

Game Programming with DirectX

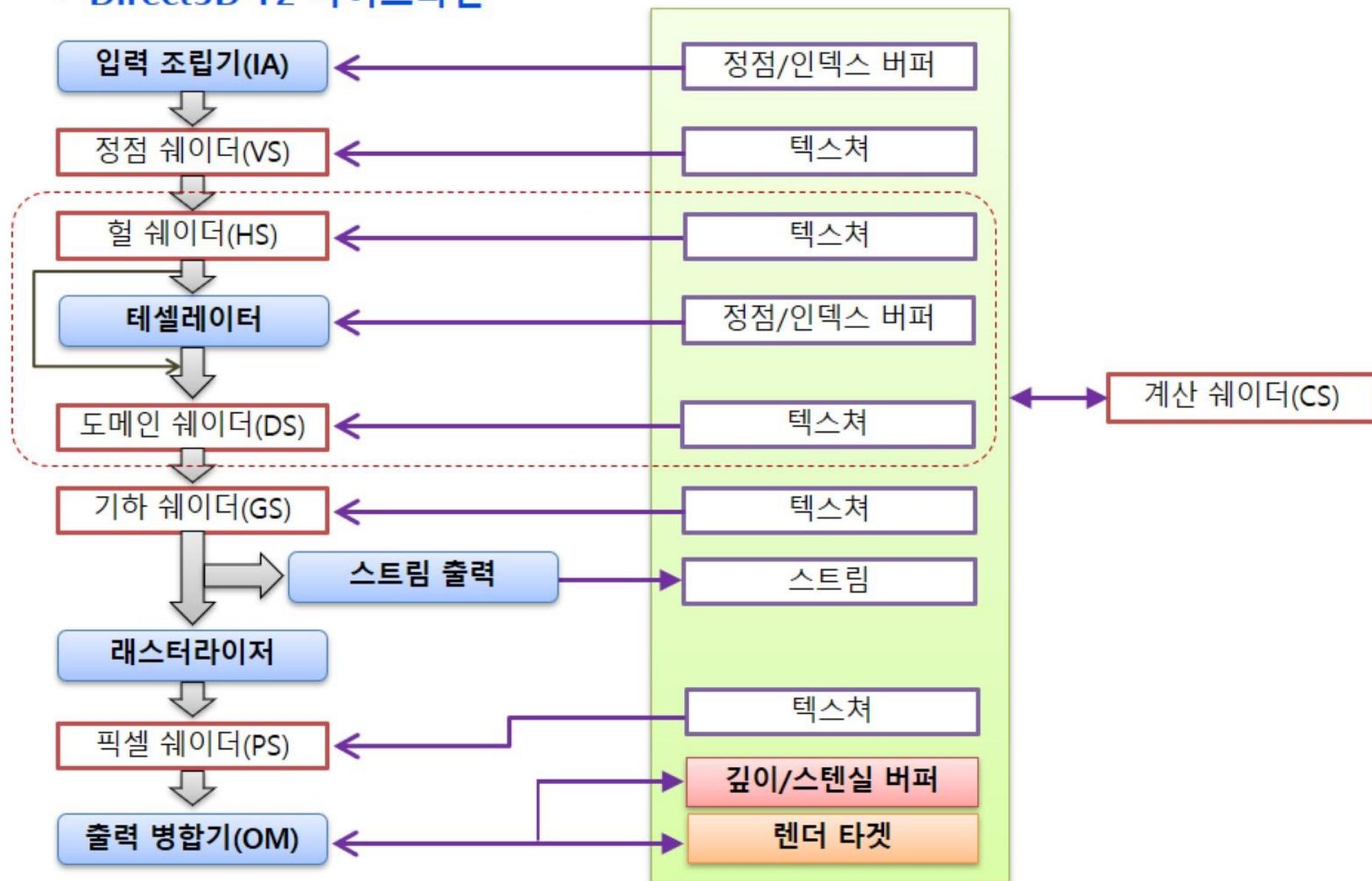
Direct3D Graphics Pipeline

스트림-출력 단계

(Stream-Output Stage)

Direct3D 파이프라인

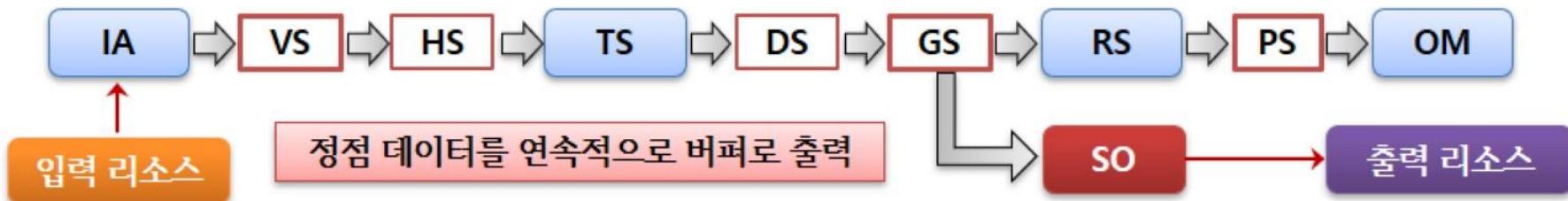
- Direct3D 12 파이프라인



Direct3D 파이프라인

• 스트림-출력 단계(Stream-Output Stage)

- 스트림-출력(SO) 단계는 기하-쉐이더 또는 정점-쉐이더에서 출력되는 정점 데이터를 연속적으로 메모리 버퍼로 GPU를 사용하여 (스트림)출력함
- 스트림-출력으로 버퍼로 출력된 데이터는 렌더링 파이프라인에서 다시 사용 가능 CPU에서 읽을 수 있도록 다른 리소스로 복사(CopyResource)할 수 있음
스트림 출력되는 데이터의 양은 스트림 출력 카운터를 통하여 알 수 있음
- 정점들은 항상 완전한 프리미티브를 구성하도록 출력됨
예: 하나의 삼각형은 3개의 정점으로 출력
스트립 프리미티브는 리스트로 변경되어 스트림 출력됨
인접성 프리미티브는 스트림 출력되기 전에 인접성 데이터가 제거되어 출력됨
- 스트림-출력 데이터를 파이프라인에서 사용하는 방법
 - 입력-조립 단계로 다시 입력
 - 쉐이더에서 Load와 같은 함수를 사용
- 스트림-출력 단계에 동시에 4개의 버퍼를 연결할 수 있음(출력 슬롯이 4개)
- 하나의 버퍼로 출력하면 정점마다 128개의 스칼라(512바이트)를 출력할 수 있음
- 여러 개의 버퍼로 출력하면 각 버퍼는 정점의 한 원소(4 요소)를 출력할 수 있음
- 루트 시그너쳐가 D3D12_ROOT_SIGNATURE_FLAG_ALLOW_STREAM_OUTPUT를 허용



Direct3D 파이프라인

• 그래픽 파이프라인 상태

```
typedef struct D3D12_GRAPHICS_PIPELINE_STATE_DESC {  
    ID3D12RootSignature *pRootSignature;  
    D3D12_SHADER_BYTECODE VS;  
    D3D12_SHADER_BYTECODE PS;  
    D3D12_SHADER_BYTECODE DS;  
    D3D12_SHADER_BYTECODE HS;  
    D3D12_SHADER_BYTECODE GS;  
    D3D12_STREAM_OUTPUT_DESC StreamOutput;  
    D3D12_BLEND_DESC BlendState;  
    UINT SampleMask;  
    D3D12_RASTERIZER_DESC RasterizerState;  
    D3D12_DEPTH_STENCIL_DESC DepthStencilState;  
    D3D12_INPUT_LAYOUT_DESC InputLayout;  
    D3D12_INDEX_BUFFER_STRIP_CULL_MODE_DESC IndexStripCull;  
    D3D12_PRIMITIVE_TOPOLOGY_DESC PrimitiveTopology;  
    UINT NumRenderTargets;  
    DXGI_FORMAT RTVFormats[8];  
    DXGI_FORMAT DSVFormat;  
    DXGI_SAMPLE_DESC SampleDesc;  
    UINT NodeMask;  
    D3D12_CACHED_PIPELINE_STATE Cache;  
    D3D12_PIPELINE_STATE_FLAGS Flags;  
} D3D12_GRAPHICS_PIPELINE_STATE_DESC;
```

```
typedef struct D3D12_SHADER_BYTECODE {  
    void *pShaderBytecode;  
    SIZE_T BytecodeLength;  
} D3D12_SHADER_BYTECODE;
```

```
typedef struct D3D12_DEPTH_STENCIL_DESC {  
    BOOL DepthEnable;  
    D3D12_DEPTH_WRITE_MASK DepthWriteMask;  
    D3D12_COMPARISON_FUNC DepthFunc;  
    BOOL StencilEnable;  
    UINT8 StencilReadMask;  
    UINT8 StencilWriteMask;  
} D3D12_DEPTH_STENCIL_DESC;
```

```
typedef struct D3D12_STREAM_OUTPUT_DESC {  
    D3D12_SO_DECLARATION_ENTRY *pSODDeclaration;  
    UINT NumEntries; //개수  
    UINT *pBufferStrides; //버퍼의 원소 크기  
    UINT NumStrides; //배열의 원소 개수  
    UINT RasterizedStream; //래스터라이저 단계 스트림 인덱스  
} D3D12_STREAM_OUTPUT_DESC;
```

```
    D3D12_RASTERIZER_DESC RasterizerState;  
    BOOL AntialiasedLineEnable;  
    UINT ForceSampleCount;  
    D3D12_CULL_MODE CullMode;
```

```
typedef
```

```
typedef enum D3D12_PIPELINE_STATE_FLAGS {  
    D3D12_PIPELINE_STATE_FLAG_NONE,  
    D3D12_PIPELINE_STATE_FLAG_TOOL_DEBUG  
} D3D12_PIPELINE_STATE_FLAGS;
```

```
    D3D12_PRIMITIVE_TOPOLOGY_TYPE_LINE,  
    D3D12_PRIMITIVE_TOPOLOGY_TYPE_TRIANGLE,  
    D3D12_PRIMITIVE_TOPOLOGY_TYPE_PATCH  
} D3D12_PRIMITIVE_TOPOLOGY_TYPE;
```

Direct3D 파이프라인

- 루트 시그너쳐(Root Signature)

- 기하 쉐이더가 스트림 출력을 포함하면 루트 시그너쳐가 SO 단계를 허용해야 함

```
HRESULT WINAPI D3D12SerializeRootSignature(  
    D3D12_ROOT_SIGNATURE_DESC *pRootSignature,  
    D3D_ROOT_SIGNATURE_VERSION Version,  
    ID3DBlob **ppBlob, //직렬화된 루트 시그너쳐  
    ID3DBlob **ppErrorBlob  
);
```

```
typedef struct D3D12_ROOT_SIGNATURE_DESC {  
    UINT NumParameters; //루트 시그너쳐의 슬롯(파라메터) 개수  
    D3D12_ROOT_PARAMETER *pParameters;  
    UINT NumStaticSamplers; //정적 샘플러의 개수(2032개)  
    D3D12_STATIC_SAMPLER_DESC *pStaticSamplers;  
    D3D12_ROOT_SIGNATURE_FLAGS Flags; //루트 시그너쳐 레이아웃을 위한 선택 사항  
} D3D12_ROOT_SIGNATURE_DESC;
```

```
typedef enum D3D12_ROOT_SIGNATURE_FLAGS {  
    D3D12_ROOT_SIGNATURE_FLAG_NONE,  
    D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT, //IA 단계를 허용  
    D3D12_ROOT_SIGNATURE_FLAG_DENY_VERTEX_SHADER_ROOT_ACCESS, //정점 쉐이더가 접근할 수 없음  
    D3D12_ROOT_SIGNATURE_FLAG_DENY_HULL_SHADER_ROOT_ACCESS, //헐 쉐이더가 접근할 수 없음  
    D3D12_ROOT_SIGNATURE_FLAG_DENY_DOMAIN_SHADER_ROOT_ACCESS, //도메인 쉐이더가 접근할 수 없음  
    D3D12_ROOT_SIGNATURE_FLAG_DENY_GEOMETRY_SHADER_ROOT_ACCESS, //기하 쉐이더가 접근할 수 없음  
    D3D12_ROOT_SIGNATURE_FLAG_DENY_PIXEL_SHADER_ROOT_ACCESS, //픽셀 쉐이더가 접근할 수 없음  
    D3D12_ROOT_SIGNATURE_FLAG_ALLOW_STREAM_OUTPUT //SO 단계를 허용  
} D3D12_ROOT_SIGNATURE_FLAGS;
```

Direct3D 파이프라인

• 스트림 출력 단계의 설정

- 스트림 출력 버퍼의 생성
스트림 출력 버퍼(정점 버퍼)는 모든 힙 유형으로 생성할 수 있음
- 스트림 출력의 위치를 저장할 버퍼를 생성
스트림 출력 카운터(Stream Output Counter, BufferFilledSize)를 위한 버퍼 생성
카운터 버퍼는 CPU에서 출력 위치(Offset)을 변경하기 위하여 접근할 수 있음
- 스트림 출력 버퍼와 카운터가 출력에 사용될 때 스트림 출력 상태이어야 함

D3D12_RESOURCE_STATE_STREAM_OUT

- 파이프라인 상태를 생성할 때 스트림 출력 버퍼의 구조를 설정해야 함

```
typedef struct D3D12_STREAM_OUTPUT_DESC {  
    D3D12_SO_DECLARATION_ENTRY *pSODDeclaration; //스트림 출력 선언  
    UINT NumEntries; //스트림 출력 선언의 개수(64개)  
    UINT *pBufferStrides; //((스트림 출력 버퍼의 원소 크기) 배열  
    UINT NumStrides; //배열의 원소 개수  
    UINT RasterizedStream; //래스터라이저 단계로 출력될 스트림 인덱스, D3D12_SO_NO_RASTERIZED_STREAM  
} D3D12_STREAM_OUTPUT_DESC;
```

```
typedef struct D3D12_SO_DECLARATION_ENTRY { //스트림 출력 버퍼(정점 버퍼)의 정점 원소를 표현  
    UINT Stream; //스트림 인덱스  
    LPCSTR SemanticName; //시맨틱 이름("POSITION", "NORMAL", "TEXCOORD0")  
    UINT SemanticIndex; //시맨틱 인덱스  
    BYTE StartComponent; //출력 시작 요소(0: x, 1: y, 2: z, 3: w)  
    BYTE ComponentCount; //출력 요소 개수(1~4)  
    BYTE OutputSlot; //스트림 출력 버퍼 슬롯(0~3), ID3D12GraphicsCommandList::SOSetTargets()  
} D3D12_SO_DECLARATION_ENTRY;
```

Direct3D 파이프라인

- **스트림 출력 단계의 설정**

- 스트림 출력 버퍼의 구조 설정

D3D12_SO_DECLARATION_ENTRY는 정점이 버퍼로 출력되는 방법을 정의

- 쉐이더 모델 5에서 기하 쉐이더는 4개까지의 출력 스트림을 사용할 수 있음

```
D3D12_SO_DECLARATION_ENTRY pSODecls[ ] = {
{ 0, "SV_POSITION", 0, 0, 4, 0 }, //위치 벡터의 모든 요소(4개: x, y, z, w)를 0번 슬롯의 버퍼로 출력
{ 0, "NORMAL", 0, 0, 3, 0 }, //법선 벡터의 요소(3개: x, y, z)를 0번 슬롯의 버퍼로 출력
{ 0, "TEXCOORD", 0, 0, 2, 0 } //텍스쳐 좌표의 요소(2개: x, y)를 0번 슬롯의 버퍼로 출력
};
```

```
D3D12_STREAM_OUTPUT_DESC d3dStreamOutputDesc;
d3dStreamOutputDesc.pSODeclaration = pSODecls;
d3dStreamOutputDesc.NumEntries = sizeof(pSODecls);
d3dStreamOutputDesc.pBufferStrides = NULL;
d3dStreamOutputDesc.NumStrides = d3dStreamOutputDesc.RasterizedStream = 0;
```

```
D3D12_GRAPHICS_PIPELINE_STATE_DESC d3dGraphicsPipelienStateDesc;
d3dGraphicsPipelienStateDesc.pRootSignature = ...;
d3dGraphicsPipelienStateDesc.GS = CreateGeometryShader(&pd3dPixelShaderBlob);
d3dGraphicsPipelienStateDesc.StreamOutput = d3dStreamOutputDesc;
...
```

```
void GS(GS_IN input[0], inout PointStream<GS_OUT> stream)
{
    GS_OUT vert = TransformVertex1(input[0]);
    stream.Append(vert);
}
```

Direct3D 파이프라인

• 기하 쉐이더(Geometry Shader)

[maxvertexcount(NumVerts)]

```
void GeometryShaderName(  
    PrimitiveType DataType Name[NumElements],  
    inout StreamOutputObject<DataType> Name  
)
```

쉐이더 모델 5에서 기하 쉐이더는 4개까지의 출력 스트림을 사용할 수 있음

NumVerts	생성할 정점의 최대 개수
ShaderName	기하 쉐이더 이름(문자열)
PrimitiveType	프리미티브 유형(point, line, triangle, lineadj, triangleadj)
DataType	입력 데이터 형식(HLSL 데이터 형식)
Name	파라미터 이름(문자열)
NumElements	파라미터 원소의 개수
StreamOutputObject	스트림-출력 객체, 템플릿 형태로 사용(PointStream, LineStream, TriangleStream)

PrimitiveType

point	점들의 리스트, [1]
line	선분들의 리스트 또는 스트립, [2]
triangle	삼각형들의 리스트 또는 스트립, [3]
lineadj	인접성 선분들의 리스트 또는 스트립, [4]
triangleadj	인접성 삼각형들의 리스트 또는 스트립, [6]

StreamOutputObject

PointStream	출력 프리미티브가 점인 경우
LineStream	출력 프리미티브가 선분인 경우
TriangleStream	출력 프리미티브가 삼각형인 경우
Append	출력 데이터를 스트림에 추가
RestartStrip	새로운 프리미티브 스트립을 시작

Direct3D 파이프라인

- **스트림 출력 단계의 설정**

- 스트림 출력 버퍼의 구조 설정

D3D12_SO_DECLARATION_ENTRY는 정점이 버퍼로 출력되는 방법을 정의

- 쉐이더 모델 5에서 기하 쉐이더는 4개까지의 출력 스트림을 사용할 수 있음

```
D3D12_SO_DECLARATION_ENTRY pSODecls[ ] = {  
    { 0, "SV_POSITION", 0, 0, 4, 0 }, //위치 벡터의 모든 요소(4개: x, y, z, w)를 0번 슬롯의 버퍼로 출력  
    { 0, "NORMAL", 0, 0, 3, 0 }, //법선 벡터의 요소(3개: x, y, z)를 0번 슬롯의 버퍼로 출력  
    { 1, "TEXCOORD", 0, 0, 2, 1 }, //텍스쳐 좌표의 요소(2개: x, y)를 1번 슬롯의 버퍼로 출력  
};
```

```
D3D12_STREAM_OUTPUT_DESC d3dStreamOutputDesc;  
d3dStreamOutputDesc.pSODeclaration = pSODecls;  
d3dStreamOutputDesc.NumEntries = sizeof(pSODecls);  
d3dStreamOutputDesc.pBufferStrides = NULL;  
d3dStreamOutputDesc.NumStrides = d3dStreamOutputDesc.RasterizedStream = 0;  
D3D12_GRAPHICS_PIPELINE_STATE_DESC d3dGraphicsPipelienStateDesc;  
d3dGraphicsPipelienStateDesc.pRootSignature = ...;  
d3dGraphicsPipelienStateDesc.GS = CreateGeometryShader(&pd3dPixelShaderBlob);  
d3dGraphicsPipelienStateDesc.StreamOutput = d3dStreamOutputDesc;  
...
```

```
void GS(GS_IN input[2], inout PointStream<GS_OUT1> stream1, inout PointStream<GS_OUT2> stream2)  
{  
    GS_OUT1 vert1 = TransformVertex1(input[0]);  
    GS_OUT2 vert2 = TransformVertex2(input[1]);  
    stream1.Append(vert1);  
    stream2.Append(vert2);  
}
```

Direct3D 파이프라인

- **ID3D12Device::CreateCommittedResource**

- 스트림 출력 버퍼는 모든 힙 유형으로 생성할 수 있음

```
HRESULT ID3D12Device::CreateCommittedResource(
```

```
    D3D12_HEAP_PROPERTIES *pHeapProperties,
```

```
    D3D12_HEAP_FLAGS HeapFlags,
```

```
    D3D12_RESOURCE_DESC *pResourceDesc,
```

```
    D3D12_RESOURCE_STATES InitialResourceState, //리소스가 사용되는 방법에 따른 리소스의 상태
```

```
    D3D12_CLEAR_VALUE *pOptimizedClearValue, //NULL: D3D12_RESOURCE_DIMENSION_BUFFER
```

```
    REFIID riidResource, //__uuidof(ID3D12Resource)
```

```
    void **ppvResource
```

```
);
```

```
typedef enum D3D12_HE
```

```
    D3D12_HEAP_FLAG_NOC
```

```
    D3D12_HEAP_FLAG_SH
```

```
    D3D12_HEAP_FLAG_DE
```

```
    D3D12_HEAP_FLAG_AL
```

```
    D3D12_HEAP_FLAG_SH
```

```
    D3D12_HEAP_FLAG_DE
```

```
    D3D12_HEAP_FLAG_AL
```

```
    } D3D12_HEAP_FLAGS;
```

```
    D3D12_TEXTURE_LAYOUT Layout;
```

```
    D3D12_RESOURCE_FLAGS Flags;
```

```
    } D3D12_RESOURCE_DESC;
```

```
    ID3D12Device::CreatePlacedResource(...)
```

```
    ID3D12Device::CreateReservedResource(...)
```

```
typedef struct D3D12_HEAP_PROPERTIES {
```

```
    D3D12_HEAP_TYPE Type;
```

```
    D3D12_CPU_PAGE_PROPERTY CPUPageProperty;
```

```
    D3D12_MEMORY_POOL MemoryPoolPreference;
```

```
    UINT CreationNodeMask;
```

```
    UINT VisibleNodeMask;
```

```
    } D3D12_HEAP_PROPERTIES;
```

```
typedef struct D3D12_CLEAR_VALUE {
```

```
    DXGI_FORMAT Format;
```

```
    union {
```

```
        FLOAT Color[4];
```

```
        D3D12_DEPTH_STENCIL_VALUE DepthStencil;
```

```
    };
```

```
    } D3D12_CLEAR_VALUE;
```

```
    D3D12_RESOURCE_STATE_PREDICATION
```

```
    } D3D12_RESOURCE_STATES;
```

Direct3D 파이프라인

- **ID3D12Device::CreateCommittedResource()**

```
typedef struct D3D12_HEAP_PROPERTIES {  
    D3D12_HEAP_TYPE Type; //힙의 유형  
    D3D12_CPU_PAGE_PROPERTY CPUPageProperty; //힙에 대한 CPU-페이지 속성  
    D3D12_MEMORY_POOL MemoryPoolPreference; //힙에 대한 메모리 풀(Pool)  
    UINT CreationNodeMask; //0, 1: 단일 GPU 어댑터  
    UINT VisibleNodeMask; //다중 어댑터에서 리소스를 볼 수 있는 노드의 집합을 나타내는 비트  
} D3D12_HEAP_PROPERTIES;
```

```
typedef enum D3D12_HEAP_TYPE {
```

D3D12_HEAP_TYPE_DEFAULT, //GPU는 읽고 쓰기 가능, CPU는 접근할 수 없음, 대부분의 리소스에 적용
D3D12_HEAP_TYPE_UPLOAD, //업로드를 위한 CPU 접근에 최적화, _RESOURCE_STATE_GENERIC_READ
D3D12_HEAP_TYPE_READBACK, //읽기 위한 힙, CPU 접근에 최적화, _RESOURCE_STATE_COPY_DEST
D3D12_HEAP_TYPE_CUSTOM

```
} D3D12_HEAP_TYPE;
```

D3D12_HEAP_TYPE_UPLOAD 힙의 전형적인 사용

- ① _DEFAULT 힙의 리소스를 CPU 데이터로 초기화
- ② 상수 버퍼의 동적 데이터를 반복적으로 업로드(갱신)

```
typedef enum D3D12_CPU_PAGE_PROPERTY {
```

D3D12_CPU_PAGE_PROPERTY_UNKNOWN,
D3D12_CPU_PAGE_PROPERTY_NOT_AVAILABLE, //CPU는 힙에 접근할 수 없음
D3D12_CPU_PAGE_PROPERTY_WRITE_COMBINE,
D3D12_CPU_PAGE_PROPERTY_WRITE_BACK

```
} D3D12_CPU_PAGE_PROPERTY;
```

```
typedef enum D3D12_MEMORY_POOL {
```

D3D12_MEMORY_POOL_UNKNOWN,
D3D12_MEMORY_POOL_L0, //물리적 시스템 메모리 풀
D3D12_MEMORY_POOL_L1 //물리적 비디오 메모리 풀

```
} D3D12_MEMORY_POOL;
```

Direct3D 파이프라인

- **ID3D12Device::CreateCommittedResource()**

```
typedef enum D3D12_HEAP_FLAGS {  
    D3D12_HEAP_FLAG_NONE,  
    D3D12_HEAP_FLAG_SHARED, //다중 프로세서에서 힙을 공유(CPU 마샬링(Marshaling)이 필요없음)  
    D3D12_HEAP_FLAG_DENY_BUFFERS, //버퍼를 포함하도록 허용되지 않음  
    D3D12_HEAP_FLAG_ALLOW_DISPLAY, //스왑 체인 표면을 포함하도록 허용  
    D3D12_HEAP_FLAG_SHARED_CROSS_ADAPTER, //다중 어댑터에서 힙을 공유  
    D3D12_HEAP_FLAG_DENY_RT_DS_TEXTURES, //RT/DS 텍스쳐를 포함하도록 허용되지 않음  
    D3D12_HEAP_FLAG_DENY_NON_RT_DS_TEXTURES, //RT/DS가 아닌 텍스쳐를 포함하도록 허용되지 않음  
    D3D12_HEAP_FLAG_ALLOW_ALL_BUFFERS_AND_TEXTURES, //모든 유형의 버퍼와 텍스쳐를 포함하도록 허용  
    D3D12_HEAP_FLAG_ALLOW_ONLY_BUFFERS, //버퍼만 포함하도록 허용  
    D3D12_HEAP_FLAG_ALLOW_ONLY_NON_RT_DS_TEXTURES, //RT/DS가 아닌 텍스쳐를 포함하도록 허용  
    D3D12_HEAP_FLAG_ALLOW_ONLY_RT_DS_TEXTURES //RT/DS 텍스쳐만 포함하도록 허용  
} D3D12_HEAP_FLAGS;
```

```
typedef struct D3D12_CLEAR_VALUE { //최적의 지우기 연산을 위한 값
```

```
    DXGI_FORMAT Format;
```

```
    union {
```

```
        FLOAT Color[4]; //D3D12_RESOURCE_FLAG_ALLOW_RENDER_TARGET
```

```
        D3D12_DEPTH_STENCIL_VALUE DepthStencil; //D3D12_RESOURCE_FLAG_ALLOW_DEPTH_STENCIL
```

```
    };
```

```
} D3D12_CLEAR_VALUE;
```

```
typedef struct D3D12_DEPTH_STENCIL_VALUE {
```

```
    FLOAT Depth; //깊이 값(0~1.0)
```

```
    UINT8 Stencil; //스텐실 값(0~255)
```

```
} D3D12_DEPTH_STENCIL_VALUE;
```

```
void ID3D12GraphicsCommandList::ClearRenderTargetView(..., FLOAT ColorRGBA[4], ...);
```

```
void ID3D12GraphicsCommandList::ClearDepthStencilView(..., FLOAT Depth, UINT8 Stencil, ...);
```

Direct3D 파이프라인

- **ID3D12Device::CreateCommittedResource()**

```
typedef enum D3D12_RESOURCE_STATES {  
    D3D12_RESOURCE_STATE_COMMON, //0x00, CPU가 텍스쳐에 접근, COPY 큐의 상태 전이 전  
    D3D12_RESOURCE_STATE_VERTEX_AND_CONSTANT_BUFFER, //0x01, 정점 버퍼 또는 상수 버퍼로 사용될 때  
    D3D12_RESOURCE_STATE_INDEX_BUFFER, //0x02, 인덱스 버퍼로 사용될 때  
    D3D12_RESOURCE_STATE_RENDER_TARGET, //0x04, 렌더 타겟으로 사용될 때, ClearRenderTargetView()  
    D3D12_RESOURCE_STATE_UNORDERED_ACCESS, //0x08, 무순서 접근으로 사용될 때  
    D3D12_RESOURCE_STATE_DEPTH_WRITE, //0x10, 깊이 버퍼에 쓸 때, ClearDepthStencilView()  
    D3D12_RESOURCE_STATE_DEPTH_READ, //0x20, 깊이 버퍼를 읽을 때  
    D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_RESOURCE, //0x40, 픽셀 쉐이더 이외의 리소스로 사용될 때  
    D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE, //0x80, 픽셀 쉐이더의 리소스로 사용될 때  
    D3D12_RESOURCE_STATE_STREAM_OUT, //0x100, 스트림 출력으로 사용될 때  
    D3D12_RESOURCE_STATE_INDIRECT_ARGUMENT, //0x200, ExecuteIndirect()  
    D3D12_RESOURCE_STATE_COPY_DEST, //0x400, 복사 연산의 목표로 사용될 때  
    D3D12_RESOURCE_STATE_COPY_SOURCE, //0x800, 복사 연산의 소스로 사용될 때  
    D3D12_RESOURCE_STATE_RESOLVE_DEST, //0x1000, 리졸브 연산의 목표로 사용될 때  
    D3D12_RESOURCE_STATE_RESOLVE_SOURCE, //0x2000, 리졸브 연산의 소스로 사용될 때  
    D3D12_RESOURCE_STATE_GENERIC_READ, //업로드 힙의 시작 상태(가급적 피할 것)  
    D3D12_RESOURCE_STATE_PRESENT, //D3D12_RESOURCE_STATE_COMMON  
    D3D12_RESOURCE_STATE_PREDICATION //리소스가 예측(프리디케이션)을 위하여 사용될 때  
} D3D12_RESOURCE_STATES;
```

D3D12_HEAP_TYPE_UPLOAD: 초기 상태는 D3D12_RESOURCE_STATE_GENERIC_READ

D3D12_HEAP_TYPE_READBACK: 초기 상태는 D3D12_RESOURCE_STATE_COPY_DEST

텍스쳐: CPU 접근을 위하여 D3D12_RESOURCE_STATE_COMMON

D3D12_RESOURCE_STATE_GENERIC_READ: (0x01 | 0x02 | 0x40 | 0x80 | 0x200 | 0x800)

Direct3D 파이프라인

- **ID3D12Device::CreateCommittedResource()**

```
typedef struct D3D12_RESOURCE_DESC {
    D3D12_RESOURCE_DIMENSION Dimension;
    UINT64 Alignment; //정렬: 0(64KB), 4KB, 64KB, 4MB
    UINT64 Width; //리소스의 가로 크기
    UINT Height; //리소스의 세로 크기
    UINT16 DepthOrArraySize; //깊이 또는 배열의 크기
    UINT16 MipLevels; // mip맵 레벨의 개수, 0(자동 계산)
    DXGI_FORMAT Format; //리소스 형식
    DXGI_SAMPLE_DESC SampleDesc; //다중 샘플링
    D3D12_TEXTURE_LAYOUT Layout; //다차원 리소스를 1차원 리소스로 매핑하기 위한 방법
    D3D12_RESOURCE_FLAGS Flags;
} D3D12_RESOURCE_DESC;
```

```
typedef enum D3D12_RESOURCE_DIMENSION {
    D3D12_RESOURCE_DIMENSION_UNKNOWN,
    D3D12_RESOURCE_DIMENSION_BUFFER,
    D3D12_RESOURCE_DIMENSION_TEXTURE1D,
    D3D12_RESOURCE_DIMENSION_TEXTURE2D,
    D3D12_RESOURCE_DIMENSION_TEXTURE3D
} D3D12_RESOURCE_DIMENSION;
```

```
typedef enum D3D12_RESOURCE_FLAGS {
    D3D12_RESOURCE_FLAG_NONE,
    D3D12_RESOURCE_FLAG_ALLOW_RENDER_TARGET, //RTV 허용
    D3D12_RESOURCE_FLAG_ALLOW_DEPTH_STENCIL, //DSV 허용
    D3D12_RESOURCE_FLAG_ALLOW_UNORDERED_ACCESS, //UAV 허용
    D3D12_RESOURCE_FLAG_DENY_SHADER_RESOURCE,
    D3D12_RESOURCE_FLAG_ALLOW_CROSS_ADAPTER,
    D3D12_RESOURCE_FLAG_ALLOW_SIMULTANEOUS_ACCESS
} D3D12_RESOURCE_FLAGS;
```

```
typedef enum D3D12_TEXTURE_LAYOUT {
    D3D12_TEXTURE_LAYOUT_UNKNOWN, //어댑터에 의존(최상의 레이아웃을 선택)
    D3D12_TEXTURE_LAYOUT_ROW_MAJOR, //텍스쳐 데이터가 행 우선 순서로 저장됨
    D3D12_TEXTURE_LAYOUT_64KB_UNDEFINED_SWIZZLE, //드라이버에 의존(레이아웃이 64KB 영역으로 배치)
    D3D12_TEXTURE_LAYOUT_64KB_STANDARD_SWIZZLE //레이아웃이 64KB 영역으로 배치(표준적인 방법)
} D3D12_TEXTURE_LAYOUT;
```

Direct3D 파이프라인

• 스트림 출력 버퍼 연결

```
void ID3D12GraphicsCommandList::SOSetTargets(  
    UINT StartSlot, //스트림 출력 버퍼를 연결할 시작 슬롯  
    UINT NumViews, //연결할 스트림 출력 버퍼의 개수  
    D3D12_STREAM_OUTPUT_BUFFER_VIEW *pViews  
);
```

```
typedef struct D3D12_STREAM_OUTPUT_BUFFER_VIEW {  
    D3D12_GPU_VIRTUAL_ADDRESS BufferLocation; //스트림 출력 버퍼(모든 유형의 힙을 사용 가능)  
    UINT64 SizeInBytes; //스트림 출력 버퍼의 크기(바이트)  
    D3D12_GPU_VIRTUAL_ADDRESS BufferFilledSizeLocation; //스트림 출력 버퍼의 출력 위치를 저장하는 버퍼  
} D3D12_STREAM_OUTPUT_BUFFER_VIEW;
```

스트림 출력 카운터는 스트림 출력 버퍼 또는 별도의 버퍼에 위치

스트림 출력 카운터(Stream Output Counter)

스트림 출력을 할 때 GPU는 출력하기 위한 버퍼의 현재 위치(BufferFilledSize)를 알아야 함
응용 프로그램은 이 위치를 저장하기 위한 버퍼(메모리: 32-비트 크기)를 할당해야 함

스트림 출력 버퍼에 있는 현재까지 출력한 바이트 수를 나타냄(GPU가 오버플로우 처리를 할 수 있음)
GPU가 스트림 출력을 할 때 이 값을 사용하여 새로운 정점을 추가할 위치를 결정. 카운터를 증가시킴
카운터 버퍼는 CPU에서 출력 위치(Offset: 4의 배수)을 변경하기 위하여 접근할 수 있음(어떻게?)

스트림 출력 버퍼와 카운터가 출력에 사용될 때 스트림 출력 상태이어야 함

D3D12_RESOURCE_STATE_STREAM_OUT

기하 쉐이더의 출력은 버퍼로 스트림 출력되거나 래스터라이저 단계로 출력될 수 있음

기하 쉐이더의 출력이 버퍼로만 스트림 출력되도록 하려면 픽셀 쉐이더와 깊이/스텐실 버퍼를 비활성화시킴

픽셀 쉐이더(NULL), DepthEnable=false, DepthWriteMask=D3D12_DEPTH_WRITE_MASK_ZERO

픽셀 쉐이더와 깊이/스텐실 버퍼를 비활성화하면 래스터라이저 단계가 비활성화됨

Direct3D 파이프라인

• 컬링(Culling)

커다란 씬을 렌더링할 때의 문제점(화면에 보이지 않는 객체를 렌더링, "Overdraw")
컬링은 카메라에 보이지 않는 객체(다각형)들을 렌더링하지 않는 것

- 은면 제거(Back Face Culling)

래스터라이저 단계에서 프리미티브의 정점 와인딩 순서에 따라 수행

- 클리핑(Clipping)

래스터라이저 단계에서 투영좌표계에서 절두체를 벗어나는 점들을 버림(GPU)

- 절두체 컬링(Frustum Culling)

카메라 절두체 영역을 벗어난 객체(다각형)들을 렌더링하지 않음(CPU)

바운딩 객체와 절두체의 교차(Intersection) 문제

- 차폐 컬링(Occlusion Culling)

다른 객체에 가려진(차폐된) 객체들을 렌더링하지 않음

정적인 객체들 사이의 차폐 관계는 미리 계산할 수 있음

- 깊이 검사(Depth Test)

[earlydepthstencil]: 픽셀 쉐이더 이전 단계에서 깊이-스텐실 검사를 미리 수행
검사에 성공하지 못하면 픽셀 쉐이더를 호출하지 않음(버림)

- PVS(Potentially Visible Set)

바운딩 객체들의 계층 구조를 사용하여 보일 가능성이 있는 객체들을 구함

- 차폐 쿼리(Occlusion Query)

GPU 쿼리를 사용

D3D12PredicationQueries 샘플

Direct3D 파이프라인

- 캠핑(Culling)

- 차폐 쿼리(Occlusion Query)

- ① 차폐 쿼리 힙(Occlusion Query Heap)을 생성

```
D3D12_QUERY_HEAP_DESC d3dQueryHeapDesc;
::ZeroMemory(&d3dQueryHeapDesc, sizeof(D3D12_QUERY_HEAP_DESC));
d3dQueryHeapDesc.Count = 1;
d3dQueryHeapDesc.Type = D3D12_QUERY_HEAP_TYPE_OCCLUSION;
m_pd3dDevice->CreateQueryHeap(&d3dQueryHeapDesc, IID_PPV_ARGS(&m_pd3dQueryHeap));
```

ID3D12QueryHeap* m_pd3dQueryHeap;

- ② 파이프라인 상태 객체

- 렌더 타겟과 깊이 버퍼 출력을 비활성화

ID3D12PipelineState* m_pd3dQueryState;

```
d3dPSODesc.BlendState.RenderTarget[0].RenderTargetWriteMask = 0;
d3dPSODesc.DepthStencilState.DepthWriteMask = D3D12_DEPTH_WRITE_MASK_ZERO;
m_pd3dDevice->CreateGraphicsPipelineState(&d3dPSODesc, IID_PPV_ARGS(&m_pd3dQueryState));
```

- ③ 쿼리의 결과를 저장할 버퍼를 생성

ID3D12Resource* m_pd3dQueryResult;

```
D3D12_HEAP_PROPERTIES d3dHeapProperties;
::ZeroMemory(&d3dHeapProperties, sizeof(D3D12_HEAP_PROPERTIES));
d3dHeapProperties.Type = D3D12_HEAP_TYPE_DEFAULT;
d3dHeapProperties.CPUPageProperty = D3D12_CPU_PAGE_PROPERTY_UNKNOWN;
d3dHeapProperties.MemoryPoolPreference = D3D12_MEMORY_POOL_UNKNOWN;
d3dHeapProperties.CreationNodeMask = 1;
d3dHeapProperties.VisibleNodeMask = 1;
```

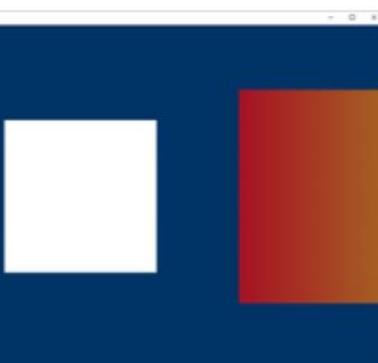
Direct3D 파이프라인

- 케링(Culling)

- 차폐 쿼리(Occlusion Query)
 - ③ 쿼리의 결과를 저장할 버퍼를 생성

```
ID3D12Resource* m_pd3dQueryResult;
```

```
D3D12_RESOURCE_DESC d3dResourceDesc;  
d3dResourceDesc.Dimension = D3D12_RESOURCE_DIMENSION_BUFFER;  
d3dResourceDesc.Alignment = 0;  
d3dResourceDesc.Width = 8;  
d3dResourceDesc.Height = 1;  
d3dResourceDesc.DepthOrArraySize = 1;  
d3dResourceDesc.MipLevels = 1;  
d3dResourceDesc.Format = DXGI_FORMAT_UNKNOWN;  
d3dResourceDesc.SampleDesc.Count = 1;  
d3dResourceDesc.SampleDesc.Quality = 0;  
d3dResourceDesc.Layout = D3D12_TEXTURE_LAYOUT_ROW_MAJOR;  
d3dResourceDesc.Flags = D3D12_RESOURCE_FLAG_NONE;  
m_pd3dDevice->CreateCommittedResource(&d3dHeapProperties, D3D12_HEAP_FLAG_NONE,  
&d3dResourceDesc, D3D12_RESOURCE_STATE_PREDICATION, NULL, IID_PPV_ARGS(&m_pd3dQueryResult));
```



Direct3D 파이프라인

- 케링(Culling)

- 차폐 쿼리(Occlusion Query)

- ④ 차폐 쿼리를 실행

```
m_pd3dCommandList->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP);
m_pd3dCommandList->IASetVertexBuffers(0, 1, &m_pd3dVertexBufferView);
```

```
m_pd3dCommandList->SetGraphicsRootDescriptorTable(0, pd3dcbvFarQuadHandle);
m_pd3dCommandList->SetPredication(m_pd3dQueryResult, 0, D3D12_PREDICATION_OP_EQUAL_ZERO);
m_pd3dCommandList->DrawInstanced(4, 1, 0, 0); //뒤쪽 사각형(바운딩 박스가 차폐되면 실행되지 않음)
```

```
m_pd3dCommandList->SetPredication(NULL, 0, D3D12_PREDICATION_OP_EQUAL_ZERO);
m_pd3dCommandList->SetGraphicsRootDescriptorTable(0, pd3dcbvNearQuadHandle);
m_pd3dCommandList->DrawInstanced(4, 1, 4, 0); //앞쪽 사각형
```

```
m_pd3dCommandList->SetGraphicsRootDescriptorTable(0, pd3dcbvFarQuadHandle);
m_pd3dCommandList->SetPipelineState(m_pd3dQueryState);
m_pd3dCommandList->BeginQuery(m_pd3dQueryHeap, D3D12_QUERY_TYPE_BINARY_OCCLUSION, 0);
m_pd3dCommandList->DrawInstanced(4, 1, 8, 0); //뒤쪽 사각형 바운딩 박스
m_pd3dCommandList->EndQuery(m_pd3dQueryHeap, D3D12_QUERY_TYPE_BINARY_OCCLUSION, 0);
```

```
::SynchronizeResourceTransition(m_pd3dCommandList, m_pd3dQueryResult,
D3D12_RESOURCE_STATE_PREDICATION, D3D12_RESOURCE_STATE_COPY_DEST);
m_pd3dCommandList->ResolveQueryData(m_pd3dQueryHeap, D3D12_QUERY_TYPE_BINARY_OCCLUSION,
0, 1, m_pd3dQueryResult, 0);
::SynchronizeResourceTransition(m_pd3dCommandList, m_pd3dQueryResult,
D3D12_RESOURCE_STATE_COPY_DEST, D3D12_RESOURCE_STATE_PREDICATION);
```

Direct3D 파이프라인

- **ID3D12Device::CreateQueryHeap**

- 큐리는 큐리들의 배열(힙)에 저장됨

```
HRESULT ID3D12Device::CreateQueryHeap(  
    D3D12_QUERY_HEAP_DESC *pDesc,  
    REFIID riid, //_uuidof(ID3D12QueryHeap)  
    void **ppvHeap //쿼리 힙: 큐리들의 배열  
)
```

```
typedef struct D3D12_QUERY_HEAP_DESC {  
    D3D12_QUERY_HEAP_TYPE Type; //힙의 유형  
    UINT Count; //저장할 큐리의 개수  
    UINT NodeMask; //0: 단일 GPU  
} D3D12_QUERY_HEAP_DESC;
```

```
typedef struct D3D12_QUERY_DATA_SO_STATISTICS {  
    UINT64 NumPrimitivesWritten;  
    UINT64 PrimitivesStorageNeeded;  
} D3D12_QUERY_DATA_SO_STATISTICS;
```

```
ID3D12QueryHeap* m_pd3dQueryHeap;
```

```
pd3dDevice->CreateQueryHeap(&d3dQueryDesc, &pd3dQuery);  
pd3dCommandList->BeginQuery(m_pd3dQueryHeap, D3D12_QUERY_TYPE_SO_STATISTICS_STREAM0, 0);  
... //Draw  
pd3dCommandList->EndQuery(m_pd3dQueryHeap, D3D12_QUERY_TYPE_SO_STATISTICS_STREAM0, 0);  
pd3dCommandList->ResolveQueryData(m_pd3dQueryHeap, ...);
```

```
typedef enum D3D12_QUERY_HEAP_TYPE {  
    D3D12_QUERY_HEAP_TYPE_OCCLUSION,  
    D3D12_QUERY_HEAP_TYPE_TIMESTAMP,  
    D3D12_QUERY_HEAP_TYPE_PIPELINE_STATISTICS,  
    D3D12_QUERY_HEAP_TYPE_SO_STATISTICS,  
    D3D12_QUERY_HEAP_TYPE_VIDEO_DECODE_STATISTICS,  
    D3D12_QUERY_HEAP_TYPE_COPY_QUEUE_TIMESTAMP  
} D3D12_QUERY_HEAP_TYPE;
```

```
typedef enum D3D12_QUERY_TYPE {  
    D3D12_QUERY_TYPE_OCCLUSION,  
    D3D12_QUERY_TYPE_BINARY_OCCLUSION,  
    D3D12_QUERY_TYPE_TIMESTAMP,  
    D3D12_QUERY_TYPE_PIPELINE_STATISTICS,  
    D3D12_QUERY_TYPE_SO_STATISTICS_STREAM0,  
    D3D12_QUERY_TYPE_SO_STATISTICS_STREAM1,  
    D3D12_QUERY_TYPE_SO_STATISTICS_STREAM2,  
    D3D12_QUERY_TYPE_SO_STATISTICS_STREAM3,  
    D3D12_QUERY_TYPE_VIDEO_DECODE_STATISTICS  
} D3D12_QUERY_TYPE;
```

Direct3D 파이프라인

- **ID3D12GraphicsCommandList::SetPredication**

- GPU가 객체에 대한 명령(Draw, Copy, Dispatch)을 수행하지 않는 것을 결정
바운딩 박스를 그렸는데 차폐가 된다면 객체를 그릴 필요가 없음(Predicated)

```
void ID3D12GraphicsCommandList::SetPredication(  
    ID3D12Resource *pBuffer, //64-비트(8-바이트) 버퍼, NULL(프리디케이션을 비활성화)  
    UINT64 AlignedBufferOffset, //버퍼를 읽을 위치(8-바이트의 배수)  
    D3D12_PREDICATION_OP Operation  
);
```

```
typedef enum D3D12_PREDICATION_OP {  
    D3D12_PREDICATION_OP_EQUAL_ZERO, //버퍼의 모든 비트가 0이면 프리디케이션을 활성화  
    D3D12_PREDICATION_OP_NOT_EQUAL_ZERO //적어도 1 비트가 0이 아니면 프리디케이션을 활성화  
} D3D12_PREDICATION_OP;
```

버퍼의 값이 지정된 연산과 같으면(프리디케이션이 활성화) 이후의 GPU 명령들이 실제로 실행되지 않도록 설정
GPU는 **SetPredication()** 함수를 실행할 때 버퍼(pBuffer)의 값을 내부적으로 저장하고 비교를 수행함
버퍼는 D3D12_RESOURCE_STATE_PREDICATION 상태이어야 함(힙 유형은 어느 것이나 가능)
직접 명령 리스트, 계산 명령 리스트, 복사 명령 리스트에서 사용할 수 있음

DrawInstanced	DrawIndexedInstanced	Dispatch	
CopyTextureRegion	CopyBufferRegion	CopyResource	CopyTiles
ResolveSubresource			
ClearDepthStencilView	ClearRenderTargetView		
ClearUnorderedAccessViewUint	ClearUnorderedAccessViewFloat		
ExecuteIndirect			

Direct3D 파이프라인

- **ID3D12GraphicsCommandList::BeginQuery, EndQuery, ResolveQueryData**
 - 쿼리의 유형은 쿼리 힙의 유형과 일치해야 함

```
void ID3D12GraphicsCommandList::BeginQuery(  
    ID3D12QueryHeap *pQueryHeap, //쿼리 힙  
    D3D12_QUERY_TYPE Type, //쿼리의 유형  
    UINT Index //쿼리를 저장할 쿼리 힙의 위치(인덱스)  
);
```

```
void ID3D12GraphicsCommandList::EndQuery(  
    ID3D12QueryHeap *pQueryHeap, //쿼리 힙  
    D3D12_QUERY_TYPE Type, //쿼리의 유형  
    UINT Index //쿼리 힙의 쿼리 인덱스  
);
```

```
void ID3D12GraphicsCommandList::ResolveQueryData( //쿼리의 결과를 가져옴  
    ID3D12QueryHeap *pQueryHeap, //쿼리 힙  
    D3D12_QUERY_TYPE Type, //쿼리의 유형  
    UINT StartIndex, //쿼리의 결과를 읽을 쿼리 힙의 시작 인덱스  
    UINT NumQueries, //쿼리의 개수(쿼리마다 64 비트를 사용), 배치(Batch) 처리 가능  
    ID3D12Resource *pDestinationBuffer, //쿼리의 결과들을 저장할 버퍼, D3D12_RESOURCE_STATE_COPY_DEST  
    UINT64 AlignedDestinationBufferOffset //쿼리의 결과들을 저장할 버퍼의 위치(오프셋, 8 바이트의 배수)  
);
```

D3D12_QUERY_TYPE_BINARY_OCCLUSION

쿼리 결과의 모든 비트가 0(객체가 완전히 차폐됨, 어떤 샘플(픽셀)도 검사를 통과하지 못함)
최하위 비트가 1(객체가 완전히 차폐되지는 않음, 객체의 일부 샘플(픽셀)이 검사를 통과함)

D3D12_QUERY_TYPE_OCCLUSION

쿼리 결과(64-비트)는 검사를 통과한 샘플(픽셀)의 개수

```
typedef enum D3D12_QUERY_TYPE {  
    D3D12_QUERY_TYPE_OCCLUSION,  
    D3D12_QUERY_TYPE_BINARY_OCCLUSION,  
    D3D12_QUERY_TYPE_TIMESTAMP,  
    D3D12_QUERY_TYPE_PIPELINE_STATISTICS,  
    D3D12_QUERY_TYPE_SO_STATISTICS_STREAM0,  
    D3D12_QUERY_TYPE_SO_STATISTICS_STREAM1,  
    D3D12_QUERY_TYPE_SO_STATISTICS_STREAM2,  
    D3D12_QUERY_TYPE_SO_STATISTICS_STREAM3,  
    D3D12_QUERY_TYPE_VIDEO_DECODE_STATISTICS  
} D3D12_QUERY_TYPE;
```

Direct3D 파이프라인

- **ID3D12GraphicsCommandList::BeginQuery, EndQuery, ResolveQueryData**
 - 쿼리의 유형은 쿼리 힙의 유형과 일치해야 함

```
void ID3D12GraphicsCommandList::ResolveQueryData( //쿼리의 결과를 가져옴
```

```
    ID3D12QueryHeap *pQueryHeap, //쿼리 힙
```

```
    D3D12_QUERY_TYPE Type, //쿼리의 유형
```

```
    UINT StartIndex, //쿼리의 시작 인덱스
```

```
    UINT NumQueries, //쿼리의 개수(쿼리마다 64 비트(8-바이트)를 사용)
```

```
    ID3D12Resource *pDestinationBuffer, //쿼리의 결과를 저장할 버퍼, D3D12_RESOURCE_STATE_COPY_DEST
```

```
    UINT64 AlignedDestinationBufferOffset //버퍼의 오프셋(8 바이트의 배수)
```

```
);
```

```
typedef struct D3D12_QUERY_DATA_PIPELINE_STATIS
```

```
    UINT64 IAVertices;
```

```
    UINT64 IAPrimitives;
```

```
    UINT64 VSInvocations;
```

```
    UINT64 GSInvocations;
```

```
    UINT64 GSPrimitives;
```

```
    UINT64 CInvocations;
```

```
    UINT64 CPrimitives;
```

```
    UINT64 PSInvocations;
```

```
    UINT64 HSInvocations;
```

```
    UINT64 DSInvocations;
```

```
    UINT64 CSInvocations;
```

```
} D3D12_QUERY_DATA_PIPELINE_STATISTICS;
```

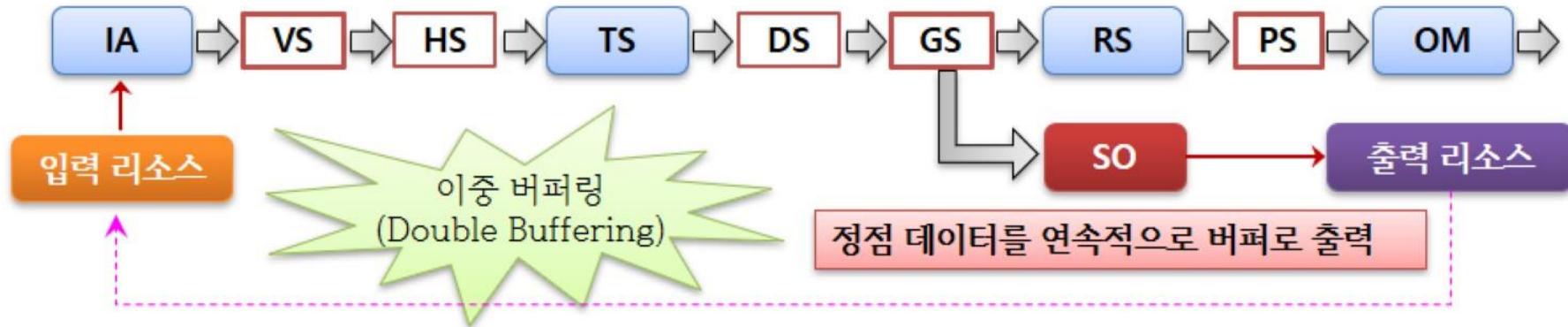
```
typedef enum D3D12_QUERY_TYPE {  
    D3D12_QUERY_TYPE_OCCLUSION,  
    D3D12_QUERY_TYPE_BINARY_OCCLUSION,  
    D3D12_QUERY_TYPE_TIMESTAMP,  
    D3D12_QUERY_TYPE_PIPELINE_STATISTICS,  
    D3D12_QUERY_TYPE_SO_STATISTICS_STREAM0,  
    D3D12_QUERY_TYPE_SO_STATISTICS_STREAM1,  
    D3D12_QUERY_TYPE_SO_STATISTICS_STREAM2,  
    D3D12_QUERY_TYPE_SO_STATISTICS_STREAM3,  
    D3D12_QUERY_TYPE_VIDEO_DECODE_STATISTICS  
} D3D12_QUERY_TYPE;
```

```
typedef struct D3D12_QUERY_DATA_SO_STATISTICS {  
    UINT64 NumPrimitivesWritten;  
    UINT64 PrimitivesStorageNeeded;  
} D3D12_QUERY_DATA_SO_STATISTICS;
```

Direct3D 파이프라인

• GPU 기반 파티클(Particle) 시스템

- 파티클은 빌보드 객체로 렌더링할 수 있음
파티클 객체는 하나의 점(Point)로 표현할 수 있음
기하 쉐이더에서 파티클을 빌보드 사각형으로 렌더링할 수 있음
파티클의 생성/소멸, 위치/방향을 처리해야 함(CPU에서 처리하는 경우 문제는?)
- GPU에서 파티클의 생성/소멸, 위치/방향을 처리하는 기하 쉐이더가 필요
변경된 정보를 사용하여 렌더링하는 기하 쉐이더가 필요
3개의 정점 버퍼가 필요(초기 정점 버퍼, 스트림 출력 정점 버퍼, 렌더링 정점 버퍼)



기하 쉐이더가 생성한 기하 데이터를 렌더링

스트림 출력된 버퍼(스트림 출력 단계에 연결)를 렌더링하기 전에 입력 조립 단계에 연결해야 함

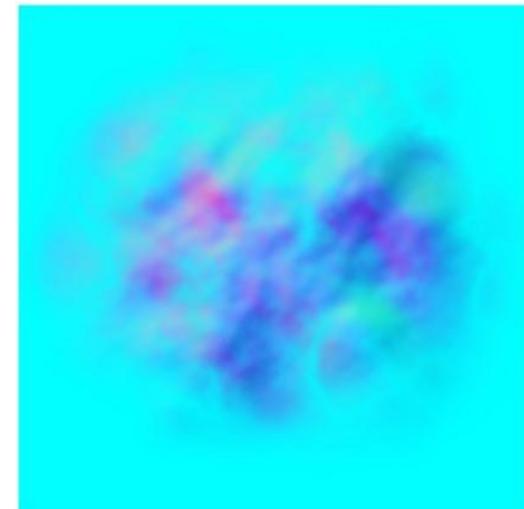
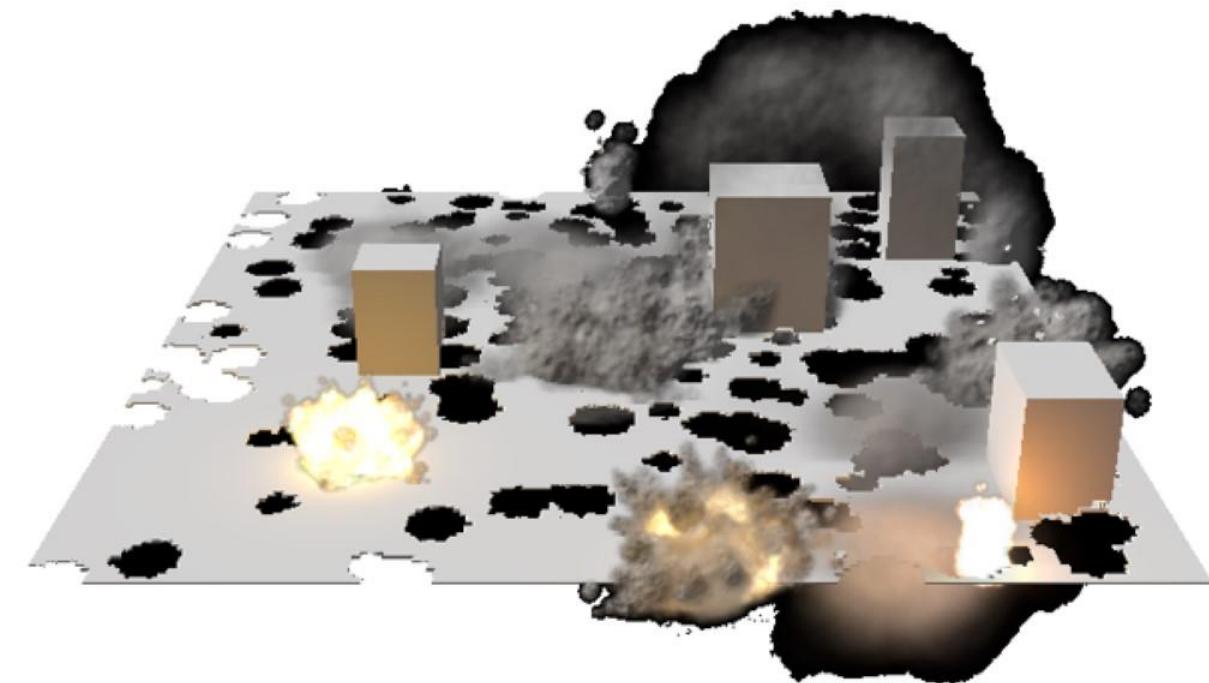
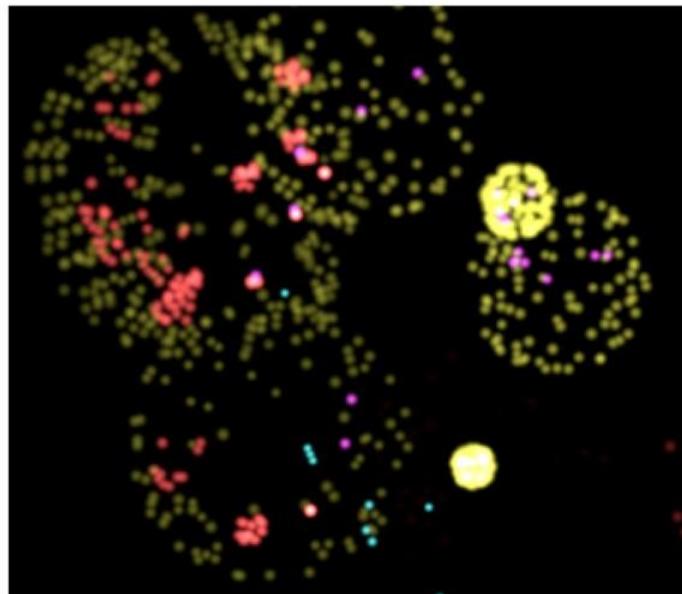
(스트림 출력 버퍼는 동시에 입력 조립 단계에 연결될 수 없으므로 SO 단계에는 다른 버퍼(또는 NULL)를 연결)

스트림 출력된 정점의 구조(Input Layout)는 입력 조립 단계에 설정해야 함

파티클 시스템(Particle System)

- GPU 파티클 시스템(Particle System)

- 파티클 시스템
파티클의 생성, 간신, 소멸의 관리와 렌더링
CPU 파티클 시스템 vs. GPU 파티클 시스템
- 이미터(Emitter) 파티클
다른 파티클을 생성 또는 방출(Emit)
쉘(Shell) 파티클 - 엠버(Ember) 파티클



Direct3D 파이프라인

- 스트림 출력 버퍼 연결

```
D3D12_STREAM_OUTPUT_BUFFER_VIEW m_d3dStreamOutputBufferView;
```

```
ID3D12Resource* m_pd3dStreamOutputBuffer;  
ID3D12Resource* m_pd3dDrawBuffer;
```

```
ID3D12Resource* m_pd3dBufferFilledSize;  
ID3D12Resource* m_pd3dUploadBufferFilledSize;  
ID3D12Resource* m_pd3dReadBackBufferFilledSize;  
UINT64* m_pnUploadBufferFilledSize;  
UINT64* m_pnReadBackBufferFilledSize;
```

```
m_pd3dStreamOutputBuffer = ::CreateBufferResource(pd3dDevice, pd3dCommandList, NULL, (m_nStride *  
m_nMaxParticles), D3D12_HEAP_TYPE_DEFAULT, D3D12_RESOURCE_STATE_STREAM_OUT, NULL);  
m_pd3dDrawBuffer = ::CreateBufferResource(pd3dDevice, pd3dCommandList, NULL, (m_nStride *  
m_nMaxParticles), D3D12_HEAP_TYPE_DEFAULT, D3D12_RESOURCE_STATE_VERTEX_AND_CONSTANT_BUFFER,  
NULL);
```

```
UINT64 nBufferFilledSize = 0;  
m_pd3dBufferFilledSize = ::CreateBufferResource(pd3dDevice, pd3dCommandList, &nBufferFilledSize,  
sizeof(UINT64), D3D12_HEAP_TYPE_DEFAULT, D3D12_RESOURCE_STATE_STREAM_OUT, NULL);  
m_pd3dUploadBufferFilledSize = ::CreateBufferResource(pd3dDevice, pd3dCommandList, NULL,  
sizeof(UINT64), D3D12_HEAP_TYPE_UPLOAD, D3D12_RESOURCE_STATE_GENERIC_READ, NULL);  
m_pd3dUploadBufferFilledSize->Map(0, NULL, (void**)&m_pnUploadBufferFilledSize);  
m_pd3dReadBackBufferFilledSize = ::CreateBufferResource(pd3dDevice, pd3dCommandList, NULL,  
sizeof(UINT64), D3D12_HEAP_TYPE_READBACK, D3D12_RESOURCE_STATE_COPY_DEST, NULL);  
m_pd3dReadBackBufferFilledSize->Map(0, NULL, (void**)&m_pnReadBackBufferFilledSize);
```

```
m_d3dStreamOutputBufferView.BufferLocation = m_pd3dStreamOutputBuffer->GetGPUVirtualAddress();  
m_d3dStreamOutputBufferView.SizeInBytes = m_nStride * m_nMaxParticles;  
m_d3dStreamOutputBufferView.BufferFilledSizeLocation = m_pd3dBufferFilledSize->GetGPUVirtualAddress();
```

파티클 시스템(Particle System)

- GPU 파티클 시스템(Particle System)

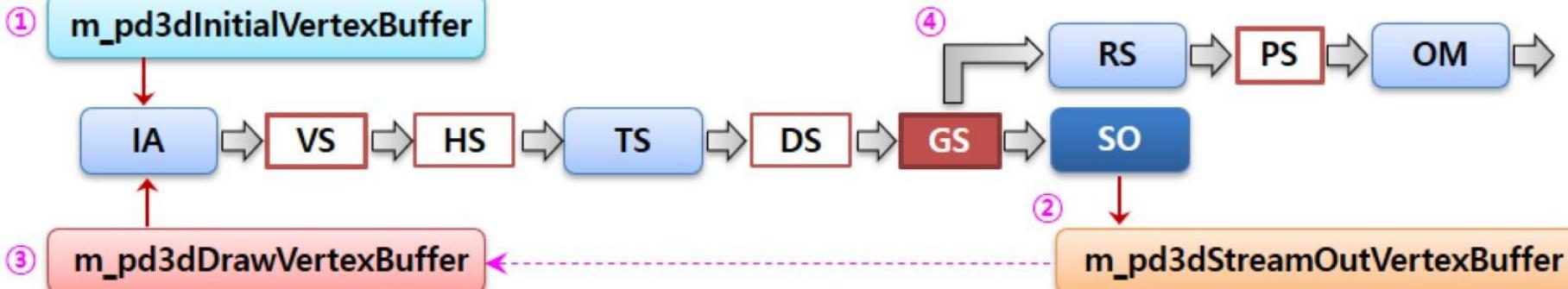
- 파티클 정점 버퍼의 생성

```
class CParticleVertex
{
    XMFLOAT3 m_xmf3Position; //파티클의 위치
    XMFLOAT3 m_xmf3Velocity; //파티클의 속도
    XMFLOAT3 m_xmf3Acceleration; //파티클의 가속도
    XMFLOAT2 m_xmf2Size; //파티클의 크기
    float m_fAge; //파티클의 나이
    float m_fLifeTime; //파티클의 수명
    UINT m_nType; //파티클의 유형
};
```

$$\mathbf{p}(t) = \mathbf{p}_0 + \frac{1}{2} \mathbf{a} t^2 + \mathbf{v} t$$

```
#define PARTICLE_TYPE_EMITTER 0
#define MAX_PARTICLES 9999999
```

```
ID3D12Resource *m_pd3dInitialVertexBuffer;
ID3D12Resource *m_pd3dStreamOutVertexBuffer;
ID3D12Resource *m_pd3dDrawVertexBuffer;
```



파티클 시스템(Particle System)

- GPU 파티클 시스템(Particle System)

- 파티클 쉐이더

```
[maxvertexcount(2)]
```

```
void GSParticleStreamOut(point VS_INPUT input[1], inout PointStream<VS_INPUT> pointStream)
{
    input[0].age += gfElapsedTime;
    if (input[0].type == PARTICLE_TYPE_EMITTER) {
        if (input[0].age > 0.005f) {
            float3 vRandom = gtxRandomTexture.SampleLevel(gssParticle, input[0].age, 0).xyz;
            vRandom = normalize(vRandom);
            vRandom.x *= 0.5f;
            vRandom.z *= 0.5f;
            VS_INPUT particle;
            particle.position = gvPosition.xyz;
            particle.velocity = 4.0f * vRandom;
            particle.size = float2(3.0f, 3.0f);
            particle.age = 0.0f;
            particle.type = 1;
            pointStream.Append(particle);
            input[0].age = 0.0f;
        }
        pointStream.Append(input[0]);
    }
    else
        if (input[0].age <= 1.0f) pointStream.Append(input[0]);
}
```

```
Texture1D gtxRandomTexture : register(t5);
SamplerState gssParticle : register(s5);
```

```
#define PARTICLE_TYPE_EMITTER 0
```

```
struct VS_INPUT
{
    float3 position : POSITION;
    float3 velocity : VELOCITY;
    float2 size : SIZE;
    uint type : TYPE;
    float age : AGE;
};
```

```
VS_INPUT VSParticleStreamOut(VS_INPUT input)
{
    return(input);
}
```

파티클 시스템(Particle System)

- GPU 파티클 시스템(Particle System)

- 파티클 쉐이더

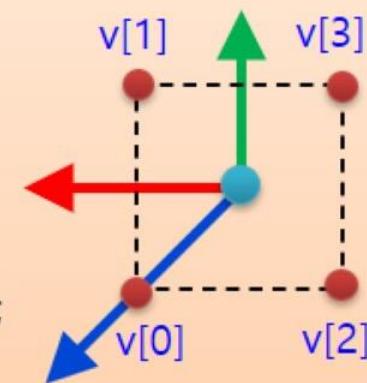
```
[maxvertexcount(4)]
```

```
void GSParticleDraw(point VS_OUTPUT input[1], inout TriangleStream<GS_OUTPUT> triStream) {
    if (input[0].type == PARTICLE_TYPE_EMITTER) return;
    float3 vLook = normalize(gvCameraPosition.xyz - input[0].position);
    float3 vRight = normalize(cross(float3(0.0f, 1.0f, 0.0f), vLook));
    float3 vUp = cross(vLook, vRight);

    float fHalfWidth = 0.5f * input[0].size.x, fHalfHeight = 0.5f * input[0].size.y;
    float4 vQuads[4];
    vQuads[0] = float4(input[0].position + fHalfWidth * vRight - fHalfHeight * vUp, 1.0f);
    vQuads[1] = float4(input[0].position + fHalfWidth * vRight + fHalfHeight * vUp, 1.0f);
    vQuads[2] = float4(input[0].position - fHalfWidth * vRight - fHalfHeight * vUp, 1.0f);
    vQuads[3] = float4(input[0].position - fHalfWidth * vRight + fHalfHeight * vUp, 1.0f);
    matrix mtxViewProjection = mul(gmtxView, gmtxProjection);
```

```
    GS_OUTPUT output;
    for (int i = 0; i < 4; i++) {
        output.position = mul(vQuads[i], mtxViewProjection);
        output.uv = gvQuadTexCoord[i];
        output.color = input[0].color;
        triStream.Append(output);
    }
}
```

```
static float2 gvQuadTexCoord[4] = { float2(0.0f, 1.0f), float2(0.0f, 1.0f), float2(1.0f, 1.0f), float2(1.0f, 0.0f) };
```



```
struct GS_OUTPUT
{
    float4 position : SV_Position;
    float4 color : COLOR;
    float2 uv : TEXCOORD;
};
```

파티클 시스템(Particle System)

- GPU 파티클 시스템(Particle System)

- 파티클 쉐이더

```
VS_OUTPUT VSParticleDraw(VS_INPUT input)
{
    VS_OUTPUT output;

    float t = input.age;
    output.position = (0.5f * gvAcceleration * t * t) + (input.velocity * t) + input.position;

    float fOpacity = 1.0f - smoothstep(0.0f, 1.0f, t);
    output.color = float4(1.0f, 1.0f, 1.0f, fOpacity);

    output.size = input.size;
    output.type = input.type;

    return(output);
}
```

```
Texture2DArray gtxtParticleTextureArray : register(t6);
Texture1D gtxtRandomTexture : register(t5);
SamplerState gssParticle : register(s5);

#define PARTICLE_TYPE_EMITTER 0
```

```
cbuffer cbParticleInfo : register(cb2) {
    float3 gvPosition;
    float gfAge; //gfGameTime
    float3 gvDirection;
    float gfElapsedTime;
    float3 gvAcceleration;
};
```

```
struct VS_INPUT {
    float3 position : POSITION;
    float3 velocity : VELOCITY;
    float2 size : SIZE;
    uint type : TYPE;
    float age : AGE;
};
```

```
struct VS_OUTPUT {
    float3 position : POSITION;
    float2 size : SIZE;
    float4 color : COLOR;
    uint type : TYPE;
};
```

```
float4 PSParticleDraw(GS_OUTPUT input) : SV_TARGET
{
    float4 cColor = gtxtParticleTextureArray.Sample(gssParticle, float3(input.uv, 0));
    return(cColor * input.color);
}
```

파티클 시스템(Particle System)

• GPU 파티클 시스템(Particle System)

```
VS_INPUT VSParticleStreamOut(VS_INPUT input) { return(input); }
```

```
[maxvertexcount(4)]
```

```
void GSParticleStreamOut(point VS_INPUT input[1], inout PointStream<VS_INPUT> pStream)
```

```
{
```

```
    input[0].age += gfElapsedTime * 4.0f;
```

```
    if (input[0].type == PARTICLE_TYPE_EMITTER) {
```

```
        if (input[0].age > 0.005f) {
```

```
            float3 vRandom = gtxtRandomTexture.SampleLevel(gssParticle, gfAge, 0).xyz;
```

```
            vRandom = normalize(vRandom);
```

```
            vRandom.x *= 0.5f;
```

```
            vRandom.z *= 0.5f;
```

```
            VS_INPUT particle;
```

```
            particle.position = gvCameraPosition.xyz + (gvCameraLook.xyz * 35.0f);
```

```
            particle.velocity = 2.0f * vRandom + gvCameraUp.xyz; // (gmtxView._12, gmtxView._22, gmtxView._32)
```

```
            particle.size = float2(16.0f, 8.0f);
```

```
            particle.age = 0.0f;
```

```
            particle.type = 1;
```

```
            pStream.Append(particle);
```

```
            input[0].age = 0.0f;
```

```
}
```

```
            pStream.Append(input[0]);
```

```
}
```

```
else
```

```
    if (input[0].age <= 1.0f) pStream.Append(input[0]);
```

```
#define PARTICLE_TYPE_EMITTER 0
```

```
Texture1D gtxtRandomTexture : register(t5);  
SamplerState gssParticle : register(s5);
```

```
struct VS_INPUT  
{  
    float3 position : POSITION;  
    float3 velocity : VELOCITY;  
    float2 size : SIZE;  
    uint type : TYPE;  
    float age : AGE;  
};
```

파티클 시스템(Particle System)

- GPU 파티클 시스템(Particle System)

```
VS_OUTPUT VSParticleDraw(VS_INPUT input)
```

```
{  
    VS_OUTPUT output;  
    float t = input.age;  
    output.position = (0.5f * gvAcceleration.xyz * t * t) + (input.velocity * t) + input.position;  
    float fOpacity = 1.0f - smoothstep(0.0f, 1.0f, t) * 0.5f;  
    output.color = float4(1.0f, 1.0f, 1.0f, fOpacity);  
    output.size = input.size;  
    output.type = input.type;  
  
    return(output);  
}
```

```
struct VS_OUTPUT  
{  
    float3 position : POSITION;  
    float2 size : SIZE;  
    float4 color : COLOR;  
    uint type : TYPE;  
};
```

```
Texture2DArray gtxtParticleTextureArray : register(t6);  
SamplerState gssParticle : register(s5);
```

```
float4 PSParticleDraw(GS_OUTPUT input) : SV_TARGET  
{  
    float4 cColor = gtxtParticleTextureArray.Sample(gssParticleSamplerState, float3(input.uv, 0));  
    cColor = cColor * input.color;  
  
    return(cColor);  
}
```

```
cbuffer cbParticleInfo : register(cb1) {  
    float3 gvPosition : packoffset(c0);  
    float gfAge : packoffset(c0.w);  
    float3 gvDirection : packoffset(c1);  
    float gfElapsedTime : packoffset(c1.w);  
    float4 gvAcceleration : packoffset(c2);  
};
```

파티클 시스템(Particle System)

• GPU 파티클 시스템(Particle System)

[maxvertexcount(4)]

```
void GSParticleDraw(point VS_OUTPUT input[1], inout TriangleStream<GS_PARTICLE_OUTPUT> triStream)
```

```
{
```

```
    if (input[0].type != PARTICLE_TYPE_EMITTER) {
        float3 vLook = normalize(gvCameraPosition.xyz - input[0].position);
        float3 vRight = normalize(cross(float3(0.0f, 1.0f, 0.0f), vLook));
        float3 vUp = cross(vLook, vRight);
        float fHalfWidth = 0.5f * input[0].size.x;
        float fHalfHeight = 0.5f * input[0].size.y;
        float4 vQuads[4];
        vQuads[0] = float4(input[0].position + fHalfWidth * vRight - fHalfHeight * vUp, 1.0f);
        vQuads[1] = float4(input[0].position + fHalfWidth * vRight + fHalfHeight * vUp, 1.0f);
        vQuads[2] = float4(input[0].position - fHalfWidth * vRight - fHalfHeight * vUp, 1.0f);
        vQuads[3] = float4(input[0].position - fHalfWidth * vRight + fHalfHeight * vUp, 1.0f);
        matrix mtxViewProjection = mul(gmtxView, gmtxProjection);
        float2 vuvQuads[4] = { float2(0.0f, 1.0f), float2(0.0f, 0.0f), float2(1.0f, 1.0f), float2(1.0f, 0.0f) };
        GS_OUTPUT output;
        for (int i = 0; i < 4; i++) {
            output = (GS_OUTPUT)0;
            output.position = mul(vQuads[i], mtxViewProjection);
            output.uv = vuvQuads[i].xy;
            output.color = input[0].color;
            triStream.Append(output);
        }
    }
}
```

```
struct VS_OUTPUT {
    float3 position : POSITION;
    float2 size : SIZE;
    float4 color : COLOR;
    uint type : TYPE;
};
```

```
struct GS_OUTPUT {
    float4 position : SV_Position;
    float4 color : COLOR;
    float2 uv : TEXCOORD;
};
```

파티클 시스템(Particle System)

- GPU 파티클 시스템(Particle System)

- 파티클 쉐이더(비: Rain)

[maxvertexcount(6)]

```
void GSParticleStreamOut(point VS_INPUT input[1], inout PointStream<VS_INPUT> pointStream)
```

{

```
    input[0].age += gfElapsedTime;
```

```
    if (input[0].type == PARTICLE_TYPE_EMITTER) {
```

```
        if (input[0].age > 0.002f) {
```

```
            for (int i = 0; i < 5; i++) {
```

```
                float3 vRandom = gtxtRandomTexture.SampleLevel(gssParticle, gfAge, 0).xyz;
```

```
                vRandom = normalize(vRandom) * 50.0f;
```

```
                vRandom.y = 20.5f;
```

```
                VS_INPUT particle;
```

```
                particle.position = gvPosition.xyz + vRandom;
```

```
                particle.velocity = float3(0.0f, 0.0f, 0.0f);
```

```
                particle.size = float2(1.0f, 1.0f);
```

```
                particle.age = 0.0f;
```

```
                particle.type = 1;
```

```
                pointStream.Append(particle);
```

```
}
```

```
    input[0].age = 0.0f;
```

```
}
```

```
    pointStream.Append(input[0]);
```

```
}
```

```
else
```

```
    if (input[0].age <= 4.0f) pointStream.Append(input[0]);
```

```
Texture1D gtxtRandomTexture : register(t5);  
SamplerState gssParticle : register(s5);
```

```
#define PARTICLE_TYPE_EMITTER 0
```

```
struct PARTICLE_INPUT {  
    float3 position : POSITION;  
    float3 velocity : VELOCITY;  
    float2 size : SIZE;  
    uint type : TYPE;  
    float age : AGE;  
};
```

```
VS_INPUT VSParticleStreamOut(VS_INPUT input)
```

```
{  
    return(input);  
}
```

파티클 시스템(Particle System)

- GPU 파티클 시스템(Particle System)

- 파티클 쉐이더(비: Rain)

```
VS_OUTPUT VSParticleDraw(VS_INPUT input) {
    VS_OUTPUT output;
    output.type = input.type;
    float t = input.age;
    output.position = (0.5f * gvAcceleration * t * t) + (input.velocity * t) + input.position;
    return(output);
}
```

```
struct VS_INPUT {
    float3 position : POSITION;
    float3 velocity : VELOCITY;
    uint type : TYPE;
    float age : AGE;
};
```

```
[maxvertexcount(2)]
```

```
void GSParticleDraw(point VS_OUTPUT input[1], inout LineStream<GS_OUTPUT> lineStream) {
    if (input[0].type == PARTICLE_TYPE_EMITTER) return;
    matrix mtxViewProjection = mul(gmtxView, gmtxProjection);
    GS_OUTPUT output;
    output.position = mul(float4(input[0].position, 1.0f), mtxViewProjection);
    output.uv = float2(0.0f, 0.0f);
    lineStream.Append(output);
    output.position = mul(float4(input[0].position + gvAcceleration * 0.07f, 1.0f), mtxViewProjection);
    output.uv = float2(1.0f, 1.0f);
    lineStream.Append(output);
}
```

```
struct VS_OUTPUT {
    float3 position : POSITION;
    uint type : TYPE;
};
```

```
float4 PSParticleDraw(GS_OUTPUT input) : SV_TARGET {
    return(gtxtParticleTextureArray.Sample(gssParticle, float3(input.uv, 0)));
}
```

```
struct GS_OUTPUT {
    float4 position : SV_Position;
    float2 uv : TEXCOORD;
};
```