DRONE SIMULATOR

Optimalisatie-opdracht P2

Rob Klein Ikink
Klas ICT SE 2B

10/12/2019

INHOUD

ONDERZOEK

1.1 Context

Het onderwerp van dit onderzoek is het versnellen van een computersimulatie.
In de simulatie worden twee groepen tanks gedurende een periode van 2 000 frames gesimuleerd. De tanks bewegen zich naar een vast punt en vuren raketten op elkaar af. Tevens zijn er lasers aanwezig die nabije tanks beschadigen.
Het doel van het onderzoek is om methoden te vinden om deze simulatie te versnellen. Het is hierbij van belang dat het verloop van de simulatie niet noemenswaardig veranderd ten opzichte van de originele situatie: alleen de tijd tussen frames mag veranderen, de inhoud van deze frames niet. Hoewel het globale verloop van de simulatie door de in dit onderzoek beschreven veranderingen niet is veranderd, zijn er enkele punten waar kleine veranderingen zichtbaar zijn. Een overzicht van deze veranderingen is terug te vinden in hoofdstuk 5.1.

1.2 Onderzoeksmethodiek

Het onderzoek is iteratief uitgevoerd. Tijdens iedere iteratie van het onderzoek is een vaste cyclus doorlopen. Door middel van profiling[1] en handmatige analyse van de code zijn aan het begin van iedere cyclus de knelpunten die de snelheid van de simulatie het meest beïnvloedden vastgesteld. Voor deze knelpunten is door middel van prototyping een oplossing bedacht en geïmplementeerd. Aan het einde van iedere cyclus is de looptijd van de simulatie gemeten (Zie ook hoofdstuk 1.3) om de effectiviteit van de oplossing te evalueren.

1.3 Testmethodiek

Om de effectiviteit van de toegepaste veranderingen effectief te kunnen evalueren, is het belangrijk dat de snelheid van de simulatie op een betrouwbare manier wordt gemeten. Om dit te garanderen zijn de volgende maatregelen genomen:

1. Met uitzondering van het benchmarken van de originele versie (Zie ook hoofdstuk 3.1) is voor iedere test dezelfde compiler gebruikt, aangeroepen met dezelfde argumenten:

Originele simulatie:	
Compiler:	MSVC
Standard library:	MSVC STL
Versie:	Toolset v142 (Compiler versie 14.2)
C++ versie:	C++14
Target:	x86_64 / Release 64-bit Windows executable zonder debug symbolen.

Veranderde simulatie:	
Compiler:	clang-cl
Standard library:	MSVC STL
Versie:	8.0.0
C++ versie:	C++17
Target:	x86_64 / Release 64-bit Windows executable zonder debug symbolen.

1 Voor het profilen van de software is gebruik gemaakt van de Visual Studio profiler en NVIDIA Nsight.

Veranderingen aan compiler flags:		
Compiler flag:	**Originele simulatie:**	**Veranderde simulatie:**
Spectre mitigatie	Ja	Nee
Optimalisaties	/Ox	/O2
C++ exceptions	Ja	Nee
Floating point exceptions	Ja	Nee
Runtime Type Information	Ja	Nee
SIMD Target Architectuur	Niet gespecificeerd	AVX2

2. Iedere test is uitgevoerd op dezelfde computer.

3. Voorafgaande aan iedere test zijn alle niet noodzakelijke achtergrondprogramma's afgesloten. Door middel van Windows Taakbeheer is nagegaan dat er geen andere processen actief waren die een significante impact op de snelheid van de simulatie konden hebben.

4. Iedere test is minimaal 3 keer uitgevoerd en het gemiddelde van deze tests is als resultaat genomen.

VERANDERINGEN

De onderstaande tabel bevat een globaal overzicht van de veranderingen die zijn toegebracht aan de simulatiecode. Voor iedere belangrijke verandering bestaat een apart subhoofdstuk met een verdere beschrijving. De locatie hiervan is terug te vinden in de *'details'* kolom.

Overzicht veranderingen	**Runtime:**	**% van originele runtime**	**Details:**
Originele simulatie (Referentiebenchmark)	01:13.520	100.00%	N/A
De compiler is veranderd naar clang-cl. Sommige compiler flags zijn aangepast. GLM is gebruikt ter vervanging van de aanwezige vectorklassen. Abseil is gebruikt ter vervanging van de standaard hashmapklasse.	00:59.180	80.50%	Hfst. 3.1
Er is een nieuwe renderer geschreven. Er is een systeem gecreëerd voor compile-time polymorfisme. De bestaande code is compleet gerefactord.	00:29.112	39.60%	Hfst. 3.2 Hfst. 3.3
Er is een systeem van spatial partitioning gecreëerd om sneller naar nabije tanks te zoeken.	00:07.872	10.71%	Hfst. 3.4
Het updaten van entities is geparallelliseerd door middel van std::async.	00:05.923	8.06%	Hfst. 3.5
Healthbars worden op de GPU gegenereerd uit de reeds bestaande tankbuffer. De insertion sort die de tankvector sorteert is geoptimaliseerd.	00:03.411	4.64%	Hfst. 3.6

UITVOERING

3.1 Setupveranderingen

De originele versie van de simulatie werd verstrekt als een Visual Studio solution met vooraf ingestelde instellingen. De geleverde configuratie was echter niet optimaal en significante tijdswinst kon behaald worden door bepaalde waarden te vervangen.

Spectre Mitigatie

Spectre is een exploit die het op bepaalde Intel CPUs mogelijk maakt om afgeschermde delen van het computergeheugen te lezen door middel van een timingaanval. (Abu-Ghazaleh et al., 2019)
Omdat de simulatie geen gevoelige data bevat is het niet nodig om deze hiertegen te beschermen.

Optimalisaties

In de originele versie van de simulatie was de optimalisatie flag op /Ox gezet. Hiermee worden bepaalde optimalisaties zoals *frame pointer emission*, *function level linking* en *string deduplication* niet toegepast. Met de flag /O2 gebeurt dit wel. (Microsoft Docs, 2017)

C++ Exceptions

Om exceptions correct te doen functioneren, is het in bepaalde gevallen nodig voor de compiler om extra code te genereren. Dit heeft een niet insignificante overhead. (Sutter, 2019)

Floating Point Exceptions

Floating point exceptions kunnen gegenereerd worden door illegale operaties op floating point getallen, zoals delen door nul. Door deze optie uit te zetten resulteren dit soort berekeningen in NaN of ±infinity. (Microsoft Docs, 2018)

Runtime Type Information (RTTI)

Runtime type information is een feature van C++ waarmee tijdens runtime bepaalde informatie van polymorfische klassen verkregen kan worden. Dit is nodig om casts te doen waarvan de validiteit niet tijdens compile time kan worden vastgesteld.
Aangezien in de simulatiecode geen gebruik is gemaakt van (dynamisch) polymorfisme, is het niet nodig om deze feature aan te laten.

SIMD Target Architectuur

SIMD instructies hebben de mogelijkheid code te versnellen door gebruik te maken van speciale hardware in de CPU die dezelfde operatie op meerdere variabelen tegelijkertijd kan uitvoeren. Door de SIMD target architectuur te specificeren kan de compiler van de aanwezigheid van deze hardware uitgaan en snellere code genereren.

Compiler

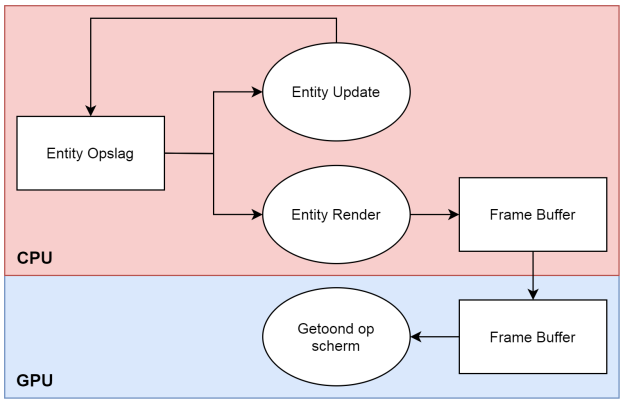
Naast de compilerinstellingen is ook de compiler zelf vervangen. De originele simulatie werd met MSVC gecompileerd. Hoewel dit de meestgebruikte compiler is voor Windows, produceert deze vaak suboptimale assembly output in vergelijking met andere compilers. (Cox, 2019)
Een ander probleem met de MSVC compiler is het lage niveau van compliance met de C++ standaard, wat vooral bij het compilen van templatecode voor problemen kan zorgen.

Gebruik van GLM
In zowel de originele simulatie als de vernieuwde versie wordt veel met vectoren gewerkt. In de originele template code is een poging gedaan om dit met SIMD instructies te accelereren. GLM heeft hier een uitgebreidere implementatie van. (G-Truc Creation, z.d.)
Daarnaast is GLM specifiek gecreëerd voor gebruik met OpenGL, wat het implementeren van een nieuwe renderer vergemakkelijkt. (Zie ook hoofdstuk 3.2)

Gebruik van Abseil Hashmap
Op bepaalde plaatsen in de simulatie is gebruik gemaakt van hash maps. Hierbij is gekozen om gebruik te maken van de Abseil Hashmap in plaats van de hashmap die door de standard library verstrekt wordt, omdat deze laatste over het algemeen trager is. (Ankerl, 2019)

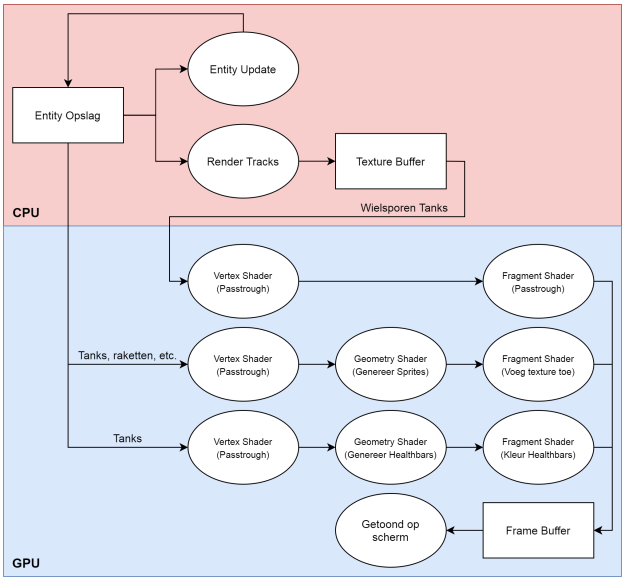
3.2 Renderer

Na het analyseren van de originele code werd het al snel duidelijk dat de gebruikte renderer niet efficiënt gebruik maakte van de GPU. De code werkt met een framebuffer in CPU-geheugen, die pixel voor pixel wordt gegenereerd en pas na voltooiing naar de GPU wordt gestuurd.



Afb. 1: originele renderer

In de nieuwe code wordt een OpenGL renderer gebruikt waarbij de buffer met entity data zonder verdere verwerking naar de GPU wordt geüpload, waarna deze via shaders verder omgezet kan worden naar de te weergeven pixels.
Het gebruik van OpenGL biedt tevens de mogelijkheid om op een later tijdstip gemakkelijk gebruik te maken van OpenCL, zonder dat de verwerkte data meerdere keren tussen CPU- en GPU-geheugen moet worden overgezet.



Afb. 2: nieuwe renderer

Voor het renderen van de tanksporen is een gewijzigde versie van de oude techniek gebruikt, omdat hier permanente blending met de achtergrondtexture nodig was. Het alternatief hiervoor was iedere wijziging in de texture pixel voor pixel naar het GPU-geheugen te sturen, (wat erg langzaam is) of een continu groeiende buffer te gebruiken, wat de simulatie met verloop van tijd steeds verder zou vertragen.

Door het gebruik van deze renderer wordt de span van het renderproces verkleind van proportioneel aan het aantal entities naar proportioneel aan het aantal entity types.

3.3 Compile-time Polymorfisme

Aan het gebruik van dynamisch polymorfisme zijn onlosmakelijk performanceverslechteringen verbonden: doordat functie-aanroepen via een vtable moeten plaatsvinden is er altijd een extra niveau van indirectie en doordat de exacte functie die zal worden aangeroepen tijdens compile time nog niet bekend is, kunnen optimalisaties zoals inlining niet plaatsvinden.

In C++17 is het mogelijk om via een templatesysteem toch een vorm van polymorfisme tijdens compile time te imiteren: in plaats van virtuele methoden kan gebruik worden gemaakt van een *curiously recurring template pattern* (CRTP) en in plaats van een container met pointers naar een baseklasse kan een tuple van containers met gederiveerde klassen worden gecreëerd, waarover met een template methode of een lambda geïtereerd kan worden alsof het één container is.
Om zo'n container te kunnen genereren is het nodig dat alle gederiveerde klassen tijdens compile time bekend zijn. Dit elimineert dus de mogelijkheid om gebruik te maken van *dynamic linking* om nieuwe klassen toe te voegen. Hiervan is echter geen sprake in de simulatiecode.

Dit systeem geeft tevens mogelijkheden waarvoor normalerwijze RTTI of zelfs reflectie nodig zouden zijn: het is mogelijk om te casten van de CRTP-baseklasse naar de gederiveerde klasse zonder *dynamic_cast* te gebruiken en om over de lijst van gederiveerde klassen te itereren.

Hoewel dit systeem geen directe performanceverbetering brengt, (in de originele code was geen polymorfisme gebruikt.) geeft het wel de mogelijkheid om kortere code te schrijven, zoals weergeven in de voorbeelden hieronder.[2]

Voorbeeld 1: Creatie van een klassenlijst

```
template <typename D> class Animal {
public:
    std::string_view noise(void) { return D::noise(); }
};


class Cat : public Animal<Cat> {
public:
    std::string_view noise(void) { return "meow"; }
};


class Dog : public Animal<Dog> {
public:
    std::string_view noise(void) { return "woof"; }
};


using Animals = DerivedClassList<Animal, Cat, Dog>;
```

Voorbeeld 2: Creatie van een polymorfische container

```
void create(void) {
    auto animal_vec = Animals::PolyVector<>();
    animal_vec.push_back(Cat());
    animal_vec.push_back(Dog());
}
```

Voorbeeld 3: Selectieve iteratie

```
struct CatSelector {
    template <typename T, std::size_t N>
    constexpr static bool select(void) {
        return std::is_same_v<typename T::value_type, Cat>;
    }
};


void iterate(void) {
    auto animal_vec = Animals::PolyVector<>();

    // Het is tevens mogelijk meerdere types tegelijk te selecteren.
    animal_vec.select<CatSelector>().forEach([](auto& cat) {
        std::cout << "Cat goes " << cat.noise() << '\n';
    });
}
```

Voorbeeld 4: Functie-aanroep met polymorfische parameter

```
template <
    typename Animal,
    typename = std::enable_if_t<Animals::Contains<Animal>()>
> void make_noise(Animal& animal) {
    std::cout << animal.noise() << '\n';
}
```

2 Bij voorbeeld 2 is het nodig om handmatig de methoden van een container te wrappen. In C++23 zal het waarschijnlijk mogelijk zijn om dit automatisch te doen door middel van compile time reflectie. (Sankel, 2019)

3.4 Spatial Partitioning

Tijdens iedere tick van de simulatie zoekt iedere tank naar nabije tanks voor collision detection. Tevens zoekt iedere raket naar nabije vijandige tanks. Als er een nieuwe raket wordt gelanceerd wordt daarnaast naar de dichtstbijzijnde vijandige tank gezocht.
Wat al deze queries gemeenzaam hebben is dat ze een tijdscomplexiteit hebben van $O(n^2)$: iedere tank of raket bekijkt iedere andere tank om te kijken welke nabij zijn en om de dichtstbijzijnde tank te vinden moeten alle afstanden met elkaar worden vergeleken.

Door de simulatiewereld op te delen in *partitions* is het mogelijk om deze queries dramatisch te versnellen. Gegeven een zoekradius r, een partitionlengte c en de maximale tankdichtheid ρ, (ontstaan door het feit dat tanks collision detection ondervinden) is het maximale aantal tanks dat bekeken moet worden voor een *nearby* query $\lceil (2r/c)^2 \rceil \times \rho c^2$. De tijdscomplexiteit van zo'n zoekopdracht is dus $O(r^2)$.
Aangezien r in de simulatie altijd relatief klein is, produceert dit een significante speedup.

Gegeven dat het gepartitionde gebied begrenst is,[3] is het maximale aantal partitions dat bekeken moet worden voor een *nearest neighbour* query $\lceil (r/c)^2 \rceil$ met r de zijdelengte van het gepartitionde gebied. Dit geeft wederom een tijdscomplexiteit van $O(r^2)$.
De afmetingen van het zoekgebied kunnen worden verkleind tot de maximale afstand tussen twee tanks, door het gebied bij te houden waarin de tanks zich werkelijk bevinden.
Aangezien de tanks zich in een relatief dichtbepakte groep voortbewegen kan dit het zoekgebied sterk verkleinen.

Een verdere versnelling kan worden bewerkstelligd door een macht van twee als de partition size te kiezen. Het omzetten van een positie kan dan als volgt gedaan worden:

```
// Converts a world position to a chunk index.
constexpr static u32 flatten_position(const Vec2f& v) {
    return flatten_index(Vec2ui(v + (float) HF_AREA) >> CHUNK_EXP);
}


// Converts a chunk position to a chunk index.
constexpr static u32 flatten_index(const Vec2ui& v) {
    return v.x + (v.y << AREA_LEN_EXP);
}
```

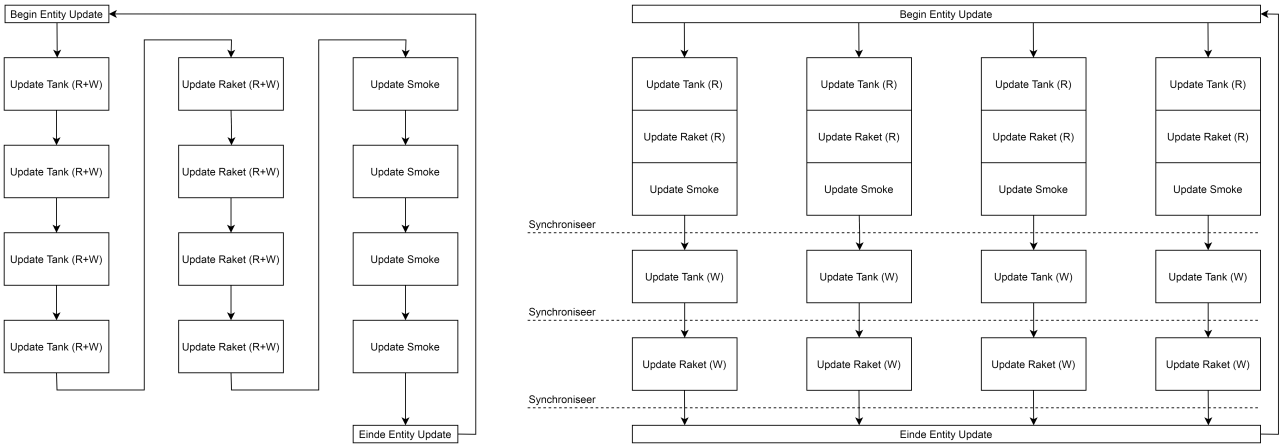
Hier zijn *HF_AREA*, *CHUNK_EXP* en *AREA_LEN_EXP* respectievelijk de helft van de lengte van het gepartitionde gebied, een getal waarvoor geldt $2^{\text{CHUNK\_EXP}} = \text{Partition Length}$ en een getal waarvoor geldt $2^{\text{AREA\_LEN\_EXP}} = \sqrt{\text{Partition Count}}$.

3 Als het gebied onbegrensd is, kan de dichtstbijzijnde tank oneindig ver weg zijn en is de tijdscomplexiteit $O(\infty)$. In werkelijkheid zou in zo'n situatie waarschijnlijk een maximale zoekradius bestaan en is de tijdscomplexiteit gelijk aan die van een *nearby* query.

3.5 Multithreading

Om de simulatie verder te versnellen is gebruik gemaakt van multithreading. Hierbij is ervoor gekozen om std::async te gebruiken. Wanneer gebruik wordt gemaakt van de MSVC STL[4] wordt in de implementatie reeds gebruik gemaakt van een *thread pool* en hoeft dit niet meer apart geïmplementeerd te worden. (Microsoft Docs, 2016)

De hoeveelheid werk die verricht moet worden om de entities in de simulatie te updaten is proportioneel aan de hoeveelheid entities die in de simulatie zijn geplaatst.
Tijdens iedere update wijzigt een tank zijn positie en leest deze de posities van andere tanks voor collision detection en om raketten af te vuren. Tevens lezen raketten de posities van tanks en wijzigen ze hun health wanneer ze ontploffen.
Om *race conditions* te voorkomen moet de update functie dus in twee delen gesplitst worden: een read-update en een write-update.
Een andere dependency ontstaat uit het feit dat tanks nieuwe raketten en smoke particles kunnen creëren. Doordat het formaat van de raket- en smokevectoren toenemen kunnen reallocaties plaatsvinden. Dit betekent dat tanks niet tegelijk met raketten en smoke particles kunnen worden geüpdatet.
Deze factoren samen geven een span van $(T_{R\ update} + T_{W\ update\ tank} + T_{W\ update\ raket})$. (Afbeelding 3)



Afb 3: Sequentiële Update (links) vs. Parallelle Update (rechts)

4 Dit is vrijwel gegarandeerd, aangezien de originele simulatie niet cross-platform is en *libc++* en *libstdc++* niet beschikbaar zijn op Windows platforms.

3.6 Healthbars

In de originele simulatie wordt een insertion sort uitgevoerd op de tankvector, waarbij de gesorteerde tanks als een nieuwe vector worden teruggegeven. Vervolgens worden uit deze gesorteerde vector de healthbars opgebouwd.

Hoewel insertion sort een slechtere tijdscomplexiteit heeft dan veel andere sorteeralgoritmes kan deze hier met enkele veranderingen toch efficiënt toegepast worden. Voor een reeds bijna gesorteerde array is insertion sort namelijk sneller dan andere algoritmes. (educative.io, z.d.)
De tankvector begint in een gesorteerde staat: alle tanks hebben dezelfde hoeveelheid health en slechts een relatief klein aantal tanks loopt iedere tick schade op.
Door de gesorteerde waarden op te slaan in de originele array in plaats van een kopie, is de tankvector dus altijd bijna gesorteerd, waardoor de insertion sort efficiënt uitgevoerd kan worden.

Het feit dat de tankvector nu gesorteerd is, maakt het tevens mogelijk om via een shader direct de healthbars te renderen uit de tankbuffer. Dit resulteert in een verdere speedup aangezien het voorkomt dat er op de CPU een transformatie van tank naar healthbar plaats moet vinden.

VERDERE MOGELIJKHEDEN

4.1 GPU Acceleratie

In de huidige versie van de simulatie wordt het updaten van de tanks op de CPU uitgevoerd. Omdat dit proces relatief makkelijk te parallelliseren is, zou het ook via OpenCL op de GPU kunnen worden uitgevoerd.
Het is mogelijk om GPU-geheugen te delen tussen OpenGL en OpenCL door gebruik te maken van een OpenGL framebuffer of texture (Shevtsov, 2014) en in de simulatiecode is reeds een systeem geïmplementeerd om OpenCL kernels te compilen en uit te voeren.

4.2 Andere Partitionsmogelijkheden

In de huidige simulatie is gebruik gemaakt van een spatial partitioning grid om de wereld op te delen. Een nadeel van deze oplossing is dat een *nearest neighbour* query relatief lang kan duren. Een KD-tree zou hiervoor mogelijk een betere oplossing kunnen zijn. Deze zou naast het huidige partitioningssysteem gebruikt kunnen worden.
Het is echter mogelijk dat de extra overhead van het bijhouden van de tree een grotere bottleneck zou kunnen vormen dan die veroorzaakt door de *nearest neighbour* queries in het grid. Dit zou eerst geprofiled moeten worden.

4.3 Compile-time Evaluatie

De simulatie verloopt deterministisch: er is geen randomisatie van de initiële waarden of updatefuncties toegepast. Dit betekent dat het volledige verloop van de simulatie van tevoren berekend zou kunnen worden en tijdens runtime als een animatie afgespeeld zou kunnen worden. Gegeven de snelheid van de huidige renderer zou dit de runtime van de simulatie naar maximaal 600 - 700 milliseconden brengen.

BIJLAGEN

5.1 Anomaliteiten

Hoewel het globale verloop van de simulatie door de aangebrachte veranderingen ongewijzigd is gebleven, zijn er enkele kleine punten waarop de nieuwe simulatie van de originele afwijkt. Een overzicht van deze anomaliteiten is hieronder weergeven.

Healthbars	In de originele simulatie wijkt de hoogte van de healthbar af van de waarde in de variabele 'HEALTH_BAR_HEIGHT'. In de nieuwe simulatie heeft de healthbar de correcte hoogte van 70 pixels.
Lettertype	De glyphs uit het origineel gebruikte lettertype hebben heterogene breedtes, wat het niet mogelijk maakt om gemakkelijk de juiste glyph uit de karakterset te verkrijgen. Via het internet was het mogelijk om het originele lettertype te achterhalen. (Digital-7) Van dit lettertype is de monospaced niet-cursieve versie gebruikt.
Frame alignment	Bij sommige sprites werd een incorrecte berekening gedaan om uv-coördinaten te genereren. Hierdoor waren soms enkele pixels van andere frames zichtbaar. Met het gebruik van de nieuwe renderer worden de correcte uv-coördinaten gegenereerd.
Collision detection	In de originele simulatie wordt het kwadraat van de som van twee radii, A en B, berekend als $(A^2 + B^2)$, in plaats van de correcte berekening $(A + B)^2$. Wanneer de correcte formule wordt gebruikt werkt het collision detection systeem niet, aangezien deze rondom deze verkeerde formule is opgebouwd. Voor berekeningen in het partition-grid is echter de correcte afstand nodig, wat verdere problemen veroorzaakt. Voor zover mogelijk is geprobeerd het originele systeem te emuleren, door de wortel van de incorrecte kwadraatafstand als de afstand te gebruiken, maar er bestaan kleine verschillen in de output van het nieuwe systeem ten opzichte van de originele situatie.

5.2 Bronvermelding

Abu-Ghazaleh, N., Ponomarev, D., & Evtyushkin, D. (2019, 28 februari). *How the Spectre and Meltdown Hacks Really Worked*. Geraadpleegd op 29 januari 2020, van https://spectrum.ieee.org/computing/hardware/how-the-spectre-and-meltdown-hacks-really-worked

Ankerl, M. (2019, 1 april). *Hashmaps Benchmarks*. Geraadpleegd op 10 december 2019, van https://martin.ankerl.com/2019/04/01/hashmap-benchmarks-01-overview/

Cox, C. (2019, 2 augustus). *Analysis - C++ Performance Benchmarks*. Geraadpleegd op 29 januari 2020, van https://gitlab.com/chriscox/CppPerformanceBenchmarks/-/wikis/Analysis

educative.io. (z.d.). *What is the best sorting algorithm for an almost sorted array?* Geraadpleegd op 30 januari 2020, van https://www.educative.io/edpresso/what-is-the-best-sorting-algorithm-for-an-almost-sorted-array

G-Truc Creation. (z.d.). *GLM Reference: Advanced Usage*. Geraadpleegd op 10 december 2019, van https://glm.g-truc.net/0.9.1/api/a00002.html#advanced_simd

Microsoft Docs. (2016, 4 november). *functions*. Geraadpleegd op 31 januari 2020, van https://docs.microsoft.com/en-us/cpp/standard-library/future-functions?view=vs-2019

Microsoft Docs. (2017, 25 september). */O Options (Optimize Code)*. Geraadpleegd op 28 januari 2020, van https://docs.microsoft.com/en-us/cpp/build/reference/o-options-optimize-code?view=vs-2019

Microsoft Docs. (2018, 9 november). */Fp (Specify floating-point behavior)*. Geraadpleegd op 28 januari 2020, van https://docs.microsoft.com/en-us/cpp/build/reference/fp-specify-floating-point-behavior?view=vs-2019

Sankel, D. (2019, 16 juli). *Working Draft, C++ Extensions for Reflection*. Geraadpleegd op 29 januari 2020, van https://cplusplus.github.io/reflection-ts/draft.pdf

Shevtsov, M. (2019, 4 oktober). *OpenCL and OpenGL Interoperability Tutorial*. Geraadpleegd op 29 januari 2020, van https://software.intel.com/en-us/articles/opencl-and-opengl-interoperability-tutorial

Sutter, H. (2019, 23 september). *De-fragmenting C++: Making Exceptions and RTTI More Affordable and Usable*. Geraadpleegd op 20 januari 2020, van https://youtu.be/ARYP83yNAWk