Capítulo 1

Introducción

En este documento se mostrará todo el código diseñado y realizado para el proyecto.

El código principal está realizado en el lenguaje C++, y tiene las características y funciones propias del motor gráfico Unreal Engine, ya que en todo momento se codifica en base a éste. Así mismo, se mostrará los Blueprints creados, al ser un lenguaje de script como tal, y se tratará de la misma manera.

Así, este documento se divide en las funciones creadas con C++ y con Blueprints, siendo a su vez separadas por clases y/o documentos.

1.1. Ficheros

Capítulo 2

Código C++

A continuación mostrará cada una de las clases codificadas en lenguaje C++:

2.1. Clase *AbrirPuerta*

Esta clase es creada para poder abrir una clase por medio de un *trigger* de superposición, y que ésta se cierre al cabo de un tiempo.

2.1.1. Función *BeginPlay*

Ésta función se activa al comienzo del nivel y guarda el personaje usado por el jugador como el actor que puede abrir la puerta.

```
1 void UAbrirPuerta::BeginPlay()
2 {
3   Super::Beginplay();
4   Owner = GetOwner();
5
6   ActorThatOpens = GetWorld()->GetFirstPlayerController()->GetPawn();
7   actorRotation = Owner->GetActorRotation();
8 }
```

2.1.2. Función *OpenDoor*

Esta función es la que se encarga de abrir la puerta

```
1 void UAbrirPuerta::OpenDoor().
2 {
3   Owner->SetActorRotation( actorRotation + FRotator(0.f, -90.f, 0.f));
4 }
```

2.1.3. Función *CloseDoor*

Esta función es la que se encarga de cerrar la puerta.

```
1 void UAbrirPuerta::CloseDoor()
2 {
3   Owner->SetActorRotation(actorRotation);
4 }
```

2.1.4. Función *TickComponent*

Esta función se activa en cada tick.

```
1  void UAbrirPuerta::TickComponent(float DeltaTime, ELevelTick TickType,
       ↪ FActorComponentTickFunction* ThisTickFunction)
2  {
3    Super::TickComponent(DeltaTime, TickType, ThisTickFunction);
4    if (PressurePlate->IsOverlappingActor(ActorThatOpens))
5    {
6      OpenDoor();
7      LastDoorOpenTime = GetWorld()->GetTimeSeconds();
8    }
9
10   if (GetWorld()->GetTimeSeconds() - LastDoorOpenTime > DoorCloseDelay)
11   {
12     CloseDoor();
13   }
14 }
```

2.2. Clase MyBlueprintFunctionLibrary

Esta clase es la usada para crear en C++ nuevos nodos para usar en tus Blueprints

2.2.1. Fichero MyBlueprintFunctionLibrary.cpp

2.2.1.1. Función SaveGame

Esta función implementa la posibilidad de guardar partida en cualquier momento.

```
1  void UMyBlueprintFunctionLibrary::SaveGame(AActor * MyActor)
2  {
3    UMySaveGame* SaveGameInstance = Cast<UMySaveGame>(UGameplayStatics::
       ↪ CreateSaveGameObject(UMySaveGame::StaticClass()));
4
5    SaveGameInstance->PlayerLocation = MyActor->GetActorLocation();
6
7    FString LevelName = MyActor->GetWorld()->GetMapName();
8    LevelName.RemoveFromStart(MyActor->GetWorld()->StreamingLevelsPrefix)
       ↪ ;
9    SaveGameInstance->LevelName = LevelName;
10
11   UGameplayStatics::SaveGameToSlot(SaveGameInstance, TEXT("MySlot"), 0)
       ↪ ;
12   FString cadenaLocation = SaveGameInstance->PlayerLocation.ToString();
13
14   GEngine->AddOnScreenDebugMessage(-1, 5.f, FColor::Green, TEXT("Juego
       ↪ Guardado."));
15 }
```

2.2.1.2. Función LoadGame

Esta función implementa la posibilidad de cargar en cualquier momento una partida anteriormente guardada.

```
1  FVector UMyBlueprintFunctionLibrary::LoadGame(AActor * MyActor)
2  {
3    UMySaveGame* SaveGameInstance = Cast<UMySaveGame>(UGameplayStatics::
       ↪ CreateSaveGameObject(UMySaveGame::StaticClass()));
4
5    SaveGameInstance = Cast<UMySaveGame>(UGameplayStatics::
       ↪ LoadGameFromSlot("MySlot", 0));
6
7    UGameplayStatics::OpenLevel(MyActor->GetWorld(), FName(*
       ↪ SaveGameInstance->LevelName));
8
9    return SaveGameInstance->PlayerLocation;
10 }
```

2.3. Clase Grabber

Esta clase implementa una mecánica de agarrar objetos.

2.3.1. Archivo Grabber.h

```
1  // Fill out your copyright notice in the Description page of Project
       ↪ Settings.
2
3  #pragma once
4
5  #include "CoreMinimal.h"
6  #include "PhysicsEngine/PhysicsHandleComponent.h"
7  #include "Components/InputComponent.h"
8  #include "Components/ActorComponent.h"
9  #include "Grabber.generated.h"
10
11 UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
12 class PRUEBA_API UGrabber : public UActorComponent
13 {
14   GENERATED_BODY()
15
16 public:
17   // Sets default values for this component's properties
18   UGrabber();
19
20 protected:
21   // Called when the game starts
22   virtual void BeginPlay() override;
23
24 public:
25   // Called every frame
26   virtual void TickComponent(float DeltaTime, ELevelTick TickType,
       ↪ FActorComponentTickFunction* ThisTickFunction) override;
27
28 private:
29   float Reach = 200.f;
30
31   UPhysicsHandleComponent* PhysicsHandle = nullptr;
32
33   UInputComponent* InputComponent = nullptr;
34
35   void Grab();
36   void Released();
37
38   FHitResult GetFirstPhysicsBodyInReach() const;
39
40 };
```

2.3.2. Archivo Grabber.cpp

2.3.2.1. Función BeginPlay

```
1  void UGrabber::BeginPlay()
2  {
3    Super::BeginPlay();
4    UE_LOG(LogTemp, Warning, TEXT("Grabber!"));
5
6    PhysicsHandle = GetOwner()->FindComponentByClass<
       ↪ UPhysicsHandleComponent>();
7    InputComponent = GetOwner()->FindComponentByClass<UInputComponent>();
8
9    if (InputComponent) {
10     UE_LOG(LogTemp, Warning, TEXT("Estamos perfectos con el input
       ↪ component"));
11
12     InputComponent->BindAction("Agarrar", IE_Pressed, this, &UGrabber::
       ↪ Grab);
13     InputComponent->BindAction("Agarrar", IE_Released, this, &UGrabber
       ↪ ::Released);
14   }
15   else
16   {
17     UE_LOG(LogTemp, Error, TEXT("%s missing input component"), *
       ↪ GetOwner()->GetName());
18   }
19
20   if (PhysicsHandle)
21   {
22     UE_LOG(LogTemp, Warning, TEXT("Estamos perfectos con el physics
       ↪ component"));
23   }
24   else
25   {
26     UE_LOG(LogTemp, Error, TEXT("%s missing physics component"), *
       ↪ GetOwner()->GetName());
27   }
28
29 }
```

2.3.2.2. Función Grab

```
1  void UGrabber::Grab()
2  {
3    auto HitResult = GetFirstPhysicsBodyInReach();
4    auto ComponentToGrab = HitResult.GetComponent();
5    auto ActorHit = HitResult.GetActor();
6
7    if (ActorHit)
8    {
9      PhysicsHandle->GrabComponent(
10       ComponentToGrab,
11       NAME_None,
12       ComponentToGrab->GetOwner()->GetActorLocation(),
13       true
14     );
15   }
16
17 }
```

2.3.2.3. Función Released

```
void UGrabber::Released()
{
  PhysicsHandle->ReleaseComponent();
}
```

2.3.2.4. Función GetFirstPhysicsBodyInReach

```
FHitResult UGrabber::GetFirstPhysicsBodyInReach() const {
  FVector PlayerViewPointLocation;
  FRotator PlayerViewPointRotation;

  GetWorld()->GetFirstPlayerController()->GetPlayerViewPoint(
    OUT PlayerViewPointLocation,
    OUT PlayerViewPointRotation
  );

  FVector LineTraceEnd = PlayerViewPointLocation +
    ↪ PlayerViewPointRotation.Vector() * Reach;

  FCollisionQueryParams TraceParameters(FName(TEXT("")), false,
    ↪ GetOwner());

  FHitResult Hit;

  GetWorld()->LineTraceSingleByObjectType(
    OUT Hit,
    PlayerViewPointLocation,
    LineTraceEnd,
    FCollisionObjectQueryParams(ECollisionChannel::ECC_PhysicsBody),
    TraceParameters
  );

  return Hit;
}
```

2.3.2.5. Función TickComponent

```
void UGrabber::TickComponent(float DeltaTime, ELevelTick TickType,
    ↪ FActorComponentTickFunction* ThisTickFunction)
{
  Super::TickComponent(DeltaTime, TickType, ThisTickFunction);
  FVector PlayerViewPointLocation;
  FRotator PlayerViewPointRotation;
  GetWorld()->GetFirstPlayerController()->GetPlayerViewPoint(
    OUT PlayerViewPointLocation,
    OUT PlayerViewPointRotation
  );

  FVector LineTraceEnd = PlayerViewPointLocation +
    ↪ PlayerViewPointRotation.Vector() * Reach;

  if (PhysicsHandle->GrabbedComponent)
```

```
14   {
15     PhysicsHandle->SetTargetLocation(LineTraceEnd);
16   }
17
18 }
```

2.4. Clase ReporteDePosicion

Esta clase implementa un sistema para conocer la posición actual de cualquier actor al comienzo del nivel, usado con el fin de realizar pruebas.

2.4.1. Función BeginPlay

Esta es la función que muestra por consola dónde se encuentra el actor que use esta clase:

```
1 void UReporteDePosicion::BeginPlay()
2 {
3   Super::BeginPlay();
4   FString Nombre = GetOwner()->GetName();
5   FString Posicion = GetOwner()->GetTransform().GetLocation().ToString
      ↪ ();
6   UE_LOG(LogTemp, Warning, TEXT("Hola, soy %s y estoy en %s"), *Nombre,
      ↪  *Posicion);
7 }
```

2.5. Clase Watcher

Esta clase es usada para la cámara de vigilancia del nivel 1.

2.5.1. Fichero Watcher.h

```
1  // Fill out your copyright notice in the Description page of Project
       ↪ Settings.
2
3  #pragma once
4
5  #include "GameFramework/Actor.h"
6  #include "Engine/TriggerVolume.h"
7  #include "CoreMinimal.h"
8  #include "Components/ActorComponent.h"
9  #include "Watcher.generated.h"
10
11
12 UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
13 class PRUEBA_API UWatcher : public UActorComponent {
14   GENERATED_BODY()
15
16 public:
17   // Sets default values for this component's properties
18   UWatcher();
19
20 protected:
21   // Called when the game starts
22   virtual void BeginPlay() override;
23   void FadeOut();
24
25 public:
26   // Called every frame
27   virtual void TickComponent(float DeltaTime, ELevelTick TickType,
       ↪ FActorComponentTickFunction* ThisTickFunction) override;
28
29   UPROPERTY(EditAnywhere, BlueprintReadWrite)
30   ATriggerVolume* WatcherTrigger;
31
32   AActor* MyActor;
33   AActor* Owner;
34
35   UPROPERTY(EditAnywhere, BlueprintReadWrite)
36   FVector ActorPosition;
37   UPROPERTY(EditAnywhere, BlueprintReadWrite)
38   bool restart = true;
39 };
```

2.5.2. Fichero Watcher.cpp

2.5.2.1. Constructor

```
1  UWatcher::UWatcher() {
2
```

```
3   // Set this component to be initialized when the game starts, and to
      ↪ be ticked every frame.  You can turn these features
4   // off to improve performance if you don't need them.
5   PrimaryComponentTick.bCanEverTick = true;
6   restart = true;
7
8 }
```

2.5.2.2. Función BeginPlay

```
1 void UWatcher::BeginPlay()
2 {
3
4   Super::BeginPlay();
5   Owner = GetOwner();
6
7   MyActor = GetWorld()->GetFirstPlayerController()->GetPawn();
8
9 }
```

2.5.2.3. Función FadeOut

```
1 void UWatcher::FadeOut()
2 {
3   APlayerController * Controller = UGameplayStatics::
      ↪ GetPlayerController(GetWorld(), 0);
4   class APlayerController * MyPC = Cast<APlayerController>(Controller);
5   MyPC->ClientSetCameraFade(true, FColor::Black, FVector2D(1.0, 0.0),
      ↪ 10.0);
6 }
```

2.5.2.4. Función TickComponent

```
1 void UWatcher::TickComponent(float DeltaTime, ELevelTick TickType,
      ↪ FActorComponentTickFunction* ThisTickFunction)
2 {
3   Super::TickComponent(DeltaTime, TickType, ThisTickFunction);
4   if (WatcherTrigger->IsOverlappingActor(MyActor))
5   {
6     FadeOut();
7     MyActor->SetActorLocation(ActorPosition);
8     restart = false;
9   }
10
11 }
```

Capítulo 3

Blueprints

En este capitulo se mostrará cada uno de los blueprints programados para el proyecto. Cada uno se dividirá en módulos lo suficientemente individuales para poder ser explicados por separado, ya que el tamaño de éstos es tan grande que no es posible mostrarlo todo a la vez.

3.1. Blueprint MyGameInstance

Este blueprint es lo que se le llama la instancia del juego. Aquí se almacenan las variables globales, así como las funciones globales.

3.1.1. Variables

- **GotInput**: Esta variable sirve para detectar cuándo se ha introducido una respuesta en un test.
- **NPC_ID**: determina el identificador del actor en una conversación.
- **LineCurrent**: determina el identificador de la línea de diálogo actual que se muestra en pantalla.

3.1.2. Get Answer

Esta función devuelve las respuestas de un test dado el identificador de la pregunta.

3.1.2.1. Leer base de datos

Lo primero que se hace es acceder a la base de datos de diálogo y sacar todas las filas, como podemos observar en la figura 3.1

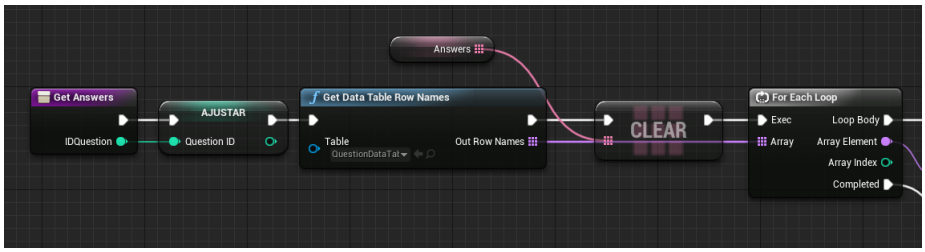

Figura 3.1: MyGameInstance GetAnswer1

3.1.2.2. Guardar respuestas correspondientes

Recoge las respuestas que tengan el identificador de pregunta que se pasó por argumento. Cuando se haya recorrido toda la base de datos, termina la función y devuelve las respuestas encontradas. Lo podemos ver en la figura 3.2

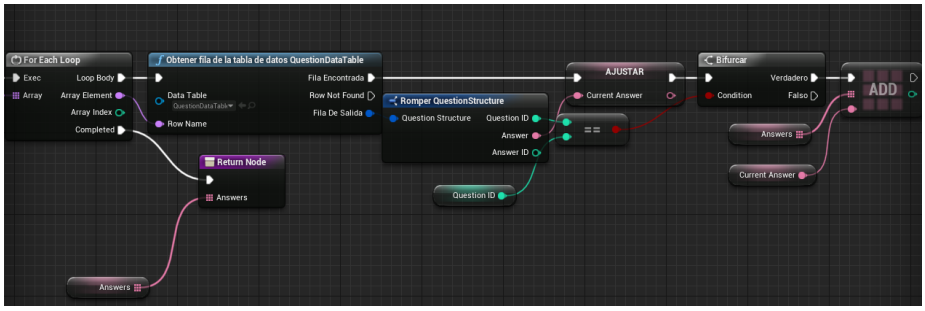

Figura 3.2: MyGameInstance GetAnswer2

3.2. Blueprint MyHUD

Este es el blueprint correspondiente al HUD, es decir, la información que aparece en pantalla en todo momento. En nuestro caso es únicamente una retícula, así que se ha realizado lo que se muestra en la figura 3.3.

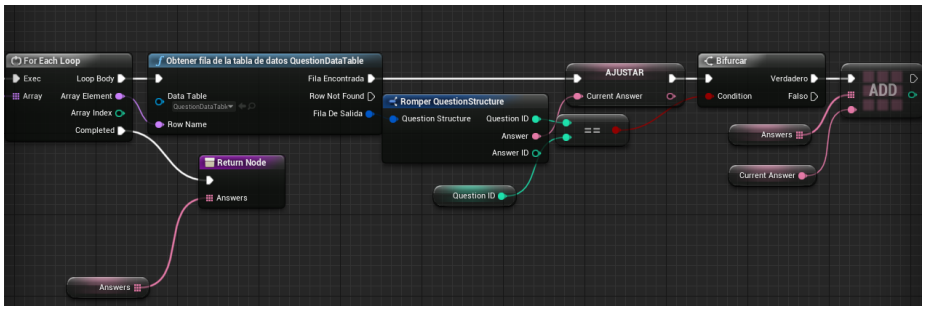

Figura 3.3: MyHUD

3.3. Blueprint ButtonTest1

Este blueprint se usa para el actor responsable de mostrar el test perteneciente al nivel 2. Se encarga tanto de mostrar el diálogo en pantalla, como de realizar las acciones y cálculos necesarios para que el test funcione correctamente.

3.3.1. Variables

Las variables correspondientes a este blueprint son las siguientes:

- **NPC_ID**: determina el identificador del actor
- **Conversation_ID**: determina el identificador de la conversación, que cambia al pasar correctamente todo el test.
- **LineCurrent**: determina el identificador de la línea de diálogo actual que se muestra en pantalla.
- **DialogueText**: texto de la línea de diálogo actual.
- **MyDialogue**: recoge todas las líneas de diálogo correspondientes al actor.
- **DialogueWidget**: la interfaz de diálogo que aparece en pantalla en las líneas de diálogo sin test.
- **DialogueBoxWidget**: la interfaz de diálogo que aparece en pantalla en las líneas de diálogo con test.

- **actorVariable**: actor que posee las variables del nivel. Se guarda para acceder a ellas rápidamente.
- **isQuestion**: booleano que indica si la línea de diálogo actual es una pregunta o no.
- **testPhase**: variable que indica la fase del test en la que nos encontramos (¡¡¡¡¡¡¡ver si se puede eliminar con la variable de la base de datos!!!!!!!!!!)
- **Answers**: array que contiene las respuestas ordenadas, para saber así cual es la correcta.
- **Options**: respuestas del test que, además de estar desordenadas, están en el tipo de variable que necesita la interfaz.

3.3.2. BeginPlay

BeginPlay es el evento que se ejecuta al comienzo del nivel.

3.3.2.1. Visibilidad de esferas y variables de nivel

Lo primero que hace es determinar la visibilidad de las esferas que formaran parte del nivel. Además, recoge el actor que posee las variables de nivel, para usarlas más adelante. Se puede ver en la figura 3.4

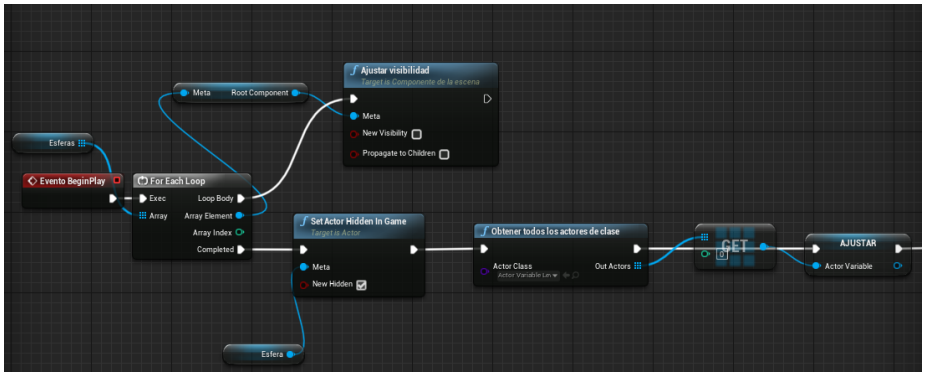

Figura 3.4: ButtonTest1 BeginPlay 1

3.3.2.2. Diálogo Test

Después, recoge y guarda en una variable todo el diálogo correspondiente a este actor, determinado por la variable NPC_ID. Se muestra en la figura 3.5

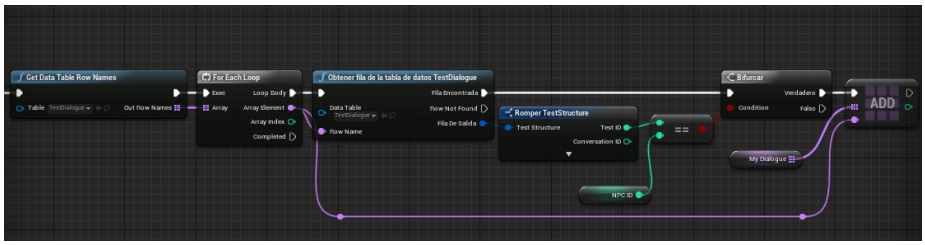

Figura 3.5: ButtonTest1 BeginPlay 2

3.3.3. Interact

Este evento se activa cuando el personaje interactúa con el actor con la tecla correspondiente.

3.3.3.1. Cambio de cámara

Lo primero que hace es cambiar la cámara, apuntando hacia el escenario en el que van a suceder los eventos, para tener una clara visión de qué ocurrirá a medida que avances por el test

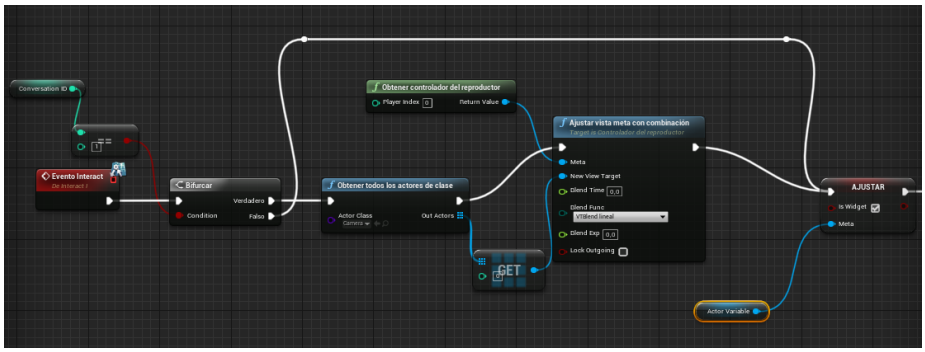

Figura 3.6: ButtonTest1 Interact 1

3.3.3.2. Crear y añadir interfaz

A continuación crea el widget o interfaz de diálogo y lo añade a la pantalla. Además, recoge la instancia de juego para acceder a las variables y funciones globales. Lo vemos en la figura 3.7

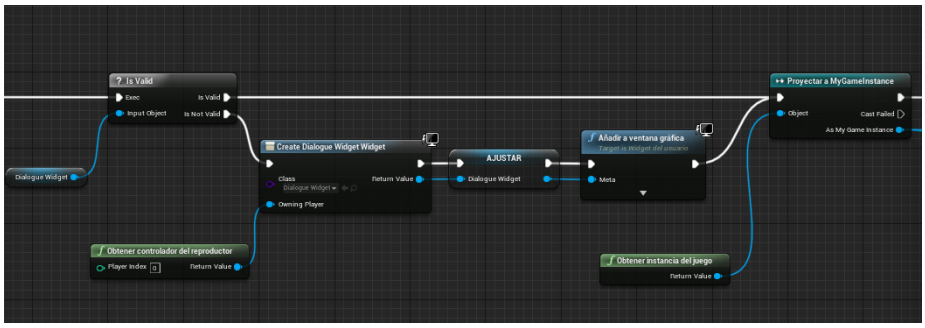

Figura 3.7: ButtonTest1 Interact 2

3.3.3.3. Recoger diálogo y detectar pregunta

Gracias a la función global, se recoge el diálogo y el resto de variables de la base de datos. Además, si fuese una pregunta, llama a la función para recoger las respuestas. Se puede observar en la figura 3.8

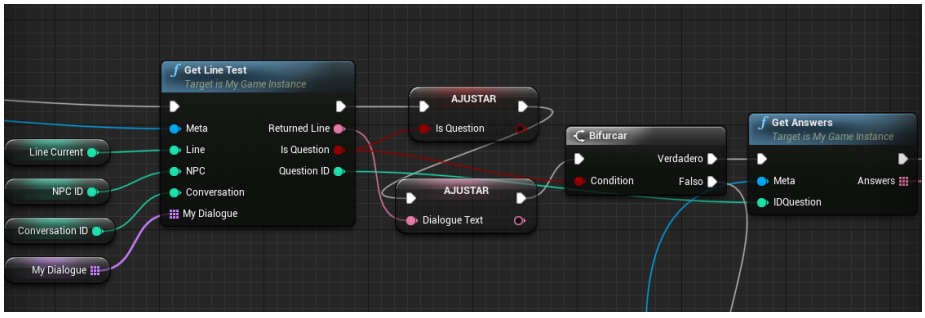

Figura 3.8: ButtonTest1 Interact 3

3.3.3.4. Recoger respuestas

A continuación, si el diálogo es una pregunta, usa una función global para recoger las respuestas correspondientes a la pregunta. Las almacena y luego las convierte al tipo de variable necesario, para después volver a almacenarlas desordenadas, como se puede observar en la figura 3.9

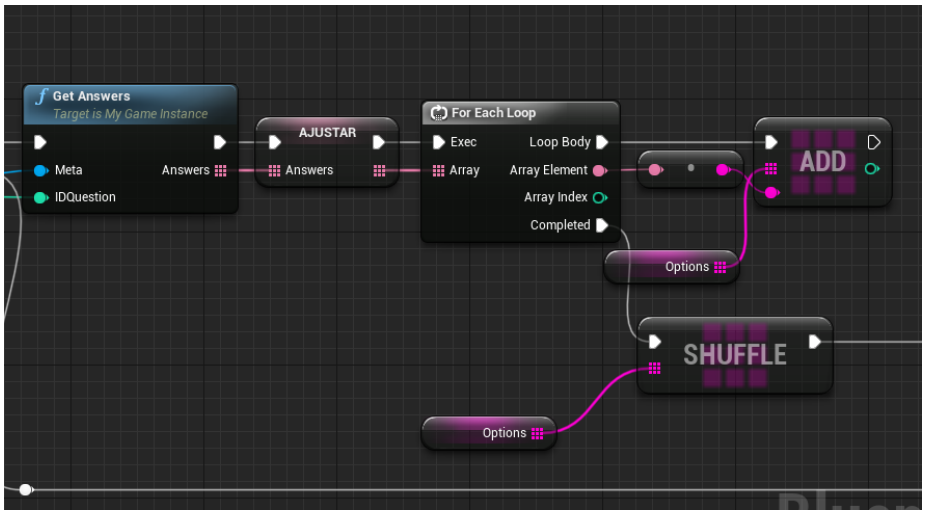

Figura 3.9: ButtonTest1 Interact 4

aaaaaaaa