User clip planes on feature level 9 hardware

Article • 08/20/2020

Starting with Windows 8, Microsoft High Level Shader Language (HLSL) supports a syntax that you can use with the Microsoft Direct3D 11 API to specify user clip planes on feature level 9_x and higher. You can use this clip-planes syntax to write a shader, and then use that shader object with the Direct3D 11 API to run on all Direct3D feature levels.

Background

You can access user clip planes in the Microsoft Direct3D 9 API via **IDirect3DDevice9::SetClipPlane** and **IDirect3DDevice9::GetClipPlane** methods. In Microsoft Direct3D 10 and later, you can access user clip planes through the SV_ClipDistance semantic. But before Windows 8, SV_ClipDistance was not available for feature level 9_x hardware in the Direct3D 10 or Direct3D 11 APIs. So, before Windows 8, the only way to access user clip planes with feature level 9_x hardware was through the Direct3D 9 API. Direct3D Windows Store apps can't use the Direct3D 9 API. Here we describe the syntax that you can use to access user clip planes through the Direct3D 11 API on feature level 9_x and higher.

Apps use clip planes to define a set of invisible planes within the 3D world that clip (throw away) all drawn primitives. Windows won't draw any pixel that is on the negative side of any clip planes. Apps can then use clip planes to render planar reflections.

Syntax

Use this syntax to declare clip planes as function attributes in a function declaration. For example, here we use the syntax on a vertex shader fragment:

syntax

```
cbuffer ClipPlaneConstantBuffer
{
       float4 clipPlane1;
       float4 clipPlane2;
};

[clipplanes(clipPlane1,clipPlane2)]
VertexShaderOutput main(VertexShaderInput input)
{
       // the rest of the vertex shader doesn't refer to the clip plane

       …

       return output;
}
```

This example for a vertex shader fragment denotes two clip planes. It shows that you need to place the new **clipplanes** attribute within square brackets immediately before the return value of the vertex shader. Within parentheses after the **clipplanes** attribute, you provide a list of up to 6 **float4** constants that define the plane coefficients for each active clip plane. The example also shows that you need to make the coefficients of each plane reside in a constant buffer.

> **Note**
>
> There is no syntax available to disable a clip plane dynamically. You must either recompile an otherwise identical shader with no **clipplanes** attribute, or your app can set the coefficients in your constant buffer to zero so that the plane no longer affects any geometry.

This syntax is available for any 4.0 or later vertex shader target, which includes vs_4_0_level_9_1 and vs_4_0_level_9_3.

Creating clip planes in clip space on feature level 9 and higher

Here we show how to create clip planes in clip space on feature level 9_x and higher.

Background reading

"Introduction to 3D Game Programming with DirectX 10" by Frank D. Luna explains the graphics math background (chapters 1, 2 and 3) you need, and the various spaces and space transformations that occur in the vertex shader (sections 5.6 and 5.8).

10Level9 feature levels

In Direct3D 10 and later, you can clip in any space that makes sense, often in world space or view space. But Direct3D 9 uses clip space, which is pre perspective divide projection space. Vectors are in clip space when the vertex shader passes them to stages that follow in the graphics pipeline.

When you write a Windows Store app, you must use 10Level9 feature levels (feature level 9_x) so the app can run on feature level 9_x and higher hardware. Because your app supports feature level 9_x and higher, you must also use the common capability of applying clip planes in clip space.

When you compile a vertex shader with vs_4_0_level_9_1 or later, that vertex shader can use the **clipplanes** attribute. A Direct3D 10 or later object has a dot product of the emitted vertex that contains each of the **float4** global constants specified in the attribute. The Direct3D 9 object has enough meta data to cause the 10Level9 runtime to issue the appropriate calls to **IDirect3DDevice9::SetClipPlane**.

Clip plane math

A clip plane is defined by a vector with 4 components. The first three components define an x, y, z vector that emanates from the origin in the space we want to clip. This vector implies a plane, perpendicular to the vector and running through the origin. Windows keeps all pixels on the vector side of the plane and clips all pixels behind the plane. The forth w component pushes the plane back and causes Windows to clip less (a negative w causes Windows to clip more) along the vector line. If the x, y, z components compose a unit (normalized) vector, w pushes the plane w units back.

The math that the graphics processing unit (GPU) performs to determine clipping is a simple dot product between the vertex vector (x, y, z, 1) and the clipping plane vector. This math operation creates a projection length on the clip plane vector. A negative dot product shows the vertex to be on the clipped side of the plane.

Clipping in view space

Here is our vertex in view space:

$$\mathbf{v} = \begin{bmatrix} v_x & v_y & v_z & 1 \end{bmatrix}$$

Here is our clip plane in view space:

$$\mathbf{C} = \begin{bmatrix} C_x & C_y & C_z & C_w \end{bmatrix}$$

Here is the dot product of vertex and clip plane in view space:

ClipDistance = $\mathbf{v} \cdot \mathbf{C} = v_xC_x + v_yC_y + v_zC_z + C_w$

This math operation works for a Direct3D 10 or later object but won't work for a Direct3D 9 object. For Direct3D 9, we must first get through our projection transform into clip space.

Projection matrix

A projection matrix transforms a vertex from view space (where the origin is the viewer's eye, +x is to the right, +y is up, and +z is straight ahead) into clip space. The projection matrix readies the vertex for hardware clipping and the rasterization stage. Here is a standard perspective matrix (other projections require different math):
r ratio of window width/height *α* viewing angle *f* distance from the viewer to the far plane *n* distance from the viewer to the near plane
![projection matrix](images/projection-matrix.png)

The next matrix is a simplified version of the previous matrix. We show the matrix simplified so we can use it later in the matrix multiply operation.

$$\mathbf{P} = \begin{bmatrix} P_x & 0 & 0 & 0 \\ 0 & P_y & 0 & 0 \\ 0 & 0 & A & 1 \\ 0 & 0 & B & 0 \end{bmatrix}$$

Now we transform our view space vertex into clip space with a matrix multiply:

$$\mathbf{v}\,\mathbf{P} = \begin{bmatrix} v_x P_x & v_y P_y & v_z A_y + B & v_z \end{bmatrix}$$

In our matrix multiply operation, our x and y components are only slightly adjusted, but our z and w components are quite mangled. Our clip plane won't give us what we want any more.

Clip space clip plane

Here we want to create a clip space clip plane whose dot product with our clip space vertex gives us the same value as **v** · **C** in the Clipping in view space section.

$$\mathbf{C_P} = \begin{bmatrix} C_{P_x} & C_{P_y} & C_{P_z} & C_{P_w} \end{bmatrix}$$

$\mathbf{v} \cdot \mathbf{C} = \mathbf{v}\,\mathbf{P} \cdot \mathbf{C_P}$

$v_x C_x + v_y C_y + v_z C_z + C_w = v_x P_x C_{P_x} + v_y P_y C_{P_y} + v_z A_y C_{P_z} + B C_{P_z} + v_z C_{P_w}$

Now we can break the preceding math operation up by vertex component into four separate equations:

$$v_x C_x = v_x P_x C_{p_x} \rightarrow C_{p_x} = \frac{C_x}{P_x}$$

$$v_y C_y = v_y P_y C_{p_y} \rightarrow C_{p_y} = \frac{C_y}{P_y}$$

$$C_w = B C_{p_z} \rightarrow C_{p_z} = \frac{C_w}{B}$$

$$v_z C_z = v_z A_y C_{p_z} + v_z C_{p_w} \rightarrow C_z = A_y C_{p_z} + C_{p_w} \rightarrow C_{p_w} = C_z - A_y C_{p_z} \rightarrow C_{p_w} = C_z - \frac{C_w A}{B}$$

Our view space clip plane and our projection matrix derive and give us our clip space clip plane.

$$\mathbf{C_P} = \begin{bmatrix} \frac{C_x}{P_x} & \frac{C_y}{P_y} & \frac{C_w}{B} & C_z - \frac{C_w A}{B} \end{bmatrix}$$

Related topics

Programming Guide for HLSL

Function Declaration Syntax

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A