Práctica Redes

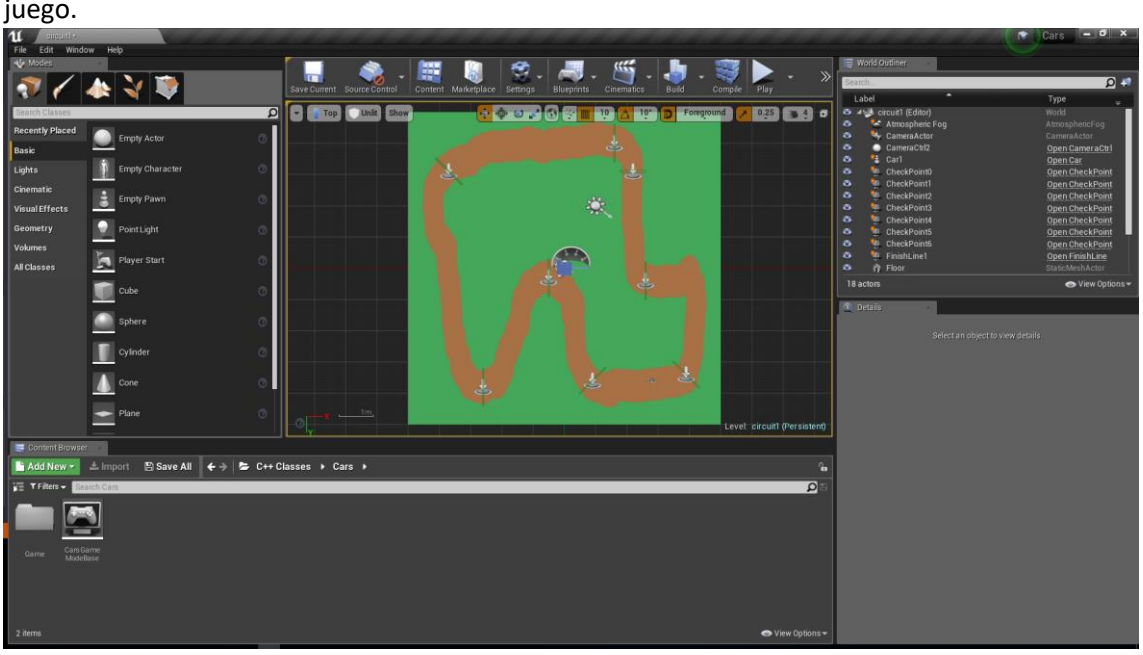
Estos días vamos a hacer diversas prácticas de red sobre un mini juego de coches muy sencillo. Todas las prácticas que hagamos serán en Unreal, sin embargo no vamos a usar las facilidades de uso en red de Unreal. Para ello nos montaremos nuestra propia infraestructura de red basada en las prácticas que 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, contabilizando la mejor vuelta.

Circuito

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



A parte del circuito ya hay colocadas unas cuantas entidades por el mapa como los check points, que nos garantizarán que hacemos una vuelta completa, la línea de meta, que comprobará que la vuelta dada es correcta y actualizará el número de vueltas y el mejor tiempo, la cámara que seguirá al coche y el propio coche que manejaremos.

Camera

Lo primero que tendremos que hacer es poner nuestra cámara como view target en el player controller, para que sea ésta la encargada de mostrar el juego. Esto lo haremos en el BeginPlay del AGameCamera.

```
#include "Kismet/GameplayStatics.h"
```

```
void AGameCamera::BeginPlay()
{
  APlayerController* OurPlayerController =
UGameplayStatics::GetPlayerController(this, 0);
  if (OurPlayerController)
  {
    OurPlayerController->SetViewTarget(this);
  }
}
```

Car

La clase ACar representa a el coche que controlaremos y será la encargada de gestionar el input que se recibe. En el proyecto se han declarado ya dos axis mappings (Move y Turn) que se controlan con WASD o con el stick izquierdo de un mando. Deberemos enlazar dos métodos de la clase ACar a dichos ejes y cuando se llamen modificaremos un vector de input que será miembro de la clase. A partir de este vector moveremos luego el coche.

El enlace de los métodos de la clase ACar con los ejes lo realizaremos en el método `SetupPlayerInputComponent`, que tendremos que sobrescribir, llamando al método `BindAxis` del `UInputComponent`.

```
class UCarMovementComponent;

class CARS_API ACar : public APawn
{
public:
  // Called to bind functionality to input
  virtual void SetupPlayerInputComponent(class UInputComponent* InputComponent)
override;

protected:
  //Input functions
  void Move(float AxisValue);
  void Turn(float AxisValue);

protected:
  //Input variables
  FVector2D m_vMovementInput = FVector2D::ZeroVector;
  //Movement
  UPROPERTY(EditAnywhere)
  UCarMovementComponent* m_pCarMovement;
};
```

```
#include "Components/InputComponent.h"

// Called when the game starts or when spawned
void ACar::BeginPlay()
{
  m_vMovementInput.Set(0.f, 0.f);
}

// Called to bind functionality to input
void ACar::SetupPlayerInputComponent(UInputComponent* PlayerInputComponent)
{
  Super::SetupPlayerInputComponent(PlayerInputComponent);
  PlayerInputComponent->BindAxis("Move", this, &ACar::Move);
  PlayerInputComponent->BindAxis("Turn", this, &ACar::Turn);
}

//Input functions
void ACar::Move(float AxisValue)
```

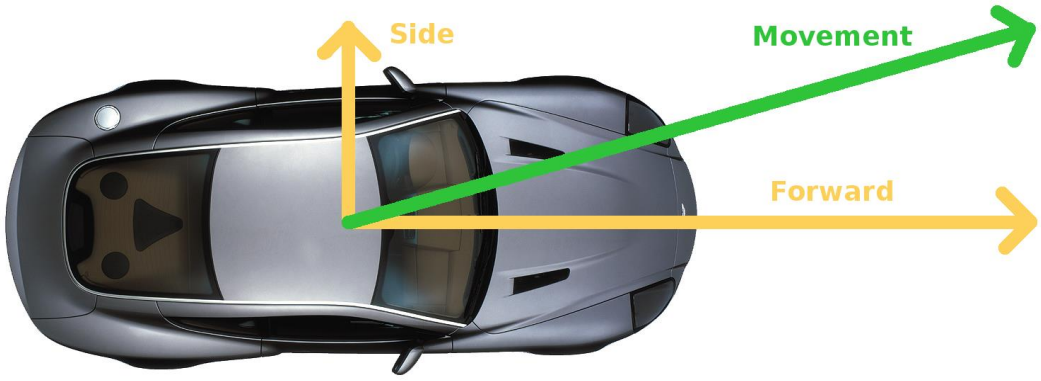
```
{
  m_vMovementInput.Y = FMath::Clamp<float>(AxisValue, -1.0f, 1.0f);
}

void ACar::Turn(float AxisValue)
{
  m_vMovementInput.X = FMath::Clamp<float>(AxisValue, -1.0f, 1.0f);
}
```

CarMovement

El movimiento del coche en si lo vamos a realizar en un componente a parte que luego añadiremos a nuestra clase ACar. Para ello creamos una clase en el editor que herede de UActorComponente y nos ponemos manos a la obra.

El movimiento del coche vamos a realizarlo mediante el uso de aceleraciones, las cuales afectarán al coche tanto en su eje vertical como en su eje horizontal.



El controlador que vamos a realizar no es el más ortodoxo ni realista del mundo, pero para este pequeño juego nos da los resultados deseados y es sencillo de implementar. Vamos a usar la fórmula del movimiento rectilíneo uniformemente acelerado que todos hemos usado en algún momento.

$$x = x_0 + (v_0 + a * t) * t$$

Por simplificar el cálculo del desplazamiento vamos a hacerlo por partes. En primer lugar calcularemos la aceleración del vehículo a partir del input. El input será un vector que tendremos como miembro en el componente y que será establecido por el actor en cada tick. Para evaluar cuánta aceleración aplicamos al vehículo tendremos declaradas y publicadas unas cuantas variables que definirán la aceleración de avance, de frenado, de rozamiento (cuando no se acelera o frena) y un factor de giro ya que la aceleración lateral se basará en la velocidad actual del coche.

```
v_Acel = 0
// Forward Accel
if is going forward
  v_Acel = FORWARD_ACCEL * Input.Y * ActorForwardVector
else if is braking
  v_Acel = BRAKE_ACCEL * Input.Y * ActorForwardVector
else
  v_Acel = -DRAG_ACCEL * ActorForwardVector
// Side Accel
v_Acel += Input.X * VelocitySize * ROTATION_FACTOR * ActorRightVector
```

Para los valores de las constantes de aceleración unos valores que funcionan bien son:

FORWARD_ACCEL = 120

BRAKE_ACCEL = 300

DRAG_ACCEL = 50

ROTATION_FACTOR = 2

```
UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
class CARS_API UCarMovementComponent : public UActorComponent
{
  GENERATED_BODY()

public:
  // Velocity magnitude
  float GetVelocityMagnitude() { return m_vVelocity.Size(); }
  // Input
  void SetInput(FVector2D _vInput) { m_vMovementInput = _vInput; }

protected:
  // Movement Calculation
  FVector CalculateAceleration();
  void UpdateVelocity(FVector& _vVelocity_, const FVector& _vAcel, float
DeltaTime);
  void MoveActor(const FVector& _vVelocity, float DeltaTime);

protected:
  // Constants
  UPROPERTY(EditAnywhere)
  float m_fAcel = 120.f;          // Forward accel
  UPROPERTY(EditAnywhere)
  float m_fDrag = 50.f;           // Drag accel (no input)
  UPROPERTY(EditAnywhere)
  float m_fBrakeAcel = 300.f;     // Brake accel
  UPROPERTY(EditAnywhere)
  float m_fMaxVelocity = 250.f;  // Max Velocity
  UPROPERTY(EditAnywhere)
  float m_fRotationFactor = 2.f; // For side accels
  //Current velocity
  FVector m_vVelocity = FVector::ZeroVector;
  //Input vector
  FVector2D m_vMovementInput = FVector2D::ZeroVector;
};
```

```
#include "GameFramework/Actor.h"

FVector UCarMovementComponent::CalculateAceleration()
{
  FVector v_Acel;
  if (m_vMovementInput.Y > 0.f)
    v_Acel = m_fAcel * m_vMovementInput.Y * GetOwner()->GetActorForwardVector();
  else if (m_vMovementInput.Y == 0.f)
    v_Acel = -m_fDrag * GetOwner()->GetActorForwardVector();
  else
    v_Acel = m_fBrakeAcel * m_vMovementInput.Y * GetOwner()-
>GetActorForwardVector();

  v_Acel += m_vMovementInput.X * m_vVelocity.Size() * m_fRotationFactor *
GetOwner()->GetActorRightVector();

  return v_Acel;
}
```

Una vez calculada la aceleración nos toca actualizar la velocidad. Digo actualizar porque como veis en la fórmula

$$v = (v_0 + a * t)$$

Lo que quiere decir que debemos almacenar el vector de la velocidad como miembro del componente. Además, para que el coche no acelere hasta el infinito pondremos un límite de velocidad, cuyo valor puede establecerse en 250. Este valor también lo publicaremos, con el objetivo de que se pueda modificar.

```
void UCarMovementComponent::UpdateVelocity(FVector& _vVelocity_, const FVector&
_vAcel, float DeltaTime)
{
  _vVelocity_ += _vAcel * DeltaTime;
  if (_vVelocity_.Size() > m_fMaxVelocity)
    _vVelocity_ *= m_fMaxVelocity / _vVelocity_.Size();
}
```

Por último deberemos actualizar la posición del actor en función de la velocidad calculada en el paso anterior. Para ello modificamos la transformación del actor, añadiéndole la nueva posición calculada con la velocidad.

```
void UCarMovementComponent::MoveActor(const FVector& _vVelocity, float DeltaTime)
{
  auto oTrans = GetOwner()->GetActorTransform();
  oTrans.AddToTranslation(_vVelocity * DeltaTime);
  FVector PlayerLoc = GetOwner()->GetActorLocation();
  GetOwner()->SetActorTransform(oTrans);
}
```

Si hemos realizado cada una de estos cálculos en un método distinto (como debiera ser) ahora es el momento de llamarlos en orden desde la función TickComponent del componente.

```
void UCarMovementComponent::TickComponent(float DeltaTime, ELevelTick TickType,
FActorComponentTickFunction* ThisTickFunction)
{
  Super::TickComponent(DeltaTime, TickType, ThisTickFunction);
  FVector vAccel = CalculateAceleration();
  UpdateVelocity(m_vVelocity, vAccel, DeltaTime);
  MoveActor(m_vVelocity, DeltaTime);
}
```

Ya tendríamos nuestro componente de movimiento listo, que mueve al actor por el mapa. Sin embargo dicho componente no pertenece a nuestro coche todavía. Por ello vamos a declarar un puntero miembro de nuestro componente en la clase ACar y vamos a crear el componente en su constructor (igual que se hace con la malla estática).

```
class UCarMovementComponent;

class CARS_API ACar : public APawn
{
protected:
  UCarMovementComponent* m_pCarMovement;
};
```

```
#include "CarMovementComponent.h"

ACar::ACar()
```

```
{
  m_pCarMovement =
CreateDefaultSubobject<UCarMovementComponent>(TEXT("CarMovement"));
}
```

Si ejecutamos vemos que el coche no se mueve… esto es porque no estamos actualizando el input del componente. Lo hacemos en el Tick, pasando el vector que se crea a partir de los ejes.

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

Si ejecutamos ahora si que hay movimiento del coche, aunque aún hay un par de problemas. El primero se debe a que podemos mover el coche por el circuito pero éste está siempre en la misma dirección, mirando a la parte superior de la pantalla. Para solventar este problema, al final de la función que mueve al coche, hacemos que el coche se oriente siempre con el vector de la velocidad. Haciendo algo parecido a

```
void UCarMovementComponent::MoveActor(const FVector& _vVelocity, float DeltaTime)
{
  if (_vVelocity != FVector::ZeroVector)
  {
    FRotator Rot = FRotationMatrix::MakeFromX(_vVelocity).Rotator();
    GetOwner()->SetActorRotation(Rot);
  }
}
```

Si ejecutamos ahora la cosa tiene mucha mejor pinta. Sin embargo el coche flickea y al acelerar puede salir hacia arriba o hacia abajo. Esto es debido a que tal y como hemos implementado el movimiento (con aceleraciones) cuando el valor de la velocidad es muy cercano a cero su velocidad cambia de positiva a negativa en cada tick. Esto se puede solucionar de diversas formas, pero lo que vamos a hacer nosotros es comprobar si en algún momento el vector director de la velocidad que almacenamos es radicalmente opuesto al sentido de la marcha. Para esto usamos el producto escalar o dot product entre la velocidad calculada (justo después de calcularla) y el ActorForwardVector. Si el resultado es menor de cero es que la velocidad ha cambiado de signo y, como no queremos ir hacia atrás, ponemos la velocidad a cero.

```
void UCarMovementComponent::UpdateVelocity(FVector& _vVelocity_, const FVector&
_vAcel, float DeltaTime)
{
  _vVelocity_ += _vAcel * DeltaTime;

  if (_vVelocity_ != FVector::ZeroVector &&
    FVector::DotProduct(_vVelocity_, GetOwner()->GetActorForwardVector())) < 0.f)
    _vVelocity_ = FVector::ZeroVector;

  else if (_vVelocity_.Size() > m_fMaxVelocity)
    _vVelocity_ *= m_fMaxVelocity / _vVelocity_.Size();
}
```

GameCamera

Una vez que tenemos el coche en movimiento lo siguiente es hacer que la cámara siga al coche por el circuito. Ya hicimos al comenzó que nuestra cámara fuera el view target por lo que lo

único que debemos hacer es actualizar su posición siguiendo al coche. Para ello partiremos de la posición del coche y modificaremos su Z, para colocarla a mayor altura.

Por tanto, en primer lugar, en `ACar::BeginPlay()` cogemos el view target del player controller, lo casteamos a `AGameCamera` y establecemos el coche como objetivo (Guardamos un puntero `ACar` en `AGameCamera`).

```
#include "GameCamera.h"

float ACar::GetVelocityMagnitude()
{
  return m_pCarMovement->GetVelocityMagnitude();
}

void ACar::BeginPlay()
{
  APlayerController* PlayerController =
UGameplayStatics::GetPlayerController(this, 0);
  if (OurPlayerController)
  {
    AGameCamera* _pCtrl = Cast<AGameCamera>(PlayerController->GetViewTarget());
    if (_pCtrl)
    {
      _pCtrl->SetTarget(this);
    }
  }
}
```

Con esto, más establecer la posición de la cámara en `AGameCamera::Tick`, tenemos una cámara que seguiría al coche. Sin embargo queremos una cámara un poco más compleja que, cuando el coche vaya a mayor velocidad, muestre más cantidad del circuito. Para ello haremos que la Z crezca en función de la velocidad del coche, partiendo de una distancia mínima (200) que se incrementará en función de la velocidad, multiplicando por un factor (0.6).

$$z = z_0 + v_{magnitude} * f$$

```
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);
  }
}
```

```
#include "GameCamera.h"

float ACar::GetVelocityMagnitude()
{
  return m_pCarMovement->GetVelocityMagnitude();
}
```

CheckPoints & FinishLine

Las últimas características que vamos a añadir al juego son las relacionadas con las vueltas. Queremos llevar el control de los pasos por línea de meta, controlando que las vueltas hayan sido válidas, y contabilizar el número de vueltas que lleva el coche dadas al circuito, mostrando al paso por meta el tiempo de vuelta y si se ha conseguido mejorar la marca actual.

Para contabilizar que las vueltas son correctas tenemos distribuidos por el circuito un número de triggers `ACheckPoint` que, cuando son atravesados comunicarán a la línea de meta que han sido pasados (para que en el paso por meta se tenga en cuenta). En `ACheckPoint` debemos guardar un puntero a `AFinishLine` para comunicarle cuando el coche lo atraviesa.

Para ser invocados cuando un actor se superpone con el trigger debemos registrar una función para que se nos avise de que se ha producido dicho evento. Luego, en dicha función se avisará a la línea de meta de que se ha sobrepasado este punto del circuito.

```
ACheckPoint::ACheckPoint()
{
  //Register Events
  OnActorBeginOverlap.AddDynamic(this, &ACheckPoint::OnOverlapBegin);
}
```

También es interesante tener una representación visual durante la partida de estos triggers por lo que los pintaremos con líneas de debug

```
void ACheckPoint::BeginPlay()
{
  Super::BeginPlay();
  DrawDebugBox(GetWorld(), GetActorLocation(),
GetComponentsBoundingBox().GetExtent(), FColor::Purple, true, -1, 0, 5);
}
```

```
#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"

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);
  }
}
```

La entidad `AFinishLine` que está puesta en el nivel es también un trigger (que pintaremos en otro color con DrawDebugBox) y que tiene ya un listado con todos los `ACheckPoint` del circuito. En `AFinishLine::BeginPlay()` deberíamos recorrer todos los `ACheckPoint` y establecer cuál es su `AFinishLine` para que puedan informar cuando son superados.

En esta entidad llevaremos la cuenta de las vueltas dadas, el último tiempo de vuelta y el mejor tiempo. Todo esto deberían ser atributos de la clase, en la cual deberíamos guardar el tiempo de la aplicación en el último paso por meta, para poder calcular los tiempos de vuelta la próxima vez que se cruce la meta.

Además, para saber si se trata de una vuelta válida, deberemos tener una lista temporal con los `ACheckPoint` que se han atravesado desde el último paso por meta. De manera que si, al paso por meta, el número de `ACheckPoint` superados es igual al número de registrados quiere decir que la vuelta es válida y se deben registrar todas las marcas.

Para mostrar las información de momento nos servirá con una llamada a `GEngine->AddOnScreenDebugMessage()` con el mensaje deseado.

```
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 "DrawDebugHelpers.h"
#include "CheckPoint.h"
#include "Engine/World.h"

AFinishLine::AFinishLine()
{
  //Register Events
  OnActorBeginOverlap.AddDynamic(this, &AFinishLine::OnOverlapBegin);
}

// Called when the game starts or when spawned
void AFinishLine::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();
    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)
      {
```

```
        GEngine->AddOnScreenDebugMessage(-1, 5.0f, FColor::Red, *FString("Invalid
lap!"));
      }
    }
    m_vPassedCheckPoints.Reset();
  }
}
```