

유니버설 렌더 파이프라인 가이드:

셰이더 및 시각 효과를 만드는 레시피



목차

들어가는 말	5
저자 및 도움을 주신 분들	7
도움을 주신 유니티 직원 분들	7
가이드 시작하기	8
새 URP 프로젝트 시작	9
전자책 샘플 씬 импорт	10
스텐실	13
렌더러 기능	14
인스턴싱	20
GPU 상주 드로어 및 GPU 오클루전 컬링	21
인스턴싱	23
SRP Batcher	24
GPU 인스턴싱	26
RenderMeshPrimitives	28
툰 및 아웃라인 셰이딩	33
단순한 툰 셰이딩	35
셰이딩	35
아웃라인 설정	36
툰 셰이딩	37
셰이딩	37
아웃라인 설정	41
앰비언트 오클루전	44
SSAO 프로퍼티	46

데칼	49
URP Decal Projection 프로퍼티	51
머티리얼 생성	52
코드로 데칼 추가	53
물	55
DepthFade 하위 그래프	57
TextureMovement 하위 그래프	58
물 셰이더	58
컬러	59
노멀 맵	61
너울	62
컬러 그레이딩용 LUT	64
적응적 프로브 볼륨	72
씬에서 APV 사용	74
조명 시나리오 에셋	77
APV 문제 해결	80
빛 번짐 현상	82
렌더링 레이어	83
APV 스트리밍	86
스카이 오클루전	87
라이트 프로브와 APV 비교	90
스크린 공간 굴절	92
볼류메트릭	102
볼류메트릭 구름	103

절차적 노이즈	114
절차적 노이즈의 유형	115
Unity에서 절차적 노이즈 구현하기	115
절차적 하이트맵 예시	116
노이즈를 사용하여 목재 텍스처 생성	119
절차적 노이즈의 장점	126
문제점과 최적화	127
컴퓨터 셰이더	128
ParticleFun	129
메시 오브젝트 추가	139
맺는말	151

들어가는 말

소량의 포스트 프로세싱 효과, 데칼 한 컵, 컬러 그레이딩 한 꼬집, 그리고 약간의 탄산수를 준비하세요. URP(유니버설 렌더 파이프라인)를 사용하여 게임에 고품질의 조명과 시각 효과를 구현할 시간입니다.

12개의 챕터로 구성된 이 URP 가이드에서는 자주 사용되는 효과를 만드는 다양한 레시피를 확인할 수 있습니다. 가이드의 주 저자인 닉 레버가 관리하는 [GitHub 저장소](#)에서 이러한 레시피를 기반으로 제작된 샘플 씬을 다운로드할 수도 있습니다.

이 가이드는 Unity를 이용한 프로젝트 개발 경험이 있고, URP 기능 사용법을 알고 있으며, HLSL 기반 셰이더 작성에 관해 어느 정도 지식을 보유한 중급 Unity 사용자를 위해 마련되었습니다.

이 책에서 소개하는 레시피로 만들 수 있는 콘텐츠는 다음과 같습니다.

- 스텐실로 엑스레이 같은 이미지 효과 생성
- Shader Graph로 툰 및 아웃라인 셰이더 제작
- 포스트 프로세싱으로 앰비언트 오클루전 효과 추가
- Photoshop과 LUT 이미지를 사용해 씬에 컬러 그레이딩 추가
- 반사 및 굴절 효과 생성을 비롯한 기타 다양한 작업

본 전자책과 각 레시피의 다운로드 가능한 샘플 씬은 Unity 6에 맞게 업데이트되었습니다.
이 가이드에 새로 추가된 내용은 다음과 같습니다.

- 절차적 노이즈 및 컴퓨트 셰이더를 위한 새 레시피 2개
- 툰 셰이더에 관한 전체 내용 개정
- 라이트 프로브를 간단하고 유연하게 대체할 수 있는 APV(적응적 프로브 볼륨) 구현 방법에 관한 섹션
- 새로운 Render Graph API를 사용하여 렌더러 기능을 만드는 방법

이 가이드와 함께 [Unity 고급 사용자를 위한 유니버설 렌더 파이프라인](#) 전자책을 참고하면 좋습니다.
또한 Unity YouTube 채널에 있는 [URP 튜토리얼](#) 재생 목록은 게임용 조명 및 효과 제작을 위한 일반적인 팁과 전문적인 팁을 제공합니다.

게임에 어울리는 멋진 효과를 즐겁게 만들어 보세요.

본 전자책에서 소개하는 레시피의 다수는 [HLSL](#)(고급 셰이더 언어)을 사용합니다. 이 언어를 처음 접한다면 아래의 리소스를 참고하세요.

- [Unity 매뉴얼의 커스텀 셰이더 예시](#)
- [Ronja의 HLSL 튜토리얼](#)(영문)
- [Udemy: Unity 셰이더 기초부터 배우기](#)(영문)
- [The Book of Shaders](#)



이 이미지는 숙련된 개발자가 URP로 어떤 결과를 낼 수 있는지 보여 주는 게임인 PRINCIPLES에서 가져왔습니다. PRINCIPLES는 Shironeko Project와 Quiz RPG: The World of Mystic Wiz 시리즈를 개발한 COLOPL Inc의 기술 브랜드인 COLOPL Creators에서 제작한 어드벤처 게임입니다. 놀라운 그래픽스와 몰입도 높은 3D 음향 등 URP를 비롯한 최신 Unity 기능이 최대한 활용된 깊은 지하 월드를 경험해 보세요. PRINCIPLES는 현재 [App Store](#) 또는 [Google Play](#)에서 사용할 수 있습니다. [여기](#)에서 스튜디오와의 인터뷰도 확인할 수 있습니다.

저자 및 도움을 주신 분들

이 전자책의 주요 저자인 닉 레버는 1990년대 중반부터 실시간 3D 콘텐츠를 제작해 왔으며, Unity는 2006년부터 사용했습니다. 닉은 30년 넘게 소규모 개발 회사인 Catalyst Pictures를 운영하며 빠르게 진화하는 업계에서 게임 개발자들이 지식을 증진할 수 있도록 2018년부터 관련 교육 과정을 제공했습니다.

도움을 주신 유니티 직원 분들

스티븐 카나반은 유니티의 시니어 소프트웨어 개발 컨설턴트로, 전문 분야는 그래픽스 및 렌더링입니다. 게임 개발 업계에서 15년 이상의 경력을 쌓았습니다.

밍와이 찰은 유니티 그래픽스 엔지니어링 팀의 시니어 테크니컬 아티스트입니다. 유니티에서 8년째 근무하고 있으며 2012년부터 Unity 에디터를 사용하고 있습니다.

올리버 슈나벨은 유니티 그래픽스 팀의 시니어 테크니컬 프로젝트 매니저로, 고객 인사이트를 통합하고 글로벌 스튜디오와 협력하여 더욱 성능과 확장성이 뛰어나고 통합된 렌더링 스택을 개발하는 데 집중하고 있습니다. 올리버는 컴퓨터 그래픽스와 실시간 개발에 관한 폭넓은 경험을 가지고 있습니다.

조나스 모텐슨은 유니티 그래픽스 팀의 시니어 테크니컬 아티스트입니다.

에이드리언 물랭은 유니티 렌더 파이프라인 팀의 시니어 그래픽스 개발자입니다. 에이드리언은 시뮬레이션 및 실시간 소프트웨어 업계에서 8년 이상의 경력을 쌓았습니다. 현재 스크립터블 렌더 파이프라인 사용자에게 최적의 기반과 API를 제공하는 데 집중하고 있습니다.

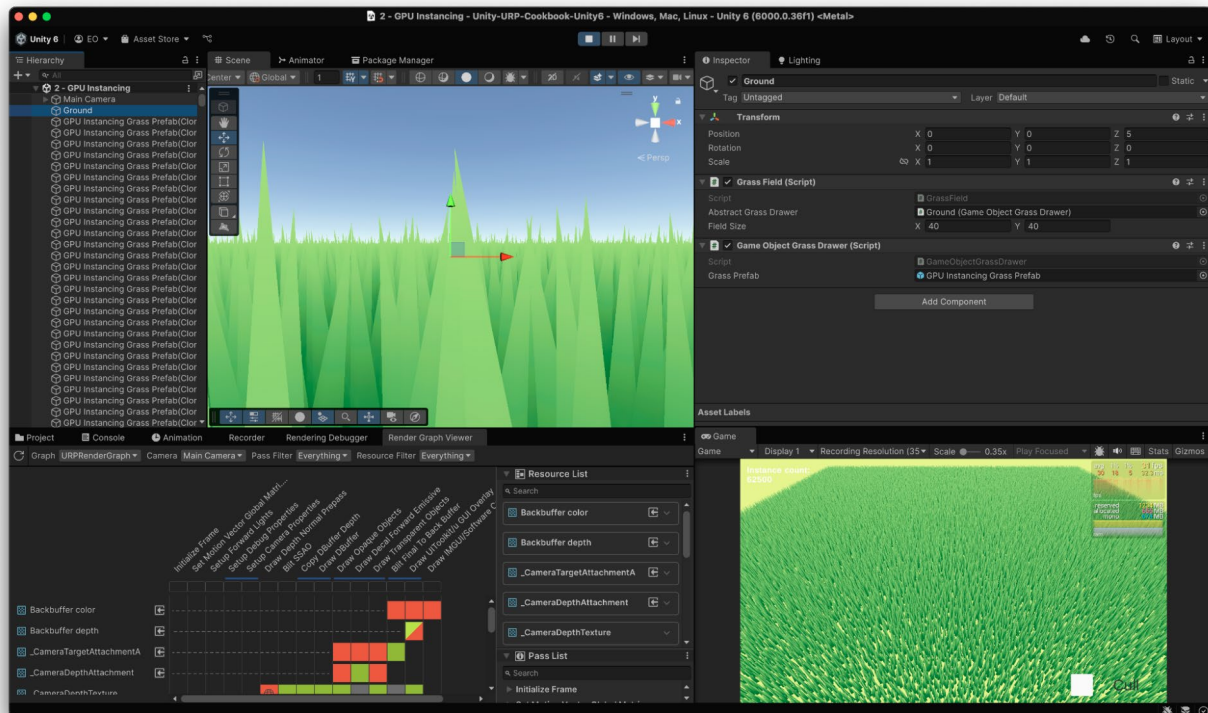
마티외 밀러는 유니티 그래픽스 팀의 리드 프로젝트 매니저입니다. 그래픽스 제품 관리 팀을 이끌며 그래픽스 로드맵과 제품 비전을 감독하고 있습니다.

데이미언 나흐만은 유니티 그래픽스 팀의 시니어 테크니컬 프로젝트 매니저로, 저사양 그래픽스 개발 및 최적화가 전문 분야입니다. 데이미언은 여러 업계에서 10년 동안 실시간 그래픽스 엔진과 벤치마킹 작업과 관련된 경력을 쌓았습니다.

가이드 시작하기

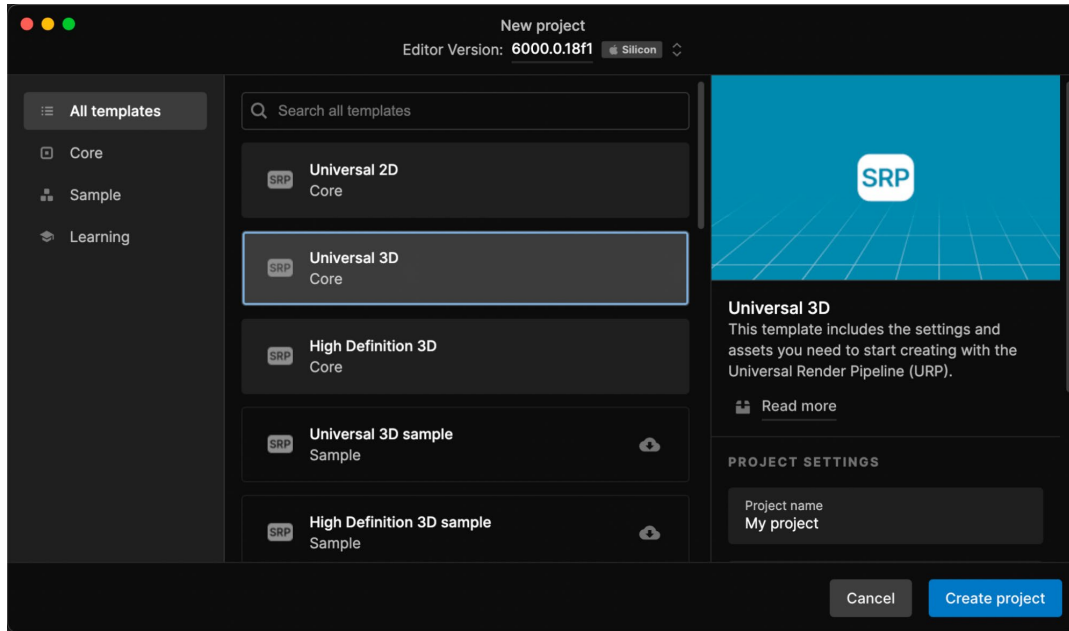
각 레시피의 단계에 따라 새 URP 프로젝트를 열어 조명 및 시각 효과를 재현할 수 있습니다. 가이드와 함께 제공되는 [GitHub 페이지](#)에서 각 레시피에 대한 샘플 씬을 다운로드할 수도 있습니다.

모든 레시피는 Unity 6에서 사용할 수 있도록 업데이트되었습니다.



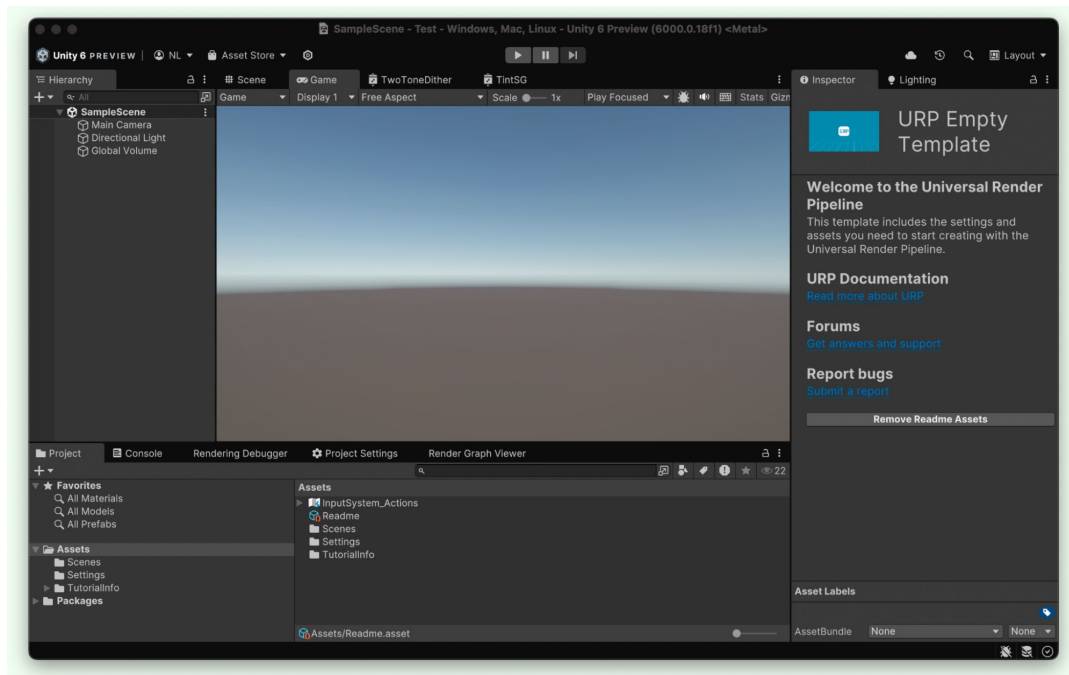
새 URP 프로젝트 시작

Unity Hub를 통해 URP를 사용하는 새 프로젝트를 엽니다. **New**를 클릭하고 선택한 Unity 버전이 6000.01 이상인지 창 상단에서 확인합니다. 프로젝트의 이름과 위치를 지정하고 **3D (URP)** 템플릿을 선택한 다음 **Create project**를 클릭합니다.



Universal 3D 템플릿을 사용해 새 프로젝트를 만드는 것이 처음이라면 템플릿을 다운로드해야 할 수 있습니다.

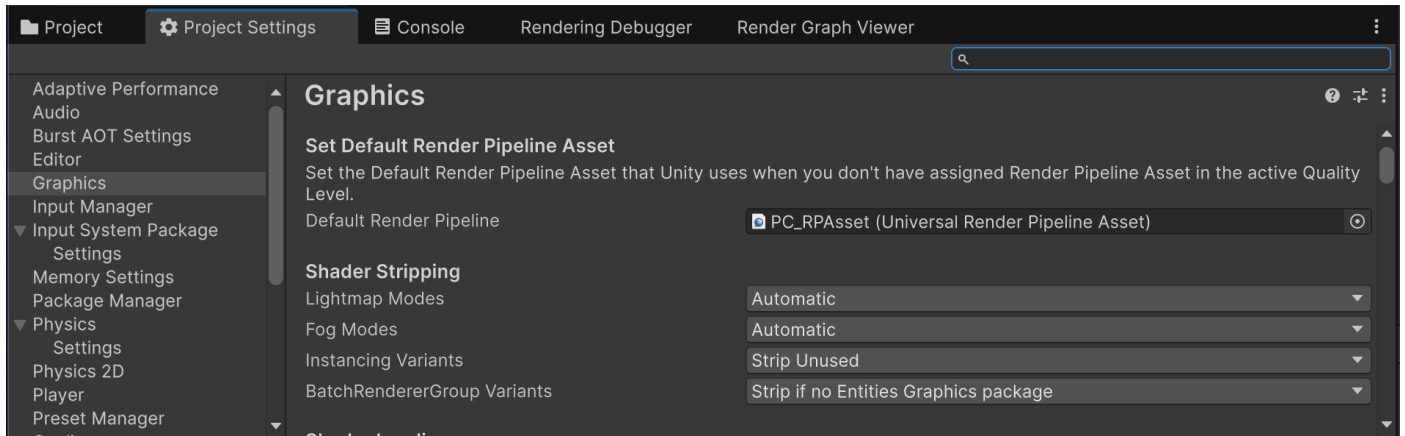
참고: 템플릿은 조명을 올바르게 계산하는 데 필요한 선형 색 공간을 사용하도록 프로젝트를 설정합니다.



이 템플릿은 비어 있지만, URP와 해당 애셋이 사전 설정 및 설치되어 있습니다.

Edit > Project Settings 로 이동하여 **Graphics** 패널을 엽니다. 그러면 **Default Render Pipeline Asset**이 표시됩니다. 이 [URP 에셋](#)은 프로젝트의 글로벌 렌더링 및 품질 설정을 제어하고 렌더링 파이프라인 인스턴스를 생성합니다. 한편, 렌더링 파이프라인 인스턴스에는 임시 리소스와 렌더 파이프라인 구현이 포함되어 있습니다.

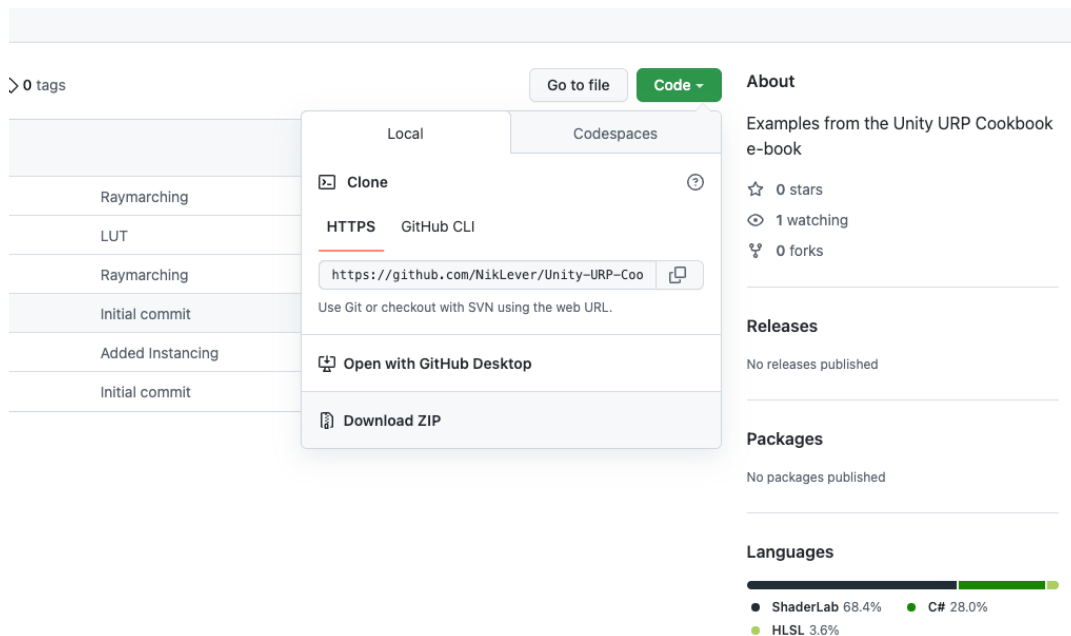
기본 URP 에셋으로 **PC_RPAsset**이 선택되지만 리소스가 제한적인 기기에 더 적합한 에셋인 **Mobile_RPAsset**으로 변경해도 됩니다.



Project Settings의 Graphics 패널

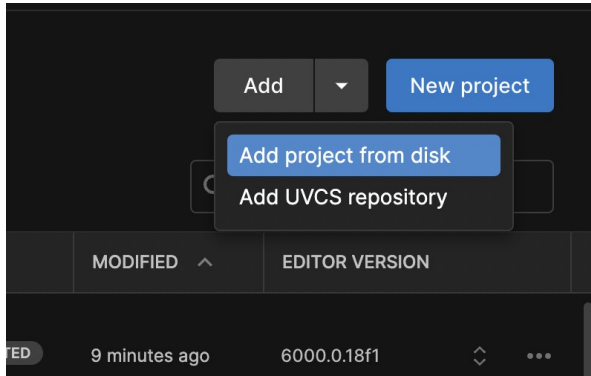
전자책 샘플 싼 импорт

[여기](#)에서 저장소를 복제하거나 ZIP 파일로 다운로드해 압축을 풉니다.



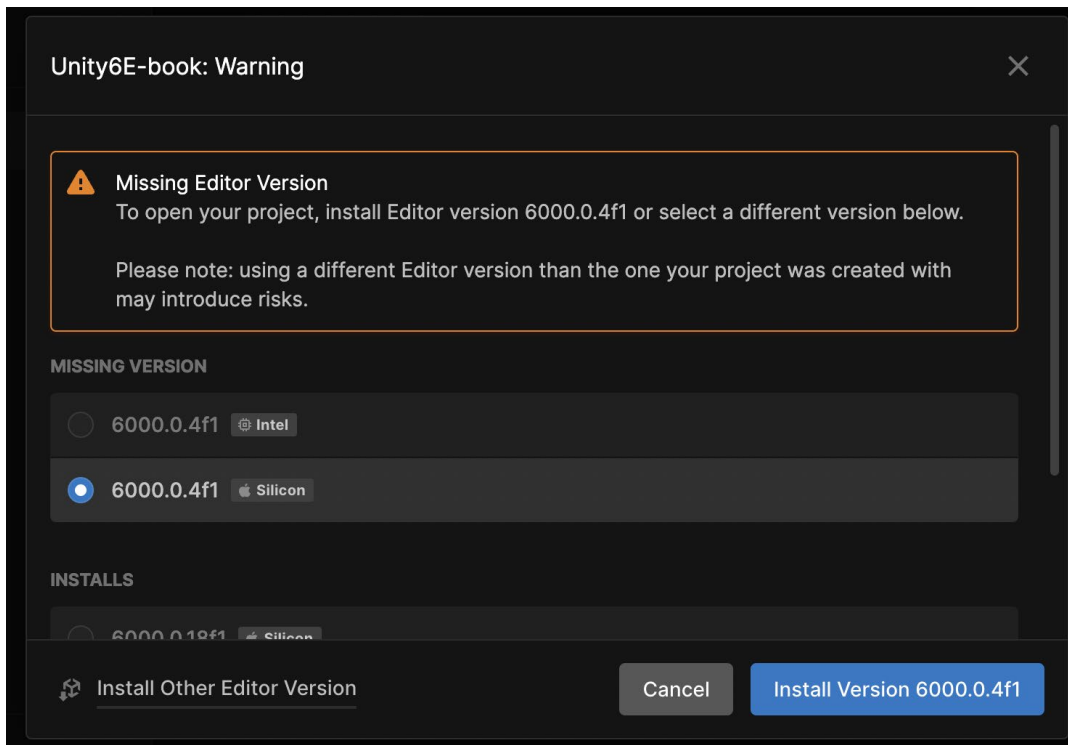
GitHub 저장소 - 초록색 Code 버튼을 클릭해 프로젝트 다운로드

프로젝트를 다운로드하고 압축을 해제했으면 Unity Hub에서 **Open > Add project from disk**를 클릭해 Unity Hub에서 프로젝트를 임포트합니다.



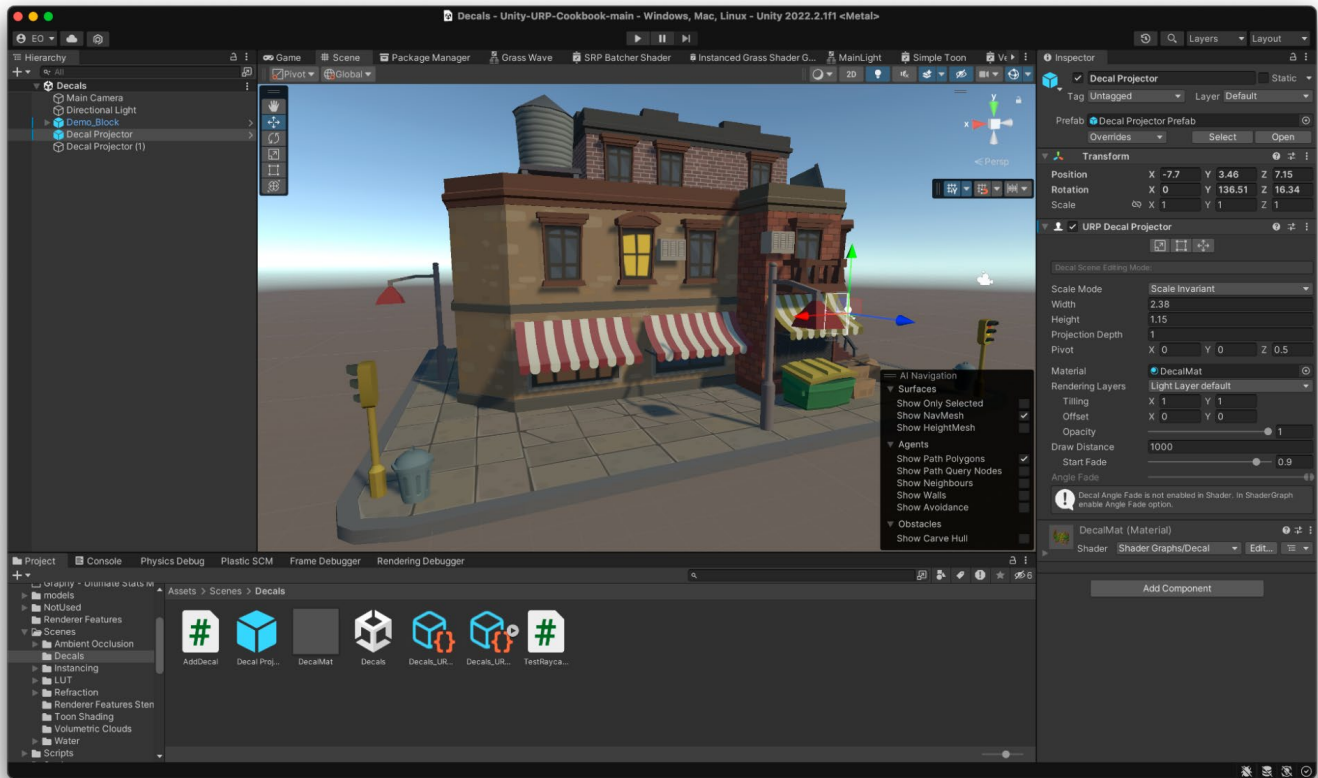
Unity Hub에서 샘플 프로젝트를 임포트합니다.

샘플 프로젝트에 사용된 것과 같은 버전의 에디터에서 작업해야 합니다. 에디터 버전이 서로 다르면 Unity Hub에 에디터 버전 누락과 관련한 경고 메시지가 표시됩니다. 아래 이미지에 표시된 대로 오른쪽 하단의 파란색 버튼을 눌러 누락된 버전을 설치할 수 있습니다.



현재 따라 하고 있거나 다운로드 중인 튜토리얼 프로젝트와 맞는 Unity 에디터 버전을 설치합니다. Unity Hub를 사용해 손쉽게 설치할 수 있습니다.

올바른 버전의 에디터를 설치했으면 프로젝트를 문제 없이 열 수 있습니다.



각 레시피는 이 전자책에서 단계별로 안내하는 파일이 포함된 폴더에 들어 있습니다.

스텐실



Unity로 제작된 게임 [TUNIC](#) (제작: [앤드루 솔다이스](#), [TUNIC Team](#), [22nd Century Toys LLC](#), [Isometricorp Games Ltd.](#), 퍼블리셔: [Finji](#))에서는 프랍이 주인공을 가리면 주인공의 실루엣이 프랍 위에 그려집니다. URP의 렌더러 기능으로 이 효과를 구현할 수 있습니다. 이 [동영상 튜토리얼](#)에서도 구현 방법을 확인할 수 있습니다.

URP에는 최종 렌더링을 제어하는 두 가지 에셋인 [Universal Renderer 에셋](#)과 [URP 에셋](#)이 있습니다. 전자의 경우, [렌더러 기능](#)을 추가해 다음과 같은 렌더링 파이프라인의 모든 단계에 삽입할 수 있습니다.

- 그림자 렌더링
- 프리패스 렌더링
- G버퍼 렌더링
- 디퍼드 광원 렌더링
- 불투명 렌더링
- 스카이박스 렌더링
- 투명도 렌더링
- 포스트 프로세싱 렌더링

렌더러 기능

렌더러 기능을 사용하면 조명과 효과를 다양하게 시도해 볼 수 있습니다. 이 섹션에서는 스텐실을 집중적으로 살펴보며 최소한의 필수적인 코드만 사용합니다.

과정을 따라 하려면 에디터에서 **Scenes > Renderer Features Stencils > SmallRoom - Stencil**로 이동해 샘플 씬을 엽니다.



스텐실 사용 사례: 돋보기를 책상 위에서 움직이면 서랍 안에 무엇이 있는지 들여다볼 수 있습니다.

위 이미지에서 볼 수 있듯이, 이 예시의 목표는 돋보기의 렌즈가 책상 속을 엑스레이 이미지처럼 투시하게 만드는 것입니다. 레이어 마스크와 셰이더, 렌더러 기능을 조합해 이를 구현했습니다. 먼저 렌즈에서 사용하는 머티리얼을 변경해야 합니다. 여기서는 **MaskMat** 머티리얼과 **Custom/StencilMask** 셰이더를 변경해야 합니다.



```
Shader "Custom/StencilMask"
{
    Properties{}

    SubShader{

        Tags {
            "RenderType" = "Opaque"
        }

        Pass {
            ZWrite Off

            HLSLPROGRAM

            #pragma vertex vert
            #pragma fragment frag

            #include
            "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Core.hlsl"

            struct Attributes
            {
                float4 positionOS    : POSITION;
            };

            struct Varyings
            {
                float4 positionHCS   : SV_POSITION;
            };

            Varyings vert(Attributes IN)
            {
                Varyings OUT;
                OUT.positionHCS = TransformObjectToHClip(IN.positionOS.xyz);

                return OUT;
            }

            half4 frag() : SV_Target
```

```

    {
        return (half4)0;
    }

    ENDHLSL
}
}
}

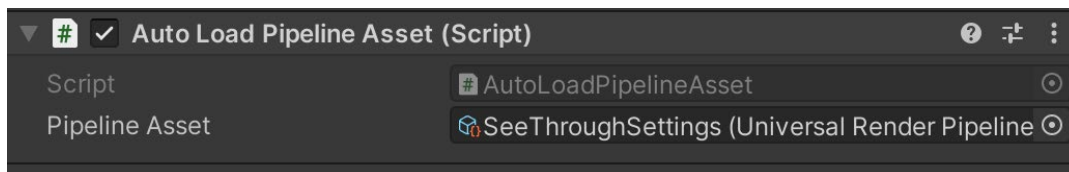
```

Custom/StencilMask의 ZWrite Off 커맨드를 주목하세요. 대부분의 경우 오브젝트에 ZWrite Off를 설정하면 오브젝트가 덤스 버퍼에 덤스 값을 쓰지 않기 때문에 사라지게 되어, 더 이상 뒤쪽에 있는 오브젝트를 가리지 않습니다. 오브젝트는 여전히 같은 순서로 렌더링되지만, 그려진 픽셀 콘텐츠는 뒤쪽 오브젝트에 의해 오버라이드됩니다. 정리하면, ZWrite Off를 설정해도 렌더링 순서는 변경되지 않습니다(하지만 렌더링 대기열 인덱스를 설정하면 변경됨). 오브젝트의 렌더링 대기열 인덱스를 지오메트리보다 높은 값으로 변경하면 오브젝트가 다시 나타납니다. 이 예시에서는 지오메트리 값인 2,000으로 남겨 두었습니다.

렌즈는 스텐실 버퍼에 값을 쓰는 작업만 하면 됩니다. 셰이더의 컬러 버퍼에 대한 출력이 아닌 스텐실 작성을 고려해야 하기 때문에 컬러 작성 ColorMask 0을 비활성화해도 됩니다. 이 방식은 약간 최적화된 방식이며, 특히 [디퍼드 렌더링 경로](#)와 함께 사용하려는 경우 씬이 렌즈 마스크보다 먼저 렌더링된다는 점에서 더욱 그렇습니다.

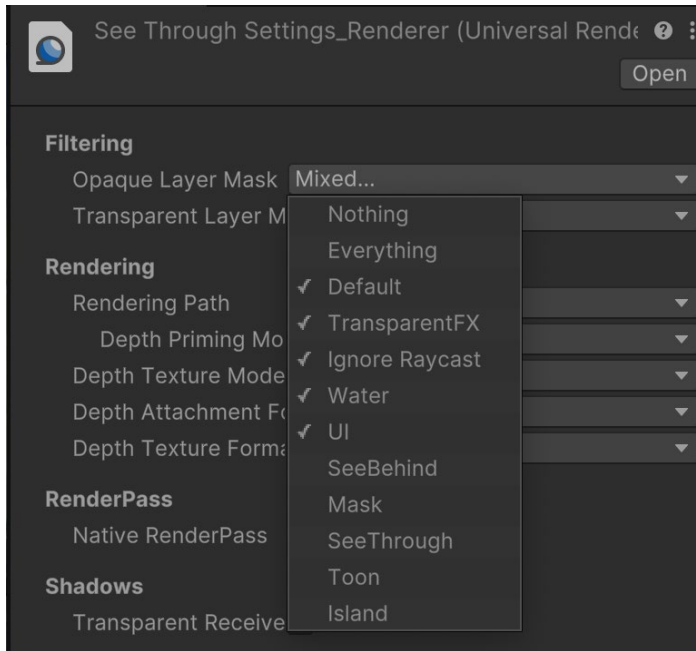
이 예시에서는 **Mask**와 **SeeThrough**라는 두 개의 커스텀 레이어를 사용합니다. 렌즈는 Mask 레이어에 있고, 책상(자식 오브젝트 제외)은 SeeThrough 레이어에 있습니다.

이 씬은 **See Through Settings_Renderer**라는 렌더러 데이터 오브젝트를 사용하며, 이 오브젝트는 씬 파일, 머티리얼, 셰이더가 있는 **Scenes > Renderer Feature Stencils** 폴더에 함께 있습니다. Main Camera에 연결된 스크립트인 AutoLoadPipelineAsset 스크립트는 **Project Settings > Graphics**에서 이 렌더러 데이터 오브젝트를 스크립터블 렌더 파이프라인 에셋으로 설정합니다. 이제 이 에셋의 설정을 확인해 보겠습니다.



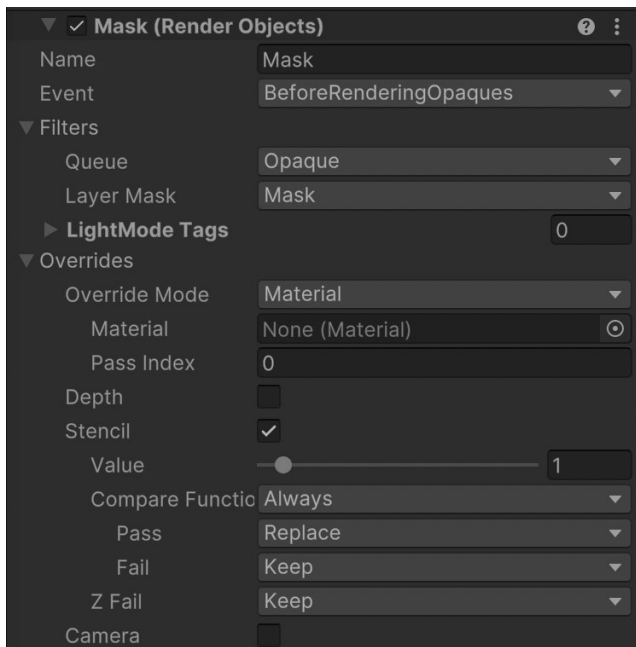
Main Camera > Auto Load Pipeline Asset 스크립트의 Pipeline Asset 설정

Scenes > Renderer Feature Stencils에서 **SeeThrough Settings_Renderer**를 선택합니다. 기본값에서 변경할 첫 번째 설정은 **Opaque Layer Mask**입니다. 참고로 이 경우 Mask와 SeeThrough는 제외됩니다.



See Through Settings_Renderer에서 Opaque Layer Mask 변경

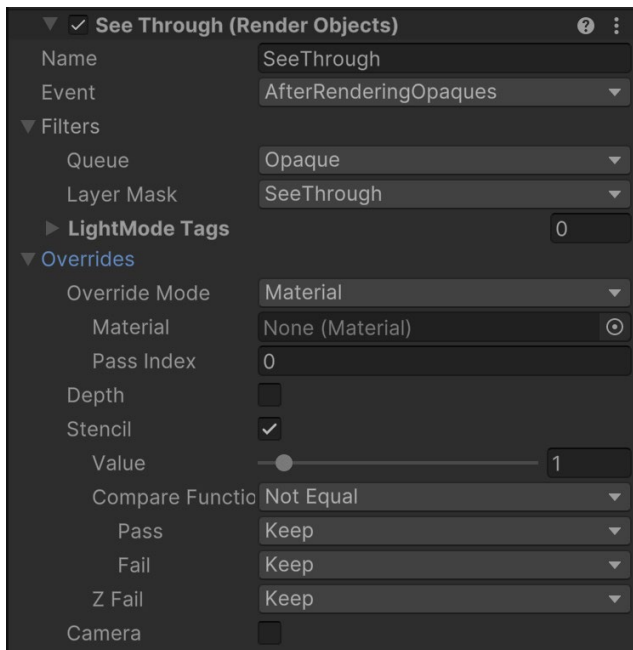
인스펙터(Inspector)의 렌더러 기능 목록에는 **Mask**와 **SeeThrough**라는 두 가지 렌더 오브젝트 기능이 있습니다. SeeThrough 옵션을 비활성화하면 책상이 사라집니다. 책상은 Opaque Layer Mask를 사용해 필터링된 레이어의 일부이며 기본 렌더의 일부가 아니므로, 오직 렌더 오브젝트 기능을 통해 렌더링되기 때문입니다.



Mask (Render Objects) 설정

위 이미지를 보면 Mask가 **BeforeRenderingOpqaues** 이벤트를 사용하고 Mask 레이어에서 렌더링된 픽셀에만 작동하도록 필터링되어 있습니다. Overrides 패널에는 **Stencil** 옵션이 활성화되어 있으며, 버퍼에 저장할 Value는 1로 설정되어 있습니다. 이러한 쓰기가 반드시 진행되도록 하려면 **Compare Function**을 **Always**로 설정하고, **Pass**를 **Replace**로 설정해 항상 기존 값을 대체하도록 해야 합니다. **Fail**과 **Z Fail**은 **Keep**으로 설정합니다.

이제 URP가 Mask 레이어 렌더링을 시도합니다. 설정된 오버라이드 머티리얼이 없으므로 이 Mask 레이어의 오브젝트(MaskMat 머티리얼과 StencilMask 셰이더가 있는 렌즈)에 정의된 머티리얼을 사용하게 됩니다. Compare Function을 **Always**로 설정하고 Pass를 **Replace**로 설정하면 시야에 있는 렌즈에 항상 스텐실 버퍼가 적용되며, 각 픽셀의 값은 1로 설정됩니다.



See Through (Render Objects) 설정

두 번째 렌더 오브젝트 렌더러 기능(위 그림)을 살펴보겠습니다. 이 패스는 **AfterRenderingOpqaues** 이벤트를 사용하도록 설정되어 있으며, 따라서 스텐실 버퍼가 설정된 이후에 적용됩니다. **Layer Mask**는 **SeeThrough**로 설정되어 있고 **Value**는 **1**로 설정되어 있습니다. Value가 1인 곳이 발견되면 픽셀이 렌더링되지 않아야 합니다.

Compare Function 설정은 **Not Equal**로 설정되고 Pass, Fail, Z Fail은 모두 **Keep**으로 설정됩니다. 이 렌더 오브젝트 패스는 스텐실 버퍼에서 읽기만 하고 스텐실 버퍼에 쓰지는 않습니다. 따라서 이 패스는 스텐실 버퍼에서 값이 1이 아닌 See Through 레이어의 모든 픽셀을 렌더링하며, 렌즈가 있는 곳에서만 기본 렌더를 유지합니다. Compare Function을 **Equal**로 변경해 렌즈에만 책상이 나타나도록 결과를 반전해 보세요.



Compare Function을 Equal로 변경했을 때의 효과

렌더러 기능은 극적인 커스텀 효과를 구현하는 훌륭한 방법입니다.

인스턴싱



HoYoverse가 Unity로 제작한 인기 게임 [원신](#)은 무성한 초목이 펼쳐진 광활한 오픈 월드를 선사합니다. 원신은 모바일 디바이스부터 최신 콘솔에 이르는 모든 주요 플랫폼에서 플레이할 수 있습니다. 이 섹션에서는 원신과 비슷한 잔디 효과를 효율적으로 재현하는 방법에 대한 팁을 소개합니다.

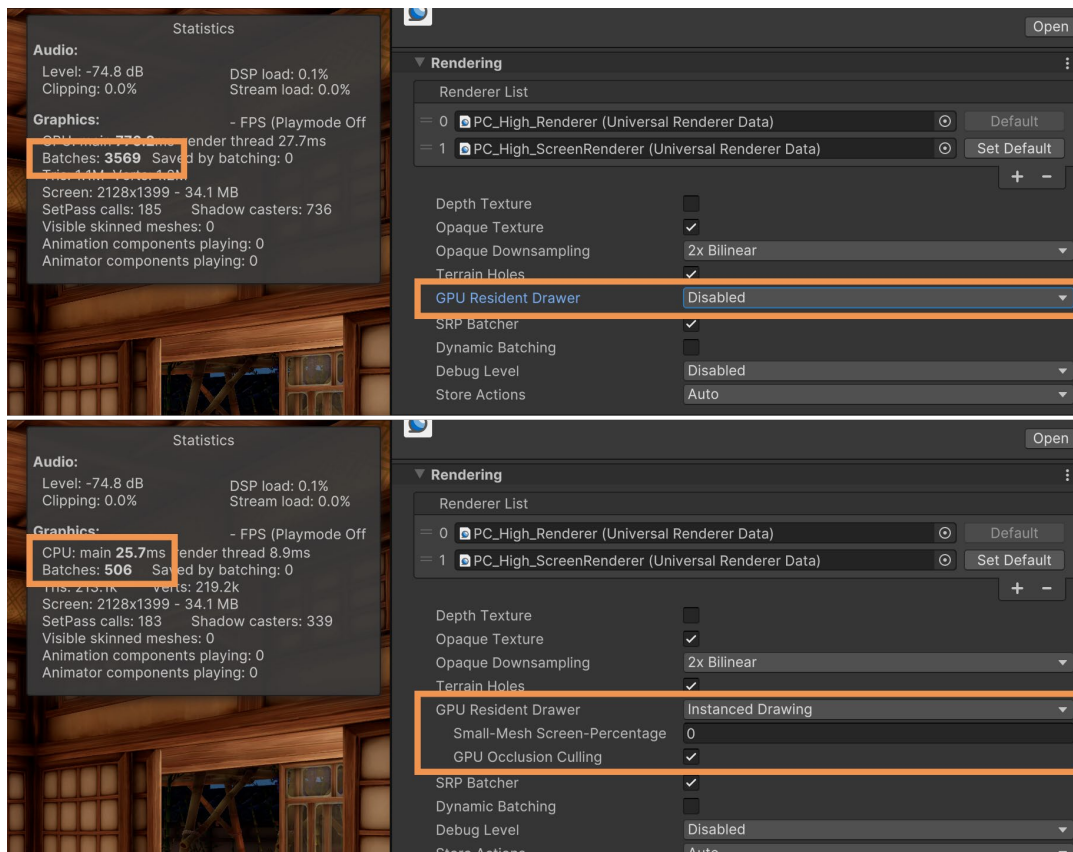
CPU와 GPU 간의 데이터 교환은 렌더링 파이프라인에서 병목 현상을 일으키는 주요 원인입니다. 똑같은 지오메트리와 머티리얼을 사용해 여러 번 렌더링해야 하는 모델이 있다면 Unity에서 제공하는 여러 뛰어난 툴을 활용해 보세요. 이 챕터에서 그러한 툴을 살펴보겠습니다.

GPU 상주 드로어 및 GPU 오클루전 컬링

인스턴싱 레시피를 시작하기 전에, Unity 6에 새로 도입된 일반 솔루션이며 URP 에셋의 Rendering 섹션을 통해 사용할 수 있는 GPU 상주 드로어에 대해 알아보겠습니다.

GPU 상주 드로어는 CPU 시간을 최적화하도록 설계된 GPU 기반 렌더링 시스템입니다. GPU 상주 드로어를 사용하면 게임 오브젝트가 BatchRenderGroup API를 활용하므로 배치가 빨라지고 CPU 성능이 향상됩니다.

GPU 상주 드로어를 사용하면 게임 오브젝트로 게임을 저작(authoring)하고, 처리 과정에서는 인스턴싱을 더 효과적으로 처리하는 특수하고 신속한 경로로 게임 오브젝트를 수집 및 렌더링할 수 있습니다. 이 기능을 활성화하면 드로우 콜이 많은 CPU 바운드 게임에서 드로우 콜의 양이 줄어 병목 현상이 완화되는 것을 확인할 수 있습니다.



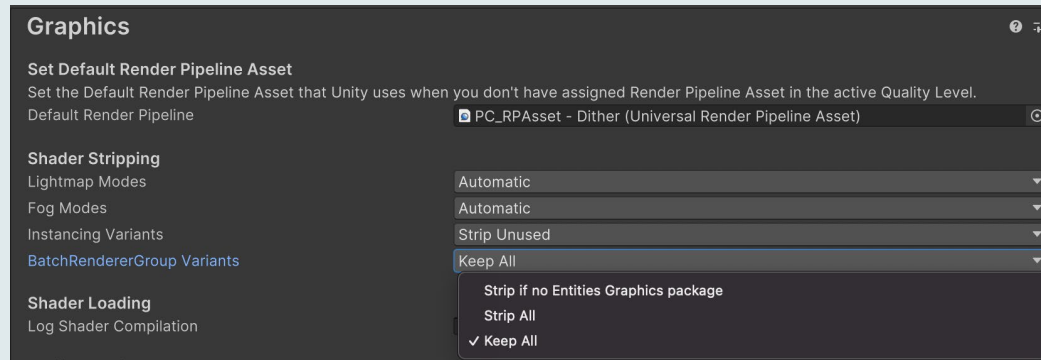
Unity 6의 URP 에셋에서 사용할 수 있는 GPU 상주 드로어 및 GPU 오클루전 컬링 옵션

위 스크린샷에서 **URP 3D 샘플**의 정원 환경을 에디터 모드에서 렌더링하는 데 필요한 배치 횟수는 3,569회입니다. GPU 상주 드로어를 **Instanced Drawing**으로 설정하면 필요한 배치 횟수가 506회로 줄어듭니다.

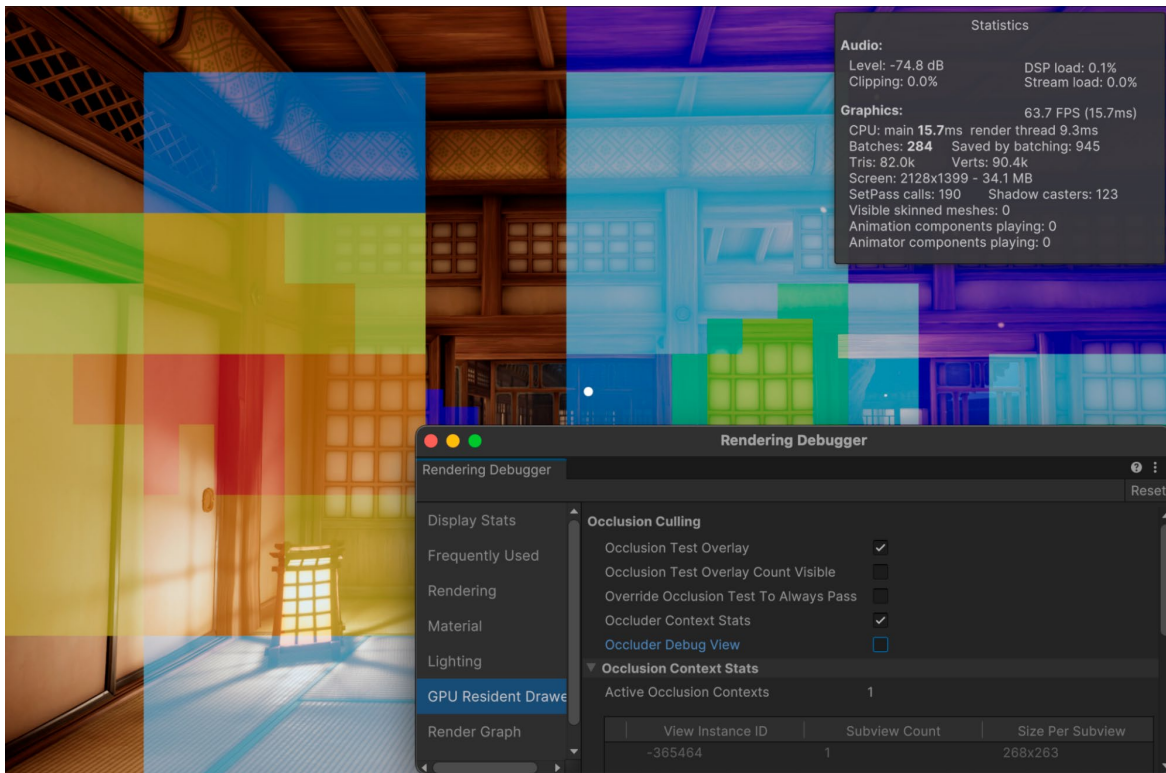
개선되는 정도는 씬의 규모와 사용되는 인스턴싱 양에 따라 달라집니다. 인스턴싱할 수 있는 오브젝트를 많이 렌더링할수록 더 많은 이점을 얻을 수 있습니다.

GPU 상주 드로어는 MeshRenderer를 처리합니다. 스킨드 메시 렌더러, VFX Graph, 파티클 시스템 또는 유사한 효과 렌더러는 처리하지 않습니다. 기존 콘텐츠를 수정하지 않아도 GPU 상주 드로어를 사용할 수 있습니다. 커스텀 셰이더를 사용할 경우 DOTS 인스턴싱과 호환되도록 해야 합니다. [이 간소화된 버전을](#) 예시로 참고하세요.

참고: GPU 상주 드로어를 사용하려면 포워드+ 렌더러를 사용해야 하며, **Project Settings > Graphics > BatchRendererGroup Variants**를 **Keep All**로 설정해야 합니다.



GPU 상주 드로어를 활성화하면 [GPU 오클루전 컬링](#) 옵션도 사용할 수 있습니다. GPU 오클루전 컬링은 화면에 표시되지 않는 요소를 렌더링하지 않는 GPU 기반 접근 방식입니다. 콘텐츠에 따라 CPU 작업을 상당히 줄일 수 있습니다.



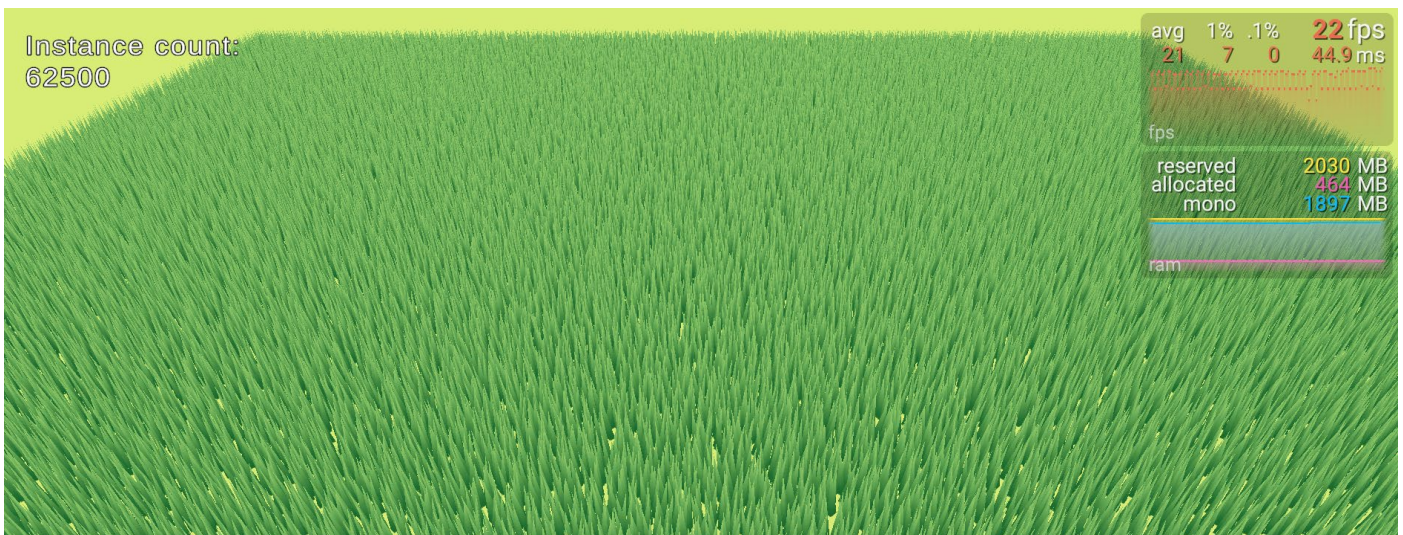
렌더링 디버거를 통한 오클루전 테스트

작업 중인 씬에 GPU 오클루전 컬링이 효과적인지 확인하려면 **Window > Analysis > Rendering Debugger**로 이동하여 **GPU Resident Drawer > Occlusion Test Overlay**를 선택하세요. 그러면 컬링되는 인스턴스의 히트맵이 표시됩니다. 히트맵은 컬링되는 인스턴스가 적을수록 파란색, 컬링되는 인스턴스가 많을수록 빨간색으로 표시됩니다. 이 설정을 활성화하면 컬링 작업이 느려질 수 있습니다.

인스턴싱

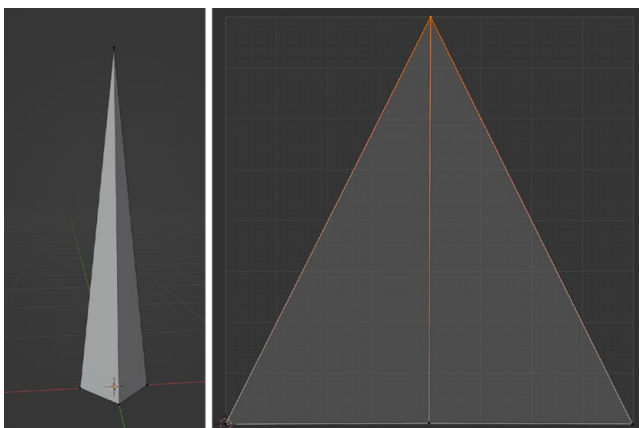
잔디가 가득한 들판을 함께 둘러보며 **인스턴싱**의 개념을 설명하겠습니다. 극사실적인 렌더링과는 거리가 있지만 관련 기법을 설명하는 데는 충분합니다. 예시는 **Scenes > Instancing** 폴더에 있습니다.

참고: 에셋을 제공해 주신 '**Unity에서 GPU 인스턴싱으로 잔디 만들기(영문)**'라는 글의 작성자에게 감사를 전합니다.



SRP Batcher 호환 머티리얼을 사용해 렌더링한 잔디밭

풀잎 한 개와 삼각형 두 개로 간단하게 시작해 보겠습니다. 각 풀잎의 밑면은 V 값이 **0**이고 끝부분은 V 값이 **1**이 되도록 UV를 설정합니다. 이를 사용해 끝부분 버텍스를 오프셋하여 바람을 시뮬레이션할 수 있습니다.

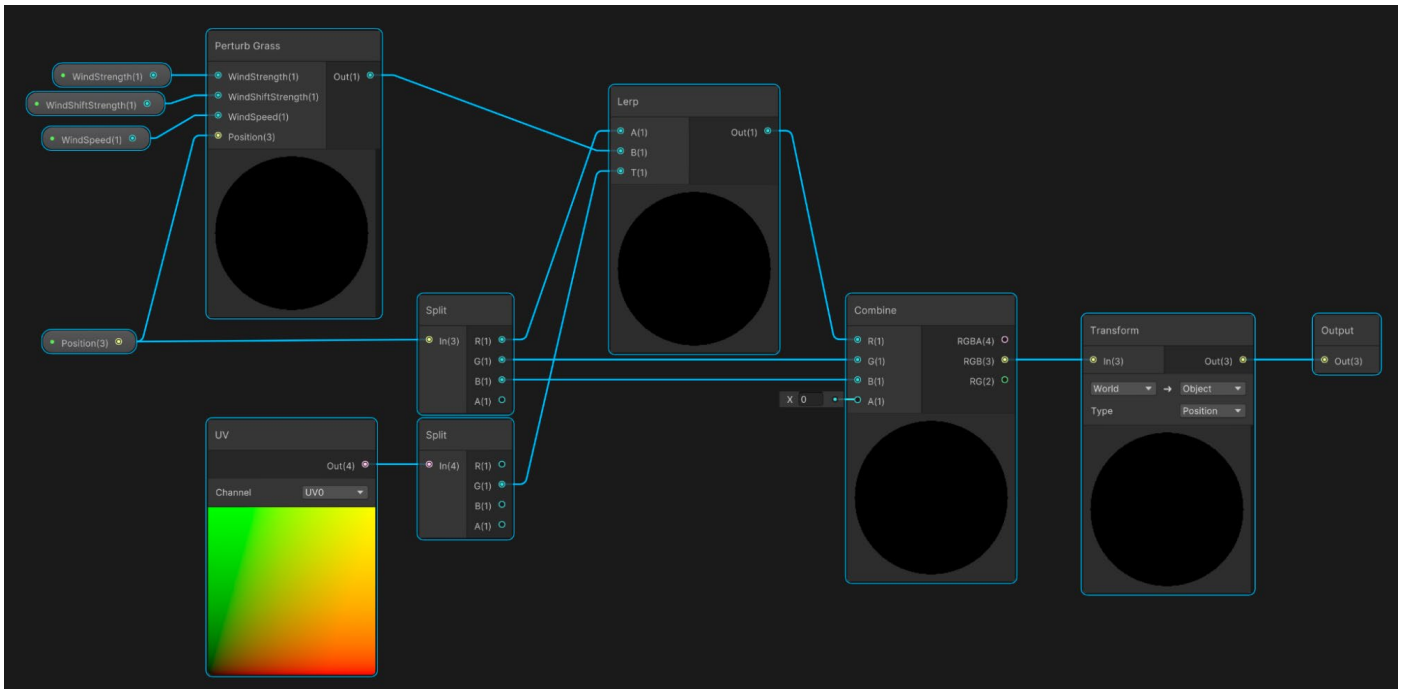


풀잎 모델 및 UV

SRP Batcher

Scenes > Instancing > Common > Grass Wave 폴더에서 Shader Graph 하위 그래프를 살펴보겠습니다. 목표는 WindSpeed, WindShiftStrength, WindStrength에 따라 오브젝트 버텍스의 X 값이 조금씩 바뀌도록 하는 것입니다. 모든 풀잎이 서로 조금씩 다르게 움직이게 하려면 **Perturb Grass** 하위 그래프에서 **Noise** 노드를 사용합니다. 버텍스 Y와 Z 위치는 출력에 직접 전달되지만, X 값에 대한 오프셋은 **Lerp** 노드를 사용해 처리됩니다.

보간을 제어하는 T 입력은 UV의 V 값에서 가져옵니다. 풀잎의 밑면에서 이 값은 0이며, 선형 보간의 결과가 모델링된 위치인 선형 보간의 A 입력이 된다는 의미입니다. 풀잎 끝부분의 V 값은 1로, 선형 보간의 결과가 처리된 오프셋인 B 입력이 되도록 합니다.



Grass Wave 하위 그래프

이제 각 풀잎을 변형하는 방법을 알았으니 완전한 셰이더로 변환해 각 풀잎의 머티리얼 셰이더로 사용해 보겠습니다.

Scenes > Instancing > 1 - SRP Batcher > SRP Batcher Shader 폴더를 확인합니다. SRP Batcher는 간단한 셰이더로, Vertex > Position을 제어하는 Grass Wave 하위 그래프와 프래그먼트 셰이더의 기본 컬러 입력 역할을 하는 Sample Texture 2D로 구성되어 있습니다.

다음 코드를 사용해 잔디밭을 채워 보겠습니다.

```
_startPosition = -_fieldSize / 2.0f;
_cellSize = new Vector2(_fieldSize.x / GrassDensity, _fieldSize.y / GrassDensity);

var grassEntities = new Vector2[GrassDensity, GrassDensity];
var halfCellSize = _cellSize / 2.0f;

for (var i = 0; i < grassEntities.GetLength(0); i++) {
    for (var j = 0; j < grassEntities.GetLength(1); j++) {
        grassEntities[i, j] =
            new Vector2(_cellSize.x * i + _startPosition.x,
                        _cellSize.y * j + _startPosition.y) +
            new Vector2(Random.Range(-halfCellSize.x, halfCellSize.x),
                        Random.Range(-halfCellSize.y, halfCellSize.y));
    }
}
_abstractGrassDrawer.Init(grassEntities, _fieldSize);
```

이 예시 코드를 자세히 살펴보면 다음 내용을 확인할 수 있습니다.

- `_fieldSize`는 (40, 40)입니다.
- `_startPosition`은 (-20, -20)입니다.
- GitHub 샘플에서 `GrassDensity`는 250으로 설정되어 있습니다.
- `cellSize`는 (0.16, 0.16)입니다.
- 루프 두 개가 반복되며 `_grassEntities` 2D 배열의 각 요소를 설정합니다.
- 각 풀잎의 밑면 위치는 `_startPosition`에 현재 셀을 더하고 작은 무작위 인수를 더한 값입니다.
- `_abstractGrassDrawer`는 잔디 채우기 코드를 사용하는 두 가지 버전을 위한 기본 클래스입니다.
 - 최초 버전에서는 **Scenes > Instancing > 1 - SRP Batcher > 1 - SRP**에서 씬을 열고 실행해 GPU 인스턴싱을 무시하고 SRP Batcher가 문제를 얼마나 잘 처리하는지 확인하겠습니다.
 - 먼저 풀잎 모델 프리팹을 `grassEntities` 2D 배열의 각 위치에 배치해서 씬을 채워야 합니다. 이 코드는 **Scenes > Instancing > Scripts > GameObjectGrassDrawer.cs** 파일에 있습니다.


```
public override void Init(Vector2[,] grassEntities, Vector2 fieldSize) {
    _grassEntities = new GameObject[grassEntities.GetLength(0),
                                    grassEntities.GetLength(1)];
    for (var i = 0; i < grassEntities.GetLength(0); i++) {
        for (var j = 0; j < grassEntities.GetLength(1); j++) {
            _grassEntities[i, j] =
                Instantiate(_grassPrefab,
                           new Vector3(
                               grassEntities[i, j].x,
                               0.0f,
                               grassEntities[i, j].y),
                           Quaternion.identity);
        }
    }
}
```

여기서는 인스턴스화를 사용해 **grassEntities** 배열을 반복하여, 할당된 프리팹에서 새 게임 오브젝트를 생성합니다. 이 방식은 제대로 작동하지만 씬의 프레임 속도에 심각한 영향을 줍니다. [29페이지](#)의 잔디밭 이미지를 보면 다음과 같은 사양의 MacBook Air 2022년형에서 실행 시 62,500개의 풀잎을 렌더링할 때 프레임 속도가 29fps로 느려지는 현상을 확인할 수 있습니다.

- 빌트인 Retina 디스플레이
- 프로세서: Apple M2 2022
- 메모리: 8GB

씬을 최적화하려면 어떻게 해야 할까요?

참고: 정사각형이 아닌 터레인의 경우, 목록에 각 풀잎의 위치를 저장하는 드로우 톨을 만들 수도 있습니다. 한 예로 게임 개발자 브론슨 즈겔이 작성한 [이 블로그 게시물](#)은 씬을 클릭할 때마다 씬에 오브젝트를 배치하는 작업을 간소화하는 톨을 어떻게 제작하는지 설명합니다.

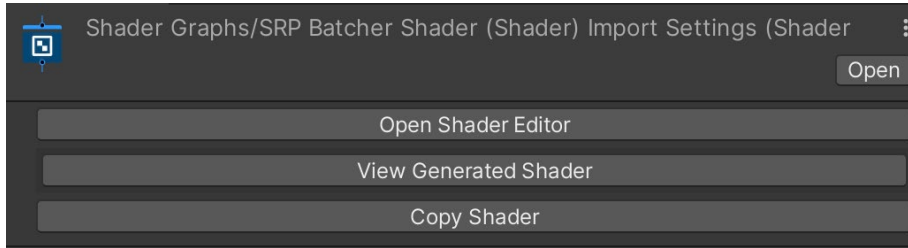
GPU 인스턴싱

GPU 인스턴싱은 유용한 최적화 기법입니다. GitHub 샘플의 **Scenes > Instancing > 2 - GPU Instancing > 2 - GPU Instancing**에서 이 기법의 예시를 확인할 수 있습니다.

Enable GPU Instancing이라는 머티리얼 설정은 렌더러가 동일한 머티리얼을 사용하는 모든 모델을 배치하도록 지시하며, 결과적으로 드로우 콜 수를 줄입니다. 이 설정은 Advanced Options 패널에 있습니다.

SRP Batch와 GPU 인스턴싱은 상호 배타적입니다. URP를 사용할 때 머티리얼이 SRP Batch와 호환되는 경우, Enable GPU Instancing 설정이 선택되어 있어도 SRP Batch를 사용합니다. Shader Graph로 생성한 셰이더는 기본적으로 SRP Batch와 호환됩니다. SRP Batch 호환성을 비활성화하려면

HLSL 셰이더를 생성할 Shader Graph를 선택하고 인스펙터에서 **View Generated Shader**를 클릭합니다.



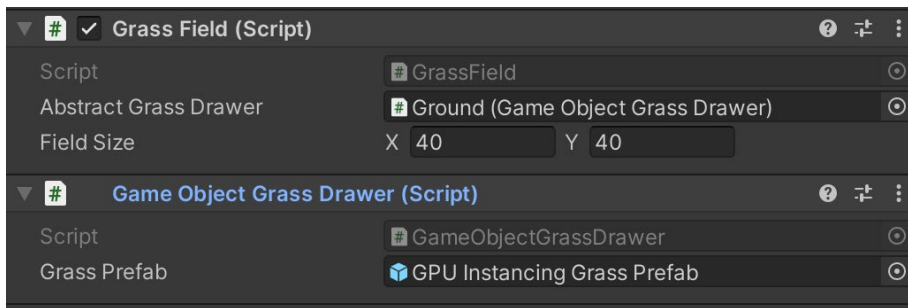
Shader Graph에서 HLSL 셰이더 생성

셰이더가 Temp 폴더에 생성되고, 선택한 텍스트 또는 코드 에디터에서 열립니다. 셰이더의 이름을 Shader “Custom/GPU Instancing Shader”로 변경합니다.

CBUFFER를 검색하고 CBUFFER 매크로를 주석 처리합니다.

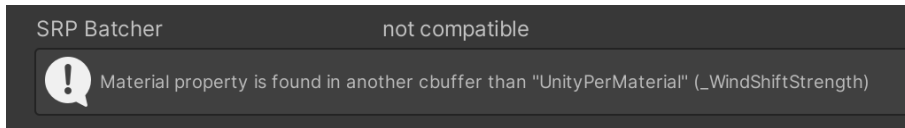
```
// 그래프 프로퍼티
//CBUFFER_START(UnityPerMaterial)
    float4 _MainTexture_TexelSize;
    half _WindShiftStrength;
    half _WindSpeed;
    half _WindStrength;
//CBUFFER_END
```

셰이더를 에셋 폴더에 저장합니다.



GPU Instancing 씬에서 Ground 게임 오브젝트에 할당된 스크립트

GPU Instancing 씬이 SRP Batcher 씬과 같은 버전의 Abstract Grass Drawer를 사용하는 것을 확인할 수 있습니다. 유일한 차이점으로 GPU Instancing 씬의 **GameObjectGrassDrawer** 버전에는 GPU Instancing 셰이더를 사용하는 머티리얼이 있는 다른 프리팹이 할당됩니다.



SRP Batcher와 호환되지 않는 GPU Instancing 셰이더

인스펙터에서 GPU Instancing 셰이더를 확인하면 SRP Batcher와 호환되지 않음을 알 수 있습니다.

코드를 생성하는 데 사용한 그래프를 변경하면 다음 커스터마이징 단계를 반복해야 합니다.

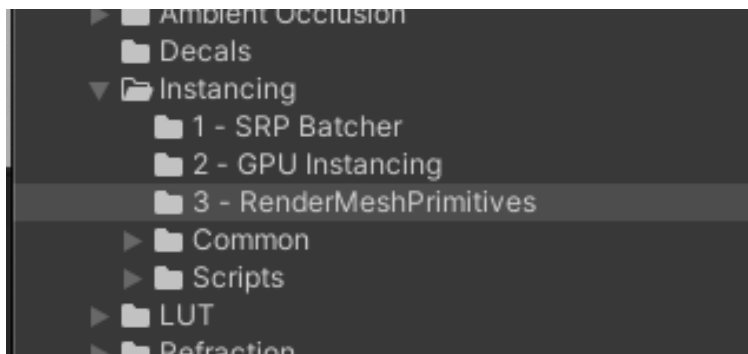
1. 생성된 셰이더 보기 또는 다시 생성
2. 셰이더 이름 수정
3. CBUFFER 매크로 주석 처리
4. 에셋에 저장

그러나 모든 작업을 마치고 테스트해 보면 SRP Batcher에 비해 미미한 정도로만 개선된 결과를 볼 수 있는데, CPU 바운드가 그 원인일 수 있습니다. 따라서 더 좋은 방법을 찾아야 합니다.

RenderMeshPrimitives

Unity [그래픽스 API](#)에는 게임 오브젝트를 건너뛰고 메시를 직접 렌더링하는 다양한 메서드가 있습니다. 여기에 사용된 기능은 Unity LTS 2021에 추가된 [RenderMeshPrimitives](#) 메서드입니다. 전에는 [DrawMeshInstancedProcedural](#)을 사용해야 했지만, 이 메서드는 이제 사용되지 않습니다.

RenderMeshPrimitives에서는 ComputeBuffer를 사용해 개별 메시 위치를 찾는 머티리얼을 사용해야 합니다. **Scenes > Instancing > 3 - RenderMeshPrimitives > 3 - RenderMeshPrimitives** 씬을 보면 어떤 식으로 작동하는지 확인할 수 있습니다.



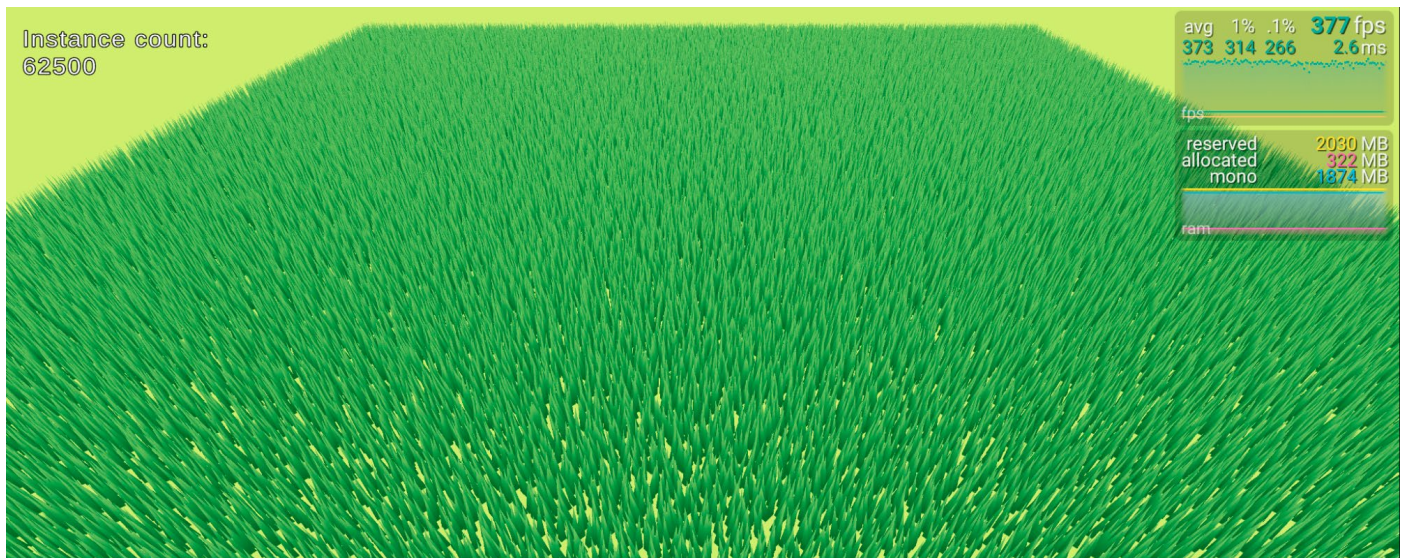
프로젝트(Project) 창의 인스턴싱 씬

아래의 잔디밭 이미지에서 볼 수 있듯이, 프레임 속도가 377fps라는 놀라운 수준으로 향상됩니다. SRP Batcher와 GPU 인스턴싱으로 생성한 씬의 속도는 각각 약 20fps와 50fps였습니다.

여기서 차이점은 잔디밭을 한 번의 드로우 콜로 렌더링하는 것입니다.

Event #4: Draw Mesh (instanced) (1 draw calls, 62500 instances)	
Shader	Shader Graphs/Instanced Grass Shader Graph, SubShader #0
Pass	Universal Forward
Keywords	PROCEDURAL_INSTANCING_ON
Blend	One Zero
ZClip	True
ZTest	LessEqual
ZWrite	On
Cull	Back
Conservative	False

잔디밭의 프레임 디버거 통계



RenderMeshPrimitives를 사용해 렌더링한 잔디밭

이 잔디밭은 각 풀잎의 위치를 Material 프로퍼티로 만들어 구현했습니다. 풀잎을 렌더링하기 위한 데이터는 GPU에 저장되며, GPU는 병렬성을 사용해 전체 잔디밭을 최적의 속도로 렌더링합니다.

위치를 생성하는 코드를 검토해 보겠습니다. 코드는 **Scenes > Instancing > Scripts > InstancedGrassDrawer.cs** 파일에 있는 UpdatePositions 메서드에서 확인할 수 있습니다.

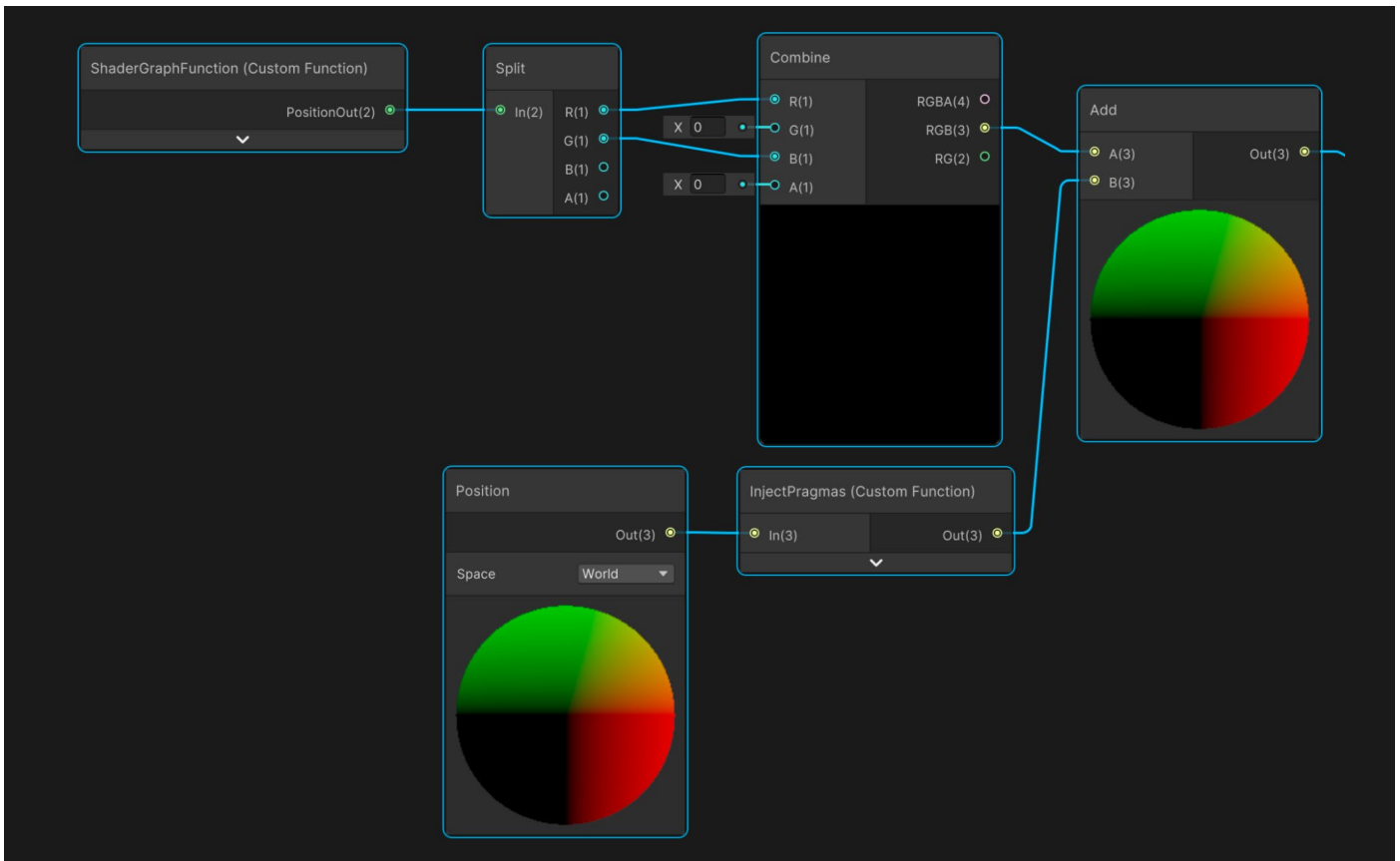
```
_positionsCount = _positions.Count;
_positionBuffer?.Release();
if (_positionsCount == 0) return;
_positionBuffer = new ComputeBuffer(_positionsCount, 8);
_positionBuffer.SetData(_positions);
_instanceMaterial.SetBuffer(Shader.PropertyToID("PositionsBuffer"), _positionBuffer);
```

_positions는 잔디 위치의 Vector2 목록을 저장합니다. _positionsBuffer가 있다면 해제하세요. 변수 뒤에 붙는 '?' 기호는 null 검사이며 다음을 줄여서 표현한 것입니다.

```
if (positionsBuffer != null) _positionsBuffer.Release();
```

카운트 파라미터와 각 항목의 바이트 크기를 사용하는 ComputeBuffer를 생성합니다. Vector2에는 두 개의 플로트가 포함됩니다. 플로트 하나가 32비트, 즉 4바이트이므로, 두 개의 플로트는 8바이트입니다. _positions 목록을 전달하는 SetData를 사용해 간단하게 ComputeBuffer를 채울 수 있습니다. 이제 SetBuffer 메서드를 사용해 이 버퍼를 머티리얼에 복사할 수 있습니다. positionsBuffer라는 이름을 사용하면 머티리얼에서 이 버퍼에 액세스할 수 있습니다.

Scenes > Instancing > 3 - RenderMeshPrimitives > Instanced Grass Shader에 있는 그래프를 살펴보겠습니다.



ComputeBuffer에서 버텍스 위치 가져오기

아래쪽부터 보면 Grass Mesh 버텍스 위치에 대한 **Space** 파라미터가 **World**로 설정되어 있습니다. 하지만 이 기법을 사용하려면 항상 반드시 추가해야 하는 중요한 코드 블록이 있습니다. `#pragma`는 `RenderMeshPrimitive`를 사용해 렌더링된 모든 메시에 필요하며, 커스텀 함수를 사용해 추가할 수 있습니다. 파일에서 함수를 찾는 대신 다음 문자열을 추가하면 됩니다.

```
#pragma instancing_options procedural:ConfigureProcedural
Out = In;
```

이제 위치 값을 생성하기 위해 이 셰이더에서 사용하는 코드 메서드를 `ConfigureProcedural`이라는 함수에서 가져옵니다. 그러한 경우가 아니면 이 [Custom Function 노드](#)는 입력(**In**)을 출력(**Out**)으로 전달하기만 합니다.

까다로운 작업은 씬 파일과 같은 폴더에 있는 `InstancedPosition` 파일의 `ShaderGraphFunction`이라는 커스텀 함수에서 수행됩니다.

```
#if defined(UNITY_PROCEDURAL_INSTANCING_ENABLED)
StructuredBuffer<float2> PositionsBuffer;
#endif

float2 position;

void ConfigureProcedural () {
    #if defined(UNITY_PROCEDURAL_INSTANCING_ENABLED)
        position = PositionsBuffer[unity_InstanceID];
    #endif
}

void ShaderGraphFunction_float (out float2 PositionOut) {
    PositionOut = position;
}
```

위치는 `ConfigureProcedural` 메서드를 사용해 설정되고 스크립트에 `float`와 `half` 버전이 있는 `ShaderGraphFunction`을 사용해 출력에 전달됩니다.

이 시점의 그래프에서 개별 풀잎의 위치는 `float2`이며, 첫 번째 플로트는 X 값이고, 두 번째 플로트는 Z입니다. [Split](#) 노드는 이를 개별 플로트로 변환하고 [Combine](#) 노드는 두 번째 플로트를 세 번째 플로트로 옮깁니다. `Split` 노드와 `Combine` 노드는 개별 플로트를 XYZW가 아닌 RGBA라고 부르지만, G를 B로 옮기면 사실상 Y를 Z로 옮기는 것과 같습니다. 이제 풀잎과 버텍스 위치가 설정되었으니 이를 결합해 버텍스의 실제 월드 위치를 구할 수 있습니다.

이 셰이더가 준비되면, 이제 SRP Batcher와 GPU 인스턴싱 버전에서 사용하는 것과 같은 `WindSpeed`, `WindStrength`, `WindShiftStrength`, `MainTexture` 입력이 있는 머티리얼과 이 셰이더를 함께 사용할 수 있습니다. 유일한 차이점은 각 버텍스의 위치를 계산하는 방법입니다. 풀잎을 어떻게 렌더링하는지 보려면 `InstancedGrassDrawer.cs` 스크립트를 다시 확인해 보세요. 해당 스크립트의 변수는 `GrassField.cs` 스크립트의 `Awake` 메서드에서 호출되는 `Init` 메서드에서 초기화됩니다.



```
public override void Init(Vector2[,] grassEntities, Vector2 fieldSize) {
    _grassEntities = grassEntities;
    _grassBounds = new Bounds(transform.position,
        new Vector3(fieldSize.x, 0.0f, fieldSize.y));
    _positions = new List<Vector2>();
    _renderParams = new RenderParams(_instanceMaterial);
    _renderParams.worldBounds = _grassBounds;
    _renderParams.shadowCastingMode = ShadowCastingMode.Off;
}
```

`Graphics.RenderMeshPrimitives`를 사용하려면 `RenderParams` 인스턴스가 필요하며, 이 인스턴스는 할당된 `_instanceMaterial` 머티리얼에서 생성됩니다. 두 개의 다른 프로퍼티가 추가로 할당됩니다.

실제 렌더링은 다음과 같이 `Update` 콜백을 사용해 이루어집니다.

```
private void Update() {
    if (_positionsCount == 0) return;
    Graphics.RenderMeshPrimitives(_renderParams, _instanceMesh, 0,
        _positionsCount);
}
```

`RenderMeshPrimitives`는 네 개의 파라미터를 취하며 각각 `RenderParams` 인스턴스, 렌더링할 메시, 하위 메시 인덱스, 렌더링할 사본 수를 나타내는 카운트 값입니다. 셰이더를 사용하면 각 사본은 0에서 카운트보다 1 적은 수까지의 값을 갖는 고유한 `unity_InstanceID`를 가집니다.

[ComputeBuffer](#)를 사용하는 렌더링은 빠르며 설정도 매우 간단합니다. `_positionBuffer`를 조정하면 잔디를 깎거나 날려 버릴 수 있습니다. CPU와 GPU 간 데이터 전달을 방지하려면 [ComputeShader](#)로 처리하는 것이 가장 좋습니다. 컴퓨트 셰이더를 사용해 얻을 수 있는 성능 향상에 대해서는 뒤에 나오는 레시피에서 더 자세히 알아보겠습니다.

더 많은 자료

- 이 레시피에서 사용한 [에셋](#)
- `DrawMeshInstancedIndirect`를 사용한 [프로젝트 예시](#)
- GPU 인스턴싱 [기술 자료](#)
- CatLikeCoding의 GPU 인스턴싱 [문서](#)
- 인스턴싱에 [ComputeBuffer](#) 사용

툰 및 아웃라인 셰이딩



Roll7에서 Unity로 제작한 3인칭 액션 슈팅 게임 **롤러드롬**은 셀 셰이딩 기법을 사용하여 마치 만화 같은 그래픽을 연출하는 독특한 방향성의 아트를 자랑합니다. 이 게임의 크리에이터가 참여한 [인터뷰](#)를 꼭 확인해 보세요.

이 레시피는 툰 셰이더와 아웃라인 셰이더를 만드는 일반적인 방법들이 기본 재료입니다.



똑같은 씬의 세 가지 다른 모습: 표준 셰이딩(왼쪽), 단순한 툰 셰이딩(중앙), 툰 셰이딩(오른쪽)

툰 셰이더와 아웃라인 셰이더는 함께 사용되는 경우가 많지만, 저마다 해결해야 하는 문제에는 큰 차이가 있습니다. 툰 셰이더는 URP와 호환되는 Lit 셰이더를 사용해 생성할 수 있는 컬러를 사용하며, 연속적인 그레디언트 대신 출력 램프(ramp)를 사용하기 때문에 커스텀 조명 모델이 필요합니다.



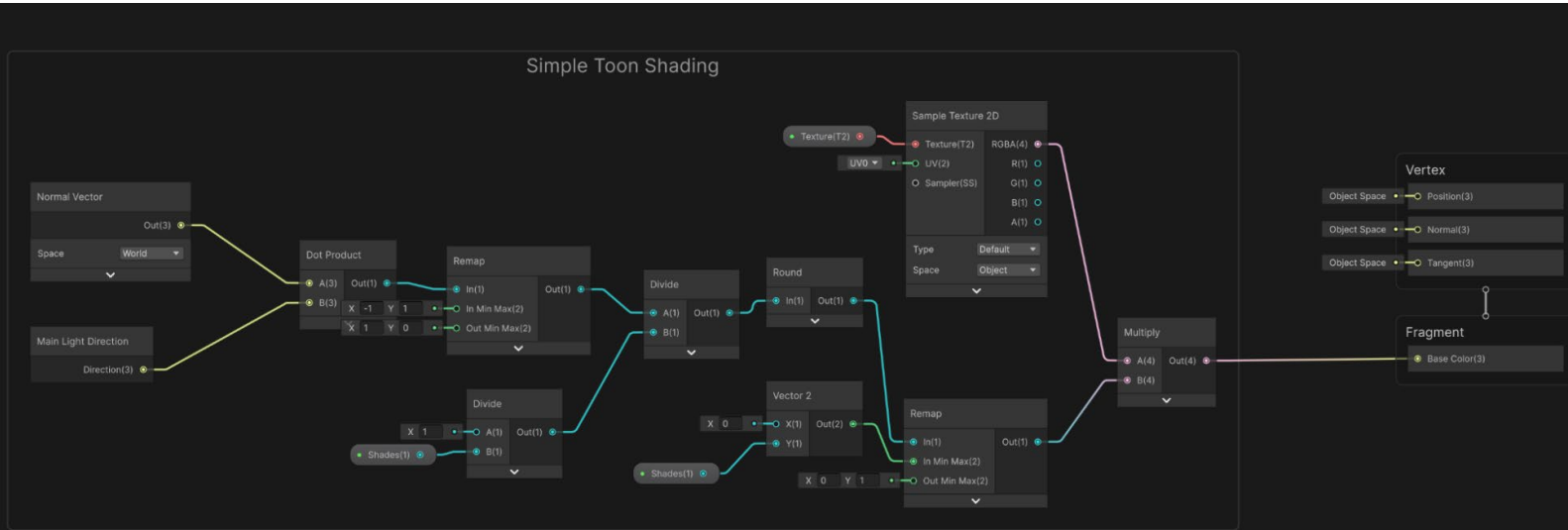
간단한 램프가 적용된 툰 셰이더를 사용한 씬

단순한 툰 셰이딩

실제 셰이더의 모습은 **Scenes > Toon Shading > Simple Toon Shading**에서 확인할 수 있습니다. 두 개의 셰이더로 나누어 과제를 진행할 것이며, 첫 번째 셰이더가 모델에 속한 각 픽셀의 주요 컬러 옵션이 됩니다.

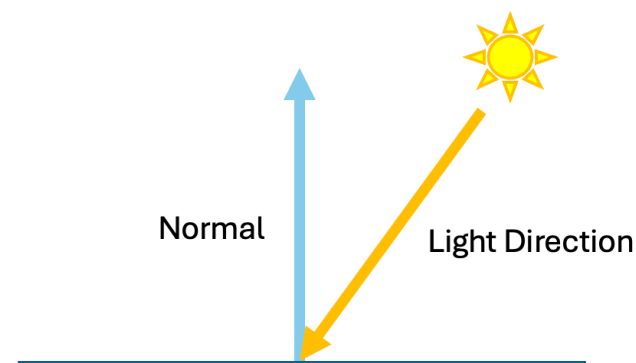
셰이딩

가장 단순한 조명 모델은 **람버트 조명**을 사용합니다. 이 조명 모델은 이 지점에서 광원 방향과 월드 공간 노멀의 내적을 취합니다. 아래 이미지에서 이 셰이더의 Shader Graph를 볼 수 있습니다.



간단한 툰 셰이딩 Shader Graph

왼쪽에는 **Normal Vector** 노드와 **Main Light Direction** 노드가 있습니다. 이 두 벡터의 출력이 **Dot Product** 노드로 입력됩니다. 여기서 출력은 두 벡터가 인라인이라면 1, 두 벡터의 방향이 서로 반대라면 -1이 됩니다. 조명의 경우 두 벡터가 반대 방향일 때 최댓값을 얻는 것이 바람직합니다.



람버트 조명 모델

여기서는 **Remap** 노드가 사용되었습니다. 이 유용한 노드는 일정 범위 내의 입력 값을 다른 범위를 기준으로 한 새로운 값으로 변환합니다. **In Min Max**는 **-1, 1**로 설정되어 있고 **Out Min Max**는 **1, 0**으로 설정되어 있습니다. In 값이 -1이면 Out 값은 1이 되고, In 값이 1이면 Out 값은 0이 됩니다.

이 그래프에는 Texture와 Shades라는 두 개의 프로퍼티가 있습니다. Shades는 0~1 범위에서 몇 개의 램프 단계가 허용되는지를 정의하는 정수 값입니다. 연속적인 보간을 적용하는 대신 이산 값을 산정하는 메카닉이 필요합니다. 먼저 **Divide** 노드를 사용하여 1을 Shades 프로퍼티로 나눕니다. 이 시점에서 Shades가 4이면 Divide 노드의 출력은 0.25가 됩니다. 그런 다음 두 번째 Divide 노드를 사용하여 Remap 노드의 출력을 Shades Divide 노드의 출력으로 나눕니다. 그러면 범위가 [0, 1]에서 [0, Shades]로 변경됩니다. **Round** 노드를 사용하여 이 값을 정수 값으로 변환합니다. 이 시점에서 가능한 값은 0, 1, ..., Shades로 한정됩니다.

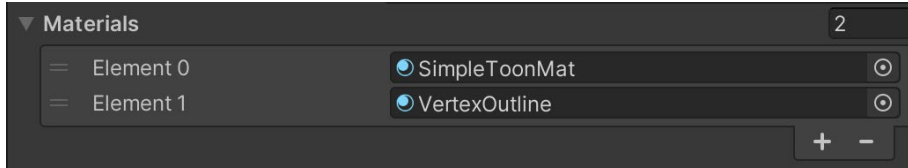
아웃라인 설정

가장 간단하게 아웃라인을 추가하는 방법은 두 번째 패스를 추가하는 것인데, 이 패스는 뒷면을 향한 폴리곤만 렌더링하고 버텍스 노멀을 따라 버텍스를 약간 옮기는 버텍스 셰이더를 사용합니다. 이 셰이더는 **Scenes > Toon Shading > VertexOutline**에 있는 GitHub 샘플에 있으며 그래프는 아래와 같습니다.



배면 버텍스 변환 기법을 사용한 아웃라인 셰이더

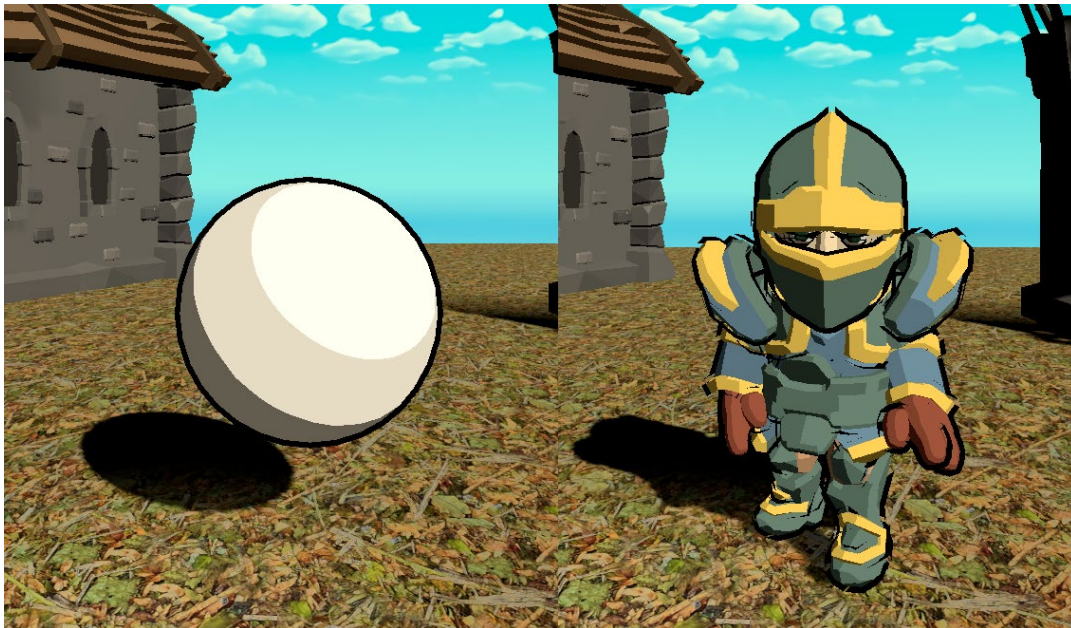
Space가 **Object**로 설정되어 있는 Normal Vector 노드의 출력은 **Multiply** 노드에 입력됩니다. 여기에 머티리얼의 **Thickness** 값을 곱합니다. 이 출력값이 Object Position 값에 더해지면서, 버텍스의 위치가 모델링된 오브젝트 위치에서 약간 바깥쪽으로 이동합니다. 이 값은 **Vertex Position** 프로퍼티의 입력이 됩니다. **Graph Inspector > Graph Settings**의 패널에서 셰이더 프로퍼티 **Universal > Render Face**는 **Back**으로 설정되어 있습니다. Shader Graph에서는 싱글 패스만 사용할 수 있으므로, 렌더링에 추가하려면 게임 오브젝트 인스펙터를 사용해 두 번째 머티리얼을 추가해야 합니다.



두 번째 머티리얼 추가

Scenes > Toon Shading > Simple Toon Shading의 씬에서 두 번째 머티리얼을 사용하는 모습을 확인할 수 있습니다. 같은 폴더에 있는 VertexOutline 머티리얼을 인스펙터에서 확인하고 Thickness를 0.02로 설정합니다.

이 기법은 간단한 컨벡스 셰이프에 효과적입니다. 그러나 아래 이미지에서 볼 수 있듯이 셰이프가 복잡해짐에 따라 선의 두께가 일관되게 구현되지 않습니다. 대안을 살펴보겠습니다.



배면법을 사용한 아웃라인 구현: 간단한 구(왼쪽), 복잡한 모델(오른쪽)

툰 셰이딩

대체 셰이더의 실제 모습은 **Scenes > Toon Shading > Toon Shading**에서 확인할 수 있습니다. 이전 작업처럼, 이 셰이딩 작업은 셰이딩과 아웃라이닝이라는 두 개의 셰이더로 나누어 진행합니다. 첫 번째 단계는 모델에 속한 각 픽셀의 주요 컬러 옵션을 설정하는 것입니다.

셰이딩

Main Light의 유일한 Shader Graph 노드는 **Main Light Direction** 노드입니다.

참고: URP에서 Main Light는 가장 강도가 높은 직접광입니다.



Main Light Direction 노드는 단순한 톤 셰이딩 레시피에서 사용되었습니다. 광원 컬러와 그림자를 추가하여 이 단순한 버전을 개선할 수 있습니다. 먼저 커스텀 함수를 사용해 Main Light에 액세스하겠습니다. 커스텀 함수는 **Shaders > HLSL > Custom Lighting.hlsl** 파일에서 확인할 수 있습니다.

```
void MainLight_float(float3 WorldPos, out float3 Direction, out float3 Color, out float
DistanceAtten, out float ShadowAtten)
{
#ifdef SHADERGRAPH_PREVIEW
    Direction = float3(0.5, 0.5, 0);
    Color = 1;
    DistanceAtten = 1;
    ShadowAtten = 1;
#else
    float4 shadowCoord = TransformWorldToShadowCoord(WorldPos);

    Light mainLight = GetMainLight(shadowCoord);
    Direction = mainLight.direction;
    Color = mainLight.color;
    DistanceAtten = mainLight.distanceAttenuation;

    #if !defined(_MAIN_LIGHT_SHADOWS) || defined(_RECEIVE_SHADOWS_OFF)
        ShadowAtten = 1.0f;
    #else
        ShadowSamplingData shadowSamplingData = GetMainLightShadowSamplingData();
        float shadowStrength = GetMainLightShadowStrength();
        ShadowAtten = SampleShadowmap(shadowCoord,
            TEXTURE2D_ARGS(_MainLightShadowmapTexture,
            sampler_MainLightShadowmapTexture),
            shadowSamplingData, shadowStrength, false);
    #endif
#endif
}
```

[Shader Graph 에셋](#)을 만드는 동안 동작을 정의하는 코드 블록을 `#ifdef SHADERGRAPH_PREVIEW` 프리 프로세서 지시문 안에 추가하는 것이 좋습니다. 그러면 그래프 미리 보기 창에서 기본값으로 사용할 값을 지정할 수 있습니다.

WorldPos는 함수 TransformWorldToShadowCoord를 사용하는 그림자 좌표로 변환됩니다. 이 코드에 사용된 함수는 [Universal Render Pipeline 패키지](#)에 있으며, Shader Graph의 커스텀 함수에서 사용할 수 있습니다. 함수 GetMainLight를 float4와 함께 사용하면, 반환되는 광원의 **ShadowAttenuation** 프로퍼티가 설정됩니다. 이 프로퍼티는 이 커스텀 함수를 사용하는 그래프에 필요합니다.

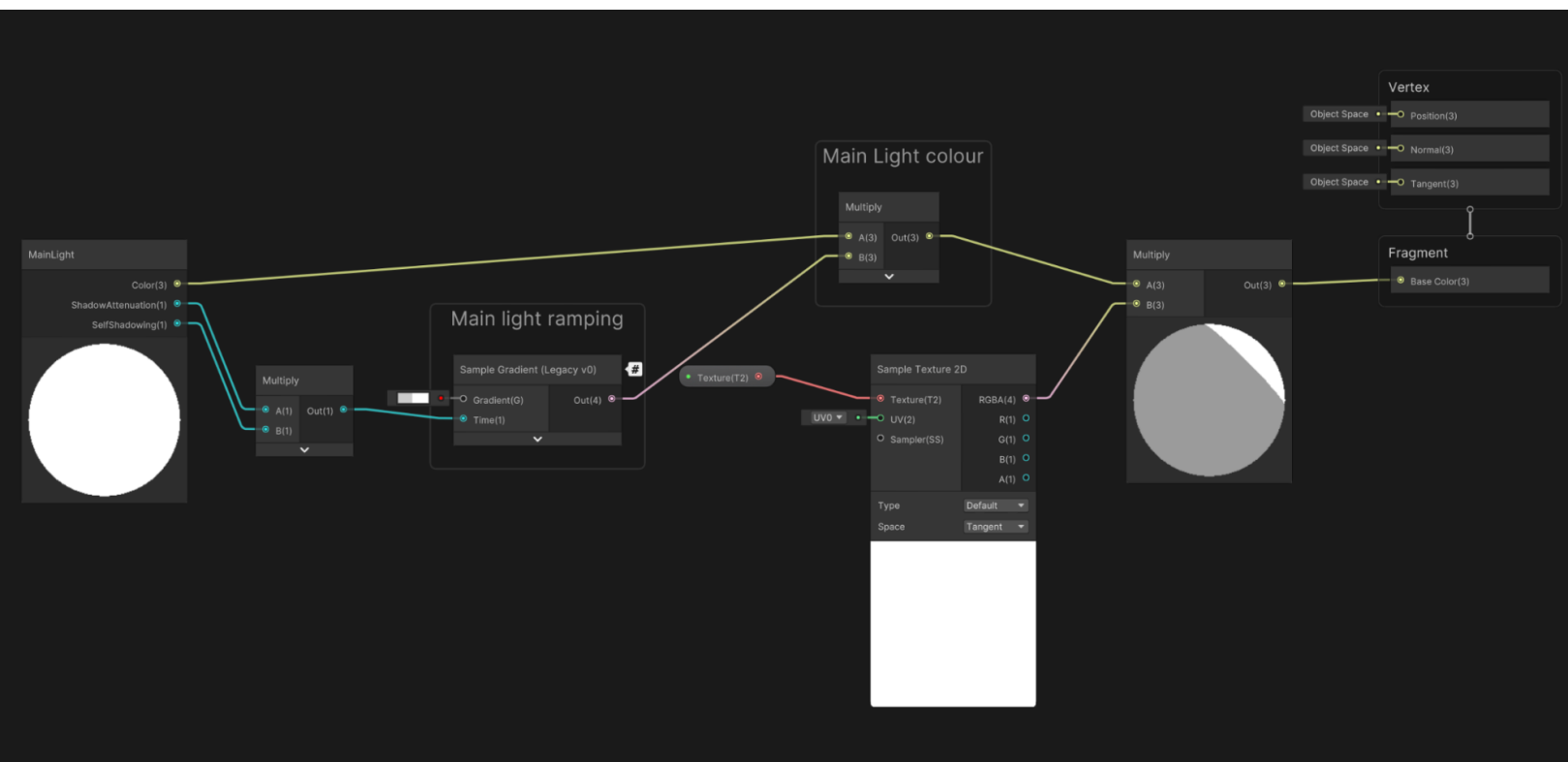
이 코드는 **Shaders > Subgraphs** 폴더에 있는 Main Light 하위 그래프(아래 이미지 참조)에서 사용됩니다. 자세히 한번 살펴보겠습니다.



Main Light 하위 그래프

Custom Function 노드는 **Absolute World**로 설정된 **Position** 노드를 유일한 입력으로 받습니다. 이 함수는 **Direction**, **Color**, **DistanceAtten**(사용되지 않은 채로 유지), **ShadowAtten**을 반환합니다. 셀프 섀도잉을 허용하려면, 광원 방향과 월드 노멀의 내적을 구하고 이 값을 0에서 1 사이로 고정해야 합니다. 음수 값은 필요하지 않습니다.

이제 Main Light에 액세스할 수 있는 경로가 생겼으므로, 이를 사용해 간단한 툰 셰이더를 만들 수 있습니다. **Scenes > Toon Shading > Toon Shading**에서 그래프(아래 이미지)를 확인해 보세요.



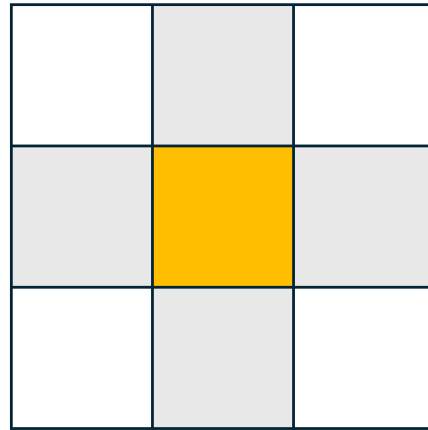
간단한 툰 그래프

첫 번째 노드는 Main Light 하위 그래프입니다. ShadowAttenuation의 출력과 SelfShadowing의 출력은 함께 곱해집니다. 그러면 이 출력이 램프 그래디언트와 함께 작동하는 **Sample Gradient** 노드로 전달되므로 광원 레벨이 매끄럽지는 않지만 그래디언트에 따라 단계적으로 바뀌도록 할 수 있습니다.

매끄럽게 변화하는 입력을 사용해 그래디언트로 처리하는 기법은 여러 셰이딩 과제에서 유용합니다. 그래프의 나머지 부분은 광원 컬러를 램프 레벨과 결합한 후, 샘플링된 텍스처와 결합해 기본 컬러에 사용할 컬러를 생성합니다.

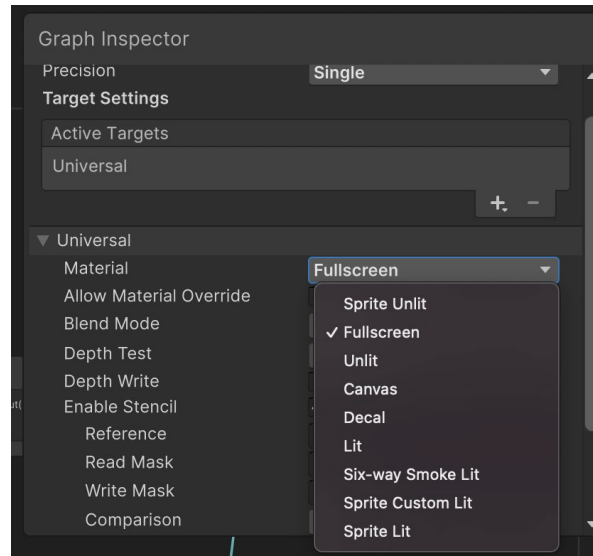
아웃라이닝

더 정교한 아웃라이닝 기법에서는 [소벨 필터](#) 에지 감지를 사용합니다. 이를 위해 포스트 프로세싱 기법을 사용할 수 있습니다. 현재 분석되고 있는 픽셀이 오른쪽 이미지의 주황색 정사각형이라고 가정해 보겠습니다. 이 주황색 픽셀의 버퍼가 회색 정사각형의 버퍼와 어떻게 다른지 확인해 보려고 합니다. 확인할 버퍼는 컬러 G버퍼와 노멀 G버퍼입니다. 배리이션이 프로퍼티 강도 값을 충족하는 경우 주황색 픽셀이 에지 경계에 있다고 간주하고 컬러를 **Outline Color** 프로퍼티로 업데이트할 수 있습니다.



소벨 필터링

이를 위해 **Create > Shader Graph > URP > Full Screen Shader Graph**를 사용하여 Shader Graph를 생성할 수 있습니다. 또는 Graph Inspector를 사용하여 기존 Shader Graph의 동작을 변경할 수 있습니다. Graph Inspector는 Shader Graph 창 오른쪽 상단의 아이콘(원 안의 'i' 아이콘)을 클릭하여 액세스할 수 있습니다.

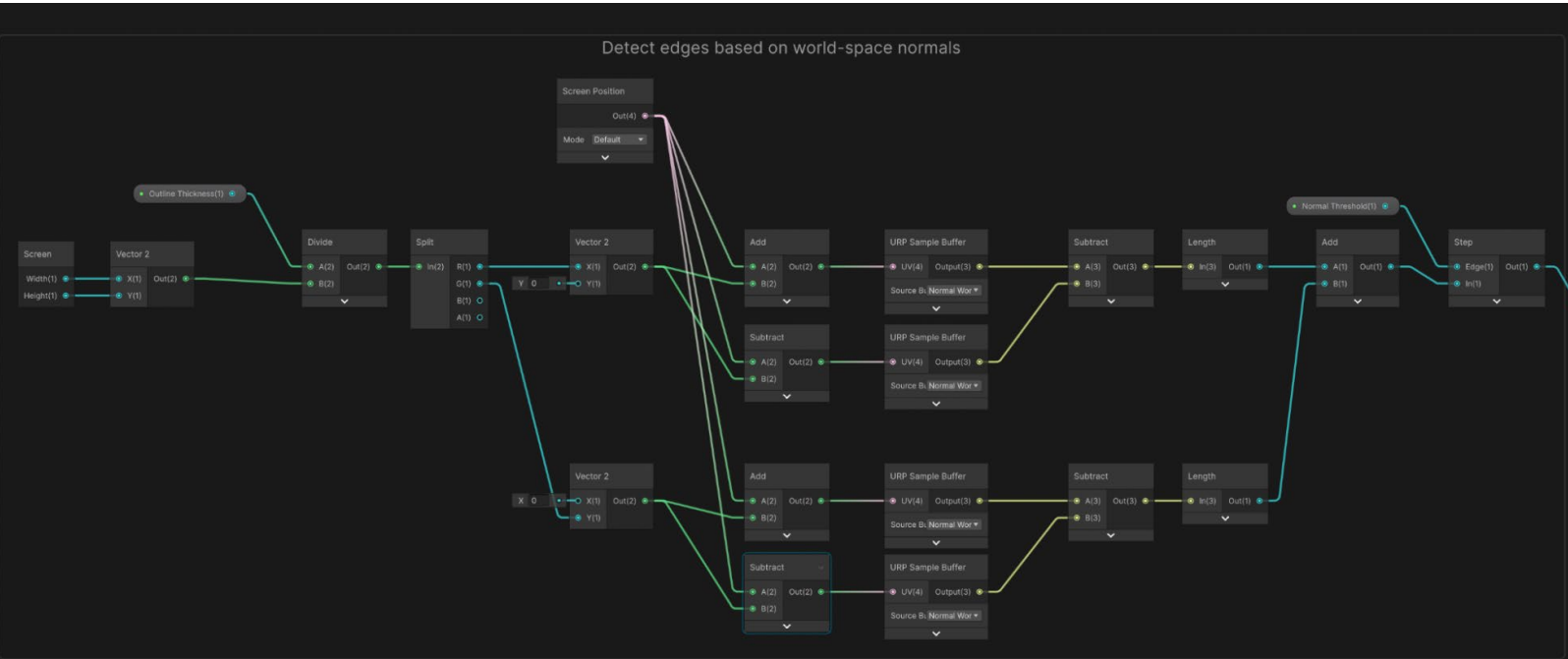


Shader Graph Inspector의 Material 프로퍼티 드롭다운

Scenes > Toon Shading > Outline에서 Shader Graph를 봅니다. 먼저 그래프에서 **Detect edges based on the world space normals** 섹션의 왼쪽부터 살펴보겠습니다.

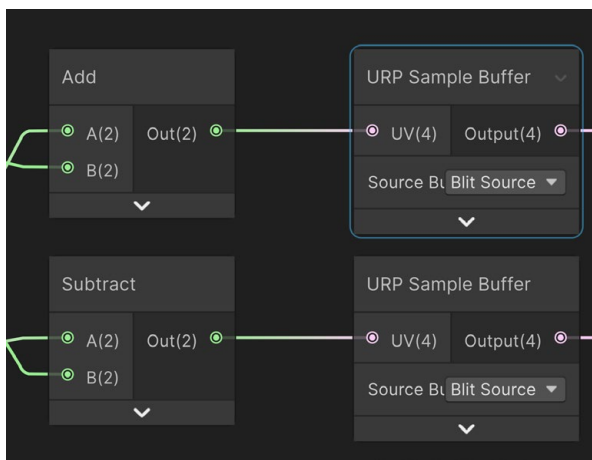
1. 첫 번째 노드는 렌더링된 이미지의 크기를 픽셀 단위로 제공하는 **Screen** 노드입니다. 이를 **Vector2**로 전환할 수 있습니다.
2. 그런 다음 Divide 노드를 사용하여 **Outline Thickness** 프로퍼티를 이 벡터로 나눕니다. 그러면 오프셋 값으로 사용할 수 있는 값이 나옵니다.
3. **Split** 노드를 사용하여 오프셋 값의 x 및 y 값에 액세스할 수 있습니다. Split 노드는 RGBA 출력값(r=x, g=y)을 제공합니다.
4. Vector2 두 개를 생성합니다. 하나는 오프셋의 x가 되고, 다른 하나는 오프셋의 y가 됩니다. x 오프셋에서는 y=0이고, y 오프셋에서는 x=0입니다. 이제 **Add** 노드와 **Subtract** 노드를 사용하여 현재 **Screen Position**에서 오프셋을 더하고 빼 수 있습니다.
5. 이 시점에서는 스크린 공간에 4개의 위치가 있습니다. 왼쪽 하단에는 [0,0]이 있고 오른쪽 상단에는 [1,1]이 있습니다. URP Sample Buffer 노드 4개를 사용하여 이 위치에서의 월드 노멀을 구합니다.

- 오른쪽 노멀에서 왼쪽 노멀을 빼서 이 새 벡터의 **Length**를 구하고, 위쪽 노멀에서 아래쪽 노멀을 빼서 벡터의 **Length**를 구합니다.
- 이제 현재 스크린 픽셀 왼쪽에 있는 픽셀 및 오른쪽에 있는 픽셀 사이의 거리의 스칼라 값과 아래쪽에 있는 픽셀 및 위쪽에 있는 픽셀 사이의 거리의 스칼라 값을 구했습니다.
- Add 노드를 사용하여 이들 값을 더하여 분석 대상인 네 개 픽셀의 차이 합계를 스칼라 값으로 구합니다. 분석을 완료하려면 **Step** 노드를 사용합니다. 해당 값이 **Normal Threshold** 프로퍼티보다 크면 1이 반환됩니다.



월드 공간 노멀을 기준으로 에지 감지

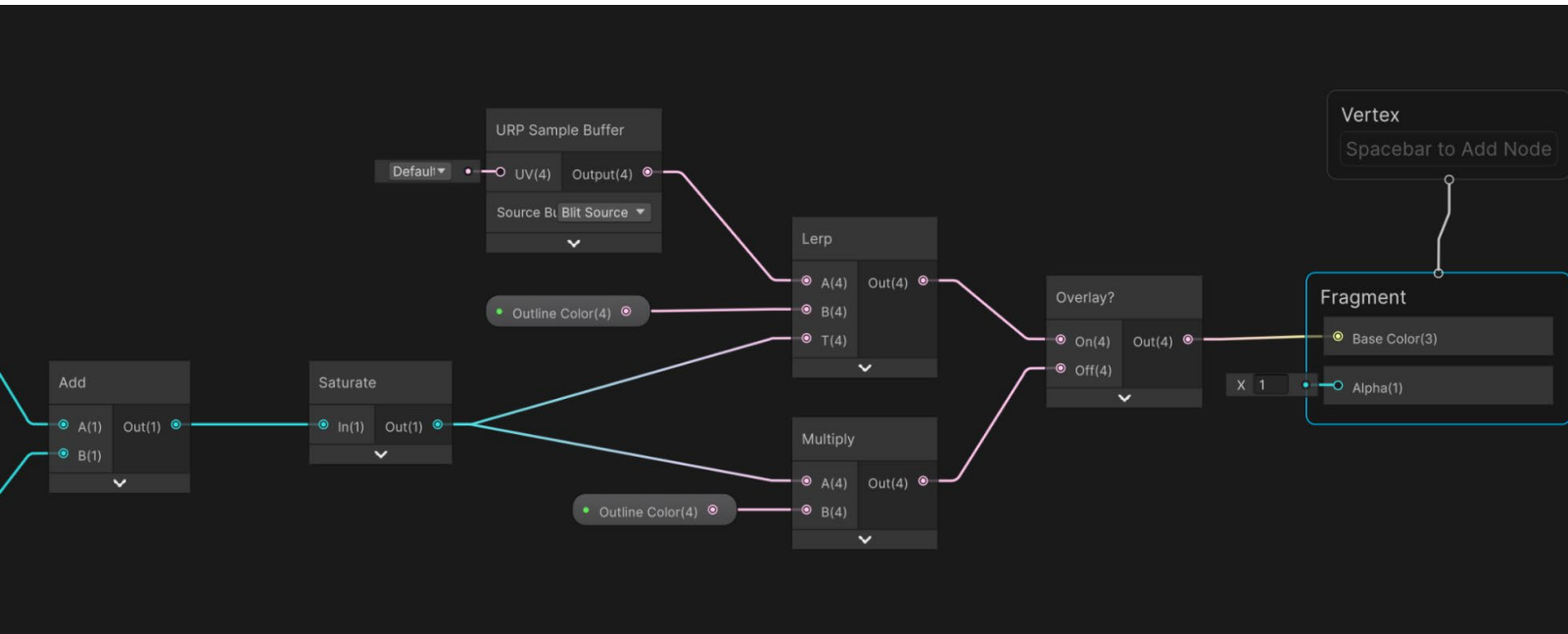
- 컬러 버퍼에도 비슷한 작업을 수행하되, Normal World Space 대신 Blit Source에 액세스하고, Normal Threshold 대신 Color Threshold를 사용합니다.



컬러 에지를 감지하기 위해 샘플 Source Buffer가 Blit Source로 설정됩니다.



10. 9단계를 마치면 값이 0 또는 1인 출력값 2개가 산출됩니다. 아래에 그래프의 나머지 부분이 나와 있습니다. 왼쪽에서 오는 선은 노멀 에지 감지의 출력(위쪽)과 컬러 에지 감지의 출력(아래쪽)입니다. Add 노드를 사용하여 두 값을 더하면 0, 1 또는 2인 값이 나옵니다.
11. **Saturate** 노드를 사용하여 이 값을 0과 1 사이로 고정합니다.
12. 그런 다음 **Lerp** 노드를 사용하여 URP Sample Buffer 노드와 Outline Color 프로퍼티를 입력받아서 기존 픽셀 사이를 보간합니다. Overlay를 사용해도 됩니다. Overlay가 **On**으로 설정되면 기존 Blit Source와 Outline Color 프로퍼티가 합성된 결과물이 산출됩니다. Overlay가 **Off**로 설정되면 Outline Color 또는 검정색이 통과합니다. 최종 출력은 Fragment Base Color 입력으로 연결됩니다.



아웃라인 셰이더의 최종 노드

이 레시피는 툰 셰이딩에 필요한 주요 기법을 제공합니다.

더 많은 자료

- [Unity Open Project](#) GitHub(이 챕터에 사용된 대부분의 코드는 Open Project에서 가져옴)
- [YouTube의 Unity Open Project](#)
- Ned Makes Games의 [YouTube 튜토리얼](#)(툰 셰이딩에 관한 간략한 시리즈 포함, 영문)
- AE Tuts의 Unity SobelFilter.shader에 관한 [YouTube 튜토리얼](#)(Shader Graph 튜토리얼 채널, 영문)
- Alexander Ameye의 [소벨 필터를 사용한 에지 감지](#)(영문)
- Daniel Ilett의 Cel Shading [시리즈](#)와 [전체 화면 아웃라인 셰이더 튜토리얼](#)(영문)

앰비언트 오클루전



[Circuit Superstars](#)는 Original Fire Games에서 Unity로 제작한 레이싱 게임으로, SSAO(스크린 공간 앰비언트 오클루전) 같은 URP 기능을 사용해 자동차와 모델을 환경에 배치하고 깊이 있는 비주얼을 구현했습니다.



앰비언트 오클루전

앰비언트 오클루전은 주름, 구멍, 교차 지점, 서로 가까운 표면을 어둡게 만드는 포스트 프로세싱 기법입니다. 현실에서 이러한 영역은 주변광을 차단하거나 가리는 경우가 많아 더 어둡게 보입니다. 위 이미지에서 왼쪽은 앰비언트 오클루전 없이 렌더링한 결과이고, 오른쪽은 앰비언트 오클루전을 적용해서 렌더링한 결과입니다. 계단 주변의 에지가 어둡게 표현된 것을 확인해 보세요.

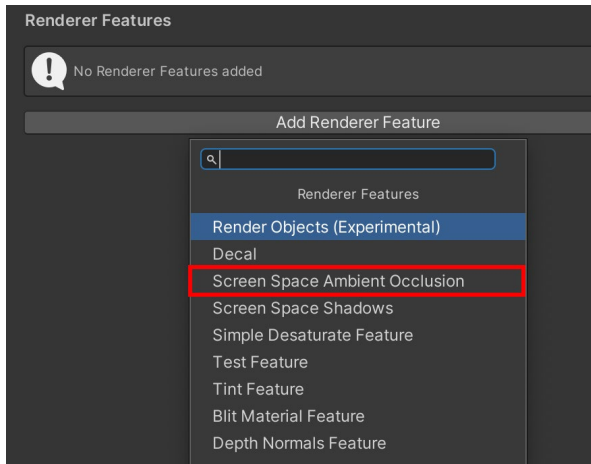
URP는 렌더러 기능으로 실시간 SSAO([스크린 공간 앰비언트 오클루전](#)) 효과를 구현합니다. 사용하는 패스 코드는 [여기](#)에서 확인할 수 있습니다.

참고: SSAO 효과는 렌더러 기능이며, URP의 포스트 프로세싱 효과와 별개로 작동합니다. 이 효과는 불륨에 의존하거나 불륨과 상호 작용하지 않습니다.

Scenes > Ambient Occlusion > Ambient Occlusion에서 실제로 어떻게 작동하는지 확인할 수 있습니다. 이 씬은 Unity 에셋 스토어에서 [무료 에셋](#)으로 제공되는 로우 폴리곤 도시 환경으로 제작되었습니다.

이 씬에서는 **Ambient_Occlusion_URP_Settings**라는 URP 에셋을 사용합니다. Scene > Main Camera에 연결된 **AutoLoadPipelineAsset** 스크립트를 통해 씬을 열면 에셋이 자동으로 로드됩니다. URP 에셋은 **Ambient_Occlusion_URP_Settings_Renderer**를 사용합니다.

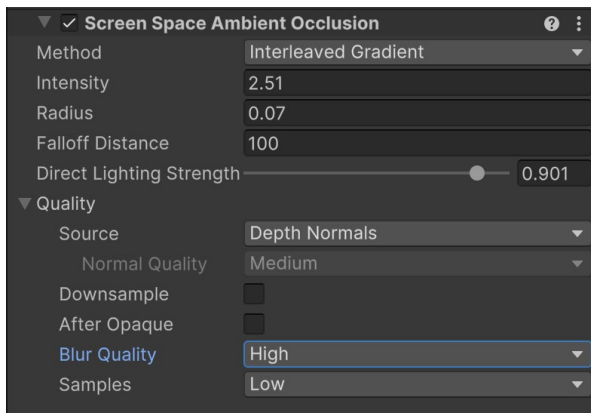
씬에 SSAO를 추가하려면 인스펙터에서 Universal Renderer Data 에셋을 확인하고 **Add Renderer Feature**를 클릭합니다. 옵션 드롭다운 메뉴에서 **Screen Space Ambient Occlusion**을 선택합니다.



SSAO 렌더러 기능 추가

SSAO 프로퍼티

SSAO 렌더러 기능을 추가하고 나면 인스펙터에서 결과를 제어할 수 있습니다. 사용할 수 있는 프로퍼티를 살펴보겠습니다.



SSAO 옵션

- **Method:** Interleaved Gradient와 Blue Noise 중에서 선택합니다.
- **Intensity:** Intensity는 어둡게 표현하는 강도를 조절합니다.
- **Radius:** 현재 픽셀 주변의 노멀 텍스처 샘플을 얼마나 많이 사용할지 제어합니다. 값을 크게 설정할수록 성능에 주는 영향도 커지므로, 최대한 낮은 값을 사용하는 것이 좋습니다. Radius 값은 카메라에서 타겟 픽셀에 렌더링되는 오브젝트까지의 거리에 따라 스케일링됩니다.
- **Falloff Distance:** 앰비언트 오클루전이 적용되지 않는 씬 거리를 제어합니다.



- **Direct Lighting Strength:** 이 프로퍼티는 조명 계산이 수행될 때 처리되는 것에 영향을 받으므로 **After Opaque** 옵션을 비활성화해야 합니다. 이 옵션은 직접광을 받는 앰비언트 오클루전의 강도에 영향을 줍니다.
- **Quality > Source:** 이 옵션은 노멀 벡터 값의 소스를 선택합니다. SSAO 렌더러 기능은 노멀 벡터를 사용해 표면의 각 지점이 주변광에 노출되는 정도를 계산합니다. **Source로 선택할 수 있는 옵션**은 다음과 같습니다.
 - **Depth Normals:** SSAO가 DepthNormals 패스로 생성된 노멀 텍스처를 사용합니다. 이 옵션을 사용하면 Unity에서 더 정확한 노멀 텍스처를 사용할 수 있습니다.
 - **Depth:** SSAO가 뎁스 텍스처를 대신 사용해 노멀 벡터를 재구성합니다. 이 옵션은 커스텀 셰이더에서 DepthNormals 패스 블록을 사용하지 않으려는 경우에만 사용합니다. 이 옵션을 선택하면 **Normal Quality** 프로퍼티가 활성화됩니다.

두 옵션을 전환할 때 타겟 플랫폼과 애플리케이션에 따라 성능에 차이가 생길 수 있습니다. 하지만 대부분의 애플리케이션에서는 성능 차이가 크지 않으며, Depth Normals를 사용하면 대부분 시각적으로 더 나은 결과를 얻을 수 있습니다.

- **Quality > Source > Normal Quality:** **Source** 프로퍼티를 **Depth**로 설정했을 때 활성화됩니다.
 - 이 프로퍼티의 옵션(Low, Medium, High)에 따라 Unity가 뎁스 텍스처에서 노멀 벡터를 재구성할 때 사용하는 뎁스 텍스처의 샘플 수가 정해집니다. 품질 옵션별 샘플 수는 다음과 같습니다.
 - Low: 1
 - Medium: 5
 - High: 9

성능 영향은 중간 정도로 간주됩니다.

- **Quality > Downsample:** Downsample을 선택하면 X와 Y 방향 양측에서 처리 해상도가 절반으로 줄어듭니다. 이 경우 처리할 픽셀 수의 75%를 효과적으로 줄일 수 있으므로 GPU 부하를 크게 줄일 수 있지만, 효과의 디테일 또한 줄어듭니다.
- **Quality > After Opaque:** After Opaque 옵션은 최종 렌더링 형태에 영향을 미치며, 그에 따라 성능도 달라집니다.
 - **비활성화된 경우:** SSAO에 Depth 또는 Depth Normals 프리패스가 적용됩니다(아래 Source 옵션 참조). SSAO는 해당 프리패스 이후에 계산된 다음, 조명을 계산할 때 DrawOpakes 패스에 적용됩니다. 그러면 더 근사한 모습의 앰비언트 오클루전을 구현할 수 있으며, 사용자가 SSAO의 Direct Lighting Strength 값을 제어할 수 있습니다. 하지만 성능에는 부정적인 영향을 미칩니다.
 - **활성화된 경우:** After Opaque를 선택한 경우 SSAO에 Depth Normals가 필요합니다. Depth를 선택한 경우 Depth 프리패스가 있다면 Depth 프리패스에서 뎁스를 가져오거나, 불투명 렌더링 후 CopyDepth 패스를 가져옵니다. 그러면 DrawOpakes 패스 이후 SSAO가 조명

계산의 일부가 되지 않고 모든 요소의 위에 추가됩니다. 이 방법은 프리패스를 건너뛸 수 있으므로 성능이 개선될 수 있다는 장점이 있습니다.

참고: Render Opaque 패스에서 덤스와 노멀을 렌더링할 수도 있으므로 이 옵션을 활성화한 상태에서 프리패스를 완전히 건너뛰어 리소스를 절약할 수 있습니다.

- **Quality > Blur Quality:** High, Medium 또는 Low로 설정할 수 있습니다. 리소스가 제한적인 기기에서는 만족스러운 결과가 보장되는 선에서 가장 낮은 설정을 사용하세요.
- **Quality > Samples:** SSAO 렌더러 기능은 모든 픽셀마다 지정된 반지를 내의 샘플 수를 사용하여 앰비언트 오클루전 값을 계산합니다. 이 값을 Low에서 Medium, High로 올리면 더 매끄럽고 디테일한 효과를 구현할 수 있지만, 성능은 저하됩니다.



Direct Lighting Strength에 서로 다른 값을 사용한 예시, 각각 0.2(왼쪽), 0.9(오른쪽). 오른쪽 이미지에서는 계단 사이의 선이 더 진한 것을 볼 수 있습니다.

SSAO는 URP의 유연성을 확인할 수 있는 좋은 예시입니다. 상상력만 있다면 렌더러 기능으로 그 어떤 문제도 해결할 수 있습니다.

더 많은 자료

- UGuruz의 YouTube [튜토리얼](#) (영문)
- 앰비언트 오클루전 [기술 자료](#)
- 이 레시피에서 사용한 [에셋](#) (마르셀로 바리오 제공)

데칼



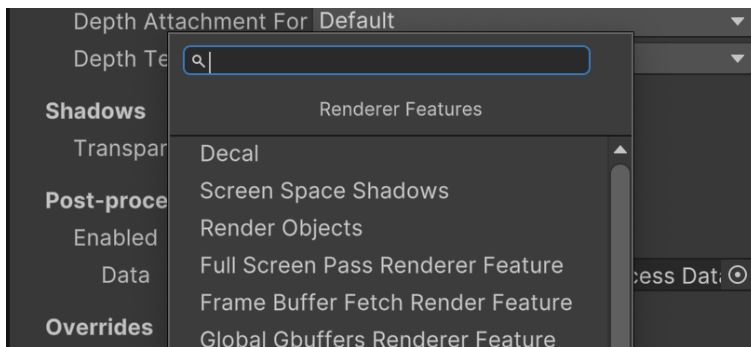
데칼의 다양한 활용 사례 중 하나는 Splashteam에서 Unity로 제작한 게임 [Tinykin](#)의 캐릭터 마일로에 적용된 것처럼 3D 표면에 블룸 그림자를 투사하는 것입니다.

데칼을 사용하면 효과적으로 표면에 오버레이를 추가할 수 있습니다. 데칼은 플레이어가 씬과 상호 작용하며 생기는 총알 구멍이나 타이어 자국과 같은 시각적 요소를 게임 환경에 추가하는 데 주로 사용됩니다. 다음 이미지의 계단에서 볼 수 있듯이 데칼은 메시를 감싸는 형태로 표현됩니다. 이 레시피에서 사용하는 씬 파일과 에셋은 **Scenes > Decals** 폴더에서 확인할 수 있습니다.



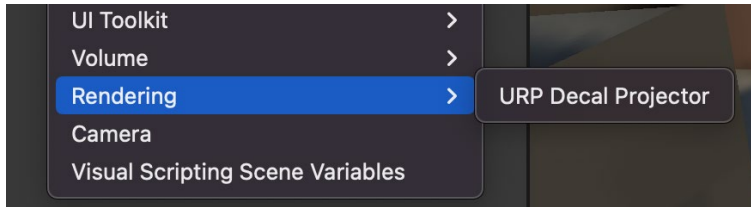
간단한 씬에 데칼을 추가한 모습

데칼은 렌더러 기능을 사용해 씬에 렌더링됩니다. 레시피 폴더에서 **Decals_URP_Settings_Renderer**를 보면 데칼 렌더러 기능이 추가되어 있는 것을 확인할 수 있습니다. 기존에 하던 것처럼 Main Camera에 **AutoLoadPipelineAsset.cs** 스크립트를 연결하면 씬을 로드할 때 올바른 파이프라인 에셋이 사용됩니다. 커스텀 씬에 데칼을 추가하려면 Player에서 현재 렌더링에 사용하는 Universal Renderer Data 에셋을 선택하고 인스펙터의 Add Renderer Feature 드롭다운에서 **Decal**을 선택합니다.



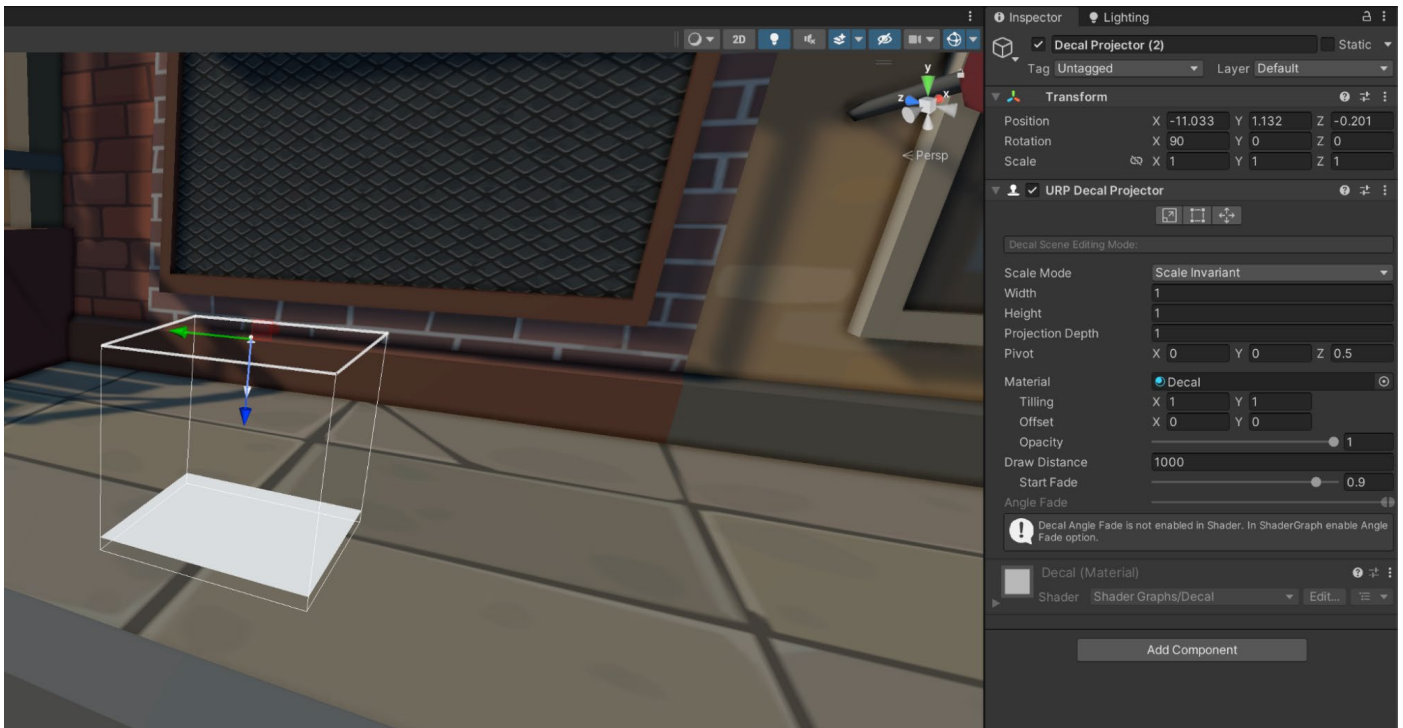
Add Renderer 드롭다운의 Decal 옵션

에디터에서 씬에 데칼을 추가하려면 계층(Hierarchy) 창을 오른쪽 클릭하고 **Rendering > URP Decal Projector**를 선택합니다.



URP Decal Projector 생성

기존과 같이 에디터에서 URP 데칼 프로젝터를 배치하고 방향을 지정합니다. 데칼 프로젝터는 직교 투사를 사용하므로 표면에 캐스팅되는 데칼의 크기는 프로젝터와 표면의 거리에 영향을 받지 않습니다. 처음에는 새 데칼 프로젝터가 흰색 블록으로 표시됩니다. 축 화살표와 함께 투사 방향을 나타내는 흰색 화살표가 표시됩니다.



새 데칼 프로젝터의 모습

URP Decal Projection 프로퍼티

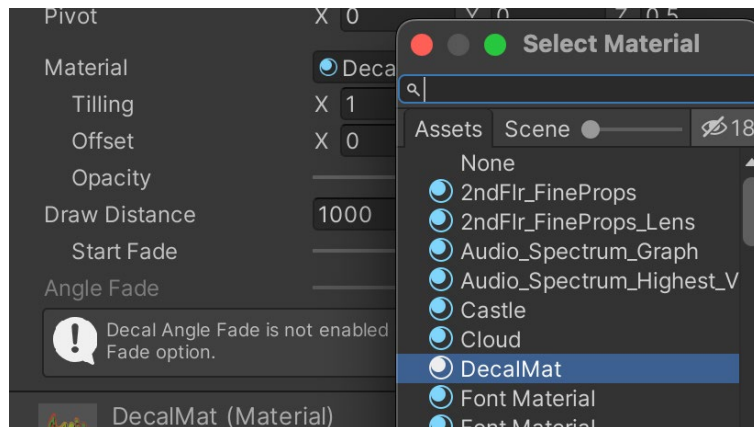
- **Scale Mode:** 기본적으로 URP Decal Projection 컴포넌트에서는 **Scale Mode**가 **Scale Invariant**로 설정되어 있습니다. 다시 말해서 데칼의 크기는 **Width** 프로퍼티와 **Height** 프로퍼티에 의해서만 정해집니다. **Inherit from Hierarchy**로 전환하면 게임 오브젝트의 Transform Scale 프로퍼티와 Width 및 Height 프로퍼티가 함께 사용됩니다.

- **Width 및 Height:** 데칼의 크기를 제어하는 프로퍼티입니다.
- **Projection Depth:** 프로젝터 바운딩 박스의 깊이를 설정하면 프로젝터가 로컬 Z축을 따라 데칼을 투사합니다.
- **Pivot:** 루트 게임 오브젝트의 원점을 기준으로 프로젝터 바운딩 박스의 중앙 오프셋 위치를 설정합니다.
- **Material:** 투사할 머티리얼을 설정합니다. 이때 머티리얼은 반드시 [Shader Graph/Decal](#)을 사용하는 머티리얼이어야 합니다(자세한 내용은 아래에서 설명).
- **Tiling 및 Offset:** UV 축을 따라 데칼 머티리얼의 타일링과 오프셋 값을 설정할 수 있습니다.
- **Opacity:** 불투명도 값을 지정할 수 있습니다. 값이 0이면 데칼이 완전히 투명해지고, 1이면 Material 프로퍼티에서 정의한 값에 따라 불투명하게 표시됩니다.
- **Draw Distance:** 카메라에서 데칼까지의 거리입니다. 프로젝터가 데칼의 투사를 멈추고 URP가 데칼을 더는 렌더링하지 않는 지점을 나타냅니다.
- **Start Fade:** 슬라이더를 사용해서 카메라부터 프로젝터가 데칼을 페이드 아웃하기 시작하는 지점까지의 거리를 설정합니다. 0에서 1까지의 값은 Draw Distance의 비율을 나타내며, 값을 0.9로 설정하면 Unity는 데칼의 페이드 아웃을 Draw Distance의 90%에서 시작해 Draw Distance에서 끝냅니다.
- **Angle Fade:** 데칼의 뒤쪽 방향과 데칼이 드로우되는 표면의 버텍스 노멀 사이의 각도를 기준으로 데칼의 페이드 아웃 범위를 설정합니다.

머티리얼 생성

데칼 프로젝터는 Shader Graph/Decal 셰이더를 사용하는 머티리얼을 사용해야 합니다. 이 예시에서는 Scene 폴더에 있는 DecalMat 머티리얼을 사용합니다. 베이스 맵은 할당되어 있지만 노멀 맵은 할당되어 있지 않은 머티리얼로, 데칼을 울퉁불퉁한 표면에 드로우할 때 유용합니다.

머티리얼은 인스펙터에서 프로젝터에 할당합니다.



URP 데칼 프로젝터 머티리얼 할당

코드로 데칼 추가

URP 데칼 프로젝트는 에디터에서 개발하면서 씬에 추가할 수도 있지만, 런타임에 사용자 상호 작용의 결과로 추가하는 경우가 더 일반적입니다. 프리팹을 생성해 Material, Width, Height 프로퍼티를 설정해도 되지만, 런타임에 코드를 사용해 쉽게 업데이트할 수도 있습니다. 이 예제 코드에서는 인스턴스화, 위치 지정, 방향을 집중적으로 다루고 있습니다. 콜라이더를 클릭하여 데칼을 추가하는 전체 코드는 레시피 폴더의 [AddDecal.cs](#) 스크립트에서 확인할 수 있습니다.

```
void AddDecalProjector(Vector3 pos, Vector3 normal)
{
    GameObject decalProjectorObject = Instantiate(decalProjectorPrefab);

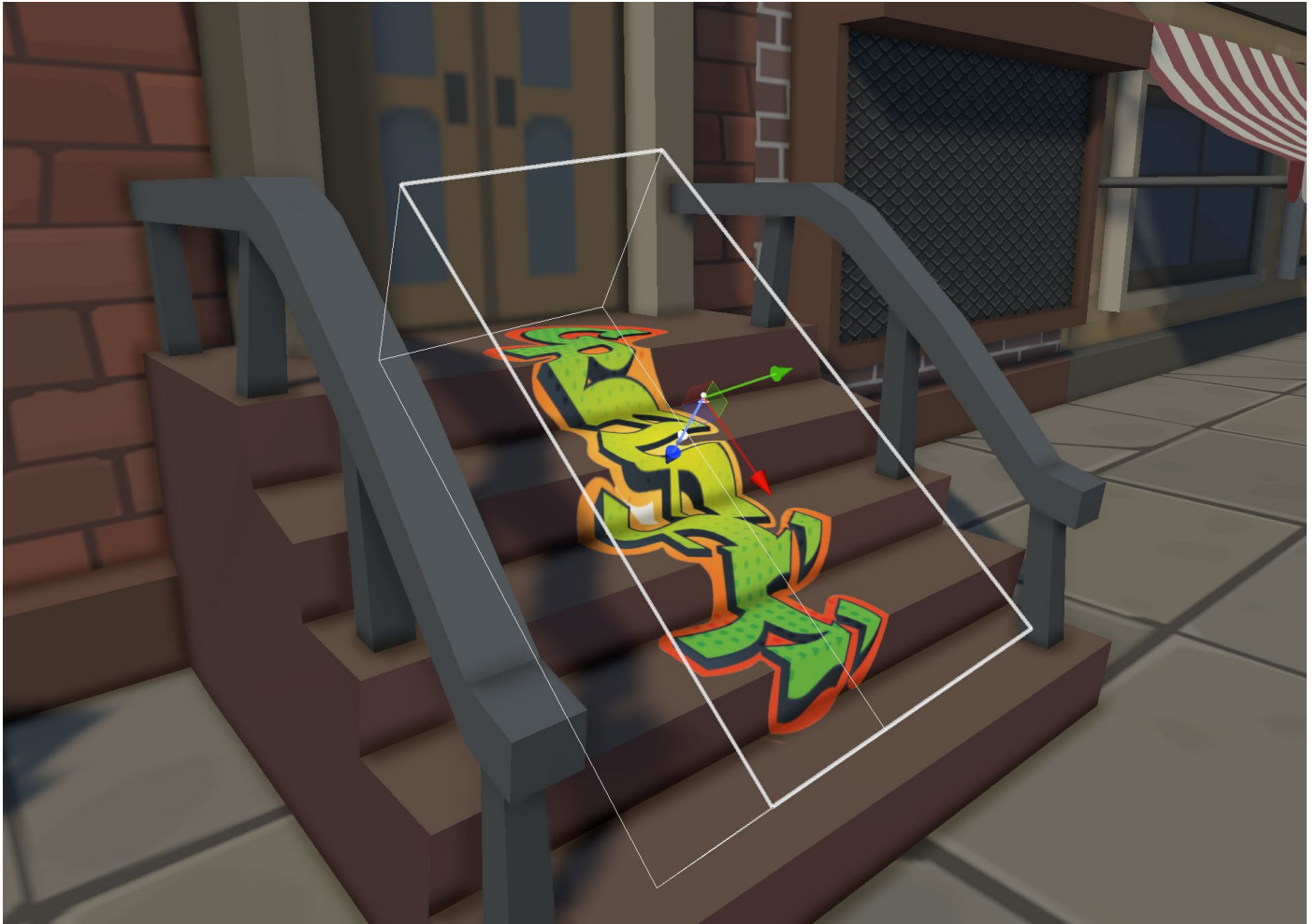
    //머티리얼에 대해 데칼을 개별적으로 제어하려면
    //DecalProjector에 새 머티리얼 인스턴스를 생성합니다.
    //DecalProjector decalProjectorComponent = decalProjectorObject.
    GetComponent<DecalProjector>();
    //decalProjectorComponent.material = new Material(decalProjectorComponent.material);

    //표면에서 멀어짐
    pos += normal * 0.5f;

    Quaternion up = Quaternion.AngleAxis(Random.Range(0, 360), Vector3.left);
    Quaternion rot = Quaternion.LookRotation(-normal, up.eulerAngles);
    decalProjectorObject.transform.SetPositionAndRotation(pos, rot);
}
```

이 함수는 콜라이더 위에 마우스다운 이벤트가 발생한 후 RaycastHit이 있을 때 호출됩니다. 여기서 pos 는 hit.point이고 normal은 hit.normal입니다. decalProjectorObject라는 프리팹이 인스턴스화됩니다. 위치를 구하려면 Projection Depth를 초과하지 않는 선에서 pos Vector3를 표면으로부터 옮겨야 하며, 노멀을 따라 지점을 옮기면 이 작업을 수행할 수 있습니다. 데칼의 방향을 지정하려면 먼저 무작위화된 상향 벡터를 생성합니다. 데칼을 표면에 정렬하기 위해 필요한 회전을 구하고 노멀을 중심으로 임의의 양을 회전하려면 파라미터 역노멀과 무작위화된 상향 벡터를 사용합니다.

데칼은 게임에서 다양한 용도로 사용되며, URP 데칼 프로젝터는 데칼 작업에 유용한 툴입니다.



씬(Scene) 뷰에서 본 데칼

더 많은 자료

- 데칼 렌더러 [기술 자료](#)
- YouTube [튜토리얼](#) - Llam Academy(영문)

물

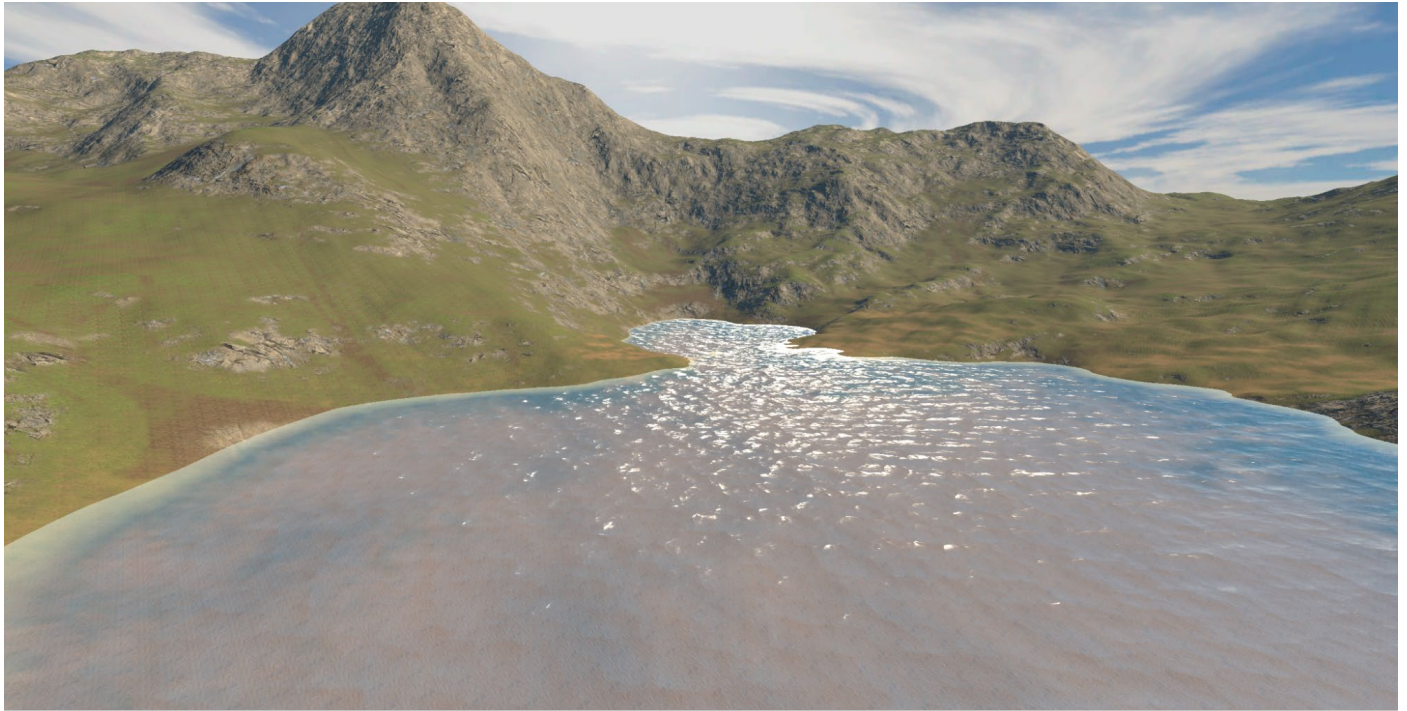


물과 수생 식물은 비디오 게임에서 아름답고 개방적인 느낌의 환경을 만드는 데 중요한 두 가지 시각적 요소입니다. 이 이미지는 Flowstudio에서 Unity로 제작한 생존 게임 [Len's island](#)에서 가져왔습니다.

간단한 물 셰이더를 만드는 레시피입니다. 아티스트와 디자이너가 더 쉽게 따라 할 수 있도록 Shader Graph로 제작되었습니다.

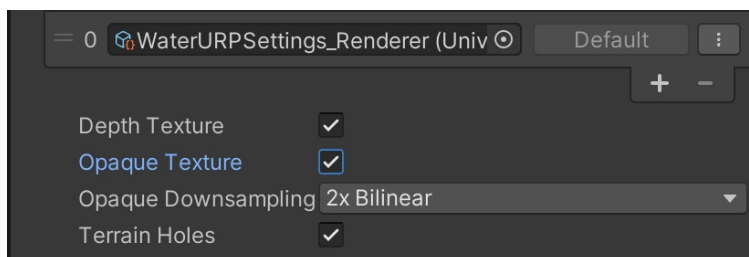
물 셰이더는 다음 세 단계로 구성됩니다.

- 물 컬러 생성
- 타일화된 노멀 맵을 움직여 표면에 잔물결 추가
- 버텍스 위치에 움직이는 변위를 추가해 너울 효과 생성



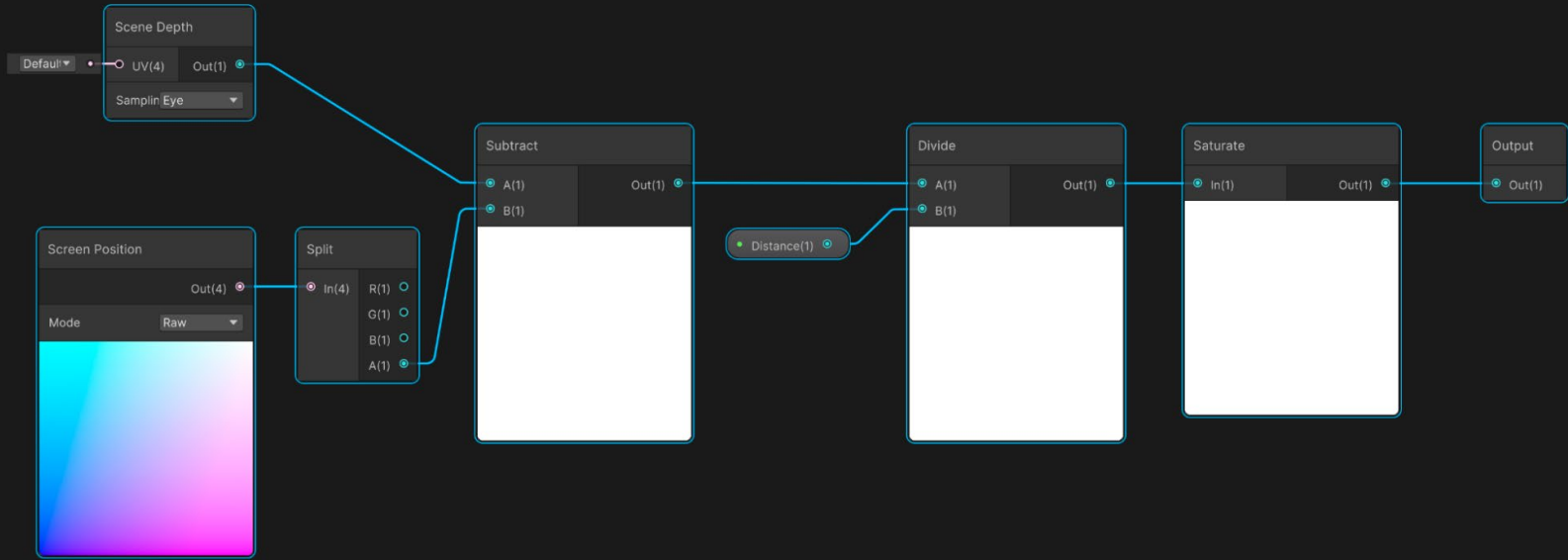
간단한 물 셰이더가 적용된 모습을 보여주는 동영상의 스틸 이미지

최종 결과물은 **Scenes > Water** 폴더에서 Water 씬을 열어 확인할 수 있습니다. 최종 셰이더는 두 개의 하위 그래프, DepthFade와 TextureMovement를 사용합니다. 따라서 물 셰이더를 살펴보기 전에 두 하위 그래프를 먼저 살펴보는 것이 좋습니다. Water 씬은 **Depth Texture**와 **Opaque Texture** 옵션이 활성화된 **WaterURPSettings** 에셋을 사용합니다. Opaque Texture는 굴절 등 이 레시피에서 다루지 않는 효과를 추가하려는 경우에만 필요합니다.



WaterURPSettings 에셋에서 Depth Texture와 Opaque Texture가 선택되어 있는 모습

DepthFade 하위 그래프



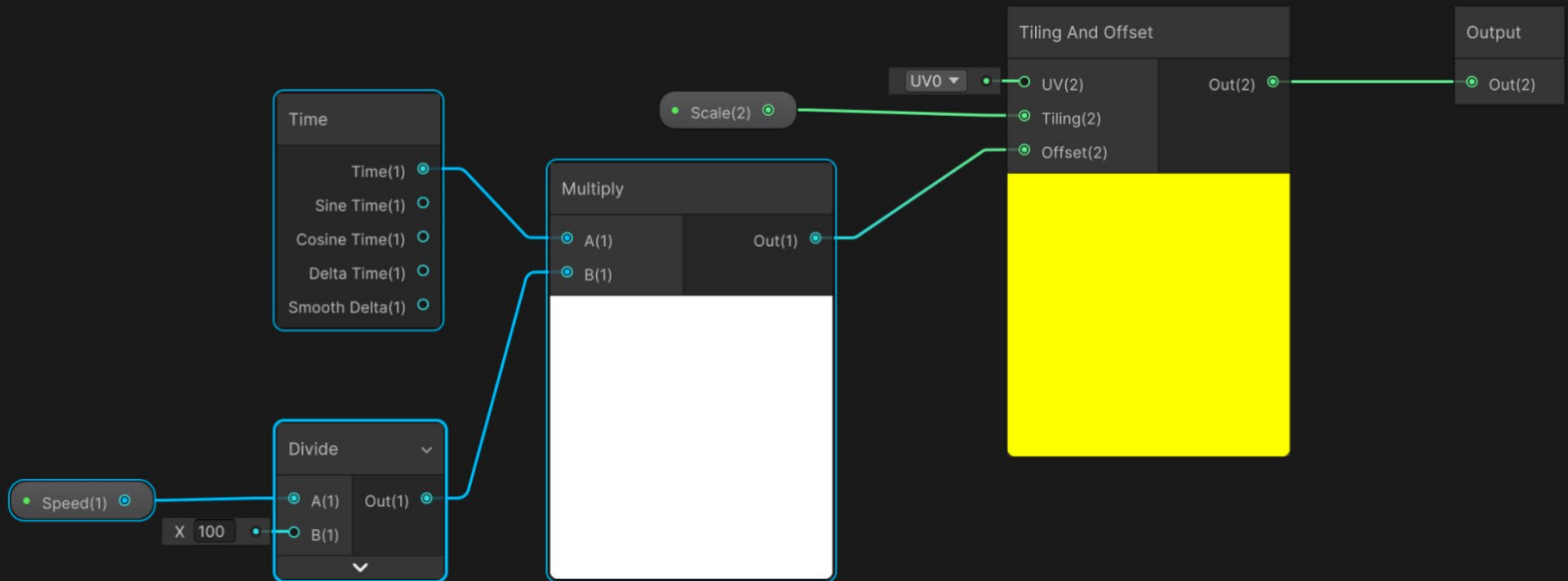
DepthFade 하위 그래프

수심이 얇은 곳과 깊은 곳은 서로 다른 고유의 컬러로 표현해야 합니다. 물의 최종 컬러는 **Depth** 프로퍼티에 따라 이 두 컬러가 블렌딩된 컬러입니다. Depth는 수면과 그 아래 지오메트리 사이의 거리입니다. 물 셰이더가 투명하게 설정되어 있으므로 불투명 지오메트리는 이미 렌더링되어 있고, URP Settings 에셋에 Depth Texture가 선택되어 있으므로 현재 텍스처를 구할 수 있습니다.

Sampling이 Eye로 설정된 **Scene Depth** 노드로 눈에서 현재 픽셀의 불투명 지오메트리까지의 거리를 확인할 수 있습니다. 출력 값 Mode가 Raw로 설정된 **Screen Position** 노드는 현재 물 픽셀의 렌더링에 관한 정보를 가집니다. Split 노드를 사용하는 이유는 눈에서 현재 물 픽셀까지의 거리를 저장하는 W 컴포넌트가 필요하기 때문입니다.

바로 아래를 향하는 광선이 아닌 눈 위치에서 나오는 광선이지만, 기존 불투명 지오메트리까지의 거리에서 물의 거리를 빼면 수심을 확인할 수 있습니다. **Divide** 노드는 수심이 얇은 곳과 깊은 곳 사이에서 에지가 나타나는 위치를 제어합니다. 이 하위 그래프의 출력은 0에서 1 사이의 값을 가져야 합니다. 따라서 항상 출력을 0에서 1 사이로 제한해 특수한 **Clamp** 노드 역할을 하는 **Saturate** 노드를 사용해야 합니다.

TextureMovement 하위 그래프

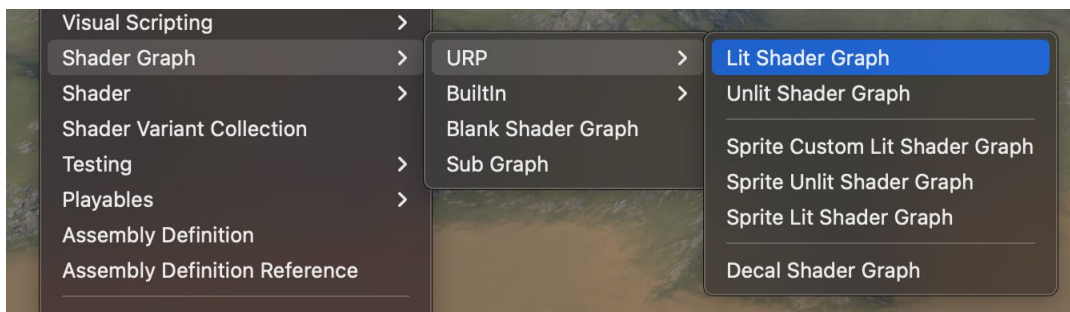


TextureMovement 하위 그래프

물 셰이더에는 TextureMovement 하위 그래프를 사용해 처리되는 여러 개의 움직이는 텍스처가 있습니다. 이 하위 그래프에서 **Time** 노드는 **Multiply** 노드에 대한 입력 중 하나로 사용됩니다. Speed 입력을 100으로 나눈 값은 Multiply 노드의 두 번째 입력이 됩니다. Multiply 노드의 출력은 **Tiling and Offset** 노드의 Offset 입력이 되고, **Scale** 프로퍼티는 Tiling 입력이 됩니다. 시간이 지남에 따라 이 간단한 하위 그래프에 Speed와 Scale 값이 입력되면 **Sample Texture2D** 노드에 사용되는 UV가 업데이트됩니다.

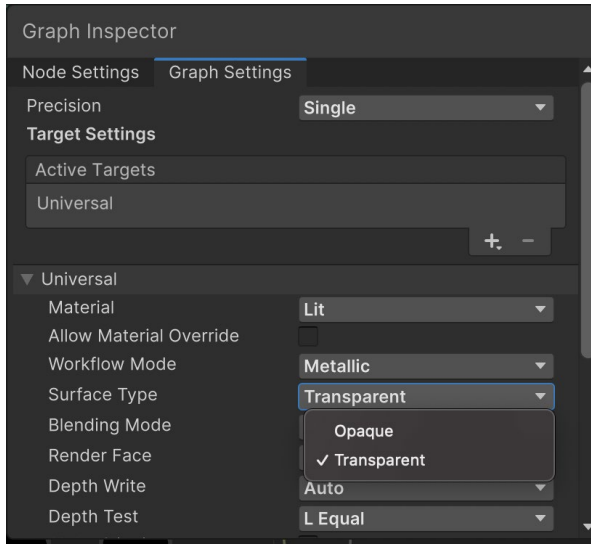
물 셰이더

이제 URP > Lit Shader Graph를 통해 Lit Shader Graph를 기반으로 물 셰이더를 생성할 차례입니다.



Create > Shader Graph > URP > Lit Shader Graph

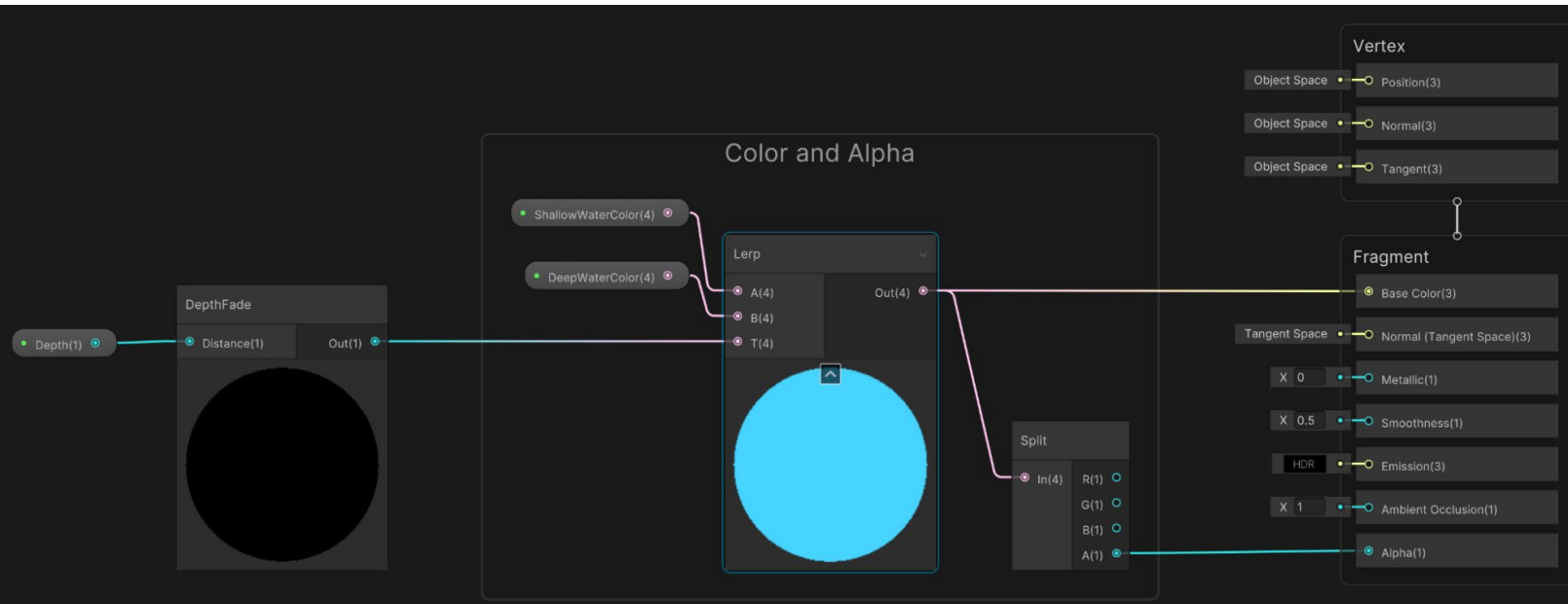
Graph Inspector를 사용해 **Surface Type**을 설정합니다.



Surface Type 설정

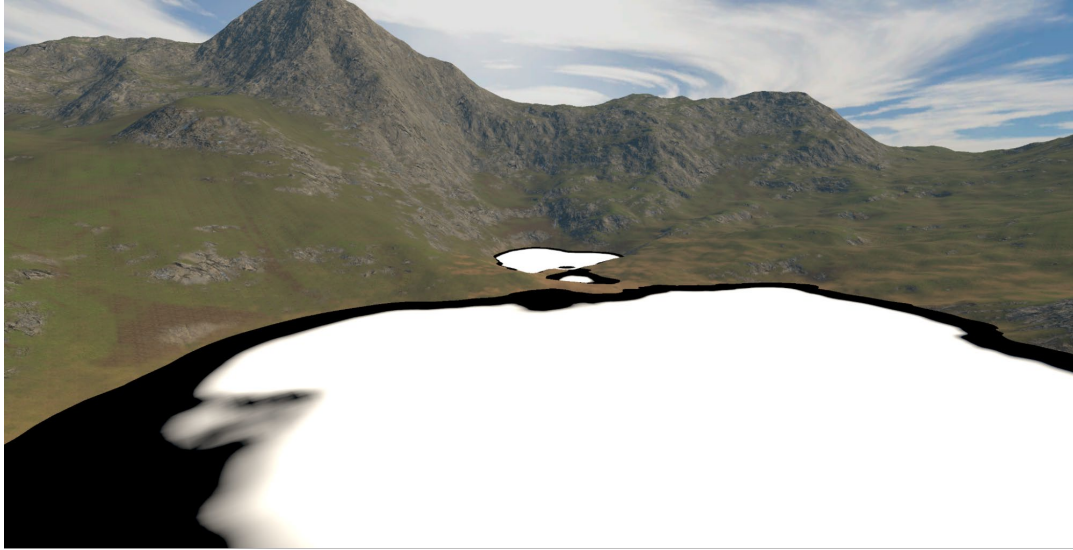
컬러부터 시작해서 그래프를 편집해 보겠습니다.

컬러



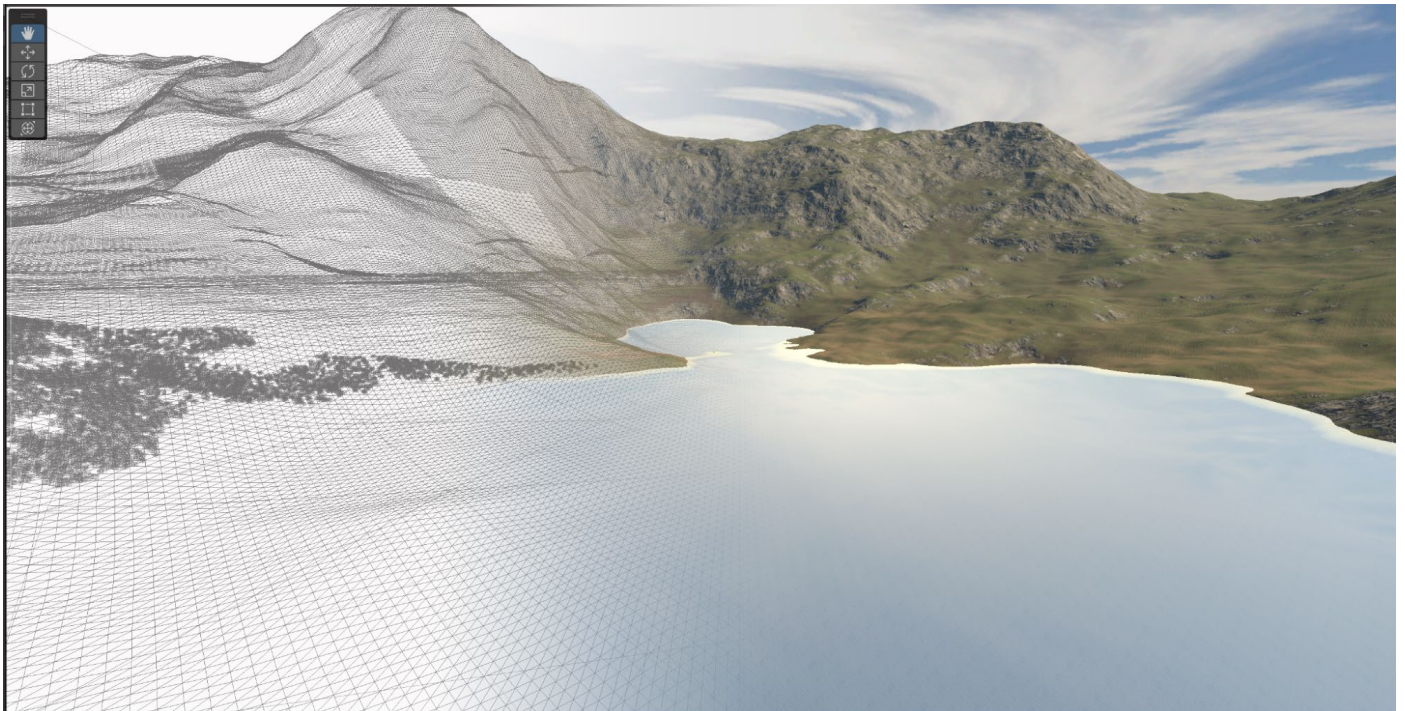
Color and Alpha 노드

컬러는 DepthFade 하위 그래프를 추가해 제어합니다. 하위 그래프는 제어를 위해 플로트 **Depth** 프로퍼티를 사용합니다. 출력이 프래그먼트 셰이더의 Base Color 입력으로 바로 연결되면, 다음 이미지처럼 수심이 얇은 곳은 검은색, 깊은 곳은 흰색으로 표시됩니다. Depth 값이 높을수록 검은색이 더 넓게 퍼집니다. 검은색은 0을, 흰색은 1을 나타냅니다.



DepthFade의 출력을 Fragment > Base Color에 바로 연결한 모습

DepthFade를 Base Color 입력에 직접 연결하는 대신 [Lerp](#) 노드로 연결합니다. **ShallowWaterColor**는 A로 입력되어 검은색을 대체하고, **DeepWaterColor**는 B로 입력되어 흰색을 대체합니다. 이러한 컬러의 알파를 설정할 때는 얇은 물이 더 투명하게 보이도록 해야 합니다. Lerp의 출력은 **Fragment > Base Color**로 연결됩니다. Alpha의 경우, Split 노드를 사용하고 A 출력을 **Fragment > Alpha**에 연결합니다. 그러면 다음과 같은 이미지가 생성됩니다.

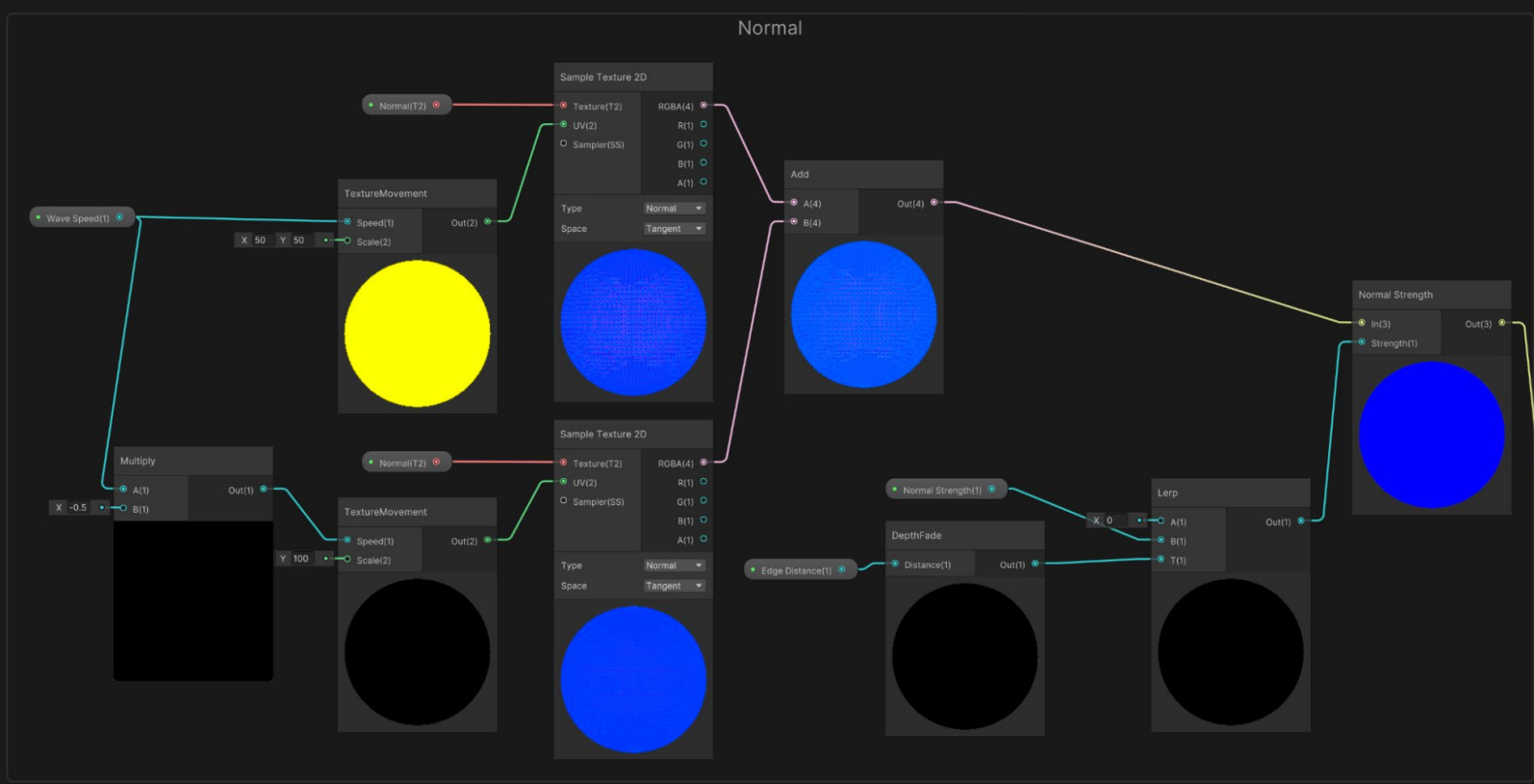


고밀도 메시와 컬러가 적용된 물

실제 물은 평면이지만, 위 이미지에 보이는 대로 메시에 더 많은 버텍스가 있어 버텍스 변위를 사용할 수 있습니다.

간단한 평면으로 시작했는데, 아직 노멀 맵 같은 더 많은 작업을 해야 합니다.

노멀 맵

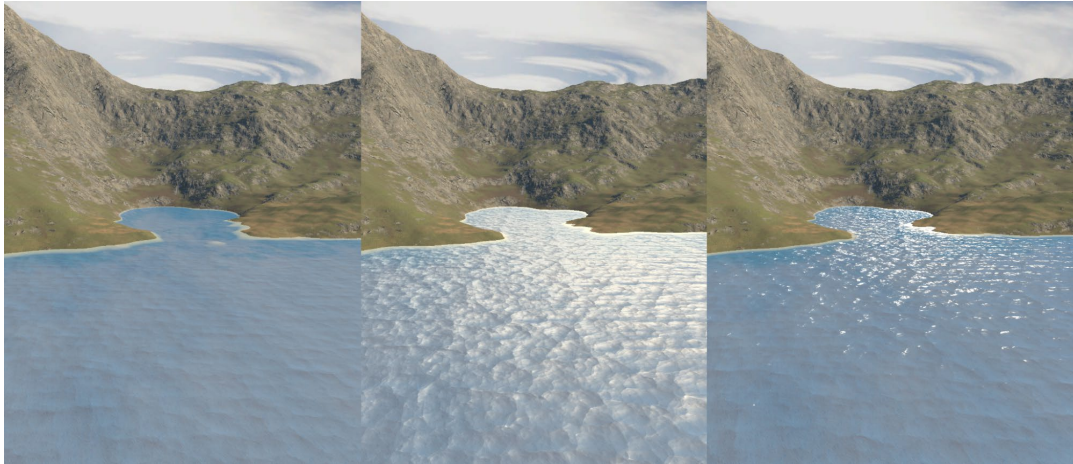


Fragment > Normal 제어

노멀 맵은 표면에 움직이는 잔물결을 추가합니다. 첫 번째 입력 프로퍼티는 **Wave Speed**로, TextureMovement 하위 그래프에서 Speed 입력으로 사용됩니다. **Scale**은 50,50의 고정된 값으로 설정하고, **Speed**는 Multiply 노드를 통해 전처리되어 두 번째 TextureMovement 노드에서 Wave Speed 프로퍼티의 절반을 뺀 값으로 설정됩니다.

노멀을 계산하기 위한 다음 단계는 두 개의 TextureMovement 하위 그래프 노드에서 처리한 UV를 사용해 노멀 텍스처를 두 번 샘플링하는 것입니다. 두 개의 노멀을 합쳐서 움직이는 두 텍스처를 결합한 효과를 구현합니다. 셰이더에는 **Normal Strength** 플롯 프로퍼티가 있으며, 이는 Normal Strength 노드의 Strength 입력으로 사용할 수 있습니다. 하지만 물가에 가까이 갈수록 잔물결이 사라지도록 구현하려 합니다. DepthFade 하위 그래프 노드와 **Edge Distance** 셰이더 프로퍼티를 사용해서 퍼짐을 제어해 이 효과를 구현할 수 있습니다. 이 값은 Lerp 노드의 T 입력으로 사용되어 0과 Normal Strength 사이를 블렌딩합니다. 그래프에서 이 단계의 출력은 Fragment > Normal로 연결됩니다.

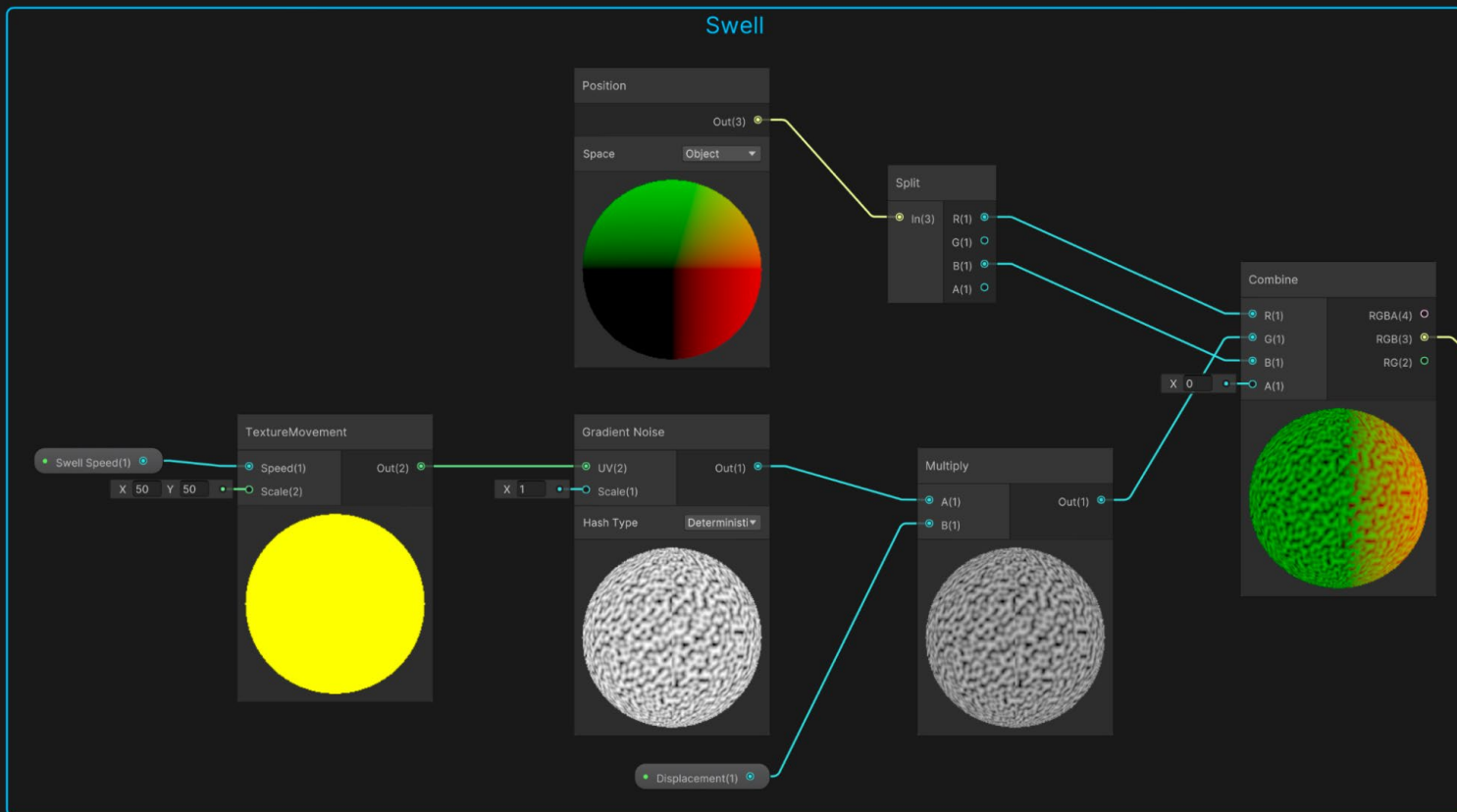
이제 제어할 수 있는 잔물결을 만들었으며, 간단한 플로트 프로퍼티를 사용해 Fragment의 **Smoothness**를 제어하여 반사 프로퍼티를 조정할 수 있습니다. 다음 이미지는 Smoothness 값을 변경했을 때 보이는 효과입니다.



다양한 Smoothness 값을 적용한 잔물결, 왼쪽에서 오른쪽으로 각각 0, 0.5, 1을 지정한 결과

다음 단계에서는 벡텍스 변위를 활성화해 물에 모션을 추가합니다.

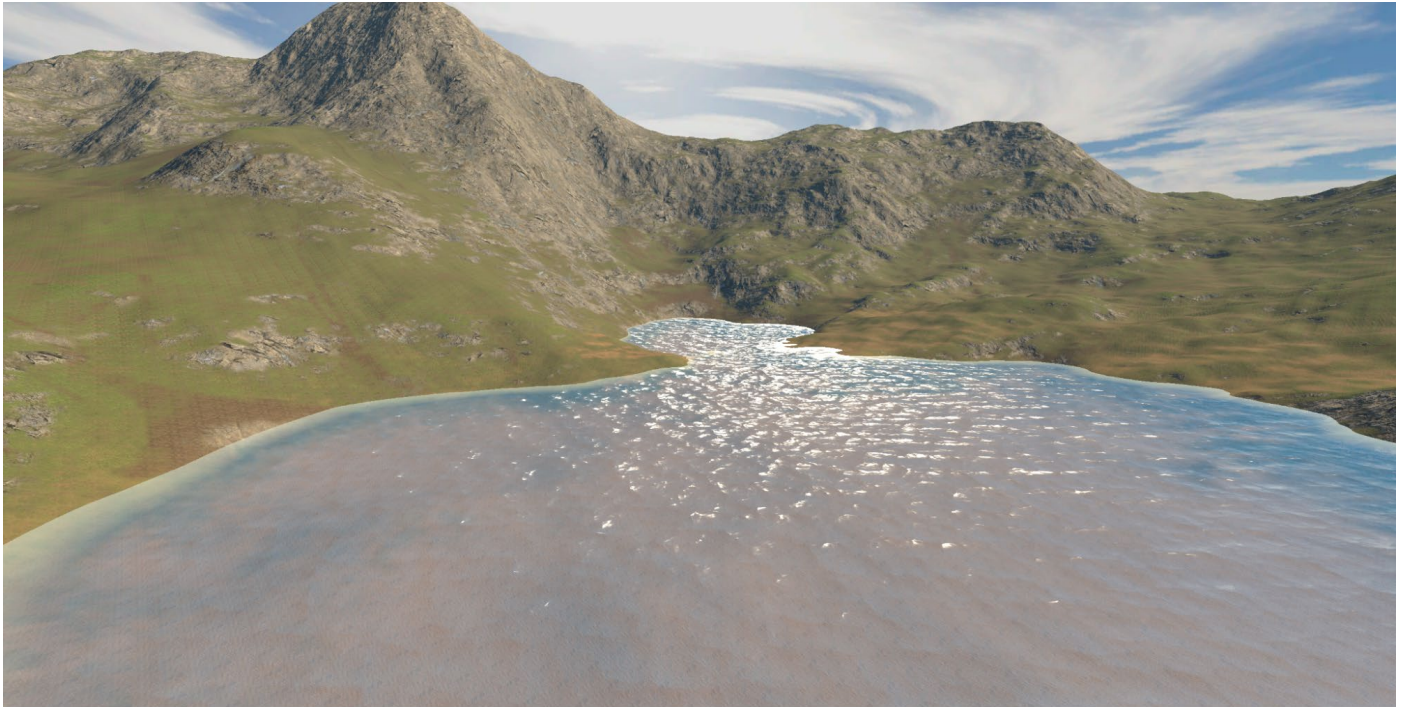
너울



Gradient Noise 노드를 사용한 너울 제어

이 단계에서는 다시 TextureMovement 하위 그래프 노드를 사용합니다. Speed는 셰이더 플롯 프로퍼티인 **Swell Speed**를 사용하여 설정하고, Scale은 50,50의 고정된 값으로 설정합니다. 이는 Scale 값이 1로 고정된 **Gradient Noise** 노드의 UV 입력 역할을 합니다. Multiply 노드에서 셰이더 플롯 프로퍼티 **Displacement**를 사용해 이 값을 제어합니다. 이 노드의 목적은 오브젝트 공간에서 버텍스에 대한 Y 값을 설정하는 것입니다. Position 노드의 **Space** 파라미터가 Object로 설정되어 있는데, 이 값은 Split 노드로 연결된 다음 Combine 노드로 연결됩니다. Combine은 Split 노드에서 직접 R 값과 B 값을 받으며 이때 R은 Position X, B는 Position Z입니다. Y에 대한 G 값은 Gradient Noise 경로에서 가져옵니다. RGB(3) 출력은 Vertex > Position으로 연결됩니다.

플레이 모드에서 씬을 확인해 보면 특히 물가에서 너울이 움직이는 모습을 볼 수 있습니다.



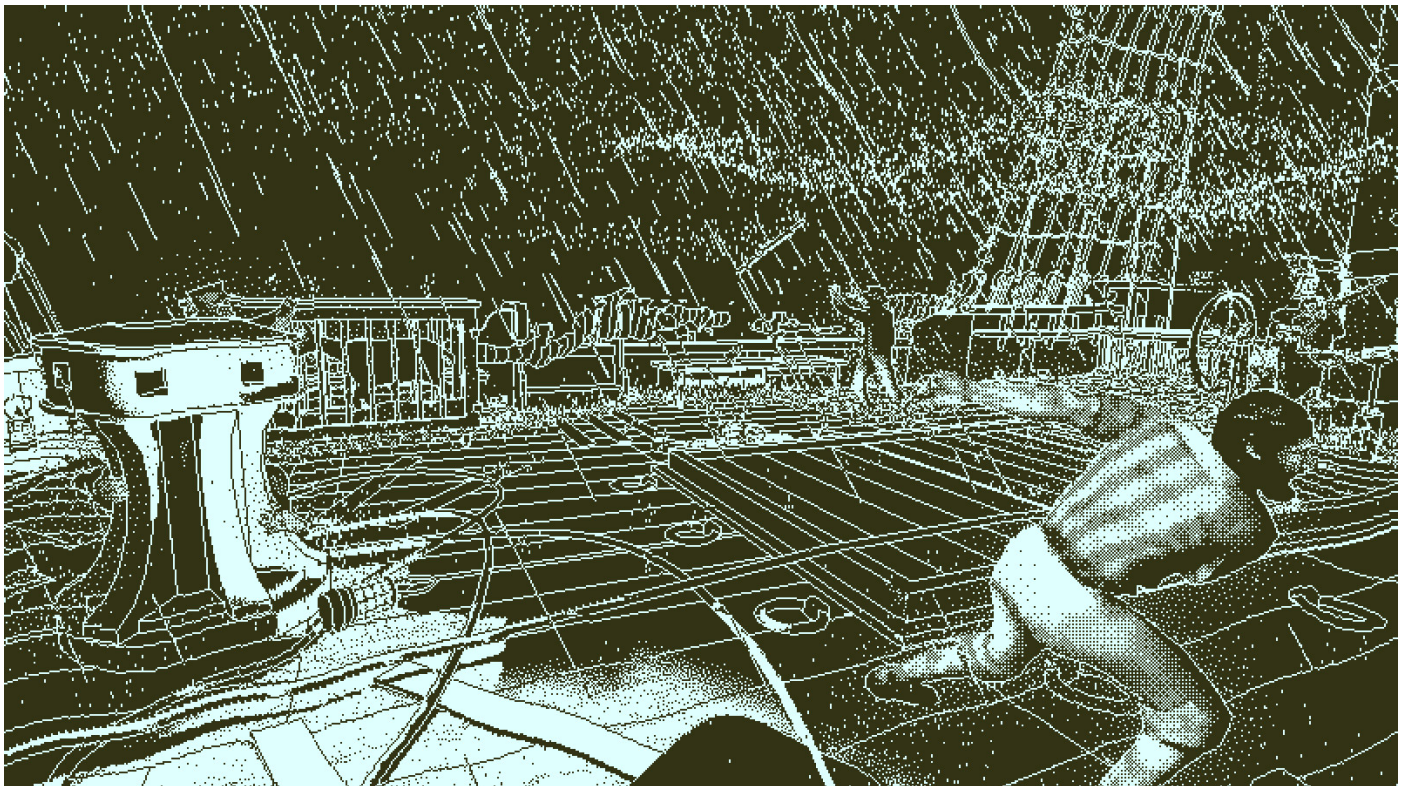
최종 결과물

이 레시피를 통해 간단한 물 셰이더의 기초를 구성할 수 있지만, 커스틱 반사와 굴절, 포말 등을 사용해 효과를 한층 더 개선할 수 있습니다. 추가 가이드는 아래 링크를 참조하세요.

더 많은 정보

- Unity [YouTube](#) 튜토리얼
- [커스틱 반사](#) 튜토리얼(영문) - 앨런 주코니
- [스타일라이즈드 물 효과](#) 튜토리얼(영문) - Binary Lunar

컬러 그레이딩용 LUT

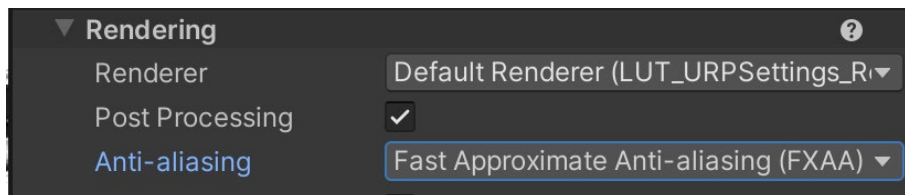


루카스 포프가 Unity를 사용해 제작한 미스터리 어드벤처 FPS 게임 [Return of the Obra Dinn](#)은 이 레시피를 참고한 로파이 아트 스타일과 컬러 팔레트로 독특한 디자인(look and feel)을 선보입니다.



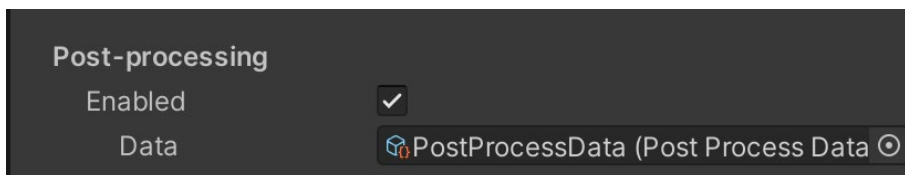
Color Lookup을 사용한 그레이딩 효과 구현

URP에서 제공하는 포스트 프로세싱 필터를 아직 사용해 보지 않았다면, 이 레시피가 많은 도움이 될 것입니다. 이 레시피에서 사용하는 필터는 하나이나, 사용된 단계는 모든 필터에 적용할 수 있습니다. 새 URP 씬은 기본적으로 포스트 프로세싱이 비활성화되어 있으므로 **Camera > Rendering** 패널에서 포스트 프로세싱을 활성화해야 합니다.



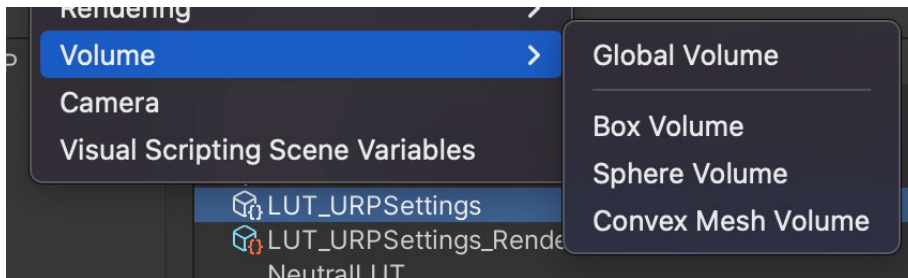
Camera > Rendering에서 Post Processing 선택

또한 Universal Renderer Data 에셋에서 포스트 프로세싱을 활성화해야 합니다.



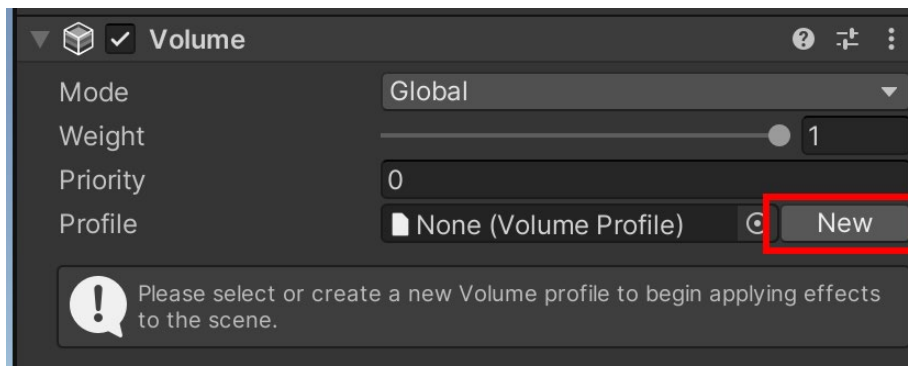
Universal Renderer Data 에셋에서 포스트 프로세싱 선택

카메라 위치에 필터를 적용하려면 Global Volume을 추가합니다. 계층 창을 오른쪽 클릭하고 **Volume > Global Volume**을 선택합니다.



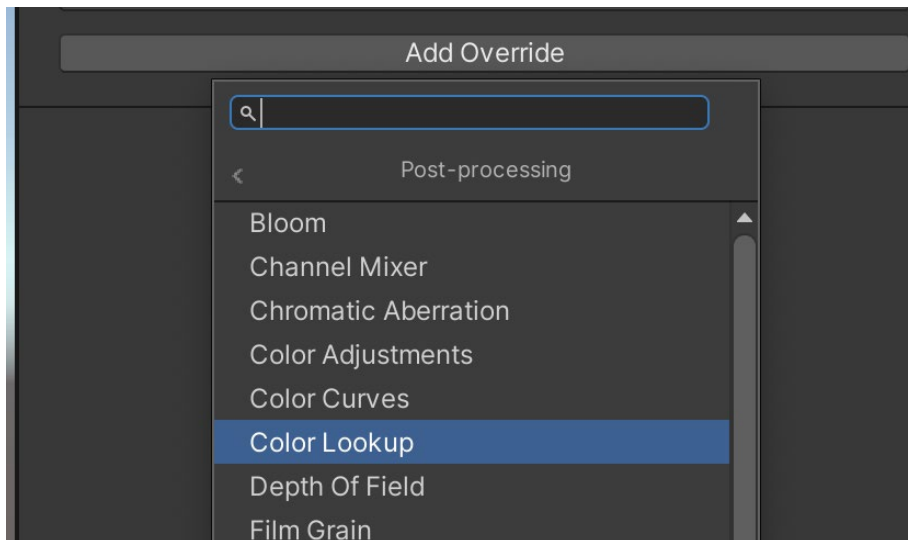
Global Volume 생성

새 게임 오브젝트를 선택하고 **New**를 클릭해 새 프로파일을 생성합니다.



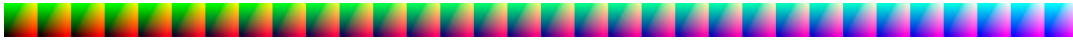
새 프로파일 생성

이제 오버라이드를 추가할 수 있습니다. **Add Override** 버튼을 누르고 Post-processing에서 **Color Lookup**을 선택합니다.



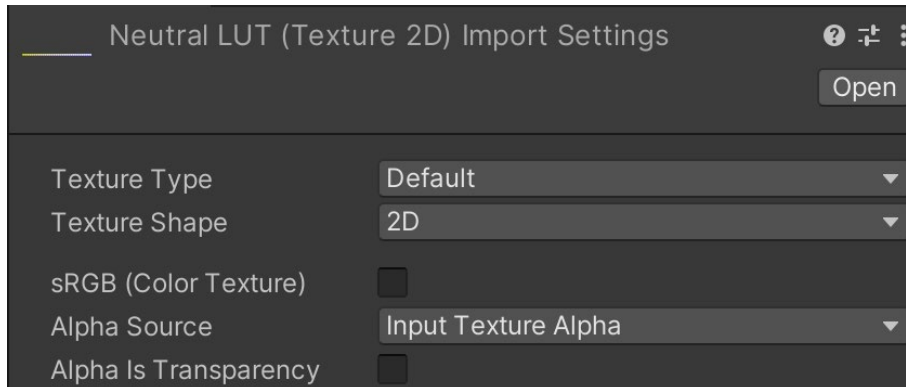
Color Lookup 포스트 프로세싱 필터 추가

All 버튼을 클릭합니다. 이제 LUT(룩업 테이블) 이미지 텍스처가 필요합니다. LUT 이미지 텍스처는 필터에서 기본 렌더링되는 컬러를 변경하기 위해 사용할 스트립 이미지입니다. 이미지 파일은 **Scenes > LUT > NeutralLUT.png**에서 확인하거나 [이 링크](#)에서 다운로드할 수 있습니다.



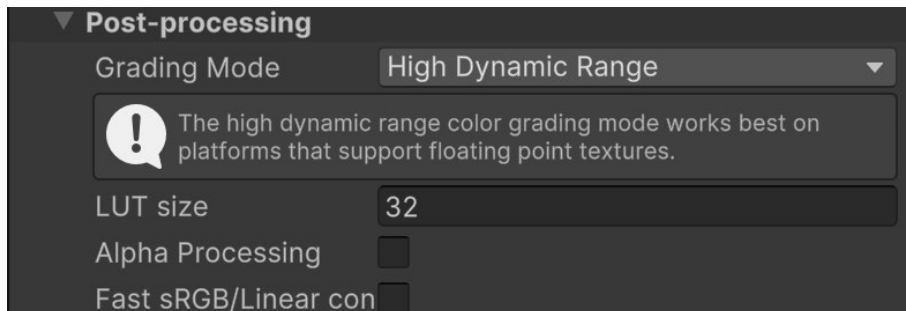
NeutralLUT.png

LUT 이미지에서 **sRGB (Color Texture)**를 비활성화해야 하므로 이미지를 선택해 인스펙터에서 비활성화합니다.



모든 LUT 텍스처에 대해 sRGB (Color Texture) 비활성화

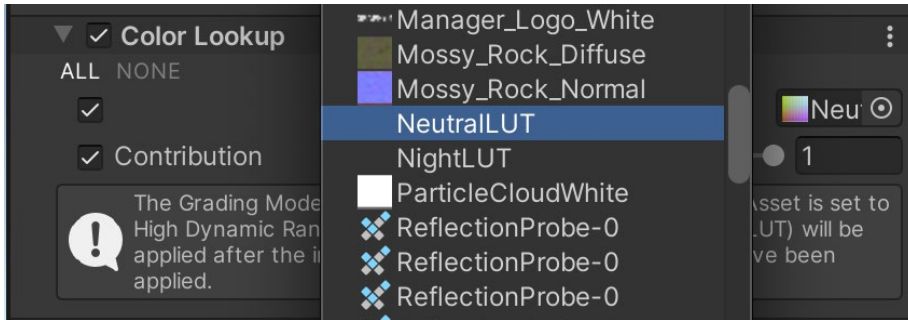
위 NeutralLUT 이미지에서 블록의 수를 세어 보면 32개임을 알 수 있습니다. 16개의 블록을 사용해도 되는데, 32개와 16개 중 무엇을 사용하든 블록의 수는 URP 에셋 설정과 일치해야 합니다. 32개의 블록을 사용하는 경우 포스트 프로세싱 패널에서 **LUT size**를 **32**로 설정해야 합니다. **Grading Mode** 옵션을 자유롭게 실험해 보세요.



LUT size 설정

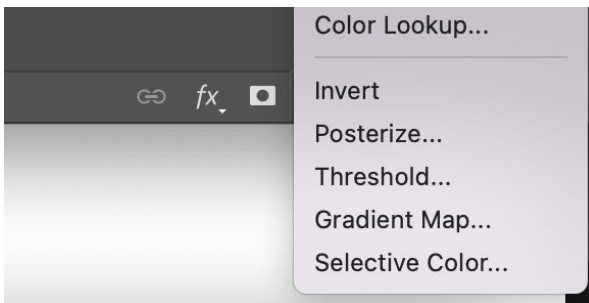
Color Lookup 설정 패널에서 **NeutralLUT.png**를 Lookup Texture로 할당하면 렌더링된 이미지가 변경되지 않습니다. 해당 필터는 텍스처를 사용해 새로운 컬러를 설정하며, 코드는 현재 픽셀 컬러를 사용해 LUT 이미지에서 텍셀을 찾습니다. 중립 LUT 이미지에서 텍셀 컬러는 현재 픽셀 컬러와 동일합니다.

Lookup Texture로 사용하는 이미지를 Photoshop이나 Krita 같은 페인트 프로그램으로 처리할 때 실질적인 효과가 나타납니다. 이 섹션 마지막 부분인 더 많은 자료에서 컬러 그레이딩에 Krita를 사용하는 방법을 설명하는 YouTube 동영상 링크를 클릭해 보세요.



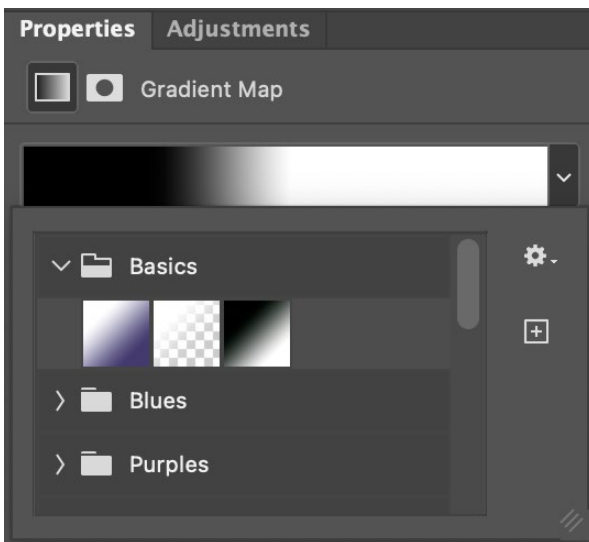
Lookup Texture 할당

씬의 스크린샷을 캡처해서 Photoshop에서 엽니다. **Layers** 패널 하단에서 절반은 검은색, 절반은 흰색인 원형 버튼을 찾습니다. 버튼을 클릭하고 패널에서 **Gradient Map**을 찾습니다. 여기서 새로운 컬러 조정 레이어를 추가할 수 있습니다.



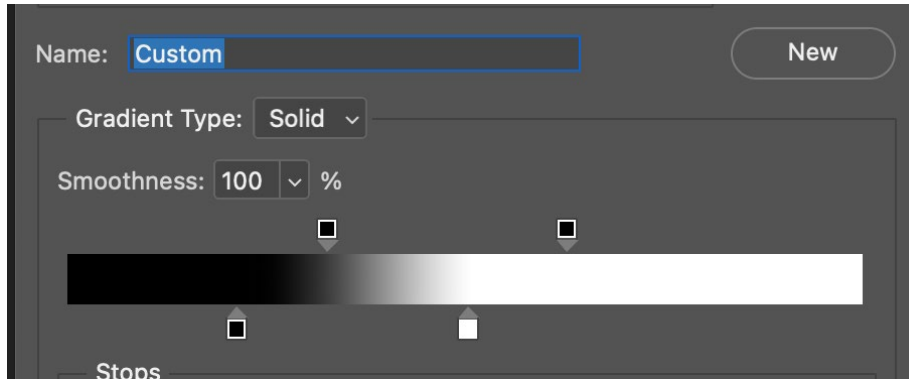
컬러 조정 레이어 생성

콘트라스트가 높은 흑백 이미지를 만드는 컬러 조정 레이어를 생성하려면 Gradient Map 드롭다운을 클릭하고 Basics를 선택한 다음 흑백 조합을 선택합니다.



흑백 그레디언트 선택

콘트라스트를 높이려면 그레디언트를 클릭해서 새 창을 열고, 정지점을 사용해 콘트라스트를 조정합니다.



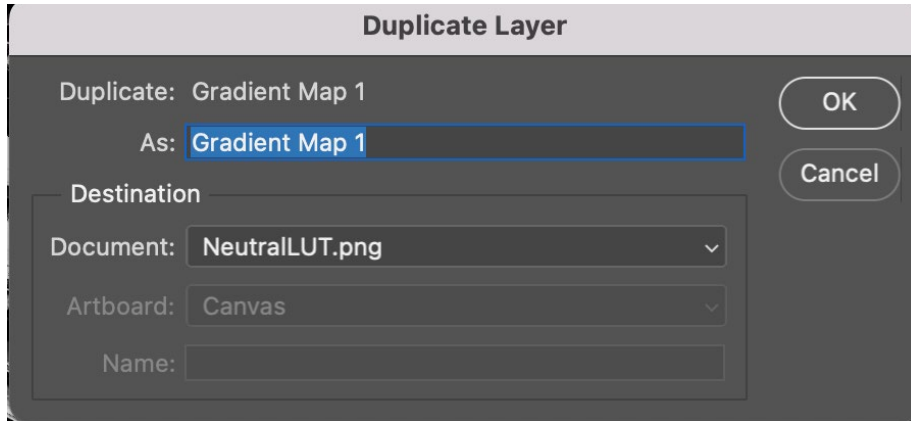
정지점을 변경해 콘트라스트 높이기

이제 스크린샷이 흑백으로 보입니다.



Gradient Map 효과

그레이딩을 선택했으면 이 레이어를 NeutralLUT.png 파일에 적용해야 합니다. 파일을 Photoshop에서 열고, 이전의 스크린샷에서 조정 레이어를 오른쪽 클릭하고 **Duplicate Layer**를 선택합니다. 새 패널에서 **Destination > Document**로 NeutralLUT.png를 선택합니다.



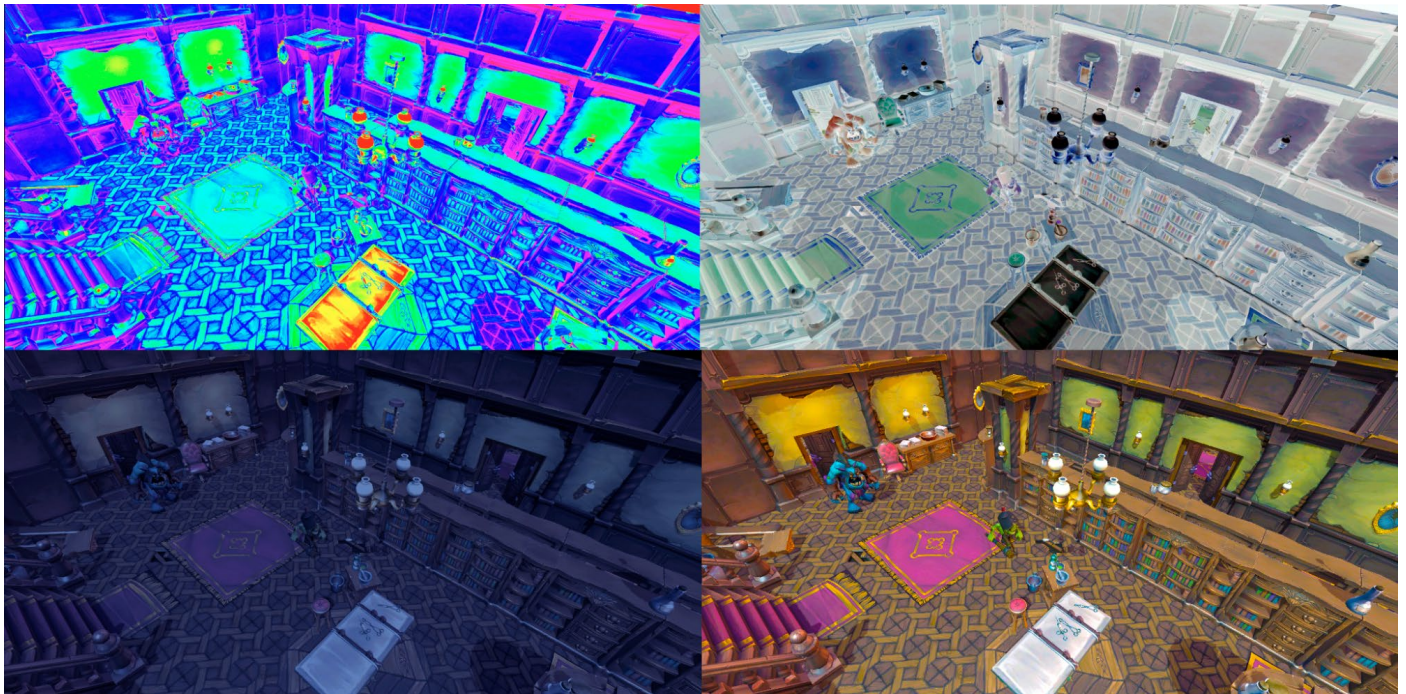
조정 레이어 복제

이제 텍스처가 다음 이미지처럼 보입니다.



B&WLUT.png

이미지를 저장하고 프로젝트의 Assets 폴더로 드래그합니다. 인스펙터 패널을 사용해 sRGB (Color Texture) 를 비활성해야 합니다. 마지막으로 새 LUT 텍스처를 Color Lookup 필터의 Lookup Texture로 할당합니다.



다양한 LUT 텍스처 활용

LUT 텍스처를 사용하면 극적인 컬러 그레이딩을 효율적으로 만들 수 있으며, 이 방식은 다양한 게임에서 유용합니다.



더 많은 리소스

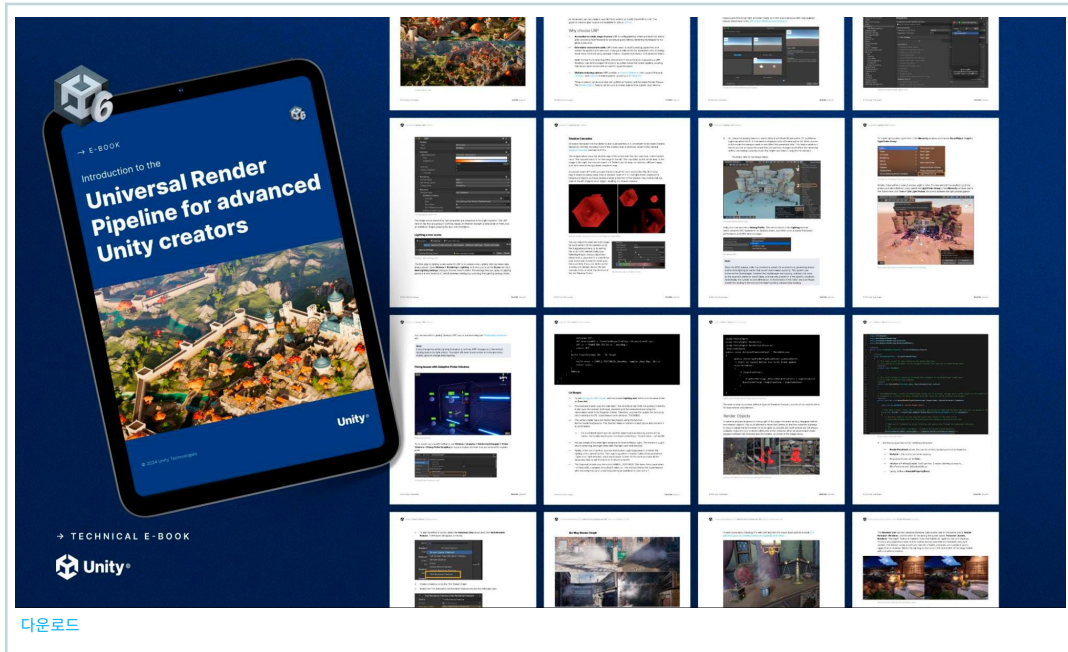
- URP의 포스트 프로세싱에 관한 [기술 자료](#)
- YouTube [튜토리얼](#) (영문) - PHLEARN
- YouTube [튜토리얼](#) (영문) - GDQuest

적응적 프로브 볼륨



Unity와 URP를 사용해 제작한 Clockstone의 [LEGO® Bricktales](#)는 플레이어를 LEGO 월드로 빠져들게 하는 게임으로, 뛰어난 조명이 블록의 분위기와 사실감을 연출하는 데 커다란 역할을 합니다.

APV(적응적 프로브 볼륨)는 혼합 모드 조명을 위한 최신 Unity 솔루션으로, 라이트 프로브에 비해 쉽게 설정하고 관리할 수 있습니다. URP의 조명 기법을 처음 접한다면 Unity 전자책 [Unity 고급 사용자를 위한 유니버설 렌더 파이프라인](#)을 확인해 보세요.



조명에 혼합 모드 설정을 사용하여 베이킹된 오브젝트와 동적 오브젝트를 조합할 수 있습니다. 혼합 모드를 사용한다면 씬에 프로브도 추가하는 것이 좋습니다. Unity 6에는 라이트 프로브와 APV라는 두 개의 옵션이 있습니다. 두 옵션은 동적 오브젝트가 씬에서 움직이더라도 전역 조명에 영향을 받을 수 있도록 하여 동일한 문제를 해결합니다. 하지만 APV는 기존 라이트 프로브에 비해 이점이 많고 효율적으로 설정할 수 있습니다.

프로브는 씬에 배치하는 한 개의 점입니다. 디자인하는 과정에서 이 위치의 전역 조명이 계산됩니다. 런타임에 프레임을 렌더링할 때, 조명 계산이 포함된 URP 셰이더가 가장 가까운 프로브를 조합하여 전역 조명 값을 계산합니다.

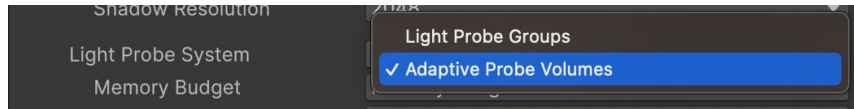
참고:

GI(전역 조명)는 직접광에서 표면에 직접 도달하는 빛에만 국한되지 않고 빛이 표면에서 다른 표면으로 반사되는 방식을 모델링하여 간접광을 생성하는 시스템입니다.

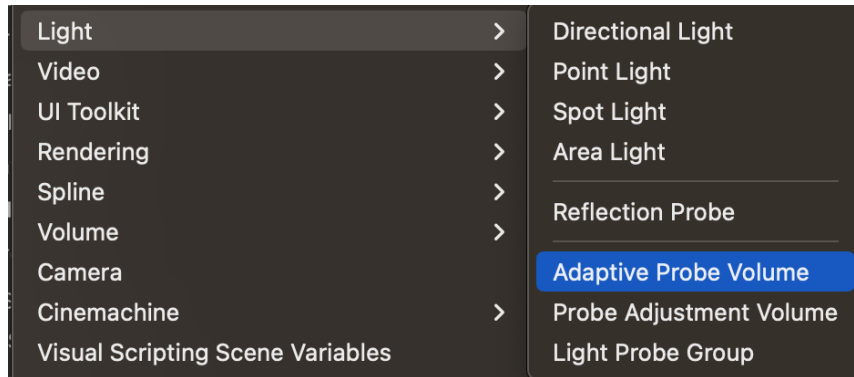
씬에서 APV 사용

씬에서 라이트 프로브의 위치를 신중하게 지정했으나 씬 레이아웃이 변경되어 있는 상황을 겪어 본 테크니컬 아티스트라면, 많은 씬에서 모든 프로브를 몇 초 이내에 배치할 수 있는 APV의 장점을 즉시 체감할 것입니다. [FPS 샘플: 더 인스펙션](#)(FPS Sample: The Inspection)을 예로 들겠습니다.

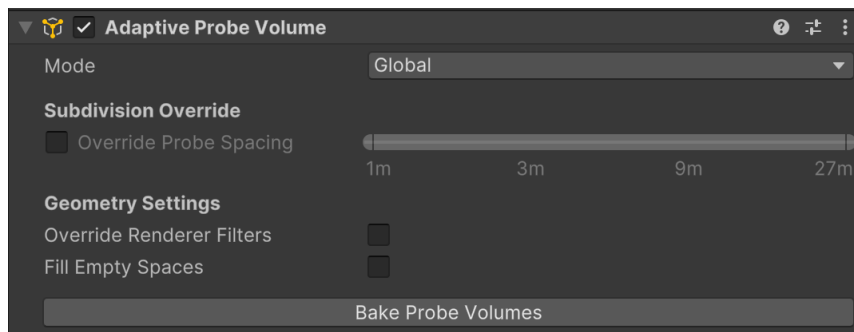
1. 먼저 활성 URP 에셋의 **Light Probe System** 옵션이 **Adaptive Probe Volumes**로 설정되어 있는지 확인합니다.



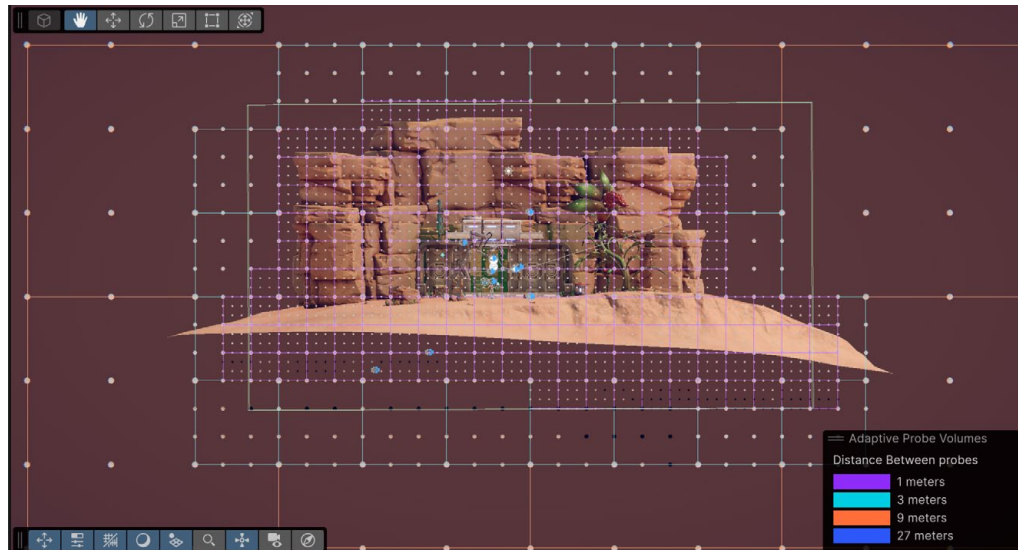
2. 계층 창을 오른쪽 클릭하고 **GameObject > Light > Adaptive Probe Volume**(APV)을 선택합니다.



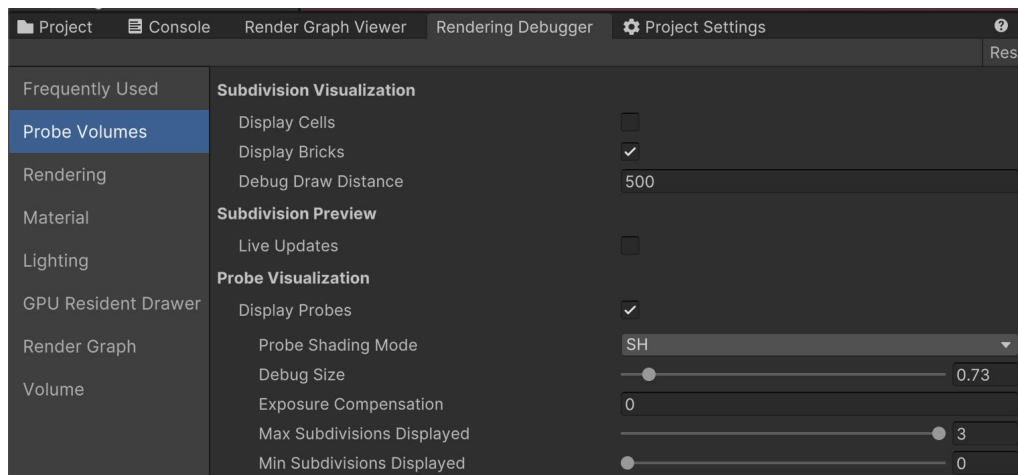
3. Mode를 **Global**로 설정하고 1m, 3m, 9m, 27m로 이뤄진 Subdivision의 기본 설정을 확인합니다.



4. **Bake Probe Volumes**를 눌러 볼륨을 베이킹합니다. 현재 씬을 스캔하고 씬 내 지오메트리에 따라 프로브가 배치됩니다. 지오메트리가 많은 영역일수록 프로브가 조밀하게 배치됩니다.



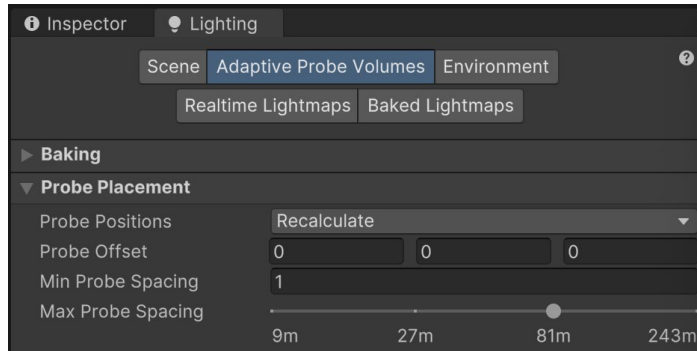
5. 베이킹 결과를 확인하려면 **Analysis > Rendering Debugger**를 엽니다. **Probe Volumes**를 선택하고 **Display Probes**를 선택합니다. 다른 해상도를 보려면 **Display Bricks**를 선택합니다.



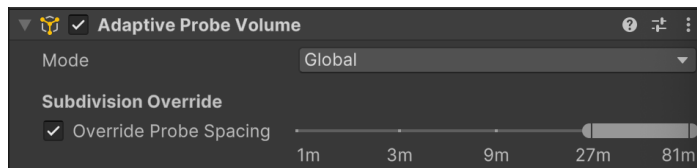
대부분의 씬에서는 이렇게 하면 작업이 완료되므로 마음 놓고 휴식을 취할 수 있습니다. 하지만 APV로 더욱 정확도를 높일 수도 있습니다. 다양한 세분화 수준을 지닌 여러 볼륨을 추가하여 프로브의 배치와 밀도를 세밀하게 제어할 수 있습니다.

URP 3D 샘플의 오아시스 환경을 예시로 보겠습니다. 씬에서 대부분의 동작이 텐트 주변에서 이루어지므로 텐트 주변에 프로브를 가장 많이 배치하려고 합니다. 이를 구현하는 방법은 다음과 같습니다.

1. **Rendering > Lighting > Adaptive Probe Volumes**를 열고 **Max Probe Spacing**을 81m로 변경합니다.



2. 적응적 프로브 볼륨을 추가하여 **Global**로 설정하고 **Override Probe Spacing**을 27m>81m로 설정합니다.

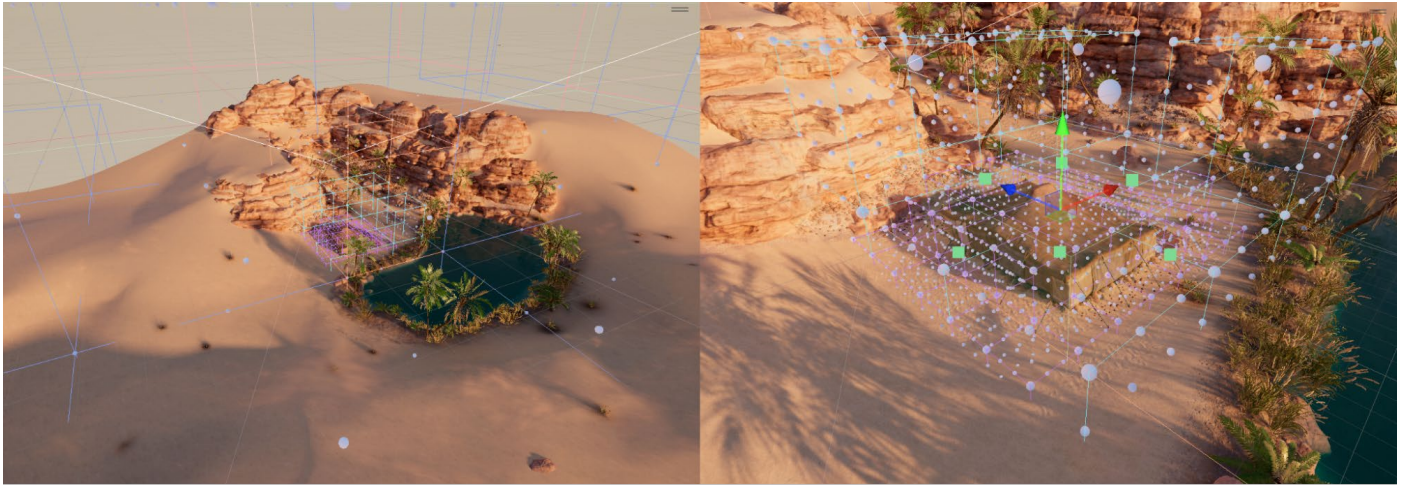


3. 적응적 프로브 볼륨을 추가하여 **Local**로 설정하고 **Override Probe Spacing**을 1m>9m로 설정합니다. 볼륨을 텐트보다 약간 더 크게 설정합니다.



4. 프로브 볼륨을 베이킹합니다.

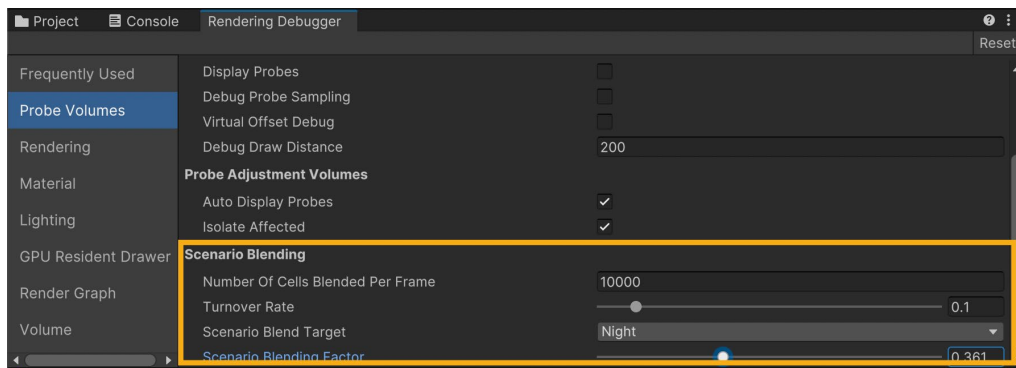
아래 이미지에서 보이는 대로 대부분의 프로브가 텐트 주변에 배치됩니다.



프로브 배치

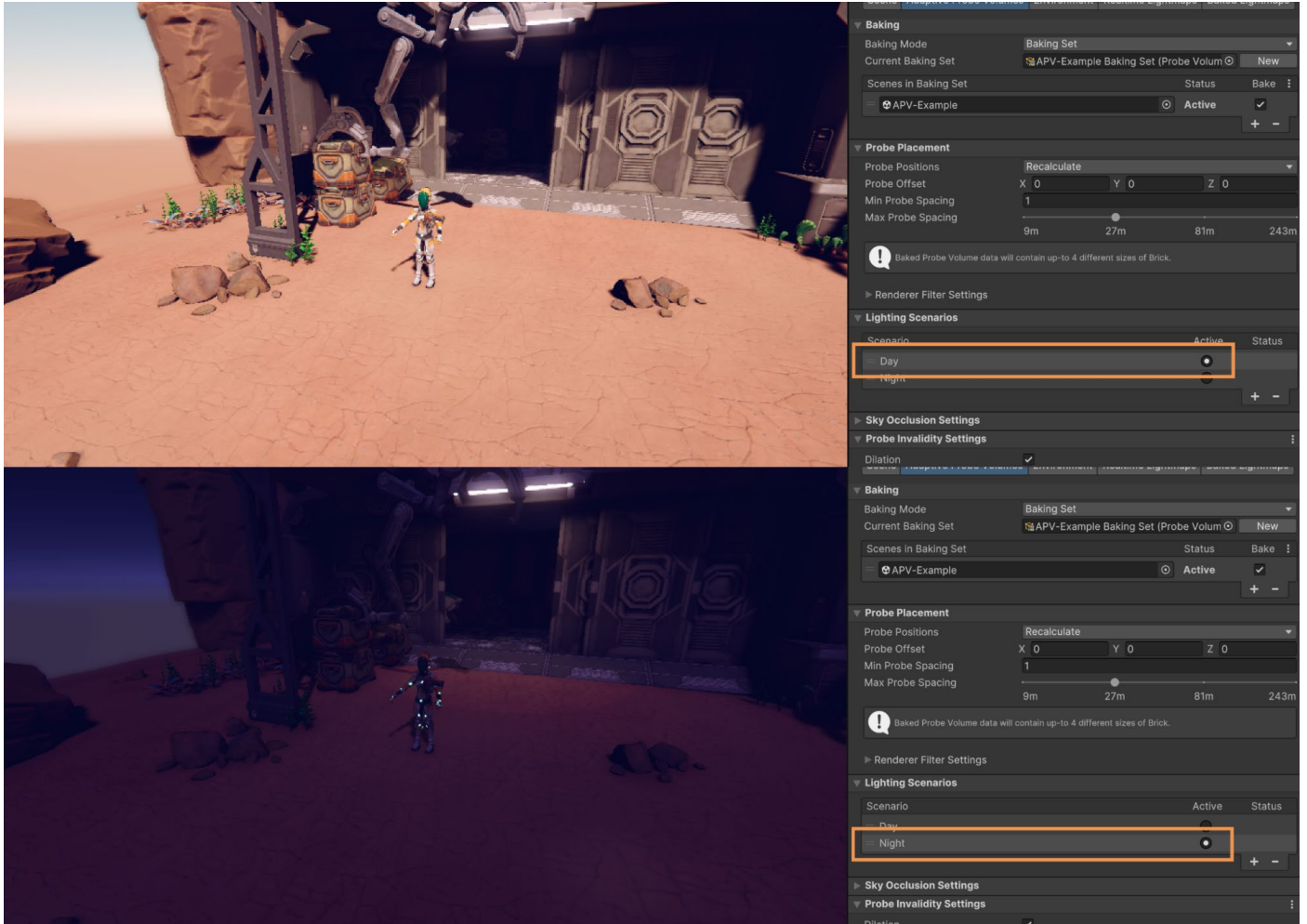
조명 시나리오 에셋

APV의 또 다른 기능은 간접 조명 데이터를 전환하는 것입니다. **조명 시나리오** 에셋에는 씬의 베이킹된 조명 데이터나 **베이킹 세트**가 저장됩니다. 다양한 조명 설정을 서로 다른 조명 시나리오로 베이킹한 다음, 렌더링 디버거를 사용하여 URP가 런타임이나 디자인 시 사용할 요소를 변경할 수 있습니다.



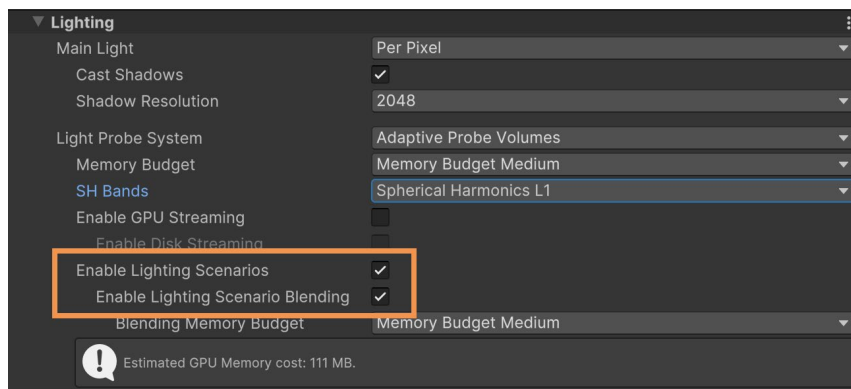
렌더링 디버거를 사용한 시나리오 블렌딩

예를 들어 낮 조명 시나리오 에셋과 밤 조명 시나리오 에셋을 생성할 수 있습니다. 런타임에는 두 시나리오를 전환하거나 블렌딩할 수 있습니다.



낮/밤 조명 시나리오

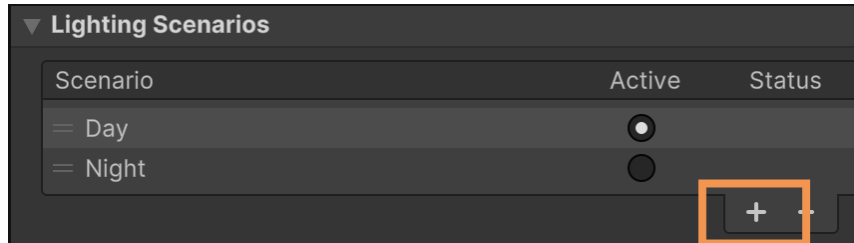
1. 조명 시나리오 에셋을 사용하려면 활성 URP 에셋에서 **Lighting > Light Probe Lighting > Lighting Scenarios**를 활성화합니다.



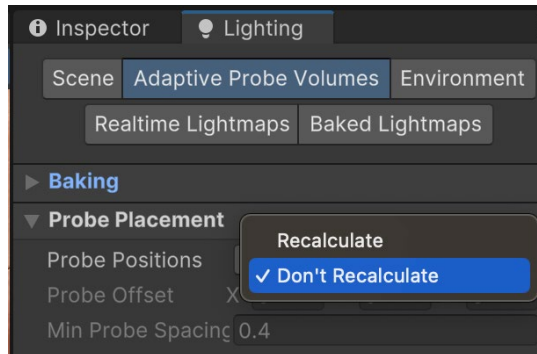
2. 베이킹 결과를 저장하기 위한 새로운 조명 시나리오 에셋을 생성하려면 다음을 수행합니다.

- a. **Lighting** 창에서 **Adaptive Probe Volumes** 패널을 엽니다.

- b. **Lighting Scenarios** 섹션에서 **추가(+)** 버튼을 선택하여 조명 시나리오 에셋을 추가합니다.



3. Lighting 창의 **Adaptive Probe Volume** 탭에서 **Probe Positions**를 **Don't Recalculate**로 설정해야 합니다. 이렇게 설정해야만 Unity에서 프로브 위치를 바꾸는 일 없이 조명만 다시 베이킹합니다. 그렇지 않을 경우에는 이전 베이킹 시나리오가 무효화될 수 있습니다.

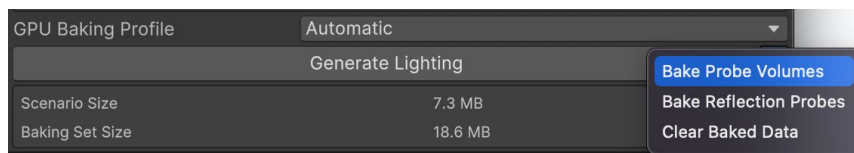


4. 조명 시나리오에 베이킹하는 방법은 다음과 같습니다.

- a. Lighting Scenarios 섹션에서 조명 시나리오를 선택하여 활성화합니다.

- b. **Generate Lighting**을 선택합니다. URP에서 베이킹 결과를 활성화된 조명 시나리오에 저장합니다.

- c. 라이트맵은 사용하지 않고 프로브만 처리하려면 **Generate Lighting** 옆의 드롭다운 버튼을 사용합니다.

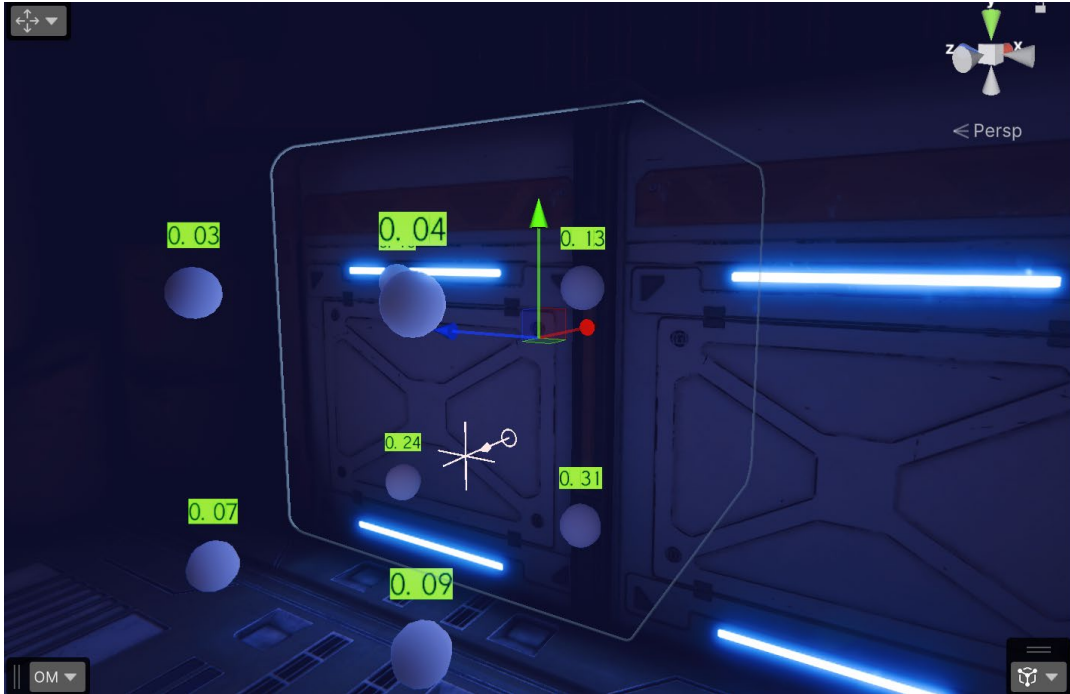


런타임에 URP가 사용할 조명 시나리오를 설정하려면 [ProbeReferenceVolume](#) API를 사용합니다.

참고:

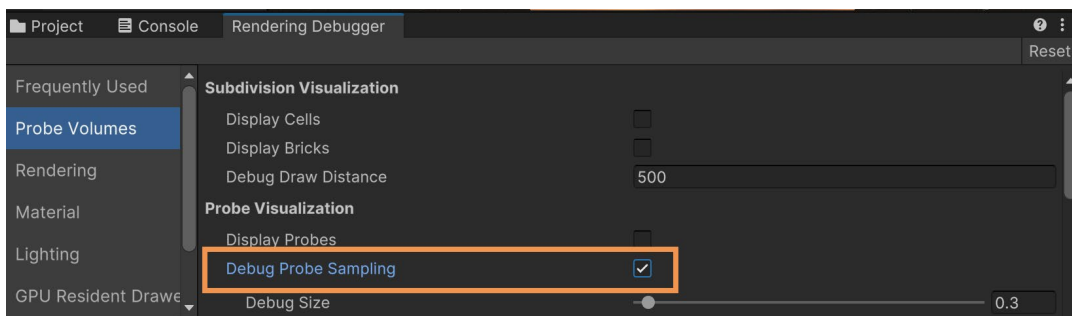
런타임에 활성 조명 시나리오를 변경하면 URP는 라이트 프로브의 간접 조명 데이터만 변경합니다. 지오메트리를 이동하거나 광원 또는 직접 조명을 수정하려면 스크립트를 사용해야 할 수 있습니다.

APV 문제 해결



Debug Probe Sampling

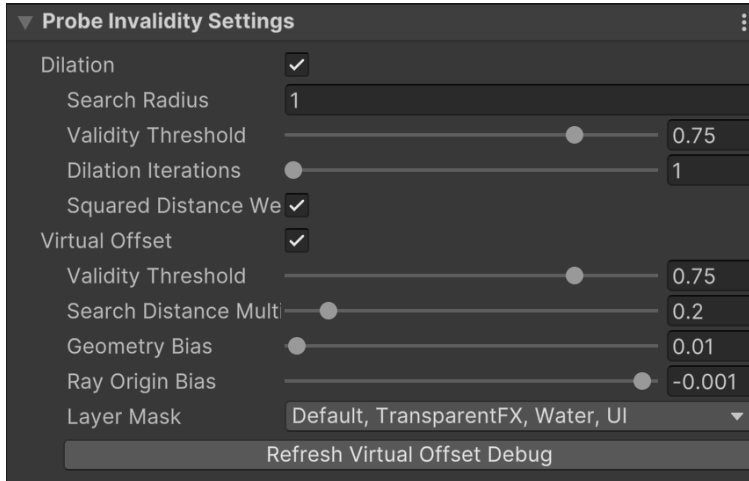
APV 결함 같은 문제를 해결하려면 **Window > Analysis > Rendering Debugger > Probe Volumes > Debug Probe Sampling**을 사용하여 프로브를 검사하고, 특정 픽셀에서 어떻게 샘플링되는지 살펴봅니다.



픽셀별 프로브 샘플링 시각화

라이트 프로브는 그리드 형태로 배치되므로 밝아야 할 영역이 어두워지거나, 그 반대와 같은 렌더링 오류가 발생할 수 있습니다. 에디터에는 테크니컬 아티스트가 이러한 문제를 빠르게 해결할 수 있는 다양한 툴이 있습니다.

지오메트리 내 라이트 프로브는 유효하지 않은 프로브라고 부릅니다. 프로브가 주변 광원 데이터를 수집하기 위해 발사한 샘플링 광선이 지오메트리에서 빛을 받지 않는 후면에 도달하면 URP는 해당 프로브를 유효하지 않은 것으로 표시합니다. APV 시스템에는 이 문제를 해결하기 위한 여러 툴이 있습니다.



Adaptive Probe Volumes 패널의 Probe Invalidation Settings

Virtual Offset은 유효하지 않은 라이트 프로브의 캡처 지점을 콜라이더 밖으로 옮겨 해당 프로브를 유효하게 만들려고 시도합니다. **Dilation**은 Virtual Offset 처리 이후에도 유효하지 않은 라이트 프로브를 감지하여 주변 유효 프로브의 데이터를 전달합니다.

유효하지 않은 라이트 프로브를 확인하려면 렌더링 디버거를 사용합니다.



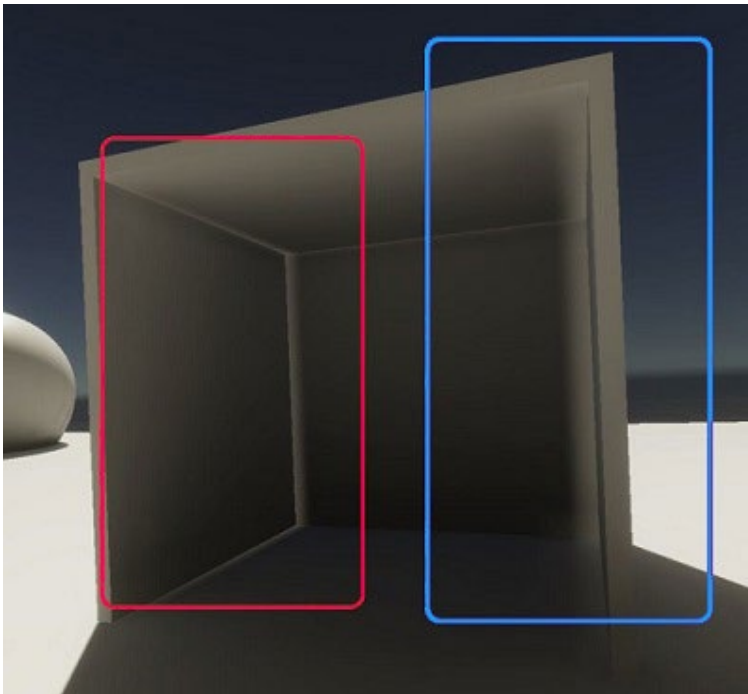
위 이미지의 왼쪽 씬에서는 Virtual Offset이 활성화되지 않아 검은색 띠를 볼 수 있습니다. 오른쪽 씬에서는 Virtual Offset이 활성화되었습니다.



위 이미지의 왼쪽 쉐에서는 Dilation이 활성화되지 않아 일부 영역이 너무 어둡습니다. 오른쪽 쉐에서는 Dilation이 활성화되었습니다.

빛 번짐 현상

빛 번짐 현상은 주로 벽이나 천장의 모서리에서 너무 밝거나 어두운 영역이 발생하는 것을 의미합니다.



빛 번짐 현상

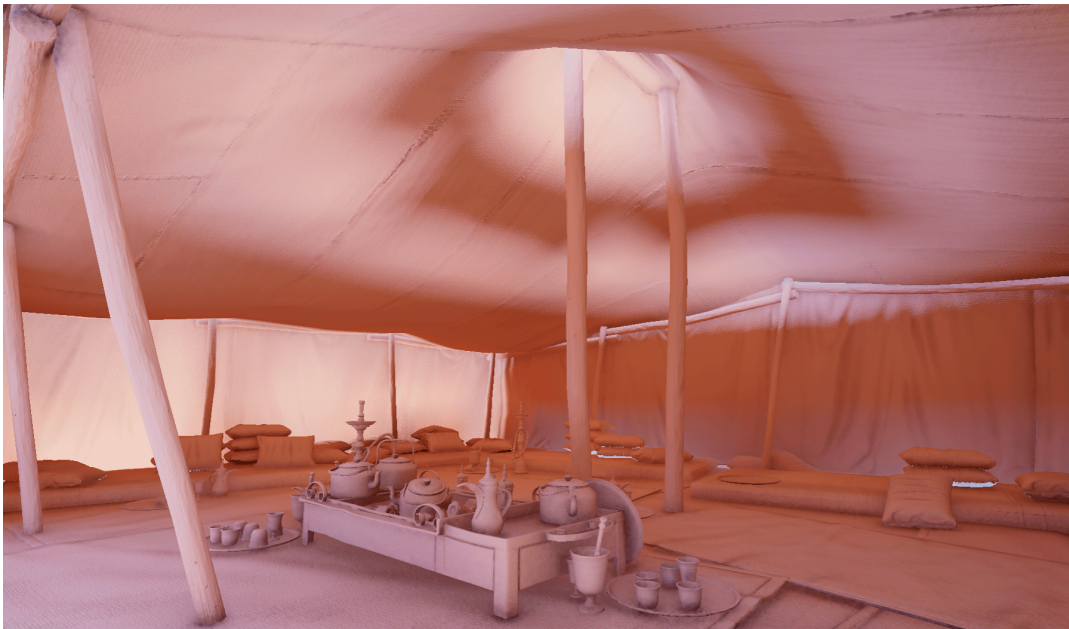
빛 번짐 현상은 지오메트리가 해당 지오메트리에서 볼 수 없는 벽 반대편의 라이트 프로브에서 빛을 받을 때 주로 발생합니다. APV는 일반적인 그리드 형태의 라이트 프로브를 사용하므로 라이트 프로브가 벽의 형태를 따르지 않거나 서로 다른 조명 영역 간의 경계에 있을 수 있습니다.

빛 번짐 현상을 해결하기 위해 시도할 만한 기법은 다음과 같습니다.

- 벽을 더 두껍게 만듭니다.
- 다음과 같이 씬에 **Adaptive Probe Volumes Options 오버라이드**를 추가합니다.
 - 볼륨을 추가한 다음, 해당 볼륨에 **Adaptive Probe Volumes Options** 오버라이드를 추가합니다. 이렇게 하면 게임 오브젝트가 라이트 프로브를 샘플링하기 위해 사용하는 위치가 조정됩니다.
- **렌더링 레이어**를 활성화합니다.
 - Lighting 창의 **Adaptive Probe Volumes** 패널에서 **Rendering Layer Masks**를 설정하여 APV가 각 라이트 프로브에 렌더링 레이어 마스크를 할당하도록 합니다.
- **Baking Set 프로퍼티**를 조정합니다.
 - 볼륨을 추가해도 해결되지 않으면 Lighting 창의 Adaptive Probe Volumes 패널에서 Virtual Offset 및 Dilation 설정을 조정합니다.
- **Probe Adjustment Volume 컴포넌트**를 사용합니다.
 - 이 컴포넌트를 사용하여 작은 영역의 라이트 프로브를 유효하지 않게 만듭니다. 이렇게 하면 베이크 시 Dilation이 트리거되며, 런타임에 빛 번짐 현상 방지 모드(Leak Reduction Mode)의 결과가 향상됩니다.

렌더링 레이어

라이트 프로브/라이트맵을 사용하던 URP 3D 샘플 오아시스 환경에서 APV만 사용하도록 전환하면, 아래 이미지의 밝은 천장과 벽에서 볼 수 있는 빛 번짐 현상 문제가 발생합니다.



URP 3D 샘플 오아시스 환경의 텐트 내 빛 번짐 현상

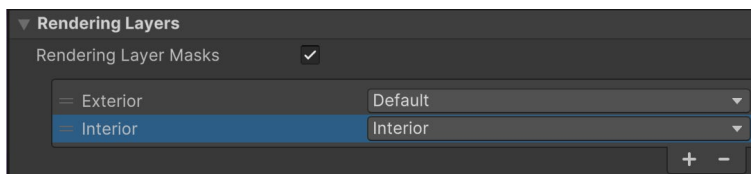
일부 픽셀에서 텐트 내부와 외부의 프로브가 블렌딩되며 발생하는 현상입니다. 픽셀의 값을 보간할 때 어떤 프로브가 사용되는지 **Window > Analysis > Rendering Debugger > Probe Volumes > Debug Probe Sampling**을 통해 확인할 수 있습니다.



픽셀의 보간된 프로브 확인

이 문제를 해결하는 방법 중 하나는 볼륨을 활용하여 **Adaptive Probe Volume Options** 오버라이드로 런타임에 APV가 샘플링되는 방식을 변경하는 것입니다. **NormalBias**와 **ViewBias** 설정은 샘플링 위치를 조정하는 데 유용할 수 있습니다. **NormalBias**는 노멀 방향을 따라 벽에서 멀어지는 방향으로 위치를 조정하고, **ViewBias**는 카메라를 향해 카메라와 같은 방향의 벽으로 위치를 조정합니다. 볼륨의 두 프로퍼티를 변경하면 그에 따라 샘플링 위치와 가중치가 업데이트되어 조명 결과와 **Debug Probe Sampling** 뷰에서 실시간으로 업데이트를 확인할 수 있습니다. 하지만 더 좋은 방법은 렌더링 레이어를 사용하는 것입니다.

APV에서 지원하는 렌더링 레이어를 사용하면 최대 4개의 서로 다른 마스크를 만들고 특정 오브젝트의 샘플링을 해당 마스크로 제한할 수 있습니다. 내부 오브젝트가 외부 프로브를 샘플링하는 것을 방지하거나 그 반대의 경우를 방지하는 데 유용한 방법입니다. **Window > Rendering > Lighting > Adaptive Probe Volumes > Rendering Layers**에서 활성화하고 추가할 수 있습니다.



Project Settings > Tags and Layers > Rendering Layers를 통해 레이어도 추가해야 합니다.

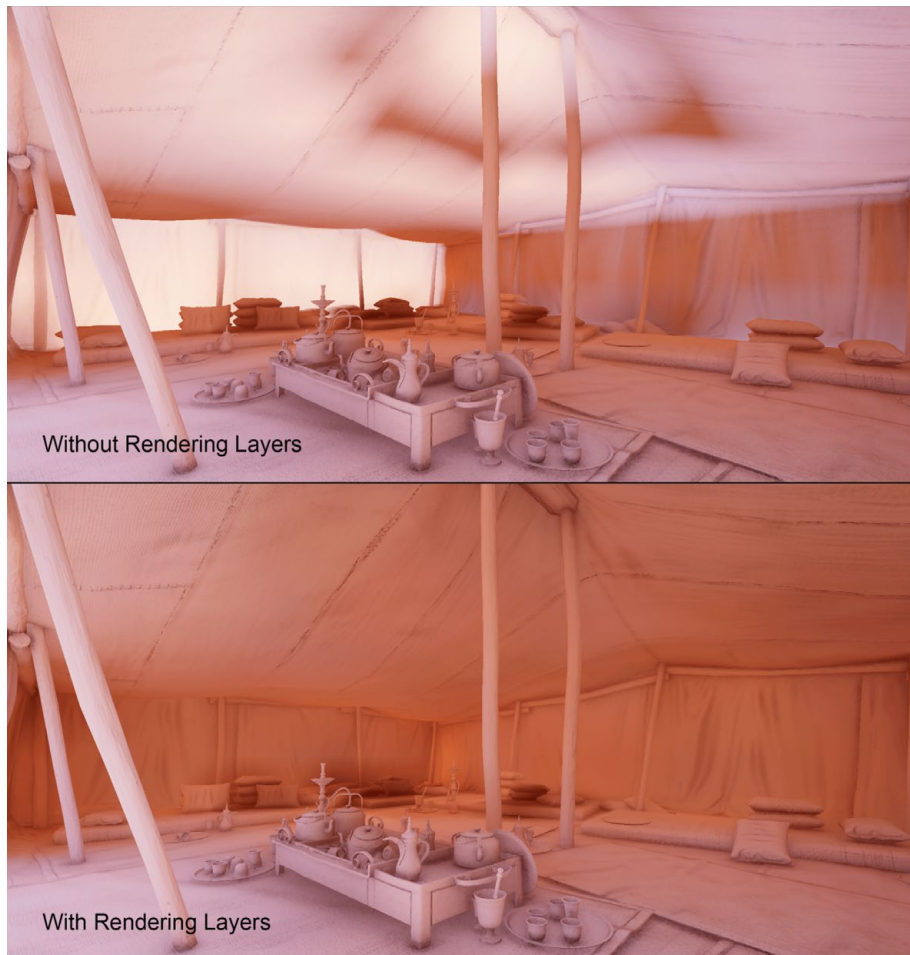


이렇게 구현하려면 메시 자체를 수정해서 생성하려는 여러 영역에 분할되도록 만들어야 합니다. 예를 들어 이 프로젝트에서는 메시를 수정하여 내부와 외부를 여러 메시로 분할합니다. 메시가 분할된 후, 올바른 렌더링 레이어를 할당하고 **Adaptive Probe Volume** 탭에서 APV가 사용할 렌더링 레이어를 지정합니다.

텐트 내 모든 오브젝트에 레이어를 할당할 필요는 없고, 벽이나 벽 주변 오브젝트처럼 빛 번짐 현상이 발생할 수 있는 오브젝트에만 할당하면 됩니다.

시스템은 조명을 생성할 때 베이크 프로세스 중에 주변 오브젝트를 기반으로 프로브에 레이어를 자동으로 할당하므로 프로브별 레이어를 수동으로 할당할 필요가 없습니다. 이 자동 프로브 할당 기능을 활용하려면 대형 오브젝트에 레이어를 할당해야 합니다. 오아시스 환경의 텐트 예시에서는 텐트의 벽과 천장에 내부 레이어가 할당되어 베이크하는 동안 대부분의 내부 프로브가 닿아 내부 마스크에 자동으로 할당됩니다. 프로브는 가장 많이 닿는 레이어에 할당됩니다.

이 작업이 완료되면 **Generate Lighting**을 클릭하여 별도의 내부 및 외부 마스크 덕분에 텐트의 빛 번짐 현상이 사라진 것을 확인할 수 있습니다.



렌더링 레이어 사용 유무에 따른 빛 번짐 현상

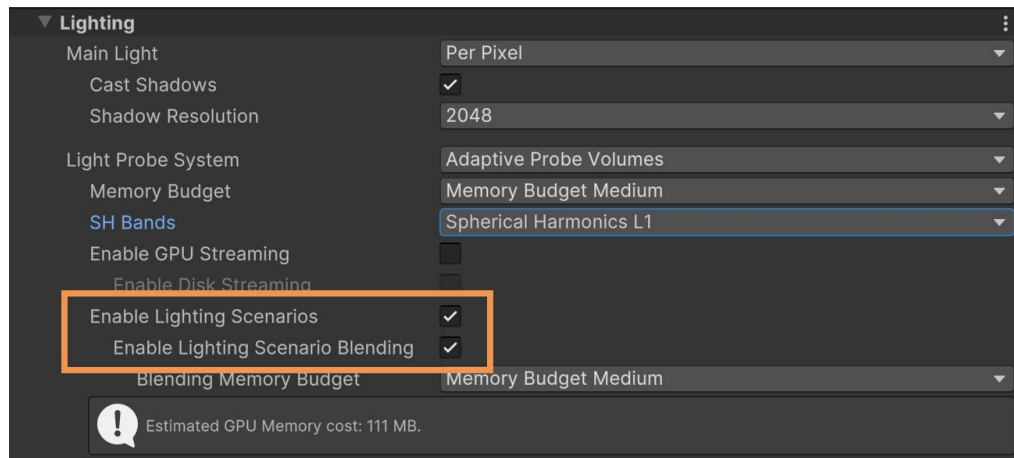
APV 관련 문제 해결 방법을 자세히 알아보려면 [여기](#)를 참고하세요.

APV 스트리밍

APV 스트리밍을 활용하면 대형 월드에서 APV 기반 조명을 사용할 수 있습니다. APV 스트리밍은 사용 가능한 CPU 또는 GPU 메모리를 초과하는 APV 데이터를 베이크한 후, 런타임에 필요할 때 로드합니다. 런타임에 카메라 움직임에 따라 URP가 카메라의 뷰 절두체 내의 셀에서 APV 데이터만 로드합니다.

여러 URP 품질 수준별로 스트리밍을 활성화하거나 비활성화할 수 있습니다. 스트리밍을 활성화하는 방법은 다음과 같습니다.

1. 메인 메뉴에서 **Edit > Project Settings > Quality**를 선택합니다.
2. 품질 수준을 선택합니다.
3. 렌더 파이프라인 에셋을 더블 클릭하여 인스펙터에서 엽니다.
4. Lighting 탭을 펼칩니다.
5. 이제 두 가지 유형의 스트리밍을 활성화할 수 있습니다.
 - a. Disk Streaming을 활성화하면 디스크에서 CPU 메모리로 스트리밍합니다.
 - b. GPU Streaming을 활성화하면 CPU 메모리에서 GPU 메모리로 스트리밍합니다. Enable Disk Streaming을 먼저 활성화해야 합니다.

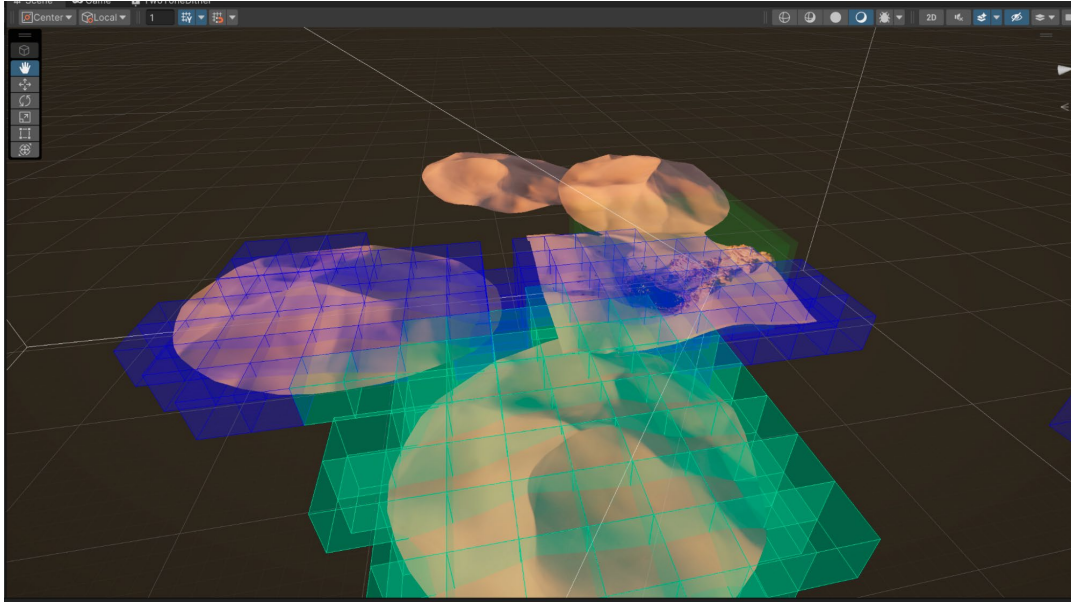


같은 창에서 스트리밍 설정도 구성할 수 있습니다. 자세히 알아보려면 URP 에셋 섹션을 참고하세요.

디버그 스트리밍

URP가 로드하고 사용하는 가장 작은 섹션 단위는 셀이며 APV의 가장 큰 브릭과 같은 크기입니다. 라이트 프로브의 밀도를 조정하여 APV 내 셀의 크기를 조절할 수 있습니다.

렌더링 디버거를 사용하면 APV 또는 디버그 스트리밍 내 셀을 확인할 수 있습니다.



APV 스트리밍

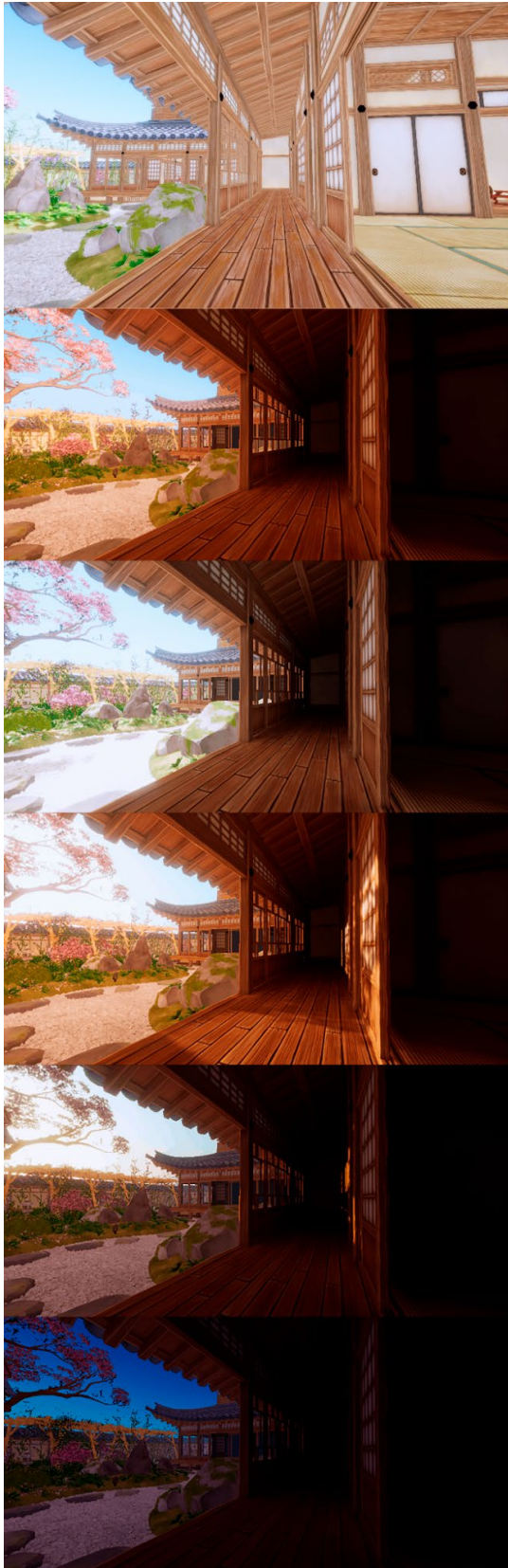
스카이 오클루전

스카이 오클루전은 게임 오브젝트가 하늘의 색상을 샘플링하는 경우, 빛이 게임 오브젝트에 도달하지 못한다면 Unity가 해당 색상을 어둡게 만드는 프로세스입니다. Unity에서 스카이 오클루전은 런타임에 업데이트되는 앰비언트 프로브의 하늘 색상을 사용합니다. 따라서 하늘 색상의 변화에 따라 동적으로 게임 오브젝트에 조명을 적용할 수 있습니다. 예를 들어 하늘을 밝은색에서 어두운색으로 변경하여 낮/밤 전환 효과를 시뮬레이션할 수 있습니다.

참고: 스카이 오클루전을 활성화하면 APV 베이크 시간이 길어지고 Unity의 런타임 메모리 사용량이 늘어날 수 있습니다.

스카이 오클루전을 활성화하면 Unity가 APV의 각 프로브에 정적 스카이 오클루전 값을 추가로 베이크합니다. 스카이 오클루전 값은 프로브가 하늘에서 받는 간접광의 양을 나타내며, 여기에는 정적 게임 오브젝트에서 반사된 빛도 포함됩니다.

스카이 오클루전의 주된 장점은 런타임에 하늘 조명을 변경할 수 있다는 점입니다.



왼쪽에 보이는 일련의 이미지를 함께 살펴보겠습니다.

- 가장 위쪽의 이미지는 하늘 조명이 런타임에 변해야 하기 때문에 하늘 조명을 베이킹할 수 없을 때 발생하는 문제를 보여 줍니다. 이 이미지는 베이킹 없이 앰비언트 프로브만 사용되므로 결과물의 품질이 좋지 않습니다.
- 두 번째부터 다섯 번째 이미지에는 앰비언트 프로브와 함께 스카이 오클루전이 사용되었습니다. 스카이 오클루전을 비활성화한 채 일반적인 APV 베이킹만으로 조명을 적용할 수도 있겠지만, 그럴 경우 런타임에 조명이 변하지 않을 것입니다.

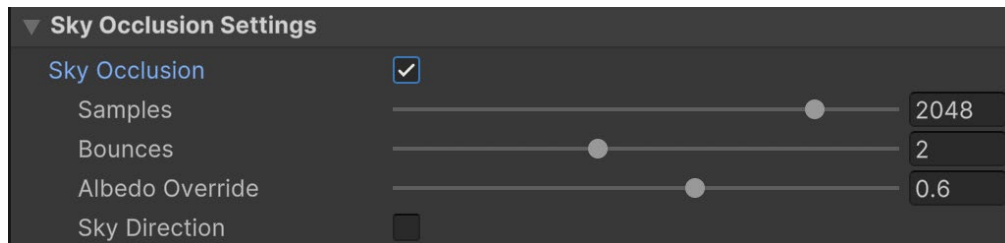
왼쪽 스카이 오클루전을 사용한 예시. 이미지 출처: Unity 에셋 스토어에 등록된 7stars의 [Azure\[Sky\] Dynamic Skybox](#) 패키지.

스카이 오클루전을 활성화하는 방법은 다음과 같습니다.

1. GPU Light Baker(이전 명칭: Progressive GPU Lightmapper)를 활성화합니다. Progressive CPU를 사용하는 경우 Unity에서 스카이 오클루전을 지원하지 않습니다. **Window > Rendering > Lighting**으로 이동합니다.
2. Scene 패널로 이동합니다.
3. Lightmapper를 Progressive GPU로 설정합니다.

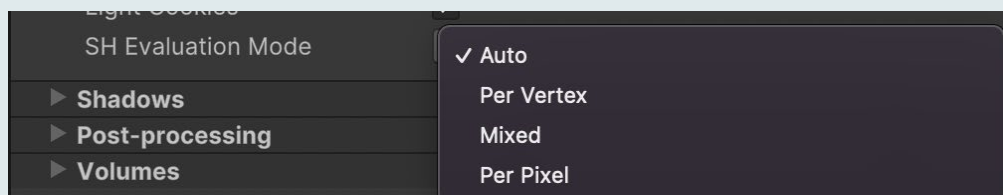


4. Adaptive Probe Volumes 패널을 엽니다.
5. Sky Occlusion을 활성화합니다.



조명 데이터를 업데이트하려면 스카이 오클루전을 활성화하거나 비활성화한 다음 APV도 베이크해야 합니다. 스카이 오클루전이 베이크되면 씬 조명이 앰비언트 프로브 업데이트에 반응합니다. URP에서는 Color 또는 Gradient 모드를 사용할 때만 앰비언트 프로브가 실시간으로 업데이트되며, Skybox 모드에서는 실시간으로 업데이트되지 않습니다. 따라서 애니메이션이 적용된 하늘 비주얼에 따라 색상을 직접 애니메이션화해야 할 것입니다.

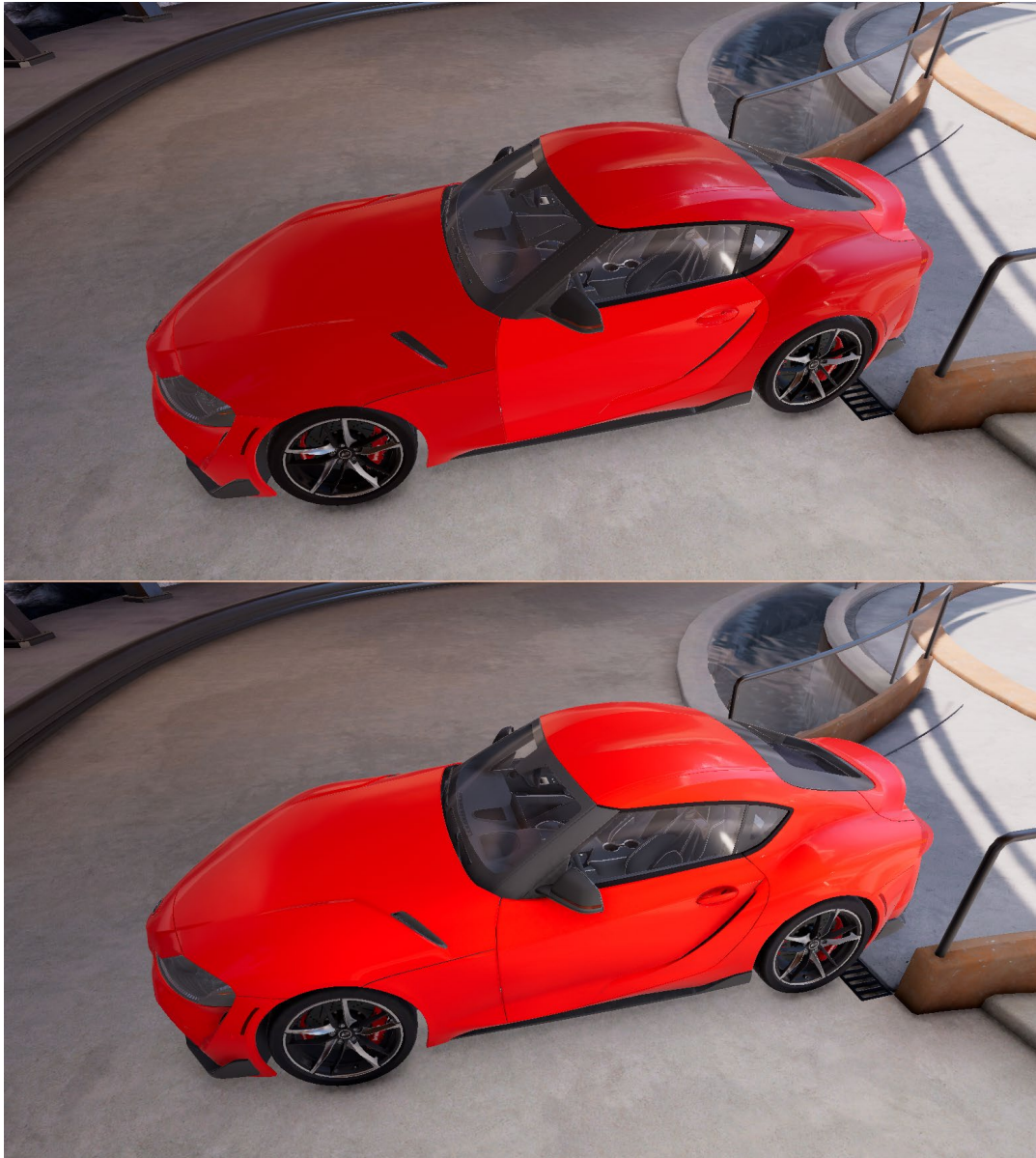
참고: 이제 URP에서 프로브에 버텍스별 품질 샘플링을 지원합니다. 특히 저사양 기기에서 성능을 향상하는 데 유용한 기능입니다. 샘플링 모드를 설정하려면 Lighting 섹션의 **URP 에셋**을 사용합니다. **Advanced Properties**를 활성화해야 해당 옵션이 표시됩니다. Lighting 패널 오른쪽 상단의 줄임표를 눌러 활성화할 수 있습니다. Advanced Properties를 활성화하면 **SH Evaluation Mode** 드롭다운이 표시됩니다.



더 많은 정보

- 적응적 프로브 볼륨 [기술 자료](#)
- GDC 2023 세션: [적응적 프로브 볼륨으로 효율적이고 효과적인 조명 구현하기](#)

라이트 프로브와 APV 비교



라이트 프로브 그룹이 사용된 상단 이미지와 APV가 사용된 하단 이미지. 이미지 출처: Unity 에셋 스토어에 등록된 ArchVizPro의 [ArchVizPRO Photostudio URP](#) 패키지

아래쪽 이미지를 참고하면 APV를 사용할 경우 어두운 영역에서 밝은 영역으로 얼마나 부드럽게 전환되는지 확인할 수 있습니다. 위쪽 이미지에서는 라이트 프로브 그룹을 사용했을 때 오브젝트당 하나의 보간된 프로브만 사용되므로 자동차 문에 밝은 빛이 생깁니다. 해당 문이 나머지 문과 별개의 게임 오브젝트이며 다른 프로브를 사용하기 때문에 이러한 렌더링 오류가 발생합니다.

아래는 라이트 프로브와 APV의 기능을 비교한 표입니다.

라이트 프로브 그룹	적응적 프로브 볼륨
지오메트리 변화하는 경우 프로브를 배치하고 이동하는 데 시간이 소모됨	지오메트리 변화에 따라 빠르게 배치하고 쉽게 업데이트할 수 있음
오브젝트를 비추는 데 하나의 보관된 프로브만 사용: — 오브젝트가 어두운 영역에서 밝은 영역으로 부드럽게 전환되지 않아 색상의 차이가 눈에 띄게 됩니다. — 대형 오브젝트에서 문제가 발생할 수 있습니다.	각 픽셀의 빛을 개별적으로 처리: — 부드럽게 전환할 수 있습니다. — APV 그리드는 어느 위치에서든 쉽게 샘플링할 수 있으므로 APV를 볼류메트릭 효과와 함께 사용하기 좋습니다.
정적 오브젝트는 주로 라이트맵으로 조명이 적용됩니다. 동적 오브젝트만 프로브를 사용합니다.	라이트맵이나 라이트맵 UV가 필요하지 않음: — 씬 내 모든 오브젝트에 하나의 조명 솔루션을 사용합니다. — 제한된 메모리 할당량으로 대형 월드에 조명을 적용할 수 있습니다.
런타임에 프로브를 자유롭게 배치하고 이동할 수 있습니다.	프로브는 그리드 구조로 배치되며 런타임에 이동할 수 없습니다.
GI 전환을 지원하지 않습니다.	조명 시나리오 에셋을 사용하여 낮/밤, 광원 켜기/끄기 등 다양한 유형의 조명을 전환할 수 있습니다.

스크린 공간 굴절



클라우드오 노로리와 안토니오 바르가스가 개발한 인디 게임 [Arctico](#)에서 플레이어는 베이스캠프를 건설하고 빙하 지역을 탐험해야 합니다. 게임의 많은 부분을 차지하는 물이 표면을 반사하며, 이 효과는 스크린 공간 반사로 구현할 수 있습니다. 스크린 공간 반사를 사용하면 실시간으로 반사 표면을 모방할 수 있으며, 스크린 공간 굴절을 사용하면 광원이 매체를 통과할 때 투명도와 빛의 굴절을 시뮬레이션할 수 있습니다.



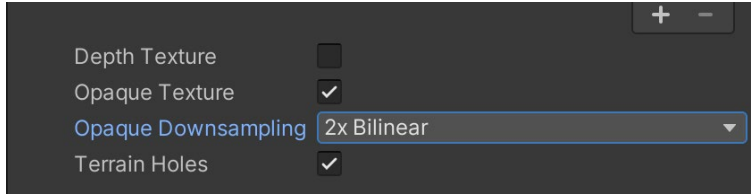
스크린 공간 굴절 예시

스크린 공간 굴절은 렌더 파이프라인에서 생성된 현재의 불투명 텍스처를 소스 텍스처로 사용해 렌더링 중인 모델에 픽셀을 매핑합니다. 불투명 텍스처의 일부가 아닌 모델은 표시할 수 없습니다. 이 방법은 이미지 샘플링에 사용되는 UV를 변형하는 것입니다.

이 레시피에서는 노멀 맵을 사용해 굴절 효과를 만들고 굴절 효과에 컬러를 지정하는 방법을 소개합니다. 위 이미지에 보이는 것처럼 추가로 컬러를 지정하려면 계산된 픽셀 컬러를 **Color** 프로퍼티로 선형 보간하면 됩니다.

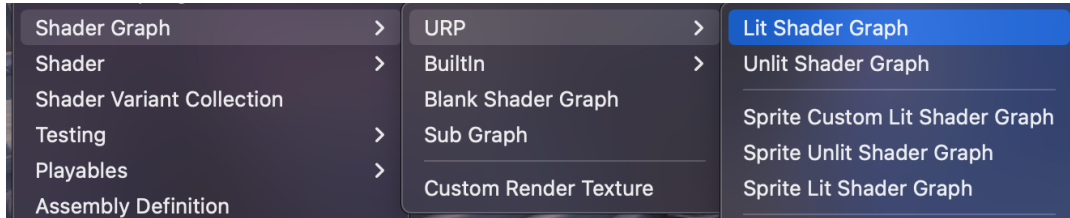
이 효과가 실제로 적용된 모습을 보려면 **Scenes > Refraction > Refraction**을 살펴보세요.

스크린 공간 굴절을 사용하려면 셰이더에서 불투명 텍스처를 사용할 수 있어야 합니다. **Edit > Project Settings... > Graphics > Scriptable Render Pipeline Settings**에서 현재 할당된 URP 설정 에셋을 찾습니다. 인스펙터에서 **Opaque Texture**가 활성화되어 있어야 합니다. Opaque Downsampling도 활성화하면 성능이 일부 향상됩니다. 굴절된 오브젝트를 통해 보이는 모습이 약간 블러 처리되므로 시각적인 향상 효과도 기대할 수 있습니다.



Opaque Texture 및 Opaque Downsampling 설정

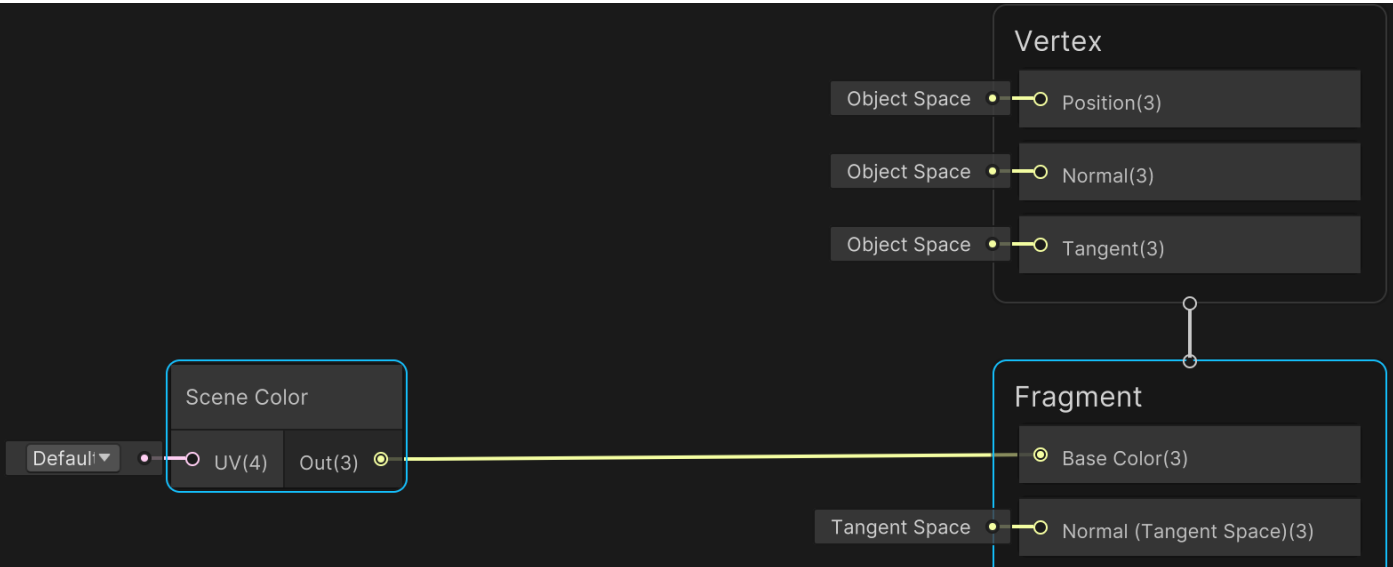
셰이더를 생성하려면 먼저 새 Shader Graph 에셋을 생성해야 합니다. 프로젝트 창을 오른쪽 클릭하고 **Create > Shader Graph > URP > Lit Shader Graph**를 선택합니다.



새 Lit Shader Graph 생성

Shader Graph 에셋을 선택하고 **Create > Material**을 선택해 이 셰이더를 사용할 머티리얼을 생성합니다. 그리고 해당 머티리얼을 굴절시키려는 오브젝트에 적용합니다.

이제 Shader Graph 에셋을 더블 클릭해 엽니다. **Scene Color** 노드를 생성하고 **Fragment > Base Color**에 연결합니다.



Scene Color 노드 사용

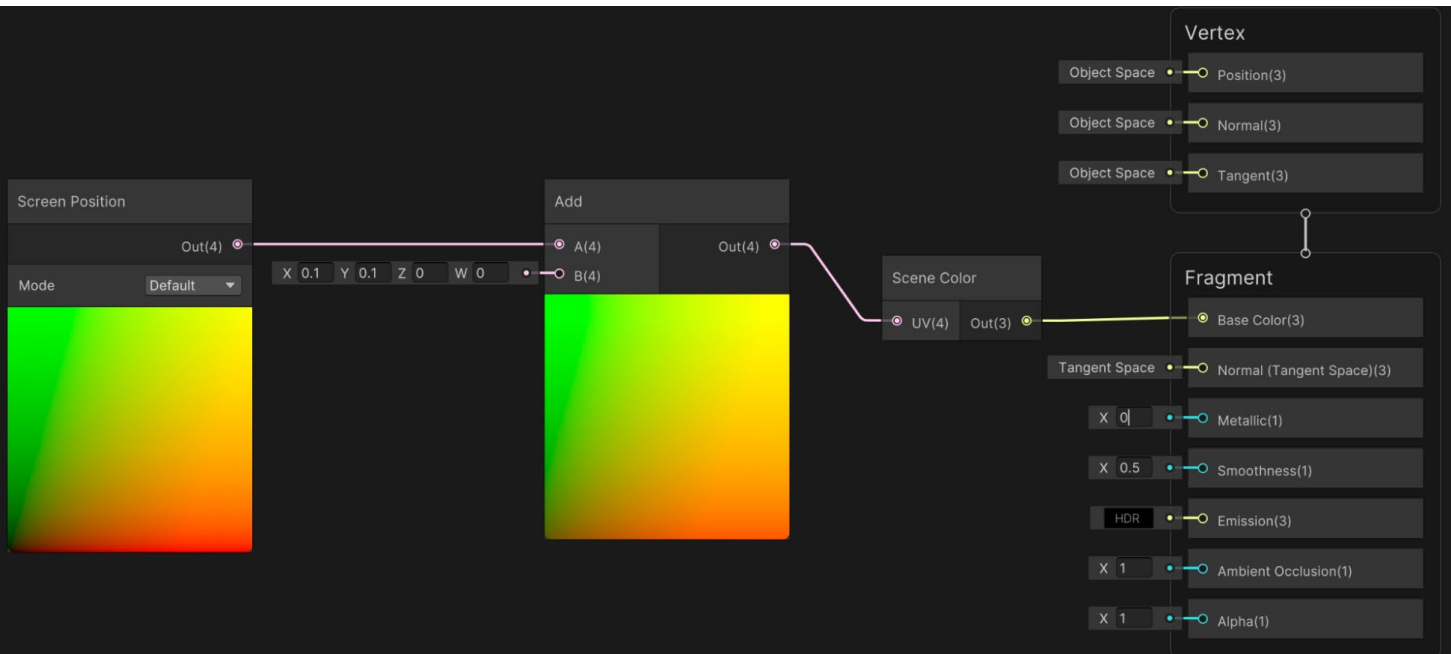
Scene Color 노드는 렌더링된 불투명 오브젝트를 사용하므로 투명한 머티리얼에서만 작동합니다. 렌더 파이프라인에서 투명 오브젝트는 불투명 오브젝트 다음에 렌더링됩니다. **Graph Inspector > Graph Settings > Surface Type**을 **Transparent**로 설정합니다.

Scene Color 노드는 기본적으로 UV에 정규화된 화면 좌표를 사용하므로, 평활도의 영향을 받는 조명이 있는 각 픽셀에 불투명 텍스처를 매핑해서 아래 이미지와 같은 결과를 얻을 수 있습니다.



Scene Color 사용 결과

Scene Color 노드에서 사용하는 UV를 조작해야 하므로 기본 UV 동작을 오버라이드해야 합니다. **Screen Position** 노드와 **Add** 노드를 생성합니다. Screen Position 노드의 출력을 Add 노드의 입력 A로 드래그하고 입력 B를 [0.1, 0.1, 0, 0]로 설정합니다.



UV 제어를 위한 노드 추가

이제 불투명 텍스처 오프셋이 표시됩니다.



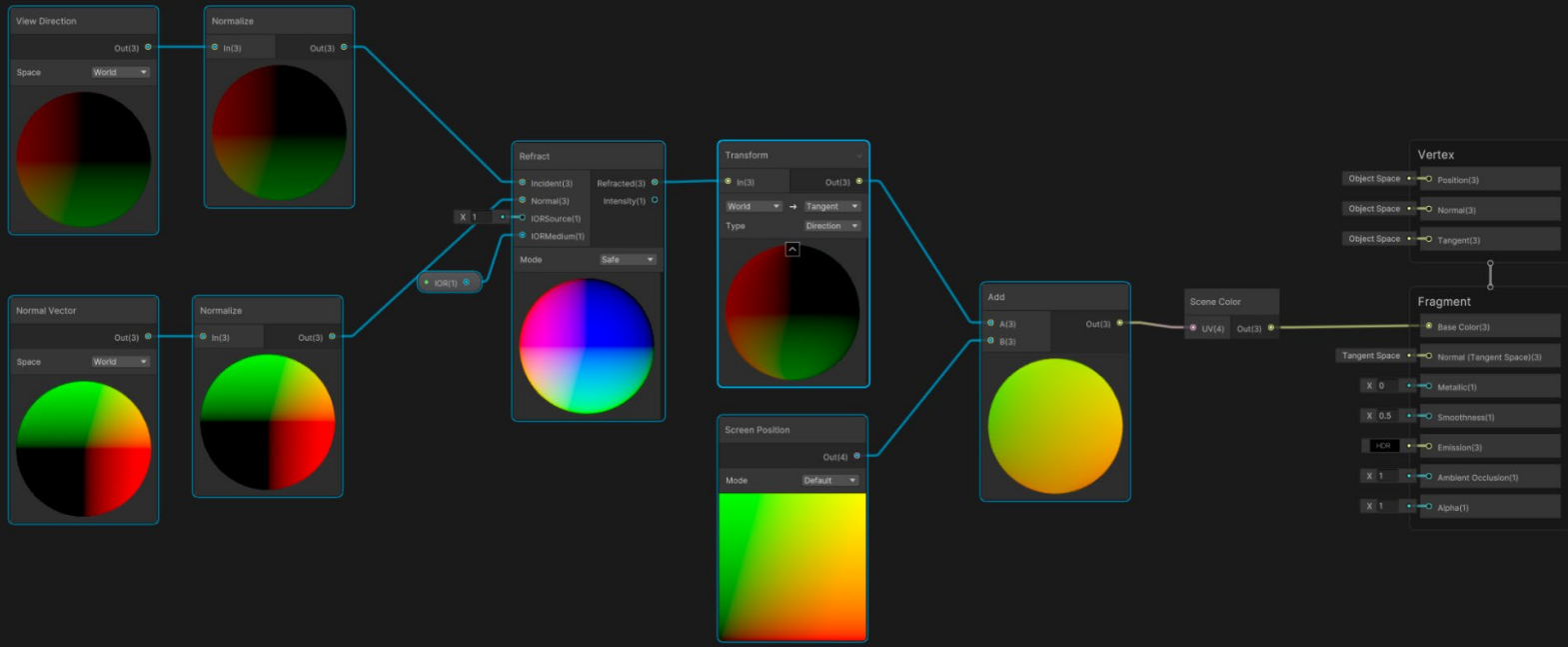
불투명 텍스처 오프셋

렌더링된 각 픽셀에 대해 카메라의 뷰 방향, 현재 화면 위치에서 오브젝트의 노멀, 스케일링 값 기준으로 오프셋을 제어해야 합니다. Shader Graph에는 이 세 가지 입력이 있을 때 굴절을 계산하는 노드가 있습니다. 실제 스케일링 값은 두 개이나 여기서는 하나만 사용하겠습니다.

스케일링을 위해 **IOR**([Index of Refraction](#)의 줄임말)이라는 새로운 플로트 프로퍼티를 추가한 다음, 최솟값이 1이고 최댓값이 6인 슬라이더로 설정합니다. 뷰 방향의 경우 **View Direction** 노드를 추가하고 단위 길이를 사용하도록 **Normalize** 노드에 연결합니다.

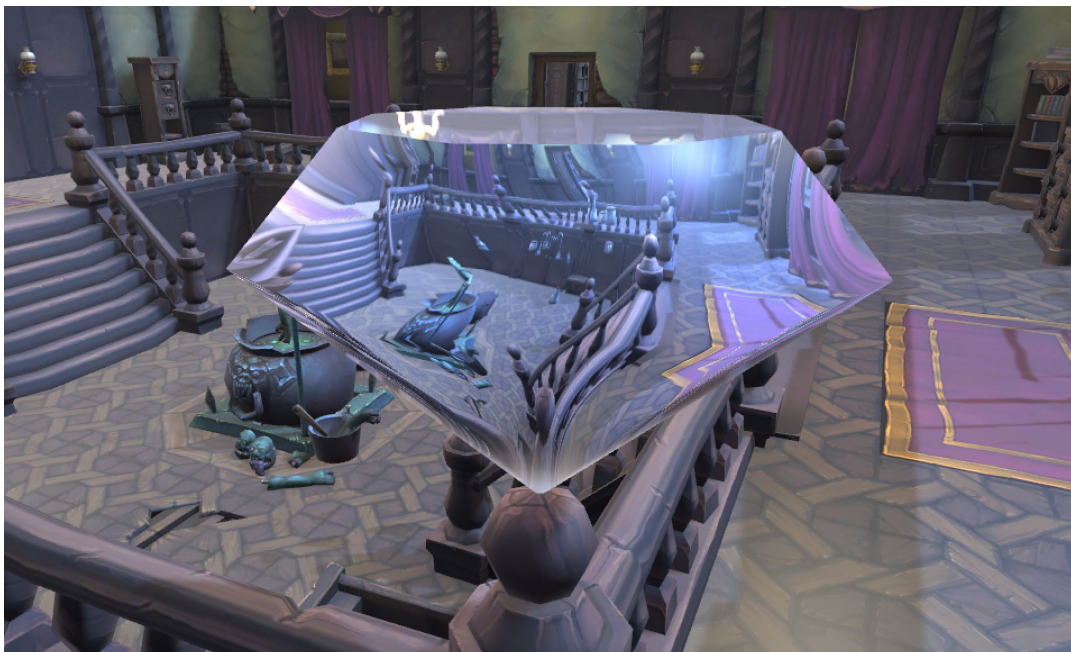
Normal 노드 세트를 **World Space**에 추가한 다음, 다시 **Normalize** 노드에 연결합니다. **Refract** 노드를 생성하고 정규화된 **View Direction** 값을 **Incident** 입력에 연결합니다. 정규화된 **Normal**을 **Refract** 노드의 **Normal** 입력에 연결하고 **IOR** 프로퍼티를 **IORMedium** 입력에 연결합니다.

이 시점에 Refracted 출력은 World Space에 있지만, 스크린 공간 UV를 오프셋하려면 Tangent Space에 있어야 합니다. Transform 노드를 추가해 입력은 World로, 출력은 Tangent로 설정합니다. Type은 Direction을 선택합니다. 이 값을 Add 노드의 A 입력으로 사용하고 Screen Position을 B 입력으로 사용합니다. 그러면 아래와 같은 그래프가 만들어집니다.



기본 굴절 그래프

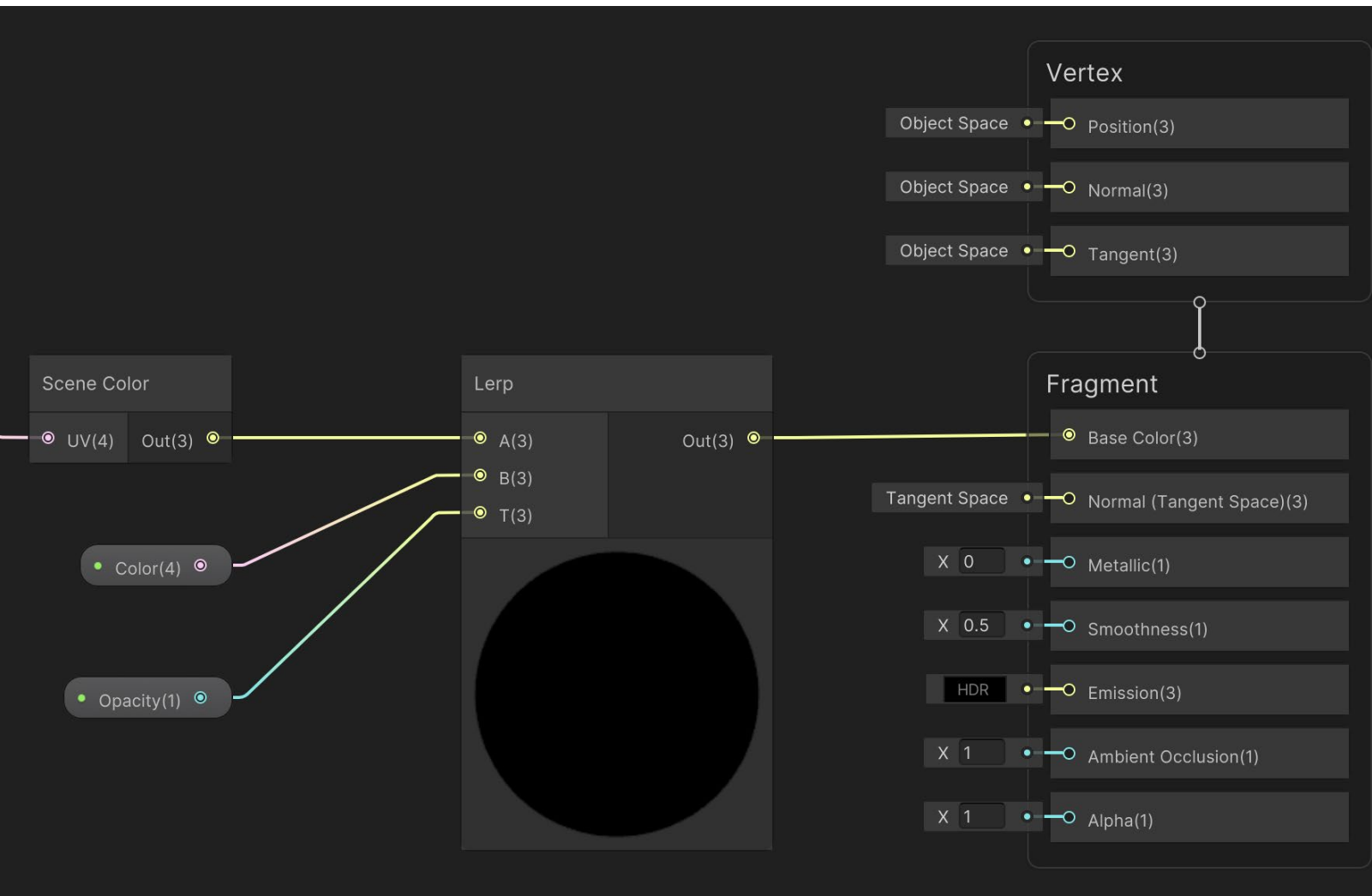
IOR이 5.44라면 다음 이미지와 같은 시각 효과를 얻을 수 있습니다.



기본 스크린 공간 굴절



Color 프로퍼티를 추가하면 결과물에 컬러를 지정할 수 있습니다. Lerp 노드를 추가하고 **Opacity** 프로퍼티를 0~1 범위로 슬라이더 모드에 설정해 T 입력으로 사용합니다. Scene Color의 출력은 입력 A로 연결하고 Color는 입력 B로 연결합니다.



그래프에 컬러 지정 단계 추가



이제 결과물에 컬러를 지정할 수 있습니다.

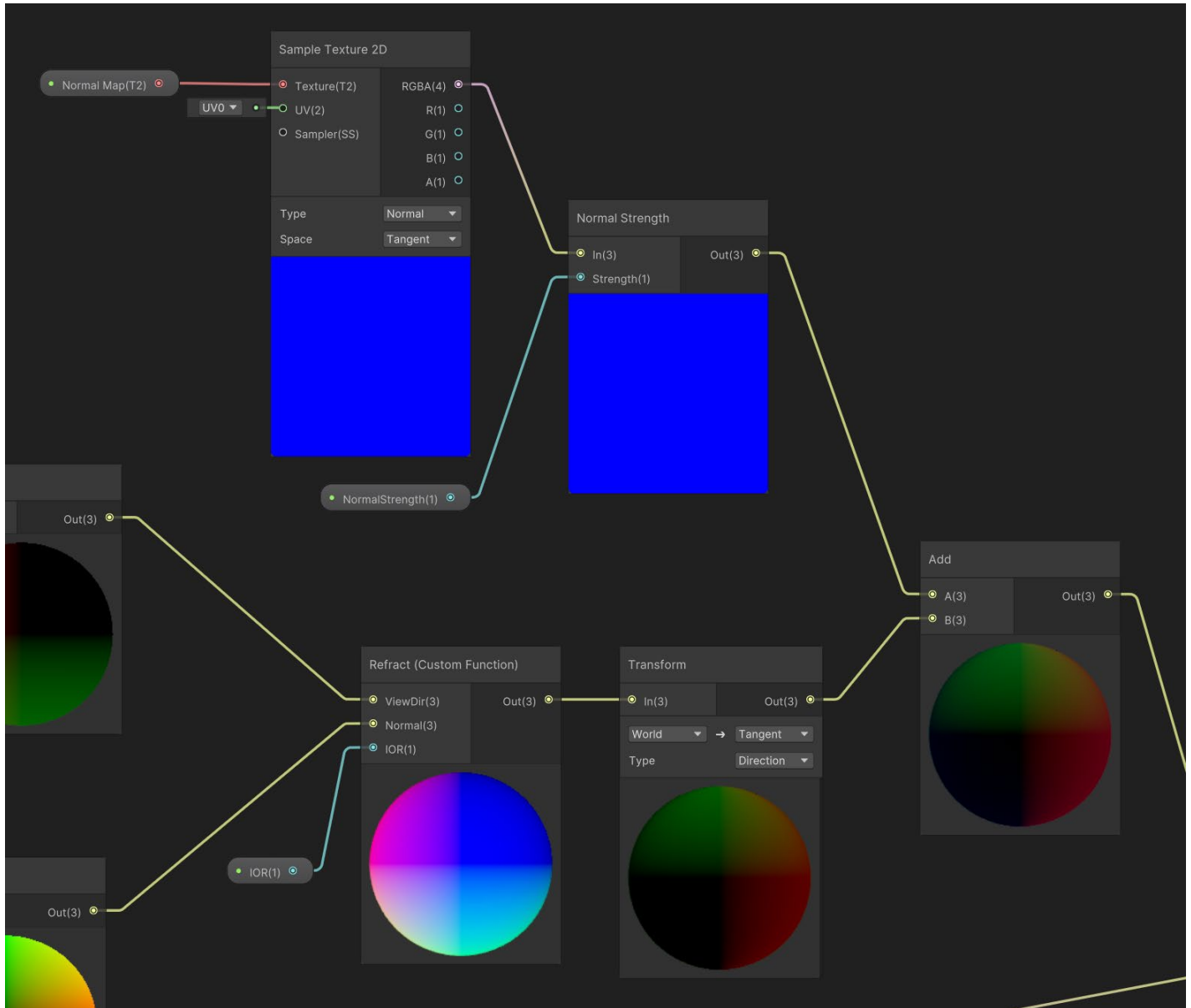


컬러를 지정한 버전

노멀은 굴절에 영향을 주기 때문에 단일 평면은 불투명 텍스처의 오프셋 버전만 얻게 됩니다.



이제 노멀 맵을 추가할 차례입니다. 먼저 셰이더에 **Texture 2D** 프로퍼티를 추가해 이름을 **Normal Map**으로 설정하고, 0~1 범위의 플롯 프로퍼티를 슬라이더로 추가해 이름을 **Normal Strength**로 설정합니다. **Sample Texture 2D** 노드를 생성하고 **Type**을 **Normal**로, **Space**를 **Tangent**로 설정합니다. **Normal Map** 프로퍼티에 **Texture** 입력을 설정합니다. **Normal Strength** 노드를 생성하고 **Sample Texture 2D** 노드의 **RGBA(4)** 출력을 입력으로 연결합니다. **Strength** 입력은 **Normal Strength** 프로퍼티로 설정합니다. **Add** 노드를 생성해 입력 A는 **Normal Strength** 노드의 출력으로 설정하고, 입력 B는 **World**를 **Tangent**로 바꾸는 **Transform** 노드의 출력으로 설정합니다. 이 단계를 따라 하면 다음과 같은 그래프가 결과로 나옵니다.



노멀 맵 추가



적절한 노멀 맵을 사용하면 아래 이미지와 같은 효과를 얻을 수 있습니다. 여기서는 다이아몬드 대신 단일 사각형을 사용했습니다. 평면 메시의 굴절은 불투명 텍스처의 오프셋을 표시합니다. 평면 메시지를 가진 노멀 맵을 사용하면 이러한 결함을 숨길 때 편리합니다.



노멀 맵 사용

Refract 노드를 사용하는 대신 Vector3 viewDir 입력, Vector3 normal 입력, IOR 입력이 있는 Custom Function 노드를 추가해도 됩니다. 이 방법을 사용하는 경우 IOR 프로퍼티의 슬라이더 범위를 1~6이 아닌 -0.15~2로 설정하고 Vector3를 출력으로 설정합니다. 매우 간단한 코드이므로 파일이 아닌 다음 문자열만 사용합니다.

```
Out = refract(viewDir, normal, IOR);
```

다양한 결과가 생성되는 코드이므로 실험해 볼 가치가 있습니다.

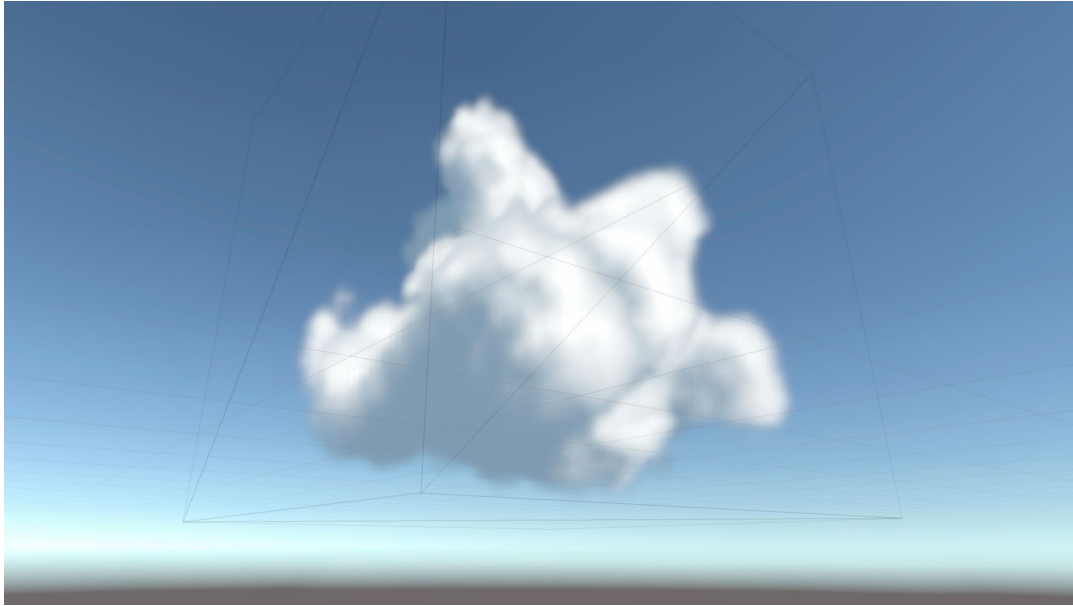
더 많은 리소스

- [스크린 공간 굴절](#)(영문) - 데이비드 레티어
- [스크린 공간 평면 반사](#)(영문) GitHub 저장소 - 스티븐 카나반
- [반사 프로브와 스크린 공간 반사 비교](#)(영문) - 카일 W. 파워즈
- [Shader Graph 굴절](#) 튜토리얼(영문) - AE Tuts
- [Unity의 크리스탈 Shader Graph](#)(영문) - Binary Lunar

볼류메트릭

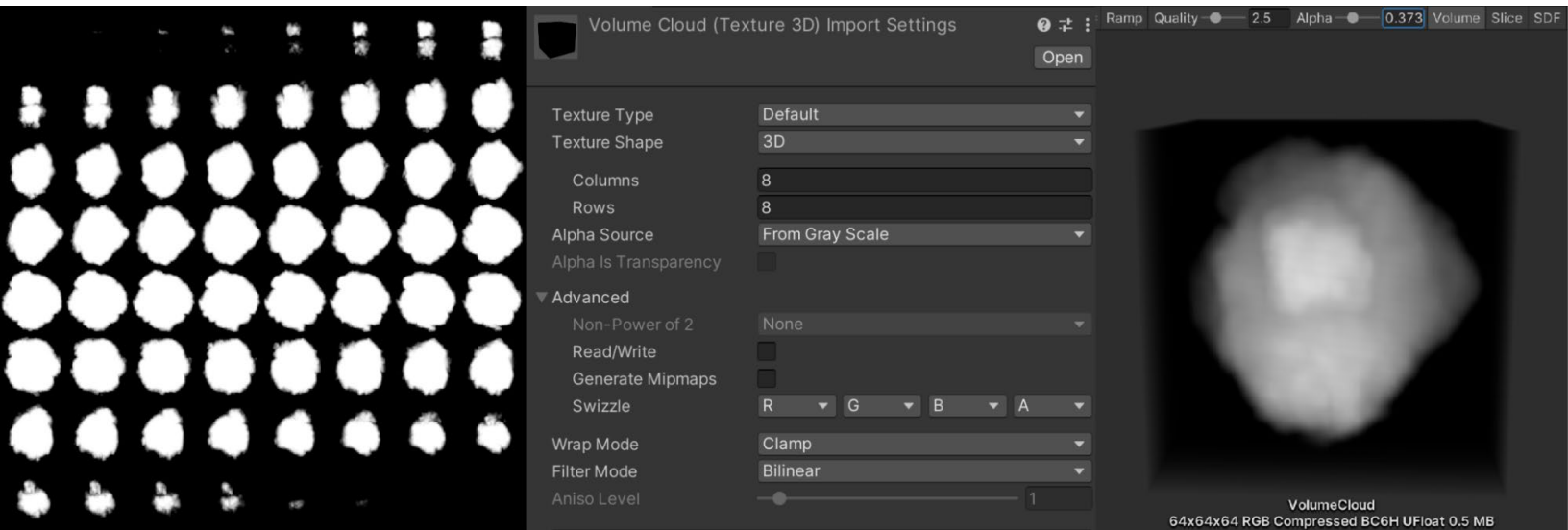


Zoink!에서 제작한 [Lost in Random](#)은 매우 독특한 아트를 선보이며 주사위 굴리기로 운명이 결정되는 랜덤 왕국으로 플레이어를 인도하는 판타지 게임입니다. 여기에서는 뛰어난 조명이 게임 분위기를 조성하는 데 중요한 역할을 합니다. [이 게시물](#)에 소개된 대로 Zoink!는 URP의 볼류메트릭 포그를 재현했습니다.

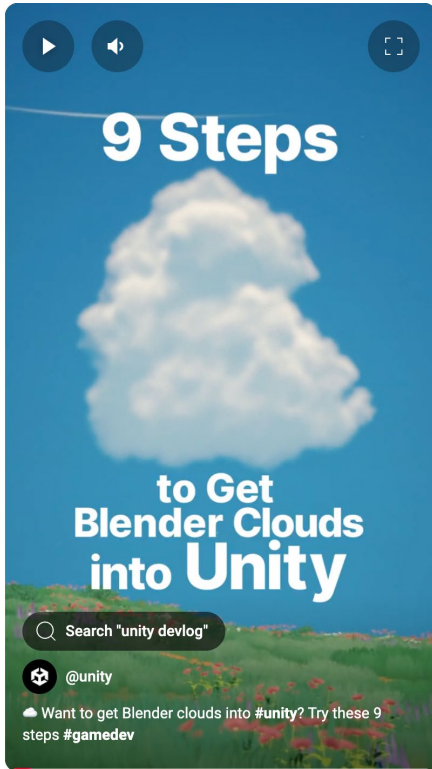


볼류메트릭 구름

이 레시피에서는 레이 마칭(ray marching)을 사용해 3D 텍스처를 렌더링하는 방법을 소개합니다. Unity는 단일 텍스처 위의 그리드에 배치된 이미지 배열로 텍스처 아틀라스와 유사한 3D 텍스처를 지원합니다. 차이점은 각 이미지의 크기가 같다는 점입니다. 3D UV 값을 사용하면, 사용할 개별 이미지의 행과 열을 정의하는 UV.Z를 사용해 이미지 그리드에서 텍셀을 찾을 수 있습니다. 아래 이미지는 일반적인 3D 텍스처, 텍스처의 임포트 설정, 인스펙터에서 프리뷰로 본 모습입니다.



왼쪽부터: 3D 텍스처, 텍스처의 임포트 설정, 인스펙터의 텍스처 프리뷰

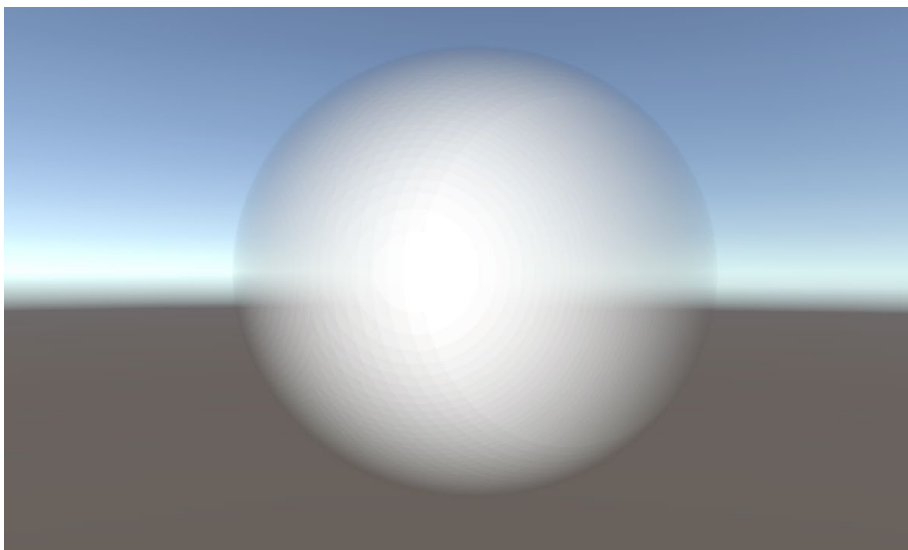


이 [YouTube Shorts](#)에서는 [Blender](#)를 사용하여 3D 구름 텍스처를 생성하는 방법을 설명합니다.

3D 구름 텍스처를 생성하는 방법을 설명하는 YouTube Shorts

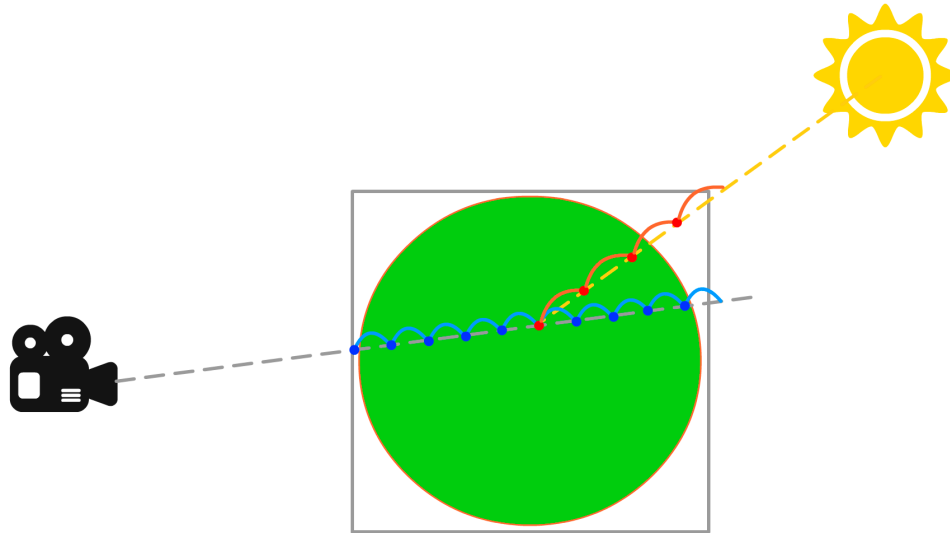
이전 레시피와 마찬가지로 이 셰이더는 Shader Graph를 사용해 제작합니다. 완성된 결과물을 보려면 **Scenes > Volumetric Clouds**로 이동해 **VolumetricClouds** 씬을 엽니다. 해당 씬에는 카메라, 방향 광원, 큐브가 있고, 큐브는 RaymarchMat 머티리얼을 사용합니다.

이 레시피를 시작하려면 닉 레버가 만든 **Shader Graphs/Raymarchv1SG** 셰이더를 RaymarchMat 머티리얼에 지정해야 합니다. 그러면 구체가 표시됩니다. **densityScale**을 조정하면 에지가 투명해지는 것을 볼 수 있습니다.



Shader Graph Raymarchv1SG 사용

큐브를 렌더링하려 하는데 구체가 나타나서 혼란스러울 수 있는데, 이는 [레이 마칭](#)을 사용하기 때문입니다. [Wikipedia 페이지](#)에 따르면 레이 마칭은 3D 컴퓨터 그래픽스에서 사용하는 렌더링 방법으로, 광선을 반복적으로 통과시켜(traverse) 각 광선을 더 작은 광선 세그먼트로 효과적으로 분할하고, 각 단계에서 일부 함수를 샘플링합니다. 이 함수는 볼륨 레이캐스팅을 위한 볼류메트릭 데이터, 표면의 가속 교차 지점을 찾기 위한 거리 필드 같은 정보를 인코딩할 수 있습니다.



레이 마칭

첫 번째 버전에서는 구체가 Vector4를 사용해 정의됩니다. XYZ는 오브젝트를 기준으로 구체의 위치를 정의하고 W는 반지름을 정의합니다. 각 픽셀에 대해 카메라에서 직접 나오는 광선의 방향을 계산합니다(위 다이어그램에서 회색 점선으로 표현). 밀도 값을 0으로 설정하고, 구체 내부의 각 파란색 점에서 계산하는 선을 따라 이동해 밀도에 작은 값을 더합니다. 광선이 구체를 통과하면 카메라에서 렌더링하는 픽셀로 뻗어 있는 선과 구체가 겹치는 부분이 어느 정도인지에 대한 값을 구할 수 있습니다. 이 밀도 값은 Shader Graph에서 Base Color로 사용됩니다. 지금은 태양과 빨간색 점은 무시해도 됩니다. 이 부분은 나중에 셰이더의 네 번째 버전에서 조명을 추가할 때 고려할 것입니다.

이 그래프는 **Scripts > HLSL > Raymarch.hls** 파일에 기반한 Custom Function 노드를 사용합니다. 첫 번째 버전에서는 raymarchv1 함수를 사용합니다. 변수 density는 0으로 초기화됩니다. 그런 다음 numSteps 카운트를 위한 for 루프를 입력합니다. rayOrigin은 rayDirection에 정의된 방향으로 stepSize만큼 이동합니다.

구체 원점으로부터 얼마나 멀리 떨어져 있는지 구하려면 HLSL 함수 거리를 사용해 구체 원점부터 현재 rayOrigin 값까지 벡터의 길이를 계산하면 됩니다. 이 값이 구체 반경(Sphere.w)보다 작으면, density 값에 0.1을 더합니다. 출력 값은 누적된 density 값에 densityScale을 곱한 값입니다.



```
void raymarchv1_float( float3 rayOrigin, float3 rayDirection, float numSteps,
                      float stepSize, float densityScale, float4 Sphere,
                      out float result )
{
    float density = 0;

    for(int i=0; i< numSteps; i++){
        rayOrigin += (rayDirection*stepSize);

        //density 계산
        float sphereDist = distance(rayOrigin, Sphere.xyz);

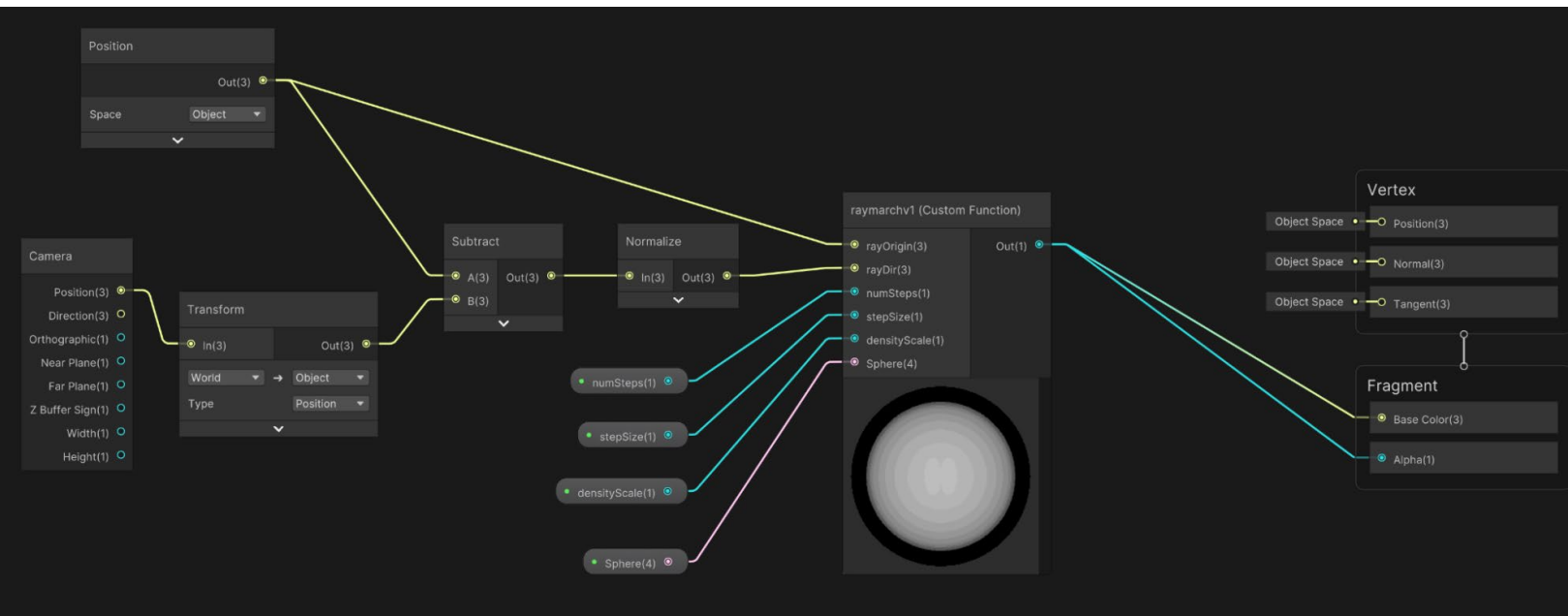
        if(sphereDist < Sphere.w){
            density += 0.1;
        }

    }

    result = density * densityScale;
}
```

계산은 오브젝트 공간에서 수행하게 됩니다. Position 노드를 사용해 rayOrigin을 구하고 rayDirection을 구하려면, [Camera](#) 노드에서 Position 출력을 [Transform](#) 노드에 연결하고 Transform 노드의 입력은 World로, 출력은 Object로 설정해야 합니다.

오브젝트 공간에서 픽셀 위치와 카메라 위치를 구했으니, [Subtract](#) 노드를 사용해 광선 방향을 구할 수 있습니다. 이때 Position을 입력 A에 연결하고 Camera의 Position을 입력 B에 연결합니다. 이 rayDirection은 Normalize 노드를 사용해 정규화됩니다. Custom Function 노드에 대한 다른 입력은 각각 플로트 프로퍼티인 numSteps, 광선별 파란색 점의 수인 stepSize, 파란색 점 사이의 거리인 densityScale, 그리고 앞서 설명한 Vector4로 정의한 Sphere입니다. 밀도 출력은 바로 Base Color와 Alpha로 연결하면 됩니다. 이 셰이더는 투명한 Unlit으로 설정되며 조명을 계산해야 합니다.



레이 마칭 셰이더의 첫 번째 버전

3D 텍스처를 추가해 셰이프를 결정하면 레이 마칭이 완성됩니다. 두 번째 버전에서는 3D 텍스처를 추가해 보겠습니다. 먼저 **Shader Graphs > Raymarch2SG**를 사용해 **RaymarchMat**을 설정합니다. 여기서 사용하는 커스텀 함수는 raymarchv2입니다.

```
void raymarchv2_float( float3 rayOrigin, float3 rayDirection, float numSteps,
    float stepSize, float densityScale, UnityTexture3D volumeTex,
    UnitySamplerState volumeSampler, float3 offset,
    out float result )
{
    float density = 0;
    float transmission = 0;

    for(int i=0; i< numSteps; i++){
        rayOrigin += (rayDirection*stepSize);

        //density 계산
        float sampledDensity = SAMPLE_TEXTURE3D(volumeTex, volumeSampler, rayOrigin +
offset).r;
        density += sampledDensity;
    }

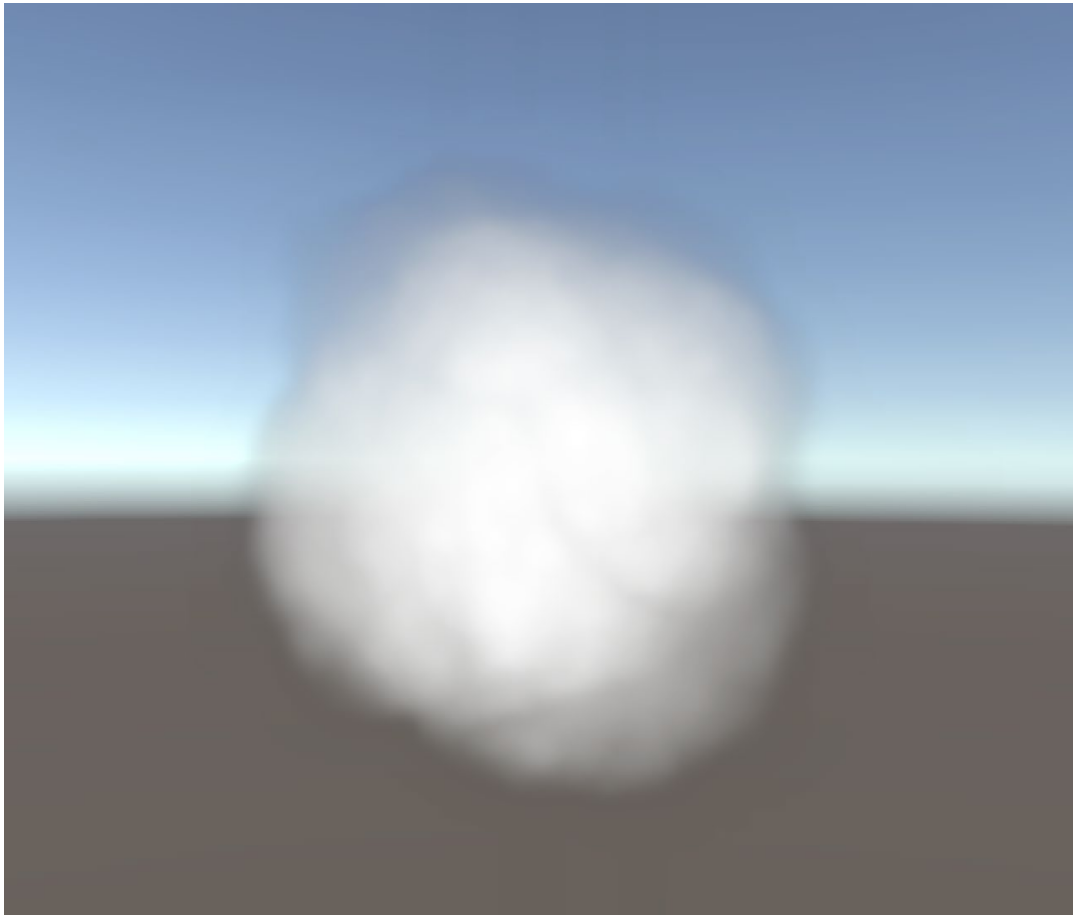
    result = density * densityScale;
}
```

세 가지 입력이 새로 추가된 것을 알 수 있습니다.

- 첫 번째는 Material 프로퍼티에서 직접 가져온 UnityTexture3D volumeTex입니다.
- 두 번째는 3D 텍스처를 사용할 때 필요한 매크로인 SAMPLE_TEXTURE3D로, SamplerState 인스턴스가 필요합니다.
 - SamplerState용 노드가 있어 래핑 옵션을 선택할 수 있습니다. 0~1 범위를 벗어난 UV 값 중에서 0보다 작은 값은 0으로, 1보다 큰 값은 1로 고정하도록 이 옵션을 설정합니다.
- 세 번째는 Offset으로, 큐브 내부의 3D 텍스처를 옮길 때 사용할 수 있는 값입니다.

이제 구체 내부에 있는지 확인하지 않고도 rayOrigin의 float3 샘플 위치에 오프셋을 더한 값을 사용해 sampledDensity 값을 구할 수 있습니다. 여기서는 빨간색인 R 채널만 있으면 됩니다.

아래 이미지는 두 번째 버전의 셰이더를 사용해 렌더링한 모습으로, 점점 구름처럼 보이기 시작합니다.



두 번째 버전 셰이더



셰이더의 최종 버전에는 조명이 추가됩니다. RaymarchMat 머티리얼에는 Shader Graphs/Raymarchv3SG라는 셰이더를 사용합니다. 이번에는 raymarch 함수를 사용합니다. 이 함수는 여섯 개의 새로운 파라미터 (**numLightSteps**, **lightStepSize**, **lightDir**, **lightAbsorb**, **transmittance**)를 사용하며 float3 벡터를 반환합니다.

최종 값을 얻으려면, 세 가지 새 변수(**transmission**, **lightAccumulation**, **finalLight**)를 초기화해야 합니다. 사용하는 코드는 광원 루프 주석까지 두 번째 버전의 코드와 동일합니다. 앞에서 본 '레이 마칭' 그림을 다시 살펴보면, 파란색 점으로 나타나는 뷰 방향 광선을 따라 단계마다 주 광원을 향하는 광선(다이아그램에서 노란색으로 표시)을 얻습니다. 빨간색 점은 3D 텍스처의 단계별 샘플링을 나타냅니다. 구름이 많을수록 뷰 방향 광선의 해당 부분에 닿는 광원이 줄어듭니다. 이 프로세스가 각 픽셀의 밝기를 결정합니다.

```
void raymarch_float( float3 rayOrigin, float3 rayDirection, float numSteps,
                    float stepSize, float densityScale, UnityTexture3D volumeTex,
                    UnitySamplerState volumeSampler, float3 offset,
                    float numLightSteps, float lightStepSize, float3 lightDir,
                    float lightAbsorb, float darknessThreshold, float transmittance,
                    out float3 result )
{
    float density = 0;
    float transmission = 0;
    float lightAccumulation = 0;
    float finalLight = 0;

    for(int i=0; i<numSteps; i++){
        rayOrigin += (rayDirection*stepSize);

        float3 samplePos = rayOrigin+offset;
        float sampledDensity =
            SAMPLE_TEXTURE3D(volumeTex, volumeSampler, samplePos).r;
        density += sampledDensity*densityScale;
        //광원 루프
        float3 lightRayOrigin = samplePos;

        for(int j = 0; j<numLightSteps; j++){
            lightRayOrigin += -lightDir*lightStepSize;
            float lightDensity =
                SAMPLE_TEXTURE3D(volumeTex, volumeSampler, lightRayOrigin).r;
            lightAccumulation += lightDensity;
        }
    }
}
```



```

float lightTransmission = exp(-lightAccumulation);
float shadow = darknessThreshold +
    lightTransmission * (1.0 - darknessThreshold);
finalLight += density * transmittance * shadow;
transmittance *= exp(-density * lightAbsorb);

}

transmission = exp(-density);

result = float3(finalLight, transmission, transmittance);

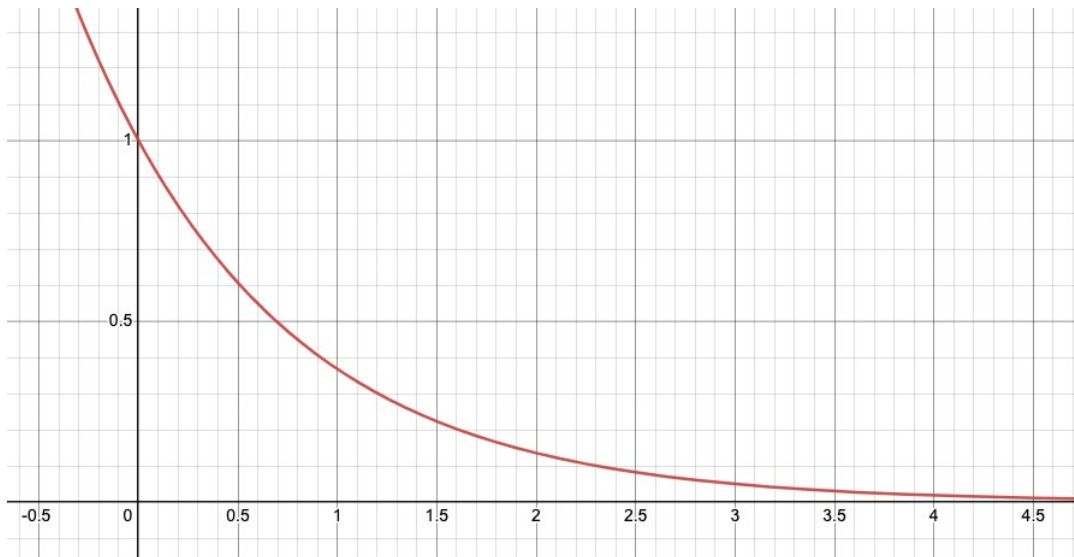
}

```

광원 루프는 **numLightSteps** 변수에 지정된 횟수를 반복하므로 쉽게 이해할 수 있습니다. 이 루프는 중첩 루프이므로 numLightSteps 카운트를 가능한 한 낮게 유지해야 합니다. 마이너스 lightDir을 사용해 samplePos에서 주 광원 쪽으로 이동합니다. 그런 다음 lightDensity를 lightAccumulation에 추가합니다. 광원 루프 외부에서는 약간의 계산이 필요합니다.

```
float lightTransmission = exp(-lightAccumulation);
```

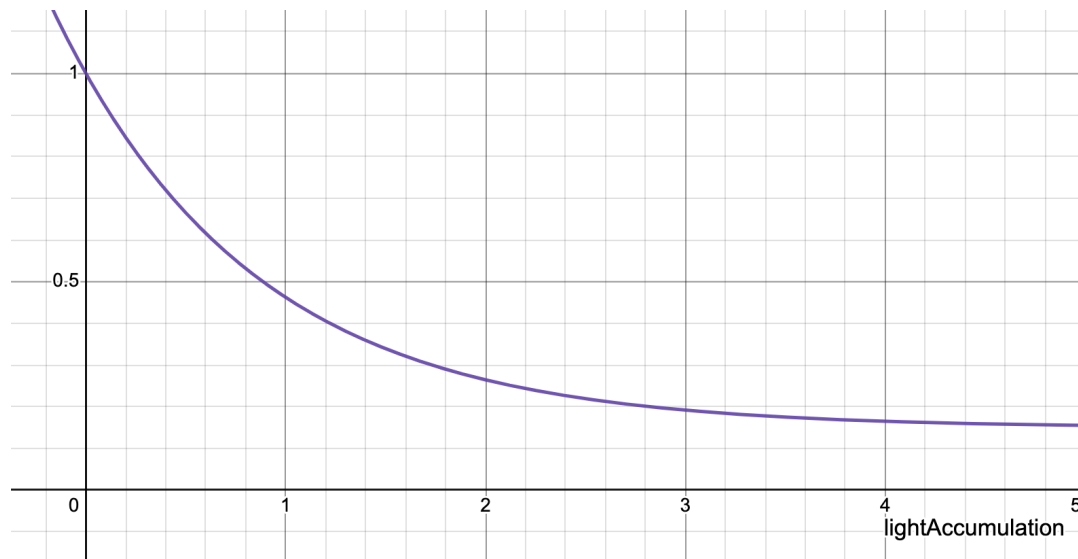
먼저 lightTransmission을 $e^{-\text{lightAccumulation}}$ 으로 설정합니다. 오일러 수인 **상수 e**는 약 2.718입니다. 아래 그래프는 이 함수의 결과를 표시합니다. 가로 축은 lightAccumulation 값이고 세로 축은 $\exp(-\text{lightAccumulation})$ 값입니다. 누적된 광원 밀도 lightAccumulation이 0이면, $\exp(-\text{lightAccumulation})$ 은 1이 됩니다. lightAccumulation이 증가함에 따라 $\exp(-\text{lightAccumulation})$ 값은 빠르게 감소하며, lightAccumulation이 5보다 크면 0에 수렴하게 됩니다.



0~4 범위에 대한 e^{-x} 그래프


```
float shadow = darknessThreshold +
    lightTransmission * (1.0 - darknessThreshold);
```

이제 shadow 값을 계산합니다. 여기서는 **darknessThreshold**라는 프로퍼티를 사용합니다. 아래 그래프에서는 darknessThreshold가 0.15인 경우의 shadow 값이 세로축에 표시됩니다. lightAccumulation이 0이면 shadow는 1이 되는 반면, lightAccumulation이 5에 가까워지면, shadow는 darknessThreshold 상수 값에 가까워집니다.



shadow 값

```
finalLight += density*transmittance*shadow;
```

Density * transmittance * shadow가 finalLight 누적 값에 추가됩니다. 누적된 광원 밀도인 lightAccumulation이 높으면, shadow가 0이 되는 경향이 있으므로, finalLight의 누적된 값은 작아집니다.

```
transmittance *= exp(-density*lightAbsorb);
```

transmittance의 초기 값은 전달된 프로퍼티이지만, 각 뷰 방향 단계마다 해당 값에 $e^{-\text{density} \times \text{lightAbsorb}}$ 를 곱합니다. **lightAbsorb** 프로퍼티는 구름에서 산란에 의해 손실되는 광원의 양을 제어합니다.

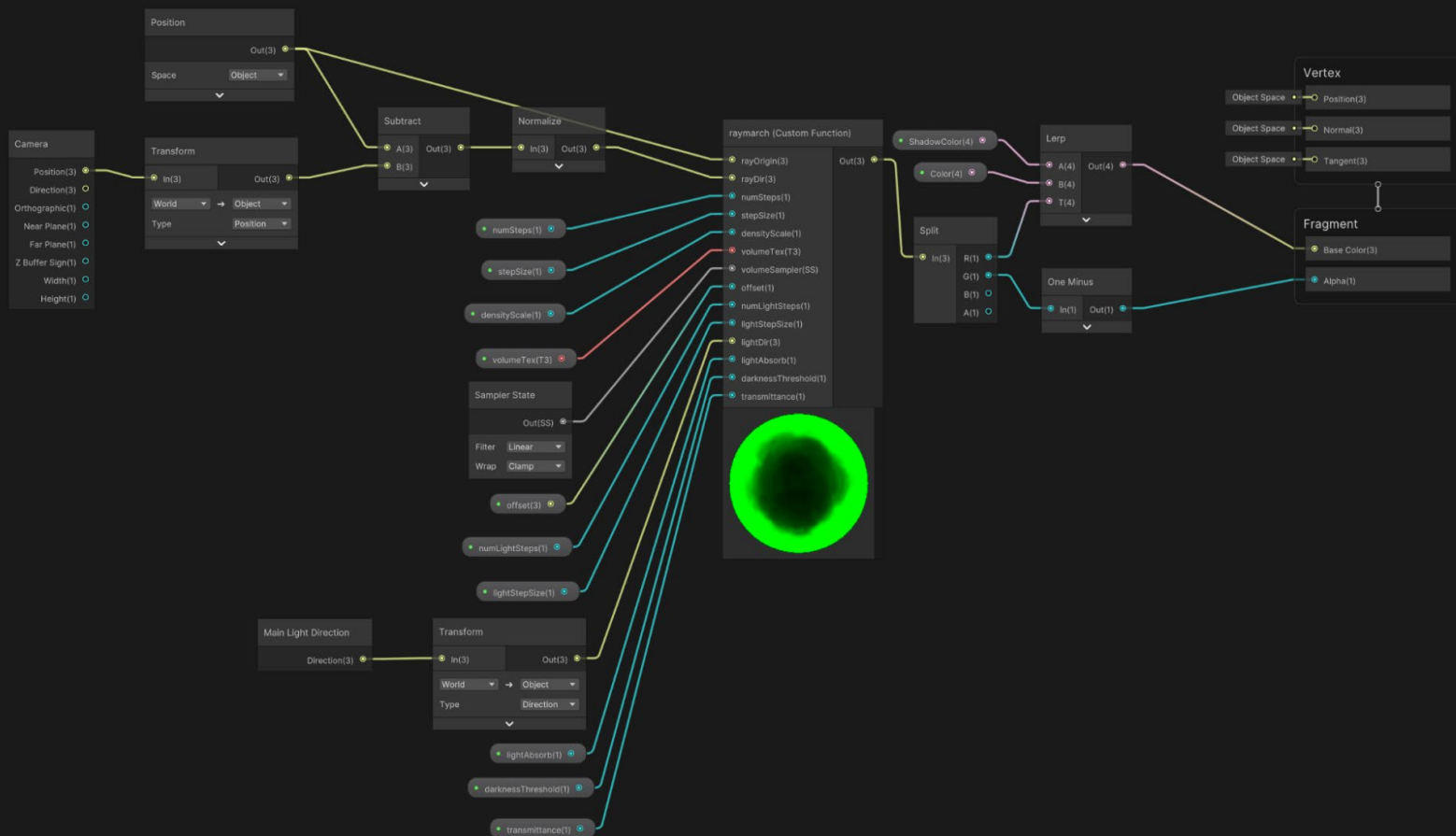
세 번째 버전의 경우 finalLight, transmission, transmittance를 포함하는 float3를 결과로 얻습니다.

세 번째 버전의 그래프는 아래와 같습니다. 이제 Custom Function 노드의 출력이 float3이므로 Split 노드를 추가해야 합니다. 출력 R은 Lerp 노드의 T 입력으로 연결됩니다. 세 번째 버전에는 **color**와 **shadowColor**를 비롯한 새 프로퍼티가 여러 개 있습니다. color는 B 입력으로, shadowColor는 A 입력으로 연결됩니다.



finalLight 레이 마치 노드 Out.x가 0이면 shadowColor가 Lerp 노드 출력으로 전달됩니다. finalLight가 1이면 color가 출력으로 전달됩니다. 0~1 범위에서 shadowColor와 color의 선형 보간이 출력되고, Lerp 노드 출력은 바로 Fragment > Base Color로 연결됩니다.

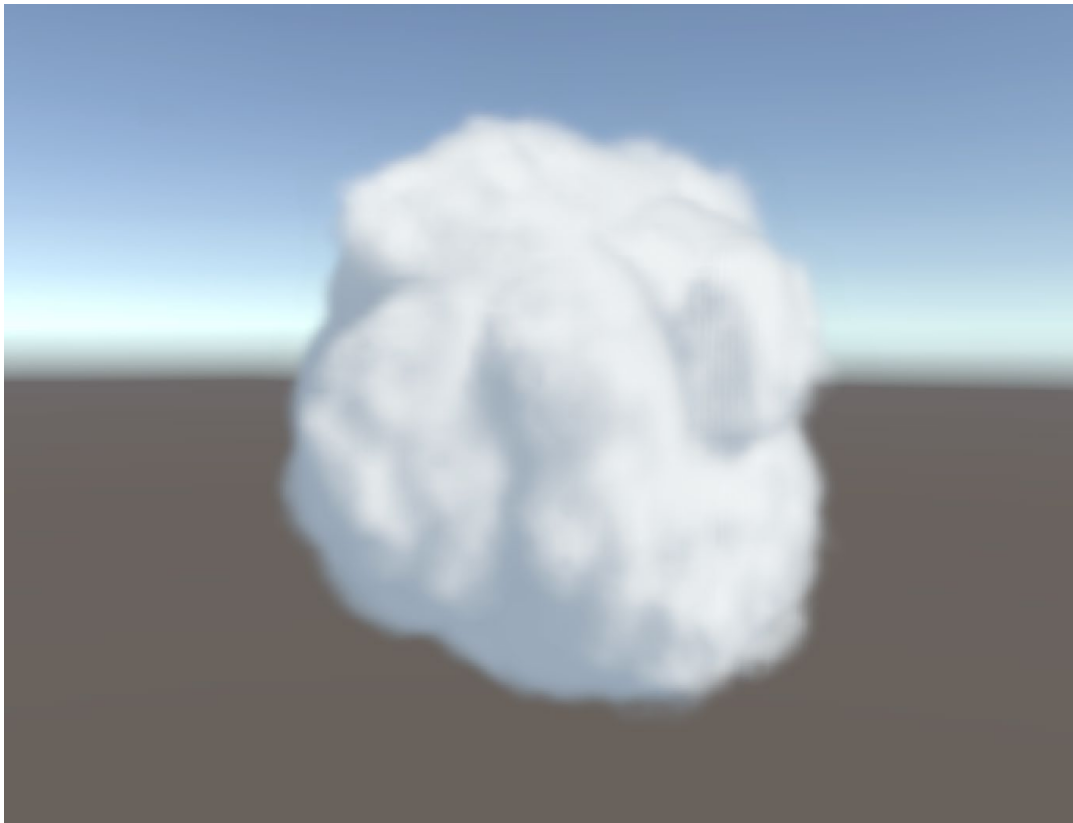
Alpha는 transmission 값 Out.y가 있는 레이 마치 노드를 사용합니다. 이 값은 Alpha가 1이어야 할 때 0이고, Alpha가 0이어야 할 때 1입니다. **One Minus** 노드를 사용해서 Split 노드의 B 값을 수정하고 이를 Fragment > Alpha로 연결합니다.



최종 버전



이 셰이더를 사용하면 아래와 같은 결과물을 얻을 수 있습니다.



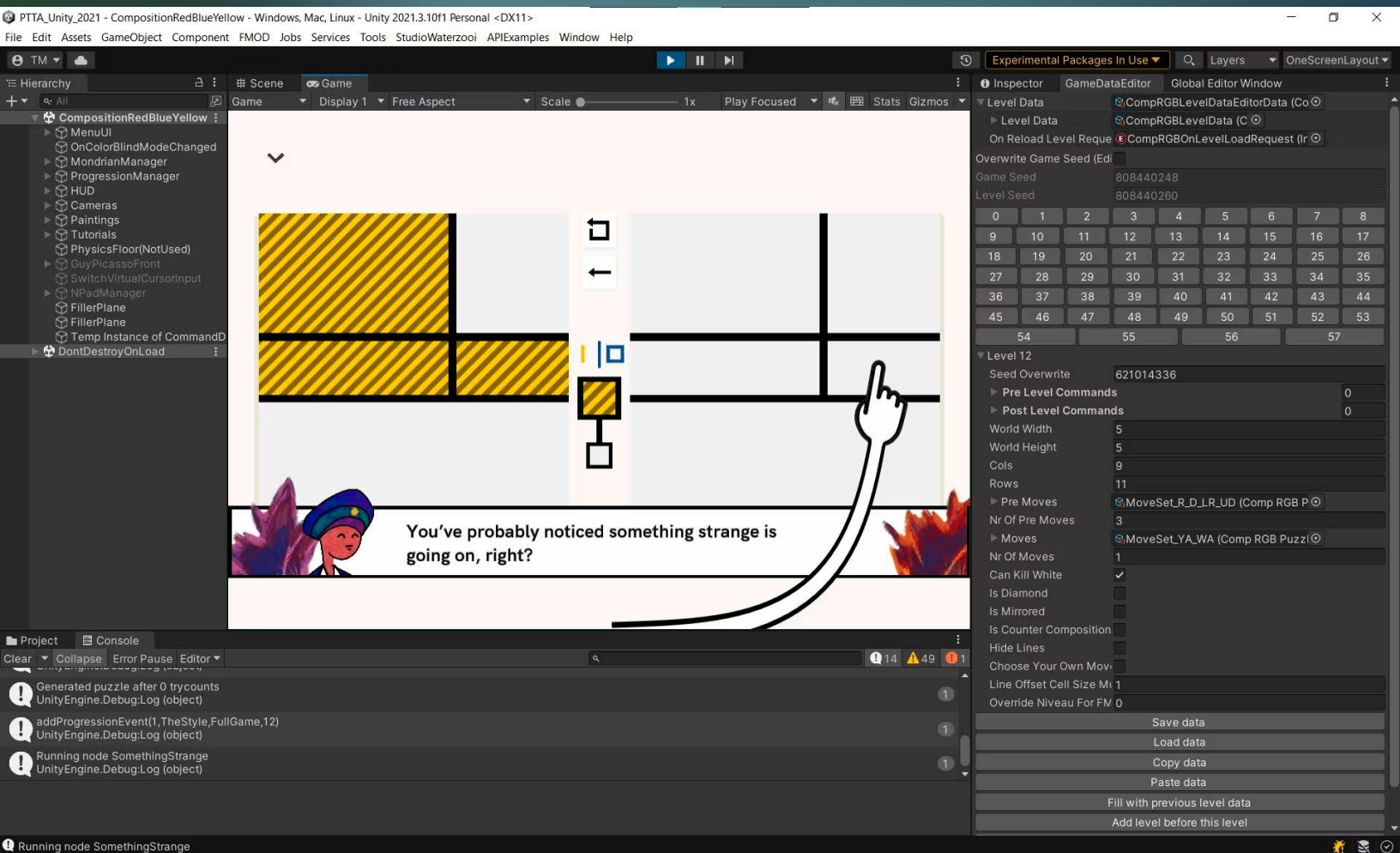
레이 마칭을 사용해 만든 구름

Houdini는 3D 텍스처를 만들 때 유용한 툴입니다. 3D 텍스처 대신 멀티 레이어 [Perlin 노이즈](#)를 사용하거나, Unity를 사용해 [타일러블 노이즈](#)(tileable noise) 텍스처를 미리 베이크해드 됩니다. 이 레시피를 시작으로 다양한 레이 마칭 활용법을 학습해 보세요.

더 많은 리소스

- [볼류메트릭 레이 마칭 구름 셰이더](#)(영문) - dmeville
- [코딩 어드벤처: 구름](#)(영문) - 세바스찬 라그
- [Houdini로 볼류메트릭 구름 만들기](#)(영문) - 카멜리아 슬리마니
- [Altos 하늘 시스템](#)(영문) - OccaSoftware
- [실시간 볼류메트릭 구름](#)(영문) - 에이드리언 폴리리메니

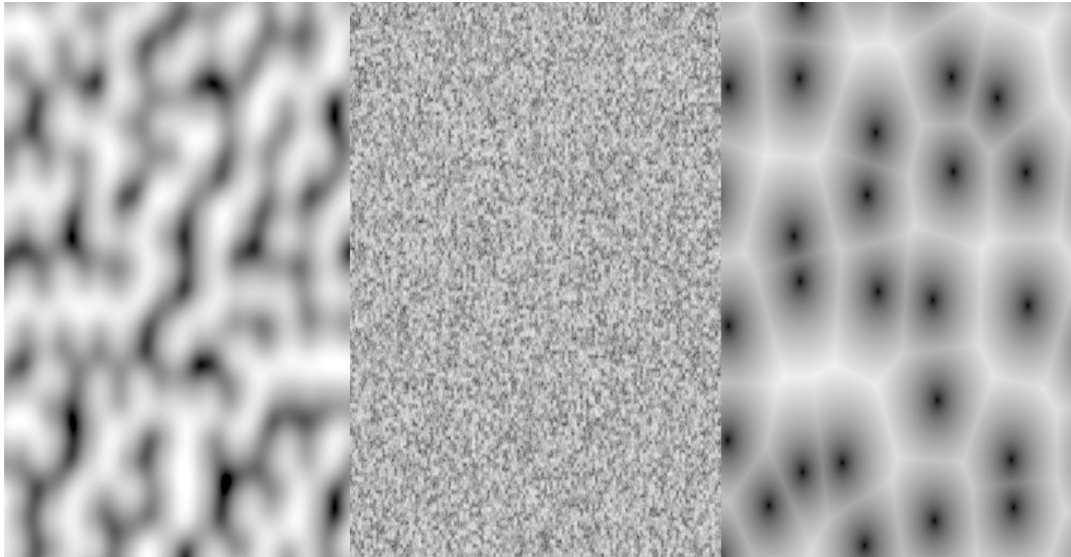
절차적 노이즈



토마스 워터홀의 [Please, Touch the Artwork](#)는 레벨 생성 프로세스에 소요되는 시간을 단축하기 위해 절차적 생성을 사용하는 게임으로, 게임 디자인 요소를 무작위화하는 좋은 사례입니다.

절차적 노이즈는 텍스처, 터레인 및 그 밖의 환경 요소를 무작위처럼 보이지만 정교하게 제어할 수 있는 방식으로 생성하는 강력한 기법입니다. 개발자는 알고리즘을 사용해 이러한 효과를 생성하여 모든 디테일을 수작업하지 않고도 방대하고 다양한 월드를 생성할 수 있습니다.

Unity는 Gradient/Perlin 노이즈, Simplex 노이즈, Voronoi/Worley와 같은 다양한 유형의 절차적 노이즈를 지원하며, 각 유형마다 장점과 활용처가 다릅니다. 절차적 노이즈를 이해하고 구현하면 게임 또는 인터랙티브 경험의 심미적인 수준과 성능을 향상할 수 있습니다.



절차적 노이즈의 예시. 왼쪽부터: Gradient, Simplex, Voronoi

절차적 노이즈의 유형

Unity에서 가장 일반적인 절차적 노이즈 유형은 Perlin 노이즈와 Simplex 노이즈로, 두 유형 모두 켄 펄린이 개발했습니다.

Perlin 노이즈는 텍스처, 터레인 및 애니메이션에서 매끄럽고 자연스러운 배리어이션을 생성할 때 가장 흔하게 사용되는 알고리즘입니다. 거친 에지 없이 매끄럽고 연속적인 패턴을 생성하므로 산악 터레인, 구름 텍스처, 파도와 같은 요소에 적합합니다. Simplex 노이즈는 더욱 효율적인 대안으로, 특히 3차원 요소에 적합합니다. 매끄러운 결과를 산출하며 Perlin 노이즈에 비해 연산량이 적어 성능이 중요한 모바일 게임이나 VR 환경에 유용합니다.

Unity에서 절차적 노이즈 구현하기

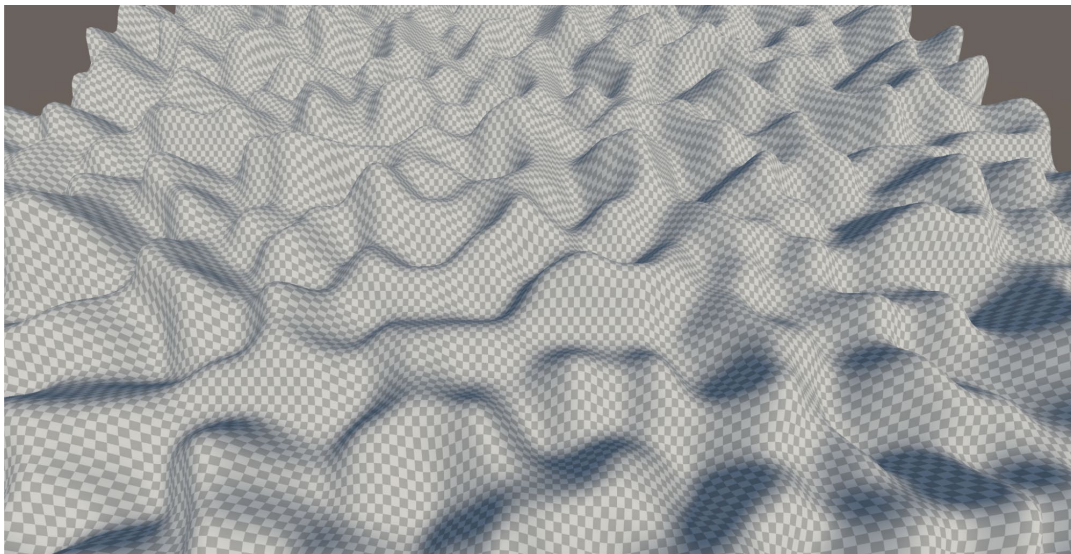
Unity에서 절차적 노이즈는 셰이더를 통해 구현하거나 C# 스크립트를 통해 구현할 수 있습니다. 2D 텍스처를 구현할 때는 `Mathf.PerlinNoise()` 함수가 특히 유용합니다. 이 함수는 X 좌표와 Y 좌표를 기준으로 노이즈 값을 생성하므로 터레인, 텍스처, 파티클 효과에 적용하기 좋습니다. 개발자는 주파수, 진폭, 스케일과 같은 파라미터로 노이즈 출력을 제어하여 다양한 시각적 효과를 얻을 수 있습니다.

예를 들어 Perlin 노이즈를 사용하여 터레인의 하이트맵을 생성하려면, Perlin 노이즈 값으로 구성된 그리드를 생성한 다음 그 값을 높이 레벨에 매핑하면 됩니다. 각 노이즈 값은 터레인의 특정 점에 대응되므로 언덕, 골짜기, 그 밖의 자연 지형을 수작업 없이 만들 수 있습니다. Unity Terrain Tools를 사용하면 이러한 하이트맵을 적용하여 3D 터레인을 생성한 다음, 텍스처를 적용하거나 기타 절차적 생성 기법을 사용하여 완성도를 높일 수 있습니다.

절차적 하이트맵 예시

다음은 C# 스크립트의 Perlin 노이즈를 사용하여 Unity에서 하이트맵을 생성하는 방법을 보여 주는 기본적인 예시입니다. 이 스크립트는 Terrain 게임 오브젝트에 Perline 노이즈를 적용하여 언덕과 골짜기가 있는 터레인을 생성합니다.

1. Unity에서 Terrain 게임 오브젝트를 생성합니다.
2. 아래 스크립트를 Terrain에 추가합니다.
3. 파라미터를 조정하여 터레인의 모습을 제어합니다.



절차적 하이트맵 예시

```
using Unity.VisualScripting;
using UnityEngine;

[ExecuteInEditMode]
public class ProceduralHeight : MonoBehaviour
{
    public int depth = 20; // 터레인의 최대 높이
    public float scale = 20f; // 노이즈가 얼마나 늘어나 보이는지를 제어
```



```
private int width = 256; // 터레인의 너비
private int height = 256; // 터레인의 높이
private float offsetX = 100f; // X 좌표의 오프셋(처음에는 무작위화)
private float offsetY = 100f; // Y 좌표의 오프셋(처음에는 무작위화)

private int _depth;
private float _scale;

private Terrain terrain;

void Start()
{
    // 매번 고유한 터레인이 생성되도록 오프셋 무작위화
    offsetX = Random.Range(0f, 9999f);
    offsetY = Random.Range(0f, 9999f);

    _depth = depth;
    _scale = scale;

    terrain = GetComponent<Terrain>();
    terrain.terrainData = GenerateTerrain(terrain.terrainData);
}

TerrainData GenerateTerrain(TerrainData terrainData)
{
    terrainData.heightmapResolution = width + 1;
    terrainData.size = new Vector3(width, depth, height);
    terrainData.SetHeights(0, 0, GenerateHeights());
    return terrainData;
}

float[,] GenerateHeights()
{
    float[,] heights = new float[width, height];
    for (int x = 0; x < width; x++)
    {
        for (int y = 0; y < height; y++)
        {
            float xCoord = (float)x / width * scale + offsetX;
            float yCoord = (float)y / height * scale + offsetY;
```



```
        heights[x, y] = Mathf.PerlinNoise(xCoord, yCoord);
    }
}
return heights;
}

void Update(){
    if ( depth != _depth || scale != _scale ){
        terrain.terrainData = GenerateTerrain(terrain.terrainData);
        _depth = depth;
        _scale = scale;
    }
}
}
```

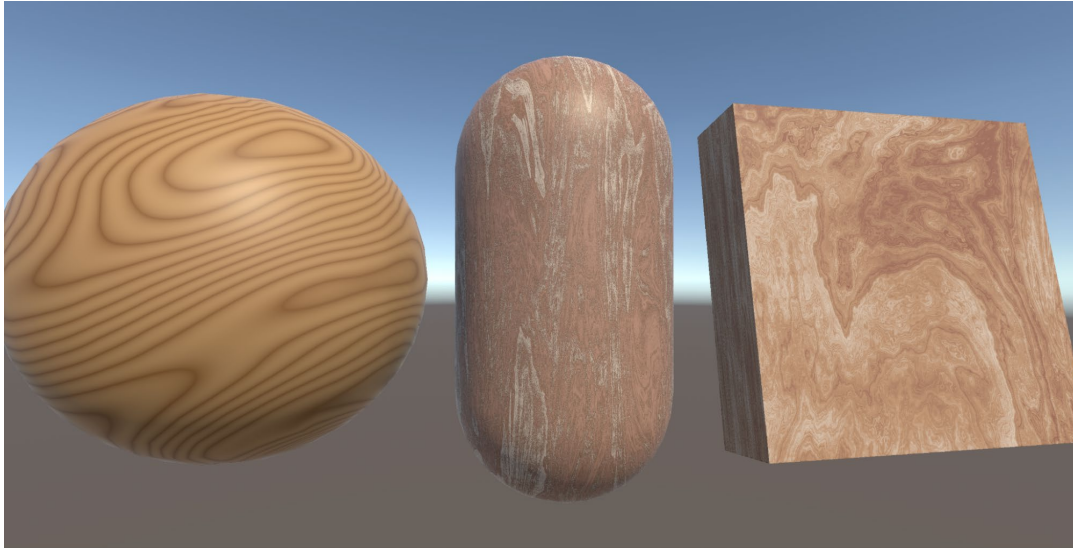
이 스크립트의 주요 값/프로퍼티를 살펴보겠습니다.

- **depth**: 터레인의 세로 방향 스케일로, 이 값이 클수록 더 높은 언덕이 생성됩니다.
- **width** 및 **height**: 터레인의 크기입니다.
- **scale**: 노이즈가 확대된 정도를 제어합니다. 값이 클수록 매끄럽고 넓은 언덕이 생성됩니다.
- **offsetX** 및 **offsetY**: 스크립트가 실행될 때마다 서로 다른 터레인 레이아웃이 생성되도록 추가된 무작위 오프셋입니다.

`GenerateHeights()` 함수는 터레인의 각 점의 높이 값을 나타내는 2D 배열을 생성합니다. 각 값은 `Mathf.PerlinNoise`를 사용하여 생성되며, `xCoord`와 `yCoord`를 기반으로 `width`, `height`, `scale` 값으로 스케일링된 매끄럽고 자연스러운 배리에이션을 구현합니다.

리소스 섹션의 **Scenes > Procedural Noise > Terrain**에서 이 예시를 확인할 수 있습니다. 하이트맵을 생성하는 이 단순한 기법은 더 복잡한 형태로 수정할 수 있습니다. 예를 들어 여러 레이어의 Perlin 노이즈를 적용하여 프랙탈(fractal)처럼 보이는 터레인을 생성하거나 추가적인 노이즈 함수를 적용할 수 있습니다.

노이즈를 사용하여 목재 텍스처 생성

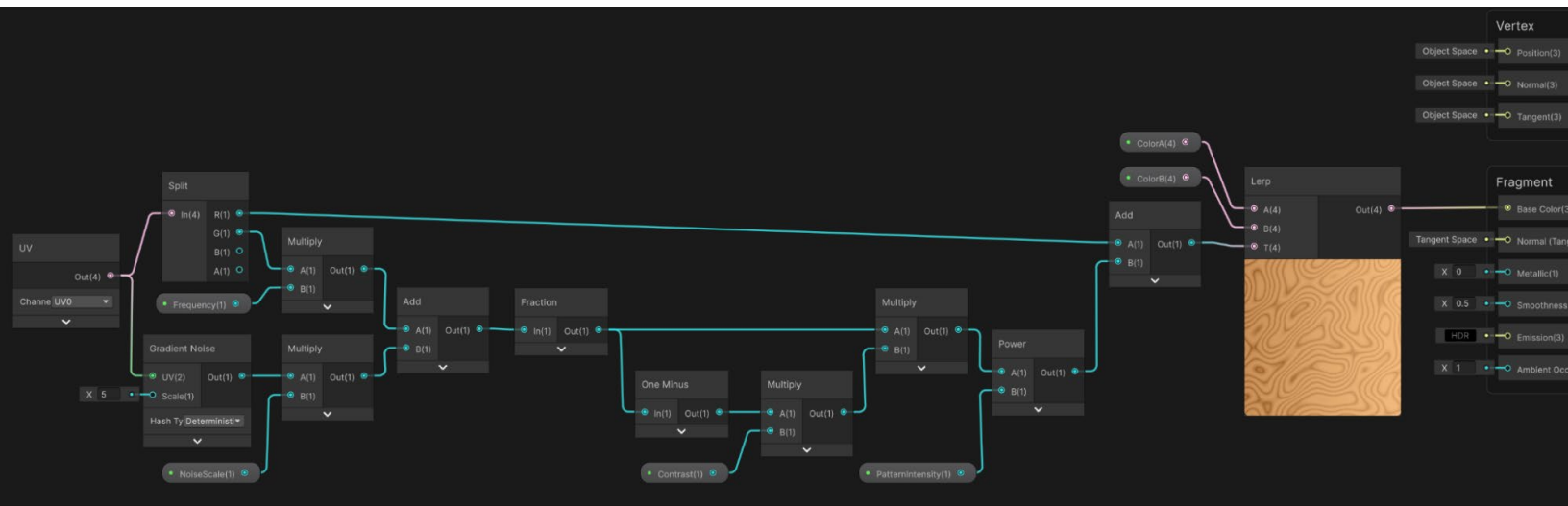


목재 텍스처에 대해 절차적 노이즈를 사용한 예시

절차적 노이즈의 또 다른 용도는 위 그림에 나와 있는 목재 텍스처와 같은 자연적인 텍스처를 반영하는 셰이더를 만드는 것입니다. 이 예시는 **Scenes > Procedural Noise > Wood**에서 확인할 수 있습니다. 여기서 스타일이 가미된 셰이더와 사실적인 셰이더, 이렇게 두 가지의 목재 셰이더를 볼 수 있습니다. 먼저 스타일이 가미된 버전을 만들어 보겠습니다.

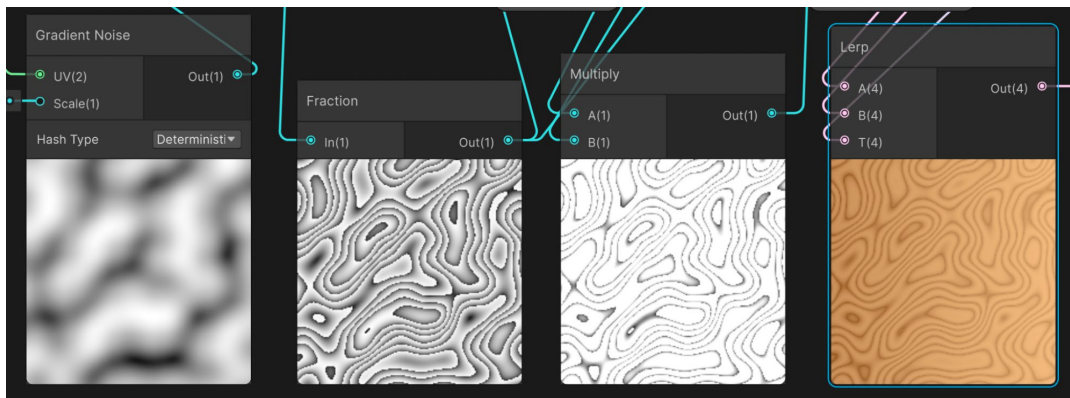
1. **Shader Graph > URP > Lit Shader Graph**를 통해 Shader Graph를 생성합니다.
2. 다음 프로퍼티를 추가합니다.
 - a. ColorA(Color 기본값: 7D490B)
 - b. ColorB(Color 기본값: BB905D)
 - c. Frequency(Float 기본값: 2)
 - d. NoiseScale(Float 기본값: 6)
 - e. PatternIntensity(Float 기본값: 0.6)
 - f. Contrast(Float 기본값: 4)
3. **Geometry > UV** 노드, Channel UV0을 만듭니다.
4. **Procedural > Noise > GradientNoise** 노드를 만듭니다. 3단계의 출력을 **UV** 입력에 연결하고 **Scale** 입력을 5로 설정합니다. **Hash Type**을 **Deterministic**으로 둡니다.
5. **Multiply** 노드를 만들고 입력 **A**를 4단계의 출력으로 설정하고 입력 **B**를 NoiseScale 프로퍼티로 설정합니다.
6. **Add** 노드를 만들고 5단계의 출력을 입력 **B**에 연결합니다.
7. 3단계의 **UV** 노드로 돌아갑니다. **Gradient Noise** 노드 위에 **Split** 노드를 만들고 그 입력에 **UV** 노드의 출력을 연결합니다.

8. **Multiply** 노드를 만들고 Split 노드의 **G** 출력을 입력 **A**로 설정하고 Frequency 프로퍼티를 입력 **B**로 설정합니다.
9. 8단계의 출력을 6단계의 입력 **A**에 연결합니다.
10. **Fraction** 노드를 만들고 그 입력에 **Add** 노드의 출력을 연결합니다.
11. **One Minus** 노드를 만들고 그 입력에 10단계의 출력을 연결합니다.
12. **Multiply** 노드를 만들고 11단계의 출력을 입력 **A**에 연결하고 Contrast 프로퍼티를 입력 **B**에 연결합니다.
13. 또 다른 **Multiply** 노드를 만들고 입력 **A**에 10단계의 출력을 연결하고 입력 **B**에 12단계의 출력을 연결합니다.
14. **Power** 노드를 만들고 입력 **A**에 13단계의 출력을 연결하고 입력 **B**에 PatternIntensity 프로퍼티를 연결합니다.
15. **Add** 노드를 만들고 입력 **A**에 7단계 Split 노드의 **R** 출력을 연결하고 입력 **B**에 14단계의 출력을 연결합니다.
16. **Lerp** 노드를 만들고 입력 **A**를 ColorA 프로퍼티로 설정하고 입력 **B**를 ColorB 프로퍼티로 설정하고 입력 **T**를 15단계의 출력으로 설정합니다.
17. 16단계의 출력을 **Fragment Base Color**에 연결합니다.



스타일이 가미된 목재 Shader Graph

UV **u** 값은 최종 Add 노드로 직접 전달되는 그레인 위치를 설정하는 데 사용됩니다. **v** 값은 복잡한 일련의 조작을 거칩니다. Gradient Noise를 통과한 후 여러 개의 Multiply 노드를 통과하며 그레인 라인(패턴)과 스케일이 조정됩니다. 이 셰이더의 핵심은 계산된 값의 소수 부분을 Fraction 노드를 통해 사용하는 것입니다. 프로퍼티를 사용하여 그레인 또는 패턴의 콘트라스트와 강도를 조정할 수도 있습니다.



세이더의 단계: Gradient Noise(맨 왼쪽), Fraction 부분을 사용하여 여러 개의 그레이 선을 적용한 효과(왼쪽에서 두 번째), 콘트라스트 증가(오른쪽에서 두 번째), 컬러 추가(맨 오른쪽)

Custom Function 노드를 사용하여 구현한 사실적인 버전을 살펴보겠습니다. 이 셰이더는 [ShaderToy의 이 예시](#)를 수정한 버전입니다. ShaderToy는 셰이더 코드를 찾아보기에 좋은 소스이지만, 모든 코드가 HLSL이 아닌 GLSL 구문을 사용하므로 Shader Graph Custom Function에서는 수정을 거쳐 사용해야 합니다. HLSL 코딩이 익숙하다면 `vec` -> `float`, `fract` -> `frac`, `mix` -> `lerp`와 같이 어느 정도 직관적인 방법으로 전환할 수 있습니다.

Shadertoy

Search...

BrowseNewSign In

MII40.2160.1 fps840 x 473REC

Procedural Wood texture

<>❤️63

Views: 1716, Tags: procedural, texture, wood, material, nature
Created by dean_the_coder in 2023-03-07

I spent some time working on some noise functions and then used them to make a wood texture. I'd never used musgrave noise before - Looks very cool.
I've tried to keep the code size down - Let me know if you find this useful!

Comments (11)

Sign in to post a comment.

TimelordQ, 2023-08-07
Amazing work. Thank you!

dean_the_coder, 2023-03-09
Thanks everyone!
@Shane My local version of this returns a vec4 where the w component is a 'depth' value. I left it out here for simplicity ('an exercise for the user'), but can easily be added by combining n1, n2, and grain.

Shane, 2023-03-09
Nicely done. I have a few simple go-to timber texture routines that I use (using noise and smooth fract), but they're not as detailed as this. One thing that I've noticed is that a lot of textures already come prelit (probably using directional derivative lighting) which can make them pop a bit more. I've been too lazy to generate my own textures lately, but when I do, I usually find that I have to provide a less detailed version for bump mapping and distance functions.

+Image

Shader Inputs

```
// 'Procedural Wood texture' dean_the_coder (Twitter: @deanthecoder)
// https://www.shadertoy.com/view/mdy3R1
//
// Processed by 'GLSL Shader Shrinker'
// (https://github.com/deanthecoder/GLSLShaderShrinker)
//
// I spent some time working on some noise functions and then
// used them to make a wood texture. I'd never used musgrave
// noise before - Looks very useful!
//
// Thanks to Evvvvil, Flopine, Nusan, BigWings, Iq, Shane,
// totetmatt, Blackle, Dave Hoskins, byt3_m3chanic, tater,
// and a bunch of others for sharing their time and knowledge!
// License: Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License

#define R iResolution
#define sat(x) clamp(x, 0., 1.)
#define S(a, b, c) smoothstep(a, b, c)
#define S01(a) S(0., 1., a)

float sum2(vec2 v) { return dot(v, vec2(1)); }

///////////////////////////////////////

float h31(vec3 p3) {
    p3 = fract(p3 * .1031);
    p3 += dot(p3, p3.yzx + 333.3456);
    return fract(sum2(p3.xy) * p3.z);
}

float h21(vec2 p) { return h31(p.yyx); }

float n31(vec3 p) {
    const vec3 s = vec3(7, 157, 113);

    // Thanks Shane - https://www.shadertoy.com/view/lstGRB
    vec3 ip = floor(p);
    p = fract(p);
    p = p * p * (3. - 2. * p);
    vec4 h = vec4(0, s.yz, sum2(s.yz)) + dot(ip, s);
    h = mix(fract(sin(h) * 43758.545), fract(sin(h + s.x) * 43758.545), p.x);
    h.xy = mix(h.xz, h.yw, p.y);
    return mix(h.x, h.y, p.z);
}
```

Compiled in 0.1 secs1879 chars

iChannel0

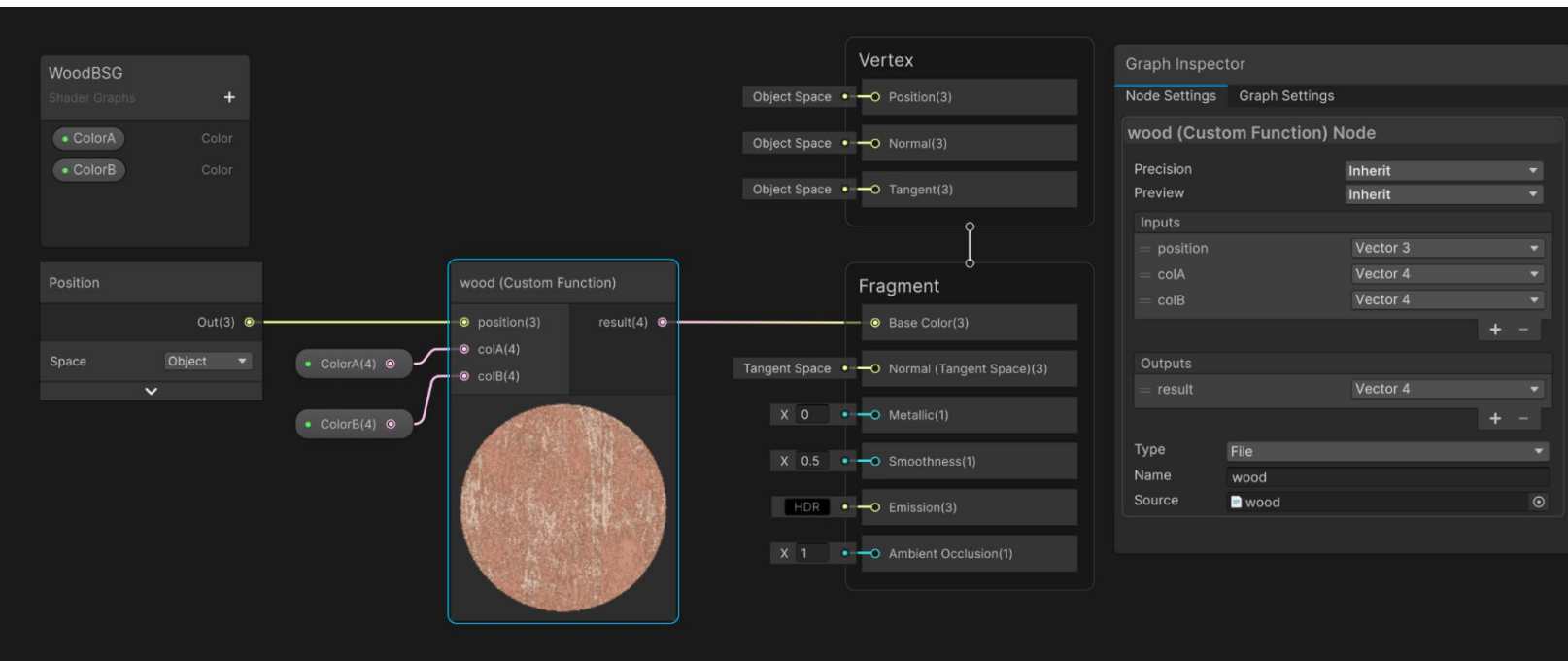
iChannel1

iChannel2

iChannel3

ShaderToy 웹사이트의 셰이더 예시

아래 코드에 나와 있는 Custom Function의 경우, 기본 컬러에 만족한다면 **Geometry > Position** 노드를 **Object**로 설정하기만 하면 됩니다. 또는 두 개의 **Color** 프로퍼티(그레인 선 컬러를 정의하는 ColorA와 베이스 컬러를 정의하는 ColorB)를 추가하여 컬러를 정의할 수 있습니다.



사용자 정의 컬러를 사용하는 Custom Function 목재 Shader Graph

```
#define sat(x)      clamp(x, 0.0, 1.0)
#define S(a, b, c)  smoothstep(a, b, c)
#define S01(a)      S(0.0, 1.0, a)

float sum2(float2 v) {return dot(v, float2(1.0, 1.0)); }

////////////////////////////////////

float h31(float3 p3) {
    p3 = frac(p3 * 0.1031);
    p3 += dot(p3, p3.yzx + 333.3456);
    return frac(sum2(p3.xy) * p3.z);
}

float h21(float2 p) {return h31(p.xyx); }

float n31(float3 p) {
    const float3 s=float3(7, 157, 113);
```



```
// 출처: Shane - https://www.shadertoy.com/view/1stGRB
float3 ip = floor(p);
p = frac(p);
p = p * p * (3. - 2. * p);
float4 h = float4(0, s.yz, sum2(s.yz)) + dot(ip, s);
h = lerp(frac(sin(h) * 43758.545), frac(sin(h + s.x) * 43758.545), p.x);
h.xy = lerp(h.xz, h.yw, p.y);
return lerp(h.x, h.y, p.z);
}

// 거칠기: (0.0, 1.0], 기본값: 0.5
// 부호 없는 노이즈 [0.0, 1.0] 반환
float fbm(float3 p, int octaves, float roughness) {
    float sum = 0.,
          amp = 1.,
          tot = 0.;
    roughness = sat(roughness);
    for (int i = 0; i < octaves; i++) {
        sum += amp * n31(p);
        tot += amp;
        amp *= roughness;
        p *= 2.;
    }
    return sum / tot;
}

float3 randomPos(float seed) {
    float4 s = float4(seed, 0, 1, 2);
    return float3(h21(s.xy), h21(s.xz), h21(s.xw)) * 1e2 + 1e2;
}

// 부호 없는 노이즈 [0.0, 1.0] 반환
float fbmDistorted(float3 p) {
    p += (float3(n31(p + randomPos(0.)), n31(p + randomPos(1.)), n31(p + randomPos(2.))) * 2. -
1.) * 1.12;
    return fbm(p, 8, .5);
}

// float3: 디테일(/octaves), 크기(/inverse contrast), lacunarity
// 부호 있는 노이즈 반환
float musgraveFbm(float3 p, float octaves, float dimension, float lacunarity)
{

```



```
float sum = 0.,
      amp = 1.,
      m = pow(lacunarity, -dimension);
for (float i = 0.; i < octaves; i++) {
    float n = n31(p) * 2. - 1.;
    sum += n * amp;
    amp *= m;
    p *= lacunarity;
}

return sum;
}

// X 축의 파형 노이즈
float3 waveFbmX(float3 p) {
    float n = p.x * 20.;
    n += .4 * fbm(p * 3., 3, 3.);
    return float3(sin(n) * .5 + .5, p.yz);
}

/////////////////////////////////////////////////////////////////
// 수식
float remap01(float f, float in1, float in2) { return sat((f - in1) / (in2 - in1)); }

/////////////////////////////////////////////////////////////////
// 목재 머티리얼
float3 matWood(float3 p, float3 colA, float3 colB) {
    float n1 = fbmDistorted(p * float3(7.8, 1.17, 1.17));
    n1 = lerp(n1, 1.0, 0.2);
    float n2 = lerp(musgraveFbm(float3(n1, n1, n1) * 4.6, 8.0, 0.0, 2.5), n1, 0.85);
    float dirt = 1. - musgraveFbm(waveFbmX(p * float3(.01, .15, .15)), 15., .26, 2.4) * .4;
    float grain = 1. - S(.2, 1., musgraveFbm(p * float3(500, 6, 1), 2., 2., 2.5)) * .2;
    n2 *= dirt * grain;

    // 3개의 float3 값은 RGB 목재 컬러 - 원하는 대로 미세 조정
    return lerp(lerp(colA, colB, remap01(n2, .19, .56)), float3(.52, .32, .19), remap01(n2, .56, 1.));
}
```




```
//하드 코딩된 컬러
void wood_float( float3 pos, out float4 result){
    float3 colA = float3(.03, .012, .003);
    float3 colB = float3(.25, .11, .04);
    result = float4(pow(matWood(pos, colA, colB), float3(1,1,1) * 0.4545), 0);
}

//사용자 선택 가능한 컬러, colA는 그레인 선
void wood_float( float3 pos, float3 colA, float3 colB, out float4 result){
    result = float4(pow(matWood(pos, colA, colB), float3(1,1,1) * 0.4545), 0);
}
```

이 코드는 wood_float 함수를 시작으로 matWood 함수를 호출합니다. 이 함수는 fbmDistorted, musgraveFbm, waveFbm 함수를 사용합니다. 코드에는 개발자가 최상의 결과를 얻고자 심혈을 기울여 선택한 수치가 적용되어 있습니다. **fBm**(Fractional Brownian motion 또는 멀티프랙탈 노이즈)은 절차적 노이즈의 하나로, [켄 머스그레이브](#)가 개발하여 Musgrave 노이즈라고도 불립니다. 사실적인 텍스처와 터레인을 생성하는 데 사용됩니다. Musgrave 노이즈는 Perlin 노이즈와 같은 기존 노이즈 함수와 달리 주파수와 진폭이 서로 다른 여러 개의 레이어(옥타브)를 결합하여 구름, 산, 흐르는 물과 같은 복잡한 자연 현상을 시뮬레이션합니다.

파라미터를 시작으로 musgraveFbm 함수에 대해 자세히 살펴보겠습니다.

- **P (float3)**: 위치 값으로, 주로 오브젝트 공간에 있습니다.
- **octaves (float)**: Musgrave 노이즈는 여러 옥타브의 노이즈 함수를 중첩하는 식으로 작동합니다. 그 결과 복합 효과가 생성되며, 레이어가 더해짐에 따라 디테일이 추가되어 텍스처가 갈수록 복잡해집니다.
- **dimension (float)**: 값의 반비례로 콘트라스트를 제어합니다.
- **lacunarity**: 옥타브 간의 주파수 비율을 제어합니다. lacunarity 값이 클수록 각 옥타브는 작고 세밀해지며, 따라서 더 정교한 패턴이 만들어집니다.

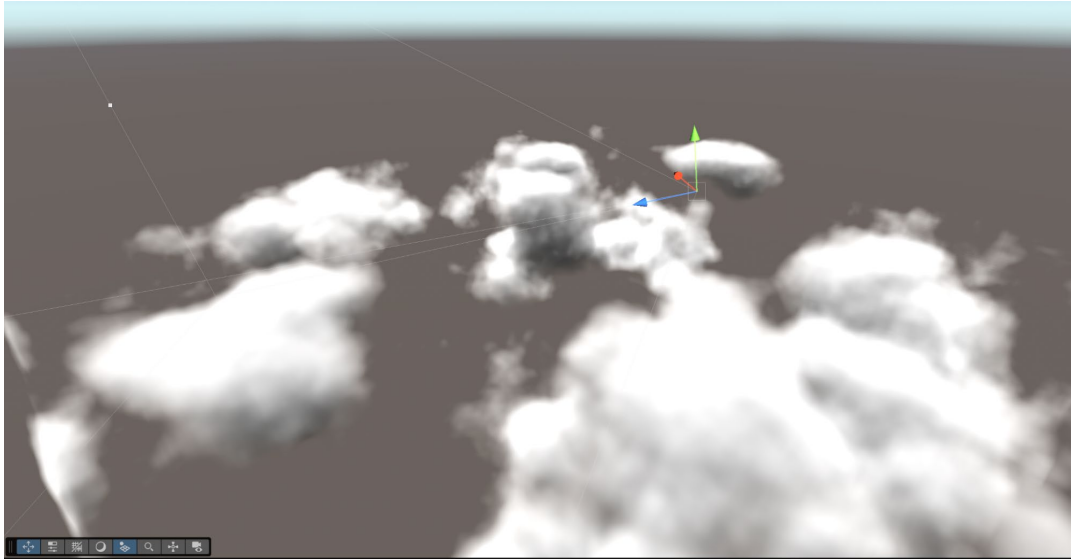
이 함수는 다음과 같은 3개의 플롯 값을 초기화합니다.

1. 반환된 sum 값
2. 진폭 값 amp
3. 멀티플라이어 m

멀티플라이어는 lacunarity를 음의 dimension으로 거듭제곱한 값입니다. 그런 다음 각 옥타브에 대한 루프를 실행합니다. n31 함수는 float3 입력을 받아서 Perlin 노이즈 값을 반환합니다. 이 값은 범위 0~1 사이에 있으므로 범위 -1~1로 다시 매핑한 다음 진폭 값 amp를 곱한 후 누적 sum 값에 더합니다. 그런 다음

amp 값에 상수인 멀티플라이어 값 m을 곱합니다. 마지막으로, 샘플 위치 p에 lacunarity 파라미터를 곱합니다.

Musgrave 노이즈는 최소한의 수동 입력만으로 사실적이고 자연적인 텍스처를 생성할 수 있기 때문에 컴퓨터 그래픽스에서 매우 중요하게 사용됩니다. 다양한 노이즈 함수를 중첩하고 변화시킴으로써 실제 환경, 구름, 그리고 위의 목재 예시와 같은 유기적인 머티리얼의 복잡성을 반영하여 정교하면서도 프랙탈 같은 패턴을 생성할 수 있습니다.



Musgrave 노이즈를 사용하여 레이 마칭 방식으로 구현한 구름

절차적 노이즈의 장점

셰이더에서 절차적 노이즈는 애니메이션 텍스처 또는 동적 머티리얼을 만들 때도 사용할 수 있습니다. 예를 들어 물 셰이더에 노이즈를 더하면 사실적인 물결 효과를 만들 수 있습니다. 시간의 흐름에 따라 노이즈에 애니메이션을 적용하면 물 표면이 움직이는 것처럼 표현되어 유동적이고 사실적인 느낌이 추가됩니다. 절차적 노이즈를 컬러 그레디언트 및 투명도 효과와 결합하여 구름, 흔들리는 잔디 등 다양하고 동적인 머티리얼을 만들 수 있습니다.

절차적 노이즈는 주로 확장성과 다양성을 비롯한 여러 장점이 있습니다. 규모가 크고 다양한 환경을 수작업으로 만들려면 긴 시간이 걸리고 많은 메모리가 필요합니다. 절차적 노이즈를 사용하면 스토리지 공간을 많이 사용하지 않아도 고유하고 일관되며 복잡한 환경을 만들 수 있습니다. 또한 절차적 노이즈 알고리즘은 결정론적이므로 같은 시드 값을 사용하면 같은 ‘무작위’ 환경을 다시 만들 수 있습니다.

이 기법으로 무한한 환경도 만들 수 있어, 오픈 월드 게임이나 생존 게임처럼 연속되는 터레인이 필요한 게임에 유용합니다. Unity는 이처럼 환경을 절차적으로 생성하여 터레인의 각 섹션을 필요에 따라 로드하거나 언로드하고, 메모리 사용량을 줄이고, 성능을 개선할 수 있습니다.

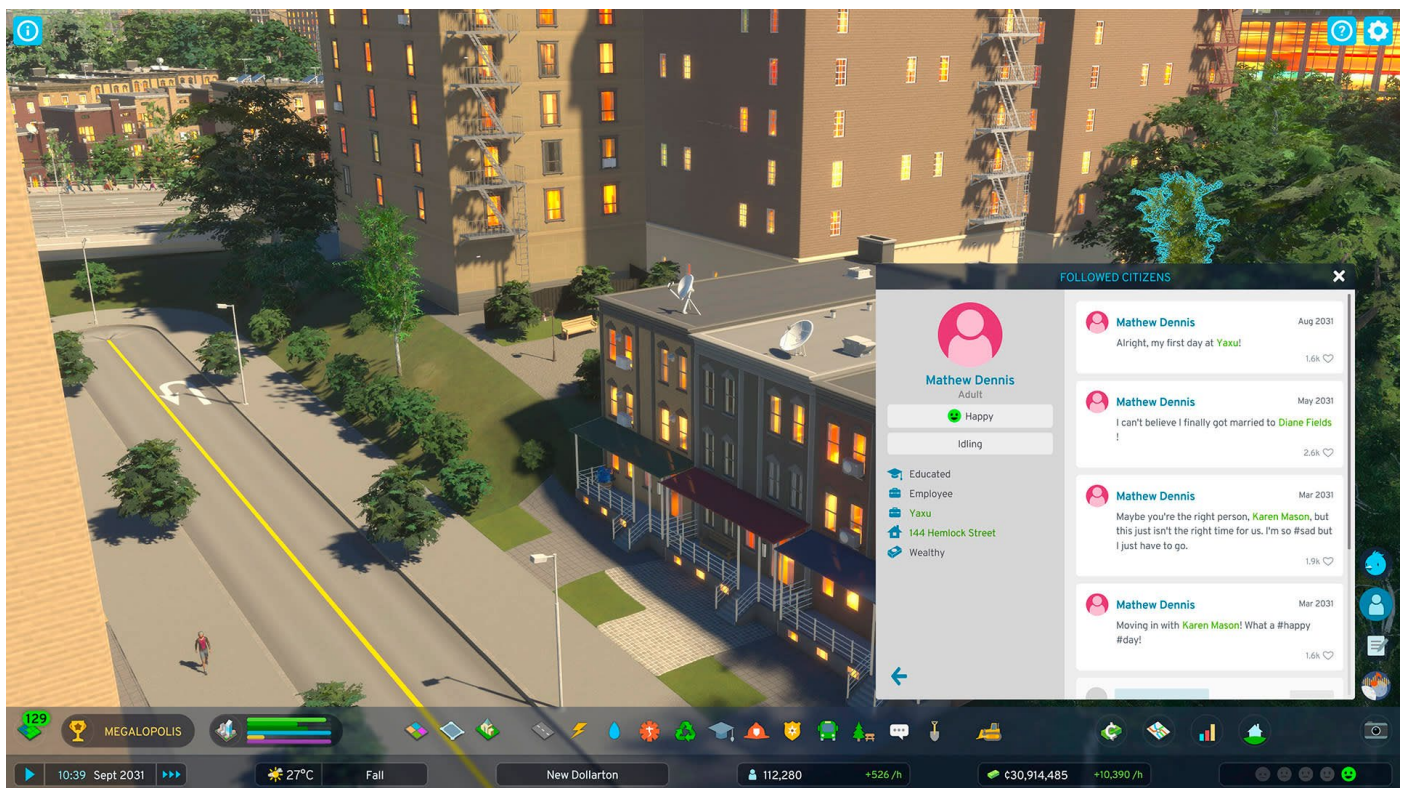


문제점과 최적화

절차적 노이즈에는 많은 양의 컴퓨팅이 필요할 수 있으며, 3D 애플리케이션이나 대규모 환경인 경우에는 더욱 그렇습니다. 성능 최적화를 위해 생성 노이즈의 해상도를 제한하거나 이전에 계산된 노이즈 값을 저장하는 캐싱 전략을 사용할 수 있습니다. 아울러 서로 다른 스케일을 갖는 노이즈 값을 함께 사용하면 복잡도를 높일 수 있고, 여러 디테일 레이어를 추가해 사실감을 높일 수 있습니다.

Unity의 절차적 노이즈는 플레이어 상호 작용에 동적으로 적응하는 다양하고, 확장 가능하고, 흥미진진한 월드를 만드는 데 도움이 되는 강력한 툴입니다.

컴퓨터 셰이더



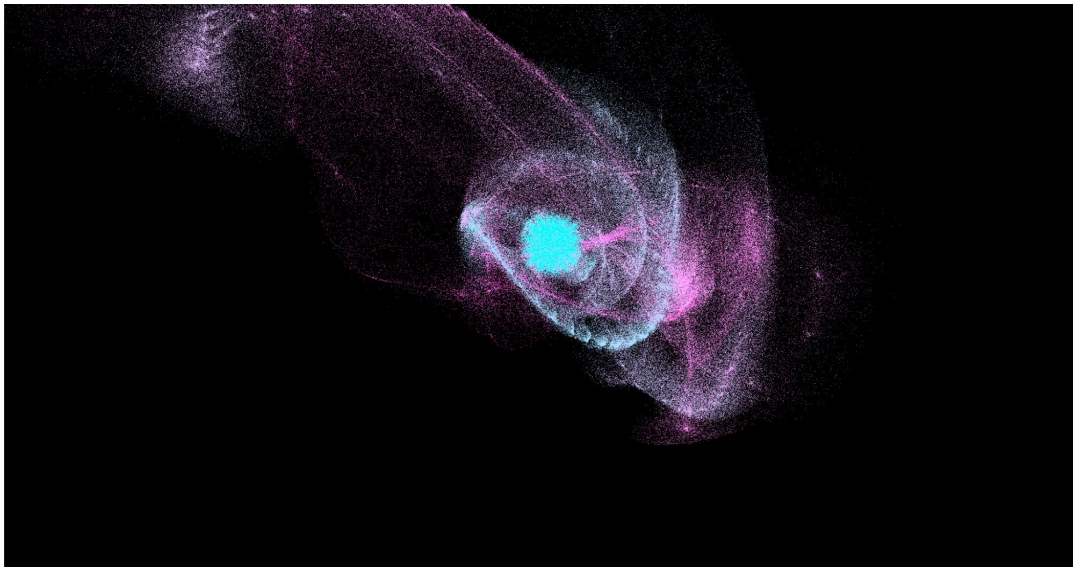
게임에서 Colossal Order와 Paradox Interactive의 [Cities: Skylines II](#)처럼 복잡한 시뮬레이션을 사용하고 싶다면, 컴퓨터 셰이더를 통해 작업의 일부를 GPU에 넘길 수 있습니다.



이 레시피에서는 컴퓨트 셰이더를 사용합니다. 컴퓨트 셰이더는 동일한 계산이 여러 개의 엔티티에 적용되며 계산에 많은 리소스를 소모하는 작업에서 사용할 수 있습니다. 파티클 효과와 군집을 예시로 살펴보겠습니다. 컴퓨트 셰이더를 처음 접한다면 본 레시피의 끝부분에 있는 리소스를 참고하세요.

Unity에서 제공하는 두 개의 파티클 시스템 생성용 시스템(빌트인 파티클 시스템 및 VFX Graph)을 사용해도 되지만, 이 레시피에서는 자체 시스템을 제작할 것입니다. 그렇게 하면 인스턴싱된 메시와 호환되는 셰이더를 만드는 데 필요한 기법을 더 쉽게 이해하고 수만 개의 메시가 사용되는 시각적 효과를 만들 수 있습니다. 사용 중인 GPU에 따라 수백만 개의 로우 폴리곤 메시도 가능합니다. 이 기법은 잔디, 머리카락, 물, 군부대, 군중 등을 생성하는 데 사용할 수 있습니다.

ParticleFun



ParticleFun 레시피 사용 사례

Scenes > Compute Shaders > ParticleFun > ParticleFun을 연 다음 Visual Studio Code로 이동하여 같은 폴더에서 `ParticleFun.cs`, `ParticleFun.compute`, `ParticleFun.shader`를 엽니다. 프로그램을 실행하면 파티클이 시간의 흐름에 따라 마우스 위치를 향해 움직이고 컬러가 변하는 것을 볼 수 있습니다. 먼저 코드부터 살펴보겠습니다. 코드는 다음 스크립트부터 시작합니다.

```
public class ParticleFun : MonoBehaviour
{
    private Vector2 cursorPos;

    // 구조체
    struct Particle
    {
        public Vector3 position;
        public Vector3 velocity;
    }
}
```




```
    public float life;
}

const int SIZE_PARTICLE = 7 * sizeof(float);

public int particleCount = 1000000;
public Material material;
public ComputeShader shader;

int kernelID;
ComputeBuffer particleBuffer;

int groupSizeX;
RenderParams rp;
```

파티클은 position, velocity, life 값을 갖습니다. 개별 파티클의 데이터는 7개의 플로트를 사용하므로 파티클의 크기는 플로트 하나의 7배 크기입니다. 그런 다음 사용자가 인스펙터에서 조정할 수 있는 public 변수의 개수를 선언합니다. 머티리얼은 ParticleFun.shader를 사용하는 Particle 머티리얼이고 셰이더는 ParticleFun.compute입니다.

Start 메서드에서 Init을 호출합니다. Init 메서드는 각 파티클을 초기화합니다. position은 **Homogeneous Clip Space**로 설정됩니다. 이는 각 축에서 -w와 w 사이에 있는 값이며, 여기서 w는 좌표의 네 번째 요소입니다. w=1이라고 가정하면 -1, -1, -1은 절두체 근처에 있는 왼쪽 하단이고 1, 1, 1은 절두체에서 멀리 떨어진 오른쪽 상단입니다.

```
Vector v = new Vector3();
v.x = Random.value * 2 - 1.0f;
v.y = Random.value * 2 - 1.0f;
v.z = Random.value * 2 - 1.0f;
v.Normalize();
v *= Random.value * 5;

particleArray[i].position.x = v.x;
particleArray[i].position.y = v.y;
particleArray[i].position.z = v.z;

particleArray[i].velocity.x = 0;
particleArray[i].velocity.y = 0;
particleArray[i].velocity.z = 0;

particleArray[i].life = Random.value * 5.0f + 1.0f;
```



먼저 x, y, z가 -1과 1 사이 값으로 무작위 설정된 Vector3를 만듭니다. 그런 다음 벡터의 길이가 1이 되도록 정규화합니다. 길이를 5배 확장합니다. 이 벡터를 사용하여 파티클 배열에 속한 개별 파티클의 위치를 설정합니다. velocity는 0으로 설정되고, life는 1과 6 사이의 랜덤 값으로 설정됩니다.

```
// 컴퓨트 버퍼 생성
particleBuffer = new ComputeBuffer(particleCount, SIZE_PARTICLE);

particleBuffer.SetData(particleArray);

// 커널의 ID 찾기
kernelID = shader.FindKernel("CSParticle");

uint threadsX;
shader.GetKernelThreadGroupSizes(kernelID, out threadsX, out _, out _);
groupSizeX = Mathf.CeilToInt((float)particleCount / (float)threadsX);

// 컴퓨트 버퍼를 셰이더 및 컴퓨트 셰이더에 바인딩
shader.SetBuffer(kernelID, "particleBuffer", particleBuffer);
material.SetBuffer("particleBuffer", particleBuffer);

rp = new RenderParams(material);
rp.worldBounds = new Bounds(Vector3.zero, 10000*Vector3.one);
```

다음 단계는 **ComputeBuffer**를 만드는 것입니다. ComputeBuffer는 요소의 수를 나타내는 count 및 각 요소의 크기를 나타내는 stride라는 두 개의 파라미터를 갖습니다(여기서는 각각 particleCount 및 SIZE_PARTICLE). 그런 다음 SetData 메서드를 사용하여 버퍼를 채웁니다. 그러면 데이터가 RAM에서 GPU 메모리로 이동합니다. **ComputeShader**에서 액세스할 수 있으려면 모든 데이터가 GPU 메모리에 있어야 합니다. ComputeShader의 코드는 ComputeShader 코드에 있는 커널이라는 특수 함수를 사용하여 호출합니다. 각 커널은 고유한 ID를 갖습니다. 이 ID는 파라미터로 함수 이름을 사용하고 **FindKernel** 메서드를 호출하여 찾을 수 있습니다.

각 커널은 x, y, z 3개의 스레드 파라미터를 갖습니다. 컴퓨트 셰이더의 비법은 병렬로 작동한다는 점에 있습니다. 파티클 예시에서 스레드 그룹의 크기는 256, 1, 1로 설정되어 있습니다. GPU의 성능을 최대한 끌어내려면 실제 기기의 아키텍처에 대해 알고 있어야 합니다.

C# 스크립트에서 컴퓨트 셰이더 메서드 **GetKernelThreadGroupSizes**를 사용하여 스레드 그룹 크기에 액세스할 수 있습니다. 모든 파티클을 포괄하는 스레드를 사용하려면 다음 코드에 나와 있는 횡수만큼 커널을 디스패치해야 합니다.

```
Mathf.CeilToInt((float)particleCount / (float)threadsX)
```

이 예시에서 모든 작업은 x 스레드에서 진행됩니다. `particleBuffer`가 머티리얼과 컴퓨트 셰이더로 전달되는 것을 볼 수 있습니다.

```
shader.SetBuffer(kernelID, "particleBuffer", particleBuffer);
material.SetBuffer("particleBuffer", particleBuffer);
```

이 예시의 핵심은 바로 GPU에 상주하는 공유 `ComputeBuffer`를 컴퓨트 셰이더와 버텍스 프래그먼트 셰이더에서 사용할 수 있다는 것입니다. 따라서 컴퓨트 셰이더에서 버퍼의 내용을 조작할 수 있으며, 버텍스 프래그먼트 셰이더를 사용하여 오브젝트를 렌더링할 때가 되면 렌더링에서 동일한 버퍼가 사용됩니다.

[RenderParams](#) 인스턴스를 초기화해야 하는데, 이때 큰 `Bounds` 인스턴스를 설정하세요. 이 인스턴스는 [Graphics.RenderPrimitives](#)를 사용하여 파티클을 렌더링할 때 필요합니다.

이제 `Update` 메서드를 살펴보겠습니다. 여기서 컴퓨트 셰이더의 `deltaTime`과 `mousePosition`을 설정합니다. 그런 다음 앞에서 찾은 `kernelID`를 [Dispatch](#)합니다. `Dispatch`할 때는 x, y, z 차원에 대해 작업 그룹의 개수를 설정합니다. 커널을 `particleCount`회 실행해야 하고 x 스레드 그룹의 크기는 256이므로 `groupSizeX`는 `particleCount / 256`의 정수 올림이 됩니다. 올림(ceiling)은 단순히 소수점 값을 가장 가까운 큰 정수로 올리는 것을 의미합니다. 예를 들어 7/2의 부동 소수점 값이 3.5일 경우, 정수 올림을 적용하면 그 값은 다음 정수인 4가 됩니다. x 차원에 대해 `groupSizeX`를 사용하면 커널은 x 값이 0부터 `particleCount-1`까지의 각 인덱스 값을 갖도록 실행되며, `particleCount`가 256의 배수가 아닌 경우 더 높은 인덱스 값을 갖도록 실행됩니다. `Dispatch`가 완료되면 `particleBuffer`는 각 파티클의 새 위치 값을 갖게 됩니다.

이 시점에서 `Graphics` 인터페이스의 메서드를 사용하는데, 그러려면 먼저 설명이 필요합니다. 이 `RenderPrimitives` 메서드는 다음과 같은 네 개의 파라미터를 받습니다.

- 최소한 `Bounds`의 면적을 정의하는 `RenderParams` 인스턴스
- 메시 토폴로지의 유형(여기서는 점을 렌더링하지만 원하는 경우 삼각형의 선을 렌더링할 수 있음)
- 단일 인스턴스의 버텍스 개수, 점의 경우 이 값은 항상 1임
- 인스턴스 개수, 이 예시에서는 파티클의 개수

```
void Update()
{
    float[] mousePosition2D = { cursorPos.x, cursorPos.y };

    // 컴퓨트 셰이더로 데이터 전송
    shader.SetFloat("deltaTime", Time.deltaTime);
    shader.SetFloats("mousePosition", mousePosition2D);
```



```
// 파티클 업데이트
shader.Dispatch(kernelID, groupSizeX, 1, 1);

Graphics.RenderPrimitives(rp, MeshTopology.Points, 1, particleCount);
}
```

실제 렌더링은 머티리얼에 연결된 셰이더에 의해 처리됩니다. 그 부분을 살펴보겠습니다.

ParticleFun.shader 파일에 해당 버퍼에 대한 참조를 추가해야 합니다. 그러려면 파티클 구조체의 정의가 필요합니다.

```
struct Particle{
float3 position;
    float3 velocity;
    float life;
};
StructuredBuffer<Particle> particleBuffer;
```

버퍼를 **RWStructuredBuffer**가 아닌 **StructuredBuffer**로 설정하는 이유는 셰이더가 이 버퍼에 데이터를 쓰지 않을 것이기 때문입니다. 데이터 쓰기는 컴퓨트 셰이더가 담당합니다.

_PointSize 프로퍼티가 있습니다. **Attributes** 구조체(**vert** 함수로 전달되는 인스턴스)는 **instanceID** 프로퍼티를 갖습니다. 점 셰이더의 경우 **instance_id**가 0부터 **particleCount-1**까지의 값으로 설정됩니다. 이 값을 버퍼를 참조하는 인덱스로 사용합니다. 컴퓨트 셰이더가 이 **position** 값을 업데이트할 것이므로, 위치 지정에는 컴퓨트 셰이더를, 렌더링에는 버텍스 프래그먼트 셰이더를 사용할 수 있습니다. **MeshTopology** 점을 사용할 때는 셰이더가 시맨틱 **PSIZE**를 갖는 입력 파라미터를 점의 픽셀 점 크기로 설정해야 합니다. 여기서는 스크립트에 의해 전달되는 변수 **PointSize**로 설정합니다.

```
Shader "Custom/ParticleFun"
{
    Properties
    {
        _PointSize("Point size", Float) = 5.0
    }

    SubShader
    {
        Tags { "RenderType" = "Opaque" "RenderPipeline" = "UniversalPipeline" }
```



```
Pass
{
    HLSLPROGRAM

    #pragma vertex vert
    #pragma fragment frag

    struct Particle{
        float3 position;
        float3 velocity;
        float life;
    };

    StructuredBuffer<Particle> particleBuffer;

    #include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Core.hlsl"

    CBUFFER_START(UnityPerMaterial)

    float _PointSize;
    CBUFFER_END

    struct Attributes
    {
        float4 positionOS : POSITION;
        uint instanceID : SV_InstanceID;
        UNITY_VERTEX_INPUT_INSTANCE_ID
    };

    struct Varyings
    {
        float4 positionHCS : SV_POSITION;
        float4 color : COLOR;
        float size: PSIZE;
    };

    Varyings vert(Attributes IN)
    {
        Varyings OUT;
```




```
Particle particle = particleBuffer[IN.instanceID];

// 컬러
float lerpVal = particle.life * 0.25f;
OUT.color = half4(1.0f - lerpVal + 0.1, lerpVal + 0.1, 1.0f, lerpVal);

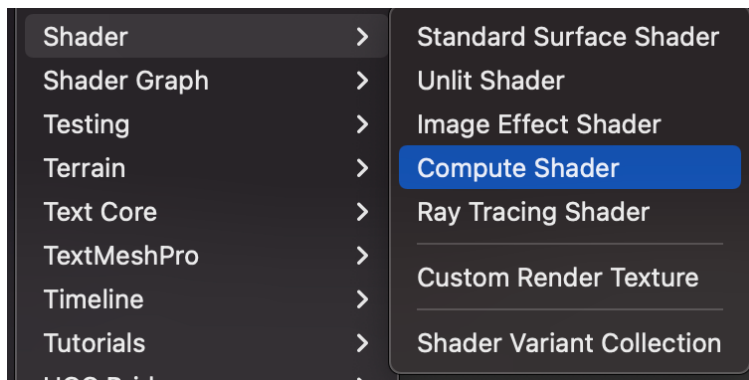
// 위치
OUT.positionHCS = TransformObjectToHClip(particle.position);

OUT.size = _PointSize;

return OUT;
}

half4 frag(Varyings IN) : SV_Target
{
    return IN.color;
}
ENDHLSL
}
}
```

이번에는 컴퓨트 셰이더를 살펴보겠습니다. 프로젝트 뷰를 오른쪽 클릭하여 **Create > Shaders > Compute Shader**를 통해 컴퓨트 셰이더를 만듭니다.



컴퓨트 셰이더를 만듭니다.



파티클에 사용할 버퍼를 정의해야 합니다. 버퍼는 스크립트의 버퍼와 일치하는 구조체를 가져야 하고, 이 셰이더는 버퍼에 데이터를 쓸 것이므로 `RWStructuredBuffer`를 선언해야 합니다.

```
// 파티클의 데이터
struct Particle
{
    float3 position;
    float3 velocity;
    float life;
};

// 셰이더와 공유되는 파티클의 데이터
RWStructuredBuffer<Particle> particleBuffer;
```

`CSParticle` 커널은 다음과 같습니다.

```
Particle particle = particleBuffer[id.x];

particle.life -= deltaTime;

float3 delta = float3(mousePosition.xy, 0) - particle.position;
float3 dir = normalize(delta);

particle.velocity += dir;
particle.position += particle.velocity * deltaTime;

particleBuffer[id.x] = particle;

if (particle.life < 0) respawn(id.x);
```

버퍼에서 인덱스가 `id.x`인 파티클을 가져옵니다. 이 커널이 `Dispatch`되는 방식 때문에 이 파티클은 앞에서 언급된 대로 0부터 `particleCount-1`까지의 값을 갖습니다. 그런 다음 `life`를 줄입니다. 스크립트를 통해 컴퓨트 셰이더의 각 스크린 업데이트와 함께 전달되는 `deltaTime` 프로퍼티를 사용합니다. 이 값은 마지막 업데이트로부터 경과한 시간(초)입니다. Unity는 새로 렌더링된 프레임을 1초에 약 60회 표시하려 하므로 이 값은 매우 작을 것입니다. 따라서 `deltaTime`은 약 16밀리초(0.016초)가 됩니다.

`z` 값을 0(파티클의 `z` 기본값)으로 설정하여 파티클부터 마우스 위치까지의 벡터를 만듭니다. 마우스 위치 값은 마우스의 월드 위치로 설정됩니다. `ParticleFun.cs` 파일에서 `OnGUI` 이벤트를 사용하세요.



```
void OnGUI()
{
    Vector3 p = new Vector3();
    Camera c = Camera.main;
    Event e = Event.current;
    Vector2 mousePos = new Vector2();

    // Event에서 마우스 위치 가져오기
    // Event에서 가져온 y 위치가 반전됨
    mousePos.x = e.mousePosition.x;
    mousePos.y = c.pixelHeight - e.mousePosition.y;

    p = c.ScreenToWorldPoint(new Vector3(mousePos.x, mousePos.y,
                                          c.nearClipPlane));

    cursorPos.x = p.x;
    cursorPos.y = p.y;
}
```

이렇게 하면 마우스 눌림이 감지되는데, 약간의 코드를 사용하여 이를 스크린 위치로 전환합니다. 스크린 좌표는 아래쪽을 0으로 사용하고, `Event.mousePosition`은 위쪽을 0으로 사용합니다. y 값을 반전하려면 `Camera.pixelHeight`에서 `mousePosition.y`를 뺍니다. 그런 다음 `Camera` 메서드 `ScreenToWorldPoint`(이 메서드에는 Z 값이 필요함)를 사용합니다. 이때 `nearClipPlane`을 사용하면 성공적인 결과를 얻을 수 있습니다.

마우스 위치를 월드 좌표로 전환하는 방법을 자세히 알아보려면 [Game Dev Beginner](#)의 [이 게시물](#)을 참조하세요.

이제 마우스 위치부터 파티클을 가속하는 데 사용할 수 있는 벡터가 준비되었습니다. 파티클의 `velocity`를 `deltaTime`만큼 변조하여 사용합니다.

파티클의 `life`가 0보다 작으면 파티클을 재생성(`respawn`)하고 `fast-random` 함수를 사용하여 필요한 무작위 값을 생성합니다. [조지 마르살리아](#)가 제안한 `XorShift` 난수 생성기는 실시간 그래픽스에 무척 유용한 기능입니다.

파티클은 중심이 마우스 위치이고 `z=0`이며 반지름이 0.8인 구 안에 위치합니다. 새 파티클의 `life`를 4초로 재설정하고 `velocity`를 0으로 재설정합니다.



```
void respawn(uint id)
{
    rng_state = id;
    float tmp = (1.0 / 4294967296.0);
    float f0 = float(rand_xorshift()) * tmp - 0.5;
    float f1 = float(rand_xorshift()) * tmp - 0.5;
    float3 normalF3 = normalize(float3(f0, f1, 0.0)) * 0.8f;
    normalF3 *= float(rand_xorshift()) * tmp;
    particleBuffer[id].position = float3(normalF3.x + mousePosition.x, normalF3.y +
    mousePosition.y, 0.0);
    // 이 파티클의 life 재설정
    particleBuffer[id].life = 4;
    particleBuffer[id].velocity = float3(0, 0, 0);
}
```

life 프로퍼티를 사용하여 파티클의 컬러를 제어해 보겠습니다. 버텍스 프래그먼트 셰이더의 `vert` 함수에서 컬러 할당에 다음 코드를 추가합니다.

```
float life = particleBuffer[instance_id].life;
float lerpVal = life * 0.25f;
o.color = fixed4(1.0f - lerpVal + 0.1, lerpVal + 0.1, 1.0f, lerpVal);
```

이제 파티클의 컬러가 변합니다. life가 `respawn` 함수에 의해 4로 설정되고 이 값에 0.25를 곱하기 때문에 lerp 값은 1과 0 사이의 값이 됩니다. 빨간색 채널은 0.1에서 시작하고 life가 줄어들수록 1.1로 늘어나며, 초록색 채널은 1.1에서 시작하고 시간이 흐름에 따라 0.1로 줄어듭니다. 파란색은 항상 1이고 알파는 시간이 흐름에 따라 줄어들기 때문에 픽셀이 흐려집니다.

이 예시는 동일한 에셋을 갖는 여러 개의 인스턴스를 렌더링할 때 컴퓨트 셰이더와 버텍스 프래그먼트 셰이더를 함께 사용하는 것이 얼마나 유용한지 보여 줍니다. 여기서는 가장 간단한 에셋인 단일 픽셀 컬러 값이 사용되었습니다. 다음 예시에서는 이 개념을 확장하여 여러 개의 메시 오브젝트를 렌더링하는 방법을 보여 줍니다.

메시 오브젝트 추가

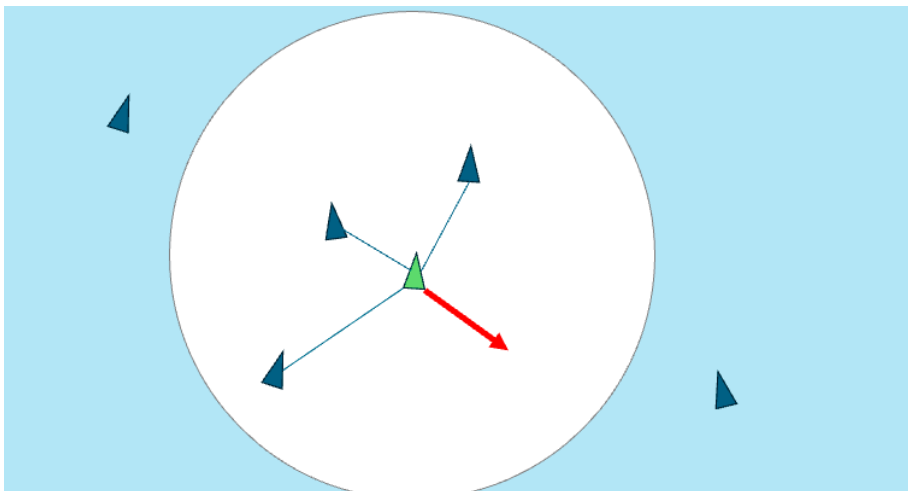


인스턴스화된 군집 씬

이 메시 오브젝트는 군집을 구현합니다. 군집이라는 개념은 하늘을 나는 새 무리를 관찰한 데서 비롯되었으며, 무리의 움직임이 단순한 규칙들만으로 제어될 수 있다는 결론에서 시작되었습니다. 첫 번째 규칙은 한 마리의 새는 주변에서 시야에 들어오는 소수 새들의 행동만 인지할 수 있다는 것입니다. [Siggraph 87](#)에서 [크레이그 레이놀즈](#)가 발표한 논문으로 처음 제안된 이론입니다. 논문에서는 군집의 구성 요소를 ‘보이드’라고 칭했으며, 이 용어는 컴퓨터로 시뮬레이션된 군집의 개별 구성원을 가리키는 일반적인 용어로 정착했습니다. 보이드 (boid)는 ‘bird-oid’ 객체의 줄임말입니다.

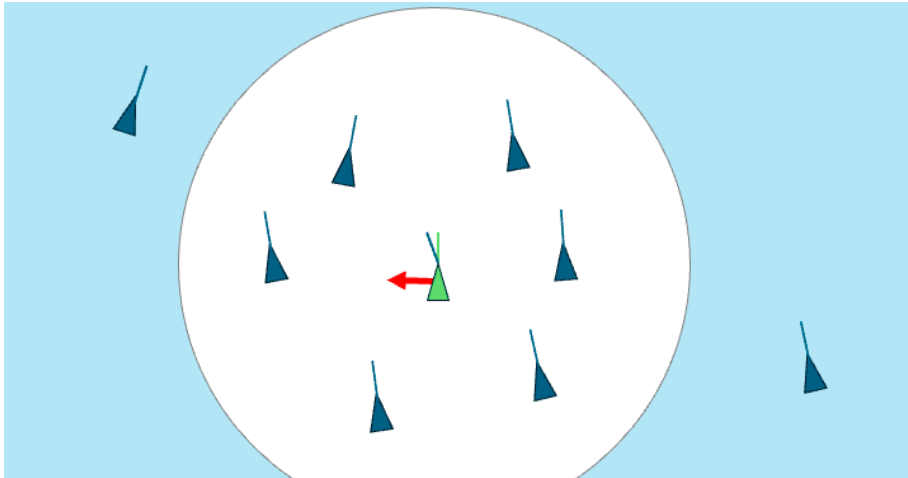
특정 사용자 정의 반지름 내에서 시야 안에 있는 군집의 구성원들을 스캔한 후 다음과 같은 3가지 규칙을 사용하여 각 보이드의 위치, 방향, 속도를 조정합니다.

1. **분리:** 근처에 있는 다른 보이드와 너무 가까워지지 않도록 방향을 조정합니다. 이를 위해, 현재 보이드에서 근처에 있는 다른 보이드로 뺄 벡터들과 다른 방향을 갖는 벡터를 찾습니다.



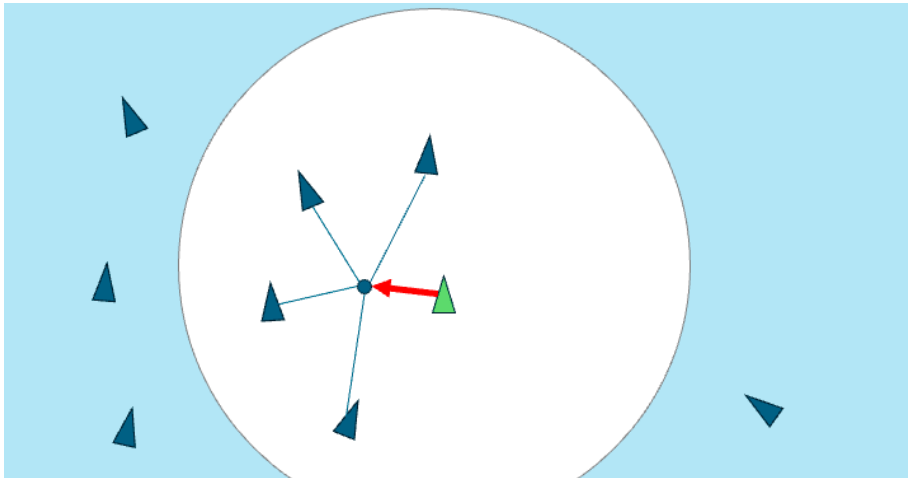
이미지: 크레이그 레이놀즈, 보이드: 배경 및 업데이트(영문)

2. **정렬:** 근처에 있는 보이드의 평균적인 방향을 향해 방향을 조정합니다. 현재 방향을 근처에 있는 보이드의 평균 방향으로 선형 보간합니다.



이미지: 크레이그 레이놀즈, 보이드: 배경 및 업데이트(영문)

3. **응집:** 근처에 있는 보이드의 평균 위치로 향하도록 방향을 조정합니다. 근처에 있는 보이드의 평균 위치를 찾고 해당 위치 쪽으로 방향을 조정합니다.



이미지: 크레이그 레이놀즈, 보이드: 배경 및 업데이트(영문)

분리와 응집이 서로 위배되는 것처럼 보인다면, 분리를 위해서는 현재 보이드로부터 뺀 벡터를 사용하고 응집을 위해서는 보이드 그룹의 평균 위치를 계산에 사용한다는 점을 기억하세요.

구체적인 예시를 살펴보겠습니다.

Scenes > ComputeShaders > Instanced에서 **InstancedFlocking** 씬을 열고, VS Code로 이동하여 같은 폴더에서 **InstancedFlocking.cs** 및 **InstancedFlocking.compute**를 엽니다. 또 다른 **ComputeBuffer**를 사용할 것이며, 이 예시에서는 컴퓨트 셰이더에 집중합니다.



보이드는 position, direction, noise offset 값을 갖습니다. 이 구조체에 생성자 메서드가 추가되지만, 구조체의 데이터는 Vector3 1개와 float 1개입니다.

```
public struct Boid
{
    public Vector3 position;
    public Vector3 direction;
    public float noise_offset;

    public Boid(Vector3 pos, Vector3 dir, float offset)
    {
        position.x = pos.x;
        position.y = pos.y;
        position.z = pos.z;
        direction.x = dir.x;
        direction.y = dir.y;
        direction.z = dir.z;
        noise_offset = offset;
    }
}
```

사용자가 군집의 행동을 조정하는 데 사용하는 여러 public 프로퍼티를 볼 수 있는데, 셰이더를 코딩하는 과정에서 하나씩 살펴보겠습니다.

InitBoids는 보이드 배열을 만들고 채웁니다. position은 구 안에 있는 무작위 값입니다.

InitShader는 보이드 배열로부터 ComputeBuffer를 만들고 설정한 다음 컴퓨트 셰이더의 여러 프로퍼티를 설정합니다.

Update 메서드는 time과 delta time을 설정하고 커널을 디스패치합니다. 컴퓨트 셰이더가 각 보이드의 새 position 및 direction 계산을 완료하면 RenderMeshIndirect를 사용하여 보이드를 렌더링합니다. [RenderMeshIndirect](#)에는 [RenderParams](#) 인스턴스, 메시, [GraphicsBuffer](#) 인스턴스가 필요합니다. RenderParams 인스턴스는 Start 메서드에서 초기화됩니다.

```
renderParams = new RenderParams(boidMaterial);
renderParams.worldBounds = new Bounds(Vector3.zero, Vector3.one * 1000);
```



생성자에 머티리얼이 전달된 것에 주목하세요. 이 머티리얼은 인스턴싱을 지원해야 합니다. 사용할 메서드는 URP Lit 셰이더를 수정하는 용도입니다. 이 작업을 위해 **Library/PackageCache/com.unity.render-pipelines.universal/Shaders** 폴더에서 다음과 같은 파일을 복사합니다.

- Lit.shader
- LitForwardPass.hlsl
- ShadowCasterPass.hlsl(그림자가 지원되는 경우)

파일을 하나의 하위 폴더에 저장합니다.

VS Code에서 **Scenes > ComputeShaders > Instanced > Shader**를 살펴보세요. PackageCache 폴더에서 복사된 3개의 파일을 볼 수 있습니다. Lit.shader에는 거의 변경된 사항이 없습니다. 셰이더의 이름이 변경되었습니다.

```
Shader "Custom/Flocking/Instanced"
{
    ...
```

LitForwardPass.hlsl의 경로가 점 하나로 변경되었으며, 이는 Lit.shader 파일과 같은 폴더에 있는 파일을 사용할 것임을 의미합니다. LitForwardPass.hlsl 파일에서 boid 구조체와 boidsBuffer를 정의합니다.

```
struct Boid
{
    float3 position;
    float3 direction;
    float noise_offset;
};

StructuredBuffer<Boid> boidsBuffer;
```

create_matrix 함수를 추가합니다. 이 함수는 position, direction, up 벡터로부터 회전 및 위치 매트릭스를 만듭니다.

```
float4x4 create_matrix(float3 pos, float3 dir, float3 up) {
    float3 zaxis = normalize(dir);
    float3 xaxis = normalize(cross(up, zaxis));
    float3 yaxis = cross(zaxis, xaxis);
    return float4x4(
        xaxis.x, yaxis.x, zaxis.x, pos.x,
        xaxis.y, yaxis.y, zaxis.y, pos.y,
        xaxis.z, yaxis.z, zaxis.z, pos.z,
        0, 0, 0, 1
    );
}
```



Attributes 구조체에 `SV_instanceID` 요소를 추가하는 것이 중요합니다. 이를 통해 버텍스 셰이더의 `instanceID`에 액세스할 수 있습니다.

```
struct Attributes
{
    float4 positionOS    : POSITION;
    float3 normalOS      : NORMAL;
    float4 tangentOS     : TANGENT;
    float2 texcoord      : TEXCOORD0;
    float2 staticLightmapUV : TEXCOORD1;
    float2 dynamicLightmapUV : TEXCOORD2;
    uint instanceID : SV_InstanceID;
    UNITY_VERTEX_INPUT_INSTANCE_ID
};
```

`Lit.shader`는 함수 `LitPassVertex`를 버텍스 셰이더로 정의합니다. 이 함수는 Forward 렌더 파이프라인을 위한 `LitForwardPass.hlsl`에 있습니다.

입력 Attributes 파라미터에 `instanceID`가 있으므로 `boidsBuffer`에서 개별 보이드를 가져올 수 있습니다. `create_matrix` 함수와 `boid position` 및 `direction`을 사용하여 매트릭스를 만듭니다. `input.positionOS`와 방금 만든 매트릭스를 곱하여 이 파일의 기본 버전에서 `GetVertexPosition`으로 전달된 파라미터를 편집합니다. `GetVertexPosition`에는 `float3`가 필요하므로 `mul` 함수 밖에 `.xyz`를 추가하여 `float4` 출력을 `float3` 출력으로 변경합니다. `NormalInput`도 동일하게 처리합니다.

```
Boid boid = boidsBuffer[input.instanceID];

float4x4 mat = create_matrix(boid.position, boid.direction, float3(0.0, 1.0, 0.0));

VertexPositionInputs vertexInput = GetVertexPositionInputs(mul(mat, input.positionOS).xyz);

VertexNormalInputs normalInput = GetVertexNormalInputs(mul(mat, input.normalOS), mul(mat, input.tangentOS));
```

이제 사용할 머티리얼이 준비되었습니다. 이 머티리얼을 사용하여 여러 개의 메시 오브젝트를 표시하는 방법을 알아보겠습니다.

그래픽스 버퍼인 또 다른 버퍼 `argsBuffer`가 스크립트에 추가되었습니다. 버텍스 프래그먼트 셰이더는 렌더링 시에 이 버퍼를 사용합니다. 이 버퍼를 초기화하는 코드를 추가해야 합니다.

버퍼를 만들 때는 유형을 `IndirectArguments`로, 배열의 요소를 1로, 요소의 크기를 `IndirectDrawIndexedArgs`의 `size`로 설정합니다. 하나의 `IndirectDrawIndexedArgs`로 구성된 배열을 만듭니다. `GetIndexCount` 메서드를 사용하여 `indexCountPerInstance`를 `boidMesh`의 버텍스 카운트로 설정하고, `instanceCount`를 `numOfBoids` 프로퍼티로 설정합니다.



SetData 메서드를 사용하여 데이터를 argsBuffer에 복사합니다. 이제 데이터가 GPU에 상주합니다.

```
argsBuffer = new GraphicsBuffer(GraphicsBuffer.Target.IndirectArguments, 1, GraphicsBuffer.IndirectDrawIndexedArgs.size);
GraphicsBuffer.IndirectDrawIndexedArgs[] data = new GraphicsBuffer.IndirectDrawIndexedArgs[1];
data[0].indexCountPerInstance = boidMesh.GetIndexCount(0);
data[0].instanceCount = (uint)numOfBoids;
argsBuffer.SetData(data);
```

이제 Update 메서드를 살펴보겠습니다. 컴퓨트 셰이더의 Time과 deltaTime을 설정합니다. 보이드의 position 및 direction 계산이 완료되면 Dispatch하고, RenderMeshIndirect를 사용하여 보이드를 렌더링합니다.

```
void Update()
{
    shader.SetFloat("time", Time.time);
    shader.SetFloat("deltaTime", Time.deltaTime);

    shader.Dispatch(this.kernelHandle, groupSizeX, 1, 1);

    Graphics.RenderMeshIndirect( renderParams, boidMesh, argsBuffer );
}
```

지금까지 스크립트에 관해 알아야 할 모든 사항을 살펴보았습니다. 이번에는 컴퓨트 셰이더의 군집 코드를 살펴보겠습니다.

앞에서 살펴본 3가지 규칙인 분리, 정렬, 응집이 적용됩니다. 먼저 id.x를 기준으로 버퍼로부터 보이드를 가져옵니다. 그런 다음 separation, alignment, cohesion의 초기 값을 설정합니다. 이러한 값을 계산할 때는 근처에 있는 보이드만 고려합니다. 현재 보이드는 이 반지름 안에 있으므로 nearbyCount는 0이 아닌 1에서 시작합니다. 그런 다음 루프가 현재 보이드를 가리키고 있다면 다른 보이드는 무시해도 됩니다.

i가 id.x와 달라야 업데이트가 적용됩니다. 그런 다음 i 변수가 가리키는 보이드를 가져옵니다. 현재 보이드에서 임시 보이드까지의 거리가 neighbor distance 프로퍼티보다 짧은지 확인하는 검사가 하나 더 있습니다.

```
Boid boid = boidsBuffer[id.x];

float3 separation = 0;
float3 alignment = 0;
float3 cohesion = flockPosition;
```




```

uint nearbyCount = 1; // 루프에서 무시된 자신 추가

for (int i = 0; i < boidsCount; i++)
{
    if (i != int(id.x))
    {
        Boid tempBoid = boidsBuffer[i];
        if (distance(boid.position, tempBoid.position) < neighbourDistance){
            //위치 계산
        }
    }
}

```

가장 복잡한 계산은 분리입니다. 분리는 이 섹션 후반부에서 자세히 살펴볼 것이며, 지금은 정렬과 응집을 살펴보겠습니다. alignment의 경우 각 보이드의 direction의 합을 구하고, cohesion의 경우 각 보이드의 position의 합을 구합니다. cohesion 변수는 군집 position의 값을 갖습니다.

```

alignment += tempBoid.direction;
cohesion += tempBoid.position;
nearbyCount++;

```

그런 다음 루프 밖에서 nearbyCount 값을 사용하여 합산 값을 nearbyCount 값으로 나누어 alignment와 cohesion의 평균 값을 구합니다. cohesion의 경우 현재 보이드의 position을 빼고 결과를 정규화해야 합니다. 이렇게 하면 타겟 direction이 3개 프로퍼티의 합이 됩니다. 하지만 새로 계산한 direction을 적용할 때는 기존 boid direction과의 매우 낮은 블렌드 수준으로 사용해야 합니다. lerp를 사용하면 계산된 direction의 6%만 사용되고 나머지 94%는 유지됩니다.

보이드의 direction(rotation 값이 아닌 forward 벡터)을 구했으니 이 값에 speed와 deltaTime을 곱하여 position을 업데이트할 수 있습니다. 마지막 단계는 버퍼에 업데이트를 적용하는 것입니다.

```

float avg = 1.0 / nearbyCount;
alignment *= avg;
cohesion *= avg;
cohesion = normalize(cohesion - boid.position);

float3 direction = alignment + separation + cohesion;

boid.direction = lerp(direction, normalize(boid.direction), 0.94);
boid.position += boid.direction * boidSpeed * deltaTime;

boidsBuffer[id.x] = boid;

```

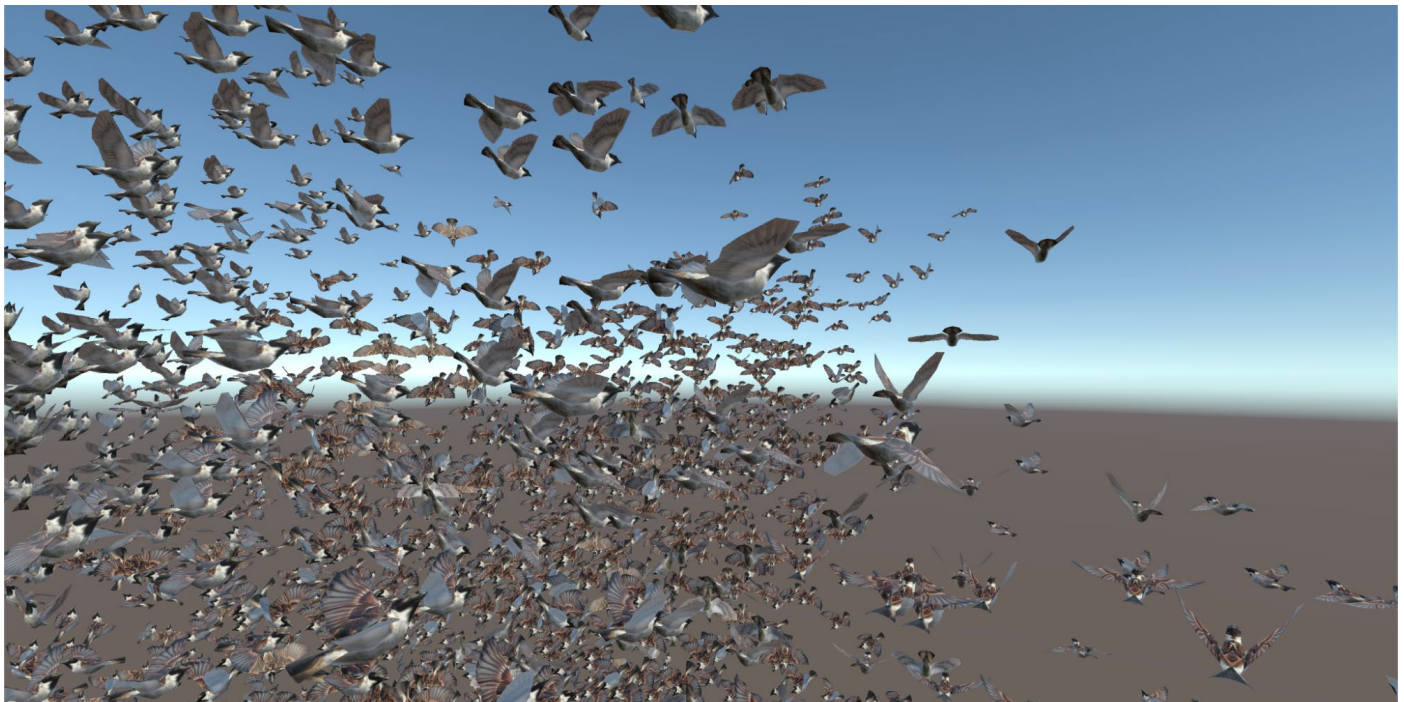
이제 프로그램을 실행하면 아직 separation 값을 적용하지 않았기 때문에 모든 보이드가 하나로 병합됩니다. 이를 수정해 보겠습니다.

앞에 나온 for 루프에서 임시 보이드에서 현재 설정하고 있는 보이드까지의 벡터와 길이를 구했습니다. 임시 보이드가 이웃 거리로 설정된 값보다 멀리 있으면 무시합니다. 이 거리보다 작으면 separation 값에 사용해야 합니다. offset 벡터를 현재 설정하고 있는 보이드에 가까워질수록 커지는 값으로 스케일링합니다. 1을 dist로 나누는 값을 사용하고 있으므로, 0으로 나누는 오류를 방지하려면 dist가 결코 0이 되면 안 됩니다.

그렇다면 $1/\text{dist}$ 빼기 $1/\text{neighbourDistance}$ 는 무엇을 의미할까요? dist가 작은 값이면 $1/\text{dist}$ 는 매우 큰 값이 되므로 작은 $1/\text{neighbor distance}$ 값을 무시할 수 있습니다. 반면에 이 보이드가 이웃 거리의 경계 가까이에 있다면 $1/\text{neighbor distance}$ 빼기 $1/\text{neighbor distance}$ 가 되어 0에 가까운 값을 얻게 됩니다. 즉, 보이드가 타겟 보이드에 가까이 있으면 separation 벡터가 매우 커지고, 이웃 거리 값에 가깝다면 무시하게 됩니다.

```
float3 offset = boid.position - tempBoid.position;
float dist = length(offset);
if (dist < neighbourDistance) {
    dist = max(dist, 0.000001);
    separation += offset * (1.0/dist - 1.0/neighbourDistance);
}
```

이제 군집 예시가 완성되었습니다. 몇 초가 지나면 모든 보이드가 무리를 이루어 군집 위치를 중심으로 회전합니다.



SkinnedFlocking 씬



이 최종 예시는 [스킨드 메시 렌더러](#)를 사용합니다. **Scenes > ComputeShaders > Skinned** 폴더에서 **SkinnedFlocking** 씬을 열고 실행합니다. 수많은 새들이 나는 모습을 볼 수 있습니다. VS Code로 이동하여 **Scripts** 폴더에서 **SkinnedFlocking.cs**를 엽니다. **Shader** 폴더에서 **SkinnedFlocking.compute** 및 **LightForwardPass.hlsl**을 엽니다.

스크립트부터 시작하겠습니다. 앞에서 본 예시와 비슷하지만 몇 가지 다른 점이 있습니다. 첫째, **boid** 구조체에 **frame** 프로퍼티가 있습니다. 이 프로퍼티를 사용하여 표시할 애니메이션 프레임을 선택합니다. 둘째, 컴퓨트 셰이더와 커스텀 릿 셰이더로 전달되는 **numOfFrames** 프로퍼티가 있습니다. 셋째, 스크립트는 셰이더의 **EnableKeyword** 메서드와 **DisableKeyword** 메서드를 사용하여 셰이더에서 정의를 추가하거나 제거합니다. 이 예시에서는 세 번째 버퍼인 **vertexAnimationBuffer**도 사용됩니다. 새 메서드인 **GenerateVertexAnimationBuffer**도 볼 수 있습니다. 이 메서드를 살펴보겠습니다.

프로그램이 시작되기 전에 [Animator](#) 컴포넌트를 사용하여 메시지를 일련의 포즈로 설정합니다. 그런 다음 포즈의 버텍스 위치를 버퍼에 저장합니다. 이 버퍼에서 올바른 인덱스를 선택하면 메시에 속한 일련의 서로 다른 포즈를 표시할 수 있습니다. 가장 먼저 필요한 것은 **Animator** 컴포넌트입니다. **Unity**의 애니메이션은 여러 개의 레이어를 사용할 수 있지만, 여기서는 간단하게 레이어 1만 사용하겠습니다.

```
animator = boidObject.GetComponentInChildren<Animator>();
int iLayer = 0;
```

Animator가 애니메이션 클립 **FlapWings**를 자동으로 재생하도록 설정된 것을 볼 수 있습니다. 포즈를 설정하려면 **state** 정보, 포즈를 저장할 새 **Mesh**, 두 개의 변수가 필요합니다.

```
AnimatorStateInfo aniStateInfo = animator.GetCurrentAnimatorStateInfo(iLayer);

Mesh bakedMesh = new Mesh();
float sampleTime = 0;
float perFrameTime = 0;
```

이 스크립트에는 **FlapWings** 애니메이션으로 설정된 **animationClip**이라는 **public** 프로퍼티가 있습니다. [AnimationClip](#)의 프레임 속도 및 길이 프로퍼티를 사용하여 베이킹할 프레임의 개수를 결정합니다. 이 값은 버텍스에 사용할 **vertexAnimationBuffer**의 인덱스를 설정하는 데 자주 사용될 것이므로 2의 제곱수이면 더 효율적입니다. **Mathf** 오브젝트에는 이 값을 설정하는 데 유용한 메서드가 있습니다. 이제 개별 프레임의 지속 시간을 가져올 수 있습니다.

```
numOfFrames = Mathf.ClosestPowerOfTwo((int)(animationClip.frameRate * animationClip.length));
perFrameTime = animationClip.length / numOfFrames;
```



메시의 버텍스 카운트가 필요합니다. 이제 버텍스 데이터를 저장하기 위해 Vector4로 구성된 배열을 설정할 수 있습니다. 배열의 길이는 vertexCount에 numOfFrames를 곱한 값입니다. 이제 버텍스 데이터를 가져올 준비가 되었습니다. 이 작업은 for 루프로 수행됩니다. aniStateInfo 오브젝트를 사용하고 레이어와 시작 시간을 설정하면 재생할 애니메이션을 가져올 수 있습니다. 그런 다음 deltaTime으로 0을 입력하여 Update를 호출합니다. 이렇게 하면 메시가 이 애니메이션에 의해 정의된 위치로 업데이트됩니다. 이제 메시지를 bakedMesh 오브젝트에 베이킹한 다음 버텍스에 대한 반복 작업을 거쳐 버텍스 배열에 값을 저장할 수 있습니다.

```
var vertexCount = boidSMR.sharedMesh.vertexCount;

Vector4[] vertexAnimationData = new Vector4[vertexCount * numOfFrames];
for (int i = 0; i < numOfFrames; i++)
{
    animator.Play(aniStateInfo.shortNameHash, iLayer, sampleTime);
    animator.Update(0f);

    boidSMR.BakeMesh(bakedMesh);
    //여기에서 버텍스 배열에 버텍스 저장
    for(int j = 0; j < vertexCount; j++)
    {
        Vector4 vertex = bakedMesh.vertices[j];
        vertex.w = 1;
        vertexAnimationData[(j * numOfFrames) + i] = vertex;
    }

    sampleTime += perFrameTime;
}
```

이 시점에서 ComputeBuffer를 설정하는 데 사용할 수 있는 버텍스 배열을 갖게 됩니다. 이 버퍼를 머티리얼로 전달합니다. 컴퓨트 셰이더에는 이 버퍼가 필요하지 않고, 셰이더에만 필요합니다. 이 버퍼에서는 읽기만 할 것입니다.

```
vertexAnimationBuffer = new ComputeBuffer(vertexCount * numOfFrames, 16);
vertexAnimationBuffer.SetData(vertexAnimationData);
boidMaterial.SetBuffer("vertexAnimation", vertexAnimationBuffer);
```

이제 컴퓨트 셰이더가 어떤 영향을 받았는지 살펴보겠습니다.

현재 속도, deltaTime, boidFrameSpeed 프로퍼티를 사용하여 boid frame 프로퍼티를 조정합니다. 움직임이 빠른 새일수록 날갯짓도 빠릅니다. frame 프로퍼티가 numOfFrames 프로퍼티를 전달하지 않았는지 확인합니다. 전달했다면 boid.frame이 0과 numOfFrames 범위 사이에 있도록 이 값을 뺍니다.



```
boid.frame = boid.frame + velocity * deltaTime * boidFrameSpeed;  
if (boid.frame >= numOfFrames) boid.frame -= numOfFrames;
```

이제 셰이더를 살펴보겠습니다. 버텍스 셰이더 함수로 전달된 `Attributes` 인스턴스가 `vertexID`와 `instanceID`를 포함하고 있어야 합니다.

```
struct Attributes  
{  
    float4 positionOS    : POSITION;  
    float3 normalOS      : NORMAL;  
    float4 tangentOS     : TANGENT;  
    float2 texcoord      : TEXCOORD0;  
    float2 staticLightmapUV : TEXCOORD1;  
    float2 dynamicLightmapUV : TEXCOORD2;  
    uint instanceID : SV_InstanceID;  
    uint vertexID : SV_VertexID;  
    UNITY_VERTEX_INPUT_INSTANCE_ID  
};
```

`LitVertexPass` 함수에서 `instanceID`를 사용하여 `boidsBuffer`로부터 `boid`를 가져옵니다. `FRAME_INTERPOLATION`이 정의되어 있지 않다면 인덱스 `vertexID * numFrames + boid.frame`에서 `positionOS.xyz`를 `vertexAnimation` 버퍼로 설정합니다.

`FRAME_INTERPOLATION`이 정의되어 있다면 `lerp`를 사용하여 `positionOS.xyz`를 현재 `boid.frame` 값과 바로 다음 프레임의 블렌드로 설정합니다. 블렌드 값은 `boid.frame`의 소수 부분에 따라 결정됩니다. 소수 부분이 0이면 블렌드는 프레임 값이 됩니다. 소수 부분이 0.5이면 현재 프레임의 50%와 바로 다음 프레임의 50%의 선형 보간이 됩니다.

```
Boid boid = boidsBuffer[input.instanceID];  
  
#ifdef FRAME_INTERPOLATION  
    uint next = boid.frame + 1;  
    if (next >= numOfFrames) next = 0;  
    float frameInterpolation = frac(boidsBuffer[input.instanceID].frame);  
    input.positionOS.xyz = lerp(vertexAnimation[input.vertexID * numFrames + boid.frame],  
vertexAnimation[input.vertexID * numFrames + next], frameInterpolation);  
#else  
    input.positionOS.xyz = vertexAnimation[input.vertexID * numFrames + boid.frame];  
#endif
```




앱을 플레이해 새들의 날갯짓 속도가 velocity에 어떻게 영향을 받는지 살펴봅니다. 그 방법은 다음과 같습니다.

1. 날갯짓 애니메이션을 수행하는 새의 포즈를 저장하는 버텍스 버퍼를 설정합니다.
2. 컴퓨트 셰이더에서 Boid 프로퍼티 frame을 0과 numOfFrames 사이의 값으로 설정합니다.
3. 이 frame 프로퍼티가 업데이트되는 속도는 새의 velocity에 따라 정해집니다.
4. 표면 셰이더에서는 Boid 프로퍼티를 사용하여 현재 프레임과 다음 프레임들을 가져옵니다.
5. 버텍스 셰이더에서는 vertexID, 현재 프레임 값과 다음 프레임 값의 조합을 사용하여 버텍스 버퍼로부터 적절한 값을 가져옵니다.

이 레시피는 컴퓨트 셰이더를 사용하여 렌더링되는 단일 점을 위치시키는 것에서 여러 개의 애니메이션 메시를 렌더링하는 것으로 확장하는 방법을 보여 줍니다. 컴퓨트 셰이더는 컴퓨팅 리소스가 많이 드는 프로세스를 처리할 때 향상된 성능을 보여 주므로 효과적으로 사용하는 법을 익혀 두는 것이 유용합니다.

더 많은 리소스

[Unity 컴퓨트 셰이더 작성 방법](#)(영문)

맺는말

본 가이드의 들어가는 말에서 언급했듯이, [Unity 고급 사용자를 위한 유니버설 렌더 파이프라인](#) 전자책은 숙련된 Unity 개발자와 테크니컬 아티스트가 [URP](#)의 최신 기능을 활용할 때 도움이 되는 유용한 가이드입니다.

유니티의 모든 고급 기술 전자책은 [Unity 베스트 프랙티스 허브](#)에서 확인하실 수 있습니다. [고급 베스트 프랙티스](#) 기술 자료 페이지에도 전자책이 제공됩니다.



unity.com/kr