전자책

Unity에서 콘솔 및 PC용 게임 성능 최적화

Unity®

Contents

들어가는 말 8

프로파일링 9

개발 초기부터 자주 타겟 기기에서 프로파일링하기 9

올바른 영역 최적화 10

Unity 프로파일러의 작동 방식 이해 11

딥 프로파일링 15

Profile Analyzer 사용하기 15

FPS: 정확하지 않을 수 있는 지표 17

GPU 바운드 또는 CPU 바운드 여부 판단 18

네이티브 프로파일링 및 디버깅 툴 사용 19

네이티브 프로파일링 툴 19

GPU 디버깅 및 프로파일링 툴 20

Project Auditor 20

메모리 21

Memory Profiler 사용하기 22

GC(가비지 컬렉션)의 영향 줄이기 22

가능한 경우 가비지 컬렉션 시간 측정하기 23

점진적 가비지 컬렉터를 활용하여 GC 워크로드 분할하기 24

에셋 25

텍스처 압축 25

텍스처 임포트 설정 26

텍스처 아틀라싱 27

폴리곤 개수 확인 28

메시 임포트 설정 30

기타 메시 최적화 30

에셋 감사 31
The AssetPostprocessor 31
UnityDataTools 31
비동기 텍스처 버퍼 31
밉맵 및 텍스처 스트리밍 32
어드레서블 사용 33
프로그래밍 및 코드 아키텍처 36
Unity PlayerLoop 이해하기 36
매 프레임에 실행되는 코드 최소화하기 38
Start/Awake에서 대규모 로직 사용 방지 38
많은 리소스를 소모하는 함수 결과 캐싱 39
빈 Unity 이벤트 함수 방지 40
커스텀 Update Manager 만들기 40
Debug Log 구문 제거하기 41
Stack Trace 로깅 비활성화 42
문자열 파라미터 대신 해시 값 사용하기 42
올바른 데이터 구조 선택 43
런타임 시 컴포넌트 추가 방지 43
오브젝트 풀 사용하기 43
트랜스폼 한 번에 이동 44
ScriptableObject 사용하기 45
람다식 사용 지양 46
C# 잡 시스템 46
버스트 컴파일러 48
프로젝트 구성 50
불필요한 플레이어 설정 또는 품질 설정 비활성화 50
IL2CPP로 전환 50
대규모 계층 구조 사용 지양 51

그래픽스 52

렌더 파이프라인 활용 52

약 90%의 PC/콘솔용 Unity 게임에서 사용되는 SRP 53

콘솔용 렌더 파이프라인 패키지 55

렌더링 경로 선택 56

포워드 57

포워드+ 57

디퍼드 셰이딩 58

Shader Graph 최적화 59

빌트인 셰이더 설정 제거 61

셰이더 배리언트 스트리핑 61

파티클 시뮬레이션: 파티클 시스템 또는 VFX Graph 63

안티앨리어싱으로 다듬기 64

시공간 포스트 프로세싱 66

일반적인 조명 최적화 67

라이트맵 베이크 67

반사 프로브 최소화 68

적응적 프로브 볼륨 68

그림자 비활성화 70

셰이더 효과 대체 70

광원 레이어 사용 71

GPU 라이트매퍼 71

GPU 최적화 72

GPU 벤치마킹 72

렌더링 통계 확인 73

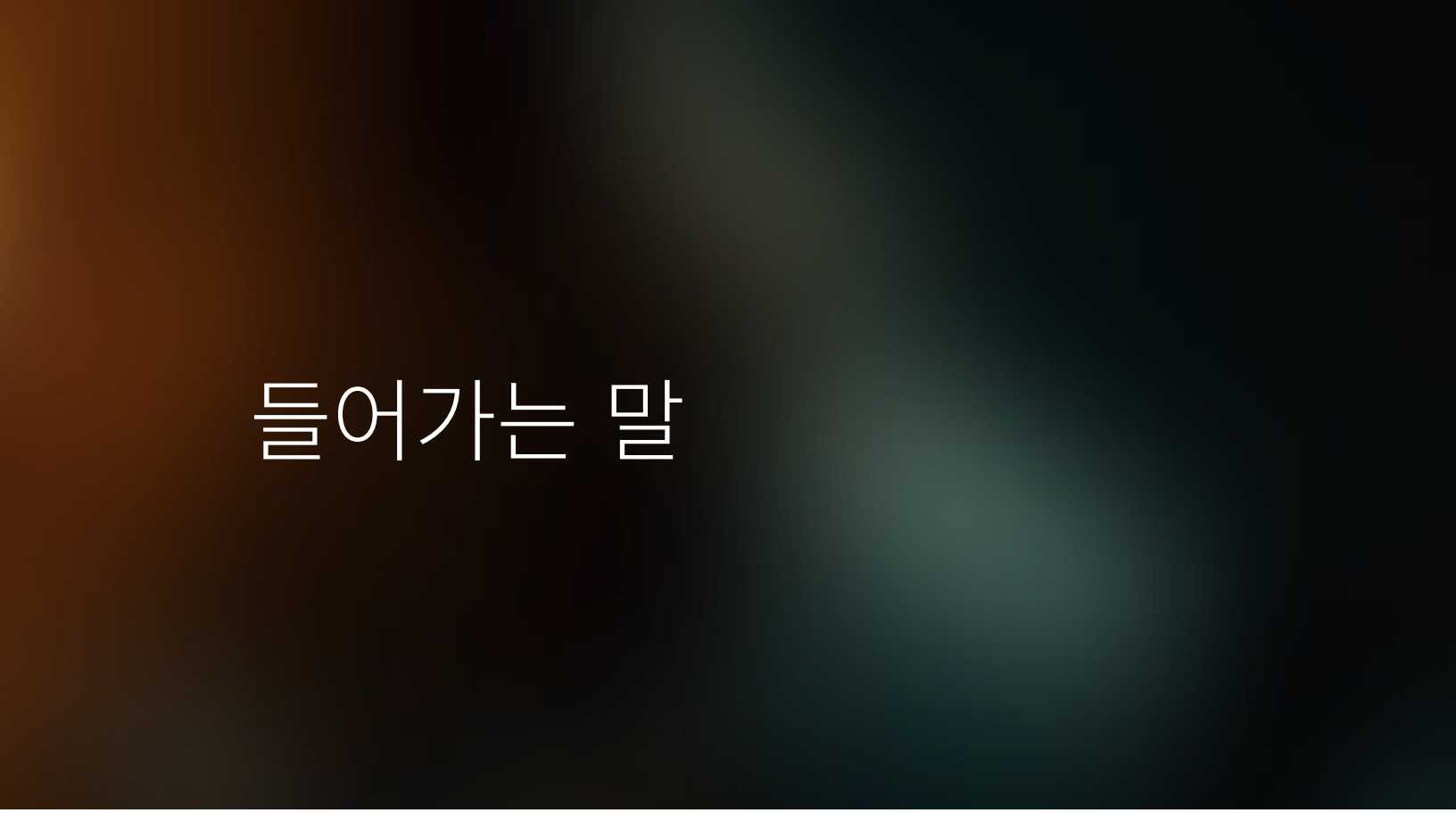
드로우 콜 배칭 사용 74

프레임 디버거 확인 76

필 레이트 최적화 및 오버드로우 줄이기 77
드로잉 순서 및 렌더링 대기열 77
콘솔을 위한 그래픽스 최적화 81
성능 병목 지점 식별 81
배치 수 줄이기 82
포스트 프로세싱 프로파일링 82
테셀레이션 셰이더 사용 지양 82
지오메트리 셰이더를 컴퓨트 셰이더로 교체 82
웨이브프론트 점유율 개선 83
HDRP 빌트인 패스와 커스텀 패스 사용 84
섀도우 매핑 렌더 타겟의 크기 줄이기 84
비동기 컴퓨트 기능 활용 84
컬링 86
다이내믹 해상도 88
다중 카메라 뷰 88
URP의 Render Objects 렌더러 기능 90
HDRP의 커스텀 패스 볼륨 91
디테일 수준(LOD) 사용 92
포스트 프로세싱 효과 프로파일링 93
GPU 상주 드로어 93
GPU 오클루전 컬링 95
Split Graphics Jobs 96
사용자 인터페이스 97
사용자 인터페이스 97
UGUI 성능 최적화 팁 97
Canvas 나누기 97

보이지 않는 UI 요소 숨기기 98
GraphicRaycaster를 제한하고 Raycast Target 비활성화하기 98
레이아웃 그룹 사용 지양 99
대형 리스트 뷰와 그리드 뷰 사용 시 주의 99
요소의 과도한 레이어링 주의 99
전체 화면 UI 사용 시 기타 요소 모두 숨기기 99
World Space 및 Camera Space Canvas 카메라 설정 99
UI 툴킷 성능 최적화 팁 100
효율적인 레이아웃 사용 100
Update 메서드에서 비용이 많이 드는 작업 지양 100
이벤트 처리 최적화 100
스타일 시트 최적화 101
프로파일링 및 최적화 101
타겟 플랫폼에서 테스트 101
오디오 102
무손실 파일을 소스로 사용 102
AudioClip 줄이기 103
AudioClip 임포트 설정 103
AudioMixer 최적화 105
물리 107
콜라이더 단순화 108
설정 최적화 109
시뮬레이션 빈도 조정 110
MeshCollider의 CookingOptions 수정 112
Physics.BakeMesh 사용 113
대형 씬에서 Box Pruning 사용 113

솔버의 반복 횟수 수정 115
자동 트랜스폼 동기화 비활성화 116
수동 동기화 시점.....116
성능 베스트 프랙티스 116
컨택트 배열 사용.....117
충돌 콜백 재사용.....117
정적 콜라이더 이동.....118
비할당 쿼리 사용.....118
2D 물리 118
레이캐스트를 위한 쿼리 배칭 119
물리 디버거를 통한 시각화.....119
애니메이션 120
휴머노이드 릭 대신 제네릭 릭 사용 120
간단한 애니메이션에 대체 기능 사용.....121
스케일 커브 사용 지양 121
시야에 들어올 때에만 업데이트 121
워크플로 최적화 121
애니메이션 계층 구조 분리.....122
바인딩 비용 최소화.....122
복합적인 계층 구조에서 컴포넌트 기반 제약 사용 지양 ...122
애니메이션 리깅의 성능 오버헤드 고려 122
워크플로 및 협업 123
버전 관리 사용 123
Unity Version Control.....124
대형 씬 분할 126
숙련된 개발자 및 아티스트를 위한 리소스..... 127



들어가는 말

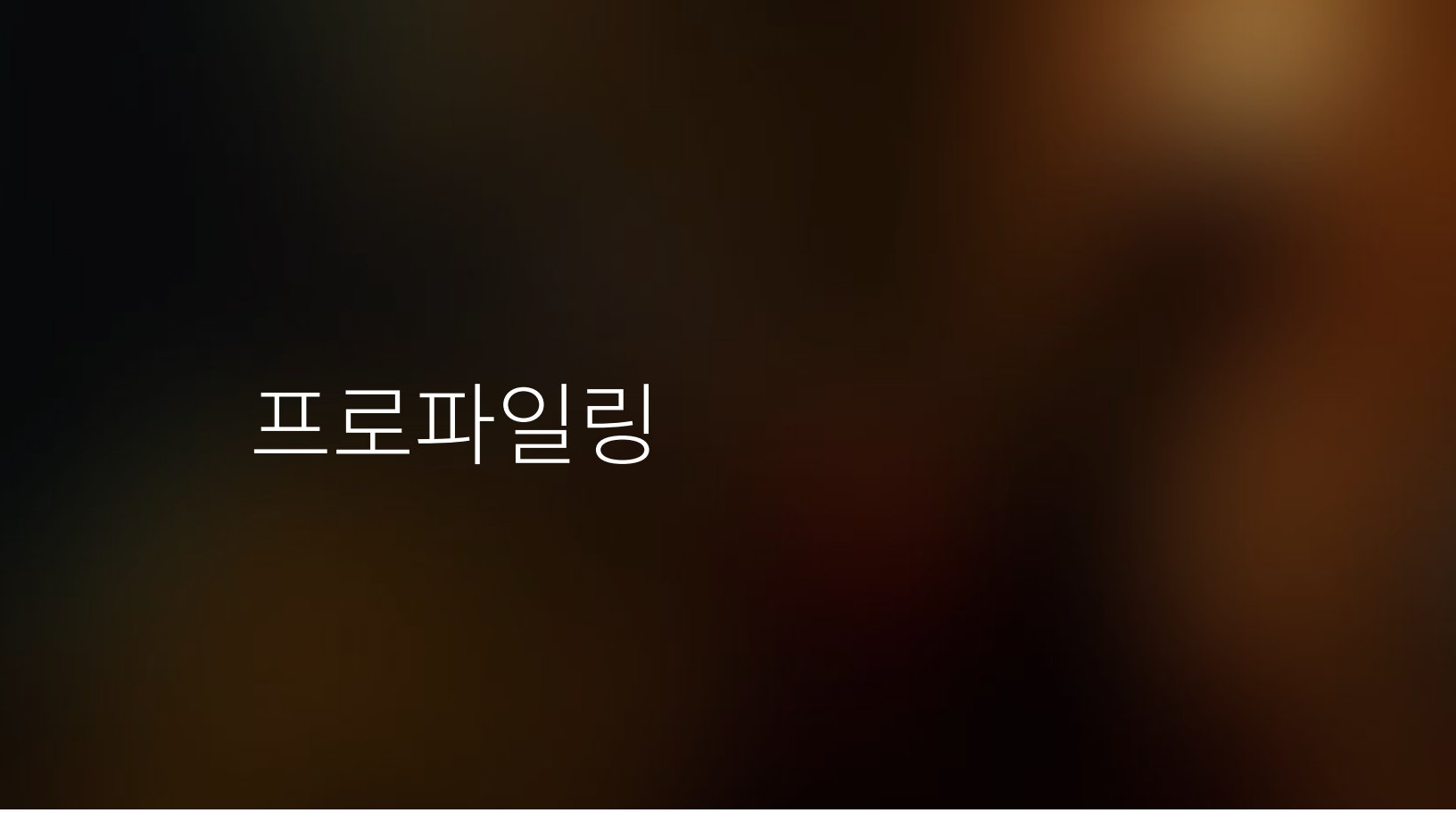
이 가이드에는 Unity 6에서 PC와 콘솔 성능을 최적화하기 위한 모든 최신 팁이 담겨 있습니다. Unity 6에서 모바일, XR, Unity Web용 게임 성능 최적화 가이드와 함께 Unity 6용으로 제공되는 두 가지 최적화 가이드 중 하나입니다.

성능 최적화는 세심한 주의를 기울여야 하는 광범위한 주제입니다. 설계 요구 사항을 충족하는 효율적인 솔루션을 찾으려면 Unity의 클래스와 컴포넌트, 알고리즘과 데이터 구조, 대상 플랫폼의 프로파일링 툴을 완전히 숙지해야 합니다.

콘솔 및 PC 애플리케이션 최적화는 전체 게임 개발 사이클의 근간이 되는 필수 프로세스입니다. 플레이어는 게임이 당연히 초당 60프레임(fps) 이상으로 매끄럽게 실행될 것이라고 여기겠지만, 다양한 플랫폼에서 우수한 성능을 유지하는 일이 그렇게 쉽지만은 않습니다. 효과적으로 최적화하려면 코드 아키텍처와 아트 에셋의 효율을 높이고 타겟 하드웨어가 각자의 제약 내에서 어떻게 작동하는지 이해해야 합니다.

이 가이드는 이러한 베스트 프랙티스를 업계 파트너와 함께 실제 시나리오에서 테스트한 Unity의 전문 소프트웨어 엔지니어의 지식과 조언을 모았습니다.

이어지는 내용에서 PC 및 콘솔 게임에서 최적의 성능을 내는 방법을 소개합니다.



프로파일링

개발 초기부터 자주 타겟 기기에서 프로파일링하기

프로파일링은 런타임에 게임의 성능을 다양한 측면에서 측정하는 프로세스로, 모든 최적화 워크플로의 핵심입니다. Unity와 타겟 하드웨어 제조업체 양측에서 제공하는 프로파일링 툴 제품군을 이해하는 것이 중요합니다.

게임이 타겟 플랫폼에서 어떻게 실행되는지 측정하고 이 정보를 통해 성능 문제의 원인을 추적하는 방법을 알아야 합니다. 내용을 변경하고 프로파일링 툴을 검토하면, 이렇게 바꾼 부분이 실제로 성능 문제를 해결하는지 확인할 수 있습니다.

Unity 프로파일러를 사용하면 애플리케이션에 대한 성능 정보를 얻을 수 있습니다.

프로파일링은 런타임에 게임의 성능을 다양한 측면에서 측정하고 성능 문제의 원인을 추적하는 프로세스입니다. 내용을 변경하고 프로파일링 툴을 모니터링하면, 이렇게 바꾼 부분이 실제로 성능 문제를 해결하는지 확인할 수 있습니다.

출시 직전이 아닌 프로젝트 초기에 프로파일링을 시작하고 개발 사이클 내내 이를 반복하세요. 글리치나 스파이크가 발생하면 즉시 검사하여 프로젝트의 큰 부분을 변경하기 전후 성능을 벤치마킹하세요. 프로젝트의 '성능 시그니처'를 개발하다 보면 새로운 문제를 더 쉽게 발견하게 될 것입니다.

에디터에서 프로파일링을 수행하면 다양한 시스템의 상대적인 게임 내 성능에 관한 정보를 얻을 수 있는 반면, 각 기기를 대상으로 프로파일링하면 더 정확한 인사이트를 확보할 수 있습니다. 가능할 때마다 타겟 기기에서 개발 빌드를 프로파일링하세요. 지원할 기기 중 최고 사양의 기기와 최저 사양의 기기를 모두 프로파일링하고 그것에 맞게 최적화해야 합니다.

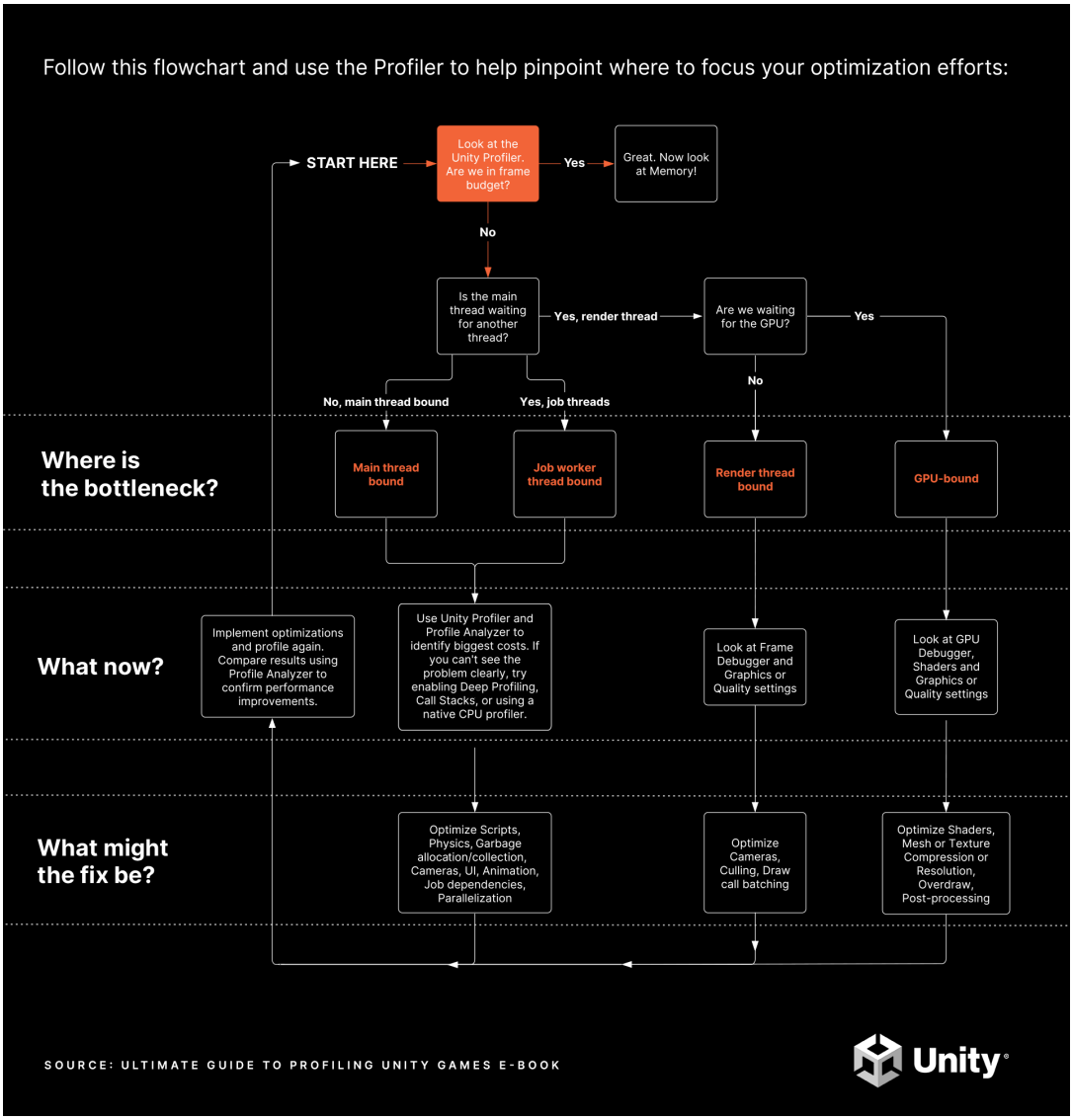
Unity는 Unity 프로파일러, Memory Profiler , Profile Analyzer 같이 병목 지점을 파악할 수 있는 프로파일링 툴 제품군을 제공합니다.

올바른 영역 최적화

게임 성능을 저하하는 요인을 추정하거나 가정하지 않도록 하세요. Unity 프로파일러 및 플랫폼별 툴을 사용하여 성능 저하의 정확한 원인을 찾아야 합니다. 프로파일링 툴을 사용하면 결과적으로 Unity 프로젝트 내부의 상황을 더 손쉽게 파악할 수 있습니다. 하지만 심각한 성능 문제가 발견될 때까지 기다렸다가 해결 도구를 살펴봐야 하는 것은 아닙니다.

물론 여기에서 설명하는 최적화가 모두 여러분의 애플리케이션에 적용되지는 않습니다. 다른 프로젝트에 적합했던 최적화라도 현재 프로젝트에 적용되지 않을 수 있습니다. 따라서 실제 병목 지점을 식별하고 실질적으로 최적화가 필요한 부분에 집중하는 것이 좋습니다.

프로파일링 워크플로를 계획하는 방법에 대해 자세히 알아보려면 Unity 게임 프로파일링 완벽 가이드 전자책을 참고하세요.



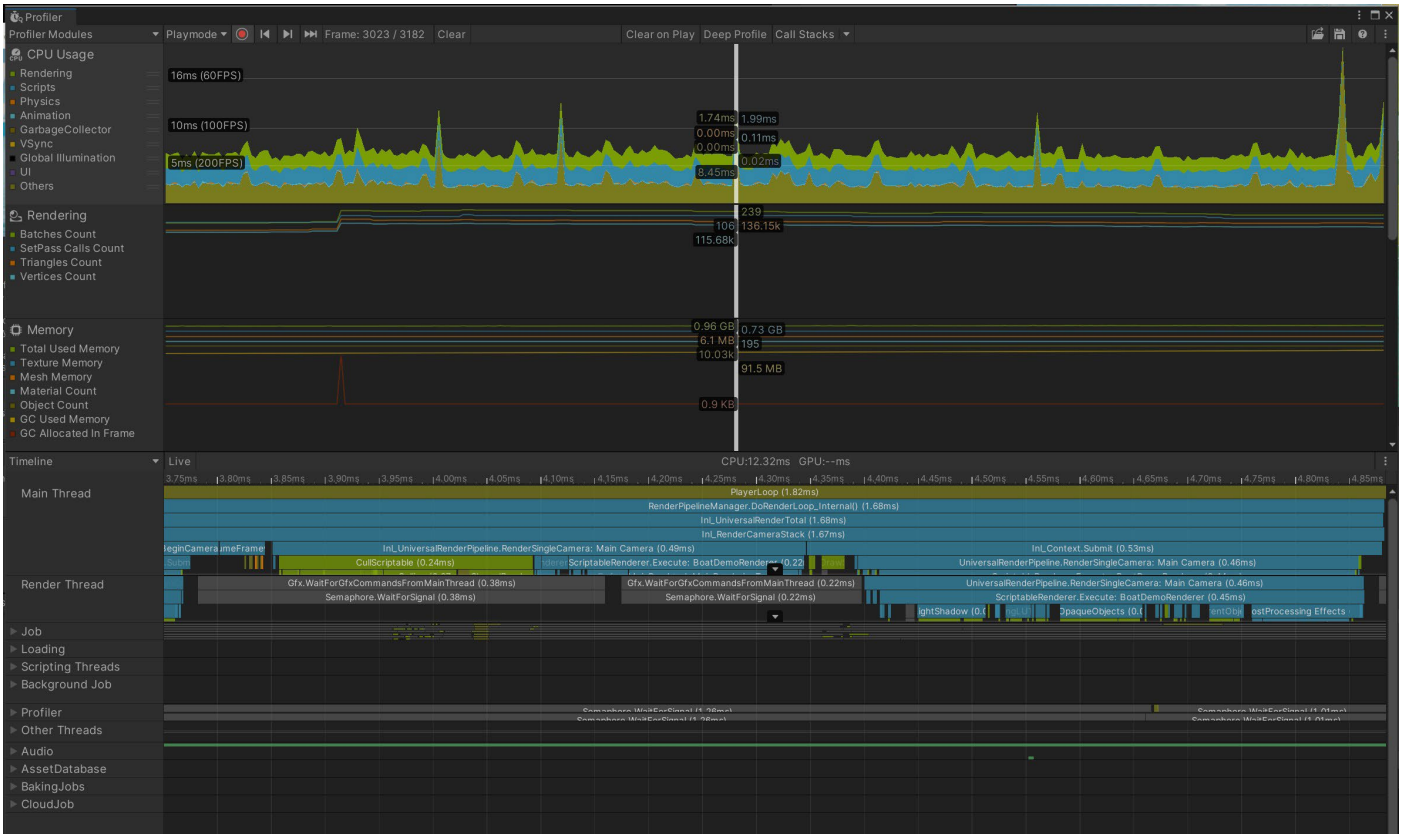
Unity 프로젝트를 효율적으로 프로파일링하는 워크플로입니다.

Unity 프로파일러의 작동 방식 이해

Unity 프로파일러는 런타임 시 병목 현상 또는 중단의 원인을 감지하고 특정 프레임 또는 시점에 발생하는 상황을 더 정확하게 이해하는 데 도움이 됩니다.

Unity 프로파일러는 계측 기반 프로파일러입니다. MonoBehaviour의 Start 또는 Update 메서드, 특정 API 호출과 같이 자동으로 마크업되거나, ProfilerMarker API를 사용하여 명시적으로 래핑된 게임 및 엔진 코드의 타이밍을 프로파일링합니다.

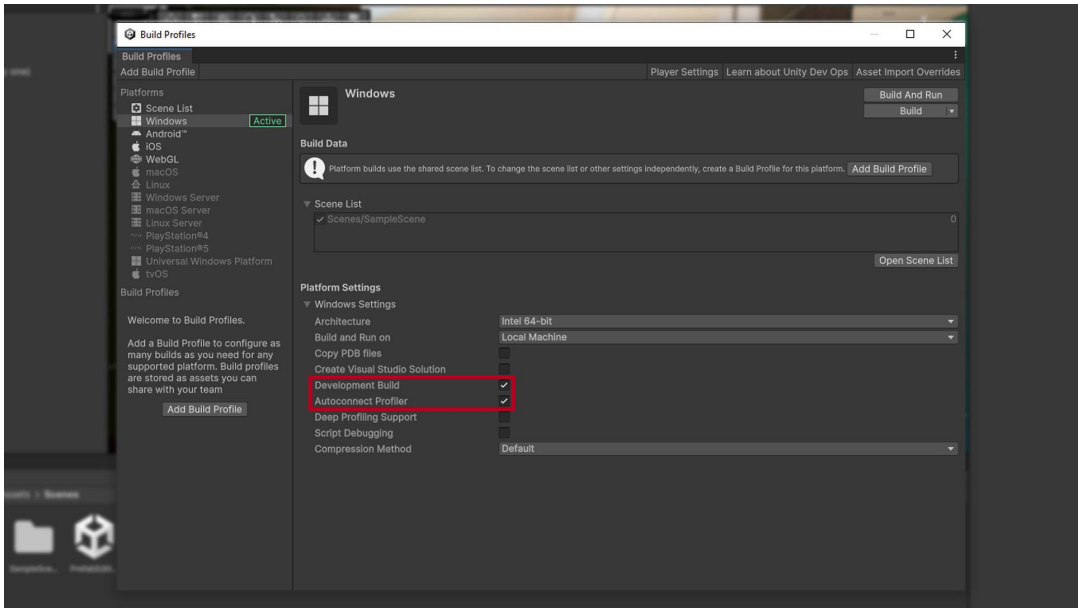
CPU 및 메모리 트랙을 기본적으로 활성화하세요. 예를 들어 물리가 많이 사용되거나 음악에 기반한 게임플레이를 필요에 따라 프로파일링한다면, 필요에 따라 렌더러, 오디오, 물리와 같은 보조 프로파일러 모듈을 모니터링할 수 있습니다. 하지만 성능이 저해되거나 결과가 왜곡되지 않도록 필요한 것만 활성화하세요.



Unity 프로파일러를 사용하여 성능과 리소스 할당을 테스트할 수 있습니다.

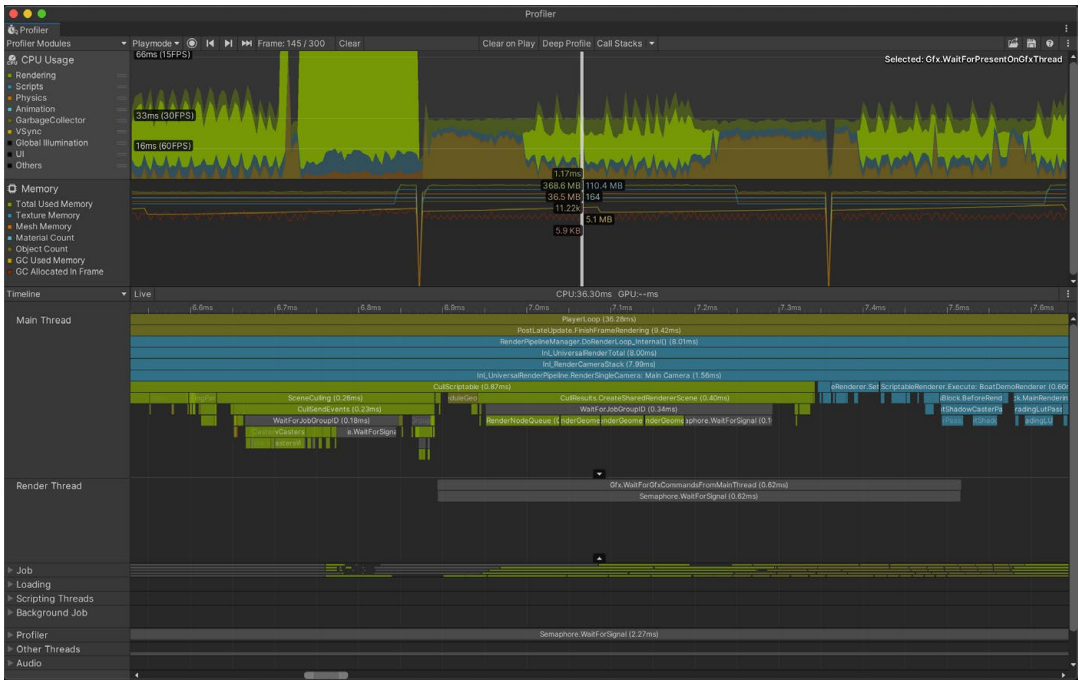
원하는 플랫폼에서 실제 기기의 프로파일링 데이터를 캡처하려면, Build And Run을 클릭하기 전에 **Development Build**를 선택하세요. 그런 다음 애플리케이션이 실행되면 프로파일러를 수동으로 애플리케이션에 연결합니다.

Build Profiles 창의 **Platform Settings**에서 **Autoconnect Profiler**를 활성화할 수도 있습니다. 특히 애플리케이션의 처음 몇 프레임을 캡처하고 싶을 때 유용합니다. 하지만 이 옵션을 사용하면 시작 시간이 5~10초 늘어날 수 있으므로 필요한 경우에만 사용하는 것이 좋습니다.



프로파일링하기 전 플랫폼 설정 조정

프로파일링을 실행할 타겟 플랫폼을 선택하세요. 녹화 버튼을 누르면 애플리케이션 플레이를 몇 초간 추적할 수 있습니다(기본 300 프레임). 더 오래 캡처해야 한다면 **Unity > Preferences > Analysis > Profiler > Frame Count**로 이동하여 이 값을 2000까지 늘리면 됩니다. 이렇게 하면 CPU 및 메모리 리소스가 더 많이 소모되지만 특정 시나리오에서는 이 설정이 유용할 수 있습니다.



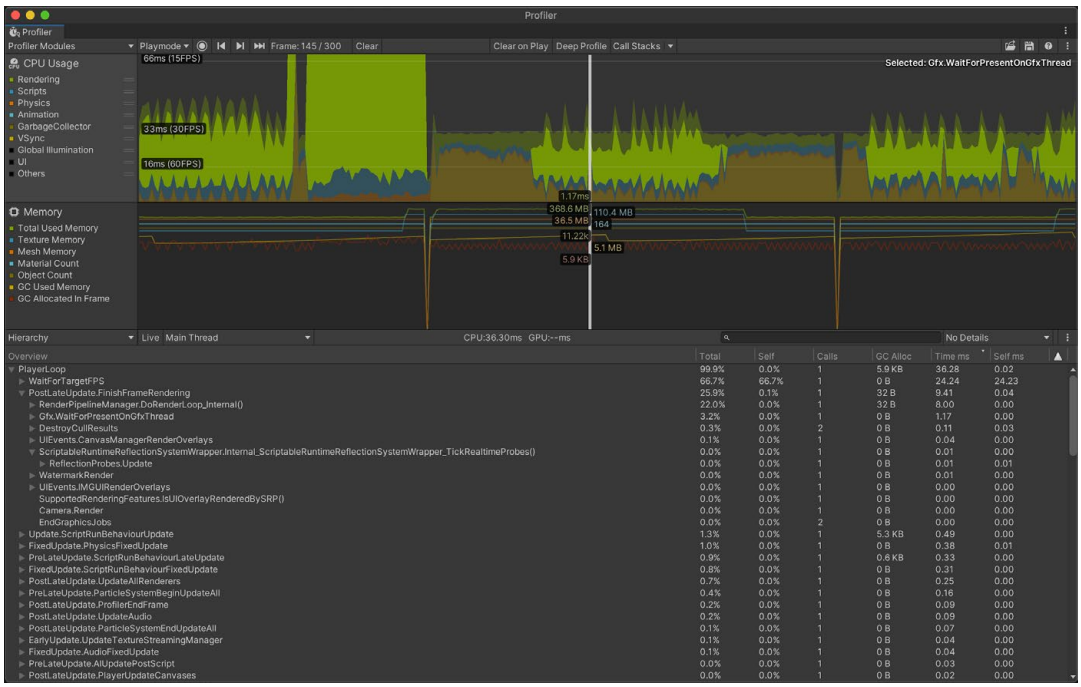
타임라인 뷰를 사용하여 CPU 바운드 또는 GPU 바운드 여부 판단

Deep Profiling 설정을 사용하면 Unity는 스크립트 코드에 있는 모든 함수 호출의 시작과 끝을 프로파일링하여 정확히 애플리케이션의 어느 부분이 실행되고 있으며 잠재적으로 지연을 유발하는지 제시합니다. 그러나 Deep Profiling을 사용하면 모든 메서드 호출의 오버헤드가 증가하며 성능 분석이 왜곡될 수 있습니다.

창을 클릭하여 특정 프레임을 분석하고, 그런 다음 Timeline 또는 계층 구조(Hierarchy) 뷰를 사용하세요.

— **Timeline**: 특정 프레임의 타이밍에 대한 요약 정보를 시각화하여 표시합니다. 이를 통해 각 활동이 다양한 스레드 전반에서 서로 어떤 관계를 맺고 있는지 시각화할 수 있습니다. 이 옵션을 사용하여 CPU 바운드 또는 GPU 바운드 여부를 판단하세요.

— **계층 구조**: 그룹화된 ProfileMarkers의 계층 구조를 제시합니다. 이를 통해 밀리초 단위(Time ms 및 Self ms)의 시간 비용을 기준으로 샘플을 정렬할 수 있고, 함수에 대한 호출 수와 프레임의 관리되는 힙 메모리(GC Alloc) 양도 알 수 있습니다.



계층 구조 뷰를 통해 시간 비용에 따라 ProfileMarkers 정렬

프로파일링이 처음이라면 다음의 짧은 동영상 튜토리얼 시리즈로 시작해 보세요.

— 프로파일러 소개

— Profile Analyzer 소개

— Memory Profiler 소개

Unity 프로파일러에 대해 자세히 알아보려면 여기에서 프로파일링 전자책을 다운로드하세요.

프로젝트에서 최적화를 수행하기 전에 먼저 Profiler .data 파일을 저장하세요. 변경 사항을 구현하고 수정 이전 및 이후에 저장된 .data 파일을 비교해 보세요. 프로파일링, 최적화, 비교를 반복하여 성능을 향상할 수 있습니다.

딥 프로파일링

Build Settings에서 Deep Profiling Support를 활성화할 수 있습니다. 딥 프로파일러는 빌드된 플레이어가 시작되면 ProfilerMarkers에 명시적으로 래핑된 코드 타이밍뿐만 아니라 코드의 모든 부분을 프로파일링합니다.

딥 프로파일링을 활성화하면 Unity가 스크립트 코드에 있는 모든 함수 호출의 시작과 끝을 프로파일링할 수 있습니다. 이를 통해 애플리케이션의 어느 부분으로 인해 속도가 저하되는지 정확하게 식별할 수 있습니다.

그러나 딥 프로파일링은 많은 리소스와 메모리를 소모합니다. 각 ProfilerMarker가 작은 양의 오버헤드(플랫폼에 따라 약 10ns)를 추가하므로, 측정 지점을 추가할 때마다 애플리케이션 속도가 더 느려집니다. 또한 함수 호출이 많은 경우, 딥 프로파일링 사용 시 오버헤드가 증폭된다는 점도 주의해야 합니다.

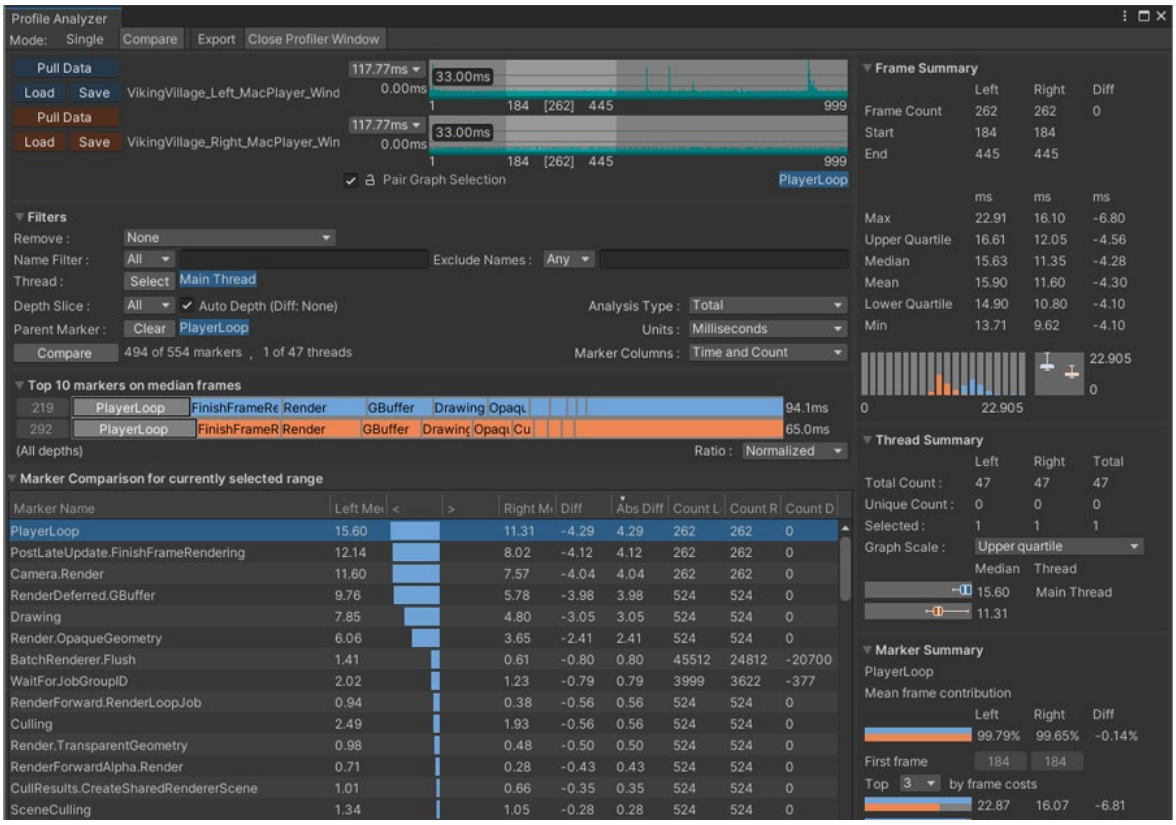
GC.Alloc이나 JobHandle.Complete와 같은 마커가 있는 샘플에 대한 자세한 내용을 보려면, 프로파일러 창 툴바에서 **Call Stacks** 설정을 활성화하세요. 이렇게 하면 샘플의 전체 호출 스택이 제공되므로 딥 프로파일링으로 인한 오버헤드 없이 필요한 정보를 얻을 수 있습니다.

가능하면 항상 Deep Profile 대신 Call Stacks를 사용하는 것이 좋습니다.

딥 프로파일링을 사용하면 애플리케이션 실행 속도가 매우 저하되므로 필요한 경우에만 사용해야 합니다.

Profile Analyzer 사용하기

Profile Analyzer를 사용하면 프로파일러 데이터의 여러 프레임을 집계한 다음 원하는 프레임을 찾을 수 있습니다. 프로젝트를 변경하면 프로파일러에 어떤 영향을 주는지 확인하고 싶으신가요? **Compare** 뷰를 사용하면 두 데이터 세트를 로드해서 차이점을 확인할 수 있으므로 변경 사항을 테스트하고 결과를 개선할 수 있습니다. Profile Analyzer는 Unity의 패키지 관리자를 통해 사용할 수 있습니다.



Profile Analyzer로 프레임과 마커 데이터를 더 자세히 살펴보고 기존 프로파일러를 보완할 수 있습니다.

Profile Analyzer의 기능에 대해 자세히 알아보려면 이 Profile Analyzer 튜토리얼을 시청해 보세요.

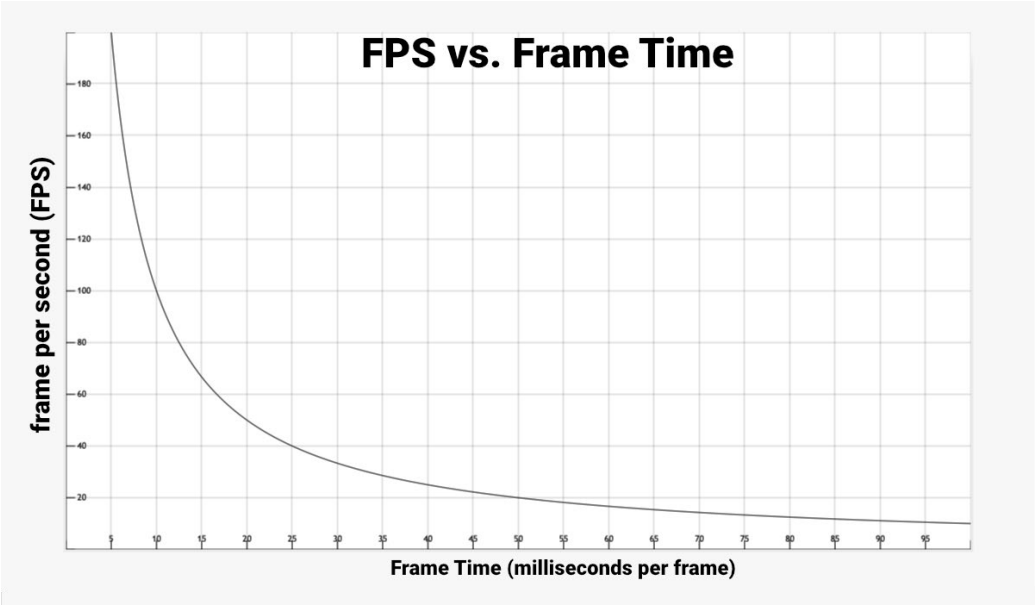
프레임당 정해진 시간 예산으로 작업하기

각 프레임에는 목표하는 초당 프레임 수(fps)를 기반으로 시간 예산이 주어집니다. 애플리케이션이 30fps로 실행되려면 프레임 예산이 프레임당 33.33ms(1000ms/30fps)를 초과할 수 없습니다. 마찬가지로 60fps의 경우 목표 시간 예산은 프레임당 16.66ms입니다.

FPS: 정확하지 않을 수 있는 지표

게이머는 보통 프레임 속도 또는 초당 프레임 수로 성능을 측정합니다. 하지만 애플리케이션 성능을 측정할 때 이 방식은 정확하지 않은 결과를 도출할 수 있습니다.

따라서 프레임 시간(밀리초)을 사용하는 것이 좋습니다. 그 이유는 **FPS와 프레임 시간을 비교**하는 다음 그래프에서 알 수 있습니다.



FPS와 프레임 시간 비교

다음 수치를 예로 생각해 보겠습니다.

1,000ms/초 / 900fps = 프레임당 1.111ms
1,000ms/초 / 450fps = 프레임당 2.222ms
1,000ms/초 / 60fps = 프레임당 16.666ms
1,000ms/초 / 56.25fps = 프레임당 17.777ms

애플리케이션이 900fps로 실행된다는 것은 프레임당 1.111ms의 프레임 시간이 걸린다는 의미입니다. 450fps에서는 프레임당 2.222ms가 걸립니다. **프레임 속도가 절반으로 줄어든 것처럼 보이지만, 프레임 시간 차이는 프레임당 1.111ms에 불과합니다**.

60fps 및 56.25fps의 차이를 살펴보면 각각 프레임 시간은 프레임당 16.666ms 및 17.777ms입니다. 프레임당 차이는 1.111ms지만 프레임 속도 하락을 백분율로 환산하면 훨씬 차이가 작다고 할 수 있습니다.

개발자가 게임 속도를 벤치마킹할 때 fps가 아닌 평균 프레임 시간을 사용하는 이유도 여기에 있습니다.

목표 프레임 속도 아래로 떨어지지 않는 한, fps는 걱정하지 않아도 됩니다. 그보다 프레임 시간에 집중하여 게임이 얼마나 빠르게 실행되는지 측정하고 프레임 예산이 초과하지 않도록 관리하는 것이 중요합니다.

자세한 내용은 로버트 던롭의 FPS와 프레임 시간 비교(영문) 문서를 참조하세요.

GPU 바운드 또는 CPU 바운드 여부 판단하기

CPU(중앙 처리 장치)는 무엇을 그려야 하는지 결정하고, GPU(그래픽 처리 장치)는 이를 그리는 역할을 담당합니다.

CPU가 프레임을 처리하여 GPU를 위해 준비하는 시간이 너무 오래 걸리는 경우, CPU의 성능에 따라 전반적인 프레임 속도가 제한되므로 CPU 바운드입니다.

마찬가지로 CPU가 준비한 프레임을 GPU가 렌더링하는 시간이 너무 오래 걸리는 경우 GPU 바운드입니다.

프로파일러를 통해 CPU가 프레임 예산을 할당된 양보다 오래 사용 중인지, 또는 GPU에 문제가 있는지 파악할 수 있습니다. 다음과 같이 Gfx가 접두어인 마커를 방출하는 방식을 사용합니다.

- **Gfx.WaitForCommands** 마커가 표시되면 렌더 스레드가 준비되었으나 메인 스레드에 병목 현상이 일어날 수 있음을 의미합니다.
- **Gfx.WaitForPresentOnGfxThread**가 빈번하게 표시될 수 있는데, 메인 스레드는 준비되었으나 렌더 스레드를 기다리는 중이었음을 의미합니다. 이는 애플리케이션이 GPU 바운드임을 나타낼 수 있습니다. CPU 프로파일러 모듈의 Timeline 뷰에서 렌더 스레드의 활동을 확인하세요.

  렌더 스레드가 **Camera.Render**에서 시간을 소모하는 경우, 애플리케이션은 CPU 바운드이며 GPU에 드로우 콜이나 텍스처를 전송하는 데 너무 많은 시간을 소모하고 있을 수 있습니다.

  렌더 스레드가 **Gfx.PresentFrame**에서 시간을 소모하는 경우, 애플리케이션은 GPU 바운드이거나 GPU에서 VSync를 기다리는 중일 수 있습니다.

마커 전체 목록을 확인하려면 공통 프로파일러 마커 기술 자료를 참조하세요. 프레임 파이프라인에 대한 자세한 내용은 더욱 원활한 게임플레이를 위한 Unity 2020.2 Time.deltaTime 개선 블로그 게시물을 확인하시기 바랍니다.

네이티브 프로파일링 및 디버깅 툴 사용

Unity 툴로 프로파일링을 시작한 다음 더 세부적인 데이터가 필요한 경우 타겟 플랫폼에서 사용할 수 있는 네이티브 프로파일링 및 디버깅 툴을 활용하세요.

네이티브 프로파일링 툴

Intel

— Intel VTune: 이 Intel 프로세서 전용 툴 세트를 사용하면 Intel 플랫폼에서 성능 병목 현상을 빠르게 찾아서 해결할 수 있습니다.

— Intel GPA 제품군: 문제 영역을 빠르게 식별하여 게임 성능을 개선하는 그래픽스 중심 툴 세트입니다.

Xbox 콘솔 및 Windows PC

— PIX: PIX는 DirectX 12를 사용하는 Windows 및 Xbox 콘솔 게임 개발자를 위한 성능 조정 및 디버깅 툴입니다. CPU와 GPU 성능을 파악하고 분석하는 툴뿐만 아니라 다양한 실시간 성능 카운터를 모니터링하는 툴도 함께 제공합니다. Windows 개발자라면 여기를 먼저 살펴보세요. Xbox용 PIX에 대해 자세히 알아보려면 Xbox 개발자로 등록해야 합니다. 처음이라면 여기를 살펴보세요.

PC/범용

— AMD µProf: AMD uProf는 AMD 하드웨어에서 실행하는 애플리케이션의 성능을 파악하고 프로파일링하는 성능 분석 툴입니다.

— NVIDIA NSight 개발자 툴: 개발자가 NVIDIA의 최신 가속 컴퓨팅 하드웨어를 사용하여 동급 최고 수준의 최첨단 소프트웨어를 빌드, 디버그, 프로파일링, 개발할 수 있도록 지원하는 툴입니다.

— Superluminal: Superluminal은 C++, Rust, .NET으로 작성된 Windows, Xbox 콘솔, PlayStation® 애플리케이션의 프로파일링을 지원하는 고성능 고빈도 프로파일러입니다. 단 이 제품은 유료로, 라이선스가 있어야 사용할 수 있습니다.

PlayStation®

— CPU 프로파일러 툴을 PlayStation 개발 환경에 사용할 수 있습니다. 자세히 알아보려면 PlayStation 개발자로 등록해야 합니다. 처음이라면 여기를 살펴보세요.

웹

— Firefox Profiler: Firefox Profiler를 사용하면 호출 스택을 자세히 살펴볼 수 있고, 무엇보다도 Unity Web 빌드에 대한 플레임 그래프를 확인할 수 있습니다. 프로파일링 캡처를 나란히 비교할 수 있는 비교 툴도 제공합니다.

— Chrome DevTools Performance: Unity Web 빌드를 프로파일링할 수 있는 웹 브라우저 툴입니다.

GPU 디버깅 및 프로파일링 툴

Unity 프레임 디버거는 CPU에서 전송하는 드로우 콜을 캡처하여 보여 줍니다. 다음 툴을 사용하면 해당 커맨드를 수신한 GPU가 어떤 작업을 수행하는지 확인할 수 있습니다.

일부는 플랫폼 전용 툴이며 더욱 긴밀한 플랫폼 통합을 제공합니다. 다음은 다양한 플랫폼별로 특별히 이용되는 여러 툴입니다.

— RenderDoc: 데스크톱 및 모바일 플랫폼용 GPU 디버거

— Intel GPA: Intel 기반 플랫폼용 그래픽스 프로파일링 툴

— Apple Frame Capture 디버깅 툴: Apple 플랫폼용 GPU 디버깅 툴

— Visual Studio Graphics Diagnostics: Windows, Xbox 등의 DirectX 기반 플랫폼은 이 툴 또는 PIX 사용

— NVIDIA Nsight Graphics: NVIDIA GPU용 그래픽스 프로파일러 및 디버거

— AMD Radeon Developer 툴 제품군: AMD GPU용 GPU 프로파일러

— Xcode 프레임 디버거: iOS 및 macOS용

Project Auditor

Project Auditor는 프로젝트의 스크립트와 설정에 대한 정적 분석을 수행할 수 있는 실험적 툴입니다. 이 툴을 활용하면 관리되는 메모리 할당, 비효율적인 프로젝트 설정, 잠재적인 성능 병목 현상의 원인을 추적할 수 있습니다.

Project Auditor는 에디터와 함께 사용할 수 있는 비공식 무료 패키지입니다. 자세한 내용은 Project Auditor 기술 자료를 참조하세요.

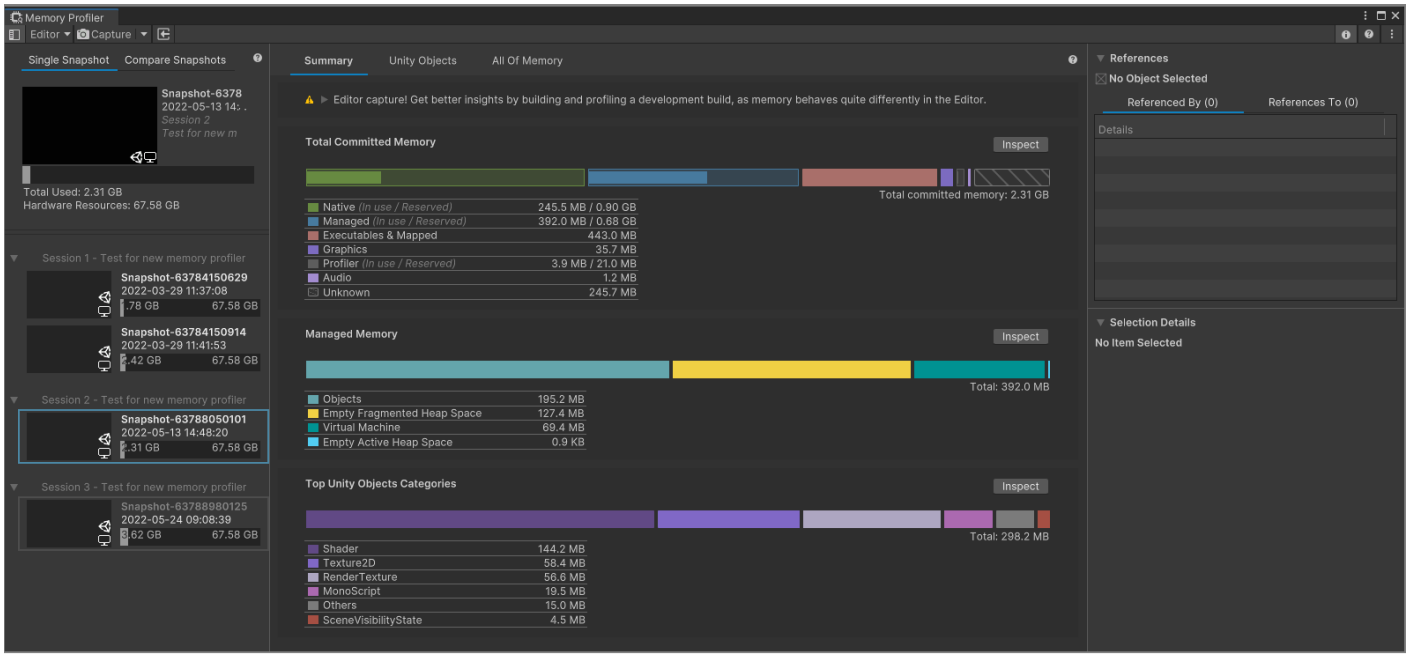
메모리

Unity는 사용자 생성 코드 및 스크립트에 자동 메모리 관리를 사용합니다. 값 유형 로컬 변수처럼 작은 데이터는 스택에 할당되고, 더 큰 데이터와 장기 스토리지는 관리되는 힙 또는 네이티브 힙에 할당됩니다.

가비지 컬렉터는 사용되지 않은 관리되는 힙 메모리를 주기적으로 파악하여 할당을 해제합니다. 에셋 가비지 컬렉션은 요청 시 또는 새 씬을 로드할 때 실행되며 네이티브 오브젝트 및 리소스의 할당을 해제합니다.
이 작업은 자동으로 실행되지만, 힙의 모든 오브젝트를 검사하는 과정에서 게임이 끊기거나 느려질 수 있습니다.

메모리 사용량을 최적화하려면 관리되는 힙 메모리의 할당 및 할당 해제 시점뿐만 아니라 가비지 컬렉션의 영향을 최소화하는 방법을 알아야 합니다.

자세한 내용은 관리되는 힙의 이해를 참조하세요.



Memory Profiler에서 스냅샷 캡처, 검토 및 비교

Memory Profiler 사용하기

Memory Profiler 패키지는 관리되는 힙 메모리의 스냅샷을 만들어 단편화 또는 메모리 누수와 같은 문제를 식별할 수 있습니다. Memory Profiler에 대해 알아보려면 이 Memory Profiler 튜토리얼을 살펴보세요.

Unity Objects 탭에서 중복된 메모리 항목을 제거할 수 있는 영역을 식별하거나 많은 양의 메모리를 사용하는 오브젝트를 파악할 수 있습니다. **All of Memory** 탭에서 Unity가 스냅샷 내에서 추적하는 모든 메모리에 대한 분석을 확인할 수 있습니다.

Unity Memory Profiler를 활용하여 메모리 사용량을 개선하는 방법을 알아보세요.

GC(가비지 컬렉션)의 영향 줄이기

Unity는 Boehm-Demers-Weiser 가비지 컬렉터를 사용하며, 이 컬렉터는 프로그램 코드의 실행을 중단하고 작업이 완료될 때만 일반 실행을 재개합니다.

힙의 불필요한 할당은 GC 스파이크를 유발할 수 있으므로 유의해야 합니다.

— **문자열:** C#에서 문자열은 값 유형이 아닌 레퍼런스 유형입니다. 다시 말해서 모든 새 문자열은 일시적으로만 사용되더라도 관리되는 힙에 할당됩니다. 문자열을 대규모로 사용하는 경우 불필요한 생성이나 조작을 줄이세요. JSON, XML 같은 문자열 기반 데이터 파일은 파싱하지 않는 것이 좋습니다. 데이터를 스크립터블 오브젝트에 저장하거나 MessagePack 또는 Protobuf 같은 포맷으로 저장하세요. 런타임에 문자열을 빌드해야 하는 경우 StringBuilder 클래스를 사용합니다.

— **Unity 함수 호출**: 일부 Unity API 함수는 특히 관리되는 오브젝트 배열을 반환하는 힙 할당을 생성합니다. 레퍼런스를 루프 도중에 할당하지 말고 배열에 캐시하세요. 가비지 생성을 방지하는 함수도 활용하세요. 예를 들어 문자열을 **GameObject.tag**와 직접 비교하는 대신 **GameObject.CompareTag**를 사용하는 방법이 있습니다(새로운 문자열 반환은 가비지를 생성함).

— **박싱**: 레퍼런스 유형의 변수 대신 값 유형 변수를 전달하는 방식은 지양하세요. 이렇게 하면 임시 오브젝트가 생성되며, 그에 수반되는 잠재적 가비지가 값 유형을 암묵적으로 유형 오브젝트로 전환합니다(예: **int i = 123; object o = i**). 대신 전달하려는 값 유형에 구체적인 오버라이드를 제공해 보세요. 이러한 오버라이드에는 제네릭이 사용될 수도 있습니다.

— **코루틴**: yield는 가비지를 생성하지 않지만 새로운 WaitForSeconds 오브젝트를 만들면 가비지가 생성됩니다. yield 라인에서 생성하는 대신 WaitForSeconds 오브젝트를 캐싱하고 재사용하세요.

— **LINQ 및 정규 표현식**: 두 가지 모두 백그라운드 박싱을 통해 가비지를 생성합니다. 성능이 문제가 된다면 LINQ와 정규식을 사용하지 마세요. 새로운 배열을 만드는 대신 for 루프와 리스트를 사용하세요.

— **제네릭 컬렉션 및 기타 관리되는 유형**: Update에서 프레임마다 List나 컬렉션(예: 플레이어의 특정 반경 내에 있는 적 목록)을 선언하고 채우면 안 됩니다. 대신 MonoBehaviour의 멤버로 List를 만들고 Start에서 초기화하세요. 사용 전에 모든 프레임에서 Clear로 컬렉션을 비우면 됩니다.

자세한 내용은 매뉴얼의 가비지 컬렉션 베스트 프랙티스 페이지를 참고하세요.

가능한 경우 가비지 컬렉션 시간 측정하기

가비지 컬렉션 멈춤 현상이 게임의 특정 지점에 영향을 주지 않는다면 **System.GC.Collect**로 가비지 컬렉션을 트리거할 수 있습니다.

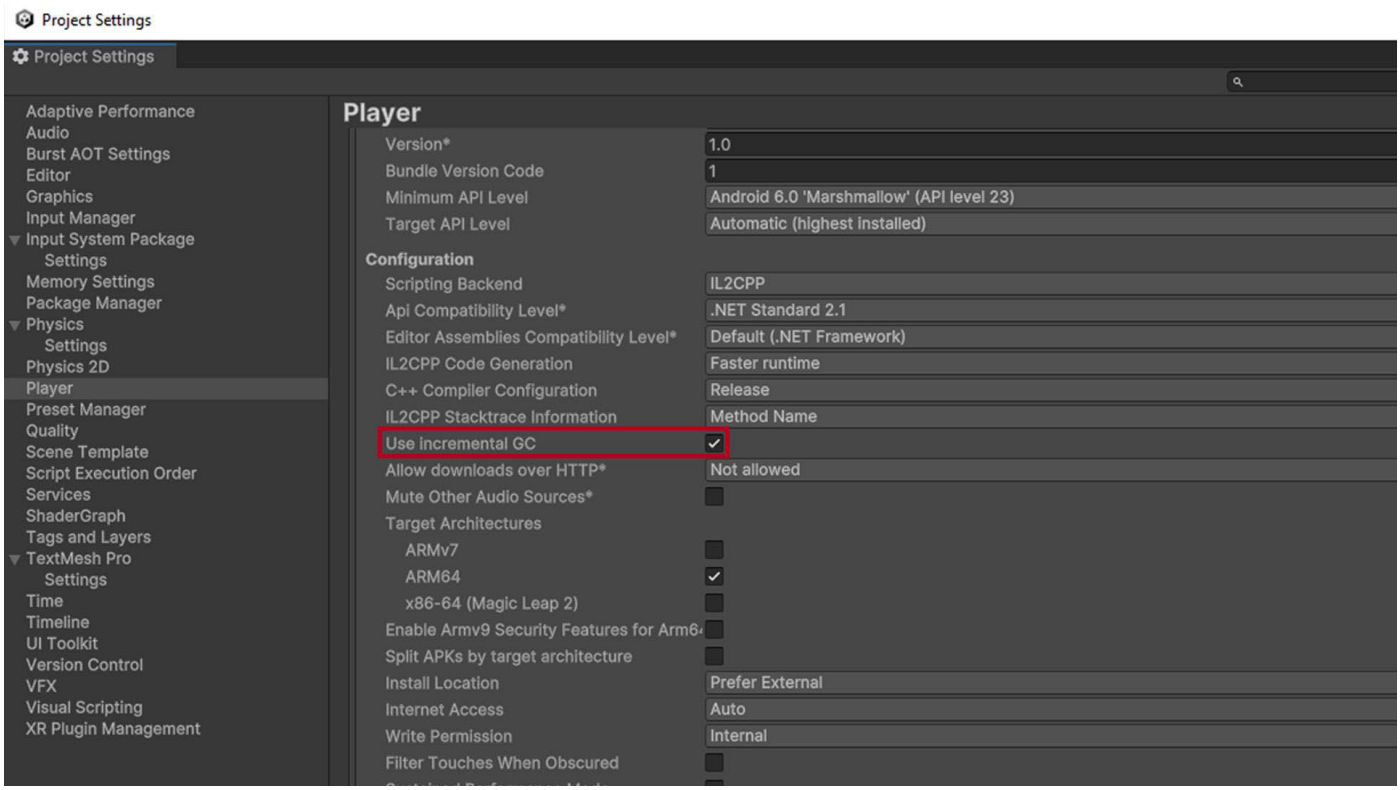
자동 메모리 관리의 이해에서 이 방식을 활용하는 방법을 참고하세요.[1]

1 GC를 사용하면 일부 C# 호출에 읽기-쓰기 배리어가 추가될 수 있으며, 이에 따라 스크립팅 호출 오버헤드 프레임당 최대 1ms가 늘어나는 오버헤드가 발생할 수 있습니다. 최적의 성능을 위해서는 메인 게임플레이 루프에 GC Allocs가 없도록 하고 사용자가 알 수 없는 곳에 GC.Collect를 숨기는 것이 좋습니다.

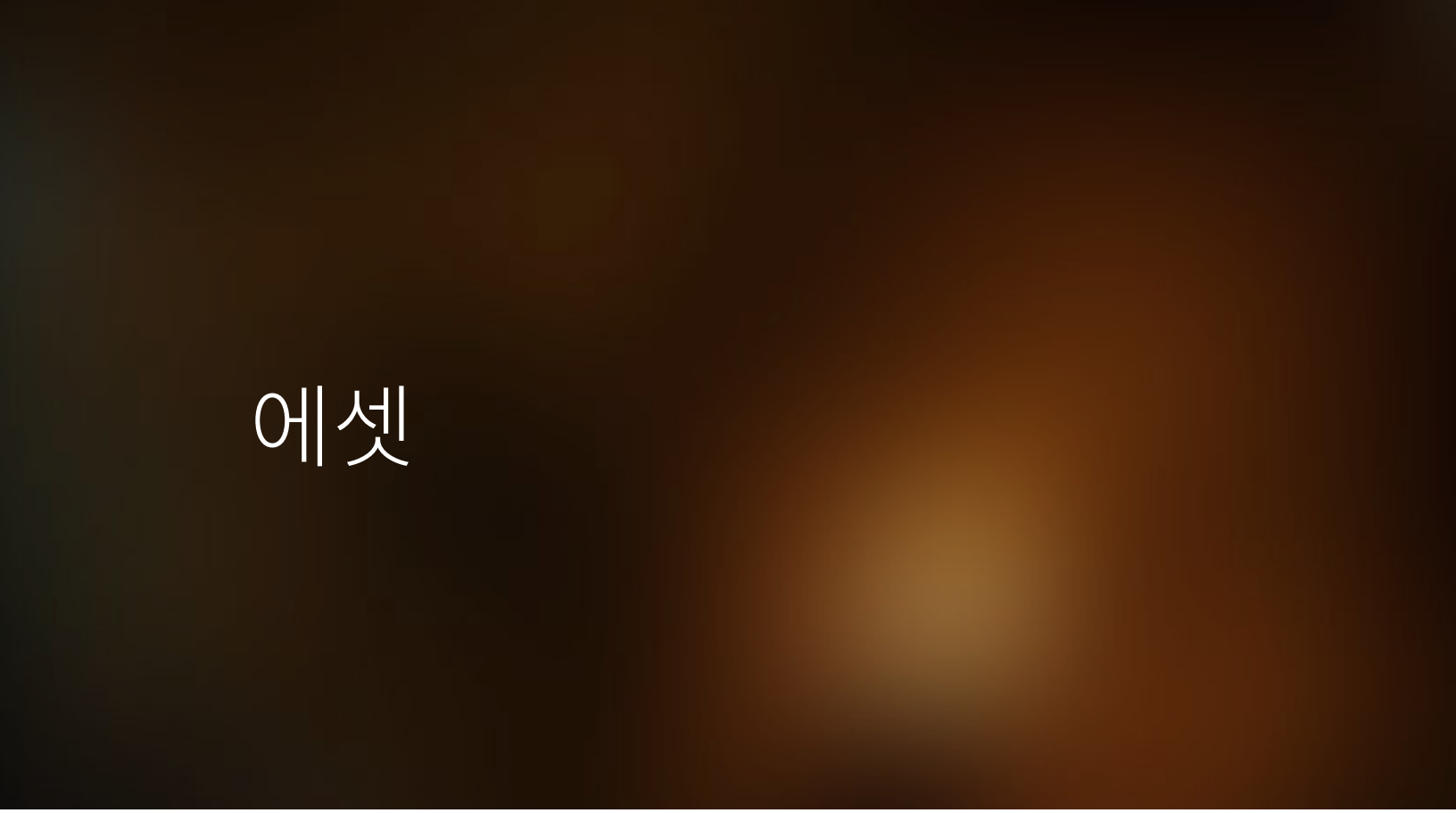
점진적 가비지 컬렉터를 활용하여 GC 워크로드 분할하기

점진적 가비지 컬렉션을 사용하면 프로그램 실행 중에 한 번 오래 중단되는 것이 아니라, 훨씬 짧은 중단이 여러 프레임에 걸쳐 여러 번 나타납니다. 가비지 컬렉션이 성능에 영향을 미친다면 이 옵션을 사용하여 GC 스파이크를 줄일 수 있는지 확인해 보세요. Profile Analyzer를 사용하여 이 방식이 애플리케이션에 도움이 되는지 확인하세요.

참고: 점진적 GC는 일시적으로 가비지 컬렉션 문제를 줄일 수 있지만, 장기적인 관점에서 문제를 해결하는 최선의 방법은 가비지 컬렉션을 트리거하는 잦은 할당을 찾아 중지하는 것입니다.



점진적 가비지 컬렉터를 사용하여 GC 스파이크 줄이기



에셋

에셋 파이프라인은 애플리케이션의 성능에 지대한 영향을 줄 수 있습니다. 숙련된 테크니컬 아티스트가 에셋 포맷, 사양, 임포트 설정을 정의하고 실행하여 프로젝트를 원활하게 진행할 수 있도록 팁을 제공해 드릴 수 있습니다.

기본 설정에 의존하지 마세요. 플랫폼별 오버라이드 탭을 사용하면 텍스처, 메시 지오메트리와 같은 에셋을 최적화할 수 있습니다. 잘못된 설정으로 인해 빌드 크기가 커지고 빌드 시간이 길어지고 GPU 성능과 메모리 효율이 떨어질 수 있습니다. 프리셋 기능으로 기본 설정을 커스터마이즈하여 프로젝트를 개선해 보세요.

아트 에셋 임포트 베스트 프랙티스에 대한 가이드를 참조하시기 바랍니다. Unity Learn 온라인 교육인 모바일 애플리케이션용 3D 아트 최적화에서 모바일 전용 가이드와 함께 여러 일반 팁도 확인할 수 있습니다. 또한 프리셋 활용법을 자세히 알아보려면 GDC 2023의 게임 제작의 모든 단계에 대한 기술 팁 세션을 시청해 보세요.

텍스처 압축

동일한 모델과 텍스처를 사용하는 두 가지 예를 생각해 보겠습니다. 위쪽의 설정은 아래쪽보다 5배 이상 많은 메모리를 사용하지만, 비주얼 품질에서 큰 차이를 보이지 않습니다.

압축되지 않은 텍스처에는 더 많은 메모리가 필요합니다.

텍스처 압축을 적절하게 적용하면 성능 측면에서 상당한 이점을 얻을 수 있습니다.

로드 시간을 단축하고 메모리 사용량을 줄이고 렌더링 성능을 크게 향상할 수 있습니다. 압축된 텍스처는 압축되지 않은 32비트 RGBA 텍스처에 필요한 메모리 대역폭의 일부만 사용합니다.

타겟 플랫폼별로 권장되는 텍스처 압축 포맷 목록을 확인해 보세요.

— **iOS/Android**: **ASTC**를 사용하는 것이 좋습니다.

— **PC/XBox One/PlayStation®4**: **BC7**(고품질) 또는 **DXT1**(저품질/일반 품질)을 사용하는 것이 좋습니다.

텍스처 임포트 설정

텍스처도 잠재적으로 많은 리소스를 사용할 수 있기 때문에 임포트 설정이 매우 중요합니다. 다음 가이드라인을 따르는 것이 좋습니다.

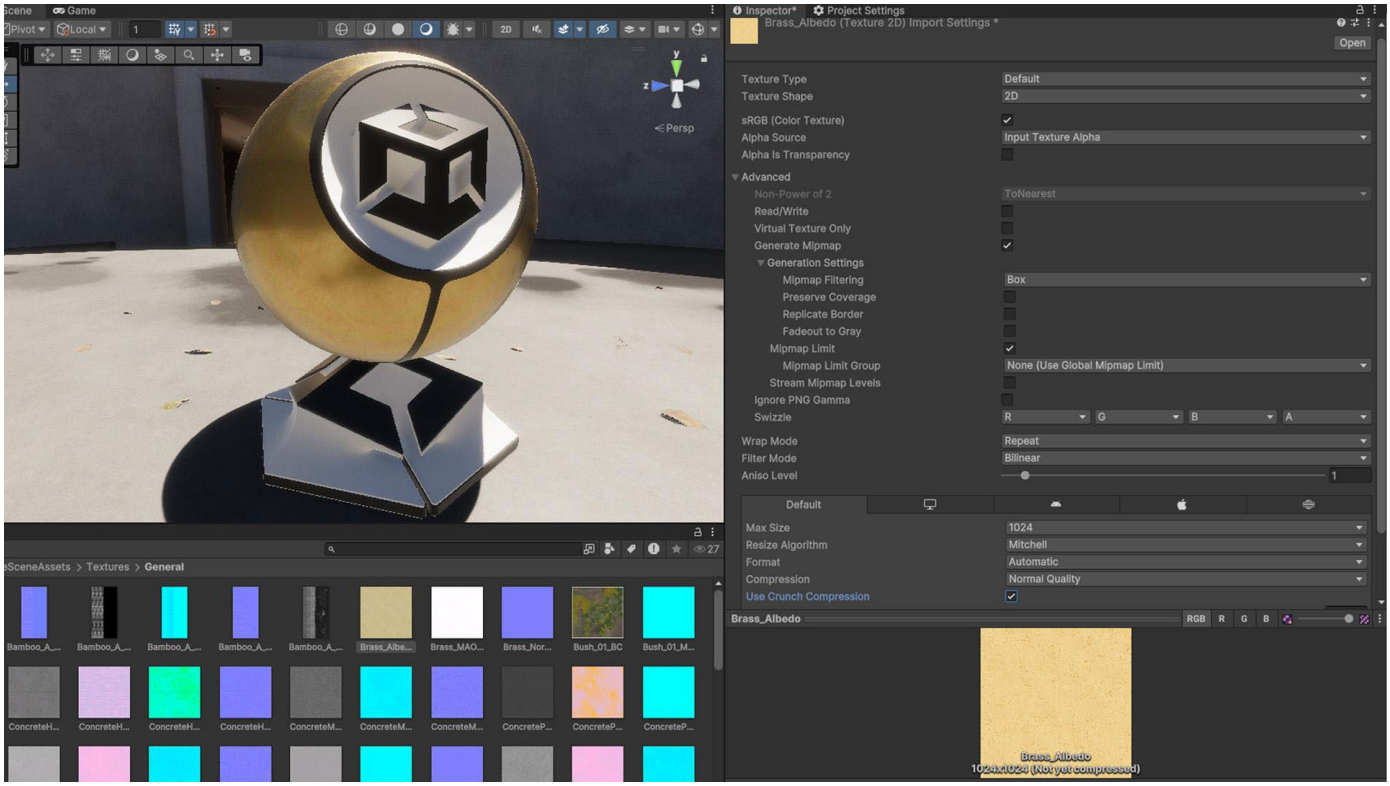
— **최대 크기 줄이기**: 시각적으로 허용 가능한 결과를 내는 최소한의 설정을 사용합니다. 이렇게 하면 결과물의 손상을 피하면서 빠르게 텍스처 메모리를 줄일 수 있습니다.

— **POT(Powers of Two) 사용**: Unity에서 텍스처 압축 포맷을 사용하려면 POT 텍스처 크기가 필요합니다.

— **Read/Write Enabled 옵션 해제**: 이 옵션을 활성화하면 CPU 및 GPU 주소 지정 가능 메모리에서 사본이 만들어지므로 텍스처의 메모리 사용량이 두 배로 늘어납니다. 대부분의 경우 이 옵션은 비활성화 상태로 유지하고, 런타임에 텍스처를 생성하고 덮어써야 하는 경우에만 활성화합니다.

Texture2D.Apply 함수를 통해 옵션을 적용하고, **makeNoLongerReadable** 값은 true로 전달할 수 있습니다.

— **불필요한 밉맵 비활성화**: 밉맵을 사용하면 카메라로부터의 거리에 따라 렌더링해야 하는 디테일을 줄여 성능을 최적화할 수 있지만, 항상 필요하지는 않습니다. 2D 스프라이트와 UI 그래픽스처럼 화면상에 일정한 크기로 유지되는 텍스처에는 사용하지 않아도 됩니다. 카메라와의 거리가 달라지는 3D 모델은 밉맵을 활성화된 상태로 유지하세요.



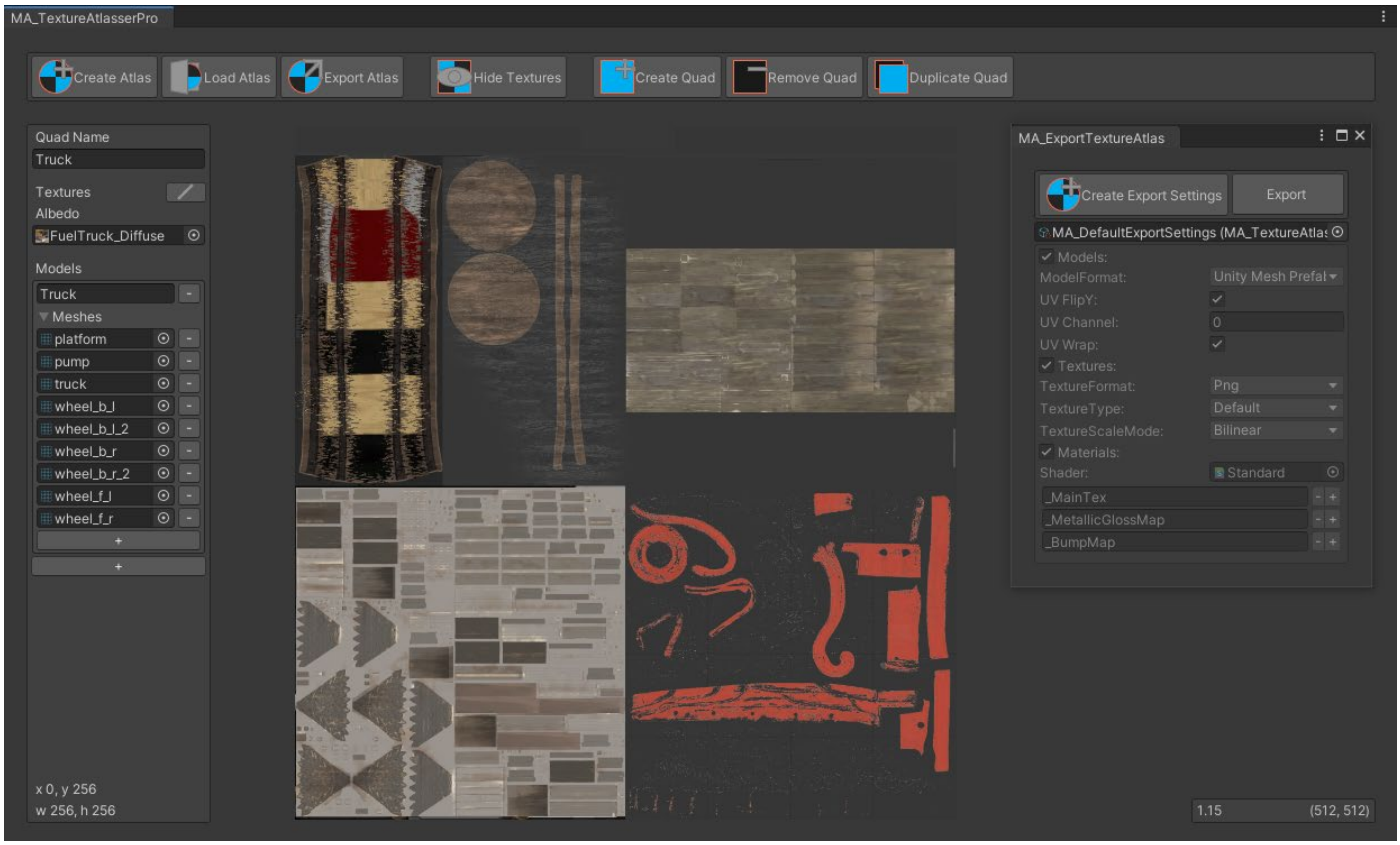
빌드 크기를 최적화하는 데 도움이 되는 적절한 텍스처 임포트 설정

텍스처 아틀라싱

아틀라싱은 여러 개의 작은 텍스처를 균일한 크기의 큰 텍스처 하나로 그룹화하는 프로세스입니다. 텍스처를 아틀라싱하면 드로우 콜을 더 적게 사용하므로 콘텐츠를 그리는 데 드는 GPU 리소스를 줄일 수 있으며, 메모리 사용량도 줄일 수 있습니다.

2D 프로젝트의 경우, 개별 스프라이트와 텍스처를 렌더링하는 대신 스프라이트 아틀라스(**Asset > Create > 2D > Sprite Atlas**)를 사용할 수 있습니다.

3D 프로젝트의 경우, 원하는 DCC(디지털 콘텐츠 제작) 패키지를 사용할 수 있습니다. MA_TextureAtlasser나 TexturePacker 등의 여러 타사 툴을 사용해서 텍스처 아틀라스를 제작할 수도 있습니다.



텍스처 아틀라스를 사용하여 드로우 콜을 줄입니다.

고해상도 맵이 필요하지 않은 3D 지오메트리의 경우 텍스처를 합치고 UV를 다시 매핑하면 됩니다. 비주얼 에디터를 사용하면 텍스처 아틀라스나 스프라이트 시트에서 크기와 위치를 설정하고 우선순위를 정할 수 있습니다.

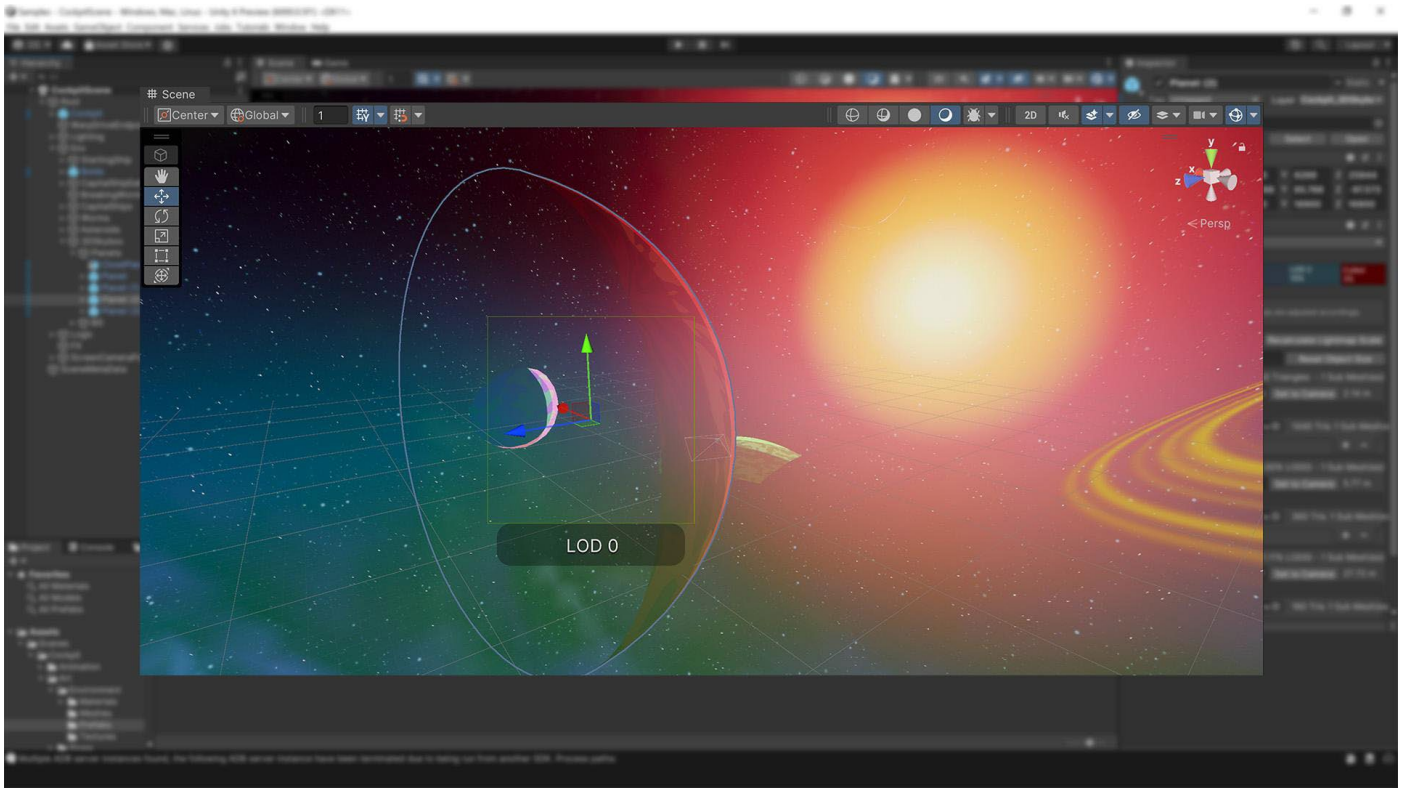
Texture Packer는 여러 개별 맵을 하나의 큰 텍스처로 통합합니다. 그런 다음 Unity에서 드로우 콜을 한 번만 보내면 성능 오버헤드를 줄이면서 패킹된 텍스처에 액세스할 수 있습니다.

폴리곤 개수 확인

해상도가 높은 모델은 메모리 사용량이 더 많고 잠재적으로 GPU 시간이 더 길 수 있습니다. 하지만 백그라운드 지오메트리에 폴리곤이 100만 개나 필요한 경우는 거의 없을 것입니다.

씬에서 게임 오브젝트의 지오메트리 복잡도를 최소 수준으로 유지하세요. 지오메트리가 너무 복잡하면 Unity가 그래픽 카드에 엄청난 양의 버텍스 데이터를 푸시해야 합니다.

DCC 패키지에서 모델 수를 줄일 수 있다면 줄이는 것이 좋습니다. 또한 카메라 시야에 보이지 않는 폴리곤은 삭제하세요. 예를 들어 벽에 붙어 있는 찬장의 뒷면을 볼 일이 전혀 없다면 모델에서 찬장의 뒷면을 제거해야 합니다.



모델을 최적화하려면 보이지 않는 면을 제거해야 합니다.

최신 GPU에서의 병목 현상은 일반적으로 폴리곤의 개수가 아니라 폴리곤의 밀도 때문에 발생합니다. 멀리 있는 오브젝트의 폴리곤 수를 줄이려면 모든 에셋에 아트 전달을 수행하는 것이 좋습니다. 작은 삼각형은 GPU 성능 저하의 주요 원인이 될 수 있습니다.

타겟 플랫폼에 따라 고해상도 텍스처를 통해 디테일을 추가해서 로우 폴리곤 지오메트리를 보완할 방법을 찾아보세요. 메시 밀도를 높이는 대신 텍스처와 노멀 맵을 사용하는 것이 좋습니다.

텍스처에 최대한 많은 디테일을 베이크하여 픽셀 복잡도를 줄이세요. 예를 들어 프래그먼트 셰이더에서 하이라이트를 계산하지 않도록 스페큘러 하이라이트를 텍스처로 캡처할 수 있습니다.

이러한 기술은 성능에 영향을 줄 수 있으며 타겟 플랫폼에 적합하지 않을 수도 있으니 정기적으로 프로파일링해야 합니다.

메시 임포트 설정

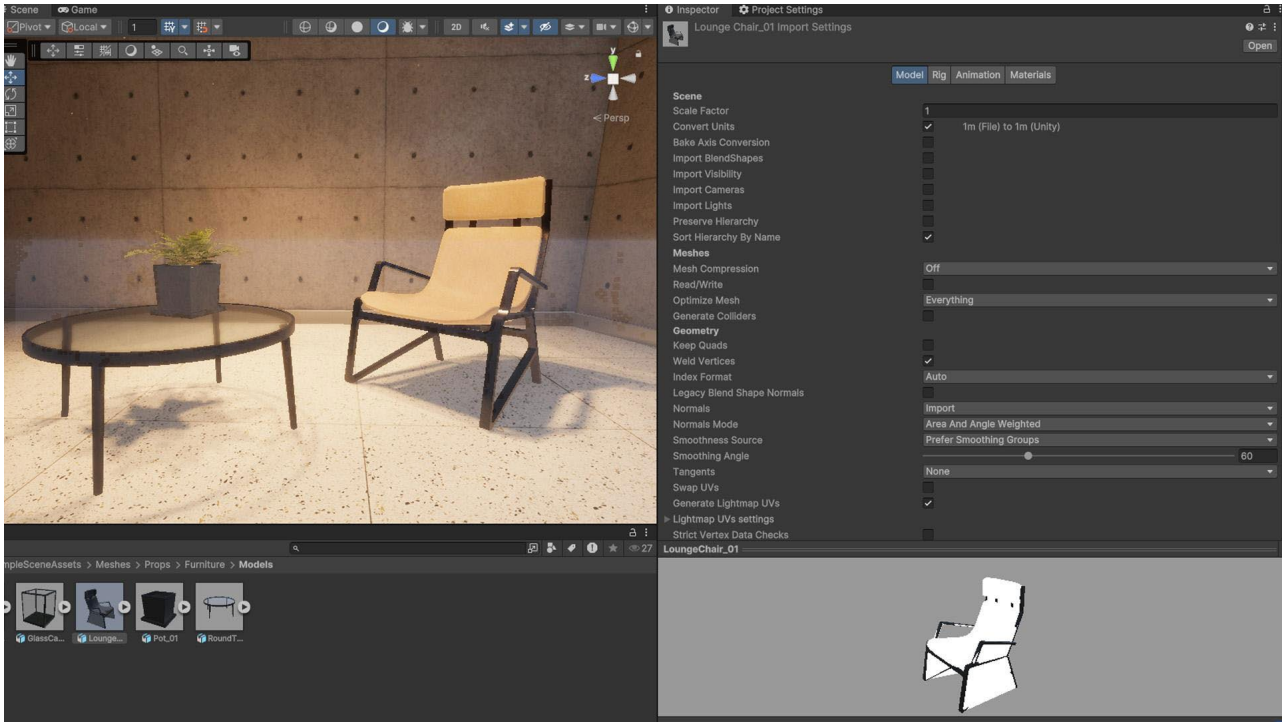
텍스처와 마찬가지로 메시도 신중하게 임포트하지 않으면 메모리를 과도하게 사용할 수 있습니다. 메시의 메모리 사용량을 최소화하려면 다음 지침을 따르세요.

— **메시 압축 사용**: 메시 압축을 적극적으로 사용하여 디스크 공간을 줄일 수 있습니다. 하지만 런타임 시 메모리에는 영향을 주지 않습니다. 메시 양자화로 인해 정확성이 떨어질 수 있으므로 압축 수준을 다양하게 조절해 보고 모델에 적합한 수준을 파악하시기 바랍니다.

— **Read/Write 비활성화**: 이 옵션을 활성화하면 메모리에서 메시가 복제되며, 메시 사본 중 하나는 시스템 메모리,다른 하나는 GPU 메모리에 유지됩니다. 대부분의 경우에는 이 옵션을 비활성화해야 합니다. Unity 2019.2 이하에서는 이 옵션이 기본으로 선택되어 있습니다.

— **Rig 및 BlendShape 비활성화**: 메시에 골격 또는 블렌드 셰이프 애니메이션이 필요하지 않다면 이 옵션을 비활성화합니다.

— **Normal과 Tangent 비활성화**: 메시의 머티리얼에 노멀 또는 탄젠트가 필요하지 않은 것이 확실하다면, 이 옵션을 선택 해제하여 추가로 메모리를 절감할 수 있습니다.



메시 임포트 설정 확인

기타 메시 최적화

Player Settings에서 메시에 다른 몇 가지 최적화를 적용할 수 있습니다.

— **Vertex Compression**은 채널당 버텍스 압축을 설정합니다. 예를 들어 포지션 및 라이트맵 UV를 제외한 모든 것을 압축하여 메시의 런타임 메모리 사용량을 줄일 수 있습니다.

각 메시의 Import Settings에 있는 Mesh Compression은 버텍스 압축 설정을 오버라이드합니다. 이 경우 메시의 런타임 복사본이 압축되지 않고 더 많은 메모리를 사용할 수도 있습니다.

— **Optimize Mesh Data**는 탄젠트, 노멀, 컬러, UV 등 메시에 적용된 머티리얼에 필요하지 않은 데이터를 메시에서 제거합니다.

에셋 감사

에셋 감사 과정을 자동화하면 실수로 인한 에셋 설정 변경을 방지할 수 있습니다. 임포트 설정을 표준화하거나 기존 에셋을 분석할 수 있는 몇 가지 툴을 사용하면 도움이 됩니다.

AssetPostprocessor

에셋을 임포트하기 전이나 임포트할 때 AssetPostprocessor로 임포트 파이프라인에 연결하여 스크립트를 실행할 수 있습니다. 이렇게 하면 프리셋과 유사하지만 코드를 통해 모델, 텍스처, 오디오 등의 임포트 전후로 설정을 커스터마이즈하라는 메시지가 표시됩니다. 이 프로세스에 대해 자세히 알아보려면 GDC 2023 발표 '게임 개발의 모든 단계에서 활용할 수 있는 기술 팁'을 시청해 보세요.

UnityDataTools

UnityDataTools는 유니티가 Unity 프로젝트의 데이터 관리 및 직렬화 기능을 향상하기 위해 제공하는 각종 오픈 소스 툴입니다. Unity DataTools에는 사용되지 않는 에셋을 식별하거나, 에셋 종속 관계를 감지하거나, 빌드 크기를 줄이는 등 프로젝트를 분석하고 최적화하기 위한 기능이 포함되어 있습니다.

이 툴에 대한 자세한 내용은 여기에서 알아볼 수 있으며, 베스트 프랙티스 가이드의 Unity에서의 최적화에 대한 이해 섹션에서 에셋 감사에 대해 자세히 알아볼 수 있습니다.

비동기 텍스처 버퍼

Unity는 링 버퍼를 사용하여 텍스처를 GPU로 푸시합니다. QualitySettings.asyncUploadBufferSize를 통해 비동기 텍스처 버퍼를 수동으로 조정할 수 있습니다.

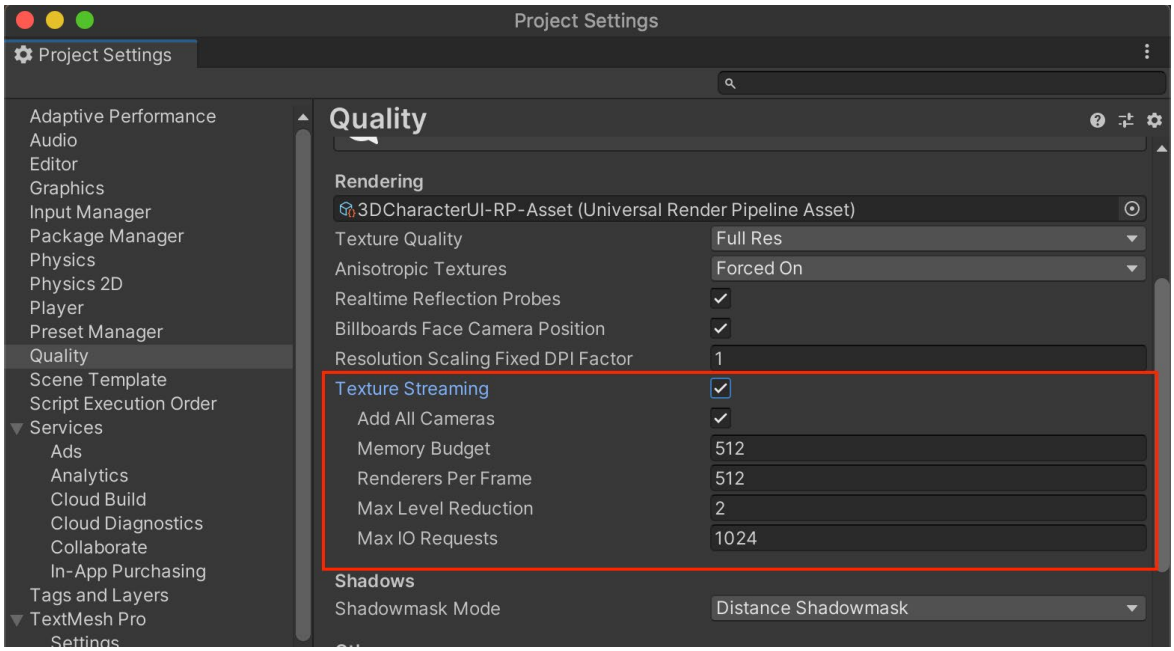
업로드 속도가 너무 느리거나 여러 텍스처를 한 번에 로드하는 도중에 메인 스레드가 멈춘다면 텍스처 버퍼를 조정하세요. 일반적으로는 씬에서 로드해야 하는 가장 큰 텍스처 크기(단위: MB)로 값을 설정할 수 있습니다.

참고: 기본값을 변경하면 메모리에 상당한 부담을 줄 수 있습니다. 또한 Unity가 할당한 링 버퍼 메모리는 이후 시스템에 반환할 수 없습니다. GPU 메모리가 과부하 상태가 되면 GPU는 가장 오랫동안 사용되지 않은 텍스처를 언로드하고, 다음에 해당 텍스처가 카메라 절두체에 들어갈 때 CPU가 텍스처를 다시 업로드하게 합니다.

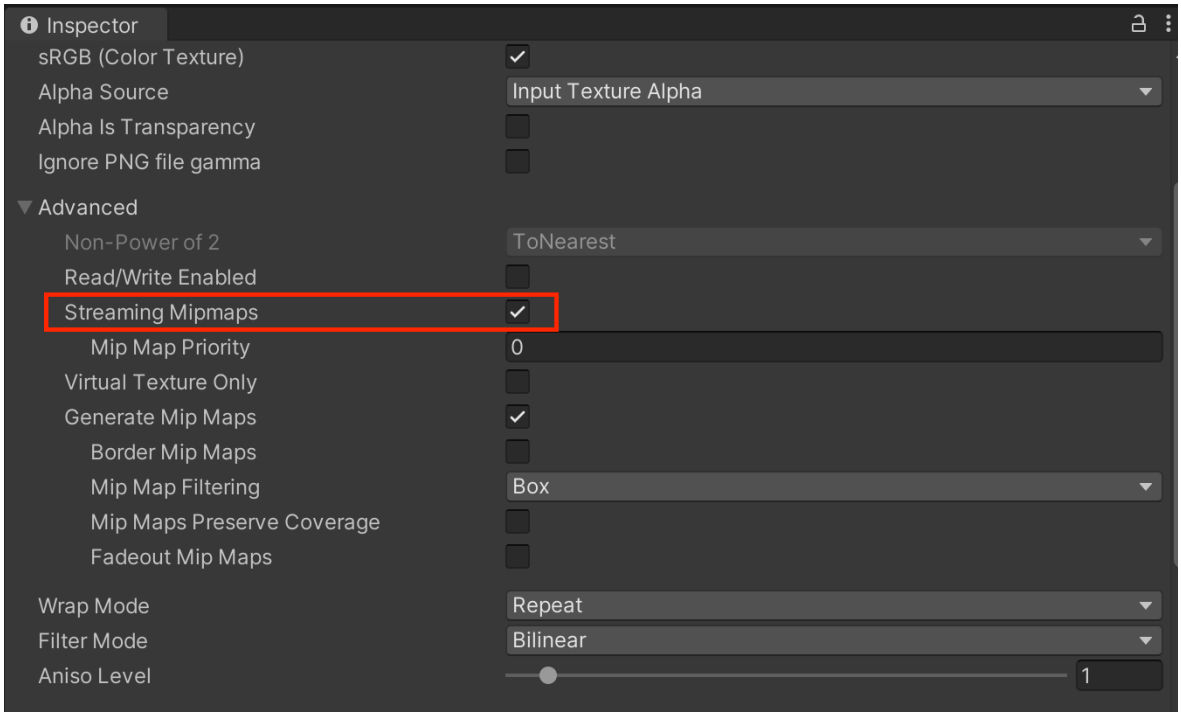
Unity에서의 메모리 관리(영문) 가이드에서 시간 분할 Awake를 사용할 때의 텍스처 버퍼 메모리 제한에 대해 자세히 알아보세요.

밉맵 및 텍스처 스트리밍

밉맵 스트리밍 시스템을 사용하면 메모리에 로드되는 밉맵 수준을 제어할 수 있습니다. 밉맵 스트리밍을 사용하려면 Quality Settings(**Edit > Project Settings > Quality**)로 이동하여 **Texture Streaming**을 선택합니다. 그런 다음 텍스처 Import Settings의 **Advanced** 아래에 있는 **Streaming Mipmaps**를 활성화합니다.



텍스처 스트리밍 설정



Streaming Mipmaps가 활성화된 모습

이 시스템은 현재 카메라 위치를 렌더링하는 데 필요한 밉맵만 로드하므로 텍스처에 필요한 메모리 양을 줄여 줍니다. 밉맵 스트리밍 시스템을 사용하지 않으면 Unity는 기본적으로 모든 텍스처를 로드합니다. 텍스처 스트리밍은 적은 양의 CPU 리소스를 사용하여 잠재적으로 많은 양의 GPU 메모리를 절약합니다.

추가적인 제어가 필요하다면 밉맵 스트리밍 API를 사용하면 됩니다. 텍스처 스트리밍은 자동으로 밉맵 수준을 줄여 사용자가 정의한 메모리 할당량 내로 유지합니다.

어드레서블 사용

어드레서블 에셋 시스템으로 게임을 구성하는 에셋을 간편하게 관리할 수 있습니다. 씬, 프리팹, 텍스트 에셋 등을 포함한 모든 에셋을 '어드레서블'로 표시하고 고유한 이름을 별칭으로 지정하여 어디서든 호출할 수 있습니다.

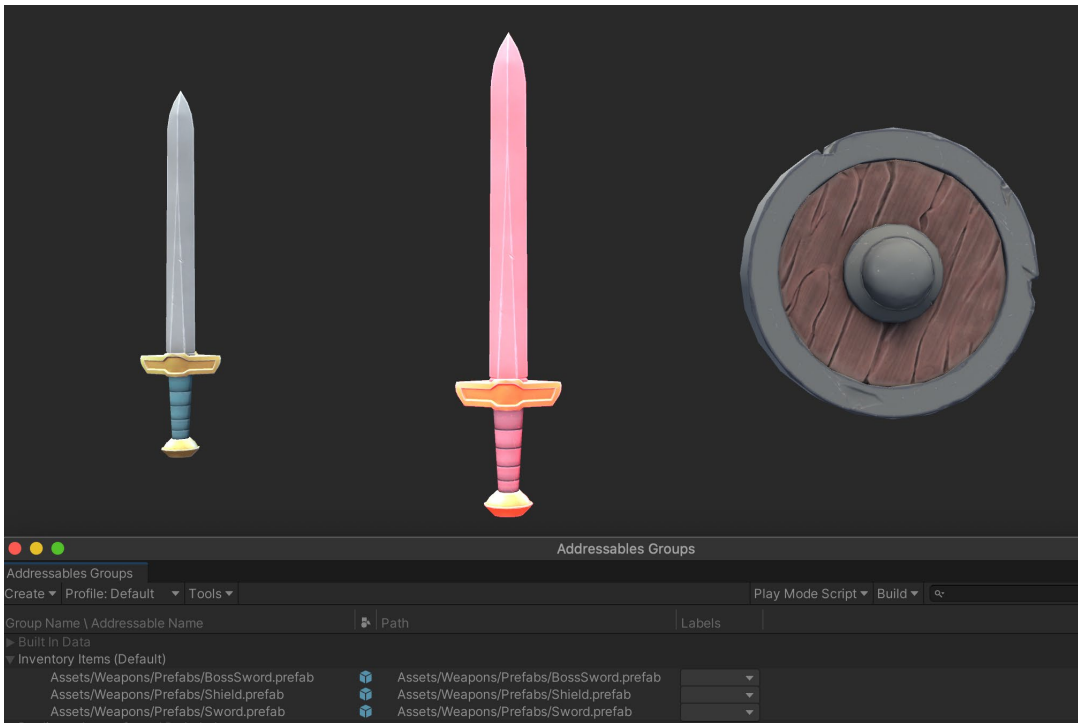
스탠드얼론 및 콘솔 기기를 비롯한 플랫폼에 개발 중인 게임이라면 Addressable Asset Groups에서 다음 기본 설정을 검토해 보세요.

— AssetBundle Compression을 LZ4로 설정하는 것은 콘솔에서 불필요하며(파일 시스템에 더 나은 압축 시스템이 있음), 대형 번들에서 작은 변경 사항을 효율적으로 패치할 수 없게 만들기도 합니다. 콘솔 플랫폼에서는 Uncompressed로 설정하고 Windows에서는 LZ4로 설정하세요.

— 콘솔 플랫폼에서 AssetBundle CRC를 Enabled, Including Caching으로 설정하면 로딩 시간이 불필요하게 많이 늘어납니다. 로컬 파일 시스템을 신뢰할 수 있는 콘솔 플랫폼에서는 Disabled로 설정해야 합니다. Windows에서는 파일 변조가 문제가 될 수 있는 경우 Enabled, Including Caching으로 설정하세요.

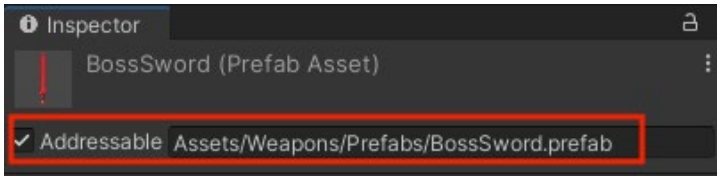
— Bundle Naming Mode 설정을 Append Hash to Filename으로 설정하면 '동일한' 파일의 이름 변경을 인식하지 못하는 콘솔의 패치 시스템에서 올바르게 동작하지 않습니다. 콘솔 플랫폼에서는 Filename으로 설정해야 합니다.

게임과 게임의 에셋 사이에 추상화 계층을 추가하면, 다운로드 가능한 별도의 콘텐츠 팩 제작과 같은 특정 작업을 간소화할 수 있습니다. 어드레서블을 사용하면 에셋 팩을 로컬로든, 원격으로든 쉽게 참조할 수 있습니다.



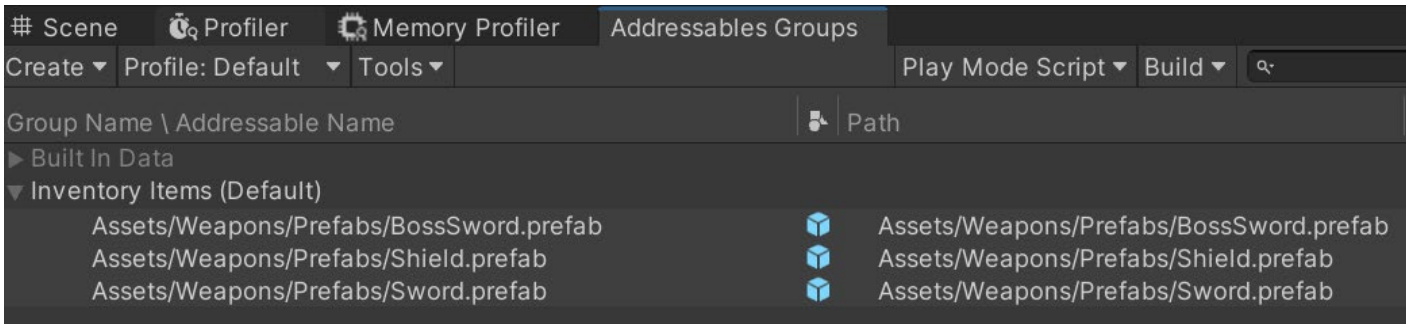
이 예시에서는 어드레서블이 프리팹 인벤토리를 추적합니다.

어드레서블 패키지는 패키지 관리자에서 설치할 수 있습니다. 프로젝트의 각 에셋이나 프리팹은 모두 '어드레서블'이 될 수 있습니다. 인스펙터에서 에셋 이름 아래에 있는 옵션을 선택하여 기본 고유 주소를 할당합니다.



기본 Addressable Name으로 활성화된 어드레서블 옵션

옵션을 선택하면 해당 에셋이 **Window > Asset Management > Addressables > Groups** 창에 나타납니다.



각 에셋의 커스텀 주소와 위치를 확인할 수 있는 Addressables Groups 화면

다른 곳에서 호스팅된 에셋이든 로컬로 저장된 에셋이든, 시스템에서는 Addressable Name 문자열을 사용하여 에셋을 찾을 수 있습니다. 필요한 상태가 되기 전까지는 어드레서블 프리팹이 메모리에 로드되지 않으며, 관련 에셋을 더 이상 사용하지 않을 때 자동으로 언로드할 수 있습니다.

다음 두 블로그 게시물에서 어드레서블에 대해 자세히 알아보세요.

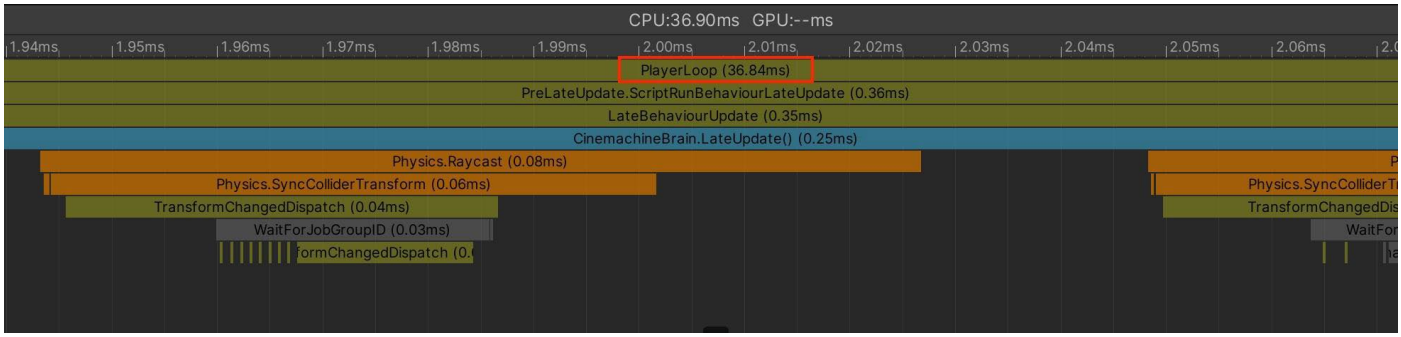
— 어드레서블 에셋 시스템으로 메모리 최적화하기에서는 Addressables Groups를 구성하여 메모리 효율성을 높이는 방법의 예시를 볼 수 있습니다.

— Addressables: 계획 및 베스트 프랙티스에서는 어드레서블 에셋을 구성, 제작, 로드, 언로드하는 방법에 관한 팁을 확인할 수 있습니다.

프로그래밍 및 코드 아키텍처

Unity PlayerLoop에는 게임 엔진의 코어와 상호 작용하기 위한 함수가 포함되어 있습니다. 이 구조에는 초기화와 프레임별 업데이트를 처리하는 여러 시스템이 포함되어 있습니다. 모든 스크립트가 이 PlayerLoop를 활용하여 게임플레이를 생성하게 됩니다.

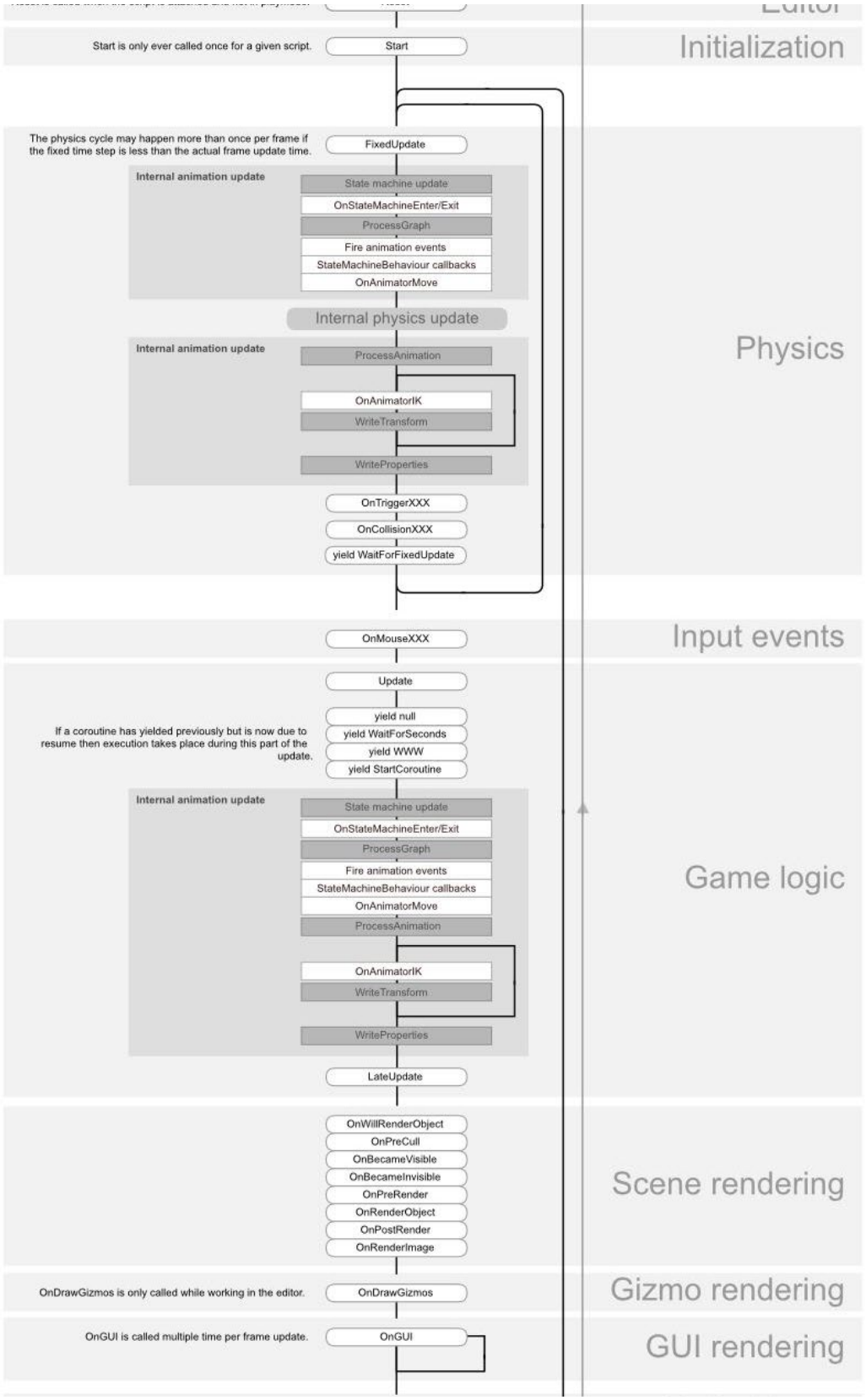
프로파일링 시에는 PlayerLoop 아래에 프로젝트의 사용자 코드가 표시됩니다(EditorLoop 아래에는 에디터 컴포넌트).



전체 엔진 실행의 컨텍스트에서 커스텀 스크립트, 설정, 그래픽스를 보여 주는 프로파일러

Unity PlayerLoop 이해하기

Unity 프레임 루프의 실행 순서를 이해해야 합니다. 모든 Unity 스크립트는 사전에 정해진 순서대로 여러 이벤트 함수를 실행합니다. **Awake**, **Start**, **Update** 및 스크립트의 라이프사이클을 생성하는 다른 함수들 사이의 차이점을 이해해야 합니다. 하위 수준 API를 활용하여 플레이어의 Update 루프에 커스텀 로직을 추가할 수 있습니다.



PlayerLoop 및 스크립트의 라이프사이클 파악

매 프레임에 실행되는 코드 최소화하기

코드를 반드시 모든 프레임에 실행해야 하는지 확인하세요. 불필요한 로직을 Update, LateUpdate, FixedUpdate에서 제외하세요. 이러한 이벤트 함수에는 프레임마다 업데이트해야 하는 코드를 편리하게 배치할 수 있으며, 같은 빈도로 업데이트할 필요가 없는 로직은 추출됩니다. 가능하다면 상황이 바뀌는 경우에만 로직을 실행하세요.

반드시 **Update**를 사용해야 한다면 **n개** 프레임마다 코드를 실행하는 방안을 검토해 보세요. 이는 여러 프레임에 대규모 워크로드를 분산하는 일반적인 기법인 타임 슬라이싱(time slicing) 방식 중 하나이기도 합니다. 이 예에서는 3개 프레임마다 한 번씩 **ExampleExpensiveFunction**을 실행합니다.

```
private int interval = 3;
void Update()
{
    if (Time.frameCount % interval == 0)
    {
        ExampleExpensiveFunction();
    }
}
```

ExampleExpensiveFunction이 데이터 세트에 대한 연산을 수행하는 경우, 타임 슬라이싱을 활용하여 프레임마다 해당 데이터의 다른 하위 세트에서 연산을 수행하는 것을 고려해 보세요. n개 프레임마다 모든 작업을 수행하는 대신, 모든 프레임마다 1/n의 작업을 수행하면 주기적인 CPU 스파이크 없이 전반적으로 더 안정적이고 예측 가능한 성능이 발휘됩니다.

이 함수를 다른 프레임에서 실행되는 다른 작업과 상호 배치해야 한다는 점을 기억하세요. 이 예에서 **Time.frameCount % interval == 1** 또는 **Time.frameCount % interval == 2**일 때 리소스가 많이 소모되는 다른 함수를 예약할 수 있습니다.

또는 커스텀 UpdateManager 클래스(아래)를 사용하고 n개 프레임마다 등록된 오브젝트를 업데이트할 수 있습니다.

Start/Awake에서 대규모 로직 사용 방지

첫 번째 씬을 로드하면 다음과 같은 함수가 각 오브젝트에 대해 호출됩니다.

- **Awake**
- **OnEnable/OnDisable**
- **Start**

애플리케이션이 첫 번째 프레임을 렌더링하기 전까지는 이러한 함수에서 비용이 많이 드는 로직을 될 수 있으면 사용하지 마세요. 그렇게 하지 않으면 필요 이상으로 로딩 시간이 길어질 수 있습니다.

첫 번째 씬 로드에 대한 자세한 내용은 이벤트 함수의 실행 순서를 참조하세요.

많은 리소스를 소모하는 함수 결과 캐싱

일반적으로 Update 메서드에서 호출하지 않도록 Awake 또는 Start에서 레퍼런스를 캐싱하는 방법이 제일 바람직합니다.

이러한 메서드를 자주 호출하면 CPU 스파이크가 발생할 수 있습니다. 가능하다면 초기화 단계에서 리소스 비용이 많이 드는 함수(예: MonoBehaviour.Awake와 MonoBehaviour.Start)를 실행해야 합니다. 필요한 레퍼런스를 캐싱하고 나중에 다시 사용하세요.

다음 예는 반복적인 GetComponent 호출의 비효율성을 보여 줍니다.

```
void Update()
{
    Renderer myRenderer = GetComponent<Renderer>();
    ExampleFunction(myRenderer);
}
```

함수의 결과가 캐싱되므로 **GetComponent**는 한 번만 호출하세요. **Update**에서 **GetComponent**를 추가로 호출하지 않아도 캐싱된 결과를 재사용할 수 있습니다.

```
private Renderer myRenderer;
void Start()
{
    myRenderer = GetComponent<Renderer>();
}
void Update()
{
    ExampleFunction(myRenderer);
}
```

Unity 2020.2 이전 버전에서는 **GameObject.Find, GameObject.GetComponent**, **Camera.main**에 비용이 많이 들었지만 이제는 그렇지 않습니다. 그렇긴 하지만 **Update** 메서드에서 호출하는 것을 피하고 결과를 캐싱하여 위 방법을 따르는 것이 가장 좋습니다.

빈 Unity 이벤트 함수 방지

빈 MonoBehaviour에도 리소스가 필요하므로 비어 있는 Update 또는 LateUpdate 메서드는 제거해야 합니다.

테스트에 이러한 메서드를 사용하고 있다면 다음 프리 프로세서 지시문을 사용하세요.

```
#if UNITY_EDITOR
void Update()
{
}
```

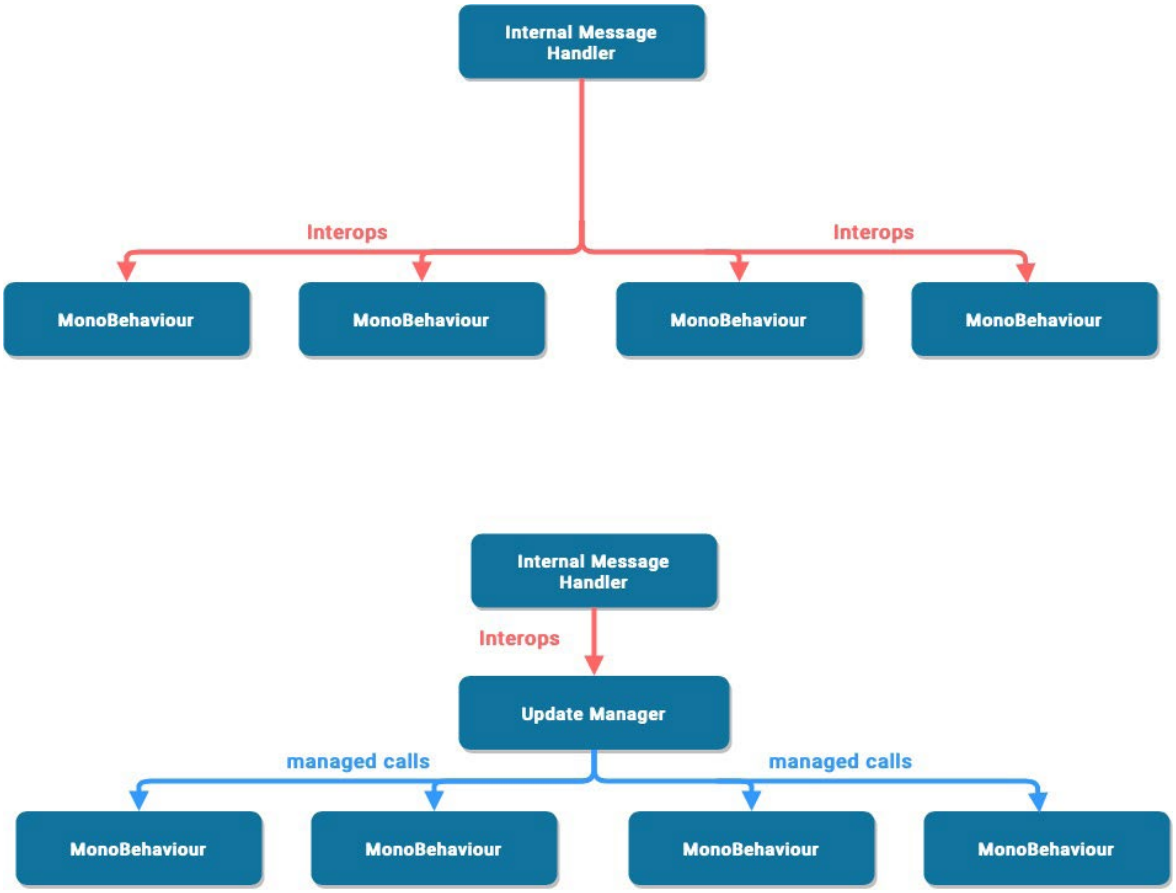
빌드에 불필요한 오버헤드가 유입되는 일 없이 에디터에서 Update를 자유롭게 사용하여 테스트할 수 있습니다.

커스텀 Update Manager 만들기

Update 또는 LateUpdate의 일반적인 사용 패턴은 일부 조건을 충족하는 경우에만 로직을 실행하는 것입니다. 그러면 조건을 확인하는 것을 제외하고는 효율적으로 코드를 실행하지 않는 프레임별 콜백이 많이 발생할 수 있습니다.

Unity는 Update나 LateUpdate 같은 Message 메서드를 호출할 때마다 C/C++에서 관리되는 C#으로 호출하는 **interop** 호출을 수행합니다. 오브젝트의 수가 적다면 문제가 되지 않는데, 오브젝트가 수천 개에 달하면 이 오버헤드 문제가 심각해지기 시작합니다.

이러한 방식으로 Update 또는 LateUpdate를 사용하는 대형 프로젝트(예: 오픈 월드 게임)에서는 커스텀 UpdateManager를 제작하는 것이 좋습니다. 활성 오브젝트에 콜백이 필요한 경우 커스텀 UpdateManager에 등록하고, 필요하지 않은 경우 등록을 취소합니다. 이러한 패턴을 사용하면 MonoBehaviour 오브젝트에 대한 많은 interop 호출을 줄일 수 있습니다.



interop 호출을 줄일 수 있는 커스텀 Update Manager 만들기

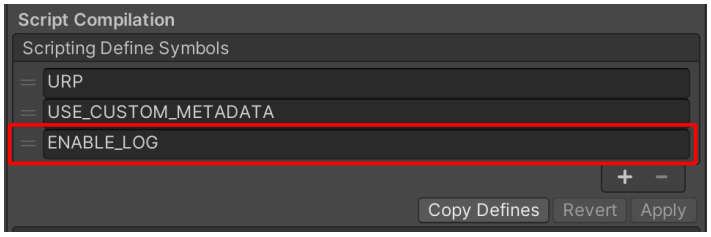
구현 예시와 잠재적 성능 개선 예시는 Unity에서의 게임 엔진별 최적화 기술(영문)을 참조하시기 바랍니다.

Debug Log 구문 제거하기

Log 구문, 특히 Update, LateUpdate 또는 FixedUpdate에 있는 Log 구문은 성능을 저해할 수 있습니다. 빌드를 만들기 전에 Log 구문을 비활성화하는 것이 좋습니다.

더 쉽게 비활성화하려면 프리 프로세서 지시문과 함께 조건부 속성을 만들어 보세요. 예를 들어 다음과 같은 커스텀 클래스를 만듭니다.

```
public static class Logging
{
    [System.Diagnostics.Conditional("ENABLE_LOG")]
    static public void Log(object message)
    {
        UnityEngine.Debug.Log(message);
    }
}
```



커스텀 프리 프로세서 지시문을 추가하여 스크립트 분리

커스텀 클래스로 로그 메시지를 생성합니다. **Player Settings > Scripting Define Symbols**에서 **ENABLE_LOG** 프리 프로세서를 비활성화하면 모든 Log 구문이 동시에 사라집니다.

Debug.DrawLine 및 Debug.DrawRay와 같은 다른 Debug 클래스의 활용 사례에도 동일하게 적용됩니다. 이러한 기능도 개발 단계에서만 사용하는 것을 전제로 설계되었으며 성능에 큰 영향을 미칠 수 있습니다.

문자열 및 텍스트 처리는 Unity 프로젝트에서 성능 문제를 유발하는 주요 원인입니다. Log 구문과 리소스 소모가 많은 문자열 포맷만 제거해도 성능 문제를 크게 개선할 수 있습니다.

Stack Trace 로깅 비활성화

Player Settings에서 Stack Trace 옵션을 사용하면 표시되는 로그 메시지 유형을 정할 수 있습니다.

애플리케이션이 릴리스 빌드에서 오류나 경고 메시지를 로깅하는 경우(예: 실제 실행 시 충돌 보고서를 생성하기 위해) Stack Trace를 비활성화하면 성능을 개선할 수 있습니다.

Stack Trace 옵션

문자열 파라미터 대신 해시 값 사용하기

Unity는 내부적으로 애니메이터나 머티리얼, 셰이더 프로퍼티를 식별할 때 문자열 이름을 사용하지 않습니다. 빠른 처리를 위해 모든 프로퍼티 이름이 프로퍼티 ID에 해시되어 있으며, 해당 ID가 실제로 프로퍼티를 식별하는 데 사용됩니다.

애니메이터, 머티리얼 또는 셰이더에서 Set 또는 Get 메서드를 사용하는 경우에는 문자열 값 메서드 대신 정수 값 메서드를 사용하세요. 문자열 메서드 역시 문자열 해싱을 수행한 다음 해시된 ID를 정수 값 메서드로 전달하기 때문입니다.

애니메이터 프로퍼티 이름에 Animator.StringToHash를, 머티리얼 및 셰이더 프로퍼티 이름에 Shader.PropertyToID를 사용하세요. 초기화 단계에서 해시 값을 가져오고 변수에 캐싱하여 Get 또는 Set 메서드에 전달할 수 있도록 준비하세요.

올바른 데이터 구조 선택

한 번 선택한 데이터 구조는 프레임당 수천 번씩 반복되므로 효율성에 영향을 줍니다. 컬렉션에 List, Array 또는 Dictionary 중 무엇을 사용해야 할지 고민하시나요? 올바른 구조를 선택하기 위한 일반적인 가이드인 C#의 데이터 구조 MSDN 가이드를 참고하세요.

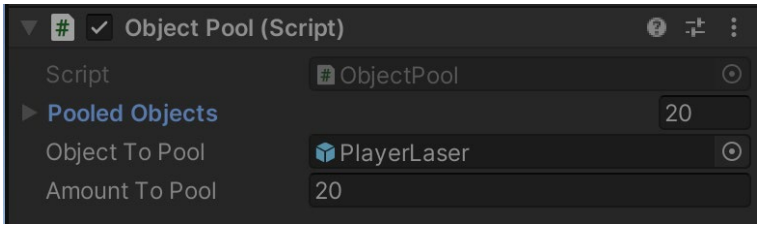
런타임 시 컴포넌트 추가 방지

런타임에 AddComponent를 호출하면 비용이 발생합니다. 런타임에 컴포넌트가 추가될 때마다 Unity가 중복 또는 기타 필요한 컴포넌트를 확인해야 하기 때문입니다.

이미 설정된 원하는 컴포넌트로 프리팹을 인스턴스화하는 것이 일반적으로 더 효과적입니다.

오브젝트 풀 사용하기

Instantiate 함수 및 Destroy 함수는 가비지와 가비지 컬렉션(GC) 스파이크를 야기하며, 일반적으로 속도가 느린 프로세스입니다. 오브젝트를 많이 인스턴스화해야 하는 경우 GC 스파이크를 피하기 위해 오브젝트 풀링 기법을 적용하세요.

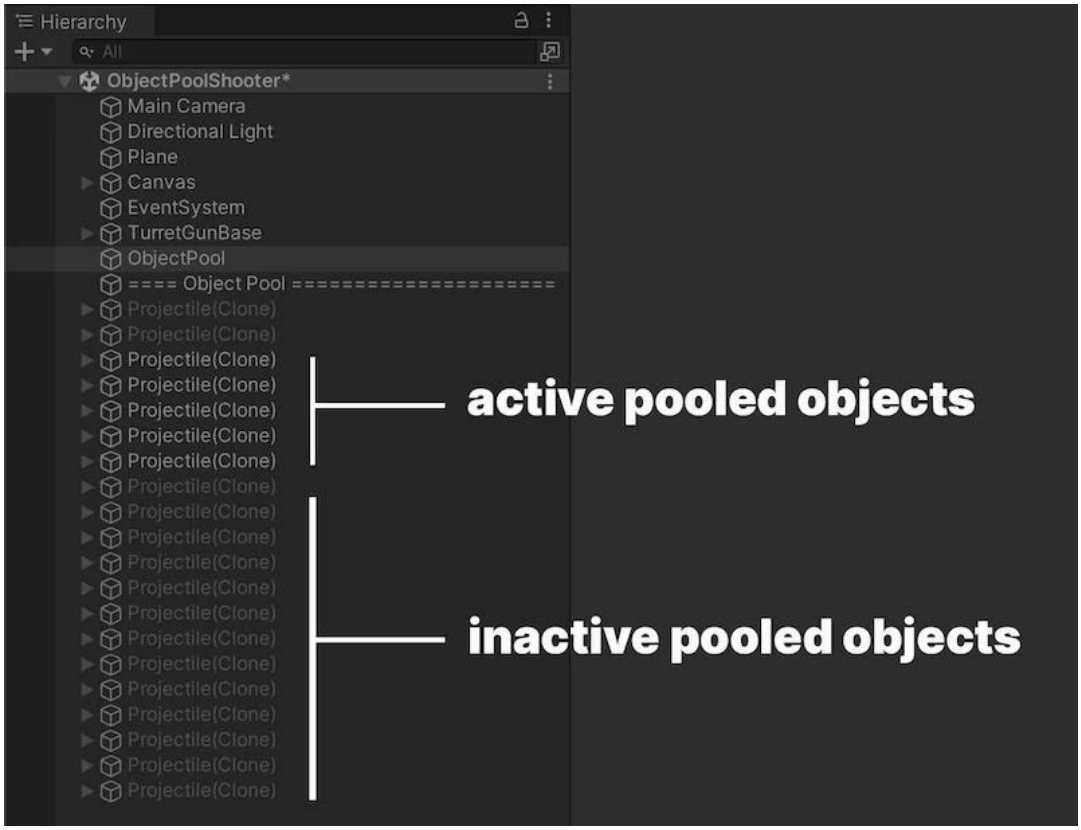


재사용 가능한 PlayerLaser 인스턴스 20개를 생성하는 ObjectPool의 예

오브젝트 풀링은 CPU가 Create 및 Destroy 호출을 반복하는 데 필요한 프로세싱 파워를 줄여 성능을 최적화할 수 있는 디자인 패턴입니다. 오브젝트 풀링을 활용하면 호출을 반복하는 대신 기존 게임 오브젝트를 계속 재사용할 수 있습니다.

오브젝트 풀링의 핵심은 오브젝트를 필요할 때마다 생성하고 삭제하는 대신, 사전에 생성하여 풀에 저장하는 것입니다. 오브젝트가 필요하면 풀에서 가져와 사용할 수 있습니다. 더 이상 필요하지 않은 오브젝트는 삭제하는 대신 풀로 돌려보내면 됩니다.

총알을 발사하는 등의 상황에서 게임 오브젝트를 계속 인스턴스화하고 삭제하기보다는, 재사용 및 재활용할 수 있는 사전 할당 오브젝트 풀을 사용해 보세요.



이렇게 하면 프로젝트에서 관리되는 할당의 수가 줄어들기 때문에 가비지 컬렉션 문제를 방지할 수 있습니다.

Unity에는 UnityEngine.Pool 네임스페이스를 통하는 빌트인 오브젝트 풀링 기능이 있습니다. Unity 2021 LTS 이상 버전에서 사용 가능한 이 네임스페이스로 오브젝트 라이프사이클이나 풀 크기 제어 같은 측면을 자동화하여 오브젝트 풀을 용이하게 관리할 수 있습니다.

Unity에서 간단한 오브젝트 풀링 시스템을 만드는 방법은 여기에서 확인할 수 있습니다. 또한 Unity 에셋 스토어에서 다운로드할 수 있는 이 샘플 프로젝트에서 오브젝트 풀링 패턴을 비롯한 여러 시스템이 구현된 Unity 씬을 살펴볼 수 있습니다.

트랜스폼 한 번에 이동

트랜스폼을 옮길 때 Transform.SetPositionAndRotation을 사용하면 위치와 회전을 한 번에 업데이트할 수 있습니다. 그러면 트랜스폼을 두 번 수정함으로 인해 발생하는 오버헤드를 방지할 수 있습니다.

런타임에 게임 오브젝트를 인스턴스화해야 한다면 다음과 같이 단순한 최적화로 인스턴스화 중에 부모 자식 관계를 설정하고 다시 포지셔닝할 수 있습니다.

```
GameObject.Instantiate(prefab, parent);
GameObject.Instantiate(prefab, parent, position, rotation);
```

Object.Instantiate에 대한 자세한 정보는 스크립팅 API를 참고하세요.

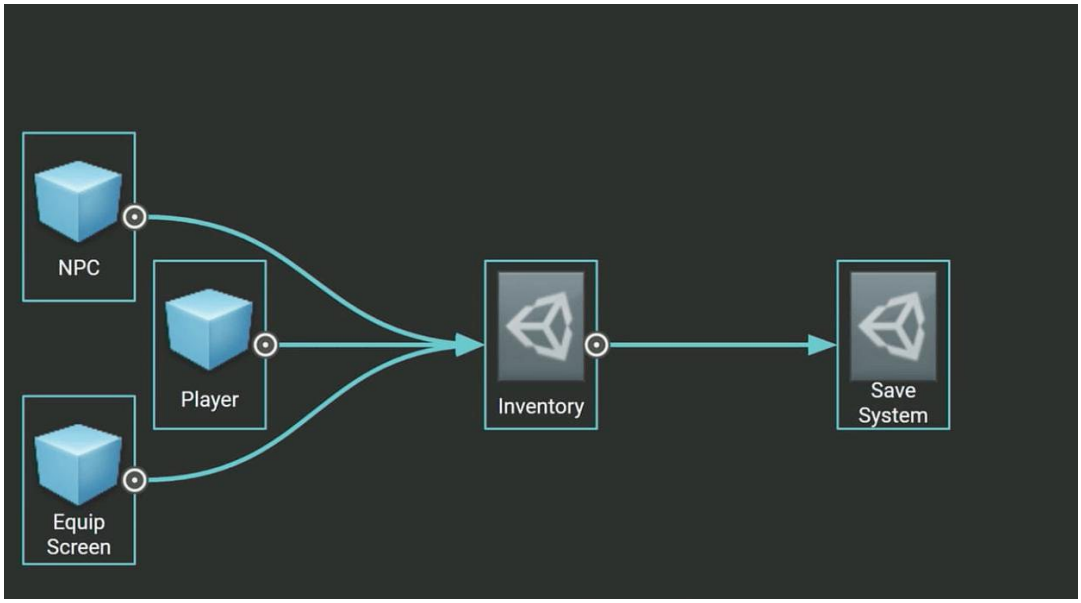
ScriptableObject 사용하기

변하지 않는 정적 값 또는 설정은 MonoBehaviour가 아닌 스크립터블 오브젝트에 저장하세요. 스크립터블 오브젝트는 한 번만 설정하면 되는 프로젝트 내부의 에셋으로

MonoBehaviour가 호스트 역할을 하려면 게임 오브젝트가 필요하므로(그리고 기본적으로 트랜스폼) 더 큰 오버헤드를 유발합니다. 따라서 하나의 값을 저장하기 전에 사용되지 않는 다량의 데이터를 생성해야 합니다. ScriptableObject는 게임 오브젝트와 트랜스폼이 필요하지 않으므로 메모리 사용을 줄일 수 있습니다. 또한 프로젝트 수준에서 데이터를 저장하므로 여러 씬에서 동일한 데이터를 쉽게 액세스할 수 있습니다.

런타임에 변경되지 않는 동일한 중복 데이터를 활용하는 게임 오브젝트가 많은 경우에 일반적으로 사용하는 방법입니다. 각 게임 오브젝트에 중복 로컬 데이터를 저장하는 대신 ScriptableObject 하나에 저장할 수 있습니다. 그런 다음 각 오브젝트가 데이터 자체를 복사하는 대신 공유 데이터 에셋에 대한 레퍼런스를 저장하면 됩니다. 이렇게 하면 수천 개의 오브젝트를 사용하는 프로젝트에서 성능을 크게 향상하는 이점을 누릴 수 있습니다.

스크립터블 오브젝트에서 필드를 생성하여 값 또는 설정을 저장한 다음 MonoBehaviour에서 스크립터블 오브젝트를 참조하세요.



다양한 게임 오브젝트의 설정을 보관하는 Inventory 스크립터블 오브젝트

스크립터블 오브젝트의 필드를 사용하면 MonoBehaviour로 오브젝트를 인스턴스화할 때마다 데이터의 불필요한 중복을 방지할 수 있습니다.

소프트웨어 디자인에서는 이를 플라이웨이트(flyweight) 패턴 최적화라고 합니다. 이렇게 스크립터블 오브젝트를 사용하여 코드를 재구성하면 많은 양의 값이 복사되는 것을 방지하고 메모리 사용량을 줄일 수 있습니다. 디자인 패턴 및 SOLID 원칙으로 코딩 스킬 업그레이드 전자책에서 플라이웨이트 패턴을 비롯한 다양한 패턴과 디자인 원칙에 대해 자세히 알아보세요.

스크립터블 오브젝트 소개(영어) 튜토리얼 동영상을 시청하고 스크립터블 오브젝트의 장점을 알아보세요. 여기의 Unity 기술 자료 및 스크립터블 오브젝트를 사용하여 Unity에서 모듈식 게임 아키텍처 만들기 기술 가이드도 도움이 됩니다.

람다식 사용 지양

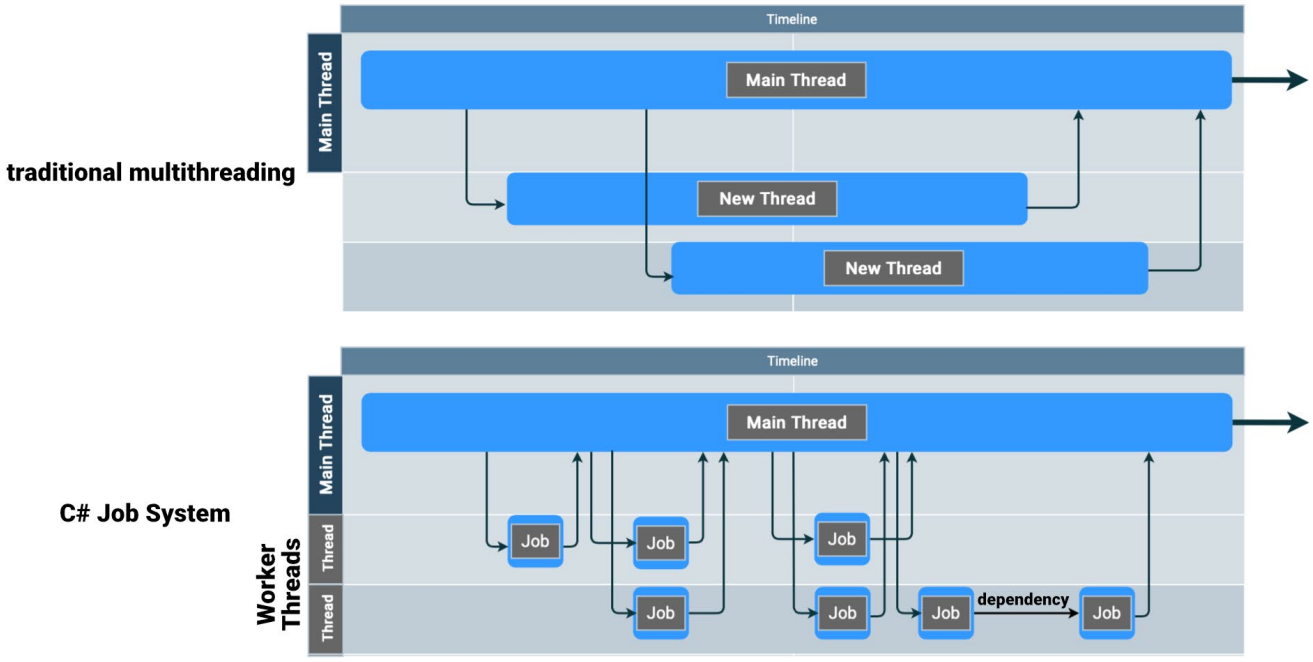
람다식을 사용하면 코드를 간소화할 수 있지만, 간소화에 따른 문제가 발생합니다. 람다식을 호출하면 델리게이트도 함께 생성되는데, 컨텍스트(예: this, 인스턴스 멤버 또는 로컬 변수)를 람다에 전달하면 델리게이트에 대한 모든 캐싱이 무효화됩니다. 그러므로 람다식을 자주 호출하면 상당한 양의 메모리 트래픽이 발생할 수 있습니다.

람다식을 사용하는 동안 클로저를 포함하는 모든 메서드를 리팩터링하세요. 리팩터링 방법에 대한 예는 여기를 참조하세요.

C# 잡 시스템

최신 CPU에는 코어가 여러 개 있지만, 애플리케이션에서 다중 코어를 활용하려면 멀티스레드 코드가 필요합니다. Unity 잡 시스템을 사용하면 대형 작업을 추가 CPU 코어에서 병렬로 실행되는 작은 청크로 분할하여 성능을 크게 개선할 수 있습니다.

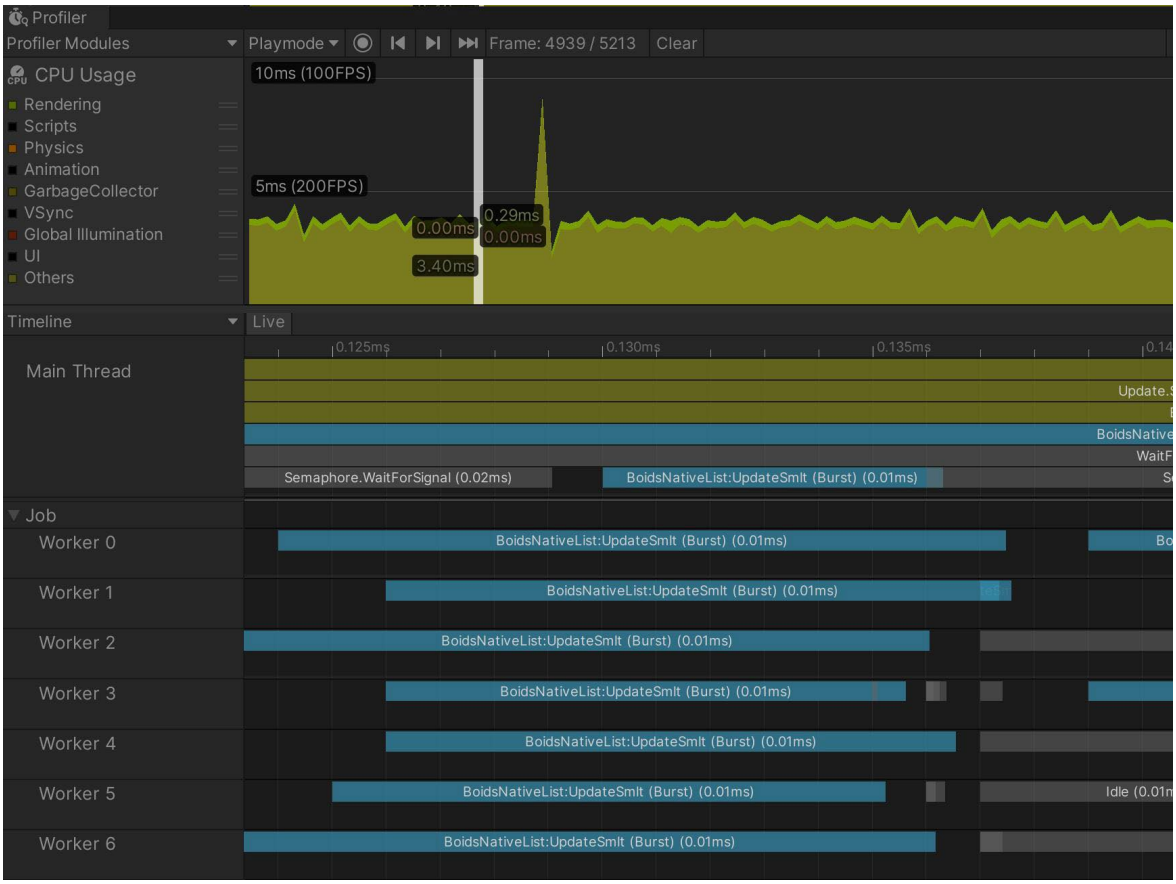
멀티스레드 프로그래밍에서는 하나의 CPU 실행 스레드인 메인 스레드가 작업을 처리하기 위해 다른 스레드를 만들기도 합니다. 추가된 워커 스레드는 작업이 완료되면 메인 스레드와 동기화됩니다.



기존 멀티스레딩 프로그래밍에서는 스레드가 생성과 제거를 거칩니다. C# 잡 시스템에서는 작은 작업이 스레드 풀에서 실행됩니다.

장시간 실행되는 작업이 있는 경우 이러한 방식의 멀티스레딩이 효과적입니다. 그러나 일반적으로 초당 30~60프레임으로 짧은 작업을 많이 처리해야 하는 게임 애플리케이션에서는 이 방법을 사용하면 효율이 떨어집니다.

그래서 Unity는 C# 잡 시스템이라는 조금 다른 멀티스레딩 방식을 사용합니다. 수명이 짧은 스레드를 많이 생성하는 대신 '**잡**'이라는 더 작은 단위로 작업을 분할합니다.



워커 스레드에서 실행 중인 잡을 보여 주는 Profiler의 Timeline 뷰

잡은 대기열에 추가되어 워커 스레드의 공유 풀에서 실행되도록 예약됩니다. JobHandles를 사용하면 종속 관계를 생성하여 잡이 올바른 순서로 실행되게 할 수 있습니다.

멀티스레딩의 잠재적인 문제 중 하나는 경쟁 상태로, 두 스레드가 동시에 공유 변수에 액세스하는 경우 발생합니다. 경쟁 상태를 방지하기 위해 Unity 멀티스레딩은 안전 시스템을 사용하여 잡이 실행해야 하는 데이터를 분리합니다. C# 잡 시스템은 잡 구조의 복사본으로 각 잡을 실행하여 경쟁 상태를 없앱니다.

Unity C# 잡 시스템을 사용하려면 다음 가이드라인을 따르세요.

— 클래스를 구조체로 변경합니다. 잡은 IJob 인터페이스를 구현하는 모든 구조체입니다. 많은 오브젝트에 대해 같은 작업을 수행하는 경우 IJobParallelFor를 사용하여 여러 코어에서 작업을 실행할 수도 있습니다.

— 잡에 전달되는 데이터는 blittable 유형이어야 합니다. 레퍼런스 유형을 제거하고 blittable 데이터만 복사본으로 잡에 전달하세요.

— 각 잡 내부의 작업은 안전을 위해 분리되기 때문에 NativeContainer를 사용해 결과를 메인 스레드로 다시 보내야 합니다. Unity Collections 패키지의 NativeContainer는 네이티브 메모리용 C# 래퍼를 제공합니다. NativeContainer의 하위 유형(예: NativeArray, NativeList, NativeHashMap, NativeQueue 등)은 동등한 C# 데이터 구조처럼 작동합니다.

C# 잡 시스템을 사용하여 프로젝트의 CPU 성능을 최적화하는 방법을 확인하려면 기술 자료를 참조하시기 바랍니다.

버스트 컴파일러

버스트 컴파일러는 잡 시스템을 보완합니다. 버스트는 LLVM을 사용하여 IL/.NET 바이트코드를 최적화된 네이티브 코드로 변환합니다. 패키지 관리자에서 **com.unity.burst** 패키지를 추가하기만 하면 버스트 컴파일러를 사용할 수 있습니다.

Unity 개발자가 버스트를 사용하면 C#의 하위 세트를 계속 사용해 편의성을 높이는 동시에 성능을 개선할 수 있습니다.

스크립트에서 버스트 컴파일러를 활성화하려면 다음을 따르세요.

— 정적 변수를 제거합니다. 목록에 작성해야 하는 경우 NativeDisableContainerSafetyRestriction 속성이 있는 NativeArray를 사용하는 것이 좋습니다. 이렇게 하면 병렬 잡이 NativeArray에 작성할 수 있습니다.

— Mathf. 함수 대신 Unity.Mathematics 함수를 사용합니다.

— 잡 정의에 BurstCompile 속성을 추가합니다.

```
  [BurstCompile]
public struct MyFirstJob : IJob
{
       public NativeArray<float3> ToNormalize;

       public void Execute()
         {
              for (int i = 0; i < ToNormalize.Length; i++)
              {
                     ToNormalize[i] = math.normalize(ToNormalize[i]);
              }
       }
}
```

위는 float3 배열에서 실행되고 벡터를 정규화하는 버스트 잡의 예시입니다. 이 예시에서는 앞서 언급한 대로 Unity Mathematics 패키지를 사용합니다.

C# 잡 시스템과 버스트 컴파일러는 모두 Unity DOTS(데이터 지향 기술 스택)의 일부입니다. 그러나 잡 시스템과 버스트 컴파일러를 기본 Unity 게임 오브젝트나 엔티티 컴포넌트 시스템과 사용할 수 있습니다.

버스트가 C# 잡 시스템과 함께 어떻게 워크플로를 가속화하는지 알아보려면 최신 기술 자료를 살펴보고 고급 Unity 개발자를 위한 데이터 지향 기술 스택 소개 전자책을 다운로드해 보세요.

프로젝트 구성

성능에 영향을 줄 수 있는 프로젝트 설정이 몇 가지 있습니다.

불필요한 플레이어 설정 또는 품질 설정 비활성화

Player Settings에서 Auto Graphics API를 비활성화하고 각 타겟 플랫폼에서 지원할 계획이 없는 그래픽스 API를 제거합니다. 그러면 셰이더 배리언트가 지나치게 생성되는 것을 막을 수 있습니다. 애플리케이션이 오래된 CPU를 지원하지 않는다면 해당 CPU에 대해 Target Architectures를 비활성화합니다.

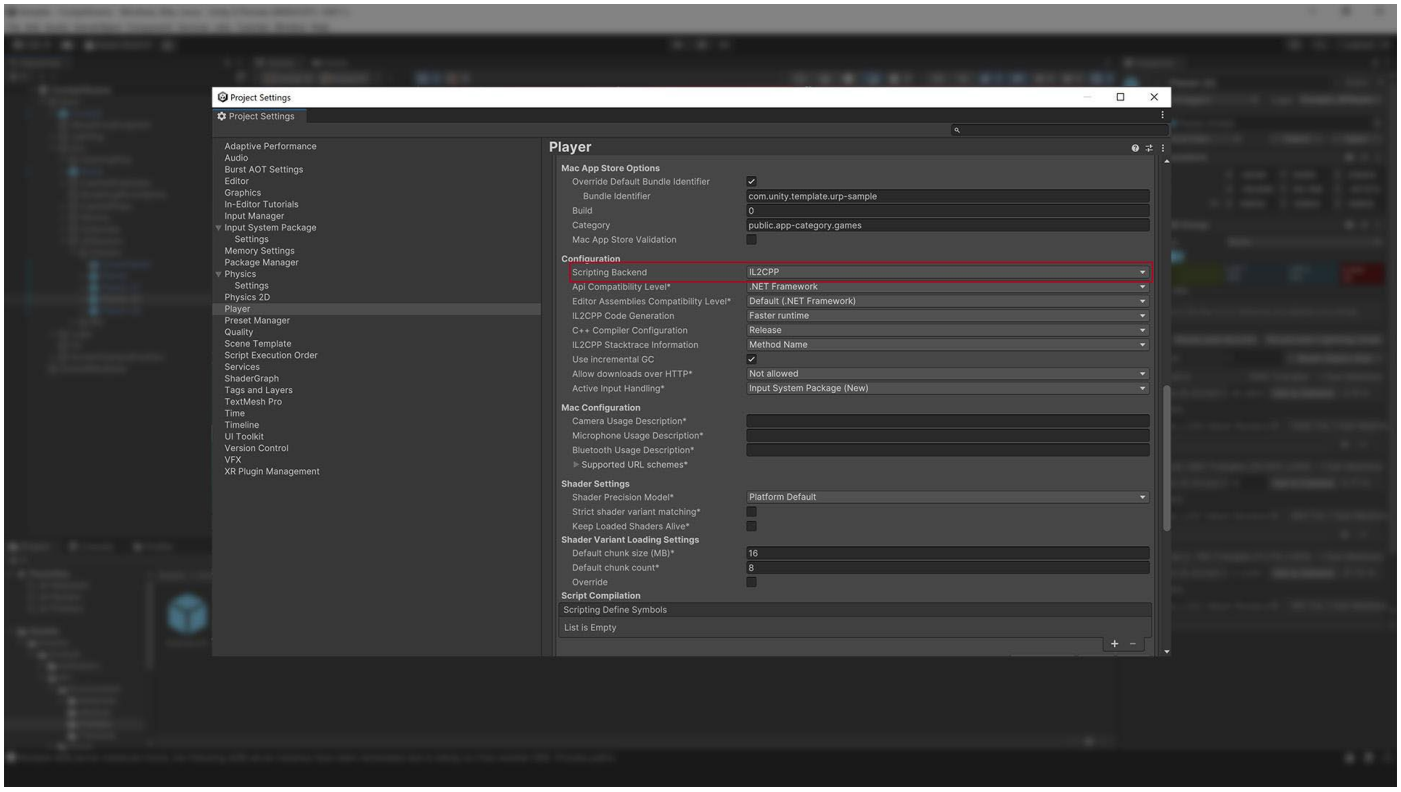
Quality Settings에서 불필요한 품질 수준을 비활성화하세요.

IL2CPP로 전환

스크립팅 백엔드를 Mono에서 IL2CPP(C++로 변환하는 중간 언어)로 전환하는 것이 좋습니다. IL2CPP로 전환하면 전반적으로 런타임 성능이 개선됩니다.

다만, 빌드 시간이 늘어날 수 있습니다. 일부 개발자는 빠른 반복 작업을 위해 로컬에서는 Mono를 사용하고, 빌드 머신이나 릴리스 예정 빌드에서는 IL2CPP로 전환하기도 합니다. 빌드 시간을 줄이려면 IL2CPP 빌드 시간 최적화 기술 자료를 참조하시기 바랍니다.

IL2CPP만 사용할 수 있는 PlayStation 플랫폼의 경우, **Player Settings > Other Settings > IL2CPP optimization level** 설정으로 이동합니다. 빌드 시간을 줄일 수 있도록 개발 중에는 비교적 최적화를 수행하지 않는 옵션을 사용하세요. 프로파일링이나 최종 릴리스에 대해서는 **Optimized Compile, Remove Unused Code, Optimized link**를 선택하세요.



스크립팅 백엔드를 IL2CPP로 전환

IL2CPP를 사용하는 경우 Unity는 타겟 플랫폼에 대한 네이티브 바이너리 파일(예: .exe, .apk, .xap)을 생성하기 전에 스크립트와 어셈블리의 IL 코드를 C++ 코드로 전환합니다.

빌드 시간을 최적화하는 방법에 대한 자세한 내용은 기술 자료를 참조하시기 바랍니다.

대규모 계층 구조 사용 지양

계층 구조를 분리하세요. 게임 오브젝트가 계층 구조 내에 중첩될 필요가 없다면 부모 자식 관계를 간소화해야 합니다. 계층 구조가 단순하면 씬에서 트랜스폼을 새로고침할 때 멀티스레딩의 이점을 누릴 수 있습니다. 계층 구조가 복잡하면 불필요한 트랜스폼 계산과 높은 가비지 컬렉션 비용이 발생합니다.

트랜스폼에 관한 베스트 프랙티스는 이 유나이트 세션(영어)을 참고하세요.

그래픽스

조명과 효과는 상당히 복잡하므로 최적화를 진행하기 전에 렌더 파이프라인 기술 자료를 먼저 살펴보는 것이 좋습니다.

렌더 파이프라인 활용

씬 조명을 최적화하는 데 특별한 방법이 있는 것은 아니고 반복적인 프로세스를 거쳐야 합니다. 이 작업은 시행착오가 발생하며 일반적으로 예술적 방향성과 렌더 파이프라인에 따라 프로세스가 변합니다.

씬 조명 작업을 시작하기 전에, 사용할 수 있는 렌더 파이프라인 중 하나를 선택해야 합니다. 렌더 파이프라인은 씬의 콘텐츠를 가져와 화면에 표시하는 일련의 작업을 수행합니다.

Unity는 서로 다른 기능과 성능 특성으로 사전 제작된 세 가지 렌더 파이프라인을 제공하지만, 직접 제작한 렌더 파이프라인을 사용해도 됩니다.

1. URP(유니버설 렌더 파이프라인)는 사전 제작된 SRP(스크립터블 렌더 파이프라인)입니다. URP는 아티스트 친화적인 워크플로를 제공하여, 모바일부터 고사양 콘솔과 PC까지 다양한 플랫폼에 최적화된 그래픽스를 제작하는 데 유용합니다. 빌트인 렌더 파이프라인의 뒤를 잇는 URP는 이전 파이프라인이 제공하지 않는 그래픽스 및 렌더링 기능을 제공합니다. URP는 성능을 유지하기 위해 조명과 셰이딩 계산 비용을 줄이는 방향으로 절충점을 찾습니다. 모바일과 VR을 비롯한 대부분의 타겟 플랫폼용 애플리케이션을 제작하려면 URP를 선택하는 것이 좋습니다.

   Unity 고급 사용자를 위한 유니버설 렌더 파이프라인 전자책에서 URP 기능의 전체 개요를 살펴보세요.

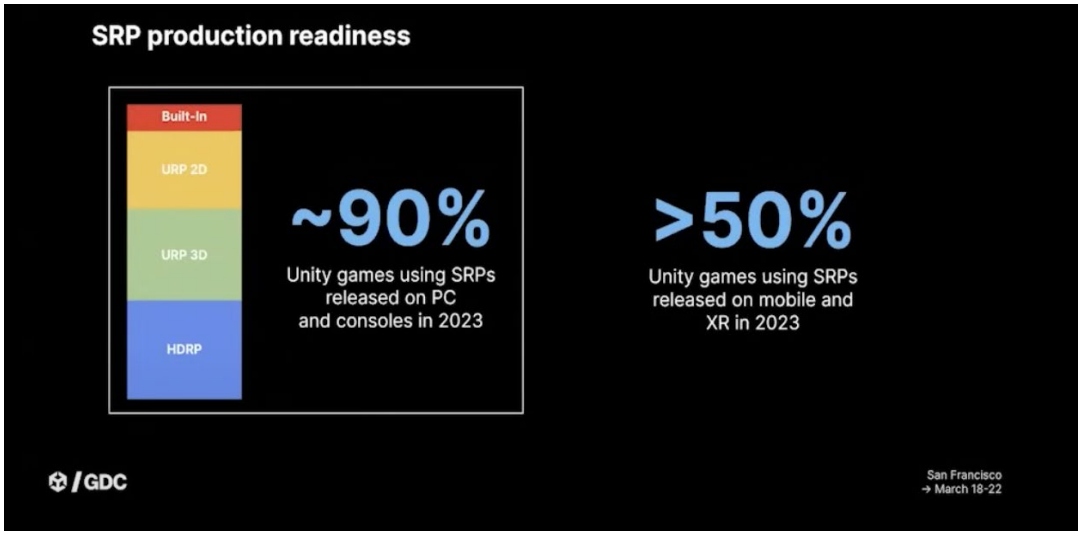
2. HDRP(고해상도 렌더 파이프라인) 역시 미리 마련된 SRP로, 정확도 높은 최첨단 그래픽스 제작을 위해 설계되었습니다. HDRP는 PC, Xbox, PlayStation 등의 고사양 하드웨어를 위한 렌더 파이프라인[2]으로, 고급 조명, 반사, 그림자를 사용하여 게임에서 최상급 품질의 사실적 그래픽스를 구현하려 할 때 권장됩니다. HDRP는 물리 기반 조명과 머티리얼을 사용하며 개선된 디버깅 툴을 지원합니다.

   고해상도 렌더 파이프라인의 조명 및 환경 전자책에서 HDRP 기능의 전체 개요를 살펴보세요.

3. 빌트인 렌더 파이프라인은 일반적으로 사용할 수 있는 Unity의 기존 렌더 파이프라인으로 커스터마이징이 제한됩니다.

PC/콘솔용 Unity 게임의 약 90%가 사용하는 SRP

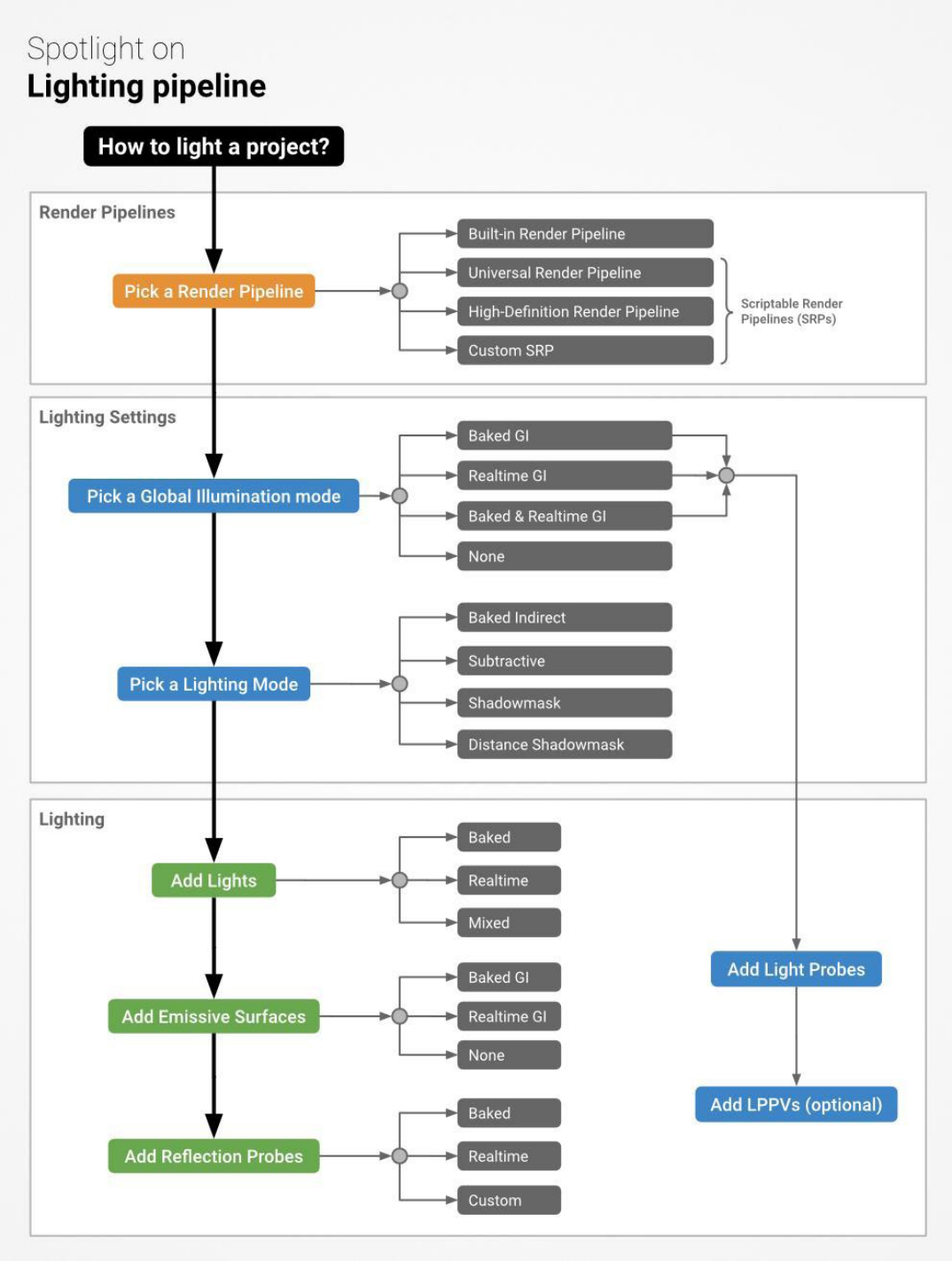
유니티의 최신 데이터에 따르면 2023년에 PC 및 콘솔에 출시된 Unity 게임에서 가장 많이 선택한 렌더 파이프라인은 URP입니다. 아래 그래프를 보면 빌트인 렌더 파이프라인은 이제 아주 소수의 개발 팀에서만 사용한다는 것을 볼 수 있습니다.



Unity에서 제공하는 다양한 파이프라인을 사용하여 출시된 게임의 비율

[2] HDRP는 현재 모바일 플랫폼에서 지원되지 않습니다. 자세한 내용은 요구 사항 및 호환성 페이지를 참조하시기 바랍니다.

현재 대부분의 Unity 프로젝트는 URP 또는 HDRP로 제작되고 있지만, 빌트인 렌더 파이프라인(목록의 3번째 옵션) 역시 Unity 6에서 계속 사용할 수 있습니다.



프로젝트를 계획하는 초기에 렌더 파이프라인을 선택해야 합니다.

HDRP의 고사양 그래픽 기능을 선보이는 유니티 데모 에너미즈(Enemies)

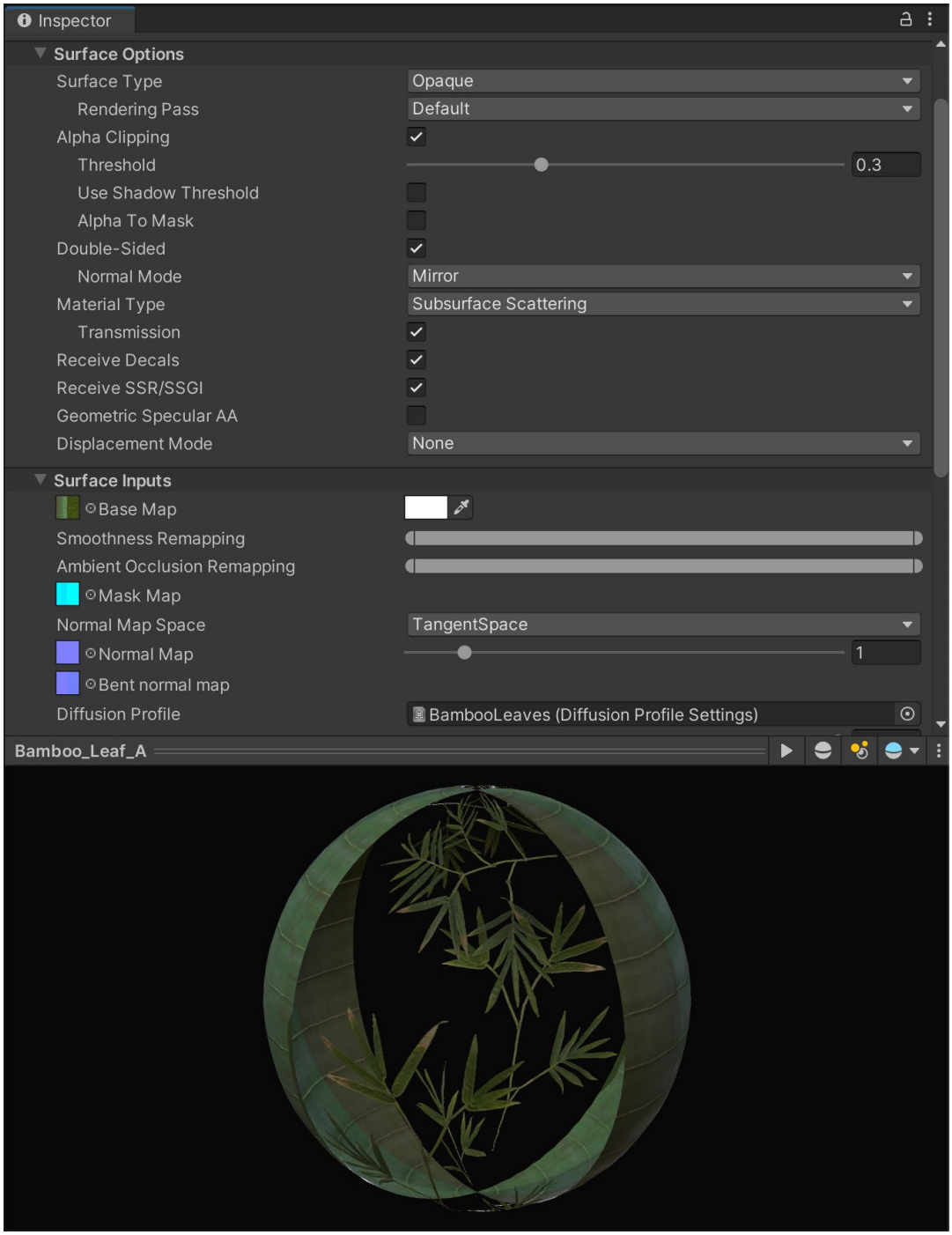
URP와 HDRP는 SRP(스크립터블 렌더 파이프라인)를 기반으로 작동합니다. SRP는 C# 스크립트를 사용하여 렌더링 커맨드를 예약하고 설정할 수 있는 가벼운 API 레이어입니다. 이러한 유연성 덕분에 파이프라인의 거의 모든 부분을 커스터마이즈할 수 있습니다. 또한 SRP를 기반으로 커스텀 렌더 파이프라인을 제작할 수도 있습니다.

사용 가능한 여러 파이프라인의 더 자세한 비교 내용은 Unity의 렌더 파이프라인 기술 자료를 참조하시기 바랍니다.

콘솔용 렌더 파이프라인 패키지

PlayStation 4, **PlayStation 5**, **Game Core Xbox**용 프로젝트를 제작하려면, 지원하려는 각 플랫폼별로 추가 패키지를 설치해야 합니다. 각 플랫폼에 필요한 패키지는 다음과 같습니다.

- **PlayStation 4:** com.unity.render-pipelines.ps4
- **PlayStation 5**: com.unity.render-pipelines.ps5
- **Xbox 콘솔**: com.unity.render-pipelines.gamecore



피부나 나뭇잎 같은 머티리얼에는 HDRP로 사전 설정된 고급 조명과 셰이딩 기능을 활용할 수 있습니다.

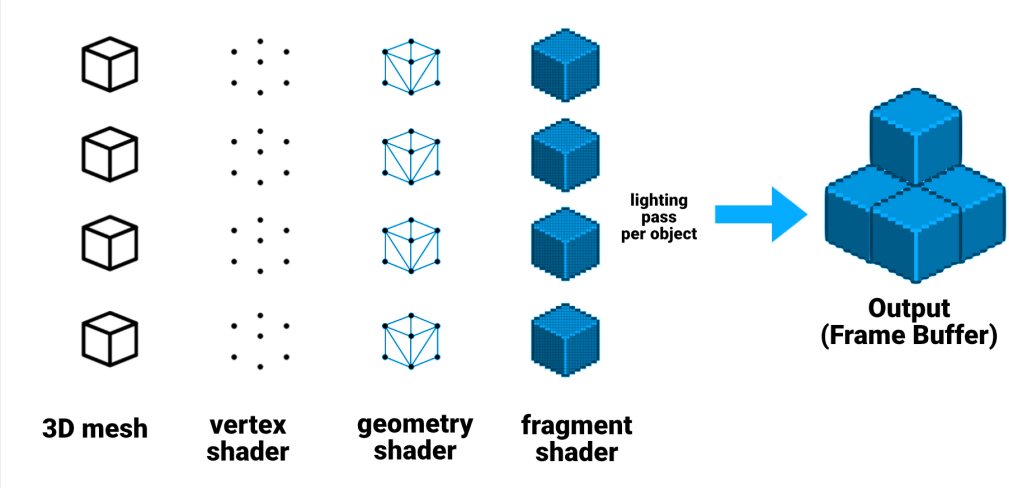
렌더링 경로 선택

렌더 파이프라인을 선택할 때는 **렌더링 경로**도 고려해야 합니다. 렌더링 경로는 조명, 셰이딩과 관련된 일련의 특정 작업을 의미합니다. 어떤 렌더링 경로를 사용할지는 애플리케이션 요구 사항과 타겟 하드웨어에 따라 다릅니다.

포워드

포워드 렌더링에서 그래픽 카드는 지오메트리를 투영하고 버텍스로 분할합니다. 버텍스는 더 나아가 프래그먼트 또는 픽셀로 세분화를 거쳐 화면에 렌더링되며 최종 이미지를 생성합니다.

파이프라인은 각 오브젝트를 한 번에 하나씩 그래픽스 API로 전달합니다. 포워드 렌더링은 각 광원마다 리소스를 소모합니다. 씬에 광원이 많을수록 렌더링 시간도 길어집니다.



포워드 렌더링 경로

빌트인 파이프라인의 포워드 렌더러는 오브젝트별로 별도의 라이트 패스를 통해 광원을 드로우합니다. 같은 게임 오브젝트에 여러 광원이 겹치는 경우, 겹치는 영역에 같은 픽셀을 두 번 이상 드로우해야 해서 상당한 오버드로우가 발생할 수 있습니다. 오버드로우를 줄이려면 실시간 광원을 최소화해야 합니다.

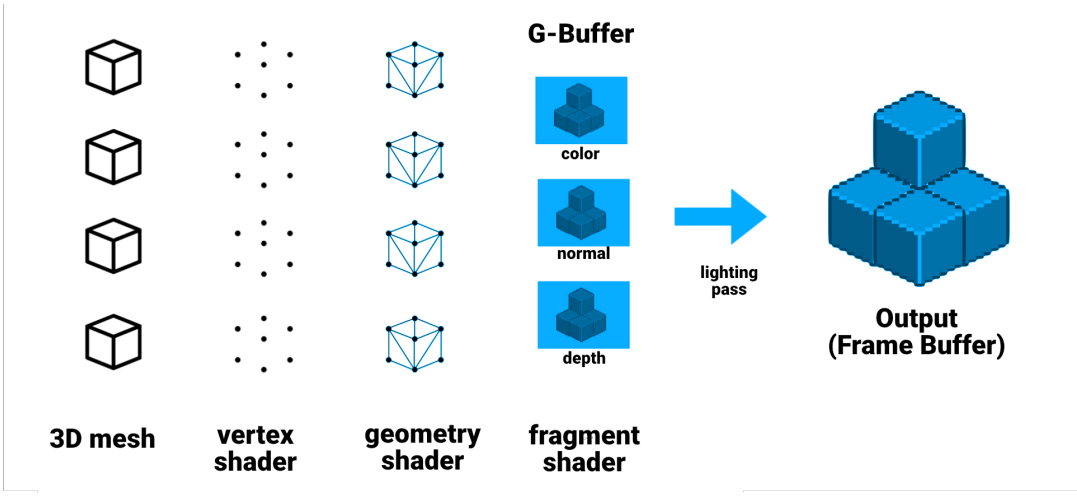
URP는 광원별로 하나의 패스를 렌더링하는 대신 오브젝트별로 광원을 컬링합니다. 그러면 하나의 패스에서 조명을 계산할 수 있기 때문에 빌트인 렌더 파이프라인의 포워드 렌더러에 비해 드로우 콜이 줄어듭니다.

포워드+

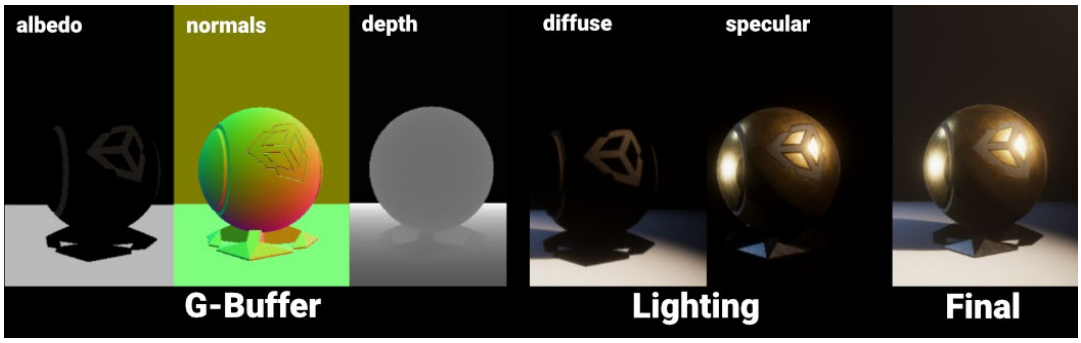
포워드+ 렌더링은 오브젝트 단위 대신 공간을 분류해 광원을 컬링하여 표준 포워드 렌더링보다 더 나은 품질을 보입니다. 이 방식을 사용하면 프레임을 렌더링할 때 전반적으로 더 많은 광원을 활용할 수 있습니다. 디퍼드 렌더링에서는 Native RenderPass API를 지원하여 G버퍼와 조명 패스가 싱글 렌더 패스로 결합될 수 있도록 합니다.

디퍼드 셰이딩

디퍼드 셰이딩에서 조명은 오브젝트별로 계산되지 않습니다.



디퍼드 셰이딩 경로



디퍼드 셰이딩은 각 오브젝트 대신 버퍼에 조명을 적용합니다.

디퍼드 셰이딩은 조명 계산 등의 작업을 후반 단계로 연기합니다. 디퍼드 셰이딩은 두 개의 패스를 사용합니다.

Unity는 첫 번째 패스, 즉 G버퍼 지오메트리 패스에서 게임 오브젝트를 렌더링합니다. 이 패스는 여러 유형의 지오메트리 프로퍼티를 검색해서 가져오고 텍스처 세트에 저장합니다. G버퍼 텍스처에는 다음 항목이 포함될 수 있습니다.

— 디퓨즈 컬러 및 스페큘러 컬러

— 표면 평활도

— 오클루전

— 월드 공간 노멀

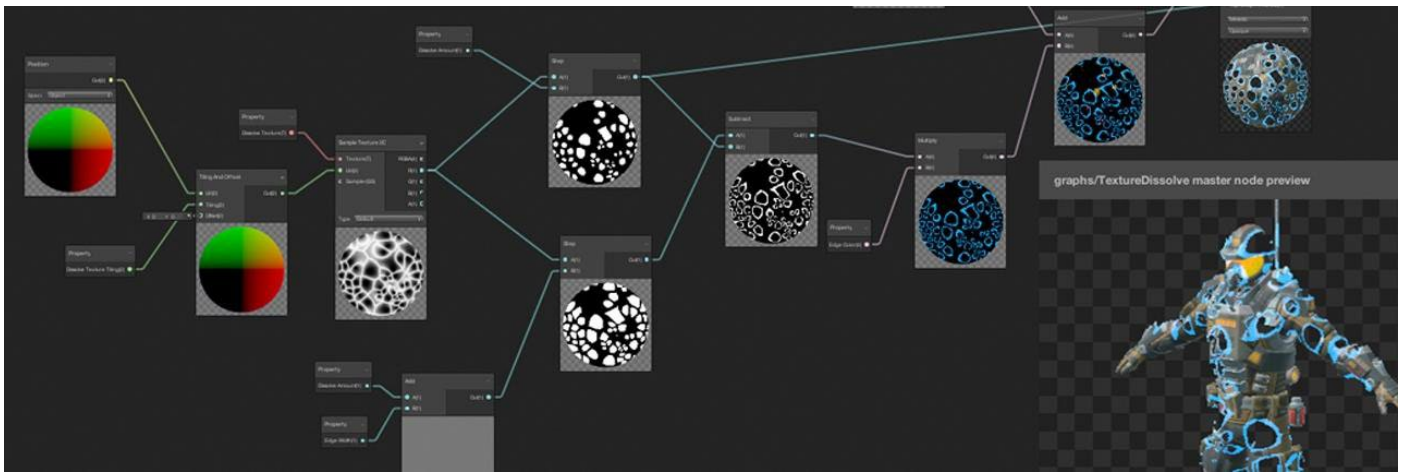
— 이미션 + 앰비언트 + 반사 + 라이트맵

두 번째 패스, 즉 조명 패스에서 Unity는 G버퍼를 기반으로 씬의 조명을 렌더링합니다. 개별 오브젝트 대신 버퍼를 기반으로 각 픽셀을 반복 작업하고 조명 정보를 계산한다고 생각하면 됩니다. 따라서 디퍼드 셰이딩에서 그림자를 드리우지 않는 광원을 추가해도 포워드 렌더링처럼 성능 저하가 발생하지 않습니다.

렌더링 경로를 선택하는 것 자체는 최적화가 아니지만, 렌더링 경로는 프로젝트를 최적화하는 방식에 영향을 줄 수 있습니다. 이 섹션에서 다루는 다른 기술과 워크플로는 사용자가 선택한 렌더 파이프라인과 렌더링 경로에 따라 달라질 수 있습니다.

Shader Graph 최적화

HDRP와 URP 모두 셰이더 제작을 위한 비주얼 인터페이스인 Shader Graph를 지원합니다. 셰이더 그래프를 사용하면 훨씬 더 복잡한 셰이딩 효과를 제작할 수 있습니다. 비주얼 그래프 시스템에 제공되는 150개 이상의 노드를 사용하여 더 많은 셰이더를 제작해 보세요. 또한 API를 사용하여 커스텀 노드를 제작할 수도 있습니다.



시각적 인터페이스를 사용한 셰이더 제작

각 Shader Graph는 그래프의 출력을 결정하는 호환 가능한 마스터 노드로 시작합니다. 비주얼 인터페이스를 통해 노드와 연산자를 추가하고 셰이더 로직을 구성하세요.

그러면 Shader Graph가 렌더 파이프라인의 백 엔드로 전달됩니다. 최종적으로 HLSL이나 Cg로 작성한 셰이더와 기능 면에서 유사한 ShaderLab 셰이더를 제작할 수 있습니다.

Shader Graph 최적화는 기존 HLSL/Cg 셰이더에 적용되는 것과 대체로 비슷한 규칙을 따릅니다. Shader Graph는 처리하는 작업이 많을수록 애플리케이션 성능에 더 많은 영향을 줍니다.

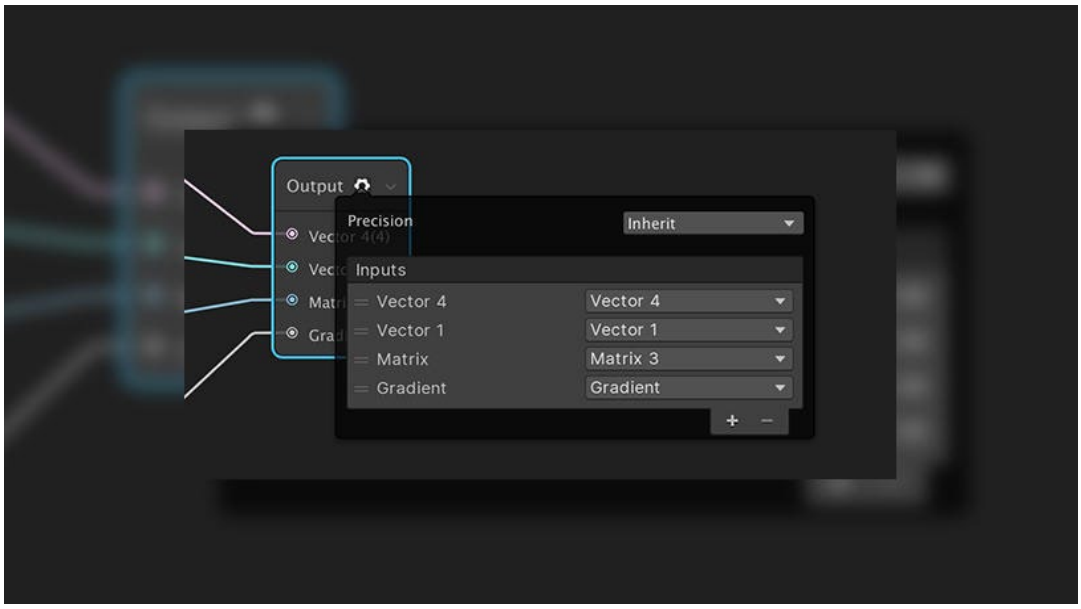
애플리케이션이 CPU 바운드라면 셰이더를 최적화해도 프레임 속도를 개선할 수 없지만, 모바일 플랫폼에서는 배터리 수명이 향상될 수 있습니다.

애플리케이션이 GPU 바운드라면 아래 가이드라인을 따라 Shader Graph 성능을 개선할 수 있습니다.

— **노드 삭제:** 사용하지 않는 노드를 제거합니다. 꼭 필요한 경우가 아니면 기본 노드나 연결 노드를 변경하지 마세요. Shader Graph는 사용하지 않는 기능을 자동으로 컴파일합니다.

가능하면 값을 텍스처로 베이크하세요. 예를 들어 노드를 사용하여 텍스처 밝기를 높이는 대신 텍스처 에셋 자체에 추가 밝기를 적용할 수 있습니다.

— **더 작은 데이터 포맷 사용**: 가능하면 더 작은 데이터 구조로 전환합니다. 프로젝트에 영향이 없다면 Vector3 대신 Vector2를 사용하는 것이 좋습니다. 가능하다면 float 대신 half를 사용하는 등 정밀도를 줄이는 것도 하나의 방법일 수 있습니다.



가능하면 Output 노드에서 Shader Graph의 정밀도를 낮추는 것이 좋습니다.

— **수학 연산 줄이기:** 셰이더 연산은 초당 여러 번 수행되므로 가능하면 수학 연산자를 최적화해야 합니다. 로직 브랜치를 만드는 것보다는 결과를 블렌딩해 보세요. 상수를 사용하고 벡터를 적용하기 전에 스칼라 값을 결합합니다. 마지막으로 인스펙터에서 인라인 노드로 표시될 필요가 없는 모든 프로퍼티를 전환합니다. 이러한 가속화 과정은 프레임 예산에 도움이 될 수 있습니다.

— **프리뷰 브랜치:** 그래프가 커질수록 컴파일 속도가 느려질 수 있습니다. 현재 미리 보고 싶은 연산만 포함하는 별도의 작은 브랜치를 만들면 워크플로를 간소화할 수 있으며, 원하는 결과를 얻을 때까지 작은 브랜치에서 더 빠르게 반복 작업할 수 있습니다.

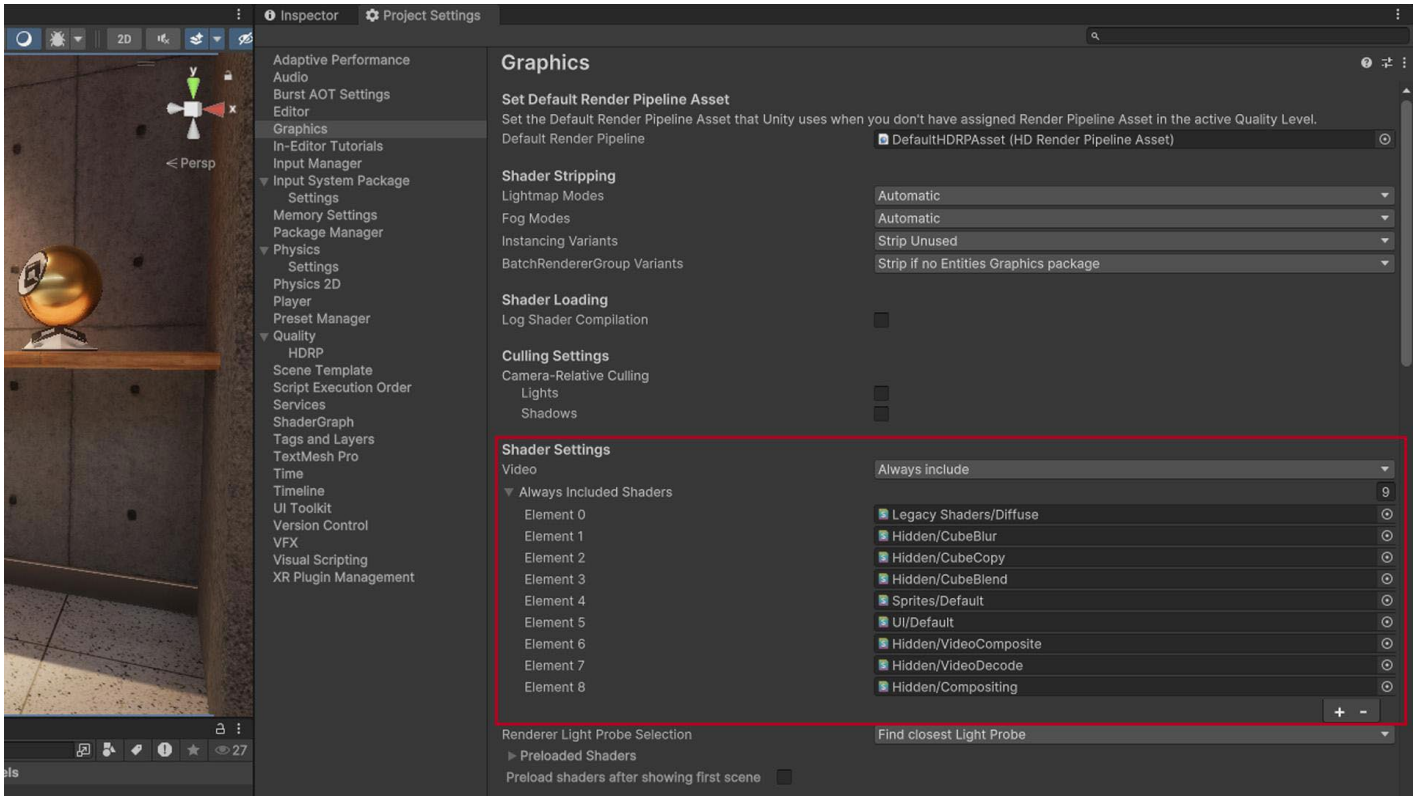
브랜치를 마스터 노드에 연결하지 않은 경우, 그래프에 프리뷰 브랜치를 안전하게 남겨둘 수 있습니다. 컴파일 중 최종 출력에 영향을 주지 않는 노드는 제거됩니다.

— **수동 최적화:** 숙련된 그래픽스 프로그래머라도 Shader Graph를 사용해 스크립트 기반 셰이더용 몇몇 상용구 코드를 작성할 수도 있습니다. Shader Graph 에셋을 선택한 다음 컨텍스트 메뉴에서 Copy Shader를 선택합니다.

새로운 HLSL/Cg 셰이더를 만든 다음 복사한 셰이더 그래프를 붙여 넣습니다. 이는 단방향 작업이지만, 수동 최적화를 통해 성능을 조금이라도 더 개선할 수 있습니다.

빌트인 셰이더 설정 제거

Graphics Settings(**Edit > ProjectSettings > Graphics**)의 **Always Included Shaders** 목록에서 사용하지 않는 모든 셰이더를 제거합니다. 그런 다음 애플리케이션 수명 기간에 필요한 셰이더를 이 목록에 추가합니다.



Always Included Shaders 예시

셰이더 배리언트 스트리핑

셰이더 배리언트는 플랫폼별 기능에 유용할 수 있지만, 빌드 시간이 늘어나고 파일 크기가 커지는 원인이 됩니다.

셰이더 컴파일 pragma 지시문을 사용하여 타겟 플랫폼마다 셰이더를 다르게 컴파일할 수 있습니다. 그런 다음 셰이더 키워드 또는 Shader Graph Keyword 노드를 사용하여 특정 기능을 활성화하거나 비활성화한 셰이더 배리언트를 제작할 수 있습니다.

셰이더 배리언트가 필요하지 않다는 사실을 알면 빌드에 셰이더 배리언트가 포함되지 않도록 막을 수 있습니다.

셰이더 타이밍과 크기에 대해 Editor.log를 파싱합니다. ‘Compiled shader’와 ‘Compressed shader’로 시작하는 라인을 찾습니다.

예제 로그에서 TEST 셰이더는 다음과 같이 표시됩니다.

```
Compiled shader ‘TEST Standard (Specular setup)’ in 31.23s
  d3d9 (total internal programs: 482, unique: 474)
  d3d11 (total internal programs: 482, unique: 466)
  metal (total internal programs: 482, unique: 480)
  glcore (total internal programs: 482, unique: 454)
Compressed shader ‘TEST Standard (Specular setup)’ on d3d9 from 1.04MB to 0.14MB
Compressed shader ‘TEST Standard (Specular setup)’ on d3d11 from 1.39MB to 0.12MB
Compressed shader ‘TEST Standard (Specular setup)’ on metal from 2.56MB to 0.20MB
Compressed shader ‘TEST Standard (Specular setup)’ on glcore from 2.04MB to 0.15MB
```

이를 통해 이 셰이더에 관한 몇 가지 내용을 알 수 있습니다.

- 셰이더는 #pragma multi_compile과 shader_feature로 인해 482개의 배리언트로 확장됩니다.
- Unity는 게임 데이터에 포함된 셰이더를 압축된 크기의 합과 거의 같은 크기로 압축합니다 (0.14+0.12+0.20+0.15 = 0.61MB).
- 런타임 시 Unity는 압축된 데이터(0.61MB)를 메모리에 유지하지만, 현재 사용하는 그래픽스 API의 데이터는 압축되지 않습니다. 예를 들어 현재 Metal API를 사용하고 있다면 그 크기는 2.56MB일 것입니다.

빌드 후, Project Auditor는 Editor.log를 파싱하여 프로젝트로 컴파일된 모든 셰이더, 셰이더 키워드, 셰이더 배리언트 목록을 표시할 수 있습니다. 또한 게임 실행 후 Player.log를 분석할 수도 있습니다. 애플리케이션이 런타임에 실제로 컴파일하고 사용한 배리언트가 표시됩니다.

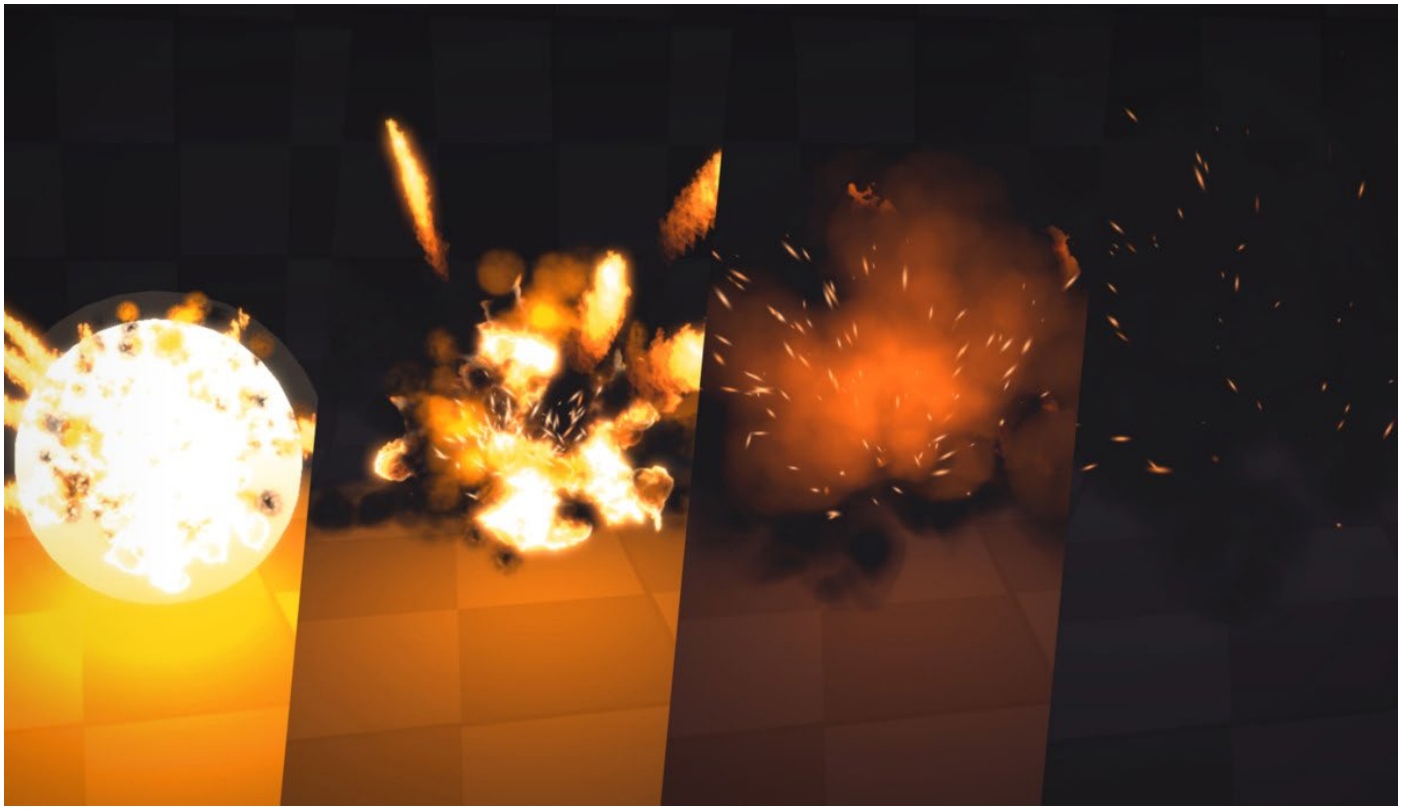
이 정보를 활용하여 스크립터블 셰이더 스트리핑 시스템을 구축하고 배리언트 수를 줄여 보세요. 그러면 빌드 시간, 빌드 크기, 런타임 메모리 사용량을 개선할 수 있습니다.

스크립터블 셰이더 배리언트 스트리핑(영문) 블로그 포스팅에서 이 과정에 대한 자세한 내용을 확인해 보세요.

파티클 시뮬레이션: 파티클 시스템 또는 VFX Graph

Unity 6에는 연기, 액체, 불꽃 등의 효과를 위한 두 가지 파티클 시뮬레이션 솔루션이 있습니다.

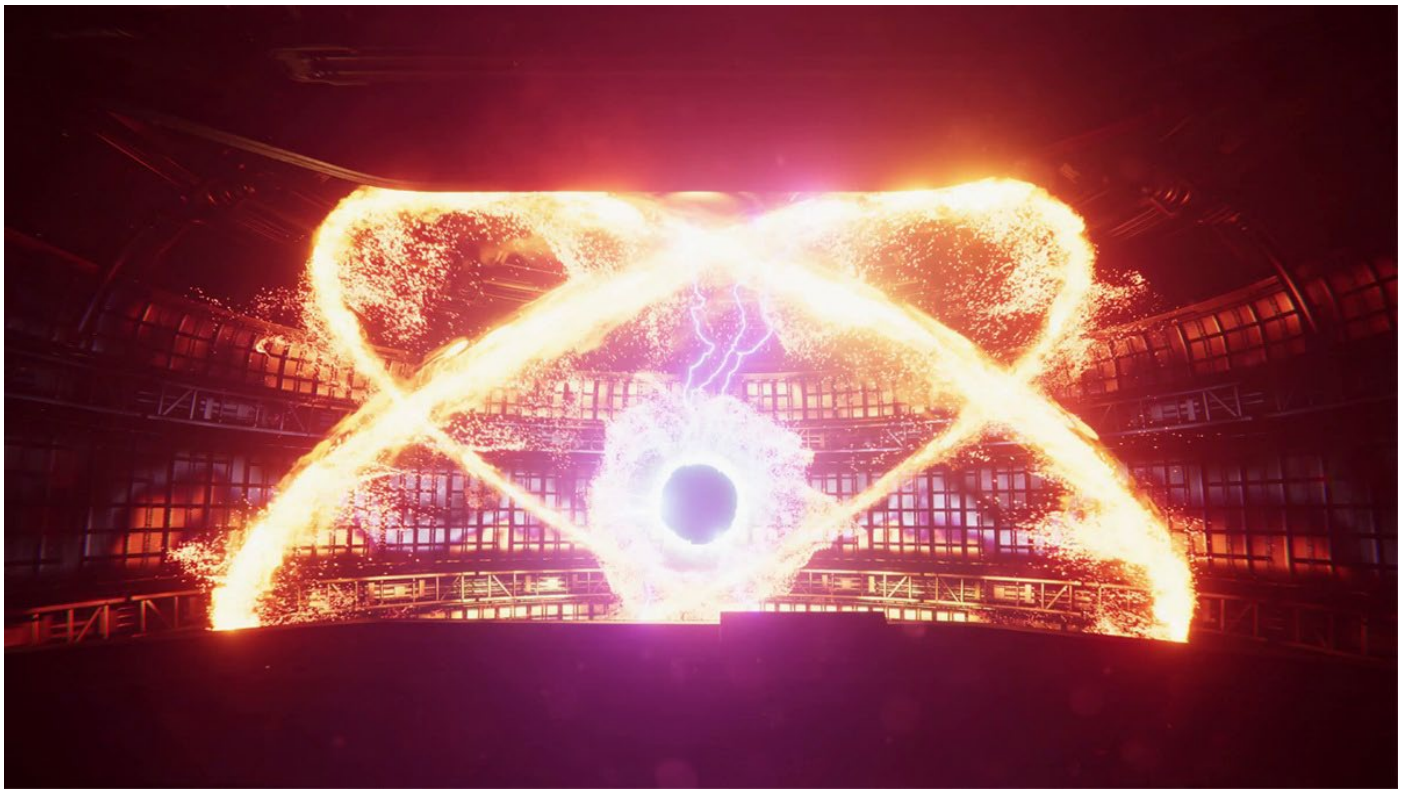
— 빌트인 파티클 시스템을 사용하면 CPU에서 수천 개의 파티클을 시뮬레이션할 수 있습니다. C# 스크립트를 사용해 시스템과 개별 파티클을 정의할 수 있습니다. 파티클 시스템은 Unity의 기본 물리 시스템뿐만 아니라 씬의 모든 콜라이더와 상호 작용할 수 있습니다. 파티클 시스템은 최고의 호환성을 제공하며, Unity가 지원하는 모든 빌드 플랫폼에서 사용 가능합니다.



빌트인 파티클 시스템을 사용한 간단한 효과 시뮬레이션

— VFX Graph는 Unity에서 정교한 시각 효과를 제작하기 위한 향상된 기능을 제공하는 새로운 시스템으로, 특히 고사양 그래픽스와 성능을 타게팅하는 프로젝트에 유용합니다. 컴퓨트 셰이더를 사용하여 GPU에서 계산을 수행하며, 대규모 시각 효과를 위한 수백만 개의 파티클을 시뮬레이션할 수 있습니다. 이 워크플로에는 자유로운 커스터마이징이 가능한 그래프 뷰가 포함되어 있습니다. 파티클은 컬러 및 뎁스 버퍼와 상호 작용할 수도 있습니다.

VFX Graph는 기본 물리 시스템에 액세스할 수 없지만, Point Cache, Vector Field, Signed Distance Field와 같은 복잡한 에셋과 상호 작용할 수 있습니다. VFX Graph는 HDRP 및 URP와 함께 사용할 수 있으며, 컴퓨트 셰이더를 지원하는 플랫폼에서 이용할 수 있습니다.



Visual Effect Graph로 제작한 수백만 개의 파티클 효과 화면

두 시스템 중 하나를 선택할 때는 기기 호환성을 고려하세요. 대부분의 PC와 콘솔은 컴퓨트 셰이더를 지원하지만, 모바일 기기는 컴퓨트 셰이더를 지원하지 않는 경우가 많습니다. 타겟 플랫폼이 컴퓨트 셰이더를 지원한다면 프로젝트에 두 가지 파티클 시뮬레이션을 모두 사용할 수 있습니다.

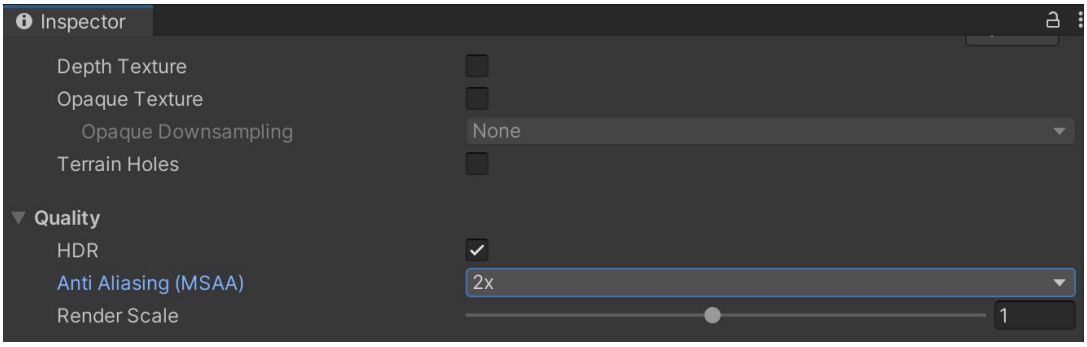
Unity에서 고급 시각 효과를 만드는 최종 가이드 전자책에서 고사양 시각 효과를 제작하는 방법을 자세히 알아보세요.

안티앨리어싱으로 다듬기

안티앨리어싱은 이미지를 다듬고, 가장자리의 계단 현상을 줄이고, 스페큘러 앨리어싱을 최소화합니다.

빌트인 렌더 파이프라인에서 포워드 렌더링을 사용하고 있다면 Quality Settings에서 MSAA(멀티샘플 안티앨리어싱)를 사용할 수 있습니다. MSAA는 고품질 안티앨리어싱을 제공하지만, 리소스를 많이 소모할 수 있습니다. 드롭다운 메뉴의 **MSAA 샘플 수**(None, 2X, 4X, 8X)는 렌더러가 효과를 평가하는 데 사용하는 샘플 수를 정의합니다.

URP나 HDRP에서 포워드 렌더링을 사용하고 있다면 렌더 파이프라인 에셋에서 MSAA를 활성화할 수 있습니다.



URP의 경우, 렌더 파이프라인 에셋에서 MSAA 설정을 찾을 수 있습니다.

또는 안티앨리어싱을 포스트 프로세싱 효과로 추가할 수도 있는데, 이 옵션은 카메라 컴포넌트의 **Anti-aliasing**에서 확인할 수 있습니다.

— **FAAA(Fast approximate anti-aliasing)**는 픽셀별 수준에서 가장자리를 다듬습니다. 리소스를 가장 적게 사용하는 안티앨리어싱으로, 최종 이미지에 약간의 블러 처리를 합니다.

— **SMAA(Subpixel morphological anti-aliasing)**는 이미지 테두리를 기반으로 픽셀을 블렌딩합니다. SMAA는 FXAA보다 훨씬 선명한 결과를 낼 수 있으며, 만화처럼 평평한 아트 스타일이나 산뜻한 아트 스타일에 적합합니다.

HDRP에서는 FXAA와 SMAA를 카메라의 **Post Anti-aliasing** 설정에서 사용할 수도 있으며, URP와 HDRP에서 또 다른 옵션도 제공합니다.

— **TAA(Temporal Anti-Aliasing)**는 히스토리 버퍼의 프레임을 사용하여 가장자리를 다듬습니다. FXAA보다 더 효율적이지만, TAA를 사용하려면 모션 벡터가 필요합니다. TAA를 사용하면 앰비언트 오클루전과 볼류메트릭 효과를 개선할 수도 있습니다. TAA는 일반적으로 FXAA보다 그래픽스 품질이 높지만, 리소스를 더 많이 소모하고 가끔 고스팅 결함을 야기할 수 있습니다.

HDRP 카메라의 포스트 안티앨리어싱 효과로 선택할 수 있는 TAA

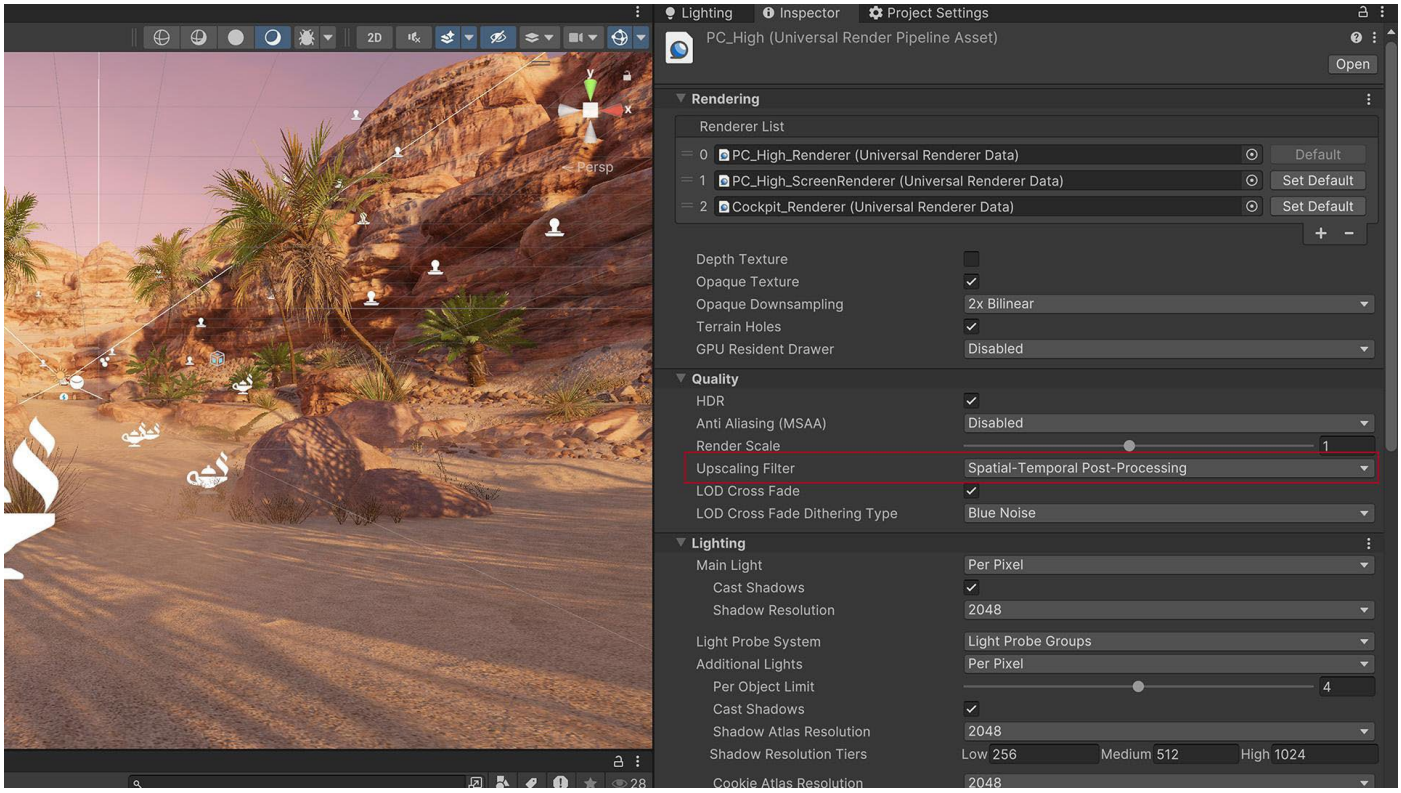
시공간 포스트 프로세싱

STP(시공간 포스트 프로세싱)는 모바일, 콘솔, PC 등 다양한 플랫폼에서 비주얼 품질을 향상하기 위해 설계되었습니다. STP는 HDRP와 URP 렌더 파이프라인 모두에서 사용할 수 있는 시공간 안티앨리어싱 업스케일러로, 기존 콘텐츠를 수정할 필요 없는 고품질 콘텐츠 스케일링을 제공합니다. 특히 GPU 성능을 위해 최적화된 솔루션으로, 렌더링 시간을 단축하고 비주얼 품질을 유지하면서 성능을 간단하게 향상할 수 있습니다.

URP에서 STP를 활성화하려면 다음 단계를 따르세요.

— 프로젝트 창에서 활성 URP 에셋을 선택합니다.

— 인스펙터에서 Quality > Upscaling Filter로 이동하고 Spatial-Temporal Post-Processing을 선택합니다.



URP 에셋에서 STP 활성화

일반적인 조명 최적화

조명은 매우 광범위한 주제이지만, 아래의 일반적인 팁을 활용하면 리소스를 최적화하는 데 도움이 됩니다.

라이트맵 베이크

조명을 만드는 가장 빠른 방법은 바로 프레임별 계산이 필요 없는 방식입니다. 프레임별로 계산하지 않으려면 실시간으로 조명을 계산하는 대신 라이트매핑을 사용하여 정적 조명을 한 번만 베이크하면 됩니다.

GI(전역 조명)를 사용하여 정적 지오메트리에 극적인 조명을 추가해 보세요. 오브젝트를 **Contribute GI**로 표시하면 고품질 조명을 라이트맵 형태로 저장할 수 있습니다.

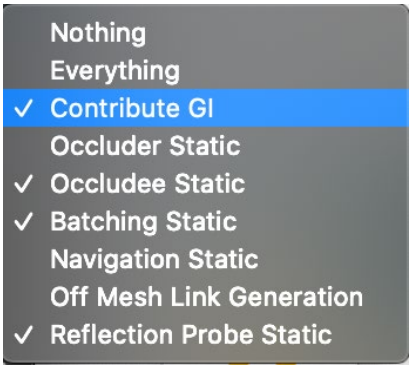
Unity에서 라이트매핑된 환경을 제작하는 과정은 씬에 광원을 배치하는 것보다 시간이 오래 걸리지만, 다음과 같은 장점이 있습니다.

— 실행 속도가 더 빠르며, 픽셀당 광원이 2개인 경우 2~3배는 더 빠릅니다.

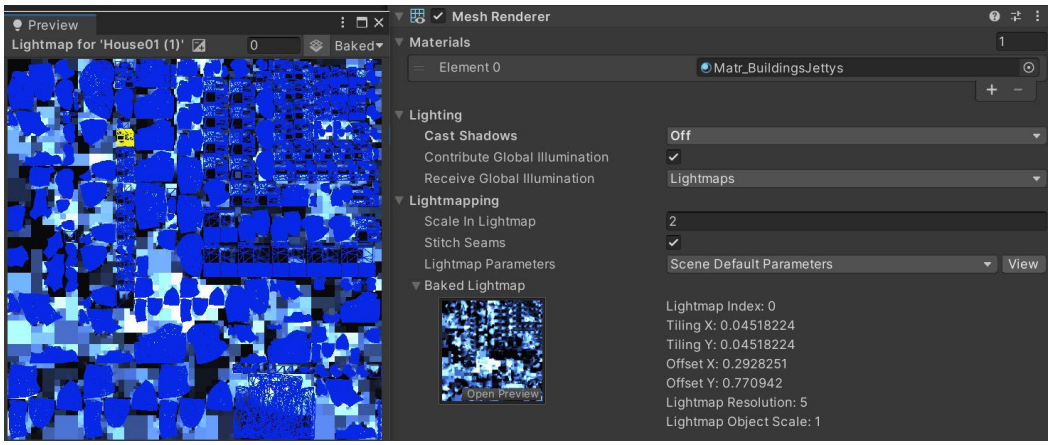
— GI는 사실적인 직접 조명과 간접 조명을 계산할 수 있으므로, 더 멋진 조명을 만들 수 있습니다. 라이트매퍼는 맵을 더 부드럽게 만들고 노이즈를 없애 줍니다.

그러므로 베이크된 그림자와 조명을 렌더링할 때 실시간 조명과 그림자처럼 성능 저하가 발생하지 않습니다.

복잡한 씬은 베이크하는 데 더 오랜 시간이 걸릴 수 있습니다. 하드웨어가 **프로그레시브 GPU 라이트매퍼**를 지원하는 경우, 이를 사용해 라이트맵 생성 속도를 최대 10배까지 높일 수 있습니다.



Contribute GI 활성화



라이트매핑 설정(**Windows > Rendering > Lighting Settings**)과 라이트맵 크기를 조정하여 메모리 사용량 제한

Unity에서 라이트매핑을 시작하려면 매뉴얼을 확인하세요.

반사 프로브 최소화

Reflection Probe 컴포넌트는 사실적인 반사 효과를 만들 수 있지만, 배치 처리 시 소모되는 리소스가 매우 높습니다. 저해상도 큐브맵, 컬링 마스크, 텍스처 압축을 사용해 런타임 성능을 높여 보세요. **Type: Baked**를 사용하면 프레임별 업데이트를 방지할 수 있습니다.

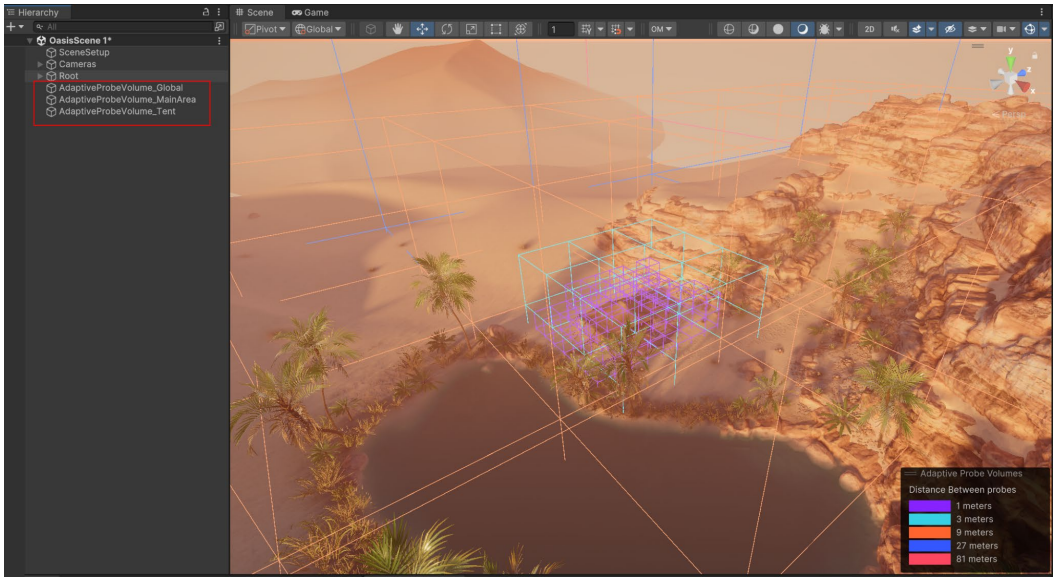
HDRP에서 **Type: Realtime**을 사용해야 하는 경우, 가능하면 **Every Frame**을 사용하지 않도록 하세요. 업데이트 속도를 줄이려면 Refresh Mode 설정과 Time Slicing 설정을 조정합니다. Via Scripting 옵션으로 새로고침을 제어하고 커스텀 스크립트를 통해 프로브를 렌더링할 수도 있습니다.

HDRP에서 **Type: Realtime**을 사용해야 하는 경우 **On Demand** 모드를 사용하세요. 또한 **Project Settings > HDRP Default Settings**에서 Frame Settings를 변경할 수 있습니다. 성능을 개선하려면 Realtime Reflection에 있는 품질과 기능을 하향 조정하세요.

적응적 프로브 볼륨

Unity 6에는 전역 조명을 처리하는 정교한 솔루션인 APV(적응적 프로브 볼륨)가 추가되어 복잡한 씬에서 효율적인 동적 조명을 구현할 수 있습니다. APV는 특히 모바일 및 저사양 기기에서 성능과 비주얼 품질을 모두 최적화할 수 있으며, 동시에 고사양 플랫폼을 위한 고급 기능도 제공합니다.

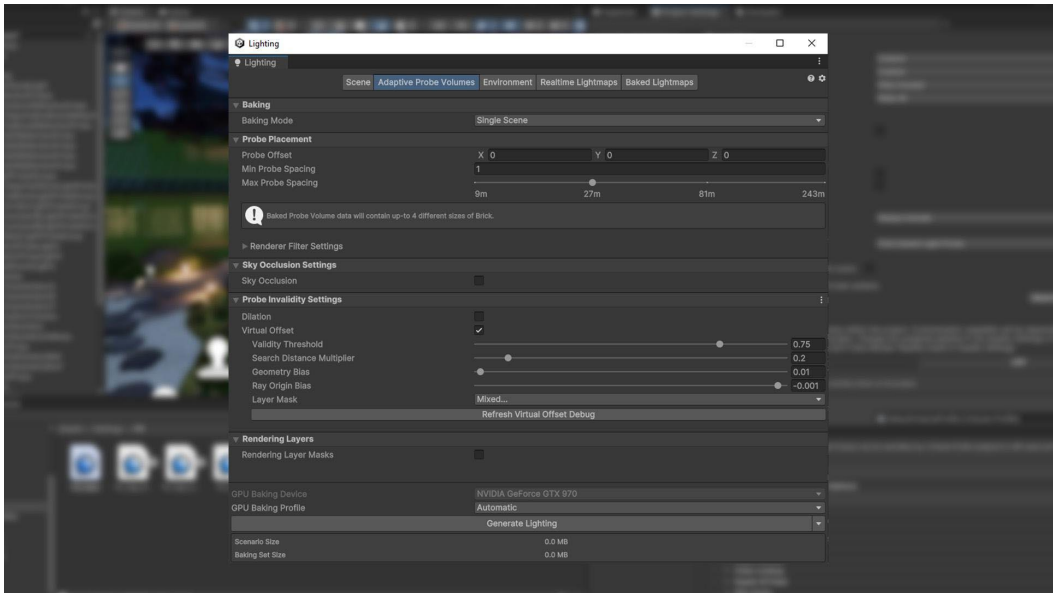
APV는 특히 동적인 대형 씬에서 전역 조명을 향상하는 다양한 기능을 제공합니다. URP는 이제 저사양 기기에서 성능을 향상할 수 있는 버텍스별 샘플링을 지원하며, VFX 파티클은 프로브 볼륨에 베이크되는 간접 조명의 이점을 활용할 수 있습니다.



APV를 여러 개 사용하면 레벨의 프로브 밀도를 더 세밀하게 제어할 수 있습니다.

APV 데이터를 디스크에서 CPU와 GPU로 스트리밍하여 대형 환경에서 조명 정보를 최적화할 수 있습니다. 개발자가 여러 조명 시나리오를 베이크하고 블렌딩하여 낮/밤 주기 같은 실시간 전환을 구현할 수 있습니다. 또한 스카이 오클루전을 지원하고, Ray Intersector API와 통합되어 프로브를 효율적으로 계산할 수 있으며, 라이트 프로브 샘플링 밀도를 제어하여 빛 번짐 효과를 줄이고 빠르게 반복 작업할 수 있습니다. 새로운 C# 베이크 API를 사용하면 라이트맵이나 반사 프로브로부터 독립적으로 APV를 베이크할 수도 있습니다.

APV를 사용해 보려면 GDC 2023의 적응적 프로브 볼륨으로 효율적이고 효과적인 조명 구현하기(영어) 강연을 시청해 보세요.

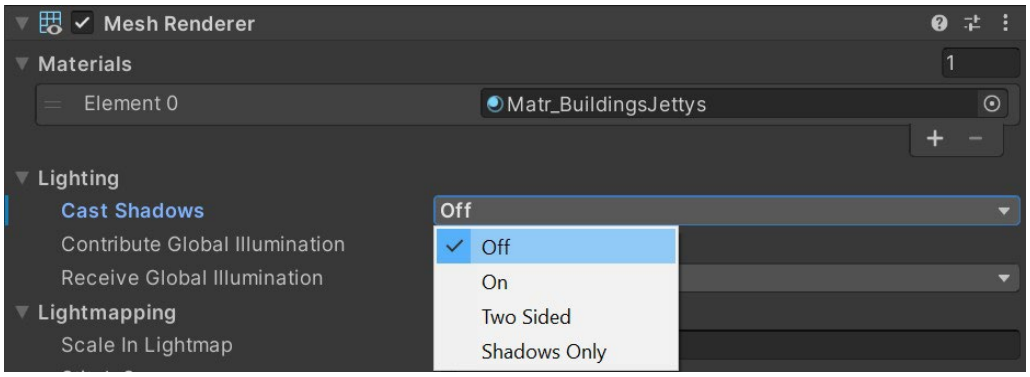


Lighting 설정의 Adaptive Probe Volumes 창

그림자 비활성화

MeshRenderer 및 광원별로 그림자 드리우기를 비활성화할 수 있습니다. 가능하다면 그림자를 비활성화하여 드로우 콜을 줄이세요.

또한 캐릭터 아래의 간단한 메시나 사각형 영역에 블러된 텍스처를 적용하여 가짜 그림자를 만들거나, 커스텀 셰이더로 블롭 섀도우를 만들 수 있습니다.



Cast Shadow를 비활성화하여 드로우 콜 줄이기

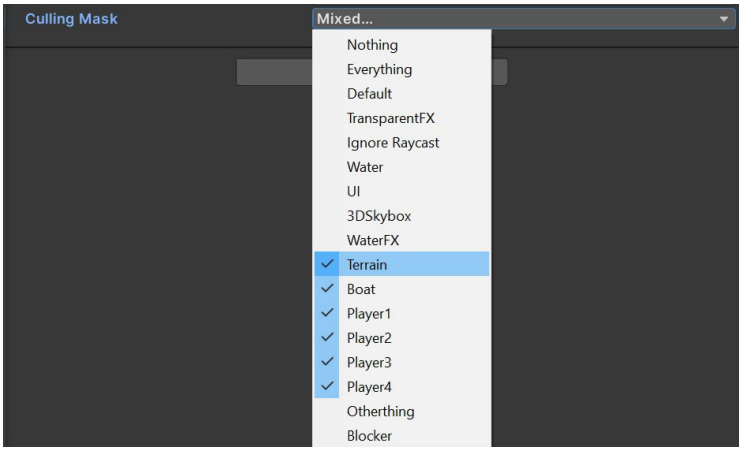
특히 점 광원에 그림자를 활성화하지 않는 것이 좋습니다. 그림자가 있는 점 광원에는 각각 6개의 섀도우맵 패스가 필요한데, 스포트라이트에 하나의 섀도우맵 패스가 필요한 것에 비하면 매우 많은 양입니다. 동적인 그림자가 꼭 필요하다면 점 광원을 스포트라이트로 교체해 보세요. 동적 그림자를 사용하지 않을 수 있다면, 점 광원 대신 큐브맵을 Light.cookie로 사용하는 것이 좋습니다.

셰이더 효과 대체

때에 따라 여러 광원을 추가하지 않고 간단한 기법을 사용할 수 있습니다. 예를 들어 림 라이팅(rim lighting) 효과를 주기 위해 카메라에 직접 비추는 광원을 만드는 대신, 림 라이팅을 시뮬레이션하는 셰이더를 사용할 수 있습니다. 이를 HLSL에서 구현하는 예시는 표면 셰이더 예제를 참조하시기 바랍니다.

광원 레이어 사용

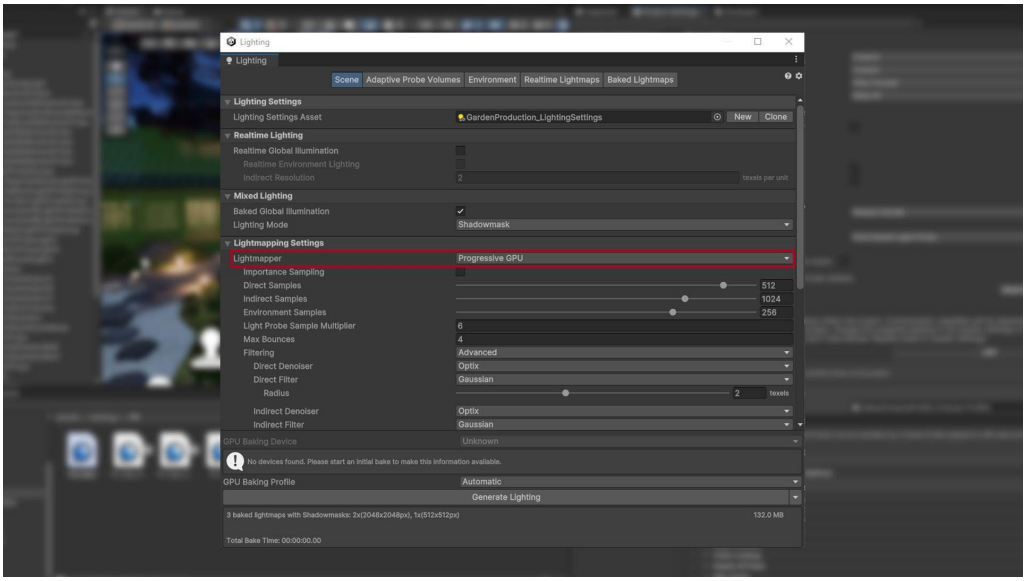
광원이 여러 개인 복잡한 씬의 경우 레이어를 사용해 오브젝트를 분리한 다음 각 광원의 영향을 특정 컬링 마스크로 한정합니다.



레이어를 사용하여 광원의 영향을 특정 컬링 마스크로 제한

GPU 라이트매퍼

Unity 6에서는 GPU 라이트매퍼를 정식으로 제작에 사용 가능합니다. GPU의 성능을 활용해 기존 CPU 라이트매핑보다 훨씬 빠르게 베이크하여 조명 데이터 생성 시간을 크게 단축합니다. 이 시스템에는 코드베이스를 간소화하고 더 예측 가능한 결과를 제공하는 새로운 광원 베이크 백엔드가 포함되어 있습니다. 또한 GPU 최소 요구 사항이 2GB로 낮아졌고, 절차적으로 생성되는 콘텐츠에 특히 유용하도록 런타임에 라이트 프로브 위치를 옮길 수 있는 새로운 API를 제공하며, 이외에도 다양한 사항이 개선되었습니다.



GPU 라이트매퍼 선택

GPU 최적화

그래픽스 렌더링을 최적화하려면 타겟 하드웨어의 제한 사항뿐만 아니라 GPU를 어떻게 프로파일링하는지 이해해야 합니다. 프로파일링을 통해 현재 진행 중인 최적화가 효과적인지 확인하고 검증할 수 있습니다.

아래에서 소개할 베스트 프랙티스를 사용하면 GPU의 렌더링 워크로드를 줄일 수 있습니다.

GPU 벤치마킹

프로파일링할 때 특정 GPU에서 어떤 프로파일링 결과를 기대하면 될지 파악하기 위해 벤치마킹부터 하는 것이 좋습니다.

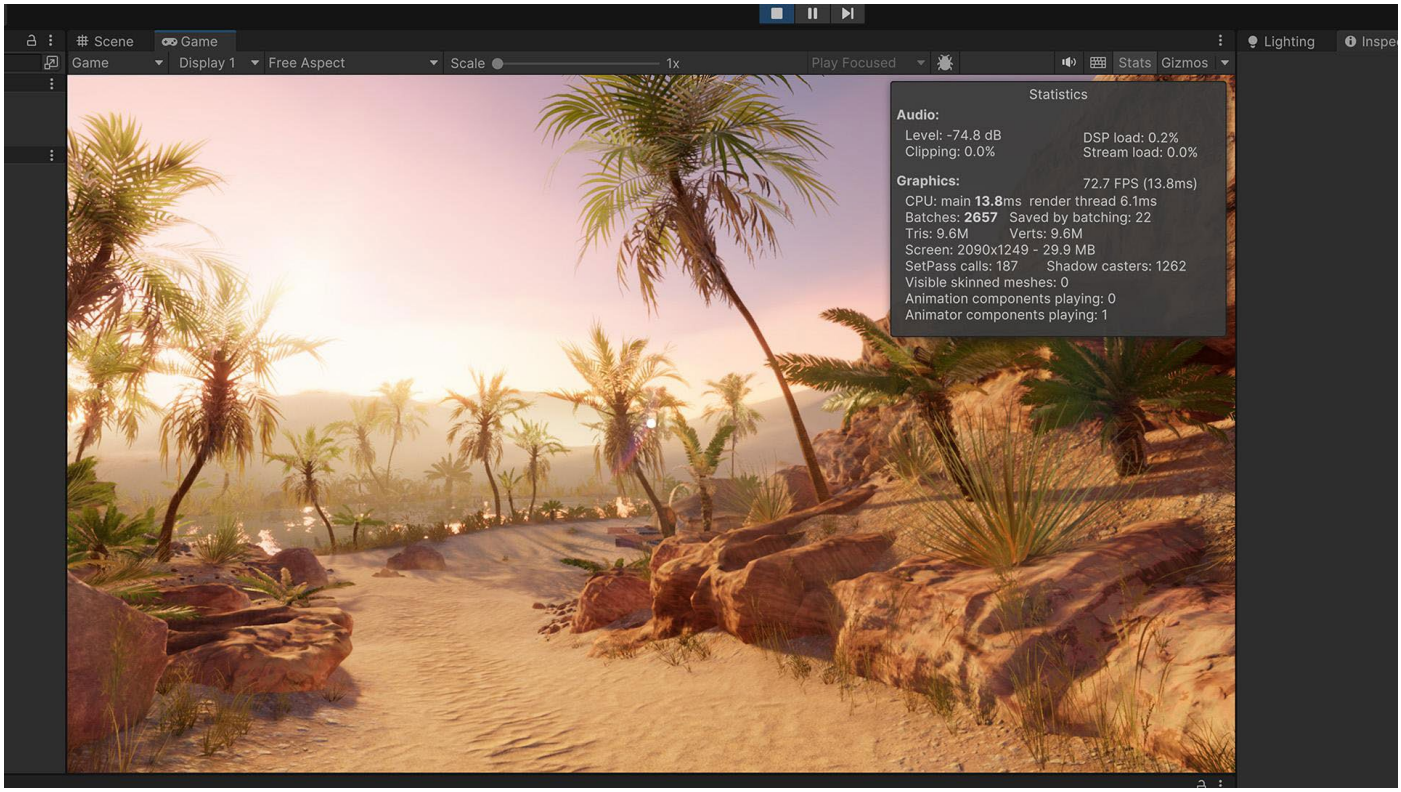
GPU 및 그래픽 카드에 대한 다양한 업계 표준 벤치마크 목록은 GFXBench를 참조하시기 바랍니다. GFXBench에서는 현재 사용할 수 있는 GPU의 개요를 확인하고 여러 GPU를 서로 비교해볼 수 있습니다.

렌더링 통계 확인

게임(Game) 뷰 오른쪽 상단에 있는 **Stats** 버튼을 클릭하세요. 이 창에서는 플레이 모드 동안 애플리케이션에 대한 실시간 렌더링 정보를 확인할 수 있습니다. 이 데이터를 성능 최적화에 사용할 수 있습니다.

- **fps:** 초당 프레임 수
- **CPU Main:** 하나의 프레임을 처리하는 데 걸리는 전체 시간(에디터의 모든 창을 업데이트하는 데 걸리는 시간 포함)
- **CPU Render:** 게임 뷰의 한 프레임을 렌더링하는 데 걸리는 전체 시간
- **Batches:** 함께 드로우할 드로우 콜 그룹
- **Tris(삼각형) 및 Verts(버텍스):** 메시 지오메트리
- **SetPass calls:** Unity가 게임 오브젝트를 화면에 렌더링하기 위해 셰이더 패스를 전환해야 하는 횟수로, 패스마다 추가로 CPU 오버헤드가 생길 수 있습니다.

참고: 에디터 내에서의 FPS가 반드시 빌드 성능으로 연결되지는 않습니다. 가장 정확한 결과를 얻으려면 빌드를 프로파일링해 보세요. 벤치마킹 시 밀리초 단위의 프레임 시간은 초당 프레임 수보다 더 정확한 지표가 됩니다. ‘FPS: 정확하지 않을 수 있는 지표’ 섹션을 참고하세요.



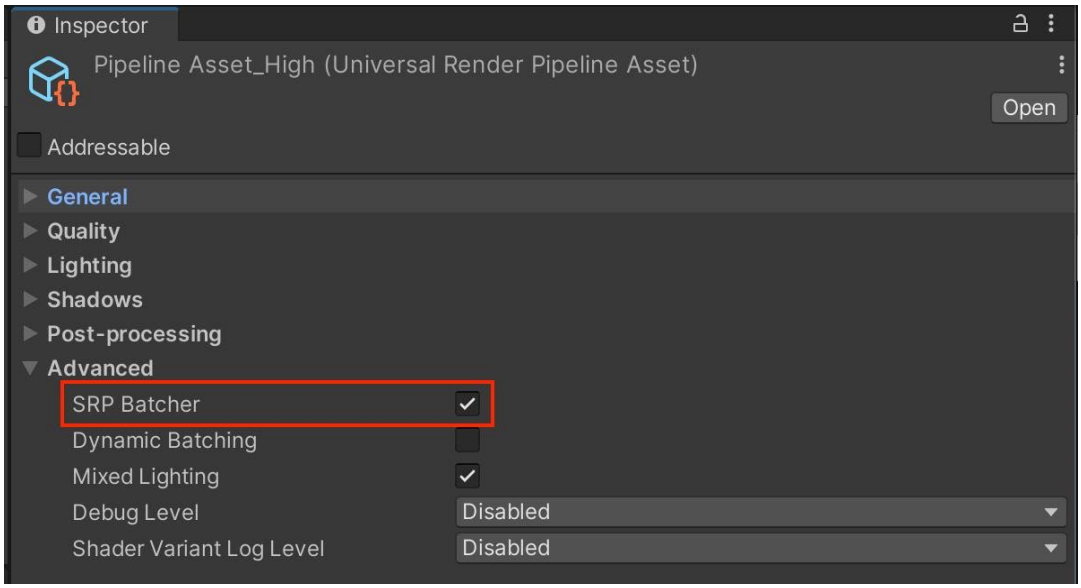
드로우 콜 배칭 사용

Unity는 게임 오브젝트를 드로우하기 위해 그래픽스 API(예: OpenGL, Vulkan, Direct3D)에 드로우 콜을 보냅니다. 각 드로우 콜은 많은 리소스를 소모합니다. 드로우 콜 사이의 상태 변경(예: 머티리얼 전환)은 CPU에 성능 오버헤드를 유발할 수 있습니다.

PC와 콘솔 하드웨어는 많은 드로우 콜을 푸시할 수 있지만, 각 드로우 콜이 높은 오버헤드를 유발하므로 가급적 줄이는 것이 좋습니다. 모바일 기기의 경우 드로우 콜 최적화는 필수적입니다. 드로우 콜 배칭을 통해 드로우 콜을 최적화할 수 있습니다.

드로우 콜 배칭은 상태 변화를 최소화하고 오브젝트 렌더링에 사용하는 CPU 비용을 줄입니다. Unity에서는 몇 가지 기법을 사용하여 다수의 오브젝트를 더 적은 수의 배치로 결합할 수 있습니다.

— **SRP 배칭**: HDRP나 URP를 사용하는 경우, 파이프라인 에셋의 **Advanced**에서 SRP Batcher를 활성화합니다. 호환되는 셰이더를 사용하는 경우, SRP 배처는 드로우 콜 간 GPU 설정을 줄이고 머티리얼 데이터가 GPU 메모리에 유지되도록 만듭니다. 그러면 CPU 렌더링 시간이 훨씬 빨라집니다. 최소한의 키워드로 더 적은 수의 셰이더 배리언트를 사용하면 SRP 배칭을 개선할 수 있습니다. 프로젝트에서 이러한 렌더링 워크플로를 어떻게 활용할 수 있는지 알아보려면 SRP 기술 자료를 참조하세요.



드로우 콜을 배칭하는 데 유용한 SRP 배처

— **GPU 인스턴싱**: 같은 메시와 머티리얼을 사용하는 빌딩, 나무, 풀처럼 동일한 오브젝트의 수가 많은 경우 GPU 인스턴싱을 사용하세요. GPU 인스턴싱을 사용하면 그래픽스 하드웨어를 사용해 오브젝트를 배칭합니다. GPU 인스턴싱을 활성화하려면 프로젝트(Project) 창에서 머티리얼을 선택하고 인스펙터(Inspector)에서 **Enable Instancing**을 선택하세요.

— **정적 배칭**: 움직이지 않는 지오메트리의 경우 Unity는 동일한 머티리얼을 공유하는 메시에 대한 드로우 콜을 줄일 수 있습니다. 동적 배칭에 비해 더 효율적이지만, 메모리 사용량은 늘어납니다.

인스펙터에서 절대 움직이지 않는 모든 메시를 **Batching Static**으로 표시하세요. Unity는 빌드 중에 모든 정적 메시를 하나의 커다란 메시로 결합합니다. StaticBatchingUtility를 사용하면 런타임에 이러한 정적 배치를 직접 만들 수도 있습니다. 예를 들어 움직이지 않는 부분의 절차적 레벨을 생성한 후 정적 배치를 만들 수 있습니다.

— **동적 배칭**: 작은 메시의 경우 Unity는 CPU에서 버텍스를 그룹화하고 변환한 다음 모두 한 번에 드로우할 수 있습니다. 참고: 로우 폴리곤 메시가 충분하지 않다면(버텍스 300개 이하, 전체 버텍스 속성 900개 이하) 동적 배칭을 사용하지 마세요. 메시가 충분하지 않은 상황에서 동적 배칭을 사용하면 배칭할 작은 메시를 찾는 데 CPU 시간을 낭비하게 됩니다.

몇 가지 간단한 규칙으로 배칭을 최대화할 수 있습니다.

— 씬에서 가능한 한 적은 수의 텍스처를 사용하세요. 텍스처가 적을수록 고유한 머티리얼이 적게 필요하여 더 쉽게 배칭할 수 있습니다. 또한 가능하면 텍스처 아틀라스를 사용하세요.

— 라이트맵은 항상 최대한 가장 큰 아틀라스 크기로 베이크하세요. 라이트맵이 적을수록 머티리얼 상태 변경이 줄어들지만, 메모리 사용량을 주의 깊게 살펴야 합니다.

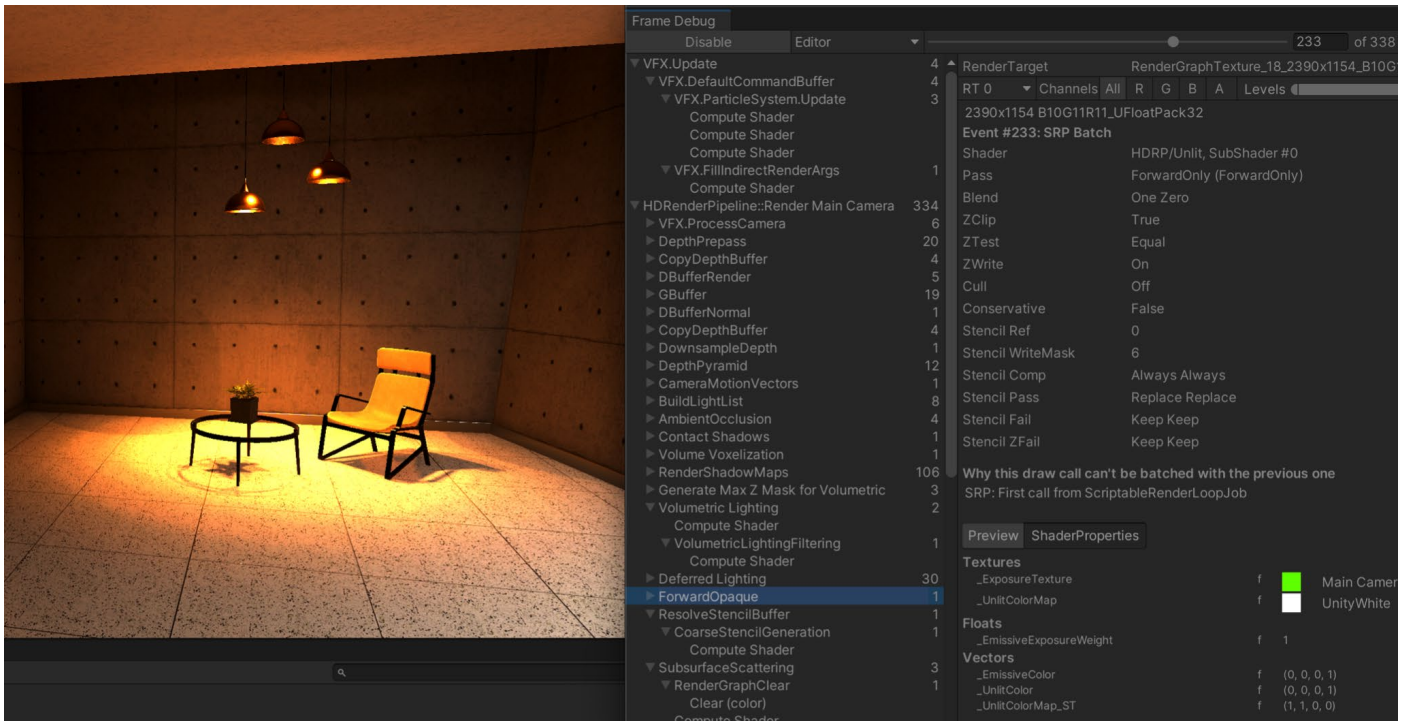
— 실수로 머티리얼을 인스턴스화하지 않도록 주의하세요. 스크립트의 Renderer.material에 액세스하면 머티리얼이 복사되고 새로운 사본에 대한 참조가 반환됩니다. 이러면 해당 머티리얼이 포함되어 있는 기존 배칭이 중단됩니다. 배칭된 오브젝트의 머티리얼에 액세스하려면Renderer.sharedMaterial을 대신 사용하세요.

— 최적화 중에 프로파일러나 렌더링 통계를 사용하여 정적 배치 및 동적 배치의 수와 전체 드로우 콜 수를 계속해서 비교하세요.

자세한 내용은 드로우 콜 배칭 기술 자료를 참조하시기 바랍니다.

프레임 디버거 확인

프레임 디버거를 사용하면 특정 프레임에서 재생을 멈추고 씬이 어떻게 구성되는지 단계별로 살펴보면서 최적화 가능한 부분을 식별할 수 있습니다. 불필요하게 렌더링되는 게임 오브젝트를 찾아 비활성화하여 프레임당 드로우 콜을 줄여 보세요.



렌더링된 각 프레임을 분석해서 보여 주는 프레임 디버거

> 참고: 프레임 디버거는 개별 드로우 콜이나 상태 변화를 표시하지 않습니다. 상세한 드로우 콜 및 타이밍 정보를 제공하는 것은 네이티브 GPU 프로파일러뿐입니다. 그러나 프레임 디버거는 파이프라인 문제나 배칭 문제를 디버깅하는 데 유용하게 사용할 수 있습니다.

Unity 프레임 디버거의 장점은 씬의 특정 게임 오브젝트에 드로우 콜을 연결할 수 있다는 점입니다. 게임 오브젝트에 드로우 콜을 연결하면 외부 프레임 디버거에서 확인할 수 없는 특정 문제를 더 쉽게 확인할 수 있습니다.

더 자세히 알아보려면 프레임 디버거 기술 자료를 참고하고, 플랫폼별 디버깅 툴의 목록을 확인하려면 '네이티브 프로파일링 및 디버깅 툴 사용' 섹션을 참고하세요.

필 레이트 최적화 및 오버드로우 줄이기

필 레이트란 GPU가 초당 화면에 렌더링할 수 있는 픽셀 수를 의미합니다.

게임이 필 레이트에 의해 제한된다면, 게임이 GPU가 처리할 수 있는 것보다 더 많은 프레임당 픽셀을 드로우하려고 한다는 의미입니다.

같은 픽셀에 여러 번 드로우하는 것을 오버드로우라고 하는데, 오버드로우는 필 레이트를 낮추고 메모리 대역폭을 추가로 사용합니다. 오버드로우가 발생하는 가장 일반적인 원인은 아래와 같습니다.

- 불투명 지오메트리 또는 투명 지오메트리 겹침
- 여러 렌더 패스가 있는 복잡한 셰이더
- 최적화되지 않은 파티클
- UI 요소 겹침

오버드로우의 영향을 최소화하는 것이 바람직하지만 모든 오버드로우 문제를 한 번에 해결하는 방법은 없습니다. 우선 위와 같은 요소들을 확인해 가며 영향을 줄여야 합니다.

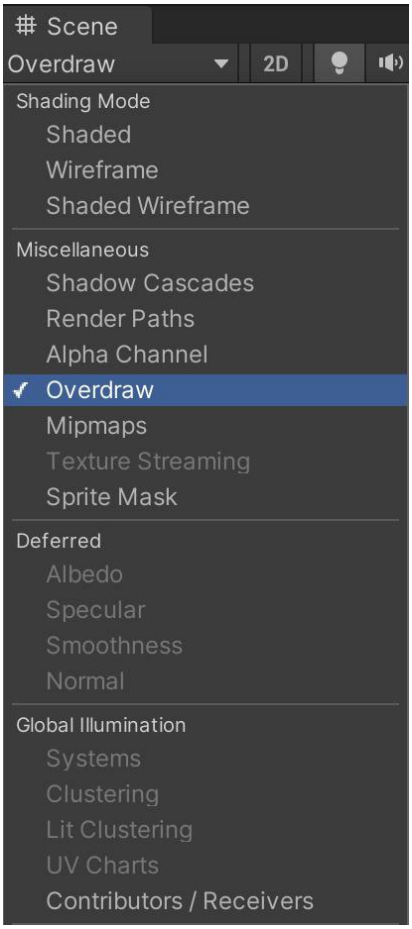
드로잉 순서 및 렌더링 대기열

오버드로우를 해결하려면 Unity가 오브젝트를 렌더링하기 전에 어떻게 정렬하는지 알아야 합니다.

빌트인 렌더 파이프라인은 렌더링 모드와 renderQueue에 따라 게임 오브젝트를 정렬합니다. 각 오브젝트의 셰이더는 렌더링 대기열에 오브젝트를 추가하고, 여기서 드로잉 순서가 결정되는 경우가 많습니다.

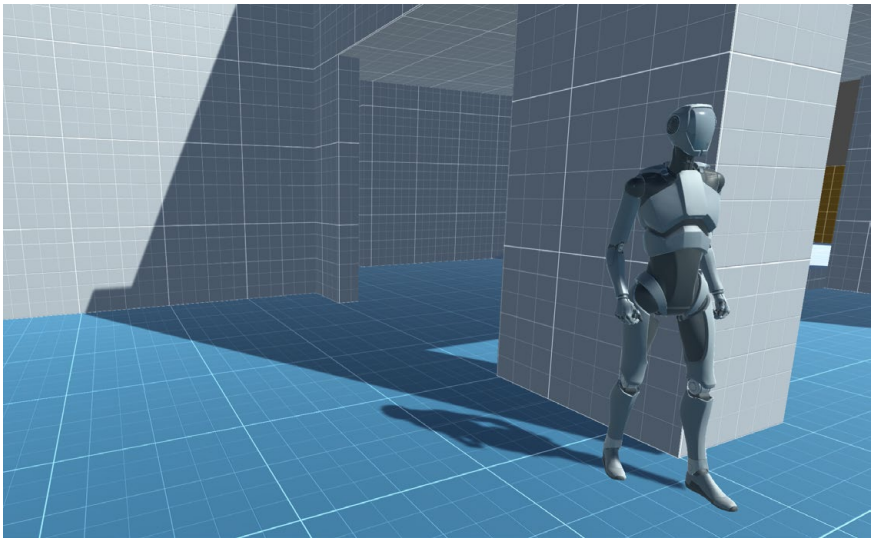
각 렌더링 대기열은 Unity가 실제로 오브젝트를 화면에 드로우하기 전에 다양한 정렬 규칙을 따를 수 있습니다. 예를 들어 Unity는 불투명 지오메트리 대기열을 앞에서 뒤로 정렬하고, 투명 지오메트리 대기열은 뒤에서 앞으로 정렬할 수 있습니다.

다른 오브젝트 위에 오브젝트가 렌더링되면 오버드로우가 발생합니다. 빌트인 렌더 파이프라인을 사용하는 경우, 씬 뷰의 컨트롤 바에서 오버드로우를 시각화할 수 있습니다. 드로우 모드를 Overdraw로 변경하면 됩니다.

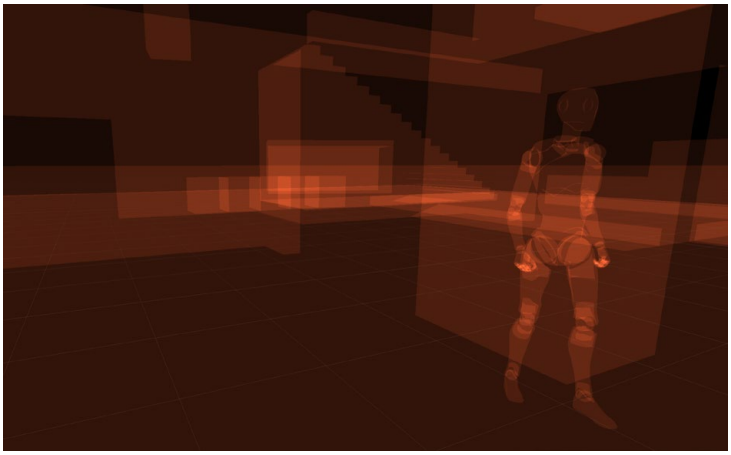


씬 뷰 컨트롤 바에 있는 Overdraw 옵션

밝은 픽셀은 오브젝트가 다른 오브젝트 위에 드로우되었음을 나타내며, 어두운 픽셀은 오버드로우가 적게 발생한다는 의미입니다.



표준 Shaded 뷰로 본 씬



같은 씬을 Overdraw 뷰로 본 모습. 겹치는 지오메트리는 보통 오버드로우의 원인이 됩니다.

HDRP는 렌더링 대기열을 조금 다르게 제어합니다. 렌더링 대기열의 순서를 계산하기 위해 HDRP는 다음을 수행합니다.

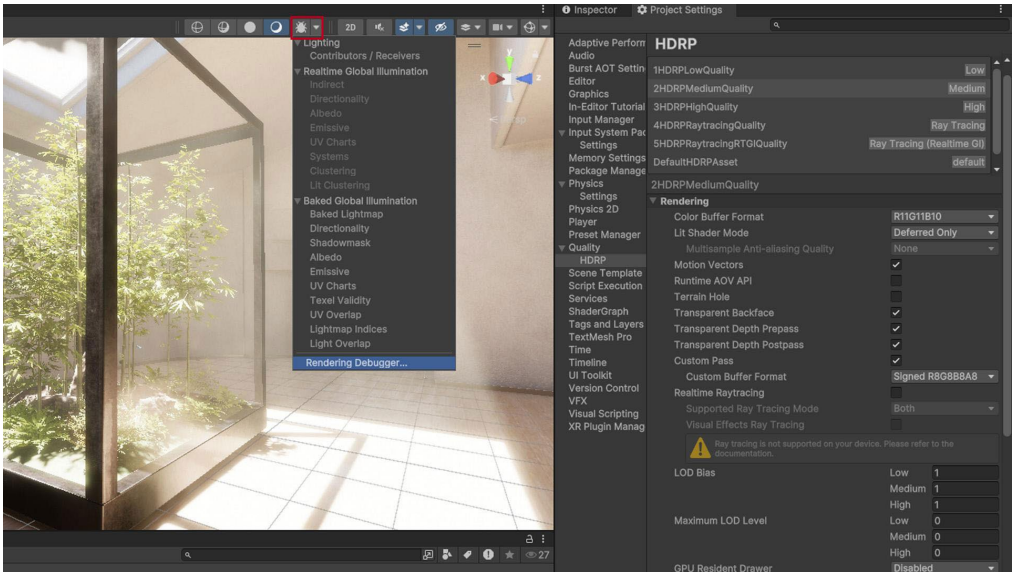
— 공유된 머티리얼별로 메시를 그룹화합니다.

— 머티리얼 우선순위에 따라 해당 그룹의 렌더링 순서를 계산합니다.

— 각 메시 렌더러의 Priority 프로퍼티를 사용하여 각 그룹을 정렬합니다.

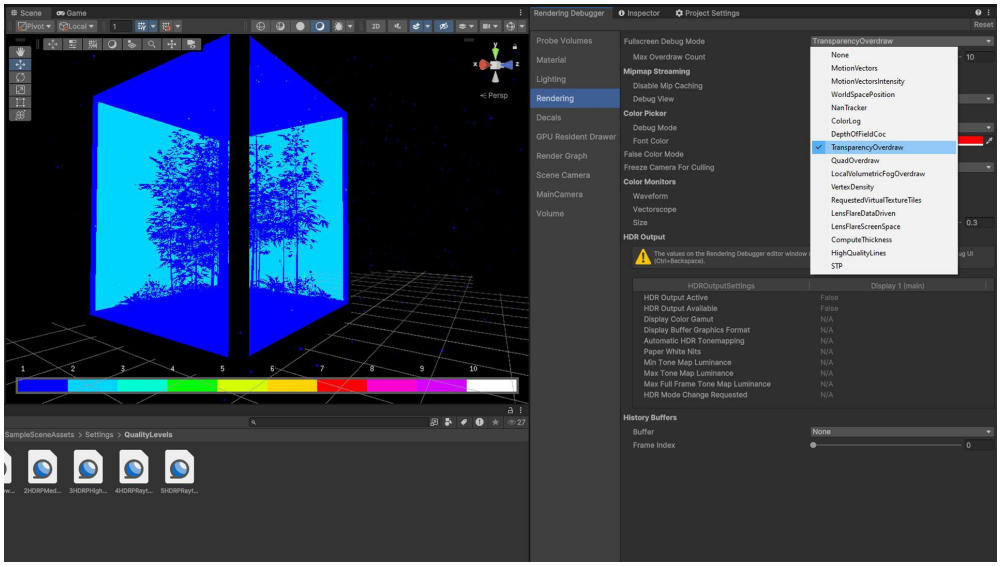
그 결과 대기열의 게임 오브젝트는 먼저 머티리얼의 우선순위로 정렬된 다음 개별 메시 렌더러의 우선순위에 따라 정렬됩니다. 렌더러와 머티리얼 우선순위를 설명하는 페이지에서 더 자세히 알아보세요.

HDRP로 투명 오버드로우를 시각화하려면, Render Pipeline Debug 창(**Window > Render Pipeline > Render Pipeline Debug**)에서 TransparencyOverdraw를 선택합니다.

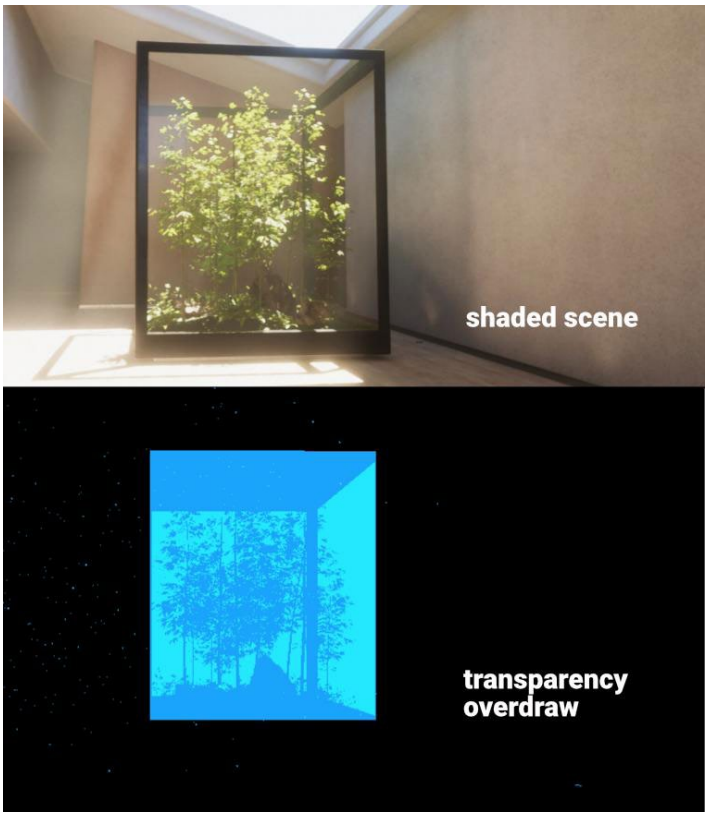


렌더링 디버거 사용하기

투명 오버드로우를 시각화하려면 Rendering을 선택하고 Fullscreen Debug Mode 드롭다운에서 TransparencyOverdraw를 선택합니다.



투명 오버드로우 시각화



투명 머티리얼의 오버드로우를 시각화할 수 있는 HDRP Render Pipeline Debug 창

히트맵이 씬에서 렌더링되는 투명 레이어를 파란색에서 흰색으로 표현합니다. 파란색은 오버드로우가 적은 영역이며 흰색은 성능에 영향을 주는 주요 투명 레이어를 나타냅니다.

투명 오버드로우를 시각화하면 개발자가 성능을 향상하기 위해 씬을 최적화할 방법을 정보에 기반하여 결정할 수 있습니다.

오버드로우를 보정할 때 진단 툴을 사용하면 최적화를 위한 시각적 지표를 얻을 수 있습니다.

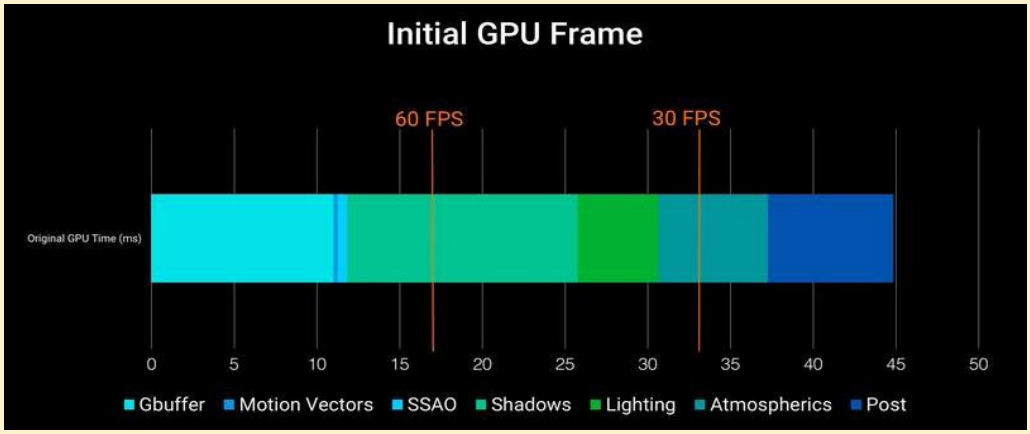
콘솔을 위한 그래픽스 최적화

Xbox 콘솔 및 PlayStation®용 게임의 개발 과정은 PC 게임 개발과 비슷하지만 플랫폼마다 고유한 기법이 필요합니다. 원활한 프레임 속도를 확보하기 위해 GPU 최적화에 집중해야 하는 경우가 많습니다.

성능 병목 지점 식별

Xbox용 PIX 및 PlayStation 프로파일러 툴을 툴박스에 추가하여 각 플랫폼에서 최적화 작업을 진행할 때 사용하세요.

우선 GPU 로드가 높은 프레임을 찾습니다. Microsoft와 Sony에서는 CPU 및 GPU에 대한 프로젝트의 성능을 분석할 수 있는 훌륭한 툴을 제공합니다. 각 네이티브 프로파일러를 사용하여 프레임 비용을 세부 분류하면 그래픽스 성능을 개선하기 위한 기본 정보로 사용할 수 있습니다.



PlayStation®4 Pro에서 프레임당 약 45밀리초로 GPU 바운드된 모습

배치 수 줄이기

다른 플랫폼과 마찬가지로 콘솔에서도 최적화 작업 시 드로우 콜 배치를 줄여야 하는 경우가 많습니다. 이때 유용한 몇 가지 기술이 있습니다.

— 오클루전 컬링을 사용하면 전경 오브젝트에 가려진 오브젝트를 제거하고 오버드로우를 줄일 수 있습니다. 이 작업을 하려면 추가로 CPU 처리가 필요하므로, 프로파일러를 사용해 작업을 GPU에서 CPU로 옮기는 것의 이점이 있어야 합니다.

— 다수의 오브젝트가 같은 메시와 머티리얼을 공유한다면 GPU 인스턴싱으로 배치를 줄일 수도 있습니다. 씬에서 모델 수를 제한하면 성능을 개선하는 데 도움이 됩니다. 세심하게 처리하면 반복되는 느낌 없이 복잡한 씬을 제작할 수 있습니다.

— SRP 배처는 Bind 및 Draw GPU 커맨드를 배칭하여 드로우 콜 사이의 GPU 설정을 줄입니다. 이러한 SRP 배칭의 장점을 활용하려면 머티리얼을 필요한 만큼 사용하되, 서로 호환되는 소수의 셰이더(예: URP와 HDRP의 릿 셰이더와 언릿 셰이더)와 함께 사용하시기 바랍니다.

Graphics Jobs 활성화

Player Settings > Other Settings에서 이 옵션을 활성화하여 PlayStation이나 Xbox 콘솔의 멀티 코어 프로세서를 활용하세요. **Graphics Jobs**를 사용하면 Unity에서 여러 CPU 코어에 렌더링 작업을 분산하므로 렌더 스레드의 부담을 해소할 수 있습니다. 자세한 내용은 멀티스레드 렌더링 및 Graphics Jobs 튜토리얼(영문)을 참조하시기 바랍니다.

포스트 프로세싱 프로파일링

콘솔에 최적화된 포스트 프로세싱 에셋을 사용하세요. PC용으로 저작된 에셋 스토어의 툴은 Xbox 콘솔이나 PlayStation에서 필요 이상으로 많은 리소스를 소모할 수 있습니다. 네이티브 프로파일러로 프로파일링하여 확인하세요.

테셀레이션 셰이더 사용 지양

테셀레이션은 셰이프를 더 작은 버전의 셰이프로 분할하며, 지오메트리 수를 늘려 디테일 수준을 높일 수 있습니다. 테셀레이션을 사용하는 것이 적절한 경우도 있지만(예: 나무껍질 구현) 콘솔에서는 보통 테셀레이션 사용이 권장되지 않습니다. 테셀레이션을 사용하면 GPU에서 많은 리소스를 소모할 수 있기 때문입니다.

지오메트리 셰이더를 컴퓨트 셰이더로 교체

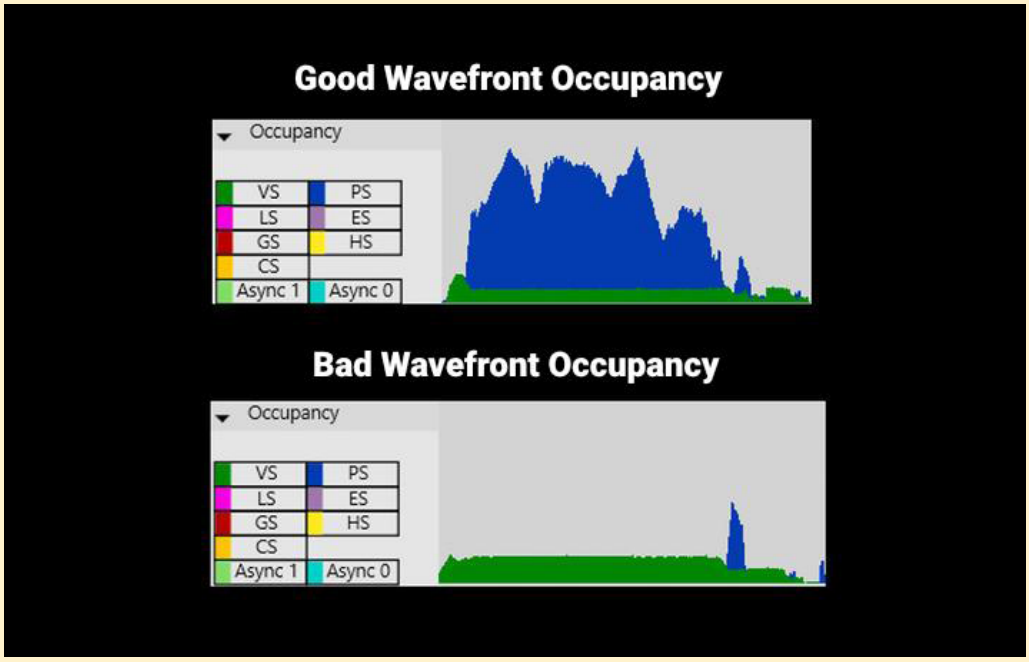
테셀레이션 셰이더와 마찬가지로 지오메트리 셰이더와 버텍스 셰이더는 GPU에서 프레임당 두 번 실행될 수 있습니다. 뎁스 프리패스 시 한 번 실행되고 섀도우 패스 시 다시 한 번 실행되기 때문입니다.

GPU에서 버텍스 데이터를 생성하거나 수정하고 싶다면, 지오메트리 셰이더보다 컴퓨트 셰이더를 사용하는 편이 바람직한 경우가 많습니다. 컴퓨트 셰이더에서 작업하면 실제로 지오메트리를 렌더링하는 버텍스 셰이더가 상대적으로 빠르고 간소해집니다.

웨이브프론트 점유율 개선

GPU에 드로우 콜을 보내면, Unity에서 GPU 내 사용 가능한 SIMD 전체에 배포하는 여러 웨이브프론트(wavefront)에 작업이 분할됩니다.

각 SIMD에는 동시 실행 가능한 최대 웨이브프론트 수가 있습니다. 웨이브프론트 점유율은 이러한 최대치와 비교하여 현재 사용 중인 웨이브프론트의 수를 나타냅니다. 이를 통해 GPU 성능이 얼마나 충분히 활용되고 있는지 측정할 수 있습니다. 콘솔별 성능 분석 툴을 사용하여 웨이브프론트 점유율을 상세히 확인할 수 있습니다.



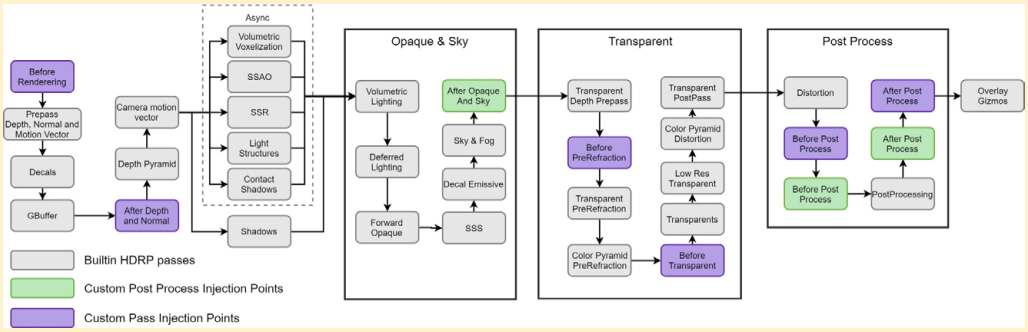
바람직한 웨이브프론트 점유율 예시와 그렇지 못한 경우의 예시

위 예시에서는 버텍스 셰이더 웨이브프론트가 초록색으로 표시됩니다. 픽셀 셰이더 웨이브프론트는 파란색으로 표시되어 있습니다. 아래쪽 그래프를 보면, 픽셀 셰이더 활동이 별로 없는데 상당한 양의 버텍스 셰이더 웨이브프론트가 나타납니다. 따라서 GPU 성능 활용도가 낮음을 알 수 있습니다.

픽셀로 표시되지 않는 버텍스 셰이더 작업이 많다면 이는 비효율적이라는 의미일 수 있습니다. 웨이브프론트 점유율이 낮다는 사실이 무조건 나쁜 것은 아니지만, 셰이더를 최적화하고 다른 병목 현상을 확인하는 지표로 활용할 수 있습니다. 예를 들어 메모리나 컴퓨팅 작업으로 인해 지연이 생긴다면, 점유율을 높여 성능을 개선할 수 있습니다. 반면에 너무 많은 웨이브프론트가 실행된 경우에도 캐시 스래싱(cache thrashing)이 발생하고 성능이 저하될 수 있습니다.

HDRP 빌트인 패스와 커스텀 패스 사용

프로젝트에서 HDRP를 사용한다면 빌트인 패스와 커스텀 패스를 활용하세요. 씬 렌더링에 도움이 될 수 있습니다. 빌트인 패스는 셰이더를 최적화하는 데 도움이 됩니다. HDRP는 셰이더에 커스텀 패스를 추가할 수 있는 여러 인젝션 포인트를 포함하고 있습니다.



HDRP 인젝션 포인트를 사용하여 파이프라인 커스터마이징

투명 머티리얼의 동작을 최적화하려면, 렌더러 및 머티리얼 우선순위(영문) 페이지를 참조하시기 바랍니다.

섀도우 매핑 렌더 타겟의 크기 줄이기

HDRP의 High Quality 설정에서는 기본적으로 4K 섀도우 맵을 사용합니다. 섀도우 맵 해상도를 낮추고 프레임 비용에 미치는 영향을 측정하세요. 참고로 광원 설정을 통해 비주얼 품질의 변화를 보완해야 할 수 있습니다.

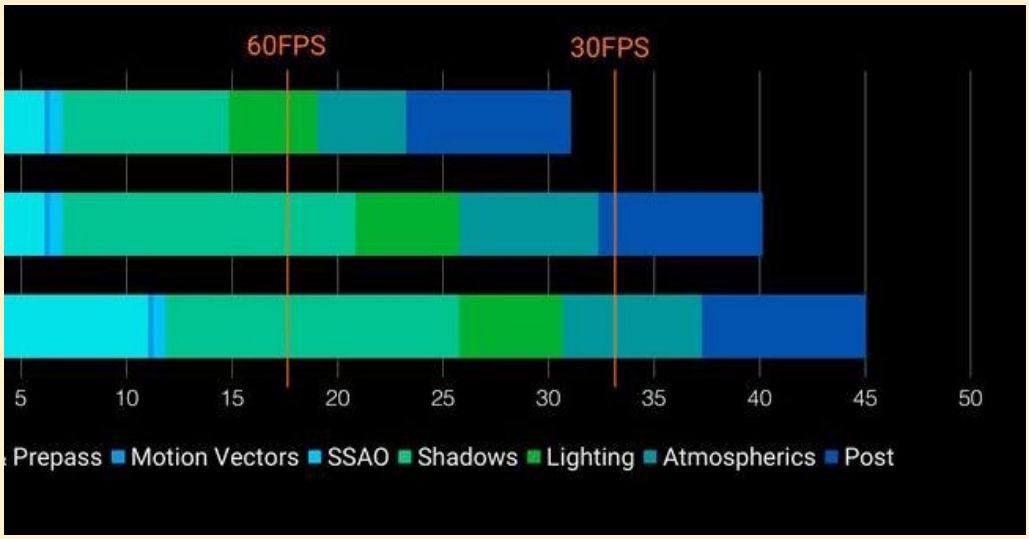
비동기 컴퓨트 기능 활용

GPU 사용률이 낮은 구간이 있을 때 비동기 컴퓨트(Async Compute) 기능을 활용해 보세요. 유용한 컴퓨트 셰이더 작업을 그래픽스 대기열에 병렬로 이동할 수 있고, 이를 통해 GPU 리소스 활용을 개선할 수 있습니다.

예를 들어 GPU는 섀도우 맵이 생성되는 동안 뎁스 전용 렌더링을 수행합니다. 이때 픽셀 셰이더 작업은 거의 수행되지 않으며, 많은 웨이브프론트가 비점유 상태로 유지됩니다.

비동기 컴퓨트를 통해 컴퓨트 셰이더 작업을 그래픽스 대기열로 병렬 이동

일부 컴퓨트 셰이더 작업을 뎁스 전용 렌더링과 동기화하면 전반적인 GPU 사용률을 개선할 수 있습니다. 사용되지 않는 웨이브프론트를 스크린 공간 앰비언트 오클루전이나 진행 중인 작업을 보완하는 데 활용할 수도 있습니다.



30fps에서 최적화된 렌더링

위 예시에서는 여러 최적화를 통해 섀도우 매핑, 조명 패스, 대기 구현에서 수 밀리초를 단축했습니다. 그 결과 프레임 비용이 적어져 PlayStation®4 Pro에서 30fps로 애플리케이션을 실행할 수 있게 되었습니다.

컬링

오클루전 컬링은 다른 게임 오브젝트로 완전히 가려진 게임 오브젝트를 비활성화합니다. 이렇게 하면 CPU와 GPU가 카메라에 보이지 않는 오브젝트를 렌더링하는 데 불필요한 시간을 들이지 않습니다.

컬링은 카메라별로 발생합니다. 컬링은 성능에 많은 영향을 줄 수 있으며, 특히 여러 카메라를 동시에 사용하는 경우 상당한 영향을 줍니다. Unity는 두 가지 컬링, 절두체 컬링과 오클루전 컬링을 사용합니다.

절두체 컬링은 모든 카메라에서 자동으로 수행됩니다. 뷰 절두체 외부의 게임 오브젝트가 렌더링되지 않게 만들어 성능 최적화를 돕습니다.

Camera.layerCullDistances를 사용해 수동으로 레이어별 컬링 거리를 설정할 수 있습니다. 그러면 기본 farClipPlane보다 짧은 거리에서 작은 게임 오브젝트를 컬링할 수 있습니다.

게임 오브젝트를 레이어로 정리하고, layerCullDistances 배열을 사용하여 32개의 레이어에 각각 farClipPlane보다 작은 값을 할당하면 됩니다. 0을 할당하면 farClipPlane을 기본값으로 사용합니다.

Unity는 먼저 레이어별로 컬링하여 카메라가 사용하는 레이어상의 게임 오브젝트만 유지합니다. 그런 다음 절두체 컬링을 통해 카메라 절두체 외부에 있는 모든 게임 오브젝트가 제거됩니다. 절두체 컬링은 사용할 수 있는 워커 스레드를 활용하기 위해 일련의 잡으로 수행됩니다.

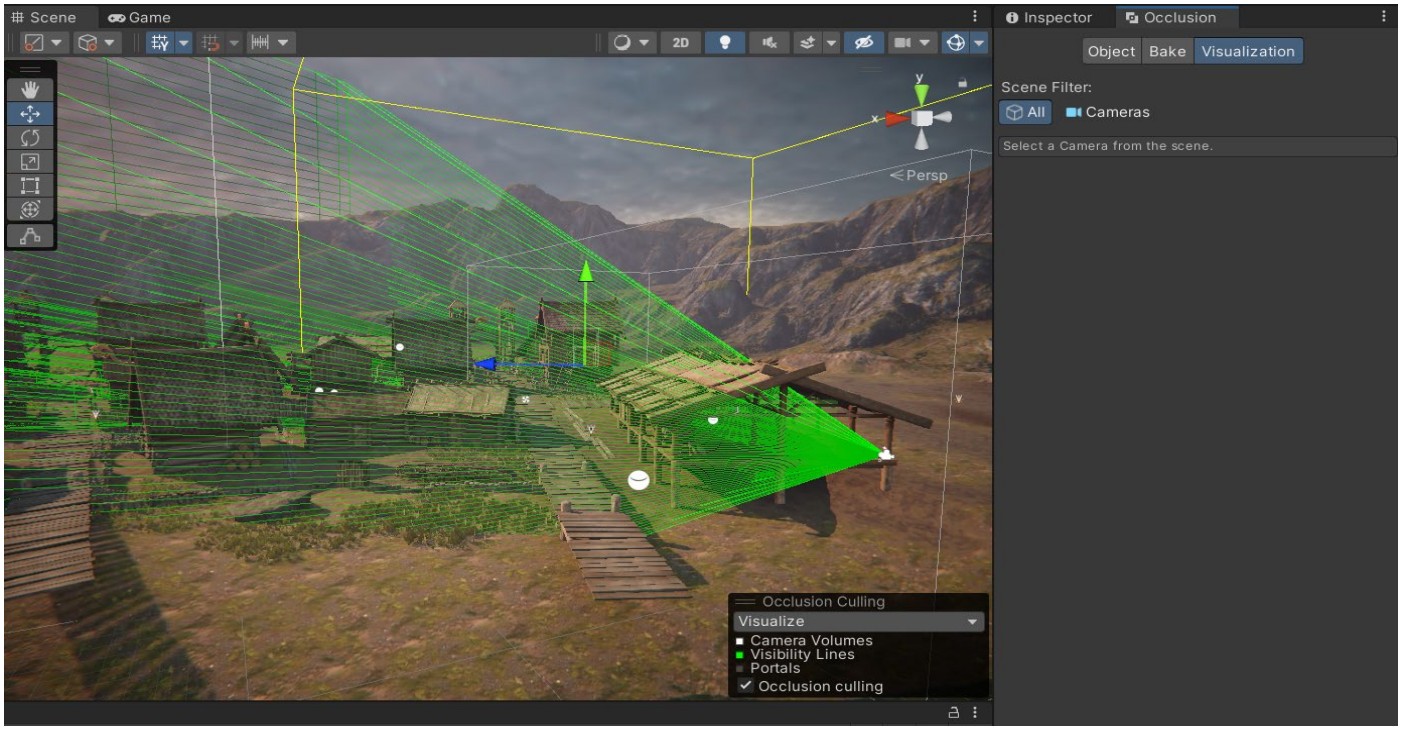
각 레이어 컬링 테스트는 기본적으로 비트 마스크 작업으로 매우 빠르게 진행됩니다. 그러나 게임 오브젝트가 매우 많으면 리소스 소모량이 늘어날 수 있습니다. 이로 인해 프로젝트에 문제가 발생하는 경우, Unity의 레이어/절두체 컬링 시스템에 대한 부담을 줄이기 위해 월드를 '구역'으로 나누고 카메라 절두체 외부에 있는 구역을 비활성화하는 시스템을 구현해야 할 수 있습니다.

오클루전 컬링은 카메라가 볼 수 없는 게임 오브젝트를 게임 뷰에서 제거합니다. 오클루전 컬링을 사용하면 다른 오브젝트에 가려진 오브젝트를 렌더링하기 위해 리소스가 소모되는 것을 방지할 수 있습니다. 예를 들어 문이 닫혀 있고 카메라가 다른 방 안쪽을 볼 수 없다면 다른 방을 렌더링할 필요가 없습니다.

오클루전 컬링을 활성화하면 성능을 크게 개선할 수 있지만, 한편으로는 디스크 공간과 CPU 시간, RAM을 더 많이 소모할 수 있습니다. Unity는 빌드 중에 오클루전 데이터를 베이크한 다음, 씬을 로드할 때 디스크에서 RAM으로 로드해야 합니다.

카메라 뷰 바깥의 절두체 컬링은 자동인 반면 오클루전 컬링은 베이크된 과정입니다. 오브젝트를 Static. Occluders 또는 Occludees로 표시한 다음 **Window > Rendering > Occlusion Culling** 다이얼로그를 통해 베이크하면 됩니다.

자세한 내용은 오클루전 컬링 사용하기 튜토리얼을 확인하세요.



오클루전 컬링의 예시

다이내믹 해상도

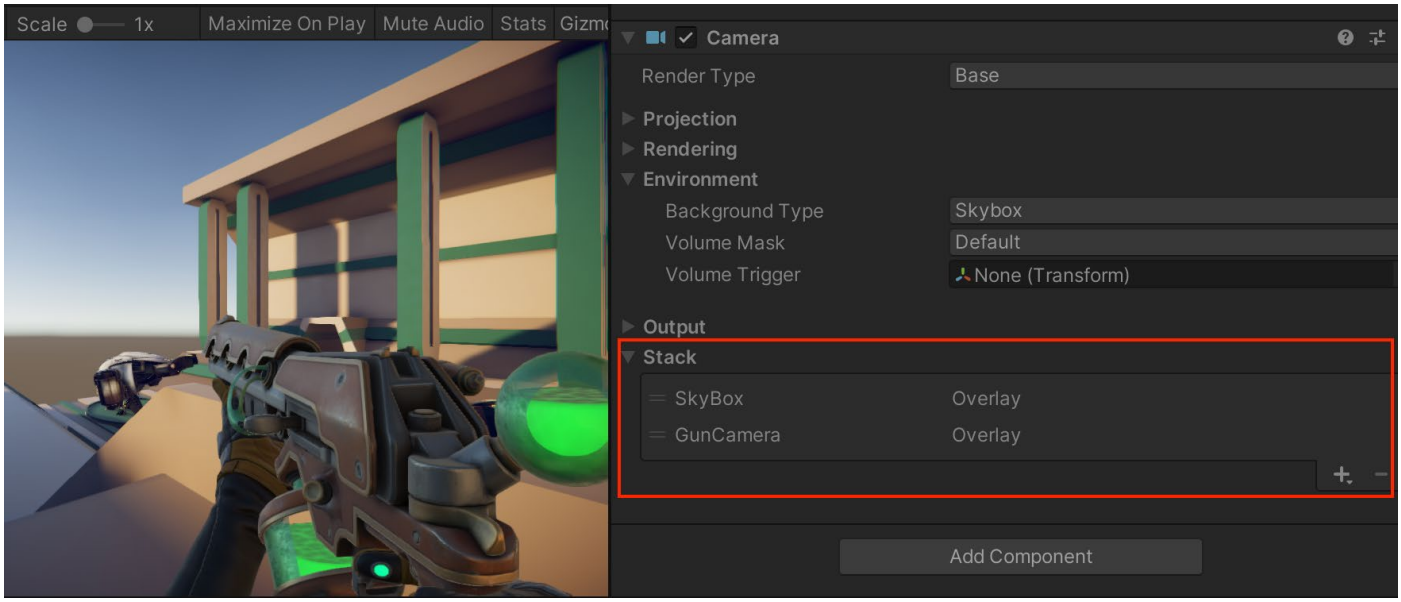
Allow Dynamic Resolution은 개별 렌더링 타겟을 동적으로 스케일링하여 GPU 워크로드를 줄일 수 있는 카메라 설정입니다. 애플리케이션의 프레임 속도가 떨어지는 경우 일관된 프레임 속도를 유지하기 위해 해상도를 점진적으로 낮출 수 있습니다.

Unity는 성능 데이터에서 GPU 바운드로 인해 프레임 속도가 떨어질 것으로 나타나면 해상도 스케일링을 트리거합니다. 스크립트를 사용하여 해상도 스케일링을 선제적으로 직접 트리거할 수도 있습니다. 수동 스케일링은 애플리케이션에서 GPU 리소스를 많이 사용하는 섹션에 접근하는 경우 유용합니다. 점진적으로 스케일링하면 다이내믹 해상도는 거의 눈에 띄지 않을 수 있습니다.

자세한 내용과 지원되는 플랫폼 목록은 다이내믹 해상도 매뉴얼 페이지를 참조하시기 바랍니다.

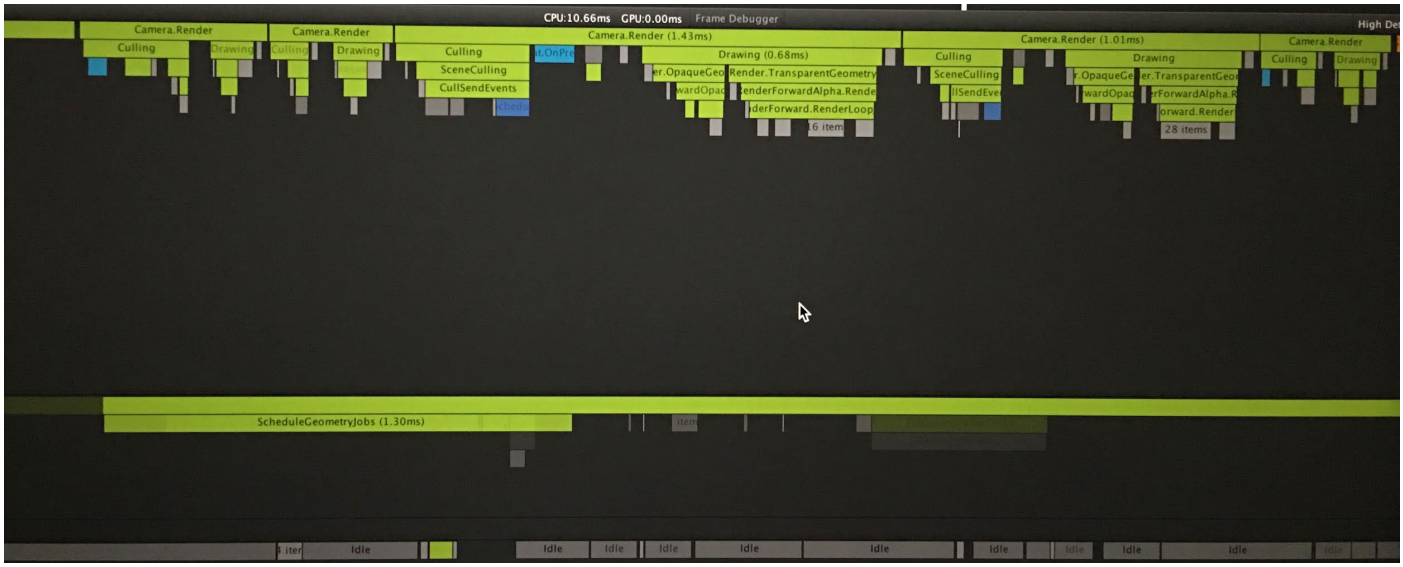
다중 카메라 뷰

때로는 게임 중에 둘 이상의 시점에서 렌더링해야 하는 경우도 있습니다. 예를 들어 FPS 게임에서는 일반적으로 플레이어의 무기와 환경을 다른 시야각(FOV)을 사용해 별도로 드로우합니다. 그러면 배경의 광각 FOV로 볼 때 전경 오브젝트가 너무 왜곡되어 보이는 문제를 방지할 수 있습니다.



총과 배경이 다른 카메라 설정으로 렌더링되는 URP의 카메라 스태킹 예시

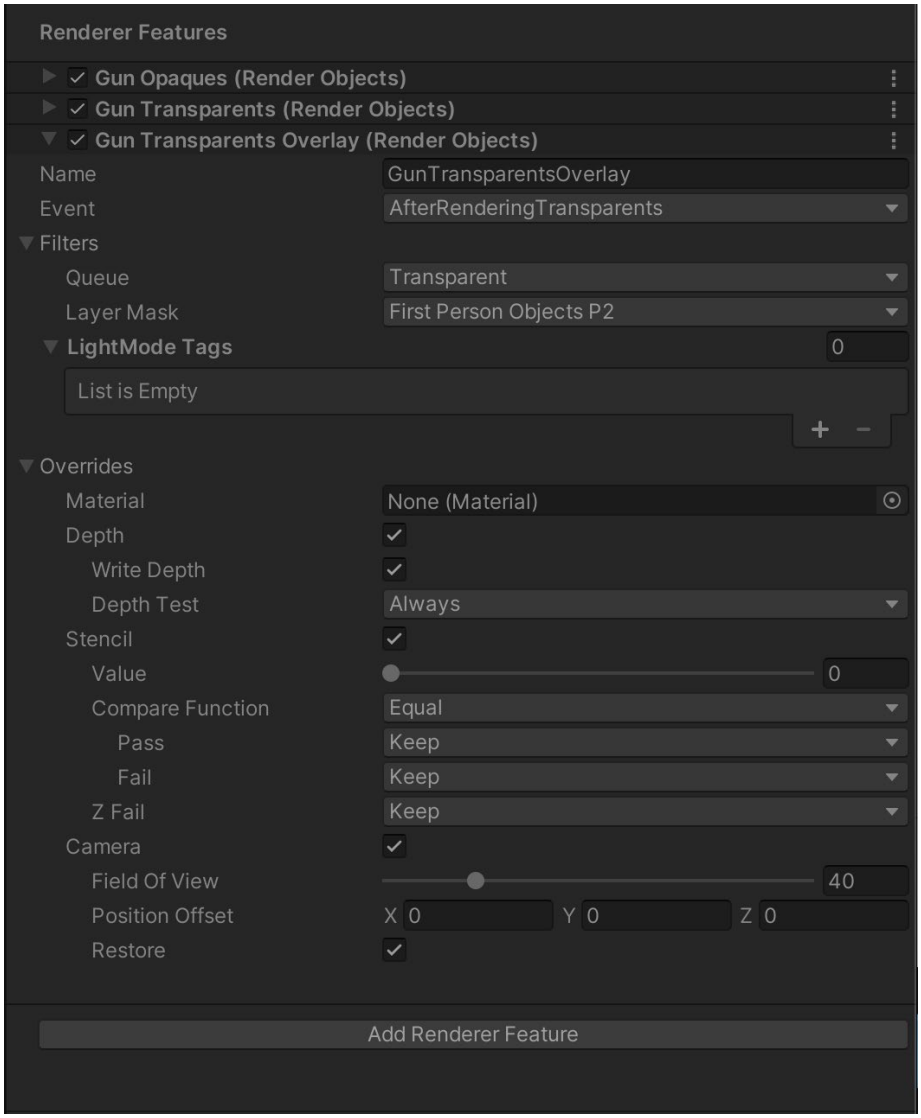
URP에서 카메라 스태킹을 사용하여 둘 이상의 카메라 뷰를 렌더링할 수 있습니다. 그러나 이 경우 카메라별로 상당한 컬링과 렌더링이 수행됩니다. 모든 카메라는 작업 수행 시 어느 정도의 오버헤드를 유발하기 때문에, 렌더링에 필요한 카메라 컴포넌트만 사용하는 것이 좋습니다. 모바일 플랫폼에서는 각 활성 카메라가 최대 1ms의 CPU 시간을 사용할 수 있으며, 렌더링할 대상이 없을 때도 마찬가지입니다.



Unity CPU 프로파일러는 타임라인 뷰에 메인 스레드를 표시하고 여러 카메라가 있음을 보여 줍니다. Unity는 카메라별로 컬링을 수행합니다.

URP의 Render Objects 렌더러 기능

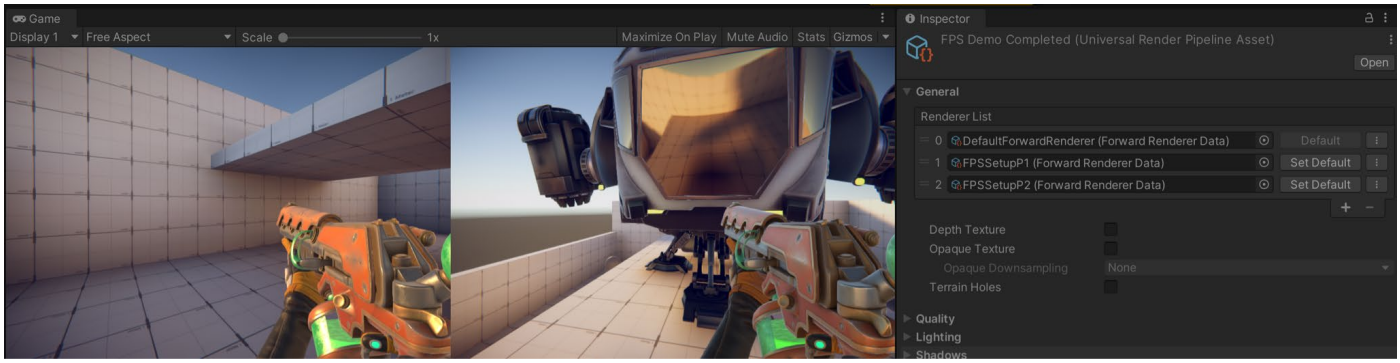
URP의 경우, 여러 카메라를 사용하는 대신 커스텀 Render Objects 렌더러 기능을 사용할 수 있습니다. **Renderer Data** 에셋에서 **Add Renderer Feature**를 선택하고 **Render Objects**를 선택합니다.



커스텀 Render Object를 생성하여 렌더 설정을 오버라이드하세요.

각 Render Object를 오버라이드하면 다음을 수행할 수 있습니다.

- Render Object를 이벤트와 연결하고 렌더 루프의 특정 타이밍에 주입
- 렌더링 대기열(투명 또는 불투명)과 LayerMask로 필터링
- 뎁스 설정과 스텐실 설정에 영향
- 카메라 설정(시야각(FOV) 및 포지션 오프셋) 수정



여러 레이어를 하나의 렌더링된 뷰로 결합하는 URP의 Render Objects 렌더러 기능

HDRP의 커스텀 패스 볼륨

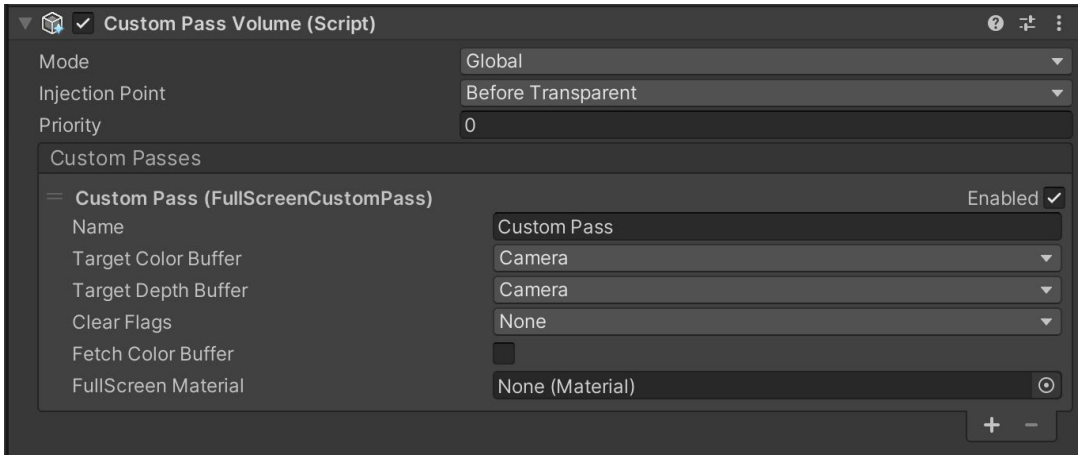
HDRP의 경우, 비슷한 효과를 위해 커스텀 패스를 사용할 수 있습니다. Custom Pass Volume을 사용해 커스텀 패스를 설정하는 것은 HDRP Volume을 사용하는 것과 비슷합니다.

커스텀 패스를 사용하면 다음을 수행할 수 있습니다.

— 씬의 머티리얼 형상 변경

— Unity가 게임 오브젝트를 렌더링하는 순서 변경

— 카메라 버퍼를 셰이더로 읽어 오기

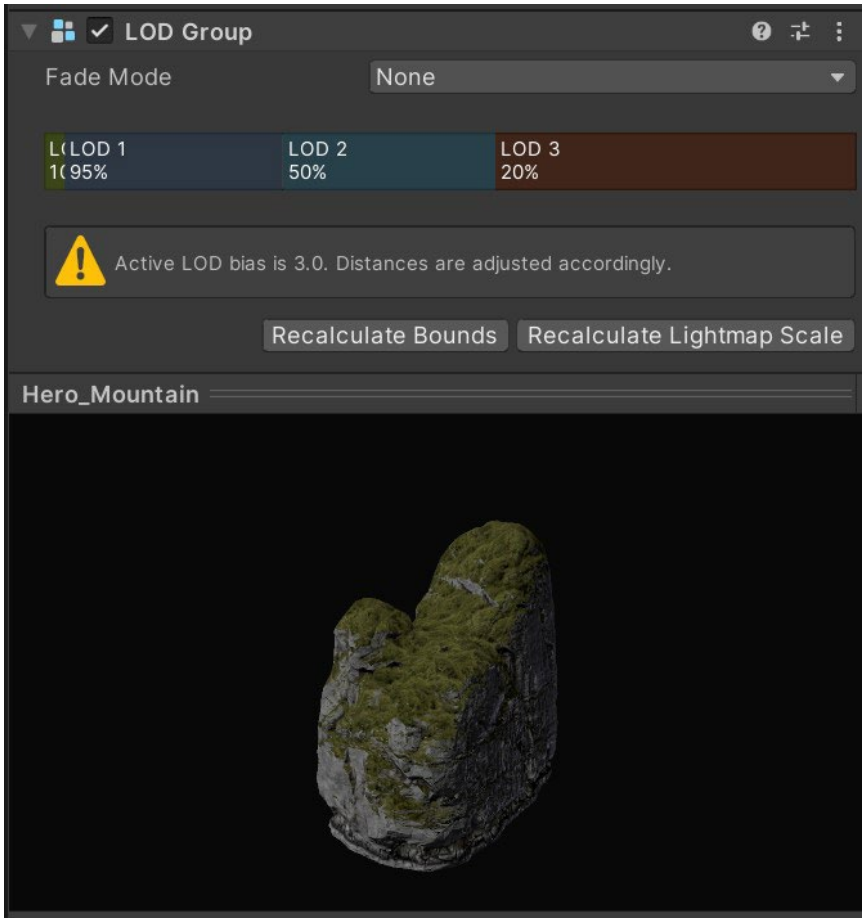


HDRP의 Custom Pass Volume

HDRP의 Custom Pass Volume을 사용하면 추가 카메라와 이로 인한 오버헤드를 방지할 수 있습니다. 커스텀 패스는 셰이더와 상호 작용할 때 더 높은 유연성을 보이며, C#으로 Custom Pass 클래스를 확장할 수도 있습니다.

디테일 수준(LOD) 사용

오브젝트의 거리가 멀어짐에 따라 디테일 수준(LOD)을 통해 머티리얼과 셰이더가 더 단순한 저해상도 메시를 사용하도록 조정하거나 전환하여 GPU 성능을 보조할 수 있습니다.



LOD 그룹을 사용하는 메시의 예시

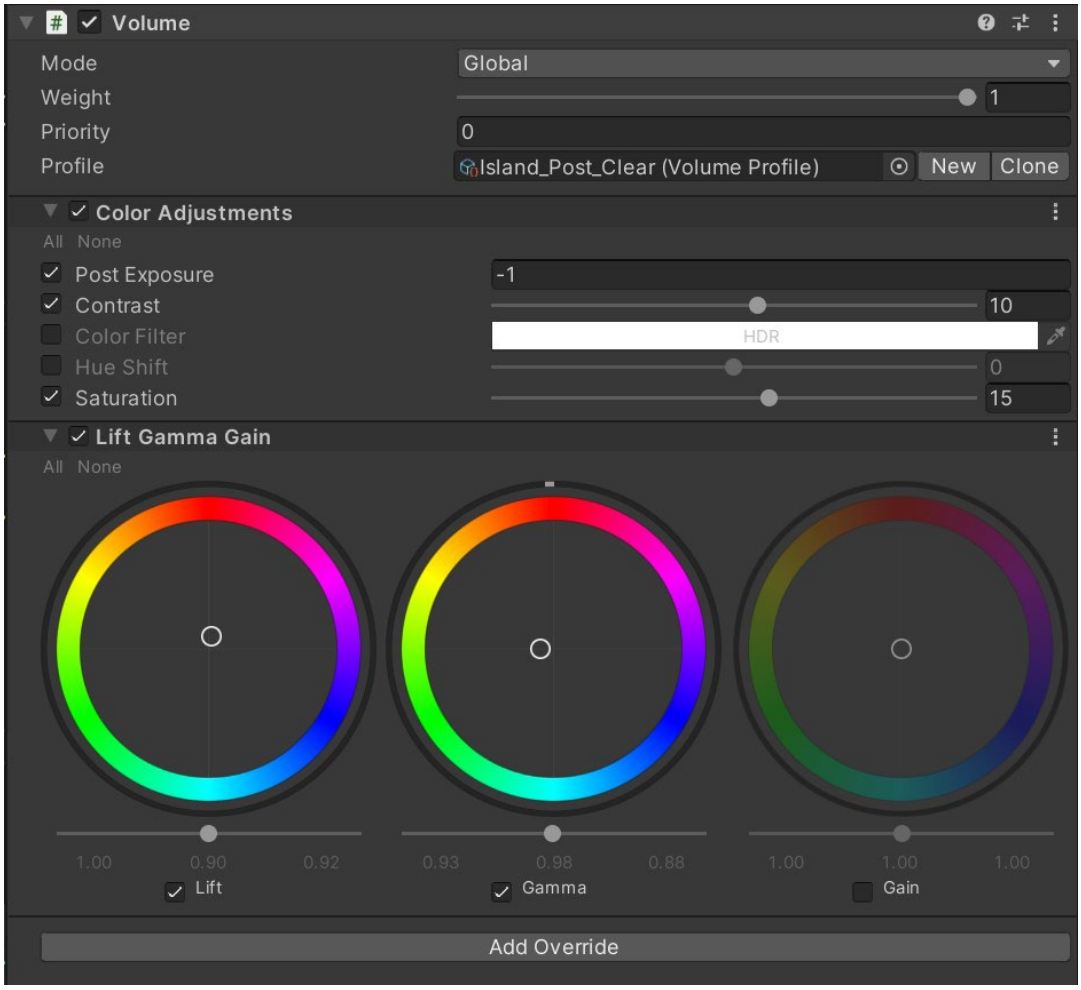
다양한 해상도로 모델링한 소스 메시

Unity Learn 온라인 교육 LOD 사용하기에서 자세한 내용을 확인하세요.

포스트 프로세싱 효과 프로파일링

포스트 프로세싱 효과가 GPU에서 사용하는 리소스량을 확인하려면 포스트 프로세싱 효과를 프로파일링하세요. 블룸과 뎁스오브필드(피사계심도) 등의 일부 전체 화면 효과는 리소스를 많이 소모할 수 있지만, 화질과 성능 사이에서 만족스러운 절충점을 찾을 때까지 실험해 봐야 합니다.

포스트 프로세싱은 런타임 시 크게 변하지 않는 편입니다. 볼륨 오버라이드를 정했으면 전체 프레임 예산의 정적 부분을 포스트 효과에 할당하세요.



포스트 프로세싱 효과는 최대한 단순하게 유지해야 합니다.

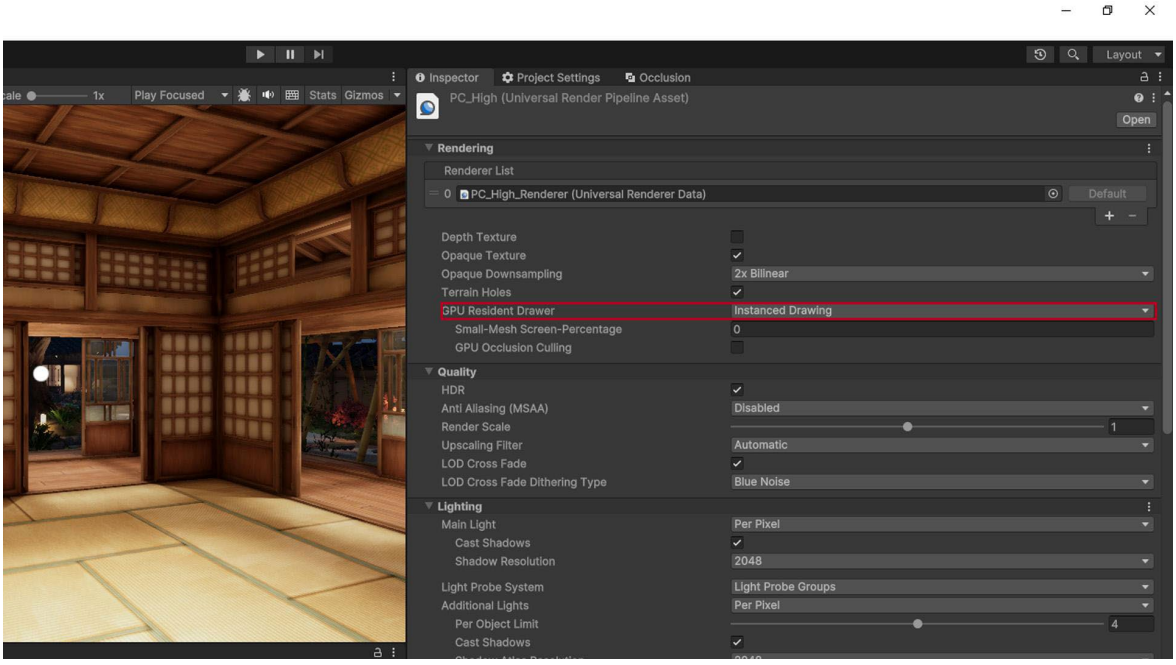
GPU 상주 드로어

URP와 HDRP에서 사용할 수 있는 GPU 상주 드로어는 CPU 시간을 최적화하여 성능을 크게 향상하도록 설계된 GPU 기반 렌더링 시스템입니다. 크로스 플랫폼 렌더링을 지원하며 기존 프로젝트에서 곧바로 사용할 수 있도록 설계되었습니다.

GPU 상주 드로어를 사용하여 실행되는 URP 3D 샘플의 정원 환경

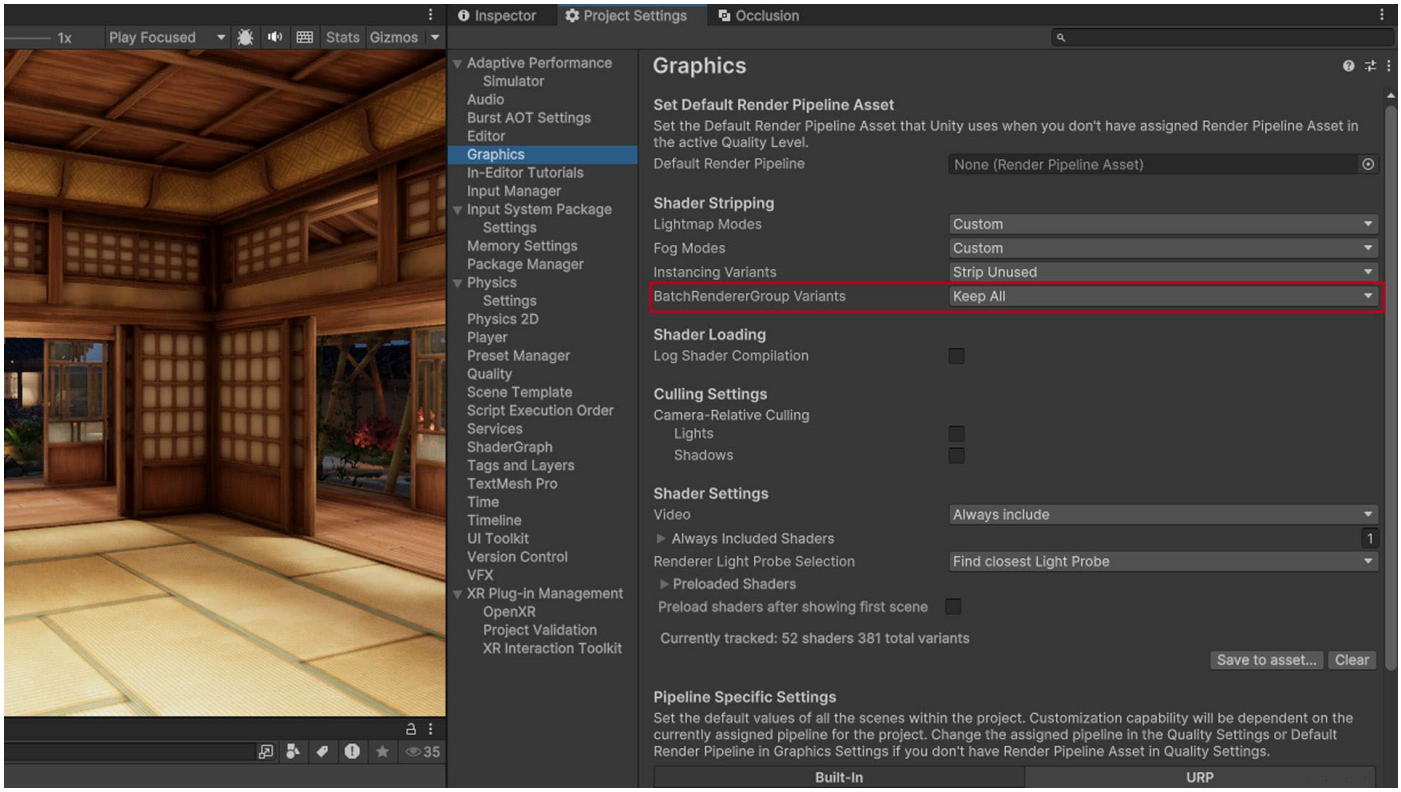
GPU 상주 드로어는 HDRP 또는 URP 렌더 파이프라인 에셋에서 활성화할 수 있습니다. 활성화하려면 **Instanced Drawing**을 선택합니다. 플레이 모드에서만 활성화할지 에디터 모드에서도 활성화할지 선택할 수 있습니다.

GPU 상주 드로어를 활성화하면 드로우 콜이 많은 CPU 바운드 게임에서 드로우 콜의 양이 줄어 성능이 향상될 수 있습니다. 개선되는 정도는 씬의 규모와 사용되는 인스턴싱 양에 따라 달라집니다. 인스턴싱할 수 있는 오브젝트를 더 많이 렌더링할수록 더 많이 개선됩니다.



GPU Resident Drawer 드롭다운에서 'Instanced Drawing' 선택

Instanced Drawing 옵션을 선택하면 UI에서 “BatchRenderGroup Variants 설정을 ‘Keep All’로 설정해야 합니다”라는 경고 메시지가 표시될 수 있습니다. 그래픽스 설정에서 이 옵션을 조정하면 GPU 상주 드로어 설정이 완료됩니다.



BatchRenderGroup Varients를 **Keep All로 설정**

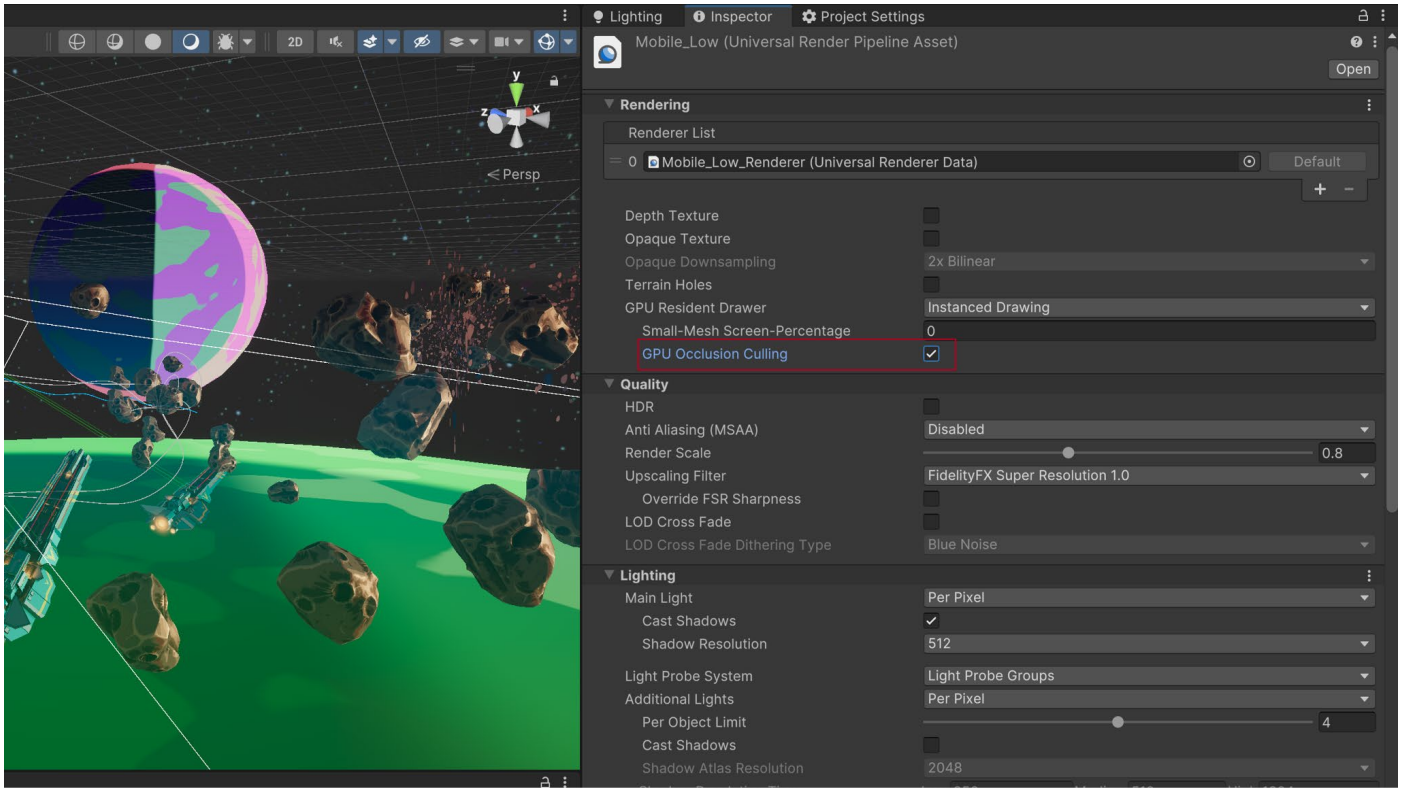
자세히 알아보려면 이 토론 게시물을 살펴보세요.

GPU 오클루전 컬링

URP와 HDRP에서 사용할 수 있는 GPU 오클루전 컬링은 GPU 상주 드로어와 함께 작동합니다. 프레임마다 오버드로우 수를 줄여 렌더러가 보이지 않는 오브젝트를 드로우하기 위해 리소스를 낭비하지 않으므로 성능이 크게 향상됩니다.

GPU 오클루전 컬링을 활성화하려면 렌더 파이프라인 에셋에서 GPU Occlusion 체크박스를 활성화하세요.

Unity 6의 디버그 모드에서 GPU 오클루전 컬링을 활성화하려면 **Window > Rendering > Occlusion Culling**으로 이동합니다. 여기에서 다양한 시각화 옵션을 활성화하여 오브젝트가 어떻게 컬링되는지 확인할 수 있습니다.



렌더 파이프라인 에셋의 GPU Occlusion Culling 옵션

Split Graphics Jobs

Split Graphics Jobs를 사용하면 여러 CPU 코어에서 렌더링 커맨드를 더 효율적으로 실행하므로 렌더링 작업의 병렬성(parallelism)을 향상하여 성능을 개선할 수 있습니다.

여러 데스크톱 및 콘솔 플랫폼에서 지원하는 스레딩 모드가 제공됩니다. 주된 개선 사항은 일반적인 게임 로직과 오케스트레이션을 담당하는 메인 스레드와 렌더링 작업을 담당하는 네이티브 그래픽스 잡 스레드 간의 불필요한 동기화를 줄이는 것으로 이루어집니다.

이 새로운 스레딩 모드는 프레임마다 제출되는 드로우 콜이 늘어날수록 성능을 크게 향상합니다. 드로우 콜이 많을수록 최적화의 혜택을 많이 받으므로 오브젝트와 텍스처가 많은 복잡한 씬은 성능이 크게 향상됩니다.

Splits Graphics Jobs는 DX12를 사용하는 Windows 또는 Vulkan 플레이어에서 지원됩니다.

Project Settings > Player > Other Settings > Graphics Jobs Mode > Split에서 활성화할 수 있습니다.

이전부터 사용할 수 있었던 레거시 및 네이티브 모드 대신 항상 Splits Graphics Jobs를 사용할 것을 권장합니다.

사용자 인터페이스

사용자 인터페이스

Unity는 두 가지 UI 시스템을 제공하는데, 바로 Unity UI와 새로운 UI 툴킷입니다. 권장되는 UI 시스템은 UI 툴킷입니다. UI 툴킷은 표준 웹 기술에서 영감을 받은 워크플로와 저작(authoring) 툴을 통해 최고의 성능과 재사용성을 발휘하도록 설계되었습니다. 따라서 웹 페이지 디자인 경험이 있는 UI 디자이너와 아티스트는 익숙하게 사용할 수 있습니다.

그러나 Unity 6에서는 Unity UI 및 IMGUI(즉시 모드 GUI)가 지원하는 일부 기능이 UI 툴킷에 포함되어 있지 않습니다. Unity UI 및 IMGUI는 특정 사용 사례에 더 적합하며 레거시 프로젝트를 지원하는 데 필요합니다. 자세한 내용은 Unity의 UI 시스템 비교를 참조하세요.

UGUI 성능 최적화 팁

UGUI(Unity UI)는 성능 문제의 원인이 되는 경우가 많습니다. Canvas 컴포넌트는 UI 요소에 대한 메시를 생성 및 업데이트하고 GPU에 드로우 콜을 보냅니다. 이러한 기능은 리소스를 많이 소모할 수 있으므로 UGUI를 사용할 때 다음 사항을 기억하세요.

Canvas 나누기

수천 개의 요소가 포함된 하나의 대형 Canvas에서는 UI 요소를 하나만 업데이트해도 전체 Canvas가 강제로 업데이트되므로 CPU 스파이크가 발생할 수 있습니다.

다수의 Canvas를 지원하는 UGUI의 기능을 활용하세요. 새로 고침해야 하는 빈도에 따라 UI 요소를 나눕니다. 정적 UI 요소는 별도의 캔버스에 두고, 동시에 업데이트되는 동적 요소는 더 작은 하위 캔버스에 두는 것이 좋습니다.

각 Canvas 내의 모든 UI 요소가 동일한 Z 값, 머티리얼, 텍스처를 갖도록 해야 합니다.

보이지 않는 UI 요소 숨기기

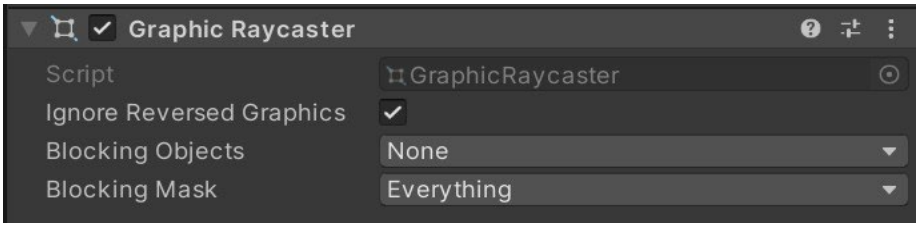
게임에서 간헐적으로만 나타나는 UI 요소(예: 캐릭터가 대미지를 입었을 때 나타나는 체력 표시줄)가 있을 수 있습니다. 보이지 않는 UI 요소가 활성화된 경우에도 드로우 콜을 사용할 수 있습니다. 보이지 않는 UI 컴포넌트를 명시적으로 비활성화하고 필요할 때 다시 활성화하세요.

캔버스만 보이지 않게 만들면 되는 경우, 모든 게임 오브젝트 대신 Canvas 컴포넌트를 비활성화하세요. 이렇게 하면 다시 활성화했을 때 게임이 메시와 버텍스를 재구성하지 않아도 됩니다.

GraphicRaycaster를 제한하고 Raycast Target 비활성화하기

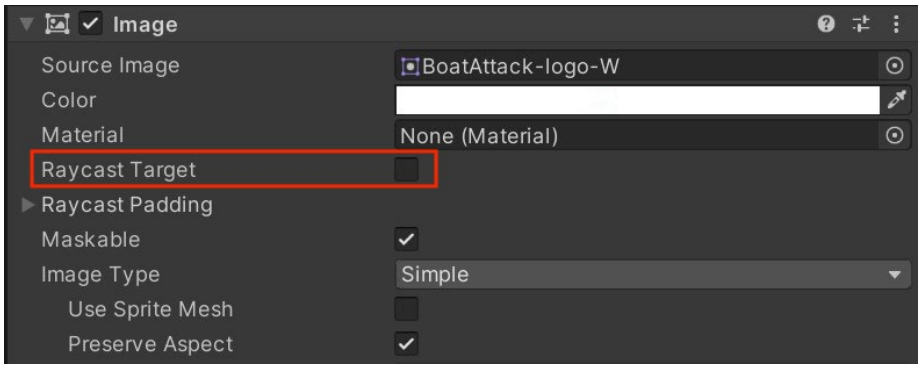
GraphicRaycaster 컴포넌트는 화면 터치나 클릭과 같은 입력 이벤트에 필요합니다. 이 컴포넌트는 화면의 각 입력 지점을 순환하며 입력 지점이 UI의 RectTransform 내에 있는지 확인합니다. Graphic Raycaster는 하위 Canvas를 비롯해 입력이 필요한 모든 Canvas에 사용해야 합니다.

이름과 달리 실제 레이캐스터는 아니지만, 각 교차 지점 검사에 약간의 리소스를 소모합니다. 상호 작용할 수 없는 UI Canvas에는 Graphic Raycaster를 추가하지 않음으로써 그 수를 최소화해야 합니다.



상호 작용할 수 없는 UI Canvas에서 GraphicRaycaster 제거

또한 모든 UI 텍스트 및 이미지에서 불필요하게 활성화된 Raycast Target도 비활성화합니다. UI에 많은 요소가 있어 복잡한 경우 이와 같이 간단히 설정을 변경하여 불필요한 계산을 줄일 수 있습니다.

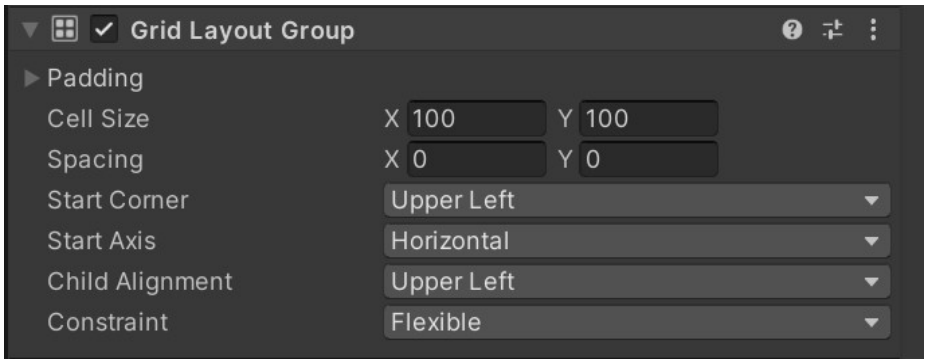


가능한 경우 Raycast Target 비활성화

레이아웃 그룹 사용 지양

레이아웃 그룹은 비효율적으로 업데이트되므로 필요할 때만 사용하세요. 콘텐츠가 동적이지 않은 경우 Layout Group을 아예 사용하지 말고, 비율 레이아웃에 앵커를 대신 사용하시기 바랍니다. 그 밖의 경우에는 UI 설정을 마친 후 커스텀 코드를 생성하여 레이아웃 그룹 컴포넌트를 비활성화합니다.

동적 요소에 Layout Group(Horizontal, Vertical, Grid)을 사용해야 하는 경우, 중첩되지 않도록 하여 성능을 개선합니다.



중첩된 경우 특히 성능을 낮추는 레이아웃 그룹

대형 리스트 뷰와 그리드 뷰 사용 시 주의

대형 리스트 뷰 및 그리드 뷰는 비용이 많이 듭니다. 대형 리스트 뷰나 그리드 뷰(예: 수백 개의 아이템이 표시되는 인벤토리 화면)를 생성해야 하는 경우, 아이템마다 UI 요소를 생성하는 대신 소형 UI 요소 풀을 재사용하는 방법을 고려해 보세요. 샘플 GitHub 프로젝트를 통해 실제 작동 모습을 확인하세요.

요소의 과도한 레이어링 주의

UI 요소를 많이 레이어링(예: 카드 배틀 게임에서 겹쳐져 있는 카드)하면 오버드로우가 발생합니다. 코드를 커스터마이즈하여 런타임에 레이어링된 요소를 병합하는 방법으로 요소나 배치의 수를 줄이세요.

전체 화면 UI 사용 시 기타 요소 모두 숨기기

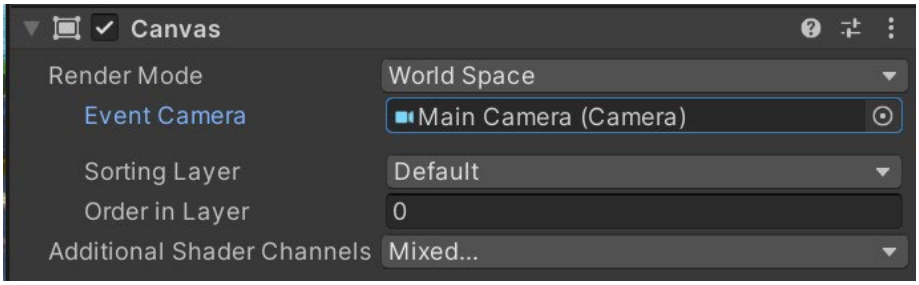
일시 중지 화면이나 시작 화면이 씬의 나머지 부분을 모두 가리는 경우, 3D 씬을 렌더링하는 카메라를 비활성화합니다. 마찬가지로 맨 위쪽 Canvas에 가려진 배경 Canvas 요소도 모두 비활성화합니다.

60fps에서는 업데이트할 필요가 없으므로 전체 화면 UI 사용 시에는 Application.targetFrameRate를 낮게 설정하는 것이 좋습니다.

World Space 및 Camera Space Canvas 카메라 설정

Event 또는 **Render Camera** 필드를 공백으로 두면 Unity가 자동으로 **Camera.main**을 채워 넣어 불필요한 리소스가 소모됩니다.

가능한 한 Canvas의 **RenderMode**에 카메라가 필요 없는 **Screen Space – Overlay**를 사용하는 것이 좋습니다.



World Space Render Mode 사용 시 Event Camera 설정 확인

UI 툴킷 성능 최적화 팁

게임에 UI가 많다면 일반적으로 Unity 6에서 UI 툴킷을 사용할 것을 권장합니다.

UI 툴킷은 Unity UI보다 향상된 성능을 제공하며, 표준 웹 기술에서 영감을 받은 워크플로와 저작 툴을 갖춰 최고의 성능과 재사용성을 제공할 수 있도록 설계되었습니다. 주요 이점 중 하나는 UI 요소를 위해 특별히 설계된 최적화 렌더 파이프라인을 사용한다는 점입니다.

UI 툴킷을 사용하여 UI의 성능을 최적화하기 위한 일반적인 권장 사항은 다음과 같습니다.

효율적인 레이아웃 사용

효율적인 레이아웃을 구현하려면 직접 UI 요소의 위치를 지정하고 크기를 조절하기보다 UI 툴킷에서 제공하는 Flexbox 같은 레이아웃 그룹을 사용해야 합니다. 레이아웃 그룹을 사용하면 레이아웃 계산이 자동으로 수행되므로 성능이 크게 향상됩니다. 지정된 레이아웃 규칙에 따라 UI 요소가 올바르게 정렬되고 크기가 조절되기도 합니다. 효율적인 레이아웃을 사용하면 직접 레이아웃을 계산하는 오버헤드를 방지하면서 일관되고 최적화된 UI 렌더링을 수행할 수 있습니다.

Update 메서드에서 비용이 많이 드는 작업 지양

Update 메서드에서 수행되는 작업 중에서도 특히 비용이 많이 드는 UI 요소 생성, 조작, 계산 같은 작업을 최소화하세요. Update 메서드는 프레임마다 호출되므로 가능하면 이러한 작업은 최소한으로 또는 초기화 때만 수행하는 것이 좋습니다.

이벤트 처리 최적화

이벤트 구독을 자주 확인하고 더 이상 필요 없는 경우에는 구독을 취소하세요. 이벤트를 지나치게 많이 처리하면 성능이 저하될 수 있으므로 꼭 필요한 이벤트만 등록해야 합니다.

스타일 시트 최적화

스타일 시트에 사용되는 스타일 클래스 및 선택자의 개수에 유의하세요. 규칙이 많은 대형 스타일 시트를 사용하면 성능이 저하될 수 있습니다. 스타일 시트는 간결하게 유지하고, 불필요하게 복잡해지지 않게 해야 합니다.

프로파일링 및 최적화

Unity의 프로파일링 툴을 사용하여 비효율적인 레이아웃 계산이나 과도한 리드로우 같은 UI의 성능 병목 지점을 식별하고 더 최적화할 수 있는 영역을 파악하세요.

타겟 플랫폼에서 테스트

다양한 기기에서 최적의 성능을 구현할 수 있도록 타겟 플랫폼에서 UI 성능을 테스트하세요. 하드웨어 사양에 따라 성능이 달라질 수 있으므로 UI를 최적화할 때 타겟 플랫폼을 고려해야 합니다.

성능 최적화는 반복적인 프로세스라는 점을 기억하세요. UI 코드를 지속적으로 프로파일링하고, 측정하고, 최적화하여 원활하고 효율적으로 실행될 수 있도록 만들어야 합니다.

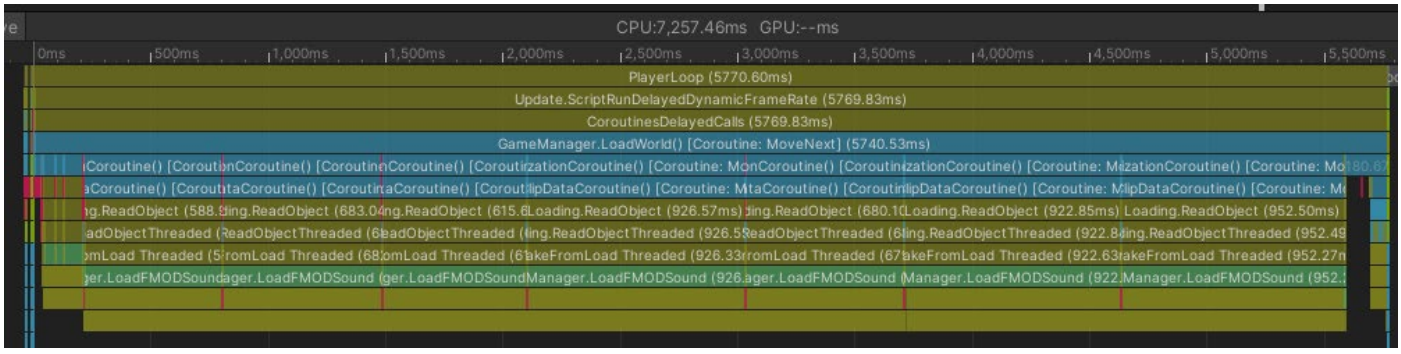
오디오

오디오는 보통 성능 저하의 원인이 되지 않지만, 오디오를 최적화하여 메모리, 디스크 공간, CPU 사용량을 절약할 수 있습니다.

무손실 파일을 소스로 사용

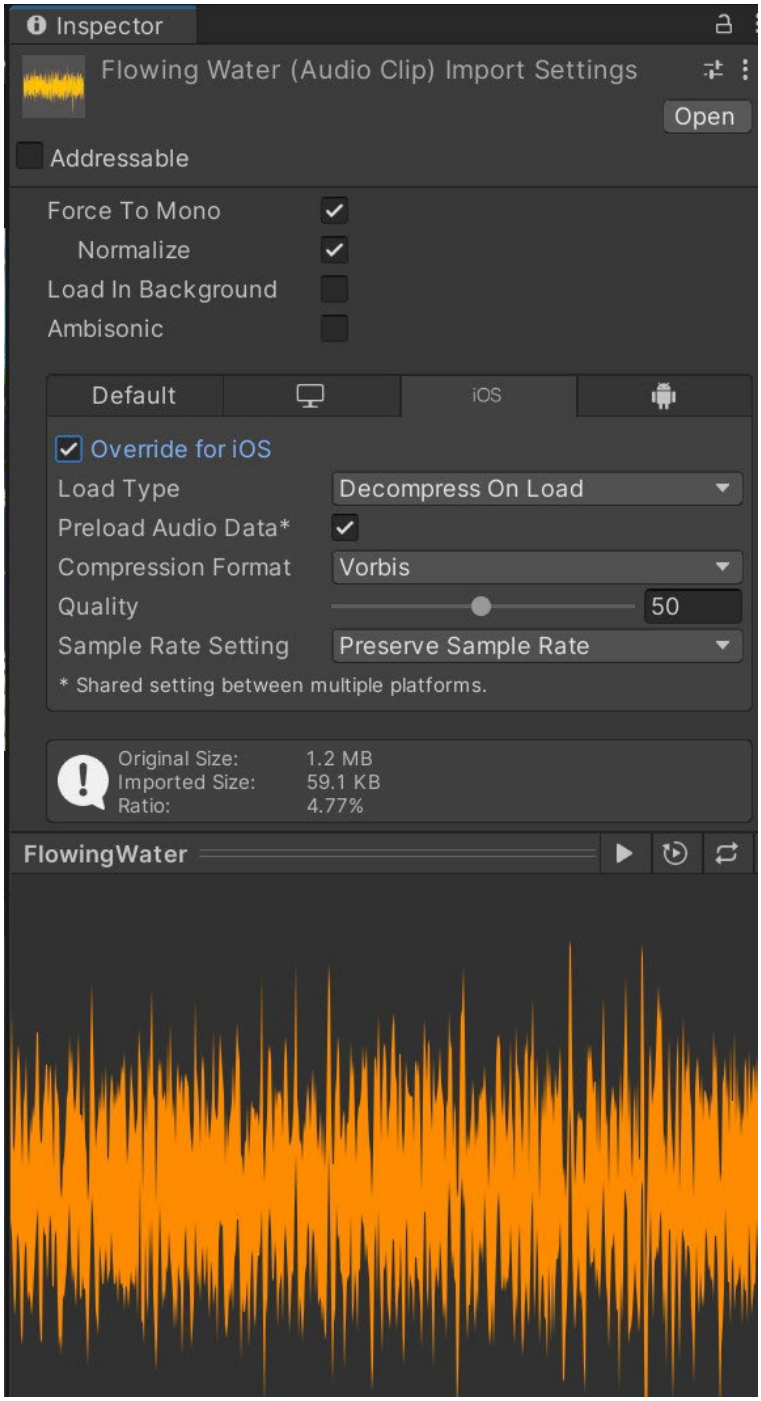
먼저 WAV 또는 AIFF 등의 무손실 파일 포맷을 사운드 에셋으로 사용하세요.

압축된 포맷(예: MP3 또는 Vorbis)을 사용하는 경우 Unity에서 빌드 시간 동안 압축을 풀고 재압축하며, 이로 인해 손실이 있는 패스가 두 번 발생하기 때문에 최종 품질이 저하됩니다.



Unity 프로파일러의 오디오 로딩 CPU 사용량

AudioClip 줄이기



AudioClip 임포트 설정

AudioClip의 임포트 설정을 활용하면 런타임 메모리와 CPU 성능 비용을 절약할 수 있습니다.

— 스테레오 사운드가 필요하지 않은 경우, 스테레오 오디오 파일에서 **Force To Mono** 옵션을 사용해 런타임 메모리와 디스크 공간을 절약할 수 있습니다.

  입체 음향 소스는 모노로 제작된 AudioClip을 사용하거나 임포트 설정에서 Force To Mono 옵션을 활성화해야 합니다. 입체 음향 소스에 스테레오 사운드를 사용하면 오디오 데이터는 디스크 공간과 메모리를 두 배로 차지합니다. 오디오 믹싱 과정 중에 Unity가 사운드를 모노로 변환해야 하는데, 이때 CPU 처리 시간이 추가로 필요하기 때문입니다.

— **Preload Audio Data** 옵션을 사용하면 Unity가 씬을 초기화하기 전에 참조된 AudioClip을 로드할 수 있습니다. 그러나 이 경우 씬 로딩 시간이 증가할 수 있습니다.

— 바로 사용해야 하는 사운드 클립이 아니라면 비동기로 로드하세요. **Load in Background** 옵션을 선택하면 됩니다. 그러면 메인 스레드를 막지 않고 별도의 스레드에서 지연된 시간에 사운드를 로드할 수 있습니다.

— **Sample Rate Setting**을 Optimize Sample Rate 또는 Override Sample Rate로 설정하세요.

  모바일 플랫폼의 경우 22,050Hz면 충분합니다. 필요할 경우에만 CD 품질인 44,100Hz를 사용하고, 48,000Hz는 지양하세요.

  PC/콘솔 플랫폼에서는 44,100Hz가 이상적입니다. 일반적으로 48,000Hz는 필요하지 않습니다.

— AudioClip을 압축하고 압축 비트레이트를 낮추세요.

  모바일 플랫폼에서는 대부분의 사운드에 Vorbis를 사용하고, 반복 재생하지 않는 사운드에는 MP3를 사용합니다. 자주 사용하는 짧은 소리(예: 발자국 소리, 총소리)에는 ADPCM을 사용하세요.

  PC나 Xbox®의 경우, Vorbis나 MP3 대신 Microsoft XMA 코덱을 사용하세요. Microsoft는 8:1에서 15:1 사이의 압축비를 권장합니다.

  Playstation의 경우, ATRAC9 포맷을 사용하세요. 그러면 Vorbis나 MP3보다 CPU 오버헤드가 줄어듭니다.

— 클립의 길이에 따라 적절한 로드 유형이 달라집니다.

클립 크기	사용 예시	로드 유형 설정
작음 (< 200KB)	시끄러운 음향 효과 (발자국 소리, 총소리), UI 사운드	**Decompress on Load**를 사용합니다. 사운드를 16비트 PCM 오디오 원시 데이터로 압축 해제하느라 약간의 CPU 리소스를 사용하지만, 런타임에서 성능이 매우 뛰어납니다. 또는 **Compressed In Memory**로 설정하고 **Compression Format**을 **ADPCM**으로 설정합니다. 이 경우 고정된 3.5:1 압축비를 제공하며 실시간으로 압축 해제 시 리소스 소모가 적습니다.

중간 (>= 200KB)	대화 소리, 짧은 음악, 적당하거나 시끄럽지 않은 음향 효과	최적의 로드 유형은 프로젝트의 우선순위에 따라 다릅니다. 무엇보다 메모리 사용량을 줄여야 한다면 **Compressed In Memory**를 선택하세요. CPU 사용량이 걱정이라면 클립을 **Decompress On Load**로 설정하세요.
큼 (> 350~400KB)	배경 음악, 주변 배경 소음, 긴 대화	**Streaming**으로 설정하세요. 스트리밍 오버헤드는 200KB이므로 매우 큰 AudioClip에만 적합합니다.

AudioMixer 최적화

AudioClip 설정과 더불어 AudioMixer 설정도 유의해야 합니다.

— SFX 리버브 효과는 AudioMixer에서 가장 리소스를 많이 소모하는 오디오 효과에 속합니다. SFX 리버브를 사용하는 믹서 그룹과 해당 믹서 그룹으로 보내는 믹서 그룹을 추가하면 CPU 비용이 증가합니다.

실제로 그룹에 신호를 보내는 AudioSource가 없는 경우에도 CPU 비용이 증가합니다. Unity의 오디오 시스템은 null 신호의 수신 여부를 구분하지 않습니다.

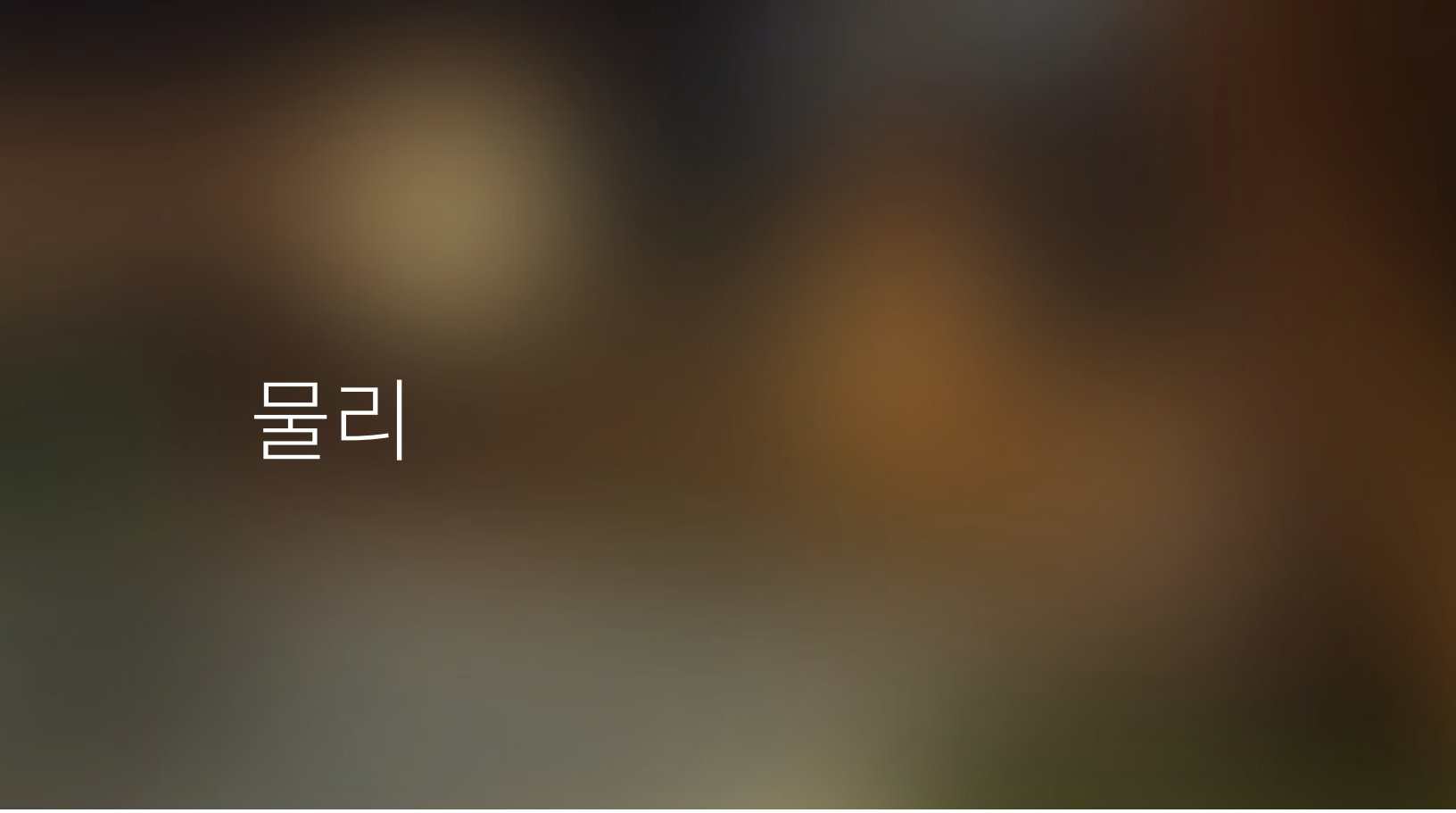
리버브 그룹과 해당 그룹으로 보내는 그룹을 추가하면 그룹에 작성하는 AudioSource가 없더라도 리소스를 많이 소모합니다.

— AudioMixer 성능을 높이려면 믹서 그룹의 수를 줄이세요. 부모 그룹 하나에 많은 자식 그룹을 추가하면 오디오 CPU 비용이 크게 증가합니다. Unity DSP는 null 신호를 구분하지 않기 때문에 모든 AudioSource가 Master로 바로 출력되는 경우에도 CPU 비용이 증가합니다.

지나치게 많은 자식 그룹을 가진 AudioMixer 그룹

— 자식 그룹이 하나밖에 없는 부모 그룹은 사용하지 않는 것이 좋습니다. 가능하면 두 믹서 그룹을 하나로 합치세요.

하나의 자식 그룹을 가진 AudioMixer

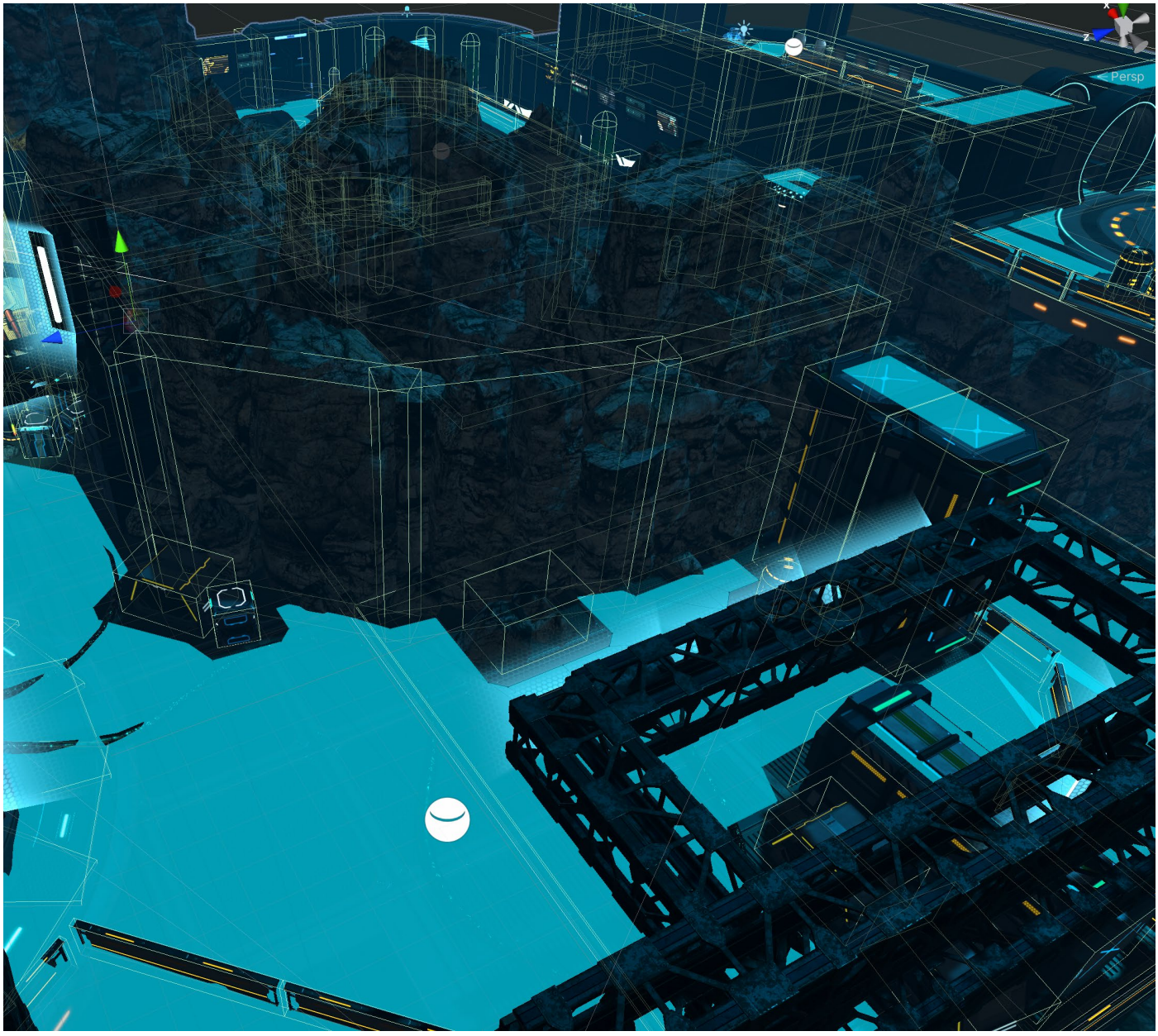


물리

물리를 활용하면 복잡한 게임플레이를 만들 수 있지만, 성능 비용을 소모하게 됩니다. 성능 비용을 정확하게 파악하면 시뮬레이션을 미세 조정하여 비용을 적절하게 관리할 수 있습니다. 아래 팁을 활용하여 목표 프레임 속도를 유지하고 NVIDIA PhysX 엔진이 통합된 Unity의 빌트인 3D 물리로 매끄러운 플레이를 만들어 보세요.

콜라이더 단순화

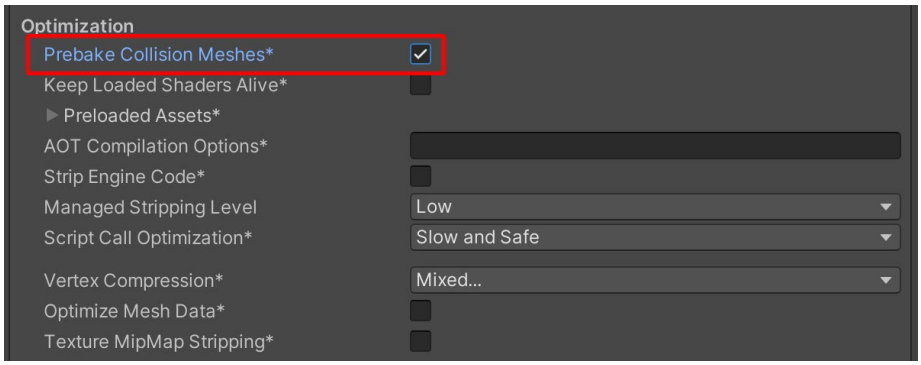
메시 콜라이더는 많은 리소스를 요합니다. 복잡한 메시 콜라이더를 기본 또는 단순화된 메시 콜라이더로 대체하여 원래 모양을 대략적으로 표현하세요.



콜라이더에 기본 또는 단순화된 메시 사용

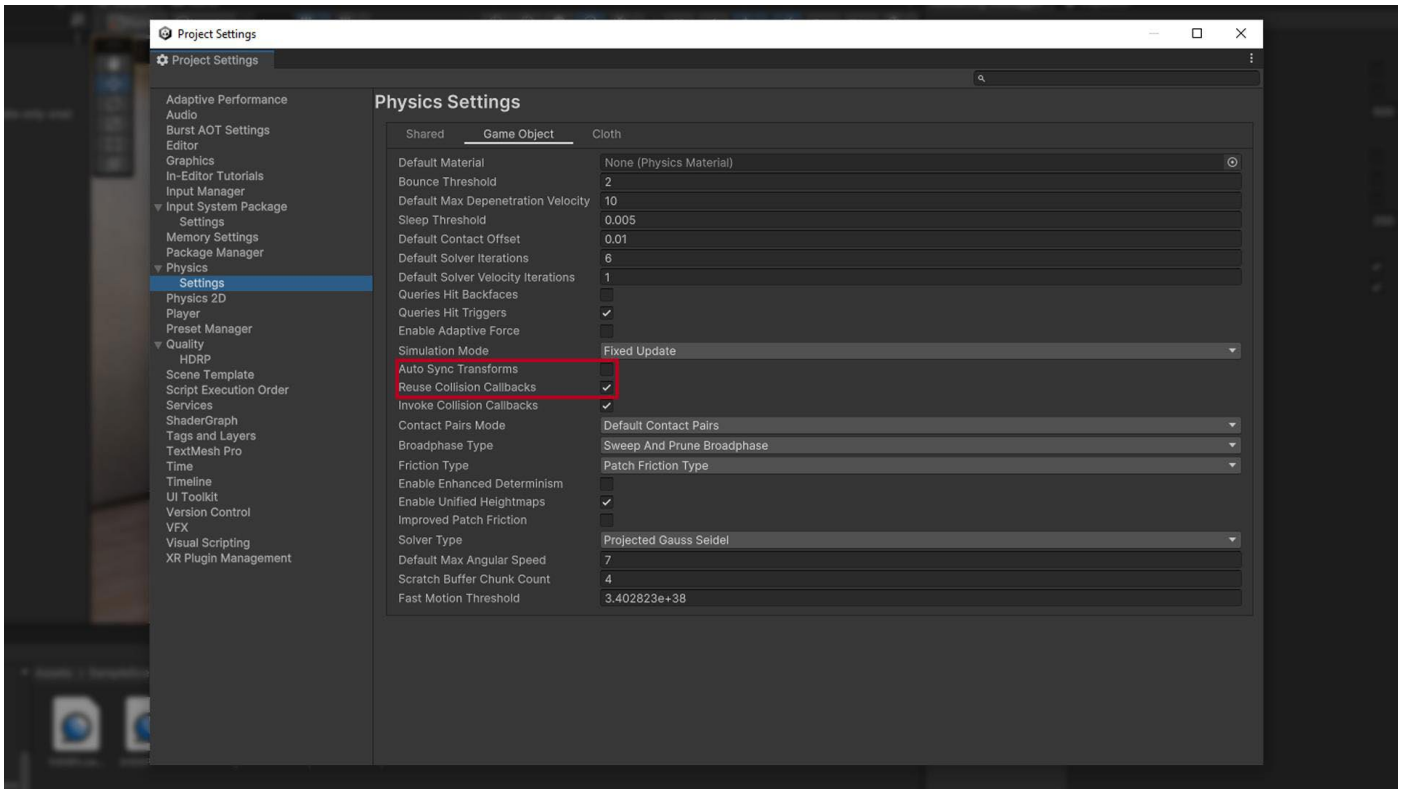
설정 최적화

가능하다면 항상 PlayerSettings에서 **Prebake Collision Meshes**를 선택합니다.



Prebake Collision Meshes 활성화

가능하다면 항상 물리 설정(**Project Settings > Physics**)을 편집하여 Layer Collision Matrix를 단순화하세요.

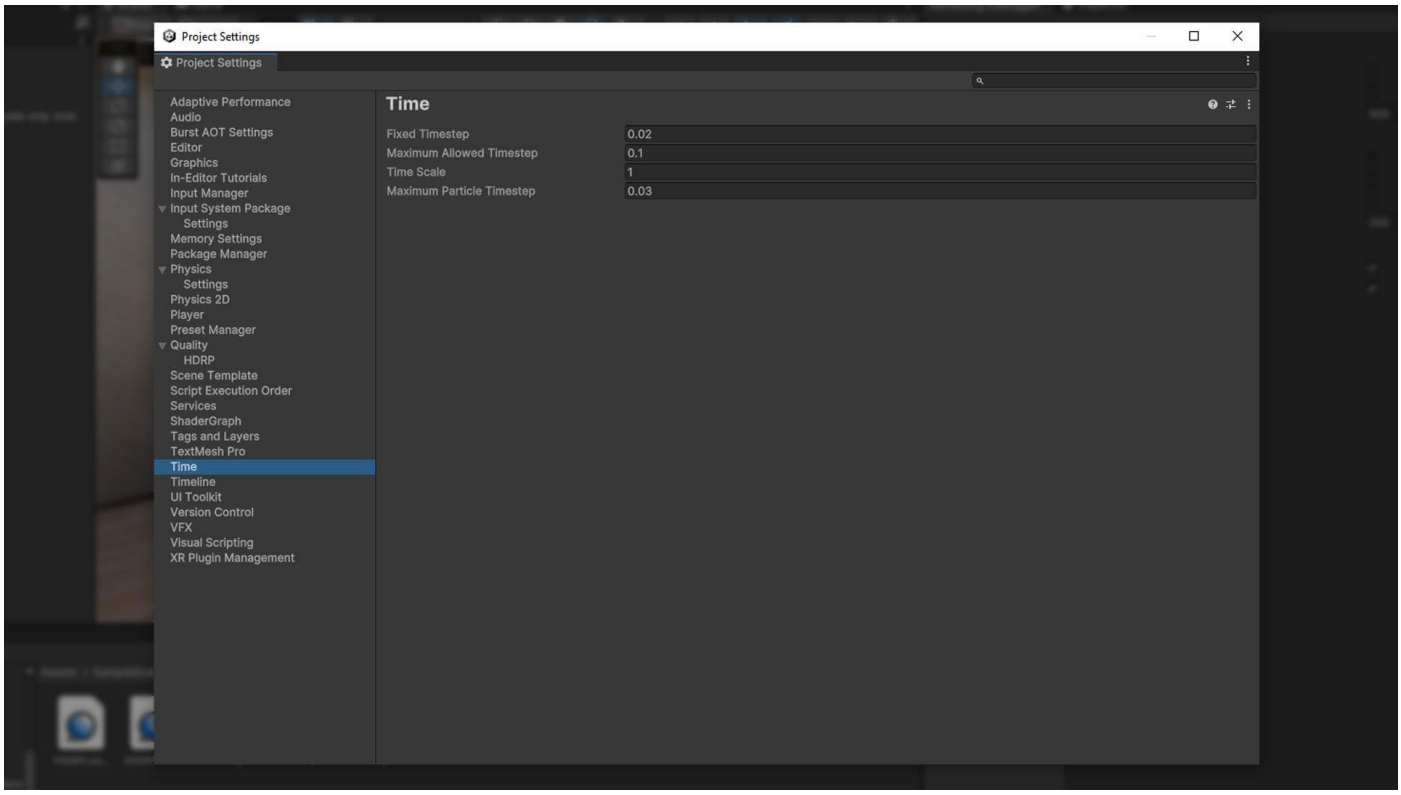


물리 프로젝트 설정을 수정하여 성능 향상

시뮬레이션 빈도 조정

물리 엔진은 고정된 타임 스텝에서 실행됩니다. 프로젝트가 실행되는 고정된 속도를 보려면 **Edit > Project Settings > Time**으로 이동하세요.

Fixed Timestep 필드는 각 물리 스텝에서 사용되는 시간 델타를 정의합니다. 예를 들어 기본값인 0.02초(20ms)는 50fps 또는 50Hz와 같습니다.



Project Settings에서 Fixed Timestep의 기본값은 0.02초(50fps)입니다.

Unity의 프레임은 각각 다른 시간을 소요하므로 물리 시뮬레이션과 완벽하게 동기화되지 않습니다. 엔진은 다음 물리 타임 스텝까지 계산합니다. 프레임이 약간 느리거나 빠르게 실행되는 경우, Unity는 경과 시간을 사용하여 물리 시뮬레이션을 실행할 적절한 타임 스텝을 파악합니다.

프레임을 준비하는 데 오랜 시간이 걸리는 경우, 성능 문제가 발생할 수 있습니다. 예를 들어 많은 게임 오브젝트를 인스턴스화하거나 디스크에서 파일을 로드하면서 게임에서 스파이크가 발생하는 경우, 프레임을 실행하는 데 40ms 이상 걸릴 수 있습니다. 기본값인 20ms Fixed Timestep을 사용하면, 가변 타임 스텝을 따라잡기 위해 다음 프레임에서 두 개의 물리 시뮬레이션이 실행됩니다.

물리 시뮬레이션을 추가적으로 사용하면 프레임을 처리하는 데 더 많은 시간이 들게 됩니다. 저사양 플랫폼에서는 이에 따라 잠재적으로 성능이 저하될 수 있습니다.

후속 프레임을 준비하는 데 시간이 더 오래 걸리면 물리 시뮬레이션도 밀리면서 점점 더 오래 걸립니다. 그러면 프레임이 느려지고 프레임당 더 많은 시뮬레이션이 실행되며, 결과적으로 성능이 계속해서 나빠집니다.

결국 물리 업데이트 사이의 시간이 Maximum Allowed Timestep을 초과할 수 있습니다. 이러한 중단이 발생하면 Unity는 물리 업데이트를 누락하기 시작하고 게임에는 끊김 현상이 발생합니다.

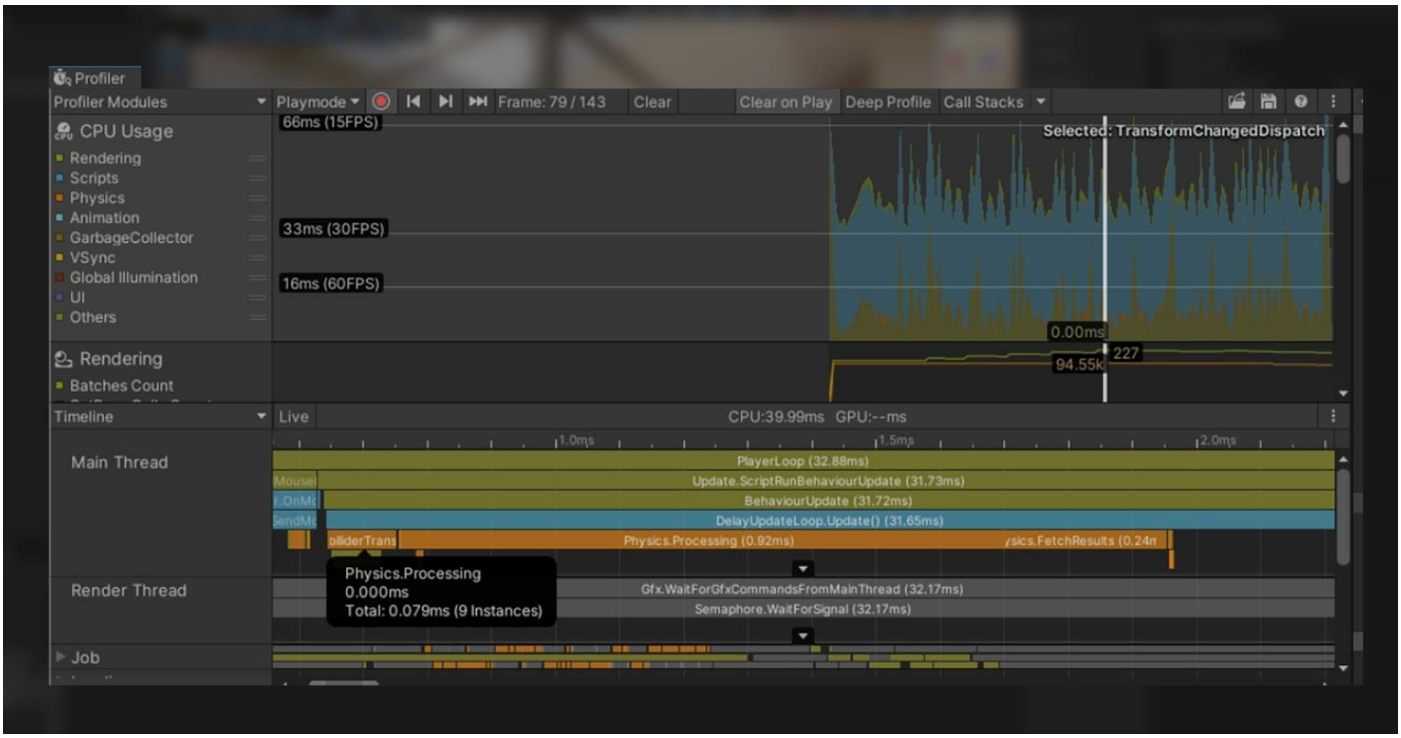
물리 관련 성능 문제를 방지하려면 다음을 수행하세요.

— 시뮬레이션 빈도를 줄입니다. 저사양 플랫폼의 경우, **Fixed Timestep**을 목표 프레임 속도보다 조금 더 높이세요. 예를 들어 모바일에서 30fps를 목표로 삼으면 Fixed Timestep을 0.035초로 설정합니다. 그러면 성능 저하 문제를 막을 수 있습니다.

— **Maximum Allowed Timestep**을 줄입니다. 0.1초 등의 더 작은 값을 사용하면 물리 시뮬레이션의 정확도는 조금 떨어지겠지만, 한 프레임에 발생할 수 있는 물리 업데이트의 수를 제한할 수 있습니다. 프로젝트 요구 사항에 맞는 값을 찾을 수 있도록 여러 값으로 테스트를 진행해 보세요.

— 물리 스텝을 수동으로 시뮬레이션할 필요가 있다면 프레임의 Update 단계에 SimulationMode를 선택하여 시뮬레이션합니다. 이를 통해 물리 스텝을 실행할 시점을 제어할 수 있습니다. 시뮬레이션 시간과 물리가 동기화되도록 유지하려면 Time.deltaTime을 Physics.Simulate에 전달하세요.

  이 방법은 복잡한 물리가 있거나 프레임 시간이 많이 변하는 씬에서 불안정한 물리 시뮬레이션을 유발할 수 있으므로, 주의해서 사용해야 합니다.



수동 시뮬레이션으로 Unity에서 씬 프로파일링

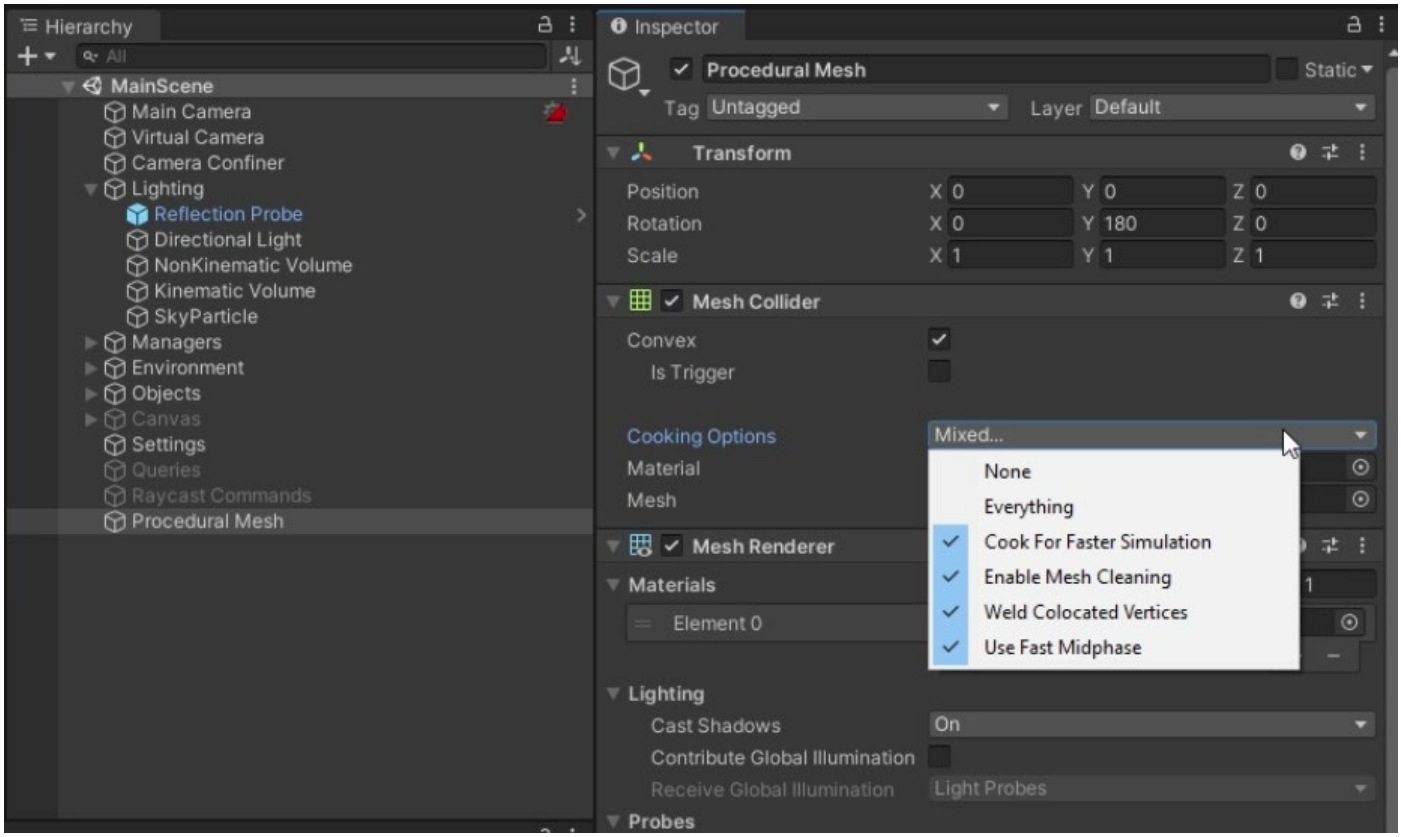
MeshCollider의 CookingOptions 수정

물리에서 사용하는 메시는 '쿠킹'이라는 프로세스를 거칩니다. 쿠킹 프로세스에서는 메시가 레이캐스트, 컨택트 등의 물리 쿼리와 함께 작동하도록 준비합니다.

MeshCollider에는 물리에 대한 메시를 검증하는 데 도움이 되는 여러 CookingOptions가 있습니다. 하지만 메시를 검증하지 않아도 된다는 확신이 있다면 옵션을 비활성화하여 쿠킹 시간을 단축할 수 있습니다.

각 MeshCollider의 CookingOption에서 EnableMeshCleaning, WeldColocatedVertices, CookForFasterSimulation의 선택을 해제하면 됩니다. 이 옵션은 런타임에 절차적으로 생성된 메시에 유용하지만, 메시에 이미 적절한 삼각형이 있다면 비활성화할 수 있습니다.

PC를 타겟 플랫폼으로 삼고 있다면 Use Fast Midphase도 계속 활성화 상태를 유지하세요. 이 옵션을 활성화하면 시뮬레이션 중간 단계에 PhysX 4.1에서 더 빠른 알고리즘으로 전환하게 되기 때문에, 물리 쿼리에 따라 교차할 가능성이 있는 소규모 삼각형 세트의 범위를 좁힐 수 있습니다.



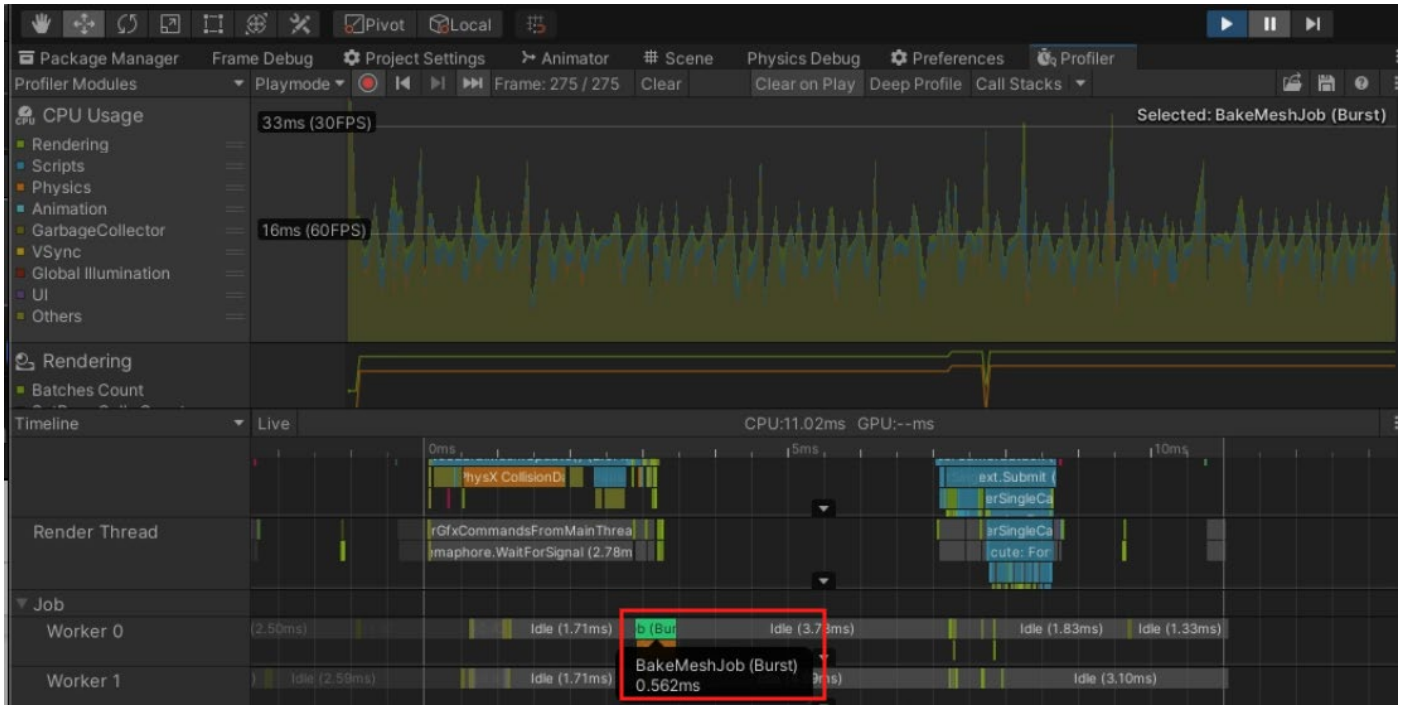
메시에 대한 쿠킹 옵션

Physics.BakeMesh 사용

게임플레이 중에 절차적으로 메시를 생성하는 경우, 메시 콜라이더를 런타임에 생성할 수 있습니다. 하지만 메시에 MeshCollider 컴포넌트를 바로 추가하면 메인 스레드에서 물리를 쿠킹/베이크하는데, 이 경우 상당한 CPU 시간을 소요할 수 있습니다.

Physics.BakeMesh를 사용하면 MeshCollider와 함께 사용할 메시를 준비하고 베이크된 데이터를 메시와 함께 저장할 수 있습니다. 이 메시를 참조하는 새로운 MeshCollider는 메시를 다시 베이크하지 않고 미리 베이크된 데이터를 재사용합니다. 그러면 나중에 씬 로드 시간이나 인스턴스화 시간을 줄일 수 있습니다.

성능 최적화를 위해 C# 잡 시스템을 사용하여 메시 쿠킹을 다른 스레드로 넘길 수 있습니다. 여러 스레드에서 메시를 베이크하는 방법에 대한 자세한 내용은 이 예제를 참조하시기 바랍니다.



프로파일러의 BakeMeshJob

대형 씬에서 Box Pruning 사용

Unity 물리 엔진은 다음 두 단계로 실행됩니다.

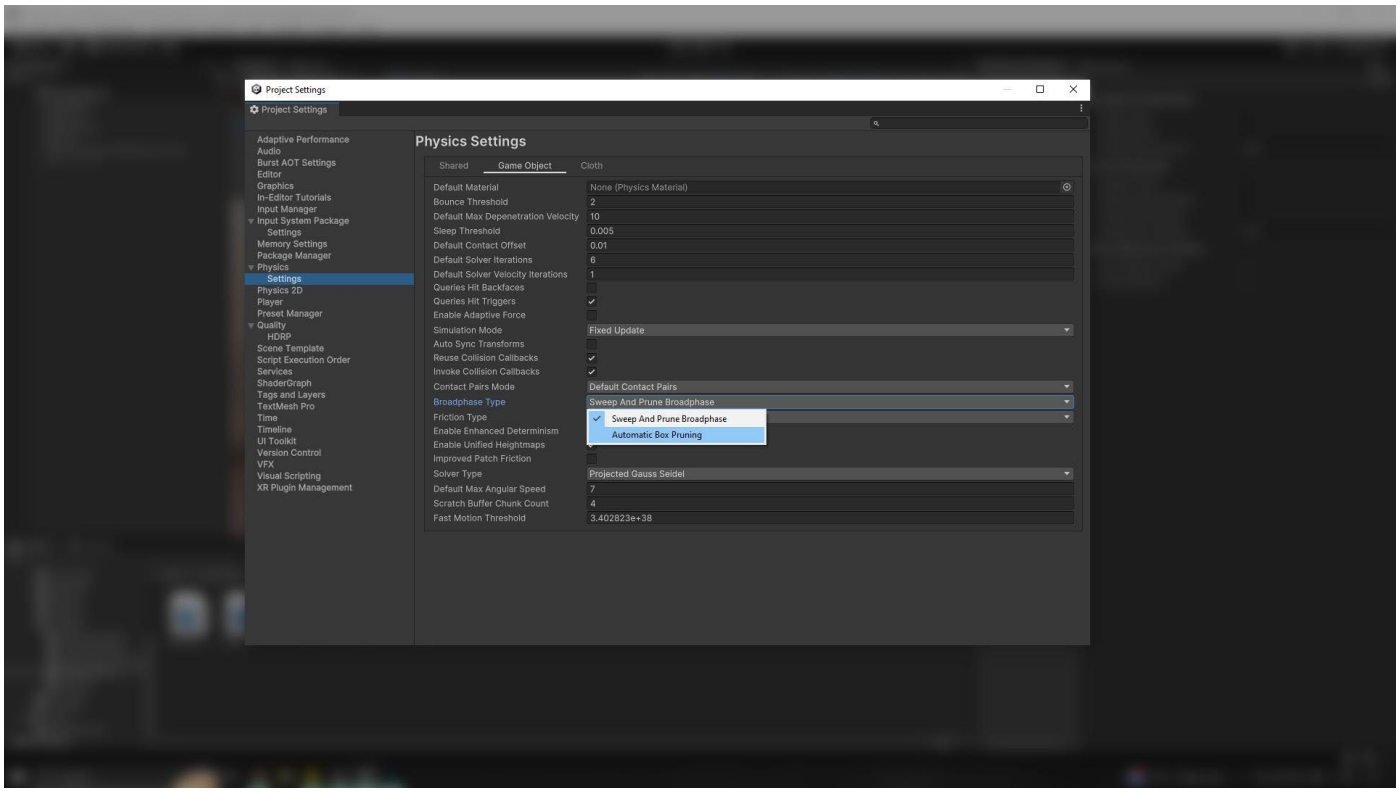
— 먼저 **광역 단계(Broad-phase)**에서 스윕 및 프룬(Sweep And Prune)알고리즘을 사용하여 잠재적 충돌을 파악합니다.

— 그런 다음 **지역 단계(Narrow-phase)**에서 엔진이 실제로 충돌을 계산합니다.

기본 광역 단계 설정인 Sweep And Prune Broadphase(**Edit > Project Settings > Physics > BroadPhase Type**)는 대체로 평평하고 콜라이더가 많은 월드에서 1종 오류를 생성할 수 있습니다.

씬이 크고 대체로 평평하다면 **Automatic Box Pruning**이나 **Multibox Pruning Broadphase**로 전환하여 이러한 오류를 방지해야 합니다. 이 옵션은 월드를 그리드로 나누고 각 그리드 셀에서 스윕 및 프룬을 수행합니다.

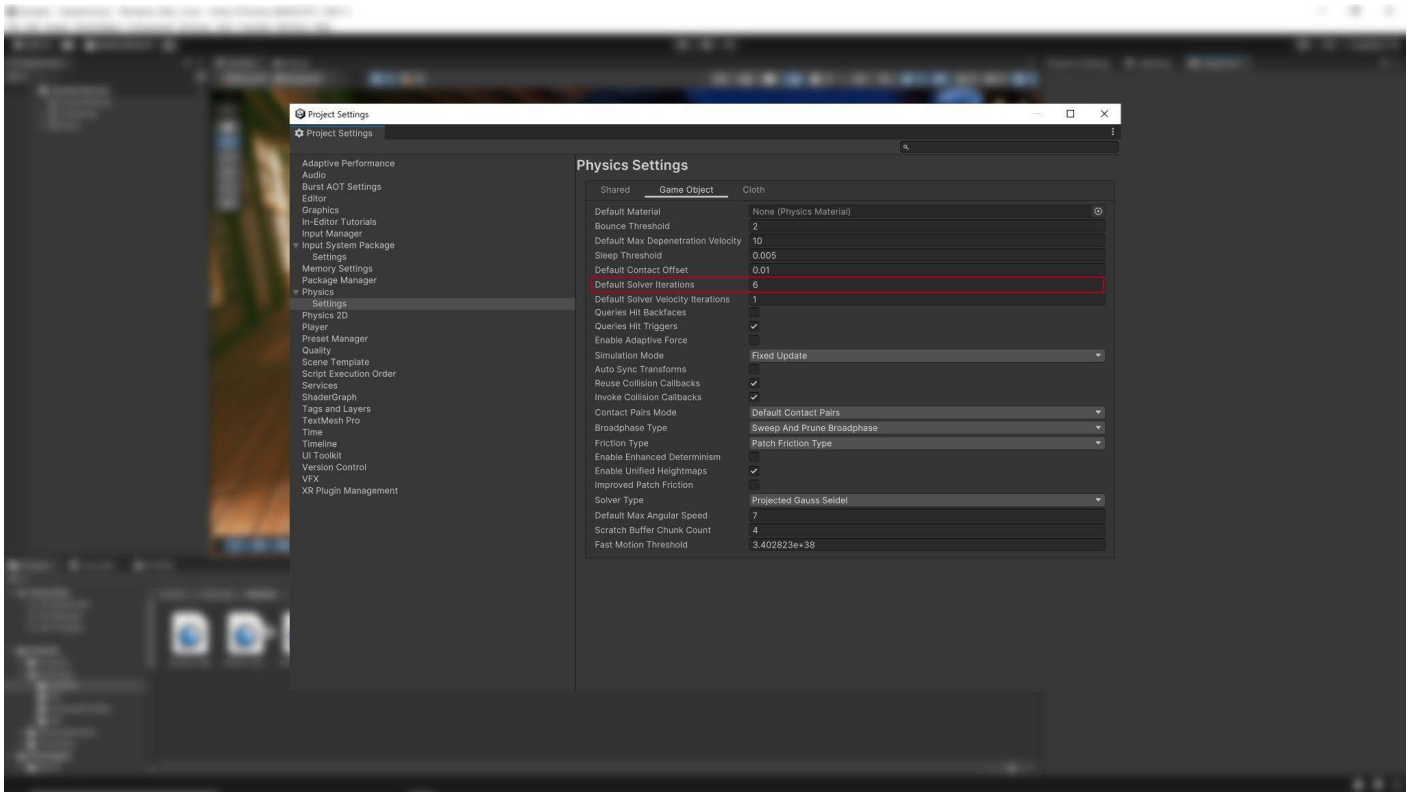
Multibox Pruning Broadphase를 사용하면 월드 경계와 그리드 셀 수를 수동으로 지정할 수 있으며, Automatic Box Pruning은 월드 경계와 그리드 수를 자동으로 계산합니다.



Physics 옵션의 Broadphase Type

솔버의 반복 횟수 수정

특정 물리 바디를 더 정확하게 시뮬레이션하려면 바디의 Rigidbody.solverIterations를 늘리면 됩니다.



리지드바디별 Default Solver Iterations 오버라이드

이 설정은 Physics.defaultSolverIterations를 오버라이드하는데, 이 옵션 역시 **Edit > Project Settings > Physics > Default Solver Iterations**에서 확인할 수 있습니다.

물리 시뮬레이션을 최적화하려면 프로젝트의 defaultSolveIterations에서 상대적으로 낮은 값을 설정하고, 더 자세한 정보가 필요한 개별 인스턴스에 더 높은 커스텀 Rigidbody.solverIterations 값을 적용해야 합니다.

자동 트랜스폼 동기화 비활성화

기본적으로 Unity는 트랜스폼 변경 사항을 물리 엔진과 자동으로 동기화하지 않습니다. 대신 다음 물리 업데이트나 수동 Physics.SyncTransforms 호출까지 기다립니다. 자동 동기화를 활성화하면 트랜스폼 또는 그 자식의 모든 리지드바디나 콜라이더가 물리 엔진과 자동으로 동기화됩니다.

수동 동기화 시점

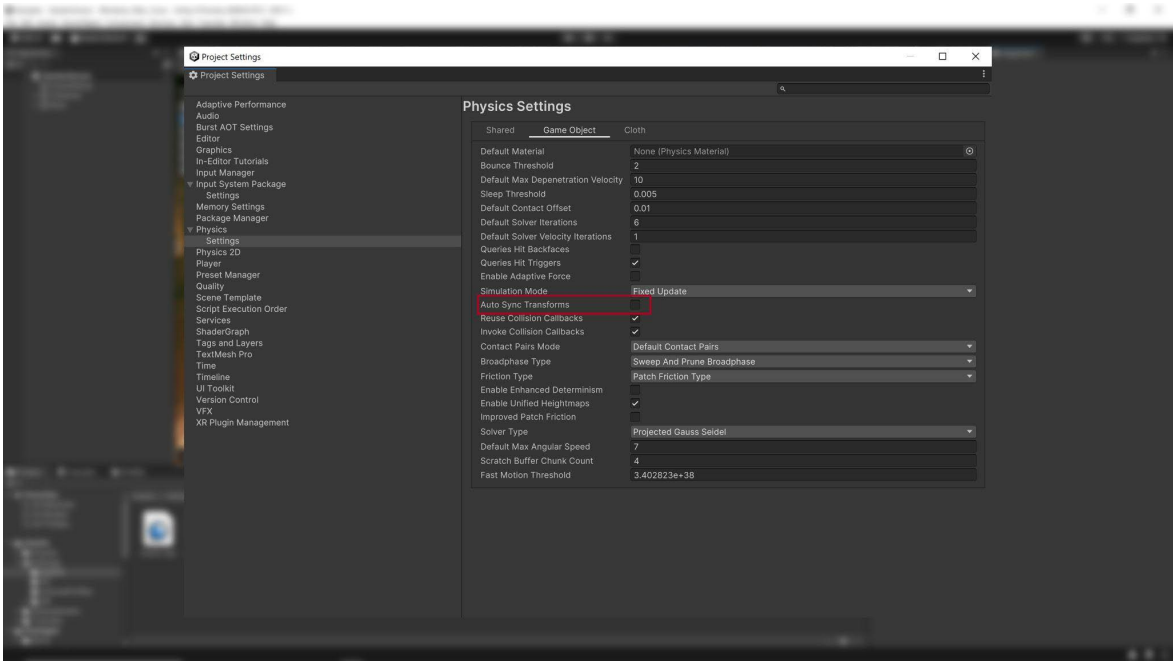
autoSyncTransforms가 비활성화된 경우, Unity는 FixedUpdate의 물리 시뮬레이션 단계 전이나 Physics.Simulate로 명시적으로 요청했을 때만 트랜스폼을 동기화합니다. 트랜스폼이 변경되고 물리 업데이트가 진행되기 전에 물리 엔진에서 직접 읽어오는 API를 사용한다면 추가로 동기화해 주어야 할 수 있습니다. **Rigidbody.position**에 액세스하거나 **Physics.Raycast**를 수행해야 하는 경우가 그 예시입니다.

성능 베스트 프랙티스

autoSyncTransforms를 사용하면 최신 물리 정보를 쿼리할 수 있지만, 성능 비용이 발생합니다. 물리 관련 API를 호출할 때마다 동기화가 강제되므로 특히 연속해서 쿼리하면 성능이 저하될 수 있습니다. 다음 베스트 프랙티스를 따르세요.

— **필요하지 않은 경우 autoSyncTransforms 비활성화**: 게임 메카닉에 정확하고 지속적인 동기화가 요구될 때만 활성화하세요.

— **수동으로 동기화**: 성능을 향상하려면 최신 트랜스폼 데이터가 필요한 API를 호출하기 전에 Physics.SyncTransforms()로 트랜스폼을 수동으로 동기화하세요. 전역에서 autoSyncTransforms를 활성화하는 것보다 더 효율적입니다.



Auto Sync Transform을 비활성화한 상태로 Unity에서 씬 프로파일링

컨택트 배열 사용

컨택트 배열은 충돌 데이터(컨택트)를 배열 포맷으로 저장하고 관리합니다. 모든 충돌 이벤트는 컨택트 지점의 배열을 생성하며, 이를 액세스하고 처리할 수 있습니다. 배열이므로 연속된 메모리 블록을 통해 충돌 데이터를 처리할 때 빠르게 액세스할 수 있으며, 배치 프로세스를 설정할 수 있고, 성능이 중요한 작업에 C# 잡 시스템과 함께 사용할 수 있습니다.

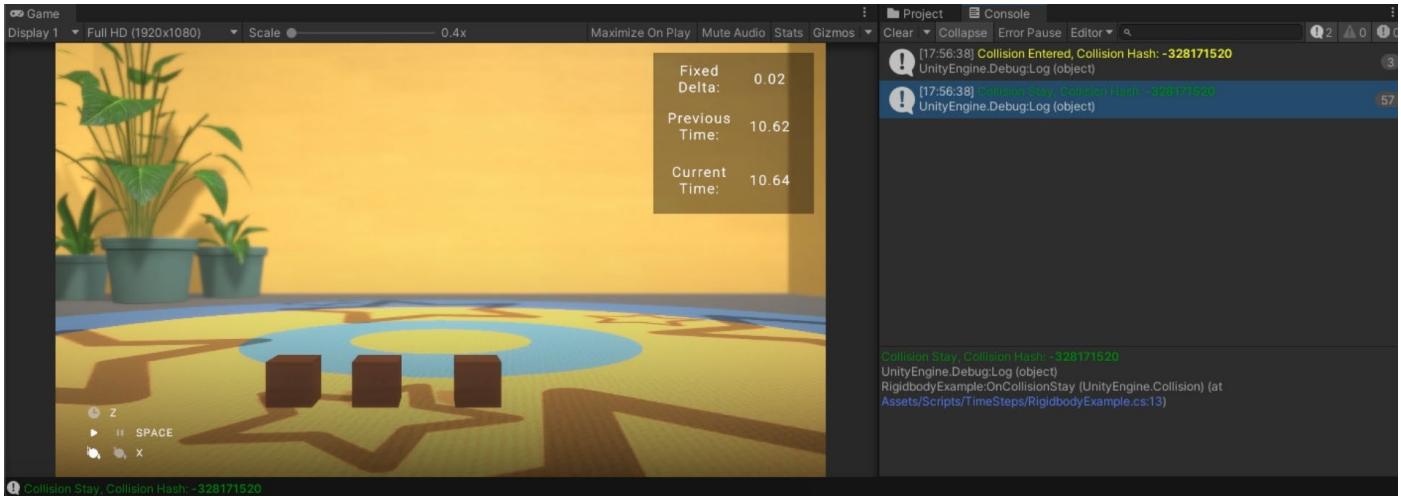
충돌 콜백 재사용

일반적으로 컨택트 배열을 사용하는 것이 충돌 콜백을 재사용하는 것보다 훨씬 빠르므로 컨택트 배열을 사용하는 것을 권장하지만, 특수한 활용 사례가 있다면 다음을 고려하세요.

MonoBehaviour.OnCollisionEnter, MonoBehaviour.OnCollisionStay, MonoBehaviour.OnCollisionExit 콜백은 모두 충돌 인스턴스를 파라미터로 사용합니다. 충돌 인스턴스는 관리되는 힙에 할당되며 가비지 컬렉션 대상이어야 합니다.

생성되는 가비지의 양을 줄이려면 Physics.reuseCollisionCallbacks를 활성화하세요. **Projects Settings > Physics > Reuse Collision Callbacks**에서도 이 설정을 확인할 수 있습니다. 이 옵션을 활성화하면 Unity는 각 콜백에 하나의 충돌 페어 인스턴스만 할당합니다. 그러면 가비지 컬렉터를 낭비하지 않고 성능을 개선할 수 있습니다.

일반적으로 성능 향상을 위해 항상 Reuse Collision Callbacks를 활성화하는 것을 권장합니다. 코드가 개별 충돌 클래스 인스턴스에 의존하여 개별 필드에 저장할 수 없는 레거시 프로젝트에서만 이 기능을 비활성화해야 합니다.



Unity 콘솔에서 Collision Entered 및 Collision Stay에는 하나의 충돌 인스턴스가 있습니다.

정적 콜라이더 이동

정적 콜라이더는 콜라이더 컴포넌트는 있지만 리지드바디는 없는 게임 오브젝트입니다.

'정적'이라는 표현과 달리 정적 콜라이더는 움직일 수 있습니다. 정적 콜라이더를 옮기려면 물리 바디의 위치를 수정하면 됩니다. 위치 변화를 축적하고 물리 업데이트 전에 동기화하세요. 정적 콜라이더를 옮기기 위해 리지드바디 컴포넌트를 추가할 필요는 없습니다.

그러나 정적 콜라이더가 다른 물리 바디와 더 복잡한 방식으로 상호 작용하게 하려면 키네마틱 리지드바디를 추가하세요. Transform 컴포넌트에 액세스하지 않고도 Rigidbody.position과 Rigidbody.rotation을 사용해 정적 콜라이더를 옮길 수 있습니다. 그러면 물리 엔진에서 더 예측 가능한 동작을 시뮬레이션할 수 있습니다.

참고: 런타임에 개별 정적 콜라이더 2D를 이동하거나 재설정해야 하는 경우, Rigidbody 2D 컴포넌트를 추가하고 Body Type을 Static으로 설정하세요. Collider 2D에 Rigidbody 2D가 있으면 시뮬레이션이 빨라집니다. 런타임에 콜라이더 2D 그룹을 이동하거나 재구성해야 하는 경우, 각 게임 오브젝트를 개별적으로 이동하는 것보다 모두 숨겨진 한 부모 리지드바디 2D의 자식으로 만드는 것이 더 빠릅니다.

비할당 쿼리 사용

특정 거리 내에서 특정 방향으로 3D 프로젝트에서 콜라이더를 탐지하고 수집하려면 레이캐스트와 기타 물리 쿼리(BoxCast 등)를 사용하면 됩니다.

OverlapSphere나 OverlapBox 등 여러 콜라이더를 배열로 반환하는 물리 쿼리는 관리되는 힙에 해당 오브젝트를 할당해야 합니다. 다시 말해 가비지 컬렉터는 결국 할당된 오브젝트를 수집해야 하므로 잘못된 시간에 오브젝트를 수집하면 성능이 저하될 수 있습니다.

오버헤드를 줄이려면 물리 쿼리의 **NonAlloc** 버전을 사용하세요. 예를 들어 OverlapSphere를 사용해 주변의 모든 잠재적 콜라이더를 수집한다면 OverlapSphereNonAlloc을 사용하면 됩니다.

그러면 버퍼 역할을 하는 콜라이더 배열(결과 파라미터)을 전달할 수 있습니다. NonAlloc 메서드는 가비지를 생성하지 않지만, 해당하는 할당 메서드처럼 작동합니다.

NonAlloc 메서드를 사용할 때는 충분한 크기의 결과 버퍼를 정의해야 합니다. 버퍼 공간이 부족하면 버퍼가 커지지 않습니다.

2D 물리

위에서 설명한 내용은 2D 물리 쿼리에는 적용되지 않습니다. Unity 2D 물리 시스템의 메서드에는 'NonAlloc' 접미사가 없기 때문입니다. 대신 여러 결과를 반환하는 메서드를 포함한 모든 2D 물리 메서드는 배열이나 리스트를 받는 오버로드된 버전을 제공합니다. 예를 들어 3D 물리 시스템에는 RaycastNonAlloc과 같은

메서드가 있지만, 2D 물리에서는 단순히 다음과 같이 배열이나 List<T>를 파라미터로 받는 Raycast의 오버로드된 버전을 사용합니다.

```
var results = new List<RaycastHit2D>();
int hitCount = Physics2D.Raycast(origin, direction, contactFilter, results);
```

오버로드를 사용하면 특수한 NonAlloc 메서드 없이도 2D 물리 시스템에서 비할당 쿼리를 수행할 수 있습니다.

레이캐스트를 위한 쿼리 배칭

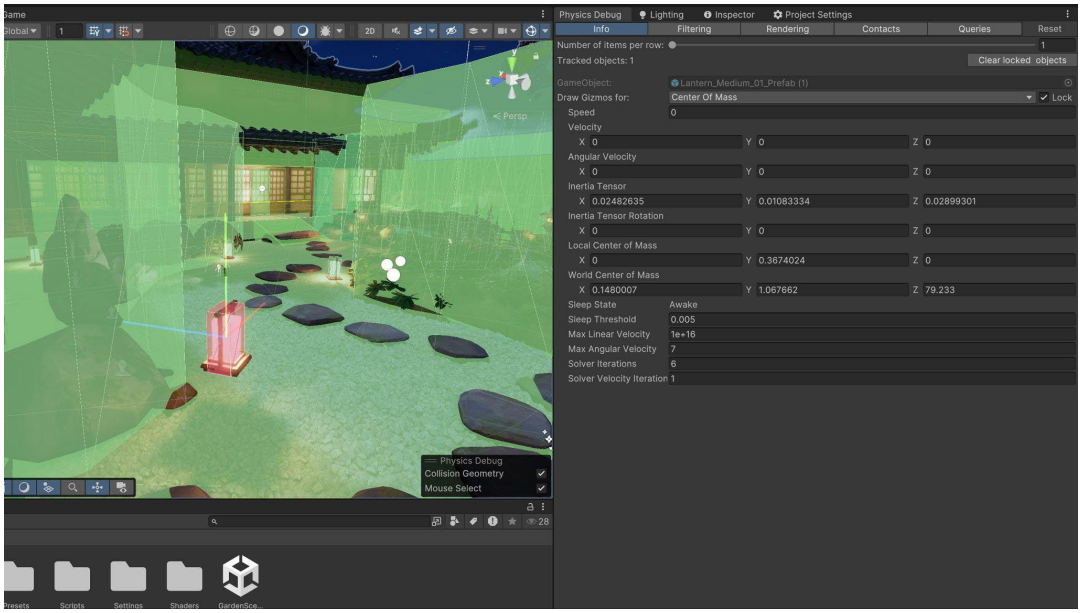
Physics.Raycast로 레이캐스트 쿼리를 실행할 수 있습니다. 그러나 10,000개의 에이전트에 대한 가시선을 계산하는 등 레이캐스트 연산이 많은 경우 CPU 시간이 많이 소요될 수 있습니다.

RaycastCommand를 사용하면 C# 잡 시스템을 사용해 쿼리를 배칭할 수 있습니다. 이렇게 하면 메인 스레드에서 작업 부하를 줄일 수 있어 레이캐스트를 비동기 병렬로 수행할 수 있습니다.

RaycastCommands 기술 자료 페이지의 사용 예제를 참조하시기 바랍니다.

물리 디버거를 통한 시각화

Physics Debug 창(**Window > Analysis > Physics Debugger**)을 사용하면 콜라이더나 불일치로 인한 문제를 해결할 수 있습니다. Physics Debug 창은 서로 충돌할 수 있는 게임 오브젝트를 색으로 구별하여 표시합니다.



물리 오브젝트의 상호 작용을 시각화할 수 있는 물리 디버거

자세한 내용은 물리 디버거 기술 자료를 참조하시기 바랍니다.

애니메이션

다음은 Unity로 애니메이션 작업 시 유용한 팁입니다. 애니메이션 시스템에 관한 종합 가이드가 필요하다면 Unity 애니메이션에 관한 최종 가이드 무료 전자책을 다운로드하세요.

휴머노이드 릭(rig) 대신 제네릭 릭 사용

Unity는 기본적으로 제네릭 릭을 사용하여 애니메이션화된 모델을 임포트하지만, 개발자가 캐릭터를 애니메이션화할 때 휴머노이드 릭으로 전환하는 경우가 많습니다. 릭과 관련하여 다음 사항을 주의하세요.

— 가능한 한 항상 제네릭 릭을 사용하세요. 휴머노이드 릭은 사용 중이지 않을 때에도 역운동학(IK)과 애니메이션 리타게팅을 프레임마다 계산합니다. 따라서 동등한 제네릭 릭보다 30-50% 많은 CPU 시간을 소비합니다.

— 휴머노이드 애니메이션을 임포트할 때 아바타 마스크를 사용하여 필요 없는 IK 골이나 손가락 애니메이션을 제거하세요.

— 뼈대 가중치를 가능한 한 적게 유지하세요.

제네릭 릭은 휴머노이드 릭보다 CPU 시간을 적게 소모합니다.

간단한 애니메이션에 대체 기능 사용

애니메이터는 휴머노이드 캐릭터를 주 대상으로 하지만, 단일 값(예: UI 요소의 알파 채널)을 애니메이션화하는 데 재사용되는 경우도 많습니다. 특히 UI 요소와 함께 사용되는 경우에는 추가 오버헤드가 발생하므로 애니메이터를 과도하게 사용하지 않도록 하세요.

현재의 애니메이션 시스템은 애니메이션 블렌딩을 비롯한 복잡한 설정에 최적화되어 있습니다. 블렌딩에 사용되는 임시 버퍼가 있고, 샘플링된 커브와 기타 데이터가 추가로 복사되어 있습니다.

정적 애니메이션에서 특히 군중과 같이 중복이 많은 애니메이션의 경우 베이크된 애니메이션을 사용해 볼 수 있습니다. 흔히 두 가지 기법을 사용합니다. 노멀 및 탄젠트 버퍼를 포함하는 버텍스 텍스처 애니메이션을 사용하는 기법과 버텍스 셰이더의 수동 스키닝에 베이크된 뼈대 매트릭스를 사용하는 기법입니다.

애니메이션 베이크에 대해 자세히 알아보려면 이 Unity Learn 튜토리얼과 Llam Academy의 이 튜토리얼을 살펴보세요.

가능하다면 애니메이션 시스템을 아예 사용하지 않는 것도 고려해 보세요. easing 함수를 만들거나 타사 트위닝 라이브러리(예: DOTween)를 사용하여, 수학식으로 매우 자연스럽게 보간할 수 있습니다.

스케일 커브 사용 지양

스케일 커브 애니메이션은 이동 및 회전 커브 애니메이션보다 많은 리소스를 소모합니다. 성능을 개선하려면 스케일 애니메이션 사용을 지양하세요.

> 참고: 이는 상수 커브(애니메이션 클립의 길이 값이 같은 커브)에 적용되지 않습니다. 상수 커브는 최적화되고 일반적인 커브보다 적은 리소스를 소모합니다.

시야에 들어올 때에만 업데이트

애니메이터의 Culling Mode를 Based on Renderers로 설정하고 Skinned Mesh Renderer의 Update When Offscreen 프로퍼티를 비활성화합니다. 이렇게 하면 캐릭터가 보이지 않을 때 Unity에서 애니메이션을 업데이트하지 않습니다.

워크플로 최적화

씬 수준에서 추가로 최적화를 진행할 수 있습니다.

— 문자열 대신 해시를 사용하여 애니메이터를 쿼리합니다.

— 작은 AI 레이어를 구현하여 애니메이터를 제어합니다. OnStateChange와 OnTransitionBegin 등의 기타 이벤트에 간단한 콜백을 제공하도록 할 수 있습니다.

— 상태 태그를 사용하면 AI 상태 머신과 Unity 상태 머신을 손쉽게 연결할 수 있습니다.

— 추가 커브를 사용하여 이벤트를 시뮬레이션합니다.

— 타겟 매칭 등과 함께 추가 커브를 사용하여 애니메이션을 마크업합니다.

애니메이션 계층 구조 분리

씬의 루트를 제외하고는 애니메이션 계층이 같은 부모를 공유하지 않도록 합니다. 이렇게 분리하면 애니메이션 결과를 게임 오브젝트에 다시 쓸 때 성능에 큰 영향을 줄 수 있는 스레딩 문제를 방지할 수 있습니다.

바인딩 비용 최소화

애니메이션 시스템에서 바인딩 연산의 높은 비용에 주의해야 합니다. 성능을 최적화하려면 클립을 자주 추가하거나, 게임 오브젝트와 컴포넌트를 추가하고 제거하거나, 런타임에 오브젝트를 활성화하고 비활성화하여 리바인딩을 수행하도록 하는 것을 피하세요. 이러한 연산은 모두 리소스를 많이 소모합니다.

깊은 계층 구조에 컴포넌트 기반 제약 사용 방지

복잡한 캐릭터처럼 깊은 계층 구조에 컴포넌트 기반 제약을 사용하면 성능이 저하될 수 있으므로 피하는 것이 좋습니다.

애니메이션 리깅의 성능 오버헤드 고려

애니메이션 리깅을 사용하는 경우, 각 제약에 따라 추가되는 성능 오버헤드에 주의하세요. 휴머노이드 모델을 작업할 때는 이를 고려하는 것이 중요합니다. 가능한 경우 휴머노이드 릭의 빌트인 IK(역운동학) 패스를 사용하여 성능을 향상하세요.

워크플로 및 협업

Unity로 애플리케이션을 제작하는 일은 많은 개발자가 참여하기도 하는 공동 과업입니다. 프로젝트를 팀 작업에 적합하게 설정하세요.

버전 관리 사용

VCS(버전 관리 시스템)을 사용하면 전체 프로젝트의 기록을 남길 수 있습니다. 작업을 조직적으로 구성하고 팀이 효율적으로 반복 작업할 수 있습니다.

프로젝트 파일은 저장소 또는 '리포(repo)'라고 하는 공유 데이터베이스에 저장됩니다. 프로젝트를 정기적으로 저장소에 백업하고, 문제가 발생하면 프로젝트를 이전 버전으로 되돌릴 수 있습니다.

VCS를 사용하면 여러 가지 개별적인 변경 사항을 적용하고 버전 관리를 위해 단일 그룹으로 커밋할 수도 있습니다. 그러면 이 커밋은 프로젝트 타임라인의 한 지점으로 자리 잡게 되므로, 이전 버전으로 되돌리면 해당 커밋의 모든 변경 사항이 취소되고 이전 상태로 복원됩니다. 커밋 내 그룹화된 각 변경 사항을 검토하고 수정할 수 있으며, 커밋 전체를 실행 취소할 수도 있습니다.

프로젝트의 전체 기록을 확인할 수 있으므로 어떤 변경 사항이 버그를 유발했는지 파악하고, 이전에 삭제한 기능을 복원하고, 게임 또는 제품 릴리스 간 변경 사항을 문서화하는 일이 쉬워집니다.

또한 버전 관리는 보통 클라우드나 분산된 서버에 저장되므로 개발 팀이 어디서든 협업할 수 있습니다. 이러한 이점은 원격 근무가 보편화되면서 그 중요도가 더욱 높아지고 있습니다.

원활한 버전 관리 병합을 위해 에디터 설정에서 **Asset Serialization Mode**가 **Force Text**로 설정되도록 하세요. 이렇게 하면 공간 효율성이 떨어지지만 Unity에서 씬 파일을 텍스트 기반 포맷으로 저장할 수 있습니다.

Asset Serialization Mode

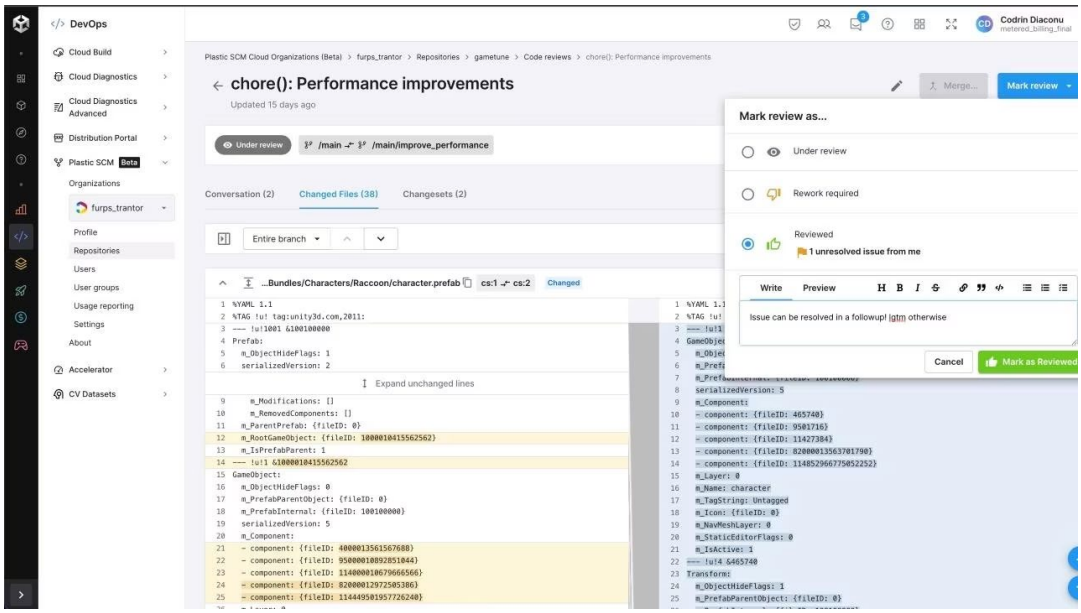
Version Control 설정에서 외부 버전 관리 시스템(예: Git)을 사용하고 있다면 Mode가 Visible Meta Files로 설정되어 있어야 합니다.

Version Control Mode

아울러 Unity의 빌트인 YAML(사람이 읽을 수 있는 데이터 직렬화 언어) 툴을 사용하면 특히 씬과 프리팹을 병합하는 데 도움이 됩니다. 자세한 내용은 Unity 기술 자료의 Smart Merge 페이지를 참조하세요.

Unity Version Control

대부분의 Unity 프로젝트에는 스크립트 코드 외에도 수많은 아트 에셋이 포함되어 있습니다. 버전 관리를 사용해 에셋을 관리하려면 UVCS(Unity Version Control)(기존 Plastic SCM)로 전환하는 것이 좋습니다. Git LFS를 사용하더라도, Git은 대형 저장소에서 Plastic SCM처럼 뛰어난 성능을 발휘하지 못합니다. Plastic SCM은 500MB가 넘는 대용량 바이너리 파일 처리 속도가 빠르기 때문입니다.



Unity 대시보드의 Unity Version Control 웹 경험 화면

UVCS는 고유한 인터페이스로 프로그래머와 아티스트를 모두 지원하는 유연한 버전 관리 시스템입니다. 대규모 저장소와 바이너리 파일 처리에 탁월하며, 파일 기반 및 체인지 세트 기반 솔루션을 모두 지원하므로 프로젝트 빌드 전체가 아닌 작업 중인 특정 파일만 다운로드할 수 있습니다.

UVCS는 3가지 방법으로 사용할 수 있습니다. UVCS 데스크톱 클라이언트를 통해 여러 애플리케이션과 저장소를 사용하거나, Unity Hub를 통해 프로젝트에 UVCS를 추가하거나, 웹 브라우저를 통해 Unity Cloud의 저장소에 액세스할 수 있습니다.

UVCS는 다음 기능을 제공합니다.

— 아트 에셋을 안전하게 백업한 상태로 작업

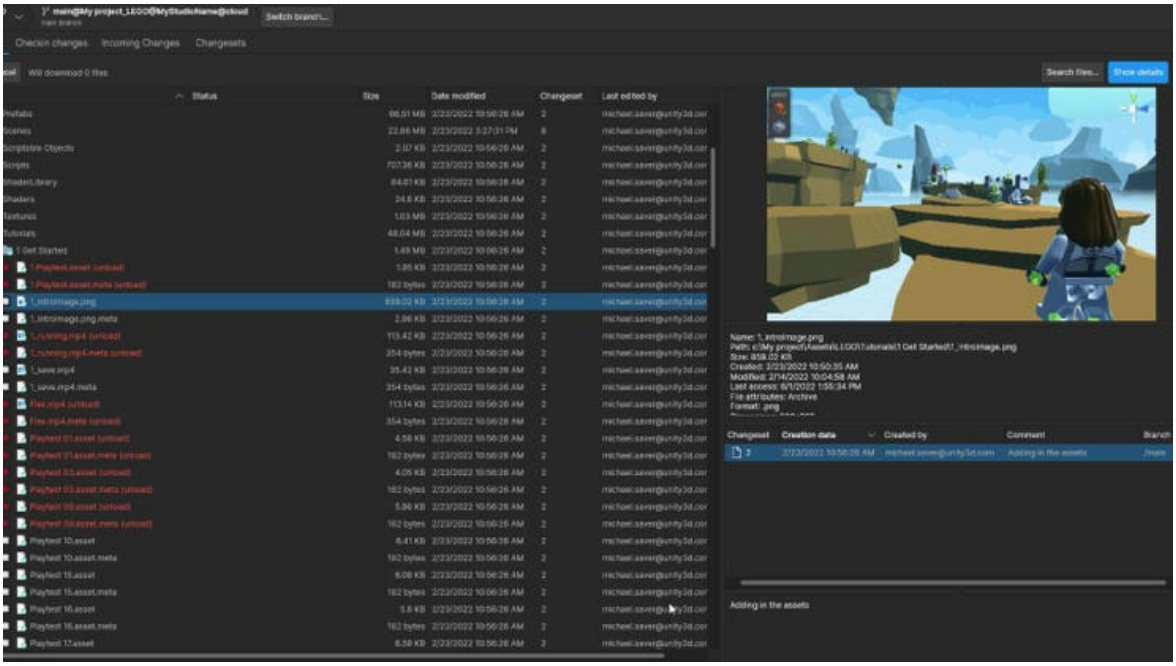
— 모든 에셋의 소유권 추적

— 에셋의 이전 반복 작업으로 롤백

— 하나의 중앙 저장소에서 자동화된 프로세스 사용

— 여러 플랫폼에 대한 브랜치를 빠르고 안전하게 생성

또한 UVCS를 사용하면 뛰어난 시각화 툴을 통해 개발을 중앙 집중화할 수 있습니다. 특히 아티스트는 전체 프로젝트 저장소의 복잡함을 신경 쓸 필요 없이 관리하려는 파일만 간단하게 확인할 수 있는 Gluon 애플리케이션을 사용하여 개발 팀과 아트 팀의 긴밀한 협업을 촉진할 수 있는 사용자 친화적 워크플로가 큰 도움이 될 것입니다. 간소화된 워크플로를 제공할 뿐만 아니라 에셋 버전을 간단하게 비교하고 통합된 버전 관리 환경에 쉽게 기여할 수 있는 툴도 제공합니다.



대용량 에셋도 처리할 수 있는 아티스트 친화적인 UI를 제공하는 UVCS

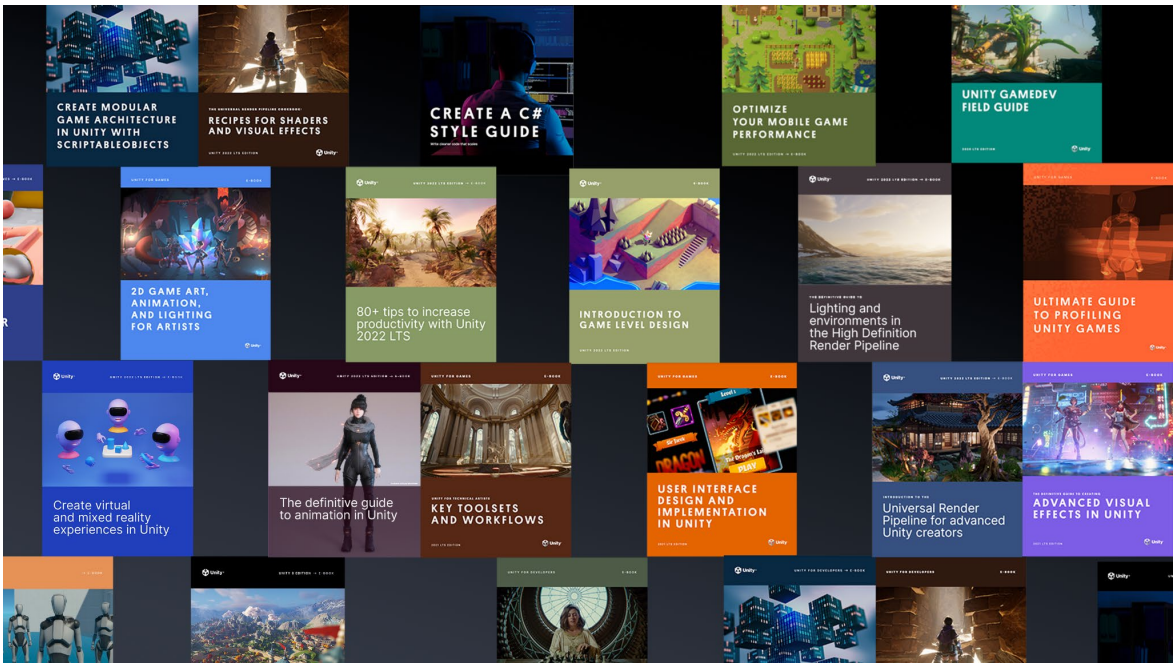
Unity Version Control을 시작하려면 버전 관리 및 프로젝트 구성 베스트 프랙티스 전자책을 살펴보세요.

대형 씬 분할

Unity에서 하나로 이루어진 대형 씬은 협업에 적합한 편이 아닙니다. 여러 개의 소형 씬으로 레벨을 분할하면 아티스트와 디자이너가 서로 다르게 작업할 위험을 최소화하며 단일 레벨에서 더 원활하게 협업할 수 있습니다.

런타임 시 프로젝트에서 **SceneManager.LoadSceneAsync**를 사용하여 씬을 가산적으로 로드하고 **LoadSceneMode.Additive** 파라미터 모드를 전달할 수 있습니다.

숙련된 개발자 및 아티스트를 위한 리소스



Unity 베스트 프랙티스 허브에서 숙련된 Unity 개발자 및 크리에이터를 위한 더 많은 전자책을 다운로드하실 수 있습니다. 게임 개발 베스트 프랙티스를 제공하고 Unity의 툴셋과 시스템을 효율적으로 사용하는 데 도움이 되도록 업계 전문가, 유니티 엔지니어 및 테크니컬 아티스트가 제작한 30개 이상의 가이드를 살펴보세요.

추가 팁, 베스트 프랙티스, 최신 소식은 Unity 블로그 및 Unity 커뮤니티 포럼에서 확인할 수 있으며, Unity Learn과 **#unitytips** 해시태그를 통해서도 확인할 수 있습니다.

unity.com/kr