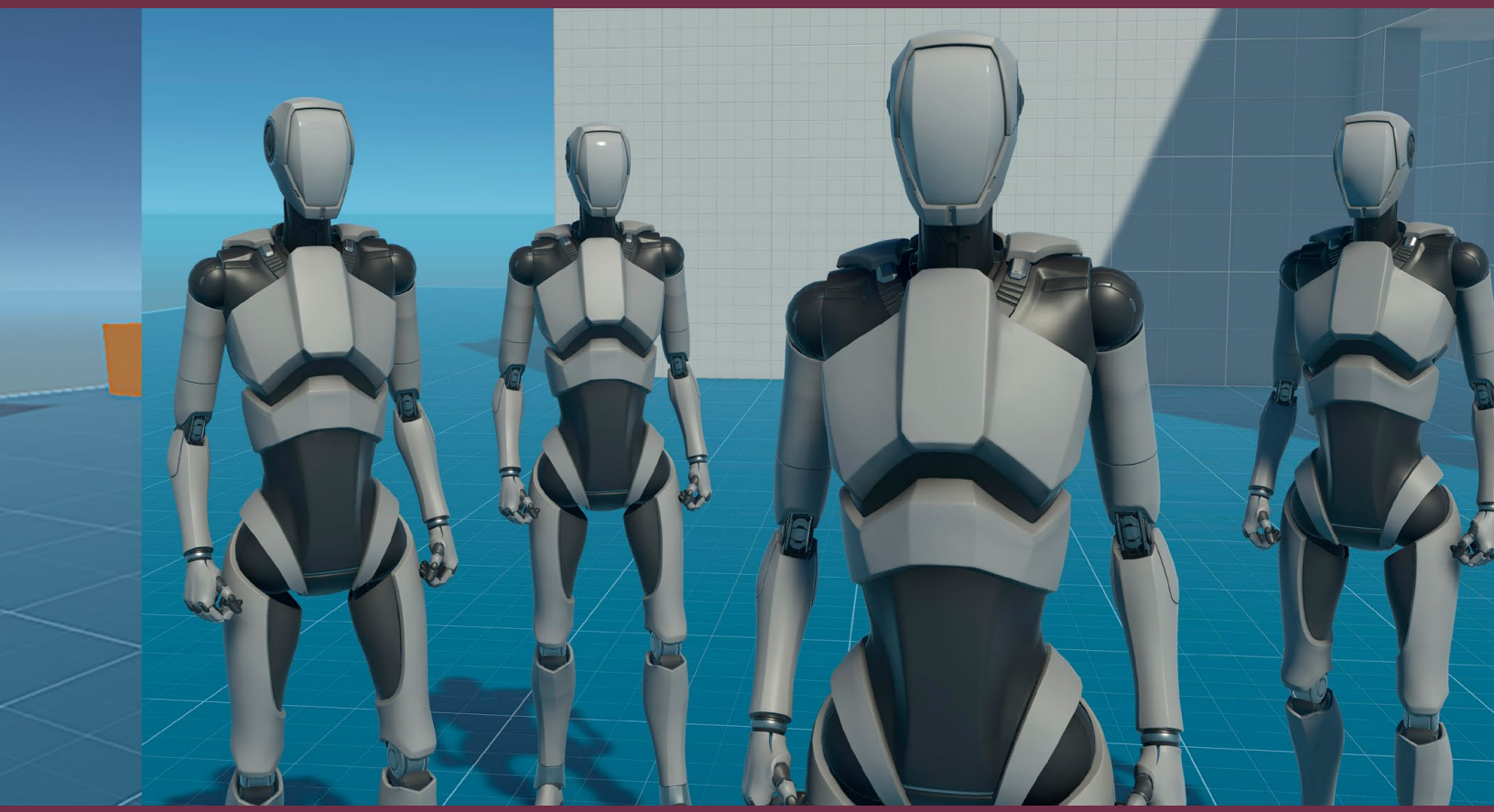
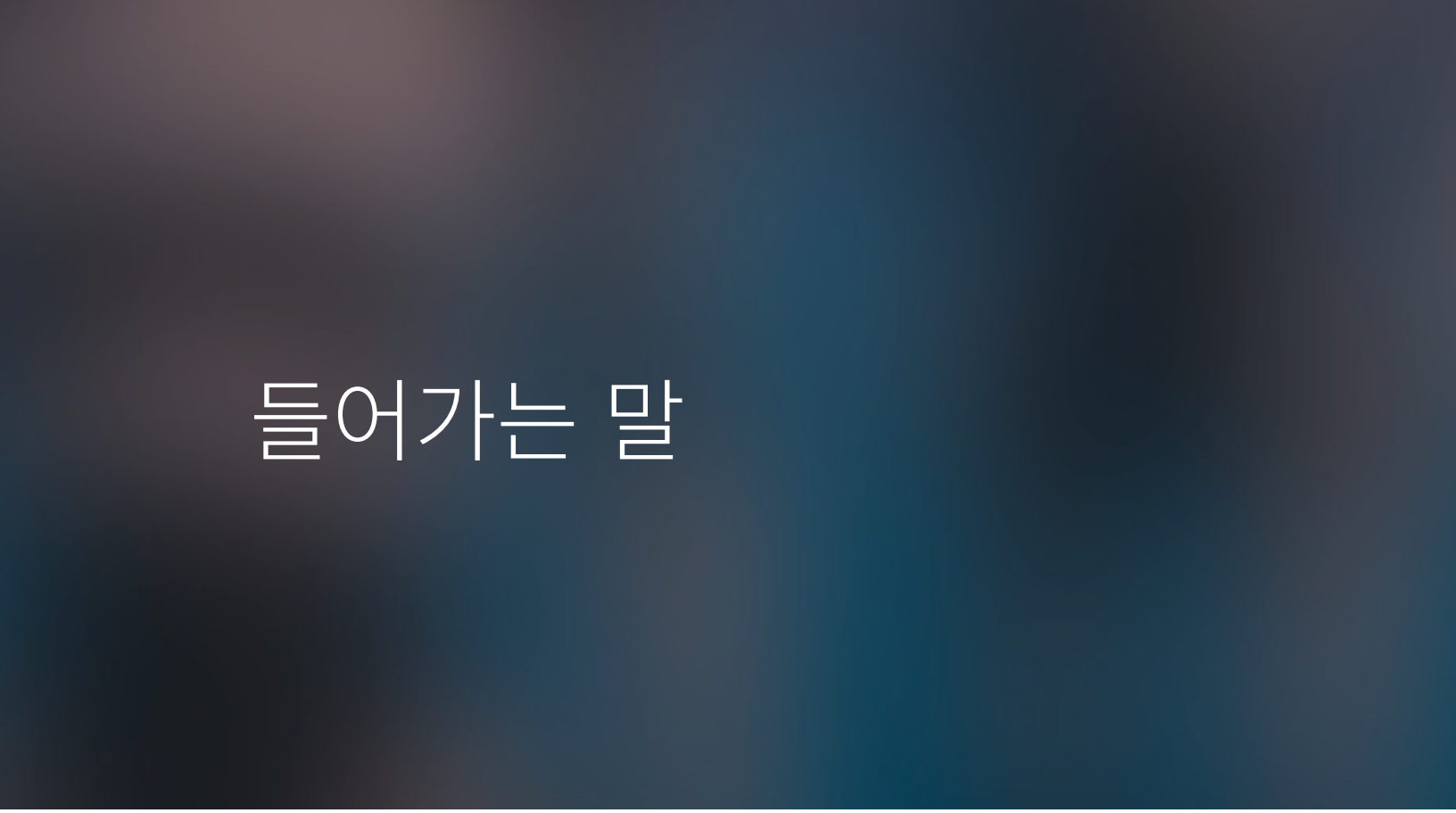
→ 전자책

Unity의 멀티플레이어 네트워킹 소개



Unity®

목차



들어가는 말

온라인 게임에 사람들이 모이면 특별한 일이 생깁니다. 단순한 레이싱 게임이나 RPG가 서로 공유하는 하나의 경험으로 바뀌죠. 온라인에서 벌어지는 일은 예측하기 어렵고, 도전 정신을 자극하며, 무엇보다도 재미있습니다.

서로 힘을 모아 은하계를 파괴하는 악당에 대항하거나, 온라인 슈팅 게임에서 서로 대결하는 등, 네트워크 멀티플레이어 게임을 통해 사람들은 물리적 한계를 넘어 협력하고 경쟁할 수 있습니다. 이러한 집단적 상호 작용을 '멀티플레이어 경험'이라고 정의합니다.

네트워크 게임플레이를 개발하는 일은 그에 상응하는 싱글플레이어 애플리케이션 개발보다 더 복잡할 수 있습니다.

이 가이드는 Unity의 네트워크 툴을 사용해 멀티플레이어 게임 개발을 시작하는 데 도움을 드리기 위해 제작되었습니다. 네트워킹 개념에 대한 기본 지식을 제공하며, Unity 네트워크 샘플을 깊게 분석하기에 앞서 살펴볼 만한 입문서와 같습니다. Unity 에셋 스토어에서 제공하는 Starter Assets – ThirdPerson 패키지를 활용한 기본 활용 사례도 하나씩 살펴보겠습니다.

이 가이드는 독자가 Unity와 C# 개발에는 익숙하지만, 네트워크 개발 경험은 전무하거나 이제 갓 입문했다고 가정하며, 멀티플레이어 개발의 이론적 측면을 빠르게 이해하고 데모 실습에 대비할 수 있도록 고안되었습니다.

이 전자책에 수록된 내용은 다음과 같습니다.

— Unity 멀티플레이어의 핵심 개념 설명

— 다양한 멀티플레이어 시스템과 네트워킹 모델 설명

— Netcode for GameObjects를 사용하여 간단한 예시 설정

멀티플레이어 게임용 Unity 툴의 진화

Unity는 다양한 멀티플레이어 개발 툴 및 솔루션을 제공합니다. Netcode for GameObjects 및 Netcode for Entities 프레임워크와 함께 UGS(Unity Gaming Services) 등이 있으며, UGS는 Game Server Hosting (Multiplay), 음성 및 텍스트 채팅을 위한 Vivox를 제공합니다. 캐주얼 협동 게임을 개발하거나 오픈 월드 MMO를 제작하는 등 어떤 경우에도 Unity를 사용하여 원활하게 멀티플레이어 게임을 개발할 수 있습니다.

Unity 6는 멀티플레이어 게임을 개발할 때 그 어느 때보다도 안정적이고 빠른 연동과 반복 작업, 배포 등이 가능한 새롭고 향상된 기능을 제공합니다. 멀티플레이어 게임을 위해 Unity 6에 추가된 새로운 기능은 다음과 같습니다.

- **Multiplayer Center**: Unity 6의 코어 패키지로 제공되는 Multiplayer Center를 사용하면 멀티플레이어 게임을 더 쉽게 설정하고 개발할 수 있습니다. 새로운 프롬프트와 워크플로는 프로젝트를 시작하는 데 사용할 수 있는 동적 템플릿을 생성하기 전에, 해당 게임의 파라미터와 요구 사항을 활용하여 관련 패키지와 서비스를 제안합니다.
- **Multiplayer 위젯**: Multiplayer 위젯을 사용하면 UGS(Unity Gaming Services)를 멀티플레이어 게임에 더 쉽게 연동할 수 있습니다. 스탠드얼론 패키지로 사용하거나 Multiplayer Center를 통해 사용할 수 있습니다.
- **Multiplayer Tools 패키지**: 이 패키지는 Unity의 멀티플레이어 게임 개발 워크플로와 Netcode for GameObjects 2.0의 성능을 향상하고, Distributed Authority를 지원합니다.
- **Multiplayer Play Mode**: 이 패키지는 디스크상의 같은 에셋에서 최대 4개의 독립된 경량 에디터 프로세스를 실행하여 게임플레이 검증 프로세스를 간소화합니다. 최고 수준의 서버 호스팅 프로젝트에서는 플레이 모드 시나리오를 통해 전용 서버의 빌드를 Multiplay Hosting 서버로 직접 업로드하는 등의 배포 단계를 구성할 수 있습니다.

— **Distributed Authority(베타)**: 2024년 11월을 기준으로 베타 버전인 Distributed Authority는 Netcode for GameObjects에서 사용할 수 있습니다. 이 패키지를 사용하면 클라이언트는 게임 세션 중에 생성된 NetcodeObjects에 대한 소유권을 나눠 가질 수 있습니다. NetCode 시뮬레이션 워크로드가 클라이언트 간에 분산되며, Unity에서 제공하는 고성능 클라우드 백엔드를 통해 네트워크 상태가 조정됩니다.

— **Multiplayer Services SDK**: 이 SDK는 Unity 6로 개발된 게임에 멀티플레이어 요소를 추가할 수 있는 원스톱 솔루션입니다. Relay, Matchmaker 등의 UGS(Unity Gaming Services) Multiplayer Services를 사용하면 Netcode for GameObjects 또는 Netcode for Entities 네트워킹 라이브러리와 연동되는 세션을 통해 게임에서 플레이어 그룹이 상호 작용하는 방식을 정의할 수 있습니다.

Unity 6 멀티플레이어 솔루션에 대해 자세히 알아보려면 다음 리소스를 확인하세요.

— 유나이트 2024: 경쟁형 멀티플레이어 게임 개발 가속

— 유나이트 2024: 멀티플레이어 개발: 스튜디오와 게임을 성공으로 이끄는 방법

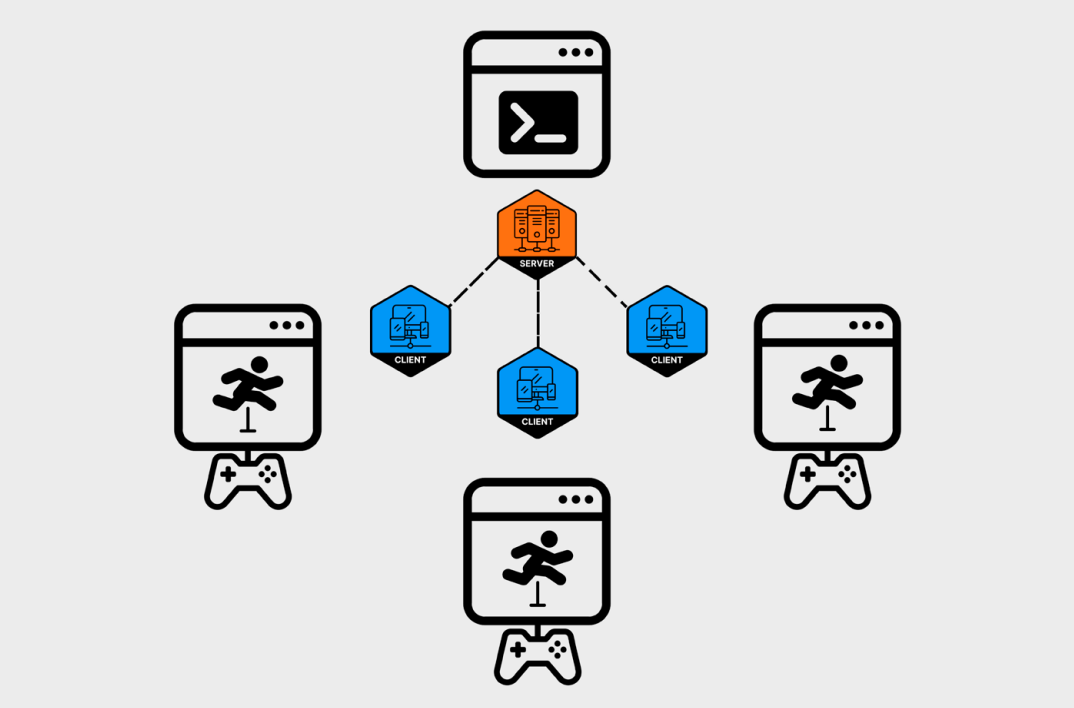
— Unity 매뉴얼: Unity 6의 새로운 멀티플레이어 기능

— Unity 6 출시: 새로운 기능 알아보기

Unity의 다양한 툴은 컨셉과 프로토타이핑부터 출시와 지속적인 운영에 이르는, 멀티플레이어 게임 개발의 전체 라이프사이클을 지원합니다. Unity 생태계 내에서 모든 작업을 처리하거나 팀의 요구 사항에 가장 적합한 툴과 서비스를 선택하세요.

기본 개념

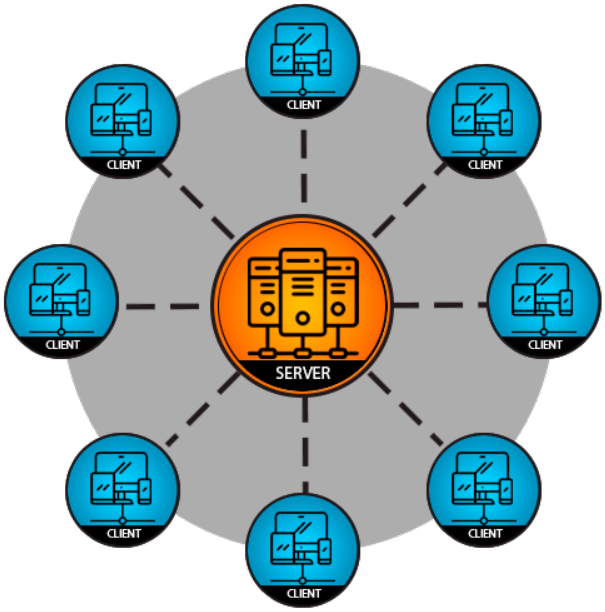
멀티플레이어 게임에서 플레이어는 네트워킹을 통해 중앙 서버에 연결하거나 직접 서로 연결할 수 있습니다. 네트워킹을 통해 플레이어는 실시간으로 데이터를 공유하고 함께 플레이합니다. 동일한 게임 애플리케이션이 동시에 각 플레이어의 기기에서 실행되며, 각 플레이어의 행동은 네트워크를 통해 동기화됩니다.



네트워크로 연결된 멀티플레이어에서는 같은 게임 애플리케이션이 여러 기기에서 동시에 실행됩니다.

일반적인 네트워크 게임의 아키텍처는 **클라이언트**와 **서버**라는 두 가지 주 요소로 구성됩니다. 클라이언트는 플레이어의 기기(PC, 콘솔, 휴대폰 등)에서 실행 중인 게임의 인스턴스입니다. 클라이언트는 게임 그래픽스를 렌더링하고, 오디오를 재생하며, 사용자 입력을 처리하고, 플레이어의 행동에 대한 업데이트를 서버로 전송합니다.

반면에 서버는 게임 상태, 즉 게임 내 모든 요소의 현재 상태를 관리하고, 클라이언트 간의 통신을 처리하며, 게임의 규칙을 적용합니다. 서버는 전용 서버로 운영하거나, 게임 개발자가 헤드리스 머신으로 운영하거나, 플레이어 중 한 명이 서버 역할도 맡는 플레이어 호스팅 서버로 운영할 수 있습니다.



클라이언트-서버 아키텍처

서버는 클라이언트에서 입력을 수신하여 게임 로직을 처리한 후 업데이트를 클라이언트에 전송합니다. 게임 상태에 대한 최종 권한을 보유하는 것은 서버지만, 게임플레이의 반응성을 높이기 위해 클라이언트가 로컬 복사본을 보유합니다. 그러면 지속적인 동기화가 이루어져 모든 플레이어에게 게임이 실시간으로 최대한 일관성을 유지할 수 있습니다.

헤드리스 서버

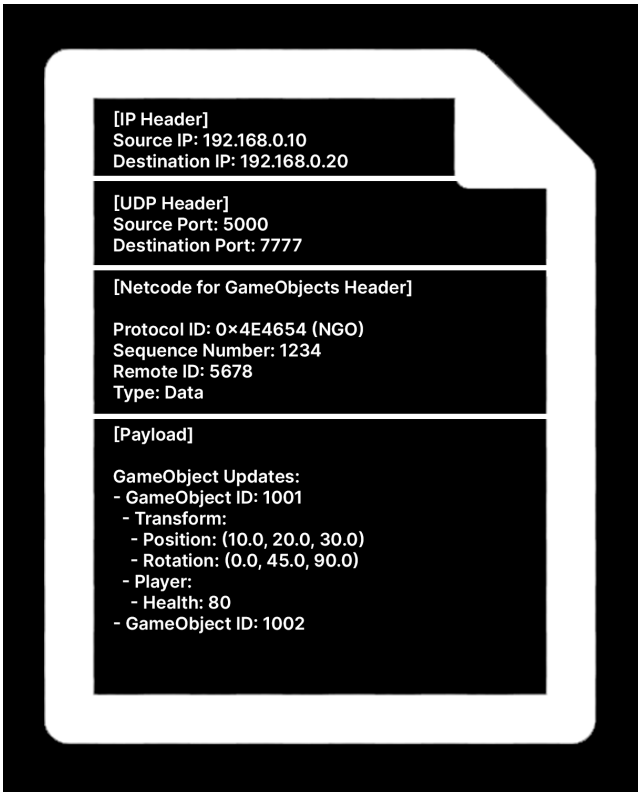
헤드리스 서버는 그래픽 사용자 인터페이스 없이 작동하며, 백엔드 작업에만 집중하는 서버입니다.

그래픽스를 렌더링하지 않으므로 더 쉽게 확장/축소할 수 있고, 주로 전용 하드웨어 또는 클라우드 환경에 배포됩니다. 자세한 내용은 **네트워크 토폴로지**의 **전용 게임 서버**를 참조하세요.

UDP 패킷

클라이언트와 서버는 UDP(사용자 데이터그램 프로토콜)와 같은 표준 인터넷 프로토콜을 사용하여 데이터 패킷을 교환하는 방식으로 통신합니다. 실시간 애플리케이션, 특히 1인칭 슈팅 게임처럼 진행 속도가 빠른 게임에서는 UDP가 TCP(전송 제어 프로토콜)보다 선호됩니다. UDP는 게임이 통신에서 다양한 측면의 우선순위를 지정할 수 있는 완벽한 통제권을 제공하기 때문입니다. TCP와 달리 UDP에서는 수신자의 확인이 필요하지 않습니다. 따라서 UDP는 기본적으로 더 효율적이고 유연하지만, 수신자 확인 같은 기능이 필요한 경우, 애플리케이션에는 해당 기능을 처리해야 하는 부담이 생깁니다.

각 UDP 패킷은 **헤더**와 **페이로드**로 구성됩니다. UTP 페이로드에는 프로토콜 관련 추가 섹션이 포함되어 있고, 각 섹션에는 고유한 헤더와 페이로드가 있습니다. 이러한 중첩을 네트워크 용어로 **캡슐화**라고 합니다.



UDP 패킷의 간단한 예시

IP 및 UDP 헤더는 고정된 크기를 가지며, 발신자와 수신자의 주소(IP 주소 및 포트)와 같은 중요한 메타데이터를 포함하고 있습니다. 페이로드의 크기, 구조, 내용은 게임과 컨텍스트에 따라 다릅니다. 예를 들어 페이로드에는 플레이어 입력이나 특정 시점의 게임 상태 스냅샷이 있을 수 있습니다.

UDP 패킷은 실시간 애플리케이션에 꼭 필요한 낮은 지연과 빠른 속도를 제공하지만 이렇게 속도가 빠르면 신뢰성이 떨어진다는 단점이 있습니다. UDP 프로토콜은 안정성, 순서 지정(ordering) 또는 혼잡 제어를 위한 메커니즘을 제공하지 않습니다. 인터넷을 통해 오가는 패킷에는 다음과 같은 오류가 발생할 수 있습니다.

— **패킷 손실:** 패킷이 손실되어 목표 지점에 도달하지 못할 수 있습니다. 네트워크 혼잡, 하드웨어 고장 등 여러 문제로 이러한 현상이 발생할 수 있습니다.

— **중복:** 패킷은 중복될 수 있고, 이에 따라 동일한 패킷이 수신자에게 여러 차례 도달할 수 있습니다. 네트워크 하드웨어 또는 소프트웨어 구성 오류로 이러한 현상이 발생할 수 있습니다.

— **순서 재지정(reordering):** 패킷이 전송된 순서와 다르게 수신자에게 도달할 수 있습니다. 패킷이 여러 경로를 통해 목표 지점으로 전송될 경우 각 경로의 지연이 서로 다르면 이러한 현상이 발생할 수 있습니다.

— **손상:** 패킷의 내용이 전송 중에 변경되어 데이터를 사용하지 못하게 될 수 있습니다.

UDP와 TCP 비교

네트워크 프로토콜은 네트워크를 통해 데이터를 주고받는 방식을 규정하는 일련의 규칙입니다. 프로토콜은 연결을 설정하고, 메시지의 포맷을 정하고, 오류를 처리하고, 데이터를 전송하는 방식을 정의합니다.

게임 개발에서는 TCP(전송 제어 프로토콜)에 비해 빠르고 가벼운 특성이 있는 UDP가 선호됩니다. 안정적이며 웹 브라우징에 적합한 TCP는 손실되거나 순서가 뒤바뀐 패킷을 다시 전송하여 데이터가 순서대로 전달되도록 처리하지만, 이러한 작업으로 인해 지연이나 끊김이 발생할 수 있으므로 실시간 게임에는 적합하지 않습니다.

반면에 UDP는 실시간 성능을 우선하도록 데이터 손실을 일부 허용합니다. 즉, 중요하지 않은 일부 데이터가 손실되더라도 게임을 60fps 이상의 속도로 끊김 없이 원활하게 실행할 수 있습니다. UDP는 응답 속도와 간헐적 데이터 손실 사이의 균형을 잡는 데 보다 뛰어나므로, 네트워크 멀티플레이어 게임에 적합합니다.

UDP의 불안정성 문제를 완화하는 기법은 다음과 같습니다. TCP의 빌트인 기능과는 달리 이러한 기술은 실시간 게임에서의 활용을 위해 미세 조정할 수 있습니다.

기법	기능
순서 번호	각 패킷에는 고유의 오름차순 번호가 있습니다. 수신자는 이 정보를 사용하여 누락되거나 순서가 뒤바뀐 패킷을 감지합니다.
ACK(확인) 및 ACK 비트마스크	패킷에는 마지막으로 수신된 패킷의 시퀀스 번호가 포함되므로, 발신자는 이를 통해 어떤 패킷이 전달되었는지 알 수 있습니다. ACK 비트마스크는 여러 패킷의 상태를 동시에 추적하여 손실된 패킷을 더 빠르게 감지할 수 있습니다.
RTO(재전송 시간 초과) 조정	이 기술은 TCP와 비슷한 방법으로 왕복 시간을 측정합니다. 예를 들어 왕복 시간을 측정한 후 클라이언트측 보간 또는 클라이언트측 예측을 조정하는 데 그 값을 사용할 수 있습니다.
시간 초과	일정 시간 안에 응답 확인이 도착하지 않으면 해당 패킷은 손실된 것으로 간주됩니다.

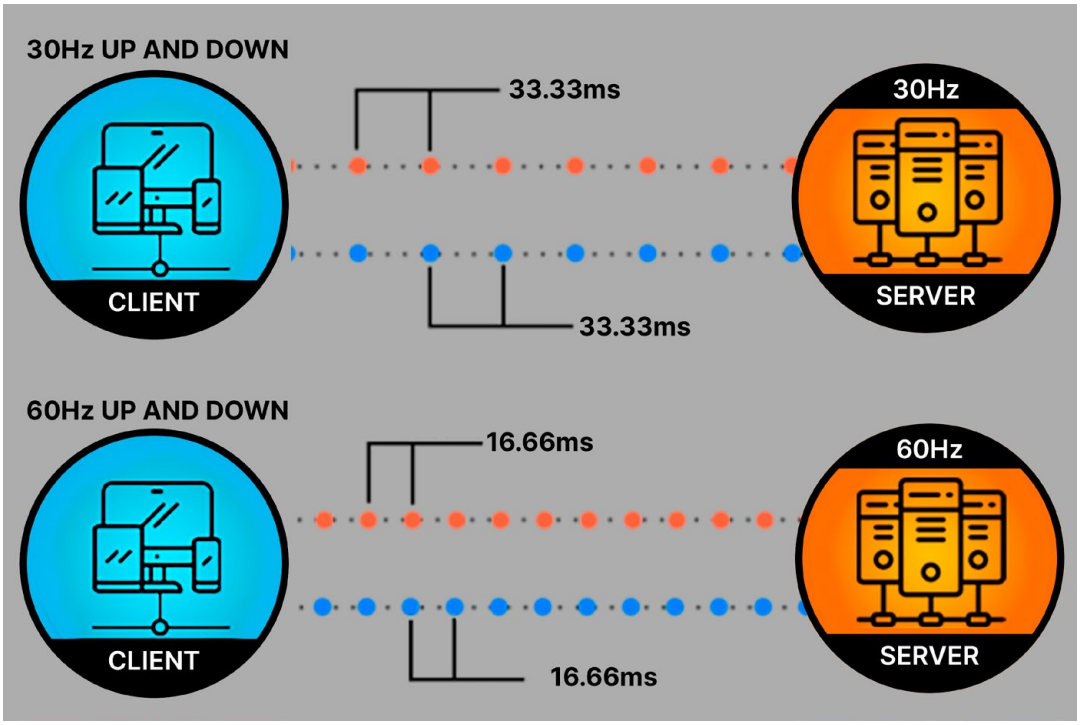
틱 및 업데이트

멀티플레이어 게임에서 서버는 화면 출력 없이 핵심 게임 로직, 물리 시뮬레이션 및 기타 게임플레이 기능을 처리합니다.

서버는 전역 게임 상태만을 처리하므로 입력은 클라이언트에서 수신합니다. 이는 싱글플레이어 게임이 로컬 플레이어에게서 입력을 받는 것과 유사합니다. 마우스와 키보드 대신 서버가 이러한 입력을 처리하여 '권한 게임 상태'를 유지합니다. 여기에는 현재 플레이어 위치와 오브젝트 상태부터 물리 계산 및 게임 진행 상황에 이르는 모든 사항이 포함됩니다.

서버 측 처리의 핵심은 **서버 틱**입니다. 서버 틱이란 서버가 수신한 입력에 따라 게임 상태를 업데이트하는 주기를 말합니다. 틱은 틱 레이트라는 일정한 간격으로 발생하며, 틱 레이트는 헤르츠(Hz) 또는 초당 틱 수로 측정됩니다. 게임 월드가 업데이트되는 빈도는 바로 이 틱 레이트로 결정됩니다.

반면에 **업데이트 속도**는 클라이언트가 서버와 데이터를 교환하는 빈도를 말합니다. 업데이트 속도가 높을수록 게임의 응답 속도는 향상되지만, 더 높은 대역폭과 처리 성능이 필요합니다. 업데이트 속도는 일반적으로 클라이언트의 네트워크 환경과 컴퓨팅 리소스의 제한을 받습니다.



틱 레이트와 업데이트 속도 비교

틱 레이트가 높을수록 게임의 응답 속도는 향상되지만 서버 리소스를 과도하게 사용할 수 있습니다.
마찬가지로, 업데이트 속도가 높을수록 상호 작용은 더 원활해지지만, 그 대가로 데이터 전송량이 많아집니다.

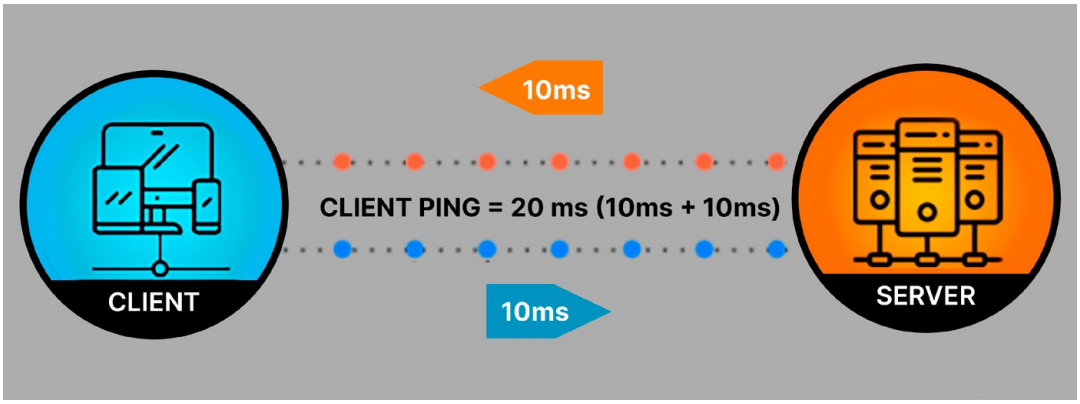
원활하고 반응이 빠른 멀티플레이어 경험을 구현하려면 틱 레이트와 업데이트 속도의 적절한 균형을 찾아야 합니다.

실제 틱 레이트는 게임플레이 요구 사항에 따라 달라질 수 있습니다. 진행 속도가 빠른 1인칭 슈팅 게임은 플레이어의 빠른 움직임과 순식간의 사격을 반영할 수 있도록 60Hz 이상의 틱 레이트로 실행되는 경우가 많습니다. 반면에 실시간 빠른 반사 신경에 의존하지 않는 전략 게임은 30Hz의 틱 레이트로도 충분할 수 있습니다. 한편, 대규모 전략 MMO에서는 많은 동시 접속자 수를 지원하기 위해 10Hz라는 상당히 낮은 틱 레이트를 사용할 수 있습니다.

지연

지연은 데이터가 소스에서 목표 지점까지 이동하는 데 걸리는 시간을 말합니다. 네트워크 지연은 패킷이 목표 지점까지 이동한 후 응답과 함께 돌아오는 데 걸리는 시간을 측정한 **RTT(왕복 시간)**로 나타냅니다.

대체로 사용자는 약 200ms의 지연이 발생하면 게임플레이가 원활하지 않다고 느낍니다. 게임의 종류에 따라 지연에 대한 관용도도 다릅니다. 예를 들어 1인칭 슈팅 게임은 100ms 미만의 지연에서 가장 원활한 플레이를 기대할 수 있지만, 실시간 전략 게임에서는 500ms에 달하는 높은 지연이 허용되기도 합니다.



네트워크 지연은 왕복 시간으로 나타냅니다.

지연이 낮으면 플레이어는 응답성 높은 경험을 즐길 수 있습니다. 멀티플레이어 게임에서는 사용자의 행동과 예상되는 결과 사이의 지연을 최소화하는 것이 좋습니다. 지연이 높으면 게임플레이 중에 지연되는 현상이 눈에 띄게 발생합니다.

지연은 네트워크가 아닌 다른 요소에서 발생하기도 합니다. 예를 들어 사용자 입력을 감지하는 작업에 지연이 발생하거나 렌더 파이프라인에서 일시적으로 멈출 수도 있습니다. Vsync가 원인이 되기도 합니다. 이 기능으로 화면 끊김을 방지할 수 있지만, 대신 지연이 추가로 발생하게 됩니다.

대부분 네트워크 자체가 지연의 주요 원인으로 작용합니다. 이는 다양한 유형의 지연을 수반하게 됩니다.

— **처리 지연:** 라우터가 패킷 헤더를 읽고 패킷을 목표 지점으로 전달하려면 시간이 필요합니다. 이 지연은 보통 미미하지만 여러 번 발생하며 누적될 수 있습니다.

— **전송 지연:** 패킷을 네트워크에 보내는 데 필요한 시간으로, 패킷 크기의 영향을 직접적으로 받습니다. 이 지연은 대역폭이 낮은 최종 사용자 네트워크에서 더 두드러집니다.

— **대기열 지연:** 혼잡 현상 또는 인터페이스 용량의 제한 때문에 패킷이 대기열에 있는 경우, 이러한 지연이 지연을 크게 높일 수 있습니다.

— **전파 지연:** 신호가 네트워크를 통해 전달되려면 시간이 걸립니다. 이러한 유형의 지연이 발생하는 원인은 주로 서버와 사용자 사이의 물리적 거리, 물리적 매체(광섬유, 구리, 공기), 신호 유형(전자파, 광파, 전파)에 있습니다.

특히 인터넷 기반 게임에서는 약간의 지연이 불가피하지만, 그 영향을 최소화하기 위한 여러 기술(예: 예상, 예측, 보간)이 마련되어 있습니다. 이 중 몇 가지를 나중에 살펴보겠습니다.

기타 네트워킹 용어

지연과 관련해서 접할 수 있는 기타 용어는 다음과 같습니다.

핑: 기본 메시지를 주고받아 네트워크 응답 속도를 측정합니다. 간소화된 버전의 RTT(왕복 시간)라고 간주할 수 있습니다.

지터: 네트워크 상태의 변동으로 인한 RTT의 배리에이션으로, 지연 완화에 영향을 미치며 패킷이 순서가 뒤섞인 채 도착하는 현상을 유발합니다.

대역폭: 일정 시간 이내에 네트워크를 통해 전송할 수 있는 데이터의 양입니다. 상태 데이터를 대량으로 전송해야 하는 게임이라면 높은 대역폭이 중요할 수 있습니다.

그 밖의 관련 용어들은 멀티플레이어 네트워킹 용어의 용어집 페이지에서 찾아볼 수 있습니다.

네트워크 동기화

동기화된 상태를 유지하기 위해 클라이언트와 서버는 계속해서 메시지를 교환하는 방식으로 플레이어 간에 일관된 게임 상태를 지속합니다. 일반적으로 클라이언트는 사용자 명령을 대개 60Hz, 약 16밀리초라는 높은 빈도로 서버에 전송합니다. 이러한 명령에는 행동이나 입력, 이를테면 마우스나 게임패드의 움직임, 점프하고 발사할 때 누르는 버튼 등이 있습니다.

서버가 클라이언트 명령을 수신하고 처리한 후, 게임 월드에 대한 업데이트를 클라이언트에 전송합니다. 게임은 플레이어의 입력에 빠르게 반응할수록, 반응성이 높다는 느낌을 줍니다.

서버의 틱 레이트, 클라이언트의 업데이트 속도, 클라이언트의 프레임 속도는 각각 용도가 다르므로 이들을 일치시킬 필요는 없습니다. 사실, 완벽한 동기화는 보통 이루어지지 않습니다. 서버와 클라이언트는 지속적으로 변화하며 동적 데이터의 연속적인 스트림을 교환합니다. 동기화의 목표는 불일치를 줄여 모든 클라이언트가 동시에 플레이하는 듯한 착각을 일으키는 것입니다.

네트워크 동기화 기법

상태 동기화: 서버에서 클라이언트로 네트워크 오브젝트의 상태를 주기적으로 전송합니다. 게임 업데이트의 빈도는 게임 장르(예: 경쟁 중심의 슈팅 게임과 협동 전략 게임)별 요구 사항에 따라 달라질 수 있습니다.

RPC(원격 프로시저 호출): 서버 또는 다른 클라이언트에서 함수를 원격으로 호출합니다. RPC를 사용하면 플레이어 입력 전송, 특정 동작 요청, 일회성 게임 이벤트 트리거 등의 클라이언트와 서버 간 통신을 수행할 수 있습니다.

대역폭 관리: 성능에 큰 영향을 미칠 수 있습니다. 동기화는 대역폭을 사용하므로 네트워크를 통해 전송되는 데이터를 줄이기 위해 다음과 같은 전략을 구현하세요.

- **데이터 컬링:** 불필요한 업데이트를 제외하고 게임플레이에 필요한 업데이트에만 집중하여 네트워크 트래픽을 줄입니다. 예를 들어 중요한 이동 축만 동기화하거나 VFX와 애니메이션을 지속적으로 동기화하는 대신 이벤트를 사용하여 로컬에서 트리거할 수 있습니다. 네트워크 트래픽이 감소하면 게임 성능이 향상될 수 있습니다.
- **델타 압축**: 델타 인코딩이라는 용어로도 사용됩니다. 이 기술을 사용하면 서버가 마지막 업데이트 이후의 변경 사항(델타)만 전송할 수 있습니다. 그러면 클라이언트는 이 델타만 로컬 게임 상태에 적용하여 서버와의 동기화를 유지합니다.
- **관심 기반 관리**: 여러 기준에 따라 데이터 동기화의 우선순위를 결정합니다. 공간 관련성은 플레이어와의 거리 및 가시성에 따라 오브젝트의 우선순위를 결정합니다. 나이(Age) 또는 신선도(Staleness)는 최근에 전송되지 않은 오브젝트 또는 데이터를 우선하며, 업데이트될 때까지 그 우선순위를 높입니다. 상호 작용은 최근 플레이어와 상호 작용했거나 곧 상호 작용할 가능성이 높은 오브젝트에 중점을 둡니다.

이러한 기술을 사용하면 더 효율적으로 네트워크 성능을 최적화하고 더 부드럽고 효율적인 게임플레이 경험을 구현할 수 있습니다.

네트워크 토폴로지

네트워크 토폴로지란 멀티플레이어 환경에서 기기들이 어떻게 연결되고 통신하는지를 정의합니다. 각 네트워크 모델에는 저마다의 장점과 단점이 있습니다. 어떤 네트워크 모델을 사용할지는 게임의 유형, 게임 상태에 대해 원하는 제어 수준, 그리고 서버 인프라에 사용할 수 있는 리소스에 따라 달라집니다.

토폴로지는 게임의 아키텍처, 성능, 전반적인 플레이어 경험에 영향을 미칠 수 있습니다. Netcode for GameObjects는 클라이언트-서버와 분산 권한이라는 두 가지 토폴로지를 지원합니다. 무슨 의미인지 알아보도록 하겠습니다.

클라이언트-서버 토폴로지

클라이언트-서버 토폴로지는 클라이언트 기기와 중앙 서버가 역할을 분담하여 성능을 최적화하고 게임을 효과적으로 관리할 수 있는 일반적인 네트워크 모델입니다.

클라이언트는 플레이어의 게임 인스턴스를 의미하며, 로컬 입력, 렌더링, 게임 상태의 부분 시뮬레이션을 처리합니다. 클라이언트는 캐릭터의 움직임과 같은 로컬 입력을 서버로 전송하고, 그에 대한 응답으로 업데이트를 수신합니다.

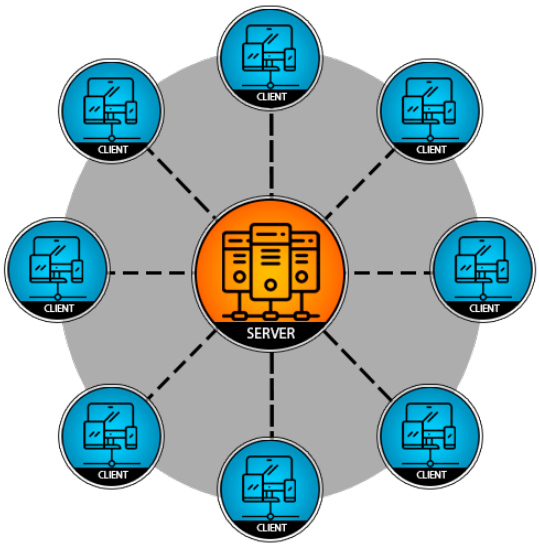
서버는 게임 월드의 최종적이고 정확한 표현을 유지하고, 플레이어의 입력을 처리하고, 게임의 규칙을 적용합니다. 이 중앙 서버는 충돌을 해결하고 동작을 검증하여 모든 플레이어가 일관되고 공정한 경험을 누릴 수 있도록 합니다. 이러한 설정에서는 중앙에서 게임 상태를 관리하므로 부정행위 방지에도 도움이 됩니다.

클라이언트와 서버는 인터넷이나 LAN(근거리 통신망)을 통해 서로 통신할 수 있습니다. 오프라인 LAN 게임에서는 인터넷에 접속할 필요 없이 로컬 네트워크를 통해 동일한 물리적 공간에 있는 여러 기기를 연결합니다. 이 설정은 인터넷을 우회하며, 기기가 서로 가까워 지연을 최소화하고 높은 수준의 보안과 안정적인 연결을 제공합니다. 따라서 LAN 파티, e스포츠 토너먼트, 인터넷 연결이 불안정한 환경에 적합합니다.

클라이언트-서버 토폴로지 내에는 두 가지 유형의 서버가 있습니다.

전용 게임 서버

전용 게임 서버는 데이터만 처리하고 플레이어로 참여하지 않는 독립된 엔티티입니다. 네트워크 게임의 주요 시뮬레이션과 플레이어 상호 작용을 모두 처리하면서도 최고의 성능을 제공할 수 있습니다.



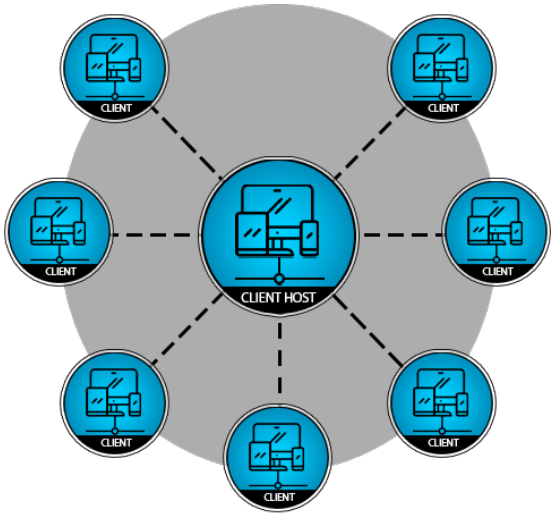
전용 게임 서버는 모든 주요 게임 시뮬레이션을 처리합니다.

부정행위를 최대한 최소화해야 하는 게임에서는 전용 서버가 꼭 필요합니다. 그러나 이 설정에서는 모든 플레이어 상태 변경이 서버에서 처리된 후에 다른 클라이언트로 전달되어야 하므로 통신 지연이 발생할 수 있습니다.

전용 서버는 특히 1인칭 슈팅 게임과 같이 성능이 중요한 경쟁형 게임에 적합합니다. 공정성을 유지하고 방해 행위를 줄이려면 전용 서버가 필요할 수 있습니다.

클라이언트 호스트 리슨 서버

클라이언트 호스트 리슨 서버는 서버와 클라이언트의 역할을 모두 수행하며, 호스트도 게임을 플레이할 수 있도록 지원합니다. 이러한 방식을 사용하면 비용도 절감되고 네트워크를 통해 패킷을 전송할 필요가 없으므로 호스트가 지연 측면의 이점을 누릴 수 있습니다.



클라이언트 호스트 리슨 서버는 서버와 클라이언트의 역할을 모두 수행합니다.

이 구조는 같은 머신에서 게임 서버를 실행하는 작업과 호스트 플레이어의 시각적 출력을 생성하는 작업을 동시에 수행하므로 서버 성능이 저하되는 경우가 많습니다. 또한 호스팅 클라이언트가 가정용 인터넷 연결을 통해 서비스를 제공하므로 원격 데이터 센터에 위치한 전용 서버를 사용하는 것보다 속도가 저하될 수 있습니다. 가정용 인터넷 서비스 제공업체에서는 일반적으로 다운로드 성능을 업로드 성능보다 우선시하기 때문입니다.

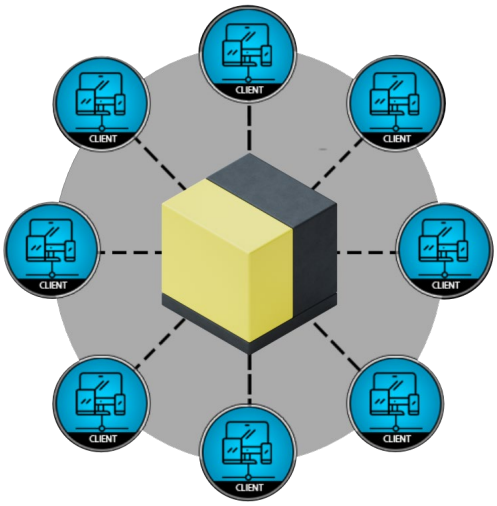
분산 권한

분산 권한 네트워크 모델은 게임 상태를 제어하고 관리하는 권한을 참여하는 모든 클라이언트 간에 분할합니다. 각 클라이언트는 게임 내 오브젝트의 상태를 부분적으로 소유, 트래킹, 관리할 책임이 있고 해당 오브젝트를 자율적으로 생성하고 관리할 수 있습니다.

중앙의 경량 서버에서 오브젝트 상태의 변화를 모니터링하고 네트워크 트래픽의 라우팅을 관리하지만 게임 자체를 시뮬레이션하지는 않습니다.

이 토폴로지는 게임에서 발생하는 모든 동작을 처리하는 데 중앙 서버가 필요하지 않고, 각 클라이언트가 자체 오브젝트에 대해 권한을 가지므로 네트워크 왕복이 감소하여 비용 절감, 입력 지연 감소 등 다양한 이점을 제공합니다. 예를 들어 이동, 공격, 기타 게임 입력과 같은 동작을 서버의 허가를 기다릴 필요 없이 로컬에서 처리할 수 있습니다. 이렇게 하면 플레이어가 더 빠르게 피드백을 받을 수 있어, 게임이 더욱 즉각적으로 반응하는 것처럼 느껴집니다. 하지만 하나의 권한 서버가 모든 동작을 검증하는 것이 아니므로 부정행위에 더 취약할 수 있습니다.

분산 권한은 정교한 시뮬레이션이나 경쟁 요소가 중요한 게임에는 적합하지 않지만, 상호 작용의 중요성이 낮은 게임에는 잘 어울립니다.



분산 권한 모델은 게임을 제어하고 관리하는 권한을 분할합니다.

관련 유나이트 2024 세션에서 Unity Netcode for GameObjects의 새로운 Distributed Authority 패키지(베타)에 대한 자세한 내용을 알아볼 수 있습니다.

로컬 또는 카우치 멀티플레이어

로컬 멀티플레이어 게임에서는 동일한 물리적 위치에 있는 둘 이상의 플레이어가 동일한 화면에서 플레이할 수 있는 단일 클라이언트 런타임 인스턴스를 사용합니다. 소셜 게임 시나리오에 적합한 이 설정에서는 네트워킹이 없어도 플레이어가 서로 직접 상호 작용할 수 있습니다. 파티 게임과 협동 모드에 널리 사용되며 친구와 가족이 함께 플레이할 수 있는 간편한 수단을 제공합니다.

P2P(피어 투 피어)

각 기기가 클라이언트와 서버의 역할을 모두 수행하며, 플레이어들이 서로 직접 연결할 수 있습니다. 분산 권한과 유사하지만, 경량 서버가 없습니다. 이 방식은 중앙 서버의 필요성을 줄여 비용과 복잡도를 낮춥니다. 그러나 게임 상태와 보안을 관리하는 중앙 기관이 없으므로 공정성과 일관된 지연을 유지하기 어려울 수 있습니다.

권한 서버란?

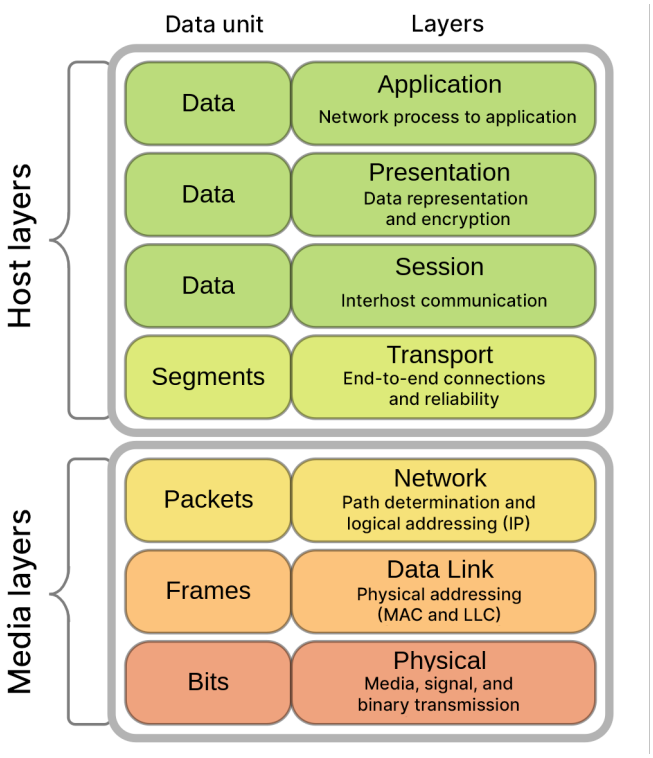
권한 서버는 게임 상태 및 로직을 중앙에서 제어하는 서버 구성을 의미합니다. 이름에서 알 수 있듯이, 네트워크 게임에서 최종적인 권한을 가진 서버입니다.

권한 서버는 게임에서 일어나는 일에 대한 권한을 플레이어 머신에 분할하는 대신, 전체 게임 시뮬레이션을 자체 실행하고 게임에서 일어나는 일을 결정합니다. 클라이언트가 이 서버에 입력을 전송하면 서버는 게임을 업데이트하고 최신 게임 상태를 반환합니다.

이 서버는 게임 규칙을 적용하고 충돌을 해결하기도 합니다. 권한 서버는 네트워크 게임 로직을 구현하는 간단한 방식 중 하나로, 부정 사용자에게 악용될 위험이 가장 적어 모든 플레이어가 동일한 경험을 누릴 수 있게 합니다.

네트워크 스택

프로토콜 스택 또는 네트워크 스택은 일반적으로 네트워크를 통한 데이터 전송을 지원하기 위해 다양한 통신 프로토콜을 구현하는 소프트웨어입니다. 다음과 같이 층층이 쌓아 올린 케이크와 같은 구조입니다.



네트워크 스택(출처: Wikipedia)

각 레이어는 오직 바로 위와 아래에 위치한 레이어와만 상호 작용하며, 이러한 구조는 모듈성을 높이고 네트워크 관리의 복잡성을 줄여줍니다.

애플리케이션 레이어: 스택의 상위 수준으로, 여기서 Unity 개발의 대부분이 이루어집니다. Netcode for GameObjects 또는 Netcode for Entities와 같은 패키지는 개발자가 멀티플레이어 기능 구현에 집중할 수 있도록 하위 수준 네트워킹의 복잡성을 추상화합니다.

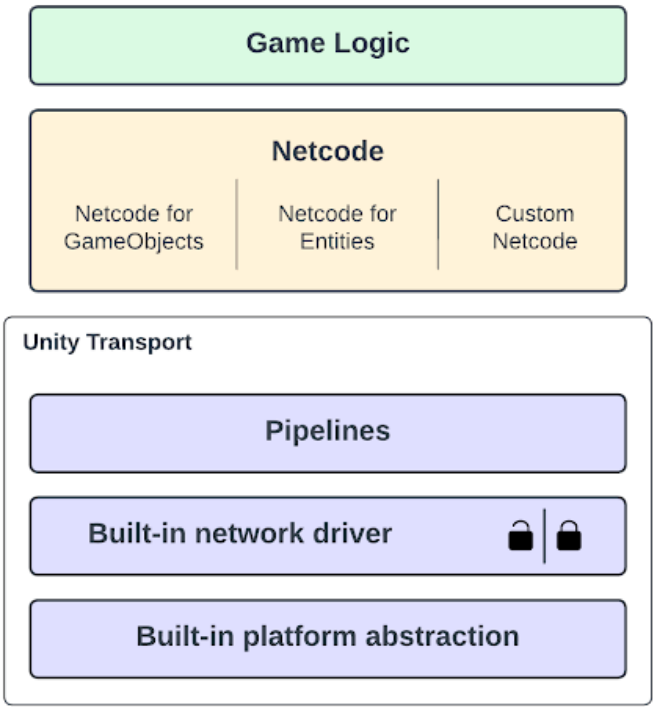
전송 레이어: 전송 레이어는 안정적인 데이터 전송, 오류 감지 및 수정, 플로 제어, 네트워크 내 기기 간의 엔드투엔드 통신을 담당합니다. 데이터 패킷의 분할과 재구성을 지원하고, 오류 복구와 데이터 무결성을 위한 메커니즘을 제공합니다.

네트워크 레이어: 네트워크 레이어는 서로 다른 네트워크에 있는 네트워크 기기 간에 데이터 패킷을 라우팅하는 역할을 하며, IP(인터넷 프로토콜)와 같은 네트워크 인프라 및 프로토콜에 기반하여 통신을 처리합니다.

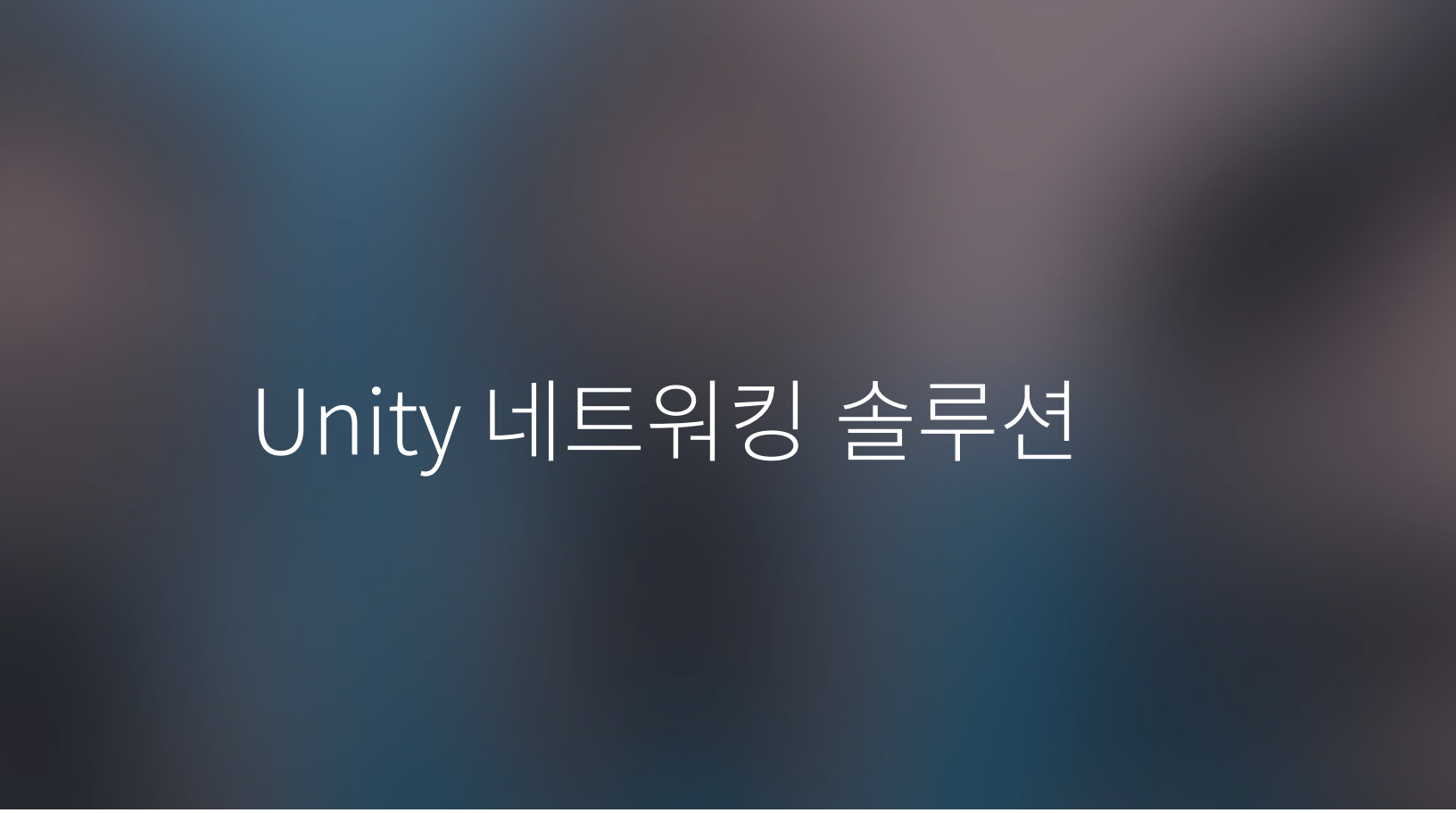
데이터 링크 및 물리 레이어: 이더넷 또는 Wi-Fi와 같은 네트워크 매체를 통해 데이터 패킷의 물리적 전송을 직접 처리합니다. 데이터 링크 레이어와 물리 레이어는 일반적으로 운영 체제와 네트워크 하드웨어에서 처리합니다.

Unity 개발자는 주로 상위 수준의 응용 레이어에서 게임 오브젝트 동기화, 게임 상태 관리, 플레이어 상호 작용 처리 등의 멀티플레이어 기능을 구현하는 작업을 수행합니다.

일반적으로 애플리케이션에 특정 요구 사항이 없는 한, 하위 레이어에 대해서는 걱정할 필요가 없습니다. Unity를 사용하면 네트워크 스택이 다음과 같은 형태로 단순화됩니다.



Netcode 개발 레이어



Unity 네트워킹 솔루션

Unity는 다양한 장르와 규모의 멀티플레이어 게임을 개발할 수 있는 종합적인 솔루션을 제공합니다. 게임 유형별로 고유의 네트워킹 요구 사항이 있으므로 적절한 Netcode 솔루션을 선택하는 것은 매우 중요합니다.

게임 장르, 규모, 경쟁 수준, 원하는 네트워크 레이어 제어 수준 등을 고려해야 합니다. 캐주얼 게임에서는 단순성과 비용 효율성을 우선시하는 경우가 많지만, 경쟁형 게임에서는 공정성과 응답성을 충분히 갖추기 위한 정교하고 안정적인 네트워크 관리가 필요합니다.

캐주얼 협동 게임 개발이나 경쟁형 액션 게임 개발 등 어떤 경우에도 Unity는 강력한 Netcode 패키지와 부가 서비스를 통해 개발자의 요구 사항을 충족합니다.

Netcode for GameObjects

캐주얼 협동 멀티플레이어 게임의 경우 NGO(Netcode for GameObjects) 패키지를 사용하는 것이 좋습니다. 이 고수준 솔루션은 모든 플레이어의 게임 상태를 더 쉽게 관리할 수 있도록 네트워킹 로직을 추상화하여 멀티플레이어 게임 개발을 간소화합니다.

멀티플레이어 개발이 처음이라면 NGO로 시작하는 것을 권장합니다.

Netcode for GameObjects의 주요 기능은 다음과 같습니다.

— **NetworkObject:** 게임 내에서 네트워크를 통해 동기화되는 모든 오브젝트를 의미하며, 오브젝트의 생성, 제거, 소유권을 처리합니다.

- **NetworkBehaviour:** 네트워킹 기능을 제공하는 데 특화된 MonoBehaviour입니다. 네트워크 이벤트를 처리하기 위한 빌트인 콜백을 제공하며 서버 측 코드와 클라이언트측 코드를 작성할 수 있습니다.
- **RPC(원격 프로시저 호출)**: 서버 및 클라이언트 RPC를 사용하여 게임 오브젝트의 원격 인스턴스에서 메시지를 전송하고 메서드를 호출합니다.
- **NetworkVariable:** 네트워크 간에 상태를 동기화할 수 있는 클래스입니다.
- **NetworkManager:** 게임의 네트워크 상태를 관리하고, 연결, 연결 해제, 씬 관리 등의 작업을 처리하는 핵심 요소입니다.

이 요소는 애플리케이션 레이어에서 작동하여 멀티플레이어 기능을 구현하는 데 도움을 줍니다. Netcode for GameObjects는 간단하면서도 강력한 성능을 발휘하도록 설계되어, 안정적이면서도 확장성이 뛰어난 멀티플레이어 게임을 제작하는 데 필요한 툴을 제공합니다.

Netcode for Entities

Unity의 DOTS(데이터 지향 기술 스택)와 ECS(엔티티 컴포넌트 시스템)를 기반으로 개발된 Netcode for Entities는 서버가 권한을 보유하는 게임플레이를 고려하여 설계되었으며, 클라이언트측 예측, 보간 및 지연 보상을 제공합니다.

이 구조에서는 중앙 서버가 모든 게임 시뮬레이션을 처리하여 게임 결과를 제어함으로써 부정행위를 줄일 수 있습니다. 클라이언트가 서버에 입력을 전송하면 서버는 이 입력을 처리한 후 업데이트된 게임 상태를 반환하여 조작 가능성을 최소화합니다.

Netcode for Entities는 클라이언트측 예측을 포함해 지연을 완화하며, 거의 지연을 느낄 수 없는 더 원활한 경험을 제공합니다. Netcode for GameObjects보다 확장성과 대역폭 최적화가 뛰어납니다.

경험이 풍부한 멀티플레이어 개발자가 높은 성능과 결정론적 분명성이 필요한 프로젝트에 참여한 경우 DOTS와 ECS가 게임에 적합한 기반이 될 수 있습니다.

Unity의 데이터 지향 디자인에 대해 자세히 알아보려면 DOTS 전자책 및 DOTS 리소스 목록을 확인하세요.

Unity 샘플인 ECS 네트워크 레이싱은 Netcode for Entities로 제작되었습니다.

어떤 Netcode 솔루션을 선택하는 것이 적합할지는 프로젝트의 요구 사항과 팀의 전문성에 따라 달라집니다. 결정을 내릴 때 아래 표가 도움이 될 수 있습니다.

기능	Netcode for GameObjects	Netcode for Entities
타겟층	초급 및 중급 개발자	고급 개발자
아키텍처	객체 지향(MonoBehaviour 기반)	데이터 지향(ECS(엔티티 컴포넌트 시스템) 및 DOTS)
성능	규모가 작은 게임에 적합	고성능과 확장성을 발휘하도록 최적화
확장성	제한됨, 플레이어가 소수인 경우에 적합	고사양의 대규모 게임을 고려하여 설계됨
네트워킹 기능	NetworkVariable, RPC, NetworkTransform, 제한적인 클라이언트측 예측(예상), 보간, UnityTransport 지원	모든 기능을 갖춘 클라이언트측 예측, 보간, 지연 보상, 최적화된 UnityTransport 지원
Unity 서비스 연동	Lobby, Relay 등을 모두 지원	Lobby, Relay 등을 모두 지원
샘플 프로젝트	보스 룸, 간단한 샘플	메가시티 메트로, ECS 레이싱, Netcode 샘플

Unity Multiplayer Center도 훌륭한 리소스입니다. Multiplayer Center는 멀티플레이어 게임 제작의 출발점으로 활용할 수 있습니다. 게임의 요구 사항에 따라 Unity 멀티플레이어 패키지를 추천해 주며 이를 활용하는 데 도움이 되는 샘플과 튜토리얼도 제공합니다. Multiplayer Center 기술 자료에서 프로젝트에 패키지를 다운로드하고 설정하는 방법을 참고하세요.

Unity Transport

Unity Transport 패키지는 Netcode에 구애받지 않는 라이브러리로, 성능과 안정성에 중점을 둔 저수준 네트워크 레이어를 제공하며, 기존 UDP를 고급 기능으로 확장하여 안전하면서도 이식성이 뛰어난 최신 전송 라이브러리입니다.

- **안정성:** UDP의 통신 안정성을 강화하며, TCP의 오버헤드 없이 중요한 메시지가 누락되지 않고 전달되도록 함
- **보안:** 암호화 및 인증 기능이 적용되어 무단 액세스로부터 데이터를 보호할 수 있음

— **성능:** 지연을 낮추고 처리량을 높이도록 최적화됨

— **크로스 플랫폼 호환성:** 다양한 플랫폼과 기기에서 원활하게 작동하도록 설계됨

Netcode for GameObjects와 Netcode for Entities는 모두 기본적으로 Unity Transport를 기반으로 합니다. 네트워크를 항상 세밀하게 제어하려는 개발자는 Unity Transport를 스탠드얼론 라이브러리로 사용하여 특정 게임 요구 사항에 맞는 커스터마이즈된 Netcode를 구축할 수도 있습니다.

이제 Unity Transport에 크로스 플랫폼 및 웹 멀티플레이어 경험을 향상할 수 있는 WebGL 지원이 추가됩니다.

타사 네트워킹

Netcode for GameObjects로 시작하는 것을 권장하지만, 반드시 그렇게 해야 하는 것은 아닙니다. Unity 커뮤니티는 특정 요구 사항에 맞게 선택할 수 있는 다양한 솔루션을 제공합니다.

Unity 에셋 스토어에서 사용할 수 있는 타사 네트워킹 옵션은 다음과 같습니다.

— **PUN(Photon Unity Networking):** Photon은 높은 동시 접속자 수를 요구하는 게임에 맞는 종합적이고 확장성이 뛰어난 솔루션을 제공합니다.

— **Mirror:** 단순성과 사용 편의성으로 유명한 Mirror는 Unity의 UNet을 기반으로 한 무료 오픈 소스 네트워킹 라이브러리입니다.

— **DarkRift Networking 2:** 이 솔루션은 뛰어난 성능과 유연성을 모두 제공하며, 세밀한 네트워크 제어가 필요한 개발자들에게 맞게 설계되었습니다.

— **Forge Networking Remastered**: 고급 커스터마이징 및 제어가 가능한 오픈 소스 옵션입니다.

첫 번째 Netcode 프로젝트 설정

Unity의 네트워킹 솔루션을 아직 사용해 보지 않은 경우, 기본 Netcode 프로젝트를 설정하려면 필요한 네트워킹 패키지를 임포트한 후, 필요한 멀티플레이어 컴포넌트를 설정해야 합니다.

이번 장에서는 첫 번째 단계로 Netcode for GameObjects를 사용하여 샘플 프로젝트에 네트워킹을 추가해 보겠습니다. Unity 6에서는 Multiplayer Center를 사용하여 새로운 멀티플레이어 프로젝트를 설정할 수 있으며 Multiplayer Widgets을 통해 Unity 서비스를 프로젝트에 추가로 연동할 수 있습니다.

시작하기 전에

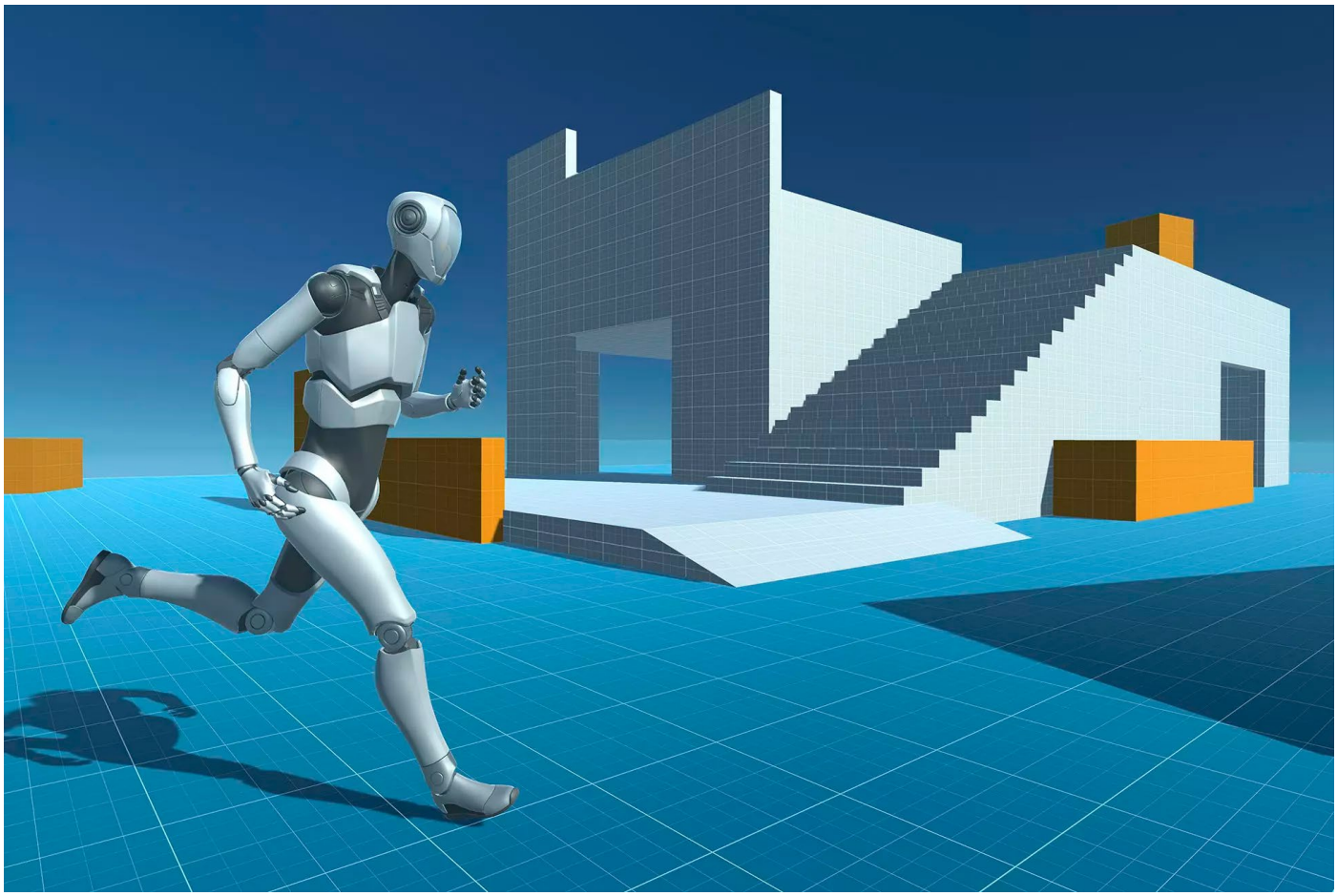
다음과 같은 준비물이 필요합니다.

- 유효한 라이선스가 사용된 활성 Unity 계정
- Unity Hub
- 지원되는 Unity 에디터 버전, 여기에서 소개되는 일부 기능은 Unity 6 이상 필요(Netcode for GameObjects 요구 사항 참고)
- 프로젝트에 필요한 Unity 서비스에 연결하기 위한 Unity Cloud 대시보드 연결(Unity Hub를 통해 연결 가능)

샘플 프로젝트 설정

이 Netcode 툴은 이동 기능이 구현된 기존 싱글플레이어 프로젝트에 사용해 보면 좋습니다. 이 가이드에서는 Unity 에셋 스토어의 Starter Assets – ThirdPerson 패키지를 사용합니다. URP(유니버설 렌더 파이프라인)를 사용하여 휴머노이드 캐릭터로 간단한 3D 게임플레이를 시뮬레이션하는 패키지입니다.

Unity 에셋 스토어에서 이 무료 에셋을 다운로드한 후, 패키지 관리자를 사용하여 임포트하세요.



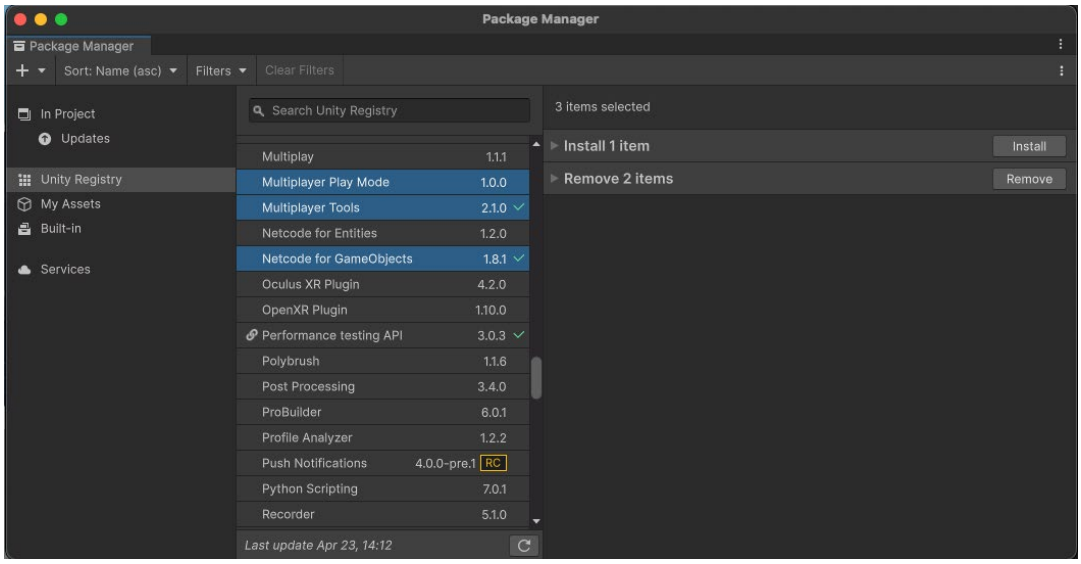
에셋 스토어의 Starter Assets 패키지

이 데모 프로젝트에는 작은 테스트 플레이그라운드 씬과 사용자 설정이 가능한 3인칭 컨트롤러가 포함되어 있습니다. 프로젝트의 목표는 이 애플리케이션을 여러 개 실행한 다음, 서로 다른 클라이언트가 동일한 환경에서 상호 작용하도록 구현하는 것입니다.

Netcode for GameObjects 설치

Package Manager(**Window > Package Manager**)에서 Unity Registry로 필터링합니다. 이제 설치할 패키지는 다음과 같습니다.

- **Netcode for GameObjects**: 기존 GameObject/MonoBehaviour 워크플로에 멀티플레이어 기능을 추가하는 기본 네트워킹 라이브러리입니다. 멀티플레이어 게임 개발을 간소화하며, 처음으로 네트워크 멀티플레이어 개발을 시작할 때 활용하면 좋습니다.
- **Multiplayer Tools 창:** 다음과 같이 Unity 6에 도입된 5가지 새로운 툴로 구성된 추가 제품군으로, 멀티플레이어 개발 워크플로를 개선합니다.
  - Multiplayer Tools 창에서는 모든 멀티플레이어 툴과 해당 기술 자료를 한 위치에서 편리하게 이용할 수 있습니다.
  - Network Simulator는 발생할 수 있는 문제를 출시 전에 파악할 수 있도록 패킷 지연, 손실, 연결 끊김 등의 실제 네트워크 상태를 재현합니다.
  - RNSM(Runtime Network Stats Monitor)은 실시간 네트워크 통계를 표시하여 사용자 설정 가능한 화면에서 네트워크 성능을 모니터링할 수 있습니다.
  - Network Scene Visualization은 네트워크 활동과 오브젝트 소유권을 씬 뷰에 시각적으로 표시하여 디버깅을 개선합니다.
  - Hierarchy Network Debug 뷰는 네트워크 오브젝트를 구별할 수 있는 오버레이를 계층(Hierarchy) 창 오른쪽에 제공합니다(작은 네트워크 큐브 로고로 표시됨).
- Multiplayer Play Mode: 이 Unity 6 패키지를 사용하면 Unity 에디터를 벗어나지 않아도 멀티플레이어 기능을 테스트할 수 있습니다. 최대 4명의 플레이어(메인 에디터 플레이어와 3명의 가상 플레이어)를 시뮬레이션할 수 있으므로 플레이를 빠르게 테스트할 수 있습니다.



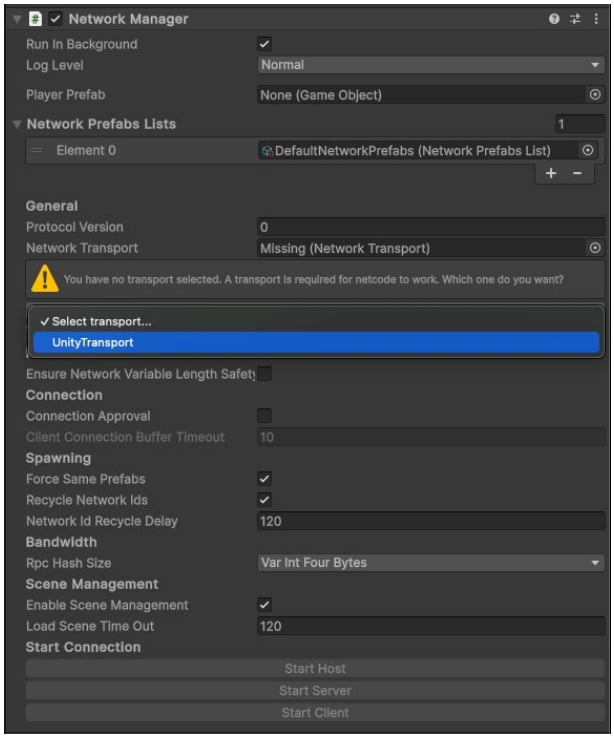
Netcode for GameObjects 및 보조 패키지를 설치합니다.

NetworkManager 추가

각 프로젝트에서 네트워크 멀티플레이어를 지원하려면 **NetworkManager** 컴포넌트가 필요합니다. 이 필수 컴포넌트는 프로젝트의 네트워크 상태를 관리하고, 연결 및 네트워크 설정을 처리합니다.

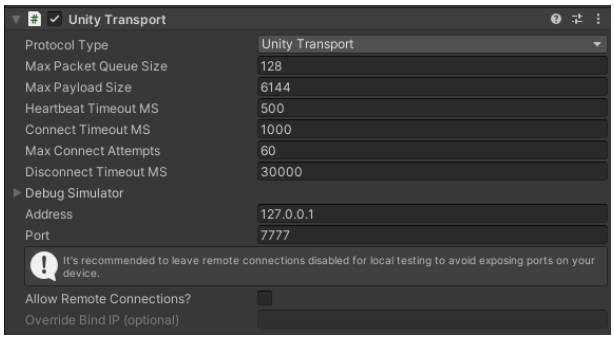
씬에 NetworkManager를 추가하려면 계층 구조에서 새로운 게임 오브젝트를 생성하고 **NetworkManager** 컴포넌트를 추가합니다(**Netcode > NetworkManager**).

NetworkManager 컴포넌트에서 **Network Transport** 레이어를 설정합니다. Unity Transport를 선택합니다.



NetworkManager에서 전송 레이어를 선택합니다.

그러면 **UnityTransport** 컴포넌트가 게임 오브젝트에 연결됩니다. 전송 레이어는 연결 관리, 데이터 전송, 패킷 암호화 등의 저수준 네트워킹 작업을 담당합니다.



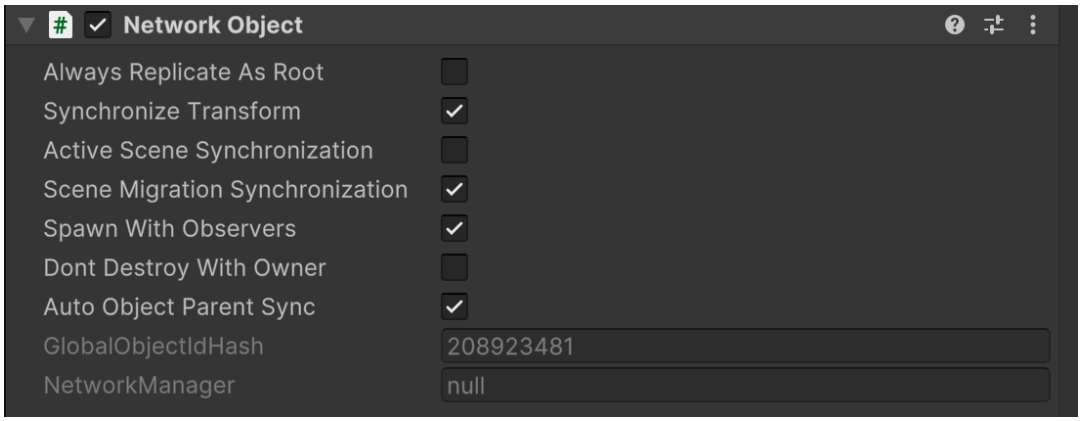
UnityTransport 컴포넌트

아직은 이 설정을 수정할 필요가 없지만, 이 컴포넌트는 에디터에서 테스트하고 디버그할 목적으로 네트워크 상태(예: 지연, 패킷 손실, 지터)를 시뮬레이션하는 데 도움이 될 수 있습니다.

씬을 저장하고 **File > Build Settings**로 이동하여 현재 씬이 **Scenes in Build** 목록에 추가되어 있는지 확인합니다. 이 목록에 있으면 새로운 NetworkManager가 게임 빌드에 확실히 포함된 것입니다.

NetworkObject

NetworkObject는 멀티플레이어 게임에서 서로 다른 클라이언트 간에 네트워크로 연동되거나 동기화되어야 하는 모든 게임 오브젝트에 필요한 컴포넌트입니다. NetworkObject 컴포넌트를 게임 오브젝트에 추가하면 해당 컴포넌트는 '네트워크 연동 가능' 상태가 되어, 상태와 동작을 네트워크를 통해 공유하고 업데이트할 수 있습니다.



NetworkObject 컴포넌트와 해당 고유 ID

각 NetworkObject에는 다음과 같은 몇 가지 식별자가 있습니다.

— **GlobalObjectIdHash**는 프로젝트 내의 프리팹 에셋을 식별합니다.

— **NetworkObjectId**는 동일한 프리팹 에셋의 인스턴스를 구분하는 고유 식별자입니다.

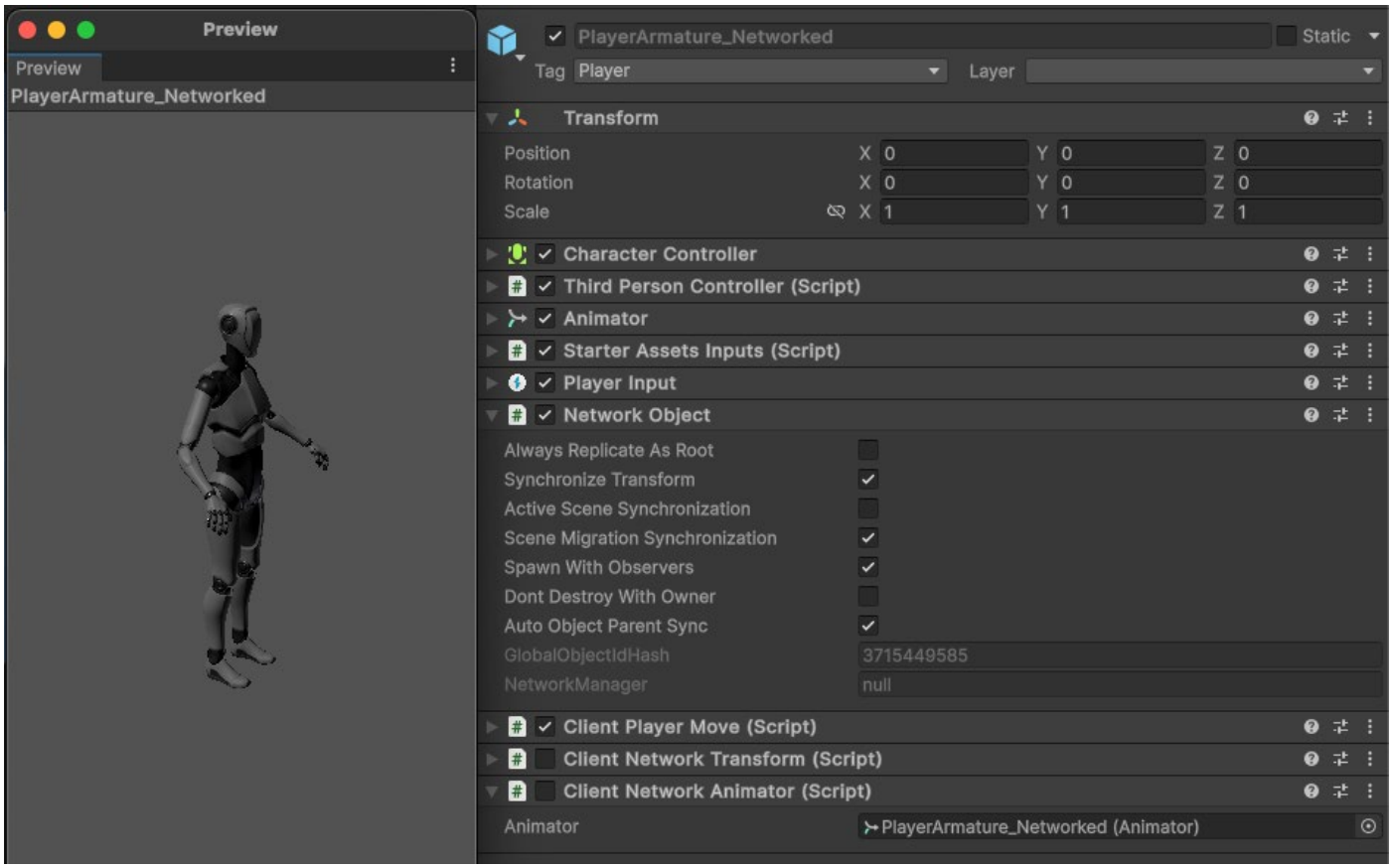
— **OwnerClientId**는 오브젝트를 '소유'한 클라이언트를 의미합니다(아래의 권한 참조).

이 식별자는 NetworkManager가 해당 클라이언트를 추적하고 연결된 모든 클라이언트에서 오브젝트의 상태를 일관되게 유지할 수 있도록 도와줍니다. NetworkObjects는 게임플레이 중에 동적으로 생성(스폰)되거나 파괴될 수 있습니다.

NetworkObject를 생성하면 연결된 모든 클라이언트에 이 오브젝트가 표시됩니다. 각 NetworkObject에는 소유자가 있으며, 이 소유자는 일반적으로 해당 오브젝트의 동작과 상태를 제어하는 클라이언트입니다.

플레이어 NetworkObject

각 플레이어는 플레이어 NetworkObject라는 자체 프리팹을 선택적으로 가질 수 있습니다. 이 프리팹은 게임 내에서 주로 플레이어의 캐릭터 컨트롤러와 시각적 표현을 포함하는 특수한 유형의 NetworkObject입니다.



샘플 프로젝트의 플레이어 NetworkObject

플레이어 NetworkObject는 주로 플레이어의 이름, 점수, 인벤토리 또는 기타 관련 정보를 비롯한 플레이어 관련 데이터를 저장하고 동기화합니다. 이 데이터는 네트워크에 걸쳐 동기화되므로 연결된 모든 플레이어가 일관된 게임 상태를 경험할 수 있습니다.

클라이언트가 연결되면 NetworkManager는 해당 플레이어가 '소유'하는 플레이어 NetworkObject를 생성합니다. 즉, 플레이어는 자신의 PlayerObject에 대한 권한을 가지고, 그 행동과 상태를 제어할 수 있습니다.

플레이어 NetworkObject를 설정하려면 먼저 프로젝트에 표준 프리팹 게임 오브젝트를 생성합니다. 이 프리팹은 PlayerObject의 템플릿 역할을 하며 플레이어의 동작과 형상을 정의하는 필요한 컴포넌트와 스크립트를 포함합니다.

그런 다음 적절한 Netcode 컴포넌트를 추가합니다. 다음과 같은 컴포넌트가 있습니다.

— **NetworkObject:** 네트워크로 연동할 오브젝트마다 NetworkObject 컴포넌트가 필요합니다. 이 컴포넌트는 생성, 제거, 소유권과 관련된 프로퍼티와 이벤트를 포함합니다.

— **NetworkBehaviour:** 이 스크립트는 MonoBehaviour 기본 클래스에 네트워킹 동작을 추가합니다. NetworkBehaviour에는 네트워크 변수, RPC(원격 프로시저 호출), 네트워크 콜백이 포함됩니다.

— **NetworkAnimator:** 이 컴포넌트는 애니메이션 상태와 파라미터를 클라이언트 간에 동기화합니다.

— **NetworkTransform:** 이 컴포넌트는 실시간으로 플레이어의 위치, 회전, 스케일을 서버에서 연결된 모든 클라이언트로 복제합니다.

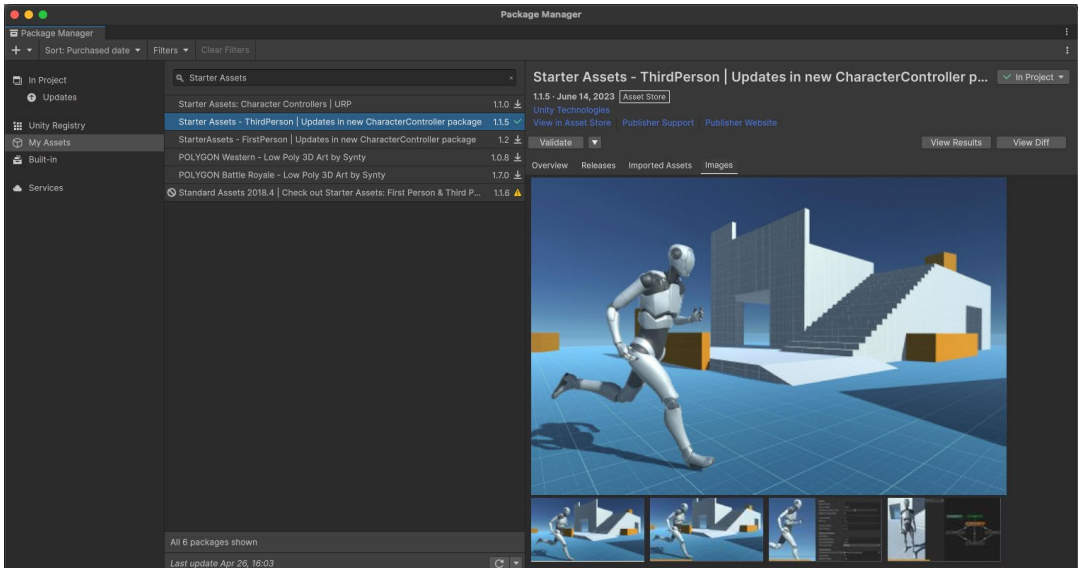
플레이어 NetworkObject는 일반적으로 플레이어 입력 처리를 담당합니다. 플레이어가 움직이거나 게임 월드와 상호 작용하는 등의 행동을 취하면 해당 입력은 처리된 후 필요에 따라 다른 연결된 플레이어들에게 전파됩니다.

플레이어 로직은 직접적인 게임 메카닉을 위한 MonoBehaviour 및 네트워크 상태를 관리하기 위한 NetworkBehaviour의 조합으로 이루어집니다. 캐릭터 컨트롤러 및 애니메이터와 같은 비네트워크 컴포넌트는 대개 각 플레이어의 로컬 인스턴스에서 작동합니다.

해당 컴포넌트를 로컬에서 사용하면 성능이 최적화될 뿐만 아니라 네트워크 트래픽도 감소합니다. 이러한 점은 클라이언트 간 대역폭이 제한된 환경에서 작업할 때 중요할 수 있습니다.

플레이어 NetworkObject 생성

샘플 프로젝트에서 **Playground** 씬을 로드합니다.

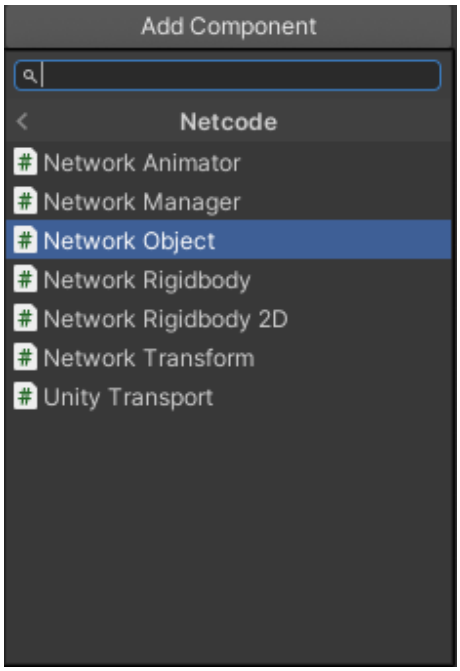


Starter Assets 번들에는 Playground 씬이 있습니다.

계층 구조에는 게임 캐릭터를 구동하는 PlayerArmature가 있습니다. 이를 플레이어 NetworkObject로 변환하려면 계층 구조에서 드래그하여 고유 프리팹을 새로 생성하거나 프로젝트에 있는 기존 프리팹의 사본을 수정합니다.

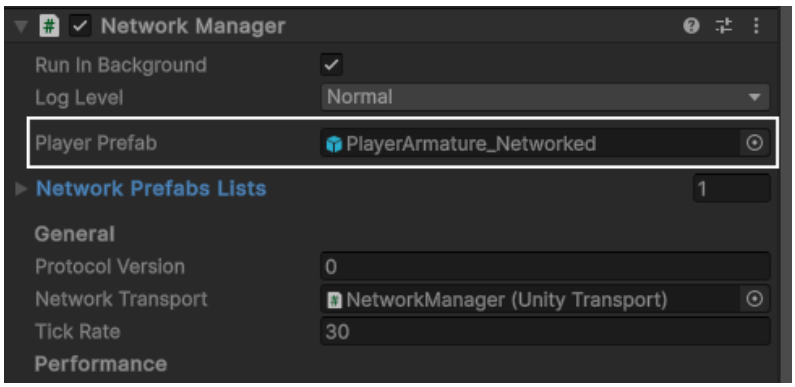
계층 창에서 PlayerArmature 게임 오브젝트를 찾습니다. 삭제하면 PlayerArmature와 그 자식 오브젝트가 씬에서 제거되고, 씬에는 게임 환경만 남게 됩니다.

그런 다음, 인스펙터(Inspector)에서 프리팹을 편집합니다. **NetworkObject** 컴포넌트를 추가합니다. 오브젝트를 네트워크에서 인식하고 관리하려면 이 컴포넌트가 필요합니다.



프리팹에 NetworkObject를 추가합니다.

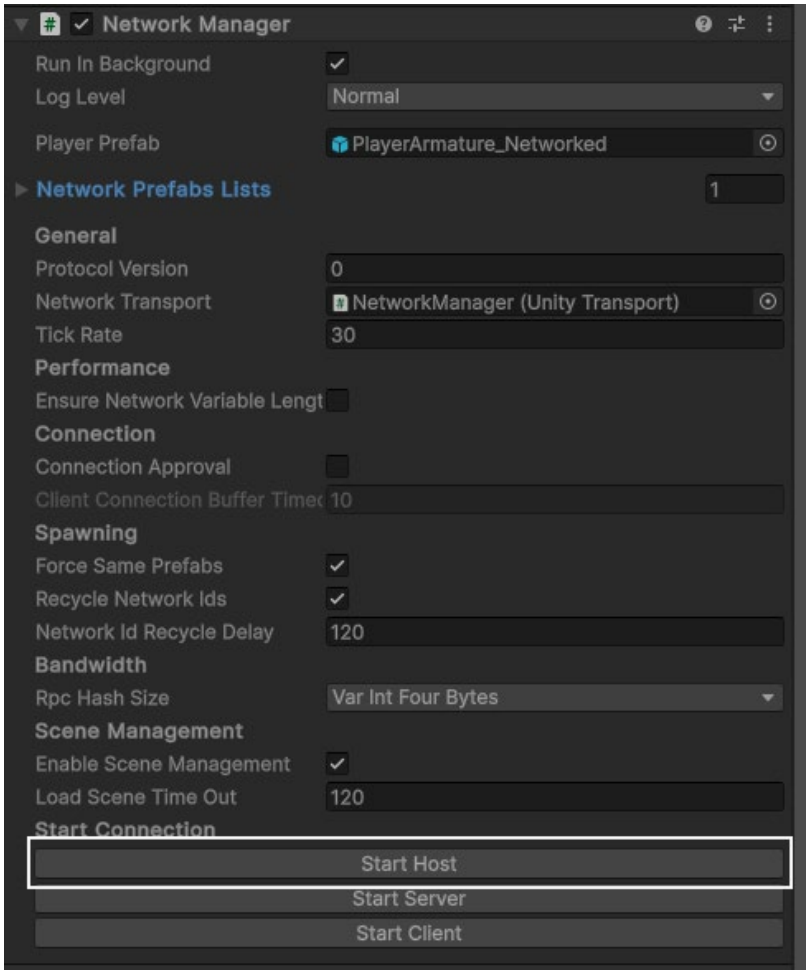
NetworkManager의 **Player Prefab** 필드에 플레이어 NetworkObject를 등록합니다.



NetworkManager에 플레이어 NetworkObject를 등록합니다.

플레이 모드에서는 게임 환경만 표시됩니다. 플레이어 NetworkObject는 클라이언트가 연결된 경우에만 표시됩니다.

이제 계층 구조에서 **DontDestroyOnLoad** 아래에 표시되는 **NetworkManager**를 선택합니다.



NetworkManager에서 호스트를 시작합니다.

Start Host를 선택합니다. 그러면 플레이어 NetworkedObject가 생성됩니다.

PlayerArmature_Network 오브젝트가 계층 구조에 표시됩니다. 다시 한 번 게임을 플레이할 수 있습니다 (단, 카메라 타겟은 비활성화되어 있음). WASD 컨트롤을 사용하여 플레이어의 움직임을 테스트합니다.

플레이 모드를 종료하고 플레이어 캐릭터는 사라집니다. NetworkManager에서는 이제 런타임 시 이 특정 플레이어 캐릭터를 생성하고 관리합니다.

여기서 유의할 점은 게임의 네트워크 요소가 여러 클라이언트가 연결될 때까지 명확하게 드러나지 않는다는 것입니다. 네트워크 요소가 멀티플레이어 씬에서 어떻게 작동하는지 파악하려면 여러 클라이언트를 사용하여 테스트해야 합니다.

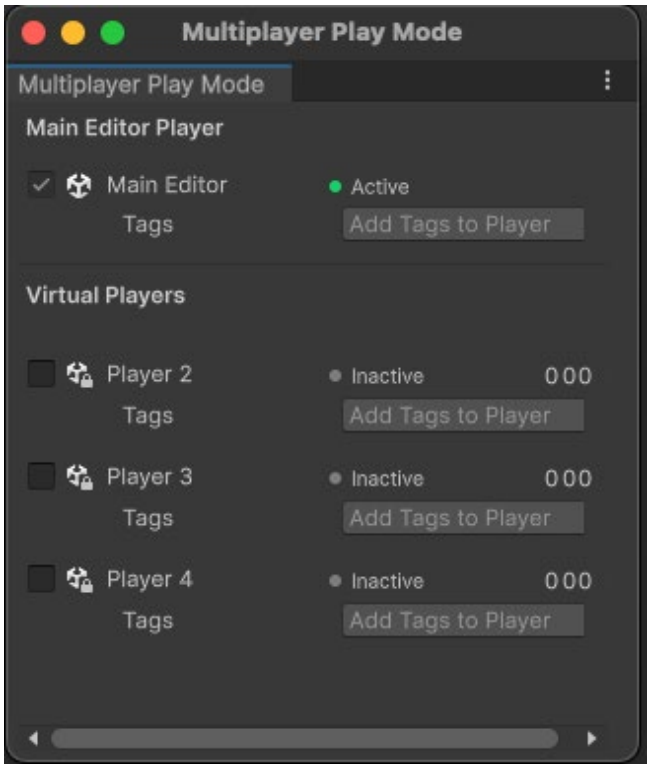
Multiplayer Play Mode

멀티플레이어를 테스트하려면 별도의 런타임 프로세스에서 애플리케이션을 실행해야 합니다. 이전에는 별도의 게임 빌드를 만들어 Unity 에디터에서 실행해야 했습니다.

이 방법을 계속 사용해도 되지만 Unity 6에는 **MPPM(Multiplayer Play Mode)**이 포함되어 있습니다. 개발자는 MPPM을 사용해 Unity 에디터에서 여러 인스턴스를 동시에 열어 멀티플레이어 환경을 모사할 수 있습니다. 이 방법으로 멀티플레이어 테스트 프로세스를 간소화할 수 있습니다.

패키지 관리자를 통해 Multiplayer Play Mode을 설치하면 새로운 기능을 테스트할 때마다 애플리케이션을 빌드할 필요가 없습니다.

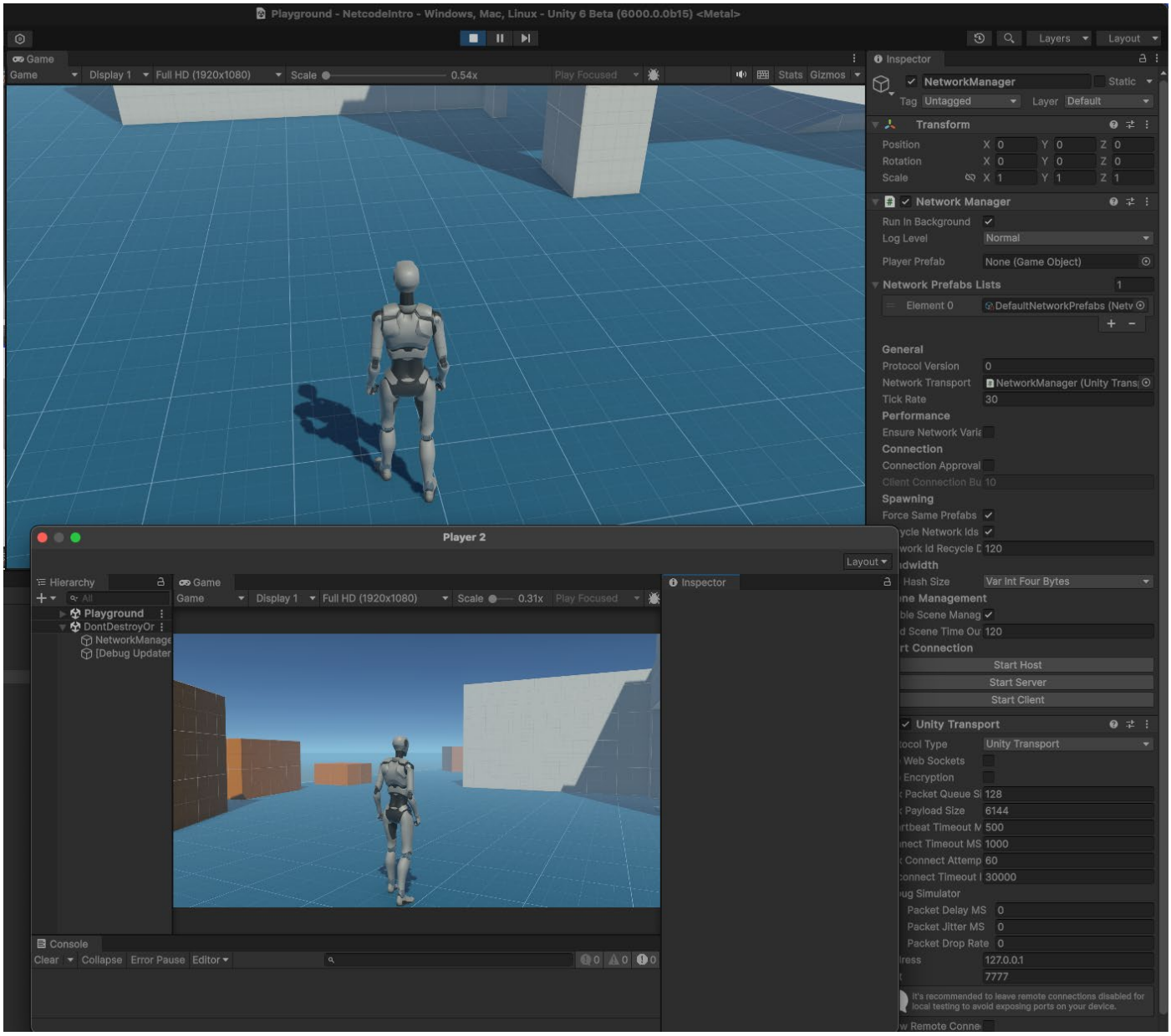
Multiplayer Play Mode(**Window > Multiplayer Play Mode**)를 엽니다.



Multiplayer Play Mode 창

그런 다음, 최소 한 쌍의 호스트와 클라이언트를 테스트할 수 있도록 위 스크린샷에 나와 있는 목록에서 1명 이상의 가상 플레이어를 추가로 선택합니다. 호스트는 서버에서 실행되는 클라이언트이기도 하다는 점을 기억하세요.

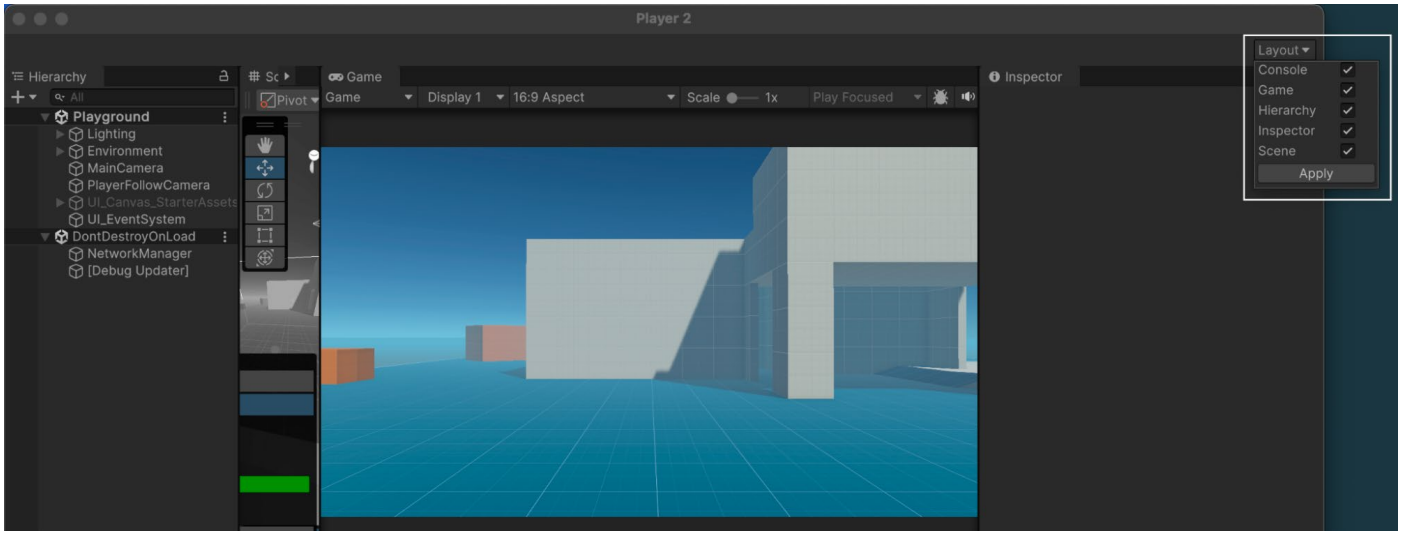
플레이 모드에 진입하면 애플리케이션의 두 번째 세션이 복제된 창에서 실행되기 시작합니다.



Multiplayer Play Mode에서는 가상 플레이어가 복제됩니다.

계층 구조에서 **NetworkManager**를 선택합니다. 인스펙터의 **Start Connection**에서 **Start Host**를 선택합니다. PlayerArmature_Networked 오브젝트가 게임(Game) 뷰에 표시됩니다.

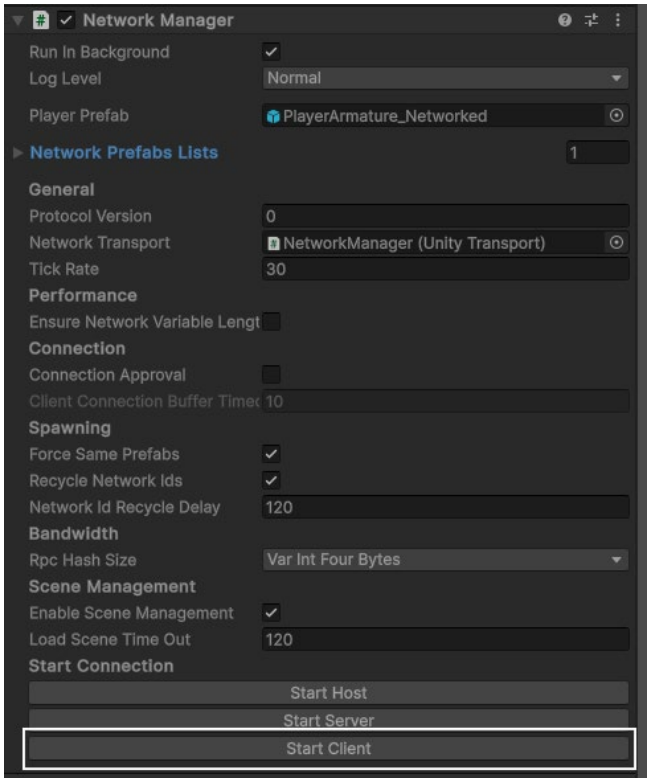
두 번째 창에서 Layout 버튼을 사용하여 인스펙터와 계층 구조를 활성화합니다. 그러면 마치 에디터의 두 번째 세션처럼 보일 것입니다. 사용할 사용자 인터페이스 컴포넌트를 선택하고 **Apply**를 누릅니다.



복제된 창에서 Layout을 활성화합니다.

복제된 에디터 창의 **DontDestroyOnLoad**에서 **NetworkManager**를 찾습니다.

인스펙터 창의 **Start Connection**에서 **Start Client를 선택합니다.**



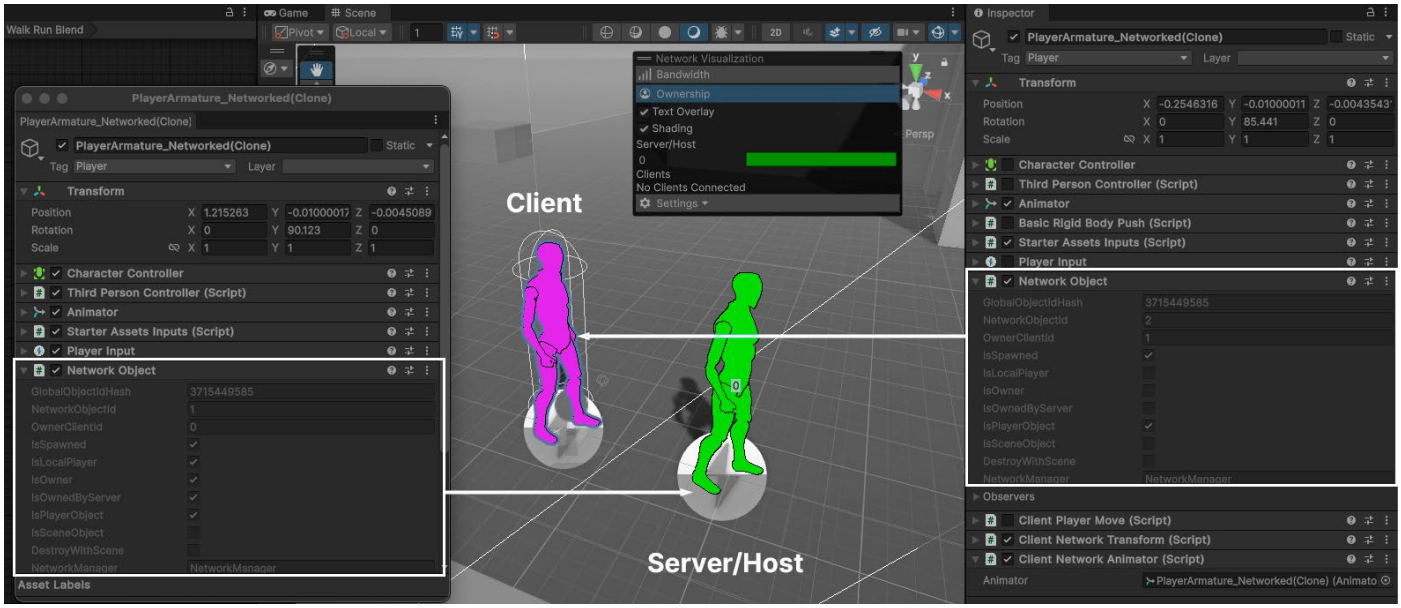
NetworkManager에는 클라이언트 연결 버튼이 있습니다.

이제 PlayerArmature_Networked의 두 인스턴스가 계층 구조에 표시됩니다. 씬(Scene) 뷰에서 두 인스턴스는 서로 겹쳐서 표시됩니다. 키보드나 게임패드를 사용하여 한 플레이어 인스턴스를 다른 플레이어 인스턴스에서 멀리 떨어뜨려 놓습니다.

계층 구조에서 PlayerArmature_Networked 인스턴스를 선택하여 해당 NetworkObject 컴포넌트를 검사합니다.

런타임에 각 인스턴스가 **GlobalObjectIdHash**(프로젝트 에셋 ID) 및 **NetworkObjectId**(고유 인스턴스 인덱스)로 어떻게 구별되는지 알아봅니다. 그 아래의 **OwnerClientId**는 호스트 또는 클라이언트가 인스턴스를 제어하는지 여부를 나타냅니다.

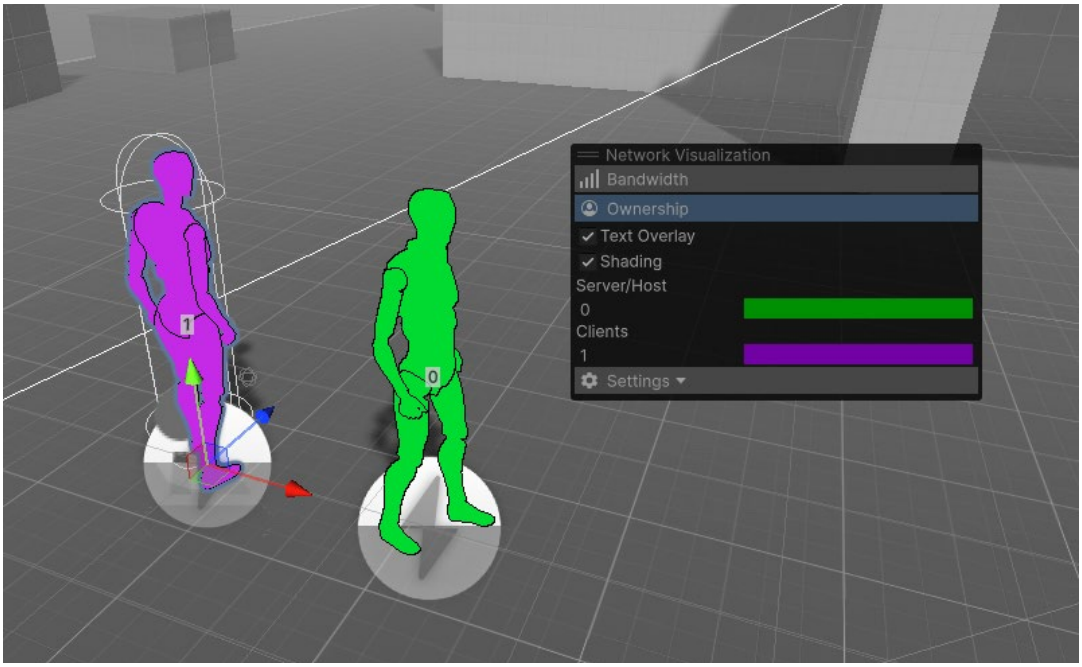
두 인스턴스를 전환하면서 **IsSpawned**, **IsLocalPlayer**, **IsOwner**, **IsOwnerByServer** 등의 플래그를 비교합니다.



두 인스턴스 간의 NetworkObject 설정을 비교합니다.

Network Visualization 패널을 사용하면 두 인스턴스를 더 명확하게 구분할 수 있습니다. Multiplayer Tools 패키지를 설치하면 이 편리한 진단 툴이 씬 뷰에 표시됩니다.

두 인스턴스는 **대역폭**(전송되는 데이터의 양) 또는 **소유권**(플레이어 NetworkObject에 대한 권한을 가진 클라이언트)에 따라 색상으로 구분됩니다.



Network Visualization은 NetworkObjects를 디버그하는 데 도움이 됩니다.

NetworkManager는 클라이언트의 인스턴스를 각각 별도로 생성하지만, 각각은 같은 캐릭터를 독립적으로 제어합니다. 씬 뷰에서 WASD 컨트롤로 유발되는 캐릭터의 움직임은 클라이언트와 호스트 간에 동기화되지 않습니다.

NetworkManager는 플레이어가 처음 연결될 때 좌표 (0, 0, 0)으로 위치를 동기화하지만, 그 이후의 움직임은 동기화하지 않습니다. 현재 여러 로컬 컴포넌트가 캐릭터의 동작을 제어하고 있습니다.

— **CharacterController**를 사용하면 복잡한 물리 계산을 수행할 필요 없이 플레이어가 게임 환경과 상호 작용하면서 움직일 수 있습니다.

— **Animator**는 상태 머신을 기반으로 애니메이션을 구현합니다. Animator는 달리기, 점프, 또는 대기(idle) 상태 간의 전환과 블렌딩을 제어합니다.

— PlayerInput은 플레이어별 입력 관리, 기기 페어링, 이벤트 알림을 처리하여 Unity Input System에 적용되는 고수준 래퍼를 제공합니다.

— StarterAssetsInputs는 해당 입력을 캐릭터의 움직임, 시점, 점프, 질주 입력으로 변환합니다.

이러한 컴포넌트는 싱글플레이어 컴포넌트입니다. 이를 멀티플레이어 애플리케이션에 적용하려면 네트워크 스크립팅을 추가해야 합니다.

자체 UI 시작 버튼 만들기

런타임 시 더 사용자 친화적인 방식으로 네트워크 세션을 시작할 수 있도록, NetworkManager의 Inspector 버튼 기능을 재현한 화면 버튼을 추가할 수 있습니다. Unity UI(UGUI) 또는 UI 툴킷을 사용하여 구현할 수 있습니다.

선택한 UI에 Client, Host, Server라는 세 개의 버튼을 만듭니다. 그런 다음 각 버튼이 NetworkManager 싱글톤에서 각각 다음과 같은 콜백을 호출하게 합니다.

— `NetworkManager.Singleton.StartClient`

— `NetworkManager.Singleton.StartHost`

— `NetworkManager.Singleton.StartServer`

이 콜백을 사용하면 인스펙터 창에서 버튼을 사용하지 않고도 네트워크 세션을 시작할 수 있습니다.

NetworkBehaviour 추가

PlayerArmature_Networked의 MonoBehaviour를 관리하기 위해 NetworkBehaviour를 사용할 수 있습니다.

NetworkBehaviour 는 네트워크 로직에 특화된 MonoBehaviour 유형으로, 다양한 게임 클라이언트 간에 동작과 상태를 동기화하는 데 필요한 프레임워크를 제공합니다.

NetworkBehaviour는 MonoBehaviour와 동일한 라이프사이클 이벤트를 공유하지만, 다음과 같은 네트워크 관련 기능을 제공합니다.

RPC 메서드: NetworkBehaviour는 RPC(원격 프로시저 호출)를 사용하여 네트워크 통신을 처리할 수 있습니다. 이러한 메서드에는 [Rpc] 속성이 추가되어 있습니다. 서버나 클라이언트에 Rpc를 전송하려면 각각 [`Rpc(SendTo.Server)`] 및 [`Rpc(SendTo.Client)`]를 호출합니다.

— **`NetworkVariable`:** 네트워크에서 동기화된 상태를 관리하는 데 특화된 변수입니다. 서버에서 `NetworkVariable`이 변경되면 해당 변경 사항이 모든 클라이언트에 자동으로 전파됩니다.

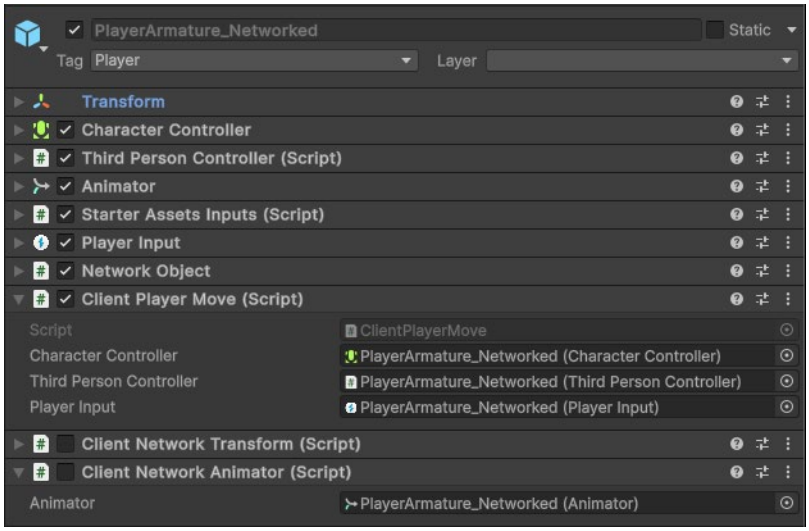
— **`OnNetworkSpawn`** 및 **`OnNetworkDespawn`**: 이 라이프사이클 메서드는 NetworkBehaviour가 인스턴스화되거나 파괴될 때 트리거됩니다. `OnNetworkSpawn`은 초기화에 사용됩니다. 네트워크 동작이라는 점을 제외하면 `OnEnable` 또는 `Start`와 유사하다고 보면 됩니다. `OnNetworkDespawn`은 오브젝트가 네트워크에서 제거되기 전에 정리하는 작업을 처리합니다(예: `OnDestroy` 또는 `OnDisable`과 유사).

— **소유권**: `NetworkBehaviour`는 특정 클라이언트(또는 서버)가 특정 오브젝트에 대한 소유권을 가질 수 있도록 허용합니다. 클라이언트나 서버가 NetworkObject를 '소유'할 수 있다는 이 권한 개념은 지정된 플레이어만 특정 오브젝트를 제어하거나 그와 상호 작용할 수 있도록 합니다.

플레이어의 움직임을 관리하기 위해 ClientPlayerMove라는 NetworkBehaviour를 구현할 수 있습니다. 그러면 호스트와 클라이언트의 입력이 해당하는 플레이어 오브젝트에서만 작용하도록 할 수 있습니다. 설정의 예시는 다음과 같습니다.

```
using Unity.Netcode;
using StarterAssets;
using UnityEngine;
using UnityEngine.InputSystem;
namespace NetcodeDemo
{
   public class ClientPlayerMove: NetworkBehaviour
   {
       [SerializeField]
       CharacterController m_CharacterController;
       [SerializeField]
       ThirdPersonController m_ThirdPersonController;
       [SerializeField]
       PlayerInput m_PlayerInput;

       [SerializeField]
       Transform m_CameraFollow;
       private void Awake()

       {
           m_PlayerInput.enabled = false;
           m_ThirdPersonController.enabled = false;
           m_CharacterController.enabled = false;
       }

       public override void OnNetworkSpawn()
       {
           base.OnNetworkSpawn();
           enabled = IsClient; // 클라이언트인 경우 활성화
           if (!IsOwner)
           {
               // 소유자가 아닌 경우 비활성화
               enabled = false;
               m_PlayerInput.enabled = false;
               m_CharacterController.enabled = false;
               m_ThirdPersonController.enabled = false;
               return;
           }

           // 소유자인 경우 활성화
           m_PlayerInput.enabled = true;
           m_CharacterController.enabled = true;
           m_ThirdPersonController.enabled = true;

       }
   }
}
```

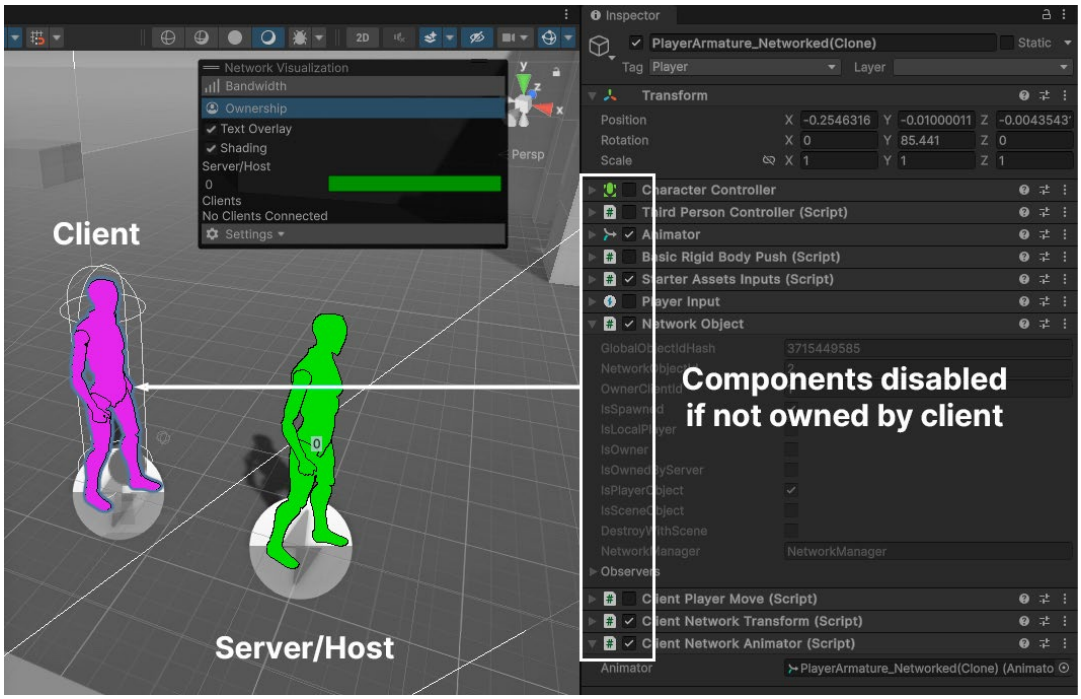
이 내용을 PlayerArmature_Networked 프리팹에 추가합니다. 그런 다음 인스펙터에서 필드를 적절히 선택합니다.



인스펙터에서 ClientPlayerMove 필드를 선택합니다.

이 스크립트를 프리팹에 적용한 후 호스트 및 클라이언트 세션을 연결합니다. 클라이언트가 NetworkManager에 연결하면 로컬 플레이어가 인스턴스의 소유자인지 확인하는 IsOwner 프로퍼티로 인해 플레이어 오브젝트의 특정 컴포넌트가 기본적으로 비활성화됩니다.

계층 구조에서 PlayerArmature_Networked의 두 인스턴스를 번갈아 가며 선택해 봅니다.



소유자가 아닌 경우 몇몇 컴포넌트가 자체적으로 비활성화됩니다.

이제 해당 클라이언트가 소유하지 않은 플레이어 인스턴스의 여러 컴포넌트(예: PlayerInput)가 비활성화된 것으로 표시됨을 확인할 수 있습니다. 이 설정을 통해 호스트는 플레이어 인스턴스 중 하나를 제어할 수 있고, 클라이언트는 나머지 인스턴스를 제어할 수 있습니다.

단, 다양한 플레이어 인스턴스를 제어할 수 있더라도 이들의 움직임은 네트워크를 통해 동기화되지 않습니다. 호스트에서 클라이언트로 움직임을 동기화하려면 `NetworkTransform`과 같은 네트워크 컴포넌트를 추가해야 합니다.

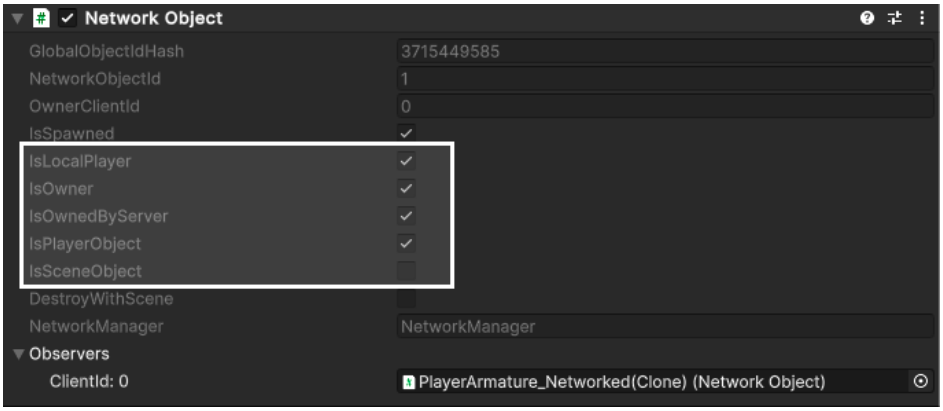
권한 및 소유권 프로퍼티

기본적으로는 서버가 NetworkObject를 소유합니다. 단, 연결되고 승인된 클라이언트는 `SpawnWithOwnership` 메서드를 사용하여 NetworkObject를 소유할 수 있습니다. Netcode for GameObjects에 대한 권한은 서버에 있으므로 서버만 NetworkObject를 생성 및 제거할 수 있습니다.

NetworkBehaviour에는 인스턴스의 권한과 소유권을 확인하는 몇 가지 간단한 방법이 있습니다.

- **`IsClient`**는 인스턴스가 클라이언트에서 실행 중인지 여부를 나타냅니다.
- **`IsServer`**는 인스턴스가 서버에서 실행 중인지 여부를 나타냅니다.
- **`IsHost`**는 인스턴스가 서버이자 클라이언트인 호스트에서 실행 중인지를 나타냅니다.
- **`IsLocalPlayer`**는 관련된 Netwo rkObject가 로컬 플레이어 오브젝트인지를 나타냅니다.
- **`IsOwner`**는 로컬 플레이어가 오브젝트를 소유하고 있는지, 아니면 오브젝트가 로컬 플레이어의 오브젝트인지를 나타냅니다.
- **`IsPlayerObject`**는 이 게임 오브젝트가 일반적으로 특정 클라이언트에 의해 제어되는 네트워크 플레이어를 의미하는지를 나타냅니다.
- **`IsSceneObject`**는 게임 오브젝트가 기본적으로 씬의 일부이며 게임플레이 중에 동적으로 생성된 것이 아님을 나타냅니다. 씬 오브젝트는 일반적으로 네트워크 전반에서 일관된 상태를 유지할 수 있도록 서버에서 관리됩니다.

런타임 시 NetworkObject를 검사하면 다음과 같은 몇몇 프로퍼티가 표시됩니다.



NetworkObject 설정

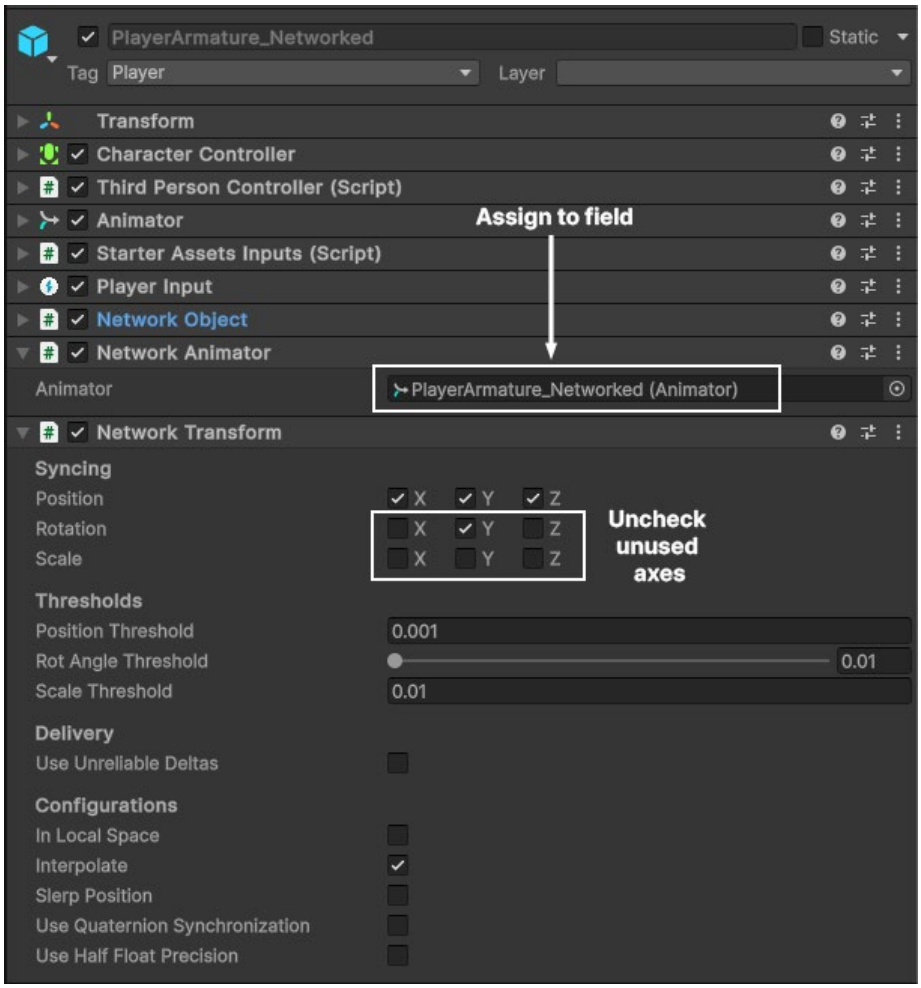
NetworkTransform과 NetworkAnimator를 사용한 동기화

NetworkBehaviour를 사용하면 여러 클라이언트에 동일한 플레이어 인스턴스를 생성할 수 있지만, 네트워크에서 플레이어 인스턴스의 움직임까지 동기화하려면 추가 컴포넌트가 필요합니다.

NetworkTransform 컴포넌트를 PlayerArmature_Networked 프리팹에 추가합니다. 게임플레이에 영향을 미치지 않는 축에 대한 선택을 해제합니다. 이 예시에서는 모든 스케일, 그리고 X 회전 축과 Z 회전 축의 선택을 해제합니다. 동기화는 대역폭을 사용하므로 불필요한 데이터의 동기화를 최소화하는 것이 중요합니다.

Multiplayer Play Mode에서 호스트 창에 포커스를 맞추면 컨트롤을 사용하여 플레이어를 움직일 수 있고 그 움직임이 클라이언트에 동기화되는 것을 볼 수 있습니다. 이는 네트워크 플레이가 시작되었음을 나타냅니다.

다음으로 **NetworkAnimator** 컴포넌트를 PlayerAramature_Networked에 추가합니다. 기존의 Animator 컴포넌트를 빈 필드로 드래그합니다.

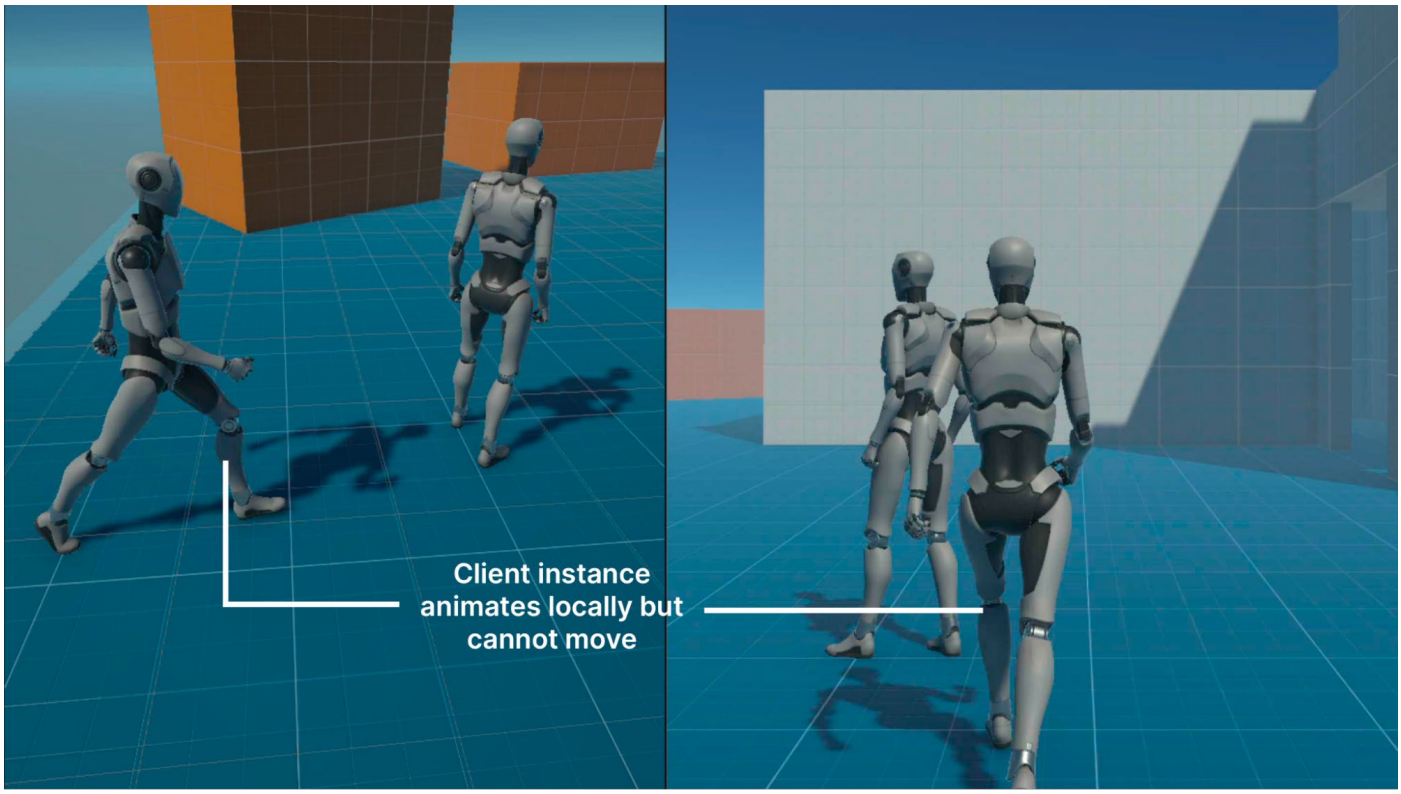


NetworkTransform 및 NetworkAnimator를 추가합니다.

클라이언트 창은 호스트에 연결된 두 번째 머신을 나타냅니다. 호스트에서 이루어진 모든 동작이 클라이언트에 반영되며, 그 반대도 성립하는 것이 이상적입니다.

NetworkTransform은 Transform의 위치, 회전, 스케일을 동기화할 수 있으며, NetworkAnimator는 애니메이션 상태를 동기화합니다. 이제 Multiplayer Play Mode에서 호스트 플레이어가 플레이그라운드 환경에서 뛰면 그 움직임이 클라이언트로 전송됩니다.

하지만 모든 것이 예상대로 작동하지는 않습니다. 클라이언트 창으로 포커스를 전환하여 컨트롤을 사용해 봅니다. 호스트는 클라이언트에 제대로 동기화되더라도, 클라이언트의 움직임은 호스트에 반영되지 않을 수 있습니다.



플레이어가 제자리에서 뛰는 것처럼 보입니다.

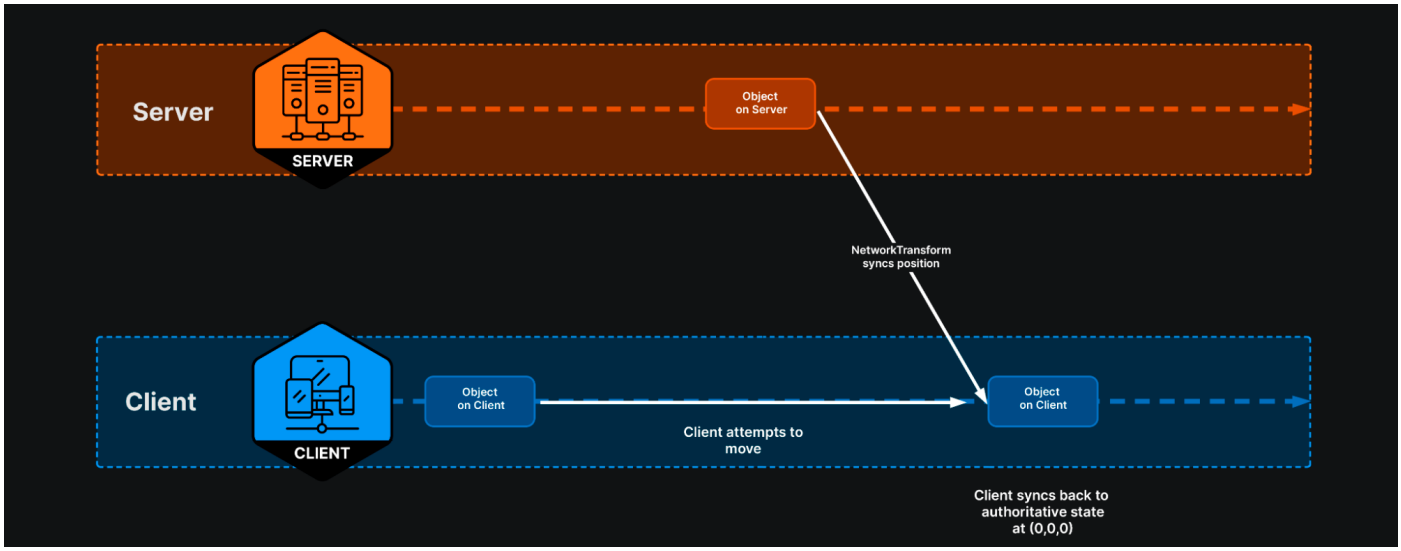
캐릭터가 제자리에서 움직인다는 것은 클라이언트에서 입력을 수신하지만 플레이어 인스턴스는 움직이지 않음을 나타냅니다. NetworkTransform이 서버 권한으로 작동하여 서버의 위치만 클라이언트와 동기화되기 때문입니다.

클라이언트에서 플레이어를 움직이려고 하면 서버에서는 클라이언트가 원하는 위치를 오버라이드하고 (0, 0, 0)으로 재설정합니다.

클라이언트 권한 적용

기본적으로 NetworkTransform은 서버 권한 모드로 작동합니다. 트랜스폼 축의 변경 사항은 서버 측에서 감지되어 연결된 클라이언트에 전송됩니다.

이 예시의 경우 클라이언트에서 플레이어를 변환하려는 시도는 실패합니다. 서버가 권한을 가지고 있는 상태로 트랜스폼을 (0,0,0)으로 설정하여 클라이언트측 변경 사항을 오버라이드하기 때문입니다.



서버 권한이 클라이언트를 오버라이드합니다.

이 문제를 해결하기 위한 한 가지 방법은 권한을 서버에서 클라이언트로 이전하는 것입니다. 그러면 클라이언트가 서버에 의해 오버라이드되지 않고 자체적인 트랜스폼을 제어할 수 있습니다.

이 동작을 구현하려면 다음 코드 예시와 같이 서버의 소유자 권한을 전환하는 **ClientNetworkTransform** 컴포넌트를 만들면 됩니다.

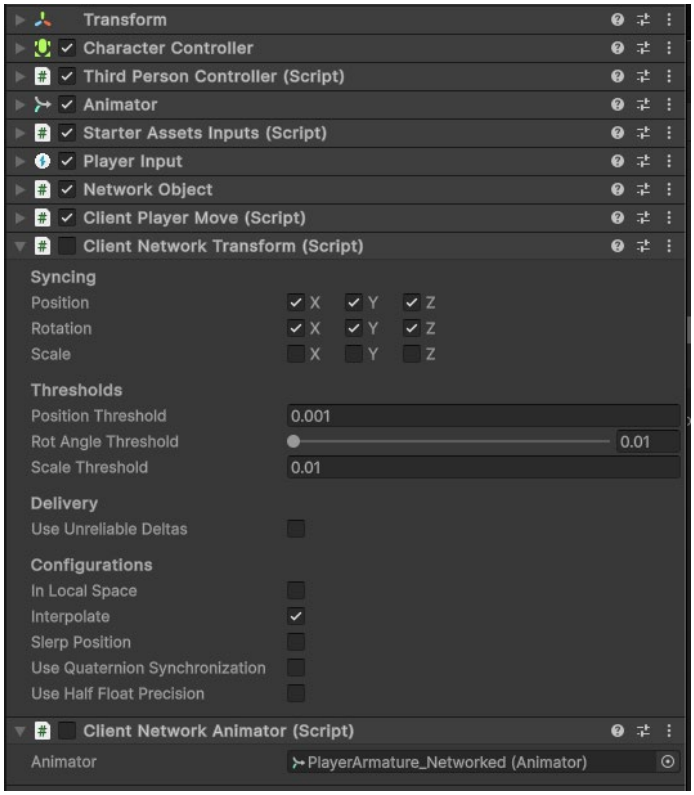
```
using Unity.Netcode.Components;
using UnityEngine;
namespace NetcodeDemo
{
    [DisallowMultipleComponent]
    public class ClientNetworkTransform : NetworkTransform
    {
        protected override bool OnIsServerAuthoritative()
        {
            return false;
        }
    }
}
```

이 코드는 `OnIsServerAuthoritative` 메서드를 오버라이드한 후 false를 반환합니다. 플레이어 NetworkObject 프리팹에서 NetworkTransform을 커스텀 ClientNetworkTransform으로 대체합니다.

마찬가지로, 클라이언트 기반의 NetworkAnimator도 생성할 수 있습니다.

```
using Unity.Netcode.Components;
using UnityEngine;
namespace NetcodeDemo
{
    [DisallowMultipleComponent]
    public class ClientNetworkAnimator : NetworkAnimator
    {
        protected override bool OnIsServerAuthoritative()
        {
            return false;
        }
    }
}
```

NetworkAnimator를 **ClientNetworkAnimator**로 대체합니다. 인스펙터에서 Animator 필드를 꼭 설정하세요.



ClientNetworkTransform과 ClientNetworkAnimator입니다.

Multiplayer Play Mode에서 이제 플레이어를 클라이언트에서 움직일 수 있으며, 해당 위치와 애니메이션 상태가 호스트에 제대로 동기화됩니다.

클라이언트 기반 동작은 네트워크 애플리케이션의 지연을 줄이는 방법이기도 합니다. '소유자 권한 모드'에서는 네트워크 기반의 동작이 즉시 반응하는 것처럼 이루어질 수 있습니다. 클라이언트는 패킷이 서버로 전송되어 다시 돌아올 때까지 기다릴 필요가 없습니다.

하지만 그러한 클라이언트 기반 동작을 만들 때는 각별한 주의가 필요합니다. 각 플레이어의 사용자 경험은 개선되지만, 보안 위험을 초래할 수 있기 때문입니다. 소유자 권한 모드에서는 애플리케이션이 변조 또는 해킹 위험에 노출됩니다. 경쟁형 온라인 게임에서는 플레이어가 부정행위를 저지를 가능성이 높습니다. 이러한 위험을 방지하고 애플리케이션의 보안을 강화하려면 서버 권한을 사용하세요.

소유자 권한 모드 컴포넌트

위의 예시 스크립트를 직접 작성할 수도 있지만, Unity 프로젝트의 Boss Room에 있는 Multiplayer Samples Utilities 패키지(com.unity.multiplayer.samples.coop)에서 사전 제작된 ClientNetworkTransform 및 ClientNetworkAnimator 컴포넌트를 가져올 수도 있습니다.

이 ClientNetworkTransform 구현 시 겪을 수 있는 잠재적인 문제는 다음과 같습니다.

- **소유권 이전**: 소유권이 항상 원활하게 전환되는 것은 아니며, 이 문제로 인해 오브젝트의 위치가 갑자기 바뀌거나 심지어 동기화가 중단되기도 합니다.
- **계층적 소유권:** ClientNetworkTransform은 서버에서 관리하는 NetworkTransform의 자식 요소로 사용할 수 없습니다.
- **업데이트 거부:** 시스템은 클라이언트 소유권만 인식하며 클라이언트와 서버의 공동 소유권을 인식하지 않으므로 서버는 클라이언트로부터의 업데이트를 거부할 수 없습니다.
- **인스턴스화 시 오브젝트 움직임**: 클라이언트에 소유권이 있는 처음 생성된 오브젝트는 서버에서 움직일 수 없습니다.

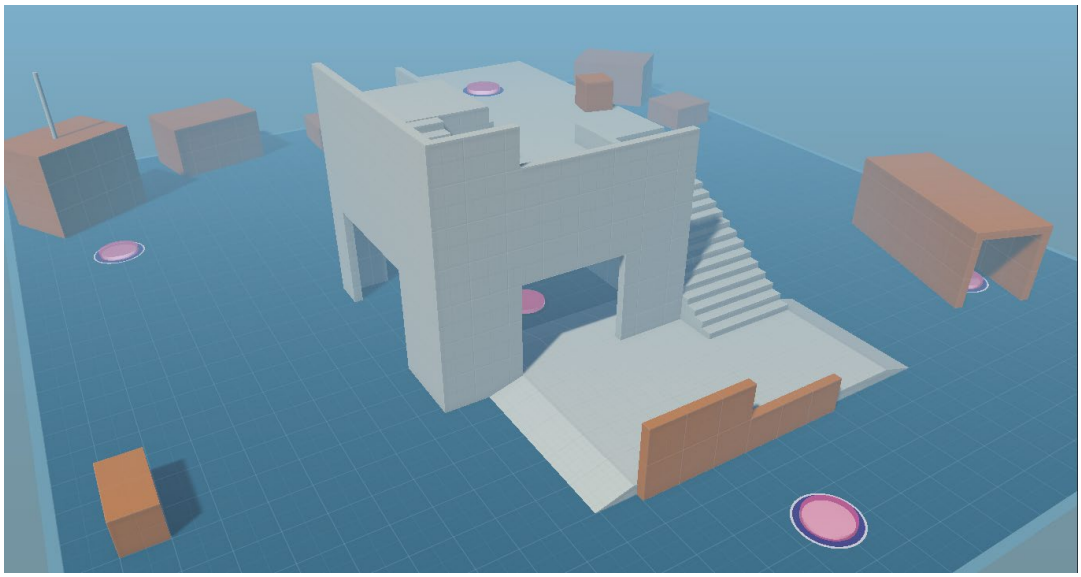
대부분의 경우 클라이언트 소유권 변환을 처리할 때에는 ClientNetworkTransform이 실용적일 수 있습니다. 하지만 이러한 제한을 고려한 다음 프로젝트의 일부로 구현해야 합니다.

서버 권한을 이용한 동기화

게임플레이의 응답 속도를 높이기 위해 클라이언트에 일부 권한을 부여할 수 있지만, 어떤 움직임은 서버 측에서만 할 수 있습니다. 일반적으로, 클라이언트에서 제어하는 동작으로 인해 불균형이나 불공정한 이점이 생기지 않도록 서버 권한을 사용해야 합니다.

예를 들어 게임 맵에서 플레이어가 자신의 생성 위치를 선택할 수 있도록 허용하면 맵 구조에 따라 플레이어가 부당한 이점을 누릴 수 있습니다. 그 대신, 서버가 사전에 정해진 생성 포인트 중 한 곳에 플레이어를 무작위로 배치하는 편이 더 공평합니다.

그렇게 하려면 단순한 외형을 가진 오브젝트를 몇 개 정의한 후, 플레이어가 생성될 가능성이 있는 위치에 여러 개 흩어 놓습니다.



생성 포인트가 레벨 전체에 걸쳐 전략적으로 배치됩니다.

네트워크에 연결되지 않은 MonoBehaviour가 이를 관리할 수 있습니다. 여기서 ServerPlayerSpawnPoints 클래스에는 각 생성 포인트 게임 오브젝트를 참조하는 m_SpawnPoints 목록이 있습니다.

이 샘플 구현에서는 Unity로 제작된 에셋 스토어 프로젝트인 디자인 패턴 및 SOLID 원칙으로 코딩 스킬 업그레이드에서 차용한 다음과 같은 일반 싱글톤 패턴도 사용합니다.

```
public class ServerPlayerSpawnPoints : Singleton<ServerPlayerSpawnPoints>
{
    [SerializeField]
    private List<GameObject> m_SpawnPoints;
    public GameObject GetRandomSpawnPoint()
    {
        if (m_SpawnPoints.Count == 0)
            return null;
        return m_SpawnPoints[Random.Range(0, m_SpawnPoints.Count)];
    }
}
```

ServerPlayerMove라는 NetworkBehaviour는 ServerPlayerSpawnPoints의 인스턴스를 사용하여 생성 포인트를 무작위로 선택할 수 있습니다.

```
using Unity.Netcode;
using UnityEngine;
[DefaultExecutionOrder(0)] // ClientNetworkTransform 전에 실행
public class ServerPlayerMove : NetworkBehaviour
{
    public override void OnNetworkSpawn()
    {
        // 서버에서만 실행
        if (!IsServer)
        {
            enabled = false;
            return;
        }
        SpawnPlayer();
        base.OnNetworkSpawn();
    }

    // 생성 시 일괄 처리 가능한 다음 위치로 이동
    void SpawnPlayer()
    {
        var spawnPoint = ServerPlayerSpawnPoints.Instance.GetRandomSpawnPoint();
        var spawnPosition = spawnPoint ? spawnPoint.transform.position : Vector3.zero;
        transform.position = spawnPosition;
    }
}
```

모든 로직은 OnNetworkSpawn에서 실행됩니다. 클라이언트가 연결될 때마다 SpawnPlayer가 호출되어 무작위로 선택된 생성 포인트에서 플레이어가 시작하도록 합니다. IsServer 검사는 게임 상태에 대한 권한이 있는 서버에서만 이 로직이 실행되도록 합니다.

ServerPlayerMove 스크립트를 PlayerArmature_Networked 프리팹에 추가합니다. Multiplayer Play Mode에 진입하면 각 클라이언트가 연결되어 플레이그라운드 환경 내의 무작위 지점에 생성됩니다.

이 구현은 NetworkBehaviour가 네트워크를 통해 제어되지 않는 씬에 포함된 요소와 상호 작용하는 방식을 보여줍니다. 여기에서 NetworkBehaviour는 계층 구조에 이미 설정된 정적 데이터 및 씬 오브젝트를 활용합니다.

클라이언트가 연결되면 ServerPlayerMove는 기존 게임플레이 씬에서 무작위 생성 포인트 하나만 가져와야 합니다. 이로 인해 네트워크를 통해 전송되는 데이터의 양이 제한됩니다.

플레이어가 무작위 생성 포인트에 나타납니다.

몇 가지 중요한 사항:

— ClientNetworkTransform은 소유자 권한에 의해 작동하므로 Awake 중에는 CharacterController 컴포넌트를 비활성화해야 합니다. 계산된 값이 오버라이드되지 않고 위치가 월드 중심점으로 재설정되지 않도록 ServerPlayerMove에서 플레이어를 배치한 후에는 CharacterController를 다시 활성화합니다.

— 마찬가지로, 플레이어가 (0,0,0)에서 생성되지 않도록 인스펙터에서 m_SpawnPoints를 선택합니다.

— DefaultExecutionOrder 속성을 더 낮은 값으로 설정하여 `ServerPlayerMove`를 `ClientPlayerMove`보다 먼저 실행합니다. 예를 들어 `[DefaultExecutionOrder(0)] prioritizes ServerPlayerMove`를 사용하면 플레이어가 먼저 달리도록 할 수 있습니다.

멀티플레이어 프로젝트에서 이제 여러 클라이언트를 호스트에 연결할 수 있습니다. 게임 내에서 3인칭 플레이어 캐릭터는 레벨 내 지정된 위치에서 생성될 수 있고, 움직임과 애니메이션을 실시간으로 동기화할 수 있습니다.

멀티플레이어 경험에서 동기화는 필수입니다. NetworkTransform, NetworkAnimator와 같은 컴포넌트로 이 프로세스를 간편하게 구현할 수 있지만, 게임플레이의 경우 NetworkBehaviour도 커스터마이즈해야 합니다.

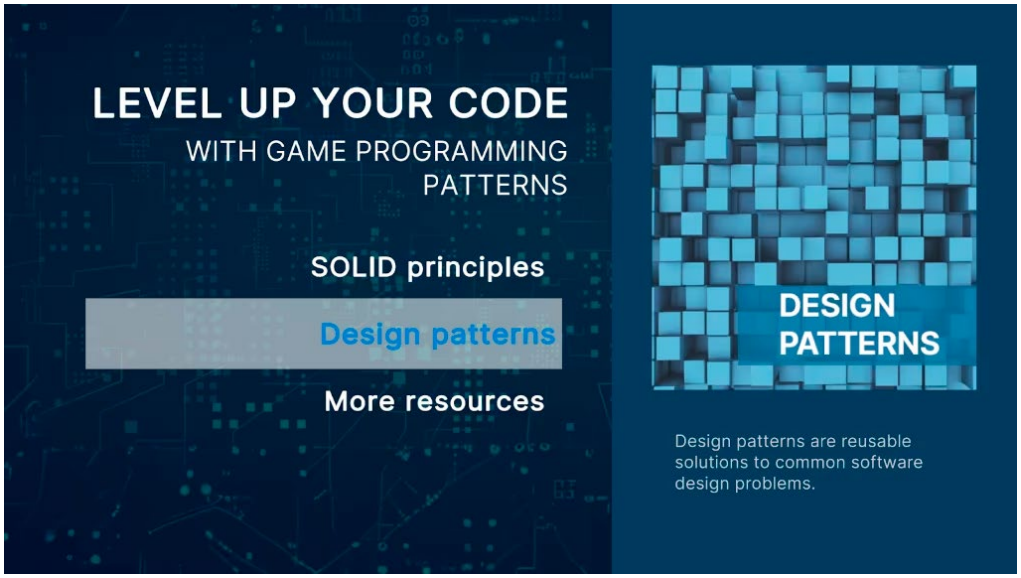
다음으로는 네트워크에서 데이터와 게임 상태를 동기화하는 방법을 추가로 살펴보겠습니다.

싱글톤 디자인 패턴

싱글톤은 런타임 시 특정 유형의 고유 인스턴스에 액세스할 수 있는 편리한 수단을 제공합니다. 하지만 싱글톤을 사용하면 종속성이 높아질 수 있으므로, 그러한 단점에 유의해야 합니다.

Netcode for GameObjects에서는 `NetworkManager.Singleton`을 참조할 때마다 싱글톤을 사용합니다. 샘플 프로젝트에는 모든 MonoBehaviour 유형에 사용할 수 있는 일반 싱글톤의 예시도 포함되어 있습니다.

싱글톤에 대해 자세히 알아보려면 디자인 패턴 및 SOLID 원칙으로 코딩 스킬 업그레이드 전자책을 참조하세요. 이 가이드북에서는 씬에서 오브젝트 통신을 하기 위한 이벤트 또는 이벤트 채널과 같은 대체 패턴도 소개합니다.



Unity 디자인 패턴에 대한 무료 전자책을 받아 보세요. 프로그래머, 테크니컬 아티스트, 아티스트, 디자이너를 위한 모든 고급 가이드는 Unity 베스트 프랙티스 허브를 확인하세요.

네트워크 동기화

네트워크 동기화는 모든 플레이어에게 일관되고 공정한 게임 경험을 유지하는 데 필수적입니다.

이미 클라이언트 기반 모델을 사용하여 플레이어 NetworkObject를 통해 플레이어의 움직임과 애니메이션을 동기화하는 방법을 알아보았습니다. 그러나 게임플레이는 플레이어 캐릭터 외에 많은 요소를 동반하는 경우가 많습니다. 게임 디자인에 따라 플레이어는 발사체를 쏘거나, 문을 열거나, 다른 씬 오브젝트와 상호 작용해야 할 수도 있습니다. 이러한 상호 작용은 클라이언트와 서버 간에 동기화해야 하는 게임 상태와 함께 네트워크를 통해 연동해야 합니다.

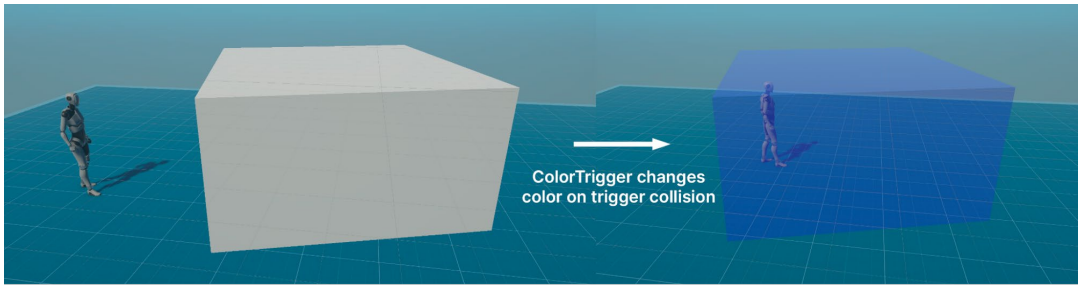
이 섹션에서는 플레이어가 게임 환경의 일부와 상호 작용할 수 있는 게임플레이 동작을 위한 클라이언트-서버 통신을 설정합니다. 그러한 과정에서 네트워크로 연동되는 게임 상태를 구현하고 서버에서 RPC(원격 프로시저 호출)를 주고받는 작업이 수반됩니다.

게임플레이 메카닉

서버-클라이언트 통신을 설명하기 위해 간단한 게임 메카닉을 재현하겠습니다.

먼저, 플레이어와 접촉하면 색상이 변하는 트리거 콜라이더를 게임 레벨에 추가해 보겠습니다. 한 플레이어가 이를 터치하면 한 색상으로 변하고, 다른 플레이어가 터치하면 다른 색상으로 변합니다.

씬에서 새로운 게임 오브젝트(예: 큐브)를 생성합니다. BoxCollider 컴포넌트를 추가하고 IsTrigger 옵션을 활성화합니다. 투명한 머티리얼을 생성하고 MeshRenderer에 할당합니다(이 예시에서는 URP/Lit 셰이더 사용). 초기 색상을 흰색과 같은 무채색으로 설정합니다.



해당 트리거는 네트워크를 통해 색상 값을 수신합니다.

다음으로는 네트워크 동기화를 추가해야 합니다. NetworkVariable과 RPC를 사용하여 색상의 변화를 네트워크에서 동기화하겠습니다.

NetworkVariable 정의

NetworkVariable은 네트워크에서 동기화된 상태를 관리하는 데 특화된 변수입니다. 서버에서 NetworkVariable이 변경되면 해당 변경 사항이 모든 클라이언트에 전파됩니다. 이러한 동작은 위치, 체력 포인트 또는 이 예시의 트리거 색상 상태와 같이 지속적인 동기화가 필요한 데이터에 적합합니다.

`ColorTrigger`라는 NetworkBehaviour를 생성하여 트리거 오브젝트에 연결합니다. 이 스크립트에는 `Color` 값을 포함한 `m_NetworkColor`라는 NetworkVariable이 있습니다.

```
using UnityEngine;
using Unity.Netcode;
public class ColorTrigger : NetworkBehaviour
{
   public NetworkVariable<Color> m_NetworkColor = new NetworkVariable<Color>(Color.white);
   private Material m_InstanceMaterial;

   public override void OnNetworkSpawn()
   {
       m_NetworkColor.OnValueChanged += OnColorChanged;
       MeshRenderer meshRenderer = GetComponent<MeshRenderer>();
       if (meshRenderer != null)
       {
           m_InstanceMaterial = new Material(meshRenderer.material);
           meshRenderer.material = m_InstanceMaterial;
           UpdateMaterialColor(m_NetworkColor.Value);
       }
   }
```

```
    public override void OnNetworkDespawn()
    {
        m_NetworkColor.OnValueChanged -= OnColorChanged;
    }
    private void OnColorChanged(Color oldColor, Color newColor)
    {
        UpdateMaterialColor(newColor);
    }

    private void UpdateMaterialColor(Color newColor)
    {
        if (m_InstanceMaterial != null)
        {
            m_InstanceMaterial.SetColor("_BaseColor", newColor);
        }
    }
}
```

플레이어가 트리거에 들어가면 이 스크립트가 해당 머티리얼 인스턴스의 기본 색상 프로퍼티를 토글합니다. 이 스크립트는 `m_NetworkColor`라는 NetworkVariable을 사용합니다.

작동 원리를 요약하면 다음과 같습니다.

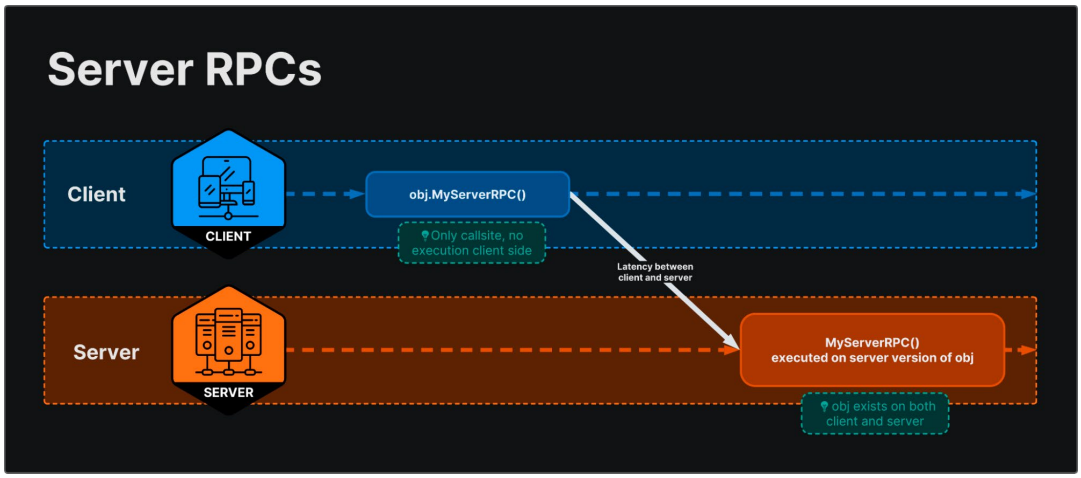
- 이 NetworkVariable은 실제 색상 값을 추적한 후 모든 클라이언트 간에 동기화합니다. 이 스크립트는 서버와 클라이언트 모두에서 실행되지만, 기본적으로 NetworkVariable에 대한 쓰기 권한은 서버에만 있습니다. 클라이언트는 해당 `Color` 값을 읽을 수만 있습니다.
- `OnValueChanged` 이벤트는 NetworkVariable이 변경될 때마다 트리거의 머티리얼 기본 색상을 업데이트합니다. 이 스크립트는 `OnNetworkSpawn` 이벤트를 구독하고 `OnNetworkDespawn`을 구독 해제합니다.

NetworkTransform은 트랜스폼 데이터(위치, 회전, 스케일)를 동기화하는 데 특화됐지만, NetworkVariable은 기본 유형, 커스텀 구조체, 그 외 게임 상태 관리에 필요한 데이터 등의 보다 일반적인 데이터 유형을 동기화할 수 있습니다.

NetworkVariable에 대한 권한은 서버에 있으므로 클라이언트에서 작업할 때는 이를 직접 변경할 수 없습니다. 그 대신 클라이언트는 변경 사항을 적용하겠다고 서버에 통지해야 합니다. 서버가 상태를 업데이트하고 이를 클라이언트에 다시 전파한 후에야 클라이언트에 변경 사항이 적용됩니다.

이러한 변경 사항에 대한 통신을 처리하려면 RPC를 사용합니다. RPC를 사용하면 한 기기가 다른 기기에 특정 동작이나 업데이트를 수행하도록 요구할 수 있습니다. RPC는 클라이언트에서 서버로 또는 그 반대 방향으로 호출될 수 있습니다.

이를 목표로 RPC를 구현하는 방법을 살펴보겠습니다.



서버 RPC는 클라이언트에서 서버로 원격 실행됩니다.

RPC 추가

RPC 를 사용하면 서버 또는 다른 클라이언트에서 함수를 원격으로 호출할 수 있습니다. `[Rpc(SendTo.Server)]`로 표시된 메서드는 클라이언트에서 서버로 호출되며, `[Rpc(SendTo.Client)]`로 표시된 메서드는 서버에서 클라이언트로 호출됩니다. 여기서 레거시 구문인 `[ServerRpc]` 또는 `[ClientRpc]`를 사용하여 서버 RPC 또는 클라이언트 RPC를 나타내도 됩니다.

RPC는 다음과 같은 몇 가지 규칙을 준수해야 한다는 점을 제외하고는 다른 메서드와 크게 다르지 않습니다.

— **Rpc 속성:** 메서드에 `[Rpc]` 속성을 추가하고 가능한 타겟을 지정합니다. 예를 들어 `[Rpc(SendTo.Server)]`는 서버에서만 메서드를 호출합니다.

— **명명 규칙:** 메서드 이름 끝에 접미사 'Rpc'를 붙입니다. 예를 들면 `DoSomethingRpc`입니다.

RPC는 플레이어의 동작이나 특정한 게임 상태 변화와 같이 연속적인 동기화가 필요하지 않은 별개의 이벤트에 더 적합합니다.

다음과 같은 메서드를 TriggerColor 스크립트에 추가합니다.

```
private void OnTriggerEnter(Collider other)
 {
     NetworkObject networkObject = other.GetComponent<NetworkObject>();
     if (IsClient && networkObject != null && networkObject.IsOwner)
     {
         ChangeColorServerRpc(networkObject.OwnerClientId);
     }
 }
```

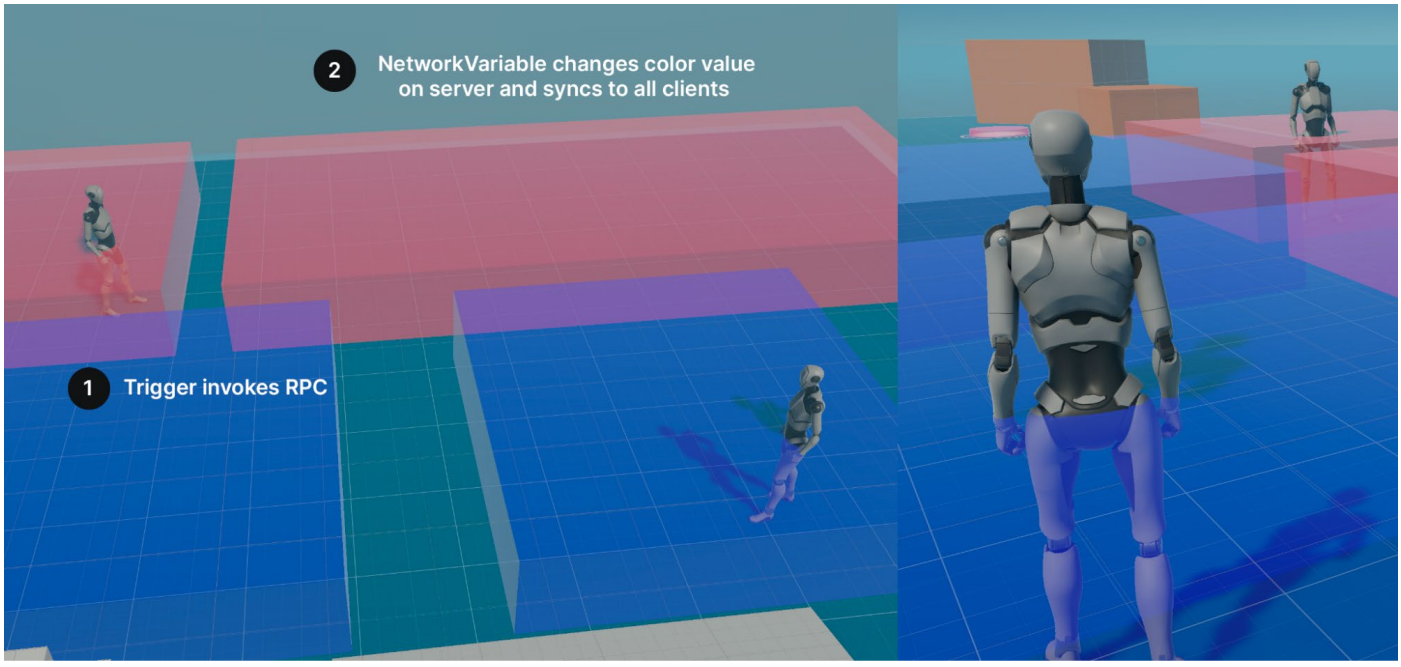
```
    [Rpc(SendTo.Server)]
private void ChangeColorServerRpc(ulong playerId)
{
    // 간단한 팀 시스템 : 짝수는 파란색, 홀수는 빨간색
    Color newColor =
        (playerId % 2 == 0) ? new Color(0, 0, 1, 0.5f) : new Color(1, 0, 0, 0.5f);

    m_NetworkColor.Value = newColor;
}
```

싱글플레이어 게임에서는 `OnTriggerEnter`에 적절한 로직을 추가하면 됩니다. 그러나 멀티플레이어 게임에서는 일반적으로 클라이언트-서버 통신을 사용해 상호 작용을 처리하여 모든 클라이언트의 동기화 상태를 유지합니다.

`m_NetworkColor` 값을 직접 설정하는 대신, `OnTriggerEnter`에서 현재 게임 인스턴스가 클라이언트인지 확인합니다. 그러한 경우에는 `ChangeColorServerRpc`를 호출하고 `OwnerClientId`를 전달합니다.

이 메서드는 플레이어의 ID를 기반으로 새로운 색상을 결정(이 예시에서는 짝수 ID는 파란색, 홀수 ID는 빨간색으로 표시됨)하고 `m_NetworkColor` 값을 업데이트합니다.

이 변경 사항은 연결된 모든 클라이언트에 전파되어 모든 게임 인스턴스에서 게임 상태가 일관되게 보이도록 합니다. `m_NetworkColor`가 변경되면 모든 클라이언트에서 `OnColorChanged` 메서드가 트리거되어 트리거의 머티리얼 색상이 업데이트됩니다.



이 간단한 게임 메카닉은 클라이언트와 서버 간의 상호 작용을 보여줍니다.

트리거 메카닉

이 예시는 비록 간단하지만, 플레이어의 동작이 환경에 대해 시각적인 변화를 트리거하는 것과 유사한 게임 메카닉은 많은 멀티플레이어 게임에서 찾아볼 수 있습니다. 예를 들면 다음과 같습니다.

- 협동 퍼즐 게임에서 플레이어가 버튼이나 트리거를 사용해 환경과 상호 작용해야 하는 경우가 있습니다. 게임 환경에서 발생하는 시각적 신호는 모든 플레이어가 그에 맞춰 행동을 조정하도록 유도하는 신호입니다.
- 경쟁형 슈팅 게임에서 플레이어는 레벨의 특정 지점을 점령할 수 있습니다. 이 지점은 점령하고 있는 팀의 색상으로 바뀌는 경우가 많습니다. 마찬가지로 이것도 플레이어들이 그에 맞게 전략을 조정하게 하는 시각적 신호로 작용합니다.
- 온라인 RPG에서 레벨의 특정 지역은 버프, 디버프 또는 기타 효과를 트리거할 수 있습니다.

RPC와 NetworkVariable 비교

데이터를 동기화할 때 NetworkVariable과 RPC 중 어느 것을 사용할지는 활용 사례에 따라 다릅니다.

- NetworkVariable과 NetworkTransform은 위치, 체력 포인트 또는 이 예시의 트리거 색상 상태와 같이 지속적인 동기화가 필요한 데이터에 적합합니다. 여기에서 색상 트리거는 NetworkVariable을 사용하여 활성 색상을 저장합니다.
    - 활용 사례: 체력 포인트, 위치 데이터, 게임 점수
- RPC(원격 프로시저 호출)는 플레이어의 동작이나 특정한 게임 상태 변화와 같이 연속적인 동기화가 필요하지 않은 별개의 이벤트에 더 적합합니다. 이 예시에서 RPC는 플레이어가 트리거 구역에 진입하면 모든 클라이언트에게 알리고 서버가 플레이어의 ID에 따라 색상을 변경하도록 안내합니다.
    - 활용 사례: 플레이어 동작(예: 발사, 능력 사용), 게임 이벤트(예: 생성, 게임 시작, 게임 승리)

NetworkVariable은 지속되는 상태를 관리하고, RPC는 특정 이벤트와 동작을 처리한다는 점에서 두 메커니즘 모두 네트워크 게임에서 데이터를 동기화하는 데 필수적입니다.

다음과 같은 게임플레이 예시가 어떤 경우에 NetworkVariable 또는 RPC를 사용하는지 파악하는 데 도움이 될 것입니다.

작업/시스템	RPC	NetworkVariable
인벤토리 관리	아이템을 획득하거나 사용하여 아이템이 인벤토리에서 추가되거나 제거되면 클라이언트에 알립니다.	각 플레이어의 현재 인벤토리를 유지합니다. 인벤토리는 아이템 목록을 동기화하는 NetworkList일 수 있습니다.
전투 시스템	공격이나 특수 기술과 같은 전투 동작을 수행합니다. RPC는 대미지와 효과를 적용할 수 있습니다.	각 플레이어의 체력과 상태를 추적합니다. 체력 포인트와 발동된 상태 효과는 NetworkVariable을 사용하여 동기화할 수 있습니다.
환경 상호 작용	문을 열거나 메커니즘을 발동하는 등의 특정 동작을 트리거합니다.	문의 열림, 닫힘 여부와 같은 인터랙티브 오브젝트의 상태를 유지합니다.
목표	목표가 완료되었음을 알립니다.	목표 달성도를 추적합니다. 수집한 아이템 수 또는 완료한 과제 수를 NetworkVariable에 저장합니다.

자세한 내용은 RPC와 NetworkVariable 비교 기술 자료 페이지를 참조하세요.

멀티플레이어를 고려한 디자인

지금까지 Netcode의 기본 사항을 익혔으므로, 이제는 게임을 제작할 때 '네트워크 멀티플레이어'를 꼭 염두에 두어야 합니다. 싱글플레이어 게임에서는 간단했던 동작이 여러 기기에서 동일하게 구현하기 위해 NetworkVariable 또는 RPC를 사용해야 할 때는 더 복잡해질 수 있습니다.

클라이언트와 서버 간 상태를 어떻게 동기화할지 계획하세요. 지속적으로 동기화되는 데이터에는 NetworkVariable을, 별개의 이벤트에는 RPC를 사용하세요. 그런 다음, 꼭 필요한 것만 동기화하여 네트워크 트래픽을 최소화하세요.

싱글플레이어 게임을 멀티플레이어 게임으로 전환하는 것도 가능하지만, 처음부터 멀티플레이어를 고려해 디자인하는 편이 더 효율적입니다. 서버가 소유할 오브젝트와 동작, 클라이언트가 관리할 수 있는 오브젝트와 동작을 미리 결정하세요. 서버 권한이 가장 높은 보안 수준과 일관성을 제공하기는 하지만, 특정 동작의 지연을 줄이려면 클라이언트 권한도 균형 있게 사용해야 합니다.

이 지연은 네트워크 게임 개발에서 매우 곤란한 문제가 될 수 있으므로, 이제 지연이 멀티플레이어 경험에 어떤 영향을 미치는지 살펴보겠습니다.

아이디어가 필요하다면 이 유나이트 2024 세션을 참조해 보세요. 해당 세션에서는 멀티플레이어 게임을 개발할 때 저지르는 가장 흔하면서도 중대한 실수와 이를 방지하여 성공 가능성을 높이는 방법을 살펴봅니다.

네트워크 지연과 성능

온라인 게임을 해 보셨다면 지연이 어떻게 게임 경험을 저해하는지 몸소 경험해 보셨을 것입니다. 대역폭이 충분하지 않고 네트워킹 연결이 불안정하면 플레이어의 움직임이 끊어지고, 프레임 속도가 일관되지 않으며, 입력 지연이 체감될 만큼 발생할 수 있습니다.

이제 클라이언트와 서버의 통신 방식을 파악하셨으므로 이러한 현상이 왜 일어나는지 이해하실 것입니다. 인터넷이 기기 사이를 연결하는 역할을 하면서 여러 문제가 발생할 수 있습니다. Unity Transport의 우수한 안정성에도 불구하고, UDP 패킷이 목표 지점에 도달하는 과정에서 손실되거나, 순서가 뒤바뀌거나, 손상될 수 있습니다. 이는 지연이나 렉을 일으키는 잠재적인 원인이 됩니다.

지연 시뮬레이션

개발 도중 지연의 영향을 파악하고 완화하기 위해 UnityTransport의 Debug Simulator 또는 Network Simulator를 사용하여 Unity 에디터에서 다양한 네트워크 상태를 시뮬레이션할 수 있습니다.

Unity Transport Debug Simulator

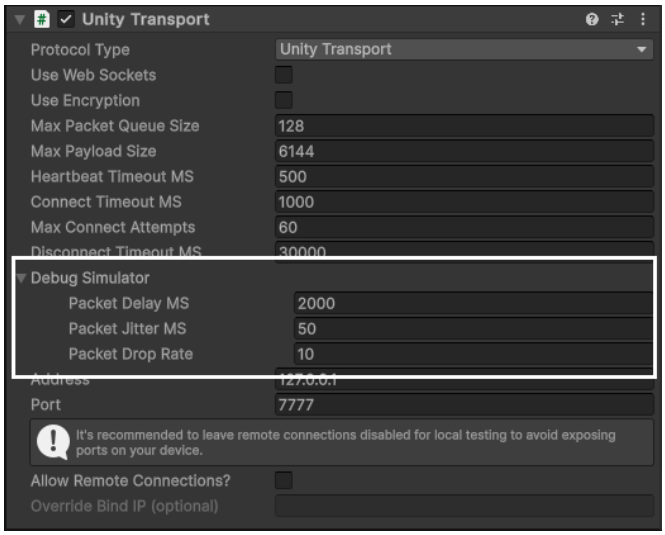
UnityTransport 컴포넌트에서 Debug Simulator를 조정하여 지연, 지터, 패킷 손실을 인위적으로 유발할 수 있습니다. 해당 설정은 씬에서 NetworkTransport 오브젝트를 선택하면 인스펙터에서 찾을 수 있습니다. 몇 가지 값을 설정하면 네트워크 혼잡을 재현할 수 있습니다.

Latency: 밀리초 단위로 지연을 추가하여 더 느린 네트워크 연결 상태를 시뮬레이션합니다. 예를 들어 지연을 100ms로 설정하여 보통 수준의 네트워크 지연을 모방할 수 있습니다.

Jitter: 지연에 변동성을 주어 변동이 심한 네트워크 상태를 시뮬레이션합니다. 예를 들어 지터를 50ms로 설정하여 연결 불안정성이 게임플레이에 미치는 영향을 확인할 수 있습니다.

Packet Loss: 패킷의 일정 비율을 버림으로써 저품질 연결 상태를 모방합니다. 패킷 손실을 5%로 설정하여 게임플레이에 미치는 영향을 확인해 봅니다.

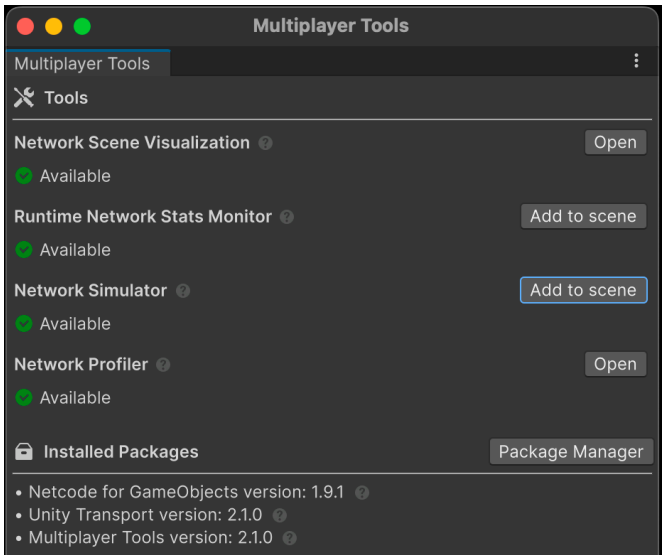
게임 애플리케이션을 다시 플레이하여 이렇게 시뮬레이션된 조건이 게임플레이에 어떤 영향을 미치는지 관찰합니다.



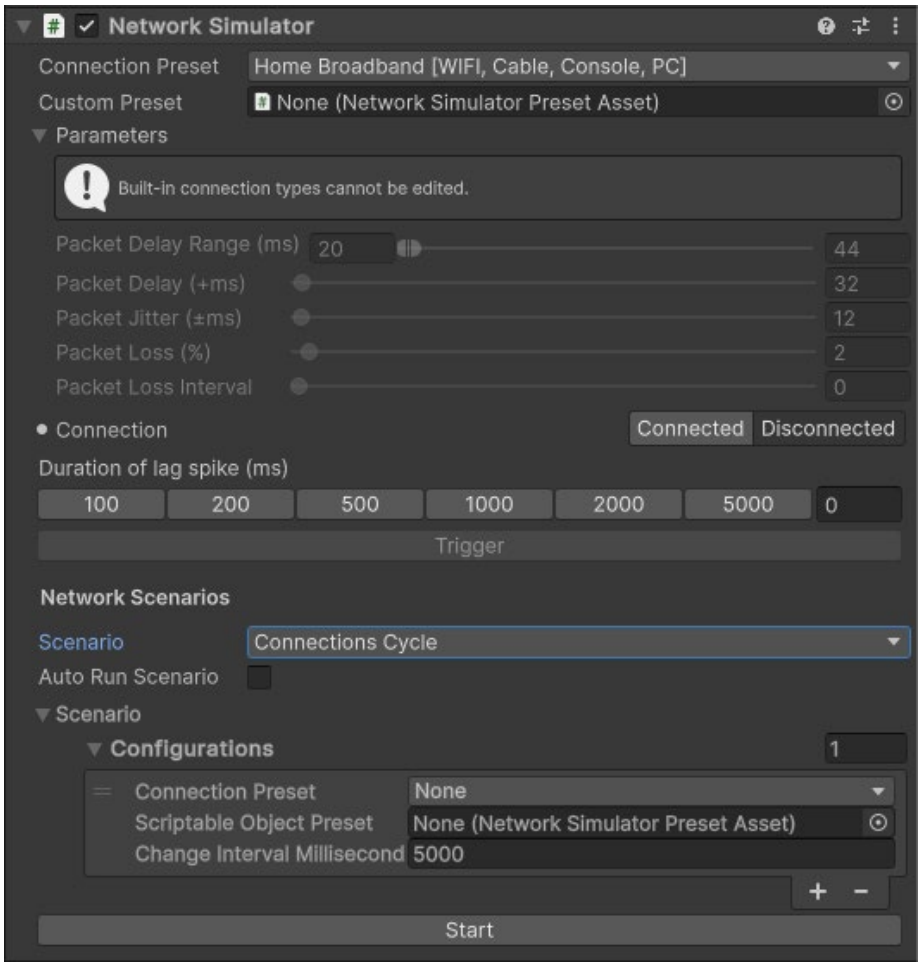
Unity Transport의 Debug Simulator를 조정합니다.

Network Simulator

Multiplayer Tools 패키지의 Network Simulator를 사용하면 최적이 아닌 네트워크 상태를 테스트할 수 있습니다. 이를 통해 제작 단계에서 문제가 발생하기 전에 발견하고 해결할 수 있으며, 네트워크 연결 끊김, 지연 급증, 패킷 손실 등의 네트워크 이벤트를 손쉽게 시뮬레이션할 수 있습니다.



Multiplayer Tools 패키지에서 Network Simulator를 설치합니다.



Network Simulator에서 지연을 테스트합니다.

기타 네트워크 컨디셔너

단, Debug Simulator 또는 Network Simulator는 에디터에서만 작동합니다. 런타임 빌드의 경우 대체 네트워크 컨디셔너를 사용하여 지연을 시뮬레이션할 수 있습니다. Windows용 clumsy, macOS/iOS용 Network Link Conditioner 등의 툴을 사용하면 다양한 네트워크 상태를 재현하여 철저히 테스트할 수 있습니다.

자세한 내용은 이 가이드 후반부의 테스트 및 디버그 섹션을 참조하세요.

지연이 애플리케이션 성능에 미치는 영향에 대응하는 것은 멀티플레이어 개발의 매우 중요한 과제 중 하나입니다. 하지만 이런 지연의 영향을 완화하는 데 도움이 되는 전략이 있습니다. 그중 몇 가지를 살펴보겠습니다.

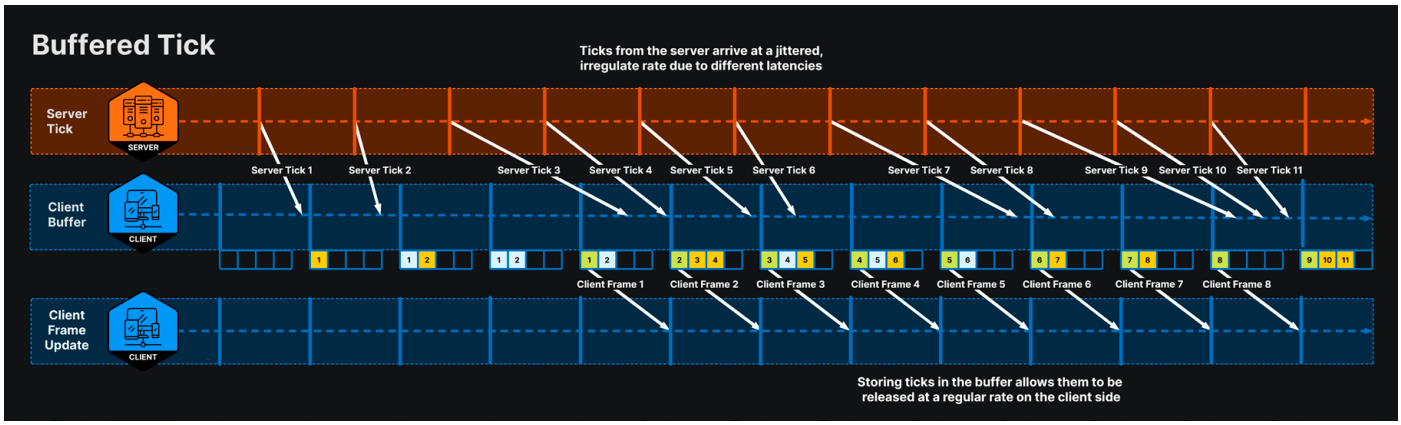
클라이언트측 보간

지연의 영향을 줄이는 방법 중 한 가지는 클라이언트측 보간입니다. 이 방법에서는 각 클라이언트가 즉시 렌더링하지 않고 짧은 보간 기간에 렌더링을 의도적으로 지연시킵니다.

클라이언트-서버 토폴로지에서 클라이언트는 일반적으로 서버보다 RTT(왕복 시간)의 절반 정도 늦게 상태를 렌더링합니다. 클라이언트측 보간은 여기에 의도적으로 지연을 추가합니다.

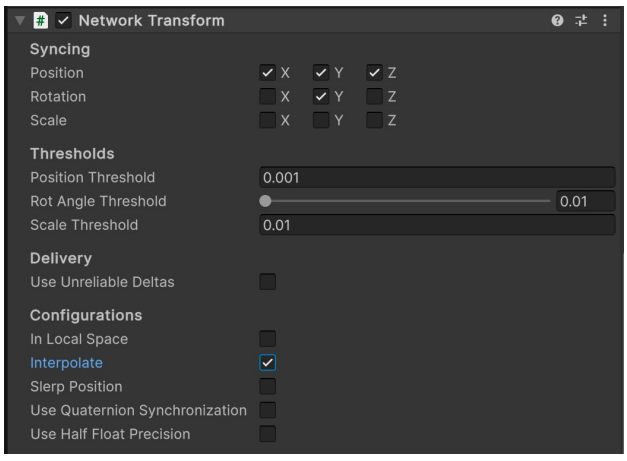
약간 늦은 실행을 통해 클라이언트가 서버에서 들어오는 상태 업데이트에 대한 버퍼 역할을 수행합니다. 다음 업데이트를 렌더링할 때 클라이언트는 최근 두 차례의 서버 틱에서 보간된 상태를 계산합니다.

이 버퍼 덕분에 클라이언트는 클라이언트 업데이트를 규칙적으로, 심지어 지터가 발생하여 불규칙한 속도로 서버의 틱이 도달하는 경우에도 렌더링할 수 있습니다. 보간된 상태로 인해 사소한 지연이나 지터는 감춰집니다.



클라이언트는 버퍼에서 보간된 상태를 렌더링합니다.

클라이언트측 보간은 Netcode for GameObjects에서 NetworkTransform 컴포넌트의 플래그로 사용할 수 있습니다. **Interpolation**를 활성화하면 관련된 게임 오브젝트의 위치, 회전, 스케일이 보간됩니다.



NetworkTransform에서 클라이언트측 보간이 활성화되었습니다.

클라이언트측 보간은 렌더링에 약간의 지연을 발생시키며 보간하려면 이러한 지연이 불가피하다는 점에 유념하시기 바랍니다.

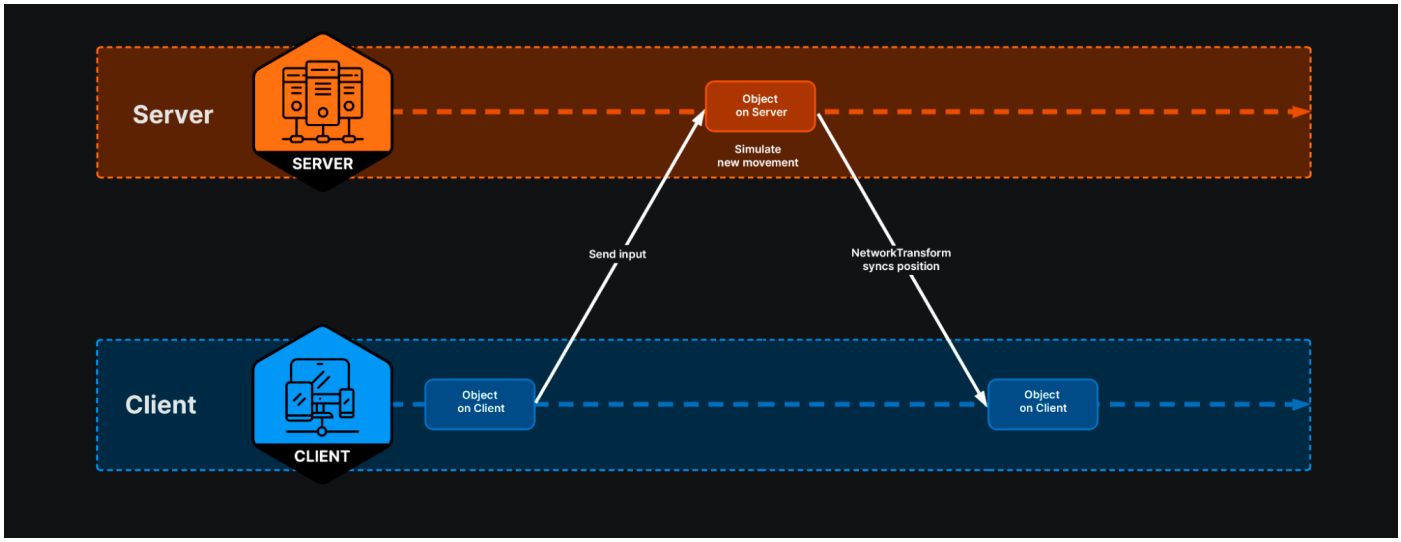
클라이언트측 예측 및 예상

이전 예시의 ClientNetworkTransform은 플레이어의 움직임에 대한 직접적인 제어 권한을 클라이언트에 부여하여, 게임이 더 즉각적으로 빠르게 느껴지도록 만듭니다. 그러나 대다수 게임에서는 게임 디자인, 공정성, 보안 등의 이유로 클라이언트 권한을 부여하기 어렵다는 점에 유의해야 합니다.

서버 권한을 사용하는 이유

공정한 경쟁이 중요한 경쟁형 게임에서 클라이언트에 권한을 부여하면 플레이어가 클라이언트측 코드나 데이터를 조작하여 게임에서 부정행위를 저지르거나 시스템을 악용할 수 있습니다. 플레이어 상호 작용이 복잡하고 게임 월드가 공유되는 게임에서 데이터 무결성을 확보하고 변조를 방지하며, 모든 플레이어가 일관된 경험을 할 수 있도록 서버 권한이 필요한 경우가 많습니다. 이 경우 공정한 플레이 환경과 게임의 무결성을 유지하기 위해 서버 권한이 필요합니다.

그러므로 소유자에게 권한이 있는 ClientNetworkTransform 대신, 표준 NetworkTransform 컴포넌트를 사용해야 합니다. 이 컴포넌트를 사용하면 클라이언트 입력으로는 플레이어를 제어할 수 없고 오직 서버만 제어 권한을 가집니다.



서버 권한을 사용하여 NetworkTransform 이동

단, 서버 권한은 지연을 악화할 수 있습니다. 각 클라이언트는 화면상의 플레이어를 직접 조작할 수 없고 입력을 서버로 전송해야 합니다. 그러면 서버가 해당 입력을 처리하여 게임을 시뮬레이션하고 게임 상태를 계산합니다. 클라이언트는 이 시뮬레이션 결과를 수신해야만 다음 프레임을 표시할 수 있습니다.

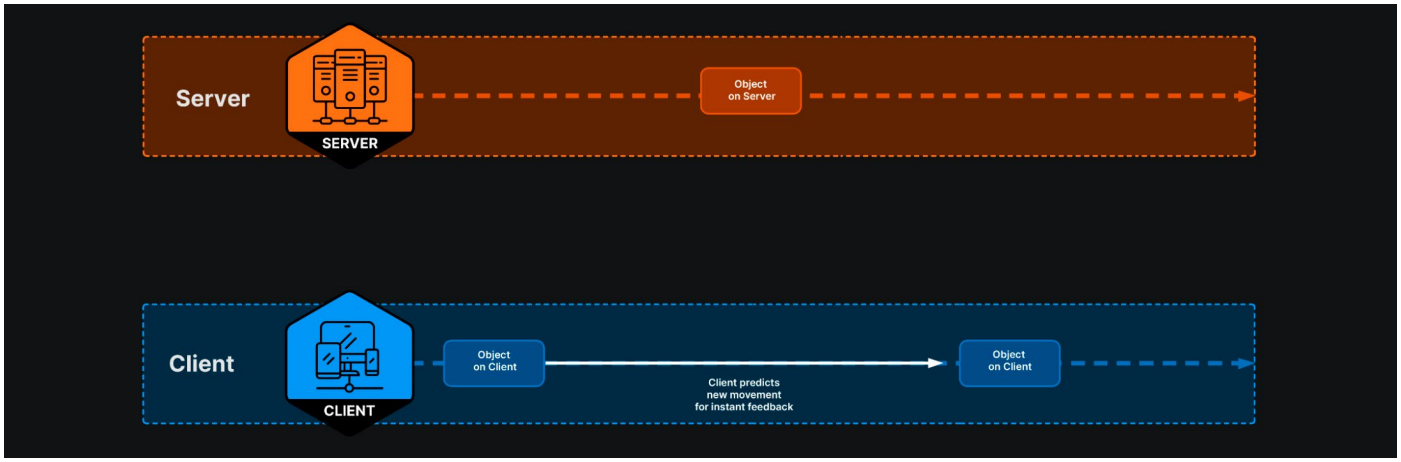
권한 서버를 사용하면 데이터가 클라이언트에서 서버로 이동한 후 다시 클라이언트로 돌아오는 데 걸리는 시간 때문에 플레이어가 지연을 체감하게 됩니다. 이러한 경우에는 특히 UDP 패킷이 인터넷을 통과해야 하므로 클라이언트가 서버보다 늦게 반응하는 경향을 보입니다.

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 예측의 작동 원리

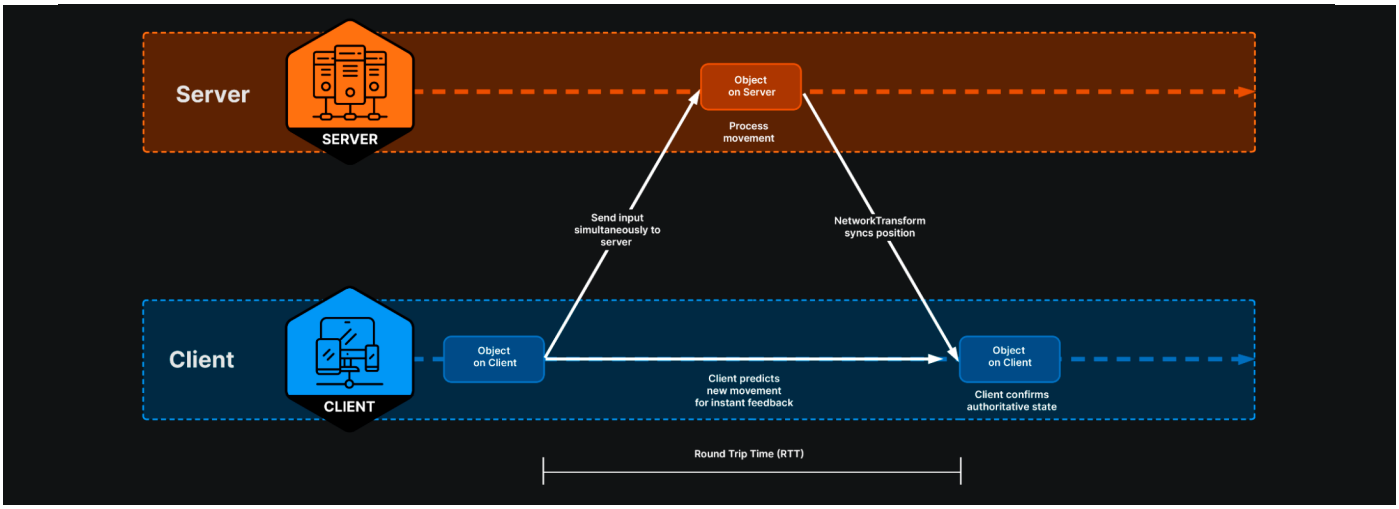
클라이언트측 예측은 서버 권한으로 인해 발생하는 지연에 대한 한 가지 해결책입니다. 클라이언트는 서버의 응답을 기다리지 않고 코앞의 게임 상태를 가늠하여 게임을 시각적으로 업데이트합니다. 이러한 방식은 클라이언트가 서버의 실제 상태를 알지 못한 채 게임 상태를 가늠하므로 '예측'이라고 합니다.

클라이언트는 예측을 통해 즉시 플레이어의 동작에 대한 시각적 피드백을 제공하여 게임의 반응이 빠르다고 느끼게 할 수 있습니다.



클라이언트가 게임 상태를 예측합니다.

동시에, 클라이언트는 플레이어의 동작을 패킷 형태로 서버에 전송합니다. 서버는 클라이언트의 입력 패킷을 수신하고 해당 입력을 사용하여 게임 상태를 시뮬레이션합니다. 서버는 이러한 입력을 처리하여 게임 상태를 시뮬레이션하고, 권한 게임 상태를 실측 데이터로 삼습니다. 서버가 권한 상태를 클라이언트로 다시 보냅니다.

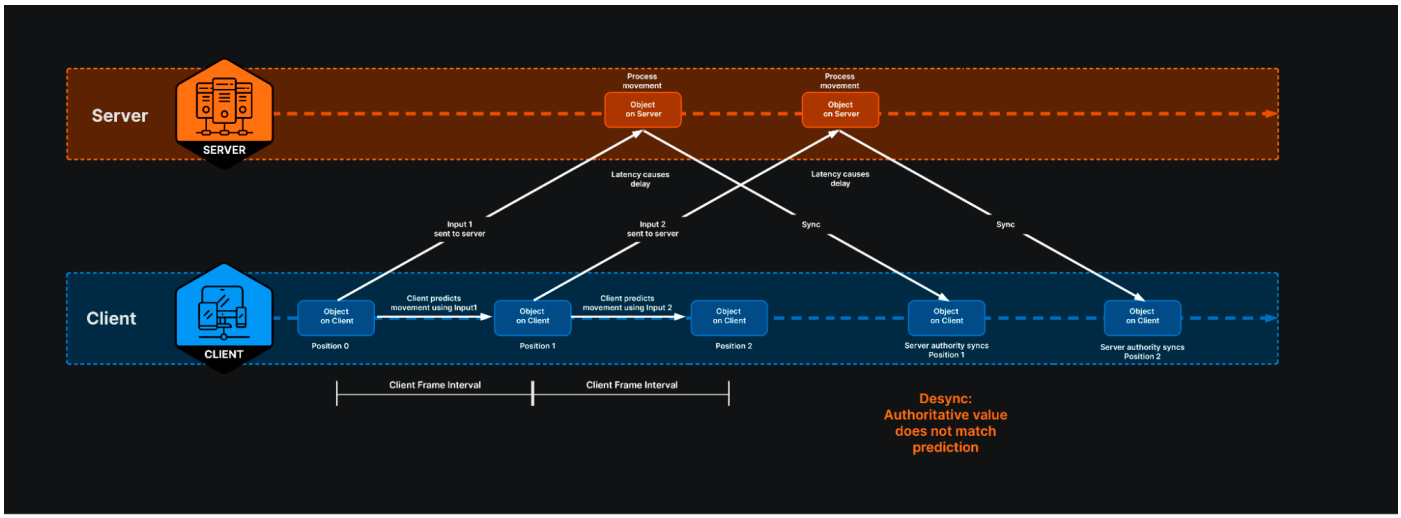


서버가 권한 상태를 다시 보냅니다.

조정 및 롤백

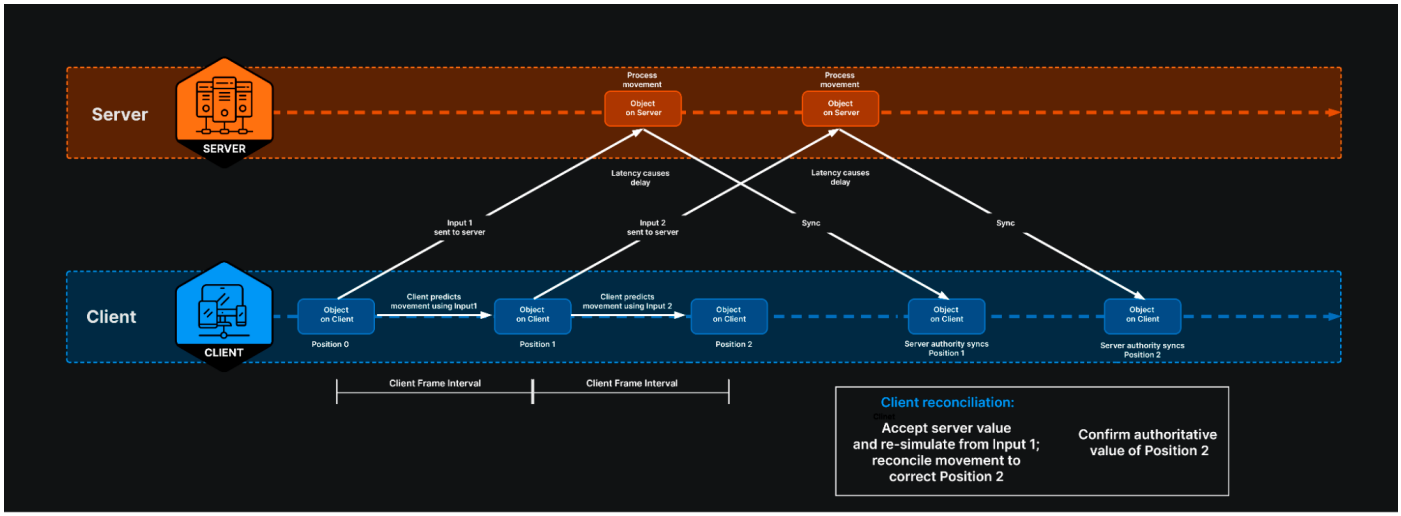
클라이언트는 권한 상태와 예상 상태를 비교하고 차이를 찾습니다. 두 상태가 아주 유사하면 아무 조치도 이루어지지 않으며 클라이언트는 플레이를 지속합니다.

상당한 불일치 또는 '디싱크'(desync)가 있는 경우, 클라이언트는 서버의 권한 상태와 일치하도록 상태를 교정할 방법을 결정해야 합니다. 이 프로세스를 **조정**(reconciliation)이라고 합니다.



클라이언트가 불일치 상태('디싱크')를 수신합니다.

지연을 처리하고 보정하기 위해 클라이언트는 과거의 입력, 그리고 특정 프레임 수에 대한 예측 상태를 저장합니다. 디싱크가 발생하면 클라이언트는 서버에서 마지막으로 확인된 정확한 상태로 상태를 롤백할 수 있습니다. 그런 다음 해당 지점부터 정확한 입력을 사용하여 게임을 재시뮬레이션합니다. 그러면 클라이언트 상태가 서버와 다시 동기화됩니다.



클라이언트가 디싱크를 조정합니다.

특정 프레임에서 클라이언트가 여러 프레임을 재시뮬레이션하는 경우도 있으므로 네트워크 애플리케이션에 더 많은 계산이 필요할 수 있습니다. 하지만 이러한 조정과 롤백은 네트워크 지연이 있는 경우에도 플레이어에게 원활하고 일관된 경험을 제공하는 데 도움이 됩니다.

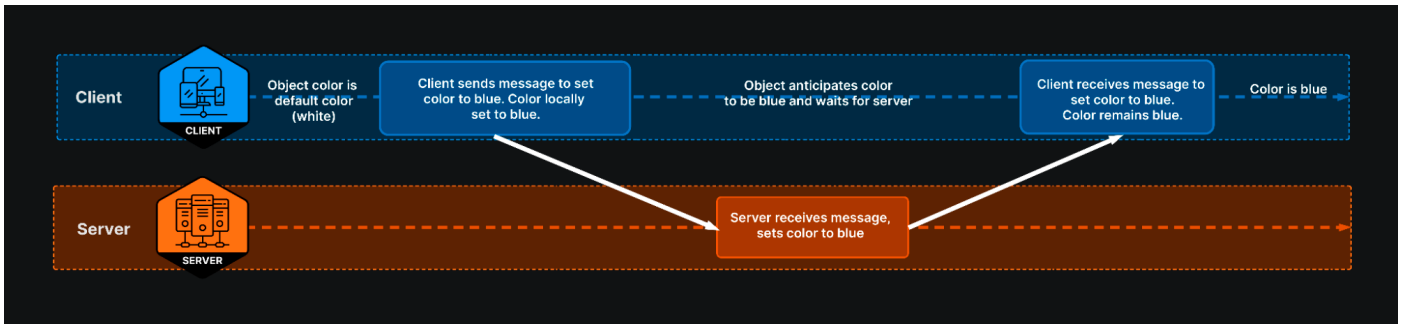
Netcode for GameObjects의 클라이언트측 예상

Netcode for GameObjects는 완전한 클라이언트측 예측 및 조정을 사용하지 않고 지연을 처리하는 간단한 클라이언트 예상 모델을 지원합니다. 여기에 사용되는 컴포넌트는 다음과 같습니다.

- `AnticipatedNetworkVariable<T>`는 정수, 플로트, 색상 등의 간단한 데이터 유형이나 스칼라에 사용되는 일반 컴포넌트입니다. 체력, 점수, 아이템 상태 등의 변환되지 않는 프로퍼티에 적합합니다.
- `AnticipatedNetworkTransform`은 `AnticipatedNetworkVariable`과 유사하게 작동하지만, 위치, 회전, 스케일 등의 트랜스폼 데이터용으로 설계되었습니다.

클라이언트 예상을 통해 클라이언트는 서버의 권한 업데이트를 기다리는 동안 사용자에게 즉각적인 시각적 피드백을 제공할 수 있습니다. 그러면 게임의 반응성을 높일 수 있습니다. 간단한 ColorTrigger 예시에서 플레이어가 오브젝트의 색상을 흰색에서 파란색으로 변경하면 클라이언트는 서버가 해당 변경 사항을 확인하는 동안 색상을 시각적으로 즉시 업데이트할 수 있습니다.

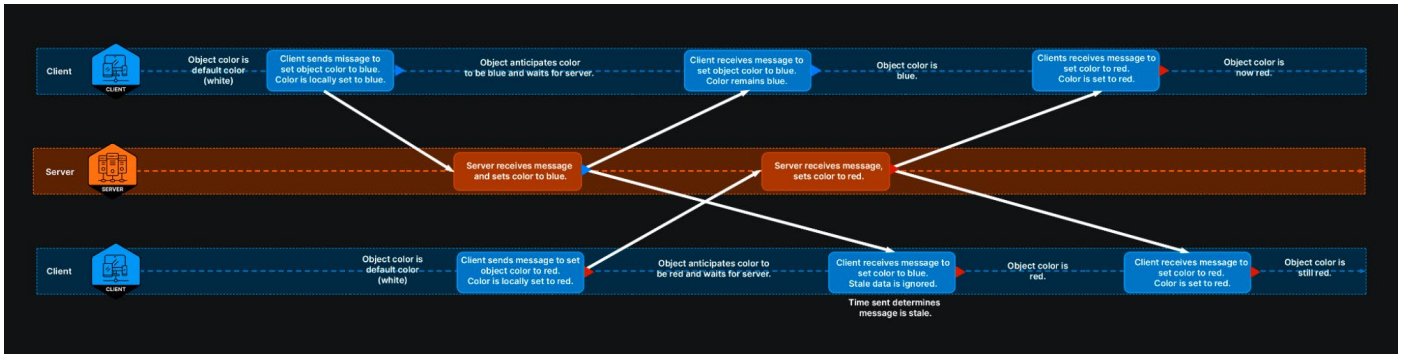
클라이언트와 서버 사이에서 예상이 이루어지는 방식은 다음과 같습니다.



클라이언트가 게임 상태를 예상합니다.

`AnticipatedNetworkVariable<T>`와 `AnticipatedNetworkTransform`은 모두 값을 예상된 시각 상태와 권한 상태로 분리하는 식으로 작동합니다.

예상은 클라이언트의 예측된 상태를 의미하며, 로컬에서 플레이어에게 시각적 피드백을 즉시 제공합니다. 권한 상태는 서버에 의해 결정됩니다. 서버의 업데이트가 도달하면 클라이언트는 예상한 상태를 권한 상태와 비교합니다. 두 상태가 크게 다른 경우 클라이언트는 서버의 상태에 맞게 클라이언트의 상태를 조정하여 모든 클라이언트를 일관되게 만듭니다.



클라이언트는 오래된 데이터를 무시할 수 있습니다.

클라이언트 예상 시, '오래된 데이터'를 고려해야 합니다. 이는 클라이언트의 마지막 요청 이전에 발생한 동작이 반영된 서버 업데이트입니다. Netcode for GameObjects는 `StaleDataHandling` 프로퍼티를 통해 오래된 데이터를 처리하는 두 가지 방법을 제공합니다.

— `StaleDataHandling.Ignore`는 오래된 데이터를 무시하고 예상 값을 유지합니다. 이 방법은 상태가 빠르게 변경되어 시각적 깜빡임이 발생하는 경우에 유용할 수 있습니다.

— `StaleDataHandling.Reanticipate`는 오래된 데이터를 다른 서버 업데이트처럼 취급하여 일관성을 유지할 수 있도록 롤백과 재예상(플레이어 입력을 재실행)을 트리거합니다.

서버 값이 예상 값과 다를 경우 시각적 업데이트가 끊기는 현상을 방지하려면 두 컴포넌트에서 사용할 수 있는 `Smooth` 메서드를 활용합니다.

`Smooth`를 사용하려면 평탄화 프로세스의 시작 값, 최종 값, 지속 시간이 필요합니다. 이 메서드를 사용하면 네트워크 지연이 있는 상황에서도 부드럽고 응답이 빠른 시각적 경험을 제공할 수 있습니다.

클라이언트 예상이 게임플레이의 응답 속도를 개선하기는 하지만, 이렇게 간단한 접근 방식으로 모든 지연 및 네트워크 문제를 해결할 수는 없습니다.

롤백과 조정이 제대로 이루어지려면 동일한 입력이 항상 동일한 결과를 낳는 결정론적 물리 시스템이 필요합니다(아래 참고). 게임 상태를 정확히 롤백하고 재시뮬레이션하려면 꼭 필요한 시스템입니다. 결정론적 분명성이 없으면 서버와 클라이언트 시뮬레이션 사이에 불일치가 발생할 수 있습니다.

Netcode for GameObjects는 진정한 클라이언트 예측과 지연 보상에 필요한 결정론적 물리를 지원하지 않습니다. 그 대신 Netcode for GameObjects는 클라이언트 예상과 평탄화에 중점을 둔 더 간단한 솔루션을 제공합니다. 이 방법은 완전한 롤백 시스템 없이도 응답 속도를 높일 수 있습니다.

결정론적 물리

결정론적 물리 시스템은 동일한 초기 조건과 입력이 주어지면 항상 동일한 물리 시뮬레이션 결과가 나오도록 설계되어 있습니다.

Netcode for GameObjects는 Unity의 빌트인 물리 엔진을 사용하며 이 엔진에는 결정론적 분명성이 없습니다. 같은 입력으로 시뮬레이션을 다시 실행하더라도 물리 시스템에서 동일한 결과가 나올 것이라고 단언할 수 없습니다.

하지만 Netcode for Entities는 Unity Physics와 Havok Physics를 지원하며, 이들은 모두 결정론적 시뮬레이션 기능을 제공합니다. 따라서 Netcode for Entities로 진정한 클라이언트측 예측을 지원할 수 있습니다.

Netcode for Entities의 클라이언트측 예측

Netcode for Entities는 클라이언트측 예측 및 지연 보상을 처리할 수 있는 고급 툴을 제공합니다. 엔티티별로 동일한 시뮬레이션 코드가 클라이언트와 서버에서 모두 실행됩니다. 그러면 클라이언트는 플레이어 입력에 따라 게임 상태를 예측하여 서버의 응답을 기다릴 필요 없이 즉각적인 피드백을 제공할 수 있습니다.

클라이언트가 서버에서 최신 스냅샷을 수신하면, 해당 데이터를 활용해 예측된 엔티티를 모두 업데이트합니다. 이 스냅샷을 적용한 후 클라이언트는 PredictedSimulationSystemGroup이라는 시뮬레이션을 실행합니다. 이 시뮬레이션은 가장 오래전에 저장된 틱부터 현재 목표 시간까지를 처리하며, 클라이언트가 게임 상태를 정확하게 시뮬레이션할 수 있도록 게임 상태를 롤백하고 재시뮬레이션을 실행합니다. 이러한 롤백 및 재시뮬레이션 프로세스를 통해 클라이언트는 불일치를 교정하고 서버의 권한 상태와 동기화를 유지할 수 있습니다.

서버 측에서는 예측 루프가 프레임마다 한 번씩 실행됩니다. 서버는 권한 게임 상태를 업데이트하고 이 업데이트된 상태를 클라이언트로 다시 전송합니다.

클라이언트는 입력 이력과 예측된 상태를 저장합니다. 디싱크가 발생하면 클라이언트는 서버에서 마지막으로 확인된 정확한 상태로 롤백하고 정확한 입력을 사용하여 게임을 재시뮬레이션합니다. 이러한 방법으로 클라이언트는 서버의 '정확한 데이터'와 동기화된 상태를 유지할 수 있습니다.

Netcode for Entities는 다음과 같은 고급 물리 기능도 지원합니다.

- 다중 물리 월드: 네트워크를 통해 복제할 필요가 없는 로컬 전용 물리 시뮬레이션이 가능합니다.
- 커스텀 물리 프록시: 클라이언트에만 있는 물리 오브젝트(예: 파편)와 고스트가 서로 상호 작용하도록 구현하려 할 때, 이 기능으로 상호 작용을 활성화할 수 있습니다.
- 결정론적 물리 시뮬레이션: 입력이 동일한 경우 클라이언트와 서버 시뮬레이션의 결과가 동일해지게 함으로써 클라이언트와 서버 상태 간 일관성을 유지할 수 있습니다.

GhostPredictionSmoothingSystem은 커스텀 평탄화 옵션을 통해 예측 상태와 권한 상태 사이를 전환하여 예측 오류를 평탄화합니다.

예측, 롤백, 지연 보상, 평탄화 기술을 함께 사용하면 지연의 영향을 최소화하고 게임의 무결성과 응답 속도를 유지하는 데 도움이 됩니다.

Netcode for GameObjects와 Netcode for Entities가 클라이언트 예측을 처리하는 방법을 비교해 보세요.

기능	Netcode for GameObjects	Netcode for Entities
예측 방법	클라이언트 예상: 클라이언트가 서버 응답을 예상합니다.	완전한 클라이언트 예측: 클라이언트가 서버와 동일한 시뮬레이션 코드를 실행합니다.
지연 완화	서버 결과 예상 및 전환 평탄화	롤백 및 재시뮬레이션을 사용하여 디싱크 교정
스냅샷	명시적으로 사용되지 않음	스냅샷을 사용하여 게임 상태를 특정 시점의 상태로 나타냅니다.
지연 보상	StaleDataHandling 옵션을 사용하는 간단한 모델	충돌 월드를 반영하는 포괄적인 지연 보상
물리 상호 작용	예상 트랜스폼을 활용한 클라이언트측 예측으로 제한됨	예측된 물리 월드와 클라이언트에만 존재하는 물리 월드 간의 상호 작용 지원
Smoothing	예상 값에 Smooth 메서드 사용	예측 상태와 권한 상태 간의 전환을 평탄화하기 위한 GhostPredictionSmoothingSystem
활용 사례	즉각적인 시각적 피드백이 필요한 간단한 게임에 적합	정확한 상태 동기화가 필요한 복잡한 게임에 적합

Netcode for GameObjects와 Netcode for Entities는 모두 클라이언트 예측을 처리하고 멀티플레이어 네트워킹에서 발생하는 지연 문제를 완화할 수 있는 메커니즘을 제공합니다. Netcode for GameObjects는 클라이언트 예상을 사용하는 간단한 모델을 제공하지만, Netcode for Entities는 완전한 예측, 롤백 및 지연 보상을 비롯하여 더 발전된 시스템을 제공합니다.

Netcode for Entities 용어

Netcode for Entities는 ECS(엔티티 컴포넌트 시스템)와 DOTS(데이터 지향 기술 스택)를 사용한다는 점에서 MonoBehaviour 워크플로를 사용하는 Netcode for GameObjects와는 다릅니다. 낯설게 느껴질 수 있는 용어를 몇 가지 소개하자면 다음과 같습니다.

엔티티: ECS에서 엔티티는 게임 내의 개별 게임 오브젝트 또는 컴포넌트를 의미하는 기본 데이터 단위입니다. 엔티티는 가벼우며, 동작은 없고 데이터만 가집니다.

게임 월드: 게임 월드는 게임이 진행되는 전체 환경을 의미합니다. 여기에는 모든 엔티티, 엔티티의 상태 및 상호 작용을 관장하는 규칙이 포함됩니다.

충돌 월드: 충돌 월드는 게임 월드에 존재하는 모든 물리 오브젝트의 상태(위치, 속도, 상호 작용 포함)로, 충돌 검사와 물리 시뮬레이션에 사용됩니다.

스냅샷: 스냅샷('고스트 스냅샷'이라고도 함)은 특정 시점의 게임 상태를 나타내는 데이터 세트입니다. 클라이언트와 서버는 스냅샷을 통해 주기적으로 게임 상태를 업데이트하여 동기화 상태를 유지합니다.

고스트: 고스트는 클라이언트와 서버에 복제된 네트워크 엔티티입니다. 프레임마다 서버는 모든 고스트의 현재 상태에 대한 스냅샷을 클라이언트에 전송합니다. 고스트는 모든 플레이어에게 게임 월드가 일관되게 보이도록 서버와 클라이언트 간의 엔티티 상태를 동기화하는 데 사용됩니다.

예측 고스트: 예측 고스트는 클라이언트측에서 로컬로 시뮬레이션되어 플레이어 동작에 대한 시각적 피드백을 즉시 제공하여 체감 지연을 줄이는 엔티티입니다. 플레이어가 직접 제어하는 엔티티나 즉각적인 피드백이 필요한 다른 인터랙티브 엔티티에 사용됩니다.

보간된 고스트: 보간된 고스트는 클라이언트에 있는 서버 측 엔티티를 나타냅니다. 클라이언트는 서버에서 수신한 스냅샷을 기반으로 상태를 표시하며, 지연으로 인한 지터를 최소화하기 위해 스냅샷을 블렌딩합니다. 다른 플레이어의 캐릭터, NPC 또는 서버가 제어하는 엔티티처럼 플레이어가 직접 제어하지 않으며 즉각적인 피드백이 필요하지 않은 엔티티에 사용됩니다.

네트워크 게임 테스트 및 디버그

멀티플레이어 게임의 테스트와 디버그는 싱글플레이어 게임과는 다르게 진행됩니다. Netcode 프로젝트의 일반적인 워크플로에 대해 알아두세요.

- **로컬 테스트** 단계에서는 게임의 여러 인스턴스를 실행하여 플레이어 간의 상호 작용을 평가합니다. 플레이어 빌드, Multiplayer Play Mode 패키지, 네트워크 씬 시각화를 사용하여 애플리케이션을 개발합니다.
- 스크립트 또는 NetcodeTransport 컴포넌트를 사용하여 **네트워크 상태를 시뮬레이션**합니다. 실제 네트워크 문제를 모방해 지연, 지터 및 패킷 손실을 재현할 수 있습니다.
- **클라이언트 연결 관리** 단계에서는 클라이언트를 연결하고, 재연결하고, 연결을 해제하는 동작을 처리합니다.
- **로깅** 단계에서는 디버그 툴을 사용하여 문제 해결 상태를 모니터링합니다.
- **커맨드 라인 헬퍼**로 특정 역할과 네트워크 상태가 적용된 게임 인스턴스를 실행하여 테스트를 자동화합니다.

로컬 테스트

멀티플레이어 게임 개발에서는 로컬 테스트를 통해 여러 게임 인스턴스를 시뮬레이션하여 네트워크 환경에서 여러 플레이어가 상호 작용하는 방식을 모방합니다.

플레이어 빌드

플레이어 빌드를 사용하면 게임 실행 파일을 여러 인스턴스로 실행하여 게임을 호스팅하고 게임에 참여할 수 있습니다. 또한 이를 Unity 에디터와 함께 실행할 수 있으므로 기기 하나에서 여러 게임을 동시에 호스팅하고 게임에 참여하여 멀티플레이어 시나리오를 시뮬레이션할 수 있습니다.

MPPM(Multiplayer Play Mode)

Unity 6에 포함된 MPPM(Multiplayer Play Mode) 패키지를 사용하면 개발 기기 하나에서 동일한 소스 에셋을 사용해 최대 4명의 플레이어를 시뮬레이션할 수 있습니다. 이 기능을 사용하면 별도의 플레이어 빌드가 필요 없으므로 테스트 중 빌드에 소요되는 시간이 줄어듭니다.

macOS 사용자

macOS 사용자의 경우 앱을 여러 인스턴스로 실행하려면 커맨드 라인으로 커맨드를 사용해야 합니다. 터미널에서 open 명령을 사용하면 애플리케이션을 별도의 인스턴스로 실행할 수 있습니다.

예를 들어 터미널에서 `open -n (앱 이름).app`을 실행하면 이름을 넣은 애플리케이션을 별도의 인스턴스로 실행할 수 있습니다.

네트워크 상태 시뮬레이션

로컬에서 테스트하면 모든 게임 인스턴스가 동일한 네트워크 인터페이스에서 실행됩니다. 클라이언트 간에는 지연이 거의 없을 것이므로, 실제 환경을 시뮬레이션하려면 인위적인 조건을 생성해야 합니다.

지연, 지터, 패킷 손실은 게임플레이에 영향을 미칠 수 있습니다. 최적이 아닌 네트워크 상태에서 애플리케이션을 테스트하면 최종 빌드가 인터넷에서 잘 작동하는지 확인할 수 있습니다.

어떤 상태를 테스트할지는 타겟 플랫폼, 지역, 게임 디자인 등의 요인에 따라 달라집니다. 처음에는 데스크톱의 경우 약 100~150ms, 모바일의 경우 약 200~300ms의 지연 값과 5~10%의 패킷 손실률을 사용합니다. 지터 및 패킷 손실 테스트는 실제 불안정성을 반영할 방법이므로 필수적입니다. 자세한 가이드라인은 이 기술 자료 페이지를 참조하세요.

에디터 내에서 로컬로 테스트하려면 Multiplayer Play Mode와 함께 Multiplayer Tools의 Network Simulator 툴을 사용하면 됩니다.

개발 빌드를 테스트하는 경우 빌드에 비정상적 네트워크 상태를 주입하는 커스텀 코드와 Network Simulator 툴을 같이 사용하는 것이 좋습니다(Network Simulator 창은 에디터에서만 사용 가능).

릴리스 빌드를 테스트하는 경우 Windows에서는 clumsy를, macOS 또는 iOS에서는 Network Link Conditioner를 사용하는 것이 좋습니다. macOS에서 Network Link Conditioner 대신 사용할 수 있는 스크립트 기반 툴로는 해당 운영 체제에서 기본 제공하며 유용한 제어 기능이 탑재된 dummynet이 있습니다.

자세한 내용은 시스템 차원에서 제공하는 네트워크 컨디셔너를 참조하세요.

클라이언트 연결 테스트

네트워크 게임에서 버그를 방지하고 원활한 게임 경험을 제공하려면 클라이언트 연결 관리를 테스트하는 것이 중요합니다. 테스트할 내용과 주의점은 다음과 같습니다.

연결하는 클라이언트:

— 테스트 사례로는 클라이언트가 새 게임 세션에 참여하거나, 퇴장 후에 재참여 또는 호스팅하거나, 진행 중인 게임에 뒤늦게 참여하거나, 연결 거부를 처리하는 경우 등이 있습니다.

— 클라이언트의 이전 상태가 연결에 영향을 미치는지, 게임 상태가 서버에서 제대로 복제되는지 여부를 고려합니다.

— 서버가 다시 연결하거나 뒤늦게 참여하는 경우를 적절히 처리하는지 확인합니다.

연결을 해제하는 클라이언트:

— 클라이언트의 정상적인 종료, 시간 초과, 호스트/서버와의 연결 손실로 인한 영향 등을 테스트합니다.

— 게임 세션과 연결된 오브젝트가 파괴되지 않으면 적절히 재설정되고 해당 클라이언트가 새로운 게임에 다시 연결할 수 있는지 확인합니다.

세션을 시작하는 호스트/서버:

— 새로운 게임 세션을 시작하는 경우를 테스트합니다. 특히 클라이언트에서 호스팅하는 게임에서는 이전 세션을 종료한 후 테스트합니다.

— 새로운 세션을 시작하기 이전의 애플리케이션 상태가 게임에 영향을 미치는지 여부를 고려합니다.

호스트/서버 종료:

— 특히 Unity Game Services나 Lobby 서비스와 같은 외부 서비스를 사용하는 경우 호스트/서버의 정상적인 종료를 테스트합니다.

— 클라이언트에 종료 사실이 알려지고 외부 서비스에 적절한 알림이 제공되는지 확인합니다.

이러한 테스트 사례를 면밀히 검토하면 안정적이고 즐거운 네트워크 게임 경험을 유지하는 데 도움이 될 수 있습니다.

자세한 내용은 클라이언트 연결 관리 테스트를 참조하세요.

멀티플레이어 게임을 디버그하기 위한 기술

멀티플레이어 게임을 디버그하는 경우, 기존의 모든 게임 개발 원칙이 적용됩니다. 그러나 멀티플레이어 게임 개발 시 흔히 발생하는 특정 상황에는 특별한 요령과 접근 방식이 필요합니다.

다음은 Unity로 멀티플레이어 게임을 개발하는 경우에 도움이 될 수 있는 기술의 목록입니다.

— **디버그 드로잉(drawing) 기술:** Debug.DrawRay와 Debug.DrawLine의 디버그 라인을 사용하여 네트워크 위치, 움직임의 의도, 오브젝트 상호 작용을 나타냅니다. 이 기술은 멀티플레이어 게임플레이를 나란히 놓고 레코딩한 영상과 함께 사용하면 유용합니다.

— **Netcode 지원 라인 렌더러:** 특정 방향, 값, 그 외 프로젝트와 관련된 유용한 디버그 지표를 보여주는 시각적 피드백이 필요한 경우에 유용합니다. Netcode 지원 라인 렌더러를 구현하는 이 스크립트 예시를 참조하세요.

— **텍스트 기반 로깅:** 텍스트 기반 로깅은 비시각적 이벤트(예: RPC) 및 정보를 추적할 수 있습니다. 로그 메시지에 네트워크 틱과 클라이언트 ID를 포함하여 로그를 읽을 때 타임라인을 쉽게 만들 수 있습니다.

— **네트워크 컨디셔닝**: Network Simulator 툴을 사용하면 애플리케이션 수준에서 네트워크 컨디셔닝을 수행할 수 있습니다. 이를 통해 인위적인 네트워크 상태를 시뮬레이션하고 지연, 지터, 패킷 손실과 관련된 오류를 테스트할 수 있습니다. 시스템 차원에서 제공하는 네트워크 컨디셔너에서 자세히 알아보세요.

— **화면 레코딩:** 클라이언트 인스턴스와 서버 인스턴스를 동시에 레코딩하여 실시간 게임플레이를 비교합니다. 디버그 빌드의 경우 각 프레임에 클라이언트 ID와 현재 프레임 번호를 표기합니다. 그러면 나란한 비교를 위한 시각적 참고 자료로써 레코딩 영상을 활용할 수 있습니다.

— **고정 시간 단계 증가:** 디버그 렌더링과 로깅을 잘 활용해도, 프레임을 하나씩 살펴보더라도 상황을 파악하기 어려울 때가 있습니다. **FixedTimeStep** 설정을 큰 값(예: 0.2)으로 증가시키면 각 프레임에서 게임의 동작을 더 명확하게 파악할 수 있습니다.

— **중단점 사용:** 중단점을 사용하여 게임을 디버깅할 때, 이 모드에 너무 오래 머무르면 연결이 시간 초과될 수 있습니다. 게임이 잠시 멈추기 때문에 연결이 끊기지 않도록 시간 초과 값을 일시적으로 늘리는 것이 좋습니다.

자세한 팁과 기술은 멀티플레이어 게임을 디버그하기 위한 기술과 요령 가이드를 참조하세요.

커맨드 라인 헬퍼

Unity 에디터 내에서 멀티플레이어 빌드를 반복적으로 실행하고 테스트하는 과정은 시간이 많이 소요될 수 있습니다. 커맨드 라인 툴을 사용하여 에디터 환경 외부에서 멀티플레이어 게임 빌드를 실행하고 테스트하는 작업을 자동화하는 것이 좋습니다.

이 샘플 NetworkCommandLine 스크립트가 시작에 도움이 될 수 있습니다. NetworkCommandLine 컴포넌트를 게임 오브젝트에 연결하여 빌드에 스크립트를 포함하고 커맨드 라인을 통해 스크립트에 액세스할 수 있습니다.

Player Settings의 **PC, Mac, & Linux Standalone** 설정 아래에서 **Resolution and Presentation**을 선택합니다. **Resolution**을 **Windowed**로 설정합니다. 이렇게 하면 게임의 여러 인스턴스를 나란히 비교하면서 테스트하기가 더 쉽습니다.

NetworkCommandLine 스크립트는 먼저 게임이 에디터 외부에서 실행 중인지를 검사해야 합니다. 에디터 외부에서 실행되는 경우, 스크립트는 커맨드 라인 인수를 읽어 원하는 모드(서버, 호스트 또는 클라이언트)를 결정하고 해당 서비스를 시작해야 합니다. 그러면 커맨드 라인에서 특정 네트워크 역할로 게임 빌드를 실행할 수 있습니다.

File > Build Settings에서 바이너리를 빌드한 후, 커맨드 라인으로 테스트합니다.

Windows:

커맨드 프롬프트를 열고 빌드를 저장한 디렉토리로 이동합니다. 특정 커맨드를 사용하여 게임의 서버 또는 클라이언트 인스턴스를 시작하고, 필요한 경우 경로를 조정합니다. 예를 들면 다음과 같습니다.

<프로젝트 경로>\HelloWorld.exe -mode server
<프로젝트 경로>\HelloWorld.exe -mode client

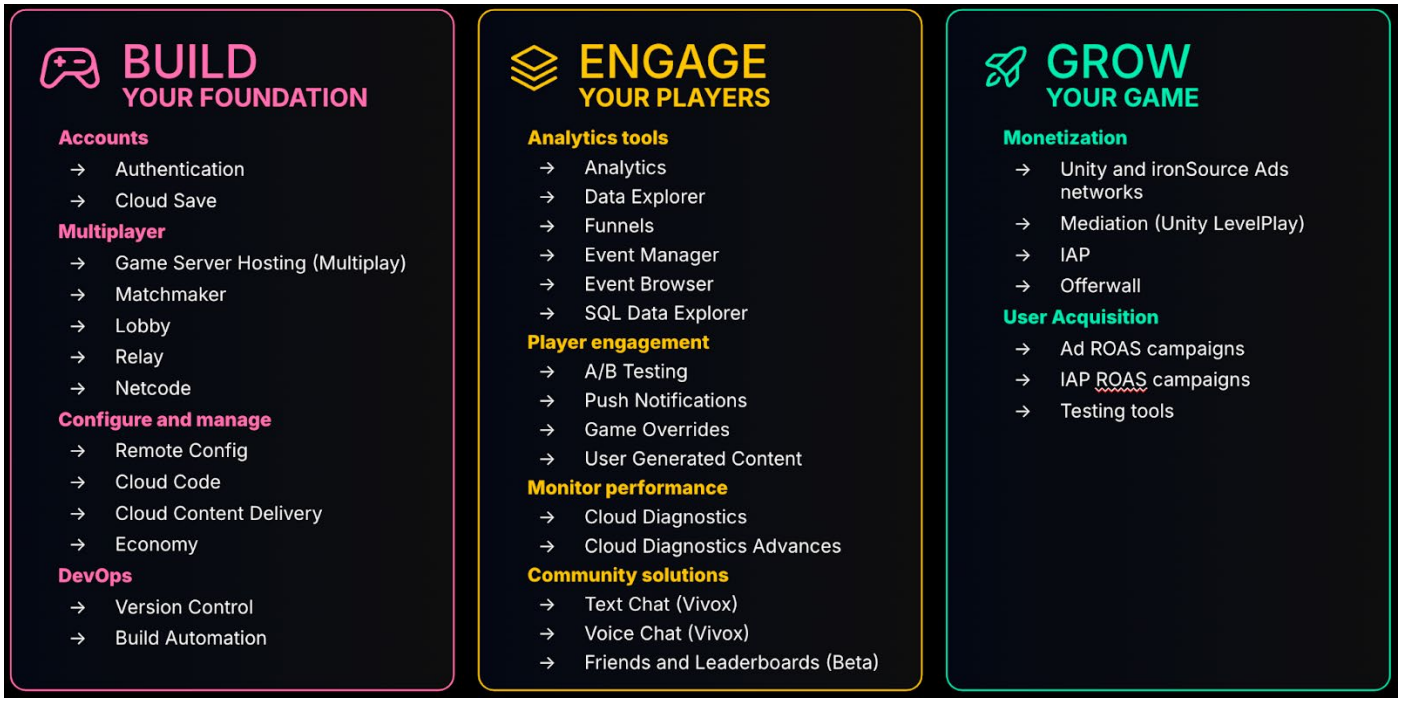
macOS:

macOS에서 터미널을 열고 위에 나온 커맨드와 유사한 것을 사용하되, macOS 디렉토리 구조에 맞게 경로를 조정합니다. 예를 들면 다음과 같습니다.

<프로젝트 경로>/HelloWorld.app/Contents/MacOS/<Project Name> -mode server
<프로젝트 경로>/HelloWorld.app/Contents/MacOS/<Project Name> -mode client

경우에 따라 **-logfile** 플래그를 사용해 게임 인스턴스의 출력을 텍스트 파일로 기록하여 트래킹과 분석을 용이하게 할 수 있습니다.

Multiplayer Services



유니티에서는 네트워크 멀티플레이어 개발을 간소화할 수 있는 다양한 서비스를 제공합니다. 먼저 Multiplayer Services 패키지 기술 자료를 살펴보면서 각 서비스에 대해 알아보세요. 이 패키지로 멀티플레이어 서비스 간 종속성을 손쉽게 관리할 수 있습니다. 활용 방안의 예를 소개하면 다음과 같습니다.

- Unity Gaming Services를 게임에 연동하여 멀티플레이어 요소를 빠르게 추가할 수 있습니다. Lobby, Relay, Distributed Authority, Matchmaker, Multiplay Hosting을 설정할 수 있습니다.
- 새로운 세션 시스템은 멀티플레이어 게임에 맞게 플레이어를 그룹화하고 공유되는 세션/플레이어 상태를 관리하는 간단한 공유 클라우드 기반 백엔드를 제공합니다.
- P2P(피어 투 피어), 전용 게임 서버 및 분산 권한이 호스팅되는 온라인 세션을 생성하고 관리할 수 있습니다. 플레이어는 매치메이킹 또는 참여 코드를 통해서나 활성화된 세션 목록을 찾아보고 세션에 참여할 수 있습니다.

각 서비스에 대해 간단히 살펴보겠습니다.

Matchmaker

매치메이킹은 특정 기준(예: 실력 또는 지리적 위치)에 따라 플레이어를 다른 플레이어나 게임 세션과 연결해, 균형 잡힌 즐거운 게임 경험을 만드는 과정입니다.

Unity의 Matchmaker 서비스를 구현하려면 다음과 같은 여러 단계를 거쳐야 합니다.

- **매치메이킹 기준 정의:** 플레이어 매칭에 사용할 파라미터를 결정합니다. 이러한 파라미터에는 기술 수준, 지리적 인접성, 지연, 플레이어의 선호도 등이 있습니다.
- **플레이어 프로필 및 등급:** 각 플레이어에게 매치메이킹 기준이 포함된 프로필을 제공합니다. 등급 시스템(예: ELO 등급)을 사용하여 플레이어의 실력을 평가하고 균형 잡힌 매칭이 이루어지도록 합니다.
- **매치메이킹 요청:** 플레이어가 매치를 요청하면 매치메이킹 서비스에서는 정의된 기준에 따라 적합한 상대나 팀원을 검색합니다. 그 과정에는 참여 가능한 플레이어 풀을 쿼리하고 가장 적합한 매치를 찾는 작업이 수반됩니다.
- **매치 할당:** 적합한 매치가 발견되면 플레이어는 게임 세션에 할당됩니다. 이 서비스는 모든 플레이어가 연결되어 게임을 시작할 준비가 되었는지 확인합니다.
- **동적 조정:** 특정 기준을 충족할 수 없는 경우 매칭 서비스에서는 품질을 과도하게 저하시키지 않으면서도 신속하게 매칭이 이루어질 수 있도록 파라미터를 동적으로 조정할 수 있습니다.

유니티의 매치메이킹을 사용하면 다음과 같은 요소를 고려해 공정하게 경쟁할 수 있는 매치를 생성할 수 있습니다.

Lobby

로비는 플레이어가 모여 게임 설정을 구성하고 게임 시작을 준비할 수 있는 게임 전 공간입니다. Lobby 서비스는 플레이어들이 다양한 멀티플레이어 게임 시나리오에서 서로를 발견하고 연결할 수 있는 수단을 제공합니다. 일반적인 예시는 다음과 같습니다.

- 활성화된 게임 세션 목록을 살펴보고 하나를 선택하여 참여
- 친구가 내 게임 세션에 연결할 수 있도록 친구와 참여 코드 공유
- 빠른 참여를 사용하여 활성화된 모든 매치를 찾아 즉시 참여
- 공개 또는 비공개 로비를 생성하고 게임 내 친구 목록에 초대 보내기
- 게임 서버에서 로비를 호스팅하여 서버 세션에 대한 액세스 관리 및 제한
- 특정 조건(예: 게임 모드, 맵 유형)에 맞는 Lobby 쿼리

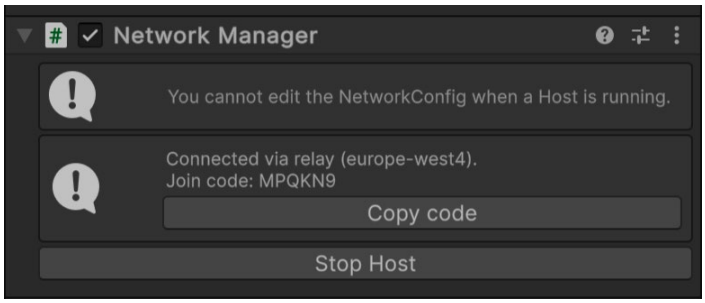
게임 로비 샘플은 Lobby 및 Relay 패키지를 사용하여 Vivox Voice Chat을 포함한 일반적인 게임 Lobby 경험을 만드는 방법을 보여줍니다. 플레이어는 다른 플레이어가 공개 Lobby 목록이나 Lobby 코드를 통해 참여할 수 있는 Lobby를 호스팅할 수 있습니다. 그런 다음, Relay를 통해 연결하면 Unity Transport를 사용한 기본적인 실시간 커뮤니케이션이 가능해집니다.

Relay

Unity Relay는 참여 코드 시스템을 통해 여러 플레이어를 간단하게 연결해 줍니다. 호스트가 게임 세션을 만들고 친구나 팀원들과 공유할 고유의 참여 코드를 생성하면 친구나 팀원들은 이 코드를 사용하여 세션에 연결합니다. 이 시스템은 연결 과정을 간소화하는 동시에 IP 주소와 같은 민감한 정보는 숨김으로써 개인정보도 보호합니다.

P2P 멀티플레이어 게임에서는 NAT(네트워크 주소 변환) 및 방화벽과 같은 네트워크 문제로 인해 클라이언트를 연결하기 어려울 수 있습니다. 직접 연결은 복잡한 네트워크 구성이 필요한 경우가 많으며 플레이어의 IP 주소를 노출해 보안 및 개인정보 보호에 대한 문제를 일으킬 수 있습니다.

Relay는 중개 서버 역할을 하여 클라이언트 간에 안전하고 간단한 연결을 제공함으로써 이러한 문제를 해결합니다. 전용 서버가 필요 없으므로 개발 비용이 절감됩니다. Relay를 사용하는 경우, 참여 코드만 생성하면 플레이어가 바로 연결할 수 있습니다.



연결되면 Relay에서 참여 코드를 제공합니다.

자세한 내용은 Unity Relay 시작 가이드를 참고하세요.

Multiplay Hosting

Multiplay Hosting은 개발 팀이 흥미로운 플레이어 경험을 만드는 데 집중할 수 있도록 복잡한 대규모 인프라 운영 및 관리를 간소화합니다. 다음과 같은 장점이 있습니다.

- 서버 상태 및 기타 분석 데이터를 추적할 수 있습니다.
- 다운타임 없이 패치를 적용하여 서버를 업데이트할 수 있습니다.
- QoS(서비스 품질) 데이터에 따라 최적의 경험을 제공하는 서버에 플레이어를 배정합니다.
- Docker와 Multiplay Hosting 컨테이너 레지스트리를 사용하여 빌드를 컨테이너화할 수 있습니다.

Vivox

Vivox는 게임 내 음성 또는 텍스트 채팅을 통해 플레이어들이 소통할 수 있도록 하여 더 나은 협동형, 경쟁형 멀티플레이어 경험을 제공합니다. Vivox를 사용하면 관리형 호스팅 솔루션을 통해 텍스트 및 음성 채팅을 서비스와 연동할 수 있습니다. 또한 음성 텍스트 변환, 채팅 필터링 등의 접근성 및 규제 관련 기능을 제공합니다.

Vivox는 게임이 Unreal, Unity 또는 커스텀 엔진으로 제작되었는지 여부와 상관없이 다양한 플랫폼에서 플레이어 간 커뮤니케이션을 지원합니다.

게임에 연동하고 프로젝트 설정을 구성하여 Unity Dashboard에서 프로젝트에 커뮤니케이션을 추가하면 됩니다. 2D 및 3D 채널에서 사용자를 무제한으로 연결할 수 있습니다. 사용자는 음성 볼륨을 조절하고, 음소거 기능을 사용하고, 자신의 채널을 관리할 수 있습니다.

메가시티 메트로 데모에 여러 멀티플레이어 서비스를 연동한 방법을 설명하는 GDC 세션을 참조하세요.

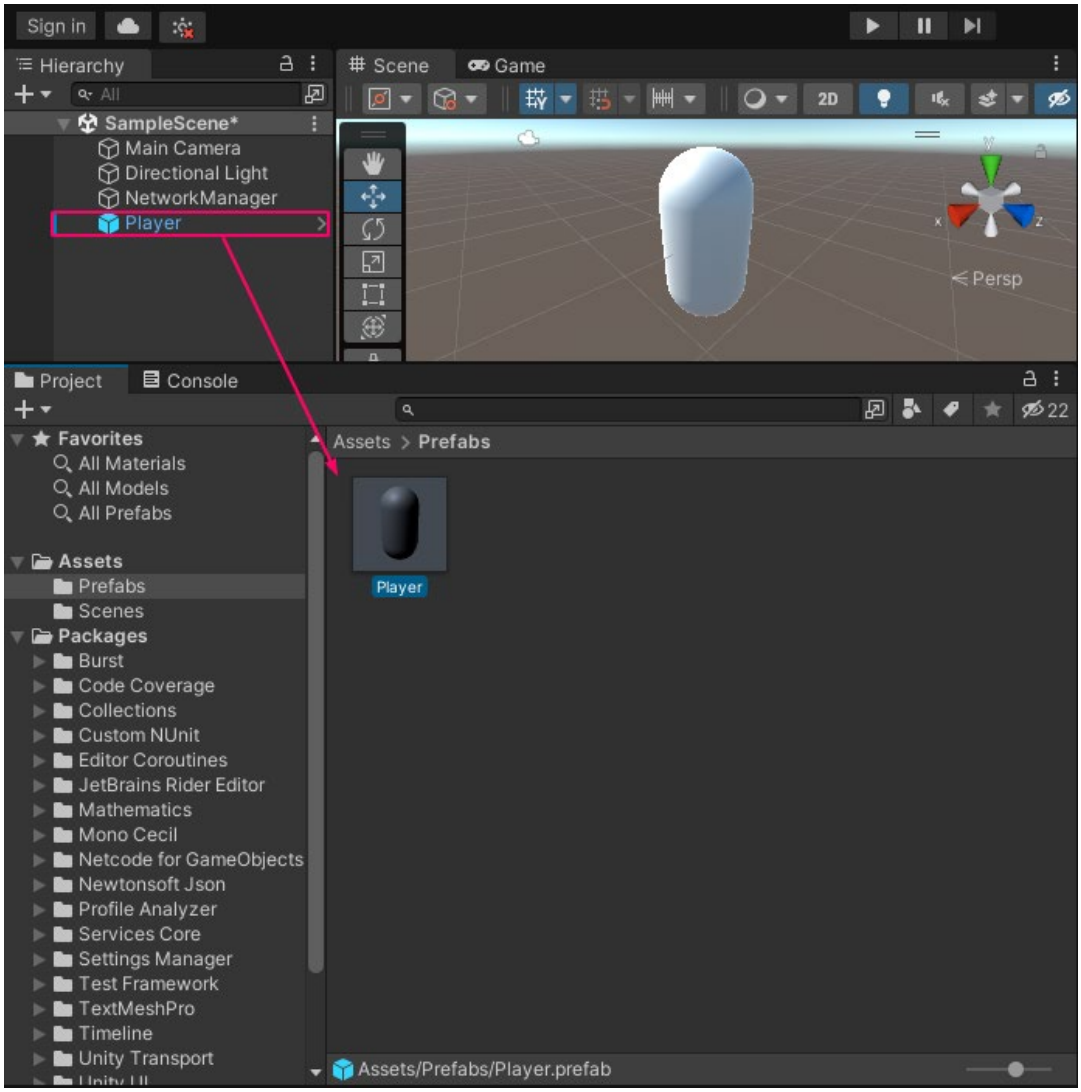
샘플 프로젝트 및 리소스

유니티에서는 Netcode for GameObjects와 Netcode for Entities를 시작하는 데 도움이 되는 다양한 샘플 프로젝트를 제공합니다. 관련 리소스에서는 애플리케이션에 멀티플레이어 네트워킹을 구현하는 데 도움이 되는 실용적인 예시를 살펴볼 수 있습니다.

Netcode for GameObjects 관련 리소스

Unity Learn: Netcode for GameObjects 시작하기

이 Unity Learn 튜토리얼에서는 Unity에서 기본 멀티플레이어 게임을 빌드하고 테스트하는 방법과 RPC (원격 프로시저 호출) 및 네트워크 변수를 활용하고 테스트하는 방법에 대한 실용적인 단계를 소개합니다. 다양한 모드(호스트, 클라이언트, 서버)에 대한 간단한 사용자 인터페이스를 만들고, 해당 모드에 기본적인 움직임을 추가하고 제어하는 방법도 알아볼 수 있습니다.

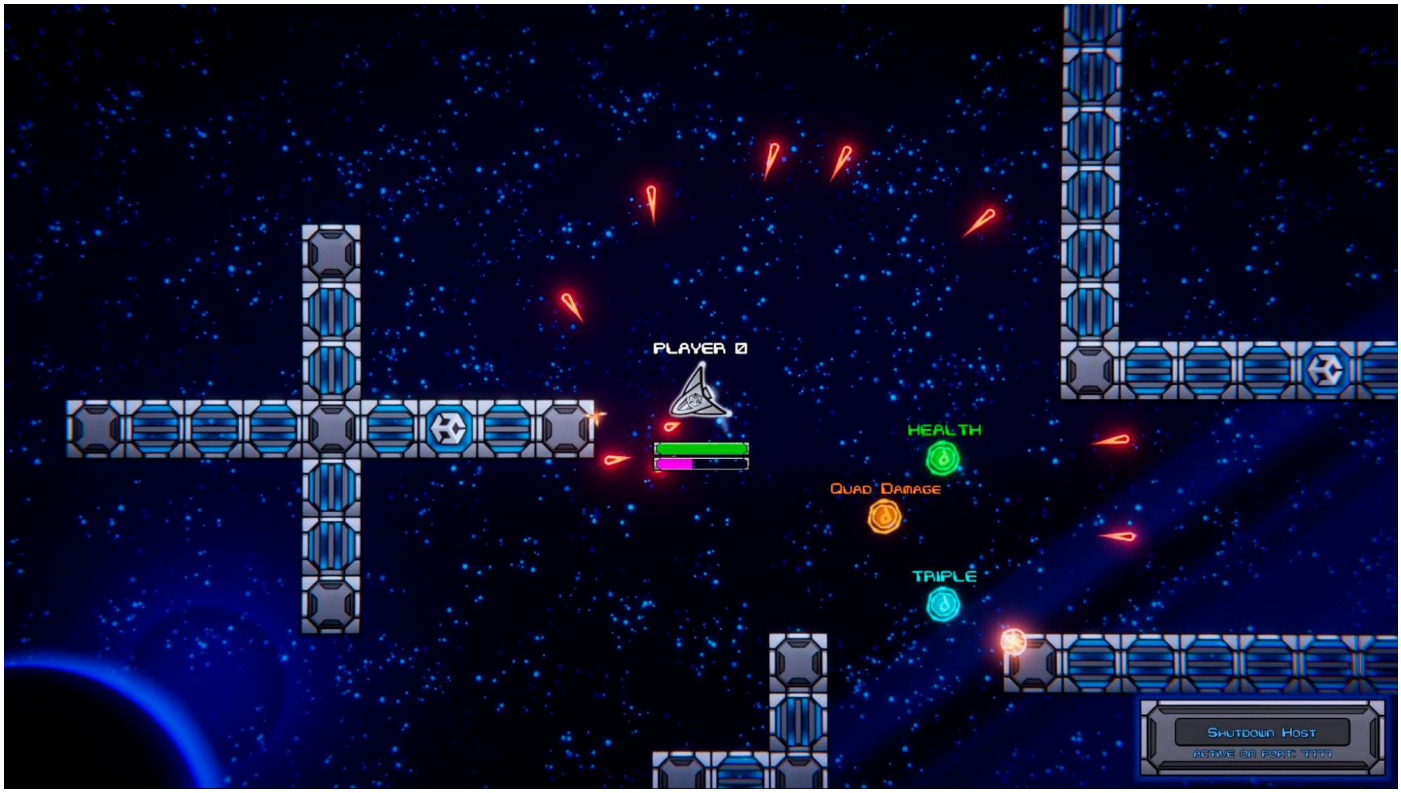


Netcode GameObjects를 다루는 Unity Learn 튜토리얼 스크린샷

Bitesize Sample

Bitesize Sample 저장소에서는 Netcode for GameObjects의 이해도를 높이는 데 도움이 되며 게임에서 사용 가능한 모듈 형태의 샘플 코드를 제공합니다. 저장소에 포함된 내용은 다음과 같습니다.

— 2D Space Shooter 샘플: Netcode NetworkVariable과 오브젝트 풀링을 사용한 물리 움직임과 상태 효과에 대해 자세히 알아보세요.



Bitesize Samples의 2D Space Shooter

— Distributed Authority Social Hub: 이 토폴로지를 설정하여 게임 상태의 제어 및 관리 역할을 여러 클라이언트에 분산하여 서버 부하를 줄이는 방법을 알아보세요.

— Multiplayer Use Cases 개요: 이 샘플은 멀티플레이어 환경에서 일반적인 동작을 수행하는 방법을 보여줍니다. 게임의 기능을 빌드할 때 이러한 동작을 고려하여 설계할 수 있습니다.

— Client Driven 샘플: 클라이언트 기반 움직임, 네트워크 물리, 게임 중 생성되는 오브젝트와 정적으로 배치된 오브젝트의 차이, 오브젝트 부모 재지정 등에 대해 자세히 알아보세요.

— Dynamic Addressables Network Prefabs: 런타임 시 새롭게 생성 가능한 프리팹을 추가할 수 있는 동적 프리팹 시스템에 대해 자세히 알아보세요.

보스 룸

보스 룸은 Netcode for GameObjects로 제작된 3D 캐주얼 협동 게임 샘플 프로젝트로, 멀티플레이어 게임 플로의 바탕이 되는 컨셉과 패턴을 살펴볼 수 있도록 설계되었습니다.

보스 룸은 완전한 기능을 갖춘 협동 멀티플레이어 게임의 축소판입니다.

온전히 작동하는 게임 샘플이자, 네트워크 멀티플레이어 게임에 관심이 있는 개발자를 위한 학습 툴이기도 한 보스 룸은 Netcode for GameObjects 워크플로로 제작된, 유니티에서 가장 오래 제공하고 있는 정식 버전의 멀티플레이어 샘플입니다. 이 샘플에는 정식 프로젝트 수준의 코드가 포함되어 있고, Lobby 및 Relay 호스팅 서비스와 연동되어 있습니다.

보스 룸은 캐릭터 능력 및 특수 공격, 네트워크 물리, 상태 트래킹(예: 파괴 가능한 오브젝트, 스위치, 문)을 비롯한 네트워크 플레이의 게임 메카닉을 보여줍니다. 이러한 기술은 지연을 감추고, 최적이 아닌 네트워크 상태에서도 원활한 게임플레이를 구현하는 방법을 보여줍니다. 자세히 살펴보겠습니다.

게임 플로 및 상태 관리: 보스 룸에는 게임 상태의 실용적인 구현이 포함되어 로비와 인게임 플레이를 모두 지원합니다. 플레이어 연결을 처리하고, 네트워크 플레이를 위해 씬을 로드 및 언로드하고, 정상적인 연결 해제를 관리하는 방법을 확인할 수 있습니다.

UGS 연동: 보스 룸은 Relay, Lobby, Authentication 등의 여러 UGS 기능과 연동되어 있습니다.

유틸리티 스크립트 및 툴: 이 저장소에는 클라이언트 권한, 씬 관리 유틸리티, 네트워크 오브젝트 풀링 등의 예시를 비롯하여 다른 프로젝트에 적용할 수 있는 다양한 유틸리티 스크립트 및 툴도 포함되어 있습니다.

클라이언트-서버 모델: 보스 룸은 플레이어 중 한 명은 호스트(서버) 역할, 다른 플레이어는 클라이언트 역할을 하는 클라이언트-서버 모델을 사용합니다. 게임 로직을 중앙 집중화하면 부정행위의 가능성을 줄이고 모든 클라이언트에서 게임 상태가 일치하도록 유지할 수 있습니다.

보스 룸 샘플 프로젝트의 시작 메뉴

프로젝트에 대해 자세히 알아보려면 이 블로그 게시물과 4부로 구성된 YouTube 웨비나, 'Netcode for GameObjects로 정식 멀티플레이어 게임 만들기(영문)'을 확인해 보세요.

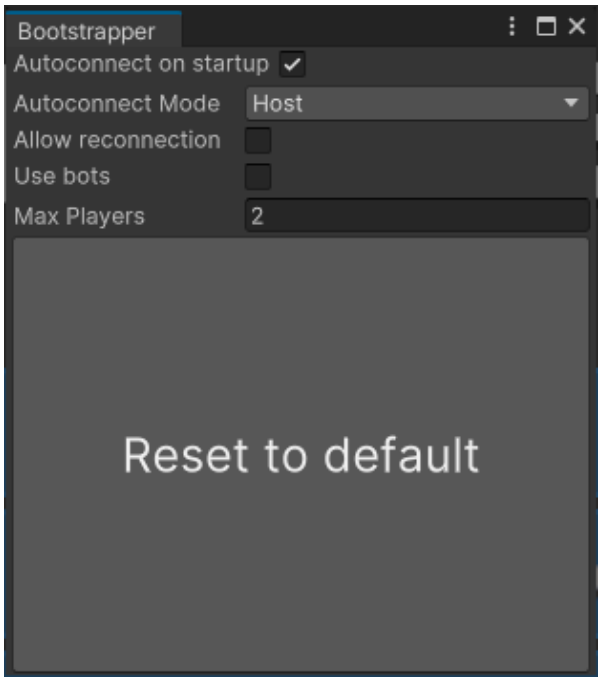
이 시리즈에서는 게임 메카닉과 서버 권한을 비롯해 네트워크 게임플레이 및 오브젝트 생성 구현 방법을 다룹니다. 플레이어 동작에 대한 게임의 안정성을 강화하는 방법과 대역폭을 더 효과적으로 관리할 수 있도록 최적화하는 방법을 소개합니다. 완전한 기능을 갖춘 멀티플레이어 프로젝트를 활용하는 실용적인 실습 가이드가 필요한 개발자를 위한 리소스입니다.

소규모 경쟁형 멀티플레이어 템플릿

Unity Hub에서 사용할 수 있는 이 템플릿은 Netcode For GameObjects와 UGS를 사용해 멀티플레이어 프로젝트를 개발하여 출시하기 위한 출발점이 됩니다.

이 템플릿에는 다양한 네트워크 모드(호스트, 클라이언트, 서버)와 동적 설정을 사용하여 테스트하는 데 도움이 되는 Bootstrapper 툴이 포함되어 있습니다.

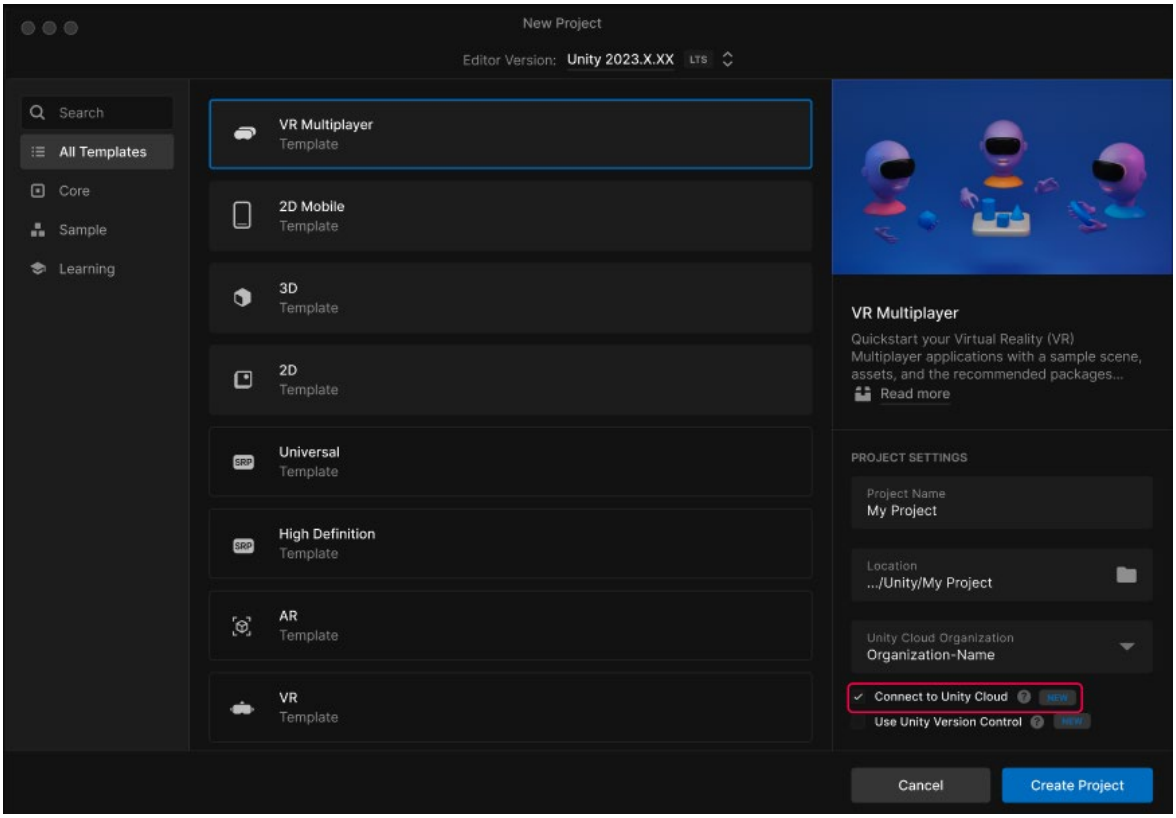
프로젝트를 열고 포함된 시작 다이얼로그의 지침을 따라 시작할 수 있습니다. 에디터 내의 튜토리얼을 따르거나 **Multiplayer > Bootstrapper** 메뉴 항목에서 직접 시작 씬을 로드합니다. 테스트할 내용에 따라 Bootstrapper의 각 필드 값을 조정하고 플레이 모드에 진입합니다.



Bootstrapper 옵션을 설정합니다.

Tutorials > Show tutorials 메뉴에서 언제든지 튜토리얼과 유용한 리소스에 액세스할 수 있습니다.

VR Multiplayer 템플릿



Unity Hub에서 VR Multiplayer 템플릿을 엽니다.

VR Multiplayer 템플릿은 OpenXR 기기를 타겟으로 새로운 프로젝트를 시작하는 크리에이터를 위해 디자인되었으며, 네트워크 상호 작용, 음성 채팅, 로비 등에 필요한 모든 것을 제공합니다. 이 템플릿은 Unity Gaming Services, Netcode For GameObjects, XR Interaction Toolkit을 활용하며, Meta Quest 플랫폼과 기타 OpenXR 호환 기기에서 원활하게 작동하도록 개발되었습니다. VR Multiplayer 템플릿의 주요 기능은 다음과 같습니다.

- XRI Interaction Toolkit 및 Netcode for GameObjects를 통한 네트워크 상호 작용
- Vivox를 통한 음성 커뮤니케이션
- Relay 및 Lobby를 통해 어디서든 모든 사용자와 연결
- 네트워크 연동 아바타 및 손 동작
- Open XR을 통한 크로스 플랫폼 작업

Unity Hub에서 Unity 2022 LTS 및 Unity 6 프리뷰용 Unity VR Multiplayer 템플릿을 다운로드하고 빠른 시작 가이드를 확인하세요.

Netcode for Entities 관련 리소스

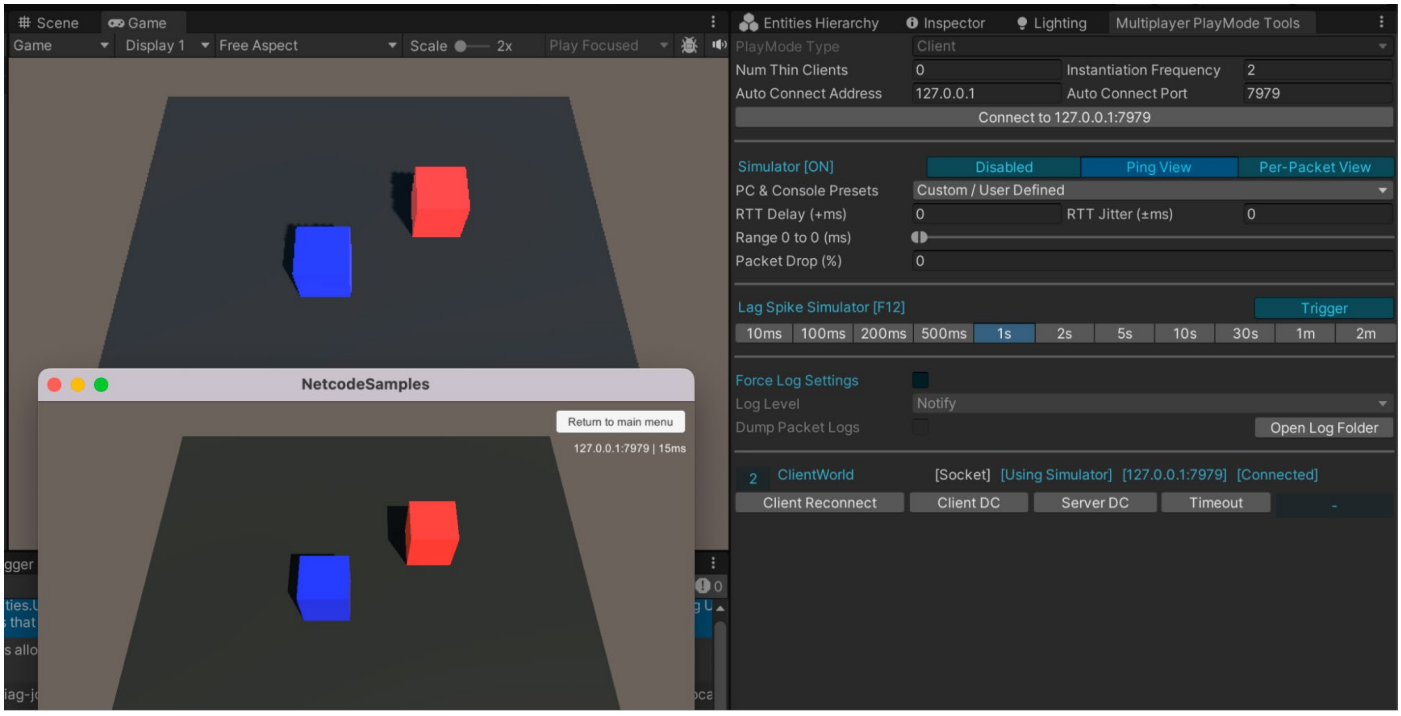
Netcode for Entities 시작하기

이 온디맨드 웨비나에서는 Netcode for Entities 및 Unity Gaming Services로 개발된 Unity의 대규모 크로스 플랫폼 멀티플레이어 게임 샘플인 메가시티 메트로를 자세히 살펴봅니다. 이 웨비나는 DOTS에 이미 익숙하고 성공적인 멀티플레이어 게임을 개발하려는 사용자를 대상으로 합니다.

ECS Netcode 샘플

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 네트워크 레이싱

이 멀티플레이어 레이싱 게임 샘플은 Unity Netcode for Entities를 사용하는 베스트 프랙티스를 보여 줍니다. 이 데모에는 클라이언트측 예측, 보간 및 지연 보상을 비롯한 클라이언트-서버 아키텍처가 구현되어 있습니다.

ECS 레이싱은 고급 멀티플레이어 기능을 선보입니다.

메가시티 메트로

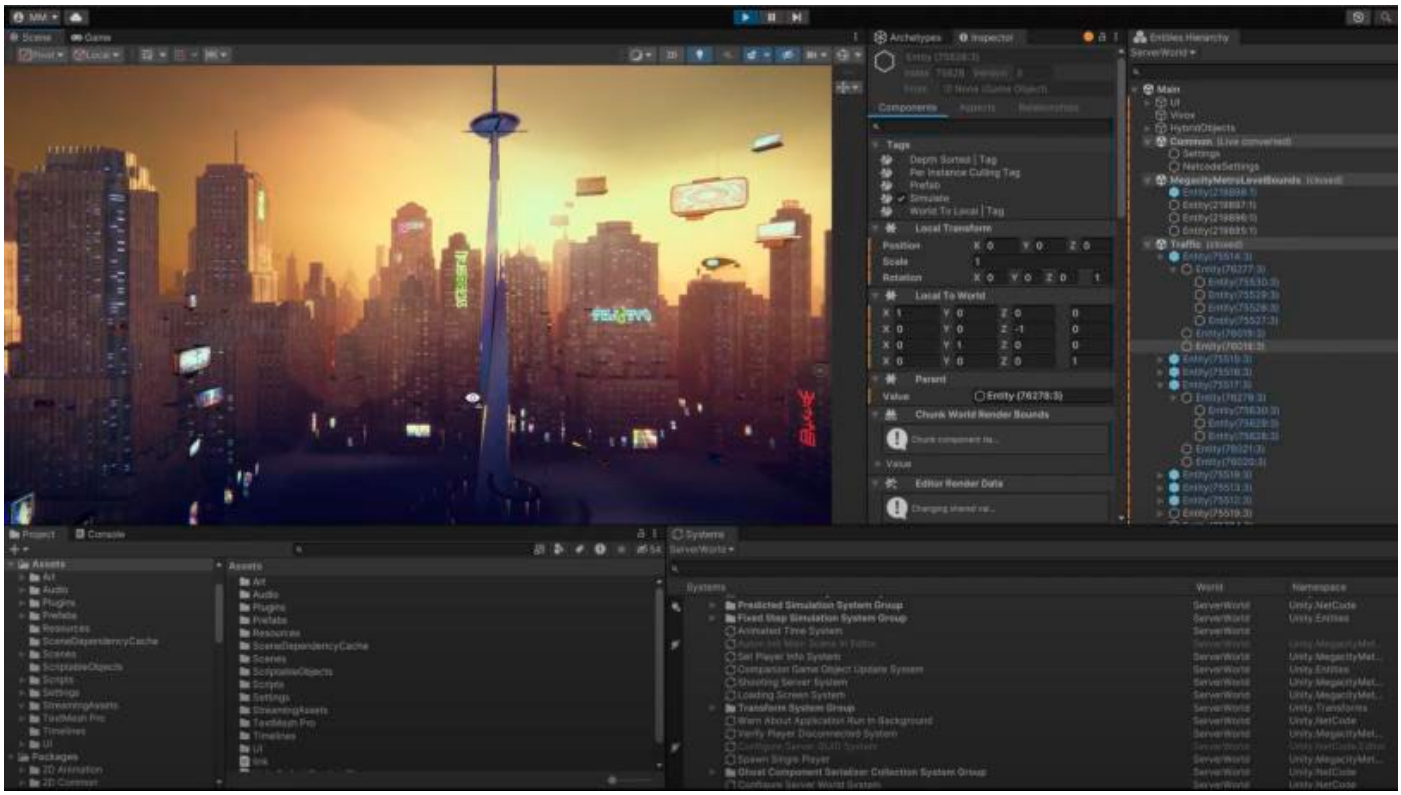
메가시티 메트로(Unity 6, Unity 2022 LTS)는 Netcode for Entities 패키지를 포함한 최신 기술이 적용되어 확장성과 동시성이 우수한 크로스 플랫폼 데모입니다. 이 3인칭 멀티플레이어 액션 데모에서는 128명 이상의 플레이어를 지원합니다.

이 멀티플레이어 샘플은 서버 권한 게임플레이 구현을 숙달하고 엔드투엔드 멀티플레이어 게임을 만들기 위해 UGS를 활용하려는 개발자에게 적합합니다.

Unity의 메가시티 메트로 데모는 128명 이상의 플레이어를 지원합니다.

Unity ECS와 멀티플레이어 솔루션을 사용하여 게임을 성공적으로 개발하는 방법에 대해 자세히 알아보세요. Unity ECS는 더 성공적인 게임을 구현하기 위해 더 많은 통제권과 결정론적 분명성을 요구하는 숙련된 Unity 크리에이터에게 적합합니다.

메가시티 메트로는 보간, 클라이언트측 예측, 지연 보상 등의 고급 메카닉을 특징으로 합니다. 데모를 다운로드하여 Multiplay Hosting, Matchmaker, Vivox Voice Chat 등의 서비스를 체험해 보세요.



메가시티 메트로는 Netcode for Entities를 사용하여 멀티플레이어 액션 데모를 구현했습니다.

Multiplayer Services 패키지(실험 단계)

멀티플레이어 게임을 개발하려면 여러 제품과 서비스를 연동해야 합니다. 새로운 **Multiplayer Services** 패키지(**com.unity.services.multiplayer**)를 사용하면 멀티플레이어 서비스의 연동과 종속성 관리를 간소화하고 새로운 방식으로 제품과 상호 작용할 수 있습니다.

Multiplayer Services 패키지는 게임에 다양한 온라인 멀티플레이어 요소를 추가할 수 있는 원스톱 솔루션입니다. UGS를 기반으로 하는 이 패키지는 Relay, Lobby와 같은 서비스의 기능을 하나의 새로운 '세션' 시스템으로 통합하여 개발자가 플레이어 그룹이 서로 연결되는 방식을 빠르게 정의할 수 있도록 지원합니다.

Multiplayer Services 패키지를 사용하면 P2P(피어 투 피어) 세션을 생성하고 플레이어가 세션에 참여할 수 있는 수단(참여 코드, 활성화된 세션 목록 검색, '빠른 참여' 등)을 제공할 수 있습니다.

다음 단계

지금까지 네트워킹 개념에 관한 기본 사항을 구체적으로 학습하고 Netcode for GameObjects를 실습해 봤습니다. 이어서 멀티플레이어 부문을 더 깊게 학습할 방법을 소개합니다.

Unity Learn에서 자세히 알아보기: 새로운 안내식 콘텐츠를 도입하여 멀티플레이어를 훨씬 더 쉽게 학습할 수 있도록 노력하고 있습니다. Unity Learn의 첫 번째 멀티플레이어 교육 과정에서는 Netcode for GameObjects의 기초를 배우고, 첫 네트워크 프로젝트를 진행하는 방법을 안내합니다.

샘플 프로젝트 살펴보기: 보스 룸, Galactic Kittens, Bitesize Samples와 같은 샘플 프로젝트에 대해 자세히 알아보세요. 네트워크 멀티플레이어를 작업하면서 이러한 프로젝트를 모딩해 보면 새로운 기술을 접하고 베스트 프랙티스를 찾을 수 있습니다.

멀티플레이어 서비스를 통한 실험: 이미 잘 만들어진 기능을 다시 만들 필요는 없습니다. Relay, Lobby, Matchmaker가 프로젝트에 바로 적용될 수 있도록 준비되어 있습니다. 이러한 서비스로 개발 시간을 대폭 줄일 수 있습니다.

고급 기술 숙달: 프로젝트의 요구 사항이 증가하면 Netcode for Entities를 사용하는 것이 좋습니다. 이 고급 패키지에는 최고의 네트워킹 성능을 발휘하기 위한 클라이언트측 예측, 지연 보상 등의 기술이 포함되어 있습니다. 메가시티 메트로와 ECS 레이싱 샘플은 이러한 기능이 적용된 모습을 보여줍니다.

여러분의 네트워킹 개발이 즐겁기를 바랍니다.

unity.com/kr