最简单的阴影技术---Planar Shadow

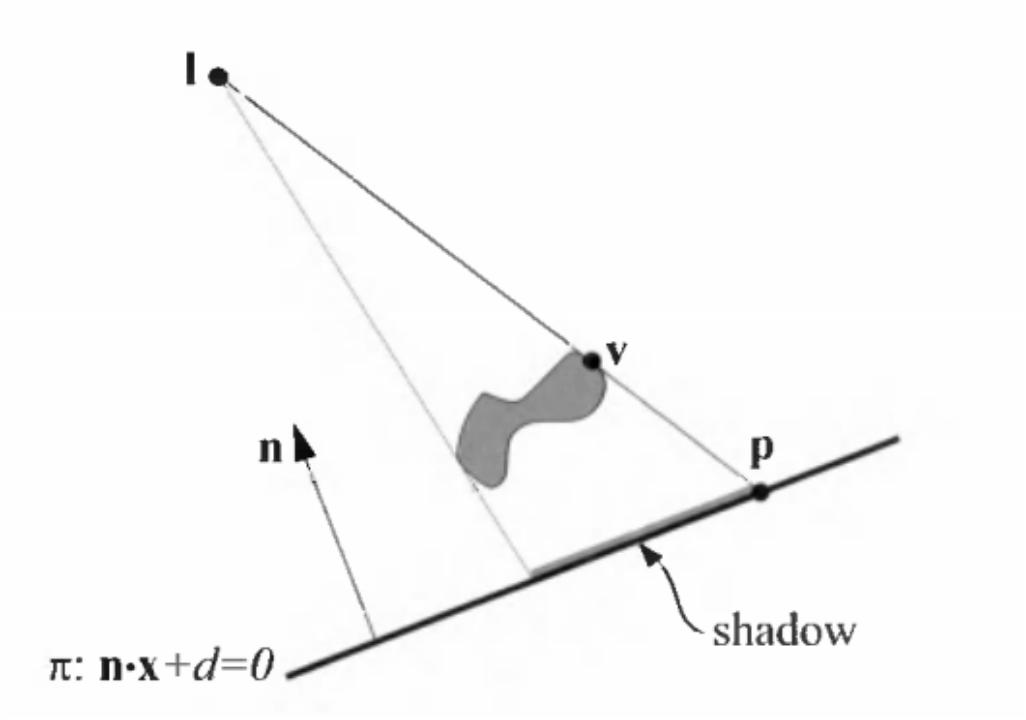
2011年12月24日
14:31

前言

这篇博客中，我将介绍一种最简单的阴影技术（Planar Shadow)和Cg Effect框架cgfx的简单使用。最近我在写一个简单的渲染引擎，设计考虑跨渲染API（opengl and direct3d)，暂时不跨平台（linux and window)，限定在Windows平台上。引擎计划加入Effect框架，就想D3D Effect那样，在OpenGL层，我暂时考虑使用cgfx，D3D的则直接使用fx框架。所以没有办法，在shader上面，暂时没有办法做到一套统一的shader。学习图形学的时间也不长，所以我是边写简单的Demo边写引擎框架，以后读研的时候，再深入研究一下。

Planar Shadow Matrix

Planar Shadow的思想很简单，根据光源和投影面位置推导一个投影矩阵，通过这个矩阵能把模型上所有顶点投射到投影面（比如地面）上，也就是将渲染物体压扁到一个平面上。投影矩阵的推到也挺容易的。



L是光源的位置，V是模型上顶点的位置，P是把模型上顶点投影到接受阴影面上的点。

投影平面（即接受阴影的平面）方程：$nx + d = 0$

推导如下

（1） 直线LV上的点P的坐标是$p = l + (v - l)t$，其中t是个参数。

（2） 其中P在接受阴影的平面上，把（1）带入平面方程，得到

$$n(l + (v - l)t) + d = 0$$

==>

$$t = -\frac{nl+d}{n(v-l)}$$

（3） 把 t 代入（1），得到

$$p = l - \frac{nl+d}{n(v-l)}(v-l)$$

根据上面推导得到的关系，按照xyz展开，就可以得到P 跟 V 之间的关系，可以用一个矩阵表示，使得 M v = P

$$\mathbf{M} = \begin{pmatrix} \mathbf{n}\cdot\mathbf{l} + d - l_x n_x & -l_x n_y & -l_x n_z & -l_x d \\ -l_y n_x & \mathbf{n}\cdot\mathbf{l} + d - l_y n_y & -l_y n_z & -l_y d \\ -l_z n_x & -l_z n_y & \mathbf{n}\cdot\mathbf{l} + d - l_z n_z & -l_z d \\ -n_x & -n_y & -n_z & \mathbf{n}\cdot\mathbf{l} \end{pmatrix}.$$

计算Shadow Matrix的函数

```
void MakePlanarShadowMatrix( float planeNormalX, float planeNormalY, float planeNormalZ,
float planeDist, float lightX, float lightY, float lightZ, Matrix4f& shadowMatrix )
{
     float nDotl = planeNormalX * lightX + planeNormalY * lightY + planeNormalZ *
     lightZ ;

     shadowMatrix.m[0][0] = nDotl + planeDist - planeNormalX * lightX;
     shadowMatrix.m[0][1] = -lightX * planeNormalY;
     shadowMatrix.m[0][2] = -lightX * planeNormalZ;
     shadowMatrix.m[0][3] = -lightX * planeDist;

     shadowMatrix.m[1][0] = -lightY * planeNormalX;
     shadowMatrix.m[1][1] = nDotl + planeDist - lightY * planeNormalY;
     shadowMatrix.m[1][2] = -lightY * planeNormalZ;
     shadowMatrix.m[1][3] = -lightY * planeDist;

     shadowMatrix.m[2][0] = -lightZ * planeNormalX;
     shadowMatrix.m[2][1] = -lightZ * planeNormalY;
     shadowMatrix.m[2][2] = nDotl + planeDist - lightZ * planeNormalZ;
     shadowMatrix.m[2][3] = -lightZ * planeDist;

     shadowMatrix.m[3][0] = -planeNormalX;
     shadowMatrix.m[3][1] = -planeNormalY;
     shadowMatrix.m[3][2] = -planeNormalZ;
     shadowMatrix.m[3][3] = nDotl;
}
```

cg shader effect Cgfx

想必用过Direct3D 10 的人都会觉得Effect的框架，对于写简单Demo很好用。Cgfx是对应于cg Shader 语言的Effect框架。在学习OpenGL的过程中，我是模仿D3D10, 完全使用可编程管线，不再使用OpenGL 固定管线。所以，需要自己计算View,Projection Matrix，每次渲染需要自己写Shader。

最简单cgfx的使用大致如下,具体见代码：

1. Create CgContext
2. Create Effect
3. Get Technique From Effect
4. Get Parameter From Effect
5. Set Parameter
6. For each pass in technique, Draw

Planar Shadow Demo 不需要复杂的Shader，绘制阴影只是简单的是世界变化矩阵里乘上一个Shadow Matrix, 把模型压扁到接受阴影的平面上。所以我使用了Per-Pixel Texture Lighting Shader，就当看完书实践一下。
下面是我使用的Effect File, 最简单的Per-Pixel Lighting，没有加入高光.

```
float4x4 WorldMatrix : World;
float4x4 ViewMatrix : View;
float4x4 ProjectionMatrix : Projection;

float3 LightPositon;
float3 LightColor;

float3 AmbientLight = float3(0.2, 0.2, 0.2);

int Shadow; // if 1, draw planar shadow

sampler2D DiffuseMap = sampler_state {
     MinFilter = Linear;
     MagFilter = Linear;
     WrapS = Wrap;
     WrapT = Wrap;
};

struct VertexShaderInput
{
     float4 Pos    : POSITION;
     float3 Normal : NORMAL;
     float2 Tex    : TEXCOORD0;
};

struct VertexShaderOutput
{
    float4 Pos : POSITION;
     float2 Tex : TEXCOORD0;
     float3 Normal : TEXCOORD1;
     float3 WorldPos : TEXCOORD2;
};


VertexShaderOutput VertexShaderFunction(VertexShaderInput input)
{
    VertexShaderOutput output;

     output.Pos = mul( WorldMatrix, input.Pos );
    output.Pos = mul( ViewMatrix, output.Pos );
    output.Pos = mul( ProjectionMatrix, output.Pos );

     output.Tex = input.Tex;

     output.WorldPos = mul(WorldMatrix, input.Pos);
     output.Normal = mul((float3x3)WorldMatrix, input.Normal);

    return output;
}

float4 PixelShaderFunction(VertexShaderOutput input) : COLOR
{
```

```
	float4 renderColor = float4(0, 0, 0, 1);

	if(!Shadow)
	{
		float3 lightVec = normalize(input.WorldPos - LightPositon);
		float diffuseFactor = saturate( dot(-lightVec, input.Normal) );
		float3 diffuseMaterial = tex2D(DiffuseMap, input.Tex);
		float3 diffuseColor = diffuseFactor * LightColor * diffuseMaterial;
		float3 ambientColor = AmbientLight * diffuseMaterial;
		renderColor = float4(diffuseColor + ambientColor, 1);
	}

	return renderColor;
}

technique TextureLightingTech
{
    pass
    {
		CullFaceEnable = true;
		CullFace = Back;
		//PolygonMode =  int2(Front, Line);
        VertexProgram   = compile gp4vp VertexShaderFunction();
        FragmentProgram = compile gp4fp PixelShaderFunction();
    }
}
```

主要注意事项和需要改进之处

1、 因为压扁后的阴影正好跟接受阴影平面的位置重合，所以在绘制接受面后，再绘制阴影的时候，就可能出现Z-Fighting现象。所以使用了PolygonOffset的方法使得Shadow的位置产生一点偏移，出现在接受平面的上方。具体设置如下，参考网上的方法

```
glEnable( GL_POLYGON_OFFSET_FILL );
glPolygonOffset( -0.1f, 0.2f );

glPolygonOffset( 0.0f, 0.0f );
glDisable( GL_POLYGON_OFFSET_FILL );
```

2、 暂时没有使用Stencil Buffer控制阴影的绘制的位置，所以不管阴影落在接受面里面还是外面，都要绘制。

3、 我发现一种情况，当遮挡物（即产生阴影的物体）有一部分位于平面下方的时候，这部分是不应该产生阴影的，但用投影的方法，在平面下方的部分还是会投影到　平面上产生阴影，按照数学公式，确实是应该这样的，但对于阴影来说，却是不正确的。