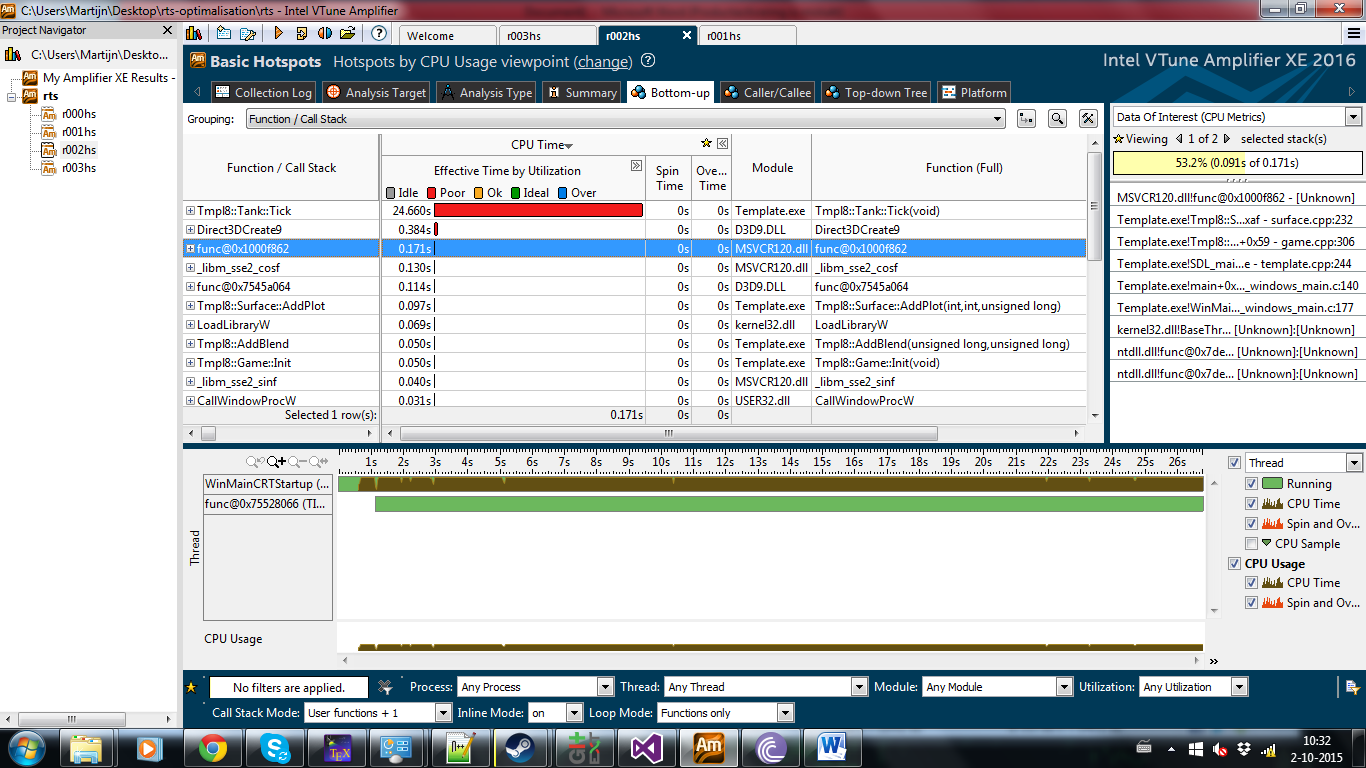
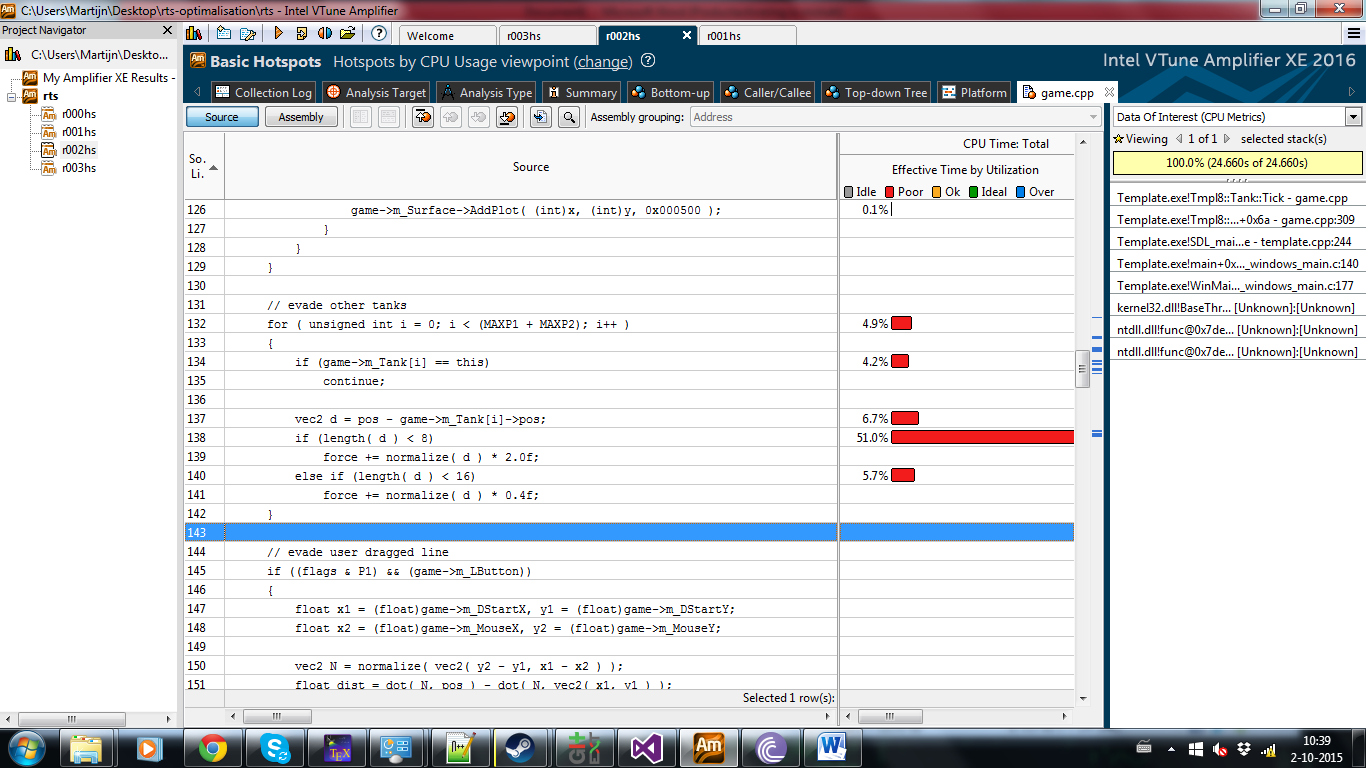
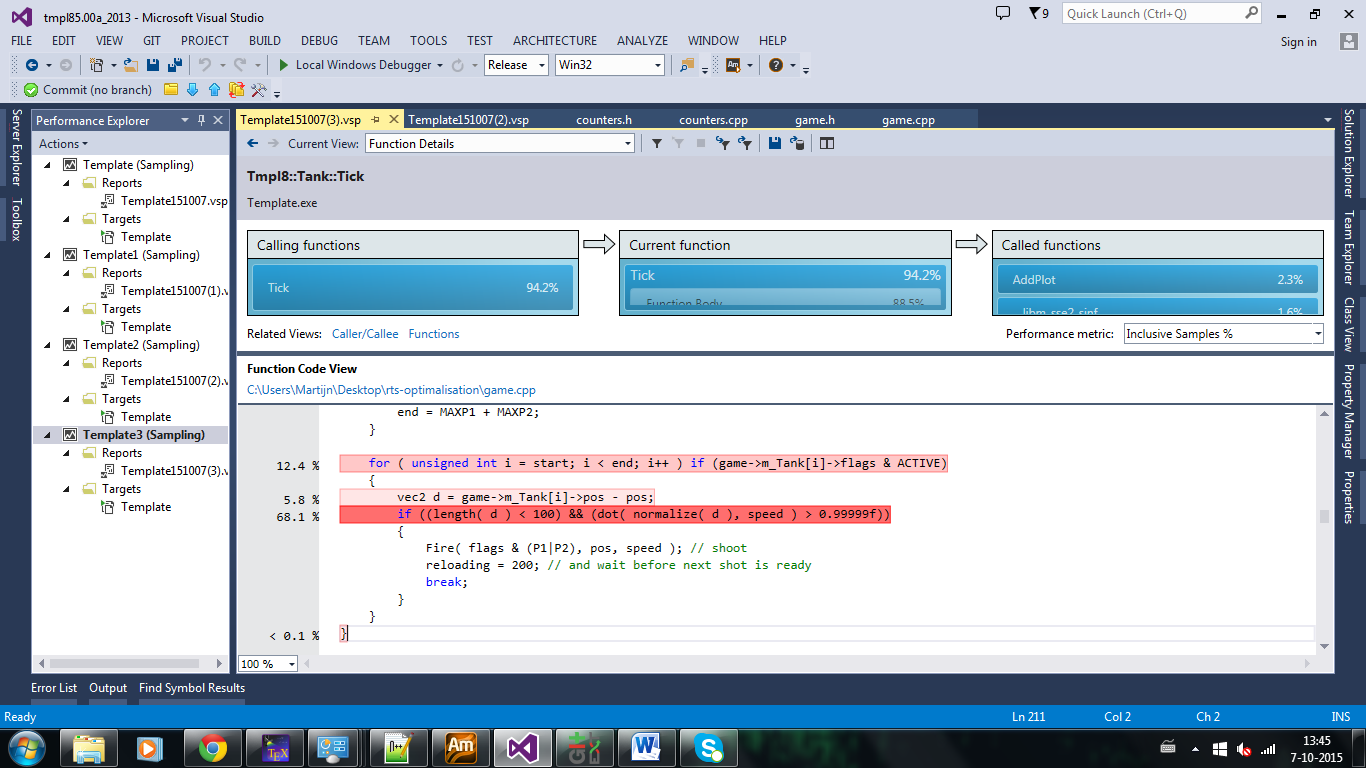
In order to obtain a good starting position for the optimisation process we increased the army size to one thousand blue tanks. We also added and printed a framerate counter that at this stages showed 3 fps. We can now use both the profiler and the fps counter to see possible improvements.

After this we run the application using the vtune profiler. This gave us these results:   
and when looking into the tank::tick function:

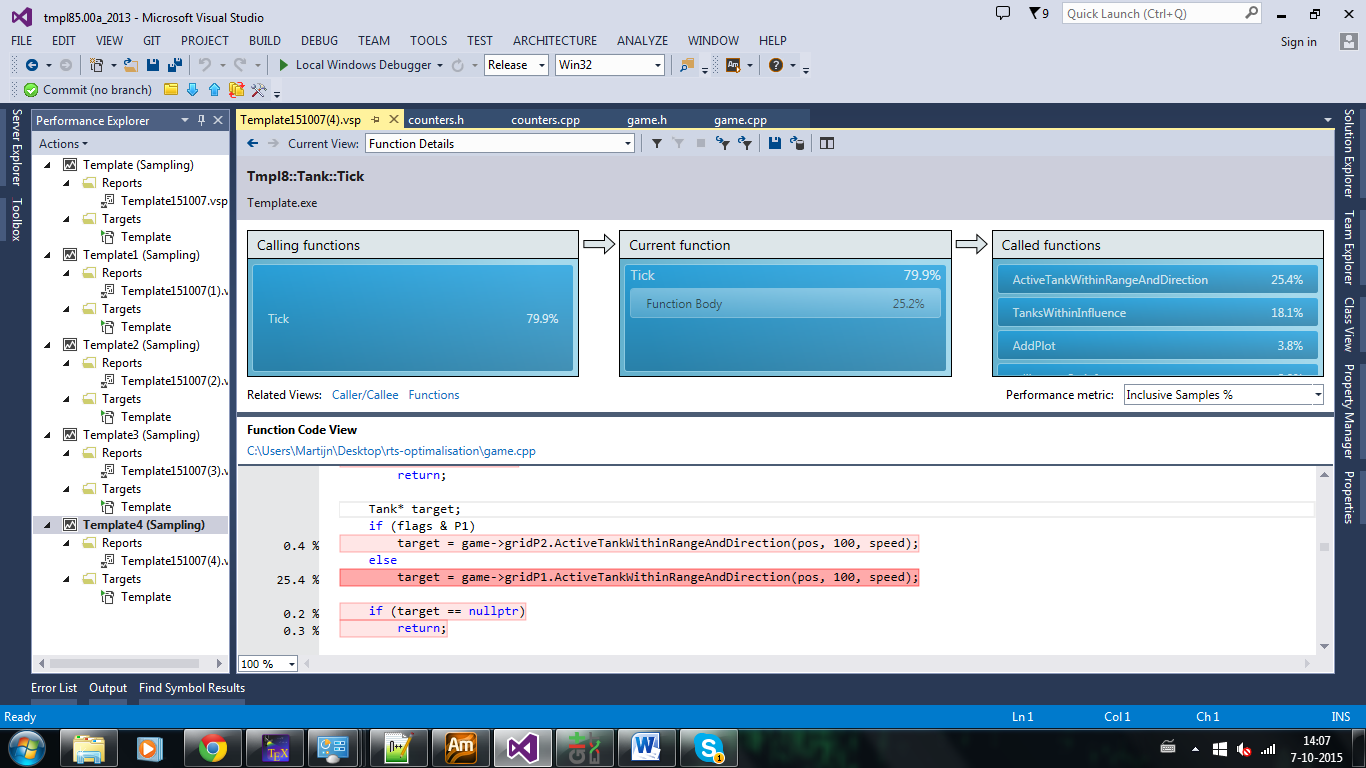
From this information we determined that the evade other tanks loop in the tank::tick function was the main bottleneck of the application. We then looked at the code and after some readability improvements we determined the speed of this “algorithm” to be O(n2) (the tank::tick() funtion loops over all tanks and is called #tank times per frame). A high level optimalisation for this could be to implement a grid into the program such that only one or a few cells need to be checked for other tanks to evade. This would reduce the number of tanks that need to be checked by each tank to a constant number (since there can at maximum be a constant number of tanks in the influence area). This would increase the speed of the algorithm to O(n). This could significantly increase performance and scalability of the application. With this grid we can also easily change the bullet collisions to also use the grid. For this however we will need to create to grids one with all red tanks and one with all blue tanks.

After creating a 16\*16 grid using linked lists to store pointers to the tanks in the cells. We ran the profiler again (with 1000 blue tanks at 32 fps) and got these retults:

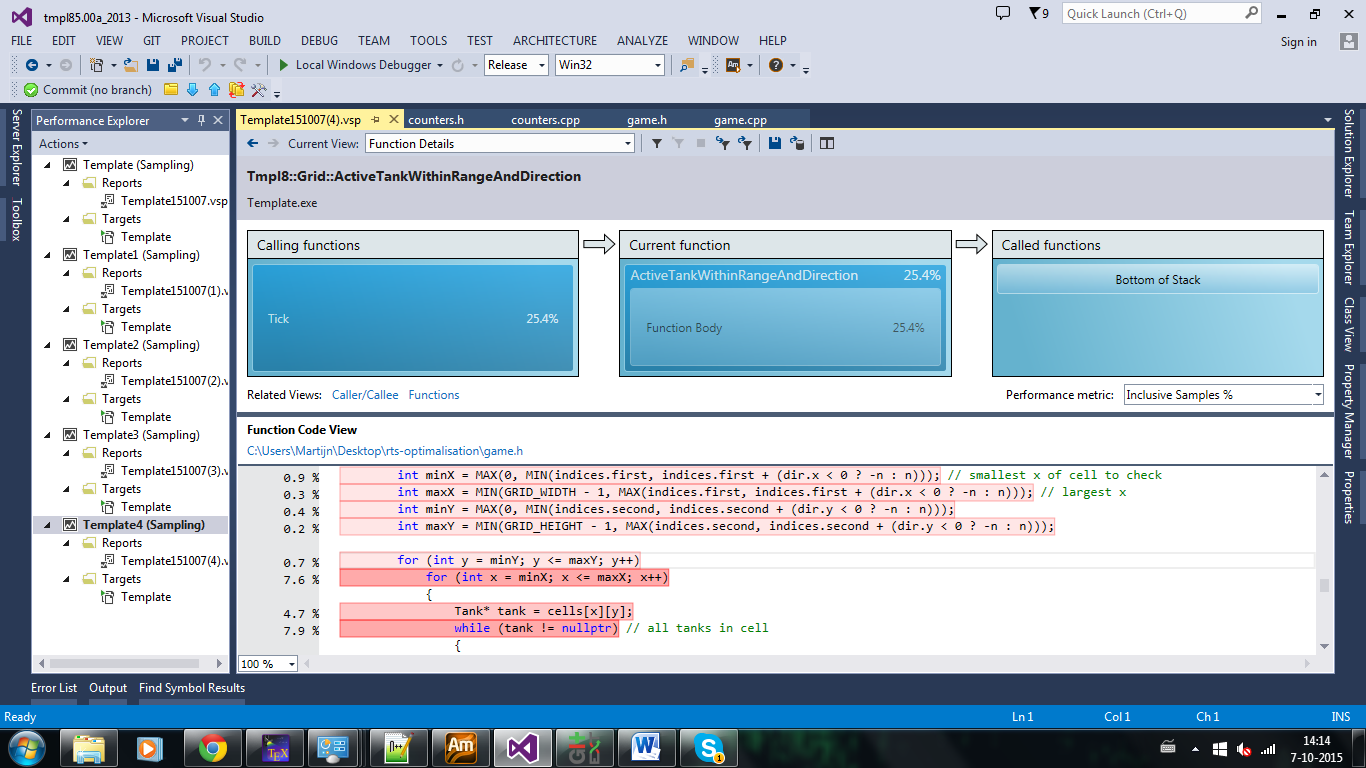


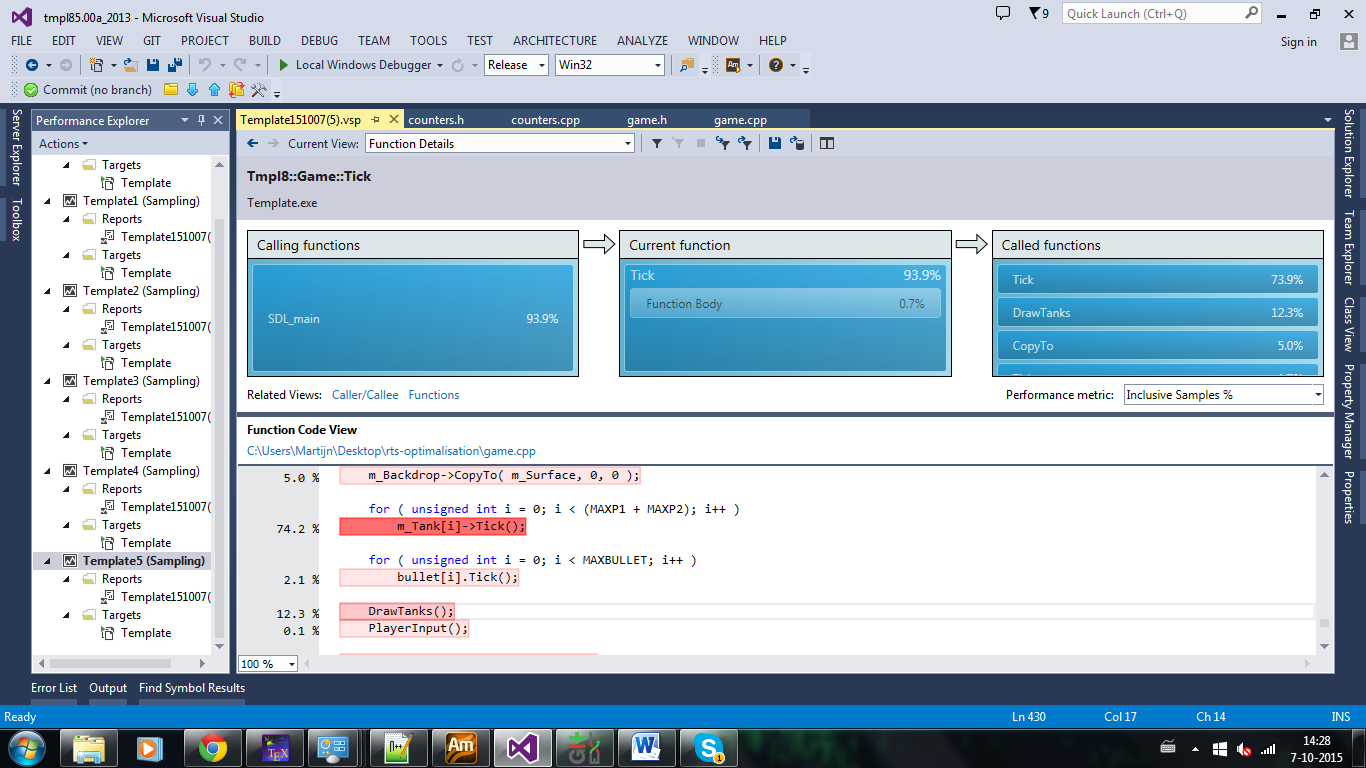
Here you can see that tank::tick() is still the main bottleneck of our application. Inside this function the loop for shooting bullets takes most of the CPU time so that will be the next focus point. The complexity of this part of the application still is O(n2). The solution is the same as with evading other tanks: use a grid which reduces the complexity to O(n).

After applying this optimisation the performance was significantly improved. We can now simulate 30,000 blue tanks at 34 fps. We once again did profiled the application and got these results:



And in the ActiveTankWithinRangeAndDirection function:

 From these results we concluded that although the optimisation of the targeting has significantly increased the performance it is still a large bottleneck for the application. This was probably due to the overhead for empty cells since almost all of the cells are empty all the time. In order to reduce this problem we wanted to create a second 100\*100 grid that can be used for the target selection instead of the smaller grid. This will significantly reduce the overhead because less cells need to be checked.

After applying this optimisation we can simulate (INSERT DATA). We then profiled the application again. This gave the following results: 

From this data we concluded that Tank::Tick() still uses most of the CPU time but now DrawTanks() also uses a significant portion of the time. We also figured that the draw code can be sped up significantly by only drawing tanks that are on the screen. For this we can use the 100\*100 grid used for the target detection. This will reduce the complexity of the drawing from O(n) to O(1) (assuming that there can only be a constant number of tank on the screen).

After applying this optimisation we can simulate (INSERT DATA). We then ran the profiler again and got this result: