→ 전자책

소개

Unity 고급 사용자를 위한 유니버설 렌더 파이프라인

Unity®

목차

들어가는 말

이 가이드는 숙련된 Unity 개발자와 테크니컬 아티스트가 Unity 6의 URP(유니버설 렌더 파이프라인)를 활용하여 최대한 효율적으로 개발하는 방법을 소개합니다.

Unity에서 제공하는 두 렌더링 솔루션인 URP와 HDRP(고해상도 렌더 파이프라인)는 SRP(스크립터블 렌더 파이프라인) 프레임워크를 기반으로 합니다. 이러한 SRP를 사용하면 C++ 등의 로우레벨 프로그래밍 언어를 사용하지 않고도 오브젝트 컬링과 표현 방식, 프레임의 포스트 프로세싱을 커스터마이즈할 수 있습니다. SRP를 완전히 커스터마이즈하여 자체적으로 만들 수도 있습니다.

URP는 모바일, XR, 무선 하드웨어용 2D 및 3D 게임을 위한 Unity의 기본 렌더러입니다. 빌트인 렌더 파이프라인의 뒤를 잇는 URP는 Unity에서 지원하는 모든 플랫폼에서 효율적으로 학습하고, 커스터마이즈하고, 스케일링할 수 있도록 디자인되었습니다. Unity 6의 URP는 빌트인 렌더 파이프라인과 같은 기능을 제공하며 여러 영역에서 품질 수준과 성능이 향상되었습니다.

이 전자책에서 안내하는 다양한 영역의 전문 가이드와 베스트 프랙티스는 다음과 같습니다.

— 새 프로젝트에 URP를 설정하거나, 기존 빌트인 렌더 파이프라인 기반 프로젝트를 URP로 전환하는 방법

— URP 품질 설정을 관리하는 방법

— 실시간 전역 조명 및 낮과 밤의 조명을 블렌딩하는 조명 시나리오에 활용할 수 있는 APV(적응적 프로브 볼륨) 같은 새로운 기능을 포함하여 URP에서 활용할 수 있는 모든 조명 툴을 사용하는 방법

— URP와 빌트인 렌더 파이프라인 셰이더의 차이를 이해하고 조명을 추가한 씬에서 URP 셰이더를 활용하는 방법

— HLSL include를 포함한 커스텀 셰이더 활용 방법

— 로컬 볼륨을 추가하거나 코드로 포스트 프로세싱을 제어하는 등 URP 포스트 프로세싱 프레임워크를 사용하는 방법

— 렌더링 레이어 사용 방법

— 새로 출시된 GPU 상주 드로어, GPU 오클루전 컬링 등 URP의 다양한 성능 최적화 툴을 적용하는 방법

— 렌더러 기능과 렌더 그래프 시스템을 활용해 렌더 파이프라인을 커스터마이즈하는 방법

멀티플랫폼 배포는 많은 게임에서 성공을 위한 핵심 요소입니다. 플레이어는 콘솔과 모바일 등 서로 다른 기기에서 동일한 게임을 플레이하는 경우가 많으므로, Unity 개발자에게는 가능한 한 적은 단계로 많은 기기에 맞게 조정할 수 있으며 복잡도가 낮은 렌더링 옵션이 필요합니다.

확장성, 커스터마이징 기능, 풍부한 기능 세트를 갖춘 URP를 사용하면 스타일라이즈드 비주얼부터 물리 기반 렌더링까지 모든 유형의 프로젝트에서 자유롭게 제작할 수 있습니다.

URP로 제작한 씬

저자 및 도움을 주신 분들

이 전자책의 저자인 **닉 레버**는 1990년대 중반부터 실시간 3D 콘텐츠를 제작했으며 2006년부터 Unity를 사용해 왔습니다. 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 Accelerate Solutions Games 팀의 시니어 디벨롭먼트 컨설턴트로 전문 분야는 스크립터블 렌더링 파이프라인이며 게임 개발 업계에서 16년 이상의 경력을 쌓았습니다.

맥심 그랑지는 8년 경력의 시니어 테크니컬 아티스트로, VR 인디 게임으로 시작하여 유니티의 조명 테크니컬 아티스트가 되었습니다. 렌더링 기법, 셰이더, 아티스트 툴 개발에 전념하는 맥심은 최적의 런타임 성능을 유지하는 동시에 뛰어난 비주얼을 구현하는 방법을 집중적으로 연구합니다.

펠리페 리라는 게임 업계에서 14년 넘게 소프트웨어 엔지니어로서 경력을 쌓았으며, 그래픽스 프로그래밍과 멀티플랫폼 게임 개발이 전문입니다.

알리 모헤발리는 게임 업계에서 21년의 경력을 보유하고 있으며 Halfbrick Studios의 Fruit Ninja와 Jetpack Joyride 를 비롯한 여러 인기 게임의 제작에 참여했습니다.

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 프로덕트 매니저입니다. 그래픽스 제품 관리 팀을 이끌며 그래픽스 로드맵과 제품 비전을 감독하고 있습니다.

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

올리버 슈나벨은 유니티 그래픽스 팀의 시니어 테크니컬 프로덕트 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.

URP: 빌트인 렌더 파이프라인의 후속 솔루션

Unity의 최대 강점 중 하나는 우수한 플랫폼 도달률입니다. 게임 스튜디오라면 게임을 만들고 나서 고사양 PC부터 저사양 모바일까지 원하는 만큼 다양한 플랫폼에 효율적으로 배포하기를 원할 것입니다.

빌트인 렌더 파이프라인은 Unity에서 지원하는 모든 플랫폼을 위한 턴키 솔루션으로 개발되었으며, 그래픽스 기능 조합을 지원하고 포워드 및 디퍼드 파이프라인과 함께 사용하기에도 편리합니다.

하지만 지원 가능한 플랫폼의 수를 지속적으로 늘리면서 유니티는 빌트인 렌더 파이프라인의 약점을 점차 인지하게 되었습니다.

— 다량의 코드가 C++로 작성되기 때문에 개발자가 수정할 수 없고 파악하기가 어렵습니다.

— 렌더 플로와 렌더 패스가 미리 구성되어 있습니다.

— 렌더링 알고리즘이 하드 코딩되어 있습니다.

— 커스터마이징에 제약이 없어 모든 플랫폼에서 우수한 성능을 구현하기가 어렵습니다.

— 렌더링 코드에 콜백이 노출되어 파이프라인에서 동기화 포인트가 트리거됩니다. 이러한 콜백으로 인해 멀티스레드 렌더링의 최적화 수준이 저해되며, 프레임의 어떤 지점에서든 C#을 호출하여 동적으로 상태를 주입, 변경할 수 있습니다.

— 데이터 캐싱으로 사용자 인젝션에 대한 지속성 상태를 관리하기가 어려워집니다.

해결책: 스크립터블 렌더 파이프라인

SRP는 다음과 같은 장점으로 효율적인 멀티플랫폼 워크플로를 지원할 수 있도록 개발되었습니다.

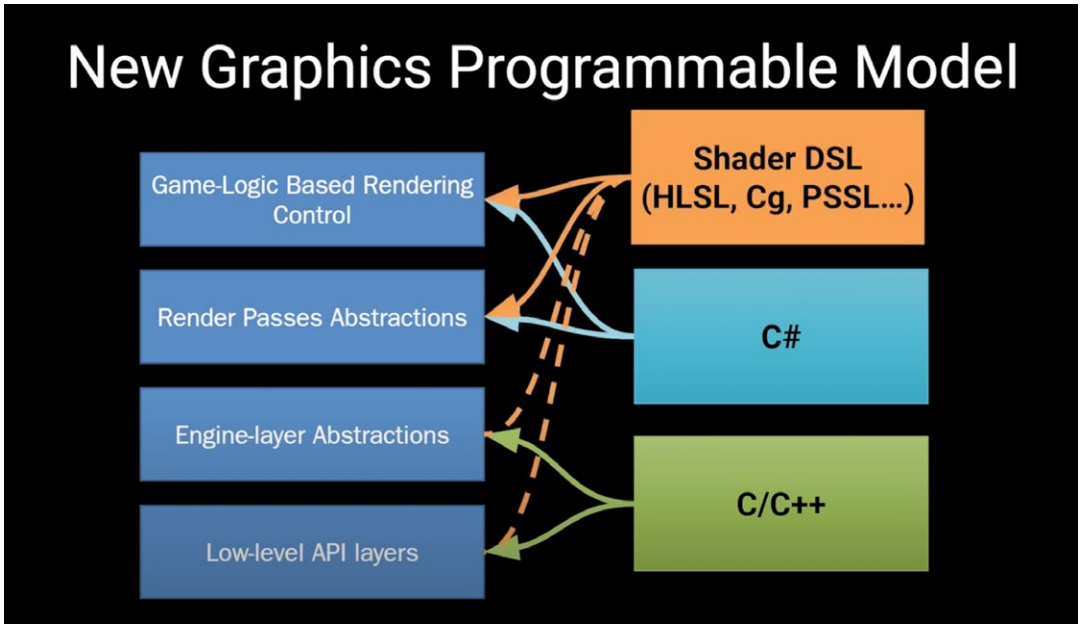
— 고사양에서 저사양 기기에 이르기까지 최대한 많은 하드웨어 플랫폼에 맞게 지능적이며 안정적인 지원을 제공합니다.

— C++이 아닌 C#을 사용하여 렌더링 프로세스를 커스터마이즈할 수 있도록 합니다. C#을 사용하면 모든 변경에 대해 새로운 실행 파일을 컴파일할 필요가 없습니다.

— 아키텍처의 변화를 지원할 수 있는 유연성을 제공합니다.

— 많은 플랫폼에서 적절한 성능을 내는 선명한 그래픽스를 만드는 데 필요한 유연성을 제공합니다.

아래 이미지에서 SRP의 작동 방식을 볼 수 있습니다. SRP를 사용하면 C#을 사용하여 렌더 패스 및 렌더링 컨트롤뿐 아니라 Shader Graph처럼 아티스트에게 익숙한 툴로 생성할 수 있는 HLSL 셰이더도 모두 제어하고 커스터마이즈할 수 있습니다. 셰이더를 통해 하위 수준 API와 엔진 레이어 추상화에도 액세스할 수 있습니다.



스크립터블 렌더 파이프라인의 프로그래밍 가능한 그래픽스 모델

고급 사용자는 SRP를 새로 생성하거나 HDRP 또는 URP를 수정할 수 있습니다. 그래픽스 스택은 오픈 소스이며 GitHub에서 사용할 수 있습니다.

URP를 선택해야 하는 이유

— **다양한 사용자의 액세스 지원:** URP는 아티스트와 테크니컬 아티스트 모두가 설정할 수 있으며, 뛰어난 유연성 덕분에 프로토타이핑뿐만 아니라 게임의 정식 제작용으로 렌더링 기법을 개선하는 데에도 사용할 수 있습니다.

— **확장 및 커스터마이징:** URP는 사용자가 기존 기능을 수정하고 파이프라인을 새 파이프라인으로 확장할 수 있도록 합니다. 따라서 에셋 스토어 및 타사 패키지 크리에이터, 숙련된 스튜디오, 고급 팀을 포함한 상급 사용자에게 믿을 수 있는 옵션이 됩니다.

하위 수준 렌더링 API는 성능을 고려해 C++로 작성되지만, URP 개발자는 렌더 파이프라인 중에 호출할 간단한 C# 스크립트만 작성하면 성능 저하 없이 높은 수준으로 커스터마이즈할 수 있습니다.

— **다양한 렌더링 옵션:** URP는 포워드, 포워드+, 디퍼드 렌더링 경로를 지원하는 유니버설 렌더러와 2D 렌더러를 제공합니다.

이러한 렌더러는 추가 기능 및 스크립터블 렌더 패스를 사용하여 확장할 수 있습니다. 오브젝트 렌더링 기능은 렌더링 파이프라인의 다양한 이벤트에서 특정 레이어 마스크로부터 오브젝트를 렌더링하는 데

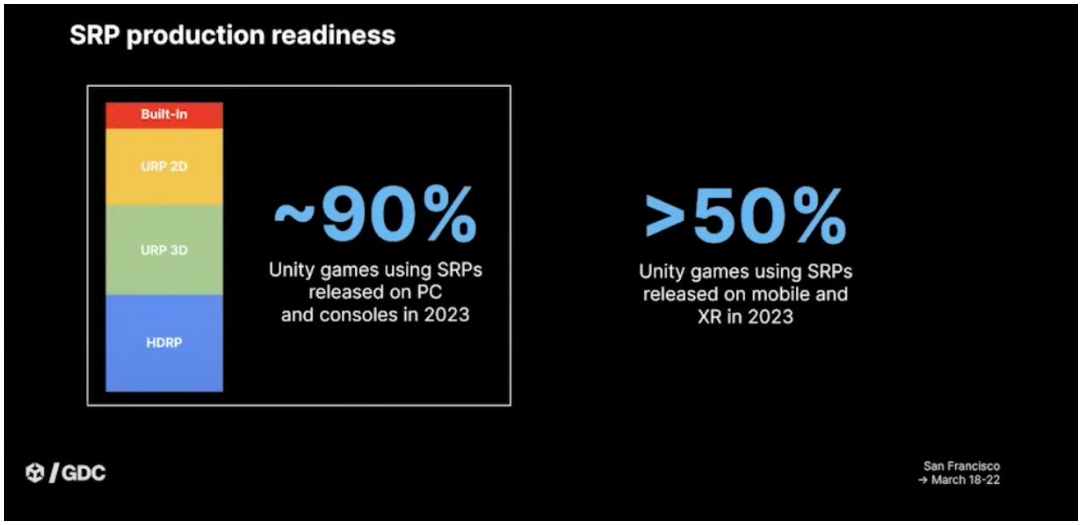
사용됩니다. 이러한 오브젝트를 렌더링할 때 머티리얼 및 기타 렌더 상태를 오버라이드할 수도 있어, 코드를 작성하지 않고도 렌더링을 커스터마이즈할 수 있습니다. 또한 특정 요구 사항에 맞춰 커스텀 렌더러를 통해 URP를 확장할 수 있습니다.

렌더 그래프 시스템을 사용하면 프레임 렌더링에 사용되는 다양한 버퍼에 액세스하고 이를 조작할 수 있으며, 렌더러 기능 워크플로를 사용하면 렌더 파이프라인의 어느 단계에서든 렌더 그래프를 추가할 수 있습니다.

- **탁월한 성능**: URP는 다양한 기기에서 강력한 성능을 제공하며 정확도와 프레임 속도 간의 균형을 조정할 수 있는 다양한 툴을 제공합니다. 특히 다음을 지원합니다.
  - URP는 실시간 조명을 매우 효율적으로 계산합니다. 포워드 렌더링의 경우 싱글 패스에서 모든 조명을 계산하며, 포워드+ 렌더링은 오브젝트 단위 대신 공간을 분류해 광원을 컬링하여 표준 포워드 렌더링보다 더 나은 품질을 보입니다. 이 방식을 사용하면 프레임을 렌더링할 때 전반적으로 더 많은 광원을 활용할 수 있습니다. 디퍼드 렌더링에서는 Native RenderPass API를 지원하여 G버퍼와 조명 패스가 싱글 렌더 패스로 결합될 수 있도록 합니다.
  - 메시를 드로우할 때 CPU 및 GPU 성능이 개선됩니다. SRP 배처를 사용하면 드로우 콜을 줄이고 뎁스 처리 방식을 개선할 수 있으며, GPU 상주 드로어와 오클루전 컬링을 사용하면 일부 씬에서 드로우 콜을 크게 줄일 수 있습니다.
  - URP를 사용하면 모바일 기기에서 타일 메모리를 더 효율적으로 사용할 수 있습니다. 따라서 전력 소비량이 줄고 배터리 수명도 길어지므로 플레이 세션이 연장될 수 있습니다.
  - URP는 통합 포스트 프로세싱 스택을 포함하여 빌트인 렌더 파이프라인에 비해 높은 성능을 제공합니다. 볼륨 프레임워크를 사용하면 코드를 작성하지 않고도 카메라 위치 기반의 포스트 프로세싱 효과를 만들 수 있습니다.
- **다른 주요 툴과 호환:** URP는 Shader Graph, VFX Graph, 렌더링 디버거 같이 아티스트와 테크니컬 아티스트에게 익숙한 툴을 지원합니다.
- **2D 렌더링:** URP는 고급 2D 조명, 그림자, 포스트 프로세싱 효과를 지원하여 성능을 그대로 유지하면서 2D 게임의 화질을 향상할 수 있습니다.

약 90%의 PC/콘솔용 Unity 게임에서 사용되는 SRP

유니티의 최신 데이터에 따르면 2023년에 PC 및 콘솔에 출시된 Unity 게임에서 가장 많이 선택한 렌더 파이프라인은 URP입니다. 아래 그래프를 보면 이제 빌트인 렌더 파이프라인은 소수의 개발 팀만 사용한다는 사실을 알 수 있습니다.

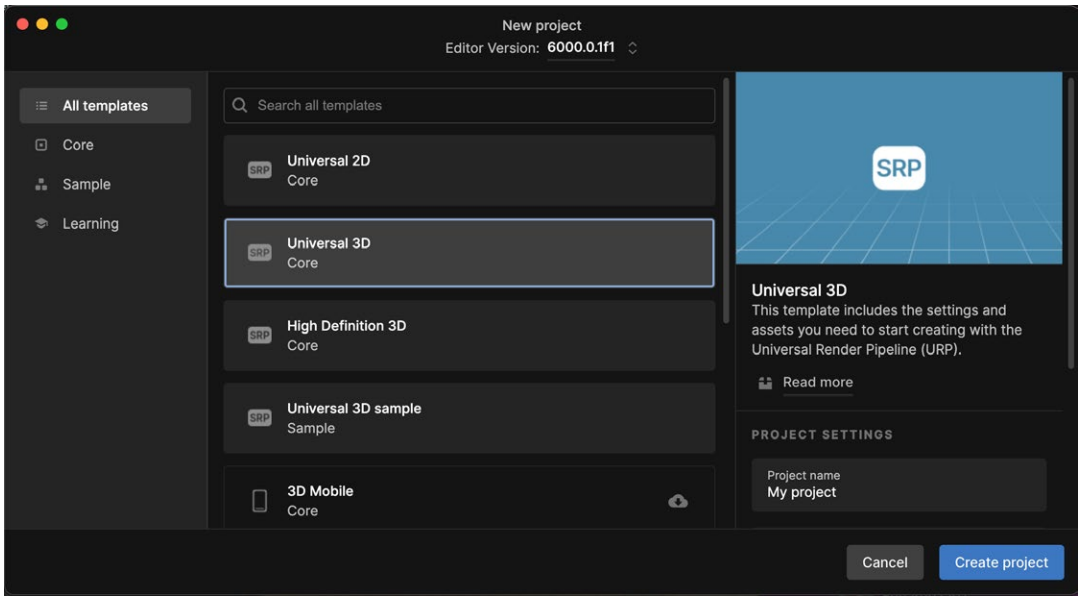


Unity에서 제공하는 다양한 파이프라인을 사용하여 출시된 게임의 비율

현재 대부분의 Unity 프로젝트는 URP 또는 HDRP로 제작되고 있지만, 빌트인 렌더 파이프라인은 Unity 6에서도 계속 사용할 수 있습니다.

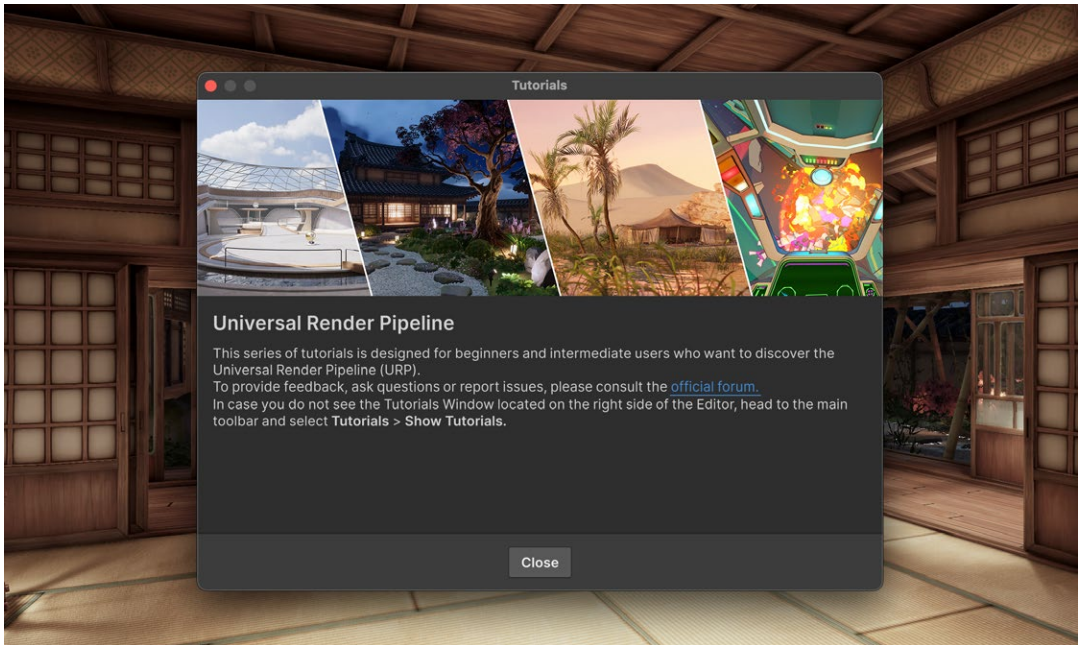
새 URP 프로젝트를 여는 방법

Unity Hub를 통해 URP를 사용하는 새 프로젝트를 엽니다. **New**를 클릭하고 창 상단의 Unity 버전이 Unity 6 이상인지 확인합니다. 프로젝트 이름과 위치를 선택한 다음 **Universal 2D** 또는 **Universal 3D** 템플릿을 선택하고 **Create**를 클릭합니다.



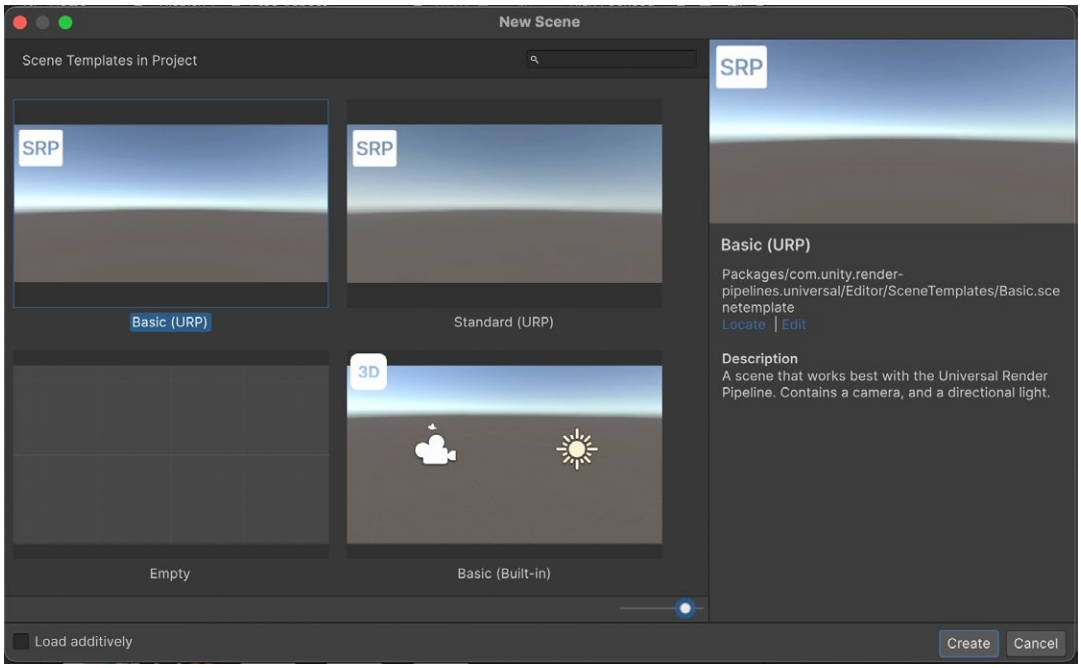
URP 템플릿을 사용해 새 프로젝트를 만드는 것이 처음이라면 템플릿을 다운로드해야 할 수 있습니다.

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



이 전자책의 마지막 부분에서 다루는 URP 3D 샘플 씬은 템플릿으로 다운로드할 수 있습니다.

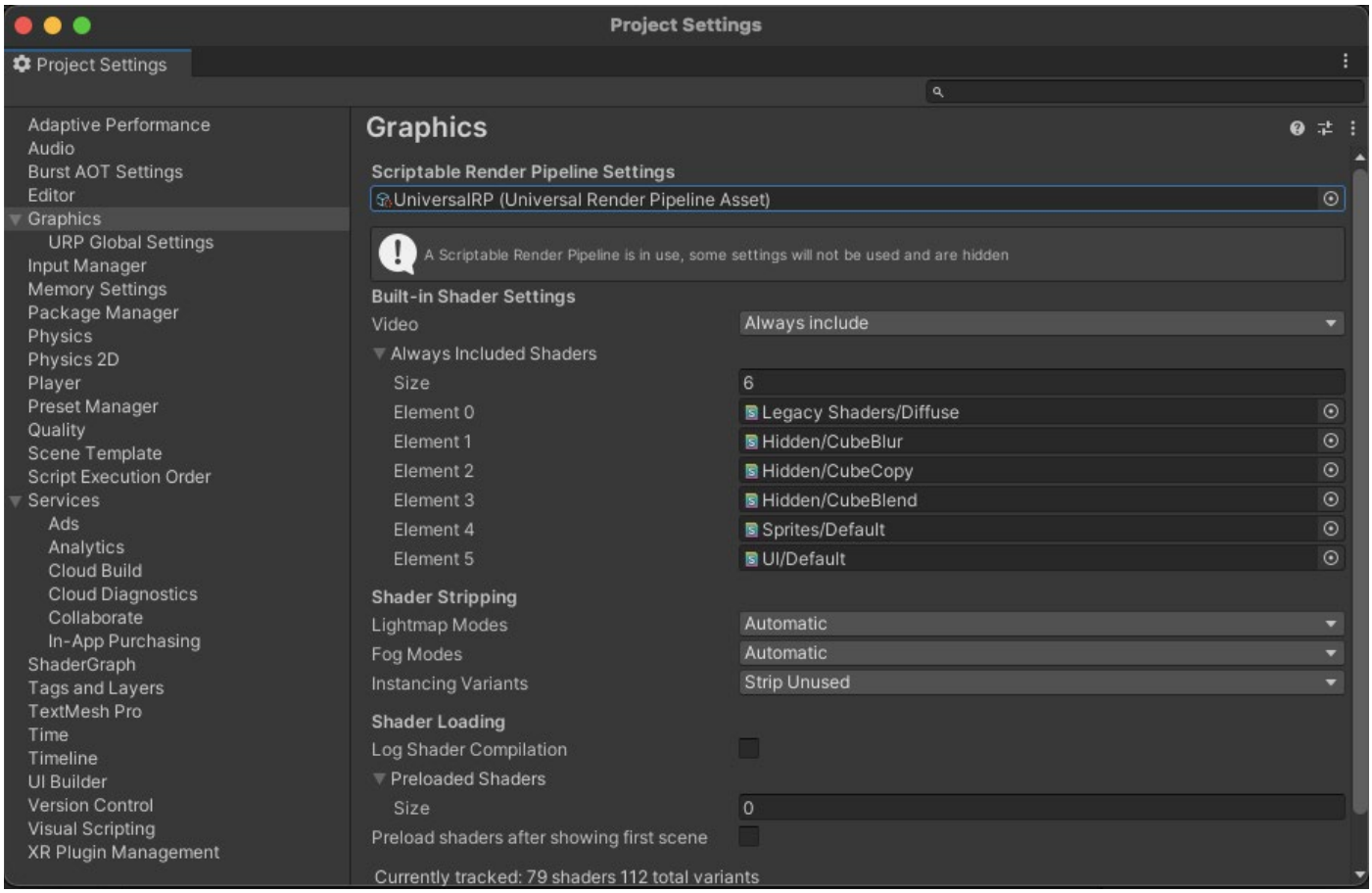
File > New Scene에서 카메라 및 방향 광원 같은 필수 게임 오브젝트로 새로운 씬을 만들 수 있으며, 미리 채워진 오브젝트로 자체 씬 템플릿을 만들 수도 있습니다. URP 씬 템플릿 기술 자료에서 자세히 알아보세요.



씬 템플릿이 표시된 New Scene 다이얼로그

Edit > Project Settings로 이동하여 **Graphics** 패널을 엽니다. 에디터 내 URP를 사용하려면 **Scriptable Render Pipeline Settings**에서 URP Asset을 선택해야 합니다. URP 에셋은 프로젝트의 글로벌 렌더링 및 품질 설정을 제어하고 렌더링 파이프라인 인스턴스를 생성합니다. 한편, 렌더링 파이프라인 인스턴스에는 임시 리소스와 렌더 파이프라인 구현이 포함되어 있습니다.

데스크톱에서는 기본 URP 에셋으로 **PC_RPAsset**이 선택되지만 **Mobile_RPAsset**으로 바꿀 수 있습니다.



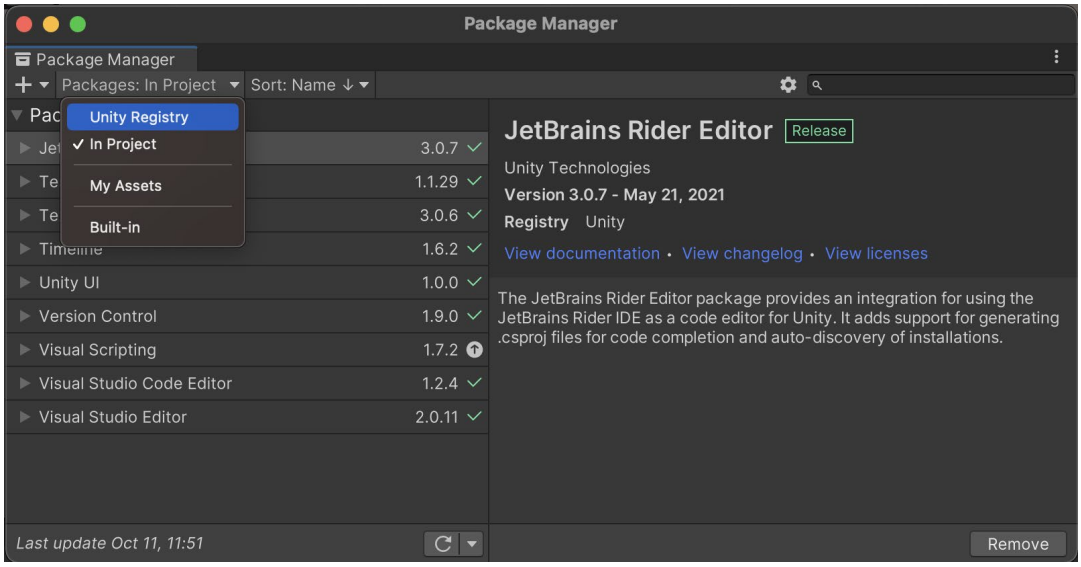
Project Settings의 Graphics 패널

이 가이드의 후반부 섹션에는 URP 에셋 설정을 조정하는 방법이 상세히 나와 있습니다.

URP를 기존 빌트인 렌더 파이프라인 프로젝트에 추가하는 방법

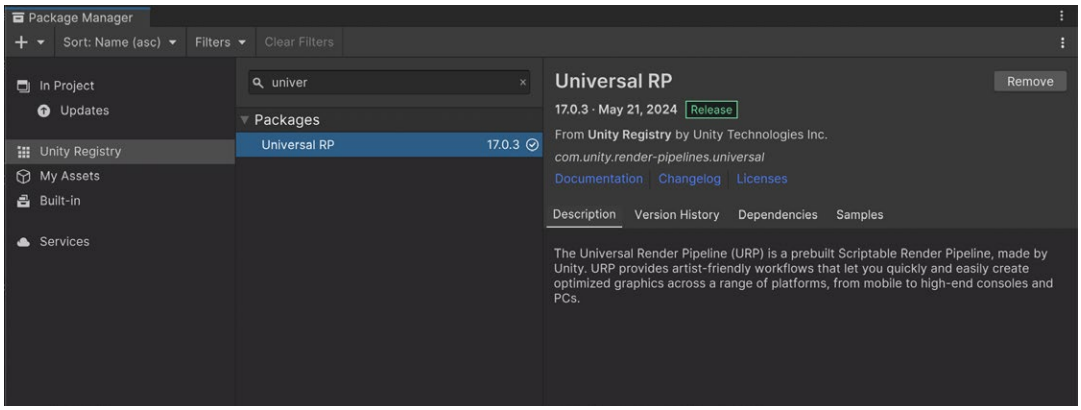
중요: 이 섹션의 단계를 따르기 전에 소스 관리를 사용하여 프로젝트를 백업하시기 바랍니다. 이 프로세스를 진행하면 에셋이 전환되며 Unity는 실행 취소 옵션을 제공하지 않습니다. 소스 관리를 사용하면 필요한 경우 이전 버전의 에셋으로 되돌릴 수 있습니다.

기존의 빌트인 렌더 파이프라인 프로젝트를 업그레이드하는 경우 **URP 패키지**가 Unity 6 이전 버전에 포함되어 있지 않기 때문에 이 패키지를 프로젝트에 추가해야 합니다.



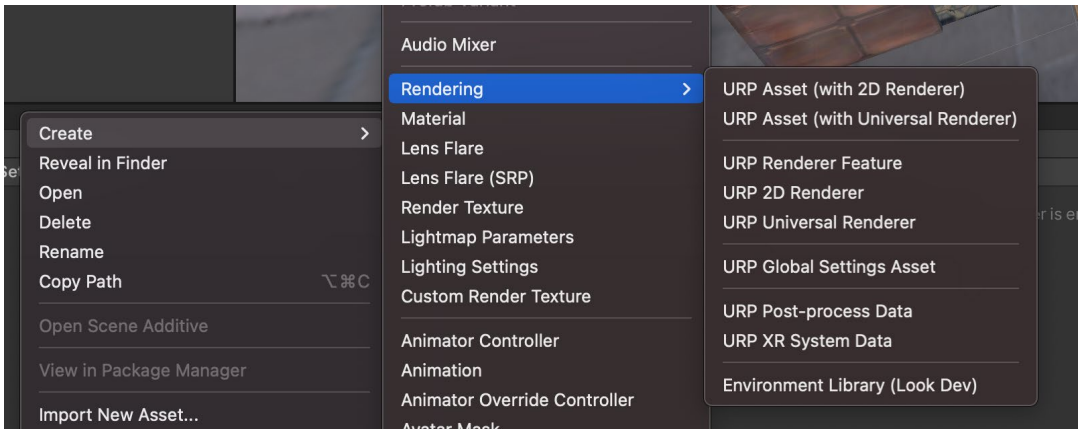
패키지 관리자에 표시된 Unity Registry 패키지

Window > Package Manager로 이동하고 **Packages** 드롭다운을 클릭하여 URP를 프로젝트에 추가합니다. **Unity Registry**, **Universal RP**를 차례로 선택해야 합니다. 개발용 컴퓨터에 URP 패키지가 아직 설치되지 않은 경우 창 오른쪽 하단의 **Download**를 클릭합니다. 그런 다음 다운로드되면 **Install**을 클릭합니다.



패키지 관리자를 통해 URP 설치

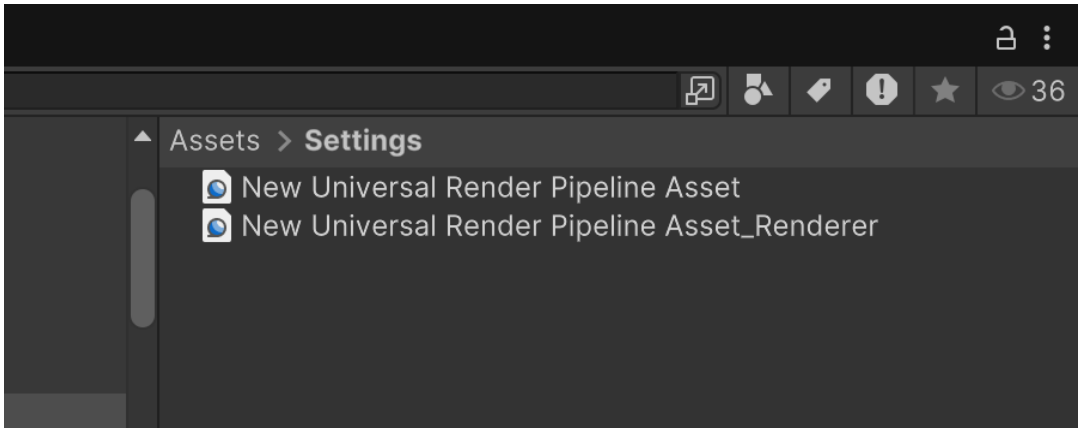
URP 에셋을 만들려면 **프로젝트(Project)** 창을 오른쪽 클릭하고 **Create > Rendering > URP Asset (with Universal Renderer)**을 선택합니다. 에셋의 이름을 지정합니다.



URP 에셋 만들기

중요: 유니버설 렌더 파이프라인 또는 3D(URP) 템플릿을 사용하여 새 프로젝트를 생성하는 경우 프로젝트에 이미 URP 에셋이 제공됩니다.

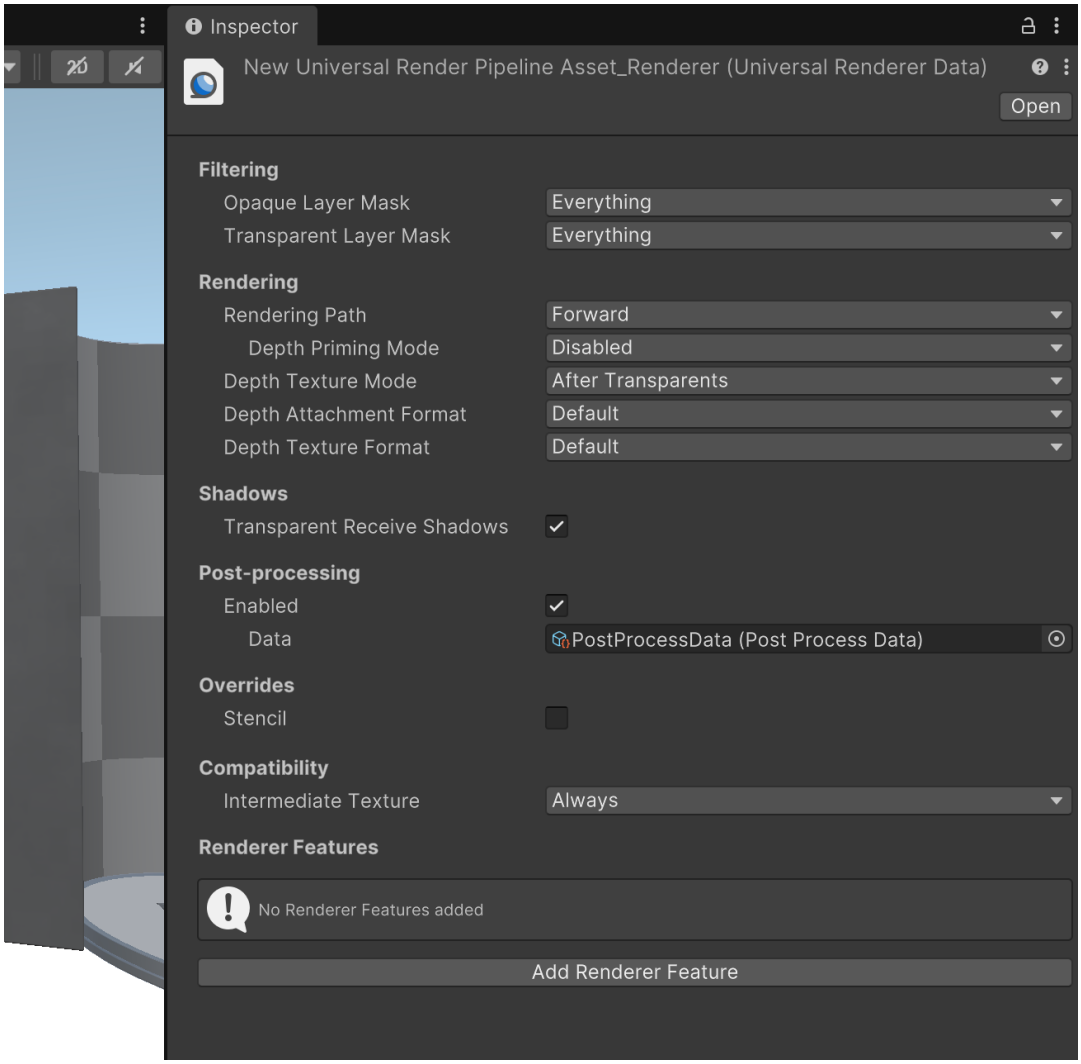
URP는 단일 URP 에셋을 만드는 것이 아니라 두 개의 파일을 사용하며, 각 파일에 Asset 확장자가 붙습니다.



각각 URP 설정 및 렌더러 데이터를 위한 URP의 두 가지 에셋

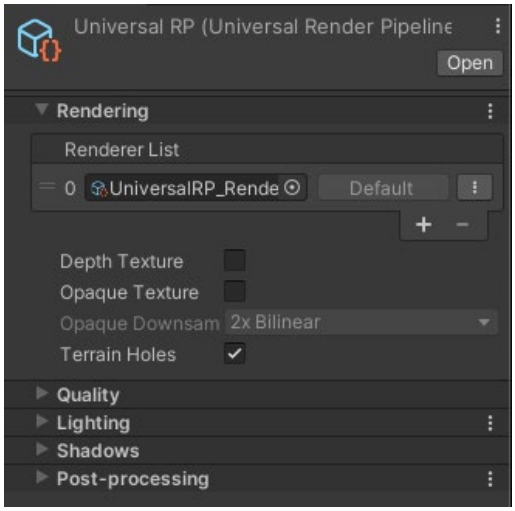
UniversalRP_Renderer 는 **렌더러 데이터 에셋**으로, 렌더러가 작동하는 레이어를 필터링하고 렌더링 파이프라인 중간에서 씬이 렌더링되는 방식을 커스터마이즈하는 데 사용됩니다. 이렇게 하면 고품질 효과를 손쉽게 만들 수 있습니다. 자세한 내용은 파이프라인 콜백 에 관한 섹션을 참조하세요.

UniversalRP_Renderer로 URP의 개괄적인 렌더링 로직과 패스를 제어할 수도 있습니다. 포워드 및 디퍼드 경로와 2D 렌더러가 지원되어 2D 광원, 2D 그림자, 광원 블렌드 스타일 등의 기능을 사용할 수 있습니다. URP를 확장하여 자체 렌더러를 만들 수도 있습니다.



UniversalRP_Renderer 데이터 에셋의 인스펙터(Inspector)

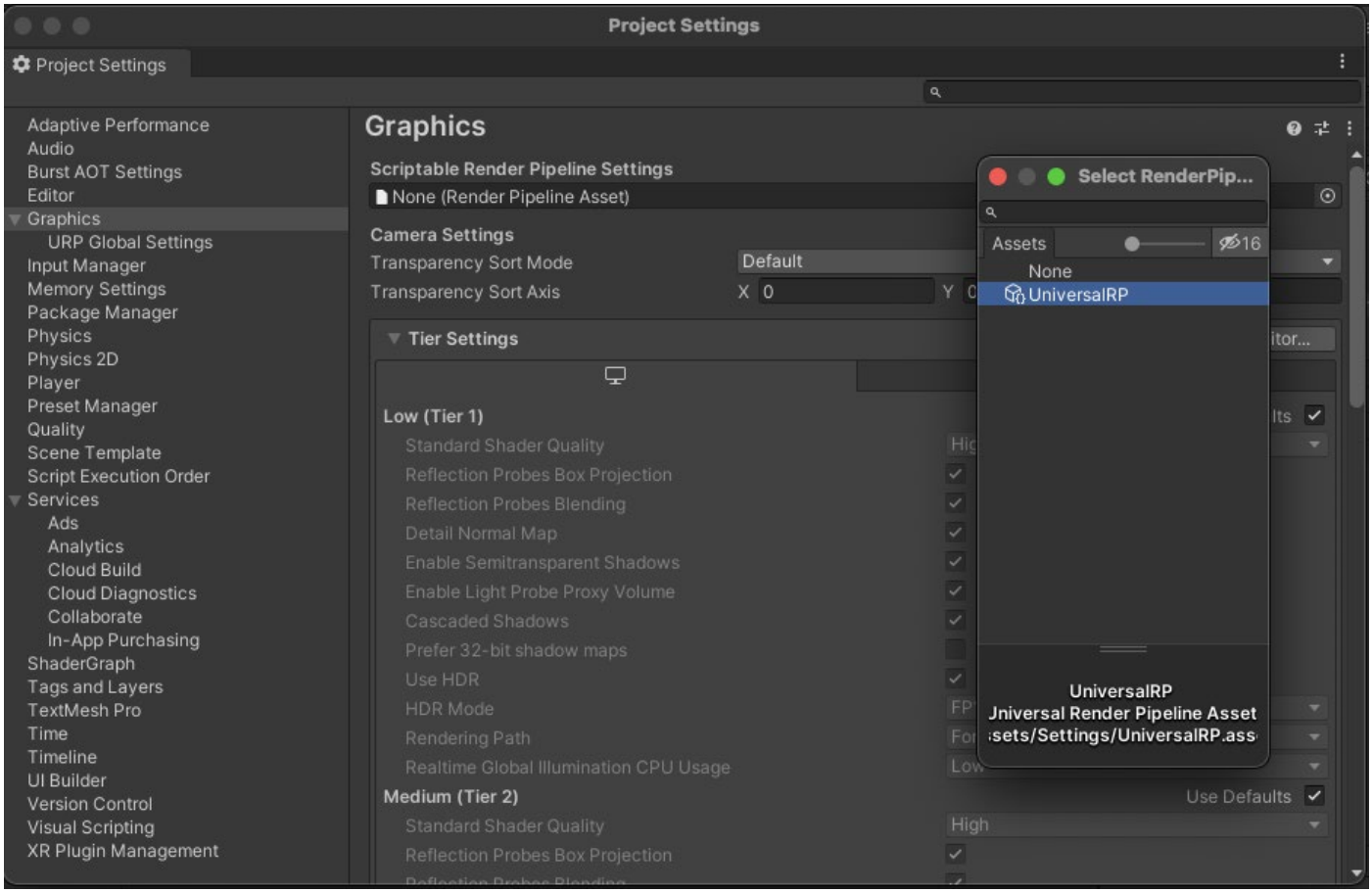
다른 URP 에셋은 품질, 조명, 그림자, 포스트 프로세싱 설정을 제어하는 옵션을 제공합니다. 다양한 URP 에셋을 사용하여 품질 설정을 제어할 수 있으며, 해당 프로세스는 이 섹션의 후반부 에 간략하게 나와 있습니다. 이 설정 에셋은 렌더러 목록을 통해 렌더러 데이터 에셋에 연결됩니다. 새 URP 에셋을 만들면 단일 항목이 포함된 렌더러 목록이 설정 에셋에 추가되며, 렌더러 데이터 에셋이 이와 동시에 생성되어 기본으로 설정됩니다. 다른 렌더러 데이터 에셋을 이 목록에 추가할 수 있습니다.



인스펙터의 URP 에셋

기본 렌더러는 씬(Scene) 뷰를 포함한 모든 카메라에 사용됩니다. 렌더러 목록에서 다른 항목을 선택하면 카메라가 기본 렌더러를 오버라이드할 수 있습니다. 필요에 따라 스크립트를 사용할 수도 있습니다.

URP 에셋을 만들기 위해 이 단계를 따르더라도, 씬 뷰 또는 게임(Game) 뷰에 이미 열려 있는 씬에는 계속 빌트인 렌더 파이프라인이 사용됩니다. 이제 마지막 단계만 마치면 URP로 전환할 수 있습니다. **Edit > Project Settings**로 이동하여 **Graphics** 패널을 엽니다. **None (Render Pipeline Asset)** 옆에 있는 작은 점을 클릭합니다. 열린 패널에서 **UniversalRP**를 선택합니다.



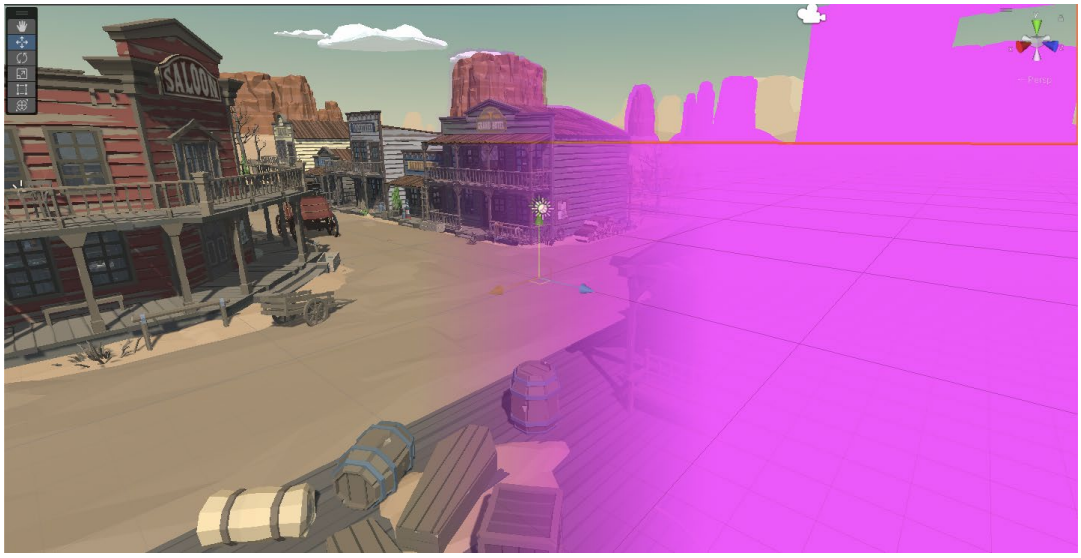
스크립터블 렌더 파이프라인 에셋 선택

전환에 관한 경고 메시지가 표시되면 **Continue**를 누릅니다.

프로젝트에 아직 콘텐츠가 없기 때문에 렌더 파이프라인 변경이 거의 즉각적으로 이루어집니다. 이제 URP를 사용할 준비가 되었습니다.

기존 프로젝트의 씬 전환

위 단계를 완료하고 나면 씬에 갑자기 마젠타색이 적용되어 있을 것입니다. 이는 빌트인 렌더 파이프라인의 머티리얼에서 사용된 셰이더가 URP에서 지원되지 않기 때문입니다. 다행히 씬을 원래 품질로 복원할 방법이 있습니다.



씬의 머티리얼이 마젠타 컬러로 표시되는 이유는 빌트인 렌더 파이프라인 기반 셰이더를 URP에서 사용할 수 있도록 전환해야 하기 때문입니다.

Window > Rendering > Render Pipeline Converter로 이동합니다. 2D 프로젝트의 경우 **Convert Built-In to 2D (URP)**를, 3D 프로젝트의 경우 **Built-In to URP**를 선택합니다. 프로젝트가 3D라고 가정하면 다음과 같이 적절한 컨버터를 선택해야 합니다.

- **Rendering Settings:** 빌트인 렌더 파이프라인 품질 설정과 가장 가깝게 매칭되는 여러 렌더 파이프라인 설정 에셋을 만들려면 이 옵션을 선택합니다. 이렇게 하면 여러 품질 수준을 더 효율적으로 테스트할 수 있습니다. 자세한 내용은 빌트인 렌더 파이프라인과 URP의 품질 옵션을 비교한 이 섹션을 참조하세요.
- **Material Upgrade**: 빌트인 렌더 파이프라인에서 URP로 머티리얼을 전환하려면 이 옵션을 사용합니다.
- **Animation Clip Converter:** 애니메이션 클립을 전환해 주는 옵션으로 Material Upgrade 컨버터가 완료되고 나면 이 컨버터가 실행됩니다.
- **Read-only Material Converter:** Unity 프로젝트에 있는 사전에 빌드된 읽기 전용 머티리얼을 전환해 주는 옵션입니다. 프로젝트를 인덱싱하고 임시 .index 파일을 생성합니다. 이 과정에는 상당한 시간이 소요될 수 있습니다.

커스텀 셰이더 전환

커스텀 셰이더는 Material Upgrade 컨버터를 사용하여 전환할 수 없습니다. 셰이더 및 Shader Graph 섹션에 커스텀 빌트인 렌더 파이프라인 셰이더를 URP로 전환하는 단계가 간략하게 설명되어 있습니다. 대부분의 경우 Shader Graph를 사용하는 것이 커스텀 셰이더를 URP로 업데이트하는 가장 빠른 방법입니다.

다음과 같이 다양한 URP 셰이더가 있습니다.

— **Universal Render Pipeline/Lit**: 이 PBR(물리 기반 렌더) 셰이더는 빌트인 스탠다드 셰이더와 유사하며 가장 현실적인 머티리얼을 표현하는 데 사용할 수 있습니다. 메탈릭(metallic)과 스페큘러 워크플로를 사용하는 모든 스탠다드 셰이더 기능을 지원합니다.

— **Universal Render Pipeline/Simple Lit**: 블린 퐁(Blinn-Phong) 모델을 사용하며 PBR 워크플로를 사용하지 않는 저사양 모바일 기기 또는 게임에 적합합니다.

— **Universal Render Pipeline/Baked Lit: 이 셰이더는** 라이트맵, 라이트 프로브, APV(적응적 프로브 볼륨)를 통해 베이크된 조명만 있으면 되는 스타일라이즈드 게임이나 앱에 사용합니다. 이 셰이더는 물리 기반이 아니며 실시간 조명이 없으므로 모든 실시간 관련 셰이더 키워드 및 배리언트가 셰이더 코드에서 제거되어 계산이 더 빨라집니다.

— **Universal Render Pipeline/Complex Lit**: 컴플렉스 릿 셰이더에는 릿 셰이더의 모든 기능이 포함되며 고급 머티리얼 기능이 추가로 제공됩니다. 이 셰이더의 일부 기능은 리소스를 많이 사용하며 Unity Shader Model 4.5 하드웨어가 필요할 수 있습니다.

— **Universal Render Pipeline/Unlit**: 조명 방정식을 사용하지 않고, GPU 성능을 효율적으로 활용하는 셰이더입니다.

— **Universal Render Pipeline/Terrain/Lit:** Terrain Tools 패키지와 함께 사용하기에 적합합니다.

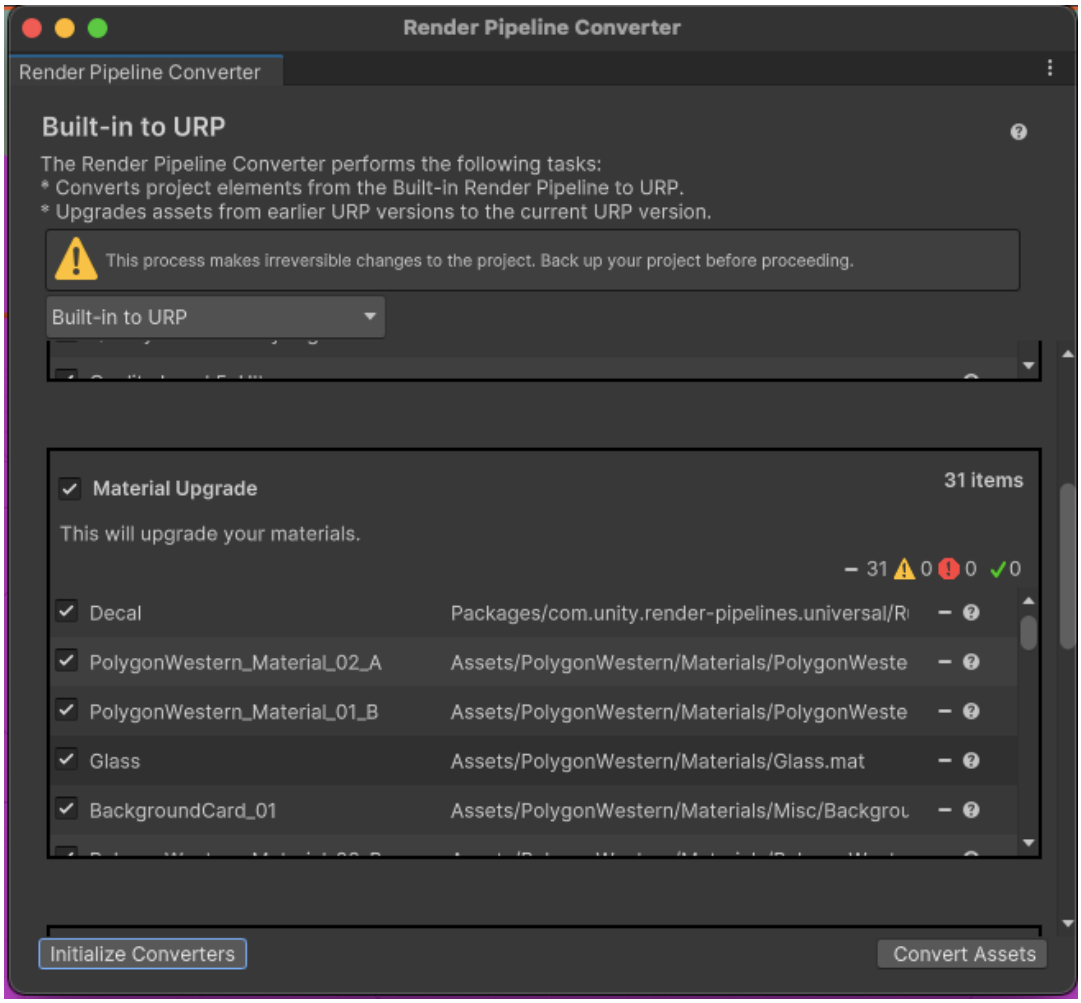
— **Universal Render Pipeline/Particles/Lit:** 이 파티클 셰이더는 PBR 조명 모델을 사용합니다.

— **Universal Render Pipeline/Particles/Unlit:** 이 언릿 파티클 셰이더는 GPU 부담이 덜합니다.

Simple Lit이 대다수의 레거시/모바일 셰이더를 대체하기는 하지만 성능은 동일하지 않습니다. 레거시/모바일 셰이더는 조명을 부분적으로만 계산하는 반면 Simple Lit은 모든 광원을 URP 에셋에 의해 정의된 대로 계산합니다.

URP 기술 자료의 이 표를 참조하면 각 URP 셰이더가 해당하는 빌트인 렌더 파이프라인 셰이더에 어떻게 매핑되는지 확인할 수 있습니다.

위 컨버터 중 하나 이상을 선택했으면 **Initialize Converters** 또는 **Initialize And Convert**를 클릭합니다. 어떤 옵션을 선택하든 프로젝트가 스캔되고 전환이 필요한 에셋이 각 컨버터 패널에 추가됩니다. **Initialize Converters**를 선택한 경우, 각각에 대해 제공된 체크박스를 사용하여 항목을 선택 해지함으로써 전환을 제한할 수 있습니다. 이 단계에서 **Convert Assets**를 클릭하면 전환 프로세스가 시작됩니다. **Initialize And Convert**를 선택하면 컨버터가 초기화된 후 자동으로 전환이 시작됩니다. 완료되고 나면 에디터에서 활성 상태인 씬을 다시 열라는 메시지가 표시될 수 있습니다.



렌더 파이프라인 컨버터

빌트인 렌더 파이프라인과 URP 간의 품질 옵션 비교

빌트인 렌더 파이프라인에서는 Very low부터 Ultra까지 여러 기본 품질 옵션을 사용할 수 있습니다. 품질 설정은 텍스처 해상도, 조명, 그림자 렌더링 등을 포함하여 씬의 정확도에 영향을 미칩니다.

Edit > Project Settings로 이동하여 **Quality** 패널을 선택합니다. 여기서 현재 품질을 선택하여 이러한 품질 옵션을 전환할 수 있습니다. 이렇게 하면 씬 뷰와 게임 뷰에서 사용되는 렌더 설정이 변경됩니다. 이 패널에서 각 품질 옵션을 편집할 수도 있습니다.

렌더 파이프라인 컨버터를 사용하고 빌트인 렌더 파이프라인에서 URP로 전환할 때 **Rendering Settings** 옵션을 선택하면 빌트인 렌더 파이프라인 품질 옵션과의 매칭을 시도하는 URP 에셋 세트가 생성됩니다. 아래 첫 번째 표에서는 저사양 설정에서 빌트인 렌더 파이프라인과 URP가 어떻게 다른지 확인할 수 있으며, 두 번째 표에서는 고사양 설정에서 차이점을 비교해 볼 수 있습니다. 빌트인 렌더 파이프라인과 URP 모두 Quality 패널을 통해 설정이 선택되었습니다. URP 에셋을 선택할 때 인스펙터를 통해 URP 에셋 설정을 사용할 수 있습니다. 자세한 내용은 URP 기술 자료를 참조하세요.

빌트인 렌더 파이프라인과 URP 비교: 저사양 설정

설정	빌트인 렌더 파이프라인	URP	URP 에셋 설정
Rendering			
Pixel Light Count	0	NA(해당되지 않음)*	NA
Anti-aliasing	Disabled	NA	Disabled
Render Scale	NA	NA	1
Real-time Reflection Probes	No	No	NA
Resolution Scaling Fixed DPI Factor	1	1	NA
VSync Count	Don’t sync	Don’t sync	NA
Depth Texture	NA	NA	No
Opaque Texture	NA	NA	No
Opaque Downsampling	NA	NA	NA
Terrain Holes	NA	NA	Yes
HDR	NA	NA	Yes
Textures			
Texture Quality	Half res	Half res	NA
Anisotropic Textures	Disabled	Disabled	NA
Texture Streaming	No	No	NA
Particles			
Soft Particles	No	NA	NA
Particle Raycast Budget	16	16	NA
Terrain			
Billboards Face Camera Position	No	No	NA

Shadows			
Shadowmask Mode	Shadowmask	Shadowmask	NA
Shadows	Disabled	NA	NA
Shadow Resolution	Low resolution	NA	NA
Shadow Projection	Stable fit	NA	NA
Shadow Distance	20	NA	NA
Shadow Near Plane Offset	3	NA	NA
Shadow Cascades	No Cascades	NA	NA
Cascade splits	NA	NA	NA
Working unit	NA	NA	NA
Depth Bias	NA	NA	NA
Normal Bias	NA	NA	NA
Soft Shadows	NA	NA	NA
Async Asset Upload			
Time Slice	2	2	NA
Buffer Size	16	16	NA
Persistent Buffer	Yes	Yes	NA
Level of Detail			
LOD Bias	0.4	0.4	NA
Maximum LOD level	0	0	NA
Meshes			
Skin Weights	4 bones	4 bones	NA
Lighting			
Main Light:	NA	NA	Per pixel
Cast Shadows	NA	NA	No
Shadow Resolution	NA	NA	NA
Additional Lights:	NA	NA	Disabled
Per Object Limit	NA	NA	NA
Cast Shadows	NA	NA	NA
Shadow Atlas Resolution	NA	NA	NA
Shadow Resolution tiers	NA	NA	NA
Cookie AtlasResolution	NA	NA	NA
Cookie AtlasFormat	NA	NA	NA
Reflection probes:	NA	NA	NA

Probe Blending	NA	NA	No
Box Projection	NA	NA	No
Post-processing			
Grading Mode	NA	NA	Low Dynamic Range
	NA	NA	16
Fast sRGB/Linear conversion	NA	NA	No

* URP에서 픽셀 광원 수는 **Additional Lights > (Per pixel) > Per Object Limit**를 사용하여 처리합니다.

빌트인 렌더 파이프라인과 URP 비교: 고사양 설정

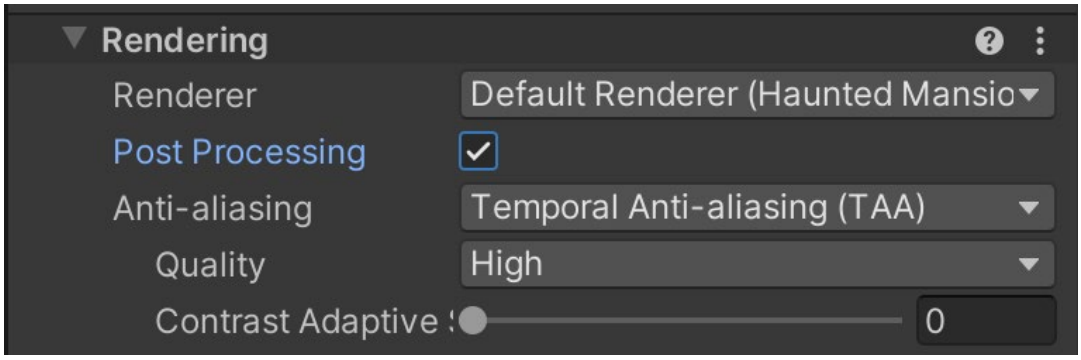
설정	빌트인 렌더 파이프라인	URP	URP 에셋 설정
Rendering			
Pixel Light Count	2	NA(해당되지 않음)	NA
Anti-aliasing	Disabled	NA	2x
Render Scale	NA	NA	1
Real-time Reflection Probes	Yes	Yes	NA
Resolution Scaling Fixed DPI Factor	1	1	NA
VSync Count	Every V Blank	Every V Blank	NA
Depth Texture	NA	NA	No
Opaque Texture	NA	NA	No
Opaque Downsampling	NA	NA	NA
Terrain Holes	NA	NA	Yes
HDR	NA	NA	Yes
Textures			
Texture Quality	Full res	Full res	NA
Anisotropic Textures	Disabled	Disabled	NA
Texture Streaming	No	No	NA
Particles			
Soft Particles	No	NA	NA
Particle Raycast Budget	256	256	NA

Terrain			
Billboards Face Camera Position	Yes	Yes	NA
Shadows			
Shadowmask Mode	Distance Shadowmask	Distance Shadowmask	NA
Shadows	Hard and Soft Shadows	NA	NA
Shadow Resolution	Medium resolution	NA	2048
Shadow Projection	Stable fit	NA	NA
Shadow Distance	40	NA	50
Shadow Near Plane Offset	3	NA	NA
Shadow Cascades	2 Cascades	NA	2
Cascade splits	33/67	NA	12.5/33.8/3.8
Working unit	Percent	Percent	Metric
Depth Bias	NA	NA	1
Normal Bias	NA	NA	1
Soft Shadows	NA	NA	Yes
Async Asset Upload			
Time Slice	2	2	NA
Buffer Size	16	16	NA
Persistent Buffer	Yes	Yes	NA
Level of Detail			
LOD Bias	1	1	NA
Maximum LOD level	0	0	NA
Meshes			
Skin Weights	Unlimited	Unlimited	NA
Lighting			
Main Light:	NA	NA	Per pixel
Cast Shadows	NA	NA	Yes
Shadow Resolution	NA	NA	NA
Additional Lights:	NA	NA	Per pixel
Per Object Limit	NA	NA	4
Cast Shadows	NA	NA	Yes
Shadow Atlas Resolution	NA	NA	2048
Shadow Resolution tiers	NA	NA	512/1024/2048
Cookie AtlasResolution	NA	NA	2048

Cookie AtlasFormat	NA	NA	Color high
Reflection probes:	NA	NA	NA
Probe Blending	NA	NA	Yes
Box Projection	NA	NA	No
Post-processing			
Grading Mode	NA	NA	Low Dynamic Range
LUT size	NA	NA	32
Fast sRGB/Linear conversion	NA	NA	No

안티앨리어싱

Unity 6의 URP에서는 **Camera > Rendering > Anti-aliasing**에서 TAA(Temporal Anti-Aliasing)를 카메라의 안티앨리어싱 옵션으로 선택할 수 있습니다.



TAA가 선택된 모습

Unity 6의 URP에서는 성능 저하 없이 전반적인 TAA 품질이 향상됩니다. 가장자리 안티앨리어싱 품질이 향상되었으며(이전 버전에서 발생하던 자연스럽지 않은 가장자리 문제 해소) 텍스처 품질을 더욱 효과적으로 유지합니다. 저품질 또는 보통 품질 SMAA 프리셋과 비교되는 품질의 결과물을 내면서도 더 뛰어난 성능을 발휘합니다.

품질 설정을 사용하는 방법

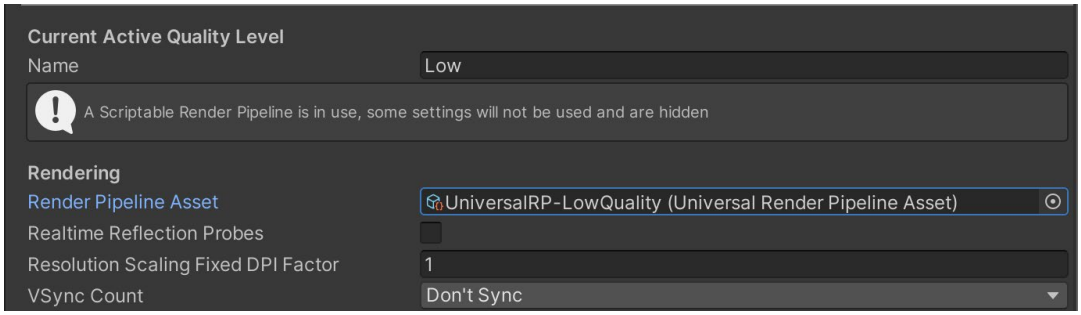
URP를 사용할 때 품질 설정은 Quality 패널과 각 URP 에셋에 대한 설정으로 나뉩니다. 다음 표에서 각 설정의 위치를 확인할 수 있습니다.

URP 사용 시 품질 설정

설정	Quality 패널	URP 에셋
Rendering		
안티앨리어싱		✅
Render Scale		✅
Resolution Scaling Fixed DPI Factor	✅	
VSync Count	✅	
Depth Texture		✅
Opaque Texture		✅
Opaque Downsampling		✅
Terrain Holes		✅
HDR		✅
Textures		
Texture Quality	✅	
Anisotropic Textures	✅	
Texture Streaming	✅	
Particles		
Particle Raycast Budget	✅	
Terrain		
Billboards Face Camera Position	✅	
Shadows		
Shadowmask Mode	✅	
Shadow Resolution		✅
Shadow Distance		✅
Shadow Cascades		✅
Cascade splits		✅
Working unit		✅
Depth Bias		✅
Normal Bias		✅
Soft Shadows		✅

Async Asset Upload		
Time Slice	✓	
Buffer Size	✓	
Persistent Buffer	✓	
Level of Detail		
LOD Bias	✓	
Maximum LOD level	✓	
Meshes		
Skin Weights	✓	
Lighting		
Main Light:		✓
— Cast Shadows		✓
— Shadow Resolution		✓
Additional Lights:		✓
— Per Object Limit		✓
— Cast Shadows		✓
— Shadow Atlas Resolution		✓
— Shadow Resolution tiers		✓
— Cookie AtlasResolution		✓
— Cookie AtlasFormat		✓
Reflection probes:	✓	
— Probe Blending		✓
— Box Projection		✓
Post-processing		
Grading Mode		✓
LUT size		✓
Fast sRGB/Linear conversion		✓

품질 옵션을 전환하는 경우 **Project Settings**를 통해 **Quality** 패널에서 Render Pipeline Asset의 **Quality Level**을 선택합니다. 품질 수준이 설정되지 않은 경우 Graphics 패널에서 Scriptable Render Pipeline Asset으로 설정된 항목이 Render Pipeline Asset의 기본값으로 설정됩니다. 이렇게 되면 URP 에셋의 품질 설정을 조정하려 할 때 혼란이 생길 수 있습니다. 이를테면 URP 에셋에서 설정된 품질 수준이 현재 씬 뷰 및 게임 뷰에서 사용되고 있다고 착각할 수 있습니다.

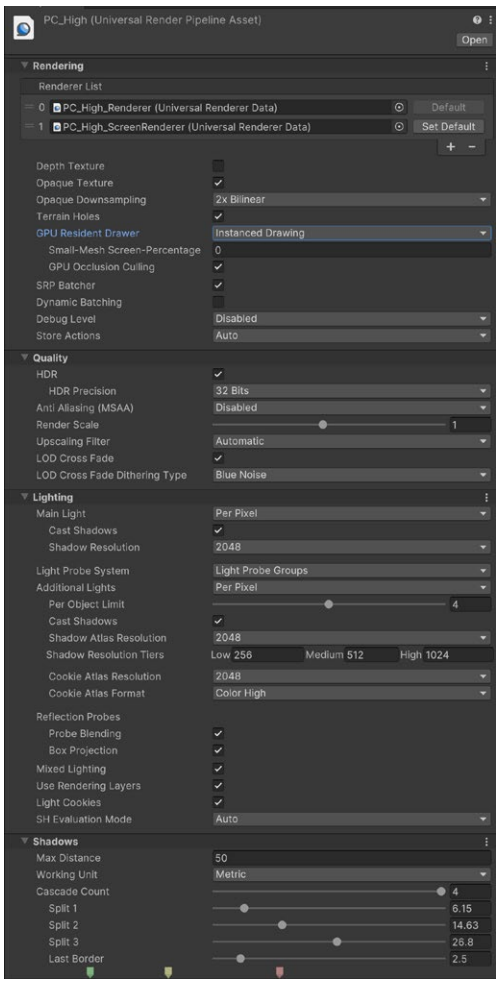


렌더 파이프라인 에셋의 품질 수준 설정

URP 에셋 수정

이 이미지에서는 인스펙터에 표시된 URP 에셋에서 사용 가능한 모든 설정을 확인할 수 있습니다. 각 설정에 대한 자세한 내용은 URP 기술 자료를 참고하세요.

참고: URP 2D 렌더러를 활성화한 경우 URP 에셋의 3D 렌더링과 관련된 일부 옵션은 최종 앱이나 게임에 영향을 미치지 않습니다. **Edit > Project Settings > Graphics**를 통해 **Scriptable Render Pipeline Settings**에서 2D 렌더러 에셋을 사용할 수 있습니다.



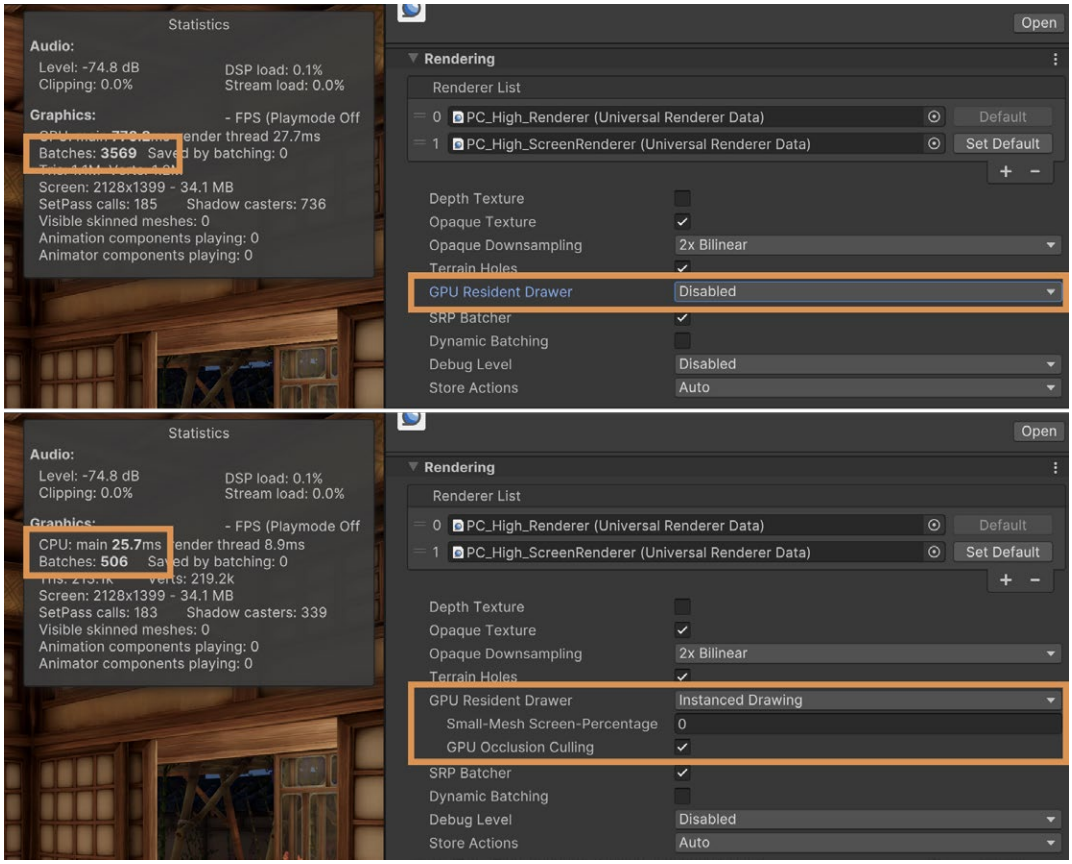
인스펙터의 URP 에셋

URP 에셋의 Quality 패널에서 HDR 포맷을 64비트로 설정하여 정확도를 높일 수 있습니다. 그러나 이렇게 하면 성능이 저하되고 추가 메모리가 필요하므로 저사양 하드웨어에서는 이 설정을 사용하지 않는 것이 좋습니다.

Quality 패널에서는 LOD Cross Fade도 활성화할 수 있습니다. LOD는 멀리 있는 메시를 렌더링하는 데 필요한 GPU 비용을 줄이는 기술입니다. 카메라의 움직임에 따라 LOD가 스왑되면서 적절한 수준의 품질을 제공합니다. LOD Cross Fade는 여러 LOD 지오메트리 간의 전환을 부드럽게 만들어 주며, 스왑 도중에 발생하는 거친 스내핑과 팝핑(popping) 현상을 방지합니다.

GPU 상주 드로어 및 GPU 오클루전 컬링

GPU 상주 드로어는 Unity 6의 새로운 기능으로 URP 에셋의 **Rendering** 섹션을 통해 사용할 수 있습니다.



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

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

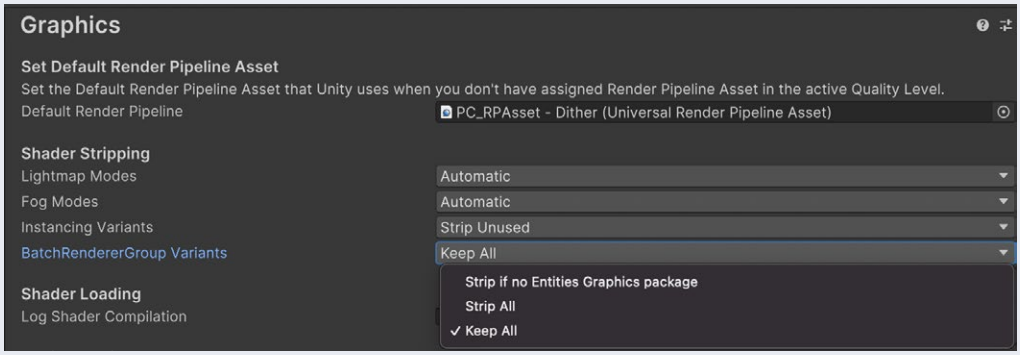
GPU 상주 드로어는 CPU 시간을 최적화하도록 설계된 GPU 기반 렌더링 시스템입니다. GPU 상주 드로어를 사용하면 게임 오브젝트가 BatchRenderGroup API를 활용하므로 배칭이 빨라지고 CPU 성능이 향상됩니다.

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.

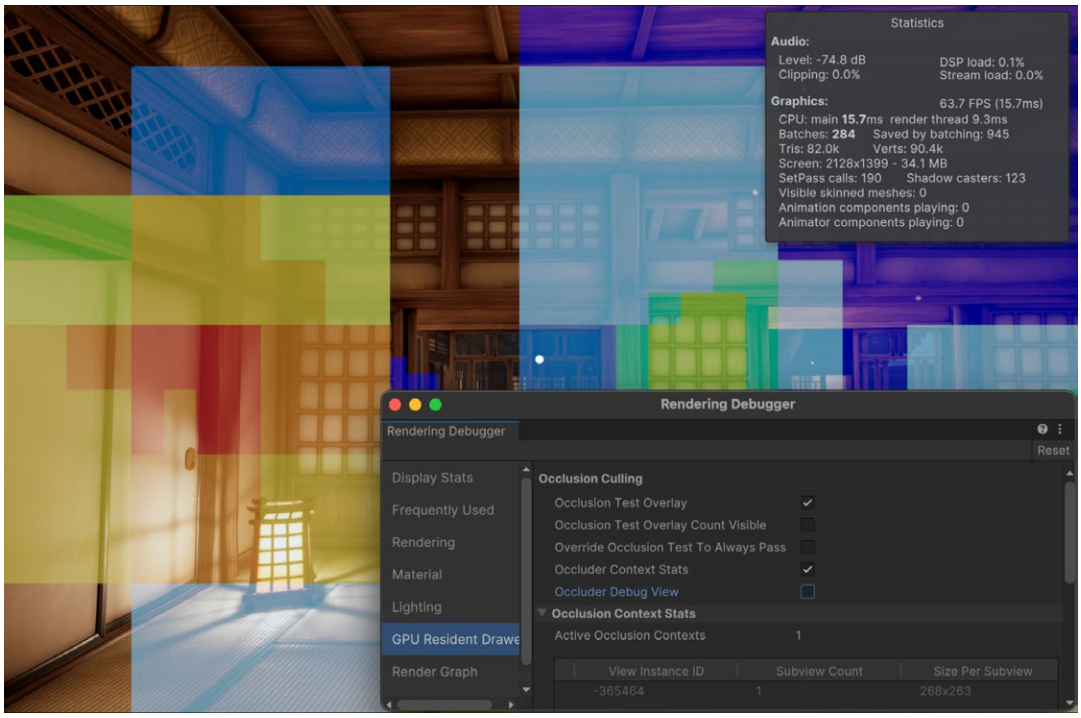
향상되는 정도는 씬의 규모와 사용되는 인스턴싱의 양에 따라 달라집니다. 인스턴싱할 수 있는 오브젝트를 많이 렌더링할수록 더 많은 이점을 얻을 수 있습니다.

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.

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.

URP의 조명

이 섹션에서는 Unity 6의 URP에서 조명이 작동하는 방식과 더불어 그래픽 정확도와 성능 간의 균형을 맞추기 위해 사용할 수 있는 기법을 알아봅니다.

Unity의 조명을 처음 접한다면 다음 리소스를 먼저 살펴보세요.

- 조명 기술 자료
- 게임 환경에 조명 적용
- Unity의 실시간 조명
- URP 및 GPU 라이트매퍼를 통한 광원 활용

렌더러 선택

URP는 다양한 렌더링 기법을 제공하며 각 기법에는 고유의 장단점이 있습니다. 프로젝트의 요구 사항과 제약에 따라 렌더링 기법을 선택하면 됩니다. URP의 포워드, 포워드+, 디퍼드 렌더링의 차이점을 살펴보겠습니다.

포워드 렌더링

작동 방식	장점	단점
포워드 렌더링은 전통적인 렌더링 기법으로, 씬 안에 있는 각 오브젝트를 개별적으로 렌더링하고 각각의 픽셀을 개별적으로 계산합니다. 즉, 화면의 모든 픽셀마다 Unity가 해당 픽셀의 최종 색상에 영향을 주는 씬의 각 오브젝트의 조명과 셰이딩을 계산합니다.	비교적 직관적으로 이해할 수 있는 워크플로이며, 광원 수가 적고 머티리얼이 단순한 씬에서 효과적입니다.	광원 수가 많고 머티리얼이 복잡한 씬을 렌더링하는 데는 효율적이지 않을 수 있습니다. 각 광원이 씬의 여러 패스를 거쳐야 하므로 렌더링 오버헤드가 늘어나기 때문입니다.

포워드+ 렌더링

작동 방식	장점	단점
포워드+ 렌더링은 포워드 렌더링의 한계를 일부 개선한 기법으로, 많은 수의 광원을 더 효과적으로 처리할 수 있습니다. 포워드+ 렌더링에서는 광원을 클러스터로 그룹화하고, 조명을 계산할 때는 각 광원마다 씬의 모든 오브젝트를 처리하는 대신 각 클러스터 내 오브젝트만 고려합니다.	특히 광원의 수가 많은 씬에서 포워드 렌더링에 비해 성능이 뛰어납니다. 각 광원에서 고려해야 하는 오브젝트의 수를 줄여 하드웨어 리소스를 더 효율적으로 활용할 수 있습니다.	광원 수가 매우 많거나 상당히 복잡한 씬에서는 그 효과가 여전히 제한적일 수 있습니다. 또한 포워드+ 렌더링을 구현하려면 포워드 렌더링에 비해 더 많은 작업과 최적화가 필요할 수 있습니다.

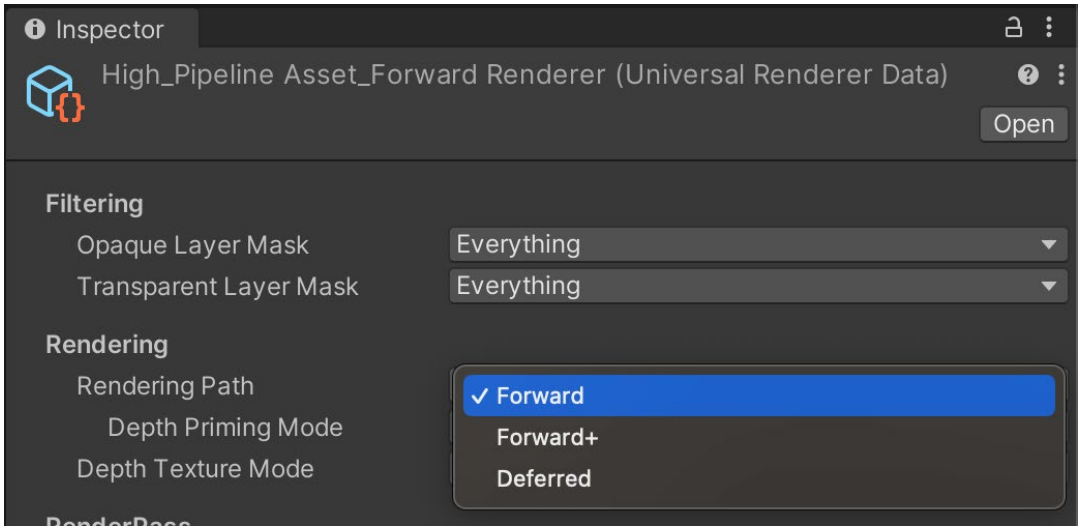
URP 디퍼드 렌더링

작동 방식	장점	단점
디퍼드 렌더링은 지오메트리 렌더링 프로세스에서 조명 및 셰이딩 계산을 분리합니다. 먼저 각 픽셀의 위치, 노멀, 머티리얼 프로퍼티 정보를 저장하는 G 버퍼 같은 버퍼 세트로 지오메트리를 렌더링합니다. 그런 다음 버퍼에 저장된 정보를 바탕으로 픽셀 단위로 조명을 계산합니다.	오브젝트 단위가 아니라 픽셀 단위로 조명을 계산하므로 광원 수가 많거나 머티리얼이 복잡한 씬에서 매우 효율적입니다. 성능 저하를 최소화하면서 많은 수의 광원을 렌더링할 수 있습니다.	디퍼드 렌더링에도 추가 버퍼를 저장하기 위한 메모리 사용량 증가, 투명 오브젝트 관련 제한, 볼류메트릭 조명 같은 특정 유형의 조명 효과를 처리할 때의 어려움 등 여러 단점이 있습니다.

아래 표에서 세 가지 렌더링 옵션에 대해 자세히 알아볼 수 있습니다.

기능	포워드	포워드+	디퍼드
오브젝트당 사용 가능한 최대 실시간 광원 수	9	무제한, 카메라별 제한 적용	무제한
픽셀당 노멀 인코딩	인코딩 없음 (정확한 노멀 값)	인코딩 없음 (정확한 노멀 값)	두 가지 옵션: — G버퍼의 노멀 양자화 (정확도 손실, 성능 향상) — 팔면체 인코딩(정확한 노멀, 모바일 GPU에 상당한 성능 영향을 미칠 수 있음) 자세한 내용은 G버퍼의 노멀 인코딩(영문)을 참조하세요.
MSAA	지원	지원	No
GPU 상주 드로어	No	Yes	지원 안 함
버텍스 조명	지원	지원 안 함	지원 안 함
카메라 스태킹	지원	지원	제한적 지원: Unity는 디퍼드 렌더링 경로를 사용하여 기본 카메라만 렌더링합니다. Unity는 포워드 렌더링 경로를 사용하여 모든 오버레이 카메라를 렌더링합니다.

Universal Renderer Data 에셋으로 렌더링 경로를 변경할 수 있습니다.



렌더링 경로 선택

포워드+를 사용하면 다음과 같은 URP 에셋 조명 설정이 오버라이드됩니다.

— **Main light**: 선택한 값과 관계없이 이 프로퍼티의 값은 **Per Pixel**입니다.

— **Additional lights**: 선택한 값과 관계없이 이 프로퍼티의 값은 **Per Pixel**입니다.

— **Additional Lights > Per Object Limit**: Unity는 이 프로퍼티를 무시합니다.

— **Reflection Probes > Probe Blending**: 반사 프로브 블렌딩은 항상 활성화됩니다.

광원 설정

아래에 나열된 세 위치에서 광원 프로퍼티를 설정합니다.

1. **Window > Rendering > Lighting:** 이 패널에서는 라이트매핑과 환경 설정을 구성하고 실시간 및 베이크된 라이트맵을 볼 수 있습니다. 이는 빌트인 렌더 파이프라인에서 URP로 전환할 때 변경되지 않습니다.
2. **광원 인스펙터:** 빌트인 렌더 파이프라인과 URP의 인스펙터 간에는 큰 차이가 있습니다. 자세한 내용은 광원 인스펙터 섹션을 참조하세요.
3. **URP 에셋 인스펙터:** 그림자를 설정하는 주요 공간입니다. URP의 조명은 이 패널에서 선택된 설정에 크게 좌우됩니다.

빌트인 렌더 파이프라인에서는 **Edit > Project Settings > Quality**를 통해 품질 설정을 처리합니다. URP에서 품질 설정은 URP 에셋 설정에 따라 달라지며 **Quality** 패널을 사용하여 바꿀 수 있습니다(품질 설정 섹션 참조).

여기서 중점이 되는 것은 조명이므로, 다음 표의 셰이더를 사용하는 머티리얼에 메서드가 적용됩니다.

릿 씬을 위한 URP 셰이더

셰이더	설명
Complex Lit	이 셰이더에는 Lit 셰이더의 모든 기능이 있습니다. 자동차 등에 메탈릭 광택을 연출하는 Clear Coat 옵션을 사용할 때 이 셰이더를 활용해 보세요. 스페큘러 반사는 베이스 레이어를 위해 한 번, 베이스 레이어 상단의 얇은 투명 레이어를 시뮬레이션하기 위해 또 한 번, 총 두 번 계산됩니다.
Lit	Lit 셰이더를 사용하면 돌, 나무, 유리, 플라스틱, 금속 등의 실제 표면을 사실적인 품질로 렌더링할 수 있습니다. 광원 수준과 반사가 실물처럼 보이며 밝은 햇빛에서 어두운 동굴에 이르기까지 다양한 조명 상태에서 반응합니다. 조명을 사용하는 대부분의 머티리얼을 위한 기본 옵션으로, 베이크, 혼합, 실시간 조명을 지원하며 포워드 또는 디퍼드 렌더링과 함께 작동합니다. Lit 셰이더는 PBS(물리 기반 셰이딩) 모델이며 복잡한 셰이딩 계산이 이뤄지므로, 저사양 모바일 하드웨어에서는 이 셰이더를 사용하지 않는 것이 좋습니다.
Simple Lit	이 셰이더는 물리 기반이 아니며, 에너지 비보존형 블린 퐁 셰이딩 모델을 사용하여 비교적 덜 사실적인 결과를 만듭니다. 하지만 뛰어난 시각적 형상을 구현할 수 있으므로 저사양 모바일 기기를 타게팅할 때 물리 기반이 아닌 프로젝트에서 사용하기에 적합합니다.
Baked Lit	이 셰이더는 동적 오브젝트나 실시간 광원, 동적 그림자에 영향을 받지 않는 원거리 정적 오브젝트를 포함하여 실시간 조명을 지원하지 않아도 되는 오브젝트에 사용해 성능을 향상시킵니다.

Lit 또는 Simple Lit 선택

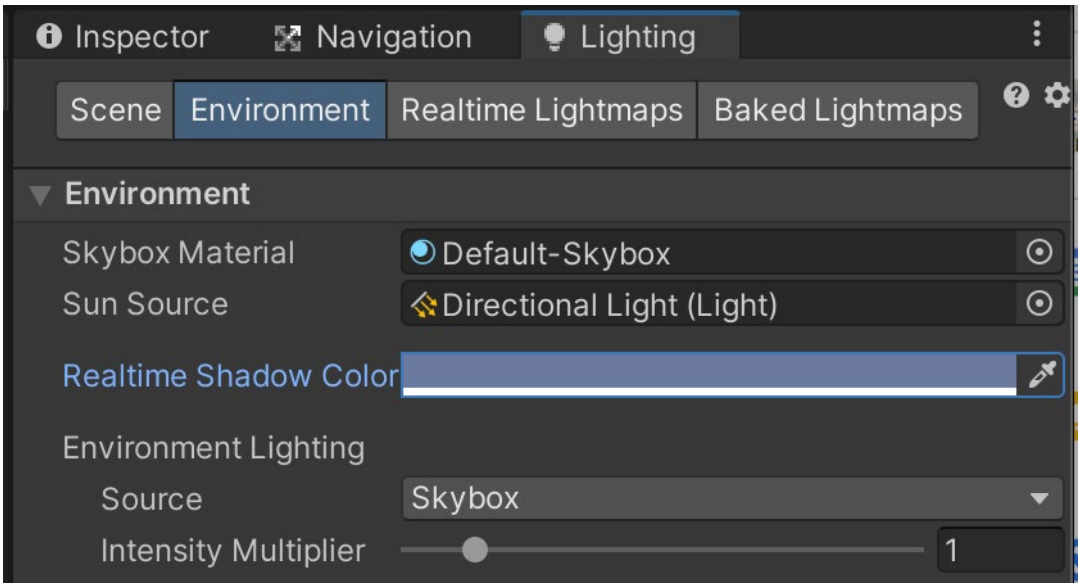
Lit 셰이더와 Simple Lit 셰이더 중 하나를 선택할 때는 대체로 아트를 고려하게 됩니다. Lit 셰이더를 사용하면 사실적인 렌더링 결과물을 쉽게 얻을 수 있지만, 스타일라이즈드 렌더링이 필요하다면 Simple Lit 셰이더로 더 뛰어난 결과를 낼 수 있습니다.

다양한 셰이더를 사용하여 렌더링한 씬 비교: 왼쪽 상단 이미지는 Lit 셰이더, 오른쪽 상단은 Simple Lit 셰이더, 하단 이미지는 Baked Lit 셰이더 사용

커스텀 셰이더를 작성하거나 Shader Graph를 사용하여 자체 커스텀 조명 모델을 구현할 수 있습니다(추가 툴 챕터 참고).

조명 개요

URP에서 광원은 메인 광원 및 추가 광원으로 나뉩니다. 메인 광원 프로퍼티의 설정은 방향 광원에 영향을 줍니다. 가장 밝은 광원이나 **Window > Rendering > Lighting > Environment > Sun Source**에서 설정한 광원이 Main Light가 됩니다.



Sun Source 설정

본 가이드 후반부에서는 URP 에셋 설정을 사용하여 동적 광원 수를 설정하는 방법을 알아봅니다. 동적 광원 수는 광원당 오브젝트 제한을 통해 오브젝트에 영향을 주며, URP 포워드 렌더러에서는 최대 8개로 제한되고 포워드+ 및 디퍼드 렌더러에서는 제한이 없습니다. 카메라당 허용되는 동적 광원 수는 하드웨어 종류에 따라 다음과 같이 제한될 수도 있습니다.

— **데스크톱 및 콘솔 플랫폼**: 메인 광원 1개, 추가 광원 256개

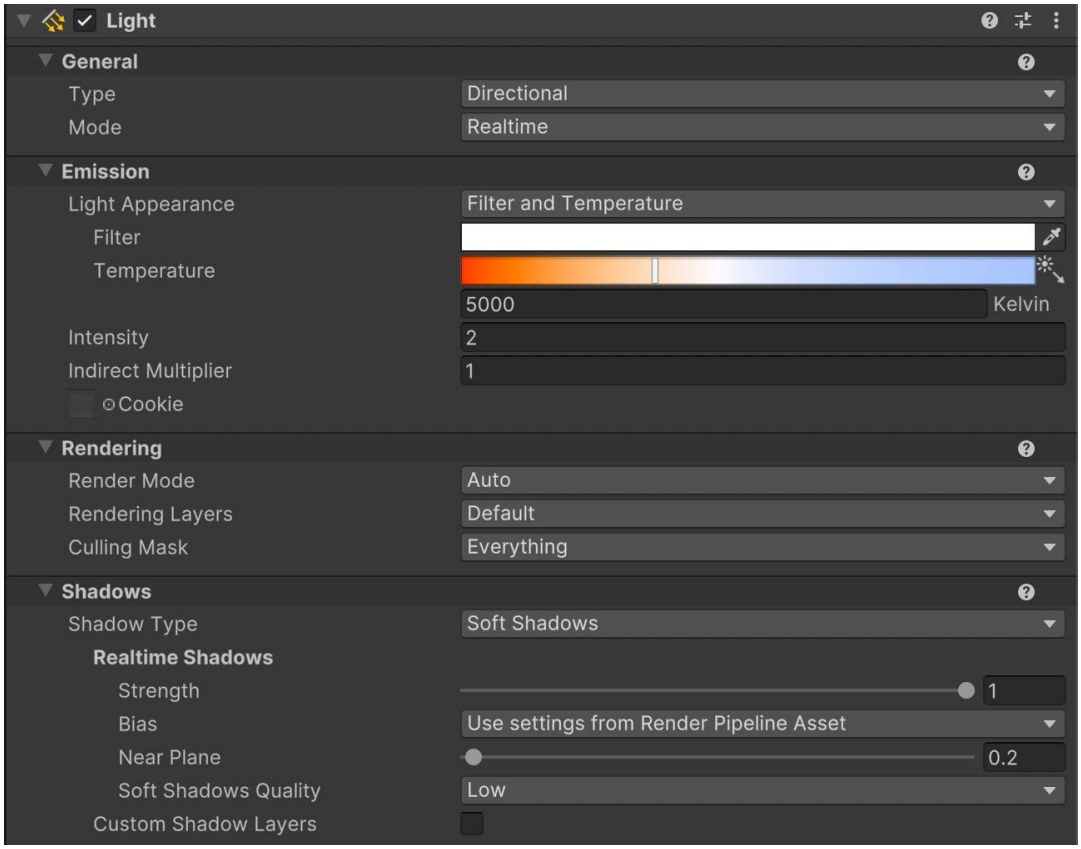
— **모바일 플랫폼**: 메인 광원 1개, 추가 광원 32개

— **OpenGL ES 3.0 및 이전 버전**: 메인 광원 1개, 추가 광원 16개

광원 인스펙터

광원 인스펙터는 조명을 설정할 수 있는 세 가지 방법 중 하나입니다.

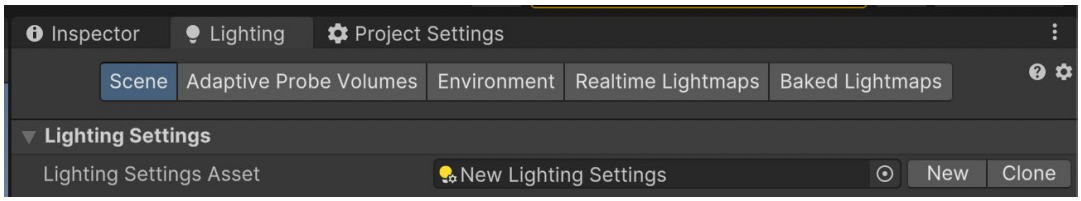
URP에서 사용할 수 있는 광원 프로퍼티에는 방향(Directional), 스폿(Spot), 점(Point), 면(Area)이 있으나, 면 광원은 Baked Indirect 모드에서만 사용할 수 있습니다. 자세한 내용은 광원 모드 섹션을 참조하세요.



URP의 광원 인스펙터 패널

위 이미지에서는 광원 인스펙터에서 광원 프로퍼티가 어떻게 표시되는지 확인할 수 있습니다. URP 버전에는 광원이 방향인지 점인지에 따라 4가지 컨트롤 그룹이 표시되며 스폿 및 면 광원에 관한 추가 셰이프 그룹도 있습니다.

새 씬에 조명 적용

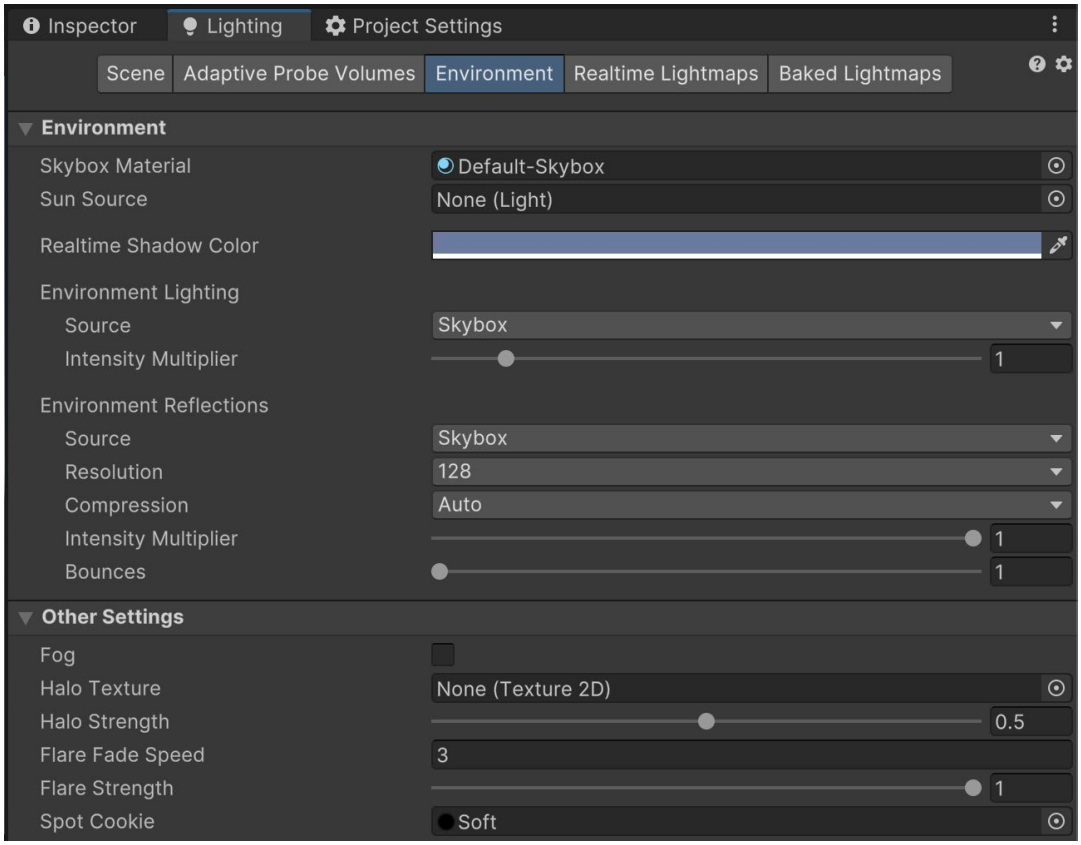


Lighting Settings 에셋 생성

URP의 새 씬에 조명을 적용하려면 가장 먼저 새 조명 설정 에셋을 만들어야 합니다(위 이미지 참조). **Window > Rendering > Lighting**을 열고 **Scene** 탭에서 **New Lighting Settings** 를 클릭하여 새 에셋의 이름을 지정합니다. 이제 Lighting 패널에서 적용한 설정이 이 에셋에 저장됩니다. Lighting Settings 에셋을 변경해 설정을 변경할 수 있습니다.

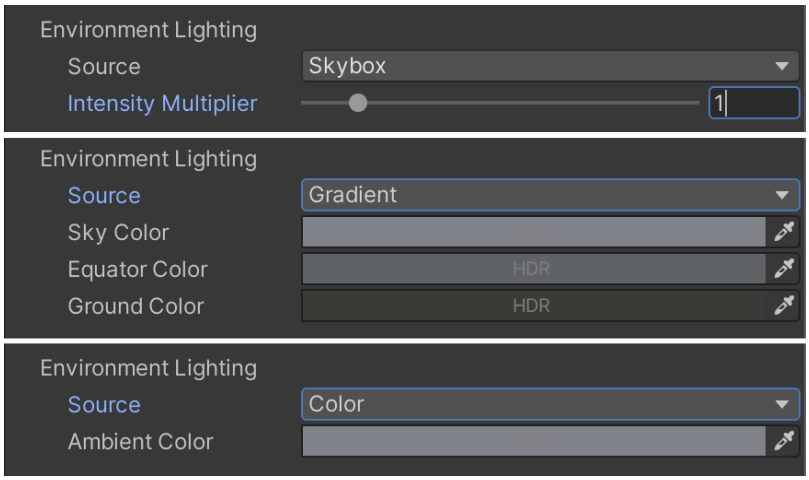
Ambient 또는 Environment 조명

메인 주변광은 **Window > Rendering > Lighting > Environment**를 통해 액세스할 수 있는 패널에서 계산됩니다.



Environment 패널에서 사용 가능한 조명 설정

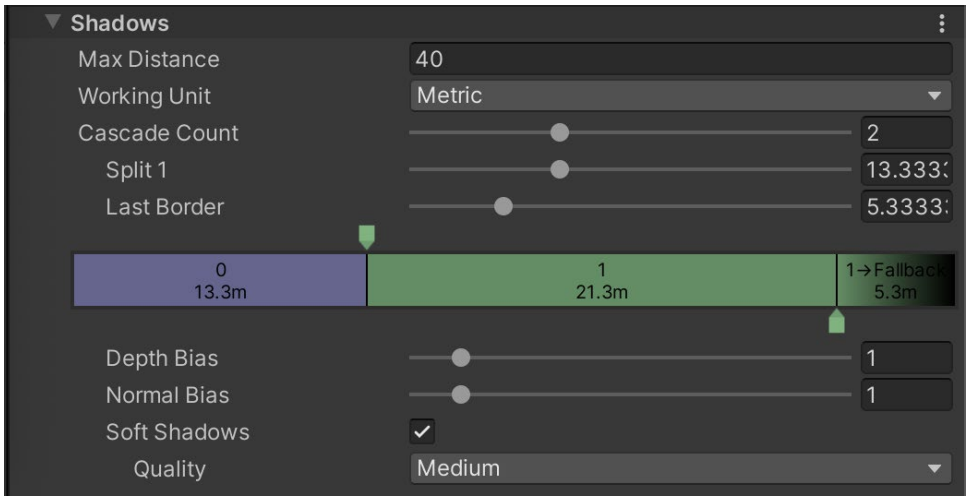
Environment Lighting에서 씬의 스카이박스를 사용하도록 설정할 수 있으며 강도 또는 그레디언트, 색상을 조정하는 옵션도 있습니다. Gradient와 Color 모드만 실시간으로 업데이트됩니다. Skybox 모드는 하늘의 앰비언트 프로브를 계산하기 위해 온디맨드 베이크가 필요합니다.



Environment Lighting 옵션

그림자

앞서 설명했듯이 URP를 사용할 때는 렌더러 데이터 오브젝트와 렌더 파이프라인 에셋이 필요합니다. URP 프로젝트를 설정하는 방법을 설명하는 섹션에서는 렌더 파이프라인 에셋을 통해 씬을 확인하는 방법을 다루며, 이를 사용하여 그림자 정확도를 정의할 수 있습니다.



URP 에셋

메인 광원 그림자 해상도

URP 에셋의 조명 및 그림자 그룹은 씬의 그림자를 설정하기 위한 핵심입니다. 먼저 **Main Light Shadow**를 **Disabled** 또는 **Per Pixel**로 설정한 다음 체크박스로 이동하여 **Cast Shadows**를 활성화합니다. 마지막으로 섀도우 맵 해상도를 설정합니다.

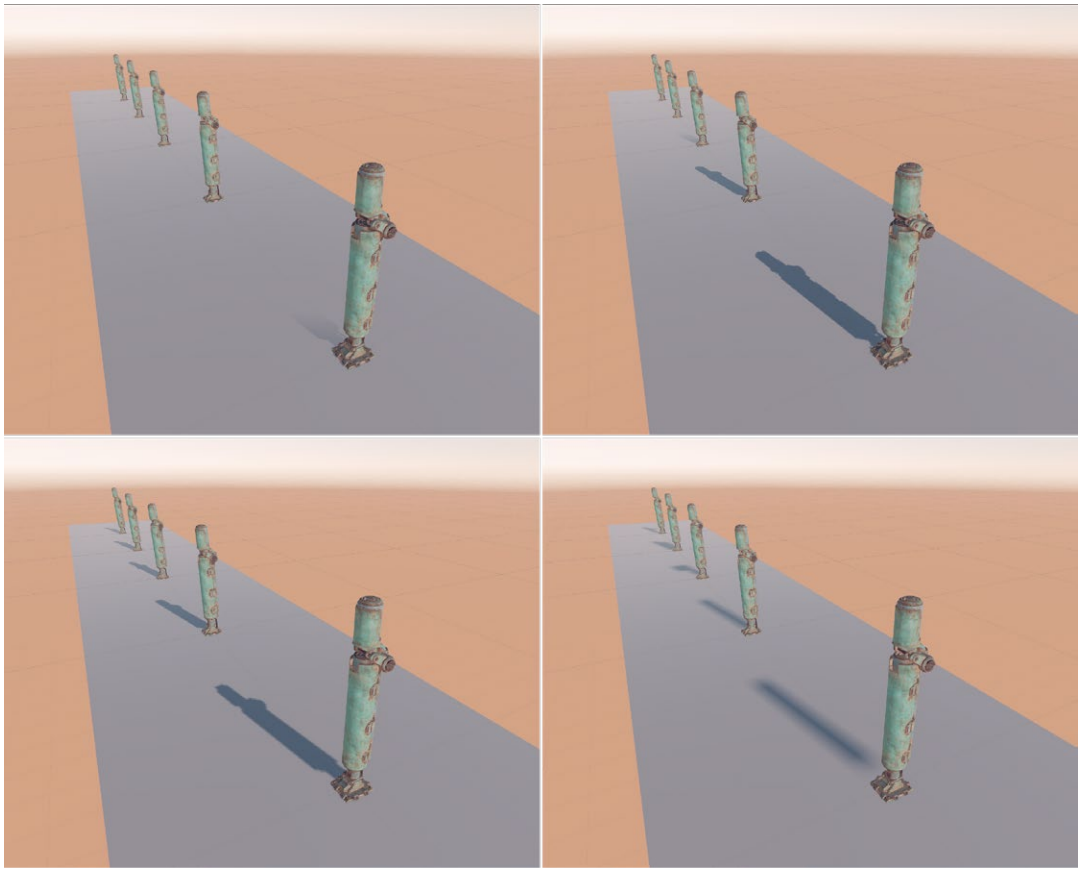
Unity에서 그림자를 사용해 본 적이 있다면, 실시간 그림자에는 광원의 시점에서 오브젝트 뎁스를 포함한 섀도우 맵을 렌더링해야 한다는 사실을 알고 계실 것입니다. 섀도우 맵의 해상도가 높아질수록 더 강력한 프로세싱 성능과 더 많은 메모리 용량이 필요하지만 시각적 정확도는 높아집니다. 그림자 프로세싱을 늘리는 요인은 다음과 같습니다.

1. 섀도우 맵에서 렌더링되는 섀도우 캐스터의 수 – 메인 광원에 대한 이 수치는 그림자 거리(그림자 절두체의 원거리 평면)에 따라 바뀜
2. 시야에 들어오는 섀도우 리시버(모두 포함해야 함)
3. 그림자 캐스케이드 분할
4. 그림자 필터링(소프트 섀도우)

최고 해상도를 사용한다고 해서 언제나 가장 이상적인 결과를 얻을 수 있는 것은 아닙니다. 예를 들어 **Soft Shadows** 옵션에는 맵을 블러 처리하는 효과가 있습니다. 아래의 카툰풍 유령의 방 이미지를 보면 전경의 의자가 책상 서랍에 그림자를 드리우고 있는데, 해상도가 1024보다 높으면 그림자가 너무 선명하게 표시됩니다.

메인 광원의 그림자 해상도 설정: 왼쪽 상단 이미지는 256, 오른쪽 상단 이미지는 512, 왼쪽 가운데 이미지는 1024, 오른쪽 가운데 이미지는 2048, 하단 이미지는 4096으로 해상도 설정

메인 광원: 그림자 최대 거리



메인 광원 그림자의 다양한 최대 거리: 왼쪽 상단 이미지 – 10, 오른쪽 상단 이미지 – 30, 왼쪽 하단 이미지 – 60, 오른쪽 하단 이미지 – 400

메인 광원 그림자의 또 다른 중요한 설정은 최대 거리이며, 씬 단위로 설정합니다. 위 이미지에서 기둥은 서로 10유닛씩 떨어져 있으며, Max Distance는 10에서 400유닛까지 다양합니다. 왼쪽 상단 이미지를 보면 첫 번째 기둥에만 그림자가 살짝 드리워져 있으며, 이 그림자는 카메라 위치부터 10유닛 떨어진 지점에서 갑자기 끊깁니다. Max Distance가 60유닛인 이미지(왼쪽 하단)에서 보면 모든 그림자가 적절한 정확도로 표시된 것을 볼 수 있습니다. Max Distance가 표시되는 에셋보다 훨씬 큰 경우, 섀도우 맵이 영역에서 너무 넓게 퍼집니다. 다시 말해서, 장면 내 영역의 해상도가 원하는 수준보다 훨씬 낮아집니다.

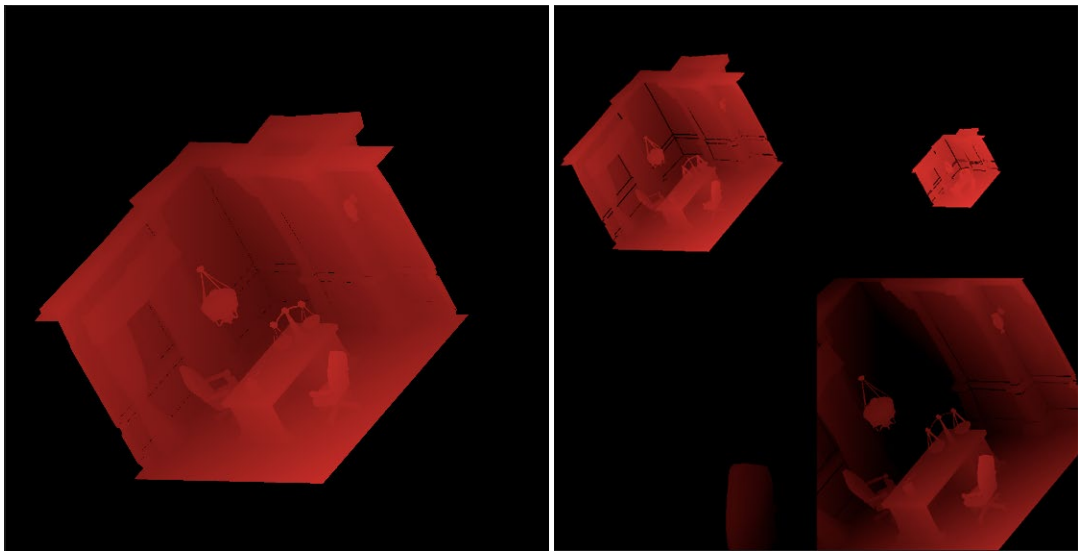
Max Distance 프로퍼티는 씬에 사용되는 단위뿐 아니라 사용자가 볼 수 있는 오브젝트에도 바로 연결되어야 합니다. 적절한 수준의 그림자가 생성되는 범위에서 최소 거리를 설정해야 합니다(아래 내용 참조).
카메라에서 60유닛 떨어진 동적 오브젝트의 그림자만 플레이어에게 표시되는 경우라면 Max Distance를 60으로 설정합니다. 혼합 광원의 조명 모드가 Shadowmask로 설정되면 Shadow Distance를 넘어서는 오브젝트의 그림자가 베이크됩니다. 정적 씬이라면 모든 오브젝트에서 그림자가 표시되지만, 동적 그림자만이 Shadow Distance까지 드로우됩니다.

그림자 캐스케이드

원근 때문에 거리가 멀어지면 에셋이 사라지므로, 그림자 해상도를 줄여 섀도우 맵에서 더 많은 부분이 카메라에 가까운 그림자에 사용되도록 하는 것이 편리합니다. 그림자 캐스케이드를 사용해 보세요.

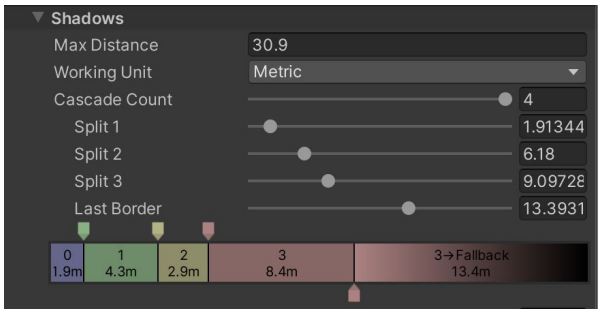
아래 이미지는 유령의 방에 의자와 책상이 있는 씬의 섀도우 맵입니다. 왼쪽 이미지에서 캐스케이드 카운트는 1이며 맵이 전체 영역을 차지하고 있습니다. 오른쪽 이미지에서 캐스케이드 카운트는 4이며, 맵에 네 개의 서로 다른 맵이 포함되어 있고 각 영역에는 낮은 해상도의 맵이 있습니다.

캐스케이드 카운트가 1이면 이처럼 작은 씬에서 최상의 결과를 낼 가능성이 높습니다. 하지만 Max Distance 값이 큰 경우 캐스케이드 카운트를 2 또는 3으로 설정하면 섀도우 맵의 많은 부분이 전달되므로 전경 오브젝트에 더 나은 그림자가 생성됩니다. 왼쪽 이미지의 의자가 훨씬 더 크기 때문에 더 선명한 그림자가 만들어지는 것을 볼 수 있습니다.



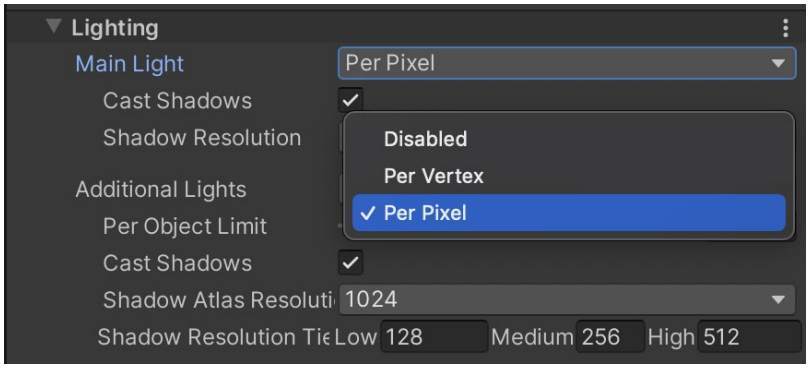
캐스케이드 카운트를 각각 1(왼쪽 이미지), 4(오른쪽 이미지)로 설정했을 때 섀도우 맵

드래그 가능한 포인터를 사용하거나 관련 필드에서 단위를 설정하여 캐스케이드의 각 섹션에 대해 시작 및 종료 범위를 조정할 수 있습니다(다음 이미지 참조). 항상 Max Distance를 씬에 딱 맞는 값으로 조정하고 슬라이더 위치를 신중하게 선택하세요. 측정 기준을 작업 단위로 사용하는 경우 항상 마지막 캐스케이드가 최대한 마지막 섀도우 캐스터의 거리가 되도록 선택합니다.



그림자 캐스케이드 범위 조정

추가 광원 그림자

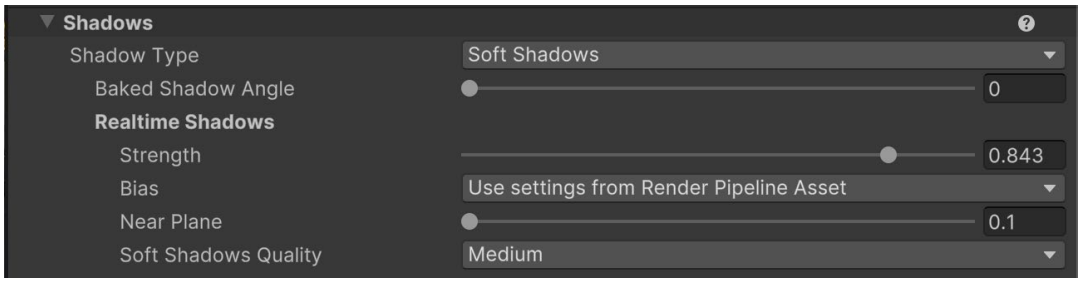


URP 에셋에서 설정할 수 있는 Additional Lights 옵션

Main Light의 그림자를 정렬했으면 이제 **Additional Lights 모드**로 넘어갈 차례입니다. URP 에셋에서 Additional Lights 모드를 **Per Pixel**로 설정해 그림자를 드리울 추가 광원을 활성화합니다. 모드를 Disabled 또는 Per Vertex, Per Pixel로 설정할 수 있지만(위 이미지 참조) Per Pixel만 그림자와 연동됩니다.

참고: URP는 추가 방향 광원에 대한 그림자를 지원하지 않습니다. 메인 광원은 항상 가장 밝은 방향 광원입니다. 추가 광원에 그림자를 사용하려면 점 광원이나 스폿 광원을 사용하세요.

Cast Shadows 체크박스를 선택한 다음 **Shadow Atlas**의 해상도를 선택합니다. 이 맵은 그림자를 드리우는 모든 광원의 맵을 모두 결합하는 데 사용됩니다. 점 광원은 모든 방향으로 빛을 드리우므로 6개의 섀도우 맵에 그림자가 드리워지고 큐브맵이 생성됩니다. 따라서 점 광원은 성능 측면에서 가장 까다롭습니다. 추가 광원 섀도우 맵의 개별 해상도는 세 가지 그림자 해상도 티어의 조합과 계층(Hierarchy) 창에서 광원을 선택할 때 광원 인스펙터에서 선택한 해상도를 기반으로 설정됩니다.



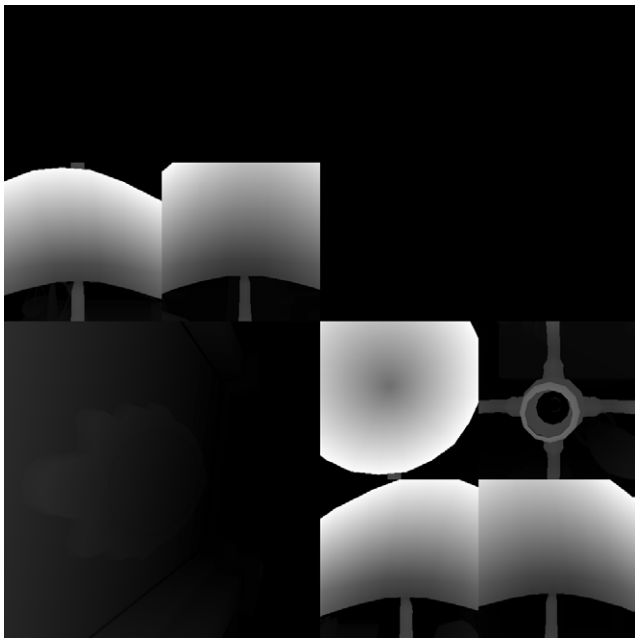
광원 인스펙터의 Shadows 그룹

Shadow Type을 **Soft Shadows**로 설정하면 **Baked Shadow Angle** 슬라이더가 활성화됩니다. 이 프로퍼티는 그림자 가장자리를 인위적으로 부드럽게 다듬어서 더 자연스러운 모습을 연출합니다. Unity 6에서는 **Soft Shadows Quality**를 Low, Medium, High로 설정할 수 있는 옵션이 추가되었습니다.

유령의 방에서 거울 위에는 스폿 광원이 있고 책상 위에는 점 광원이 있습니다. 맵은 7개가 있으며, 7개 맵을 1024px 정사각형 맵에 맞추려면 각 맵의 크기가 256px 이하여야 합니다. 이 크기를 초과하면 섀도우 맵의 해상도가 아틀라스에 맞게 축소되고 콘솔에 관련 경고 메시지가 표시됩니다.

맵 개수	아틀라스 타일링	아틀라스 크기(그림자 티어 크기의 배수)
1	1x1	1
2~4	2x2	2
5~16	4x4	4

Additional Lights 섀도우 맵의 수 및 맵별로 선택한 티어 크기에 따른 섀도우 아틀라스 크기 설정



Additional Lights의 섀도우 아틀라스

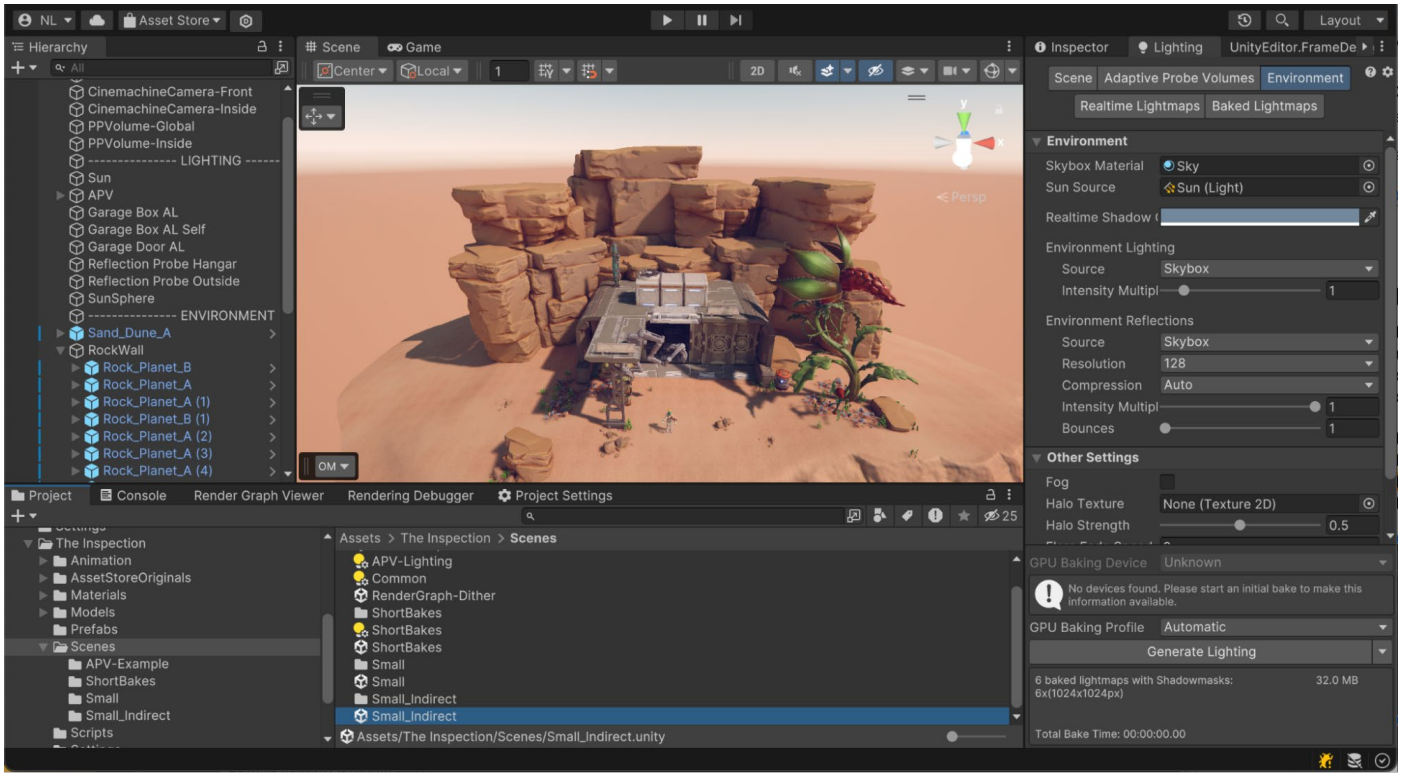
위 이미지는 점 광원에서 사용하는 6개의 맵으로, 해상도는 중간으로 설정되었고 티어 값은 256px입니다. 스폿 광원의 해상도는 높음으로 설정되었고 티어 값은 512px입니다.

주요 방향 광원, 책상 위 점 광원, 거울 위 스폿 광원이 적용된 로우 폴리곤 버전 유령의 방으로, 모든 조명은 실시간이며 그림자가 추가되어 있습니다.

광원 모드

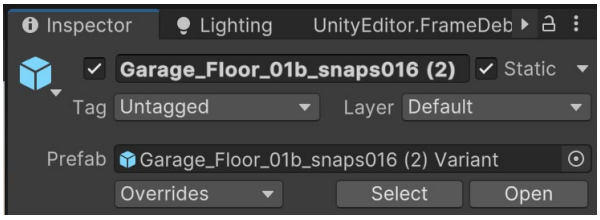
환경은 대부분 정적 지오메트리이므로 광원이 정적인 경우 이에 대한 조명과 그림자를 반복적으로 계산하지 않아도 됩니다. 디자인 시에 한 번 계산한 다음, 지오메트리를 렌더링할 때 해당 데이터를 사용할 수 있습니다. 이를 라이트매핑 또는 베이크라 합니다.

Unity의 FPS 샘플 프로젝트를 사용해 라이트매핑의 각 단계를 살펴보겠습니다. 아래의 스크린샷은 여기에서 다운로드할 수 있는 프로젝트의 스크린샷입니다. **Inspection > Scenes > Small Indirect** 씬은 URP에서 실시간 조명과 베이크된 조명을 어떻게 사용하는지 보여 줍니다.

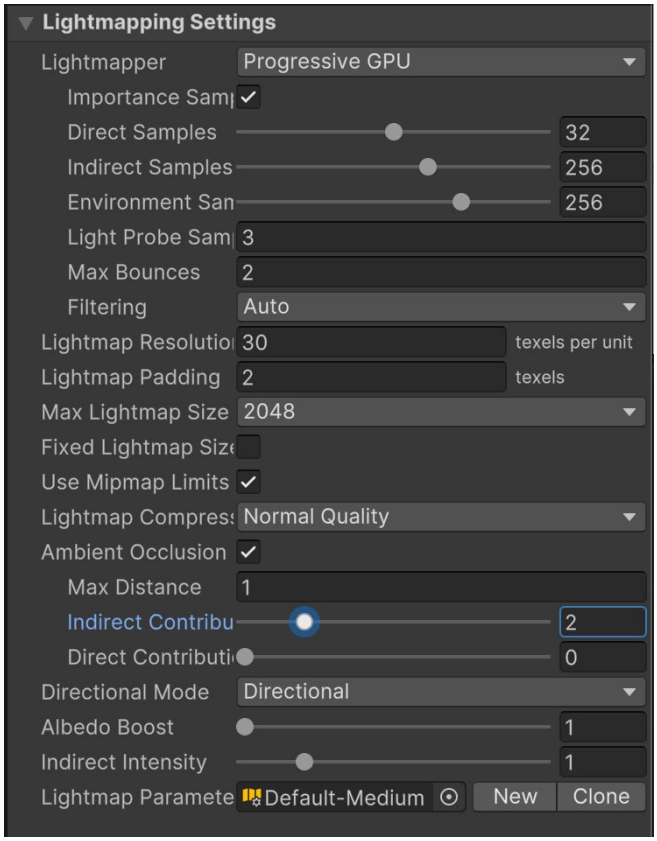


GitHub의 Unity 6 URP 전자책 저장소의 Inspection > Scenes > Small Indirect 씬

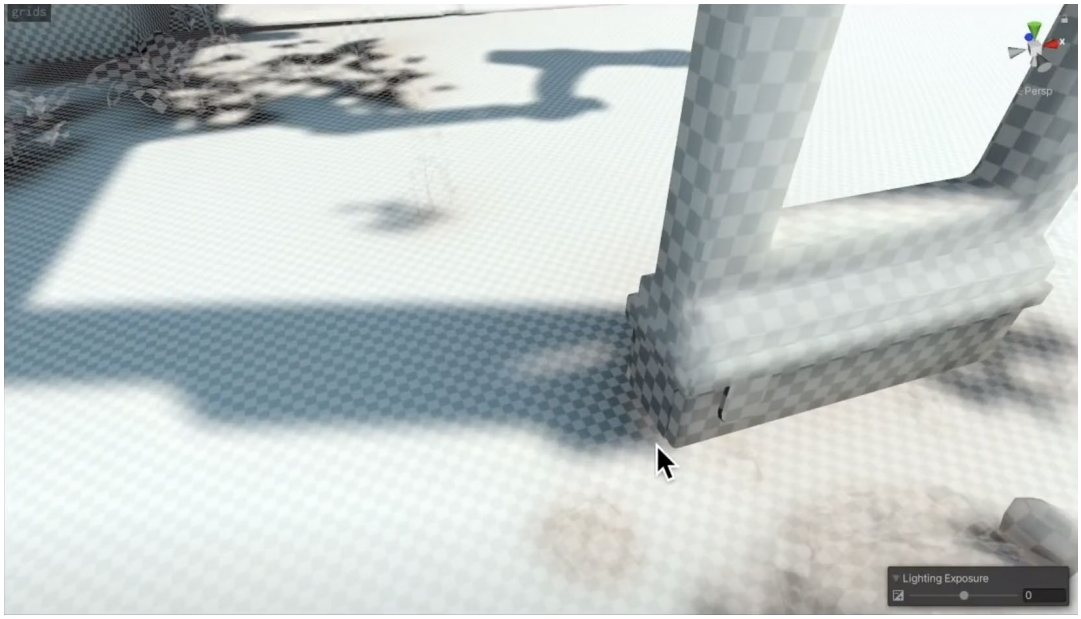
1. 이 FPS 샘플 프로젝트의 씬에서는 대체로 정적 지오메트리를 사용합니다. 라이트매핑에 지오메트리를 포함하려면 인스펙터 오른쪽의 **Static** 체크박스를 클릭합니다.



2. **Window > Rendering > Lighting > Scene**을 통해 라이트매핑 설정을 선택합니다. 설정을 조정하는 동안 Lightmap Resolution을 낮게 유지합니다. 원하는 대로 설정하고 나서 최종 라이트맵을 생성할 때 값을 늘리면 됩니다. GPU에서 지원하는 경우 **Progressive GPU**를 선택해 라이트맵을 더 빠르게 생성할 수 있습니다.

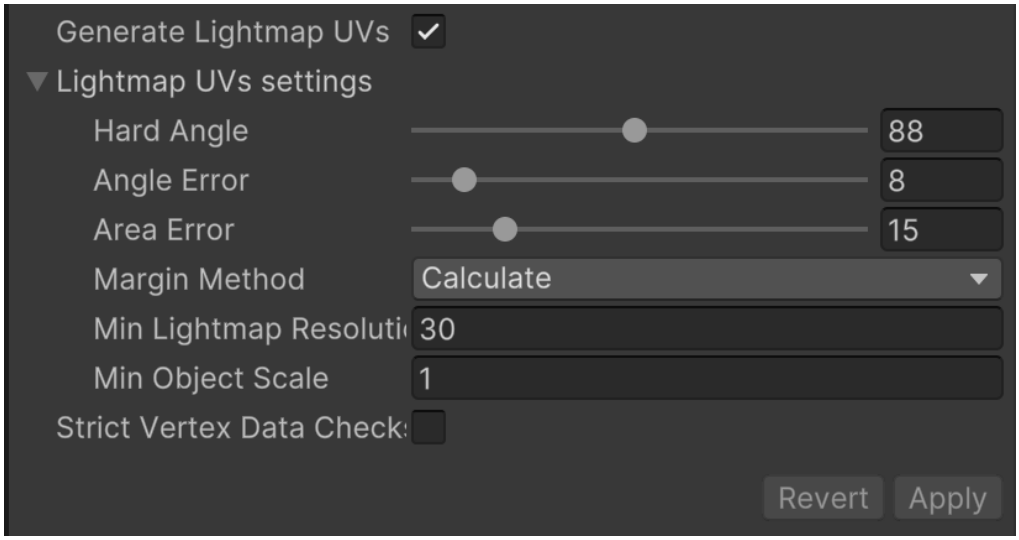


3. 필터링으로 맵을 블러 처리해 노이즈를 최소화할 수 있습니다. 오브젝트가 서로 만나는 곳에서 그림자에 간격이 생길 수 있는데, **A-Trous** 필터링을 사용하면 이러한 문제를 최소화할 수 있습니다. 라이트매핑에 사용할 수 있는 설정에 대한 자세한 내용은 프로그레시브 라이트매핑 기술 자료를 참조하세요.

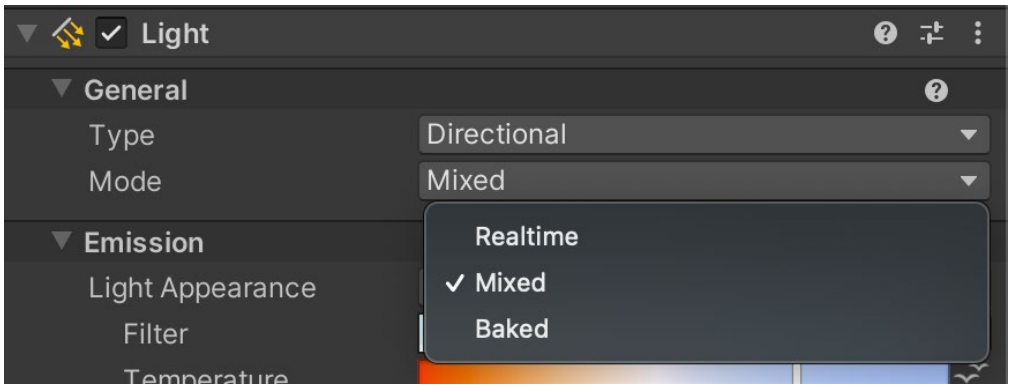


필터링이 오브젝트 사이의 그림자에 미치는 영향

4. 모든 정적 지오메트리가 중복되는 UV 값을 갖지 않거나 임포트 시 조명 **UV**를 생성하도록 합니다.



5. **Light Mode**를 **Baked** 또는 **Mixed**로 설정합니다. **계층** 창에서 광원을 선택하고 **인스펙터**를 사용합니다. 혼합 광원은 동적 오브젝트와 정적 오브젝트에 조명을 적용합니다.

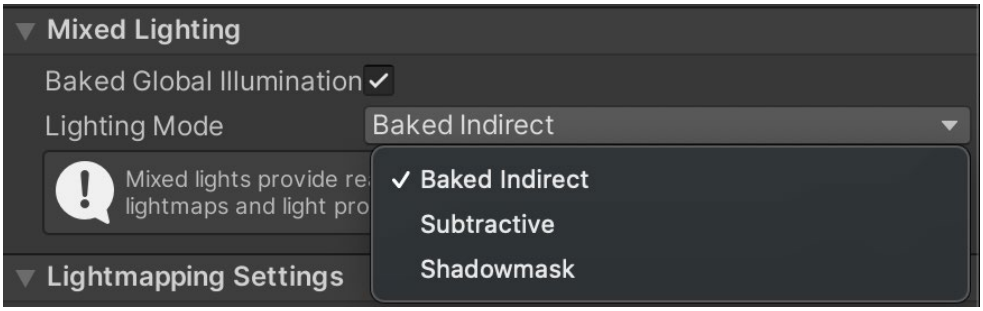


6. 혼합 광원을 사용할 때 **Window > Rendering > Lighting > Scene**에서 **Light Mode**를 **Baked Indirect**, **Subtractive** 또는 **Shadowmask**로 설정합니다.

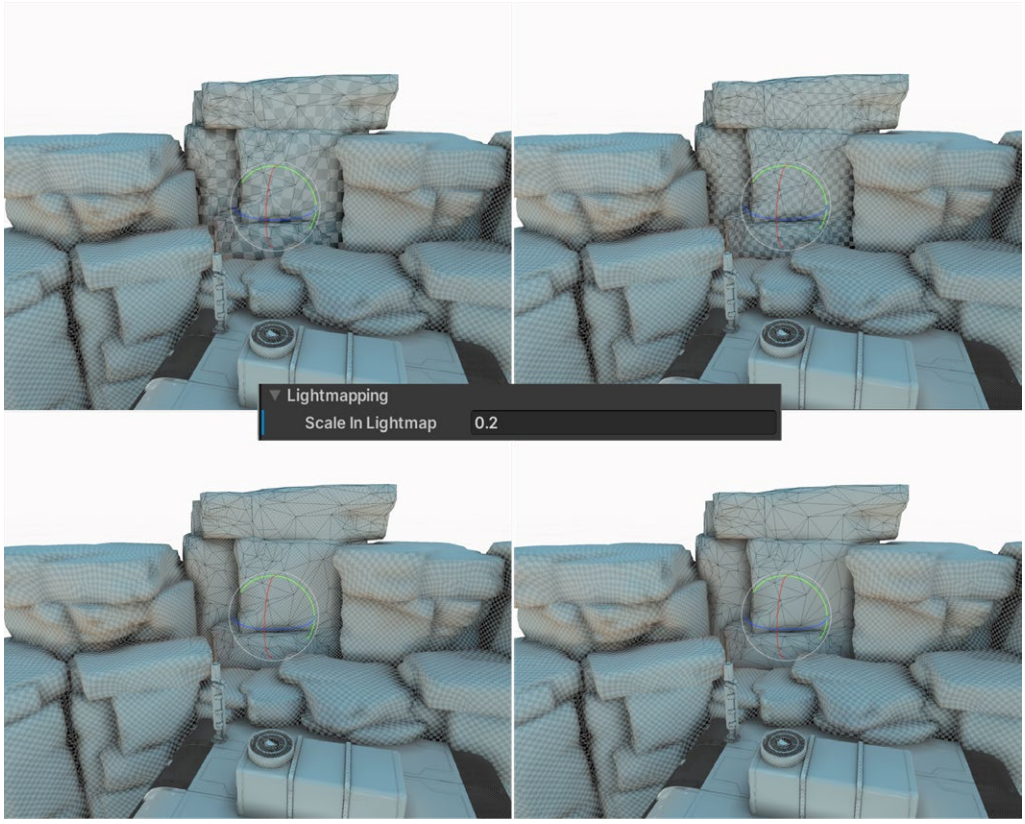
   a. **Baked Indirect:** 간접광의 영향만 라이트맵과 라이트 프로브로 베이크됩니다(광원 반사만). 직접 조명과 그림자는 실시간으로 처리됩니다. 이 옵션은 리소스를 많이 소모하므로 모바일 플랫폼에는 적합하지 않습니다. 하지만 정적 지오메트리와 동적 지오메트리 양측에 대해 정확한 그림자와 직접광을 구현할 수 있습니다.

b. **Subtractive:** 여기서는 Mixed로 설정된 방향 광원에서 정적 지오메트리로 직접 조명을 베이크하며, 동적 지오메트리에 의해 드리워진 그림자에서 조명을 감산합니다. 그러면 라이트 프로브 또는 APV(적응적 프로브 볼륨)를 사용하지 않는 한, 정적 지오메트리가 동적 오브젝트에 그림자를 드리울 수 없으므로 눈에 거슬리는 시각적 끊김 현상이 발생할 수 있습니다. URP는 방향 광원의 광원 영향 추정값을 계산하고 베이크된 전역 조명에서 이를 감산합니다. 이 추정값은 Lighting 창의 Environment 섹션에 있는 Real-time Shadow Color 설정으로 고정되므로 감산된 색상은 이 색상보다 어두울 수 없습니다. 그런 다음 감산된 값의 최소 색상과 원래의 베이크된 색상을 선택합니다. 이 모드는 저사양 하드웨어에 가장 적합한 옵션이지만, 베이크된 그림자와 실시간 그림자를 런타임에 올바르게 결합할 수 없으므로 결함이 발생하게 됩니다. 품질보다 성능을 우선시하는 절충안입니다.

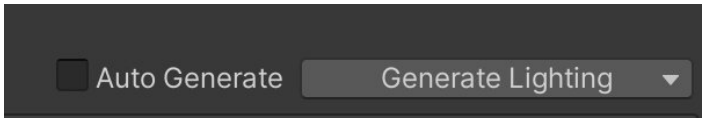
c. **Shadowmask:** Baked Indirect 모드와 비슷하지만, Shadowmask는 동적 그림자와 베이크된 그림자를 모두 조합하고 원거리에서 그림자를 렌더링합니다. 추가 Shadowmask 텍스처를 사용하고 라이트 프로브에 추가 정보를 저장해 이 작업을 수행합니다. 이 옵션을 사용하면 가장 정확도 높은 그림자를 구현할 수 있지만, 메모리 사용이나 성능 측면에서 가장 많은 비용이 드는 옵션이기도 합니다. 근거리에서는 시각적으로 Baked Indirect와 전혀 차이가 없지만, 원거리에서 보면 차이가 뚜렷하므로 오픈 월드 씬에 가장 적합한 옵션입니다. 프로세싱 비용을 감안해 중고급 사양 하드웨어에서만 사용하는 것이 바람직합니다.



7. 라이트맵에서 원거리의 오브젝트가 공간을 덜 차지하도록 **Asset > Inspector > Mesh Renderer > Lightmapping > Scale In Lightmap**에서 **Lightmap Scale**을 조정합니다. 다음 이미지에는 0.05~0.5로 다양하게 설정된 배경 암석 라이트맵의 텍셀 크기가 나와 있습니다.



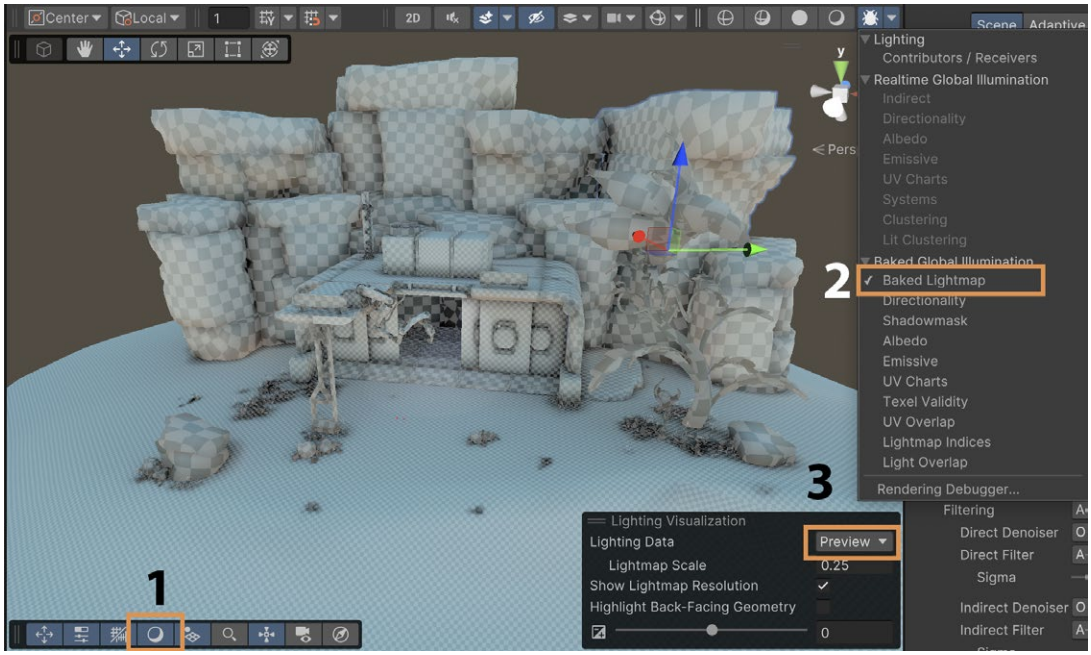
8. **Generate Lighting**을 클릭해 베이크합니다. 베이크 시간은 정적 오브젝트의 개수, Mixed 또는 Baked 모드로 설정된 광원, 라이트매핑에 대해 선택된 설정, 특히 최대 라이트맵 크기 및 라이트맵 해상도에 따라 달라집니다. 베이크 시간은 베이크에 사용되는 광선 수에 비례하므로 Sample Count(Direct Samples, Indirect Samples and Environment Samples)도 베이크 시간에 직접적인 영향을 미칩니다.



Window > Rendering > Lighting > Generate Lighting으로 라이트맵 베이크

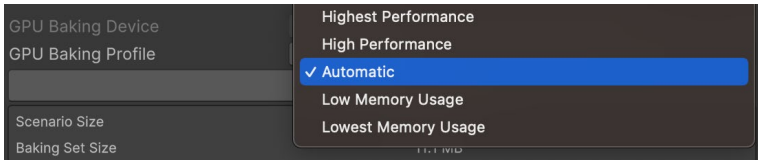
9. Unity 6에서는 드로우 모드를 활성화(1)*하고 Baked Lightmap 옵션을 선택(2)하면 새로운 인터랙티브 베이크 기능을 사용할 수 있습니다. Preview 옵션(3)이 있는 새로운 패널이 표시됩니다. 이 모드에 있는 동안에는 변경한 내용이 생성된 데이터에 영향을 주지 않습니다. 테크니컬 아티스트는 이 기능을 사용하여 프로퍼티를 조정하고, 앞서 오랫동안 계산한 이전 베이크를 파괴하지 않고도 수정 사항이 렌더링에 미치는 영향을 확인할 수 있습니다.

   *숫자는 아래 이미지를 참조하세요.



인터랙티브 미리 보기 모드 활성화

Unity는 이제 **베이킹 프로파일**도 지원합니다. 이 기능은 온디맨드 모드에서 GPU 백엔드를 사용할 때 **Lighting** 창에서 사용할 수 있으며, 사용자가 성능과 GPU 메모리 사용량 사이에서 원하는 수준을 절충할 수 있습니다.



GPU 베이킹 프로파일

참고:

Unity는 2019 릴리스부터 명시적으로 베이크되지 않은 씬에 베이크된 환경 조명을 자동으로 생성하는 시스템을 제공해 왔습니다. 이 시스템의 명칭은 SkyManager입니다. 그러나 SkyManager는 자동 동작이 명확하지 않고 특정 상황에서만 작동한 탓에 사용자에게 혼란을 주었습니다. 또한 에디터의 동작과 빌드된 플레이어의 동작에 차이를 유발하는 바람에 환경 조명이 예기치 못하게 누락되는 경우가 발생했습니다.

Unity 6에서는 SkyManager가 에디터의 새로운 기본 조명 데이터 에셋으로 대체되며, 씬을 새로 생성할 때마다 이 에셋이 할당됩니다. 이 에셋에는 환경 조명의 기본 설정에 부합하는 환경 조명이 포함됩니다. Skybox 모드에서 이 설정을 변경하면 Lighting 창의 **Generate Lighting** 버튼을 사용하여 조명을 직접 다시 베이크해야 합니다.

더 많은 리소스:

- 라이트매핑 기술 자료
- 조명 설정 에셋 기술 자료
- 조명 탐색기 기술 자료
- 라이트매핑에서 자주 발생하는 다섯 가지 문제와 해결을 위한 팁

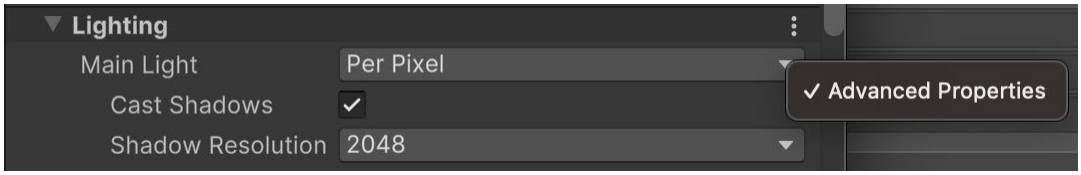
렌더링 레이어

렌더링 레이어 기능을 사용하면 특정 광원을 특정 게임 오브젝트에만 영향을 주도록 설정해 해당 오브젝트가 씬에서 부각되고 시선을 끌도록 할 수 있습니다. 아래 이미지에서는 주요 아이템인 주사기가 씬의 그늘진 부분에 있습니다. 렌더링 레이어를 사용하면 플레이어가 아이템을 놓치지 않고 주울 수 있도록 잘 보이게 할 수 있습니다.

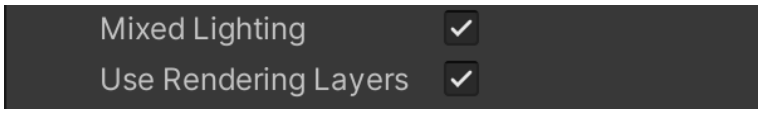
렌더링 레이어를 사용하여 오브젝트 강조

렌더링 레이어를 설정하는 단계는 다음과 같습니다.

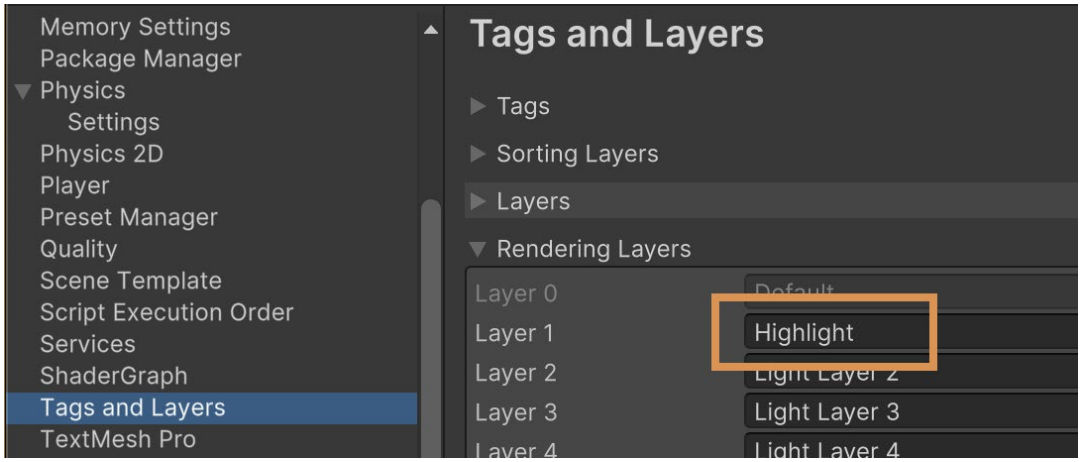
1. **URP Asset**을 선택합니다. Lighting 섹션에서 세로 줄임표 아이콘( )을 클릭하고 **Advanced Properties**를 선택합니다.



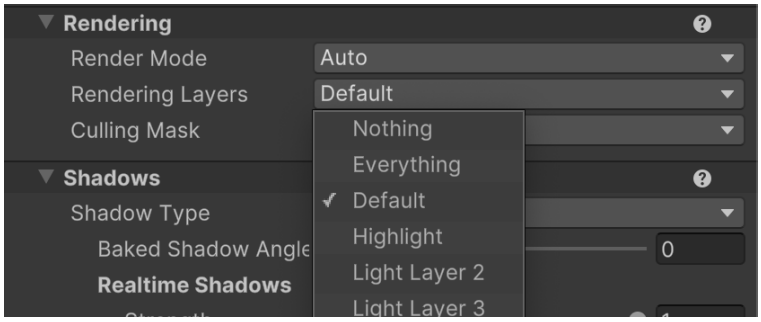
2. 새로운 설정인 **Use Rendering Layers**가 Lighting 섹션에 표시됩니다.



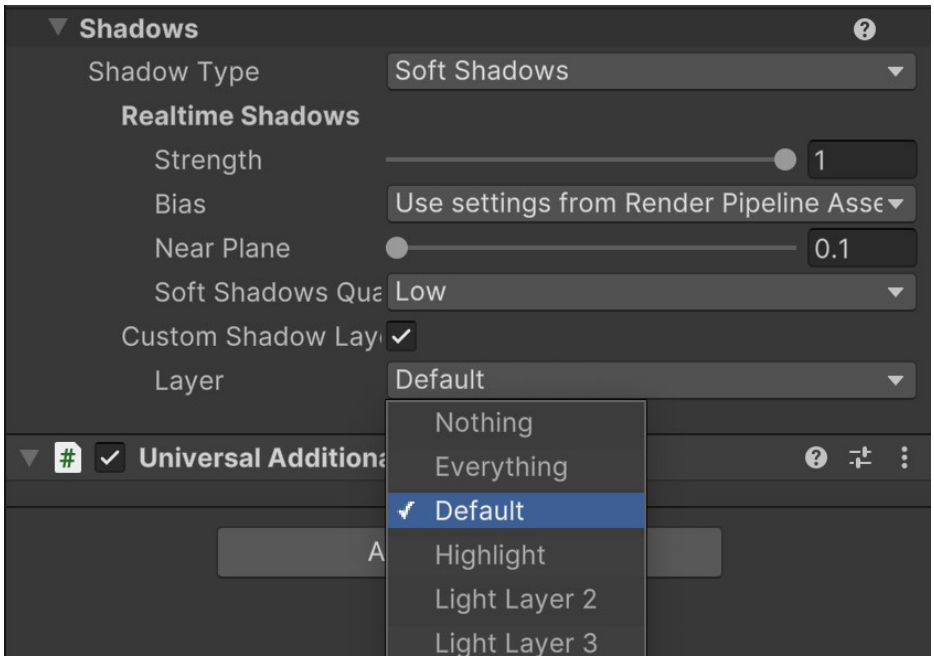
3. **Project Settings > Tags and Layers > Rendering Layers**에서 렌더링 레이어의 이름을 변경합니다.



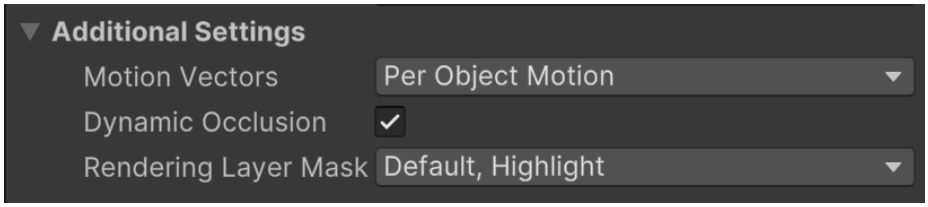
4. **Light Inspector > Rendering** 섹션에 **Rendering Layers** 드롭다운이 있습니다. 하나의 광원이 여러 레이어에 영향을 미칠 수 있습니다.



5. 렌더링 레이어가 활성화되었으니 이제 새로운 광원을 생성하고 커스텀 그림자 레이어를 설정합니다. 새 광원은 씬의 **메인 광원** 또는 해당 광원의 자체 절두체로부터 그림자를 드리울 수 있습니다.



6. 마지막으로 **계층** 창에서 이를 적용할 오브젝트를 선택한 다음 **Rendering Layer Mask**를 설정합니다.



이 설정은 코드로도 구현할 수 있습니다.

```
Renderer renderer = GetComponent<Renderer>();
int layerID = 1;
int mask = 1 << layerID;
renderer.renderingLayerMask = (uint)mask;
```

라이트 프로브

광원 모드 섹션에서 다루었듯이 혼합 조명 모드를 사용하여 베이크된 오브젝트와 동적 오브젝트를 조합할 수 있습니다. 혼합 조명 모드를 사용한다면 씬에 프로브도 추가하는 것이 좋습니다. Unity 6에서는 라이트 프로브와 새로운 APV(적응적 프로브 볼륨)라는 두 가지 옵션을 사용할 수 있습니다. 두 옵션은 동적 오브젝트가 씬에서 움직이더라도 전역 조명에 영향을 받을 수 있도록 하여 동일한 문제를 해결합니다.

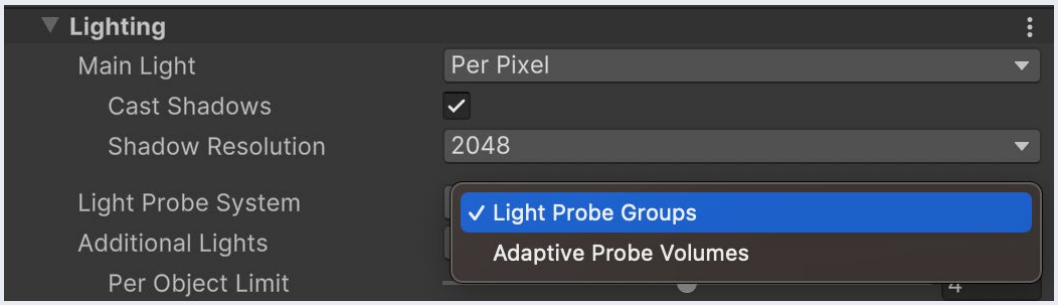
프로브는 씬에 배치하는 한 개의 점입니다. 디자인하는 과정에서 이 위치의 전역 조명이 계산됩니다. 런타임에 프레임을 렌더링할 때, 조명 계산이 포함된 URP 셰이더가 가장 가까운 프로브를 조합하여 전역 조명 값을 계산합니다.

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

라이트 프로브

라이트 프로브는 **Window > Rendering > Lighting** 패널을 통해 **Generate Lighting**을 클릭하여 조명을 베이크할 때 환경 내 특정 위치에 광원 데이터를 저장합니다. 그러면 환경에서 움직이는 동적 오브젝트가 베이크된 오브젝트에서 사용하는 조명 수준에 따라 빛을 낼 수 있습니다. 어두운 영역에서는 어두워지고, 밝은 영역에서는 밝아집니다. 오브젝트별로 샘플링이 수행되므로 대형 오브젝트가 어두운 영역에서 밝은 영역까지 걸쳐져 있는 상황에서는 조명 결함이 발생할 수 있습니다. 씬에서 이러한 문제가 발생한다면 픽셀별로 샘플링하는 APV를 사용하는 방식을 고려해 보세요(가이드 후반부의 APV 섹션 참고).

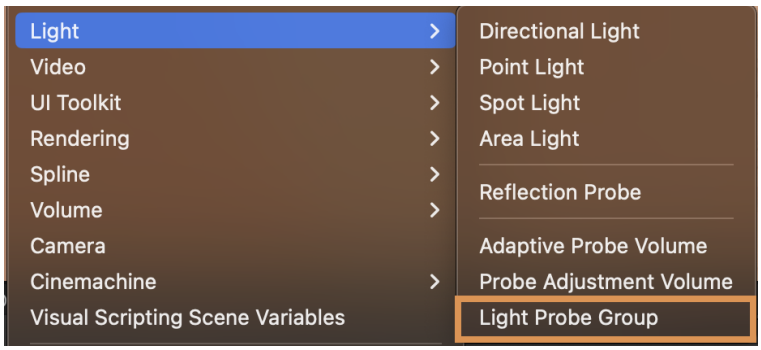
참고: 라이트 프로브를 사용할 때는 활성 URP 에셋의 **Lighting > Light Probe System**을 **Light Probe Group**으로 설정해야 합니다.



아래를 보면 FPS 샘플: 더 인스펙션의 격납고 내부와 외부에 로봇 캐릭터가 있습니다.

격납고 내부 및 외부에 로봇이 있고, 조명 수준이 라이트 프로브의 영향을 받는 모습

라이트 프로브를 만들려면 **계층** 창을 오른쪽 클릭하고 **GameObject > Light > Light Probe Group**을 선택합니다.



새 Light Probe Group 게임 오브젝트 생성

처음에는 8개의 라이트 프로브가 큐브 형태로 배치됩니다. 프로브의 위치를 확인, 수정하고 프로브를 더 추가하려면, **계층** 창에서 **Light Probe Group**을 선택하고 씬 뷰에서 **Tools > Edit Light Probes**를 클릭합니다. 라이트 프로브 기즈모도 활성화해야 합니다.

인스펙터에서 라이트 프로브를 추가/제거하고 위치를 수정

그러면 씬 뷰가 편집 모드로 전환되고 라이트 프로브만 선택할 수 있습니다. 이동 툴 을 사용해 라이트 프로브를 옮기면 됩니다.

라이트 프로브 이동

라이트 프로브는 동적 오브젝트가 이동할 수 있고 조명 수준이 크게 바뀌는 영역에 위치해 있어야 합니다. 오브젝트의 조명 수준을 계산할 때 엔진은 가장 가까운 라이트 프로브의 피라미드를 찾아 조명 수준의 보간된 값을 결정할 때 사용합니다.

선택한 상자에서 가장 가까운 위치의 라이트 프로브

프로브의 위치를 지정하려면 시간이 오래 걸릴 수 있지만, 이 스크립트 같은 코드 기반 접근 방식을 사용하면 특히 대규모 씬이나 APV로 배치를 빠르게 전환할 때 편집 속도를 높일 수 있습니다.

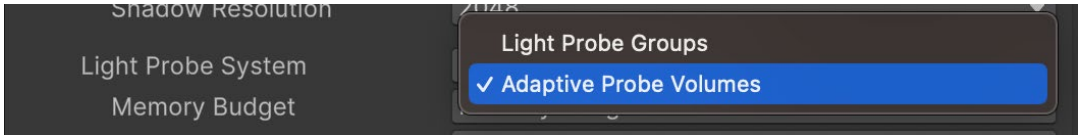
크리에이터는 프로젝트의 씬에 모듈식 콘텐츠를 제작하는 경우가 많습니다. 이러한 씬은 이후 런타임에 '허브' 씬에서 위치가 재지정됩니다. 하지만 라이트 프로브가 포함된 모듈식 콘텐츠를 제작할 때, 프로브의 위치는 읽기 전용이므로 크리에이터가 프로브 위치를 씬과 함께 재지정하지 못했습니다. Unity 6에서는 런타임에 라이트 프로브의 위치를 재지정할 수 있는 신규 API를 통해 이 문제가 해결되었습니다.

메시 렌더러가 라이트 프로브와 함께 작동하는 방식에 대한 자세한 내용과 설정을 조정하는 방법은 이 기술 자료에서 확인할 수 있습니다.

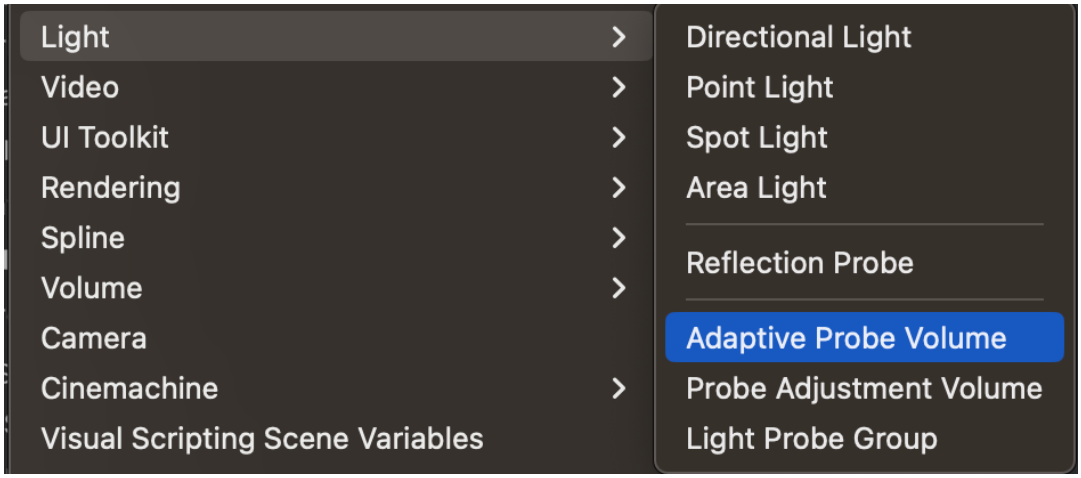
적응적 프로브 볼륨

씬에 라이트 프로브의 위치를 신중하게 지정했더니 씬 레이아웃이 변경되어 있는 상황을 경험해 본 테크니컬 아티스트라면 APV(적응적 프로브 볼륨)의 장점을 즉시 체감할 것입니다. 이제 많은 씬에서 모든 프로브를 몇 초 이내에 배치할 수 있습니다. FPS 샘플: 더 인스펙션을 다시 예시로 보겠습니다. 이 예시는 **The Inspection > Scenes > APV-Example에서 찾을 수 있습니다.**

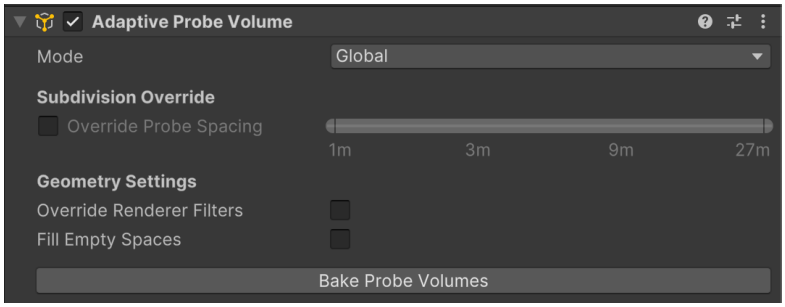
1. 먼저 활성 URP 에셋의 **Light Probe System** 옵션이 **Adaptive Probe Volumes**로 설정되어 있는지 확인합니다.



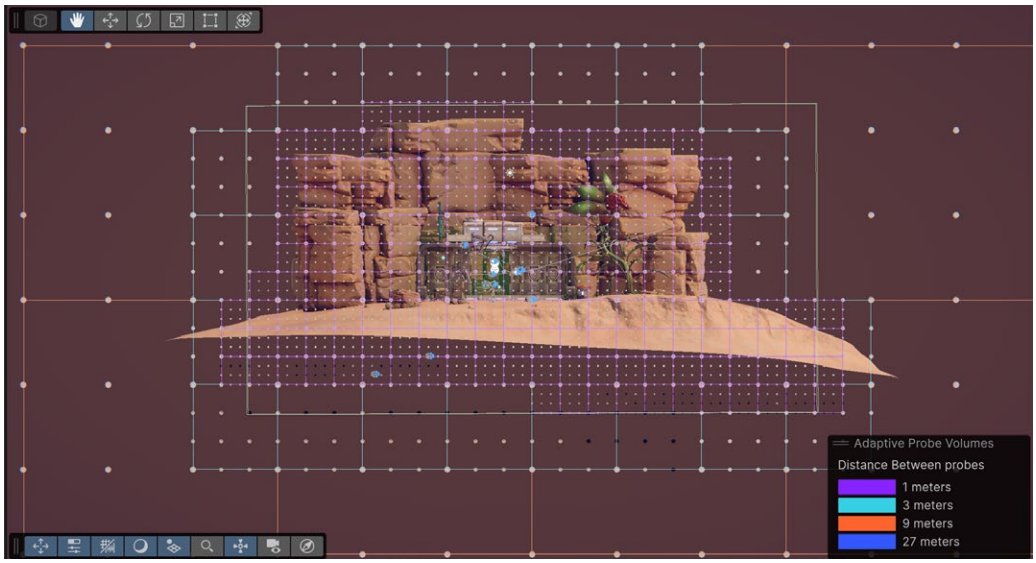
2. 계층 창을 오른쪽 클릭하고 **GameObject > Light > Adaptive Probe Volume**(APV)을 선택합니다.



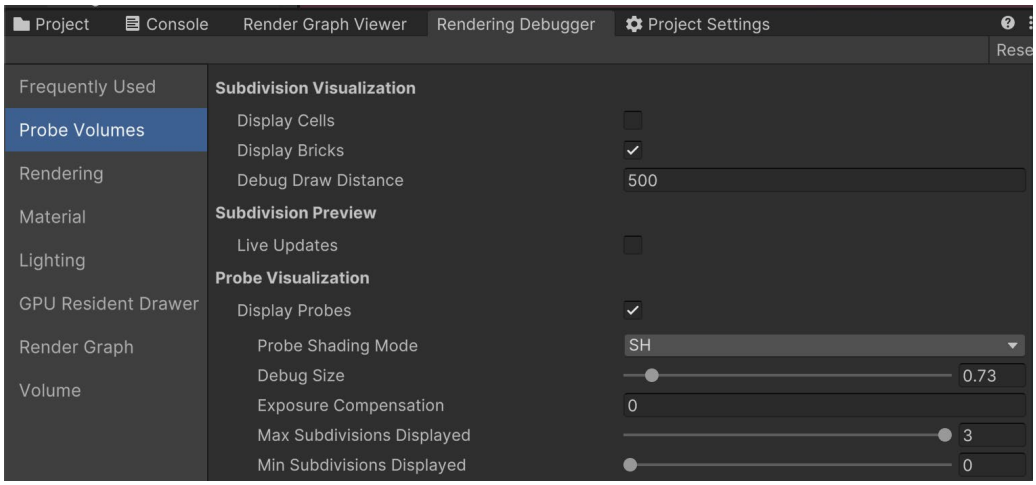
3. Mode를 **Global**로 설정하고 1m, 3m, 9m, 27m로 이뤄진 Subdivision의 기본 설정을 확인합니다.



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



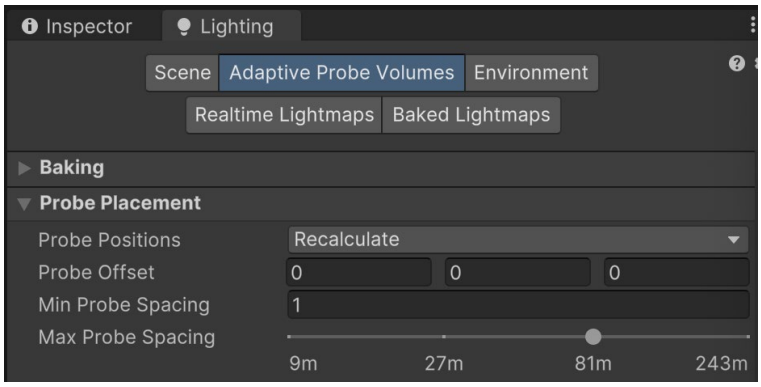
5. 베이크 결과를 확인하려면 **Analysis > Rendering Debugger**를 엽니다. **Probe Volumes**를 선택하고 **Display Probes**를 선택합니다. 다른 해상도를 보려면 **Display Bricks**를 선택합니다.



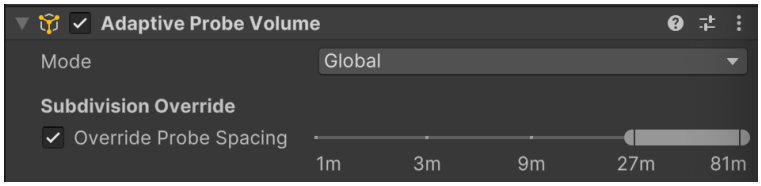
대부분의 씬에서는 이렇게 하면 작업이 완료되므로 마음 놓고 휴식을 취할 수 있습니다. 하지만 APV로 더욱 정확도를 높일 수도 있습니다. 다양한 세분화 수준을 지닌 여러 볼륨을 추가하여 프로브의 배치와 밀도를 세밀하게 제어할 수 있습니다.

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 볼륨**을 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.

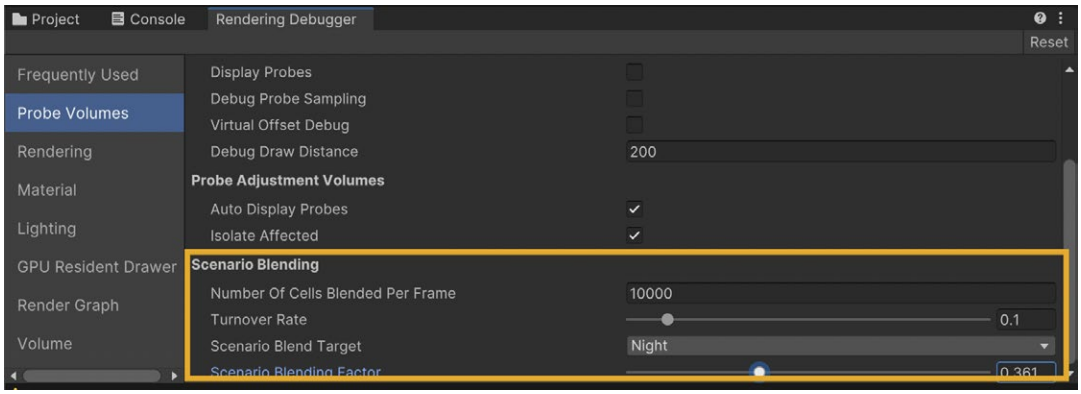
4. 프로브 볼륨을 베이크합니다.

아래 이미지에서 보이는 대로 대부분의 프로브가 텐트 주변에 배치됩니다.

프로브 배치

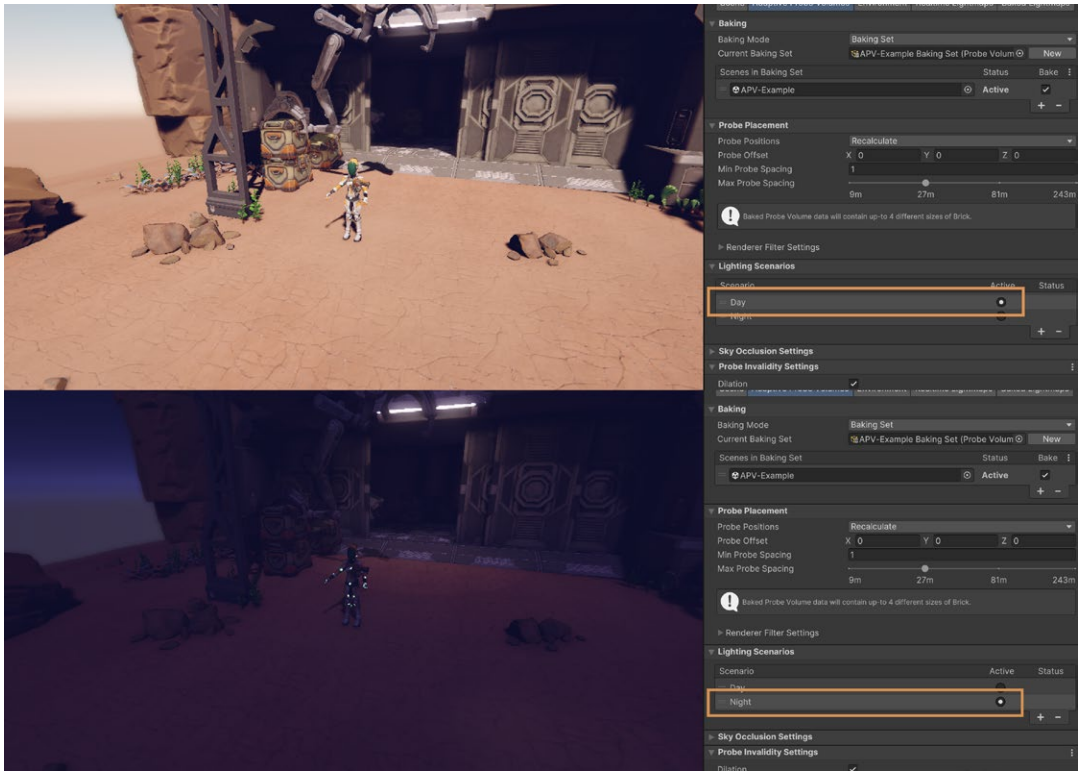
조명 시나리오 에셋

적응적 프로브 볼륨의 또 다른 기능은 간접 조명 데이터를 전환할 수 있는 기능입니다. **조명 시나리오** 에셋에는 씬의 베이크된 조명 데이터나 **베이킹 세트**가 저장됩니다. 다양한 조명 설정을 서로 다른 조명 시나리오로 베이크한 다음, 렌더링 디버거를 사용하여 URP가 런타임이나 디자인 시 사용할 요소를 변경할 수 있습니다.



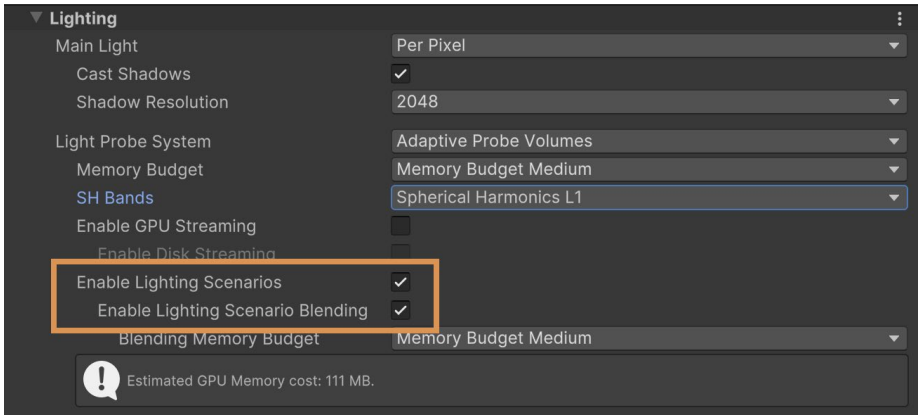
렌더링 디버거를 사용한 시나리오 블렌딩

예를 들어 낮 조명 시나리오와 밤 조명 시나리오를 생성할 수 있습니다. 런타임에는 두 시나리오를 전환하거나 블렌딩할 수 있습니다.



낮/밤 조명 시나리오

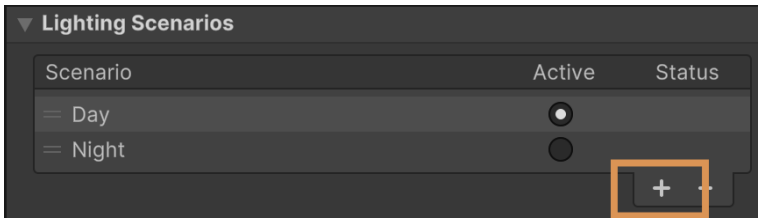
1. 조명 시나리오 에셋을 사용하려면 활성 URP 에셋에서 **Lighting > Light Probe Lighting > Lighting Scenarios**를 활성화합니다.



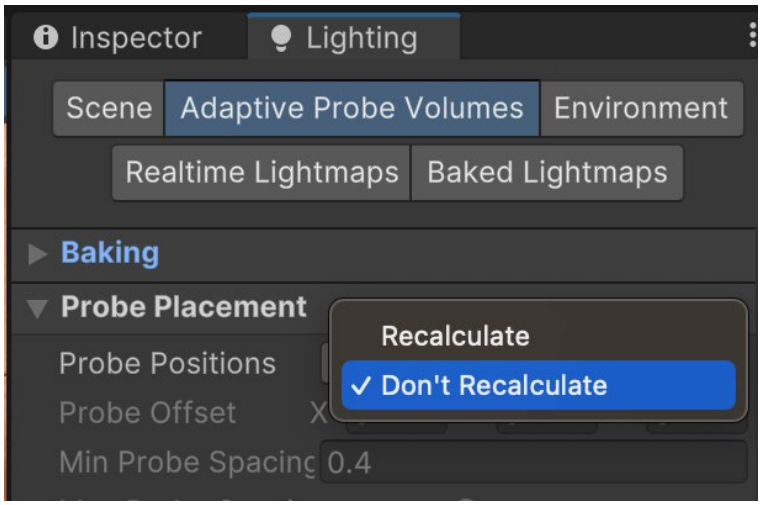
2. 베이크 결과를 저장하기 위한 새로운 조명 시나리오 에셋을 생성하려면 다음을 수행합니다.

   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 베이크합니다. 그러지 않을 경우에는 이전 베이크 시나리오가 무효화될 수 있습니다.

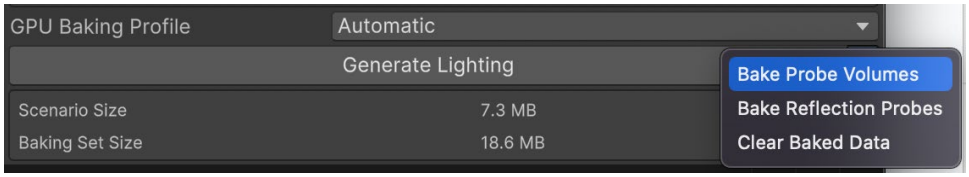


4. 조명 시나리오에 베이크하는 방법은 다음과 같습니다.

   a. Lighting Scenarios 섹션에서 조명 시나리오를 선택하여 활성화합니다.

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

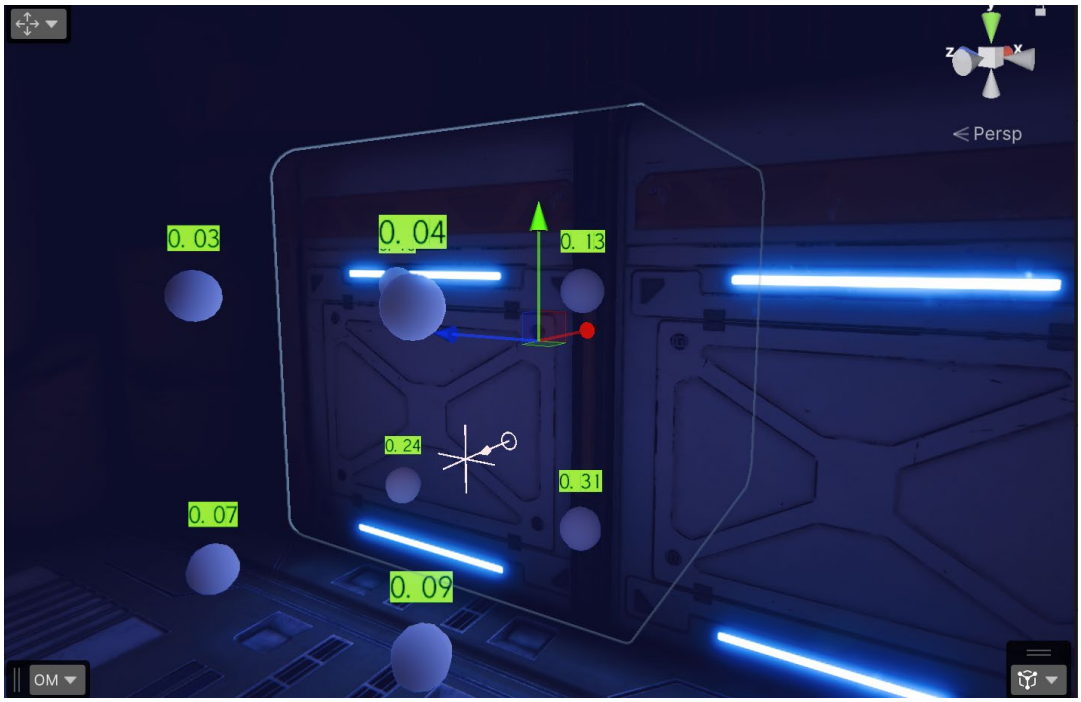
   c. 라이트맵은 사용하지 않고 프로브만 처리하려면 **Generate Lighting** 옆의 드롭다운 버튼을 사용합니다.



런타임에 URP가 사용할 조명 시나리오를 설정하려면 ProbeReferenceVolume API를 사용합니다.

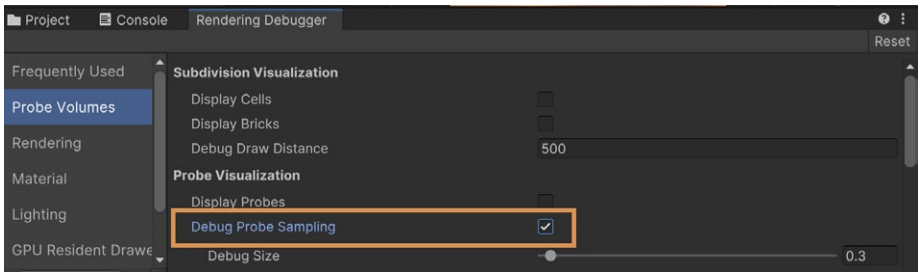
참고:
런타임에 활성 조명 시나리오를 변경하면 URP는 라이트 프로브의 간접 조명 데이터만 변경합니다. 지오메트리를 이동하거나 광원 또는 직접 조명을 수정하려면 스크립트를 사용해야 할 수 있습니다.

적응적 프로브 볼륨 문제 해결



Debug Probe Sampling

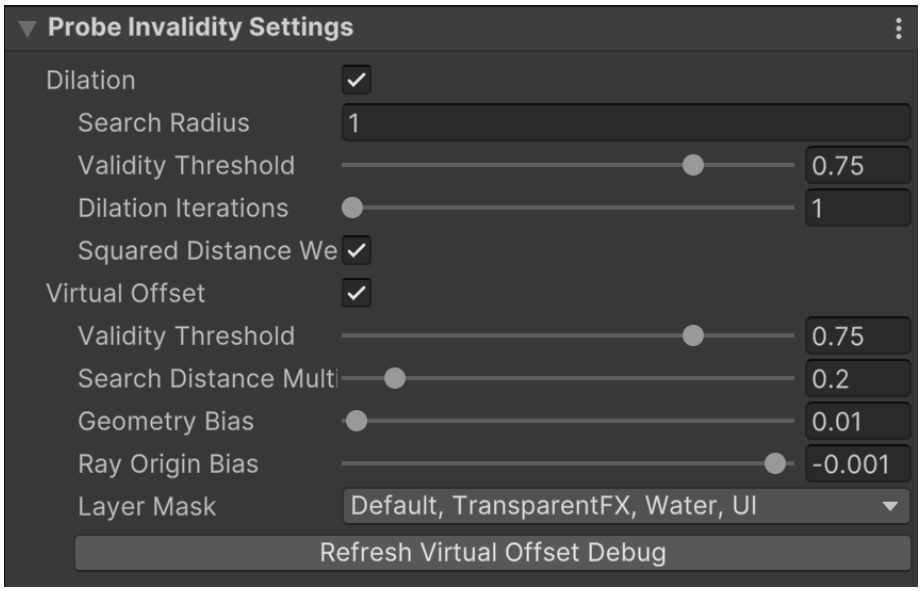
APV 결함 같은 문제를 해결하려면 **Window > Analysis > Rendering Debugger > Probe Volumes > Debug Probe Sampling**을 사용하여 프로브를 검사하고, 특정 픽셀에서 어떻게 샘플링되는지 살펴봅니다.



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 Invalidity 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.

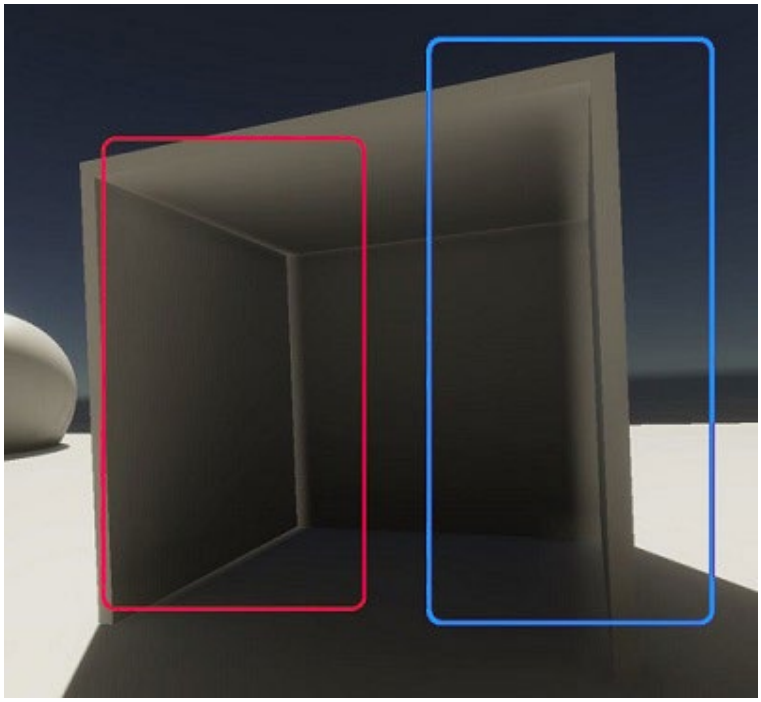
유효하지 않은 라이트 프로브를 확인하려면 **렌더링 디버거를 사용합니다.**

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

위 이미지의 왼쪽 씬에서는 Dilation이 활성화되지 않아 일부 영역이 너무 어둡습니다. 오른쪽 씬에서는 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 오버라이드를 추가합니다.

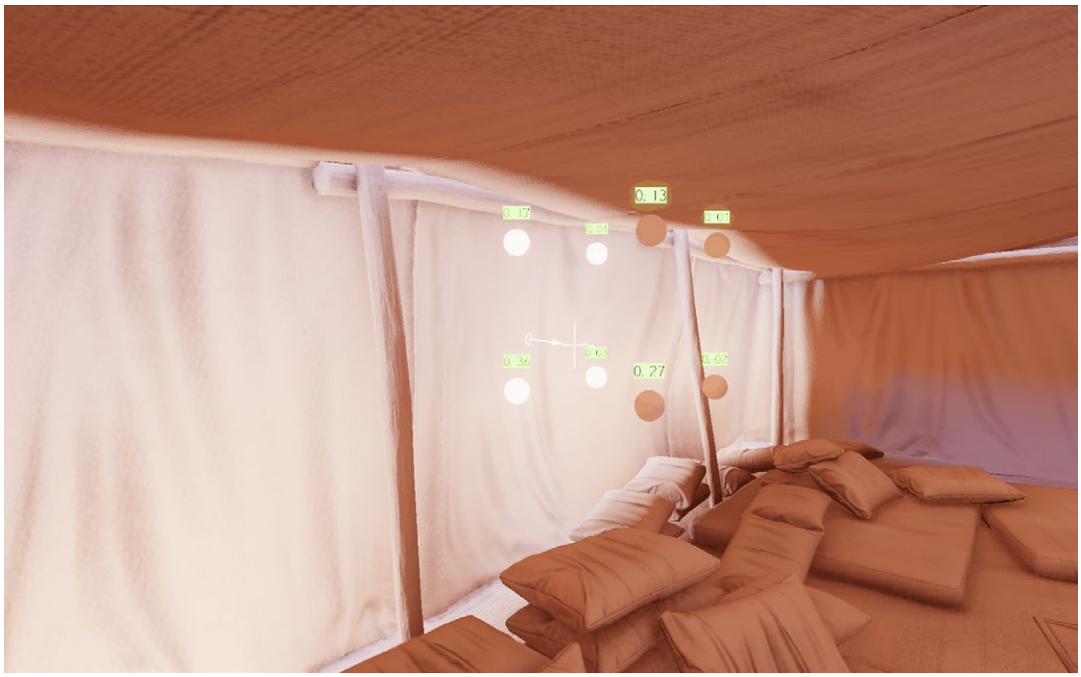
- 볼륨을 추가한 다음, 해당 볼륨에 **Adaptive Probe Volumes Options** 오버라이드를 추가합니다. 이렇게 하면 게임 오브젝트가 라이트 프로브를 샘플링하기 위해 사용하는 위치가 조정됩니다.
- 렌더링 레이어를 활성화합니다.
    - Lighting 창의 Adaptive Probe Volumes 패널에서 **Rendering Layer Masks**를 설정하여 APV가 각 라이트 프로브에 렌더링 레이어 마스크를 할당하도록 합니다.
- Baking Set 프로퍼티를 조정합니다.
    - 볼륨을 추가해도 해결되지 않으면 Lighting 창의 Adaptive Probe Volumes 패널에서 Virtual Offset 및 Dilation 설정을 조정합니다.
- Probe Adjustment Volume 컴포넌트를 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

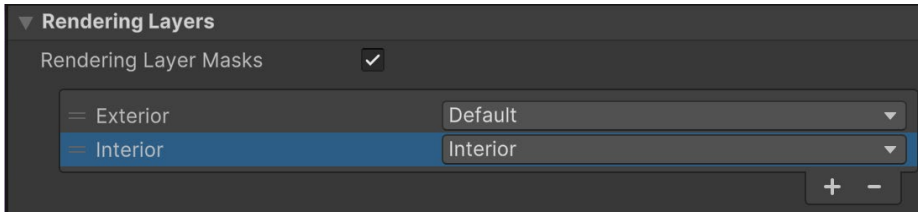
일부 픽셀에서 텐트 내부와 외부의 프로브가 블렌딩되며 발생하는 현상입니다. 픽셀의 값을 보간할 때 어떤 프로브가 사용되는지 **Window > Analysis > Rendering Debugger > Probe Volumes > Debug Probe Sampling**을 통해 확인할 수 있습니다.



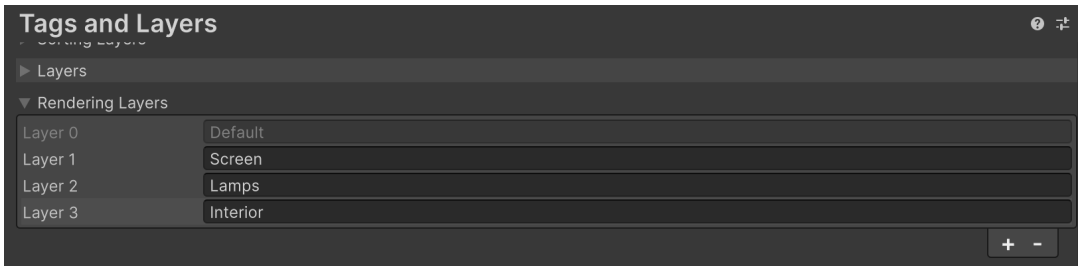
픽셀의 보간된 프로브 확인

이 문제를 해결하는 방법 중 하나는 **볼륨**을 활용하여 **Adaptive Probe Volume Options** 오버라이드로 런타임에 APV가 샘플링되는 방식을 변경하는 것입니다. **NormalBias**와 **ViewBias** 설정을 통해 샘플링 위치를 조정합니다. NormalBias는 노멀 방향을 따라 벽에서 멀어지는 방향으로 위치를 조정하고, ViewBias는 카메라를 향해 카메라와 같은 방향의 벽으로 위치를 조정합니다. 볼륨의 두 프로퍼티를 변경하면 그에 따라 샘플링 위치와 가중치가 업데이트되어 조명 결과와 **Debug Probe Sampling 뷰**에서 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 탭**에서 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

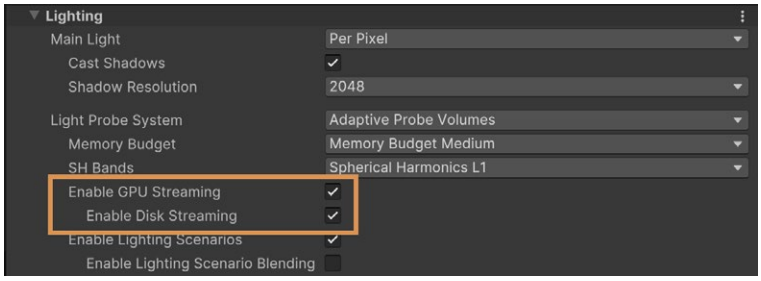
APV를 통한 문제 해결 방법을 자세히 알아보려면 여기를 참고하세요.

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.

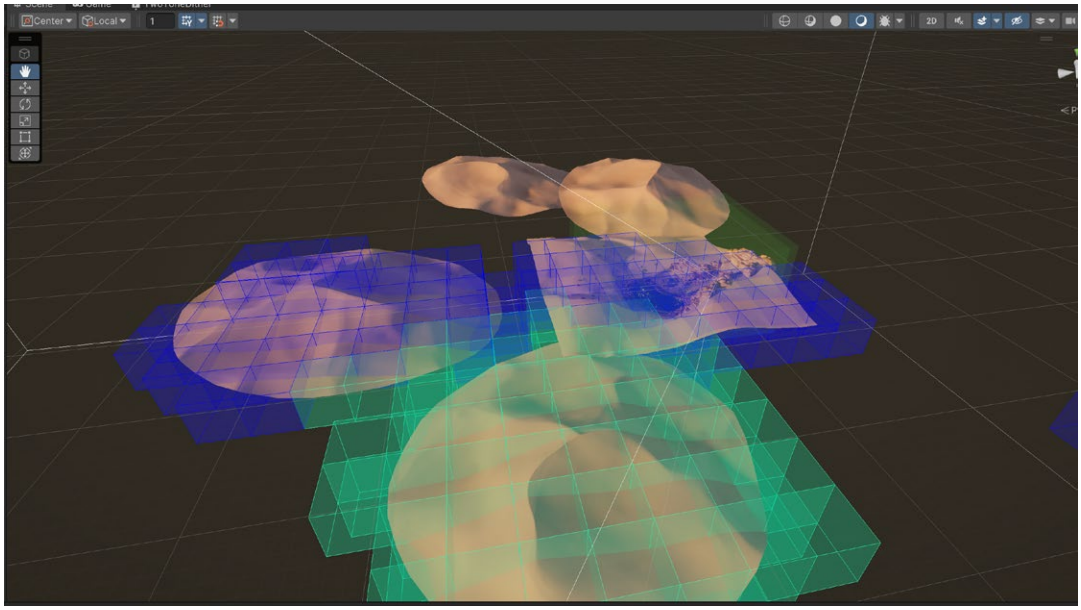


같은 창에서 스트리밍 설정도 구성할 수 있습니다. 자세히 알아보려면 URP 에셋 섹션을 참고하세요.

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

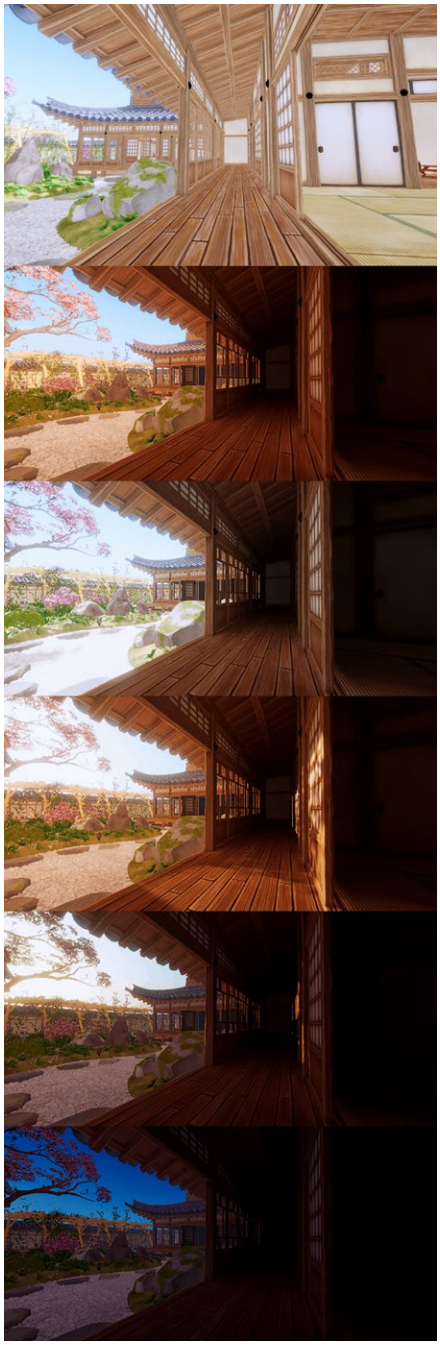
스카이 오클루전

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.

a. 가장 위쪽의 이미지는 하늘 조명이 런타임에 변해야 하기 때문에 하늘 조명을 베이크할 수 없을 때 발생하는 문제를 보여 줍니다. 이 이미지는 베이크 없이 앰비언트 프로브만 사용되므로 결과물의 품질이 좋지 않습니다.

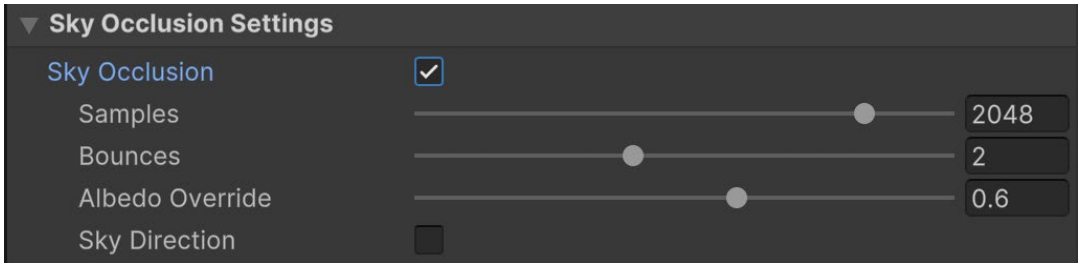
b. 두 번째부터 다섯 번째 이미지에는 앰비언트 프로브와 함께 스카이 오클루전이 사용되었습니다. 스카이 오클루전을 비활성화한 채 일반적인 APV 베이크만으로 조명을 적용할 수도 있겠지만, 그럴 경우 런타임에 조명이 변하지 않을 것입니다.

씬에 스카이 오클루전을 사용한 예시. 이미지 출처: Unity 에셋 스토어에 등록된 7stars의 Azure[Sky] Dynamic Skybox 패키지.

스카이 오클루전을 활성화하는 방법은 다음과 같습니다.

1. **Progressive GPU 라이트매퍼**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.

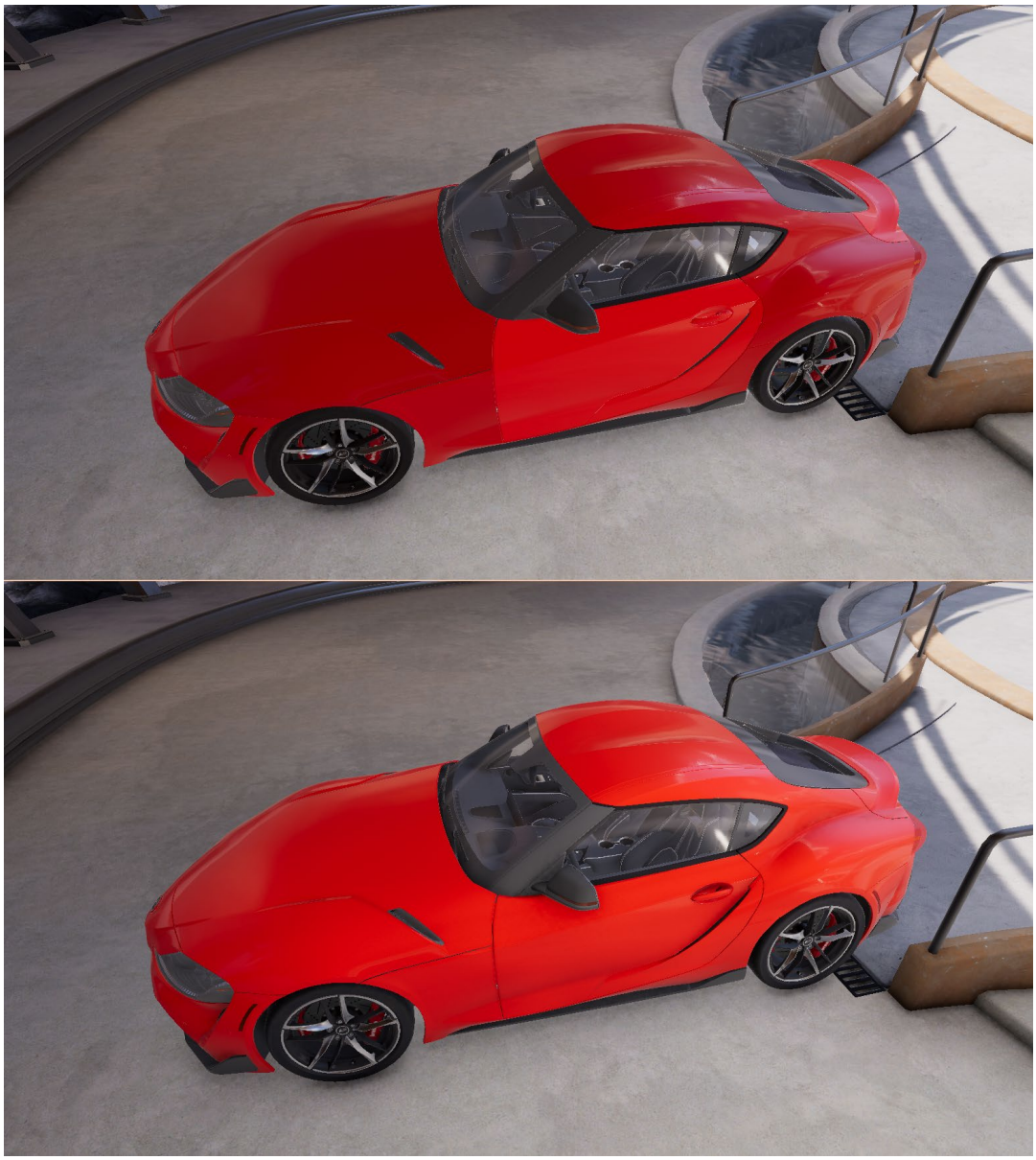
참고:
이제 URP에서 프로브에 버텍스별 품질 샘플링을 지원합니다. 특히 저사양 기기에서 성능을 향상하는 데 유용한 기능입니다. 샘플링 모드를 설정하려면 Lighting 섹션의 **URP 에셋**을 사용합니다. **Advanced Properties**를 활성화해야 해당 옵션이 표시됩니다. Lighting 패널 오른쪽 상단의 줄임표를 눌러 활성화할 수 있습니다. Advanced Properties를 활성화하면 **SH Evaluation Mode** 드롭다운이 표시됩니다.

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

위에서 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
지오메트리가 변화하는 경우 프로브를 배치하고 이동하는 데 시간이 소모됨	지오메트리 변화에 따라 빠르게 배치하고 쉽게 업데이트할 수 있음
오브젝트를 비추는 데 하나의 보간된 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.

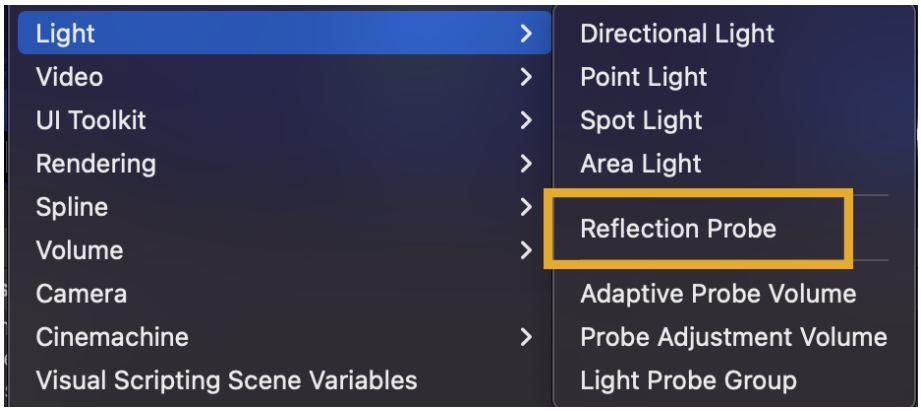
반사 프로브

Maya나 Blender와 같은 레이트레이싱 툴에서는 반사 표면의 각 프레임 픽셀 값을 정확히 계산하는 데 시간이 걸릴 수 있습니다. 실시간 렌더러의 경우 이 과정에 지나치게 긴 시간이 소요되기 때문에 간소화된 방법이 사용되는 경우가 많습니다.

실시간 렌더러의 반사에는 환경 맵(사전 렌더링된 큐브맵)이 사용되며, Unity는 SkyManager를 사용하는 기본 맵을 제공합니다. 씬 내 모든 위치의 반사 소스로 단일 맵을 사용하면 반사 효과가 자연스럽지 않을 수 있습니다. 이 섹션에 나온 로봇을 예로 들겠습니다. 캐릭터의 금속 부품에 항상 하늘이 반사되어 보인다면, 격납고 안에서는 하늘을 볼 수 없으므로 상당히 이상하게 보일 것입니다. 이럴 때 반사 프로브가 유용합니다.

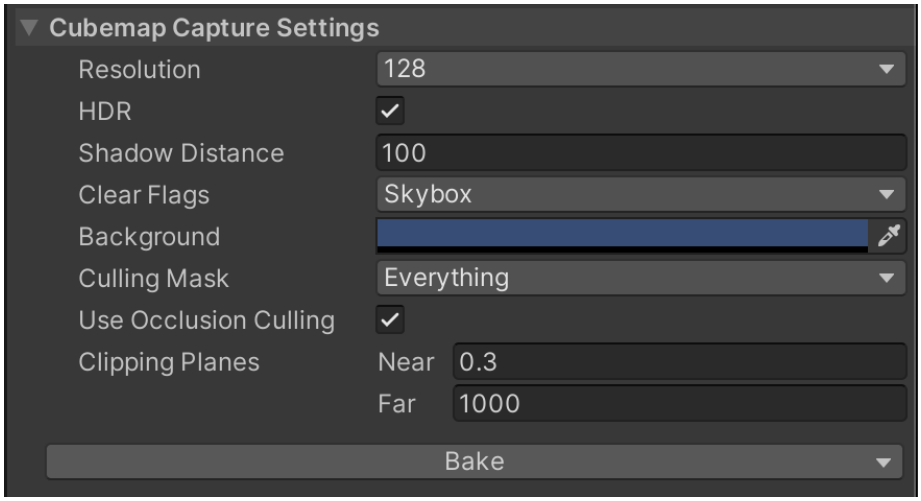
Reflection Probe 컴포넌트는 사전 렌더링된 큐브맵으로 씬의 핵심 위치에 놓여 있습니다. 하나의 씬에서 반사 프로브를 여러 개 사용할 수도 있습니다. 동적 오브젝트는 씬에서 이동하면서 가장 가까운 반사 프로브를 선택해 반사 기준으로 사용할 수 있습니다. 씬이 프로브 사이에서 블렌딩되도록 설정할 수도 있습니다.

Reflection Probe 컴포넌트를 추가하려면 **계층** 창에서 오른쪽 클릭하고 **Light > Reflection Probe**를 선택합니다.



Reflection Probe 컴포넌트 추가

그런 다음 프로브의 위치를 지정하고 설정을 조정합니다. 프로브가 올바른 위치에 배치되고 설정이 조정되었으면 **Bake**를 클릭해 큐브맵을 생성합니다.



Reflection Probe 컴포넌트의 설정

다음 이미지는 FPS 샘플: 더 인스펙션에 사용된 두 개의 반사 프로브를 나타냅니다. 하나는 격납고 안에, 하나는 밖에 있습니다.

각 반사 프로브는 큐브맵 텍스처에 주변 이미지를 캡처합니다.

반사 프로브 블렌딩

반사 프로브에는 블렌딩이라는 뛰어난 기능이 있습니다. **Renderer Asset Settings** 패널에서 블렌딩을 활성화할 수 있습니다. 포워드+ 경로를 선택한 경우 렌더러 에셋 설정과 관계없이 항상 블렌딩이 활성화됩니다.

블렌딩은 반사 오브젝트가 한 영역에서 다른 영역으로 이동할 때 프로브의 큐브맵을 서서히 페이드아웃하면서 다른 큐브맵을 페이드인합니다. 이렇게 점진적으로 전환하면 움직이는 오브젝트가 두 반사 프로브의 경계를 넘을 때 갑자기 완전히 다른 반사가 적용되는 현상을 방지할 수 있습니다.

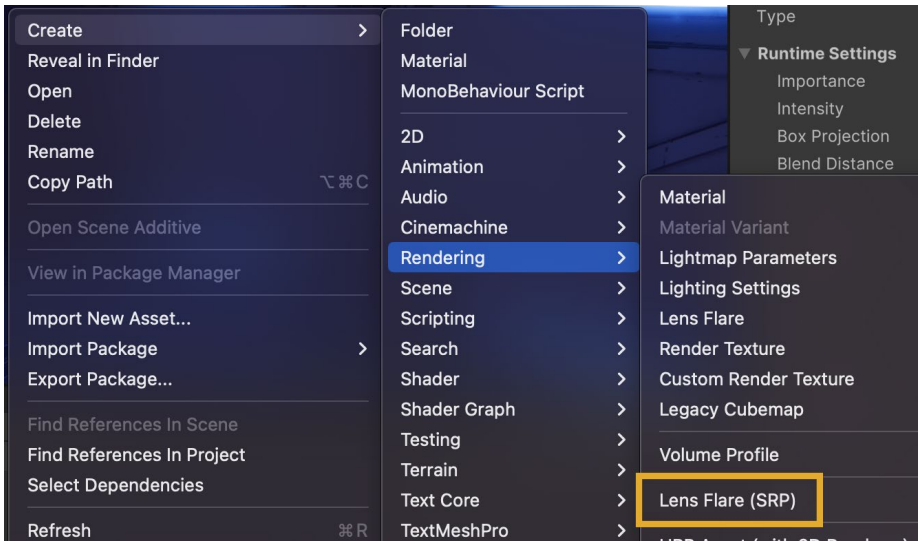
박스 투영

반사 큐브맵은 보통 모든 오브젝트로부터 무한하게 먼 거리에 있는 것으로 간주됩니다. 오브젝트가 회전하면 다양한 각도의 큐브맵이 표시되지만, 오브젝트가 반사된 주변 환경을 기준으로 더 가깝게 또는 더 멀리 이동하는 것은 불가능합니다. 야외 씬에서는 괜찮지만 실내 씬에서는 이에 따른 한계가 드러납니다. 방 내부의 벽은 결코 무한하게 떨어져 있지 않으며, 오브젝트가 근접할수록 벽의 반사는 커져야 합니다.

Box Projection 옵션을 사용하면 프로브로부터 일정 거리만큼 떨어진 반사 큐브맵을 생성해 큐브맵 벽과의 거리에 따라 오브젝트에 다양한 크기의 반사를 표시할 수 있습니다. 환경 큐브맵의 크기는 Box Size 프로퍼티에 따라 프로브의 영향 영역을 기준으로 결정됩니다. 예를 들어 방 내부를 반사하는 프로브라면 크기를 방 면적과 일치하도록 설정해야 합니다.

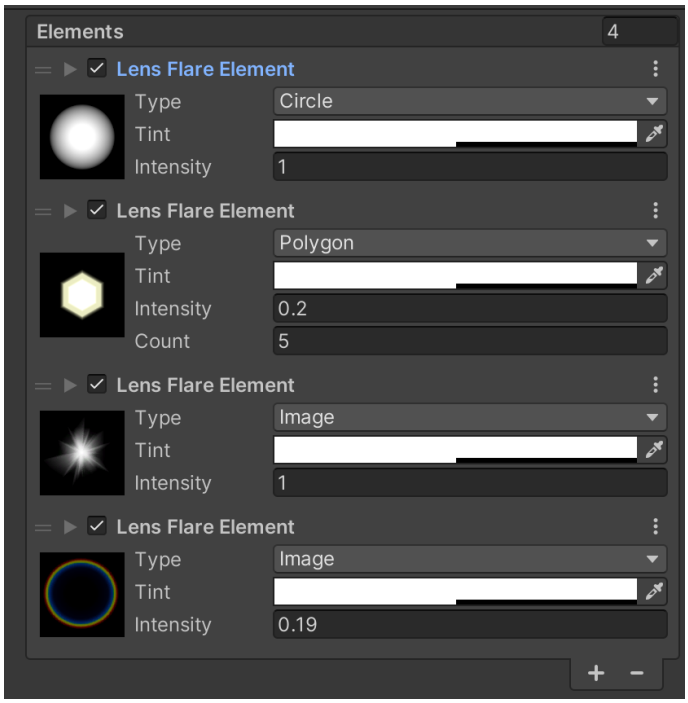
렌즈 플레어

빌트인 렌더 파이프라인을 사용하다가 URP로 넘어왔다면 렌즈 플레어 생성 워크플로가 URP에 맞게 업데이트되었을 것입니다. 설정의 첫 단계는 Lens Flare (SRP) 데이터 에셋을 만드는 것입니다. **프로젝트** 창을 오른쪽 클릭하고 적절한 Assets 폴더에서 **Create > Rendering > Lens Flare (SRP)**를 선택합니다.



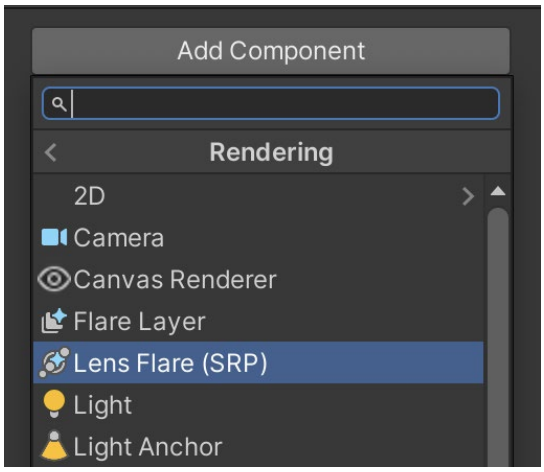
Lens Flare (SRP) 데이터 에셋 만들기

이 에셋을 사용하여 플레어의 형태를 설정할 수 있습니다. **Type** 을 Circle, Polygon, Image 에셋으로 설정하고 Tint 및 Intensity 설정을 조정합니다.



렌즈 플레어 요소 추가 및 설정

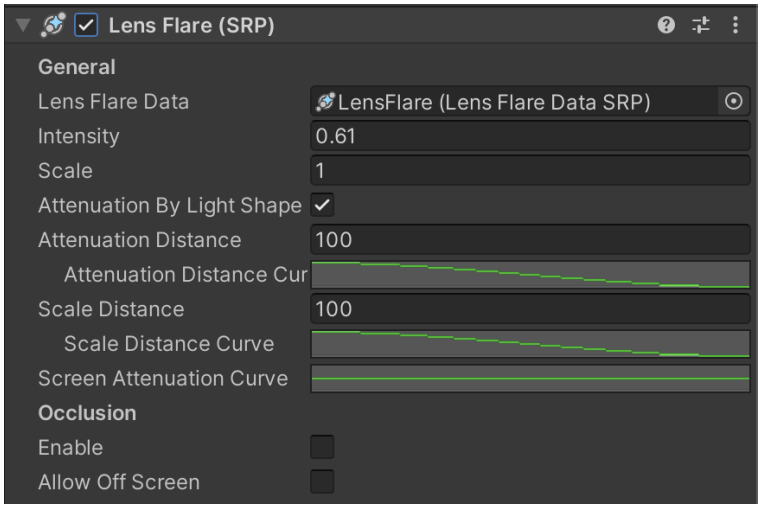
렌즈 플레어를 렌더링하려면 플레어를 유발하는 광원을 선택한 다음 **Add Component > Rendering > Lens Flare (SRP)**를 선택합니다.



렌즈 플레어 렌더링 설정

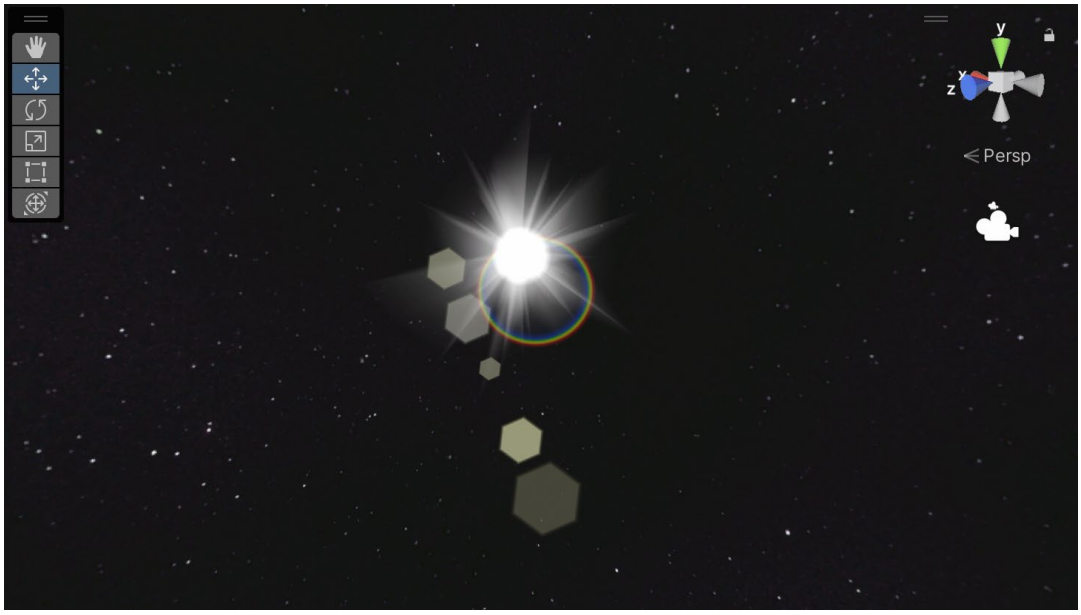
Lens Flare 데이터 에셋을 선택합니다.

이 컴포넌트의 **Settings** 패널에서, 직접 만든 **Lens Flare 데이터 에셋**을 **Lens Flare Data 프로퍼티**에 할당합니다.



Lens Flare (SRP) 컴포넌트 설정

렌즈 플레어를 사용하면 렌즈 플레어의 추가 및 조정 워크플로가 매우 유연하다는 점을 느낄 수 있을 것입니다.



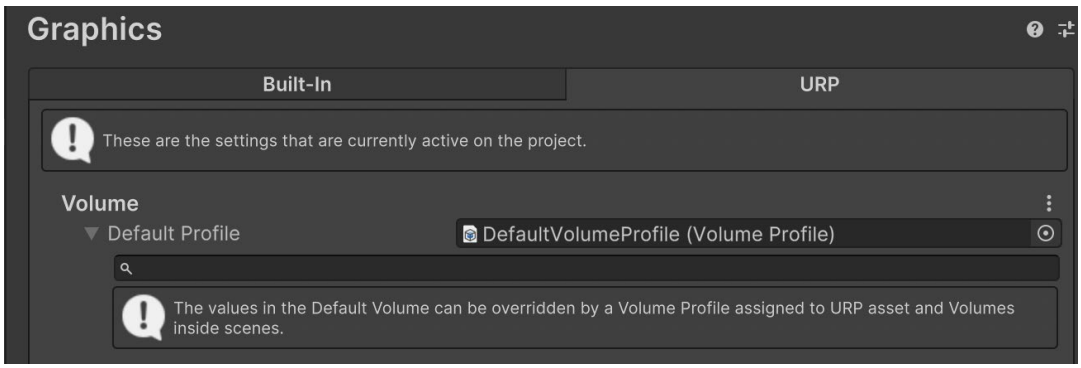
렌즈 플레어 예시

스크린 공간 렌즈 플레어

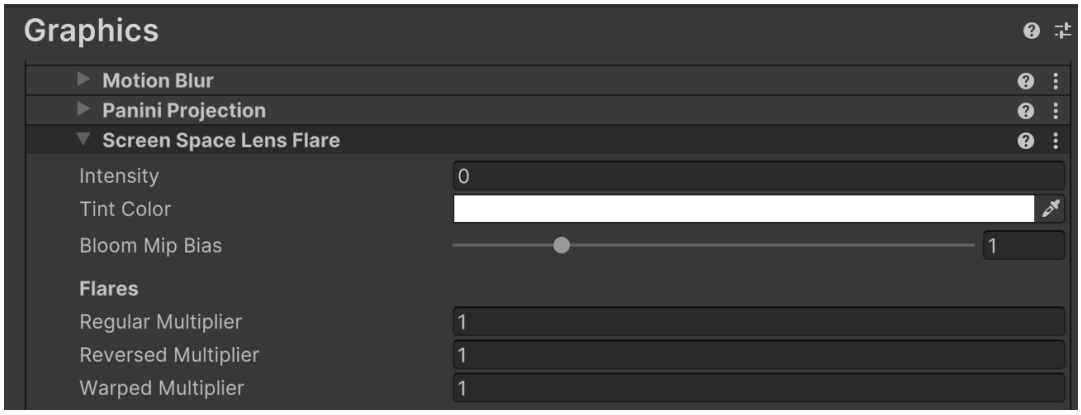
여러 광원에 렌즈 플레어를 설정하려면 시간이 오래 걸릴 수 있습니다. Unity 6에는 새로운 포스트 프로세싱 기법인 SSLF(스크린 공간 렌즈 플레어) 오버라이드가 추가되었습니다. 이 기법을 사용하면 밝은 스페큘러 하이라이트나 발광 메시와 같은 모든 밝은 표면에서 플레어를 생성할 수 있습니다. 반면 Lens Flare (SRP) 효과는 광원에서만 플레어를 생성할 수 있습니다. 스크린 공간 렌즈 플레어는 사후 제작 기법을 사용합니다.

SSLF를 사용하는 방법은 다음과 같습니다.

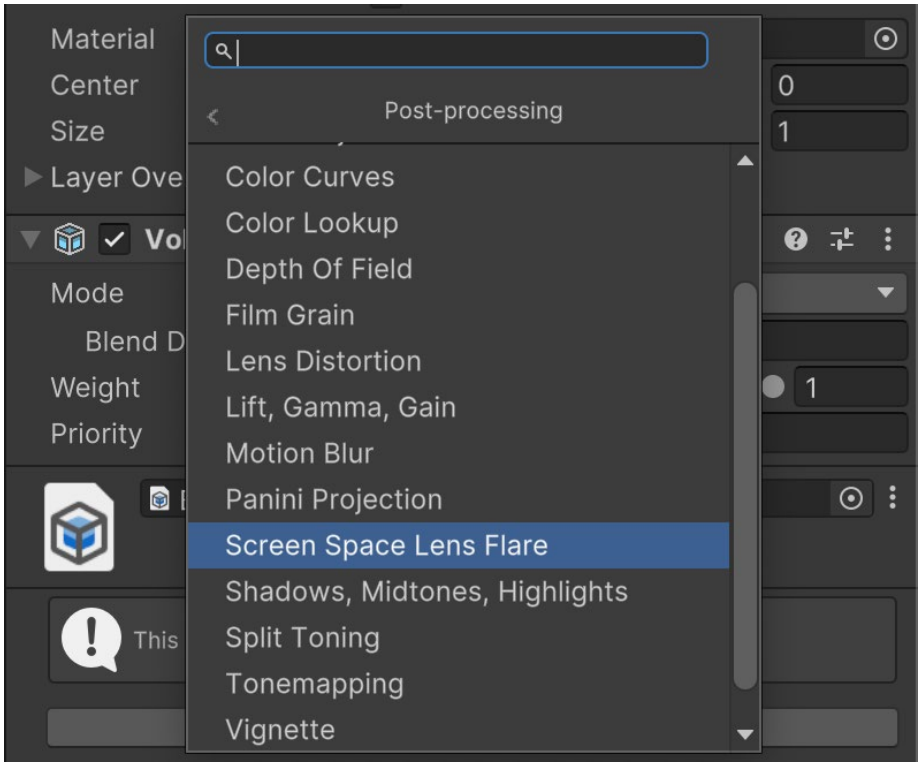
1. SSLF는 포스트 프로세싱 필터이므로 적용하려면 Volume 컴포넌트를 사용합니다. 씬에 Volume 컴포넌트를 추가하거나 Unity 6의 새로운 Default Volume을 사용합니다. Default Volume의 설정은 **Edit > Project Settings > Graphics**에서 변경할 수 있습니다.



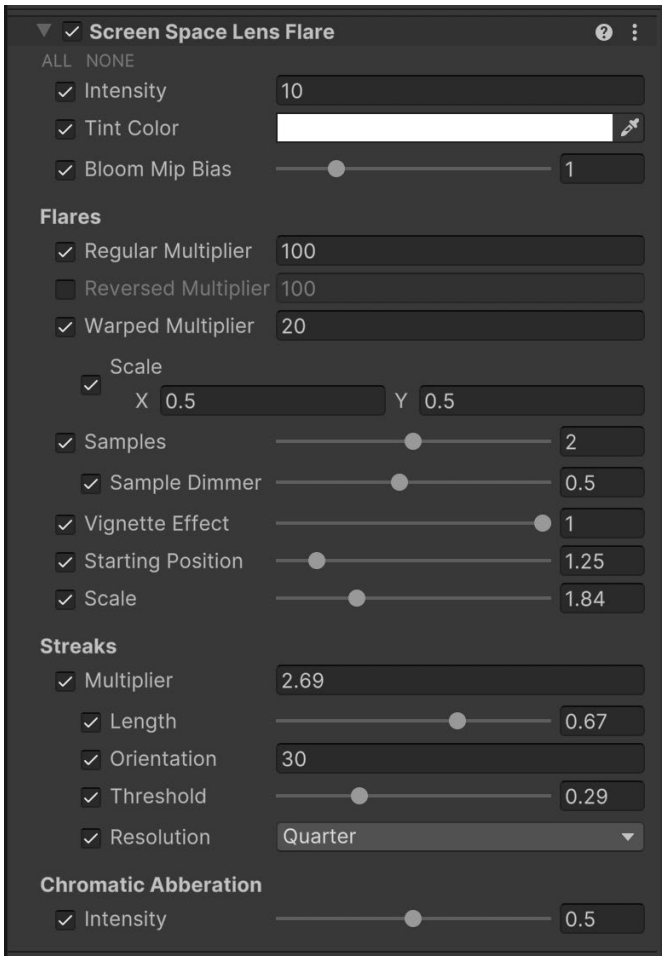
2. Default Volume을 사용하는 경우 설정 패널에서 Screen Space Lens Flare 오버라이드를 찾습니다.



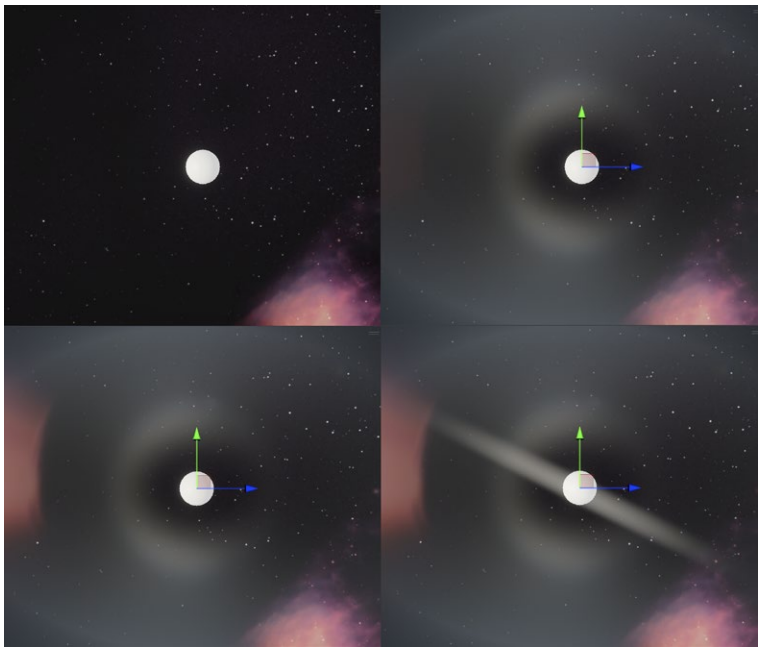
3. 씬 볼륨을 사용하는 경우 오버라이드를 추가할 수 있도록 Volume Profile을 생성합니다.
그런 다음 **Post-processing > Screen Space Lens Flare**를 통해 오버라이드를 추가합니다.



4. 이제 씬에서 원하는 스타일을 얻기 위해 설정을 테스트할 수 있습니다.
강도는 0보다 크게 설정해야 합니다.



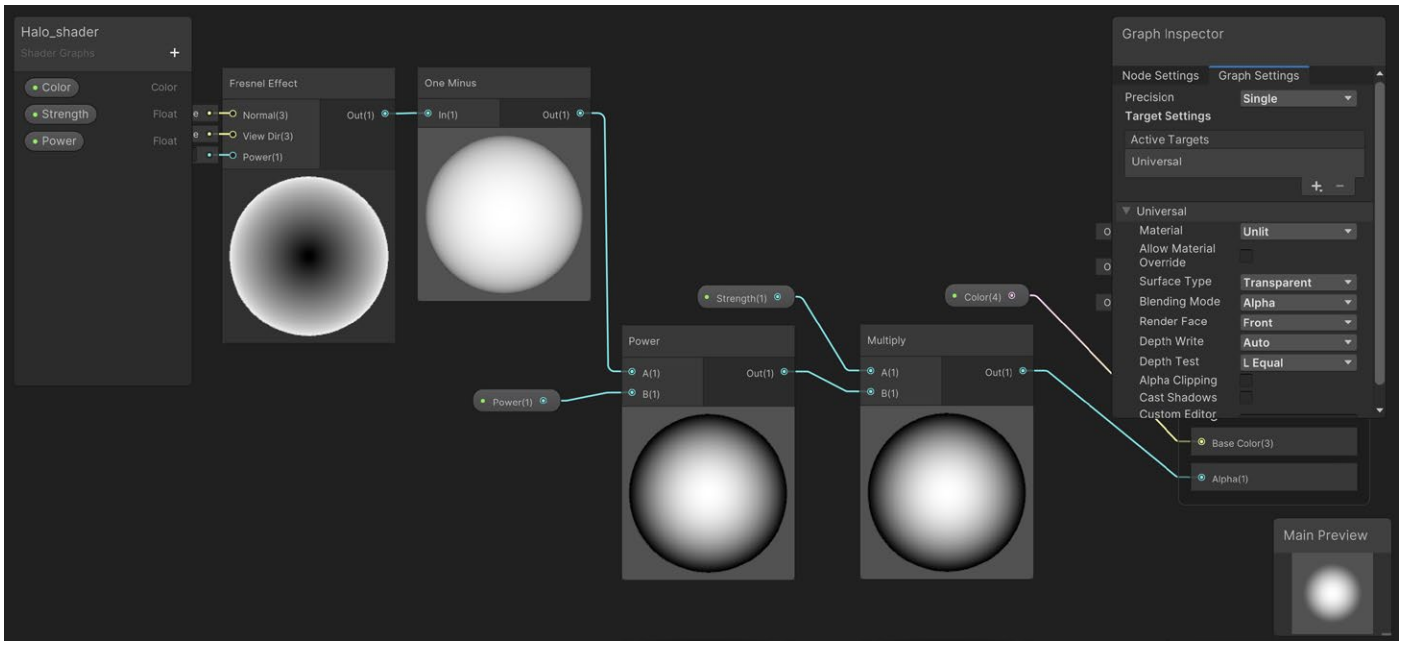
SSLF와 Lens Flare (SRP) 컴포넌트를 결합하여 더 세밀하게 제어할 수도 있습니다.



왼쪽 상단부터 시계 방향으로: SSLF 비활성화, 플레어, 플레어 및 왜곡된 플레어 및 잔상, 플레어 및 왜곡된 플레어

광원 헤일로

URP에서 광원에 **Draw Halo** 프로퍼티를 사용할 수는 없으나, 빌보드를 사용하면 쉽게 모방할 수 있습니다. 구체의 알파 투명도를 설정하는 방법도 있습니다. 아래에서 첫 번째 이미지는 그러한 셰이더의 Shader Graph를, 두 번째 이미지는 그 결과를 표시합니다. Shader Graph를 사용하여 이러한 셰이더를 만드는 방법은 추가 툴 챕터에서 자세히 알아보세요.

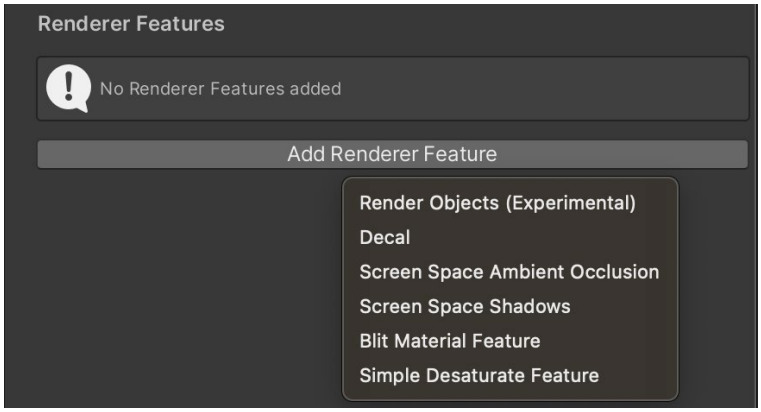


Shader Graph를 사용한 프레넬 투명도

위 Shader Graph의 셰이더가 적용된 머티리얼의 구체를 사용한 헤일로 효과

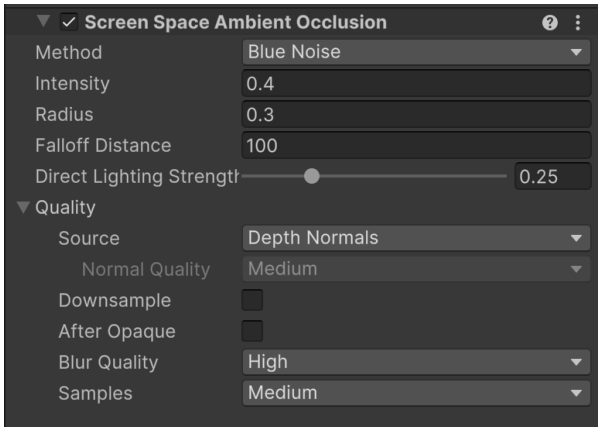
스크린 공간 앰비언트 오클루전

기본적으로 주변광은 지오메트리를 고려하지 않기 때문에 높은 수준의 주변광으로 인해 렌더링이 적절하지 않게 이루어질 수 있습니다. 현실에서는 두 오브젝트 사이의 간격이 좁으면 간격이 넓을 때보다 더 어두울 것입니다. 앰비언트 오클루전을 사용하면 Unity 프로젝트에서 이 문제를 해결할 수 있습니다. URP에서 이를 사용하려면 URP 에셋에서 사용되는 렌더러를 선택합니다. **Add Renderer Feature**로 이동하여 **SSAO(Screen Space Ambient Occlusion)**를 선택합니다.



Add Renderer Feature에서 SSAO 선택

다음과 같이 기본 SSAO 설정을 사용하거나 필요에 따라 조정합니다.



SSAO 설정

SSAO 프로퍼티를 살펴보겠습니다.

— **Method**: 이 프로퍼티는 SSAO 효과가 사용하는 노이즈의 유형을 정의합니다.

— **Intensity**: 이 프로퍼티는 어두워지는 효과의 강도를 정의합니다.

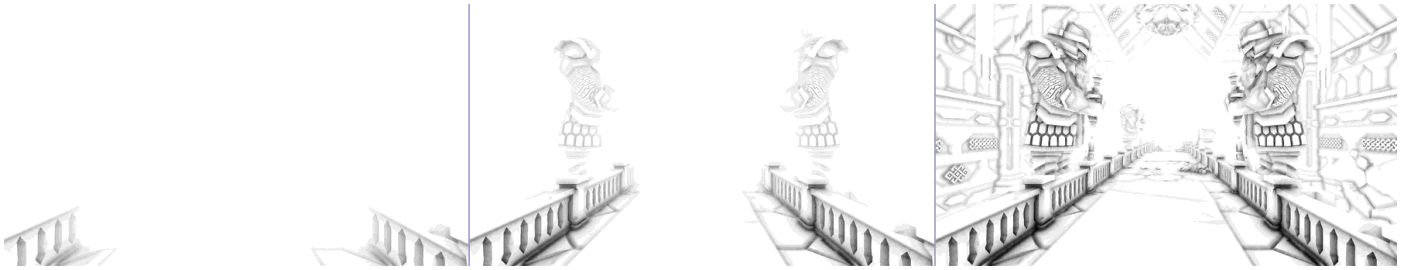
— **Radius**: Unity가 앰비언트 오클루전 값을 계산할 때, SSAO 효과는 현재 픽셀에서 이 반지름 내의 노멀 텍스처 샘플을 가져옵니다. Radius 값이 낮을수록 SSAO 렌더러 기능이 소스 픽셀에서 가까운 픽셀을 샘플링하므로 성능이 향상됩니다.

— **Falloff Distance:** 카메라에서 이 거리보다 멀리 있는 오브젝트에는 SSAO가 적용되지 않습니다. 멀리 있는 오브젝트가 많은 씬에서는 이 값이 낮을수록 성능이 향상됩니다.

— **Direct Lighting Strength:** 이 프로퍼티는 직접 조명에 노출된 영역에서 효과가 얼마나 눈에 띄는지 정의합니다.

— **Quality**: 품질 설정에 대해 자세히 알아보려면 기술 자료**를 참고하세요.**

- — Source
- — Downsample
- — After Opaque
- — Blur Quality
- — Samples



앰비언트 오클루전 텍스처만으로 구성된 씬에서 다양한 감쇠 거리 비교

SSAO는 좁은 간격에 셰이딩을 추가합니다. 왼쪽의 이미지 3개를 살펴보겠습니다.

상단 이미지에는 SSAO가 없습니다. 가운데 이미지는 계산된 SSAO를 표시하고, 하단 이미지는 SSAO의 결과를 표시합니다. 그라인더와 저울이 책상과 만나는 지점에서 테두리가 더 선명해진 것을 볼 수 있습니다.

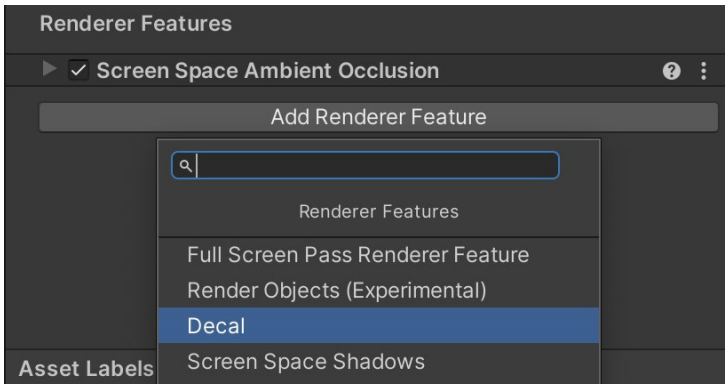
SSAO는 포스트 프로세싱 기술로, 자세한 내용은 이 가이드의 후반부에서 다루겠습니다.

세 가지 버전의 유령의 방. 상단: SSAO 비활성화, 중간: SSAO 적용, 하단: SSAO로 렌더링

데칼

Decal Projector

Decal Projector 컴포넌트는 메시에 디테일을 추가하는 효과적인 방법입니다. 총알 구멍, 발자국, 간판, 균열 등의 요소에 적합합니다. 투영 프레임워크를 사용하므로 울퉁불퉁하거나 휘어진 표면에도 적용할 수 있습니다. URP에서 데칼 프로젝터를 사용하려면 렌더러 데이터 에셋에 **데칼 렌더러 기능**을 추가해야 합니다.

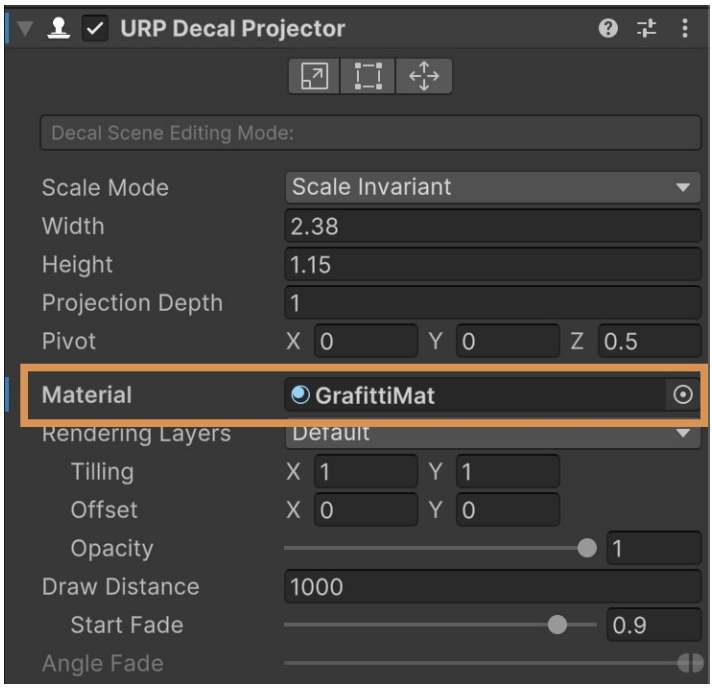


데칼 렌더러 기능 추가

대부분의 용도에는 기본 설정을 사용하면 됩니다.

이제 씬에 데칼을 사용할 준비가 완료되었습니다. 계층 뷰에서 오른쪽 클릭하고 **Rendering > URP Decal Projector**를 선택하여 데칼을 생성합니다. 프로젝터는 표면에 흰색 정사각형을 투영하는 Decal 머티리얼을 기본적으로 사용합니다. 평소에 사용하는 툴로 프로젝터의 위치와 방향을 지정하세요. 인스펙터에서 **Width**, **Height**, **Projection Depth**를 조정하면 됩니다.

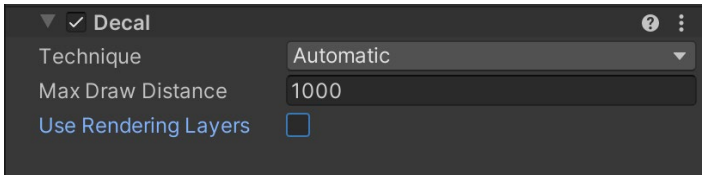
데칼을 커스터마이즈하려면 **Shader Graph > Decal** 셰이더를 사용하여 머티리얼을 생성합니다. 그런 다음 URP Decal Projector에 할당합니다.



Decal Projector 컴포넌트 설정

Decal Projector의 인스펙터에는 세 가지 **Editing Mode** 버튼이 있습니다. 각각 Scale, Crop, Pivot/UV 모드이며, 자세한 내용은 여기에서 확인할 수 있습니다.

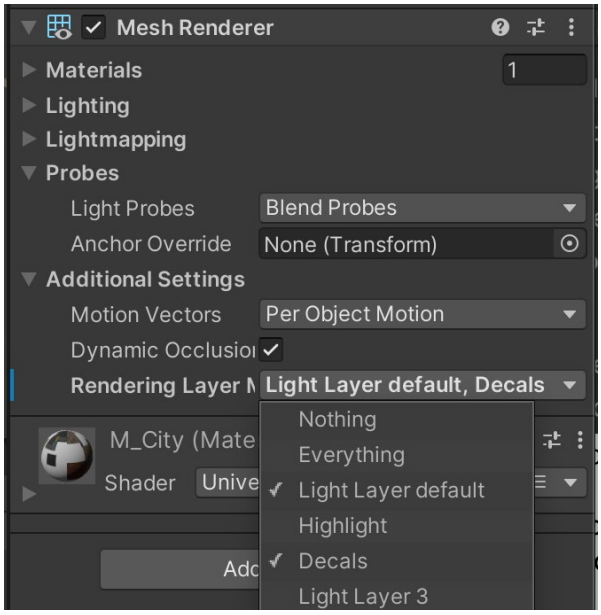
프로젝터는 기본적으로 절두체 내의 모든 표면에 영향을 미칩니다. 데칼 렌더러 기능에는 **Use Rendering Layers** 설정이 포함되어 있습니다. 특정 메시만 타게팅하려면 이 설정을 활성화하세요.



데칼 렌더러 기능 설정

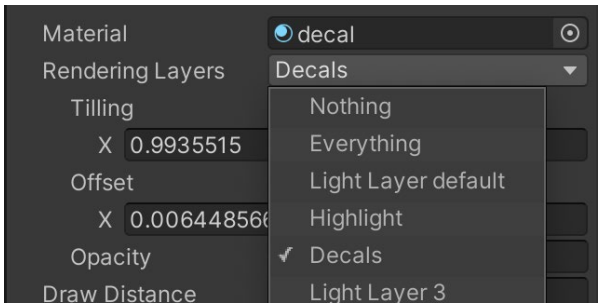
이 렌더링 옵션을 설정하고 사용하는 방법을 자세히 알아보려면 렌더링 레이어 섹션을 다시 살펴보세요. 데칼을 설정하는 단계는 다음과 같습니다.

1. **Edit > Project Settings ... > Tags and Layers > Rendering Layers**에서 렌더링 레이어의 이름을 지정합니다.
2. 프로젝터를 적용할 메시를 하나 이상 선택합니다. 인스펙터에서 **Mesh Renderer > Additional Settings > Rendering Layer Mask**로 이동하여 이름을 지정한 렌더링 레이어를 마스크에 추가합니다.



Mesh Renderer의 Rendering Layer Mask에 렌더링 레이어 추가

3. URP Decal Projector를 선택하고 인스펙터의 Rendering Layers 프로퍼티에서 이름을 지정한 렌더링 레이어를 선택합니다.



아래는 데칼이 없는 씬, 데칼이 있는 씬, 렌더링 레이어로 벽에만 데칼이 투영되도록 제한한 씬을 비교한 이미지입니다.

왼쪽부터 데칼이 없는 이미지, 모든 오브젝트에 데칼이 적용된 이미지, 렌더링 레이어를 사용하여 벽에만 데칼을 적용한 이미지

셰이더

이 섹션은 기존의 커스텀 셰이더를 URP에서 작동하도록 전환하거나 Shader Graph 없이 커스텀 셰이더를 코드로 작성하려는 사용자에게 적합합니다. 또한 URP에서 빌트인 렌더 파이프라인으로 기본 및 고급 셰이더를 모두 포팅하는 데 필요한 정보를 제공합니다. 이 섹션에 포함된 표에는 사용 가능한 HLSL 셰이더 함수, 매크로 등의 유용한 샘플이 나와 있으며 기타 유용한 함수가 포함된 관련 include에 대한 링크도 제공됩니다.

셰이더 코딩 경험이 있는 경우, include를 사용하면 단순하면서도 효율적인 셰이더를 작성하기 위해 HLSL을 활용할 방안을 명확하게 이해할 수 있습니다. 여기에서 안내하는 정보를 살펴보고 나면 URP에 셰이더를 포팅하는 작업을 더 손쉽게 진행할 수 있을 것입니다.

Shader Graph로 직접 만든 버전의 셰이더를 사용하는 식으로 접근할 수도 있습니다. Shader Graph 소개는 추가 툴 섹션에서 확인할 수 있습니다.

URP와 빌트인 렌더 파이프라인의 셰이더 비교

아래 코드 스니핏에서 볼 수 있듯이 URP 셰이더는 ShaderLab 구조를 사용합니다. 셰이더 코딩 경험이 있다면 Property, SubShader, Tags, Pass 등이 모두 익숙할 것입니다.

```
SubShader {
  Tags {“RenderPipeline” = “UniversalPipeline” }
   Pass {
      HLSLPROGRAM
      ...
      ENDHLSL
   }
}
```

SubShader 블록의 기본 구조

가장 두드러지는 특징은 URP 셰이더의 SubShader 태그에서 `“RenderPipeline” = “UniversalPipeline”` 키-값 쌍을 사용한다는 것입니다.

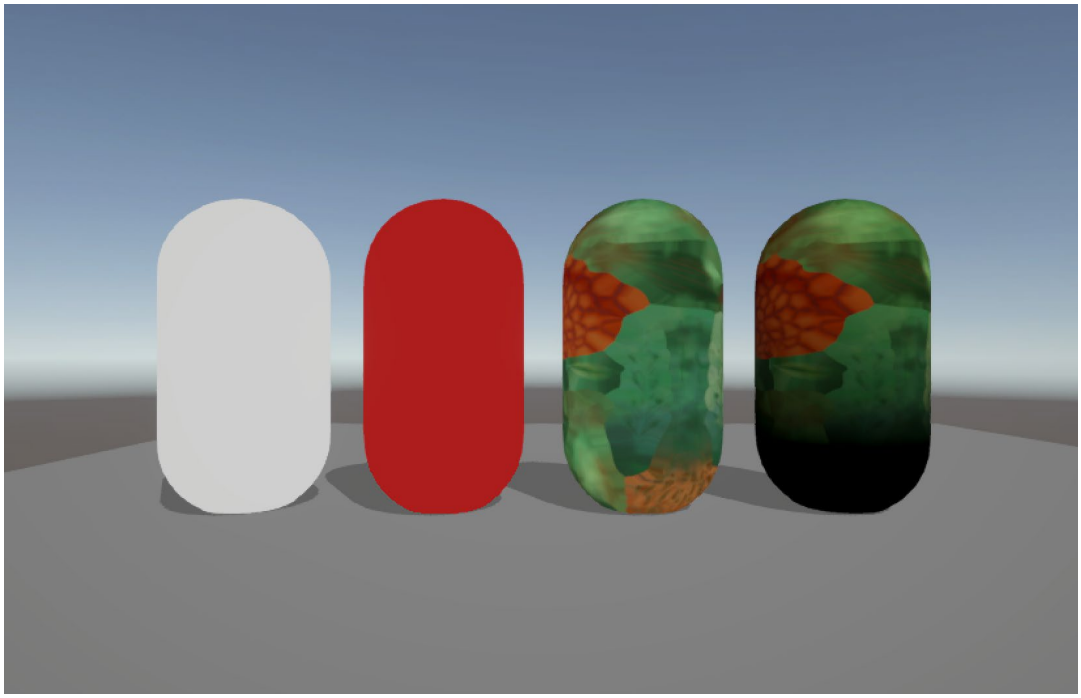
이름이 RenderPipeline 인 SubShader 태그는 이 SubShader가 사용될 렌더 파이프라인을 Unity에 전달합니다. `UniversalPipeline` 값은 Unity가 URP에서 이 SubShader를 사용해야 함을 나타냅니다.

렌더 Pass 코드를 살펴보면 `HLSLPROGRAM / ENDHLSL` 매크로 사이에 셰이더 코드가 포함되어 있습니다. URP에서 이러한 패스 내의 셰이더 코드는 HLSL로 작성됩니다.

Unity는 GPU에서 지원되는 첫 번째 SubShader 블록을 사용합니다. 첫 번째 SubShader 블록에 `“RenderPipeline” = “UniversalPipeline”` 태그가 없으면 URP에서 실행되지 않습니다. Unity에서는 대신 SubShader가 있는 경우 다음 순서로 실행합니다. 어떤 SubShader도 지원되지 않는 경우 Unity에서는 잘 알려진 마젠타 오류 셰이더를 렌더링합니다.

커스텀 셰이더

Create > Shader > Unlit Shader를 통해 커스텀 셰이더 생성을 시작하면 빌트인 렌더 파이프라인용 템플릿을 사용하게 됩니다. 이 템플릿은 SRP 배처와 호환되지 않는 코드를 사용합니다. 왜 이렇게 작동할까요? 대부분의 경우 개발자는 URP용 커스텀 셰이더를 생성할 때 Shader Graph를 사용하는 것이 더 편리하다고 느낄 것입니다. 하지만 많은 개발자는 수년 동안 Unity를 사용하며 생성한 셰이더를 URP에서 가장 효과적으로 사용하는 방법을 알고 싶어 할 것입니다. 이 섹션에서는 5가지 예제와 함께 올바른 방향을 제시합니다. 에디터에서 씬을 확인하려면 앞서 언급한 예시 저장소의 **Shaders > Shaders**에서 확인할 수 있습니다. 자세히 알아보려면 기술 자료를 참고하세요.



URP의 셰이더(왼쪽부터): 하드 코딩된 흰색 언릿, 프로퍼티로 색상을 설정한 언릿, 텍스처를 사용한 언릿, 바닥 메인 광원을 사용한 단순한 램버트 조명. 그림자에 액세스하고 그림자를 사용하는 방법에 대한 자세한 내용은 아래의 'Shadows' 섹션 참조.

Unlit

먼저 가장 단순한 셰이더인 기본 언릿 예시를 살펴보겠습니다.

- 눈에 보이는 픽셀이 이 셰이더를 사용하면 동일한 색상으로 설정됩니다.
- 색상은 하드 코딩되므로 **Properties**는 비어 있습니다.
- **Tags**에서 **RenderType**은 **Opaque**, **RenderPipeline**은 **UniversalPipeline**으로 설정됩니다. #include로는 유용한 함수와 매크로를 많이 제공하는 **Core.hlsl**을 사용합니다.
    - 빌트인 셰이더에 익숙하다면 UnityCG.cginc와 유사하게 활용할 수 있을 것입니다. `TransformObjectToHClip` 함수가 Core.hlsl include에 포함되어 있습니다. 이 함수는 오브젝트 공간을 동차 공간으로 전환합니다.
- 버텍스 셰이더 vert는 프래그먼트 셰이더 frag에 전달되는 **Varyings** 구조체의 positionHCS 값을 설정합니다.
- HLSL은 셰이더 단계 간 전달되는 모든 값에 시맨틱을 사용합니다. 시맨틱은 셰이더 입력 또는 출력에 연결된 문자열이며 파라미터의 용도에 관한 정보를 전달합니다.
    - 예를 들어 **Attributes** `positionOS`의 시맨틱은 `POSITION`입니다. 컴파일 시 버텍스 셰이더의 POSITION은 오브젝트 공간 내 버텍스의 위치입니다.

프래그먼트 셰이더는 단순히 흰색을 반환하므로 위 이미지의 왼쪽 흰색 캡슐처럼 표시됩니다.

```
Shader “CustomURP/Unlit”
{
  Properties
  { }
  SubShader
  {
      Tags { “RenderType” = “Opaque” “RenderPipeline” = “UniversalPipeline” }
      Pass
      {
          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
          {
              half4 customColor = half4(1, 1, 1, 1);
              return customColor;
          }

          ENDHLSL
      }
  }
}
```

Unlit Color

— 두 번째 셰이더 예시인 Unlit Color에서는 색상을 설정할 프로퍼티를 추가해야 합니다. `[MainColor]` 속성을 사용하세요. 필수는 아니지만 이 프로퍼티가 어떻게 쓰이는지 Unity에 알려 주는 데 도움이 됩니다. 프로퍼티 이름은 _BaseColor, 인스펙터 내 이름은 **Base Color**, 유형은 `Color`입니다.

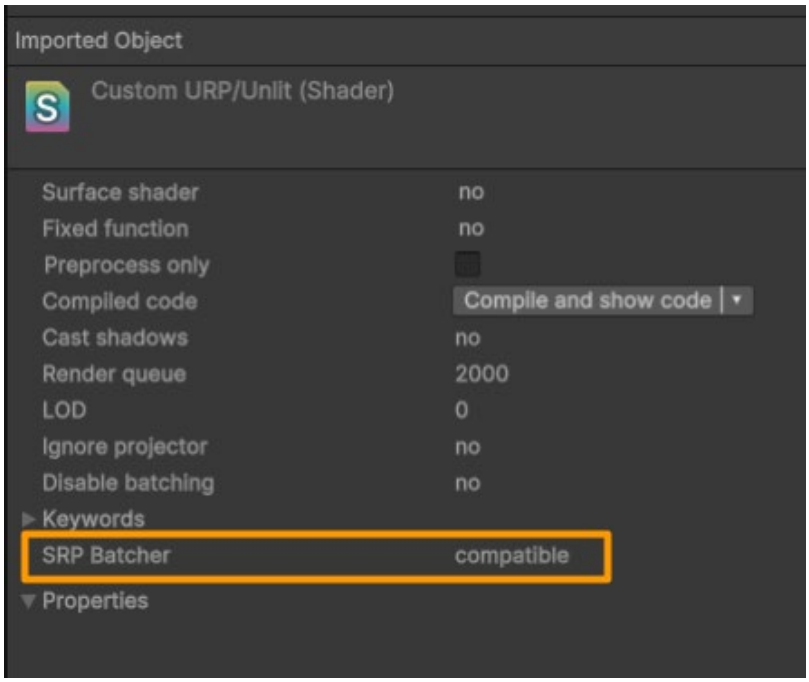
— URP의 셰이더 변수는 SRP 배처와 호환되어야 합니다. 이 예시에서는 `CBUFFER_START(UnityPerMaterial)` 및 `CBUFFER_END` 매크로(`CBUFFER` 블록) 사이에 선언하여 호환성을 실현합니다.

— 이제 프래그먼트 셰이더가 `_BaseColor`의 값을 반환합니다.

```
Shader “CustomURP/UnlitColor”
{
   Properties
   {
       [MainColor] _BaseColor(“Base Color”, Color) = (1, 1, 1, 1)
   }
   SubShader
   {
       Tags { “RenderType” = “Opaque” “RenderPipeline” = “UniversalPipeline” }
       Pass
       {
           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;
           };
           CBUFFER_START(UnityPerMaterial)
               half4 _BaseColor;
           CBUFFER_END
           Varyings vert(Attributes IN)
           {
               Varyings OUT;
               OUT.positionHCS = TransformObjectToHClip (IN.positionOS.xyz);
               return OUT;
           }
```

```
            half4 frag() : SV_Target
            {
                return _BaseColor;
            }
            ENDHLSL
        }
    }
}
```

셰이더가 SRP 배처와 호환되는지 확인하려면 셰이더를 선택한 후 인스펙터에서 해당 셰이더의 SRP 배처 호환 여부를 확인합니다.



셰이더와 SRP 배처의 호환 여부를 표시하는 인스펙터

Unlit Textured

이미지의 세 번째 예시에서는 텍스처를 사용합니다.

— 셰이더의 `_BaseMap` 프로퍼티(인스펙터에서는 **Base Map**)는 `[MainTexture]` 속성을 사용하고 2D 유형입니다.

— 텍스처를 사용하려면 보간된 UV 값을 버텍스에서 프래그먼트 셰이더로 전달해야 합니다. Attributes와 Varyings 구조체에 시맨틱이 `TEXCOORD0`인 `float2 uv`를 추가합니다.

— `_BaseMap` 프로퍼티를 받는 `TEXTURE2D`와 `SAMPLER(sampler_BaseMap)` 매크로를 `CBUFFER` 블록 직전에 추가합니다.

— 타일링과 오프셋이 작동하려면 2D 프로퍼티의 이름에 접미사 _ST가 붙은 float4 정의가 필요합니다. 버텍스 셰이더에서 사용하는 TRANSFORM_TEX 매크로를 사용할 때 필수적인 요소입니다. 이 매크로는 예시의 IN.uv와 같은 버텍스 UV 값과 TEXTURE2D 매크로로 선언된 2D 프로퍼티를 받습니다. 그러면 타일링과 오프셋이 지원됩니다.

— 프래그먼트 셰이더는 SAMPLE_TEXTURE2D 매크로를 사용합니다. 이 매크로는 Texture2D, 샘플러, UV 값의 3가지 파라미터를 받고 색상 값을 반환합니다. 이 값은 frag 메서드에서 반환됩니다.

```
{
    Properties
    {
        [MainColor] _BaseColor(“Base Color”, Color) = (1, 1, 1, 1)
        [MainTexture] _BaseMap(“Base Map”, 2D) = “white” {}
    }
    SubShader
    {
        Tags { “RenderType” = “Opaque” “RenderPipeline” = “UniversalPipeline” }
        Pass
        {
            HLSLPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #include “Packages/com.unity.render-pipelines.universal/ShaderLibrary/Core.hlsl”
            struct Attributes
            {
                float4 positionOS   : POSITION;
                float2 uv           : TEXCOORD0;
            };
            struct Varyings
            {
                float4 positionHCS  : SV_POSITION;
                float2 uv           : TEXCOORD0;
            };
            TEXTURE2D(_BaseMap);
            SAMPLER(sampler_BaseMap);
            CBUFFER_START(UnityPerMaterial)
                half4 _BaseColor;
                float4 _BaseMap_ST;
            CBUFFER_END
            Varyings vert(Attributes IN)
            {
```

```
                Varyings OUT;
                OUT.positionHCS = TransformObjectToHClip (IN.positionOS.xyz);
                OUT.uv = TRANSFORM_TEX(IN.uv, _BaseMap);
                return OUT;
            }
            half4 frag(Varyings IN) : SV_Target
            {
                half4 color = SAMPLE_TEXTURE2D(_BaseMap, sampler_Base Map, IN.uv);
                return color;
            }
            ENDHLSL
        }
    }
}
```

Lit Simple

— URP 셰이더에서 조명을 사용하려면 include **Lighting.hlsl**을 추가합니다. 이 파일은 **Core.hlsl**과 같은 폴더에 있습니다.

— 이 예시 셰이더는 가장 강도가 높은 방향 광원을 메인 광원으로 사용합니다. 또한 램버트 기법을 사용하여 버텍스 수준에서 계산하고 프래그먼트 셰이더에서 보간된 값을 사용합니다. 따라서 시맨틱이 TEXCOORD2인 `half3 lightAmount`를 `Varyings` 구조체에 추가해야 합니다.

— 버텍스 셰이더에는 `GetVertexNormalInputs` 함수를 사용하는 `VertexNormalInputs`가 있습니다. 이 함수는 오브젝트 공간의 노멀을 월드 공간으로 전환합니다.

  — 원점이 중심에 있는 오브젝트는 오브젝트 공간 위치를 노멀 대신 사용할 수 있습니다. `VertexNormalInputs` 구조체에는 `float4` 값인 `normalWS`가 있습니다.

— `GetMainLight` 함수를 사용해 메인 광원의 세부 정보를 가져옵니다. 이 함수는 광원 색상과 방향을 비롯한 데이터가 담긴 `Light` 구조체를 반환합니다.

— 마지막으로 `vert` 함수에서는 `LightingLambert` 함수를 사용하여 현재 버텍스의 조명을 반환합니다. `LightingLambert` 함수는 광원 색상, 광원 방향, 월드 공간 노멀의 3가지 파라미터를 받습니다. 이제 함수를 호출하기 위한 모든 데이터가 준비되었습니다. 함수는 `half3`를 반환합니다.

— 프래그먼트 셰이더는 `SAMPLE_TEXTURE2D` 매크로를 사용합니다. 이 매크로는 `Texture2D`, 샘플러, `float2` UV 값의 3가지 파라미터를 받습니다. 여기에 `lightAmount`에 `w` 값 `1`을 추가하여 `half4`로 만든 값을 곱합니다.

```
{
     [MainColor] _BaseColor(“Base Color”, Color) = (1, 1, 1, 1)
     [MainTexture] _BaseMap(“Base Map”, 2D) = “white” {}
 }
 SubShader
 {
     Tags { “RenderType” = “Opaque” “RenderPipeline” = “UniversalPipeline” }
     Pass
     {
         HLSLPROGRAM
         #pragma vertex vert
         #pragma fragment frag
         #include “Packages/com.unity.render-pipelines.universal/ShaderLibrary/Core.hlsl”
         #include “Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl”
         struct Attributes
         {
             float4 positionOS   : POSITION;
             float2 uv           : TEXCOORD0;
         };
         struct Varyings
         {
             float4 positionHCS  : SV_POSITION;
             float2 uv           : TEXCOORD0;
             half3 lightAmount   : TEXCOORD2;
         };
         TEXTURE2D(_BaseMap);
         SAMPLER(sampler_BaseMap);
         CBUFFER_START(UnityPerMaterial)
             half4 _BaseColor;
             float4 _BaseMap_ST;
         CBUFFER_END
         Varyings vert(Attributes IN)
         {
             Varyings OUT;
             OUT.positionHCS = TransformObjectToHClip(IN.positionOS.xyz);
             OUT.uv = TRANSFORM_TEX(IN.uv, _BaseMap);
             VertexNormalInputs positions =
             GetVertexNormalInputs(IN.positionOS);
             Light light = GetMainLight();
```

```
                OUT.lightAmount = LightingLambert(light.color, light.direction, positions.normalWS.xyz);
                return OUT;
            }
            half4 frag(Varyings IN) : SV_Target
            {
                half4 color = SAMPLE_TEXTURE2D(_BaseMap, sampler_BaseMap, IN.uv) * half4(IN.lightAmount, 1);
                return color;
            }
            ENDHLSL
        }
    }
}
```

Shadows

— 그림자의 강도를 제어하려면 인스펙터에서 **Shadow Strength**로 표시되는 _ShadowStrength 프로퍼티를 추가합니다. Float 값이며 초기 값은 0.5입니다. 이 값을 CBUFFER 블록에 추가합니다.

  — 그림자를 사용하려면 pragma multi_compile을 추가하고 다음 코드를 추가합니다. _MAIN_LIGHT_SHADOWS

  — MAIN_LIGHT_SHADOWS_CASCADE

  — _MAIN_LIGHT_SHADOWS_SCREEN

— Varyings 구조체에 시맨틱이 TEXCOORD3인 float4 값 shadowCoords를 추가합니다.

— vert 함수에서는 GetVertexPositionInputs 함수를 사용하여 오브젝트 공간을 월드 공간으로 전환합니다. 이제 GetShadowCoord 함수를 사용하여 버텍스 위치를 섀도우 맵의 위치로 전환할 수 있습니다. 이 값을 Varyings 값인 shadowCoords로 저장합니다.

— frag 함수에서는 shadowAmount가 MainLightRealtimeShadow 함수를 사용하여 보간된 shadowCoord를 전달합니다. Strength는 1에서 _ShadowStrength를 뺀 값으로 설정됩니다. color 값은 _BaseColor가 strength와 shadowAmount 중 큰 값으로 조절된 값입니다.

```
{
    [MainColor] _BaseColor(“Base Color”, Color) = (1, 1, 1, 1)
    _ShadowStrength(“Shadow Strength”, Float) = 0.5
}
SubShader
{
    Tags { “RenderType” = “AlphaTest” “RenderPipeline” =  “UniversalPipeline” }
    Pass
    {
        HLSLPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        #pragma multi_compile _ _MAIN_LIGHT_SHADOWS _MAIN_LIGHT_
        SHADOWS_CASCADE _MAIN_LIGHT_SHADOWS_SCREEN
        #include “Packages/com.unity.render-pipelines.universal/
        ShaderLibrary/Lighting.hlsl”
        struct Attributes
        {
            float4 positionOS  : POSITION;
        };
        struct Varyings
        {
            float4 positionCS  : SV_POSITION;
            float4 shadowCoords : TEXCOORD3;
        };
        CBUFFER_START(UnityPerMaterial)
            half4 _BaseColor;
            float _ShadowStrength;
        CBUFFER_END
        Varyings vert(Attributes IN)
        {
            Varyings OUT;
            OUT.positionCS = TransformObjectToHClip(IN.positionOS.xyz);
            VertexPositionInputs positions =
            GetVertexPositionInputs(IN.positionOS.xyz);
            // 버텍스 위치를 섀도우 맵의 위치로 전환
            float4 shadowCoordinates = GetShadowCoord(positions);
            OUT.shadowCoords = shadowCoordinates;
            return OUT;
        }
        half4 frag(Varyings IN) : SV_Target
        {
```

```
                // 그림자 좌표의 섀도우 맵 값 가져오기
                half shadowAmount = MainLightRealtimeShadow(IN.shadowCoords);
                half strength = 1.0 - _ShadowStrength;
                half4 color = _BaseColor * max(strength, shadowAmount);

                return color;
            }

            ENDHLSL
        }
    }
}
```

커스텀 셰이더를 URP로 업그레이드 하려면 약간의 작업이 필요합니다. 아래에 소개된 함수 표를 활용해 보세요.

- 커스텀 URP 셰이더에서 위치 변환하기
- 커스텀 URP 셰이더에서 카메라 사용하기
- 커스텀 URP 셰이더에서 조명 사용하기
- 커스텀 URP 셰이더에서 그림자 사용하기

Unity 프로젝트를 사용하여 커스텀 언릿 빌트인 셰이더를 URP로 전환하는 방법을 이 단계별 동영상 튜토리얼에서 알아보세요.

튜토리얼 보기

참고: Cyanilux의 튜토리얼을 읽어 보세요. URP 셰이더를 작성하려는 사용자에게 유용한 리소스입니다.

파이프라인 콜백

SRP는 C# 스크립트를 사용하여 렌더링 프로세스의 어느 단계에서든 코드를 추가할 수 있는 뛰어난 기능을 제공합니다. 다음과 같은 단계에서 스크립트를 삽입할 수 있습니다.

— 그림자 렌더링

— 프리패스 렌더링

— G버퍼 렌더링

— 디퍼드 광원 렌더링

— 불투명 렌더링

— 스카이박스 렌더링

— 투명도 렌더링

— 포스트 프로세싱 렌더링

유니버설 렌더러 데이터 에셋에 대한 인스펙터에 있는 **Add Renderer Feature** 옵션을 통해 렌더링 프로세스에서 스크립트를 삽입할 수 있습니다. URP를 사용할 때 유니버설 렌더러 데이터 오브젝트와 URP 에셋이 있다는 것을 기억하세요. URP 에셋에는 최소 하나의 유니버설 렌더러 데이터 오브젝트가 할당된 렌더러 목록이 있습니다. 이는 **Project Settings > Graphics > Scriptable Render Pipeline Settings**에서 할당한 에셋입니다.

서로 다른 씬에서 여러 설정 에셋으로 실험하는 경우, 다음 스크립트를 메인 카메라에 연결하면 유용합니다. 인스펙터에서 **Pipeline Asset**을 설정하면 새 씬이 로드될 때 에셋이 전환됩니다.

```
using UnityEngine;
using UnityEngine.Rendering;
using UnityEngine.Rendering.Universal;
[ExecuteAlways]
public class AutoLoadPipelineAsset : MonoBehaviour
{
       public UniversalRenderPipelineAsset pipelineAsset;
       // 첫 번째 프레임 업데이트 전에 Start가 호출됨
       void OnEnable()
       {
           if (pipelineAsset)
           {
                GraphicsSettings.defaultRenderPipeline = pipelineAsset;
             QualitySettings.renderPipeline = pipelineAsset;
           }
       }
}
```

씬 로드 시 유니버설 렌더 파이프라인 에셋을 전환하는 스크립트

다음 섹션에서는 두 가지 유형의 렌더러 기능을 다룹니다. 하나는 아티스트에게, 하나는 숙련된 프로그래머에게 적합합니다.

오브젝트 렌더링

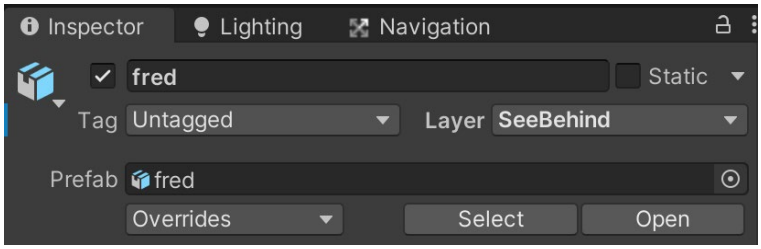
게임에서는 환경 오브젝트 뒤에 가려져 플레이어 캐릭터의 모습을 볼 수 없게 되는 문제가 흔하게 발생합니다. 캐릭터가 항상 보이도록 카메라를 이동하거나 최대한 개방적으로 환경을 조정할 수 있지만, 이러한 옵션을 항상 사용할 수 있는 것은 아닙니다. 이때 아래 이미지에 보이는 대로 캐릭터와 카메라 사이에 환경 모델이 표시될 때 캐릭터의 실루엣을 표시하면 좋습니다.

환경 모델에 의해 캐릭터에 마스킹이 적용되었을 때 실루엣 표시

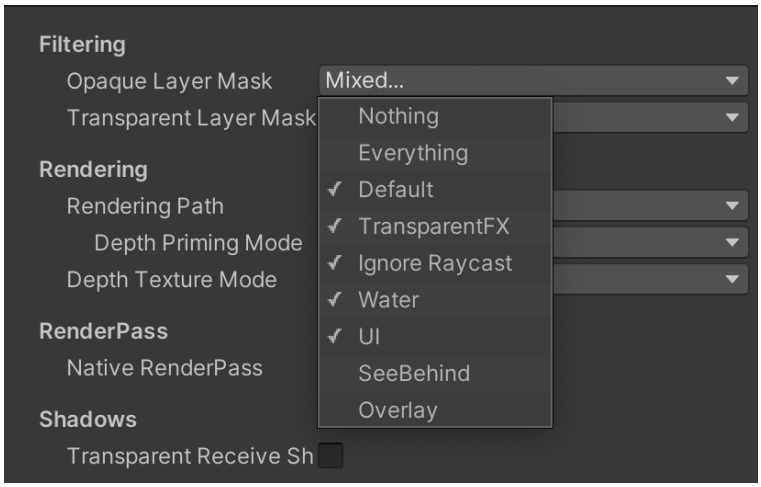
실루엣을 만드는 방법은 다음과 같습니다.

1. 우선 캐릭터에 마스킹을 적용할 때 사용할 머티리얼이 필요합니다. 머티리얼을 만들고 셰이더를 **Universal Render Pipeline > Lit 또는 Unlit**으로 설정합니다(이전 이미지에는 Lit 옵션이 나와 있음). **Surface Inputs > Base Map** 색상을 설정합니다. 이 예시에서 해당 머티리얼을 캐릭터라고 칭하겠습니다.

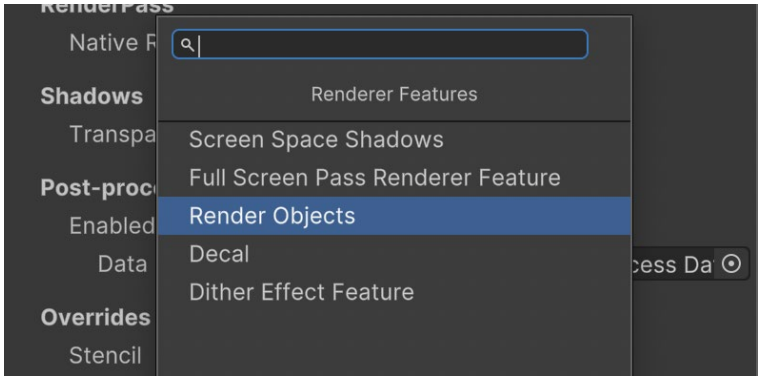
2. 필요 이상으로 캐릭터를 렌더링하지 않기 위해 특수 레이어에 배치하겠습니다. 캐릭터를 선택하고 **SeeBehind** 레이어를 레이어 목록에 추가한 후 캐릭터에 대해 선택합니다.



3. URP 에셋에서 사용되는 **Renderer Data** 오브젝트를 선택합니다. **Opaque Layer Mask**로 이동하여 SeeBehind 레이어를 제외합니다. 그러면 캐릭터가 사라집니다.

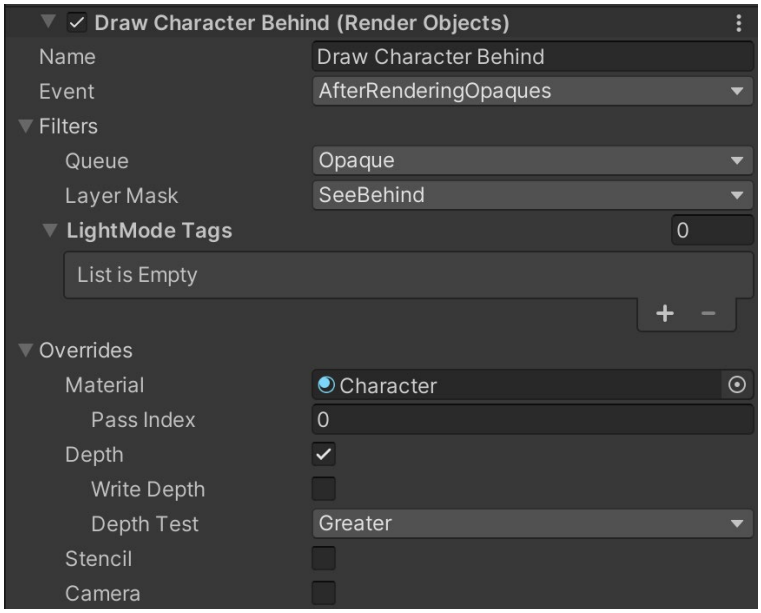


4. **Add Renderer Feature**를 클릭하고 **Render Objects**를 선택합니다.

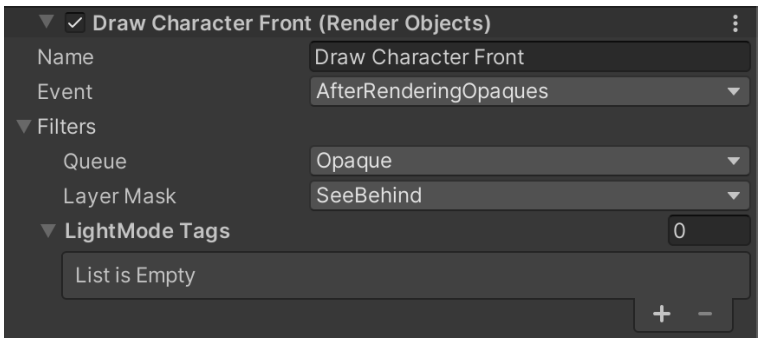


5. 이 렌더 오브젝트의 **패스**에 대한 설정을 입력합니다. 이름을 지정하고 렌더가 트리거되어야 하는 시점을 선택합니다. 이 예시에서는 AfterRenderingOpaques라고 칭합니다.

   Layer Mask를 캐릭터에 대해 선택된 레이어인 **SeeBehind** 레이어로 설정합니다. **Overrides**를 확장하고 **Override Mode**를 **Material**로 설정합니다. 1단계에서 만든 머티리얼을 선택합니다. 렌더링할 때 뎁스 버퍼에 작성하여 업데이트하지 않고도 **Depth**를 사용할 수 있습니다. 그리고 **Depth Test**를 **Greater**로 설정하면 렌더링된 픽셀까지의 거리가 현재 뎁스 버퍼에 저장된 거리보다 카메라로부터 더 멀 때만 이 패스가 렌더링을 수행합니다.



6. 이 단계에서는 다른 오브젝트 뒤에 있는 캐릭터의 실루엣만 볼 수 있으며 전체 뷰일 때는 캐릭터가 보이지 않습니다. 이 문제를 해결하려면 또 다른 **Render Objects** 기능을 추가합니다. 이번에는 Overrides 패널을 업데이트하지 않아도 되며, 다른 오브젝트에 의해 마스킹이 적용되어 있지 않을 때 패스가 캐릭터를 드로우합니다.



실루엣 기법은 SRP 워크플로를 활용해 렌더 파이프라인에 간편하게 요소를 더할 수 있다는 장점을 알 수 있는 좋은 예시입니다.

렌더 그래프 시스템

렌더 그래프 시스템을 사용하면 커스텀 SRP를 유지 관리하기 쉬운 모듈식으로 저작할 수 있습니다. RenderGraph API를 사용하면 커스텀 SRP 렌더 패스를 개괄적으로 표현한 렌더 그래프를 생성하여 렌더 파이프라인의 렌더 패스에서 리소스를 어떻게 사용할지 명시적으로 나타낼 수 있습니다. 렌더 그래프는 그래프 형태의 시스템이 아닙니다.

렌더 패스를 이런 방식으로 표현하면 두 가지 장점이 있습니다. 첫째는 렌더 파이프라인 구성이 간소화되는 것이며, 둘째는 렌더 그래프 시스템이 렌더 파이프라인의 일부를 효율적으로 관리할 수 있어 런타임 성능이 향상되는 것이죠.

렌더 그래프 시스템을 사용하려면 보통의 커스텀 SRP에서 요구되는 것과는 다른 방식으로 코드를 작성해야 합니다.

주요 원칙

RenderGraph API로 렌더 패스를 작성하기 전에 알아야 할 기본 원칙은 다음과 같습니다.

- 리소스를 더 이상 사용자가 직접 관리하지 않으며, 대신 렌더 그래프 시스템 전용 핸들을 사용합니다. 모든 **RenderGraph** API는 이러한 핸들을 사용하여 리소스를 조작합니다. 렌더 그래프에서 관리하는 리소스 유형은 **RTHandles, ComputeBuffers**, **RendererLists**입니다.
- 실제 리소스 레퍼런스는 렌더 패스의 실행 코드에서만 액세스할 수 있습니다.
- 이 프레임워크에서는 렌더 패스를 명시적으로 선언해야 합니다. 각 렌더 패스는 어떤 리소스에서 읽고 쓰는지 명시해야 합니다.
- 렌더 그래프의 실행 간에는 지속성이 없습니다. 즉, 렌더 그래프의 한 실행에서 생성한 리소스는 다음 실행으로 이어지지 않습니다.
- 한 프레임에서 다음 프레임으로 이어지는 등의 지속성이 필요한 리소스의 경우, 일반적인 리소스처럼 렌더 그래프 밖에서 생성한 다음 임포트할 수 있습니다. 종속성 추적 측면에서는 다른 렌더 그래프 리소스와 동일하게 동작하지만, 그래프가 수명을 관리하지는 않습니다.
- 렌더 그래프는 텍스처 리소스로 대부분 RTHandles를 사용합니다. 이는 셰이더 코드를 작성하고 설정하는 방법에 많은 영향을 미칩니다.

리소스 관리

렌더 그래프 시스템은 전체 프레임을 개괄적으로 표현하여 각 리소스의 수명을 계산합니다. 즉, RenderGraph API를 통해 리소스를 생성하면 렌더 그래프 시스템은 해당 시점에 리소스를 생성하지 않습니다. 대신 API는 해당 리소스를 나타내는 핸들을 반환하며, 사용자는 이 핸들을 모든 RenderGraph API에 사용합니다. 렌더 그래프는 해당 리소스를 작성해야 하는 첫 번째 패스 직전에 리소스를 생성합니다. 여기에서 '생성'이 반드시 렌더 그래프 시스템이 리소스를 할당한다는 것을 의미하지는 않습니다. 렌더 패스에서 리소스를 사용할 수 있도록 리소스를 표현하는 데 필요한 메모리를 제공한다는 의미입니다. 같은 방식으로, 렌더 그래프는 해당 리소스 메모리를 읽어야 하는 마지막 패스 이후에 리소스 메모리를 해제합니다. 이렇게 하면 패스에서 선언한 리소스에 따라 렌더 그래프 시스템이 가장 효율적인 방법으로 메모리를 재사용할 수 있습니다. 렌더 그래프 시스템이 특정 리소스가 필요한 패스를 실행하지 않으면, 시스템은 해당 리소스를 위한 메모리를 할당하지 않습니다.

렌더 그래프 실행 개요

렌더 그래프 실행은 렌더 그래프 시스템이 프레임마다 처음부터 다시 완료하는 3단계 프로세스입니다. 예를 들어 사용자의 행동에 따라 그래프가 프레임마다 동적으로 변할 수 있기 때문입니다.

— **설정:** 첫 번째 단계에서는 모든 렌더 패스를 설정합니다. 실행할 모든 렌더 패스와 각 렌더 패스가 사용하는 리소스를 여기에서 선언합니다.

— **컴파일:** 두 번째 단계에서는 그래프를 컴파일합니다. 이 단계에서 특정 렌더 패스의 출력을 다른 렌더 패스에서 사용하지 않는다면 렌더 그래프 시스템이 해당 렌더 패스를 컬링합니다. 이렇게 하면 그래프를 설정할 때 특정 로직을 줄일 수 있으므로 설정을 덜 체계적으로 구성해도 됩니다. 이 단계에서 리소스의 수명도 계산됩니다. 이를 통해 렌더 그래프 시스템이 리소스를 효율적으로 생성하고 해제할 수 있으며, 비동기 계산 파이프라인에서 패스를 실행할 때 적절한 동기화 지점을 계산할 수 있습니다.

— **실행:** 마지막으로 그래프를 실행합니다. 렌더 그래프 시스템은 컬링되지 않은 모든 렌더 패스를 선언된 순서대로 실행합니다. 각 렌더 패스 전에 렌더 그래프 시스템은 적절한 리소스를 생성하고, 다음 렌더 패스에서 리소스가 사용되지 않으면 기존 렌더 패스 이후에 리소스를 해제합니다.

한 예시를 살펴보겠습니다. 이 예시의 최종 코드는 여기에서 확인할 수 있습니다.

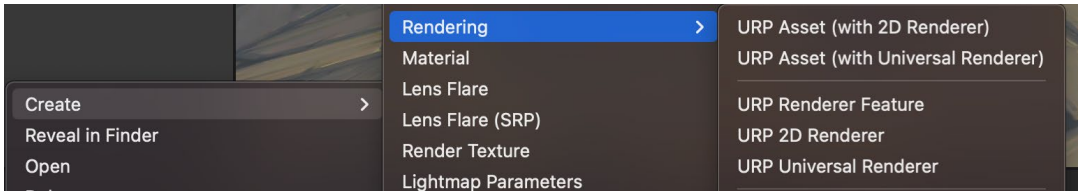
렌더 그래프 시스템을 사용하여 렌더 파이프라인에 삽입되는 렌더러 기능을 생성하는 방법을 알아보세요. 이 튜토리얼에서는 최신 방법을 사용해 대역폭을 줄여 모바일 플랫폼에 적합한 고성능 포스트 프로세싱 패스를 생성합니다.

튜토리얼 보기

렌더러 기능

렌더러 기능은 URP의 어떤 단계에서든 사용하여 최종 렌더에 영향을 줄 수 있습니다. 이 예시는 머티리얼로 이미지의 각 픽셀을 처리하여 틴트 효과를 생성하는 포스트 프로세싱 기법입니다.

1. 먼저 프로젝트의 Assets 폴더에서 적합한 폴더를 찾습니다. 오른쪽 클릭하고 **Create > Rendering > URP Renderer Feature**를 선택합니다. 이름을 **TintFeature**로 설정합니다.



2. 기본 **TintFeature** 파일을 더블 클릭합니다. 렌더러 기능을 위한 상용구가 포함된 C# 스크립트가 나타납니다.

```
using UnityEngine;
using UnityEngine.Rendering;
using UnityEngine.Rendering.Universal;
using UnityEngine.Rendering.RenderGraphModule;

0 references
public class TintRendererFeature : ScriptableRendererFeature
{
    2 references
    class CustomRenderPass : ScriptableRenderPass
    {
        // This class stores the data needed by the RenderGraph pass.
        // It is passed as a parameter to the delegate function that executes the RenderGraph pass.
        3 references
        private class PassData
        {
        }

        // This static method is passed as the RenderFunc delegate to the RenderGraph render pass.
        // It is used to execute draw commands.
        1 reference
        static void ExecutePass(PassData data, RasterGraphContext context)
        {
        }

        // RecordRenderGraph is where the RenderGraph handle can be accessed, through which render passes can be added to th
        // FrameData is a context container through which URP resources can be accessed and managed.
        0 references
        public override void RecordRenderGraph(RenderGraph renderGraph, ContextContainer frameData)
        {
            const string passName = "Custom Render Pass";

            // This adds a raster render pass to the graph, specifying the name and the data type that will be passed to the
            using (var builder = renderGraph.AddRasterRenderPass<PassData>(passName, out var passData))
            {
                // Use this scope to set the required inputs and outputs of the pass and to
                // setup the passData with the required properties needed at pass execution time.

                // Make use of frameData to access resources and camera data through the dedicated containers.
                // Eg:
                // UniversalCameraData cameraData = frameData.Get<UniversalCameraData>();
                UniversalResourceData resourceData = frameData.Get<UniversalResourceData>();
```

렌더러 기능의 기본 코드

3. TintRendererFeature에 다음 프로퍼티를 추가합니다.

   — **RenderPassEvent**를 통해 사용자가 인스펙터에서 삽입 지점을 설정할 수 있습니다.

   — **Material**은 복사에 사용됩니다.

   — `requirements`는 **Color**로 설정합니다.

   — ProfilingSampler를 초기화합니다. `_BlitTexture`와 `_BlitScaleBias`에 액세스할 수 있는 두 ID 값을 얻습니다.

   — 마지막으로 **MaterialPropertyBlock**을 정의합니다.

```
public RenderPassEvent injectionPoint = RenderPassEvent.AfterRendering;
public Material passMaterial;
public ScriptableRenderPassInput requirements =
                              ScriptableRenderPassInput.Color;
private ProfilingSampler m_Sampler;
private static readonly int m_BlitScaleBiasID =
                              Shader.PropertyToID("_BlitScaleBias");
private static MaterialPropertyBlock s_SharedPropertyBlock = null;
```

4. **CustomRenderPass**의 이름을 **TintPass**로 변경하고 프로퍼티를 **TintPass** 클래스에 추가합니다. 렌더링된 이미지의 현재 상태에 적용되는 셰이더가 Material에 포함됩니다. PassData는 패스를 선언할 때 사용되는 데이터를 정의합니다. 여기에 렌더링 코드가 액세스할 수 있는 데이터를 설정합니다.

```
private Material m_Material;
private string m_PassName;
private ProfilingSampler m_Sampler;
private class PassData
{
    internal Material material;
    internal TextureHandle source;
}
```

5. TintPass에 생성자를 추가하여 머티리얼을 초기화하고, 렌더 파이프라인에서 이 패스의 위치를 지정합니다.

```
public TintPass(Material mat, string name)
       {
           m_PassName = name;
           m_Material = mat;
           m_Sampler ??= new ProfilingSampler(GetType().Name + "_" + name);
        }
```

6. `ExecutePass`, `OnCameraSetup`, `Execute`, `OnCameraCleanup` 함수를 삭제합니다.

7. RecordRenderGraph 함수에 아래 코드를 추가합니다.

   — resourceData 인스턴스를 가져옵니다. 포스트 프로세싱 이후 활성 색상 텍스처에서 텍스처 기술자(descriptor)를 가져오는 데 사용합니다.

   — 이름을 변경하고 새 텍스처를 요청합니다. 필요할 때 렌더 그래프가 할당할 것입니다.

   — 첫 번째 Blit은 현재 색상 버퍼를 다음 패스의 입력으로 사용할 수 있도록 중간 텍스처에 복사합니다. AddRasterRenderPass를 사용하여 패스를 생성합니다. 첫 번째 패스에서는 passData 인스턴스의 source를 resourceData 인스턴스의 activeColorTexture로 설정합니다.

   — UseTexture 메서드로 builder 입력 텍스처를 설정하고 SetRenderAttachment로 출력을 설정합니다.

   — 마지막으로 builder가 사용할 렌더 함수를 설정합니다. 아래 단계에서 생성할 ExecuteCopyColorPass 함수를 사용합니다.

   — 두 번째 Blit은 복사할 때 머티리얼을 사용합니다. 첫 번째 Blit과 유사하지만 passData 인스턴스에 머티리얼을 할당하고 SetRenderFunc 할당에 ExecuteMainPass 함수를 사용합니다.

```
public override void RecordRenderGraph(RenderGraph renderGraph, ContextContainer frameData)
{
  UniversalResourceData resourceData = frameData.Get<UniversalResourceData>();
  var colCopyDesc =
          renderGraph.GetTextureDesc(resourceData.afterPostProcessColor);
  colCopyDesc.name = "_TempColorCopy";
  TextureHandle copiedColorTexture = renderGraph.CreateTexture(colCopyDesc);
  using (var builder = renderGraph.AddRasterRenderPass<PassData>(m_PassName +
                "_CopyPass", out var passData, m_Sampler))
  {
      passData.source = resourceData.activeColorTexture;
      builder.UseTexture(resourceData.activeColorTexture, AccessFlags.Read);
      builder.SetRenderAttachment(copiedColorTexture, 0, AccessFlags.Write);
      builder.SetRenderFunc(
         (PassData data, RasterGraphContext rgContext) =>
         {
             ExecuteCopyColorPass(rgContext.cmd, data.source);
         });
  }
  using (var builder = renderGraph.AddRasterRenderPass<PassData>(m_PassName +
                "_FullScreenPass", out var passData, m_Sampler))
  {
      passData.source = resourceData.activeColorTexture;
```

```
        passData.material = m_Material;
        builder.UseTexture(copiedColorTexture, AccessFlags.Read);
        builder.SetRenderAttachment(resourceData.activeColorTexture, 0,
                    AccessFlags.Write);
        builder.SetRenderFunc(
           (PassData data, RasterGraphContext rgContext) =>
           {
               ExecuteMainPass(rgContext.cmd, data.material, data.source);
           });
    }
}
```

8. ExecuteCopyColorPass를 제공해야 합니다. Blitter 메서드 BlitTexture를 사용하는 함수입니다. 이 함수는 다양한 파라미터 버전이 있습니다. 이 예시에서는 이 버전을 사용합니다.

```
public static void BlitTexture(CommandBuffer cmd,
RTHandle source, Vector4 scaleBias, float mipLevel, bool
bilinear)
```

이 함수는 RecordRenderGraph 함수 전에 배치해야 합니다.

```
private static void ExecuteCopyColorPass(RasterCommandBuffer cmd, RTHandle sourceTexture)
    {
      Blitter.BlitTexture(cmd, sourceTexture, new Vector4(1, 1, 0, 0), 0.0f, false);
    }
```

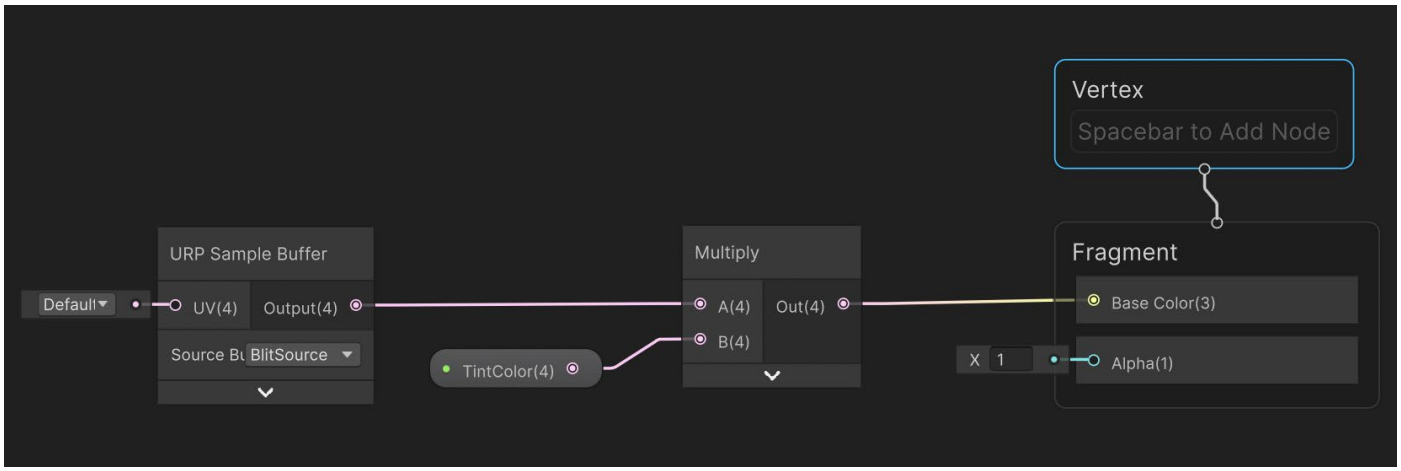
9. 이제 ExecuteMainPass 함수를 정의합니다. 핵심 Blit.hlsl에 응답하는 셰이더가 있는 사용자 머티리얼이 동작하려면 _BlitScaleBias를 균등하게 설정해야 합니다.

```
private static void ExecuteMainPass(RasterCommandBuffer cmd, Material material,
     RTHandle copiedColor)
{
    s_SharedPropertyBlock.Clear();
     s_SharedPropertyBlock.SetVector(m_BlitScaleBiasID, new Vector4(1, 1, 0, 0));
     cmd.DrawProcedural(Matrix4x4.identity, material, 0, MeshTopology.Triangles,
                        3, 1, s_SharedPropertyBlock);
}
```

10. `CustomRenderPass`를 TintPass로, m_ScriptablePass를 `m_pass`로 이름을 각각 변경합니다.
11. Create 메서드 내 기존 코드를 삭제합니다. 커스텀 생성자를 사용하여 새로운 TintPass를 생성합니다. renderPassEvent를 정의하고 입력을 설정합니다.

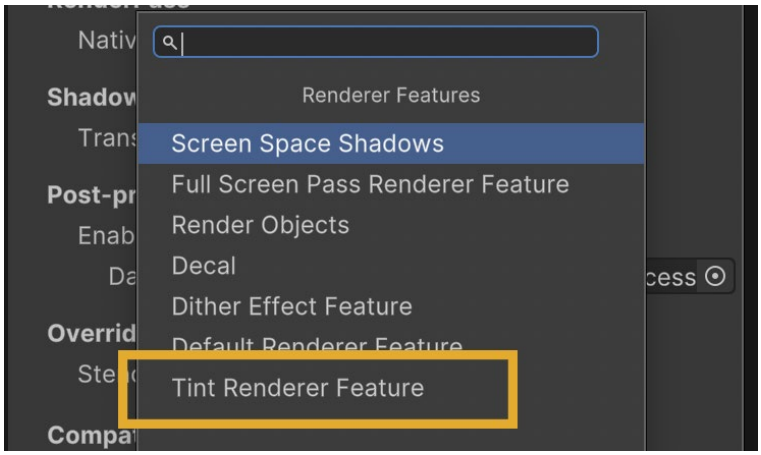
```
public override void Create()
    {
        m_pass = new TintPass(passMaterial, name);
        m_pass.renderPassEvent = injectionPoint;
        m_pass.ConfigureInput(requirements);
    }
```

다음 일련의 단계를 통해 Shader Graph 셰이더와 작동하도록 설계된 이 렌더러 기능 예시를 마무리합니다. **TintColor** 프로퍼티, BlitSource를 사용하는 **URP Sample Buffer** 노드, TintColor로 기존 BlitSource를 조절하는 **Multiply** 노드로 구성된 간단한 예시입니다. 아래에서 단계를 살펴보겠습니다.

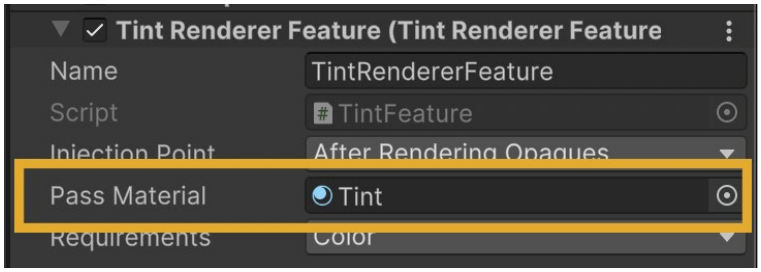


Tint Shader Graph

1. 효과가 구현되는 모습을 확인하려면 **Renderer Data** 오브젝트를 선택하고 **Add Renderer Feature**를 클릭합니다. 목록에 TintFeature가 표시됩니다.



2. Tint Shader Graph를 사용하여 머티리얼을 생성합니다.
3. 렌더러 기능에 Tint 머티리얼을 할당하고 머티리얼 색상을 설정합니다.



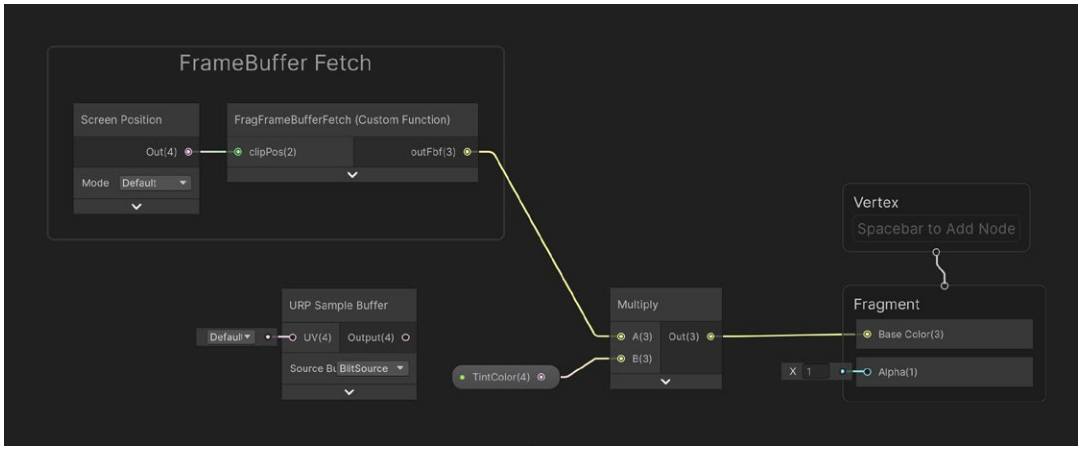
TintFeature 효과: 왼쪽은 아무런 처리 없음, 오른쪽은 틴트 처리

Unity 6에서는 Render Graph Viewer 창을 제공하며, **Window > Analysis > Render Graph Viewer**를 통해 열 수 있습니다. 이 뷰어를 통해 파이프라인에서 어떤 작업이 수행되는지 파악할 수 있습니다.

Render Graph Viewer 창

Draw Objects Pass, **TintRendererFeature_CopyPass**,, **TintRendererFeature_FullScreen Pass**는 모두 별개의 패스입니다. 단일 패스로 결합한다면 좋을 것입니다. 하지만 텍스처가 사용되는 방식에 따라 별개의 패스가 필요한 것입니다. Pass List를 펼치고 패스 이름을 클릭하면 해당 패스의 세부 정보를 확인할 수 있습니다. Draw Objects Pass를 클릭하면 'Failed to merge' 메시지가 표시됩니다. 이 패스에서 출력한 데이터를 다음 패스에서 일반 텍스처로 읽기 때문입니다. `TintRendererFeature_CopyPass`에서도 동일한 문제가 발생합니다. 이 문제를 해결해 보겠습니다.

새 머티리얼이 필요합니다. 프로젝트의 Resources 폴더에 **FrameBufferFetch** 머티리얼이 있습니다. 두 개의 패스를 지닌 같은 이름(FrameBufferFetch)의 셰이더를 사용합니다. 현재 활성 프레임 버퍼를 샘플링하는 두 번째 패스를 활용해야 합니다. 이 접근 방식을 바탕으로 Tint Shader Graph를 수정해야 합니다. URP Sample Buffer 노드를 사용하는 대신 커스텀 함수 노드를 사용합니다. **FrameBufferFetch.hlsl** 파일의 HLSL을 사용합니다. 사실상 `LOAD_FRAMEBUFFER_X_INPUT` 매크로만 사용합니다.



FrameBufferFetch 커스텀 노드를 사용하는 Tint Shader Graph

TintFeature로 돌아가겠습니다. TintPass에 새 private static 프로퍼티를 추가해야 합니다.

```
private static Material s_FrameBufferFetchMaterial;
```

Load 메서드를 사용하여 커스텀 생성자에서 할당할 수 있습니다.

```
s_FrameBufferFetchMaterial ??= UnityEngine.Resources.Load("FrameBufferFetch") as Material;
```

ExecuteCopyColorPass에서 Blit을 삭제하고 대신 DrawProcedural을 사용해야 합니다. identity 행렬을 다시 사용하며, 두 번째 패스에 s_FrameBufferFetchMaterial 머티리얼을 사용합니다. 첫 번째 패스의 인덱스는 0, 두 번째 패스의 인덱스는 1입니다. 토폴로지 삼각형을 다시 사용하므로 indexCount는 3, instanceCount는 1을 사용합니다.

```
cmd.DrawProcedural(Matrix4x4.identity, s_FrameBufferFetchMaterial, 1, MeshTopology.Triangles, 3, 1, null);
```

이제 RecordRenderGraph 메서드를 사용합니다. 첫 번째 패스의 UseTexture를 SetInputAttachment로 대체합니다. 이 메서드는 FrameBufferFetch를 사용하여 이전 패스에 액세스합니다.

```
builder.SetInputAttachment(resourceData.activeColorTexture, 0 )
```

두 번째 패스도 동일한 단계로 처리합니다.

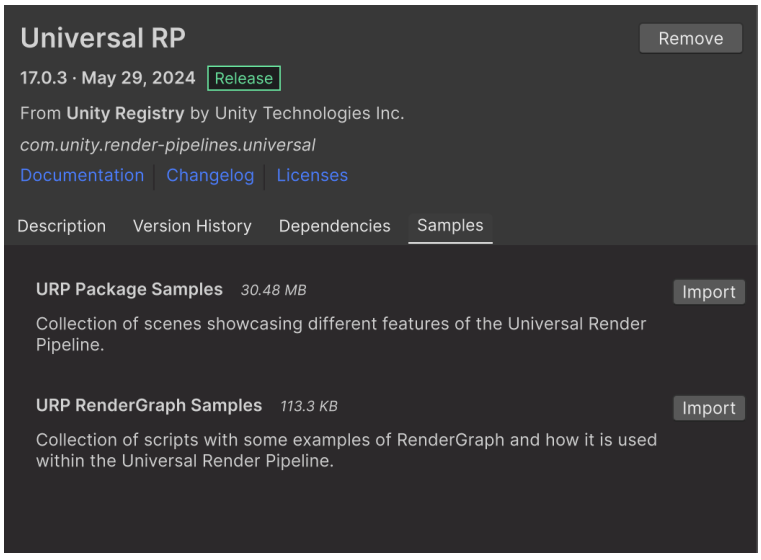
```
builder.SetInputAttachment( copiedColorTexture, 0 )
```

Render Graph Viewer를 확인하고 필요하다면 새로 고침 합니다. Draw Objects Pass와 두 TintRendererFeature 패스가 단일 패스로 결합되어 성능이 향상되는 것을 볼 수 있습니다. Viewer 창의 **F**는 FrameBufferFetch를 통해 입력에 액세스한다는 것을 의미합니다.

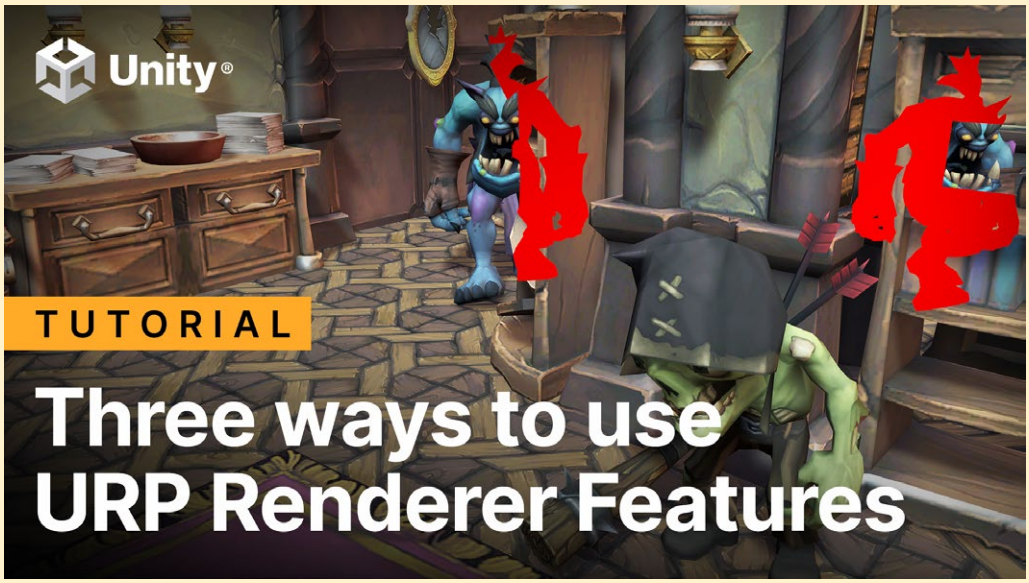
병합된 3개의 패스

FramebufferFetch는 Vulkan, Metal, DirectX 12를 타게팅하는 모바일 플랫폼에서 지원됩니다. 다른 플랫폼에서는 엔진이 텍스처 샘플링을 활용합니다. FrameBufferFetch를 사용하면 대역폭 사용량이 줄어들어 대역폭 바운드인 경우 성능이 향상될 수 있으며, 일반적으로 배터리 사용량이 줄어듭니다.

렌더 그래프 시스템을 사용하는 예시를 더 찾아보려면 패키지 관리자에서 패키지 샘플을 다운로드하세요. Universal RP를 검색하고 Samples 탭을 클릭합니다. 여기에서 URP RenderGraph Samples를 임포트할 수 있습니다.



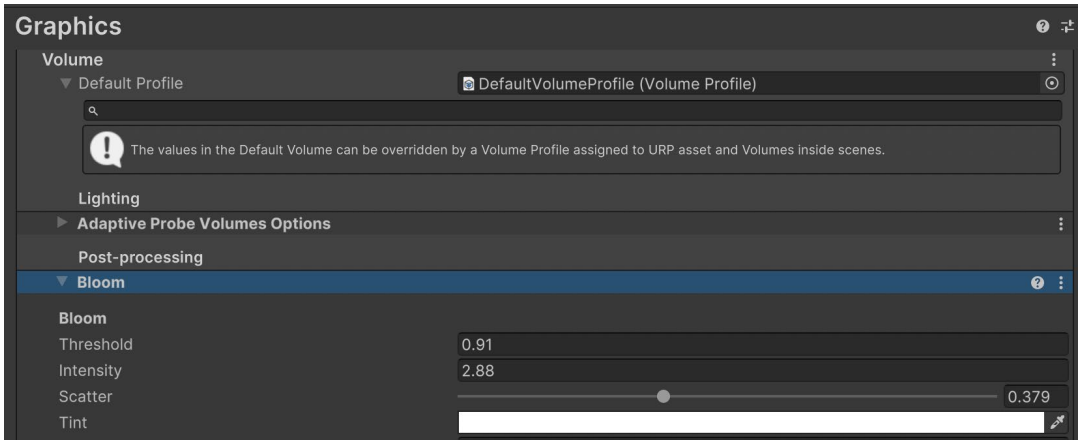
패키지 관리자의 Universal RP 패키지에서 URP RenderGraph Samples 임포트



이 동영상 튜토리얼에서는 렌더러 기능을 사용하는 세 가지 실습을 살펴봅니다. 커스텀 포스트 프로세싱 효과 및 스텐실 효과 제작 방법과 환경으로 캐릭터를 가리는 방법을 알아봅니다.

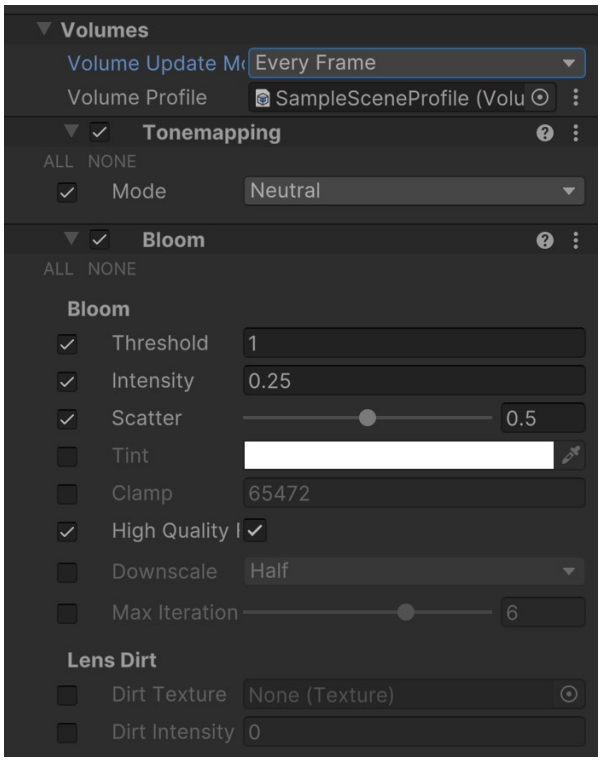
포스트 프로세싱

URP는 포스트 프로세싱 효과를 추가할 때 볼륨 프레임워크를 사용합니다. Unity 6에서는 Default Volume을 제공합니다. **Project Settings > Graphics > Volume에서 사용할 수 있습니다.** 여기에서 적용되는 설정은 전체 프로젝트에 영향을 미치지만 씬에 추가되는 볼륨으로 오버라이드할 수 있습니다.



Default Volume 옵션

Unity 6의 또 다른 새로운 기능은 활성 URP 에셋을 위한 볼륨이며 이 기능도 씬에 추가되는 볼륨으로 오버라이드할 수 있습니다.



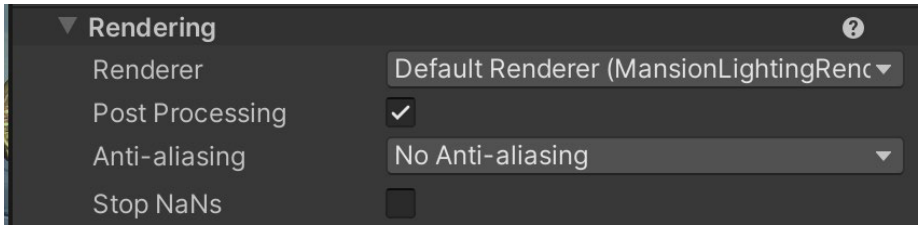
URP Asset > Volumes

씬에 볼륨을 추가할 때 볼륨에 적용할 포스트 프로세싱 효과를 선택할 수 있습니다. 볼륨 모드에는 글로벌과 로컬이 있으며, 글로벌로 설정한 경우 볼륨은 씬의 모든 곳에서 카메라에 영향을 미칩니다. 반면 로컬 모드를 설정하면 카메라가 콜라이더 경계 내에 있을 때만 볼륨의 영향을 받습니다.

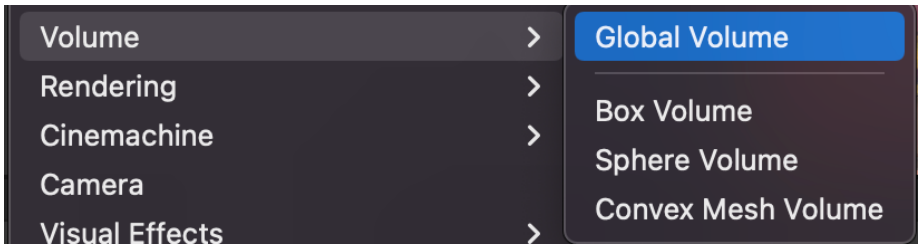
포스트 프로세싱 효과 적용: 왼쪽 상단 이미지에는 효과가 적용되어 있지 않으며 오른쪽 상단 이미지에는 블룸, 왼쪽 하단에는 비네트(Vignette), 오른쪽 하단에는 색상 조정이 추가되어 있음

URP 포스트 프로세싱 프레임워크 사용

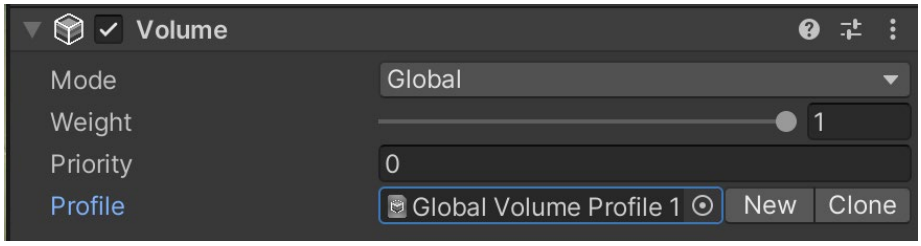
1. 첫 단계는 메인 카메라에서 포스트 프로세싱을 활성화하는 것입니다. **계층** 창에서 **Main Camera**를 선택하고 **인스펙터**로 이동하여 **Rendering** 패널을 확장합니다. **Post Processing** 옵션을 선택합니다.



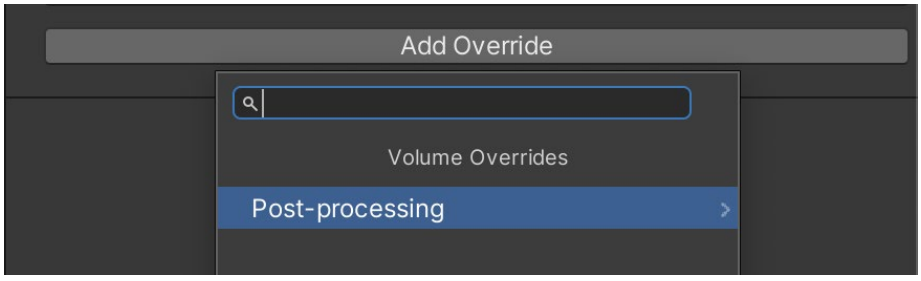
2. **계층** 창에서 오른쪽 클릭하고 **Create > Volume > Global Volume**을 선택하여 Global Volume을 만듭니다.

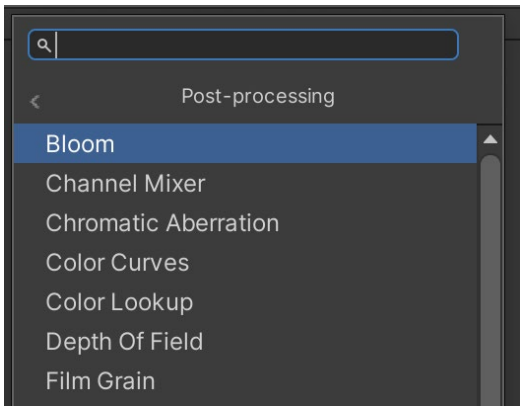


3. 계층 창에서 Global Volume이 선택된 상태에서 인스펙터의 **Volume** 패널을 찾아 **New**를 클릭하여 새 **Profile**을 만듭니다.



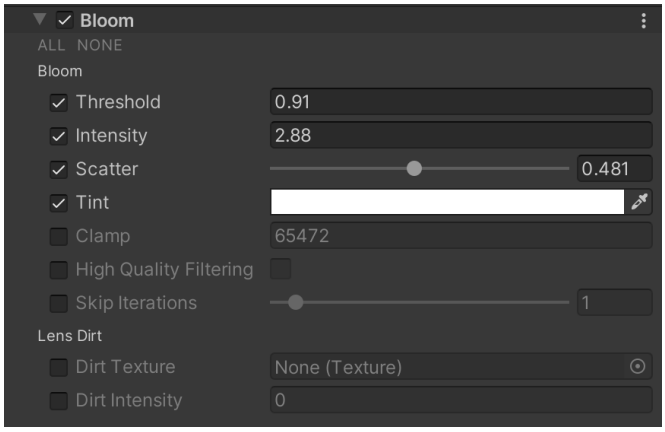
4. 포스트 프로세싱 효과를 추가합니다. 사용 가능한 효과가 나열된 표를 아래쪽에서 확인할 수 있습니다. **Add Override**를 클릭하고 **Post-processing**을 선택합니다. 이 예시에서는 Bloom 효과를 선택했습니다.



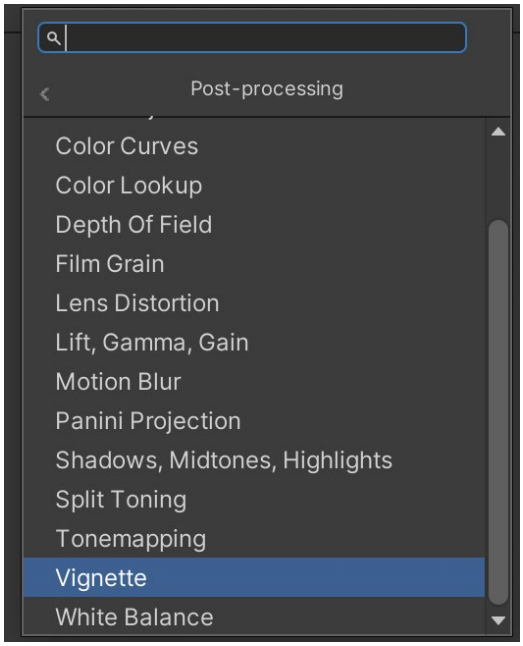


Bloom 효과 선택

5. 각 효과에는 전용 Settings 패널이 있으며, 다음 이미지에는 블룸에 대한 설정이 나와 있습니다.



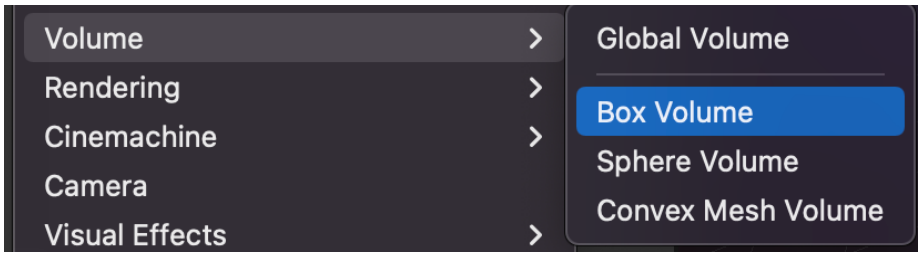
6. 이 예시의 비네트처럼 여러 효과를 쉽게 추가하고 해당 Settings 패널을 사용하여 각 항목을 설정할 수 있습니다.



로컬 볼륨 컴포넌트 추가

볼륨 프레임워크를 사용하면 카메라가 이동함에 따라 다양한 포스트 프로세싱 프로파일이 트리거되도록 씬을 설정할 수 있습니다. 이를 위해서는 Local Volume 컴포넌트를 추가하면 됩니다. 어떻게 설정하는지 단계별로 살펴보겠습니다.

1. **계층** 창에서 오른쪽 클릭하여 **Create > Volume > Box Volume**을 선택합니다. 구체가 목적에 더 적합한 경우에는 **Sphere Volume**을 선택하고, 볼륨 영역을 정의하는 콜라이더 컴포넌트 셰이프를 더 정밀하게 제어하려면 **Convex Mesh Volume**을 선택합니다.

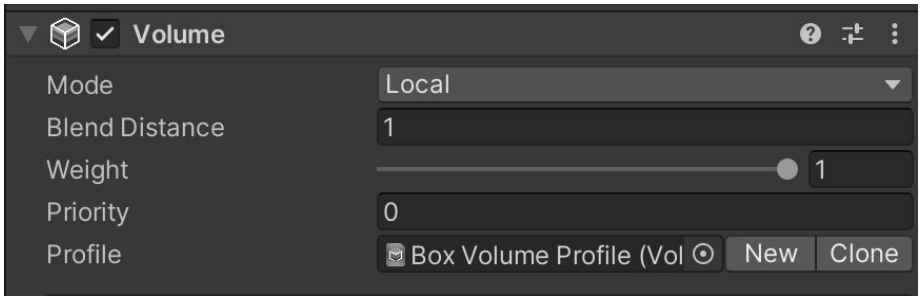


2. **인스펙터**의 **Volume** 패널에서 이 볼륨 데이터를 저장할 새 **Profile**을 선택합니다. 이 패널을 사용하여 다음을 설정할 수도 있습니다.

   a. **Blend Distance:** URP가 블렌딩을 시작하는 볼륨 콜라이더로부터 가장 먼 거리이며, 이 프로파일이 페이드인되는 콜라이더 범위의 거리입니다. 포스트 프로세싱 효과는 콜라이더 에지에서 페이드아웃되며 콜라이더 에지로부터의 Blend Distance는 완전히 페이드인됩니다.

   b. **Weight:** 포스트 프로세싱 효과의 최대 강도를 정의합니다. Weight를 1로 설정하면 효과의 강도는 최대가 됩니다. 0으로 설정하면 효과가 적용되지 않으며, 0.5로 설정하면 이펙트 강도가 최대 50%가 됩니다.

   c. **Priority:** 이 값을 사용하면 씬에서 여러 볼륨에 동일한 양의 영향이 있을 때 사용할 볼륨 URP를 결정할 수 있습니다. 숫자가 높을수록 우선순위가 높아집니다. 글로벌과 로컬을 병합하는 경우 Global을 기본 설정값인 0으로, Local Volume을 1 이상으로 유지합니다.



Local Volume 설정

3. 아래 이미지에 나와 있는 대로 **Box Collider** 컴포넌트를 사용하여 **볼륨** 위치를 지정하고 범위를 제어합니다.

연결된 Box Collider 컴포넌트를 사용하여 박스 볼륨의 위치를 지정하고 크기를 조절

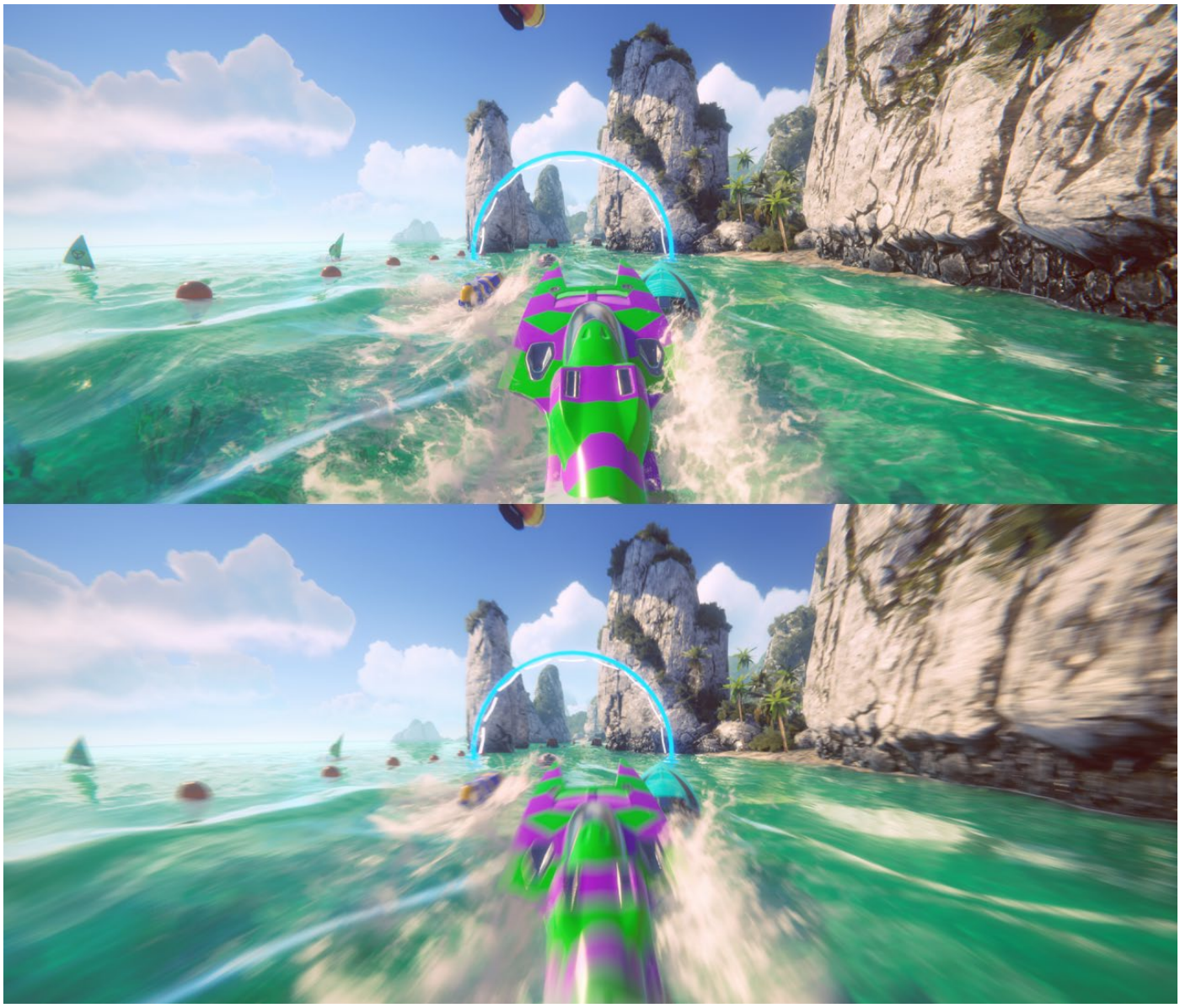
포스트 프로세싱은 프로세서에 큰 부하를 줄 수 있으므로 저사양 하드웨어 및 모바일 기기에서는 효과 구현을 신중히 고려하세요. 프로젝트에서 반드시 사용해야 한다면 타겟 하드웨어에서 테스트하시기 바랍니다. 특정 필터는 다른 필터에 비해 프로세서 소모가 덜합니다. 이 문서에서 모바일에 적합한 효과를 간략하게 살펴볼 수 있습니다.

다음은 URP에서 사용 가능한 포스트 프로세싱 효과입니다.

효과	설명
블룸	정의된 밝기 수준 이상으로 픽셀 주위에 글로우를 추가합니다.
채널 믹서	전체 믹스에서 각 입력 색상 채널의 영향도를 수정합니다.
색 수차	이미지의 어두운 부분과 밝은 부분을 분리하는 경계를 따라 색상 띠를 만듭니다.

색상 조정	최종 렌더링된 이미지의 전반적인 톤, 밝기, 콘트라스트를 조정합니다.
색상 커브	색조, 채도 또는 광도의 특정 범위를 조정하는 고급 방식입니다.
색상 룩업	룩업 텍스처를 사용하여 각 픽셀의 색상을 새 값으로 매핑합니다.
뎁스오브필드 (피사계심도)	카메라 렌즈의 포커스 프로퍼티를 시뮬레이션합니다.
필름 그레인	사진 필름의 무작위 광학 텍스처를 시뮬레이션합니다.
렌즈 왜곡	최종 렌더링 이미지를 왜곡하여 실제 카메라 렌즈의 모양을 시뮬레이션합니다.
리프트 감마 게인	여러 가지 트랙볼을 사용하여 이미지 내 다양한 범위에 영향을 줄 수 있습니다. 트랙볼 아래의 슬라이더를 조정하면 해당 범위의 색상 밝기를 오프셋할 수 있습니다.
모션 블러	실제 카메라가 카메라의 노출 시간보다 빠르게 움직이는 오브젝트를 촬영할 때 이미지에서 발생하는 블러 현상을 시뮬레이션합니다.
파니니 투영	시야각(FOV)이 매우 넓은 씬에서 원근 뷰를 렌더링하는 데 도움이 됩니다.
스크린 공간 렌즈 플레어	단일 광원이 아닌 전체 씬에 적용되는 렌즈 플레어를 추가합니다.
그림자 미드톤 하이라이트	렌더의 그림자, 미드톤 및 하이라이트를 별도로 제어합니다.
분할 토닝	다양한 색조를 씬의 그림자와 하이라이트에 추가할 수 있습니다.
톤 매핑	이미지의 HDR 값을 새로운 범위의 값으로 다시 매핑합니다.
비네트	이미지가 중심과 비교하여 에지를 향해 갈수록 어두워집니다.
화이트 밸런스	현실에서 흰색으로 나타나는 항목을 최종 이미지에서 흰색으로 렌더링할 수 있도록 비현실적인 색조를 제거합니다.

모션 블러



상단 이미지: 모션 블러 비활성화, 하단 이미지: 모션 블러 활성화

모션 블러 포스트 프로세싱 효과는 실제 카메라가 카메라의 노출 시간보다 빠르게 움직이는 오브젝트를 촬영할 때 이미지에서 발생하는 블러 현상을 시뮬레이션합니다. 주로 오브젝트가 빠르게 움직이거나 노출 시간이 길 때 발생합니다.

모션 블러 사용

URP를 사용하는 다른 모든 포스트 프로세싱 효과와 마찬가지로 모션 블러도 볼륨 시스템을 사용하므로, 모션 블러 프로퍼티를 활성화하고 수정하려면 씬 내 볼륨에 모션 블러 오버라이드를 추가해야 합니다.

프로퍼티	설명
Mode	모션 블러 기법을 선택합니다. 옵션: — **Camera Only**: 카메라의 움직임만 활용하여 오브젝트에 블러를 적용합니다. 이 기법은 모션 벡터를 사용하지 않으며 Camera and Objects 기법보다 성능이 좋습니다. — **Camera and Objects**: 카메라와 게임 오브젝트의 움직임을 모두 활용합니다. 게임 오브젝트 모션 벡터가 카메라 모션 벡터를 덮어씁니다.
Quality	효과의 품질을 설정합니다. 품질이 낮은 프리셋일수록 성능은 좋지만 시각적 품질은 저하됩니다.
Intensity	모션 블러 필터의 강도를 0에서 1까지의 값으로 설정합니다. 값이 클수록 블러 효과가 강해지지만 Clamp 파라미터에 따라 성능이 저하될 수 있습니다.
Clamp	카메라 회전으로 인한 속도가 유지될 수 있는 최대 길이를 설정합니다. 이 설정은 빠른 속도의 블러를 제한하여 성능 비용이 너무 늘어나는 것을 방지합니다. 값은 화면의 최대 해상도 대비 비율로 측정됩니다. 값의 범위는 0에서 0.2까지입니다. 기본값은 0.05입니다.

성능 문제 해결

모션 블러로 인한 성능 저하를 줄일 수 있는 방법은 다음과 같습니다.

— **Quality 낮추기**: 품질 설정을 낮추면 성능이 향상되지만 시각적 결함이 생길 수 있습니다.

— **Mode 프로퍼티를 Camera and Objects에서 Camera Only로 변경:** 이 기법이 Camera and Objects보다 성능이 좋습니다.

— **Clamp 낮추기:** Unity가 고려하는 최대 속도를 낮출 수 있습니다. 값이 낮을수록 성능이 향상됩니다.

코드를 사용하여 포스트 프로세싱 제어

C# 스크립트를 사용하여 포스트 프로세싱 프로파일을 동적으로 조정할 수도 있습니다. 다음 코드 예시는 블룸 효과의 강도를 조정하는 방법을 보여 줍니다. 비네트가 적용된 경우 코드를 통해 색상 비네트 처리를 제어할 수 있습니다. 예를 들면, 플레이어 캐릭터가 대미지를 입은 경우 일시적으로 빨간색을 입힐 수 있습니다.

```
using UnityEngine;
using UnityEngine.Rendering;
using UnityEngine.Rendering.Universal;
public class PPController : MonoBehaviour
{
       // 첫 번째 프레임 업데이트 전에 Start가 호출됨
       void Start()
       {
       Volume volume = GetComponent<Volume>();
       Bloom bloom;
       if (volume.profile.TryGet<Bloom>(out bloom))
       {
              bloom.intensity.value = 0;
       }
   }
}
```

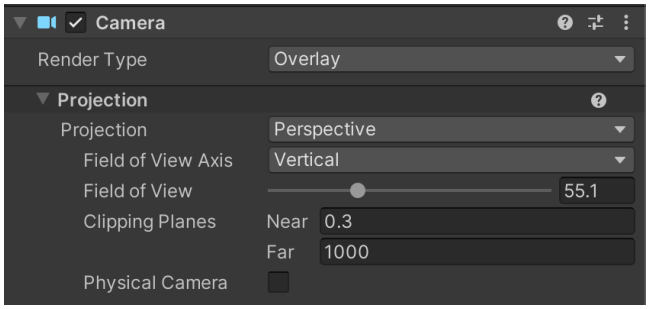
카메라 스태킹

게임에서는 일반적으로 다양한 카메라에서 바라보는 지오메트리를 하나의 렌더에 결합하는 기능이 필요합니다. 아래 이미지의 앞부분에서는 게임 내 인벤토리 역할을 하는 선반을 볼 수 있습니다. 수집한 아이템은 선반에 추가되며 주요 포인트에서 플레이어가 선택할 수 있습니다. 다양한 시야각(FOV)이 있으며 조명과 포스트 프로세싱도 서로 다른 것을 알 수 있습니다. 이는 URP의 카메라 스태킹 기능을 사용하여 설정되었습니다.

카메라 스태킹 사용 예시

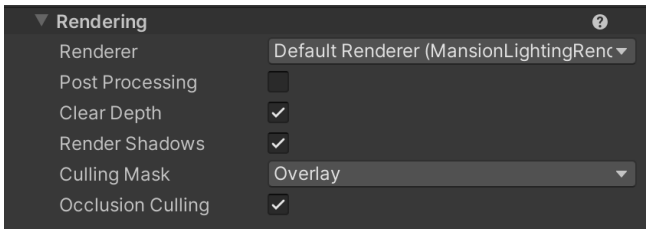
이 기능을 설정하는 방법을 살펴보겠습니다.

1. **계층** 뷰에서 오른쪽 클릭하고 **Camera**를 선택하여 카메라를 만듭니다. 오디오 리스너 컴포넌트를 제거합니다.
2. **Inspector > Camera Settings** 패널을 사용하여 이 카메라를 **Render Type Overlay**로 설정합니다.



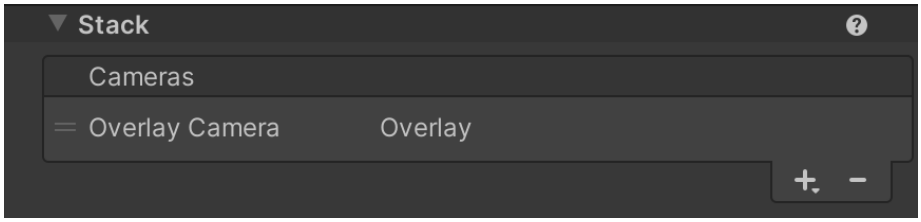
3. 카메라와 렌더링할 게임 오브젝트에 대해 새 **레이어**를 만듭니다.

4. 인스펙터를 사용하여 카메라의 **Rendering > Culling Mask**를 업데이트합니다.



5. 씬에서 적합한 장소로 카메라를 이동한 다음 **레이어 오버레이**에 배치하여 **게임 오브젝트**를 추가하고 위치를 지정합니다.

6. **Rendering > Culling Mask**를 업데이트하여 **메인 카메라**에서 오버레이를 렌더링하지 않도록 합니다.



7. **Stack** 패널에서 '**+**' 버튼을 사용하여 **Overlay Camera**를 추가합니다.

코드를 사용한 스택 제어

포스트 프로세싱에서 하듯이 코드로 스택을 제어하고 런타임에 카메라를 동적으로 추가하거나 제거할 수 있습니다. 다음 코드 예시를 참조하세요.

```
using UnityEngine;
using UnityEngine.Rendering.Universal;
public class StackController : MonoBehaviour
{
      public Camera overlayCamera;
      // 첫 번째 프레임 업데이트 전에 Start가 호출됨
      void Start()
      {
      Camera camera = GetComponent<Camera>();
      var cameraData = camera.GetUniversalAdditionalCamera
      Data();
      cameraData.cameraStack.Remove(overlayCamera);
      }
}
```

포스트 프로세싱과 카메라 스태킹은 둘 다 URP를 사용하여 쉽게 설정할 수 있으며 게임에서 풍부한 대기 효과를 만들 수 있는 강력한 툴입니다.

SubmitRenderRequest API

게임을 사용자의 화면이 아닌 다른 대상에 렌더링하고 싶을 때도 있습니다. **SubmitRenderRequest** API는 이러한 목적으로 설계되었습니다. 활용 사례를 살펴보겠습니다.

스크린샷 기능 코딩

아래 스크립트는 사용자가 화면 내 GUI를 누르면 게임을 화면 밖 **RenderTexture**에 렌더링합니다. 이 스크립트는 메인 카메라에 연결해야 합니다. **RenderTexture**가 **Start** 콜백에서 생성되며, 이는 비트 뎁스가 24인 1920x1080픽셀 텍스처입니다. 사용자가 'Render Request' 버튼을 누르면 RenderRequest 메서드가 호출됩니다.

RenderRequest 메서드에는 Camera 컴포넌트에 대한 레퍼런스가 있습니다. RenderPipeline.StandardRequest 인스턴스를 생성한 다음 현재 사용 중인 파이프라인이 RenderRequest 프레임워크를 지원하는지 확인합니다. 지원하는 경우, Start 콜백에서 초기화한 RenderTexture를 이 요청 오브젝트의 대상으로 설정하고 RenderPipeline.SubmitRenderRequest를 사용하여 렌더링을 초기화합니다. 이 메서드는 카메라 인스턴스와 요청 오브젝트를 사용합니다.

이 시점에 Texture2D에는 현재 씬의 렌더링이 담겨 있습니다. 이를 파일로 저장하려면 먼저 RenderTexture를 Texture2D 인스턴스로 전환해야 합니다. **ToTexture2D** 메서드를 한 가지 방법으로 사용할 수 있습니다. Texture2D로 전환한 다음 Texture2D 인스턴스의 **EncodeToPNG** 메서드를 사용하여 바이트 배열을 가져올 수 있습니다. 그런 다음 System.IO.File의 **WriteAllBytes** 메서드를 사용하여 바이트 배열을 파일로 저장합니다.

스크립트를 바로 사용하면 게임의 **Assets** 폴더 아래에 새로 생성된 **RenderOutput** 폴더에 스크린샷이 저장됩니다. 파일 이름은 R_로 시작하며 0에서 100,000 사이의 무작위 정수가 뒤에 붙습니다.

```
using UnityEngine;
using UnityEngine.Rendering;
[RequireComponent(typeof(Camera))]
public class StandardRenderRequest : MonoBehaviour
{
   [SerializeField]
   RenderTexture texture2D;
   private void Start()
   {
       texture2D = new RenderTexture(1920, 1080, 24);
   }
   // 사용자가 GUI 버튼을 클릭하면
   // 해당 프레임을 렌더링할 수 있도록 다양한 출력 텍스처가 포함된 렌더 요청이 전송됩니다.
   private void OnGUI()
   {
       GUILayout.BeginVertical();
       if (GUILayout.Button("Render Request"))
       {
           RenderRequest();
       }
       GUILayout.EndVertical();
   }
   void RenderRequest()
   {
       Camera cam = GetComponent<Camera>();
       RenderPipeline.StandardRequest request = new RenderPipeline.StandardRequest();
       if (RenderPipeline.SupportsRenderRequest(cam, request))
       {
           // 2D 텍스처
           request.destination = texture2D;
           RenderPipeline.SubmitRenderRequest(cam, request);
           SaveTexture(ToTexture2D(texture2D));
       }
   }
   void SaveTexture(Texture2D texture)
   {
       byte[] bytes = texture.EncodeToPNG();
       var dirPath = Application.dataPath + "/RenderOutput";
       if (!System.IO.Directory.Exists(dirPath))
```

```
        {
            System.IO.Directory.CreateDirectory(dirPath);
        }
        System.IO.File.WriteAllBytes(dirPath + "/R_" + Random.Range(0,
        100000) + ".png", bytes);
        Debug.Log(bytes.Length / 1024 + "Kb was saved as: " + dirPath);
        #if UNITY_EDITOR
                UnityEditor.AssetDatabase.Refresh();
        #endif
    }
    Texture2D ToTexture2D(RenderTexture rTex)
    {
        Texture2D tex = new Texture2D(rTex.width, rTex.height,
        TextureFormat.RGB24, false);
        RenderTexture.active = rTex;
        tex.ReadPixels(new Rect(0, 0, rTex.width, rTex.height), 0, 0);
        tex.Apply();
        Destroy(tex);// 메모리 누수 방지
        return tex;
    }
}
```

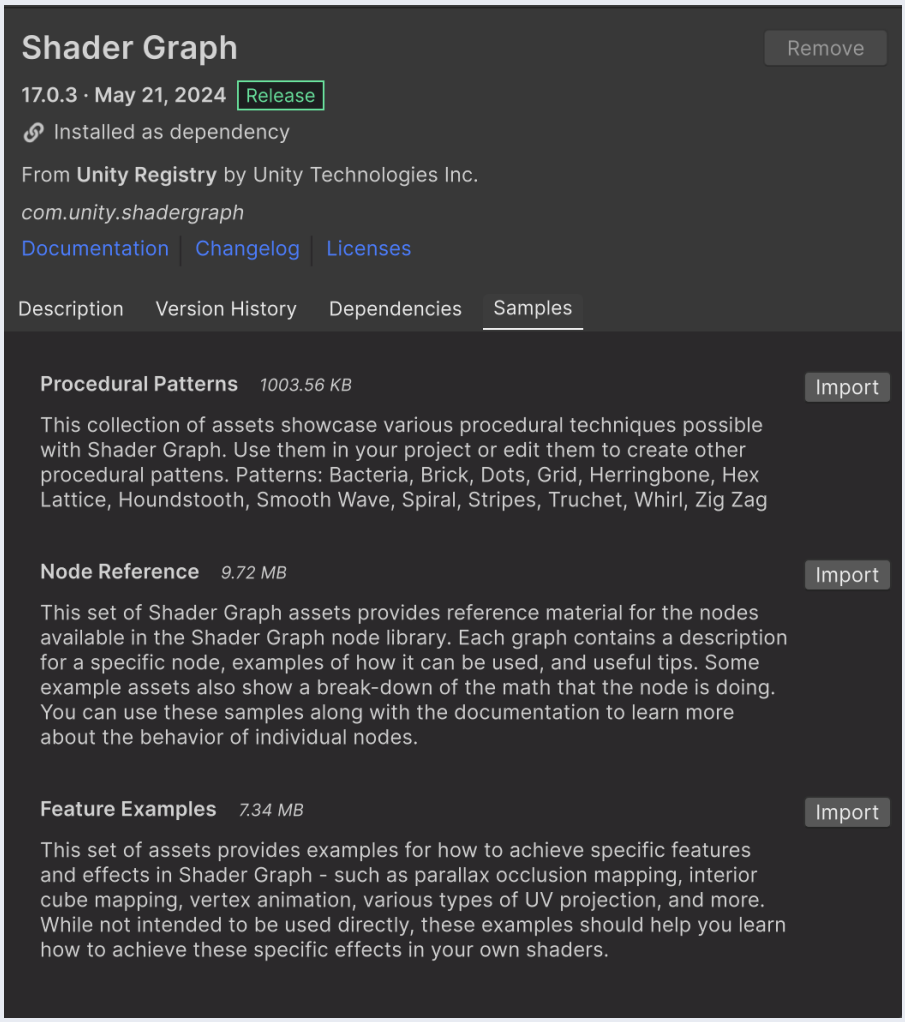
URP와 호환되는 추가 툴

URP는 Unity의 저작 툴과 호환된다는 또 다른 장점 덕분에 테크니컬 아티스트가 복잡한 콘텐츠를 제작하는 데 도움이 됩니다. 이 챕터에서는 Shader Graph와 VFX Graph를 소개합니다.

Shader Graph

Shader Graph를 사용하여 아티스트 워크플로에 커스텀 셰이더를 활용할 수 있습니다. Shader Graph 툴은 URP 템플릿을 사용하여 프로젝트를 시작하거나 URP 패키지를 임포트할 때 포함됩니다.

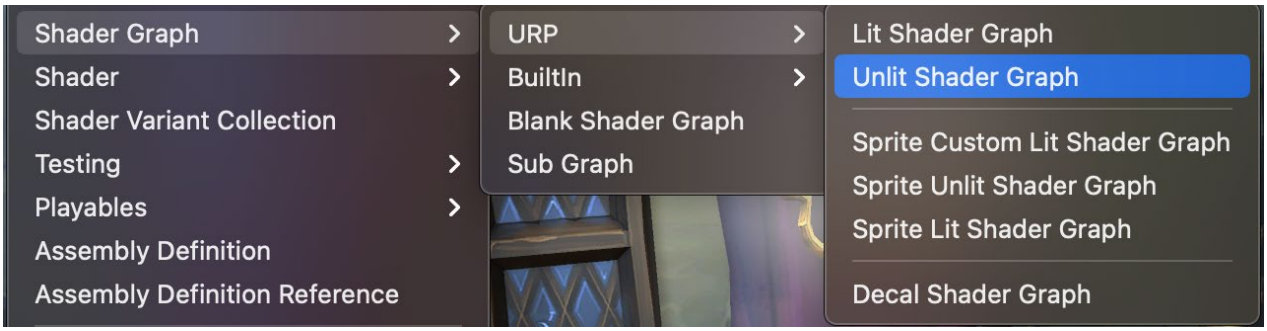
참고:



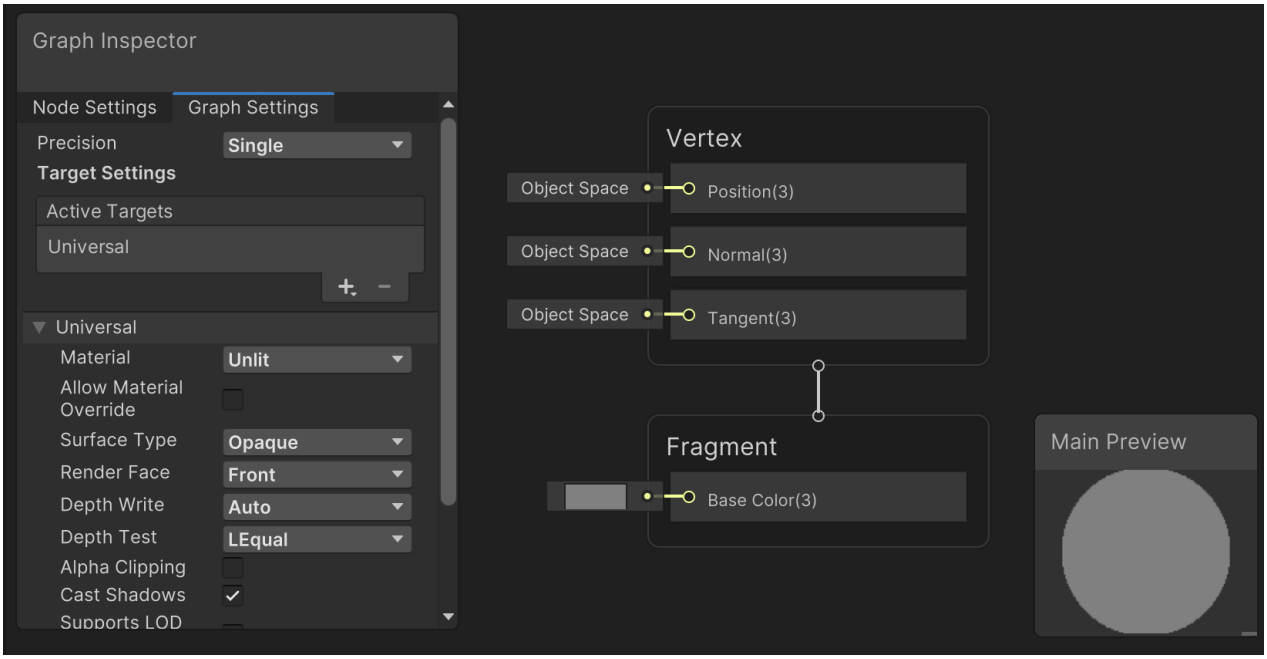
패키지 관리자 창의 Shader Graph 패키지에서 Shader Graph 노드 예시 라이브러리를 확인할 수 있습니다. 이 블로그 게시물에서 정식으로 제작에 사용 가능한 Unity 6의 새로운 Shader Graph 셰이더에 대해 알아보세요.

Shader Graph를 다루려면 별도의 가이드가 필요하지만, 조명 챕터에서 광원 헤일로 셰이더를 만들어 기초적이면서도 매우 중요한 단계를 알아보겠습니다.

1. **프로젝트** 창에서 오른쪽 클릭하고 적합한 폴더를 찾아 **Create > Shader Graph > URP > Unlit Shader Graph**를 선택합니다. 이 예시에서는 Unlit을 선택합니다. 새 에셋의 이름을 FresnelAlpha로 지정합니다.

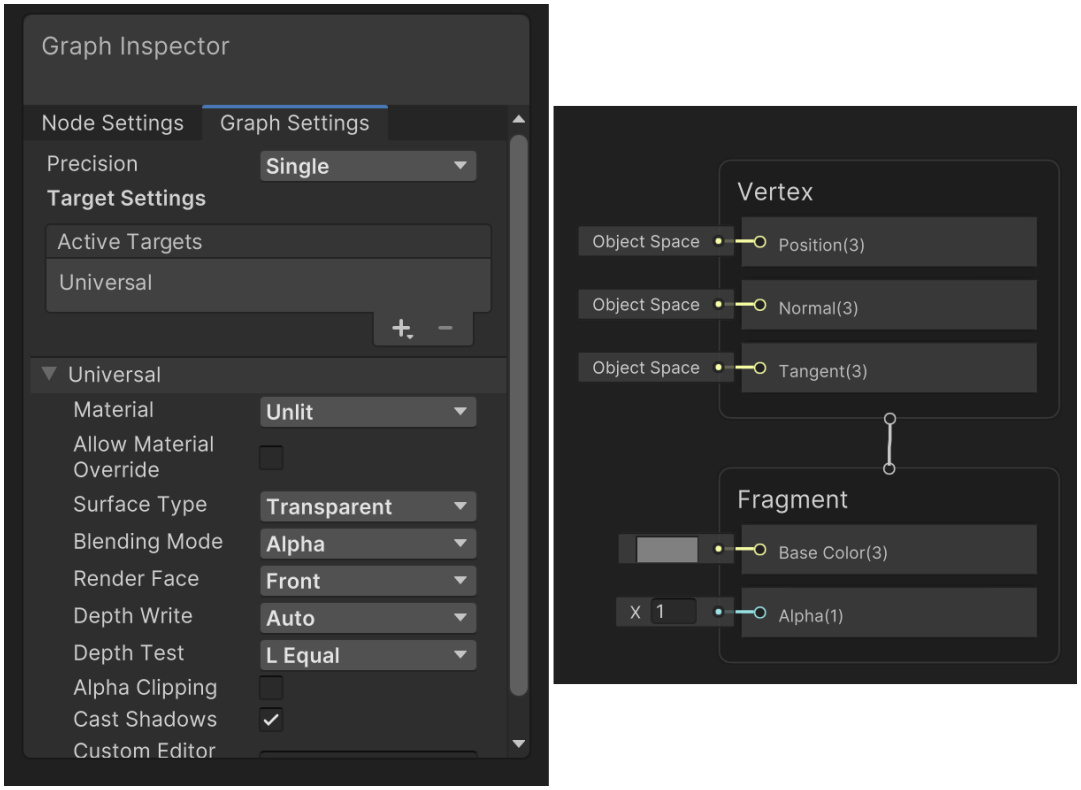


2. 새 **Shader Graph 에셋**을 더블 클릭하여 Shader Graph 에디터를 실행합니다.

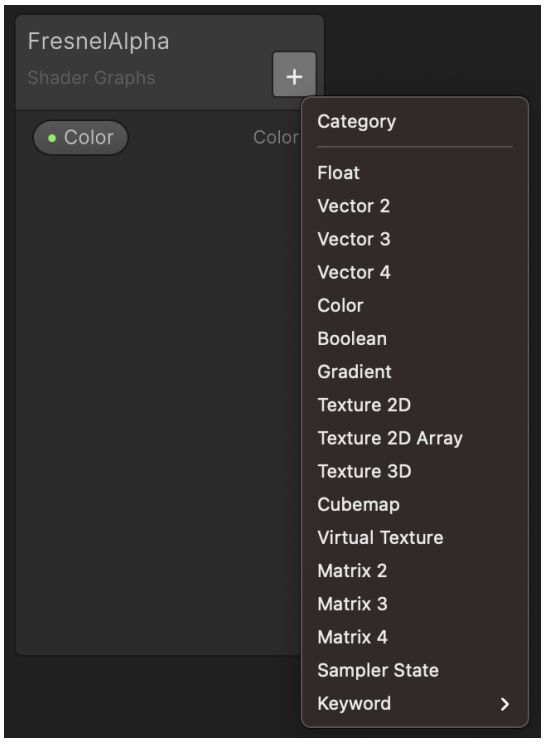


셰이더에 익숙하다면 Vertex 및 Fragment 노드를 금방 알아볼 수 있을 것입니다. 기본적으로 이 셰이더는 셰이더를 사용하는 머티리얼이 포함된 모든 모델이 Vertex 노드를 통해 카메라 뷰에 올바르게 배치되고, 각 픽셀이 Fragment 노드를 사용해 회색으로 설정되도록 합니다.

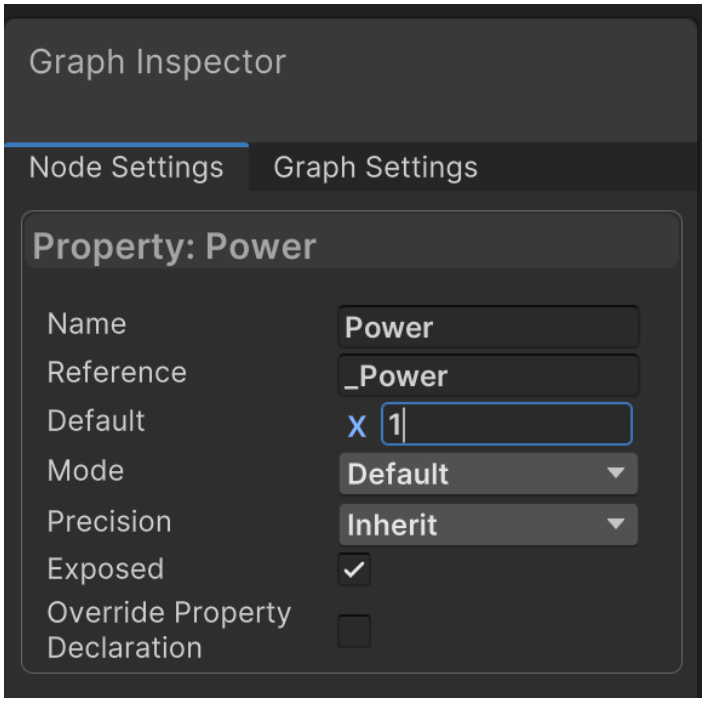
3. 이 셰이더에서 오브젝트의 알파 투명도가 설정되며, 따라서 Transparent 대기열에 이를 적용해야 합니다. **Graph Inspector > Graph Settings > Surface Type**을 **Transparent**로 변경합니다. 그러면 Fragment 노드에 Alpha 입력과 Base Color가 있는 것을 확인할 수 있습니다.



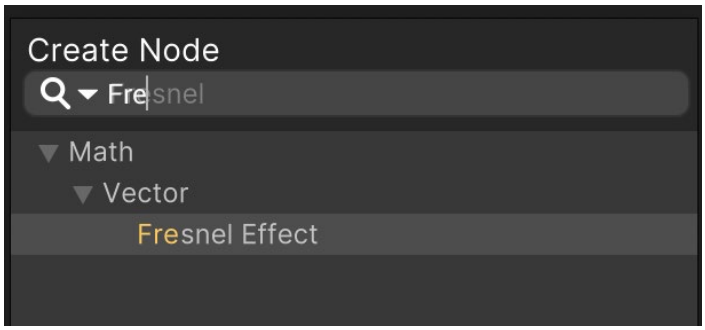
4 프로퍼티를 셰이더에 추가합니다. 예를 들어 Color를 Color로, Power 및 Strength를 Float 값으로 추가할 수 있습니다.



5. **Graph Inspector > Node Settings > Default**를 사용하여 기본값을 설정합니다. Color를 흰색으로, Power를 4로, Strength를 1로 설정합니다.



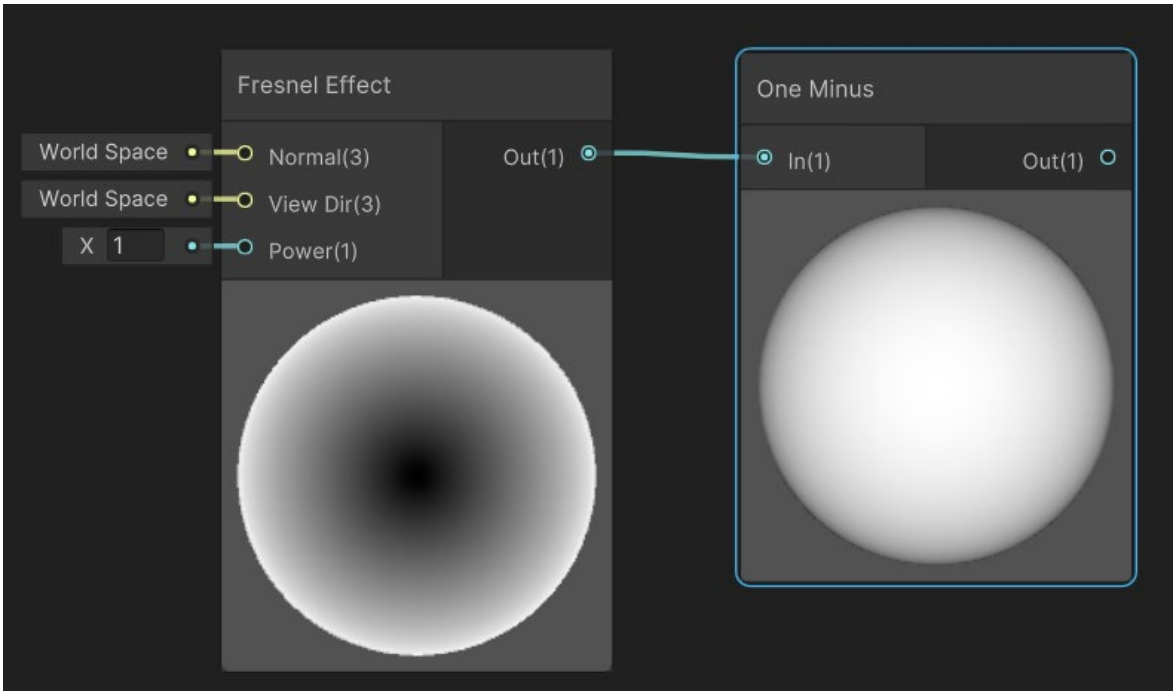
6. Shader Graph는 노드를 연결하여 작동하며, 노드에는 하나 이상의 입력과 출력이 있습니다. 노드를 추가하려면 오른쪽 클릭하고 상단의 **Search** 패널에서 **Create Node**를 선택한 다음 **Fre**를 입력합니다. 그러면 결과에 **Fresnel Effect** 노드가 표시됩니다.



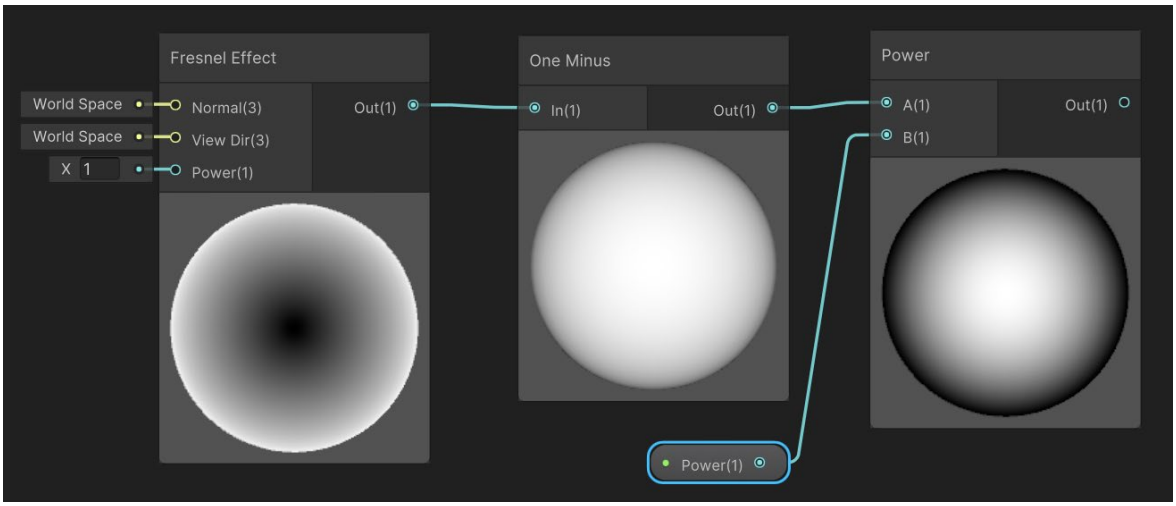
7. 노드에는 효과 프리뷰가 표시됩니다. 에지로 갈수록 프레넬 효과가 밝아지는 것을 볼 수 있습니다. 이 값은 뷰 방향과 노멀 방향 간의 차이에 해당되며, 구체의 경우 에지에서 값이 가장 큽니다.

   알파 값은 에지에서 가장 작습니다. One Minus 노드를 사용하여 결과를 뒤집을 수 있습니다. **Create Node**를 클릭하고 **One**을 입력하면 됩니다. **One Minus** 노드를 선택합니다. Fresnel Effect 노드의

Out(1)에서 One Minus 노드의 In(1)으로 드래그합니다. 1은 값 유형이 단일 플로트임을 의미하며, 값이 3이라면 세 개의 컴포넌트가 있는 벡터가 됩니다. 노드는 다음과 같이 연결됩니다.

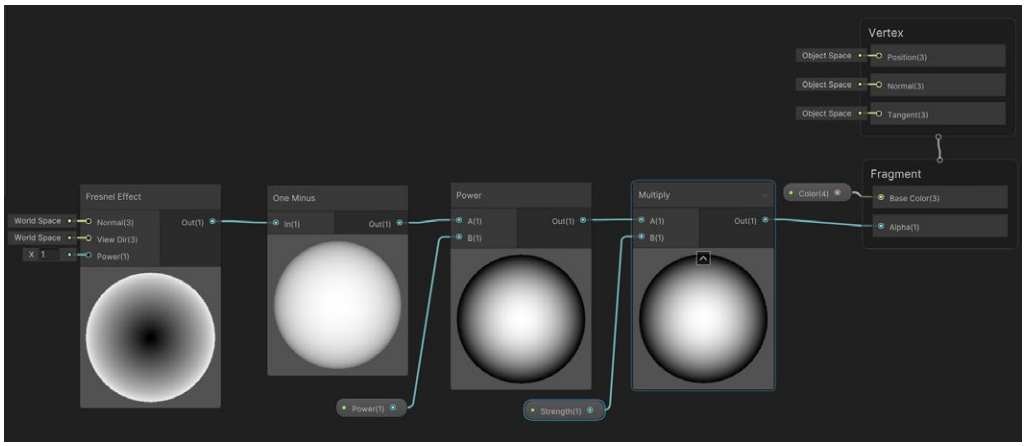


8. 그레디언트 크기와 전체 투명도를 제어하는 방법을 살펴보겠습니다. **Power** 노드를 사용하여 그레디언트 크기를 조절합니다. Power 노드를 만들고 One Minus의 Out(1)을 Power의 A(1)에 연결합니다. Power 프로퍼티를 그래프로 드래그하고 Power의 B(1)에 연결합니다. 그러면 그래프가 다음과 같이 표시됩니다.

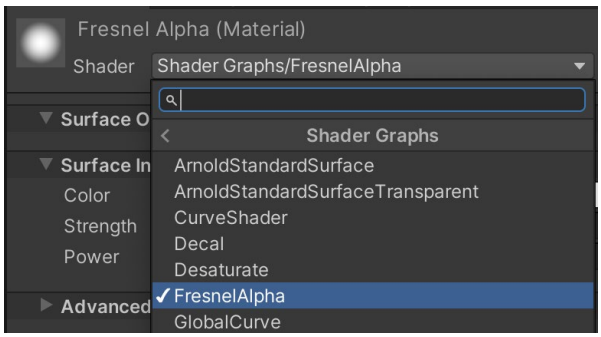


9. **Multiply** 노드를 사용하여 전체 투명도를 제어합니다. 노드를 만들고 Power의 Out(1)을 Multiply의 A(1)에 연결합니다. Strength 프로퍼티를 그래프로 드래그하고 Multiply의 B(1)에 연결합니다. 그런 다음 Multiply의 Out(1)을 Fragment의 Alpha(1)에 연결하고, Color(4) 프로퍼티를 그래프로 드래그하여 Fragment의 Base Color(3)에 연결합니다.

Color 프로퍼티는 네 개의 컴포넌트 벡터로 구성된 반면, Base Color는 세 개의 컴포넌트 벡터로 구성됩니다. Shader Graph는 Color의 첫 번째 세 개 컴포넌트를 Base Color 벡터에 매핑합니다.



10. 에셋을 저장하고 새 머티리얼을 만듭니다. 이 셰이더를 **Shader Graphs/FresnelAlpha**에 있는 새 머티리얼에 할당합니다.

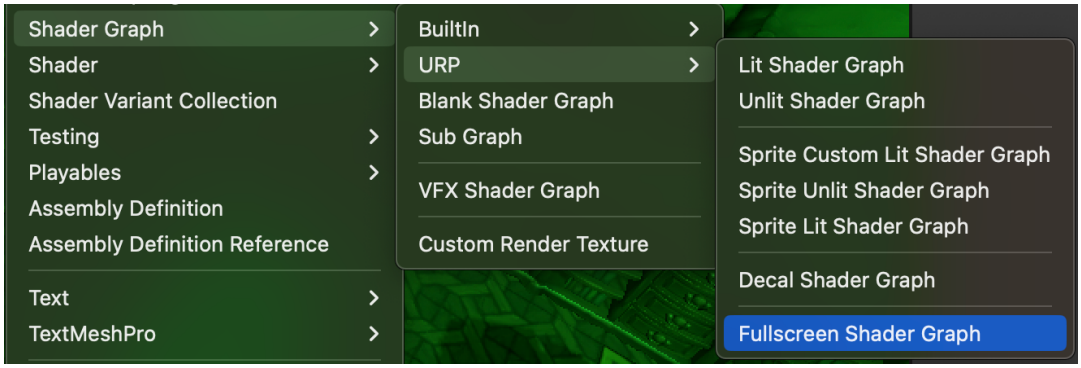


11. 이제 머티리얼을 오브젝트에 적용하여 에지의 가시성을 제어할 수 있습니다.

구체 모양의 점 광원에 셰이더를 적용하여 주위에 헤일로 효과 구현

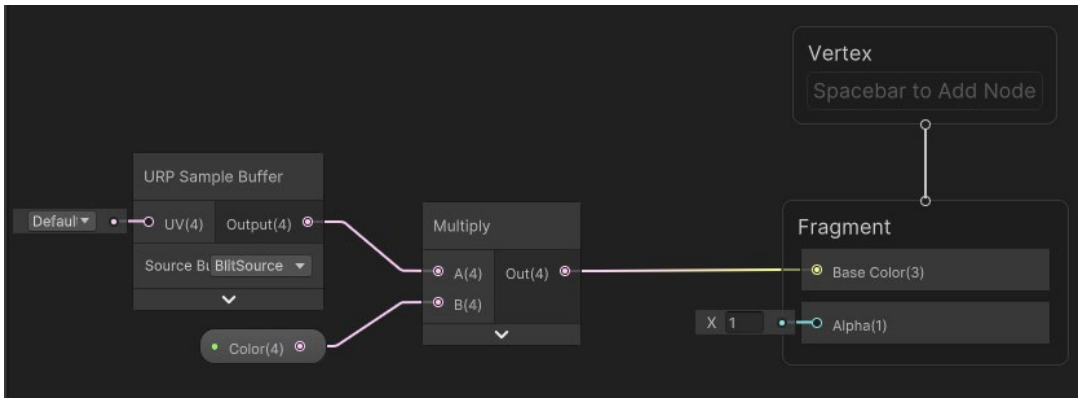
Fullscreen Shader Graph

Fullscreen Shader Graph를 사용하면 커스텀 포스트 프로세싱 패스를 만들 수 있습니다. 프로젝트 창에서 오른쪽 클릭하고 **Create > Shader Graph > URP > Fullscreen Shader Graph**를 선택합니다.



Fullscreen Shader Graph 생성

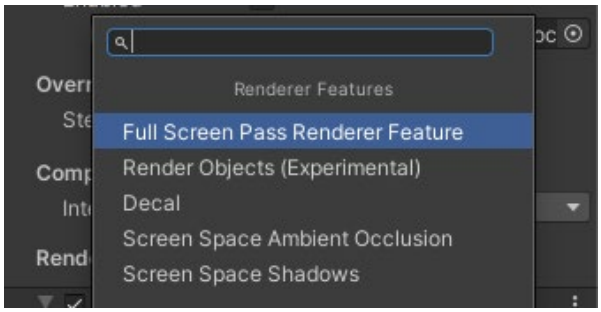
BlitSource 옵션을 사용하는 URP Sample Buffer 노드로 프래그먼트 셰이더의 픽셀 색상에 액세스할 수 있습니다. 아래 그래프는 간단한 틴트 예시를 보여 줍니다. URP Sample Buffer를 사용하면 에지 감지와 모션 트레일에 유용한 월드 노멀과 모션 벡터에 액세스할 수도 있습니다.



간단한 틴트 예시

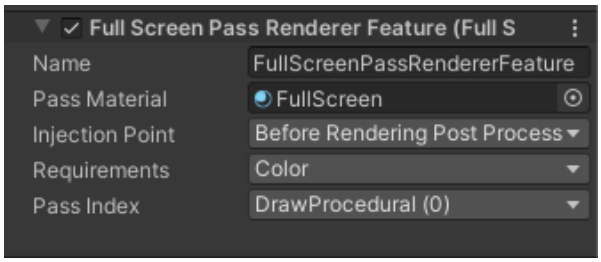
이 예시를 사용하려면 이 셰이더를 사용하는 머티리얼로 현재 카메라 렌더 텍스처의 결과를 Blit 처리할 방법이 필요합니다.

활성 렌더러 데이터 에셋을 선택한 다음 인스펙터에서 렌더러 기능을 추가합니다. **Full Screen Pass Renderer Feature**를 선택합니다.



Full Screen Pass Renderer Feature 추가

계속해서 이 렌더러 기능의 설정을 업데이트할 수 있습니다. 앞서 생성한 Fullscreen Shader Graph를 사용하는 머티리얼을 설정하고, 렌더 파이프라인에서의 위치를 지정합니다.



렌더러 기능 설정

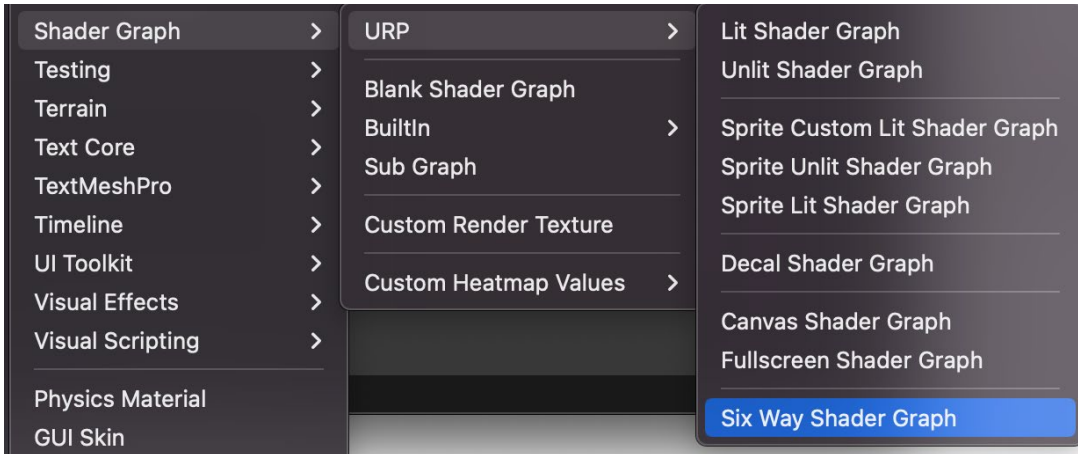
아래 이미지의 좌측에서 틴트 효과를 볼 수 있습니다. Fullscreen Shader Graph는 커스텀 포스트 프로세싱 효과를 만드는 데 유용합니다.

틴트 효과

Six Way Shader Graph

다양한 조명 효과에서 구현된 같은 연기 효과의 4가지 버전. 왼쪽 상단: 방향 광원 및 앰비언트, 오른쪽 상단: 앰비언트, 프로브 볼륨 및 방향 광원, 왼쪽 하단: 앰비언트 및 프로브, 오른쪽 하단: 스폿 광원 및 프로브

Six Way Shader Graph는 Houdini, Blender, Embergen 같은 DCC 툴에서 베이크할 수 있는 6방향 라이트맵을 사용해 연기 효과의 빛을 동적으로 재처리하여 더 사실적으로 표현할 수 있는 Unity 6 기능입니다.



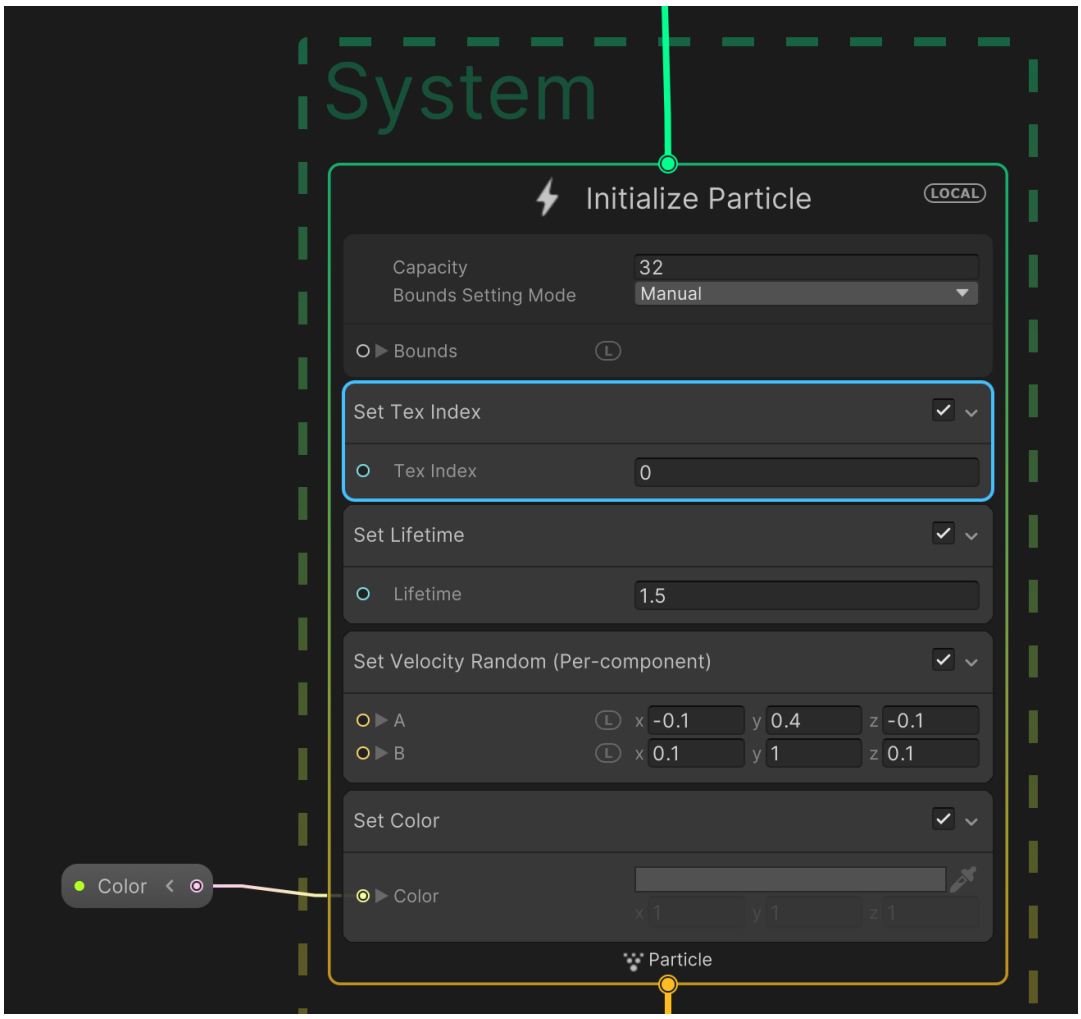
Six Way Shader Graph 생성

6방향 조명에 대해 자세히 알아보려면 VFX Graph에서 6방향 조명으로 사실적인 연기 조명 구현 블로그 게시물을 참고하세요. Shader Graph 프로세스를 살펴보며 예시 프로젝트와 몇 가지 고급 팁을 제공하는 이 블로그 게시물도 확인해 보세요.

VFX Graph

Unity에서는 캐릭터의 손끝에서 불덩이를 발사하거나 웜홀을 통과하는 등 거의 모든 시각 효과를 구현할 수 있습니다. 게임의 VFX(시각 효과)는 분위기를 조성하고, 스토리를 전달하는 데 도움이 되며, 플레이어의 깊은 몰입을 유도하는 디테일을 추가할 수 있습니다.

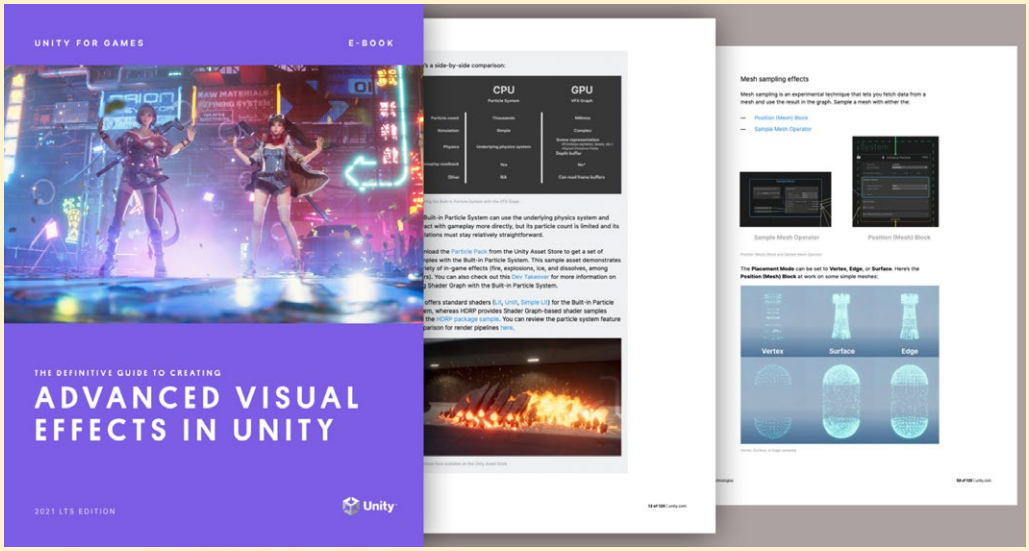
유니티는 VFX Graph 같은 툴로 실시간 그래픽스의 한계를 넓히고 있습니다. 테크니컬 아티스트와 VFX 아티스트는 노드 기반 에디터인 VFX Graph를 사용하여 단순하고 평범한 파티클 동작부터 파티클, 선, 리본, 트레일, 메시 등이 포함된 복잡한 시뮬레이션까지 역동적인 시각 효과를 디자인할 수 있습니다.



VFX Graph 편집

URP에서 VFX Graph로 VFX를 제작하는 방법에 대해 자세히 알아보려면 Unity의 고급 시각 효과 집중 가이드 전자책을 다운로드하세요.

VFX Graph로 제작한 연기 효과

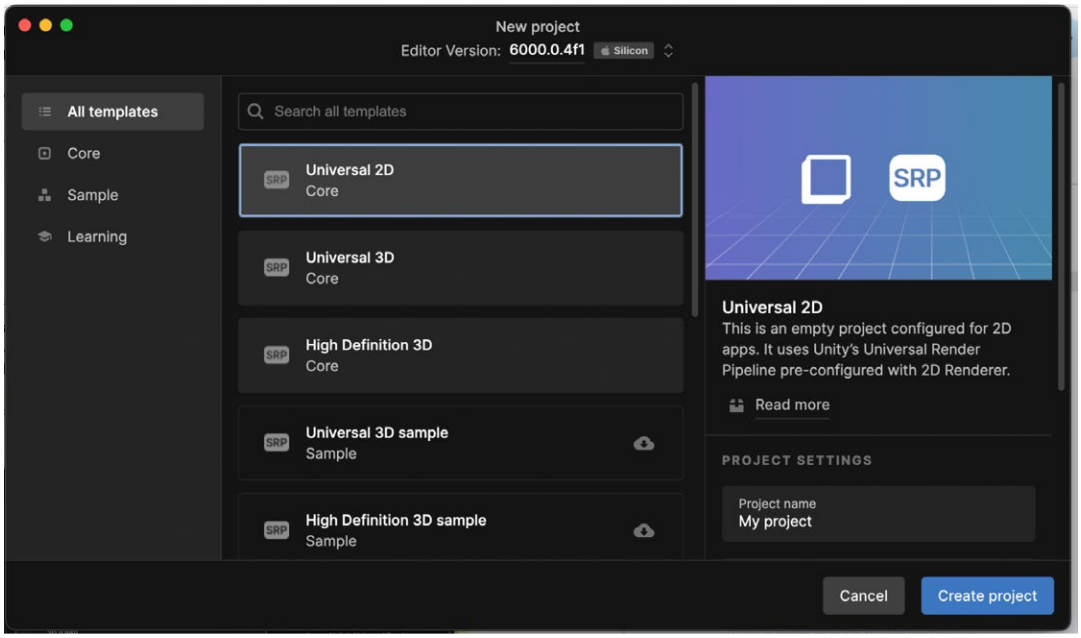


Unity 전자책 다운로드

2D 렌더러 및 2D 광원

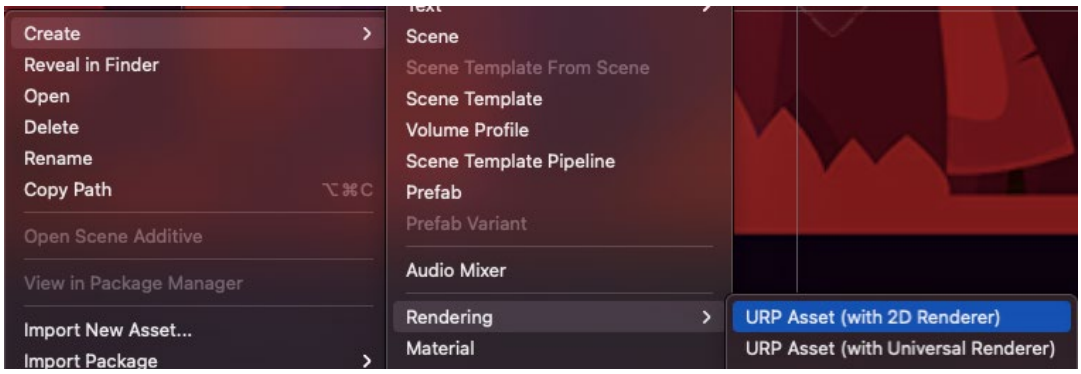
이제 2D 게임의 혁신에는 한계가 없습니다. 하드웨어, 그래픽스, 게임 개발 소프트웨어의 발전 덕분에 실시간 광원, 고해상도 텍스처, 거의 무한에 가까운 수의 스프라이트를 사용하여 2D 게임을 만들 수 있습니다.

2D 게임을 제작하고 있다면 전용 URP 2D 렌더러가 익숙하게 느껴질 것입니다. Unity Hub에서 2D URP 템플릿을 사용하면 가장 간단하게 시작할 수 있습니다. 이 템플릿을 사용하면 프로젝트에서 Project Settings > Graphics > Scriptable Render Pipeline Settings를 통해 URP 2D Renderer가 할당됩니다. 2D URP 템플릿과 함께 모두 검증되고 사전 컴파일된 2D 패키지가 설치되며 기본 설정은 2D 프로젝트에 최적화됩니다. 또한 모든 패키지를 수동으로 설치하는 것보다 프로젝트가 더 빨리 로드됩니다.



Unity Hub의 2D URP 템플릿

기존 프로젝트를 업그레이드하는 경우 프로젝트의 Assets 폴더에서 적합한 폴더를 찾아야 합니다. 오른쪽 클릭하고 Create > Rendering > URP Asset (with 2D Renderer)을 선택합니다. 이름을 지정하고 Project Settings > Graphics > Scriptable Render Pipeline Settings를 사용하여 선택합니다. 씬 뷰에서 편집할 때 2D 버튼을 반드시 눌러야 합니다.



2D 렌더러 및 설정 에셋 만들기

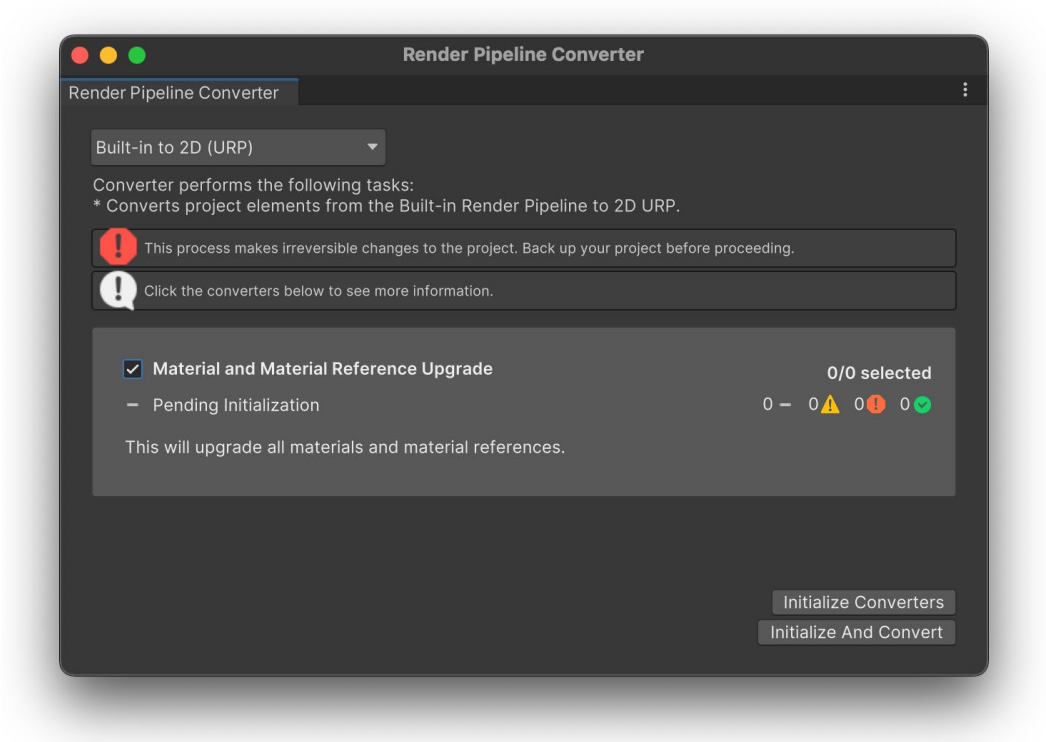
기존 프로젝트를 URP 2D 렌더러로 전환하면 렌더링 오류를 '대표하는' 마젠타 색상이 나타날 수 있습니다.

URP 2D 렌더러로 기존 프로젝트를 업데이트하면 씬에서 렌더링 오류가 발생할 수 있습니다.

다행히도 이러한 문제는 **Window > Rendering > Render Pipeline Converter**를 통해 해결할 수 있습니다. **Built-in to 2D (URP)**를 선택하고 Material 및 Material Reference Upgrade 패널을 클릭합니다. 그런 다음 Initialize Converters, Convert Assets를 차례로 클릭하여 일부 항목을 선택 해제하거나 Initialize And Convert를 클릭하여 한 번의 클릭으로 프로세스를 처리합니다. 아직 마젠타 색상 스프라이트가 표시되는 경우 일부 머티리얼에서 셰이더를 수동으로 대체해야 할 수도 있습니다. 다음 표에서 셰이더를 하나 선택합니다.

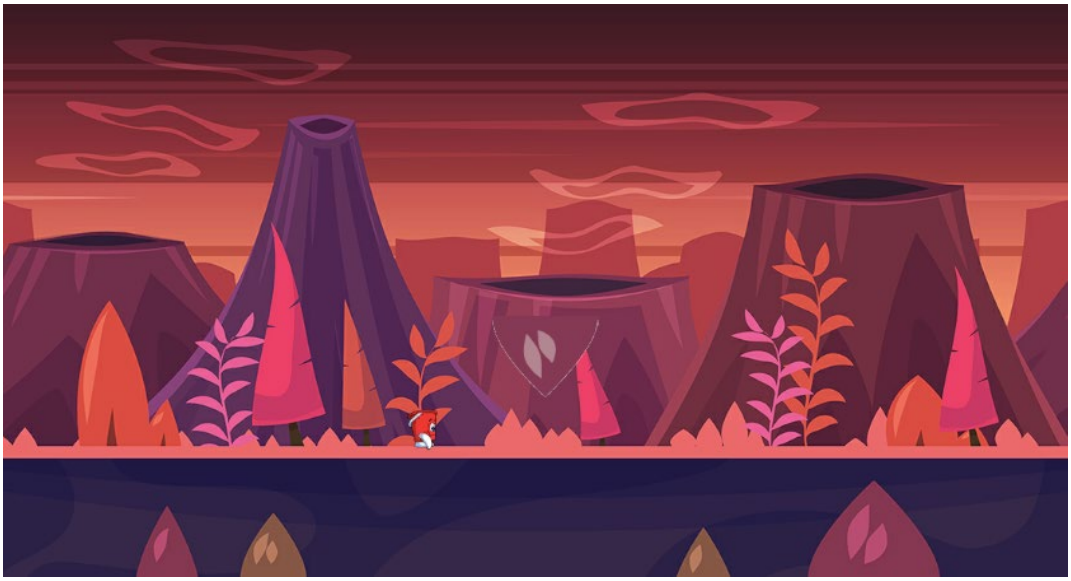
URP에서 제공되는 2D 셰이더

셰이더	설명
Sprite-Lit-Default	렌더링 시 2D 광원을 사용합니다.
Sprite-Mask-Default	스텐실 버퍼와 함께 작동합니다.
Sprite-Unlit-Default	렌더링 시 텍스처 색상만 사용합니다.



빌트인 렌더 파이프라인 2D 프로젝트를 URP 2D로 전환

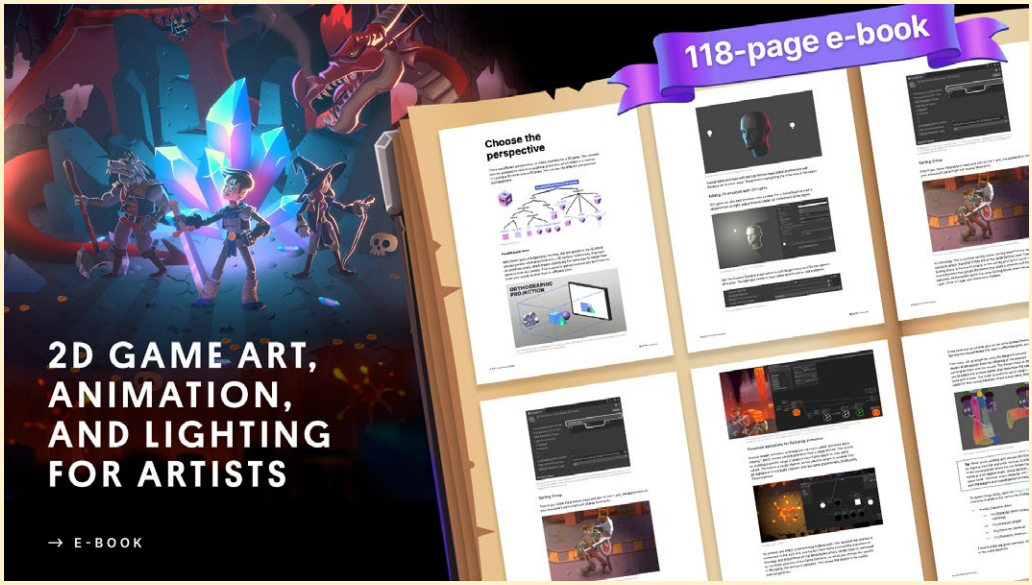
URP 2D 렌더러에서 2D 광원을 사용할 수 있으며 이를 통해 성능과 유연성을 향상할 수 있습니다. 최신 툴을 사용하면 더욱 몰입도 높은 경험을 만들 수 있고, 베이크된 광원을 사용해 다양한 스프라이트 배리에이션을 준비하는 시간을 단축하여 보다 창의적인 게임플레이를 구현할 수 있습니다. 기존 프로젝트를 마이그레이션한 경우에는 씬에 URP 2D 광원이 없을 것입니다. 스프라이트에서 Sprite-Lit-Default 셰이더를 사용한다면, 릿 렌더를 보고 놀랄 수도 있습니다. 하지만 광원이 없으면 언릿 형상에 대해 기본 전역 광원이 씬에 할당됩니다.



씬에 광원이 없으면 렌더가 Unlit으로 기본 설정됩니다.

유니티의 2D 게임 개발 리소스

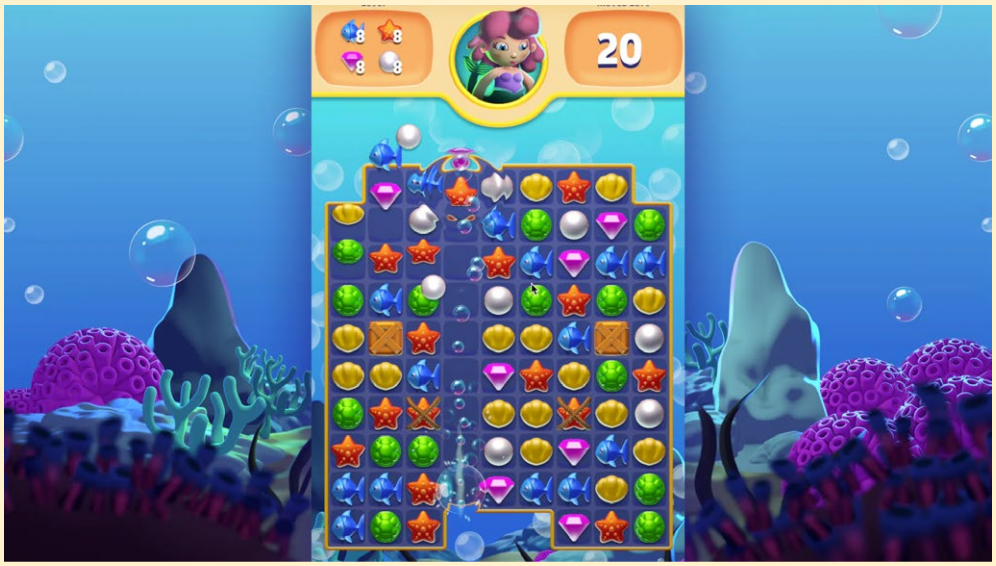
아티스트를 위한 2D 게임, 애니메이션 및 조명 전자책은 상업용 2D 게임을 제작하려는 Unity 개발자와 아티스트를 위한 가장 포괄적인 2D 개발 가이드입니다.



2D 그래픽스, 툴, 애니메이션에 관한 Unity 전자책 다운로드

유니티의 2D 샘플 프로젝트

젬 헌터 매치



젬 헌터 매치(Gem Hunter Match)는 URP로 매력적인 조명 및 시각 효과를 구현하여 차별화된 2D 퍼즐/매치3 게임을 선보이는 샘플입니다. 2D 스프라이트에 광원을 적용해 뎁스를 더하고, 스프라이트 커스텀 릿 셰이더로 일렁이는 빛 효과를 구현하고, 눈부심 및 잔물결 효과를 만드는 방법을 비롯한 다양한 기법을 배울 수 있습니다.

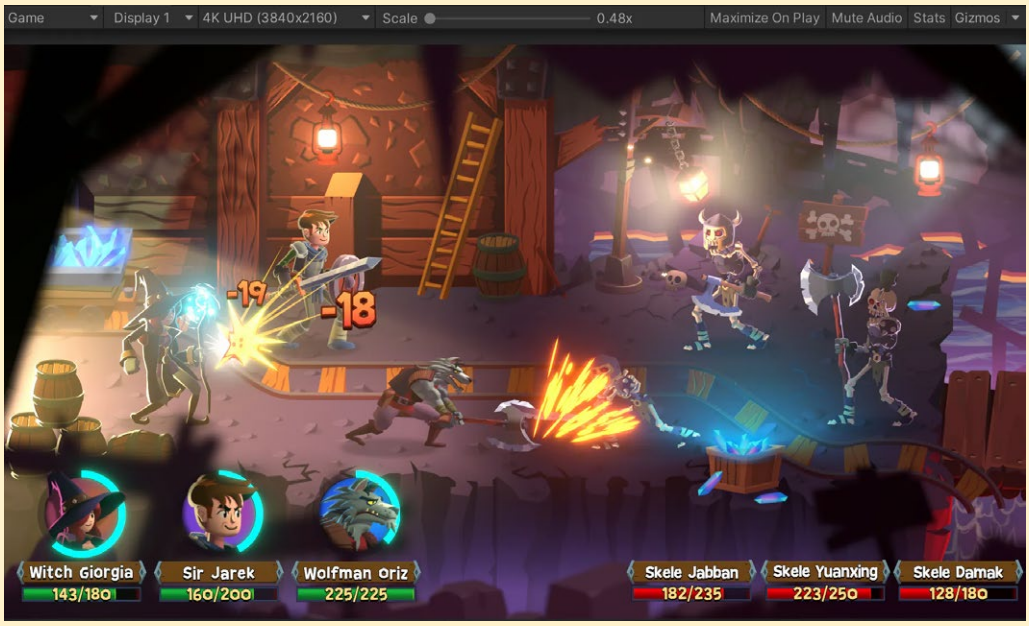
- 젬 헌터 매치 다운로드
- 젬 헌터 매치에 대해 알아보기

해피 하비스트

해피 하비스트(Happy Harvest)에서는 URP로 2D 광원, 그림자, 특수 효과를 연출하는 최신 기능을 어떻게 활용하는지 살펴볼 수 있습니다. 스프라이트에 그림자 베이크하지 않기, 스프라이트 플랫 상태 유지하기, 보조 텍스처에 그림자 및 볼륨 정보 옮기기, 고급 타일맵 기능 등 모든 2D 크리에이터가 활용할 수 있는 베스트 프랙티스가 담겨 있습니다.

- 해피 하비스트 다운로드
- 해피 하비스트에 대해 알아보기

UI 툴킷 샘플 – 드래곤 크래셔



이 2D 샘플 프로젝트는 횡스크롤 방치형 RPG 게임의 버티컬 슬라이스(시연 버전)로, 2D 툴 제품군에 아트워크를 더해 비전을 실현할 수 있다는 것을 보여 줍니다. 이 데모에 있는 모든 콘텐츠는 사용자의 자체 프로젝트에 추가할 수 있습니다.

- UI 툴킷 – 드래곤 크래셔 다운로드
- UI 툴킷 – 드래곤 크래셔에 대해 알아보기

URP용 Unity 6 기능 더 알아보기

이 섹션에서는 다양한 기기에서 성능이나 정확도를 향상할 수 있는 Unity 6의 URP용 주요 기능을 살펴봅니다.

STP(시공간 포스트 프로세싱)

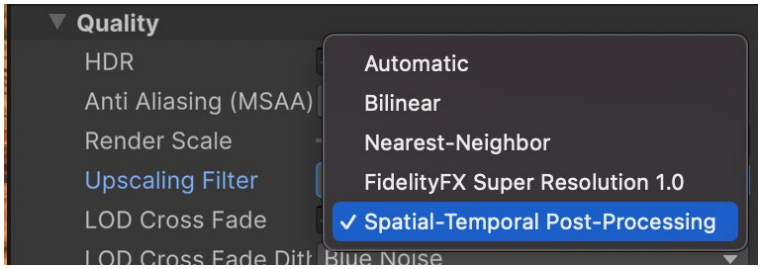
3가지 방식으로 렌더링한 URP 3D 샘플의 오아시스 환경. 왼쪽: 렌더 스케일 1, 가운데: 렌더 스케일 0.25, 오른쪽: 렌더 스케일 0.25 및 STP 활성화

점점 더 많은 초고해상도 기법이 생기고 있지만 여전히 많은 기법이 특정 하드웨어 벤더 전용으로 남아 있습니다. 게다가 이러한 솔루션은 모바일에 맞게 설계되지 않았습니다.

STP(시공간 포스트 프로세싱)는 모바일에서 콘솔까지 모든 기기에서 사용할 수 있는 Unity의 초고해상도 솔루션으로 URP에서 오버헤드는 줄이며 고품질의 결과물을 제공합니다. 위의 블렌딩된 이미지에서도 뛰어난 품질을 확인할 수 있습니다. 왼쪽 1/3은 STP가 비활성화된 렌더 스케일 1의 이미지입니다. 가운데 1/3은 STP가 비활성화된 렌더 스케일 0.25의 흐릿하고 정확도가 낮은 이미지입니다. 오른쪽 1/3은 렌더 스케일을 0.25로 유지하면서 STP를 활성화하여 픽셀의 1/16만 렌더링하면서도 STP로 선명하게 업스케일링했습니다.

STP는 셰이더 모델 5.0을 지원하는 모든 기기에서 사용할 수 있는 시공간 안티앨리어싱 업스케일링 솔루션입니다. 모바일을 염두에 두고 성능 비용은 낮추면서 DLSS2, FSR2, XeSS와 비슷한 시각적 품질을 제공하는 것을 목표로 설계되었습니다.

활성 URP 에셋의 **Quality > Upscaling Filter**에서 STP를 활성화할 수 있습니다. 그런 다음 렌더 스케일을 설정합니다.



시공간 포스트 프로세싱 활성화

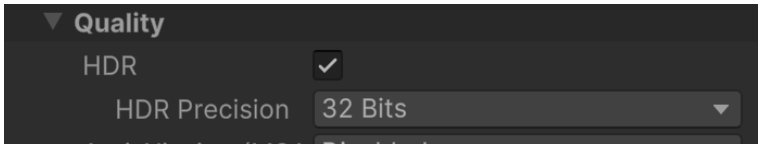
PC 및 콘솔용 HDR 디스플레이 출력

Rendering Debugger > Lighting > HDR Debug Mode > Gamut View

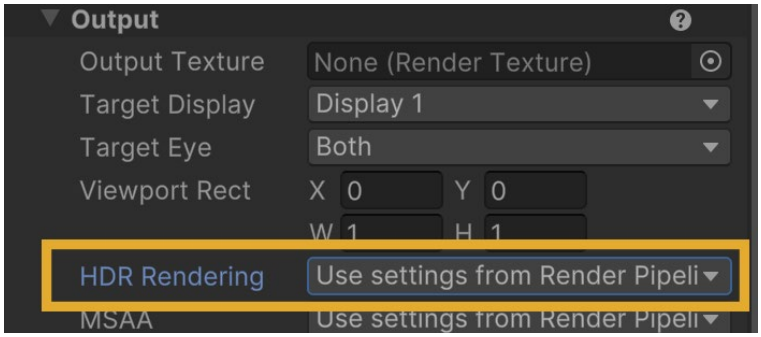
HDR(High Dynamic Range) 디스플레이는 휘도 차이가 큰 이미지를 자연광 환경에 더욱 유사하도록 재생성할 수 있는 디스플레이 기기입니다. HDR Output을 사용하면 선형 조명 렌더 및 HDR 이미지의 콘트라스트와 품질을 더 효과적으로 보존하여 해당 기기에서 표현할 수 있습니다.

HDR은 두 곳에서 활성화합니다.

1. URP 에셋



2. 카메라 인스펙터: **Output > HDR Rendering**을 **Use settings from Render Pipeline Asset**으로 설정합니다.



HDR을 활성화하면 향상된 품질의 URP 렌더 출력 이미지를 HDR 디스플레이에 표시하는 옵션을 사용할 수 있습니다. 그 결과 해당 기기에서 자연광 환경을 더욱 효과적으로 모사하는 색상과 콘트라스트로 최종 이미지를 구현할 수 있습니다.

Unity 2022에서 추가되었던 기존 데스크톱 및 콘솔 지원에 더해 Unity 6에서는 다음 플랫폼에 모바일 지원이 추가되었습니다.

— iOS 플레이어(iOS 16+, iPadOS 16+)

— Vulkan 및 GLES를 사용하는 Android 플레이어(Android 9+, 기기 성능에 따라 상이)

HDR 디스플레이를 지원하는 일반적인 모바일 기기로는 iPhone X 이상, 삼성 갤럭시 S10 이상, 갤럭시 노트10 이상, 갤럭시 탭 S6 이상 버전이 있습니다.

파이프라인 상태 오브젝트 사용

Unity 6에 새롭게 추가된 강력한 PSO(파이프라인 상태 오브젝트) 추적 및 사전 쿠킹 워크플로를 사용하면 최신 플랫폼을 타게팅할 때 부드럽고 끊김 없는 플레이어 경험을 제공할 수 있습니다.

이 API 세트는 이전 Unity 버전에 추가된 '셰이더 웜업(warmup)' API보다 훨씬 업그레이드된 기능을 제공합니다. 이전 그래픽스 API(OpenGL, DirectX11 등)는 기존 셰이더 웜업만으로 충분했지만, 새로운 PSO 워크플로를 사용하면 Vulkan, DirectX12, Metal 같은 최신 그래픽스 API를 더 잘 활용할 수 있습니다.

PSO 생성 및 캐싱

최신 그래픽스 API를 타게팅하는 경우, GPU 벤더의 그래픽스 드라이버가 PSO 생성 프로세스 중 런타임 셰이더 컴파일 및 기타 렌더링 상태 이동을 수행합니다. 그 결과, PSO 생성은 긴 프로세스이므로 런타임 애플리케이션에서 눈에 띄는 끊김 현상이 발생할 수 있습니다. 복잡한 프로젝트일수록 애플리케이션에서 대량의 PSO를 즉석에서 컴파일해야 하므로 이 오버헤드는 더 심해질 수 있습니다.

Unity 프로파일러에서 GraphicsPipelineImpl 마커를 사용하여 PSO 생성 끊김 현상을 확인할 수 있습니다.

PSO 생성 프로파일링

많은 경우 GPU 벤더의 그래픽스 드라이버는 향후 애플리케이션 실행 시 PSO 생성 속도를 단축하기 위해 컴파일된 모든 PSO를 디스크에 자동으로 캐싱합니다. 하지만 새로운 셰이더 배리언트나 머티리얼은 애플리케이션이 여전히 PSO를 컴파일해야 할 수 있습니다. 또한 OS나 드라이버 업데이트로 드라이브에서 관리되는 PSO 캐시를 사용할 수 없게 될 수도 있습니다.

PSO를 웜업하는 이상적인 방법은 애플리케이션과 활용 사례에 따라 다를 수 있습니다. 예를 들어 레벨 전환과 씬 로딩 시 PSO를 동기식으로 사전 쿠킹하는 방식을 선택할 수 있습니다. 애플리케이션 반응성을 개선하려면 점진적으로 시간을 분할하여 프레임당 고정된 양의 PSO를 생성하는 방식으로 처리할 수 있습니다.

또는 애플리케이션 백그라운드에서 비동기식으로 PSO를 사전 쿠킹할 수도 있습니다. 이렇게 하면 애플리케이션이 중단되지 않지만, 웜업하는 기간 동안 일시적으로 CPU 성능이 저하될 수 있습니다.

새로운 PSO 컬렉션 추적

렌더링 도중 애플리케이션이 생성한 PSO부터 추적을 시작해야 합니다.

1. C# 스크립트에 새로운 GraphicsStateCollection을 생성합니다. 이 컬렉션이 애플리케이션 또는 씬의 PSO를 나타냅니다.
2. 컬렉션에 PSO를 추적해 넣으려면 GraphicsStateCollection.BeginTrace 메서드를 호출합니다. 애플리케이션에서 생성한 모든 신규 그래픽스 파이프라인이 컬렉션에 추가됩니다. 대부분의 경우, 씬 또는 애플리케이션 시작 시 추적을 시작하는 것이 좋습니다.
3. 추적 프로세스를 종료하려면 GraphicsStateCollection.EndTrace 메서드를 호출합니다. 대부분의 경우, 씬 또는 애플리케이션 종료 시 추적을 종료하는 것이 좋습니다.

추적을 완료한 후, GraphicsStateCollection.SaveToFile을 사용하여 PSO 컬렉션을 디스크에 저장할 수 있습니다.

추가적으로 제어하려면 저장된 PSO와 배리언트 데이터를 확인하고 필요에 따라 컬렉션을 수정할 수 있습니다. GraphicsStateCollection.GetVariants를 사용하면 PSO 컬렉션에 저장된 모든 셰이더 배리언트를 가져올 수 있습니다. 그런 다음 GraphicsStateCollection.GetGraphicsStatesForVariant를 통해 각 배리언트가 사용하는 그래픽스 상태를 확인할 수 있습니다. 마지막으로, AddGraphicsStateForVariant/RemoveGraphicsStatesForVariant를 사용하여 각 배리언트와 연관된 그래픽스 상태를 수정할 수 있습니다.

참고:

> 플랫폼마다 GPU에서 나타나는 PSO가 달라질 수 있습니다. 관련 그래픽스 API를 타게팅하여 플레이어에서 추적을 수행하고 타겟별로 별도의 컬렉션을 유지하는 것이 강하게 권장됩니다.

Player Connection을 사용하는 타겟 기기에서 추적하는 경우, GraphicsStateCollection.SendToEditor를 사용하여 PSO 컬렉션을 에디터로 전송하고 디스크에 저장할 수 있습니다. 또한 GraphicsStateCollection.runtimePlatform을 사용하면 PSO 컬렉션 추적 시 사용되는 플랫폼을 쿼리할 수 있습니다.

PSO 컬렉션 사전 쿠킹

추적이 완료되면 Unity에 PSO 컬렉션의 사전 쿠킹을 요청할 수 있으며, 드로우하기 훨씬 전에 수행하는 것이 좋습니다. 대부분의 경우 애플리케이션 또는 씬 로딩 시퀀스 때 웜업을 수행하는 것이 이상적입니다.

2가지 웜업 메서드를 사용하여 PSO 사전 쿠킹을 수행할 수 있습니다.

- GraphicsStateCollection.WarmUp은 컬렉션 내 모든 PSO의 생성을 예약합니다.
- GraphicsStateCollection.WarmUpProgressively는 컬렉션 내 일정 개수의 PSO 생성을 예약합니다.

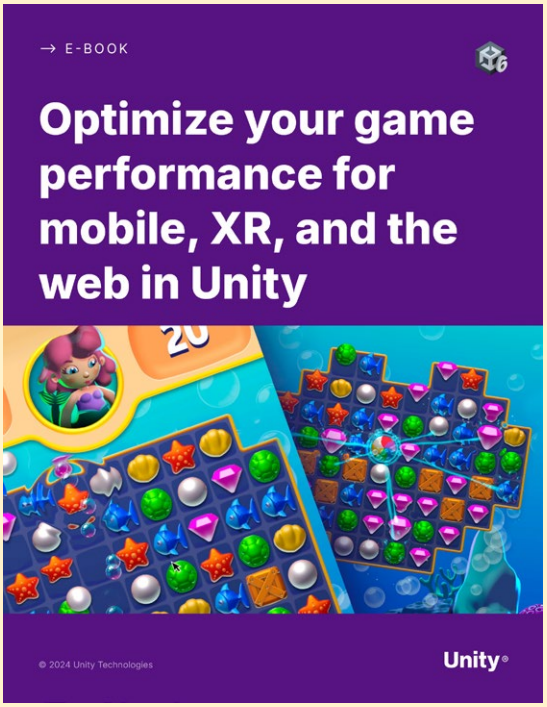
두 메서드 모두 잡 핸들을 반환하며, 이를 통해 PSO 웜업을 동기식으로 수행할지 비동기식으로 수행할지 결정할 수 있습니다.

PSO가 생성되면 드라이버는 생성된 PSO를 디스크에 캐싱합니다. 다음에 PSO를 사전 쿠킹할 때 캐시에서 바로 로드할 수 있습니다.

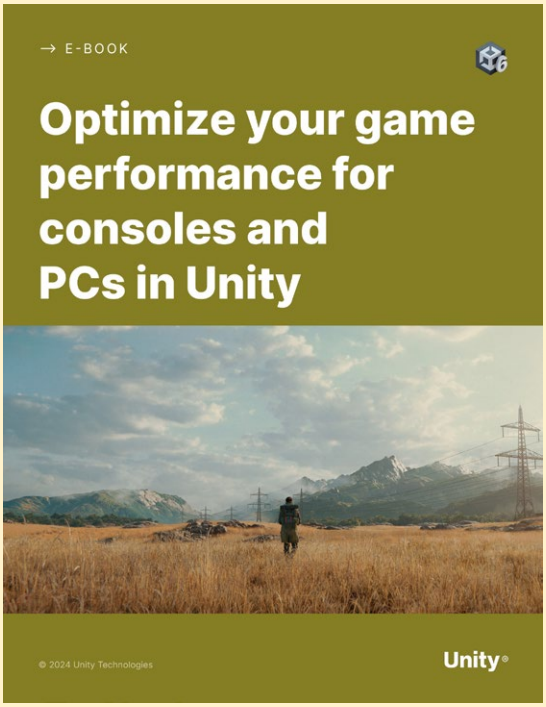
플랫폼 지원

새로운 PSO 워크플로는 Unity 6에서 Metal, Vulkan, Direct3D 12를 타게팅하는 플레이어에 사용할 수 있습니다. 향후 버전에서는 암시적 셰이더 웜업 폴백의 형태로 OpenGLES나 Direct3D 11 같은 이전 그래픽스 API와 호환되도록 지원할 예정입니다.

성능

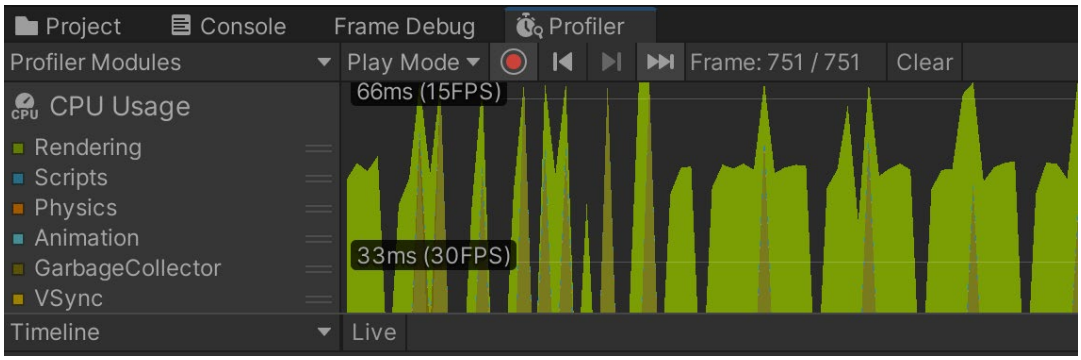


Unity 프로파일링 툴, 프로그래밍 및 코드 아키텍처, 프로젝트 설정 및 에셋 등에 관한 팁을 확인해 보세요. 모바일 게임의 성능을 향상하는 방법을 소개합니다.
다운로드



콘솔 및 PC 프로젝트의 광범위한 프로파일링, 프로그래밍 코드 및 아키텍처, 에셋 및 그래픽스 최적화, UI, 물리, 애니메이션 최적화에 관한 팁을 확인해 보세요.
다운로드

성능은 진행하는 프로젝트에 따라 크게 좌우될 수 있습니다. 따라서 개발 사이클 전체에서 항상 게임을 프로파일링하고 테스트해야 합니다. **Window > Analysis > Profiler** 로 이동하여 프로파일러를 열고 이 챕터에서 소개하는 팁을 따라 해 보세요.



프로파일러 창

이 섹션에서는 게임의 성능을 개선하는 다음 7가지 방법을 살펴봅니다.

- 조명 관리
- 라이트 프로브
- 반사 프로브
- 카메라 설정
- 파이프라인 설정
- 프레임 디버거
- 프로파일러

이러한 최적화는 이 튜토리얼(한국어 자막)에서도 확인할 수 있습니다.

다음 동영상 튜토리얼에서 프로파일링 팁을 살펴보세요.

시청하기

시청하기

시청하기

URP에서 조명 및 렌더링 최적화

URP는 최적화된 실시간 조명을 염두에 두고 만들어졌습니다. URP 포워드 렌더러는 오브젝트당 최대 8개의 실시간 광원을 지원하며, 데스크톱 게임용 카메라당 최대 256개의 실시간 광원에 더해 모바일 및 무선 하드웨어용 카메라당 32개의 실시간 광원을 지원합니다. URP에서는 파이프라인 에셋 내부에서 오브젝트당 광원을 설정하여 조명을 세부적으로 제어할 수 있습니다.

조명 챕터에서 설명한 내용과 같이, 베이크된 조명은 씬의 성능을 개선할 수 있는 좋은 방법입니다. 실시간 조명은 비용을 소모할 수 있지만, 그 반면에 베이크된 광원은 씬의 광원이 정적이라고 가정하고 성능을 높이도록 도움을 줄 수 있습니다. 베이크된 조명 텍스처는 연속으로 계산할 필요 없이 단일 드로우 콜로 배칭됩니다. 이는 씬에 여러 광원이 사용되는 경우에 특히 유용합니다. 조명을 베이크해야 하는 또 다른 합리적 이유는 씬에서 반사 조명과 간접 조명을 렌더링하여 렌더의 시각적 품질을 개선할 수 있기 때문입니다.

전역 조명은 조명 섹션에서 유사하게 다룹니다. 이 프로세스는 환경 주위에서 반사되고 반사광으로 주변에 있는 다른 오브젝트를 비추는 광원의 빛줄기를 시뮬레이션합니다. 아래 이미지는 베이크된 광원 데이터가 없는 경우와 있는 경우, 포스트 프로세싱이 적용된 경우의 세 가지 조명 설정을 보여 줍니다.

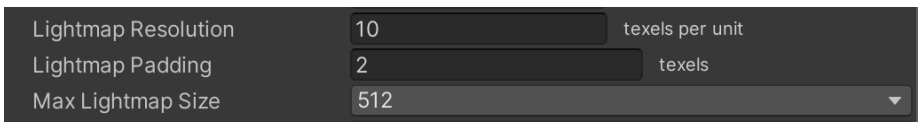
왼쪽에서 오른쪽으로: 조명 데이터 없음, 베이크된 조명, 포스트 프로세싱 추가

베이크한 경우 씬에서 반사광을 받은 그림자 영역은 빛이 납니다. 미미해 보일 수 있지만, 이 기술은 씬에 광원을 더 현실적으로 확산하여 전체적인 형상을 개선해 줍니다.

이전 이미지에서 베이크를 했을 때 지면의 스페큘러 하이라이트가 사라진 것을 볼 수 있습니다. 베이크된 광원에는 분산광만 포함됩니다. 가능하다면 항상 실시간으로 직접광의 영향을 계산하고 IBL(이미지 기반 조명)/섀도우 맵/프로브의 전역 조명을 사용하세요.

광원 베이크가 그림자에 미치는 영향: 왼쪽은 베이크 전, 오른쪽은 베이크 후

광원을 베이크할 때는 가장 낮은 **라이트맵 해상도**와 **라이트맵 크기**를 사용합니다. **Window > Rendering > Lighting > Scene**으로 이동합니다. 이렇게 하면 텍스처 메모리 요구 사항을 낮출 수 있습니다.

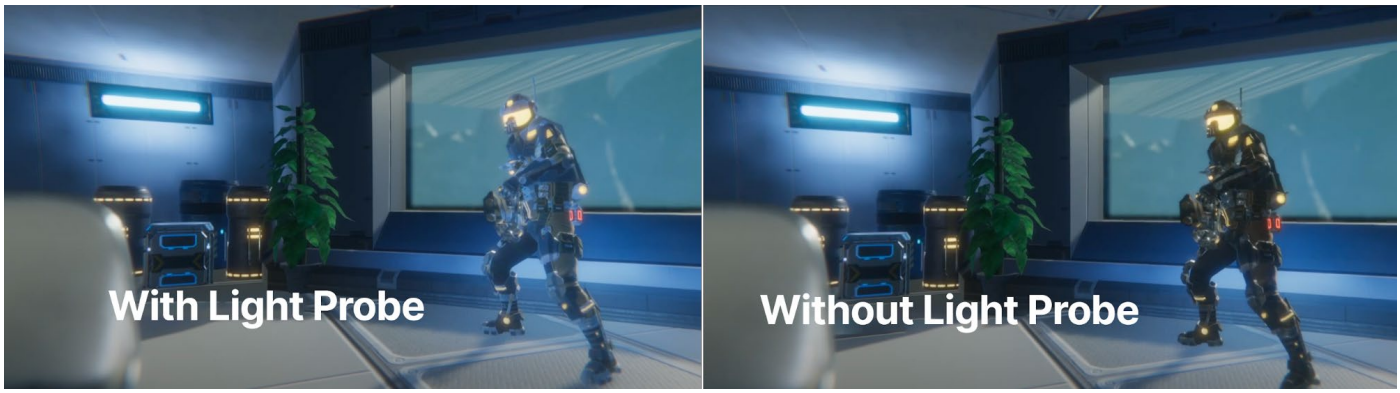


Lightmap Resolution과 Max Lightmap Size 설정

라이트 프로브

조명 섹션에서 설명한 것처럼, 라이트 프로브는 베이크 중에 씬의 조명 데이터를 샘플링하며 동적 오브젝트가 이동하거나 변경될 때 반사광 정보가 사용되도록 합니다. 이렇게 하면 베이크된 조명 환경에 블렌딩되어 더 자연스러운 느낌을 줄 수 있습니다. (이제 Unity 엔진에는 라이트 프로브의 대안이 마련되어 있습니다. APV 섹션을 참고하세요.)

라이트 프로브는 렌더링된 프레임에 필요한 프로세싱 시간을 늘리지 않고도 렌더에 자연스러움을 더합니다. 따라서 모든 하드웨어는 물론 저사양 모바일 기기에도 적합합니다.



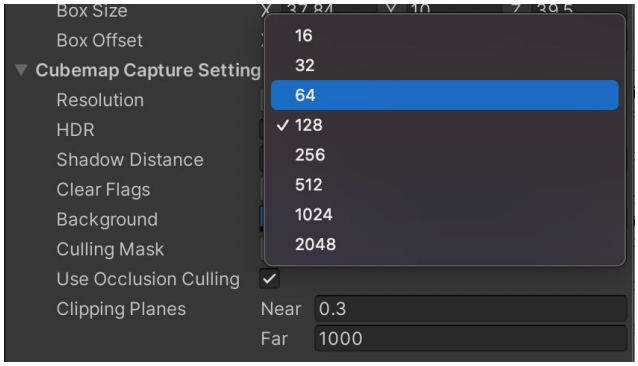
동적 오브젝트 렌더링 시 라이트 프로브를 사용한 효과: 왼쪽은 라이트 프로브 사용, 오른쪽은 미사용

반사 프로브

반사 프로브를 사용하여 씬을 최적화할 수도 있습니다. 반사 프로브는 환경의 일부를 주변에 있는 지오메트리로 투영하여 더 현실적인 반사를 만듭니다. 기본적으로 Unity에서는 스카이박스가 반사 맵으로 사용됩니다. 하지만 하나 이상의 반사 프로브를 사용하면 반사가 주변 환경과 더 밀접하게 이루어집니다.

매끄러운 표면에 반사 프로브를 사용한 효과: 왼쪽은 반사 프로브 사용, 오른쪽은 미사용

반사 프로브를 베이크할 때 생성되는 큐브맵의 크기는 카메라가 반사 오브젝트에 얼마나 가까이 접근하는지에 좌우됩니다. 항상 씬 최적화 요구 사항에 맞는 최소 맵 크기를 사용하시기 바랍니다.



반사 프로브 큐브맵의 크기 조정

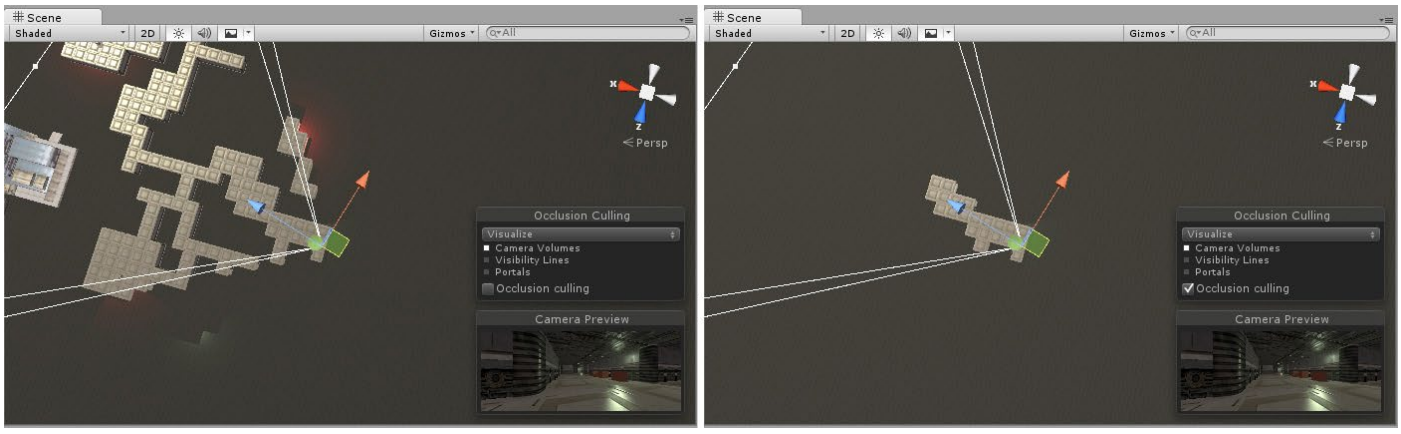
카메라 설정

URP에서는 성능 최적화를 위해 카메라에서 원치 않는 렌더러 프로세스를 비활성화할 수 있습니다. 프로젝트에서 고사양 및 저사양 기기 모두를 타게팅하는 경우에 유용한 방식입니다. 포스트 프로세싱이나 그림자 렌더링, 뎁스 텍스처 등 비용이 많이 드는 프로세스를 비활성화하면 시각적 정확도는 낮아질 수 있으나 저사양 기기에서의 성능이 개선됩니다.

오클루전 컬링

카메라를 최적화하는 또 다른 방법은 오클루전 컬링입니다. 기본적으로 Unity의 카메라는 항상 카메라의 절두체에 있는 모든 것을 드로우하며, 여기에는 벽이나 다른 오브젝트에 의해 숨겨질 수 있는 지오메트리가 포함됩니다. 플레이어가 볼 수 없는 지오메트리는 드로우할 필요가 없으며, 그렇게 할 경우 밀리초 단위의 시간을 허비하게 됩니다. 이런 상황에서 오클루전 컬링을 활용할 수 있습니다.

오클루전 컬링은 오브젝트와 카메라 사이에 또 다른 항목이 표시될 때 엄청난 수의 오브젝트에 마스킹이 적용될 수 있는 씬에 가장 적합합니다. 아래 이미지에서 볼 수 있듯이 복도로 연결되는 미로형 게임은 오클루전 컬링을 사용하기에 좋은 장르입니다.

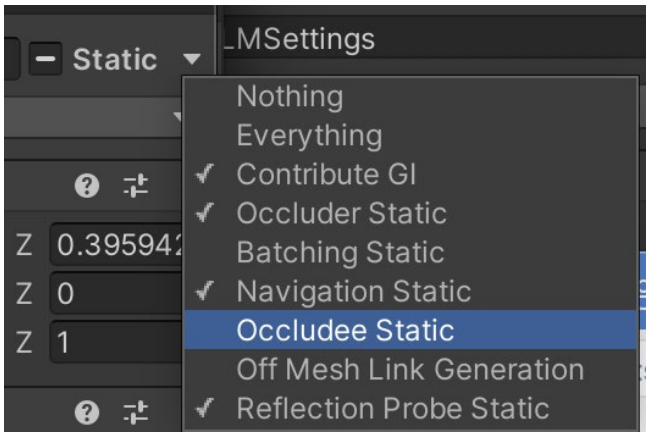


왼쪽 이미지는 절두체 컬링, 오른쪽 이미지는 오클루전 컬링

Unity에서는 오클루전 데이터를 베이크하여 씬에서 차단된 부분을 무시합니다. 프레임당 드로우되는 지오메트리를 줄이면 성능을 크게 증대할 수 있습니다.

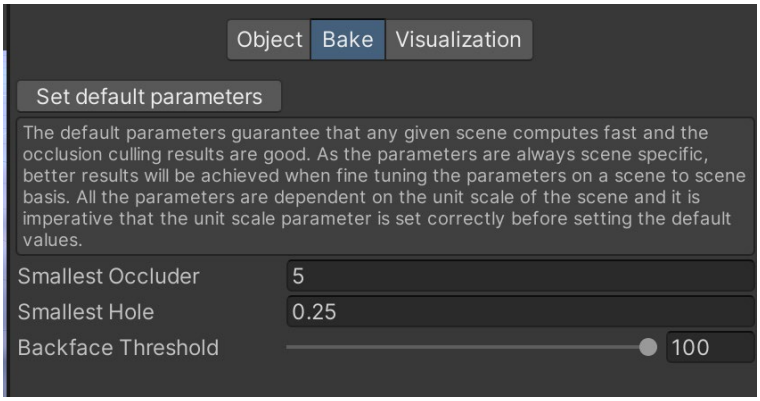
씬에서 오클루전 컬링을 활성화하려면 지오메트리를 **Occluder Static** 또는 **Occludee Static**으로 표시합니다. 오클루더(Occluder)는 중간 크기에서 큰 크기의 오브젝트로서 오클루디(Occludees)로 표시된 오브젝트를 가릴 수 있습니다. 오클루더가 되려면 오브젝트가 불투명하고 Terrain 또는 Mesh Renderer 컴포넌트가 있어야 하며 런타임 시 이동하지 않아야 합니다. 오클루디는 이와 유사하게 런타임 시 이동하지 않는 작고 투명한 오브젝트를 포함하여 Renderer 컴포넌트가 있는 오브젝트면 됩니다.

드롭다운을 사용하여 정적 프로퍼티를 설정합니다.



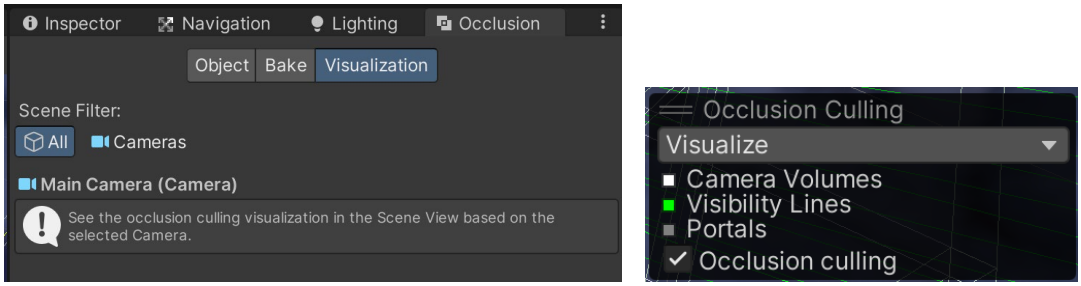
오클루전 데이터에 포함된 오브젝트의 설정

Window > Rendering > Occlusion Culling을 열고 **Bake** 탭을 선택합니다. **인스펙터** 오른쪽 하단 모서리에서 **Bake**를 누릅니다. Unity에서는 오클루전 데이터를 생성하며 프로젝트에서 데이터를 에셋으로 저장하고 현재 씬에 에셋을 연결합니다.



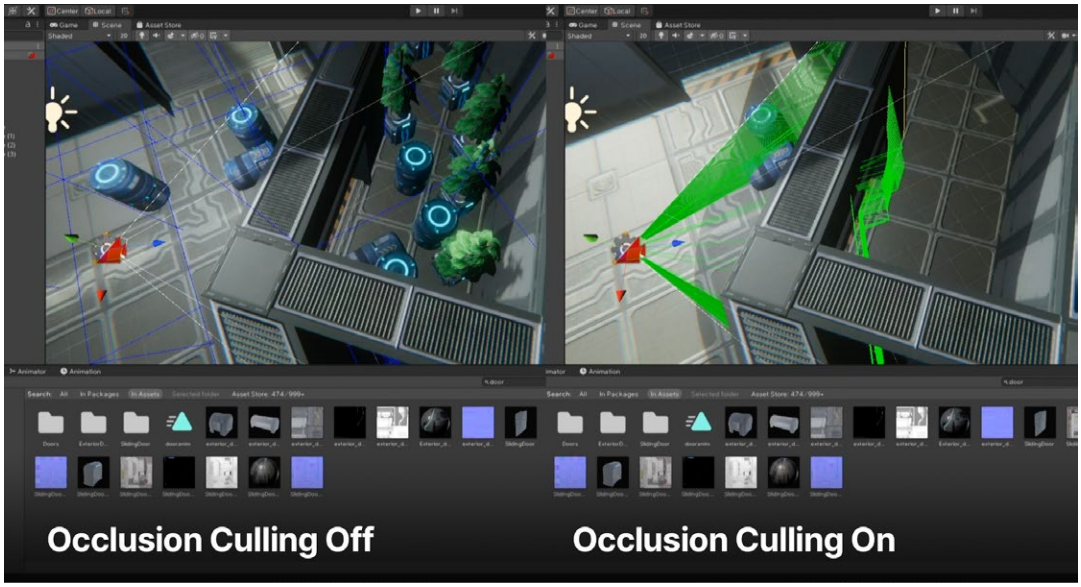
오클루전 컬링의 Bake 탭

Visualization 탭을 사용하여 오클루전 컬링이 실행된 것을 볼 수 있습니다. 씬에서 **카메라**를 선택하고 **씬** 뷰의 **Occlusion Culling** 팝업 창을 사용하여 시각화를 설정합니다. 작은 카메라 뷰 창 뒤에 팝업이 가려질 수 있습니다. 이런 경우 두 줄 아이콘을 오른쪽 클릭하고 **Collapse**를 선택합니다. 팝업을 옮긴 다음 오른쪽 클릭으로 확장하여 카메라 뷰를 복원합니다.



Visualization 탭과 Occlusion Culling 팝업

카메라를 이동함에 따라 오브젝트가 나타났다가 사라집니다.



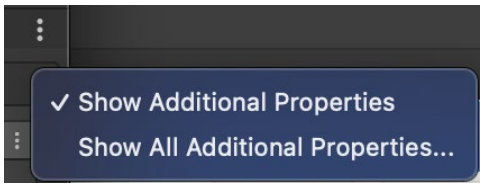
오클루전 컬링이 꺼졌을 때(왼쪽 이미지)와 켜졌을 때(오른쪽 이미지)

Unity 6에서 GPU 상주 드로어를 사용하는 경우, GPU 오클루전 컬링을 사용할 수 있습니다.

파이프라인 설정

URP 에셋 설정을 변경하고 다양한 품질 티어를 사용했을 때 발생하는 영향에 대해서는 이전 섹션에서 다루었습니다. 여기서는 프로젝트에서 최고의 결과를 얻기 위해 품질 티어를 테스트하는 추가 팁을 살펴봅니다.

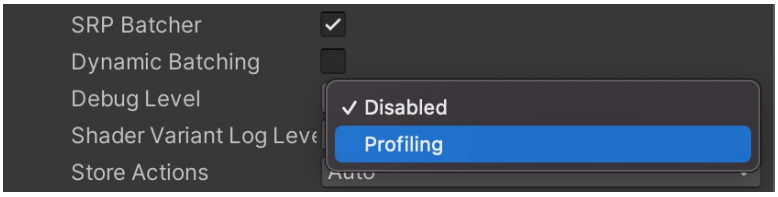
- 성능 향상을 위해 그림자 해상도와 거리를 줄입니다.
- 뎁스 텍스처 및 불투명 텍스처 등 프로젝트에 필요하지 않은 기능을 비활성화합니다.
- 새 배칭 메서드를 사용하기 위해 SRP 배처를 활성화합니다. SRP 배처는 동일한 셰이더 배리언트를 사용하는 메시를 자동으로 함께 배칭하여 드로우 콜을 줄입니다. 씬에 동적 오브젝트가 많은 경우 이 방식을 사용하면 성능을 향상할 수 있습니다. SRP Batcher 체크박스가 표시되지 않는 경우 세로 줄임표 아이콘(⋮)을 클릭하고 **Show Additional Properties**를 선택합니다.



URP 에셋 인스펙터의 추가 프로퍼티 활성화

프레임 디버거

프레임 디버거를 사용하면 렌더링 중에 일어나는 상황을 더 잘 이해할 수 있습니다. 프레임 디버거 창에서 추가 정보를 보려면 **URP Asset**을 사용하여 **Debug Level**을 조정합니다. SRP Batcher 체크박스처럼 이는 **Show Additional Properties**가 활성화되어 있을 때만 인스펙터에 표시됩니다.

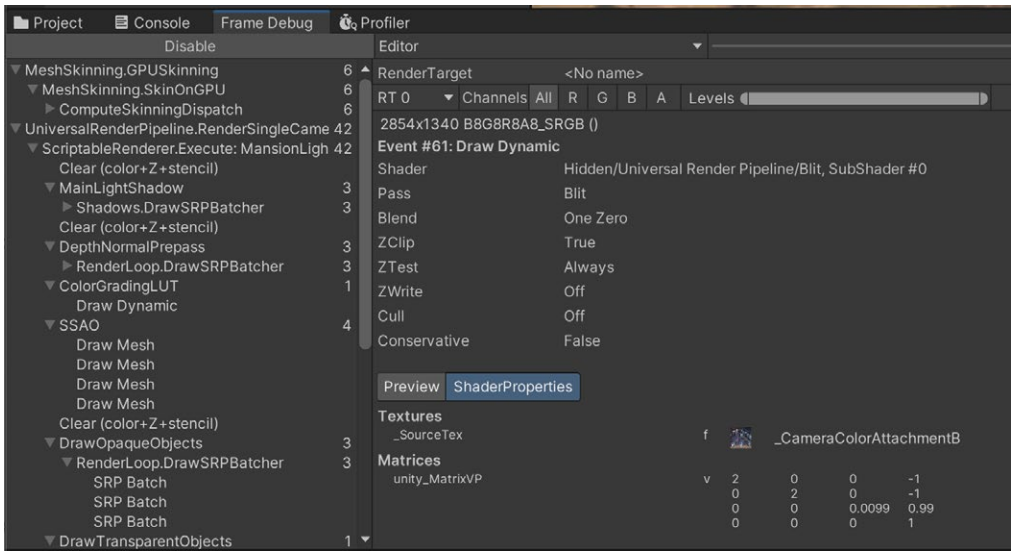


Debug Level 설정

Debug Level을 조정하면 성능에 영향을 줄 수 있습니다. 프레임 디버거를 사용하지 않을 때는 Debug Level을 항상 꺼 두시기 바랍니다.

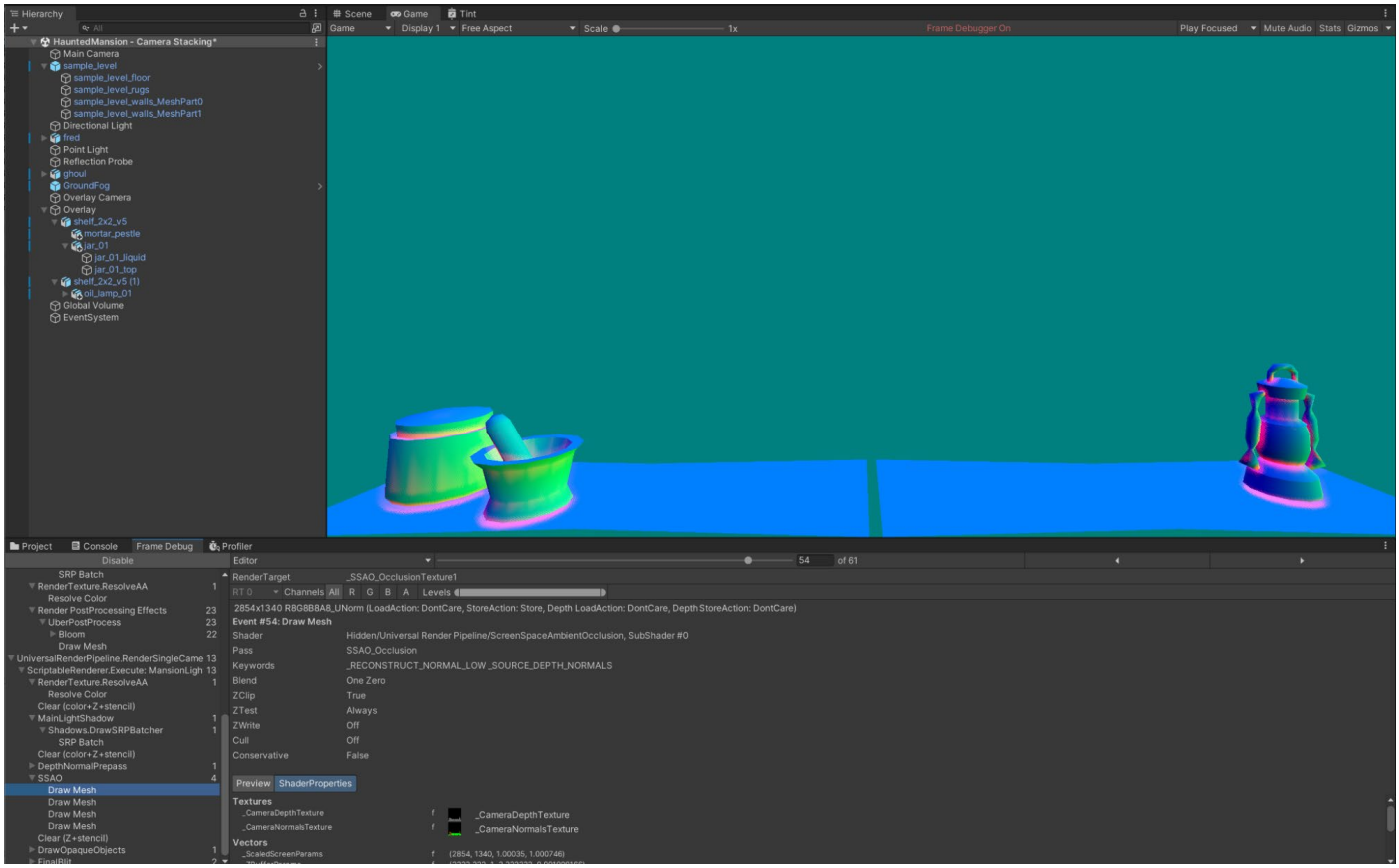
프레임 디버거에는 최종 이미지를 렌더링하기 전에 만들어진 모든 드로우 콜의 목록이 표시되므로, 특정 프레임의 렌더링에 시간이 오래 걸리는 이유를 정확히 파악할 수 있습니다. 또한 씬의 드로우 콜 수가 많은 이유를 확인할 수 있습니다.

Window > Analysis > Frame Debugger로 이동하여 프레임 디버거를 엽니다. 게임이 플레이되고 있다면 **Enable** 버튼을 선택합니다. 그러면 게임이 일시 중지되어 드로우 콜을 살펴볼 수 있습니다.



프레임 디버거 세부 사항

렌더 파이프라인(왼쪽 창)에서 스테이지를 클릭하면 이 스테이지의 프리뷰가 **게임** 뷰에 표시됩니다.



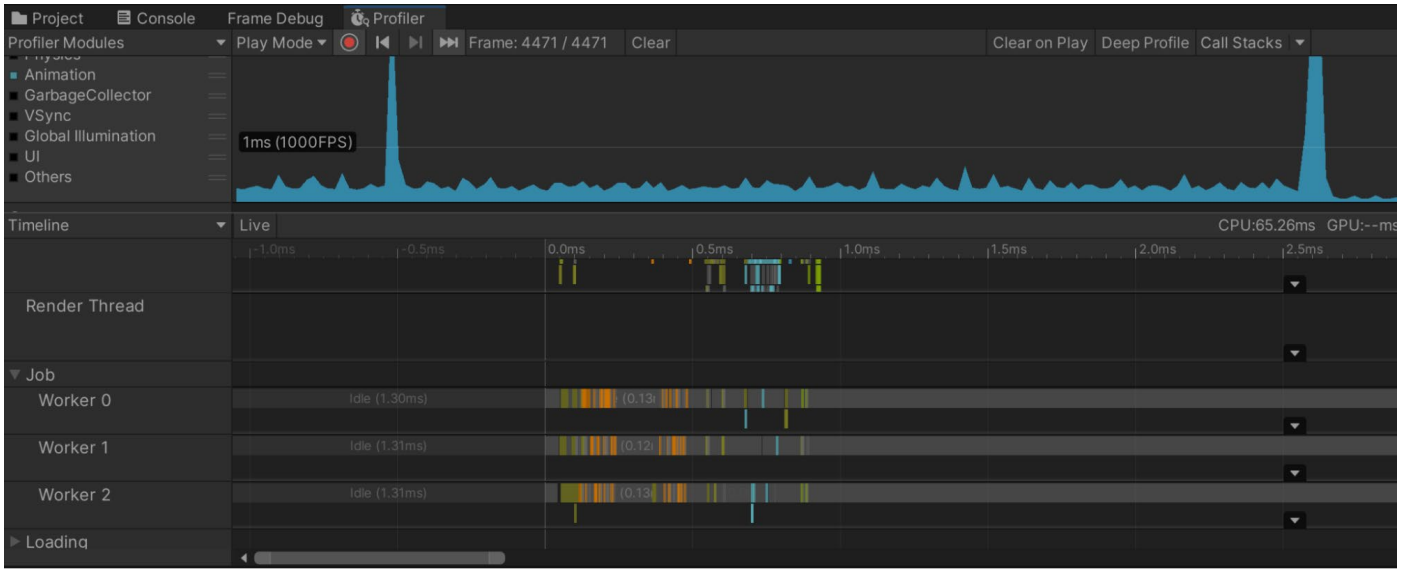
프레임 디버거에서 렌더링 프로세스의 모든 단계를 게임 뷰에 표시(여기서는 SSAO 생성 단계)

Unity 프로파일러

프레임 디버거와 마찬가지로 프로파일러는 프로젝트에서 프레임 사이클을 완료하는 데 걸리는 시간을 파악하기에 유용한 방법이며, 렌더링, 메모리, 스크립팅에 대한 개요를 제공합니다. 완료하는 데 오랜 시간이 소요되는 스크립트를 식별하여 코드에서 잠재적인 병목 지점을 정확히 파악할 수 있습니다.

Window > Analysis > Profiler로 이동하여 프로파일러를 엽니다. **Play Mode**인 경우 게임의 전반적인 성능에 대한 개요가 창에 제공됩니다. 실시간 뷰를 일시 중지하고 **Hierarchy Mode**를 사용하여 단일 프레임을 완료하는 데 걸린 시간의 요약 정보를 볼 수 있습니다. 프레임 중에 Unity에서 생성한 각 콜이 프로파일러에 표시됩니다.

더 상세한 분석이 필요하면 하위 수준 네이티브 플러그인 프로파일러 API를 사용하세요. 이 프로파일러 API를 사용하여 프로파일러를 확장하거나, 네이티브 플러그인 코드의 성능을 프로파일링하거나, Sony Playstation의 Razor, Microsoft의 PIX(Windows 및 Xbox) 그리고 Chrome Tracing, ETW, ITT, VTune, Telemetry 등의 타사 프로파일링 툴로 전송할 프로파일링 데이터를 준비할 수 있습니다.



하위 수준 네이티브 플러그인 프로파일러 API를 사용하는 프로파일러 창

다음은 로우레벨 네이티브 플러그인 프로파일러 API의 사용 예시입니다.

```
#include <IUnityInterface.h>
#include <IUnityProfiler.h>
static IUnityProfiler* s_UnityProfiler = NULL;
static const UnityProfilerMarkerDesc* s_MyPluginMarker = NULL;
static bool s_IsDevelopmentBuild = false;
static void MyPluginWorkMethod()
{
	if (s_IsDevelopmentBuild)
	s_UnityProfiler->BeginSample(s_MyPluginMarker);
	// Unity 프로파일러에서 “MyPluginMethod”로 보고 싶은 코드.
	// ...
	if (s_IsDevelopmentBuild)
	s_UnityProfiler->EndSample(s_MyPluginMarker);
}
extern “C” void UNITY_INTERFACE_EXPORT UNITY_INTERFACE_API UnityPluginLoad(IUnityInterfaces*
unityInterfaces)
{
	s_UnityProfiler = unityInterfaces->Get<IUnityProfiler>();
	if (s_UnityProfiler == NULL)
	return;
	s_IsDevelopmentBuild = s_UnityProfiler->IsAvailable() != 0;
```

```
        s_UnityProfiler->CreateMarker(&s_MyPluginMarker,
        “MyPluginMethod”, kUnityProfilerCategoryOther,
        kUnityProfilerMarkerFlagDefault, 0);
}
extern “C” void UNITY_INTERFACE_EXPORT UNITY_INTERFACE_API UnityPluginUnload()
{
        s_UnityProfiler  = NULL;
}
```

추가 리소스

Unity에서 고급 프로파일링 기술을 습득하는 데 관심이 있다면 우선 Unity 게임 프로파일링 완벽 가이드 전자책을 무료로 다운로드하여 참조해 보시기 바랍니다. 이 가이드에서는 Unity에서 애플리케이션을 프로파일링하고, 메모리를 관리하고, 전력 소비를 최적화하는 방법에 관한 조언과 지식을 처음부터 끝까지 소개합니다.

다른 유용한 리소스로는 Catlike Coding의 성능 측정(영문), The Gamedev Guru의 Unity 드로우 콜 배칭(영문)이 있습니다.

URP 3D 샘플

URP 3D 샘플은 Unity Hub에서 다운로드할 수 있으며 아트 스타일, 렌더링 경로, 씬 복잡도가 각기 다른 4가지 환경을 통해 URP로 제작된 다양한 3D 프로젝트를 살펴볼 수 있습니다.

URP 3D 샘플은 지난 몇 년 동안 URP를 사용해 온 여러 개발자에게 친숙할 만한 건설 현장 씬을 대체합니다. 이 신규 샘플 프로젝트에는 URP의 기능을 보여 주는 여러 미니 씬이 포함되어 있습니다.

각 씬을 살펴보겠습니다.

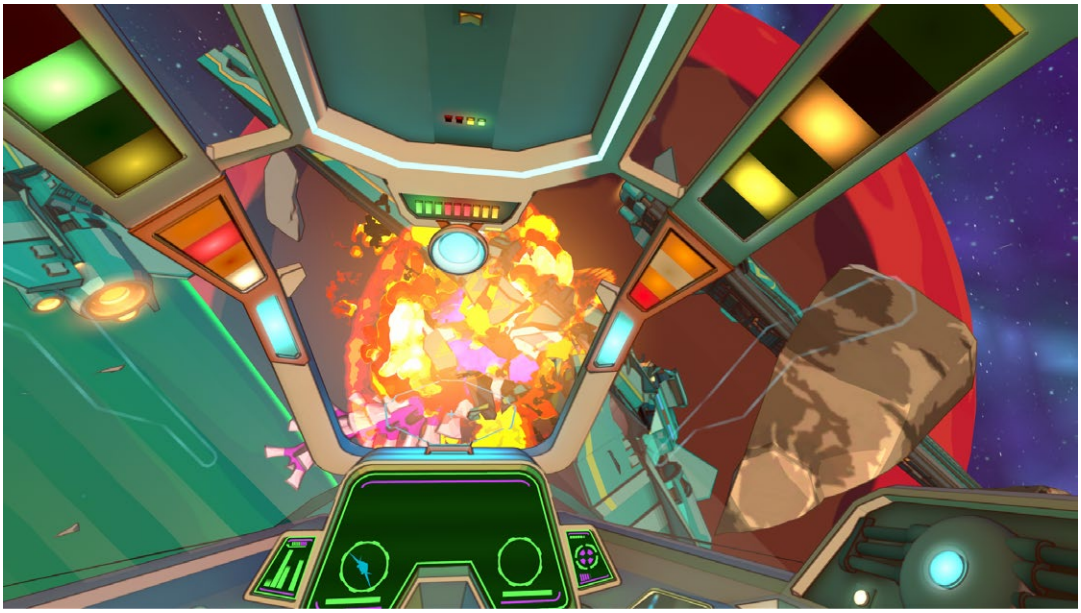
정원

이 씬은 URP로 모바일 및 콘솔에서부터 고사양 게이밍 데스크톱까지 다양한 플랫폼에 맞춰 콘텐츠를 효율적으로 스케일링하는 방법을 보여 줍니다. 스타일라이즈드 PBR 렌더링, 커스터마이즈할 수 있는 초목, 그리고 새로운 포워드+ 렌더러로 기존의 광원 수 제한을 뛰어넘는 수많은 광원 렌더링을 선보입니다.

오아시스

오아시스는 뛰어난 디테일의 텍스처, VFX Graph 효과, SpeedTree, 커스텀 물 솔루션을 선보이는 고도로 사실적인 씬입니다. 컴퓨트 셰이더를 지원하는 기기를 타게팅합니다.

조종석



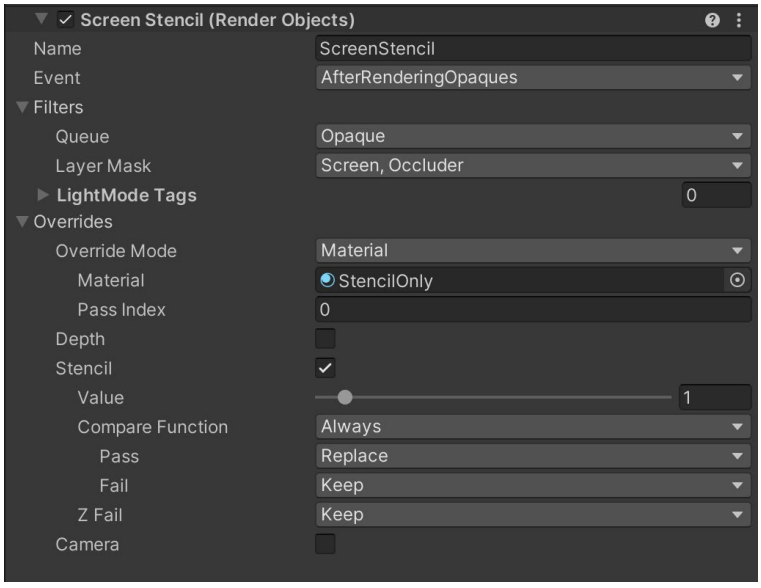
이 씬은 Shader Graph와 커스텀 조명 코드를 사용합니다. Meta Quest 2와 같은 무선 VR 기기를 타겟으로 제작되었습니다.

터미널

이 씬은 다른 샘플 씬을 연결하는 공간으로, 한 씬에서 다른 씬으로 이동하기 위한 전환 효과를 제공합니다.

씬 간 이동

이 샘플 프로젝트는 씬 간의 이동을 위해 전환 효과를 사용합니다. 전환 효과는 전환이 완료되기 전에 화면 밖에 있는 렌더 타겟을 사용해서 들어오는 씬을 렌더링합니다. 이후 들어오는 씬이 Shader Graph로 생성된 커스텀 셰이더를 통해 나가는 씬에 배치된 대형 모니터에 렌더링되며, 렌더 오브젝트 렌더러 기능을 통해 스텐실로 전체 화면 스왑이 처리됩니다.



Screen Stencil 렌더러 기능

효과를 확인하려면 Unity 로고가 표시될 때까지 받침대를 향해 걸어간 후, 로고가 화면 중앙에 있는 상태를 유지합니다. 이렇게 하면 전환이 트리거됩니다.

로드할 때 모든 씬 에셋이 로드되지만 하나의 씬만 활성화됩니다. 터미널 씬에서 시작할 때 런타임 시 사용되는 카메라는 FPS_Controller 게임 오브젝트의 카메라와 동일합니다. **MainCamera**가 활성 씬을 렌더링하며, **ScreenCamera**가 모니터에 표시되는 씬을 렌더링합니다.

터미널 씬의 FPS_Controller

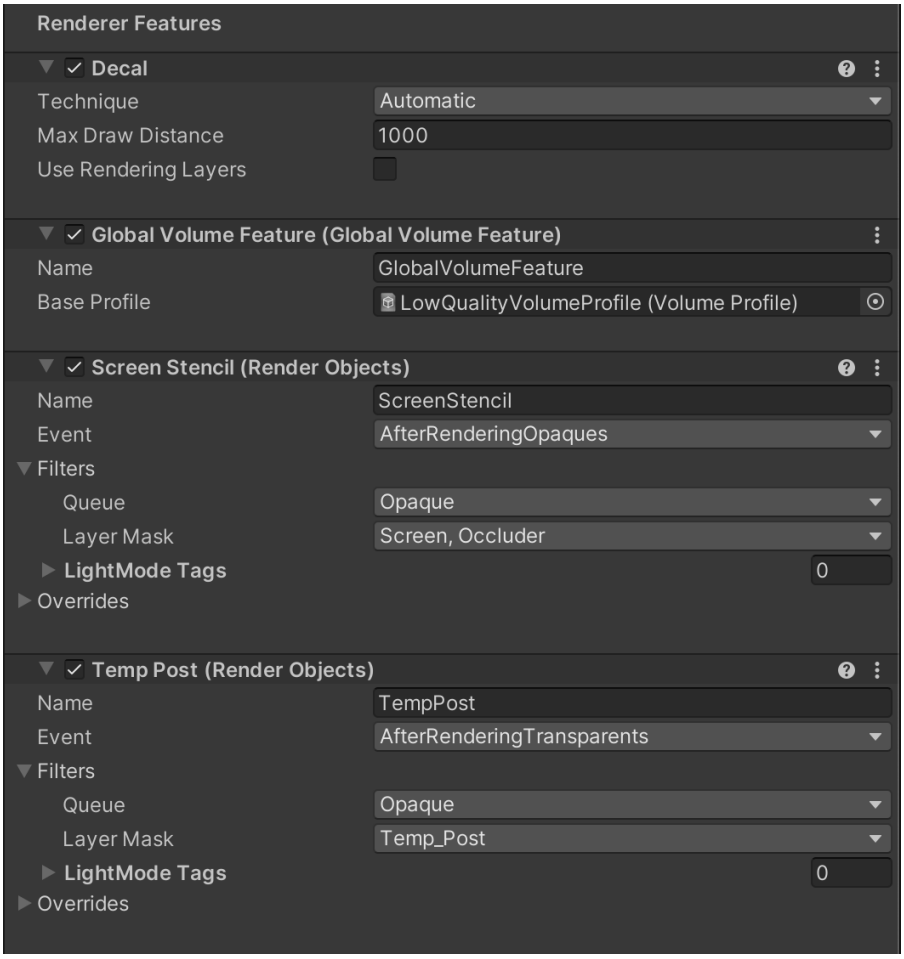
전환 시 들어오는 씬의 카메라가 렌더 타겟에 렌더링됩니다. URP는 하나의 메인 방향 광원만 지원하므로 여기에서 문제가 발생할 수 있습니다. 렌더링 전에 **Scripts > SceneManagement > SceneTransitionManager.cs** 스크립트가 실행되므로, 활성 씬의 메인 광원을 활성화하고 나머지는 비활성화하여 이러한 제한 사항을 준수할 수 있습니다.

아래의 스크립트를 살펴보세요. **OnBeginCameraRendering** 메서드에서 먼저 메인 카메라를 렌더링하고 있는지 확인합니다. **isMainCamera**가 true인 경우, **ToggleMainLight** 호출이 **currentScene**의 메인 방향 광원을 활성화하고 들어오는 씬(**screenScene**)의 메인 방향 광원을 비활성화합니다. **isMainCamera**가 false인 경우에는 그 반대로 처리됩니다.

동일한 스크립트에서 RenderSettings 오브젝트의 설정을 조정하여 렌더링되는 씬에 맞춰 안개, 반사, 스카이박스를 전환합니다.

```
179         /// <summary>
180         /// This function is called per camera by the render pipeline.
181         /// We use it to set up light and render settings (skybox etc) for the different scenes as they are displayed
182         /// </summary>
183         void OnBeginCameraRendering(ScriptableRenderContext context, Camera camera)
184         {
185             bool isMainCamera = camera.CompareTag("MainCamera");
186 
187             if (!isMainCamera && screenScene == null)
188             {
189                 //If no screen scene is loaded, no setup needs to be done for it
190                 return;
191             }
192 
193             //Toggle main light
194             ToggleMainLight(currentScene, isMainCamera);
195             ToggleMainLight(screenScene, !isMainCamera);
196 
197             //Setup render settings
198             SceneMetaData sceneToRender = isMainCamera ? currentScene : screenScene;
199             RenderSettings.fog = sceneToRender.FogEnabled;
200             RenderSettings.skybox = sceneToRender.skybox;
201             if (sceneToRender.reflection != null)
202             {
203                 RenderSettings.customReflectionTexture = sceneToRender.reflection;
204             }
205 
206             if (!isMainCamera && camera.cameraType == CameraType.Game)
207             {
208                 camera.GetComponent<OffsetCamera>().UpdateWithOffset();
209             }
210         }
211 
212         private void ToggleMainLight(SceneMetaData scene, bool value)
213         {
214             if (scene != null && scene.mainLight != null)
215             {
216                 scene.mainLight.SetActive(value);
217             }
218         }
```

렌더 오브젝트 렌더러 기능을 통해 들어오는 씬과 나가는 씬 간의 전환이 처리됩니다. 스텐실 버퍼에 값을 작성하여 후속 패스에서 이를 확인할 수 있습니다. 렌더링되는 픽셀에 특정 스텐실 값이 있다면 색상 버퍼의 값을 유지하고, 값이 없으면 자유롭게 덮어쓸 수 있습니다. **렌더러 기능**은 고도로 유연한 방식으로 패스의 조합을 통해 최종 렌더링을 구현합니다.



모바일 포워드+ 렌더러의 렌더러 기능

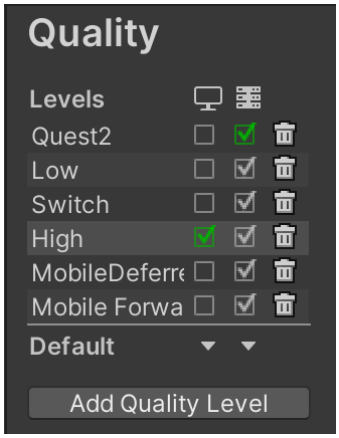
전환 중에 카메라 위치를 일치시키기 위해 프로젝트에 각 씬의 오프셋 트랜스폼을 저장하는 **SceneMetaData** 스크립트가 있으며, 전환 중에 들어오는 씬과 나가는 씬을 처리하는 **SceneTransitionManager** 스크립트가 있습니다. **Update** 메서드가 전환의 진행 현황을 추적합니다. **ElapsedTimeInTransition**이 **m_TransitionTime**보다 크면 **TriggerTeleport**가 호출되며, 이를 통해 **Teleport** 메서드가 호출됩니다. 여기에서 나가는 씬과 들어오는 씬 간의 부드러운 전환을 위해 플레이어의 위치와 방향을 조정합니다.

```
120     void Update()
121     {
122         float t = m_OverrideTransition ? m_ManualTransition : ElapsedTimeInTransition / m_TransitionTime;
123
124         if (InTransition)
125         {
126             ElapsedTimeInTransition += Time.deltaTime;
127
128             if (ElapsedTimeInTransition > m_TransitionTime)
129             {
130                 TriggerTeleport();
131             }
132
133             ElapsedTimeInTransition = Mathf.Min(m_TransitionTime, ElapsedTimeInTransition);
134         }
135         else
136         {
137             ElapsedTimeInTransition -= Time.deltaTime * 3;
138
139             if (ElapsedTimeInTransition < 0 && CoolingOff)
140             {
141                 CoolingOff = false;
142             }
143
144             ElapsedTimeInTransition = Mathf.Max(0, ElapsedTimeInTransition);
145         }
146
147         //Update weights of post processing volumes
148         if (m_Loader != null && !CoolingOff)
149         {
150             float tSquared = t * t;
151             m_Loader.SetVolumeWeights(1 - tSquared);
152         }
153
154         Shader.SetGlobalFloat(m_TransitionAmountShaderProperty, t);
155     }
```

ScreenTransitionManager.cs Update 메서드

확장성

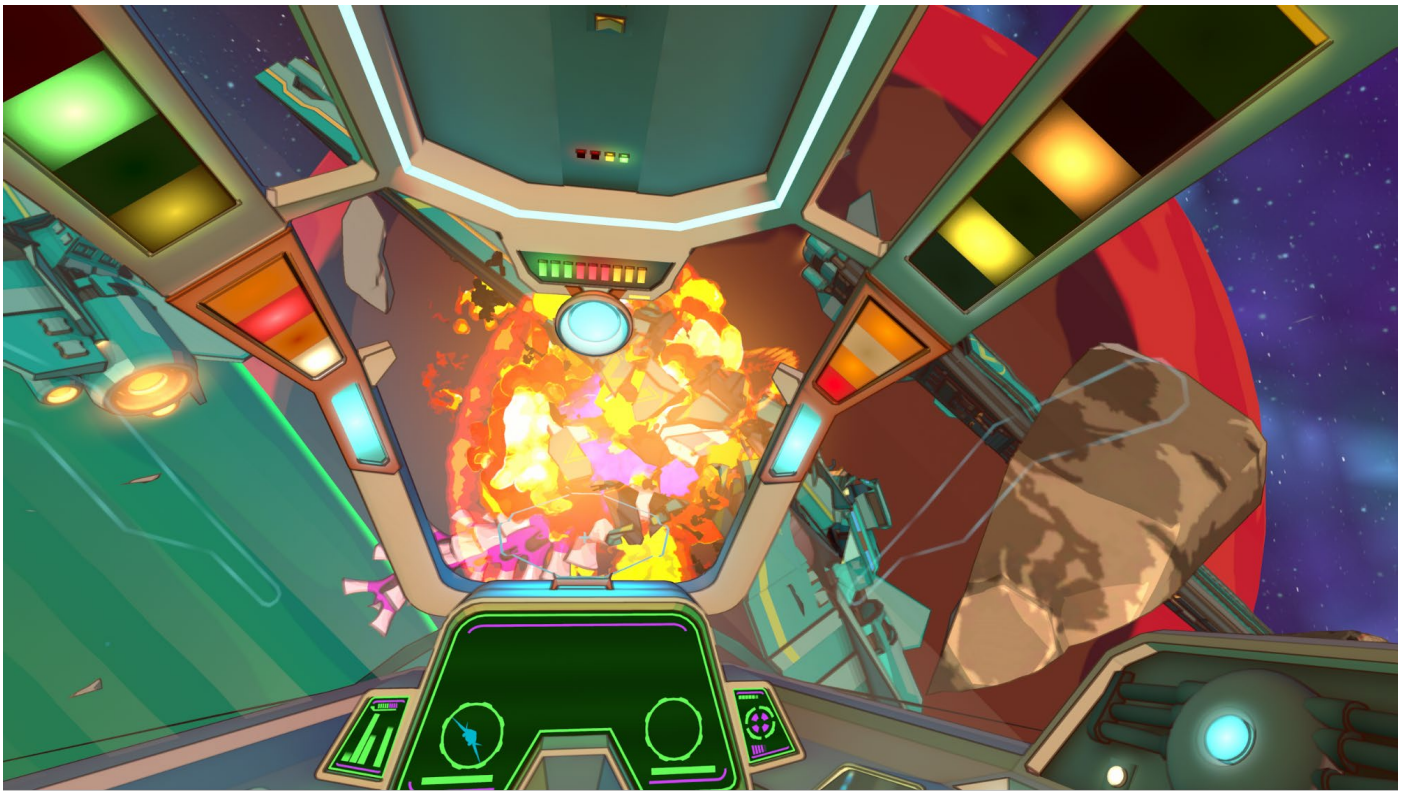
URP는 다양한 하드웨어를 지원하며, 새로운 샘플 씬을 사용하면 여러 기기에서의 작동 방식을 살펴볼 수 있습니다. **Project Settings > Quality**에서 다양한 옵션을 확인할 수 있습니다.



품질 수준

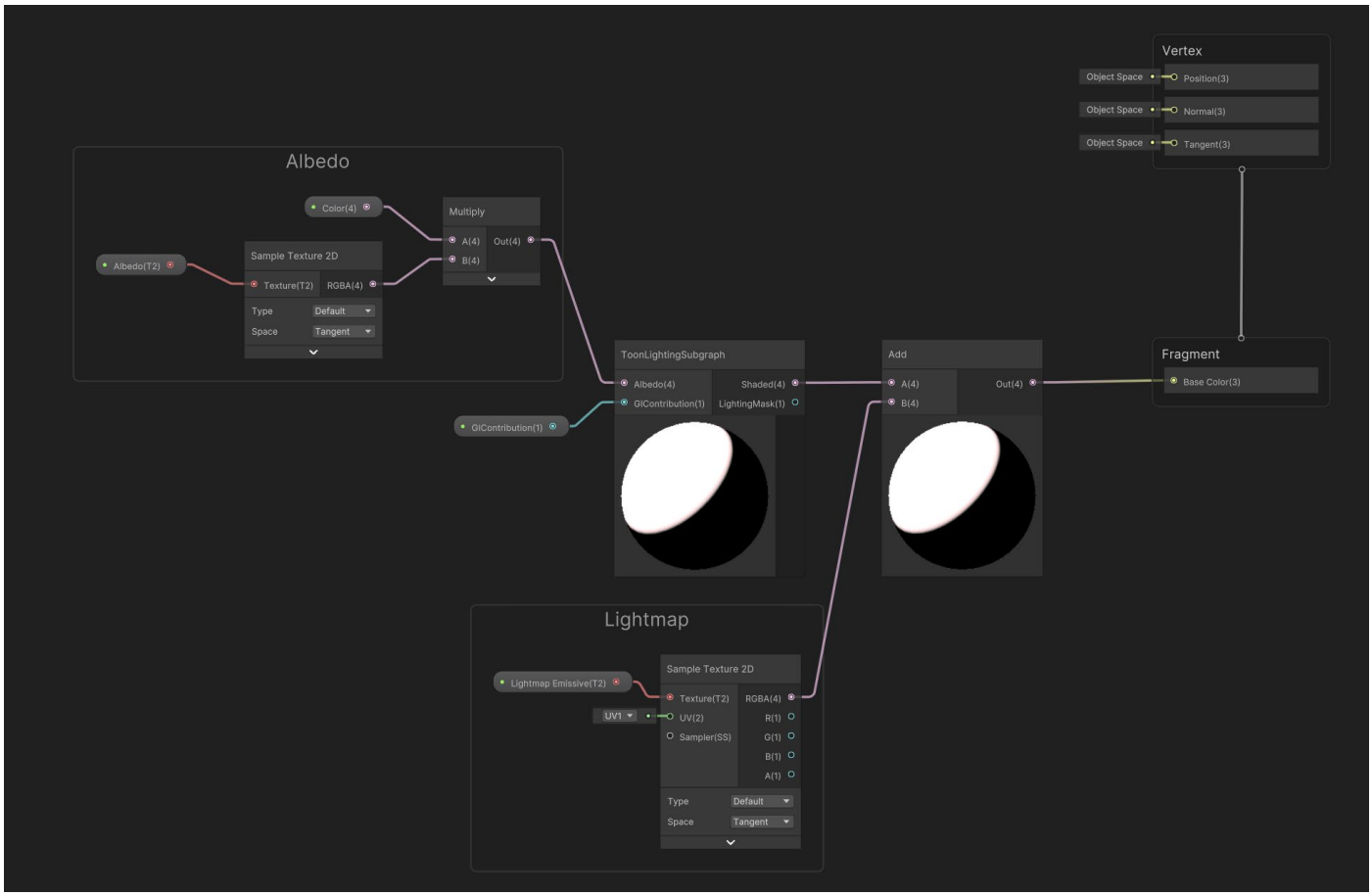
각 옵션은 서로 다른 **렌더 파이프라인 에셋**을 사용합니다. Quality 섹션에서 설명한 것처럼 URP는 Quality 패널과 렌더 파이프라인 에셋의 설정을 조합하여 품질을 처리합니다.

스탠드얼론 VR 헤드셋에 실시간 3D 그래픽스를 표시하는 일은 굉장히 어렵습니다. 고해상도 화면에 양쪽 눈이 개별적으로 처리되어야 하므로 렌더링되는 프레임마다 두 배의 작업이 필요합니다. 게다가 72FPS 이상을 타겟으로 하면 필요한 초당 픽셀 수가 더 많아집니다. 스타일라이즈드 조명을 사용하면 이 문제를 해결할 수 있습니다. 아래의 조종석 씬은 툰 셰이더(Toon Shaded) 조명 모델을 사용합니다.



URP 3D 샘플의 조종석 환경

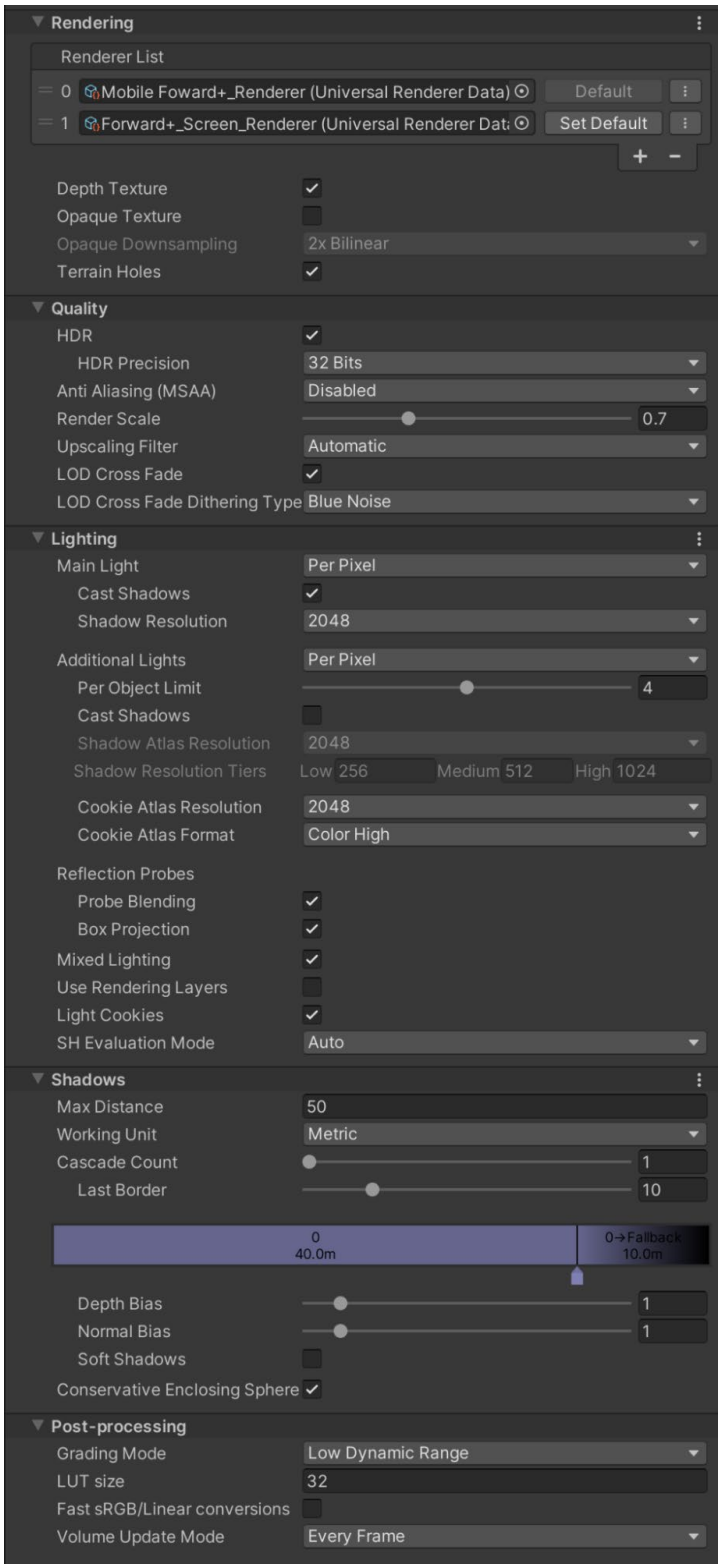
최소한의 코드 작성만으로 Shader Graph를 통해 커스텀 조명이 처리됩니다.



툰 셰이더에서 기존에 하던 방식대로, 내적으로 노멀 벡터와 메인 광원 방향을 결합하여 조명 수준을 결정합니다. 그런 다음 값을 부드럽게 변경하는 대신 램프를 사용하여 단계별 광원 수준을 설정합니다. 또한 조종석 씬에 사용된 조명 모델은 계산에 베이크된 전역 조명을 사용하며, 에지 감지를 통해 미세한 아웃라인 효과를 더합니다. 커스텀 조명은 Shader Graph로 처리됩니다.

툰 셰이더 제작 튜토리얼이 필요하다면 유니버설 렌더 파이프라인 가이드(영문)를 참고하세요.

모바일 기기에서 샘플 프로젝트 실행



Mobile Forward+ URP 에셋

게임 개발자들은 종종 모바일 기기에서 게임을 원활하게 실행하는 데 문제를 겪습니다. 새로운 샘플 프로젝트의 **Settings** 폴더에 **Mobile Forward+** URP 에셋이 포함되어 있습니다. URP 에셋이 품질 설정을 조정하는 기본이라는 점을 기억해 두세요. 포워드+는 CPU가 수행하는 막대한 양의 프레임별 컬링 연산에 의존하므로 저사양 모바일 기기에 가장 적합한 옵션은 아닙니다. 이러한 기기에 가장 적합한 옵션은 샘플 프로젝트의 URP 에셋이 사용하는 디퍼드 렌더러입니다.

왼쪽 이미지에서 Mobile Forward+ 에셋의 설정을 확인할 수 있습니다.

Renderer List에는 두 가지 유니버설 렌더러 데이터 에셋이 있는데, 각각 활성 씬의 **Mobile Forward+_Renderer**와 화면 씬을 렌더링하는 **Forward+_Screen_Renderer**입니다. Depth Texture는 활성화되어 있습니다. Additional Lights의 Cast Shadows가 비활성화되어 있다는 점에 유의하세요. 이 옵션은 큰 리소스를 소모하며, 모바일 기기에서는 보통 광원 쿠키를 사용하여 모방할 수 있습니다. 특히 정원 씬에 상당히 많은 수의 광원이 있으며, 이 중 다수가 쿠키를 사용하여 그림자를 표현합니다. 아래 이미지의 좌측 하단에 있는 바위에서 쿠키를 사용할 때와 사용하지 않을 때의 조명 차이를 확인해 보세요.

쿠키를 사용할 때와 사용하지 않을 때 정원 환경의 점 광원

모바일 플랫폼을 타게팅할 때 세 가지 팁은 다음과 같습니다.

- 렌더링되는 픽셀의 수를 줄입니다. 대부분의 최신 모바일 기기는 DPI(인치당 도트 수)가 높습니다. 대부분의 게임에서는 96DPI면 충분합니다. 예를 들어 Screen.DPI가 300인 경우, 2400x1200 화면에서 렌더링 스케일을 96/300으로 설정하면 768x384픽셀으로 렌더링하여 픽셀 수가 거의 10분의 1로 줄어들기 때문에 성능이 크게 향상됩니다. 렌더링 스케일은 URP 에셋에서 설정하거나 런타임 시에 값을 조정할 수 있습니다.
- Mobile Forward+_Renderer 에셋의 데칼 렌더러 기능에서 Technique 옵션은 Automatic으로 설정되어 있습니다. GPU에서는 이 옵션이 Screen Space로 전환되고 숨겨진 표면을 제거합니다. 이를 통해 이러한 기기에서 리소스가 낭비되는 뎁스 프리패스를 방지하여 성능이 향상됩니다.
- 포워드+의 CPU 오버헤드 비용이 너무 높은 기기에서는 디퍼드 렌더링을 사용합니다.
- 렌더 그래프 시스템에서 더 적극적으로 렌더 패스를 병합할 때 고려할 만한 내용은 다음과 같습니다.
    - 프로젝트에서 필요하지 않다면 URP 에셋 설정에서 Depth Texture와 Opaque Texture 설정을 비활성화합니다.
    - Opaque Texture를 사용한다면 Opaque Downsampling을 'None'으로 설정합니다. 불투명 텍스처를 다운샘플링하면 URP 임시 텍스처(intermediate texture)의 해상도가 변해 패스가 병합되지 않습니다.

— Depth Texture를 사용한다면 Depth Texture Mode를 ‘After Transparents’로 설정합니다. 이렇게 하면 메인 렌더 패스(Opaque, Sky, Transparents) 사이에 CopyDepth 패스가 삽입되지 않으며, 렌더 그래프가 해당 패스를 성공적으로 병합할 수 있습니다.

이 4개 씬과 URP 에셋 설정 및 기술 자료를 주의 깊게 살펴보면 프로젝트에서 디스플레이에 이러한 기법을 사용하는 방법을 익힐 수 있습니다.

맺는말

URP로 전환하려는 개발자 및 아티스트는 전체 Unity 기술 자료, Unity Learn, Unity 블로그, 토론 페이지를 참고해 보시기 바랍니다.

Unity Product Board에서는 향후 출시될 기능에 더해 현재 개발 중인 URP 기능의 개요를 볼 수 있습니다. 기능을 직접 요청할 수도 있습니다.

게임 개발에서 좋은 성과가 있기를 기원합니다.

unity.com/kr