Software-ontwikkeling I

Academiejaar 2012 - 2013

SOI@intec.ugent.be

Project opgave
Tower Defense

Inhoudstafel

1 Introductie

Alvorens de volledige opgave van het project toe te lichten, wordt eerst een inleiding gegeven van een aantal typische *game programming* aspecten. Het kan nuttig zijn om hiervan op de hoogte te zijn en zo direct een goede achtergrondkennis te hebben voordat er aan de eigenlijke opgave begonnen wordt. Er wordt dan ook van uit gegaan dat deze Introductie volledig doorgenomen is alvorens aan het project te beginnen.

1.1 Doel project

Het uiteindelijke doel van dit project is het implementeren van Tower Defense. Dit spel zal aan bepaalde voorwaarden moeten voldoen. In de opgave worden deze voorwaarden en vereisten volledig uit de doeken gedaan.

Uiteraard moet dit spel volledig in C geprogrammeerd worden. Besteed genoeg tijd en aandacht aan dit project, want het gaat om 3 van de 20 punten voor het vak Software Ontwikkeling I die je kunt winnen, maar dus ook verliezen.

1.2 Game programming

Spellen programmeren is op een aantal vlakken anders dan een gewone applicatie programmeren.

1.2.1 De game loop

De belangrijkste component van eender welk spel – vanuit het standpunt van de programmeur – is de *game loop*. De *game loop* zorgt ervoor dat het spel vlot blijft lopen, onafhankelijk van het feit of de gebruiker al dan niet input genereert.

Meer traditionele software programma's reageren op gebruikers input en doen niets zonder deze input. Neem bijvoorbeeld een word processor. Deze formatteert woorden en tekst terwijl de gebruiker typt. Als de gebruiker niets typt, dan doet de word processor niets. Sommige functies mogen dan wel lang duren om uit te voeren, maar al deze functies zijn gestart door een gebruiker die het programma vroeg om iets te doen.

Spellen moeten daarentegen constant blijven werken, onafhankelijk van de al dan niet aanwezige input van een gebruiker. Een *game loop* in pseudocode zou er zo kunnen uitzien:

```
while( user doesn't exit )
  check for user input
  move & run AI[1]
  resolve collisions
  draw graphics & play sounds
end while
```

Hierbij wordt zo lang de gebruiker het programma niet stopt, achtereenvolgens gecontroleerd welke input er gegeven is, de beweging in het spel uitgevoerd, de artificiële intelligentie (AI) van de vijand en logica van het spel uitgevoerd, worden eventuele botsingen (=collisions, zie ook 0) opgelost, de beelden getekend en de geluiden afgespeeld.

De *game loop* kan verder verfijnd en aangepast worden terwijl de ontwikkeling van het spel voortgaat, maar de meeste spellen zijn op dit basisprincipe gebaseerd.

Game loops kunnen verder nog verschillen afhankelijk van het platform waarop ze ontwikkeld zijn. Als voorbeeld, een spelletje voor DOS en consoles kon alle processing resources voor zijn eigen rekening nemen en gebruiken naar eigen wens. Een spel voor een Microsoft Windows besturingssysteem, moet uitgevoerd worden binnen de beperkingen van de *process scheduler*.

Verder is het zo dat sommige spellen in meerdere threads lopen, zodat bijvoorbeeld de berekening van de AI losgekoppeld kan worden van de generatie van bewegende beelden in het spel. Dit creëert uiteraard een kleine overhead, maar kan het spel vlotter doen lopen. Het zal er zeker voor zorgen dat het meer efficiënt kan uitgevoerd worden op hyper-threaded of multicore processoren. Nu de computerindustrie focust op CPU's met meerdere cores die meerdere threads parallel kunnen uitvoeren, wordt dit steeds belangrijker.

1.2.2 Bewegende beelden

Een ander belangrijk aspect bij games is het visuele. 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 te zien, is veel ingewikkelder dan meestal wordt aangenomen. Er zijn een paar vuistregels die in de meeste gevallen (blijken te) kloppen:

- Hoe meer *fps* (*frames per second*) hoe vloeiender het beeld.
- Hoe kleiner het verschil tussen (de beeldinformatie op) de verschillende frames hoe vloeiender de beweging.

Dit is niet in alle gevallen correct, maar voor ons volstaat het om hier zo over te denken.

[1] Artificiële Intelligentie

Het is gemakkelijk om de snelheid van de game loop (logische lus) ook in frames per second uit te drukken, op die manier is er een maat om mee te vergelijken. *Frames per second* is dan eigenlijk hoeveel keer per seconde de *gamestate* aangepast wordt.

Het aantal *fps* is dus bij games enerzijds gelimiteerd door de hardware van de computer, maar ook door hoe zwaar de *game loop* wordt. Bij een actie ondernomen door de speler die meer berekeningen vraagt bijvoorbeeld, zal de loop-iteratie langer duren en dus het aantal zichtbare *fps* tijdelijk dalen.

1.2.3 Collision detection

Bij games wordt dikwijls over *collision detection* gesproken. Dit is het detecteren wanneer twee of meerdere objecten tegen elkaar aan (gaan) botsen. In elke stap in de *game loop*, direct nadat een beweging is uitgevoerd door objecten, moet er vóór het hertekenen van het frame eerst gecontroleerd worden of er geen collision (=botsing) is. zodat er eventueel gepast op kan gereageerd worden. Dit kan bijvoorbeeld het tegen elkaar plaatsen zijn van de twee objecten in plaats van 'in' elkaar of een botsing zijn die de objecten beiden de tegengestelde kant opstuurt. Eens deze *collision* gedetecteerd en opgelost is, kan het frame hertekend worden.

In een 2-dimensionale ruimte valt detectie van bostingen goed mee, maar in 3-dimensionale ruimtes kan *collision detection* snel geavanceerd worden. Er zijn veel artikels en boeken geschreven rond dit onderwerp en elk van deze artikels en boeken vereist een goede wiskundige kennis.

In deze opgave zal de *collision detection* zich beperken tot de 2-dimensionale variant.

2 Opgave

2.1 Omschrijving

Het doel van het project is om een variant op het spel Tower Defense[2] te programmeren. In Tower Defense is het de bedoeling dat de speler zijn/haar eigen kasteel zo lang mogelijk beschermt tegen de aankomende vijanden. Hiervoor heeft hij/zij een bepaalde hoeveelheid geld tot zijn/haar beschikking. Met het geld kunnen torens op het speelveld geplaatst worden, die de aankomende vijanden aanvallen. Per verslagen vijand, verdient de speler extra geld. Vijanden kunnen niet zomaar door torens lopen, en de speler kan dit dus in zijn/haar voordeel gebruiken om hun pad naar het kasteel langer te maken. Per vijand die het kasteel bereikt, gaan er levenspunten af van het kasteel. Komt de speler op 0 levenspunten te staan, dan verliest hij/zij het spel.

In de volgende puntjes overlopen we de verschillende onderdelen van het spel en leggen we de exacte spelregels vast.

2.2 Spelregels

2.2.1 Waves

Vijanden komen in het spel op het *spawn* punt. Dit gebeurt in zogenaamde *waves*. Elke wave spawnen er dus een aantal vijanden. Voor het beginnen van een nieuwe wave, wordt er telkens een kleine pauze ingelast. De bedoeling is dat elke wave moeilijker is dan de vorige, op deze manier wordt het een uitdaging voor de speler om zo ver mogelijk te geraken in het spel.

2.2.2 Obstakels

De spelwereld bevat een aantal obstakels: water en bergen. Niemand kan over deze obstakels heen, behalve vliegende vijanden, deze kunnen over water vliegen.

2.2.3 Vijanden

Er zijn verschillende soorten vijanden. Alle vijanden proberen van het moment dat ze spawnen, op een relatief efficiënte manier naar het *castle* te geraken. Eens ze daar aankomen, worden ze van het speelveld gehaald en heeft het castle schade op gelopen.

De verschillende soorten vijanden zijn de volgende:

NORMAL	Dit zijn normale soldaten.
ELITE	Dit zijn tanks. Voor dit type gelden dezelfde regels als voor NORMAL, maar ze hebben behoorlijk wat meer levenspunten.
FAST	Dit zijn snelle ninja's. Ze kunnen raketten ontwijken, hebben minder levenspunten dan een NORMAL type, maar lopen dubbel zo snel.

[2] http://en.wikipedia.org/wiki/Tower_defense

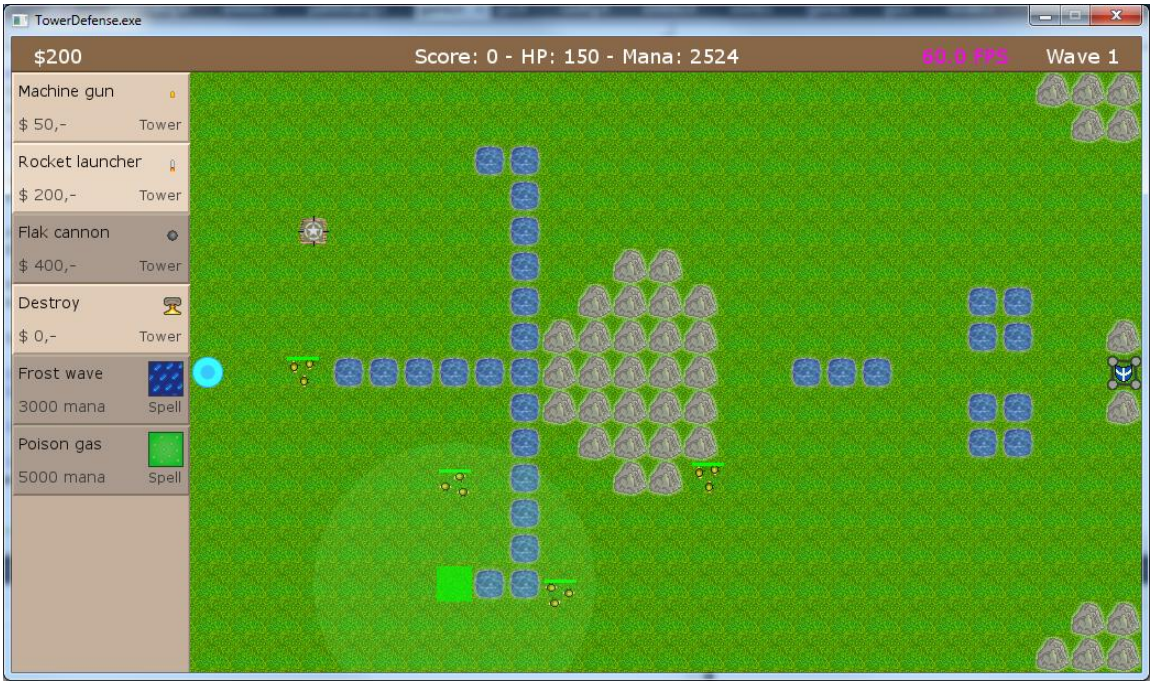
AIR	Deze vliegtuigen kunnen over water vliegen, maar niet over bergen. Ze kunnen enkel geraakt worden door flak (luchtafweergeschut).
BOSS	Dit is een grote robot, vergelijkbaar met een tank, maar met veel meer levenspunten.

2.2.4 Torens

Er zijn verschillende type torens die door de speler kunnen gebouwd worden. De verschillende types zijn:

MACHINE GUN	Een simpele toren, kost niet veel geld, en schiet met een machine geweer binnen een beperkt bereik.
ROCKET LAUNCHER	Een raket afvurende toren, kost redelijk wat geld, maar schiet raketten af die redelijk wat schade aanrichten. Hun bereik is redelijk groot, maar de raketten vliegen relatief traag.
FLAK CANNON	Dit luchtafweergeschut heeft een groot bereik, maar kost ook veel geld. Het kan enkel op vliegende vijanden schieten.
DESTROY TOWER	Dit is geen toren, maar onder de toren knoppen wordt een knop voorzien om bestaande torens weer af te breken.

De verschillende torens kunnen door de speler op het speelveld gezet worden, hierbij is een *blueprint* te zien van de toren met zijn bereik (Figuur 1). Torens plaatsen kan enkel volgens een vast raster (zie ook 3.1.1). Torens kunnen ook enkel geplaatst worden op lege vakjes (met uitzondering van een voorbijlopende vijand). Wanneer het plaatsen van een toren ervoor zou zorgen dat er geen geldig pad meer is van *spawn* naar *castle*. Dan moet het plaatsen van die toren mislukken.



Figuur 1: Blueprint van een toren

2.2.5 Spells

Naast de torens, heeft de speler ook nog twee *spells* in zijn/haar arsenal. Gedurende het spel wordt de *mana* (= magische grondstof) van de speler automatisch verhoogd tot een bepaald maximum. De speler kan op elk moment in het spel beslissen om een van volgende spells te gebruiken (indien genoeg mana):

FROST WAVE	Een blauwe waas verschijnt over het scherm, alle vijanden staan bevroren op het speelscherm, maar torens kunnen nog schieten. Ook spawnen er tijdelijk geen nieuwe vijanden. Dit effect blijft geldig voor 10 seconden.
POISON GAS	Een groene waas verschijnt over het scherm, alle vijanden krijgen een toxisch gas over zich heen. Hierdoor worden gedurende vijf seconden hun levenspunten stapsgewijs gereduceerd tot een fractie van hun totale levenspunten.

2.2.6 Hud

De heads-up-display geeft op elke moment de volgende informatie aan de speler:

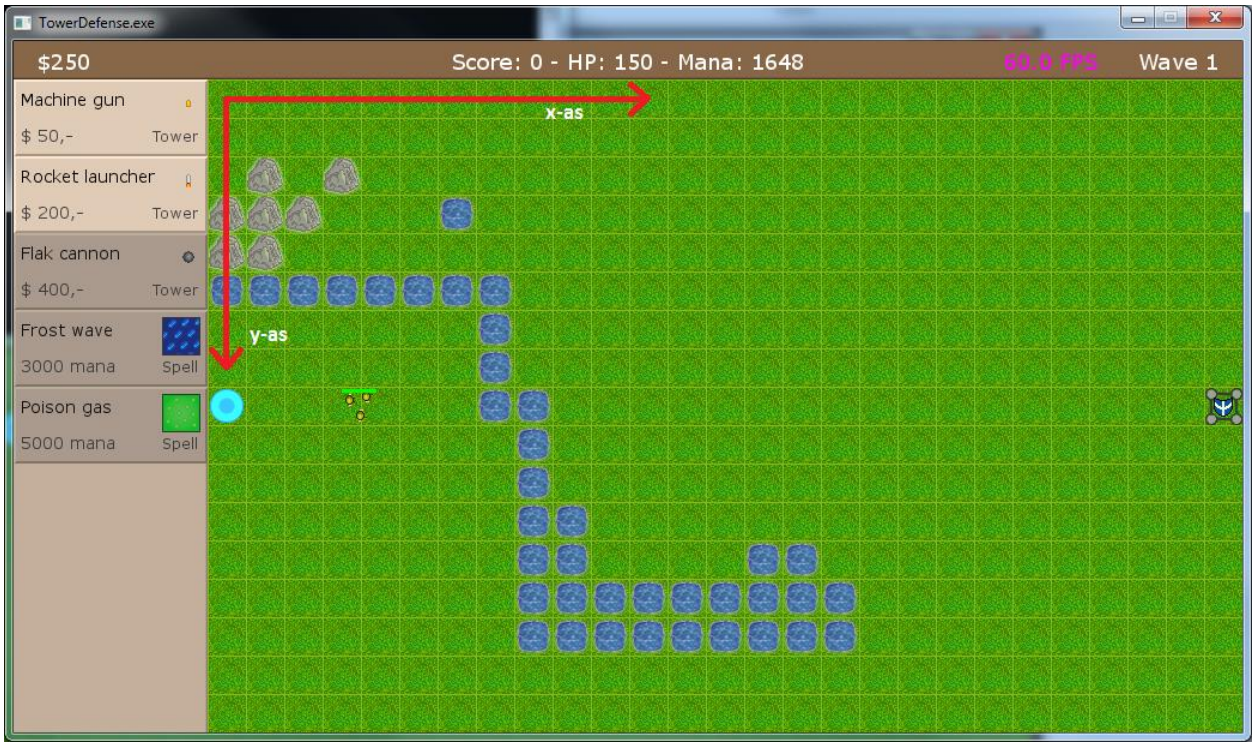
- Huidige hoeveelheid geld
- Huidige score
- Huidige levenspunten van het castle
- Huidig aantal mana
- Huidige wave

3 Implementatie

3.1 Assenstelsel, coördinaten en tiles

3.1.1 Assenstelsel en tiles

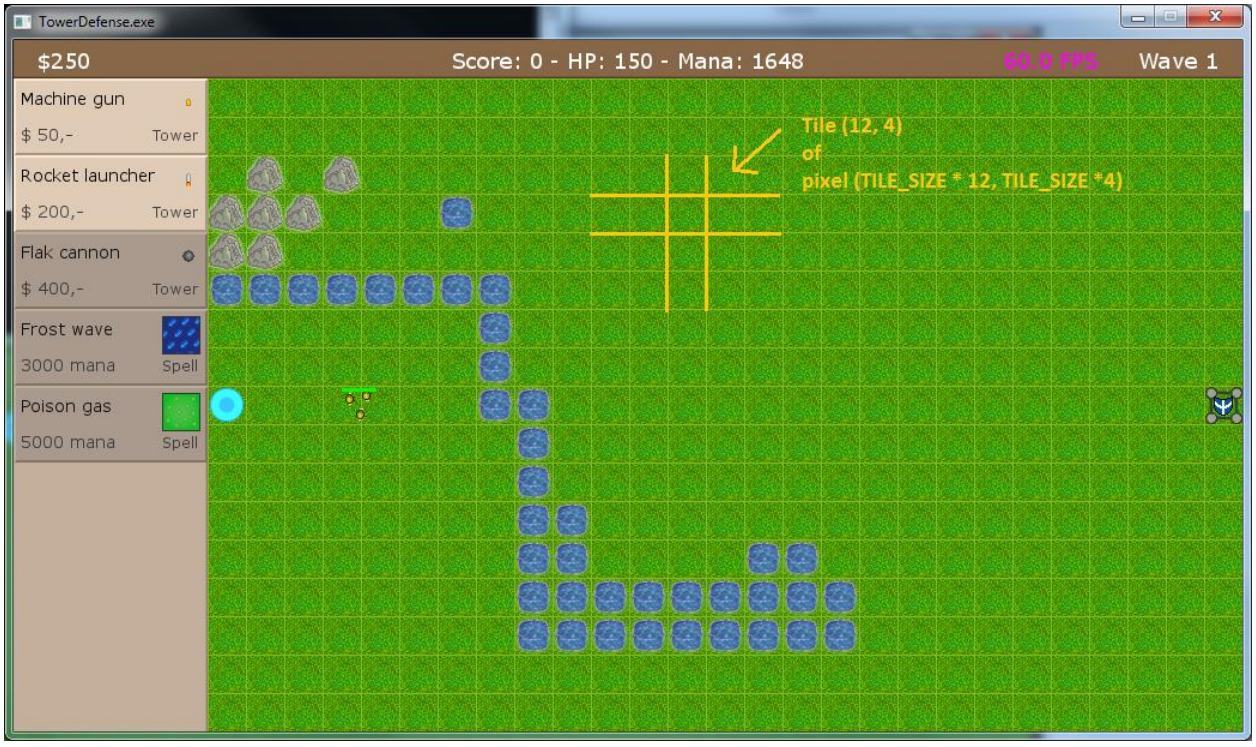
Daar de spelwereld van bovenaan bekeken wordt, kunnen we deze in twee dimensies voorstellen waarbij vijanden horizontaal bewegen volgens de X-as en verticaal volgens de Y-as.



Figuur 2: X en Y as in het spel

De locatie van elk object in de wereld kan dus voorgesteld worden aan de hand van een X,Y coördinaat. Het gebruikte coördinatensysteem is equivalent aan dat van Java Swing, waarbij de X-coördinaat toeneemt naar rechts en de Y-coördinaat toeneemt naar beneden (zie Figuur 2). Voor de eenvoudigheid delen we de wereld verder in tegels (*tiles*) in. Een tile heeft een vaste breedte en hoogte, zoals gedefinieerd in de macro constante TILE_SIZE, en kan een object bevatten.

Elk object heeft dus een grootte gelijk aan de TILE_SIZE en zal dus een X en Y coördinaat hebben die een veelvoud is van deze TILE_SIZE. **Uitzonderingen** hierop zijn de vijanden en projectielen (zie 3.2), omdat deze zich stapsgewijs over deze tegels kunnen verplaatsen om zo een vlotte beweging te garanderen.



Figuur 3: Tiles op een speelveld

3.1.2 Coördinaten

In het spel werken we met een aantal coördinaten stelsels. Om het duidelijk te houden wordt er een conventie afgesproken.

- Coördinaten waarvan de oorsprong in de linkerbovenhoek van het scherm ligt, en die dus voornamelijk gebruikt worden om te renderen naar het scherm zijn: *screen_x* en *screen_y*.
- Coördinaten waarvan de oorsprong in de linkerbovenhoek van het World scherm (groene achtergrond) ligt, zijn: *world_x* en *world_y*. Deze worden voornamelijk gebruikt met betrekking tot pixel-precieze beweging op het speelveld.
- Coördinaten die in tiles rekenen en waarvan de oorsprong-tile in de linker bovenhoek ligt op het World scherm, zijn: *tile_x* en *tile_y*.

Wanneer een bepaalde functie een coördinaat als argument neemt, zal deze variabele volgens de conventie aangeduid worden, zodoende duidelijk te maken met welk soort coördinaat de functie in kwestie moet aangeroepen worden.

3.2 Spelobjecten

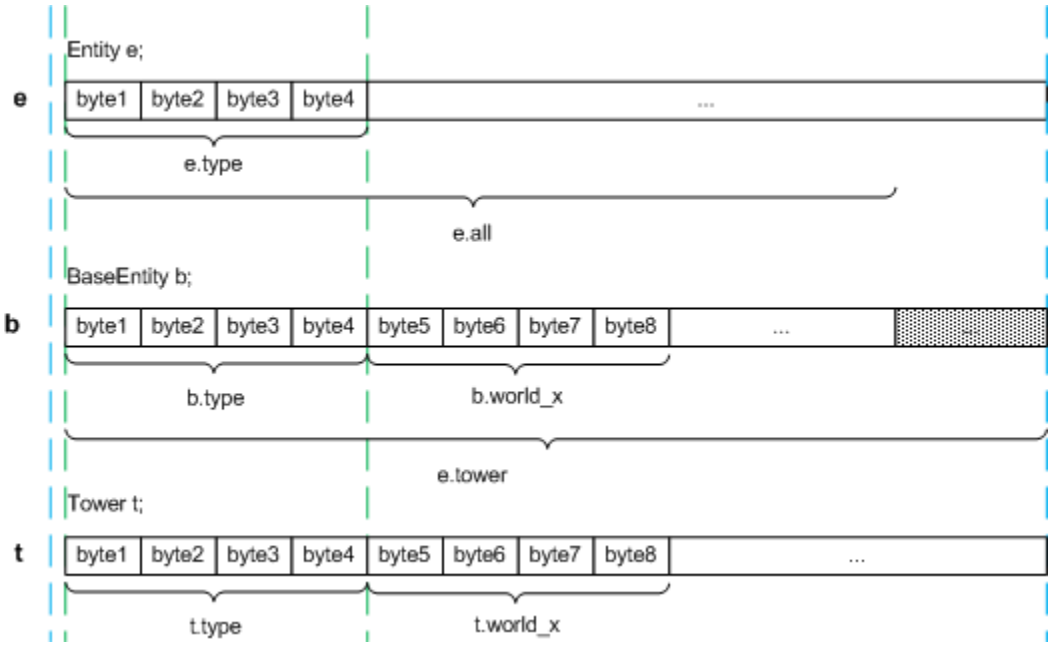
Daar een spel-object verschillende zaken kan voorstellen (vijand, toren, obstakel …), maar wel op een eenduidige manier moet kunnen gebruikt worden voor onder andere collision-detectie en plaatsing op de wereld, hebben we besloten dit te modeleren aan de hand van een union type. Union types worden vaak gebruikt in C als een truc om overerving te simuleren. De union Entity ziet er als volgt uit:

```
typedef union {
	EntityType type;
	BaseEntity all;
	Projectile projectile;
	Enemy enemy;
	SpawnLocation spawn_location;
	Castle castle;
	Tower tower;
	Obstacle obstacle;
} Entity;
```

Het veld type duidt aan welk type spel-object een instantie van Entity is en op basis hiervan weten we dan welk van de overige velden momenteel in gebruik is. Het veld *all* stelt de gemeenschappelijke velden voor aan de hand van de struct *BaseEntity* (zie verder).

Strikt genomen maakt ook *type* deel uit 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. In dit geval is dit het veld enemy[3].

Als we dus voor elk van de struct velden (all, projectile, enemy, …) plaats voorzien voor een EntityType veld *type* aan het begin van de struct definitie (BaseEntity, Projectile, Enemy, …), 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 geheugenlokatie van het veld type in één van de onderliggende structs te bevragen. Dit principe heet *alignatie* en wordt visueel voorgesteld in Figuur 4.



Figuur 4: Alignatie van het type veld

[3] Je kan dit controleren aan de hand van de sizeof() functie.

De gemeenschappelijke velden voor een entity worden dus gebundeld in de struct BaseEntity, die er als volgt uitziet:

- `EntityType type:` het type van deze Entity
- `float world_x:` de X positie van deze Entity op de wereld
- `float world_y:` de Y positie van deze Entity op de wereld
- `int width:` de breedte van deze Entity
- `int height:` de hoogte van deze Entity
- `FrameAnimator frameAnimator:` de frameAnimator voor deze Entity (zie 3.3)

In wat volgt overlopen we voor elk van de entities de additionele velden.

struct Enemy:

- `EnemyType enemy_type:` het type van deze vijand
- `float angle:` de hoek waarnaar de vijand kijkt (in radialen, wijzerzin vanaf noorden)
- `int direction_x:` de X richting van deze vijand {-1,0,1}
- `int direction_y:` de Y richting van deze vijand {-1,0,1}
- `float speed:` de snelheid (in pixels per frame) van deze vijand
- `int damage:` de schade die deze vijand toebrengt aan het kasteel
- `int health:` het huidige aantal levens van deze vijand
- `int health_max:` het maximum aantal levens van deze vijand
- `int alive:` 1 als de vijand in leven is, anders 0
- `Path path:` het pad dat deze vijand aflegt

struct Projectile:

- `ProjectileType projectile_type:` het type van dit projectiel
- `Enemy * target:` het doelwit van dit projectiel
- `float angle:` de hoek waarnaar dit projectiel wijst (in radialen, wijzerzin vanaf noorden)
- `int direction_x:` de X richting van dit projectiel {-1,0,1}
- `int direction_y:` de Y richting van dit projectiel {-1,0,1}
- `int damage:` de schade dat dit projectiel toebrengt aan een vijand
- `int live:` 1 als dit projectiel actief is, anders 0
- `float speed:` de snelheid (in pixels per frame) van dit projectiel

struct SpawnLocation

(geen additionele velden)

struct Castle:

`int health:` het huidig aantal levens van het kasteel

struct Tower:

- `TowerType tower_type: het type van deze toren`

- `float range:` de straal van de cirkel die het bereik van deze toren voorstelt. Het middelpunt van deze cirkel ligt in het midden van de toren (en niet de linkerbovenhoek)
- `int shoot_interval:` de wachttijd tussen schoten van deze toren (in aantal frames)
- `int frames_since_last_shot:` aantal frames sinds het laatste schot
- `int ammo:` het maximaal aantal kogels dat tegelijkertijd actief kan zijn voor deze toren
- `int cost:` de kost voor het bouwen van deze toren
- `Enemy * target:` het huidige doelwit van deze toren
- `Projectile* projectiles:` de rij van projectiel van deze toren

struct Obstacle:

`ObstacleType obstacle_type:` het obstakel type (water of berg)

3.3 FrameAnimator

Een Entity die geanimeerde sprites heeft, heeft een FrameAnimator veld. Deze frame animator regelt de animatie volledig voor jullie. Hij werkt door een teller bij te houden die geïncrementeerd wordt bij elke aanroep. Is de teller hoger dan een bepaalde waarde, dan verspringt de index in de sprite sheet, zodat een ander frame getekend wordt. Een FrameAnimator kan geïnitialiseerd worden met de reeds geïmplementeerde functie:

```
void init_frameAnimator(FrameAnimator * animator, int animationColumns,
int maxFrame, int frameDelay, int frameHeight, int frameWidth);
```

Deze heeft een aantal parameters:

- animator: de FrameAnimator zelf die als pointer doorgegeven wordt, zodat de interne velden van de struct kunnen aangepast worden.
- animationColumns: het aantal kolommen dat er in de spritesheet staan.
- maxFrame: het aantal frames waaruit de animatie bestaat.
- frameDelay: hoe veel iteraties er moet gewacht worden om naar het volgende frame te springen.
- frameHeight: de hoogte van één frame.
- frameWidth: de breedte van één frame.

De correcte oproep voor de verschillende types Entities vinden we terug in Tabel 1.

Entity	animationColumns	maxFrame	frameDelay	frameHeight	frameWidth
Enemy: NORMAL	2	4	5	TILE_SIZE	TILE_SIZE
Enemy: ELITE	3	4	5	TILE_SIZE	TILE_SIZE
Enemy: BOSS	3	4	5	TILE_SIZE	TILE_SIZE
Enemy: AIR	2	4	5	TILE_SIZE	TILE_SIZE

Enemy: FAST	2	4	10	TILE_SIZE	TILE_SIZE
SPAWN_LOCATION	3	9	5	TILE_SIZE	TILE_SIZE
CASTLE	3	36	4	TILE_SIZE	TILE_SIZE

Tabel 1: FrameAnimator parameter waardes per type Entity

3.4 Pathfinding

Pathfinding (het vinden van een geldig pad in de spelwereld tussen twee punten) wordt gespecificeerd in de header-file pathfinding.h. Deze bevat de volgende functies:

- `int create_path(Path * path, World * world, Entity * from, Entity * to, TrajectoryType trajectory_type);`
  Maakt een pad aan tussen de gegeven entiteiten volgens het gegeven traject type. Geeft 1 terug als een geldig pad gevonden werd, anders 0.

- `int refresh_path(Path * path, World * world);`
  Herberekent het pad op basis van de huidige locatie in het pad. Moet opgeroepen worden voor bestaande paden als de wereld gewijzigd is. Geeft 1 terug als een geldig pad gevonden werd, anders 0.

- `void destroy_path(Path * path);`
  Geeft de resources die gebruikt worden door het pad vrij.

In wat volgt wordt eerst uitleg gegeven over de gebruikte datastructuren alvorens in te gaan op de werking van het algoritme zelf.

3.4.1 Hash set

Een hash set wordt gebruikt om een ongesorteerde verzameling van elementen op een efficiënte manier op te slaan. De hash set die gebruikt wordt voor dit project is gespecificeerd in de header-file hash_set.h en ondersteunt de volgende functies:

- `void hs_init(HashSet * hs);`
  Initialiseert de hash set.

- `void hs_add(HashSet * hs, Node * node);`
  Voegt een nieuw element aan de hash set toe.

- `int hs_contains(HashSet *hs, Node * node);`
  Geeft 1 terug als de hash set het gegeven element bevat, anders 0.

- `void hs_remove(HashSet * hs, Node * node);`
  Verwijdert het gegeven element uit de hash set.

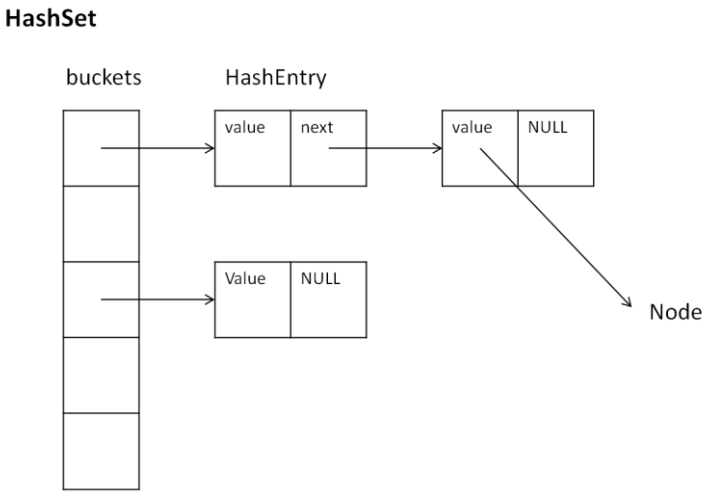
- `Node * hs_get_node(HashSet * hs, int tile_x, int tile_y);`
  Geeft de Node in de hash set terug voor de gegeven x en y coördinaten of NULL als de hash set geen dergelijke Node bevat.

- `void hs_rehash(HashSet *hs);`
  Herberekent de verdeling van de hash set.

- `unsigned int hs_calc_hash(Node * node);`
  Geeft de hashcode terug voor het gegeven element.

- `void hs_destroy(HashSet * hs);`
  Geeft de resources die gebruikt worden door de hash set vrij.

Voor de implementatie maken we gebruik van een hash tabel met afzonderlijk geschakelde lijsten (zie cursus p. 253). Op basis van de berekende hash-waarde wordt een element toegekend aan een zogenaamde bucket (een geschakelde lijst van element). Het aantal buckets hangt af van de bezettingsgraad van de set. Bij botsingen wordt het nieuwe element vooraan in de bucket toegevoegd. Figuur 5 geeft een schematisch overzicht van deze benadering.



Figuur 5: Schematische voorstelling HashSet

De prestaties van een hash set hangen sterk af van de gebruikte hash-functie. Een goede hash-functie zorgt voor zo weinig mogelijk overlappende hash-waarde. Daar we deze hash set specifiek gebruiken voor path-finding hebben we besloten de hash-functie te laten afhangen van de x en y waarden van de Nodes die toegevoegd worden:

```
h(x,y) = (x << 16) | (y & 0xFFFF)
```

Deze functie stelt een integer-waarde samen aan de hand van de 4 minst beduidende bytes van x en y.

Belangrijk: om te beslissen wanneer een rehash van de set moet uitgevoerd worden, hebben we een reeds geïmplementeerde hulp-functie rehash_nr_of_buckets opgegeven. Deze geeft een nieuwe waarde voor het aantal buckets terug, of 0 als er geen rehash moet uitgevoerd worden.

Alle benodigde structs (HashSet en HashEntry) worden opgegeven in de header-file hash_set.h en mogen niet aangepast worden!

3.4.2 Priority queue

Een queue is een FIFO (First-in, First out) datastructuur die minstens volgende operaties ondersteunt:

- offer: voegt een element toe (aan het einde van de queue)
- peek: geeft het eerste element in de queue terug.
- poll: geeft het eerste element in de queue terug en verwijdert deze uit de queue.

Een priority queue is een uitbreiding hierop waarbij de elementen gesorteerd worden volgens prioriteit. Hierbij zal poll/peek steeds het meest prioritaire element teruggeven.

De priority queue die gebruikt wordt in dit project is gespecificeerd in de header-file priority_queue.h, dat de volgende functies bevat:

- `void pq_init(PriorityQueue * pq);`
  Initialiseert de priority queue.

- `void pq_offer(PriorityQueue * pq, Node * node);`
  Voegt een element toe aan de queue op basis van zijn prioriteit.

- `int pq_is_empty(PriorityQueue * pq);`
  Geeft 1 terug als de queue leeg is, anders 0.

- `Node * pq_poll(PriorityQueue * pq);`
  Geeft het element met de hoogste prioriteit[4] terug en verwijdert deze.

- `Node * pq_peek(PriorityQueue * pq);`
  Geeft het element met de hoogste prioriteit terug.

- `int pq_contains(PriorityQueue * pq, Node * node);`
  Geeft 1 terug als de queue het gegeven element bevat, anders 0.

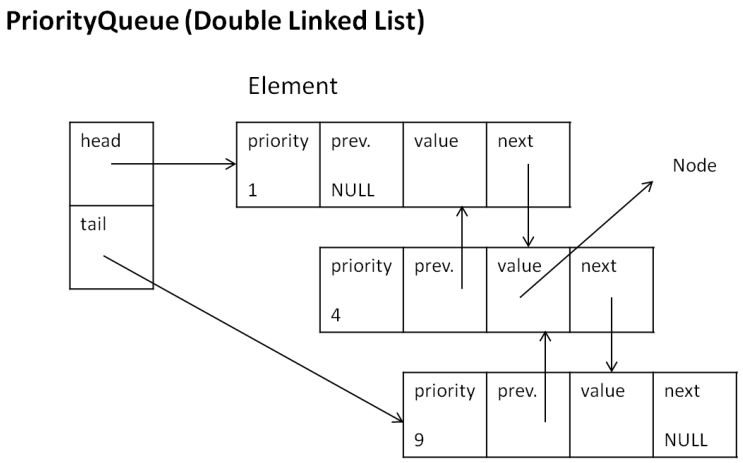
- `void pq_remove(PriorityQueue * pq, Node * node);`
  Verwijdert het gegeven element uit de queue.

- `void pq_destroy(PriorityQueue * pq);`
  Geeft de resources die gebruikt worden door de queue vrij.

Voor de implementatie maken we gebruik van een dubbel geschakelde lijst. Dit is een gesorteerde[5] opeenvolging van elementen, waarbij elk element een pointer bijhoudt naar zowel de voorganger als de

[4] Let op, wanneer we hier over een hogere prioriteit spreken, betekent dit een lager getal.

opvolger in de lijst. Het begin en einde van de lijst worden afgebakend aan de hand van een NULL-pointer. Verder worden pointers bijgehouden naar de head (eerste element) en de tail (laatste element) van de lijst. De priority queue implementatie aan de hand van een dubbel geschakelde lijst wordt visueel voorgesteld in Figuur 6.



Figuur 6: Schematische voorstelling PriorityQueue

De structs PriorityQueue en Element die nodig zijn voor de implementatie zijn vooraf gedefinieerd in de header-file priority_queue.h en mogen niet aangepast worden!

3.4.3 A*-algoritme

Het A* algoritme wordt gebruikt voor het zoeken van het kortste pad tussen twee knopen in een graaf. Het maakt hiervoor gebruik van een heuristische functie die elke knoop rangschikt volgens een schatting van de kost van het best mogelijk pad naar de bestemming vanuit deze knoop. De knopen worden dan bezocht door het algoritme in volgorde van de score die toegekend werd door de heuristische functie.

We vereenvoudigen het zoekgebied van het algoritme door enkel de tegels van de spelwereld in beschouwing te nemen en niet de afzonderlijke coördinaten, m.a.w. de tegels vormen de knopen van de graaf. Een knoop wordt voorgesteld aan de hand van de struct Node (entities.h), deze bevat de volgende velden:

- `int tile_x:` de x-tegel index van de knoop
- `int tile_y:` de y-tegel index van de knoop
- `NodeScore score:` een struct die de score-componenten van de knoop voorstelt
- `struct _Node * parent:` een pointer naar ouder van deze knoop (zie verder)

De werking van het algoritme wordt hieronder in pseudo-code beschreven en in bijlage (in de opgave zip file op minerva) vind je ook een volledig uitgewerkt voorbeeld, in de vorm van een powerpoint presentatie. Dit kan nuttig zijn bij de implementatie.

[5] Volgens de prioriteit van het element

```
open = priority queue containing start
closed = empty set

while (next in open != goal && open is not empty) {

      current = next in open (top priority)
      add current to closed

      /* Check accessible neighbours for the current node */

      foreach (valid neighbour n) {

            /* Skip if n has been closed */

            if (n not in closed) {

                  /* Calculate score for move to this neighbour */
                  g = current.score.g + move cost
                  h = (|n.x - goal.x| + |n.y - goal.y|) * move cost
                  f = g + h

                  if (n is in open) {

                        if (g < n.score.g) {

                              /*  A better route to n is found */
                              /* Update properties of n */

                              n.parent = current
                              n.score.f = f;
                              n.score.g = g;
                              n.score.h = h;

                              update n in open (resort)

                        }
                  } else {

                        /* Init properties of n */

                        n.parent = current
                        n.score.f = f;
                        n.score.g = g;
                        n.score.h = h;

                        add n to open

                  }
            }
      }
}

path = reverse path from goal to start (using parent pointers)
```

Eerst wordt een priority queue “open” en een hash set “closed” geïnitialiseerd. Aan open wordt de startknoop toegevoegd (let er op dat de score-velden geïnitialiseerd zijn). Daarna wordt er geïtereerd totdat ofwel het doel gevonden is, of er geen knopen meer kunnen bezocht worden (i.e. open bevat geen knopen meer). In het laatste geval kan er geen geldig pad gevonden worden naar het doel en zal het teruggeven pad enkel de doelknoop bevatten en dus een pad-lengte van 1 hebben.

Elke iteratie van de hoofdlus gaat als volgt: de pointer die de huidige knoop aanwijst wordt verzet naar de knoop in open met de meeste prioriteit. Deze wordt daarna onmiddellijk aan closed toegevoegd (waarmee we indiceren dat deze knoop vanaf nu niet meer opnieuw hoeft bezocht te worden). Hierna volgt er opnieuw een lus die zal itereren over elke aangrenzende (aan de huidge knoop), bewandelbare knoop. Dergelijke knoop noemen we een buur van de huidige knoop.

Voor elke buur wordt dan eerst nagegaan of deze al in closed zit. Als dit het geval is, dan kunnen we deze buur overslaan, daar hij al bezocht geweest is door het algoritme. Anders wordt de score berekend voor deze buur. De score bestaat uit drie componenten:

- **g** = de kost om van de startknoop naar deze buur te gaan volgens het beschouwde pad.
- **h** = een schatting van de kost om van deze buur naar de doelknoop te gaan.
  (hier op basis van de Manhattan afstandsfunctie)
- **f** = de som van de g en h componenten, bepaalt de prioriteit van deze knoop.

De “move cost” stellen we standaard gelijk aan 10, maar dit zou echter ook kunnen dynamisch bepaald worden. Op die manier zou het algoritme bijvoorbeeld kunnen aangepast worden om torens te proberen vermijden.

Daarna kijken we of deze buur al in open zit. Als dit het geval is en de nieuw berekende g score is kleiner dan de huidige waarde voor de buur, dan impliceert dit dat er een beter pad gevonden is naar deze buur (en dus potentieel ook een beter pad naar het doel). Bijgevolg passen we de scores aan en laten we de parent pointer verwijzen naar de huidige knoop. Ten slotte zorgen we ervoor dat deze aanpassing ook gereflecteerd wordt in de priority queue open.

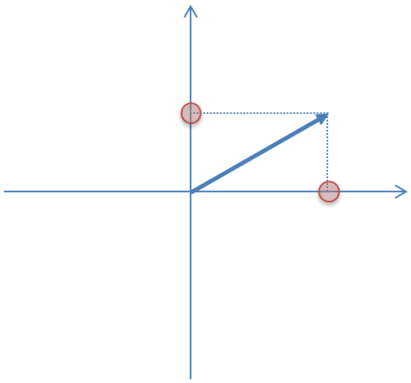
Als de buur nog niet in open zat, dan voegen we deze toe nadat de score waarden en de parent pointer op de correcte waarden gezet zijn.

Eens het doel bereikt is, kan het pad geconstrueerd worden door de parent pointers vanuit de doelknoop te volgen tot aan de startknoop.

3.5 Targetting

Een toren schiet op vijanden. Om het zo eenvoudig mogelijk te houden, zal elke lus van de iteratie gekeken worden onder welke hoek het projectiel moet toenemen met hoeveel x en hoeveel y pixels.

We weten dat vijanden enkel langs de x- of y-as bewegen. Hun snelheid is dus altijd constant. Wanneer echter projectielen beschouwd worden, valt op te merken dat deze niet evenwijdig met de assen bewegen. Hun snelheid zal dus ontleed moeten worden in een speed_x en speed_y.

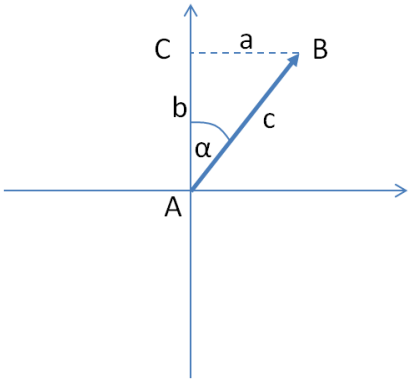


Figuur 7: Snelheid ontleden in x en y component

Om dit te berekenen gebruiken we goniometrie. Er moeten een aantal dingen gezocht worden.

1. De lengte van de zijden.
2. De grootte van de hoek α (in Figuur 8).
3. De snelheid waarmee de x en y coördinaten moeten aangepast worden.
4. De render hoek in functie van alpha

Eens deze gevonden zijn, hebben we alles om de projectielen correct af te beelden op het scherm.



Figuur 8: Uitgangspunt voor het vinden van alpha

3.5.1 Lengte van de zijden

Er vanuit gaand dat het projectiel (A) op de oorsprong ligt en het punt B het doelwit is, wordt eerst het punt C gezocht. (Figuur 8)

Als A coördinaat (x_A, y_A) heeft en B coördinaat (x_B, y_B), dan is C dus logischer wijs (x_A, y_B).

Vervolgens worden de lengtes van de zijden a, b en c berekend. Dit wordt gedaan door middel van de formule voor euclidische afstand:

$$\boldsymbol{d(A,B) = \sqrt{(x_A - x_B)^2 + (y_A - y_B)^2}}$$

3.5.2 Hoek α

Nu zijn alle elementen gekend om door middel van de cosinus regel:

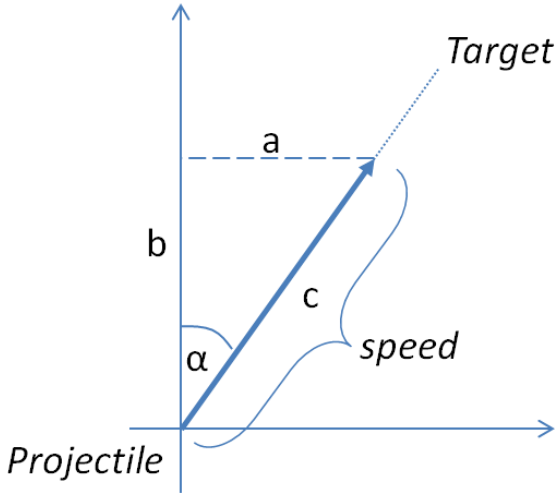
$$a^2 = b^2 + c^2 - 2bc\cos A$$

de hoek α te bepalen:

$$\cos\alpha = \frac{b^2 + c^2 - a^2}{2bc}$$

$$\boldsymbol{\alpha = \cos^{-1}\left(\frac{b^2 + c^2 - a^2}{2bc}\right)}$$

3.5.3 Snelheid x en y berekenen



Figuur 9: Aandeel van x en y in speed berekenen

Om het aandeel x en y te berekenen in de snelheid, gebruiken we de hoek α die zonet werd berekend, alsook twee basisregels uit de goniometrie:

$$\cos\alpha = \frac{b}{c}$$

en

$$\sin\alpha = \frac{a}{c}$$

We weten dat c in dit geval gelijk is aan de variabele speed en b en a respectievelijk de $speed_y$ en $speed_x$ voorstellen. Dit wordt dus:

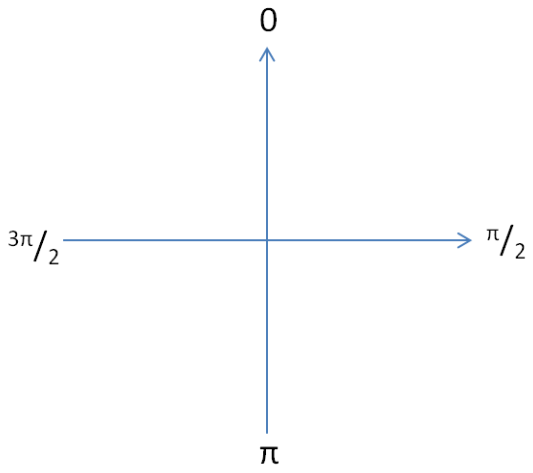
$$\boldsymbol{speed_x = \sin\alpha * speed}$$

en

$$\boldsymbol{speed_y = \cos\alpha * speed}$$

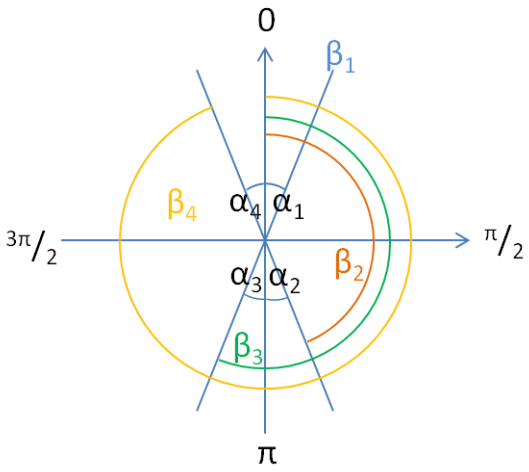
3.5.4 Render hoek versus alpha

De hoek α is enkel nuttig bij de berekening van $speed_x$ en $speed_y$. Voor het effectief renderen van projectielen onder een bepaalde hoek (angle veld van een Projectile), moet aandacht geschonken worden voor het feit, dat daar de hoek telt (in radialen) startend bij 0 zoals aangegeven in Figuur 10 .



Figuur 10: Radialen

De hoeken β (Figuur 11) worden daar dus als volgt verkregen:



Figuur 11: Render hoek β in functie van α

$$\alpha_1 \Rightarrow \beta_1 = \alpha_1$$

$$\alpha_2 \Rightarrow \beta_2 = \pi - \alpha_2$$

$$\alpha_3 \Rightarrow \beta_3 = \pi + \alpha_3$$

$$\alpha_4 \Rightarrow \beta_4 = 2\pi - \alpha_4$$

In util.h zijn hulpfuncties opgenomen om jullie te helpen bij deze berekeningen. Implementeer deze functies eerst, zodat je ze kan gebruiken.

3.6 Game-loop

De basis structuur van de game-loop wordt opgegeven. 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 GameState (zie 4.3.6) en wordt telkens meegegeven bij het oproepen van elke stap.

In eerste instantie wordt init_gui() aangeroepen. Deze functie moet altijd als eerste aangeroepen worden en initialiseert de gehele gui. Hier geef je de grootte van het scherm mee, deze parameters zijn in config.h vastgelegd door middel van constanten.

Vervolgens wordt de init_sprite_cache() functie aangeroepen. Deze gaat alle sprites inladen, zodat ze in het geheugen zitten alvorens we met de eigenlijke game-loop beginnen.

De functie render_world_to_sprite() dient om eerst de wereld stuk voor stuk te renderen en dan het geheel als één enkele sprite op te slaan. Zo kan ze gebruikt worden als één sprite, in plaats van elke iteratie van de game-loop het gehele speelveld op nieuwe te moeten samenstellen en tekenen.

Met de functie update_hud() wordt dan alle informatie voor de hud ingesteld. Dit wordt gedaan door de adressen van de plaatsen waar de informatie staat, door te geven. Op deze manier kan de Hud struct zijn pointers daar naartoe laten wijzen en is de informatie altijd up to date.

init_game_loop() is de laatse functie die voor de game-loop mag aangeroepen worden. Daarin worden ook het aantal frames per second ingesteld. Ook die staat gedefinieerd in de config.h. Een goede waarde is 60 fps.

Hierna begint de game-loop. Door de initialisaties vooraf aan te roepen, is er een event systeem opgezet. Dus het eerste wat moet gedaan worden in de loop, is luisteren naar binnenkomende events.

De functie wait_for_event() wacht totdat het volgende Event binnenkomt. Eens dat gebeurd is wordt er gekeken welk soort Event het is. Aan de hand daarvan worden de juiste acties uitgevoerd.

- EVENT_TIMER
  Dit Event wordt elke FPS keer per seconde gegenereerd. Dit is het kloppend hart van het spel. Er moeten een aantal dingen gebeuren. Eerste en vooral wordt in de GameState een vlag aangezet die zegt dat er eventueel mag worden gerendered naar het scherm straks. Vervolgens updaten we alle logica van het spel, maar enkel en alleen als de GameState niet aangeeft dat het spel verloren is.
    - check_spells() zal kijken of er bepaalde spreuken actief zijn, zo ja zullen deze uitgevoerd worden.
    - check_enemy_wave() zal kijken of de huidige wave al verslaan is. Indien dat zo is, zal de cooldown periode gecontroleerd worden, is deze verstreken, dan wordt de volgende wave klaargemaakt.

    - update_movement() zal alle beweging in het spel een stap verder zetten. Dit gaat zowel over de Enemies die rondlopen, als de Projectiles van de Towers die rondvliegen.
    - do_tower_attacks() zal controleren of torens nu een doelwit binnen hun bereik vinden, en zo ja, zorgen dat er op geschoten wordt (Projectiles klaarzetten die in de volgende iteratie van de loop beginnen bewegen)
- EVENT_MOUSE_MOVE
  Dit Event wordt telkens de muis beweegt over het venster, gegenereerd. Deze roept enkel een mouse_move() functie op die dan de muis beweging afhandelt.
- EVENT_MOUSE_DOWN
  Dit Event wordt telkens een muisknop ingedrukt wordt, gegenereerd. Deze roept enkel een mouse_down() functie op die dan de muis klik afhandelt.
- EVENT_MOUSE_UP
  Dit Event wordt telkens een muisknop losgelaten wordt, gegenereerd. Deze roept enkel een mouse_up() functie op die dan de muis klik afhandelt.
- EVENT_DISPLAY_CLOSE
  Dit Event wordt gegenereerd wanneer de gebruiker op het kruisje van het venster klikt, om het spel af te sluiten. Hierin wordt gewoon de loop conditie op 0 gezet, waardoor we uit de game-loop springen.

Het laatste wat gedaan wordt in de game-loop is het effectief renderen (tekenen) naar het scherm. Dit wordt echter enkel gedaan wanneer twee voorwaarden voldaan zijn. De eerste is uiteraard, wanneer in de GameState ingesteld staat dat er mag hertekend worden (omdat de game logica gewijzigd is). De tweede conditie zegt dat er enkel mag getekend worden, wanneer er momenteel geen Events meer wachten. Dit zorgt ervoor dat we geen input verwerking uitstellen, gewoon om een frame te tekenen. Hierdoor blijft een spel goed reageren, het feit dat je misschien eens een seconde 59 frames per second haalt in plaats van 60 frames per second, merkt een menselijk oog toch niet.

Eens er uit de game-loop gesprongen wordt worden nog twee opkuis functies aangeroepen: cleanup_game_loop() en cleanup_sprite_cache(). Deze zullen gealloceerd geheugen opruimen en een aantal achterliggende componenten stop zetten.

```
int run_game_loop(GameState * state)
{
	int done = 0;
	int i = 0;

	// Initialise gui
	init_gui(SCREEN_WIDTH,SCREEN_HEIGHT);

	// Load resources
	init_sprite_cache();
	render_world_to_sprite(&state->world);
	update_hud(
		&state->hud,
		&state->score,
		&state->mana,
		&state->money,
		&state->world.castle->castle.health,
		&state->wave.wave_number);

	// Initialise game loop
	init_game_loop(FPS);
	while (!done)
	{
		// Get events
		Event ev;
		wait_for_event(&ev);

		// Event handlers
		switch (ev.type) {
			case EVENT_TIMER:
				state->redraw = 1;
				if(!state->game_over) {
					check_spells(state);
					check_enemy_wave(state);
					update_movement(state);
					do_tower_attacks(state);
				}
				break;
			case EVENT_MOUSE_MOVE:
				mouse_move(&ev.mouseMoveEvent, state);
				break;
			case EVENT_MOUSE_DOWN:
				mouse_down(&ev.mouseDownEvent, state);
				break;
			case EVENT_MOUSE_UP:
				mouse_up(&ev.mouseUpEvent, state);
				break;
			case EVENT_DISPLAY_CLOSE:
				done = 1;
				break;
		}

		// Render only on timer event AND if all movement and logic was processed
		if (state->redraw && all_events_processed()) {
			render_game(state);
		}
	}

	// Cleanup
	cleanup_game_loop();
	cleanup_sprite_cache();
	return 0;
}
```

4 Overige componenten

4.1 World

World (in world.c en world.h) is een struct die de spelwereld voorstelt. Het heeft volgende velden:

- `int width_in_tiles:` breedte van de wereld (in tegels, vastgelegd in config.h)
- `int height_in_tiles:` hoogte van de wereld (in tegels, vastgelegd in config.h)
- `Entity ** entities:` 2D array van entities, stelt het tegel-raster voor
- `Entity * spawn:` pointer naar de spawn-entiteit
- `Entity * castle:` pointer naar de castle-entiteit

Op een World kunnen de volgende te implementeren operaties uitgevoerd worden (world.h):

- `void init_world(World * world, int width_in_tiles, int heigth_in_tiles);`
  Initialiseert de wereld en reserveert geheugenruimte voor elk van de tegels.

- `void init_world_from_file(World * world, char * worldFile);`
  Initialiseert de wereld aan de hand van een file.

- `Entity * place_tower(World * world, TowerType type, float world_x, float world_y);`
  Plaatst een toren van het gegeven type op de gegeven locatie.

- `void destroy_tower(World * world, Entity * tower);`
  Verwijdert de toren uit de World door het geheugen ingenomen door de Projectiles vrij te geven en het type van de tower Entity op EMPTY te zetten.

- `void destroy_world(World * world);`
  Geeft de geheugenruimte die ingenomen is door de wereld vrij.

4.2 Hud

De Hud struct heeft een aparte hud.c en hud.h file gekregen. Deze bevat ook een aantal andere structs:

struct Bounds:

- `float screen_sx:` start x coördinaat van de linkerbovenhoek van de rechthoek.
- `float screen_sy:` start y coördinaat van de linkerbovenhoek van de rechthoek.
- `float screen_dx:` destination x coördinaat van de rechteronderhoek van de rechthoek.
- `float screen_dy:` destination y coördinaat van de rechteronderhoek van de rechthoek.

struct Button:

- `ButtonState state:` enum die beschrijft in welke toestand de achtergrond van een Button zich momenteel bevindt: BUTTON_UP, BUTTON_HOVER of BUTTON_DOWN.
- `Bounds bounds:` een Bounds struct om de grootte en vorm van de Button te bepalen, waarbinnen de muis cursor wordt gedetecteerd.

struct Hud:

- `long * score:` een pointer naar een veld waar de score bijgehouden wordt.
- `long * mana:` een pointer naar een veld waar de mana bijgehouden wordt.
- `long * money:` een pointer naar een veld waar het geld bijgehouden wordt.
- `int * castle_health:` een pointer naar een veld waar de levenspunten van het castle bijgehouden worden.
- `int * wave_number:` een pointer naar een veld waar het wave nummer bijgehouden wordt.
- Button buttons[BUTTON_AMOUNT]: een array met alle buttons van de UI in.

Verder worden nog de volgende voor jullie reeds geïmplementeerde functies beschreven:

- `void init_buttons(Hud * hud);`
  Initialiseert de buttons. Hun bounds en backgrounds worden goed gezet.

- `void render_buttons(Hud * hud);`
  Rendert de buttons met hun tekst.

- `int in_bounds(float screen_x, float screen_y, Bounds bounds);`
  Kijkt of een bepaalde screen coördinaat binnen een Bounds struct valt.

- `void update_hud(Hud * hud, long * score, long * mana, long * money, int * castle_health, int * wave_number);`
  Doet alle pointers in de hud struct wijzen naar de juist velden. Hierdoor moet de Hud struct zelf niet meer geupdatet worden en kunnen deze pointers gevolgd worden bij het renderen van de Hud informatie.

4.3 Game

In game.h staan een aantal structs die allemaal deel uitmaken van de GameState Struct. We bekijken ze eerst afzonderlijk en vervolgens de GameState struct. De World (4.1) en Hud (4.2) struct werd hierboven reeds uitgelegd.

4.3.1 WaveInfo

WaveInfo is een struct die een wave beschrijft. Een wave is een golf van vijanden die één voor één op het scherm zullen verschijnen. Deze struct bevat de volgende velden:

- `int wave_number:` de hoeveelste wave dit is
- `int spawn_interval:` hoeveel tijd er tussen het spawnen van vijanden zit
- `int spawn_counter:` aantal frames verstreken sinds het spawnen van de vorige vijand
- `int started:` is de wave al gestart
- `int completed:` is de wave voltooid
- `int cool_down:` hoe lang de cooldown voor de wave moet duren
- `int cool_down_counter:` aantal frames verstreken sinds start wave
- `int nr_of_enemies:` aantal enemies in de wave
- `int nr_of_spawned_enemies:` aantal gespawnede enemies tot nu toe
- `int boss_wave:` is de wave een wave met een boss in

De inhoud van deze velden en hoe ze evolueren tussen elke nieuwe wave, bepalen het algemeen verloop en de moeilijkheidsgraad van het spel. We laten jullie vrij in hoe je dit precies implementeert, maar omdat het zoeken naar een goede balans tussen de waarden niet evident is, geven we volgende informatie op die als basis kan gebruikt worden:

- spawn_interval van vier keer het aantal FPS
- cool_down tussen waves van acht keer het aantal FPS
- nr_of_enemies is gelijk aan 4, plus het huidige wave nummer

Verder wordt de moeilijkheidsgraad bepaald door het aantal levens van de vijand en de frequentie waarmee bepaalde types spawnen. Een leuke toevoeging is als het aantal levens van een vijand exponentieel toeneemt naarmate het spel vordert, wij hebben hiervoor volgende formule gebruikt:

```
health = base health + (wave nummer * (wave_nummer / 3) * base health * 0.1)
```

Voor de spawn frequenties van de verschillende types vijanden zijn we als volgt tewerk gegaan:

- Als geen van de volgende condities voldaan is, dan spawnt een gewone vijand
- Elke vijf vijanden wordt een ELITE vijand gespawned.
- Vanaf wave vier is er 10% kans dat een AIR vijand gespawned wordt
- Vanaf wave acht is er 5% kans dat een FAST vijand gespawned wordt
- Elke vijf waves wordt een BOSS gespawned als laatste vijand in de wave

4.3.2 Blueprint

Blueprint is een struct die een soort van prototype omschrijft van een Tower die zal gebouwd worden. Het bevat een aantal velden die nuttig zijn voor het al dan niet renderen van een groen of rood doorschijnende selectie tile onder de muis cursor. De volgende velden staan in deze struct:

- `int valid:` is de blueprint valid, mag een Tower geplaatst worden?
- `Entity entity:` de Tower Entity die geplaatst zal worden (om range en type uit te kunnen halen. Bij het renderen van een eventuele selectie-tile moet ook in een doorzichtig wit een cirkel van het bereik van de toren getekend worden er rond.

4.3.3 Mouse

Deze struct houdt alle coördinaten, zoals in de conventie (3.1.2 Coördinaten) afgesproken, bij. Volgende velden staan in de struct:

- `float screen_x:` de x scherm coördinaat.
- `float screen_y:` de y scherm coördinaat.
- `float world_x:` de x world coördinaat.
- `float world_y:` de y world coördinaat.
- `int tile_x:` de x tile coördinaat.
- `int tile_y:` de y tile coördinaat.
- `int tile_changed:` is de tile veranderd ten opzicht van de vorige iteratie?

4.3.4 Spells

Spells is een simpele struct die een aantal velden bundelt die nodig zijn om spell-effecten uit te voeren.

- `int frost_wave_active:` is de frost wave actief?
- `int poison_gas_active:` is het poison gas actief?
- `int spell_duration:` hoe lang duurt de actieve spell?

4.3.5 Action

Action is geen struct, maar een *enum*. Deze enum bevat een waarde naargelang welke Button net ingedrukt is in de UI:

- `NONE:` er is recent geen Button ingedrukt.
- `BUILD_TOWER:` er is op één van de Buttons om een toren te bouwen gedrukt.
- `DESTROY_TOWER:` er is op de Button om torens te vernietigen gedrukt.
- `GENERATE_FROST_WAVE:` er is op de Frost wave spell Button gedrukt.
- `POISON_GAS:` er is op de Poison gas spell Button gedrukt.

Vergeet niet na het verwerken van deze actie in een iteratie van de game loop, om de actie terug op NONE te zetten, anders wordt de actie opnieuw uitgevoerd in de volgende iteratie.

4.3.6 GameState

GameState is een struct en deze houdt de state van het spel bij. Het heeft de volgende velden:

- `Entity * enemies:` array van Entity structs om de huidige Enemies op het speelveld bij te houden
- `Entity ** towers:` array van Entity **pointers**. Alle torens worden bijgehouden in World, maar met deze array kan men aan de torens via de GameState.
- `int enemies_length:` lengte van de enemies array.
- `int towers_length:` lengte van de towers array.
- `World world:` World struct, waarin de wereld en de statische objecten bijgehouden worden.
- `WaveInfo wave:` WaveInfo struct die alle relevante informatie met betrekking tot de huidige wave bijhoudt.
- `Action action:` Action enum, die bepaalt welke actie er momenteel actief is.
- `Blueprint blueprint: de` Blueprint struct bevat alle informatie over een eventuele Tower blueprint die moet gerenderd worden op het scherm.
- `Mouse mouse:` de Mouse struct bevat alle muis coördinaten die nodig kunnen zijn in het spel.
- `Hud hud:` de Hud struct houdt alles bij die op de hud moet komen, zoals geld, score, buttons en dergelijke.
- `Spells spells:` de Spells struct bundelt alle informatie over een eventueel actieve spells.
- `int refresh_paths:` int die als boolean wordt gebruikt om aan te geven wanneer de pathfinding moet herberekend worden.
- `int redraw:` int die als boolean wordt gebruikt om aan te geven wanneer alles opnieuw naar het scherm moet gerenderd worden.
- `long money:` hoeveel geld de speler heeft.
- `long score:` wat de score van de speler is.
- `long mana:` hoeveel mana de speler heeft.
- `int game_over:` int die als boolean wordt gebruikt om aan te geven wanneer de speler het spel verloren heeft.

4.3.7 Functies

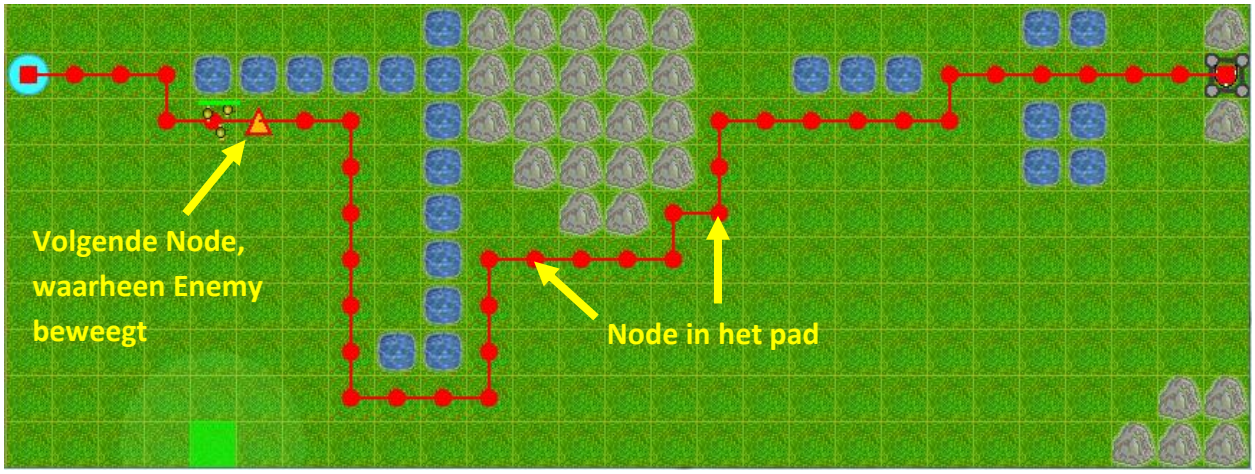
Volgende te implementeren functies zijn beschikbaar:

- `int run_game_loop(GameState * state);`
  Voert de game loop uit.

- `void check_spells(GameState * state);`
  Controleert of er op bepaalde spells gedrukt is (Action). Zo ja, dan wordt indien de speler genoeg mana heeft hun effect op actief gezet en de mana kost afgetrokken. Staat een spell al op actief, dan wordt gekeken of ze nog steeds actief mag zijn, zo niet wordt ze op non-actief gezet. De Action wordt ook terug op NONE gezet.

- `void check_enemy_wave(GameState * state);`
  Controleert of een nieuwe wave mag gestart worden, door eerst de *cooldown* periode van de nieuwe wave af te wachten. Na de cooldown periode wordt de wave geïnitialiseerd. Vervolgens wordt er na een bepaald interval telkens een nieuwe Enemy van de wave gespawned (op het spawnpunt tot leven gewekt), tenzij er juist een Frost wave bezig is. Is de wave klaar, dan worden de voorbereiding getroffen voor de volgende wave.

- `void update_movement(GameState * state);`
  Alvorens iets te doen wordt gekeken of de GameState vraagt om de pathfinding opnieuw uit te voeren, zo ja, dan wordt dit voor alle levende Enemies eerst gedaan en vervolgens de vlag in de GameState uitgezet. Daarna wordt van alle levende Enemies de positie richting hun volgende path-node opgeteld met hun speed, vermenigvuldigd met hun *direction vector* (Figuur 12). Het is dus belangrijk eerst de direction vector te bepalen (zie 7.2). Gebruik hiervoor de hulpfunctie in 4.6. Ook de hoek waarin deze sprite uiteindelijk zal gerenderd worden, moet berekend worden. (0 rad = noord georiënteerd). Komt een vijand aan het kasteel, dan wordt daar schade berokkend en gaat die vijand dood.
  (Vergeet niet: Enemies moeten enkel horizontaal of verticaal kunnen bewegen)



Figuur 12: Beweging langs een pad

Hierna wordt iets gelijkaardigs gedaan met de movement van de Projectiles van Towers. Het grote verschil is dat die Projectiles ook diagonaal onder om het even welke hoeken kunnen vliegen. Om de correcte speed_x en speed_y alsook de hoek te kunnen berekenen, verwijzen we naar 3.5 Targetting.
(Een raket zijn snelheid mag tijdens zijn traject versnellen met een factor bv. 1.2);

- `void do_tower_attacks(GameState * state);`
  Voor alle Towers wordt gekeken of ze een Enemy binnen hun bereik als target kunnen vastleggen. Zo ja, dan blijft dit het target van de Tower in kwestie, tot die Enemy buiten het bereik van de toren gaat of dood is. Dan zoekt de Tower naar een nieuw doelwit. Heeft de Tower al een target? Dan wordt er op geschoten door een Projectile klaar te zetten (correcte locatie en live) indien het *shoot_interval* verstreken is. Reeds bestaande Projectiles moeten gecontroleerd worden op collision. Indien ze een Enemy raken, moeten ze schade aan de Enemy doen (indien ze dit kunnen, bv. machine gun op AIR Enemy gaat niet) en verdwijnen (live vlag uit).
  Indien Enemies hierdoor doodgaan, moet de speler uiteraard geld en score punten verdienen. Enkele hulp-functies voor de implementatie van deze functie zijn gespecificeerd in tower_ai.h (zie 4.4).

- `void init_game_state(GameState * state);`
  Hier worden alle velden van de GameState en zijn onderliggende structs geïnitialiseerd op standaard waarden. Op deze manier kunnen geen onverwachte waarden opduiken bij het begin van het spel. Ook de spelwereld wordt hier best ingeladen.
  Vergeet hier ook niet de voor jullie reeds geïmplementeerde functie init_buttons(Hud * hud) aan te roepen.

- `void destroy_game_state(GameState * state);`
  De volledige GameState wordt opgeruimd. Eventueel gealloceerd geheugen moet hier zeker vrijgemaakt worden!
  (Let ook op dat je niet reeds vrijgegeven geheugen probeert vrij te maken, want dit doet je programma crashen)

- `void render_game(GameState * state);`
  Deze functie is reeds geïmplementeerd en wordt wat verder volledig overlopen. (zie 4.5)

- `void update_mouse(GameState * state, float screen_x, float screen_y);`
  Deze functie is reeds geïmplementeerd. Hier worden de screen coördinaten uit het EVENT_MOUSE_MOVE gebruikt en gekopieerd in de Mouse struct. Om makkelijker te werken, worden deze screen coördinaten ook volgens de afgesproken conventie (3.1.2), geconverteerd naar world en tile coördinaten en ook gekopieerd in de Mouse struct. Hier kan ook de *tile_changed* vlag correct ingevuld worden.

- `void mouse_move(MouseMoveEvent * ev, GameState * state);`
  Het stuk dat de Buttons afhandelt, is hier reeds geïmplementeerd voor jullie. In het geval dat de muis cursor in het world-screen zit en er is een Action actief die een Blueprint onder de muis moet tekenen, moet gekeken worden of deze blueprint *valid* is. Dat betekent de valid vlag van de Blueprint struct aanpassen naargelang welk type Entity onder de cursor zit en naargelang de speler genoeg geld heeft om te bouwen wat hij/zij wil bouwen.

- `void mouse_down(MouseDownEvent * ev, GameState * state);`
  Het stuk dat de Buttons afhandelt, is hier reeds geïmplementeerd voor jullie. In het geval dat de muis cursor in het world-screen zit en er is een Action actief, dan moet er gehandeld worden naar die Action. Dat betekent, een Toren zetten indien nodig, maar enkel en alleen als er nog een bestaand Path is van de *spawn* naar het *castle* en als de Blueprint valid was. Wordt de Tower gebouwd, dan moet deze aan de World toegevoegd worden, maar ook een pointer ernaartoe aan de GameState toegevoegd worden, zodoende later gemakkelijk aan de Tower in kwestie te geraken. Bij het plaatsen van een Toren in het spel, moet ook de vlag in GameState aangezet worden, om de pathfinding van Enemies opnieuw te berekenen. Indien de actie het vernietigen van een Toren is, dan moeten de Projectiles van deze toren correct opgeruimd worden (zie 4.1 World bij destroy_tower) en moet de pathfinding ook hier opnieuw herberekend worden (door middel van de vlag).

- `void mouse_up(MouseUpEvent * ev, GameState * state);`
  Dient enkel om Buttons code aan te roepen, die reeds geïmplementeerd is voor jullie. Geen aanpassingen nodig dus.

4.4 Tower AI

Een aantal functies die belangrijk zijn voor de afhandeling van de torens hebben we voor de duidelijkheid in een aparte component Tower AI ondergebracht. Deze wordt gespecificeerd in de header-file tower_ai.h. De volgende functies dienen hiervoor geïmplementeerd te worden:

- `int is_valid_target(Tower * t, Enemy * e);`
  Geeft 1 terug als de gegeven Enemy een geldig doelwit is voor de gegeven toren. Dit wil zeggen dat het binnen bereik is en dat de toren mag schieten op dit type vijand (vb. enkel flak toren kan op vliegende vijanden schieten). Als dit niet het geval is, dan geeft deze functie 0 terug.

- `Enemy * find_target(Tower * t, Entity * enemies, int enemies_length);`
  Zoekt een geldig doelwit voor de gegeven toren op basis van de is_valid_target functie. Als er geen doelwit kan gevonden worden, dan geeft deze functie NULL terug.

- `void shoot(Tower * t);`
  Laat de gegeven toren een projectiel afvuren op het huidige doelwit.

- `int is_out_of_range(Projectile * p, Tower * t, World * w);`
  Controleert of het gegeven projectile nog binnen bereik is (i.e. binnen het bereik van de toren en binnen de dimensies van de wereld). Geeft 1 terug als dit het geval is, anders 0;

- `int do_damage(Projectile * p, Enemy * e);`
  Past de schade van het gegeven projectiel toe op de gegeven vijand. Deze functie controleert hierbij of dit type projectiel de gegeven vijand wel schade kan toedienen (e.g. een projectiel van het type ROCKET zal geen schade toebrengen aan een AIR vijand). Als dit het geval is, dan wordt de eigenschap live van het projectiel op 0 gezet.

4.5 Rendering

De hoofd render functie staat in game.h. Deze is reeds geïmplementeerd en rendert het volledige spel. De verschillende functies die aangeroepen worden, komen uit render.h. Hieronder overlopen we eerste de volledige render_game functie.

Eerste wordt de *redraw* vlag uit de GameState terug op 0 gezet. (Aangezien we in de render_game functie zitten, stond die op 1).

Het renderen naar een scherm gebeurt in lagen. Wat eerst gerenderd wordt, staat helemaal onderaan de lagen, alles wat vervolgens gerenderd wordt, staat erboven, enzoverder. Daarom wordt eerst de volledige wereld gerenderd, het "speelveld". We doen dit door de *render_world* functie op te roepen.

Dan renderen we de pathfinding paden van ventjes, maar enkel indien deze vlag aangezet is in de config.h configuratie. De functie *render_paths* is hiervoor al geïmplementeerd.

Vervolgens worden alle torens overlopen en ook gerenderd. Een Tower is een Entity, dus we doen dit met de *render_entity* functie.

We doen hetzelfde met de Enemies. We loopen over alle enemies die nog leven en tekenen ze met de *render_entity* functie.

Bovenop alles moet een doorzichtige kleurlaag gerenderd worden, wanneer een Spell actief is. Daarom volgt nu de *render_spellscreen* functie.

Boven dit spellscreen mogen de projectielen nog getekend worden. Alle Torens worden nog eens overlopen, maar nu wordt de *render_projectiles* functie aangeroepen.

Bovenop dit alles moet er soms een blueprint onder de muiscursor getekend worden, met een aanduiding van het bereik van de gekozen Tower. Daarvoor wordt de functie *render_mouse_actions* aangeroepen.

Als laatste wordt de ui er rond getekend met *render_ui*. Daarin staat al een stuk geïmplementeerde code voor de Buttons.

We renderen nog het aantal frames per seconds op het scherm. En als allerlaatste wordt bovenop alles nog een "Game Over" getekend, indien nodig, met *render_game_over*.

Belangrijk om weten is dat al deze render statements eigenlijk naar een verborgen (onzichtbare) bitmap tekenen. Enkel wanneer we *flip_display()* aanroepen, wordt deze bitmap gewisseld met die op het scherm. Op deze manier zie je geen flikkering bij het tekenen. Deze techniek wordt ook *double buffering* genoemd. Op het einde wordt nog een *clear_to_color* functie opgeroepen, die de nieuwe verborgen bitmap (het vorige scherm) wist, zodat de volgende oproep naar de render_game functie met een schone lei kan beginnen.

```
void render_game(GameState * state)
{
	// Variables
	Color color = {255, 0, 255, 255};
	Color black = {0, 0, 0, 255};
	Color white = {255, 255, 255, 255};
	char buffer [20];
	int i,n;

	// Set redraw off
	state->redraw = 0;

	// Render world
	render_world(&state->world);

	// Render pathfinding, if enabled
	if (SHOW_PATHFINDING)
		render_paths(state);

	// Render towers
	for(i = 0; i < state->towers_length; i++) {
		render_entity(state->towers[i]);
	}

	// Render enemies if they are alive
	for(i = 0; i < state->enemies_length; i++) {
		if(state->enemies[i].enemy.alive)
			render_entity(&state->enemies[i]);
	}

	// Render spellscreen
	render_spellscreen(&state->spells);

	// Render projectiles
	for(i = 0; i < state->towers_length; i++) {
		render_projectiles(state->towers[i]);
	}


	// render selection
	render_mouse_actions(state);

	// Render ui on top
	render_ui(state);

	// Fps rendering
	n = sprintf(buffer, "%#.1f FPS", get_current_fps());
	draw_text(buffer, FONT_LARGE, color, SCREEN_WIDTH-20 -100, 6, ALIGN_RIGHT);

	// Game over?
	render_game_over(state);

	// Render to screen
	flip_display();
	clear_to_color(color);
}
```

Opmerking: *In het geval van renderen werk je steeds met screen_x en screen_y coordinaten. De bedoeling is dat je tijdens de logica van het spel niet moet denken aan screen coördinaten, maar dat je de conversies naar screen-coördinaten allemaal* ***in*** *de render.c functies doet.*

Render.h bevat de volgende functies, waarvan de meesten nog moeten geïmplementeerd worden:

- `void render_world_to_sprite(World * world);`
  Deze functie wordt aangeroepen voor de game_loop, na het initialiseren van de sprite cache (init_sprite_cache). Hiermee wordt de gehele world gerenderd, maar naar een aparte bitmap **(gebruik hiervoor de functies start_drawing_world() vóór het tekenen en stop_drawing_world() na het tekenen)**. Deze wordt vervolgens als sprite bewaard en kan dan later via de render_world functie opgeroepen worden als een sprite. We doen dit omdat we anders verschillende keren per seconde, bv. 60 keer bij 60 fps, het gehele speelveld, tile per tile moeten tekenen. We hebben 27x17 tiles, dus dat zou 459 x 60 = 27540 draw operaties zijn per seconde, enkel en alleen voor de achtergrond. Via deze manier, kunnen we dat reduceren tot 1 x 60 draw operaties per seconde.
  Deze functie zal dus alle Entities overlopen van de World en telkens een Grass sprite tekenen (draw_sprite), en indien nodig een specifieke entity sprite daar bovenop (render_entity).
  Indien er in de config.h SHOW_GRID aanstaat, teken je ook een lijnraster over de tiles, zodat je gemakkelijk tiles kunt zien, tijdens het debuggen.

- `void render_world(World * world);`
  Deze functie zal de volledige world sprite tekenen en daarbovenop de castle Entity en spawn Entity renderen met render_entity (omdat deze geanimeerd zijn).

- `void render_spellscreen(Spells * spells);`
  Render een half-doorzichtige rechthoek over het speelveld in een aangepaste kleur (blauw-achtig voor frost wave, groen-achtig voor poison gas). Om transparantie te gebruiken moet je voor het renderen van een transparante kleur de functie *set_transparancy_on()* aanroepen en nadien de functie *set_transparancy_off()*. Deze beide functies komen uit de gui.h file.

- `void render_entity(Entity * entity);`
  Deze functie krijgt een Entity. Via een switch statement op het type veld, kan je beslissen welk soort Entity dit is en vervolgens de juist sprite tekenen. Sommige types hebben subtypes. Enemies hebben een levenspunten bar bovenop hun sprite. Die bestaat uit twee rechthoeken, een rode op 100% en een groene erboven op het percentage levenspunten die de Enemy nog overheeft.
  Opmerking: *Sommige sprites hebben animaties. Om het eenvoudig te houden in het begin, kan je deze in plaats van met de draw_sprite_animated functie eerst nog met de draw_sprite functie aanroepen. Dan wordt enkel hun eerste animatie frame getoond als sprite.*

- `void render_projectiles(Entity * tower);`
  Hier krijgen we een Tower Entity. We loopen over de Projectiles die momenteel in leven zijn en renderen ze onder de juiste hoek. Normaal gezien is in de update_movement functie van game.c hun *angle* correct bewaard.

- `void render_ui(GameState * state);`
  Teken de volledige ui. De Buttons werden reeds gerenderd voor jullie. Onder de Buttons laag, moet nog een ui sprite gerenderd worden. Bovenop dit alles moet de hud (heads up display) informatie komen. (render_hud)

- `void render_hud(Hud * hud);`
  Render de hud informatie. Dit gaat over de volgende informatievelden: money, score, castle levenspunten, mana, nummer van de huidige wave. Deze kunnen allemaal bovenaan in het scherm gerenderd worden.

- `void render_game_over(GameState * state);`
  Deze functie moet gewoon in het midden van het scherm in het grootst mogelijk font: “GAME OVER” weergeven.

- `void render_mouse_actions(GameState * state);`
  Deze functie kijkt eerst of de muis cursor boven het World scherm staat. Zoja, dan moet deze in sommige gevallen (aangegeven door de action van GameState) een blueprint tekenen. Dit is een groen transparante tile wanneer de blueprint valid is, en een rood transparante tile wanneer niet. Daar rond wordt het bereik van de te bouwen toren getekend door een witte redelijk doorzichtige cirkel.
  Opmerking: *door de manier waarop de ui bovenop de render_mouse_actions getekend wordt (render_mouse_actions komt dus voor render_ui in de render_game functie), moeten we geen rekening houden met een bereik-cirkel die eventueel over de Buttons zou getekend worden. Visueel zie je dit niet, want de cirkel wordt onder de Buttons getekend, dit mag zo blijven, er moet geen speciaal randgeval voor geprogrammeerd worden.*

- `void render_paths(GameState * state);`
  Deze functie is reeds geïmplementeerd voor jullie. Ze tekent de pathfinding paden van levende enemies op het scherm. Hier moet dus niets aan aangepast worden!
  Opmerking*: natuurlijk moet voor het gebruik van deze functie, die kan geactiveerd worden met een debug vlag in config.h, de pathfinding reeds geïmplementeerd zijn. Het werkt namelijk enkel wanneer alle levende vijanden, ook een path hebben gekregen.*

4.6 Hulpfuncties en structuren

In util.h staan een aantal hulp structs en functies die het makkelijker maken om bepaalde zaken te gebruiken/coderen. Hieronder worden deze overlopen:

struct Color

- `unsigned char r`: waarde van 0-255 voor rood aandeel in rgb kleur.
- `unsigned char g`: waarde van 0-255 voor groen aandeel in rgb kleur.
- `unsigned char b`: waarde van 0-255 voor blauw aandeel in rgb kleur.
- `unsigned char a`: waarde van 0-255 voor alpha kanaal. 0 = doorzichtig; 255 = ondoorzichtig.

struct Event

De structs die gebruikt worden voor de Events zijn in feite niet belangrijk voor jullie. Dit wordt intern afgehandeld en het enige moment dat ze nodig zijn, is bij de opgeroepen handler functies in de game-loop, om waarden uit te lezen. Ze gebruiken de Union op dezelfde manier als uitgelegd in 3.2. Er is een basis union Event. Deze bevat alle onderliggende structs als velden, en ook een enum die het type bepaalt.

De onderliggende structs zijn: TimerEvent, DisplayCloseEvent, MouseMoveEvent, MouseDownEvent en MouseUpEvent. De enige plaats waar deze Events eventueel uitgelezen moeten worden door jullie is in de mouse functies, waar ze als argument meegegeven worden.

De util.h file bevat verder nog volgende te implementeren functies:

- `float convert_world2screen_x(float world_x);`
  Converteer een world_x coördinaat naar een screen_x coördinaat door de SCREEN_START_X offset uit config.h bij te tellen.

- `float convert_world2screen_y(float world_y);`
  Converteer een world_y coördinaat naar een screen_y coördinaat door de SCREEN_START_Y offset uit config.h bij te tellen.

- `float convert_tile2screen_x(float tile_x);`
  Converteer een tile_x coördinaat naar een screen_x coördinaat door tile_x te vermenigvuldigen met de TILE_SIZE en daar de SCREEN_START_X offset bij te tellen. Beiden staan in config.h gedefinieerd.

- `float convert_tile2screen_y(float tile_y);`
  Converteer een tile_y coördinaat naar een screen_y coördinaat door tile_y te vermenigvuldigen met de TILE_SIZE en daar de SCREEN_START_Y offset bij te tellen. Beiden staan in config.h gedefinieerd.

- `float convert_tile2world_x(int tile_x);`
  Converteer een tile_x coördinaat naar een world_x coördinaat door tile_x te vermenigvuldigen met de TILE_SIZE uit config.h.

- `float convert_tile2world_y(int tile_y);`
  Converteer een tile_y coördinaat naar een world_y coördinaat door tile_y te vermenigvuldigen met de TILE_SIZE uit config.h.

- `float convert_screen2world_x(float screen_x);`
  Converteer een screen_x coördinaat naar een world_x coördinaat door ze met de SCREEN_START_X offset uit config.h te verminderen.

- `float convert_screen2world_y(float screen_y);`
  Converteer een screen_y coördinaat naar een world_y coördinaat door ze met de SCREEN_START_Y offset uit config.h te verminderen.

- `int convert_screen2tile_x(float screen_x);`
  Converteer een screen_x coördinaat naar een tile_x coördinaat door ze met de SCREEN_START_X offset uit config.h te verminderen en vervolgens een gehele deling door TILE_SIZE uit te voeren. (cast naar int)

- `int convert_screen2tile_y(float screen_y);`
  Converteer een screen_y coördinaat naar een tile_y coördinaat door ze met de SCREEN_START_Y offset uit config.h te verminderen en vervolgens een gehele deling door TILE_SIZE uit te voeren. (cast naar int)

- `int convert_world2tile_x(float world_x);`
  Converteer een world_x coördinaat naar een tile_x coördinaat een gehele deling door TILE_SIZE uit config.h uit te voeren. (cast naar int)

- `int convert_world2tile_y(float world_y);`
  Converteer een world_y coördinaat naar een tile_y coördinaat een gehele deling door TILE_SIZE uit config.h uit te voeren. (cast naar int)

- `float euclidean_distance(float x1, float y1, float x2, float y2);`
  Bereken de euclidische afstand tussen twee punten. De formule om dit te doen staat uitgelegd in 3.5.1 Lengte van de zijden.
  Opmerking: *een machtsverheffing is efficiënter als je gewoon een vermenigvuldiging doet met zichzelf in plaats van een machtsfunctie te gebruiken.*

- `float find_alpha(float horizontal_length, float vertical_length, float diagonal_length);`
  Zoek de hoek alpha tussen de vertical en schuine zijde van de driekhoek opgebouwd uit de gegeven zijde-lengtes. Hoe je dit moet doen, vind je ook terug in 3.5.2 Hoek α. Geef de hoek terug in radialen!

- `void dissolve_speed(float alpha_radians, float speed, float * horizontal, float * vertical);`
  Ontbind de speed variabele in een horizontale en verticale component. De opgegeven alpha moet in radialen staan! Hoe je dit doet, vind je terug in 3.5.3 Snelheid x en y berekenen. De laatste twee argumenten zijn pointers, zodat deze kunnen aangepast worden door de functie.

- `void calc_direction(float from_x, float from_y, float to_x, float to_y, int * direction_x, int * direction_y);`
  Zoek de richtingsvectoren die aangeven waarheen het punt (to_x, to_y) ligt vanuit (from_x, from_y). De richtingvectoren worden meegegeven als pointer zodat de functie deze kan aanpassen.
  De richtingsvectoren hebben de waarde -1, 0 of 1. Ze slaan respectievelijk op terugkeren via de as, stilstaan of de as volgen. Bijvoorbeeld: direction_x 1 en direction_y 0 betekent naar rechts op het scherm (richting oost). (zie 7.2 Richtingsvectoren)

- `float find_render_angle(float alpha_radians, int direction_x, int direction_y);`
  Zoek vertrekkende van de hoek alpha en de richtingsvectoren die aangeven in welk kwadrant de hoek zich bevindt, de hoek waaronder de sprite van de Entity moet gerenderd worden. Hoe je dit doet, vind je terug in 3.5.4 Render hoek versus alpha.

- `int in_world_screen(float screen_x, float screen_y);`
  Deze functie is reeds geïmplementeerd voor jullie en mag dus onveranderd blijven. Ze wordt al gebruikt in de geïmplementeerde stukken uit de mouse_move en mouse_down functies van game.c.
  Ze wordt gebruikt om te kijken of de opgegeven coördinaat binnen het World speelveld valt of niet.

5 Opgegeven bestanden

[*] = Volledig geïmplementeerd, mag in aangepast worden, indien gewenst.
[!] = Volledig geïmplementeerd, niet in aanpassen.
[ ] = Moet verder geïmplementeerd worden door jullie.

[*] config.h: bevat alle #defines in het spel
[!] entities.h
[ ] entities.c
[!] game.h
[ ] game.c
[!] gui.h
[!] gui.c
[!] hash_set.h
[ ] hash_set.c
[!] hud.h
[ ] hud.c
[*] main.c
[!] pathfinding.h
[ ] pathfinding.c
[!] priority_queue.h
[ ] priority_queue.c
[!] render.h
[ ] render.c
[*] test_functions.h
[*] test_functions.c
[!] tower_ai.h
[ ] tower_ai.c
[!] util.h
[ ] util.c
[!] world.h
[ ] world.c

6 Concrete opgave

Implementeer het spelletje Tower Defense zoals omschreven in dit document. Afwijkingen van wat voorgesteld wordt, eigen toegevoegde functionaliteiten en ontwerpsbeslissingen schrijf je in een verslag document dat je mee indient.

*Dit project moet met **2 personen** gemaakt worden!*

7 Tips

7.1 Tijd meten

Als je tijd wil meten binnen de game-loop kan je dit op een eenvoudige manier doen. Je houdt een teller bij die incrementeert telkens de functie aangeroepen wordt. Dit zal op een consequente manier gebeuren binnen de game-loop, namelijk net zo veel keer in een seconde als de FPS waarde op is ingesteld.
Een eenvoudig voorbeeld. We willen maar om de 5 seconden iets doen. En ons spel loopt aan 60 FPS.

```
teller++;

if (teller > (5*60)) {
    teller = 0;
    do_action();
}
```

7.2 Richtingsvectoren

Een richtingsvector kan wordt gebruikt om te weten in welke zin een beweging gebeurt langs een as. Als we bijvoorbeeld naar rechts bewegen op de x-as (oplopende kant) dan is de vector 1. Indien we naar links bewegen (afnemende kant) dan is de vector -1. Indien we blijven stil staan op de x-as is de vector 0. Hetzelfde geldt voor de y-as. Op deze manier kunnen twee richtingsvectoren altijd een richting aanduiden.

direction_x	direction_y	richting vanuit oorsprong
0	-1	noord
1	-1	noordoost
1	0	oost
1	1	zuidoost
0	1	zuid
-1	1	zuidwest
-1	0	west
-1	-1	noordwest

Tabel 2: Richtingsvectoren en hun respectievelijke richting

Een typisch gebruik van de richtings vectoren is deze vervolgens te vermenigvuldigen met de speed in die bepaalde richting (x of y) en dat dan bij de respectievelijke coördinaat te tellen, om te bewegen.

7.3 De game-loop en renderen

Wees je ervan bewust hoe een spel en dus ook dit spel omgaat met state en renderen. Het is dus de bedoeling dat gedurende heel de game-loop de state van het spel aangepast wordt. Dit betekent alle velden worden aangepast, coördinaten van Entities worden aangepast, informatie over wat er zal moeten staan onder de cursor wordt klaar gezet, enz. **Alles wordt klaar gezet, maar nog niets wordt al getekend!** Enkel op het einde van de loop, aangeroepen door de render_game functie, wordt dan de gehele state omgezet naar effectieve rendering naar het scherm. Alles wat voorheen klaargezet en aangepast was, wordt nu bekeken en uitgelezen en wordt gebruikt om de juiste dingen naar het scherm te tekenen.

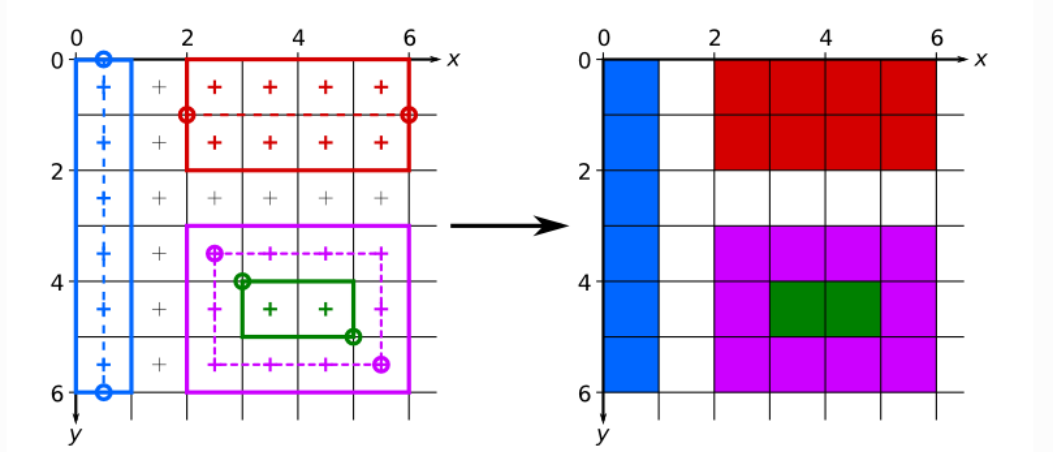
7.4 Pathfinding bij plaatsing Tower

Er wordt gevraagd om bij het plaatsen van een toren te controleren of deze niet het allerlaatste pad van spawn naar castle blokkeert. Dit is simpel te doen, door de toren op het speelveld te zetten, en vervolgens te kijken of je nog een pad terugvindt van spawn naar castle. (onthoud: zolang we render_game nog niet opnieuw aangeroepen hebben, is dit toch niet zichtbaar op het scherm). Vergeet niet na deze controle, als blijkt dat de Tower daar niet mag staan wegens geen pad meer, de Tower weer correct van de World te verwijderen!

7.5 Werking draw functies

De draw functies gebruiken achterliggende de Allegro bibliotheek. Deze heeft een eigen manier waarop een lijn van punt A naar punt B wordt geïnterpreteerd. In Figuur 13 zien we hoe dit werkt.

```
/* Blue vertical line */
draw_line(color_blue, 0.5, 0, 0.5, 6, 1);
/* Red horizontal line */
draw_line(color_red, 2, 1, 6, 1, 2);
/* Purple and green rectangle do not matter in this example */
```



Figuur 13: Line drawing interpretatie

7.6 Sprites

Volgende sprites zijn ter beschikking om te tekenen:

- [ ] `SPRITE_WORLD`: de volledige World achtergrond.
- [*] `SPRITE_SPAWN`: de spawn locatie.
- [*] `SPRITE_CASTLE`: de castle locatie.
- [ ] `SPRITE_GRASS`: gras achtergrond van een tile.
- [ ] `SPRITE_TOWER_MACHINE_GUN`: machine gun Tower.
- [ ] `SPRITE_TOWER_ROCKET_LAUNCHER`: rocket launcher tower.
- [ ] `SPRITE_TOWER_FLAK_CANNON`: flak cannon tower.
- [ ] `SPRITE_DESTROY_TOWER`: afbeelding voor de Button Destroy tower.
- [ ] `SPRITE_MOUTAIN`: berg tile.
- [ ] `SPRITE_WATER`: water tile.
- [ ] `SPRITE_UI`: het volledig ui frame (achtergrond).
- [*] `SPRITE_NORMAL`: normale Enemy.
- [*] `SPRITE_ELITE`: elite Enemy.
- [*] `SPRITE_FAST`: snelle Enemy.
- [*] `SPRITE_AIR`: vliegende Enemy.
- [*] `SPRITE_BOSS`: baas Enemy.
- [ ] `SPRITE_BUTTON_BLANCO_UP`: afbeelding van Button.
- [ ] `SPRITE_BUTTON_BLANCO_DOWN`: afbeelding van ingedrukte Button.
- [ ] `SPRITE_BUTTON_BLANCO_HOVER`: afbeelding van gehighlighte Button.
- [ ] `SPRITE_AMMO_BULLET`: machine gun kogel.
- [ ] `SPRITE_AMMO_ROCKET`: rocket launcher raket.
- [ ] `SPRITE_AMMO_FLAK`: flak cannon kogel.
- [ ] `SPRITE_SPELL_FREEZE`: afbeelding voor de Button Frost Wave.
- [ ] `SPRITE_SPELL_POISON`: afbeelding voor de Button Poison Gas.

Alle sprites met een sterretje `[*]` voor zijn geanimeerde sprites en kunnen mits correcte initialisatie van hun FrameAnimator getekend worden met de draw_sprite_animated functie.

7.7 Losse tips

- Let op de hoek alpha bij targetting werkt met radialen, omdat ook de sinus en cosinus functie met radialen werken.
- De angle die je meegeeft bij het tekenen van een entity, verwacht ook dat deze in radialen staat.
- Er is een constante PI gedefinieerd in config.h. Gebruik steeds deze, anders kom je in de problemen, omdat de gui code van deze PI uit gaat.
- Het gebruik van de standaard bibliotheken als math.h, stdio.h, stdlib.h en string.h zijn toegestaan.
- Het gebruik van externe bibliotheken is niet toegestaan.

7.8 Algemeen

- Respecteer de signatuur van de opgave. Het aanpassen van de bestaande inhoud van de header-bestanden is NIET toegestaan!
- **Begin niet onmiddellijk te programmeren. Lees eerst aandachtig de opgave en denk na hoe je de verschillende problemen zou oplossen.**
- Alle opmerkingen over de practica gelden ook hier: wees zuinig met geheugen, let op voor dangling pointers, **geef alles wat gealloceerd wordt ook weer vrij**, controleer of je open bestanden wel correct geopend zijn en gesloten worden, bescherm je header files…
- Er is een tool in Visual Studio om memoryleaks op te sporen, gebruik die dan ook!
- Typisch wordt begonnen met een eenvoudige basisversie van het uiteindelijk spel. Deze versie moet probleemloos kunnen werken, daarna kan dan geleidelijk de *game loop* verder uitgebreid worden met nieuwe aspecten. Dit gebeurt gemakkelijkst door te itereren over het volgende stappenplan:
  1. Bepaal het doel van deze iteratie
     - Zorg dat dit een kleine toevoeging is ten opzichte van de vorige versie van je spel.
  2. Analyseer de mogelijke manieren om dit nieuwe doel te bereiken.
     - Misschien zijn er verschillende manieren om een feature te implementeren?
  3. Ontwikkelen en testen
     - Ontwikkel de nieuwe feature zoals in de vorige stap beslist.
     - Voer het spel uit en kijk of de *game loop* nog altijd mooi en foutloos doorlopen wordt.
  4. Plan de volgende iteratie
     - Plan de volgende mogelijke uitbreiding van het spel.
- 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.
- Als referentiecompiler wordt Microsoft Visual C++ Express Edition gebruikt. Zorg er dus voor dat je project daarmee compileert en uitvoerbaar is!
- Probeer zoveel mogelijk compiler warnings te vermijden; deze duiden meestal op fouten die Visual Studio C++ EE voor jou zal oplossen, maar die je evengoed kan vermijden door je code aan te passen.
- Vanzelfsprekend zijn er zaken waar deze opgave je vrij in laat; deze mag je zelf kiezen naar eigen inzicht. Verduidelijk in alle geval je broncode met commentaar waar dat nuttig is! Neem belangrijke zaken ook zeker op in het verslag (zie 8).

8 Indienen

- Het project wordt gemaakt in groepen van 2 studenten. **Het is verplicht hiervoor een groep op minerva te kiezen**!
    - Schrijf een kort verslag (max. 2 bladzijden) met daarin:
    - De naam, voornaam en richting van de groepsleden + groepsnummer van minerva
    - Een korte uitleg bij de belangrijkste ontwerpbeslissingen, bv. de manier waarop vijanden gespawned worden.
    - De taakverdeling: wie heeft wat gedaan?
- Zip je broncode bestanden (header files en implementatiebestanden) samen met het verslag in een bestand naam_voornaam_project_nr.zip. De naam en voornaam zijn die van degene die het project indient. Het nummer is je groepsnummer van op minerva.
- Gebruik een standaard zip formaat, geen rar of 7zip.
- Zet in je bronbestanden bovenaan in commentaar de namen van de groepsleden en de naam van het bestand + groepsnummer.
- Stuur je oplossing door middel van de dropbox op minerva door naar **Bruno Volckaert**, ten laatste op **zondag 9 december, 23u59**.

9 Appendix

9.1 Gui.h

Omdat ANSI C naast het gebruik van de console geen mogelijkheden biedt voor visualisatie wordt er een GUI header en implementatie file opgegeven die jullie zal toelaten om het spel grafisch weer te geven.

GUI.h specificeert de volgende functies:

- `int init_gui(int width, int height);`
  Deze functie initialiseert de gui volgens de gegeven width en height (gebruik de opgegeven width en height in de config.h constanten). Ook worden een aantal allegro systemen geïnitialiseerd.

- `void init_game_loop(int fps);`
  Deze functie initialiseert de gameloop en zijn timer. Het aantal fps dat meegeven wordt bepaalt hoe vaak de lus doorlopen wordt in één seconde. (gebruik ook hier best de meegegeven waarde in de config.h constanten).

- `void cleanup_game_loop(void);`
  Ruimt al het gereserveerde geheugen op en deïnitialiseert een aantal allegro systemen.

- `void draw_sprite(SPRITE_TYPE type, float screen_x, float screen_y, float angle_radians);`
  Tekent de opgegeven sprite naar de backbuffer (= offscreen bitmap) met de linker bovenhoek op positie (screen_x, screen_y) onder een hoek van angle_radians rad.

- `void draw_sprite_animated(SPRITE_TYPE type, FrameAnimator * animator, float screen_x, float screen_y, float angle_radians);`
  Tekent de opgegeven sprite naar de backbuffer (= offscreen bitmap) met de linker bovenhoek op positie (screen_x, screen_y) onder een hoek van angle_radians rad. Door de meegegeven FrameAnimator, wordt automatisch op het juist moment, het juist frame ingeladen om de animatie te laten afspelen op het scherm.

- `void draw_line(Color color, float screen_sx, float screen_sy, float screen_dx, float screen_dy, float thickness);`
  Tekent een lijn tussen (screen_sx, screen_sy) en (screen_dx, screen_dy) met een zekere thickness in een gevraagde color (zie ook 7.5 Werking draw functies) naar de backbuffer.

- `void draw_rectangle(float screen_sx, float screen_sy, float screen_dx, float screen_dy, Color color);`
  Tekent een gevulde rechthoek met als linker bovenhoek (screen_sx, screen_sy) en als rechter onderhoek (screen_dx, screen_dy) in een gevraagde color naar de backbuffer.

- `void draw_circle(float screen_x, float screen_y, float r, Color color);`
  Tekent een gevulde cirkel met als middelpunt (screen_x, screen_y) en als straal r in een gevraagde color naar de backbuffer.

- `void draw_triangle(float screen_x1, float screen_y1, float screen_x2, float screen_y2, float screen_x3, float screen_y3, Color color);`
  Tekent een gevulde driehoek met als hoekpunten (screen_x1, screen_y1), (screen_x2, screen_y2) en (screen_x3, screen_y3) in een gevraagde color naar de backbuffer.

- `void draw_text(char* txt, Font font, Color color, float screen_x, float screen_y, ALIGN align);`
  Tekent een tekst in een bepaald font (FONT_SMALL, FONT_MEDIUM, FONT_LARGE of FONT_HUGE) naar het scherm in een bepaalde color naar de backbuffer. (screen_x, screen_y) is de linker bovenhoek.

- `void set_transparency_on(void);`
  Deze functie zet transparantie aan. Deze moet aangeroepen worden voordat iets in een transparante kleur wordt getekend. (Color met a < 255). Als dit niet wordt gedaan, zullen onderliggende kleuren er raar uitzien.
  Opmerking: *sluit af met set_transparency_off().*

- `void set_transparency_off(void);`
  Deze functie zet transparantie af. Deze moet aangeroepen nadat iets in een transparante kleur werd getekend. (Color met a < 255). Als dit niet wordt gedaan, zal het spel er vanaf de volgende iteratie raar uitzien (onder andere tekst zal slechter weergegeven worden).

- `void clear_to_color(Color color);`
  Deze functie wist de gehele backbuffer bitmap naar een bepaalde color.

- `void flip_display();`
  Deze functie wisselt de backbuffer bitmap (off-screen) met de on-screen bitmap.

- `void start_drawing_world();`
  Deze functie moet aangeroepen worden alvorens de World te beginnen renderen voor de eerste keer. Deze zorgt ervoor dat alle render functies naar een aparte bitmap worden gestuurd. (zie ook 4.5 Rendering, bij functie render_world_to_sprite(World * world))
  Opmerking: *sluit af met stop_drawing_world().*

- `void stop_drawing_world();`
  Deze functie moet aangeroepen worden nadat de World voor de eerste keer is gerenderd. Deze functie zorgt ervoor dat de gemaakt bitmap opgeslaan wordt als sprite onder SPRITE_WORLD, zodat ze nadien kan gebruikt worden bij het normale renderen van de World.
  (zie ook 4.5 Rendering, bij functie render_world_to_sprite(World * world))

- `void wait_for_event(Event *ev);`
  Deze functie wacht op het volgende binnenkomende Event. Een Event moet meegeven worden als pointer, zodat het Event kan ingevuld worden.

- `int all_events_processed(void);`
  Deze functie geeft 1 terug als er geen Events meer wachten op dit moment om verwerkt te worden. Ander geeft ze 0 terug.

- `double get_current_fps(void);`
  Deze functie rekent de huidige frames per seconde uit en geeft ze terug.

- `void init_sprite_cache(void);`
  Deze functie initialiseert alle sprites door ze in te laten in een cache. Op deze manier moeten de sprites niet tijdens het spel van de harddisk ingelezen worden, maar staan zo op voorhand klaar in het geheugen.

- `void cleanup_sprite_cache(void);`
  Deze functie geeft al het geheugen ingenomen door de sprite cache terug vrij.

Deze functies worden achterliggend geïmplementeerd aan de hand van een library, die zal moeten gelinked worden aan jullie project (zie 9.2).

9.2 Allegro

Allegro is een cross-platform, open-source library die het makkelijker maakt om computerspellen en multimediaprogramma's te ontwikkelen. Het bundelt noodzakelijke functies voor dergelijke progamma's, zoals het afhandelen van gebruikers input, het tekenen naar het scherm en het afspelen van geluid, in een set van gemakkelijk te gebruiken APIs.

De versie van Allegro die wij zullen gebruiken (5.0.7) ondersteunt de volgende platformen:

- Unix/Linux
- Windows
- MacOS X
- iPhone

In de gekregen solution file is alles ingesteld voor gebruik van Allegro in Visual Studio. Voor de studenten die willen ontwikkelen in een andere omgeving verwijzen we naar de Allegro website: http://alleg.sourceforge.net/

9.3 Stand-alone executable builden

Op de volgende manier kan je een exe maken die stand-alone kan uitgevoerd worden:

1. Build het project in Release mode in plaats van Debug mode.
2. Ga naar je Solution folder en onder de Release folder vind je de exe.
3. Kopieer deze exe file naar een folder naar keuze (bv. C:\TD).
4. Kopieer de map assets in de Project folder ook naar C:\TD.
5. Kopieer de allegro file allegro-5.0.7-monolith-md.dll uit allegro/bin ook naar C:\TD
6. Normaal gezien werkt de exe nu.

Je kan deze C:\TD folder bijvoorbeeld zippen en dan kan je je executable met de nodige assets en library op een andere pc zetten.