Customizing Character Movement Component

Created	@March 4, 2022 4:34 PM
Tags	

- 출처 : https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Networking/CharacterMovementComponent/
- 위 출처 문서에서 잘못된 부분을 고쳤습니다.
- 본 문서는 UE 4.27 기준입니다.
- 아래에서부터는 캐릭터 무브먼트 컴포넌트를 “CMC”라고 칭하겠습니다.

What We Should Be Interested In

- 클라이언트에서 먼저 캐릭터의 움직임을 시뮬레이션하고 서버에서는 그 결과를 받아 마찬가지로 캐릭터의 움직임을 시뮬레이션합니다.
- 그리고 그 결과를 토대로 클라이언트에게 보정하라 마라 통보하게 되죠.
- 새로운 움직임을 정의했을 때 우리가 관심있게 보아야 할 것은 다음과 같습니다.
  1. 클라이언트와 서버에서 어떻게 새로운 움직임을 시뮬레이션할 것인지
  2. 클라이언트와 서버가 어떤 형식으로 데이터를 주고 받고 어떻게 새로운 움직임에 대한 정보를 넣을 것인지
- 먼저 2번에 대해 자세히 알아보고 1번을 알아보겠습니다.

What Data Client And Server Exchange

- CMC는 클라이언트에서 네트워크를 통해 구조체 `FSavedMove_Character`를 서버로 전송합니다.
- 시스템 내부적으로 살펴보면 CMC는 `CallServerMovePacked(const FSavedMove_Character* NewMove, const FSavedMove_Character* PendingMove, const FSavedMove_Character* OldMove)` 함수에서 여러개(NewMove, PendingMove, OldMove)의 `FSavedMove_Character`들을 `FCharacterNetworkMoveDataContainer`에 넣고 `FCharacterServerMovePackedBits`로 직렬화하여 서버에 전송합니다.

```
// UCharacterMovementComponent.cpp
// 설명에 필요없는 코드들을 모두 생략했습니다.

void UCharacterMovementComponent::CallServerMovePacked(const FSavedMove_Character* NewMove, const FSavedMove_Character* PendingMove, const
{
  // Get storage container we'll be using and fill it with movement data
  // MoveDataContainer는 UCharacterMovementComponent의 멤버입니다.
  FCharacterNetworkMoveDataContainer& MoveDataContainer = GetNetworkMoveDataContainer();

  // FCharacterNetworkMoveDataContainer는 데이터를 채우면서 FCharacterNetworkMoveData 구조체를 사용합니다.
  // 위에 언급되진 않았지만 CharacterMovementComponent를 확장하기 위해 FCharacterNetworkMoveData도 확장해야 한다는 것입니다.
  MoveDataContainer.ClientFillNetworkMoveData(NewMove, PendingMove, OldMove);

  // ........

  // Serialize move struct into a bit stream
  if (!MoveDataContainer.Serialize(*this, ServerMoveBitWriter, ServerMoveBitWriter.PackageMap) || ServerMoveBitWriter.IsError())
  {
    UE_LOG(LogNetPlayerMovement, Error, TEXT("CallServerMovePacked: Failed to serialize out movement data!"));
    return;
  }

  // Copy bits to our struct that we can NetSerialize to the server.
  // 'static' to avoid reallocation each invocation
  static FCharacterServerMovePackedBits PackedBits;
  PackedBits.DataBits.SetNumUninitialized(ServerMoveBitWriter.GetNumBits());

  check(PackedBits.DataBits.Num() >= ServerMoveBitWriter.GetNumBits());
```

```
  FMemory::Memcpy(PackedBits.DataBits.GetData(), ServerMoveBitWriter.GetData(), ServerMoveBitWriter.GetNumBytes());

  // Send bits to server!
  ServerMovePacked_ClientSend(PackedBits);

  // ........
}
```

- 서버에서는 `FClientAdjustment` 에 보정데이터를 담아 클라이언트에 전송합니다.
- 위와 과정과 비슷하게 `UCharacterMovementComponent::ServerSendMoveResponse(const FClientAdjustment& PendingAdjustment)` 를 호출, 직렬화하여 클라이언트에 전송합니다.
- 그러면 클라이언트에서는 어떻게 데이터를 역직렬화하여 어디에 보관할까요? UCharacterMoveComponent에는 `FCharacterMoveResponseDataContainer` 자료형 멤버변수를 가지고 있는데 서버로 부터 받은 데이터를 역직렬화하여 여기에 보관합니다.
- 그렇다면 우리가 새로 만들 움직임을 만들때, 클라이언트와 서버가 같은 데이터로 움직임을 계산하도록 보장하기 위해 `FSavedMove_Character`, `FCharacterNetworkMoveData`, `FCharacterNetworkMoveDataContainer`, `FClientAdjustment` , `FCharacterMoveResponseDataContainer` 를 확장해야 할 것입니다.
- 그리고 `UCharacterMovementComponent` 도 당연히 확장해야 할 것이고 여러 계산함수도 확장해야 한다는 것입니다.
- 또한 필요에 따라 `FNetworkPredictionData_Client_Character` 를 확장해야 합니다. 우리는 위 확장해야 하는 목록 중에 `FSavedMove_Character`, `FNetworkPredictionData_Client_Character` , `UCharacterMovementComponent` 이 세개의 구조체 및 클래스를 확장하여 사용할 것입니다.

Extending UCharacterMovementComponent

- 우리가 새로 만들 움직임을 Sprint, Teleport로 하겠습니다.
- Sprint는 Sprint 키를 입력하고 캐릭터가 땅에서 움직이는 경우 1000의 Speed로 움직이게 됩니다.
- Teleport는 Teleport 키를 입력하면 플레이어가 바라보는 방향으로 5미터 앞으로 순간이동하게 됩니다.
- 위와 같은 동작을 구체화하고 구현하기 위해 다음과 같이 UCharacterMovementComponent를 확장합니다.

```
// MT는 MultiplayTest의 줄임말입니다.

UCLASS()
class MULTIPLAYTEST_API UMTCharacterMovementComponent : public UCharacterMovementComponent
{
  GENERATED_BODY()
#pragma region Members
  // 캐릭터가 Sprint할 때의 최대 Speed
  UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="Character Movement: Walking", meta=(AllowPrivateAccess="true"))
  float MaxWalkSprintSpeed;

  // 캐릭터가 순간이동할 때의 거리
  UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="Character Movement: Teleport", meta=(AllowPrivateAccess="true"))
  float TeleportDistance;

  // 플레이어가 Sprint 키를 누르고 있는지
  bool bPressingSprint;

  // 플레이어가 텔레포트 키를 눌렀는지
  bool bPressedTeleport;
#pragma endregion

public:
  UMTCharacterMovementComponent();

#pragma region Overrides
protected:
  // UCharacterMovementComponent는 FNetworkPredictionData_Client_Character를 가지고 있고 FSavedMove_Character를 사용
  // 여러 이동정보를 저장하게 됩니다. FNetworkPredictionData_Client_Character는 FNetworkPredictionData_Client를 상속하고
  // 사용할 때는 FNetworkPredictionData_Client으로 사용해서 다형적으로 동작하게 됩니다.
  // GetPredictionData_Client()에서는 호출될 때 게으르게 FNetworkPredictionData_Client_Character를 생성해서 반환합니다.
  // 우리가 상속내린 FNetworkPredictionData_Client_Character를 사용하려면 이 함수를 오버라이드해야 합니다.
  virtual FNetworkPredictionData_Client* GetPredictionData_Client() const override;

  // 캐릭터의 회전을 자신만의 방식대로 하고 싶으면 밑의 함수를 오버라이드하면 됩니다.
  virtual void PhysicsRotation(float DeltaTime) override;
```

```
  // 클라이언트에서나 서버에서나 캐릭터의 움직임은 PerformMovement() 를 호출하여 계산합니다.
  // 캐릭터의 움직임을 수정하거나 추가하고 싶다면 이 함수를 눈여겨 봐 주십시오.
  virtual void PerformMovement(float DeltaTime) override;

  // 움직임을 계산할 때 현재 캐릭터의 상태에 따라 최대 Speed를 반환하는 함수입니다.
  // 캐릭터가 걷고 있는 중인데 bPressingSprint가 true이면 MaxWalkSprintSpeed를 반환합니다.
  virtual float GetMaxSpeed() const override;

  // 클라이언트와 서버가 데이터를 주고 받을 때 데이터를 직렬화하면서 플레이어의 입력을 압축,
  // uint8 CompressedFlag라는 데이터를 추가하여 전송합니다.
  // 즉, 클라이언트에서나 서버에서나 CompressedFlag를 포함한 이동정보를 받게 될 것인데
  // 이때 CompressedFlag의 내용을 bitmasking을 통해 플레이어의 입력을 재현하는 것입니다.
  // 아래 내용은 비트마스킹을 하기 위해 각 비트가 무엇으로 정의되어 있는지를 나타냅니다.

  // enum CompressedFlags
  // {
  //  FLAG_JumpPressed  = 0x01, // Jump pressed
  //  FLAG_WantsToCrouch  = 0x02, // Wants to crouch
  //  FLAG_Reserved_1   = 0x04, // Reserved for future use
  //  FLAG_Reserved_2   = 0x08, // Reserved for future use
  //  // Remaining bit masks are available for custom flags.
  //  FLAG_Custom_0   = 0x10,
  //  FLAG_Custom_1   = 0x20,
  //  FLAG_Custom_2   = 0x40,
  //  FLAG_Custom_3   = 0x80,
  // };

  // 예를 들면 0번 비트가 켜져 있으면 점프 키를 눌렀다고 생각할 수 있기 때문에
  // 아래 함수에서는 bPressedJump를 true로 셋합니다.
  virtual void UpdateFromCompressedFlags(uint8 Flags) override;
#pragma endregion

#pragma region GettersAndSetters
public:
  FORCEINLINE void SetPressingSprint(bool bInPressingSprint) { bPressingSprint = bInPressingSprint; }
  FORCEINLINE void SetPressedTeleport(bool bInPressedTeleport) { bPressedTeleport = bInPressedTeleport; }
  FORCEINLINE bool IsPressingSprint() const { return bPressingSprint; }
  FORCEINLINE bool IsPressedTeleport() const { return bPressedTeleport; }
#pragma endregion
};
```

Extending Saved Move Data

1. 새로운 데이터를 넣기 위해 먼저 `FSavedMove_Character` 클래스를 확장해야 합니다.
2. `FSavedMove_Character`를 확장하게 되면 변수를 비롯, 여러 유틸 함수를 오버라이드해야 합니다.

```
// MT는 MultiplayTest의 줄임말입니다.
// FSavedMove_Character를 상속하는 FSavedMove_MTCharacter를 선언합니다.

class MULTIPLAYTEST_API FSavedMove_MTCharacter : public FSavedMove_Character
{
  using Super = FSavedMove_Character;

  // 이동 정보에 플레이어가 스프린트 키를 누르고 있는지 저장합니다.
  uint8 bPressingSprint : 1;

  // 이동 정보에 플레이어가 순간이동 키를 눌렀는지를 저장합니다.
  uint8 bPressedTeleport : 1;

public:
  FSavedMove_MTCharacter() = default;
  virtual ~FSavedMove_MTCharacter() override = default;

  // 단순히 FSavedMove_Character의 변수들을 초기화하는데 사용됩니다.
  // bPressingSprint, bPressedTeleport를 false로 만듭니다.
  virtual void Clear() override;

  // 클라이언트와 서버에서 이동을 계산하기 전 캐릭터의 이동 정보를 저장합니다.
  // 예를 들면 위치, 회전, Velocity, 타임스탬프, 가속도 등을 FSavedMove_Character에 저장합니다.
  // 이 상속된 함수는
  // bPressingSprint = CMC->IsPressingSprint();
  // bPressedTeleport = CMC->IsPressedTeleport();
```

```
  // 를 포함합니다.
  virtual void SetMoveFor(ACharacter* C, float InDeltaTime, FVector const& NewAccel, FNetworkPredictionData_Client_Character& ClientData) o

  // 클라이언트에서 이동을 재현하기 전 FSavedMove_Character의 정보들을
  // UCharacterMovementComponent에 정보들을 대입합니다. 특, 이동정보를 이용해서
  // 캐릭터의 움직임을 재현하는 것입니다.
  // 이 상속된 함수는
  // CMC->SetPressingSprint(bPressingSprint);
  // CMC->SetPressedTeleport(bPressedTeleport);
  // 를 포함합니다.
  virtual void PrepMoveFor(ACharacter* C) override;

  // 클라이언트와 서버가 데이터를 주고 받을 때 대역폭을 줄이기 위해
  // 클라이언트에서는 아직 서버로 보내지 못한 이동정보를 새로 생긴 이동정보를
  // 병합하기도 합니다. NewMove와 병합을 할 수 있는지를 판단하는 함수입니다.
  // 이 상속된 함수는 보무 함수 호출에 더해서
  // if (bPressingSprint != NewMovePtr->bPressingSprint || bPressedTeleport != NewMovePtr->bPressedTeleport)
  // {
  //  return false;
  // }
  // 를 포함합니다.
  virtual bool CanCombineWith(const FSavedMovePtr& NewMove, ACharacter* InCharacter, float MaxDelta) const override;

  // 입력 정보들을 1비트로 압축해서 반환합니다.
  virtual uint8 GetCompressedFlags() const override;
};
```

Extending FNetworkPredictionData_Client_Character

- `UCharacterMovementComponent` 는 `FNetworkPredictionData_Client_Character*` 를 멤버로 가지고 있고 이 클래스는 이동정보들을 감싸고 있습니다.
- 우리가 상속내린 `FSavedMove_Character` 를 사용하기 위해 `FNetworkPredictionData_Client_Character` 를 다음과 같이 상속합니다.

```
class MULTIPLAYTEST_API FNetworkPredictionData_Client_MTCharacter : public FNetworkPredictionData_Client_Character
{
  using Super = FNetworkPredictionData_Client_Character;

public:
  explicit FNetworkPredictionData_Client_MTCharacter(const UCharacterMovementComponent& ClientMovement);
  virtual ~FNetworkPredictionData_Client_MTCharacter() override = default;

  virtual FSavedMovePtr AllocateNewMove() override
  {
    return MakeShareable(new FSavedMove_MTCharacter());     // 이렇게 하면 우리가 상속내린 FsavedMove_MTCharacter를 사용합니다.
  }
};
```

Implement Inherited UCharacterMovementComponet's Functions

- 이제 클라이언트와 서버가 주고 받을 데이터는 마련해두었으니 `UMTCharacterMovementComponent` 의 함수들이 어떻게 구현되어 있는지 살펴보겠습니다.

```
UMTCharacterMovementComponent::UMTCharacterMovementComponent()
  : MaxWalkSprintSpeed(1000.f)
  , TeleportDistance(500.f)
  , bPressingSprint(false)
  , bPressedTeleport(false)
{
}

FNetworkPredictionData_Client* UMTCharacterMovementComponent::GetPredictionData_Client() const
{
  if (ClientPredictionData == nullptr)
  {
    UMTCharacterMovementComponent* MutableThis = const_cast<UMTCharacterMovementComponent*>(this);
```

```
    MutableThis->ClientPredictionData = new FNetworkPredictionData_Client_MTCharacter(*this);
  }

  return ClientPredictionData;
}

void UMTCharacterMovementComponent::PhysicsRotation(float DeltaTime)
{
  // 부모 클래스인 UCharacterMovementComponent에서는 bOrientRotationToMovement가 켜져 있으면
  // RotationRate를 이용 캐릭터를 회전시킵니다.
  // 이는 RotationRate를 보간할때 일정한 비율(RotationRate)를 이용하기 때문에 회전을 할 때 마다
  // 같은 속도로 캐릭터가 회전하게 됩니다.
  // 아래 코드는 그와는 다르게 보간의 비율을 회전할 때마다 다르게 하여 목표값에 가까울 수록
  // 느리게 회전하여 더욱 부드럽게 회전합니다.

  const FRotator CurrentRotation = UpdatedComponent->GetComponentRotation();

  if (Acceleration.Size() > 0.f)
  {
    FRotator TurnTo(FMath::RInterpTo(CurrentRotation, Acceleration.GetSafeNormal().Rotation(), DeltaTime,
                    CharacterTurnRate));
    TurnTo.Pitch = 0.f;
    TurnTo.Roll = 0.f;
    TurnTo.Yaw = FRotator::NormalizeAxis(TurnTo.Yaw);

    MoveUpdatedComponent( FVector::ZeroVector, TurnTo, /*bSweep*/ false );
  }
}

void UMTCharacterMovementComponent::PerformMovement(float DeltaTime)
{
  Super::PerformMovement(DeltaTime);

  // 부모 클래스의 PerformMovement()를 호출한 뒤 순간이동을 계산합니다.
  if (bPressedTeleport)
  {
    check(CharacterOwner);
    FRotator Rot = CharacterOwner->GetControlRotation();

    if (Rot.Pitch > 90)
    {
      Rot.Pitch = 0;
    }
    const FVector Direction = Rot.Vector();
    CharacterOwner->SetActorLocation(Direction * TeleportDistance + CharacterOwner->GetActorLocation(), true);
    bPressedTeleport = false;
  }
}

float UMTCharacterMovementComponent::GetMaxSpeed() const
{
  // UE에서는 속도를 계산할 때 MovementMode에 따라 이미 정해진 최대 속도를 구하여 정합니다.
  float MaxSpeed;

  switch(MovementMode)
  {
  case MOVE_Walking:
  case MOVE_NavWalking:
    MaxSpeed = IsCrouching() ? MaxWalkSpeedCrouched : MaxWalkSpeed;
    MaxSpeed = bPressingSprint ? MaxWalkSprintSpeed : MaxWalkSpeed;
    break;
  case MOVE_Falling:
    MaxSpeed = MaxWalkSpeed;
    break;
  case MOVE_Swimming:
    MaxSpeed = MaxSwimSpeed;
    break;
  case MOVE_Flying:
    MaxSpeed = MaxFlySpeed;
    break;
  case MOVE_Custom:
    MaxSpeed = MaxCustomMovementSpeed;
    break;
  case MOVE_None:
  default:
    MaxSpeed = 0.f;
    break;
  }

  return MaxSpeed;
}
```

```
void UMTCharacterMovementComponent::UpdateFromCompressedFlags(uint8 Flags)
{
  Super::UpdateFromCompressedFlags(Flags);

  // 커스텀 Flag를 다음과 같이 정의했습니다.
  /*
    FLAG_Custom_0   = 0x10,   //bPressedSprint
    FLAG_Custom_1   = 0x20,   //bPressedTeleport
    FLAG_Custom_2   = 0x40,
    FLAG_Custom_3   = 0x80,
  */
  bPressingSprint = (Flags & FSavedMove_Character::FLAG_Custom_0) != 0;
  bPressedTeleport = (Flags & FSavedMove_Character::FLAG_Custom_1) != 0;
}
```

Additional Information

- 새로 상속내린 UMTCharacterMovementComponent를 사용하기 위해 우리가 사용하는 캐릭터의 생성자에서 다음과 같이 코드를 작성해야 합니다.

```
AMTCharacter::AMTCharacter(const FObjectInitializer& ObjectInitializer)
  : Super(ObjectInitializer.SetDefaultSubobjectClass<UMTCharacterMovementComponent>(CharacterMovementComponentName))
{}
```

- `CharacterMovementComponentName` 은 ACharacter에 static으로 선언된 `FName` 입니다.
- 이런식으로 자식 클래스의 컴포넌트의 클래스를 셋할 수 있습니다.
- 위에 언급했던 다른 구조체들을 상속하지 않은 이유에 대해 궁금하실 겁니다.
- 위에 작성된 코드에는 2개의 부울변수가 추가됐지만 CompressedFlag에 압축해서 보내기 때문에 전송되는 데이터의 크기는 변함이 없습니다.
- 만약 2개의 부울변수를 압축해서 보내지 않는다고 가정했을 때 비트 필드를 사용하면 데이터는 1바이트가 추가될 것이고, 그렇지 않다면 2바이트가 추가될 것입니다.
- 이런 상황을 방지하기 위해 입력정보를 압축하는 것이고, 이렇게 하면 굳이 위에 언급했었던 `FCharacterNetworkMoveData`, `FCharacterNetworkMoveDataContainer`, `FClientAdjustment` , `FCharacterMoveResponseDataContainer` 에 변수를 추가할 이유가 없어지는 것입니다.
- 만약 새로 정의할 움직임의 정보를 보내야 하는데 CompressedFlag에 다 담을 수 없다거나 아니면 boolean말고도 다른 변수를 보내야 겠다고 하면 `FCharacterNetworkMoveData`, `FCharacterNetworkMoveDataContainer`, `FClientAdjustment` , `FCharacterMoveResponseDataContainer` 에 변수를 추가하고 상속한 클래스나 구조체를 사용하도록 설정해야 할 것입니다.
- 관련 정보는 https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Networking/CharacterMovementComponent/ 의 **Customizing Networked Character Movement** 섹션에 나와 있으니 참고하시면 됩니다.