Temporal Anti-Aliasing

목차

Temporal Anti-Aliasing 이란

Tmporal Anti-Aliasing(이하 TAA)은 공간축선 안티에일리어싱(Spatial anti-aliasing)의 일종으로 과거 프레임의 이미지를 현재 프레임의 이미지에 결합하여 이미지의 계단 현상을 제거하는 기법입니다. TAA는 다른 AA기법들과 비교했을 때 적은 비용을 사용하여 안티에일리어싱 효과를 얻을 수 있다는 장점이 있지만 이전 프레임의 이미지와 결합하는 과정에서 흐릿한 이미지를 얻게 되는 경우가 있습니다. (이를 해결하기 위해서 추가로 Sharpen 필터를 적용하는 경우가 있습니다.) 여기서는 간단한 TAA의 구현 방식을 살펴보도록 하겠습니다.

구현 방식

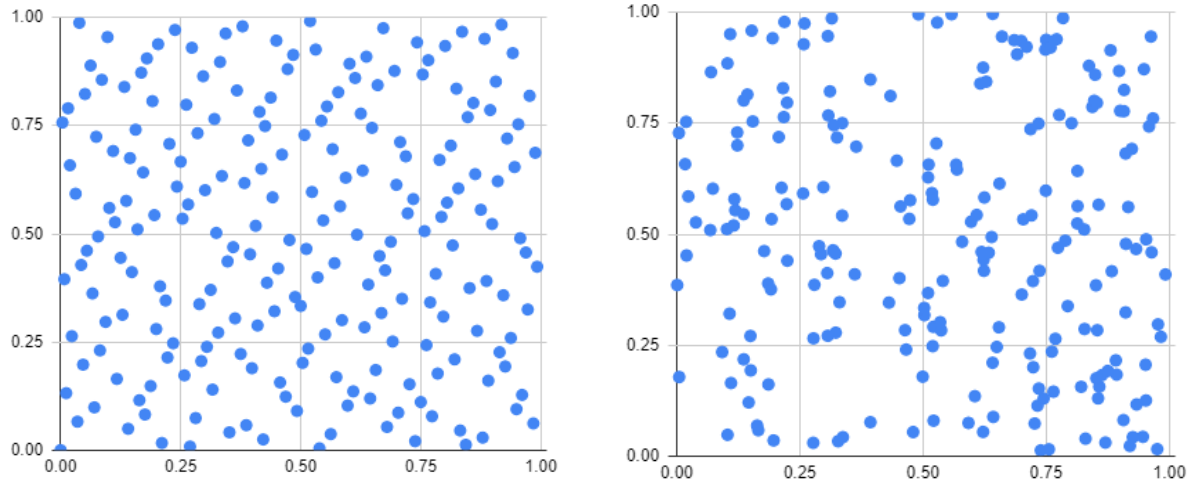
TAA는 다음과 같은 순서로 구현됩니다.

1. 매 프레임에 씬을 그릴 때마다 스크린 평면의 각기 다른 방향으로 위치를 살짝 이동하여 씬을 그립니다. (Jittering 적용)
2. 이렇게 렌더링 된 현재 프레임의 이미지와 이전 프레임들의 이미지(= History buffer)를 조합합니다. (혼합)
3. 조합한 이미지를 Frame Buffer에 복사합니다.

이후로 각 단계를 좀 더 자세히 살펴보도록 하겠습니다.

할톤 수열 (Halton Sequence)

매 프레임마다 씬의 위치를 조금씩 이동시키기 위해 TAA에서는 할톤 수열이라는 것 사용합니다. 할톤 수열은 몬테카를로 시뮬레이션 등에서 사용되는 공간상의 점을 생성하는데 사용되는 수열로 결정론적(= 같은 입력에 항상 같은 출력값을 가짐) 이며 저 불일치(low discrepancy)인 수열입니다. 저 불일치 수열은 기존 숫자로 부터 가능한 멀리 떨어져 있는 숫자를 연속하여 생성하기 때문에 군집화를 방지합니다. 아래 도표는 할톤 수열로 생성한 2차원 점과 랜덤하게 생성한 2차원의 점 216개를 서로 비교한 결과로 저 불일치 수열의 특징을 보여줍니다.



랜덤하게 생성한 2차원 점은 다음과 같은 코드를 통해 생성하였습니다.

```
std::random_device rd;
std::mt19937 mt( rd() );
std::uniform_real_distribution<float> ud( 0.f, 1.f );

std::ofstream random( "random.txt" );
for ( int i = 0; i < 216; ++i )
{
  random << ud( mt ) << "\t" << ud( mt ) << "\n";
}
```

이제 할톤 수열을 어떻게 생성하는지 살펴보겠습니다. 할톤 수열은 Radical inverse라는 것에 기초하여 생성할 수 있습니다. Radical inverse이 무엇인지 알아보기 전에 양의 정수를 특정 기수(=기저, 밑, base)로 나타내는 방법을 살펴 보겠습니다.

양의 정수 a는 특정 기수 b를 사용하여 다음과 같이 나타낼 수 있습니다.

$$a = \sum_{i=1}^{m} d_i(a) b^{i-1}$$

여기서 $d_i(a)$는 0에서 $b-1$ 사이의 숫자입니다.

수식보다 좀 더 알기 쉬운 예시를 보자면 2진수가 있겠습니다. 양의 정수 13을 2진수로 표현하면 ($b=2$) 1101_2 가 됩니다. 이를 위의 수식으로 표현하면

$$a = 1 * 2^0 + 0 * 2^1 + 1 * 2^2 + 1 * 2^3$$

가 됩니다. 이 식에 $d_0(a) = 1, d_1(a) = 0, d_2(a) = 1, d_3(a) = 1$ 인 것을 확인할 수 있습니다.

Radical inverse 함수 ϕ_b 는 기수 b 를 통해 나타낸 양의 정수 a를 소수점을 기준으로 반전시키는 것으로 [0, 1) 범위의 분수로 변환합니다.

$$\phi_b(a) = 0.d_1(a)d_2(a)...d_m(a)$$

즉 양의 정수 13을 기수 2를 통해 Radical inverse를 적용하면 다음과 같습니다.

$$0.1011_2 = \frac{11}{16}_{10}$$

이를 이용해 n차원의 할톤 수열을 만들려면 각 차원마다 서로 다른 기수 b를 사용하면 됩니다. 여기서 b는 반드시 서로소여야 하므로 소수를 사용하게 됩니다.

$$x_a = (\phi_2(a), \phi_3(a), \phi_5(a), ...\phi_{p_n}(a))$$

TAA에서는 스크린 평면상에서 물체를 이동시킬 것이므로 2차원 할톤 수열을 통해 2차원의 점을 생성하면 됩니다. 즉 $p = (\phi_2(a), \phi_3(a))$인 쌍을 생성합니다.

여기서 $\phi_2(a)$는 좀 더 효율적으로 계산할 수 있는데 컴퓨터에서 정수를 이미 2진수로 표현하고 있기 때문에 비트를 뒤집어 주기만 하면 됩니다.

```
uint32 ReverseBits( uint32 n )
{
  n = ( n << 16 ) | ( n >> 16 );
  n = ( ( n & 0x00ff00ff ) << 8 ) | ( ( n & 0xff00ff00 ) >> 8 );
  n = ( ( n & 0x0f0f0f0f ) << 4 ) | ( ( n & 0xf0f0f0f0 ) >> 4 );
  n = ( ( n & 0x33333333 ) << 2 ) | ( ( n & 0xcccccccc ) >> 2 );
  n = ( ( n & 0x55555555 ) << 1 ) | ( ( n & 0xaaaaaaaa ) >> 1 );
  return n;
}

uint64 ReverseBits( uint64 n )
{
  uint64 hi = ReverseBits( static_cast<uint32>( n ) );
  uint64 lo = ReverseBits( static_cast<uint32>( n >> 32 ) );
  return ( hi << 32 ) | lo;
}

float Halton::RadicalInverse( uint32 baseIndex, uint64 a ) const
{
  switch ( baseIndex )
  {
  case 0:
    return static_cast<float>( ReverseBits( a ) * 0x1p-64 );
  case 1:
    return ::RadicalInverse<3>( a );
  default:
    assert( false );
    break;
  }

  return 0.5f;
}
```

$\phi_3(a)$ 부터는 이 방법을 사용할 수 없기 때문에 정수를 기수 b로 나눠가면서 직접 생성해 줍니다.

```
template <uint64 base>
float RadicalInverse( uint64 a )
{
  constexpr float invBase = 1.f / base;
  uint64 reversedDigits = 0;
  float invBaseN = 1;

  while ( a )
  {
    uint64 next = a / base;
    uint64 digit = a % base;
    reversedDigits = reversedDigits * base + digit;
```

```
    invBaseN *= invBase;
    a = next;
  }

  return std::min( reversedDigits * invBaseN, (float)0x1.fffffep-1 );
}
```

해당 코드를 통해 생성한 점은 다음과 같습니다.

```
0.5 0.333333
0.25  0.666667
0.75  0.111111
0.125 0.444444
0.625 0.777778
0.375 0.222222
0.875 0.555556
0.0625  0.888889
0.5625  0.037037
0.3125  0.37037
0.8125  0.703704
0.1875  0.148148
0.6875  0.481482
0.4375  0.814815
0.9375  0.259259
.
.
.
```

Jittering 적용

생성한 2차원 할톤 수열을 이용해 매 프레임마다 지터링을 적용하여 씬을 그리도록 합니다. 할톤 수열은 다음과 같이 앞에서 16개를 골라 상수로 사용하도록 하였습니다.

```
static const int MAX_HALTON_SEQUENCE = 16;

static const float2 HALTON_SEQUENCE[MAX_HALTON_SEQUENCE] = {
  float2( 0.5, 0.333333 ),
  float2( 0.25, 0.666667 ),
  float2( 0.75, 0.111111 ),
  float2( 0.125, 0.444444 ),
  float2( 0.625, 0.777778 ),
  float2( 0.375, 0.222222 ),
  float2( 0.875, 0.555556 ) ,
  float2( 0.0625, 0.888889 ),
  float2( 0.5625, 0.037037 ),
  float2( 0.3125, 0.37037 ),
  float2( 0.8125, 0.703704 ),
  float2( 0.1875, 0.148148 ),
  float2( 0.6875, 0.481482 ),
  float2( 0.4375, 0.814815 ),
  float2( 0.9375, 0.259259 ),
  float2( 0.03125, 0.592593 )
};
```

지터링은 투영변환까지 적용한 위치에 아래와 같이 적용합니다. 할톤 수열은 [0, 1)의 구간을 가지므로 [-1, 1)의 구간으로 변경한 다음 픽셀 하나의 크기가 될 수 있도록 프레임 버퍼의 크기로 나눠 줍니다.

```
float4 ApplyTAAJittering( float4 clipSpace )
{
#if TAA == 1
  int idx = FrameCount % MAX_HALTON_SEQUENCE;

  // [0, 1) -> [-1, 1) -> 픽셀 하나 크기의 uv 좌표로 변경
  float2 jitter = HALTON_SEQUENCE[idx];
  jitter.x = ( jitter.x - 0.5f ) / ViewportDimensions.x * 2.f;
  jitter.y = ( jitter.y - 0.5f ) / ViewportDimensions.y * 2.f;

  clipSpace.xy += jitter * clipSpace.w; // Pixel Shader로 전달시 w로 나뉘므로 곱해준다.
#endif

  return clipSpace;
}
```

혼합

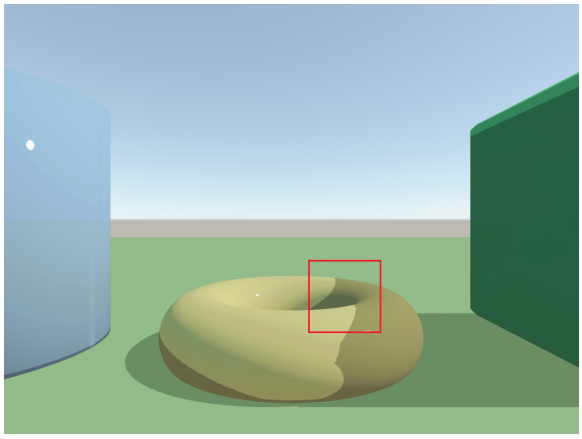
이제 지터링이 적용된 이미지를 과거 프레임의 이미지와 섞어주면 됩니다. A Survey of Temporal Antialiasing Techniques(behindthepixels.io/assets/files/TemporalAA.pdf) 에 따르면 TAA의 혼합은 다음과 같은 단순한 선형 보간식에 의해서 이뤄집니다.

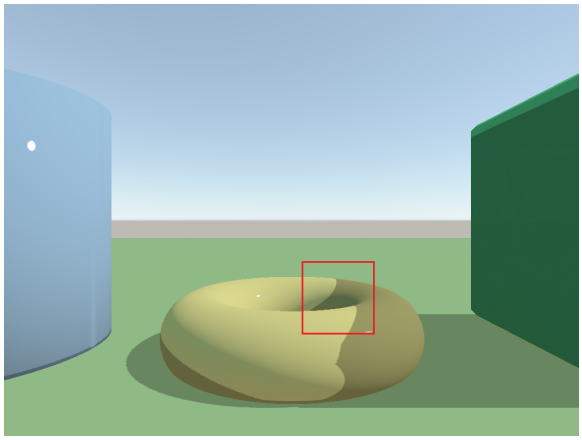
$$f_n(p) = a * s_n(p) + (1 - a) * f_{n-1}(\pi(p))$$

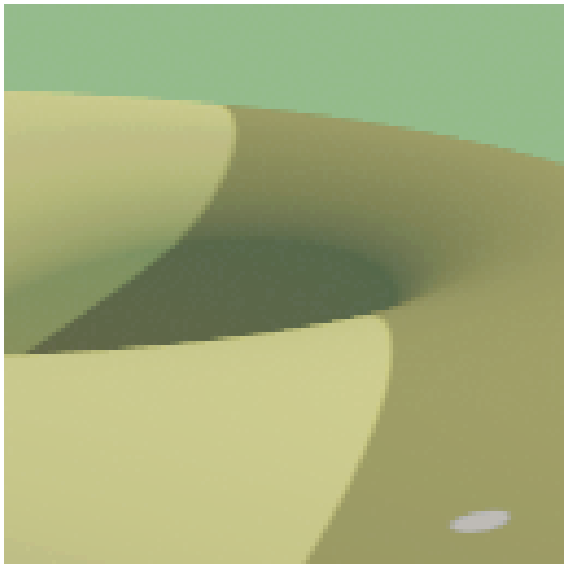
$f_n(p)$는 n프레임의 픽셀 p에 대한 색상 출력이며 a는 블렌딩 가중치. $s_n(p)$는 n프레임에 새롭게 계산된 색상, $f_{n-1}(\pi(p))$는 지금까지 혼합된 이전 프레임의 색상입니다.

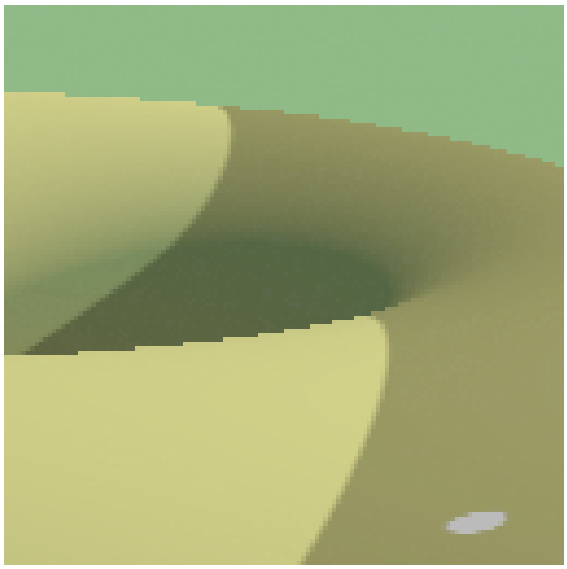
가중치 수치 a는 과거 색상과 현재 색상간의 균형을 맞추는데 대부분의 TTA 구현은 고정된 a를 사용하며 대체로 0.1이 사용된다고 합니다. 따라서 다음과 같이 간단하게 혼합할 수 있습니다.

```
float4 main( PS_INPUT input ) : SV_Target0
{
  float3 historyColor = HistoryTex.Sample( HistoryTexSampler, input.uv ).rgb;
  float4 sceneColor = SceneTex.Sample( SceneTexSampler, input.uv );

  // sceneColor.rgb * ( 1 - 0.9 ) + historyColor * 0.9
  // sceneColor.rgb * 1.0 + historyColor * 0.9
  float3 resolveColor = lerp( sceneColor.rgb, historyColor, 0.9 );

  return float4( resolveColor, sceneColor.a );
}
```





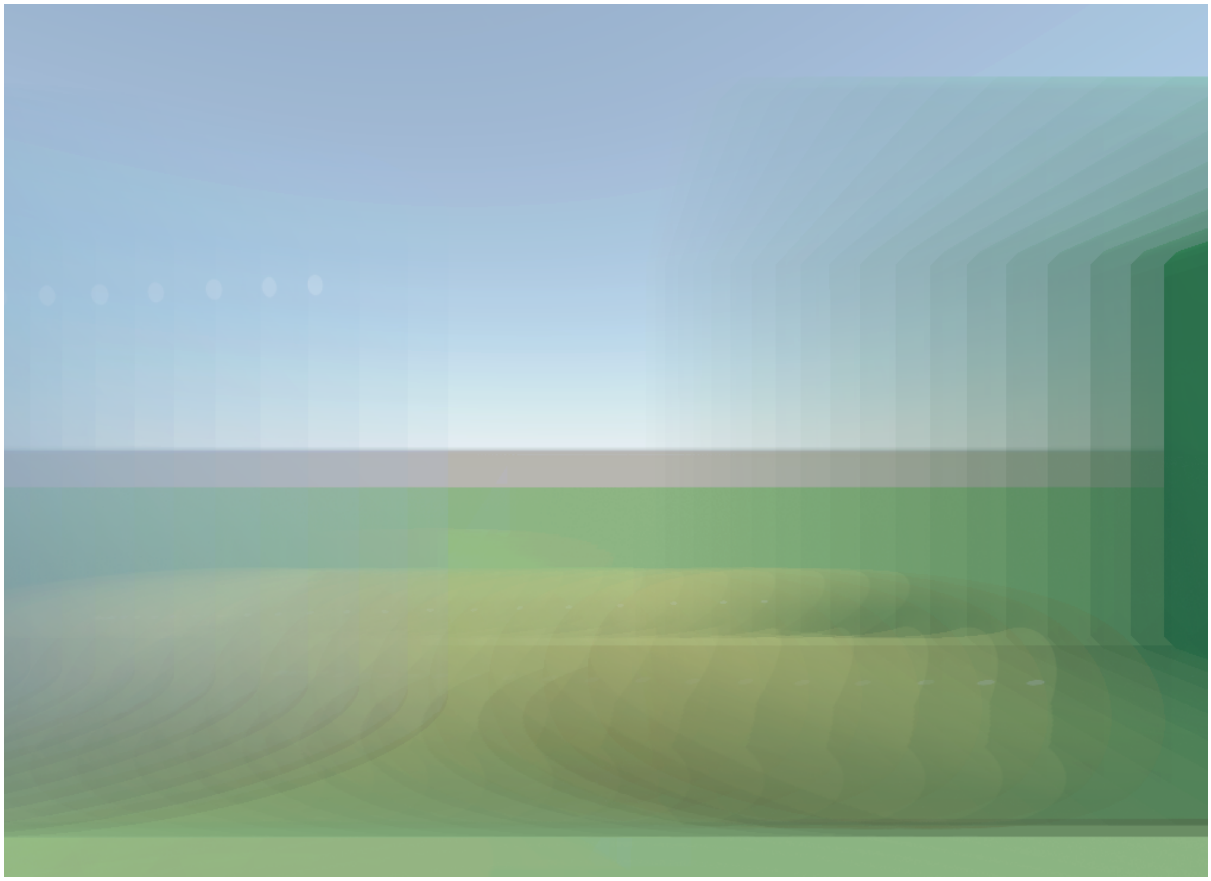




TAA On

TAA Off

적용전과 비교 했을 때 에일리어싱 현상이 개선된 것을 확인할 수 있습니다. 그런데 이 상태에서 카메라를 움직여 보면 문제가 발생합니다.



Ghosting

고스팅 현상은 이전 프레임의 이미지를 섞음으로 인해서 잔상과 같은 효과가 생기는 아티팩트를 의미합니다. 이전 프레임의 이미지가 유령처럼 남아 있다는 의미로 이렇게 불립니다. 지금까지의 방법은 정적인 씬에서는 잘 동작하지만 카메라나 물체의 위치 변환이 있는 동적인 씬에서는 이전 프레임의 색상 위치가 변경되었는데 이를 적절하게 처리하지 못하기 때문에 고스팅 현상을 방지할 수 없습니다. 고스팅 현상을 처리하는 방식은 다양한데 여기서는 이를 해결하기 위한 2가지 방식을 보겠습니다.

동적씬에서 아티팩트 해결

Velocity Buffer

첫번째는 Velocity Buffer를 도입하는 것입니다. Velocity Buffer는 현재 프레임과 이전 프레임의 픽셀간 위치 차이를 저장하고 있는 버퍼로 이를 이용하여 현재 프레임의 픽셀에 대한 알맞는 과거 프레임의 픽셀을 샘플링 할 수 있도록 텍스쳐 uv를 조정합니다.

Velocity Buffer를 구성해 보겠습니다. 준비해야 할 것은 이전 프레임의 월드, 카메라, 투영 변환 행렬입니다. 정점 셰이더에는 해당 행렬을 가지고 이전 프레임의 위치와 현재 프레임의 위치를 계산하여 픽셀 셰이더로 전달합니다.

```
VS_OUTPUT main( VS_INPUT input )
{
```

```
  VS_OUTPUT output = (VS_OUTPUT)0;

  PrimitiveSceneData primitiveData = GetPrimitiveData( input.primitiveId );
  output.curFramePosition = mul( float4( input.position, 1.0f ), primitiveData.m_worldMatrix );
  output.curFramePosition = mul( float4( output.curFramePosition.xyz, 1.0f ), ViewMatrix );
  output.curFramePosition = mul( float4( output.curFramePosition.xyz, 1.0f ), ProjectionMatrix );
  output.worldNormal = mul( float4( input.normal, 0.f ), transpose( primitiveData.m_invWorldMatrix ) ).xyz;

  output.prevFramePosition = mul( float4( input.position, 1.0f ), primitiveData.m_prevWorldMatrix );
  output.prevFramePosition = mul( float4( output.prevFramePosition.xyz, 1.0f ), PrevViewMatrix );
  output.prevFramePosition = mul( float4( output.prevFramePosition.xyz, 1.0f ), PrevProjectionMatrix );

  output.position = ApplyTAAJittering( output.curFramePosition );

  return output;
}
```

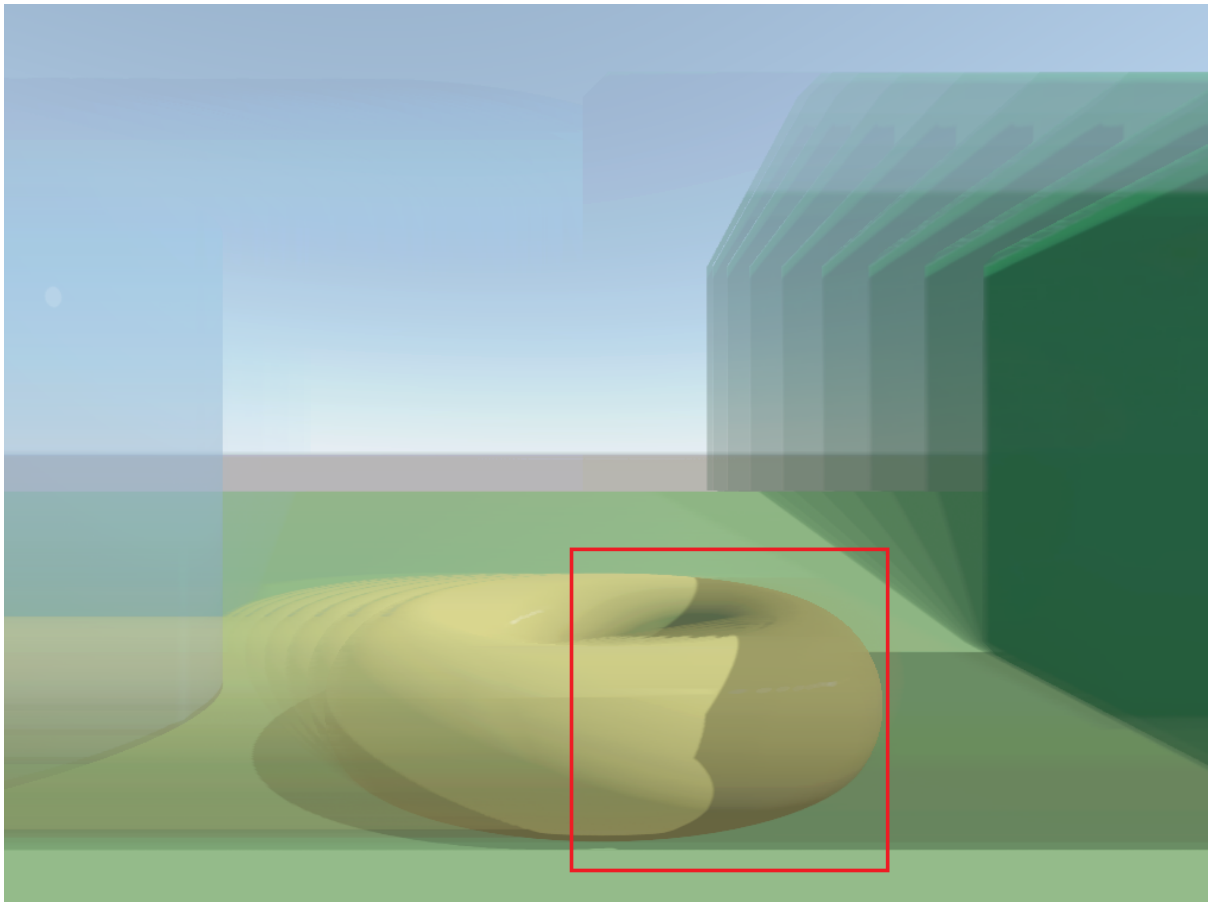
픽셀 셰이더는 전달 받은 위치를 스크린의 uv좌표로 변경하여 저장합니다.

```
float2 CalcVelocity( float4 curFramePosition, float4 prevFramePosition )
{
  float2 curFrameUV = curFramePosition.xy / curFramePosition.w;
  curFrameUV = curFrameUV * 0.5f + 0.5f;
  curFrameUV.y = 1.f - curFrameUV.y;

  float2 prevFrameUV = prevFramePosition.xy / prevFramePosition.w;
  prevFrameUV = prevFrameUV * 0.5f + 0.5f;
  prevFrameUV.y = 1.f - prevFrameUV.y;

  return curFrameUV - prevFrameUV;
}

Output main( PS_INPUT input )
{
  Output output = (Output)0;
  output.depth = input.position.w / FarPlaneDist;
  float3 enc = SignedOctEncode( normalize( input.worldNormal ) );
  output.packedNormal = float4( 0.f, enc );
  output.velocity = CalcVelocity( input.curFramePosition, input.prevFramePosition );

  return output;
}
```

이제 TAA 셰이더는 Velocity Buffer에 저장된 uv값을 사용하여 현재 프레임의 픽셀에 알맞는 이전 프레임의 픽셀을 샘플링합니다.

```
float4 main( PS_INPUT input ) : SV_Target0
{
  float2 velocity = VelocityTex.Sample( VelocityTexSampler, input.uv );
  float2 previousUV = input.uv - velocity;

  float3 historyColor = HistoryTex.Sample( HistoryTexSampler, previousUV ).rgb;
  float4 sceneColor = SceneTex.Sample( SceneTexSampler, input.uv );

  float3 resolveColor = lerp( sceneColor.rgb, historyColor, BlendWeight );

  return float4( resolveColor, sceneColor.a );
}
```

Velocity Buffer를 적용한 결과 다음과 같은 결과를 얻을 수 있습니다.



붉은 네모 부분을 보면 고스팅 현상이 사라진 것을 확인 할 수 있습니다. 하지만 여전히 잔상이 발생하는 부분이 있습니다.

색상 Clamp

두번째는 색상 Clamp입니다. 만약 이전 프레임의 픽셀 색상이 현재 프레임의 픽셀 색상과 유사하다면 잔상 현상이 발생할까요? 파란색에 (R : 0, G : 0, B : 255) 약간 흐린 파란색 (R : 0, G : 0, B : 240)을 섞는다고 해도 그리 티가 나지 않을거라 예상할 수 있습니다. 색상 Clamp는 현재 프레임의 색상을 기준으로 이전 프레임의 색상을 조정하는 방법입니다.

여기서는 색상 Clamp을 위해 현재 프레임의 상하좌우로 이웃하는 4픽셀을 추가로 샘플링하여 색상의 최소 최대 범위를 계산합니다.

```
float3 left = SceneTex.Sample( SceneTexSampler, input.uv, int2( -1, 0 ) ).rgb;
float3 right = SceneTex.Sample( SceneTexSampler, input.uv, int2( 1, 0 ) ).rgb;
float3 top = SceneTex.Sample( SceneTexSampler, input.uv, int2( 0, -1 ) ).rgb;
float3 bottom = SceneTex.Sample( SceneTexSampler, input.uv, int2( 0, 1 ) ).rgb;

float3 lower = min( sceneColor, min( min( left, right ), min( top, bottom ) ) );
float3 upper = max( sceneColor, max( max( left, right ), max( top, bottom ) ) );
```

그리고 이 lower, upper 색상 범위로 이전 프레임의 색상을 조정합니다.

```
historyColor = clamp( historyColor, lower, upper );
```

색상 Clamp를 적용한 TAA 셰이더의 전체 모습은 다음과 같습니다.

```
float4 main( PS_INPUT input ) : SV_Target0
{
  float2 velocity = VelocityTex.Sample( VelocityTexSampler, input.uv );
  float2 previousUV = input.uv - velocity;

  float3 historyColor = HistoryTex.Sample( HistoryTexSampler, previousUV ).rgb;
  float4 sceneColor = SceneTex.Sample( SceneTexSampler, input.uv );

  float3 left = SceneTex.Sample( SceneTexSampler, input.uv, int2( -1, 0 ) ).rgb;
  float3 right = SceneTex.Sample( SceneTexSampler, input.uv, int2( 1, 0 ) ).rgb;
  float3 top = SceneTex.Sample( SceneTexSampler, input.uv, int2( 0, -1 ) ).rgb;
  float3 bottom = SceneTex.Sample( SceneTexSampler, input.uv, int2( 0, 1 ) ).rgb;

  float3 lower = min( sceneColor, min( min( left, right ), min( top, bottom ) ) );
  float3 upper = max( sceneColor, max( max( left, right ), max( top, bottom ) ) );

  historyColor = clamp( historyColor, lower, upper );
  float3 resolveColor = lerp( sceneColor.rgb, historyColor, BlendWeight );

  return float4( resolveColor, sceneColor.a );
}
```

이를 적용하면 다음과 같이 고스팅 현상을 개선할 수 있습니다.

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/4fbdd80c-15a1-4dea-9bff-201642823616/2022-10-12_23_12_55.mp4

올바르지 않은 과거 프레임의 픽셀을 조정하는 방식은 이외에도 다양한데 게임 '인사이드'의 GDC 자료를 보면 3x3 범위의 이웃 픽셀에 대해서 과거 프레임의 픽셀을 clipping하는 방법을 사용한 것을 볼 수 있습니다.

마치며…

준비한 내용은 여기까지입니다. 이 내용은 훌륭한 TAA 튜토리얼 글인 https://sugulee.wordpress.com/2021/06/21/temporal-anti-aliasingtaa-tutorial/ 를 참고 하여 Direct3D11을 사용한 개인 프로젝트의 코드를 기반으로 작성되었습니다.

전체 코드는 아래의 변경점에서

https://github.com/xtozero/SSR/commit/8f732f29d23063c914e0285120abaed024f9bba3

Source/RenderCore/Private/Renderer/TemporalAntiAliasingRendering.cpp

Source/Shaders/Private/TemporalAntiAliasing/PS_TAAResolve.fx

Source/Shaders/Private/VS_DepthWrite.fx

Source/Shaders/Private/PS_DepthWrite.fx

등의 파일을 참고하시면 됩니다.

Reference

behindthepixels.io/assets/files/TemporalAA.pdf

https://sugulee.wordpress.com/2021/06/21/temporal-anti-aliasingtaa-tutorial/

https://en.wikipedia.org/wiki/Temporal_anti-aliasing

https://pbr-book.org/3ed-2018/Sampling_and_Reconstruction/The_Halton_Sampler

https://ziyadbarakat.wordpress.com/2020/07/28/temporal-anti-aliasing-step-by-step/

http://s3.amazonaws.com/arena-attachments/655504/c5c71c5507f0f8bf344252958254fb7d.pdf?1468341463