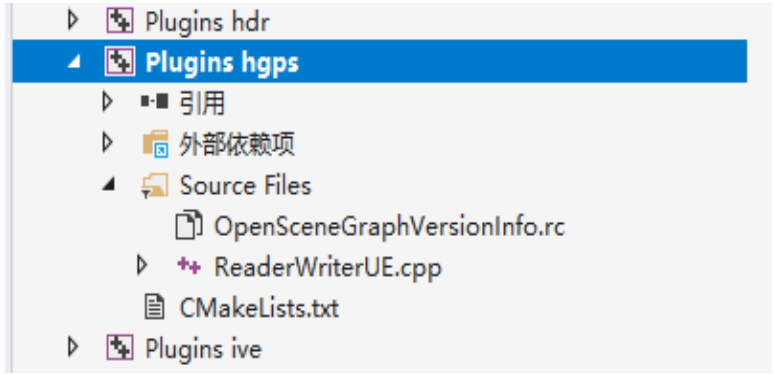
FinalTexture

# 将最终图像渲染到主场景

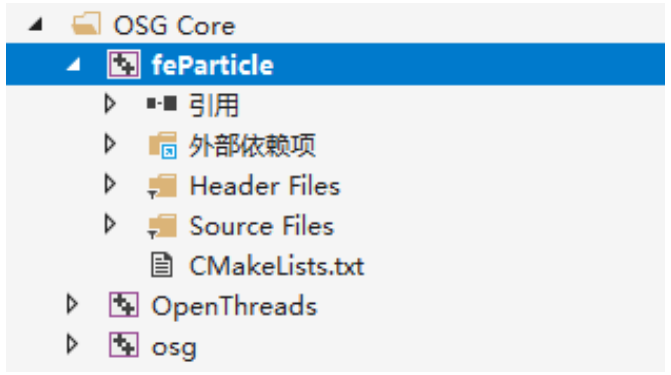
最后通过创建与主场景大小相同的四边形，将粒子图像与场景进行融合



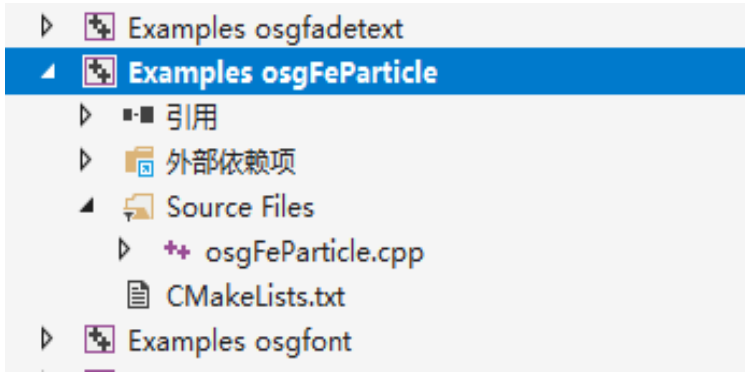
# 目录结构




osgDB读取插件




feParticle渲染模块




feParticle使用示例

# 代码示例

创建结点 ⟶

```
std::string path = "feParticle_data\\";
std::string fileName = "P_Sparks.hgps";
std::string fileName1 = "P_Fire.hgps";

osg::Node* pActor = osgDB::readNodeFile(path +
fileName);
osg::Node* pActor1 = osgDB::readNodeFile(path
+ fileName1);
```

# 代码示例

设置矩阵 ⟶

```
osg::Matrix scaleMat;
double fScale = 10;
scaleMat.makeScale(osg::Vec3d(fScale, fScale, fScale));

osg::Matrix rotateMat;
rotateMat.makeRotate(osg::PI_2, osg::Y_AXIS);

osg::MatrixTransform* transMT = new osg::MatrixTransform();
transMT->setMatrix(scaleMat * rotateMat);
transMT->addChild(pActor);

osg::Matrix transMat1;
transMat1.makeTranslate(2000, 0, 100);
osg::MatrixTransform* transMT1 = new osg::MatrixTransform();
transMT1->setMatrix(transMat1);
transMT1->addChild(pActor1);
```

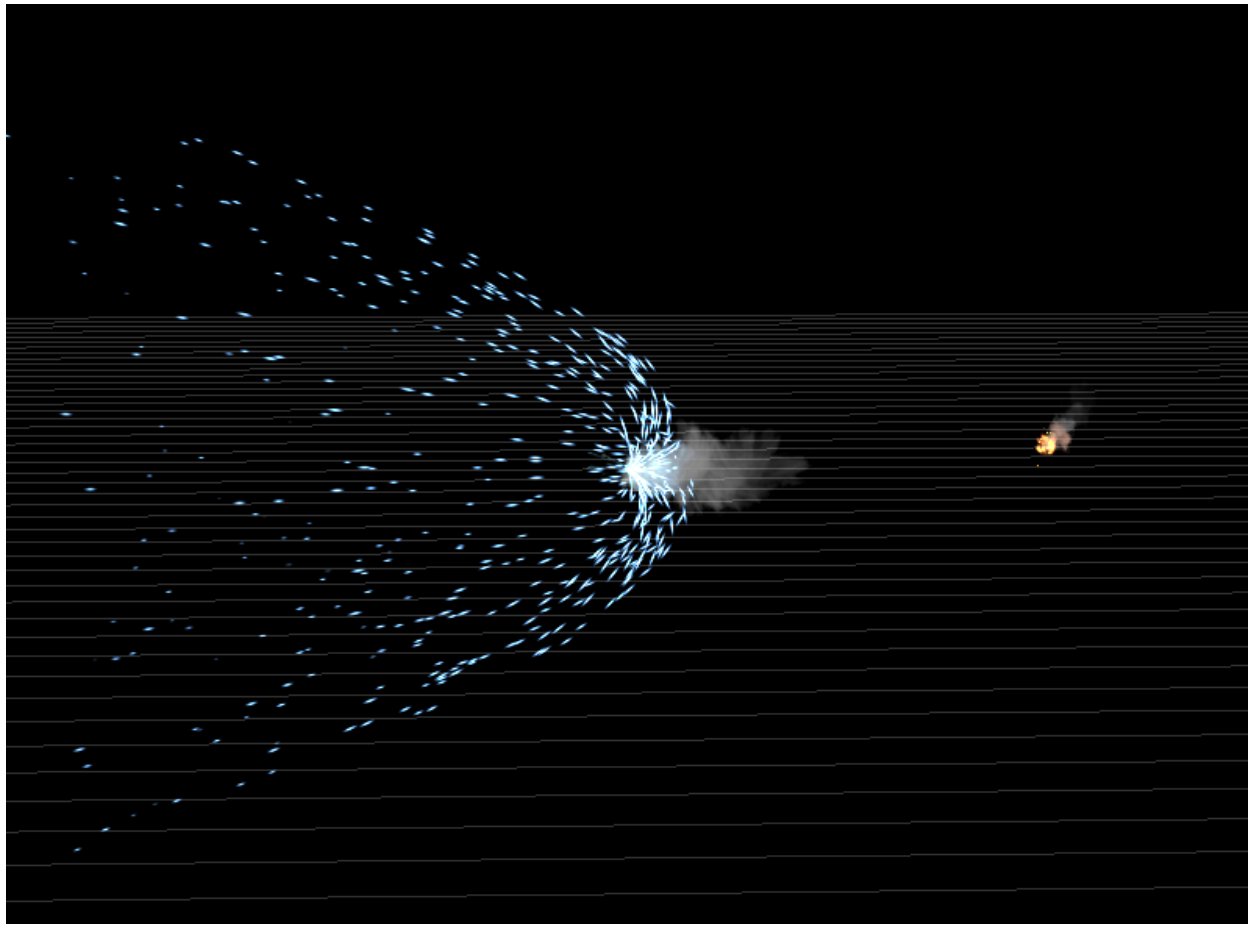
# 代码示例

挂接到根结点 →

```
pRoot->addChild(transMT);
pRoot->addChild(transMT1);
```

# 代码示例

运行效果 →



# 遗留问题

- 内存占用较高(一个高质量爆炸特效的内存占用在40MB左右)

- UE4顶点着色器中基于世界坐标系下的部分运算，在地球场景中会导致精度丢失

# 感谢观看！

销售及技术咨询-利智明