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.

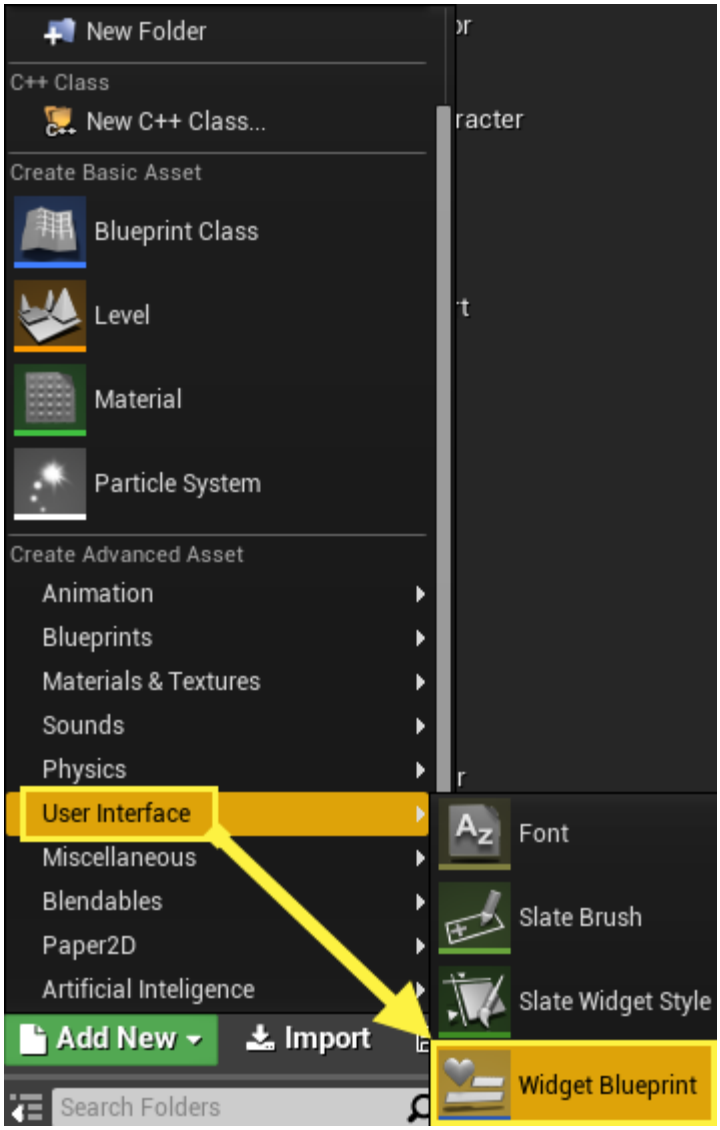
```
class CARS_API ACarsGameModeBase : public AGameModeBase
{
  GENERATED_BODY()
public:
  UFUNCTION(BlueprintCallable, Category = CarsNet)
  void ChangeMenuWidget(TSubclassOf<UUserWidget> NewWidgetClass);
protected:
  /** Called when the game starts. */
  virtual void BeginPlay() override;
  UPROPERTY(EditAnywhere,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

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

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

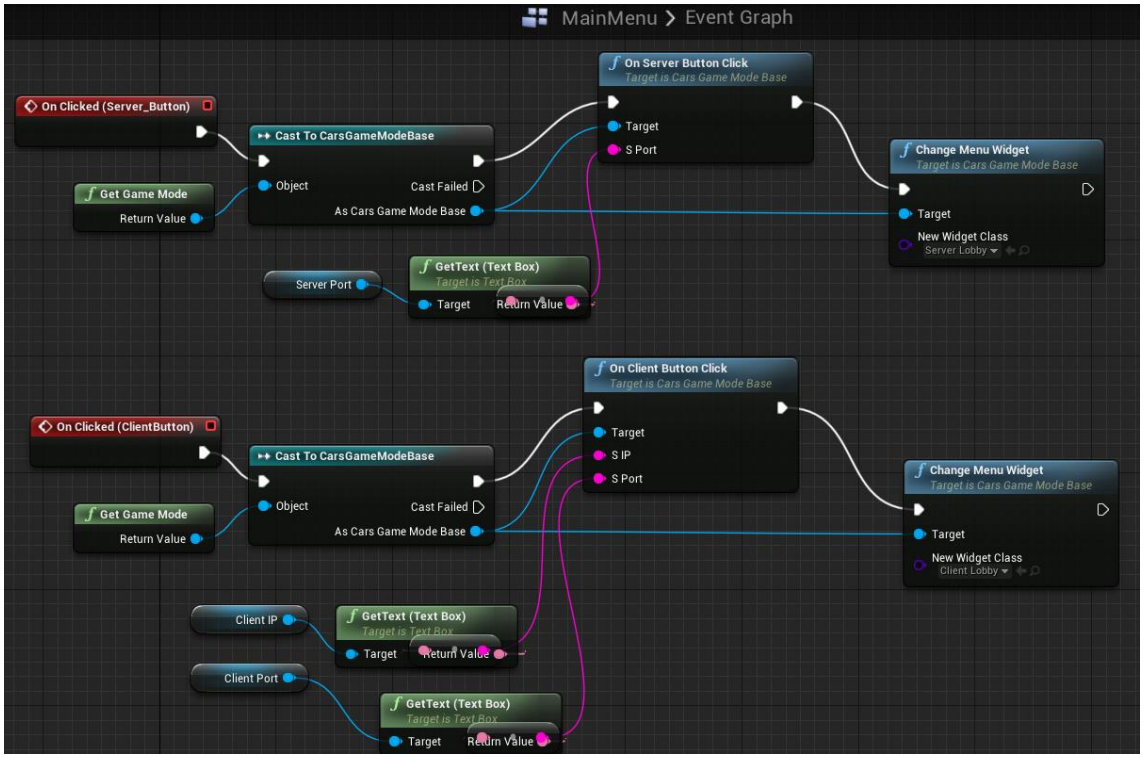
```
void ACarsPlayerController::BeginPlay()
{
  Super::BeginPlay();
  SetInputMode(FInputModeGameAndUI());
}
```

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 CServerObserver : public Net::CManager::IObserver
  {
  public:
    CServerObserver();
    CServerObserver(ACarsGameModeBase* _pController);
    virtual ~CServerObserver() { }

    // Net::CManager::IObserver
```

```
    virtual void dataPacketReceived(Net::CPaquete* packet);
    virtual void connexionPacketReceived(Net::CPaquete* packet);
    virtual void disconnexionPacketReceived(Net::CPaquete* packet);
  private:
    Net::CManager* m_pManager = nullptr;
    ACarsGameModeBase* m_pController;
  };
```

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.

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 (`ACarsGameModeBase::CServerObserver::connexionPacketReceived`), usando para crear el mensaje un `Net::CBuffer`. En el cliente, cuando se recibe el mensaje (`ACarsGameModeBase::CServerObserver::dataPacketReceived`) probaremos a imprimir el contenido de éste. Apoyándonos también en `Net::CBuffer` para la lectura.

Sería bastante interesante que en `Net::CBuffer` tuviese métodos propios para serializar y deserializar diferente tipos básicos como int, float, char*… e incluso otros no tan básicos como FVector por ejemplo.

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