DOTS/ECS - TRAFFIC SIMULATOR

By

292645

287549

269901

1. Introduction

Unity’s **Data-Oriented Tech Stack (DOTS)** is a new combination of technologies, which requires a new way of working with Unity in a different approach about code, these technologies work together to provide a data-oriented approach to coding in Unity.
DOTS enables you to take advantage of multicore processors to parallelize data processing and increase the performance of Unity projects.

DOTS consists of the following elements:

- The **Entity Component System (ECS)**, which provides the framework for coding using a Data-Oriented approach. With a data-oriented approach, data structures are organized to avoid cache misses that subsequently make access to your data more efficient and faster.
- The **C# Job System**, which provides a simple method of generating multithreaded code.
- The **Burst Compiler**, which generates highly optimized code that takes advantage of the platform hardware that you’re compiling for.
- The **Native Containers**, which are ECS data structures that provide control over memory.

1.1 Objective of the project

The objective of the project is the development of a Traffic Simulator of a city through the multithread system DOTS. We have built a 3D simulation of traffic that manage the various situations (traffic light, cars lanes change, yield the right of way, ecc.) in real time.

2. Instructions

The player’s camera can be moved over the city with **WASD control-system** , to increase and decrease zoom you can use “Q” and “E” on the keyboard. You can change the camera by clicking the "Change Camera" button on the right, so you can follow a random car, or you can use Eitan with the button “use Eitan” to freely explore the city, in this case you can move the character with WASD control-system.

The buttons “+/- x0.25” on the left side can be used to increase or decrease the timescale of the simulator.

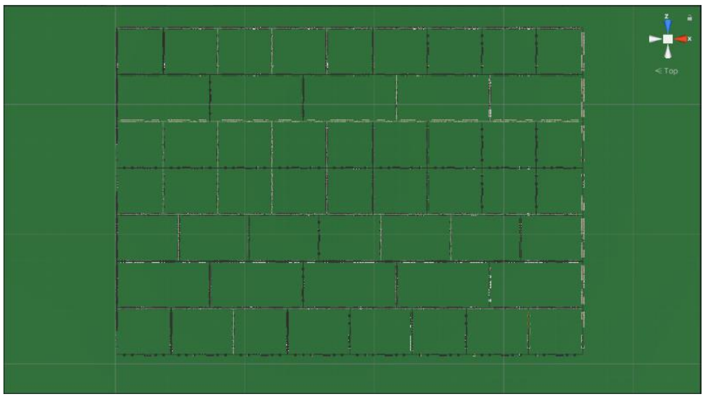
The buttons “Disable Graphic” on the right side can be used to disable the render of the meshes and increase the fps of the simulator (**NOTE: this can be used only with the camera from above**).

The user can configure the following parameters to create a custom infrastructure:

Number of cars to spawn (int): this is the number of cars that will spawn around the city.

Number of buses to spawn (int): this is the number of buses that will spawn at city bus stops.

Number of horizontal streets (int): each street running horizontally is considered as a horizontal street. This setting sets the desired number of such streets to create. Horizontal streets run parallel to each other and are connected by the vertical streets.

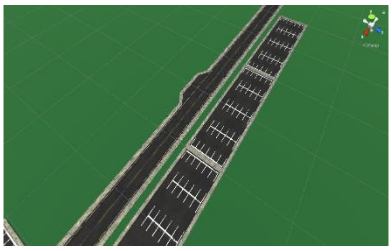
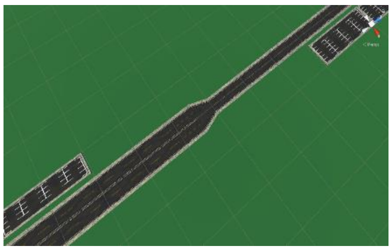


Minimum and maximum number of vertical streets (int): the number of streets that connect two horizontal streets will be a random number of streets between the minimum and the maximum value.

City intersection types (bool): create a city with only simple, only with semaphores or both simple and with semaphore intersections.

Car Prefabs list: this list contains all the different cars that can run on the city. Cars are randomly selected from this list and are spawned on the city as parked cars or at random points in the streets. The user can also insert their own car prefabs, taking care to configure each prefab, as needed, so that the cars are compatible with the different DOTS systems.

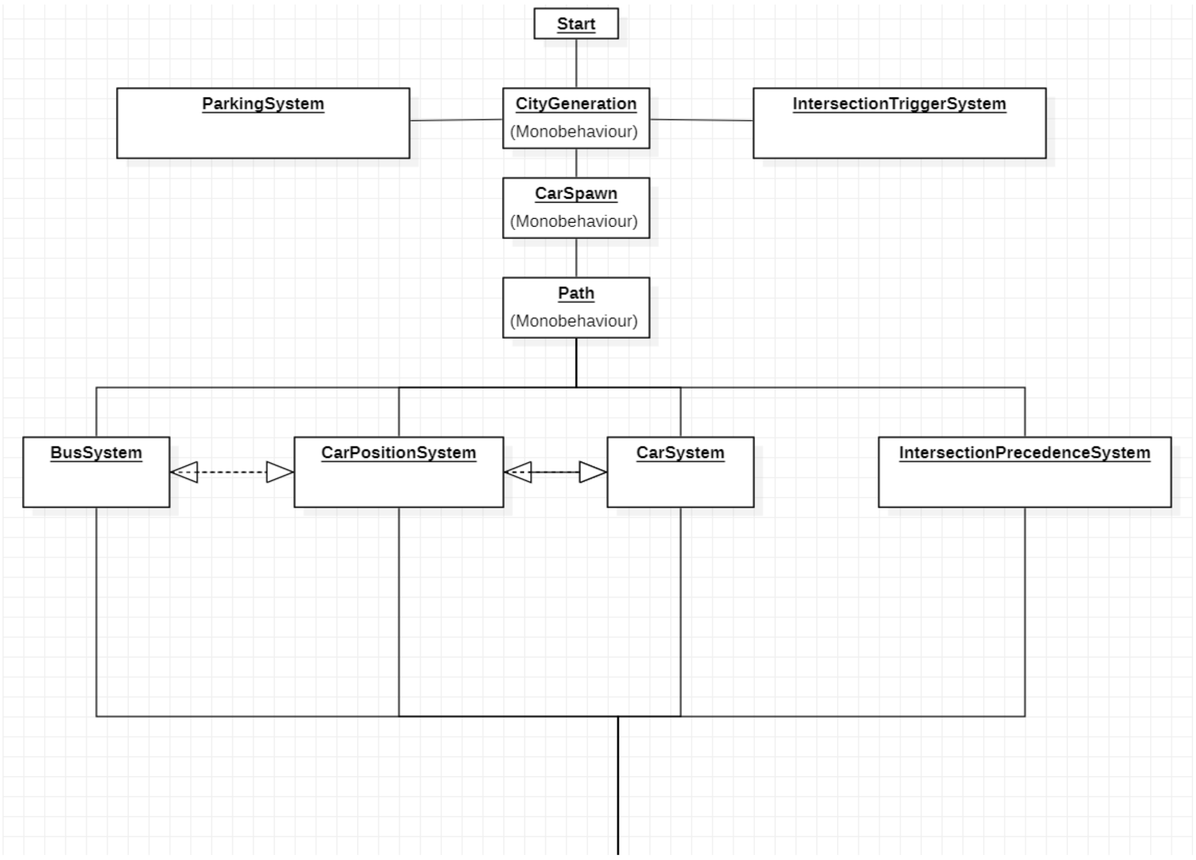
City road types (bool): create a city with only 1 lane, only 2 lane or both 1 and 2 lane roads. 1 lane and 2 lane roads are connected to each other by lane adapters with on one end have a 1 lane road and on the other a 2-lane road.

The data can be passed in two ways: through a JSON file "*./Asset/File/Data.json*" (remember to check the checkbox "Use Data From JSON" from CityGenerator) or configuring the data manually in the *CityGenerator* script on Unity Editor.

3. Features of our application

The simulator is composed by different systems, each of one has implemented a particular function, such as move the cars around the city in a path, avoiding collisions between them, ensure that the cars respect the traffic light, the intersection rules and to enable the cars to enter and exit parking zones. All the system works together as represented in the UML diagram.



1. CarSystem/BusSystem

This system is responsible for the most important functionalities of the simulator. It updates the car movements and their rotation in correspondence to the waypoints placed around the city, it avoids collisions between cars by checking for traffic that may be in the cars' path and it makes the cars respect the traffic lights. This system loops through every car, every frame. The system is implemented to be run in parallel from many workers to ensure maximum performance even with an elevated number of cars in the city. *CarSystem* depends on many components and a *DynamicBuffer* in order to control every car.

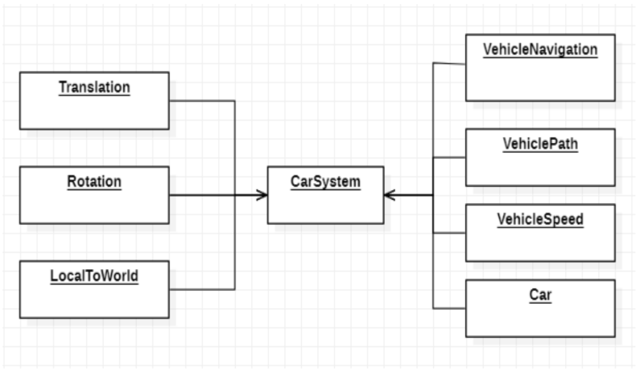


Figure 1 CarSystem components

The most important component is *VehicleNavigation*:

- *currentNode*: the index of the next target node to reach
- *needParking*: flag to show if a car reached its destination (and consequently needs to park) or not
- *isParked*: flag to show whether a car is now parked or not
- *parkingGateWay*: the position of the node where the car will exit
- *intersectionStop*: flag to show if the car has reached an intersection
- *intersectionCrossing*: flag to show if the is now crossing the intersection
- *intersectionCrossed*: flag to show if the has successfully crossed the intersection
- *intersectionId*: the id of the intersection the car is in now
- *intersectionDirection*: direction that the car entered the intersection
- *isSimpleIntersection*: flag to show if the car is in a simple intersection
- *isSemaphoreIntersection*: flag to show if the car is in a semaphore intersection
- *intersectionNumRoads*: type of the intersection the car is in now
- *trafficStop*: flag to show if the car is in traffic
- *timeExitBusStop*: time(sec) after which the bus will exit the bus stop
- *timeExitParking*: time(sec) after which the car will exit the parking
- *isChangingLanes*: flag to show if a car is changing lanes in a 2-lane street
- *isCar*: flag to distinguish a car from a bus
- *isBus*: flag to distinguish a bus from a car

2. CarsPositionSystem

The collision avoidance system is the result of the cooperation of 2 systems: the *CarSystem* and the *CarsPositionSystem*. The *CarsPositionSystem* is basically a system that iterates through every car and inserts their position in a *NativeHashMap* by converting the actual car position into a unique key. The CarsPositionSystem works also as an "intersections trigger", because when a car is on an intersection access node, it will signal the *IntersectionPrecedenceSystem* that this car has enter in an intersection.

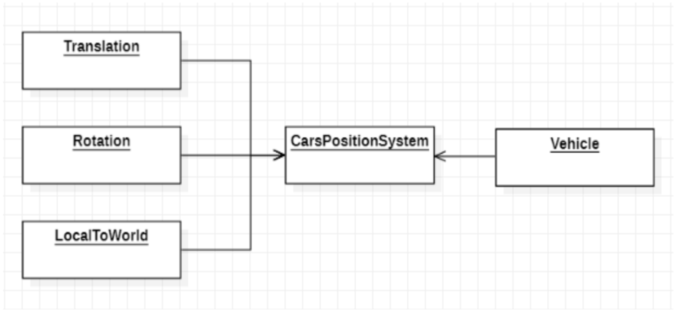
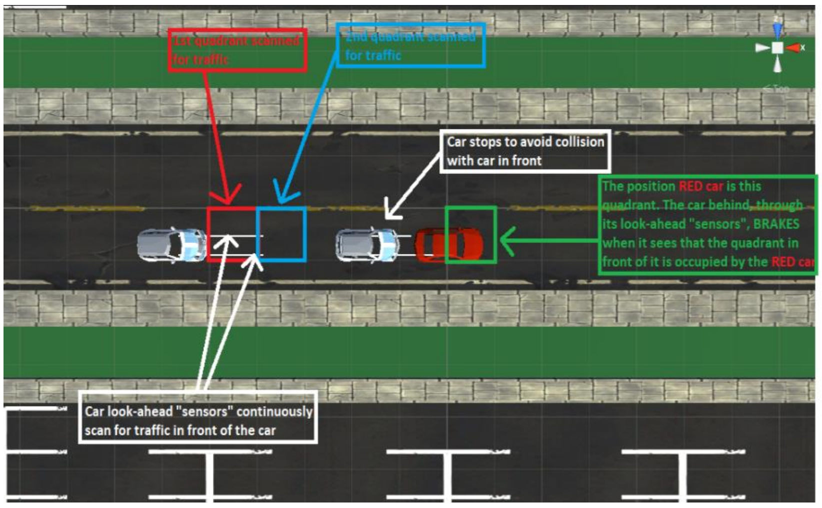


Figure 2 CarPositionSystem Components

This system is an advanced implementation of the **Quadrant System**, but to be more efficient, it divides the map into 1-unit quadrants, in which at most one car can be at a certain moment in time. By doing this, we ensure that the car sensors in the *CarSystem* need only to check 2 units in front of it for traffic, drastically reducing the number of calculations performed compared to the traditional Quadrant System.


1st quadrant scanned for traffic
2nd quadrant scanned for traffic
Car stops to avoid collision with car in front
The position RED car is this quadrant. The car behind, through its look-ahead "sensors", BRAKES when it sees that the quadrant in front of it is occupied by the RED car
Car look-ahead "sensors" continuously scan for traffic in front of the car

3. IntersectionPrecedenceSystem

Every intersection is composed by an intersection Hash Map, *intersectionIdMap*, which act as a notifier, to the *CarSystem* and to the *IntersectionPrecedenceSystem*, to signal that car has entered an intersection and it needs to abide to the intersection rules.
These right of way rules are managed by the *IntersectionPrecedenceSystem*, which iterates through every car that has the *intersectionStop* flag active, checks the situation of the intersection by checking if no other car from another direction is now crossing to the intersection and that the current car has its right side of the intersection free. If one of these checks fails, the car remains stopped in the intersection, until both the conditions are satisfied. To prevent potential intersection deadlocks when all the sides of the intersection are filled with waiting cars, the *IntersectionPrecedenceSystem* will signal cars to start moving from an arbitrary chosen intersection direction.

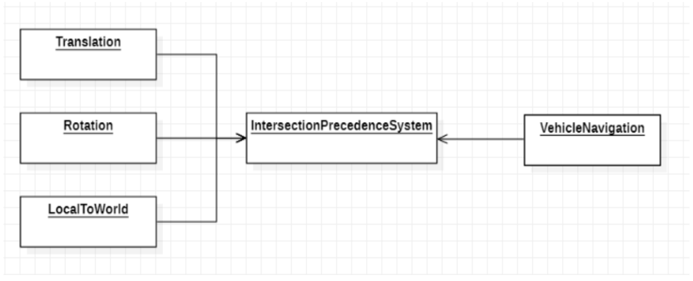


Figure 3 IntersectionPrecedenceSystem Components

4. Traffic light intersections and the CarSystem/BusSystem

Respecting traffic light rules is a duty of the *CarSystem*.
By using a fixed timer for each traffic light in each direction, getting the elapsed time of the simulation and knowing the number of streets of the intersection, the *CarSystem* can calculate the actual direction that is allowed to cross the intersection. Cars that are waiting on an intersection direction excluding the green-lighted one, will remain stopped until their turn comes. To avoid large-scale traffic deadlocks in the city, when their turn comes, cars will cross the intersection only if all the cars from other directions have successfully crossed the intersection.

5. CityGenerator (Mono)

The *CityGenerator* script starts the entire simulation. This script has several jobs to complete before the simulation can start. The city, that is generated, is encircled by a “ring of city”, which is a unique road that goes all around the circumference of the city and is present in most of the modern cities around the world; it includes a different number of buses that run around the city in random pre-planned routes. The city traffic simulator includes the possibility to have a city that contains different types of buildings. The user can configure the building prefabs, a list that contains different prefabs, where the city generator can randomly choose from; and city buildings toggle, by switching this toggle to TRUE, the city generator will automatically generate buildings between every road of the city.

6. CarSpawn / BusSpawner (Mono)

This script is responsible for cars or buses spawner, it sets the car/buses variables and convert it to entities. Each car and bus prefabs have an already set component (CarComponet) that is used by CarSystem.

7. Path (Mono)

This script is using the A-Star algorithm to calculate a path for a car or bus.

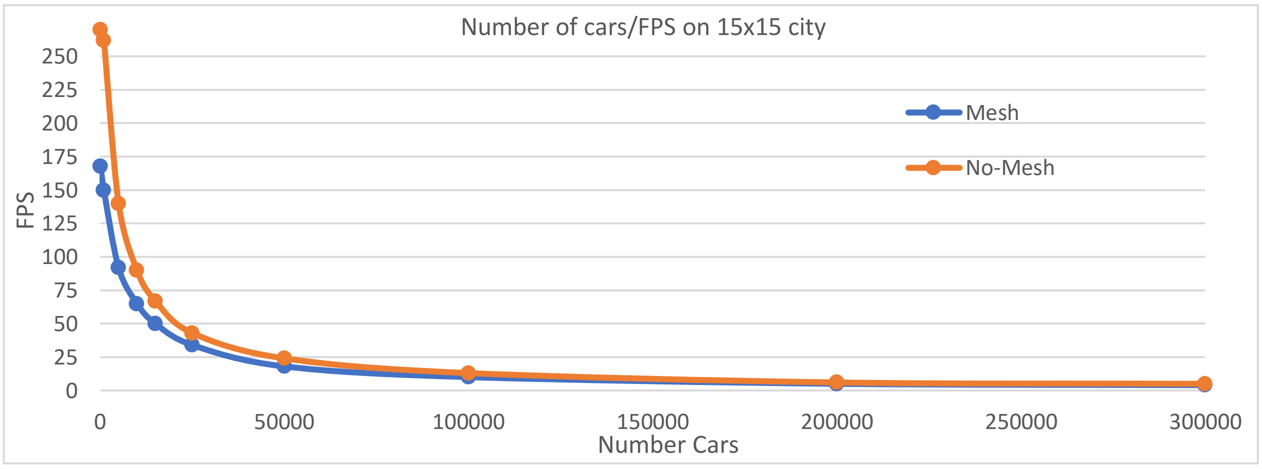
4. Performance

To have a lower density of machines, to avoid the formation of deadlocks, each road can generate a limited number of machines, so you need to increase the size to spawn more cars.

We analyzed the system taking the number of FPS per car when the system is in a situation of mesh rendering of all prefabs, ie when rendermeshv2 is active, and when the system is in a situation of non-rendering mesh (no-mesh). The time is relative to start the simulation.

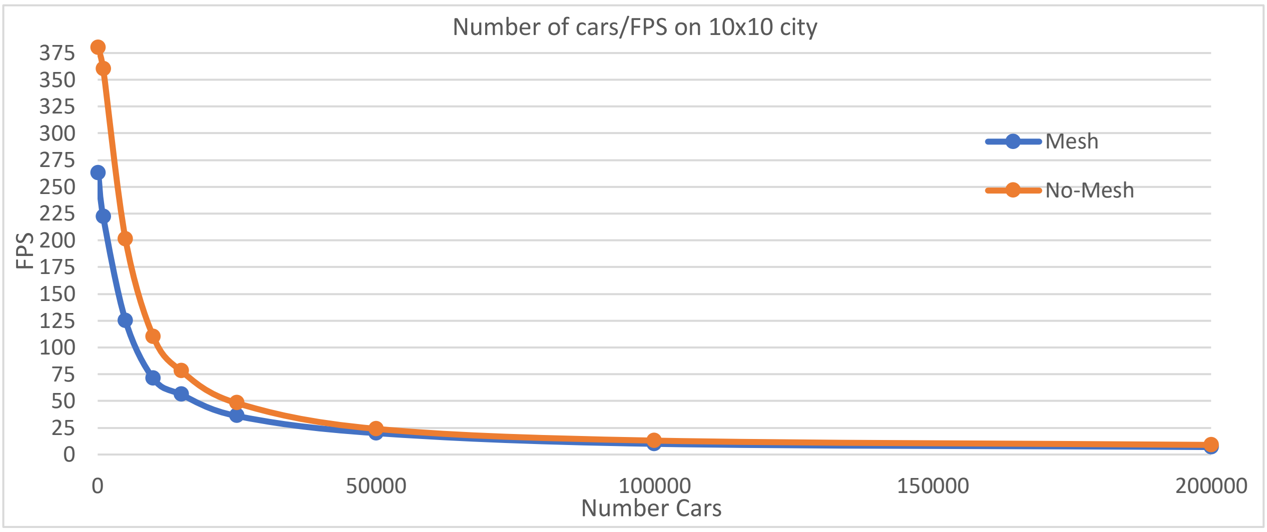
Number of cars	100	1000	5000	10000	15000	25000	50000	100000	200000	300000
Time (mm:ss)	00:26	00:29	00:36	00:39	00:41	00:55	01:19	02:15	04:16	06:32
Mesh (FPS)	168	150	92	65	50	34	18	10	5	4
No-Mesh (FPS)	270	262	140	90	67	43	24	13	6	5

Table 1 Generated with a 15x15 city



Number of cars	100	1000	5000	10000	15000	25000	50000	100000	200000
Time (mm:ss)	00:13	00:15	00:17	00:26	00:30	00:38	01:06	01:57	03:03
Mesh (FPS)	263	222	125	71	56	36	20	10	7
No-Mesh (FPS)	380	360	201	110	78	48	24	13	9

Table 2 Generated with a 10x10 city



The first thing you can notice from the tables is that a larger map increases the time needed to load the simulation, also not using the rendering of the mesh you have better performance, since the system does not render all the meshes of all prefabs for each movement, but only focuses on the logic of implemented systems, taking less time. We can see this in the entity debug.

The following table summarizes the times, collected by Entity Debugger, used by the various systems:

	10x10 city 15000 cars with mesh (ms)	**10x10 city 15000 cars with no-mesh (ms)**	**10 10x10 city 200000 cars with mesh (ms)**	**10x10 city 200000 cars with no-mesh (ms)**	**15x15 city 15000 cars with mesh (ms)**	**15x15 city 15000 cars with no-mesh (ms)**	**15x15 city 300000 cars with mesh (ms)**	**15x15 city 300000 cars with no-mesh (ms)**
CarsPositionSystem	0.09	0.11	0.18	0.20	0.08	0.15	0.28	0.31
CarSystem	0.05	0.04	0.05	0.04	0.03	0.04	0.04	0.04
IntersectionPrecedenceSystem	0.01	0.02	0.02	0.02	0.01	0.02	0.02	0.02
BusSystem	0.04	0.03	0.04	0.03	0.02	0.04	0.03	0.03
RenderMeshSystemV2	11.80	-	116.25	-	11.99	-	286.53	-

In conclusion, it can be noted that DOTS systems such as *CarsPositionSystem*, *CarSystem* and BusSystem, which control entities, are very performant and efficient (<1ms) this because they are in .Schedule() and .ScheduleParallel(), so they perform more calculations in parallel; what really decreases performance is the presence of render meshes.

Another interesting thing is that the time taken by the individual systems remains, more or less, stable below 1ms, which means that the system is scalable, the only limitation is hardware, because to generate a million cars require a lot of memory, both to render all entities and its mesh, trying our system on a machine with less memory, there is a possibility that unity could crash.

All this Tests are performed on a machine with a i7-10750H with 16GB of RAM.