# 20.7 火焰

下面的effect文件（fire.fx）用于渲染火焰粒子系统。它由两个technique 组成：

**1．**一个technique用于更新粒子系统。

**2．**一个technique用于绘制粒子系统。

这两个technique中的代码一般会随着特定的粒子系统而变化，实现不同的销毁、生成和渲染规则。在本例中，火焰粒子在同一个位置上发射，但是会被赋予不同的初始速度，以使火焰逐渐散开，形成一个火球。

****

**图20.5由粒子系统演示程序生成的火焰。**

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// GLOBALS \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

cbuffer cbPerFrame

{

float3 gEyePosW;

// for when the emit position/direction is varying

float3 gEmitPosW;

float3 gEmitDirW;

float gGameTime;

float gTimeStep;

float4x4 gViewProj;

};

cbuffer cbFixed

{

// 净常量加速度用于使粒子加速.

float3 gAccelW = {0.0f, 7.8f, 0.0f};

// 当将一个点粒子扩展为一个四边形时，

// 需要用到下面的纹理坐标将纹理平铺到四边形上.

float2 gQuadTexC[4] =

{

float2(0.0f, 1.0f),

float2(1.0f, 1.0f),

float2(0.0f, 0.0f),

float2(1.0f, 0.0f)

};

};

// 纹理数组.

Texture2DArray gTexArray;

// 用于生成随机数的随机纹理.

Texture1D gRandomTex;

SamplerState samLinear

{

Filter = MIN\_MAG\_MIP\_LINEAR;

AddressU = WRAP;

AddressV = WRAP;

};

DepthStencilState DisableDepth

{

DepthEnable = FALSE;

DepthWriteMask = ZERO;

};

DepthStencilState NoDepthWrites

{

DepthEnable = TRUE;

DepthWriteMask = ZERO;

};

BlendState AdditiveBlending

{

AlphaToCoverageEnable = FALSE;

BlendEnable[0] = TRUE;

SrcBlend = SRC\_ALPHA;

DestBlend = ONE;

BlendOp = ADD;

SrcBlendAlpha = ZERO;

DestBlendAlpha = ZERO;

BlendOpAlpha = ADD;

RenderTargetWriteMask[0] = 0x0F;

};

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// 辅助函数\*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

float3 RandUnitVec3(float offset)

{

// 使用游戏时间给随机纹理的采样值添加偏移量.

float u = (gGameTime + offset);

// [-1,1]区间的坐标

float3 v = gRandomTex.SampleLevel(samLinear, u, 0).xyz;

// 投影到一个单位球上

return normalize(v);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// STREAM-OUT TECH \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#define PT\_EMITTER 0

#define PT\_FLARE 1

struct Particle

{

float3 InitialPosW : POSITION;

float3 InitialVelW : VELOCITY;

float2 SizeW : SIZE;

float Age : AGE;

uint Type : TYPE;

};

Particle StreamOutVS(Particle vin)

{

return vin;

}

// The stream-out GS is just responsible for emitting

// new particles and destroying old particles. The logic

// programed here will generally vary from particle system

// to particle system, as the destroy/spawn rules will be

// different.

[maxvertexcount(2)]

void StreamOutGS(point Particle gin[1],

inout PointStream<Particle> ptStream)

{

gin[0].Age += gTimeStep;

if( gin[0].Type == PT\_EMITTER )

{

// time to emit a new particle?

if( gin[0].Age > 0.005f )

{

float3 vRandom = RandUnitVec3(0.0f);

vRandom.x \*= 0.5f;

vRandom.z \*= 0.5f;

Particle p;

p.InitialPosW = gEmitPosW.xyz;

p.InitialVelW = 4.0f\*vRandom;

p.SizeW = float2(3.0f, 3.0f);

p.Age = 0.0f;

p.Type = PT\_FLARE;

ptStream.Append(p);

// reset the time to emit

gin[0].Age = 0.0f;

}

// always keep emitters

ptStream.Append(gin[0]);

}

else

{

// Specify conditions to keep particle; this may vary from system to system.

if( gin[0].Age <= 1.0f )

ptStream.Append(gin[0]);

}

}

GeometryShader gsStreamOut = ConstructGSWithSO(

CompileShader( gs\_5\_0, StreamOutGS() ),

"POSITION.xyz; VELOCITY.xyz; SIZE.xy; AGE.x; TYPE.x" );

technique11 StreamOutTech

{

pass P0

{

SetVertexShader( CompileShader( vs\_5\_0, StreamOutVS() ) );

SetGeometryShader( gsStreamOut );

// disable pixel shader for stream-out only

SetPixelShader(NULL);

// we must also disable the depth buffer for stream-out only

SetDepthStencilState( DisableDepth, 0 );

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// DRAW TECH \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

struct VertexOut

{

float3 PosW : POSITION;

float2 SizeW : SIZE;

float4 Color : COLOR;

uint Type : TYPE;

};

VertexOut DrawVS(Particle vin)

{

VertexOut vout;

float t = vin.Age;

// 常量加速度的运动学方程

vout.PosW = 0.5f\*t\*t\*gAccelW + t\*vin.InitialVelW + vin.InitialPosW;

// fade color with time

float opacity = 1.0f - smoothstep(0.0f, 1.0f, t/1.0f);

vout.Color = float4(1.0f, 1.0f, 1.0f, opacity);

vout.SizeW = vin.SizeW;

vout.Type = vin.Type;

return vout;

}

struct GeoOut

{

float4 PosH : SV\_Position;

float4 Color : COLOR;

float2 Tex : TEXCOORD;

};

// DrawGS用于将点扩展为一个面向相机的四边形.

[maxvertexcount(4)]

void DrawGS(point VertexOut gin[1],

inout TriangleStream<GeoOut> triStream)

{

// do not draw emitter particles.

if( gin[0].Type != PT\_EMITTER )

{

//

// 计算世界矩阵使公告牌面向相机.

//

float3 look = normalize(gEyePosW.xyz - gin[0].PosW);

float3 right = normalize(cross(float3(0,1,0), look));

float3 up = cross(look, right);

//

// 在世界空间中计算三角形条带的顶点.

//

float halfWidth = 0.5f\*gin[0].SizeW.x;

float halfHeight = 0.5f\*gin[0].SizeW.y;

float4 v[4];

v[0] = float4(gin[0].PosW + halfWidth\*right - halfHeight\*up, 1.0f);

v[1] = float4(gin[0].PosW + halfWidth\*right + halfHeight\*up, 1.0f);

v[2] = float4(gin[0].PosW - halfWidth\*right - halfHeight\*up, 1.0f);

v[3] = float4(gin[0].PosW - halfWidth\*right + halfHeight\*up, 1.0f);

//

// 将四边形顶点转换到世界空间并输出为三角形条带.

//

GeoOut gout;

[unroll]

for(int i = 0; i < 4; ++i)

{

gout.PosH = mul(v[i], gViewProj);

gout.Tex = gQuadTexC[i];

gout.Color = gin[0].Color;

triStream.Append(gout);

}

}

}

float4 DrawPS(GeoOut pin) : SV\_TARGET

{

return gTexArray.Sample(samLinear, float3(pin.Tex, 0))\*pin.Color;

}

technique11 DrawTech

{

pass P0

{

SetVertexShader( CompileShader( vs\_5\_0, DrawVS() ) );

SetGeometryShader( CompileShader( gs\_5\_0, DrawGS() ) );

SetPixelShader( CompileShader( ps\_5\_0, DrawPS() ) );

SetBlendState(AdditiveBlending, float4(0.0f, 0.0f, 0.0f, 0.0f), 0xffffffff);

SetDepthStencilState( NoDepthWrites, 0 );

}

}