Draw Predicated Sample

⊟ Collapse All

This sample uses the occlusion predicate query to avoid drawing unnecessary geometry.

Path

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

How the Sample Works

The scene contains four meshes, the center column mesh, the microscope mesh, the city mesh, and the microscope occlusion mesh. The microscope mesh is a high resolution mesh that comprises the focus of the scene. Because of the large number of vertices, this mesh is costly to draw especially when duplicated 6 times throughout the scene. The microscope occlusion mesh is a much lower resolution mesh that is used as an approximation of the microscope. For games, this could be a mesh created specifically for occlusion purposes or a collision mesh. When the camera traverses the scene, the city mesh occludes the microscope mesh. Traditionally, the vertex processing cost of the 6 microscope meshes would still be present even when the meshes are completely occluded by the city mesh.

The DrawPredicated sample removes the vertex processing cost by using the D3D10_QUERY_OCCLUSION_PREDICATE query type in Direct3D 10. This query allows a drawing operation to succeed or fail based upon whether pixels from a previous draw operation were visible. Because the predicate query is created with D3D10_QUERY_MISC_PREDICATEHINT, the result of the query can be used to draw or not draw geometry without waiting for the result to be transferred from the GPU to the CPU.

Creating the Query

```
//
// Create 6 occlusion predicate queries
//
ID3D10Predicate            * g_pPredicate;
...
D3D10_QUERY_DESC qdesc;
qdesc.MiscFlags = D3D10_QUERY_MISC_PREDICATEHINT;
qdesc.Query = D3D10_QUERY_OCCLUSION_PREDICATE;
for( int i=0; i<NUM_MICROSCOPE_INSTANCES; i++ )
{
    V_RETURN(pd3dDevice->CreatePredicate( &qdesc, &g_pPredicate[i] ));
}
```

Issuing the Query

```
// Render the occluder mesh
g_pPredicate[i]->Begin();
g_OccluderMesh.Render( pd3dDevice, g_pRenderOccluder );
g_pPredicate[i]->End();
```

Drawing based upon the results

```
//
// Render the vertex heavy meshes
//

// Render the vertex heavy mesh (but only if the predicate passed)
pd3dDevice->SetPredication( g_pPredicate[i], FALSE );
...
g_HeavyMesh.Render( pd3dDevice, g_pRenderTextured, g_pDiffuseTex );
...
pd3dDevice->SetPredication( NULL, FALSE );
```

Limitations

These drawing and execution APIs honor predication:

- **ID3D10Device::Draw**
- **ID3D10Device::DrawAuto**
- **ID3D10Device::DrawIndexed**
- **ID3D10Device::DrawIndexedInstanced**
- **ID3D10Device::ClearRenderTargetView**
- **ID3D10Device::ClearDepthStencilView**
- **ID3D10Device::CopySubresourceRegion**
- **ID3D10Device::CopyResource**
- **ID3D10Device::UpdateSubresource**

These state setting API calls do not honor predication:

- **ID3D10Device::IASetPrimitiveTopology**
- **ID3D10Device::IASetInputLayout**
- **ID3D10Device::IASetVertexBuffers**
- **ID3D10Device::IASetIndexBuffer**
- Present
- Map/Unmap
- Any resource creation call

Predicate hints are non-stalling and are therefore not guaranteed. Because the hardware may not have finished drawing the occluding mesh before the result of the predicate is needed, the vertex-laden meshes may draw even though the occluder mesh is completely hidden. As a result, an application needs to be careful that the depth and stencil buffers contain the correct information regardless of whether the rendering occurs. Use occlusion culling to guarantee the integrity of the depth and stencil buffers. To ensure that the results of the predicate hints are useful, the application should ensure that enough drawing takes place between the issue of the query and the use of the query to ensure synchronization.

On the other hand, using a predicate is generally more efficient than locking a buffer and testing a rectangle, as the CPU and GPU do not have to synchronize with each other. The predicate operation occurs within the command buffer, so you may need to add a small delay between the predicate generation and the actual predication (implementing a 100 ms stall may yield better performance than a CPU/GPU synchronize). Of course, you will want to evaluate your particular scenario to see if predication yields the best performance.