DirectX11

지형 툴 및 다양한 기능

https://youtu.be/1Db5MEnrOJ0 포트폴리오 영상 링크
https://github.com/hoya1215/DirectX11.git 깃허브 주소
https://velog.io/@hoya1215/posts 포트폴리오 만드는 과정 기록 블로그 주소

포트폴리오의 의도와 목표 및 얻게 된 결과

의도 : DirectX11 과 그래픽스 지식을 키우고 다양한 효과와 기능을 구현하는 연습을 하고 다양한 방법을 생각해 보면서 안목을 키우고자 제작하였습니다.

목표 : 모든 Shader 를 직접 사용해보고 현재 다양한 엔진에서 실제로 적용하고 있는 렌더링 기술 및 최적화 기술을 구현해보고 실제 게임내에 있는 기능들을 구현

얻게 된 결과 : 제대로 된 프로그래밍 공부 방법을 알게 되었고 다양한 기술들이 실제로 어떻게 적용되어서 나오는지 잘 알 수 있었습니다.

또한 포기하지 않고 도전하는 마음가짐과 무엇이든지 할 수 있다는 자신감을 가질 수 있었습니다.

설명 순서

01 Basic Scene

02 지형 Edit

- 지형 높이
- 지형 충돌 상자
- 지형 **Brush**
- 빌보드
- **Lake**

03 물체 Edit

04 날씨

05 그림자

06 충돌 처리

07 최적화

- 프러스텀 컬링 , 그래픽 상 중 하
- **LOD , Clip**
- **LOD** 적용 시 발생하는 문제 해결
- 이외에 최적화 할 수 있는 방법 정리

설명 순서

08 기타 기능

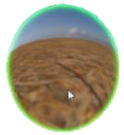
- 미니맵 , 방향키뷰어 **(UI)**
- 캐릭터 실루엣 외곽선 효과 **(** 투과 기능 **)**
- 투명도 이용한 투과 기능
- 터닝 포인트
- **SSAO**
- **Deferred Rendering**
- **Depth Of Field**
- **Screen Space Reflection** 시도 **(** 미완성 **)**

Basic Scene

RTT , 화면전환 후처리

카메라를 Basic Scene 카메라와 Edit Scene 두개의 카메라를 분리하여
Basic Scene 일때 Edit Scene 일때 두 개의 Scene 을 각각 렌더링 해주고
현재 Scene 에 맞는 렌더타겟을 화면에 나타날 렌더타겟으로 지정해줍니다.

Basic Scene 에서 Edit Scene 으로 가는 역할을 하는 물체는
Edit Scene 을 그려준 렌더타겟을 텍스처로 샘플링하였고
마우스로 picking 되면 Rim 효과를 이용하여 초록색으로 표시해주었습니다.



```
// Rim 효과
if (pick) // 일반 물체
{
    float rim = (1.0f - dot(input.normalWorld, normalize(eyeWorld - input.posWorld)));
    rim = smoothstep(0.0, 1.0, rim);
    rim = pow(abs(rim), rimPower);

    output.pixelColor.xyz += rim * rimStrength * rimColor;
}
```

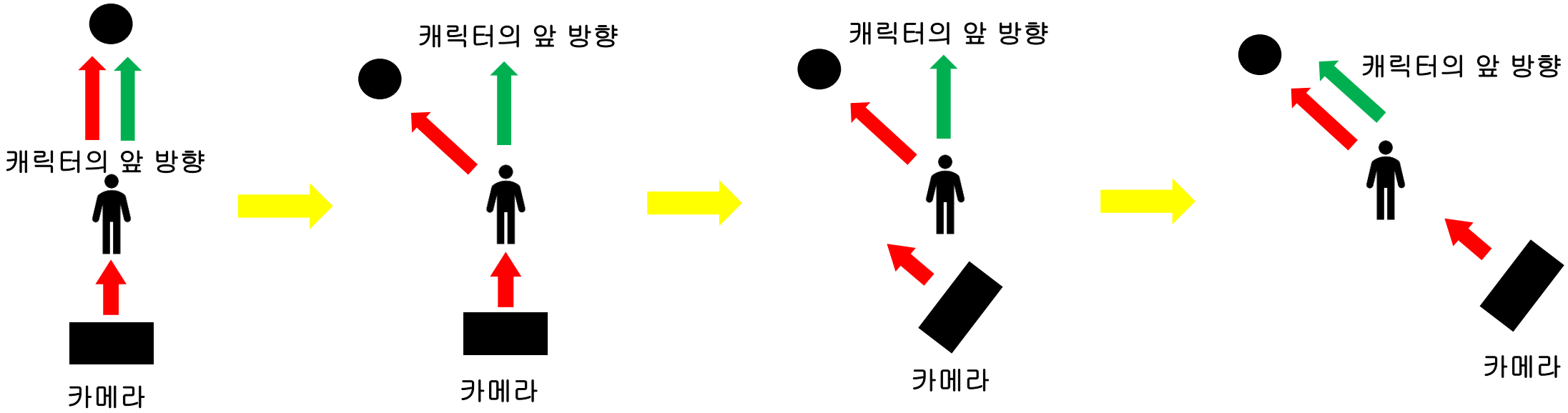
클릭을 하면 영역의 반지름이 커지고 두개의 Scene 장면을 후처리 셰이더에서 섞어줍니다.

```
if (length(center - TextoNDC) <= cutAwayRadius)
{
    combined = EditRTV.Sample(linearWrapSampler, input.texcoord).xyz;
}
```

Basic Scene

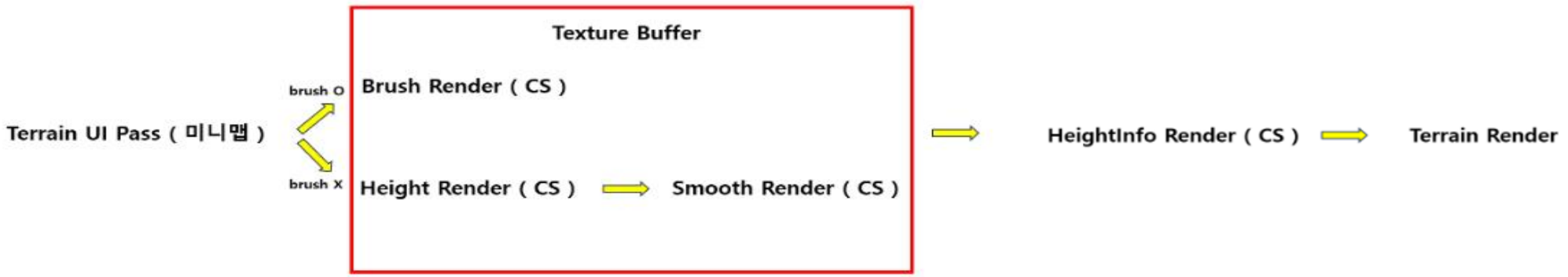
카메라 회전 , 캐릭터의 이동 방향

Basic Scene 에서 물체를 기준으로 카메라가 회전 및 이동을 하고 캐릭터도 특정 축을 기준으로 이동하는 것이 아니라 카메라의 View 방향을 기준으로 자유롭게 움직일 수 있게 설정했습니다.



지형 Edit - Render 순서

Terrain Edit 상태일때



Terrain Edit 상태가 아닐때

Terrain UI Pass (미니맵) ⟹ HeightInfo Render (CS) ⟹ Terrain Render

지형 Edit - Render 순서

Terrain UI Pass (미니맵)

- Terrain 을 위에서 봤을때의 모습을 UI 의 텍스처로 이용

Terrain Brush Render

- Terrain 의 여러가지 텍스처의 가중치를 픽셀마다 저장

Terrain Height Render

- Terrain의 높이 값을 픽셀 단위로 조절하고 저장
- Terrain Render 에서 Domain Shader 에서 불러와서 정점의 높이값을 변경
- 캐릭터의 높이 충돌 검사를 할때 Height Texture Buffer 사용하여 검사

Terrain Smooth Render

- Terrain Height Render 에서 변경한 높이 값을 주변 9개의 픽셀의 높이 값을 평균내서 조정

HeightInfo Render

- HeightInfo 구조적 버퍼의 각 정점에다가 픽셀 단위로 저장한 높이 값을 저장
- Terrain의 경계상자를 만드는 용도로 사용

Terrain 을 Initialize 할때 Texture Buffer 의 모든 픽셀값을 초기화시켜주는 Compute Shader 를 이용하여 모든 Texture Buffer 를 초기화 시켜줍니다.

지형 Edit - 높이

마우스를 클릭했을때 클릭한 지점과 반지름을 높이 CS 로 전달하고 CS 에서 범위 안에 있는 모든 픽셀의 높이값을 변경해줍니다.

우 클릭 : 높이 올리기　　좌 클릭 : 높이 내리기

```
float dist = length(mouseTexcoord - texcoord);
float range = radius / (terrainScale * 2.0);
float newDist = sin((PI / 2) + (PI / 2) * (dist / range));

if (dist < range) {

    if (leftclicked == 1) {
        world.y += heightlength * newDist;
    } else if (rightclicked == 1) {
        world.y -= heightlength * newDist;
    }
}
```

범위 안에 있는 픽셀의 높이를 일정하게 올리지 않고 클릭한 지점과의 거리에 따라 둥글게 언덕처럼 올릴 수 있게 sin 함수를 이용했습니다.

CS 에서 DispatchThreadID 값을 픽셀 위치에 맞게 texture 좌표로 변환해주어야 합니다.

```
int width, height;
TerrainPass.GetDimensions(width, height);

float2 texcoord = dtID.xy;

texcoord.x /= (float)width;
texcoord.y /= (float)height;
texcoord.y = 1.0 - texcoord.y;
```

후에 Smooth CS 에서 주변 픽셀 9 개값을 평균내서 높이를 부드럽게 조정합니다.

CS : Compute Shader

지형 Edit – 충돌 상자

충돌 상자 자동 생성

- 높이를 올리면 올린 영역의 최대 최소 X축 , Z축을 충돌 상자를 생성하기 전까지 계속 기록합니다.
- AutoCollisionBox 버튼을 클릭하면 Terrian의 GetMaxheight 함수를 이용하여 저장한 축의 범위 안에서 최고 높이를 찾아서 자동으로 충돌 상자를 생성해줍니다.

특징 : 충돌상자를 간단하고 쉽게 생성할 수 있지만 최대 최소 축이 상자를 생성하기 전까지 계속 갱신되므로 중간에 특정 영역의 상자를 생성할 수 없고 한번에 하나만 생성이 가능합니다.

충돌 상자 수동 생성

- 사용자가 직접 생성하고 싶은 영역을 클릭하면 Terrian의 GetMaxheight 함수를 이용하여 충돌 상자를 생성해줍니다.

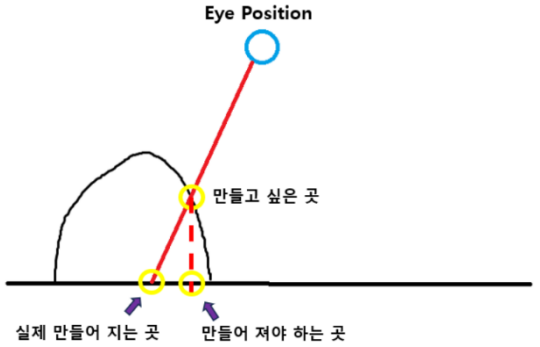
특징 : 세밀한 충돌이나 특정 영역만 충돌이 필요한 경우 유용하게 설치할 수 있지만 생성하고 싶은 곳과 실제 생성되는 곳의 오차가 발생하여 위치가 정확하지 않을 수 있습니다.

-> 드래그로 상자를 생성하는 방법 생각 (드래그는 높이 보정 기능으로 구현하였습니다.

GetMaxheight 함수

HeightInfo Render 에서 정점 단위로 저장한 높이값을 staging buffer 를 이용해서 Map , UnMap 을 이용하여 주어진 범위안에 있는 정점들 중에서 최고 높이값을 찾아서 반환해줍니다.

오차가 발생하는 이유 생각하기




```
float Terrain::GetMaxheight(ComPtr<ID3D11DeviceContext> &context,
                            float minX, float maxX, float minZ,
                            float maxZ)
{
// 최대 최소 높이 찾기
float heightMax = 0.f;

// UAV 매핑
HeightInfo *pData;
D3D11_MAPPED_SUBRESOURCE mappedResource;
context->Map(m_stagingBuffer.Get(), 0, D3D11_MAP_READ_WRITE, 0,
             &mappedResource);
pData = (HeightInfo *)mappedResource.pData;

// 높이가 최대인 점 찾기

for (int i = 0; i < m_heightInfo.size(); i++) {


    if (-pData[i].pos.z > heightMax && pData[i].pos.x >= minX &&
        pData[i].pos.x <= maxX && pData[i].pos.y >= minZ && pData[i].pos.y <= maxZ)
            heightMax = -pData[i].pos.z;
}

// 매핑 해제
context->Unmap(m_stagingBuffer.Get(), 0);

return heightMax;
}
```

지형 Edit - Brush

지형에 각기 다른 텍스처들을 섞어서 색을 나타낼 수 있게 구현했습니다.

픽셀의 가중치 Texture Buffer 의 포맷을 R16G16B16A16_FLOAT 로 설정하여 각 픽셀의 rgba 값을 각각의 텍스처들의 가중치 값으로 사용하였습니다.

원리

- 텍스처들을 클릭하면 해당 텍스처들에 해당하는 Brush Index 가 상수버퍼에 설정되고 지형을 왼쪽 버튼으로 클릭 (꾹 누르기) 하면 Brush CS 에서 해당 Index 에 해당하는 픽셀의 rgba 값에 가중치를 더해주는 방식입니다.
  (오른쪽 버튼 : 가중치 마이너스)

한번에 전체 색 바꾸는 기능

- 툴에 있는 Change 버튼을 클릭하면 현재 해당 텍스처로 지형의 전체 색을 한번에 바꿀 수 있습니다.
- 다른 텍스처들의 가중치의 값이 0 이 아니라 조금이라도 있다면 그 텍스처들의 색은 그대로 유지한채로 바꿔줍니다.

최종적으로 지형의 PS 에서 TextureBuffer 에서 각 텍스처의 가중치값을 샘플링 해 와서 색상을 가중치값에 따라 결정해주고 더해줍니다.

기타

- Brush 모드일때는 영역을 나타내주는 원의 색상을 초록색이 아닌 바꿀 텍스처 색상과 현재 지형 색상을 반반씩 섞어서 나타내주었습니다.

CS : Compute Shader

PS : Pixel Shader

지형 Edit - 빌보드

빌보드 모드일때 지형을 클릭하면 빌보드 객체를 동적으로 생성해줍니다.

점 하나를 생성하면 GS 에서 생성한 점을 중심으로 4개의 정점을 생성해주고 텍스처를 입혀줍니다.

GS에서 정점들이 카메라 방향을 바라볼 수 있게 생성해주었습니다.

빌보드 높이 설정

- GS에서 보내준 정점을 월드 위치 -> 해상도 상의 좌표 -> 텍스처 좌표로 변환하고 현재 텍스처 좌표에 해당하는 높이값을 지형의 높이 Texture Buffer 에서 샘플링해서 가져옵니다.
- 가져온 높이값을 더해주면 빌보드가 지형의 높이에 맞게 생성됩니다.

기타

- 빌보드 설치 -> 지형 높이 변경 과 지형 높이 변경 -> 빌보드 설치 둘 다 지형의 높이를 반영하여 빌보드가 생성되지만 전자의 방식이 더 정확하게 생성됩니다.

이유 : 지형의 높이를 먼저 변경하고 설치를 하면 이전의 마우스 Picking 오차와 마찬가지로 설치하고 싶은 곳과 실제 설치 되는 곳의 오차 차이로 인해 정확하지 않을 수도 있습니다.

좌표 변환

```
float2 heightTex;
heightTex.x = abs(-terrainScale - input[0].pos.x) / (terrainScale * 2);

float tex = terrainScale * (terrianHeight / terrainWidth);
float cpos = input[0].pos.z * (terrianHeight / terrainWidth);
heightTex.y = (tex - cpos) / (tex * 2.0);
heightTex.y = 1 - heightTex.y;
```

GS : Geometry Shader

지형 Edit - Lake

지형의 밑에 Lake 를 배치했고 지형의 높이가 낮은 지역에 나타날 수 있게 했습니다.

Lake 의 높이가 지형의 높이보다 낮은 정점은 HS 에서 tess 를 0 으로 설정해서 그려주지 않게 설정했고 PS 에서는 픽셀단위로 검사해서 clip 해주었습니다.

Lake 의 색은 반사행렬을 이용하여 반사되는 물체를 그려준 렌더타겟의 색과 투명도를 적용하여 Lake 내부까지 볼 수 있도록 설정했습니다.

```
float2 reflectPos;

reflectPos.x =
    input.reflectSamplingPos.x / input.reflectSamplingPos.w * 0.5 + 0.5;
reflectPos.y =
    -input.reflectSamplingPos.y / input.reflectSamplingPos.w * 0.5 + 0.5;

refractPos.x = input.posProj.x / input.posProj.w * 0.5 + 0.5;
refractPos.y = -input.posProj.x / input.posProj.w * 0.5 + 0.5;

float4 reflectColor = g_reflect.Sample(linearWrapSampler, reflectPos);

output.pixelColor = reflectColor;
output.pixelColor.w = lakeAlpha;
```

```
// 물결
output.posWorld.y += sin(output.posWorld.x *globalTime * 0.25) * waveStrength;
```

Lake 의 DS 에서 sin 함수를 이용하여 간단한 물결효과도 넣어주었습니다.

HS : Hull Shader
DS : Domain Shader

물체 Edit

물체 Edit 에선 다양한 게임 오브젝트들을 생성 , 삭제 , 배치 , 수정을 할 수 있습니다.

생성
- 오브젝트를 동적으로 생성하여 List 형식으로 관리해서 생성 , 삭제를 쉽게 할 수 있게 구현했습니다.

삭제
- 리스트를 순환하면서 현재 picking 된 모델과 일치하면 리스트에서 삭제합니다.
- 물체의 경계상자가 마우스와 충돌하면 현재 picking 중인 물체인 것을 표시하기 위해 초록색으로 나타냈습니다.

배치
- Picking 된 물체를 마우스 오른쪽 클릭을 하면 이동가능한 상태가 되어서 마우스를 따라다닙니다.
- 오른쪽 클릭을 다시 하기 전까지 picking 된 물체는 바뀌지 않고 고정됩니다.
- 오른쪽 클릭을 한번 더 하게 되면 그 위치에 물체를 배치 할 수 있습니다.

수정
- 물체를 90도씩 회전 , 사이즈 변경이 가능합니다.
- 물체의 경계상자의 사이즈 변경이 가능하게 물체 내부 클래스에서 rotation , scale 값을 따로 저장하고 저장한 값을 이용하여 경계상자도 실시간으로 바뀔 수 있게 적용했습니다.

충돌
- CollisionBoxCreate 버튼을 클릭하면 물체 내부 클래스에서 가지고 있는 경계상자를 충돌 검사 리스트에 추가합니다.
- 충돌 상자로 지정한 물체의 선을 빨간색으로 표시했습니다.
- CollisionBoxDelete 버튼을 클릭하면 충돌 리스트에서 삭제합니다.

날씨 - 눈 , 비

눈 과 비 파티클을 다양한 방식으로 구현하고 각 방식의 속도 비교를 하기 위해 서로 다른 방식으로 구현해보았습니다.

눈

- Instancing 방식으로 구현
- 초기화 시에 각 인스턴스의 위치 , 속도 , 크기를 랜덤으로 지정하여 행렬을 만들어주었습니다.

```
for (int i = 0; i < m_snow.m_snowCount; i++) {

    SnowInstance snow;

    Vector3 startPos = Vector3(position(gen), 10.f, position(gen));

    snow.instanceWorld = Matrix::CreateScale(Vector3(scale(gen))) *Matrix::CreateTranslation(startPos);
    snow.velocity = velocity(gen);

    snowInstances.push_back(snow);
}
```

각 눈들이 아래까지 내려왔을때 다시 처음 위치로 되돌리기 위해서 VS 에서 나머지 연산을 이용했습니다.

```
float v = (globalTime * input.velocity)% 11.0f;
input.posModel.y -= v;
```

비

- Structured Buffer 를 이용한 CS , GS 로 구현
- CS 에서 각 파티클의 높이를 계산해주고 GS에서 정점을 추가하여 아주 얇은 직사각형으로 비 모양을 만들었습니다.

```
if (output[dtID.x].pos.y < -1.0) {

    output[dtID.x].pos.y = 10.0;
}

output[dtID.x].pos.y -= output[dtID.x].velocity;
```

CS 에서 위치 조절

비교

- 같은 양의 파티클을 생성했을때 CPU 에서 GPU 로 보내는 데이터를 최소화하기 위해 VS 에서 위치 계산을 했음에도 Structured Buffer 를 이용한 방식이 속도가 훨씬 빨랐습니다.

이유 : 인스턴싱은 메모리 사용량은 줄지만 물체당 API 추가부담은 줄지 않기 때문

그림자

햇빛

Directional Light 조명을 설치해서 낮 과 밤 , 그림자를 표현했습니다.

햇빛의 기준에서 바라볼때 오브젝트들의 깊이값을 저장하고 오브젝트들을 Render 할때 셰이더에서 BasicLightRadiance 함수를 이용해 빛 연산 , 깊이 판별을 하여 조명에 의한 색상을 결정하고 결정한 색상을 최종적으로 픽셀 색상에 곱해줘서 물체의 색상을 결정했습니다.

BasicLightRadiance 함수

```
float3 BasicLightRadiance(Light Sun, float3 normalWorld, float3 posWorld, float ambientCoefficient, float diffuseCoefficient, float specularCoefficient, Texture2D SunShadowMap)
{
    float3 sunDir = -Sun.direction;
    float3 r = reflect(-sunDir, normalWorld);
    float3 viewDir = eyeWorld - posWorld;

    float ambient = ambientCoefficient;
    float diffuse = dot(normalWorld, sunDir) * diffuseCoefficient;
    float specular = pow(max(dot(r, viewDir), 0.0), 32) * specularCoefficient;

    float shadowFactor = 1.0;

    // depth 비교
    float4 posDepth = mul(float4(posWorld, 1.0), Sun.viewProj); // 해 기준 depth
    posDepth.xyz /= posDepth.w;
    float2 Suntexcoord = posDepth.xy;
    Suntexcoord.x = Suntexcoord.x * 0.5 + 0.5;
    Suntexcoord.y = -Suntexcoord.y * 0.5 + 0.5;

    int width, height, numMips;
    SunShadowMap.GetDimensions(0, width, height, numMips);
    float dx = 1.0 / (float)width;
    float percent = 0.0;
    const float2 offsets[9] = { ... }

    [unroll] for (int i = 0; i < 9; i++) {
        percent += SunShadowMap.SampleCmpLevelZero(shadowCompareSampler, Suntexcoord.xy + offsets[i] * dx, posDepth.z - 0.0001).r;
    }

    shadowFactor = percent / 9.0;

    return (ambient + specular) * Sun.radiance + diffuse * Sun.radiance * shadowFactor;
```

물체마다 ambient , diffuse , specular 계수를 다르게 설정할 수 있습니다.
프로젝트를 할때는 모든 물체의 계수를 동일하게 설정하고 진행했습니다.

그림자는 PCF 기법을 이용했고 부드러운 효과를 주기 위해 샘플링 개수를 9개로 늘렸습니다.

문제점

조명과 물체 사이의 거리가 멀면 그림자가 잘 나타나지 않거나 피터패닝 현상이 일어납니다.

그림자 맵의 해상도를 높이고 그림자가 깨지는 것을 방지하는 Bias 값을 적당히 잘 조절하면 그림자가 선명하게 나타나고 피터패닝 현상도 완화됩니다.

Bias 값을 해상도에 비해 너무 작게 설정해도 그림자가 깨지는 현상이 일어나기 때문에 해당 해상도에 맞는 Bias 를 사용해야 합니다.

충돌 처리 – 앞 충돌

캐릭터의 경계 상자와 충돌 상자 리스트에 들어있는 모든 충돌 상자와 충돌 검사를 하여 나온 dist 의 값이 캐릭터의 경계상자의 Extent.z 의 값보다 작거나 같다면 충돌한 것으로 설정했습니다.

충돌을 하면 캐릭터가 향하는 방향으로 가지 못하도록 설정했습니다.

충돌을 했음에도 대각선으로 이동하면 앞은 가로막혀 있고 옆으로 움직일 수 있게 두 방향으로 나누어서 충돌 검사를 했습니다.

```
for (auto &i : m_collisionBox) {
    if (cRay1.Intersects(i->m_box, dist)) {
        if (dist <= m_character->extent.z) {
            frontUp = Vector3(0.0f, 0.0f, 0.0f);
            break;
        }
    } else if (cRay2.Intersects(i->m_box, dist)) {
        if (dist <= m_character->extent.z) {
            frontRight = Vector3(0.0f, 0.0f, 0.0f);
            break;
        }
    }
}

for (auto &i : m_collisionObj) {
    if (cRay1.Intersects(i->m_box, dist)) {
        if (dist <= m_character->extent.z) {
            frontUp = Vector3(0.0f, 0.0f, 0.0f);
            break;
        }
    } else if (cRay2.Intersects(i->m_box, dist)) {
        if (dist <= m_character->extent.z) {
            frontRight = Vector3(0.0f, 0.0f, 0.0f);
            break;
        }
    }
}

front = frontUp + frontRight;
```

충돌 상자를 하나의 리스트에 모아서 관리하지 않고
m_collisionBox , m_collisionObj 두개의 리스트로 나누어서 관리한 이유

m_collisionBox 는 주로 지형과 관련된 충돌 상자리스트 이고
m_collisionObj 는 물체 객체를 넣어놓은 리스트 입니다.

둘을 나눈 이유는 지형과 물체의 경계상자를 생성 , 삭제할때 헷갈리지 않고 쉽게 하고 물체의 충돌 상자를 삭제할때 리스트 안에서 비교할때 pick 된 물체를 기준으로 비교하여 삭제를 하게 설정했기 때문입니다.

속도면에서는 차이가 별로 나지 않겠지만 코드의 양이 늘어난다는 단점이 있는것 같습니다.

충돌 처리 - 높이 충돌 (GPU)

캐릭터가 지형의 높이에 따라 자유롭게 움직일 수 있도록 지형과의 충돌도 필요했습니다.

높이 충돌은 CPU 에서 하는 방식과 GPU 에서 하는 방식 두 가지를 구현했습니다.
각 충돌은 장단점이 있고 지형에 따라 알맞은 방식을 사용할 수 있습니다.
캐릭터에 중력도 작용합니다.

GPU 방식

- 캐릭터를 렌더링할때 VS 에서 높이 Texture Buffer 에서 현재 캐릭터의 위치에 해당하는 높이 값을 샘플링 해와서 정점에다가 더해줍니다.

(일반 오브젝트들은 같은 쉐이더를 쓰는데 캐릭터의 경우만 ifdef 로 실행할 수 있도록 정의했습니다.)

장점

- GPU 에서 높이 충돌을 처리하므로 속도에 대한 부담이 없고 샘플링만 해오면 되기 때문에 매우 간단합니다.

단점

- GPU 에서 높이 위치 값을 더해주기 때문에 높이 차이가 많이 나는 절벽같은 경우 캐릭터의 높이가 갑자기 확 바뀌어서 부자연스러운 느낌이 있습니다.
- 카메라가 캐릭터의 위치에 고정되어 움직이는데 GPU 에서 하면 캐릭터의 높이에 따라 카메라의 시점을 바꿀수가 없습니다.

```
float height = TerrainPass.SampleLevel(linearWrapSampler, CTex, 0).y;

if (characterWorldPos.y - extent < height)
    pos.y += TerrainPass.SampleLevel(linearWrapSampler, CTex, 0).y;
```

충돌 처리 - 높이 충돌 (CPU)

CPU 방식

- 캐릭터의 위치를 상수버퍼로 보내주고GetheightGPUtoCPU 함수를 통해서 현재 위치에 해당하는 높이 값을 CPU로 가져와서 CPU 에서 캐릭터의 위치값을 변경해줍니다.

GetheightGPUtoCPU 함수

```
float Terrain::GetheightGPUtoCPU(ComPtr<ID3D11DeviceContext>& context)
{
context->CSSetShader(m_getheightCS.Get(), 0, 0);
context->CSSetShaderResources(0, 1, m_terrainPassSRV.GetAddressOf());
context->CSSetUnorderedAccessViews(0, 1,
                                   m_CharacterHeightUAV.GetAddressOf(), NULL);

context->Dispatch(1, 1, 1);
this->ComputeShaderBarrier(context);

context->CopyResource(m_heightStagingBuffer.Get(),
                      m_CharacterHeightBuffer.Get());

float height = -1.0f; // height 값 제대로 나오는지 디버그 확인용으로 0 이 아닌 -1 값으로 지정

 CharacterHeight *pData;
 D3D11_MAPPED_SUBRESOURCE mappedResource;
 context->Map(m_heightStagingBuffer.Get(), 0, D3D11_MAP_READ_WRITE, 0,
              &mappedResource);
 pData = (CharacterHeight *)mappedResource.pData;

 height = pData->height;

 context->Unmap(m_heightStagingBuffer.Get(), 0);

 return height;
}
```

CS 에서 현재 높이값을 Structured Buffer UAV 에 저장하고 staging Buffer 에 복사하고 staging Buffer를 이용해서 CPU로 값을 복사해서 높이값을 반환해줍니다.

다른 방법으로는 현재 Texture Buffer 의 포맷이 R16G16B16A16_FLOAT 이므로 여기에 저장된 값을 그대로 불러오기 위해서는 float32 형식으로 변환 후 copySubResource 함수를 이용해서 부분적으로 픽셀의 값을 가져올 수 있다.

장점

- CPU 에서 현재 위치의 높이값을 알 수 있어서 캐릭터가 높이가 급격하게 변하는 곳을 내려갈때 한번에 확 내려가지 않고 중력에 의해 천천히 내려갈 수 있게 자연스럽게 이동할 수 있습니다.
- 캐릭터의 높이에 따른 카메라의 시점을 원활하게 변경할 수 있습니다.

단점

- CS 생성 , 사용으로 인한 메모리 증가 , GPU 에서 CPU 로 복사해오는 작업으로 인한 속도 저하 등의 문제가 발생합니다.

충돌 처리 – 높이 충돌 정리

CPU 와 GPU 의 공통적인 문제점

높이가 급격하게 변하는 지점에서 아래에서 위로 갈때도 높이가 확 바뀝니다.
➔ 가지 않게 하려면 지형의 충돌 상자를 생성해서 막는 방법이 있습니다.

높이가 급격하게 변하는 지점이 없고 엄청 높은 지점도 없다면 GPU 에서 높이 충돌을 하는 것이 좋고

반대로 높이가 급격하게 변하는 지점이 있거나 지형 자체가 높은 지형이면 CPU 에서 높이 충돌을 하는 것이 좋을 것 같습니다.

최적화 - 프러스텀 컬링 , 그래픽 상 중 하

카메라의 절두체를 생성하여 모든 물체와 절두체 검사를 합니다.

절두체의 6개의 면과 각 물체의 중심사이의 거리를 구해서 물체 클래스 내부에 만들어놓은 경계 구의 반지름보다 크면 절두체 밖에 있는 것이므로 그려주지 않습니다.

```
bool Frustum::FrustumCulling(shared_ptr<Model> obj)
{
    for (auto& i : m_plane)
    {
        Vector3 normal = Vector3(i.x, i.y, i.z);

        if (normal.Dot(obj->m_worldRow.Translation())+ i.w > obj->m_sphere.Radius)
            return false;
    }

    return true;
}
```

그래픽 상 중 하

- 그래픽 상 중 하 기능은 각 옵션마다 절두체의 가장 멀리 있는 면의 범위를 다르게 설정하여 컬링할 영역을 줄이거나 넓힐 수 있습니다.
- 절두체의 영역이 줄어들면 줄어든 만큼의 영역안에 있는 물체를 덜 그릴 수 있으므로 속도가 향상됩니다.

```
if (ImGui::RadioButton("Upper", (int*)&m_graphicState,
                       (int)GraphicState::UPPER))
{
    currentFarZ = 30.0f;
    m_Editcamera.m_farZ = currentFarZ;
    m_globalConstsCPU.graphicLower = false;
}


if (ImGui::RadioButton("Middle", (int*)&m_graphicState,
    (int)GraphicState::MIDDLE))
{
    currentFarZ = 20.0f;
    m_Editcamera.m_farZ = currentFarZ;
    m_globalConstsCPU.graphicLower = false;
}
if (ImGui::RadioButton("Lower", (int*)&m_graphicState,
    (int)GraphicState::LOWER))
{
    currentFarZ = 10.0f;
    m_Editcamera.m_farZ = currentFarZ;
    m_globalConstsCPU.graphicLower = true;
}
```

최적화 - LOD , Clipping

LOD

- 지형의 HS 에서 카메라와의 거리에 따라 LOD 를 적용했고 LOD ON / OFF 와 테셀레이션 레벨도 조절할 수 있도록 하였습니다.

- LOD 를 OFF 하면 거리에 상관없이 모든 패치를 테셀레이션 레벨에 따라 세분화 해줍니다.
- 테셀레이션 레벨에 따라 최대로 얼마나 세분화 할지 조절할 수 있습니다.

```
float dis = length(center - eyeWorld);
float disMin = 1.0f;
float disMax = 10.0f;
float tessLevel = 64.0;

if (!Tessellation) // LOD ON / OFF
    dis = 1;

if (TessellationLevel == 4) // LOD Level
    tessLevel = 64.0;
else if (TessellationLevel == 3)
    tessLevel = 32.0;
else if (TessellationLevel == 2)
    tessLevel = 16.0;
else if (TessellationLevel == 1)
    tessLevel = 8.0;
else
    tessLevel = 0.0;
```

Clipping

그래픽 하 상태일때 멀리있는 지형의 정점은 HS 에서 그려주지 않게 했습니다.

```
if (graphicLower)
    tess = tessLevel * smoothstep(disMin, disMax, disMax - dis);
else
    tess = tessLevel * smoothstep(disMin, disMax, disMax - dis) + 1.0;
```

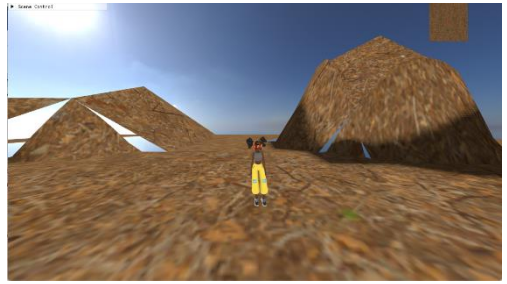
거리가 먼 패치의 tess 값이 0 이 되면 테셀레이션 단계가 실행되지 않고 DS 이후의 셰이더들도 실행되지 않습니다.

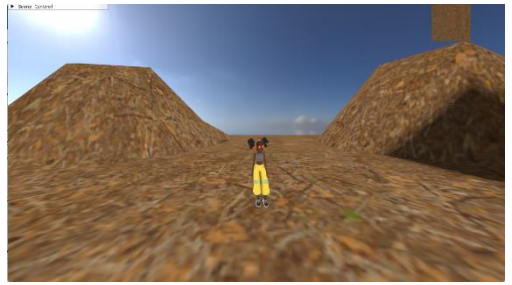
Lake 의 HS 에서도 똑같이 적용합니다.

LOD 적용시 발생하는 문제

– Crack 현상

- 지형의 각 패치가 인접 패치와의 **LOD** 단계가 다를때 메시간의 연결이 매끄럽지 못해서 발생하는 균열 현상





해결 방법

- 현재 패치와 상하좌우로 인접한 4개의 패치의 LOD 단계를 검사해서 각 인접한 패치의 LOD 단계와 현재 패치의 LOD 단계중 큰 LOD 단계를 각 정점에 적용시켰습니다.

```
// Crack 현상 해결
output.edges[0] = max(tess, GetTess(center0, tessLevel, disMin, disMax));
output.edges[1] = max(tess, GetTess(center1, tessLevel, disMin, disMax));
output.edges[2] = max(tess, GetTess(center2, tessLevel, disMin, disMax));
output.edges[3] = max(tess, GetTess(center3, tessLevel, disMin, disMax));
```

– Popping 현상

- 거리에 따라 **LOD** 단계가 갑작스럽게 변하면 그래픽이 튀어보이는 현상

해결 방법

- 거리에 따른 **tess** 의 값을 **smoothstep** 함수를 이용하여 보간해주어서 **Popping** 현상을 줄일 수 있도록 했습니다.

이외에 최적화 할 수 있는 방법

1. 상수버퍼 분류

- 매 프레임 변하는 상수버퍼와 변하지 않는 상수버퍼를 분류하여 CPU -> GPU 로 보내는 상수버퍼의 양 최소화

2. Compute Shader

- 공용캐시 메모리 사용 -> 성능 향상

(예 : 텍스처의 각 픽셀값을 공용캐시 메모리에 담아놓고 사용하면 샘플링해오는 것보다 성능이 향상)

3. SRV 배열로 설정하여 서술자 개수 최소화

4. 렌더링 파이프라인

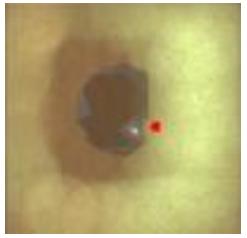
- 물체 렌더링과 Compute Shader 전환에는 추가 부담이 있기 때문에 CS 몰아서 수행 후 렌더링 몰아서 수행
- 테셀레이션 그리기 호출들도 묶어서 처리 (Terrain , Lake)

기타 기능 - 미니맵 , 방향키뷰어 (UI)

UI

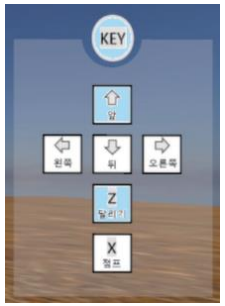
- UI 를 그려주기 위해 UI 카메라를 생성하여 UI로 쓰일 물체는 UI 카메라의 Orthographic Projection 행렬을 이용했습니다.

미니맵



- 지형의 위에서 위치를 설정하여 그 위치를 기준으로 지형과 Lake 를 위에서 내려다 봤을때의 모습을 렌더타겟에 저장했고 저장한 렌더타겟을 UI 를 렌더할때 텍스처로 사용했습니다.
- 위에서 내려다 봤을때 크기가 딱 맞게 시야각 , Far_Z 값을 설정해서 행렬을 생성해주었습니다.
- 플레이어의 위치는 미니맵 상의 빨간점으로 나타냈고 지형의 어디쯤에 위치하여 있는지 알 수 있게 표시했습니다.

방향키뷰어




```
enum KEY_TYPE
{
    KEY_BOARD,
    KEY_UP,
    KEY_DOWN,
    KEY_RIGHT,
    KEY_LEFT,
    KEY_X,
    KEY_Z,

    KEY_END
};
```

색깔 처리

```
if (push == 1)
{
    if (color.r + color.g + color.b >= 2.8)
        color = float4(0.4, 0.7, 0.9, 1.0);
}
```

- 캐릭터의 어떤 행동 입력 키를 눌렀는지 키뷰어를 통해 볼 수 있습니다.
- 키를 누르면 키뷰어에 불빛이 들어오게 했습니다.
- KeyViewer 클래스에서 여러 개의 UI 클래스를 가질 수 있게 했습니다.
- K 키를 누르면 키뷰어를 On / Off 할 수 있습니다.
- 한번 누를때 여러 번 눌리는 것을 방지하기 위해 누르고 일정시간 뒤에 눌러야 눌린 것으로 인정되게 설정했습니다. (On / Off 시)
- 키를 눌렀을때 PS 에서 흰색과 가까운 색만 색을 변경해주었습니다.

기타 - 캐릭터 실루엣 외곽선 효과 (투과 기능)

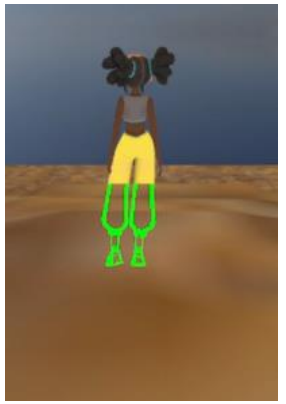
스텐실 버퍼를 이용하여 캐릭터가 지형 뒤에 가려져도 확인 할 수 있는 실루엣 On / Off 기능을 구현했습니다.

렌더링 순서 : 지형 -> 캐릭터 -> 실루엣

지형과 캐릭터의 깊이 test 를 켜주고 지형은 스텐실 값 1 , 캐릭터는 스텐실 값 2 로 먼저 렌더링 해줍니다.

실루엣을 렌더링할때 깊이 test 를 꺼줘서 항상 그려질 수 있게 해주었고 스텐실 값이 1인 곳에만 렌더링을 해줄 수 있게 했습니다.

```
// 실루엣
AppBase::SetPipelineState(m_drawAsWire ? Graphics::terrainWirePSO
                                       : Graphics::terrainSolidPSO);
AppBase::SetGlobalConsts(m_globalConstsGPU);
m_context->OMSetDepthStencilState(Graphics::maskDepth.Get(), 1.0f);
m_terrain.Render(m_context);

if (EditScene) {
    AppBase::SetPipelineState(Graphics::skinnedSolidPSO);
    m_context->OMSetDepthStencilState(Graphics::maskDepth.Get(), 2.0f);
    m_character->Render(m_context, m_terrain.m_terrainPassSRV);
}

if (silhouette)
{
    if (EditScene) {
    AppBase::SetPipelineState(Graphics::skinnedSilhouettePSO);
    m_context->OMSetDepthStencilState(Graphics::drawMasked.Get(), 1.0f);
    m_character->Render(m_context, m_terrain.m_terrainPassSRV);
    }
}
```



캐릭터의 실루엣을 Rim 공식을 활용하여 구현했고 외곽선의 굵기도 조절할 수 있게 했습니다.

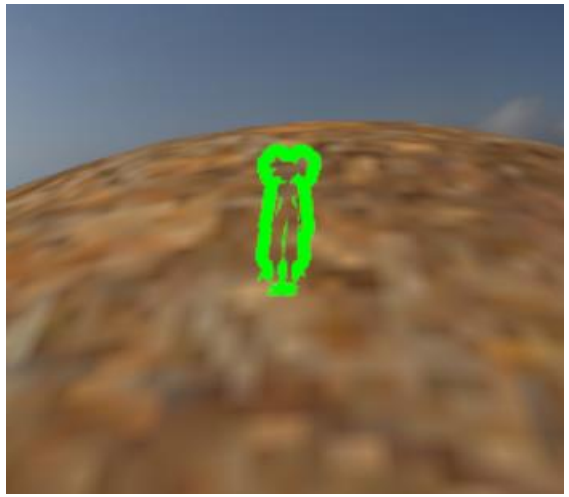
```
PixelShaderOutput output;

output.pixelColor = float4(0.0, 1.0, 0.0, 1.0);

// Rim
float rim = (1.0f - dot(input.normalWorld, normalize(eyeWorld - input.posWorld)));

float rimC = 1.0 - thickness;

if (rim <= rimC)
    output.pixelColor.w = 0.0;

return output;
```

* thickness 로 굵기 조절

기타 - 캐릭터 실루엣 외곽선 효과 (투과 기능)

다른 방법

캐릭터의 스텐실 옵션에서 깊이 테스트를 실패했을때 스텐실 값 유지가 아니라 replace 로 바꿔서 덮어버리고
실루엣의 크기를 VS 에서 normal 방향으로 조금 키워서 렌더링하면 외곽선 효과를 나타낼 수 있습니다.



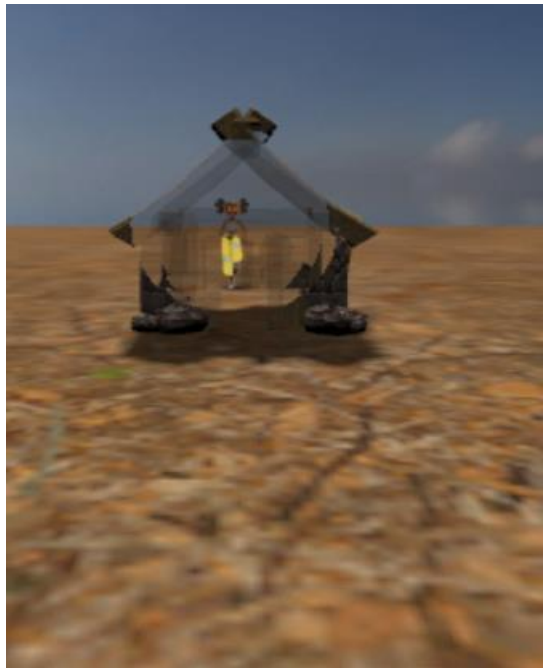
기타 - 투명도를 이용한 투과 기능

캐릭터가 물체 뒤에 가려졌을때 물체의 투명도를 이용하면 캐릭터를 보이게 할 수 있습니다.

View 시점을 기준으로 물체와 캐릭터의 Z 값을 비교하여 물체의 픽셀이 캐릭터보다 앞에 있다면
일정 범위내의 픽셀들을 살짝 투명하게 그려줬습니다.

투명도를 적용할 물체는 가장 나중에 그려줬습니다.

캐릭터가 벽에 의해서 가려서 안보인다면 실루엣 효과나 투명도를 이용하여 투과하여 볼 수 있게 만들 수 있습니다.

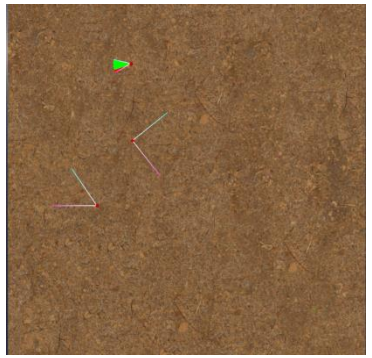


기타 - 터닝 포인트

캐릭터의 이동 방향과 카메라의 시점을 변경시킬 수 있는 지점을 설치할 수 있습니다.

GS 에서 영역의 범위를 나타내주는 선을 그렸습니다. (On / Off 가능)
- 지형범위는 지형 PS 에서 따로 색칠

(빨간색 : 시작지점 , 초록색 : 끝나는지점)



구현 방법

매프레임 캐릭터와 제일 가까운 지점을 찾고 벡터의 내적을 이용해서 캐릭터가 영역 안에 들어와 있는지 확인합니다.

영역안에 들어오면 지점과 캐릭터의 벡터의 수직인 벡터의 방향으로 카메라의 방향을 바꿔줍니다. -> 캐릭터는 카메라의 방향대로 이동하므로 캐릭터의 이동방향도 바뀝니다.

시작지점을 반대로 설정한 지점이면 카메라의 방향도 반대로 설정

영역 검사

```
bool TurningPoint::InArea(const Vector3 characterPos) {

    // 모든 점과 거리비교해서 최소인점 찾기
    float dist = 100.0f;
    shared_ptr<Point> p; // 영역 검사할 점

    for (auto& i : m_points)
    {
        Vector3 Vec = (characterPos - i->pos);
        Vec.y = 0.0f;

        float length = Vec.Length();

        if (length < dist)
        {
            dist = length;
            p = i;
        }
    }

    // 영역 검사
    Vector3 CVec = Vector3(characterPos - p->pos);
    CVec.y = 0.0f;
    CVec.Normalize();
    float cosTheta = CVec.Dot(p->centerVector);
    float theta = acos(cosTheta);

    if (dist <= p->radius && theta <= p->angle * 0.5f) // 영역안에 있음
    {
        m_currentPoint = p;
        return true;
    }
    else
    {
        m_currentPoint = nullptr;
        return false;
    }
}
```

기타 - SSAO

SSAO 과정

1. 법선 렌더타겟 , 깊이맵 세팅

- 차폐도를 계산하기 위해서는 픽셀마다 법선 정보와 깊이 정보가 필요하므로
- SSAO 를 적용할 물체들을 렌더링할때 법선 렌더타겟 , 깊이 버퍼를 연결해줍니다.
  (법선 시야공간으로 변환)

2. 차폐도 계산해주는 Compute Shader 세팅

- 각 픽셀마다 저장한 법선 렌더타겟과 깊이맵을 이용하여 Compute Shader 에서 차폐도를 계산해줍니다.
- SSAO 맵의 해상도는 기존 해상도의 절반으로 내려도 별 차이가 없으므로 성능 차이에 따라 기존 해상도를 쓸지 절반을 쓸지 결정해 줄 수 있습니다.

3. 기준점 P 찾기

- 차폐도를 계산하기 위해서 해야 할 일은 기준점 P 를 찾아줘야 합니다.
- 현재 투영텍스처로 깊이맵에서 투영 z 값을 샘플링 해옵니다.
- 투영텍스처 x , y 값 , 샘플링 한 z 값을 기준으로 시야 공간 좌표로 변환하면 차폐도를 계산해줄 기준점 P 를 구할 수 있습니다.

4. 무작위 표본의 점 구하기

- 무작위 표본의 벡터를 이용하여 14개의 벡터 샘플을 이용하여 시야 공간에서의 표본 점 Q 를 구합니다.
  (무작위 벡터는 임의로 Vector3 (0 , 0, -1) 로 설정하였고 14개의 벡터 샘플은 루나 책에 나와있는 방법을
  사용했습니다. -> 어떤 무작위 벡터가 나와도 균등하게 벡터 샘플 가능)

SSAO ComputeShader

```
#include "Common.hlsli"

cbuffer OffsetVectors : register(b0){
    float4 offSets[14];
    float ssaoRadius;
    float dist;
    float ssaoStart;
    float ssaoEnd;
};

Texture2D depthMap : register(t0);
Texture2D normalMap : register(t1);

RWTexture2D<float4> ssaoUAV : register(u0);

static const float3 ssaoVec = float3(0.0, 0.0, -1.0);
static const int SampleCount = 14;
static const float width = 1280.0;
static const float height = 720.0;

float CalculateOcclusion(float distZ) {
    float occlusion = 0.0;
    if (distZ > dist) {
        float fadelength = ssaoEnd - ssaoStart;

        occlusion = saturate((ssaoEnd - distZ) / fadelength);
    }
    return occlusion;
}

float4 TexcoordToView(float2 texcoord) {
    float4 posProj;

    posProj.xy = texcoord * 2.0 - 1.0;
    posProj.y *= -1;
    posProj.z = depthMap.SampleLevel(linearClampSampler, texcoord, 0).r;
    posProj.w = 1.0;

    float4 posView = mul(posProj, invProj);
    posView.xyz /= posView.w;

    return posView;
}
```

기타 - SSAO

5. 차폐점 구하기

- 각 Q 점을 시야 공간에서 투영좌표로 바꾸고 투영텍스처로 다시 깊이맵에서 가장 가까운 Z 값을 가져옵니다.
- 이 Z 값으로 또 가장 가까운 시야공간의 점을 구합니다.
- 이 점이 차폐판정을 하여 차폐도를 결정해 줄 점이 됩니다.

6. 차폐 판정으로 차폐도 계산

- 기준점 P 와 차폐점 Q 의 깊이값 차이에 따른 차폐도를 계산해줍니다.
- 또 법선 렌더타겟에서 현재 픽셀에 해당하는 법선값을 샘플링 해와 Q - P 벡터의 내적값을 이용해 너무 멀면 가리지 않는 것으로 판정해줍니다.

7. 계산한 차폐도 맵 Blur 해주기

- 단순히 다음 Compute Shader 에서 주변 9개의 값들을 평균내주었습니다.

8. SSAO 적용

- 빛의 연산을 통해 그림자를 구해주는 함수 내부에서 SSAO 값을 샘플링해와서 곱해주었습니다.

```
[numthreads(32, 32, 1)]
void main( uint3 dtID : SV_DispatchThreadID )
{
    // 현재 기준점 찾기 ( 텍스쳐 좌표 -> 깊이값 추출 -> 투영 좌표 -> 시야공간으로 변환 )
    float2 texcoord = dtID.xy;
    texcoord.x /= width;
    texcoord.y /= height;

    float3 posView = TexcoordToView(texcoord).xyz;

    // 법선 찾기
    float3 n = normalMap.SampleLevel(linearWrapSampler, texcoord, 0).xyz;

    float ssaoSum = 0.0;

    // 차폐점 찾기
    for (int i = 0; i < SampleCount; i++)
    {
        float3 offset = reflect(offSets[i].xyz, normalize(ssaoVec));

        float flip = sign(dot(offset, n));

        float3 ssaoPoint = posView + flip * ssaoRadius * offset;

        // 차폐점 투영텍스쳐 좌표
        float4 ssaoProj = mul(float4(ssaoPoint, 1.0), proj);
        float2 ssaoTex = ssaoProj.xy;
        ssaoTex.xy /= ssaoProj.w;
        ssaoTex.x = ssaoTex.x * 0.5 + 0.5;
        ssaoTex.y = -ssaoTex.y * 0.5 + 0.5;

        // 차폐점에서의 최소 깊이값 -> 시야 공간 변환
        float3 ssaoView = TexcoordToView(ssaoTex).xyz;

        // 차폐판정
        float distZ = posView.z - ssaoView.z;
        float dp = max(dot(n, normalize(ssaoView - posView)), 0.0);
        float occlusion = dp * CalculateOcclusion(distZ);

        ssaoSum += occlusion;
    }
```

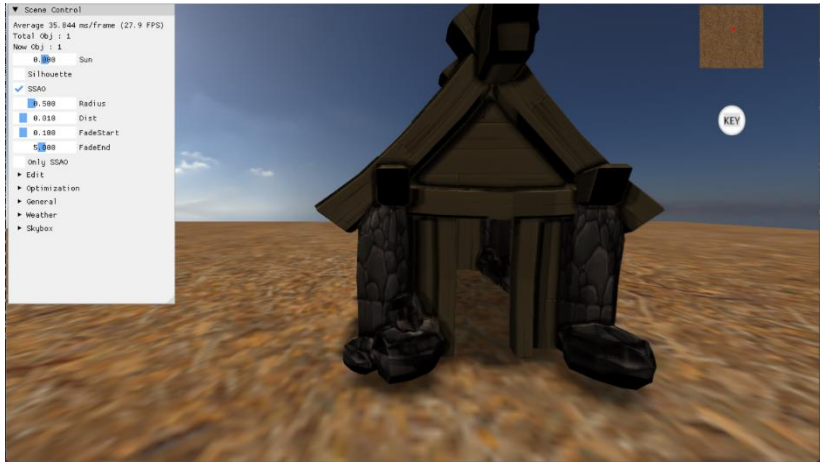
```
    ssaoSum /= SampleCount;
    float access = 1.0 - ssaoSum;
    access = saturate(pow(access, 2.0f));

    float4 ssao = float4(access, access, access, 1.0);

    ssaoUAV[dtID.xy] = ssao;
}
```

기타 – SSAO 결과

SSAO 맵

SSAO 적용 전

SSAO 적용 후



기타 - Deferred Rendering

구현한 Deferred Rendering 방식

한 물체를 렌더링할때 셰이더에서 Deferred 와 Forward 방식을 동시에 렌더링했습니다.

순서

- 모든 물체를 4개의 렌더타겟을 세팅하고 렌더링합니다.

```
struct PixelShaderOutput
{
    float4 pixelColor    : SV_Target0; // 기존 포워드 방식으로 색상값 저장
    float4 position      : SV_Target1; // 디퍼드 방식을 사용하는 물체면 추가 정보 저장
    float4 normal        : SV_Target2; // 디퍼드 방식을 사용하는 물체면 추가 정보 저장
    float4 depth         : SV_Target3; // 깊이 값 저장
};
```

- 셰이더 매크로를 이용하여 Deferred 방식이면 추가 정보를 저장하고 Deferred 방식이 아니면 추가 정보를 0 값으로 저장합니다.

```
    output.position = float4(0.0, 0.0, 0.0, 0.0);
    output.normal = float4(0.0, 0.0, 0.0, 0.0);

#ifdef DEFERRED

    output.position = float4(input.posWorld, 1.0);
    output.normal = float4(input.normalWorld, 1.0);
    float depth = input.posProj.z;
    depth /= input.posProj.w;
    depth = 1 - depth;
    output.depth = float4(depth,  depth,  depth, 1.0);


#endif
```

- 후처리 셰이더에서 저장한 추가 정보의 w 값으로 Deferred 인지 Forward 인지 구분하여 Deferred 방식이면 조명 연산을 수행하여 기존 색상값에 적용시킵니다. (그림자 포함)

```
// deferred ( position 의 w 값으로 deferred 인지 forward 인지 구분 )

float3 color = RenderTex.Sample(linearWrapSampler, input.texcoord).rgb;
float4 position = positionTex.Sample(linearWrapSampler, input.texcoord);
float4 normal = normalTex.Sample(linearWrapSampler, input.texcoord);

if (position.w  == 1.0)
{
    float3 sunRadiance = BasicLightRadiance(Sun, normal.xyz, position.xyz, aC, dC ,sC, SunShadowMap);
    color = color * sunRadiance;
}
```

장점

- 조명이 많다면 물체마다 한번의 드로우 콜로 많은 조명연산을 줄여서 속도를 향상시킬 수 있습니다.
  (조명 수 x 물체 수 -> 물체 수 + 조명 수)
- 물체의 투명도 적용 가능

단점

- 모든 렌더타겟의 포맷이 동일해야 제대로 그려집니다.
- 렌더타겟의 resolve 해주는 연산의 수가 증가합니다.

기타 - Deferred Rendering 출력

Position Normal Depth

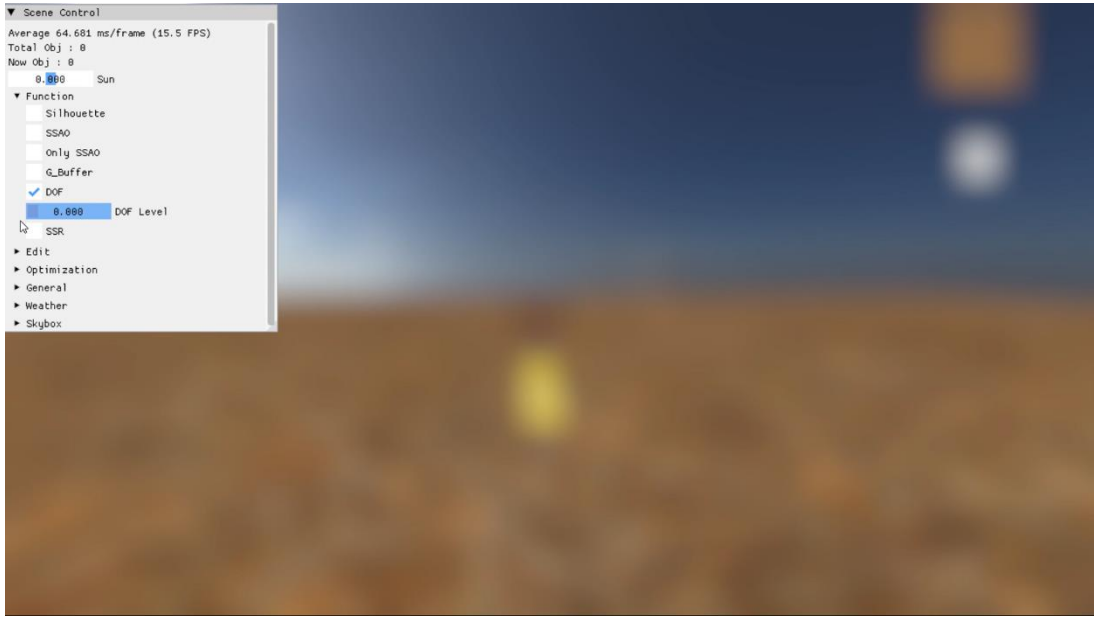
Position Normal Depth

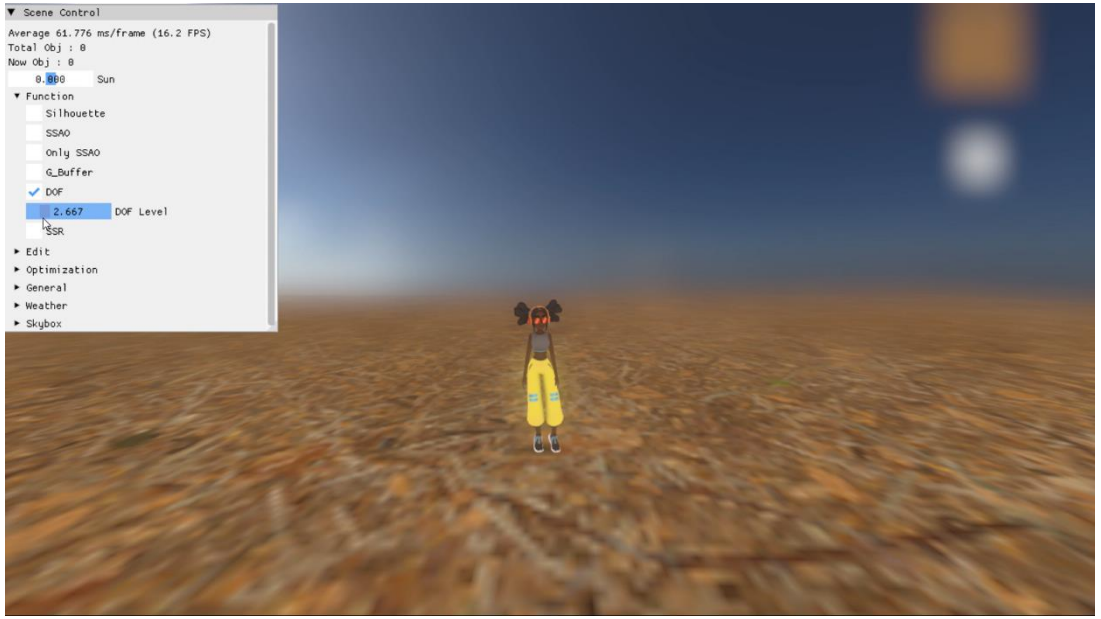
기타 - Depth Of Field

DOF 기능을 켜주면 거리에 따라 Blur 해준 이미지와 원래 이미지를 섞어줍니다.

```
float strength = 0.0;
if (dof)
    strength = pow(depthTex.Sample(linearWrapSampler, input.texcoord).r, dofLevel);

float3 combined = (1.0 - strength) * color0 + strength * color1;
```

DOF Level 에 따라 Blur 거리 조절





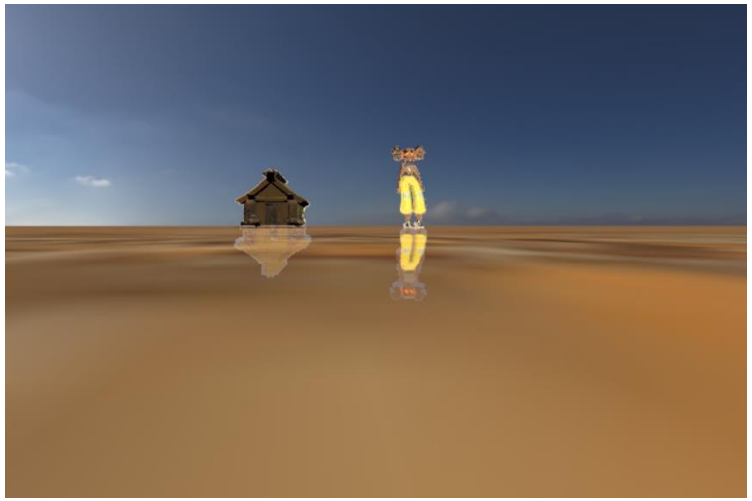
기타 - Screen Space Reflection

원리

현재 픽셀에 해당하는 View 공간 기준의 좌표에서 반사벡터를 이용하여 Ray Marching 을 통해 깊이 값을 비교하여 반사되는 색상을 가져옵니다.

Ray Marching 의 2가지 방법

1. View 공간에서 Ray를 계산



원근 오류가 발생할 수 있고 반사되는 곳이 이상하게 그려집니다. (특정 시점에서만 잘 그려집니다.)
또한 같은 곳이 여러 번 그려지는 현상도 나타납니다.

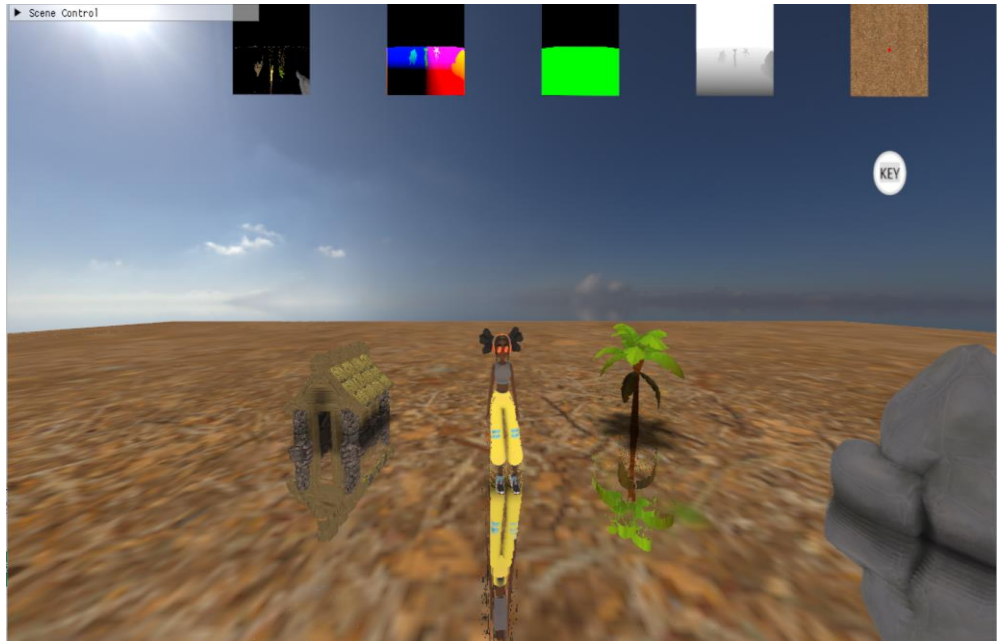
2. DDA 알고리듬을 이용하여 화면 공간에서 Ray를 계산

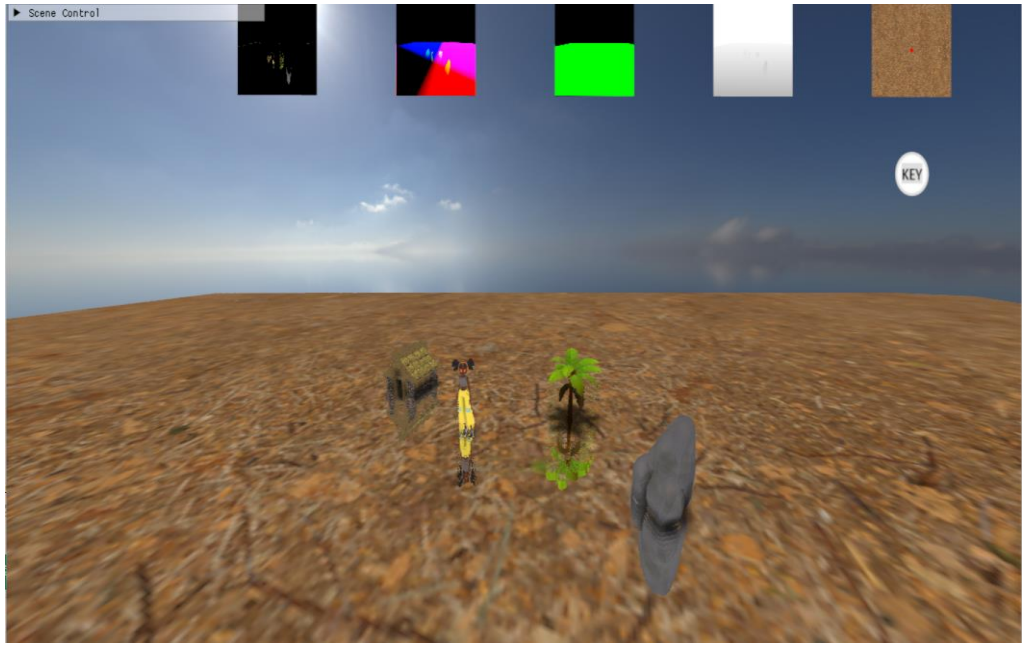
반사되는 곳이 이상하게 그려지고 여러 번 덧붙여서 그려지는 현상도 나타납니다.

기타 - Screen Space Reflection 개선사항

기존 Normal : invert , transpose 한 World 행렬 -> View 행렬

변경 Normal : invert , transpose 한 World 행렬 -> invert, transpose 한 View 행렬





문제점 : 캐릭터의 반사가 원하는 대로 잘 이루어지지 않고 있고 물체와 반사된 물체 사이에도 노이즈가 조금 생깁니다.