Práctica Redes II

Lobby

En este apartado empezaremos a montar todo el tema de inicialización y conexión de red. Para ello nos apoyaremos en UMG para crear un sencillo sistema de GUI.

GUI

Para poder crear nuestros modos de juego cargamos el módulo UMG.

```
public Cars(ReadOnlyTargetRules Target) : base(Target)
{
  PCHUsage = PCHUsageMode.UseExplicitOrSharedPCHs;
  PublicDependencyModuleNames.AddRange(new string[] { "Core", "CoreUObject",
"Engine", "InputCore", "UMG" });
}
```

Vamos a llevar en nuestro game mode (`ACarsGameModeBase`) la gestión de cambios de estado. Para la gestión de estados vamos a tener diferentes UserWidgets en el que gestionaremos eventos de GUI y llamaremos a funciones propias que declararemos en `ACarsGameModeBase`. Lo primero será declararnos una función para cambiar de widget, así como un widget actual y otro inicial. Esta función se encargará de llamar a las funciones `AddToViewport()` al widget entrante y a `RemoveFromViewport()` al saliente.

```
class CARS_API ACarsGameModeBase : public AGameModeBase
{
  GENERATED_BODY()
public:
  UFUNCTION(BlueprintCallable, Category = CarsNet)
  void ChangeMenuWidget(TSubclassOf<UUserWidget> NewWidgetClass);
protected:
  /** Called when the game starts. */
  virtual void BeginPlay() override;
  UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = CarsNet)
  TSubclassOf<UUserWidget> StartingWidgetClass;
  UPROPERTY()
  UUserWidget* CurrentWidget;
};
```

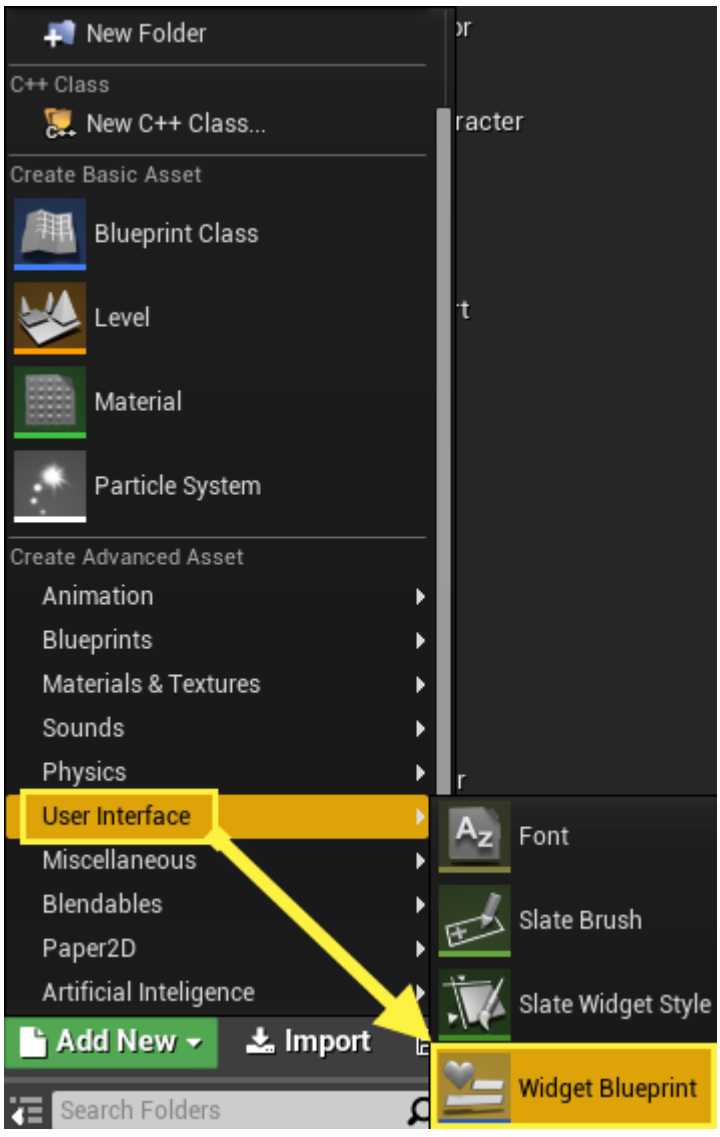
```
#include "Blueprint/UserWidget.h"

void ACarsGameModeBase::BeginPlay()
{
  Super::BeginPlay();
  ChangeMenuWidget(StartingWidgetClass);
}

void ACarsGameModeBase::ChangeMenuWidget(TSubclassOf<UUserWidget> NewWidgetClass)
{
  if (CurrentWidget != nullptr)
  {
    CurrentWidget->RemoveFromViewport();
    CurrentWidget = nullptr;
  }
  if (NewWidgetClass != nullptr)
  {
    CurrentWidget = CreateWidget<UUserWidget>(GetWorld(), NewWidgetClass);
    if (CurrentWidget != nullptr)
```

```
    {
      CurrentWidget->AddToViewport();
    }
  }
}
```

Además tenemos que cambiar el input mode del player controller para que reacciones con los elemento del GUI

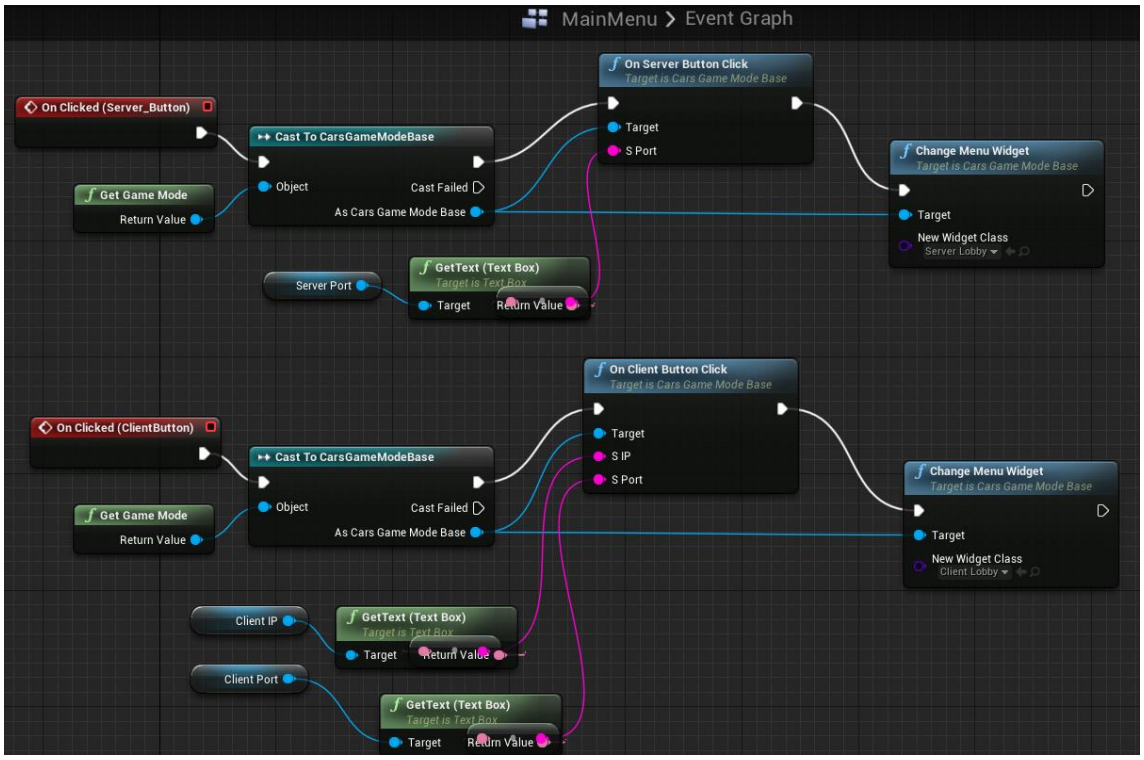
```
void ACarsPlayerController::BeginPlay()
{
  Super::BeginPlay();
  SetInputMode(FInputModeGameAndUI());
}
```

Añadimos nuestro primer modo



Y le llamamos MainMenu. Crearemos botones y campos de texto para obtener lo siguiente:

Y asignaremos a los botones eventos, creando con ellos el siguiente blueprint



Pero para poder terminarlo tendremos que publicar e implementar las funciones que se lanzarán con los eventos de pulsado de botones.

Net

Cuando se pulsen los botones del menú lo que tendremos que hacer por fin es empezar con la conexión gestión de conexión.

```
class CARS_API ACarsGameModeBase : public AGameModeBase
{
  GENERATED_BODY()
public:
  UFUNCTION(BlueprintCallable, Category = CarsNet)
  void OnServerButtonClick(FString sPort);
  UFUNCTION(BlueprintCallable, Category = CarsNet)
  void OnClientButtonClick(FString sIP, FString sPort);
};
```

De momento será todo muy básico, simplemente atenderemos conexiones de los clientes y enviaremos algún paquete de prueba de un extremo a otro. Para ello disponemos de un gestor que nos da el interfaz que necesitamos para la comunicación en red, ya seamos clientes o servidores. Deberemos inicializarlo en el constructor y para ser notificados de los eventos deberemos implementar la interfaz `Net::CManager::IObserver`. Lo suyo sería implementarlo directamente en `ACarsGameModeBase`, pero Unreal no nos permite hacer herencia múltiple así que lo apañamos implementándolo en otra clase, de la cual `ACarsGameModeBase` te tendrá un variable miembro.

```
class CGameNetMrg : public Net::CManager::IObserver
{
public:
  CGameNetMrg();
  CGameNetMrg(ACarsGameModeBase* _pController);
  virtual ~CGameNetMrg() { }

  // Net::CManager::IObserver
  virtual void dataPacketReceived(Net::CPaquete* packet);
  virtual void connexionPacketReceived(Net::CPaquete* packet);
  virtual void disconnexionPacketReceived(Net::CPaquete* packet);

  void Tick();
private:
  Net::CManager* m_pManager = nullptr;
  ACarsGameModeBase* m_pController;
  };
```

```
#include "GameNetMrg.h"

CGameNetMrg::CGameNetMrg() : m_pController(nullptr)
{
  m_pManager = Net::CManager::getSingletonPtr();
}

CGameNetMrg::CGameNetMrg(ACarsGameModeBase* _pController) :
m_pController(_pController)
{
  m_pManager = Net::CManager::getSingletonPtr();
}

void CGameNetMrg::Tick()
{
  m_pManager->tick();
}
```

La instancia de esta clase que tendremos en nuestro modo de juego deberá registrarse como observer en el gestor de red y deregistrarse cuando ya no sea necesaria. Esto lo hacemos en `ACarsGameModeBase::BeginPlay()` y en EndPlay(). También deberemos encargarnos de llamar a su tick todos los frames para que se notifiquen los eventos de red.

```
ACarsGameModeBase::ACarsGameModeBase(const class FObjectInitializer&
ObjectInitializer) : AGameModeBase(ObjectInitializer), m_oObserver(this)
{
  PrimaryActorTick.bCanEverTick = true;
  PlayerControllerClass = ACarsPlayerController::StaticClass();
  m_pManager = Net::CManager::getSingletonPtr();
}

void ACarsGameModeBase::BeginPlay()
```

```
{
  Super::BeginPlay();
  ChangeMenuWidget(StartingWidgetClass);
  m_pManager->addObserver(&m_oGameNetMrg);
}

void ACarsGameModeBase::EndPlay(EEndPlayReason::Type eEndPlayReason)
{
  Super::EndPlay(eEndPlayReason);
  m_pManager->removeObserver(&m_oGameNetMrg);
}

void ACarsGameModeBase::Tick(float DeltaSeconds)
{
  Super::Tick(DeltaSeconds);
  m_oGameNetMrg->tick();
}
```

Por fin podemos dedicarnos a implementar las funciones a las que se invocan cuando hay algún evento de pulsado de ratón. Como el gestor de red ya está inicializado lo único que tendremos que hacer es inicializarnos como cliente o servidor y, en el caso del cliente, establecer la conexión.

```
void ACarsGameModeBase::OnServerButtonClick(FString sPort)
{
  GEngine->AddOnScreenDebugMessage(-1, 5.0f, FColor::Green, *FString("Server"));
  m_pManager->activateAsServer(FCString::Atoi(*sPort));
}

void ACarsGameModeBase::OnClientButtonClick(FString sIP, FString sPort)
{
  GEngine->AddOnScreenDebugMessage(-1, 5.0f, FColor::Green, *FString("Client"));

  m_pManager->activateAsClient();
  m_pManager->connectTo(TCHAR_TO_ANSI(*sIP), FCString::Atoi(*sPort));
}

void ACarsGameModeBase::OnServerStartButtonClick()
{
  GEngine->AddOnScreenDebugMessage(-1, 5.0f, FColor::Green, *FString("Server
Start!"));
  UGameplayStatics::OpenLevel(GetWorld(), "Circuit1");
}
```

Para comprobar que los paquetes de red se envían y reciben bien en los dos extremos lo que haremos será enviar un mensaje cuando recibamos una conexión en el servido (`CGameNetMrg::connexionPacketReceived`), usando para crear el mensaje un `CGameBuffer`. En el cliente, cuando se recibe el mensaje (`CGameNetMrg::dataPacketReceived`) probaremos a imprimir el contenido de éste. Apoyándonos también en `CGameBuffer` para la lectura.

```
void CGameNetMrg::dataPacketReceived(Net::CPaquete* packet)
{
  if (m_pManager->getID() == Net::ID::SERVER)
  {
  }
  else
  {
    // Creamos un buffer con los datos para leer de manera más cómoda
```

```
    Net::CBuffer data;
    data.write(packet->getData(), packet->getDataLength());
    data.reset();
    char sInfo[128];
    data.read(sInfo, data.getSize());
    if (GEngine)
    {
      GEngine->AddOnScreenDebugMessage(-1, 5.0f, FColor::Red, sInfo);
    }
  }
}

void CGameNetMrg::connexionPacketReceived(Net::CPaquete* packet)
{
  if (m_pManager->getID() == Net::ID::SERVER)
  {
    // Creamos un buffer con los datos para leer de manera más cómoda
    Net::CBuffer data;
    const char* sHello = "Connected";
    data.write(sHello, sizeof(sHello));
    m_pManager->send(data.getbuffer(), data.getSize());
    if (GEngine)
    {
      GEngine->AddOnScreenDebugMessage(-1, 5.0f, FColor::Red, "Client connected!
");
    }
  }
  else
  {

  }
}

void CGameNetMrg::disconnexionPacketReceived(Net::CPaquete* packet)
{

}
```

A la clase `Net::CBuffer` se le han añadido métodos propios para serializar y deserializar diferente tipos básicos como int, float, char*. Por otro lado, para la implementación de métodos de serializar y deserializar de tipos complejos se ha creado la clase que estamos usando, `CGameBuffer`.

```
class CGameBuffer : public Net::CBuffer
{
public:
  void write(const FVector& data);
  void read(FVector& data);
};

void CGameBuffer::write(const FVector& data)
{
  write(data.X);
  write(data.Y);
  write(data.Z);
}

void CGameBuffer::read(FVector& data)
{
  read(data.X);
  read(data.Y);
```

```
  read(data.Z);
}
```

El siguiente paso que daremos es ya empezar a transmitir mensajes con criterio. Para diferenciar los tipos de mensaje tenemos creado un enumerado `NetMessageType` que podemos serializar y añadir valores.

```
void ACarsGameModeBase::OnServerStartButtonClick()
{
  GEngine->AddOnScreenDebugMessage(-1, 5.0f, FColor::Green, *FString("Server
Start!"));
  UGameplayStatics::OpenLevel(GetWorld(), "Circuit1");
  Net::CBuffer data;
  Net::NetMessageType iID = Net::START_GAME;
  data.write(&iID, sizeof(iID));
  m_pManager->send(data.getbuffer(), data.getSize());
}
```

Player creation:

En lugar de simplemente mandar mensajes, como hemos hecho antes, vamos a empezar a dar estructura a los mensajes que se mandan. Todo mensaje empezará por un identificador, que será de tipo `NetMessageType` en función de su valor, al recibirlo, actuaremos de una u otra manera.

Al pulsar el botón de 'Start', después de que se haya producido la conexión, empezaremos el proceso de carga. El Servidor mandará que se cargue un mapa concreto, los clientes lo cargarán y, una vez se haya cargado el mapa en todos los clientes, el servidor enviará mensajes para que se carguen los coches correspondientes a cada uno de los jugadores. En este proceso, al crear los coches en el cliente, cada cliente deberá ver qué coche le corresponde y establecer el input a dicho coche.

```
void ACarsGameModeBase::OnServerStartButtonClick()
{
  GEngine->AddOnScreenDebugMessage(-1, 5.0f, FColor::Green, *FString("Server
Start!"));
  UGameplayStatics::OpenLevel(GetWorld(), "Circuit1");
  CGameBuffer data;
  NetMessageType iID = NetMessageType::LOAD_MAP;
  data.write(iID);
  const char* sLevel = "Circuit1";
  data.write(sLevel);
  m_pManager->send(data.getbuffer(), data.getSize());
}
```

```
class CGameNetMrg : public Net::CManager::IObserver
{
private:
  std::map<unsigned int, ACar*> m_vPlayers;
  unsigned int m_uMapLoadedNotifications = 0u;
};
```

```
#include "GameNetMrg.h"
#include "GameNet/GameBuffer.h"
#include "Net/paquete.h"
#include "Kismet/GameplayStatics.h"
#include "CarsGameModeBase.h"
#include "Engine/World.h"
#include "Engine/LevelStreaming.h"
#include "Game/Car.h"
```

```
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::LOAD_MAP:
  {
    char sMap[128];
    data.read(sMap);
    UGameplayStatics::OpenLevel(m_pController->GetWorld(), sMap);
    NetMessageType iID = NetMessageType::MAP_LOADED;
    m_pManager->send(&iID, sizeof(iID));
  } break;
  case NetMessageType::MAP_LOADED:
  {
    ++m_uMapLoadedNotifications;
    if (m_uMapLoadedNotifications >= m_pManager->getConnections().size())
    {
      for (auto client : m_pManager->getConnections())
      {
        CGameBuffer data;
        NetMessageType iID = NetMessageType::LOAD_PLAYER;
        data.write(iID);
        data.write(client.first);
        FVector vPos(220, -310.f + client.first * 40.f, 0.f);
        data.write(vPos);
        FActorSpawnParameters SpawnInfo;
        SpawnInfo.Name = FName("Car", client.first);
        SpawnInfo.SpawnCollisionHandlingOverride =
ESpawnActorCollisionHandlingMethod::AlwaysSpawn;
        ACar* pCar = m_pController->GetWorld()->SpawnActor<ACar>(vPos,
FRotator::ZeroRotator, SpawnInfo);
        if (pCar)
        {
          m_vPlayers[client.first] = pCar;
          m_pManager->send(data.getbuffer(), data.getSize());
        }
    }
  } break;
  case NetMessageType::LOAD_PLAYER:
  {
    unsigned int uClient;
    data.read(uClient);
    FVector vPos;
    data.read(vPos);
    FActorSpawnParameters SpawnInfo;
    SpawnInfo.Name = FName("Car", uClient);
    SpawnInfo.SpawnCollisionHandlingOverride =
ESpawnActorCollisionHandlingMethod::AlwaysSpawn;
    if (m_pController->GetWorld())
    {
      ACar* pCar = m_pController->GetWorld()->SpawnActor<ACar>(vPos,
FRotator::ZeroRotator, SpawnInfo);
      if (pCar)
      {
        m_vPlayers[uClient] = pCar;
        if (uClient == m_pManager->getID())
        {
```

```
          APlayerController* OurPlayerController = GEngine-
>GetFirstLocalPlayerController(m_pController->GetWorld());
          if (OurPlayerController)
          {
            OurPlayerController->SetPawn(pCar);
          }
          pCar->AutoPossessPlayer = EAutoReceiveInput::Player0;
        }
      }
    }
  } break;
  default:
    break;
  }
}

void CGameNetMrg::connexionPacketReceived(Net::CPaquete* packet)
{
}

void CGameNetMrg::disconnexionPacketReceived(Net::CPaquete* packet)
{
}
```

Deberemos acordarnos también de borrar de la escena el coche que ya teníamos y borrar la línea del constructor de ACar que hacía que se le asignase el input al coche.

```
ACar::ACar()
{
  AutoPossessPlayer = EAutoReceiveInput::Player0;
}
```