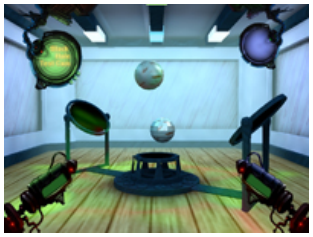
FixedFuncEMU Sample

Collapse All

This sample attempts to emulate certain aspects of the Direct3D 9 fixed function pipeline in a Direct3D 10 environment.



Path

Source	*SDK root*\Samples\C++\Direct3D10\FixedFuncEMU
Executable	*SDK root*\Samples\C++\Direct3D10\Bin*x86 or x64*\FixedFuncEMU.exe

How the Sample Works

This sample attempts to emulate the following aspects of the Direct3D 9 fixed-function pipeline:

- Fixed-function Transformation Pipeline
- Fixed-function Lighting Pipeline
- AlphaTest
- User Clip Planes
- Pixel Fog
- Gouraud and Flat shade modes
- Projected texture lookups (texldp)
- Multi-Texturing
- D3DFILL_POINT fillmode
- Screen space UI rendering

Fixed-function Transformation Pipeline

The Direct3D 9 fixed-function transformation pipeline required 3 matrices to transform the raw points in 3d space into their 2d screen representations. These were the World, View, and Projection matrices. Instead of using SetTransform(D3DTS_WORLD, someMatrix), we pass the matrices in as effect variables. The shader multiplies the input vertex positions by each of the World, View, and Projection matrices to get the same transformation that would have been produced by the fixed-function pipeline.

```
//output our final position in clipspace
float4 worldPos = mul( float4( input.pos, 1 ), g_mWorld );
float4 cameraPos = mul( worldPos, g_mView ); //Save cameraPos for fog calculations
output.pos = mul( cameraPos, g_mProj );
```

Fixed-function Lighting Pipeline

```
ColorsOutput CalcLighting( float3 worldNormal, float3 worldPos, float3 cameraPos )
{
    ColorsOutput output = (ColorsOutput)0.0;

    for(int i=0; i<8; i++)
    {
        float3 toLight = g_lights[i].Position.xyz - worldPos;
        float lightDist = length( toLight );
        float fAtten = 1.0/dot( g_lights[i].Atten, float4(1,lightDist,lightDist*lightDist,0) );
        float3 lightDir = normalize( toLight );
        float3 halfAngle = normalize( normalize(-cameraPos) + lightDir );

        output.Diffuse += max(0,dot( lightDir, worldNormal ) * g_lights[i].Diffuse * fAtten) + g_lights[i].Ambient;
        output.Specular += max(0,pow( dot( halfAngle, worldNormal ), 64 ) * g_lights[i].Specular * fAtten );
    }

    return output;
}
```

AlphaTest

Alpha test is perhaps the simplest Direct3D 9 functionality to emulate. It does not require the user to set a alpha blend state. Instead the user can simply choose to discard a pixel based upon its alpha value. The following pixel shader does not draw the pixel if the alpha value is less than 0.5.

```
//
// PS for rendering with alpha test
//
```

```
float4 PSAlphaTestmain(PSSceneIn input) : COLOR0
{
        float4 color =  tex2D( g_samLinear, g_txDiffuse, input.tex ) * input.colorD;
        if( color.a < 0.5 )
                discard;
        return color;
}
```

User Clip Planes

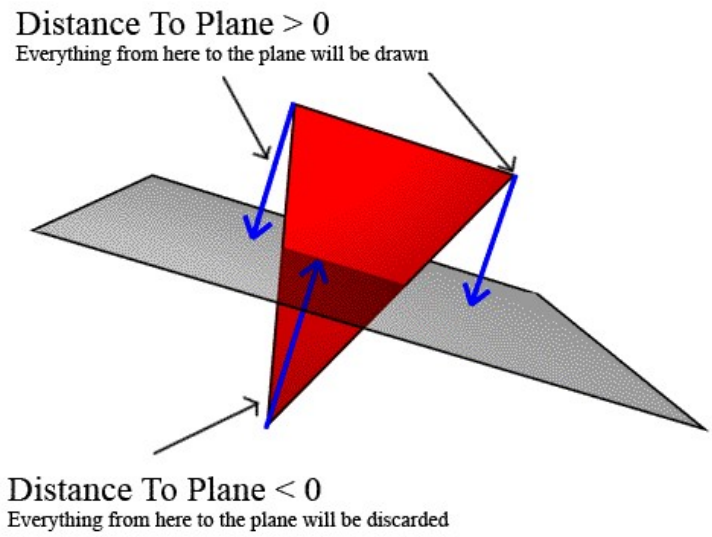
User Clip Planes are emulated by specifying a clip distance output from the Vertex Shader with the SV_ClipDistance[n] flag, where n is either 0 or 1. Each component can hold up to 4 clip distances in x, y, z, and w giving a total of 8 clip distances.

In this scenario, each clip planes is defined by a plane equation of the form:

$$Ax + By + Cz + D = 0$$

Where <A,B,C> is the normal of the plane, and D is the distance of the plane from the origin. Plugging in any point <x,y,z> into this equation gives its distance from the plane. Therefore, all points <x,y,z> that satisfy the equation Ax + By + Cz + D = 0 are on the plane. All points that satisfy Ax + By + Cz + D < 0 are below the plane. All points that satisfy Ax + By + Cz + D > 0 are above the plane.

In the Vertex Shader, each vertex is tested against each plane equation to produce a distance to the clip plane. Each of the three clip distances are stored in the first three components of the output component with the semantic SV_ClipDistance0. These clip distances get interpolated over the triangle during rasterization and clipped if the value every goes below 0.



Pixel Fog

Pixel fog uses a fog factor to determine how much a pixel is obscured by fog. In order to accurately calculate the fog factor, we must have the distance from the eye to the pixel being rendered. In Direct3D 9, this was approximated by using the Z-coordinate of a point that has been transformed by both the World and View matrices. In the vertex shader, this distance is stored in the fogDist member of the PSSceneIn struct for all 3 vertices of a triangle. It is then interpolated across the triangle during rasterization and passed to the pixel shader.

The pixel shader takes this fogDist value and passes it to the CalcFogFactor function which calculates the fog factor based upon the current value of g_fogMode.

```
//
// Calculates fog factor based upon distance
//
// E is defined as the base of the natural logarithm (2.71828)
float CalcFogFactor( float d )
{
        float fogCoeff = 1.0;

        if( FOGMODE_LINEAR == g_fogMode )
        {
                fogCoeff = (g_fogEnd - d)/(g_fogEnd - g_fogStart);
        }
        else if( FOGMODE_EXP == g_fogMode )
        {
                fogCoeff = 1.0 / pow( E, d*g_fogDensity );
        }
        else if( FOGMODE_EXP2 == g_fogMode )
        {
                fogCoeff = 1.0 / pow( E, d*d*g_fogDensity*g_fogDensity );
        }

        return clamp( fogCoeff, 0, 1 );
}
```

Finally, the pixel shader uses the fog factor to determine how much of the original color and how much of the fog color to output to the pixel.

```
return fog * normalColor + (1.0 - fog)*g_fogColor;
```

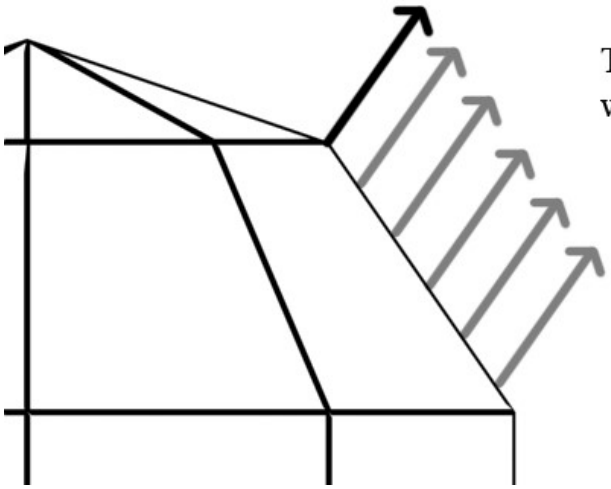
Gouraud and Flat shade modes

Gouraud shading involves calculating the color at the vertex of each triangle and interpolating it over the face of the triangle. By default Direct3D 10 uses D3D10_INTERPOLATION_MODE D3D10_INTERPOLATION_LINEAR to interpolate values over the face of a triangle during rasterization. Because of this, we can emulate Gouraud shading by calculating the lighting using Lambertian lighting (*dot(Normal, LightDir)*) at each vertex and letting Direct3D 10 interpolate these values for us. By the time the color gets to the pixel shader, we simply use it as our lighting value and pass it through. No further work is needed.

Direct3D 10 also provides another way to interpolate data across a triangle, D3D10_INTERPOLATION_CONST. A naive approach would be to use this to calculate the color at the first vertex and allow that color to be spread across the entire face during rasterization. However, there is a problem with using this approach. Consider the case where the same sphere mesh needs to be rendered in both Gouraud and Flat shaded modes. To give the illusion of a faceted mesh being smooth, the normals at the vertices are averages of the normals of the adjacent faces. In short, on a sphere, no vertex will have a normal that is exactly perpendicular to any face it is a part of. For Gouraud shading, this is intentional. This is what allows the sphere to look smooth even though it is comprised of a finite number of polygons. However, for flat shading, this will give ill results as shown by the diagram below.

Values Interpolated with D3D_INTERPOLATION_CONST

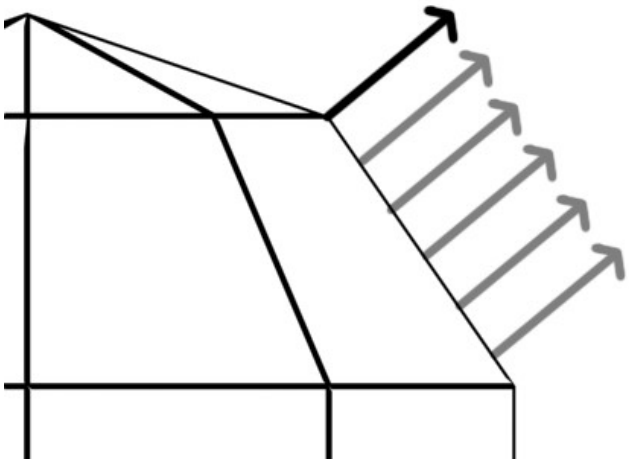
The lighting values for the first vertex are replicated across the triangle.
In many cases, this value is based off of a smooth normal.



D3D10_INTERPOLATION_CONST takes the value of the color calculated at the first vertex and spreads it across the entire triangle giving us shading that looks as if the normal was *bent* compared to the orientation of the triangle. A better method using the geometry shader to calculate the normal is shown below.

Normals constructed in the Geometry Shader

These lighting values are from normals constructed in the geometry shader.
Because the normal is constructed from the triangle vertices, the lighting matches
what we expect to see.



The second method gives more accurate results. The following code snippet illustrates how the geometry shader constructs a normal from the input world positions (wPos) of the triangle vertices. The lighting value is then calculated from this normal and spread to all vertices of the triangle.

```
//
// Calculate the face normal
//
float3 faceEdgeA = input[1].wPos - input[0].wPos;
```

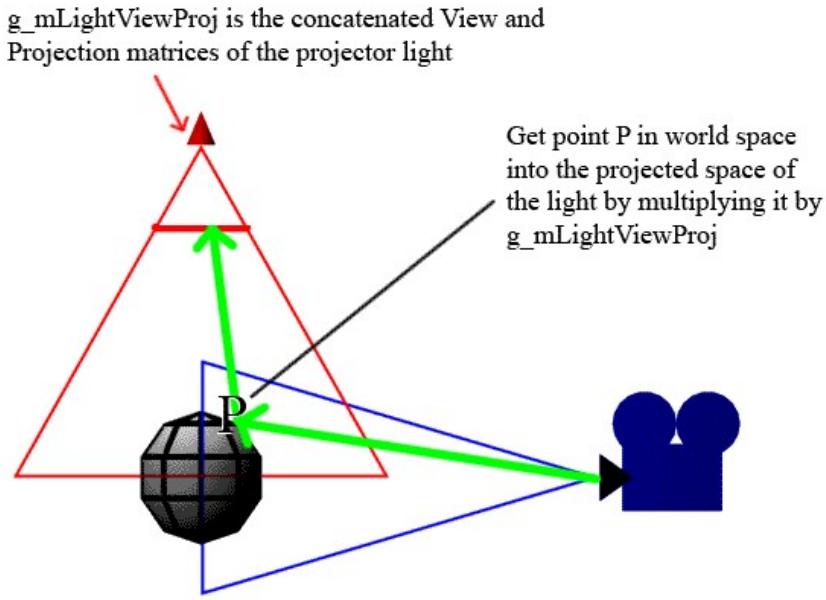
```
float3 faceEdgeB = input[2].wPos - input[0].wPos;

//
// Cross product
//
float3 faceNormal = cross(faceEdgeA, faceEdgeB);
```

Projected Texture Lookups

Projected texturing simply divides the <x,y,z> coordinates of a 4d texture coordinate by the w coordinate before using it in a texture lookup operation. However, without the discussion of projected texturing, it becomes unclear why this functionality is useful.

Projecting a texture onto a surface can be easily illustrated by imagining how a projector works. The projector projects an image onto anything that happens to be in front of the projector. The same effect could be taken care of in the fixed-function pipeline by carefully setting up texture coordinate generation and texture stage states. In Direct3D 10, it is handled in shaders.

To tackle projected texturing, we must think about the problem in reverse. Instead of projecting a texture onto geometry, we simply render the geometry, and for each point, determine where that point would be hit by the projector texture. To do this, we only need to know the position of the point in world space as well as the view and projection matrices of the projector.



By multiplying the world space coordinate by the view and projection matrices of the light, we now have a point that is in the space of the projection of the light. Unfortunately, because we are converting this to texture coordinates, we would really like this point in some sort of [0..1] range. This is where the w coordinate comes into play. After the projection into the projector space, the w coordinate can be thought of as how much this vertex was scaled from a [-1..1] range to get to its current position. To get back to this [-1..1] range, we simply divide by the w coordinate. However, the projected texture is in the [0..1] range, so we must bias the result by halving it and adding 0.5.

```
//calculate the projected texture coordinate for the current world position
float4 cookieCoord = mul( float4(input.wPos,1), g_mLightViewProj );

//since we don't have texldp, we must perform the w divide ourselves befor the texture lookup
cookieCoord.xy = 0.5 * cookieCoord.xy / cookieCoord.w + float2( 0.5, 0.5 );
```

Multi-Texturing

The texture stages from the fixed-function pipeline are officially gone. In there place is the ability to load textures arbitrarily in the pixel shader and to combine them in any way that the math operations of the language allow. The FixedFuncEMU sample emulates the D3DTOP_ADD texture blending operation. The first color is loaded from the diffuse texture at the texture coordinates defined in the mesh and multiplied by the input color.

```
float4 normalColor = tex2D( g_samLinear, g_txDiffuse, input.tex ) * input.colorD + input.colorS;
```

The second color is loaded from the projected texture using the projected coordinates described above.

```
cookieColor = tex2D( g_samLinear, g_txProjected, cookieCoord.xy );
```

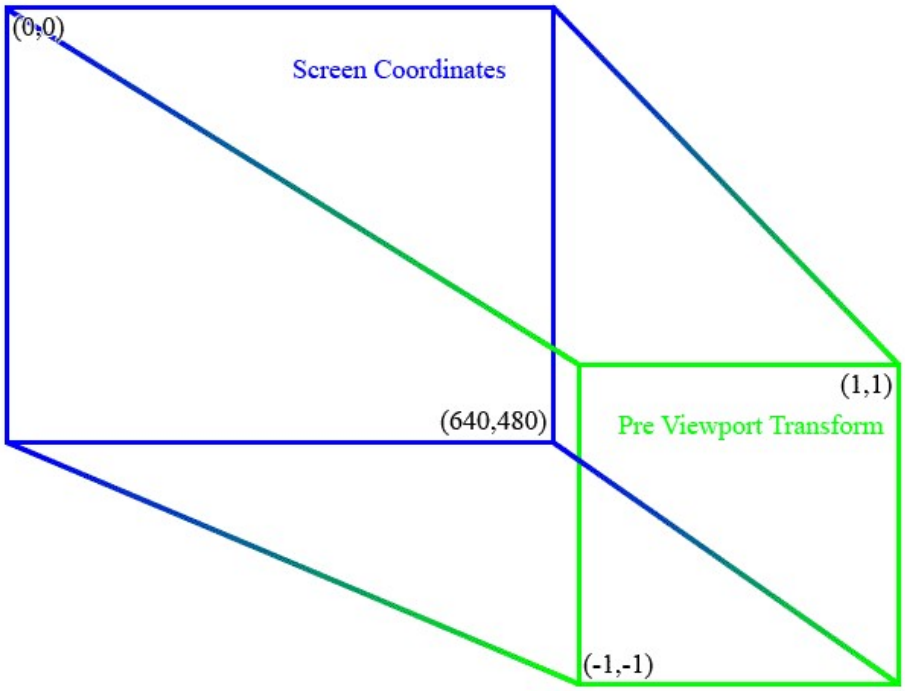
D3DTOP_ADD is simply emulated by adding the projected cookie texture to the normal texture color

```
normalColor += cookieColor;
```

For D3DTOP_MODULATE, the shader would simply multiply the colors together instead of adding them. The effects file can also be extended to handle traditional lightmapping pipeline by passing a second set of texture coordinates stored in the mesh all the way down the pixel shader. The shader would then lookup into a lightmap texture using the second set of texture coordinates instead of the projected coordinates.

Screen space UI rendering

Rendering objects in screen space in Direct3D 10 requires that the user scale and bias the input screen coordinates against the viewport size. For example, a screen space coordinate in the range of <[0..639],[0..479]> needs to be transformed into the range of <[-1..1],[-1..1]>. So that it can be transformed back by the viewport transform.

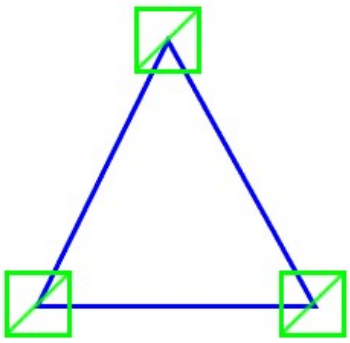


The vertex shader code below performs this transformation. Note that the w coordinate of the position is explicitly set to 1. This ensures that the coordinates will remain the same when the w-divide occurs. Additionally, the Z position is passed into the shader in clip-space, meaning that it is in the [0..1] range, where 0 represents the near plane, and 1 represents the far plane.

```
//output our final position
output.pos.x = (input.pos.x / (g_viewportWidth/2.0)) -1;
output.pos.y = -(input.pos.y / (g_viewportHeight/2.0)) +1;
output.pos.z = input.pos.z;
output.pos.w = 1;
```

D3DFILL_POINT fillmode

The emulation of point fillmode requires the use of the geometry shader to turn one triangle of 3 vertices into 6 triangles of 12 vertices. For each vertex of the input triangle, the geometry shader emits 4 vertices that comprise a two-triangle strip at that position. The positions of these vertices are displaced such that the screen-space size in pixels is equal to the g_pointSize variable.



This sample does not show how to emulate point sprite functionality, which is closely related to point rendering. For point sprite rendering please see the ParticlesGS sample.