João Silva 98737, Rafael Henriques 89530, Tiago Jacinto 87127

Using the Unity profiler tool, we studied the performance of the code. We enabled the deep profile configuration in order to see the computation time used by all functions. Each performance analysis was done with 10 vehicles at the same time in the roundabout scene.

Priority and blending movement algorithms

After analysing the performance while running the priority and blending movements we concluded that the task that was having the largest computational time was the Collider.Raycast method present in the Dynamic Avoid Obstacle algorithm. (Attachments “Blending.png” and “Priority.png).

To try to optimize the code we tried adding conditions that avoided casting additional Raycasts. In order to do that, instead of calculating the collision with the three rays (central ray and small whiskers) we checked if there was a collision, one ray at a time, and only tried to check for collisions with the next ray if there was no collision with a previous one. Since the Obstacle Avoidance algorithm is of complexity O (1) and therefore impossible to lower to the complexity, we could not find any more optimizations. After running Unity profiler again, the results shown were the same, showing that the optimizations were not relevant enough to have significant results.

RVO movement algorithms

Analysing the performance of the RVO algorithm lead us to conclude that the method that was taking up the most CPU time was GetBestSample because it has to iterate over each sample velocity, each obstacle and each character. Within this method, Collider.Raycast was still the most computationally intensive task as seen in attachment “RVO_no_optimizations.png”,

The first way we tried to optimize the code was by avoiding unnecessary loops in four cases. The first one is when calculating a sample penalty, if we come across an obstacle or character that will collide the penalty will be at its maximum and therefore it is unnecessary to continue calculating the sample penalty for other characters or obstacles. The second case is when the algorithm comes across a sample penalty of zero, in this case there will not be a better velocity and we choose that sample. We decided that the time spent continuing the loop to find another sample with penalty 0 that is faster was not worth the performance loss. The third case is when checking if the time penalty is larger than the maximum time penalty. Since the maximum time penalty starts at zero, we only have to check if the current time penalty is larger the maximum time penalty if the time penalty is bigger than zero. The fourth case is when a time penalty is equal to infinity (in our case float.MaxValue). Once we find a time penalty that is infinity there is no need to continue iterating over the characters or obstacles, so we skip this sample.

To reduce the amount of calls to Collider.Raycast, we found that we could ignore obstacles that are too far away. However, there is a problem, some of the obstacles have different sizes and shapes and some are larger than others, which means we can’t simply calculate our distance to their center. Our solution involves calculating the obstacle extent (distance from the furthest away vertex within the obstacle to its center) plus an arbitrary ignore distance. If our distance to the center of the obstacle is larger than this value, then we can safely ignore it. This is represented in figure 1.

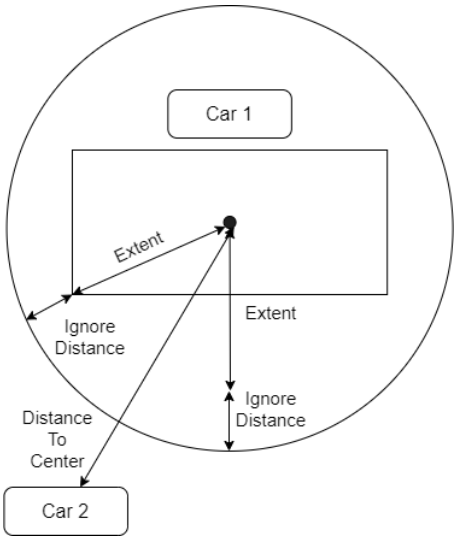

Figure 1 – Illustrative example of our approach to ignoring obst

In this situation Car 1 cannot ignore the obstacle since its distance to the center of the obstacle is smaller than the extent + ignore distance while the Car 2 can ignore the obstacle because its distance to center is larger than the extent + ignore distance.

After performing all these optimizations, the algorithm showed an improvement in performance which can be seen in the attachment “RVO_optimization.png”. Before the algorithm was running between 30 and 60 fps and now it is always above 60 fps.

Micro optimizations

By using the Unity profiler we also noticed that we still had some space for improvement in the performance of our code which lead us to do some micro optimizations.

The first one being that every time we had to compute the magnitude of vectors we computed the squared magnitude to avoid the computation of a square root.

The second one was that all operation and variables that were repeated every loop without changing their result were put outside of the loop so that we don't have to compute them every loop iteration. For example, instead of creating a Ray object every time we test a sample (because the constructor was consuming a lot of CPU time), we create it outside the loop since we can use the same Ray for each Raycast.

Lastly, we noticed that some Vector3 operators had some impact in the overall program performance due to being in a loop and repeated many times. To solve this problem we replaced these operators with the same operations, but applied to the vector components (A = B + C became A.x = B.x + C.x and so on).

Performance comparison

After comparing the results obtained by using the Unity profiler we concluded that the priority movement algorithm was the most performant of the three followed by blending and in the last place was RVO.