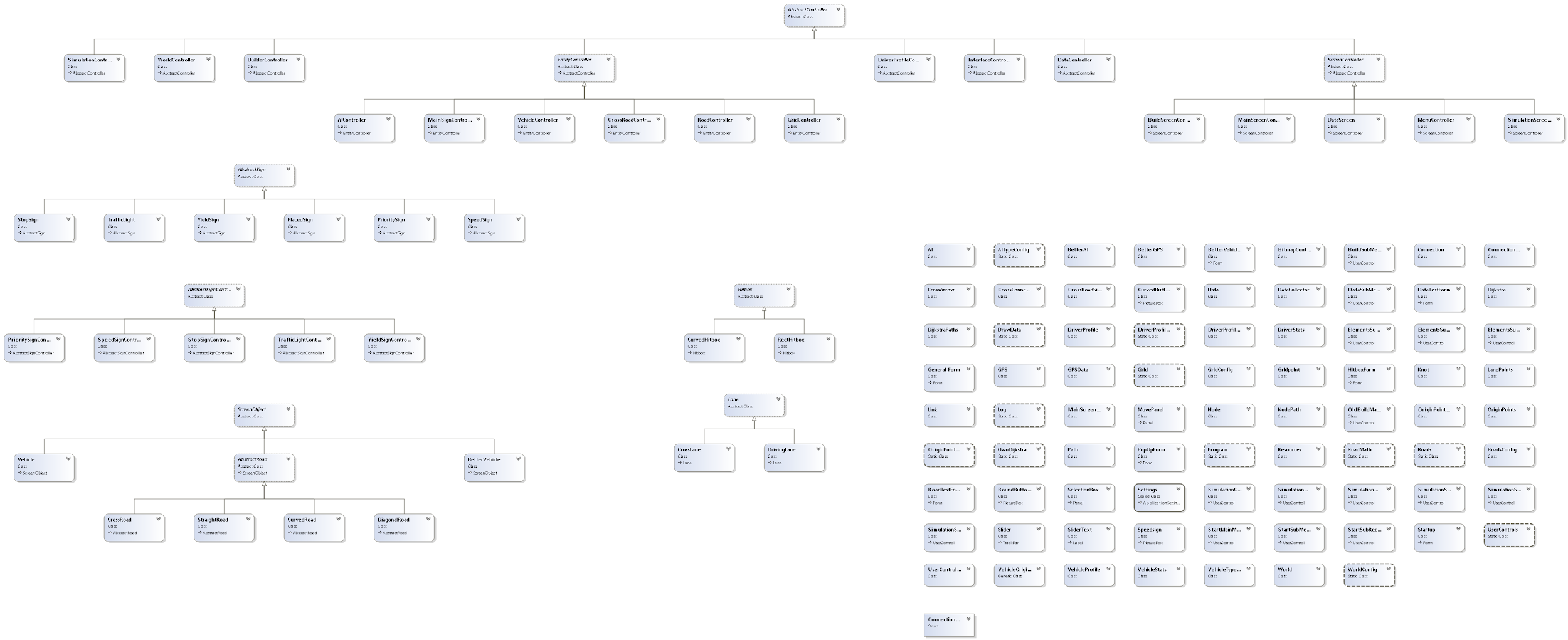
**Final Report**

Green Light District - A Traffic Jam Project (Szilagyi Group 2)



Introduction

This is the final report of Green Light District’s Traffic Jam project. Here we explain every part of the project’s complex code in detail to give a clear description of how everything works together. The image above is the code’s flowchart, in which all classes and their connections are visualized.

Controllers

Everything that happens in our project is handled by a few controller classes. All controllers are subclasses of the AbstractController class, which forces them all to have an Initialize function. AbstractController has five direct subclasses.

There is an EntityController, which is added for structure, and is the superclass of all controllers that deal with entities: the grid, roads, signs, AI and drivers.   
The BuilderController deals with building roads and initializes the grid-, road- and sign controller.

The SimulationController deals with simulations and initializes the AI- and driver controller.

The InterfaceController deals with switching between (sub)menus.

And lastly the ScreenController, which is a superclass of four other important controllers. It forces all of them to have an activate and deactivate function.

* The most important controller is the MainScreenController, which is created first and enables all other controllers and initializes the program.
* The Menucontroller enables switching between the different main menus/screens.
* The BuildScreenController handles everything that happens visually in the build menu/screen.
* The SimulationController handles everything that happens visually in the simulation menu/screen.

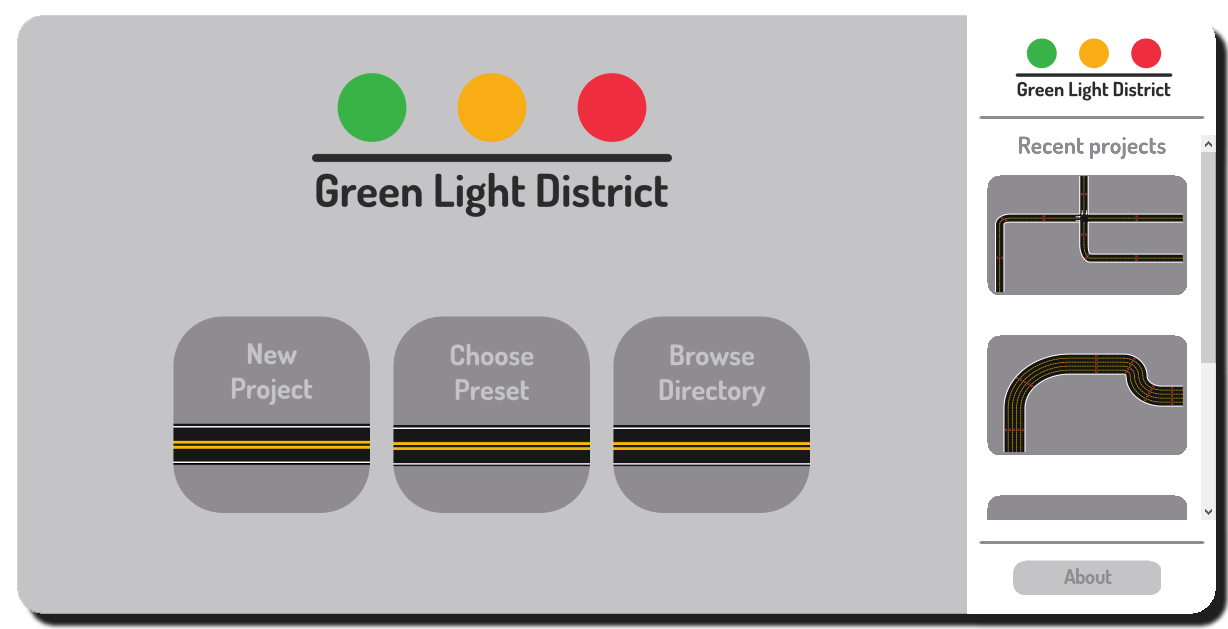
Grid

The grid consists of three distinct parts: The GridController class, which manages the list of grid points and can be called upon to populate the screen with a grid, the GridConfig class and json file, which manages the settings for the grid and the GridPoint class, which contains the behaviour of a point in the grid.

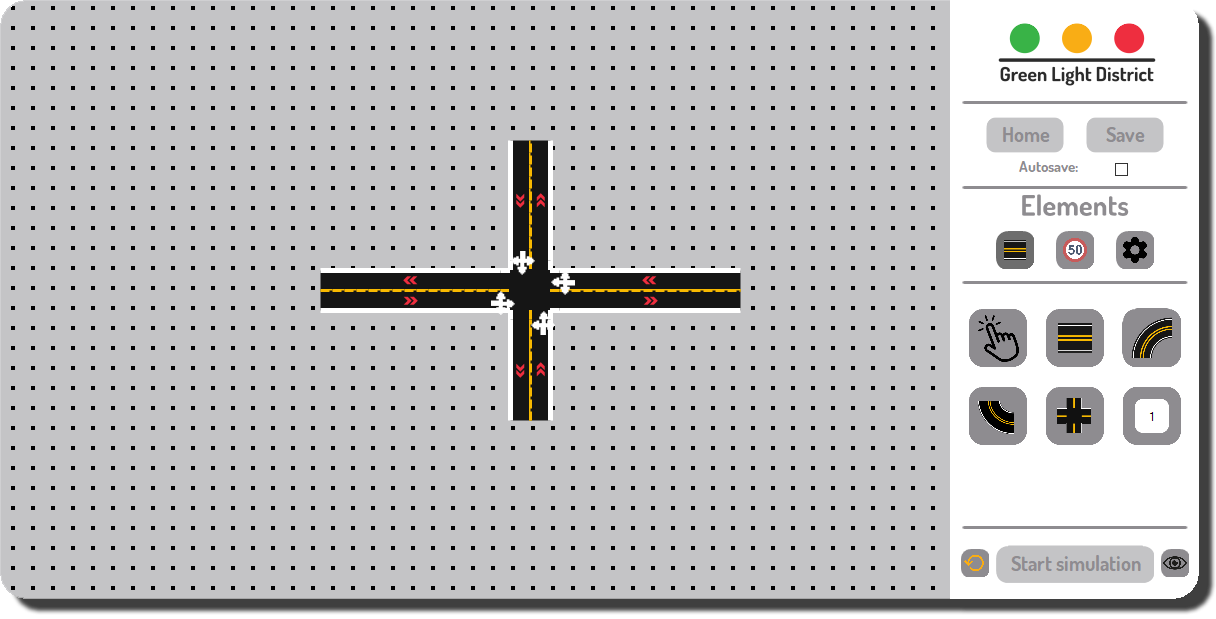
The GridConfig class reads in the settings from a json file of the same name. This makes it easier to change variables such as the space between points, which can now be done by changing a number in the json file.

The GridController contains a list of GridPoints, as well as a list of lanes. This is necessary because the number of grid points needs to change with the screen size, which is done through the CreateGridPoints function, which goes through each row and fills it with points, then moves on to the next. The OnClick method handles user input. The user can click on the first point, then drag their mouse to the desired point. Meanwhile, a rectangle will appear that tells the user if the road is legal to place. The right mouse button can be clicked to reset the clicked points. The method also contains methods for determining this legality.

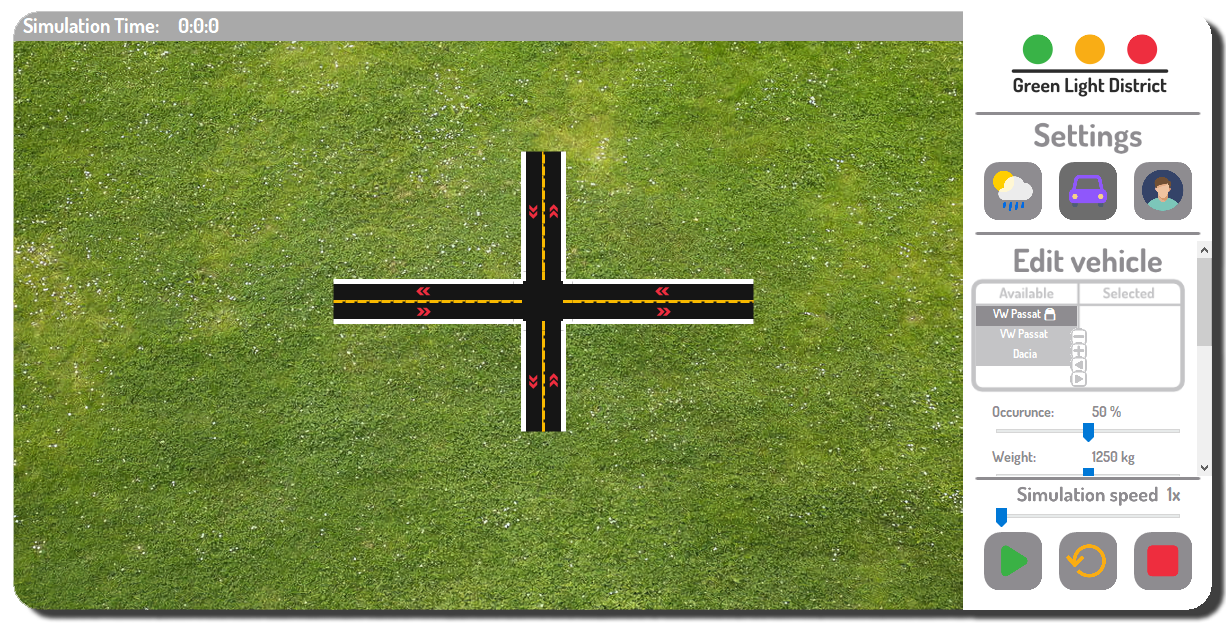
A gridpoint has the following functionality. First of all, there is a constructor method which sets the size of the point. Next, a collision function to detect when a user has clicked on the point’s hitbox and a Draw function which contains the method in which a point is drawn to the screen. Finally, an override of the ToString method so that it now returns the coordinates of the point.

User Interface

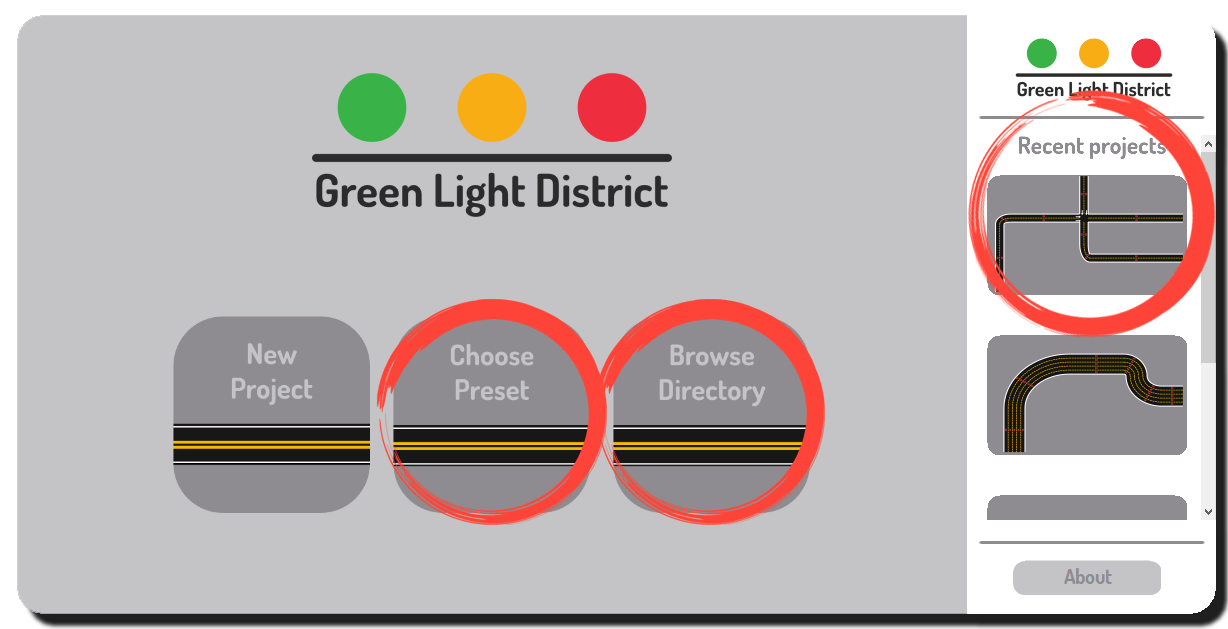
For our user interface we created a new class for each object which we wanted to add. For example the “CurvedButtons” class: We made it a subclass of PictureBox, so we could design our own buttons in photoshop and have it inherit all of its properties so that we don’t have to define them ourselves. In the “CurvedButtons” and “RoundButtons” class we override the “OnPaint” method, to create a curve at the edges of the buttons to make them more circular. Right after we designed our logo we thought that the circles in our logo resembled the title bar, so for a consistent visual identity we removed the border of the form. When the user hovers over one of the circles in the logo it will display its title bar function. We added a panel around the title bar, which allows the user to drag the application. The “About” button will direct the user to our GitHub page for additional information. When the user presses the “New Project” button it will open our build menu. We did this by using multiple user interfaces, which we can hide or show. For each menu we created a method which hides all the user interfaces and shows us the relevant user interface.



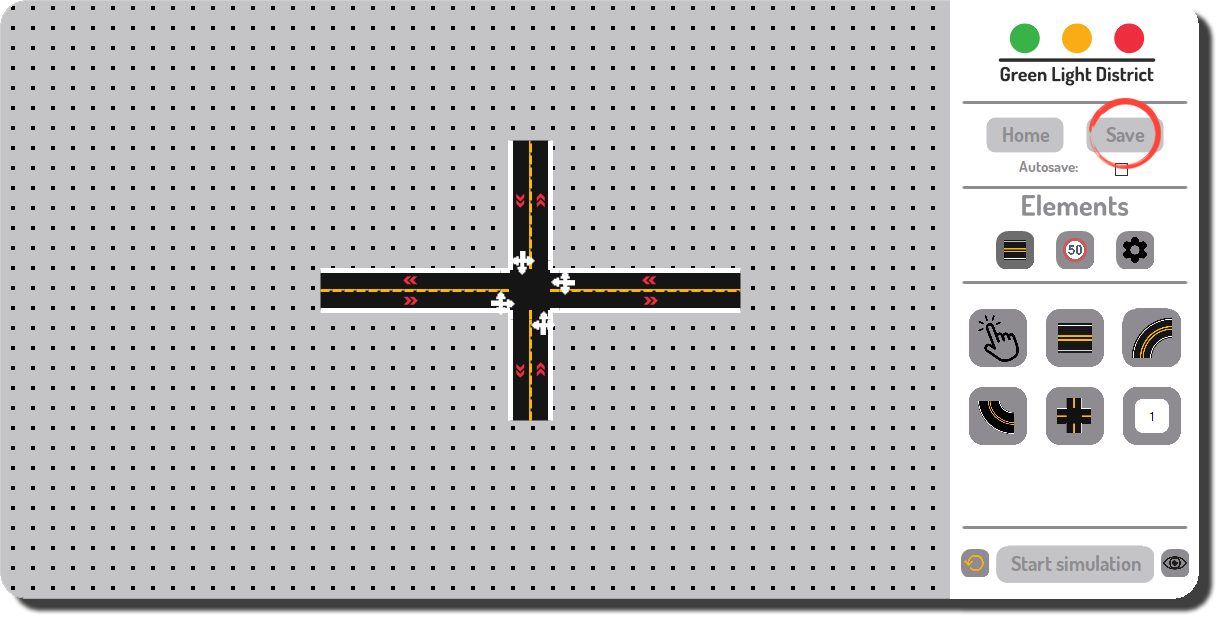
In the Build Menu the user can select the different types of elements that they can use to build their traffic junction. We made a clear distinction between elements, so the design is more accessible for the user. When the user selects one of the three buttons on the top row, it will show a new user interface, with the elements that correspond with its type. The home button closes the build menu and opens the home menu. When the user is done building his junction he can press the “Start simulation” button, which will open the simulation menu.



In the simulation menu the user can change the world settings, which will have an effect on the traffic simulation. Below these settings the user can also check the simulation graphs, which show the amount of brake ticks and the average speed per second of the vehicles. The user can also add new types of vehicles and drivers or edit the existing vehicle’s properties and driver’s behaviour to his desire. These settings can be adjusted using trackbars. We also created a selection box, where the user can select which vehicles and drivers will be used during the simulation. At the bottom of the Simulation’s sub menu, we created the controls for the simulation: the user can control the speed of the simulation with a trackbar, can play or pause the simulation with the green button, can restart the simulation with the orange button or can go back to the build menu with red button.



The “Choose Preset” button opens a menu with a few presets that the user can open. The “Browse Directory” button allows the user to open a project which they have stored on their computer. The “Recent Projects” lists a screenshot of the recent projects of the user.



The Save button allows the user to save a project on their computer. This project will then appear in the recent project area in the homescreen.

The user can build different types of roads. First of all, the user can build a diagonal road. This allows the user to create a road between two points on the grid. The user can also create a curved road, where the first type of curved road allows the user to create two curved roads representing the top of a circle, while the other type of curved road allows the user to create two curved roads representing the bottom of a circle.

The next type of road is a crossroad. This crossroad can connect to other roads at four different points, and the user can create lanes between these points. An example of a crossroad is shown above, in the middle of the four diagonal roads.

All of these types of roads can have 1 up to and including 5 lanes.

From the signs menu, the user can add different signs to the roads. These signs can be ‘read’ by vehicles driving on that road. The vehicles can then react to these signs. For example, when vehicles are approaching a stop sign, they will brake in advance so they stop at the stop sign.

The third menu allows the user to see the driving lanes in the builder and the simulation screen, which are represented by yellow lines in the middle of the lanes. In this menu, the driver can also enable or disable the display of all the points on the grid.

Roads

A road can be added by clicking and dragging with a road making tool selected. While dragging the contours of the road will appear green if the road can be placed and red if it can not.

Every road consists of a list of points that the driver can navigate from and to. These points are quite specific, and have an angle in which a car has to position itself when it drives over it. We achieved this by creating a LanePoint and a DrivingLane class.

In our project we want to be able to use different types of roads to make a road junction. We have three different road types; Diagonal-, Curved- and CrossRoad. Because these roads have a lot in common, we have made an abstract road class. In this class it is defined that roads always have a starting point, an ending point and a number of lanes. The starting and ending point of a created road are the starting and ending point of the rectangle. In this class a list of LanePoints and a list of Signs are declared, but they are not yet assigned. All roads have two booleans that become true when they are connected to other roads, either at the beginning or end. They also save the connected roads in two AbstractRoad variables.

An abstract method for calculating the actual driving lane is added. This is abstract because every road has a different kind of driving lane, and so the calculations for these different driving lanes differ. A static method to calculate the angle between two points is added.

The diagonal road inherits from the abstract road class. The method for calculating the driving lane starts with assigning the list of LanePoints. In this case, the actual begin- and endpoint are the same as the begin- and endpoint of the dragged rectangle, so there is no need to recalculate those. The slope between begin- and endpoint is calculated, and a new direction integer is declared and given a calculated value 1 or -1, depending on if the X direction of the road is positive or negative. In a for loop, every point between the begin- and endpoint is calculated, made a Lanepoint and added to the list of Lanepoints.

The curved road inherits from the abstract road class. There are two types of curved roads, one with its curve towards the bottom, the other with its curve towards the top. A string that gives the direction of the road is added to represent what curve needs to be made. The method for calculating the driving lane starts, once again, with assigning the list of Lanepoints. As is the case with the diagonal road, the actual begin- and endpoint are the same as the begin- and endpoint of the dragged rectangle, so there is no need to recalculate those. A null point is declared. With the direction string and the begin- and endpoint the null point is determined. The distance between the X and Y values of both points are calculated. With a tuple of the X and Y direction of the road (both can be 0, 1 or -1), calculated from the actual begin and endpoint, the difference in X and Y, the null point and a for loop, every Lanepoint can be calculated and added to the list of Lanepoints.

The crossroad differs from the other two types quite a bit. They can have up to four sides that can connect to other roads. When placing a CrossRoad, a pop-up menu immediately opens and shows a few green points on all sides. Each side has a number of Connectionpoints that is equal to the amount of lanes the Crossroad consists of. The user can select a Connectionpoint, after which it becomes blue. Then the selected Connectionpoint can be linked to others, creating a Connectionlink. After the user is done in the pop-up menu, a Crosslane is calculated for every Connectionlink and the crossroad is finished.

When a road is clicked with the selection tool, a pop-up menu opens. In this menu the direction of every lane can be reversed and the road can be deleted.

Signs

A sign can be placed by double-clicking on an existing road with a sign tool selected. Signs provide the traffic rules in the simulation. There are four different signs; speed, stop, priority and yield. These all inherit some basic features from the AbstractSign class. After placing a sign, a pop-up menu opens where the user can flip the direction of the sign and can change the speed limit if it is a speed sign.

The speed sign gives all drivers on the road a speed limit. While driving past it, drivers will change their speed accordingly. The speed limit can be changed between 30 and 130 km/h in the pop-up menu.  
The stop sign makes drivers that drive toward it stop for a while.  
The priority sign gives cars that drive past it a higher priority level on the next crossroad, meaning drivers coming from different directions have to wait for them.  
The yield sign gives cars that drive past it a lower priority level on the next crossroad, meaning their drivers have to wait for cars coming from different directions.

Vehicles

For the vehicles that we want to be driving on our designed crossroads, we have made a vehicle class. In this class there is a constructor method to create a Vehicle object. The vehicle object has a VehicleStat (the VehicleStat has a name, a weight, a length, a maximum speed that the vehicle can reach, motor power, a resistance coefficient, an occurrence and the front surface of the vehicle), a starting node and an AI. The constructor method of the Vehicle calls a method in its AI, which sets the vehicle in the right place.

Every Vehicle has an update method that is called from the Simulationcontroller every 16 ms, which calls a method that changes the speed when braking or accelerating. After that, it calls a method to update the location of the Vehicle, based on its speed and direction.

The Vehicle class also has a draw method. This method is called in the simulationController to draw the Vehicle. A method to change the currentRoad and currentlane of the vehicle can be called from the AI. A method that creates the hitbox of the car is called from the driverprofile controller. The vehicle also has a method to delete itself, which is used when the AI has reached its destination.

AI

The idea is that the vehicle doesn’t do anything on its own, except moving at the speed it is at. All other actions are called by the driver. The driver can react to its surroundings and control the vehicle.

For the driver, we created a class, called AI, with a constructor method that creates an AI object. This AI object is constructed with an DriverStats (a DriverStats has a name, a reaction time, a follow interval (how much distance it wants to keep from the vehicle in front of it), a speedRelativeToLimit (if it wants to go slower or faster than the speed limit, and by how much), a rulebreakingchance and an occurrence.

The AI also has a GPS that it calls to get its next paths (read more about this in “Pathfinding”). It has a method to set the Vehicle, that is called from the Vehicle, that uses the startpoint of the Vehicle to construct a GPS using itself and the starting node of the Vehicle. It also calls a function that sets up the first path and road for the car to drive. This function calls a function to force the car on the right location, a function to change the angle of the car to match the angle of the road and a function that checks whether the car needs to switch lanes on this road, to go to the right direction on the next road if that road is a crossroad.

The AI has an update method that is called from the simulation controller that calls a method that checks the distance from the vehicles in front of it, and if this distance is too small, sets a bool to true that the it wants to switch lanes, and a bool to true that it’s close to other cars.

The update method then calls a method that checks if there is a crossroad coming up, and then a method that checks if the crossroad is empty, if it’s empty and it’s the turn of the driver or the car is already on the crossroad, then it will keep or start driving. If it isn’t empty, tell the vehicle it needs to brake.

The update method then calls the needToBrake method, that checks the booleans for needing to brake, and resets them to false if the car isn’t driving. The same is done with a method that checks the accelerating booleans, and sets them to false if the car is driving faster or the same speed as the target speed.

The update method then calls a method to collect data for the graphs in the data menu, and a method that is called when the wantsToSwitch bool is true, and that checks if a change of lanes is possible, and changes the lanes if it is.

The AI also has a method that changes the target speed of the vehicle. This method is called by speed signs. It also has a method that changes the priority of the AI that works approximately the same way.

It has methods that switches roads when on the end of the road, and a method that changes the current point in the drivinglane it’s on. This method also calls a method that checks for signs for that point of the road.

The last method is a method that can be called by the gps when the driver has reached its destination. It calls the method of the vehicle to delete itself.

Pathfinding

For the pathfinding of the AI, we have made a class called GPSData, which uses the list of roads from the roadcontroller and the properties of these roads to create knots (a point where two roads connect, and the two roads it connects), links (one for every lane between roads, it has a start knot and an end knot and the index of a lane), nodes (one for every knot, that holds the knot, and all the links that have that knot, plus a boolean for being able to spawn and a list with other nodes that have knots that lay on the same road as it), paths (one for every direction a road can go, which has that road, a list of lane-indexes possible on that road, a list of lane-indexes possible on the next road and a list of tuples called laneswap (which holds the two lane indexes that connect, to use when changing roads)).

The GSPData then uses the methods in the OwnDijkstra class to calculate for every starting and ending node (the nodes that can spawn), the route of the least nodes from start to end.

All these routes are added to a List of NodePaths (NodePaths have a starting node and a list of nodes that need to be visited). Every instance of the betterGPS the GPSData, and has a method that the AI that has the GPS can call to get the next path it should take.

Hitboxes

Every road, drivinglane and sign have their own hitbox. These are calculated and created while placing roads or signs. Hitboxes are used to check if roads or signs are clicked upon, to make sure roads can not overlap and to see if cars are driving in an area that is affected by a sign. There are two types of hitboxes, that both inherit most of their features from the Hitbox class; RectHitbox and CurvedHitbox.

Curvedhitboxes are only used for curved roads and their drivinglanes, while Recthitboxes are used for diagonalroads and their drivinglanes, crossroads and signs. Every road hitbox precisely covers it’s road, and the same goes for drivinglane hitboxes. Sign hitboxes only cover a part of a road.

To show hitboxes on screen, there is a toggle button in the roadbuilder screen.

In conclusion

It was 3 months ago now that we started this project. Now that it is (almost) finished, we have a product that everyone in our team can surely be proud of. Of course, there are still some parts from our original analysis & design document that have been scrapped or are still unimplemented. Road markings, interchanges, weather and even gravity were all sent to the chopping block in favour of refining the core of our project.

Yet, there was also a lot that went well. Our visual identity was set in stone from the very start, when we made a video for our pitch. It turned out to be a great asset in the construction of our project, as it laid out a clear goal for our team to achieve. The general structure of our project layed out in the document was kept nearly intact throughout, with separate AI and vehicles, a drag and drop system for roads and a system that connects them. A pathfinding algorithm, which was removed from roads and instead given to the AI, the saving and loading of projects and a tab for adding vehicles and drivers. All these features and more were implemented, from which one can only conclude that we did indeed solve the problem we posed.

Some of us did indeed learn their lesson, and will start work on their next project a bit earlier.