实 验 报 告 三

（2017-2018 学年第二学期）

3D 动画游戏设计算法

实验题目：Ogre Particle System

目录：

1. 分析 Ogre Particle System： Ogre::ParticleSystemManager， Ogre::ParticleSystem， Ogre::ParticleEmitter， Ogre::ParticleAffector，Ogre::Particle，Ogre::ParticleSystemRenderer

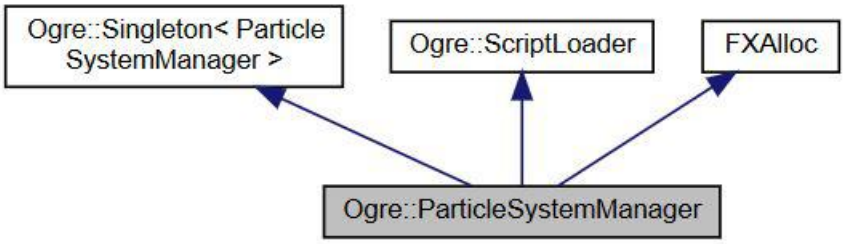
实验报告：

1. Introduction：

OGRE 版本：1.10

前言：粒子系统(Particle System)，是一种特殊的可渲染对象，可以用于模拟各种特效，例如火焰、烟花、烟雾、溅血等等。很明显可以看出这些模拟的特效都可以由一系列相同的小元素组成，这些小元素就是这个语境下的**粒子**(Particle)。粒子系统里面的粒子都有自己的运动模式和生存周期，这些都是实时计算更新的。接下来分析一下粒子系统的实现。

2. Ogre::ParticleSystemManager



/** Manages particle systems, particle system scripts (templates) and the available emitter & affector factories.

This singleton class is responsible for creating and managing particle systems. All particle systems must be created and destroyed using this object, although the user interface to creating them is via SceneManager. Remember that like all other MovableObject subclasses, ParticleSystems do not get rendered until they are attached to a SceneNode object.

This class also manages factories for ParticleEmitter and ParticleAffector classes. To enable easy extensions to the types of emitters (particle sources) and affectors (particle modifiers), the ParticleSystemManager lets plugins or applications register factory classes which submit new subclasses to ParticleEmitter and ParticleAffector. Ogre comes with a number of them already provided, such as cone, sphere and box-shaped emitters, and simple affectors such as constant directional force and colour faders. However using this registration process, a plugin can create any behaviour required.

This class also manages the loading and parsing of particle system scripts, which are text files describing

```
named particle system templates. Instances of particle systems using these templates canthen be created easily through the createParticleSystem method. */
```

粒子系统管理器管理了多种东西：

1. 粒子系统
2. 粒子系统脚本
3. 发射器 Emitter/影响器 Affector 的工厂类

ParticleSystemManager 继承了 Singleton<ParticleSystemManager>，所以它是一个用了单例模式的类，所有的粒子系统都要通过这个类来创建和销毁（跟 Noise3D 的资源控制类似）。但是从 Ogre 引擎使用者的角度来讲，这些粒子系统生存周期的管理要通过 SceneManager 提供的相应接口来操作。要注意一点，跟其他 Renderable 一样，ParticleSystem 也是要绑定到 SceneNode 之后才可以进行渲染的。

粒子系统管理器还管理 ParticleEmitter 和 ParticleAffector 的工厂类。通过**工厂的抽象类 ParticleEmitterFactory，ParticleAffectorFactory**（但感觉并不是 Abstract Factory 模式，因为抽象工厂模式是要同一个抽象工厂类通过继承和重写来生成多个具体工厂，然后用具体工厂又可以生成对应的产品），Ogre 给引擎使用者提供了一定的自由度，让他们以类似**插件**的形式修改 Emitter/Affector 的创建。

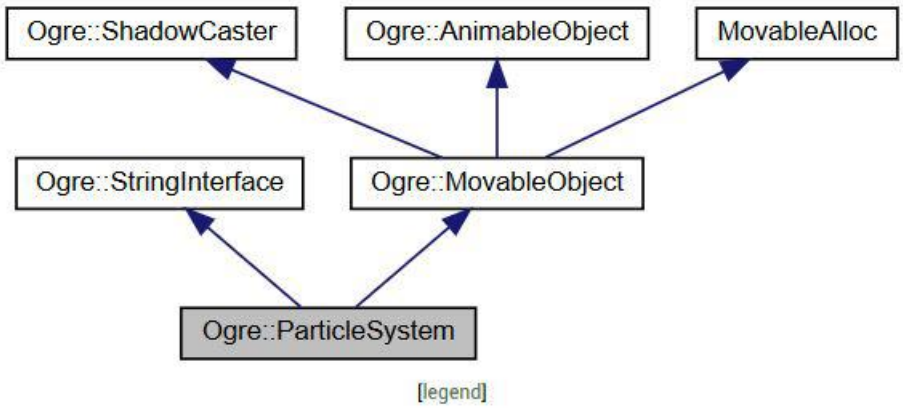
```
class _OgreExport ParticleEmitterFactory : public FXAlloc
{
protected:
    vector<ParticleEmitter*>::type mEmitters;
public:
    ParticleEmitterFactory() {}
    virtual ~ParticleEmitterFactory();
    virtual String getName() const = 0;
    virtual ParticleEmitter* createEmitter(ParticleSystem* psys) = 0;
    virtual void destroyEmitter(ParticleEmitter* e);
 };

class _OgreExport ParticleAffectorFactory : public FXAlloc
{
protected:
    vector<ParticleAffector*>::type mAffectors;
public:
    ParticleAffectorFactory() {}
    virtual ~ParticleAffectorFactory();
    virtual String getName() const = 0;
    virtual ParticleAffector* createAffector(ParticleSystem* psys) = 0;
    virtual void destroyAffector(ParticleAffector* e);
};
 }
```

粒子系统管理器还管理脚本(Script)，因为在 Ogre 里面粒子系统不支持 hardcode，只能通过脚本/字符串命令的方式灵活地配置/创建粒子系统。所以 Ogre::ParticleSystemManager 还做了粒子系统脚本的 parsing，parse 之后执行相应的操作。

```
protected:
  void parseNewEmitter(const String& type, DataStreamPtr& chunk, ParticleSystem* sys);
  void parseNewAffector(const String& type, DataStreamPtr& chunk, ParticleSystem* sys);
```

```
void parseAttrib(const String& line, ParticleSystem* sys);
void parseEmitterAttrib(const String& line, ParticleEmitter* sys);
void parseAffectorAttrib(const String& line, ParticleAffector* sys);
void skipToNextCloseBrace(DataStreamPtr& chunk);
 void skipToNextOpenBrace(DataStreamPtr& chunk);
```

3. Ogre::ParticleSystem




```
/*Particle systems are special effects generators which are based on a number of moving points to create the impression of things like like sparkles, smoke, blood spurts, dust etc. This class simply manages a single collection of particles in world space with a shared local origin for emission. The visual aspect of the  particles is handled by a ParticleSystemRenderer instance.*/
```

顾名思义，粒子系统就是管理一堆粒子及其运动的一个系统。前文也提到过了，一个粒子系统可以用一系列的粒子（点、公告板、sprite 等）来模拟一些类似烟雾、火焰的特效。这个类主要是管理一组在世界空间下的粒子（Particle），这些粒子都有同一个局部原点（一个粒子系统就有一个局部坐标系原点）。所以粒子系统可以看作是能绑定到 SceneNode 的一个实体，而且也必须要把粒子系统绑定到 SceneNode 以后才能进行粒子的渲染。

在一个粒子系统里面有几个重要的组成元素：

1. 粒子 Particles
2. 发射器 Emitter
3. 影响器 Affector

Ogre::ParticleSystem 类的有一些重要的 public 接口是用来管理上面这三种重要组成元素的生命周期。可以看到基本上都是 Particle、Emitter、Affector 的创建、Get/Set、销毁之类的操作。例如粒子发射器 Ogre::ParticleEmitter 的生存期管理接口如下：

```
ParticleEmitter* addEmitter(const String& emitterType);
ParticleEmitter* getEmitter(unsigned short index) const;
unsigned short getNumEmitters(void) const;
void removeEmitter(unsigned short index);
void removeAllEmitters(void);
void removeEmitter(ParticleEmitter *emitter);
```

那么同样的，Ogre::ParticleAffector 和 Ogre::Particle 也有一些类似的接口。

一个粒子系统还有一些其他的属性，例如：每个粒子的材质，粒子系统的 Bounding Box/Bounding Sphere，粒子移动的速度缩放因子，粒子的大小等。

```
        AxisAlignedBox mAABB;
        Real mBoundingRadius;
        bool mBoundsAutoUpdate;
        Real mBoundsUpdateTime;

        Real mUpdateRemainTime;
        MaterialPtr mMaterial;
        Real mDefaultWidth;
        Real mDefaultHeight;
        Real mSpeedFactor;
        .......
```

前文提到 Ogre 的粒子系统不支持用户 hard-code 的，引擎使用者需要自定义一个粒子系统描述脚本文件来创建并描述粒子系统。Ogre::ParticleSystemManager 里面负责 parseScript，而很多具体的脚本执行操作就在 Ogre::ParticleSystem 里面实现。

```
        /** Command object for particle_height (see ParamCommand).*/
        class _OgrePrivate CmdHeight : public ParamCommand
        {
        public:
            String doGet(const void* target) const;
            void doSet(void* target, const String& val);
         };
        ............
```

Ogre::ParticleSystem 里面定义了很多内部类，用于具体执行某一个脚本命令。例如 Ogre::ParticleSystem::CmdWidth::doSet()里实现就是：

```
static_cast<ParticleSystem*>(target)->setDefaultHeight( StringConverter::parseReal(val));
```

这个就是把具体的某一条脚本的参数 parse 出来，并且设置相应的参数。

具体的粒子更新逻辑在 void ParticleSystem::_update(Real timeElapsed) 里面实现，Ogre 会在每一帧渲染的时候调用这个函数，根据距上一帧开始流逝的时间来更新粒子系统（因为粒子需要移动，所以肯定需要时间间隔参数 dt）。我们来分析一下粒子更新逻辑的大意（调用链较长，详见代码）：

1. 更新 nonvisibleTimeout。这个意思是如果某个粒子不可见，还需不需要更新粒子，这算是在做一定程度的优化。如果某个粒子不可见的持续时间达到设定的阈值 nonvisibleTimeout，那就不对它进行更新了。
2. 根据 mSpeedFactor 缩放时间间隔 timeElapsed
3. 配置粒子系统的渲染器（如果已经配置过就跳过）：configureRenderer()
4. 初始化已发射的发射器，创建一个 emitted emitter 的 pool（如果已经初始化过就跳过）：initialiseEmittedEmitters();
5. 计算每次更新迭代的时间间隔 iterationInterval，单位与 timeElapsed 一致，一般为毫秒。这里要实现的逻辑是：每隔 iterationInterval 就更新一次粒子，所以如果相邻两次_update()的时间比较长，那么当前帧的_update()需要更新多几次。mUpdateRemainTime 如果达到 iterationInterval 的阈值，就意味着要更新粒子了。

```
        if (iterationInterval > 0)
        {
            mUpdateRemainTime += timeElapsed;
            while (mUpdateRemainTime >= iterationInterval)//如果总流逝时间大于更新间隔，那么就更新吧
            {
            ....
```

```
                mUpdateRemainTime -= iterationInterval;
            }
        }
```

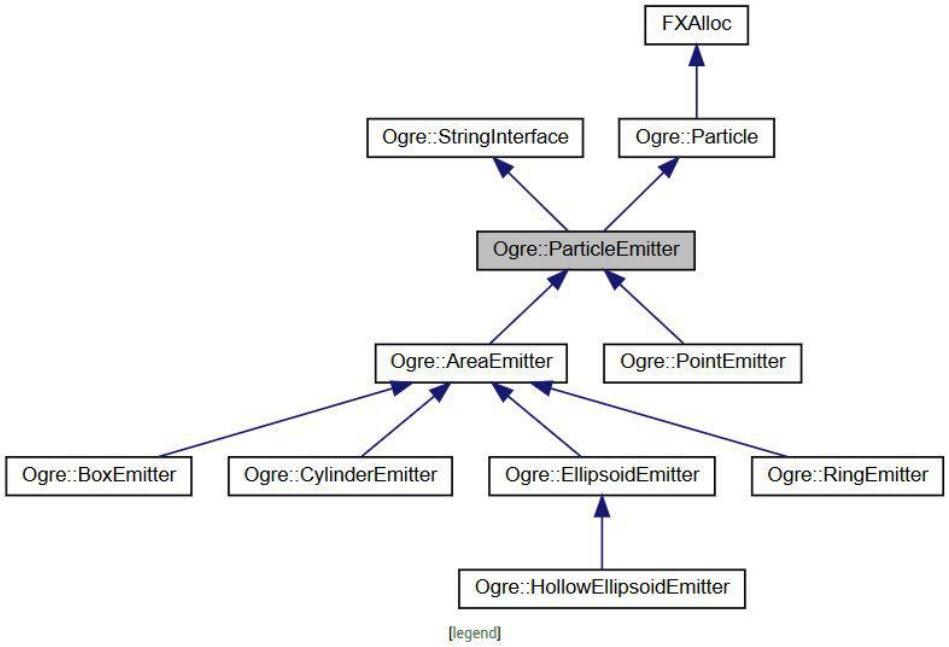
6. 在每次粒子的更新里，就要干几件事：

```
            // Update existing particles
            _expire(iterationInterval);
            _triggerAffectors(iterationInterval);
            _applyMotion(iterationInterval);
            if(mIsEmitting)
            {
                // Emit new particles
                _triggerEmitters(iterationInterval);
            }
```

- 处理过期的粒子
- 让 Affector 施加对粒子的影响
- 让 Particles 动起来,s+=v * dt。如果当前粒子的类型是 Emitter，那还要把 Particle* cast 成 ParticleEmitter*，然后设置 emitter 的 position。
- 发射粒子_triggerEmitters()。具体在 ParticleSystem::_executeTriggerEmitters(ParticleEmitter* emitter, unsigned requested, Real timeElapsed)里面发射/初始化单个粒子，新发射的粒子的参数初始化可以重写虚函数 virtual void ParticleEmitter::_initParticle(Particle* pParticle)

以上就是 Ogre::ParticleSystem 对粒子、粒子发射器、粒子影响器的管理和更新的逻辑。这些逻辑的调用链都很长，更多的细节留在其他类进行实现。

4. Ogre::ParticleEmitter



/** Abstract class defining the interface to be implemented by particle emitters.
Particle emitters are the sources of particles in a particle system. This class defines the ParticleEmitter interface, and provides a basic implementation for tasks which most emitters will do (these are of course overridable). Particle emitters can be grouped into types, e.g. 'point' emitters, 'box' emitters etc; each type will create particles with a different starting point, direction and velocity (although within the types you can

```
configure the ranges of these parameters). */
```

Ogre::ParticleEmitter 定义了粒子发射器的接口，其具体的实现可以有很多种，而且是以插件 plugin 的形式出现的，所以具体的实现不是在 OgreMain 项目里，是在 Plugin_ParticleFX 这个项目里面实现的。这里先分析一下这个接口类的设计，然后简单分析下这个接口类一些具体的实现。

在粒子系统里面，**发射器 Emitter 是粒子 Particle 的源头**。不同类型的粒子发射器可以创建不同起始点、不同运动方向、不同运动速度的粒子，这取决于具体的实现。因为有太多的发射器种类可以选择并实现，所以 Ogre 就不限制太多，就定义了接口类并容许用户做更多的不同的实现。为此，Ogre 还搞了个 ParticleEmitterFactory 的接口类让用户实现自定义 Emitter 的自定义工厂类，为了实现可扩展性(extensibility)真的是用心良苦啊= =。

```
    protected:

        // Command object for setting / getting parameters
        static EmitterCommands::CmdAngle msAngleCmd;
        static EmitterCommands::CmdColour msColourCmd;
        static EmitterCommands::CmdColourRangeStart msColourRangeStartCmd;
        ............
```

顺手提一下，Particle System 都是要用脚本系统来配置的，所以 ParticleEmitter 依旧是搞了很多的 CmdXXX 类来执行脚本操作，配置粒子发射器的属性。

```
/** Gets the number of particles which this emitter would like to emit based on the time elapsed.**/
        virtual unsigned short _getEmissionCount(Real timeElapsed) = 0;

 /** Initialises a particle based on the emitter's approach and parameters. **/
        virtual void _initParticle(Particle* pParticle) {
            // Initialise size in case it's been altered
            pParticle->resetDimensions();
        }
```

上面两个虚函数会在 void ParticleSystem::_update(Real timeElapsed)里面调用，用来具体地发射一个粒子。要注意的是，**粒子系统里面新生成的粒子并不是真的是创建一个新的粒子实例**，实际上，新发射出来的粒子实在重复利用之前超过粒子生存周期而“死去”的粒子实例，重新设置一下参数就可以了（这也说明了为什么有 particle pool 这个东西）。

_getEmissionCount(Real timeElapsed)这个函数要实现的是，给定流逝的时间，返回要发射的粒子数量。然后 ParticleSystem 根据这个函数的返回值，给被批准发射的所有粒子都执行一次 _initParticle(Particle* pParticle)。

ParticleEmitter 的成员变量提供了很多信息给具体的 Emitter(如 BoxEmitter, ColourFaderEmitter, CylinderEmitter 等 Ogre::ParticleEmitter 的派生类）用来重写 _initParticle(Particle* pParticle)：

- Vector3 mPosition：粒子相对于粒子系统中心的位置
- Real mEmissionRate：粒子发射的频率（粒子数/秒）
- String mType：发射器的类型，一定要被 emitter 的派生类初始化
- Vector3 mDirection：发射器的基本发射方向
- Vector3 mUp：名义上的“向上”向量
- Radian mAngle：粒子的发射角，也就是发射圆锥的母线与圆锥旋转对称轴的夹角
- Real mMinSpeed：发射粒子的最小速度（粒子速度在范围内随机生成）
- Real mMaxSpeed：发射粒子的最大速度（粒子速度在范围内随机生成）
- Real mMinTTL：粒子最小寿命（粒子存活时间在范围内随机生成）
- Real mMaxTTL：粒子最大寿命（粒子存活时间在范围内随机生成）
- ColourValue mColourRangeStart：粒子颜色的起始范围（粒子颜色在范围内随机生成）
- ColourValue mColourRangeEnd：粒子颜色的结束范围（粒子颜色在范围内随机生成）

- Real mStartTime：第一次 call ParticleSystem 之后多久才开始更新
- Real mDurationMin：发射器最少运行时间（0 是一直）
- Real mDurationMax：发射器最多运行时间（0 是一直）
- Real mDurationRemain：发射器还剩多少时间需要运行（那就是随着时间流逝而减少的）
- Real mRepeatDelayMin：Emitter 相邻两次重复运行的最小时间延迟/间隔
- Real mRepeatDelayMax;Emitter 相邻两次重复运行的时最大间延迟/间隔
- Real mRepeatDelayRemain：这个两次 repeat 之间的间隔还剩多少时间（随着时间流逝而减少的）

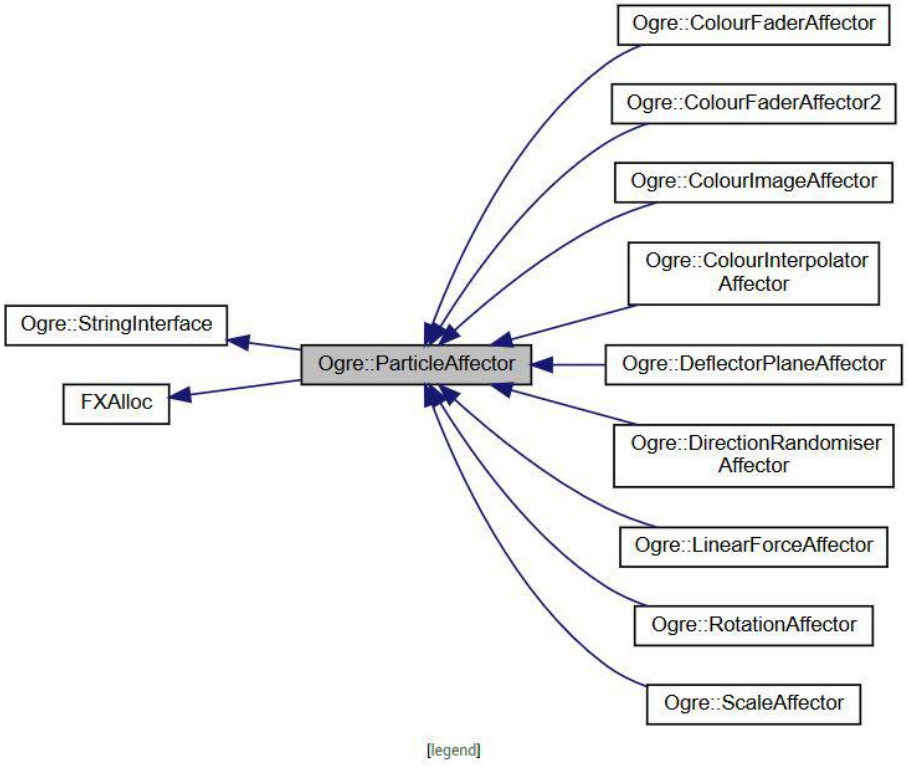
.....

拿一个具体的 Emitter 插件来举例，下面是实现是 Plugin_ParticleFX 项目的 Ogre::CylinderParticleEmitter

```
void CylinderEmitter::_initParticle(Particle* pParticle)
{
    Real x, y, z;
    AreaEmitter::_initParticle(pParticle);

    /* First we create a random point inside a bounding cylinder with aradius and height of 1 (this is easy to do). The distance of the point from 0,0,0 must be <= 1 (== 1 means on the surface and we  count this as inside, too).*/
    while (true)
    {
        // three random values for one random point in 3D space
        x = Math::SymmetricRandom();
        y = Math::SymmetricRandom();
        z = Math::SymmetricRandom();

        // the distance of x,y from 0,0 is sqrt(x*x+y*y), but as usual we can omit the sqrt(), since sqrt(1) == 1 and we
        // use the 1 as boundary. z is not taken into account, since all values in the z-direction are inside the cylinder:
        if ( x*x + y*y <= 1) { break;}// found one valid point inside
    }

    // scale the found point to the cylinder's size and move it relatively to the center of the emitter point
    pParticle->mPosition = mPosition + x * mXRange + y * mYRange + z * mZRange;

    // Generate complex data by reference
    genEmissionColour(pParticle->mColour);
    genEmissionDirection( pParticle->mPosition, pParticle->mDirection );
    genEmissionVelocity(pParticle->mDirection);

    // Generate simpler data
    pParticle->mTimeToLive = pParticle->mTotalTimeToLive = genEmissionTTL();
}
```

首先 CylinderParticleEmitter 重写了 Ogre::ParticleEmitter 的重要接口_initParticle()，在这里插件类改变了 Emitter 对 Particle 发射的初始状态。那么在 CylinderParticleEmitter 里面，粒子的初始位置就是在一个圆柱体里面，然后再根据初始位置随机生成颜色、方向、速度、生命周期（借用了基类的一些工具函数 genXXX()）。

5. Ogre::ParticleAffector



/** Abstract class defining the interface to be implemented by particle affectors.

Particle affectors modify particles in a particle system over their lifetime. They can be grouped into types, e.g. 'vector force' affectors, 'fader' affectors etc; each type will modify particles in a different way, using different parameters.

Because there are so many types of affectors you could use, OGRE chooses not to dictate the available types. It comes with some in-built, but allows plugins or applications to extend the affector types available. This is done by subclassing ParticleAffector to have the appropriate emission（哈哈哈哈） behaviour you want, and also creating a subclass of ParticleAffectorFactory which is responsible for creating instances of your new affector type. You register this factory with the ParticleSystemManager using addAffectorFactory, and from then on affectors of this type can be created either from code or through text particle scripts by naming the type.*/

Ogre::Affector，粒子影响器，可以对所有粒子的**整个生命周期都产生影响**（这一点区别于发射器，发射器只做粒子的发射/初始化）。跟 Emitter 一样，Affector 也可以分成很多类，有很多不同的具体实现的策略。

还是跟 Emitter 一样，影响器可以有很多种对粒子的影响方式，所以 Ogre 也不限制用户实现，而是以插件的形式来让你实现自定义的 Affector。可以看到 ParticleAffector 的文档里面有一处笔误“Emission”(2018.5.6, ver 1.10.11)，很明显这段文档就是直接复制 Emitter 的介绍文档而且还改漏了。那么可见 Emitter 和 Affector 在设计上其实是很相似的东西了。

```
class _OgreExport ParticleAffector : public StringInterface, public FXAlloc
{
protected:
    String mType;
    void addBaseParameters(void) { /* actually do nothing - for future possible use */ }
    ParticleSystem* mParent;
public:
    ParticleAffector(ParticleSystem* parent): mParent(parent) {}
    virtual ~ParticleAffector();

    virtual void _initParticle(Particle* pParticle){(void)pParticle;}
```

```
        /** Method called to allow the affector to 'do it's stuff' on all active particles in the system.
            This is where the affector gets the chance to apply it's effects to the particles of a system.The affector is expected to apply
    it's effect to some or all of the particles in the system passed to it, depending on the affector's approach. */
        virtual void _affectParticles(ParticleSystem* pSystem, Real timeElapsed) = 0;


        const String &getType(void) const { return mType; }
    };
```

上面是 Ogre::ParticleAffector 的接口定义。

ParticleAffector 对粒子的影响是在 void ParticleSystem::_update(Real timeElapsed)里面施加的（与 Emitter 一样）。虽然 ParitcleAffector 的类定义了一个 void _initParticle()，但据我观察，Ogre 自带的原生插件类里面的 Affector 并没有重写 void _initParticle(),这一点有点令人疑惑,因为讲道理来说初始化粒子应该是 Emitter 的工作，声明这个 virtual void _initParticle()虚函数不知道有什么用处。但是 Affector 插件都要重写 _affectParticles()，这个是毋庸置疑的。在运行时，Affector 所属的 ParticleSystem 将在每一帧都_update()，并调用_triggerAffector()来使用影响器施加对粒子的影响。

Ogre::ParticleAffector 的成员变量都是些比较 trivial 的东西，而且也没有 Emitter 那么多，从这个角度来看 Affector 的构成是更加的简单。拿一个具体的 Affector 来举例：

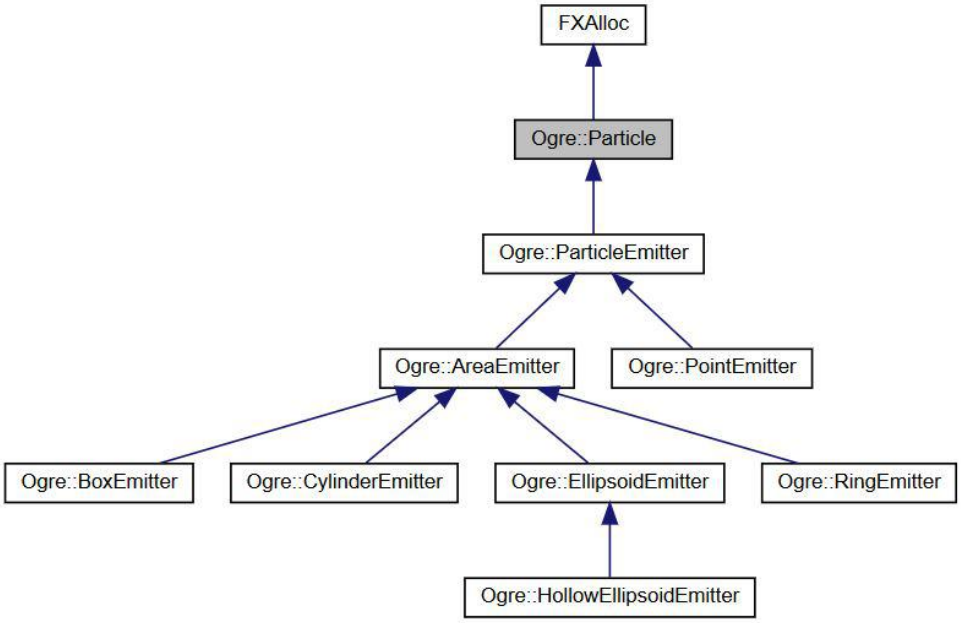
```
    /** This class defines a ParticleAffector which applies a linear force to particles in a system.*/
    class _OgreParticleFXExport LinearForceAffector : public ParticleAffector
    {
    public:
      /// Default constructor
        LinearForceAffector(ParticleSystem* psys);

        /** See ParticleAffector. */
            void _affectParticles(ParticleSystem* pSystem, Real timeElapsed);

        /** Sets the force vector to apply to the particles in a system. */
        void setForceVector(const Vector3& force);

        /** Gets the force vector to apply to the particles in a system. */
            Vector3 getForceVector(void) const;
    ..................
    }
```

这是一个线性力影响器，一个很经典的应用就是模拟重力。在 LinearForceAffector 的实现里面，_affectParticle()就被重写了，并按照相应的设置 particle 的 direction：

```
     scaledVector = mForceVector * timeElapsed;
     p->mDirection += scaledVector;
```

上面大概就是粒子系统 Affector 的设计。

6. Ogre::Particle




```
/**This Class representing a single particle instance. */
```

Ogre::Particle 就是粒子系统里面最基本的元素了。前面的 Emitter/Affector 影响的对象就是这个类的实例。

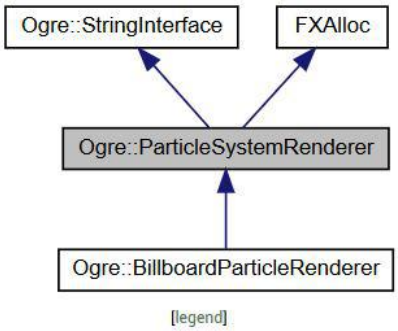
粒子有两种类型，一种是用来显示的，一种是粒子本身就是个 emitter(所以 emitter 也可以发射出 emitter，从而递归地、树状地产生新粒子）：

```
enum ParticleType
{
    Visual,
    Emitter
};
```

每个粒子都有很多自己的属性，这些属性看起来都很熟悉，因为之前分析的 emitter/affector 都可以改变这些 particle 属性：

- bool mOwnDimensions：粒子是否有自己的维度
- Real mWidth：长度
- Real mHeight：高度
- Radian mRotation：旋转角度（类似 billboard？）
- Vector3 mPosition：**世界空间**的位置
- Vector3 mDirection：移动方向(或者说是速度)
- ColourValue mColour：颜色
- Real mTimeToLive：剩余的生存时间
- Real mTotalTimeToLive;总的生存时间
- Radian mRotationSpeed：旋转的角速度
- ParticleType mParticleType：粒子类型(visual/emitter)

7. Ogre::ParticleSystemRenderer




```
/** Abstract class defining the interface required to be implemented by classes which provide rendering capability to ParticleSystem instances. */
```

粒子系统只做了粒子的运动更新的逻辑，但是其渲染还需要有一定的载体。在 Ogre 里面，原生的粒子系统渲染载体就是 Billboard，在这里就不展开讲了。Ogre::ParticleSystemRender 是个抽象类，大部分的接口定义都是纯虚函数，这就需要派生类具体的实现渲染队列的生成了。

8. 结语

粒子系统的大致设计就是这样子了，Ogre 的粒子系统为了可扩展性，为了能让用户自己实现 Emitter/Affector 插件，在设计上可谓是用心良苦。理论上粒子系统更新粒子的逻辑不算复杂，但是 Ogre 的设计模式和对软件架构的思考占据了很大一部分，导致代码看起来似乎很复杂。但是虚函数多了，要迅速地看懂设计就不容易了，Ogre 可能有点过度设计的嫌疑。

End.