Assignment 2 “RTS”

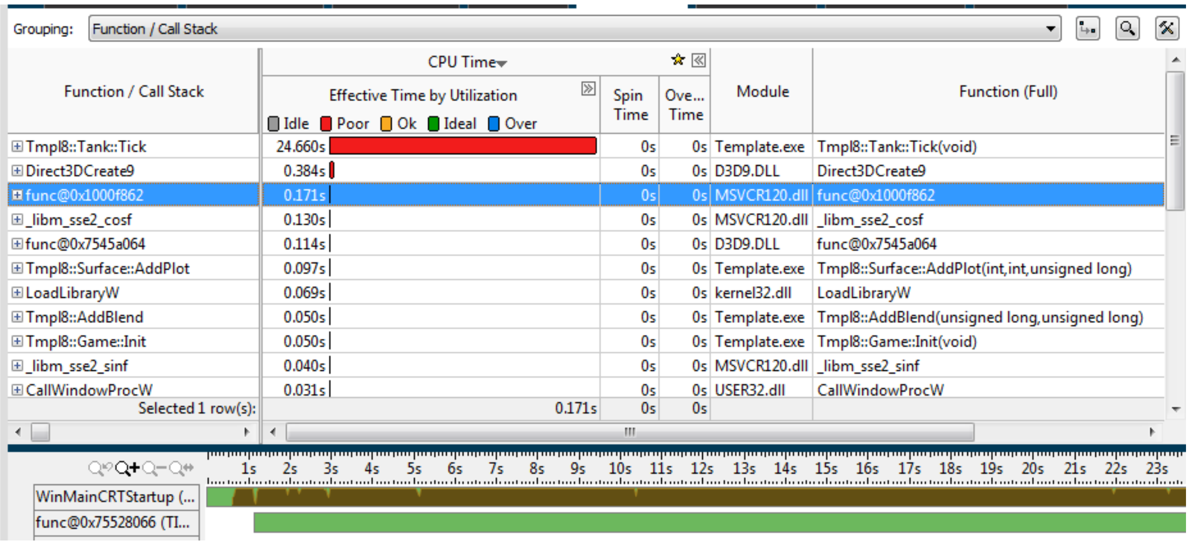
Jordi Vermeulen (3835634)

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:

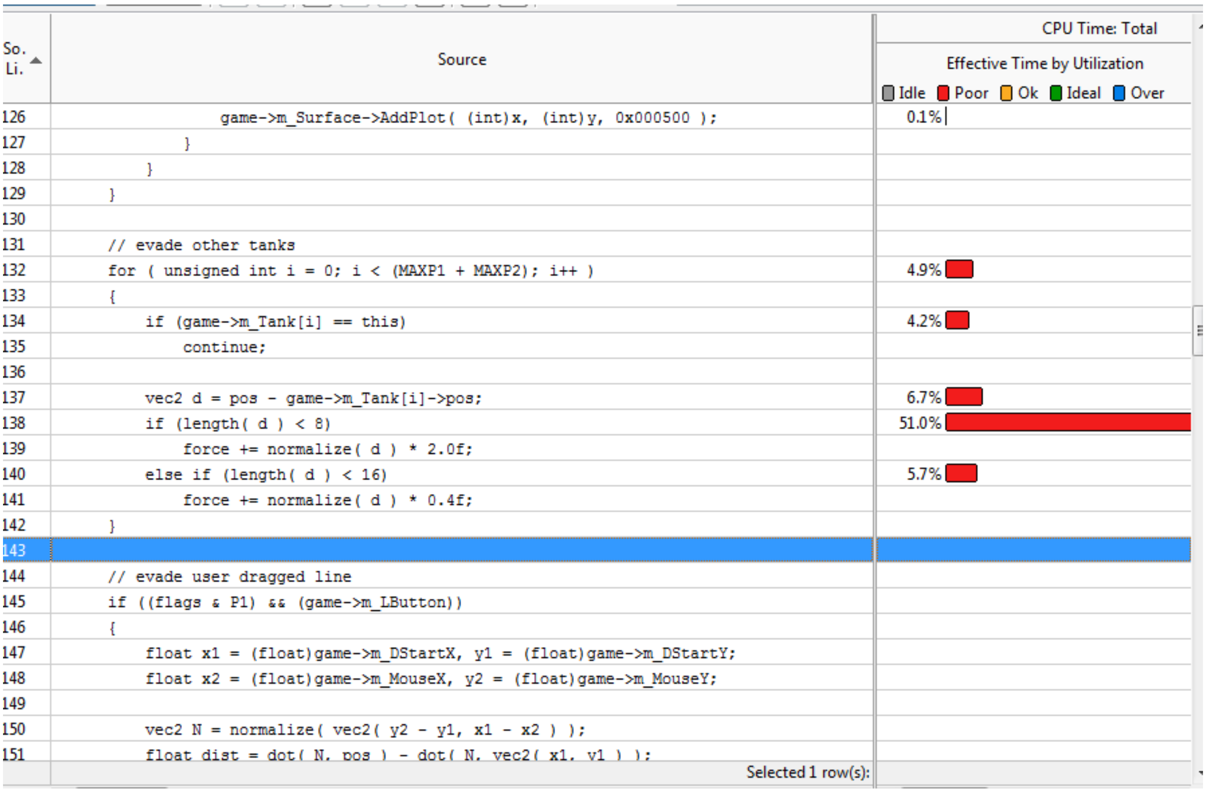


Grouping: Function / Call Stack

Function / Call Stack	CPU Time: Effective Time by Utilization ▢ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over	Spin Time	Ove... Time	Module	Function (Full)
⊞Tmpl8::Tank::Tick	24.660s	0s	0s	Template.exe	Tmpl8::Tank::Tick(void)
⊞Direct3DCreate9	0.384s	0s	0s	D3D9.DLL	Direct3DCreate9
⊞func@0x1000f862	0.171s	0s	0s	MSVCR120.dll	func@0x1000f862
⊞_libm_sse2_cosf	0.130s	0s	0s	MSVCR120.dll	_libm_sse2_cosf
⊞func@0x7545a064	0.114s	0s	0s	D3D9.DLL	func@0x7545a064
⊞Tmpl8::Surface::AddPlot	0.097s	0s	0s	Template.exe	Tmpl8::Surface::AddPlot(int,int,unsigned long)
⊞LoadLibraryW	0.069s	0s	0s	kernel32.dll	LoadLibraryW
⊞Tmpl8::AddBlend	0.050s	0s	0s	Template.exe	Tmpl8::AddBlend(unsigned long,unsigned long)
⊞Tmpl8::Game::Init	0.050s	0s	0s	Template.exe	Tmpl8::Game::Init(void)
⊞_libm_sse2_sinf	0.040s	0s	0s	MSVCR120.dll	_libm_sse2_sinf
⊞CallWindowProcW	0.031s	0s	0s	USER32.dll	CallWindowProcW
Selected 1 row(s):	0.171s	0s	0s		

1s 2s 3s 4s 5s 6s 7s 8s 9s 10s 11s 12s 13s 14s 15s 16s 17s 18s 19s 20s 21s 22s 23s
WinMainCRTStartup (...
func@0x75528066 (TI...

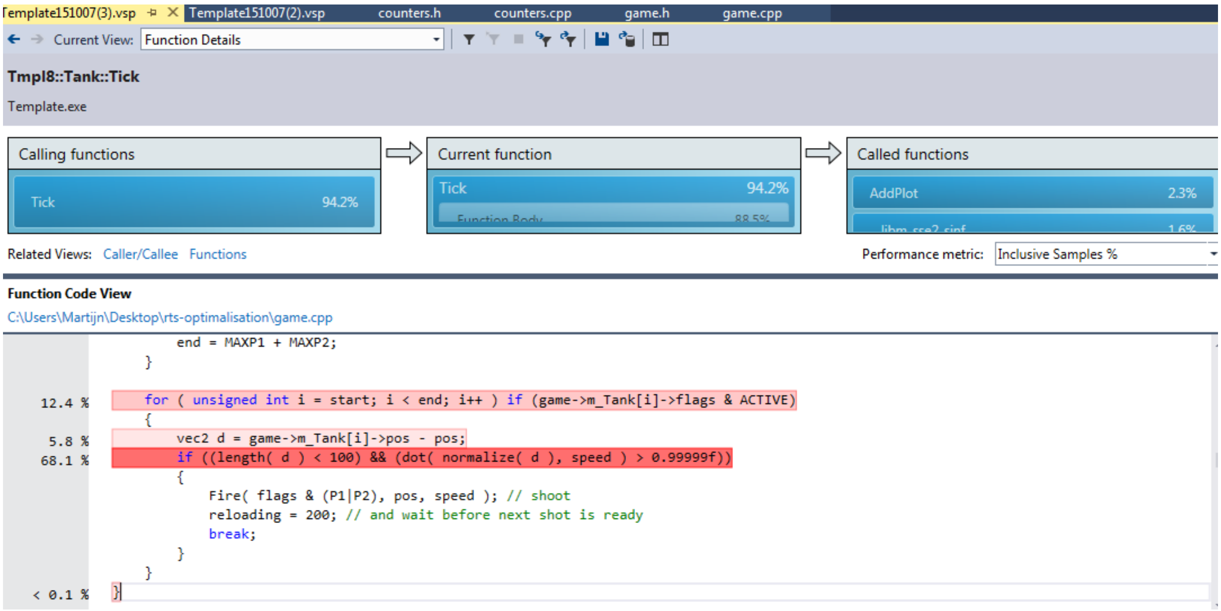
and when looking into the Tank::Tick() function:



So. Li.	Source	CPU Time: Total – Effective Time by Utilization ▢ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over
126	game->m_Surface->AddPlot((int)x, (int)y, 0x000500);	0.1%
127	}	
128	}	
129	}	
130		
131	// evade other tanks	
132	for (unsigned int i = 0; i < (MAXP1 + MAXP2); i++)	4.9%
133	{	
134	if (game->m_Tank[i] == this)	4.2%
135	continue;	
136		
137	vec2 d = pos - game->m_Tank[i]->pos;	6.7%
138	if (length(d) < 8)	51.0%
139	force += normalize(d) * 2.0f;	
140	else if (length(d) < 16)	5.7%
141	force += normalize(d) * 0.4f;	
142	}	
143		
144	// evade user dragged line	
145	if ((flags & P1) && (game->m_LButton))	
146	{	
147	float x1 = (float)game->m_DStartX, y1 = (float)game->m_DStartY;	
148	float x2 = (float)game->m_MouseX, y2 = (float)game->m_MouseY;	
149		
150	vec2 N = normalize(vec2(y2 - y1, x1 - x2));	
151	float dist = dot(N, pos) - dot(N, vec2(x1, y1));	
	Selected 1 row(s):	

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(n^2) (the Tank::Tick() function loops over all tanks and is called #tank times per frame). A high level optimisation 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.

We created a grid with cells of 16x16 pixel with in each cell a linked lists which stores pointers to the tanks in that cell. We later replaced linked list of pointers with a vector with indices of the tank in m_Tank array. We ran the visual studio profiler (with 1000 blue tanks at 32 fps) and got these results:



Current View: Function Details

Tmpl8::Tank::Tick

Template.exe

Calling functions	Current function	Called functions
Tick 94.2%	Tick 94.2%	AddPlot 2.3%
	Function Body 88.5%	libm_sse2_sinf 1.6%

Performance metric: Inclusive Samples %

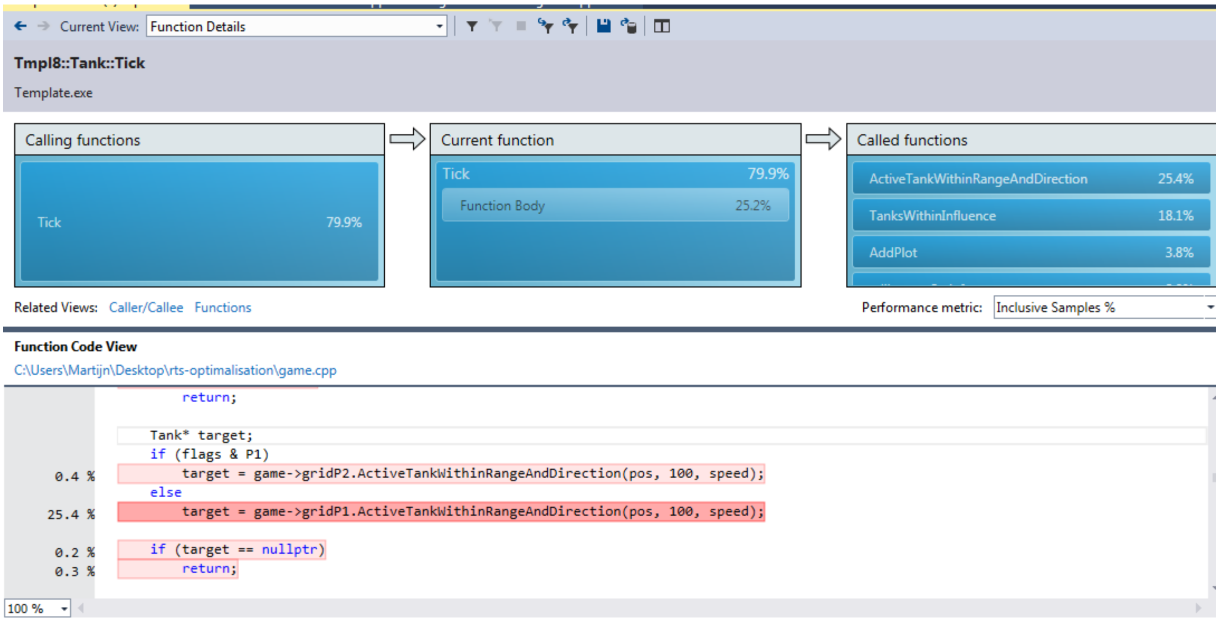
Function Code View

C:\Users\Martijn\Desktop\rts-optimalisation\game.cpp

```
                end = MAXP1 + MAXP2;
            }

 12.4 %     for ( unsigned int i = start; i < end; i++ ) if (game->m_Tank[i]->flags & ACTIVE)
            {
  5.8 %         vec2 d = game->m_Tank[i]->pos - pos;
 68.1 %         if ((length( d ) < 100) && (dot( normalize( d ), speed ) > 0.99999f))
                {
                    Fire( flags & (P1|P2), pos, speed ); // shoot
                    reloading = 200; // and wait before next shot is ready
                    break;
                }
            }
< 0.1 % }
```

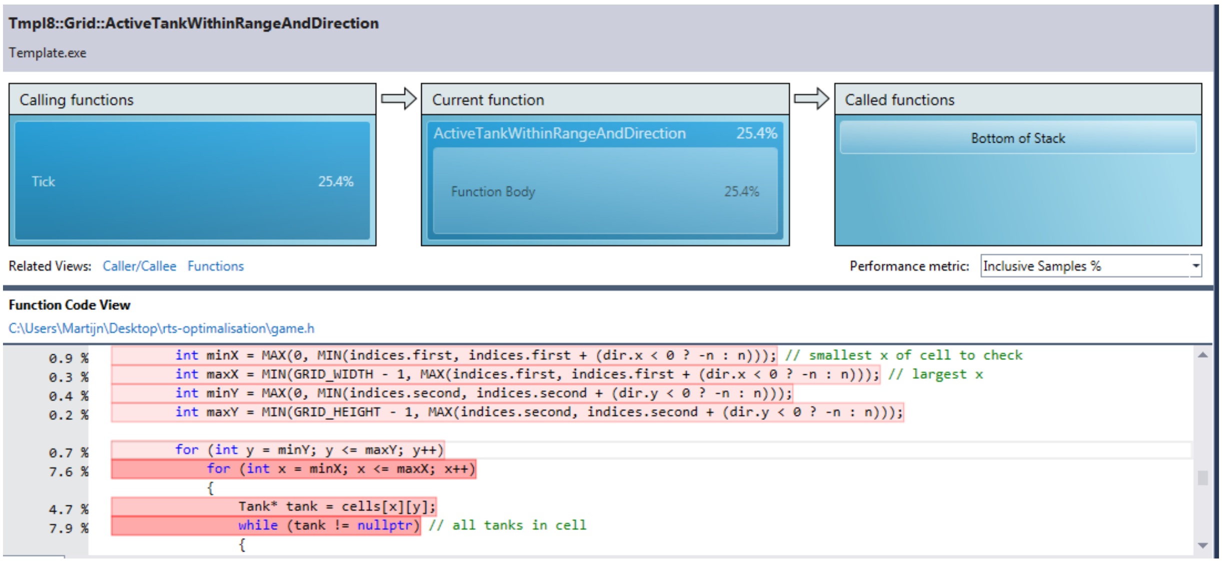
Here you can see that Tank::Tick() is still the main bottleneck of our application. Inside this function the loop for shooting bullets takes most of the CPU time so that will be the next focus point. The complexity of this part of the application still is O(n^2). The solution is the same as with evading 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:


Current View:
Function Details
Tmpl8::Tank::Tick
Template.exe
Calling functions
Tick 79.9%
Current function
Tick 79.9%
Function Body 25.2%
Called functions
ActiveTankWithinRangeAndDirection 25.4%
TanksWithinInfluence 18.1%
AddPlot 3.8%
Related Views: Caller/Callee Functions
Performance metric: Inclusive Samples %
Function Code View
C:\Users\Martijn\Desktop\rts-optimalisation\game.cpp
100 %

```
                     return;

                 Tank* target;
                 if (flags & P1)
  0.4 %              target = game->gridP2.ActiveTankWithinRangeAndDirection(pos, 100, speed);
                 else
 25.4 %              target = game->gridP1.ActiveTankWithinRangeAndDirection(pos, 100, speed);

  0.2 %          if (target == nullptr)
  0.3 %              return;
```

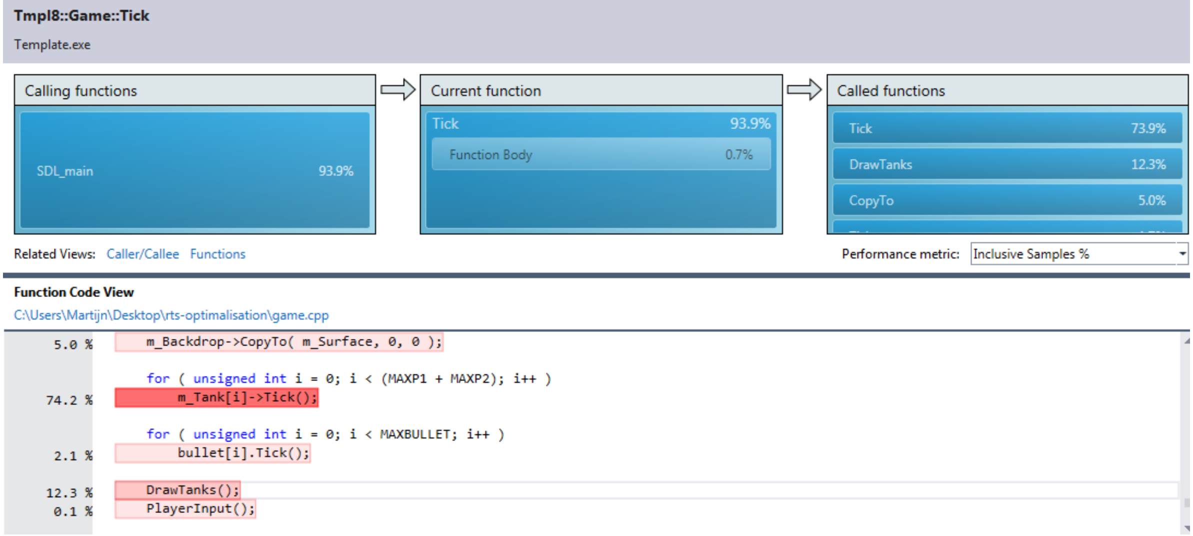
And in the ActiveTankWithinRangeAndDirection() function:


Tmpl8::Grid::ActiveTankWithinRangeAndDirection
Template.exe
Calling functions
Tick 25.4%
Current function
ActiveTankWithinRangeAndDirection 25.4%
Function Body 25.4%
Called functions
Bottom of Stack
Related Views: Caller/Callee Functions
Performance metric: Inclusive Samples %
Function Code View
C:\Users\Martijn\Desktop\rts-optimalisation\game.h

```
 0.9 %          int minX = MAX(0, MIN(indices.first, indices.first + (dir.x < 0 ? -n : n))); // smallest x of cell to check
 0.3 %          int maxX = MIN(GRID_WIDTH - 1, MAX(indices.first, indices.first + (dir.x < 0 ? -n : n))); // largest x
 0.4 %          int minY = MAX(0, MIN(indices.second, indices.second + (dir.y < 0 ? -n : n)));
 0.2 %          int maxY = MIN(GRID_HEIGHT - 1, MAX(indices.second, indices.second + (dir.y < 0 ? -n : n)));

 0.7 %          for (int y = minY; y <= maxY; y++)
 7.6 %              for (int x = minX; x <= maxX; x++)
                    {
 4.7 %                  Tank* tank = cells[x][y];
 7.9 %                  while (tank != nullptr) // all tanks in cell
                        {
```

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. We created two of these larger grids, one containing all blue tanks and one containing all red tanks, to prevent false positives taking much CPU time.

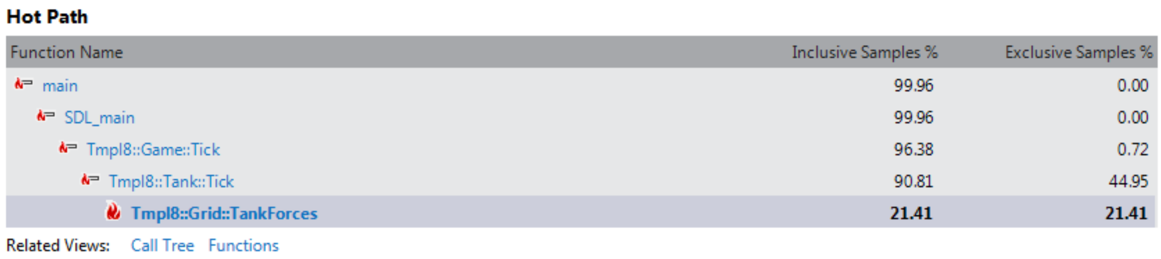
After applying this optimisation we can simulate 40k blue tanks at 30 fps. We then profiled the application again. This gave the following results:



```
5.0 %     m_Backdrop->CopyTo( m_Surface, 0, 0 );

          for ( unsigned int i = 0; i < (MAXP1 + MAXP2); i++ )
74.2 %        m_Tank[i]->Tick();

          for ( unsigned int i = 0; i < MAXBULLET; i++ )
2.1 %         bullet[i].Tick();

12.3 %    DrawTanks();
0.1 %     PlayerInput();
```

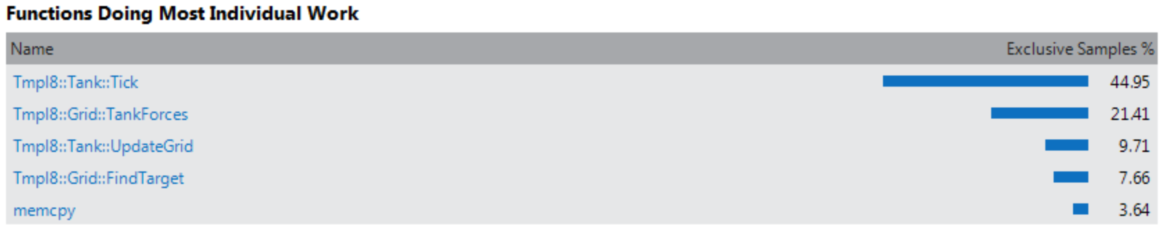
From this data we concluded that Tank::Tick() still uses most of the CPU time but now DrawTanks() also uses a significant portion of the time. We also figured that the draw code can be sped up significantly by only drawing tanks that are on the screen. For this we can use the 100*100 grid used for the 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:



Hot Path

Function Name	Inclusive Samples %	Exclusive Samples %
main	99.96	0.00
SDL_main	99.96	0.00
Tmpl8::Game::Tick	96.38	0.72
Tmpl8::Tank::Tick	90.81	44.95
Tmpl8::Grid::TankForces	**21.41**	**21.41**

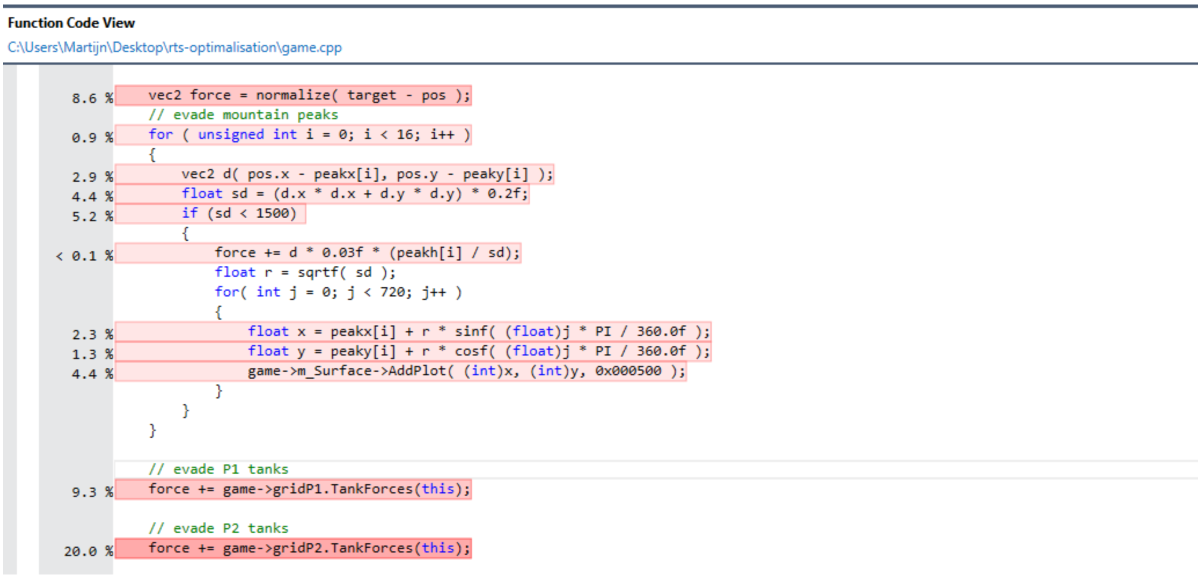


Functions Doing Most Individual Work

Name	Exclusive Samples %
Tmpl8::Tank::Tick	44.95
Tmpl8::Grid::TankForces	21.41
Tmpl8::Tank::UpdateGrid	9.71
Tmpl8::Grid::FindTarget	7.66
memcpy	3.64

From this we decided to try to improve the Tank::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:



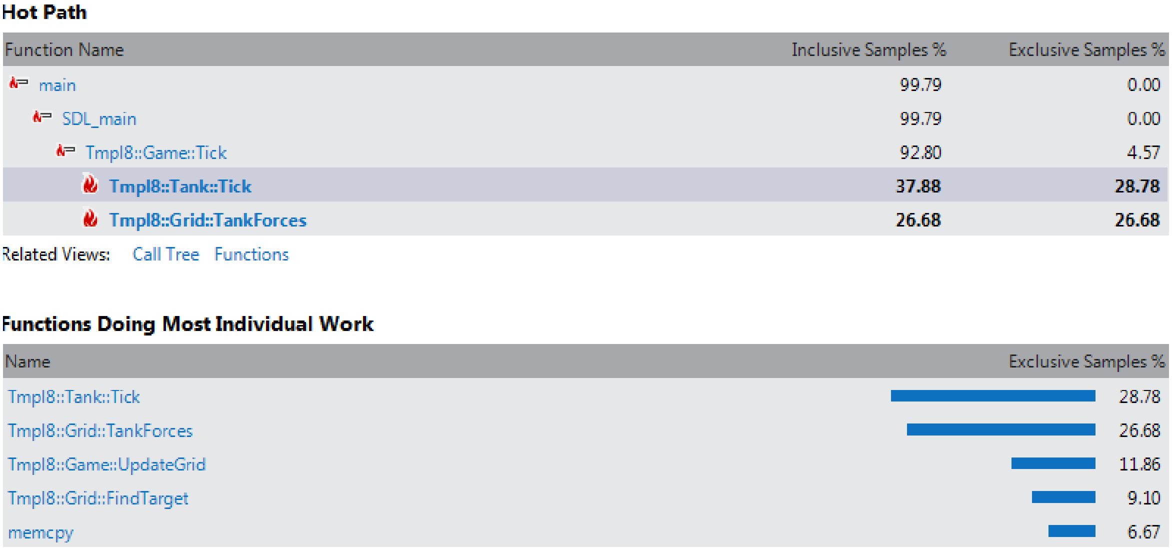
Function Code View

C:\Users\Martijn\Desktop\rts-optimalisation\game.cpp

```
  8.6 %    vec2 force = normalize( target - pos );
           // evade mountain peaks
  0.9 %    for ( unsigned int i = 0; i < 16; i++ )
           {
  2.9 %        vec2 d( pos.x - peakx[i], pos.y - peaky[i] );
  4.4 %        float sd = (d.x * d.x + d.y * d.y) * 0.2f;
  5.2 %        if (sd < 1500)
               {
< 0.1 %            force += d * 0.03f * (peakh[i] / sd);
                   float r = sqrtf( sd );
                   for( int j = 0; j < 720; j++ )
                   {
  2.3 %                float x = peakx[i] + r * sinf( (float)j * PI / 360.0f );
  1.3 %                float y = peaky[i] + r * cosf( (float)j * PI / 360.0f );
  4.4 %                game->m_Surface->AddPlot( (int)x, (int)y, 0x000500 );
                   }
               }
           }

           // evade P1 tanks
  9.3 %    force += game->gridP1.TankForces(this);

           // evade P2 tanks
 20.0 %    force += game->gridP2.TankForces(this);
```

From this data we concluded that the next optimisation target would be the evasion of the mountain peaks, 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 loop over the peaks each tick and then loop over the cells influenced 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, so we created lookup tables for the sinf and cosf.

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



Hot Path

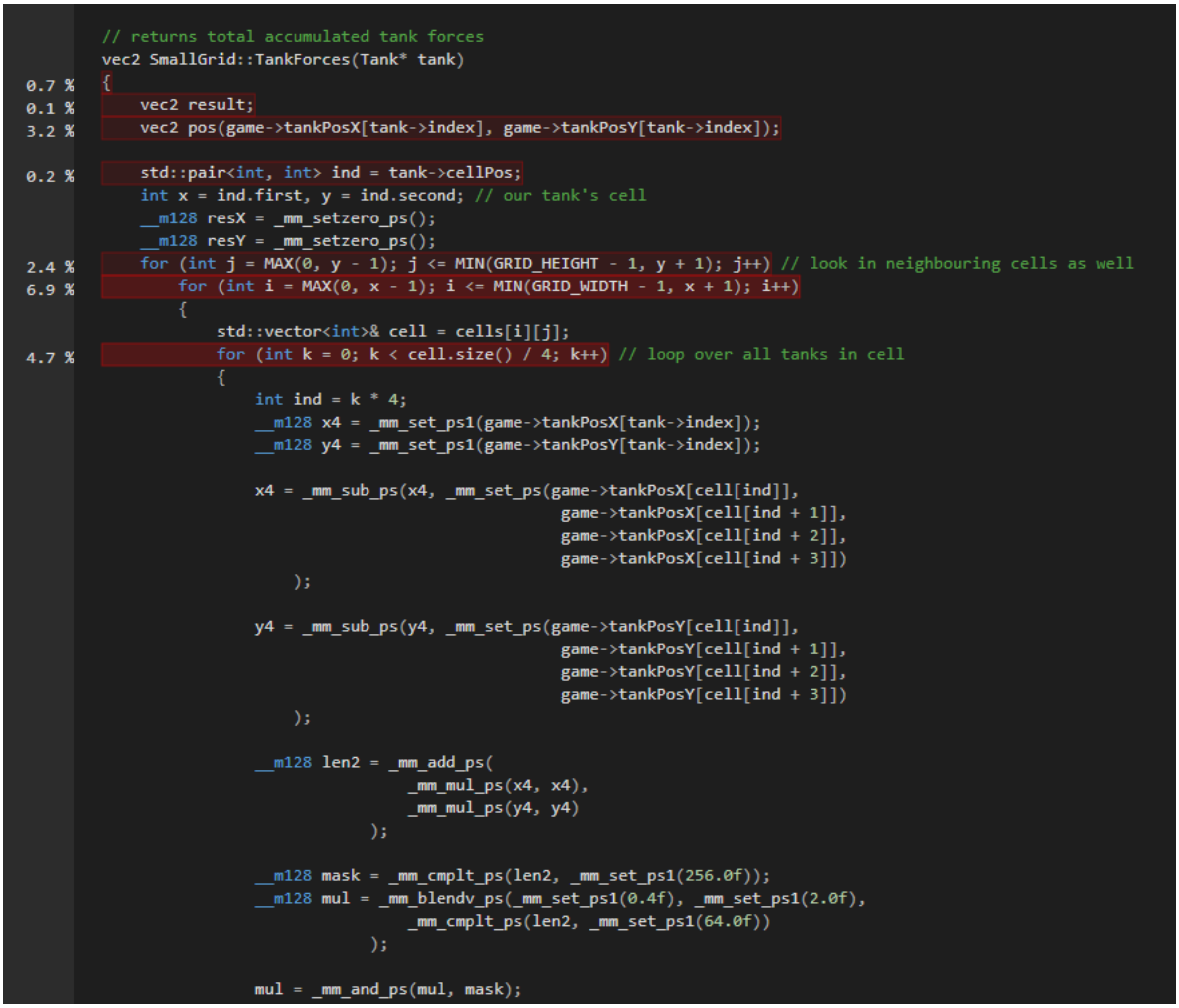
Function Name	Inclusive Samples %	Exclusive Samples %
main	99.79	0.00
SDL_main	99.79	0.00
Tmpl8::Game::Tick	92.80	4.57
Tmpl8::Tank::Tick	**37.88**	**28.78**
Tmpl8::Grid::TankForces	**26.68**	**26.68**

Related Views: Call Tree Functions

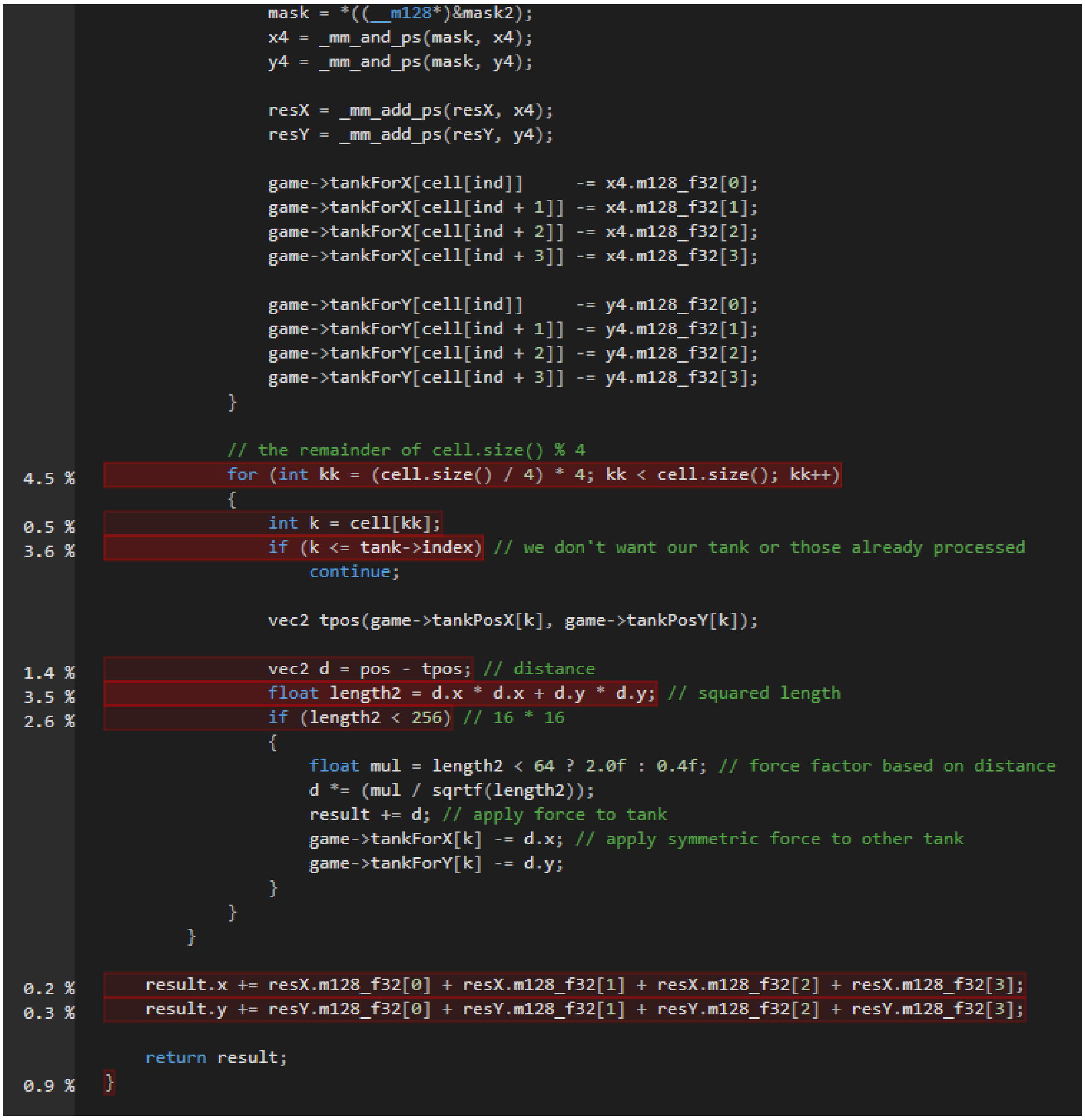
Functions Doing Most Individual Work

Name	Exclusive Samples %
Tmpl8::Tank::Tick	28.78
Tmpl8::Grid::TankForces	26.68
Tmpl8::Game::UpdateGrid	11.86
Tmpl8::Grid::FindTarget	9.10
memcpy	6.67

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:

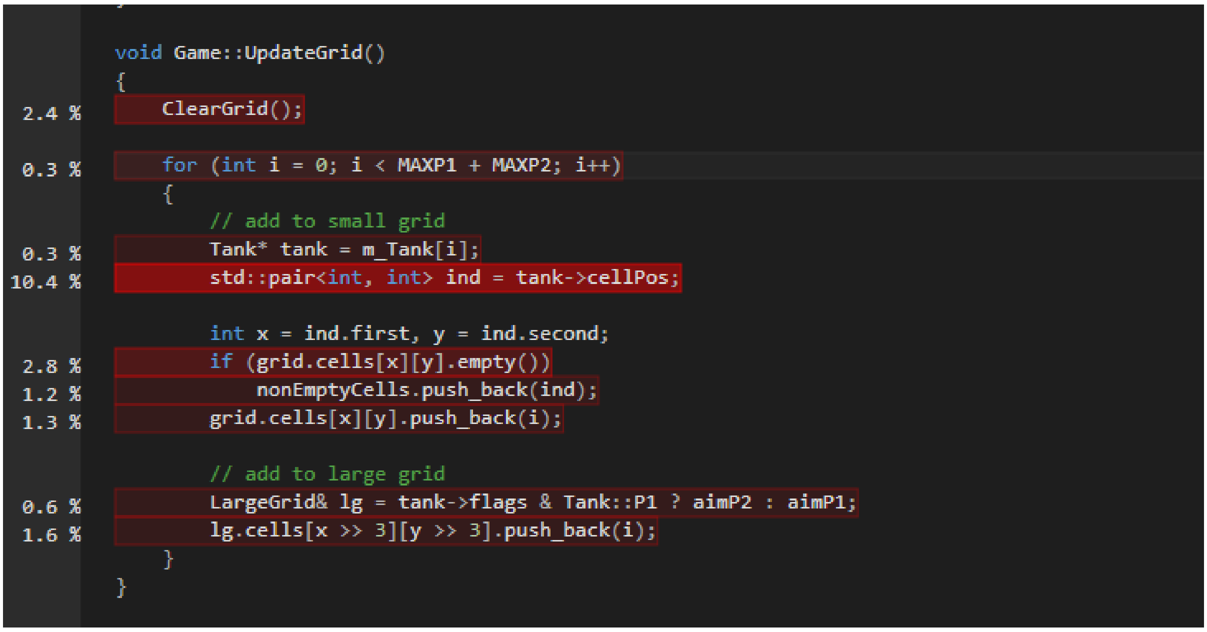


```
       // returns total accumulated tank forces
       vec2 SmallGrid::TankForces(Tank* tank)
0.7 %  {
0.1 %      vec2 result;
3.2 %      vec2 pos(game->tankPosX[tank->index], game->tankPosY[tank->index]);

0.2 %      std::pair<int, int> ind = tank->cellPos;
           int x = ind.first, y = ind.second; // our tank's cell
           __m128 resX = _mm_setzero_ps();
           __m128 resY = _mm_setzero_ps();
2.4 %      for (int j = MAX(0, y - 1); j <= MIN(GRID_HEIGHT - 1, y + 1); j++) // look in neighbouring cells as well
6.9 %          for (int i = MAX(0, x - 1); i <= MIN(GRID_WIDTH - 1, x + 1); i++)
               {
                   std::vector<int>& cell = cells[i][j];
4.7 %              for (int k = 0; k < cell.size() / 4; k++) // loop over all tanks in cell
                   {
                       int ind = k * 4;
                       __m128 x4 = _mm_set_ps1(game->tankPosX[tank->index]);
                       __m128 y4 = _mm_set_ps1(game->tankPosY[tank->index]);

                       x4 = _mm_sub_ps(x4, _mm_set_ps(game->tankPosX[cell[ind]],
                                                       game->tankPosX[cell[ind + 1]],
                                                       game->tankPosX[cell[ind + 2]],
                                                       game->tankPosX[cell[ind + 3]])
                           );

                       y4 = _mm_sub_ps(y4, _mm_set_ps(game->tankPosY[cell[ind]],
                                                       game->tankPosY[cell[ind + 1]],
                                                       game->tankPosY[cell[ind + 2]],
                                                       game->tankPosY[cell[ind + 3]])
                           );

                       __m128 len2 = _mm_add_ps(
                                           _mm_mul_ps(x4, x4),
                                           _mm_mul_ps(y4, y4)
                                       );

                       __m128 mask = _mm_cmplt_ps(len2, _mm_set_ps1(256.0f));
                       __m128 mul = _mm_blendv_ps(_mm_set_ps1(0.4f), _mm_set_ps1(2.0f),
                                           _mm_cmplt_ps(len2, _mm_set_ps1(64.0f))
                                       );

                       mul = _mm_and_ps(mul, mask);
```



```
                        mask = *((__m128*)&mask2);
                        x4 = _mm_and_ps(mask, x4);
                        y4 = _mm_and_ps(mask, y4);

                        resX = _mm_add_ps(resX, x4);
                        resY = _mm_add_ps(resY, y4);

                        game->tankForX[cell[ind]]     -= x4.m128_f32[0];
                        game->tankForX[cell[ind + 1]] -= x4.m128_f32[1];
                        game->tankForX[cell[ind + 2]] -= x4.m128_f32[2];
                        game->tankForX[cell[ind + 3]] -= x4.m128_f32[3];

                        game->tankForY[cell[ind]]     -= y4.m128_f32[0];
                        game->tankForY[cell[ind + 1]] -= y4.m128_f32[1];
                        game->tankForY[cell[ind + 2]] -= y4.m128_f32[2];
                        game->tankForY[cell[ind + 3]] -= y4.m128_f32[3];
                    }

                    // the remainder of cell.size() % 4
4.5 %               for (int kk = (cell.size() / 4) * 4; kk < cell.size(); kk++)
                    {
0.5 %                   int k = cell[kk];
3.6 %                   if (k <= tank->index) // we don't want our tank or those already processed
                            continue;

                        vec2 tpos(game->tankPosX[k], game->tankPosY[k]);

1.4 %                   vec2 d = pos - tpos; // distance
3.5 %                   float length2 = d.x * d.x + d.y * d.y; // squared length
2.6 %                   if (length2 < 256) // 16 * 16
                        {
                            float mul = length2 < 64 ? 2.0f : 0.4f; // force factor based on distance
                            d *= (mul / sqrtf(length2));
                            result += d; // apply force to tank
                            game->tankForX[k] -= d.x; // apply symmetric force to other tank
                            game->tankForY[k] -= d.y;
                        }
                    }
                }

0.2 %       result.x += resX.m128_f32[0] + resX.m128_f32[1] + resX.m128_f32[2] + resX.m128_f32[3];
0.3 %       result.y += resY.m128_f32[0] + resY.m128_f32[1] + resY.m128_f32[2] + resY.m128_f32[3];

            return result;
0.9 %   }
```

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 as well. 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 optimisation target. In the profiler result we saw this:



```
           void Game::UpdateGrid()
           {
  2.4 %        ClearGrid();

  0.3 %        for (int i = 0; i < MAXP1 + MAXP2; i++)
               {
                   // add to small grid
  0.3 %            Tank* tank = m_Tank[i];
 10.4 %            std::pair<int, int> ind = tank->cellPos;

                   int x = ind.first, y = ind.second;
  2.8 %            if (grid.cells[x][y].empty())
  1.2 %                nonEmptyCells.push_back(ind);
  1.3 %            grid.cells[x][y].push_back(i);

                   // add to large grid
  0.6 %            LargeGrid& lg = tank->flags & Tank::P1 ? aimP2 : aimP1;
  1.6 %            lg.cells[x >> 3][y >> 3].push_back(i);
               }
           }
```

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

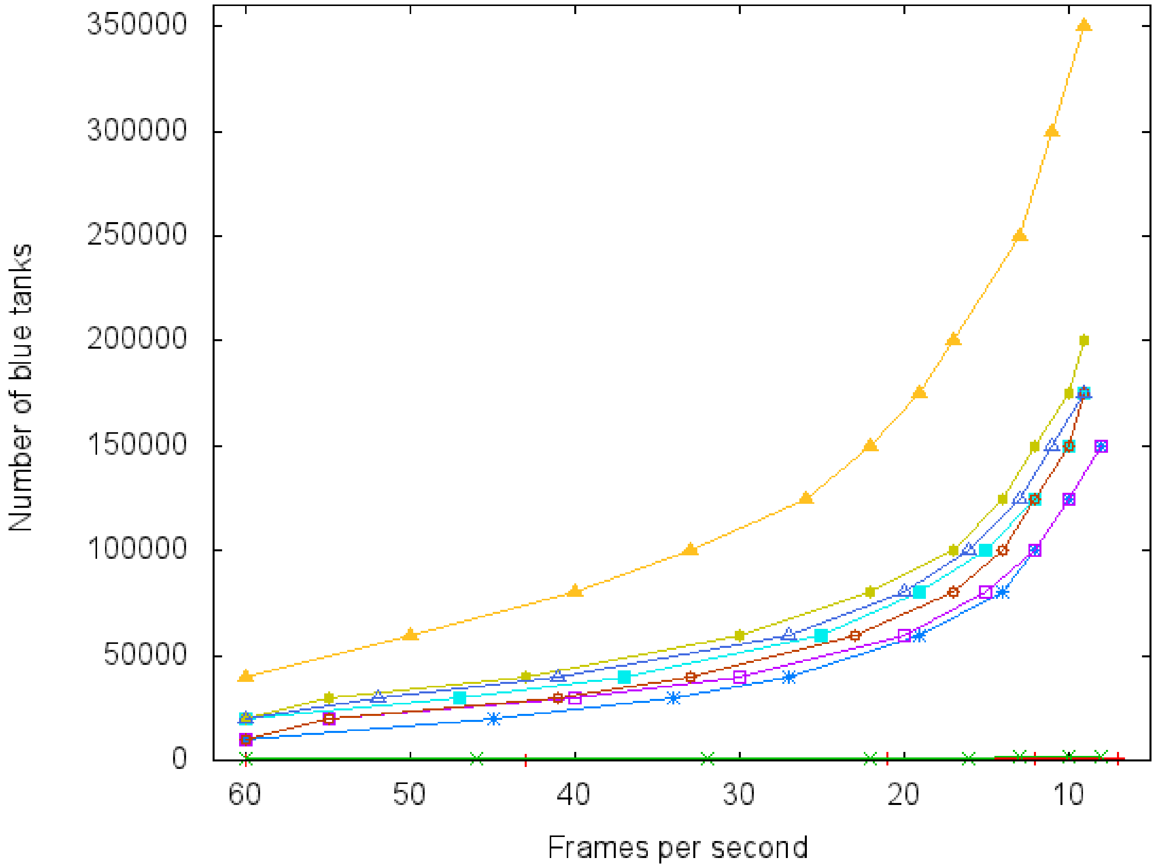
Tank* tank = m_Tank[i];.

We decided to try to solve this by storing data in the cells instead of in the tanks. This should make sure that the data is stored locally 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 ensure that data is stored more locally and thus reduce cache misses. 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 two 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.