Virtual Reality Car Driving Simulator

USER MANUAL

Anthony Rizk Gustavsson, Filip Granqvist, Jim Bengtsson, Manne Engelke,
Rasmus Lorentzon & Robin Sveningson

Annex to bachelor's thesis
Chalmers University of Technology
DATX02-16-10
2016-05-08

Table of content

1. Hardware requirements

Computer

The computer used should at least have the minimum specs of Unreal Engine 4. You can read about the minimum specs of UE4 here:

https://docs.unrealengine.com/latest/INT/GettingStarted/RecommendedSpecifications/

We recommend using an advanced graphics card. During our development we have used a NVIDIA GeForce GTX 970 which has worked alright.

If you want to use VR with the simulator there are different system requirements for the different VR-HMD:s, the requirements for the Oculus Rift are found here:

https://www.oculus.com/en-us/blog/powering-the-rift/

OSVR doesn't seem to have a specified minimum system requirements.

Virtual Reality (optional)

The simulator is designed to use a VR-headset, but it does work without one. We recommend using Oculus Rift DK2 which is plug and play with UE4. There are other VR-HMD:s that can be used, but they might not be plug and play.

Steering Wheel and pedals (optional)

A steering wheel and pedals is recommended for best simulator experience, but the simulator can be run with the keyboard. We have been using the Thrustmaster Ferrari Racing Wheel Red Legend Edition steering wheel and pedals. With the correct driver it works well with Unreal Engine 4. Any steering wheel can be used as long as it is supported by your computer setup.

The simulator uses the plugin JoystickPlugin in order to find the Wheel and Pedals. If you choose to use another set of pedals and wheel you will have to re-code a small part of the input system.

You will have to create a new set of Axis Mappings for your setup and bind your wheel and pedals to those. Furthermore you will have to map those axis events to the correct output. The value of the Axis Mappings might be bugged (it was for the Thrustmaster wheel) and you will then have to use the Get Previous Joystick State

function in order to retrieve the correct the axes values of your wheel and pedals. Those values then need to transformed to function properly.

The gas pedal should follow the pattern of [0,1] where 1 is maximum throttle and 0 is no throttle at all. The brake pedal should be transformed to follow the pattern of [0,x] where x is greater than 0, 0 is no braking and > 0 is full braking power. The wheel output should follow the pattern of [-1,1] where at -1 the wheel is at its leftmost position and vice versa. The wheel might already follow that pattern.

Note: It's important to identify which axes index corresponds to which pedal or wheel in the Axes array given by the Get Previous Joystick State function; this is easiest done by trial-and-error.

2. Installation

Source code

The code used for the simulator is available at Github:

https://github.com/sveningsonrobin/VRDrivingSimulator

After you have cloned the repository, the project is opened by opening the file "VRDrivingSimulator\MyProject2\MyProject2.uproject" with Unreal Engine 4.

Unreal Engine

Unreal Engine 4 has been used during our development. We have been using UE4 version 4.10.4, but any version that can load the project can be used. Newer or older versions of UE4 might require you to convert the project to another version of UE4.

Visual Studio

Visual Studio c++ 2015 or a later version is required if you want to export the project to an executable application. You can however run the simulator directly from UE4 without having to install VS.

Getting started

When you are building the application yourself, you want to open up the UE4 project to begin with. The simulator can be played directly by pressing *Play*, it is recommended to build the project before running it by hitting the *Build* button. An executable may be created by using the built in function under "File\Package project" in Unreal Engine. On Unreal Engine's website there are a lot of good tutorials for getting started with UE4.

https://docs.unrealengine.com/latest/INT/GettingStarted/index.html

To run the so called Open World, which is the city which allows you to freely drive around, you will have to open up the level *Open World* which is found under "VRDrivingSimulator\Maps\Open_world\". In the folder "VRDrivingSimulator\Maps" you can also find example scenarios. To run the project with the user interface, open up the level *StartingScene* in the folder "VRDrivingSimulator\UI".

3. Performance improvement

This chapter describes things that can be done to increase the performance of the simulator, if you for instance have a low FPS.

General tips

- Decrease the lightning quality in Unreal Engine at “Build\Lightning Quality”.
- Create a new level which contains less actors than Open World (you can copy smaller parts of the Open World to your new level).
- Decrease the amount of AI cars and AI humans.
- Make a flat terrain with a single material.
- Dig into the static/dynamic shadow problem. Read more about the problem in chapter 5.

Other things to try

If the general tips didn’t give you a good enough performance, you can look at removing certain objects in the world that aren’t necessary. Things that don’t serve any purpose in the world, other than creating a realistic look are:

- Nature (trees and bushes)
- Fancy terrain (replace with flat using one material, as suggested in general tips)
- Stores, restaurants and outer diners
- Advertisement (billboards, building ads etc.)
- Park benches and trash cans
- Rails on the highway
- Traffic signs (though collision boxes on speed limits are needed for AI cars to change speed)

Performance analysis

To help you decide which elements might be tactical to remove, we have made performance analyses on the level Open World. The result of the latest analysis, which was performed the 5th of April 2016, is shown below.

Based on the FPS-difference, here is an approximate result:

Roads: *Very heavy*
Buildings: *No difference*
Landscape: *Improves performance*

Light:	*Heavy*
Nature:	*No difference*
AI-nodes:	*Very heavy*

The full analysis can be found in the project's Github repository:

https://github.com/sveningsonrobin/VRDrivingSimulator

4. Directory structure

To make it easier to understand and get to know the project structure we felt that a short explanation of the directory structure was necessary. We worked a bit with finding an appropriate way of structuring different types of objects, such as static meshes, textures, materials, blueprints and combinations of blueprints. Eventually we ended up with the following structure. Note that all directories listed below are placed under our root directory, which is the directory "VRDrivingSimulator/MyProject2/Content/VRDrivingSimulator".

Subdirectory:	**Description:**
AI	Contains blueprints for AI-logic.
Blueprints	Contains blueprints that are supposed to be placed on the map or in a combination with other blueprints. A blueprint is often just a combination of a static mesh and a texture, and in most cases there is no blueprint logic in the object. The reason materials aren't directly placed on static meshes is because many different objects can use the same static mesh, but with different materials. One example is the speed limit traffic signs. All signs use the same static mesh (the circle sign on a pole), but they display different numbers. As you may notice, building pieces aren't to be found in the blueprints directory. They are instead placed in the *finished_components* directory directly for various reasons.
Finished_components	Contains combinations of several blueprints from the *blueprints* directory, or of combinations of blueprints and static meshes. The finished components are also placed on the level. The reason this directory exists is because often you want to combine different blueprints or other items in one parent blueprint. An example of finished components are the road components, which are a combination of a static mesh and material for the road itself, as well as street signs, AI-nodes, railings, streetlights and traffic lights.
Foliage	Contains the foliage assets, which are necessary to place trees and bushes.
GameInstance	Contains files which are used to store data between

	simulation runs, and is required by the GUI.
Landscape_resources	Contains information about how the materials are painted on the terrain.
Maps	Contains the official maps and scenarios that are available to the users.
Materials	Contains materials used in the simulator.
Static_meshes	Contains static meshes.
Test_levels	Contains test levels used by us for developing purposes.
Textures	Contains textures used in the simulator.
UI	Contains the logic used by the user interface.

5. Instructions

5.1. Roads

Roads are created by placing the predefined road pieces found in the *Finished Component* folder in your level. These road pieces have important AI-nodes, traffic signs and railings already placed on the blueprint, which is the difference between them and the ones in the *Blueprint* folder.

When placing road pieces, make sure that the AI-nodes are connected. A connector pin is found at the very end of every road. Simply connect two roads with each other by connecting their pins. A connection of two pins is only allowed if both have the same color.

Figure 1: The three different road connector pins.

Blue = 2 way 2 file road connection.
Pink = 2 way 1 file road connection.
Yellow = 1 way 1 file road connection.

Here are exact instructions on how to connect the AI-nodes on the road pieces with each other.

Step1: Drag out a road component onto the map from the *Finished Components* directory and click on it.

Step2: In the default settings, click on the pipette to the right of the connector you wish to use.

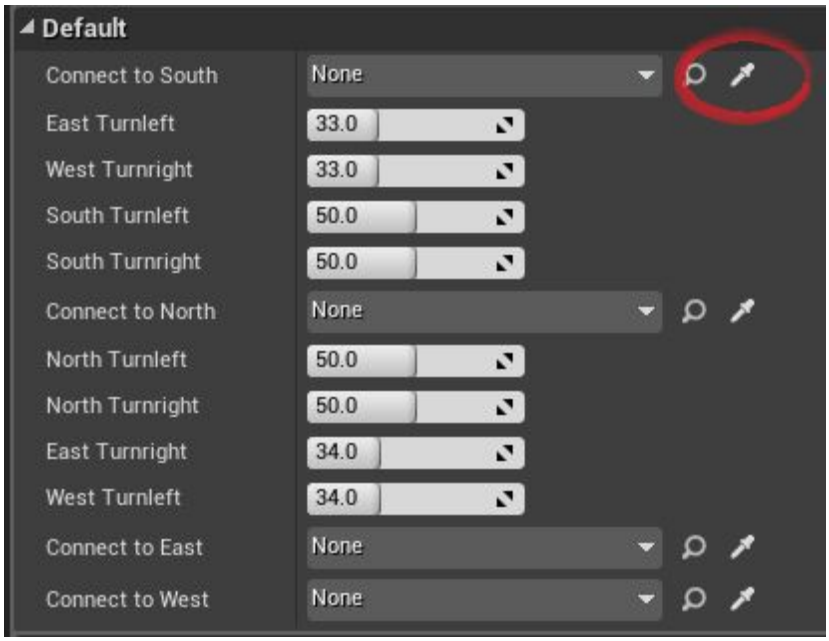

Figure 2: The pipette located in the default settings.

Step3: now click on a connector on a different road component in the viewport, but with the same color. The nodes attached to the connector pair is now connected.

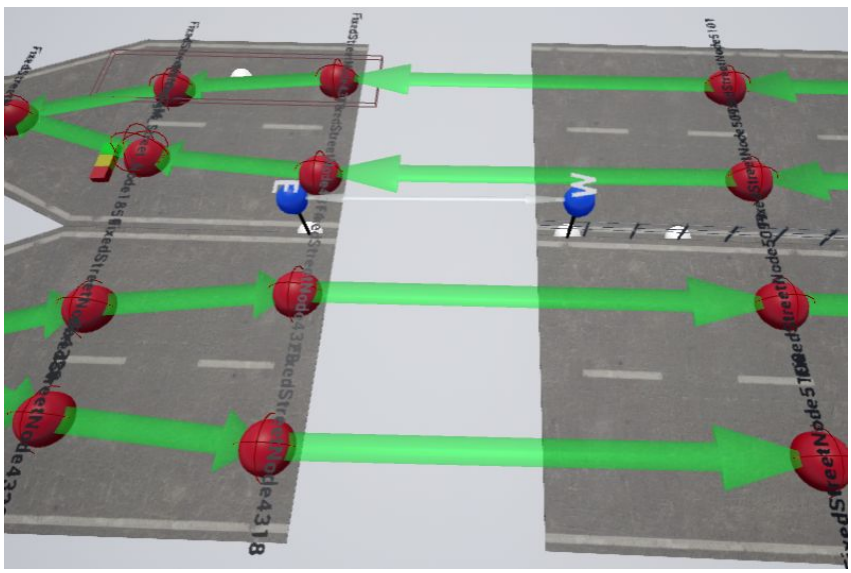

Figure 3: Two road components with connected AI-nodes.

If the road component is any kind of crossing, then there will be additional parameters in the default settings. You can specify what the percent chance of a car turning right, left or straight will be. Here follows some examples on why that is useful.

Example

If you specify the following settings:

- *North Turnleft*: 100.0
- *North Turnright*: 0.0
- *East Turnright*: 100.0
- *West Turnleft*: 50.0

It will result in no cars ever driving out of the west side of the road component (marked with a red cross in the image below). This technique can be used to shut down different parts of your level that you don't want AI-vehicles to drive in.

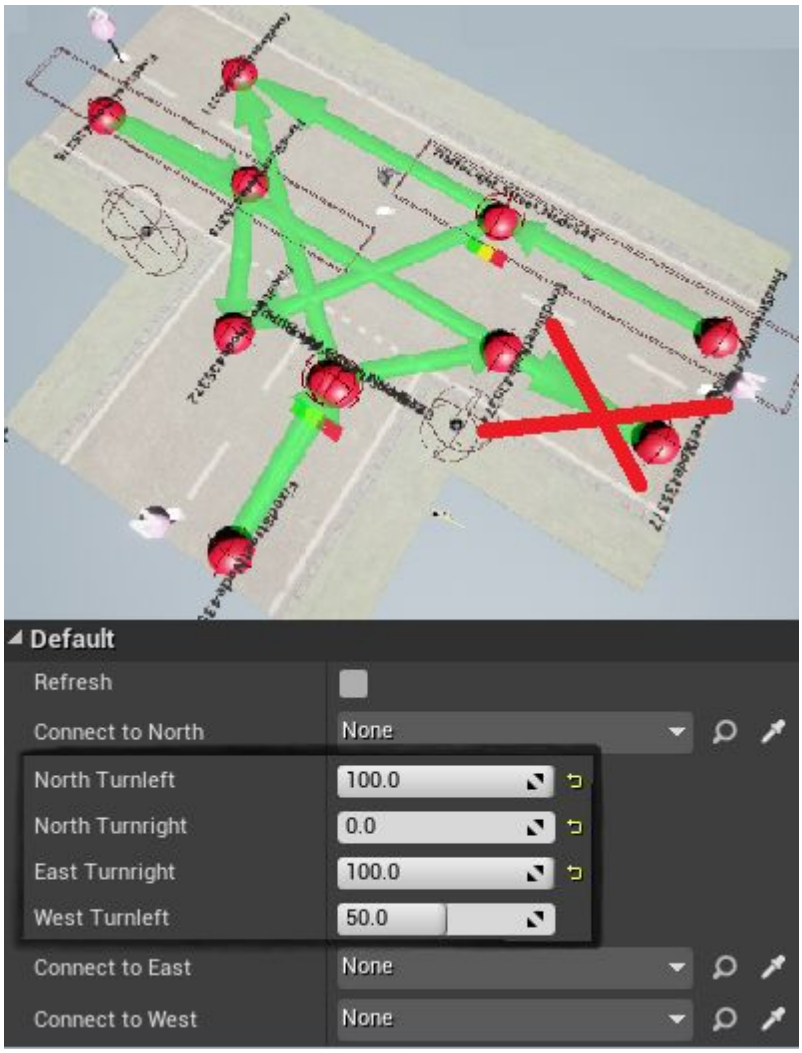

Figure 4: An example of how you can make sure no AI-cars exits through the west road.

Example

If you recognize one particular area of your level is getting too crowded after a while, then you can turn down the percent chances of cars driving into that area with the road component settings. Placing straight road components is different from placing other road components. Here follows instructions on how to place the straight road components.

Step1: Place a straight road actor in the viewport, taken from the *Finished Components* directory.

Step2: Click on the vector called *Endpoint.*

Step3: Rotate and scale the straight road by dragging this vector out in the horizontal plane.

Note! Don't rotate straight roads with rotation transform setting, rotate it by moving the *Endpoint* vector.

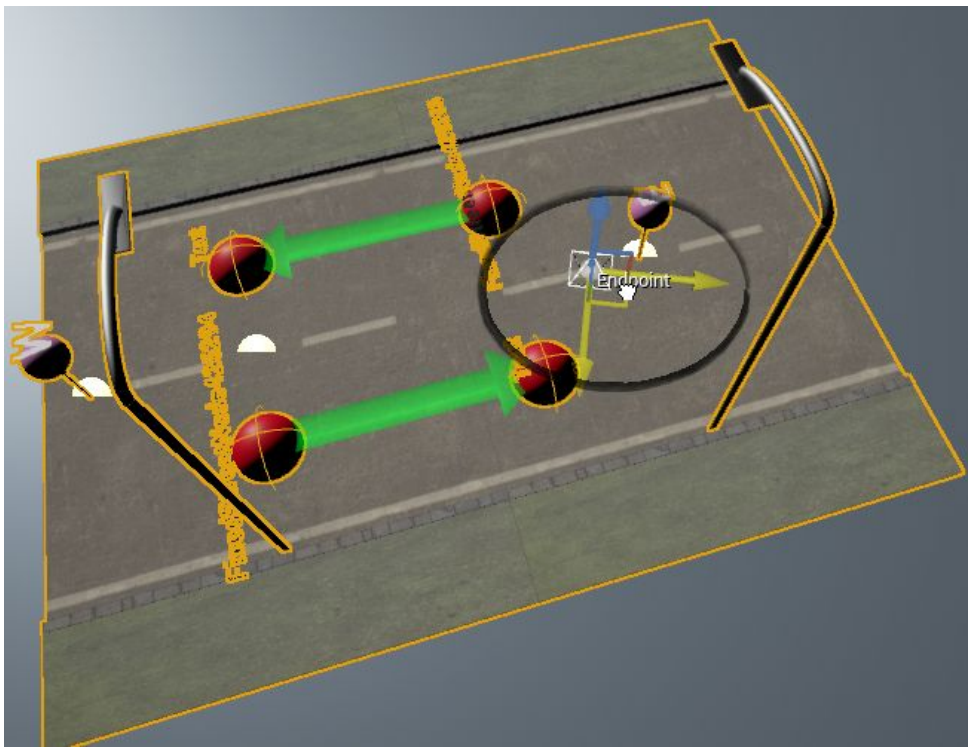

Figure 5: An image of a straight road component and it's *Endpoint* vector.

5.2. Buildings

Buildings are created by placing our included building blocks, or by creating your own. The finished buildings blocks are placed in the *Finished Components* directory. These blocks have the same number of floors and the same materials to create a unique building appearance, and the blocks can be combined to create different shapes of the same building type in the level. There are currently 18 different building types, and the different building types contains blocks such as plain walls, walls with windows and walls with windows and a door.

Figure 5: The folder structure in the “Finished_components/Buildings” directory.

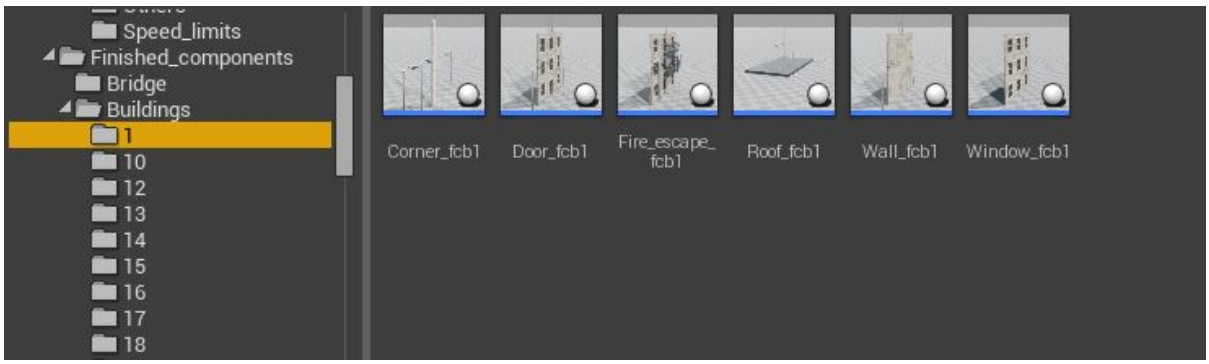

Figure 6: Example of which blocks one of 18 buildings types can include.

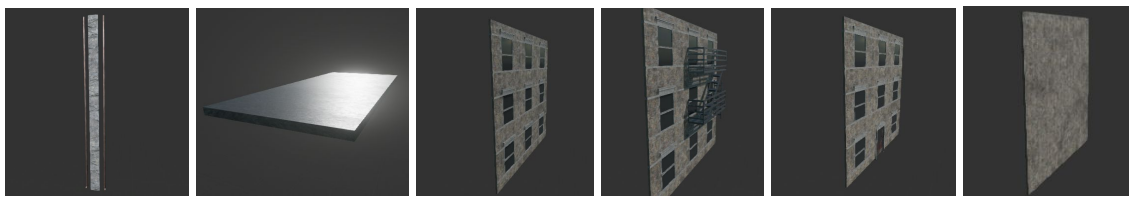

Figure 7: Examples of how different buildings blocks can look. From left to right; corner, roof, wall with windows, wall with fire escape, wall with windows + door and a plain wall.

Figure 8: An example of how the different building blocks can be combined to create a unique building.

An easy way to create new building types are by copying an existing building type and changing either the material combinations used or by changing the static meshes to other ones with a different amount of floors.

Remember that if you use roads with sidewalk, you might want to consider placing the buildings at the same height as the sidewalk. Then you also have to raise the terrain under the buildings, see chapter 5.4.

5.3. Nature

Nature is created with the built in *Foliage Tool* in UE4. That means that all trees and bushes placed on the map belong to the same actor, and that you need to use the Foliage Tools to change anything regarding the nature. The reasons foliages are used is mainly because the tools have randomisation features which are really useful when placing trees and bushes.

A tutorial on UE4 foliage can be found here:

https://docs.unrealengine.com/latest/INT/Engine/Foliage/index.html

5.4. Terrain

To change anything with the terrain, simply use the built in *Landscape Tools*. These allow you to modify the height of the terrain as well as painting different materials on the terrain. In our project we use two terrains, one for grass/dirt/cobblestone and one for the water. Hence the two different terrain objects.

A tutorial on UE4 *Landscape Tools* can be found here:

https://docs.unrealengine.com/latest/INT/Engine/Landscape/index.html

5.5. AI humans & vehicles

AI humans and vehicles can be spawned by either placing them on the map manually or by using the automatic spawn function.

Spawning an AI car manually

Step1: Choose a car model in the folder “AI/AIVehicles/Carmodels” and place the actor in the viewport.

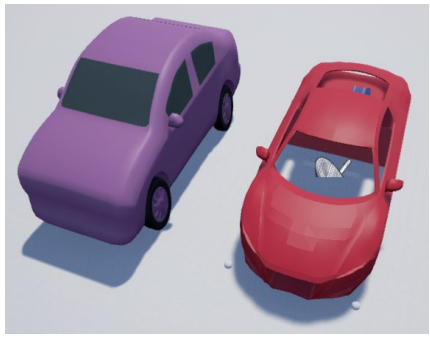

Figure 9: The two available car models.

Step2: Click on the car actor and in default settings choose a startnode for the AI by using the pipette and clicking on a red AI-node in the viewport. The startnode is the node which the car will start driving towards when the simulator is ran.

Step3: Rotate the AI car towards the node.

Note! If no start node is specified, the car will stand still in Idle state.

Spawning an AI human manually

Step1: Make sure there are multiple *AIwalkerTP* in the Level. These are the target points AI humans will select by random and walk to. If you want the AI human to walk a specific path, take a look the headline *Making events for AI humans* below . *AIwalkerTP* are already implemented in some road components with sidewalks, but you can also add additional *AIwalkerTP* by dropping them into the viewport (found in “AI/AIHuman/walkroad”).

Step2: Also make sure you have successfully created a *navmesh* covering all the areas where you wish the AI human to be able to walk. You can find information about how to create a *navmesh* here:

https://docs.unrealengine.com/latest/INT/Resources/ContentExamples/NavMesh/

Step3: Select the AI human from the folder "AI/AIHuman/AI_walker" and drop onto the navigable path, *navmesh*.

Note! AI humans have one default setting: *Start as Idle*. If set to true, the AI-walker will stand still always (unless an event is triggered). If set to false, the AI-walker will start walking towards random *AIwalkerTPs* (which you placed in step 1).

Making manual events for AI vehicles

There is a different set of street nodes that can be used to program manual behaviour for the AI vehicles that differ from the standard automated behaviour of the already implemented street nodes on road components. These are located in "AI/AIVehicle/TrafficSystem/Blueprints/EventBlueprints".

To build a path of events for vehicles you need *AIcarEventNodes* and preferably an *AIcarEventTriggerBox*. To build a path which you want a car to drive along, simply drop *AIcarEventNodes* into the level, connect them, and use an *AIcarEventTriggerBox* to toggle the states of the nodes.

Step1: Place *AIcarEventNodes* at desired locations in the level.

Step2: Set the appropriate settings on a node.

- To connect a node to another one:
    a. Press the plus sign on the *Next Nodes* array in default settings.
    b. Select a reference for next node by choosing in the viewport with the pipette symbol to the right.
    c. Select what the percent chance of this node should have to be picked. If theres only one next node, this should be 100%.
    d. Add more than one path of next node by clicking on the plus sign again.
    e. If you have more than one node to choose from, they should all add up to a pickpercent of 100.
- *Prefered speed*. This node in default settings. This is the speed which the vehicles will match when driving to this node
- *Spawn location*. This is used with the AI vehicle spawn function. If set to true, a vehicle will be able to spawn at this node.
- *State*. This is a setting from its parent class. Should be ignored for *AIcarEventNodes*.
- *Starting State*. This is the state which the node should start in
    a. Red: incoming vehicles will stop at this node.
    b. Yellow: have no effect for event nodes.
    c. Green: able to drive through to next node.

- *Follow player speed*. While driving to this node and this option is checked, the prefered speed is overridden and the vehicle will match the player's speed instead. Very useful if you want an AI vehicle to drive along with the player car.
- *Use car horn*. When the node is reached, the AI vehicle will use its car horn for this amount of seconds.

Example: When an AI vehicle has the node which settings are shown below as next node, it will drive towards the node with the same speed as the player vehicle and not in 10 km/h because it is being overridden by the check of *Follow Player Speed*. When the node is reached, the AI vehicle will immediately start driving to the node its pointing to and use its car horn for 5 seconds simultaneously.

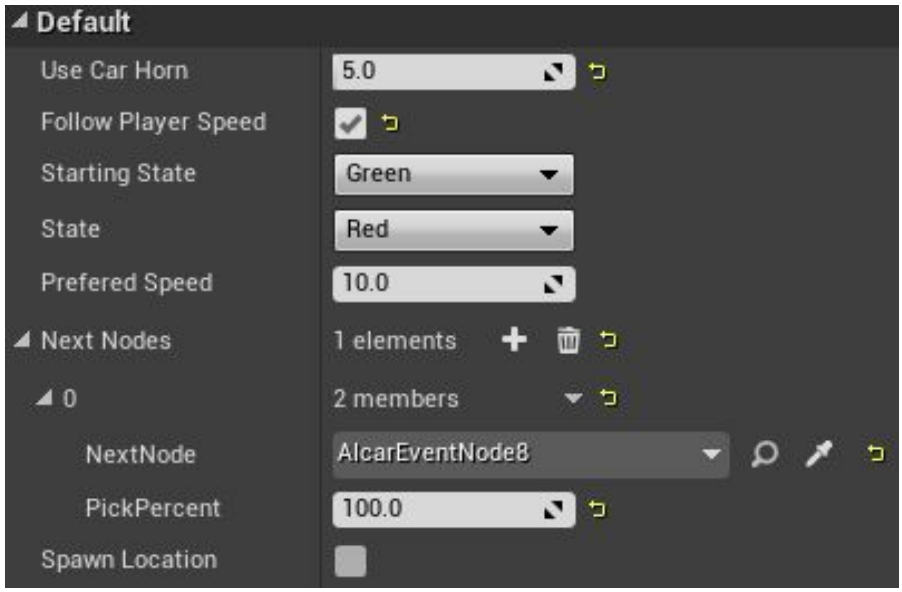

Figure 10: Example of the default settings.

Step3 (optional): If you need a trigger to change states of event nodes, place an *AIcarEventTriggerBox* at the location you want to trigger the event. Setup:

- *Trigger Actor*. Pick a reference of the Actor you want to be able to trigger the event when overlapping with the trigger box. If left as none, the player will act as trigger actor.
- *Set red*. Click on the plus sign to add a slot into the array. Then pick an event node reference with the pipette. When *AIcarEventTriggerBox* is overlapped with trigger actor, the referenced event node will change state to red after *DelayBeforeActivation* seconds.
- *Set Green*. Same principle as *Set red*.
- *Toggle*. Same principle as above, but if the event node is state red, it will change to green, and change to red if the previous state was green.

Example: When the player drives into the *AIcarEventTriggerBox* shown in the images below, these things will happen:

- EventNode6 is set to red state after 0.05 seconds. Any car driving towards this node will now stop when reaching it.
- EventNode7 is set to green after 2 seconds. Any car that was waiting at the node because the previous state was red will now continue.
- EventNode8: if previous state was green, it will change to red, if it was red it will change to green.

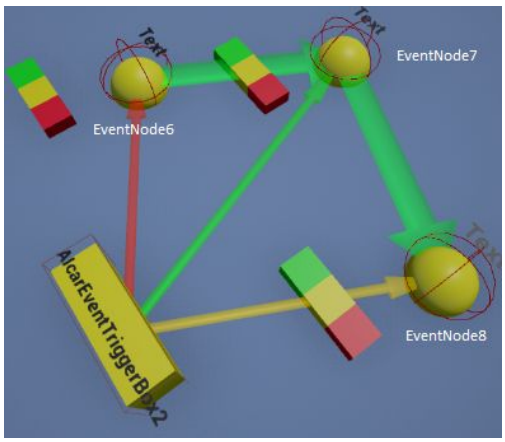

Figure 11: Example of *AIcarEventTriggerBox*.

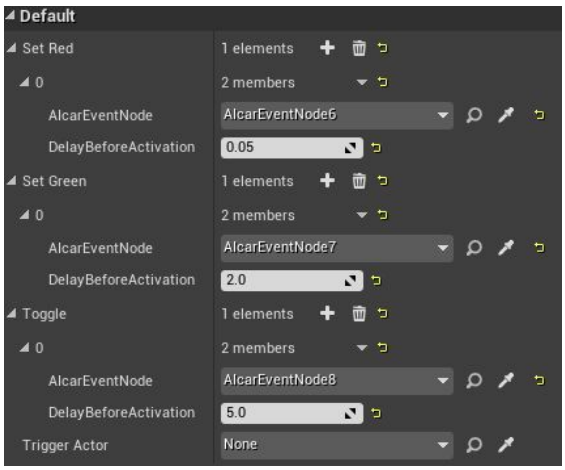

Figure 12: Example of the default settings of the *AIcarEventTriggerBox*.

Step4 (optional): The blue box is called an *AIcarEvent_switcher*. This is a useful tool used to change an AI vehicle's path from regular street nodes to event nodes. And at the last event node in the path you can then point it back in the street node structure

on the road components. This way you can make use of the predefined street nodes in your scenarios by mixing with event nodes and switchers.

Example: When the Trigger Actor AIcarVehicle20 overlaps with the blue switcher box, it will immediately switch target node to AIcarEventNode7 and follow its path. This is a scenario of a car switching lane.

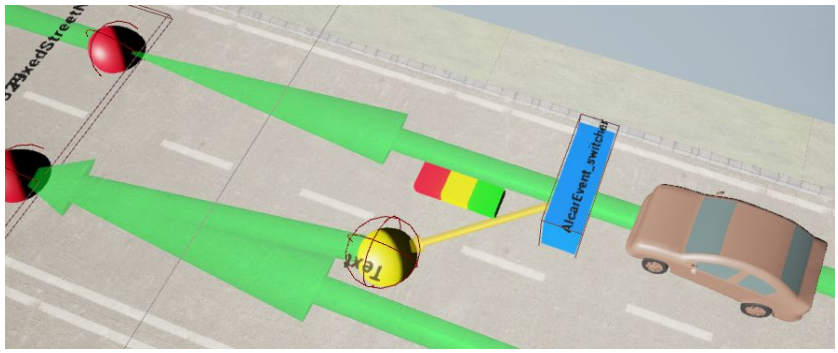

Figure 13: Example of how *AIcarEvent_switcher* can be used.

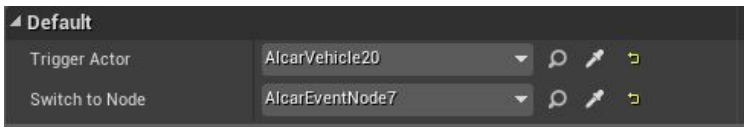

Figure 14: Example of the default settings of the *AIcarEvent_switcher.*

Making events for AI humans

Event nodes for AI humans are used to program static behaviour that can be used in scenarios.

All you need is a *AIwalkerEvent Actor* to create custom behaviour for an AI human, located in "AI/AIHuman/AI_walker". It consists of two objects, the first being a mesh called *AIwalkerTP*. This is where the AI human will walk to when the event is triggered. The second object is a *TriggerBox*, which triggers the event when the specified *Trigger Actor* overlaps with the *TriggerBox*.

Step1: Place an AI human to be affected by the event in the level.

Step2: Determine if AI human should start with walking its *RandomWalk* or *start as Idle* with default settings in the AI human actor.

Step3: Place an *AIwalkerEvent* into the Level. In the details window, select the *TriggerScene* to position the *Triggerbox* and select the *TargetScene* to position the target point for the affected AI human.

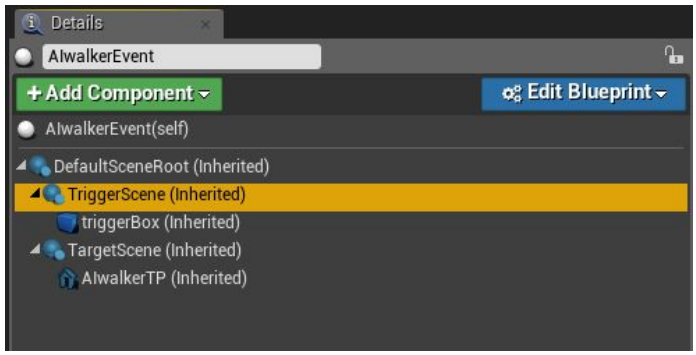

Figure 15: *AIwalkerEvent* details window.

Step 4: In default settings:

- *Trigger Actor*: Use the pipette, to select an Actor that should trigger the event when overlapping with the *TriggerBox*.
- *AIwalker*: This is the AI human who should walk to the target point when the event is triggered. Use the pipette and select in viewport.
- *Delay*: This is the delay between the moment Trigger Actor overlaps with TriggerBox and the targeted AI human is being ordered to walk to the target point.
- *Walk Speed*: The speed the AI human should use when walking to the target point.
- *Resume Random Walk*: If this box is checked then the AI human will continue in the state *RandomWalk* after he reached the target, otherwise he will stand still.
- *Destroy Event*: If this box is checked then the *AIwalkerEvent* Actor will be destroyed after the event is over, ie. the event cannot trigger twice.
- *Ignore rules*: Should the AI human ignore the *NavmeshBoundaries* on the roads or not.

Example:

The images below shows that when the AI vehicle, AIpawn_audi3, overlaps with the *TriggerBox* the event will trigger. First, there is a 2 second delay, then the AI human, AIwalkerCharacter2, will walk to the yellow box with a speed of 300. Finally, the event Actor will be destroyed and the AI human will change state to Idle.

Tip! You can use the affected AI walker as the trigger Actor itself, then you can create a path of *AIwalkerEvents* the AI human will walk along.

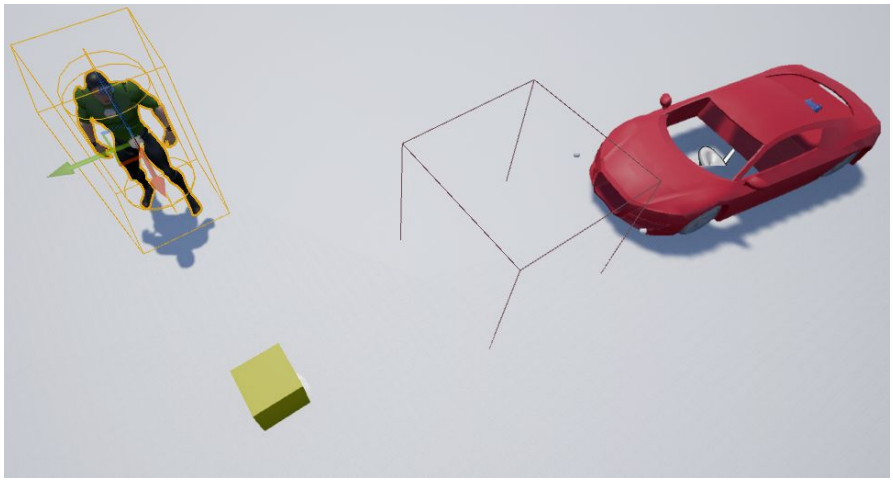

Figure 16: The example scene.

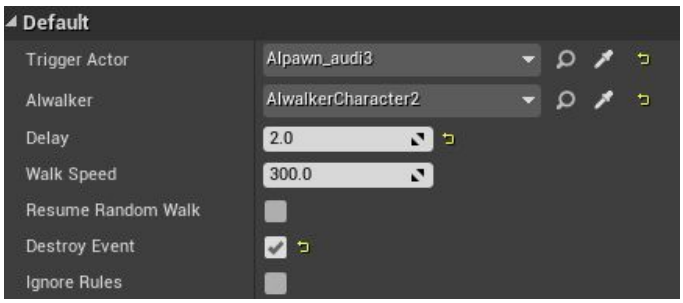

Figure 17: The default settings used in the example.

5.6. Scenarios

Building scenario

We created two very simple scenarios to show how they can be implemented. What we did was we copied the Open World level and then removed all the parts that we did not want to have in that specific scenario.

This was mainly done because of performance, the more actors there are in a level the lower the performance. Since we had crossings in our open world we placed out barriers at all the roads in the crossings where we did not want the driver to be able to go. This can also be achieved by simply using roads where you can only drive in one direction.

We then put out a starting location for the player in the level, more info on that topic is found here:

https://docs.unrealengine.com/latest/INT/Engine/Actors/PlayerStart/

To end the scenario we used the *EndOfSimulationBox* which is found in the *Scenarios* folder in the project. When you place this in the level you will see a big box. When the driver enters this box when running the scenario it will automatically end the scenario and all the collected data will be saved in the folder selected by the user in the Statistics menu.

By default the logs will be saved in a folder named *Logs* found at the root of the simulator folder. This *EndOfSimulationBox* is however only visible in the editor, the player will not see it when driving in the scenario.

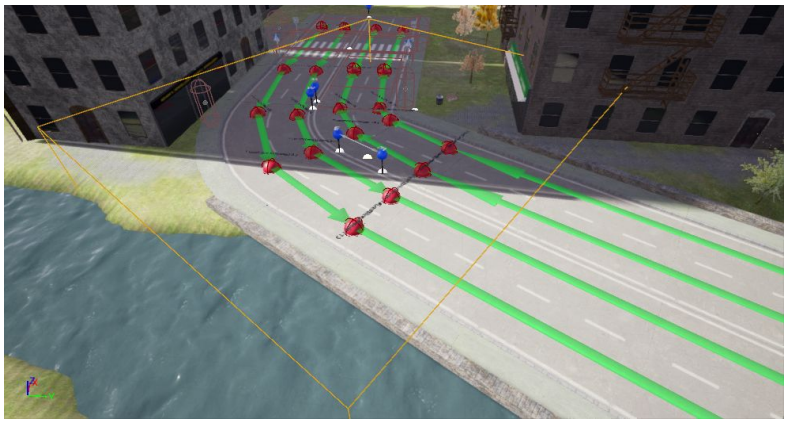

Figure 18: The yellow box is the *EndOfSimulationBox* in our first scenario.

You can of course implement the AI as you see fit for your scenario, see chapter 5.5 for more information. You can also create your own level from scratch, see chapter 5.1-5.4 for more information.

In our first scenario we used an event for an AI-walker to run out on the road when the driver passes a specific point in the scenario. This was done using an *AIwalkerEvent* found in the *AI_walker* folder. When the walker reaches the road it triggers another *AIwalkerEvent*, but on this one we added a delay of 5 seconds. So after 5 seconds it continues to the sidewalk on the other side of the road.

In the second scenario we had a car driving out in front of the driver at a T intersection, but then the car continues to follow the basic node-system implemented in the road components. This was accomplished using an *AIcarEventTriggerBox* found in the *EventBlueprints* folder.

When the driver passes this box it switches the state of the *AIcarEventNode* where the car is waiting to green. This allows the car to continue to the next *AIcarEventNode*. When we wanted the car to follow the node-system we simply connected the *AIcarEventNode* to a regular node in a road component, and then the car continue to drive using the node-system in the road components.

Adding scenario to the Level Menu

In order to add a scenario to the level menu you simply have to add an *AddLevel* node to the *FillMenu* function in the blueprint *LevelMenu* located in the *LevelMenu* folder found at “Content/UI/Menus/”.

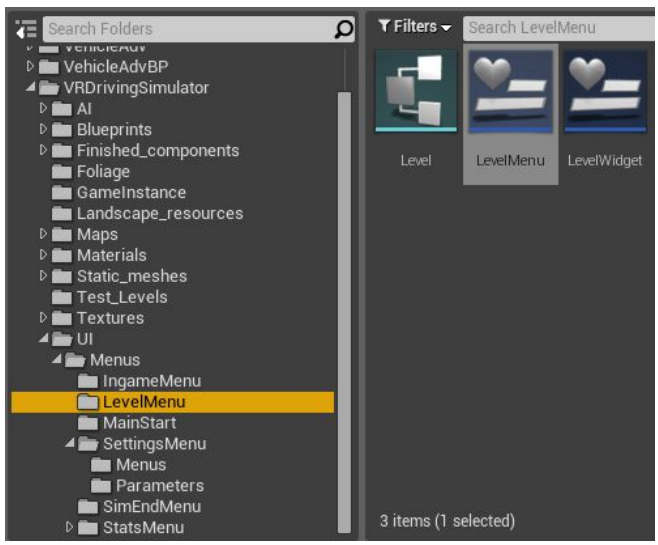

Figure 19: Shows the location of the *LevelMenu* blueprint in the folder structure.

The *Add Level* node has three important parameters, excluding the target parameter as it's supposed to be left empty. The *Level Name* parameter should be set to the name of the created scenario and needs to be spelled correctly in order for the level to launch properly when selected in game. The second input, *Description*, is only visual and has no impact on the game but for clarity it's suggested that a short and concise description of the scenario is written.

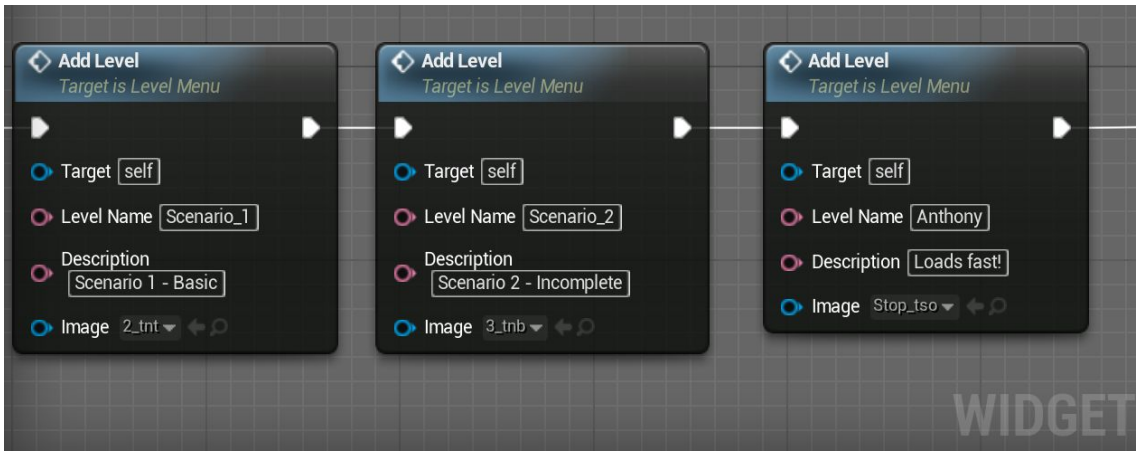

Figure 20: A working example of three levels, or scenarios, in the *FillMenu* function.

The third input *Image* lets you select an image that is shown in the menu when you select a level in the game. You can import new image files by pressing the import button which is by default located at the top of the content browser of the UE4 Editor. UE4 accepts a multitude of formats such as but not limited to JPEG, PSD, TGA and PNG.

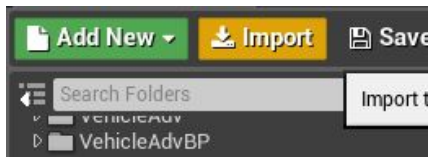

Figure 21: The import button is depicted in yellow in this picture.

Once the import has been completed you only have to click the dropdown menu of the *Image* parameter of the *Add Level* node and search for the name of your imported Image asset. There are also several image assets that are already a part of project that you can select from the dropdown menu if you want to.

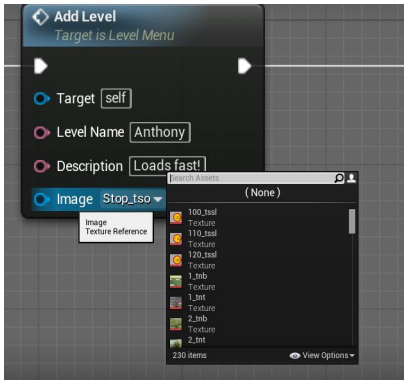

Figure 22: Pressing the arrow pointing down shows this dropdown menu which the user can choose an image from.

Note: Don't forget to attach the last *Add Level* nodes output execute wire (the white wire) to the next node in line as seen in the last figure!

5.7. Parameters

In order to add a new setting to the settings menu you have to first locate the *ParameterContainer* file located in the *Parameters* folder at “Content/UI/Menus/SettingsMenu/Parameters/”.

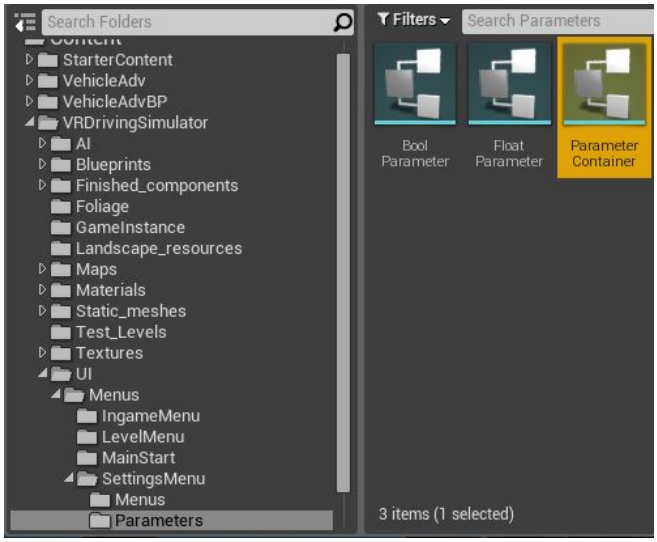

Figure 23: Displays the location of the *ParameterContainer* file.

Once located the user then has to open the file and add the setting as an element to the default values of either of the two arrays provided in the struct. If the setting is an on/off setting (such as a setting like *Honking*) then the *BoolParams* array should be chosen and if it’s a scalar value (such as player height) then the *FloatParams* array should be selected instead.

Figure 24: The marked plus signs is where elements are added to the arrays.

Once pressed you are made to fill out a three fields, *Parameter*, *Description* and either *Enabled* or *Value* depending on if you made a *BoolParam* or a *FloatParam*. The *Enabled*/*Value* field will be the default value of the setting in the simulations and menus. The parameter field is the name of the setting and the description field is the text describing the setting in the *Settings Menu*.

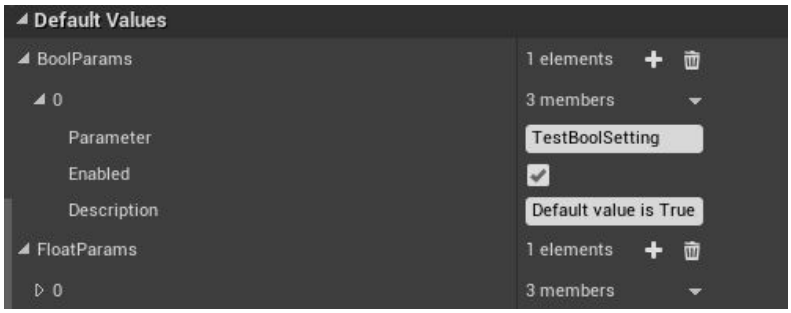

Figure 25: A new setting of boolean type by the name of TestBoolSetting was added and its default status is enabled.

Note: It's important to remember the name of the setting (the Parameter field) as it is used for programming the setting!

Once the above has been done all that remains is doing the programming for the setting. In order to access the variable for the setting (the *Enabled* or *Value* parameter depending on type) usage of the *GetParamByName*, *GetBoolParam* and *GetFloatParam* functions located in the *VehicleAdvGameMode* blueprint. They can be accessed by using the *Get Game Mode* node from any blueprint.

Note: The *GetParamByName* function has a parameter named *Is Boolean* which should be set to true if the setting you're attempting to access is of the boolean type, otherwise it should be false.

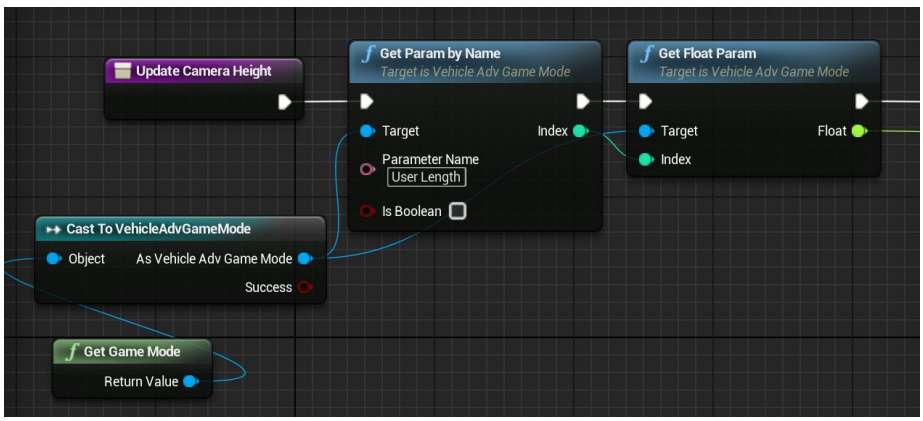

Figure 26: An example of how to get the *Value* parameter from a float, or scalar, setting in the *Update Camera Height* function.

The logic for updating the simulation with your new setting should be contained in one update function such as the *Update Camera Height* function in the figure above. That function should then be added to the *ActivateParams* function in the *SettingsMenu* blueprint graph (located at "Content/UI/Menus/SettingsMenu/Menus/") as seen below:

Figure 27: An example of an update function in the *ActivateParams* function.

The *ActivateParams* function is ran every time a simulation is started or unpaused and is intended to update all settings to their (potential) new values if they were changed while paused or before starting. Variables to both the player car and the game mode can be found as local variables and should be used as the *Target* of the update functions.

Note: Don't forget to attach the output execute wire (the white line) to the next node as seen in the figure above when you add new settings!

5.8. Statistics

Changing the default Log directory

In order to change the default log directory all you have to do is open the file *StatParameterContainer* located in “Content/UI/Menus/StatsMenu/Parameters” and edit default value of the *LogDirectory* string variable in the UE4 Editor.

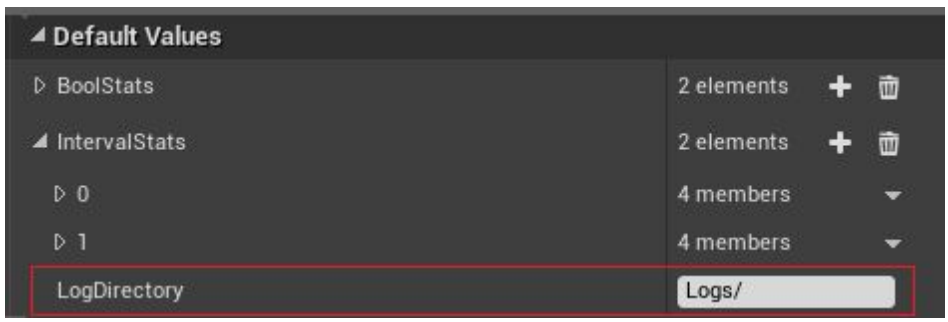

Figure 28: The squared field should be edited in order to change the default log directory.

Note: Don’t leave the default value of the *LogDirectory* variable empty. That will cause nasty things to happen.

Adding new statistics

In order to add more statistics you first have to open the file *StatParameterContainer* located in “Content/UI/Menus/StatsMenu/Parameters”. In the file you need to and insert a new element to either the *BoolStats* or the *IntervalStats* array depending on if the statistic you intend to create is one that is updated automatically upon a condition being met (such as car collisions) or if it is a statistic that needs to be polled at a certain interval (such as speed or the angle of the wheels). The latter of the two should be added to the *IntervalStats* array whereas the first one should be inserted into the *BoolStats* array.

The insertion is made by pressing the plus signs to the right of the names in the *Default Values* box.

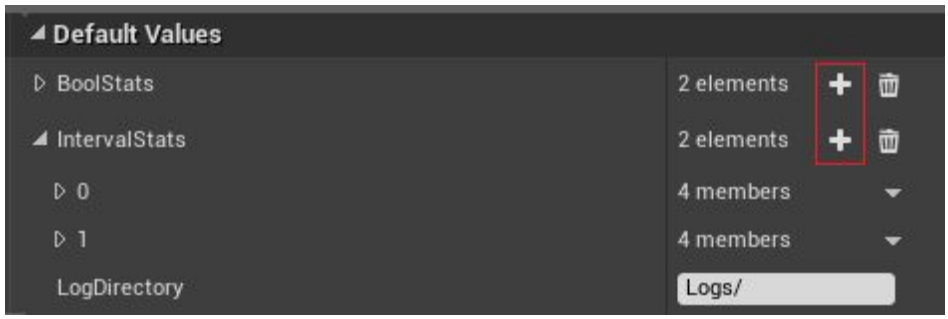

Figure 29: By pressing the plus sign you add a member to the associated array, and thus a new statistics is shown in the menus.

The fields *ParameterName*, *Description* and *Enabled* then need to be filled. If you made an *Interval* statistic you also need to fill in the default polling interval of the statistic, ie how often the statistic should be retrieved by default. The *ParameterName* and *Description* parameters are for the visual appearance in the statistics menu.

Figure 30: The fields need to be filled in and if you made an *Interval* statistic, ie one that polls data repeatedly, then you also need to fill in the Interval field.

Note: It's important to remember the *ParameterName* as it's used when setting the polling function (the function that retrieves the data).

Tracking Boolean Statistics

Tracking, or programming, the new boolean statistic (the name is somewhat) is done by creating an update function in the *Statistics* file (located in "Content/UI/Menus/StatsMenu/") and then running that function when the event fires. In order to get the *Target* you can use the *Get Game State* function and cast the reference you get from it to *Statistics* as shown in the figure below.

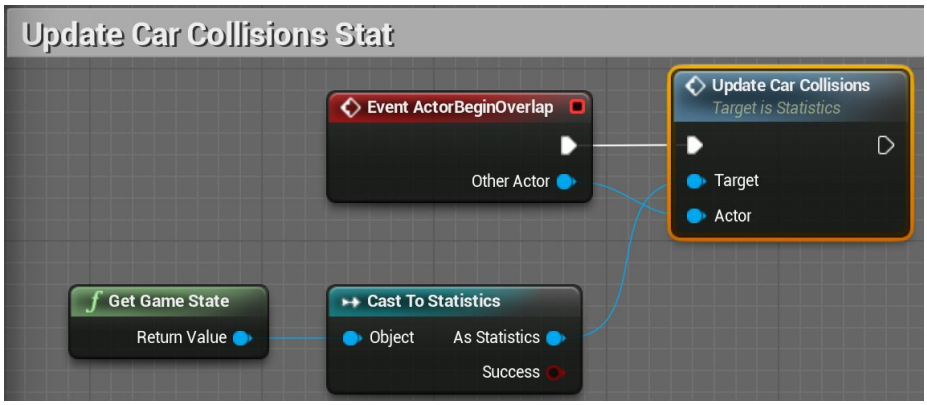

Figure 31: The function *Update Car Collisions* is ran when the event *ActorBeginOverlap* has fired.

The update function should add a string with the new data to the *Log Data* array in the *Statistics* blueprint as that array is later used for creating the log file.

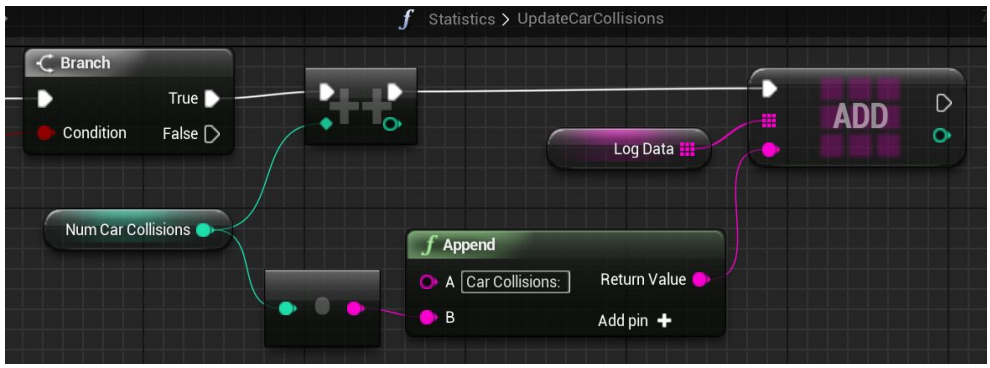

Figure 32: The *UpdateCarCollisions* function adds a string containing the current number of car collisions to the *Log Data* array. The logic for the update itself is mostly not shown in this figure.

Note: A variable in the *Statistics* blueprint can be created to hold the value of the data which you are tracking (such as the number of car collisions the user has been involved in, *Num Car Collisions* as shown in the figure above).

Polling Interval Statistics

To fully set up a new *Interval Statistic* you then need to create a function in the *Statistics* blueprint which gathers the data you want to track. It then needs to add that data as a string to the *Log Data* array in the *Statistics* blueprint as that's where the data used to create the log file is taken from.

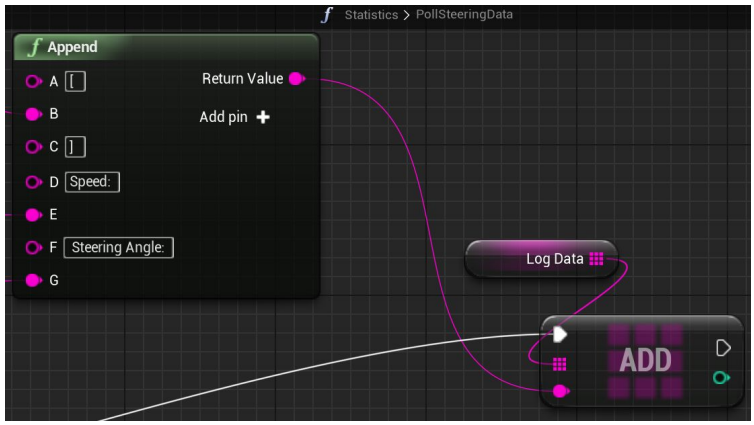

Figure 33: The PollSteeringData function adds a string to the Log Data array containing the data of the speed of the player driven car and its steering angle.

Note: The polling function you create should poll only once, not several times. The interval polling will be taken care of.

Finally you need to tell the simulator to use that function for you just created for the new statistic by using the *SetPollingFunction* function in the *Polling* graph of the *Statistics* blueprint. The *SetPollingFunction* has two important input parameters. The first one is the name of the statistics which you want to track, and it needs to be filled in correctly in order for the tracking to work. The second parameter is a string which takes the name of the function you created to gather your statistical data.

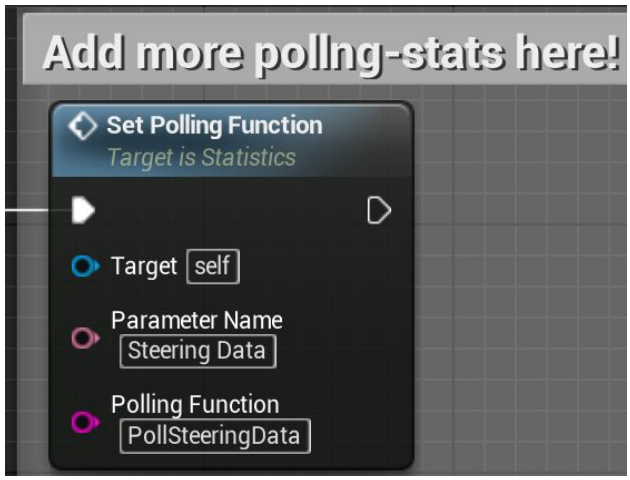

Figure 34: The *PollSteeringData* is set as the polling function for the steering data statistic (parameter). The *SetPollingFunction* ensures that the *PollSteeringData* function is ran as often as the interval value is set to.

Note: Do not set several polling functions for one statistic, that could potentially create issues.

5.9. Car

Making changes to the standard model

Select all the materials, the skeletal mesh and the static mesh for the car model. Right click and click on *export selection*. Pick a name and select the *fbx* format. Import the *fbx* to any 3D modelling program that supports this format, such as Maya or Blender, and make your changes to the model. Export it as an *fbx* and continue the tutorial below.

Adding a new model with standard functionality

This guide assumes that you have a model consisting of a skeletal mesh with at least 9 joints. One for each wheel, one for each rearview mirror, one for the steering wheel and one root bone.

To change the car you need to import the model (in fbx format) of your desired replacement car to the project as a skeleton mesh.

Now go into the vehicle blueprint. In the Component view (usually in the the top-left corner) click on *mesh* and in the details view (usually to the right) you will now see a lot of details about the current car mesh. To change the car model you simply change the skeletal mesh in the details view to the one corresponding to the model that you just imported.

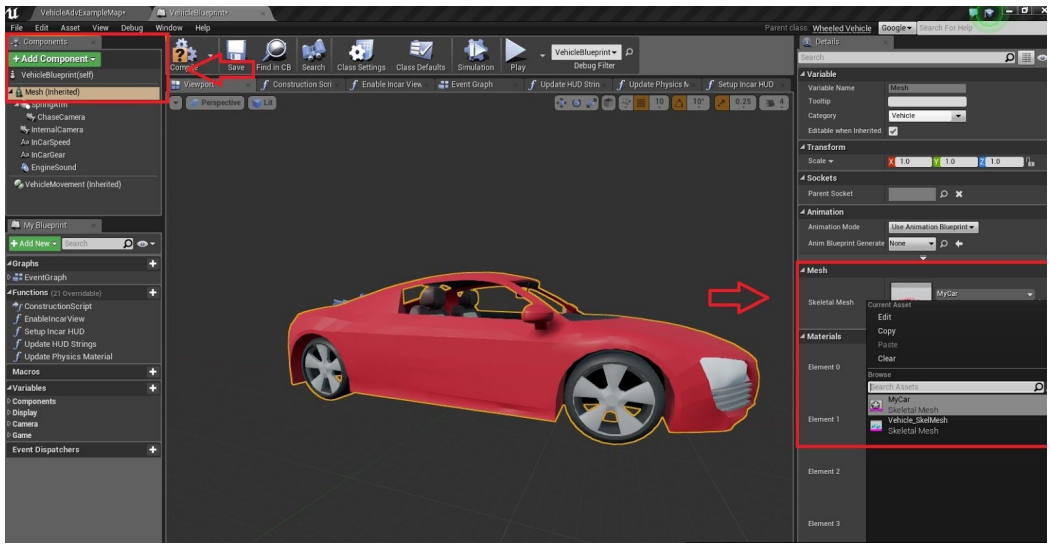

Figure 35: Where to change the skeletal mesh.

Now go into the physics asset of your new model. Unreal will have automatically generated collision boxes for every part of your mesh that has a joint. Remove all of

the collision boxes except the one for the body of the car and the ones for the wheels by selecting them and clicking the delete button.

To the right you will see a hierarchy for your bones. Select each wheel bone and in the details view, set their *Physics Type* to *Kinematic*. Also take note of the names of these bones. You will use them later.

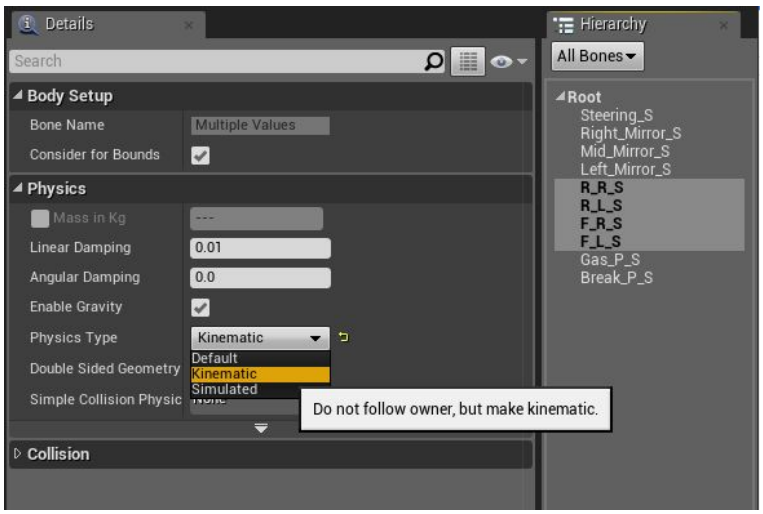

Figure 36: How to change *Physics Type*.

You might also want to change the shape of the collision box for the car body to a box shape. Right click the root bone and click *reset*. Change the *Collision Geometry* to *box*, uncheck the *orient along bone-option* and hit *ok*.

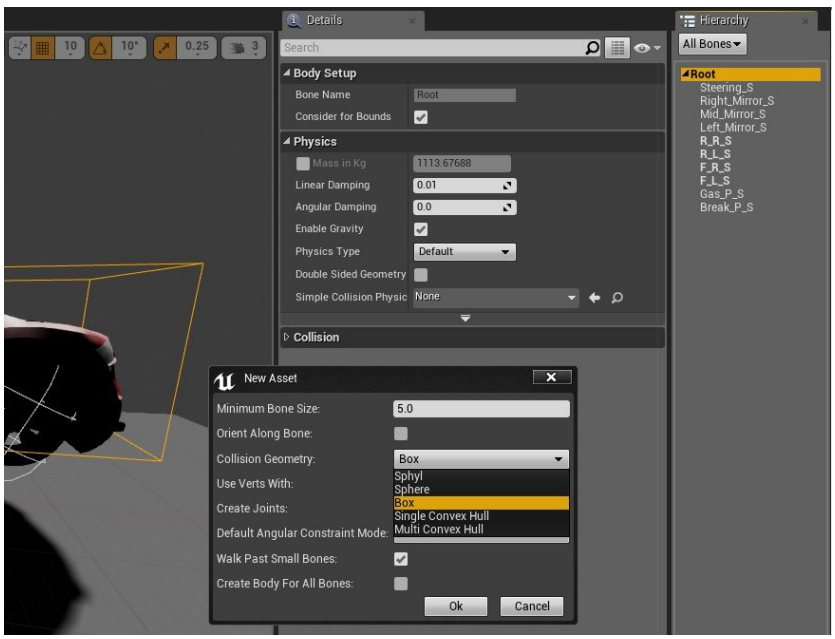

Figure 37: How to change collision geometry.

Now we want to create an animation blueprint for our model. Right click in the content browser and select *Animation blueprint*. Select *VehicleAnimInstance* as the parent class and select your skeleton mesh in the bottom table. Also rename the animation blueprint to something fitting.

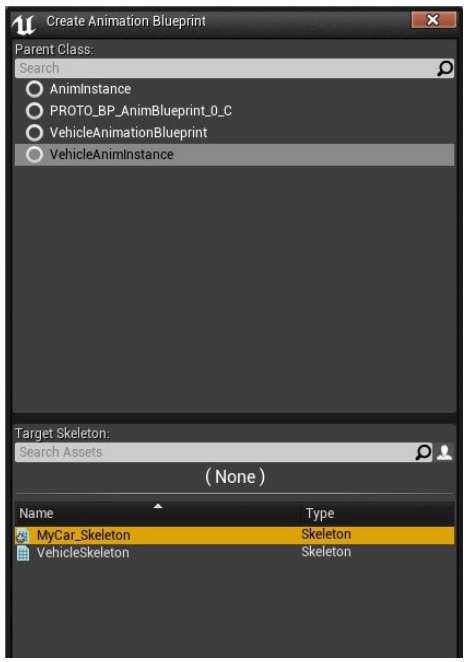

Figure 38: How to create new *Animation blueprint*.

Now copy all of the nodes from the previous car's animation blueprint into your new one .

Change the name references to the bones in the animation blueprint to the names of your new models bones which you saw in the physics asset.

Return to the vehicle blueprint. Click on *mesh* in the components view and change the animation blueprint to the one you just created.

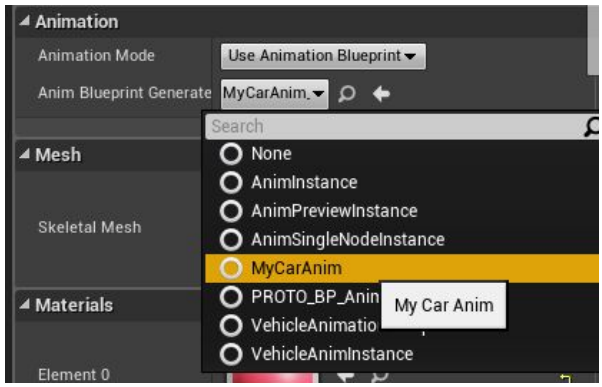

Figure 39: How to select the correct *Animation blueprint*.

Also click on *VehicleMovement* in the components view and under *Vehicle Setup* change all the bone names to the wheel bone names of your new model.

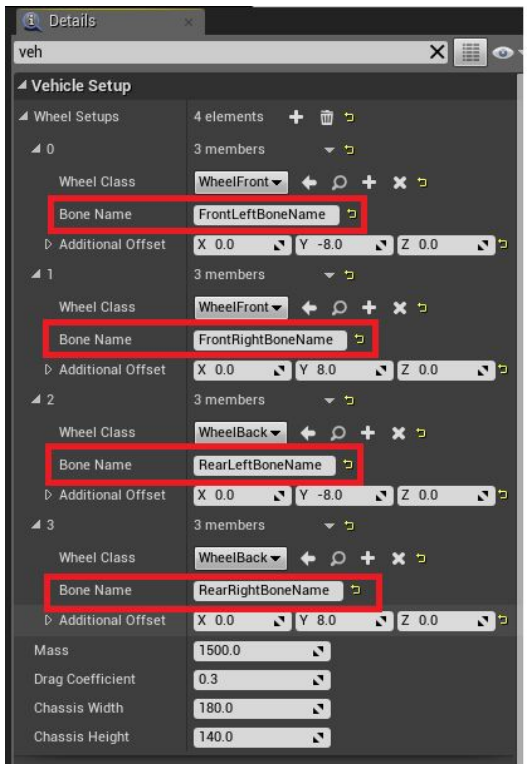

Figure 40: How to set up wheel bones.

Finally, go into the wheel blueprints (front and back) and set the size of them to that of the wheels on your new model.

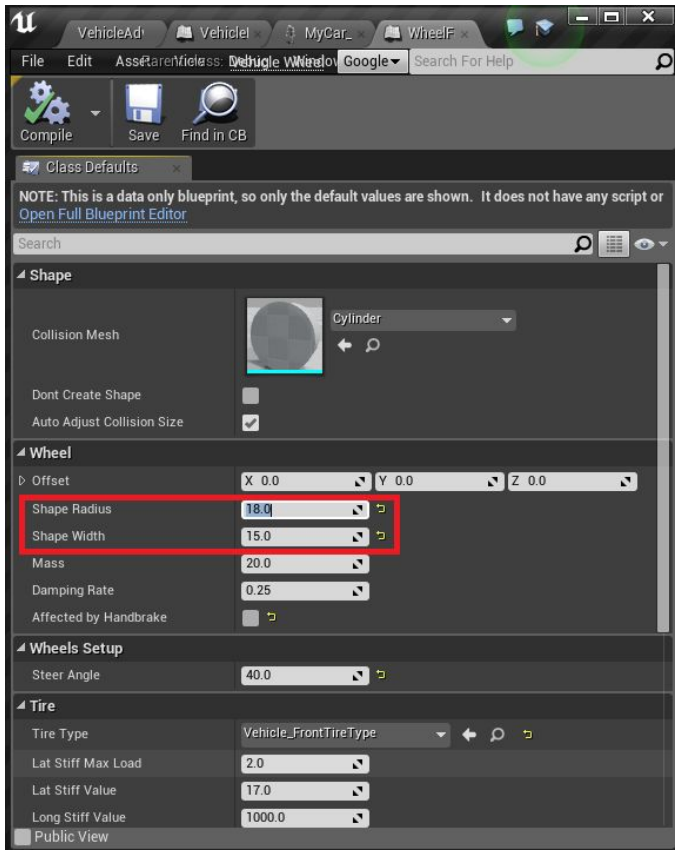

Figure 41: Where to change size of wheels.

Adding model without standard functionality

If you have a more simple model that you want to add. For example one that lacks rearview mirrors you can follow this guide.

https://docs.unrealengine.com/latest/INT/Engine/Physics/Vehicles/VehicleUserGuide/

Keep in mind that you will have to create a new vehicle blueprint, set up all the functionality for your new car from scratch and change the default starting vehicle pawn in every map where you want to use it.

6. Known problems

In the following chapter we describe problems that were known, as of May 8th 2016, that we knew existed but didn't have time to fix. These problems would be a good start when continuing development of the simulator.

Shadows

Shadows aren't working properly on all objects because we didn't have time to fix them. Things that don't have working shadows are for instance road poles, traffic lights, street lights and road signs. To fix these shadows you need to look at the *static/movable* settings, the UV mappings (so they are not overlapping) as well as the shadow settings (*shadows enabled*, *cast dynamic shadow* and *cast static shadow*).

AI

The AI is complicated and contain a lot of code. Most of the functionality is there, but it might not work properly in some cases. We tested the AI a bit, but a lot more time can be spent on testing and improving the AI-logic.

Nature

Trees and bushes are made with a simple texture-trick. If you look at the static mesh for trees and bushes you will see that they are 3-4 planes rotated around an axis. Then by using textures with transparency an illusion of nice looking trees and bushes is created. This does however result in that there are no shadows rendered.

A better way of creating trees and bushes might be considered. Our way is fast and it gets the job done. But for a better performance, real static meshes might be considered. Another thing to look out for is shadow complexity, since the current solution have terrible shadow complexity. Finally, you can also look into texture MipMap, since they are currently disabled for the nature.

Roads

Some road pieces might be too narrow and hard to drive on. Consider creating larger road pieces for a better driving experience.