전자책

Unity에서 모바일, XR, 웹용 게임 성능 최적화

Unity®

목차

들어가는 말

이 가이드에는 Unity 6에서 모바일, XR, Unity Web 성능을 최적화하는 모든 최신 팁이 담겨 있습니다. 유니티는 본 가이드와 Unity에서 콘솔 및 PC 게임 성능 최적화 가이드 등 2개의 최적화 가이드를 제공합니다.

모바일, XR 및 Unity Web 애플리케이션 최적화는 전체 게임 개발 사이클의 기반이 되는 필수 프로세스입니다. 진화를 거듭하는 하드웨어 사양에 맞춰, 게임 최적화는 아트, 게임 디자인, 오디오, 수익화 전략과 함께 플레이어 경험을 형성하는 핵심적인 역할을 합니다.

모바일, XR 및 웹 게임의 액티브 사용자 수는 수십억 명에 달합니다. 모바일의 경우 게임의 최적화 수준을 높이면 플랫폼별 스토어에서 요구하는 인증을 더 원활하게 통과할 수 있습니다. 최대한 다양한 기기에서 성능이 보장된 애플리케이션을 제공한다는 목표를 삼아야만 출시 후 성공의 기회를 극대화할 수 있습니다.

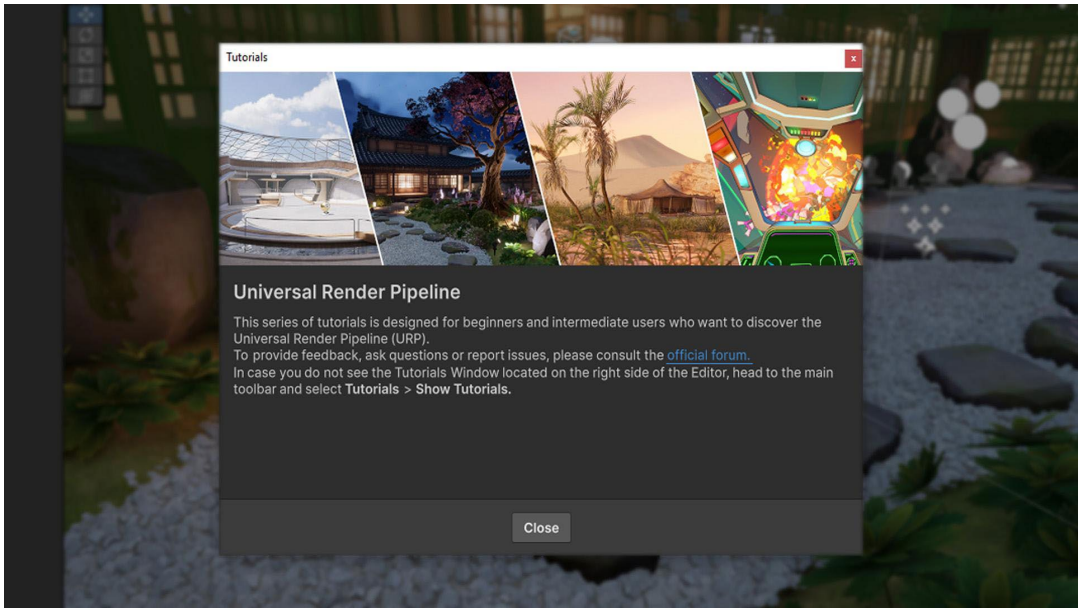
이 전자책에는 업계 전반의 개발자들과 협력하여 최고의 게임을 출시할 수 있도록 지원하는 Unity 엔지니어들의 지식과 조언이 담겨 있습니다.

유니티 팀의 지원을 받아 최적화를 시작해 보세요.[1]

[1] 참고로 여기에서 소개하는 최적화 방안 중 다수는 복잡도를 높일 가능성이 있으며, 이에 따라 유지 관리 비용이 늘어나고 버그가 발생할 수 있습니다. 따라서 베스트 프랙티스를 구현할 때는 성능 향상에 필요한 시간과 노력을 감안하시기 바랍니다.

URP로 성능과 비주얼 품질 향상하기

유니티는 XR(확장 현실), 웹, 모바일 게임 및 애플리케이션 개발에 URP(유니버설 렌더 파이프라인)를 사용할 것을 권장합니다. URP는 높은 성능과 확장성을 제공하기 위해 만들어졌으며, 다양한 하드웨어에서 원활하게 작동하는 효율적인 렌더링을 제공합니다. 양호한 성능을 유지하면서 비주얼 품질을 향상할 수 있으므로 WebGL이나 모바일 기기와 같이 리소스 효율성이 중요한 플랫폼에 이상적입니다. URP를 사용하면 손쉽게 커스터마이징할 수 있으므로 애플리케이션을 여러 환경에서 최적의 상태로 실행할 수도 있습니다.

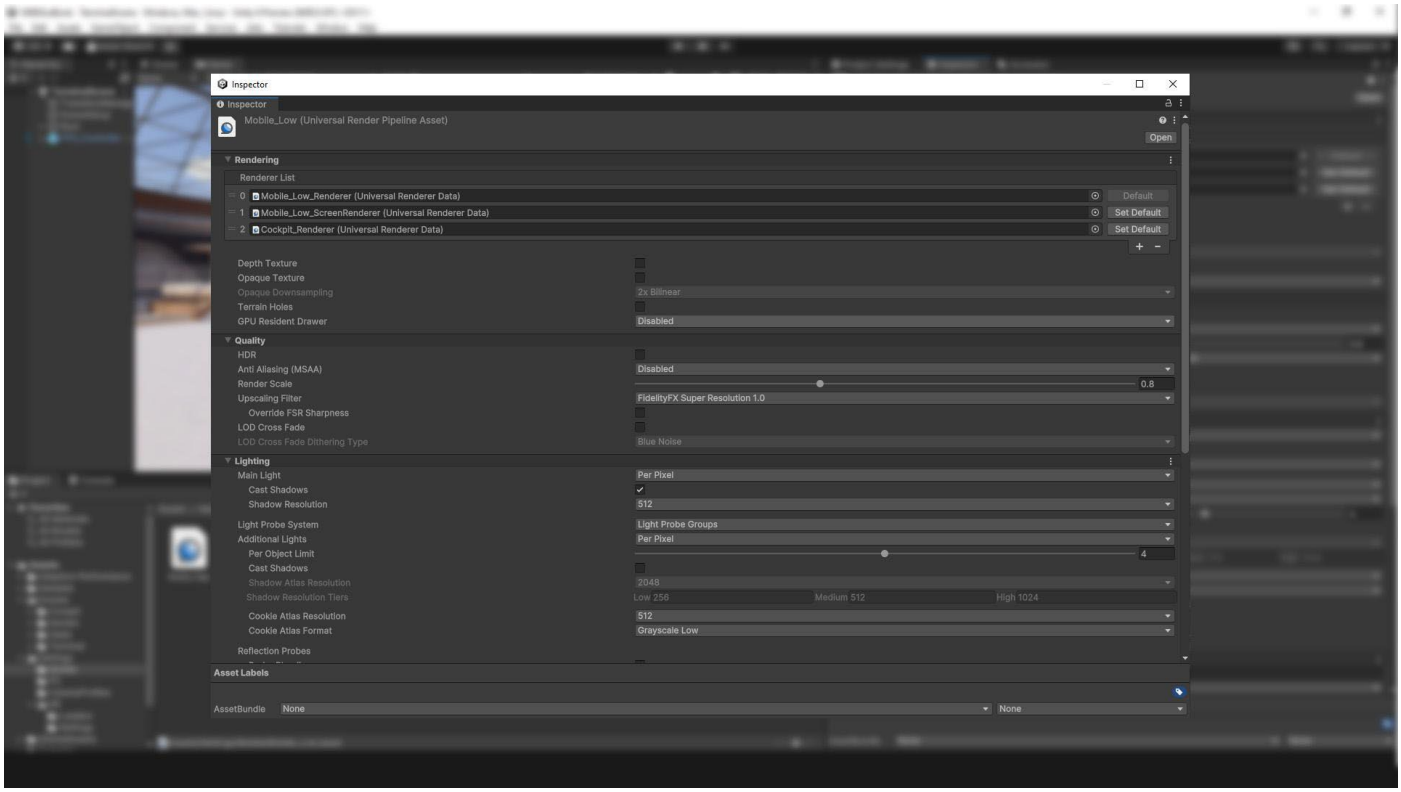


Unity 모바일, XR 또는 웹 게임을 개발하는 경우 URP를 렌더 파이프라인으로 선택하세요.

URP를 사용하는 것 외에도 렌더 파이프라인 에셋을 조정해 더 구체적으로 설정을 커스터마이징할 수 있습니다.

렌더링 최적화

URP는 무선 VR 경험이나 모바일 기기의 AR 앱에서 품질과 성능을 향상하기 위한 프리셋을 제공합니다. 적절한 렌더링 설정을 선택하면 모바일 하드웨어에 맞게 애플리케이션을 최적화하여 효율적인 렌더링과 원활한 성능을 보장할 수 있습니다. URP 설정을 최적화하면 텍스처 품질, 그림자 해상도, 조명을 효율적으로 관리하여 시각적 정확도와 성능 간의 균형을 맞추고 모바일 및 무선 XR 기기의 제약에 맞는 성능을 제공할 수 있습니다.



렌더 파이프라인 에셋

프로파일링 팁

개발 초기부터 자주 타겟 기기에서 프로파일링하기

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.

모바일, XR 및 웹 프로젝트의 경우 출시가 임박했을 때뿐만 아니라 개발 초기부터 애플리케이션을 지속적으로 프로파일링하는 것이 중요합니다. 글리치(glitch)나 스파이크 같은 성능 문제가 발생하면 즉시 해결하고, 주요 변경 사항을 적용할 때 전후 성능을 벤치마킹하세요. 프로젝트의 '성능 프로파일'을 명확하게 수립하면 새로운 문제를 더 쉽게 발견하고 해결하면서 모든 타겟 플랫폼에서 최적의 성능을 보장할 수 있습니다.

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

Unity는 병목 현상을 식별하는 데 도움이 되는 여러 프로파일링 툴을 제공하며,

이러한 툴로는 Unity 프로파일러, Memory Profiler 및 Profile Analyzer가 있습니다. iOS 및 Android에서 자사 하드웨어의 성능을 추가로 테스트하기 위해 제공하는 네이티브 툴도 있습니다.

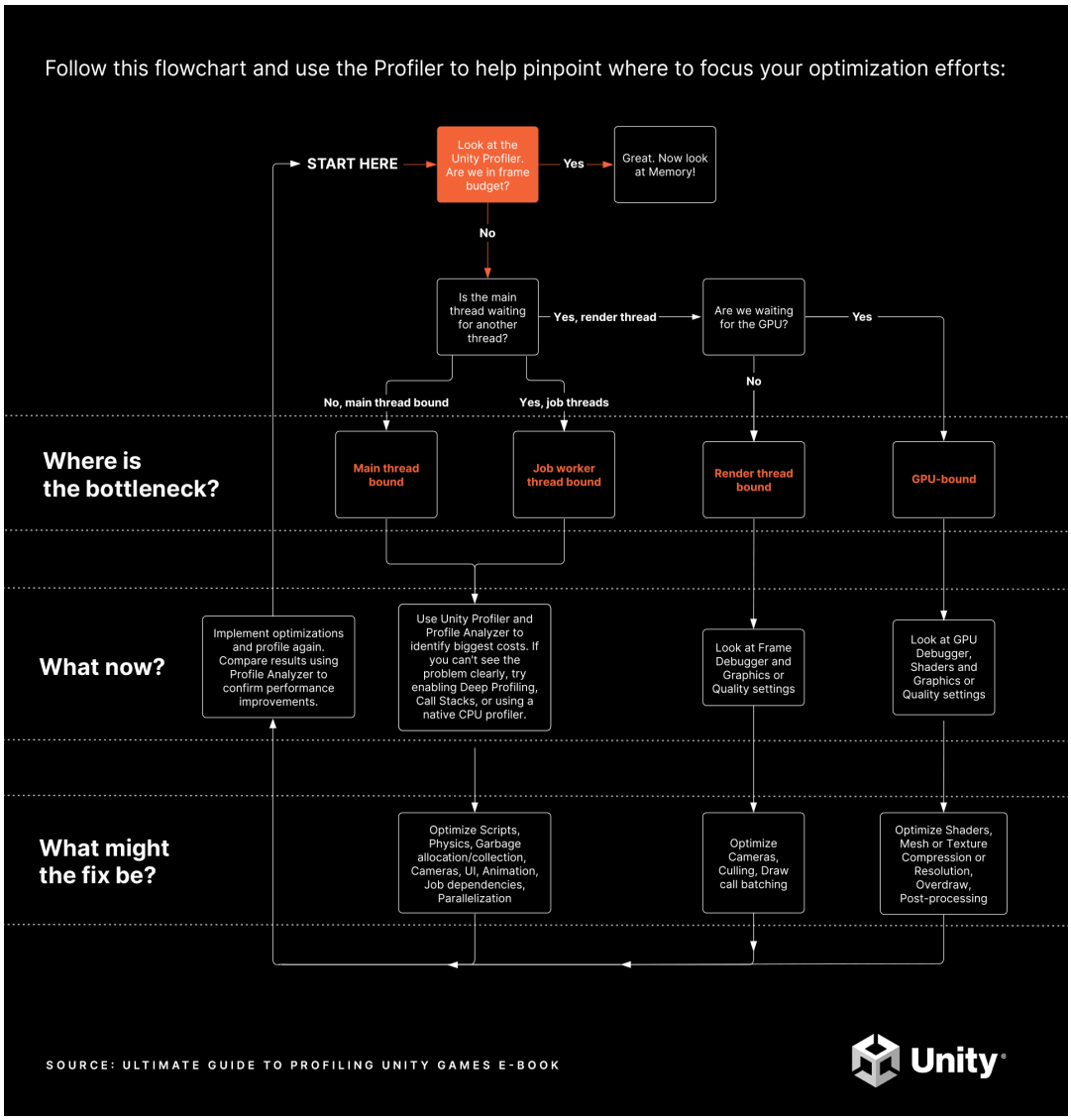
- iOS: Xcode 및 Instruments
- Android/Arm:
  - Android Studio: 최신 Android Studio에는 이전의 Android Monitor 툴을 대체하는 새로운 Android Profiler가 포함되어 있습니다. 이 툴을 사용하여 Android 기기의 하드웨어 리소스에 대한 실시간 데이터를 수집할 수 있습니다.
  - Arm Mobile Studio: Arm 하드웨어를 활용하는 기기에 맞춰 게임을 매우 세세하게 프로파일링하고 디버깅하는 툴 세트입니다.
  - Snapdragon Profiler: Snapdragon 칩셋을 탑재한 기기 전용 툴입니다. CPU, GPU, DSP, 메모리, 전력, 발열, 네트워크 데이터를 분석하여 성능 병목 현상을 찾고 해결합니다.
  - Meta Quest용 개발자 툴: Meta Quest 헤드셋용 앱 개발에 대해 자세히 알아보려면 Meta의 개발자 툴 웹사이트를 참고하세요.

특정 하드웨어에서는 Intel VTune을 사용하여 Intel 플랫폼에서의 성능 병목 지점을 파악하고 해결할 수 있습니다(Intel 프로세서 전용).

올바른 영역 최적화

게임 성능을 저하하는 요인을 추정하거나 가정하지 않도록 하세요. Unity 프로파일러 및 플랫폼별 툴을 사용하여 성능 저하의 정확한 원인을 찾아야 합니다. 프로파일링 툴이 본질적으로 Unity 프로젝트의 내부 상황을 파악하는 데 도움이 되기는 하지만, 심각한 성능 문제가 발생한 이후에야 감지 툴을 살펴보는 일은 피해야 합니다.

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.



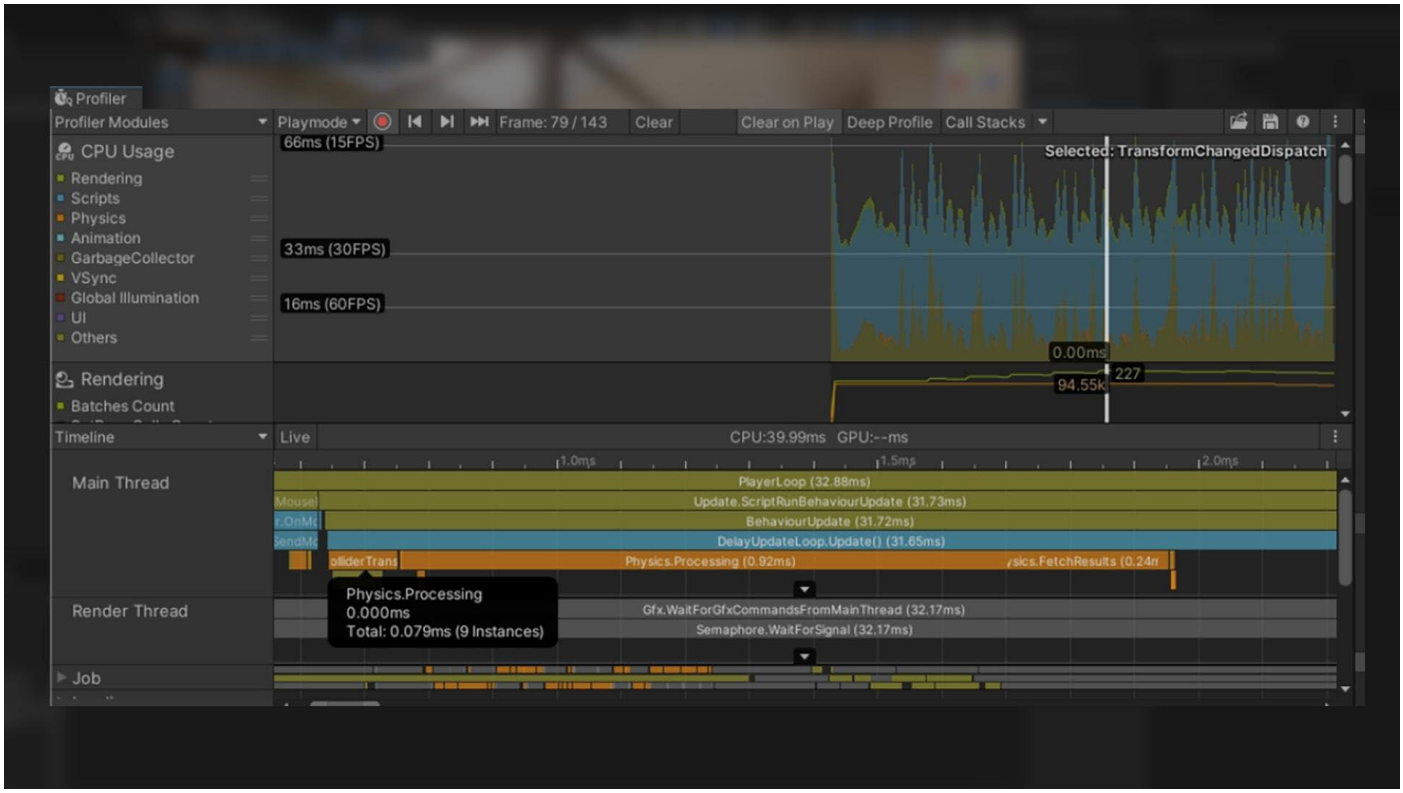
Unity 프로젝트를 효율적으로 프로파일링하기 위한 워크플로의 예시를 보여 주는 차트(프로파일링 전자책에서 발췌)

Unity 프로파일러의 작동 방식 이해

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

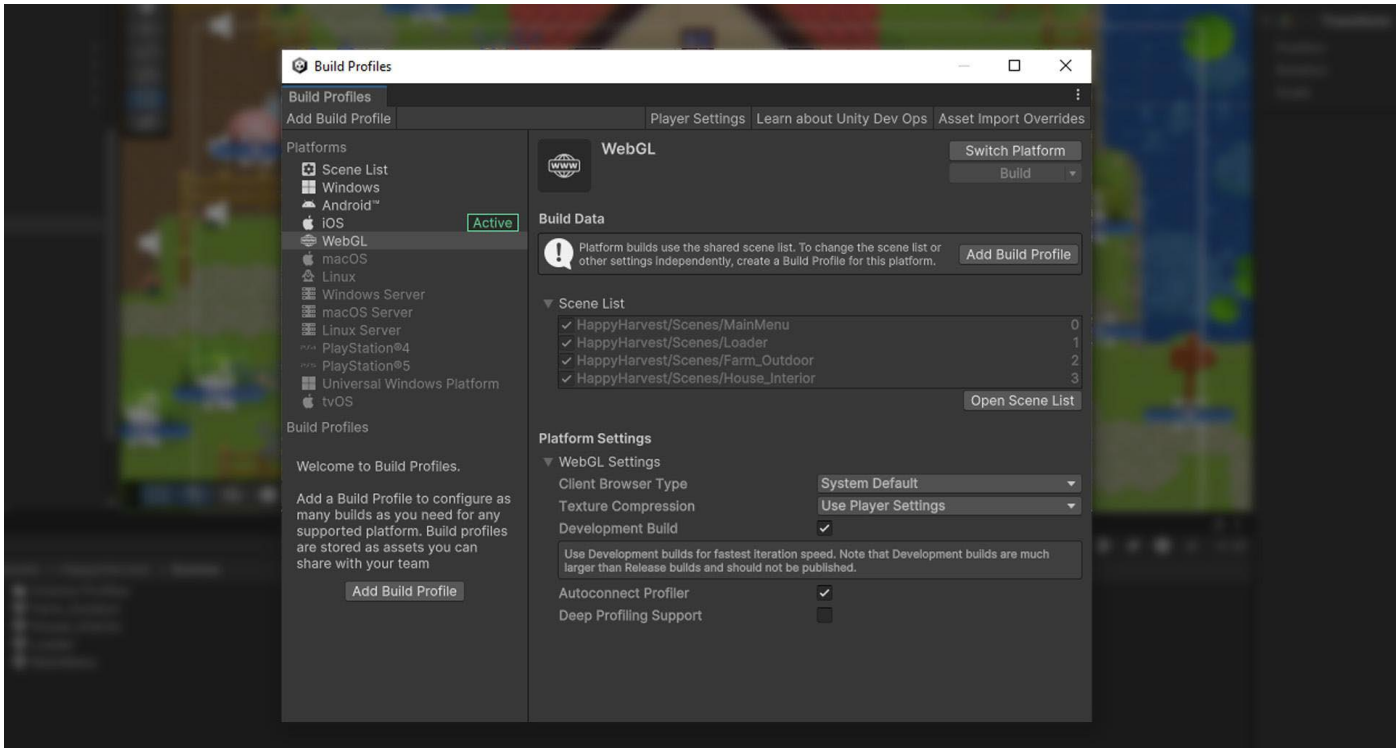
Unity 프로파일러는 계측 기반 프로파일러입니다. MonoBehaviour의 Start 또는 Update 메서드나 특정 API 호출처럼 자동으로 마크업되거나, ProfilerMarker API를 사용하여 명시적으로 래핑된 게임 및 엔진 코드의 타이밍을 프로파일링합니다.

CPU 및 메모리 트랙을 기본적으로 활성화하세요. 예를 들어 물리가 많이 사용되거나 음악을 기반으로 하는 게임플레이에서는 필요에 따라 렌더러, 오디오, 물리 같은 보조 프로파일러 모듈을 모니터링할 수 있습니다.



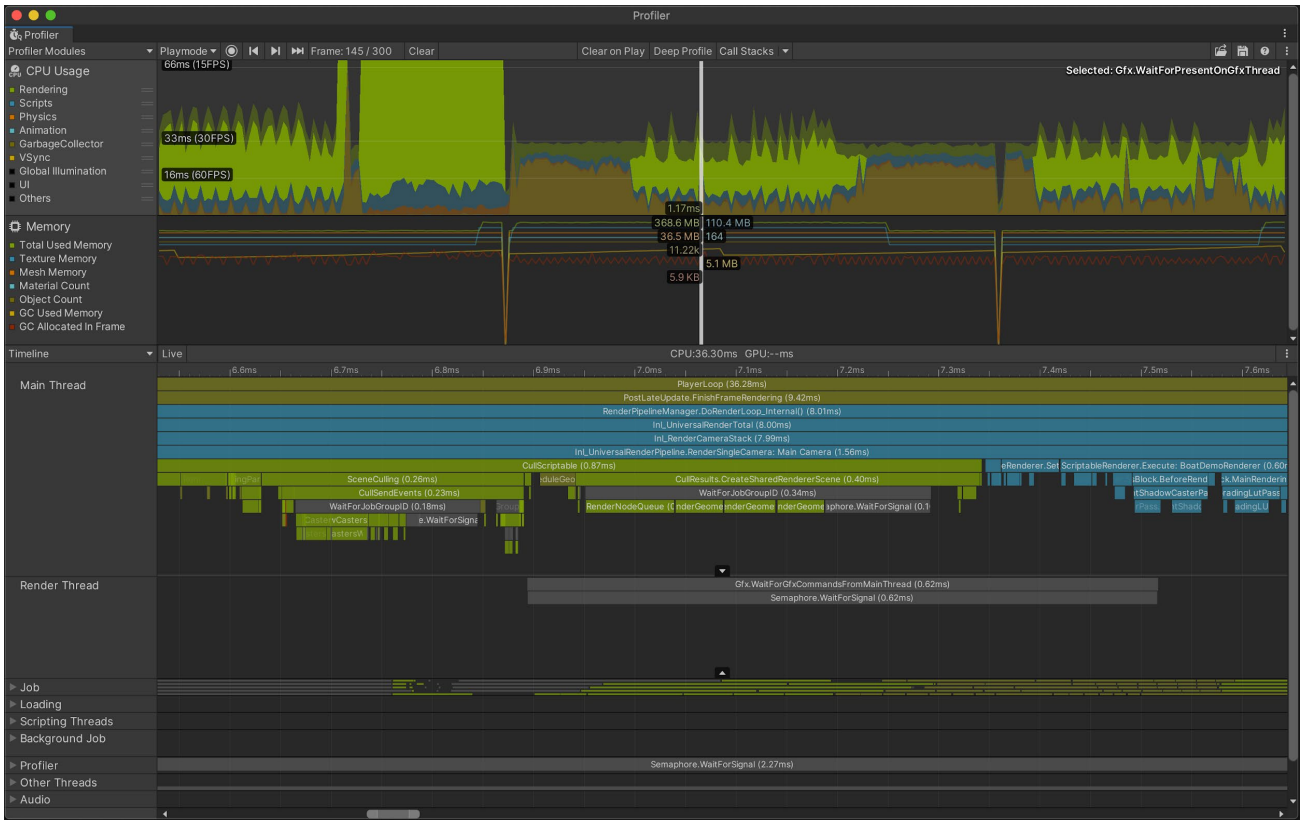
Unity 프로파일러를 사용하여 애플리케이션의 성능 및 리소스 할당 테스트

원하는 플랫폼에서 실제 모바일 기기의 프로파일링 데이터를 캡처하려면 **Build and Run**을 클릭하기 전에 **Development Build** 및 **Autoconnect Profiler** 박스를 선택하세요. 프로파일링과 별도로 앱을 시작하고 싶다면 **Autoconnect Profiler** 박스를 선택 해제하고 앱이 실행된 후에 수동으로 연결하면 됩니다.



프로파일링하기 전 빌드 설정 조정

프로파일링을 실행할 타겟 플랫폼을 선택하세요. **녹화** 버튼을 누르면 애플리케이션 플레이를 몇 초간 추적할 수 있습니다(기본 300프레임). 더 오래 캡처해야 한다면 **Unity > Preferences > Analysis > Profiler > Frame Count**로 이동하여 이 값을 2000까지 늘릴 수 있습니다. 그러면 Unity 에디터가 더 많은 CPU 작업을 수행하고 더 많은 메모리를 사용하게 되지만 상황에 따라 이 설정이 더 유용할 수 있습니다.



Timeline 뷰를 사용하여 CPU 바운드 또는 GPU 바운드 여부를 판단합니다.

애플리케이션에 대한 상세한 정보를 파악해 심도 있게 분석해야 하는 경우에는 Deep Profiling 설정을 사용하세요. 이 설정을 사용하면 Unity는 스크립트 코드에 있는 모든 함수 호출의 시작과 끝을 프로파일링하여 정확히 애플리케이션의 어느 부분이 실행되고 있으며 잠재적으로 지연을 유발하는지 제시합니다. 하지만 Deep Profiling을 사용할 경우 모든 메서드 호출에 오버헤드가 추가되고, 프로파일링 세션 중에 게임의 실행 속도가 느려지므로 성능 분석 결과가 왜곡될 수 있습니다.

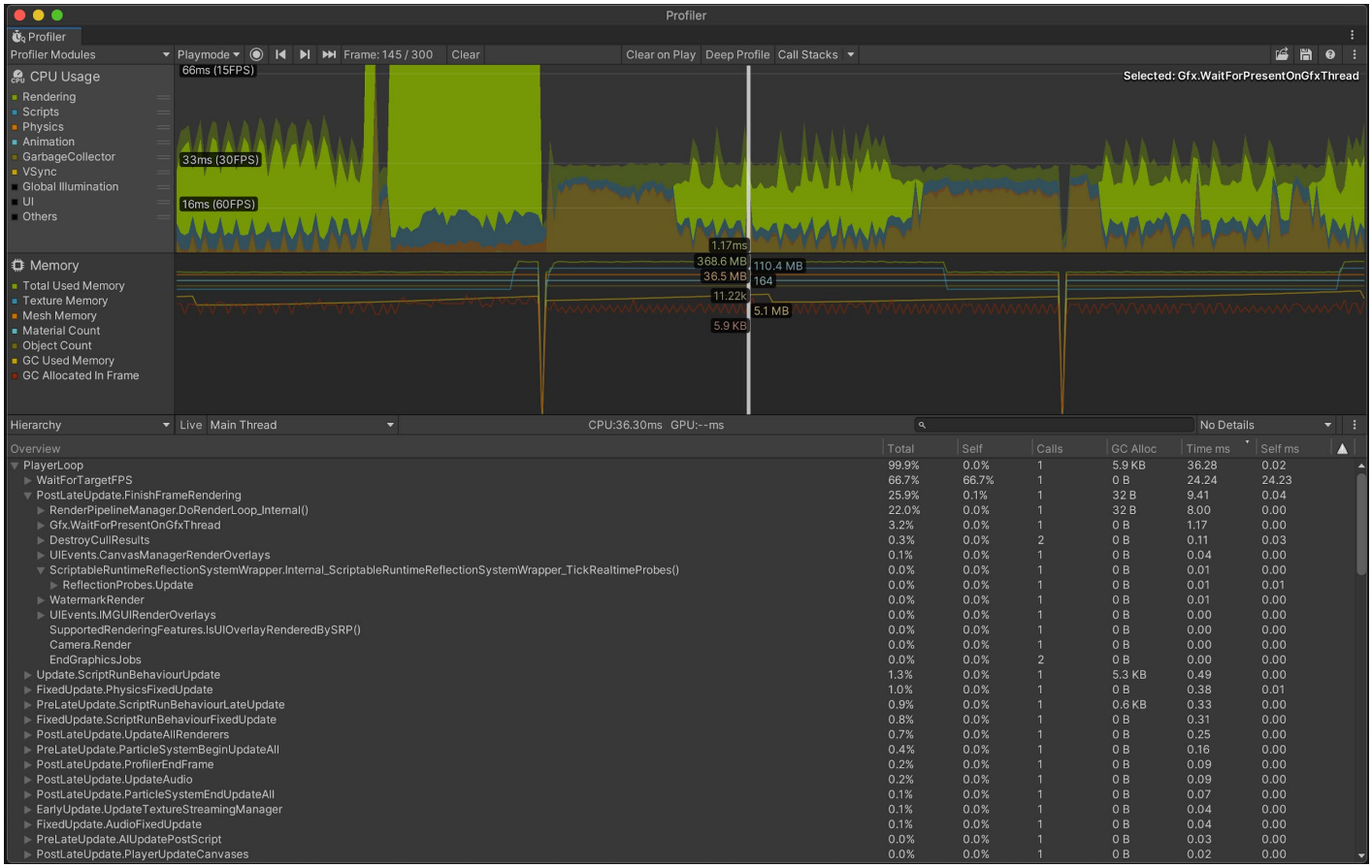
창을 클릭하여 특정 프레임을 분석한 다음, **Timeline** 또는 **계층 구조(Hierarchy)** 뷰를 사용하세요.

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

  — CPU 프레임 시간이 GPU 프레임 시간보다 훨씬 많다면 게임은 CPU 바운드입니다. 이는 CPU가 게임 로직, 물리, 기타 계산을 처리하는 데 오랜 시간이 걸리고, GPU는 CPU가 작업을 완료하기를 기다린다는 것을 의미합니다.

  — 마찬가지로 GPU 프레임 시간이 CPU 프레임 시간보다 훨씬 많다면 게임은 GPU 바운드입니다. GPU가 그래픽스를 렌더링하는 데 시간이 오래 걸리고 CPU가 GPU의 렌더링 완료를 기다린다는 의미입니다.

— Timeline 계층 구조에는 그룹화된 ProfileMarkers의 계층 구조가 표시됩니다. 이를 통해 밀리초 단위(**Time ms** 및 **Self ms**)의 시간 비용을 기준으로 샘플을 정렬할 수 있고, 프레임에서 함수에 대한 호출 수 및 관리되는 힙 메모리(GC Alloc)의 양도 파악할 수 있습니다. Time ms 또는 Self ms로 정렬하면 자체적으로, 또는 호출하는 다른 함수에 의해 가장 많은 시간을 소요하는 함수를 파악할 수 있습니다. 그러면 가장 큰 성능 향상이 기대되는 영역을 고려하여 더 집중적으로 최적화할 수 있습니다.



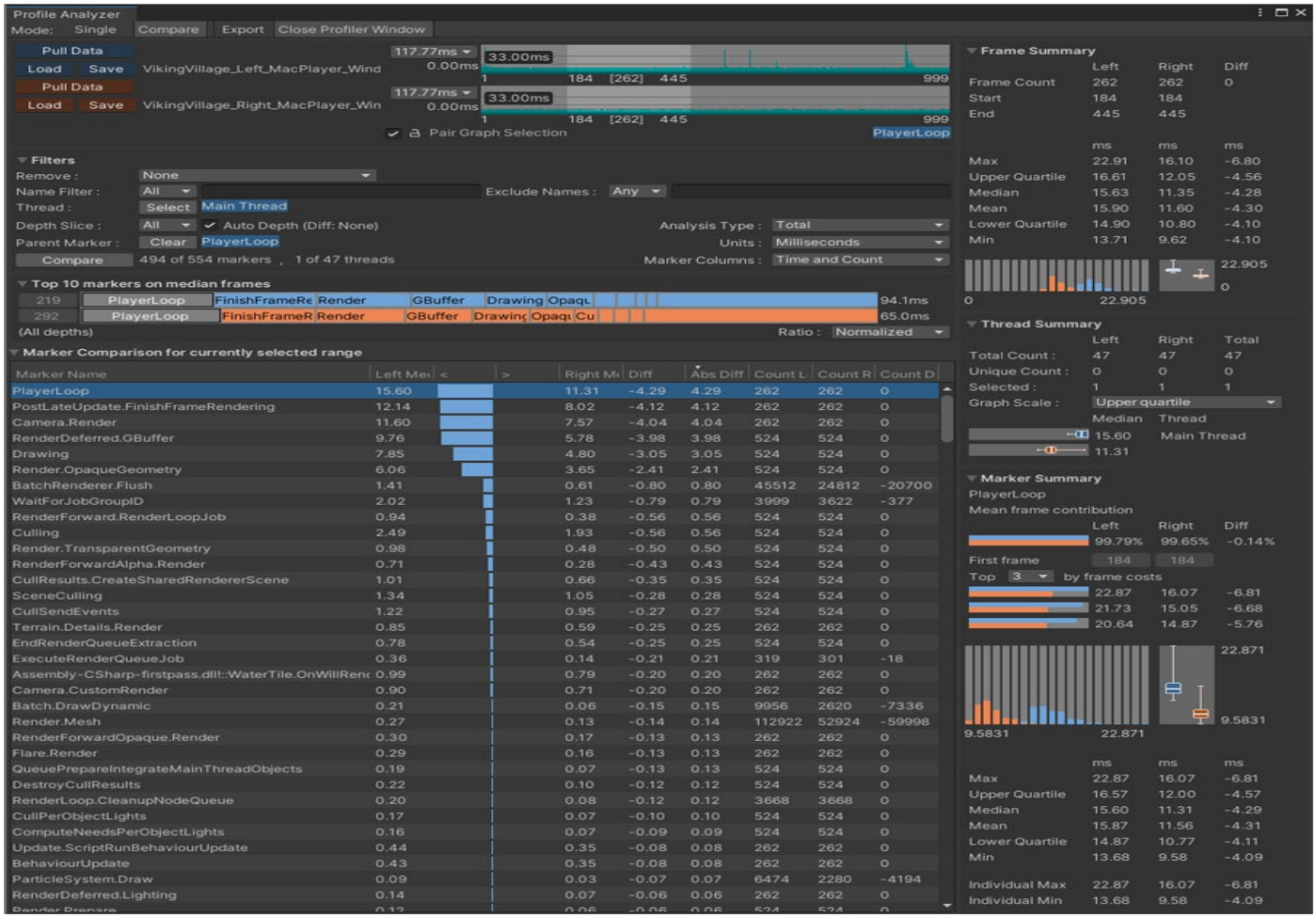
계층 구조 뷰를 통해 시간 비용에 따라 ProfileMarkers 정렬

여기에서 Unity 프로파일러의 전체 개요를 확인할 수 있습니다. 프로파일링이 생소하다면 Unity 프로파일링 소개 영상을 시청해 보세요.

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. 그런 다음 이 과정을 반복합니다.

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 Analyzer 는 Unity의 패키지 관리자를 통해 사용할 수 있습니다. Profile Analyzer의 기능에 대해 자세히 알아보려면 이 Profile Analyzer 튜토리얼을 시청해 보세요.



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

프레임당 정해진 시간 예산으로 작업하기

각 프레임에는 목표 초당 프레임 수(fps)를 기반으로 시간 예산이 주어집니다. 애플리케이션이 30fps로 실행되려면 프레임 예산이 프레임당 33.33ms(1,000ms/30fps)를 초과할 수 없습니다. 마찬가지로 60fps의 경우 목표 시간 예산은 프레임당 16.66ms(1,000ms/60fps)입니다.

VR(가상 현실) 앱을 개발할 때는 원활하고 몰입감 있는 경험을 제공하고 멀미를 방지하기 위해 높은 프레임 속도를 안정적으로 유지하는 것이 더욱 중요합니다. VR 애플리케이션의 일반적인 목표는 90fps로, 이때 시간 예산은 프레임당 11.11ms(1,000ms/90fps)로 제한적입니다. 요구 사항이 높은 이유는 VR의 경우 프레임당

각 눈마다 한 번씩 총 두 번을 렌더링해야 하며, 타이밍이 조금만 어긋나도 사용자가 쉽게 인지할 수 있기 때문입니다.

브라우저의 효율성과 하드웨어 성능에 따라 성능이 크게 달라지는 Unity Web 빌드에서도 일관되고 높은 프레임 속도가 중요합니다. 제한적인 프레임당 시간 예산 역시 중요한 요소입니다. 예를 들어 Unity WebGL 빌드에서 60fps가 목표라면 프레임당 16.66ms 안에 모든 작업이 처리되어야 합니다. 이 예산에는 렌더링, 물리 계산, 게임 로직의 모든 요소가 포함되며, 따라서 핵심은 애플리케이션의 모든 부분을 최적화하는 것입니다. 에셋을 효율적으로 관리하고, 씬의 복잡도를 줄이고, 셰이더와 스크립트를 최적화하는 등의 모든 단계가 애플리케이션의 목표 성능을 충족하는 데 필요합니다.

Unity가 브라우저에서 코드를 컴파일하고 실행하기 위해 사용하는 WASM(WebAssembly) 성능의 영향도 중요하게 고려해야 합니다. WASM은 기존 JavaScript보다 훨씬 성능이 우수하지만, 프레임 시간을 최대한 활용할 수 있으려면 여전히 코드를 프로파일링하고 최적화해야 합니다.

기기 온도 고려하기

그러나 모바일에서는 기기가 과열되거나 OS가 CPU 및 GPU에 서멀 스로틀링(thermal throttling)을 걸 수 있으므로 이러한 최대 시간을 일관되게 사용하는 것은 일반적으로 권장하지 않습니다. 일반적으로 프레임 간 쿨다운을 위해 가용 시간의 약 65%만 사용하는 원칙이 권장됩니다. 일반적인 프레임 예산은 30fps에서는 프레임당 약 22ms, 60fps에서는 11ms입니다.

기기에서 컷씬이나 로딩 시퀀스 같은 짧은 기간이라면 예산을 초과해도 되지만 긴 시간 동안 초과할 수는 없습니다.

대부분의 모바일 기기에는 데스크톱과 달리 냉각 장치가 없습니다. 물리적인 열기가 성능에 직접적인 영향을 미칠 수 있습니다.

기기가 과열되는 경우 프로파일러가 성능 저하를 인식하여 보고할 수 있습니다. 장기적으로 우려할 만한 요소가 아닌 경우에도 마찬가지입니다. 프로파일링 시 과열을 방지하려면 짧은 간격으로 프로파일링하세요. 이렇게 하면 기기의 온도가 낮아지고 실제 상황을 시뮬레이션할 수 있습니다. 일반적으로 기기를 10~15분 동안 식힌 다음 다시 프로파일링할 것을 권장합니다.

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 렌더링하는 데 너무 오래 걸려 렌더링 성능 문제가 발생한다면 CPU 바운드 게임이라고 합니다. GPU가 프레임을 렌더링하는 데 너무 오래 걸려 렌더링 성능 문제가 발생한다면 GPU 바운드 게임이라고 합니다.

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.WaitForPresent**가 빈번하게 발생한다면 메인 스레드는 준비되었으나 GPU가 프레임을 표시할 때까지 대기하는 것입니다.

최저 사양 및 최고 사양 기기 모두에서 테스트

현재 시중에서는 매우 다양한 사양의 iOS 및 Android 기기를 볼 수 있습니다. 애플리케이션에서 지원하려는 최저 사양과 최고 사양의 기기에서 가급적 반드시 프로젝트를 테스트해 보시기 바랍니다.

XR, 웹, 모바일 게임의 메모리 관리

원활한 성능을 보장하려면 효과적인 메모리 관리가 매우 중요합니다. Unity는 작은 임시 데이터를 스택에 할당하고 대용량 장기 데이터는 관리되는 힙이나 네이티브 힙에 할당하여 스크립트와 사용자 생성 코드의 자동 메모리 관리를 처리합니다. 그러나 XR, 웹, 모바일 애플리케이션에서 메모리를 비효율적으로 관리하면 프레임 속도 저하, 로드 시간 증가, 심지어 애플리케이션 크래시와 같은 성능 문제가 발생할 수 있으므로 더 신중하게 메모리 사용량을 관리해야 합니다. 이 섹션에서는 이러한 플랫폼 전반에서 메모리 사용량을 최적화하여 반응성이 뛰어나고 안정적인 애플리케이션을 제작하는 전략을 살펴봅니다.

효율적인 메모리 관리

여러 플랫폼에서 원활하고 반응성이 뛰어난 경험을 제공하려면 필수적으로 오브젝트 라이프사이클을 신중하게 관리하고, 가비지 컬렉션 오버헤드를 최소화하며, 에셋 로딩 전략을 최적화해야 합니다.

오브젝트 라이프사이클 관리: 오브젝트의 생성과 파괴를 적절하게 관리하여 메모리 누수와 불필요한 리소스 사용을 방지하세요. 사용되지 않는 오브젝트는 **Destroy()**를 사용하여 제거하고 필요하지 않은 레퍼런스는 null로 설정하여 메모리를 확보하세요.

오브젝트 풀링: 총알, 적, UI 요소 등 자주 사용하는 오브젝트는 반복해서 생성하고 파괴하는 대신 재사용하세요. 오브젝트 풀을 구현하면 오브젝트 인스턴스화 및 파괴와 관련된 오버헤드를 크게 줄이고 메모리 리소스를 절약할 수 있습니다. Unity 프로젝트의 오브젝트 풀링 페이지에서 자세히 알아보세요.

가비지 컬렉션의 영향 줄이기: 할당을 최소화하여 성능 저하를 초래할 수 있는 가비지 컬렉션의 빈도를 줄이고 영향을 완화하세요. 가능할 때마다 배열과 리스트를 미리 할당하는 방법으로 업데이트 루프에서 잦은 할당을 방지할 수 있습니다. 할 수 있다면 레퍼런스 타입(클래스) 대신 값 타입(구조체)를 사용하세요. 구조체는 스택에 할당되며 가비지 컬렉션 오버헤드가 발생하지 않습니다.

에셋 로딩을 최적화하는 기법은 다음과 같습니다.

— **지연 로딩**: 실제로 리소스가 필요한 시점까지 로딩을 미룹니다. 이렇게 하면 초기 로드 시간을 줄이고 리소스를 효율적으로 관리할 수 있습니다.

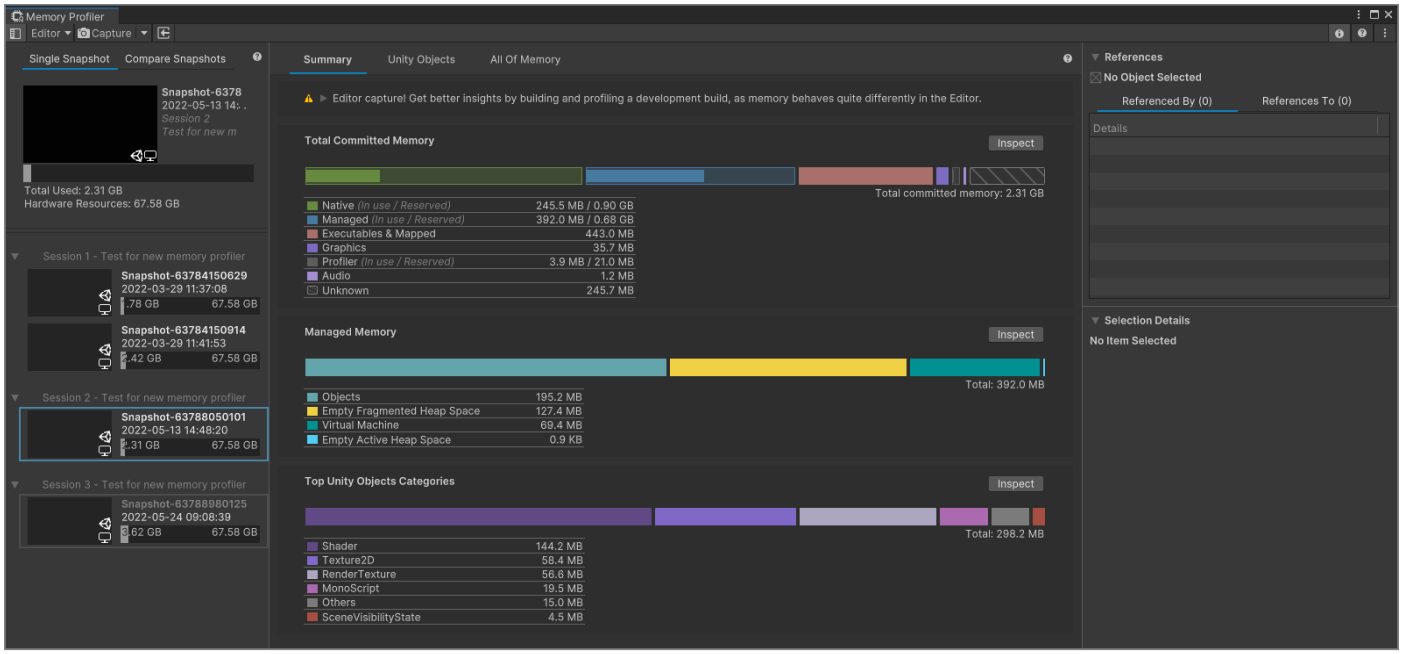
— **사용되지 않는 에셋 언로드**: Resources.UnloadUnusedAssets()를 사용하여 더 이상 필요하지 않은 에셋이 차지하는 메모리를 비웁니다.

— **어드레서블 에셋 시스템 사용**: 어드레서블 에셋 시스템(Addressable Asset System)을 사용하여 런타임에 에셋을 비동기식으로 관리합니다. 이 시스템은 웹 및 모바일 플랫폼에서 특히 유용하며 원격 에셋 호스팅, 동적 콘텐츠 업데이트, 지연 로딩을 지원합니다.

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 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 패키지 는 관리되는 힙 메모리의 스냅샷을 만들어 단편화 또는 메모리 누수와 같은 문제를 식별할 수 있습니다. Memory Profiler에 대해 알아보려면 이 Unity 동영상을 시청해 보세요.

Unity Objects 탭에서 중복된 메모리 항목을 제거할 부분을 식별하거나 가장 많은 메모리를 사용하는 오브젝트를 파악할 수 있습니다. **All of Memory** 탭에서 Unity가 스냅샷 내에서 추적하는 모든 메모리에 대한 요약 정보를 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.

힙을 불필요하게 할당하면 GC 스파이크를 유발할 수 있으므로 유의해야 합니다.

— **문자열**: C#에서 문자열은 값 유형이 아닌 레퍼런스 유형입니다. 문자열을 대규모로 사용하는 경우에는 불필요한 생성이나 조작을 줄이세요. JSON, XML 같은 문자열 기반 데이터 파일은 파싱하지 않는 것이 좋습니다. 대신 데이터를 ScriptableObjects에 저장하거나 MessagePack 또는 Protobuf 같은 포맷으로 저장하세요. 런타임에 문자열을 빌드해야 하는 경우 StringBuilder 클래스를 사용합니다.

— **Unity 함수 호출**: 힙 할당을 생성하는 함수도 있습니다. 레퍼런스를 루프 도중에 할당하지 말고 배열에 캐싱하세요. 가비지 생성을 방지하는 함수도 활용하세요. 예를 들어 문자열을 GameObject.tag와 직접 비교하는 대신 GameObject.CompareTag를 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 타입 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 정규식**: 두 가지 모두 백그라운드 박싱을 통해 가비지를 생성합니다. 성능이 문제가 된다면 LINQ와 정규식을 사용하지 마세요. 새로운 배열을 만드는 대신 for 루프와 리스트를 사용하세요.

자세한 내용은 매뉴얼의 가비지 컬렉션 베스트 프랙티스 페이지를 참고하세요.

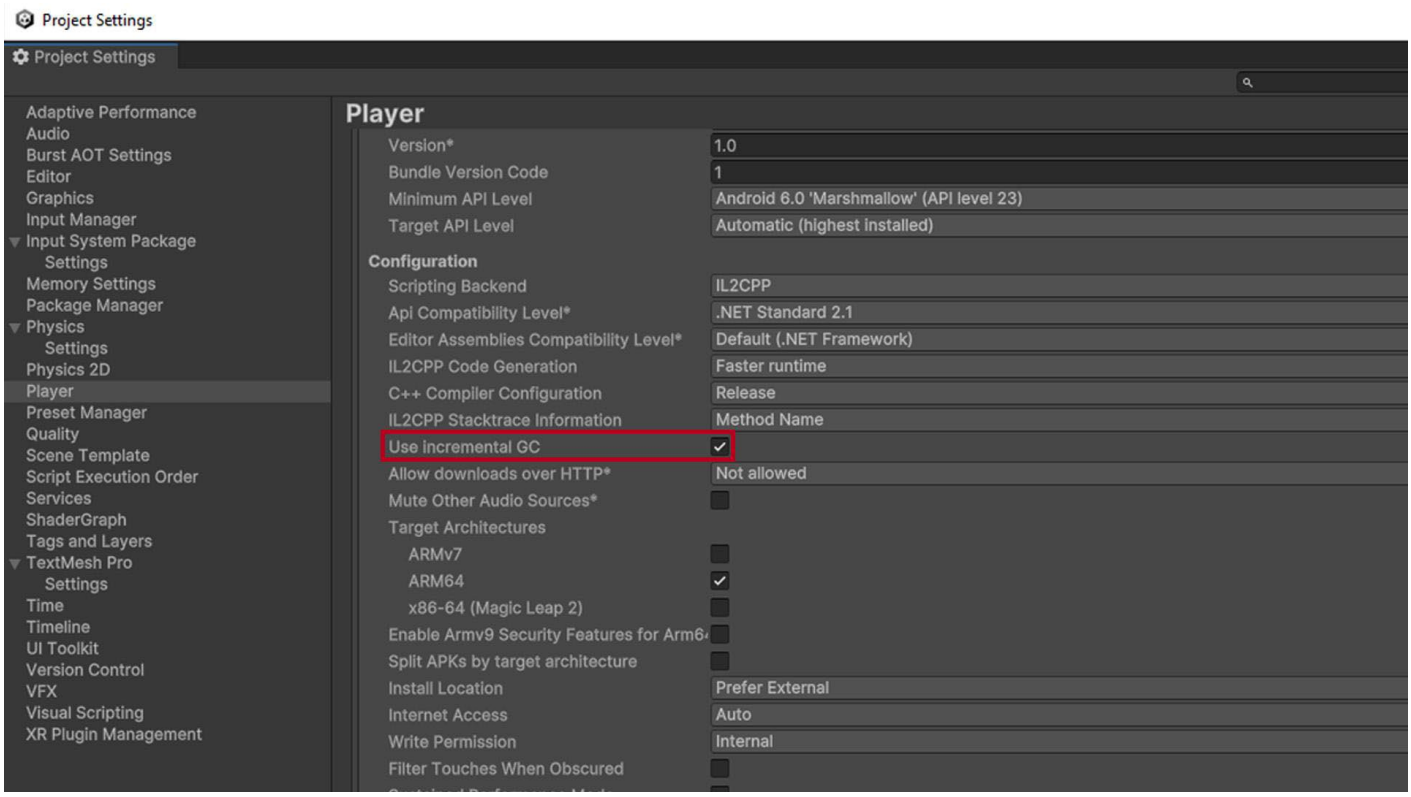
가능한 경우 가비지 컬렉션 시간 측정하기

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.

점진적 가비지 컬렉터를 활용하여 GC 워크로드 분할하기

점진적 가비지 컬렉션은 프로그램을 실행하는 동안 길게 1회 중단하는 것이 아닌, 여러 프레임에 걸친 워크로드에서 여러 번 발생하는 훨씬 짧은 중단을 사용합니다. 가비지 컬렉션이 성능에 영향을 미친다면 이 옵션을 사용하여 GC 스파이크를 줄일 수 있는지 확인해 보세요. Profile Analyzer를 사용하여 이 방식이 애플리케이션에 도움이 되는지 확인하세요.



점진적 가비지 컬렉터를 사용하여 GC 스파이크 줄이기

Adaptive Performance

Unity와 삼성의 Adaptive Performance를 사용하면 기기의 온도와 전력 상태를 모니터링하며 적절하게 대응할 준비를 갖출 수 있습니다. 예를 들어 사용자가 게임을 장시간 플레이하면 디테일 수준(LOD) 바이어스를 동적으로 낮춰서 게임을 계속 원활하게 실행할 수 있습니다. Adaptive Performance를 활용하면 개발자가 그래픽스 정확도를 유지하면서 통제된 방식으로 성능을 향상할 수 있습니다.

Adaptive Performance API를 사용하여 애플리케이션을 세밀하게 조정할 수 있으며, Adaptive Performance는 자동화 모드도 제공합니다. 자동화 모드에서는 다음을 포함한 주요 지표를 기반으로 하여 Adaptive Performance가 게임 설정을 조정합니다.

— 이전 프레임을 기준으로 이상적인 프레임 속도

— 기기 온도 수준

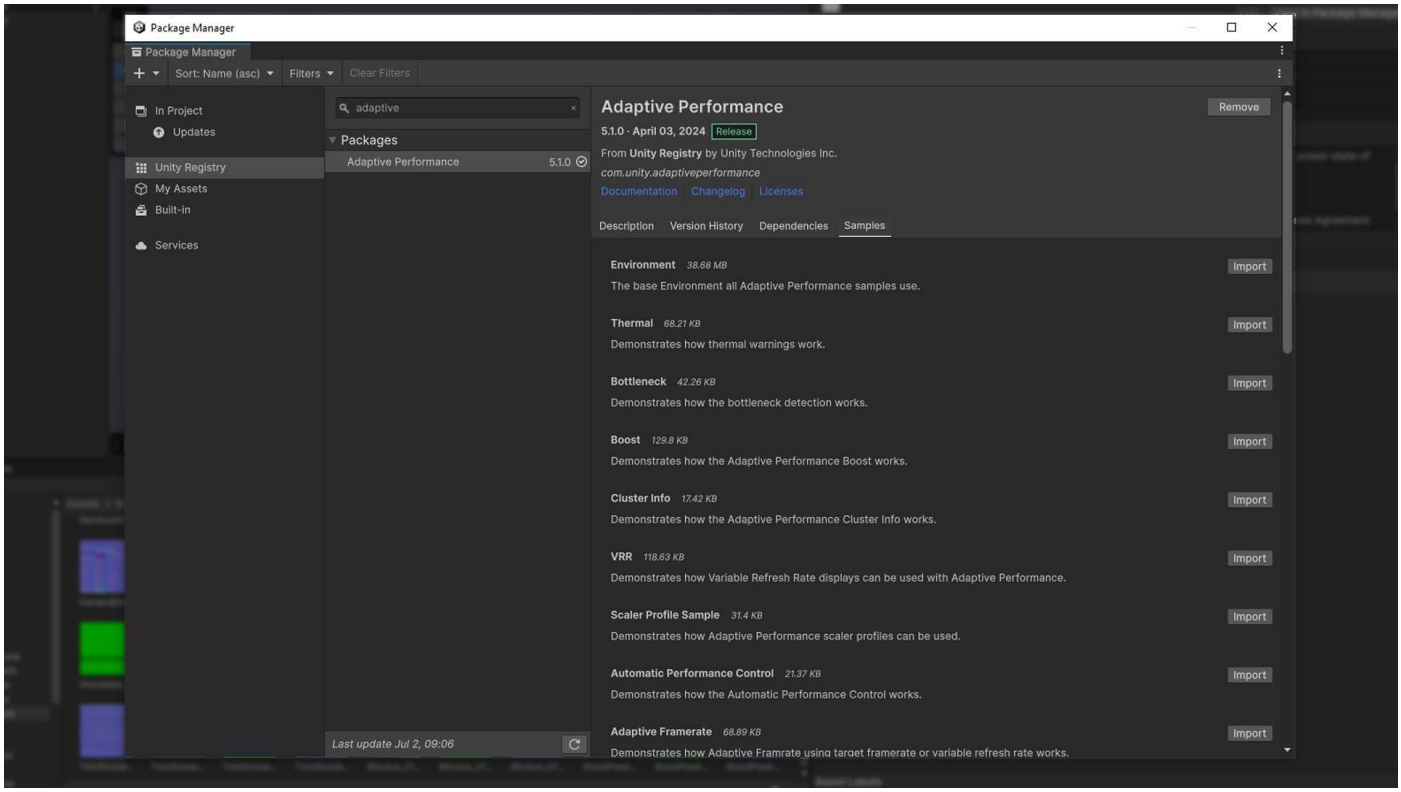
— 기기가 발열 관련 이벤트에 근접한 정도

— 기기의 CPU 또는 GPU 의존도

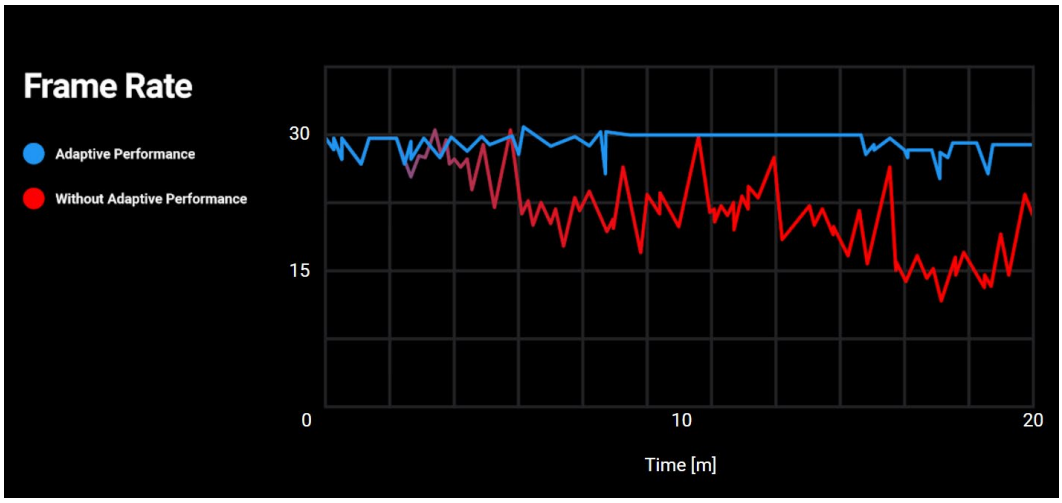
위 4가지 지표가 기기의 상태를 규정하면 Adaptive Performance가 설정을 미세 조정하고 병목 현상을 완화합니다. 이 작업은 기기의 상태를 설명하는 정수 값인 인덱서(Indexer)를 제공하는 방식으로 이루어집니다.

Adaptive Performance에 대해 자세히 알아보려면 **Package Manager > Adaptive Performance > Samples를 선택하여 패키지 관리자에서 제공되는** 샘플을 확인하세요. 각 샘플은 특정 스케일러와 상호작용하므로 해당 스케일러가 게임에 어떤 영향을 미치는지 확인할 수 있습니다. Adaptive Performance 설정 및 API와 직접 상호작용할 수 있는 방법을 자세히 알아보려면 최종 사용자를 위한 기술 자료를 확인하는 것도 좋습니다.

참고: Adaptive Performance는 삼성 기기에서만 사용 가능합니다.



Adaptive Performance 패키지



에셋

에셋 파이프라인이 제대로 최적화되면 로딩 시간을 단축하고, 메모리 사용량을 줄이고, 런타임 성능을 향상할 수 있습니다. 숙련된 테크니컬 아티스트와 협력하면 에셋 형식, 사양, 임포트 설정을 정의하여 효율적이고 간소화된 워크플로를 성공적으로 구축할 수 있습니다.

기본 설정에만 의존하지 마세요. 플랫폼별 오버라이드 탭을 활용하여 텍스처, 메시 지오메트리, 오디오 파일과 같은 에셋을 최적화하세요. 설정이 잘못되면 빌드 크기가 커지고 빌드 시간이 길어지며 메모리 사용 상황이 열악해질 수 있습니다.

프리셋을 사용하여 프로젝트 요구 사항에 맞는 기준 설정을 정의해 보세요. 이렇게 사전 예방적으로 접근하면 에셋을 처음부터 최적화하여 모든 플랫폼에서 성능을 향상하고 더 일관된 경험을 제공할 수 있습니다.

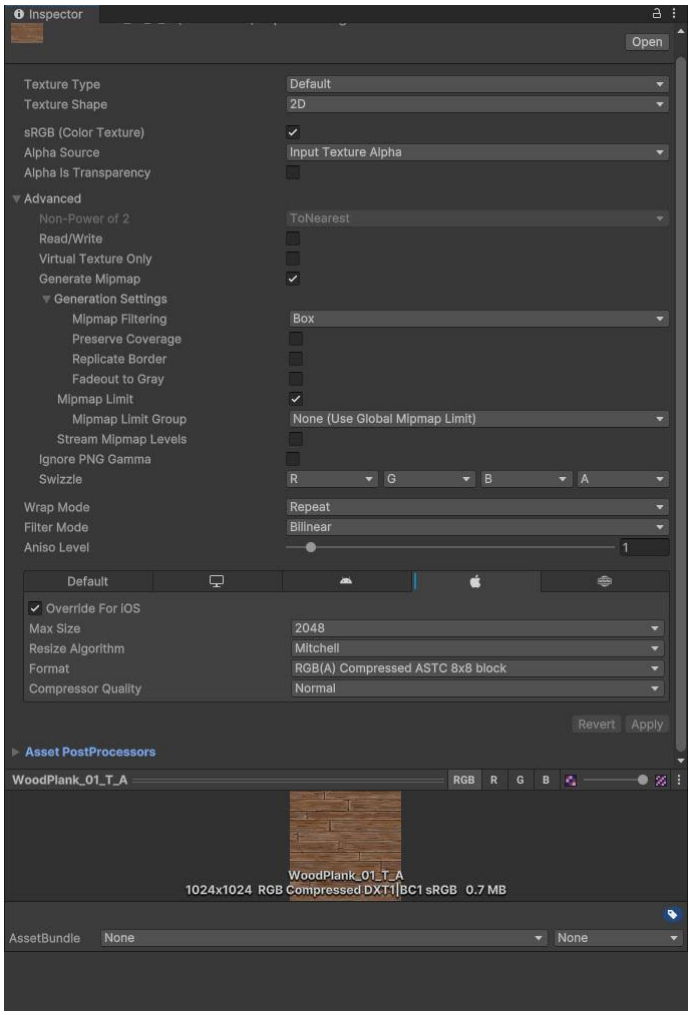
자세히 알아보려면 아트 에셋 작업 베스트 프랙티스를 참고하거나 Unity Learn의 모바일 애플리케이션용 3D 아트 최적화 교육 과정을 살펴보세요. Unity 웹 빌드, 모바일 및 XR 애플리케이션용 에셋 최적화에 대해 정확한 정보를 기반으로 의사 결정을 내릴 수 있도록 귀중한 분석 자료를 제공하는 리소스입니다.

올바른 텍스처 임포트

텍스처가 가장 많은 양의 메모리를 점유할 때가 많기 때문에 임포트 설정은 매우 중요합니다. 다음 가이드라인에 따라 텍스처를 최적화해 보세요.

— **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 모바일 텍스처 압축 포맷(PVRCT 또는 ETC)을 사용하려면 POT 텍스처 차원이 필요합니다.



빌드 크기를 최적화하는 데 도움이 되는 적절한 텍스처 임포트 설정

— **텍스처 아틀라싱**: 아틀라싱은 여러 개의 작은 텍스처를 균일한 크기의 더 큰 텍스처 하나로 그룹화하는 프로세스입니다. 여러 텍스처를 하나의 텍스처로 배치하면 드로우 콜을 줄이고 렌더링 속도를 높일 수 있습니다. Unity Sprite Atlas 또는 타사의 TexturePacker를 사용하여 텍스처의 아틀라스를 만들어 보세요.

— **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 유지합니다. 런타임에서 텍스처를 생성할 경우 **Texture2D.Apply** 함수를 사용하고 **makeNoLongerReadable** 값은 **true**로 전달합니다.

— **불필요한 밉맵 비활성화**: 밉맵은 카메라로부터의 거리에 따라 렌더링해야 하는 디테일 수준을 낮춰 성능을 최적화할 수 있지만, 항상 필요한 것은 아닙니다. 2D 스프라이트와 UI 그래픽스처럼 화면상에 일정한 크기로 유지되는 텍스처에서는 사용하지 않아도 됩니다. 카메라와의 거리가 다양한 3D 모델에 대해서는 밉맵을 계속 활성화하세요.

텍스처 압축

동일한 모델과 텍스처를 사용하는 두 가지 예를 생각해 보겠습니다. 왼쪽의 설정은 오른쪽에 비해 거의 26배 많은 메모리를 사용하지만, 비주얼 품질에서 큰 차이를 보이지 않습니다.

압축되지 않은 텍스처에는 더 많은 메모리가 필요합니다.

모바일, XR 및 웹에서 ASTC(Adaptive Scalable Texture Compression)를 사용하세요. 현재 개발 중인 게임은 대부분 ATSC 압축을 지원하는 최저 사양 기기를 타게팅하는 경향이 있습니다.

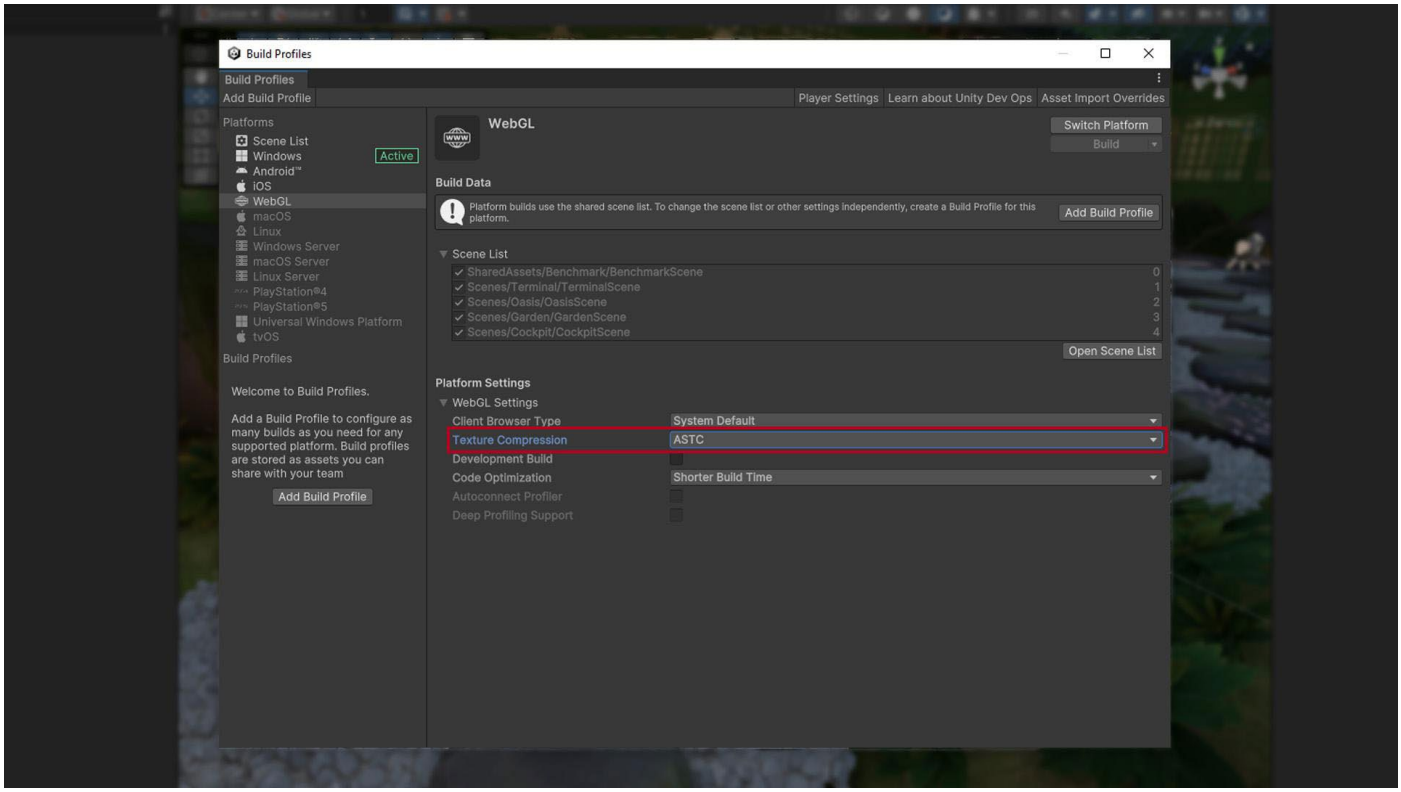
유일한 예외는 다음과 같습니다.

— A7 또는 그 이전 사양의 기기(예: iPhone 5, 5S 등)를 타게팅하는 iOS 게임 - PVRTC 사용

— 2016년 이전에 출시된 기기를 타게팅하는 Android 게임 - ETC2 (Ericsson Texture Compression) 사용

PVRTC 및 ETC 같은 압축 형식이 충분히 고품질이 아니고 타겟 플랫폼에서 ASTC를 완전히 지원하지 않는다면 32비트 텍스처 대신 16비트 텍스처를 사용해 보세요.

플랫폼별 권장 텍스처 압축 포맷 매뉴얼에서 자세한 내용을 확인하세요.



빌드 설정에서 텍스처 압축 방법으로 ASTC 선택

메시 임포트 설정

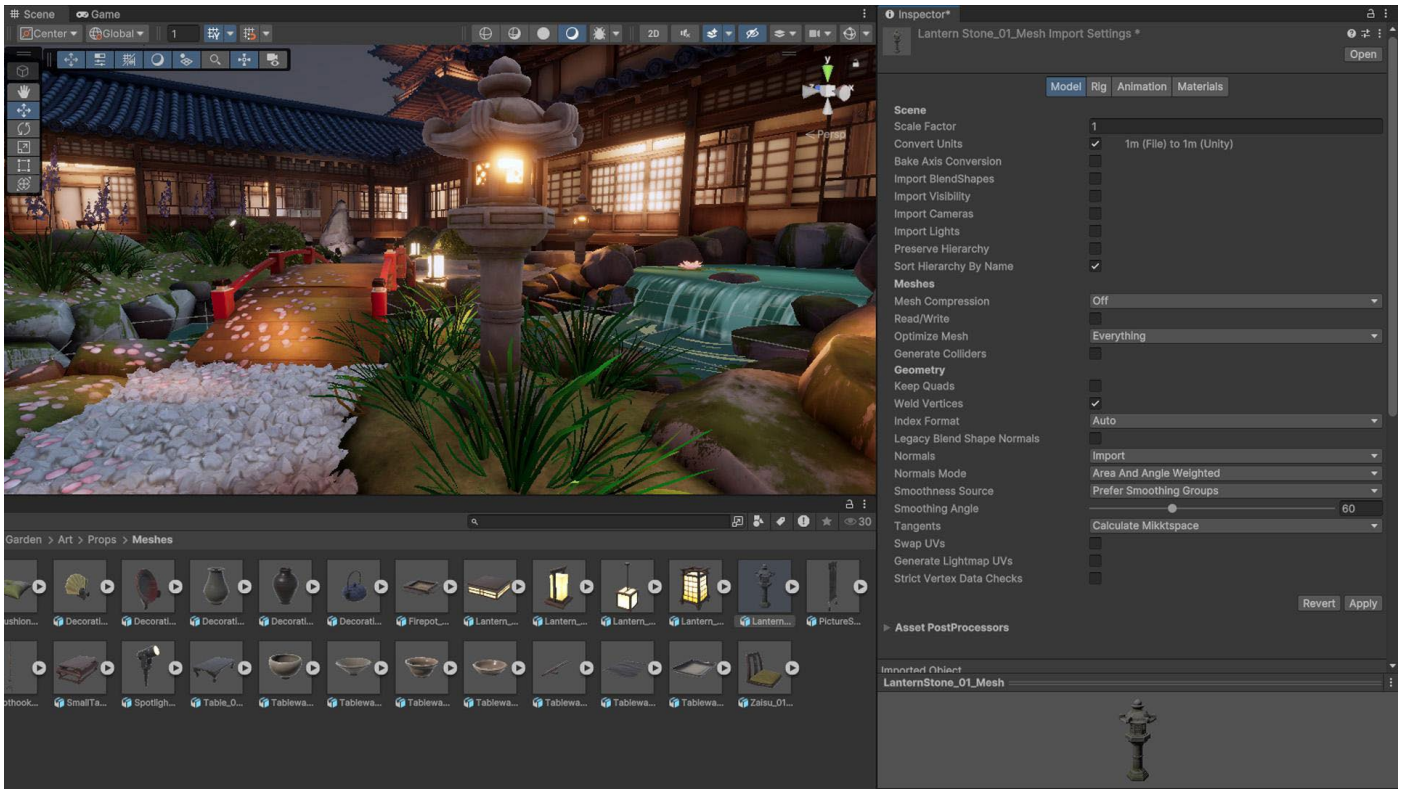
텍스처와 마찬가지로 메시도 메모리를 많이 사용하므로 최적의 임포트 설정을 선택해야 합니다. 메시의 메모리 사용량을 줄이는 방법은 다음과 같습니다.

— **메시 압축:** 적극적인 압축을 통해 디스크 공간을 줄일 수 있습니다(런타임 시 메모리에는 영향을 주지 않음). 메시 양자화(quantization)로 인해 정확성이 떨어질 수 있으므로 다양한 수준으로 압축해 보면서 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.

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

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



메시 임포트 설정 확인

폴리곤 개수 확인

해상도가 높은 모델은 메모리 사용량이 더 많고 잠재적으로 GPU 시간이 더 길 수 있습니다. 하지만 배경 지오메트리에 폴리곤이 50만 개씩 필요한 경우는 거의 없을 것입니다. DCC 패키지에서 모델 수를 줄일 수 있다면 줄이는 것이 좋습니다. 카메라 시야에 보이지 않는 폴리곤은 삭제하고, 고밀도 메시 대신 텍스처와 노멀 맵을 사용하여 정교한 디테일을 표현하세요.

AssetPostprocessor를 사용하여 임포트 설정 자동화

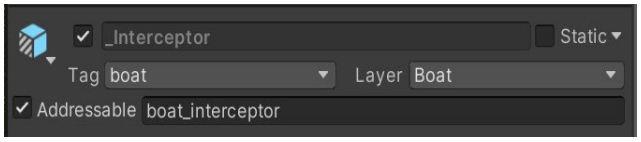
에셋을 임포트하기 전이나 임포트할 때 AssetPostprocessor로 임포트 파이프라인에 연결하여 스크립트를 실행할 수 있습니다. 그러면 프리셋과 유사하지만 코드를 통한 방식으로 모델, 텍스처, 오디오 등을 임포트하기 전후에 설정을 커스터마이즈하라는 메시지가 표시됩니다. 이 프로세스에 대해 자세히 알아보려면 GDC 2023 발표 '게임 개발의 모든 단계에서 활용할 수 있는 기술 팁'을 시청해 보세요.

Unity DataTools

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

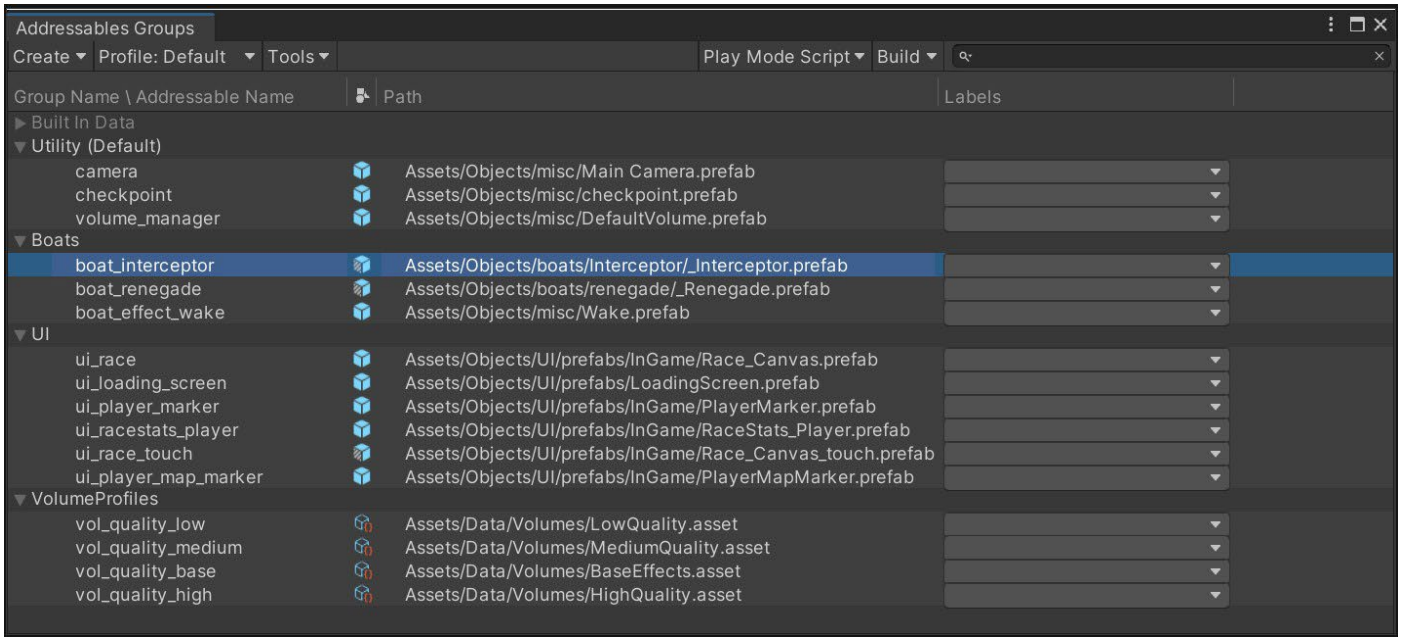
어드레서블 에셋 시스템 사용

어드레서블 에셋 시스템 를 활용해 콘텐츠를 간단하게 관리할 수 있습니다. 이 통합된 시스템은 로컬 경로 또는 원격 CDN(콘텐츠 전송 네트워크)에서 비동기적으로 '주소' 또는 별칭에 따라 AssetBundle을 로드합니다.



코드가 아닌 에셋(모델, 텍스처, 프리팹, 오디오, 전체 씬)을 AssetBundle로 나눌 경우 DLC(다운로드 가능한 콘텐츠)로 분리할 수 있습니다.

그런 다음 어드레서블을 사용하여 모바일 애플리케이션에 사용할 더 작은 초기 빌드를 만듭니다. 클라우드 콘텐츠 전송을 사용하면 게임 콘텐츠를 호스트하고 게임이 진행됨에 따라 플레이어에게 데이터를 전송할 수 있습니다.



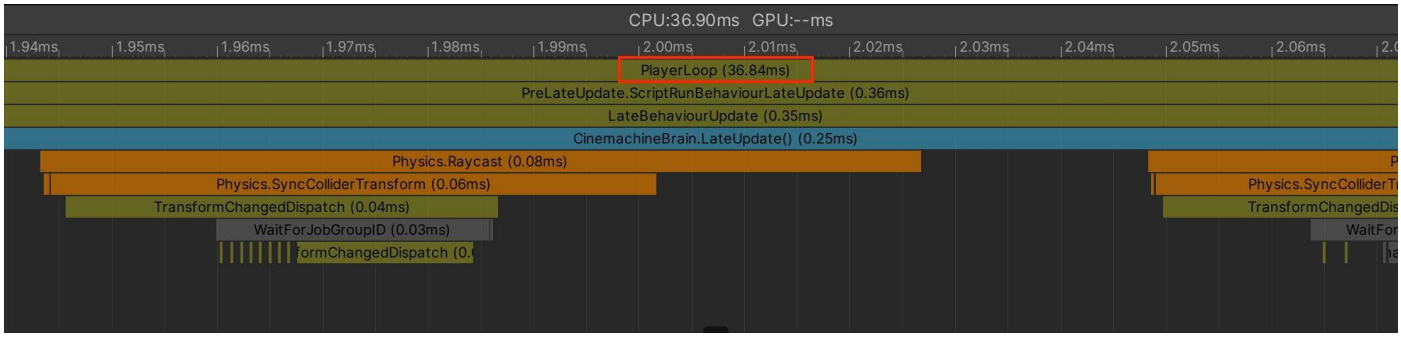
어드레서블 에셋 시스템을 사용하여 '주소'에 따라 에셋 로드

여기를 클릭하여 어드레서블 에셋 시스템으로 에셋 관리의 번거로움을 어떻게 완화하는지 알아보세요.

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).

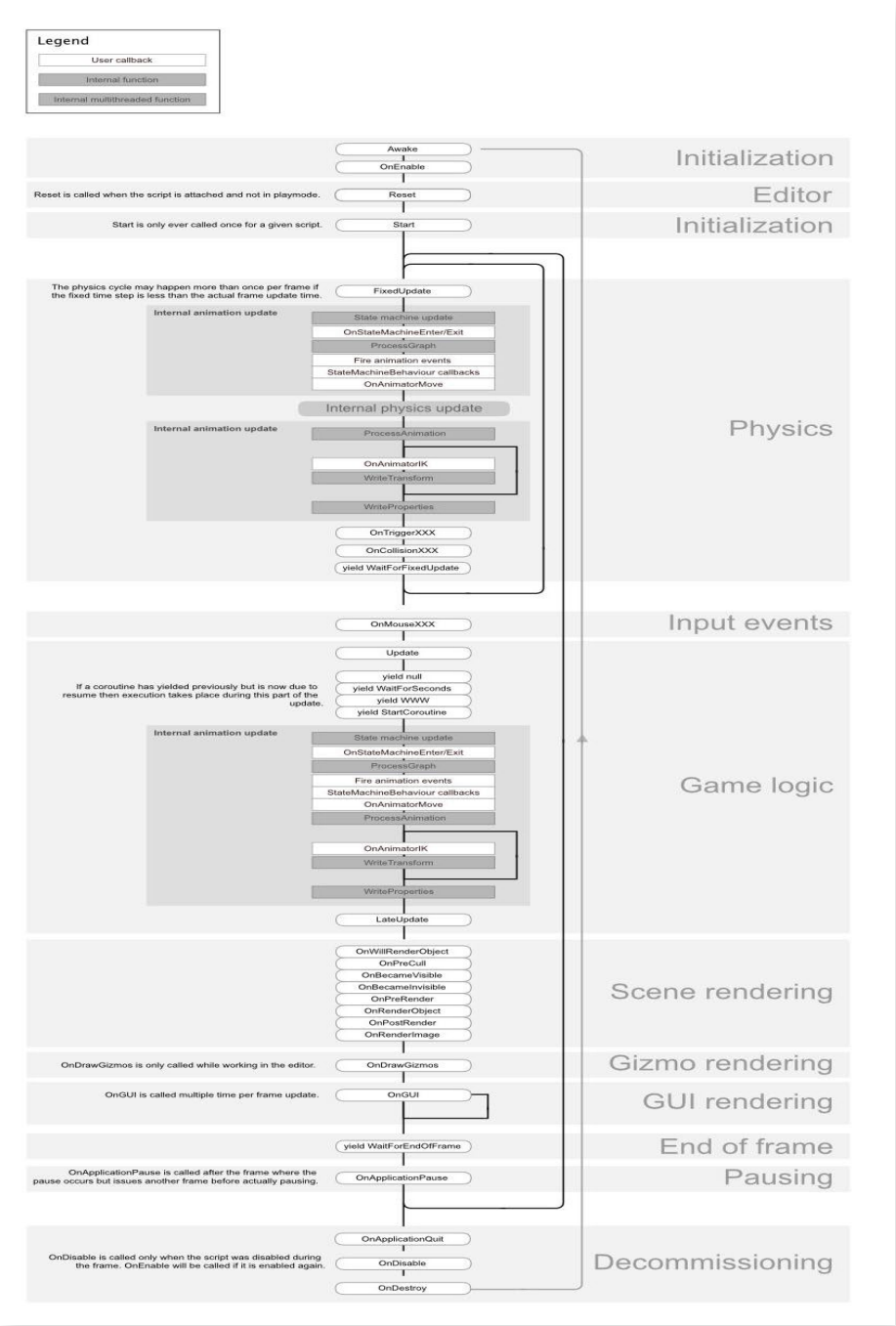


전체 엔진 실행의 컨텍스트에서 커스텀 스크립트, 설정, 그래픽스를 보여 주는 프로파일러

스크립트를 최적화하는 유용한 팁을 소개합니다.

Unity PlayerLoop 이해하기

Unity 프레임 루프의 실행 순서 를 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**, **FixedUpdate**에서 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 나옵니다.

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.

자세히 알아보려면 이벤트 함수의 실행 순서를 참고하세요.

빈 Unity 이벤트 방지

빈 MonoBehaviour도 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()
{
}
#endif
```

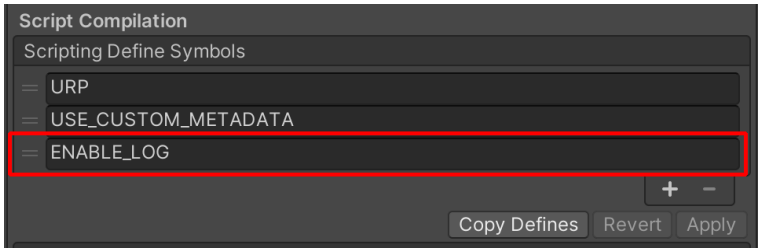
빌드에 불필요한 오버헤드가 유입되는 일 없이 에디터에서 **Update**를 자유롭게 사용하여 테스트할 수 있습니다.

Debug Log 구문 제거하기

Log 구문, 특히 **Update**, **LateUpdate** 또는 **FixedUpdate**에 있는 Log 구문으로 인해 성능이 저하될 수 있습니다. 빌드를 만들기 전에 **Log** 구문을 비활성화하세요.

더 쉽게 Log 구문을 비활성화하려면 전처리 지시문과 함께 조건부 속성 을 만들면 됩니다. 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**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 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.StringToHash 를, 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 반복(iteration)되므로 효율성에 영향을 줍니다. 컬렉션에 List, Array 또는 Dictionary 중 무엇을 사용해야 할지 고민하시나요? 올바른 구조를 선택하기 위한 일반적인 가이드인 C#의 데이터 구조 MSDN 가이드 를 참고하세요.

런타임 시 컴포넌트 추가 방지

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

이미 설정된 원하는 컴포넌트로 프리팹을 인스턴스화 하는 것이 일반적으로 더 효과적입니다.

게임 오브젝트 및 컴포넌트 캐싱

Update 메서드에서 호출하지 않도록 Awake 또는 Start에서 레퍼런스를 캐싱하는 것이 가장 좋습니다.

다음 예는 반복적인 **GetComponent** 호출의 비효율성을 보여 줍니다.

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

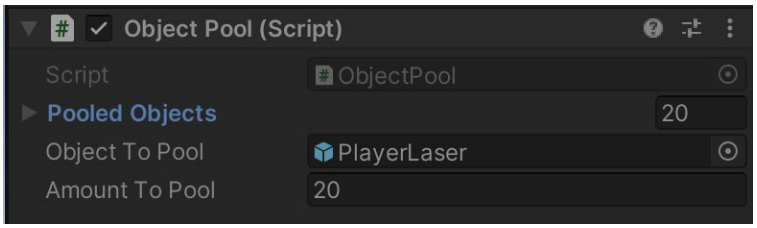
함수의 결과가 캐싱되므로 **GetComponent**는 한 번만 호출하는 것이 더 효율적입니다. **Update**에서 **GetComponent**를 추가로 호출하지 않아도 캐싱된 결과를 재사용할 수 있습니다.

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

Unity 2020.2 이전 버전에서는 **GameObject.Find**, **GameObject.GetComponent**, **Camera.main**에 많은 리소스가 필요했지만 이제는 그렇지 않습니다. 하지만 그렇다 해도 최선의 방법은 **Update** 메서드에서 호출하지 말고 결과를 캐싱하여 위 방법을 따르는 것입니다.

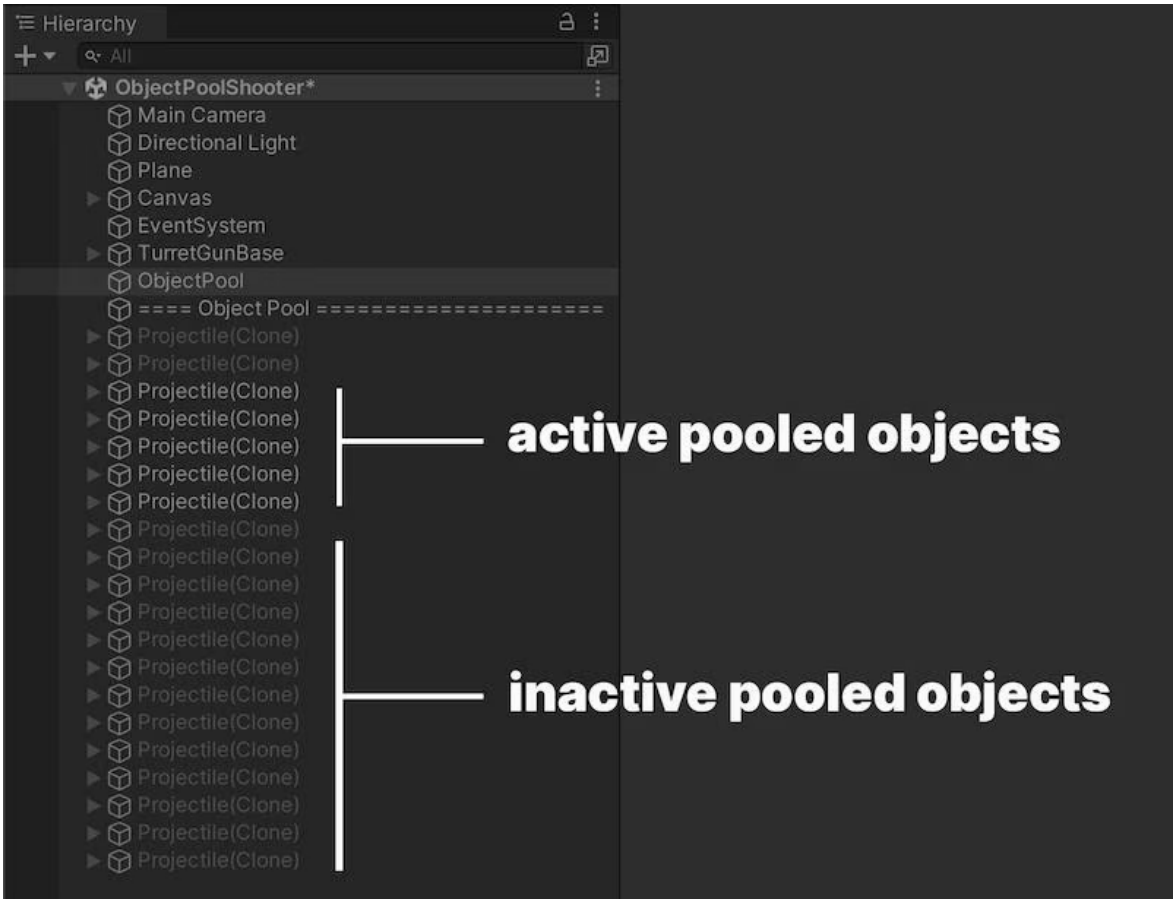
오브젝트 풀 사용하기

Instantiate 함수와 **Destroy** 함수는 가비지와 GC(가비지 컬렉션) 스파이크를 야기하여 일반적으로 프로세스를 느리게 만들 수 있습니다. 많은 수의 오브젝트를 인스턴스화해야 한다면 오브젝트 풀링 기법을 적용하세요.



재사용 가능한 PlayerLaser 인스턴스 20개를 생성하는 ObjectPool의 예

오브젝트 풀링의 베스트 프랙티스 중 하나는 CPU 스파이크가 눈에 띄지 않을 때(예: 메뉴 화면) 재사용 가능한 인스턴스를 생성하는 것입니다. 그런 다음 컬렉션으로 이러한 오브젝트 풀을 추적합니다. 게임플레이 중에는 필요할 때 다음으로 사용 가능한 인스턴스를 활성화했다가, 사용을 마치면 오브젝트를 파괴하는 대신 비활성화한 다음 풀로 돌려보내면 됩니다.



비활성 상태로 발사할 준비가 된 발사체 오브젝트 풀의 예시

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. Unity 에셋 스토어에서 다운로드할 수 있는 이 샘플 프로젝트에서 오브젝트 풀링 패턴을 비롯한 여러 시스템이 구현된 Unity 씬을 살펴볼 수도 있습니다.

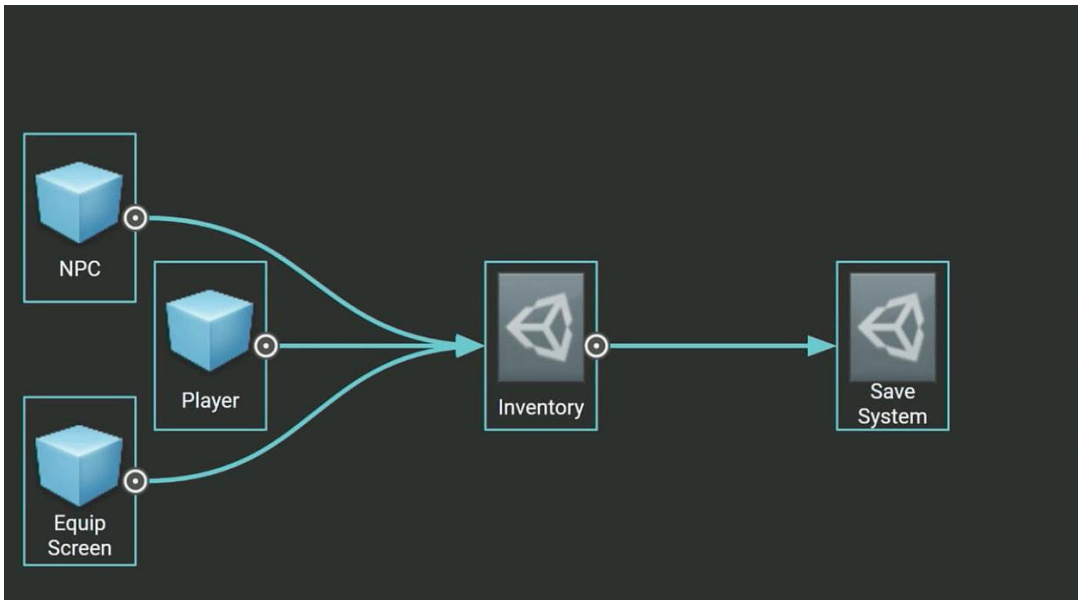
ScriptableObject 사용하기

정적 값 또는 설정은 MonoBehaviour가 아닌 **ScriptableObject**에 저장하세요. ScriptableObject는 한 번만 설정하면 되는 프로젝트 내부의 에셋입니다.

MonoBehaviour가 호스트 역할을 하려면 게임 오브젝트뿐만 아니라 기본적으로 트랜스폼이 필요하므로 더 큰 오버헤드를 유발합니다. 따라서 하나의 값을 저장하기 전에 사용되지 않은 다량의 데이터를 생성해야 합니다. ScriptableObject는 게임 오브젝트와 트랜스폼이 필요하지 않으므로 메모리 사용량을 줄일 수 있습니다. 또한 프로젝트 수준에서 데이터를 저장하므로 여러 씬에서 동일한 데이터에 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 ScriptableObject에 퍼널(funnel) 처리할 수 있습니다. 그런 다음 각 오브젝트가 데이터 자체를 복사하는 대신 공유 데이터 에셋에 대한 레퍼런스를 저장하면 됩니다. 이렇게 하면 수많은 오브젝트를 사용하는 프로젝트에서 성능을 크게 향상할 수 있습니다.

ScriptableObject에서 필드를 생성하여 값 또는 설정을 저장한 다음 MonoBehaviour에서 ScriptableObject를 참조하세요.



다양한 게임 오브젝트의 설정을 보관하는 Inventory ScriptableObject의 예시

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

소프트웨어 디자인에서는 이를 플라이웨이트(flyweight) 패턴 최적화라고 합니다. 이렇게 ScriptableObject를 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.

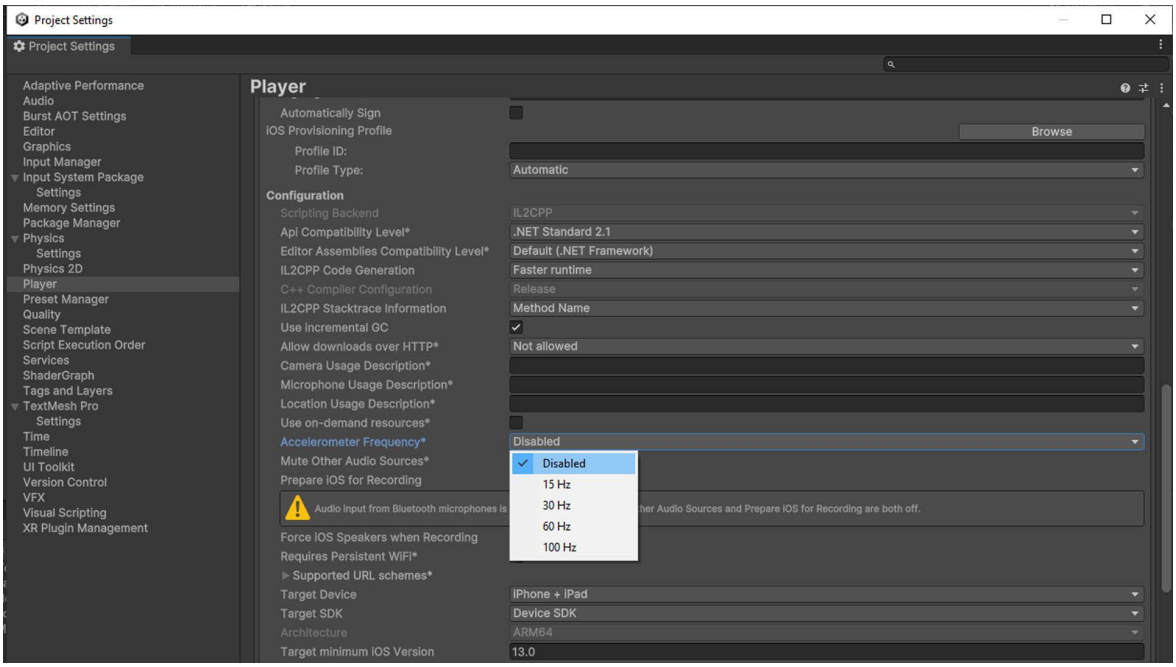
ScriptableObject 소개 개발 로그를 시청하고 ScriptableObject의 장점을 알아보세요. 여기의 Unity 기술 자료 및 ScriptableObject를 사용하여 Unity에서 모듈식 게임 아키텍처 구축하기를 참조해도 됩니다.

프로젝트 설정

프로젝트 설정 중 몇 가지는 모바일 성능에 영향을 줄 수 있습니다.

Accelerometer Frequency 감소 또는 비활성화

Unity는 모바일 기기의 가속 센서를 1초에도 몇 번씩 풀링합니다. 애플리케이션에서 가속 센서를 사용하지 않는다면 비활성화하거나 빈도를 줄여 전반적인 성능을 개선하세요.



모바일 게임에서 활용하지 않는다면 Accelerometer Frequency를 비활성화해야 합니다.

불필요한 플레이어 설정 또는 품질 설정 비활성화

Player 설정에서 지원되지 않는 플랫폼의 **Auto Graphics API**를 비활성화하면 셰이더 배리언트가 과도하게 생성되지 않도록 방지할 수 있습니다. 애플리케이션이 오래된 CPU를 지원하지 않는다면 해당 CPU에 대해 **Target Architectures**를 비활성화합니다.

Quality 설정에서 불필요한 품질 수준을 비활성화하세요.

불필요한 물리 비활성화

게임에서 물리를 사용하지 않는다면 **Auto Simulation**과 **Auto Sync Transforms**를 선택 해제합니다. 해당 기능을 선택하면 별다른 이득 없이 애플리케이션의 속도가 저하될 수 있습니다.

올바른 프레임 속도 선택

모바일 프로젝트에서는 프레임 속도와 배터리 수명, 서멀 스로틀링이 균형을 이루어야 합니다. 기기 한계치인 60fps로 실행하기보다 30fps 정도로 타협하는 것이 좋습니다. Unity의 모바일 기본 설정은 30fps입니다.

XR 플랫폼을 타게팅하는 경우, 프레임 속도가 더욱 중요합니다. 몰입감을 유지하고 멀미를 방지하려면 72fps, 90fps, 심지어 120fps까지도 프레임 속도가 필요한 경우가 많습니다. 프레임 속도가 높으면 원활하고 반응성이 뛰어난 경험을 보장할 수 있으며, 편안한 VR 환경을 조성하려면 이러한 요소가 매우 중요합니다. 그러나 XR 플랫폼, 특히 스탠드얼론 VR 헤드셋에서는 전력 소비와 발열 관리에 대한 어려움을 겪을 수 있습니다.

최적의 프레임 속도를 선택하려면 모바일 기기, 스탠드얼론 VR 헤드셋, AR 기기 등 타겟 플랫폼의 구체적인 요구 사항과 제약을 이해해야 합니다. 최적의 프레임 속도를 신중하게 채택하면 다양한 플랫폼에서 성능과 사용자 경험을 모두 최적화할 수 있습니다.

또한 **Application.targetFrameRate**를 활용해 런타임 중에 프레임 속도를 동적으로 조정할 수도 있습니다. 예를 들어 속도가 느리거나 비교적 정적인 씬의 경우 30fps 아래로 낮추고 게임플레이 중에는 더 높은 fps 설정을 유지할 수 있습니다.

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.

트랜스폼 한 번에 이동

트랜스폼을 이동할 때, Transform.SetPositionAndRotation을 사용하여 위치와 회전을 한 번에 업데이트하세요. 그러면 트랜스폼을 두 번 수정함으로 인해 발생하는 오버헤드를 방지할 수 있습니다.

런타임에 게임 오브젝트를 인스턴스화해야 한다면 다음과 같은 간단한 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.

XR, 웹, 모바일 개발에서의 Vsync

XR, 웹 및 모바일 플랫폼용 콘텐츠를 개발할 때는 Vsync(수직 동기화)가 활성화되어 있다고 가정해야 하며, 이는 Unity 에디터에서 Vsync를 비활성화(Project Settings > Quality)한 경우에도 마찬가지입니다. 이러한 플랫폼에서는 화면 찢김 현상을 방지하고 비주얼을 원활하게 출력할 수 있도록 하드웨어 수준에서 Vsync를 적용하는 경우가 많습니다. GPU가 디스플레이의 새로고침 속도에 맞춰 프레임을 매우 빠르게 렌더링하지 못하는 경우, 현재 프레임이 다시 표시되므로 실질적인 fps가 낮아집니다. 플랫폼별 작동 방식을 살펴보겠습니다.

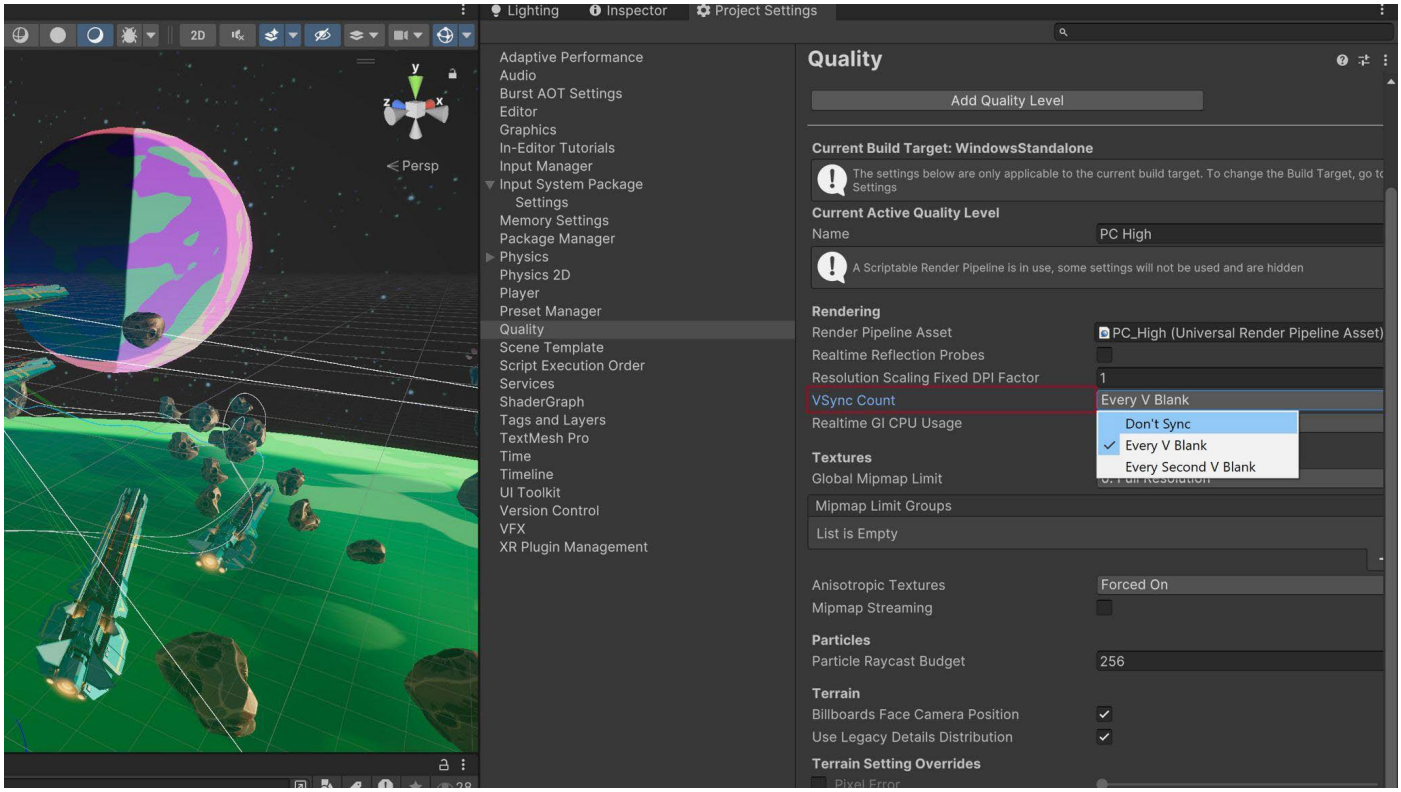
- 모바일 플랫폼: 모바일 기기는 일반적으로 디스플레이 새로고침 속도인 60Hz 또는 최신 기기의 경우 더 높은 속도로 Vsync를 적용합니다. 애플리케이션의 프레임 속도가 이 목표 아래로 떨어지면 기기는 이전 프레임을 유지하게 되어 눈에 띄는 끊김 현상이나 입력 지연이 발생합니다. 렌더링 성능을 최적화하여 프레임 속도를 안정적으로 유지해야 다양한 성능의 모바일 기기에서 원활한 실행을 보장할 수 있습니다.
- 웹 플랫폼: 웹 브라우저, 특히 Unity Web에서도 디스플레이 새로고침 속도와 동기화되도록 Vsync를 적용하는 경우가 많습니다. 브라우저 내에서 실행되며 추가되는 오버헤드를 고려했을 때, 눈에 띄는 성능 저하를 피하려면 애플리케이션을 최적화하여 일관된 프레임 속도를 유지하는 것이 중요합니다. 웹 플랫폼마다 성능이 다르므로 다양한 브라우저와 기기에서 테스트하세요.
- XR 플랫폼: XR 환경의 경우, 본질적으로 몰입형 경험을 제공해야 하므로 높은 프레임 속도를 안정적으로 유지하는 것이 더욱 중요합니다. 대부분의 XR 기기는 90Hz 이상에서 Vsync를 적용하며, 프레임 속도가 조금이라도 떨어지면 사용자가 불편함을 느끼거나 멀미를 겪을 수 있습니다. GPU가 이러한 높은 요구 사항을 지속적으로 충족하려면 렌더링에서 물리 계산까지 애플리케이션의 모든 부분을 최적화하는 것이 중요합니다.

XR, 웹 및 모바일 플랫폼에서 Vsync가 관리되는 방법을 이해하고 애플리케이션을 최적화하여 일관된 프레임 속도를 유지하면 다양한 플랫폼에서 원활하고 반응성이 뛰어난 경험을 제공하여 사용자의 기대치를 충족할 수 있습니다.

VSync Count

Unity Quality 설정의 **VSync Count** 설정은 프레임 렌더링이 디스플레이 새로고침 속도와 동기화되는 방법을 결정합니다. **Every V Blank**(VSync Count 1과 같음)로 설정하면 Unity가 프레임 렌더링을 각 수직 공백에 동기화하므로 실질적으로 프레임 속도가 디스플레이 새로고침 속도와 일치하게 됩니다(예: 60Hz = 60fps). 이렇게 하면 화면 찢김 현상을 방지하고 비주얼을 원활하게 출력할 수 있습니다.

또는 **Every Second V Blank**(VSync Count 2)로 설정하면 프레임 속도가 절반이 되며, 애플리케이션이 완전한 새로고침 속도 성능을 유지하지 못하는 상황에 유용할 수 있습니다. Vsync를 비활성화(Don't Sync)하면 FPS를 최대로 높일 수 있지만 화면 찢김 현상이 발생할 수 있습니다. 이렇게 설정하더라도 하드웨어 수준에서 Vsync가 적용되는 플랫폼도 있습니다.



Quality 설정 내 VSync Count

그래픽스 및 GPU 최적화

Unity는 프레임마다 렌더링해야 하는 오브젝트를 지정한 다음 드로우 콜을 만듭니다. 드로우 콜은 오브젝트(예: 삼각형)를 그리기 위한 그래픽스 API 호출이며, 배치는 함께 실행되는 드로우 콜의 그룹입니다.

프로젝트가 복잡해질수록 GPU의 워크로드를 최적화해 주는 파이프라인이 필요합니다. **URP(유니버설 렌더 파이프라인)**는 포워드, 포워드+, 디퍼드라는 세 가지 렌더링 옵션을 지원합니다.

포워드 렌더링은 싱글 패스에서 모든 조명을 계산하며 모바일 게임에서 일반적으로 권장되는 기본 설정입니다. Unity 2022 LTS에서 도입된 포워드+ 렌더링은 오브젝트 단위 대신 공간을 분류해 광원을 컬링하여 표준 포워드 렌더링보다 더 나은 품질을 보입니다. 이 방식을 사용하면 한 프레임을 렌더링할 때 전반적으로 훨씬 더 많은 광원을 활용할 수 있습니다. 디퍼드 렌더링은 동적 광원을 많이 사용하는 게임처럼 특정 사례에 사용하면 좋습니다. 콘솔과 PC에서의 동일한 물리 기반 조명 및 머티리얼도 스마트폰이나 태블릿으로 확장 및 축소할 수 있습니다.

다음은 URP의 세 가지 렌더링 옵션을 비교한 표입니다.

기능	Forward	Forward+	Deferred
오브젝트당 사용 가능한 최대 실시간 광원 수	9	무제한, 카메라별 제한 적용	무제한
픽셀당 노멀 인코딩	인코딩 없음 (정확한 노멀 값)	인코딩 없음 (정확한 노멀 값)	두 가지 옵션: G-buffer의 노멀 양자화(정확도 손실, 성능 향상) 팔면체 인코딩(정확한 노멀, 모바일 GPU에 상당한 성능 영향을 미칠 수 있음) 자세한 내용은 G버퍼의 노멀 인코딩(영문)을 참조하세요.
MSAA	지원	지원	지원 안 함
버텍스 조명	지원	지원 안 함	지원 안 함
카메라 스태킹	지원	지원	제한적 지원: Unity는 디퍼드 렌더링 경로를 사용하여 기본 카메라만 렌더링하며, 포워드 렌더링 경로를 사용하여 모든 오버레이 카메라를 렌더링합니다.

Unity 프로젝트에서 URP를 사용하는 방법을 자세히 알아보려면 Unity 고급 사용자를 위한 유니버설 렌더 파이프라인 전자책을 참고하세요.

GPU 최적화

그래픽스 렌더링을 효과적으로 최적화하려면 VR, 모바일, 웹 등 타겟 하드웨어의 제한 사항뿐만 아니라 GPU를 효과적으로 프로파일링하는 방법을 이해해야 합니다. 프로파일링을 통해 최적화로 원하는 결과가 나오는지 확인하고 검증할 수 있습니다.

— **VR**: VR 하드웨어는 원활한 몰입형 경험을 유지할 수 있도록 프레임 속도가 높고(보통 90fps 이상) 지연 시간이 짧아야 합니다. GPU가 양쪽 눈에 대해 각각 한 번씩, 총 두 번 렌더링해야 하므로 성능과 시각적 정확도를 모두 신중하게 최적화해야 합니다.

— **모바일**: 모바일 기기는 데스크톱이나 콘솔에 비해 처리 능력과 메모리가 제한적입니다. 드로우 콜을 최소화하고, 텍스처 크기를 줄이며, 간소화된 셰이더를 사용하는 최적화에 집중해야 배터리 소모나 기기 과열을 방지하며 원활한 성능을 보장할 수 있습니다.

— **웹**: 웹 플랫폼, 특히 Unity Web을 사용할 때는 브라우저 환경에서 실행된다는 제약점을 고려하여 성능과 균형을 맞춰야 합니다. 빌드 크기를 줄이고, 로딩 시간을 최소화하고, 다양한 브라우저 및 하드웨어 설정과의 호환성을 보장하는 것을 우선시하여 최적화해야 합니다.

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

GPU 벤치마킹

프로파일링 시에는 벤치마킹부터 하는 것이 좋습니다. 벤치마킹을 통해 특정 GPU에서 어느 정도의 프로파일링 결과를 얻을 수 있는지 알 수 있습니다.

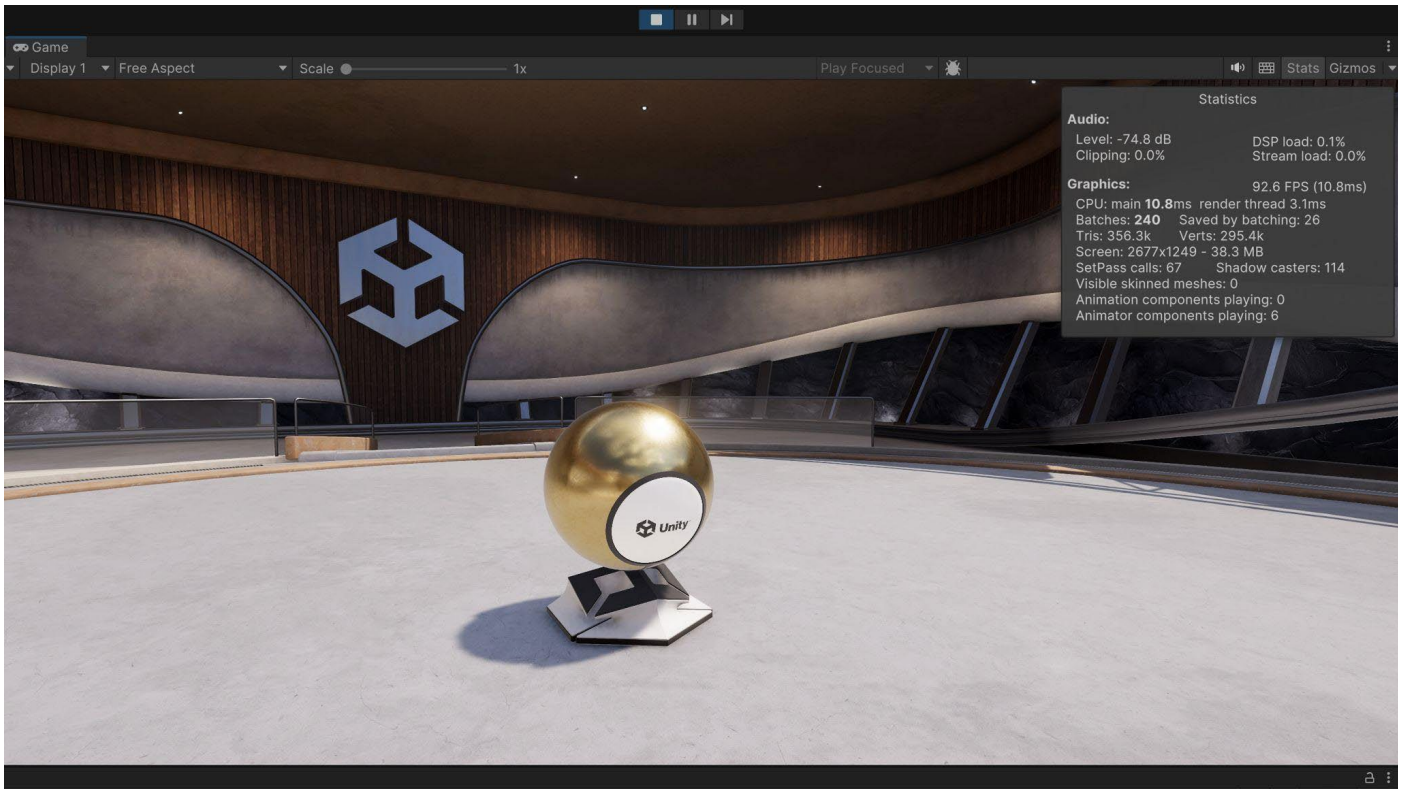
GPU 및 그래픽 카드에 대한 다양한 업계 표준 벤치마크 목록은 GFXBench를 참조하시기 바랍니다. 이 웹사이트에서는 현재 사용할 수 있는 GPU의 개요를 확인하고 여러 GPU를 서로 비교할 수 있습니다.

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 thread:** 게임 뷰의 프레임 하나를 렌더링하는 데 걸리는 전체 시간
- **Batches:** 함께 드로우할 드로우 콜 그룹
- **Tris(삼각형) 및 Verts(버텍스):** 메시 지오메트리 복잡도
- **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 단위 프레임 시간이 초당 프레임 수보다 더 정확한 지표입니다.



실시간 렌더링 정보가 표시되는 Stats 창

드로우 콜 줄이기

Unity는 게임 오브젝트를 렌더링할 때 그래픽스 API(예: OpenGL, Vulkan, Direct3D)에 드로우 콜을 보냅니다. CPU가 필요한 데이터를 준비하고 GPU에 전송하며, GPU가 커맨드를 처리하여 오브젝트를 렌더링해야 하므로 각 드로우 콜에는 리소스가 많이 소모됩니다. 드로우 콜 사이의 잦은 상태 변경(예: 머티리얼 전환)은 CPU 오버헤드를 늘릴 수 있습니다.

PC 및 콘솔 하드웨어는 대량의 드로우 콜을 처리할 수 있지만, 오버헤드가 여전히 크기 때문에 최적화하는 것이 좋습니다. 모바일 기기, VR 헤드셋, 웹 브라우저에서 드로우 콜을 최적화하는 것은 성능을 유지하는 데 아주 중요합니다. 드로우 콜을 줄이면 특히 리소스가 제한된 플랫폼에서 원활하고 효율적인 렌더링을 보장할 수 있습니다.

특히 웹, VR 및 모바일 플랫폼에서 성능을 최적화하려면 드로우 콜을 줄이는 것이 핵심입니다. 다음은 드로우 콜을 줄이기 위한 주요 전략입니다.

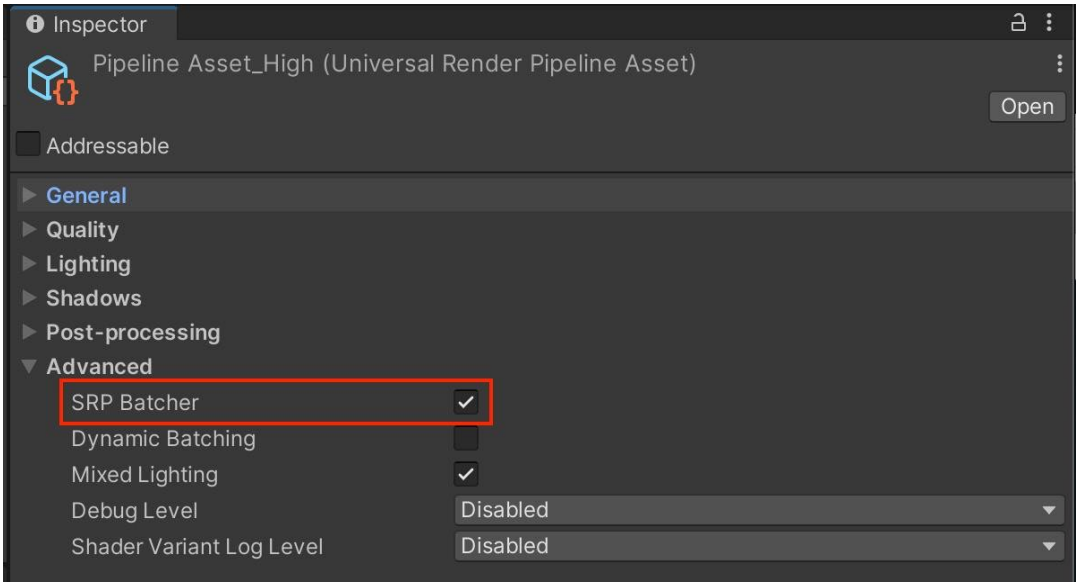
1. **텍스처 아틀라스 사용**: 여러 텍스처를 하나의 텍스처 아틀라스로 통합하여 텍스처 바인딩과 드로우 콜을 최소화하세요. 상태 변경을 줄이면 렌더링 효율이 향상되는 웹과 모바일 환경에서 특히 중요합니다.
2. **머티리얼 최적화**: 프로젝트에서 사용하는 머티리얼과 셰이더의 수를 줄이세요. 공유되는 머티리얼은 함께 배칭하기 쉽고 드로우 콜 오버헤드가 줄어듭니다.
3. **디테일 수준(LOD) 활용**: LOD 기법을 사용하여 멀리 있는 오브젝트의 복잡도를 줄이고 카메라에서 먼 오브젝트의 드로우 콜을 줄이세요. 이 접근 방식은 멀미를 방지하기 위해 높은 프레임 속도를 유지해야 하는 VR과 처리 성능이 제한적인 모바일 플랫폼에서 아주 중요합니다.
4. **컬링 기법 적용**: 절두체 컬링과 오클루전 컬링을 사용하여 화면에 보이는 오브젝트만 렌더링하세요. 카메라의 시야를 벗어나거나 다른 지오메트리에 가려진 오브젝트를 드로우하지 않으면 드로우 콜의 수를 줄여 모든 플랫폼, 특히 리소스가 제한된 웹과 모바일 환경에서 성능을 향상할 수 있습니다.

드로우 콜 배칭 사용

드로우 콜 배칭은 메시를 결합하여 Unity가 더 적은 드로우 콜만으로 렌더링할 수 있게 만드는 최적화 방법입니다.

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 에셋 설정의 **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 배리언트 를 사용하면 SRP 배칭을 향상할 수 있습니다. 프로젝트에서 이러한 렌더링 워크플로를 어떻게 활용하는지 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 모두를 한 번에 드로우합니다. 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 배칭의 효과를 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 레퍼런스가 반환됩니다. 이러면 해당 머티리얼이 포함되어 있는 기존 배칭이 중단됩니다. 배칭된 오브젝트의 머티리얼에 액세스하려면 Renderer.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.

GPU 상주 드로어

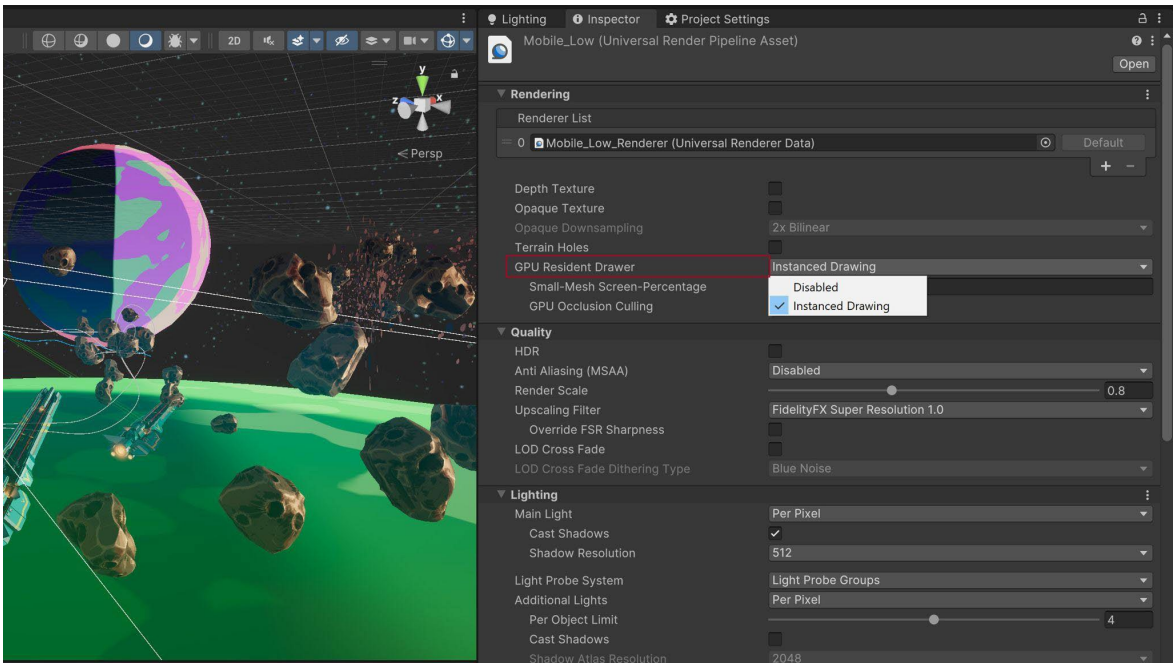
URP 와 HDRP에서 사용할 수 있는 GPU 상주 드로어는 CPU 시간을 최적화하여 성능을 향상하는 GPU 기반 렌더링 시스템입니다. Vulkan과 Metal을 사용하는 고사양 모바일 플랫폼을 포함하여 크로스 플랫폼 렌더링을 지원하며, 기존 프로젝트에서 바로 사용할 수 있도록 설계되었습니다.

GPU 상주 드로어는 BatchRendererGroup API를 사용하여 GPU 인스턴싱으로 게임 오브젝트를 드로우하므로 드로우 콜을 줄이고 CPU 처리 시간을 확보할 수 있습니다. GPU 상주 드로어는 아래 설정 및 환경에서만 사용할 수 있습니다.

— 포워드+ 렌더링 경로

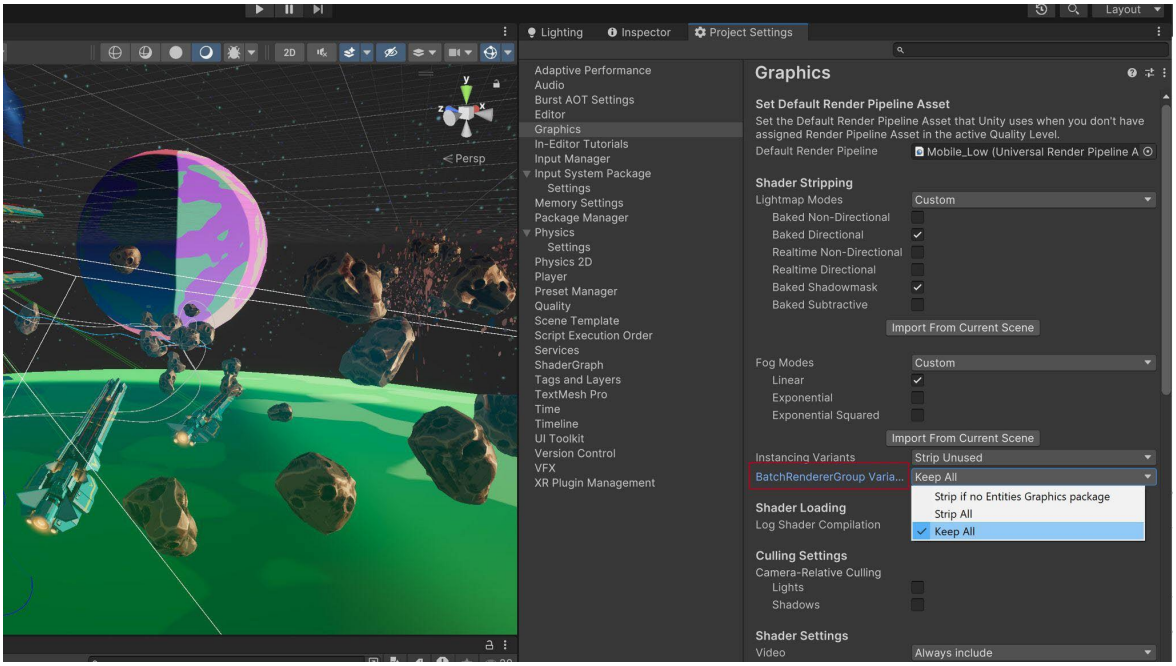
— 그래픽스 API 및 컴퓨트 셰이더를 지원하는 플랫폼(OpenGL ES 제외)

— Mesh Renderer 컴포넌트가 있는 게임 오브젝트



GPU 상주 드로어: 렌더 파이프라인 에셋에서 **Instanced Drawing** 선택

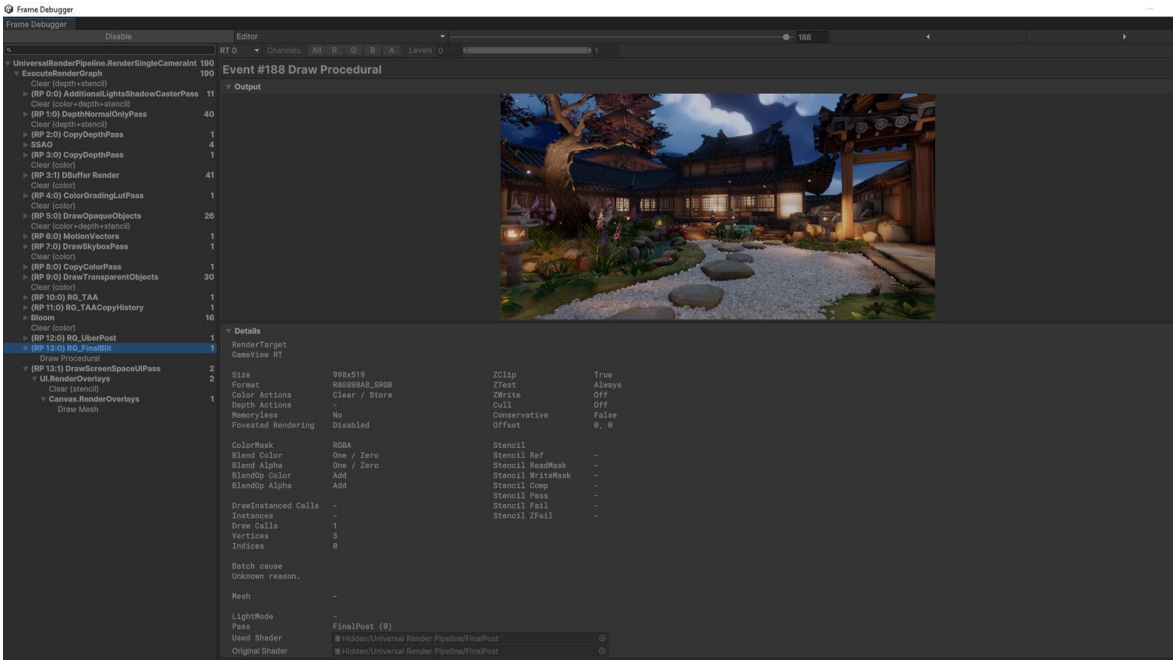
Instanced Drawing 옵션을 선택하면 UI에서 ‘BatchRenderGroup Variants setting must be ‘Keep All’’ 이라는 경고 메시지가 표시될 수 있습니다. 그래픽스 설정에서 이 옵션을 조정하면 GPU 상주 드로어 설정이 완료됩니다.



Graphics 설정에서 BatchRenderGroup Variant를 **Keep All**로 설정

프레임 디버거 사용

프레임 디버거 를 사용하면 개별 드로우 콜을 통해 각 프레임이 구성되는 방식을 확인할 수 있습니다. 셰이더 프로퍼티의 문제를 해결하고 게임이 어떻게 렌더링되는지 분석하는 매우 유용한 툴입니다.



각 프레임을 개별 단계로 나누는 프레임 디버거

프레임 디버거를 처음 접한다면 여기에서 튜토리얼을 살펴보세요.

Split Graphics Jobs

여러 데스크톱 및 콘솔 플랫폼에서 지원하는 이 스레딩 모드의 목적은 CPU 멀티 스레딩 성능을 향상하는 것입니다. 주된 향상은 일반적인 게임 로직과 오케스트레이션을 담당하는 메인 스레드와 렌더링 작업을 담당하는 네이티브 그래픽스 잡 스레드 간의 불필요한 동기화를 줄이는 데서 비롯됩니다.

이 새로운 스레딩 모드는 프레임마다 제출되는 드로우 콜이 늘어날수록 성능을 크게 향상합니다. 오브젝트와 텍스처가 많아 복잡한 씬과 같이 드로우 콜이 많은 씬에서 성능이 크게 향상될 수 있습니다.

동적 광원 수 줄이기

XR, 모바일 또는 웹 플랫폼용 콘텐츠를 개발하는 경우, 특히 포워드 렌더링을 사용한다면 동적 광원 수를 줄이는 것이 중요합니다. 동적 광원은 성능에 큰 영향을 주므로 프레임 속도가 떨어지고 전력 소비가 늘어날 수 있으며, 이는 특히 리소스가 제한된 환경에서 치명적입니다.

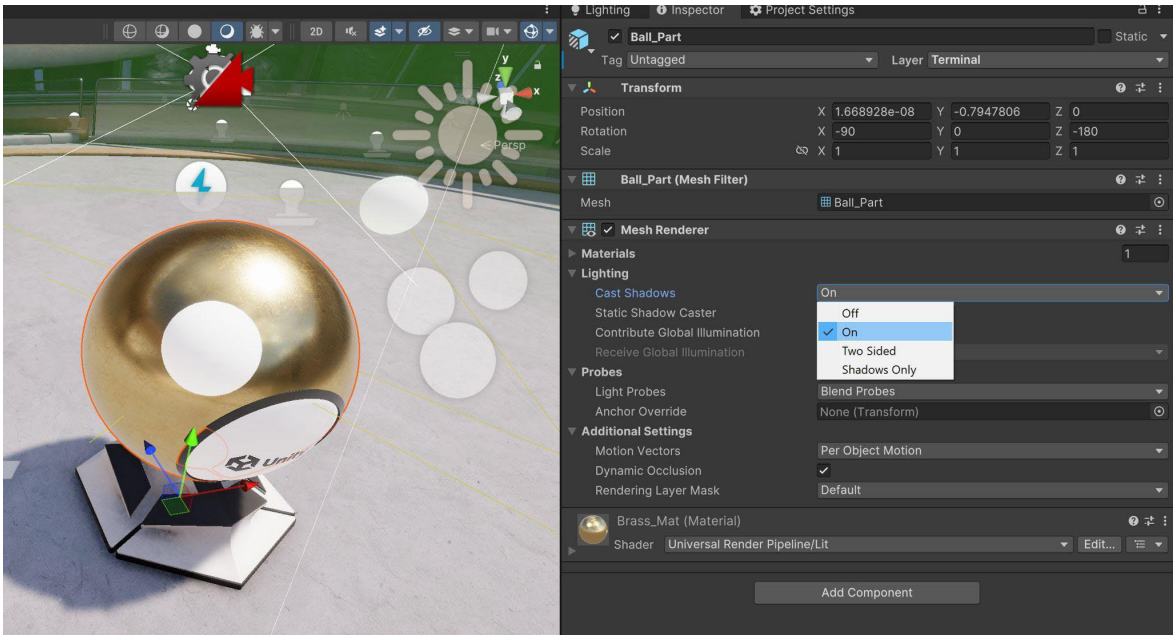
대신 동적 오브젝트에 커스텀 셰이더 이펙트나 라이트 프로브를 사용해 낮은 성능 비용으로 조명을 시뮬레이션해 보세요. 정적 오브젝트의 경우, 런타임 오버헤드 없이 고품질 조명을 제공하는 베이크된 광원을 선택하는 편이 더 효율적입니다. 조명 관리에 신경을 쓰면 XR, 모바일 및 웹 애플리케이션에서 비주얼 품질을 유지하면서도 성능을 최적화할 수 있습니다.

URP와 빌트인 렌더 파이프라인에서 실시간 광원이 저마다 구체적으로 어떻게 제한되는지 알아보려면 이 기능 비교 테이블을 참고하세요.

그림자 비활성화

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

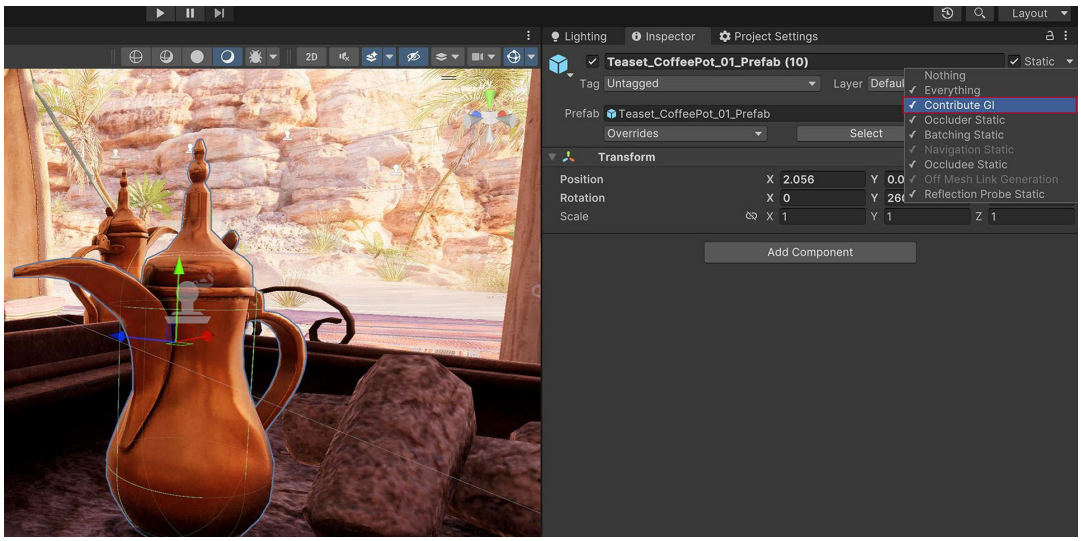
또한 캐릭터 아래의 간단한 메시나 사각형 영역에 블러된 텍스처를 적용하여 가짜 그림자를 만들거나, 커스텀 셰이더로 블롭 섀도우를 만들 수 있습니다.



Cast Shadows를 비활성화하여 드로우 콜 줄이기

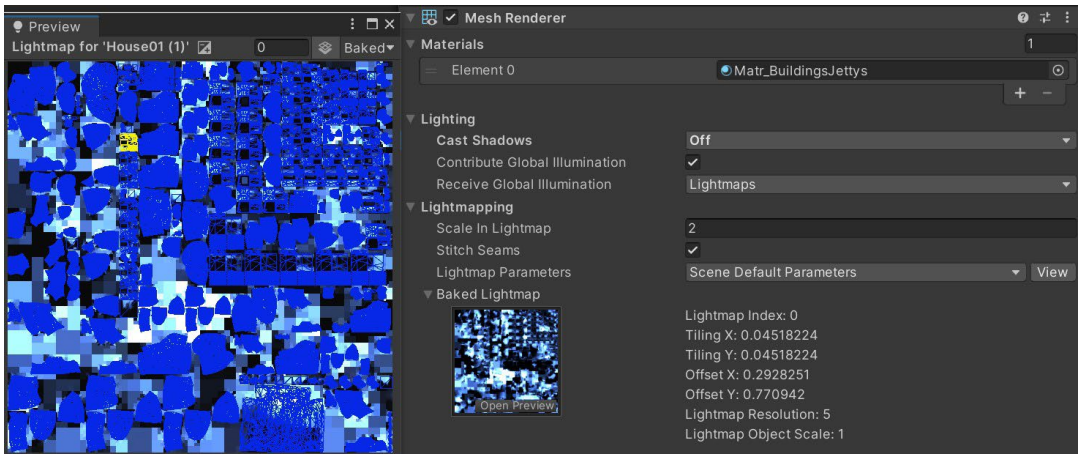
라이트맵에 조명 베이크

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



Contribute GI 설정 활성화

이제 베이크된 그림자와 조명을 런타임 시 성능 저하 없이 렌더링할 수 있습니다. 프로그레시브 CPU 및 GPU 라이트매퍼로는 전역 조명의 베이크를 가속화할 수 있습니다.

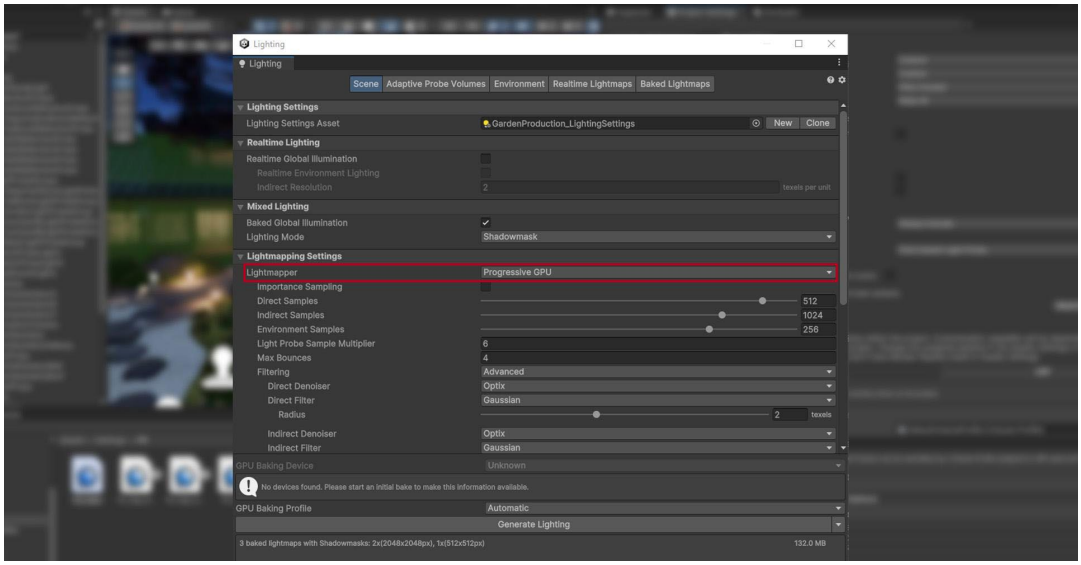


라이트매핑 설정(Windows > Rendering > Lighting Settings)과 **라이트맵 크기**를 조정하여 메모리 사용량 제한

Unity에서 라이트매핑을 사용하려면 매뉴얼 가이드와 광원 최적화를 다룬 이 글을 참조하시기 바랍니다.

GPU 광원 베이크

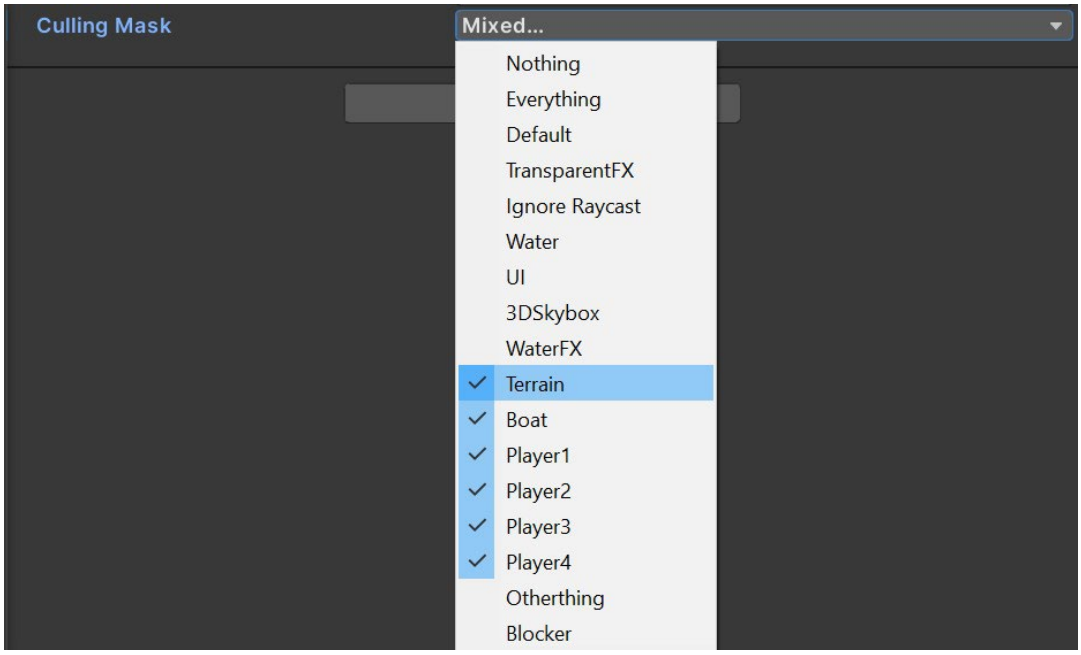
Unity 6에서는 프로그레시브 GPU 라이트매퍼를 정식 제작에 사용할 수 있습니다. GPU의 성능을 활용해 기존 CPU 라이트매핑보다 훨씬 빠르게 베이크하여 조명 데이터 생성 시간을 크게 단축하도록 설계되었습니다. 이 시스템에는 코드베이스를 간소화하고 더 예측 가능한 결과를 제공하는 새로운 광원 베이크(light baking) 백엔드가 포함되어 있습니다. 또한 GPU 최소 요구 사항이 2GB로 낮아져 이제 더 많은 개발자가 이 기능을 이용할 수 있습니다. 런타임에 라이트 프로브 위치를 옮길 수 있어 절차적으로 생성되는 콘텐츠에 특히 유용한 새로운 API도 제공하며, 이외에도 다양한 사항이 개선되었습니다.



Lightmapper로 Progressive GPU 선택

광원 레이어 사용

여러 개의 광원이 있는 복잡한 씬에서는 레이어를 사용해 오브젝트를 분리한 다음 각 광원의 영향을 특정 컬링 마스크로 한정합니다.



레이어를 사용하여 광원의 영향을 특정 컬링 마스크로 제한

적응적 프로브 볼륨

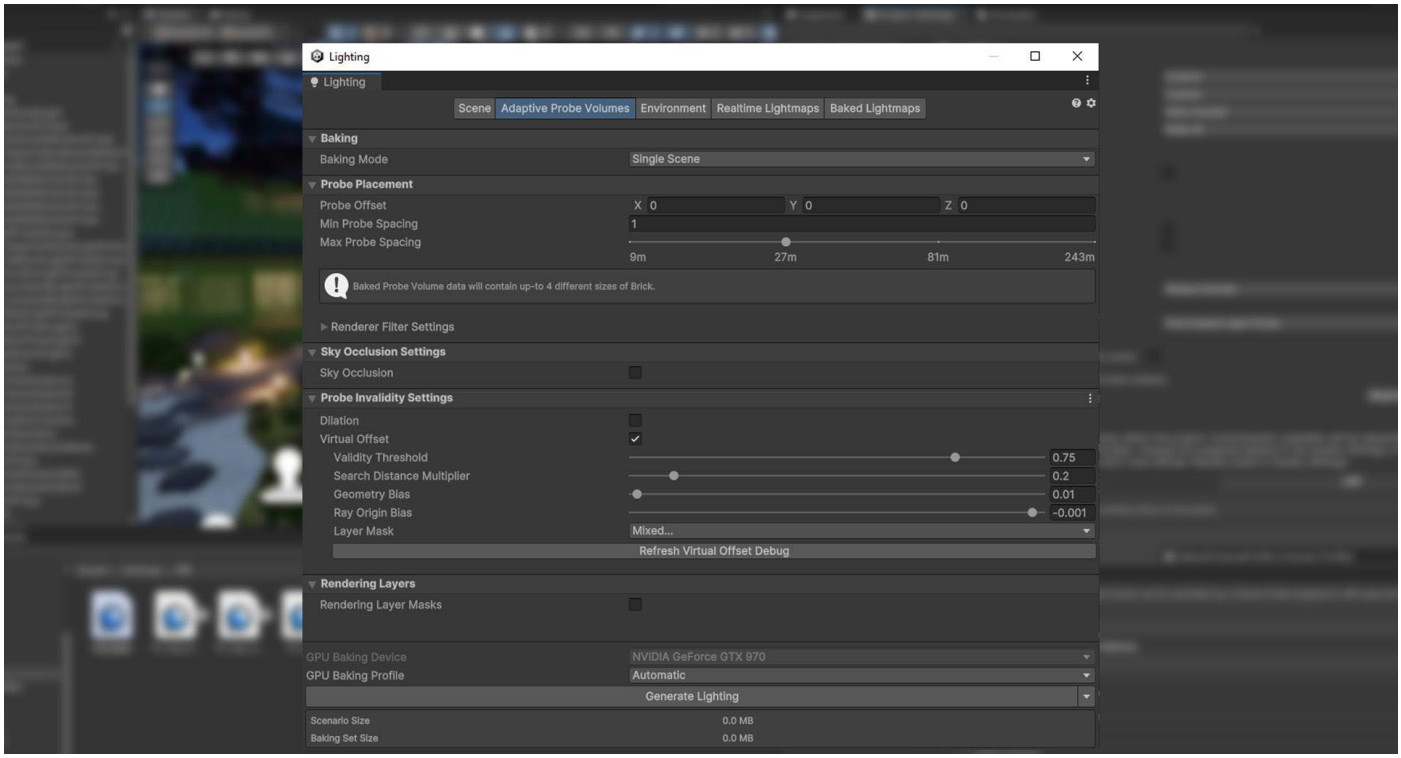
Unity 6에는 전역 조명을 처리하는 정교한 솔루션인 APV(적응적 프로브 볼륨)가 추가되어 복잡한 씬에서 효율적인 동적 조명을 구현할 수 있습니다. APV는 특히 모바일 및 저사양 기기에서 성능과 비주얼 품질을 모두 최적화할 수 있으며, 고사양 플랫폼을 위한 고급 기능도 함께 제공합니다.

Unity의 APV(적응적 프로브 볼륨)는 특히 동적인 대형 씬에서 전역 조명을 개선할 다양한 기능을 제공합니다. URP는 이제 저사양 기기에서 성능을 높이는 버텍스별 샘플링을 지원하며, VFX 파티클은 프로브 볼륨에 베이크되는 간접 조명의 이점을 활용할 수 있습니다.

URP 3D 샘플의 오아시스 환경에 APV를 배치하는 모습

APV 데이터를 디스크에서 CPU와 GPU로 스트리밍하여 대규모 환경에서 조명 정보를 최적화할 수 있습니다. 개발자는 여러 조명 시나리오를 베이크하고 블렌딩하여 낮/밤 사이클 같은 실시간 전환을 구현할 수 있습니다. 또한 스카이 오클루전을 지원하고, Ray Intersector API와 통합되어 프로브를 더 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 프로브와 독립적으로 APV를 베이크할 수 있어 워크플로가 더욱 개선됩니다.

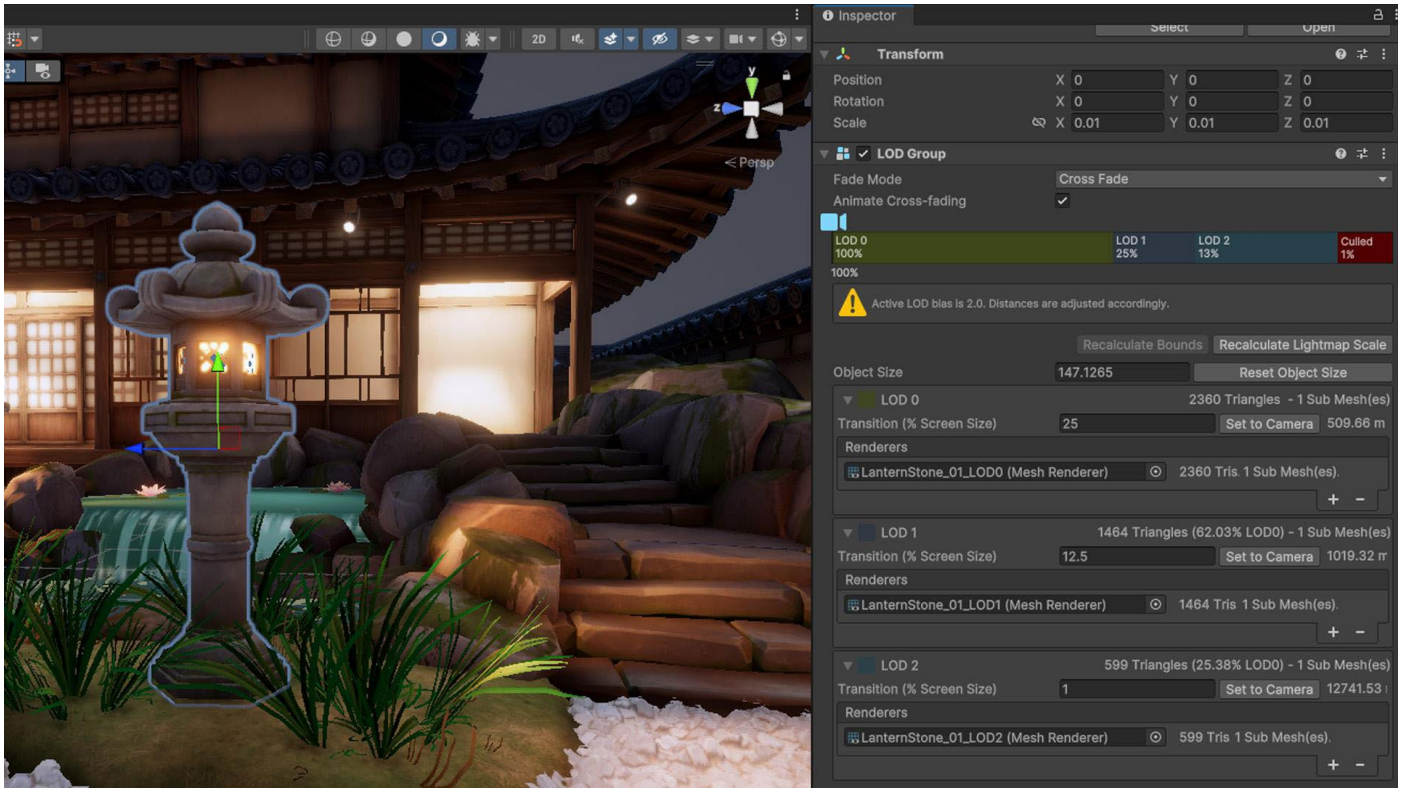
사용해 보려면 GDC 2023의 적응적 프로브 볼륨으로 효율적이고 효과적인 조명 구현하기(영문) 강연을 먼저 시청해 보세요.



Lighting 설정의 Adaptive Probe Volumes 창

디테일 수준(LOD) 사용

디테일 수준(LOD)을 통해 오브젝트가 카메라에서 멀어질수록 단순한 머티리얼과 셰이더의 단순한 메시를 사용하도록 전환하여 GPU 성능을 향상할 수 있습니다.



LOD Group을 사용하는 메시의 예시

다양한 해상도로 모델링한 소스 메시

Unity Learn의 LOD 사용하기 튜토리얼에서 자세한 내용을 확인하세요.

오클루전 컬링을 사용하여 숨겨진 오브젝트 제거

다른 오브젝트 뒤에 숨겨진 오브젝트는 계속 렌더링되며 리소스 비용을 발생시킬 수 있습니다. 오클루전 컬링을 사용하면 이러한 오브젝트를 폐기할 수 있습니다.

카메라 뷰 바깥의 절두체 컬링은 자동이지만 오클루전 컬링은 베이크된 과정입니다. 오브젝트를 **Static Occluders** 또는 **Occludees**로 표시한 다음 **Window > Rendering > Occlusion Culling**을 통해 베이크하면 됩니다. 모든 씬에 필요한 것은 아니지만 컬링이 성능을 향상하는 경우가 있으므로, 오클루전 컬링 활성화 전후를 프로파일링하여 성능이 향상되는지 확인하는 게 좋습니다.

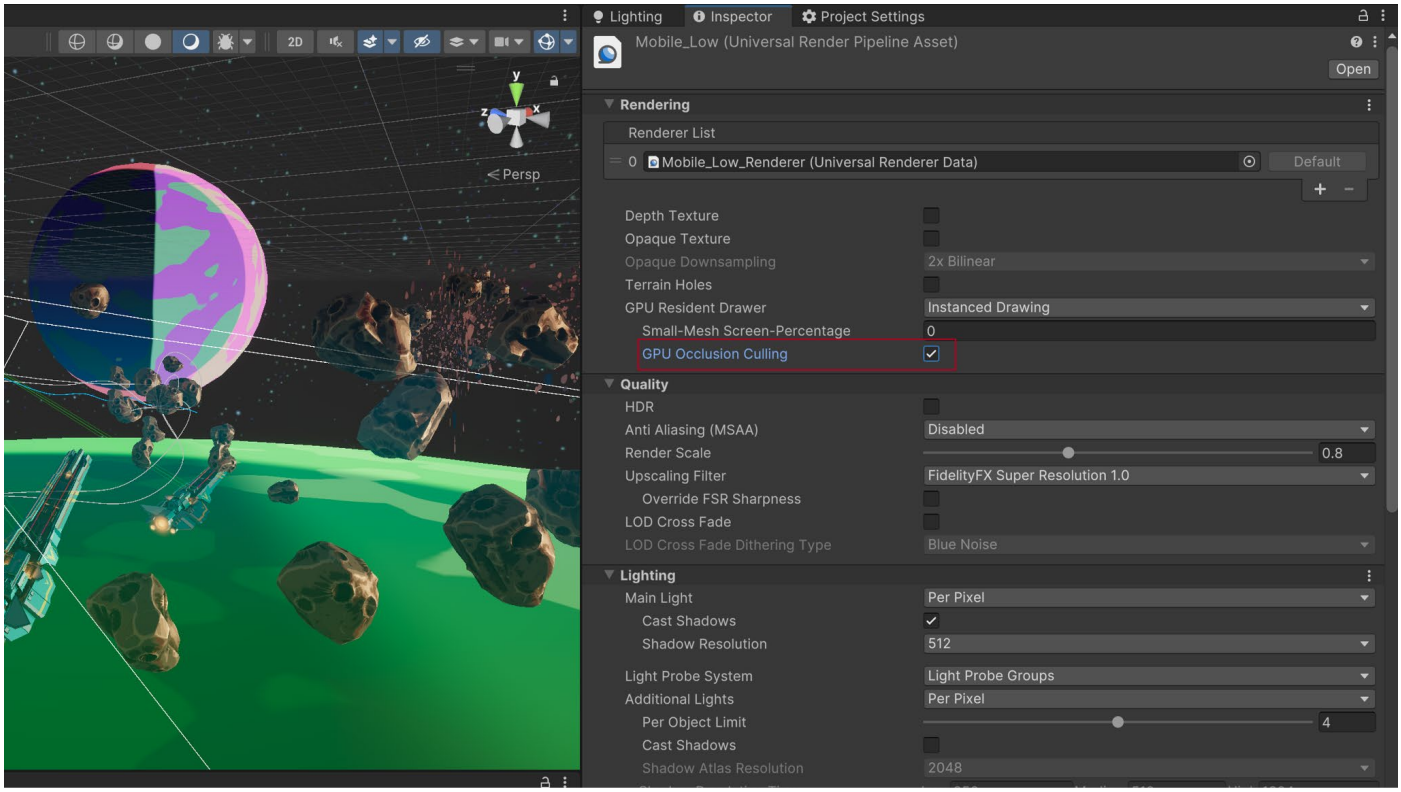
자세한 내용은 오클루전 컬링 사용하기 튜토리얼을 확인하세요.

GPU 오클루전 컬링

GPU 오클루전 컬링은 특히 복잡한 지오메트리와 가려진 오브젝트가 많은 씬에서 렌더링 성능을 크게 향상합니다. GPU 오클루전 컬링은 프레임마다 오버드로우되는 양을 줄여 게임 오브젝트의 성능을 향상합니다. 즉, 렌더러가 보이지 않는 오브젝트를 드로우하느라 리소스를 낭비하지 않게 합니다. 전통적으로 3D 환경에서는 보이지 않는 오브젝트를 드로우하는 것이 큰 성능 병목 지점이었습니다. GPU 오클루전 컬링의 주요 기능은 다음과 같습니다.

1. **GPU 가속**: 오클루전 컬링 시 CPU에 크게 의존했던 이전 버전과는 달리, Unity 6는 GPU 가속을 활용합니다. 따라서 더 효율적인 실시간 계산이 가능하며, CPU 오버헤드가 줄어들고, 성능 저하 없이 복잡한 씬을 실행할 수 있습니다.
2. **GPU 상주 드로어와 연동**: GPU 오클루전 컬링은 대규모 오브젝트 세트와 그 가시성을 처리하는 GPU 상주 드로어와 함께 작동하여 정적 및 동적 오브젝트의 렌더링 파이프라인을 더욱 최적화합니다.
3. **동적 및 정적 오브젝트 컬링:** Unity 6의 오클루전 컬링 시스템은 정적 오브젝트와 동적 오브젝트를 모두 효과적으로 관리할 수 있습니다. 동적 오브젝트는 이제 포털 기반 시스템으로 컬링되며, 씬 내에서 움직이더라도 보이는 오브젝트만 처리됩니다.
4. **베이크 및 실시간 조정:** 개발자는 에디터에서 오클루전 데이터를 베이크한 다음 런타임에 사용할 수 있습니다. 이 프로세스는 씬을 셀로 나누고 셀 간의 가시성을 계산하여 카메라가 움직이면 실시간으로 조정합니다. 에디터에서 오클루전 컬링을 시각화할 수도 있으며, 이를 바탕으로 개발자가 씬을 더 최적화할 수 있습니다.
5. **메모리 관리:** Unity 6는 오클루전 데이터의 메모리 사용량을 관리하는 툴을 제공하므로 오클루전 컬링 프로세스를 조정하여 성능과 메모리 사용량의 균형을 맞출 수 있습니다.

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



렌더 파이프라인 에셋의 GPU Occlusion Culling 옵션

모바일 기기의 네이티브 해상도 사용 지양

스마트폰과 태블릿이 점점 발전함에 따라 새로 출시되는 기기일수록 매우 높은 해상도의 디스플레이가 탑재되고 있습니다.

Screen.SetResolution(width, height, false)을 사용하여 출력 해상도를 낮추면 어느 정도의 성능을 다시 확보할 수 있습니다. 다양한 해상도를 프로파일링하여 품질과 속도 간에 최적의 균형점을 찾으세요.

카메라 사용 제한

활성화된 모든 카메라는 작업 수행 여부에 상관없이 어느 정도의 오버헤드를 유발합니다. 따라서 렌더링에 필요한 카메라 컴포넌트만 사용하는 것이 좋습니다. 저사양 모바일 플랫폼에서는 카메라당 최대 1ms의 CPU 시간을 사용할 수 있습니다.

시공간 포스트 프로세싱

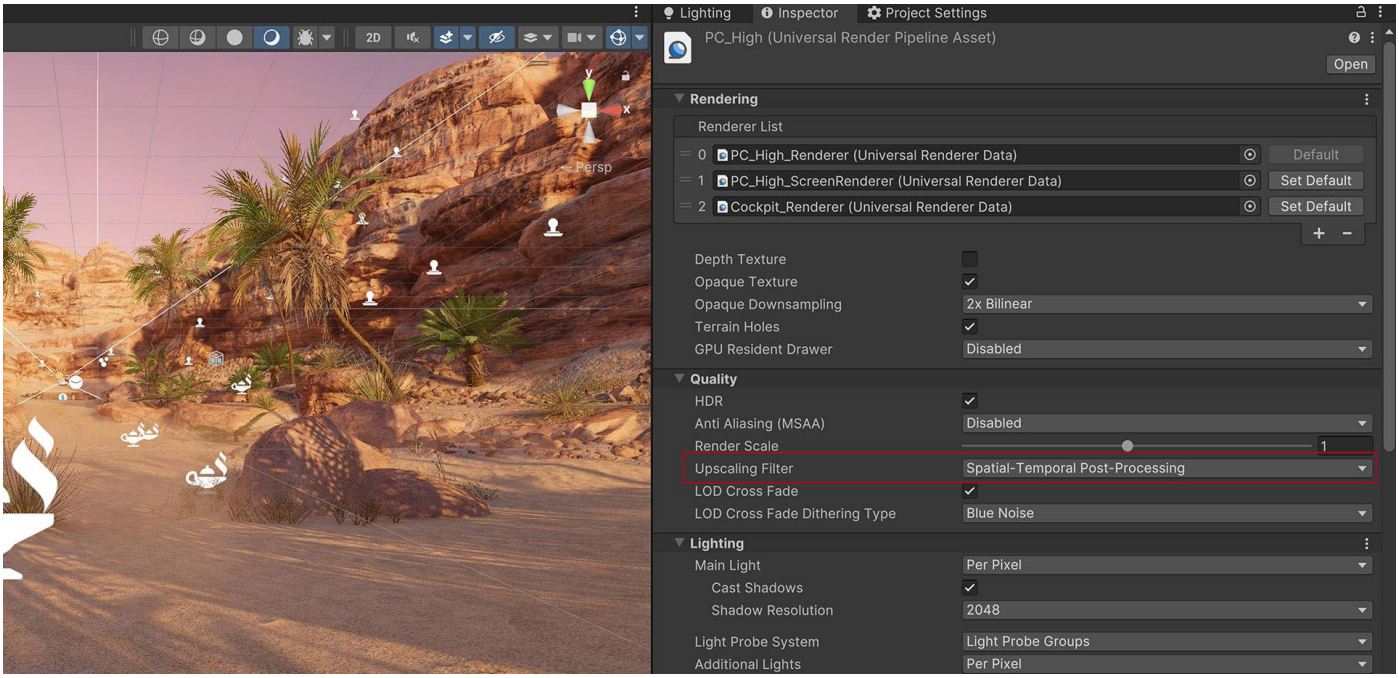
STP(시공간 포스트 프로세싱)는 모바일 기기, 콘솔, PC 등 다양한 플랫폼에서 비주얼 품질을 향상하기 위해 설계되었습니다. STP는 HDRP와 URP 렌더 파이프라인 모두에서 사용할 수 있는 시공간 안티앨리어싱 업스케일러로, 기존 콘텐츠를 수정하지 않아도 적용할 수 있는 고품질 콘텐츠 스케일링을 제공합니다.

GPU 성능을 위해 최적화된 솔루션으로, 렌더링 시간을 단축하고 비주얼 품질을 유지하면서 성능을 간단하게 향상할 수 있습니다.

URP에서 STP를 활성화하는 방법은 다음과 같습니다.

— 프로젝트 창에서 활성 URP 에셋을 선택합니다.

— 인스펙터에서 **Quality > Upscaling Filter**로 이동하고 **Spatial-Temporal Post-Processing**을 선택합니다.



URP 에셋에서 STP 활성화

셰이더

단순하고 최적화된 셰이더 사용

URP는 모바일 플랫폼에 최적화된 가벼운 릿 셰이더와 언릿 셰이더를 다양하게 제공하므로 웹, 모바일 및 XR 프로젝트에서 좋은 시작점으로 활용할 수 있습니다. 성능을 극대화하려면 셰이더 배리에이션을 최소로 유지하세요. 배리언트가 많으면 런타임 메모리 사용량에 영향을 줄 수 있으며, 특히 리소스가 제한된 기기라면 영향이 더 클 수 있습니다.

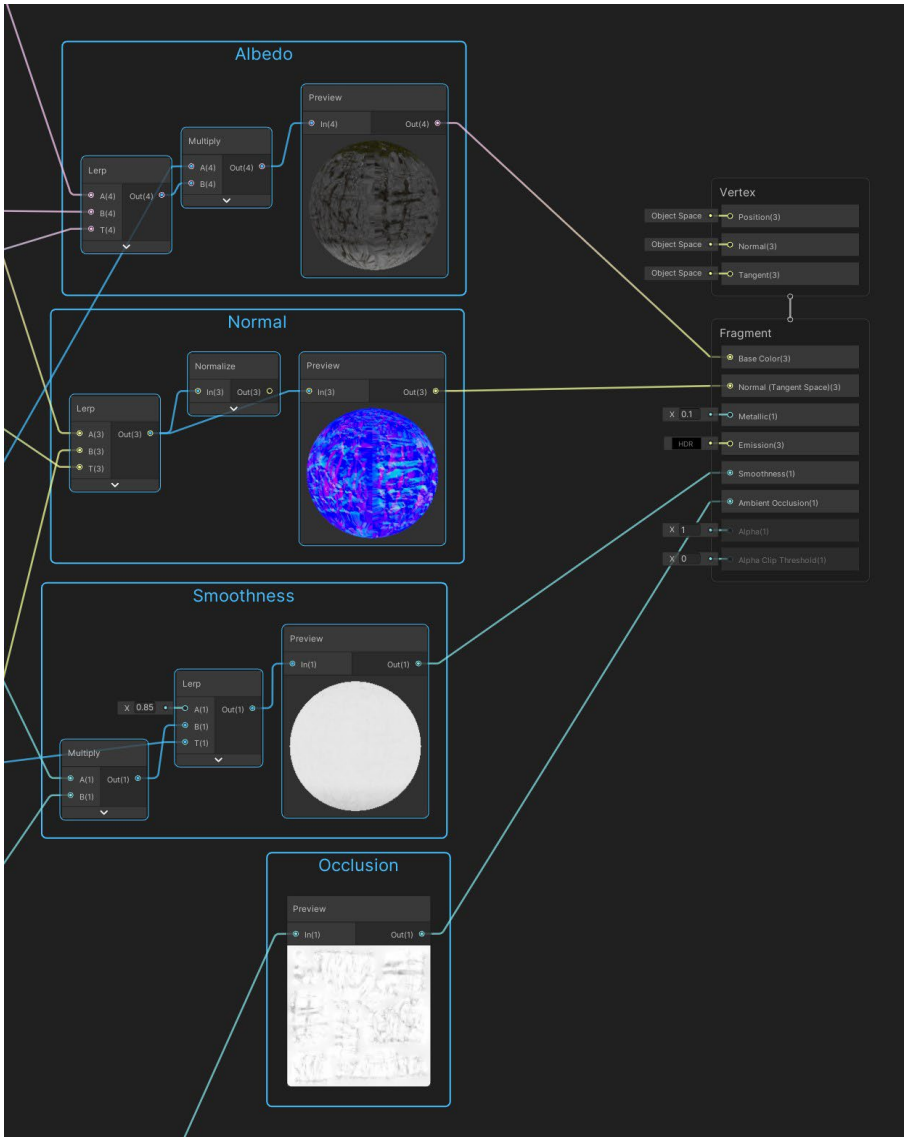
요구 사항을 충족하는 데 기본 URP 셰이더로 충분하지 않다면 프로젝트의 셰이더를 시각화하여 디자인하고 최적화하는 Shader Graph를 사용해 커스터마이징할 수 있습니다. 셰이더를 최적화하기 위한 몇 가지 팁은 다음과 같습니다.

— **계산 최소화**: 연산 횟수를 줄여 셰이더를 단순화하세요. 특히 각 픽셀을 계산해야 하는 프래그먼트 셰이더에서 연산을 줄이는 것이 좋습니다. 모바일이나 XR 애플리케이션에서 GPU에 부담이 되는 복잡한 수학 연산이나 if 문과 같은 무거운 브랜칭 로직을 피하세요.

— **통합 텍스처 사용**: ORM(오클루전, 거칠기, 메탈릭) 맵과 같은 통합 텍스처를 사용하여 텍스처 룩업 횟수를 줄이세요. 이 접근 방식을 사용하면 여러 맵을 하나의 텍스처로 통합하여 모바일, 웹 및 XR 플랫폼에서 성능을 유지하는 데 매우 중요한 GPU 워크로드를 줄일 수 있습니다.

— **Shader Graph 최적화**: Shader Graph를 사용할 때는 셰이더 로직을 간소화하여 성능을 향상하는 데 집중하세요. 이는 각 셰이더의 효율성이 전반적인 성능에 직접적인 영향을 미치는 모바일 및 XR 애플리케이션에서 특히 중요합니다.

— **정기적인 프로파일링**: 웹, 모바일, XR 등의 타겟 기기에서 지속적으로 셰이더를 테스트하고 프로파일링하여 성능 요구 사항을 만족하는지 확인하세요. 정기적으로 프로파일링하면 잠재적인 문제를 초기에 발견하고 각 플랫폼의 요구 사항에 맞게 최적화하는 데 도움이 됩니다.



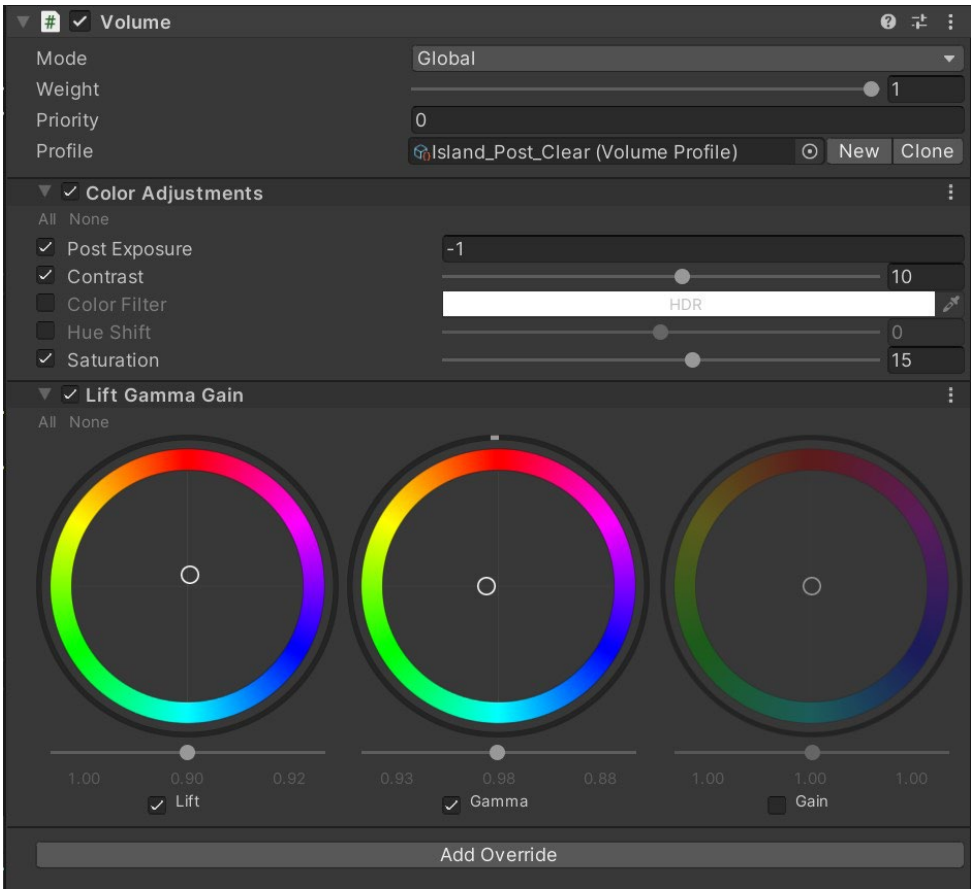
Shader Graph로 커스텀 셰이더 만들기

오버드로우 및 알파 블렌딩 최소화

불필요한 투명 또는 반투명 이미지를 드로우하는 것은 지양하고, 거의 보이지 않는 이미지나 효과는 오버랩하지 않는 것이 좋습니다. 모바일 플랫폼은 오버드로우와 알파 블렌딩의 영향을 받습니다. RenderDoc 그래픽스 디버거를 사용해 오버드로우를 확인할 수 있습니다. 렌더링 디버거를 사용하여 다양한 조명, 렌더링, 머티리얼 프로퍼티를 시각화할 수도 있습니다. 이러한 시각화를 통해 렌더링 문제를 식별하고 씬과 렌더링 설정을 최적화할 수 있습니다.

포스트 프로세싱 이펙트 제한

글로우와 같은 전체 화면 포스트 프로세싱 이펙트는 성능을 크게 떨어뜨릴 수 있습니다. 프로젝트의 아트 방향성에 따라 주의하여 사용하세요. 모바일, XR 및 웹에서 포스트 프로세싱이 성능 병목 지점이 되는 경우가 많으므로 특별히 주의해서 벤치마킹하고 그에 따라 아트를 조정해야 합니다.



모바일 애플리케이션에서는 포스트 프로세싱 효과를 단순하게 유지하세요.

Renderer.material 유의하기

스크립트의 **Renderer.material**에 액세스하면 머티리얼이 복사되고 새로운 사본에 대한 레퍼런스가 반환됩니다. 이러면 해당 머티리얼이 포함되어 있는 기존 배칭이 중단됩니다. 배칭된 오브젝트의 머티리얼에 액세스하려면 **Renderer.sharedMaterial**을 대신 사용하세요.

SkinnedMeshRenderers 최적화

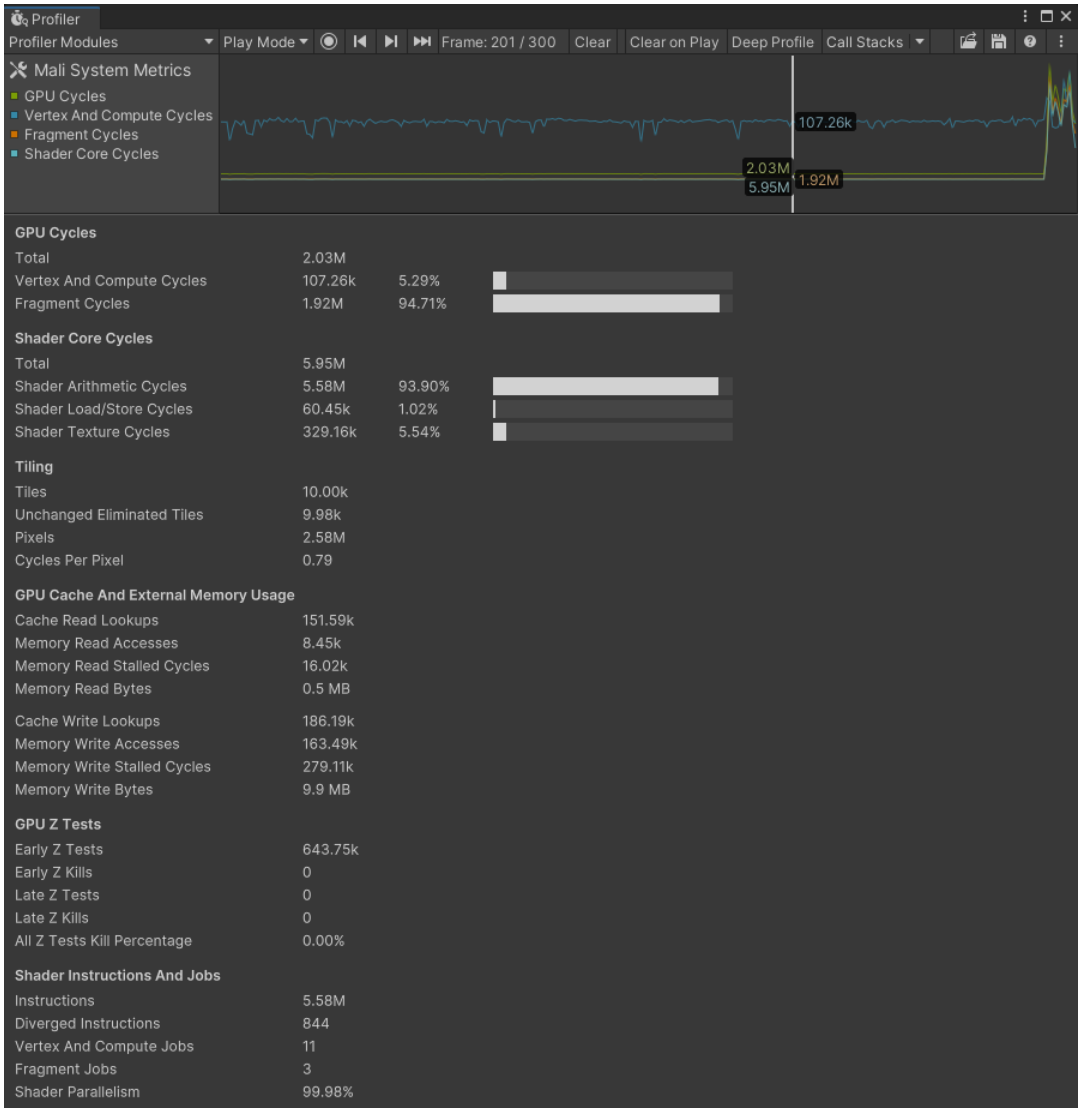
스킨드 메시를 렌더링하는 데는 비용이 많이 듭니다. 모든 오브젝트에 **SkinnedMeshRenderer**를 사용할 필요가 있는지 확인하세요. 게임 오브젝트에 가끔 애니메이션만 필요한 경우 **BakeMesh** 함수를 사용하여 스킨드 메시를 정적인 포즈로 고정하고 런타임에는 더 단순한 **MeshRenderer**로 대체하세요.

반사 프로브 최소화

Reflection Probe 컴포넌트는 사실적인 반사를 만들어 낼 수 있지만 배치 처리 시 소모되는 리소스가 매우 높습니다. 저해상도 큐브맵, 컬링 마스크, 텍스처 압축을 사용해 런타임 성능을 높여 보세요.

System Metrics Mali

System Metrics Mali 패키지를 활용해 ARM GPU를 사용하는 기기에서 하위 수준 시스템 또는 하드웨어 성능 지표에 액세스할 수도 있습니다. Unity 프로파일러에서 하위 수준 GPU 지표를 모니터링하거나, Recorder API를 사용하여 런타임에 하위 수준 GPU 지표에 액세스하거나, CI(지속적 통합) 테스트를 통해 성능 테스트를 자동화하는 등의 기능을 활용할 수 있습니다.



Mali System Metrics 프로파일러 모듈

다음 가이드를 통해 Unity의 조명 워크플로에 대해 더 자세히 알아보세요.

— Unity 고급 사용자를 위한 URP 소개

— URP의 2D 조명 및 그림자 기법

— 2D 게임을 위한 조명 및 AI 기법(영문)

— URP 사용한 대중적인 시각 효과 레시피

사용자 인터페이스

Unity는 기존 Unity UI와 새로운 UI 툴킷의 두 가지 UI 시스템을 제공합니다. 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(Immediate Mode GUI)가 지원하는 일부 기능이 UI 툴킷에 포함되어 있지 않습니다. Unity UI 및 IMGUI는 특정 활용 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.

캔버스 나누기

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

다수의 캔버스를 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 캔버스 내의 모든 UI 요소가 동일한 Z 값, 머티리얼, 텍스처를 갖도록 해야 합니다.

보이지 않는 UI 요소 숨기기

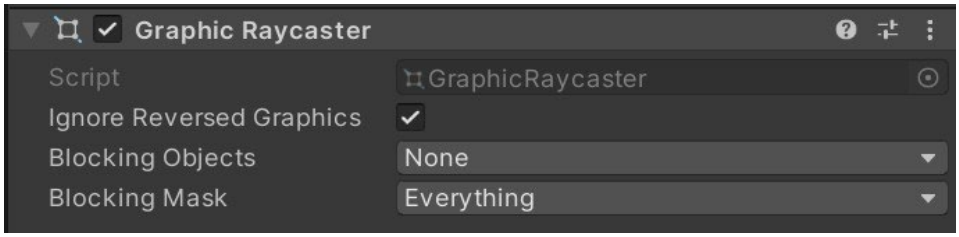
게임에서 간헐적으로만 나타나는 UI 요소(예: 캐릭터가 대미지를 입었을 때 나타나는 체력 표시줄)가 있을 수 있습니다. 보이지 않는 UI 요소가 활성화된 경우에도 드로우 콜을 사용할 수 있습니다. 보이지 않는 UI 컴포넌트를 명시적으로 비활성화하고 필요할 때 다시 활성화하세요.

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

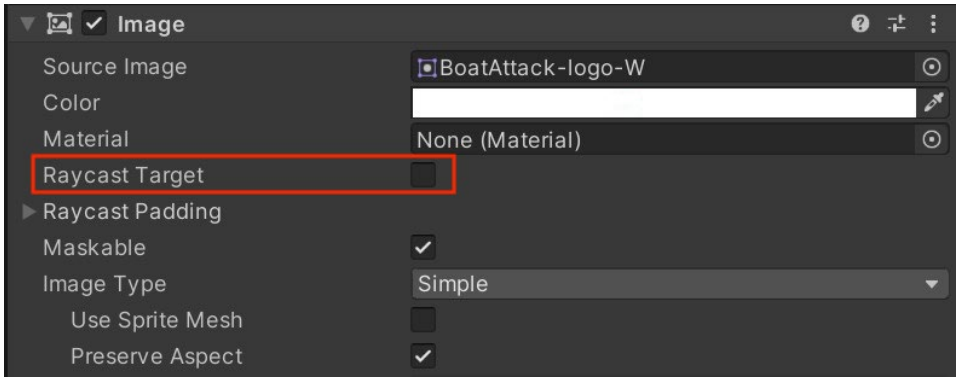
GraphicRaycaster 컴포넌트는 화면 터치나 클릭과 같은 입력 이벤트에 필요합니다. 이 컴포넌트는 화면의 각 입력 지점을 순환하며 입력 지점이 UI의 RectTransform 내에 있는지 확인합니다.

계층 구조의 맨 위쪽 Canvas에서 기본 **GraphicRaycaster**를 제거하세요. 대신, 상호 작용이 필요한 개별 요소(버튼, 스크롤 사각 영역 등)에만 **GraphicRaycaster**를 추가합니다.



기본적으로 활성화되어 있는 Reversed Graphics 비활성화

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

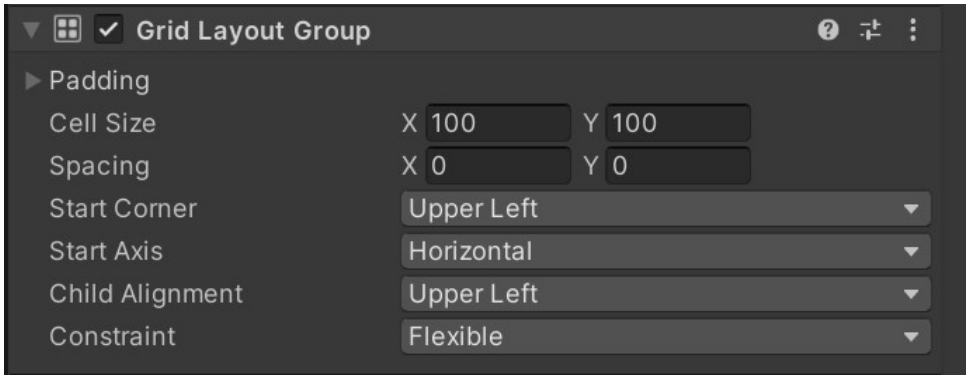


가능한 경우 Raycast Target 비활성화

레이아웃 그룹 사용 지양

레이아웃 그룹은 비효율적으로 업데이트되므로 필요할 때만 사용하세요. 하지만 콘텐츠가 동적이지 않은 경우 일반적으로 레이아웃 그룹을 사용하기보다는 비율 레이아웃에 앵커를 대신 사용하는 것이 가장 좋습니다. 그 밖의 경우에는 UI 설정을 마친 후 커스텀 코드를 생성하여 Layout Group 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 지양

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 프로젝트 를 통해 실제 작동 모습을 확인하세요.

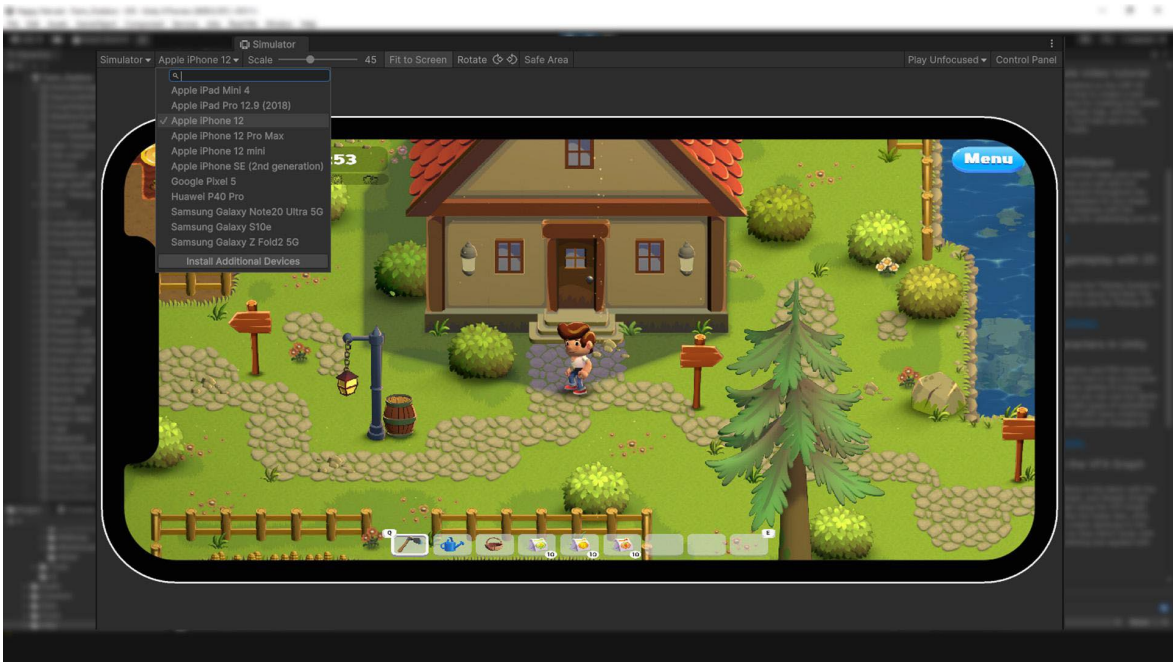
요소의 과도한 레이어링 지양

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.

다양한 해상도 및 종횡비 사용

현재 모바일 기기에서 사용 중인 해상도와 화면 크기가 매우 다양하므로, 각 기기에서 최상의 경험을 제공하는 대체 버전의 UI를 만드세요.

기기 시뮬레이터를 사용하여 지원되는 다양한 기기의 UI를 미리 볼 수 있습니다. XCode와 Android Studio 에서 가상 기기를 만들 수도 있습니다.



기기 시뮬레이터를 사용하여 다양한 화면 형식 미리 보기

전체 화면 UI 사용 시 기타 요소 모두 숨기기

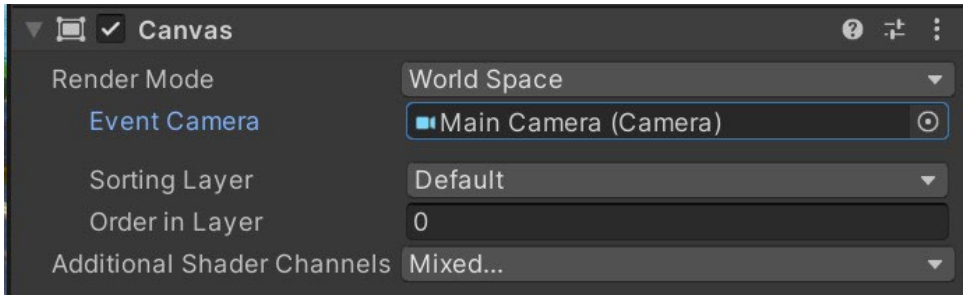
일시 중지 화면이나 시작 화면이 씬의 나머지 부분을 모두 가리는 경우, 3D 씬을 렌더링하는 카메라를 비활성화합니다. 마찬가지로 맨 위쪽 캔버스에 가려진 배경 캔버스 요소도 모두 비활성화합니다.

60fps에서는 업데이트할 필요가 없으므로 전체 화면 UI 사용 시에는 **Application.targetFrameRate**를 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 **Render Mode**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

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 레이아웃을 직접 계산하는 오버헤드를 피하고 일관되면서도 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 구독해야 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.

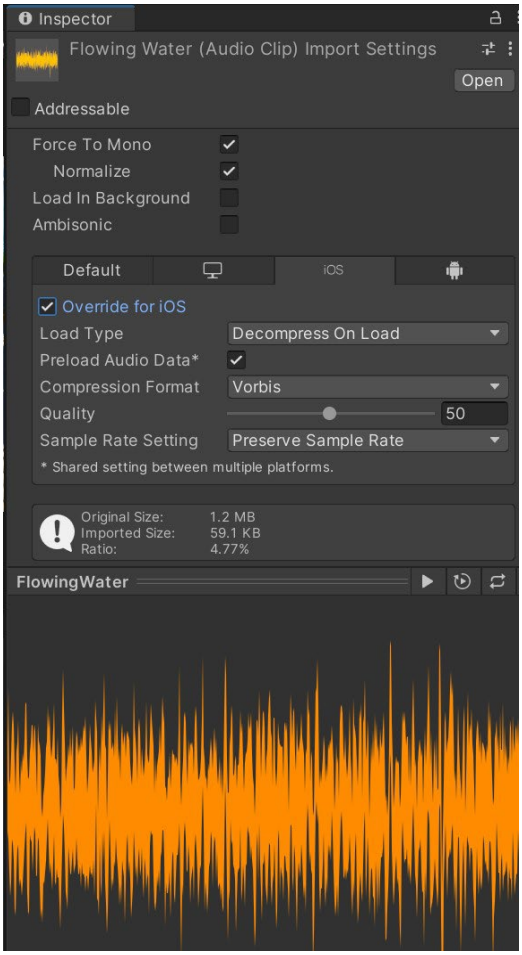
타겟 플랫폼에서 테스트

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 성능을 낮추는 요소가 아니지만,
최적화를 통해 메모리를 절약할 수 있습니다.



AudioClip의 Import Settings 최적화

가능한 한 사운드 클립을 모노로 만들기

3D 입체 음향을 사용한다면 사운드 클립을 모노(단일 채널)로 작성하거나 Force To Mono 설정을 활성화하세요. 그렇지 않으면 런타임 시 위치와 관련하여 사용되는 멀티채널 사운드는 모노 소스로 축소되므로 CPU 비용을 늘리고 메모리 낭비를 초래합니다.

압축되지 않은 원본 WAV 파일을 소스 에셋으로 사용

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

클립을 압축하고 압축 비트레이트 낮추기

압축을 통해 클립의 크기와 메모리 사용량을 줄이세요.

— 대부분의 사운드에 **Vorbis**(반복 재생 목적이 아닌 경우에는 **MP3**)를 사용하세요.

— 자주 사용하는 짧은 소리(예: 발걸음 소리, 총소리)에 **ADPCM**을 사용하세요. 이렇게 하면 압축되지 않은 PCM에 비해 파일이 줄어들지만 재생 중 디코딩 속도는 빨라집니다.

모바일 기기에서의 음향 효과는 최대 22,050Hz로 설정해야 합니다. 낮은 설정을 사용해도 최종 품질에 미치는 영향은 보통 미미하므로 직접 듣고 판단하는 것이 좋습니다.

적절한 로드 유형 선택

클립 크기에 따라 설정이 달라집니다. 자세한 내용은 다음 표를 참조하세요.

클립 크기	사용 예시	로드 유형 설정
작음 (< 200KB)	시끄러운 음향 효과(발걸음 소리, 총소리), UI 사운드	**Decompress on Load**를 사용합니다. 사운드를 16비트 PCM 오디오 원시 데이터로 압축 해제하느라 약간의 CPU 리소스를 사용하지만, 런타임에서 가장 성능이 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. 메모리 사용량 감소가 가장 중요하다면 **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 200KB 이므로 매우 큰 AudioClip에만 적합합니다.

메모리에서 음소거된 AudioSource 언로드

음소거 버튼을 구현할 때 음량을 그냥 0으로 설정하지 마세요. 플레이어가 해당 버튼을 자주 켜거나 끌 필요가 없다면 AudioSource 컴포넌트를 삭제하여 메모리에서 언로드할 수 있습니다.

Sample Rate Setting 사용

Sample Rate Setting을 Optimize Sample Rate 또는 Override Sample Rate로 설정하세요.

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

애니메이션

Unity에서 애니메이션 작업 시 유용한 팁은 다음과 같습니다. 애니메이션 시스템에 관한 종합 가이드가 필요하다면 Unity 애니메이션 완벽 가이드 무료 전자책을 다운로드하세요.

휴머노이드 릭 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 때도 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.

— 제네릭 릭의 경우 루트 모션을 사용하는 편이 반대의 경우보다 더 많은 리소스를 소모합니다. 애니메이션에서 루트 모션을 사용하지 않는다면 루트 뼈대를 지정하지 마세요.

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.

가능하다면 애니메이션 시스템을 아예 사용하지 않는 것도 고려해 보세요. easing 함수 를 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 최적화되어 있으며, 일반적인 커브보다 적은 리소스를 소모합니다.

시야에 들어올 때에만 업데이트

애니메이터의 Culling Mode를 **Based on Renderers**로 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 제외하면 애니메이션 계층 구조는 같은 부모를 공유하지 않아야 합니다. 이렇게 분리하면 애니메이션 결과를 게임 오브젝트에 다시 쓸 때 성능에 큰 영향을 주는 스레딩 문제를 방지할 수 있습니다.

바인딩 비용 최소화

애니메이션 시스템에서는 바인딩 연산에 따른 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.

복합적인 계층 구조에서 컴포넌트 기반 제약 사용 지양

구조가 복잡한 캐릭터 등 계층 구조가 복합적인 요소에는 성능이 저하될 수 있으므로 컴포넌트 기반 제약을 사용하지 않는 것이 좋습니다.

애니메이션 리깅의 성능 오버헤드 고려

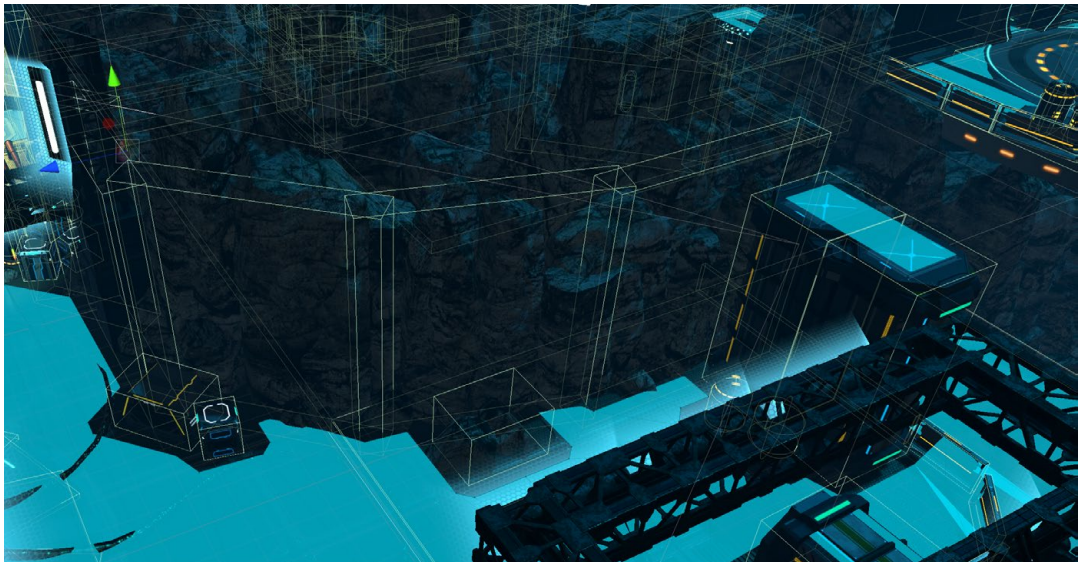
애니메이션 리깅을 사용하는 경우, 각 제약에 따라 추가되는 성능 오버헤드에 주의하세요. 휴머노이드 모델을 작업할 때는 이를 고려하는 것이 중요합니다. 가능한 한 휴머노이드 릭의 빌트인 IK(역운동학) 패스를 사용하여 성능을 향상하세요.

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 Unity의 빌트인 물리(NVIDIA PhysX)로 매끄러운 플레이를 만들어 보세요.

콜라이더 단순화

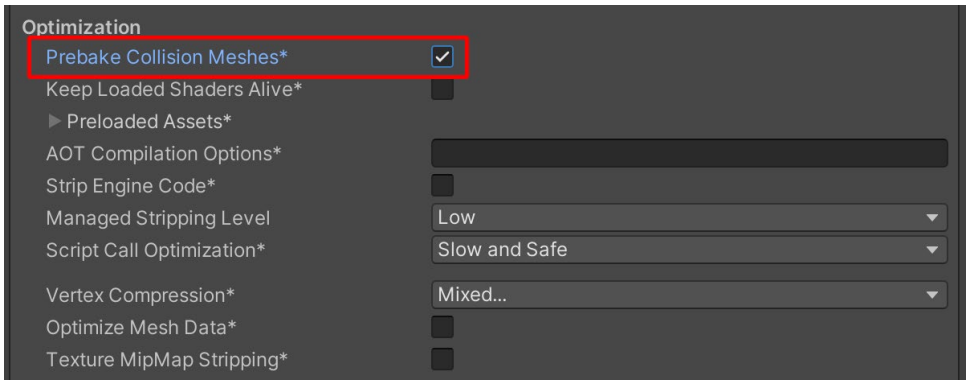
메시 콜라이더는 많은 리소스를 요합니다. 복잡한 메시 콜라이더를 기본 또는 단순화된 메시 콜라이더로 대체하여 원래 모양을 대략적으로 표현하세요.



콜라이더에 기본 또는 단순화된 메시 사용

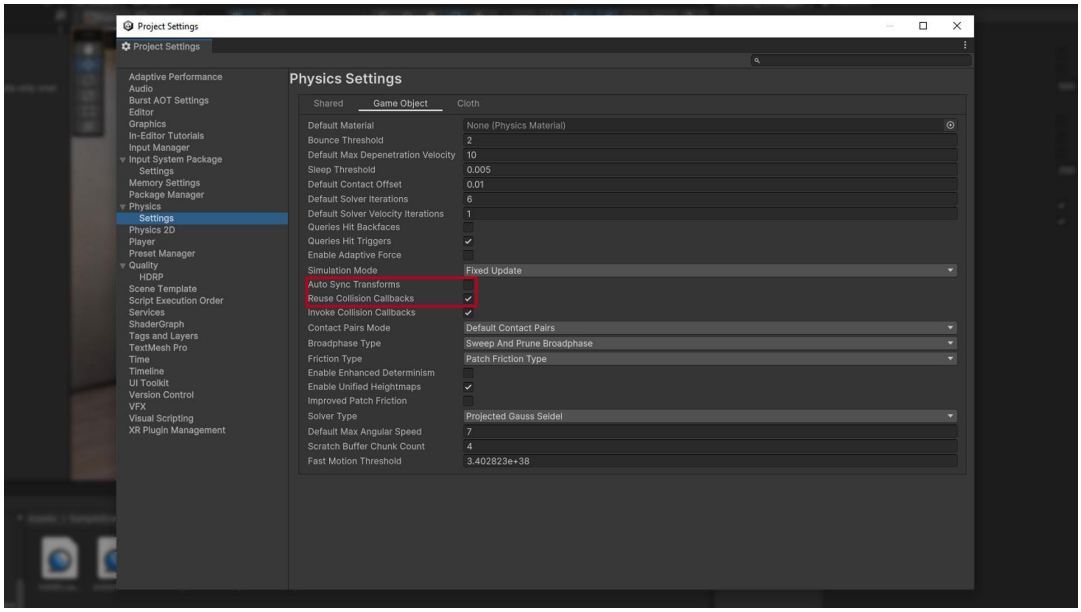
설정 최적화

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.

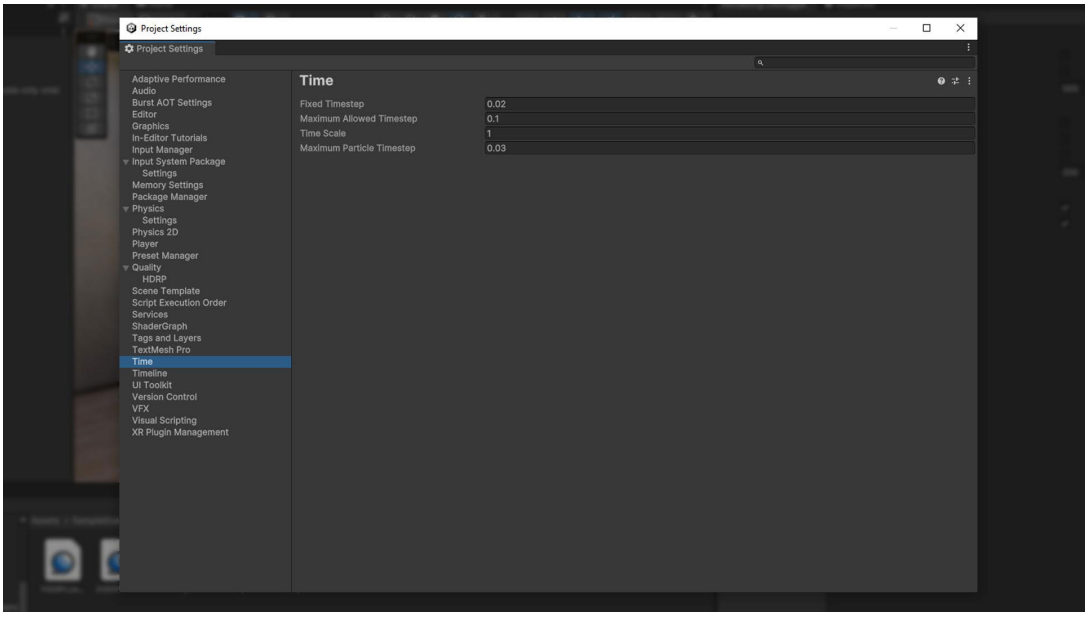


물리 프로젝트 설정을 수정하여 성능 향상

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 게임에 끊김 현상이 발생합니다.

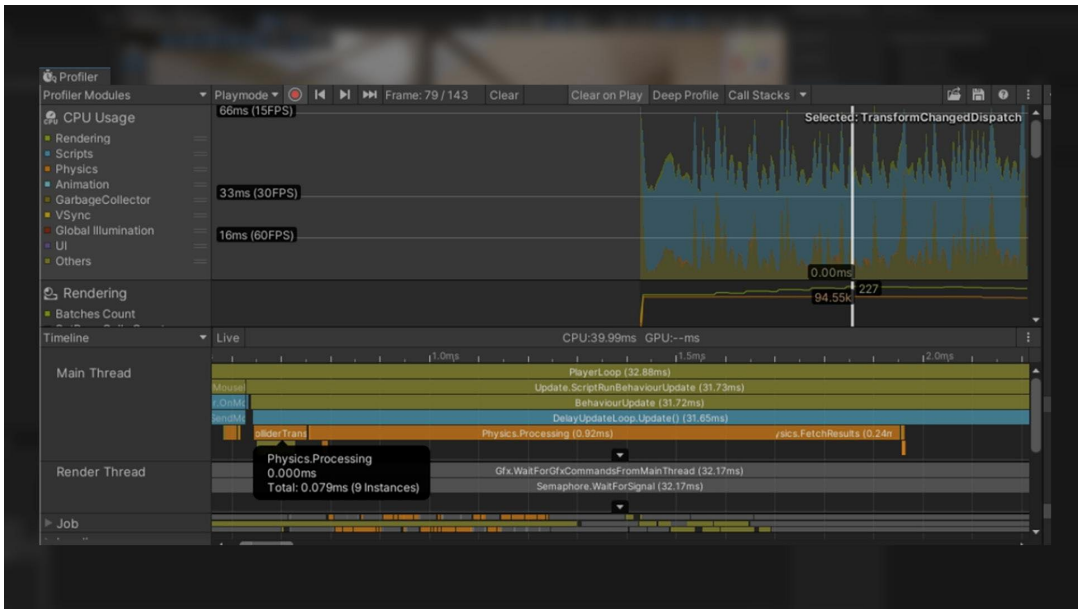
물리 관련 성능 문제를 방지하려면 다음을 수행하세요.

— 시뮬레이션 빈도를 줄입니다. 저사양 플랫폼의 경우, Fixed Timestep을 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 Timestep을 줄입니다. 0.1초 등의 더 작은 값을 사용하면 물리 시뮬레이션의 정확도는 조금 낮아질 수 있지만, 한 프레임 내에서 일어나는 물리 업데이트 횟수를 제한할 수도 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. 이 방법으로 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

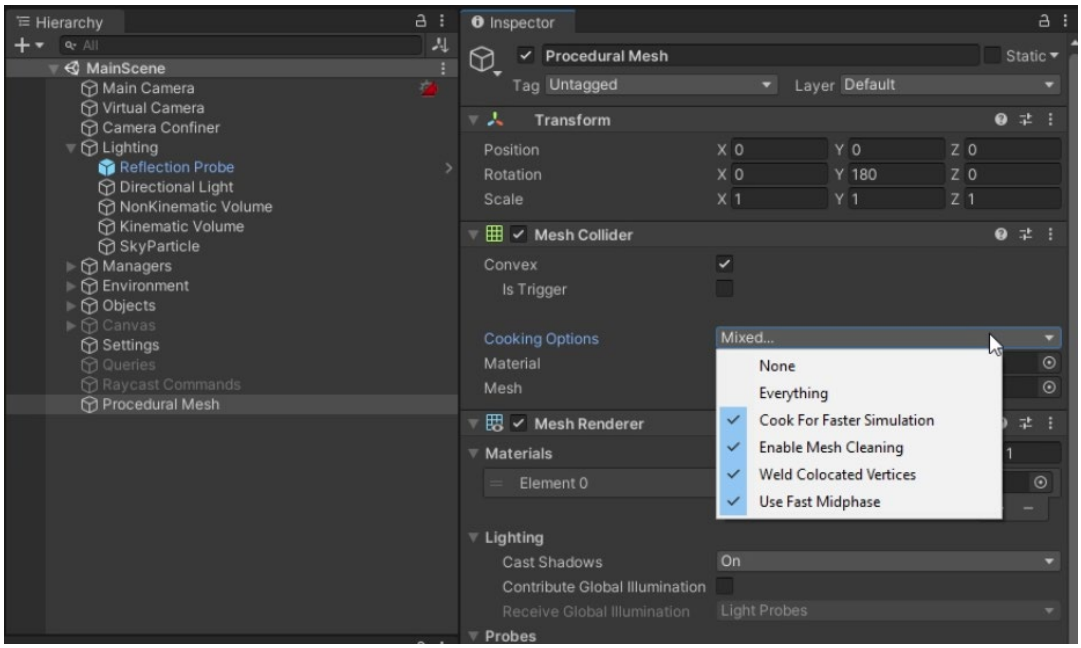
MeshCollider의 CookingOptions 수정

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 CookingOptions에서 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 단계(Mid-phase)에 PhysX 4.1에서 더 빠른 알고리즘으로 전환하게 되기 때문에, 물리 쿼리에 따라 교차할 가능성이 있는 소규모 삼각형 세트의 범위를 좁힐 수 있습니다.



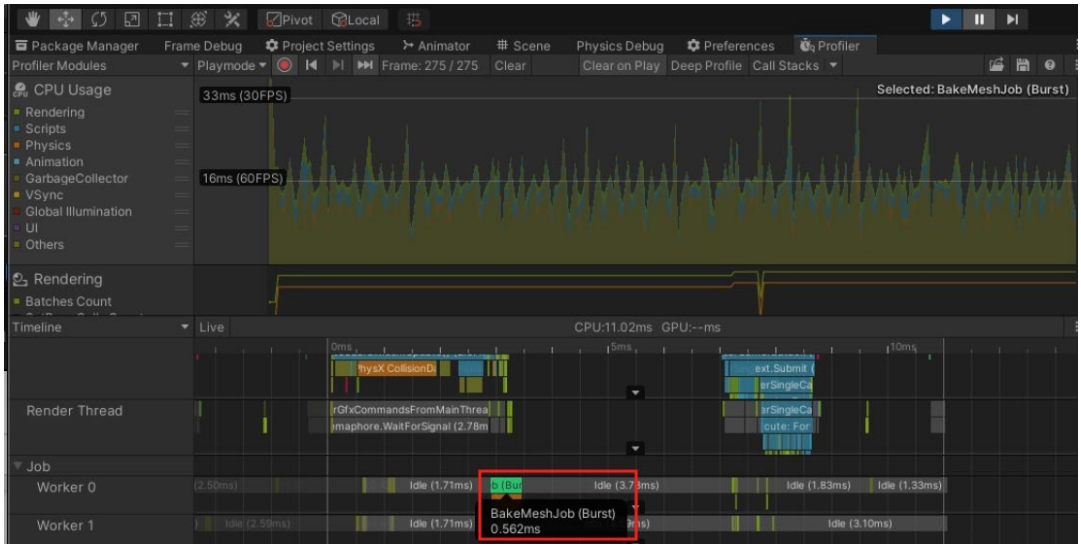
메시에 대한 쿠킹 옵션

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.

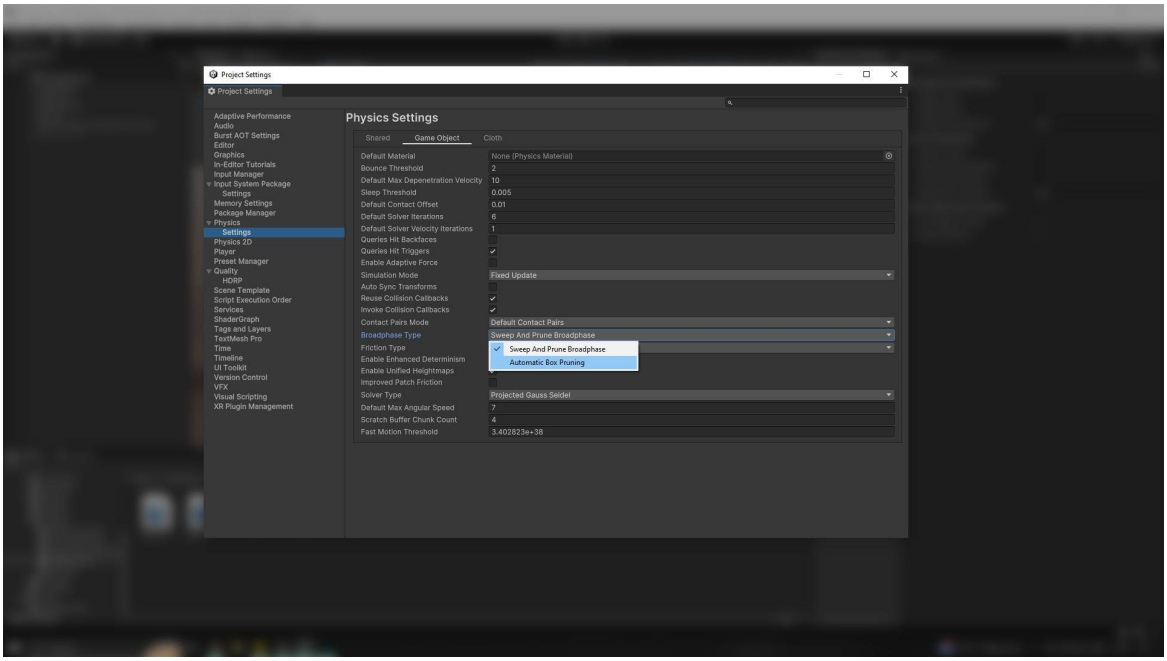
— 먼저 **광역 단계(Broad-phase)**에서 스윕 및 프룬(Sweep And Prune) 알고리즘을 사용하여 잠재적 충돌을 파악합니다.

— 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.

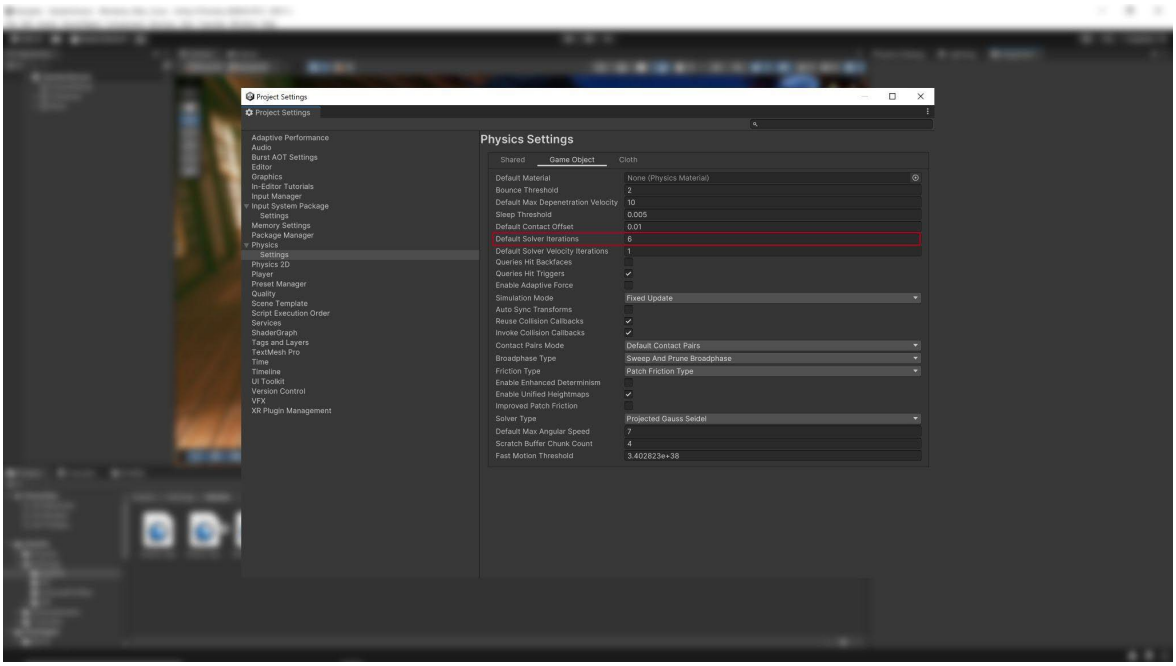
Multibox Pruning Broadphase를 사용하면 월드 경계와 그리드 셀 수를 수동으로 지정할 수 있으며, Automatic Box Pruning은 월드 경계와 그리드 수를 자동으로 계산합니다.



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 대기합니다. 자동 동기화를 활성화하면 트랜스폼 또는 그 자식의 모든 리지드바디나 콜라이더가 물리 엔진과 자동으로 동기화됩니다.

수동 동기화 시점

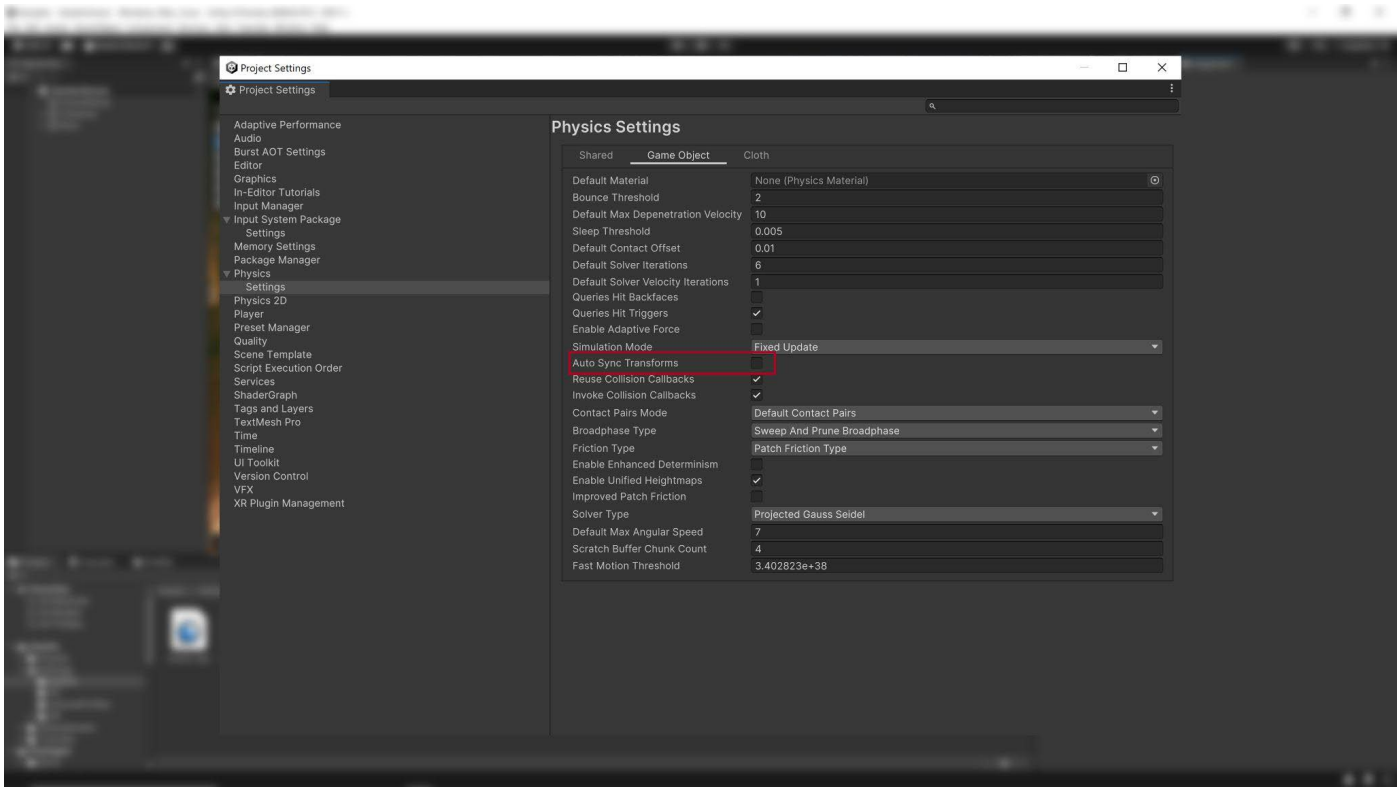
autoSyncTransforms가 비활성화된 경우, Unity는 FixedUpdate의 물리 시뮬레이션 단계 전 또는 Physics.Simulate로 분명히 요청했을 때만 트랜스폼을 동기화합니다. 트랜스폼이 변경되고 물리 업데이트가 진행되기 전에 물리 엔진에서 직접 읽어오는 API를 사용한다면 추가로 동기화가 필요할 수 있습니다. **Rigidbody.position**에 액세스하거나 **Physics.Raycast**를 수행하는 것이 그러한 경우입니다.

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. 권장되는 베스트 프랙티스는 다음과 같습니다.

— **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. 전역적으로 autoSyncTransforms를 활성화하는 방식보다 효율적입니다.



Auto Sync Transform을 비활성화한 상태로 Unity에서 씬 프로파일링

컨택트 배열 사용

컨택트 배열은 충돌 데이터(컨택트)를 배열 형식으로 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 활용 사례에서 C# 잡 시스템과 함께 사용할 수 있습니다.

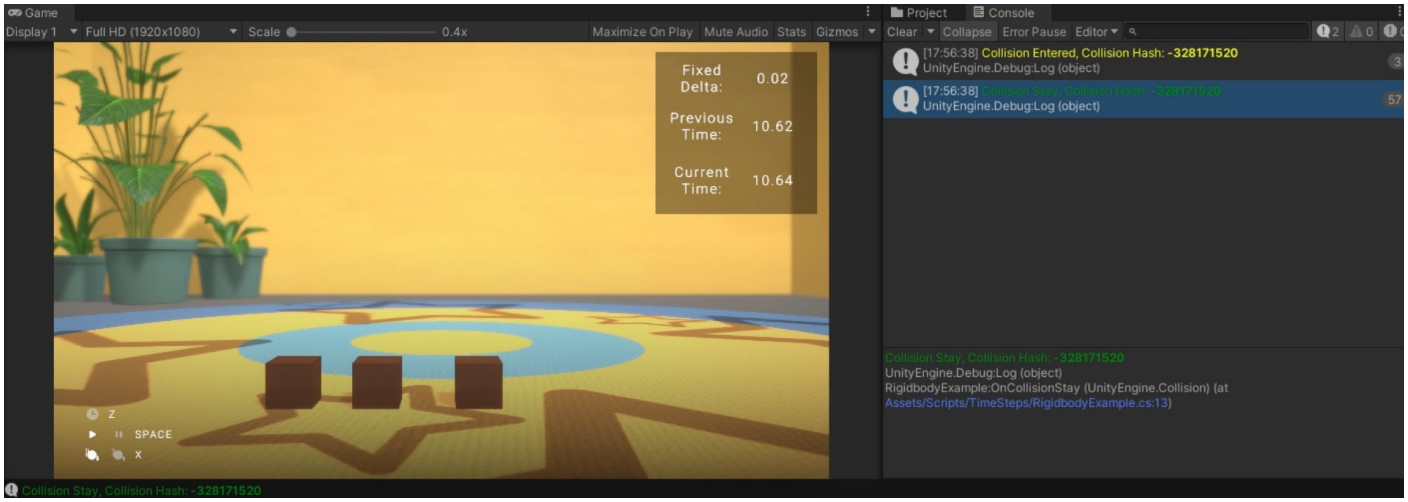
충돌 콜백 재사용

일반적으로 컨택트 배열이 훨씬 빠르므로 충돌 콜백을 재사용하는 것보다 권장되는 방식이지만, 특별한 목적이 있다면 다음과 같은 내용을 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 성능을 향상하려면 항상 Reuse Collision Callbacks를 활성화하는 것이 좋습니다. 코드가 개별 Collision 클래스 인스턴스에 의존하여 개별 필드에 저장할 수 없는 레거시 프로젝트에서만 이 기능을 비활성화해야 합니다.



Unity 콘솔(Console)의 Collision Entered 및 Collision Stay에는 하나의 충돌 인스턴스가 있습니다.

정적 콜라이더 이동

정적 콜라이더는 Collider 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 Rigidbody 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 **Static**으로 설정하세요. Collider 2D에 Rigidbody 2D가 있으면 시뮬레이션이 빨라집니다. 런타임에 콜라이더 2D 그룹을 이동하거나 재설정해야 하는 경우, 각 게임 오브젝트를 개별적으로 이동하는 것보다 모든 게임 오브젝트를 숨겨진 부모 리지드바디 2D 하나의 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 수집하는 경우, 대신 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

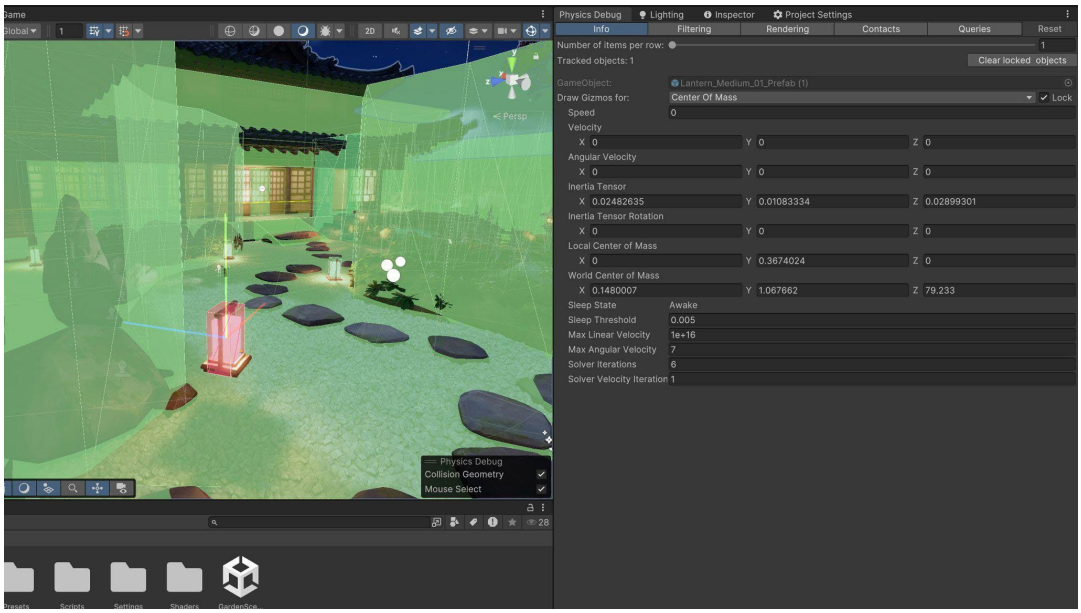
Physics.Raycast로 레이캐스트 쿼리를 실행할 수 있습니다. 그러나 10,000개의 에이전트에 대한 가시선을 계산하는 등 레이캐스트 연산이 많은 경우 CPU 시간이 많이 소요될 수 있습니다.

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 시각화하는 물리 디버거

자세한 내용은 물리 디버거 기술 자료를 참조하시기 바랍니다.

워크플로 및 협업

버전 관리를 사용하는 이유

Unity를 이용한 애플리케이션 제작은 까다로운 작업이며 많은 개발자가 참여하는 경우가 많습니다. 따라서 프로젝트를 팀 협업에 적합하게 설정하세요.

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.

Unity Version Control

UVCS(Unity Version Control)는 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.

UVCS에서 제공하는 기능은 다음과 같습니다.

- 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 사용하여 개발을 중앙 집중화할 수 있습니다. 특히 아티스트는 Gluon 애플리케이션을 통해 개발 팀과 아트 팀의 긴밀한 협업을 촉진하는 사용자 친화적인 워크플로의 장점을 크게 활용할 수 있습니다. 이 애플리케이션을 사용하면 복잡한 전체 프로젝트 저장소 구조에 신경 쓸 필요 없이 관리하려는 파일만 간단하게 확인할 수 있습니다. 간소화된 워크플로를 제공할 뿐만 아니라 에셋 버전을 간단하게 비교하고 통합된 버전 관리 환경에 쉽게 도움을 주는 툴도 제공합니다.

원활한 버전 관리 병합을 위해 **에디터 설정**에서 **Asset Serialization Mode**가 **Force Text**로 설정되도록 하세요.

Version Control 설정에서 외부 버전 관리 시스템(예: Git)을 사용하고 있다면 **Mode**가 **Visible Meta Files**로 설정되어 있어야 합니다.

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

게임 개발자를 위한 버전 관리 및 프로젝트 구성 베스트 프랙티스 전자책에서 Unity VCS와 일반적인 버전 관리 및 프로젝트 구성 베스트 프랙티스에 대해 자세히 알아보세요.

대형 씬의 분할

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

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

사용하지 않는 리소스 제거

타사 플러그인이나 라이브러리에 사용되지 않는 에셋이 포함되어 있는지 확인하세요. 많은 경우 테스트 에셋이나 스크립트가 내장되어 있으며, 제거하지 않으면 빌드에 포함됩니다. 프로토타이핑 이후에 남은 불필요한 리소스를 제거하세요.

이 전자책에서 설명한 모든 최적화 팁은 플랫폼과 관계없이 게임 제작에 도움이 될 것입니다. 다음 섹션에서는 XR과 웹 전용 최적화 팁을 살펴보겠습니다.

Unity Web 빌드 플랫폼 전용 팁

웹 브라우저 내에서 애플리케이션을 구동하는 데에는 태생적인 한계와 어려움이 따르므로, Unity Web 빌드는 고유의 접근 방식으로 최적화해야 합니다. 웹 빌드는 다양한 기기와 브라우저에서 성능, 로딩 시간, 호환성의 균형을 맞춰야 합니다. 이 섹션에서는 Unity Web 프로젝트에 적합한 주요 전략과 베스트 프랙티스를 살펴봅니다. 에셋 관리에서 메모리 최적화, 빌드 크기 축소, 사용자 경험 개선까지 모두 다루는 팁을 참고하여 여러 환경에서 원활한 경험을 제공하는 고성능 웹 애플리케이션을 제작해 보세요.

프레임 속도

— Unity Web 빌드의 경우 **Application.targetFrameRate**를 프로젝트 설정 기본값인 -1로 유지하세요. targetFrameRate를 -1로 설정하면 브라우저가 프레임 속도를 브라우저의 '애니메이션 속도'에 맞춥니다. 즉, 최대한 빠른 렌더링 속도를 제공하며, 이는 Firefox나 Chrome을 비롯한 일반적인 브라우저에서 디스플레이의 네이티브 새로고침 속도와 같습니다. 하지만 Safari에서는 새로고침 속도의 상한이 항상 60fps입니다.

— 프로젝트의 기본 설정을 변경했다면 새 C# 스크립트를 생성하거나 기존 스크립트를 열어 목표 프레임 속도를 설정할 수 있습니다. 일반적으로 게임을 초기화하는 스크립트나 중앙 게임 관리자 스크립트에 작성합니다.

```
using UnityEngine;

public class FrameRateManager : MonoBehaviour
{
    void Start()
    {
        // Set the target frame rate to -1 to let the browser control the frame rate
        Application.targetFrameRate = -1;
    }
}
```

Unity Web의 퍼블리싱 설정

압축

압축을 사용하면 사용자가 브라우저에서 다운로드하는 파일의 크기를 크게 줄일 수 있습니다. 파일이 작을수록 적은 양만 빠르게 다운로드할 수 있고 대역폭 소모량이 줄어듭니다.

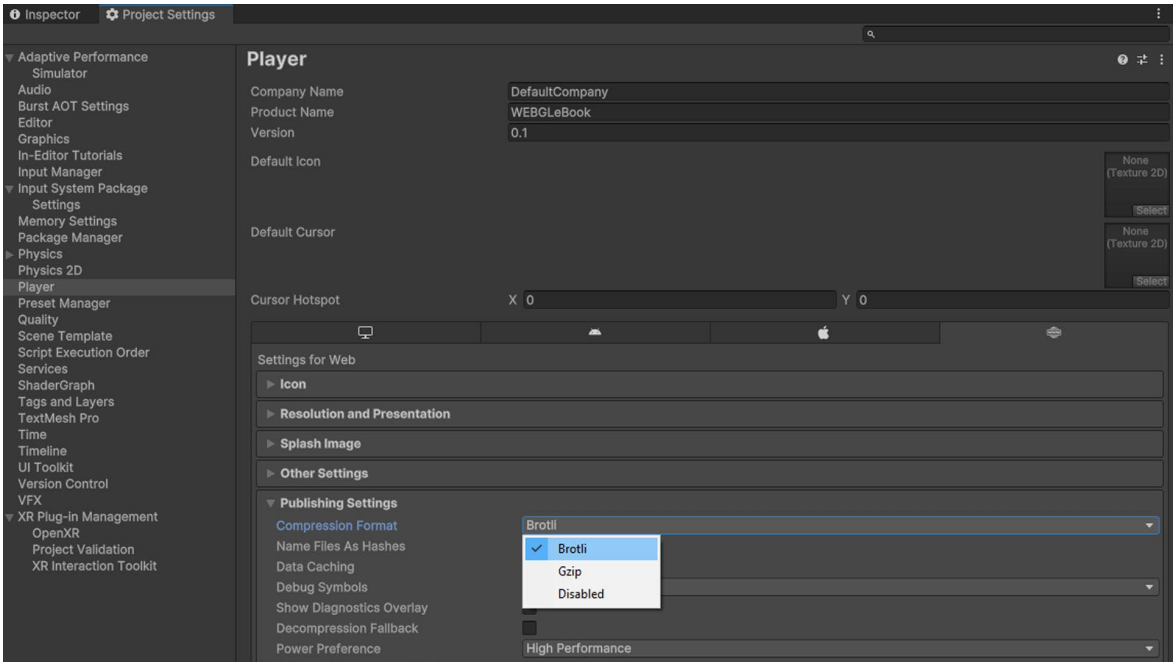
지원하는 압축 방식은 다음과 같습니다.

Brotli: gzip보다 압축비가 높으므로 파일 크기를 더 줄이고 로딩 시간을 단축할 수 있습니다. 최신 브라우저에서 지원되며, 웹 사이트가 안전한 https:// URL 또는 http://localhost/ 테스트 URL로 연결되어야 합니다.

Gzip: 지원 범위가 넓으면서도 효과적인 방식입니다. 콘텐츠가 안전하지 않은 http:// 서버에서 제공되거나, Brotli 방식으로 압축된 콘텐츠를 제공할 준비가 되지 않은 웹 서버에서 호스팅되거나, 아직 Brotli와 호환되지 않는 복잡한 CDN 로드 밸런싱이나 캐싱 인프라를 사용하는 경우 이 방식을 사용하세요.

미압축: 파일이 너무 커지고 로딩 시간이 길어지므로 일반적으로 정식 제작에는 권장되지 않습니다. 웹 서버에서 사전 압축된 Brotli 또는 Gzip 콘텐츠를 지원하지 않고 즉석 압축 캐시만 사용하도록 설정된 경우에만 이 방법을 사용하세요.

일반적으로 Unity Web 빌드를 퍼블리시할 때는 압축비, 브라우저 지원, 성능이 우수한 Brotli가 가장 좋은 압축 방식입니다.



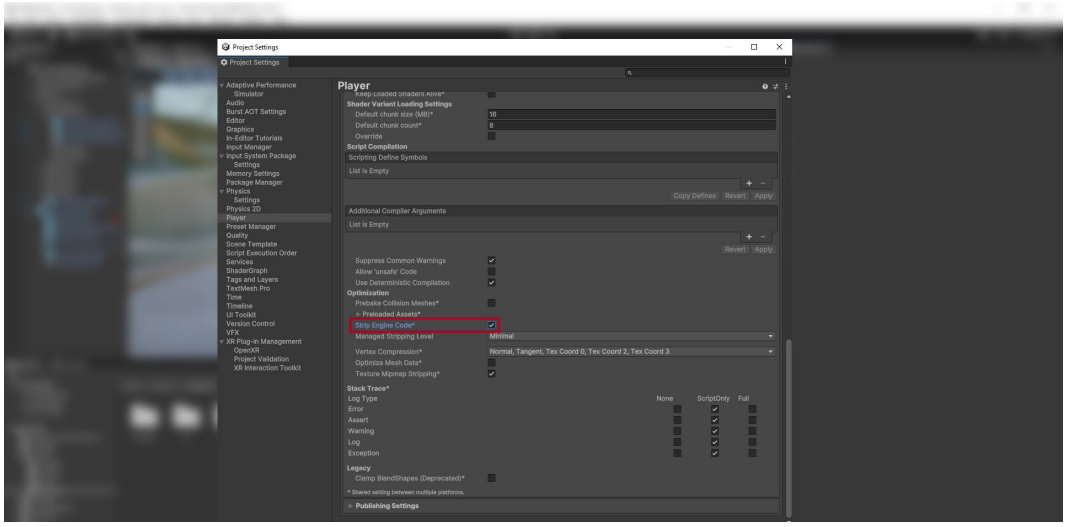
Publishing Settings에서 Brotli 선택

사이트 시작 시간을 단축하려면 **Decompression Fallback** 설정을 **Disabled**로 설정하는 것이 좋습니다. 또한 페이지를 호스팅하는 웹 서버가 사전 압축된 Unity 콘텐츠를 제공하도록 설정되어야 합니다.

Decompression Fallback을 활성화하면 모바일 브라우저의 배터리 사용량에 부정적인 영향을 미칠 수 있으며, 게임을 시작하는 시간이 지연됩니다. 이 기술 자료 페이지의 웹 서버 설정 가이드라인을 따르세요.

엔진 코드 제거

효율적인 빌드를 위해 플레이어 설정에서 확인할 또 다른 설정은 **Player settings > Other Settings** 패널의 **Strip Engine Code 활성화**입니다.



Strip Engine Code 활성화

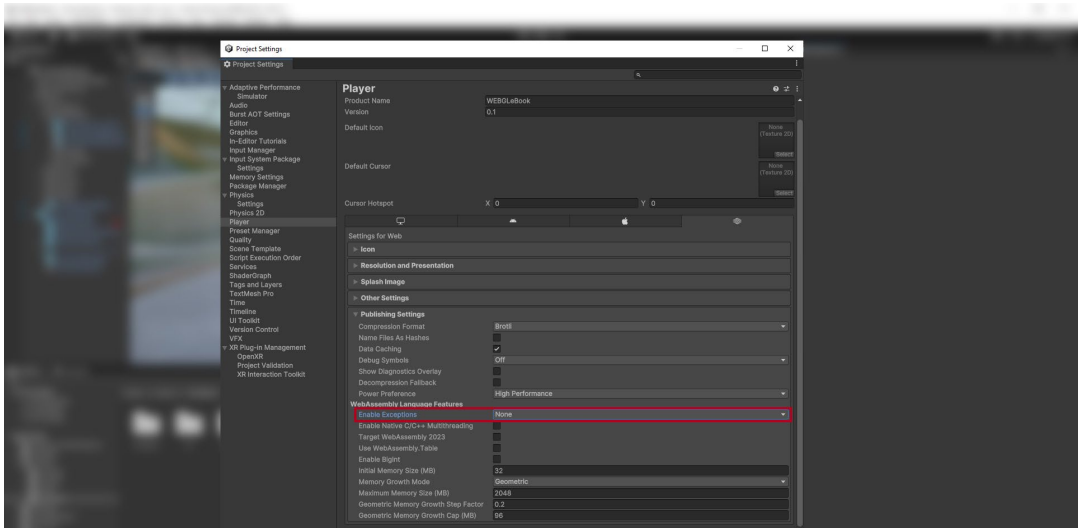
특히 Unity Web 프로젝트의 경우 효율적인 빌드를 위해 Strip Engine Code를 활성화하는 것이 좋습니다. 이 기능을 활용하면 사용되지 않는 엔진 코드를 제거해 최종 빌드의 크기를 크게 줄이므로 로딩 시간이 짧아지고 성능이 향상됩니다.

Enable Exceptions를 None으로 설정

예외를 처리하려면 빌드에 코드를 추가해야 합니다. 예외 처리를 비활성화하면 런타임 시 확인하는 과정과 예외 처리 코드를 덜 수 있습니다. 프로젝트에서 예외를 제어하여 처리하지 않고 모든 예외 발생 시 실행을 중단해도 된다면 예외 처리 지원을 비활성화하는 조치를 통해 코드 크기를 줄일 수 있습니다.

또는 애플리케이션에 예외 처리 지원이 필요하다면 일반적으로 예외 처리 코드 크기를 최적화해 주는 WebAssembly 2023을 활성화해 보세요.

빌드에서 예외 처리를 사용하지 않는다면 Unity Web 빌드 탭의 **Player Settings** 창에서 **Publishing Settings**를 펼치고 **Enable Exceptions**를 **None**으로 설정합니다.



Enable Exceptions를 None으로 설정

WebAssembly 2023 기능 세트 활용

게임이 Firefox 89 이상 버전, Chrome 91 이상 버전, Safari 16.4 이상 버전을 타게팅해도 된다면 WebAssembly 2023 기능 세트를 활성화하여 코드 크기를 줄이세요.

WebAssembly 2023은 JS BigInt, 대용량 메모리 연산, 비트래핑 float-to-int 변환, 부호 확장 연산자 및 고정 너비 SIMD 기능을 제공합니다. WebAssembly 로드맵 페이지(https://webassembly.org/features/)에서 어떤 브라우저가 어떤 WebAssembly 기능을 지원하는지 확인해 보세요.

코드 최적화 설정

최종 릴리스 버전의 경우 프로젝트 빌드 플랫폼 설정에서 Code Optimization:Disk Size with LTO(LTO 빌드의 경우 완료하는 데 오래 걸릴 수 있음)로 설정하고, 개발 중인 경우 Code Optimization: Disk Size for code size optimized builds로 설정하세요.

Unity Web 빌드 프로파일링

Unity의 프로파일링 툴 제품군뿐만 아니라 Chrome DevTools나 Firefox Profiler 같은 툴을 활용하여 프로파일링하고 성능을 분석할 수 있습니다.

Chrome DevTools

Chrome DevTools는 Google Chrome 브라우저에 내장된 종합적인 웹 개발 툴 세트입니다. 성능 프로파일링, JavaScript 디버깅, 네트워크 활동 분석, 렌더링 검사를 위한 기능을 제공합니다. Chrome DevTools를 활성화하는 기본적인 단계는 다음과 같습니다.

1. F12나 **Ctrl+Shift+I**(Windows/Linux), **Cmd+Option+I**(Mac)를 눌러 Chrome DevTools를 엽니다. 페이지를 오른쪽 클릭하고 **검사**를 선택해도 됩니다.
2. **성능** 탭으로 이동합니다. **기록** 버튼을 클릭하고 Unity Web 게임과 상호 작용하여 성능 데이터를 캡처합니다. **중지**를 클릭하여 기록을 종료한 다음 프레임 속도, CPU 사용량, 렌더링 성능을 중점에 두고 데이터를 분석합니다.
3. **네트워크** 탭으로 이동하고 Unity Web 게임을 다시 로드하여 모든 네트워크 요청을 캡처합니다. 타임라인, 요청 세부 사항, 로딩 시간을 검사하여 네트워크 관련 병목 현상이 발생하는지 확인합니다.
4. **소스** 탭에서 JavaScript 코드에 중단점을 설정하고 실행을 멈춰 변수를 검사합니다. 호출 스택과 범위 정보를 확인하여 코드를 디버깅하고 최적화합니다.
5. **콘솔** 탭에서 Unity Web 상태 기록을 확인하고 렌더링 문제를 디버깅합니다. 더 심층적으로 분석하려면 Unity Web Insight나 Unity Web Debugging과 같은 Unity Web 전용 툴과 확장 프로그램을 활용합니다.

XR 최적화 팁

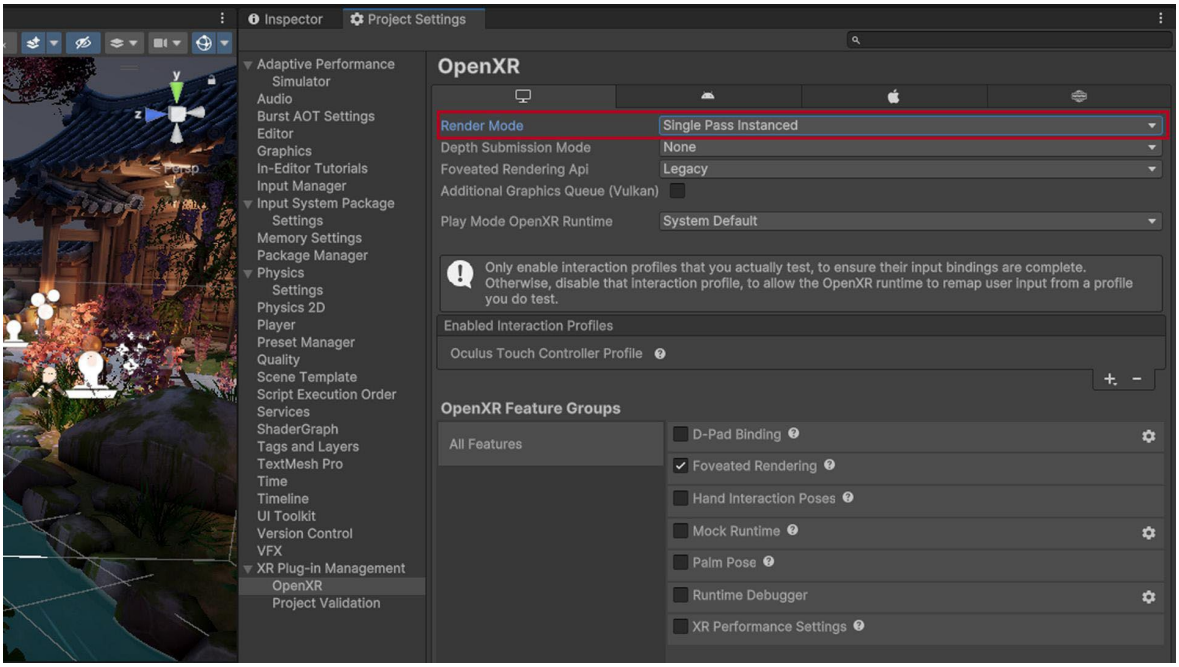
이 섹션에서는 Unity로 제작된 VR, AR, MR(통틀어 XR) 애플리케이션을 위한 최적화 팁을 살펴봅니다. 여기에서 설명할 많은 기법이 일반적으로 모바일 기기에도 적용되므로 이 가이드의 다른 부분에서도 이미 소개된 바 있지만, XR 앱에만 집중하려는 독자를 위해 이 섹션에 모았습니다.

여기에서 설명하는 기법을 활용하여 XR 애플리케이션을 효율적으로 실행해 보세요. 특히 높은 성능과 짧은 지연 시간을 바탕으로 몰입감을 유지하고 멀미를 방지해야 하는 VR 경험에 효과적입니다. 고해상도 3D 렌더링 환경과 반응형 상호 작용을 최적화해야만 물리적으로 편안하면서도 원활한 경험을 보장할 수 있습니다.

Unity의 XR 앱 개발 종합 가이드를 살펴보려면 Unity로 가상 및 혼합 현실 경험 제작 전자책을 다운로드해 보세요.

렌더 모드

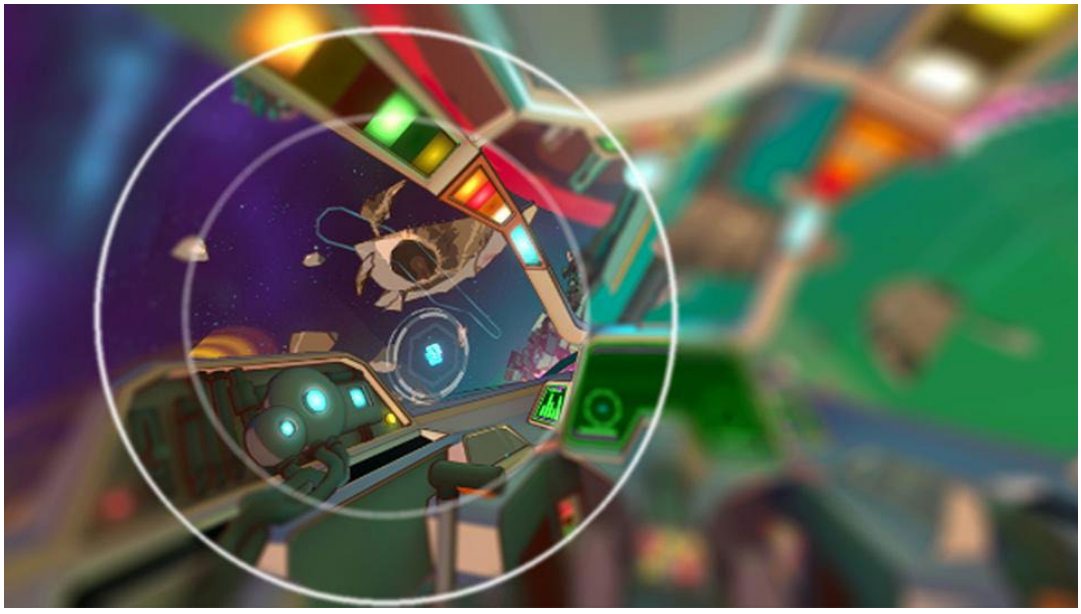
렌더 모드를 올바르게 설정하면 VR 게임의 성능이 크게 달라집니다. Unity의 OpenXR 플러그인을 사용하는 경우, **Project Settings** > **XR Plug-in-management** > **plugin provider**에 **Render Mode** 옵션이 있습니다. 드롭다운에서 **Single Pass Instanced**를 선택하세요. 인스턴싱을 활용하여 두 눈을 하나의 패스로 렌더링하는 모드로, 씬을 한 번 렌더링하고 두 눈에 대해 셰이더가 동시에 실행됩니다.



XR 애플리케이션을 개발하는 경우 Render Mode로 Single Pass Instanced를 선택

포비티드 렌더링

Unity 6에는 포비티드 렌더링 기능이 통합되었으며 PlayStation VR2를 포함하여 Oculus XR과 OpenXR이 지원됩니다. 포비티드 렌더링은 사람의 눈이 한 번에 작은 영역에만 집중하는 경향을 활용하는 VR용 최적화 기법입니다. 초점 영역은 고해상도로 렌더링하고 주변 영역은 저해상도로 렌더링하여 GPU 워크로드를 크게 줄일 수 있습니다. 포비티드 렌더링을 구현하면 성능을 향상하여 프레임 속도를 높이고 가장 중요한 부분의 비주얼 품질을 향상할 수 있습니다.



초점 영역을 고해상도로 표시하는 포비티드 렌더링

XR Interaction Toolkit 활용

Unity의 XR Interaction Toolkit 은 XR 프로젝트에서 입력 처리를 최적화하기 위한 탁월한 선택입니다. VR 및 AR 하드웨어와 효율적으로 작동하도록 설계된 사전 구성 컴포넌트와 상호 작용 시스템 세트를 제공합니다. 개발자가 이 툴킷으로 할 수 있는 일은 다음과 같습니다.

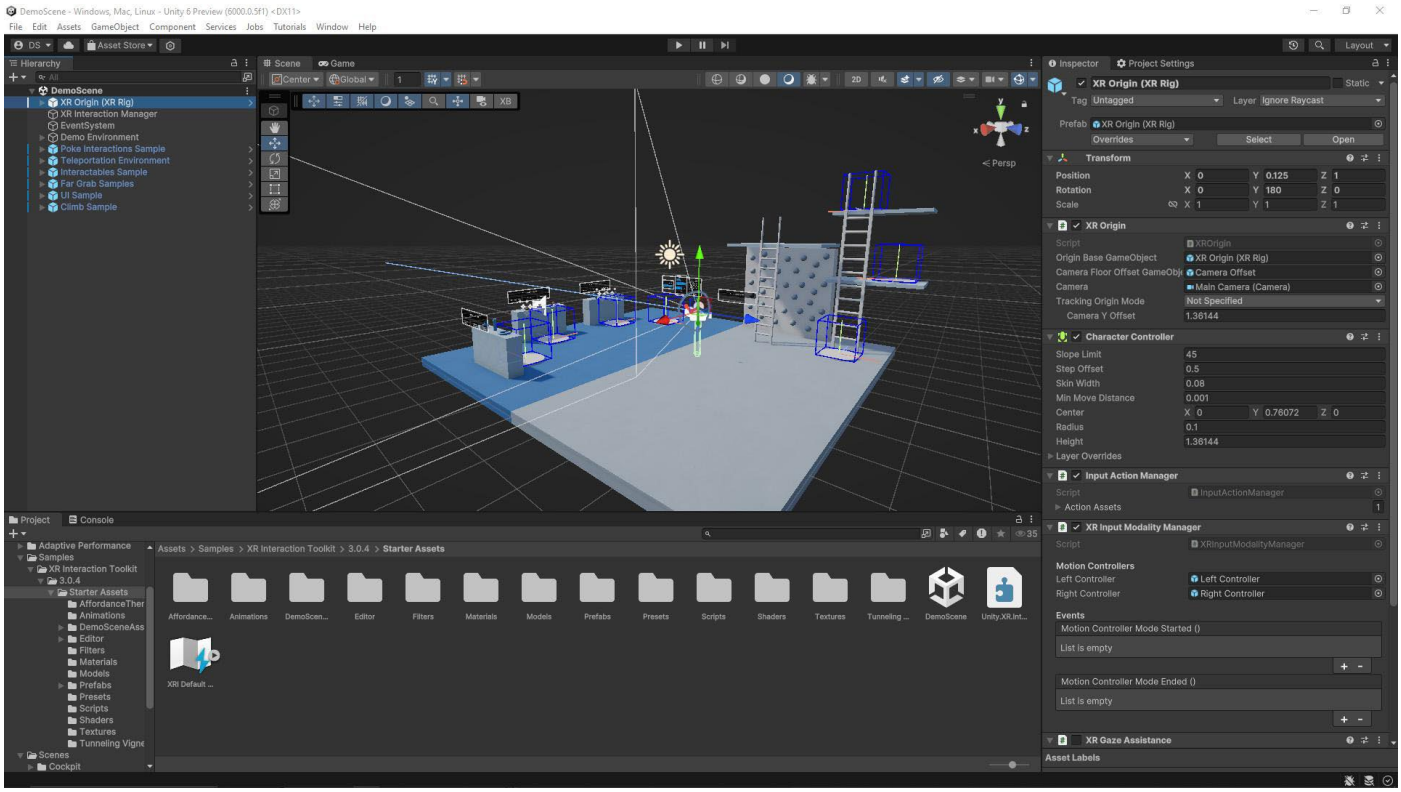
상호 작용 표준화: 빌트인 상호 작용 패턴을 사용하여 커스텀 코드를 줄이고 일관성을 높일 수 있습니다.

이벤트 기반 아키텍처 사용: 이벤트 기반 입력 처리를 활용하여 폴링을 최소화하고 성능을 향상할 수 있습니다.

사용 편의성 개선: 바로 사용할 수 있는 컴포넌트를 통해 개발 프로세스를 간소화할 수 있으므로 반복 작업과 최적화가 빨라집니다.

전반적으로 XR Interaction Toolkit을 활용하면 XR 애플리케이션에서 입력 처리를 간소화하고 최적화하며, 반응성과 사용자 경험을 개선할 수 있습니다.

이러한 전략을 활용하면 XR 애플리케이션에서 원활하고 반응성이 뛰어난 입력 처리 기능을 제공하고, 전반적인 사용자 경험을 개선할 수 있습니다.



XR Interaction Toolkit 샘플 씬

XR 최적화를 위한 성능 테스트

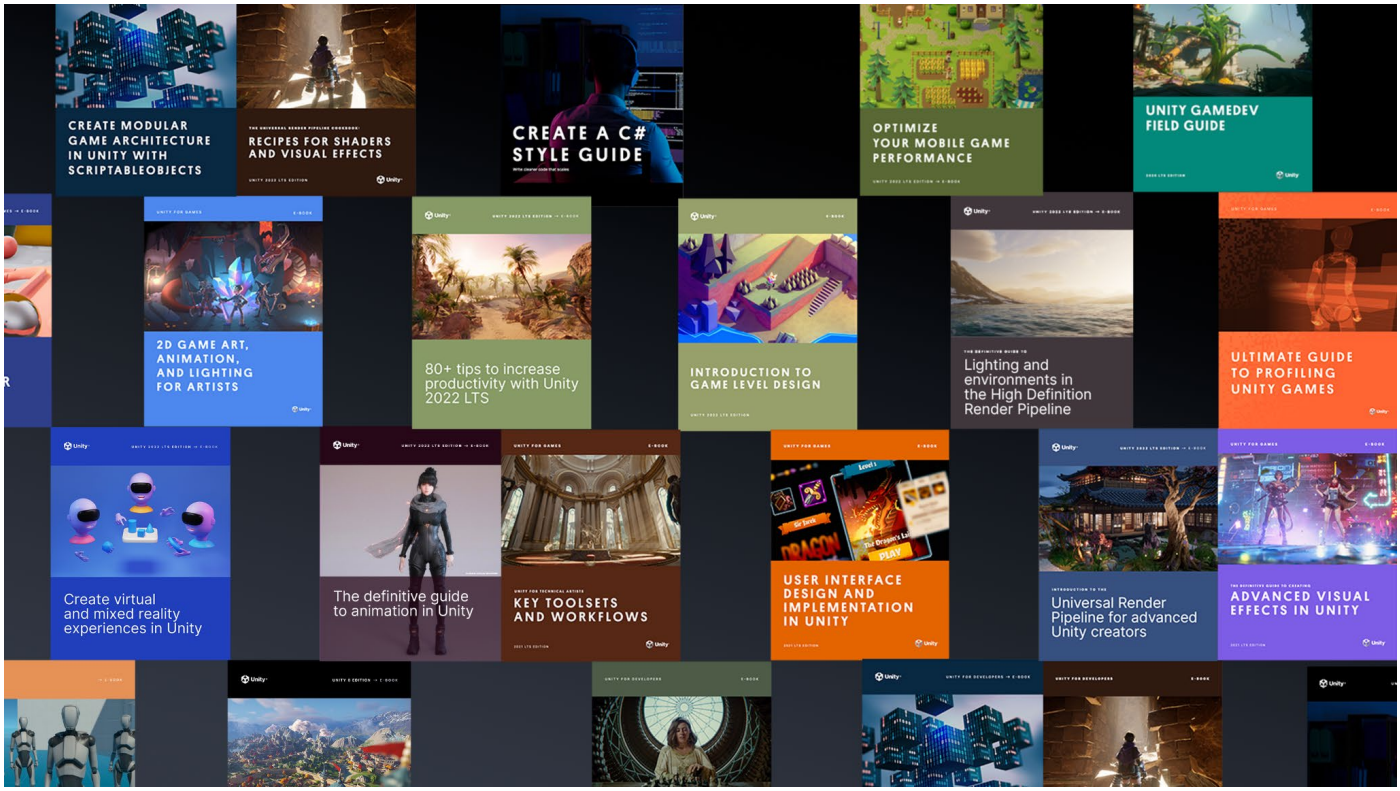
XR 애플리케이션에서 원활하고 몰입도 높은 경험을 제공하려면 성능 테스트가 필수적입니다.

다음과 같은 XR 전용 프로파일러를 사용할 수 있습니다.

— **Oculus Performance HUD(Head-Up Display):** 실시간 성능 지표 확인

— **SteamVR 성능 툴:** VR 애플리케이션 성능 분석

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 팁과 베스트 프랙티스, 새 소식은 Unity 블로그나 Unity 커뮤니티 포럼에서 확인할 수 있으며, Unity Learn이나 **#unitytips** 해시태그를 통해서도 확인할 수 있습니다.

unity.com/kr