IAJ Project 1 - Performance Report

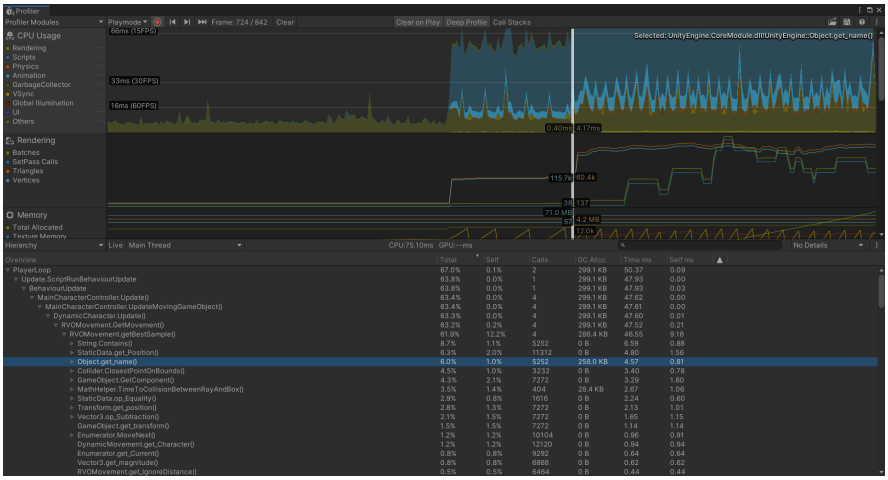
This report has been made in accordance to the guidelines provided for the IAJ's (Inteligência Artificial para Jogos) first project. More specifically this report pertains to the 5th level of the assignment, which consisted in the usage of the Unity Profiler.

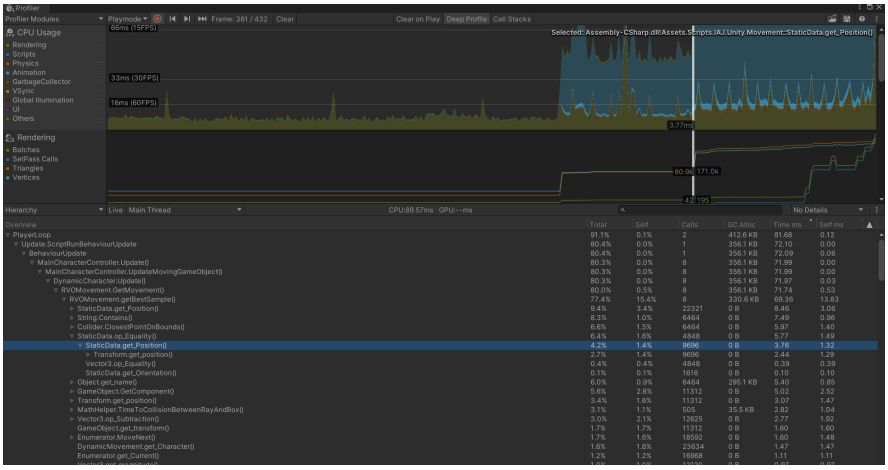
Authors

- Diogo Silva (98776)
- Carlos Marques (98639)
- Rui Melo (98823)

Initial Analysis and Improvements

At first we ran the profiler just after starting the scene *PriorityAndBlending*. The image below shows the results obtained.





We chose to start with the RVO movement selected and spawning 4 cars/second. We can see a clear spike in frame drops which corresponds to when our 4 initial cars were spawned, and then another one when the next 4 cars are spawned and then remains consistent, meaning, the more cars we spawn, the bigger our performance hit is. Looking at our hierarchy we can see that indeed the RVOMovement class is the one eating away at our CPU, with its *GetBestSample* method representing, at times, upwards of 80% of our scene's CPU usage. Outside of this method, we have the *RandomHelper.getRandomBinomial* methods (used to generate samples) being the second largest and spiking at certain intervals, corresponding to the generation of our samples. Within the *GetBestSample* method (which is the one with most optimization opportunities) we have the following tasks consuming the most of our CPU:

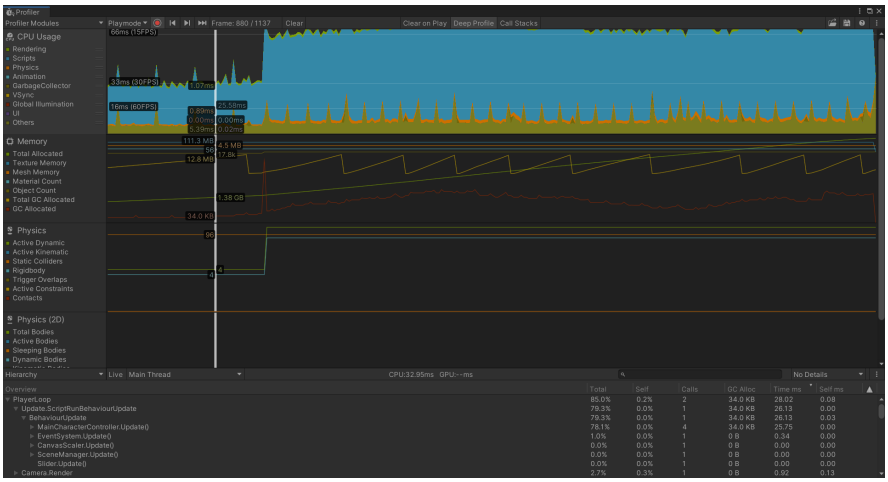
- String.Contains() & Object.getName()
    - This method was being utilized to distinguish between RoadBlocks and Road limits. However, after some delliberation it was deemed that there is no need to distinguish between these two types of objects as they can be both treated as static boxes. This method was then removed. Furthermore it should also be noted that the getName method is also causing a lot of GarbageCollection allocation, which should be avoided.
- StaticData.getPosition() & StaticData.op_Equality()
    - getPosition() is used to get the positions of cars we want to check collisions with and is needed for our algorithms to work, hence cannot be removed or stored (since they're constatly changing). The problem here lies in the unpotimization of the Transform.get_position() which gets called.
    - The equality is used to, when iterating over the characters, check if the character we're looking at is ourselves (since we don't need to check collisions with it). It suffers from the same problem as the getPosition, since it also has to call the Transform.get_position(). However, we could improve it by removing our character from the Characters list upon instantiation of the class.
- GameObject.getComponent() & getPosition
    - The getComponent() method is being used to get the obstacle's colliders, which are then, in turn, used to check whether or not we need to try to avoid said obstacle. Currently, each time we want to check the collision with an obstacle we are calling this method. A way to improve performance would be to instead, store the colliders of all obstacles in a list, created when instantiating the class, and then pull the colliders from that list rather than re-call the method.

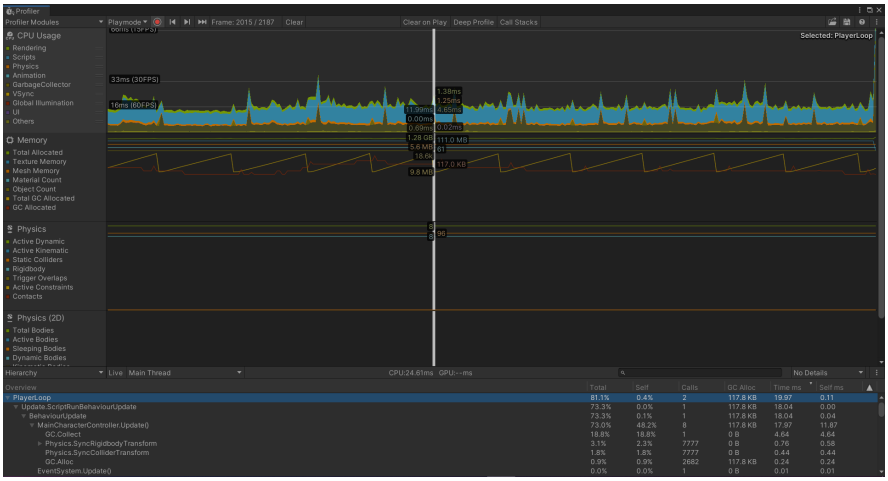
- On the other hand we can also increase our performance by not constantly trying to get each obstacle's position, and instead storing them in a list from which we can then retrieve them. Note that this can be done because the obstacles are static, hence their positioning isn't expected to change.

- Collider.closestPointToBounds()
  - We use this method to get the closest point in the bounds of an obstacle to our car's center point. We need this to then check if the distance between those points is larger or not than the ignore distance

Additionally, we have a bunch of Debug.DrawLines which were used to draw bounding boxes around the obstacles and vehicles we're checking collisions with. While not being the most consuming, they're entirely optional and can be commented out for some performance gain.

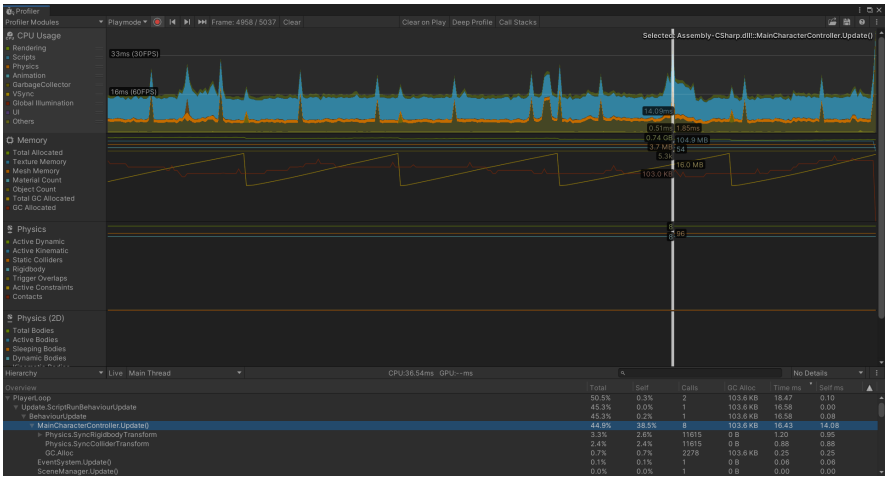
Taking into account that, using the **Deep Profile** option of the profiler (which does slow us down considerably) we were getting frame rates under 5 fps when using 8 cars, and around 20 with 4. With our previously mentioned improvements we managed to constatly nearly double that, staying at an average of 14-15fps for 8 cars, and at around 30 for 4 cars (assertained using both the profiler and the **Stats** functionality). Better results were obtained, obviously, after disabling the Deep Profile option, at which point we were getting well over 30fps with 8 cars as can be seen on the last picture.





Collision algorithms performance comparison

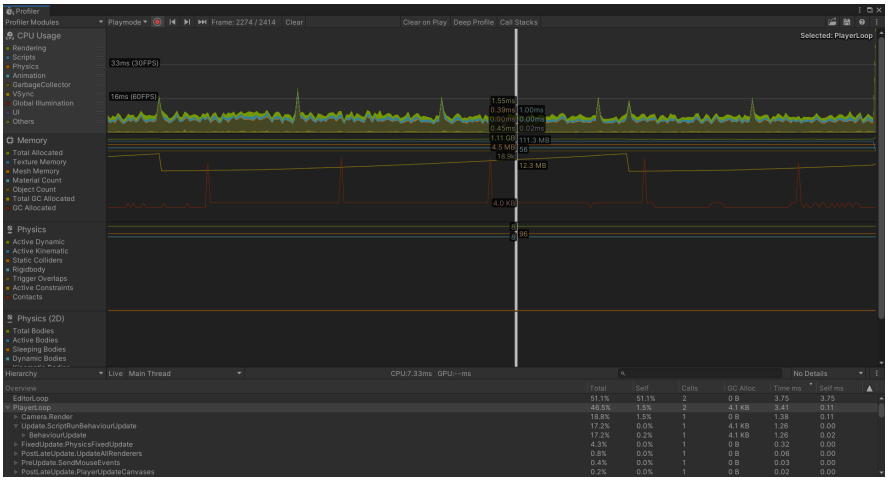
Let us now compare RVO's performance versus the **Priority** and **Blending** movement algorithms. The following image shows the RVO performance which we will be comparing to. This result was taken from the *PriorityAndBlending* scene, after a fair amount of cars had been spawned.



Priority

As can be seen on the picture below, the Priority movement does showcase better overall performance comparatively to the RVO movement, achieving over 60fps on

average, whilst RVO stuck around the 50-60fps. This can obviously be attributed to the fact that the Priority movement algorithm is much simpler than the RVO with much less computations involved. Afterall, the only thing this algorithm has to do is pick from a list of movements, rather than having to generate dozens of samples and pick the best one.



Blending

As for the Blending movement, while it does have slightly lower frame rates, compared to the Priority movement, in the end it still does end up with much better performance results than the RVO, which does make sense since, again, the amount of computations needed isn't nearly as many since all Blending does is get a list of movements and blend them together using weights.

Conclusion

All in all, RVO is the least efficient algorithm in terms of raw performance and required computations, although not by as large a margin as we expected. It shows the best performance in terms of actual obstacle and character avoidance since our characters will rarely collide when using it, but one does have to wander to which extent is the performance tradeoff worthwhile.

Addendum

On Monday, 12/10/2020, we met with the teacher to ask for feedback and possible suggestions. One of them was to replace the way we were detecting if our character was close enough to an obstacle that it shouldn't be ignored.

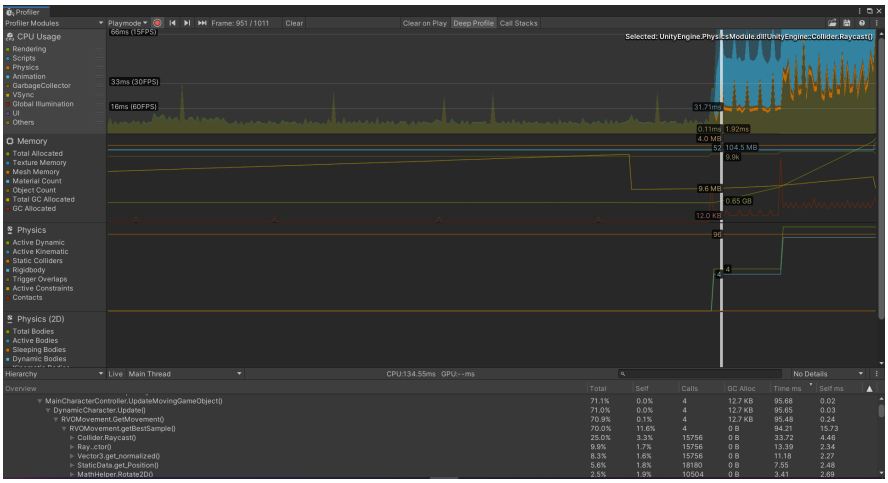
Originally, this was achieved with the code shown in the following image. We detected the closest point to the collider's bounds in relation to our car's center, and checked if the distance between them was smaller than the ignore distance.

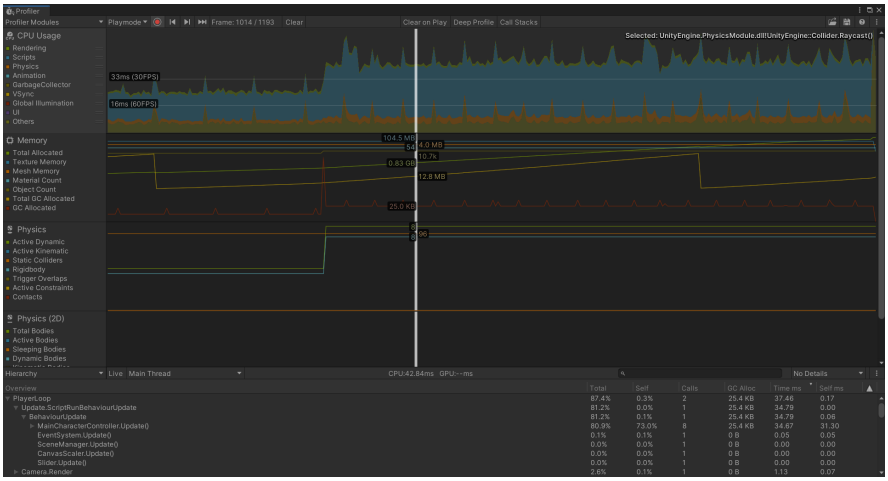
```
// FIRST METHOD OF COLLISION DETECTION (using the closest point on the collider bound
var closestPoint = obstacleCollider.ClosestPointOnBounds(this.Character.Position);
var deltaP = closestPoint - this.Character.Position;
if (deltaP.magnitude > (this.IgnoreDistance)) continue; // Ignore this wall (too far away)
```

What the professor suggested instead was that we utilized RayCast's to detect whether or not a collision was present. As such we replaced the previous code with the following, which implements three rays (a main center one and two whiskers).

```
// SECOND METHOD OF COLLISION DETECTION (using raycasts) - SLOWER but more effective
if (this.Character.velocity.magnitude == 0) continue;

Ray mainRay = new Ray(this.Character.Position, this.Character.velocity.normalized);

Vector3 leftWhisker = MathHelper.Rotate2D(this.Character.velocity, -MathConstants.MATH_PI_6);
Vector3 rightWhisker = MathHelper.Rotate2D(this.Character.velocity, MathConstants.MATH_PI_6);

Ray leftRay = new Ray(this.Character.Position, leftWhisker.normalized);
Ray rightRay = new Ray(this.Character.Position, rightWhisker.normalized);

RaycastHit hit;

if (!obstacleCollider.Raycast(mainRay, out hit, this.IgnoreDistance) && !obstacleCollider.Raycast(leftRay, out hit, this.IgnoreDist
{
    continue;
}
```

Overall, we noted that indeed, our cars were colliding with obstacles less often, meaning the algorithm was more effective. However, due to the fact that we were now having to check for three raycasts, our performance took a heavy hit. We dropped from averages of 60fps to 30fps (at max cars), as can be seen in the following profiler images (the first using the Deep profiler and second without it).





It should be noted that this change was made after the previous conclusions, comparisons and improvements were documented. However, taking into account there is no obvious or direct way to improve the raycast's performance, we believe there wouldn't be much improvement to be done on this front. In the end we opted to go with the teacher's suggestion since it was more effective, but the tradeoff in performance is surely non-trivial and something that should be considered, were this algorithm to be implemented in a non-pedagogical project.