Práctica Redes III

Multiplayer

En los anteriores ejercicios creamos la lógica de juego y establecimos la conexión a través del GUI, creando además los coches de los diferentes clientes y dándoles el control. En esta práctica vamos a probar diferentes maneras de comunicación para la actualización de la lógica de juego.

NetComponent

Vamos a crearnos un componente nuevo que será en el que realizamos las tareas de red. Añadirecos este componente a nuestro coche y publicaremos una función en nuestro gestor que nos dará el coche que manejamos como jugadores.

```
class UNetComponent;
class CARS_API ACar : public APawn
{
protected:
  //Net
  UPROPERTY(EditAnywhere)
  UNetComponent* m_pNet;
};
```

```
#include "GameNet/NetComponent.h"
ACar::ACar()
{
  m_pNet = CreateDefaultSubobject<UNetComponent>(TEXT("Net"));
}

// Called every frame
void ACar::Tick(float DeltaTime)
{
  m_pNet->SetInput(m_vMovementInput);
}
```

```
class CGameNetMrg : public Net::CManager::IObserver
{
public:
  ACar* GetOwnCar() const;
};
```

```
ACar* CGameNetMrg::GetOwnCar() const
{
  if (m_pManager && m_vPlayers.find(m_pManager->getID()) != m_vPlayers.end())
  {
    return m_vPlayers.at(m_pManager->getID());
  }
  return nullptr;
}
```

Con esta información ya podemos crear nuestro componente de red, en el que el primer paso será discernir en el Tick() qué debemos hacer: Enviar información del estado del coche (si es el nuestro) o simplemente actualizarla (si hemos recibido de fuera la información del estado porque pertenece a otro cliente.

```
#include "CoreMinimal.h"
#include "Components/ActorComponent.h"
```

```
#include "NetComponent.generated.h"

namespace Net
{
  class CManager;
}

UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
class CARS_API UNetComponent : public UActorComponent
{
  GENERATED_BODY()
public:
  // Sets default values for this component's properties
  UNetComponent();
  // Called every frame
  virtual void TickComponent(float DeltaTime, ELevelTick TickType,
FActorComponentTickFunction* ThisTickFunction) override;
  // Input
  void SetInput(FVector2D _vInput) { m_vMovementInput = _vInput; }

protected:
  // Called when the game starts
  virtual void BeginPlay() override;

protected:
  //Input vector
  FVector2D m_vMovementInput = FVector2D::ZeroVector;
  Net::CManager* m_pManager = nullptr;
};
```

```
#include "NetComponent.h"
#include "GameFramework/Actor.h"
#include "GameNet/GameBuffer.h"
#include "Net/Manager.h"
#include "Engine/World.h"
#include "CarsGameModeBase.h"
#include "Kismet/GameplayStatics.h"
#include "Game/Car.h"

UNetComponent::UNetComponent()
{
  PrimaryComponentTick.bCanEverTick = true;
  m_pManager = Net::CManager::getSingletonPtr();
}

void UNetComponent::BeginPlay()
{
      Super::BeginPlay();
}

void UNetComponent::TickComponent(float DeltaTime, ELevelTick TickType,
FActorComponentTickFunction* ThisTickFunction)
{
  Super::TickComponent(DeltaTime, TickType, ThisTickFunction);

  ACarsGameModeBase* pGameMode = Cast<ACarsGameModeBase>(GetWorld()-
>GetAuthGameMode());
  if (pGameMode)
  {
    if (pGameMode->GetGameNetMgr().GetOwnCar() == GetOwner())
```

```
    {
      GEngine->AddOnScreenDebugMessage(-1, 5.0f, FColor::Green,
*FString("Owner"));
    }
    else
    {
      GEngine->AddOnScreenDebugMessage(-1, 5.0f, FColor::Green,
*FString("Other"));
    }
  }
}
```

Una vez hayamos identificado cuál es nuestro papel (en función de si es nuestro coche o no), nuestra primera aproximación va a ser enviar el input por la red. Para eso crearemos un paquete con un identificador especial (como los de cargar partida, jugador, etc.), en el que luego añadiremos la información que necesitemos.

```
void UNetComponent::TickComponent(float DeltaTime, ELevelTick TickType,
FActorComponentTickFunction* ThisTickFunction)
{
  Super::TickComponent(DeltaTime, TickType, ThisTickFunction);

  ACarsGameModeBase* pGameMode = Cast<ACarsGameModeBase>(GetWorld()-
>GetAuthGameMode());
  if (pGameMode)
  {
    if (pGameMode->GetGameNetMgr().GetOwnCar() == GetOwner())
    {
      CGameBuffer oBuffer;
      oBuffer.write(NetMessageType::ACTOR_MSG);
      oBuffer.write(m_pManager->getID());
      oBuffer.write(m_vMovementInput);
    }
    else
    {
      //ACar* pCar = Cast<ACar>(GetOwner());
      //if (pCar)
      //{
      //  pCar->SetInput(m_vMovementInput);
      //}
    }
  }
}
```

Por otro lado, en el gestor de red del juego capturaremos el paquete enviado y actuaremos en consecuencia, actualizando el actor “marioneta”. También deberemos contemplar que si el que lo recibe es el servidor deberá reenviarlo al resto de clientes, para que también vean el movimiento.

```
void CGameNetMrg::dataPacketReceived(Net::CPaquete* packet)
{
  // Creamos un buffer con los datos para leer de manera más cómoda
  CGameBuffer data;
  data.write(packet->getData(), packet->getDataLength());
  data.reset();
  NetMessageType eMessageType;
  data.read(eMessageType);
  switch (eMessageType)
  {
  case NetMessageType::ACTOR_MSG:
```

```
  {
    unsigned int uClient;
    data.read(uClient);
    if (uClient != m_pManager->getID())
    {
      ACar* pCar = m_vPlayers.at(uClient);
      FVector2D vInput;
      data.read(vInput);
      pCar->SetInput(vInput);
    }
  } break;
  default:
    break;
  }
}
```

```
class CARS_API ACar : public APawn
{
public:
  void SetInput(FVector2D _vInput) { m_vMovementInput = _vInput; }
};
```