# 20.6 基于GPU的粒子系统

粒子系统通常要随着时间的推移来发射和销毁粒子。从表面上看，这一工作应该用动态顶点缓冲区来实现，并在CPU上生成和销毁粒子。然后，用当前存活的粒子来填充顶点缓冲区，并对其进行绘制。不过，我们在上一节讲过，一个独立的仅有流输出的pass完全可以在GPU上控制一生成、销毁更新循环。这样做是为了提高效率——无论何时CPU向GPU更新数据都要占用时间和资源；而且，把原来属于CPU的工作转移到GPU上，可以腾出时间让CPU去完成人工智能或物理方面的工作。在本节中，我们讲解了粒子系统框架的实现细节。

## 20.6.1 粒子效果

特定粒子系统的行为方式由特定的effect文件来实现。也就是，我们为每个粒子系统（例如，雨景、火焰、烟雾等）定义一个不同的（但又相似的）effect文件。由于粒子的发射、销毁和渲染方式会随着特定的粒子系统而变化，所以这些细节被编写在了相应的effect文件中。例如：

1．当一个雨点粒子触及地面时，我们会销毁该粒子；而火焰粒子会通过倒计时的方式来进行销毁。

2．烟雾粒子的颜色会随时间而减淡，而雨点粒子的颜色不会改变。同样，烟雾粒子的尺寸会随时间而增大，而雨点粒子的尺寸不会改变。

3．直线图元通常适合于模拟雨景，而广告牌四边形适合于模拟火焰和烟雾。通过为不同的粒子系统定义不同的effect文件，我们可以在几何着色器中生成最为适宜的图元。当模拟雨景时，几何着色器将点扩展为直线；当模拟火焰和烟雾时，几何着色器将点扩展为四边形。

4．雨点粒子和烟雾粒子的初始位置和初始速度明显不同。

我们再次强调，这些粒子系统的特定细节会实现在特定的effect文件中，我们用着色器代码来处理粒子的创建、更新和销毁。这是一种非常实用的设计方式，因为我们只需要在一个新的effect文件中描述粒子系统的行为，就可以添加一个新的粒子系统。

## 20.6.2 粒子系统类

下面的类用于在C++代码中创建、更新和绘制粒子系统。这些代码构成了一个通用框架，它将用于我们创建的所有粒子系统。

class ParticleSystem

{

public:

 ParticleSystem();

 ~ParticleSystem();

 // 自重置粒子系统后经过的时间.

 float GetAge()const;

 void SetEyePos(const XMFLOAT3& eyePosW);

 void SetEmitPos(const XMFLOAT3& emitPosW);

 void SetEmitDir(const XMFLOAT3& emitDirW);

 void Init(ID3D11Device\* device, ParticleEffect\* fx,

 ID3D11ShaderResourceView\* texArraySRV,

 ID3D11ShaderResourceView\* randomTexSRV,

 UINT maxParticles);

 void Reset();

 void Update(float dt, float gameTime);

 void Draw(ID3D11DeviceContext\* dc, const Camera& cam);

private:

 void BuildVB(ID3D11Device\* device);

 ParticleSystem(const ParticleSystem& rhs);

 ParticleSystem& operator=(const ParticleSystem& rhs);

private:

 UINT mMaxParticles;

 bool mFirstRun;

 float mGameTime;

 float mTimeStep;

 float mAge;

 XMFLOAT3 mEyePosW;

 XMFLOAT3 mEmitPosW;

 XMFLOAT3 mEmitDirW;

 ParticleEffect\* mFX;

 ID3D11Buffer\* mInitVB;

 ID3D11Buffer\* mDrawVB;

 ID3D11Buffer\* mStreamOutVB;

 ID3D11ShaderResourceView\* mTexArraySRV;

 ID3D11ShaderResourceView\* mRandomTexSRV;

};

除了draw方法外，粒子系统类的所有方法实现起来都很简单（例如、创建顶点缓冲区等）；所以，我们不再赘述（请直接参见本章源代码）。

**注意**：我们在对粒子进行纹理映射时使用了一个纹理数组。这样做的原因是我们不希望所有的粒子看上去完全相同。例如，当实现一个烟雾效果时，我们希望使用多个烟雾纹理添加一些视觉上的变化；我们可以在像素着色器中使用图元ID来交替地使用纹理数组中的烟雾纹理。

## 20.6.3 发射器粒子

因为几何着色器负责创建和销毁粒子，所以我们必须定义一个或多个特殊的发射器粒子（emitter particle）。发射器粒子可以被渲染出来，也可以不被渲染出来。例如，当你希望隐藏发射器粒子时，只需要让几何着色器屏蔽该粒子的输出即可。发射器粒子的行为与其他粒子不同，因为它可以生成其他粒子。例如，发射器粒子可以跟踪已经逝去的时间，当到达某个时间点时，它发射一个新粒子。通过这一方式，新粒子就可以随着时间的推移而不断生成。我们使用**Vertex:: Particle**结构体的**Type**成员指定发射器粒子。另外，通过调整发射器粒子可以控制其他粒子的发射方式。例如，我们可以很容易地通过一个发射器粒子来控制每帧创建的粒子数量。几何着色器至少要输出一个发射器粒子来维持粒子系统的运行，当一个粒子系统中一个发射器粒子都没有时，该粒子系统将随之消失；不过，对于某些粒子系统来说，这也许正是想要得到的结果。

本书演示程序中的粒子系统只使用一个发射器粒子来控制粒子系统的生命周期。不过， 我们可以对粒子系统框架进行扩展，使它支持多个发射器粒子。另外，其他的粒子也可以发射粒子。例如，SDK中的ParticlesGS示例有一个发射器粒子（用于生成弹壳粒子的隐形粒子）和一些炮弹粒子（未爆炸的烟花，它会在几秒之后爆炸为多个新的烟花粒子，其中的一些还会再次爆炸为多个新的烟花粒子，形成二次爆炸的效果）。也就是说，发射器粒子可以生成其他的发射器粒子。练习1要求读者研究这一示例。

## 20.6.4 初始化顶点缓冲区

在我们的粒子系统中有一个特殊的初始化顶点缓冲区，它只包含一个单个发射器粒子。我们通过绘制该顶点缓冲区来启动粒子系统。该发射器粒子会在随后的帧中生成其他粒子。注意，初始化顶点缓冲区只被绘制一次（当重置粒子系统时除外）。在完成粒子系统的初始化之后，我们将按照互换方式使用两个流输出顶点缓冲区，维持粒子系统的运行。

当我们需要重新启动粒子系统时，也可使用初始化顶点缓冲区。我们使用如下代码重新启动粒子系统：

void ParticleSystem::Reset()

{

 mFirstRun = true;

 mAge = 0.0f;

}

将**mFirstRun**设为true，指示粒子系统在下次调用draw方法时绘制初始化顶点缓冲区，从而使用一个单个发射器粒子来重新启动粒子系统。

## 20.6.5 Update/Draw 方法

前面讲过，我们需要在GPU上使用两个technique来渲染粒子系统：

1．一个technique用于更新粒子系统。

2．一个technique 用于绘制粒子系统。

下面的代码用于完成一工作，并实现了两个顶点缓冲区的互换机制：

void ParticleSystem::Draw(ID3D11DeviceContext\* dc, const Camera& cam)

{

 XMMATRIX VP = cam.ViewProj();

 //

 // 设置常量缓冲.

 //

 mFX->SetViewProj(VP);

 mFX->SetGameTime(mGameTime);

 mFX->SetTimeStep(mTimeStep);

 mFX->SetEyePosW(mEyePosW);

 mFX->SetEmitPosW(mEmitPosW);

 mFX->SetEmitDirW(mEmitDirW);

 mFX->SetTexArray(mTexArraySRV);

 mFX->SetRandomTex(mRandomTexSRV);

 //

 // 设置IA阶段.

 //

 dc->IASetInputLayout(InputLayouts::Particle);

 dc->IASetPrimitiveTopology(D3D11\_PRIMITIVE\_TOPOLOGY\_POINTLIST);

 UINT stride = sizeof(Vertex::Particle);

 UINT offset = 0;

 // 如果是第一个pass，则使用初始化的VB.

 // 否则，使用包含当前粒子列表的VB.

 if( mFirstRun )

 dc->IASetVertexBuffers(0, 1, &mInitVB, &stride, &offset);

 else

 dc->IASetVertexBuffers(0, 1, &mDrawVB, &stride, &offset);

 //

 // 只使用流输出更新粒子列表，更新后的顶点通过流输出到目标VB.

 //

 dc->SOSetTargets(1, &mStreamOutVB, &offset);

 D3DX11\_TECHNIQUE\_DESC techDesc;

 mFX->StreamOutTech->GetDesc( &techDesc );

 for(UINT p = 0; p < techDesc.Passes; ++p)

 {

 mFX->StreamOutTech->GetPassByIndex( p )->Apply(0, dc);

 if( mFirstRun )

 {

 dc->Draw(1, 0);

 mFirstRun = false;

 }

 else

 {

 dc->DrawAuto();

 }

 }

 // 流输出完毕后，解除与SO阶段的绑定

 ID3D11Buffer\* bufferArray[1] = {0};

 dc->SOSetTargets(1, bufferArray, &offset);

 // 切换顶点缓冲

 std::swap(mDrawVB, mStreamOutVB);

 //

 // 绘制流输出的更新过的粒子系统.

 //

 dc->IASetVertexBuffers(0, 1, &mDrawVB, &stride, &offset);

 mFX->DrawTech->GetDesc( &techDesc );

 for(UINT p = 0; p < techDesc.Passes; ++p)

 {

 mFX->DrawTech->GetPassByIndex( p )->Apply(0, dc);

 dc->DrawAuto();

 }

}