unity

베스트 프랙티스 가이드

게임 개발을 최적화하는 9가지 방법

Unity 전문가가 소개하는 게임 출시 팁

The Explorer: 3D 게임 키트 - Made with Unity

개요 **4**

1. 계획 **5**

기능 요구 사항 및 타겟 플랫폼 조사 5
메모리 및 성능 할당량 정의 5
빌드 및 QA 프로세스 계획 6
맨 처음부터 프로젝트 제작하기 6

2. 개발 및 워크플로 **7**

반복적인 수동 작업의 자동화 7
버전 관리 사용 7
캐시 서버 활용 9
플러그인 관련 주의 사항 9
협업을 통한 씬과 프리팹 작업 9

3. 프로파일링 **10**

프로파일링을 자주 실시해 사전에 문제 해결 10
Unity 툴 11
Unity 프로파일러로 핵심 프로젝트 영역 조사 11
Profile Analyzer가 회귀 테스트를 실행하도록 설정 13
메모리 사용량 캡처 및 시각화 14
렌더링 및 배칭 플로 점검 14
플랫폼/공급업체별 툴 14

4. 에셋 **15**

텍스처를 위한 올바른 임포트 설정 16
메시 옵션을 비활성화하여 메모리 재확보 17
오디오 클립에 가장 적합한 압축 포맷 선택 18
리소스 폴더 대신 AssetBundle 사용 18

© 2019 Unity Technologies

5. 프로그래밍 및 코드 아키텍처 **19**

추상 코드 사용 지양 19
Unity 플레이어 루프 이해 19
적합한 프레임 속도 선택 20
동기식 로딩 지양 20
사전 할당된 오브젝트 풀 사용 20
표준 동작 메서드 사용 최소화 21
많은 리소스가 필요한 API 결과 캐싱 21
런타임 도중 문자열 작업 지양 21
의도치 않은 디버그 로깅 지양 21
중요한 경로에 LINQ 쿼리 사용 지양 22
비할당 API 사용 22
정적 데이터 파싱 지양 22

6. 물리 **23**

7. 애니메이션 **25**

8. GPU 성능 **26**

오버드로우 및 알파 블렌딩에 유의 27
적은 셰이더 배리에이션 사용 27
과도한 카메라 컴포넌트 사용 지양 27
정적 배칭 및 동적 배칭 27
포워드 렌더링 및 LOD 27

9. 사용자 인터페이스(UI) **28**

다양한 해상도 및 종횡비 28
소수의 캔버스 사용 지양 29
레이아웃 그룹에 유의 29
많은 리소스를 소요하는 List 및 Grid 뷰 29
여러 요소가 겹치지 않도록 유의 29
Mask 및 RectMask2D 컴포넌트의 사용 29
UI 텍스처를 아틀라스 처리하여 배칭 개선 29
새 UI 창이나 화면을 추가할 때 유의할 사항 30
필요하지 않은 경우 Raycast Target 비활성화 30

추가 팁 **31**

© 2019 Unity Technologies

개요

게임을 개발하는 경우 대부분 창의적인 아이디어를 구현하고 최고의 게임으로 만들어 출시하려는 목표를 가지고 시작합니다. 게임 개발은 큰 도전이며, 그만큼 값진 결실을 가져다줄 수 있습니다. 그러나 완성도 높은 뛰어난 게임을 출시하는 여정에는 수많은 난관이 존재합니다.

본 가이드는 유니티의 ISS(Integrated Success Services) 전문가가 그러한 문제를 염두에 두고 9가지의 중요한 개발 영역으로 구성한 안내서입니다. 본 안내서를 참고하여 일반적으로 많이 발생하는 메모리, 성능, 플랫폼 문제를 미리 알아보고 대비하시기 바랍니다.

Integrated Success Services

유니티의 ISS 프로그램 파트너는 프로젝트 최적화와 원활한 진행을 위해 게임 스튜디오에 전담 개발사 관리 엔지니어를 비롯한 전문가들을 배치했습니다. 유니티의 ISS 전문가들은 풍부한 경험과 뛰어난 전문성으로 게임 스튜디오에서 생산성을 높이고 개발 기한을 준수하며, 새로운 영역에 진출하는 노하우를 전해 줍니다.

이 가이드에서 제공하는 ISS 전문가의 전략 분석과 최적화 권장 사항은 출시 지연을 초래하는 문제를 *사전에* 방지하고 게임 출시 초기부터 플레이어들에게 최고의 경험을 선사하는 데 도움이 될 것입니다. 유용한 베스트 프랙티스를 바탕으로 성공을 견인하시기를 바랍니다.

제공된 스크린샷은 여러 Unity 버전에서 캡처한 것으로, 현재 사용 중인 Unity 버전과 정확하게 일치하지 않을 수 있습니다.

© 2019 Unity Technologies

1. 계획

기능 요구 사항 및 타겟 플랫폼 조사

프로젝트를 시작하거나 규모가 큰 작업을 진행하기에 앞서 기능 요구 사항과 타겟 플랫폼을 철저히 조사해야 합니다. 모든 타겟 플랫폼이 개발자에게 필요한 사항을 지원하는지 확인하세요. 예를 들면 저사양 모바일 디바이스에서 인스턴스드 렌더링이 지원되지 않을 수 있습니다. 제한 사항과 활용할 수 있는 차선책, 절충안 등을 생각해두는 것이 좋습니다.

타겟 플랫폼마다 최소 사양을 결정하고 개발 및 QA 팀에서 사용할 하드웨어 유닛을 넉넉하게 확보해 두세요. 개발을 진행하는 동안 테스트를 위해 타겟 하드웨어가 다양하게 필요할 것입니다. 이렇게 하면 실질적인 성능과 프레임 할당량을 개발 과정 내내 모니터링하는 동시에 간편하게 분석하고 수정할 수 있습니다.

메모리 및 성능 할당량 결정

목표 사양 및 지원이 필요한 기능을 확정했으면 메모리와 성능 할당량을 결정할 차례입니다. 이러한 요소는 결정하기도 까다롭고 개발을 진행하는 동안 세부적으로 조정해야 할 수도 있지만, 아무 계획도 없이 프로젝트에 필요한 요소를 마구잡이로 추가하기보다는 합리적인 계획을 세우고 진행하는 편이 훨씬 낫습니다.

먼저 목표 프레임 속도와 원하는 CPU 성능 할당량을 결정합니다. 모바일 플랫폼의 경우, 서멀 스로틀링이 발생하여 CPU 및 GPU 클럭 속도를 저하시킬 수 있으므로 계획 단계에서 오버헤드를 허용합니다.

CPU 할당량을 가늠할 때는 렌더링, 이펙트, 핵심 로직 등 필요한 각각의 시스템에 어느 정도의 시간을 사용할지 결정합니다.

메모리 할당량은 결정하기가 꽤 어려울 수 있습니다. 에셋은 가장 많은 메모리를 사용하며 개발자가 주로 관리하게 되는 영역입니다. 에셋에는 전반적인 메모리를 얼마나 사용할지 생각해 볼 수 있습니다. 텍스처, 메시, 사운드 등은 많은 메모리를 사용하므로 주의하지 않으면 통제가 어려워질 수 있습니다. 과도하게 큰 텍스처 사용을 지양하고 화면에 표시되는 크기와 타겟 플랫폼의 화면 해상도에 따라 크기를 적절히 조절합니다. 또한 메시의 버텍스 및 폴리곤 수도 사용 사례에 맞게 적절히 조절합니다. 멀리서만 보이는 오브젝트에는 디테일 수준이 높은 메시가 필요하지 않습니다.

마지막으로, 프로젝트 시스템에 메모리를 얼마나 할당할지 고려합니다. 예를 들어 업데이트당 CPU 연산량을 줄이기 위해 많은 데이터를 사전에 계산하는 특정한 시스템을 구현했다고 가정해 보겠습니다. 하지만 과연 이 시스템이 합리적인지, 불균형하게 많은 메모리를 소모하지는 않는지 점검해야 합니다. 만약 성능에 가장 중요한 시스템이라면, 메모리를 희생해야 할 수도 있습니다. 여러 요인을 사전에 고려하여 계획하세요.

© 2019 Unity Technologies

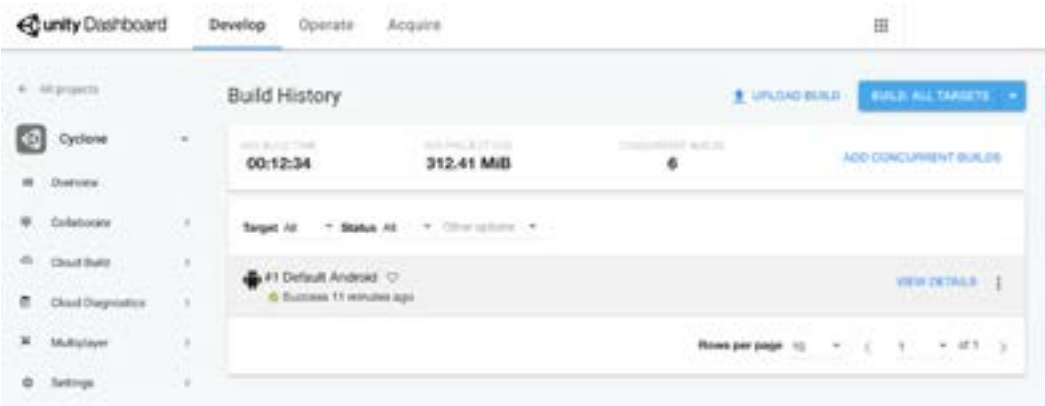
빌드 및 QA 프로세스 계획

빌드 및 QA 프로세스 계획은 매우 중요한 과정입니다. 빌드와 테스트를 로컬 환경에서 수행하면 유용한 측면도 있지만, 시간이 많이 걸리고 오류가 발생하기 쉽다는 단점도 있습니다. 다양한 솔루션을 고려해 보세요(널리 사용되는 솔루션의 예로는 Jenkins가 있음). 자체적으로 빌드 및 테스트 전용 빌드 머신을 준비할 수도 있고, 다양한 클라우드 서비스 중 하나를 사용해 자체적으로 머신을 관리할 경우 소요되는 각종 비용을 줄일 수도 있습니다. 또는 Unity 팀즈의 클라우드 빌드 기능을 활용하는 것도 고려해볼 수 있습니다. 충분히 고민한 다음 요구 사항에 어울리는 솔루션을 선택하세요.

릴리스 빌드에 기능이 어떻게 퍼블리시될지 계획하는 것도 도움이 됩니다. 버전 관리를 사용하여 개발 브랜치를 어떻게 빌드하고 검증할지 고려하세요. 빌드 프로세스의 일환으로 실행하는 자동화된 테스트는 여러 가지 문제를 포착할 수 있지만 완벽하다고 보기는 어렵습니다. 자동화 테스트는 지표를 수집하고 빌드 전반에 걸친 테스트 성과 이력을 확보하는 데 유용하며, 이를 통해 재발하는 문제를 더 빠르게 발견할 수 있습니다. 프로세스에 수동 품질 검증 및 승인 단계도 포함해야 할 수 있습니다. 빌드 프로세스에서는 검증된 브랜치를 릴리스 빌드에 병합할 수 있습니다.

맨 처음부터 프로젝트 제작하기

프로젝트 프로토타이핑이 끝나면 모든 것을 처음부터 만들며 제작을 시작하는 것에 대해 신중히 검토합니다. 프로토타이핑 단계에서 내린 의사 결정은 대부분 작업 속도를 우선시했기 때문에 프로토타입 프로젝트에는 간소화된 요소가 다수 포함되어 있는 경우가 많으므로, 프로젝트를 시작하는 기반으로 삼기에는 부족할 수 있습니다.



Unity 클라우드 빌드를 사용하여 프로젝트의 자동 빌드 생성

© 2019 Unity Technologies

2. 개발 및 워크플로

대부분의 개발자는 워크플로 자체의 문제와 끊임없이 씨름하기보다는 창의적이고 생산적인 작업에 시간을 쏟고 싶어합니다. 하지만 한 순간의 잘못된 판단으로 개발 과정에서 효율성이 낮아지며 팀 단위의 실수가 발생하고, 결과적으로 최종 결과물의 품질이 저하될 수 있습니다. 이를 완화하기 위해 공통 작업을 더 효과적이며 간소하게 만들어 팀의 작업 진행에 방해가 될 만한 오류를 최소화할 수 있습니다.

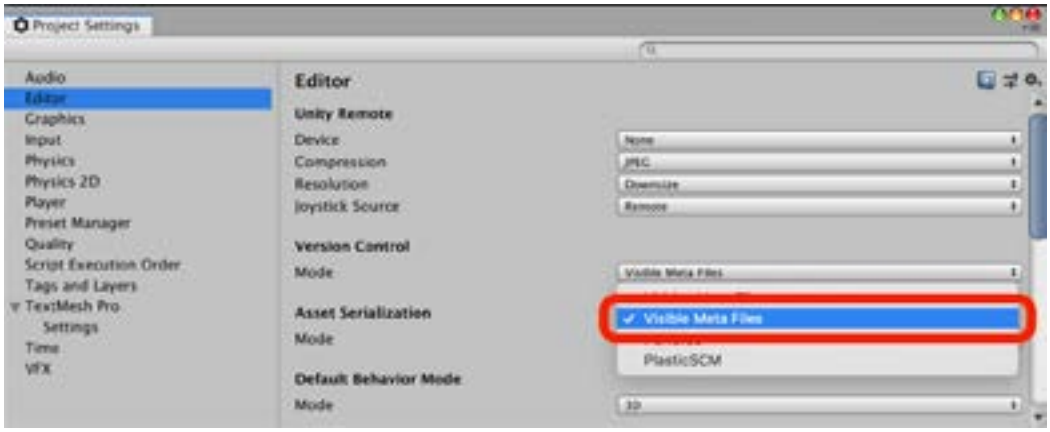
반복적인 수동 작업의 자동화

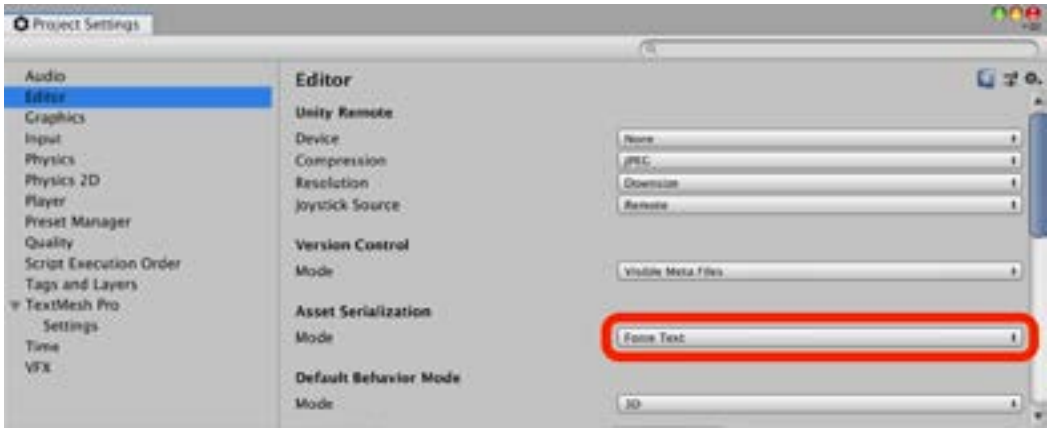
자주 반복되는 수동 작업은 우선적으로 자동화를 고려해야 합니다(예: 커스텀 스크립트 또는 에디터 툴 사용). 자동화를 통해 프로젝트가 진행되는 동안 시간과 노력을 크게 절약할 수 있습니다. 작업을 자동화하면 사람의 실수로 인한 오류를 방지할 수도 있습니다.

특히, 빌드 프로세스를 완전히 자동화하고 로컬이나 지속적 통합(Continuous Integration) 서버에서 한 번의 동작만으로 빌드가 되도록 해야 합니다.

버전 관리 사용

개발자라면 누구나 버전 관리를 사용할 것입니다. Unity는 다양한 솔루션을 위한 빌트인 지원을 제공합니다.





Unity 에디터 내 버전 관리 설정

Unity의 버전 관리를 사용하려면 프로젝트의 에디터 설정에서 Asset Serialization Mode가 기본 설정인 Force Text로 설정되어 있어야 합니다.

또한 Unity에는 씬 및 프리팹 병합에 특화된 빌트인 YAML(사람이 읽을 수 있는 데이터 직렬화 언어) 툴이 있습니다. 이 툴이 설정되어 있는지도 확인하세요. Unity 매뉴얼의 스마트 병합 항목에서 더 자세한 내용을 알아볼 수 있습니다.

자체 버전 관리 솔루션이 Git과 같은 커밋 훅을 지원하는 경우, 커밋 훅을 활용하여 특정한 표준을 적용할 수 있습니다. Unity Git Hooks 페이지에서 더 자세한 내용을 알아보세요.

개발 작업 시 주요 브랜치가 아닌 작업 버전을 별도로 관리하여 프로젝트의 버전이 혼재되지 않게 합니다. 브랜치와 태그는 마일스톤과 릴리스를 관리하는 용도로 활용합니다.

명확한 작업 방침을 팀 내에 공유하세요. 메시지가 명확하고 상세하면 개발이 진행되는 동안 문제를 추적하는 데 도움이 되는 반면, 무의미한 메시지는 혼란만 초래할 뿐입니다.

마지막으로, 버전 관리를 잘하면 문제가 발생할 때 신속하게 추적할 수 있습니다. 예를 들어 Git의 경우 올바른 수정 버전과 잘못된 수정 버전을 표시하도록 하는 바이섹트(bisect) 기능을 제공하므로, '분할 정복(divide and conquer)' 접근 방식을 사용하여 수정 버전을 확인하고 테스트해 어느 쪽이 올바른 버전인지 표시할 수 있습니다. 버전 관리 시스템에 어떤 기능이 있는지, 개발을 진행하며 직접 사용할 기능은 무엇이 있을지 알아보는 데 시간을 투자하는 편이 좋습니다.

© 2019 Unity Technologies

캐시 서버 활용

Unity 에디터에서 타겟 플랫폼을 전환하는 데에는 시간이 걸리며, 이 시간은 프로젝트 규모가 클수록 늘어나므로 Unity 캐시 서버를 사용하는 것이 좋습니다. 지원해야 하는 프로젝트나 Unity 엔진 버전이 다양할 경우에는 여러 캐시 서버를 실행할 수 있으며, 이때 개별 포트를 사용하도록 선택할 수도 있습니다.

또한 캐시 서버를 사용하면 로컬에서 플랫폼 전환에 소요되는 시간이 크게 단축되므로 꼭 사용해 볼 것을 권장합니다.

플러그인 관련 주의 사항

프로젝트에 다수의 플러그인과 타사 라이브러리가 포함된 경우에는 그중 사용되지 않는 에셋이 게임에 빌드될 가능성이 높습니다. 플러그인에는 임베드된 테스트 에셋이나 스크립트가 포함된 경우가 많기 때문입니다.

에셋 스토어에서 받은 에셋을 사용하는 경우에는 에셋에서 프로젝트로 가져오는 종속 관계를 확인해야 합니다. 의도치 않게 JSON 라이브러리가 다수 생성되는 경우도 더러 있습니다.

또한, 오래된 에셋이나 프로토타이핑 단계에 사용했던 스크립트처럼 필요 없는 리소스를 플러그인에서 제거해야 합니다.

협업을 통한 씬과 프리팹 작업

개발팀이 어떤 방식으로 협업하며 콘텐츠를 만들어 나갈지 검토해야 합니다. 하나의 커다란 Unity 씬에서는 협업이 원활하게 이뤄지지 않을 수 있습니다. 씬을 작은 단위로 쪼개면 아티스트와 디자이너가 더 원활하게 협업할 수 있으며, 충돌이 발생할 위험도 낮아집니다. 런타임 시 프로젝트가 LoadSceneMode.Additive 파라미터 모드를 전달하는 SceneManager.LoadSceneAsync() API를 사용하여 씬을 가산적으로 로드할 수 있습니다.

Unity 2018.3 이후 버전에는 향상된 프리팹이 적용되어 프리팹 중첩을 지원합니다. 따라서 프리팹 콘텐츠를 소규모의 개별 항목으로 쪼개어 팀원들이 충돌의 위험 없이 독립적으로 작업할 수 있습니다. 하지만 팀에서 동일한 에셋에 액세스하여 작업해야 하는 경우에는 어떤 방식으로 작업할지 미리 합의하세요. 이 경우 Slack 채널 및 이메일과 같은 경로를 통해 팀 멤버들이 작업 현황을 서로 알릴 수 있는 커뮤니케이션 정책을 마련하면 간편합니다. 버전 관리 솔루션에서 커뮤니케이션이 지원되는 경우, 솔루션의 내 파일 체크인/체크아웃 메커니즘을 통해 커뮤니케이션을 관리할 수도 있습니다.

© 2019 Unity Technologies

3. 프로파일링

프로파일링을 자주 실시해 사전에 문제 해결

프로파일링을 자주 실시해 일정이 막바지에 다다르거나 프로젝트에 성능 문제가 보이기 전에 해결하세요. 평소 프로젝트의 일반적인 '성능 시그니처'를 잘 알아두면 새로운 성능 문제를 더욱 손쉽게 파악할 수 있습니다. 글리치나 스파이크를 발견하면 되도록 빨리 확인해야 합니다.

프로파일링을 위한 중요 팁

- 추측을 기반으로 프로젝트를 최적화하지 말고, 반드시 프로파일링을 진행하면서 발견한 내용을 토대로 최적화합니다.

- Unity의 프로파일링 툴과 플랫폼/벤더별로 특정한 프로파일링 툴을 모두 사용해 프로젝트가 전반적으로 어떻게 동작하는지 확실하게 파악합니다.

- Unity 에디터 내에서 프로젝트를 프로파일링하는 방법도 유용할 수 있지만, 프로파일링은 항상 타겟 플랫폼에서 진행합니다.

- 자동 테스트를 사용하여 재발하는 문제를 찾아내고 문제 원인을 조사합니다. 프로파일링을 캡처한 내용을 저장하고 Profile Analyzer 등의 툴을 사용해 수동으로 비교하는 방법도 고려할 수 있습니다.

© 2019 Unity Technologies

Unity 툴

Unity의 자체 프로파일링 툴은 프로젝트의 개발 빌드로 실행하는 경우가 일반적이며, 최종 빌드와는 성능에 차이가 있습니다. 그러나 디버그 빌드 대신 릴리스 빌드로 설정하면 성능 개요를 비교적 정확하게 확인할 수 있습니다.

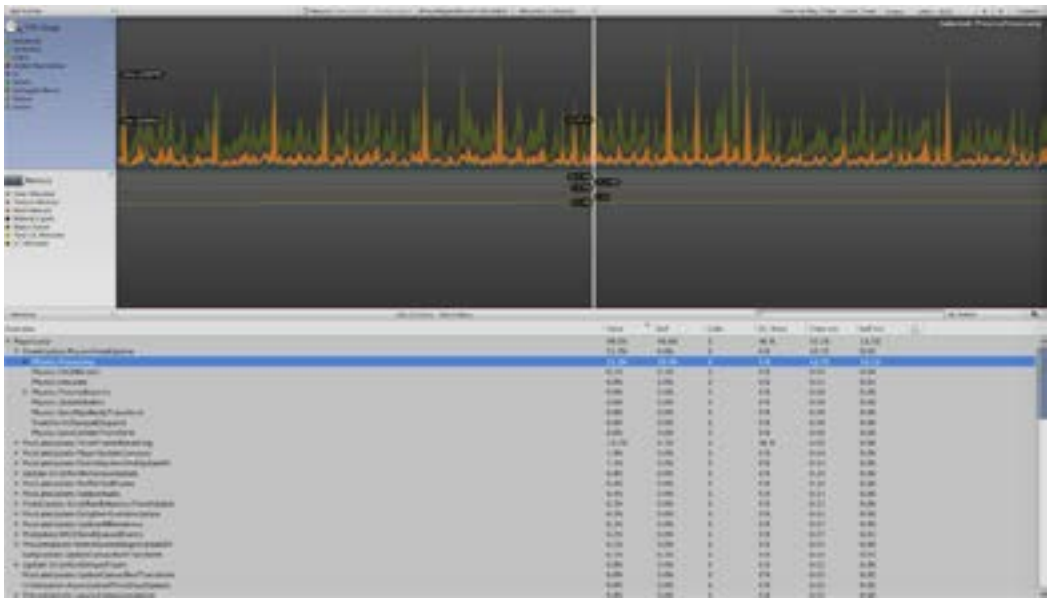
Unity 프로파일러로 핵심 프로젝트 영역 조사

Unity 프로파일러를 자주 사용하여 프로젝트의 핵심 영역을 조사합니다.

유의미한 Unity 프로파일러 샘플러를 스크립트에 추가하면 딥 프로파일링(Deep Profiling) 없이도 가치 있는 정보를 확보할 수 있습니다. 이러한 방식은 성능에 큰 영향을 미치며 Unity 에디터와 JIT(Just-in-time) 모노 컴파일링을 지원하는 플랫폼(예: Windows, macOS 및 Android)에서만 유효합니다.

개별 프로파일러 툴 트랙(CPU, GPU, 물리 등) 은 재배열하거나, 추가하거나, 제거할 수 있습니다. Unity 2018.3 이후 버전을 사용하면 현재 필요하지 않은 트랙을 삭제하여 타겟 플랫폼에서의 런타임 시 CPU 오버헤드를 줄일 수 있습니다. 지난 버전의 Unity에서는 런타임까지 모든 프로파일링 데이터가 무조건 수집되어 프로파일러 오버헤드가 증가했습니다.

프로파일러의 계층 뷰에서 Total 열을 정렬해 CPU 성능 요구치가 가장 높은 영역을 파악하고, 조사가 필요한 부분을 집중적으로 둘러볼 수 있습니다.

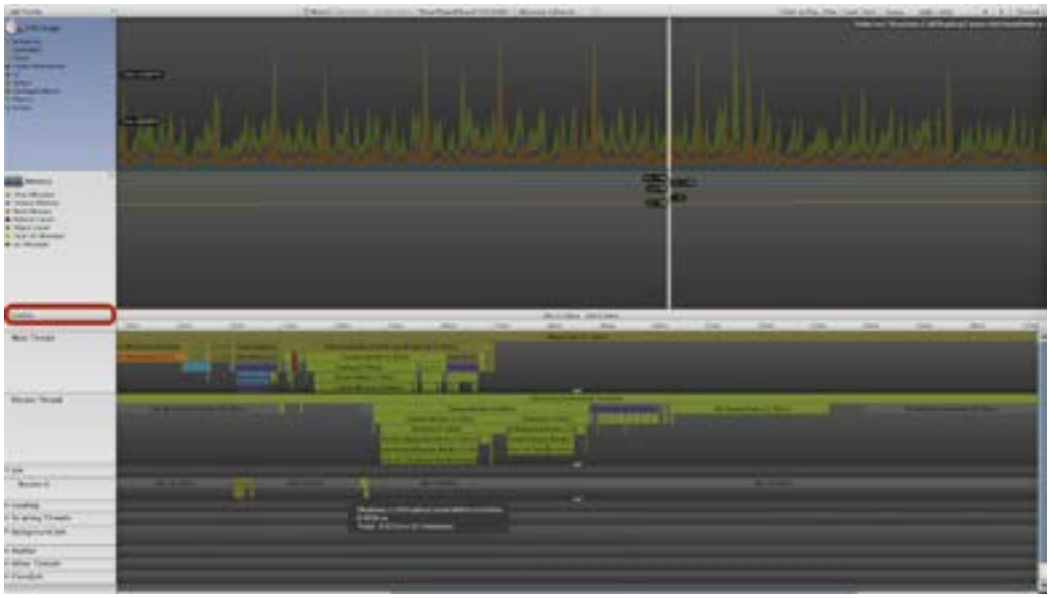


Unity 프로파일러 계층 뷰를 통해 가장 많은 리소스를 사용하는 코드 함수 프로파일링

© 2019 Unity Technologies

이와 유사하게 GC Alloc 열을 정렬하면 관리되는 할당량을 생성하는 영역을 파악할 수 있습니다. 정기적으로, 또는 프레임마다 발생하는 할당량을 최대한 줄이는 것을 중점으로, 관리되는 힙을 급증시켜 가비지 컬렉션을 유발하는 할당량 스파이크가 발생하는 부분이 있는지 유심히 살펴보세요. 관리되는 힙 크기를 최대한 작게 유지하세요. 메모리를 많이 사용할수록 더 많은 조각으로 나뉘어 가비지 컬렉션이 되고, 이로 인해 리소스에 부담이 갈 수 있습니다.

Unity 메인 스레드에만 집중하는 계층 뷰와는 달리, 프로파일러의 Timeline 뷰는 Unity 메인 및 렌더링 스레드, 잡 시스템 스레드 및 사용자 스레드(적절한 프로파일러 샘플러가 추가된 경우)를 비롯한 여러 스레드에 대한 유용한 정보를 시각적으로 전달합니다.

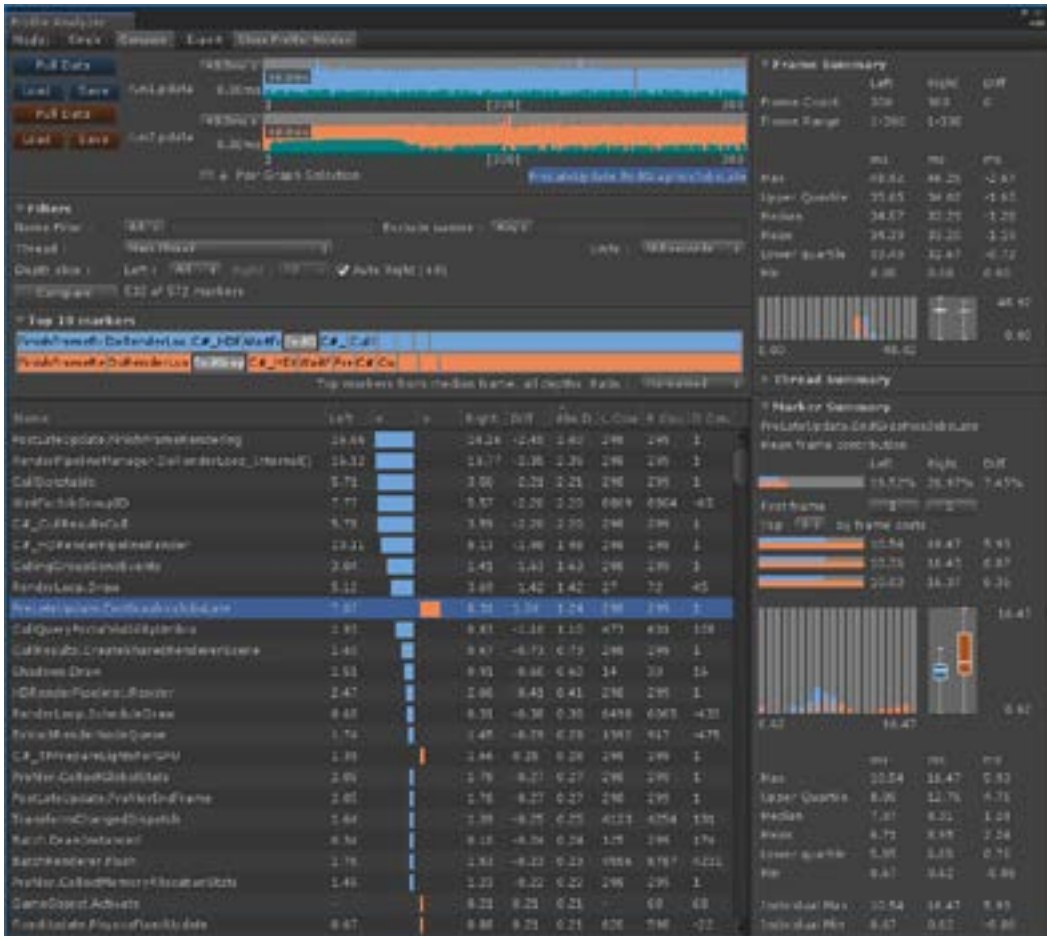


Unity 프로파일러 Timeline 뷰에 표시되는 스레드 간 프로파일링

© 2019 Unity Technologies

Profile Analyzer가 회귀 테스트를 실행하도록 설정

Profile Analyzer를 사용하면 현재 캡처된 파일이나 기존에 디스크에 저장된 캡처 파일의 데이터를 Unity 프로파일러에서 임포트하고 다양하게 분석할 수 있습니다. 평균, 중간 및 최대 비용으로 분류된 프로파일링 마커를 확인할 수 있을 뿐만 아니라 데이터 세트를 비교할 수도 있습니다. 이러한 방식은 변경 전후에 캡처된 데이터를 비교하는 회귀 테스트 등에 유용합니다.



전후 프로파일러 데이터를 비교해 개선 또는 회귀 확인

© 2019 Unity Technologies

메모리 사용량 캡처 및 시각화

메모리 프로파일러는 애플리케이션에서 사용하는 일부 메모리를 캡처하고 시각화하는 강력한 툴로, 프리뷰 패키지를 통해 패키지 관리자에서 사용할 수 있습니다. Tree Map 뷰는 텍스처 및 메시와 같은 일반적인 유형의 에셋뿐 아니라 문자열이나 프로젝트별 에셋 등 여타 관리되는 유형의 메모리 사용량을 즉시 시각화하는 데 특히 유용합니다.

에셋이 소모하는 메모리 영역에서 발생할 수 있는 문제를 확인합니다. 예를 들어, 텍스처가 비정상적으로 크게 표시된다면 임포트 설정이 잘못되었거나 해상도가 지나치게 높은 경우일 수 있습니다. 중복 에셋도 여기에서 찾을 수 있습니다. 서로 다른 에셋이 동일한 이름을 가질 수 있지만, 이름과 크기가 같은 에셋이 여럿 중복될 수 있으므로 확인해 보아야 합니다. 이 경우 AssetBundle이 잘못 할당된 것이 원인일 수 있습니다.

항상 그런 것은 아니지만, 일반적인 프로젝트에서는 텍스처 메모리가 메시 메모리보다 더 큰 경향이 있습니다. 이를 고려하여 텍스처 메모리 사용량보다 메시 메모리 사용량이 더 높지는 않은지 확인합니다.

렌더링 및 배칭 플로 점검

프레임 디버거 툴은 렌더링 흐름 및 배칭과 같은 영역을 검사하는 데 매우 유용할 수 있습니다. 예를 들면 툴에서 배치가 손상된 이유를 알려주므로 이를 확인하고 콘텐츠를 최적화할 수 있습니다.

프레임이 어떻게 구성되어 있는지 확인하면 동일한 콘텐츠가 중복해서 렌더링된다든가(예: 실제 프로젝트에서 카메라가 중복됨), 완전히 가려진 콘텐츠가 렌더링되는(예: 전체 화면 2D UI 뒤에 가려진 3D 씬을 계속 렌더링) 등의 문제도 발견할 수 있습니다.

플랫폼/벤더 툴

Unity의 프로파일링 툴도 다양한 기능을 제공하지만, 타겟 플랫폼의 기본 프로파일링 툴을 효과적으로 사용하는 방법을 배워두면 좋습니다.
예:

플랫폼	툴
iOS 및 macOS	Xcode 및 기기
Android	Android Studio
Windows	VTune
Windows, Xbox One	PIX
Windows	NVidia NSight
PlayStation 4	Razor

이러한 툴셋은 샘플링 및 계측 기반 CPU 프로파일링, 전체 네이티브 메모리 프로파일링, 셰이더 디버깅을 포함한 GPU 프로파일링과 같이 더 강력한 프로파일링 옵션을 제공합니다. 일반적으로 이러한 툴은 프로젝트의 비개발 빌드에서 사용하게 됩니다.

Unity 프로파일러 마커를 사용하면 모든 CPU 코어, 스레드 및 기능과 더불어 다양한 요소를 측정할 수 있습니다.

iOS를 예로 들면, Instruments' Allocations와 VM Tracker 툴을 사용한 메모리 프로파일링을 통해 타사 플러그인에서 할당한 원시 할당량 등 Unity의 메모리 프로파일러에 나타나지 않는 메모리 사용처를 알아낼 수 있습니다.

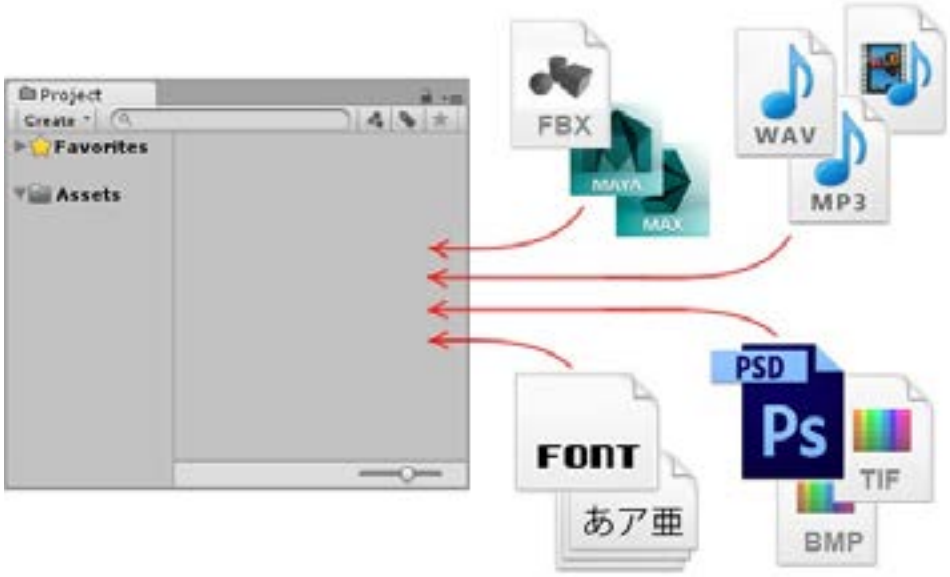
© 2019 Unity Technologies

4. 에셋

에셋 파이프라인은 매우 중요합니다. 텍스처, 메시 및 사운드와 같은 에셋은 프로젝트 메모리를 상당량 사용하며, 설정을 최적화하지 못하면 매우 많은 메모리가 소모될 수 있습니다.

이러한 문제를 방지하려면 적합한 사양의 효과적인 아트 콘텐츠 플로를 설정해야 합니다. 가급적 테크니컬 아티스트와 함께 이러한 프로세스를 함께 결정하고 프로젝트를 시작하는 편이 좋습니다.

먼저 어떤 포맷과 사양을 사용할지 명확한 가이드라인을 결정하세요.



여러 유형의 에셋 파일을 임포트할 수 있는 Unity

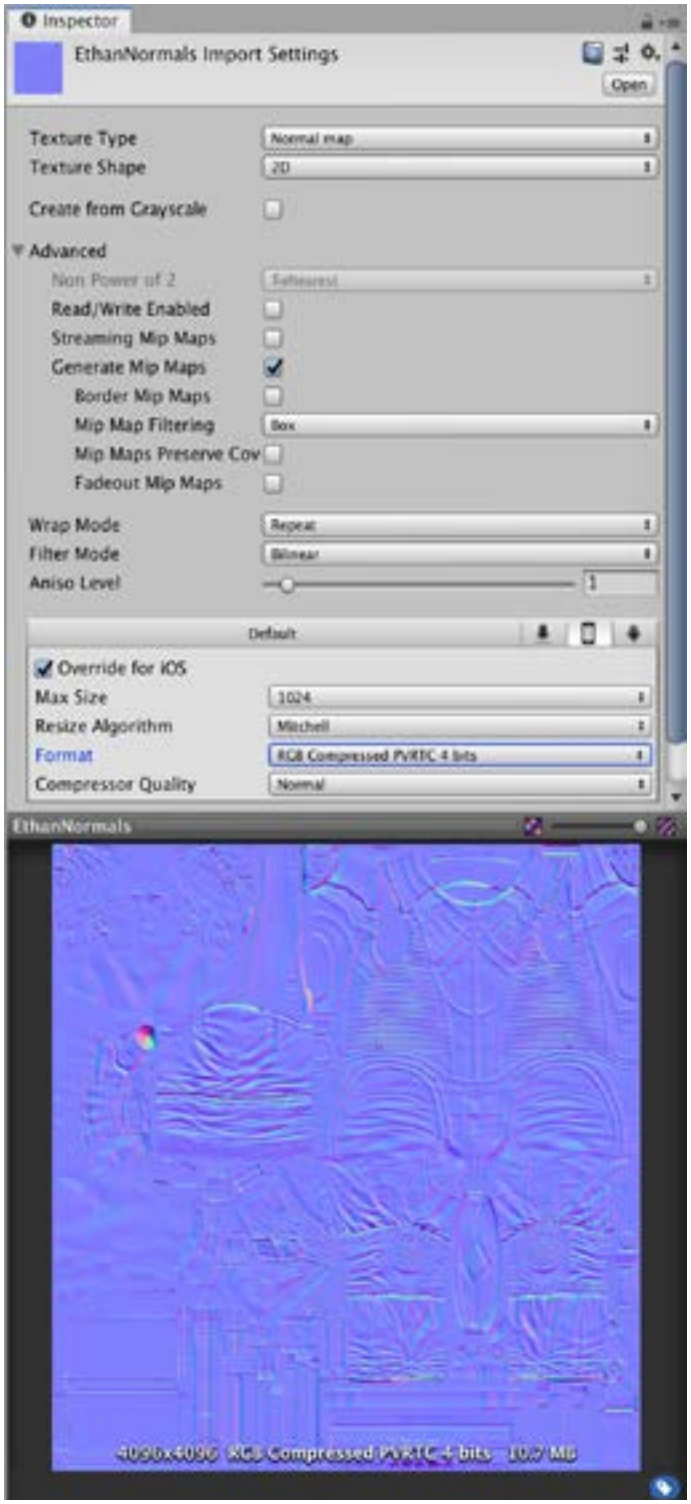
프로젝트의 모든 에셋에 올바른 임포트 설정을 구성하는 것이 매우 중요합니다. 팀원들이 작업 중인 에셋 유형의 설정을 이해하고 있는지 확인하고, 모든 에셋에 일관된 설정을 적용하도록 규칙을 따르게 합니다.

또한 모든 플랫폼에 기본 설정을 동일하게 사용하지 말고 플랫폼별 오버라이드 탭을 사용하여 에셋을 최적화합니다(예: 최대 텍스처 크기 또는 오디오 품질을 각각 다르게 설정).

이 프로젝트와 같이 AssetPostprocessor API를 사용하여 새 에셋에 대한 임포트 설정을 자동으로 적용하는 방법도 고려합니다.

참고 항목: 아트 에셋 베스트 프랙티스 가이드

© 2019 Unity Technologies



텍스처 에셋 임포트 설정

텍스처 임포트 올바르게 설정하기

텍스처는 일반적으로 모든 에셋 유형 중 가장 많은 메모리를 소비하므로 임포트 설정이 올바르게 되어 있는지 확인해야 합니다.

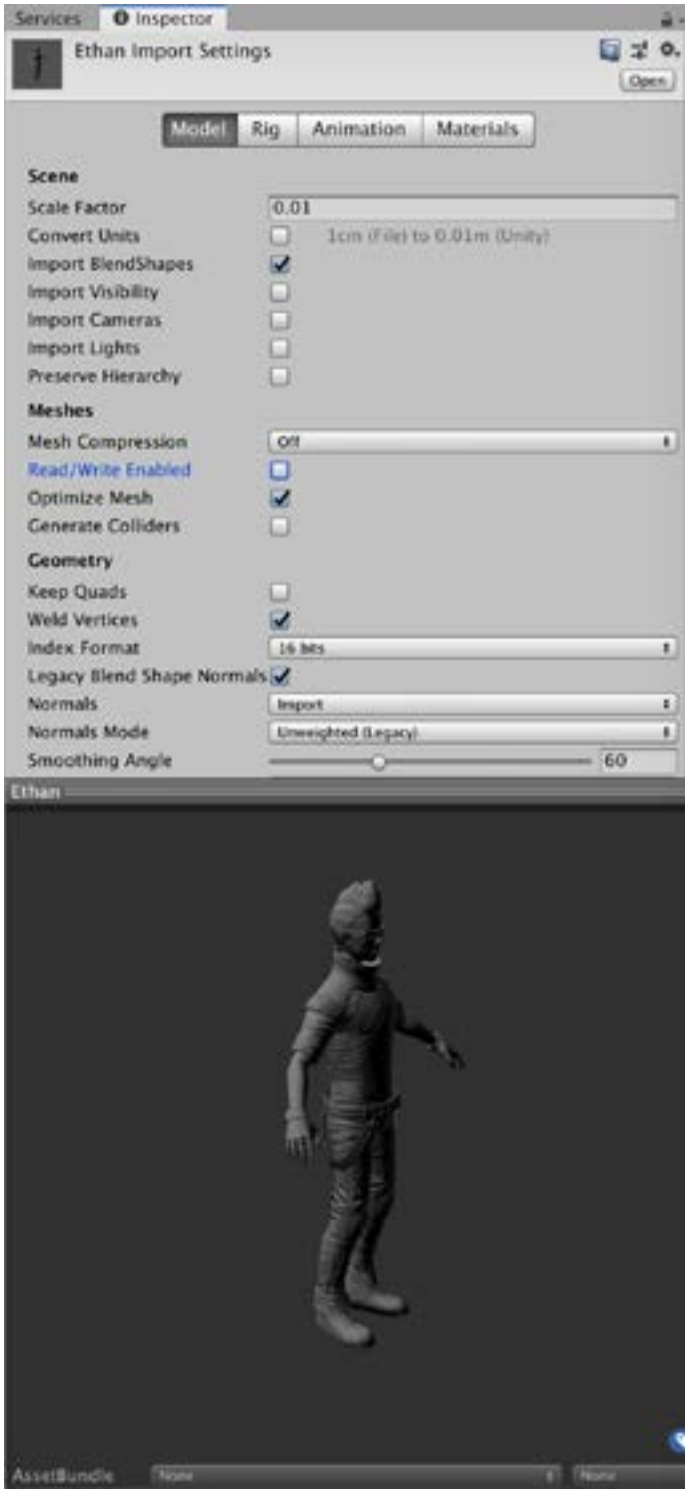
스크립트의 픽셀 데이터에 반드시 액세스해야 하는 경우가 아닌 이상 Read/Write Enabled 옵션을 활성화해서는 안 됩니다. 이 옵션을 활성화하면 CPU 주소 지정 가능 메모리(CPU-addressable memory)의 픽셀 데이터 사본과 GPU 주소 지정 가능 메모리에 업로드된 사본을 모두 유지하여 전체 메모리 사용량이 두 배로 늘어나게 됩니다. 다행히 런타임에 파라미터 *makeNoLongerReadable = false*를 전달하고 Texture2D.Apply() API를 통해 텍스처 에셋이 CPU 주소 지정 가능 복사본을 폐기하도록 할 수 있습니다.

밉맵은 필요한 경우에만 활성화합니다. 일반적으로 3D 씬에서 사용되는 텍스처에는 필요하지만, 2D의 경우에는 대개 필요하지 않습니다. 필요하지 않은 경우에 이 옵션을 활성화하면 텍스처의 메모리 사용량이 1/3 정도 증가합니다.

가능하면 항상 압축을 사용합니다. 단, UI 텍스처와 같은 일부 콘텐츠는 압축 시 명확하게 보이지 않을 수 있다는 점은 염두에 둬야 합니다. 타겟 플랫폼에 적용 가능한 압축 포맷의 장단점을 이해하고 올바른 포맷을 선택합니다. 예를 들어, iOS에서 PVRT 대신 ASTC를 사용하면 대개 정확도가 더 높은 결과를 얻을 수 있지만 Apple A8 칩 버전보다 낮은 저사양 기기에서는 해당 포맷이 지원되지 않습니다. 일부 개발자는 다양한 형태로 압축된 에셋을 준비해 특정 기기별로 지원하는 가장 좋은 품질의 에셋을 사용하기도 합니다.

여러 스프라이트를 그룹화하려면 스프라이트 아틀라스를 최대한 많이 사용합니다. 그러면 배칭이 향상되고 드로우 콜 횟수가 줄어듭니다. 현재 사용하고 있는 Unity 버전에 따라 Unity SpritePacker 또는 Unity SpriteAtlas를 사용하거나 Texture Packer와 같은 타사 툴을 사용할 수 있습니다.

© 2019 Unity Technologies



모델 에셋 임포트 설정

메시 옵션을 비활성화하여 메모리 재확보

스크립트에서 메시 데이터를 읽어야 하는 경우에만 Read/Write Enabled 옵션을 활성화합니다. 이 옵션은 기본으로 비활성화된 Unity 2019.3 버전 이전의 모든 Unity 버전에서는 활성화된 상태가 기본값이며 대개 그대로 유지됩니다. 이 옵션을 비활성화하면 많은 프로젝트에서 메모리를 재확보할 수 있습니다.

프로젝트가 여러 모델 소스 파일(예: FBX)을 사용하여 보이는 메시와 애니메이션을 임포트하는 경우, 해당 소스 파일에서 메시가 제거되었는지 확인합니다. 그렇지 않으면 메시 데이터가 최종 결과물에 포함되어 메모리를 낭비하게 됩니다.

© 2019 Unity Technologies

오디오 클립에 가장 적합한 압축 포맷 선택

대부분의 오디오 클립에서는 Load Type을 Compressed In Memory로 설정하는 것이 적절한 기본값입니다. 플랫폼마다 적합한 압축 포맷을 선택하세요. 콘솔에는 자체 커스텀 포맷이 있으며, 그 외의 모든 플랫폼에는 Vorbis가 적합합니다. Unity는 모바일 환경에서 하드웨어 압축 해제를 사용하지 않으므로 iOS에서 MP3를 채택해도 별다른 장점이 없습니다.

타겟 플랫폼에 적합한 샘플 속도를 선택합니다. 예를 들어, 모바일 기기에서 짧은 사운드 효과는 최대 22050Hz 이하여야 하며, 대부분의 경우 이보다 훨씬 낮아도 최종 품질에 미치는 영향은 미미합니다.

긴 음악이나 효과음의 Load Type이 Streaming으로 설정되어 있는지 확인합니다. 그렇지 않으면 전체 에셋이 한 번에 메모리에 로드됩니다.

Load Type을 Decompress On Load로 설정하면 원시 16비트 PCM 오디오 데이터로 사운드 압축을 풀어 CPU 및 메모리 소모가 발생합니다. 이 설정은 여러 동시 인스턴스가 자주 발생하는 발소리나 칼이 부딪히는 소리와 같은 짧은 사운드에만 적합합니다.

가능하면 원본의(압축되지 않은) 비압축 WAV 파일을 소스 에셋으로 사용합니다. MP3나 Vorbis 등의 압축 포맷을 사용하면 Unity는 빌드 시간 동안 압축을 풀고 다시 압축하며, 이로 인해 손실이 있는 패스가 두 번 발생하기 때문에 최종 품질이 저하됩니다.

프로젝트의 3D 공간 내에 사운드를 배치하려면 모노(싱글 채널)로 제작하거나 Force To Mono 설정을 활성화해야 합니다. 런타임 시 위치에 따라 멀티채널 사운드가 실시간으로 모노 소스로 평탄화되어 CPU 소모가 증가하고 메모리가 낭비되기 때문입니다.

리소스 폴더 대신 AssetBundle 사용

리소스 폴더에 에셋을 배치하는 대신 AssetBundle을 사용합니다. 리소스 폴더의 에셋은 일종의 내부 AssetBundle로 빌드되며, 이에 대한 헤더 정보는 시작하는 동안 로드됩니다. 따라서 많은 에셋을 이렇게 보관하면 시작 시간이 길어질 수 있습니다.

에셋 중복을 피하려면 종속 관계가 있는 에셋을 비롯한 모든 에셋을 AssetBundle에 할당합니다. 예를 들어, 동일한 텍스처를 사용하는 두 개의 머티리얼 에셋이 있다고 가정해 보겠습니다. 텍스처가 AssetBundle에 할당되지 않고 두 머티리얼이 별도의 AssetBundle에 할당된 경우, 텍스처는 레퍼런스 머티리얼과 함께 각 AssetBundle에 복제됩니다. AssetBundle에 텍스처를 명시적으로 할당하면 이러한 중복을 방지할 수 있습니다.

AssetBundle Browser 툴을 사용하면 여러 AssetBundle에 에셋과 종속 관계를 설정하고 추적할 수 있으며, AssetBundle Analyzer 툴을 사용하면 중복 에셋과 부차적인 설정이 있는 에셋을 강조 표시할 수 있습니다.

모든 상황에 적합한 단 하나의 솔루션은 없으므로, 콘텐츠를 AssetBundle에 빌드하는 계획을 세웁니다. 논리적 콘텐츠로 AssetBundle을 그룹화하는 경우가 많습니다. 예를 들어, 레이싱 게임에서는 차량 유형마다 필요한 모든 에셋을 고유한 AssetBundle에 넣을 수 있습니다.

하지만 패치를 지원하는 플랫폼의 경우 이러한 방법을 사용할 때는 주의해야 합니다. 압축 알고리즘이 사용되어 에셋을 조금만 변경하더라도 AssetBundle의 바이너리 데이터가 상당히 변경되어 효율적인 패치 델타를 생성할 수 없게 됩니다. 따라서 크기가 큰 AssetBundle을 소수만 사용하는 경우 큰 문제가 발생할 수 있습니다. 패치에 대비해 큰 AssetBundle을 소수만 사용하기보다는 작은 AssetBundle을 다수 사용하세요.

© 2019 Unity Technologies

5. 프로그래밍 및 코드 아키텍처

여기에 소개된 베스트 프랙티스를 따라 아키텍처를 올바르게 구성하면 팀의 생산성은 올라가고 사용자 경험도 향상시킬 수 있습니다.

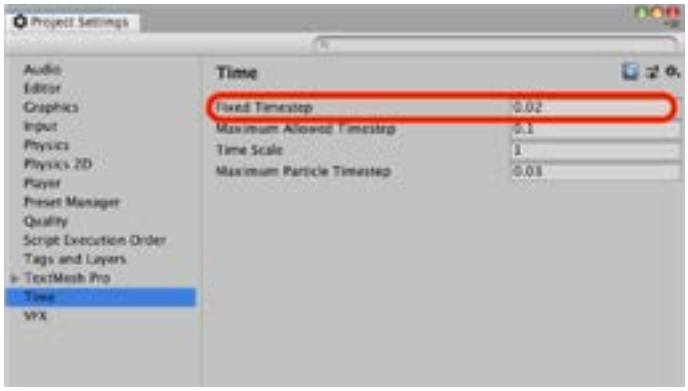
참고 항목: 스크립팅 경험을 향상하는 방법

추상 코드 사용 지양

과도하게 엔지니어링되고 추상적인 코드는 팀에 새로 합류한 인원이 파악하기가 매우 까다로울 수 있으며, 일반적으로 변경 사항을 분석하고 논의하기가 어렵습니다. 또한 이러한 유형의 코드가 있으면 더 많은 코드를 작성하게 되고, 빌드 시간이 길어지며(IL2CPP 빌드 포함, 더 많은 IL을 변환해야 함), 최종 코드의 성능이 상대적으로 떨어질 수 있습니다.

Unity 플레이어 루프 이해

Unity 프레임(또는 플레이어) 루프에 대해 충분히 이해해야 합니다. 예를 들어, Awake, OnEnable, Update 및 기타 메서드가 호출되는 시기를 파악하는 것이 매우 중요합니다. Unity 기술 자료에서 자세한 내용을 확인할 수 있습니다.



Fixed Timestep 값을 기본으로 설정하면 CPU 프로세싱이 추가로 필요한 경우가 자주 발생함

Update와 FixedUpdate의 차이를 파악하는 것이 특히 중요합니다. FixedUpdate 메서드는 프로젝트의 Fixed Timestep 값에 의해 제어되며, 기본값은 0.02(50Hz)입니다. 즉, Unity는 FixedUpdate 메서드가 초당 50회 호출되도록 보장하는데 이로 인해 단일 프레임에서 다중 호출이 발생할 수 있습니다. 프레임 속도가 떨어지면 FixedUpdate 호출 수가 증가하므로 이 문제가 악화됩니다. 이는 심각한 글리치를 유발하며 소위 '죽음의 나선(spiral of death)'이라고 불리는 사이클을 초래할 수 있습니다.

물리 시스템 업데이트는 FixedUpdate 단계의 일부이므로 물리 콘텐츠가 많은 게임은 이러한 문제의 영향을 심하게 받을 수 있습니다. 일부 프로젝트는 FixedUpdate를 통해 다양한 게임 시스템을 실행하므로 이러한 종류의 성능 문제를 확인하고 방지해야 합니다. 따라서 Fixed Timestep 값을 타겟 프레임 속도와 거의 일치하도록 설정하는 것이 좋습니다.

© 2019 Unity Technologies

적합한 프레임 속도 선택

적정한 수준의 타겟 프레임 속도를 선택합니다. 일반적으로 모바일 프로젝트에서는 배터리 수명 및 서멀 스로틀링과 유동적인 프레임 속도 간의 균형을 유지해야 합니다. 60FPS로 게임을 구동하고 프레임 시간의 대부분이 CPU와 GPU에 부하가 걸린 상태가 지속되면 배터리 수명이 짧아지고 기기 자체에 의해 CPU 및 GPU 클럭 속도가 더 빠르게 조절됩니다. 대부분의 프로젝트는 30FPS를 목표로 하는 것이 가장 좋습니다. Application.targetFrameRate 프로퍼티를 통해 런타임 동안 프레임 속도를 동적으로 조정하는 방법도 고려할 수 있습니다.

Unity 2019.3 베타 버전부터 제공되는 새로운 기능을 사용하면 입력 시스템과 같은 기타 시스템에 영향을 주지 않고 렌더링 빈도를 줄일 수 있습니다.

로직이나 애니메이션의 스크립트를 작성할 때 프레임 속도를 간과하는 경우가 자주 발생합니다. 업데이트를 상수로 처리하는 대신 Update 메서드에서 Time.deltaTime을 사용하고 FixedUpdate 메서드에서 Time.fixedDeltaTime을 사용합니다.

동기식 로딩 지양

씬이나 에셋을 로드할 때 스파이크가 발생하는 경우가 자주 있는데, 이는 로드가 Unity 메인 스레드에서 동기적으로 수행되기 때문입니다. 따라서 비동기 로딩을 사용하여 스파이크를 최소화하고 안정적인 타이틀을 제작하세요. AssetBundle.LoadFromFile() 및 AssetBundle.LoadAsset() 대신 AssetBundle.LoadFromFileAsync() 및 AssetBundle.LoadAssetAsync() API를 사용합니다.

프로젝트를 비동기식으로 설정하면 다른 측면에서도 도움이 됩니다. 사용자 상호 작용이 항상 유동적으로 유지되며, 씬 및 에셋 로드와 서버 인증 및 교환이 쉽게 병렬로 진행될 수 있으므로 전체적인 시작 및 로드 시간이 단축됩니다. 완전 비동기식 설계를 적용하면 향후 새로운 어드레서블 에셋 시스템(Addressable Asset System)으로 더 손쉽게 마이그레이션할 수 있습니다.

사전 할당된 오브젝트 풀 사용

오브젝트가 자주 인스턴스화되고 파괴되는 경우(예: 총알 또는 NPC) 미리 할당된 오브젝트 풀을 사용하여 오브젝트를 재활용하고 재사용합니다. 오브젝트를 재사용할 때마다 특정 구성 요소를 재설정하는 데 약간의 CPU 리소스가 소요될 수 있지만, 전체를 인스턴스화하는 것보다는 적은 리소스가 사용됩니다. 또한 이 접근 방식을 사용하면 프로젝트의 관리되는 할당 수가 크게 줄어듭니다.

© 2019 Unity Technologies

표준 동작 메서드 사용 최소화

모든 커스텀 동작은 Update(), Awake(), Start() 등의 메서드를 정의하는 추상 클래스에서 상속됩니다. 특정 동작이 이러한 메서드를 요구하지 않는 경우(전체 리스트는 Unity 기술 자료에서 확인 가능) 비워두지 말고 삭제하세요. 그렇지 않으면 Unity에서 빈 메서드를 계속 호출합니다. 이러한 메서드는 네이티브 코드(C++)에서 관리되는 코드(C#)를 호출하며 리소스를 사용해 약간의 CPU 오버헤드를 유발합니다.

이러한 현상은 Update() 메서드에서 특히 문제가 될 수 있습니다. Update() 메서드를 포함하는 오브젝트가 많은 경우 CPU 소모량이 상당히 증가할 수 있습니다. 하나 이상의 관리자 클래스가 Update() 메서드를 구현하지만 모든 개별 오브젝트를 업데이트하는 '관리자 패턴(manager pattern)'을 사용해 보세요. 그러면 네이티브 코드와 관리되는 코드 간의 전환 횟수를 크게 줄일 수 있습니다. 이 블로그 포스팅에서 자세한 내용을 확인할 수 있습니다.

모든 프레임마다 프로젝트에 포함된 여러 시스템을 업데이트할지도 검토합니다. 시스템이 비교적 덜 자주 실행되는 경우가 많아 차례로 호출할 수 있습니다. 예를 들면 2개의 시스템을 1프레임마다 교대로 업데이트할 수 있습니다.

또한 시스템에서 많은 항목을 처리하지만 여러 프레임에 로드를 분산시켜 프레임당 일정 수의 항목만 처리하는 타임 슬라이싱(time-slicing) 방식을 사용하는 것도 좋습니다. 이 방식은 최대 CPU 로드를 낮추는 데도 도움이 됩니다.

여기서 더 발전된 형태는 포괄적인 '할당된 업데이트 관리자'를 구현하는 것입니다. 이때 각 프로젝트의 시스템에는 최대 프레임당 시간 예산이 할당되고, 각 시스템은 할당된 시간 내에 최대한 많은 작업을 수행하는 표준 인터페이스를 기반으로 관리자를 구현합니다. 이 접근 방식은 프로젝트 수명에 걸쳐 최대 CPU 로드를 관리하는 데 많은 도움이 되며, 이 패턴대로 디자인된 시스템은 더 유연하게 사용할 수 있습니다.

많은 리소스가 필요한 API 결과 캐싱

데이터를 가능한 한 많이 캐싱합니다. 일반적으로 GameObject.Find(), GameObject.GetComponent() 및 Camera.main 액세스와 같은 API를 호출하면 리소스가 많이 소모될 수 있으므로 Update() 메서드에서 호출하지 않습니다. 대신 Start()에서 호출하고 결과를 캐싱하면 됩니다.

© 2019 Unity Technologies

런타임 도중 문자열 작업 지양

런타임 중에는 연결과 같은 다수의 문자열 작업을 피합니다. 관리되는 할당이 다수 생성되어 가비지 컬렉션이 급증할 수 있습니다. 플레이어 점수와 같은 문자열은 업데이트 때마다 재생성하지 말고 실제로 변경될 때만 재생성합니다. 또한 StringBuilder 클래스를 사용하면 할당 수를 상당히 줄일 수 있습니다.

의도치 않은 디버그 로깅 지양

의도치 않은 디버그 로깅은 프로젝트를 불안정하게 만들 수 있습니다. Debug.Log()와 같은 API는 개발자의 예상을 벗어나 비개발 빌드에서도 로깅을 계속할 수 있습니다. 이러한 문제를 방지하려면 Debug.Log()와 같은 API를 호출할 때 메서드에서 Conditional 속성을 사용해 고유한 클래스로 래핑합니다. 예를 들면 **[Conditional("ENABLE_LOGS")]**와 같은 형태를 사용해 볼 수 있습니다. Conditional 속성에 사용된 정의가 없으면 메서드와 모든 호출 사이트가 제거됩니다.

중요한 경로에 LINQ 쿼리 사용 지양

LINQ 쿼리는 기능이 강력하고 사용이 편리해 매우 유용하지만, 정기 업데이트 등 중요한 경로에서는 사용하지 않는 편이 좋습니다. 관리되는 할당이 다수 생성되고 CPU에 부담이 클 수 있습니다. 반드시 사용해야 하는 경우에는 심각한 CPU 스파이크가 발생하지 않는 선에서 레벨 초기화와 같은 상황에 제한적으로 현명하게 활용하세요.

비할당 API 사용

Unity에는 Component.GetComponents()와 같이 관리되는 할당을 생성하는 일부 API가 있습니다. 이와 같은 배열을 반환하는 API는 내부적으로 할당됩니다. 비할당 API로 대체할 수 있으면 항상 비할당 API를 사용하는 편이 좋습니다. 많은 물리 API는 새로운 비할당 방식을 사용합니다. 예를 들어, Physics.RaycastAll() 대신 Physics.RaycastNonAlloc()을 사용하는 방식입니다.

정적 데이터 파싱 지양

프로젝트에는 JSON 또는 XML처럼 읽을 수 있는 포맷으로 저장된 데이터를 처리하는 경우가 많습니다. 서버에서 다운로드한 데이터를 처리하는 경우가 대부분이지만, 임베드된 정적 데이터를 처리하는 경우도 일반적입니다. 이러한 파싱은 속도가 느릴 수 있으며 보통은 관리되는 할당을 다수 생성합니다. 타이틀에 정적 데이터를 빌드할 때는 커스텀 에디터 툴에서 ScriptableObjects를 대신 사용하세요.

참고 항목: 스크립터블 오브젝트로 게임을 설계하는 3가지 방법

© 2019 Unity Technologies

6. 물리

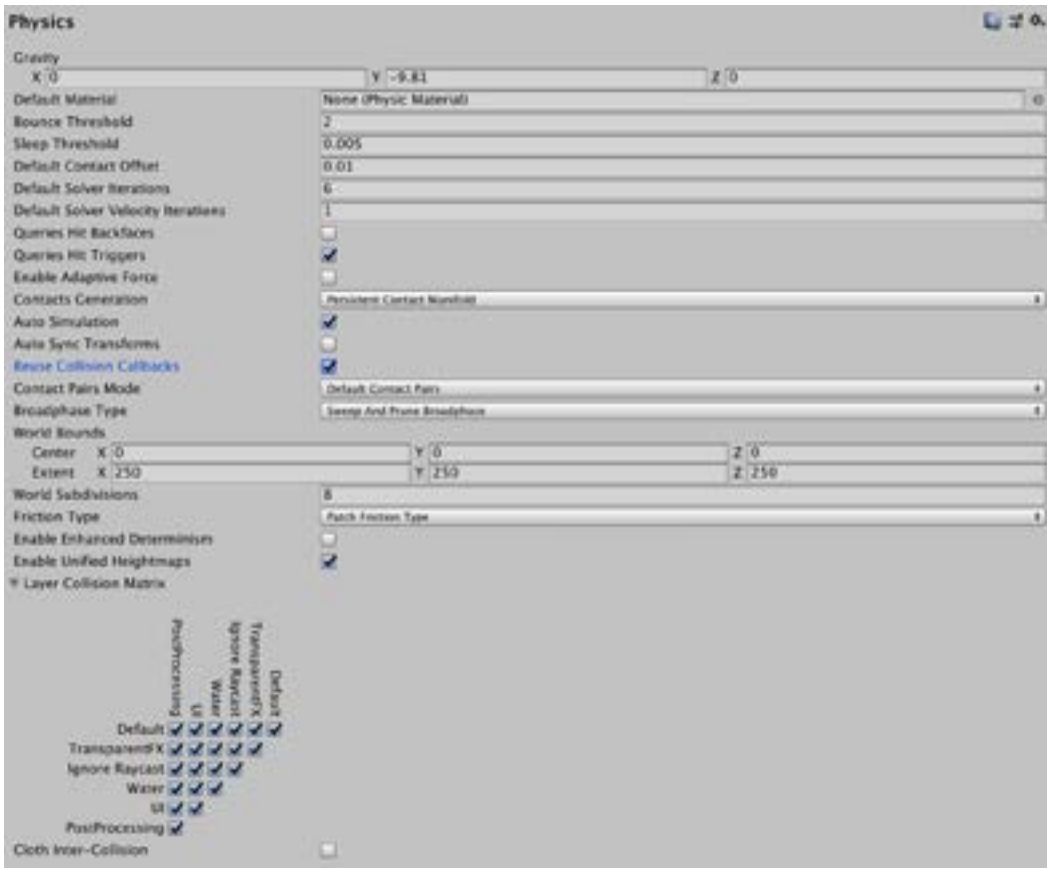
FixedUpdate가 진행되는 동안 프레임마다 물리 처리가 여러 번 이루어질 수 있어 타이틀에서 소모하는 리소스가 많은 것으로 인식되는 경우가 많습니다. 그러나 다른 문제가 성능에 영향을 미치는 것은 아닌지도 확인해 봐야 합니다.

가능한 경우 Player Settings에서 Prebake Collision Meshes 옵션을 활성화하여 빌드 시 런타임 물리 메시 데이터를 생성합니다. 그렇지 않으면 런타임에 에셋이 로드되어 이 데이터가 생성되며, 물리 메시가 많이 포함된 콘텐츠가 생성되는 동안 Unity 메인 스레드에서 CPU 리소스가 많이 소모될 수 있습니다.

메시 콜라이더는 많은 리소스를 소모할 수 있으므로, 복잡한 메시 콜라이더를 하나 이상의 비교적 단순한 기본형으로 적절히 대체하여 원래 형태에 가깝게 구현합니다.

© 2019 Unity Technologies

설정



Unity 에디터에서 물리 설정 구성

Auto Sync Transforms 옵션을 비활성화하고 이전 버전과의 호환성을 유지합니다. 이 옵션은 Unity 2017에 도입된 후 기본 설정으로 활성화되어 있습니다. 이를 통해 트랜스폼의 모든 수정 사항이 기본 물리 오브젝트에 자동으로 즉시 동기화됩니다. 대부분은 물리 시뮬레이션이 수행될 때까지 이 옵션을 비활성화할 필요는 없지만, 프레임에서 CPU 처리 시간이 현저하게 늘어날 수 있습니다. 이 옵션을 비활성화하면 나중에 동기화가 지연되지만 전체 CPU 처리 시간을 줄일 수 있습니다.

충돌 콜백을 사용하는 경우 Reuse Collision Callbacks 옵션을 활성화합니다. 이렇게 하면 각 콜백에 대해 새 콜백을 생성하지 않고 콜백 중에 단일 내부 충돌 오브젝트를 재사용하여 각 콜백에 대한 관리되는 할당이 생성되는 것을 방지할 수 있습니다. 콜백을 사용할 때는 콜백이 진행되는 동안 복잡한 작업을 가급적 지양합니다. 수많은 충돌 이벤트가 포함된 무거운 씬이 더 복잡해질 수 있기 때문입니다. 콜백에 Unity 프로파일러 마커를 추가하면 성능 문제가 발생했을 때 쉽게 추적할 수 있습니다.

마지막으로 레이어 충돌 매트릭스는 필요한 레이어 조합만 사용하는 방식으로 최적화합니다.

© 2019 Unity Technologies

7. 애니메이션

애니메이션은 프로젝트에서 엄청난 리소스를 소모하는 경우가 많은데, 보통은 개발자가 애니메이터를 지나치게 많이 사용하는 것이 원인입니다.

애니메이터는 휴머노이드 캐릭터에 주로 사용하도록 디자인되었지만, 단일 값(예: UI 요소의 알파 채널)을 애니메이션화하는 데에도 자주 사용됩니다. 애니메이터는 상태 머신 플로에 사용하기 편리하지만, 이 용도로 사용하는 것은 비교적 효율이 낮습니다. 내부 벤치마크에 따르면 iPhone 4S와 같은 저사양 기기에서 애니메이터가 레거시 애니메이션보다 성능이 좋은 경우는 약 400개의 곡선을 애니메이션화하는 수준에서입니다.

알파 값이나 크기를 펄싱하는 등 비교적 간단한 경우에는 자체 유틸리티 스크립트를 작성해 더 가볍게 구현하거나 DOTween과 같은 타사 라이브러리를 사용합니다.

마지막으로, 애니메이터는 일반적으로 Unity의 잡 시스템을 사용하여 병렬로 처리하지만 수동으로 업데이트할 경우 메인 스레드에서 처리하게 됩니다. 애니메이터를 수동으로 업데이트할 때에는 이러한 점을 주의하세요.

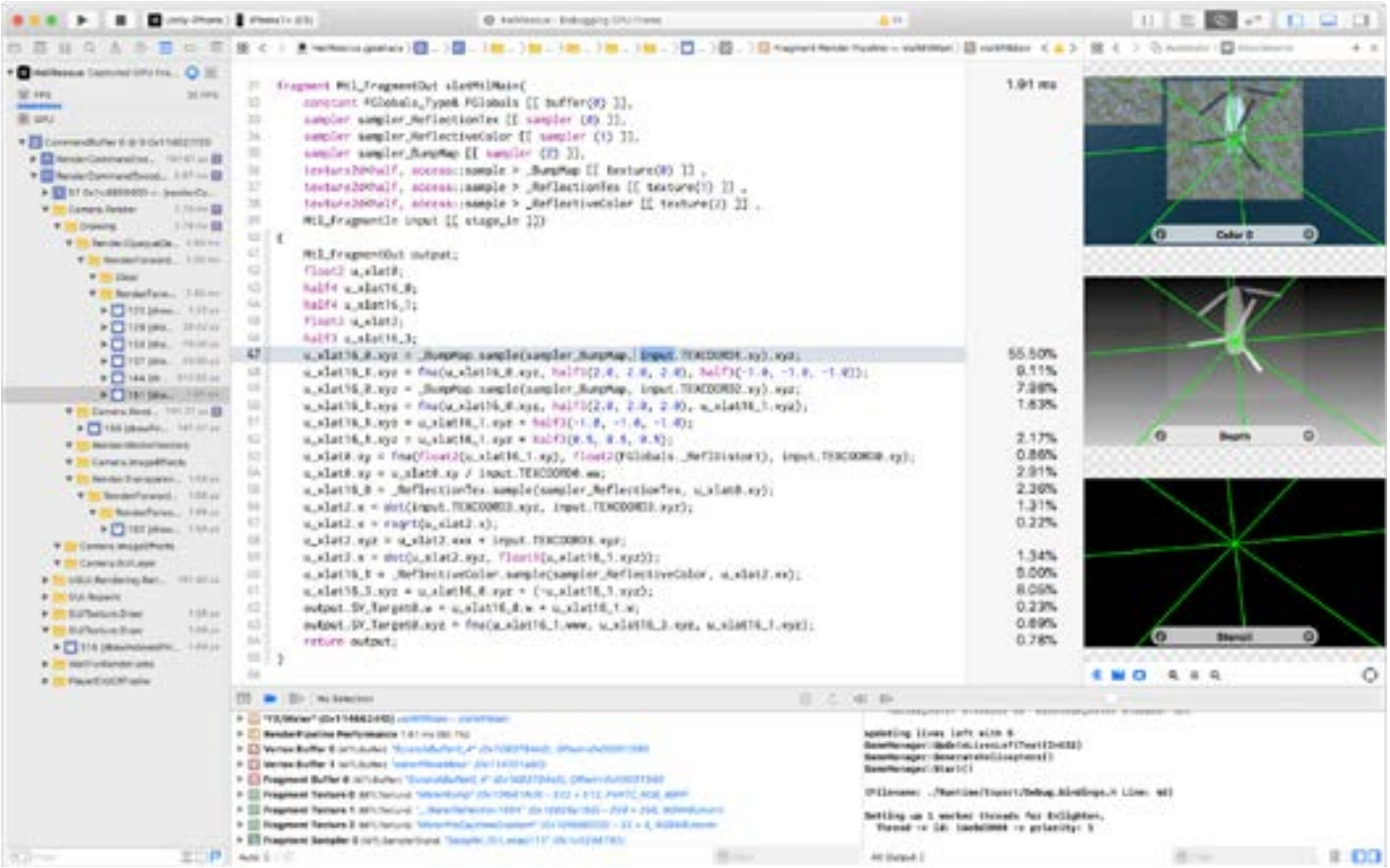
© 2019 Unity Technologies

8. GPU 성능

버텍스, 프래그먼트 및 컴퓨트 셰이더에 GPU 성능이 얼마나 사용되는지 파악하려면 GPU 프로파일링을 수행하는 것이 좋습니다. 그러면 가장 많은 리소스를 소모하는 드로우 콜을 조사하고 리소스 소모가 가장 큰 셰이더를 식별하여 더 효과적으로 최적화할 수 있습니다.

참고 항목:

그래픽스 성능 최적화
성능 최적화를 위한 캐릭터 모델링
셰이더 프로파일링 및 최적화 팁



Xcode의 GPU 캡처 툴을 사용한 iOS 환경 셰이더 디버깅 및 프로파일링

© 2019 Unity Technologies

오버드로우 및 알파 블렌딩에 유의

특히 모바일 플랫폼은 알파 블렌딩 및 오버드로우에 큰 영향을 받습니다. 알파값이 0인 픽셀이 많이 포함된 알파 블렌딩 스프라이트를 여러 레이어로 겹쳐 연출한, 눈에 잘 띄지 않는 큰 오버레이나 이펙트로 인해 GPU 렌더링 시간이 오래 걸리는 상황은 어렵지 않게 찾아볼 수 있습니다.

또한 불필요한 투명 이미지는 가급적 사용하지 말고, 전체 화면 비네트 오버레이와 같이 완전히 투명한 영역이 큰 이미지는 커스텀 메시를 만들어 알파값이 0인 영역을 렌더링하지 않는 편이 좋습니다.

적은 셰이더 배리에이션 사용

모바일의 경우 셰이더를 최대한 단순하게 유지합니다. 스탠다드 셰이더를 사용하는 대신 커스텀 셰이더를 사용하는 편이 가장 가볍게 만들 수 있습니다. 저사양 타겟 기기에서는 단순화된 버전을 사용하거나 효과를 비활성화할 수도 있습니다.

셰이더 배리에이션은 최대한 적게 사용합니다. 셰이더 배리에이션은 성능에 영향을 줄 수 있으며 런타임 메모리 사용량에 상당한 영향을 끼칠 수 있습니다.

과도한 카메라 컴포넌트 사용 지양

Unity 카메라 컴포넌트는 렌더링에 필요한 것보다 많이 사용하지 않도록 합니다. 예를 들어, 여러 카메라를 사용하여 UI 레이어를 구성하는 프로젝트를 쉽게 찾아볼 수 있습니다. 모든 카메라 컴포넌트는 실제로 유용하게 사용되지 않더라도 오버헤드를 유발합니다. 고사양 타겟 플랫폼에서는 무시할 수 있는 수준이지만, 저사양 또는 모바일 플랫폼에서는 각각 최대 1ms의 CPU 처리 시간이 소요될 수 있습니다.

정적 배칭 및 동적 배칭

동일한 머티리얼을 공유하는 환경 메시에서 정적 배칭을 활성화합니다. 이를 통해 Unity는 드로우 콜과 상태 변경 렌더링을 줄이면서도 오브젝트 컬링의 장점은 유지하는 방식으로 병합할 수 있습니다.

동적 배칭이 프로젝트에 항상 유용한 것만은 아니므로 프로파일링을 통해 동적 배칭이 도움이 되는지 확인해야 합니다. 동적 배칭을 수행하려면 오브젝트가 유사해야 하고, 비교적 간단하지만 어느 정도 엄격한 기준을 충족해야 합니다. Unity의 프레임 디버거를 사용하면 특정 오브젝트가 배칭되지 않은 이유를 확인할 수 있습니다.

포워드 렌더링 및 LOD

포워드 렌더링을 사용할 때 너무 많은 동적 조명을 사용하지 않도록 합니다. 동적 조명은 조명의 영향을 받는 모든 오브젝트에 새로운 렌더 패스를 추가합니다.

가능한 경우 디테일 수준(LOD)을 사용합니다. 오브젝트가 멀리 있을수록 간단한 머티리얼과 셰이더로 구성된 메시를 사용해 GPU 성능을 크게 향상할 수 있습니다.

© 2019 Unity Technologies

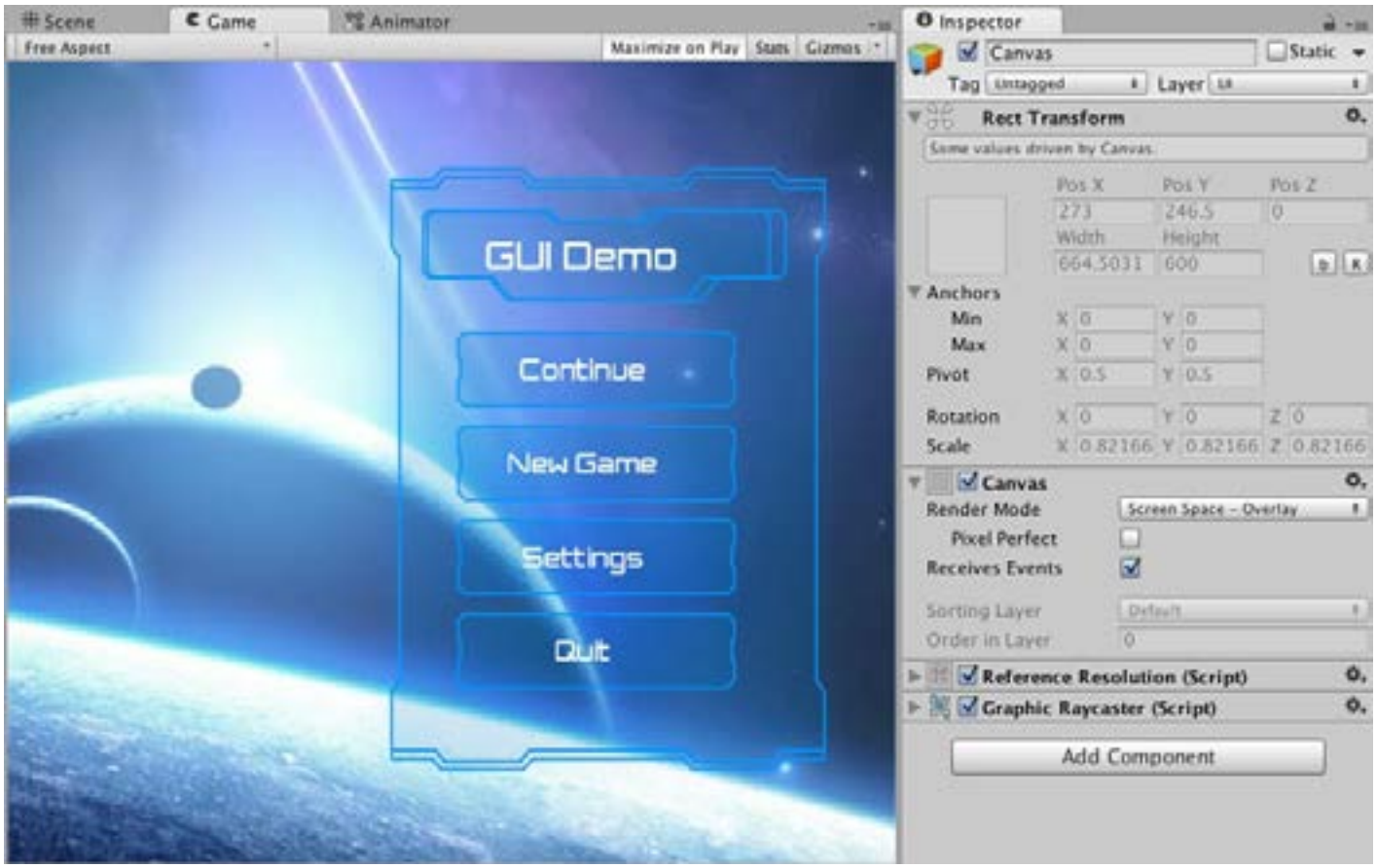
9. 사용자 인터페이스(UI)

Unity UI(UGUI)가 프로젝트 성능 문제를 유발하는 경우도 많습니다.
Unity UI 최적화 팁에서 자세한 내용을 알아보세요.

다양한 해상도 및 종횡비

Unity UI를 사용하면 다양한 화면 해상도와 종횡비에 맞춰 UI의 위치와 크기를 쉽게 조정할 수 있습니다. 그러나 하나의 레이아웃/디자인이 다른 기기에서 제대로 작동하지 않는 경우가 있을 수 있으므로, 모든 기기에서 최상의 경험을 제공하기 위해 모든 UI나 일부 UI를 여러 버전으로 만들어 두는 편이 좋습니다.

지원되는 기기에서 UI를 다양하게 테스트하여 모든 기기에서 일관된 사용자 경험을 제공하도록 합니다.



Unity UI 캔버스 설정

© 2019 Unity Technologies

소수의 캔버스 사용 지양

모든 UI 콘텐츠를 단일 또는 소수의 '일체형' 캔버스에 배치하지 않는 것이 좋습니다. 각 캔버스는 모든 콘텐츠의 메시를 유지하며, 한 가지 요소라도 변경되면 이 메시가 다시 만들어집니다.

콘텐츠마다 다른 캔버스에 배치합니다. 가급적이면 업데이트 빈도에 따라 배치하는 것이 좋습니다. 동적 구성 요소를 정적 요소와 분리하면 정적 메시 데이터를 계속해서 다시 만드는 데 CPU 리소스가 사용되는 것을 방지할 수 있습니다.

레이아웃 그룹에 유의

레이아웃 그룹도 중첩된 경우에는 성능 문제를 일으키는 일반적인 요인 중 하나입니다. 레이아웃 그룹 내 UI 그래픽 컴포넌트가 변경되면(예: ScrollRect 이동) UGUI가 씬 계층에서 최종적인 부모 그룹을 찾을 때까지 부모 레이아웃 그룹을 재귀적으로 검색합니다. 모든 하위 요소들의 레이아웃이 추후에 다시 만들어집니다.

가급적 레이아웃 그룹을 사용하지 않습니다. 콘텐츠가 동적이지 않을 경우에는 더욱 지양해야 합니다. 레이아웃 그룹이 콘텐츠의 초기 레이아웃을 수행하는 데만 사용되고 추후에 변경되지 않는 경우, 콘텐츠 초기화 이후에 Layout Group 컴포넌트를 비활성화하는 커스텀 코드를 추가해 보세요.

많은 리소스를 소요하는 List 및 Grid 뷰

List 및 Grid 뷰는 또 다른 일반적인 UI 패턴입니다(예: 인벤토리 또는 샵 화면). 수백 개의 항목이 존재하지만 한 번에 소수의 항목만이 표시되는 경우, 매우 많은 리소스가 소요될 수 있으므로 모든 항목에 대한 UI 요소를 만들지 않도록 합니다. 그 대신 구성 요소를 재사용하는 패턴을 적용하고, 한 쪽에서는 항목을 가져오는 동시에 다른 항목은 이동하는 방식을 사용합니다. Unity 엔지니어가 제공한 GitHub 프로젝트 예시를 참고하세요.

여러 요소가 겹치지 않도록 유의

많은 요소가 겹친 여러 영역이 있는 UI를 흔히 볼 수 있는데, 카드 대전 게임의 카드 프리팹이 그러한 예에 해당합니다. 이러한 접근 방식으로 디자인을 유연하게 커스터마이즈할 수는 있으나, 많은 픽셀 오버드로우가 발생하여 성능에 큰 영향을 미칠 수 있습니다. 또한 드로우 배치가 추가로 발생할 수 있습니다. 겹쳐 있는 여러 요소를 더 줄이거나 하나의 요소로 병합할 수 있는지 확인하세요.

Mask 및 RectMask2D 컴포넌트의 사용

Mask 및 RectMask2D 컴포넌트는 UI에서 흔히 볼 수 있습니다. Mask는 렌더 타겟의 스텐실 버퍼를 사용하여 픽셀을 그리거나 거부하며, GPU에서 거의 모든 처리를 담당합니다. 반면 RectMask2D는 CPU에서 경계 검사를 수행하여 마스크 외부의 요소를 거부합니다. RectMask2D 컴포넌트가 많은 복잡한 UI는 경계 검사를 수행하는 데 상당한 CPU 리소스를 사용할 수 있으며, 중첩된 경우에는 더 많은 리소스를 사용합니다. RectMask2D 컴포넌트를 너무 많이 사용하지 않도록 주의하고, GPU 로드가 CPU 로드보다 작은 경우 전체 로드 균형을 맞추기 위해 Mask 컴포넌트로 전환하는 것도 생각해 봅니다.

© 2019 Unity Technologies

UI 텍스처를 아틀라스 처리하여 배칭 개선

배칭을 개선하기 위해 UI 텍스처를 최대한 아틀라스 처리합니다. 텍스처의 논리 그룹에 많은 아틀라스를 사용하는 것이 맞지만, 아틀라스의 크기가 지나치게 크거나 밀도가 낮아지지 않도록 유의합니다. 메모리가 낭비되는 주된 원인이 될 수 있습니다.

또한 가능한 경우 아틀라스를 압축해야 합니다. UI 텍스처를 압축하면 열화 문제가 발생할 수도 있지만, 이는 콘텐츠 자체의 특성에 따라 달라집니다.

대부분의 경우 UI 텍스처에는 밉맵이 필요하지 않으므로, 특별히 요구되지 않는 한(예: 월드 공간 UI) 이 임포트 설정을 비활성화합니다.

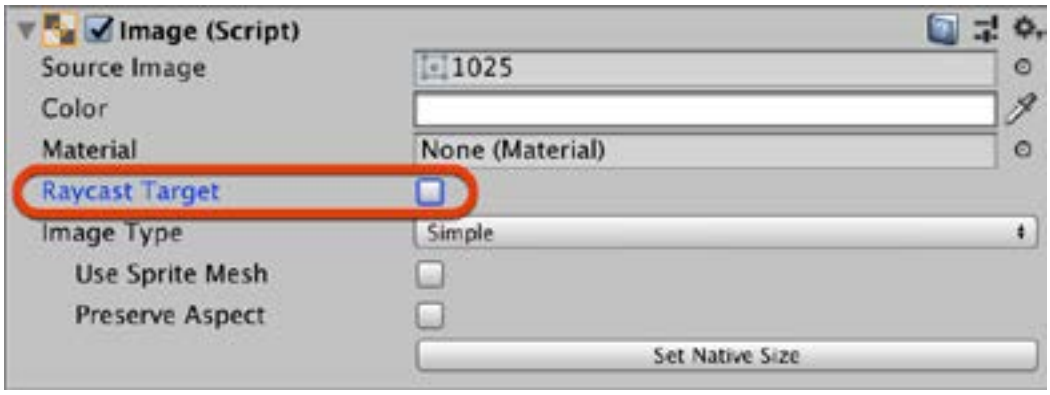
새 UI 창이나 화면을 추가할 때 유의할 사항

프로젝트에 새로운 UI 창이나 화면이 도입될 때 글리치가 발생하는 경우가 많습니다. 그 이유에는 여러 가지가 있지만, 에셋을 온디맨드로 로딩하고 UI 컴포넌트에 담긴 복잡한 계층 구조를 인스턴스화하는 데 소모되는 순수 리소스 등을 예로 들 수 있습니다.

UI 자체의 복잡도를 낮추는 동시에 비교적 자주 사용하는 UI 컴포넌트는 캐싱하는 편이 좋습니다. 인스턴스를 파괴하고 다시 만드는 작업을 반복하는 대신, 비활성화했다가 다시 활성화합니다.

필요하지 않은 경우 Raycast Target 비활성화

입력 이벤트를 수신할 필요가 없는 UI 그래픽 요소에서 Raycast Target 옵션을 비활성화합니다. 버튼의 텍스트나 상호 작용이 불가능한 이미지 등 많은 UI 요소는 입력 이벤트를 수신할 필요가 없습니다. 그러나 UI 그래픽 컴포넌트에는 기본적으로 Raycast Target 옵션이 활성화되어 있습니다. 복잡한 UI에는 Raycast Target 옵션이 불필요하게 활성화된 요소가 포함되었을 수 있으므로, 이 옵션을 비활성화하면 CPU 소모량을 상당히 절감할 수 있습니다.



입력 이벤트가 필요 없는 요소에서 Raycast Target 비활성화

© 2019 Unity Technologies

추가 팁

지금까지 설명한 베스트 프랙티스와 팁을 통해 프로젝트의 내실을 다지고 게임 디자인, 개발, 테스트, 출시에 필요한 올바른 툴과 워크플로를 마련하는 데 도움이 되기를 바랍니다. 마감을 얼마 남겨두지 않았을 때, 또는 점점 더 많은 도움이 필요해질 때는 유니티에서 공유하는 다른 리소스도 참고하시기 바랍니다.

추가 리소스

유니티 블로그, #unitytips 해시태그, Unity 커뮤니티 포럼, Unity Learn에서 더 많은 최적화 팁, 베스트 프랙티스 및 새로운 소식을 확인할 수 있습니다.

Unity ISS에 문의하기

맞춤 지원이 필요하신가요? 유니티 Integrated Success Services를 이용하세요. ISS는 단순한 지원 패키지보다 더 큰 도움을 드립니다. 전담 DRM(개발사 관리 매니저)과 협업하면 팀의 성장에 도움을 주는 Unity 전문가의 도움을 받아 더욱 견고한 프로젝트를 만들 수 있습니다. DRM은 문제를 사전에 발견하고 프로젝트 출시부터 그 이후까지 모든 과정을 원활하게 진행하는 데 필요한 기술 및 운영 관련 전문 지식을 제공합니다. 문의 양식을 이용해 문의 사항을 전달해주세요.

© 2019 Unity Technologies

unity