

PROJECT PROGRAMMEREN 2022

GAME CATCH THEM ALL IN C

prof. dr. ir. Filip De Turck

pgm@lists.ugent.be

2de Bachelor Ingenieurswetenschappen:
Computerwetenschappen en Elektrotechniek

Academiejaar 2021-2022

INHOUDSOPGAVE

1	Introductie	1
1.1	Doel van het project	1
1.2	Game programming	1
1.2.1	Game loop	1
1.2.2	Bewegende beelden	1
1.2.3	Collision detection	2
2	Opgave	3
3	Implementatie	4
3.1	Spelwereld	4
3.1.1	Tegels en coördinaten	4
3.1.2	LevelInfo	4
3.1.3	Level	5
3.2	Spelobjecten	6
3.2.1	Actoren	6
3.2.2	Levelobjecten	7
3.3	Game loop	9
3.3.1	Input controleren	10
3.3.2	Status aanpassen	10
3.3.3	Scherm tekenen	11
3.4	Speelbaarheid	11
3.5	Highscores bijhouden	11
4	Bestanden	13
4.1	catch_them_all.h	13
4.2	game.h	13
4.3	game.c	13
4.4	gui.h/c	13
4.5	util.h	13
4.6	highscores.h	13
4.7	highscores.c	14
4.8	level.h	14
4.9	level.c	14
4.10	main.c	14
5	Compileren en uitvoeren	15
6	Tips en opmerkingen	16
7	Indienen	17
8	Appendix I: GUI	18
9	Appendix II: Memory leaks detecteren met Address Sanitizer	19

1 INTRODUCTIE

Hieronder wordt kort het doel van het project geschetst en wordt een inleiding gegeven over een aantal aspecten die typisch aan bod komen bij game programming. Het advies is om dit zeker door te nemen zodat er voldoende achtergrond-kennis aanwezig is om met de opgave te starten.

1.1 Doel van het project

In dit project is het de bedoeling om een spel genaamd 'Catch Them All' te implementeren in C. De uitwerking van het project dient te gebeuren in groepen van twee personen. Het doel is om zoveel mogelijk aspecten van softwareontwikkeling aan bod te laten komen. Hierbij denken we behalve het programmeren zelf aan het gebruik van de Visual Studio Code editor, de integratie van bestaande libraries en het efficiënt samenwerken aan een project door middel van Git. Het project is gequoteerd op 6 van de 20 punten voor het vak Programmeren.

1.2 Game programming

Het programmeren van games verschilt op een aantal vlakken van het programmeren van traditionele applicaties. Hieronder worden de belangrijkste zaken overlopen.

1.2.1 Game loop

De game loop is het belangrijkste onderdeel van het spel aangezien deze ervoor zorgt dat het spel vlot blijft lopen, onafhankelijk van het feit of de gebruiker al dan niet input genereert. Dit in tegenstelling tot meer traditionele software-programma's zoals tekstverwerkers of browsers die enkel reageren wanneer de gebruiker een actie triggert, bijvoorbeeld het tekenen van een menu wanneer de gebruiker op een knop klikt. Een game loop in pseudocode zou er als volgt kunnen uitzien:

Algorithm 1 Een generiek voorbeeld van een game loop.

```
1: while user does not exit do
2:   Capture user input
3:   Run (simple) AI
4:   Move objects
5:   Resolve collisions
6:   Render graphics
7: end while
```

Hierbij wordt achtereenvolgens gecontroleerd welke input er door de gebruiker gegeven is, wordt de logica van het spel en de artificiële intelligentie (AI) uitgevoerd, wordt de beweging van verschillende objecten uitgevoerd, worden eventuele botsingen afgehandeld en worden de objecten opnieuw getekend. De game loop kan uiteraard verder verfijnd worden naarmate de ontwikkeling van het spel verloopt, maar de meeste spellen zijn op dit basisprincipe gebaseerd.

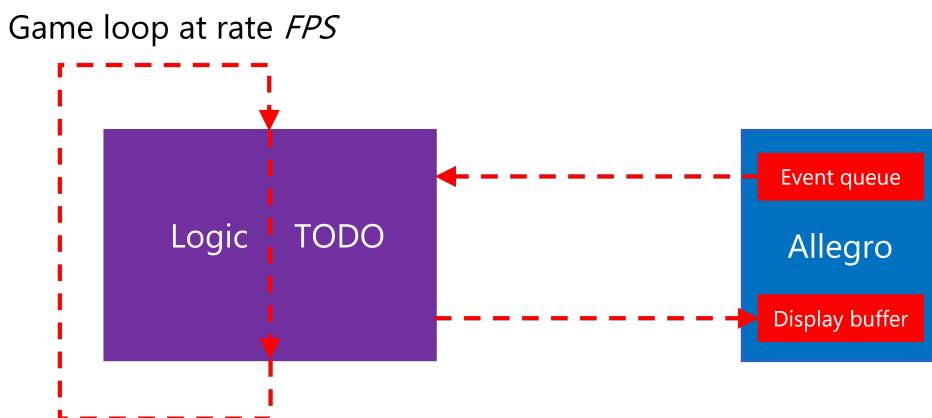
Game loops kunnen anders geïmplementeerd worden afhankelijk van het platform waarvoor ze ontwikkeld zijn. Een spel voor game consoles kan bijvoorbeeld alle hardware resources naar eigen wens gebruiken, terwijl een spel voor Windows uitgevoerd moet worden binnen de beperkingen van de process scheduler. Verder is het zo dat de meeste spellen multi-threaded zijn, zodat de berekening van de AI losgekoppeld kan worden van de generatie van bewegende beelden in het spel. Dit creëert een kleine overhead, maar zorgt ervoor dat het spel efficiënter uitgevoerd kan worden op hyper-threaded of multicore processoren.

1.2.2 Bewegende beelden

Het visuele aspect is bij games zeer belangrijk: er moet een vloeiend beeld weergegeven worden, wil men het spel er goed doen uitzien. De theorie rond hoeveel beelden per seconde het menselijk oog minimaal moet zien om iets als een vloeiende beweging waar te nemen, is veel ingewikkelder dan meestal wordt aangenomen. Er zijn een aantal vuistregels die in de meeste gevallen lijken te kloppen:

- Hoe groter het aantal frames per seconde (FPS), hoe vloeiender de beweging;
- Hoe kleiner het verschil tussen opeenvolgende frames, hoe vloeiender de beweging.

Het is gemakkelijk om de snelheid van de game loop ook in frames per seconde uit te drukken, zodat er een metriek is om verschillende implementaties te kunnen vergelijken. Het aantal frames per seconde duidt dan eigenlijk aan hoeveel keer per seconde de toestand van het spel aangepast wordt. Het aantal gegenereerde FPS is enerzijds gelimiteerd door de hardware van de computer, maar anderzijds ook door hoe de berekeningen die in de game loop gebeuren. Wanneer een bepaalde actie bijvoorbeeld een groot aantal berekeningen vraagt, zal de iteratie van de game loop langer duren en zal het aantal zichtbare FPS (tijdelijk) dalen. In dit project werken we met de open source game programming library Allegro¹² op de manier zoals weergegeven in Figuur 1. De generieke functionaliteit vereist voor grafische visualisatie van het spel wordt reeds aan jullie aangeboden, zoals initialisatie van Allegro, het ontvangen van user input, het instellen van timers, etc. Het bestuderen van deze meegeleverde code, vanaf nu gerefereerd als 'GUI', zal nodig zijn om de reeds beschikbare functies te begrijpen en correct te kunnen integreren in de spellogica.



Figuur 1: Opsplitsing tussen de spellogica enerzijds en de game programming library Allegro anderzijds.

1.2.3 Collision detection

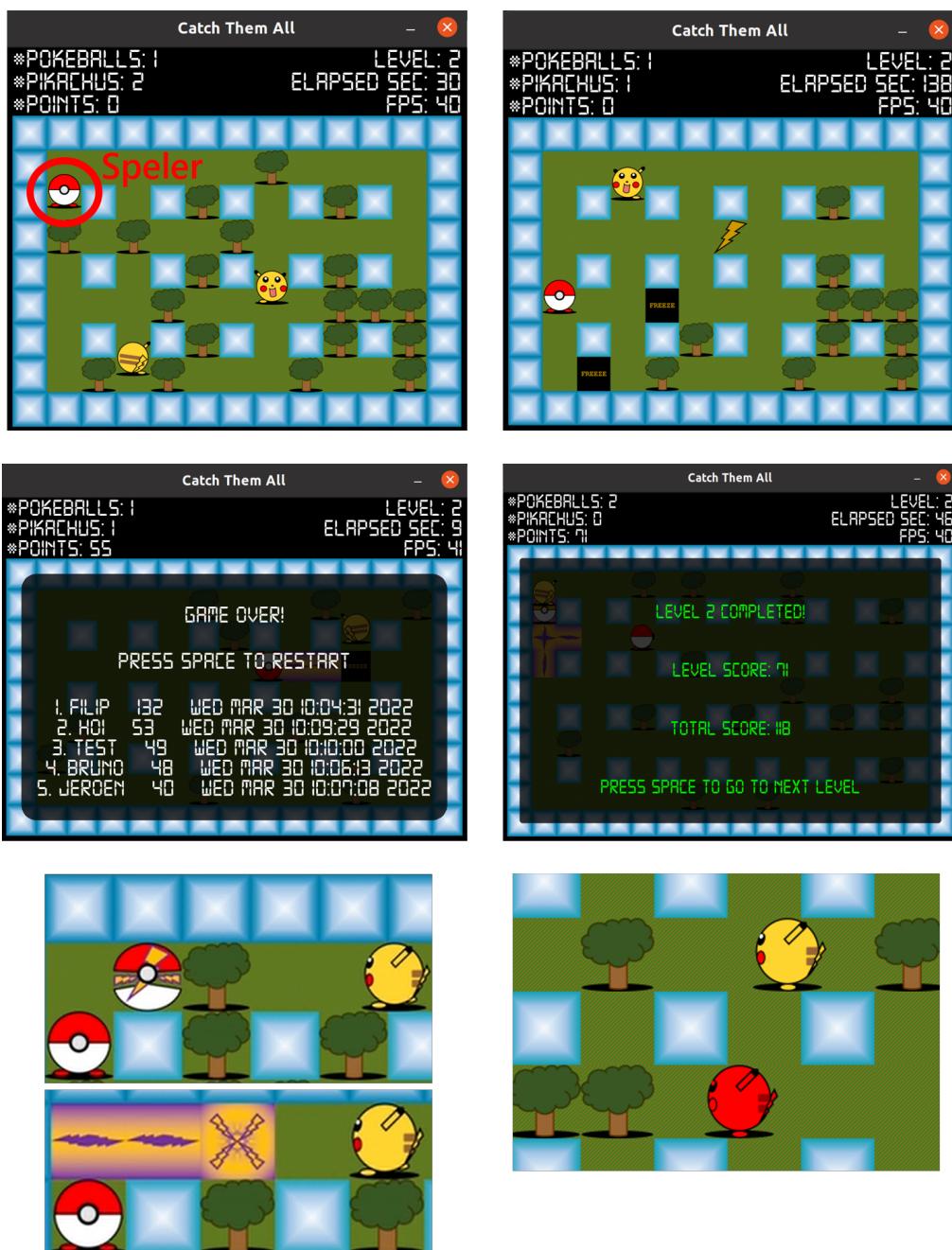
Bij games wordt dikwijls gesproken over “collision detection”, het detecteren van een botsing tussen twee of meerdere objecten. In elke iteratie van de game loop moet er, vóór het hertekenen van het nieuwe frame, gecontroleerd worden of er al dan niet een botsing veroorzaakt werd door het uitvoeren van de bewegingen van alle verschillende objecten. Dit kan bijvoorbeeld het tegen elkaar plaatsen zijn van de twee objecten in plaats van “in elkaar” of een botsing zijn die de objecten de tegenovergestelde richting opstuurt. Eens deze botsing gedetecteerd en opgelost is, kan het frame hertekend worden. In dit project zal collision detection zich beperken tot de tweedimensionale variant.

¹<https://liballeg.org>

²<https://www.allegro.cc>

2 OPGAVE

Het doel van het project is om een spel te programmeren waarin objecten, in dit geval meerdere Pikachu's, gevangen dienen te worden. De speler loop rond in een bos met daarin water en bomen dat van bovenaf bekeken wordt. De Pikachu's bewegen zich ook voort in dit bos volgens een bepaald traject. De speler kan door middel van openklikkende Pokeballs de verschillende Pikachu's vangen. Deze laten zich echter niet eenvoudig vangen, aangezien ze de speler onmiddellijk kunnen uitschakelen bij enig fysiek contact. Om door te gaan naar het volgende level dient de speler alle aanwezige Pikachu's te vangen, hetgeen dus enkel mogelijk is door de Pokeballs op strategische locaties te plaatsen. Naast het vangen van Pikachu's kunnen Pokeballs ook gebruikt worden om bepaalde obstakels uit het bos uit de weg te ruimen. Let echter op dat de speler zelf kwetsbaar is voor de geplaatste Pokeballs. Het spel eindigt dan ook wanneer de speler zichzelf laat vangen. Een eerste indruk van het spel wordt weergegeven in Figuur 2. In de volgende sectie overlopen we de verschillende onderdelen van het spel en leggen we de exacte spelregels vast.



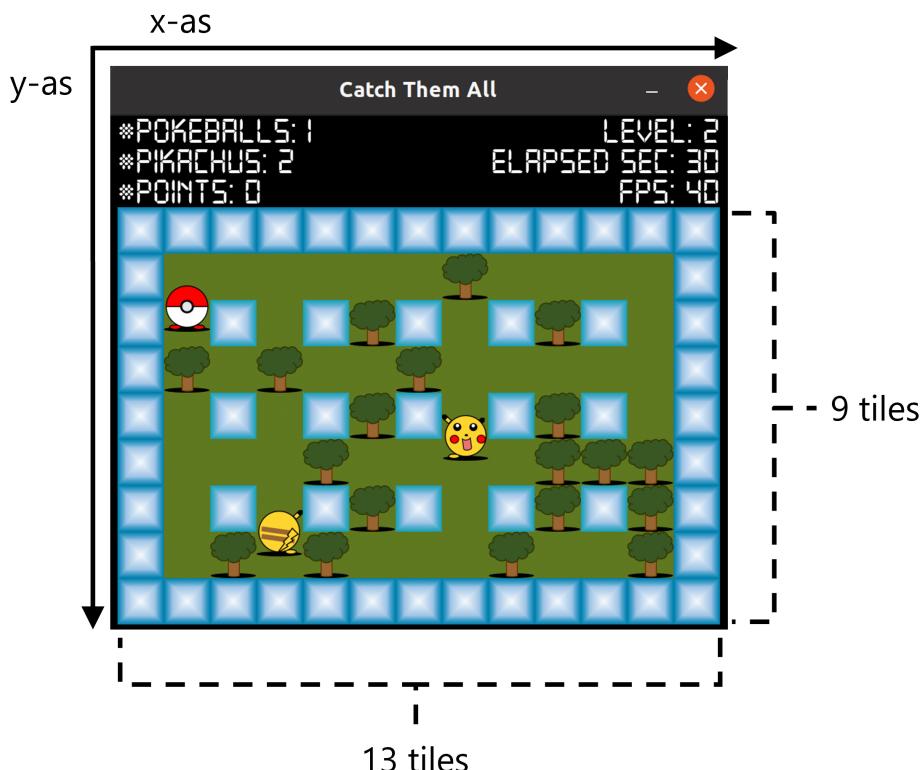
Figuur 2: Een overzicht van de te implementeren spelwereld.

3 IMPLEMENTATIE

3.1 Spelwereld

3.1.1 Tegels en coördinaten

Daar de spelwereld van bovenaan bekeken wordt, kunnen we deze in twee dimensies voorstellen waarbij men horizontaal beweegt volgens de x-as en verticaal volgens de y-as. De locatie van elk object in de wereld kan dus voorgesteld worden aan de hand van een (x,y)-coördinaat. In het gebruikte coördinatensysteem neemt de x-coördinaat toe naar rechts en de y-coördinaat neemt toe naar beneden (zie Figuur 3). Voor de eenvoudigheid delen we de wereld verder in tegels (tiles) in. Een tegel heeft een vaste breedte en hoogte zoals gedefinieerd in de macro constante `TILE_SIZE` en kan een object bevatten (zoals bv. een Pokéball of een bonus, zie 3.2.2).



Figuur 3: Indeling van de spelwereld.

Elk object heeft dus een grootte gelijk aan de `TILE_SIZE` en zal dus een (x,y)-coördinaat hebben dat een veelvoud is van deze `TILE_SIZE`. Uitzonderingen hierop zijn de speler en Pikachu's (zie 3.2.1) aangezien deze zich stapsgewijs over deze tegels kunnen verplaatsen om een vlotte beweging te garanderen.

3.1.2 LevelInfo

De struct `LevelInfo` bepaalt de eigenschappen van een level aan de hand van een aantal parameters:

- `width`: de breedte (in tegels) van het level
- `height`: de hoogte (in tegels) van het level
- `level_nr`: het levelnummer
- `fill_ratio`: een getal tussen 0 en 1, bepaalt het percentage van het level dat gevuld wordt met vangbare obstakels (zie 3.2.2.1)
- `nr_of_pikachus`: het aantal aanwezige Pikachu's

- `spawn_strong_pikachu`: bepaalt of er in dit level een extra sterke Pikachu aanwezig is (zie 3.2.1.2)
- `bonus_spawn_ratio`: een getal tussen 0 en 1, bepaalt de kans dat een bonus beschikbaar wordt op de plaats waar een vangbaar obstakel of een Pikachu gevangen werd (zie 3.2.2.3)

Instanties van `LevelInfo` worden gegenereerd door de functie `generate_level_info`, gespecificeerd in `level.h`, op te roepen. Het genereren gebeurt op een willekeurige manier maar wordt beïnvloed door het levelnummer. Op deze manier kunnen we levels steeds moeilijker maken. De precieze details van deze generatie van levels laten we aan jullie over.

Tip: Je kan de functie `rand()` uit `stdlib` gebruiken om een willekeurige integer waarde te genereren tussen nul en de maximum waarde (`RAND_MAX`). Aan de hand van de modulo-operator kunnen we een getal in een bepaald bereik verkrijgen:

```
/* Getal tussen 0 en 99 */
int random_value = rand() % 100;
```

Door de functie `srand()` op te roepen met een bepaalde waarde kan je de seed voor random generatie instellen. Opeenvolgende invocaties van `rand()` met dezelfde seed leveren dezelfde resultaten op. Dit kan een handige eigenschap zijn voor de generatie van levels (een leuke reeks van levels kan herspeeld worden door dezelfde seed in te stellen).

De seed wordt automatisch ingesteld aan de hand van de systeemklok in de meegegeven `main`. Als je zelf een seed wilt meegeven kan je dit doen via een command line parameter (eerste parameter is het level waarin gestart moet worden, tweede parameter de random seed die gebruikt moet worden).

3.1.3 Level

De struct `Level` stelt de spelwereld voor en heeft twee velden:

- `level_info`: bevat de karakteristieken van het level
- `entities`: een array voor het bijhouden van de verschillende levelobjecten (zie 3.2.2)

Elk element van de `entities` array stelt dus een tegel van het level voor. Het levelobject op de tegel met coördinaat `(t_x, t_y)` kan dan opgehaald worden uit de array door de `TILE_SIZE` in rekening te brengen:

```
level.entities[t_x / TILE_SIZE][t_y / TILE_SIZE]
```

Op een `Level` kunnen de volgende (te implementeren) operaties uitgevoerd worden:

- `void init_level(Level* level, LevelInfo level_info)`:
Initialiseert het level op basis van de meegegeven info. Concreet wordt dus de `entities` array aangemaakt op basis van de meegegeven dimensies en worden de nodige obstakels toegevoegd. (*)
- `void render_level(Level* level)`:
Tekent de volledige inhoud van het level gebruikmakend van de GUI.
- `void destroy_level(Level* level)`:
Zorgt voor het correct opruimen van het level.

(*) **Let op:** bij de initialisatie moet het level voldoen aan de volgende basiseigenschappen:

- De startpositie van de speler op coördinaat `(1 * TILE_SIZE, 1 * TILE_SIZE)` moet altijd vrij zijn. Zorg er ook voor dat de speler minstens een vakje naar rechts of naar onder kan bewegen.
- De tegels aan de rand van het level worden opgevuld met onvangbare obstakels (zie 3.2.2.1).
- Elk level bevat naast de rand een raster van onvangbare obstakels.

3.2 Spelobjecten

3.2.1 Actoren

3.2.1.1 Speler De speler wordt voorgesteld door de struct `Player` die de volgende velden bevat:

- `x`: de x-coördinaat van de speler
- `y`: de y-coördinaat van de speler
- `orientation`: de huidige oriëntatie van de speler
- `pokeball_power`: bepaalt de kracht van de Pokeballs die de speler plaatst (zie 3.2.2.2)
- `remaining_pokeballs`: bepaalt hoeveel Pokeballs de speler nog kan leggen
- `is_catched`: bepaalt of de speler zelf al dan niet gevangen is door een Pokeball

Voor een speler gelden de volgende regels:

- Een speler wordt geblokkeerd door obstakels en Pokeballs (*), maar niet door bonussen
- Als een speler in aanraking komt met een Pikachu of een vangpoging, dan is het spel gedaan
- Als een speler in aanraking komt met een bonus, dan wordt deze opgenomen (zie 3.2.2.3)
- Een speler kan Pokeballs plaatsen zolang het veld `remaining_pokeballs` groter is dan nul

(*) Behalve door de Pokeball die de speler op dat moment aan het plaatsen is.

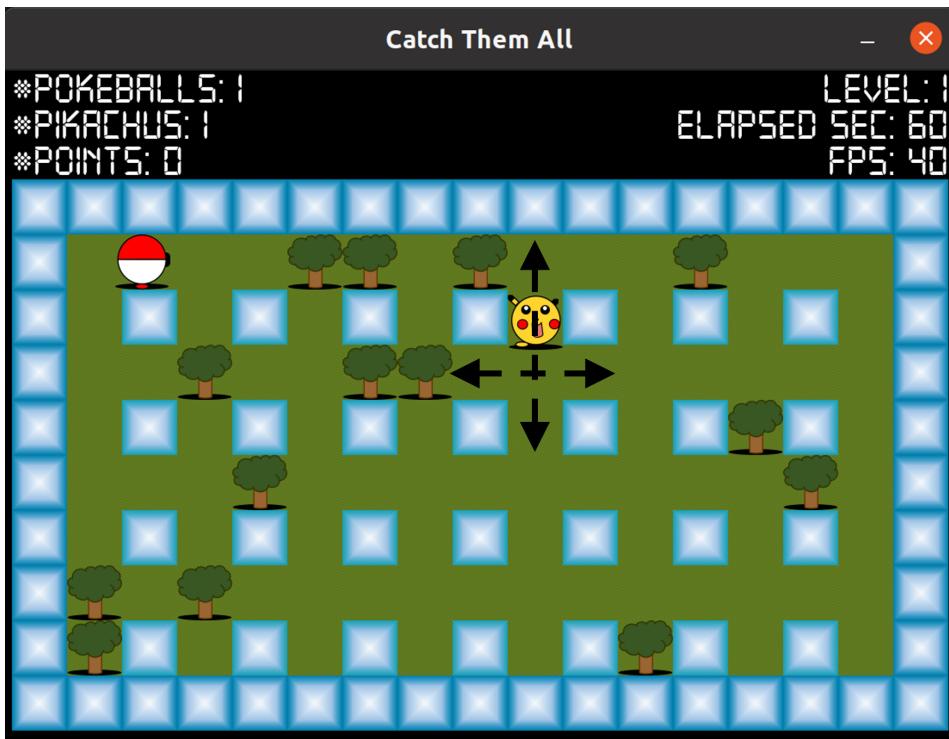
3.2.1.2 Pikachu's Pikachu's worden voorgesteld door de struct `Pikachu` die de volgende velden bevat:

- `x`: de x-coördinaat van de Pikachu
- `y`: de y-coördinaat van de Pikachu
- `orientation`: de huidige oriëntatie van de Pikachu
- `is_strong`: bepaalt of de Pikachu al dan niet extra sterk is
- `remaining_attempts`: bepaalt hoeveel vangpogingen de Pikachu nog kan weerstaan
- `is_catched`: bepaalt of de Pikachu al dan niet gevangen is
- `frozen`: bepaalt of de Pikachu al dan niet bevroren is
- `attempts`: vangpogingen die op de extra sterke Pikachu zijn toegepast
- `num_attempts`: aantal vangpogingen die op de extra sterke Pikachu zijn toegepast

Als de Pikachu extra sterk is, weergegeven door de rode kleur zoals zichtbaar in Figuur 2, dan beweegt deze sneller dan een gewone Pikachu en kan het meerdere vangpogingen overleven. Bevroren Pikachu's kunnen niet meer bewegen voor een bepaalde tijdsduur, maar blijven wel kwetsbaar voor vangpogingen.

Voor het bewegen van de Pikachu's gelden de volgende regels:

- Een Pikachu probeert elke iteratie een beweging te maken (tenzij hij ingesloten is)
- Een Pikachu wordt geblokkeerd door obstakels en Pokeballs, maar niet door bonussen
- Een Pikachu komt nooit vast te zitten in een bepaalde zone, tenzij hij ingesloten is door obstakels. Met andere woorden, als er een pad is dat hij kan bereiken, dan moet de Pikachu zo geprogrammeerd zijn, dat hij daar ooit naartoe kan gaan (zie Figuur 4).



Figuur 4: Een Pikachu moet in staat zijn om in alle mogelijke richtingen te bewegen.

Zorg er ten slotte voor dat een Pikachu start op een willekeurige vrije plaats in het level.

We laten jullie vrij om de precieze details van het bewegingsalgoritme zelf te bepalen. Hoe intelligenter en gevarieerder de Pikachu's bewegen, hoe uitdagender het spel wordt.

3.2.2 Levelobjecten

Daar een levelobject verschillende zaken kan voorstellen (een obstakel, een Pokéball), maar wel op een éénduidige manier moet kunnen opgeslagen worden in de `Level.entities` array, hebben we besloten dit te modeleren aan de hand van een union type. Union types worden vaak gebruikt in C als een truc om overerving te simuleren. De union `Entity` ziet er als volgt uit:

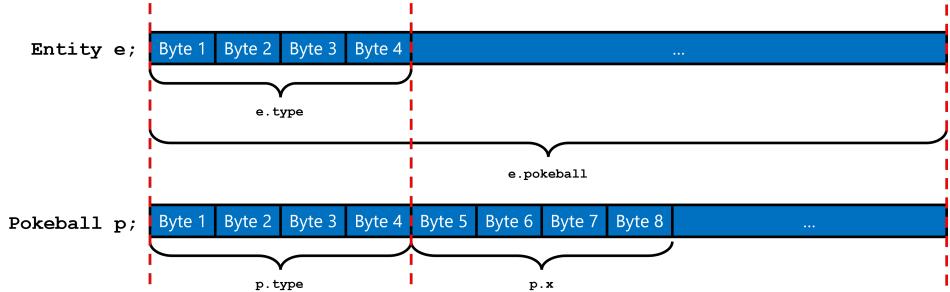
```
typedef union {
    ENTITY_TYPE type;
    Pokeball pokeball;
    CatchAttempt catch_attempt;
    Bonus bonus;
    Obstacle obstacle;
    EmptySpace empty_space;
} Entity;
```

Het veld `type` duidt aan welk type levelobject een instantie van `Entity` is en op basis hiervan weten we dan welk van de overige velden momenteel in gebruik is.

Strikt genomen zou ook `type` moeten deel uitmaken van de union en is het dus niet mogelijk dat zowel `type` als één van de overige velden ingevuld zijn. Het geheugen ingenomen door een union is echter gelijk aan de lengte van het grootste veld (in bytes uitgedrukt) en afgerond naar de woordlengte van de processor.

Als we dus voor elk van de struct velden (`pokeball`, `catch_attempt`, `bonus`,...) plaats voorzien voor een `ENTITY_TYPE` veld `type` aan het begin van de struct definitie (`Pokeball`, `CatchAttempt`, `Bonus`,...), dan zal het veld `type` in `Entity` altijd verwijzen naar de waarde die hier wordt opgeslagen. Hierdoor wordt `Entity.type` een soort van shortcut om de geheugenlocatie van het veld `type` in de structs die deel uitmaken van de union `Entity` te

bevragen. Dit principe heet allignatie en wordt visueel voorgesteld in Figuur 5.



Figuur 5: Allignatie van het type veld.

Omdat `Entity` instanties in een array zullen opgeslaan worden en er dus ook een nul-element nodig is, hebben we de struct `EmptySpace` gedefinieerd die enkel `type` en een (x,y) -coördinaat bevat.

3.2.2.1 Obstakels Obstakels in de spelwereld worden voorgesteld door de struct `Obstacle` die de volgende velden bevat:

- `type`: (zie Entity)
- `x`: de x-coördinaat van het obstakel
- `y`: de y-coördinaat van het obstakel
- `is_catchable`: bepaalt of het obstakel al dan niet kan gevangen worden

De bomen en het water, zoals afgebeeld in voorgaande screenshots zoals Figuur 3, zijn respectievelijk voorbeelden van vangbare en onvangbare obstakels.

3.2.2.2 Pokeballs en vangpogingen Een Pokeball wordt voorgesteld door de struct `Pokeball` die de volgende velden bevat:

- `type`: (zie Entity)
- `x`: de x-coördinaat van de Pokeball
- `y`: de y-coördinaat van de Pokeball
- `ticks_left`: bepaalt hoeveel iteraties van de game loop de Pokeball nog blijft staan

Als de speler een Pokeball plaatst, zullen de coördinaten soms moeten afferond worden aangezien de Pokeball altijd op een tegel wordt geplaatst. Denk goed na over hoe dit dan precies gebeurt!

Als de timer (`ticks_left`) van de Pokeball afloopt, wordt er een vangpoging uitgevoerd. Een vangpoging wordt voorgesteld door de struct `CatchAttempt` die de volgende velden bevat:

- `type`: (zie Entity)
- `x`: de x-coördinaat van de kern van de vangpoging
- `y`: de y-coördinaat van de kern van de vangpoging
- `spread`: bepaalt hoeveel blokken de vangpoging zich propageert in elk van de vier windrichtingen (*)
- `power`: de effectieve power van de vangpoging (=het veld `pokeball_power` van de speler op het moment dat de vangpoging wordt aangemaakt)
- `ticks_left`: bepaalt hoeveel iteraties van de game loop de vangpoging nog blijft staan

(*) `power` bepaalt de initiële propagatie van een vangpoging, maar de effectieve spread stopt na het vangen van de eerste Pikachu of uit de weg ruimen van een eerste obstakel.

Zolang de timer (`ticks_left`) van de vangpoging niet afgelopen is, moeten dus alle objecten binnen het bereik van de vangpoging gevangen worden. Uitzonderingen zijn obstakels die niet kunnen gevangen worden, bonussen en een extra sterke Pikachu (Pikachu waarvan het `is_strong` veld op 'waar' staat, zie 3.2.1.2). Deze laatste kan namelijk meerdere vangpogingen overleven.

3.2.2.3 Bonussen

Een bonus wordt voorgesteld door de struct `Bonus` die de volgende velden bevat:

- `type`: (zie Entity)
- `x`: de x-coördinaat van de bonus
- `y`: de y-coördinaat van de bonus
- `bonus_type`: bepaalt om welk soort bonus het gaat (zie hieronder)

Er zijn drie verschillende soorten bonussen (enum `BONUS_TYPE`):

- `EXTRA_POWER`: geeft de Pokeballs van de speler extra kracht
- `EXTRA_POKEBALL`: geeft de speler een extra Pokeball
- `FREEZE_PIKACHUS`: bevriest de Pikachu's

De maximale kracht die een Pokeball kan hebben wordt gedefinieerd door de macro constante `POKEBALL_MAX_POWER`.

Bonussen kunnen tevoorschijn komen wanneer de speler een obstakel of een Pikachu vangt. De kans dat dit gebeurt, wordt bepaald door het veld `bonus_spawn_ratio` van de `LevelInfo`.

3.3 Game loop

De basisstructuur van de game loop wordt meegegeven. In de volgende puntjes overlopen we wat er juist moet gebeuren in de verschillende stappen van de lus.

De status van het spel wordt bijgehouden in een struct van het type `Game` en wordt telkens meegegeven bij het oproepen van elke stap. De struct bevat volgende velden:

- `level`: houdt informatie over de spelwereld bij (zie 3.1)
- `player`: de speler (zie 3.2.1.1)
- `pikachus`: een array met de aanwezige Pikachu's (zie 3.2.1.2)
- `pikachus_left`: het aantal Pikachu's dat momenteel nog niet gevangen is
- `state`: de fase waarin het spel zich bevindt (gestart, game over, level uitgespeeld, etc.)
- `input`: houdt informatie over eventuele input bij
- `score`: houdt de huidige score bij

De score wordt geïncementeerd telkens de speler een Pikachu weet te vangen:

- Voor een extra sterke Pikachu (Pikachu waarvan het `is_strong` veld op 'waar' staat, zie 3.2.1.2) wordt de waarde van de constante `STRONG_PIKACHU_SCORE` bijgeteld.
- Voor een gewone Pikachu tellen we de waarde van `PIKACHU_SCORE` bij, maar trekken we de waarde van de GUI-timer (aantal seconden dat het spel bezig is) ervan af. De uiteindelijke waarde die bijgeteld wordt, mag echter niet kleiner worden dan nul.

De game loop wacht op Events van Allegro door de methode `gui_game_event()` op te roepen. Er zijn 3 mogelijke events gedefinieerd in util.h: `TimerEvent`, `DisplayCloseEvent` en `KeyDownEvent`. Het `TimerEvent` wordt gegeven wanneer de rendering van een nieuw frame noodzakelijk is. Het `KeyDownEvent` wordt gegeven wanneer een knop ingedrukt of losgelaten wordt door de gebruiker. Op basis van deze events moeten de correcte handelingen gebeuren die hieronder beschreven staan:

3.3.1 Input controleren

Een eerste stap is het controleren van eventuele spelerinput. De methode `check_game_input` controleert de input aan de hand van de GUI-methodes en slaat dan de verwerkte gegevens op in de struct `Input` zoals beschikbaar in de struct `Game`.

Het `KEY_DOWN` event bevat een positieve integer waarmee achterhaald kan worden welke toetsen ingedrukt werden (of 0 als er geen toetsen werden ingedrukt). Deze integer is de som van de waarden van de ingedrukte toetsen die als volgt toegekend worden: **UP = 1, DOWN = 2, RIGHT = 4, LEFT = 8, SPACE = 16 en EXIT = 32**.

Om de toetsen correct uit te lezen wordt de volgende techniek toegepast. Elke bewegingswaarde heeft een andere bit op 1 staan als we deze binair lezen: **UP = 000001, DOWN = 000010, RIGHT = 000100, LEFT = 001000, SPACE = 010000 en EXIT = 100000**. De integer die we terugkrijgen en die de som is van de toetswaardes, kan je dus ook binair voorstellen. We leggen dit uit met een voorbeeldje:

DOWN + RIGHT = 6 = 000110.

Aan de hand van een zogenaamde bitwise AND kunnen we nu gemakkelijk achterhalen welke toetsen ingedrukt zijn. Een bitwise AND gaat beide binaire waardes onder elkaar zetten en daar waar bij beiden een 1 staat, ook een 1 zetten, anders een 0.

000110 (down + right)

000010 (down)

----- &

000010 (down)

Als we met andere woorden een bitwise AND doen met de integer die de som voorstelt en een toets, en we komen die toets uit, dan is die ingedrukt. Dit kan je voor elke toets doen. Om dit toe te passen in je code, kan je gewoon met integers werken, het is niet nodig ze te casten naar een ander type. De bitwise AND gebruikt automatisch de binaire waardes van de integers. De bitwise AND is de `&`-operator. Let op: niet de `&&`-operator, dat is een logische AND.

3.3.2 Status aanpassen

De tweede en belangrijkste stap is het aanpassen van de status van het spel. De methode `update_game` roeft hiervoor een aantal methodes op die door jullie geïmplementeerd moeten worden:

- `void do_player_movement(Game* game);`

Voer eventuele bewegingen van de speler uit en hou hierbij rekening met obstakels.

- `void do_pikachu_ai(Game* game);`

Voer de bewegingen van de Pikachu's uit (opnieuw rekening houdend met obstakels) en controleer of deze in aanraking komen met de speler.

- `void process_bonus_items(Game* game);`

Controleer of de speler op een bonus staat en voer de corresponderende actie uit.

- `void process_pokeballs(Game* game);`

Pas de timers van de Pokeballs aan en voer de eventuele vangpogingen uit.

3.3.3 Scherm tekenen

De methode `render_game` is verantwoordelijk om de huidige toestand van het spel te tekenen aan de hand van de (grotendeels) meegegeven GUI-methodes.

3.4 Speelbaarheid

Een belangrijk aspect bij het implementeren van een spel, naast het pure functionele, is de speelbaarheid. Met andere woorden: is het spel leuk om te spelen, al zijn alle theoretische regeltjes goed geïmplementeerd?

Voorbeelden van zaken die de speelbaarheid kunnen verbeteren zijn:

- Een vlotte manier van besturen. Als een speler bv. naar boven loopt en ook de knop om naar rechts te gaan indrukt, zorg dan dat automatisch de volgende beschikbare bocht naar rechts wordt genomen (zie Figuur 6).
- In sommige gevallen is het beter dat de collision detection niet te strikt werkt. Bv. als een Pikachu slechts voor een fractie in een tegel met een vangpoging staat, markeren we deze Pikachu dan onmiddellijk als gevangen? Visueel kan het namelijk lijken alsof er niets aan de hand is.



Figuur 6: Automatisch naar rechts als right én up ingedrukt zijn.

3.5 Highscores bijhouden

Ten slotte is het ook de bedoeling dat er een array met highscores wordt bijgehouden in een bestand. Hiervoor dienen jullie de operaties zoals beschreven in `highscores.h` te implementeren:

- `void load_highscores(HighScoreTable* highscores);`

Leest het bestand waar de highscores in opgeslagen worden in. De array met highscores is beperkt in grootte (zie `MAX_HIGHSCORE_ENTRIES` macro constante), toch verwachten we dat de tabel dynamisch gealloceerd wordt en dat er niet meer geheugen wordt gebruikt dan nodig is!

- `void check_highscore_entry(HighScoreTable* highscores, int score);`

Controleert of de meegegeven score een nieuwe highscore is. Indien dit het geval is, dan wordt de speler gevraagd zijn/haar naam in te geven en wordt de highscore toegevoegd aan de tabel met highscores. Een score wordt gezien als een highscore als het groter is dan één van de bestaande highscores of als er nog plaats is in de tabel. Naast de score en de naam van de speler houden we ook de tijd/datum waarop de highscore werd bereikt bij.

- `void save_highscores(HighScoreTable* highscores);`

Schrijft de tabel weg naar het bestand waar de highscores in opgeslagen worden, enkel en alleen als deze aangepast is! Het is de bedoeling dat de highscore tabel als binaire data wordt opgeslagen in het bestand aangeduid door de

macro constante HIGHSCORE_FILE.

De plaats van oproep van deze functies dienen jullie te bepalen. Ook is het nodig om, zoals weergegeven in Figuur 2, de highscores naar het scherm weg te schrijven eens het spel beëindigd is vanwege een game over.

Naam	Waarde	Beschrijving
MAX_LEVEL_WIDTH	90	Maximale breedte (in tegels) van de wereld voor generatie van het level
MAX_LEVEL_HEIGHT	60	Maximale hoogte (in tegels) van de wereld voor generatie van het level
TILE_SIZE	64	De grootte van een tegel
PLAYER_MOVEMENT_INCREMENT	4	Grootte van de beweging die een speler kan maken in één tick
PIKACHU_MOVEMENT_INCREMENT	2	Grootte van de beweging die een Pikachu kan maken in één tick
STRONG_PIKACHU_MOVEMENT_INCREMENT	4	Grootte van de beweging die een extra sterke Pikachu kan maken in één tick
POKEBALL_TICKS	60	Aantal ticks dat een Pokeball blijft staan
CATCH_ATTEMPT_TICKS	30	Aantal ticks dat een vangpoging blijft duren
POKEBALL_MAX_POWER	5	Maximale kracht van een Pokeball
FREEZE_DURATION	100	Aantal ticks dat de Pikachu's bevoren blijven
PIKACHU_SCORE	60	Basisscore die een Pikachu oplevert
STRONG_PIKACHU_SCORE	150	Score die een extra sterke Pikachu oplevert
STRONG_PIKACHU_LIVES	3	Aantal vangpogingen dat een sterke Pikachu kan overleven

Tabel 1: Constanten in het spel.

4 BESTANDEN

Een deel van de code wordt bij de opgave meegegeven. Deze code is te vinden in de repository die voor elke groep reeds klaargezet is. Het is de bedoeling de bestanden waar nodig aan te passen om zo tot een werkende oplossing te komen. De onderstaande bestanden zijn geheel of gedeeltelijk gegeven.

4.1 catch_them_all.h

Dit headerbestand bevat alle definities die betrekking hebben op het spel zelf: constanten voor spelaspecten, enumeraties, structdefinities voor de speler, de Pikachu's en de levelentiteiten.

Tabel 1 geeft een overzicht van de meegegeven constanten vastgelegd via macro's. Merk op dat 'één tick' overeenkomt met een enkele iteratie van de game loop.

4.2 game.h

Het headerbestand game.h specificeert een spel als een struct en een aantal operaties die erop kunnen toegepast worden. Voorbeelden van operaties zijn: het initialiseren van het spel, het uitvoeren van de spelerbewegingen en het tekenen.

4.3 game.c

Implementeer de lege functies. Pas de meegegeven game loop aan zodat deze op een correcte manier beëindigd wordt, rekening houdend met de status van het spel, het sluiten van de GUI,...

4.4 gui.h/c

Specificatie en implementatie van de GUI. De GUI-implementatie verzorgt de communicatie met Allegro. De code is grotendeels meegegeven, enkele functies dienen (deels) ingevuld te worden. Een overzicht van de functies is weergegeven in appendix I.

4.5 util.h

Dit headerbestand bevat alle informatie omtrent de events die kunnen gegenereerd worden. Bekijk dit gerust eens, aanpassingen worden hier niet verwacht.

4.6 highscores.h

Het headerbestand highscores.h specificeert een aantal operaties die gebruikt worden om de highscores van het spel bij te houden.

4.7 highscores.c

Implementeer de lege functies.

4.8 level.h

Het headerbestand level.h specificeert een level als een struct en een aantal operaties die erop kunnen toegepast worden.

4.9 level.c

Implementeer de lege functies.

4.10 main.c

De main.c implementatie is reeds beschikbaar. Aanvullingen zijn toegestaan.

5 COMPILEREN EN UITVOEREN

Let op dat Windows en Mac OSX gebruikers de PGM virtuele machine³ dienen te gebruiken om het project te kunnen ontwikkelen. Voor alle duidelijkheid: WSL wordt niet gebruikt voor het project.

Enkel de Allegro libary moeten jullie nog toevoegen en dit kan op de volgende manier:

```
sudo add-apt-repository ppa:allegro/5.2  
sudo apt install liballegro5-dev
```

De volledige installatiehandleiding voor het project is ter referentie te vinden in de algemene documentatie repository van het vak Programmeren⁴.

³<https://github.ugent.be/Programmeren/docs/blob/master/setup-vm.md>

⁴<https://github.ugent.be/Programmeren/docs/blob/master/setup-project.md>

6 TIPS EN OPMERKINGEN

Hieronder worden een aantal tips en opmerkingen gegeven waar best rekening mee gehouden wordt in dit project:

- Weet dat de basisimplementatie zoals deze gevraagd wordt in deze opgave genoeg is om het maximum van de punten te halen. Uitbreidingen dienen dus slechts toegevoegd te worden eens alles werkt en enkel indien je dit zelf zou willen. Uitbreidingen zullen geen extra punten met zich meebrengen.
- Aangezien dit project met twee studenten gemaakt dient te worden, zal een vlotte samenwerking noodzakelijk zijn. Zoals eerder vermeld, zal er reeds een repository voor elke groep beschikbaar zijn. Het gebruik van een andere repository is niet toegestaan. **Het publiek plaatsen van de code zal gedetecteerd worden door onze interne monitoring tools en zal leiden tot puntenverlies.**
- Begin niet onmiddellijk te programmeren. Lees eerst aandachtig de opgave en denk na hoe je de verschillende problemen zou oplossen.
- Denk goed na over de structuur van je programma. Een lange functie kan vaak opgesplitst worden in een aantal kleinere functies die elk instaan voor een bepaald deelprobleem. Dit zorgt niet alleen voor meer overzicht, maar bevordert ook het hergebruik van code.
- Alle opmerkingen bij de practica gelden ook hier: wees zuinig met geheugen, let op voor dangling pointers, geef alle gealloceerde geheugen ook weer vrij, controleer of bestanden wel correct geopend en gesloten worden, etc.
- Probeer compiler warnings te vermijden; deze duiden meestal op fouten die gemakkelijk op te lossen zijn.
- Vanzelfsprekend zijn er zaken waar deze opgave je vrij in laat; deze zijn naar eigen inzicht in te vullen. Verduidelijk in elk geval je broncode met commentaar waar dit nuttig is en leg belangrijke ontwerpsbeslissingen uit in het verslag.
- Vragen over het project kunnen gesteld worden via (1) het mailadres pgm@lists.ugent.be en (2) Github issues. Wij trachten alle vragen zo snel mogelijk te beantwoorden. Indien er een probleem is met de samenwerking tussen de groepsleden, gelieve dit dan zo snel mogelijk door te geven, zodat er een oplossing uitgewerkt kan worden.

7 INDIENEN

Het project wordt gemaakt in groepen van twee studenten. Om in te dienen zorg je ervoor dat de volgende twee zaken uitgevoerd zijn ten laatste op zondag 15 mei 2022, 23u59:

1. De code staat klaar in de Github repository van jullie groep. De code van de master branch zal opgehaald worden.
2. Een verslag van het type PDF met een maximale lengte van twee pagina's is ingediend via de Ufora opdracht "Verslag Project". Het verslag omvat de volgende zaken:
 - De naam en richting van de groepsleden
 - Een korte uitleg over de belangrijkste ontwerpbeslissingen
 - Hoe worden de levels aangemaakt?
 - Hoe zit het collision model in elkaar? (voor de speler, Pikachu's, vangpogingen,...)
 - ...
 - Bij het niet (correct) uitvoeren van het programma: waarom?
 - De taakverdeling: wie heeft wat gedaan?

Als minimumvooraarde wordt vereist dat het project compileerbaar is, zoniet wordt een nulscore toegekend. Wanneer het programma niet correct uitgevoerd kan worden, wordt er gevraagd hierover een korte toelichting te geven in het verslag.

8 APPENDIX I: GUI

De code voor de GUI, zoals beschikbaar in gui.h en gui.c, bestaat uit de volgende functies:

- `void gui_initialize():`
Initialiseert de GUI.
- `void gui_clean():`
Ruimt de GUI op.
- `void gui_set_level_info(LevelInfo* level_info):`
Geeft de nodige parameters voor de visualisatie van het level door aan de GUI, zoals de breedte en hoogte in tiles.
- `void gui_game_event(Event* event):`
Vraagt het volgende event op uit de queue.
- `void gui_add_bonus(Bonus* bonus):`
Voegt een Bonus toe aan de grafische buffer.
- `void gui_add_pikachu(Pikachu* pikachu):`
Voegt een Pikachu toe aan de grafische buffer.
- `void gui_add_catch_attempt_tile(int x, int y):`
Voegt een onderdeel van een vangpoging toe aan de grafische buffer.
- `void gui_add_player(Player* player):`
Voegt een Player toe aan de grafische buffer.
- `void gui_add_obstacle(Obstacle* obstacle):`
Voegt een Obstacle toe aan de grafische buffer.
- `void gui_add_pokeball(Pokeball* pokeball):`
Voegt een Pokeball toe aan de grafische buffer.
- `void gui_draw_buffer():`
Tekent de inhoud van de grafische buffer op het scherm en leidigt de buffer.
- `void gui_start_timer():`
Start de timer component (zichtbaar bovenaan het scherm).
- `int gui_get_timer_score():`
Vraagt de huidige waarde van de timer op. Deze wordt gebruikt om de score te berekenen.
- `void gui_set_game_over():`
Stelt in dat game over is bereikt.
- `void gui_set_finished_level(int score):`
Stelt in dat een level is uitgespeeld met een finale score.
- `void gui_set_pikachus_left(int pikachus):`
Stelt het aantal overblijvende Pikachu's in.

- `void gui_set_pokeballs_left(int pokeballs):`
Stelt het aantal overblijvende Pokeballs in.
- `void gui_set_level_score(int level_score):`
Stelt de tussentijdse score van het level in.

9 APPENDIX II: MEMORY LEAKS DETECTEREN MET ADDRESS SANITIZER

Op Linux, Mac en in WSL is het heel eenvoudig om memory leaks en andere geheugenfouten zoals een "double free" op te sporen door middel van AddressSanitizer (ASan). Dit is een geheugen foutdetector origineel ontwikkeld door Google voor de Clang compiler die sinds versie 4.8 ook in GCC verwerkt zit.

ASan staat automatisch aan voor de labo's van PGM. Je kan die voor andere programma's zelf aanzetten door de flags "-fsanitize=address -fno-omit-frame-pointer" mee te geven met de compiler en de linker.

De compiler linkt alle geheugenoperaties zoals `malloc` en `free` door naar ASan, zodat deze alle geheugenoperaties kan bijhouden. Bij het afsluiten van het programma wordt er gekeken welk geheugen er aangevraagd is zonder het te verwijderen en rapporteert ASan alle memory leaks in het formaat dat hieronder te zien is.

```
=====
```

```
==22505==ERROR: LeakSanitizer: detected memory leaks
```

```
Direct leak of 20000 byte(s) in 1 object(s) allocated from:
#0 0x7f9ad892a848 in __interceptor_malloc (/usr/lib/x86_64-linux-gnu/
    libasan.so.5+0xee848)
#1 0x55b9aede4800 in graphics_init /sysprog/game/graphics/
    opengl_game_renderer.c:364
#2 0x55b9aede4605 in graphics_alloc /sysprog/game/graphics/
    opengl_game_renderer.c:343
#3 0x55b9aedb75ee in main /sysprog/puzzle_bots_part2_main.cpp:43
#4 0x7f9ad6f32b96 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so
    .6+0x21b96)
```

Deze stack trace toont aan welke code het geheugen aangemaakt heeft. Zoals je ziet worden alle geheugenoperaties uiteindelijk uitgevoerd door `libasan`, de geheugenchecker zelf. Je mag regel #0 dus overslaan. Het eigenlijke geheugenlek doet zich voor in de `graphics_init` functie in `opengl_game_renderer.c` op lijn 364. De rest van de stacktrace toont de volledige volgorde van oproepen: de `main` functie roept op lijn 43 de `graphics_alloc` functie op, die op lijn 343 de `graphics_init` functie oproept.

Bij het debuggen wordt een extern consolevenster gebruikt dat afsluit bij het stoppen van het programma. Hierdoor is het niet mogelijk om de memory leaks te zien na een debugsessie. Start een normale uitvoering met het programma door "Terminal > Run Build Task" uit te voeren (`ctrl-shift-b`) en "run (+ build)" uit te voeren. Dit zal het project eerst builden en dan uitvoeren. De output zie je in het terminalvenster van Visual Studio Code. Merk op dat de optie "start without debugging" de debugger toch opstart. Dit is een gekende fout in de C/C++ extensie van Visual Studio Code.