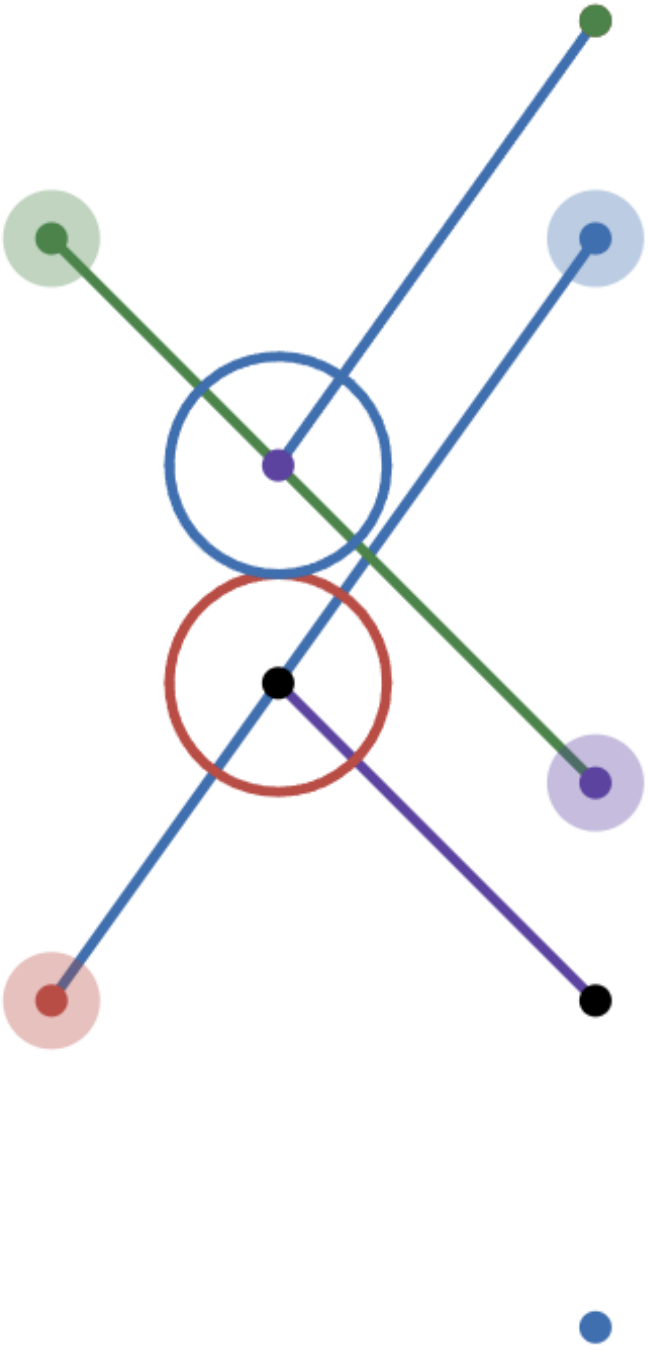

Eindopdracht Informatica V6

Een Algebraïsche 2D Physics Engine zonder Tijdstappen

Toegepast in een "One More Brick"-achtige Game

Joost Koch
Klas V6B
Het Amsterdams Lyceum

April 2025

Inhoudsopgave

1 Inleiding

1.1 Motivatie en Onderwerpkeuze

De keuze voor dit project kwam voort uit een fascinatie voor game physics en de inherente beperkingen van traditionele, op discrete tijdstappen gebaseerde simulaties. Het fenomeen 'tunneling', waarbij snel bewegende objecten door dunne muren kunnen gaan, en de accumulerende onnauwkeurigheden waren specifieke frustraties. Dit leidde tot de ambitie om een physics engine te ontwikkelen die botsingen fundamenteel anders aanpakt: volledig algebraïsch, zonder tijdstappen. De uitdaging lag in het vinden en implementeren van de exacte wiskundige oplossingen voor botsingsmomenten, wat een dieper begrip van zowel wiskunde als programmeren vereist. Het doel was een preciezere en conceptueel robuustere engine, gedemonstreerd in een game zoals "One More Brick", waar nauwkeurige bal-botsingen essentieel zijn.

1.2 Doelstelling

Het doel van deze eindopdracht was het ontwikkelen en evalueren van een functionele 2D physics engine in Python die botsingen tussen cirkels en lijnen volledig algebraïsch berekent en afhandelt (event-based), zonder gebruik te maken van discrete tijdstappen. De effectiviteit en precisie van deze engine worden gedemonstreerd door de integratie in een speelbare "One More Brick-achtige game, ontwikkeld met Pygame. Het project moest binnen de gestelde periode van de eindopdracht (ca. 3 weken effectieve werktijd) worden voltooid.

1.3 De Uitdaging

De primaire uitdaging van dit project was de overstap van een tijdstap-gebaseerde benadering naar een puur algebraïsche (event-based) simulatie. Dit omvatte:

- **Wiskundige Complexiteit:** Het afleiden, begrijpen en correct implementeren van formules voor het exacte *Time of Impact* (TOI) voor diverse botsingsscenario's (cirkel-lijn, cirkel-eindpunt, cirkel-cirkel).
- **Event Management:** Het ontwerpen van een systeem dat efficiënt de *eerstvolgende* botsing in het gehele systeem kan identificeren en de simulatie exact naar dat moment kan 'vooruitspoelen'.
- **Robuustheid:** Omgaan met floating-point onnauwkeurigheden die kunnen leiden tot gemiste botsingen of objecten die vast komen te zitten.
- **Botsingsafhandeling:** Correct toepassen van natuurkundige principes (zoals reflectie of impulsbehoud) op het exacte moment van impact.
- **Performance:** Balanceren van de computationele kosten van exacte berekeningen met de noodzaak voor een speelbare framerate.

Deze aspecten vereisten een dieper begrip van geometrie, analyse en numerieke methoden dan typische game physics implementaties.

1.4 Leeswijzer

Dit verslag beschrijft het ontwikkelproces en het eindresultaat van de algebraïsche physics engine. Hoofdstuk 2 detailleert de gekozen aanpak, de gebruikte technologieën en de kernalgoritmes. Hoofdstuk 3 beschrijft de architectuur en functionaliteiten van zowel de engine als de game. Hoofdstuk 4 evalueert het product kritisch, inclusief sterktes, zwaktes en een vergelijking met traditionele methoden. Hoofdstuk 5 bespreekt de ondervonden uitdagingen en de gevonden oplossingen. Hoofdstuk 7 vat de resultaten samen en reflecteert op het leerproces. De gebruikte bronnen zijn te vinden in de bibliografie. Een samenvatting van het logboek is te vinden in Sectie 6.

2 Methode van Aanpak

2.1 Concept: Algebraïsche (Event-Based) Simulatie

In tegenstelling tot traditionele physics engines die de toestand van objecten bijwerken met een vaste, kleine tijdstap Δt (bijv. $\mathbf{p}_{new} = \mathbf{p}_{old} + \mathbf{v} \cdot \Delta t$), hanteert deze engine een *event-based* aanpak (Sedgewick & Wayne, z.d.). De simulatie springt direct van de ene significante gebeurtenis (meestal een botsing) naar de volgende:

1. **Voorspel TOI:** Voor elk relevant paar interacterende objecten (bal-lijn, bal-bal), bereken algebraïsch het *Time of Impact* (t_{impact}), het exacte tijdstip in de toekomst waarop een botsing zou plaatsvinden als hun huidige beweging ongestoord doorgaat.
2. **Vind Eerste Event:** Bepaal de minimale t_{impact} over alle mogelijke botsingen in het systeem. Dit is de tijd tot de eerstvolgende gebeurtenis, t_{min}.
3. **Vooruitspoelen:** Verplaats *alle* objecten in het systeem exact t_{min} vooruit in de tijd naar hun posities op het moment van deze eerste botsing: $\mathbf{p}_{new} = \mathbf{p}_{old} + \mathbf{v} \cdot t_{min}$.
4. **Handel Event Af:** Pas de snelheden aan van de objecten die betrokken zijn bij de botsing(en) die plaatsvinden op t_{min}.
5. **Update Voorspellingen:** Herbereken de potentiële TOI's voor de objecten die zojuist gebotst zijn (en objecten waarmee zij nu mogelijk gaan botsen), aangezien hun snelheden zijn veranderd.
6. **Herhaal:** Ga terug naar stap 2.

Deze methode, ook bekend als *Continuous Collision Detection* (CCD) (Catto, 2013; Catto & Contributors, 2024), garandeert dat geen botsingen worden gemist, ongeacht de snelheid van objecten.

2.2 Gebruikte Technologieën

- **Programmeertaal:** Python 3.
- **Bibliotheken:**
  - `NumPy` (NumPy Developers, 2024): Essentieel voor efficiënte vector- en matrixberekeningen die de basis vormen van de geometrische en wiskundige operaties.
  - `Pygame` (Pygame Community, 2024): Gebruikt voor het creëren van het game window, het tekenen van de simulatie (visualisatie), en het afhandelen van gebruikersinvoer (muis, toetsenbord).
  - `math`: Standaard Python library voor basale wiskundige functies zoals $\sqrt{\cdot}$.

2.3 Data Structuur en Klassen

De code is modulair opgezet:

- `data_types.py`: Bevat de fundamentele klassen `Point`, `Vector`, en `Line`, die de geometrische basis vormen en gebruikmaken van NumPy arrays.
- `oneMoreBrickEngine.py`: De kern van de physics engine.
  - `MovingObject`, `Ball`: Representeren objecten met positie $\mathbf{p}$, snelheid $\mathbf{v}$, en (voor `Ball`) radius r.
  - `Collision`: Data container voor een gedetecteerde botsing (betrokken bal, botsingspunt, raakpunt, type, etc.). Cruciaal is de `time_left` eigenschap.
  - `BallLineInteraction`, `BallBallInteraction`: Klassen die de complexe logica bevatten voor het *berekenen* van de TOI voor specifieke interactietypes (zie Sectie 2.4).
  - `PhysicsEnvironment`: Orkestreert de simulatie. Beheert lijsten van objecten (`objects`) en statische structuren (`lines`, `collision_objects`). De `run_tick` methode implementeert de event-based loop.
- `oneMoreBrickGame.py`: Implementeert de specifieke game.
  - `GridCell`, `GameGrid`: Representeert het speelveld met blokken en power-ups.
  - `GameBall`, `PowerupBall`, `CollisionBall`: Spelspecifieke subklassen van `Ball` met extra eigenschappen.
  - `Renderer`: Verantwoordelijk voor het tekenen van de game state m.b.v. Pygame.
  - `Game`: Hoofdklasse van de applicatie, beheert spelverloop, levels, interactie tussen speler, engine en renderer.
- `grid_utils.py`: Hulpfuncties om het discrete `GameGrid` om te zetten naar continue `Line` objecten voor de physics engine.

2.4 Kernalgoritmes: Berekenen van Time of Impact (TOI)

2.4.1 Bal-Lijn Interactie

De interactie tussen een bal (centrum $\mathbf{p}_b$, snelheid $\mathbf{v}_b$, radius r) en een statisch lijnsegment $\mathbf{L}$ (eindpunten $\mathbf{L}_1, \mathbf{L}_2$) wordt berekend in `BallLineInteraction`. Een botsing kan optreden met de eindpunten of het lijnsegment zelf.

Botsing met Eindpunt E: We zoeken $t \geq 0$ waarvoor de afstand tussen het balcentrum op tijd t, $\mathbf{p}_b(t) = \mathbf{p}_b + \mathbf{v}_b t$, en het eindpunt $\mathbf{E}$ gelijk is aan r:

$$\|\mathbf{p}_b(t) - \mathbf{E}\|^2 = r^2$$

$$\|(\mathbf{p}_b - \mathbf{E}) + \mathbf{v}_b t\|^2 = r^2$$

Uitwerken via het dot product $(\mathbf{a}+\mathbf{b}) \cdot (\mathbf{a}+\mathbf{b}) = \mathbf{a}\cdot\mathbf{a} + 2\mathbf{a}\cdot\mathbf{b} + \mathbf{b}\cdot\mathbf{b}$ levert een kwadratische vergelijking $At^2 + Bt + C = 0$:

$$\begin{aligned} A &= \mathbf{v}_b \cdot \mathbf{v}_b = \|\mathbf{v}_b\|^2 \\ B &= 2(\mathbf{p}_b - \mathbf{E}) \cdot \mathbf{v}_b \\ C &= \|\mathbf{p}_b - \mathbf{E}\|^2 - r^2 \end{aligned}$$

De kleinste niet-negatieve oplossing t (indien $\Delta = B^2 - 4AC \geq 0$) is een potentiële TOI.

Botsing met Lijnsegment: Het vinden van het moment waarop de bal het lijnsegment raakt, is complexer. De code lijkt een methode te gebruiken die de dichtstbijzijnde punten berekent (zie `Line.closest_point`) en vervolgens via de afstand tot de bewegingslijn van de bal de t bepaalt waarbij de afstand tot het lijnstuk of eindpunt r wordt (Feronato, 2022; Leong, 2018b). Cruciaal is de check of het dichtstbijzijnde punt op de (oneindige) lijn ook daadwerkelijk *op het segment* $\mathbf{L}_1\mathbf{L}_2$ ligt.

De kleinste geldige $t \geq 0$ van alle scenario's (eindpunt 1, eindpunt 2, lijnsegment) is de TOI voor deze interactie.

2.4.2 Bal-Bal Interactie

Voor twee ballen (Bal 1: $\mathbf{p}_1, \mathbf{v}_1, r_1, m_1$; Bal 2: $\mathbf{p}_2, \mathbf{v}_2, r_2, m_2$) zoeken we $t \geq 0$ waarvoor de afstand tussen de centra gelijk is aan $r = r_1 + r_2$. De posities op tijd t zijn $\mathbf{p}_1(t) = \mathbf{p}_1 + \mathbf{v}_1 t$ en $\mathbf{p}_2(t) = \mathbf{p}_2 + \mathbf{v}_2 t$. De botsingsvoorwaarde is $\|\mathbf{p}_2(t) - \mathbf{p}_1(t)\|^2 = r^2$. Definieer relatieve positie $\mathbf{p}_{rel} = \mathbf{p}_2 - \mathbf{p}_1$ en relatieve snelheid $\mathbf{v}_{rel} = \mathbf{v}_2 - \mathbf{v}_1$. De voorwaarde wordt:

$$\|\mathbf{p}_{rel} + \mathbf{v}_{rel} t\|^2 = r^2$$

Dit leidt weer tot de kwadratische vergelijking $At^2 + Bt + C = 0$ (Aviram, z.d.; Leong, 2018a; op bronnen, 2024):

$$\begin{aligned} A &= \mathbf{v}_{rel} \cdot \mathbf{v}_{rel} = \|\mathbf{v}_{rel}\|^2 \\ B &= 2\mathbf{p}_{rel} \cdot \mathbf{v}_{rel} \\ C &= \mathbf{p}_{rel} \cdot \mathbf{p}_{rel} - r^2 = \|\mathbf{p}_{rel}\|^2 - r^2 \end{aligned}$$

Deze coëfficiënten A, B, C komen overeen met de implementatie in `BallBallInteraction.calc_collisions` en de Desmos-visualisatie (op bronnen, 2024). De kleinste niet-negatieve oplossing $t = \frac{-B - \sqrt{B^2 - 4AC}}{2A}$ (indien $B^2 - 4AC \geq 0$) is de TOI.

2.4.3 Botsingsafhandeling (Response)

Op het moment t_{min} worden snelheden aangepast.

Bal-Lijn: De snelheid $\mathbf{v}_b$ wordt gereflecteerd t.o.v. de normaal $\mathbf{N}$ van het botsingsoppervlak: $\mathbf{v}_{new} = \mathbf{v}_{old} - 2\,\text{proj}_{\mathbf{N}}(\mathbf{v}_{old}) = \mathbf{v}_{old} - 2\frac{\mathbf{v}_{old}\cdot\mathbf{N}}{\|\mathbf{N}\|^2}\mathbf{N}$. De implementatie in `Collision.calc_new_vel` berekent de nieuwe snelheid gebaseerd op de reflectie van het centrum t.o.v. het raakpunt. Een factor `collision_efficiency` wordt toegepast.

Bal-Bal: Voor elastische botsingen wordt impuls behouden. De snelheidsverandering vindt plaats langs de normaalvector $\mathbf{n} = \text{normalize}(\mathbf{p}_2(t_{min}) - \mathbf{p}_1(t_{min}))$. De formules voor de nieuwe snelheden $\mathbf{w}_1, \mathbf{w}_2$ in de code (`BallBallInteraction.calc_new_vels`) lijken gebaseerd op (Leong, 2018a), maar de exacte afleiding vereist verificatie:

$$\begin{aligned}
\mathbf{p} &= (\mathbf{v}_1 \cdot \mathbf{n})\mathbf{n} - (\mathbf{v}_2 \cdot \mathbf{n})\mathbf{n} \quad \text{(Gerelateerd aan relatieve normaal-snelheid)} \\
\mathbf{w}_1 &= \mathbf{v}_1 + \|\mathbf{p}\|\,\mathbf{n} \quad \text{(Let op: teken + ongebruikelijk?)} \\
\mathbf{w}_2 &= \mathbf{v}_2 - \|\mathbf{p}\|\,\mathbf{n} \quad \text{(Let op: teken - ongebruikelijk?)}
\end{aligned}$$

Standaard impuls formules (op bronnen, 2024) zijn: $j = \frac{-2(\mathbf{v}_1 - \mathbf{v}_2)\cdot\mathbf{n}}{1/m_1 + 1/m_2}$, $\mathbf{w}_1 = \mathbf{v}_1 + j/m_1\mathbf{n}$, $\mathbf{w}_2 = \mathbf{v}_2 - j/m_2\mathbf{n}$. De implementatie gebruikt $m_1 = m_2 = 1$ (impliciet) en een mogelijk vereenvoudigde of andere impulsberekening.

3 Het Product: Engine & Game

3.1 Architectuur Overzicht

Het product bestaat uit twee hoofdcomponenten: de herbruikbare, generieke `oneMoreBrickEngine.py` en de specifieke game-implementatie `oneMoreBrickGame.py`. De `Game` klasse gebruikt een instantie van `PhysicsEnvironment` om de fysica te simuleren en een `Renderer` om de staat te visualiseren. Data types zijn gescheiden in `data_types.py` en grid-conversie in `grid_utils.py`.

3.2 Functionaliteiten Physics Engine

- Algebraïsche (event-based) simulatie zonder tijdstappen.
- Continue botsingsdetectie (CCD) en exacte TOI-berekening voor:
  - Bewegende cirkel vs. statisch lijnsegment (incl. eindpunten).
  - Bewegende cirkel vs. bewegende cirkel.
  - Bewegende cirkel vs. statische cirkel (gebruikt voor power-ups/obstacles).
- Elastische botsingsafhandeling (reflectie/impuls-gebaseerd).
- Beheer van meerdere dynamische `Ball` objecten.
- Beheer van statische `Line` objecten en `collision_objects`.
- Identificatie en afhandeling van het eerstvolgende event in het systeem.
- Optionele functionaliteiten: zwaartekracht, instelbare botsingsefficiëntie.
- Mechanisme voor het corrigeren van kleine overlappingen (`fix_clipping`).

3.3 Functionaliteiten Game ("One More Brick Clone")

- Grid-gebaseerd speelveld (`GameGrid`).
- Vernietigbare blokken met levenspunten (`GridCell.value`).
- Diverse blokvormen (vierkant, driehoeken - type 1-5).
- Vaste cirkelvormige obstakels (type 6).
- Power-ups met diverse effecten (extra bal, laserlijnen, etc. - type < 0).
- Richt- en schietmechanisme voor ballen vanuit een vast punt.
- Level progressie: nieuwe rij blokken/power-ups per ronde.
- Game over conditie (blokken bereiken onderkant).
- Visuele feedback en interactie via Pygame (`Renderer`).
- Checkpoint systeem gebaseerd op volledig legen van het veld.

4 Evaluatie

4.1 Mogelijkheden en Sterke Punten

- **Precisie:** Het grootste voordeel. Botsingen worden exact berekend en afgehandeld op het correcte moment, waardoor 'tunneling' effectief wordt geëlimineerd, zelfs bij hoge snelheden. Dit is cruciaal voor de beoogde game.
- **Determinisme:** Gegeven dezelfde initiële condities en input, levert de simulatie altijd exact hetzelfde resultaat, wat testen en debuggen vergemakkelijkt.
- **Stabiliteit:** De event-based aanpak is inherent stabieler dan tijdstap-methoden die kunnen exploderen bij grote stappen of stijve interacties.
- **Conceptuele Correctheid:** De methode modelleert de fysica op een fundamenteler niveau (continue tijd) dan discrete benaderingen.

4.2 Beperkingen en Zwakke Punten

- **Performance:** Het berekenen van alle paarsgewijze TOI's kan computationeel intensief zijn. Zonder optimalisaties zoals spatial partitioning, schaalt de complexiteit voor n objecten naar $O(n^2)$ voor object-object interacties. Dit kan merkbaar worden bij een zeer groot aantal ballen of complexe levels.
- **Complexiteit Implementatie:** De wiskunde en de logica van de event-loop zijn aanzienlijk complexer dan een eenvoudige tijdstap-integrator.
- **Floating-Point Gevoeligheid:** Ondanks de exacte aanpak, blijft de implementatie gevoelig voor afrondingsfouten in floating-point getallen. Dit kan leiden tot 'bijna'-botsingen die net wel of net niet gedetecteerd worden, of kleine overlappingen na een botsing. De noodzaak voor `fix_clipping` bevestigt dit.
- **Uitbreidbaarheid:** Het toevoegen van complexere fysica (rotatie, frictie, complexere vormen zoals polygonen) is significant moeilijker in een puur algebraïsche/event-based engine dan in een tijdstap-engine waar benaderingen makkelijker toe te passen zijn.
- **Bal-Bal Realisme:** De huidige botsingsafhandeling lijkt vereenvoudigd (zie Sectie 2.4). Voor perfect realisme is correcte impuls- en energiebehoud met massa vereist.

4.3 Vergelijking met Tijdstap-gebaseerde Engines

Tabel 1: Vergelijking Algebraïsch vs. Tijdstap

Aspect	Algebraïsch (Event-Based)	Tijdstap-gebaseerd
Precisie	Hoog (exact TOI)	Beperkt (afhankelijk van Δt)
Tunneling	Geen	Mogelijk bij hoge snelheid / kleine Δt
Stabiliteit	Hoog	Kan instabiel worden
Complexiteit Impl.	Hoog (wiskunde, event loop)	Lager (integratie formules)
Performance (n obj.)	Potentieel $O(n \log n)$ - $O(n^2)$	Vaak sneller voor veel obj. (benadering)
Uitbreidbaarheid	Moeilijker (complexe fysica)	Makkelijker (benaderingen toevoegen)

De algebraïsche aanpak is superieur waar absolute precisie en stabiliteit vereist zijn (zoals in deze game), maar inferieur waar performance bij zeer veel objecten of de eenvoud van implementatie de doorslag geeft (Wikipedia Contributors, 2024).

4.4 Voldoet het Product aan de Doelstelling?

Ja, het ontwikkelde product voldoet aan de gestelde doelstelling. Er is een functionele 2D physics engine gerealiseerd die botsingen algebraïsch berekent zonder discrete tijdstappen. Deze engine is succesvol geïntegreerd in een speelbare "One More Brick-achtige game die de precisie van de engine demonstreert. De kernuitdagingen van de algebraïsche aanpak zijn aangegaan en geïmplementeerd.

5 Ondervonden Problemen en Oplossingen

Gedurende het project (zie logboek in Sectie 6) kwamen diverse uitdagingen naar voren:

- **Wiskundige Hürden:** Het correct afleiden en implementeren van de TOI-formules, met name voor bal-bal interactie, was complex. *Oplossing:* Grondig online onderzoek (o.a. (Aviram, z.d.; Leong, 2018a, 2018b)), gebruik van visualisaties zoals Desmos (op bronnen, 2024) ter verificatie, en stapsgewijze implementatie en testen.
- **Floating-Point Precisie Issues:** Kleine afrondingsfouten zorgden soms voor onverwacht gedrag, zoals ballen die 'vast' leken te zitten of net door objecten heen gingen na een berekende botsing. *Oplossing:* Implementatie van de `fix_clipping` routine die na elke stap controleert op overlapping en objecten fysiek uit elkaar duwt. Gebruik van `math.isclose` waar mogelijk in plaats van exacte gelijkheidstests.
- **Event Management Complexiteit:** Het correct beheren van de 'wachtrij' van toekomstige botsingen en het updaten ervan na elke afgehandelde botsing was een uitdaging. *Oplossing:* Implementatie van de `run_tick` loop die iteratief de eerstvolgende actie (kleinste `time_left`) selecteert, afhandelt, en de lijst van `collisions` bijwerkt door relevante nieuwe TOI's te berekenen en oude/ongeldige te verwijderen. De `collisions_per_ball` limiet voorkomt oneindige loops.
- **Debugging:** Het vinden van de oorzaak van incorrect gedrag in een event-based systeem is lastig, omdat de simulatietijd niet lineair verloopt. *Oplossing:* Intensief gebruik van visuele debugging in Pygame (tekenen van voorspelde paden, botsingspunten) en gerichte print statements. Testen met minimale, controleerbare scenario's.

6 Logboek

Uitgevoerde Activiteiten (Per Dag)

Tabel 2: Logboek Uitgevoerde Activiteiten (Per Dag)

Datum	**Waar**	**Tijd (Ca.)**	**Activiteit & Reflectie (STARR - Vul aan!)**
19-Mar-2025	Thuis/School	0.5u	Start project, onderwerp brainstormen.
20-Mar-2025	Thuis/School	0.5u	Onderwerp verder overwegen, globale aanpak.
21-Mar-2025	Thuis/School	0.5u	Definitieve keuze algebraïsch, ruwe planning.

Vervolg op volgende pagina

Tabel 2 vervolg

Datum	Waar	Tijd (Ca.)	Activiteit & Reflectie
22-Mar-2025	Thuis/School	0.5u	Planning verfijnen, nadenken over eerste stappen.
23-Mar-2025	Thuis/School	0.5u	Benodigde technologieën/libraries checken.
24-Mar-2025	-	0u	(Geen activiteit)
25-Mar-2025	-	0u	(Geen activiteit)
26-Mar-2025	Thuis	1u	Diepgaand onderzoek algebraïsche botsingen (CCD, TOI), bronnen verzamelen.
27-Mar-2025	Thuis	1u	Onderzoek vervolgen, links/tutorials (Leong, Toptal) bekijken.
28-Mar-2025	Thuis	1u	Implementatie Data Types (`Point`, `Vector`, `Line`) met NumPy, basis tests.
29-Mar-2025	Thuis	1u	Wiskunde Bal-Lijn: afleiden/controleren kwadratische vergelijking eindpunt botsing.
30-Mar-2025	Thuis	0.5u	Wiskunde Bal-Lijn: aanpak lijnsegment botsing uitwerken. Start implementatie `BallLineInteraction`.
31-Mar-2025	Thuis	1.5u	Afronden `BallLineInteraction` (`calc_collisions`, `calc_new_vel`), testen met simpele gevallen (horizontale/verticale lijn).
01-Apr-2025	Thuis	1.5u	Wiskunde Bal-Bal: Analyseren Desmos-link (op bronnen, 2024), relatie met kwadratische formule `t` en impuls/snelheidsformules (`w1`,`w2`) begrijpen.
02-Apr-2025	**Mediatheek**	**1u 40m**	**Implementatie** `BallBallInteraction`: Code schrijven voor `calc_collisions` en `calc_new_vels`. **Testen:** Direct getest in simpele Pygame-opzet. **Reflectie:** Efficiënt gewerkt tijdens lesuren, basis werkt.
03-Apr-2025	Thuis	1.5u	Implementatie `PhysicsEnvironment`: Opzet `run_tick` loop, `active_actions` logica, `travelled_time` beheer.
04-Apr-2025	Thuis	1.5u	Integratie Pygame: `Renderer` klasse opzetten, basis visualisatie van ballen en lijnen uit de engine.

Vervolg op volgende pagina

Tabel 2 vervolg

Datum	Waar	Tijd (Ca.)	Activiteit & Reflectie
05-Apr-2025	Thuis	0.5u	Game Logic: `Game` klasse, `GameGrid`/`GridCell`, `grid_utils` gebruiken om lijnen te genereren, basis game state.
06-Apr-2025	Thuis	1.5u	Game Logic: Power-up types definiëren en logica in `register_collision` toevoegen, level progressie (nieuwe rij).
07-Apr-2025	Thuis	1u	Testen & Debuggen: Game spelen, randgevallen zoeken (botsingen hoeken, snelle ballen), `fix_clipping` nalopen, bugs oplossen.
08-Apr-2025	Thuis	1u	Schrijven Verslag: Hoofdstuk 1 (Inleiding), Hoofdstuk 2 (Methode), focus op gedetailleerde uitleg kernalgoritmes (Sectie 2.4).
09-Apr-2025	Thuis	1u	Schrijven Verslag: Hoofdstuk 3 (Product), 4 (Evaluatie), 5 (Problemen), 6 (Conclusie). Logboek opschonen en reflecties aanvullen.
10-Apr-2025	Thuis	0.5u	Afronden: Laatste checks verslag/logboek, PDF genereren, ZIP maken, inleveren.
Totaal Geschat:		**20u**	

7 Conclusie

7.1 Samenvatting Resultaten

Deze eindopdracht heeft succesvol geresulteerd in de ontwikkeling van een 2D physics engine die fundamenteel verschilt van traditionele methoden door botsingen volledig algebraïsch en event-based af te handelen. De engine berekent exacte botsingstijden (TOI) voor interacties tussen cirkels en lijnen, waardoor problemen zoals 'tunneling' worden voorkomen. De functionaliteit en precisie zijn gedemonstreerd in een werkende "One More Brick-achtige game.

7.2 Belangrijkste Leerpunten

Dit project was een waardevolle leerervaring op meerdere vlakken:

- **Toegepaste Wiskunde:** Diepgaand begrip en toepassing van lineaire algebra, geometrie en analyse voor het oplossen van concrete fysische problemen (botsingsdetectie en -afhandeling).
- **Algoritmisch Denken:** Ontwerpen en implementeren van complexe algoritmes zoals de event-based simulatie loop en TOI-berekeningen.
- **Software Engineering:** Modulaire code schrijven (scheiding engine/game), gebruik van libraries (NumPy, Pygame), versiebeheer (impliciet), en systematisch debuggen.
- **Physics Engines:** Inzicht verkregen in de fundamentele werking, voordelen en nadelen van zowel event-based als tijdstap-gebaseerde simulaties.
- **Zelfstandigheid en Probleemoplossing:** Het succesvol doorlopen van het project vereiste doorzettingsvermogen, zelfstandig onderzoek en het overwinnen van technische en wiskundige uitdagingen.

7.3 Toekomstige Verbeteringen

Hoewel het project succesvol was, zijn er diverse mogelijkheden voor uitbreiding:

- Implementeren van rotatie voor objecten.
- Toevoegen van frictie en luchtweerstand.
- Ondersteuning voor complexere vormen (bijv. polygonen via Separating Axis Theorem aangepast voor CCD).
- Performance optimalisatie door middel van spatial partitioning (bijv. een grid of quadtree specifiek voor event prediction).
- Implementeren van meer realistische bal-bal botsingen gebaseerd op massa en correct impulsbehoud.
- Verdere uitbreiding van de game features.

Referenties

Aviram, O. (z.d.). *Video Game Physics Tutorial - Part II: Collision Detection for Solid Objects*. Verkregen april 10, 2024, van https://www.toptal.com/game/video-game-physics-part-ii-collision-detection-for-solid-objects

Catto, E. (2013). Continuous Collision. Verkregen april 10, 2024, van https://box2d.org/files/ErinCatto_ContinuousCollision_GDC2013.pdf

Catto, E., & Contributors, B. (2024). *Box2D Documentation - Simulation*. Verkregen april 10, 2024, van https://box2d.org/documentation/md_simulation.html

Feronato, E. (2022). *Continuous collision detection between a moving circle and one or more static line segments – vertex collision included*. Verkregen april 10, 2024, van https://www.emanueleferonato.com/2022/07/14/continuous-collision-detection-between-a-moving-circle-and-one-or-more-static-line-segments-vertex-collision-included/

Leong, E. (2018a). *Circle-Circle Collision Tutorial*. Verkregen april 10, 2024, van https://ericleong.me/research/circle-circle/

Leong, E. (2018b). *Circle-Line Collision Detection Tutorial*. Verkregen april 10, 2024, van https://ericleong.me/research/circle-line/

NumPy Developers. (2024). *NumPy - The fundamental package for scientific computing with Python*. Verkregen april 10, 2024, van https://numpy.org/

op bronnen, J. /. G. (2024). Desmos Graph: Ball-Ball Collision Calculation [Visualisatie van de kwadratische vergelijking en impulsberekening voor bal-bal botsing.]. Verkregen april 10, 2024, van https://www.desmos.com/calculator/zu3eygltvc

Pygame Community. (2024). *Pygame Documentation*. Verkregen april 10, 2024, van https://www.pygame.org/docs/

Sedgewick, R., & Wayne, K. (z.d.). *Event-Driven Simulation*. Verkregen april 10, 2024, van https://algs4.cs.princeton.edu/61event/

Wikipedia Contributors. (2024). *Collision detection*. Verkregen april 10, 2024, van https://en.wikipedia.org/wiki/Collision_detection