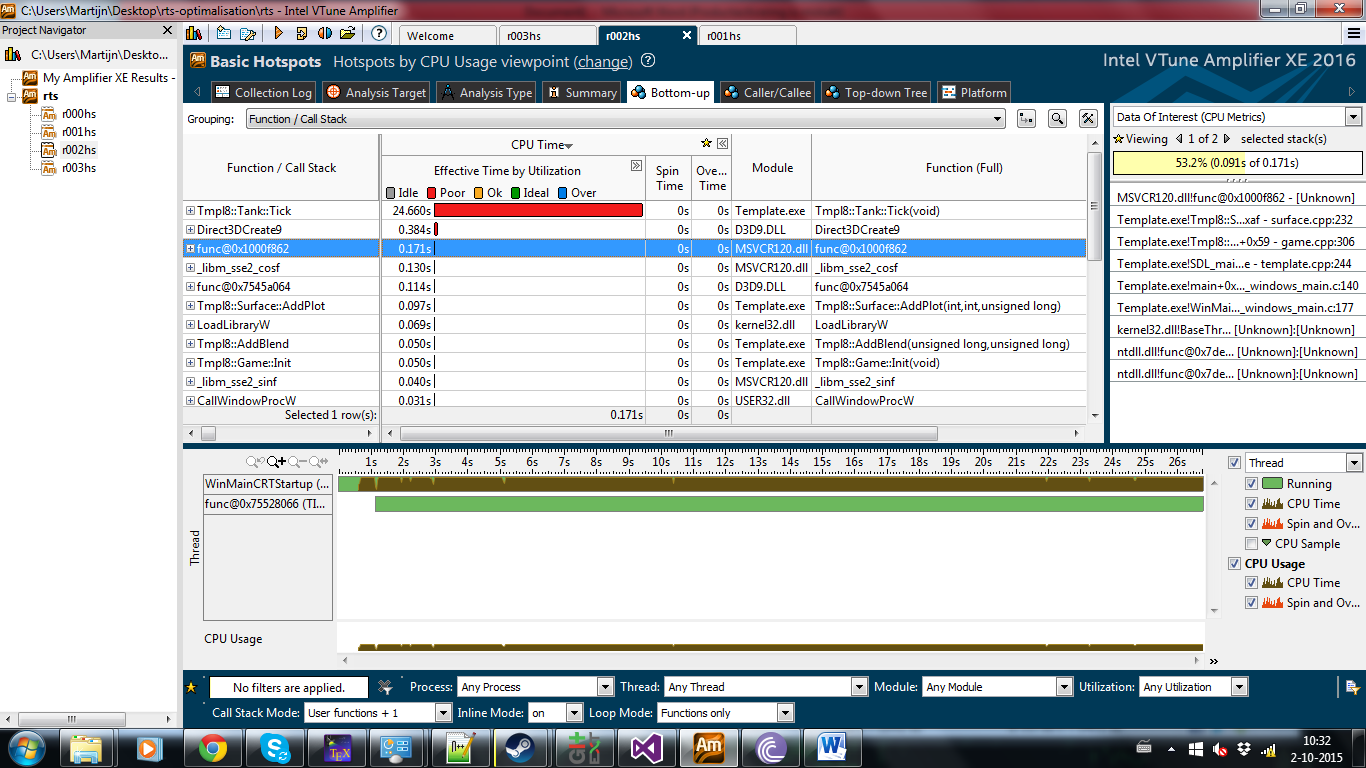
# Assignment 2 “RTS”

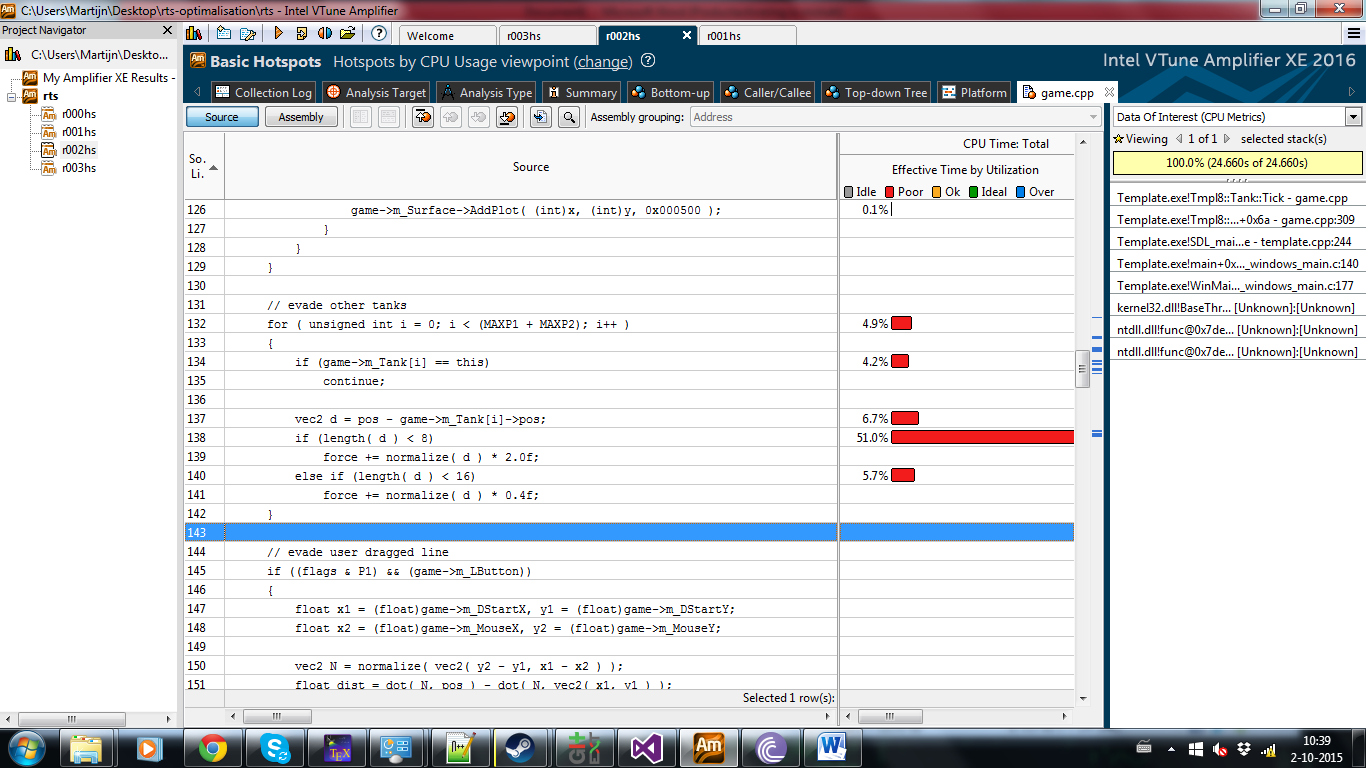
Jordi Vermeulen (3835634)

Martijn Koenis (3770214)

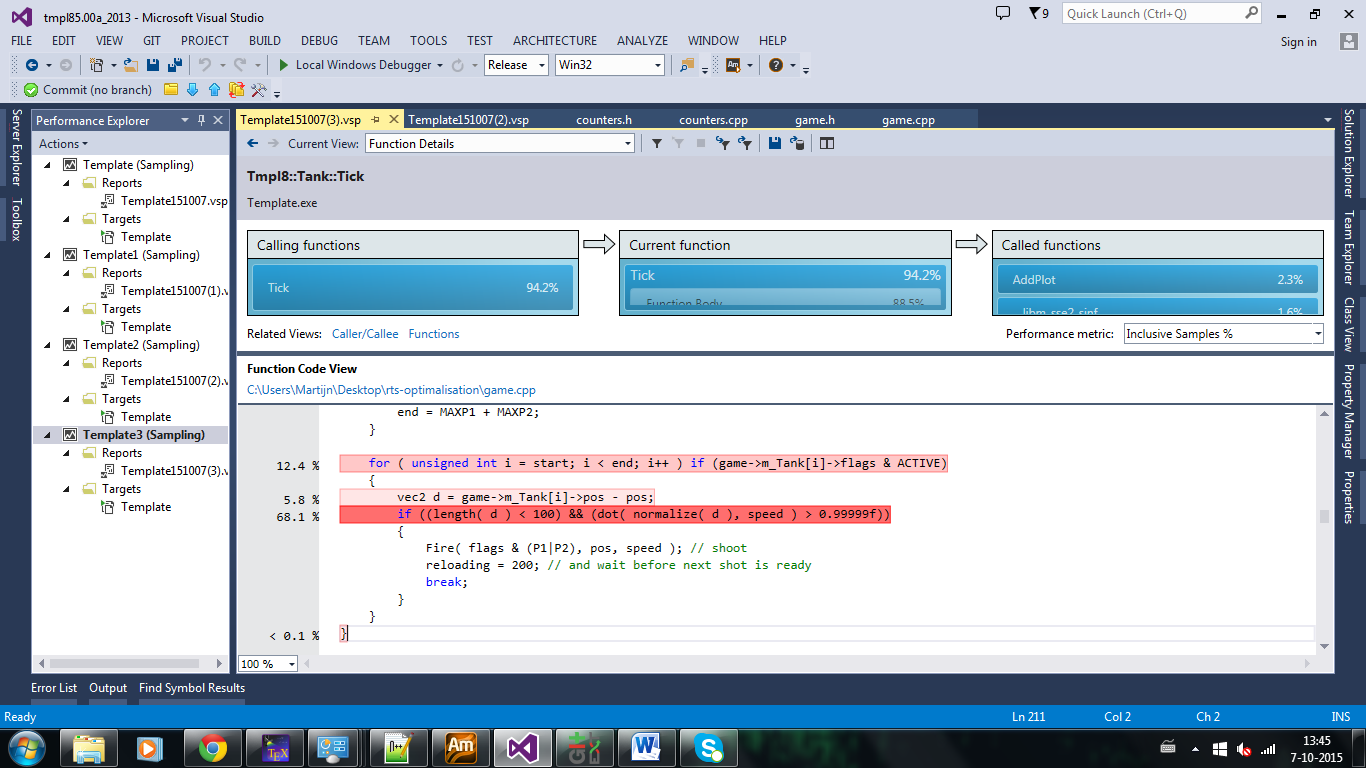
# Improvement process

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 frame rate counter that at this stage 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:

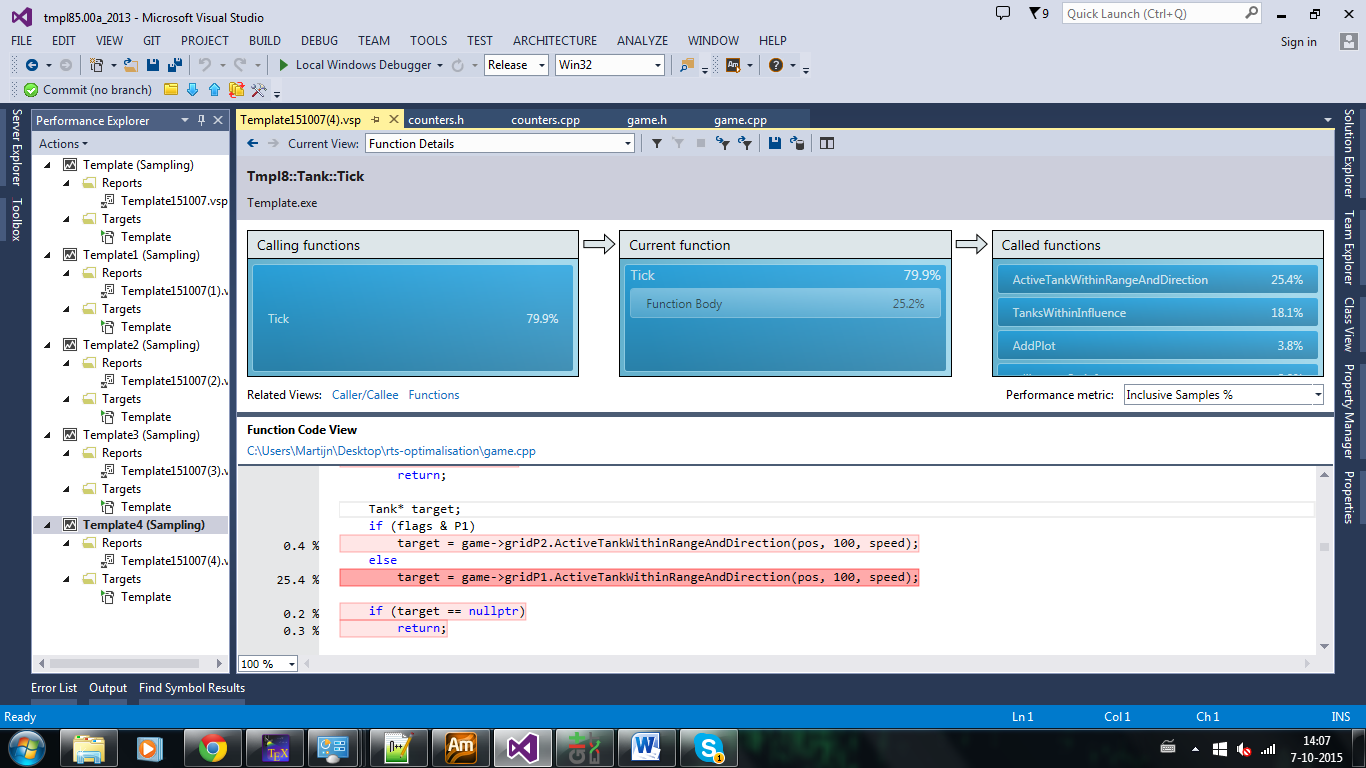
From this information we determined that the evade other tanks loop in the tank::tick function was the main bottleneck of the application. We then looked at the code and after some readability improvements we determined the speed of this “algorithm” to be O(n2) (the tank::tick() funtion loops over all 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 visual studio profiler (with 1000 blue tanks at 32 fps) and got these retults:

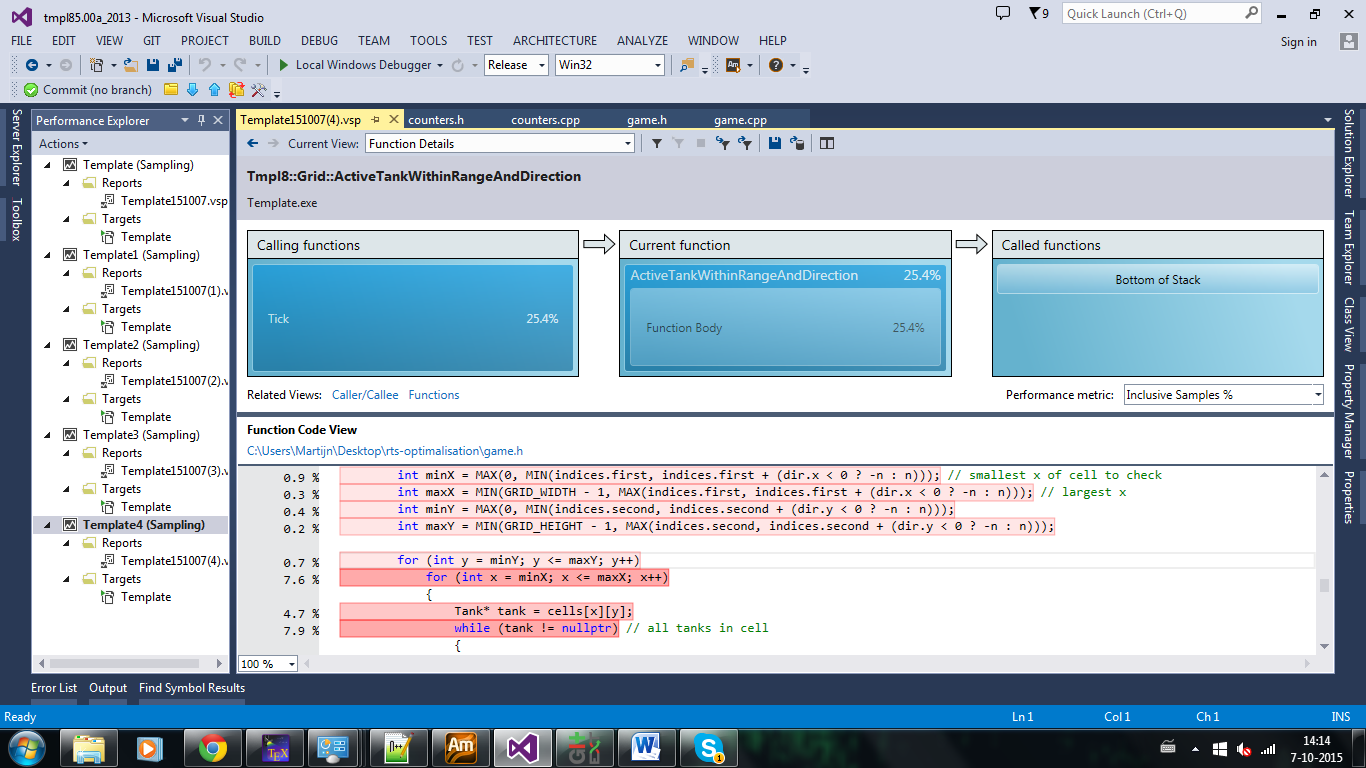


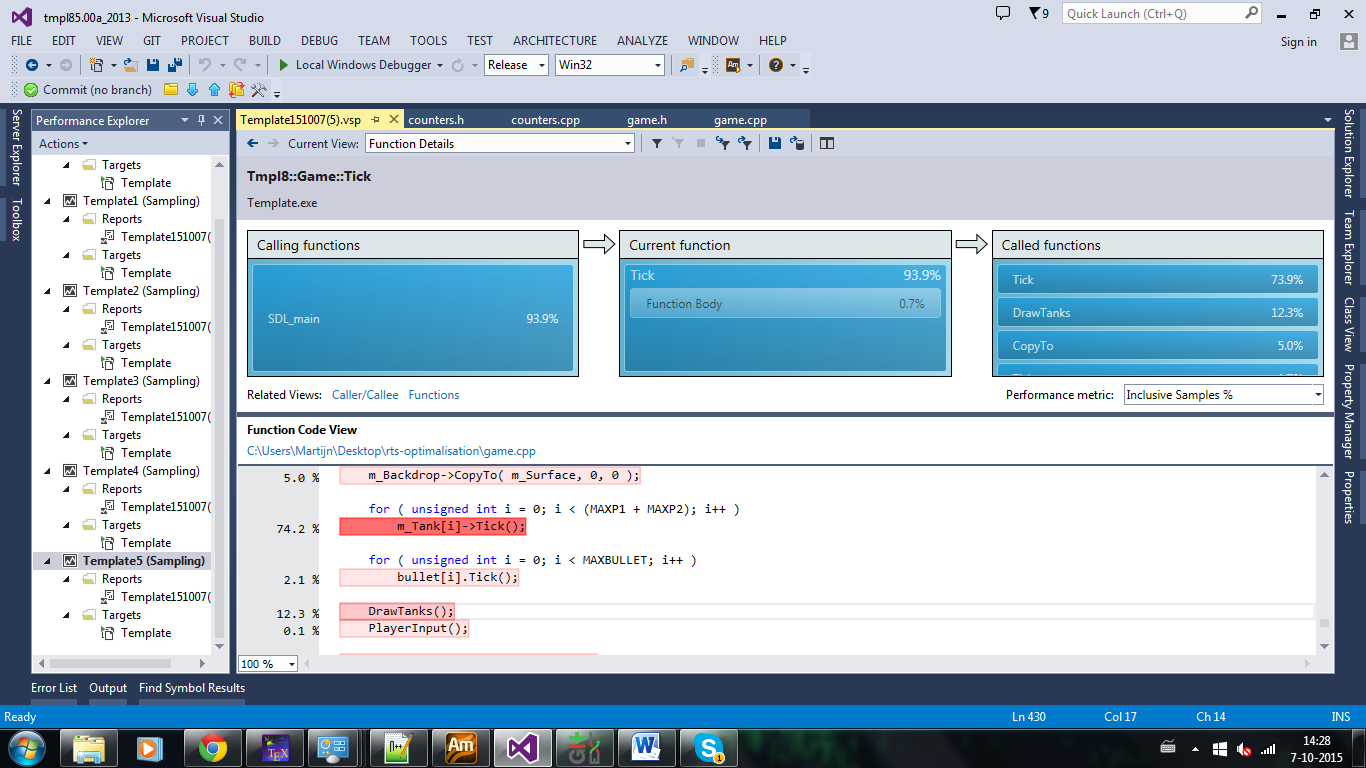
Here you can see that tank::tick() is still the main bottleneck of our application. Inside this function the loop for shooting bullets takes most of the CPU time so that will be the next focus point. The complexity of this part of the application still is O(n2). The solution is the same as with evading 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 a profile the application and got these results:



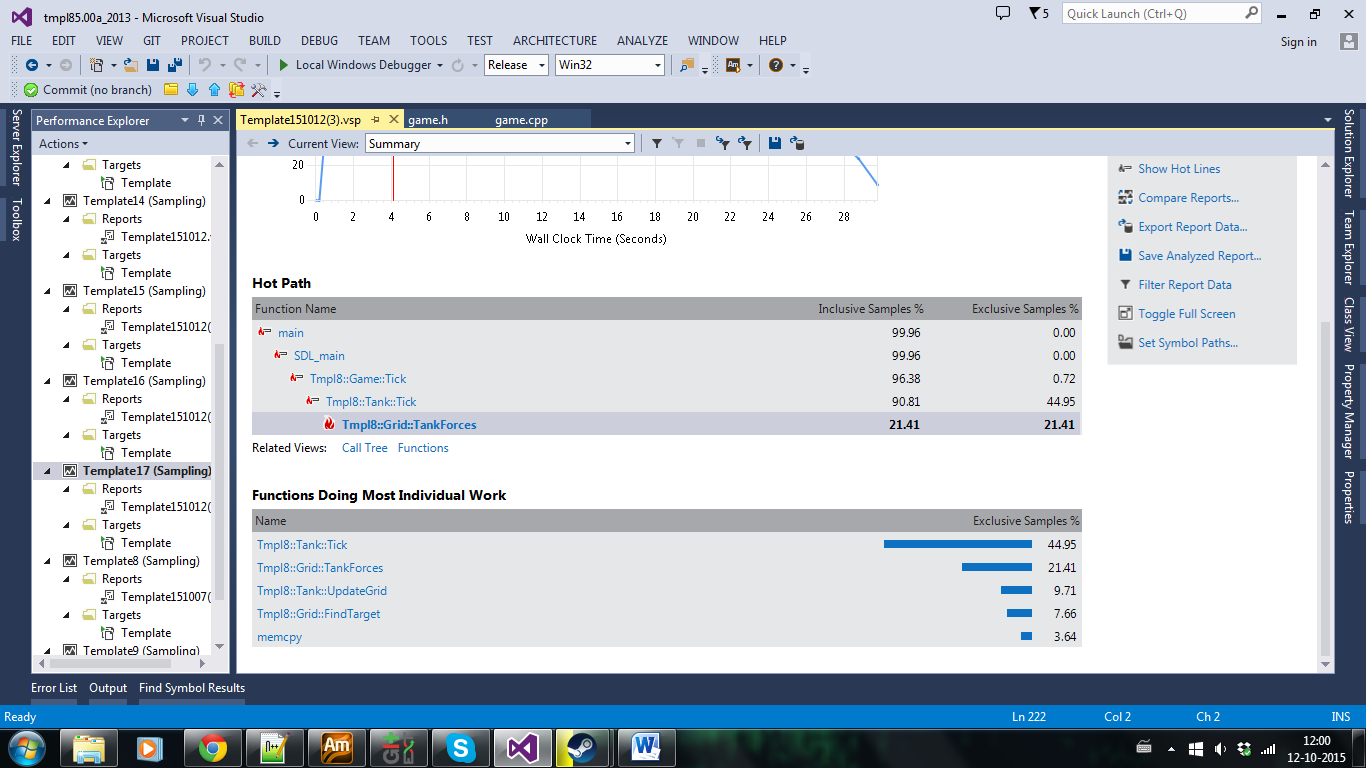
And in the ActiveTankWithinRangeAndDirection function:

 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.

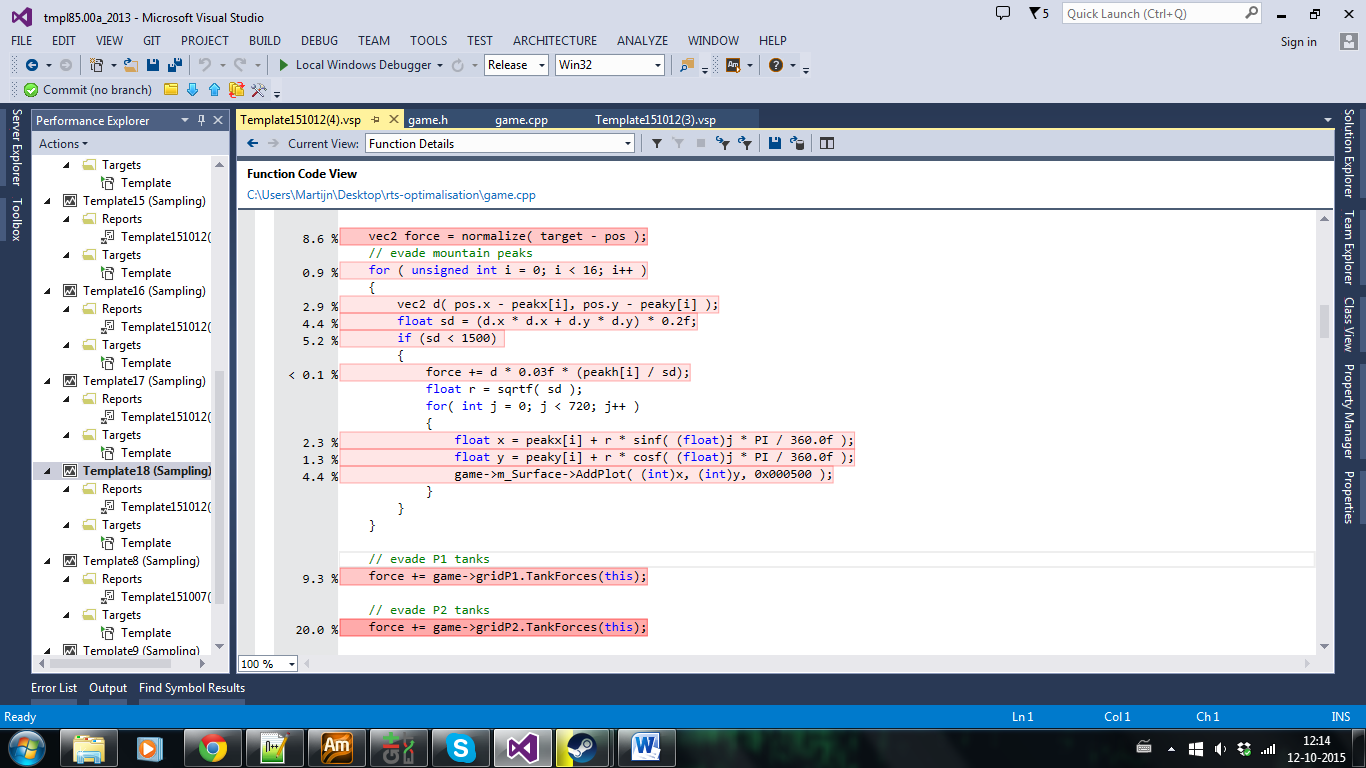
After applying this optimisation we can simulate 40k blue tanks at 30 fps. We then profiled the application again. This gave the following results: 

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).

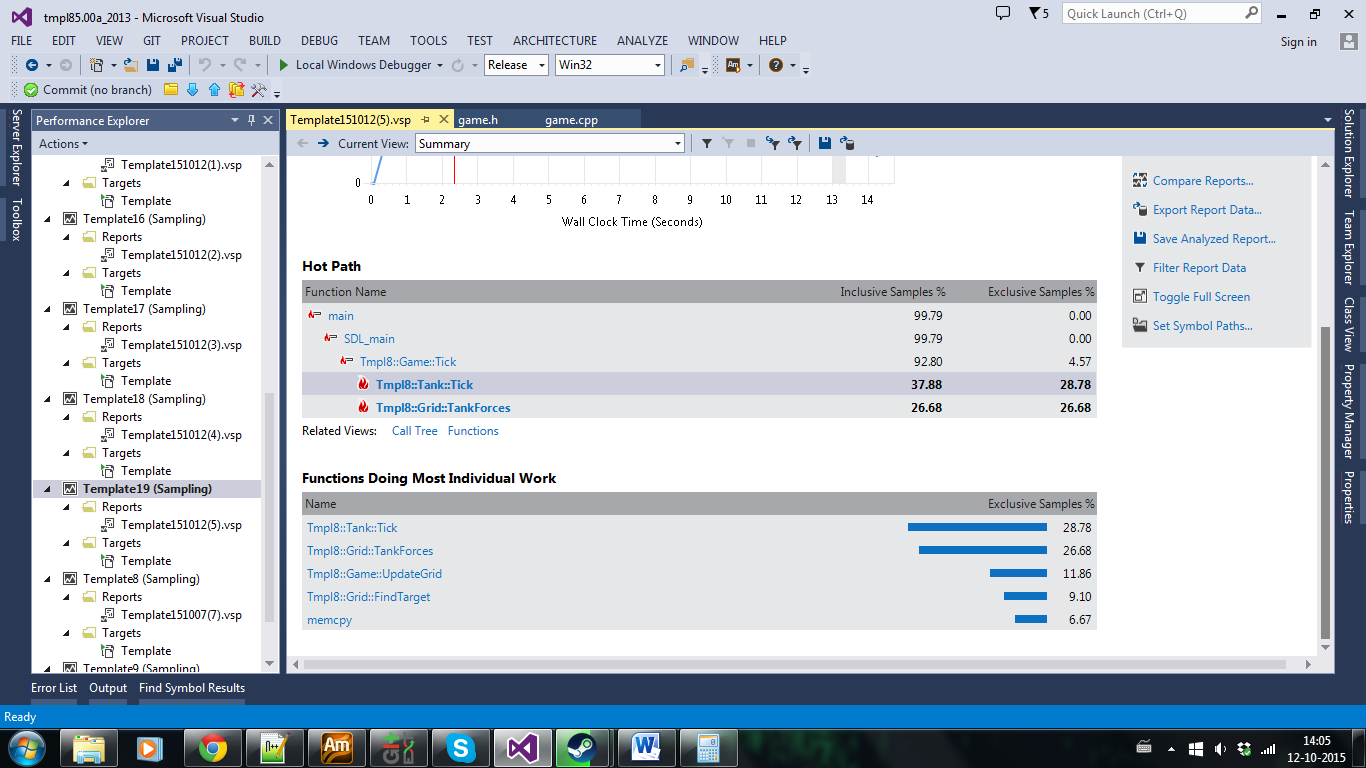
After applying this optimisation we can simulate 40k blue tanks at 37 fps. We then ran the profiler again and got this result:



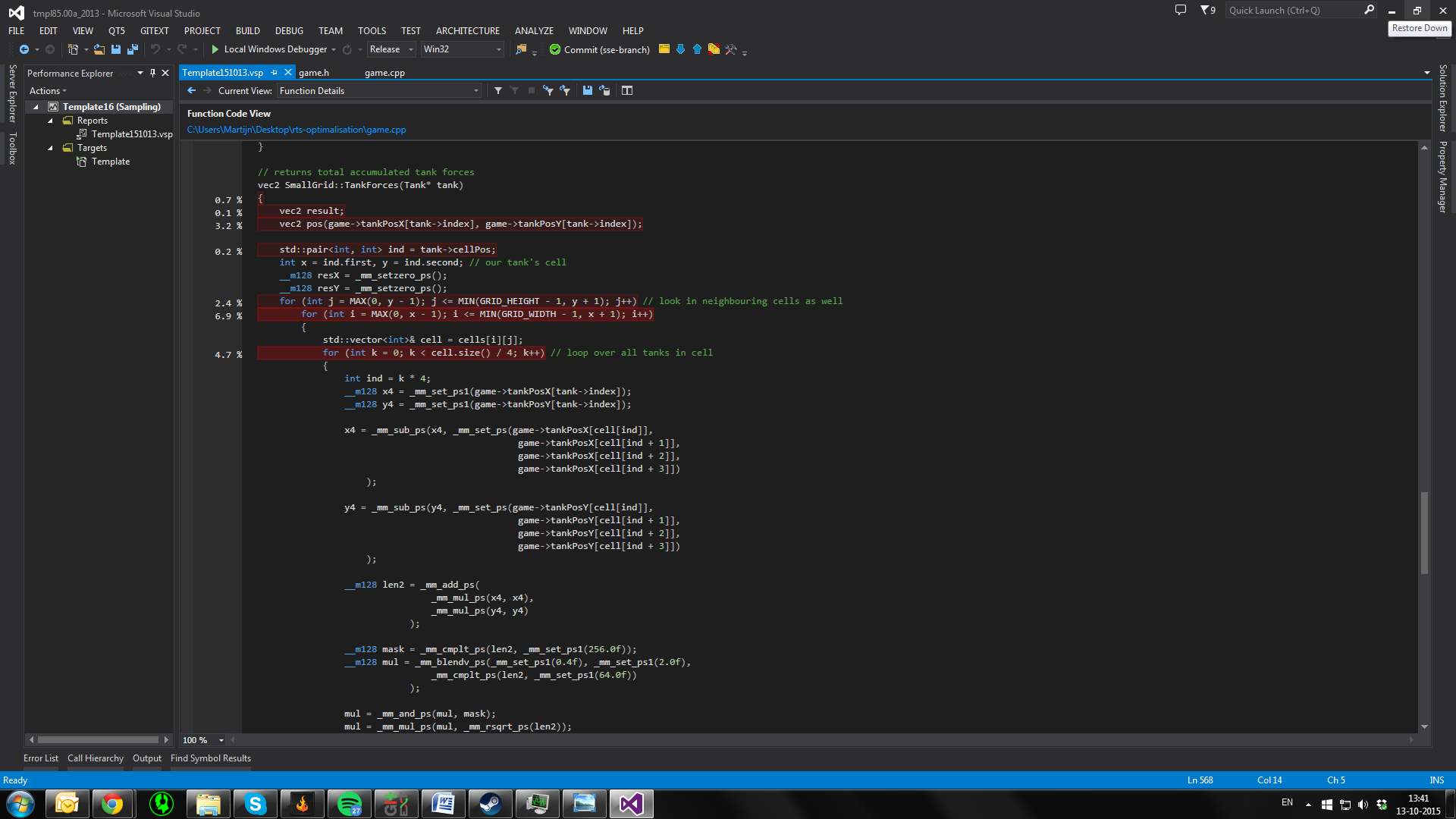
From this we decided to try to improve the updateGrid because it was still taking up quite some time and we thought we could get a performance increase doing the calculations for all tank once per tick instead of letting all tanks update themselves. We also changed the large grid to be 128 (a multiple of the size of the small grid: 16). This can help because we can easily convert the indices in the small grid to indices in the large grid. However during this change we found a bug where we used < instead of <= which meant that when applying forces to the tank only part of the cells that needed to be checked where checked. After merging the improvement and the bug fix the application ran slightly slower (33 fps for 40k blue tanks).

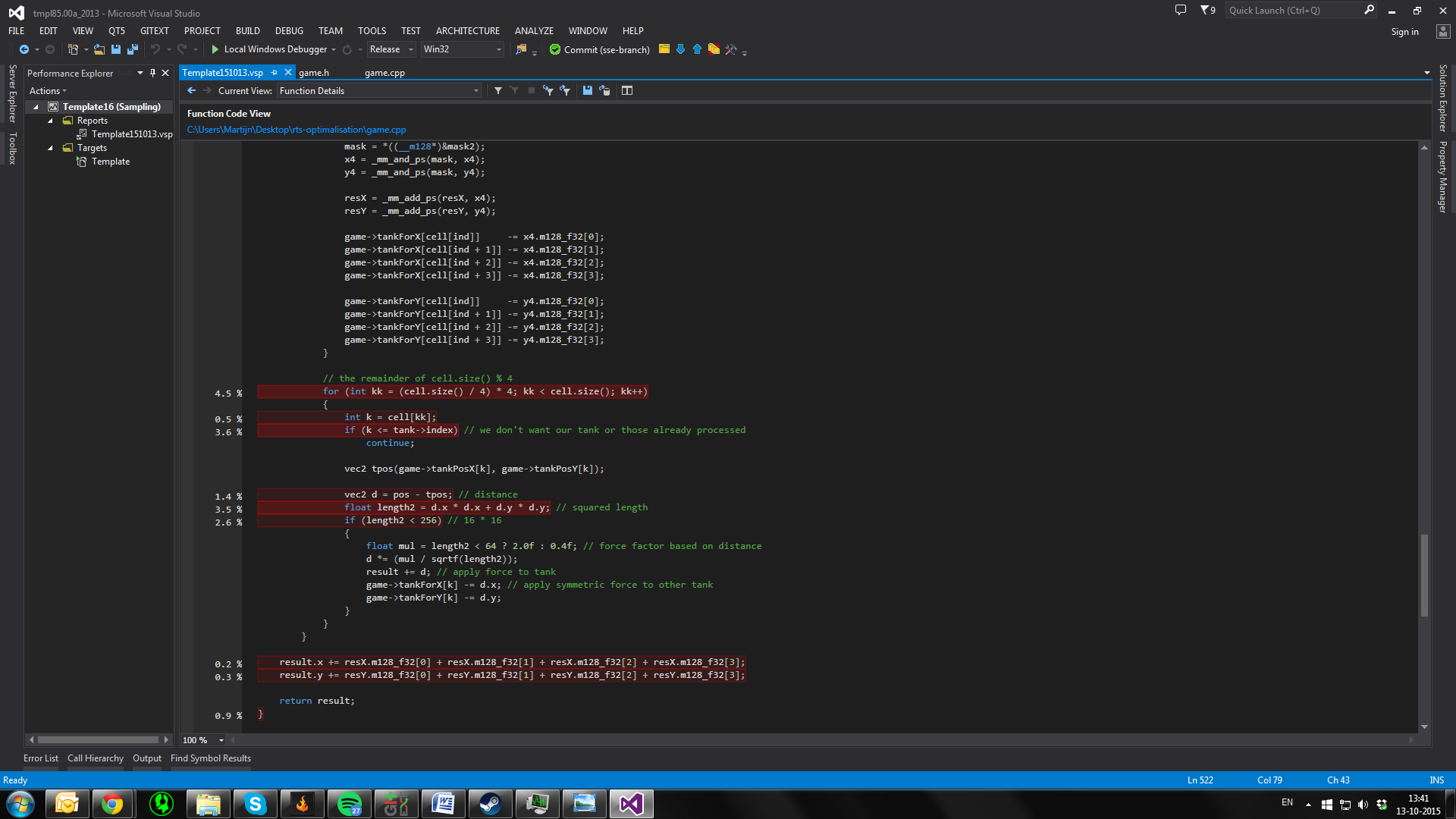
We once again ran the profiler and saw the following image: 

From this data we concluded that the next optimalisation target would be the evading the mountainpeaks computations since it took 21.4% of the CPU time and was untouched so far. We decided to instead of calculating the forces from the tanks we would each tick loop over the peaks and then loop over the cells influences by the current peak. This would reduce the amount of misses on the if statement significantly. We also decided to do some low level optimisation since we where rewriting it anyway. We created lookuptables for the sinf and cosf and remove the \*0.2f.

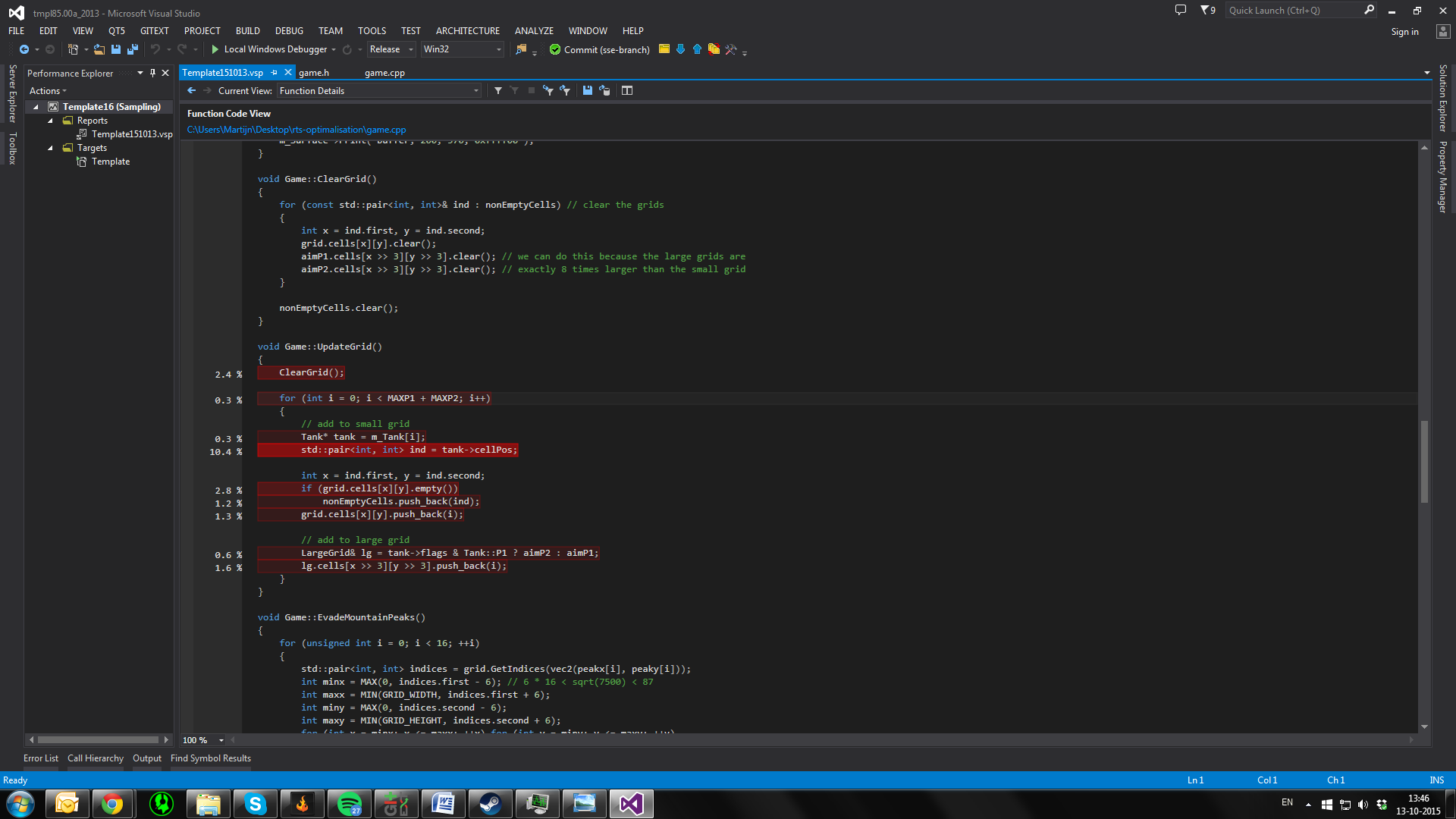
This gave a significant increase in performance to 43 fps for 40k blue tanks. We then profiled again and saw this: 

From this we concluded that the tankforces which is responsible for the evasion of other tanks was the next target for optimisation. To optimise this we want to apply low level optimisation like SSE. For this we will need to change the structure of our application from AoS to SoA. After doing this we rewrote the TankForces function to use SSE instructions and work on 4 tanks at the same time. However this did not gave us any speed increase. Instead it slowed the application down slightly to 41 fps for 40k blue tanks. When we looked closely at the profiling and results we saw this:





From this information we figured that the SSE improvements did not work because there are almost never more than 4 tanks in one cell. We confirmed our suspicions by running with a breakpoint in the SSE code. To solve this we tried to use SSE instructions for the remaining tanks aswell. This did not give any speed increase. We then tried to use larger cells (with a check whether neighbours need to be checked) in order to increase the number of tanks in a cell. This also turned out to be slower. We then decided to look for another opimisation target. In the profiler result we saw this:



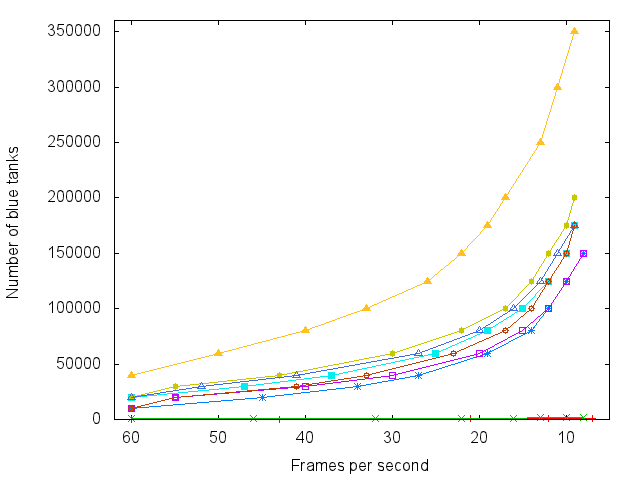
From this we concluded that the UpdateGrid function was slow because of cache misses on:

Tank\* tank = m\_Tank[i];.

We decided to try to solve this by storing a data in the cells instead of in the tanks. This should make sure that the data is stored localy and reduce cache misses. This however did not work at all. We then tried to change the type of m\_Tank from Tank\* to Tank&. This could also enure that data is stored more locally and thus reduce cache missen. This sadly also did not work.

We then decided that because of the time constraints we could not do any further optimisation except for parallelise the application using OpenMP multithreading. This gave us a significant speed increase. We can now run 80k blue tanks at 40 fps.

# Frame rate measurements

In order to keep an overview of the improvement in speed we did fps measurements after each step described in the section above. For each step we measured the fps for a range of number of blue tanks. The results of the measurements can be seen in the following graph. 

This graph shows for each step how many tanks could be simulated at a certain fps. You can see that the first two iterations are not even close to being capable of the same as the rest of the iterations. You can also see that after the first to iterations progress slowed down significantly until we parallelised the application in the last step.

# Work distribution

Similar to the previous assignment most of the work was done while working together at the university. At home we worked in parallel on different attempts, approaches and or targets.