Controlador de Tráfico Urbano en Unreal Engine 4 con C++

Villar Ramírez, Jesús Fermín

ESNE, Escuela Universitaria de Diseño, Innovación y Tecnología

Code available in: https://github.com/Pokoi/City-Simulator

Abstract

En la mayoría de los videojuegos en un entorno urbano se necesitan controladores de tráfico para administrar el flujo de NPCs por la escena, tanto de tráfico rodado como de peatones. En este proyecto se propone una implementación en Unreal Engine 4 con C++, haciendo uso de sensores trigger y de la estructura de datos TArray.

1. Introducción

El tráfico supone un elemento muy importante en todos aquellos videojuegos cuya acción se desarrolle en una ciudad o en cualquier tipo de población y se quiera dar un aspecto vivo a la misma. Es un elemento vistoso, que genera un ambiente y define, del mismo modo, un universo. ¿En la ciudad de nuestro videojuego hay transporte público? ¿Los coches conducen por la izquierda o por la derecha? ¿Hay conducción temeraria? Controlar el tráfico de una ciudad supone controlar cómo se relacionan los que viven en ella y, por tanto, influye en cómo son los personajes que estamos mostrando en el videojuego.

Grandes titulos tienen gestiones del tráfico que nos vienen fácilmente a la mente como puede ser *GTA V*[1]. Pero hay otro ejemplo bastante importante en cuanto a gestión del tráfico: *Cities: Skylines*[2]**.**

En este titulo la gestión del tráfico se realiza mediante la generación de nodos en las carreteras que va poniendo el usuario, de manera que cada caerretera tiene un nodo inicial y un nodo final con los que los vehículos pueden mapear y generar una trayectoria. Del mismo modo, los peatones siguen una ruta que generan y recalculan en función de este entramado de vías. **[0]**

Con fin de simular un comportamiento parecido, en este proyecto se plantea el funcionamiento de unos controladores de tráfico por medio de semáforos que redistribuyen el tráfico. A esperas de incluir un sistema de planificación de rutas, los peatones y los autos de este proyecto se desplazan siguiendo una ruta rectilínea preestablecida.

2. Métodos

A la hora de establecer los elementos y funcionalidades básicas de este ejemplo tenemos: los objetos móviles (vehículos y peatones), los semáforos y los límites del mapa o de las carreteras. Estos elementos configuran los siguientes comportamientos:

- Movimiento de los actores
- Regulación del tráfico en los semáforos
- Generación de los peatones

2.1 Movimiento de los actores

Propiedades y funciones en la clase base

Todos los actores móviles heredan de la clase base MovableActor. Esta clase proporciona las funcionalidades propias, comunes y elementales de estos actores. Las relativas al movimiento de los actores son:

- La **velocidad** del actor **[1]** ya que todos los actores se desplazan a una velocidad constante.
- El margen de **aleatoriedad** de la velocidad del actor **[2]**, necesario para calcular la velocidad aleatoria que tendrá cada uno de los actores.
- El **estado de movimiento** **[3]** que permite o inhibe el movimiento del actor.
- El vector de **dirección** del movimiento **[4]** que controla hacia dónde se desplaza el actor.
- Un **sensor** delantero de proximidad **[5]** que permite detectar cuándo un actor se aproxima frontalmente a otro, para evitar colisiones.
- El **estado** de activación **del sensor** **[6]**.
- El **índice de contacto** del sensor delantero **[7]**
- **Actualizar la dirección** mediente el vector director del movimiento **[8]** que obtiene el vector frontal (forward) del actor siempre que este recibe un cambio en su rotación.
- **Rotar al actor** **[9]** que permite al actor realizar una rotación de 180º y orientarse en sentido contrario.
- **Teletransportar** a una posición dada **[10]** trasladando al actor a esa nueva posición en coordenadas de mundo.
- **Parar** al actor **[11]** que actualiza el estado del movimiento inhibiendo el desplazamiento.
- **Mover** al actor **[12]** que actualiza el estado del movimiento permitiendo el desplazamiento en el instante en el que tiene que empezar a moverse.
- **Detección de la entrada** en el sensor delantero **[13].**
- **Detección de la salida** del sensor de proximidad delantero **[14].**

Propiedades y funciones en las clases hijas

La clase hija de los vehículos (Car) solo posee propiedades relativas al aspecto visual del actor (la malla estática, las luces

1 Dan Houser (2013), Grand Theft Auto V
2 Colossal Order (2015), Cities: Skylines

delanteras y traseras...). Sin embargo, la clase de los peatones sí posee unos campos específicos:

- El factor de **modificación de la velocidad** [15] .
- La **aceleración** del actor [16].
- La **restauración** del actor [17].

Este modelo de datos expuesto permite:

Movimiento rectilíneo de los actores

El desplazamiento de los actores se realiza en cada frame siguiendo un **movimiento rectilíneo uniforme** siguiendo la expresión *posición = velocidad * tiempo.*

```
void AMovableActor::Tick(float DeltaTime)
{
	Super::Tick(DeltaTime);

	if(movable && !sensor_activated) SetActorLocation(GetActorLocation() + movement_direction * DeltaTime * current_speed);
}
```

Fragmento del archivo MovableActor.cpp donde se implementa su método Tick, ejecutado en cada frame del juego.

El desplazamiento viene permitido por el estado *movable* y por el estado del sensor delantero.

Establecimiento de la velocidad

La velocidad de desplazamiento de los actores es aleatoria dentro del margen de aleatoriedad que poseen estos objetos. Así, en el primer frame del juego (o en el primer frame desde que son instanciados), adquieren un **valor aleatorio** en función del valor base y dicho margen de la siguiente forma:

```
current_speed += FMath::RandRange(-speed_random_offset, speed_random_offset);
```

Fragmento del archivo MovableActor.cpp donde se establece el valor aleatorio de la velocidad en su método BeginPlay().

Ya que *current_speed* y *speed_random_offset* son propiedades editables, el diseñador de juego puede modificarlas desde el propio editor.

Cambio de dirección y posición

Cuando los actores llegan al límite de su vía, estos entran en colisión con un actor de la clase Wall. Este actor recibe la colisión [18] por medio de una subscripción al evento de superposición de su componente colisionador [19] en el método BeginPlay() de su misma clase.

Al detectar dicha colisión, el actor Wall hará que el objeto móvil rote y se traslade a su posición de salida [20] situada en el carril opuesto de la calle.

Ya que esta posición de salida es una propiedad, desde el propio editor se puede modificar su posición para ajustarla al lugar adecuado.

Distancia de seguridad

Para evitar accidentes de tráfico, solapamientos entre los vehículos en los cruces y para evitar usar cuerpos rígidos que simulen colisiones avanzadas, el proyecto cuenta con un sistema de sensores de proximidad [5] que regulan esta funcionalidad.

De esta forma, el actor detecta [13] otro móvil gracias a su

```
void AWall::TriggerEnter(class UPrimitiveComponent * HitComp, class AActor * OtherActor, class UPrimitiveComponent * OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult & SweepResult)
{
	if (OtherActor != nullptr && (OtherActor != this) && OtherComp != nullptr)
	{
		AMovableActor* movable = Cast<AMovableActor>(OtherActor);
		if (movable != nullptr)
		{
			movable->ChangeDirection();
			movable->ChangePosition({ exit_position.X + FMath::RandRange(-100.0f, 100.0f) , exit_position.Y, movable-> GetActorLocation().Z});
		}
	}
}
```

Fragmento del archivo Wall.cpp donde se implementa su método TriggerEnter, llamado tras el evento de sobreposición de su componente colisionador.

sensor, lo que activa un mecanismo que inhibe el movimiento del actor, que solo es reaunudado cuando el sensor deja de estar activo al detectar la salida [14] de dicha área de seguridad.

```
void AMovableActor::FrontSensorTriggerEnter(class UPrimitiveComponent * HitComp, class AActor * OtherActor, class UPrimitiveComponent * OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult & SweepResult)
{
	if (OtherActor != nullptr && (OtherActor != this) && OtherComp != nullptr)
	{
		AMovableActor* movable = Cast<AMovableActor>(OtherActor);
		if (movable != nullptr)
		{
			sensor_activated = true;
			sensor_contacts++;
		}
	}
}


void AMovableActor::FrontSensorTriggerExit(class UPrimitiveComponent * HitComp, class AActor * OtherActor, class UPrimitiveComponent * OtherComp, int32 OtherBodyIndex)
{
	if (OtherActor != nullptr && (OtherActor != this) && OtherComp != nullptr)
	{
		AMovableActor* movable = Cast<AMovableActor>(OtherActor);
		if (movable != nullptr)
		{
			sensor_contacts--;
			if (sensor_contacts==0) sensor_activated = false;
		}
	}
}
```

Fragmento del archivo MovableActor.cpp donde se implementa los métodos que controlan el funcionamiento del sensor delantero de los móviles.

2.2 Regulación del tráfico en los semáforos

Propiedades y funciones de los semáforos

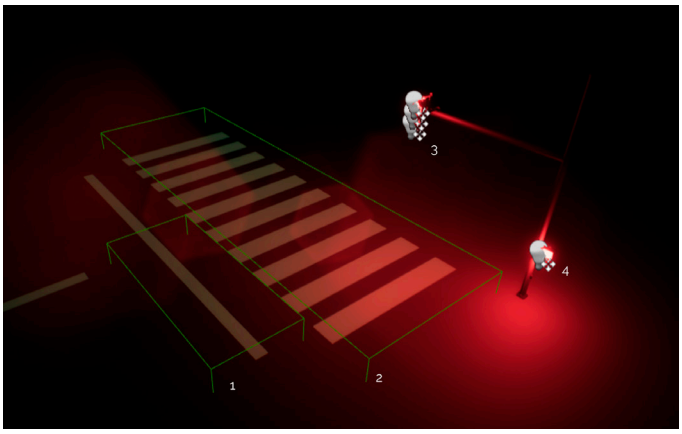
Los semáforos pertenecen a la clase TrafficLight, que contiene todos los elementos necesarios para el funcionamiento de la regulación del tráfico. Estos son:

- **Detector de los coches** [21] , para recibir la información sobre los coches que esperan a pasar el semáforo.
- **Detector de los peatones** [22], para recibir la información sobre los peatones que esperan a cruzar el semáforo.
- **Luces del semáforo** para el tráfico rodado (luz verde [23], luz ámbar [24] y luz roja [25]), como representación

gráfica del flujo del tráfico en el semáforo.
- **Luz para los peatones** [26] , una única luz para representar el permiso o no de flujo peatonal por el paso de cebra.
- **Permiso de tráfico rodado** [27]
- **Permiso de tráfico peatonal** [28]
- Colección de **coches que esperan** a cruzar [29]
- Colección de **peatones que esperan** a cruzar [30]
- **Detección de la entrada de coches** en el área de influencia del semáforo [31] .
- **Detección de la salida de coches** en el área de influencia del semáforo [32] .
- **Detección de la entrada de peatones** en el área de influencia del semáforo [33] .
- **Detección de la salida de peatones** en el área de influencia del semáforo [34] .
- **Encendido** de cada una de las luces del semáforo (de la luz verde [35] , de la luz ámbar [36] , de la luz roja [37] , de la luz peatonal [38]).
- **Apagado** de cada una de las luces del semáforo (de la luz verde [39] , de la luz ámbar [40] , de la luz roja [41] , de la luz peatonal [42]).

Estructura del semáforo

Los semáforos, por tanto, poseen dos detectores o colisionadores: uno para los peatones y otro para el tráfico rodado; y cuatro luces para la representación gráfica del flujo: tres para los vehículos y una para los peatones.



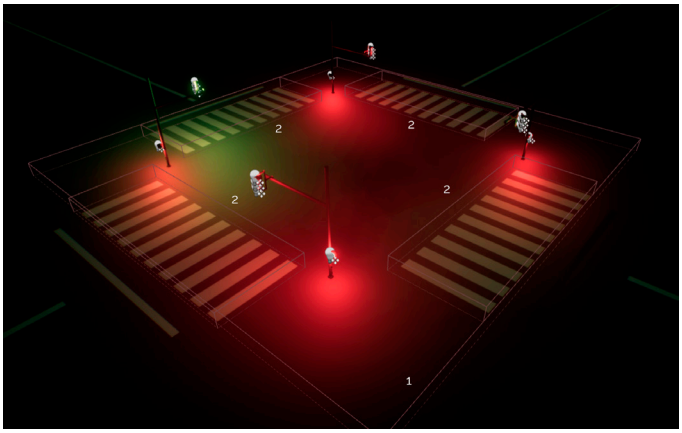
Estructura del semáforo: 1. Detector de los vehículos; 2. Detector de los peatones, 3. Luces para los vehículos; 4. Luz para los peatones

Propiedades y funciones de los cruces

Los semáforos conforman cruces, que controlan y manejan y el flujo general del tráfico en los cuatro sentidos en el cruce. Así, las propiedades y funcionalidades del controlador del cruce son:
- **Duración en verde** [43] , que maneja la cantidad de segundos que ha de pasar el semáforo encendido en verde para el tráfico rodado.
- **Duración en ámbar** [44] , que maneja la cantidad de segundos que ha de pasar el semáforo encendido en ámbar para el tráfico rodado.
- **Detector de los coches** dentro del cruce [45]
- **Detector de los peatones** dentro del cruce [46]
- Colección de los **semáforos del cruce** [47]
- Colección de los **coches cruzando** [48]
- Colección de los **peatones cruzando** [49]
- **Detección de la entrada de coches** en el cruce [50]
- **Detección de la entrada de peatones** en el cruce [51]
- **Detección de la salida de coches** del cruce [52]
- **Detección de la salida de coches** del cruce [53]
- **Detección del vaciado** del cruce [54]

Estructura del cruce

Los cruces están formados por cuatro semáforos, los cuales manejan el tráfico de cada una de las calles que confluyen en el cruce.



Estructura del cruce: 1. Detector de los vehículos; 2. Detector de los peatones

Temporizador de los semáforos

Como en la vida real, los semáforos tienen un temporizador que rige el tiempo que estos permanencen en verde (y en rojo de la misma forma). Esto, controlado por la propiedad editable [43] del CrossManager, administra cuándo es necesario cambiar el flujo (es decir, inhibir o permitir el flujo en semáforos opuestos).
La duración de la luz ámbar [44] sirve para poner en ámbar los semáforos en el momento en el que se precisa.
El cambio de flujo consta de dos fases: una fase inicial en la que inhibe la entrada de nuevos móviles al cruce, poniendo los semáforos en rojo (tanto de tráfico rodado como peatonal), y una segunda fase en la cual se encienden las luces verdes de los semáforos correspondientes. Esta segunda fase solo se da una vez que no haya ningún actor móvil en el cruce. Esto es para evitar posibles atropellos o accidentes en el cruce. Además, una vez se pone en rojo el semáforo, el controlador del cruce se encarga de apresurar a los peatones que todavía estén en él para que incrementen su velocidad durante el trayecto que les queda en el paso de cebra, restaurándose una vez salen del semáforo.

```
void ACrossManager::Tick(float DeltaTime)
{
	Super::Tick(DeltaTime);

	if (remaining_seconds > 0.0)
	{
		remaining_seconds -= DeltaTime;

		if (remaining_seconds < seconds_remaing_in_orange) TurnOrangeTrafficLight();
	}
	else ChangeFlow();
}
```

Fragmento del archivo CrossManager.cpp donde se implementa el método Tick que maneja el control del temporizador de los semáforos del cruce.

```
void ACrossManager::TurnOrangeTrafficLight()
{
	int initial_index = current_turn == FlowTurn::North_South ? 1 : 0;
	while (initial_index < (cross_traffic_lights.Num()))
	{
	cross_traffic_lights[initial_index]->TurnOnOrangeLight();
	cross_traffic_lights[initial_index]->TurnOffRedLight();
	cross_traffic_lights[initial_index]->TurnOffGreenLight();
	initial_index += 2;
	}
}


void ACrossManager::ChangeFlow()
{
	switch (current_turn)
	{
	  case FlowTurn::North_South :
	    TurnOffTrafficLight(1);
	    TurnOffTrafficLight(3);
	    TurnOffWalkTraffic(0);
	    TurnOffWalkTraffic(2);
	  break;
	  case FlowTurn::Western_East:
	    TurnOffTrafficLight(0);
	    TurnOffTrafficLight(2);
	    TurnOffWalkTraffic(1);
	    TurnOffWalkTraffic(3);
	  break;
	}

	if (EmptyRoad())
	{
	  switch (current_turn)
	  {
	    case FlowTurn::North_South:
	      TurnOnTrafficLight(0);
	      TurnOnTrafficLight(2);
	      TurnOnWalkTraffic(1);
	      TurnOnWalkTraffic(3);
	    break;
	    case FlowTurn::Western_East:
	      TurnOnTrafficLight(1);
	      TurnOnTrafficLight(3);
	      TurnOnWalkTraffic(0);
	      TurnOnWalkTraffic(2);
	    break;
	  }
	  remaining_seconds = maximum_seconds_on;
	  current_turn = current_turn == FlowTurn::North_South ? FlowTurn::Western_East : FlowTurn::North_South;
	}
	else
	{
	  for (AWalker* walker : passing_walkers) walker->HurryUp();
	}
}
```

Fragmento del archivo CrossManager.cpp donde se implementan los métodos que controlan el correcto funcionamiento de los semáforos del cruce y su sincronización.

Consulta sobre el estado del semáforo

Cuando un móvil llega a un semáforo [31] [33], el semáforo evalúa si para el desplazamiento de dicho móvil o si le permite continuar en función del estado del permiso de tráfico [27] [28].

2.3 Generación de los peatones

Propiedades y funciones

La generación de los peatones se realiza mediante la clase StreetSpawn. Esta clase posee todos los miembros necesarios para, al comenzar el juego, crear los peatones de la escena. Esto es gracias a:

- **Área de spawn** [55] en la cual se crearán los actores.
- **Direcciones disponibles** [56] [57] [58] [59] del tráfico de ese área
- **Cantidad** de actores que crear [60]
- **Actor a duplicar** [61]
- **Generación** de actores [62]
- Obtención de una **posición aleatoria** dentro del área en la que se crean los actores [63]

Generación

A la hora de crear los actores, la localización de estos depende de una previa generación de una posición dentro del área del spawner o generador; mientras que la rotación del nuevo actor viene dada por la dirección o direcciones disponibles dentro de ese spawn.
La generación aleatoria se realiza mediante una coordenada aleatoria dentro de los márgenes que tiene el área.

```
void AStreetSpawn::BeginPlay()
{
	//Set extents
	FVector collider_bounds_extent = street_collider ->GetScaledBoxExtent();

	max_x_position = (collider_bounds_extent.X) + street_collider->GetComponentLocation().X;
	max_y_position = (collider_bounds_extent.Y) + street_collider->GetComponentLocation().Y;
	min_x_position = street_collider->GetComponentLocation().X - (collider_bounds_extent.X);
	min_y_position = street_collider->GetComponentLocation().Y - (collider_bounds_extent.Y);
}


FVector AStreetSpawn::GetRandomPositionInsideCollider()
{

	return { FMath::RandRange(min_x_position, max_x_position), FMath::RandRange(min_y_position, max_y_position) ,0 };

}


void AStreetSpawn::spawn_walkers()
{
	walker_prefab->SetActorHiddenInGame(false);
	walker_prefab->SetActorEnableCollision(true);

	for (int i = 0; i < walkers_amount; i++)
	{
	  FRotator rotation = { 0, (north_south_flow) ? ((FMath::RandRange(0.0f, 10.0f) > 5.0f) ? -north_south_yaw : north_south_yaw) : (FMath::RandRange(0.0f, 1.0f) > 0.5f) ? 0 : west_east_yaw, 0};

	  FActorSpawnParameters spawn_info;
	  spawn_info.Template = walker_prefab;
	  GetWorld()->SpawnActor<AWalker>(GetRandomPositionInsideCollider(), rotation - walker_prefab->GetActorRotation(), spawn_info);
	}

	walker_prefab->SetActorHiddenInGame(true);
	walker_prefab->SetActorEnableCollision(false);
}
```

Fragmento del archivo StreetSpawn.cpp donde se implementan los métodos que controlan la generación de los peatones.

3. Resultados

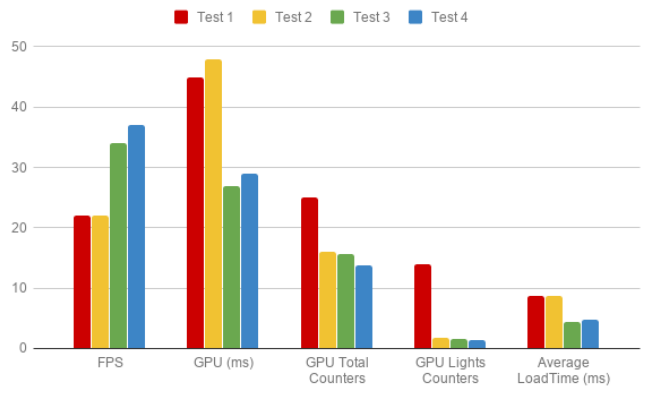
Rendimiento

	Test 1	Test 2	Test 3	Test 4
FPS	22	22	34	37
GPU (ms)	45	48	27	29
GPU Total Counters	25	16	15.6	13.7
GPU Lights Counters	13.99	1.76	1.58	1.36
Average Load Time (ms)	8.7	8.7	4.4	4.69

Tabla con los datos recogidos en las pruebas de rendimiento

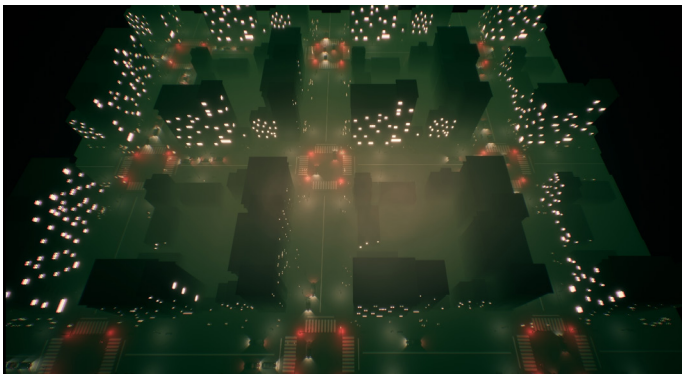
Teniendo en cuenta los siguientes parámetros:
- **LDE** : luces dinámicas y estáticas del escenario activadas.
- **CL**: luces de los vehículos rodados activadas.
- **nC** : cantidad de vehículos rodados en la escena.
- **nW** : cantidad de peatones en la escena.

Los diferentes tests se estructuran de la siguiente forma:
- **Test 1**: LDE + CL + 55C + 768W
- **Test 2**: LDE + 55C + 768 W.
- **Test 3**: LDE + CL + 27C + 384W
- **Test 4**: LDE + 27C + 384W



Gráfica con los datos recogidos en las pruebas de rendimiento

Apariencia



4. Discusión

Las pruebas de rendimiento ejan ver que las caídas de FPS se deben al gasto gráfico de las luces dinámicas que posee cada coche (ya que cada uno de los coches tiene cuatro luces dinámicas). Por ello aumenta a casi el doble la performance al reducir a la mitad la cantidad de actores de la escena.
A la hora de desarrollar en C++ para Unreal Engine hay que tener especial cuidado con respecto a algunas de las herramientas que se han usado a lo largo de este desarrollo, como por ejemplo:

- A la hora de **crear un componente**, este método recibe un string que, si bien es engañoso y puede dar lugar al equívoco creyendo que se trata de una categoría como las categorías de las Propiedades, en realidad se trata del nombre que tendrá el componente; por lo que no puede haber dos componentes que reciban el mismo string o crasheará el programa.
- Cuando estamos creando **propiedades accesibles desde blueprint,** no todos los tipos de datos están soportados. Por ejemplo, el tipo "double" da error ya que no está soportado por los blueprints.
- Todo actor que tenga **más de un colisionador** dará problemas si esos colisionadores están en jerarquía entre sí. Para solucionarlo se deben colocar en el mismo nivel de jerarquía.
- En Unreal las **rotaciones** se guardan bajo el tipo de dato FRotator. El constructor de este tipo de dato posee una sobrecarga que recibe tres parámetros (la rotación en cada uno de los ejes), pero es engañoso ya que no los recibe en orden convencional x,y,z. FRotator recibe como primer parámetro la rotación en el eje Y, como segundo parámetro la rotación en el eje Z, y como tercer parámetro la rotación en el eje X.

Referencias

[0] Antti Lehto, Damien Morello, Karoliina Korppoo (Marzo 2015) *Game Design Deep Dive: Traffic systems in Cities:Skylines*, Gamasutra: http://www.gamasutra.com/view/news/239534/Game_Design_Deep_Dive_Traffic_systems_in_Cities_Skylines.php
[1] float AMovableActor::current_speed
[2] float AMovableActor::speed_random_offset
[3] bool AMovableActor::movable
[4] FVector AMovableActor::movement_direction
[5] UBoxComponent* AMovableActor::front_proximity_sensor
[6] bool AMovableActor::sensor_activated
[7] int AMovableActor::sensor_contacts
[8] void AMovableActor::UpdateMovementDirectionVector()
[9] void AMovableActor::ChangeDirection()
[10] void AMovableActor::ChangePosition(FVector new_position)
[11] void AMovableActor::Stop()
[12] void AMovableActor::Move()
[13] void AMovableActor::FrontSensorTriggerEnter(class UPrimitiveComponent* HitComp, class AActor* OtherActor, class UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult & SweepResult)
[14] void AMovableActor::FrontSensorTriggerExit(class UPrimitiveComponent* HitComp, class AActor* OtherActor, class UPrimitiveComponent* OtherComp, int32 OtherBodyIndex)
[15] float AWalker::speed_modificator
[16] void AWalker::HurryUp()
[17] void AWalker::Relax()
[18] void AWall::TriggerEnter(class UPrimitiveComponent* HitComp, class AActor* OtherActor, class UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult & SweepResult)
[19] UBoxComponent* AWall::box_collider
[20] USceneComponent* AWall::exit_position_scene_component
[21] UBoxComponent* ATrafficLight::car_detector_trigger
[22] UBoxComponent* ATrafficLight::walker_detector_trigger
[23] UPointLightComponent* ATrafficLight::green_light
[24] UPointLightComponent* ATrafficLight::orange_light
[25] UPointLightComponent* ATrafficLight::red_light
[26] UPointLightComponent* ATrafficLight::walker_light
[27] bool ATrafficLight::cars_can_pass
[28] bool ATrafficLight::walkers_can_pass
[29] TArray<ACar*> ATrafficLight::waiting_cars
[30] TArray<AWalker*> ATrafficLight::waiting_walkers
[31] void ATrafficLight::CarTriggerEnter(class UPrimitiveComponent* HitComp, class AActor* OtherActor, class UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult & SweepResult)
[32] void ATrafficLight::CarTriggerExit(class UPrimitiveComponent* HitComp, class AActor* OtherActor, class UPrimitiveComponent* OtherComp, int32 OtherBodyIndex)
[33] void ATrafficLight::WalkTriggerEnter(class UPrimitiveComponent* HitComp, class AActor* OtherActor, class UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult & SweepResult)
[34] void ATrafficLight::WalkTriggerExit(class UPrimitiveComponent* HitComp, class AActor* OtherActor, class UPrimitiveComponent* OtherComp, int32 OtherBodyIndex)
[35] void ATrafficLight::TurnOnGreenLight()
[36] void ATrafficLight::TurnOnOrangeLight()
[37] void ATrafficLight::TurnOnRedLight()
[38] void ATrafficLight::TurnOnWalkerLight()
[39] void ATrafficLight::TurnOffGreenLight()
[40] void ATrafficLight::TurnOffOrangeLight()
[41] void ATrafficLight::TurnOffRedLight()
[42] void ATrafficLight::TurnOffWalkerLight()
[43] float ACrossManager::maximum_seconds_on
[44] float ACrossManager::seconds_remaing_in_orange
[45] UBoxComponent* ACrossManager::car_box_collider
[46] UBoxComponent* ACrossManager::crosswalk_collider
[47] TArray<ATrafficLight*> ACrossManager::cross_traffic_lights
[48] TArray<ACar*> ACrossManager::passing_cars
[49] TArray<ACar*> ACrossManager::passing_walkers
[50] void ACrossManager::CarTriggerEnter(class UPrimitiveComponent* HitComp, class AActor* OtherActor, class UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult & SweepResult)
[51] void ACrossManager::CrossWalkTriggerEnter(class UPrimitiveComponent* HitComp, class AActor* OtherActor, class UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult & SweepResult)
[52] void CarTriggerExit(class UPrimitiveComponent* HitComp, class AActor* OtherActor, class UPrimitiveComponent* OtherComp, int32 OtherBodyIndex)
[53] void ACrossManagert::CrossWalkTriggerExit(class UPrimitiveComponent* HitComp, class AActor* OtherActor, class UPrimitiveComponent* OtherComp, int32 OtherBodyIndex);
[54] bool ACrossManager::EmptyRoad()
[55] UBoxComponent* AStreetSpawn::street_collider
[56] bool AStreetSpawn::north_south_flow
[57] bool AStreetSpawn::south_north_flow
[58] bool AStreetSpawn::east_west_flow
[59] bool AStreetSpawn::west_east_flow
[60] int AStreetSpawn::walkers_amount
[61] AWalker* AStreetSpawn::walker_prefab
[62] void AStreetSpawn::spawn_walkers()
[63] FVector AStreetSpawn::GetRandomPositionInsideCollider()