9주차 정리

개요

▸최적화와 성능 관리 학습

▸프레임레이트 관리와 리소스 최적화 실습

▸게임 성능 향상을 위한 방법과 도구 학습

▼ 최적화와 성능 관리 학습

CPU 기반과 GPU 기반의 비교

세션 중 말씀드린 것처럼 프로젝트를 최적화하려면 먼저 실제 병목 현상을 찾아야 합니다. 한 가지 방법은 Unity 프로파일러를 통해 CPU 사용 내역을 확인하는 것입니다. 아래 이미지에서 보는 바와 같이 대부분의 프레임 시간이 렌더링에 소요되었다면 CPU와 GPU 중 어느 것을 주로 사용하는지 파악해야 합니다.

렌더링은 CPU와 GPU를 모두 사용하는 프로세스입니다. 본문에서 이 프로세스의 전반을 설명할 수는 없지만, 요약하자면 씬의 렌더링은 다음 단계로 구성됩니다.

1. 머티리얼을 공유하는 오브젝트의 그룹별로 다음 과정이 수행됩니다.
    a. CPU가 GPU로 커맨드를 전송하여 GPU의 내부 상태(예: 셰이더, 기반 텍스처, 버텍스 포맷 등)를 설정합니다. 이 단계를 'set pass' 호출이라고도 합니다.
    b. CPU는 GPU로 지오메트리 배치를 전송하고 1.A 단계에서 설정한 상태를 사용하여 렌더링합니다. 이 단계를 '드로우 콜'이라고도 하며, 여기에는 상당히 많은 비용이 소요됩니다.
    c. 동일한 머티리얼 유형의 지오메트리를 더 렌더링해야 하는 경우 1.B 단계로 진행합니다.

위의 알고리즘에는 보다 세부적인 내용과 유의점이 있지만, 중요한 사실은 렌더링이 CPU와 GPU 사이에서 이루어지는 작업이라는 점입니다. 아래의 스크린샷에서 보는 바와 같이 Xcode와 같은 툴을 통해 두 리소스에서 실제로 소요되는 시간을 상세하게 확인할 수 있습니다.

이러한 정보는 Unity 프로파일러에서도 확인할 수 있지만, GPU 지표는 그래픽 카드와 드라이버의 지원 사항에 따라 표시되지 않을 수도 있습니다.

▼ 프레임레이트 관리와 리소스 최적화 실습

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. 기기의 한계인 60fps까지 밀어 붙이기 보다는 30fps 정도에서 타협해 실행하는 것이 좋습니다. Unity는 모바일의 경우 30fps를 기본으로 설정합니다.

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

Split your hierarchies. 게임 오브젝트가 계층 구조 내에 중첩될 필요가 없다면 부모 자식 관계를 간소화하세요. 계층 구조가 단순하면 씬에서 트랜스폼을 새로고침할 때 멀티스레딩의 이점을 누릴 수 있습니다. 계층 구조가 복잡하면 불필요한 트랜스폼 연산과 높은 가비지 컬렉션 비용이 발생합니다.

트랜스폼에 관한 베스트 프랙티스는 계층 구조 최적화와 이 Unite 세션을 참고하세요.

트랜스폼 한 번에 이동

아울러 트랜스폼 이동 시, Transform.SetPositionAndRotation을 사용하여 위치와 회전을 한 번에 업데이트하세요. 이렇게 하면 트랜스폼을 두 번 수정함으로써 발생하는 오버헤드를 방지할 수 있습니다.

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

▼ 게임 성능 향상을 위한 방법과 도구 학습

프로파일링

모바일 성능 데이터의 프로파일링과 수집 및 활용 과정은 진정한 모바일 성능 최적화가 시작되는 지점입니다.

개발 초기부터 타겟 기기에서 자주 프로파일링 실행하기

Unity 프로파일러는 사용 시 애플리케이션에 대한 필수 성능 정보를 제공합니다. 출시가 멀지 않은 시점이 아닌 개발 초기에 프로젝트를 프로파일링하세요. 여러 오류나 성능 문제를 발생 즉시 조사하세요. 프로젝트의 '성능 시그니처'를 개발하면서 새로운 문제를 보다 쉽게 발견할 수 있습니다.

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

Unity 프로파일러와 더불어 다음과 같은 iOS 및 Android의 네이티브 툴을 사용하면 각 엔진에서 추가로 성능 테스트를 수행할 수 있습니다.

- iOS: Xcode 및 Instruments
- Android: Android Studio 및 Android Profiler

특정 하드웨어는 Arm Mobile Studio, Intel VTune, 및 Snapdragon Profiler 등 추가 프로파일링 툴을 활용할 수 있습니다. 자세한 내용은 Profiling Applications Made with Unity 학습 자료를 참고하세요.

올바른 영역 최적화하기

게임 성능을 저하시키는 요인을 추정하거나 가정하지 않도록 합니다. Unity 프로파일러 및 플랫폼별 툴을 사용하여 성능 저하의 정확한 원인을 찾으세요.

물론 여기에서 설명하는 최적화가 모두 애플리케이션에 적용되지는 않습니다. 다른 프로젝트에는 적합했던 최적화라도 현재 프로젝트에는 적용되지 않을 수 있습니다. 실제 병목 지점을 파악하고 실질적으로 최적화가 필요한 부분에 집중하세요.

Unity 프로파일러의 작동 방식 이해하기

Unity 프로파일러는 런타임 시 성능 저하 또는 중단의 원인을 감지하고 특정 프레임 또는 시점에 발생하는 상황을 보다 정확하게 이해하는 데 도움이 될 수 있습니다. CPU 및 메모리 트랙을 기본적으로 활성화하세요. 물리를 많이 사용하는 게임이나 음악에 기반한 게임플레이를 개발 중이라면 필요에 따라 렌더러, 오디오, 물리와 같은 보조 프로파일러 모듈을 모니터링할 수 있습니다.

Development Build 및 **Autoconnect Profiler**를 선택하여 기기에 애플리케이션을 빌드하거나 수동으로 연결하여 앱 시작 시간을 단축하세요.

프로파일링을 실행할 타겟 플랫폼을 선택하세요. **녹화** 버튼은 애플리케이션 재생을 몇 초 동안 추적합니다(기본 300 프레임). 더 오래 캡처해야 한다면 **Unity > Preferences > Analysis > Profiler > Frame Count**로 이동하여 이 값을 2000까지 늘릴 수 있습니다. 이렇게 하면 Unity 에디터가 더 많은 CPU 작업을 수행하고 더 많은 메모리를 사용하지만, 상황에 따라 유용할 수 있습니다.

이는 계측 기반 프로파일러로 ProfileMarkers에 명시적으로 래핑된 코드 타이밍을 프로파일링합니다(예: MonoBehaviour의 Start나 Update 메서드 또는 특정 API 호출). 또한 Deep Profiling 설정을 사용하면 Unity는 스크립트 코드에 있는 모든 함수 호출의 시작과 끝을 프로파일링하여 정확히 애플리케이션의 어느 부분에서 성능 저하가 발생하는지 제시합니다.

게임을 프로파일링할 때는 성능 불안정과 평균 프레임의 비용을 모두 살펴보는 것이 좋습니다. 각 프레임에서 발생하는 높은 비용의 작업을 파악하고 최적화하면 타겟 프레임 속도 이하에서 실행되는 애플리케이션에 더 유용할 수 있습니다. 스파이크 현상을 살펴볼 때는 물리, AI, 애니메이션 등의 고비용 작업을 먼저 검토한 다음 가비지 컬렉션을 살펴봅니다.

창을 클릭하여 특정 프레임을 분석하세요. 그리고 나서 **타임라인** 또는 **계층 구조** 뷰를 사용합니다.

- **타임라인**: 특정 프레임의 타이밍을 시각적으로 요약하여 제시합니다. 이를 통해 각 활동이 다양한 스레드 전반에서 서로 어떤 관계를 맺고 있는지 시각화할 수 있습니다. 이 옵션을 사용하여 CPU 바운드 또는 GPU 바운드 여부를 판단하세요.
- **계층 구조**: 그룹화된 ProfileMarkers의 계층 구조를 제시합니다. 이를 통해 밀리초 단위(**Time ms** 및 **Self ms**)의 시간 비용을 기준으로 샘플을 정렬할 수 있고, 함수에 대한 **호출** 수와 프레임의 관리되는 힙 메모리(**GC Alloc**)의 양도 알 수 있습니다.