Unity®

UNITY 2022 LTS 에디션 전자책

고급 Unity 개발자를 위한 데이터 지향 기술 스택

목차

들어가는 말

이 가이드는 Unity의 DOTS(데이터 지향 기술 스택)에서 얻을 수 있는 성능상의 장점에 관해 설명합니다. DOTS에 포함된 각 패키지 및 기능에 관한 대략적인 개요와 함께, DOD(데이터 지향 디자인)와 관련되었으며 그 영향을 받는 여러 핵심 개념을 다루겠습니다. 자세한 API까지 다루지는 않지만, 가이드 전반에 걸쳐 추가 학습에 도움이 되는 여러 새로운 DOTS 튜토리얼과 학습 리소스 링크가 제공됩니다.

이 전자책의 목표는 Unity 프로젝트에 DOTS의 일부 또는 모든 기능을 활용하면 어떤 장점이 있을지 고려하는 데 도움이 되는 지식을 제공하는 것입니다.

저자 및 도움을 주신 전문가들

이 전자책은 Unity DOTS 엔지니어와 외부 전문가의 도움을 받아 제작되었습니다. 주 저자는 유니티의 시니어 소프트웨어 엔지니어 브라이언 윌입니다. 이외에 다음 전문가분들이 이 가이드 제작에 도움을 주셨습니다.

— 닉 레버, 실시간 3D 및 Unity 교육 담당자

— 스티브 맥그릴, 소프트웨어 엔지니어

— 다니엘 키르케고르 앤더슨, 유니티 소프트웨어 엔지니어

— 로랑 기벗, 유니티 제품 관리 팀 디렉터

성능

숙련된 게임 개발자는 타겟 플랫폼에 성능을 최적화하는 작업이 개발 사이클 전체에 걸쳐 진행된다는 사실을 알고 있습니다. 게임이 고사양 PC에서는 원활하게 실행될 수 있지만, 저사양 모바일 플랫폼도 타게팅하고 있다면 어떨까요? 유독 시간이 소요되는 프레임이 있어 눈에 띄는 성능 문제가 발생하나요? 로딩에 지루할 만큼 긴 시간이 걸리거나, 플레이어가 문을 통과할 때마다 게임이 몇 초 동안 멈추는 일이 발생하나요? 이런 상황이라면 당장 최적의 경험을 제공하지 못할 뿐만 아니라, 더 풍부한 환경 디테일이나 스케일, 메카닉, 캐릭터 및 동작, 물리, 플랫폼과 같은 더 많은 기능을 추가할 수가 없습니다.

문제의 원인은 무엇일까요? 대부분의 프로젝트에서는 렌더링이 문제입니다. 지나치게 큰 텍스처, 너무 복잡한 메시, 큰 비용을 요구하는 셰이더 외에도 배칭, 컬링, LOD(디테일 수준)을 효율적으로 활용하지 못하는 상황 등이 원인입니다.

복잡한 메시 콜라이더를 너무 많이 사용하여 물리 시뮬레이션의 비용이 크게 증가하는 것도 흔히 발생하는 문제입니다. 또는 게임 시뮬레이션 자체가 느릴 수도 있습니다. 특별한 게임을 만들기 위해 작성한 C# 코드가 프레임당 CPU 시간을 지나치게 소모하고 있을 수 있습니다.

그렇다면 빠르게 작동하는 코드, 아니면 최소한 느리지는 않은 게임 코드를 작성하려면 어떻게 해야 할까요?

지난 수십 년 동안은 시간이 흐르는 것만으로 PC 게임 개발자의 이러한 문제가 해결되는 경우가 많았습니다. 1970년대부터 21세기 들어서까지는 무어의 법칙으로 알려진 현상에 따라 CPU 싱글스레드 성능이 일반적으로 몇 년마다 두 배씩 증가하면서, PC 게임의 성능도 라이프사이클에 걸쳐 '저절로' 좋아지고는 했습니다. 그러나 지난 20년 동안은 CPU 싱글스레드 성능이 비교적 크게 향상되지 않았습니다. 대신 CPU 코어의 수가 점점 늘어나는 추세로, 지금은 스마트폰 같은 소형 휴대 기기에도 여러 개의 코어가 탑재됩니다. 또한 고사양 게임 기기와 저사양 게임 기기 간의 격차는 더 커졌고, 몇 년 전 사양의 하드웨어를 사용하는 플레이어층이 큰 규모를 차지합니다. 이제 더 빠른 하드웨어를 기다리는 전략을 기대할 수는 없습니다.

그렇다면 '내 CPU 코드가 애초에 느린 이유'를 따져 봐야 합니다. 흔히 발생하는 문제는 다음과 같습니다.

— **가비지 컬렉션에 의해 발생하는 두드러진 오버헤드와 일시 정지:** 가비지 컬렉터가 애플리케이션의 메모리 할당 및 해제를 관리하는 자동 메모리 관리자의 역할을 하기 때문에 발생합니다. 가비지 컬렉션은 CPU 및 메모리 오버헤드의 발생 원인이 되며, 이로 인해 모든 코드 실행이 수 밀리초 동안 일시 정지하기도 합니다. 사용자는 이러한 정지 현상을 성능 문제나 성가신 끊김 현상으로 인식할 수 있습니다.

— **최적화되지 않은 컴파일러 생성 머신 코드:** 일부 컴파일러는 다른 컴파일러에 비해 덜 최적화된 코드를 생성하므로 플랫폼에 따라 다른 결과가 나올 수 있습니다.

— **비효율적인 CPU 코어 사용:** 요즘에는 저사양 기기에도 멀티코어 CPU가 탑재되지만, 멀티스레드 코드를 작성하는 것은 어렵고 오류가 발생하기 쉽다는 이유로 많은 게임이 대부분의 로직을 메인 스레드에서만 실행합니다.

— **캐시 친화적이지 않은 데이터:** 캐시에서 데이터에 액세스하는 것이 메인 메모리에서 가져오는 것보다 훨씬 빠릅니다. 하지만 시스템 메모리에 액세스하려면 CPU가 수백 CPU 사이클 동안 대기해야 할 수 있으므로, 가능한 한 CPU가 캐시에서 데이터를 읽고 쓰도록 만들어야 합니다.

  그러려면 메모리를 연속적으로 읽고 쓰는 것이 가장 간단하므로, 연속되는 배열에 데이터를 빽빽하게 채우면 가장 캐시 친화적으로 데이터를 저장할 수 있습니다. 반대로 데이터가 메모리에 불연속적으로 분산되어 있으면, CPU에서 요청하는 데이터가 캐시 메모리에 존재하지 않는 캐시 부적중(Cache Miss)이 자주 발생하여 더 느린 메인 메모리에서 가져와야 하므로 비용이 늘어나게 됩니다.

— **캐시 친화적이지 않은 코드:** 코드가 실행될 때, 이미 캐시에 저장되어 있지 않은 코드라면 시스템 메모리에서 로드해야 합니다. 한 가지 전략은 함수를 가능한 한 적은 곳에서 호출하여 시스템 메모리에서 로드해야 하는 빈도를 줄이는 것입니다. 특정 함수를 프레임 내 여러 곳에서 호출하는 대신 단일 루프 내에서만 호출하여 코드의 로딩이 프레임당 최대 한 번만 이루어지도록 하는 방식을 예로 들 수 있습니다.

— **과도하게 추상화된 코드:** 여러 문제 중에서도 추상화는 데이터와 코드 양측의 복잡도를 높이는 경향이 있으며, 이로 인해 앞서 언급한 문제가 악화됩니다. 가비지 컬렉션 없이 할당을 관리하기가 어려워지고, 컴파일러가 효율적으로 최적화를 수행하지 못할 수 있으며, 안전하고 효율적인 멀티스레딩이 어려워지고, 데이터와 코드가 캐시 친화적이지 않게 됩니다. 무엇보다도 추상화는 여러 성능 비용에 영향을 미치는 경우가 많아 코드가 전반적으로 느려지고 최적화할 병목 현상이 명확하게 드러나지 않는 경향이 있습니다.

지금까지 언급한 문제는 모두 Unity 프로젝트에서 흔하게 발생합니다. 각각의 문제를 더 구체적으로 살펴보겠습니다.

- C#에서 가비지 컬렉션이 되지 않는 수동 할당 오브젝트를 생성할 수 있지만, C#과 Unity 프로젝트의 기본 표준은 가비지 컬렉션이 가능한 **C# 클래스 인스턴스**를 사용하는 것입니다. 애초에 가비지 컬렉션이 적용되는 언어를 사용한다는 목적에 반대되기는 하지만, 실제로 Unity 사용자들은 풀링이라는 기법을 활용하여 이 문제를 예전부터 완화해 왔습니다. 오브젝트 풀링의 주요 이점은 사전 할당된 풀에서 오브젝트를 효율적으로 재사용할 수 있어 오브젝트를 자주 생성하고 할당 해제할 필요가 없다는 점입니다.
- Unity 에디터에서 C# 코드는 기본적으로 **Mono 컴파일러** 를 통해 기계어 코드로 컴파일됩니다. 스탠드얼론 빌드의 경우 IL2CPP(C# 중간 언어를 C++로 크로스 컴파일)를 사용하여 더 좋은 결과를 얻을 수 있지만, 빌드가 더 오래 걸리고 모드 지원 이 더 어려워진다는 단점이 있습니다.
- Unity 프로젝트에서는 **모든 코드를 메인 스레드에서 실행**하는 경우가 많은데, Unity를 간편하게 이용할 수 있는 방법 중 하나이기 때문입니다.
    - MonoBehaviour의 Update() 메서드와 같은 Unity 이벤트 함수가 모두 메인 스레드에서 실행됩니다.
    - 대부분의 Unity API가 메인 스레드에서만 안전하게 호출될 수 있습니다.
- 일반적인 Unity 프로젝트 내에서 데이터는 **메모리 전체에 분산된 무작위 오브젝트** 구조를 가지는 경우가 많아 캐시 활용도가 매우 떨어집니다. 이 또한 이 구조가 Unity 이용을 간편하게 만들기 때문입니다.
    - 게임 오브젝트와 그 컴포넌트는 모두 개별적으로 할당되므로 메모리의 서로 다른 위치에 저장되는 경우가 많습니다.
- **일반적인 Unity 프로젝트의 코드는 캐시 친화적이지 않은 경향이 있습니다**.
    - 기존 C# 및 Unity API에서 객체 지향 스타일로 코드를 작성하는 것이 권장되는데, 이는 수많은 작은 메서드와 복잡한 호출 체인으로 이어지는 경향이 있습니다. 데이터 지향 스타일과 달리, 이 방식은 별로 하드웨어 친화적이지 않습니다.
    - 모든 MonoBehaviour의 이벤트 함수가 개별적으로 호출되며, 이러한 호출이 반드시 MonoBehaviour 타입별로 그룹화되지는 않습니다. 예를 들어 1,000개의 **Monster** MonoBehaviour가 있다면 각 Monster는 개별적으로 업데이트되며, 반드시 다른 Monster와 함께 업데이트되지는 않습니다.
- 기존 C#과 많은 Unity API의 객체 지향 스타일은 일반적으로 **과도하게 추상화된 솔루션**으로 이어집니다. 그 결과 비효율적이고 전반적으로 얽혀 있어 분리하기 어려운 코드를 얻게 됩니다.

이러한 문제들에 대한 배경 정보를 소개하기 위해, 이 가이드 끝부분의 부록에서는 다음과 같은 개념을 소개합니다.

- 메모리 할당 및 가비지 컬렉션
- 멀티스레드 프로그래밍
- 메모리 및 CPU 캐시
- 객체 지향 프로그래밍 및 추상화
- 데이터 지향 디자인

DOTS 및 엔티티 컴포넌트 시스템

Unity의 ECS(엔티티 컴포넌트 시스템)는 DOTS 패키지 및 기술의 기반을 이루는 데이터 지향 아키텍처입니다. ECS는 Unity의 메모리 내 데이터와 런타임 프로세스 스케줄링에 높은 수준의 제어와 결정론적 분명성을 제공합니다.

Unity 2022 LTS용 ECS에는 호환되는 두 개의 물리 엔진이 포함됩니다. 하나는 높은 수준의 Netcode 패키지이며, 다른 하나는 URP(유니버설 렌더 파이프라인) 및 HDRP(고해상도 렌더 파이프라인)를 포함한 Unity의 SRP(스크립터블 렌더 파이프라인)에 ECS 데이터를 대규모로 렌더링할 수 있는 렌더링 프레임워크입니다. 게임 오브젝트 데이터와 호환되는 이 프레임워크를 이용하면 Unity 2022 LTS에서 아직 ECS를 기본적으로 지원하지 않는 애니메이션, 내비게이션, 입력 및 터레인 등의 시스템도 활용할 수 있습니다.

이 섹션에서는 DOTS의 기능을 설명하고, 이러한 기능이 이전 섹션에서 설명한 CPU 성능 문제를 방지하는 코드를 작성하는 데 어떻게 도움이 되는지를 집중적으로 살펴보겠습니다.

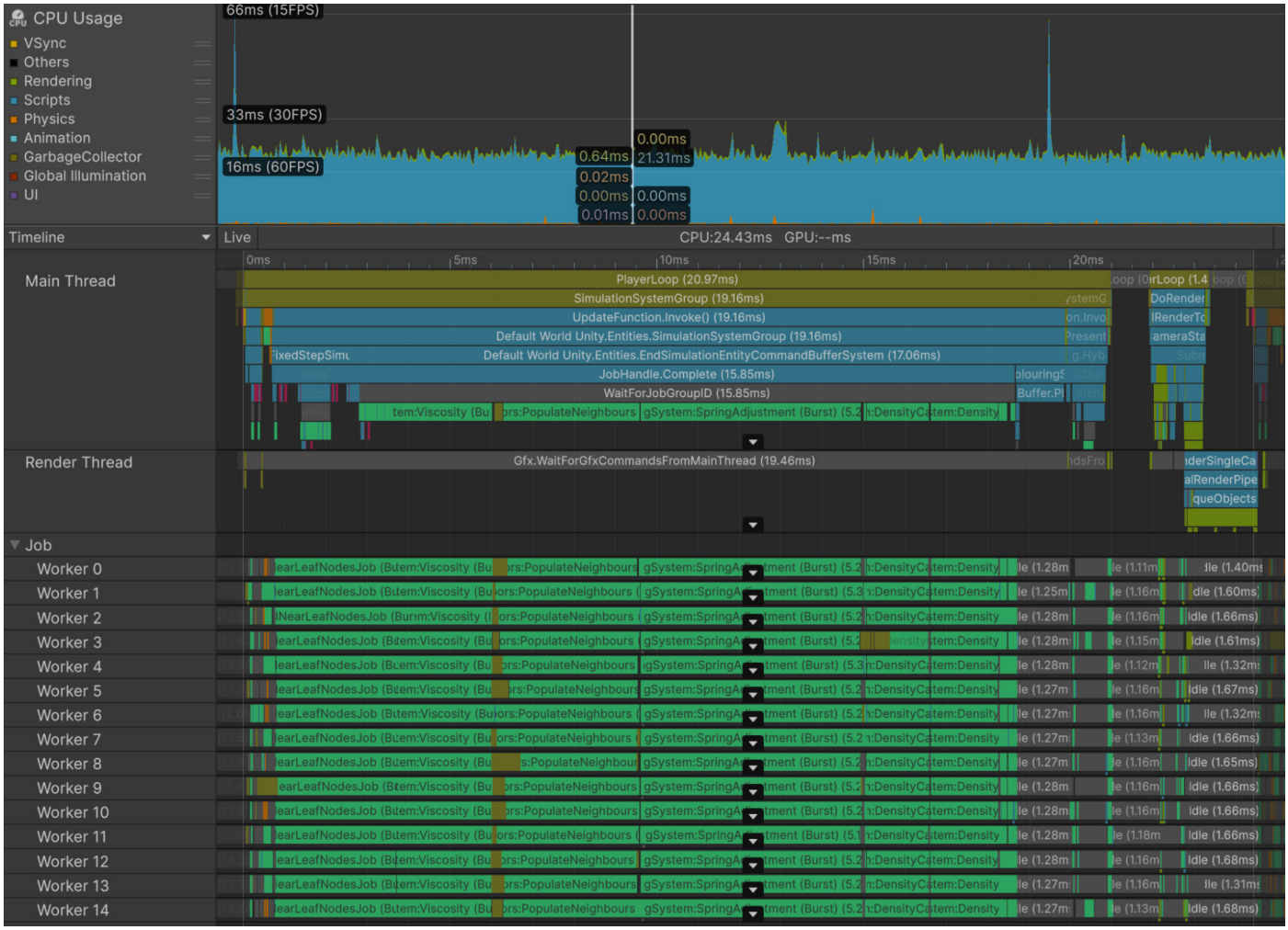
특정 DOTS 패키지에 대한 학습 자료가 필요하다면, 다양한 샘플과 함께 설명 문서와 동영상이 제공되는 EntityComponentSystemSamples Github부터 살펴보는 것이 좋습니다.

하지만 샘플을 살펴보기 전에 우선 DOTS를 구성하는 기능과 패키지부터 살펴보겠습니다.

C# 잡 시스템

C# 잡 시스템은 애플리케이션이 사용 가능한 모든 CPU 코어를 활용할 수 있는 멀티스레드 코드를 간단하고 효율적으로 작성하는 데 필요한 수단을 제공합니다.

다른 DOTS 기능과 달리 잡 시스템은 패키지가 아니며, Unity 핵심 모듈에 포함되어 있습니다.



버스트 컴파일된 잡이 CPU를 최대한 활용하고 여러 워커 스레드에서 실행되는 것을 보여 주는 프로파일

MonoBehaviour 업데이트는 메인 스레드에서만 실행되므로, 많은 Unity 게임이 결국 모든 게임 로직을 하나의 CPU 코어에서만 실행하게 됩니다. 추가 코어를 활용하려면 직접 추가 스레드를 생성하고 관리해야 하지만, 이를 안전하고 효율적으로 수행하기는 상당히 어려울 수 있습니다.

C# 잡 시스템은 Unity가 제공하는 더 간단한 대안입니다.

— 잡 시스템은 타겟 플랫폼의 추가 코어마다 하나씩 늘어나는 워커 스레드(worker thread) 풀을 유지합니다. 예를 들어 8개의 코어에서 실행되는 Unity는 1개의 메인 스레드와 7개의 워커 스레드를 생성합니다.

— 워커 스레드는 잡이라는 단위의 작업을 실행합니다. 워커 스레드는 대기 상태(idle)일 때 잡 대기열에서 실행 가능한 다음 잡을 가져와 실행합니다.

— 워커 스레드에서 일단 실행된 잡은 완료될 때까지 계속 실행됩니다. 중단되는 일이 없다는 의미이기도 합니다.

```
// 두 배열의 요소를 곱하는
// 간단한 예시 잡.
// IJob을 구현하면 구조체가 잡 타입이 됩니다.
struct MyJob : IJob
{
  // NativeArray는 '관리되지 않으므로'
  // 가비지를 만들지 않습니다.
  public NativeArray<float> Input;
  public NativeArray<float> Output;
  // Execute 메서드는
  // 잡 시스템이 이 잡을 실행할 때 호출됩니다.
  public void Execute()
  {
    // Output 배열의 모든 값에
    // Input 배열의 대응하는 값을 곱합니다.
    for (int i = 0; i < Input.Length; i++)
    {
      Output[i] *= Input[i];
    }
  }
```

잡 스케줄링 및 완료

— 잡은 메인 스레드에서만 스케줄링(잡 대기열에 추가)할 수 있으며 다른 잡에서 스케줄링할 수는 없습니다.

— 메인 스레드가 Complete() 메서드를 예약된 잡에서 호출하면, 잡 실행이 완료될 때까지 대기합니다 (잡이 아직 완료되지 않은 경우).

— 오직 메인 스레드만 Complete()를 호출할 수 있습니다.

— Complete() 메서드가 반환되면, 이 잡에서 사용된 데이터를 다시 메인 스레드에서 안전하게 액세스하고 다음 예약된 잡에 안전하게 전달할 수 있다는 사실을 확인할 수 있습니다.

잡 안전성 검사 및 종속성

멀티스레드 프로그래밍에서 스레드 간 안전성을 보장하고 종속성을 관리하는 것은 경쟁 상태, 데이터 손상 및 기타 동시성 문제를 방지하는 데 아주 중요합니다. 다만 이러한 문제에 대한 설명은 본 가이드에서 다루는 내용에 포함되지 않습니다. 여기에서 가장 중요하게 알아야 할 내용은 잡 시스템이 어떻게 안전성 검사와 종속성을 처리하는지 이해하는 것입니다.

— 확실한 분리(isolation) 상태를 보장하기 위해 각 잡에는 메인 스레드나 다른 잡에서 액세스할 수 없는 비공개(private) 데이터가 있습니다.

- 하지만 잡은 다른 잡이나 메인 스레드와 데이터를 공유해야 할 수도 있습니다. 경쟁 상태를 방지해야 하므로, 같은 데이터를 공유하는 잡이 동시에 실행되면 안 됩니다. 따라서 다른 잡과 충돌할 수 있는 잡을 스케줄링하면 잡 시스템의 '안전성 검사'에서 오류가 발생하게 됩니다.
- 잡을 스케줄링할 때, 이전에 스케줄링된 잡에 종속되도록 선언할 수 있습니다. 종속된 모든 잡의 실행이 완료될 때까지 워커 스레드가 잡을 실행하지 않으므로, 충돌할 수 있는 잡을 안전하게 스케줄링할 수 있습니다.
    - 예를 들어 A 잡과 B 잡이 동일한 배열에 액세스한다면 B 잡이 A 잡에 종속되도록 만들 수 있습니다. 이렇게 하면 A 잡이 완료될 때까지 B 잡이 실행되지 않으므로 충돌을 피할 수 있습니다.
- 특정 잡을 완료하면 직간접적으로 종속된 모든 잡 또한 완료됩니다.

많은 Unity 기능이 내부적으로 잡 시스템을 사용하므로 프로파일러의 워커 스레드에서는 사용자가 직접 스케줄링한 잡 이외의 잡도 실행 중일 수 있습니다.

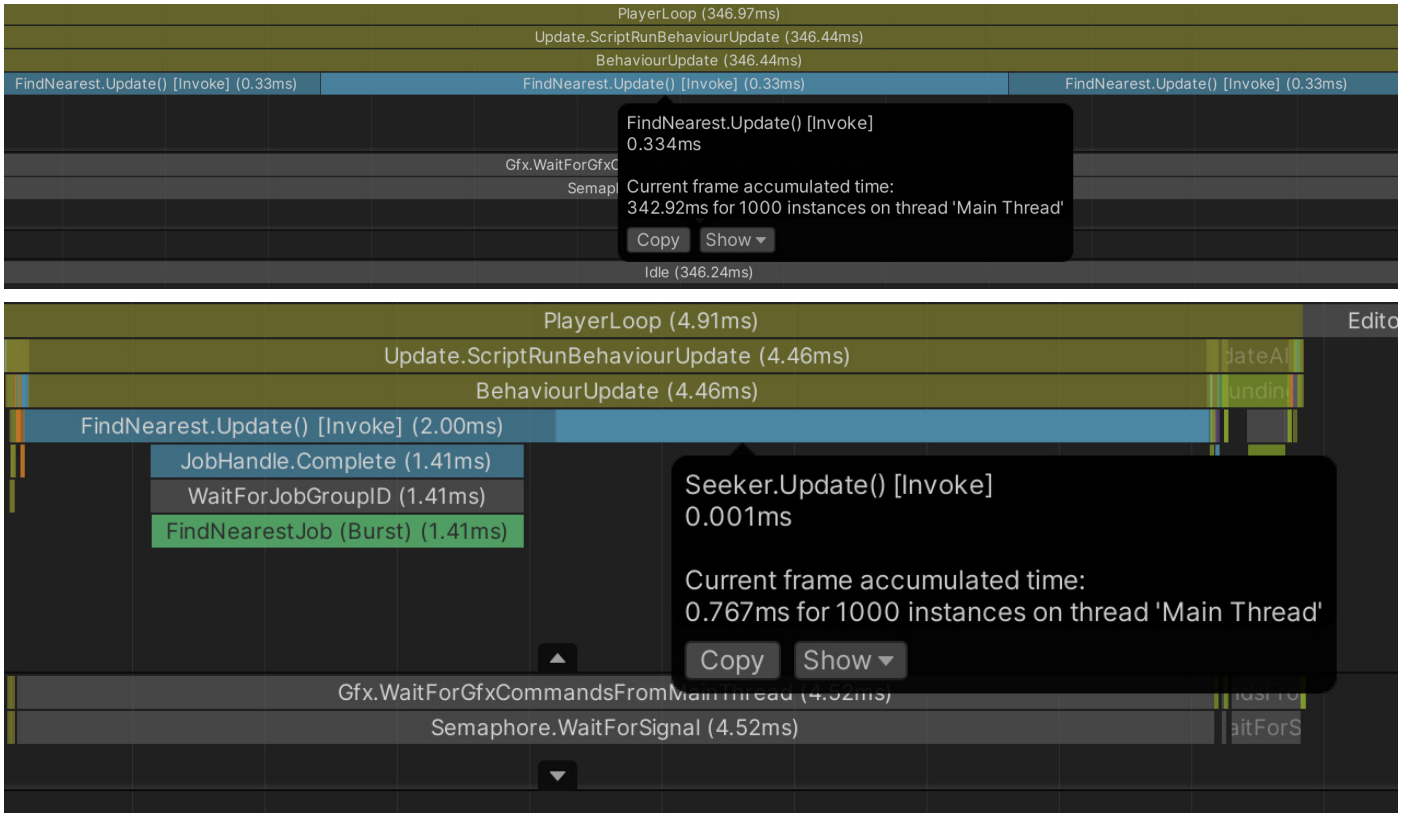
잡은 메모리에서 데이터를 처리하기 위한 용도로만 디자인되었으며, 파일 읽기/쓰기나 연결된 네트워크상 데이터 전송/수신과 같은 I/O(입력 및 출력) 작업을 수행하기 위한 용도가 아니라는 점을 참고하시기 바랍니다. 일부 I/O 작업은 호출하는 스레드를 차단할 수 있으므로, 잡에서 이러한 I/O 작업을 수행하는 것은 CPU 코어를 최대한 활용한다는 목표에 어긋납니다. 멀티스레드 I/O 작업을 수행하려면 메인 스레드에서 비동기 API를 호출하거나 기존의 C# 멀티스레딩 방식을 사용해야 합니다.

잡에 대해 알아보려면 샘플 저장소의 jobs 튜토리얼부터 살펴보세요. Unity Learn 버전의 튜토리얼도 있습니다.

버스트 컴파일러

앞서 설명한 대로 Unity에서 C# 코드는 기본적으로 JIT(just-in-time) 컴파일러인 Mono를 통해 컴파일되지만, 대체로 더 나은 런타임 성능을 보이고 일부 타겟 플랫폼에서 더 잘 지원되는 AOT(ahead-of-time) 컴파일러인 IL2CPP가 대신 사용되기도 합니다.

Burst 패키지가 제공하는 세 번째 컴파일러는 상당한 수준의 최적화를 수행하며, Mono 또는 IL2CPP보다 훨씬 높은 수준의 성능 향상도 기대할 수 있습니다. 다음 이미지에서 볼 수 있듯이, 버스트 컴파일러를 사용하면 많은 계산이 필요한 문제에서 성능과 확장성을 크게 향상할 수 있습니다.



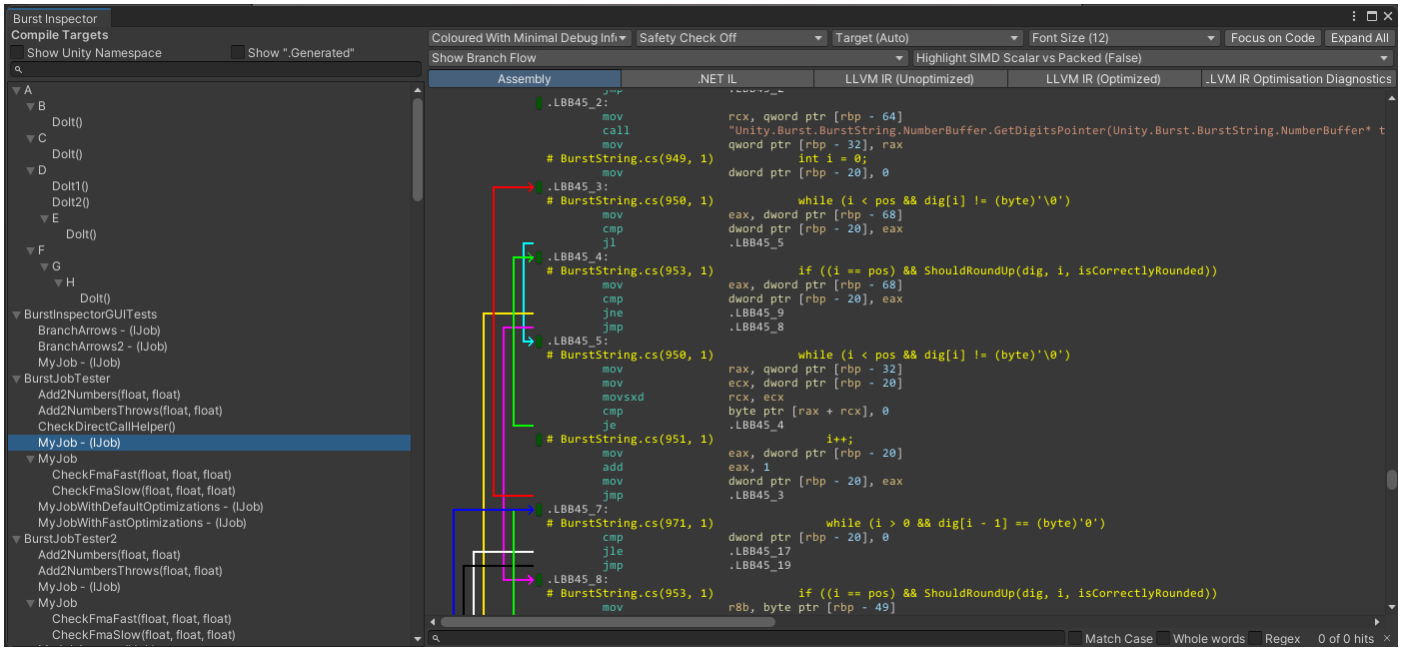
상단 이미지: jobs 튜토리얼의 FindNearest 업데이트, Mono로 컴파일, 소요 시간 342.9ms
하단 이미지: 동일한 jobs 튜토리얼의 FindNearestJob, 버스트 컴파일, 소요 시간 1.4ms

하지만 버스트 컴파일러는 일부 C# 코드만 컴파일할 수 있으며, 수많은 일반적인 C# 코드는 컴파일하지 못합니다. 버스트 컴파일된 코드는 모든 클래스 인스턴스를 비롯하여 관리되는 오브젝트에 액세스할 수 없다는 것이 주된 한계점입니다. 대부분의 기존 C# 코드가 이 기준을 벗어나므로, 버스트 컴파일러는 잡과 같은 코드의 특정 부분만 컴파일할 수 있습니다.

```
// BurstCompile 속성이 이 잡을 버스트 컴파일하도록 지정합니다.
[BurstCompile]
struct MyJob : IJob
{
   public NativeArray<float> Input;
   public NativeArray<float> Output;

   public void Execute()
   {
      for (int i = 0; i < Input.Length; i++)
      {
         Output[i] *= Input[i];
      }
   }
}
```

이 동영상의 설명대로 버스트 컴파일러는 동시에 여러 데이터 요소에 같은 작업을 수행하기 위한 기법인 SIMD와 2개 이상의 포인터 또는 레퍼런스가 같은 메모리 위치를 참조할 때 별칭 지정(aliasing)을 더 잘 탐지하는 등의 기법을 활용하여 성능을 향상합니다.



버스트는 전문가를 위해 인트린직(intrinsics) 및 생성된 어셈블리 코드를 보여 주는 Burst Inspector(위 이미지) 등 몇 가지 고급 기능을 제공합니다.

Collections

Collections 패키지는 관리되지 않는 컬렉션 타입을 제공합니다. 여기에는 잡과 버스트 컴파일된 코드용으로 최적화된 리스트나 해시맵이 포함됩니다.

'관리되지 않는'다는 의미는 C# 런타임 또는 가비지 컬렉터가 이러한 컬렉션을 관리하지 않는다는 것입니다. 관리되지 않는 컬렉션이 더 이상 필요하지 않은 경우 직접 `Dispose()` 메서드를 호출하여 명시적으로 할당 해제해야 합니다.

이러한 컬렉션은 관리되지 않으므로 가비지 컬렉션 부하를 유발하지 않으며 잡과 버스트 컴파일된 코드에서 안전하게 사용할 수 있습니다.

컬렉션 타입은 다음과 같은 몇 가지 카테고리로 분류됩니다.

— 이름이 **Native**로 시작되는 타입은 안전성 검사를 수행합니다. 다음은 안전성 검사에서 오류로 판정되는 경우입니다.

  — 컬렉션이 올바르게 해제되지 않은 경우

  — 컬렉션이 잡에서 스레드 안전(thread-safe)을 보장하지 않는 방식으로 사용되는 경우

— 이름이 **Unsafe**로 시작되는 타입은 안전성 검사를 수행하지 않습니다.

— **Native** 또는 **Unsafe**로 시작하지 않는 나머지 타입은 포인터가 없는 작은 구조체 타입이므로 할당이 이루어지지 않습니다. 따라서 해제할 필요가 없고 스레드 안전 문제가 발생할 위험도 없습니다.

여러 **Native** 타입에는 동등한 **Unsafe** 타입이 있습니다. 예를 들어 **NativeList**와 **UnsafeList**, **NativeHashMap**과 **UnsafeHashMap** 등의 페어가 있습니다. 안전을 위해 가능한 한 **Unsafe** 컬렉션 대신 그에 대응하는 **Native** 컬렉션을 사용하는 것이 좋습니다.

Mathematics

Mathematics 패키지는 C# 수학 라이브러리입니다. Collections와 마찬가지로 버스트 컴파일러와 잡 시스템이 C#/IL 코드를 고도로 효율적인 네이티브 코드로 컴파일할 수 있도록 만들어졌습니다. Mathematics 패키지에서 제공하는 내용은 다음과 같습니다.

— **float3**, **quaternion**, **float3x3** 등의 벡터 및 행렬 타입

— HLSL식 셰이더 규칙을 따르는 다양한 수학 메서드 및 연산자

— 다양한 메서드와 연산자를 위한 특수 버스트 컴파일러 최적화 훅(hook)

자세히 알아보려면 Unity.Mathematics 요약 자료를 참조하세요.

기존 **UnityEngine.Mathf** 라이브러리 타입 및 메서드를 버스트 컴파일된 코드 대부분에 사용할 수 있지만, 그에 대응하는 **Unity.Mathematics**의 타입 및 메서드 성능이 더 좋은 경우도 있습니다.

Entities(ECS)

Entities 패키지는 **엔티티**, 데이터를 나타내는 **컴포넌트**, 코드를 나타내는 **시스템**으로 구성된 아키텍처 패턴인 ECS를 구현합니다.

간단히 말해 엔티티는 컴포넌트로 구성되며, 각 컴포넌트는 보통 C# 구조체입니다. 게임 오브젝트와 마찬가지로 엔티티는 수명이 지속되는 동안 컴포넌트가 추가되거나 제거될 수 있습니다.

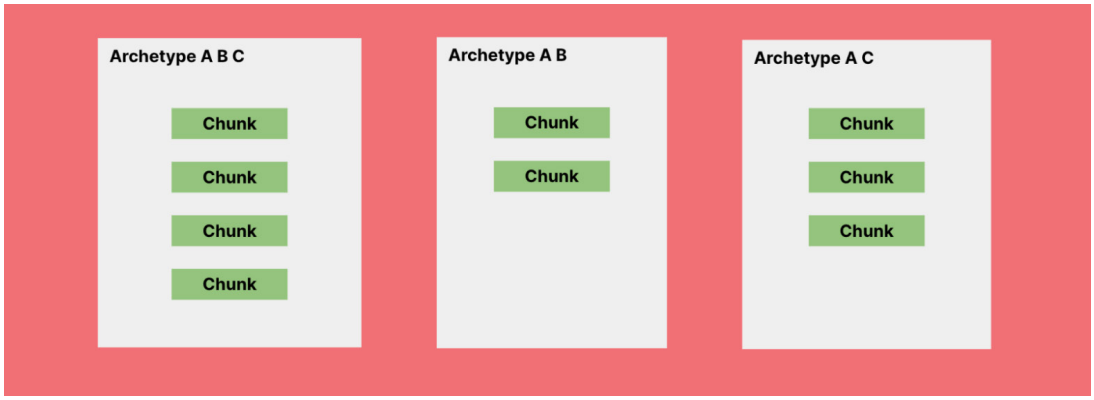
하지만 게임 오브젝트와 다르게 엔티티의 컴포넌트는 보통 자체 메서드를 가지지 않습니다. 대신 ECS의 각 '시스템'에는 대체로 프레임마다 한 번 호출되는 업데이트 메서드가 있으며, 이 업데이트가 일부 엔티티의 컴포넌트를 읽고 수정합니다. 예를 들면 Monster로 구성된 게임에서 MonsterMoveSystem의 업데이트 메서드가 각 Monster 엔티티의 Transform 컴포넌트를 수정할 수 있습니다.

아키타입

Unity의 ECS에서 동일한 컴포넌트 타입 세트로 구성된 모든 엔티티는 같은 '아키타입(archetype)'으로 함께 저장됩니다. 예를 들어 A, B, C라는 세 개의 컴포넌트 타입이 있을 때 컴포넌트 타입의 각 고유 조합은 서로 별개의 아키타입을 나타냅니다.

- A와 B, C 컴포넌트를 가진 모든 엔티티가 하나의 아키타입으로 함께 저장됩니다.
- A와 B 컴포넌트를 가진 모든 엔티티가 두 번째 아키타입으로 함께 저장됩니다.
- A와 C 컴포넌트를 가진 모든 엔티티가 세 번째 아키타입으로 저장됩니다.

엔티티에 컴포넌트를 추가하거나 엔티티에서 컴포넌트를 제거하면 해당 엔티티는 다른 아키타입으로 바뀝니다.



Unity의 ECS에서 동일한 컴포넌트 타입 세트로 구성된 모든 엔티티는 같은 '아키타입(archetype)'으로 함께 저장됩니다.

청크

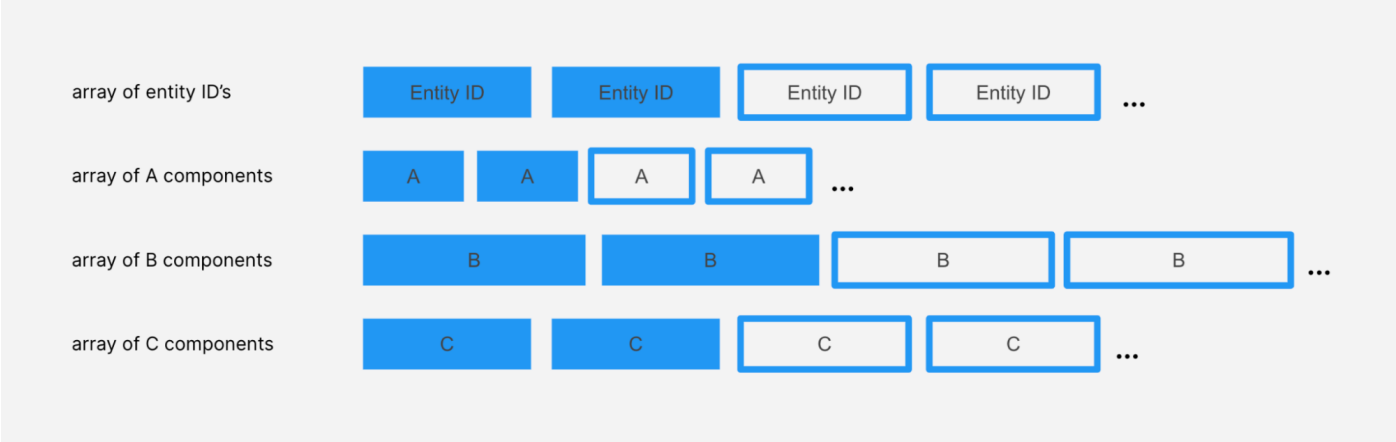
아키타입 내에서 엔티티와 그 컴포넌트는 청크(chunk)라는 메모리 블록으로 저장됩니다. 각 청크에는 최대 128개의 엔티티가 저장되며, 각 타입의 컴포넌트는 청크 내의 고유 배열에 저장됩니다. 예를 들어 A와 B 컴포넌트를 가진 엔티티의 아키타입에서 각 청크는 3개의 배열을 저장합니다.

— 엔티티 ID를 저장하는 첫 번째 배열

— A 컴포넌트를 저장하는 두 번째 배열

— B 컴포넌트를 저장하는 세 번째 배열

청크 내 첫 번째 엔티티의 ID 및 컴포넌트는 각 배열의 0번 인덱스에 저장되며, 두 번째 엔티티는 1번 인덱스, 세 번째 엔티티는 2번 인덱스에 저장됩니다.



Unity ECS 아키텍처에서 청크의 작동 방식

청크의 배열은 항상 빽빽하게 채워집니다.

— 청크에 새 엔티티가 추가되면 배열의 첫 번째 빈 인덱스에 저장됩니다.

— 엔티티가 소멸되거나 다른 아키타입으로 바뀌어 청크에서 엔티티가 제거되면 청크 내 마지막 엔티티가 빈 인덱스를 채웁니다.

쿼리

아키타입 및 청크 기반 데이터 레이아웃의 주요 이점은 엔티티를 효율적으로 쿼리하고 반복 작업(iteration)할 수 있다는 점입니다.

특정 컴포넌트 타입 세트로 구성된 모든 엔티티를 순회하려는 엔티티 쿼리의 경우, 먼저 조건을 만족하는 모든 아키타입을 찾은 후 해당 아키타입의 청크에서 엔티티를 순회합니다.

— 청크 내에서 컴포넌트는 빽빽하게 채워진 배열에 저장되므로 컴포넌트 값을 순회할 때 캐시 부적중을 최소화할 수 있습니다.

— 대부분의 프로그램에서 아키타입 세트는 대체로 변하지 않으므로, 쿼리와 일치하는 아키타입 세트는 보통 캐싱하여 쿼리의 속도를 더 높일 수 있습니다.

```
// 간단한 예시 시스템.
public partial struct MonsterMoveSystem : ISystem
{
  [BurstCompile]
  public void OnUpdate(ref SystemState state)
  {
    // LocalTransform, Velocity 및 Monster 컴포넌트로 구성된
    // 모든 엔티티를 순회하는 쿼리
    foreach (var (transform, velocity) in
      SystemAPI.Query<RefRW<LocalTransform>, RefRO<Velocity>>()
        .WithAll<Monster>())
    {
      // deltaTime을 감안한 속도로
      // 트랜스폼 위치 업데이트
      transform.ValueRW.Position +=
          velocity.ValueRO.Value * SystemAPI.Time.deltaTime;
    }
  }
}
```

잡 시스템 연동

엔티티 컴포넌트 타입이 관리되지 않는 타입이라면 버스트 컴파일된 잡에서 엔티티에 액세스할 수 있습니다. 엔티티 액세스를 위한 두 개의 특별한 잡 타입으로 IJobChunk와 IJobEntity가 있습니다.

```
// IJobEntity를 스케줄링하는 간단한 예시 시스템.
public partial struct MonsterMoveSystem : ISystem
{
    [BurstCompile]
    public void OnUpdate(ref SystemState state)
    {
        //            .
        var job = new MonsterMoveJob {
            DeltaTime = SystemAPI.Time.DeltaTime
        };
        job.ScheduleParallel();
    }
}
// LocalTransform, Velocity 및 Monster 컴포넌트로 구성된
// 모든 엔티티를 처리하는 버스트 컴파일된 잡
[WithAll(typeof(Monster))]
[BurstCompile]
public partial struct MonsterMoveJob : IJobEntity
{
    public float DeltaTime;
  // LocalTransform을 수정하기 위해 'ref' 사용.
  // Velocity만 읽고 싶으므로 'in' 사용.
   public void Execute(ref LocalTransform, in Velocity)
    {
        transform.Position += velocity.Value * DeltaTime;
    }
}
```

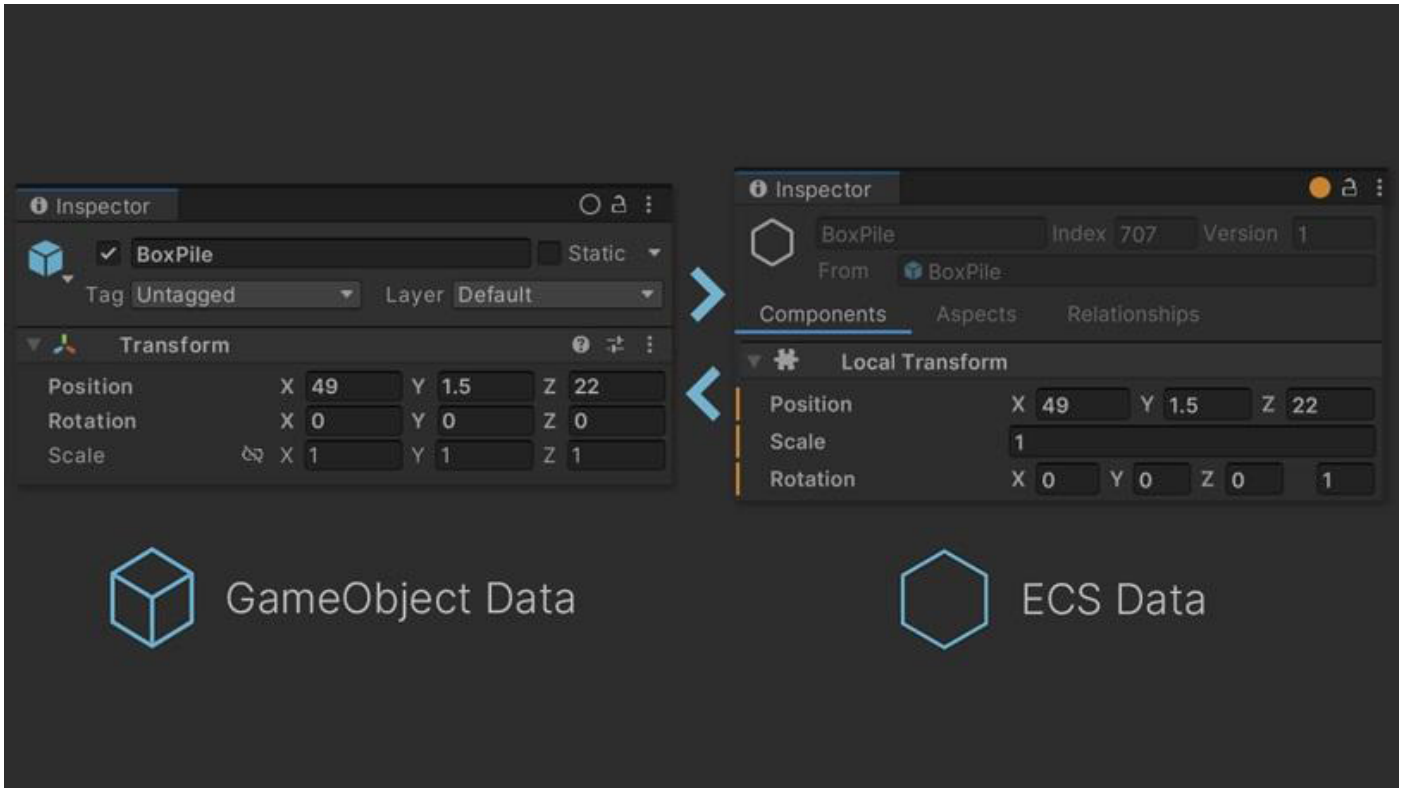
사용하기 쉽도록 시스템이 전반적으로 잡 종속성과 잡 완료를 자동 처리할 수 있습니다.

하위 씬 및 베이킹

Unity ECS는 씬 대신 하위 씬을 사용하여 애플리케이션의 콘텐츠를 관리합니다. Unity의 기본 씬 시스템이 ECS와 호환되지 않기 때문입니다.

엔티티를 Unity 씬에 직접 추가할 수는 없지만, 베이킹이라는 기능을 통해 씬에서 엔티티를 로드하고 게임 오브젝트와 MonoBehaviour 컴포넌트를 엔티티 및 ECS 컴포넌트로 변환할 수 있습니다.

하위 씬은 씬 안에 중첩된 씬으로 생각할 수 있으며, 하위 씬을 편집할 때마다 재실행되는 베이킹을 통해 처리됩니다. 베이킹 과정에서 하위 씬 안에 있는 모든 게임 오브젝트의 엔티티가 생성되고 파일로 직렬화되며, 하위 씬이 로드될 때 런타임에 게임 오브젝트가 아닌 이 엔티티가 로드됩니다.



왼쪽: 게임 오브젝트 인스펙터, 오른쪽: 게임 오브젝트로 베이크된 엔티티 인스펙터

베이크된 엔티티에 추가될 컴포넌트는 게임 오브젝트 컴포넌트와 연관된 '베이커'가 결정합니다. 예를 들어 MeshRenderer 같은 표준 그래픽스 컴포넌트와 연관된 베이커는 엔티티에 그래픽스 관련 컴포넌트를 추가합니다. 자체 MonoBehaviour 타입에서 베이커를 정의하면 베이크된 엔티티에 추가될 컴포넌트를 제어할 수 있습니다.

```
// 체력, 최대 체력, 재충전 시간, 재충전 속도로 구성된
// 에너지 실드를 나타내는 엔티티 컴포넌트 타입
public struct EnergyShield : IComponentData
{
    public int HitPoints;
    public int MaxHitPoints;
    public float RechargeDelay;
    public float RechargeRate;
}
// 간단한 예시 저작 컴포넌트.
// 저작 컴포넌트는 단순히 일반적인 MonoBehaviour이며
// Baker 클래스가 정의되어 있습니다.
public class EnergyShieldAuthoring : MonoBehaviour
{
    public int MaxHitPoints;
    public float RechargeDelay;
    public float RechargeRate;
  // EnergyShield 저작 컴포넌트의 베이커.
  // 이 베이커는 하위 씬의 각 게임 오브젝트에 연결된
  // 모든 EnergyShieldAuthoring 인스턴스마다 한 번씩 실행됩니다.
   class Baker : Baker<EnergyShieldAuthoring>
    {
        public override void Bake(EnergyShieldAuthoring authoring)
        {
     // TransformUsageFlags는
     // 엔티티에 주어질 트랜스폼 컴포넌트를 지정합니다.
     // None 플래그는 트랜스폼이 필요 없음을 의미합니다.
      var entity = GetEntity(TransformUsageFlags.None);
     // 엔티티에 하나의 컴포넌트만 추가하는 간단한 베이커.
      AddComponent(entity, new EnergyShield
             {
                 HitPoints = authoring.MaxHitPoints,
                 MaxHitPoints = authoring.MaxHitPoints,
                 RechargeDelay = authoring.RechargeDelay,
                 RechargeRate = authoring.RechargeRate,
             });
         }
     }
}
```

단순하게 활용하는 경우에 씬에 직접 엔티티를 추가할 수 없는 것은 불편하지만, 보다 고급 단계에서 활용할 때는 이 베이킹 프로세스가 유용할 수 있습니다. 베이킹은 저작(authoring) 데이터(에디터에서 편집하는 게임 오브젝트)와 런타임 데이터(베이크된 엔티티)를 분리하므로 직접 편집하는 내용과 런타임에 로드되는 내용이 일대일로 대응되지 않아도 됩니다. 예를 들어 베이킹 과정에서 데이터를 절차적으로 생성하기 위한 코드를 작성하여 런타임에 발생하는 비용을 줄일 수 있습니다.

스트리밍

규모가 크고 디테일이 많은 환경일수록 플레이어 또는 카메라가 환경 내에서 이동할 때 많은 요소를 효율적이고 비동기식으로 로드하고 언로드할 수 있어야 합니다. 예를 들어 대규모 오픈 월드에서는 시야 안으로 들어오는 많은 요소를 로드하고 시야 밖으로 벗어나는 많은 요소를 언로드해야 합니다. 이러한 기법을 스트리밍이라고도 합니다.

엔티티는 메모리 사용량과 처리 오버헤드가 게임 오브젝트보다 적고 더욱 효율적으로 직렬화 및 역직렬화할 수 있으므로 스트리밍에 훨씬 적합합니다.

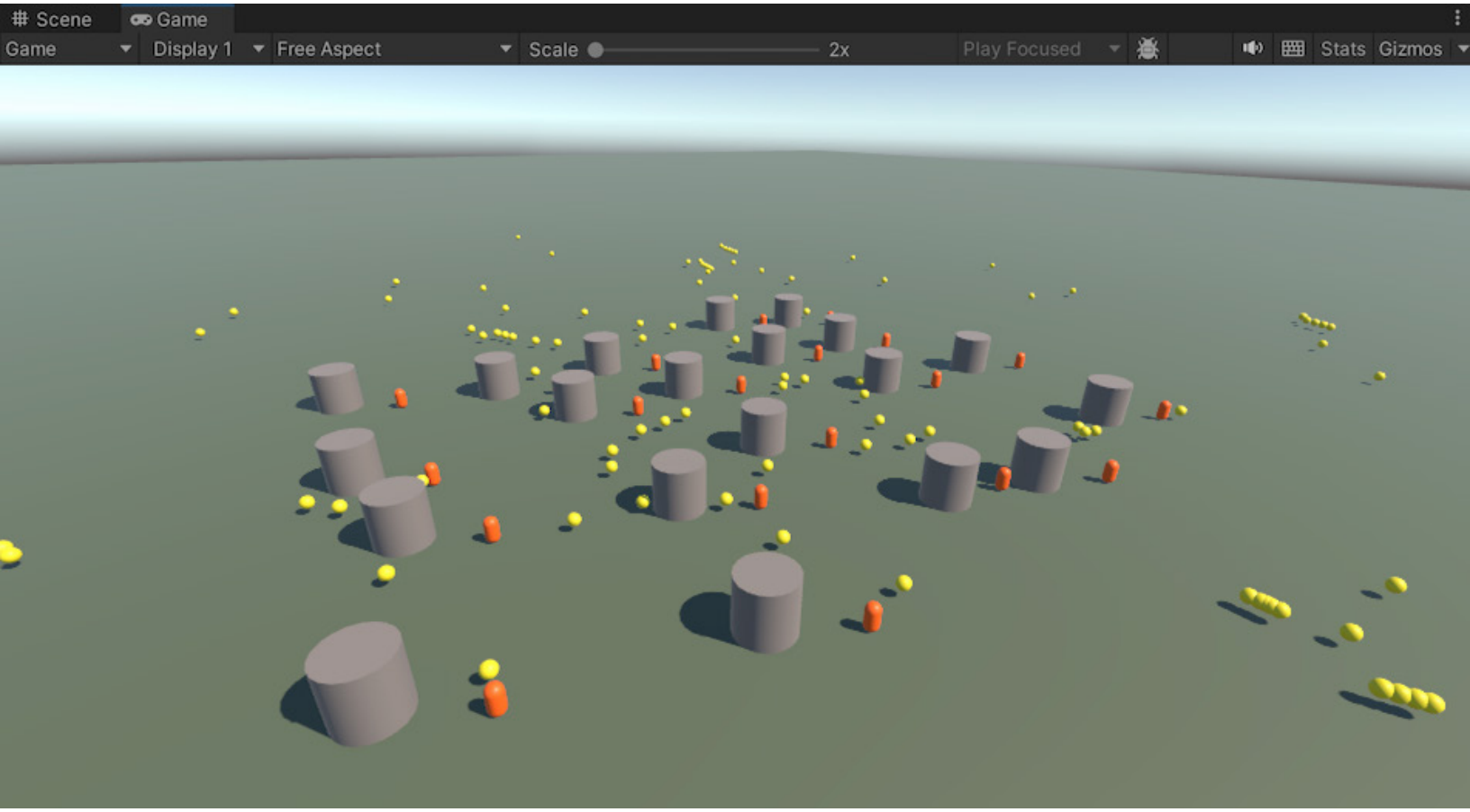
EntityComponentSystemSamples Github

EntityComponentSystemSamples Github 저장소에서는 기초 및 고급 DOTS 기능을 모두 소개하는 다양한 샘플이 제공됩니다. 각 샘플 모음집의 README 파일에서 자세한 내용을 확인할 수 있으며, 이 가이드에서는 몇 가지 샘플을 선정하여 간략하게 설명하겠습니다.

Github 저장소의 일부 샘플은 Unity Learn의 새로운 DOTS 교육 과정인 DOTS 알아보기에서도 자세히 살펴볼 수 있습니다. Unity Learn 튜토리얼에 대한 링크가 제공되는 샘플의 경우 튜토리얼을 살펴보세요.

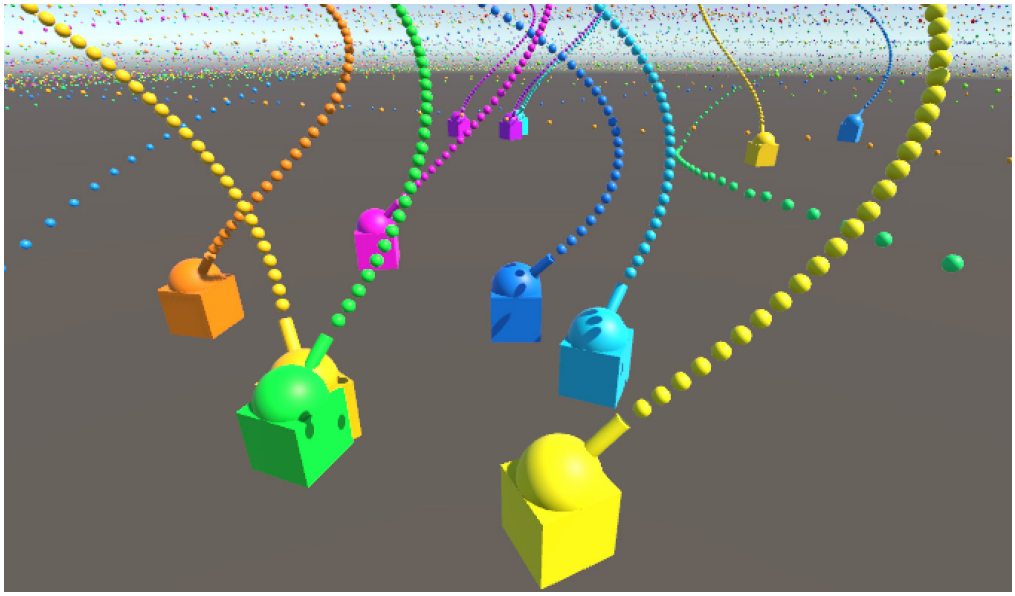
HelloCube 샘플은 DOTS를 시작하는 초보자들에게 적합하며 엔티티 생성 및 소멸, 컴포넌트 추가 및 제거, 시스템을 통한 엔티티 액세스 등 Entities API의 가장 기본적인 개념을 보여 줍니다. HelloCube 가이드 동영상을 통해 샘플의 콘텐츠를 살펴보거나 Unity Learn의 HelloCube 튜토리얼을 단계별로 진행해 보세요.

Baking 샘플은 런타임에 엔티티를 로드할 수 있도록 빌드 시 엔티티 데이터를 직렬화하는 베이킹 프로세스를 보여 줍니다.



Kickball 튜토리얼의 스크린샷

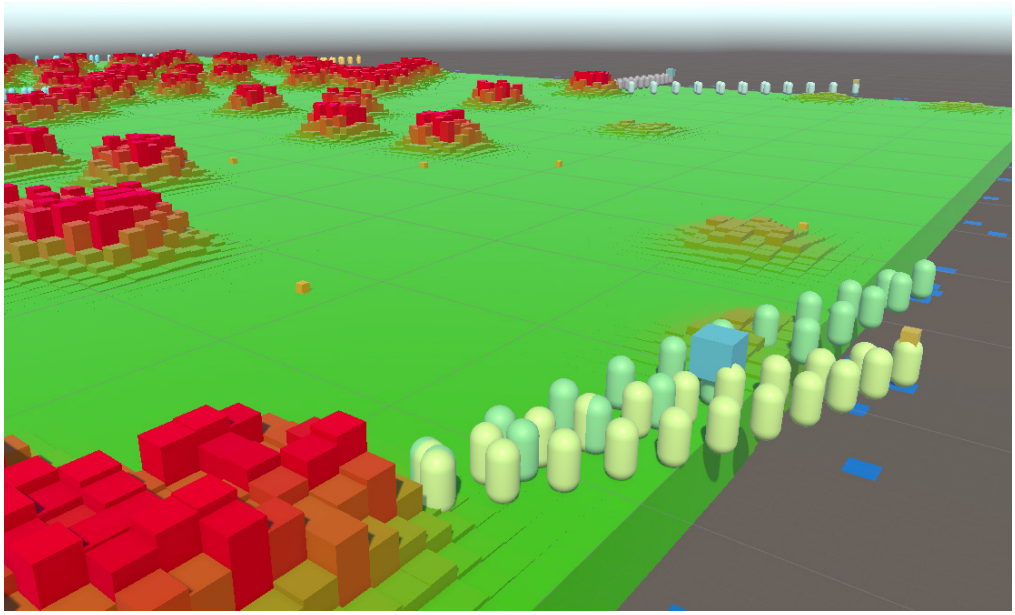
Kickball 샘플은 더 많은 엔티티 기본 기능과 함께 HelloCube보다 조금 더 많은 게임플레이를 보여 줍니다. 플레이어는 컨트롤을 사용해 주황색 캡슐을 움직이고, 노란색 공을 생성하고, 공을 차서 캡슐로부터 멀리 떨어뜨릴 수 있습니다. 회색 원기둥은 캡슐과 공의 움직임을 방해하는 장애물입니다. Kickball 가이드 동영상도 시청해 보세요.



Tanks 튜토리얼의 스크린샷

Tanks 샘플에서는 잡 시스템과 엔티티를 활용하여 평면 위에서 탱크를 움직이고, 포탑을 회전하며, 여러 색상의 발사체를 발사합니다. 목표는 회전하는 포탑에서 포탄을 발사하면서 이동하는 탱크를 생성하는 것입니다. 탱크는 포탄에 맞으면 파괴되며, 플레이어는 탱크 한 대의 움직임을 제어합니다. Tanks 가이드 동영상을 시청하거나 Unity Learn의 Tanks 튜토리얼을 따라 진행해 보세요.

Firefighters 샘플에서는 들판에 불이 번져 봇들이 단체로 양동이를 들고 대열을 맞춰 불을 끕니다. 많은 개념이 결합된 고급 튜토리얼이므로 위 튜토리얼을 먼저 완료하는 편이 좋습니다. 이 프로젝트는 DOTS 부트캠프에서 4개의 세션으로 다루었습니다.

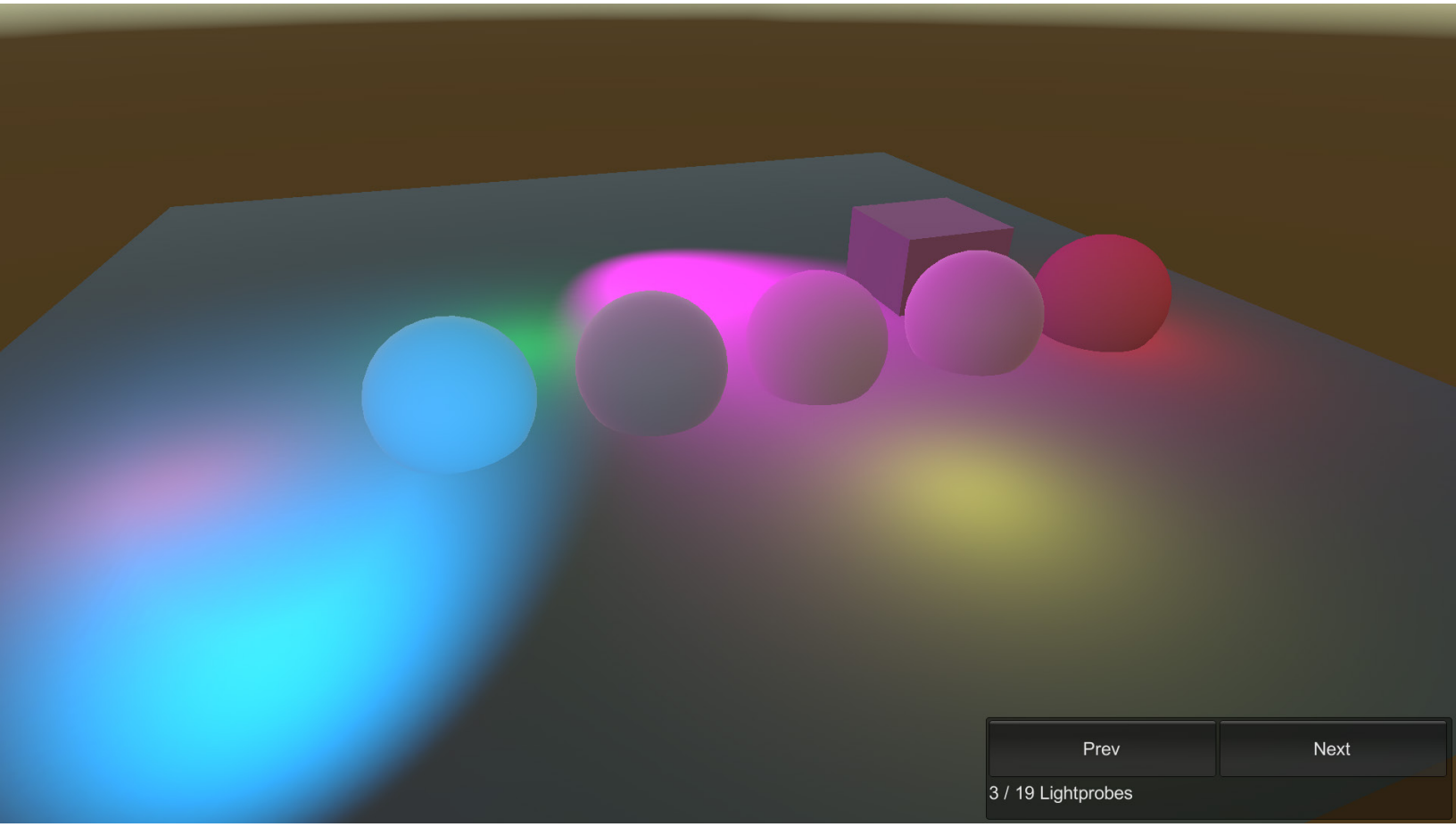


Firefighters 샘플의 씬

Entities Graphics

Entities Graphics 패키지에는 URP(유니버설 렌더 파이프라인) 또는 HDRP(고해상도 렌더 파이프라인)로 엔티티를 렌더링하기 위한 컴포넌트 및 시스템이 제공됩니다. Entities Graphics는 BatchRendererGroup API를 기반으로 제작되었습니다.

Entities Graphics 샘플에서는 라이트 프로브와 라이트맵, 머티리얼 프로퍼티 오버라이드, LOD 등 다양한 그래픽스 기능을 확인할 수 있습니다.



EntityComponentSystemSamples 저장소의 Entites.Graphics 샘플 씬

Physics

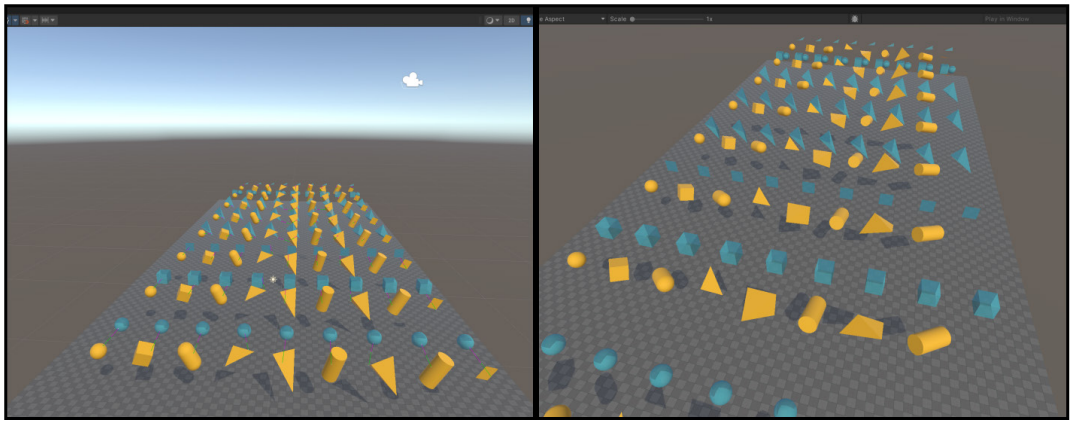
Unity Physics 패키지에서는 리지드바디 시뮬레이션 및 충돌 검사 기능이 제공됩니다.

Unity Physics는 동일한 표면 수준 API를 유지하면서 '백엔드'를 대체하는 기능을 지원하므로 기존 코드 또는 에셋을 수정하지 않으면서 물리 구현을 대체할 수 있습니다.

패키지에서 제공하는 기본 백엔드는 결정론적이므로 초기 조건 및 입력이 동일하다면 산출되는 결과도 같습니다.

Havok Physics 패키지에서는 업계를 선도하는 여러 AAA 게임을 지원하는 전용 Havok Physic 엔진을 기반으로 한 대체 백엔드를 제공합니다.

Physics 샘플에서는 콜라이더, 질량 및 모션 프로퍼티, 머티리얼 프로퍼티, 이벤트, 조인트 및 모터 등 패키지의 다양한 기능을 살펴볼 수 있습니다.



EntityComponentSystemSamples 저장소의 Physics 샘플 씬

Netcode for Entities

Netcode for Entities 패키지는 Unity에서 제공하는 두 가지 넷코드 솔루션 중 하나입니다. Netcode for GameObjects 솔루션과 달리 Netcode for Entities는 권한 서버를 사용하며 클라이언트측 예측(client-side prediction)을 지원하므로 빠른 페이스의 경쟁 게임에 적합합니다.

권한 서버

권한 서버는 게임에서 일어나는 일에 대한 권한을 플레이어 머신에 분산하는 대신, 전체 게임 시뮬레이션을 자체 실행하고 게임에서 일어나는 일을 결정합니다. 클라이언트가 서버에 플레이어 입력을 전송하면 서버가 게임 시뮬레이션을 업데이트한 다음, 게임 상태의 새로운 스냅샷을 클라이언트에 다시 전송합니다. 이는 네트워크 게임 로직을 구현하는 가장 간단한 방법이며 부정 사용자가 취약점을 악용할 위험을 가장 크게 줄이는 방법입니다.

클라이언트측 예측

서버에서 전송한 데이터가 클라이언트에 도달하는 데 시간이 걸리므로 클라이언트는 항상 서버보다 늦은 상태를 수신하게 됩니다. 게임의 많은 요소에 대해 이러한 지연은 대체로 문제가 되지 않지만, 플레이어 캐릭터 같은 일부 요소의 경우에는 지연이 발생할 경우 게임플레이 품질이 저하되고 플레이하기가 어려워질 수 있습니다.

클라이언트측 예측으로 이 문제를 해결할 수 있습니다. 플레이어 캐릭터와 같은 지정된 요소의 경우, 클라이언트는 수 밀리초 후의 상태를 예측하려 시도합니다. 예측이 서버측 상태와 충분히 정확하고 일관되게 일치하는 한 게임에 지연이 없는 것처럼 느낄 수 있습니다.

이러한 핵심 기능 두 가지 외에도 Netcode for Entities는 Netcode for GameObjects보다 확장성이 양호하고 대역폭을 최적화할 수 있는 더 바람직한 수단을 제공합니다.

Netcode for Entities를 시작하려면 Netcode for Entities 샘플 저장소를 살펴보세요. 이 샘플에서 동기화, 연결 플로, Unity Physics 연동 등 다양한 기초 및 고급 기능을 살펴볼 수 있습니다. 다음 내용을 다루는 Networked Cube 튜토리얼부터 살펴보세요.

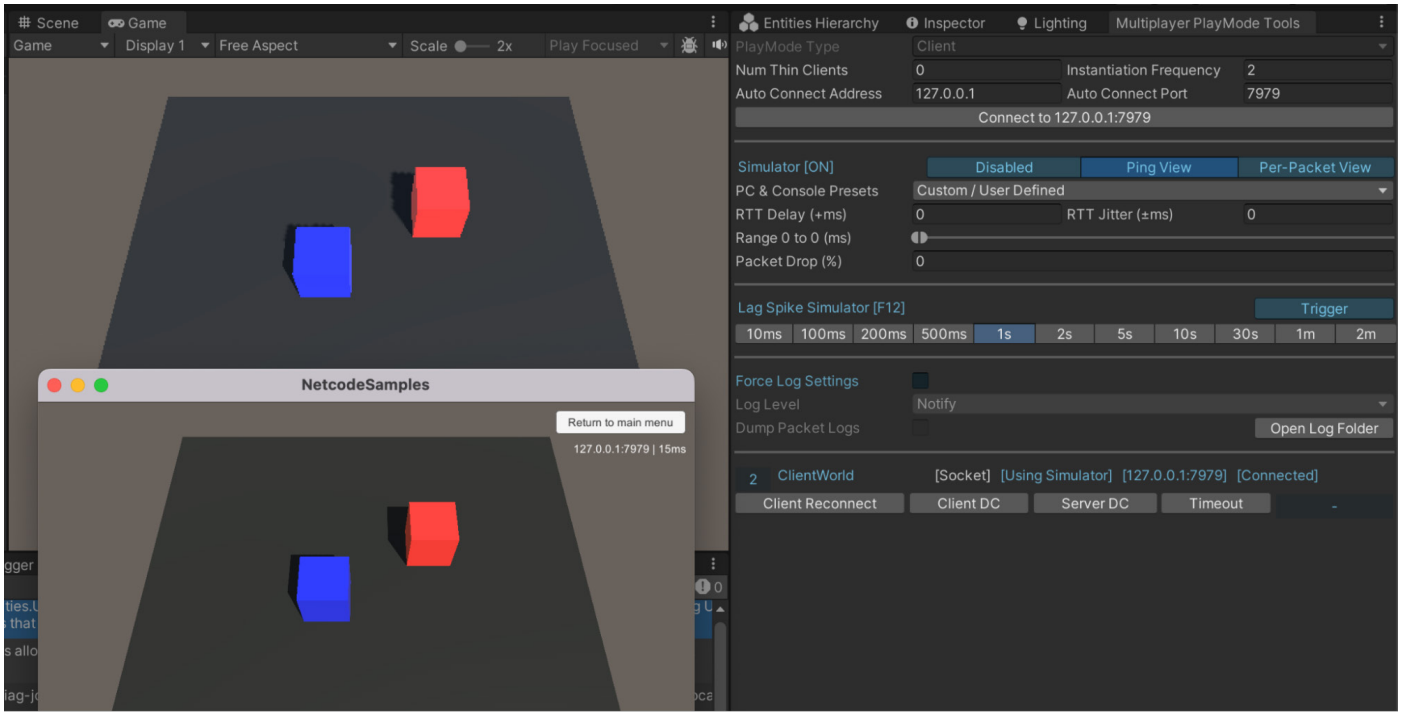
— 서버와 연결 구축

— 서버와 통신

— 서버에 동기화된 엔티티 생성

— 서버와 클라이언트의 스탠드얼론 빌드 생성

— 에디터에서 서버와 클라이언트를 플레이 모드로 실행



에디터에서 스탠드얼론 빌드로 실행 중인 Networked Cube 튜토리얼

ECS Network Racing 샘플

자동차 레이싱 멀티플레이어 메카닉을 선보이는 ECS Network Racing 샘플

ECS Network Racing 샘플은 Unity Physics와 Vivox 음성 채팅이 적용된 로비 기반 멀티플레이어 자동차 레이싱 샘플입니다.

Character Controller

Character Controller 패키지에서는 Unity Physics 및 Netcode for Entities와 함께 작동하는 ECS 기반 1인칭 및 3인칭 캐릭터 컨트롤러의 구현 방법을 제공합니다. 이 컨트롤러는 달리기나 더블 점프와 같은 다양한 일반적인 캐릭터 동작을 지원합니다. 학습용 예제 자료가 더 필요하다면 CharacterController 튜토리얼 및 샘플을 살펴보세요.

Unity 에셋 스토어에서 살펴볼 수 있는 캐릭터 컨트롤러 패키지

DOTS의 로드맵

핵심 DOTS 패키지는 정식 제작에 사용할 수 있지만, 부차적인 DOTS 기반 시스템의 일부는 아직 개발 중에 있습니다. 본 가이드가 작성된 2024년 봄 현재, 관련 시스템의 현황은 다음과 같습니다.

애니메이션
유니티는 엔티티와 함께 작동하는 스킨드 메시 애니메이션 시스템을 개발하고 있으나, 이 가이드 작성 시점 기준으로는 아직 사용할 수 있는 버전의 Unity는 없습니다.

지금으로서는 애니메이션 캐릭터를 게임 오브젝트로 렌더링하고 트랜스폼 및 애니메이션 상태를 엔티티와 동기화하는 방식이 가장 일반적인 솔루션입니다. 다시 말해 게임 시뮬레이션은 엔티티에 완전히 구현하되 애니메이션 캐릭터는 게임 오브젝트로 표현하는 것입니다. 이 아이디어의 간단한 데모를 살펴보려면 샘플 저장소의 'AnimateGameObject' 샘플을 확인하세요. 추가 코드를 작성해야 하고 약간의 오버헤드가 발생하긴 하지만, 대부분의 게임에 적용할 수 있는 해결책입니다.

일부 게임 제작자와 에셋 개발자분들이 구현한 기타 커스텀 애니메이션 솔루션도 있습니다. 커뮤니티에서 다양한 솔루션을 제공하고 있으며 몇몇 솔루션은 Unity 에셋 스토어에서도 사용할 수 있습니다.

사용자 인터페이스

현재 ECS 기반의 UI 시스템은 없습니다. 대신 ECS 기반 게임은 기존의 게임 오브젝트 기반 UI 툴킷을 사용할 수 있습니다. UI 툴킷과 ECS를 함께 사용하는 간단한 예시를 살펴보려면 Firefighters 튜토리얼을 참고하세요. 게임에 UI 툴킷을 사용하는 방법은 다음의 고급 리소스에서 보다 자세히 소개하고 있습니다.

- Unity의 사용자 인터페이스 디자인 및 구현
- UI 툴킷 샘플 – 드래곤 크래셔(Dragon Crashers)
- QuizU – UI 툴킷 샘플
- 토론 섹션의 QuizU 기술 자료 시리즈

Unity ECS 및 DOTS에 관한 최신 소식은 DOTS 로드맵에서 확인하세요.

현재 프로젝트에 DOTS가 필요할까요?

코드에서 CPU 병목 현상이 발생하는 경우, 버스트 컴파일된 잡으로 다시 구현해 보는 것도 좋습니다. 버스트 컴파일된 코드는 Mono 또는 IL2CPP로 컴파일된 코드보다 몇 배 더 빠르게 실행될 뿐만 아니라, 잡을 통해 CPU의 모든 코어에 워크로드를 분할할 수 있습니다.

프로젝트가 기존에 DOTS를 사용하지 않더라도 버스트 컴파일된 잡은 기존 프로젝트 대부분에 비교적 쉽게 통합할 수 있습니다. 관리되지 않는 컬렉션 내외부로 데이터를 복사해야 할 수 있다는 점을 제외하면 기존 코드를 버스트 컴파일된 잡으로 다시 작성하면서 코드 구조를 크게 변경해야 할 일은 없습니다.

그러나 Entities 패키지는 그렇지 않습니다. 특정 기능을 구현하기 위해 일부분에만 엔티티를 통합하는 것이 가능한 경우도 있지만, 대부분의 경우 ECS 아키텍처를 사용하려면 전체 프로젝트의 코드를 특정한 구조로 작성해야 합니다.

엔티티를 사용하여 새 프로젝트를 만드는 것을 고려해 볼 만한 네 가지 이유는 다음과 같습니다.

— 프로젝트에서 디테일한 대형 환경을 렌더링하는 등 다양한 정적 요소가 있는 경우입니다. 기존 메가시티(Megacity) 프로젝트에서 엔티티로 제작된 복잡한 환경을 살펴볼 수 있습니다.

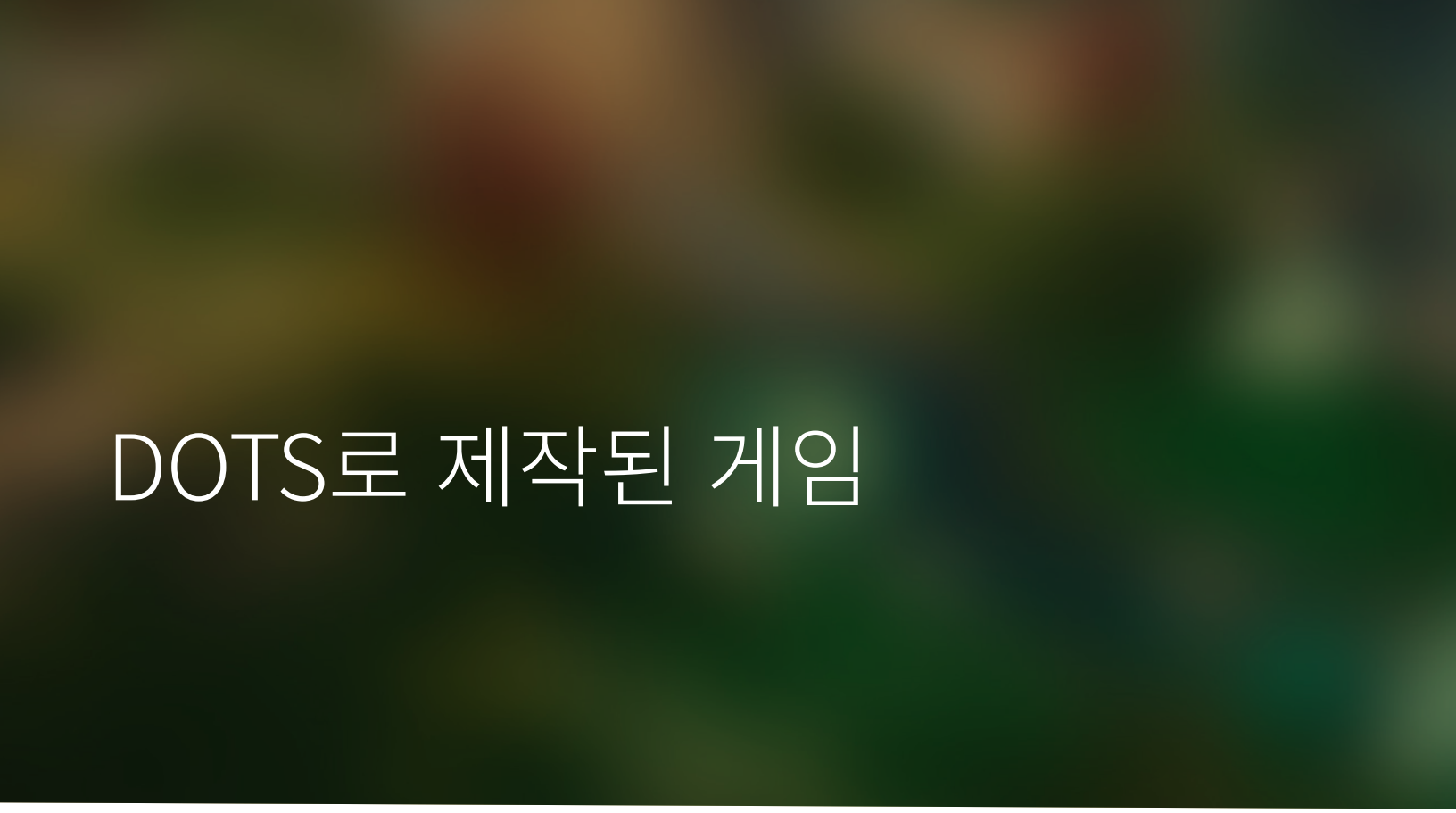
— 프로젝트에 컴퓨팅 리소스가 많이 필요한 동적 요소가 많은 경우입니다. 예를 들어 실시간 전략 게임에서는 수백 또는 수천 개 유닛의 경로 탐색을 계산해야 하는 경우가 많습니다.

— 데이터와 코드를 구성하는 데 일반 객체 지향 방식보다 추론 및 유지 관리가 더 쉬운 ECS 방식을 선호하는 경우입니다. ECS를 사용하면 적어도 보통은 병목 현상의 프로파일링 및 식별이 쉬워진다는 장점이 있습니다.

— 프로젝트가 슈팅 게임과 같이 빠른 액션이 필요한 멀티플레이어 경쟁 게임이라 플레이어 경험 향상을 위해 권한 서버와 클라이언트측 예측 기능이 필요한 경우입니다. 위에서 설명한 것처럼 이러한 기능은 Netcode for Entities에서만 지원되고 Netcode for GameObjects에서는 지원하지 않습니다.

반면 많은 게임이 주로 GPU에서 병목 현상을 겪는데, DOTS는 CPU 효율만 향상하므로 Entities와 나머지 DOTS 패키지 및 기술이 그다지 도움이 되지 않을 수 있습니다. 그렇다 하더라도 DOTS를 사용했을 때 같은 양의 작업을 더 짧은 CPU 시간에 수행할 수 있다면 기능을 추가할 여유가 생기게 되며, 여유가 생기면 나중에 저사양 기기를 타게팅할 때 큰 도움이 될 수 있습니다.

다음 섹션에서는 Unity ECS 및 DOTS 기술을 사용하여 제작된 몇 가지 게임을 소개합니다.



DOTS로 제작된 게임

지난 몇 년 동안 수많은 개발 팀에서는 멀티플랫폼 게임 개발에 DOTS 패키지 및 기술을 사용해 큰 도움을 받을 수 있었습니다. 고객 사례에서 발췌한 내용을 보면, 각 팀이 DOTS를 구현하기로 결정하기 전에 이 기술이 게임에 어떤 도움이 될지 신중하게 고민했다는 사실을 알 수 있을 것입니다.

유니티 리소스 허브에서 더 많은 유니티 고객 사례와 프로파일을 살펴볼 수 있습니다.

DOTS로 제작된 게임: Bare Butt Boxing(Tuatara Games)

Bare Butt Boxing(Tuatara Games), Unity로 제작, PC 및 콘솔 플레이 지원

Tuatara Games는 개발 초기부터 Unity의 DOTS를 사용하여 Bare Butt Boxing을 제작했습니다. 소프트웨어 엔지니어 헨드릭 뒤 투아는 "새로운 팀으로 처음 제작하는 게임이었기에, 올바른 방향으로 디자인을 할 수 있을 만큼 충분히 탄탄한 기반을 이용해 얼리 액세스를 제공하고 싶었어요"라고 말합니다. "DOTS 덕분에 몇 주에 걸쳐 코드를 다시 작성하지 않고도 게임플레이 아이디어를 테스트할 수 있도록 시스템을 모듈화할 수 있었죠."

Tuatara의 데이터 지향 디자인 접근 방식 덕에 반복 작업을 간소화하며 최적화도 유연하게 처리할 수 있었습니다. 게임 프로그래머 이완 아구스는 "ECS를 사용하면 직렬화된 데이터에 영향을 주지 않고 간단하게 런타임 데이터 레이아웃을 조정할 수 있죠"라고 강조합니다.

"ECS 덕분에 문제 없이 게임을 여러 레이어로 나눌 수 있었습니다. 단순하고 시뮬레이션에 직접 적용되는 게임 디자인을 구축할 수 있으며, 더 멋지게 표현하기 위한 시스템을 더할 수 있습니다. 덕분에 시뮬레이션은 클라이언트측에서 예측하여 CPU 부담을 줄이면서 복잡한 표현을 선보일 수 있습니다."

- 이완 아구스, Tuatara Games 게임 프로그래머

DOTS로 제작된 게임: Histera(StickyLock Games)

Histera(StickyLock Games), Unity로 제작, Steam 얼리 액세스로 PC 플레이 지원

"Histera 는 FPS 장르를 전체적으로 혁신한다는 특별한 아이디어에서 시작되었어요. '글리치'(glitch) 덕분에 가능했죠. 글리치는 저희의 주요 USP입니다. 맵의 특정 부분을 완전히 새로운 시대로 전환하는 메커니즘이에요. 선사 시대에서 미래 시대로 이동할 수도 있고, 레이아웃도 완전히 바뀝니다.

DOTS를 선택한 이유는 당시 네트워크 솔루션을 찾고 있었으나 선택지가 많지 않았기 때문입니다. 1인칭 슈팅 게임을 개발하던 중이었으므로 P2P는 신뢰할 수 있는 선택지가 아니라는 점을 알고 있었죠. 전용 게임 서버를 사용하고 싶었습니다. 그러던 중 Unity Netcode와 DOTS의 출시를 자세히 소개하는 Unity 블로그 게시물을 봤고, 함께 공개된 몇 가지 샘플도 확인했습니다. 샘플을 살펴보며 상당한 호기심이 생겼어요. 이 기술로 FPS 게임을 만들 수 있겠다고 생각했으니까요.

DOTS 패키지에 관해 심층적으로 분석하다 보니 개발자 관점에서 꽤 흥미롭더군요. 완전히 새로운 패러다임이었죠. 객체 지향이 아니라 데이터 지향인 점도 말이에요. 저희는 많은 회의와 토론을 거쳐 DOTS와 ECS에 관한 지식을 쌓기 위해 투자하기로 했습니다."

- 자멜 지아티, StickyLock Games 프로듀서

DOTS로 제작된 게임: V Rising(Stunlock Studios)

V Rising(Stunlock Studios), Unity로 제작, PC 플레이 지원

Stunlock Studios는 V Rising 제작에 착수하며, 원하는 규모의 비전을 달성하려면 이전 게임에서와 다른 디자인 패턴이 필요하다는 사실을 빠르게 깨달았습니다. 공동 창업자 겸 테크니컬 디렉터 라스무스 회크는 "파괴와 상호 작용이 가능한 다양한 요소로 생동감 넘치는 월드를 구현하고 싶었어요"라고 합니다.

회크는 DOTS를 테스트해 본 이유가 'DOTS의 활용 분야가 해결하려는 문제에 정확히 부합했기 때문'이라고 설명합니다. Stunlock Studios는 DOTS와 ECS를 사용하여 서버 부하를 줄이고 클라이언트 CPU 리소스 사용량을 최소화했습니다. 또한 동시 접속자 수를 늘리고, 시스템 요구 사항을 낮추며, 스튜디오의 창의적인 비전을 충족할 만큼 확장성이 높고 견고한 기술 스택을 구축할 수 있었습니다.

"ECS는 에디터 데이터와 런타임 데이터를 명확하게 분리한다는 큰 장점이 있습니다. 에디터에서 작업하며 저작 프리팹을 생성하는데, 이 프리팹은 본질적으로 MonoBehavior가 담긴 일반적인 게임 오브젝트죠. 하지만 이 프리팹은 편집에만 사용되며 게임 자체에 직접 사용되지는 않습니다. 대신 베이킹이라는 프로세스를 통해 런타임 컴포넌트로 변환됩니다. 저작 프리팹은 에디터에서만 사용되므로 실제 게임에 대한 영향을 걱정하지 않고 기능과 데이터를 추가하여 워크플로를 개선할 수 있죠.

덕분에 에디터 데이터에 영향을 주지 않으면서 런타임 컴포넌트를 자유롭게 수정하고 최적화할 수 있습니다. 이렇게 분리된다는 점이 V Rising과 같은 복잡한 프로젝트를 관리하는 데 큰 도움이 되었습니다."

– 라스무스 회크, Stunlock Studios 테크니컬 디렉터

DOTS로 제작된 게임: Zenith: The Last City(Ramen VR)

Zenith: The Last City(Ramen VR), Unity로 제작, PC 및 콘솔 VR 플레이 지원

시스템 기반 게임인 MMO에는 강력하고 확장 가능한 기술 기반이 필요합니다. 개발 초기 단계에서 Ramen VR은 MonoBehavior를 사용하여 Zenith의 시스템을 구성했지만, 수백 개의 동일한 게임 오브젝트에서 로직을 수백 번 실행하는 것은 비효율적이었습니다.

Ramen VR은 Unity의 ECS 프레임워크를 활용하여 객체 지향 프로그래밍의 단점에서 벗어날 수 있었습니다. CTO 로렌 프레이저는 “MMO는 ECS를 응용하기에 매우 적절한 분야입니다”라고 말합니다. “Zenith에서는 수천 개의 엔티티가 공존해야 하는데, ECS 덕분에 대규모로 실행할 수 있죠.”

새로운 워크플로에서 모든 ‘액터’ 게임 오브젝트(플레이어, 몹, 수집품 등)는 대응하는 ECS 엔티티가 주어집니다. ECS는 게임 오브젝트를 순회하며 관련 태그를 확인하고, 태그가 발견될 때마다 로직을 트리거합니다.

“상황에 적합한 워크플로를 선택할 수 있어 좋습니다. 오브젝트만 사용할 수도, 엔티티만 사용할 수도 있었어요. 하지만 무조건 하나를 선택할 필요는 없다고 생각합니다.”

– 로렌 프레이저, Ramen VR CTO

DOTS로 제작된 게임: 메가시티 메트로 샘플

메가시티 메트로 샘플

유니티의 메가시티 메트로(Megacity Metro) 샘플은 기존의 메가시티 샘플을 모바일 중심 멀티플레이어로 변경한 샘플입니다. 메가시티 메트로는 URP, Entities, Netcode for Entities 및 Unity Physics로 제작되었으며, 저사양 모바일 플랫폼부터 고사양 콘솔 플랫폼까지 다양한 기기에서 실행할 수 있습니다. 100명 이상의 플레이어가 참여할 수 있는 강력한 크로스 플레이를 지원하며 Authentication, Game Server Hosting, Matchmaker, Vivox와 같은 Unity Gaming Services 요소가 포함되어 있습니다.

여기에서 메가시티 메트로에 대해 자세히 알아보고 프로젝트를 다운로드할 수 있습니다.

부록: ECS와 관련된 개념

DOTS는 데이터 지향적이므로 MonoBehaviour 프로젝트의 객체 지향 프로그래밍 방식보다 하드웨어 친화적입니다. DOD는 MonoBehaviour 기반 프로젝트의 C# 프로그래밍에서는 일반적으로 연관성이 별로 없었으나, DOD와 관련되어 있으며 그로 인해 영향을 받은 몇 가지 주요 개념을 잘 이해하면 도움이 될 것입니다.

메모리 할당 및 가비지 컬렉션

최신 운영 체제에서는 프로그램이 개별 프로세스로 실행되며, 운영 체제가 각 프로세스의 메모리를 관리합니다. 프로세스는 더 많은 메모리가 필요한 경우 운영 체제에 메모리를 요청하며, 운영 체제는 요청에 따라 연속된 메모리 블록을 프로세스에 제공합니다. 이를 메모리 할당이라고 합니다.

프로세스가 종료되면 운영 체제는 메모리를 회수하고, 다른 곳에서 사용할 수 있도록 할당을 해제합니다. 하지만 오래 실행되는 프로그램의 경우 더 이상 사용하지 않는 메모리 블록을 반환하는 것이 바람직할 때가 많습니다. 이를 메모리 해제 또는 할당 해제라고 합니다. 짧게 실행되는 단순한 프로그램이라면 메모리를 해제하는 일 없이 할당하기만 해도 무방한 경우가 많습니다. 그러나 오래 실행되는 프로그램에 메모리를 새로 할당하기만 하고 해제하지 않으면 이 프로그램은 결국 너무 많은 양의 메모리를 사용하게 될 수 있습니다. 이러한 상황을 메모리 누수라고 하며, 이는 성능 저하 및 불안정성으로 이어질 수 있습니다.

프로그램은 자체 내부 할당자를 사용하는 경우가 많으며, 다음과 같은 방식으로 작동합니다.

1. 프로그램이 운영 체제로부터 대규모 메모리 블록을 할당받습니다.
2. 프로그램의 자체 내부 할당자가 블록 내 현재 사용 중인 범위를 추적합니다.
3. 더 많은 메모리가 필요하면 프로그램은 운영 체제에 요청하는 대신 내부 할당자에 메모리를 요청합니다.
4. 내부에 할당된 메모리 블록이 더 이상 필요하지 않으면 프로그램이 할당자에 메모리를 해제하도록 알립니다.

내부 할당자에는 몇 가지 장점이 있습니다.

— 운영 체제로부터 할당을 받고 할당을 해제할 때와 달리, 내부 할당자로부터 할당을 받고 해제할 때는 비용이 많이 드는 시스템 호출이 보통 필요하지 않습니다.

— 프로그램에서 다양한 용도에 보다 잘 대응하기 위해 여러 커스텀 할당자를 사용할 수 있습니다. 규모가 작고 수명이 짧은 할당에 적합한 할당자가 있는 반면, 규모가 크고 수명이 긴 할당에 적합한 할당자도 있습니다. 예를 들어 소위 '아레나 할당자'(arena allocator)는 모든 할당을 동시에 해제하므로 내부 로직과 추적이 아주 단순하고 비용이 적습니다.

C#을 비롯해 현재 많이 사용되는 여러 언어에서는 런타임에 메모리를 스캔하여 사용되지 않는 할당을 찾고 해제하는 가비지 컬렉터를 사용합니다. 수동으로 할당하는 것에 비해 자동으로 처리하면 더 간편하고 메모리 누수를 비롯한 여러 메모리 관련 문제를 더 쉽게 회피할 수 있습니다. 반면 가비지 컬렉션으로 인해 오버헤드가 발생하고 프로그램 실행을 중단해야 하므로, 플레이어 경험에 부정적인 영향을 주는 일시 정지 현상이 눈에 띄게 발생할 수 있다는 단점이 있습니다.

DOTS에서는 엔티티와 네이티브 컬렉션이 관리되지 않으므로, 런타임 또는 가비지 컬렉터로 할당되거나 관리되지도 않습니다.

— 엔티티의 경우 EntityManager를 통해 메모리가 할당 및 해제되므로, 메모리 누수는 엔티티가 더 이상 필요하지 않아 제거하는 것을 잊은 경우에만 발생합니다. 실제로는 알아채기 쉬우므로 이런 실수는 쉽게 발견하고 고칠 수 있습니다.

— 네이티브 컬렉션의 경우, DOTS에서 각각 장단점이 있는 여러 할당자를 제공합니다. 예를 들어 **Allocator.Temp** 할당자는 할당된 프레임 또는 잡의 종료 시 자동으로 해제되며 비용이 아주 적은 할당을 제공합니다. 반면 **Allocator.Persistent** 할당자는 수동으로 해제할 때까지 수명이 계속 유지되는 비용이 큰 할당을 제공합니다. Allocator.Temp와 다르게 Allocator.Persistent는 큰 메모리를 할당할 수 있고, 잡에 전달할 수도 있습니다. 이외에도 **Allocator.TempJob**이나 **WorldUpdateAllocator** 등의 할당자가 있습니다.

자세히 알아보려면 Unity의 가비지 컬렉터 기술 자료를 참고하세요.

멀티스레드 프로그래밍

대부분의 최신 CPU에는 두 개 이상의 코어가 탑재되어 있으며, 이러한 추가 코어를 활용하면 CPU를 많이 소모하는 게임의 성능을 크게 향상할 수 있습니다. 하지만 멀티스레딩에는 많은 C# 개발자에게 익숙하지 않을 영역인 하위 수준의 수동 프로그래밍이 필요한 경우가 많으므로 어렵고 안전하지 않을 수 있습니다. DOTS에서는 C# 잡 시스템을 통해 흔히 발생하는 문제를 피하면서 간단하게 안전한 멀티스레드 코드를 작성할 수 있지만, 이 섹션에서는 멀티스레딩 문제의 근본을 살펴보겠습니다.

운영 체제에서 생성한 프로세스는 하나의 실행 스레드를 가지고 시작합니다. 프로세스는 시스템 호출을 통해 동일한 프로세스에 속하고 같은 메모리를 공유하는 추가 스레드를 생성할 수 있습니다.

CPU의 각 논리적 코어는 한 번에 하나의 스레드를 실행할 수 있으며, 어떤 스레드를 언제 어느 코어에서 실행할지는 운영 체제가 제어합니다. 운영 체제는 언제든지 실행 중인 스레드를 중단하여 다른 스레드가 코어를 사용하도록 할 수 있습니다. 두 스레드가 같은 리소스(데이터)에 액세스할 때, 한 스레드가 예상하지 못하는 상태에서 다른 스레드가 리소스를 변경하면 위험할 수 있습니다. 일반적으로 스레드는 공유 리소스에 대한 독점적인 액세스 권한이 있는 코드의 '임계 구역'(critical section)에서만 공유 리소스를 읽고 변경해야 합니다.

스레드 간의 액세스 권한을 제어하기 위해 공유 리소스는 뮤텍스(mutex) 같은 동기화 프리미티브로 관리됩니다. 그러나 이러한 동기화 프리미티브를 사용하려면 일반적으로 모든 스레드가 엄격한 프로토콜을 따라야 하며, 따르지 못하면 프리미티브가 효과를 발휘하지 못하거나 프로그램이 중단될 수 있습니다.

또 다른 문제는 특정 시스템 호출로 인해 호출 스레드가 차단되어 실행이 중단될 수 있다는 것입니다. 예를 들어 스레드가 파일을 읽기 위해 시스템 호출을 실행할 때, 대부분의 경우 데이터가 메모리에 없으므로 시스템 호출이 반환되려면 먼저 기기에서 메모리로 데이터를 복사해야 합니다. CPU 입장에서 데이터가 복사되기를 기다리는 시간은 매우 길 수 있으므로, 운영 체제는 데이터가 로드되는 동안 호출 스레드를 차단하고 CPU 코어에서 다른 스레드를 실행할 수 있습니다. 운영 체제는 데이터가 준비되어야만 스레드의 차단을 해제하고 실행을 재개합니다.

따라서 멀티스레딩의 용법 중 하나는 '메인 스레드'에서는 작업을 계속하면서, '백그라운드 스레드'에서는 파일 읽기/쓰기와 같이 오래 실행되는 차단 작업을 수행하는 것입니다. 예를 들어 인터랙티브 프로그램의 경우 백그라운드에서 파일을 로드하면서 메인 스레드는 계속 사용자 입력에 응답하고 화면을 다시 드로우해야 할 수 있습니다.

또한 단순히 프로그램의 CPU 워크로드를 여러 코어에 분할하여 작업 완료 속도를 높이는 활용법도 있습니다. 예를 들어 데이터 압축은 CPU를 많이 소모하므로 멀티스레딩의 장점을 활용할 수 있습니다.

한 가지 기억해야 할 사항으로, CPU 코어는 스토리지 기기, 시스템 메모리 및 기타 시스템 리소스를 사용하려면 다른 코어와 경쟁을 벌여야 합니다. 예를 들어 두 스레드가 동시에 같은 메모리에 액세스하려 하면 함께 액세스하지 못하고 차례를 기다려야 합니다. 다행히 이러한 중복 문제는 하드웨어 수준에서 처리되지만, 여전히 각 스레드의 메모리 액세스가 나머지 모든 스레드의 중복 메모리 액세스 속도를 늦춘다는 문제가 있습니다. 특히 CPU 계산에 비해 많은 메모리 액세스가 필요한 작업의 경우, 작업을 여러 스레드로 분할했을 때 얻는 장점이 점점 줄어들 때가 많습니다.

예를 들어 작업을 10개의 스레드로 분할하면 같은 작업을 하나의 스레드에서 실행할 때보다 10배 빨라질 것으로 생각할 수 있지만, 실제로는 메모리 경쟁 때문에 이러한 이론적 수치가 달성되는 경우는 드뭅니다. 10개의 스레드를 사용한다고 가정하고, 상황에 따라 성능이 대략 5~7배 향상될 것으로 보는 것이 현실적입니다. 사실상 계산 작업에 비해 많은 메모리 액세스가 필요한 작업의 경우, 더 적은 수의 스레드를 사용하면 메모리 경쟁이 줄어들어 성능이 향상될 수도 있습니다.

메모리 및 CPU 캐시

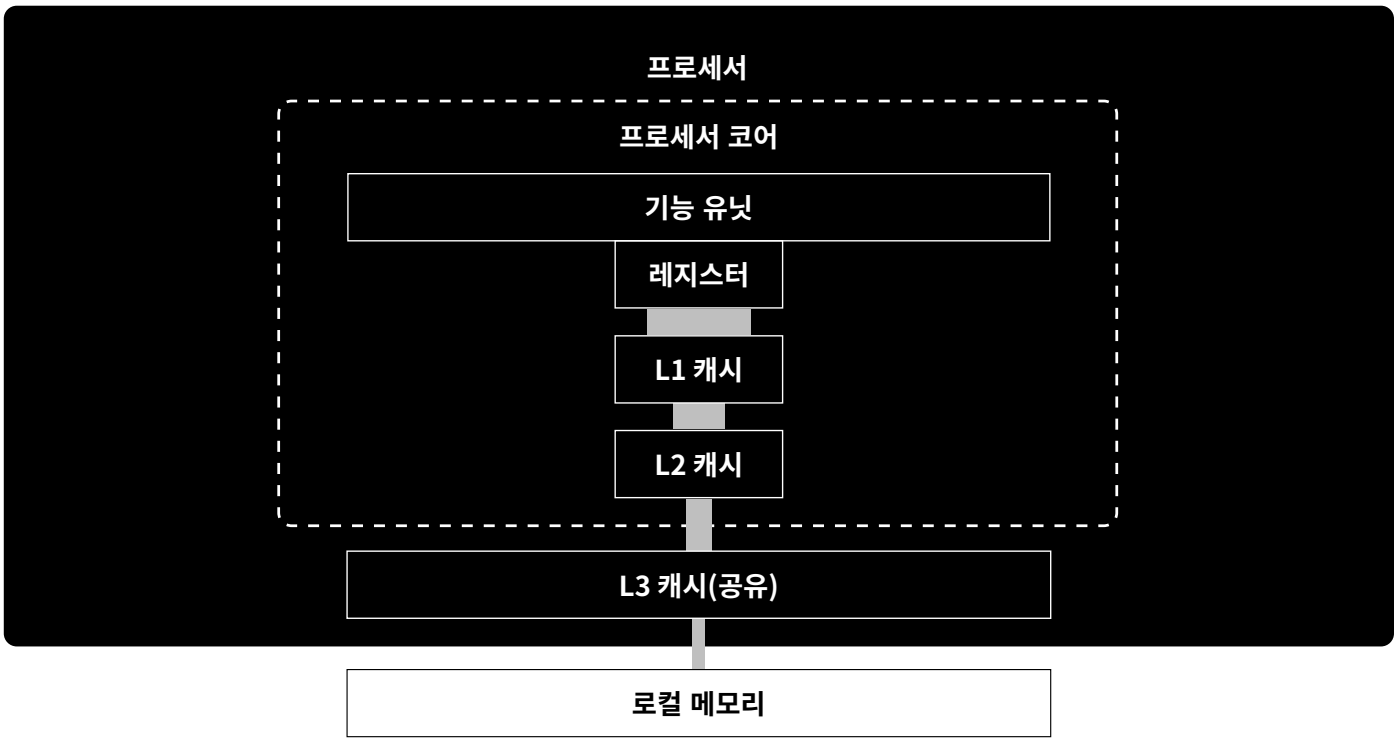
최신 CPU에는 대부분 1~3개 레벨의 캐시가 있으며, 이는 L1, L2 및 L3와 같이 지정됩니다.

CPU가 시스템 메모리의 주소를 읽는 명령어를 실행할 때, 하드웨어는 먼저 해당 주소의 데이터 사본이 L1 캐시에 저장되어 있는지 확인합니다. 사본이 없고 L2 캐시가 존재한다면 하드웨어가 L2 캐시를 확인하고, 그다음 L3 캐시가 존재한다면 L3 캐시를 확인합니다. 어느 캐시에도 사본이 저장되어 있지 않다면 하드웨어가 시스템 메모리 자체를 읽습니다. 하드웨어는 데이터를 읽으면서 하위 레벨 캐시의 일부분에 데이터를 복사합니다. 예를 들어 L3에서 데이터를 읽으면 L1 및 L2에 복사하고, 시스템 메모리에서 데이터를 읽으면 모든 레벨의 캐시에 복사하는 방식입니다. 주소를 읽은 후 곧 같은 주소를 다시 읽는 경우가 많기 때문에 유효한 캐싱 전략입니다. 데이터를 캐시에 복사하면 다음에 다시 필요할 때 캐시에서 바로 데이터를 읽을 수 있습니다.

물론 모든 시스템 메모리를 캐시에 저장할 수는 없습니다. 각 레벨의 캐시는 하위 레벨의 캐시보다 크기가 작으며, 가장 큰 캐시도 전체 시스템 메모리보다 훨씬 작습니다. 따라서 메모리의 일부를 캐시에 복사할 때, 이전에 캐싱된 다른 메모리를 덮어써야 합니다.

캐시에서 데이터를 읽으면 이것을 '캐시 적중(hit)'이라고 합니다. 현재 데이터 사본이 캐싱되어 있지 않아 시스템 메모리에서 데이터를 읽어야 하면 이것을 '캐시 부적중(miss)'이라고 합니다.

캐시의 정확한 성능 특성은 칩마다 크게 다르지만 대략 L1은 L2보다 최소 몇 배 빠르며, L2는 L3보다 최소 10배 빠르고, L3는 시스템 메모리보다 최소 2배 빠릅니다. 종합하면 CPU가 L1 캐시에서 데이터에 액세스하는 것이 시스템 메모리에서 데이터에 액세스하는 것보다 100배 이상 빠릅니다. 하위 레벨의 캐시와 시스템 메모리 간의 속도 격차가 아주 크므로, 프로그램에서 발생하는 캐시 부적중 횟수를 최소화하는 것이 성능을 향상하는 데 중요합니다.



레벨별 캐시, 출처: https://tech4gamers.com

프리페치(Prefetch)라는 하드웨어 기능 덕분에, 메모리 주소에 무작위로 액세스하지 않고 순차적으로 액세스하는 간단하고 효율적인 방법으로 캐시 부적중을 최소화할 수 있습니다. 따라서 캐시 효율적인 데이터 구조는 요소를 빽빽하게 채워진 연속된 형태로 저장합니다. 말하자면 데이터를 배열로 저장하는 것입니다.

메모리 주소를 순차적으로 읽기 시작하면 하드웨어가 패턴을 파악하고 계속 반복할 것이라 예상하여 메모리를 미리 읽고 캐시에 복사합니다. 미리 복사한 데이터가 필요하지 않은 경우 낭비가 될 수 있지만, 배열을 순차적으로 읽는 경우 프리페치 동작을 통해 CPU가 필요로 하는 데이터를 사용 직전에 캐시에 저장합니다. 따라서 배열의 첫 바이트를 읽을 때 발생하는 초기 캐시 부적중을 제외하고는 캐시 부적중 없이 배열을 읽을 수 있습니다. CPU라는 열차가 달릴 때 딱 맞춰서 열차 앞에 선로가 깔린다고 생각해 보세요.

게임 오브젝트나 MonoBehaviour와 같이 관리되는 C# 오브젝트는 개별적으로 인스턴스화되므로 메모리 내 다른 부분에 저장될 수 있습니다. 따라서 관리되는 오브젝트에 하나씩 액세스하려면 일반적으로 메모리 내 여러 위치에 액세스해야 하므로 캐시 부적중이 많이 발생합니다.

DOTS에서는 디자인상 엔티티와 그 컴포넌트가 배열에 연속적으로 빽빽하게 채워지므로 캐시 부적중을 최소화하며 순차적으로 액세스할 수 있습니다.

메모리와 캐시에 대해 자세히 알아보려면 스콧 마이어스가 발표한 CPU 캐시와 그 중요성을 확인해 보세요.

객체 지향 프로그래밍의 비용

OOP(객체 지향 프로그래밍)는 흔히 한 가지로 정의하기 어렵다고 합니다. 어떤 사람들은 OOP의 핵심이 상속, 다형성, 캡슐화 또는 이 세 가지의 조합이라고 하고, 또 어떤 사람들은 독특한 이론을 제시합니다. 위키백과에서는 다음과 같이 정의합니다.

"OOP(객체 지향 프로그래밍)는 객체라는 개념을 기반으로 한 프로그래밍 패러다임이다. 객체는 필드(혹은 속성 또는 프로퍼티) 형태의 데이터와 절차(혹은 메서드) 형태의 코드로 구성된다. **OOP에서 컴퓨터 프로그램은 서로 상호 작용하는 객체를 통해 디자인된다.**" – 위키백과(영문)

다시 말해 객체 지향적인 프로그램은 상호 작용하는 '객체'로 구성되며, 각 객체는 데이터와 코드가 캡슐화된 단위이고 다른 객체로부터 일정한 자주성과 독립성을 가지고 있습니다. 네트워크상의 프로그램이 서로 메시지를 전송하며 협력하는 것처럼, 객체 지향적인 프로그램의 객체는 서로의 메서드를 호출하여 협력합니다. 사실상 객체 지향 프로그래밍을 정의하는 것은 개별 객체 자체가 아니라 객체 간의 상호 작용입니다.

이론상으로 OOP의 장점은 다음과 같습니다.

- **결합성**: 객체로 구성된 프로그램은 계속 합치고 수정할 수 있습니다.
- **재구성**: 객체를 추가, 제거, 대체하여 기능을 간단하게 추가, 제거, 수정할 수 있습니다.
- **코드 재사용**: 여러 프로그램에서 객체를 간단하게 재사용할 수 있습니다.
- **직관성**: 실제 사물과 절차를 자연스럽게 객체에 대응할 수 있습니다.
- **추상화**: 객체를 사용하면 프로그래머가 하위 수준의 디테일에 신경 쓰지 않고 상위 수준에서 문제를 해결할 수 있습니다.

이 부분에 대해 스티브 잡스는 1994년 6월 16일 자 Rolling Stone 인터뷰에서 다음과 같이 이야기했습니다.

> "객체는 사람과 같습니다. 살아 숨쉬는 존재로서 일을 처리하는 방식에 대한 지식이 있고 정보를 기억할 수 있는 메모리가 있습니다. 그리고 객체와 상호 작용할 때는 저수준이 아니라 고수준으로 추상화하여 상호 작용할 수 있습니다. 여러분과 제가 지금 하듯이 말이죠.
>
> 예시를 들어 보겠습니다. 제가 여러분의 빨래 객체라 해 보죠. 여러분은 저한테 지저분한 옷을 주면서 "옷을 빨래해 주세요"라고 말합니다. 저는 다행히 샌프란시스코 최고의 세탁소가 어디에 있는지 알고 있습니다. 그리고 영어를 구사하고, 주머니에 현금도 있습니다. 저는 밖으로 나가 택시를 잡고 운전사에게 샌프란시스코의 이 세탁소로 가자고 합니다. 세탁소에서 옷을 빨래한 다음, 다시 택시를 타고 여기로 돌아옵니다. 여러분에게 옷을 건네며 "깨끗해진 옷을 받으세요"라고 말하죠.

> 여러분은 제가 어떻게 한 것인지 전혀 모릅니다. 세탁소에 대해서도 알지 못하죠. 여러분은 영어를 몰라 택시도 잡지 못하는 사람일 수 있습니다. 가진 현금이 없어 택시비를 지불하지 못할 수도 있겠고요. 하지만 저는 그 모든 정보를 알고 있었고, 여러분은 그 어떤 정보도 알 필요가 없었습니다. 모든 복잡한 정보는 제 안에 숨겨져 있었고, 여러분과 저는 매우 고수준의 추상화로 상호 작용할 수 있었습니다. 이것이 바로 객체입니다. 객체는 복잡한 정보를 캡슐화하고, 그러한 정보에 대해 고수준의 인터페이스를 제공하는 것입니다."

OOP의 성능 비용

단점이 있다면, OOP에서는 여러 성능 비용이 발생하는 경우가 많습니다.

- **분산된 데이터 레이아웃:** OOP 코드는 흔히 여러 작은 객체로 나뉘어 있으므로, 데이터가 메모리 전반에 분산되는 경우가 많아 이전 섹션에서 설명한 것처럼 캐시 비효율로 이어집니다.
- **과도한 추상화:** 객체 지향 디자인에서는 실제 작업을 고수준에서 저수준으로 위임하는 델리게이트(delegate) 레이어가 자주 사용되는데, 이로 인해 실제 작업을 별로 수행하지 않는 객체와 메서드가 많아집니다.
- **복잡한 호출 체인:** 추상화 레이어가 많으며 작은 함수가 선호되므로 호출 체인이 아주 복잡해집니다.
- **가상 호출**: 가상 디스패치 테이블은 일반 함수 호출에 비해 오버헤드를 유발하며, 가상 호출은 일반적으로 인라인 처리할 수 없습니다(일부 JIT 컴파일러는 런타임에 인라인 처리 가능).
- **바람직하지 않은 할당 패턴:** OOP에서는 코드 경로가 복잡해 객체 수명을 추론하기 어려울 때가 많으므로 OOP 코드는 더 효율적인 대안보다 작고 빈번한 할당과 가비지 컬렉션에 의존하는 경우가 많습니다.
- **한 번에 하나만 처리:** 객체를 직접 조작하는 코드가 객체 안에 있으므로 OOP에서는 기본적으로 객체를 대량으로 한 번에 처리하는 대신 한 번에 하나만 처리하는 경우가 많습니다.

OOP의 구조적 비용

성능을 다소 타협하더라도 프로그램의 작성과 유지 관리만 쉬우면 된다고 생각할 수 있으나, OOP는 이 부분에서도 단점이 있습니다. OOP의 몇 가지 단점은 다음과 같습니다.

1. 데이터와 코드를 결합하면 둘 다 더 지저분하고 복잡해집니다.

OOP에서는 코드보다 데이터가 우선시된다고 주장하는 경우가 많습니다.

> "OOP(객체 지향 프로그래밍)는 함수와 로직 대신 데이터 또는 객체를 중심으로 소프트웨어 디자인을 구성하는 컴퓨터 프로그래밍 모델이다.[…] OOP에서는 개발자가 객체를 조작하는 데 필요한 로직보다 조작하고자 하는 객체에 집중한다."
>
> – 알렉산더 S. 길리스, 객체 지향 프로그래밍이란(영문), TechTarget Network 게시

그러나 실제로 OOP에서는 데이터와 코드가 서로 얽히게 됩니다. 객체의 기능이 데이터를 직접적으로 따라야 하고 그 반대도 만족해야 한다면, 객체가 수행할 수 있는 작업은 객체의 정의와 밀접하게 연관되며 객체의 데이터와 분리될 수 없습니다.

이러한 얽힘으로 인해 디자인 중 다음과 같이 의문이 드는 선택을 하게 되는 경우가 많습니다.

— 실제로는 데이터만 있어야 하지만 코드도 있는 객체

— 실제로는 코드만 있어야 하지만 데이터도 있는 객체

— 코드를 위해 데이터를 그룹화하는 객체

— 데이터를 위해 코드를 그룹화하는 객체

— 데이터를 위해 여러 객체에 나뉘어 있는 코드

— 코드를 위해 여러 객체에 나뉘어 있는 데이터

2. 집중된 복잡도를 분산하면 전체 복잡도가 증가합니다.

객체 지향 디자인의 원칙에 따르면 '책임'이 너무 많은 객체는 작은 객체로 분할해야 합니다. 하지만 큰 객체를 작은 조각으로 나누면 결국 전체 복잡도를 줄이기는커녕 복잡도를 분산시키게 됩니다. 실제로 여러 작은 조각으로 구성된 코드 베이스에서는 한 조각의 데이터 또는 코드가 어떤 목적으로 작성되었는지 파악하거나 특정 기능과 연관된 부분의 코드가 무엇인지 알기 어렵습니다.

따라서 적절하게 디자인된 객체 조합에 올바르게 책임을 위임하는 한 코드를 명확하게 만든다는 객체 지향 디자인의 목적이 달성되지만, 객체 지향 디자인 프로세스 자체는 번거롭고 추측으로 가득하며, 과도하게 분할된 프로그램 구조가 되는 경우가 많습니다.

3. 객체를 사용하면 어떤 코드가 어떤 데이터에 액세스하는지 추적하기 어렵습니다.

프로그램을 이해하려면 결국 데이터를 이해하고 데이터가 어떻게 변환되는지 파악해야 합니다. 데이터에 대해 추론하기 쉬울수록 프로그램을 추론하기 쉽습니다. 기능을 추가하거나 버그를 고칠 때, 프로그래머는 어떤 코드가 특정 데이터에 영향을 주는지 밝힐 수 있어야 하며, 반대로 어떤 데이터가 특정 코드의 영향을 받는지도 확인할 수 있어야 합니다.

객체 지향 프로그램에서는 객체가 많이 연결될수록 이를 파악하기 어려워집니다. 객체 캡슐화를 통해 특정 데이터를 비공개해 직접 액세스하는 것을 막을 수 있으나, 간접적으로 연결된 객체에서 특정 공개 메서드 호출 경로를 통해 간접적으로 액세스가 가능할 수도 있습니다. 예를 들어 특정 값이 올바르지 않게 설정되는 이유를 디버깅할 때, 관련된 모든 경로의 코드를 식별하려면 많은 분석 작업이 필요합니다. 반면 엄격하게 절차적인 프로그램에서는 글로벌 변수를 분별없이 사용하지 않는 한 특정 데이터에 영향을 주는 모든 코드 경로를 식별하는 데 고려해야 하는 경우의 수가 더 적습니다.

데이터 지향 디자인

DOD(데이터 지향 디자인)라는 용어는 2000년대에 일부 게임 프로그래머와 고성능 소프트웨어에 관심을 가진 사람들 사이에서 생겨난 아이디어를 설명하기 위해 만들어졌습니다. 데이터 지향 디자인에 대해 공식적으로 정의된 바는 없지만, 다음 리소스에서 가장 근접한 정의를 찾을 수 있습니다.

- 데이터 지향 디자인과 C++ 및 데이터 지향적인 미래 형성(영문): 마이크 액톤의 발표 동영상 2개
- 데이터 지향 디자인(영문): 리처드 파비안의 책
- 데이터 지향 디자인 리소스: DOD에 관한 링크 모음

이 가이드에서는 이론적인 내용보다는 DOD에 관한 몇 가지 실용적인 내용을 설명하겠습니다.

코드 이상으로 중요한 데이터 디자인

DOD의 핵심 전제는 **데이터가 적어도 코드만큼 중요하다**는 것입니다. 거시적 수준에서 봐도, 또한 미시적 수준에서 봐도, **프로그램은 결국 데이터를 변환하고 생성하는 것이 핵심입니다**.

따라서 코드가 데이터의 구조를 결정하는 게 아니라 본질적인 데이터가 코드의 구조를 결정해야 합니다. 이는 프로젝트 시작 단계뿐만 아니라 모든 단계에서 유효하므로 기능을 추가하거나 변경할 때, 코드의 구조를 변경하기 전에 먼저 데이터의 구조를 다시 평가해야 합니다.

이는 객체에서 데이터와 코드가 떼려야 뗄 수 없이 연결되는 객체 지향 디자인과 상충합니다. 코드를 고려하여 데이터를 디자인하면 디자인 프로세스가 복잡해지고 최적화와는 거리가 있는 디자인으로 이어질 때가 많습니다. 반대로 코드를 고려하지 않고 데이터를 수정할 수 있는 자유가 주어지면 디자인 프로세스가 단순해지고 일반적으로 더 단순한 최적의 데이터를 얻을 수 있습니다.

단순한 데이터 선호

일반적으로 데이터가 단순할수록 코드가 단순하고 효율적입니다. 특히 계층 구조나 그래프 구조보다 배열을 사용하는 것이 좋습니다. 배열은 많은 데이터 요소를 저장하는 가장 간단한 방법이며, 1차원 배열을 순차적으로 순회하는 것이 메모리에 액세스하는 가장 효율적인 방법입니다.

또한 포인터와 배열 인덱스를 사용하여 데이터 요소 간에 불필요한 연결이 없도록 주의해야 합니다. 이러한 연결을 올바르게 관리하려면 코드가 복잡해지며, 연결을 순회하려면 최적이 아닌 무작위 룩업이 필요합니다.

코드를 데이터 파이프라인으로 간주

데이터 디자인의 대략적인 초안을 만들었다면 이제 데이터가 어떻게 변환되는지 생각해야 합니다.

— 서버의 경우, 클라이언트 요청과 데이터베이스 데이터가 서버 응답으로 변환됩니다.

— 컴파일러의 경우, 소스 코드가 기계어 코드 또는 일종의 중간 코드로 변환됩니다.

— 오디오 인코더의 경우, 특정 형태의 오디오 데이터가 다른 형태로 변환됩니다.

— 비디오 게임의 경우, 특정 순간의 사용자 입력과 게임 상태가 새로운 게임 상태로 변환되며, 이후 새로운 렌더링 프레임으로 변환됩니다.

물론 이러한 거시적 수준의 변환은 여러 하위 단계로 나눌 수 있지만, 목표는 여전히 같습니다. 데이터의 초기 상태에서 단순히 점을 연결하여 예상하는 최종 상태에 도달하는 것입니다. 그러면 코드를 자연스럽게 '데이터 파이프라인' 구조로 생각하여 각 단계에서 파이프라인의 다음 단계에 전달하기 위해 데이터를 변환하거나 생성하면 됩니다.

프로그래밍을 이렇게 설명하면 너무 간단하고 당연하게 느껴질 수 있지만, 여타 소프트웨어 제작 방법론과 비교하면 아주 명확합니다. 시작점과 종료점을 잘 정의하고 나면 A 지점에서 B 지점에 도달하는 정확한 방법을 파악하는 것은 아주 명확하고 처리하기 쉬운 문제이며, 각 변환 과정은 나머지 과정으로부터 독립적으로 작성하고 재작성할 수 있습니다.

이 모델을 통해 올바르게 동작하는 솔루션을 간단하게 작성할 수 있을 뿐만 아니라 최적화하기도 쉽습니다.

첫째, 연속된 단계에서 병목 현상을 식별하려면 각 단계를 프로파일링하기만 하면 됩니다. 일반적으로 적은 수의 단계에서 대부분의 비용이 발생하므로, 어느 부분에서 최적화하면 가장 큰 영향이 있을지 명확하게 파악할 수 있습니다. 따라서 최적화 작업의 우선순위를 지정할 때 비용 대비 결과를 고려하고 성능 최적화에 실용적으로 접근하는 것이 중요합니다.

둘째, 파이프라인 모델을 사용하면 최적화할 부분을 쉽게 발견할 수 있습니다. 다음과 같은 경우는 흔하게 볼 수 있습니다.

— 특정 데이터가 이후 단계에서 효율적으로 처리될 수 있도록 중간 형태로 변환하는 것이 유리한 경우

— 여러 단계에서 반복적으로 생성되는 데이터를 초기 단계에서 한 번 캐싱하는 것이 유리한 경우

— 동일한 데이터에 액세스하는 여러 단계를 전체 또는 부분적으로 결합하여 단계 수를 줄여서 반복 액세스 오버헤드를 줄이는 것이 좋은 경우

— 한 번에 하나씩 처리되는 데이터 요소를 일괄 처리하면 효율적으로 메모리에 액세스하고, 브랜칭을 줄이고, 함수 호출 오버헤드를 줄이는 등 효율성이 향상되는 경우

마지막으로, 데이터 파이프라인은 병렬화로 이어집니다. 어느 단계에서 어떤 데이터에 액세스하는지만 명확하게 구분해 두면, 어느 단계를 안전하게 동시에 처리할 수 있는지 쉽게 식별할 수 있습니다.

개발의 모든 단계에서 성능 측정, 예측, 예산 책정

게임 개발 과정에서 흔히 범하는 실수는 프로젝트 후반에 비로소 성능 개선을 시작하는 것입니다. 최적화를 늦게 시작하면 비용이 많이 들고 위험도 커지는 이유는 다음과 같습니다.

— 많은 최적화 작업이 프로젝트 후반에 진행하기 어렵습니다.

— 최적화를 늦게 시작하면 필요한 시간과 노력을 예측할 수 없습니다.

— 최적화를 늦게 시작하면 만족할 만한 결과를 얻는 데 실패할 수 있습니다.

프로젝트 후반까지 기다리는 대신 프로젝트 초기부터 성능을 고려하는 것이 좋습니다. 프로토타이핑 및 베타 단계에서 최적화되지 않은 성능을 견딜 수 있다 하더라도, 적어도 프로젝트의 요구 사항을 계속해서 다시 평가하고 성능 예산을 편성해야 합니다. 각 기능과 전체적인 게임에 메모리, CPU, GPU, 스토리지 공간, 네트워크 대역폭 등이 얼마나 필요하며, 타겟 플랫폼에 따라 목표 수치가 다르지는 않은가요? 개발의 모든 단계별로 이러한 질문을 돌아봐야 합니다.

프로파일링에 대해 자세히 알아보려면 Unity 게임 프로파일링 완벽 가이드 를 살펴보세요. 유니티 내외부의 전문가들이 모여 Unity에서의 애플리케이션 프로파일링, 메모리 관리, 전력 소비 최적화 등에 대한 팁 등 자세한 내용을 70여 페이지가 넘는 가이드에 담았습니다.

추상화 대신 구체적인 솔루션 선호

프로그래밍에서 '추상화'란 내부 디테일을 숨기고 단순화된 외부만 보여 주는 일반화된 솔루션입니다. 추상화는 함수, 객체, 라이브러리, 프레임워크, 프로그래밍 언어, 심지어 게임 엔진 등 다양한 형태로 제공됩니다.

어느 정도의 추상화는 합리적이지만 과도한 추상화는 문제를 야기할 수 있습니다.

- 라이브러리, 프레임워크 또는 게임 엔진 등의 바로 사용할 수 있는 추상화를 사용하는 주된 이유는 어렵고 시간이 오래 걸리는 구현 작업을 피하기 위해서입니다. 그러나 이러한 편리함에는 종종 제공된 솔루션과 프로젝트의 구체적인 요구 사항이 일치하지 않는 불편한 상황이 따라옵니다. 결국 제공된 추상화를 용도에 맞게 변형하다가 직접 구체적인 솔루션을 작성하는 것보다 더 많은 작업을 하게 될 수도 있습니다.
- 추상화에는 메모리 사용량이나 CPU 오버헤드가 늘어나는 등 숨겨진 성능 비용이 따라오는 경우가 많으며, 이러한 비용은 결국 사용하지도 않을 기능 때문에 발생하게 됩니다.
- 직접 구현할 때는 나중에 프로젝트에 사용할지도 모른다는 생각에 현재 요구 사항 이상으로 일반화된 추상적 솔루션을 제작해야겠다는 생각에 빠지는 경우가 있습니다. 그러나 대부분의 경우 이렇게 추측하여 작업하면 해결되는 것보다 늘어나는 작업이 많아지며, 최적이 아니거나 최적화하기 어려운 솔루션이 만들어질 때가 많습니다. 실제로 추상화하면 나중에 요구 사항에 더 이상 적합하지 않다는 것을 발견했을 때 변경하는 것이 더 어려워집니다.

대부분의 경우 나중에 요구 사항이 어떻게 바뀔지 걱정하기보다 지금 파악하고 있는 현재의 요구 사항을 해결하는 것이 더 좋습니다. 반복 작업에 익숙해지세요. 문제를 해결하려고 시도하기 전에는 문제를 완전히 이해할 수 없으며, 나중에 실제로 요구 사항이 변경되면 그때 코드를 수정하면 됩니다. 코드 작성에도 적용되는 글쓰기 관련 격언으로 '좋은 글은 고쳐 쓴 글이다'라는 말이 있습니다. 그리고 코드를 쉽게 재작성하는 데 가장 도움이 되는 것은 무엇일까요? 바로 단순함입니다.

추상화에 대한 유혹이 느껴지면 기다리는 것이 가장 좋습니다. 적어도 몇 가지 특정 문제를 해결한 다음에나 그 솔루션들을 하나의 추상화로 결합하는 것을 고려하세요. 리처드 파비안이 자신의 글에서 이렇게 밝힌 것처럼요.

> "데이터 지향 디자인은 현재의 것입니다. 문제의 역사를 보여 주거나 최근에 업데이트된 솔루션이 아니며, 앞으로 발생할 모든 것을 처리할 수 있도록 일반적인 솔루션으로 구성된 미래의 것도 아닙니다. 과거에 집착하면 유연성이 저하되고, 프로그래머는 점쟁이가 아니므로 미래를 내다보는 일은 대개 수확이 없습니다. 미래에 사용할 수 있다는 시스템은 대부분 그런 식이죠."

한 마디로 경고하자면 성급한 추상화는 모든 문제의 근원입니다.

unity.com/kr