DirectX12 구현 내용 설명

조작법

- 이동 : 방향키
- 총알 발사 : 스페이스바
- 씬 전환 : I(i)
- 지형 출력 방식 전환 : D(d)

구현 내용

총알

- 박스와 충돌하면 폭발 효과가 나타납니다.

빌보드

- 비트맵에서 정보를 읽어 빌보드의 위치와 종류를 정했습니다.

스카이 박스

- 큐브 맵을 사용하였습니다.

파티클

- 스트림 출력을 이용해 파티클을 구현하였습니다.

지형

- 카메라와 거리에 따라 지형 정점의 수가 변합니다.
  - D를 누르면 와이어 프레임으로 렌더링 됩니다
- 플레이어가 지형 위를 따라 움직입니다.
- 두 개의 uv좌표를 이용해 자연스러운 지형 매핑을 하였습니다.
- 여러 개의 텍스처를 이용해 도로, 빌보드의 종류를 정하였습니다.

그림자

- 쉐도우 맵을 생성하고 그림자를 렌더링하였습니다.
- I를 누르면 그림자가 있는 씬으로 전환됩니다.
- 쉐도우 맵을 렌더링하는 스레드와 씬을 렌더링하는 스레드를 분리하였습니다.

블러링

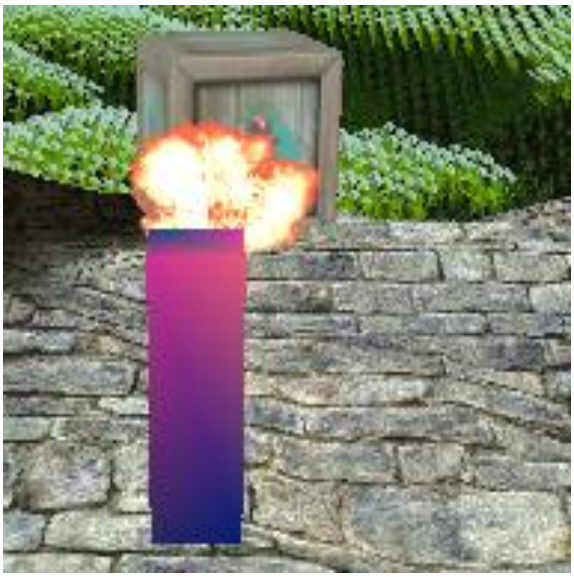
- 계산 셰이더를 이용해 포스트 프로세싱을 하였습니다.

실행결과

폭발 효과

- 목표 -> 스프라이트 텍스쳐를 이용해 폭발 효과를 나타내보자
- 구현

  시간에 따라 스프라이트 텍스쳐를 자르기 위해 폭발 효과를 그리는 게임 오브젝트의 상수 버퍼에 에니메이션 시간을 추가했습니다. 충돌한 순간부터 흘러가는 시간에 따라 스프라이트의 uv좌표를 바꾸어 가며 빌보드에 매핑할 수 있도록 하였고 스프라이트를 다 사용하였다면 효과가 끝나도록 하였습니다. 빌보드 오브젝트이기 때문에 항상 플레이어를 바라보고 있습니다.



빌보드

- 목표 ->인스턴싱을 사용해 오버헤드를 줄이자, 블렌딩을 사용해 투명한 부분을 표시하자, 8비트 비트맵에서 색을 읽어 빌보드의 종류와 위치를 정하자
- 구현

  블렌딩을 활성화 해주고(출력 병합단계에서 합니다. 최종 색상을 픽셀 셰이더에서 텍스쳐를 두 개 이상 사용해 최종색상을 결정하는 단일 패스 렌더링과는 다르게 이미 그려진 렌더 타겟에 픽셀셰이더의 출력 색상을 섞어 색상을 결정하는 다중 패스 렌더링을 사용합니다.) D3D12_BLEND_SRC_ALPHA, D3D12_BLEND_INV_SRC_ALPHA명령어를 이용해 알파 값을 0이라면 투명한 것으로 하고 투명도를 계산해 색상을 혼합할 수 있도록 합니다. 이때 빌보드의 투명한 부분은 유리 창과는 다르게 완전 투명하기 때문에 그리는 순서를 신경 쓸 필요가 없습니다. 또 AlphaToCoverage를 Enable시켜 샘플링 작업 중 커버리지(픽셀의 포함여부)를 계산할 때 알파 값을 사용해 계산할 수 있도록 합니다.

  인스턴싱을 사용하기 위해 인스턴스별 정보(위치, 텍스처정보)를 버퍼에 담아서 사용하였습니다. 그리고 셰이더에서 위치를 이용해 플레이어 카메라를 바라보게 하는 월드 행렬을 생성해 빌보드 오브젝트들이 플레이어 카메라를 바라보게 만들었습니다..

```
d3dBlendDesc.RenderTarget[0].BlendEnable = TRUE;
d3dBlendDesc.RenderTarget[0].LogicOpEnable = FALSE;
d3dBlendDesc.RenderTarget[0].SrcBlend = D3D12_BLEND_SRC_ALPHA;
d3dBlendDesc.RenderTarget[0].DestBlend = D3D12_BLEND_INV_SRC_ALPHA;
d3dBlendDesc.RenderTarget[0].BlendOp = D3D12_BLEND_OP_ADD;
```

스카이 박스

- 목표 -> 텍스쳐를 큐브 맵을 사용하고, 깊이 검사와 z값을 이용해 오버 드로우를 줄이자
- 구현

  플레이어와 위치를 따라다니며 텍스쳐가 잘 그려질 적절한 크기 정육면체를 만들고 정육면체의 안쪽면에 스카이 박스 텍스쳐를 매핑하고 정육면체와의 깊이 검사 때문에 안 그려질 씬의 오브젝트를 그리기 위해 원근 나눗셈을 할 때 z값이 절두체 맨 뒤에 있는 z값이 될 수 있도록 z값을 w로 만든 후 변환을 하였습니다. 깊이버퍼랑 깊이 z값이 작거나 같으면 그리는 D3D12_COMPARISON_FUNC_LESS_EQUAL명령어를 줘 스카이 박스 오브젝트가 씬의 맨 뒤에 그려질 수 있도록 하였습니다.

```
output.positionH = mul(mul(mul(float4(input.position, 1.0f), gmtxGameObject), gmtxView), gmtxProjection).xyww;
```

  마이크로소프트에서 제공하는 Texassemble을 이용해 위치 벡터를 이용해 큐브 면과 텍스쳐 좌표를 계산하는 큐브 맵을 생성하여 사용하였습니다.

지형

- 목표 -> 텍스쳐를 이용해 원하는 부분에 도로를 그려보자, 지형 정보를 나타내는 비트맵이용해 빌보드의 위치, 종류를 나타내 보자
- 구현

  빌보드 위치를 표현한 비트맵 파일에 도로의 색상도 추가해 해당 부분에는 빌보드가 위치할 수 없도록 하였고 해당 비트맵 파일을 dds로 변환하여 디테일 텍스쳐로 이용하여 도로를 표현하였습니다.

파티클

- 목표
  - 파이프라인단계 중 스트림 출력 단계를 이용해 동적 파티클을 생성해보자
- 구현
  - CGSParticleShader::CreateShader에서 버퍼에 저장하는 단계(Pass1)를 위한 PSO와 버퍼의 정점 정보를 그리는 단계를(Pass2) 위한 PSO를 만들어 주었습니다.
  - Pass1에서는 기하셰이더를 통과해 변화한 정점을 스트림 출력단계를 통해 버퍼에 쓰쓰는 것 목적이기 때문에 렌더링이 필요가 없는 단계입니다. 그래서 픽셀 셰이더 단계를 없애 깊이 스텐실 검사, 블렌딩, 레스터라이져 단계의 클리핑 등을 하지 않도록 해주었습니다. 그런 다음 D3D12_STREAM_OUTPUT_DESC의 값을 채워 PSO를 만들어 주었습니다.
  - Pass2에서는 정점을 그리기 위해 pass1과 다른 셰이더 함수를 사용하였고, 깊이 스텐실 검사, 픽셀 셰이더, 블렌딩 등을 다시 사용하도록 PSO를 만들었습니다.
  - 기하셰이더에서 생성한 정점을 저장하고 그리기 위한 버퍼를 CParticle이라는 클래스에 만들었습니다. CParticle은 CParticleVertex라는 정점을 가지는 버퍼를 3개 생성합니다. Start버퍼는 처음 파티클이 생성될 때를 위한 버퍼입니다. 파티클 정보를 가지고 있는 정점을 하나 가지고 있어 처음 파티클 이펙트가 시작될 때 사용됩니다. Draw버퍼는 스트림 출력단계에서 출력된 정점을 저장하고 그리기 위한 버퍼입니다. Stream버퍼가 스트림 출력단계를 통과한 후 Stream 버퍼의 내용을 Draw버퍼에 복사해 pass2에서 사용합니다. Stream 버퍼는 스트림 출력단계에서 나온 버퍼를 저장하기 위한 버퍼입니다. 스트림 출력단계에서 append된 정점의 수를 알기 위해 FilledSize라는 리소스를 사용해 BufferFilledSizeLocation에 연결해 주었습니다 해당 리소스를 읽고 쓰기 위해 read용 리소스와 upload용 리소스를 사용하였습니다.
  - 스트림 출력단계에서 출력된 정점의 수를 알기 위해 GS단계에서 호출된 Append횟수를 알려주는 리소스(FilledSize)를 만들고 FilledSize를 읽기위한 리소스와 FilledSize에 쓰기 위한 리소스를 만들어 한 프레임마다 출력된 정점의 수에 맞춰 Draw를 호출하고 다시 0부터 쓰기 시작하도록 해 주었습니다.
  - 파티클 타이머 정보를 이용해 파티클의 알파 값을 시간에 따라 변경해 서서히 희미해지게 만들었습니다.

```cpp
void CGSParticleShader::Render(ID3D12GraphicsCommandList* pd3dCommandList, CCamera* pCamera)
{
    if (m_ppd3dPipelineStates)pd3dCommandList->SetPipelineState(m_ppd3dPipelineStates[0]);
    //pass 1
    //so 를 사용해 입자 계산결과를 버퍼에 저장
    ID3D12Resource* Draw = pParticlebuf->m_pd3dDrawBuffer;
    ID3D12Resource* stream = pParticlebuf->m_pd3dStreamBuffer;

    D3D12_VERTEX_BUFFER_VIEW v;
    if (bFirst) {// -> 다시 실행 되는 경우 런처 한 개 부터 시작해야한다.
        v = pParticlebuf->GetStartBufferView();
        bFirst = false;
    }
    else
        v = pParticlebuf->GetDrawBufferView();

    pd3dCommandList->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_POINTLIST);
    pd3dCommandList->IASetVertexBuffers(0, 1, &v);

    D3D12_STREAM_OUTPUT_BUFFER_VIEW SOBUFVIEW = pParticlebuf->m_d3dStreamBufferView;

    pd3dCommandList->SOSetTargets(0, 1, &SOBUFVIEW);
    pd3dCommandList->DrawInstanced(static_cast<UINT>((*pParticlebuf->m_pnReadBackBufferFilledSize) / stride), 1, 0, 0);
    //카운터에 갯수를 적었을것
    pd3dCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(pParticlebuf->m_pd3dBufferFilledSize, D3D12_RESOURCE_STATE_STREAM_OUT, D3D12_RESOURCE_STATE_COPY_SOURCE));
    pd3dCommandList->CopyResource(pParticlebuf->m_pd3dReadBackBufferFilledSize, pParticlebuf->m_pd3dBufferFilledSize);
    pd3dCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(pParticlebuf->m_pd3dBufferFilledSize, D3D12_RESOURCE_STATE_COPY_SOURCE, D3D12_RESOURCE_STATE_STREAM_OUT));

    *pParticlebuf->m_pnUploadBufferFilledSize = 0;
    pd3dCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(pParticlebuf->m_pd3dBufferFilledSize, D3D12_RESOURCE_STATE_STREAM_OUT, D3D12_RESOURCE_STATE_COPY_DEST));
    pd3dCommandList->CopyResource(pParticlebuf->m_pd3dBufferFilledSize, pParticlebuf->m_pd3dUploadBufferFilledSize);
    pd3dCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(pParticlebuf->m_pd3dBufferFilledSize, D3D12_RESOURCE_STATE_COPY_DEST, D3D12_RESOURCE_STATE_STREAM_OUT));

    //pass2
    //stream out된 버퍼를 이용해 IA단계부터 렌더링 시작
```

```cpp
    //pass2
    //stream out된 버퍼를 이용해 IA단계부터 렌더링 시작

    //다음 프레임을 위해 Draw버퍼에 기하쉐이더가 만든 정점을 저장함
    {
        pd3dCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(Draw, D3D12_RESOURCE_STATE_VERTEX_AND_CONSTANT_BUFFER, D3D12_RESOURCE_STATE_COPY_DEST));
        pd3dCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(stream, D3D12_RESOURCE_STATE_STREAM_OUT, D3D12_RESOURCE_STATE_COPY_SOURCE));
        pd3dCommandList->CopyResource(Draw, stream);
        pd3dCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(stream, D3D12_RESOURCE_STATE_COPY_SOURCE, D3D12_RESOURCE_STATE_STREAM_OUT));
        pd3dCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(Draw, D3D12_RESOURCE_STATE_COPY_DEST, D3D12_RESOURCE_STATE_VERTEX_AND_CONSTANT_BUFFER));
    }
    //만든 뷰를 이용해 다시 입력조립단계에 묶는다. SOSetTarget이 IA단계에 streamout버퍼를 묶어 준다 가정

    if (m_ppd3dPipelineStates)pd3dCommandList->SetPipelineState(m_ppd3dPipelineStates[1]);
    pd3dCommandList->SetDescriptorHeaps(1, &m_pd3dCbvSrvDescriptorHeap);
    if (m_pBillboardMaterial) m_pBillboardMaterial->UpdateShaderVariables(pd3dCommandList);

    pd3dCommandList->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_POINTLIST);
    pd3dCommandList->IASetVertexBuffers(0, 1, &v);
    pd3dCommandList->DrawInstanced(static_cast<UINT>((*pParticlebuf->m_pnReadBackBufferFilledSize) / stride), 1, 0, 0);
}
```

파티클을 동적으로 생성해 렌더링하는 코드 Shader.cpp

```cpp
[maxvertexcount(100)]
void GSAdvanceParticles(point VS_PARTICAL_INPUT input[1], inout PointStream<VS_PARTICAL_INPUT> ParticleOutputStream)
{
    if (input[0].Type == PT_LAUNCHER)
        GSLauncherHandler(input[0], ParticleOutputStream);
    else if (input[0].Type == PT_SHELL)
        GSShellHandler(input[0], ParticleOutputStream);
    else if (input[0].Type == PT_EMBER1 ||
             input[0].Type == PT_EMBER3 || input[0].Type == PT_EMBER2)
        GSEmber1Handler(input[0], ParticleOutputStream);
}

[maxvertexcount(4)]
void GSDrawParticles(point VS_PARTICAL_OUTPUT input[1], inout TriangleStream<GS_PARTICAL_OUTPUT> SpriteStream)
{
    float3 vUp = float3(0.0f, 1.0f, 0.0f);
    float3 vLook = gvCameraPosition.xyz - input[0].pos;
    vLook = normalize(vLook);
    float3 vRight = cross(vUp, vLook);
    float fHalfW = input[0].radius;

    float4 pVertices[4] =
    {

        float4(input[0].pos + fHalfW * vRight - fHalfW * vUp, 1.0f),
        float4(input[0].pos + fHalfW * vRight + fHalfW * vUp, 1.0f),
        float4(input[0].pos - fHalfW * vRight - fHalfW * vUp, 1.0f),
        float4(input[0].pos - fHalfW * vRight + fHalfW * vUp, 1.0f)
    };
    //uint frame = (1 / input[0].color.a)*10;
    float2 UVs[4] = { float2(0.0f, 0.2f), float2(0.0f, 0.0f), float2(0.16f, 0.2f), float2(0.16f, 0.0f) };

    GS_PARTICAL_OUTPUT output;
    for (int i = 0; i < 4; ++i)
    {
        output.posH = mul(mul(pVertices[i], gmtxView), gmtxProjection);
        output.uv = UVs[i];
        output.color = input[0].color; // 수정
        SpriteStream.Append(output);
    }
}
Texture2D<float4> gSmokeTexture : register(t7);
```

파티클을 생성하고 그리는 코드(DirectX Sample ParticlesGS를 참고해 만들었습니다)

테셀레이션

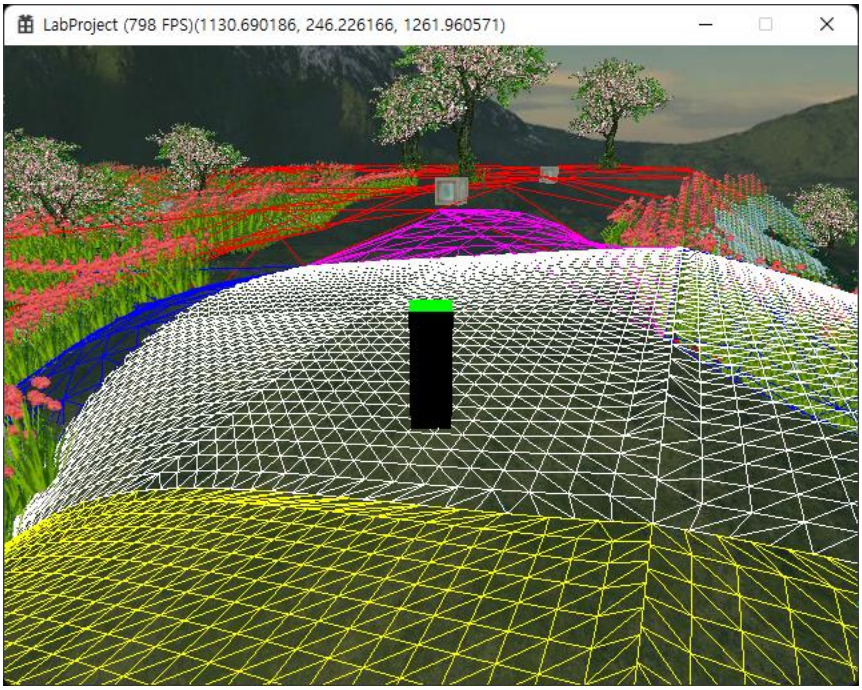
- 목표
  - 카메라와 거리에 맞춰 동적 LOD를 구현해보자
  - 동적으로 변하는 지형에 맞게 플레이어를 이동시키자
- 구현
  - 기존의 지형의 높이 정보를 가지고 있던 높이 맵의 높이 값을 제어점으로 사용하여 5차 베지어 곡면으로 나타냈습니다. 그러기 위해 헐 셰이더에서 제어점을 입력받아 패치 제어점과 패치 상수 데이터로 나타내야합니다. 패치 제어점은 별다른 변화 없이 그대로 통과하여 25개 모아 보내주고 패치 상수의 값은 카메라와의 거리에 따라 어느 정도로 쪼갤지, 수평, 수직으로 어느 정도로 나눌지를 정해줍니다. 그리고 해당 값에 따라 테셀레이터가 정점을 나눠주면 그 정점을 도메인 셰이더가 입력받아 패치 제어점에 따른 베지어 곡면으로 나타내고 월드, 뷰, 프로젝션 변환을 해 SV_Position을 만들고 픽셀 셰이더로 넘겨줍니다.
  - 지형이 동적으로 변하기 때문에 플레이어를 지형 위에 올리기 위해서는 변화되는 지형의 높이에 맞춰 플레이어가 위치해야 합니다. 그러기 위해 지형의 높이를 구할 때 셰이더와 같이 25개의 제어점으로 나타낸 베지어 곡면에서 현제 플레이어의 높이를 계산해줍니다.

```
float CalculateTessFactor(float3 p)
{
    float fDistToCamera = distance(p, gvCameraPosition);
    float s = saturate((fDistToCamera) / (200.0f));

    return (lerp(64.0f, 1.0f, s));
}
HS_TERRAIN_CONSTANT_OUTPUT ConstantHS(InputPatch<VS_TERRAIN_OUTPUT, 25> input)
{
    HS_TERRAIN_CONSTANT_OUTPUT output;

    //변들의 중점
    float3 e0 = 0.5f * (input[0].positionW + input[4].positionW);
    float3 e1 = 0.5f * (input[0].positionW + input[20].positionW);
    float3 e2 = 0.5f * (input[4].positionW + input[24].positionW);
    float3 e3 = 0.5f * (input[20].positionW + input[24].positionW);

    //카메라와 거리에 따라 각 변을 쪼개는 개수를 정한다.
    output.fTessEdges[0] = CalculateTessFactor(e0);
    output.fTessEdges[1] = CalculateTessFactor(e1);
    output.fTessEdges[2] = CalculateTessFactor(e2);
    output.fTessEdges[3] = CalculateTessFactor(e3);

    float3 f3Sum = float3(0.0f, 0.0f, 0.0f);
    for (int i = 0; i < 25; ++i)
        f3Sum += input[i].positionW;
    float3 f3Center = f3Sum / 25.0f;
    //수직, 수평으로 패치 내부의 테셀레이션 정도를 정한다.
    output.fTessInsides[0] = output.fTessInsides[1] = CalculateTessFactor(f3Center);
    return output;
}
```

지형 Constant hull shader입니다. Shaders.hlsl

```hlsl
[domain("quad")]
DS_TERRAIN_OUPUT DSTerrain(HS_TERRAIN_CONSTANT_OUTPUT input, float2 uv : SV_DomainLocation, OutputPatch<HS_TERRAIN_OUTPUT, 25> patch)
{
    DS_TERRAIN_OUPUT output = (DS_TERRAIN_OUPUT) 0;

    //테셀레이션된 정점의 위치가 uv정보로 나타나므로 실제 정점 위치를 찾아야한다.
    float uB[5], vB[5];
    BernsteinCoeffcient5x5(uv.x, uB);
    BernsteinCoeffcient5x5(uv.y, vB);

    output.color = lerp(lerp(patch[0].color, patch[4].color, uv.x), lerp(patch[20].color, patch[24].color, uv.x), uv.y);
    output.uv0 = lerp(lerp(patch[0].uv0, patch[4].uv0, uv.x), lerp(patch[20].uv0, patch[24].uv0, uv.x), uv.y);
    output.uv1 = lerp(lerp(patch[0].uv1, patch[4].uv1, uv.x), lerp(patch[20].uv1, patch[24].uv1, uv.x), uv.y);

    //실제 위치
    float3 p = CubicBezierSum5x5(patch, uB, vB);

    matrix mtxWorldViewProjection = mul(mul(gmtxGameObject, gmtxView), gmtxProjection);
    output.position = mul(float4(p, 1.0f), mtxWorldViewProjection);

    //와이어 프레임일 경우테셀레이션 정도에 따라 색을 변화시키기 위해 전달한다.
    output.tessellation = float4(input.fTessEdges[0], input.fTessEdges[1], input.fTessEdges[2], input.fTessEdges[3]);

    return (output);
}
```

지형 Domain shader입니다. Shaders.hlsl



실행 화면입니다.

그림자 맵

- 목표
  - 그림자 맵을 이용해 조명에 대한 그림자를 생성해보자
  - 바이어스와 PCF를 이용해 그림자 여드름 현상을 막고, 부드러운 그림자 경계를 만들어 보자
  - 그림자 맵을 다른 스레드에서 생성해보자
- 구현

  1단계를 위한 자원 생성

  - 오브젝트와 조명의 거리에 대한 정보를 버퍼에 쓰는 단계를 수행하기 위해 조명의 개수만큼의 버퍼

와 버퍼 뷰(파이프라인을 거쳐 광원 시점에서 렌더링 된 결과를 렌더 타겟에 출력할 것이기 때문에 렌더 타겟 뷰로 생성), 뷰를 담을 서술자 힙을 생성합니다. 또 조명의 위치에서 오브젝트를 바라보고 해당 위치 정보를 깊이 버퍼에 쓰기 때문에 조명의 위치를 표현하기 위한 변환 행렬(카메라 행렬, 투영 행렬, NDC공간에서 텍스처 공간으로 변환하기 위한 텍스처 행렬)도 생성합니다.

1단계 렌더링

- 조명의 위치와 방향에 맞춰 카메라 행렬, 투영 행렬을 만들고 새로운 렌더 타겟에 렌더링을 시작합니다. 그러면 렌더 타겟에 오브젝트와 조명의 거리에 대한 깊이 정보가 저장되게 됩니다. 2단계 렌더링에서 렌더 타겟의 텍스처를 읽기 위해 위에서 만든 카메라 행렬, 투영 행렬, 텍스처 변환 행렬을 사용하게 됩니다.

2단계를 위한 자원 생성

- 기존 렌더 타겟에 렌더링하기 위한 자원을 생성하는 과정과 동일하기 때문에 생략하겠습니다.

2단계 렌더링

- 기존에 렌더링하던 대로 렌더링을 합니다. 이때 1단계에서 생성한 깊이 정보를 가진 텍스처(쉐도우 맵)를 사용해 그림자를 출력하기 위한 과정이 추가됩니다. 1단계에서 만든 조명의 위치와 방향으로 설정된 카메라 행렬, 투영행렬과 NDC공간에서 텍스처 공간으로 변환하기 위한 행렬을 이용해(원근 나눗셈 수행)현재 픽셀 위치의 정점을 변환합니다(월드 공간의 좌표 -> 조명의 위치에서의 NDC좌표 -> NDC좌표를 uv로 나타낸 좌표) 그러면 변환된 정점은 해당 조명의 쉐도우 맵을 이용해 조명의 위치에서의 깊이 값을 알 수 있게 됩니다.

- 이제 월드 공간의 정점(A)와 조명과의 깊이 값과(d(p)) 쉐도우 맵을 이용해 똑같은 월드 공간의 정점(A)를 조명에서 봤을 때 가장 가까운 물체와의 깊이 값(s(p))를 알게 되었습니다. 이걸 SampleCmpLevelZero를 이용해 두 값을 비교해 현재 픽셀이 그림자인지 아닌지 구별합니다.

- 이 때 조명 절두체 바깥은 빛을 받지 않으므로 샘플러를 설정할 때 boarder로 설정해 영역 밖의 점은 프로그래머가 정한 값을 반환하도록 하였습니다.

- 이렇게 생성한 그림자에는 문제가 생길 수 있습니다. 먼저 월드 공간의 정점을 쉐도우 맵의 uv좌표로 변환 했는데 uv좌표에 해당하는 텍셀이 없어 주변점을 샘플러가 **근접점샘플링** 하여 가져왔다고 가정해보겠습니다. 이때 다른 위치의 정점이 같은 쉐도우 맵의 텍셀을 사용하면서 실제로는 그림자 영역이 아닌데 그림자 영역으로 나타날 수 있습니다. 이처럼 같은 쉐도우 맵의 텍셀을 공유하는 정점이 생기게 됩니다. 이는 그림자 맵의 해상도를 높이거나 그림자 맵의 깊이 값에 일정한 값(bias)를 더해서 해결할 수 있습니다. 하지만 이 bias값은 기울기에 따라 달라져야 합니다. 이것은 Directx12에서 기울기에 따른 bias값을 계산해 적용해 줄 수 있습니다.
  D3D12_RASTERIZER_DESC -> SlopScaledDepthBias
  또 다른 문제로는 그림자 판정 여부를 한 픽셀에 대해서만 정해서 출력하면 해당 픽셀은 그림자인지, 아닌지로만 구분되게 됩니다. 이는 그림자 경계가 뚜렷해지고 쉐도우 맵의 해상도가 낮을 경우 계단 현상도 나타날 수 있습니다. 이를 방지하기 위해 한 픽셀에 대해서만 계산하지 않고 주변 픽셀의 그림자 여부까지 확인한 다음 보간하여 해당 픽셀이 주변의 그림자에 영향을 받을 수 있도록 하여 부드러운 그림자 경계를 나타낼 수 있고 계단 현상을 막을 수 있습니다.

다른 스레드에서 쉐도우 맵 생성

- 1단계를 다른 스레드에서 수행하기 위해 쉐도우 맵 생성이 끝나면 씬에서 그림자가 생기는 오브젝트를 그리도록 하였고 한 프레임이 다 그려지면 다시 그림자 맵을 그리도록 하였습니다. 이때 쉐도우 맵을 위한 명령 리스트와 씬을 그리기 위한 명령 리스트를 따로 둬 명령 리스트의 명령을 동시에 추가할 수 있도록 하였습니다

```
void CDepthRenderShader::PrepareShadowMap(ID3D12GraphicsCommandList* pd3dCommandList)
{
    for (int j = 0; j < MAX_LIGHTS; j++)
    {
        if (m_pLights[j].m_bEnable)
        {
            XMFLOAT3 xmf3Position = m_pLights[j].m_xmf3Position;
            XMFLOAT3 xmf3Look = m_pLights[j].m_xmf3Direction;
            XMFLOAT3 xmf3Up = XMFLOAT3(0.0f, +1.0f, 0.0f);

            XMMATRIX xmmtxView = XMMatrixLookToLH(XMLoadFloat3(&xmf3Position), XMLoadFloat3(&xmf3Look), XMLoadFloat3(&xmf3Up));

            float fNearPlaneDistance = 10.0f, fFarPlaneDistance = m_pLights[j].m_fRange;

            XMMATRIX xmmtxProjection;
            if (m_pLights[j].m_nType == DIRECTIONAL_LIGHT)
            {
                float fWidth = _PLANE_WIDTH, fHeight = _PLANE_HEIGHT;
                xmmtxProjection = XMMatrixOrthographicLH(fWidth, fHeight, fNearPlaneDistance, fFarPlaneDistance);
            }
            else if (m_pLights[j].m_nType == SPOT_LIGHT)
            {
                float fFovAngle = 60.0f; // m_pLights->m_pLights[j].m_fPhi = cos(60.0f);
                float fAspectRatio = float(_DEPTH_BUFFER_WIDTH) / float(_DEPTH_BUFFER_HEIGHT);
                xmmtxProjection = XMMatrixPerspectiveFovLH(XMConvertToRadians(fFovAngle), fAspectRatio, fNearPlaneDistance, fFarPlaneDistance);
            }

            m_ppDepthRenderCameras[j]->SetPosition(xmf3Position);
            XMStoreFloat4x4(&m_ppDepthRenderCameras[j]->m_xmf4x4View, xmmtxView);
            XMStoreFloat4x4(&m_ppDepthRenderCameras[j]->m_xmf4x4Projection, xmmtxProjection);

            XMMATRIX xmmtxToTexture = XMMatrixTranspose(xmmtxView * xmmtxProjection * m_xmProjectionToTexture);
            XMStoreFloat4x4(&m_pToLightSpaces->m_pToLightSpaces[j].m_xmf4x4ToTexture, xmmtxToTexture);

            m_pToLightSpaces->m_pToLightSpaces[j].m_xmf4Position = XMFLOAT4(xmf3Position.x, xmf3Position.y, xmf3Position.z, 1.0f);

            pd3dCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(m_pDepthTexture->GetResource(j), D3D12_RESOURCE_STATE_COMMON, D3D12_RESOURCE_STATE_RENDER_TARGET));

            FLOAT pfClearColor[4] = { 1.0f, 1.0f, 1.0f, 1.0f };
            pd3dCommandList->ClearRenderTargetView(m_pd3dRtvCPUDescriptorHandles[j], pfClearColor, 0, NULL);

            pd3dCommandList->ClearDepthStencilView(m_d3dDsvDescriptorCPUHandle, D3D12_CLEAR_FLAG_DEPTH, 1.0f, 0, 0, NULL);

            pd3dCommandList->OMSetRenderTargets(1, &m_pd3dRtvCPUDescriptorHandles[j], TRUE, &m_d3dDsvDescriptorCPUHandle);

            Render(pd3dCommandList, m_ppDepthRenderCameras[j]);

            pd3dCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(m_pDepthTexture->GetResource(j), D3D12_RESOURCE_STATE_RENDER_TARGET, D3D12_RESOURCE_STATE_COMMON));
        }
        else
        {
```

오브젝트와 조명의 거리를 나타낸 그림자 맵을 만드는 코드 Shader.cpp

```
float Compute3x3ShadowFactor(float2 uv, float fDepth, uint nIndex)
{
    float fPercentLit = gtxtDepthTextures[nIndex].SampleCmpLevelZero(gssComparisonPCFShadow, uv, fDepth).r;
    fPercentLit += gtxtDepthTextures[nIndex].SampleCmpLevelZero(gssComparisonPCFShadow, uv + float2(-DELTA_X, 0.0f), fDepth).r;
    fPercentLit += gtxtDepthTextures[nIndex].SampleCmpLevelZero(gssComparisonPCFShadow, uv + float2(+DELTA_X, 0.0f), fDepth).r;
    fPercentLit += gtxtDepthTextures[nIndex].SampleCmpLevelZero(gssComparisonPCFShadow, uv + float2(0.0f, -DELTA_Y), fDepth).r;
    fPercentLit += gtxtDepthTextures[nIndex].SampleCmpLevelZero(gssComparisonPCFShadow, uv + float2(0.0f, +DELTA_Y), fDepth).r;
    fPercentLit += gtxtDepthTextures[nIndex].SampleCmpLevelZero(gssComparisonPCFShadow, uv + float2(-DELTA_X, -DELTA_Y), fDepth).r;
    fPercentLit += gtxtDepthTextures[nIndex].SampleCmpLevelZero(gssComparisonPCFShadow, uv + float2(-DELTA_X, +DELTA_Y), fDepth).r;
    fPercentLit += gtxtDepthTextures[nIndex].SampleCmpLevelZero(gssComparisonPCFShadow, uv + float2(+DELTA_X, -DELTA_Y), fDepth).r;
    fPercentLit += gtxtDepthTextures[nIndex].SampleCmpLevelZero(gssComparisonPCFShadow, uv + float2(+DELTA_X, +DELTA_Y), fDepth).r;

    return(fPercentLit / 9.0f);
}
```

해당 픽셀의 그림자 여부를 판단하는 코드 Light.hlsl

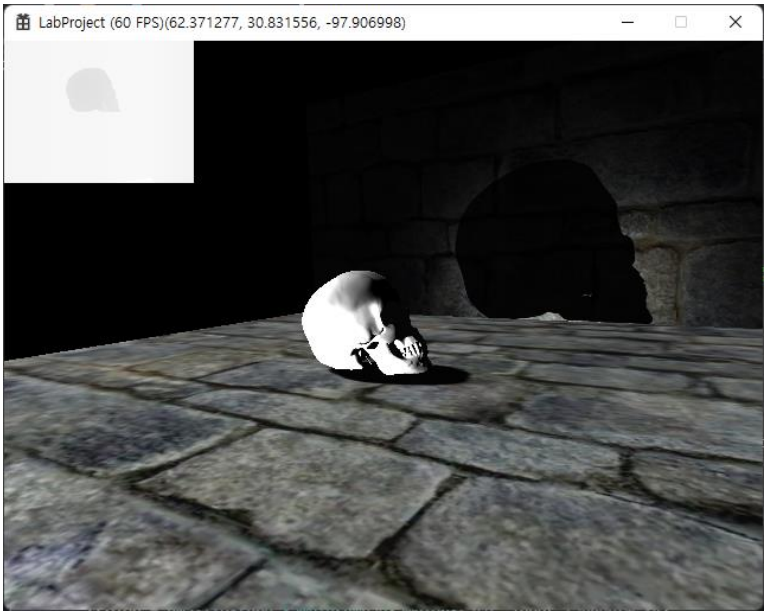
```hlsl
float4 Lighting(float3 vPosition, float3 vNormal, bool bShadow, float4 uvs[MAX_LIGHTS])
{
    float3 vCameraPosition = float3(gvCameraPosition.x, gvCameraPosition.y, gvCameraPosition.z);
    float3 vToCamera = normalize(vCameraPosition - vPosition);

    float4 cColor = float4(0.0f, 0.0f, 0.0f, 0.0f);
[unroll]
    for (int i = 0; i < MAX_LIGHTS; i++)
    {
        if (gLights[i].m_bEnable)
        {
            float fShadowFactor = 1.0f;
            if (bShadow) fShadowFactor = Compute3x3ShadowFactor(uvs[i].xy / uvs[i].ww, uvs[i].z / uvs[i].w, i);

            if (gLights[i].m_nType == DIRECTIONAL_LIGHT)
            {
                cColor += DirectionalLight(i, vNormal, vToCamera) * fShadowFactor;
            }
            else if (gLights[i].m_nType == POINT_LIGHT)
            {
                cColor += PointLight(i, vPosition, vNormal, vToCamera) * fShadowFactor;
            }
            else if (gLights[i].m_nType == SPOT_LIGHT)
            {
                cColor += SpotLight(i, vPosition, vNormal, vToCamera) * fShadowFactor;
            }
            cColor += gLights[i].m_cAmbient * gMaterials[gnMaterial].m_cAmbient;;
        }
    }

    cColor += (gcGlobalAmbientLight * gMaterials[gnMaterial].m_cAmbient);
    cColor.a = gMaterials[gnMaterial].m_cDiffuse.a;

    return(cColor);
}
```

빛과 그림자에 영향을 받는 픽셀의 색을 구하는 코드 Light.hlsl

```cpp
void CGameFramework::ShadowFrameAdvance()
{
    while (1) {

        DWORD ret = ::WaitForSingleObject(m_hShadowStartEvent, INFINITE);

        HRESULT hResult = m_pd3dShaderCommandAllocator->Reset();
        hResult = m_pd3dShaderCommandList->Reset(m_pd3dShaderCommandAllocator, NULL);
        m_ppScene[1]->PrepareRender(m_pd3dShaderCommandList);

        m_ppScene[1]->OnPreRender(m_pd3dShaderCommandList);

        hResult = m_pd3dShaderCommandList->Close();

        ID3D12CommandList* ppd3dCommandLists[] = { m_pd3dShaderCommandList };
        m_pd3dCommandQueue->ExecuteCommandLists(1, ppd3dCommandLists);
        ::SetEvent(m_hShadowEndEvent);

    }
}
```

-

- Intel i5-10400 RTX 2060 기준으로 스레드를 나누기 전 600~610fps 정도의 성능이 나왔고 스레드를 나눌 경우 704~710fps정도의 성능이 나왔습니다.



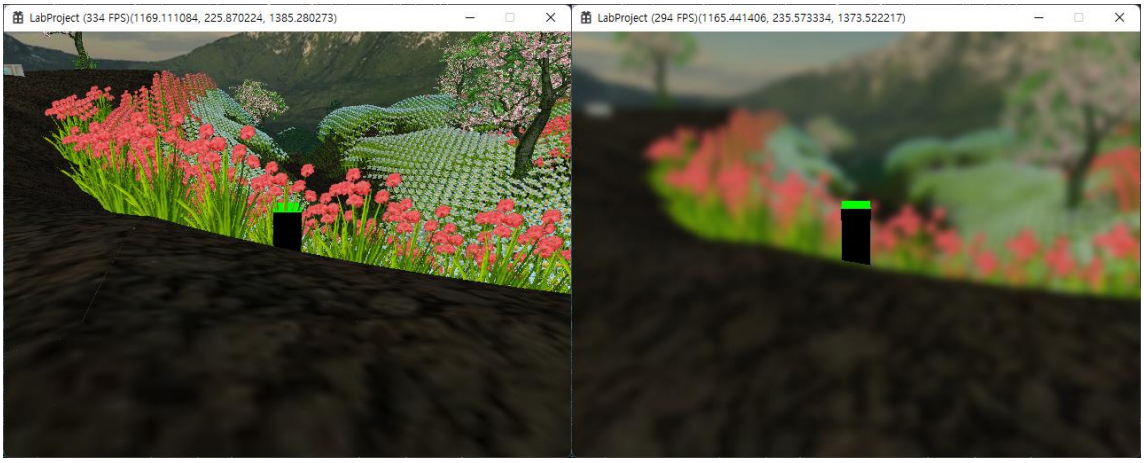
-Directional light, Spot light를 사용해 그림자를 만든 화면입니다.

블러링

- 목표
  - 계산 셰이더를 사용해 포스트 프로세싱 효과를 만들어보자
- **구현**
  - 계산 셰이더는 그래픽 파이프라인에 포함되지 않고 따로 존재하므로 PSO를 CreateCompute PipelineState를 이용해 만들어 줍니다.
  - 생성한 자원을 총 4개의 뷰(한 개의 리소스를 Srv,Uav로 나타냄)로 나타내고 서술자 힙에 담습니다. 그리고 흐리기 연산을 수행하면서 입력에 해당하는 리소스와 출력에 해당하는 리소스를 Set해 줍니다.
  - 화면의 모든 픽셀을 흐리기 위해 스레드 그룹의 스레드 xyz개수와 화면 픽셀의 수를 고려해 Dispatch의 xyz값을 정해줘야 합니다.
  - 흐리기 연산의 샘플을 수를 줄여 흐리기 연산을 단순화하기 위해 수직으로 흐리기와 수평으로 흐리는 것으로 나누어 계산을 합니다. 리소스 두 개를 A,B라고 하겠습니다. 렌더링을 시작하면 A리소스에 후면 버퍼의 내용을 복사하고 A리소스를 입력(srv)으로 Set하고 B리소스를 출력(uav)으로 Set합니다. 첫 번째 수평 흐리기를 수행한 후 출력과 입력의 위치를 바꿔(srv->uav, uav->srv) 수직 흐리기를 수행합니다. 해당 과정을 지정된 흐리기 횟수만큼 실행합니다. 모든 흐리기 연산이 수행되고 나면 후면 버퍼에 흐리기가 적용된 리소스인 A버퍼를 복사합니다.
  - 흐리기 연산에 사용할 텍셀에 대한 가중치는 중심에서 멀어 질수록 작아지고 총 가중치의 합이 1이 되도록(흐려진 픽셀이 밝아지거나 어두워지는 것을 막기 위해) 가우스 함수로 구했습니다.
  - 후면 버퍼에 렌더링을 한 후 후면 버퍼를 텍스쳐에 복사해 블러를 적용하고 적용된 텍스처를 다시 후면 버퍼에 복사했습니다.

```cpp
void CBlurShader::Execute(ID3D12GraphicsCommandList* pd3dCommandList, ID3D12RootSignature* pd3dRootSignature, ID3D12Resource* input, int blurCount)
{
    int blurRadius = nWeight / 2;


    pd3dCommandList->SetComputeRootSignature(pd3dRootSignature);

    pd3dCommandList->SetComputeRoot32BitConstants(2, nWeight, gfWeights, 0);

    pd3dCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(input,
        D3D12_RESOURCE_STATE_RENDER_TARGET, D3D12_RESOURCE_STATE_COPY_SOURCE));

    pd3dCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(mBlurMap0,
        D3D12_RESOURCE_STATE_COMMON, D3D12_RESOURCE_STATE_COPY_DEST));

    pd3dCommandList->CopyResource(mBlurMap0, input);

    pd3dCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(mBlurMap0,
        D3D12_RESOURCE_STATE_COPY_DEST, D3D12_RESOURCE_STATE_GENERIC_READ));

    pd3dCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(mBlurMap1,
        D3D12_RESOURCE_STATE_COMMON, D3D12_RESOURCE_STATE_UNORDERED_ACCESS));

    pd3dCommandList->SetDescriptorHeaps(1, &m_pd3dCbvSrvUavDescriptorHeap);
    for (int i = 0; i < blurCount; ++i)
    {
        //가로
        pd3dCommandList->SetPipelineState(m_phorzPipelineState);
        pd3dCommandList->SetComputeRootDescriptorTable(0, mBlur0GpuSrv);
        pd3dCommandList->SetComputeRootDescriptorTable(1, mBlur1GpuUav);

        UINT numGroupsX = (UINT)ceilf(mWidth / 256.0f);
        pd3dCommandList->Dispatch(numGroupsX, mHeight, 1);

        pd3dCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(mBlurMap0,
            D3D12_RESOURCE_STATE_GENERIC_READ, D3D12_RESOURCE_STATE_UNORDERED_ACCESS));
        pd3dCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(mBlurMap1,
            D3D12_RESOURCE_STATE_UNORDERED_ACCESS, D3D12_RESOURCE_STATE_GENERIC_READ));

        //세로
        pd3dCommandList->SetPipelineState(m_pvertPipelineState);
        pd3dCommandList->SetComputeRootDescriptorTable(0, mBlur1GpuUav);
        pd3dCommandList->SetComputeRootDescriptorTable(1, mBlur0GpuSrv);

        UINT numGroupsY = (UINT)ceilf(mHeight / 256.0f);
        pd3dCommandList->Dispatch(mWidth, numGroupsY, 1);

        pd3dCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(mBlurMap0,
            D3D12_RESOURCE_STATE_UNORDERED_ACCESS, D3D12_RESOURCE_STATE_GENERIC_READ));
        pd3dCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(mBlurMap1,
            D3D12_RESOURCE_STATE_GENERIC_READ, D3D12_RESOURCE_STATE_UNORDERED_ACCESS));
    }

    pd3dCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(input,
        D3D12_RESOURCE_STATE_COPY_SOURCE, D3D12_RESOURCE_STATE_COPY_DEST));
    pd3dCommandList->CopyResource(input, mBlurMap0);

    // Transition to PRESENT state.
    pd3dCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(input,
        D3D12_RESOURCE_STATE_COPY_DEST, D3D12_RESOURCE_STATE_RENDER_TARGET));
    pd3dCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(mBlurMap0,
        D3D12_RESOURCE_STATE_GENERIC_READ, D3D12_RESOURCE_STATE_COMMON));
    pd3dCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(mBlurMap1,
        D3D12_RESOURCE_STATE_UNORDERED_ACCESS, D3D12_RESOURCE_STATE_COMMON));
}
```

후면 버퍼의 내용을 복사한 텍스쳐에 블러를 적용한 뒤 다시 후면 버퍼에 복사하는 코드입니다.


LabProject (334 FPS)(1169.111084, 225.870224, 1385.280273)
LabProject (294 FPS)(1165.441406, 235.573334, 1373.522217)