ADVANCED SHADERS

Durante los próximos días continuaremos aprendiendo lo necesario para implementar shaders avanzados que nos permitirán o bien acelerar el rendimiento de nuestro videojuego o bien mejorar la representación gráfica del mismo.

Skinning

El primer shader que explicaremos será Skinning un shader que nos permitirá acelerar el rendimiento en el renderizado de los modelos animados con cal3D.

Para realizar dicho shader nos basaremos en el proyecto de ejemplo del cal3D que se encuentra en la carpeta miniviewer_d3d_vs. Dicho proyecto se basa a su vez en el proyecto miniviewer_d3d modificándolo para poder aplicar el shader de skinning.

Recordar primero de todo como funciona la animación esqueletal, para ello podemos revisar la documentación de este tema que dimos previamente. Lo que debemos destacar del funcionamiento es:

"Animamos los huesos del esqueleto y con las matrices de transformación resultantes las multiplicamos por los vértices y los pesos dándonos la malla animada"

Por tanto mediante un vertex shader podremos realizar esta tarea, pasándole a cada uno de los vértices los índices de los huesos y los pesos aplicados a cada uno de los vértices para poder calcular estas transformaciones. Para ello empezaremos creando un nuevo tipo de vértice como el siguiente:

```
struct CAL3D_HW_VERTEX
{
	float			x, y, z;
	float			weights[4];
	float			indices[4];
	float			nx, ny, nz;
	//En caso de utilizar NormalMap
	//float			nx, ny, nz, nw;
	//float			tangentx, tangenty, tangentz, tangentw;
	//float			binormalx, binormaly, binormalz, binormalw;

	float			tu,tv;

	static inline unsigned short GetVertexType();
	static inline unsigned int GetFVF()
	{
		return 0;
	}
	static LPDIRECT3DVERTEXDECLARATION9		s_VertexDeclaration;
	static LPDIRECT3DVERTEXDECLARATION9 & GetVertexDeclaration();
	static void ReleaseVertexDeclaration()
	{
		CHECKED_RELEASE(s_VertexDeclaration);
```

```
        }
};
```

Como podemos apreciar hemos creado un tipo de vértice que incluye un array de pesos y un array de índices, con esta nueva estructura deberemos crear el vertexdeclaration correspondiente así como el vertex type.

Para llevar a cabo este shader deberemos modificar el funcionamiento de nuestras clases de mallas animadas. Para empezar el Vertex Buffer y el Index Buffer ya no van a ser independientes para cada instancia del modelo animado porque estos modelos se encontrarán en la pose inicial y el shader será el encargado de recalcular sus posiciones, por tanto el Vertex Buffer y el Index Buffer los pasaremos a la clase CAnimatedCoreModel, pero creándolo como CRendeableVertex y añadiremos también un objeto nuevo de la clase del Cal3D CalHardwareModel la cual deberemos crear y destruir. Por tanto nuestro fichero .h añadiremos los siguientes propiedades y métodos.

```
class CAnimatedCoreModel : public CNamed
{
private:
      //El código que ya teníais
      CalHardwareModel                    *m_CalHardwareModel;
      CRenderableVertexs
      *m_RenderableVertexs;
public:
      //El código que ya teníais
      CalHardwareModel * GetCalHardwareModel() const;
      bool LoadVertexBuffer(CalModel *Model);
      CRenderableVertexs * GetRenderableVertexs() const;
};
```

Lo siguiente que haremos será mover el método LoadVertexBuffer de la clase CAnimatedInstanceModel a la clase CAnimatedCoreModel y modificarlo para copiar los vértices y los índices en nuestra clase que envuelve el Cal3D, utilizando un código similar al siguiente.

```
CAL3D_HW_VERTEX* pVertex;

m_CalHardwareModel = new CalHardwareModel(m_CalCoreModel);

CAL3D_HW_VERTEX *l_Vtxs=new CAL3D_HW_VERTEX[m_NumVtxs*2]; //Cogemos el
      doble de vértices necesarios porque al crear el model de hardware
      puede necesitar más vértices que el modelo por software
unsigned short *l_Idxs=new unsigned short[m_NumFaces*3];


m_CalHardwareModel->setVertexBuffer((char*) l_Vtxs,
      sizeof(CAL3D_HW_VERTEX));
m_CalHardwareModel->setWeightBuffer(((char*)l_Vtxs) + 12,
      sizeof(CAL3D_HW_VERTEX));
m_CalHardwareModel->setMatrixIndexBuffer(((char*)l_Vtxs) + 28,
      sizeof(CAL3D_HW_VERTEX));
m_CalHardwareModel->setNormalBuffer(((char*)l_Vtxs)+44,
      sizeof(CAL3D_HW_VERTEX));
m_CalHardwareModel->setTextureCoordNum(1);
m_CalHardwareModel->setTextureCoordBuffer(0,((char*)l_Vtxs)+92,
      sizeof(CAL3D_HW_VERTEX));
```

```
m_CalHardwareModel->setIndexBuffer(l_Idxs);

m_CalHardwareModel->load( 0, 0, MAXBONES);

m_NumVtxs=m_CalHardwareModel->getTotalVertexCount();
//En caso de utilizar NormalMap
//CalcTangentsAndBinormals(l_Vtxs, l_Idxs, m_NumVtxs, m_NumFaces*3,
      sizeof(CAL3D_HW_VERTEX),0, 44, 60, 76, 92);
m_RenderableVertexs=new CIndexedVertexs<CAL3D_HW_VERTEX>(l_Vtxs, l_Idxs,
      m_NumVtxs, m_NumFaces*3);
delete []l_Vtxs;
delete []l_Idxs;
```

Lo que estamos realizando en este código es primero crear el modelo de hardware, crear un Vertex Buffer y Index Buffer en memoria RAM dónde escribiremos los vértices y los índices de nuestro modelo.

Para realizar esta tarea le deberemos:

- decir en que byte se sitúa dentro de la estructura la estructura de vértice con el método setVertexBuffer
- decir en que byte se sitúa dentro de la estructura la estructura de pesos con el método setWeightBuffer
- decir en que byte se sitúa dentro de la estructura la estructura los índices de las matrices que se aplican sobre este vértice con el método setMatrixIndexBuffer
- decir en que byte se sitúa dentro de la estructura la estructura de normales con el método setNormalBuffer
- decir el número de texturas que tendrá nuestro vertex buffer con el método setTextureCoordeNum
- decir en que byte se sitúa dentro de la estructura la estructura de coordenadas de textura con el método setTextureCoordBuffer
- decir en que dirección de memoria se han de copiar los índices con el método setIndexBuffer
- por último cargaremos toda la estructura de vértices e índices en el vertex buffer e index buffer con el método load

A continuación deberemos crear el método RenderByHardware dentro de la clase CAnimatedInstanceModel y establecer todas las variables que utilizaremos dentro de nuestro shader.

El código que utilizaremos será similar al siguiente.

```
void CAnimatedInstanceModel::Render(CRenderManager *RM)
{
      CEffectManager &l_EffectManager=CORE.GetEffectManager();
      CEffectTechnique *l_EffectTechnique =
           l_EffectManager.GetAnimatedModelTechnique();

      if(l_EffectTechnique==NULL)
            l_EffectTechnique=m_EffectTechnique;

      if(l_EffectTechnique==NULL)
            return;

      l_EffectManager.SetWorldMatrix(GetTransform());
```

```
        CEffect *m_Effect=l_EffectTechnique->GetEffect();

        if(m_Effect==NULL)
              return;
        LPD3DXEFFECT l_Effect=m_Effect->GetD3DEffect();
        if(l_Effect)
        {
              l_EffectTechnique->BeginRender();
              CalHardwareModel *l_CalHardwareModel=m_AnimatedCoreModel->
                    GetCalHardwareModel();

              D3DXMATRIX transformation[MAXBONES];
              for(int hardwareMeshId=0;hardwareMeshId<l_CalHardwareModel->
                    getHardwareMeshCount(); hardwareMeshId++)
              {
                    l_CalHardwareModel->selectHardwareMesh(hardwareMeshId);

                    for(int boneId = 0; boneId < l_CalHardwareModel->
                          getBoneCount(); boneId++)
                    {
                          D3DXMatrixRotationQuaternion(
                                &transformation[boneId],(CONST
                                D3DXQUATERNION*)&l_CalHardwareModel->
                                getRotationBoneSpace(boneId, m_CalModel->
                                getSkeleton()));
                          CalVector translationBoneSpace =
                                l_CalHardwareModel->
                                getTranslationBoneSpace(boneId,m_CalModel->
                                getSkeleton());
                          transformation[boneId]._14 =
                                translationBoneSpace.x;
                          transformation[boneId]._24 =
                                translationBoneSpace.y;
                          transformation[boneId]._34 =
                                translationBoneSpace.z;
                    }
                    float l_Matrix[MAXBONES*3*4];
                    for(int b=0;b<l_CalHardwareModel->getBoneCount();++b)
                    {
                          memcpy(&l_Matrix[b*3*4], &transformation[b],
                                sizeof(float)*3*4);
                    }
                    l_Effect->SetFloatArray(m_Effect->m_BonesParameter,
                          (float *)l_Matrix,(l_CalHardwareModel-
                          >getBoneCount())*3*4);
                    m_TextureList[0]->Activate(0);
                    m_NormalTextureList[0]->Activate(1);
                    m_AnimatedCoreModel->GetRenderableVertexs()->
                          Render(l_EffectTechnique, l_CalHardwareModel->
                          getBaseVertexIndex(), 0, l_CalHardwareModel->
                          getVertexCount(), l_CalHardwareModel->
                          getStartIndex(),l_CalHardwareModel->
                          getFaceCount());
              }
        }
}
```

Por último deberemos crear el shader que nos permitirá renderizar este modelo animado, para ello utilizaremos el código del shader siguiente extraído del proyecto miniviewer_d3d_vs y modificado para poder utilizarlo con una technique.

```
#define MAXBONES 29
```

```
struct CAL3D_HW_VERTEX_VS  {
    float3 Position          : POSITION;
    float4 Weight            : BLENDWEIGHT;
    float4 Indices           : BLENDINDICES;
    float4 Normal            : NORMAL;
//  float4 Tangent           : TANGENT0;
//  float4 BiNormal          : BINORMAL0;
    float2 TexCoord          : TEXCOORD0;
};

struct CAL3D_HW_VERTEX_PS
{
    float4 HPosition         : POSITION;
    float2 UV                : TEXCOORD0;
    float3 WorldNormal       : TEXCOORD1;
    float3 WorldPosition     : TEXCOORD2;
//  float3 WorldTangent      : TEXCOORD3;
//  float3 WorldBinormal     : TEXCOORD4;
};

sampler DiffuseTextureSampler : register( s0 ) = sampler_state
{
    MipFilter = LINEAR;
    MinFilter = LINEAR;
    MagFilter = LINEAR;
};


float4x4 g_WorldViewProjMatrix;
float4x4 g_WorldMatrix;
float3x4 g_Bones[MAXBONES];

float3 CalcAnimtedPos(float4 Position, float4 Indices, float4 Weight)
{
    float3 l_Position=0;

    l_Position  = mul(g_Bones[Indices.x], Position) * Weight.x;
    l_Position += mul(g_Bones[Indices.y], Position) * Weight.y;
    l_Position += mul(g_Bones[Indices.z], Position) * Weight.z;
    l_Position += mul(g_Bones[Indices.w], Position) * Weight.w;


    return l_Position;
}

void CalcAnimatedNormalTangent(float3 Normal, float3 Tangent, float4 Indices, float4 Weight, out
    float3 OutNormal, out float3 OutTangent)
{
    OutNormal = 0;
    OutTangent =0;

    float3x3 m;

    m[0].xyz = g_Bones[Indices.x][0].xyz;
    m[1].xyz = g_Bones[Indices.x][1].xyz;
    m[2].xyz = g_Bones[Indices.x][2].xyz;

    OutNormal += mul(m, Normal.xyz)* Weight.x;
    OutTangent += mul(m, Tangent.xyz)* Weight.x;

    m[0].xyz = g_Bones[Indices.y][0].xyz;
    m[1].xyz = g_Bones[Indices.y][1].xyz;
    m[2].xyz = g_Bones[Indices.y][2].xyz;
```

```
    OutNormal += mul(m, Normal.xyz)* Weight.y;
    OutTangent += mul(m, Tangent.xyz)* Weight.y;

    m[0].xyz = g_Bones[Indices.z][0].xyz;
    m[1].xyz = g_Bones[Indices.z][1].xyz;
    m[2].xyz = g_Bones[Indices.z][2].xyz;

    OutNormal += mul(m, Normal.xyz)* Weight.z;
    OutTangent += mul(m, Tangent.xyz)* Weight.z;

    m[0].xyz = g_Bones[Indices.w][0].xyz;
    m[1].xyz = g_Bones[Indices.w][1].xyz;
    m[2].xyz = g_Bones[Indices.w][2].xyz;

    OutNormal += mul(m, Normal.xyz)* Weight.w;
    OutTangent += mul(m, Tangent.xyz)* Weight.w;

    OutNormal = normalize(OutNormal);
    OutTangent = normalize(OutTangent);
}

CAL3D_HW_VERTEX_PS RenderCal3DHWVS(CAL3D_HW_VERTEX_VS IN)
{
    CAL3D_HW_VERTEX_PS OUT=(CAL3D_HW_VERTEX_PS)0;

    float3 l_Normal= 0;
    float3 l_Tangent=0;

    CalcAnimatedNormalTangent(IN.Normal.xyz, /*IN.Tangent.xyz*/0, IN.Indices, IN.Weight,
    l_Normal, l_Tangent);
    float3 l_Position=CalcAnimtedPos(float4(IN.Position.xyz,1.0), IN.Indices, IN.Weight);

    float4 l_WorldPosition=float4(l_Position, 1.0);

    OUT.WorldPosition=mul(l_WorldPosition,g_WorldMatrix);
    OUT.WorldNormal=normalize(mul(l_Normal,g_WorldMatrix));
    //OUT.WorldTangent=normalize(mul(l_Tangent,g_WorldMatrix));
    //OUT.WorldBinormal=mul(cross(l_Tangent,l_Normal),(float3x3)g_WorldMatrix);
    OUT.UV = IN.TexCoord.xy;


    OUT.HPosition = mul(WorldPosition, g_WorldViewProjectionMatrix );

    return OUT;
}

float4 RenderCal3DHWPS(CAL3D_HW_VERTEX_PS IN) : COLOR
{
    //float3 Nn=CalcBumpMap(IN.WorldPosition, IN.WorldNormal, IN.WorldTangent,
    IN.WorldBinormal, IN.UV);
    float3 Nn=normalize(IN.WorldNormal);

    float4 l_SpecularColor = 1.0;
    float4 l_DiffuseColor=tex2D(DiffuseTextureSampler, IN.UV);
    return CalcLighting (IN.WorldPosition, Nn, l_DiffuseColor, l_SpecularColor);
}

technique Cal3DTechnique
{
    pass p0
    {
        VertexShader = compile vs_3_0 RenderCal3DHWVS();
```

```
        PixelShader = compile ps_3_0 RenderCal3DHWPS();
    }
}
```

Una vez implementado todo este código deberíamos ser capaces de reproducir la animación directamente a través del Vertex Shader consiguiendo una mejora del frame rate más que considerable.

NormalMap

El efecto de normal map parte de la idea de la iluminación que ya hemos explicado previamente, sin embargo mediante esta técnica lo que haremos será recalcular la normal según el píxel dónde nos encontramos. Para realizar esto nos basaremos en una textura de normales que el RGB del color corresponde con las coordenadas de normales para ese punto.

Para producir este efecto vamos a necesitar pasar la información de tangenciales en el vértice, por tanto deberemos definir un vertex type como el siguiente:

```
struct TNORMAL_TANGENT_BINORMAL_TEXTURED_VERTEX
{
        float                           x, y, z;
        float                           nx, ny, nz, nw;
        float                           tangentx, tangenty, tangentz, tangentw;
        float                           binormalx, binormaly, binormalz, binormalw;
        float                           tu,tv;

        static inline unsigned short GetVertexType();
        static inline unsigned int GetFVF()
        {
                return 0;
        }
        static LPDIRECT3DVERTEXDECLARATION9             s_VertexDeclaration;
        static LPDIRECT3DVERTEXDECLARATION9 & GetVertexDeclaration();
        static void ReleaseVertexDeclaration()
        {
                CHECKED_RELEASE(s_VertexDeclaration);
        }
};
```

Las tangenciales son unos vectores que nos permitirá implementar un sistema de coordenadas aplicado a la textura. Se calculan a través de las coordenadas de textura y las normales del vértice. Para más información podéis leer el artículo de gamasutra http://www.gamasutra.com/view/feature/1515/messing_with_tangent_space.php.

Para calcular las tangenciales utilizaremos el siguiente fragmento de código.

```
void CalcTangentsAndBinormals(void *VtxsData, unsigned short *IdxsData, size_t VtxCount,
                size_t IdxCount, size_t VertexStride, size_t GeometryStride, size_t NormalStride,
                size_t TangentStride, size_t BiNormalStride, size_t TextureCoordsStride)
{
        D3DXVECTOR3 *tan1 = new D3DXVECTOR3[VtxCount * 2];
        D3DXVECTOR3 *tan2 = tan1 + VtxCount;
        ZeroMemory(tan1, VtxCount * sizeof(D3DXVECTOR3) * 2);
```

```
unsigned char *l_VtxAddress=(unsigned char *)VtxsData;

for(size_t b=0;b<IdxCount;b+=3)
{
	unsigned short i1=IdxsData[b];
	unsigned short i2=IdxsData[b+1];
	unsigned short i3=IdxsData[b+2];

	D3DXVECTOR3 *v1=(D3DXVECTOR3 *)
		&l_VtxAddress[i1*VertexStride+GeometryStride];
	D3DXVECTOR3 *v2=(D3DXVECTOR3 *)
		&l_VtxAddress[i2*VertexStride+GeometryStride];
	D3DXVECTOR3 *v3=(D3DXVECTOR3 *)
		&l_VtxAddress[i3*VertexStride+GeometryStride];

	D3DXVECTOR2 *w1=(D3DXVECTOR2 *)
		&l_VtxAddress[i1*VertexStride+TextureCoordsStride];
	D3DXVECTOR2 *w2=(D3DXVECTOR2 *)
		&l_VtxAddress[i2*VertexStride+TextureCoordsStride];
	D3DXVECTOR2 *w3=(D3DXVECTOR2 *)
		&l_VtxAddress[i3*VertexStride+TextureCoordsStride];

	float x1=v2->x-v1->x;
	float x2=v3->x-v1->x;
	float y1=v2->y-v1->y;
	float y2=v3->y-v1->y;
	float z1=v2->z-v1->z;
	float z2=v3->z-v1->z;

	float s1=w2->x-w1->x;
	float s2=w3->x-w1->x;
	float t1=w2->y-w1->y;
	float t2=w3->y-w1->y;

	float r = 1.0F / (s1 * t2 - s2 * t1);

	D3DXVECTOR3 sdir((t2 * x1 - t1 * x2) * r, (t2 * y1 - t1 * y2) * r,
		(t2 * z1 - t1 * z2) * r);
	D3DXVECTOR3 tdir((s1 * x2 - s2 * x1) * r, (s1 * y2 - s2 * y1) * r,
		(s1 * z2 - s2 * z1) * r);

	assert(i1<VtxCount);
	assert(i2<VtxCount);
	assert(i3<VtxCount);

	tan1[i1] += sdir;
	tan1[i2] += sdir;
	tan1[i3] += sdir;

	tan2[i1] += tdir;
	tan2[i2] += tdir;
	tan2[i3] += tdir;
}

for (size_t b=0;b<VtxCount;++b)
{
	D3DXVECTOR3 *l_NormalVtx=(D3DXVECTOR3 *)
		&l_VtxAddress[b*VertexStride+NormalStride];
	D3DXVECTOR3 *l_TangentVtx=(D3DXVECTOR3 *)
		&l_VtxAddress[b*VertexStride+TangentStride];
	D3DXVECTOR4 *l_TangentVtx4=(D3DXVECTOR4 *)
		&l_VtxAddress[b*VertexStride+TangentStride];
```

```
                D3DXVECTOR3 *l_BiNormalVtx=(D3DXVECTOR3 *)
                        &l_VtxAddress[b*VertexStride+BiNormalStride];
                const D3DXVECTOR3& t=tan1[b];

                // Gram-Schmidt orthogonalize
                D3DXVECTOR3 l_VAl=t-(*l_NormalVtx)*D3DXVec3Dot(l_NormalVtx, &t);
                D3DXVec3Normalize(l_TangentVtx,&l_VAl);
                //tangent[a] = (t - n * Dot(n, t)).Normalize();

                // Calculate handedness
                D3DXVECTOR3 l_Cross;
                D3DXVec3Cross(&l_Cross,l_NormalVtx,l_TangentVtx);
                l_TangentVtx4->w=(D3DXVec3Dot(&l_Cross,&tan2[b])< 0.0f ) ? -1.0f : 1.0f;
                //tangent[a].w = (Dot(Cross(n, t), tan2[a]) < 0.0F) ? -1.0F : 1.0F;

                D3DXVec3Cross(l_BiNormalVtx,l_NormalVtx,l_TangentVtx);
        }

        delete[] tan1;
}
```

Con esta función podemos pasarle una estructura de datos que contenga toda la información necesaria para calcular sus tangenciales y binormales. Para el uso de esta función deberemos pasarle:

- la dirección de memoria de los vértices
- la dirección de memoria de los índices
- el número de vértices
- el número de índices
- el stride del vértice (el sizeof del vértice),
- el número de bytes que debemos trasladarnos en el vértice para llegar a la geometría
- el número de bytes que debemos trasladarnos en el vértice para llegar a la normal
- el número de bytes que debemos trasladarnos en el vértice para llegar a la tangencial
- el número de bytes que debemos trasladarnos en el vértice para llegar a la binormal
- el número de bytes que debemos trasladarnos en el vértice para llegar a la coordenada de textura

Una vez tenemos implementado el nuevo vértice podemos modificar nuestro shader para representar el NormalMap, para ello deberemos crear un nuevo sampler que recibirá la textura de normales. Similar al siguiente código.

```
sampler NormalMapTextureSampler : register( s1 ) = sampler_state
{
   MipFilter = LINEAR;
   MinFilter = LINEAR;
   MagFilter = LINEAR;
};
```

La implementación del vertex shader y del píxel shader variará de la siguiente forma.

```
// En el vertex shader deberemos calcular la tangencial y la binormal en coordenadas de mundo
OUT.WorldTangent=mul(IN.Tangent.xyz,(float3x3)g_WorldMatrix);
OUT.WorldBinormal = mul(cross(IN.Tangent.xyz,IN.Normal),(float3x3)g_WorldMatrix);

// En el pixel shader recalcularemos la normal del pixel a través del siguiente fragmento de código
float3 Tn=normalize(IN.WorldTangent);
float3 Bn=normalize(IN.WorldBinormal);
//La variable g_Bump es una constante que nos dará la profundidad, podemos utilizar un valor de 2.4
float3 bump=g_Bump*(tex2D(NormalMapTextureSampler,IN.UV).rgb - float3(0.5,0.5,0.5));
Nn = Nn + bump.x*Tn + bump.y*Bn;
Nn = normalize(Nn);
```

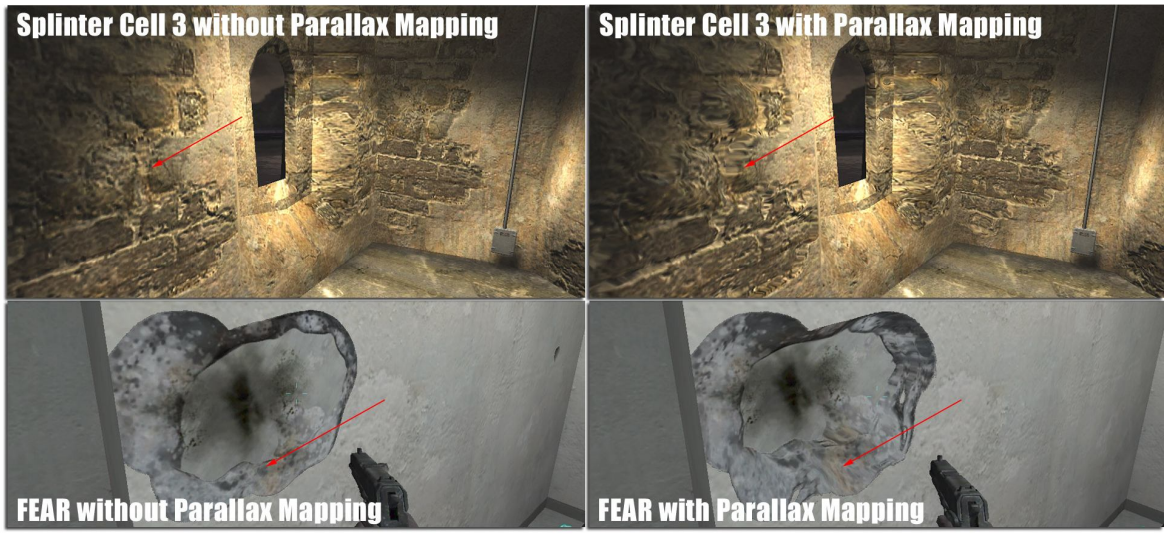
ParallaxMap

Una vez visto el efecto del NormalMapping podemos pasar a ver el efecto de ParallaxMapping que es simplemente una mejora de la técnica anterior dando unos mejores efectos a un coste muy bajo. La técnica del Parallax Mapping consiste en modificar la Normal del objeto mediante la textura del NormalMap, pero añadiendo un nuevo canal en la textura dónde tendremos un mapa de alturas (heightmap) que nos dará la profundidad del modelo recalculando las coordenadas de textura del difuso.

Utilizamos el siguiente píxel shader para calcular la nueva normal y las nuevas coordenadas de textura de difuso.

```
float3 CalcParallaxMap(float3 Vn, float3 WorldNormal, float3 WorldTangent, float3 WorldBinormal,
        float2 UV, out float2 OutUV)
{
        float2 l_UV = UV;
        // parallax code
        float3x3 tbnXf = float3x3(WorldTangent,WorldBinormal,WorldNormal);
        float4 l_NormalMapColor = tex2D(NormalMapTextureSampler,l_UV);
        float height = l_NormalMapColor.w * 0.06 - 0.03;
        l_UV += height * mul(tbnXf,Vn).xy;
        // normal map
        float3 tNorm = l_NormalMapColor.xyz - float3(0.5,0.5,0.5);

        // transform tNorm to world space
        tNorm = normalize(tNorm.x*WorldTangent -
                        tNorm.y*WorldBinormal +
                        tNorm.z*WorldNormal);
        OutUV=l_UV;
        return tNorm;
}
```

Ejemplo de Parallax Mapping.



Texturas de difuso, normal (canales rgb) y mapa de alturas (canal alfa) para generar un ParallaxMapping

EnvironmentMap

Para implementar este tipo de efecto nos basaremos en el tutorial de DirectX "Environment Mapping Explained", el EnvironementMap se basa en reflejar el contorno de un modelo sobre ese modelo.

Para realizar esta tarea deberemos crear una textura cúbica del entorno bien en tiempo de ejecución, bien en producción con lo cuál el resultado quedará falseado. Una textura cúbica es un tipo de textura que contiene la información de seis caras mapeadas, una para cada una de las direcciones de un punto.

Para implementar el Environment Map debemos modificar nuestro shader para implementarlo de la siguiente manera.

// En el vertex shader deberemos calcular el vector reflejo mediante el vector que une el ojo de la cámara con el vértice
float3 EyeToVertex=normalize(OUT.pos.xyz-CameraEye);

```
OUT.Reflect=CalcReflectionVector(EyeToVertex,OUT.WorldNormal);

//La función para calcular el vector reflejo
float3 CalcReflectionVector(float3 ViewToPos, float3 Normal)
{
	return normalize(reflect(ViewToPos, Normal));
}

// En el pixel shader calcularemos el color del reflejo según la textura por el factor de reflejo del material utilizando el siguiente fragmento de código
return g_EnvironmentFactor * texCUBE(EnvironmentSampler, IN.Reflect);
```

CRenderableObjectsVectorMapManager

Para poder gestionar los CRenderableObjects de forma más estructurada vamos a introducir el concepto de layers, es decir deberemos tener los CRenderableObjects en layers diferentes, como vimos en la documentación de MaxScript teníamos una etiqueta dónde decíamos el layer a la que pertenecía el CRenderableObject.

Por consecuencia tendremos una clase que gestionará todos los CRenderableObjectsManager, si antes leíamos un fichero xml dónde introducíamos los CRenderableObjects en la clase CRenderableObjectsManager ahora el fichero xml lo leeremos en la clase CRenderableObjectsLayersManager y desde aquí iremos insertando los CRenderableObjects en el CRenderableObjectsManager correspondiente dependiendo de la capa que nos digan. Deberemos gestionar un fichero .xml como el que vemos a continuación.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<renderable_objects>
  <layer name="solid" default="true"/>
  <layer name="alpha_objects"/>
  <layer name="alpha_blend_objects"/>
  <layer name="particles"/>
  <instance_mesh name="box01" core_name="box01"  pos="-0.964345 12.3275 589.132"
yaw="0.0" pitch="0.0" roll="0.0" scale="1.0 1.0 1.0" visible="true"
create_physics="true" physics_type="triangle_mesh" />
  <instance_mesh name="plane" core_name="plane"  pos="627.34 57.8755 296.19"
yaw="0.0" pitch="0.0" roll="0.0" scale="1.0 1.0 1.0" visible="true"
create_physics="true" physics_type="triangle_mesh" />
  <instance_mesh name="transparent_box" core_name="transparent_box"
layer="alpha_objects" pos="9.34357 8.83837 -176.701" yaw="-1.0472" pitch="-1.5708"
roll="0.0" scale="1.0 1.0 1.0" visible="true" />
</renderable_objects>
```

Implementaremos la clase CRenderableObjectsLayersManager de la siguiente manera.

```
class CRenderableObjectsLayersManager : public
CTemplatedVectorMapManager<CRenderableObjectsManager>
{
private:
	std::string                     m_FileName;
	CRenderableObjectsManager       *m_DefaultRenderableObjectManager;

	CRenderableObjectsManager * GetRenderableObjectManager(xmlTreeNode &Node);
public:
	CRenderableObjectsLayersManager();
	~CRenderableObjectsLayersManager();
```

```
        void Destroy();
        void Load(const std::string &FileName);
        void Reload();
        void Update(float ElapsedTime);
        void Render(CRenderManager *RM);
        void Render(CRenderManager *RM, const std::string &LayerName);
};
```

Encontramos los siguientes métodos:

- *Destroy*, destruirá todos los elementos de la clase.
- *Load,* cargará un fichero xml dónde introducirá todos los elementos en cada uno de los CRenderableObjectsManager dependiendo de la layer.
- *Reload,* recargará el fichero xml.
- *Update,* actualizará todos los elementos CRenderableObjects de todas las layers.
- *Render,* renderizará todos los elementos CRenderableObjects de todas las layers.
- *Render*, renderizará todos los elementos CRenderableObjects de la layer según el nombre.

CRenderableObjectTechnique

La clase CRenderableObjectTechnique nos permitirá enlazar un CRenderableObject con una CEffectTechnique, a partir de ahora los CRenderableObject contendrá una referencia a esta clase y cogerán la technique que deberá utilizar para renderizarse según esta clase.

```
class CRenderableObjectTechnique : public CNamed
{
private:
        CEffectTechnique                        *m_EffectTechnique;
public:
        CRenderableObjectTechnique(const std::string &Name, CEffectTechnique
*EffectTechnique);
        void SetEffectTechnique(CEffectTechnique *EffectTechnique);
        CEffectTechnique * GetEffectTechnique() const;
};
```

CPoolRenderableObjectTechnique

Para implementar un pool de CRenderableObjectTechnique crearemos la clase CPoolRenderableObjectTechnique, esta clase nos permitirá establecer CEffectTechnique sobre una CRenderableObjectTechnique.

Esta clase contiene una clase que contiene una propiedad m_RenderableObjectTechnique que será la CRenderableObjectTechnique que deberemos aplicar sobre la CRenderableObjectTechnique m_OnRenderableObjectTechniqueManager.

```
class CPoolRenderableObjectTechnique : public CNamed
{
private:
```

```
	class CPoolRenderableObjectTechniqueElement
	{
	public:
		CRenderableObjectTechnique m_RenderableObjectTechnique;
		CRenderableObjectTechnique *m_OnRenderableObjectTechniqueManager;

		CPoolRenderableObjectTechniqueElement(const std::string &Name,
CEffectTechnique *EffectTechnique, CRenderableObjectTechnique
*OnRenderableObjectTechniqueManager);
	};
	std::vector<CPoolRenderableObjectTechniqueElement *>
	m_RenderableObjectTechniqueElements;
public:
	CPoolRenderableObjectTechnique(xmlTreeNode &TreeNode);
	virtual ~CPoolRenderableObjectTechnique();
	void Destroy();
	void AddElement(const std::string &Name, const std::string &TechniqueName,
CRenderableObjectTechnique *ROTOnRenderableObjectTechniqueManager);
	void Apply();
};
```

Esta clase contiene los siguientes métodos:

- *Destroy*, como siempre destruiremos los elementos del pool.
- *AddElement*, añadiremos un elemento dentro del pool.
- *Apply*, recorreremos el pool y estableceremos las CEffectTechnique de m_RenderableObjectTechnique sobre el m_OnRenderableObjectTechniqueManager de los elementos.

CRenderableObjectTechniqueManager

La clase principal que gestionará las CEffectTechnique será CRenderableObjectTechniqueManager, esta clase contiene un mapa de CRenderableObjectTechnique vacíos que serán los que utilizarán los CRenderableObjects y que estableceremos según los CPoolRenderableObjectTechnique.

Además contiene un mapa de CPoolRenderableObjectTechnique dónde contendremos las diferentes pools que podemos utilizar.

```
class CRenderableObjectTechniqueManager : public
CTemplatedMapManager<CRenderableObjectTechnique>
{
private:
	CTemplatedMapManager<CPoolRenderableObjectTechnique>
	m_PoolRenderableObjectTechniques;

	void InsertRenderableObjectTechnique(const std::string &ROTName, const
std::string &TechniqueName);
public:
	CRenderableObjectTechniqueManager();
	virtual ~CRenderableObjectTechniqueManager();
	void Destroy();
	void Load(const std::string &FileName);
	std::string GetRenderableObjectTechniqueNameByVertexType(unsigned int
VertexType);
	CTemplatedMapManager<CPoolRenderableObjectTechnique> &
GetPoolRenderableObjectTechniques();
};
```

Esta clase contiene los siguientes métodos:

- *Constructor,* construye la clase CRenderableObjectTechniqueManager.
- *Destructor,* destruye la clase CRenderableObjectTechniqueManager llamando al método Destroy.
- *Destroy,* destruye todos los CPoolRenderableObjectTechnique y los CRenderableObjectTechnique.
- *Load,*carga un fichero xml que rellenará nuestra estructura de datos.
- *InsertRenderableObjectTechnique*, introducirá un CRenderableObjectTechnique en el mapa según un nombre de renderable object technique y el nombre de la technique que utilizará.
- *GetRenderableObjectTechniqueNameByVertexType,* nos devolverá un nombre de RenderableObjectTechnique según el tipo de Vértice, simplemente podría devolver algo como “DefaultROTTechnique_31” para el vertex type 31.
- *GetPoolRenderableObjectTechniques,* nos devuelve el mapa de CPoolRenderableObjectTechniques.

Esta clase leerá un fichero xml similar al siguiente.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<renderable_object_techniques>
	<pool_renderable_object_technique name="dummy_manager">
		<default_technique vertex_type="31"
technique="NormalMapAmbientLightNormalTextureVertexTechnique"/>
		<default_technique vertex_type="19"
technique="AmbientLightNormalTextureVertexTechnique"/>
		<default_technique vertex_type="159" technique="Cal3DTechnique"/>

		<default_technique vertex_type="35"
technique="AmbientLightNormalTextureVertexTechnique"/>
	</pool_renderable_object_technique>
</renderable_object_techniques>
```

CStaticMesh

Con la nueva implementación de CEffectTechnique deberemos pedirle al cargar la CStaticMesh el CRenderableObjectTechnique al manager CRenderableObjectTechniqueManager.

Podríamos implementar el código siguiente dentro de la clase CStaticMesh.

```
bool CStaticMesh::GetRenderableObjectTechnique()
{
	CRenderableObjectTechniqueManager *l_ROTM= C3DEngine::GetInstance()
.GetRenderableObjectsTechniqueManager();
	bool l_Ok=true;
	for(size_t i=0;i<m_VertexTypes.size();++i)
	{
		if(m_RenderableObjectTechniqueName=="")
			m_RenderableObjectTechniqueName=l_ROTM-
>GetRenderableObjectTechniqueNameByVertexType(m_VertexTypes[i]);

		CRenderableObjectTechnique *l_ROT=l_ROTM-
>GetResource(m_RenderableObjectTechniqueName);
```

```
			m_RenderableObjectTechniques.push_back(l_ROT);
			if(l_ROT==NULL)
				Info("Error trying to GetRenderableObjectTechnique '%s' on
CStaticMesh", m_RenderableObjectTechniqueName.c_str());
			l_Ok=l_Ok && l_ROT!=NULL;
		}
		return l_Ok;
}
```

Deberemos implementar algo similar también para la clase CAnimatedInstanceModel.

CSceneRendererCommandManager

A continuación vamos a realizar un cambio drástico en nuestra pipeline de renderizado, hasta ahora hemos implementado directamente en código C++ el código de renderizado de la escena dentro del método RenderScene del proceso actual.

Lo que vamos a hacer ahora, es intentar externalizar todo el sistema de renderizado de manera que podamos modificar las fases de renderizado, sin necesidad de tocar ningún fichero de c++ y sin necesidad de tener que recompilar el código y con las consiguiente posibilidad de realizarlo todo en caliente.

Para ello vamos a realizar la clase CSceneRenderCommandManager, esta clase va a tener todos los comandos de renderizado que deberemos realizar para renderizar mi escena actual.

La clase será capaz de leer un fichero xml dónde le diremos todos los comandos a realizar para conseguir el render de nuestro frame.

```
class CSceneRendererCommandManager
{
private:
		CTemplatedVectorMapManager<CSceneRendererCommand> m_SceneRendererCommands;

		void CleanUp();
		std::string GetNextName();
public:
		CSceneRendererCommandManager();
		~CSceneRendererCommandManager();
		void Load(const std::string &FileName);
		void Execute(CRenderManager &RM);
};
```

Esta clase tiene los siguientes métodos:

- *CleanUp,* elimina todos los elementos del vector de elementos CSceneRendererCommand.
- *GetNextName*, en caso de no tener nombre el SceneRendererCommand nos da uno según el índice total de elementos en el vector
- *Load,* carga un fichero xml dónde encontraremos todos los comandos de renderizado de nuestra escena
- *Execute,* ejecuta toda la secuencia de comandos de renderizado

Destacar que esta clase contiene una clase variable miembro m_SceneRendererCommands que es una clase templatizada CTemplatedVectorMapManager que nos permite tener un vector y un mapa de elementos para poder buscarlos por nombre y recorrerlos como un vector.

Un ejemplo de fichero xml de scene_renderer_commands.xml sería el siguiente.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<scene_renderer_commands>
  <!--pre_render-->
              <begin_scene/>
              <clear_scene color="true" depth="true" stencil="true"/>
              <enable_z_write/>
              <enable_z_test/>
              <set_pool_renderable_objects_technique
pool="generate_shadow_map_renderable_obejcts_technique"/>
              <generate_shadow_maps />
              <set_pool_renderable_objects_technique
pool="generate_deferred_shading_pool_renderable_object_technique"/>
              <set_render_target name="deferred_multiple_render_target">
                     <dynamic_texture stage_id="0" name="DiffuseMapTexture"
texture_width_as_frame_buffer="true" format_type="A8R8G8B8"/>
                     <dynamic_texture stage_id="1" name="LightMapTexture"
texture_width_as_frame_buffer="true" format_type="A8R8G8B8"/>
                     <dynamic_texture stage_id="2" name="NormalMapTexture"
texture_width_as_frame_buffer="true" format_type="A8R8G8B8"/>
                     <dynamic_texture stage_id="3" name="DepthMapTexture"
texture_width_as_frame_buffer="true" format_type="R32F"/>
              </set_render_target>
              <clear_scene color="true" depth="true" stencil="true"/>
              <enable_z_write/>
              <enable_z_test/>

              <render_scene layer="solid" active="true"/>

              <unset_render_target render_target="deferred_multiple_render_target"/>
              <disable_z_write/>

              <set_render_target name="ssao_multiple_render_target">
                     <dynamic_texture stage_id="1" name="SSAOMapTextureBlur"
texture_width_as_frame_buffer="true" format_type="A8R8G8B8"/>
              </set_render_target>
              <set_pool_renderable_objects_technique
pool="ssao_pool_renderable_object_technique"/>
              <render_draw_quad>
                     <texture stage_id="0" file="data/textures/RandomNormal.dds"
load_file="true"/>
                     <texture stage_id="2" file="NormalMapTexture"/>
                     <texture stage_id="3" file="DepthMapTexture"/>
              </render_draw_quad>

              <capture_frame_buffer>
                     <dynamic_texture stage_id="1" name="SSAOMapTexture"
texture_width_as_frame_buffer="true" format_type="A8R8G8B8"/>
              </capture_frame_buffer>

              <set_pool_renderable_objects_technique
pool="ssao_blur_pool_renderable_object_technique"/>
              <render_draw_quad>
                     <texture stage_id="0" file="SSAOMapTexture"/>
                     <texture stage_id="2" file="NormalMapTexture"/>
              </render_draw_quad>
              <unset_render_target render_target="ssao_multiple_render_target"/>
              <end_scene/>
  <!--end pre_render-->
       <disable_z_write/>
       <set_pool_renderable_objects_technique
pool="draw_quad_deferred_shading_hemispheric_ambient_pool_renderable_object_techniq
ue"/>
       <render_draw_quad active="true">
```

```
            <texture stage_id="0" file="DiffuseMapTexture"/>
            <texture stage_id="1" file="LightMapTexture"/>
            <texture stage_id="2" file="NormalMapTexture"/>
            <texture stage_id="3" file="DepthMapTexture"/>
      </render_draw_quad>
      <set_pool_renderable_objects_technique
pool="draw_quad_deferred_shading_per_light_pool_renderable_object_technique"/>
      <render_deferred_shading>
            <texture stage_id="0" file="DiffuseMapTexture"/>
            <texture stage_id="1" file="LightMapTexture"/>
            <texture stage_id="2" file="NormalMapTexture"/>
            <texture stage_id="3" file="DepthMapTexture"/>
      </render_deferred_shading>

      <enable_z_test/>
      <render_debug_scene layer="solid" active="true"/>
      <render_debug_lights active="true"/>
      <set_pool_renderable_objects_technique
pool="alpha_blend_pool_renderable_object_technique"/>
      <render_scene layer="alpha_objects" active="true"/>

      <capture_frame_buffer>
            <dynamic_texture stage_id="0" name="FrameBufferAfterDeferredShading"
texture_width_as_frame_buffer="true" format_type="A8R8G8B8"/>
      </capture_frame_buffer>

      <!--ssaoo-->
      <set_pool_renderable_objects_technique
pool="ssao_final_composition_pool_renderable_object_technique"/>
      <render_draw_quad active="true">
            <texture stage_id="0" file="FrameBufferAfterDeferredShading"/>
            <texture stage_id="1" file="SSAOMapTextureBlur"/>
      </render_draw_quad>
      <!--end ssaoo-->
      <!--fog-->
      <set_pool_renderable_objects_technique
pool="fog_pool_renderable_object_technique"/>
      <render_draw_quad active="true">
            <texture stage_id="3" file="DepthMapTexture"/>
      </render_draw_quad>
      <!--end fog-->

      <!--zblur-->
      <capture_frame_buffer>
        <dynamic_texture stage_id="0" name="FrameBufferCompleteTexture"
texture_width_as_frame_buffer="true" format_type="A8R8G8B8"/>
      </capture_frame_buffer>
      <set_pool_renderable_objects_technique
pool="z_blur_pool_renderable_object_technique"/>
      <render_draw_quad>
            <texture stage_id="0" file="DepthMapTexture"/>
            <texture stage_id="1" file="FrameBufferCompleteTexture"/>
      </render_draw_quad>
      <!--end zblur-->
      <!--noise and vignetting-->
      <set_pool_renderable_objects_technique
pool="noise_and_vignetting_pool_renderable_object_technique"/>
      <render_draw_quad name="noise_and_vignetting_scene_effect" active="true">
            <texture stage_id="0" file="data/gui/textures/noise.tga"
load_file="true"/>
            <texture stage_id="1" file="data/gui/textures/vignetting.tga"
load_file="true"/>
      </render_draw_quad>
      <!--end noise and vignetting-->
```

```
        <!--render shadow_maps_debug-->
        <set_pool_renderable_objects_technique
pool="draw_shadow_map_2d_debug_pool_renderable_object_technique"/>
        <render_debug_shadow_maps screen_width="0.15" screen_height="0.15"/>
        <!--end render shadow_maps_debug-->
        <render_gui active="true"/>
        <present/>
</scene_renderer_commands>
```

CSceneRendererCommand

Para implementar los diferentes comando de render de escena vamos a utilizar una clase base CSceneRenderCommand que nos permitirá gestionar los diferentes comandos de render.

La clase sería como vemos a continuación.

```
class CSceneRendererCommand : public CUABActive, public CNamed
{
public:
        CSceneRendererCommand(xmlTreeNode &atts);
        virtual ~CSceneRendererCommand();
        virtual void Execute(CRenderManager &RM) = 0;
};
```

Esta clase tiene los siguientes métodos:

- *Constructor,* recibe por un conjunto de atributos xml los atributos de esta clase
- *Execute*, contiene el código para ejecutar el comando de renderizado

CClearSceneRendererCommand

Esta clase nos permitirá hacer un clear del device.

```
class CClearSceneRendererCommand : public CSceneRendererCommand
{
protected:
        bool                    m_Color;
        bool                    m_Depth;
        bool                    m_Stencil;
public:
        CClearSceneRendererCommand(xmlTreeNode &atts);
        virtual void Execute(CRenderManager &RM);
};
```

CBeginRenderSceneRendererCommand

Esta clase nos permitirá hacer un begin render del device.

```
class CBeginRenderSceneRendererCommand : public CSceneRendererCommand
{
public:
        CBeginRenderSceneRendererCommand(xmlTreeNode &atts);
        void Execute(CRenderManager &RM);
};
```

CEndRenderSceneRendererCommand

Esta clase nos permitirá hacer un end render del device.

```
class CEndRenderSceneRendererCommand : public CSceneRendererCommand
{
public:
	CEndRenderSceneRendererCommand (xmlTreeNode &atts);
	void Execute(CRenderManager &RM);
};
```

CPresentSceneRendererCommand

Esta clase realizará el present del device.

```
class CPresentSceneRendererCommand : public CSceneRendererCommand
{
public:
	CPresentSceneRendererCommand(xmlTreeNode &atts);
	void Execute(CRenderManager &RM);
};
```

CSetMatricesSceneRendererCommand

Esta clase realizará el set de las matrices de view y projection de la cámara activa.

```
class CSetMatricesSceneRendererCommand : public CSceneRendererCommand
{
public:
	CSetMatricesSceneRendererCommand(xmlTreeNode &atts);
	void Execute(CRenderManager &RM);
};
```

CDisableZWriteSceneRendererCommand

Esta clase nos permitirá deshabilitar el zwrite del device.

```
class CDisableZWriteSceneRendererCommand : public CSceneRendererCommand
{
public:
	CDisableZWriteSceneRendererCommand(xmlTreeNode &atts);
	void Execute(CRenderManager &RM);
};
```

CEnableZWriteSceneRendererCommand

Esta clase nos permitirá habilitar el zwrite del device.

```
class CEnableZWriteSceneRendererCommand : public CSceneRendererCommand
{
public:
	CEnableZWriteSceneRendererCommand(xmlTreeNode &atts);
	void Execute(CRenderManager &RM);
};
```

CDisableZTestSceneRendererCommand

Esta clase nos permitirá deshabilitar el ztest del device.

```
class CDisableZTestSceneRendererCommand : public CSceneRendererCommand
{
public:
	CDisableZTestSceneRendererCommand(xmlTreeNode &atts);
	void Execute(CRenderManager &RM);
};
```

CEnableZTestSceneRendererCommand

Esta clase nos permitirá habilitar el ztest del device.

```
class CEnableZTestSceneRendererCommand : public CSceneRendererCommand
{
public:
	CEnableZTestSceneRendererCommand(xmlTreeNode &atts);
	void Execute(CRenderManager &RM);
};
```

CRenderSceneSceneRendererCommand

Esta clase nos permitirá renderizar un RenderableObjectManager determinado.

```
class CRenderSceneSceneRendererCommand : public CSceneRendererCommand
{
private:
	CRenderableObjectsManager	*m_Layer;
public:
	CRenderSceneSceneRendererCommand(xmlTreeNode &atts);
	void Execute(CRenderManager &RM);
};
```

CRenderDebugSceneSceneRendererCommand

Esta clase nos permitirá renderizar el debug de un RenderableObjectManager determinado.

```
class CRenderDebugSceneSceneRendererCommand : public CSceneRendererCommand
{
private:
	CRenderableObjectsManager		*m_Layer;
public:
	CRenderDebugSceneSceneRendererCommand(xmlTreeNode &atts);
	void Execute(CRenderManager &RM);
};
```

CRenderDebugLightsSceneRendererCommand

Esta clase nos permitirá renderizar el debug de las luces.

```
class CRenderDebugLightsSceneSceneRendererCommand : public CSceneRendererCommand
{
private:
public:
	CRenderDebugLightsSceneSceneRendererCommand(xmlTreeNode &atts);
	void Execute(CRenderManager &RM);
};
```

CRenderGUISceneRendererCommand

Esta clase nos permitirá renderizar una gui determinada.

```
class CRenderGUISceneRendererCommand : public CSceneRendererCommand
{
private:
	CGUI		*m_GUI;
public:
	CRenderGUISceneRendererCommand(xmlTreeNode &atts);
	void Execute(CRenderManager &RM);
};
```

CStagedTexturedRendererCommand

Esta clase nos permitirá derivar de ella y podremos asignar diferentes texturas según etapas en un RendererCommand.

```
class CStagedTexturedRendererCommand : public CSceneRendererCommand
{
protected:
	class CKGStageTexture
	{
	public:
		int			m_StageId;
		CTexture		*m_Texture;
		CKGStageTexture(int StageId, CTexture *Texture)
		{
			m_StageId=StageId;
			m_Texture=Texture;
		}
		void Activate();
	};

	std::vector<CKGStageTexture>		m_StageTextures;
public:
	CStagedTexturedRendererCommand(xmlTreeNode &atts);
	virtual ~CStagedTexturedRendererCommand();
	void ActivateTextures();
	void AddStageTexture(int StageId, CTexture *Texture);
	virtual void Execute(CRenderManager &RM) = 0;
};
```

CDrawQuadRendererCommand

Esta clase nos permitirá renderizar un cuadro en 2D según un color.

```
class CDrawQuadRendererCommand : public CStagedTexturedRendererCommand
{
protected:
	CColor			m_Color;
public:
	CDrawQuadRendererCommand(xmlTreeNode &atts);
	void Execute(CRenderManager &RM);
};
```

Esta clase nos permitirá renderizar un cuadro en 2D según un color. El código necesario para poder renderizar un quad 2D con una CEffectTechnique sería el siguiente.

```
void DrawColoredQuad2DTexturedInPixelsByEffectTechnique(CRenderManager *RM,
CEffectTechnique *EffectTechnique, RECT Rect, CColor Color, CTexture *Texture,
float U0=0.0f, float V0=0.0f, float U1=1.0f, float V1=1.0f)
{
	EffectTechnique->BeginRender();
```

```
	LPD3DXEFFECT l_Effect=EffectTechnique->GetEffect()->GetD3DEffect();
	if(l_Effect!=NULL)
	{
		l_Effect->SetTechnique(EffectTechnique->GetD3DTechnique());
		UINT l_NumPasses;
		l_Effect->Begin(&l_NumPasses, 0);
		for (UINT iPass = 0; iPass < l_NumPasses; iPass++)
		{
			l_Effect->BeginPass(iPass);
			RM->DrawColoredQuad2DTexturedInPixels(Rect,Color,Texture, U0,
V0, U1, V1);
			l_Effect->EndPass();
		}
		l_Effect->End();
	}
}
```

La technique que crearíamos en HLSL para poder renderizar un quad sería similar al siguiente código.

```
technique DrawQuadTechnique
{
    pass p0
    {
		AlphaBlendEnable = false;
		CullMode = CCW;
		PixelShader = compile ps_3_0 DrawQuadPS();
    }
}
```

CRenderableObjectTechniquesSceneRendererCommand

Esta clase nos permitirá establecer un pool de techniques para que los elementos se rendericen a través de las techniques según los tipos de vértice.

```
class CRenderableObjectTechniquesSceneRendererCommand : public
CSceneRendererCommand
{
private:
	CPoolRenderableObjectTechnique	*m_PoolRenderableObjectTechnique;
public:
	CRenderableObjectTechniquesSceneRendererCommand(xmlTreeNode &atts);
	void Execute(CRenderManager &RM);
};
```

CSetRenderTargetSceneRendererCommand

Esta clase nos permitirá establecer las texturas de la clase CStagedTextureRendererCommand que derivamos como render target. Además tiene el métdo que hace el unset.

```
class CSetRenderTargetSceneRendererCommand : public CStagedTexturedRendererCommand
{
public:
	CSetRenderTargetSceneRendererCommand(xmlTreeNode &atts);
	void Execute(CRenderManager &RM);
	void UnsetRenderTarget();
};
```

CUnsetRenderTargetSceneRendererCommand

Esta clase nos permitirá desestablecer las texturas del render target. Para ello llamamos al método UnsetRenderTarget de nuestra variable miembro m_SetRenderTargetRendererCommand.

```
class CUnsetRenderTargetSceneRendererCommand : public CSceneRendererCommand
{
private:
	CSetRenderTargetSceneRendererCommand	*m_SetRenderTargetRendererCommand;
public:
	CUnsetRenderTargetSceneRendererCommand(CSetRenderTargetSceneRendererCommand
*SetRenderTargetRendererCommand, xmlTreeNode &atts);
	void Execute(CRenderManager &RM);
};
```

CDeferredShadingSceneRendererCommand

Esta clase nos permitirá renderizar las luces en modo deferred según las texturas de la clase CStagedTexturedRendererCommand.

```
class CDeferredShadingSceneRendererCommand : public CStagedTexturedRendererCommand
{
private:
	CRenderableObjectTechnique		*m_RenderableObjectTechnique;

	void SetLightsData(CRenderManager &RM);
public:
	CDeferredShadingSceneRendererCommand(xmlTreeNode &atts);
	void Execute(CRenderManager &RM);
};
```

CCaptureFrameBufferSceneRendererCommand

Esta clase nos permite capturar el frame actual y copiarlo en la textura 0 de la clase CStagedTextureRendererCommand.

```
class CCaptureFrameBufferSceneRendererCommand : public
CStagedTexturedRendererCommand
{
public:
	CCaptureFrameBufferSceneRendererCommand(xmlTreeNode &atts);
	void Execute(CRenderManager &RM);
};
```

CGenerateShadowMapsSceneRendererCommand

Esta clase nos permite generar los shadow maps de las luces que proyecten sombras.

```
class CGenerateShadowMapsSceneRendererCommand : public CSceneRendererCommand
{
public:
	CGenerateShadowMapsSceneRendererCommand(xmlTreeNode &atts);
	void Execute(CRenderManager &RM);
};
```

CRenderDebugShadowMapsSceneRendererCommand

Esta clase nos permite renderizar en 2D las texturas de shadowmap de las diferentes luces que generan sombras dinámicas.

```
class CRenderDebugShadowMapsSceneRendererCommand : public CSceneRendererCommand
{
protected:
	float 			m_ScreenWidth, m_ScreenHeight;

	void AdvanceTexturePosition(Vect2f &Position);
public:
	CRenderDebugShadowMapsSceneRendererCommand(xmlTreeNode &atts);
	void Execute(CRenderManager &RM);
};
```

Método Render en CProc

Para poder terminar la implementación de los scene renderer commands deberemos modificar el método Render del proceso por.

```
void C3DProc::Render(CRenderManager *RM)
{
	CORE->GetSceneRendererCommandManager()->ExecuteRender(RM);
}
```

Métodos CTexture

Para poder implementar algunos comandos de escena vamos a modificar la clase CTexture dándole unas nuevas funcionalidades como son poder crear una textura de forma dinámica y poder establecerlas como RenderTarget.

```
enum TPoolType {
	DEFAULT=0,
	SYSTEMMEM
};
enum TUsageType {
	DYNAMIC=0,
	RENDERTARGET
};
enum TTextureType {
	TGA=0,
	JPG,
	BMP
};
enum TFormatType {
	A8R8G8B8=0,
	R8G8B8,
	X8R8G8B8,
	R32F
};
bool Create(const std::string &Name, unsigned int Width, unsigned int
Height, unsigned int MipMaps, TUsageType UsageType, TPoolType PoolType,
TFormatType FormatType)
{
	SetName(Name);
	D3DPOOL l_Pool=D3DPOOL_DEFAULT;
	DWORD l_UsageType=D3DUSAGE_DYNAMIC;
	D3DFORMAT l_Format=D3DFMT_A8R8G8B8;
```

```
	bool l_CreateDepthStencilSurface=false;
	switch(UsageType)
	{
		case DYNAMIC:
			l_UsageType=D3DUSAGE_DYNAMIC;
			break;
		case RENDERTARGET:
			l_CreateDepthStencilSurface=true;
			l_UsageType=D3DUSAGE_RENDERTARGET;
			break;
	}
	switch(PoolType)
	{
		case DEFAULT:
			l_Pool=D3DPOOL_DEFAULT;
			break;
		case SYSTEMMEM:
			l_Pool=D3DPOOL_SYSTEMMEM;
			break;
	}
	switch(FormatType)
	{
		case A8R8G8B8:
			l_Format=D3DFMT_A8R8G8B8;
			break;
		case R8G8B8:
			l_Format=D3DFMT_R8G8B8;
			break;
		case X8R8G8B8:
			l_Format=D3DFMT_X8R8G8B8;
			break;
		case R32F:
			l_Format=D3DFMT_R32F;
			break;
	}
	HRESULT hr=CORE.GetRenderManager().GetDevice()->CreateTexture(
		Width, Height, MipMaps, l_UsageType, l_Format, l_Pool,
		&m_Texture,NULL);
	if(l_CreateDepthStencilSurface)
	{
		CORE->GetRenderManager().GetDevice()->
			CreateDepthStencilSurface(Width, Height, D3DFMT_D24S8,
			D3DMULTISAMPLE_NONE, 0, TRUE,
			&m_DepthStencilRenderTargetTexture, NULL);
		assert(m_DepthStencilRenderTargetTexture!=NULL);
	}
	assert(m_Texture!=NULL);
	assert(hr==D3D_OK);
	m_Width=Width;
	m_Height=Height;
	return hr!=D3D_OK;
}
```

Desactivar una etapa de textura crearemos el siguiente método estático.

```
void CTexture::Deactivate(size_t Stage)
{
	PSRender.GetDevice()->SetTexture((DWORD)Stage,NULL);
}
```

Establecer una textura como RenderTarget, cuando rendericemos mediante el device se renderizará directamente sobre la textura.

```
bool CTexture::SetAsRenderTarget(size_t IdStage=0)
{
      LPDIRECT3DDEVICE9 l_Device= CORE->GetRenderManager()->GetDevice();
      l_Device->GetRenderTarget((DWORD)IdStage, &m_OldRenderTarget);
      if(FAILED( m_Texture->GetSurfaceLevel( 0, &m_RenderTargetTexture )
) )
            return false;
    l_Device->SetRenderTarget( (DWORD)IdStage, m_RenderTargetTexture );
      CHECKED_RELEASE(m_RenderTargetTexture);
      if(FAILED( l_Device->GetDepthStencilSurface(
&m_OldDepthStencilRenderTarget ) ) )
            return false;

      l_Device->SetDepthStencilSurface(
m_DepthStencilRenderTargetTexture );

      return true;
}
```

Desestablecer una textura como RenderTarget.

```
void CTexture::UnsetAsRenderTarget(size_t IdStage=0)
{
      LPDIRECT3DDEVICE9 l_Device=CORE->GetRenderManager().GetDevice();

      l_Device->SetDepthStencilSurface(m_OldDepthStencilRenderTarget);
    CHECKED_RELEASE(m_OldDepthStencilRenderTarget);
      l_Device->SetRenderTarget(IdStage, m_OldRenderTarget);
    CHECKED_RELEASE(m_OldRenderTarget);
}
```

Para capturar el frame buffer en una textura utilizaremos el siguiente código.

```
void CTexture::CaptureFrameBuffer(size_t IdStage)
{
      LPDIRECT3DDEVICE9 l_Device=CORE->GetRenderManager()->GetDevice();
      LPDIRECT3DSURFACE9 l_RenderTarget, l_Surface;

      m_Texture->GetSurfaceLevel(0,&l_Surface);
      l_Device->GetRenderTarget(IdStage,&l_RenderTarget);
      l_Device->StretchRect(l_RenderTarget,NULL, l_Surface,NULL,D3DTEXF_NONE);
      l_RenderTarget->Release();
}
```

Por último para extraer el tipo de una textura a partir de un string.

```
CTexture::TFormatType CTexture::GetFormatTypeFromString(const
std::string &FormatType)
{
      if(FormatType=="R32F")
            return CTexture::R32F;
      else if(FormatType=="A8R8G8B8")
            return CTexture::A8R8G8B8;
      else if(FormatType=="R8G8B8")
            return CTexture::R8G8B8;
      else if(FormatType=="X8R8G8B8")
            return CTexture::X8R8G8B8;
      else
            Info("Format Type '%s' not recognized", FormatType.c_str());
      return CTexture::A8R8G8B8;
}
```

CLight

Para poder implementar sombras dinámicas en nuestro juego utilizaremos la técnica del shadowmap, para ello vamos a necesitar nuevos atributos en la clase de luces, añadiremos los siguientes métodos y atributos en la clase.

```
class CLight : public C3DObject, public CNamed
{
protected:
	bool                                m_GenerateDynamicShadowMap;
	bool                                m_GenerateStaticShadowMap;
	bool                                m_MustUpdateStaticShadowMap;
	CTexture     *m_StaticShadowMap, *m_DynamicShadowMap, *m_ShadowMaskTexture;
	std::vector<CRenderableObjectsManager *>
	m_StaticShadowMapRenderableObjectsManagers,
m_DynamicShadowMapRenderableObjectsManagers;
	Mat44f              m_ViewShadowMap, m_ProjectionShadowMap;
public:
	virtual void SetShadowMap(CRenderManager *RM)=0;
	void SetGenerateDynamicShadowMap(bool GenerateDynamicShadowMap);
	bool GetGenerateDynamicShadowMap() const;
	void SetGenerateStaticShadowMap(bool GenerateStaticShadowMap);
	bool GetGenerateStaticShadowMap() const;
	void SetMustUpdateStaticShadowMap(bool MustUpdateStaticShadowMap);
	bool GetMustUpdateStaticShadowMap() const;
	CTexture * GetStaticShadowMap() const;
	CTexture * GetDynamicShadowMap() const;
	CTexture * GetShadowMaskTexture() const;
	std::vector<CRenderableObjectsManager *> &
GetStaticShadowMapRenderableObjectsManagers();
	std::vector<CRenderableObjectsManager *> &
GetDynamicShadowMapRenderableObjectsManagers();
	void GenerateShadowMap(CRenderManager *RM);
	const Mat44f & GetViewShadowMap() const;
	const Mat44f & GetProjectionShadowMap() const;
	void BeginRenderEffectManagerShadowMap(CEffect *Effect);
};
```

Esta clase contiene los siguientes atributos:

- *m_GenerateDynamicShadowMap*, nos dirá si el shadowmp lo generaremos cada frame
- *m_GenerateStaticShadowMap,* nos dirá si creamos una textura de shadowmap con los objetos estáticos que no deberemos generar por cada frame.
- *m_MustUpdateStaticShadowMap,* simplemente nos dirá si debemos volver a realizar el shadowmap de las mallas estáticas.
- *m_StaticShadowMap*, contiene la textura de shadowmap para los elementos estáticos.
- *m_DynamicShadowMap,* contiene la textura de shadowmap para los elementos dinámicos.
- *m_ShadowMaskTexture*, contiene una textura que nos hará de máscara para la proyección de la sombra.
- *m_StaticShadowMapRenderableObjectsManager*, contiene un vector de los CRenderableObjectsManagers que debemos renderizar para generar el StaticShadowMap.

- *m_DynamicShadowMapRenderableObjectsManager*, contiene un vector de los CRenderableObjectsManagers que debemos renderizar para generar el DynamicShadowMap.
- *m_ViewShadowMap, m_ProjectionShadowMap,* contienen la matriz de projection y de view del shadowmap.

Esta clase contiene los siguientes métodos:

- *SetShadowMap,* es un método virtual puro que deberemos implementar en todas las luces que deriven de CLight y que establecerá las matrices de view y de projection según el tipo de luz.
- *Set/GetGenerateDynamicShadowMap,* nos devuelve o establece el valor de m_DynamicShadowMap.
- *Set/GetGenerateStaticShadowMap,* nos devuelve o establece el valor de m_StaticShadowMap.
- *Set/GetMustUpdateStaticShadowMap,* nos devuelve o establece el valor de m_MustUpdateStaticShadowMap.
- *GetStaticShadowMapRenderableObjectsManagers,* nos devuelve el vector de CRenderableObjectsManager que debemos renderizar para generar el shadowmap de mallas estáticas.
- *GetDynamicShadowMapRenderableObjectsManager,* nos devuelve el vector de CRenderableObjectsManager que debemos renderizar para generar el shadowmap de mallas dinámicas.
- *GenerateShadowMap,* implementa el código para generar los shadowmaps, tanto los estáticos como los dinámicos en caso de deber generarse.
- *GetViewShadowMap,* devuelve la matriz de view del shadowmap.
- *GetProjectionShadowMap,* devuelve la matriz de projection del shadowmap.
- *BeginRenderEffectManagerShadowMap,* establece en el effect las matrices de View y de Projection de la shadow, activa las texturas de los shadow maps tanto estático, dinámico como de máscara y por último establece los booleanos de las texturas de shadowmap que estamos utilizando (estático, dinámico o máscara).

El código del método BeginRenderEffectManagerShadowMap de la luz sería como el siguiente:

```
void CLight::BeginRenderEffectManagerShadowMap(CEffect *Effect)
{
	if(m_GenerateDynamicShadowMap)
	{
		CEffectManager &l_EM=CORE->GetEffectManager();
		l_EM.SetLightViewMatrix(m_ViewShadowMap);
		l_EM.SetShadowProjectionMatrix(m_ProjectionShadowMap);

		if(m_ShadowMaskTexture!=NULL)
			m_ShadowMaskTexture->Activate(SHADOW_MAP_MASK_STAGE);
		if(m_GenerateStaticShadowMap)
			m_StaticShadowMap->Activate(STATIC_SHADOW_MAP_STAGE);
		m_DynamicShadowMap->Activate(DYNAMIC_SHADOW_MAP_STAGE);
```

```
            Effect->SetShadowMapParameters(m_ShadowMaskTexture!=NULL,
m_GenerateStaticShadowMap, m_GenerateDynamicShadowMap &&
m_DynamicShadowMapRenderableObjectsManagers.size()!=0);
      }
}
```

El código del método SetShadowMapParameters de la clase CEffect sería como el siguiente:

```
void CEffect::SetShadowMapParameters(bool UseShadowMaskTexture, bool
UseStaticShadowmap, bool UseDynamicShadowmap)
{
      m_Effect->SetBool(m_UseShadowMaskTextureParameter, UseShadowMaskTexture ?
TRUE : FALSE);
      m_Effect->SetBool(m_UseStaticShadowmapParameter, UseStaticShadowmap ? TRUE :
FALSE);
      m_Effect->SetBool(m_UseDynamicShadowmapParameter, UseDynamicShadowmap ? TRUE
: FALSE);
}
```

Los nuevos parámetros los leeremos de la luz que nos vendrá por xml, debiendo exportar los nuevos parámetros desde MaxScript.

```
<light name="mylight2" type="spot" pos="169 18 14" dir="0.39344111 0.0096235331
0.91929936" color="1.0 1.0 1.0" att_start_range="20" att_end_range="40"
generate_shadow_map="true" shadow_map_format_type="R32F" shadow_map_width="256"
shadow_map_height="256" generate_static_shadow_map="true"
static_shadow_map_format_type="R32F" static_shadow_map_width="256"
static_shadow_map_height="256" angle="1.05" fall_off="1.05"
shadow_texture_mask="./data/textures/shadow_mask.tga">
            <static renderable_objects_manager="solid"/>
            <dynamic renderable_objects_manager="solid"/>
</light>
```

CDirectionalLight

En la clase CDirectionalLight deberemos introducir el tamaño para la cámara ortogonal del shadowmap de la luz, ese tamaño será en escala de mundo en 2 dimensiones ancho y alto. El valor lo asignaremos en el xml de la luz.

```
class CDirectionalLight : public CLight
{
protected:
      Vect2f                  m_OrthoShadowMapSize;
};
```

Para crear la matriz de projection de una luz direccional deberemos utilizar una matriz ortogonal, para ello utilizaremos el método de DirectX D3DXMatrixOrthoLH(D3DXMATRIX *pOut, FLOAT w, FLOAT h, FLOAT zn, FLOAT zf). Dónde los parámetros w y h defines el ancho y alto de la cámara ortogonal y el zn y el zf definen los planos near y far de la cámara.

CSpotLight

En la clase CSpotLight deberemos introducir generar la matriz de proyección según el FOV y el aspect ratio de la luz. Para ello utilizaremos el código estándar de creación de matriz de proyección.

ShadowMap

ShadowMap es una técnica utilizada para el renderizado de sombras en tiempo real, mediante el uso de una textura con la información de las Z's creada desde el punto de vista de la luz, esta sombra se proyecta sobre la escena dando el sombreado correspondiente.

Para la realización de este tipo de efecto vamos a necesitar renderizar la escena en dos pasadas.

- La primera pasada la renderizaremos desde el punto de vista de la cámara y guardaremos en la textura la información de distancia de ese píxel hasta la cámara en vez del color.
- La segunda pasada, renderizaremos la escena normalmente y calcularemos de nuevo la distancia de ese píxel respecto a la luz y la compararemos con la distancia escrita en ese punto en el shadowmap, si nuestra distancia es mayor que en el shadowmap quiere decir que estamos tapados por la sombra de otro objeto.

La textura que utilizaremos no tendrá el formato típico de RGBA si no que utilizaremos un formato de DirectX D3DFMT_R32F en el cual ser guardarán las Z's como un único float de 32 bytes.

Para la realización de este efecto nos vamos a basar en el ejemplo que trae DirectX de shadowmap.

Para implementar esta técnica vamos a utilizar el SceneRendererCommand CGenerateShadowMapsSceneRendererCommand el cual recorrerá todas las luces que deben generar shadowmap y lo crearán según las propiedades de las luces.

Para ello estableceremos los valores de cámara según la luz, es decir crearemos la matriz de View y de Projection a través de los parámetros de la cámara en el método SetShadowMap que dependerá del tipo de luz.

La matriz View la crearemos a través del Eye, LookAt y vector Up de la cámara.

La matriz Projection la crearemos mediante el Fov de la luz, Aspect ratio, near y far plane.

Nos guardaremos las matrices de ShadowProjection y de ShadowView según nuestras matrices de View y de Projection que serán utilizadas más tarde al renderizar la luz que genera la sombra.

Una vez calculadas las matrices nos disponemos a renderizar la escena estableciendo nuestra textura como RenderTarget. Renderizamos nuestra escena del Proc y desestablecemos nuestra textura como RenderTarget.

El código sería similar al siguiente.

```
void CDirectionalLight::SetShadowMap(CRenderManager *RM)
{
	COrthoFixedCameraController l_OrthoFixedCameraController(m_Position-
m_Direction, m_Position, m_OrthoShadowMapSize.x, m_OrthoShadowMapSize.y, 1.0f,
m_EndRangeAttenuation);

	CEffectManager &l_EffectManager=CORE->GetEffectManager();

	m_ViewShadowMap= l_Camera.GetViewMatrix();
	m_ProjectionShadowMap= l_Camera.GetProjectionMatrix();

	l_EffectManager.ActivateCamera(m_ViewShadowMap, m_ProjectionShadowMap,
l_Camera.GetPosition());
}

void CLight::GenerateShadowMap(CRenderManager *RM)
{
	SetShadowMap(RM);

	if(m_GenerateStaticShadowMap && m_MustUpdateStaticShadowMap)
	{
		m_StaticShadowMap->SetAsRenderTarget(0);
		RM->BeginRender();
		RM->Clear(true, true, true, 0xffffffff);
		for(size_t i=0;i<
m_StaticShadowMapRenderableObjectsManagers.size();++i)
			m_StaticShadowMapRenderableObjectsManagers[i]->Render(RM);
		m_MustUpdateStaticShadowMap=false;
		RM->EndRender();
		m_StaticShadowMap->UnsetAsRenderTarget(0);
	}
	if(m_DynamicShadowMapRenderableObjectsManagers.size()>0)
	{
		m_DynamicShadowMap->SetAsRenderTarget(0);
		RM->BeginRender();
		RM->Clear(true, true, true, 0xffffffff);
		for(size_t i=0;i<
m_DynamicShadowMapRenderableObjectsManagers.size();++i)
			m_DynamicShadowMapRenderableObjectsManagers[i]->Render(RM);
		RM->EndRender();
		m_DynamicShadowMap->UnsetAsRenderTarget(0);
	}
}
```

La technique que utilizaremos para generar el shadowmap de una malla estática será similar a la siguiente con su vertex shader y pixel shader.

```
//Vertex Shader
void VertShadow(float4 Pos : POSITION,
		float3 Normal : NORMAL,
		out float4 oPos : POSITION,
		out float2 Depth : TEXCOORD0 )
{
	//
	// Compute the projected coordinates
	//
	oPos = mul( Pos, g_WorldViewProj );
	//
	// Store z and w in our spare texcoord
```

```
	//
	Depth.xy = oPos.zw;
}

//Pixel Shader
void PixShadow( float2 Depth : TEXCOORD0, out float4 Color : COLOR )
{
	//
	// Depth is z / w
	//
	Color = Depth.x / Depth.y;
}
```

Una vez creada la textura de shadowmap renderizaremos la escena de forma "estándar". Añadiendo nuevos parámetros a nuestro renderizado, para comenzar estableceremos la textura del ShadowMap en la etapa que utilicemos como sampler de sombras. Por último estableceremos la matriz de conversión desde el sistema de coordenadas que convierte un vértice de View al LightProjection para ello utilizaremos un código como el siguiente.

```
g_ pd3dDevice ->SetTexture( IdStageShadowMap, g_pShadowMap );

if(m_UseViewToLightProjectionMatrix)
{
	Mat44f l_ViewToLightProjectionMatrix=l_ViewMatrix;
	l_ViewToLightProjectionMatrix.Inverse();
	l_ViewToLightProjectionMatrix=l_ViewToLightProjectionMatrix*l_EffectManager.GetLightVi
		ewMatrix();
	l_ViewToLightProjectionMatrix=l_ViewToLightProjectionMatrix*l_EffectManager.GetShado
		wProjectionMatrix();
	l_Effect->SetMatrix(m_Effect->m_ViewToLightProjectionMatrixParameter,
		&l_ViewToLightProjectionMatrix.GetD3DXMatrix());
}
//Renderiza la escena
```

Por último deberemos establecer las matrices de world, view y projection y renderizar la escena como hemos hecho hasta ahora introduciendo un nuevo cálculo que nos dirá si un píxel está bajo la influencia de la sombra o no. Para ello deberemos implementar el shader de la siguiente forma.

```
sampler ShadowMapTextureSampler : register( s5 ) = sampler_state
{
	MipFilter = LINEAR;
	MinFilter = LINEAR;
	MagFilter = LINEAR;
	AddressU = Clamp;
	AddressV = Clamp;
};

//Define global
#define SHADOW_EPSILON 0.00005f

//En el vertex shader deberemos introducir el siguiente código
float3 vPos = mul( float4(iPos.xyz,1.0), g_mWorldView ).xyz;
OUT.PosLight = mul( float4(vPos, 1.0), g_mViewToLightProj );

//En el pixel shader deberemos introducir el siguiente código
float2 ShadowTexC = 0.5 * IN.PosLight.xy / IN.PosLight.w + float2( 0.5, 0.5 );
ShadowTexC.y = 1.0f - ShadowTexC.y;
```

```
float LightAmount =  (tex2D( g_ ShadowMapTextureSampler, ShadowTexC ) +
SHADOW_EPSILON < IN.PosLight.z / IN.PosLight.w)? 0.0f: 1.0f;
// 1 – Píxel iluminado
// 0 – Píxel en sombra
```

Vignetting

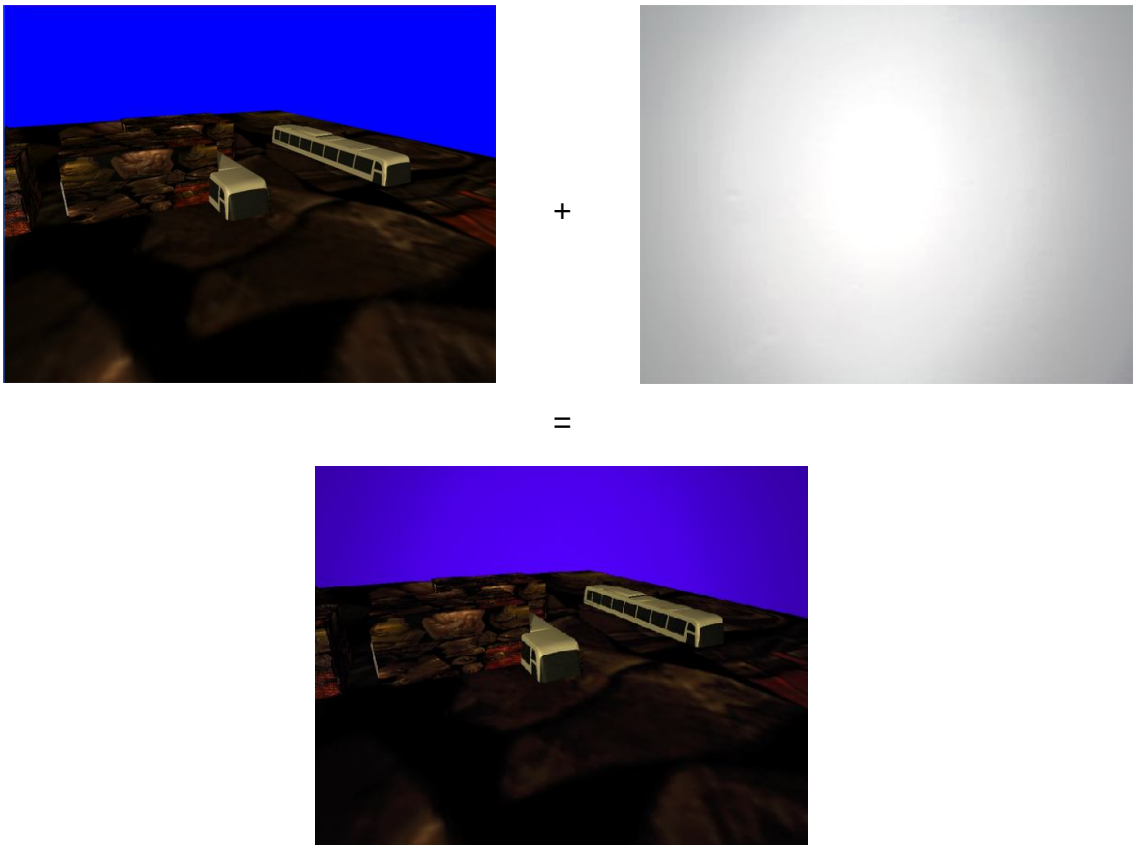
El siguiente efecto de escena que vamos a explicar consistirá en un efecto de post-procesado que debe pasar desapercibido, pero enfatizará la parte central de la pantalla que es dónde normalmente se centra la acción.

El efecto se basa simplemente en cambiar la saturación o el brillo de la imagen en las zonas externas de la imagen. Podéis apreciar mejor el efecto en las siguientes imágenes o en la web de la wikipedia.

http://en.wikipedia.org/wiki/Vignetting

Para implementar este efecto simplemente deberemos renderizar un plano encima de la imagen final del juego con rgb a negro y el canal alfa activado de forma que en el centro esté desactivado y se active un pequeño valor en las partes externas a través de la clase CDrawQuadSceneEffect.

Un ejemplo sería el siguiente.



Noise

Otro efecto similar al efecto de vignetting que nos dará un sensación de mayor realismo en la imagen, la idea es poner una pequeña textura de ruido sobre el renderizado final para dar una imagen menos sintética, de nuevo lo crearemos con un comando de renderizado de escena a nivel de post-procesado con la clase CDrawQuadRendererCommand.

+

=

Para crear este efecto deberemos crear una textura de ruido similar a la que vemos y multiplicarla por el renderizado final o utilizar el canal alpha y renderizar con alpha blend.

Un ejemplo de technique para renderizar el noise y el vignetting de una sóla pasada podría ser el siguiente.

```
float4 RenderNoiseAndVignettingPS(in float2 UV : TEXCOORD0) : COLOR
{
	float2 l_Offset=float2(cos(g_Time),sin(g_Time));
	float2 l_UV=UV+l_Offset;
	float4 l_VignettingColor=tex2D(NormalMapTextureSampler, UV);
	float4 l_NoiseColor=tex2D(DiffuseTextureSampler, l_UV);
	return float4(l_NoiseColor.xyz*l_VignettingColor.xyz, l_NoiseColor.a+l_VignettingColor.a);
}
```

```
technique RenderNoiseAndVignettingTechnique
{
	pass p0
	{
		CullMode = CCW;
		PixelShader = compile ps_3_0 RenderNoiseAndVignettingPS();
	}
}
```

Fog

El siguiente efecto que explicaremos será el de Fog, el efecto consiste en renderizar un quad a pantalla completa y dependiendo de la distancia del píxel respecto a la cámara y un color de niebla aplicaremos un porcentaje de ese color.

El efecto sería similar al siguiente, en la primera imagen vemos una ciudad sin niebla y en la segunda imagen la misma escena con un color de niebla aplicado según la distancia.

Para implementar dicho efecto podemos aplicar diferentes fórmulas para calcular la cantidad de color de niebla que le afecta al píxel. Las fórmulas pueden ser lineales, exponenciales o exponenciales al cuadrado. Podemos utilizar una technique como la siguiente, para más información mirar la documentación de DirectX.

```
float4 CalcLinearFog(float Depth, float StartFog, float EndFog, float4 FogColor)
{
	return float4(FogColor.xyz, FogColor.a*(1.0-CalcAttenuation(Depth, StartFog, EndFog)));
}
```

```
float4 CalcExp2Fog(float Depth, float ExpDensityFog, float4 FogColor)
{
	//Versión de directx
	float l_ExpDensity=Depth*ExpDensityFog;
	float l_Fog=1.0/exp(l_ExpDensity*l_ExpDensity);
	return float4(FogColor.xyz,FogColor.a*(1.0-l_Fog));

	//Versión que mejora el cálculo
	/*const float LOG2E = 1.442695; // = 1 / log(2)
	float l_Fog = exp2(-ExpDensityFog * ExpDensityFog * Depth * Depth * LOG2E);
	return float4(FogColor.xyz,FogColor.a*(1.0-l_Fog));*/
}

float4 CalcExpFog(float Depth, float ExpDensityFog, float4 FogColor)
{
	//Versión de directx
	/*float l_Fog=1.0/exp(Depth*ExpDensityFog);
	return float4(FogColor.xyz,FogColor.a*(1.0-l_Fog));*/

	//Versión que mejora el cálculo
	const float LOG2E = 1.442695; // = 1 / log(2)
	float l_Fog = exp2(-ExpDensityFog * Depth * LOG2E);
	return float4(FogColor.xyz,FogColor.a*(1.0-l_Fog));
}

float4 RenderQuadFogPS(in float2 UV : TEXCOORD0) : COLOR
{
	float4 l_DepthColor=tex2D(GUIZMapTextureSampler, UV);
	float3 l_ViewPosition=GetPositionFromZDepthViewInViewCoordinates(l_DepthColor, UV,
g_InverseProjectionMatrix);
	float l_Depth=length(l_ViewPosition);

	//return CalcExp2Fog(l_Depth, g_Exp2DensityFog, g_FogColor);
	return CalcExpFog(l_Depth, g_ExpDensityFog, g_FogColor);
	return CalcLinearFog(l_Depth, g_StartLinearFog, g_EndLinearFog, g_FogColor);
}

technique DrawQuadFogTechnique
{
	pass p0
	{
		AlphaBlendEnable = true;
		BlendOp=Add;
		SrcBlend=SrcAlpha;
		DestBlend=InvSrcAlpha;
		PixelShader=compile ps_3_0 RenderQuadFogPS();
	}
}
```

Deferred Shading

Para implementar el efecto de deferred shading nos basaremos en el tutorial que podemos encontrar en la siguiente url.

http://www710.univ-lyon1.fr/~jciehl/Public/educ/GAMA/2007/Deferred_Shading_Tutorial_SBGAMES2005.pdf

Hasta ahora hemos implementado la luz a nivel de forward lighting.

```
for each object do
        for each light do
                framebuffer = light_model(object,light);
```

En el deferred shading la implicación de la luz sobre los objetos se calcula en diferido. Para conseguir esto deberemos generar el denominado GBuffer.

```
for each object do
        G-buffer = lighting properties of object;
for each light do
        framebuffer += light_model(G-buffer,light);
```

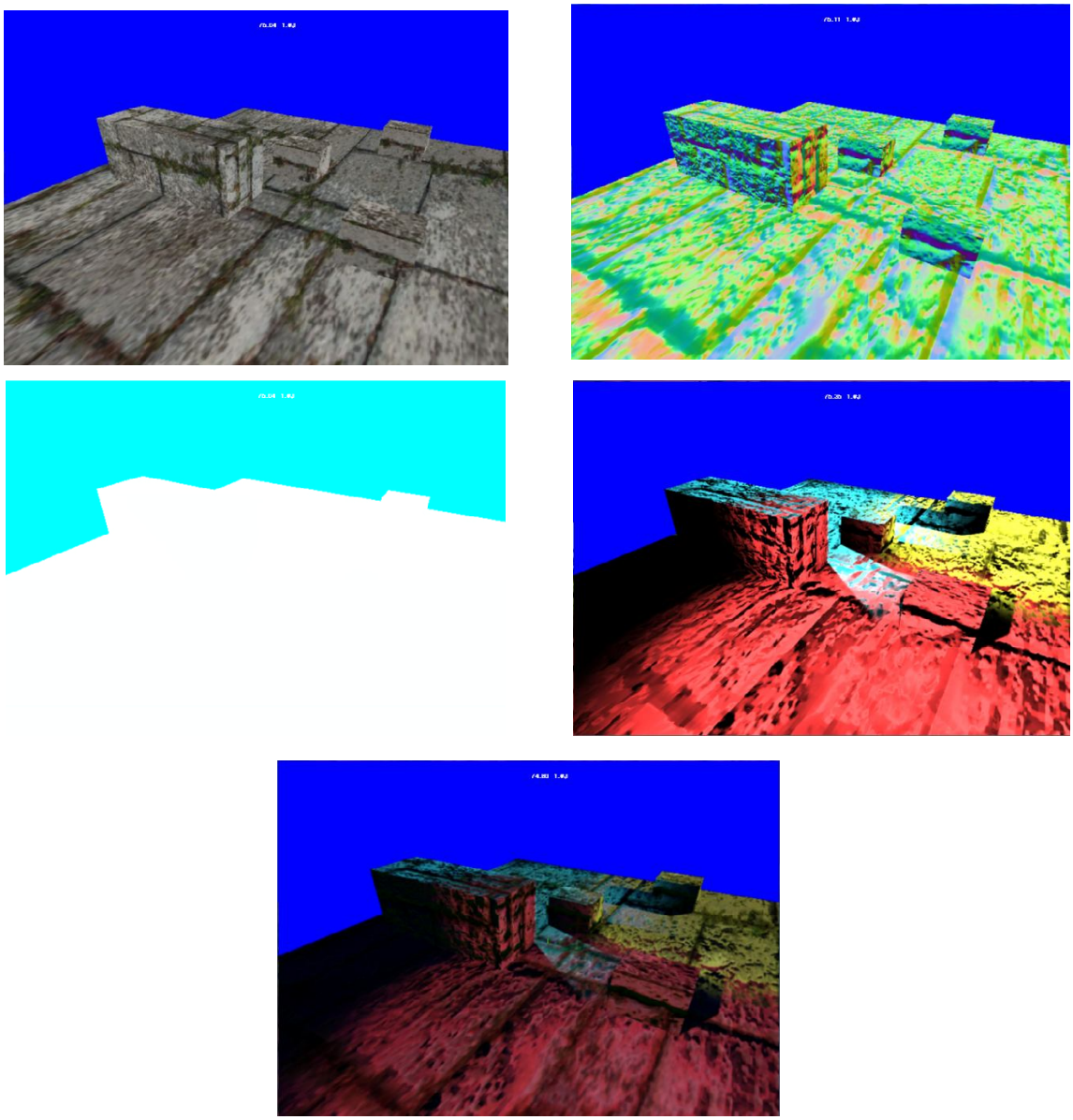
El GBuffer es una estructura de pixel que tiene la información de cuatro píxeles de salida. En cada píxel de color codificamos información. Un ejemplo podría ser el siguiente:

```
struct TMultiRenderTargetPixel
{
        float4 RT0 : COLOR0;   //Albedo (float3) + (float) SpecularFactor
        float4 RT1 : COLOR1;   //AmbientLight (float3) + (float) SpecularPow
        float4 RT2 : COLOR2;   //Normal (float3) + (float) Not used
        float4 RT3 : COLOR3;   //Depth (float4)
};
```

En esta estructura de vértice codificamos:

- COLOR0, guardamos la información de albedo rgb de la escena, en el canal alfa el SpecularFactor del material que estamos utilizando
- COLOR1, guardamos la información de la luz ambiente en formato rgb, podemos guardar la información del lightmap en caso de pertenece el píxel a una malla con lightmap o la luz ambiente general de la escena, en el canal alfa guardamos el SpecularPower del material del píxel
- COLOR2, guardamos la normal del píxel en formato entre 0 y 1 utilizando 3 canales, la normal la podemos guardar en 2 canales ya que la suma de sus componentes debe dar 1 y luego podemos recuperarlo realizando la inversa. El canal alfa de este píxel no lo estamos utilizando.
- COLOR3, guardamos la información de distancia del pixel respecto a la cámara view en formato float en un float4.

En imágenes podríamos conseguir resultados como los siguients de los diferentes colores.



De izquierda a derecha y de arriba abajo, mapas de albedo, normales, depths, ambientlight, complete Deferred.

Una vez generamos en una primera pasada el GBuffer de toda la geometría, pasamos a trabajar en 2D dónde aplicaremos la iluminación sobre la escena:

- primero renderizamos a pantalla completa dónde se generará luz ambiente de la escena
- pasamos después a renderizar todas las luces con el alpha blend activo dónde se calculará la luz difusa y especular que genera esta luz sobre la escena

Para convertir la información de la normal a un color válido deberemos escalar la normal a valores entre 0 y 1, valores válidos para un color, para ello utilizaremos un código como el siguiente.

```
float3 Normal2Texture(float3 Normal)
{
        return Normal*0.5+0.5;
```

```
}

float3 Texture2Normal(float3 Color)
{
	return (Color-0.5)*2;
}
```

Para convertir la información de la profundidad a la normal utilizaremos el mismo código que ya hemos utilizado al generar el shadowmap.

```
//En el Vertex shader calcularemos la posición en cordenadas de proyección
OUT.Pos=OUT.HPosition;
//En el Píxel shader calcularemos la z en formato color
float4 l_Depth=IN.Pos.z/IN.Pos.w
```

Para convertir la información de profundidad a coordenadas de mundo utilizaremos el siguiente código.

```
float3 GetPositionFromZDepthView(float ZDepthView, float2 UV, float4x4 InverseViewMatrix, float4x4 InverseProjectionMatrix)
{
	float3 l_PositionView=GetPositionFromZDepthViewInViewCoordinates(ZDepthView, UV, InverseProjectionMatrix);
	return mul(float4(l_PositionView,1.0), InverseViewMatrix).xyz;
}

float3 GetPositionFromZDepthViewInViewCoordinates(float ZDepthView, float2 UV, float4x4 InverseProjectionMatrix)
{
	// Get the depth value for this pixel
	// Get x/w and y/w from the viewport position
	float x = UV.x * 2 - 1;
	float y = (1 - UV.y) * 2 - 1;
	float4 l_ProjectedPos = float4(x, y, ZDepthView, 1.0);
	// Transform by the inverse projection matrix
	float4 l_PositionVS = mul(l_ProjectedPos, InverseProjection atrix);
	// Divide by w to get the view-space position
	return l_PositionVS.xyz / l_PositionVS.w;
}
```

Por último sólo necesitamos crear una technique que nos permita renderizar una luz como la siguiente.

```
technique RenderLightDeferredShadingTechnique
{
	pass p0
	{
		AlphaBlendEnable = true;
		BlendOp=Add;
		SrcBlend		= one;
		DestBlend		= one;
		PixelShader = compile ps_3_0 RenderLightDeferredShadingPS();
	}
}
```

Para renderizar las luces sobre la pantalla podemos realizarlo de dos formas:

- la primera forma es mediante un quad 2D a pantalla completa o limitándolo al tamaño en coordenadas de pantalla,

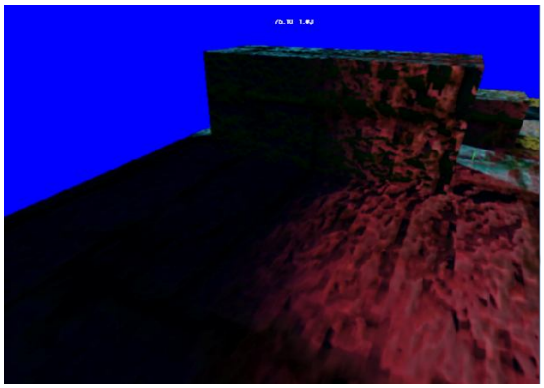
- la segunda forma es renderizando en 3D mallas con las formas de las luces, omni o spot, las direccionales siempre serán en 2D a pantalla completa

El mayor problema que podemos encontrar al renderizar una escena con Deferred Shading es el hecho de que no se pueden utilizar elementos transparentes ya que los píxeles transparentes manchan el zbuffer y no funcionaría. Una posible solución es renderizar los elementos con transparencia mediante forward lighting.

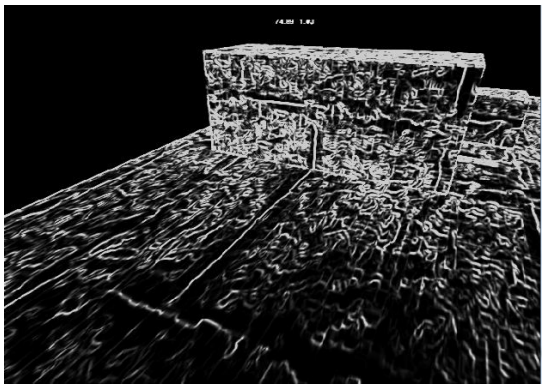
SSAA

El siguiente efecto que explicaremos será el de SSAA, Screen Space Anti Aliasing, mediante este efecto vamos a conseguir reducir el efecto de dientes de sierra que se generar al renderizar mallas con caras que hacen pendiente y baja resolución.

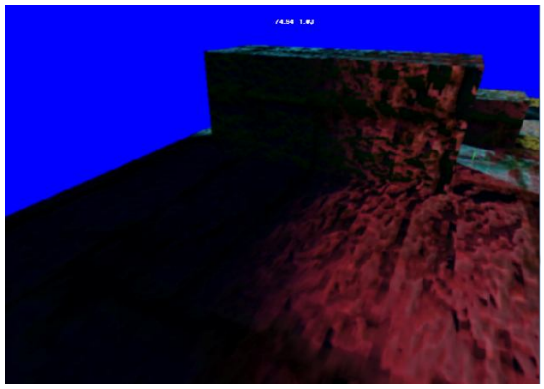
La forma de implementar este código es basándonos en un efecto de detección de filos a través de las normales y las profundidades de los píxeles. Una vez detectados los filos lo que realizamos es un emborronado con los píxeles cercanos.



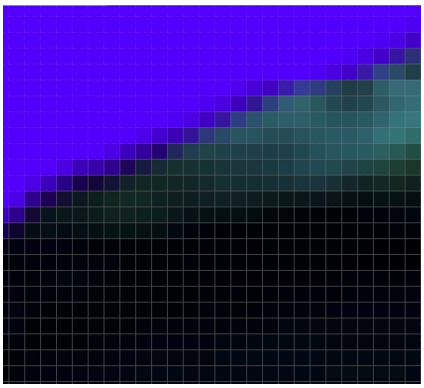
+



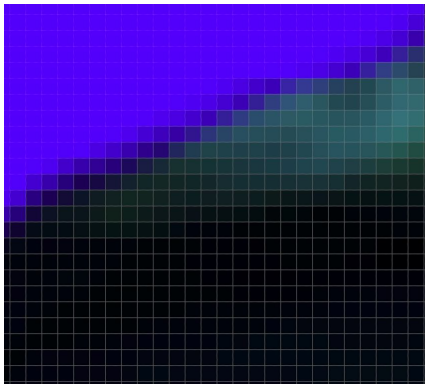
=



Visto desde una perspectiva más cercana y en detalle.



SSAA desactivado



SSAA activado

El código HLSL sería similar al siguiente.

```
float g_SSAAWeight=0.8;

float4 SSAAPS(in float2 UV : TEXCOORD0) : COLOR
{
	const float2 delta[8] =
	{
		float2(-1,1),float2(1,-1),float2(-1,1),float2(1,1),
		float2(-1,0),float2(1,0),float2(0,-1),float2(0,1)
	};

	float3 tex = Texture2Normal(tex2D(GUINormalMapTextureSampler,UV).xyz);
	float factor = 0.0f;

	for( int i=0;i<4;i++ )
	{
		float3 t = Texture2Normal(tex2D(GUINormalMapTextureSampler, UV+ delta[i] *
1/g_RenderTargetSize).xyz);
		t -= tex;
		factor += dot(t,t);
	}
	factor = min(1.0,factor)*g_SSAAWeight;

	//return float4(factor,factor,factor,1.0);
	float4 color = float4(0.0,0.0,0.0,0.0);

	float4 l_AlbedoColor=tex2D(GUIDiffuseMapTextureSampler,UV);

	for( int i=0;i<8;i++ )
		color += tex2D(GUIDiffuseMapTextureSampler,UV +
delta[i]*(1/g_RenderTargetSize))*factor+(1-factor)*l_AlbedoColor;

	color += 2.0*l_AlbedoColor;


	color = color*(1.0/10);
	return color;
}

technique SSAATechnique
{
	pass p0
	{
		AlphaBlendEnable = false;
		CullMode = CCW;
		PixelShader = compile ps_3_0 SSAAPS();
	}
}
```

Blur

El siguiente efecto que explicaremos será el de Blur, el efecto consiste en renderizar el último frame sobre el frame actual con un porcentaje de alfa dando la sensación de emborronado.

Para realizar este efecto deberemos primero capturar el frame búfer actual mediante la clase CCaptureFrameBufferSceneRendererCommand que nos

permite capturar el frame búfer para después renderizar este mismo frame sobre la escena con un alfa que hará que la imagen se vaya acumulando encima.

Z-Blur

El siguiente efecto que implementaremos será el mencionado Zblur o depth of field, dicho efecto genera el emborronado o desenfoque que vemos cuando apreciamos una escena a través de una cámara.

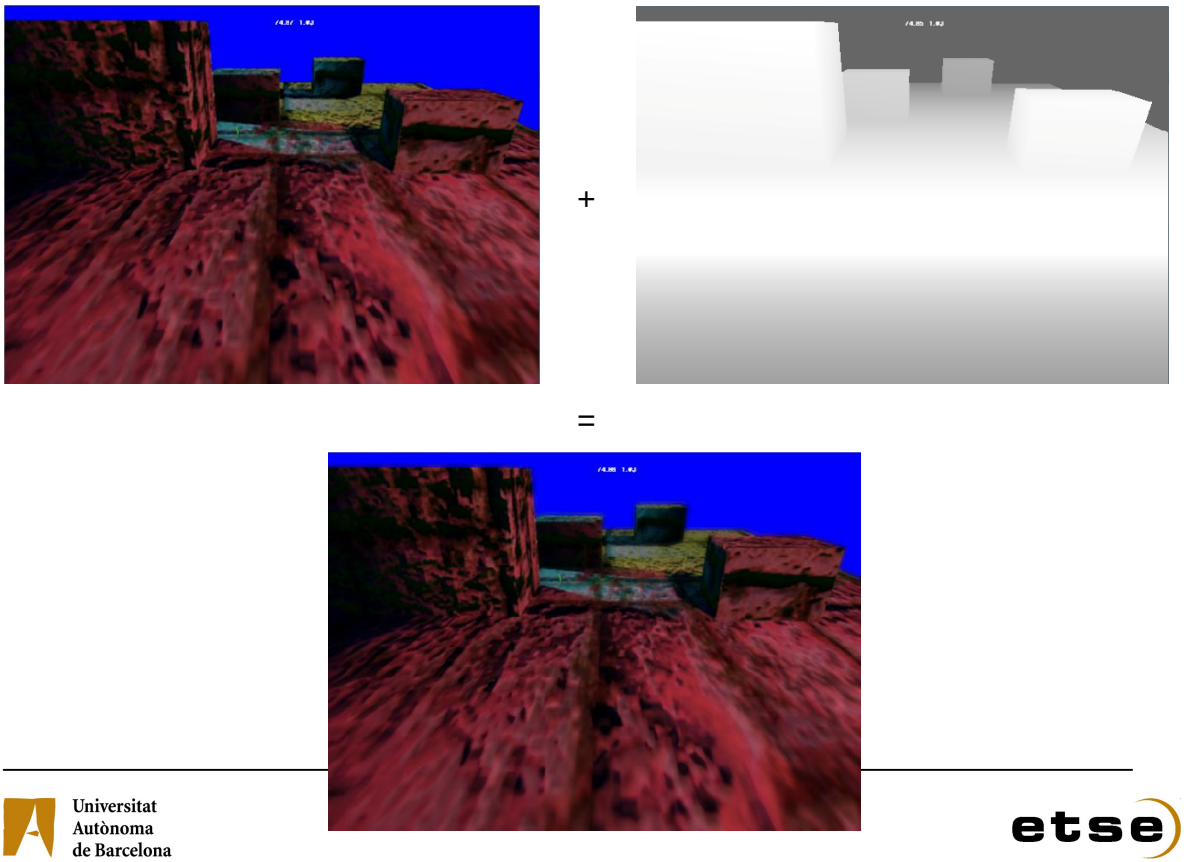
Para desarrollar dicho efecto nos vamos a basar en el código que hemos implementado al realizar el SSAA.

Para poder implementar este efecto deberemos tener diferentes constantes:

- la distancia mínima focal dónde los píxeles estarán comenzarán a estar enfocados,
- la distancia máxima focal dónde los píxeles comenzarán a dejar de estar enfocados,
- la distancia máxima dónde los vértices estarán totalmente desenfocados
- constante de desenfoque, nos dirá el mínimo valor de desenfoque de los píxeles desenfocados.

Una vez tenemos estos parámetros, calculamos la distancia de los píxeles a la cámara a través del mapa de distancias y calculamos el nivel de desenfoque de forma lineal.

Por último calculamos el color del píxel final según los píxeles cercanos y el valor de desenfoque. En imágenes sería algo similar a lo siguiente.



El código HLSL sería similar al siguiente.

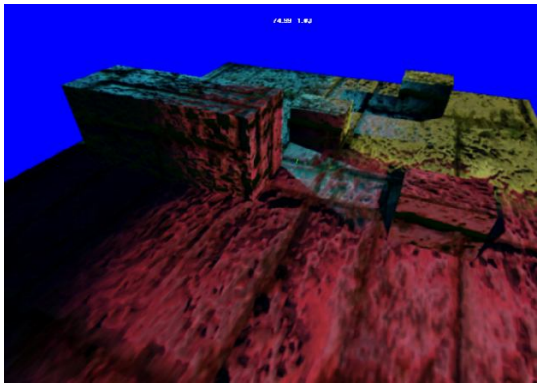
```
float g_ZBlurFocalStart=50;
float g_ZBlurFocalEnd=65;
float g_ZBlurEnd=300;
float g_ConstantBlur=0.4;

float4 ZBlurPS(in float2 UV : TEXCOORD0) : COLOR
{
	float4 l_DepthMap=tex2D(GUIZMapTextureSampler,UV);
	float3 l_CameraPosition=g_InverseViewMatrix[3].xyz;
	float3 l_WorldPosition=GetPositionFromZDepthView(l_DepthMap, float2(0,0),
g_InverseViewMatrix, g_InverseProjectionMatrix);
	float l_Distance=length(l_WorldPosition-l_CameraPosition);
	float4 l_Color=float4(0,0,0,0);
	float l_Blur=1.0;

	if(l_Distance<g_ZBlurFocalStart)
		l_Blur=max(l_Distance/g_ZBlurFocalStart, g_ConstantBlur);
	else if(l_Distance>g_ZBlurFocalEnd)
		l_Blur=max(1.0-(l_Distance-g_ZBlurFocalEnd)/g_ZBlurEnd, g_ConstantBlur);

	//return float4(l_Blur,l_Blur,l_Blur,1.0);

	const float2 delta[8] =
	{
		float2(-1,1),float2(1,-1),float2(-1,1),float2(1,1),
		float2(-1,0),float2(1,0),float2(0,-1),float2(0,1)
	};

	float2 l_PixelInc=4*1/g_RenderTargetSize; //4 pixeles a la redonda

	float4 l_AlbedoColor=tex2D(GUIDiffuseMapTextureSampler,UV);

	for( int i=0;i<8;i++ )
		l_Color += tex2D(GUIDiffuseMapTextureSampler,UV + delta[i]*l_PixelInc)*(1-
l_Blur)+l_Blur*l_AlbedoColor;

	l_Color = l_Color*(1.0/8.0);
	return l_Color;
}

technique ZBlurTechnique
{
	pass p0
	{
		AlphaBlendEnable = false;
		CullMode = CCW;
		PixelShader = compile ps_3_0 ZBlurPS();
	}
}
```

Color grading

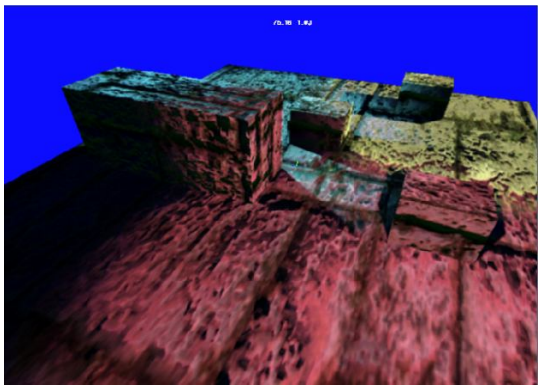
El siguiente efecto que explicaremos será el del Color grading, este efecto permite a los artistas graduar el color de la escena a nivel general mediante los niveles de corrección de los diferentes niveles de color.

Además permitirá controlar el nivel de contraste y brillo de la escena.

A continuación vemos diferentes imágenes con diferentes parámetros

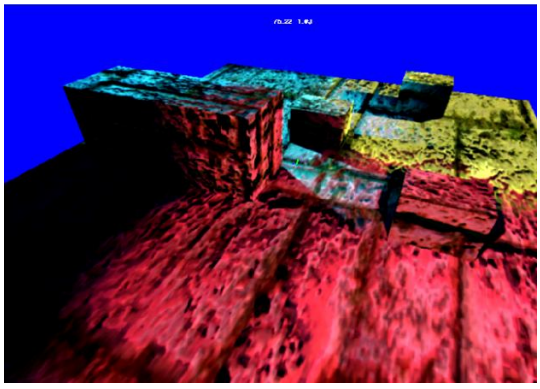


Color grading off



Color grading on

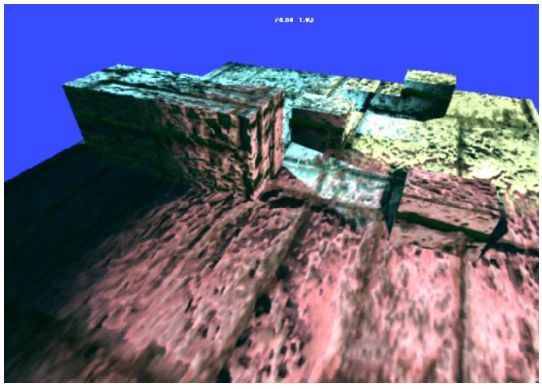
0.5 monochrome



Color grading on

Brightness -0.1

Contrast 2.0



Color grading on

0.5 brown color

0.2 green color

0.3 brown color

El código HLSL sería similar al siguiente.

```
float g_MonochromeColorGrading = 0.0;
float g_BrownSepiaColorGrading = 0.5;
float g_GreenSepiaColorGrading = 0.2;
float g_BlueSepiaColorGrading = 0.3;
float g_ContrastColorGrading = 1.0;
float g_BrightnessColorGrading = 0.0;
float g_ColorColorGrading = 1.0;

float4 ColorGrading(float4 _Color)
{
```

```
	//-------Color Matrices for Color Correction----------

	float4x4 gray = {0.299,0.587,0.184,0,
					 0.299,0.587,0.184,0,
					 0.299,0.587,0.184,0,
					 0,0,0,1};


	float4x4 sepia = {0.299,0.587,0.184,0.1,
					  0.299,0.587,0.184,0.018,
					  0.299,0.587,0.184,-0.090,
					  0,0,0,1};

	float4x4 sepia2 = {0.299,0.587,0.184,-0.090,
					   0.299,0.587,0.184,0.018,
					   0.299,0.587,0.184,0.1,
					   0,0,0,1};

	float4x4 sepia3 = {0.299,0.587,0.184,-0.090,
					   0.299,0.587,0.184,0.1,
					   0.299,0.587,0.184,0.1,
					   0,0,0,1};
	float4x4 sepia4 = {0.299,0.587,0.184,-0.090,
					   0.299,0.587,0.184,0.018,
					   0.1299,0.587,0.184,0.1,
					   0,0,0,1};

	float3 monochrome = (_Color.r * 0.3f + _Color.g * 0.59f + _Color.b * 0.11f);
	float4 monochrome4 = float4(monochrome,1);

	float4 result2 = _Color;

	float4 brownsepia = mul(sepia,result2) ;
	float4 greensepia = mul(sepia3,result2) ;
	float4 bluesepia = mul(sepia2,result2) ;

	float4 combine =  (brownsepia *g_BrownSepiaColorGrading ) + (greensepia
	*g_BrownSepiaColorGrading )+ (bluesepia * g_BlueSepiaColorGrading )+ (monochrome4
	* g_MonochromeColorGrading)+(g_ColorColorGrading * result2);
	return (combine  * g_ContrastColorGrading)  + g_BrightnessColorGrading;
}

float4 ColorGradingPS(in float2 UV : TEXCOORD0) : COLOR
{
	float4 l_Color= tex2D(GUIDiffuseMapTextureSampler, UV);
	return ColorGrading(l_Color);
}

technique ColorGradingTechnique
{
	pass p0
	{
		AlphaBlendEnable = false;
		CullMode = CCW;
		PixelShader = compile ps_3_0 ColorGradingPS();
	}
}
```

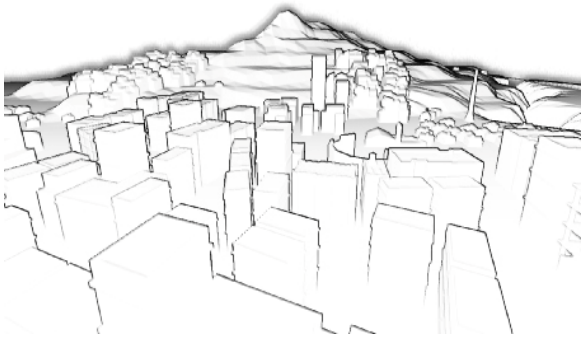
SSAO

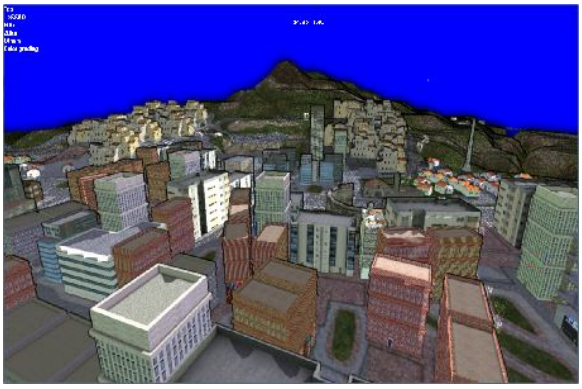
El siguiente efecto que explicaremos será el de SSAO (Screen Space Ambient Occlusion), este efecto consiste en introducir ambient occlusion en la escena a nivel de pantalla, el primer juego que lo introdujo fue Crysis.

Su implementación consiste en, utilizando un mapa de profundidades, recoger los píxeles de alrededor para calcular la cantidad de oclusión según la distancia de estos respecto al píxel de origen.

Para realizar de forma más correcta este proceso deberíamos recoger un area muy grande dentro de la pantalla, lo que realizamos es recoger sólo 16 píxeles de forma aleatoria dentro de un radio que le damos por constante al shader y devolvemos el valor intermedio.

A continuación vemos diferentes imágenes que demuestran el efecto.





El código HLSL para implementar el efecto sería similar al siguiente.

```
float g_SampleRadiusSSAO=0.023;
float g_DistanceScaleSSAO=0.405;

float4 SSAOPS(in float2 UV : TEXCOORD0) : COLOR
{
	float4 OUT=0;

	float4 samples[16] =
	{
		float4(0.355512,	-0.709318,	-0.102371,	0.0 ),
		float4(0.534186,	0.71511,	-0.115167,	0.0 ),
		float4(-0.87866,	0.157139,	-0.115167,	0.0 ),
		float4(0.140679,	-0.475516,	-0.0639818,	0.0 ),
		float4(-0.0796121,	0.158842,	-0.677075,	0.0 ),
		float4(-0.0759516,	-0.101676,	-0.483625,	0.0 ),
		float4(0.12493, -0.0223423,	-0.483625,	0.0 ),
		float4(-0.0720074,	0.243395,	-0.967251,	0.0 ),
		float4(-0.207641,	0.414286,	0.187755,	0.0 ),
		float4(-0.277332,	-0.371262,	0.187755,	0.0 ),
		float4(0.63864, -0.114214,	0.262857,	0.0 ),
		float4(-0.184051,	0.622119,	0.262857,	0.0 ),
		float4(0.110007,	-0.219486,	0.435574,	0.0 ),
		float4(0.235085,	0.314707,	0.696918,	0.0 ),
		float4(-0.290012,	0.0518654,	0.522688,	0.0 ),
```

```
                float4(0.0975089,       -0.329594,      0.609803,       0.0 )
        };

        float l_WidthScreenResolutionOffset=1/g_RenderTargetSize.x;
        float l_HeightScreenResolutionOffset=1/g_RenderTargetSize.y;

        float depth = tex2D(GUIZMapTextureSampler, UV);
        float3 se=GetPositionFromZDepthViewInViewCoordinates(depth, UV,
g_InverseProjectionMatrix);

        float4 vPositionVS = mul(float4(UV.x,UV.y,depth,1.0), g_InverseProjectionMatrix);
        depth=vPositionVS.z/vPositionVS.w;

        float3 randNormal = tex2D( S0LinearWrapSampler, UV * 200.0 ).rgb;

        float finalColor = 0.0f;

        for (int i = 0; i < 16; i++)
        {
                float3 ray = reflect(samples[i].xyz,randNormal) * g_SampleRadiusSSAO;

                float4 sample = float4(se + ray, 1.0f);

                float4 ss = mul(sample, g_ProjectionMatrix);

                float2 sampleTexCoord = 0.5f * ss.xy/ss.w + float2(0.5f, 0.5f);

                sampleTexCoord.x += l_WidthScreenResolutionOffset;
                sampleTexCoord.y += l_HeightScreenResolutionOffset;

                sampleTexCoord.y=1.0-sampleTexCoord.y;

                float sampleDepth = tex2D(GUIZMapTextureSampler, sampleTexCoord);

                vPositionVS = mul(float4(sampleTexCoord.x,sampleTexCoord.y,
sampleDepth,1.0), g_InverseProjectionMatrix);
                sampleDepth=vPositionVS.z/vPositionVS.w;

                if (sampleDepth == 1.0)
                {
                        finalColor ++;
                }
                else
                {
                        //float occlusion = g_DistanceScaleSSAO* max(sampleDepth - depth,
0.0f);
                        float occlusion = g_DistanceScaleSSAO* abs(sampleDepth - depth);
                        finalColor += 1.0f / (1.0f + occlusion * occlusion * 0.1);
                }
        }
        return float4(finalColor/16, finalColor/16, finalColor/16, 1.0f);
}

float4 SSAOFinalCompositionPS(in float2 UV : TEXCOORD0) : COLOR
{
        //return float4(tex2D(GUILightMapTextureSampler,UV).xyz,1.0);
        return float4(tex2D(GUIDiffuseMapTextureSampler,UV).xyz *tex2D(
GUILightMapTextureSampler,UV).r,0.0);
}


//Technique que genera la textura de SSAO
technique SSAOTechnique
```

```
{
	pass p0
	{
		AlphaBlendEnable = false;
		CullMode = CCW;
		PixelShader = compile ps_3_0 SSAOPS();
	}
}

//Technique que renderiza el SSAO con la imagen completa
technique SSAOFinalCompositionTechnique
{
	pass p0
	{
		AlphaBlendEnable = false;
		CullMode = CCW;
		PixelShader = compile ps_3_0 SSAOFinalCompositionPS();
	}
}
```

Por último a este efecto se le puede añadir un efecto de emborronado antes de renderizar la última pasada utilizando un código HLSL como el siguiente.

```
float4 SSAOBlurPS(in float2 UV : TEXCOORD0) : COLOR
{
	float4 OUT=0;

	float l_RTWidth=g_RenderTargetSize.x;
	float l_RTHeight=g_RenderTargetSize.y;
	float2 blurDirection=float2(0.0, 1.0/l_RTHeight); //Vector Up screen

	UV.x += 1.0/l_RTWidth;
	UV.y += 1.0/l_RTHeight;

	float3 normal = tex2D(GUINormalMapTextureSampler, UV).rgb;
	normal=normalize(Texture2Normal(normal));

	float color = tex2D( GUIDiffuseMapTextureSampler, UV).r;

	float num = 1;
	int blurSamples = 8;

	for( int i = -blurSamples/2; i <= blurSamples/2; i+=1)
	{
		float4 newTexCoord = float4(UV + i * blurDirection.xy, 0, 0);

		float sample = tex2D(GUIDiffuseMapTextureSampler, newTexCoord).r;
		float3 samplenormal = tex2D(GUINormalMapTextureSampler, newTexCoord).rgb;
		samplenormal=normalize(Texture2Normal(samplenormal));

		if (dot(samplenormal, normal) > 0.99 )
		{
			num += (blurSamples/2 - abs(i));
			color += sample * (blurSamples/2 - abs(i));
		}
	}

	return color / num;
}


technique SSAOBlurTechnique
{
```

```
        pass p0
        {
                AlphaBlendEnable = false;
                CullMode = CCW;
                PixelShader = compile ps_3_0 SSAOBlurPS();
        }
}
```

RNM

El siguiente efecto que explicaremos será el de Radiosity Normal Mapping, esta técnica la utiliza el motor de valve Source, la idea consiste en introducir iluminación mediante lightmap que se vea afectado por el normal map.

Para conseguir este efecto vamos a necesitar generar tres lightmaps con una información específica en el canal de bump del 3DStudio MAX. Para conseguirlo podemos generar una malla que contenga un lightmap como realizamos normalmente, una vez conseguido el lightmap sin normalmap que queremos conseguir, deberemos realizar los siguientes pasos:

- Crear un material de NormalMap en el canal de bump de todos los materiales de nuestra malla
- En el material aplicamos en Method World
- En normal metemos un material de tipo checker y en los colores aplicamos los siguientes colores para generar:
    - El lightmap en el eje x, color RGB (231, 127, 201)
    - El lightmap en el eje y, color RGB (75, 217, 201)
    - El lightmap en el eje z, color RGB (75, 37, 201)

Una vez tenemos los 3 lightmaps deberemos aplicarlos en la malla en el 3DStudio MAX, para ello podemos utilizar el canal de selfIllumination del material y aplicarle un material de tipo MultiMap/Sub-Map y establecer los tres lightmaps en los tres primeros mapas, si no apareciese el material MultiMap/Sub-Map pulsar sobre el botón incompatible y buscarlo dentro de los elementos en gris.

Deberemos modificar nuestros scripts de MAXScript para exportar la nueva malla con la información de Radiosity Normal Mapping.

Una vez exportada la malla deberemos importarla en nuestro motor y generar una nueva technique dónde crearemos un pixel shader dónde calcularemos la iluminación del píxel a través de los tres lightmaps. Podemos utilizar un código similar al siguiente.

```
float3 GetRadiosityNormalMap(float3 Nn, float2 UV, float3x3 WorldMatrix)
{
        float3 l_LightmapX=tex2D(S1LinearWrapSampler, UV)*2;
        float3 l_LightmapY=tex2D(S2LinearWrapSampler, UV)*2;
        float3 l_LightmapZ=tex2D(S3LinearWrapSampler, UV)*2;

        float3 l_BumpBasisX=normalize(float3(0.816496580927726, 0.5773502691896258, 0 ));
        float3 l_BumpBasisY=normalize(float3(-0.408248290463863, 0.5773502691896258,
0.7071067811865475 ));
        float3 l_BumpBasisZ=normalize(float3(-0.408248290463863, 0.5773502691896258, -
0.7071067811865475));
```

```
        float3 diffuseLighting=saturate( dot( Nn, l_BumpBasisX ) ) * l_LightmapX +
                               saturate( dot( Nn, l_BumpBasisY ) ) * l_LightmapY +
                               saturate( dot( Nn, l_BumpBasisZ ) ) * l_LightmapZ;

        return diffuseLighting;
}

//En el pixel shader calculamos el normalmap del pixel
float4 l_NormalMap=tex2D(S5LinearWrapSampler,IN.UV);
float3 Tn=normalize(IN.WorldTangent);
float3 Bn=normalize(IN.WorldBinormal);
float3 Nn=normalize(IN.WorldNormal);

Nn=CalcNormalMap(Nn, Tn, Bn, l_NormalMap);
//Por último calculamos el color del pixel del lightmap según los valores
float3 l_Lightmap=GetRadiosityNormalMap(Nn, IN.UV2, g_WorldMatrix);
```

En esta primera parte generamos la luz difusa a través del lightmap, además podemos añadir una capa con la información specular que podríamos calcular a través de cubemaps dónde quemamos la información de specular.

Podéis encontrar más información en la siguiente web.

http://www2.ati.com/developer/gdc/D3DTutorial10_Half-Life2_Shading.pdf

A continuación podemos ver un ejemplo demostrativo.

HDR

El siguiente efecto que explicaremos será el de HDR, para ello nos basaremos en el ejemplo que acompaña a la documentación de DirectX HDR Lighting.

El ejemplo se base en la adaptación del ojo humano a los cambios de iluminación en las escenas, por ejemplo al pasar de zonas oscuras a zonas muy iluminadas o lo contrario. Los colores RGBA hemos visto hasta ahora que tenían valores comprendidos entre 0 y 1, para poder representar la iluminación completa de una escena necesitamos valores que contengan la intensidad de iluminación con valores comprendidos superiores a estos valores, para más tarde volver a renderizar en valores entre 0 y 1.

Para implementar este efecto necesitaremos renderizar nuestra escena en una textura en formato A16R16G16B16 que guarde la información del color con valores por encima de unos.

Una vez tenemos la textura generada de la escena en 64 bits, podemos rescalarla a un tamaño menor para trabajar con menos píxeles en las siguientes pasadas.

A continuación debemos calcular la media de iluminación que tenemos en la escena, para ello vamos reescalando la imagen hasta llegar a una imagen de 1x1 que contendrá el color con la media de iluminación de la escena en el frame dado.

Con el valor medio de iluminación de la escena podemos calcular una imagen con las escenas brillantes que destacan dentro de la escena. Dentro de la documentación de DirectX es lo que denomina la Bright-pass filtered, a partir de la bright-pass filter podemos crear la textura de bloom y la textura de star effect.

La textura de bloom nos dará la sobreexposición de los píxeles que están más iluminados que la media de la escena.

La textura de star es un efecto que añade para dar los efectos que genera al mirar a través de una cámara a las luces, encontramos diferentes ejemplos en la documentación.

Por último realizamos una última pasada dónde renderizamos con todas las texturas generadas previamente para conseguir el resultado deseado.

Para generar este efecto en comandos de escena podemos utilizar un xml de comandos de renderizado de escena similar al siguiente.

```
<!--HDR-->
<set_render_target name="SceneHDR64BitsRenderTarget">
	<dynamic_texture stage_id="0" name="SceneHDR64BitsTexture"
texture_width_as_frame_buffer="true" format_type="A16B16G16R16F"/>
</set_render_target>

<set_renderable_objects_technique pool="GenerateHDRDeferredShadingPool"/>
<render_draw_quad>
	<texture stage_id="0" file="FinalDeferredShadingAmbientTexture"/>
	<texture stage_id="1" file="FinalDeferredShadingLightingTexture"/>
</render_draw_quad>

<unset_render_target render_target="SceneHDR64BitsRenderTarget"/>

<set_render_target name="SceneHDR64512x512BitsRenderTarget">
	<dynamic_texture stage_id="0" name="SceneHDR64BitsTexture512x512" width="512"
height="512" format_type="A16B16G16R16F"/>
</set_render_target>

<set_renderable_objects_technique
pool="draw_quad_opaque_pool_renderable_object_technique"/>
<render_draw_quad active="true">
	<texture stage_id="0" file="SceneHDR64BitsTexture"/>
</render_draw_quad>

<unset_render_target render_target="SceneHDR64512x512BitsRenderTarget"/>

<!--crear tonemap-->
<set_render_target name="Tonemap0HDRBitsRenderTarget">
	<dynamic_texture stage_id="0" name="Tonemap0HDRTexture" width="64" height="64"
format_type="R16F"/>
```

```
</set_render_target>
<set_scissor_rect left="1" top="1" right="63" bottom="63"/>

<set_renderable_objects_technique pool="GenerateTonemapInitialHDRPool"/>
<clear_scene color="true" depth="false" stencil="false"/>
<render_draw_quad active="true">
        <texture stage_id="0" file="SceneHDR64BitsTexture512x512"/>
</render_draw_quad>
<unset_scissor_rect/>
<unset_render_target render_target="Tonemap0HDRBitsRenderTarget"/>
<set_render_target name="Tonemap1HDRBitsRenderTarget">
        <dynamic_texture stage_id="0" name="Tonemap1HDRTexture" width="16" height="16"
format_type="R16F"/>
</set_render_target>
<set_scissor_rect left="1" top="1" right="15" bottom="15"/>
<set_renderable_objects_technique pool="GenerateTonemapIterativeHDRPool0"/>
<clear_scene color="true" depth="false" stencil="false"/>
<render_draw_quad active="true">
        <texture stage_id="0" file="Tonemap0HDRTexture"/>
</render_draw_quad>
<unset_scissor_rect/>
<unset_render_target render_target="Tonemap1HDRBitsRenderTarget"/>

<set_render_target name="Tonemap2HDRBitsRenderTarget">
        <dynamic_texture stage_id="0" name="Tonemap2HDRTexture" width="4" height="4"
format_type="R16F"/>
</set_render_target>

<set_scissor_rect left="1" top="1" right="3" bottom="3"/>
<set_renderable_objects_technique pool="GenerateTonemapIterativeHDRPool1"/>
<clear_scene color="true" depth="false" stencil="false"/>
<render_draw_quad active="true">
        <texture stage_id="0" file="Tonemap1HDRTexture"/>
</render_draw_quad>
<unset_scissor_rect/>
<unset_render_target render_target="Tonemap2HDRBitsRenderTarget"/>


<set_render_target name="Tonemap3HDRBitsRenderTarget">
        <dynamic_texture stage_id="0" name="Tonemap3HDRTexture" width="1" height="1"
format_type="R16F"/>
</set_render_target>

<set_renderable_objects_technique pool="GenerateTonemapFinalHDRPool"/>
<render_draw_quad active="true">
        <texture stage_id="0" file="Tonemap2HDRTexture"/>
</render_draw_quad>
<unset_render_target render_target="Tonemap3HDRBitsRenderTarget"/>

<set_render_target name="DummyLastLuminance">
        <dynamic_texture stage_id="0" name="LastLuminanceHDRTexture" width="1" height="1"
format_type="R16F"/>
</set_render_target>
<unset_render_target render_target="DummyLastLuminance"/>

<!--CREAR ADAPTACION-->
<set_render_target name="GenerateLuminanceHDRRenderTarget">
        <dynamic_texture stage_id="0" name="CurrentLuminanceHDRTexture" width="1"
height="1" format_type="R16F"/>
</set_render_target>
<set_renderable_objects_technique pool="GenerateLuminanceHDRPool"/>
<render_draw_quad active="true">
        <texture stage_id="0" file="LastLuminanceHDRTexture"/>
```

```
        <texture stage_id="1" file="Tonemap3HDRTexture"/>
</render_draw_quad>
<unset_render_target render_target="GenerateLuminanceHDRRenderTarget"/>

<set_render_target name="DummyLastLuminance2">
        <dynamic_texture stage_id="0" name="LastLuminanceHDRTexture" width="1" height="1"
format_type="R16F"/>
</set_render_target>
<set_renderable_objects_technique
pool="draw_quad_opaque_pool_renderable_object_technique"/>
<render_draw_quad active="true">
        <texture stage_id="0" file="CurrentLuminanceHDRTexture"/>
</render_draw_quad>
<unset_render_target render_target="DummyLastLuminance2"/>

<!--CREAR BRIGHTNESS-->
<set_render_target name="GenerateBrightnessHDRRenderTarget">
        <dynamic_texture stage_id="0" name="BrightnessHDRTexture" width="512" height="256"
format_type="A8R8G8B8"/>
</set_render_target>
<set_scissor_rect left="1" top="1" right="511" bottom="255"/>
<set_renderable_objects_technique pool="GenerateBrightnessHDRPool"/>
<render_draw_quad active="true">
        <texture stage_id="0" file="SceneHDR64BitsTexture512x512"/>
        <texture stage_id="1" file="CurrentLuminanceHDRTexture"/>
</render_draw_quad>
<unset_scissor_rect/>
<unset_render_target render_target="GenerateBrightnessHDRRenderTarget"/>

<!--CREAR STAR-->
<set_render_target name="GenerateSourceStarHDRRenderTarget">
        <dynamic_texture stage_id="0" name="StarSourceHDRTexture" width="512" height="256"
format_type="A8R8G8B8"/>
</set_render_target>
<set_scissor_rect left="1" top="1" right="511" bottom="255"/>
<set_renderable_objects_technique pool="GenerateStarHDRPool"/>
<render_draw_quad active="true">
        <texture stage_id="0" file="BrightnessHDRTexture"/>
</render_draw_quad>
<unset_scissor_rect/>
<unset_render_target render_target="GenerateSourceStarHDRRenderTarget"/>

<!--CREAR BLOOM SOURCE-->
<set_render_target name="GenerateBloomSourceHDRRenderTarget">
        <dynamic_texture stage_id="0" name="BloomSourceHDRTexture" width="512"
height="256" format_type="A8R8G8B8"/>
</set_render_target>
<set_scissor_rect left="1" top="1" right="511" bottom="255"/>
<set_renderable_objects_technique pool="GenerateBloomSourceHDRPool"/>
<render_draw_quad active="true">
        <texture stage_id="0" file="StarSourceHDRTexture"/>
</render_draw_quad>
<unset_scissor_rect/>
<unset_render_target render_target="GenerateBloomSourceHDRRenderTarget"/>

<!--CREAR BLOOM-->
<set_render_target name="GenerateBloom0HDRRenderTarget">
        <dynamic_texture stage_id="0" name="Bloom0HDRTexture" width="256" height="128"
format_type="A8R8G8B8"/>
</set_render_target>
<set_scissor_rect left="1" top="1" right="255" bottom="127"/>
<set_renderable_objects_technique pool="GenerateBloom0HDRPool"/>
<clear_scene color="true" depth="false" stencil="false"/>
```

```
<render_draw_quad active="true">
        <texture stage_id="0" file="BloomSourceHDRTexture"/>
</render_draw_quad>
<unset_scissor_rect/>
<unset_render_target render_target="GenerateBloom0HDRRenderTarget"/>

<set_render_target name="GenerateBloom1HDRRenderTarget">
        <dynamic_texture stage_id="0" name="Bloom1HDRTexture" width="256" height="128"
format_type="A8R8G8B8"/>
</set_render_target>
<set_scissor_rect left="1" top="1" right="255" bottom="127"/>
<set_renderable_objects_technique pool="GenerateBloom1HDRPool"/>
<render_draw_quad active="true">
        <texture stage_id="0" file="Bloom0HDRTexture"/>
</render_draw_quad>
<unset_scissor_rect/>
<unset_render_target render_target="GenerateBloom1HDRRenderTarget"/>

<set_render_target name="GenerateBloom2HDRRenderTarget">
        <dynamic_texture stage_id="0" name="Bloom2HDRTexture" width="256" height="128"
format_type="A8R8G8B8"/>
</set_render_target>
<set_scissor_rect left="1" top="1" right="255" bottom="127"/>
<set_renderable_objects_technique pool="GenerateBloom2HDRPool"/>
<render_draw_quad active="true">
        <texture stage_id="0" file="Bloom1HDRTexture"/>
</render_draw_quad>
<unset_scissor_rect/>
<unset_render_target render_target="GenerateBloom2HDRRenderTarget"/>

<set_render_target name="GenerateBloomHDRRenderTarget">
        <dynamic_texture stage_id="0" name="BloomHDRTexture" width="256" height="128"
format_type="A8R8G8B8"/>
</set_render_target>
<set_scissor_rect left="1" top="1" right="255" bottom="127"/>
<set_renderable_objects_technique pool="GenerateBloom3HDRPool"/>
<render_draw_quad active="true">
        <texture stage_id="0" file="Bloom2HDRTexture"/>
</render_draw_quad>
<unset_scissor_rect/>
<unset_render_target render_target="GenerateBloomHDRRenderTarget"/>

<!--CREATE STAR NATURAL BLOOM -->
<set_render_target name="GenerateStarHDRRenderTarget">
        <dynamic_texture stage_id="0" name="StarHDRTexture" width="256" height="128"
format_type="A8R8G8B8"/>
</set_render_target>
<clear_scene color="true" depth="false" stencil="false"/>
<unset_render_target render_target="GenerateStarHDRRenderTarget"/>

<!--RENDER FINAL HDR-->
<set_renderable_objects_technique pool="GenerateFinalPassHDRPool"/>
<render_draw_quad active="true">
        <texture stage_id="0" file="SceneHDR64BitsTexture"/>
        <texture stage_id="1" file="BloomHDRTexture"/>
        <texture stage_id="2" file="StarHDRTexture"/>
        <texture stage_id="3" file="CurrentLuminanceHDRTexture"/>
</render_draw_quad>
```

HDR en imágenes