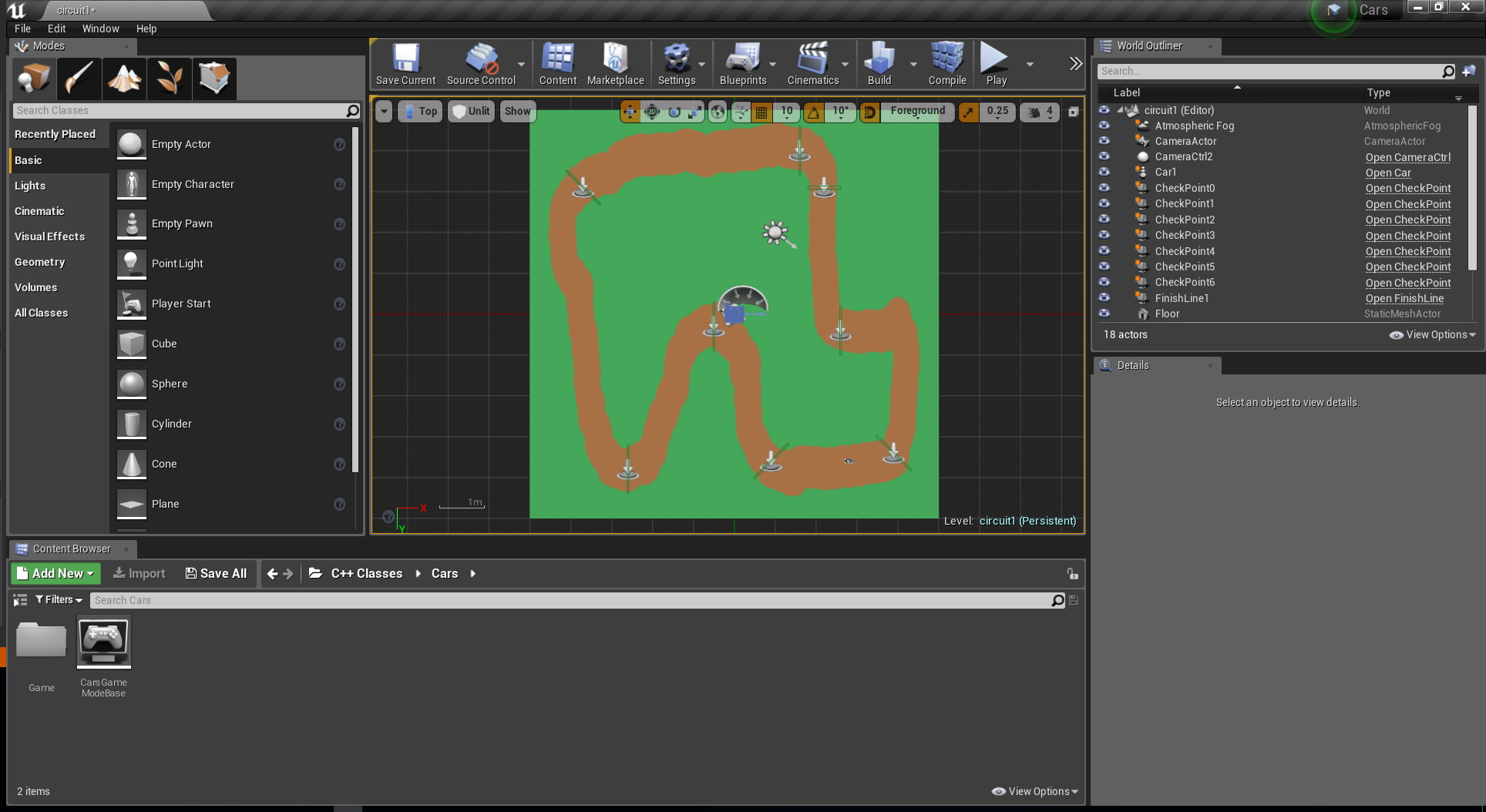
# Practica Redes

Estos días vamos a hacer diversas prácticas de red sobre un mini juego de coches muy sencillo. Todas las prácticas que hagamos serán en Unreal, sin embargo no vamos a usar las facilidades de uso en red de Unreal. Para ello nos montaremos nuestra propia infraestructura de red basada en las prácticas que hicisteis en C++ con Juanan. Durante estas prácticas vamos a promover el uso de C++, intentando huir de los blueprints en la medida de lo posible. Lo que aprendamos aquí será extrapolable a otros motores e incluso nos permitirá comunicar aplicaciones realizadas en diferentes motores, plataformas o lenguajes (por ejemplo una aplicación hecha en Unreal para PC comunicada con una aplicación hecha en Unity para android).

## Juego monojugador

Antes de ponernos con la práctica en si vamos a crear un sencillo juego sobre el que luego realizaremos el resto de ejercicios. El juego será un juego de coches 2D visto desde arriba, donde controlaremos un coche con WASD al que seguirá la cámara. La única funcionalidad extra que tendremos que implementar será la acumulación vueltas al circuito y contabilizando la mejor vuelta.

### Circuito

Al abrir el proyecto tendremos un cuidado y detallado circuito sobre el que se realizará el juego. 

Camera

#include "Kismet/GameplayStatics.h"

// Called when the game starts or when spawned

void AGameCamera::BeginPlay()

{

APlayerController\* OurPlayerController = UGameplayStatics::GetPlayerController(this, 0);

if (OurPlayerController)

{

OurPlayerController->SetViewTarget(this);

}

}

Lo primero que haremos será crear nuestro coche, que heredará de APawn

UCLASS()

class CARS\_API ACar : public APawn

{

GENERATED\_BODY()

public:

// Sets default values for this pawn's properties

ACar();

protected:

// Called when the game starts or when spawned

virtual void BeginPlay() override;

public:

// Called every frame

virtual void Tick(float DeltaTime) override;

// Called to bind functionality to input

virtual void SetupPlayerInputComponent(class UInputComponent\* PlayerInputComponent) override;

float GetVelocityMagnitude() { return m\_vVelocity.Size(); }

protected:

UPROPERTY(EditAnywhere)

float m\_fAcel = 120.f;

UPROPERTY(EditAnywhere)

float m\_fDrag = 50.f;

UPROPERTY(EditAnywhere)

float m\_fBrakeAcel = 300.f;

UPROPERTY(EditAnywhere)

float m\_fMaxVelocity = 250.f;

UPROPERTY(EditAnywhere)

float m\_fRotationFactor = 2.f;

FVector m\_vVelocity = FVector::ZeroVector;

//Input variables

FVector m\_vMovementInput;

//Input functions

void Move(float AxisValue);

void Turn(float AxisValue);

FVector CalculateAceleration(float DeltaTime);

void CalculateVelocity(FVector \_vAcel, float DeltaTime);

};

Cámara de juego

class ACar;

UCLASS()

class CARS\_API AGameCamera : public ACameraActor

{

public:

//

inline void SetTarget(ACar\* \_pTarget) { m\_pTarget = \_pTarget; }

private:

UPROPERTY(EditAnywhere)

ACar\* m\_pTarget;

UPROPERTY(EditAnywhere)

float m\_fMinDistance = 200.f;

UPROPERTY(EditAnywhere)

float m\_fDistanceFromVelocityFactor = 0.6f;

};

#include "GameCamera.h"

#include "Car.h"

#include "Kismet/GameplayStatics.h"

// Called every frame

void AGameCamera::Tick(float DeltaTime)

{

Super::Tick(DeltaTime);

if (m\_pTarget)

{

FVector vNewPos = m\_pTarget->GetActorTransform().GetTranslation();

vNewPos.Z = m\_fMinDistance + m\_pTarget->GetVelocityMagnitude() \* m\_fDistanceFromVelocityFactor;

FTransform oTransform = GetActorTransform();

oTransform.SetTranslation(vNewPos);

SetActorTransform(oTransform);

}

}

void ACar::BeginPlay()

{

APlayerController\* PlayerController = UGameplayStatics::GetPlayerController(this, 0);

if (OurPlayerController)

{

AGameCamera\* \_pCtrl = Cast<AGameCamera>(PlayerController->GetViewTarget());

if (\_pCtrl)

{

\_pCtrl->SetTarget(this);

}

}

}

Funcionalidad de CheckPoints y FinishLine

#include "CoreMinimal.h"

#include "Engine/TriggerBox.h"

#include "CheckPoint.generated.h"

class AFinishLine;

UCLASS()

class CARS\_API ACheckPoint : public ATriggerBox

{

GENERATED\_BODY()

public:

// overlap begin function

UFUNCTION()

void OnOverlapBegin(class AActor\* OverlappedActor, class AActor\* OtherActor);

inline void SetFinishLine(AFinishLine\* \_pFinishLine) { m\_pFinishLine = \_pFinishLine; }

private:

AFinishLine \* m\_pFinishLine;

};

#include "CheckPoint.h"

// include draw debu helpers header file

#include "DrawDebugHelpers.h"

#include "FinishLine.h"

ACheckPoint::ACheckPoint()

{

//Register Events

OnActorEndOverlap.AddDynamic(this, &ACheckPoint::OnOverlapBegin);

}

// Called when the game starts or when spawned

void ACheckPoint::BeginPlay()

{

Super::BeginPlay();

DrawDebugBox(GetWorld(), GetActorLocation(), GetComponentsBoundingBox().GetExtent(), FColor::Purple, true, -1, 0, 5);

}

void ACheckPoint::OnOverlapBegin(class AActor\* OverlappedActor, class AActor\* OtherActor)

{

// check if Actors do not equal nullptr and that

if (OtherActor && (OtherActor != this) && m\_pFinishLine)

{

m\_pFinishLine->AddPassedCheckPoint(this);

}

}

class CARS\_API AFinishLine : public ATriggerBox

{

public:

// overlap begin function

UFUNCTION()

void OnOverlapBegin(class AActor\* OverlappedActor, class AActor\* OtherActor);

void AddPassedCheckPoint(ACheckPoint\* \_pCheckPoint);

protected:

UPROPERTY(EditAnywhere, BlueprintReadWrite)

TArray<ACheckPoint\*> m\_vCheckPoints;

TArray<ACheckPoint\*> m\_vPassedCheckPoints;

unsigned int m\_uLaps = 0u;

float m\_fPreviousTime = 0.f;

float m\_fLastLapTime = 0.f;

float m\_fBestTime = FLT\_MAX;

#include "FinishLine.h"

// include draw debu helpers header file

#include "DrawDebugHelpers.h"

#include "CheckPoint.h"

#include "Engine/World.h"

AFinishLine::AFinishLine()

{

//Register Events

OnActorEndOverlap.AddDynamic(this, &AFinishLine::OnOverlapBegin);

}

// Called when the game starts or when spawned

void AFinishLine::BeginPlay()

{

Super::BeginPlay();

for (ACheckPoint\* pCP : m\_vCheckPoints)

{

pCP->SetFinishLine(this);

}

DrawDebugBox(GetWorld(), GetActorLocation(), GetComponentsBoundingBox().GetExtent(), FColor::Green, true, -1, 0, 5);

}

void AFinishLine::AddPassedCheckPoint(ACheckPoint\* \_pCheckPoint)

{

if (!m\_vPassedCheckPoints.Contains(\_pCheckPoint))

{

m\_vPassedCheckPoints.Add(\_pCheckPoint);

}

}

void AFinishLine::OnOverlapBegin(class AActor\* OverlappedActor, class AActor\* OtherActor)

{

// check if Actors do not equal nullptr and that

if (OtherActor && (OtherActor != this))

{

float fCurrentTime = GetWorld()->GetTimeSeconds();// UGameplayStatics::GetRealTimeSeconds(GetWorld());

if (m\_vPassedCheckPoints.Num() == m\_vCheckPoints.Num())

{

m\_fLastLapTime = fCurrentTime - m\_fPreviousTime;

m\_fPreviousTime = fCurrentTime;

if (m\_fBestTime > m\_fLastLapTime)

{

m\_fBestTime = m\_fLastLapTime;

// Put up a debug message for five seconds. The -1 "Key" value (first argument) indicates that we will never need to update or refresh this message.

GEngine->AddOnScreenDebugMessage(-1, 5.0f, FColor::Green, \*FString("Best Lap!"));

}

++m\_uLaps;

GEngine->AddOnScreenDebugMessage(-1, 5.0f, FColor::Green, \*FString("Lap " + FString::FromInt(m\_uLaps) + "(" + FString::SanitizeFloat(m\_fLastLapTime) + " seconds)"));

}

else

{

if (GEngine)

{

// Put up a debug message for five seconds. The -1 "Key" value (first argument) indicates that we will never need to update or refresh this message.

GEngine->AddOnScreenDebugMessage(-1, 5.0f, FColor::Red, \*FString("Invalid lap!"));

}

}

m\_vPassedCheckPoints.Reset();

}

}