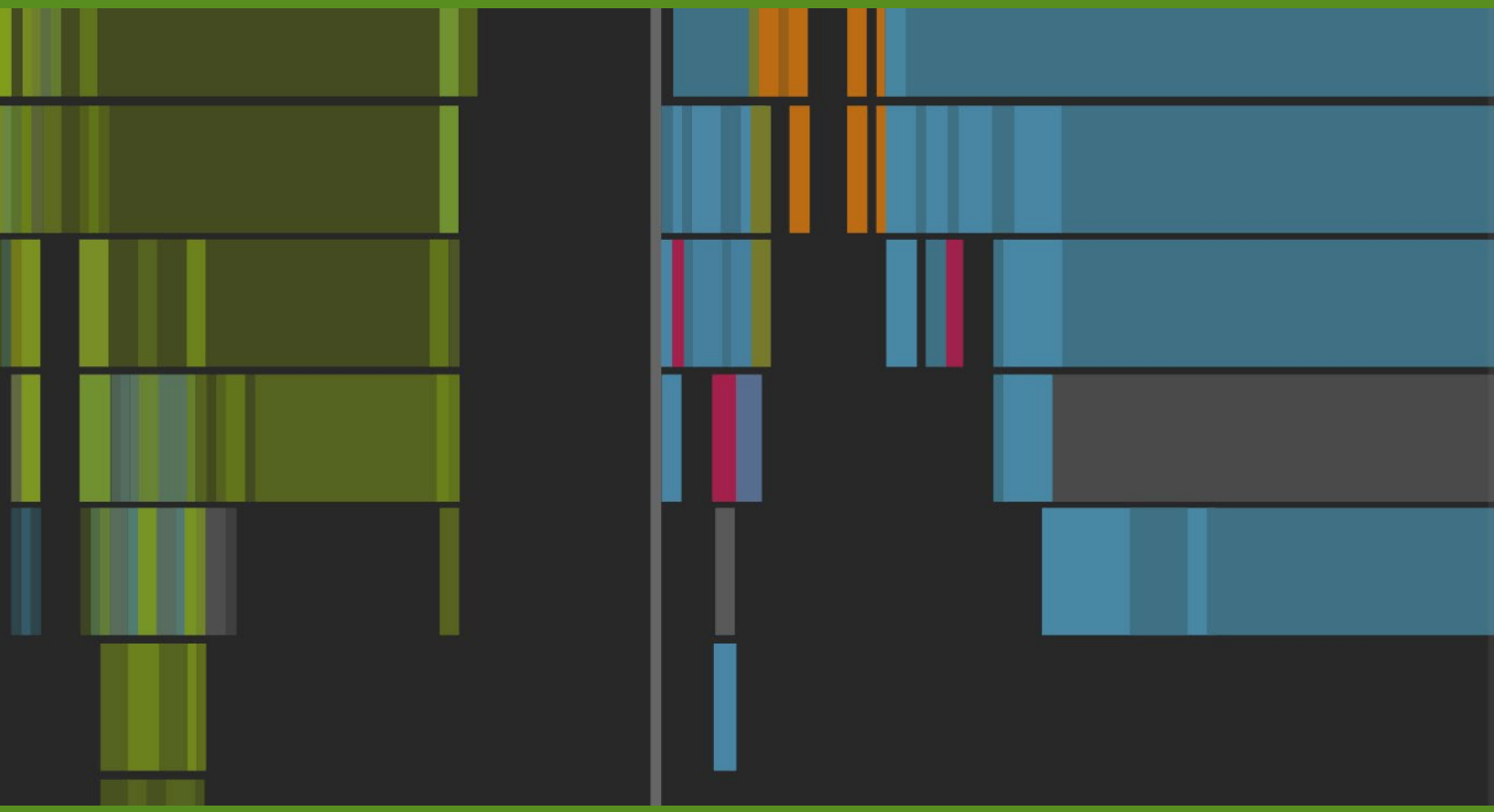
→ 전자책

Unity 게임 프로파일링 완벽 가이드

(Unity 6 에디션)



Unity®

목차

들어가는 말

다양한 기기와 플레이어를 대상으로 탁월한 게임 경험을 제공하려면, 반드시 원활한 성능이 뒷받침되어야 합니다. 유니티는 Unity 개발자들이 타겟 플랫폼용으로 제공되는 네이티브 프로파일링 툴과 함께 사용할 수 있는 다양한 프로파일링 및 메모리 관리 툴을 제공합니다.

이 가이드에서는 Unity에서 애플리케이션을 프로파일링하고, 메모리를 관리하고, 전력 소비를 최적화하는 방법에 관한 실질적인 가이드를 처음부터 끝까지 소개합니다.

본 전자책은 두 번째 에디션으로, Unity 6의 최신 기능과 커뮤니티 피드백에 따라 적용된 개선 사항을 반영하여 업데이트되었습니다.

Unity의 프로파일링 가이드는 다음과 같은 사내 Unity 전문가들과 외부 게임 개발자들의 협력을 바탕으로 제작되었습니다.

— 스티븐 카나반, 시니어 소프트웨어 개발 컨설턴트

— 션 더피, 소프트웨어 엔지니어 겸 게임 개발자

— 피터 홀, 소프트웨어 엔지니어링 시니어 매니저

— 토마스 크로그 야콥센, 콘텐츠 마케팅 관리 시니어 매니저

— 스티브 맥그릴, 소프트웨어 엔지니어

— 마틴 틸로 슈미츠, 시니어 소프트웨어 엔지니어

— 피터 해릿, Arm

Unity 6의 성능 최적화에 대해 다루는 다른 가이드로는 콘솔 및 PC 게임 성능 최적화와 Unity에서 모바일, XR, 웹 게임 성능 최적화가 있습니다.

프로파일링 기초

Unity에서 게임을 어떻게 프로파일링하는지 자세히 알아보기 전에 몇 가지 주요 개념과 프로파일링 원칙부터 간략하게 살펴보겠습니다.

간결하고 효과적인 코드를 사용하고 메모리 사용량을 최적화하면 다양한 사양의 기기에서 더 나은 사용자 경험을 제공할 수 있습니다. 이는 발열과 배터리 소모를 최적화하여 더 많은 저사양 기기 사용자들에게 도달하는 것부터, 플레이어의 만족도를 높이고, 궁극적으로 도입률과 리텐션을 높이는 것까지 모든 사항에 적용됩니다. 또한 배포 플랫폼 사양을 준수하기 위한 요구 사항이 될 수도 있습니다.

효율적인 게임 개발을 위한 '필수 요소'인 일관되고 포괄적인 프로파일링 워크플로는 다음과 같은 세 가지 간단한 절차로 시작됩니다.

— 주요 변경 사항을 적용하기 전에 프로파일링하여 기준을 설정합니다.

— 개발 중 프로파일링을 진행하여, 변경 사항으로 인해 성능 또는 메모리 할당량에 문제가 발생하지 않도록 추적하고 확인합니다.

— 개발 이후 프로파일링을 통해 변경 사항으로 원하는 효과를 구현했는지 입증합니다.

개발자에게 매우 유용한 툴인 프로파일러를 사용하면 코드의 메모리 사용량과 성능 병목 현상을 식별할 수 있습니다.

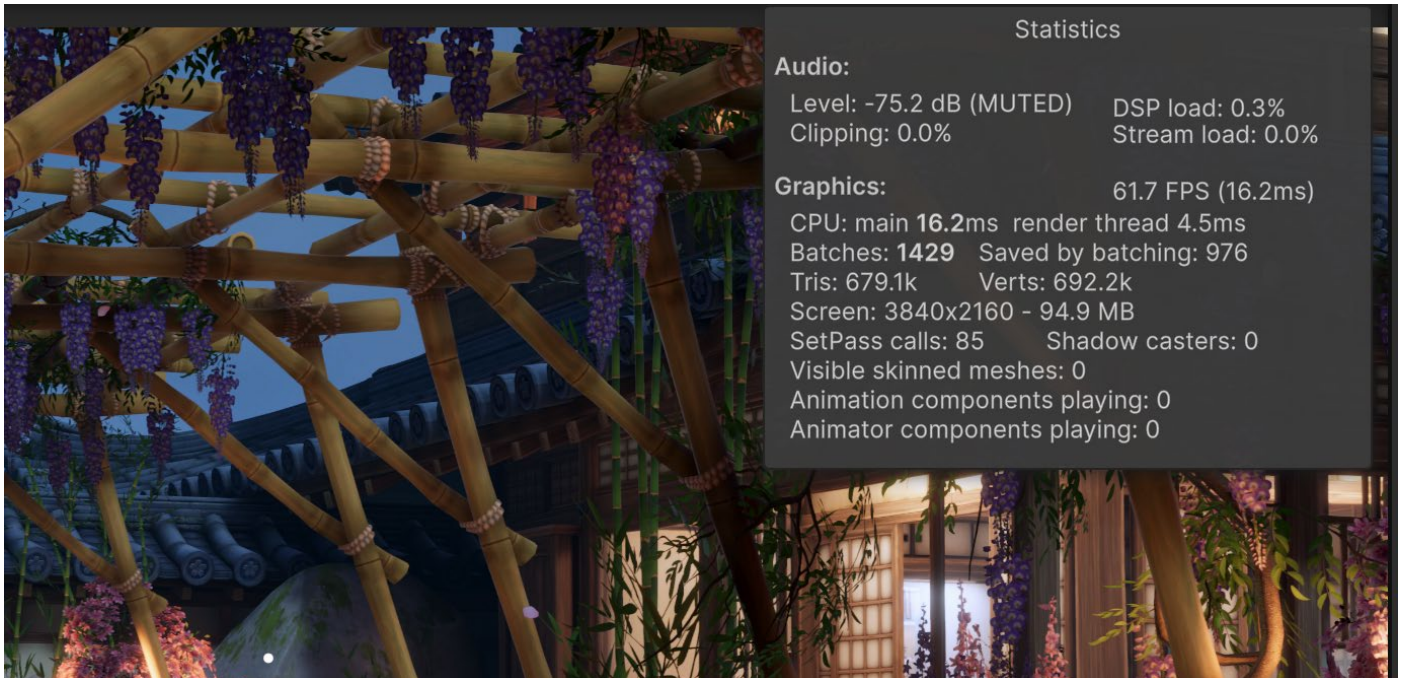
프로파일러는 애플리케이션 성능 저하 문제가 발생하는 이유, 코드가 메모리를 지나치게 할당하는 이유 등을 밝히는 데 도움을 주는 감지 툴이라고 볼 수 있습니다. 프로파일러를 사용하면 내부의 상황을 더 쉽게 파악할 수 있습니다.

Unity에는 에디터와 하드웨어 양측에서 코드를 분석하고 최적화할 수 있는 다양한 프로파일링 툴이 제공됩니다. 본 전자책에서 각 툴에 관해 자세히 살펴보겠지만, 모바일 및 기타 무선 기기, 콘솔, PC 등 각 타겟 플랫폼의 네이티브 프로파일링 툴도 사용해 보실 것을 권장합니다.

프레임 예산의 이해

게이머들은 주로 프레임 속도, 즉 FPS(초당 프레임 수)를 사용하여 성능을 측정하지만, 개발자는 **밀리초 단위 프레임 시간**을 사용하는 것이 바람직합니다. 다음과 같은 간단한 시나리오를 생각해 보면 좋습니다.

런타임 중에 게임은 0.75초 동안 59프레임을 렌더링합니다. 그러나 다음 프레임은 렌더링하는 데 0.25초가 걸립니다. 평균적으로는 60fps의 프레임 속도이므로 괜찮아 보이지만, 마지막 프레임을 렌더링할 때 1/4초가 걸리므로 플레이어는 끊김 현상을 경험하게 됩니다.



Unity는 정밀하고 구체적인 분석에 사용할 수 있는 다양한 프로파일링 툴을 제공하지만, 게임(Game) 뷰의 Statistics 패널에서도 간단히 성능을 확인할 수 있습니다.

이것이 바로 프레임당 정해진 시간 예산을 목표로 삼는 것이 중요한 이유입니다. 프레임당 정해진 시간 예산 목표가 있으면 게임 프로파일링과 최적화를 진행할 때 확실한 목표를 세울 수 있으므로 플레이어에게 더 원활하고 일관된 경험을 제공할 수 있습니다.

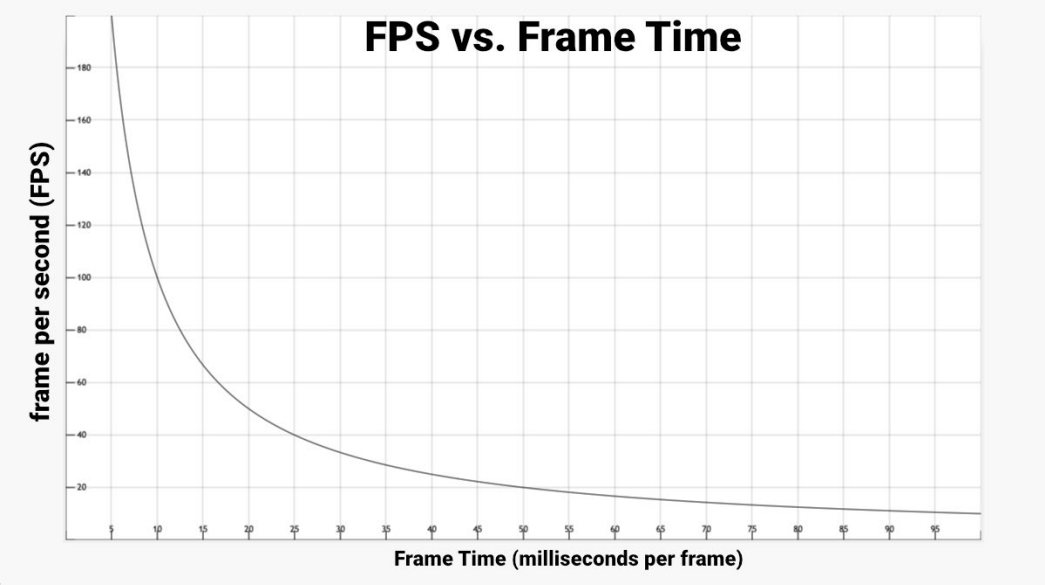
각 프레임에는 목표 FPS에 따른 시간 예산이 있습니다. 30fps가 목표인 애플리케이션은 항상 프레임 속도를 프레임당 33.33ms(1,000ms/30fps) 미만으로 유지해야 합니다. 마찬가지로 60fps의 경우 목표 시간 예산은 프레임당 16.66ms입니다.

UI 메뉴 표시나 씬 로딩 등 인터랙티브 방식이 아닌 시퀀스에서는 몰입형 게이밍 경험에 방해가 되지 않는 경우 이 예산을 초과할 수 있지만, 게임플레이 중에 초과해서는 안 됩니다. 단 하나의 프레임이라도 목표 프레임 예산을 초과하면 장애가 발생합니다.

VR 게임의 경우 플레이어에게 멀미나 불편한 느낌을 유발하지 않기 위해 높은 프레임 속도를 일관되게 유지해야 하며, 게임이 플랫폼 제공업체의 인증을 받아야 하는 경우도 많습니다.

FPS: 정확하지 않을 수 있는 지표

어째서 초당 프레임 수보다 밀리초 단위 프레임 시간을 측정하는 것이 바람직할까요? 그래프를 통해 그 이유를 알아보겠습니다.



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, 그 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.

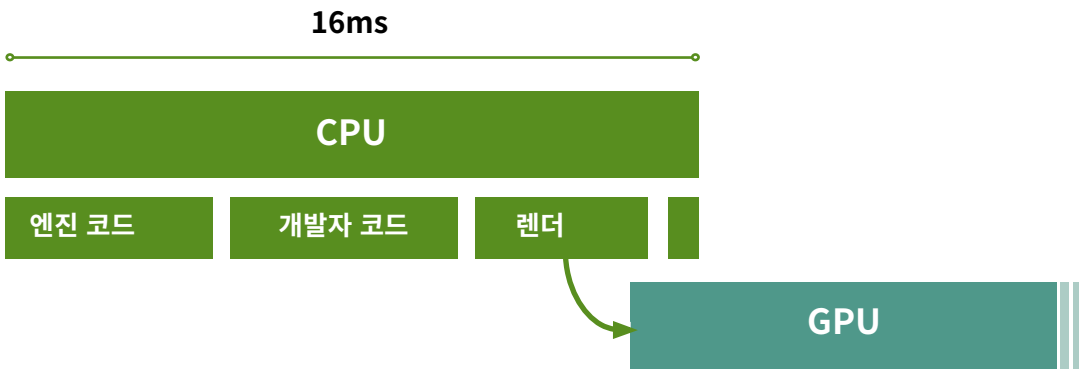
개발자가 게임 속도를 벤치마킹할 때 fps가 아닌 평균 프레임 시간을 사용하는 이유도 여기에 있습니다.

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 참고하세요.

프레임의 구조

위에서 살펴본 내용을 바탕으로, Unity가 프레임을 구성하는 방식과 이 과정에서 CPU(중앙 처리 장치) 및 GPU(그래픽스 처리 장치)가 함께 어떤 역할을 하는지 알아보겠습니다. 60fps를 목표로 하면 결과는 프레임당 16.66ms가 되며, Unity는 CPU와 GPU가 같은 시점에 서로 다른 프레임을 처리하는 파이프라인을 운용합니다.



CPU는 렌더링 명령을 준비하여 GPU로 전달합니다.

CPU에서는 Unity의 내부 엔진 코드(개발자가 제어할 수 없음)가 먼저 실행되고, 이어서 개발자의 커스텀 게임 로직(스크립트)이 실행됩니다. 그런 다음 CPU는 렌더링 명령을 준비합니다. 렌더링 명령은 GPU로 전달되고, GPU가 프레임 N의 렌더링을 시작하는 시점에 CPU는 이미 후속 프레임(N+1)을 처리하고 있습니다.

Unity는 다음과 같은 이중 스레드 시스템을 사용하여 이 워크플로를 간소화합니다.

— 메인 스레드가 게임 로직, 물리, 애니메이션, 입력을 처리하고 렌더링 커맨드를 대기열에 추가합니다.

— 렌더 스레드가 이 커맨드를 GPU 친화적인 명령으로 변환합니다.

렌더링 명령을 수신한 GPU는 이를 그래픽스 파이프라인을 통해 처리합니다. 그런 다음 버텍스 셰이딩, 프래그먼트 셰이딩, 포스트 프로세싱과 같은 작업을 수행한 후 디스플레이에 프레임을 출력합니다.

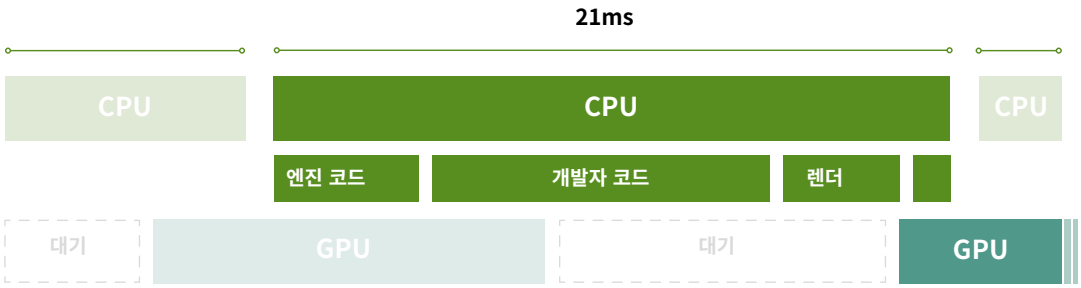
이러한 병렬 처리 덕분에 CPU는 GPU가 현재 프레임을 렌더링하는 동안 다음 프레임의 준비를 시작할 수 있습니다. 단, GPU는 CPU가 렌더링 데이터의 준비를 마칠 때까지 기다린 후에야 작업을 진행할 수 있으므로 CPU와 GPU의 동기화는 성능에 큰 영향을 미칩니다.

프레임 구조에 대해 더 자세히 알아보려면 프레임마다 이루어지는 입력 처리, 물리, 렌더링, GUI 업데이트와 같은 작업 시퀀스에 관해 설명하는 이벤트 함수 실행 순서를 참고하세요.

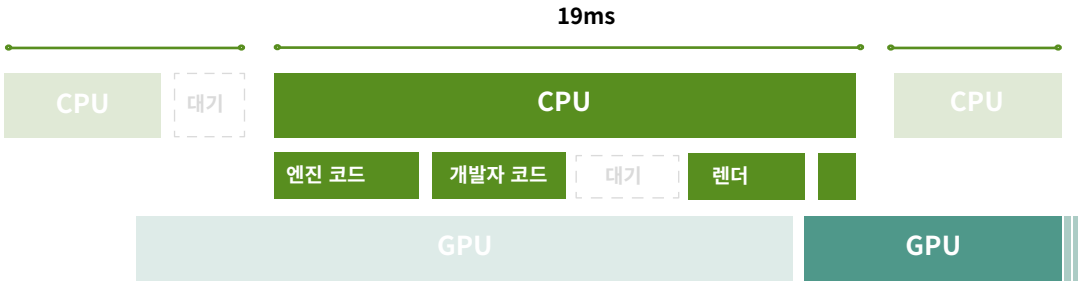
GPU 바운드인지 CPU 바운드인지 알아보기

CPU 바운드 및 GPU 바운드는 시스템의 어느 부분이 게임의 성능을 제한하는지를 가리키며, 따라서 최적화 과정의 첫 단계를 파악하기 위한 핵심 정보입니다.

CPU 바운드란 CPU에서 병목 현상이 발생함을 의미합니다. 스크립트, 물리, 드로우 콜 관리와 같은 작업을 처리하는 데 GPU보다 오랜 시간이 걸리는 것입니다. 이 경우에는 GPU 설정을 최적화해도 프레임 속도가 개선되지 않습니다. 병목 현상을 없애려면 CPU 워크로드를 줄여야 합니다.

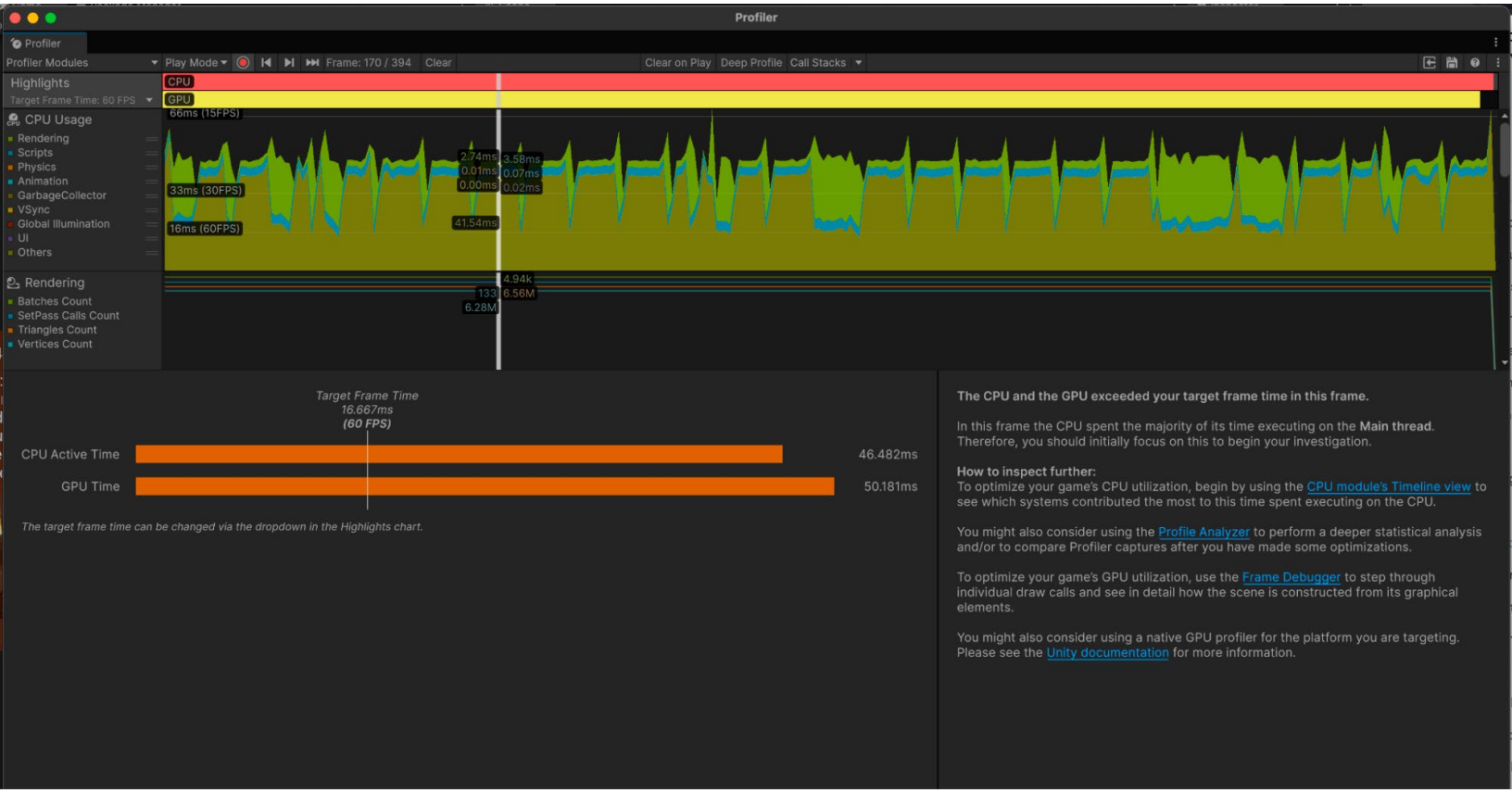


GPU 바운드란 GPU에서 병목 현상이 발생함을 의미합니다. GPU가 그래픽스를 렌더링하는 데 CPU가 로직에 소요하는 시간보다 오랜 시간이 걸리는 것입니다. 이 경우에는 코드를 CPU에서 최적화해도 프레임 속도가 개선되지 않으며, 성능을 개선하려면 셰이더를 간소화하거나, 조명 복잡도를 줄이거나, 해상도를 낮추어야 합니다.



이 대기는 프로파일러에서 `Gfx.WaitForPresent`

Unity 6의 최신 Highlights 모듈을 사용하면 애플리케이션이 CPU 바운드인지 GPU 바운드인지 모든 플랫폼에서 쉽게 확인할 수 있습니다.



Highlights 모듈을 통해 설정된 타겟 프레임 시간을 기준으로 게임의 성능을 쉽게 확인할 수 있습니다. 이 예시에서는 목표인 60fps에 도달하려면 CPU와 GPU 양쪽에서 최적화 작업을 수행해야 합니다.

VSync란?

VSync는 애플리케이션의 프레임 속도를 모니터의 새로고침 속도와 동기화합니다. 만약 60Hz 모니터를 사용하는데 게임이 16.66ms의 프레임 예산 내에서 실행되는 경우, 게임이 더 빠르게 실행되지 않고 60fps로 강제 실행됩니다. 모니터의 새로고침 속도와 FPS를 동기화하면 GPU 부하를 줄이고 화면 찢김 현상 같은 시각적 결함을 방지할 수 있습니다. Unity에서는 Quality 설정(**Edit > Project Settings > Quality**)을 통해 **VSync Count**를 프로퍼티로 설정할 수 있습니다.

Unity 프로파일링 이해

Unity의 프로파일링 툴은 에디터와 패키지 관리자를 통해 사용할 수 있습니다. ‘Unity 프로파일링 및 디버그 툴’ 섹션에서 이러한 툴과 Unity 프레임 디버거를 사용하는 자세한 방법을 확인할 수 있습니다. 여기서는 간략한 개요만 살펴보겠습니다.

- Unity 프로파일러는 Unity 에디터의 성능, 플레이 모드에서 애플리케이션의 성능을 측정하며, 기기에 연결되면 개발 모드에서 애플리케이션의 성능을 측정합니다.
- Profiling Core 패키지는 Unity 프로파일러 캡처에 컨텍스트 정보를 추가하는 데 사용할 수 있는 API를 제공합니다.
- Memory Profiler는 게임이 사용하고 있는 메모리의 양과 메모리를 사용 중인 오브젝트에 대한 심층 분석을 제공합니다.
- Profile Analyzer는 두 개의 프로파일링 데이터 세트를 나란히 비교하여 변경 사항이 애플리케이션 성능에 어떤 영향을 주는지 분석합니다.
- Project Auditor는 프로젝트의 스크립트, 에셋, 코드에 관한 인사이트와 문제를 보고하며, 이들 중 다수가 성능과 관련이 있습니다.

Unity는 프로파일링 툴을 보완하는 여러 디버깅 툴도 제공합니다. 일례로 렌더링 디버거의 Display Stats 패널에서는 에디터를 연결하지 않은 상태에서도 개발 빌드의 일부 성능 지표와 마커(CPU + GPU)를 확인할 수 있습니다.

샘플 기반 프로파일링과 계측 기반 프로파일링

게임 성능을 프로파일링하는 일반적인 방법은 두 가지입니다.

- 샘플 기반 프로파일링
- 계측 기반 프로파일링

샘플 기반 프로파일링

샘플 기반 프로파일링은 코드가 규칙적인 간격으로(주로 밀리초 단위) 수행하는 작업의 스냅샷을 주기적으로 캡처합니다. 이러한 스냅샷을 분석하면 코드의 어느 부분이 가장 자주 실행되어 가장 많은 시간을 소모하는지 알아볼 수 있습니다. 집계된 스냅샷이 캡처되므로 일반적으로 오버헤드는 낮지만 데이터도 개략적인 수준입니다. 샘플링 빈도를 높이면 더 많은 데이터를 얻을 수 있으나 계측 기반 프로파일러 수준의 정밀도는 제공되지 않습니다.

샘플링 프로파일러는 주로 플랫폼 인프라를 사용하여 최소한의 오버헤드와 최대한의 샘플링 빈도를 제공합니다. Windows Performance Analyzer와 Windows용 이벤트 추적, Instruments, Android Studio 등을 샘플링 프로파일러의 예로 들 수 있습니다.

계측 기반 프로파일러

계측 기반 프로파일링은 프로파일러 마커를 추가하여 코드를 계측하는 방식으로, 각 마커에 있는 코드가 실행되기까지 걸리는 시간에 대한 상세한 타이밍 정보를 기록합니다. 이 프로파일러는 각 마커의 시작 및 종료 이벤트 타임스탬프 스트림을 캡처합니다. 이 방법을 사용하면 정보가 누락되지 않지만, 프로파일링 데이터를 캡처하려면 마커를 순서대로 배치해야 합니다.

이를 통해 코드의 성능을 살펴보고, 성능 문제를 쉽게 찾으며, 최적화 지점을 빠르게 식별할 수 있습니다. 더욱 심도 있는 분석을 위해 커스텀 프로파일러 마커를 추가하거나 딥 프로파일링을 활용하는 것도 가능합니다. 이로 인해 오버헤드가 늘어나지만, 샘플 기반 프로파일링에 비해 매우 정밀한 정보를 캡처하여 특정 문제를 식별할 수 있습니다.

딥 프로파일링은 C# Getter 및 Setter 프로퍼티를 비롯해 모든 스크립팅 메서드 호출에 시작 및 종료 마커를 자동으로 삽입합니다. 딥 프로파일링 시스템을 사용하면 스크립팅 측면에서 완전한 프로파일링 세부 정보를 확인할 수 있습니다. 다만 오버헤드가 수반되어 캡처된 프로파일링 범위 내의 호출 수에 따라 보고되는 타이밍 데이터가 늘어날 수 있습니다.

계측과 샘플링 비교

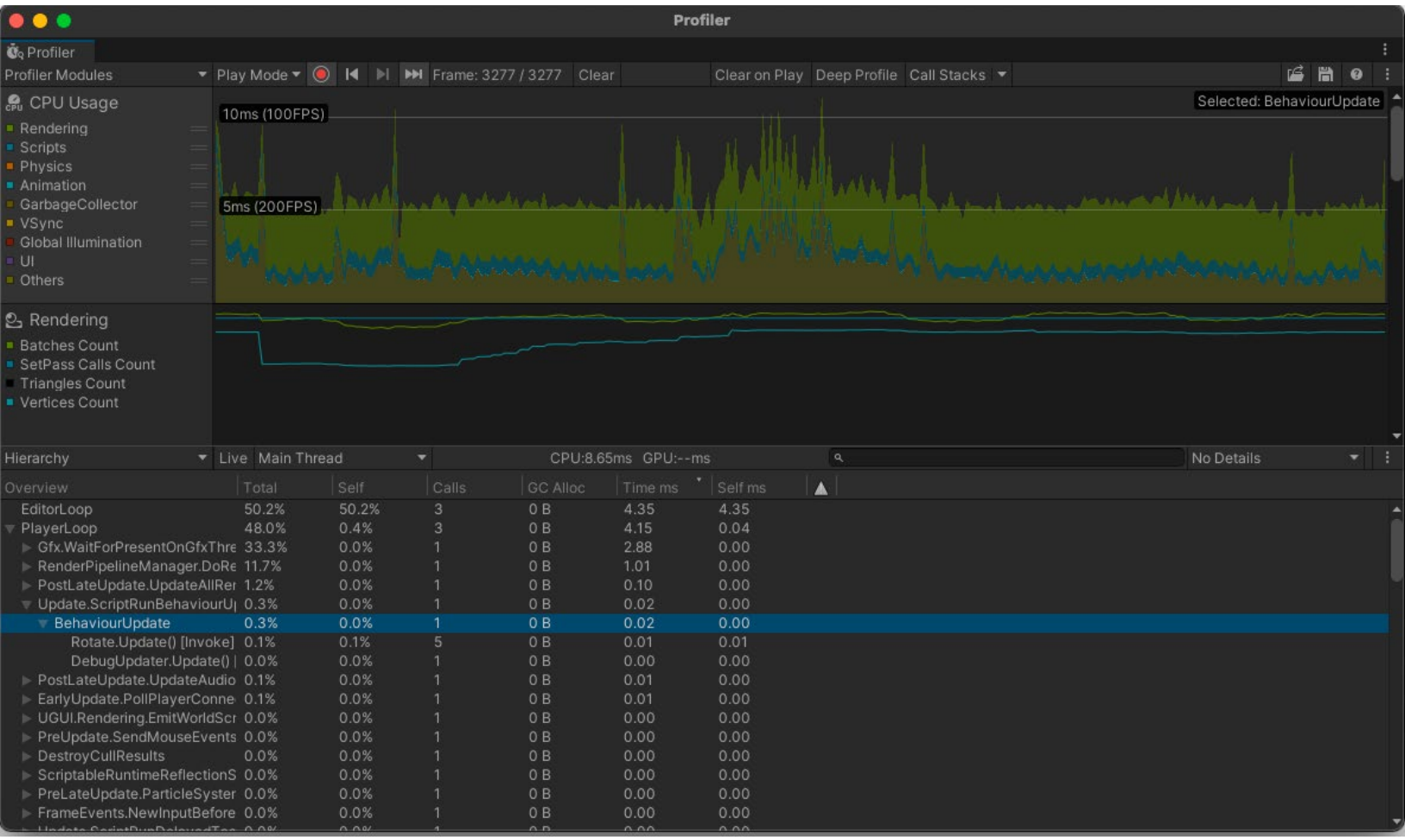
일반적으로 샘플 기반 프로파일링은 애플리케이션의 개략적인 성능을 분석하며, 계측 기반 프로파일링은 중요한 성능 문제를 가려내는 대신 오버헤드가 많이 발생합니다.

샘플링 프로파일러에 의해 발생하는 오버헤드는 프로파일링 중에 CPU가 어떤 작업을 수행하든 일정합니다. 샘플링 빈도를 필요에 따라 변경할 수 있지만, 일반적으로는 낮게 설정합니다. 계측 프로파일링으로 발생하는 오버헤드는 마커의 수에 따라 달라집니다. 즉, 마커를 많이 추가하면 마커 자체도 시간을 소모하므로 캡처의 비용이 커집니다. 계측 프로파일러를 사용할 때는 이 점에 유의하세요. 프로젝트에서 함수 호출이 많이 이루어지는 부분은 실제보다 더 비용이 커 보일 수 있기 때문입니다. 이 현상은 특히 딥 프로파일링 중에 발생할 수 있으며, 이에 따라 프로파일러가 캡처하는 타이밍이 왜곡될 수 있습니다.

Unity의 계측 기반 프로파일링

Unity 프로파일러는 사용하는 모드에 따라 계측 기반 프로파일링과 샘플 기반 프로파일링을 함께 사용합니다. 대부분의 Unity API 표면에서 마커를 설정하면 세부 정보와 오버헤드 간의 균형을 이룰 수 있습니다. 중요한 네이티브 기능 및 스크립팅 코드 베이스 메시지 호출은 과도한 오버헤드를 발생시키지 않으면서 가장 중요한 '핵심 데이터'를 캡처할 수 있도록 계측이 이뤄집니다.

앞서 언급한 스크립팅 코드 기반 메시지 호출(기본적으로 명시적 계측)은 보통 Unity 네이티브 코드에서 사용자의 관리형 코드로 호출되는 첫 번째 호출 스택 뎁스가 포함됩니다. 예를 들어 Start(), Update(), FixedUpdate() 등의 일반적인 MonoBehaviour 메서드가 포함됩니다.



Update() 메서드 호출을 보여 주는 예제 스크립트에 대한 프로파일링

Unity API로 다시 호출되는 관리형 스크립팅 코드의 자식 샘플을 프로파일러에서 확인할 수도 있습니다. 여기에서 한 가지 주의할 사항은 해당 Unity API 코드 자체에 계측 프로파일러 마커가 있어야 한다는 점입니다. 성능 오버헤드를 유발하는 대부분의 Unity API가 계측됩니다. 예를 들어 **Camera.main**을 사용하면 프로파일 캡처에 **FindMainCamera** 마커가 나타납니다. 캡처한 프로파일링 데이터 세트를 살펴볼 때 다양한 마커가 각각 무엇을 의미하는지 알면 유용합니다. 일반 프로파일러 마커에 대해 자세히 알아보려면 이 목록을 참고하세요.

프로파일러 마커로 프로파일링 세부 정보 개선

기본적으로 Unity 프로파일러는 프로파일러 마커를 통해 명시적으로 래핑된 코드 타이밍을 프로파일링합니다. 코드의 주요 함수에 프로파일러 마커를 수동으로 삽입하면 프로파일링의 디테일 수준을 효율적으로 높일 수 있습니다. 마커를 수동으로 추가하면 전체 딥 프로파일링 오버헤드와 캡처의 부정확한 시간이라는 관련 문제가 발생하지 않습니다.

딥 프로파일링이 활성화되어 있으면 Unity는 계측 기반 프로파일링을 사용합니다. 즉, 모든 함수 호출에 대해 정밀하고 정교한 데이터가 수집되도록 추가 명령이 삽입되어, 런타임 오버헤드가 더 높아지는 대신에 실행 시간과 동작을 측정할 수 있게 됩니다.

프로파일러 모듈

프로파일러는 프레임당 성능 지표를 캡처하여 병목 현상을 파악하는 데 도움을 줍니다. 프로파일러에 포함된 모듈을 사용하여 **CPU 사용량**, **GPU**, **렌더링**, **메모리**, **물리** 등의 세부 정보를 상세하게 확인할 수 있습니다.

왼쪽에 모듈이 표시되고 하단에 세부 정보 패널이 표시된 메인 프로파일러 창

프로파일러 창에는 현재 선택한 Profiler 모듈로 캡처한 세부 정보가 뷰 하단의 패널에 목록으로 표시됩니다. 예를 들어 CPU Usage Profiler 모듈은 구체적인 시간과 함께 CPU 작업의 **Timeline** 또는 **계층 구조 (Hierarchy)** 뷰를 표시합니다.

CPU Usage 모듈의 Timeline 뷰: Main Thread 및 Render Thread 마커 세부 정보를 표시

Unity 프로파일러를 사용하면 애플리케이션의 성능을 평가하고 특정 영역이나 문제를 자세히 살펴볼 수 있습니다. 기본적으로 프로파일러는 Unity 에디터의 플레이어 인스턴스에 연결됩니다.

에디터에서 프로파일링하는 것과 스탠드얼론 빌드를 프로파일링하는 것은 성능 면에서 큰 차이가 있다는 점에 유의해야 합니다. 프로파일러를 타겟 하드웨어에서 직접 실행되는 스탠드얼론 빌드에 연결하는 것이 좋습니다. 에디터 오버헤드 없이 가장 정확한 결과를 얻는 방법이기 때문입니다.

프로파일링 워크플로

이 섹션에서는 프로파일링할 때 유용한 목표와 CPU 바운드, GPU 바운드 같은 일반적인 성능 병목 현상에 대해 알아봅니다. 이러한 상황을 식별하여 더 자세히 조사하는 방법도 알아봅니다. 메모리 프로파일링에 대해서도 살펴보는데, 메모리 프로파일링은 런타임 성능과는 크게 관련이 없지만 게임 크래시를 방지할 수 있기 때문에 알아 두어야 합니다.

상위 수준에서 하위 수준까지의 프로파일링

프로파일링 시에는 가장 큰 영향을 미칠 수 있는 영역에 시간과 노력을 집중해야 합니다. 따라서 하향식 접근 방식으로 프로파일링을 시작하는 것이 권장됩니다. 렌더링, 스크립트, 물리, GC(가비지 컬렉션) 할당과 같은 카테고리의 상위 수준 개요부터 시작하고, 문제가 되는 영역을 식별한 후 심층적으로 살펴보며 세부 정보를 알아내는 것입니다. 이 전반적인 접근 방식을 사용하면 핵심 게임 루프에서 원치 않는 관리형 할당이나 너무 많은 CPU 사용량을 유발하는 시나리오를 비롯하여 가장 중요한 성능 문제에 대한 데이터를 수집하고 기록할 수 있습니다.

먼저 GC.Alloc 마커에 대한 호출 스택을 수집해야 합니다. 이 프로세스에 익숙하지 않은 경우, 애플리케이션 수명 동안 반복되는 메모리 할당 찾기에서 몇 가지 유용한 팁을 확인해 보시기 바랍니다.

보고된 호출 스택이 할당 또는 다른 속도 저하의 원인을 찾을 수 있을 만큼 세부적이지 않은 경우, 딥 프로파일링을 활성화한 상태에서 두 번째 프로파일링 세션을 진행하여 할당의 원인을 찾을 수 있습니다. 딥 프로파일링은 본 가이드의 뒷부분에서 자세히 다룹니다. 요약하면 딥 프로파일링은 프로파일러의 한 가지 모드로, 모든 함수 호출에 대한 상세한 성능 데이터를 캡처하여 실행 시간과 동작에 대한 세부적인 인사이트를 제공합니다. 이때 표준 프로파일링에 비해 오버헤드가 많이 늘어난다는 단점이 있습니다.

프레임 시간을 초과하는 주요 원인에 대한 기록을 수집할 때는 해당 프레임이 나머지 프레임과 무엇이 다른지 반드시 기록해야 합니다. 이 상대적인 영향은 딥 프로파일링이 활성화되면 왜곡될 수 있습니다. 딥 프로파일링은 모든 메서드 호출을 계측하므로 오버헤드가 많이 늘어나기 때문입니다.

초기 프로파일링

프로파일링은 프로젝트 개발 주기 전체에서 항상 수행해야 하지만, 그 장점을 최대한 활용하려면 초기 단계부터 시작해야 합니다.

개발 초기에 자주 프로파일링하면 팀 전체가 프로젝트의 성능 특이점을 파악하고 기억할 수 있으며, 이를 벤치마크로 활용할 수 있습니다. 성능이 급격히 나빠진 경우 문제가 발생한 지점을 쉽게 찾아 해결할 수 있습니다.

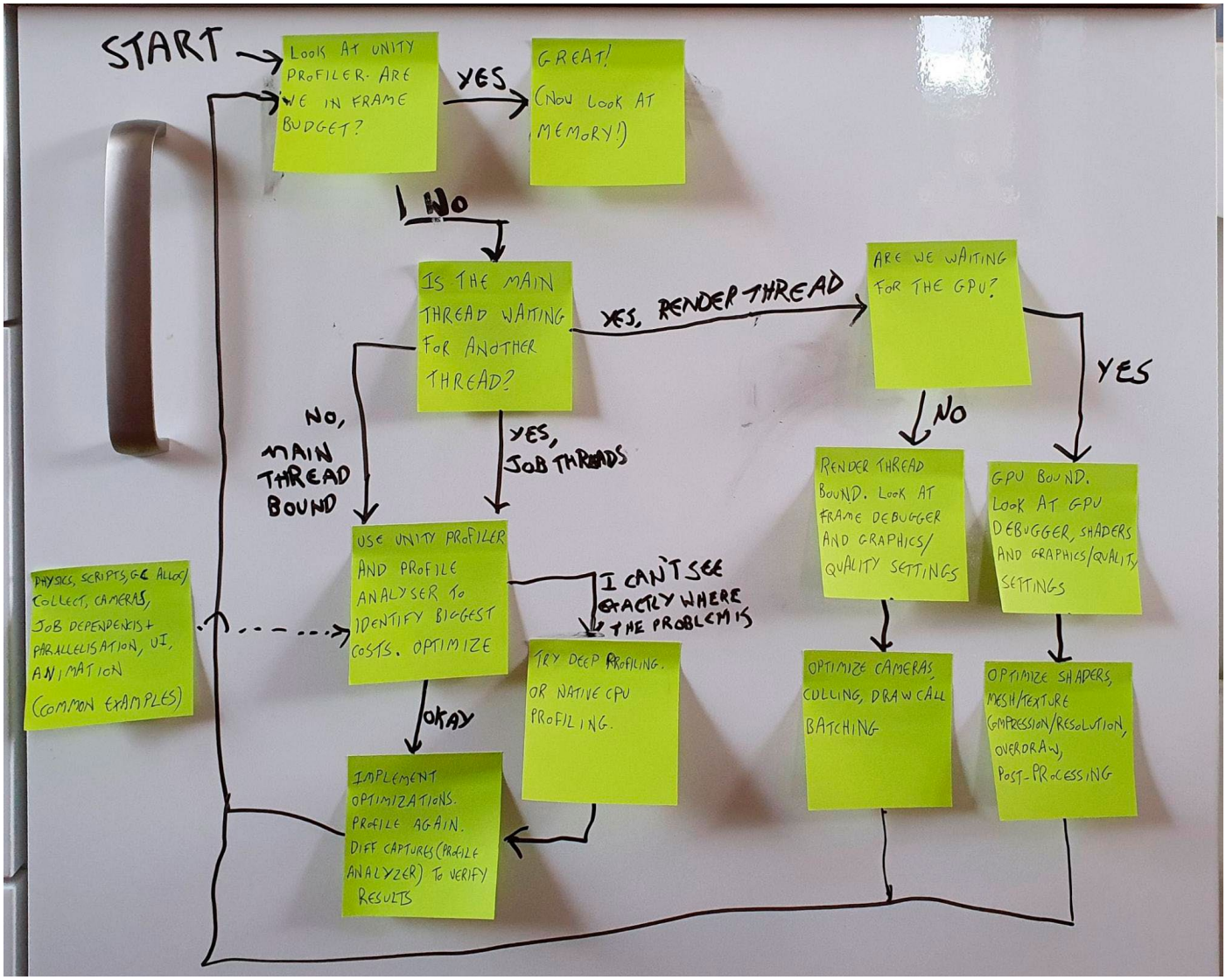
주요 문제를 식별하는 가장 간편한 방법은 에디터에서 수행하는 프로파일링이지만, 빌드를 타겟 기기에서 실행해 프로파일링하고 플랫폼별 툴을 활용하여 각 플랫폼의 하드웨어 특성을 파악하면 가장 정확한 프로파일링 결과를 얻을 수 있습니다. 이렇게 툴을 조합해 사용하면 모든 타겟 기기에서 애플리케이션 성능을 종합적으로 파악할 수 있습니다. 예를 들어 특정 모바일 기기는 GPU 바운드인 반면 다른 모바일 기기는 CPU 바운드일 수 있으며, 해당 기기에서 측정하지 않으면 이를 알아낼 수 없습니다.

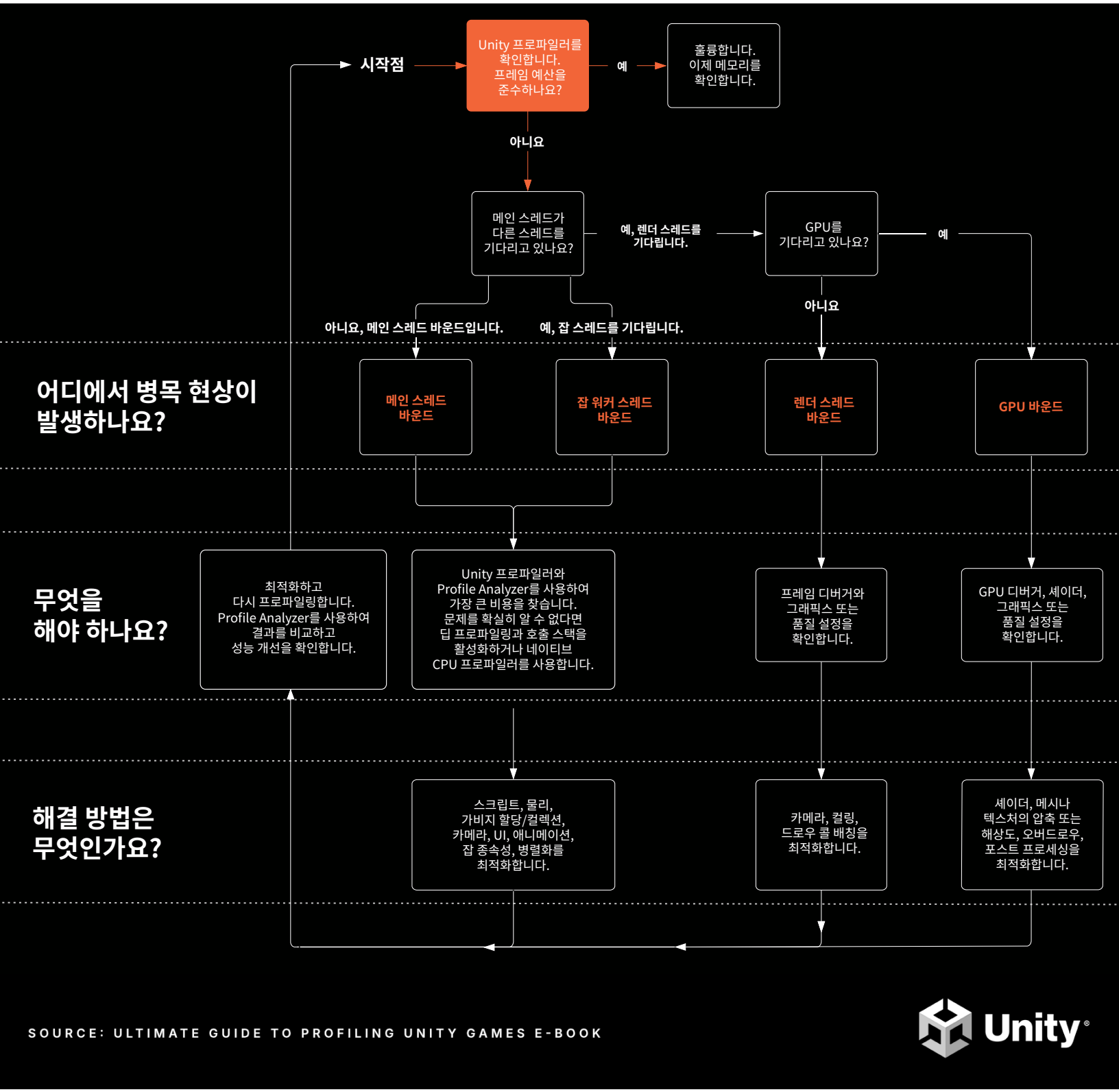
프로파일링 방법론 수립

프로파일링은 체계적인 과정입니다. 문제가 우연히 발견될 때까지 방치하지 말고, 미리 프로파일링 방법론을 정해 두세요.

프로파일링의 핵심은 최적화의 대상이 되는 병목 현상을 식별하는 것입니다. 추측에만 의존하면 실제로는 병목 현상을 일으키지 않는 게임의 일부분만 최적화하게 되어 전체 성능이 거의 개선되지 않거나 전혀 개선되지 않을 수도 있습니다. 어떤 최적화는 원래 의도와 달리 게임의 전체적인 성능에 악영향을 끼칠 수도 있고, 어떤 경우에는 최적화에 큰 노력이 들었지만 결과는 미미할 수도 있습니다. 투입한 시간 대비 최대한의 효과가 나타나도록 최적화하는 것이 중요합니다.

아래 플로 차트에는 초기 프로파일링 프로세스가 나와 있으며, 이어지는 섹션에서는 각 단계를 보다 자세히 살펴보겠습니다. 실제 Unity 프로젝트를 프로파일러로 캡처하고 어떤 부분을 살펴봐야 하는지도 알 수 있습니다.





아래의 플로 차트를 따라 프로파일러를 사용하여 어떤 부분을 집중적으로 최적화해야 하는지 파악할 수 있습니다.

GPU 대기 시간을 포함하여 모든 CPU 활동을 종합적으로 파악하려면 프로파일러의 **CPU 모듈**에서 **Timeline 뷰**를 사용하세요. 캡처를 올바르게 해석하려면 일반 프로파일러 마커에 익숙해져야 합니다. 일부 프로파일러 마커는 타겟 플랫폼에 따라 다르게 표시될 수 있으므로, 시간을 들여 각 타겟 플랫폼에서 게임 캡처를 살펴보고 프로젝트가 정상적인 상태에서 어떻게 캡처되는지 파악하는 것이 좋습니다.

프로젝트 성능은 가장 많은 시간을 소모하는 칩이나 스레드에 좌우되니 바로 이 부분을 집중적으로 최적화해야 합니다. 목표 프레임 시간 예산이 33.33ms이고 VSync가 활성화된 다음과 같은 게임 시나리오를 예로 들어 보겠습니다.

- CPU 프레임 시간(VSync 제외)이 25ms이고 GPU 프레임 시간이 20ms인 경우에는 아무런 문제가 없습니다. CPU 및 GPU 프레임 시간을 모두 16.66ms 미만으로 낮춰 60fps를 목표하지 않는 한, CPU 바운드여도 모든 것이 예산 내에 있다면 최적화를 해도 프레임 속도가 향상되지는 않습니다.
- CPU 프레임 시간이 40ms이고 GPU 프레임 시간이 20ms인 경우는 CPU 바운드이며 CPU 성능을 최적화해야 합니다. 이 경우에는 GPU 성능을 최적화해도 도움이 되지 않습니다. 차라리 해당하는 경우 C# 코드 대신 컴퓨트 셰이더를 사용하는 등 CPU 작업 일부를 GPU로 옮겨 균형을 맞추는 것이 좋습니다.
- CPU 프레임 시간이 20ms이고 GPU 프레임 시간이 40ms인 경우는 GPU 바운드이며 GPU 작업을 최적화해야 합니다.
- CPU 및 GPU 프레임 시간이 둘 다 40ms인 경우 CPU 바운드와 GPU 바운드 둘 다에 해당하며, 30fps를 달성하기 위해 CPU와 GPU 모두 33.33ms 미만으로 최적화해야 합니다.

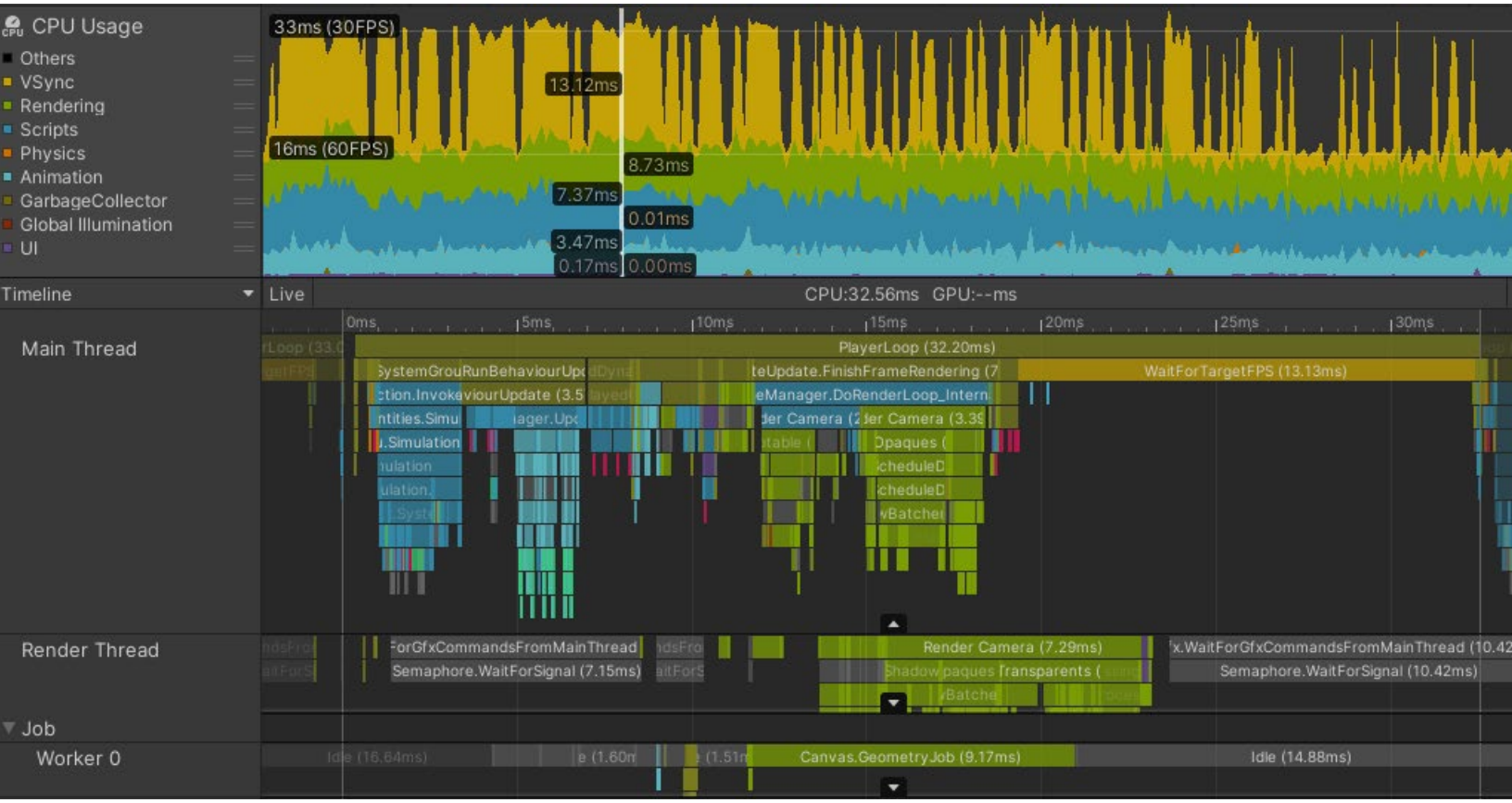
CPU 또는 GPU 바운드에 대한 자세한 내용은 다음 자료를 참고하세요.

- 프레임의 구조: CPU와 GPU
- 게임이 드로우 콜 바운드인가요?(영문)

프레임 예산 준수 여부 확인

개발 과정 초기에 프로젝트의 프로파일링과 최적화를 자주 진행하면 애플리케이션의 모든 CPU 스레드와 전체 GPU 프레임 시간을 프레임 예산 범위 내로 유지하는 데 도움이 됩니다. 이 프로세스를 이끄는 질문은 바로 '프레임 예산을 벗어나지 않았는가?'입니다.

아래 이미지는 지속적으로 프로파일링과 최적화를 진행한 팀에서 개발한 Unity 모바일 게임의 프로파일 캡처 이미지입니다. 캡처에서 볼 수 있듯이 이 게임은 고사양 휴대전화에서는 60fps, 중/저사양 휴대전화에서는 30fps의 프레임 속도를 목표로 합니다.



30fps에 필요한 최대 22ms의 프레임 예산 내에서 과열 없이 안정적으로 실행되는 게임의 프로파일입니다. **WaitForTargetfps**가 렌더 스레드와 워커 스레드의 VSync 및 회색 대기 상태 시간까지 메인 스레드의 마지막 부분에 존재한다는 점을 알 수 있습니다. 또한 프레임마다 **Gfx.Present**의 종료 시점을 확인하여 VBlank 간격을 확인할 수 있으며, Timeline 뷰나 타임 룰러 위에 타임 스케일을 표시하여 한 프레임에서 다음 프레임까지 측정할 수 있습니다.

선택한 프레임에서 거의 절반에 가까운 시간을 노란색의 **WaitForTargetFPS** 프로파일러 마커가 차지하고 있음을 확인할 수 있습니다. 애플리케이션에서 **Application.targetFrameRate**를 30fps로 설정했고 **VSync**가 활성화된 상태입니다. 메인 스레드의 실제 프로세싱 작업은 19ms 정도에 완료되고, 나머지 시간은 다음 프레임을 시작하기 전에 나머지 33.33ms가 경과할 때까지 대기하는 데 사용됩니다. 이 시간에도 프로파일러 마커가 표시되지만, 이 시간 동안 메인 CPU 스레드는 기본적으로 대기 상태이므로 CPU가 냉각되고 최소한의 배터리 전력만 사용합니다.

다른 플랫폼을 사용하거나 VSync가 비활성화되어 있는 경우 확인해야 할 마커가 다를 수 있습니다. 메인 스레드가 프레임 예산 내에서 실행 중인지, 아니면 애플리케이션이 VSync를 기다리고 있음을 나타내는 마커와 함께 정확하게 프레임 예산을 맞추고 있는지, 그리고 다른 스레드에 대기 상태 시간이 있는지 확인해야 합니다.

대기 상태 시간은 회색 또는 노란색 프로파일러 마커로 표시됩니다. 위 스크린샷에서는 렌더 스레드가 **Gfx.WaitForGfxCommandsFromMainThread**에서 대기 상태인 것으로 나타납니다. 한 프레임에서 GPU로 드로우 콜 전송을 완료한 시간과 다음 프레임에서 CPU의 추가 드로우 콜 요청을 기다리는 시간을 의미합니다. 마찬가지로 **Job Worker 0** 스레드는 **Canvas.GeometryJob**에서 약간의 시간을 소요하지만 대부분 대기 상태로 있습니다. 전부 애플리케이션이 프레임 예산 내에서 무리 없이 실행된다는 신호입니다.

게임이 프레임 예산을 준수하는 경우

배터리 사용량과 서멀 스로틀링을 고려한 예산 조정까지 포함해 프레임 예산 내에 있다면 주요 프로파일링 작업을 마무리한 것입니다. 이때 **Memory Profiler** 를 실행하여 애플리케이션도 메모리 할당량 내에 있도록 할 수 있습니다.

CPU 바운드

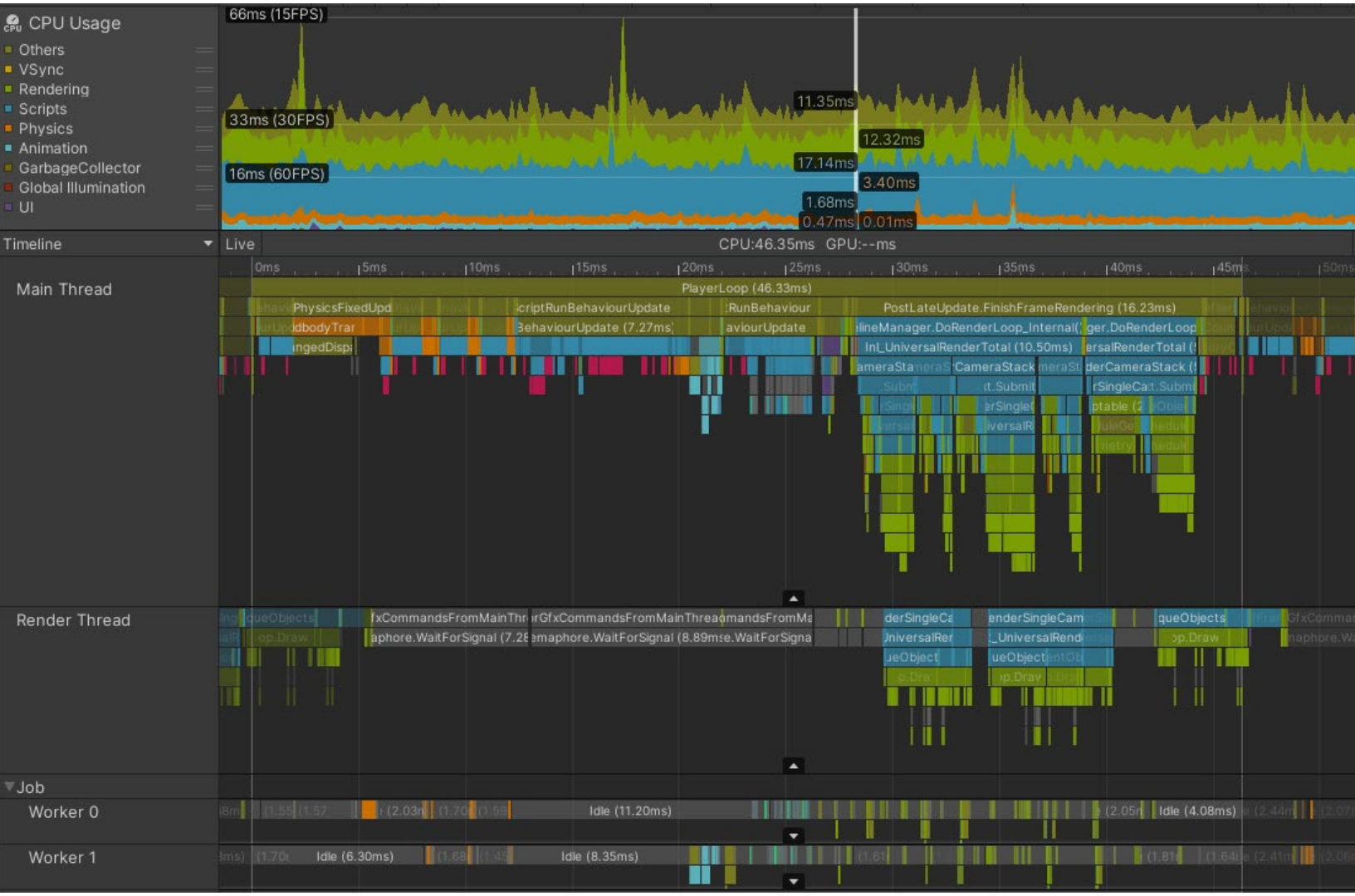
게임이 CPU 프레임 예산을 초과한 경우 다음 단계는 CPU의 어느 부분에서 병목 현상이 일어나는지, 다시 말해 어떤 스레드가 가장 바쁜지 조사하는 것입니다.

전체 CPU 워크로드가 병목 현상을 일으키는 경우는 거의 없습니다. 최신 CPU에는 작업을 동시에 독립적으로 수행할 수 있는 코어가 여럿 탑재되어 있으며, 각 CPU 코어에서 서로 다른 스레드가 작동할 수 있습니다. 전체 Unity 애플리케이션은 다양한 용도로 여러 스레드를 사용하지만, 가장 일반적으로 성능 문제를 찾게 되는 스레드는 다음과 같습니다.

- **메인 스레드**: 기본적으로 대부분의 게임 로직/스크립트가 작업을 수행하는 곳입니다. 물리, 애니메이션, UI, 렌더링의 초기 단계와 같은 대부분의 Unity 시스템이 여기에서 실행됩니다.
- **렌더 스레드**: GPU에 렌더링 명령을 전송하기 전에 완료되어야 하는 준비 작업(예: 씬에서 카메라에 표시될 오브젝트와 뷰 절두체 밖에 있거나, 가려졌거나, 다른 기준에 의해 컬링되어 제외되거나 표시되지 않을 오브젝트)이 처리됩니다.
    - 렌더링 프로세스 중에 메인 스레드는 씬을 검사하고 카메라 컬링, 뎁스 정렬, 드로우 콜 배칭을 수행하여 렌더링할 대상의 목록을 정리합니다. 이 목록은 렌더 스레드로 전달되고, 이 과정에서 플랫폼에 구애받지 않는 Unity 내부 표현이 특정 플랫폼의 GPU에 명령하는 데 필요한 특정 그래픽스 API 호출(DirectX, Vulkan, Metal 등)로 변환됩니다.
- **잡 워커 스레드**: 잡 시스템을 사용하여 특정 종류의 작업이 워커 스레드에서 실행되도록 예약할 수 있습니다. 그 결과 메인 스레드의 워크로드가 줄어들게 됩니다. 물리, 애니메이션, 렌더링 등 Unity의 일부 시스템과 기능도 잡 시스템을 활용합니다.

메인 스레드 최적화의 실제 예시

아래 이미지는 메인 스레드 바운드인 프로젝트를 프로파일링한 예시입니다. Meta Quest 2에서 실행되는 프로젝트의 일반적인 프레임 예산 목표는 13.88ms(72fps) 또는 8.33ms(120fps)입니다. VR 기기를 사용할 때 멀미가 발생하지 않도록 하려면 높은 프레임 속도가 중요하기 때문입니다. 그러나 이미지를 보면 30fps를 달성하는 데에도 문제가 있는 프로젝트임을 알 수 있습니다.



메인 스레드 바운드인 프로젝트의 캡처

렌더 스레드와 워커 스레드는 프레임 예산 내에 있는 예시와 비슷해 보이지만, 메인 스레드가 분명히 전체 프레임에서 계속 사용되고 있음을 알 수 있습니다. 프레임 끝 부분에 있는 약간의 프로파일러 오버헤드를 고려하더라도 메인 스레드가 45ms 이상 사용되고 있습니다. 따라서 이 프로젝트의 프레임 속도는 22fps 미만입니다. 메인 스레드가 대기 상태로 VSync를 기다리고 있음을 나타내는 마커가 없으므로, 전체 프레임 동안 사용되는 것을 알 수 있습니다.

이제 조사를 통해 프레임에서 가장 오랜 시간이 걸리는 부분을 찾고 그 이유를 파악해야 합니다. 이 프레임에서 **PostLateUpdate.FinishFrameRendering**에는 전체 프레임 예산보다 긴 시간인 16.23ms가 소요됩니다. 자세히 살펴보면 **Inl_RenderCameraStack**이라는 마커 인스턴스가 다섯 개 있음을 알 수 있습니다. 다섯 개의 카메라가 활성화되어 씬을 렌더링하고 있다는 의미입니다. Unity의 모든 카메라는 컬링, 정렬, 배칭 등 전체 렌더 파이프라인을 호출하므로, 이 프로젝트에서 가장 급한 작업은 활성 카메라의 수를 줄이는 것입니다. 카메라는 하나만 사용하는 것이 가장 이상적입니다.

모든 MonoBehaviour Update() 메서드 실행을 포괄하는 프로파일러 마커인 **BehaviourUpdate**는 이 프레임에서 7.27ms가 걸립니다.

Timeline 뷰에서 마젠타색 부분은 스크립트가 관리되는 힙 메모리를 할당하는 시점을 나타냅니다. **계층 구조** 뷰로 전환하고 검색창에 **GC.Alloc**을 입력하여 필터링하면 프레임에서 이 메모리를 할당하는 데 약 0.33ms가 걸린다는 것을 알 수 있습니다. 하지만 메모리 할당이 CPU 성능에 미치는 영향을 정확하게 측정한 것은 아닙니다.

GC.Alloc 마커는 일반적인 프로파일러 샘플처럼 시작 지점과 종료 지점을 기록하는 방식으로 시간이 측정되지 않습니다. Unity는 오버헤드를 최소화하기 위해 할당의 타임스탬프와 할당된 크기만 기록합니다.

프로파일러는 프로파일러 뷰에 표시되도록 하기 위한 용도로만 GC.Alloc 마커에 짧은 샘플 지속 시간을 인위적으로 할당합니다.

특히 시스템에서 새로운 범위의 메모리를 요청해야 하는 경우 실제 할당에 더 오랜 시간이 걸릴 수 있습니다. 할당의 영향을 더 정확하게 파악하려면 할당을 수행하는 코드 주변에 프로파일러 마커를 배치합니다. 딥 프로파일링에서 Timeline 뷰의 마젠타색 GC.Alloc 샘플 간격을 확인하여 할당에 얼마나 오랜 시간이 걸렸는지 알 수 있습니다.

또한 새 메모리를 할당하면 성능에 부정적인 영향을 줄 수 있으며, 이러한 영향은 직접적으로 측정하거나 원인을 특정하기가 어렵습니다.

- 시스템에서 새 메모리를 요청하면 모바일 기기의 전력 예산에 영향을 줄 수 있기 때문에 시스템이 CPU나 GPU 속도를 낮출 가능성이 있습니다.
- 새 메모리는 보통 CPU의 L1 캐시에 로드되어야 하므로 기존 캐시 라인을 밀어내게 됩니다.
- 관리되는 메모리의 기존 여유 공간을 초과하면 점진적 가비지 컬렉션이나 동기식 가비지 컬렉션이 곧바로 또는 약간의 지연 후에 트리거될 수 있습니다.

프레임이 시작될 때 **Physics.FixedUpdate**의 인스턴스 네 개에 최대 4.57ms가 소요됩니다. 이후에 **LateBehaviourUpdate**(MonoBehaviour.LateUpdate() 호출)에는 4ms, **애니메이터**에는 약 1ms가 소요됩니다. 이 프로젝트가 목표 프레임 예산과 프레임 속도를 달성하려면 이러한 메인 스레드 문제를 전부 조사하고 적절한 최적화 방안을 찾아야 합니다.

메인 스레드 병목 현상으로 흔히 발생하는 문제

가장 오랜 시간을 소요하는 부분을 최적화하면 성능을 가장 크게 향상할 수 있습니다. 메인 스레드 바운드인 프로젝트에서 최적화했을 때 가장 큰 효과를 볼 수 있는 영역은 다음과 같습니다.

- 물리 계산
- MonoBehaviour 스크립트 업데이트
- 가비지 할당 또는 컬렉션
- 메인 스레드의 카메라 컬링 및 렌더링
- 불충분한 드로우 콜 배칭
- UI 업데이트, 레이아웃, 재빌드
- 애니메이션

일반적으로 발생하는 문제를 최적화하는 데 필요한 실용적인 팁이 포함된 최적화 가이드를 읽어 보세요.

전자책 다운로드

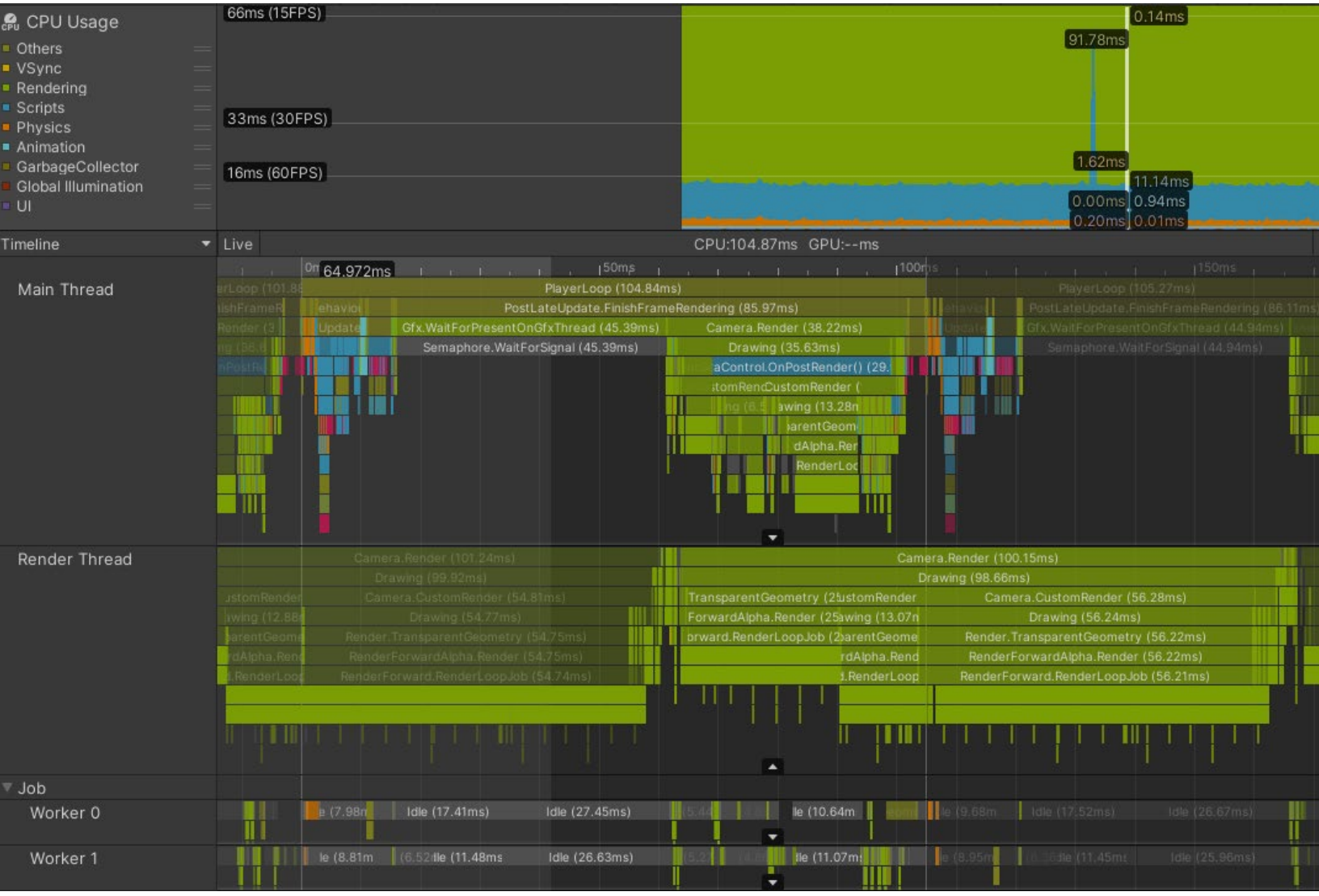
전자책 다운로드

조사하려는 문제에 따라 다른 툴을 활용하는 것도 도움이 될 수 있습니다.

- 시간이 오래 걸리지만 그 원인을 정확히 알 수 없는 MonoBehaviour 스크립트의 경우 코드에 프로파일러 마커를 추가하거나 딥 프로파일링을 통해 전체 호출 스택을 확인합니다.
- 관리되는 메모리를 할당하는 스크립트의 경우, Allocation 호출 스택을 활성화하여 할당의 출처를 정확히 확인할 수 있습니다. 딥 프로파일링을 활성화하거나 메모리별로 필터링된 코드 문제를 보여 주는 Project Auditor를 사용하여 관리되는 할당을 유발하는 모든 코드 라인을 식별할 수도 있습니다.
- 프레임 디버거를 사용하여 바람직하지 않은 드로우 콜 배칭이 일어나는 원인을 조사합니다.

렌더 스레드 최적화의 실제 예시

다음은 렌더 스레드 바운드인 실제 프로젝트입니다. 아이소메트릭 시점을 사용하는 이 콘솔 게임의 목표 프레임 예산은 33.33ms입니다.



렌더 스레드 바운드 시나리오

위의 프로파일러 캡처를 보면 **Gfx.WaitForPresentOnGfxThread** 마커에 표시된 것처럼 현재 프레임에서 렌더링을 시작하기 전에 메인 스레드가 렌더 스레드를 기다리는 것을 알 수 있습니다. 렌더 스레드는 이전 프레임의 드로우 콜 커맨드를 계속 제출하며, 메인 스레드로부터 새로운 드로우 콜을 수락할 준비는 되어 있지 않기 때문에 **Camera.Render**에서 계속 시간을 소모하고 있습니다.

다른 프레임의 마커는 더 어둡게 표시되므로 현재 프레임과 관련된 마커와 구분할 수 있습니다. 또한 메인 스레드가 계속 나아가서 렌더 스레드가 처리할 드로우 콜을 발행하기 시작하면 렌더 스레드가 현재 프레임을 처리하는 데 100ms가 넘게 걸리며, 이로 인해 다음 프레임 동안 병목 현상이 발생한다는 것을 알 수 있습니다.

추가로 조사한 결과 이 게임에는 카메라가 아홉 대나 사용되며 교체 셰이더로 인해 여러 추가 패스가 있는 등 복잡한 렌더링 설정이 사용되고 있었습니다. 포워드 렌더링 경로를 사용하여 130개 이상의 점 광원을 렌더링하기도 했는데, 그러면 광원마다 여러 개의 투명 드로우 콜이 추가될 수 있습니다. 이러한 문제가 모두 합쳐져서 프레임당 3,000번이 넘는 드로우 콜이 발생했습니다.

렌더 스레드 병목 현상으로 흔히 발생하는 문제

렌더 스레드 바운드인 프로젝트에서 조사해야 하는 가장 일반적인 원인은 다음과 같습니다.

— **바람직하지 않은 드로우 콜 배칭**: 특히 OpenGL이나 DirectX 11 등 이전 그래픽스 API를 사용하는 경우 문제가 될 수 있습니다.

— **너무 많은 수의 카메라**: 분할 화면 멀티플레이어 게임을 제작하는 경우가 아니라면 활성 카메라는 하나만 사용해야 합니다.

— **바람직하지 않은 컬링**: 지나치게 많은 항목을 드로우하게 됩니다. 카메라의 절두체 크기를 조사하고 레이어 마스크를 컬링하세요.

Rendering Profiler 모듈은 모든 프레임의 드로우 콜 배치와 SetPass 호출 수를 개략적으로 표시합니다. 렌더 스레드가 GPU에 발행하는 드로우 콜 배치를 조사하기에 가장 적합한 툴은 프레임 디버거입니다.

식별된 병목 현상을 해결하기 위한 툴

본 전자책은 성능 문제를 식별하는 방법을 집중적으로 다루며, 앞에서 소개한 2개의 보완적인 성능 최적화 가이드는 타겟 플랫폼이 PC, 콘솔 또는 모바일 중 무엇인지에 따라 병목 현상을 해결하기 위한 방법을 제시합니다. 렌더 스레드 병목 현상의 경우 Unity는 식별된 문제에 따라 다양한 배칭 시스템과 옵션을 제공합니다. 아래에서 몇 가지 옵션을 간단하게 확인할 수 있으며, 자세한 내용은 여기에 안내된 전자책에서 소개하고 있습니다.

— **SRP 배칭**은 머티리얼 데이터를 GPU 메모리에 지속적으로 저장하여 CPU 오버헤드를 줄입니다. 실제 드로우 콜 수를 줄이는 것은 아니지만, 각 드로우 콜의 비용이 낮아집니다.

— **GPU 인스턴싱**은 동일한 머티리얼을 사용하는 동일한 메시의 여러 인스턴스를 하나의 드로우 콜로 결합합니다.

— **정적 배칭**은 동일한 머티리얼을 공유하는 정적(움직이지 않는) 메시를 결합합니다. 따라서 정적 요소가 많은 레벨 디자인을 작업할 때 효과적입니다.

— **GPU 상주 드로어**는 자동으로 GPU 인스턴싱을 사용하여 비슷한 게임 오브젝트를 하나로 그룹화해서 CPU 오버헤드와 드로우 콜을 줄입니다.

— **동적 배칭**은 규모가 작은 메시들을 런타임에 결합합니다. 드로우 콜 비용이 큰 기존의 모바일 기기에서 효과적이지만, 버텍스 변환에 많은 리소스가 사용될 수 있다는 단점이 있습니다.

— **GPU 오클루전 컬링**은 컴퓨트 셰이더를 사용하여 현재 프레임과 이전 프레임의 뎁스 버퍼 비교를 통해 오브젝트 가시성을 확인하고, 사전에 베이크된 데이터 없이도 가려진 오브젝트의 불필요한 렌더링을 줄입니다.

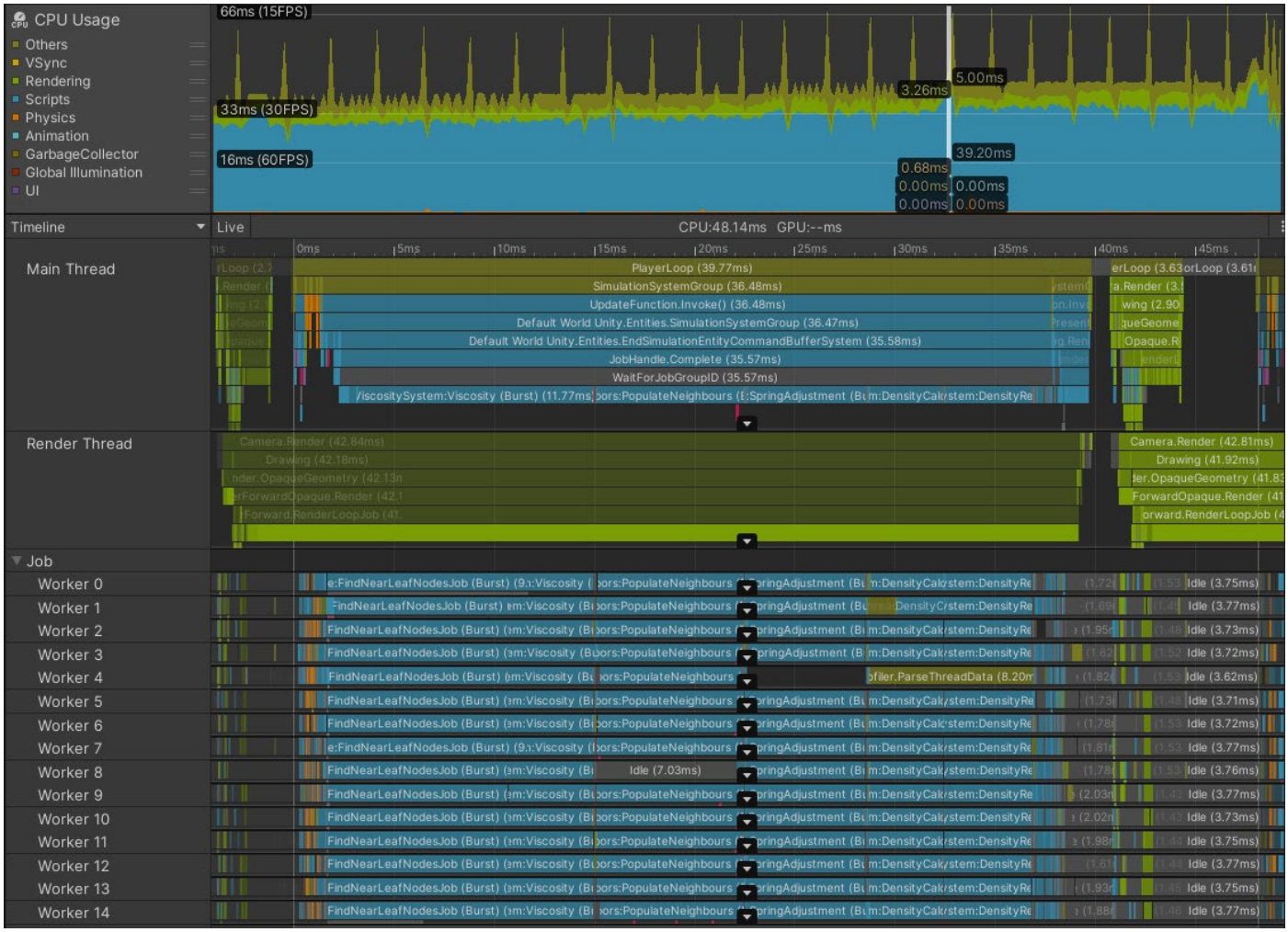
CPU 쪽에서는 **Camera.layerCullDistances**와 같은 기법을 사용하여 카메라에서의 거리를 기준으로 오브젝트를 컬링해서 렌더 스레드로 전송되는 오브젝트의 수를 줄일 수 있습니다. 이렇게 하면 카메라 컬링 중에 CPU 병목 현상을 완화할 수 있습니다.

여기서 소개한 기법은 일부에 불과합니다. 각 옵션은 저마다 장단점이 있으며, 특정 플랫폼에만 한정되는 옵션도 있습니다. 프로젝트에서 여러 시스템의 조합을 사용해야 하는 경우가 많으며, 그렇게 하려면 각 옵션을 최대한 활용할 방법을 알아 두어야 합니다.

워커 스레드

메인 스레드나 렌더 스레드가 아닌 CPU 스레드 바운드인 프로젝트는 그리 흔하지 않습니다. 그러나 프로젝트에서 DOTS(데이터 지향 기술 스택)를 사용하는 경우, 특히 잡 시스템을 사용하여 메인 스레드에서 워커 스레드로 작업을 이동시키는 경우 이러한 문제가 발생할 수 있습니다.

다음은 에디터 내 플레이 모드의 캡처로, CPU에서 파티클 유체 시뮬레이션을 실행하는 DOTS 프로젝트의 모습입니다.



워커 스레드 바운드로 무거운 시뮬레이션을 실행하는 DOTS 기반 프로젝트

언뜻 보기에는 전혀 문제가 없어 보입니다. 워커 스레드는 버스트 컴파일된 잡으로 가득 채워져 있습니다. 많은 양의 작업이 메인 스레드에서 이동되었다는 의미입니다. 일반적으로는 올바른 결정입니다.

하지만 프레임 시간이 48.14ms이고 메인 스레드의 회색 **WaitForJobGroupID** 마커에 35.57ms가 걸린다는 것은 모든 것이 정상적이지는 않다는 신호입니다. WaitForJobGroupID는 메인 스레드가 워커 스레드에서 비동기식으로 실행하기로 예약한 잡이 있음을 보여 줍니다. 하지만 메인 스레드는 워커 스레드에서 실행을 끝내기 전에 해당 잡의 결과가 필요합니다. WaitForJobGroupID 아래의 파란색 프로파일러 마커는 메인 스레드가 대기하는 동안 잡의 완료 시간을 절약하기 위해 잡을 실행하고 있음을 나타냅니다.

잡이 버스트 컴파일되었음에도 여전히 많은 작업이 수행되고 있습니다. 이 프로젝트에서 어떤 파티클이 서로 가까운지 빠르게 찾기 위해 사용하는 공간 쿼리 구조를 최적화하거나 더 효율적인 구조로 변경해야 할 수도 있습니다. 또는 공간 쿼리 잡을 프레임의 시작 지점이 아닌 종료 지점으로 예약하여 다음 프레임이 시작될 때까지 결과가 필요하지 않게 할 수도 있습니다. 이 프로젝트에서 너무 많은 파티클을 시뮬레이션하려는 것일 수도 있습니다. 해결책을 찾으려면 잡의 코드를 추가로 분석해야 합니다. 더 세분화된 프로파일러 마커를 추가하면 가장 시간이 오래 걸리는 부분을 파악할 수 있습니다.

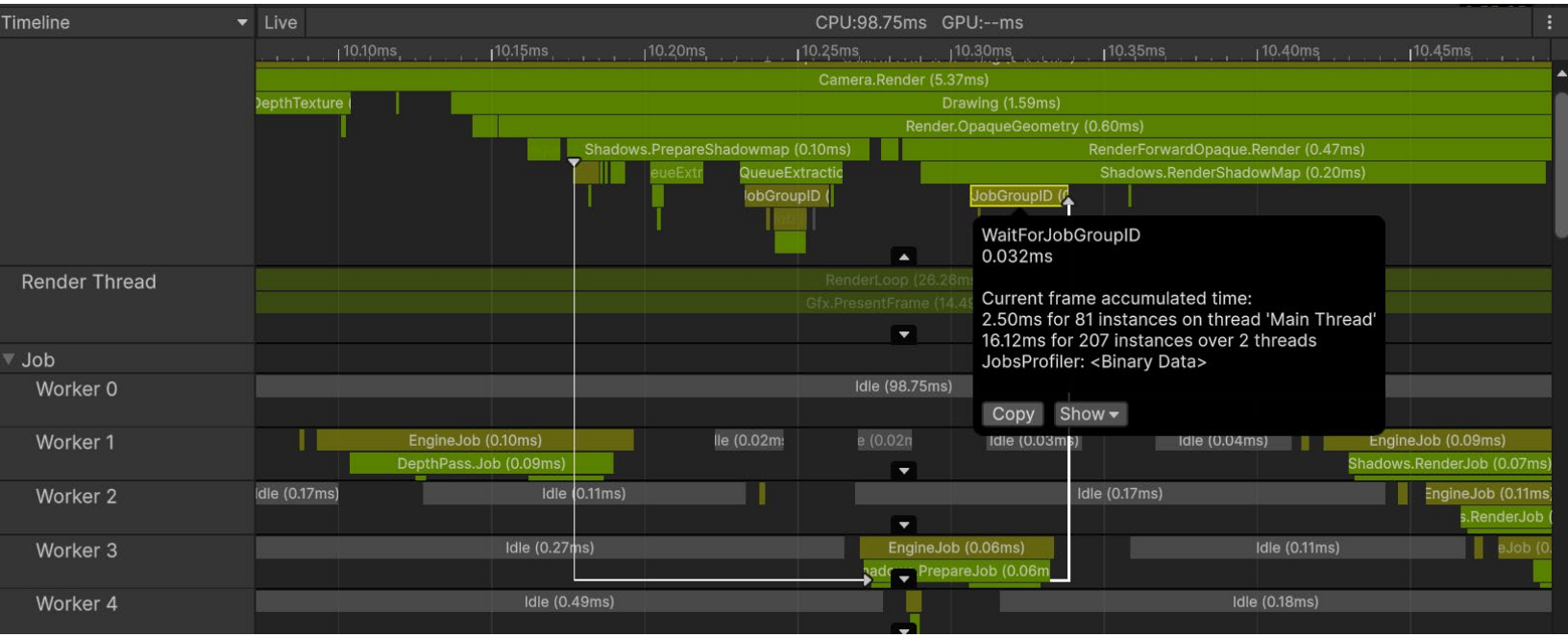
프로젝트에 따라 이 예시처럼 잡이 병렬로 실행되지 않을 수도 있습니다. 하나의 워커 스레드에서 긴 잡이 하나만 실행될 수도 있습니다. 이 경우 예약된 잡과 해당 잡을 완료하는 데 필요한 시간이 충분히 분리되어 있다면 문제는 없습니다. 그렇지 않을 경우에는 위 스크린샷처럼 메인 스레드가 멈춰서 잡이 완료될 때까지 대기하게 됩니다.

워커 스레드 병목 현상으로 흔히 발생하는 문제

동기화 지점과 워커 스레드 병목 현상의 일반적인 원인은 다음과 같습니다.

- 잡이 버스트 컴파일러에서 컴파일되지 않음
- 오래 실행되는 잡이 여러 워커 스레드에서 병렬로 실행되지 않고 하나의 워커 스레드에서 실행됨
- 잡이 예약된 프레임 시점과 결과가 필요한 시점 사이의 시간이 충분하지 않음
- 프레임에 '동기화 지점'이 여럿 있어서 모든 잡이 즉시 완료되어야 함

CPU Usage Profiler 모듈의 Timeline 뷰에서 Flow Events 기능을 사용하면 잡이 예약된 시점과 메인 스레드에서 그 결과가 필요한 시점을 조사할 수 있습니다.



효율적인 DOTS 코드 작성에 관한 자세한 내용은 Unity Learn의 DOTS 베스트 프랙티스 가이드를 참고하세요.

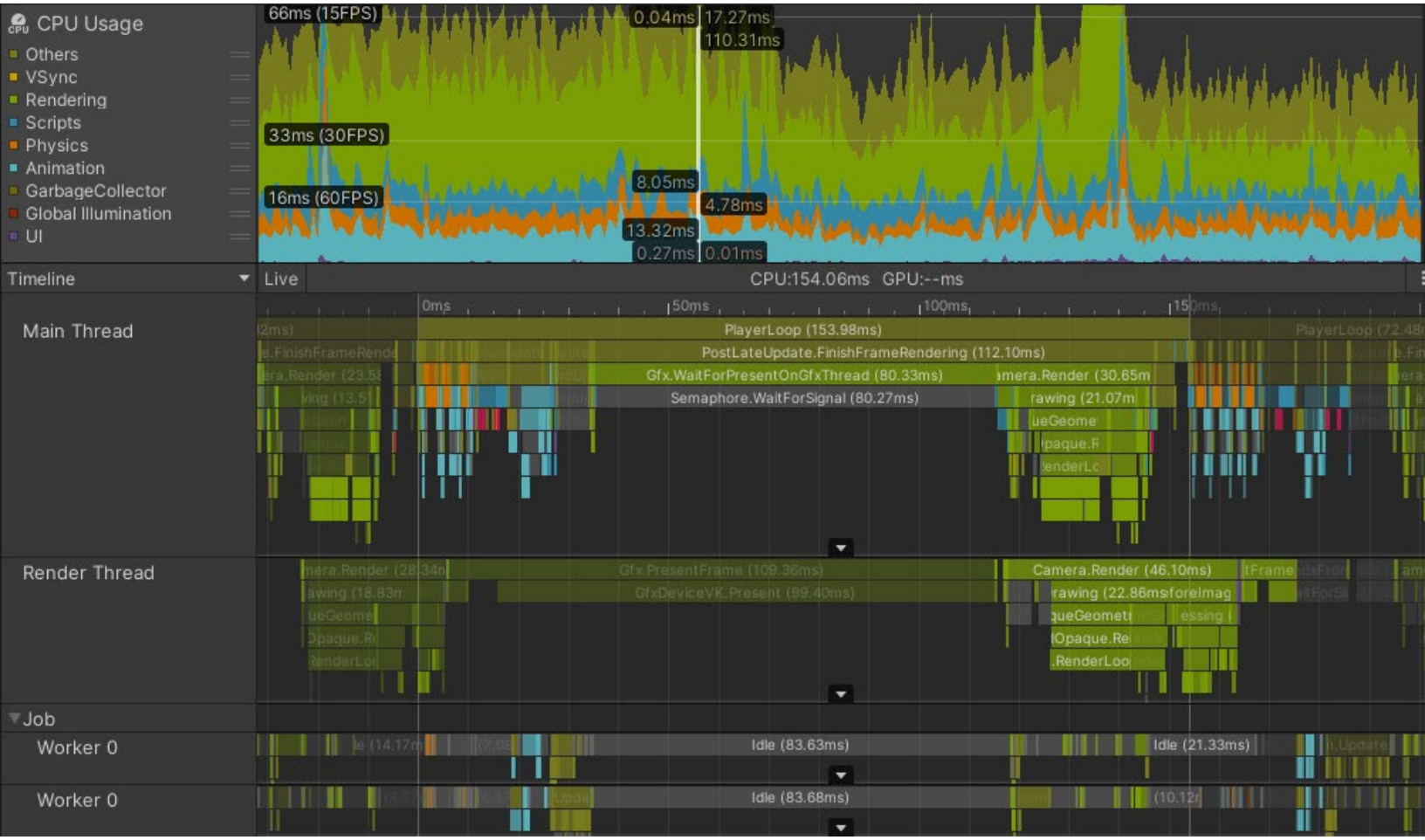
GPU 바운드

메인 스레드가 **Gfx.WaitForPresentOnGfxThread** 등의 프로파일러 마커에 많은 시간을 소모하고 렌더 스레드가 **Gfx.PresentFrame**이나 **<GraphicsAPIName>.WaitForLastPresent** 등의 마커를 동시에 표시하는 애플리케이션은 GPU 바운드입니다.

GPU 프레임 시간을 측정하는 가장 좋은 방법은 타겟 플랫폼 전용 GPU 프로파일링 툴을 사용하는 것이나, 기기에 따라 안정적인 데이터를 캡처하기가 어려울 수 있습니다.

이 경우 CPU와 GPU 양쪽에서 낮은 오버헤드로 개략적인 프레임 시간을 제공하는 FrameTimingManager API가 유용할 수 있습니다.

다음은 Vulkan 그래픽스 API를 사용하는 Android 휴대폰의 캡처입니다. 이 예시에서 Gfx.PresentFrame에 소요된 시간 중 일부는 VSync를 기다리는 것과 관련이 있겠지만, 프로파일러 마커의 길이가 과도하게 길다는 것은 GPU의 이전 프레임 렌더링이 끝나길 기다리는 데 대부분의 시간이 사용되었음을 나타냅니다.



GPU 바운드인 모바일 게임의 캡처

이 게임에서는 특정 게임플레이 이벤트가 셰이더 사용을 트리거하여 GPU에서 렌더링하는 드로우 콜의 수가 세 배나 늘었습니다. GPU 성능을 프로파일링할 때 조사해야 하는 일반적인 문제는 다음과 같습니다.

- 앰비언트 오클루전과 블룸 등 리소스 소모가 많은 전체 화면 포스트 프로세싱 효과
- 다음의 원인으로 발생하며 리소스 소모가 심한 프래그먼트 셰이더
  - 셰이더 코드 안의 브랜칭 로직
  - 특히 모바일에서 반정밀도 대신 전체 플로트 정밀도 사용
  - GPU의 웨이브프론트 점유율에 영향을 주는 레지스터의 과도한 사용

- 다음과 같은 원인으로 인한 투명 렌더 대기열의 오버드로우
  - 비효율적인 UI 렌더링
  - 파티클 시스템의 중첩 또는 과도한 사용
  - 포스트 프로세싱 효과
- 다음과 같이 과도하게 높은 화면 해상도
  - 4K 디스플레이
  - 모바일 기기에서 Retina 디스플레이
- 다음으로 인한 마이크로 트라이앵글
  - 고밀도 메시 지오메트리
  - LOD(디테일 수준) 시스템 부족(모바일 GPU에서 특히 문제가 되지만 PC나 콘솔 GPU에도 영향을 줄 수 있음)
- 다음으로 인한 캐시 누락 및 GPU 메모리 대역폭 낭비:
  - 압축되지 않은 텍스처
  - 밉맵이 없는 고해상도 텍스처
- 동적 그림자가 활성화된 경우 프레임당 여러 번 실행될 수 있는 지오메트리 혹은 테셀레이션 셰이더

애플리케이션이 GPU 바운드 특성을 보이면 프레임 디버거를 사용해 GPU로 전송되는 드로우 콜 배치를 빠르게 파악할 수 있습니다. 하지만 이 툴로는 구체적인 GPU 타이밍 정보를 알 수 없고, 전체 씬이 구성된 방식에 대한 정보만 알 수 있습니다.

GPU 병목 현상의 원인을 조사하는 효과적인 방법은 적절한 GPU 프로파일러를 사용하여 GPU 캡처를 살펴보는 것입니다. 타겟 하드웨어와 선택한 그래픽스 API에 따라 다른 툴을 사용해야 합니다. 자세한 내용은 이 가이드의 프로파일링 및 디버깅 툴 섹션을 참고하세요.

모바일 문제 해결: 발열 관리와 배터리 수명

발열 관리는 모바일 기기용 애플리케이션 개발에서 매우 중요한 최적화 영역입니다. 비효율적인 코드로 인해 CPU나 GPU가 풀 스로틀 상태에서 너무 오래 작동하면 CPU 또는 GPU 칩이 과열됩니다. 과열 및 그로 인한 칩 손상을 방지하기 위해 운영 체제는 기기의 클럭 속도를 낮춰서 칩을 냉각시키며, 이에 따라 프레임이 끊기거나 사용자 경험이 저해될 수 있습니다. 이러한 성능 저하를 서멀 스로틀링이라고 합니다.

프레임 속도가 높아지고 코드 실행이나 DRAM 액세스 연산이 늘어나면 배터리가 더 많이 소모되고 발열이 증가합니다. 성능 문제가 발생하면 저사양 모바일 기기 전반에서 게임을 플레이할 수 없게 되어 시장에서의 기회를 놓치는 결과로 이어질 수 있습니다.

발열 문제를 다룰 때는 작업 예산을 시스템 전체 예산으로 생각해야 합니다.

서멀 스로틀링과 배터리 소모 문제는 초기부터 프로파일링을 통해 게임을 최적화하여 대비하는 편이 좋습니다. 타겟 플랫폼 하드웨어에 맞게 프로젝트 설정을 조정하면 마찬가지로 발열 문제와 배터리 소모 문제를 방지할 수 있습니다.

모바일에서의 프레임 예산 조정

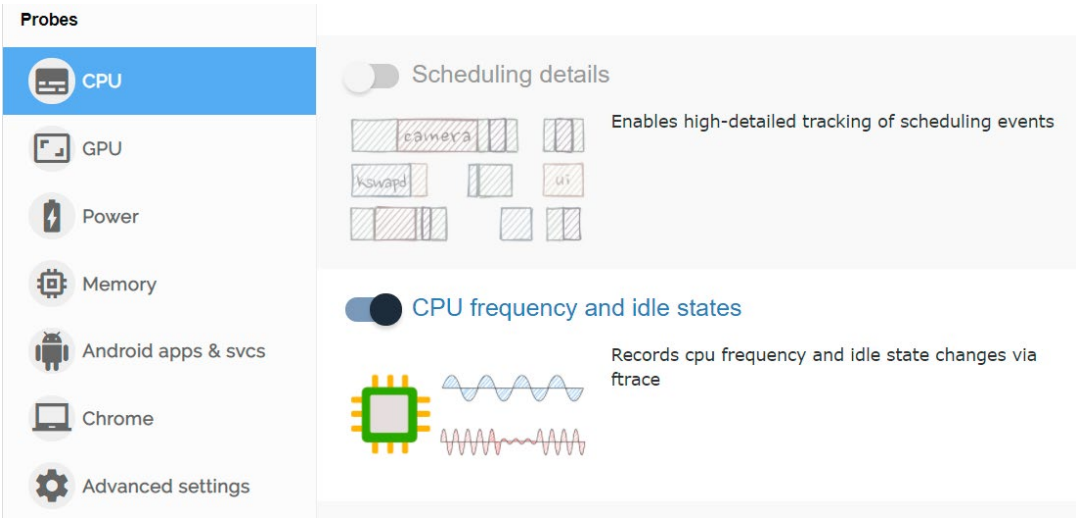
장시간 플레이할 때 발생하는 기기 과열을 방지하려면 일반적으로 프레임 대기 상태 시간을 35% 정도로 유지하는 것이 좋습니다. 그러면 모바일 칩이 냉각될 시간을 확보하고 과도한 배터리 소모를 방지할 수 있습니다. 목표 프레임 시간을 프레임당 33.33ms(30fps의 경우)로 설정하면 모바일 기기의 프레임 예산은 프레임당 약 22ms가 됩니다.

계산식: **(1,000ms / 30) * 0.65 = 21.66ms**

모바일에서 60fps를 달성하려면 같은 계산식을 사용하여 (1,000ms / 60) * 0.65 = 10.83ms의 목표 프레임 시간이 필요합니다. 대부분의 모바일 기기에서는 이 목표를 달성하기 어려우며, 30fps를 목표로 할 때보다 배터리를 두 배 더 빨리 소모합니다. 그래서 많은 모바일 게임은 60fps가 아닌 30fps를 목표로 합니다. Application.targetFrameRate를 사용하여 이 설정을 제어할 수 있습니다. 프레임 시간에 대한 자세한 내용은 프레임 예산 설정 섹션을 참고하세요.

모바일 칩의 주파수 스케일링으로 인해 프로파일링 시 할당된 프레임 대기 상태 시간 예산을 파악하기 어려울 수 있습니다. 게임을 개선하고 최적화하는 것 자체로 긍정적인 효과가 있을 뿐 아니라, 모바일 기기 쪽에서 스스로 주파수를 하향 스케일링하여 발열이 감소하는 결과가 나타나기도 합니다. FTrace나 Perfetto 등의 커스텀 툴을 사용하여 최적화 전후의 모바일 칩 주파수, 대기 상태 시간, 스케일링을 모니터링할 수 있습니다.

목표 fps를 위한 전체 프레임 시간 예산(30fps의 경우 33.33ms)을 초과하지 않는 선에서 이 프레임 속도를 유지하기 위해 기기의 작업량이 줄어들거나 온도가 낮아진다면 올바른 방향으로 가고 있는 것입니다.



FTrace나 Perfetto 등의 툴로 CPU 주파수와 대기 상태를 모니터링하면 프레임 예산 비용 최적화의 결과를 쉽게 파악할 수 있습니다.

모바일 기기의 프레임 예산에 여유를 두어야 하는 또 다른 이유는 실제 세계의 온도 변화를 고려해야 하기 때문입니다. 더운 날에는 모바일 기기가 뜨거워지고 열을 방출하는 데 문제가 생겨서 서멀 스로틀링이 발생하고 게임 성능이 저하될 수 있습니다. 프레임 예산의 일정 비율을 따로 확보해 두면 이러한 상황을 방지할 수 있습니다.

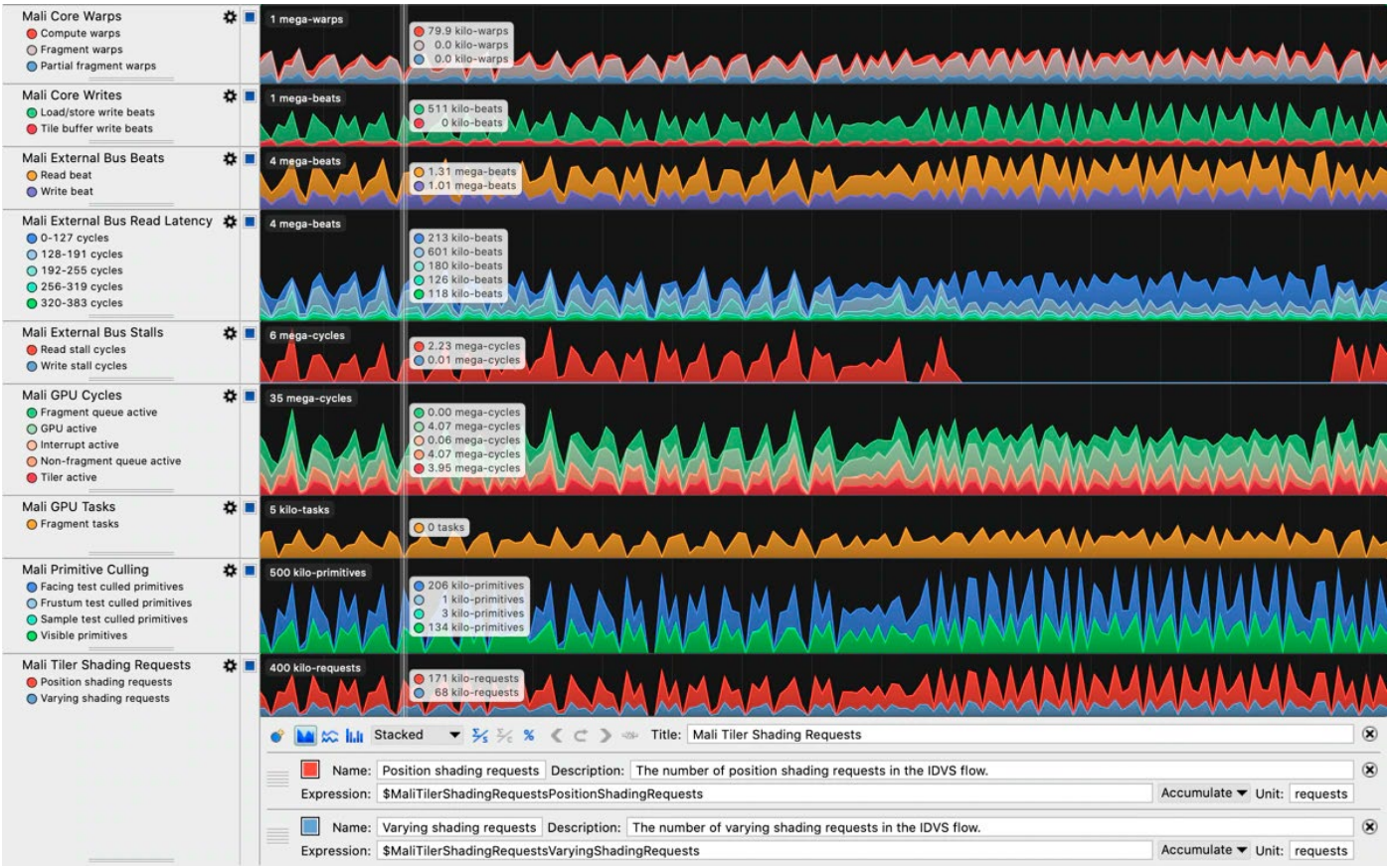
메모리 액세스 연산 줄이기

DRAM 액세스는 보통 모바일 기기에서 전력 소모가 심한 연산입니다. Arm의 모바일 기기 그래픽스 콘텐츠 최적화 관련 어드바이스 기술 자료(영문)에 따르면 LPDDR4 메모리에 액세스하는 데 바이트당 약 100피코줄의 전력이 필요합니다.

프레임당 메모리 액세스 연산 수를 줄이는 방법은 다음과 같습니다.

- 프레임 속도 줄이기
- 가능한 한 디스플레이 해상도 줄이기
- 버텍스 수와 속성 정밀도를 줄인 더 단순한 메시 사용
- 텍스처 압축 및 밉매핑 사용

Arm CPU 또는 GPU 하드웨어를 활용하는 기기에 집중해야 하는 경우 Arm Performance Studio 툴 중에서도 특히 Streamline Performance Analyzer가 메모리 대역폭 문제를 식별하는 데 유용한 몇 가지 성능 카운터를 제공합니다. Mali-G710 성능 카운터 레퍼런스 가이드(영문) 같은 사용자 가이드에서 각 Arm GPU 세대의 카운터 목록과 설명을 확인할 수 있습니다. Arm Performance Studio GPU 프로파일링에는 Arm Immortalis 또는 Mali GPU가 필요합니다.



Arm의 Streamline Performance Analyzer에는 타겟 Arm 하드웨어에서 라이브 프로파일링 세션 동안 캡처할 수 있는 풍부한 성능 카운터 정보가 포함됩니다. 오버드로우로 인해 발생하는 메모리 대역폭 포화 같은 성능 문제를 파악하는 데 유용합니다.

Systemmetrics 패키지를 통해 일부 ARM 하드웨어 지표가 Unity 프로파일러와 플레이어 빌드에 노출됩니다.

벤치마킹을 위한 하드웨어 티어 설정

플랫폼 전용 프로파일링 툴을 사용하는 것 외에도, 각 플랫폼과 지원 품질 티어에 맞는 티어나 최저 사양 기기를 설정하고 각 사양의 성능을 프로파일링 및 최적화할 수 있습니다.

예를 들어 모바일 플랫폼을 타게팅하는 경우 타겟 하드웨어에 따라 기능을 토글하는 품질 제어를 통해 세 가지 티어를 지원할 수 있습니다. 그런 다음 각 티어의 최저 사양 기기에 최적화하면 됩니다. 다른 예를 들자면 콘솔용 게임을 개발하는 경우 이전 버전과 최신 버전 양쪽에서 프로파일링을 진행해야 합니다.

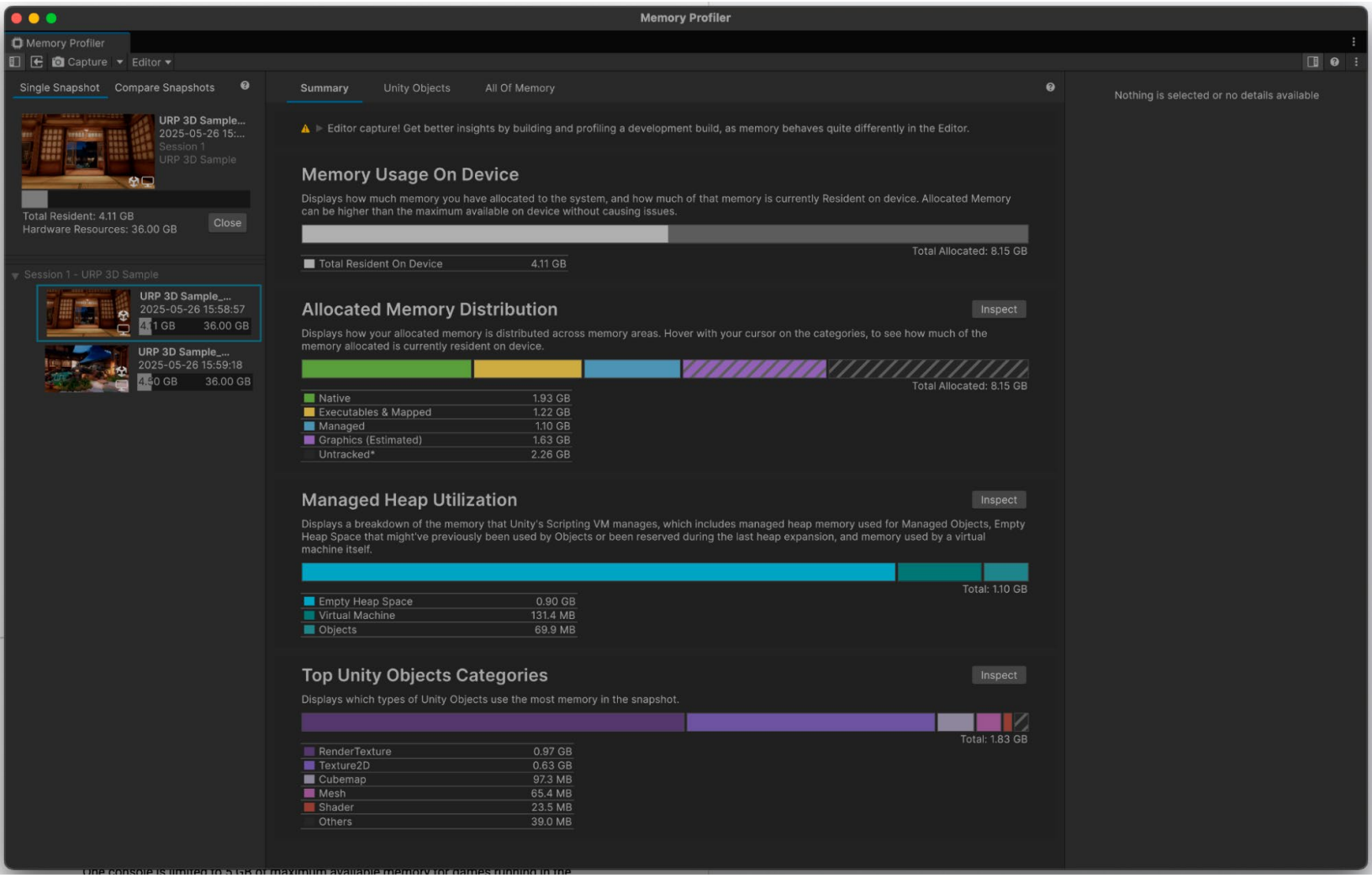
Unity의 최신 모바일 최적화 가이드(이 가이드 및 PC/콘솔 최적화 가이드의 링크는 앞 섹션에서 확인 가능)에서 게임을 실행하는 모바일 기기의 서멀 스로틀링을 줄이고 배터리 수명을 늘리는 데 도움이 되는 여러 유용한 팁을 확인할 수 있습니다.

메모리 프로파일링

메모리 프로파일링은 런타임 성능과는 거의 관련이 없지만 하드웨어 플랫폼 메모리 제한을 테스트하거나 게임 크래시가 발생하는 경우에 유용합니다. 실제로는 메모리 사용량을 늘리도록 변경하여 CPU/GPU 성능을 개선하려는 경우에도 도움이 될 수 있습니다.

Unity에서 애플리케이션의 메모리 사용량을 분석하는 방법은 두 가지입니다.

1. Memory Profiler 모듈: 기본 프로파일러에서 애플리케이션이 메모리를 사용하는 위치에 대한 기본 정보를 제공하는 빌트인 프로파일러 모듈입니다.
2. Memory Profiler: 프로젝트에 추가할 수 있는 Unity 패키지로 제공되는 전용 툴입니다. 패키지를 추가하면 Unity 에디터에 Memory Profiler 창이 추가되며, 여기에서 애플리케이션의 메모리 사용량을 더 자세하게 분석할 수 있습니다. 스냅샷을 저장하고 비교하여 메모리 누수를 찾거나 메모리 레이아웃을 확인하여 메모리 단편화 문제를 발견할 수 있습니다. 자세한 내용은 본 가이드의 뒷부분에서 다루며, 여기서는 일반적으로 고려할 부분을 살펴봅니다.



Memory Profiler 패키지는 Unity 애플리케이션과 Unity 에디터의 메모리 사용량을 조사하는 데 사용할 수 있는 툴입니다.

이 두 가지 툴을 사용하면 메모리 사용량을 모니터링하고, 애플리케이션에서 예상보다 메모리 사용량이 높은 영역을 찾으며, 메모리 단편화를 파악하고 개선할 수 있습니다.

메모리 할당량에 대한 이해 및 정의

멀티플랫폼 개발에서는 타겟 기기의 메모리 제한을 이해하고 메모리를 할당하는 것이 중요합니다. 씬과 레벨을 디자인할 때는 각 타겟 기기에 설정된 메모리 할당량을 준수해야 합니다. 제한과 가이드라인을 설정하면 각 플랫폼의 하드웨어 사양 범위 내에서 애플리케이션이 정상적으로 작동하도록 만들 수 있습니다.

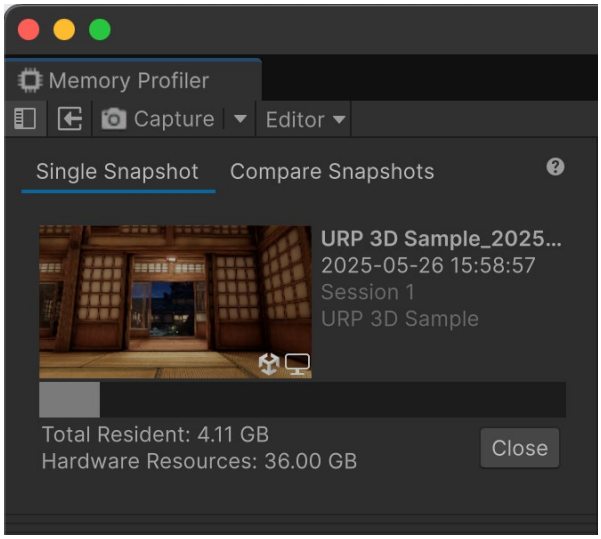
기기 메모리 사양은 개발자 기술 자료에서 확인할 수 있습니다.

메모리 할당량을 알면 텍스처 압축뿐만 아니라 메시 및 셰이더 복잡도와 관련된 콘텐츠 할당량을 정할 때도 유용합니다. 이 모든 요소가 메모리 할당량에 영향을 줍니다. 프로젝트의 개발 주기 동안 메모리 할당량 수치를 참조할 수 있습니다.

물리적 RAM 제한 결정

플랫폼별로 다른 메모리 제한이 적용되므로 애플리케이션에는 각 타겟 기기별 메모리 할당량이 필요합니다. Memory Profiler를 사용하여 메모리 사용량의 캡처 스냅샷을 살펴보세요. **Hardware Resources** 스냅샷(아래 이미지 참조)에는 **물리적 RAM(랜덤 액세스 메모리)**과 **VRAM(비디오 랜덤 액세스 메모리)** 크기가 표시됩니다. 이 수치는 해당 메모리 공간을 모두 사용할 수 있는 것이 아니라는 사실은 고려하지 않습니다. 그러나 대략적인 수치를 확인할 수 있어서 메모리 할당량 설정을 시작하기에 유용합니다.

여기에 표시된 수치가 항상 전체 수치를 나타내지는 않을 수 있기 때문에 타겟 플랫폼의 하드웨어 사양을 교차 참조하는 것이 좋습니다. 개발자 키트 하드웨어의 메모리가 더 많거나 통합 메모리 아키텍처가 적용된 하드웨어로 작업하는 경우도 있습니다.



Hardware Resources 스냅샷에는 스냅샷을 캡처한 기기의 RAM과 VRAM 수치가 표시됩니다.

타겟 플랫폼별로 지원할 최저 사양 결정

지원하는 각 플랫폼에서 RAM 사양이 가장 낮은 하드웨어를 식별하고, 이 정보를 기반으로 메모리 할당량을 결정하세요. 모든 물리적 메모리를 사용할 수 있는 것은 아니라는 점을 기억해야 합니다. 예를 들어 콘솔에서 이전 세대의 게임을 지원하기 위해 하이퍼바이저를 실행하고 있을 수 있는데, 여기에도 전체 메모리의 일부가 사용됩니다. 구체적인 시나리오에 따라 팀 전체가 어느 정도의 비율(예: 전체의 80%)을 사용할지 생각하세요. 모바일 플랫폼의 경우, 고사양 기기를 사용하는 사용자에게 더 좋은 품질과 기능을 지원하기 위해 하드웨어 사양을 여러 티어로 분할하는 것도 좋은 방법입니다.

대규모 팀을 위한 팀별 할당량 고려

메모리 할당량을 정했으면 팀별 메모리 할당량을 설정할 차례입니다. 예를 들어 환경 아티스트에게는 로드되는 각 레벨이나 씬에서 사용할 메모리를 일정량 할당하고, 오디오 팀에는 음악과 음향 효과에 사용할 메모리를 할당하는 등 팀별로 구분하여 메모리를 할당할 수 있습니다. 엄격해 보일 수 있지만, 제작 방향성을 결정할 때 리소스 비용과의 균형을 판단하는 데 도움이 되는 가이드라인으로 간주하시기 바랍니다.

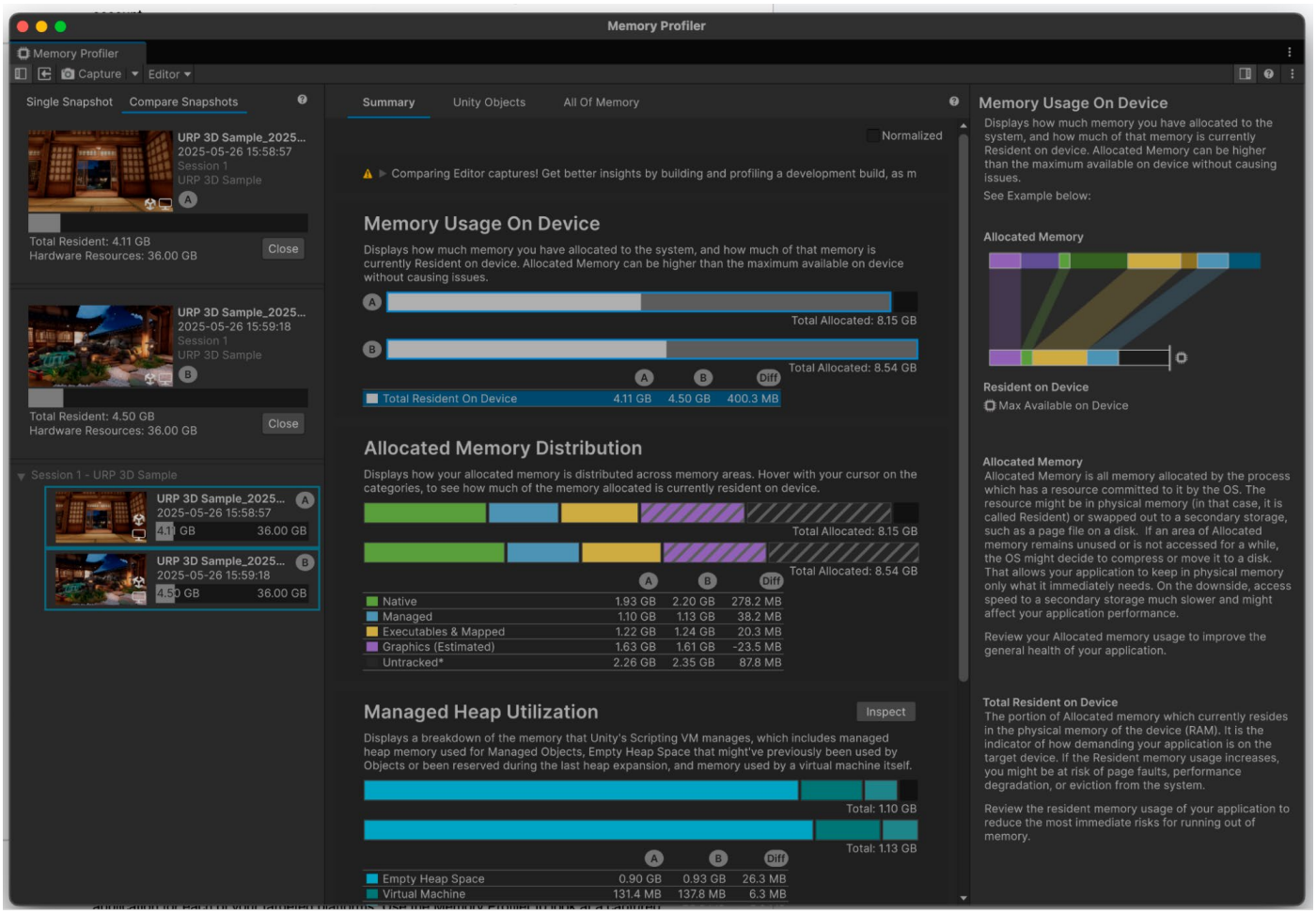
할당량은 프로젝트가 진행됨에 따라 유연하게 조정해야 합니다. 한 팀이 할당된 메모리를 전부 사용하지 못한 경우 잉여 메모리를 다른 팀에 할당하여 개발 중인 게임 영역을 개선하도록 도울 수 있습니다.

타겟 플랫폼의 메모리 할당량을 정하고 설정까지 마쳤다면 프로파일링 툴을 사용하여 게임의 메모리 사용량을 모니터링하고 추적하여 정보에 입각한 결정을 내리고 필요에 따라 조치를 취할 수 있습니다.

Memory Profiler 패키지를 활용하는 심층 분석

Memory Profiler 패키지는 더 세부적인 메모리 분석에 유용합니다. Memory Profiler를 사용하면 스냅샷을 저장하고 비교하여 메모리 누수를 찾거나 애플리케이션의 메모리 레이아웃을 확인하여 최적화할 영역을 파악할 수 있습니다.

Memory Profiler 패키지의 주요 장점 한 가지는 Memory Profiler 모듈을 사용할 때와 마찬가지로 네이티브 오브젝트를 캡처할 뿐만 아니라 관리되는 메모리를 확인하고, 스냅샷을 저장 및 비교하며, 메모리 사용량을 시각적으로 분석하여 메모리가 어디에 사용되는지 자세히 살펴볼 수 있다는 것입니다.



Memory Profiler 패키지를 사용하면 스냅샷을 비교할 수 있습니다.

Memory Profiler 패키지에 관한 자세한 내용은 Unity 프로파일링 및 디버그 툴 섹션을 참고하세요.

메모리 프로파일링 시 유념해야 할 몇 가지 팁

메모리 할당량을 설정할 때는 전체 타겟 플랫폼에서 가장 사양이 낮은 기기를 기준으로 프로파일링해야 합니다. 타겟의 한계를 염두에 두고 메모리 사용량을 면밀히 모니터링하세요.

보통 프로파일링할 때는 많은 메모리를 사용할 수 있는 강력한 개발자 시스템을 사용하는 것이 좋습니다. 이때 대용량 메모리 스냅샷을 저장하거나 스냅샷을 빠르게 로드하고 저장할 수 있는 공간이 있어야 합니다.

메모리 프로파일링은 그 자체로 추가적인 메모리 오버헤드가 발생할 수 있다는 점에서 CPU 및 GPU 프로파일링과 완전히 다른 문제입니다. 더 많은 메모리를 사용하는 고사양 기기에서 메모리 프로파일링을 진행해야 할 수도 있지만, 저사양 타겟의 메모리 할당량 제한에 특히 주의해야 합니다.

품질 수준, 그래픽스 티어, AssetBundle 배리언트와 같은 설정의 메모리 사용량은 더 높은 사양의 기기에서 달라질 수 있습니다. 메모리 프로파일링을 최대한 활용하기 위해 유념해야 할 몇 가지 내용은 다음과 같습니다.

- 품질 및 그래픽스 설정은 섀도우 맵에 사용되는 렌더 텍스처의 크기에 영향을 줄 수 있습니다.
- 해상도 스케일링은 화면 버퍼, 렌더 텍스처, 포스트 프로세싱 효과의 크기에 영향을 줄 수 있습니다.
- 텍스처 설정은 모든 텍스처의 크기에 영향을 줄 수 있습니다.
- 최대 LOD는 모델 등에 영향을 줄 수 있습니다.

- HD(고해상도) 및 SD(표준 해상도) 버전 등 AssetBundle 배리언트가 있고, 타겟 기기의 사양에 따라 사용할 버전을 선택하는 경우에는 프로파일링하는 기기에 따라 에셋의 크기가 달라질 수 있습니다.
- 타겟 기기의 화면 해상도는 포스트 프로세싱 효과에 사용되는 렌더 텍스처의 크기에 영향을 줍니다.
- 기기에서 지원되는 그래픽스 API는 지원하는 셰이더 배리언트 및 지원 여부에 따라 셰이더 크기에 영향을 줄 수 있습니다.
- 다양한 품질 및 그래픽 설정, AssetBundle 배리언트를 사용하는 티어 시스템을 구현하면 더욱 다양한 기기를 타게팅할 수 있습니다.
    - 예를 들어 4GB 모바일 기기에는 AssetBundle의 HD 버전을 로드하고 2GB 기기에는 SD 버전을 로드할 수 있습니다. 그러나 위의 메모리 사용량 배리에이션을 염두에 두고 두 가지 유형의 기기뿐만 아니라 화면 해상도나 지원되는 그래픽스 API가 다른 기기도 테스트해야 합니다.

참고: Unity 에디터는 에디터와 프로파일러에서 로드되는 추가 오브젝트로 인해 항상 많은 메모리를 사용합니다. 이에 더해, 모두 에디터에서 강제로 읽기/쓰기가 활성화되므로 텍스처 메모리 사용량도 높습니다.

Unity 프로파일링 및 디버그 툴

Unity는 성능 문제를 방지, 식별, 수정하는 데 도움이 되는 다양한 툴을 제공하며, 지금까지 이 가이드에서 그러한 툴을 여럿 살펴보았습니다. 이번에는 어떤 경우에 어떤 툴을 사용해야 하는지 자세히 알아보겠습니다.

이 섹션에서 설명하는 툴 중 일부는 정적 분석 툴 카테고리나 디버깅 툴 카테고리에 속합니다(예: 프레임 디버거). 해당 툴은 프로파일러가 아니지만, Unity 프로젝트를 분석하고 개선할 때 툴킷에 포함해 활용해야 하는 중요한 툴입니다.

프로파일링 툴, 디버깅 툴, 정적 분석 툴의 차이점은 무엇일까요?

프로파일링 툴은 코드 실행과 관련된 타이밍 데이터를 계측하고 수집합니다.

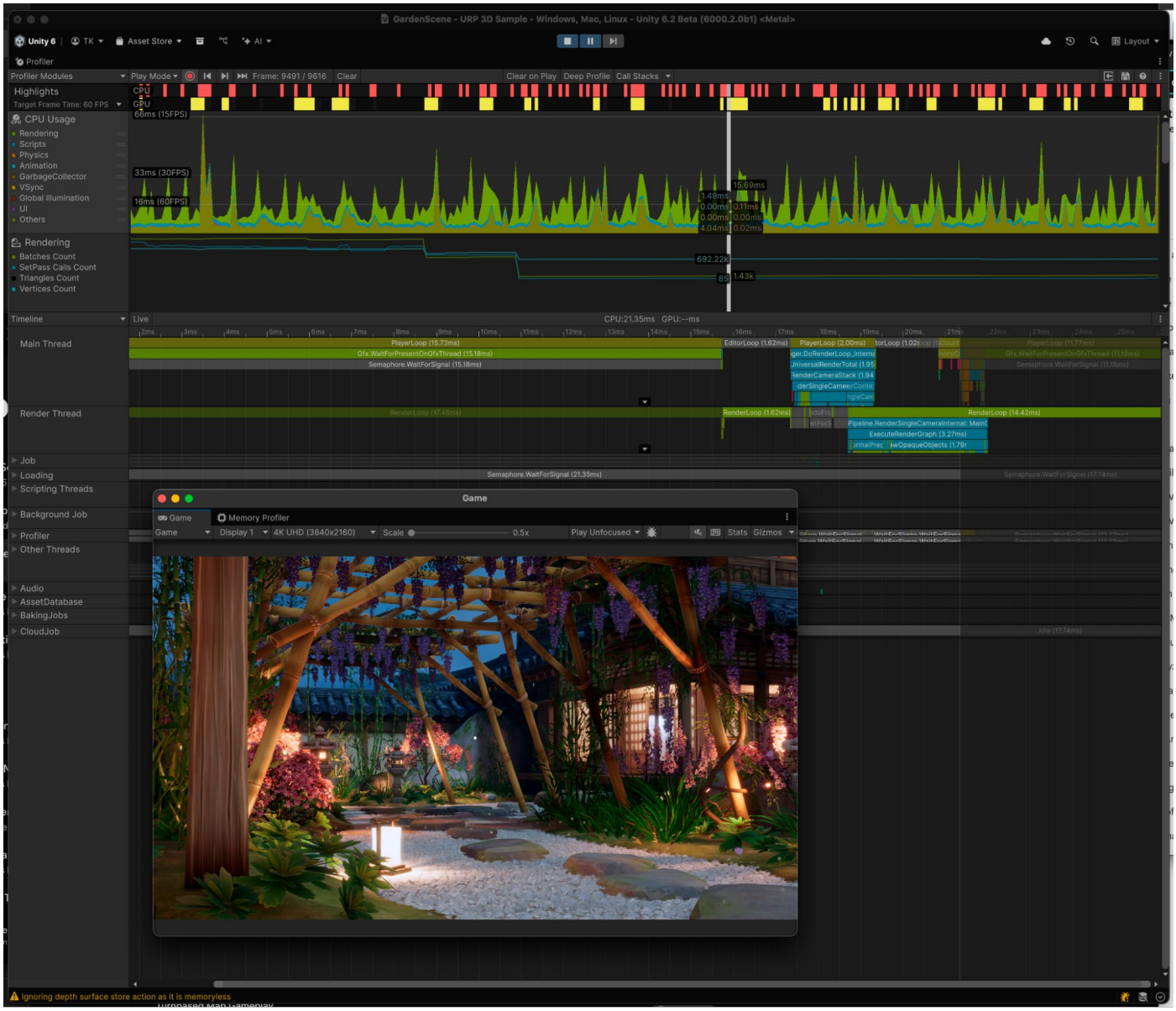
디버깅 툴을 사용하면 프로그램을 단계별로 실행하며 일시 정지하고 값을 검사할 뿐만 아니라, 다양한 고급 기능도 활용할 수 있습니다. 예를 들어 프레임 디버거를 사용하면 프레임 렌더링을 단계별로 살펴보고 셰이더 값을 검사하는 등의 작업을 할 수 있습니다.

정적 분석 툴은 프로젝트를 실행하지 않고도 소스 코드나 기타 에셋을 입력 항목으로 가져와 빌트인 규칙을 사용해서 분석하고 입력 항목의 정확성을 추론하는 프로그램입니다.

Unity 프로파일러

빌트인 Unity 프로파일러를 사용하면 런타임에 병목 현상이나 중단 문제의 원인을 파악하고 특정 프레임 또는 시점에 발생하는 상황을 더 정확하게 이해할 수 있습니다. 앞에서 설명했듯이, 프로파일러를 사용하면 애플리케이션의 빌드를 프로파일링할 수도 있고 에디터에서 직접 프로파일링할 수도 있습니다. 하지만

오버헤드가 증가하고 결과가 왜곡될 우려가 있습니다. 개발 머신이 타겟 기기보다 성능이 좋다는 사실도 항상 염두에 두세요.



Unity 프로파일러 사용 사례: URP 3D 샘플의 정원 씬 프로파일링

프로파일러에는 Deep Profile 설정이 있습니다. 이 설정은 런타임에 실행되는 특정 코드에 대한 자세한 인사이트가 필요한 경우에 유용합니다.

Unity 프로파일러는 다양한 모듈 간 비교에 유용하게 사용할 수 있으므로, 애플리케이션에서 필요한 부분에 집중하는 데 도움이 됩니다. 애플리케이션 성능을 종합적으로 확인할 수 있도록 항상 **CPU**, **Memory**, **Renderer** 모듈을 활성화할 것을 권장합니다. 이에 더해, 살펴보려는 문제의 유형에 따라 Audio나 Physics 등의 다른 모듈을 활성화하세요.

Unity에서 프로파일링 시작하기

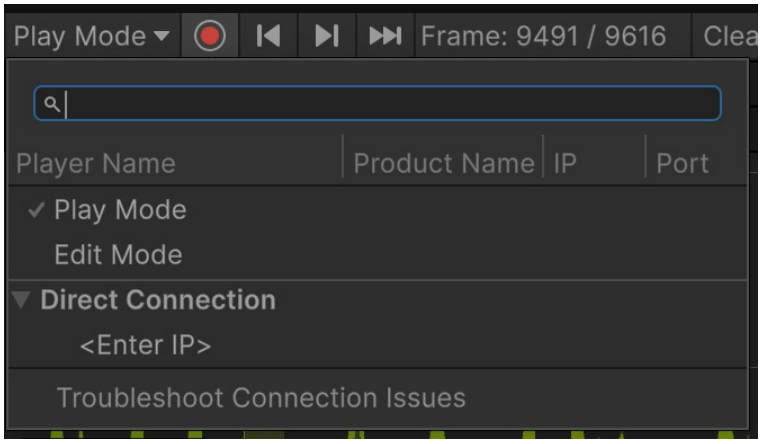
Unity 프로파일러를 처음 사용한다면 여기에서 동영상 튜토리얼을 시청하세요.

또는 아래의 단계에 따라 시작하셔도 됩니다.

- 프로파일링할 때는 개발 빌드를 사용해야 합니다. **File > Build Profiles**에서 **Development Build** 체크박스를 선택합니다.
- **Autoconnect Profiler** 체크박스를 선택합니다(선택 사항).
    - 참고: Autoconnect Profiler를 사용하면 초기 시작 시간에 최대 10초가 추가될 수 있으며, 첫 번째 씬의 초기화를 프로파일링할 때만 이 기능을 활성화해야 합니다. Autoconnect Profiler를 활성화하지 않은 경우, 언제든지 실행 중인 개발 빌드에 프로파일러를 수동으로 연결할 수 있습니다.
- 타겟 플랫폼에 맞게 빌드합니다.
- **Window > Analysis > Profiler**에서 Unity 프로파일러를 엽니다.
- 필요하지 않은 프로파일러 모듈은 모두 비활성화합니다. 활성화된 각 모듈은 플레이어에게 성능 오버헤드를 발생시킵니다. 이 오버헤드 중 일부는 Profiler.CollectGlobalStats 마커를 사용하여 관찰할 수 있습니다.
- **Preferences > Profiler 창에서 Frame Count 옵션을 설정합니다. 값이 클수록 프로파일러 창에서 분석할 수 있는 프레임이 늘어나지만, 그 대신 에디터 머신에서 더 많은 메모리를 사용하게 됩니다.**
- 기기의 모바일 네트워크를 비활성화하고 WiFi만 활성화합니다.
- 타겟 기기에서 빌드를 실행합니다.
    - Autoconnect Profiler를 선택하면 빌드에 에디터 머신의 IP 주소가 베이크됩니다. 애플리케이션은 실행 시 이 IP 주소로 Unity 프로파일러에 직접 연결을 시도합니다. 프로파일러가 자동으로 연결되고 프레임 및 프로파일링 정보가 표시되기 시작합니다.
    - Autoconnect Profiler를 선택하지 않은 경우, **Target Selection** 드롭다운을 사용하여 직접 플레이어에 연결해야 합니다.

상단 표시줄의 새로운 Highlights 모듈에서는 프로젝트에서 타겟 프레임 속도를 충족하는 데 어려움이 있는 부분을 쉽게 식별할 수 있습니다.

Unity 에디터에서 직접 실행 중인 애플리케이션을 프로파일링하면 빌드 시간을 절약할 수 있는 대신 정확도가 떨어집니다. Profiler 창의 **Attach to Player** 드롭다운 메뉴에서 Play Mode를 선택합니다.



플레이 모드에서 실행 중인 게임을 프로파일러로 타게팅하는 과정

Unity 프로파일러 팁

CPU Usage Profiler 모듈에서 VSync 및 Others 마커 비활성화

VSync 마커는 CPU 메인 스레드가 VSync를 기다리는 동안 대기 상태로 있는 낭비되는 시간을 나타냅니다. 이 마커를 숨기면 다른 카테고리 시간이 어떻게 이루어져 있는지와 전체 프레임 시간이 어떻게 이루어져 있는지도 이해하기 어려울 수 있습니다. 이런 점을 고려했을 때 사용할 수 있는 또 다른 옵션은 VSync가 맨 위에 오도록 목록을 다시 정렬하는 것입니다. 그러면 VSync 마커로 인한 노이즈가 줄어들어 그래프가 더 깔끔해지고 전반적인 상황을 더 명확하게 파악할 수 있습니다.

Others 마커는 프로파일링 오버헤드를 나타내며 프로젝트의 최종 빌드에는 존재하지 않는 것이니 무시해도 됩니다.

빌드에서 VSync 비활성화

메인 스레드, 렌더 스레드, GPU가 어떻게 상호 작용하는지 더 명확하게 파악할 수 있는 또 다른 방법은 VSync가 완전히 비활성화된 빌드를 프로파일링하는 것입니다. 사용 방법은 다음과 같습니다.

1. **Edit > Project Settings...**로 이동합니다.
2. **Quality**를 선택하고 타겟 기기에서 사용할 Quality Level을 클릭합니다.
3. **VSync Count**를 **Don't Sync**로 설정합니다.
4. 게임의 개발 빌드를 생성하고 프로파일러에 연결합니다.

그러면 게임은 다음 VBlank까지 기다리지 않고 이전 프레임이 완료되는 즉시 다음 프레임을 시작합니다. VSync를 비활성화하면 일부 플랫폼에서 찢김 현상 같은 시각적 결함이 발생할 수 있습니다. 이 경우에는 릴리스 빌드에서 VSync를 다시 활성화해야 합니다. 하지만 이렇게 인위적인 대기 시간을 제거하면 특히 프로젝트에서 병목 현상이 발생하는 지점을 조사할 때 프로파일러 캡처를 더 쉽게 읽을 수 있습니다.

플레이 모드 또는 에디터 모드 프로파일링이 필요할 때

프로파일러를 사용할 때 플레이 모드, 에디터, 원격 기기 또는 연결된 기기를 플레이어 타겟으로 선택할 수 있습니다.

플레이 모드에서 게임이나 애플리케이션을 프로파일링하고, 에디터 모드에서는 게임과 관련하여 Unity 에디터가 어떤 작업을 수행하는지 확인할 수 있습니다.

에디터를 프로파일링 타겟으로 사용하면 프로파일링 정확도에 많은 영향을 줍니다. Profiler 창은 효과적인 자체 프로파일링을 재귀적인 방식으로 수행합니다. 하지만 에디터 성능이 느려지는 경우 에디터를 프로파일링하는 것도 도움이 될 수 있습니다. 그러면 에디터 속도를 저하하고 생산성을 떨어트리는 스크립트와 확장 기능을 파악할 수 있습니다.

에디터를 프로파일링해야 하는 상황의 예시:

— 플레이 버튼을 누른 후 플레이 모드로 전환하는 데 시간이 오래 걸리는 경우

— 에디터가 느리거나 응답이 없는 경우

— 프로젝트를 여는 데 시간이 오래 걸리는 경우

에셋 데이터베이스로 더 효과적으로 작업하기 위한 팁이라는 블로그 게시글에서는 **-profiler-enable** 커맨드 라인 옵션을 사용하여 에디터가 실행되는 순간부터 프로파일링을 시작하는 방법을 설명합니다.

스탠드얼론 프로파일러 사용

플레이 모드나 에디터 프로파일링을 수행하려면 스탠드얼론 프로파일러를 사용하여 Unity 에디터와 별도의 프로세스로 프로파일러를 실행합니다. 이렇게 하면 프로파일러 UI나 에디터가 측정된 타이밍에 영향을 주지 않습니다. 더 깔끔한 프로파일링 데이터 세트를 확보하여 필터링하고 활용할 수도 있습니다.

스탠드얼론 프로세스로 프로파일러 시작

빠른 반복 작업을 위한 에디터 내에서의 프로파일링

성능 문제 수정 시에 빠르게 반복 작업(iteration)을 수행하려는 경우 에디터에서 프로파일링하는 것이 좋습니다. 예를 들어 빌드에서 성능 문제가 발견되면 에디터에서 프로파일링하여 같은 문제가 발견되는지 확인합니다. 에디터에서도 동일한 문제가 발견되면 플레이 모드 프로파일링을 통해 빠르게 수정 작업을 반복하여 잠재적인 해결책을 찾을 수 있습니다. 문제를 해결하고 나면 빌드를 만든 다음 해당 해결책이 타겟 기기에서도 효과적인지 확인합니다.

이 워크플로를 사용하면 변경 사항을 빌드하고 기기에 배포하는 데 소요되는 시간을 줄일 수 있습니다. 에디터에서 빠르게 반복 작업을 수행하고 프로파일링 툴을 사용하여 수정 사항의 결과를 확인할 수도 있습니다.

Memory Profiler 모듈 사용

Memory Profiler 모듈의 많은 기능이 Memory Profiler 패키지로 대체되었지만, 여전히 이 모듈을 사용하여 메모리 분석 작업을 보완할 수 있습니다.

Memory Profiler 모듈의 **Detailed** 뷰를 사용하면 메모리 사용량이 가장 많은 트리를 자세히 분석하여 어떤 항목이 메모리를 가장 많이 사용하는지 찾을 수 있습니다.

Memory Profiler 모듈에서 시스템에 얼마만큼의 메모리를 할당했는지 손쉽게 확인할 수 있습니다.

다음은 Unity 프로파일러의 추가 사용 사례와 기능을 살펴볼 수 있는 몇 가지 자료입니다.

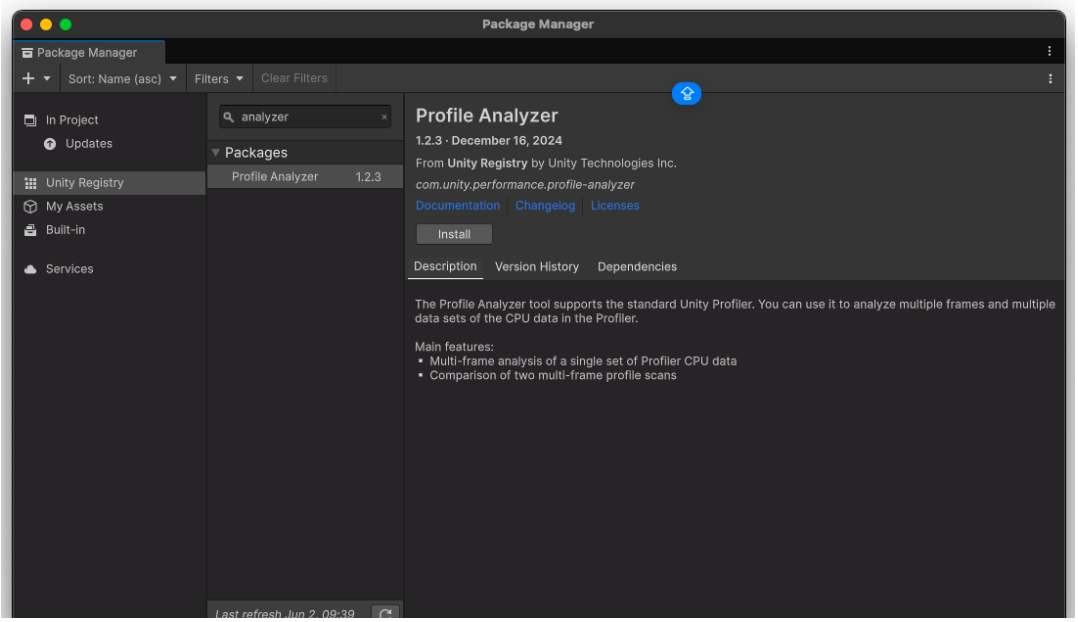
- Unity 매뉴얼의 프로파일러 개요
- Unity의 프로파일링 소개
- 게임을 프로파일링하고 최적화하는 방법
- Unity 프로파일러 소개 및 튜토리얼

Profile Analyzer

기본 Unity 프로파일러를 사용하면 개별 프레임을 상세히 분석할 수 있지만, Profile Analyzer는 여러 Unity 프로파일러 프레임에서 캡처한 마커 데이터를 집계하고 시각화하여 더 전체적인 '큰 그림'을 보여 줍니다. 이를 통해 여러 프레임 또는 여러 프로파일링 세션의 성능 데이터를 쉽게 비교하고 분석할 수 있습니다.

Profile Analyzer를 시작하는 방법은 다음과 같습니다.

1. **Window > Package Management > Package Manager**에서 Profile Analyzer 패키지를 설치합니다.
2. Unity Registry로 이동하여 검색 필터를 사용하거나 목록에서 직접 Profile Analyzer 패키지를 찾습니다.

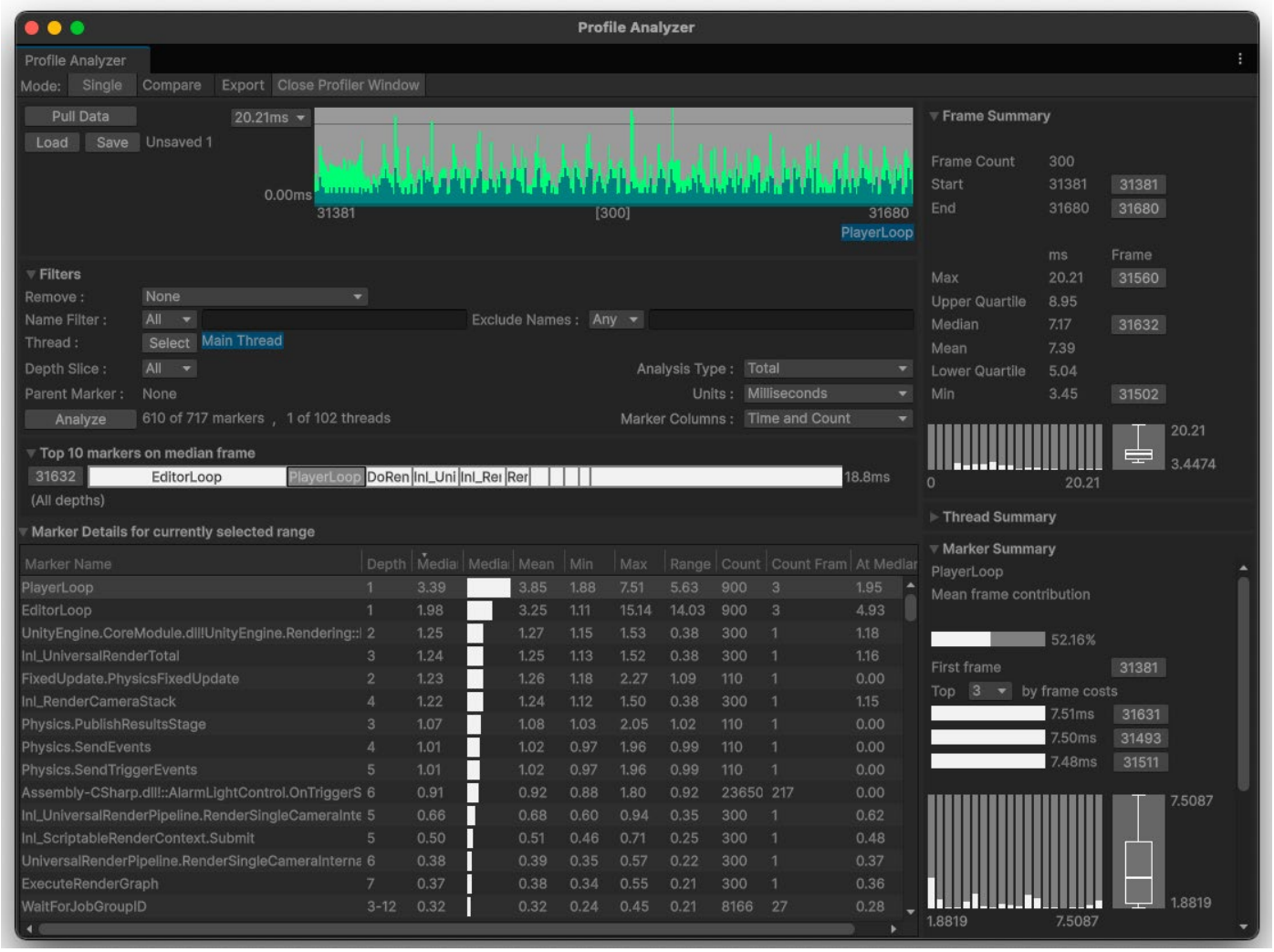


패키지 관리자에서 Profile Analyzer를 설치합니다.

Profile Analyzer는 Unity 프로파일러에서 캡처한 프레임 세트를 가져와 통계적으로 분석합니다. 최소, 최대, 평균, 중간값 타이밍 등 함수별로 유용한 성능 타이밍 정보를 표시합니다.

Profile Analyzer는 데이터 세트의 성능을 비교하는 데 유용하므로, 게임 개발의 전 과정에서 사용하여 성능 및 최적화 문제를 명확하게 파악하세요. 또한 게임 시나리오의 A/B 테스트를 통해 성능 차이를 확인하여 코드 리팩터링 및 최적화, 새로운 기능 활용, Unity 버전 업그레이드 등 전후의 프로파일링 데이터를 비교할 수 있습니다.

Profile Analyzer를 사용할 때는 프로파일링 세션을 저장하여 성능 최적화 작업 전후를 비교하는 것이 좋습니다.



Profile Analyzer는 프로파일링 세션에서 캡처한 여러 프레임을 집계하고 비교하는 데 사용할 수 있어서 Unity 프로파일러와 함께 활용하기에 좋습니다. 위는 Single 뷰의 스크린샷입니다.

시작하려면 먼저 프로파일러를 사용하여 데이터를 캡처한 후 Profile Analyzer에 이 데이터를 채워서 분석을 수행해야 합니다.

한 번에 하나의 프레임만 살펴보는 방법에 비해, 집계된 데이터를 사용하면 전체적인 게임의 상황을 더 상세히 알아볼 수 있습니다. 예를 들어 300프레임(10초)의 게임플레이 캡처나 20초의 로딩 시퀀스를 확인하고 다음과 같은 질문에 대한 답을 찾을 수 있습니다.

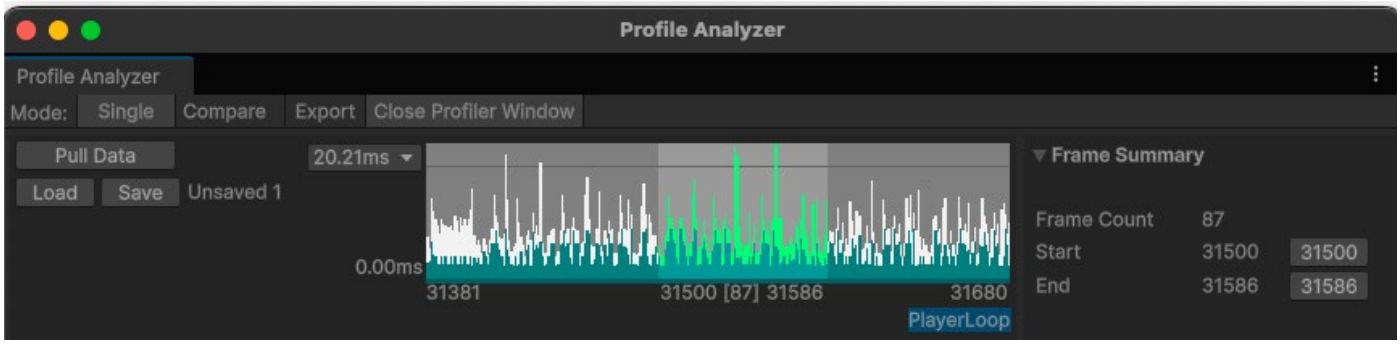
— 메인 스레드와 렌더 스레드에서 CPU 리소스를 가장 많이 사용하는 부분은 어디인가?

— 각 마커의 평균/중간/전체 리소스 소모 값은 얼마인가?

이처럼 중요한 질문에 대한 답을 찾으면 더 큰 문제를 발견하여 우선적으로 최적화하는 데 도움이 될 수 있습니다.

Profile Analyzer에서 제공하는 통계와 세부 정보를 활용하면 여러 프레임에 걸쳐 코드를 실행하거나 이전 프로파일 캡처 세션과 비교했을 때 코드의 성능 특성을 더 자세히 분석할 수 있습니다.

Frame Control 패널을 사용하여 하나의 프레임 또는 프레임 범위를 선택합니다. 프레임을 선택하면 Marker Details 창이 업데이트되어 해당 프레임의 집계 데이터와 함께 유용한 통계가 포함된 마커 목록(정렬 가능)이 표시됩니다.



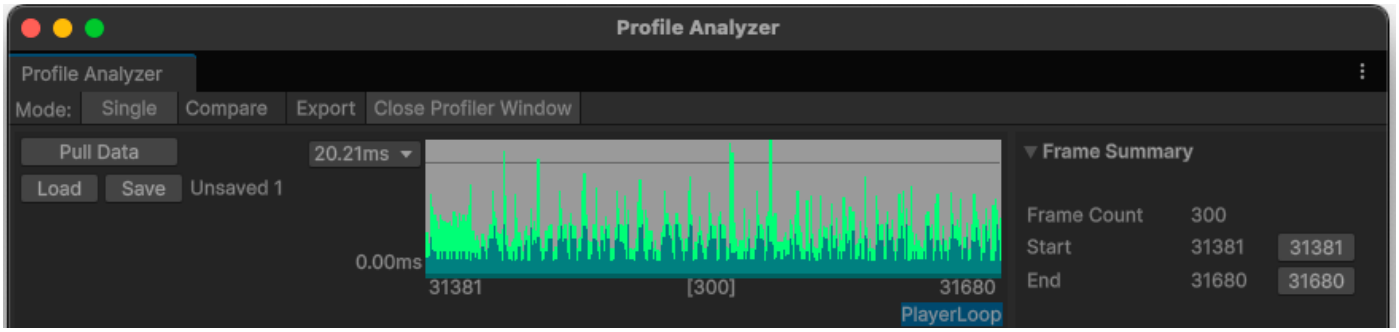
Frame Control 패널을 사용하여 살펴볼 프레임 범위를 선택합니다.

Marker Summary 창은 선택한 마커에 관한 깊이 있는 정보를 표시합니다. 목록의 각 마커는 선택한 프레임 범위 내의 필터링된 모든 스레드에서 해당 마커의 모든 인스턴스를 집계한 것입니다.

Marker Summary 패널에는 Marker Details 패널에서 선택한 각 마커 집계에 대한 자세한 정보가 표시됩니다.

Profile Analyzer 뷰

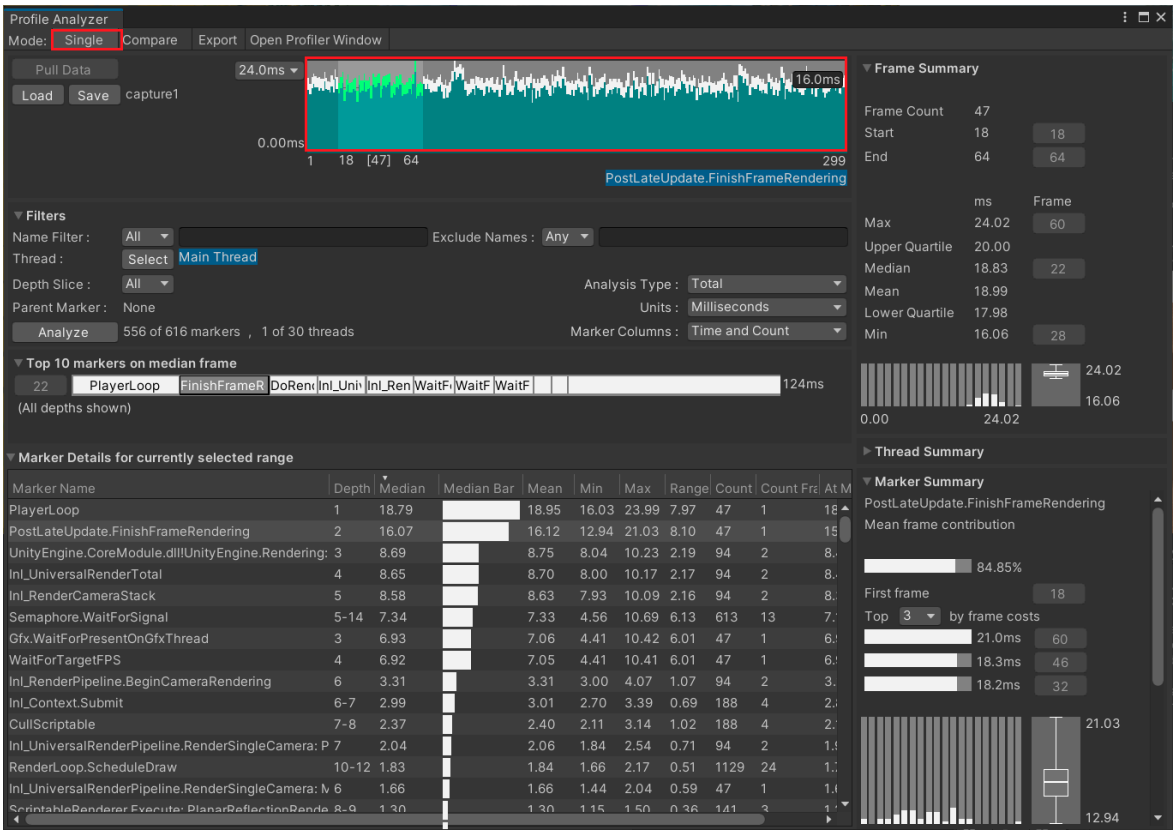
창 상단에서 **Mode** 옵션 중 하나를 선택할 수 있습니다. Profile Analyzer는 프로파일링 데이터 분석을 위한 다양한 뷰와 접근 방식을 제공합니다. 다양한 뷰를 사용하여 프로파일링 데이터 세트를 선택, 정렬, 조회, 비교해 보세요.



패널 상단의 여러 모드 중에서 선택할 수 있습니다.

Single 뷰

Single 뷰는 Profile Analyzer의 기본적인 시작 화면으로, 시간 경과에 따른 성능 변화를 개략적으로 한눈에 확인할 수 있습니다. Single 뷰에는 캡처한 프로파일 데이터의 단일 세트에 대한 정보가 표시됩니다. 프레임 전반의 프로파일 마커 성능을 분석하는 데 사용되는 뷰입니다. Single 뷰는 여러 패널로 구성되며, 타이밍에 대한 정보뿐만 아니라 프레임, 스레드, 마커의 최솟값, 최댓값, 중간값, 평균값, 상위/하위 사분위 값 등을 제공합니다.



Single 뷰에는 단일 프레임 또는 특정 범위 프레임의 프로파일 마커 통계와 타이밍이 표시됩니다.

여러 유용한 인사이트를 제공하는 Single 뷰는 모든 프로파일링 툴킷의 필수적인 부분입니다.

Compare 뷰

Compare 뷰는 데이터 세트 2개를 로드하여 나란히 비교할 수 있도록 서로 다른 색상으로 표시하며, 성능 차이를 분석할 때 특히 유용합니다.

Compare 뷰의 Marker Comparison 창과 컬러 코딩을 사용하여 데이터 세트 마커 타이밍을 쉽게 비교할 수 있습니다.

아래의 단계에 따라 Profile Analyzer를 사용하여 성능 변화를 비교하세요. 활성 Unity 프로파일러 캡처의 **Pull Data** 옵션을 사용하거나 저장된 세션의 **Load Data** 옵션을 사용할 수 있습니다. 로드할 때는 파일이 Profile Analyzer의 .pdata 포맷이어야 합니다. Unity 프로파일러 .data 파일의 경우 먼저 프로파일러 창에서 해당 파일을 연 다음 Profile Analyzer에서 Pull Data를 사용하세요. 프로파일러의 원본 .data 파일을 저장해 두는 것이 좋습니다.

1. **테스트 준비:** 유의미한 벤치마크 비교가 이루어질 수 있도록 게임에서 프로파일링할 일관된 섹션을 선택합니다. 스크립팅되었거나 반복 가능한 수동 플레이가 가장 효과적으로, 성능에 영향을 주는 무작위성 부작용을 최소화할 수 있습니다.
2. **'비포' 데이터 캡처**:
   - **Window > Analysis > Profile Analyzer**에서 Profile Analyzer를 엽니다.
   - Unity 프로파일러에서 최적화를 적용하기에 앞서 선택한 게임플레이의 프로파일링 세션을 기록합니다.
   - Analyzer의 **Compare** 탭에서 첫 번째 **Pull Data** 버튼을 클릭합니다. 그러면 프로파일러의 현재 캡처가 로드되며, 아니면 세션을 저장해도 됩니다.
3. **최적화 후 '애프터' 데이터 캡처**:
   - 코드 또는 성능 개선을 적용합니다.
   - Unity 프로파일러의 이전 데이터를 지운 다음 동일한 게임플레이의 새로운 프로파일링 세션을 기록합니다.
   - Profile Analyzer에서 두 번째 Pull Data 버튼을 클릭하여 새 세션을 로드합니다.

4. **차이 분석**:

   — **Marker Comparison** 창에 '비포'(왼쪽) 및 '애프터'(오른쪽) 캡처의 마커 타이밍 차이가 표시됩니다.

   — **<** 또는 **>** 기호로 표시된 열은 해당 지표에 대해 어느 캡처의 값이 더 큰지 나타냅니다.

   — **Marker Columns** 필터를 사용하여 비교되는 지표를 변경할 수 있습니다.

각 Marker Comparison 열에 대한 자세한 내용은 Compare 뷰 항목 페이지를 참고하세요.

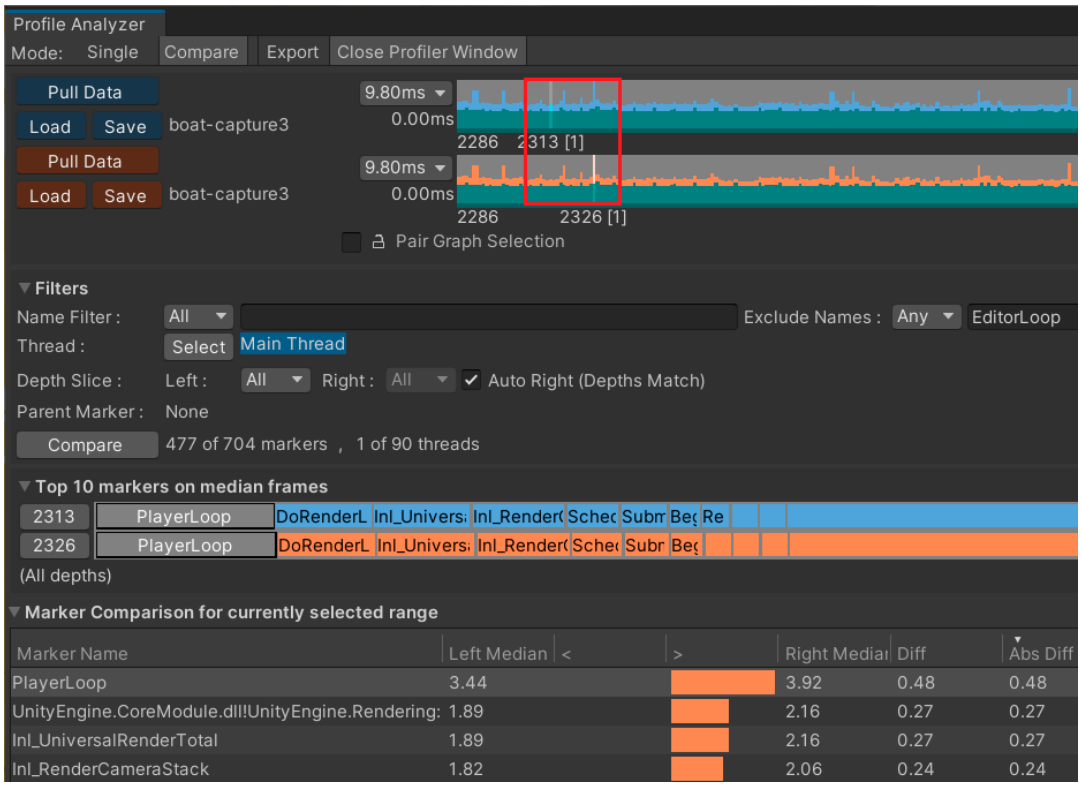
중간 프레임과 최장 프레임 비교

단일 프로파일러 캡처 내의 중간 프레임과 최장 프레임을 비교하면 중간 프레임에서는 발생하지 않지만 최장 프레임에서는 발생하는 문제를 정확하게 찾아내거나, 완료하는 데 평균보다 오래 걸리는 부분이 어디인지 확인할 수 있습니다.

Profile Analyzer의 Compare 뷰를 열고 왼쪽과 오른쪽에 모두 동일한 데이터 세트를 로드합니다. Single 뷰에서 데이터 세트를 로드한 다음 Compare 뷰로 전환해도 됩니다.

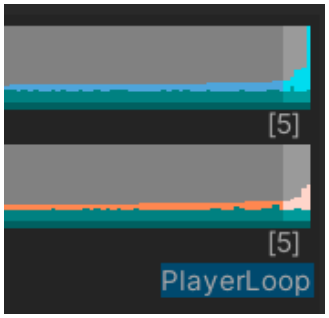
위쪽 **프레임 제어** 그래프를 오른쪽 클릭하고 **Select Median Frame**을 선택합니다. 아래쪽 그래프를 오른쪽 클릭하고 **Select Longest Frame**을 선택합니다.

Profile Analyzer의 Marker Comparison 패널이 업데이트되어 차이점을 표시합니다.



캡처의 중간 프레임과 최장 프레임 비교

두 그래프를 프레임 지속 시간으로 정렬(**오른쪽 클릭 > Order By Frame Duration**)한 다음, 각 세트에서 일반적인 수치를 크게 벗어난 프레임(불균형적으로 길거나 짧은 프레임)에 초점을 맞추거나 이를 제외한 범위를 선택하여 데이터를 비교할 수도 있습니다.



지속 시간순으로 프레임을 정렬하여 일반적인 수치를 크게 벗어난 범위 선택

이렇게 하면 가장 일반적인 프레임과 가장 극단적인 프레임을 비교할 수 있습니다. 선택된 범위의 데이터가 Marker Comparison 표에 표시되므로, 어떤 이유로 성능 스파이크나 비일관성 문제가 발생하는지 쉽게 분석할 수 있습니다.

Profiler Analyzer에 대해 더 알아보려면 다음 리소스를 참고하세요.

- Profile Analyzer 소개 및 튜토리얼
- Unity의 Profile Analyzer를 사용한 CPU 성능 분석
- 프로파일링 소개

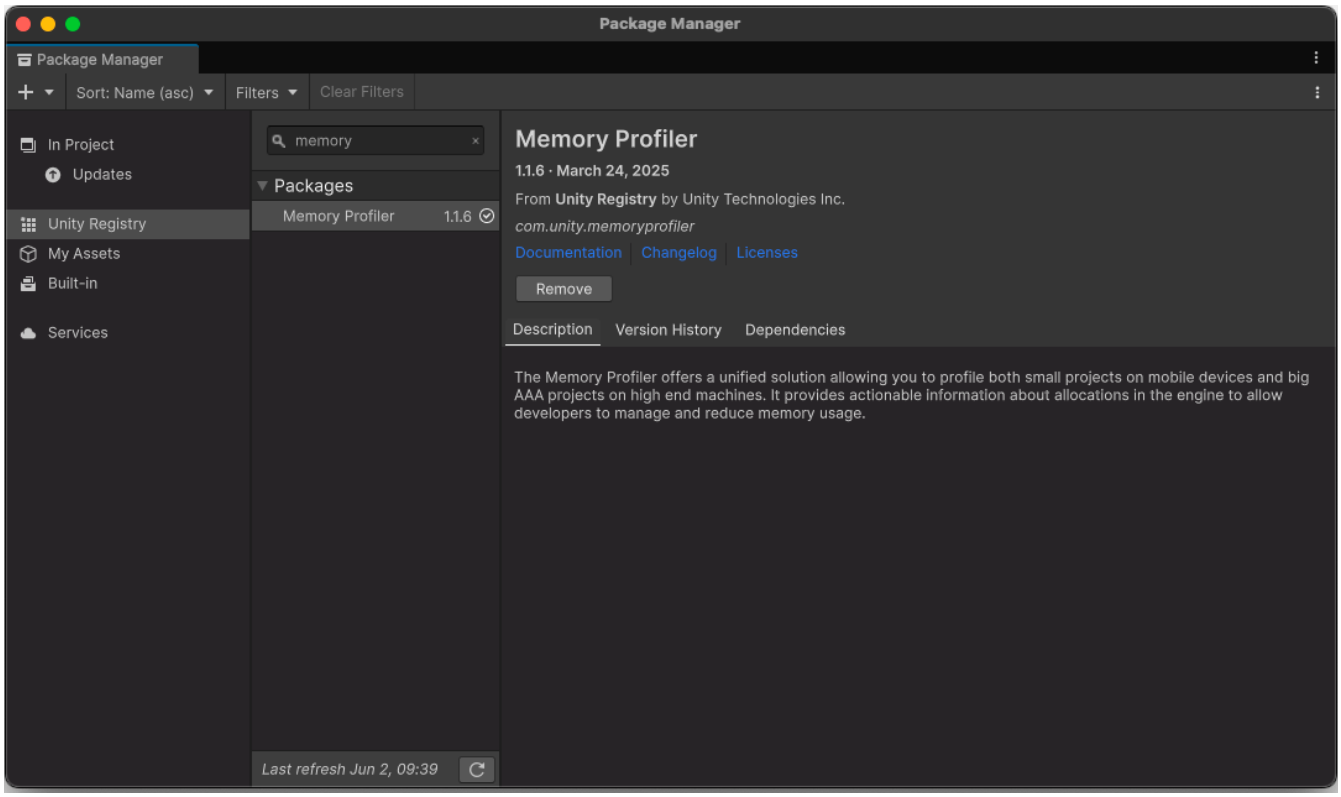
Profile Analyzer 팁

- **뎁스 레벨**을 **4**로 선택하면 Unity 엔진 API 수준은 무시하고 사용자 스크립트를 자세히 살펴볼 수 있습니다. 이 레벨로 필터링하고 **Timeline 모드**에서 Unity 프로파일러를 보면 호출 스택 뎁스와의 상관 관계를 파악하여 스크립트를 선택할 수 있으며, 이때 Monobehaviour 스크립트는 네 번째 레벨 아래에 파란색으로 표시됩니다. 이를 통해 특정 로직과 게임플레이 스크립트가 다른 노이즈 없이 그 자체로 부담을 주는지 빠르게 확인할 수 있습니다.
- 애니메이터나 엔진 물리 등 Unity 엔진의 다른 영역에서도 똑같은 방식으로 데이터를 필터링할 수 있습니다.
- **Frame Summary** 섹션의 오른쪽에는 메서드의 성능 범위 히스토그램이 강조 표시되어 나타납니다. **Max Frame** 번호(최대 타이밍이 발견된 프레임) 위에 마우스 커서를 올리면 클릭할 수 있는 링크가 표시되어 Unity 프로파일러에서 선택한 프레임을 확인할 수 있습니다. 이 뷰를 사용하여 최대 프레임 시간에 기여할 가능성이 있는 다른 요인을 분석할 수 있습니다.
- 와이드스크린 모니터가 있거나 모니터가 두 대 있다면 Profile Analyzer와 Unity 프로파일러를 나란히 열어서 확인하는 것이 편할 수 있습니다. 이러한 환경에서는 Profile Analyzer에서 프레임을 더블 클릭하면 Unity 프로파일러에서 동일한 프레임이 자동으로 선택되어 Timeline 또는 계층 구조 뷰에서 자세히 검토할 수 있습니다.

Memory Profiler

Memory Profiler 패키지를 사용하면 프로젝트의 메모리 사용량을 더 쉽게 파악하고 최적화할 수 있습니다. 특정 순간에 Unity 에디터에서, 그리고 타겟 기기에서 실행 중인 플레이어 빌드에서 애플리케이션의 메모리 '스냅샷'을 캡처할 수 있습니다.

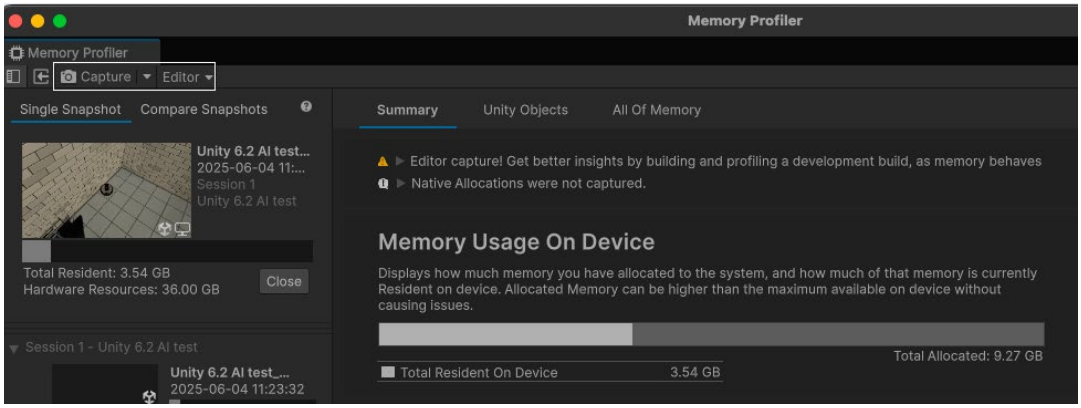
스냅샷은 메모리가 어떻게 사용되고 있는지를 종합적으로 요약해 보여 주며, 엔진 전반의 할당 상황을 나타냅니다. 이를 통해 과도하거나 불필요한 메모리 사용량의 출처를 식별하고, 메모리 누수를 추적하고, 힙 단편화와 같은 문제를 살펴볼 수 있습니다.



Memory Profiler는 패키지 관리자에서 사용할 수 있는 패키지입니다.

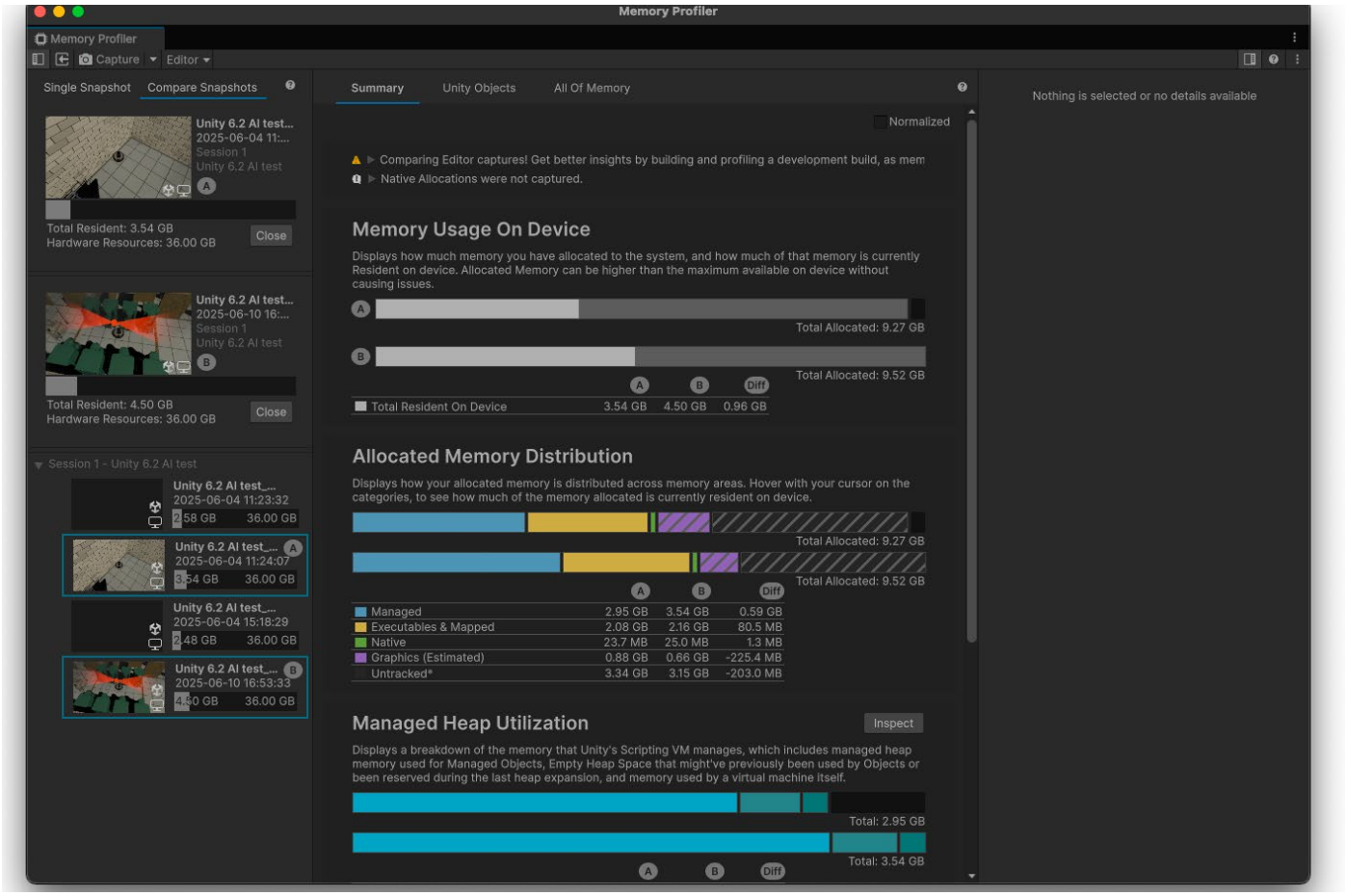
Memory Profiler 패키지를 설치한 후에는 **Window > Analysis > Memory Profiler**에서 Memory Profiler를 열 수 있습니다.

Memory Profiler의 상단 메뉴 바에서 플레이어 선택 대상을 변경하고 스냅샷을 캡처하거나 임포트할 수 있습니다. 왼쪽 상단의 타겟 선택 드롭다운(아래 이미지에서는 타겟이 'Editor'로 선택되어 있음)을 사용하면 Memory Profiler를 원격 기기에 연결하여 타겟 하드웨어에서 바로 메모리를 프로파일링할 수 있습니다. Unity 에디터에서 프로파일링하면 에디터나 기타 툴에서 추가되는 오버헤드로 인해 수치가 부정확해질 수 있습니다.



플레이어 선택을 변경하고 메모리 스냅샷을 캡처 또는 임포트합니다.

Memory Profiler 창 왼쪽에는 Snapshots 컴포넌트가 있습니다. 저장된 메모리 스냅샷을 이 영역에서 관리하거나 열고 닫을 수 있으며, Snapshot 컴포넌트는 Single Snapshot 및 Compare Snapshot이라는 2개의 뷰를 제공합니다.



여러 개의 메모리 스냅샷을 관리할 수 있습니다.

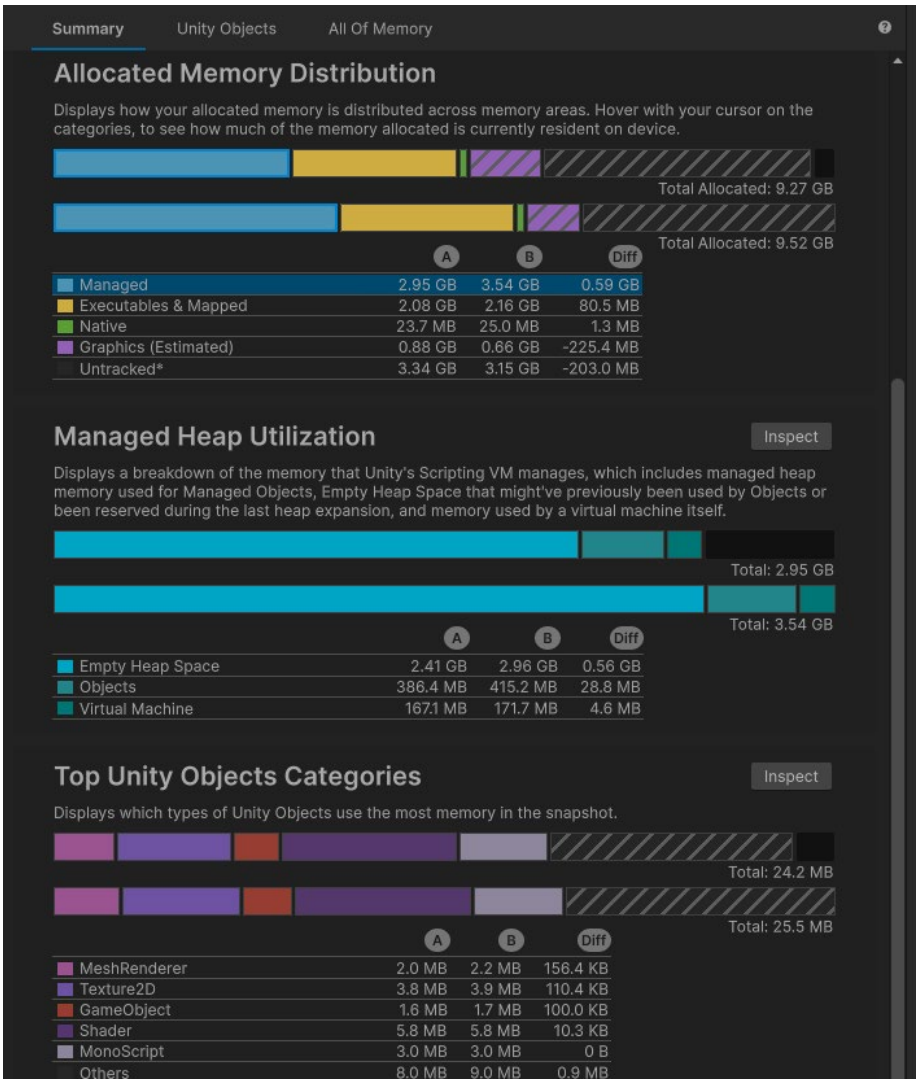
Memory Profiler를 사용하면 Profile Analyzer와 마찬가지로 두 개의 메모리 스냅샷을 로드하여 나란히 비교할 수 있습니다. 비교를 통해 시간 경과에 따른 메모리 증가를 추적하고, 각 씬 간의 사용량을 분석하고, 잠재적인 메모리 누수를 확인할 수 있습니다.

Memory Profiler의 메인 창에는 **Summary**, **Unity Objects**, **All Of Memory** 등 메모리 스냅샷을 자세히 살펴볼 수 있는 여러 탭이 있습니다. 각 탭에 대해 자세히 살펴보겠습니다.

Summary 탭

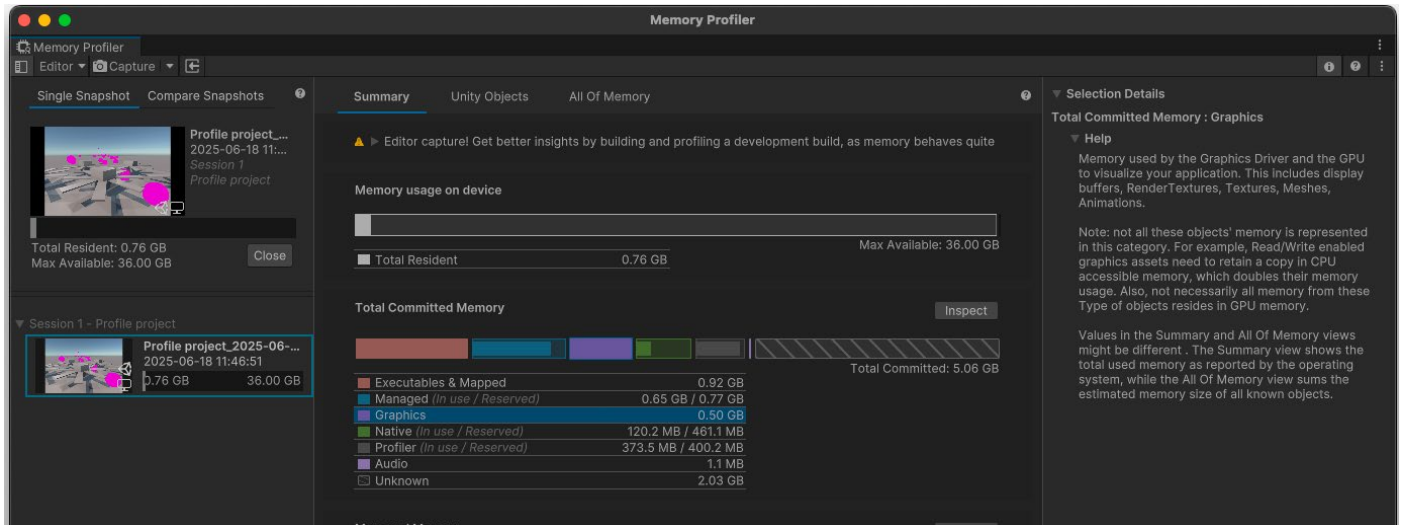
Summary 탭은 메모리가 캡처된 순간의 프로젝트 메모리 사용량에 대한 개괄적인 스냅샷을 제공합니다. 상세한 분석 없이 간단한 개요를 확인하려 할 때 매우 유용합니다.

이 뷰는 주요 지표를 강조하여 나타내며, 잠재적인 메모리 문제나 예기치 않은 사용량 패턴을 빠르게 확인하는 데 도움이 됩니다. 특히 스냅샷을 비교하거나 시간 경과에 따른 메모리 사용량을 디버깅할 때 적합합니다. 이 탭의 주요 섹션을 살펴보겠습니다.



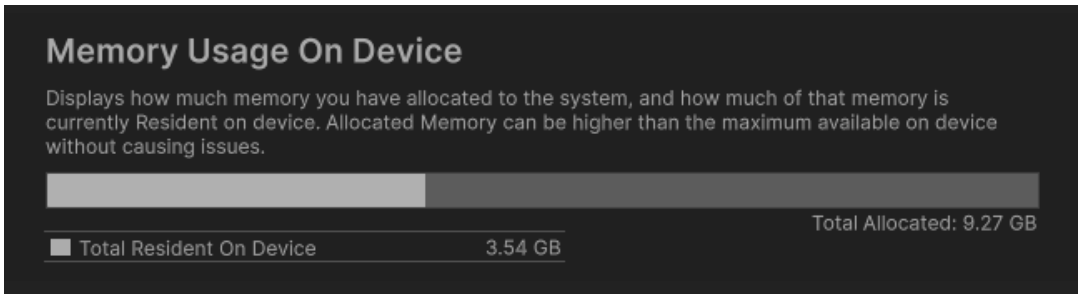
Summary 탭에는 스냅샷이 캡처된 시점의 메모리 개요가 표시됩니다.

팁: 오른쪽 패널에는(아래 이미지 참고) 스냅샷에 대한 유용한 컨텍스트 정보가 표시되어 잠재적인 문제를 살펴보거나 결과를 해석하는 데 도움을 줍니다.

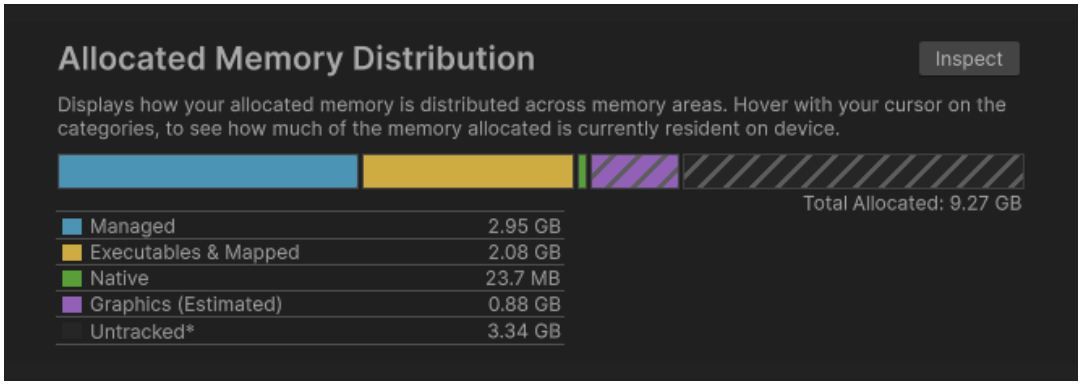


오른쪽 패널은 스냅샷에 대한 유용한 정보를 제공합니다.

Memory Usage On Device: 물리적 메모리에서 애플리케이션의 사용량을 보여 줍니다. 스냅샷이 캡처된 시점에 메모리에 있던 모든 Unity 및 Unity 외 할당이 포함됩니다.

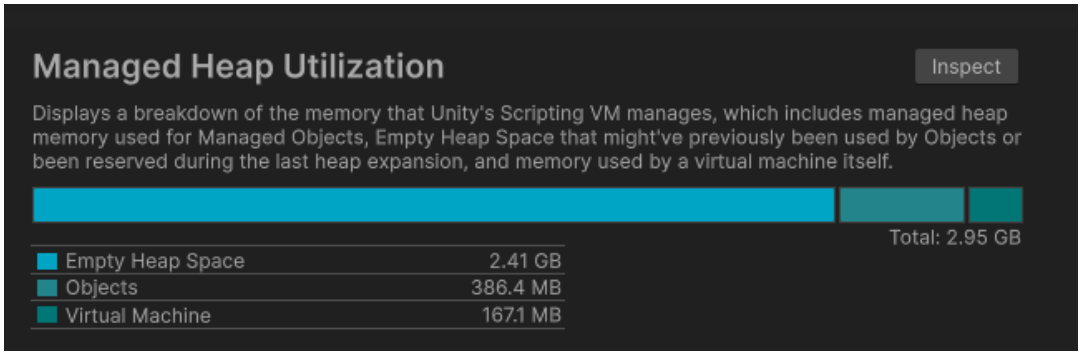


Allocated Memory Distribution: 이 뷰는 할당된 메모리가 각 메모리 카테고리에 분포된 방식을 가시적으로 표시합니다.

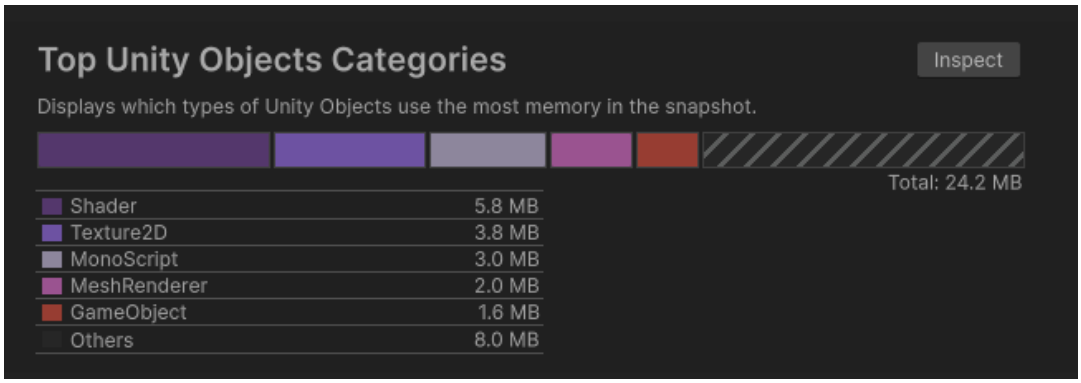


Untracked* 메모리 바는 Unity가 메모리 관리 시스템을 통해 추적하지 않는 메모리를 나타냅니다. 이는 네이티브 플러그인과 드라이버로부터 할당된 메모리일 수 있습니다. 타겟 기기의 추적되지 않은 메모리 사용량을 분석하려면 플랫폼 전용 프로파일러를 사용하세요.

Managed Heap Utilization: 이 뷰는 Unity의 스크립팅 VM이 관리하는 메모리의 분석 정보를 표시합니다. 여기에는 관리되는 오브젝트에 사용되는 관리되는 힙, 이전에 오브젝트에 의해 사용되었을 수 있거나 마지막 힙 확장 중에 예약된 빈 힙 공간, 가상 머신에 의해 사용되는 메모리가 포함됩니다.

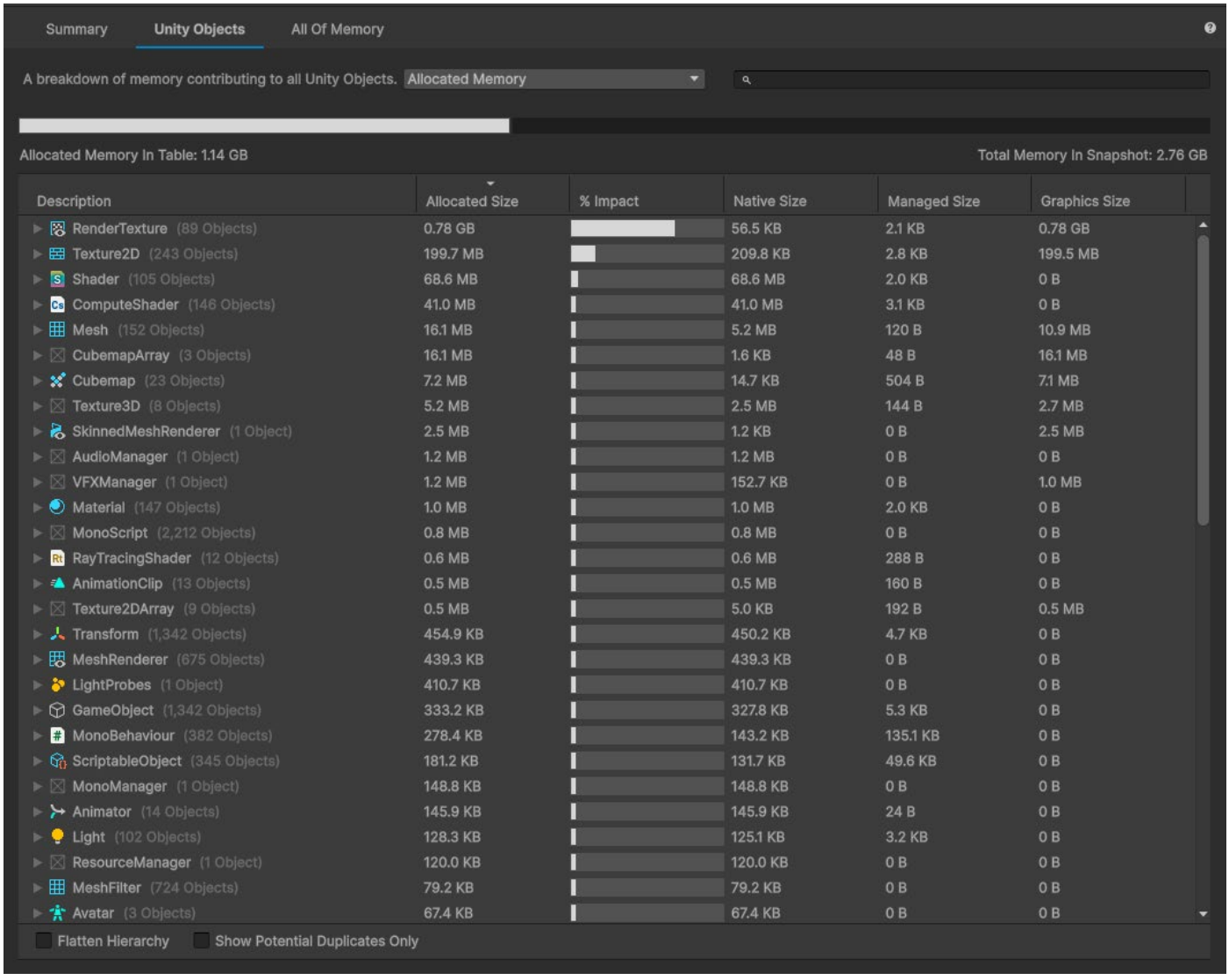


Top Unity Object Categories: 스냅샷에서 어느 유형의 Unity 오브젝트가 가장 많은 메모리를 사용하는지 표시합니다(예: Texture2D, 메시, 게임 오브젝트).



Unity Objects 탭

Unity Objects 탭에는 메모리를 할당한 Unity 오브젝트, 해당 오브젝트가 사용하는 네이티브 메모리와 관리되는 메모리의 양, 총 합계가 표시됩니다. 이 정보를 사용하여 중복된 메모리 항목을 제거할 수 있는 영역을 식별하거나 많은 양의 메모리를 사용하는 오브젝트를 식별할 수 있습니다. 또한 검색창을 사용하면 표에서 입력한 텍스트가 포함된 항목을 찾을 수 있습니다.



Unity Objects 탭을 이용하면 캡처된 스냅샷 메모리 사용량을 세밀하게 살펴볼 수 있습니다.

표에는 기본적으로 모든 관련 오브젝트가 할당된 크기를 기준으로(내림차순) 정렬됩니다. 열 헤더 이름을 클릭하여 표를 해당 열로 정렬하거나, 내림차순 정렬과 오름차순 정렬을 변경할 수 있습니다.

메모리 사용량을 최적화하고 메모리 할당량이 제한된 하드웨어 플랫폼에서 메모리를 더 효율적으로 패킹하고자 할 때 이 기능을 사용할 수 있습니다.

메모리 프로파일링 기술 및 워크플로

먼저 Memory Profiler 스냅샷을 분석하여 메모리 사용량이 높은 영역을 식별합니다. Memory Profiler 스냅샷을 캡처 또는 로드한 후 Unity Objects 탭을 사용하여 메모리 사용량이 큰 순서로 정렬된 카테고리를 살펴봅니다.

대개 프로젝트 에셋에서 메모리를 가장 많이 소비합니다. Table 모드를 사용하면 텍스처, 메시, 오디오 클립, 렌더 텍스처, 셰이더 배리언트, 사전 할당된 버퍼를 찾을 수 있습니다. 이는 메모리 사용량을 최적화할 때 우선적으로 살펴보는 경우가 많은 항목입니다. Project Auditor는 에셋의 메모리 사용량을 줄이기 위한 권장 사항을 제공하는 유용한 보조 툴입니다. 먼저 **Import Settings** 인스펙터(Inspector)에서 에셋이 올바르게 설정되어 있는지 확인하세요.

메모리 누수 찾기

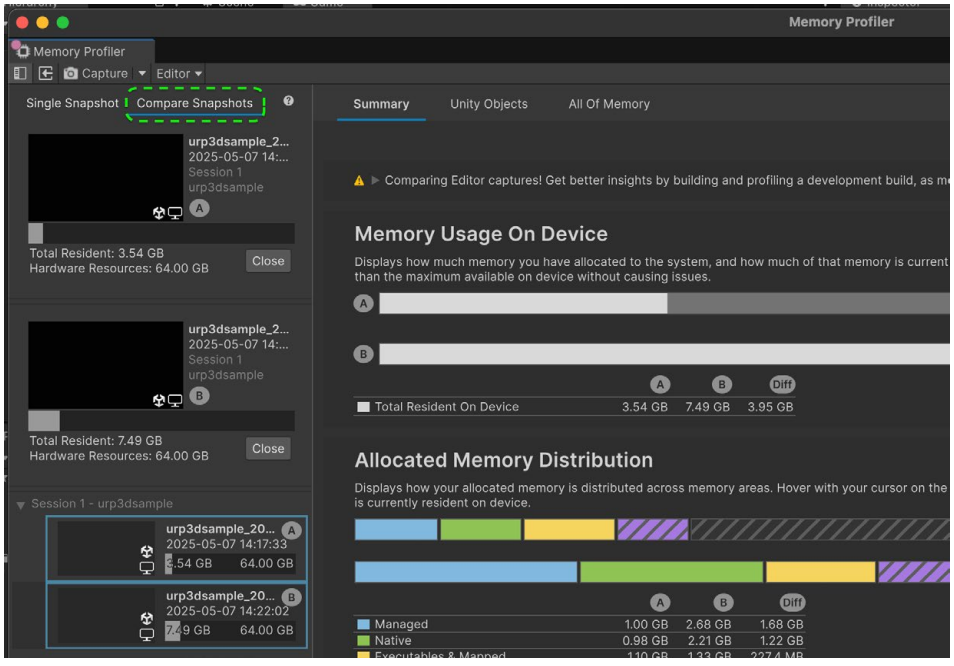
메모리 누수는 사용되지 않는 에셋, 오브젝트 또는 리소스가 메모리에서 올바르게 해제되지 않은 상황을 가리킵니다. 이러한 상황은 메모리 사용량의 점진적인 증가와 성능 문제 또는 크래시로 이어질 수 있습니다.

메모리 누수는 일반적으로 다음과 같은 경우에 발생합니다.

— 오브젝트가 코드를 통해 메모리에서 수동으로 해제되지 않는 경우

— 오브젝트가 다른 오브젝트에 의해 참조되고 있어 메모리에 의도치 않게 남아 있는 경우

Memory Profiler의 Compare Snapshots 모드를 사용하면 특정 기간에 걸친 두 개의 스냅샷을 비교하여 메모리 누수를 찾을 수 있습니다. 할당이 해제되어야 함에도 메모리에 남아 있는 오브젝트를 이렇게 비교하며 찾아낼 수 있습니다.



두 개의 스냅샷을 비교하여 차이점을 확인합니다.

Unity 게임에서는 주로 씬을 언로드한 후에 메모리 누수가 발생합니다. 언로드된 씬에서 오브젝트에 대한 참조가 여전히 유지되는 경우, 해당 오브젝트에 대한 가비지 컬렉션이 올바르게 이루어지지 않을 수 있습니다.

애플리케이션 수명 동안 반복되는 메모리 할당 찾기

여러 메모리 스냅샷의 차이를 비교하면 애플리케이션 수명 동안 지속되는 메모리 할당의 출처를 식별할 수 있습니다.

다음은 프로젝트에서 관리되는 힙 할당을 식별하는 데 도움이 되는 팁입니다.

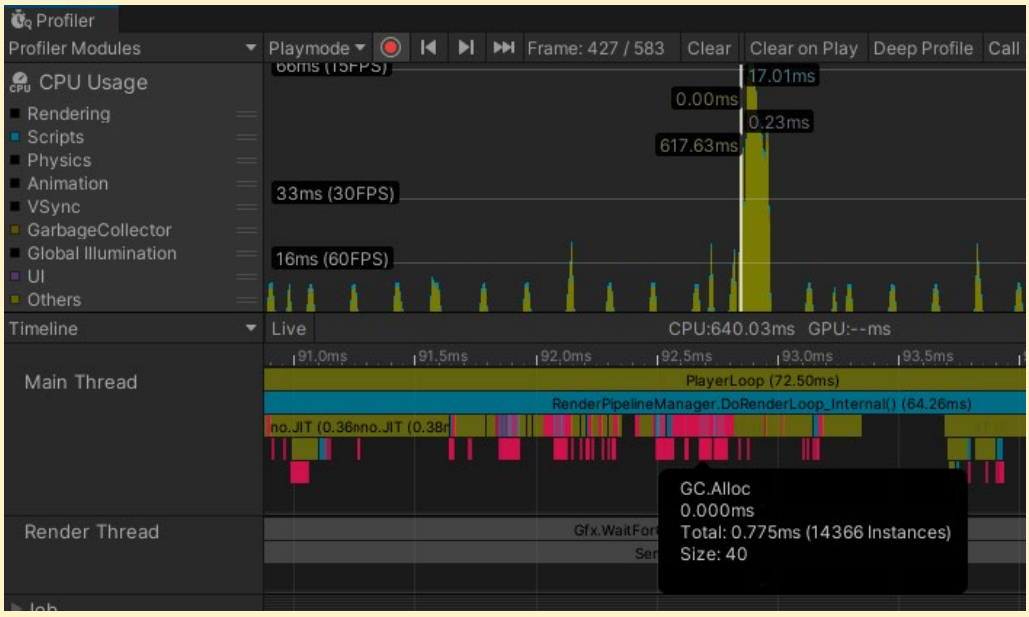
Unity 프로파일러의 Memory Profiler 모듈

Unity 프로파일러의 Memory Profiler 모듈은 프레임당 관리되는 할당을 빨간색 선으로 표시합니다. 대개 이 값은 0이어야 하며, 이 선에 스파이크가 있는 경우 해당 프레임의 관리되는 할당을 조사해야 합니다.

GC Allocated In Frame에서 스파이크가 발생하면 관리되는 할당을 조사할 수 있는 포인터가 나타납니다.

CPU Usage Profiler의 Timeline 뷰

CPU Usage Profiler의 Timeline 뷰에는 관리되는 할당을 비롯한 할당이 분홍색으로 표시되므로 할당을 쉽게 확인하고 살펴볼 수 있습니다.



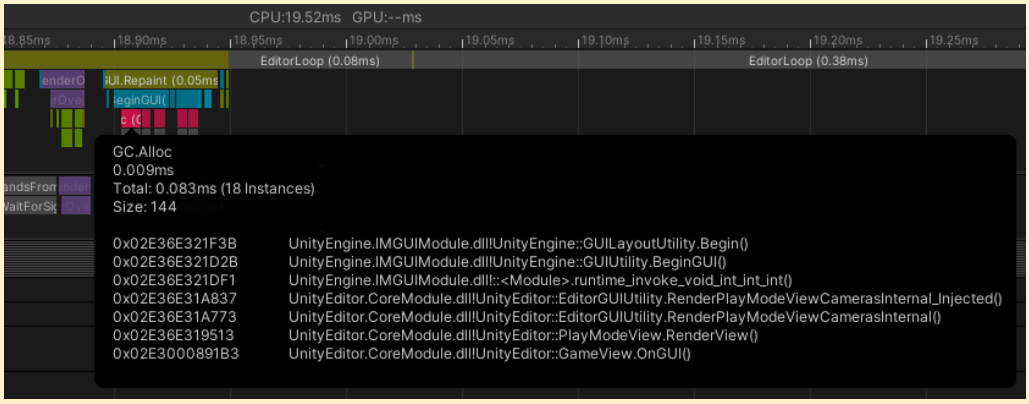
관리되는 할당은 Timeline 뷰에 분홍색 마커로 표시됩니다.

Allocation 호출 스택

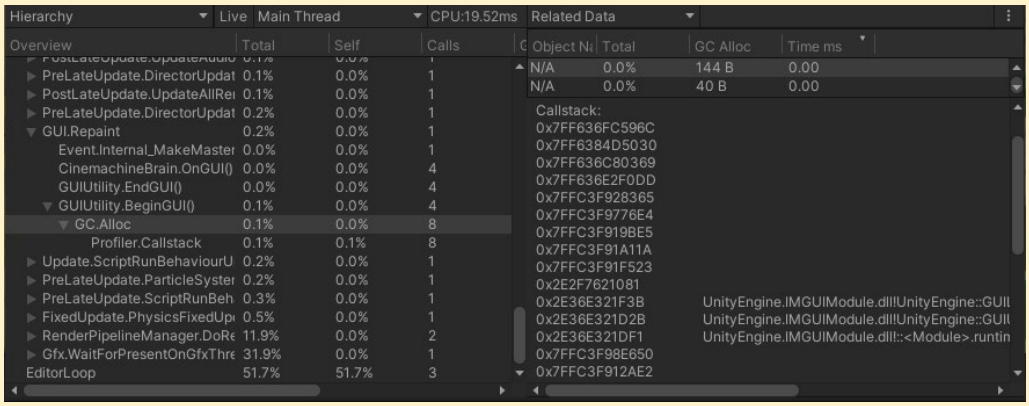
Allocation 호출 스택을 사용하면 코드에서 관리되는 메모리 할당을 빠르게 찾을 수 있습니다. 딥 프로파일링으로 발생하는 것보다 적은 오버헤드로 호출 스택에 대해 필요한 세부 정보를 확인할 수 있으며, 기본 프로파일러를 사용하여 즉시 활성화할 수 있습니다.

Allocation 호출 스택은 Profiler에서 기본적으로 비활성화되어 있습니다. 활성화하려면 Profiler 창의 메인 툴바에서 **Call Stacks** 버튼을 클릭합니다. 그런 다음 **Details** 뷰를 **Related Data**로 변경합니다.

참고: 이전 버전의 Unity(Allocation 호출 스택을 지원하기 전 버전)를 사용하는 경우 딥 프로파일링을 사용하면 전체 호출 스택을 확보하여 관리되는 할당을 더욱 쉽게 찾을 수 있습니다.



Profiler에서 Allocation 호출 스택을 활성화하면 호출 스택을 추적하여 관리되는 할당의 출처를 확인할 수 있습니다.

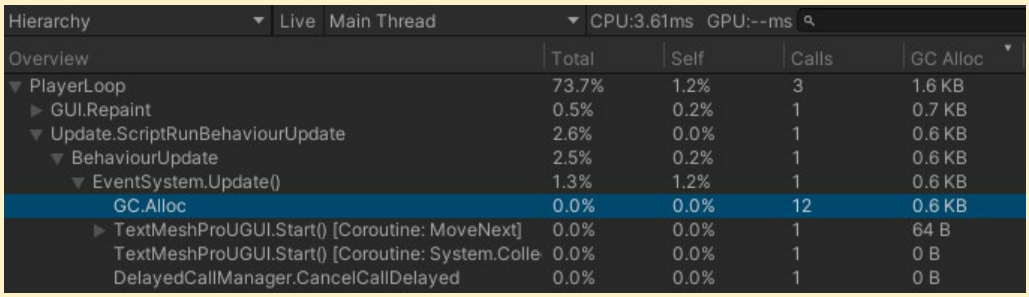


계층 구조 뷰의 Related Data 패널에도 Allocation 호출 스택의 세부 정보가 표시됩니다.

이제 계층 구조나 원시 계층 구조에서 선택한 GC.Alloc 샘플에 해당 호출 스택이 포함됩니다. Timeline의 선택 툴팁에서도 GC.Alloc 샘플의 호출 스택을 확인할 수 있습니다.

CPU Usage Profiler의 계층 구조 뷰

CPU Usage Profiler의 계층 구조 뷰에서 열 헤더를 클릭하여 정렬 기준으로 사용할 수 있습니다. GC Alloc을 기준으로 정렬하면 GC Alloc을 집중적으로 살펴볼 수 있습니다.



CPU Usage Profiler 모듈의 계층 구조 뷰를 사용하면 관리되는 할당을 필터링하여 집중적으로 살펴볼 수 있습니다.

메모리 및 GC 최적화

GC(가비지 컬렉션)의 영향 줄이기

Unity는 Boehm-Demers-Weiser 가비지 컬렉터를 사용하여 애플리케이션에 더 이상 필요하지 않은 메모리를 자동으로 정리합니다. GC가 프로그램 코드의 실행을 중단하고 작업이 완료된 후에 정규 실행을 재개합니다.

자동 관리는 편리한 방식이지만, 가비지 컬렉터가 게임을 일시 정지하여 사용되지 않는 메모리를 정리해야 하므로 불필요하거나 잦은 할당이 성능 문제로 이어질 수 있습니다(GC 스파이크). 유의해야 하는 일반적인 문제는 다음과 같습니다.

— **문자열:** C#에서 문자열은 값 유형이 아닌 레퍼런스 유형입니다. 다시 말해서 모든 새 문자열은 일시적으로만 사용되더라도 관리되는 힙에 할당됩니다. 따라서 불필요한 문자열의 생성 또는 조작을 줄여야 합니다. 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 임시 오브젝트 배열을 반환하는 힙 할당을 생성합니다. 레퍼런스를 루프 도중에 할당하지 말고 배열에 캐싱하세요. 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).

  Project Auditor를 사용하여 이러한 대안을 나열할 수 있으며, 이렇게 하면 가능한 한 항상 비할당 버전을 사용하도록 할 수 있습니다.

— **박싱:** 박싱은 값 유형(예: int, float, struct)이 레퍼런스 유형(예: 오브젝트)으로 변환된 경우에 발생합니다. 레퍼런스 유형의 변수 대신 값 유형 변수를 전달하는 방식은 지양하세요. 그렇게 하면 임시 오브젝트가 생성되며 동반되는 잠재적 가비지가 값 유형을 암묵적으로 타입 오브젝트로 변환하기 때문입니다(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 방식 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 List를 MonoBehaviour의 멤버로 만들고 Start로 초기화하세요. 사용 전에 모든 프레임에서 **Clear**로 컬렉션을 비우면 됩니다.

가능한 경우 가비지 컬렉션 시간 측정하기

가비지 컬렉션 멈춤 현상이 게임의 특정 지점에 영향을 주지 않는다면 System.GC.Collect로 가비지 컬렉션을 트리거할 수 있습니다. 사용자가 메뉴를 보거나 게임을 일시 정지하는 경우가 대표적인 예로, 이러한 경우에는 사용자가 가비지 컬렉션이 실행되는 것을 알아채지 못합니다.

자동 메모리 관리의 이해에서 이 방식을 활용하는 방법을 참고하세요.

점진적 가비지 컬렉터를 활용하여 GC 워크로드 분할하기

점진적 가비지 컬렉션을 사용하면 프로그램 실행 중에 한 번 길게 중단되는 것이 아니라 다수의 프레임에 워크로드를 분산하는 짧은 중단이 여러 번 나타납니다. 가비지 컬렉션으로 인해 프레임 속도가 불규칙해진다면 이 옵션을 사용하여 GC 스파이크를 줄일 수 있는지 확인해 보세요. Profile Analyzer를 사용하여 이 방식이 애플리케이션에 도움이 되는지 확인하세요.

점진적 모드에서 GC를 사용하면 일부 C# 호출에 읽기-쓰기 배리어가 추가될 수 있으며, 이에 따라 스크립팅 호출 오버헤드가 프레임당 최대 1ms까지 늘어날 수 있습니다. 최적의 성능을 위해서는 원활한 프레임 속도를 위해 점진적 GC를 사용할 필요가 없도록 메인 게임플레이 루프에 GC Alloc을 사용하지 않는 것이 좋습니다. 예를 들어 메뉴를 열거나 새로운 레벨을 로드하는 시점에 사용자가 알 수 없는 곳에 GC.Collect를 숨길 수 있습니다. 이처럼 최적화된 시나리오에서는 System.GC.Collect()를 사용하여 완전하고 점진적이지 않은 가비지 컬렉션을 수행할 수 있습니다.

Memory Profiler에 대해 자세히 알아보려면 다음 자료를 참고하세요.

- Memory Profiler 소개 및 튜토리얼
- Memory Profiler 기술 자료
- Unity Memory Profiler로 메모리 사용량 개선하기
- Memory Profiler: 메모리 관련 문제 해결을 위한 툴
- Memory Profiler 사용하기

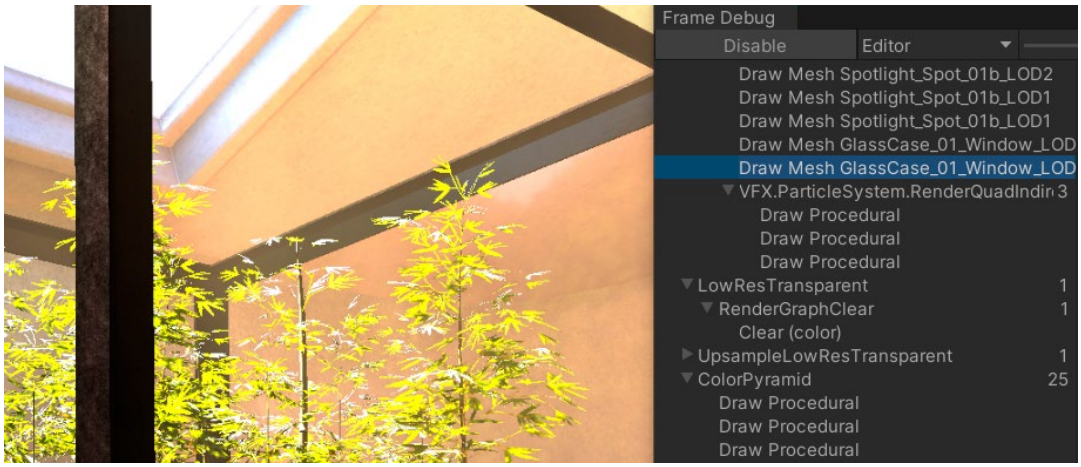
프레임 디버거

프레임 디버거를 사용하면 실행 중인 게임을 특정 프레임에서 멈추고 해당 프레임의 렌더링에 사용된 개별 드로우 콜을 확인하여 렌더링을 디버깅하고 최적화할 수 있습니다. 드로우 콜 목록을 하나씩 살펴볼 수 있으므로 그래픽 요소에서 씬을 형성하는 프레임이 어떻게 구성되는지 확인할 수도 있습니다.

프레임 디버거를 사용하면 드로우 콜이 게임 오브젝트의 지오메트리에 대응되는 부분을 쉽게 확인할 수 있습니다. 메인 계층 구조 패널에서 게임 오브젝트가 강조 표시되어 즉시 확인 가능합니다.

드로우 콜의 이해:
Unity의 드로우 콜은 특정 머티리얼과 셰이더를 사용하여 특정 지오메트리 세트(메시, 스카이박스, 사용자 인터페이스 등)를 렌더링하도록 CPU에서 GPU로 전송되는 요청입니다. Unity는 전과 다른 오브젝트, 머티리얼 또는 상태 변화를 렌더링해야 할 때마다 새 드로우 콜을 전송합니다.

또한 프레임 디버거를 사용하면 렌더링 순서를 프레임별로 분석하여 오버드로우를 테스트할 수 있습니다. 더 자세한 내용은 아래의 최적화 팁을 참고하세요.

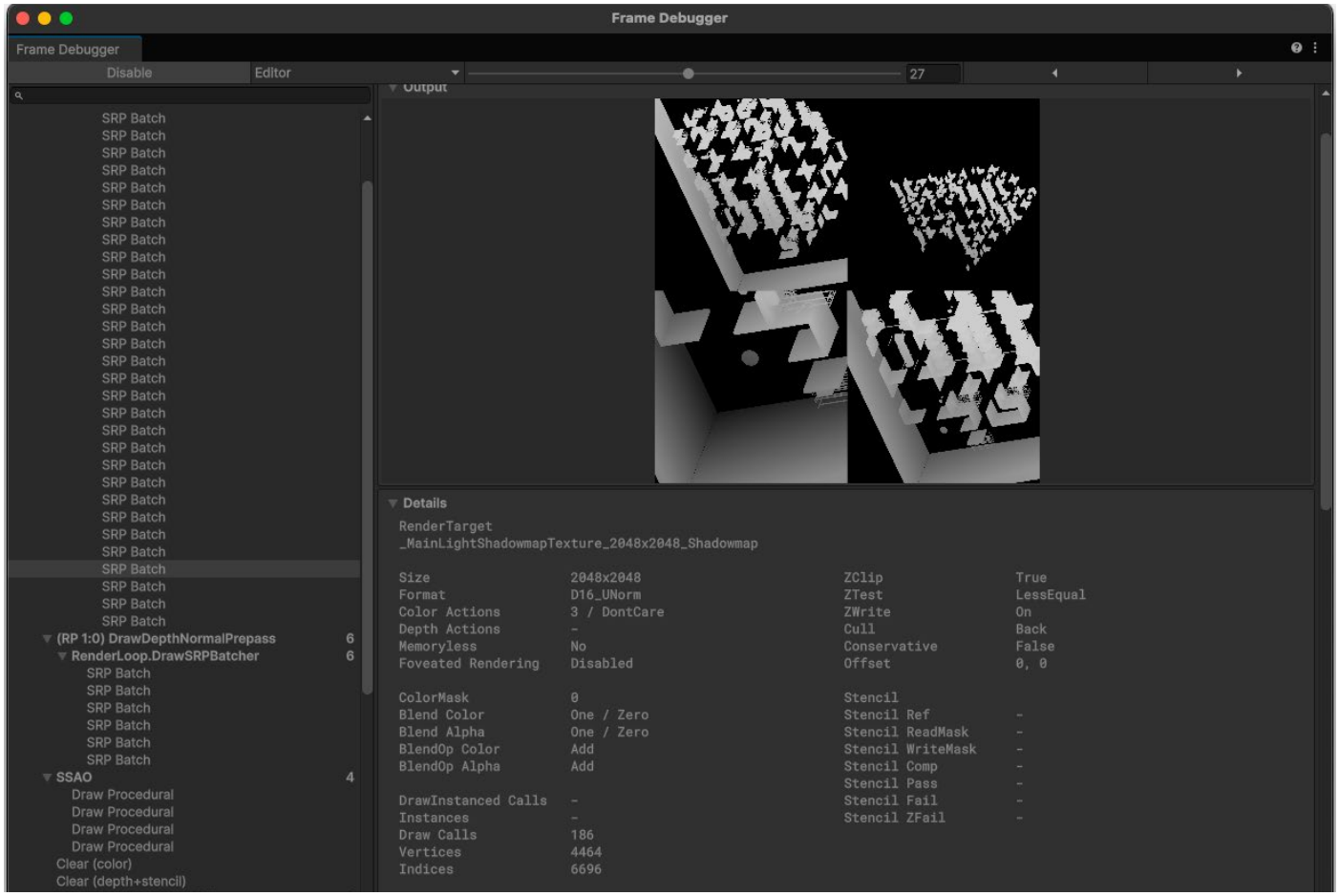


발견된 오버드로우가 어떻게 발생하는지 프레임 디버거를 사용하여 분석합니다.

Window > Analysis > Frame Debugger 메뉴에서 프레임 디버거를 엽니다.

에디터나 기기에서 애플리케이션을 실행 중인 상태로 **Enable**을 클릭합니다. 그러면 애플리케이션이 일시 중지되고 Frame Debugger 창 왼쪽에 현재 프레임에 대한 모든 드로우 콜이 캡처되어 순서대로 표시됩니다. 캡처에는 프레임버퍼 제거 이벤트 등의 추가 세부 정보도 포함됩니다.

Debugger 창 상단의 슬라이더를 사용해 드로우 콜을 빠르게 스크럽하여 확인하려는 항목을 금방 찾을 수 있습니다.



Frame Debugger 창 왼쪽에 드로우 콜과 이벤트 목록이 표시되고 슬라이더를 통해 각 항목을 시각화하여 단계별로 살펴볼 수 있습니다.

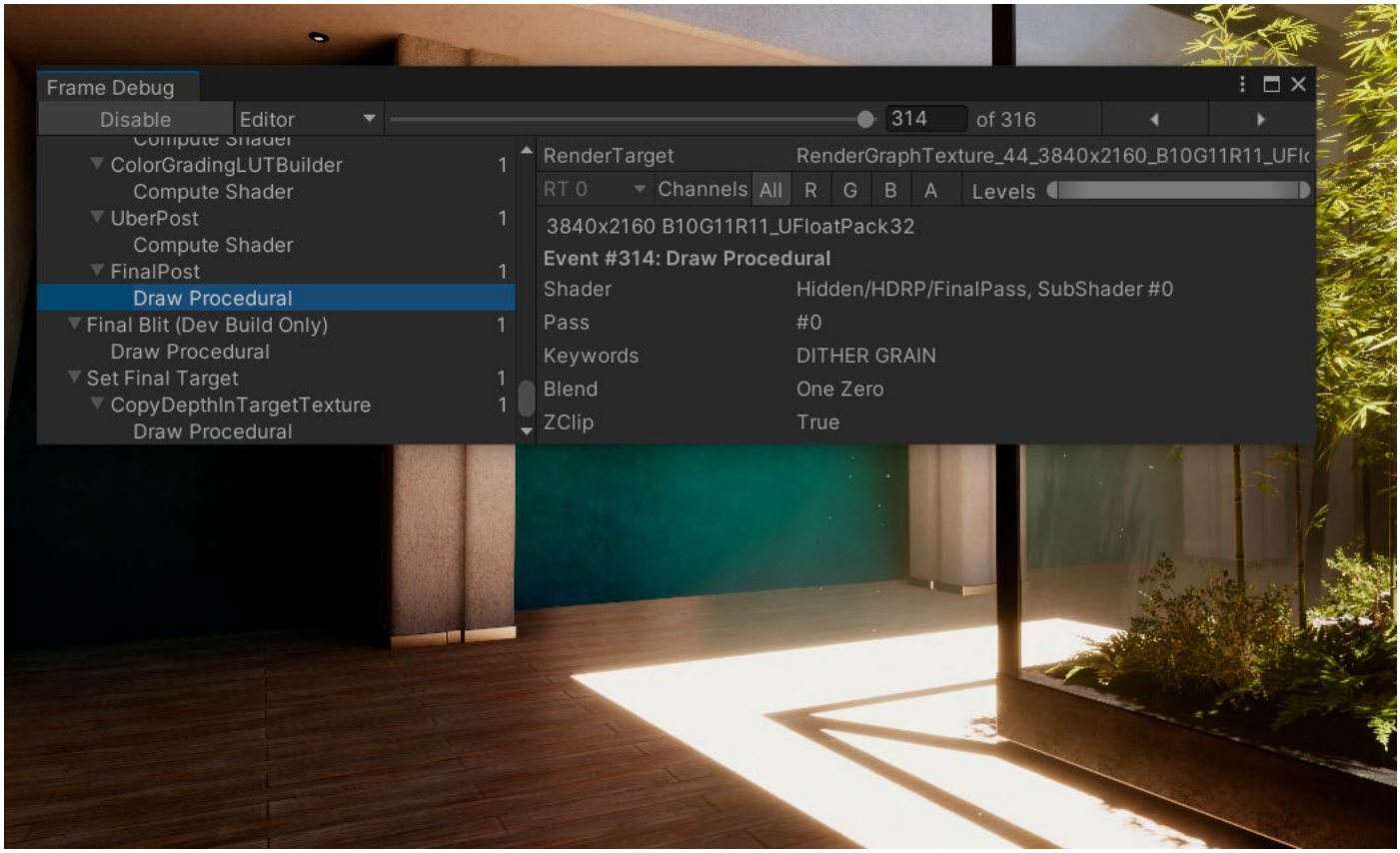
Unity는 CPU에서 그래픽스 API로 드로우 콜을 발행하여 화면에 지오메트리를 드로우합니다. 드로우 콜은 그래픽스 API에 무엇을 어떻게 드로우해야 하는지 알려 줍니다. 각 드로우 콜에는 텍스처, 셰이더, 버퍼에 관한 정보 등 그래픽스 API에 필요한 모든 정보가 포함되어 있습니다. 드로우 콜을 준비하는 과정이 드로우 콜 자체보다 리소스를 더 많이 사용하는 경우도 많습니다.

이 준비 과정을 렌더 상태라고 하며, 이 상태의 변경을 최소화하여 성능을 향상할 수 있습니다.

프레임 디버거를 사용하여 어디에서 드로우 콜을 보냈는지 식별하고 렌더링 과정을 시각화할 수 있습니다. 이 작업은 렌더링 상태 변화를 줄이기 위해 드로우 콜을 그룹화할 방법에 관한 결정을 내리는 데 도움이 됩니다.

프레임 디버거의 목록 계층 구조를 참조하여 확인하려는 드로우 콜의 발생 위치를 파악할 수 있습니다. 목록에서 원하는 항목을 선택하면 게임 창에 해당 드로우 콜을 포함하는 씬이 나타납니다.

모든 호출이 CPU 오버헤드를 유발하므로 드로우 콜 최소화는 성능 향상을 위한 필수 작업이며, 특히 모바일이나 VR 플랫폼에서는 더욱 그렇습니다. 배칭, GPU 인스턴싱, 텍스처 아틀라싱 같은 기법은 동일한 머티리얼 또는 메시를 사용하는 오브젝트를 결합하여 드로우 콜 수를 줄입니다.



게임(Game) 창의 프레임 디버거에는 선택된 드로우 콜까지 포함된 씬 프레임 구조가 포스트 프로세싱 효과 적용이 끝날 무렵까지 표시됩니다. 목록 계층 구조의 오른쪽에 있는 패널에는 지오메트리 세부 정보와 렌더링에 사용된 셰이더 등 각 드로우 콜에 관한 정보가 표시됩니다.

그 외에도 드로우 콜을 이전 드로우 콜에 배칭할 수 없었던 이유나 셰이더에 적용된 정확한 프로퍼티 값에 대한 분석 데이터를 비롯한 유용한 정보를 확인할 수 있습니다.

원격 프레임 디버깅

지원되는 플랫폼에서 실행 중인 Unity 플레이어에 프레임 디버거를 연결하여 프레임을 원격으로 디버깅할 수 있습니다(WebGL은 지원되지 않음). 데스크톱 플랫폼의 경우 빌드에 **Run In Background**를 활성화하면 됩니다.

원격 프레임 디버깅은 다음과 같이 설정할 수 있습니다.

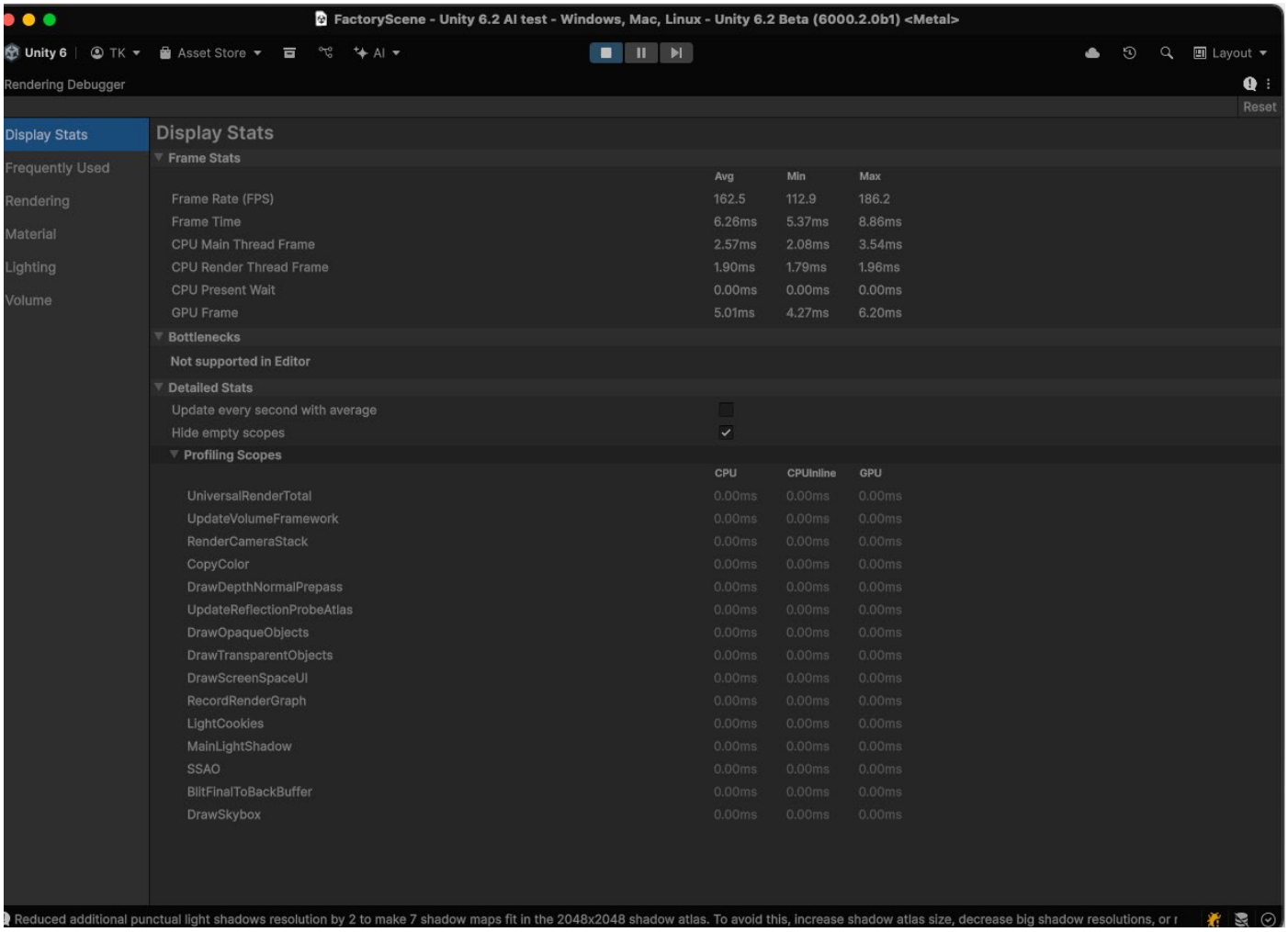
1. 타겟 플랫폼에 프로젝트의 표준 빌드를 생성합니다(Development Build 선택).
2. 플레이어를 실행합니다.
3. 에디터에서 Frame Debugger 창을 엽니다.
4. Player 선택 드롭다운을 클릭하고 실행 중인 활성 플레이어를 선택합니다.
5. Enable 버튼을 클릭합니다.

이렇게 하면 Frame Debug의 목록 계층 구조에서 드로우 콜과 이벤트를 단계별로 살펴보고 활성 플레이어에서 결과를 관찰할 수 있습니다.

렌더링 디버거

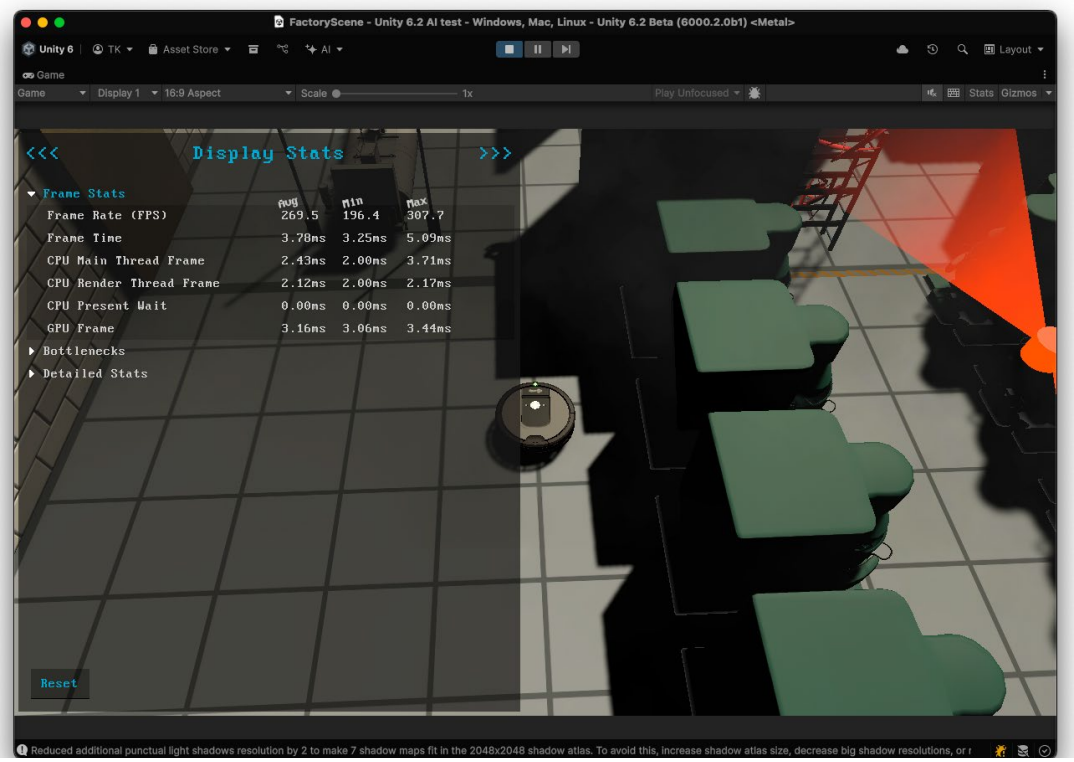
렌더링 디버거에는 오버드로우, 조명 복잡도, 렌더링, 머티리얼 프로퍼티에 대한 정보를 표시하는 다양한 디버그 뷰와 모드가 있어, 렌더링 문제를 파악하고 URP 및 HDRP에 맞게 씬을 최적화할 수 있습니다.

이 툴을 실행하려면 에디터에서 **Window > Analysis > Rendering Debugger**로 이동하세요. 플레이 모드 또는 데스크톱 플레이어 빌드에서 단축키 **왼쪽 Ctrl+Backspace**(**macOS에서는 왼쪽 Ctrl+Delete**)를 사용할 수도 있습니다.



렌더링 디버거로 다양한 조명, 렌더링, 머티리얼 프로퍼티를 시각화하여 렌더링 문제를 식별하고 씬을 최적화할 수 있습니다.

렌더링 디버거는 시각적 결함이나 렌더링 관련 성능 병목 현상을 진단하는 데 도움이 됩니다. 상세 통계가 표시되는 창은 개발 빌드에서만 사용할 수 있으며, 파이프라인용이 아닌 셰이더나 외부 렌더링 오브젝트와 상호 작용하는 방식에 제한이 있을 수 있습니다.



Rendering Debugger 창은 에디터에서 열거나 게임 뷰, 플레이 모드 또는 빌드된 애플리케이션에서 오버레이로 열 수 있습니다.

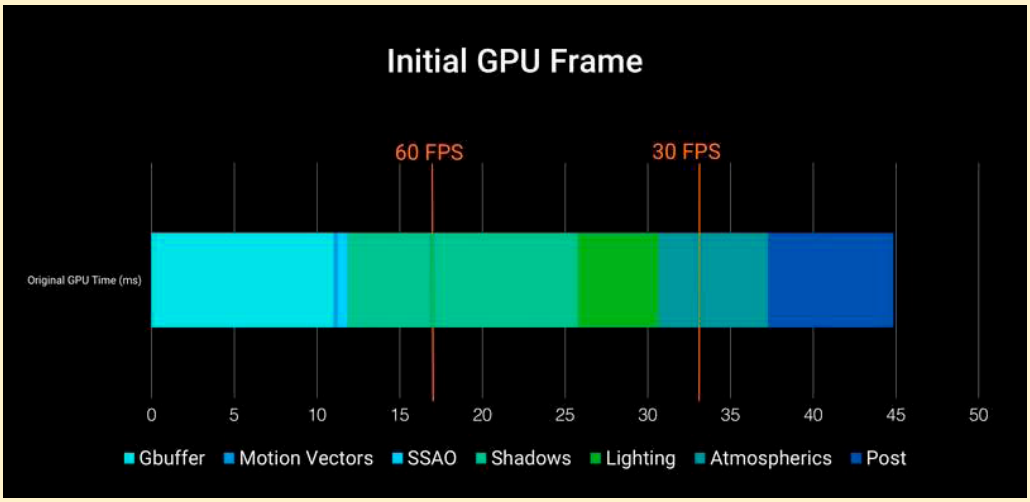
흔히 발생하는 문제를 해결하는 다섯 가지 렌더링 최적화 방법

여기에서 소개하는 유용한 팁을 사용하여 프레임 디버거와 기타 렌더 디버그 툴을 통해 파악할 수 있는 일반적인 렌더링 성능 문제를 최적화하세요.

성능 병목 지점 먼저 파악하기

우선 GPU 로드가 높은 프레임을 찾습니다. 대부분의 플랫폼은 CPU와 GPU 양측에서 프로젝트 성능을 분석할 수 있는 우수한 툴을 제공합니다. 예를 들어 Arm 하드웨어/Immortalis 및 Mali GPU용 Arm Performance Studio, Microsoft Xbox용 PIX, Sony PlayStation용 Razor, Apple iOS용 Xcode Instruments 등이 있습니다.

각 네이티브 프로파일러를 사용하여 프레임 비용을 세부 분류하면 그래픽스 성능을 개선하기 위한 기본 정보로 활용할 수 있습니다.



PS4 Pro에서 GPU 바운드로 프레임당 약 45ms가 소요되었습니다.

드로우 콜 최적화

PC와 현세대 콘솔 하드웨어에서는 많은 드로우 콜을 푸시할 수 있지만, 각 드로우 콜의 오버헤드는 여전히 높기 때문에 오버헤드를 줄이기 위해 노력해야 합니다. 모바일 기기에서는 드로우 콜 최적화가 필수적인 경우가 많습니다. 드로우 콜 배칭은 메시를 결합하여 Unity가 더 적은 드로우 콜만으로 렌더링할 수 있게 만드는 방법입니다.

위에서 설명한 프레임 디버거를 사용하면 최적의 그룹과 배치를 위해 어떤 드로우 콜을 재구성할 수 있는지를 식별할 수 있습니다. 특정 드로우 콜이 배칭되지 않는 이유를 파악할 수도 있습니다.

드로우 콜 배칭을 줄일 수 있는 기술은 다음과 같습니다.

— **오클루전 컬링**은 전경 오브젝트에 가려진 오브젝트를 제거하여 보이지 않는 요소의 오버드로우(투명한 오브젝트가 중첩되어 있어 GPU가 동일한 픽셀을 여러 번 다시 드로우하는 상황)를 줄입니다. 이 작업에는 추가로 CPU 처리가 필요하므로, 프로파일러를 사용해 작업을 GPU에서 CPU로 옮기는 것이 적절한 선택이며 새로운 병목 현상이 발생하지 않는지 확인해야 합니다.

— **GPU 인스턴싱**은 동일한 메시와 머티리얼을 공유하는 여러 개의 오브젝트를 적은 수의 배치로 렌더링하여 드로우 콜을 줄입니다. 이를 통해 낮은 성능 비용과 최소한의 시각적 반복으로 복잡한 씬을 구현할 수 있습니다.

— **SRP 배처**는 Bind 및 Draw GPU 커맨드를 배칭하여 드로우 콜 사이에 소모되는 GPU 설정을 줄입니다. SRP 배칭을 활용하려면 필요한 만큼 머티리얼을 사용하되 적은 수의 호환 가능한 셰이더 배리언트(예: URP와 HDRP의 릿 및 언릿 셰이더)로 머티리얼을 제한해야 하며, 이때 키워드 조합 간의 배리에이션이 가능한 한 적어야 합니다.

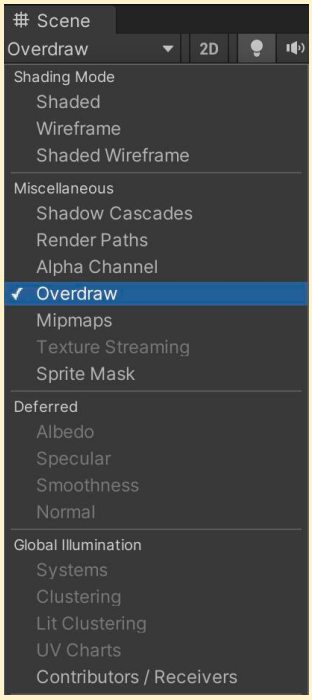
— **GPU 상주 드로어**는 GPU 인스턴싱을 사용하여 여러 개의 게임 오브젝트를 드로우하며, 이에 따라 드로우 콜의 수가 크게 줄어듭니다. 렌더링 워크로드의 많은 부분을 GPU로 이전하여 CPU의 처리 시간을 확보할 수 있기 때문에 다수의 유사한 오브젝트가 등장하는 씬에서 특히 성능 향상이 두드러집니다.

오버드로우 감소를 통한 필 레이트 최적화

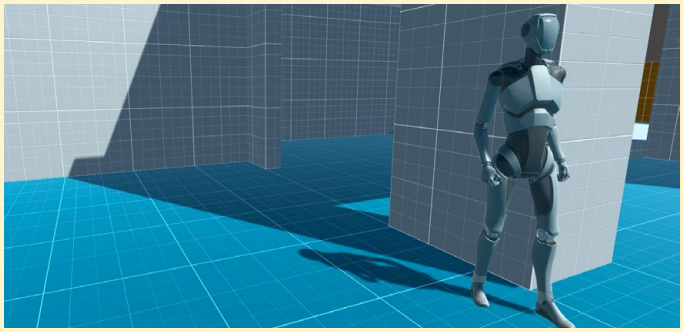
다른 오브젝트 위에 오브젝트가 렌더링되면 오버드로우가 발생합니다. 오버드로우가 발생하는 것은 애플리케이션에서 GPU가 한 프레임당 처리할 수 있는 것보다 더 많은 픽셀을 드로우하려 한다는 의미일 수 있습니다. 오버드로우는 성능을 저하할 뿐만 아니라 모바일 기기의 발열과 배터리 수명 문제도 일으킵니다. Unity가 렌더링 전에 오브젝트를 어떻게 정렬하는지 이해하면 오버드로우를 줄일 수 있습니다.

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

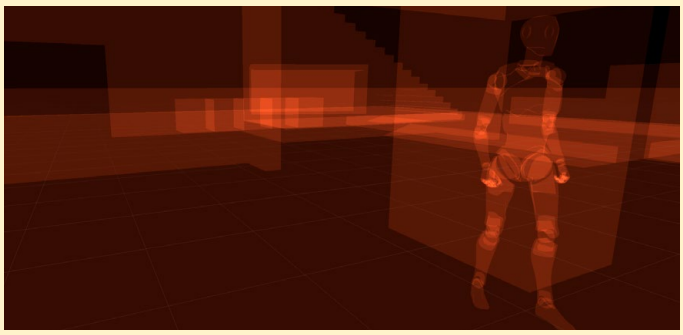
빌트인 렌더 파이프라인을 사용하는 경우 씬 뷰 컨트롤 바를 통해 오버드로우를 시각화할 수 있습니다. 드로우 모드를 Overdraw로 변경하면 됩니다.



씬 뷰 컨트롤 바에 있는
Overdraw 옵션



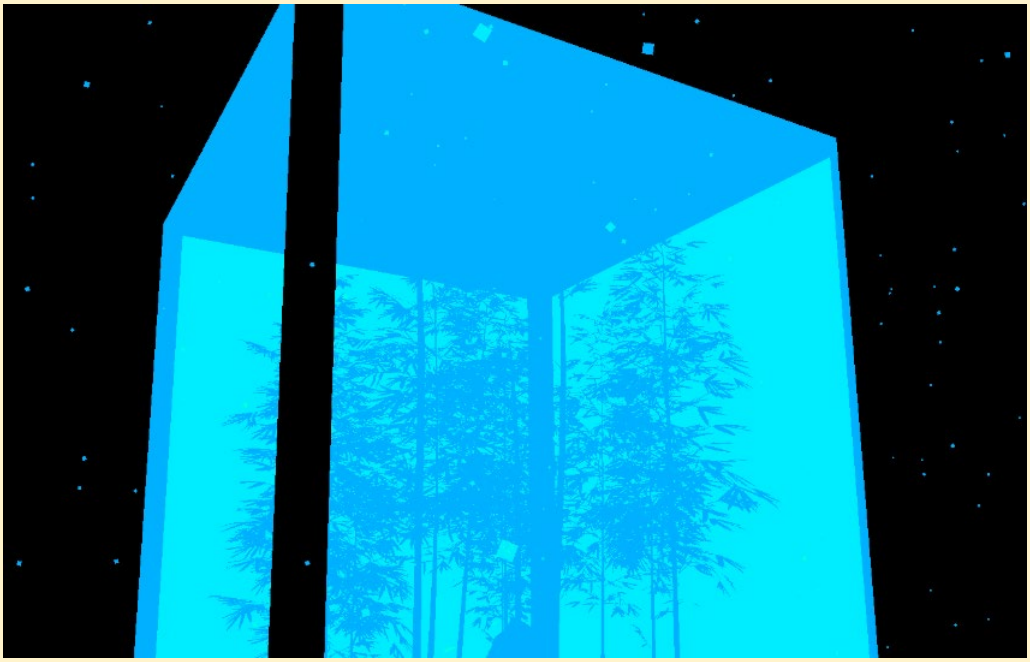
표준 Shaded 뷰로 본 씬



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

HDRP는 렌더링 대기열을 조금 다르게 제어합니다. HDRP의 접근 방식을 더 자세히 알아보고 싶다면 렌더러 및 머티리얼 우선순위 섹션을 읽어 보시기 바랍니다.

위에서 설명한 바와 같이 렌더링 디버거를 사용하여 URP 및 HDRP에서 오버드로우를 식별할 수 있습니다.



HDRP와 Fullscreen Debug Mode로 오버드로우 시각화

리소스를 가장 많이 소모하는 셰이더 검사

이 주제는 다소 복잡하지만 전반적으로 셰이더 복잡도를 최대한 줄이는 것을 목표로 합니다. 반정밀도 부동 소수점 변수를 사용하여 정밀도를 최대한 낮추면 쉽게 복잡도를 줄일 수 있습니다. 타겟 플랫폼의 웨이브프론트 점유율, 그리고 GPU 프로파일링 툴을 사용하여 정상적인 점유율을 확보하는 방법에 대해 알아보는 것도 좋습니다.

렌더링을 위한 멀티 코어 최적화

Player Settings > Other Settings에서 Graphics Jobs를 활성화하여 데스크톱과 콘솔의 멀티 코어 프로세서를 활용합니다. Graphics Jobs를 사용하면 Unity에서 여러 CPU 코어에 렌더링 작업을 분산하므로 렌더 스레드의 부담을 해소할 수 있습니다. 자세한 내용은 멀티스레드 렌더링 및 Graphics Jobs 튜토리얼을 참고하세요.

포스트 프로세싱 효과 프로파일링

포스트 프로세싱 에셋이 타겟 플랫폼에 최적화되어 있는지 확인합니다. PC 게임용으로 저작(authoring)된 Unity 에셋 스토어의 툴은 콘솔이나 모바일 기기에서 필요 이상으로 많은 리소스를 소모할 수 있습니다. 네이티브 프로파일러 툴을 사용하여 타겟 플랫폼을 프로파일링하세요. 모바일이나 콘솔 타겟을 위해 직접 포스트 프로세싱 효과를 저작할 때는 가능한 한 단순하게 만드는 것이 좋습니다.

이 외에도 프레임 디버깅과 분석에 사용할 수 있는 다양한 툴이 있습니다. 프로파일링 및 디버그 툴 일람을 살펴보고 어떤 툴이 있는지 알아보세요.

Unity 프레임 디버거에 대해 더 자세히 알아보려면 다음 자료를 참고하세요.

— Unity 프레임 디버거 기술 자료

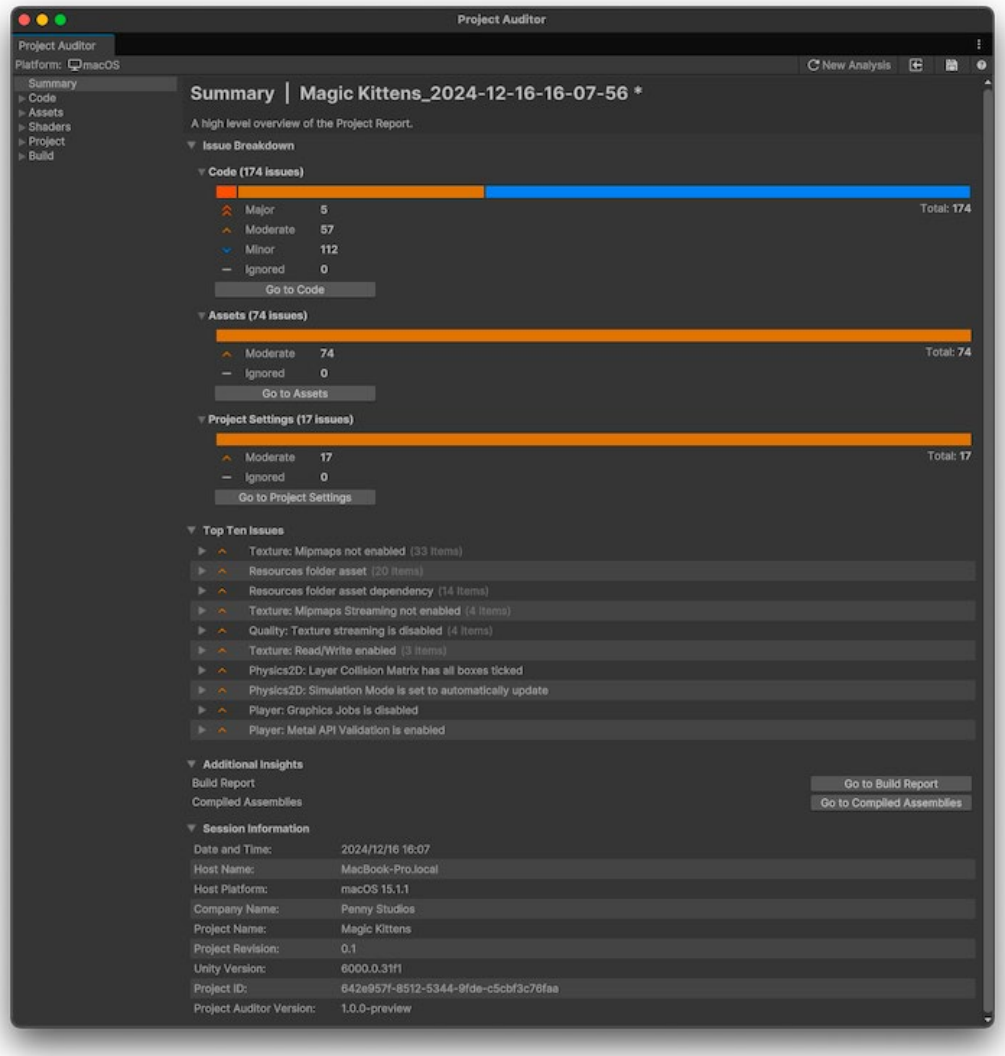
— 프레임 디버거로 작업하기

— 프로파일링 렌더링

Project Auditor

Unity 6.1에서 패키지로 도입된 Project Auditor는 Unity 프로젝트를 위한 강력한 분석 툴입니다. 개발자는 이 툴을 사용해 성능을 최적화하고, 베스트 프랙티스를 적용하고, 프로젝트에서 잠재적인 문제와 병목 현상이 있는지 검토할 수 있습니다.

Project Auditor는 프로젝트 전체를 스캔한 다음, 지나친 스크립팅 호출과 사용되지 않는 에셋, 지나치게 많은 항목 수와 같은 비효율적인 부분에 대한 상세한 리포트를 제공합니다.



Project Auditor Summary 뷰

Project Auditor는 다음과 같은 영역에 유용합니다.

— **성능 최적화:** 과도한 가비지 생성, 불필요한 오브젝트 할당, 리소스가 많이 소모되는 함수 호출 등 프로젝트의 런타임 성능에 영향을 줄 수 있는 문제를 식별합니다.

— **코드 및 에셋 검토**: 사용되지 않는 에셋, 비효율적인 코드 패턴 또는 리팩터링이 가능한 오래된 API를 확인합니다. 이러한 작업은 빌드 크기를 줄이고, 프로젝트의 전반적인 유지 관리 효율을 높이고, 메모리 사용량을 최적화하는 데 도움이 됩니다.

— **진단 및 베스트 프랙티스**: Unity 베스트 프랙티스에 기반한 권장 사항을 제공하며 참조 누락, 최적화되지 않은 Player 또는 Quality 설정과 같은 프로젝트 설정 관련 오류와 경고를 보여 줍니다.

— **리포트 커스터마이징**: 결과를 범주화하여 최적화 우선순위를 정하는 데 도움을 줍니다. 특정 프로젝트나 수요에 따른 분석을 위해 커스텀 규칙을 만들 수도 있습니다.

💡**팁**:

— 주요 개발 단계(예: 마일스톤, 베타 릴리스, 최종 빌드 전에)에 Project Auditor를 실행하세요. 정기적으로 감사를 시행하면 성능 병목 현상, 사용되지 않는 에셋, 오래된 코드를 초기에 발견하여 프로젝트의 규모가 커짐에 따라 문제가 커지는 일을 방지할 수 있습니다.

— Project Auditor가 CI 또는 빌드 설정의 일환으로 실행되도록 자동화하고(매뉴얼의 이 부분 참고), 리포트를 사용하여 새로운 문제를 유발하는 에셋이나 코드를 체크인하는 경우를 방지할 수 있습니다 (여기에 나와 있는 API 사용).

— 게임에서 발견하려는 특정 항목이 있는 경우(예: 텍스처 설정, 크기, 복잡한 규칙 등) 커스텀 규칙을 추가할 수 있습니다. 방법은 매뉴얼의 이 페이지를 참고하세요.

Project Auditor에서 생성되는 리포트는 심각도(Major, Moderate, Info)를 기준으로 분류됩니다. 가장 심각한 문제에 우선 집중하세요. 메모리 과잉 할당, 과도한 가비지 컬렉션 등 성능에 큰 영향을 주는 문제인 경우가 많습니다. 이러한 문제는 빈번하게 호출되는 코드 경로에 있는 경우도 많습니다. 예를 들어 Update에 문제가 있어 성능이 저하되면 플레이어들은 즉시 알아차릴 수 있습니다.

Project Auditor는 **Player 설정**, **Quality 설정**과 같은 설정을 확인하고 무엇을 변경하면 좋을지에 관한 권장 사항을 제공합니다. 이 기능을 사용하여 빌드 타겟, 해상도, 텍스트 압축 및 그 밖의 프로젝트 설정을 타겟 플랫폼에 최적화하세요.

도메인 리로드

Unity 에디터를 사용하면 플레이 모드 진입에 관한 설정을 구성할 수 있습니다. 이 페이지에 자세한 설명이 나와 있는데, 도메인 리로드를 비활성화하면 에디터로 반복 작업하는 시간이 단축되는 경우가 많습니다. 단, 이렇게 하면 플레이 모드에 진입할 때마다 스크립팅 상태가 재설정되지 않으므로 코드에서 수동으로 재설정해야 합니다.

Project Auditor의 Code 영역에서 프로젝트의 스크립트 분석 결과를 보고 스크립트 변수를 재설정해야 하는 곳을 확인할 수 있습니다. Domain Reload 뷰에 표시된 모든 문제를 수정한 후 도메인 리로드를 비활성화하는 것이 좋습니다. 이 뷰에 데이터를 표시하려면 Preferences 창에서 Use Roslyn Analyzers 설정을 활성화해야 합니다. 그런 다음 문제를 하나씩 검토하면서 매뉴얼의 지침에 따라 수정합니다. 모든 문제를 해결했으면 플레이 모드 진입 시 도메인 리로드를 비활성화할 수 있습니다.

딥 프로파일링

프로파일링 기초 섹션에서 언급했듯이 기본적으로 Unity는 프로파일러 마커로 명시적으로 래핑된 코드만 프로파일링합니다. 여기에는 엔진의 네이티브 코드가 호출하는 관리되는 코드의 첫 번째 호출 스택 뎁스가 포함됩니다.

딥 프로파일링을 활성화하면 각 함수 호출의 시작과 끝에 프로파일러 마커가 삽입됩니다. 이를 통해 많은 세부 정보를 캡처할 수 있습니다. 딥 프로파일 설정을 사용하여 호출 스택이 충분히 표시되지 않는 긴 프로파일러 마커 내에서 어떤 일이 일어나고 있는지 파악할 수 있습니다.

게임 성능 측정에 대한 세분화된 접근 방식을 활용하면 캡처 시에 세부 정보가 누락될 수 있는 스냅샷 기반 프로파일링(샘플 프로파일링)보다 더 유용한 결과를 얻을 수 있습니다.

문제가 있는 코드 영역을 수동으로 계측하려면 ProfileMarker API를 확인해 보세요.

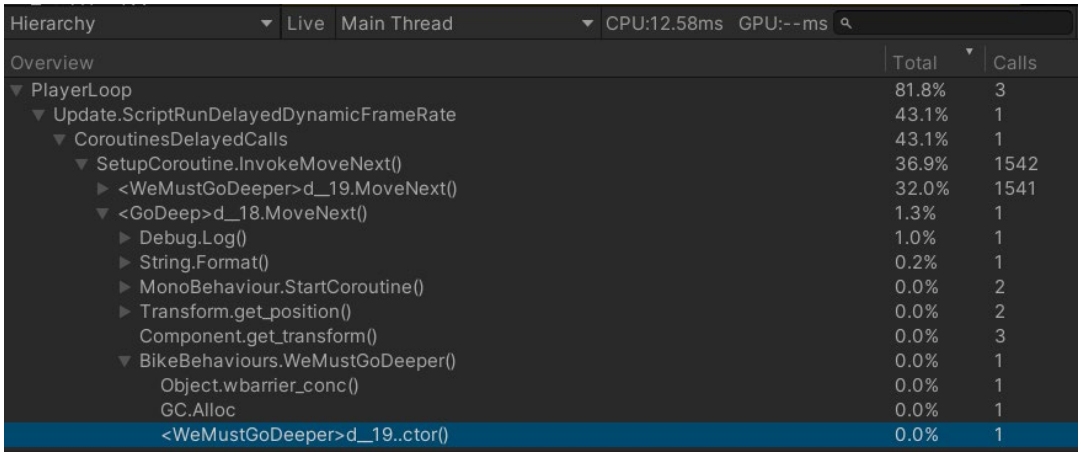
이 방법은 딥 프로파일링보다 성능에 주는 영향이 훨씬 적을 수 있습니다. 때로는 딥 프로파일링을 활성화한 상태에서 테스트하려는 게임 부분을 찾는 것보다 ProfileMarker를 추가하고 게임을 다시 빌드하는 것이 훨씬 더 빠를 수도 있습니다.

기기 빌드에서 전체 호출 스택을 확보하는 또 다른 대안은 네이티브 CPU Usage Profiler를 실행하는 것입니다. 때로는 이 방법이 딥 프로파일링보다 더 쉽고 성능에 미치는 영향도 더 적습니다.

딥 프로파일링을 사용해야 하는 경우

딥 프로파일 설정은 애플리케이션이나 관리되는 코드에서 더 자세히 조사해야 하는 부분을 파악한 경우에만 활성화해야 합니다. 딥 프로파일링은 리소스 소모가 심하고 많은 메모리를 사용하며, 활성화 시 애플리케이션 실행 속도가 느려집니다.

딥 프로파일링을 사용하면 호출 트리를 심층적으로 살펴보고 비효율적인 코드나 관련 문제를 발견할 수 있습니다.



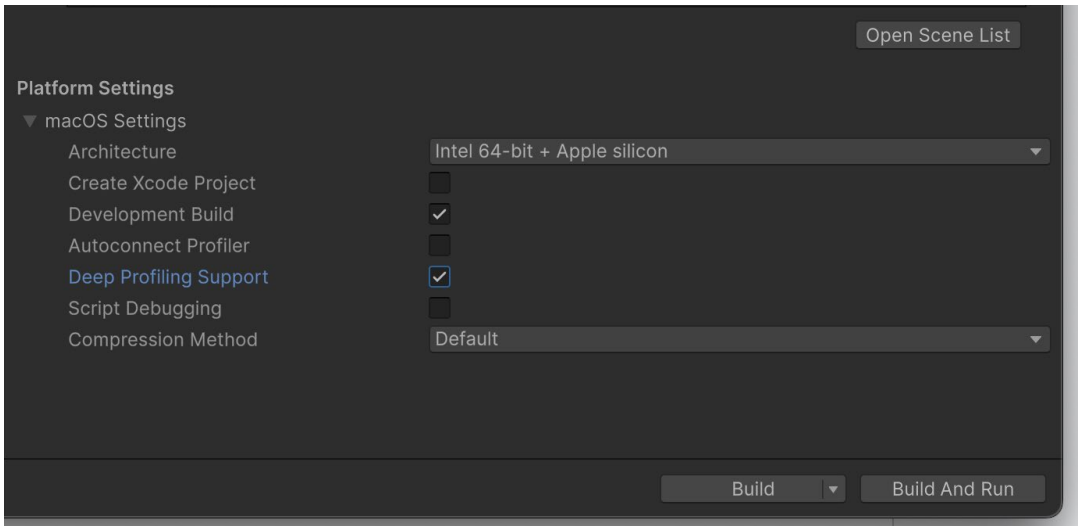
특정 문제를 추적하고 이해해야 할 때 딥 프로파일링을 사용하면 방대한 양의 세부 정보를 확보할 수 있습니다.

하지만 딥 프로파일링은 모든 함수 호출의 시작 지점과 종료 지점에 마커를 추가하며, 각 마커는 오버헤드를 유발합니다. 따라서 코드에서 딥 콜스택이 있는 부분(예: MyDeepFunction)은 하나의 함수 안에서 모든 작업을 처리하는 부분(예: MySingleFunction)보다 많은 리소스가 소모되는 것으로 표시됩니다. 따라서 양측 코드의 상대적인 타이밍을 그대로 받아들이지 않도록 하세요. 딥 프로파일링이 활성화된 MyDeepFunction은 MySingleFunction보다 많은 리소스가 소모되는 것처럼 보일 수 있으나, 이러한 비용은 모두 추가된 마커에서 발생했을 수도 있기 때문입니다.

참고: Unity 2019.3 이상 버전부터 Mono 및 IL2CPP 백엔드에서 딥 프로파일링이 지원됩니다. iOS처럼 IL2CPP가 필수인 플랫폼에서 개발하는 개발자에게는 반가운 소식입니다.

딥 프로파일링 사용

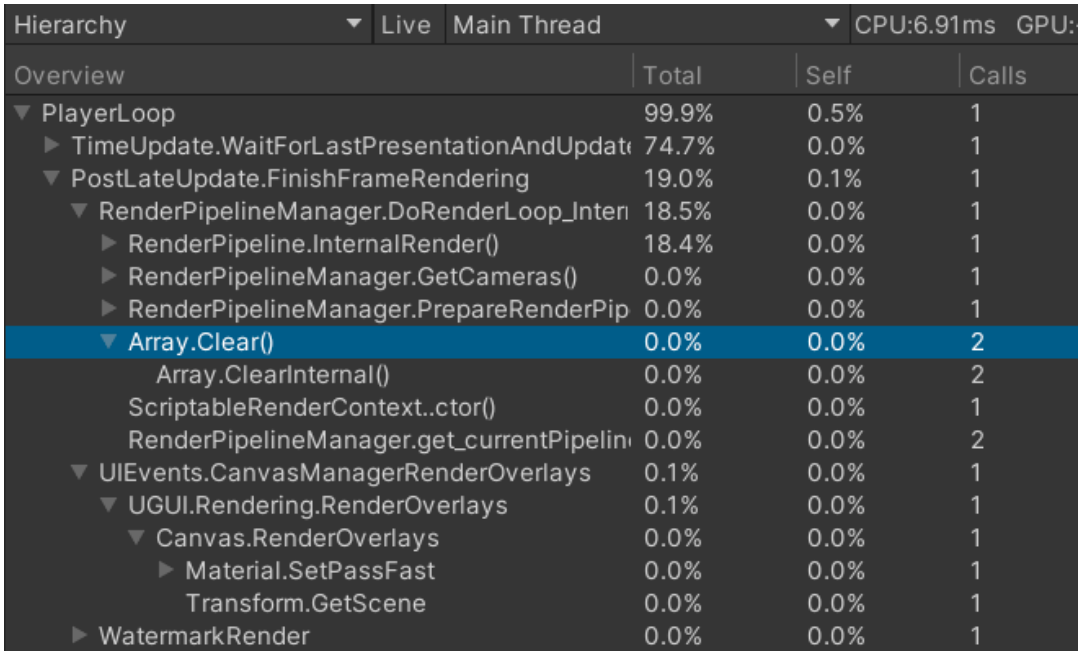
플레이어 빌드에서 딥 프로파일링을 사용하려면 **File > Build Settings > Enable Deep Profiling Support**에서 활성화 과정을 거쳐야 합니다.



Deep Profiling Support 활성화

딥 프로파일링 지원을 활성화하면 Profiler 창에서 언제든지 빌드의 딥 프로파일링을 쉽게 토글할 수 있습니다.

플레이어에 연결했을 때 Deep Profile 버튼이 희미하게 표시되면 빌드에 Deep Profiling Support가 활성화되지 않았다는 의미입니다.



딥 프로파일링을 통해 애플리케이션 코드의 성능 및 타이밍과 관련하여 더 많은 정보를 확인할 수 있습니다. 전체 메서드 호출 트리가 표시되므로 관리되는 할당이 어디에서 발생하는지 심층적으로 살펴볼 수 있습니다.

딥 프로파일링 팁

하향식 접근 방식

애플리케이션을 프로파일링할 때는 먼저 상위 수준에서 시작하여 딥 프로파일링을 사용하지 않고도 성능을 개선할 수 있는 영역이 있는지 찾아보는 것이 좋습니다. 더 많은 정보가 필요하면 프로파일러에서 딥 프로파일링을 활성화하여 더 세부적인 수준에서 분석할 수 있습니다. 이러한 접근 방식을 사용하면 프로파일러 계층 구조에 표시되는 정보 수준을 최소한으로 유지하여 현재 목표에 집중할 수 있습니다.

필요한 경우에만 딥 프로파일링하기

일반적으로는 코드 성능에 대해 하위 수준의 상세한 정보가 필요할 때 딥 프로파일링을 사용하는 것이 가장 좋습니다. 빌드에서 딥 프로파일링 플래그를 활성화한 상태로 두어도 실제로 이 기능을 토글하여 활성화하지 않는 한 성능에 영향을 미치지 않지만, 일단 활성화하면 애플리케이션 실행이 느려질 수 있습니다.

코드에서 관리되는 할당의 출처를 찾기만 하면 되는 경우, Unity 2019.3 이상에서는 딥 프로파일링을 활성화하지 않고도 해당 작업을 수행할 수 있습니다. 프로파일러에서 Call Stacks 토글과 Calls 드롭다운을 사용하면 관리되는 할당을 찾을 수 있습니다.

자동화된 프로세스로 딥 프로파일링 사용

빌드 실행 파일에 **-deepprofiling** 인자를 추가하면 커맨드 라인에서 프로파일링할 때 딥 프로파일링을 토글할 수 있습니다. Android/Mono 스크립팅 백엔드 빌드의 경우 다음과 같은 adb 커맨드 라인 인자를 사용합니다.

adb shell am start -n com.company.game/com.unity3d.player.UnityPlayerActivity -e ‘unity’ ‘-deepprofiling’

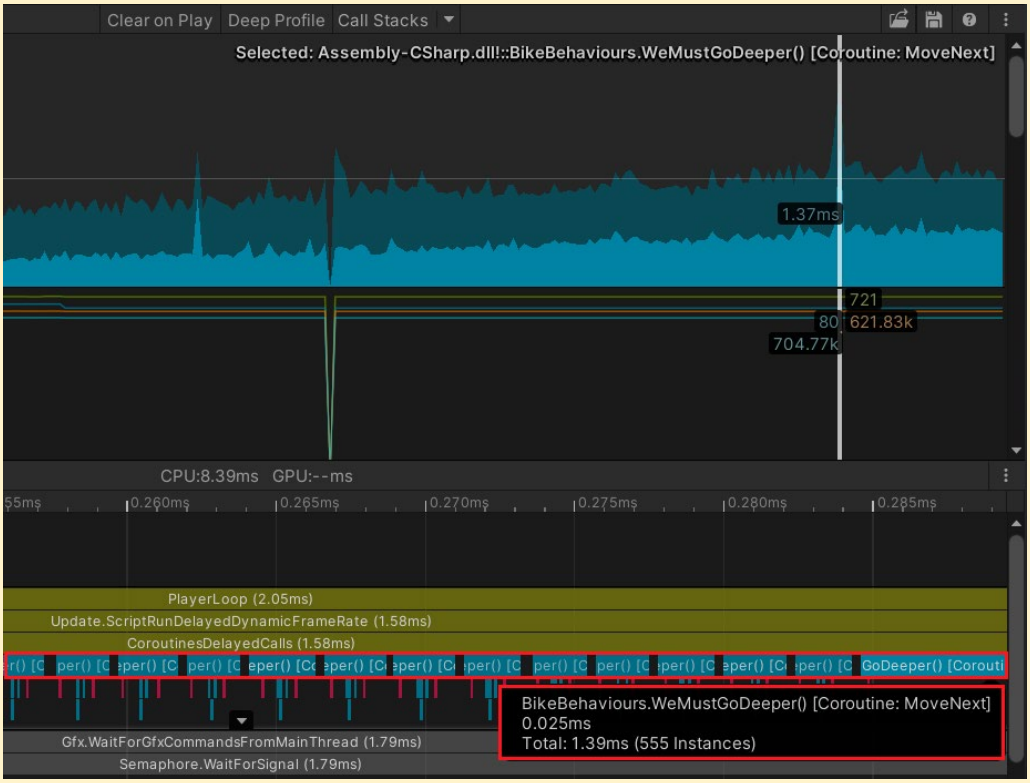
저사양 하드웨어에서의 딥 프로파일링

저사양 하드웨어는 메모리와 성능이 제한적이기 때문에 딥 프로파일링을 사용할 때 문제가 생길 수 있습니다. Unity 프로파일러 샘플은 링 버퍼에 저장되는데, 속도가 느린 기기에서 딥 프로파일 설정을 사용하면 링 버퍼가 가득 찰 수 있습니다. 그러면 Unity에 오류 메시지가 표시됩니다.

Profiler.maxUsedMemory 프로퍼티(바이트)를 설정하면 버퍼링 데이터를 위한 추가 메모리를 프로파일러에 할당할 수 있습니다. 기본적으로 플레이어는 128MB, 에디터는 512MB입니다.
딥 프로파일링을 실행할 때 문제가 발생하면 저사양 기기의 플레이어 빌드에서 필요에 따라 이 값을 늘리면 됩니다.

딥 프로파일링으로 인한 오버헤드 때문에 너무 느리거나 아예 실행되지 않는 하드웨어에서 코드를 세부적으로 프로파일링하려는 경우, 자체 마커를 사용하여 해당 작업을 수행할 수 있습니다.

딥 프로파일 설정을 활성화하는 대신 코드에서 확인하려는 구체적인 영역에 프로파일러 마커를 추가해 보세요. 해당 마커는 CPU Usage 모듈에서 프로파일러 타임라인이나 계층 구조에 표시됩니다.



팁 프로파일링 때문에 오버헤드가 너무 많이 발생하는 경우 프로파일러 마커를 추가하여 코드 레이어를 더 세부적으로 프로파일링합니다.

사용할 프로파일링 툴과 시기 결정

프로파일링은 프로젝트 라이프사이클 초기에 시작할 때 가장 효과적입니다. 초기에 프로파일링을 시작하면 게임 및 애플리케이션 개발의 중후반부 체크포인트에 활용 가능한 비교 기준선을 설정할 수 있습니다. 여러 프로파일링 툴 중에서 어떤 툴을 선택하여 언제 사용할지 파악하는 것이 중요합니다.

각 툴의 용도와 이점을 이해하면 툴의 사용 시기를 더 쉽게 알 수 있습니다. Unity 프로파일링 및 디버그 툴 섹션에서 유니티가 제공하는 각 프로파일링 툴에 대해 자세히 알아보세요.

프로파일링 툴의 사용 시기를 정할 때는 프로젝트 라이프사이클의 체크포인트 목록을 참조하면 프로파일링 전략을 세울 때 도움이 될 것입니다.

— **프로토타이핑**: 프로파일링은 프로젝트의 프로토타입 단계에서 위험을 줄이는 데 중요한 역할을 합니다. 게임 디자인 문서에서 화면에 10,000명의 적을 생성해야 한다고 정했다면, 타겟 플랫폼에서 이를 구현할 수 있음을 증명하는 프로토타입을 빌드하고 프로파일링할 수 있어야 합니다. 이 프로세스가 불가능하다면 게임 디자인을 변경해야 합니다.

— **프로젝트 초기 단계**: 다양한 타겟 기기 하드웨어에서 프로젝트 성능에 대한 기준선을 정하는 단계입니다. Memory Profiler로 대략적인 메모리 사용량을 파악하여 추후에 타겟 하드웨어의 메모리 제한으로 문제가 발생하지 않도록 프로젝트 범위를 계획해야 합니다.

— **스프린트 종료 시점**: 애자일 팀에서 스프린트로 작업하는 경우, 스프린트 종료 시점의 RC(릴리스 후보)에 표준화된 프로파일링 툴 세트를 실행하는 것이 좋습니다. 데이터베이스나 스프레드시트 등 결과와 지표를 기록할 수 있는 표준 포맷이 있어야 합니다. Unity 프로파일러로 다음 부문을 프로파일링하고 데이터를 캡처합니다.

  — CPU 사용량

  — GPU 사용량

  — 메모리 사용량

  — Rendering

  — 물리

더 깊이 있는 분석을 위해 다음과 같은 툴을 사용하여 차이가 나타나는 주요 지표(이전 스프린트 릴리스와의 차이)와 결과를 기록합니다.

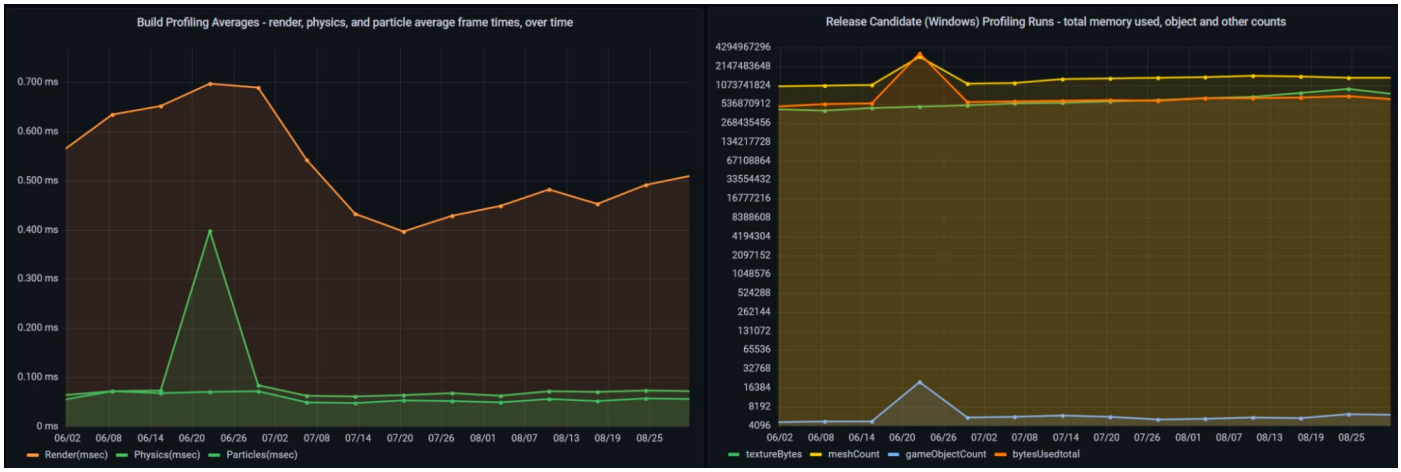
— **Profile Analyzer**: 이전 릴리스의 프로파일링 데이터 캡처를 로드하고 차이점을 비교 및 기록합니다.

— **Memory Profiler**: 이전 RC 빌드의 메모리 스냅샷을 비교하고, 메모리 사용량의 증감 차이를 기록합니다.

주요 성능 및 프로파일링 지표 자동화

공통적으로 발생하고 반복되는 작업을 자동화하면 프로젝트 프로파일링과 데이터 캡처를 개선할 수 있습니다. 자동화를 통해 시간을 절약하면서 항상 최신 지표를 활용할 수 있습니다.

지표를 그래프로 표현해 프로젝트 대시보드에 추가하면 새로 추가된 기능이나 버그 등으로 인해 성능이 급격히 저하된 부분이나 최적화 및 버그 수정 스프린트 후에 개선된 부분을 확인할 수 있습니다.

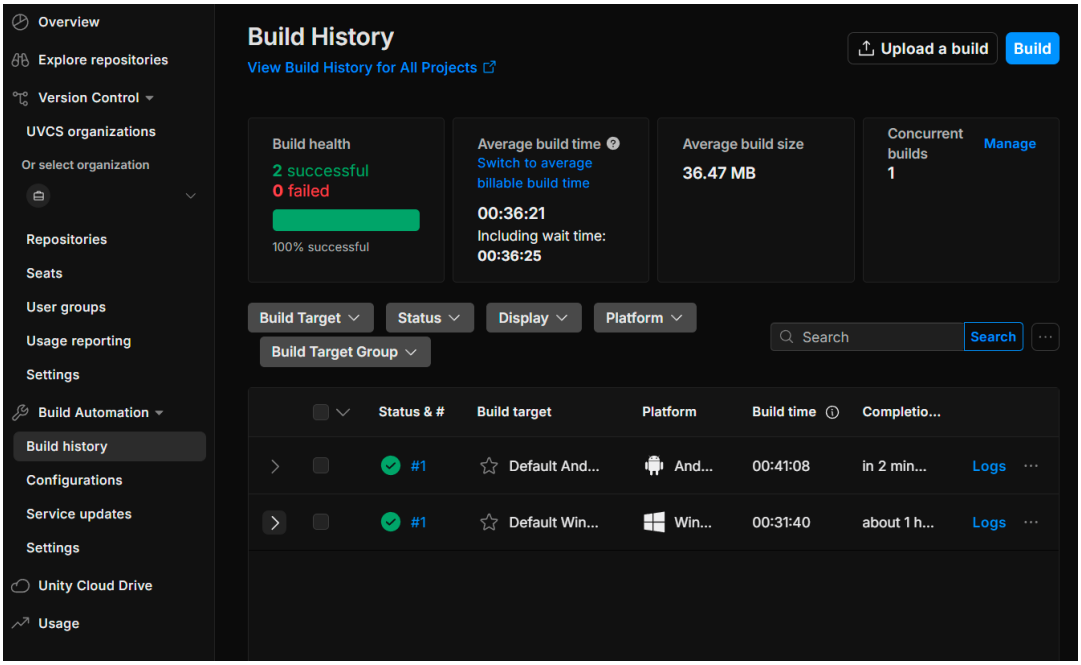


자동화된 주간 빌드 프로파일링 데이터가 캡처되고 시각화된 Grafana 대시보드

개발을 진행하면서 게임의 모든 레벨에서 프로젝트의 전반적인 메모리 사용량 프로파일을 도식화하는 것이 좋습니다. Memory Profiler로 메모리 스냅샷을 캡처하고 모든 레벨에서 수치를 평균화하면 타겟 기기/플랫폼, 스프린트, 릴리스 주기별로 메모리 사용량을 기록할 수 있습니다.

개략적인 프로파일링 통계를 기록하고 싶다면 ProfilerRecorder로 Total Reserved Memory 또는 System Used Memory 등의 지표를 기록하고 CI(지속적 통합)에 출력하여 차트로 만들거나 Grafana 같은 그래프 생성 툴로 내보낼 수 있습니다.

Unity DevOps 툴로 릴리스 빌드 생성을 자동화하고 해당 프로세스를 자동화된 기기 프로파일링 워크플로에 통합해 보세요.



Unity Cloud 대시보드의 **Build Automation > Build History** 기능을 사용하여 빌드의 결과를 확인합니다. 실패한 빌드가 있다면 로그를 확인하여 문제를 해결할 수 있습니다. 프로젝트 구성 및 버전 관리 베스트 프랙티스(Unity 6 에디션) 전자책에서 프로젝트 구성 및 Unity 서비스에 대해 자세히 알아보세요.

자동화된 프로파일링 파이프라인 예시

프로파일링을 자동화하면 시간 제약으로 인해 프로파일링의 우선순위가 낮아질 걱정 없이 빌드 프로파일링의 이점을 적극적으로 활용할 수 있습니다.

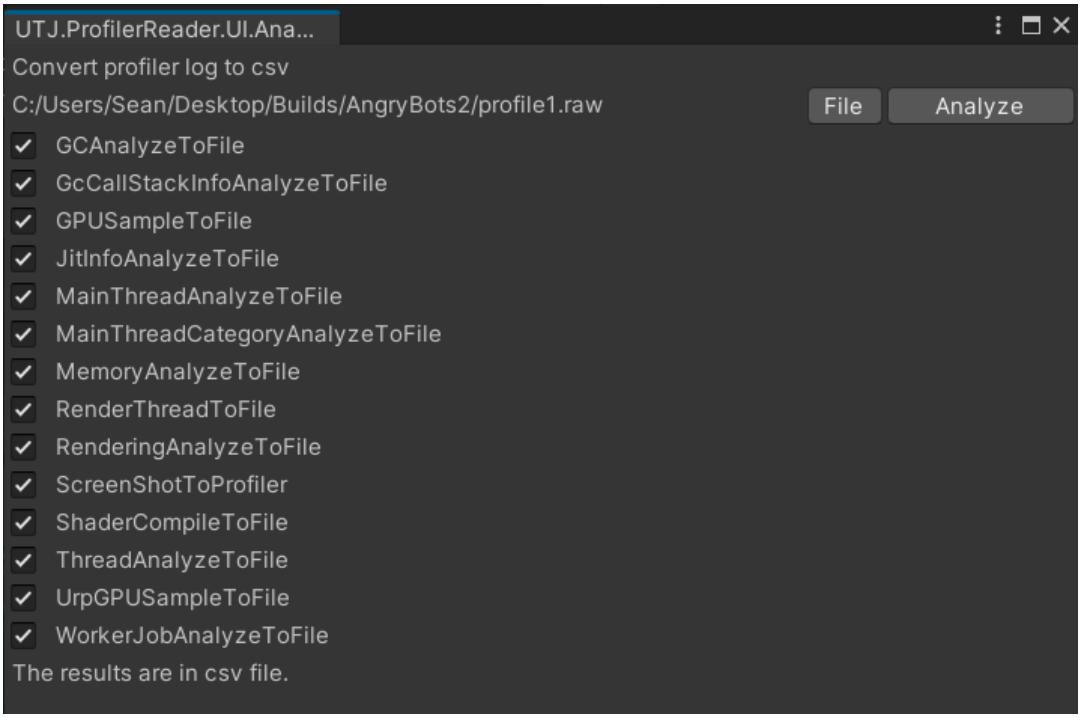
이 예시 워크플로에서는 자동화를 사용하여 높은 빈도로 정확하게 빌드 프로파일링을 수행하는 방법을 소개합니다.

- Unity Build Automation을 사용하여 자동화된 빌드 릴리스를 생성합니다.
- 빌드를 릴리스할 때마다 스크립트로 빌드된 플레이어를 시작하고 2,000프레임 이상의 원시 프로파일링 데이터를 캡처합니다. 예:
  - **AngryBots2.exe -profiler-enable -profiler-log-file profile1.raw -profiler-capture-frame-count 2000.** 이 커맨드 라인 인자에 대한 자세한 내용은 Unity 기술 자료에서 확인할 수 있습니다.

참고: 스크립트를 사용하여 300~2,000프레임마다 새 로그 파일(예: profile_<N+1>.raw)로 전환하거나 애플리케이션 테스트 주기의 주요 지점(자동화된 레벨 플레이의 체크포인트)을 프로파일링하는 것도 하나의 방법이 될 수 있습니다. 나중에 대시보드 그래프에서 문제가 있는 부분을 발견하면 이렇게 저장된 데이터를 참조할 수 있습니다.

— 프로파일링 데이터는 **profile1.raw** 파일에 캡처되고 파싱하여 원하는 지표를 확인할 수 있습니다.

— 다음 단계에서는 원시 프로파일 데이터를 파싱하고 CSV 포맷으로 전환하는 툴인 ProfileReader로 원시 프로파일 데이터를 읽기 쉬운 CSV 포맷으로 전환합니다.

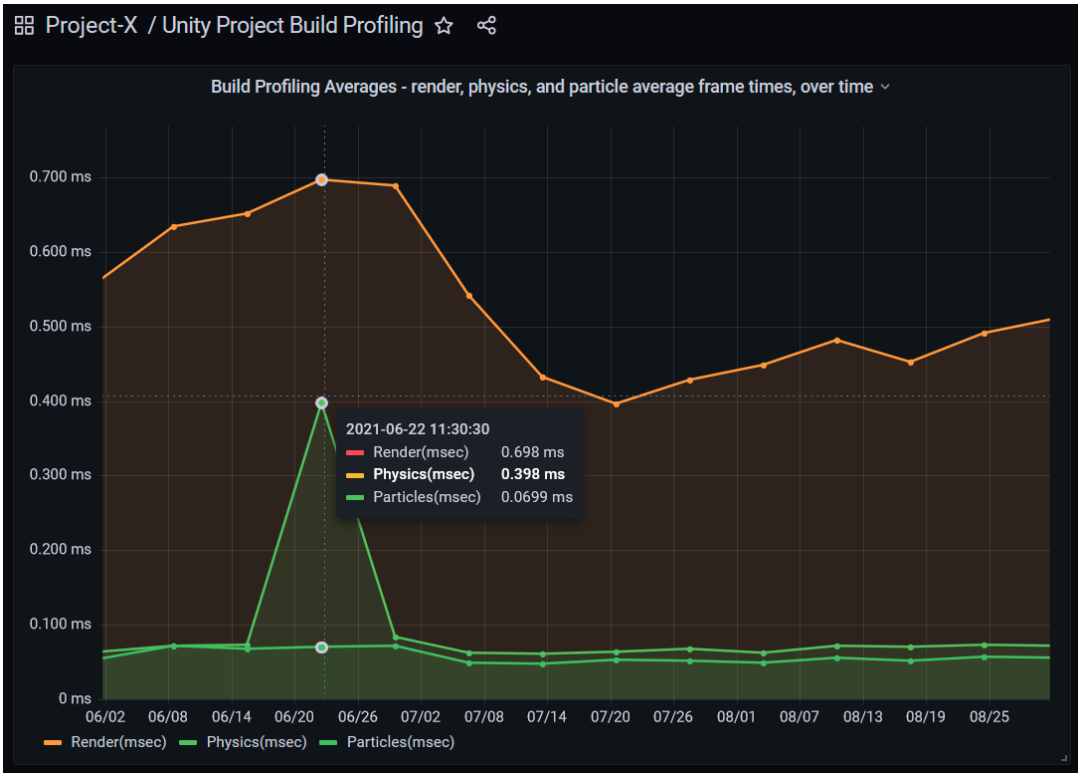


ProfileReader용 에디터 인터페이스

— ProfileReader는 커맨드 라인에서 사용할 수 있으므로 파이프라인의 이 단계가 실행 스크립트가 됩니다.

  — Unity.exe -batchMode -projectPath “AngryBots2” -logFile .\Editor.log -executeMethod UTJ.ProfilerReader.CUIInterface.ProfilerToCsv -PH.inputFile “profile1.raw” -PH.timeout 2400 -PH.log

— 자동화된 파이프라인은 CSV에서 파싱된 데이터와 함께 일간, 주간, 스프린트 릴리스 데이터를 Grafana 같은 툴에 업로드하여 시각화합니다.



Grafana 대시보드에 자동화된 프로파일링 데이터가 시각화된 모습입니다. 빌드에 물리 오브젝트 생성 버그가 발생한 것처럼 보입니다.

데이터를 자동으로 시각화 및 업데이트하면 그래프에서 스파이크를 쉽게 확인하여 문제를 더 빠르게 파악할 수 있습니다. 성능 최적화 작업의 결과나 레벨 디자인 팀이 게임의 여러 레벨에서 메모리 최적화 패스를 수행한 결과도 확인할 수 있습니다.

Unity Test Framework를 위한 Performance Testing 패키지

Unity Performance Testing 패키지는 성능 테스트용 툴을 추가하여 Unity Test Framework를 개선합니다. 이 패키지는 에디터와 빌트인 플레이어에서 Unity 프로파일러 마커와 커스텀 성능 지표의 샘플을 캡처할 수 있도록 지원하는 API와 테스트 데코레이터를 제공합니다.

측정 기능을 제공하는 것 외에도 이 패키지는 빌드 설정, 하드웨어 세부 정보와 같은 구성 메타데이터를 수집하여 여러 환경의 결과를 쉽게 비교할 수 있도록 지원합니다.

이 패키지는 Unity Test Framework와 함께 작동하도록 설계되었습니다. 효과적으로 사용하려면 Unity Test Framework 기술 자료에 설명된 테스트 만들기 및 실행 방법을 숙지해야 합니다.

프로파일링 및 디버깅 툴 일람

Unity 툴로 프로파일링을 시작한 다음 더 세부적인 데이터가 필요한 경우 타겟 플랫폼에서 사용할 수 있는 네이티브 프로파일러와 디버깅 툴을 활용하세요. 아래에서 프로파일링 및 디버깅 툴 일람을 확인할 수 있습니다.

네이티브 프로파일링 툴

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 Performance Studio: Arm 하드웨어를 활용하는 기기에 맞게 게임을 매우 세세하게 프로파일링 및 디버깅할 수 있는 툴 세트입니다.

— Snapdragon Profiler: Snapdragon 칩셋을 탑재한 기기 전용 툴입니다. CPU, GPU, DSP, 메모리, 전력, 발열, 네트워크 데이터를 분석하여 성능 병목 현상을 찾고 해결합니다.

Intel

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

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

Xbox/PC

— PIX: PIX는 DirectX 12를 사용하는 Windows 및 Xbox 게임 개발자를 위한 성능 조정 및 디버깅 툴입니다. CPU와 GPU 성능을 파악하고 분석하는 툴뿐만 아니라 다양한 실시간 성능 카운터를 모니터링하는 툴도 함께 제공합니다.

PC/범용

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

— NVIDIA NSight: 개발자가 NVIDIA의 최신 비주얼 컴퓨팅 하드웨어를 사용하여 동급 최고 수준의 최첨단 소프트웨어를 빌드, 디버그, 프로파일, 개발하도록 지원하는 툴입니다.

— Samply: Samply는 Firefox 프로파일러를 UI로 사용하는 오픈 소스 커맨드 라인 CPU 프로파일러입니다. macOS, Linux, Windows에서 작동합니다.

— Superluminal: Superluminal은 C++, Rust, .NET으로 작성된 Windows, Xbox One, PlayStation 애플리케이션의 프로파일링을 지원하는 고성능 고빈도 프로파일러입니다. 이 제품은 유료로, 라이선스가 있어야 사용할 수 있습니다. 시작하는 방법을 빠르게 살펴보려면 Discussions 문서를 확인하세요.

Playstation

— PlayStation 하드웨어용 CPU 프로파일러 툴을 사용할 수 있습니다. PlayStation® 개발자로 등록되어 있는 경우 여기에서 자세한 내용을 확인할 수 있습니다.

iOS

— Xcode Instruments 및 XCode Frame Debugger: Instruments는 Xcode 툴 세트에 포함된 강력하고 유연한 성능 분석 및 테스트 툴입니다.

WebGL

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

— Chrome DevTools Performance: Unity WebGL 빌드를 프로파일링할 수 있는 또 다른 웹 브라우저 툴입니다.

GPU 디버깅 및 프로파일링 툴

Unity 프레임 디버그 툴은 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. 아래에서 원하는 플랫폼과 관련된 툴을 살펴보세요.

— Arm Streamline: Arm Performance Studio 소프트웨어 제품군의 일부로, CPU와 GPU의 낮은 오버헤드 성능 측정에 중점을 둡니다.

— Arm Frame Advisor: Arm Performance Studio 소프트웨어 제품군의 일부로, 프레임 기반 API 프로파일링에 중점을 둡니다.

— RenderDoc: 데스크톱 및 모바일 플랫폼용 GPU 디버거로, 프레임 기반 API 디버깅에 중점을 둡니다.

— Intel GPA: Intel 기반 플랫폼용 그래픽스 프로파일링 툴

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

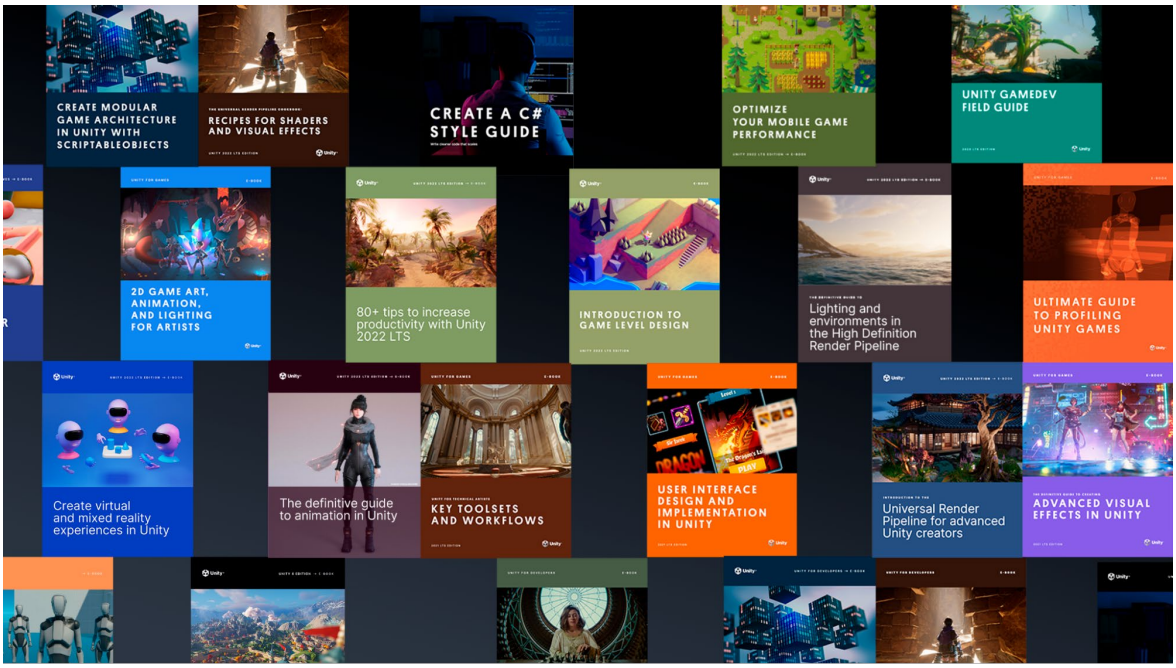
— Visual Studio Graphics Diagnostics: Windows, Xbox 등의 DirectX 기반 플랫폼은 이 툴 또는 PIX 사용

— NVIDIA Nsight Frame Debugger: NVIDIA GPU용 OpenGL 기반 프레임 디버거

— AMD Radeon Developer 툴 제품군: AMD GPU용 GPU 프로파일러

— Xcode 프레임 디버거: iOS 및 macOS용

숙련된 개발자 및 아티스트를 위한 리소스



Unity 베스트 프랙티스 허브에서 숙련된 Unity 개발자 및 크리에이터를 위한 더 많은 전자책을 다운로드하실 수 있습니다. 업계 전문가, 유니티 엔지니어 및 테크니컬 아티스트가 제작한 30개 이상의 가이드에서 Unity의 툴셋과 시스템을 사용하여 효율적으로 게임을 개발하기 위한 베스트 프랙티스를 살펴보세요.

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

unity.com/kr