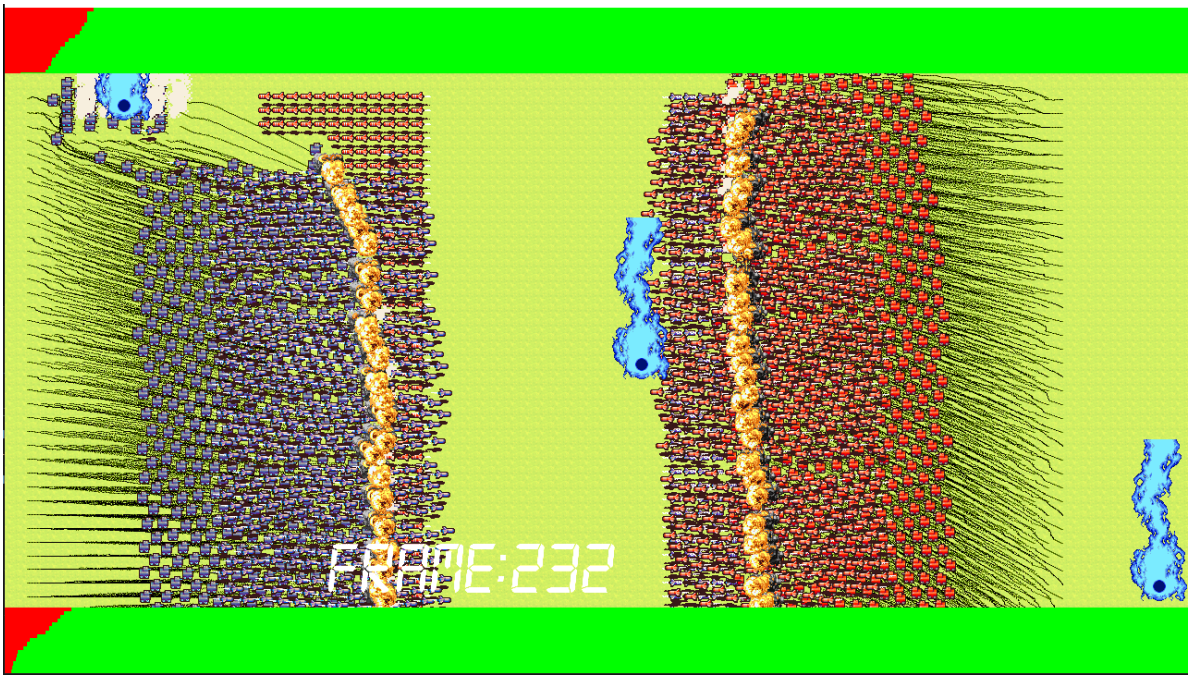
Machine intelligence Verbeteringen



Inhoud

Samenvatting

Eerste run. Zonder optimalisatie.

Algoritmische optimalisaties:

- Toevoegen van een grid. Toegepast op tank collision. Speedup 1.0 naar 1.9
- Healthbars worden getekend met alleen benodigde informatie. Ipv alle objecten binnen Tank. Speedup van 1.9 naar 2.0
- Healthbars Sorteren d.m.v een merge sort. In plaats van de huidige insertion sort. Geen grote speedup.
- Particle beams controleren alleen collision op tanks binnen de opgegeven locatie. Ipv alle tanks op het veld. Sim duurt korter maar niet genoeg voor een speedup.
- Toevoegen van grid op rocket collision speedup van 2.0 naar 2.7

Concurrent optimalisaties:

- Threads worden dynamisch aangemaakt.
- Toevoegen van threadpool en deze gebruiken op de rocketUpdate en tankUpdate speedup van 2.7 naar 4.0

BIG O ANALYSIS

Big o van de meegegeven code.

Tank collision detection:

```
//Update tanks
for (Tank& tank : tanks) //N
{
   if (tank.active)//1
   {
      //Check tank collision and nudge tanks away from each other
      for (Tank& oTank : tanks) //N
      {
         if (&tank == &oTank) continue; //1

         vec2 dir = tank.Get_Position() - oTank.Get_Position(); //1
         float dirSquaredLen = dir.sqrLength(); //1

         float colSquaredLen = (tank.Get_collision_radius() * tank.Get_collision_radius()) + (oTank.Get_collision_radius() * oTank.Get_collision_radius()); //1

         if (dirSquaredLen < colSquaredLen) //1
         {
            tank.Push(dir.normalized(), 1.f); //1
         }
      }
   }
```

O(N6N), O(N^2)

FindClosestEnemy:

```
Tank& Game::FindClosestEnemy(Tank& current_tank)
{
   float closest_distance = numeric_limits<float>::infinity();
   int closest_index = 0;

   for (int i = 0; i < tanks.size(); i++)//N
   {
      if (tanks.at(i).allignment != current_tank.allignment && tanks.at(i).active)//1
      {
         float sqrDist = fabsf((tanks.at(i).Get_Position() - current_tank.Get_Position()).sqrLength());//1
         if (sqrDist < closest_distance)//1
         {
            closest_distance = sqrDist;//1
            closest_index = i;//1
         }
      }
   }

   return tanks.at(closest_index);
}
```

O(5N), O(N)

Update rockets:

```
//Update rockets
for (Rocket& rocket : rockets)//N
{
   rocket.Tick();//1

   //Check if rocket collides with enemy tank, spawn explosion and if tank is destroyed spawn a smoke plume
   for (Tank& tank : tanks)//M
   {
      if (tank.active && (tank.allignment != rocket.allignment) && rocket.Intersects(tank.position, tank.collision_radius))//1
      {
         explosions.push_back(Explosion(&explosion, tank.position));//1

         if (tank.hit(ROCKET_HIT_VALUE))//1
         {
            smokes.push_back(Smoke(smoke, tank.position - vec2(0, 48)));//1
         }

         rocket.active = false;//1
         break;//1
      }
   }
}
```

O(N6M), O(NM)

Update Particle_Beams:

```
//Update particle beams
for (Particle_beam& particle_beam : particle_beams)//N
{
    particle_beam.tick(tanks);//1

    //Damage all tanks within the damage window of the beam (the window is an axis-aligned bounding box)
    for (Tank& tank : tanks)//M
    {
        if (tank.active && particle_beam.rectangle.intersectsCircle(tank.Get_Position(), tank.Get_collision_radius()))//1
        {
            if (tank.hit(particle_beam.damage))//1
            {
                smokes.push_back(Smoke(smoke, tank.position - vec2(0, 48)));//1
            }
        }
    }
}
```

O(N3M), O(NM)

InsertionSortTanksHealth:

```
void Tmpl8::Game::insertion_sort_tanks_health(const std::vector<Tank>& original, std::vector<const Tank*>& sorted_tanks, UINT16 begin, UINT16 end)
{
    const UINT16 NUM_TANKS = end - begin;//1
    sorted_tanks.reserve(NUM_TANKS);//1
    sorted_tanks.emplace_back(&original.at(begin));//1

    for (int i = begin + 1; i < (begin + NUM_TANKS); i++)//N
    {
        const Tank& current_tank = original.at(i);//1

        for (int s = (int)sorted_tanks.size() - 1; s >= 0; s--)//M
        {
            const Tank* current_checking_tank = sorted_tanks.at(s);//1

            if ((current_checking_tank->CompareHealth(current_tank) <= 0))//1
            {
                sorted_tanks.insert(1 + sorted_tanks.begin() + s, &current_tank);//1
                break;
            }

            if (s == 0)//1
            {
                sorted_tanks.insert(sorted_tanks.begin(), &current_tank);//1
                break;
            }
        }
    }
}
```

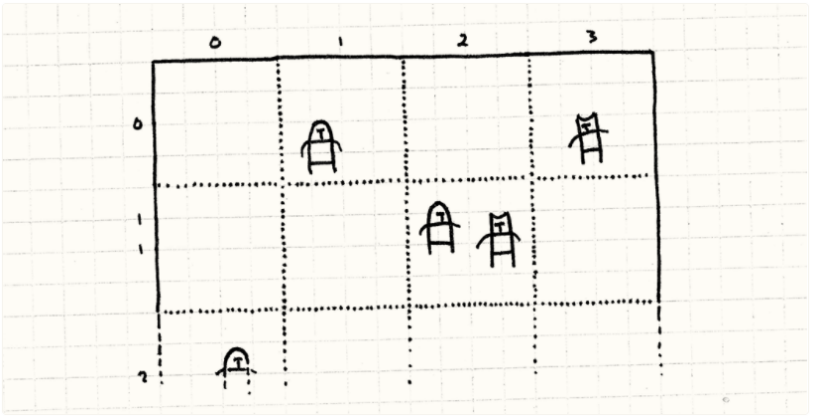
O(N5N), oftewel O(N^2)

Toegepaste verbeteringen

De tank collision detection met spatial partitioning grid:

De *nested for-loop collision detection* en *particle beam checkt alle tanks.* Deze problemen kunnen opgelost worden door het verkleinen van de ‘n’ in n^2 algoritmen.

Het Spatial partitioning Grid is hier een oplossing voor. Zie Figuur.



Grid voorbeeld

Met een grid verdelen we de tanks over meerdere cellen gebaseerd op de positie van de tank. Een cel bevat een lijst met tanks die dichtbij elkaar staan.
Collision detection uitvoeren op tanks die niet dichtbij elkaar staan is hierdoor verholpen.

Door de tanks te verdelen over meerdere lijsten is de grote van elke lijst minimaal geworden. Wanneer we op elke cel de n^2 algoritmen los laten zal dit sneller zijn dan het checken van alle tanks in één keer. Een n^2 algoritme met een klein aantal *elements* zal significant veel minder *operations* vereisen dan eentje met een groot aantal *elements.* Dit resulteert in een snellere check van de tanks.

In de broncode zijn er twee files aangemaakt grid.h en grid.cpp. Deze bevatten de code voor een grid class.

In game.cpp wordt een grid object aangemaakt. Hier worden tanks aan toegevoegd. Wanneer een update plaatsvindt wordt er HandleCollision() aangeroepen op het grid object. Omdat de tanks bewegen wordt er na elke tank tick() gecheckt of de tank verplaatst moet worden in de grid. Dit doen we door move(Tank* tank) op het grid object.

Draw Sorted Healthbars:

```
    sort(tanks.begin(), tanks.end(), CompareAllignment);

    blue_tanks_health.clear();
    red_tanks_health.clear();

    //Add bluetank healthbars to list
    for (int i = 0; i < NUM_TANKS_BLUE; i++)
    {
        blue_tanks_health.push_back(tanks[i].health);
    }

    //Add Red tank healthbars to list
    for (int i = NUM_TANKS_BLUE; i < NUM_TANKS_BLUE + NUM_TANKS_RED; i++)
    {
        red_tanks_health.push_back(tanks[i].health);
    }
```

```
    merge_sort(red_tanks_health);
    merge_sort(blue_tanks_health);

//Draw sorted health bars (using only usefull information instead of every object in tanks.)
for (int t = 0; t < 2; t++)
{
    const int NUM_TANKS = ((t < 1) ? NUM_TANKS_BLUE : NUM_TANKS_RED);

    const int begin = ((t < 1) ? 0 : NUM_TANKS_BLUE);
    if (t == 0)
    {
        for (int i = 0; i < NUM_TANKS; i++)
        {
            int health_bar_start_x = i * (HEALTH_BAR_WIDTH + HEALTH_BAR_SPACING) + HEALTH_BARS_OFFSET_X;
            int health_bar_start_y = 0;
            int health_bar_end_x = health_bar_start_x + HEALTH_BAR_WIDTH;
            int health_bar_end_y = HEALTH_BAR_HEIGHT;

            screen->bar(health_bar_start_x, health_bar_start_y, health_bar_end_x, health_bar_end_y, REDMASK);
            screen->bar(health_bar_start_x, health_bar_start_y + (int)((double)HEALTH_BAR_HEIGHT * (1 - ((double)blue_tanks_health[i] / (double)TANK_MAX_HEALTH))), health_bar_end_x, health_bar_end_y, GREENMASK);
        }
    }
    if (t == 1)
    {
        for (int i = 0; i < NUM_TANKS; i++)
        {
            int health_bar_start_x = i * (HEALTH_BAR_WIDTH + HEALTH_BAR_SPACING) + HEALTH_BARS_OFFSET_X;
            int health_bar_start_y = (SCHEIGHT - HEALTH_BAR_HEIGHT) - 1;
            int health_bar_end_x = health_bar_start_x + HEALTH_BAR_WIDTH;
            int health_bar_end_y = SCHEIGHT - 1;

            screen->bar(health_bar_start_x, health_bar_start_y, health_bar_end_x, health_bar_end_y, REDMASK);
            screen->bar(health_bar_start_x, health_bar_start_y + (int)((double)HEALTH_BAR_HEIGHT * (1 - ((double)red_tanks_health[i] / (double)TANK_MAX_HEALTH))), health_bar_end_x, health_bar_end_y, GREENMASK);
        }
    }
}
```

Elke update wordt tanks gesorteerd op alignement. Zo staan alle blauwe tanks in de eerste helft van de vector en de rode tanks op de tweede helft. Deze vectors worden gevuld met int waarden, bestaande uit de health waarden van de blauwe tanks en de health waarden van de rode tanks.

In de functie draw worden deze vectors gesorteerd en dit wordt gebruikt om de health bars te maken.

Hierbij is geen verandering van big O. Wel is dit efficiënter dan het sorteren en gebruiken van vectors gevuld met Tank objecten.

Update Particle_Beams:

```
//Update particle beams
for (Particle_beam& particle_beam : particle_beams) //N
{
    particle_beam.tick(tanks); //1

    // positions of the cells where particle_beam collision has to be detected.
    int particle_beam_left = particle_beam.min_position.x / grid.CELL_SIZE; //1
    int particle_beam_right = (particle_beam.min_position.x + particle_beams[0].max_position.x) / grid.CELL_SIZE; //1
    int particle_beam_top = particle_beam.min_position.y / grid.CELL_SIZE; //1
    int particle_beam_bottom = (particle_beam.min_position.y + particle_beams[0].max_position.y) / grid.CELL_SIZE; //1

    //Damage all tanks within the collision area.
    for (auto& fut : futs)
    {
        for (int x = particle_beam_left; x <= particle_beam_right; x++)
            for (int y = particle_beam_top; y <= particle_beam_bottom; y++)
            {
                {
                    for (Tank* tank : grid.cells[x][y]) //M
                    {
                        if (tank->active && particle_beam.rectangle.intersects_circle(tank->get_position(), tank->get_collision_radius())) //1
                        {
                            if (tank->hit(particle_beam.damage)) //1
                            {
                                smokes.push_back(Smoke(smoke, tank->position - vec2(0, 48))); //1
                            }
                        }
                    }
                }
            }

        fut.wait();
    }
}
```

Particlebeams controleren alleen op collision op de tanks binnen de opgegeven area.

Ook hier is geen verandering van de Big O. Wel is dit efficiënter omdat er eerst gecontroleerd werd op collision voor elke tank op het veld.

Insertion Sort Tank Health:

De insertion sort is gewijzigd tot een merge sort.

```
void Tmpl8::Game::merge(vector<int>& left, vector<int>& right, vector<int>& sorted)
{
    int i = 0;
    int j = 0;
    int k = 0;

    // O(N)
    // When merging two sorted lists, the list with the highest amount of elements dictates the time complexity

    while (j < left.size() && k < right.size()) // both lists are not empty
    {
        // check for smallest first elem
        // add smallest to result list
        // remove smallest from origin
        if (left[j] < right[k]) // left[0] < right[0]
        {
            sorted[i] = left[j];
            j++;
        }
        else
        {
            sorted[i] = right[k];
            k++;
        }
        i++;
    }
    while (j < left.size())
    {
        sorted[i] = left[j];
        j++;
        i++;
    }
    while (k < right.size())
    {
        sorted[i] = right[k];
        k++;
        i++;
    }


}
```

Merge gedeelte: O(N)

```cpp
void Tmpl8::Game::merge_sort(vector<int>& unsorted)
{
    // O(log2N) for the halving part
    // Merge sort is a divide and conquer algo which results in halving the input until it cannot be halved.
    // operations | items
    //  1 | 2
    //  2 | 4
    //  3 | 8
    //  4 | 16
    //  5 | 32
    //  6 | 64
    //  7 | 128
    //  10 | 1024
    // for n amount of items increasing exponentially, the amount of operation only increases in linear time
    //
    // When we include the merge part in our time complexity it results in: O(N) * O(log2N) = O(N*logN)

    if (unsorted.size() <= 1) // if list is empty or just one elem
    {
        return; // return cos it's already sorted
    }

    vector<int> left;
    left.reserve(unsorted.size() / 2);
    vector<int> right;
    right.reserve(unsorted.size() / 2);

    int middle = (unsorted.size() + 1) / 2; // find middle index

    for (int i = 0; i < middle; i++)
    {
        left.push_back(unsorted[i]); // push left side to left
    }

    for (int i = middle; i < unsorted.size(); i++)
    {
        right.push_back(unsorted[i]);
    }

    merge_sort(left); // divide until 0 or 1 elem is left
    merge_sort(right); // divide until 0 or 1 elem is left
    merge(left, right, unsorted); // merge left and right
}
```

Divide gedeelte: O(logN)

Merge sort totaal: O(N* logN)

Update Rockets:

```cpp
//Update rockets
for (Rocket& rocket : rockets) //N
{
    rocket.tick(); //1

    //Outer most bounds of rocket using collision radius
    int rocket_grid_left = (rocket.position.x - rocket.collision_radius) / grid.CELL_SIZE; //1
    int rocket_grid_right = (rocket.position.x + rocket.collision_radius) / grid.CELL_SIZE; //1
    int rocket_grid_top = (rocket.position.y - rocket.collision_radius) / grid.CELL_SIZE; //1
    int rocket_grid_bottom = (rocket.position.y + rocket.collision_radius) / grid.CELL_SIZE; //1

    int grid_max = grid.CELL_SIZE * grid.NUM_CELLS;


    //check if rocket is on the grid
    if (rocket.position.x < grid_max - rocket.collision_radius && rocket.position.x > 0 &&
        rocket.position.y < grid_max - rocket.collision_radius && rocket.position.y > 0)
    {
        //make sure to check for collisions within the current cell and one cell around the rocket.
        for (int x = rocket_grid_left; x <= rocket_grid_right; x++) //1
            for (int y = rocket_grid_top; y <= rocket_grid_bottom; y++) //1
                for (Tank* tank : grid.cells[x][y]) //M
                {
                    if (tank->active && (tank->allignment != rocket.allignment) && rocket.intersects(tank->position, tank->collision_radius)) //1
                    {
                        explosions.push_back(Explosion(&explosion, tank->position));

                        if (tank->hit(ROCKET_HIT_VALUE)) //1
                        {
                            smokes.push_back(Smoke(smoke, tank->position - vec2(0, 48)));
                        }

                        rocket.active = false;
                        break;
                    }
                }
    }
}
```

rockets controleren alleen op collision op de tanks binnen de opgegeven area. Dit word berekend op basis van de positie en grote van de rockets.

Ook hier is geen verandering van de Big O. Wel is dit efficiënter omdat er eerst gecontroleerd werd op collision voor elke tank op het veld.

Concurrency

```
std::vector<std::future<void>> futs;

const unsigned int threadCount = thread::hardware_concurrency(); //Asign threadCount based on available threads on this pc
ThreadPool tp(threadCount);


std::mutex myMutex;
```

Bij het starten van de simulatie wordt het beschikbare aantal threads meegegeven aan de pool.

```
for (int i = 1; i <= threadCount; i++)
{
    futs.push_back(tp.enqueue([&, i]
        {
            updateTanks(i);
            updateRockets(i);
        }));

}
//Wait for results on every future.
for each (const future<void> & fut in futs)
{
    fut.wait();
}
```

Vervolgens worden de bovenstaande functies verdeeld over de beschikbare threads, en wordt er voor elke thread gewacht op de resultaten.

```cpp
void Game::updateRockets(int i)
{
    int chunkSize = rockets.size() / threadCount;

    int mod = rockets.size() % threadCount;

    if (i <= mod)
    {
        chunkSize++;
    }

    //cout << chunkSize << endl;
    int end = i * chunkSize;//1

    for (int start = end - chunkSize; start < end; start++)
    {
        Rocket& rocket = rockets[start];//1

        rocket.tick(); //1

        //Outer most bounds of rocket using collision radius
        int rocket_grid_left = (rocket.position.x - rocket.collision_radius) / grid.CELL_SIZE; //1
        int rocket_grid_right = (rocket.position.x + rocket.collision_radius) / grid.CELL_SIZE; //1
        int rocket_grid_top = (rocket.position.y - rocket.collision_radius) / grid.CELL_SIZE; //1
        int rocket_grid_bottom = (rocket.position.y + rocket.collision_radius) / grid.CELL_SIZE; //1

        int grid_max = grid.CELL_SIZE * grid.NUM_CELLS;

        std::lock_guard<std::mutex> guard(myMutex);
        //check if rocket is on the grid
        if (rocket.position.x < grid_max - rocket.collision_radius && rocket.position.x > rocket.collision_radius &&
            rocket.position.y < grid_max - rocket.collision_radius && rocket.position.y > rocket.collision_radius)
            //make sure to check for collisions within the current cell and one cell around the rocket.
            for (int x = rocket_grid_left; x <= rocket_grid_right; x++) //1
                for (int y = rocket_grid_top; y <= rocket_grid_bottom; y++) //1
                    for (Tank* tank : grid.cells[x][y]) //M
                    {
                        if (tank->active && (tank->allignment != rocket.allignment) && rocket.intersects(tank->position, tank->collision_radius)) //1
                        {
                            explosions.push_back(Explosion(&explosion, tank->position));

                            if (tank->hit(ROCKET_HIT_VALUE)) //1
                            {
                                smokes.push_back(Smoke(smoke, tank->position - vec2(0, 48)));
                            }

                            rocket.active = false;
                            break;
                        }
                    }
    }
}
```

```cpp
void Game::updateTanks(int i)
{
    int chunkSize = tanks.size() / threadCount; // =319,75

    int mod = tanks.size() % threadCount; // = 6

    if (i <= mod)
    {
        chunkSize++;
    }

    //cout << chunkSize << endl;
    int end = i * chunkSize;//1

    for (int start = end - chunkSize; start < end; start++)
    {
        Tank& tank = tanks[start];
        if (tank.active)
        {
            //Move tanks according to speed and nudges (see above) also reload
            std::lock_guard<std::mutex> guard(myMutex);
            tank.tick();
            grid.move(&tank);
            //Shoot at closest target if reloaded
            if (tank.rocket_reloaded())
            {
                Tank& target = find_closest_enemy(tank);

                rockets.push_back(Rocket(tank.position, (target.get_position() - tank.position).normalized() * 3, rocket_radius, tank.allignment, ((tank.allignment == RED) ? &rocket_red : &rocket_blue)));

                tank.reload_rocket();
            }
        }
    }
}
```

Binnen de functies wordt de workload evenredig verdeeld over de threads. In het voorbeeld van de tanks moeten er 2558 tanks verdeeld worden. Omdat dit met bijvoorbeeld een processor van 8 threads niet eerlijk verdeeld kan worden(2558/8 = 319,75) krijgen een aantal threads 1 tank meer of minder.

Om dit te bereiken wordt in het begin van de update de modulo genomen van het aantal tanks gedeeld door het aantal threads. In dit geval is de modulo 6. Om de workload daarna juist te verdelen geven we 6 threads 1 tank meer mee.

```
320
320
320
320
319
320
319
320
```

hetzelfde doen we met de rocket Update.

Het algoritme moet een keer uitgevoerd worden per rocket. Als we het algoritme zien als 1 actie, is de span hier 1 en de work gelijk aan het aantal rockets (N). De work wordt hier evenredig verdeeld over alle beschikbare threads. In mijn geval is de hardware concurrency 16, dus de work per thread wordt verminderd van N tot N/16. Voor de updateTanks geldt hetzelfde.

Door het implementeren van deze thread pool hebben wij ons een speedup van 2.7 naar 4.0 kunnen realiseren. Wel heeft dit ook problemen met zich mee gebracht

Zo runt de simulatie sneller wanneer er niet op de windows gefocust wordt. Ook runt de simulatie sneller hoe minder cores er beschikbaar zijn. Zo is de speedup met 16 threads zo'n 3.5 en met 2 threads 4.0. Vermoedelijk omdat de processor op hogere clock speeds haalt met minder cores in gebruik. Wanneer de window in focus is, is er geen speedup.

Binnen visual studio kan de workload van verschillende functies op de cpu bekeken worden. Na het implementeren van de bovenstaande functies in een thread pool is te zien dat er bij het tekenen van de objecten nog een grote performance increase te halen valt.