Software-ontwikkeling 1

Academiejaar 2015 – 2016

SOI@intec.ugent.be

Project: Strategic War

Inhoudsopgave

1 Inleiding

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 achtergrond kennis te hebben voordat er aan de eigenlijke opgave begonnen wordt. Er wordt dan ook van uitgegaan dat deze introductie (1.2 & 1.3) volledig doorgenomen is alvorens aan het project te beginnen.

1.1 Algemene project informatie

Dit project loopt gedurende heel het semester en moet in een **groep van 4 personen** uitgevoerd worden. Het project heeft als focus om zo veel mogelijk van de aspecten van software development, zoals dit de dag van vandaag gaat, aan het licht te laten komen. Hierbij denken we behalve het programmeren zelf, aan volgende aspecten: collaboration & tooling, testing, integration of libraries, reading documentation and API's, designing & implementing algorithms, etc.

Gedurende de ontwikkeling heeft elke groep een assistent die jullie begeleidt. Bij vragen of problemen kan deze assistent jullie verder helpen. Ook zal af en toe een kleine meeting met deze assistent nodig zijn met de ganse groep.

Het einddoel van dit project is een werkende C++ implementatie van een spel af te leveren. Er is echter ook een deadline halverwege, waarin onder andere een level-editor moet afgeleverd worden in C. We noemen het spel Strategic War, maar het is los gebaseerd op de spelregels uit de "Advance Wars" game reeks [i]. Uiteraard zullen de spelregels niet allemaal overgenomen worden, en betreft het een eenvoudigere implementatie. Vooral het turn-based aspect waarin de speler en de computer elk om de beurt hun zetten mogen doen, wordt overgenomen. Aan welke voorwaarden het spel precies moet voldoen, zal in de volgende secties volledig uit de doeken gedaan worden.

Lees ook zeker de sectie over opzetten van github en indienen (0 & 8), want jullie vooruitgang wordt in het oog gehouden via een github-space van UGent. We gaan er immers ervan uit dat alle collaboratie en documentatie online gebeurt!

Besteed genoeg tijd en aandacht aan dit project, want het gaat om 5 van de 20 punten voor het vak Software Ontwikkeling I die je kan winnen, maar dus ook verliezen.

1.2 Game programming

Spellen programmeren is op een aantal vlakken anders dan een gewone applicatie programmeren. In deze sectie worden een aantal basis aspecten van game development uit de doeken gedaan. Lees dit eens goed door zodat je een beter beeld hebt van hoe dit er normaal gezien aan toe gaat.

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 1.2.3) opgelost, de beelden getekend en de geluiden afgespeeld.

De game loop kan verder verfijnd en aangepast worden terwijl de ontwikkeling van het spel voort gaat, 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.

[1] Artificial Intelligence

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

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 botsingen 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 tweedimensionale variant.

1.3 Entity system framework

Uit ervaring hebben we geleerd dat een klassieke objectgeoriënteerde aanpak niet altijd de beste resultaten oplevert bij het ontwerpen van een game. Abstracties die het ene moment logisch lijken, kunnen later een hindernis vormen bij het toevoegen van nieuwe functionaliteit. Voorbeelden hiervan zijn:

- Tekenen naar het scherm gebeurt door elke klasse een render-methode te laten implementeren, maar dit zorgt ervoor dat dit aspect van het spel verspreid wordt over de volledige codebase, wat het moeilijker onderhoudbaar en uitbreidbaar maakt.
- Generieke functionaliteit wordt geïmplementeerd in klassen waarvan de objecten die dit gebruiken kunnen overerven, bv. een speler erft over van de klasse InputController om te kunnen reageren op input. Maar wat als het gedrag at runtime plotseling moet veranderen op basis van de toestand? Bv. de speler dient tijdelijk te worden bestuurd door de AI voor een bepaalde cutscene (animatie-filmpje tussenin).

Bovenstaande problematiek wordt ook erkend in de game industrie en één van de voorgestelde oplossingen is het gebruik van een entity system framework. Bij deze data gedreven aanpak worden de verschillende game-objecten niet meer voorgesteld als afzonderlijke klassen, maar als generieke *entities* die samengesteld worden aan de hand van een bepaald aantal componenten. Deze componenten bevatten heel specifieke data die kunnen gekoppeld worden aan een game-object (bv. de positie).

Entities en hun bijhorende componenten zijn dus pure data containers en implementeren geen functionaliteit of gedrag. Dat is de verantwoordelijkheid van de verschillende entity systems die zullen inwerken op de geregistreerde entities. Voorbeelden hiervan zijn een systeem om bewegingsvectoren toe te passen en een systeem dat entities op het scherm kan tekenen.

Voor meer details en het precieze gebruik van een entity system framework, verwijzen we naar volgende artikels:

- http://www.richardlord.net/blog/what-is-an-entity-framework
- http://www.richardlord.net/blog/why-use-an-entity-framework

2 Opgave

De effectieve opgave bestaat uit een aantal aspecten waarover we hieronder apart even uitleg geven.

2.1 Level editor en pathfinding (C)

Jullie moeten een editor programmeren in C die je toelaat om een level te tekenen en opslaan die dan later kan gebruikt worden in het Strategic War spel. Omdat de editor aan bepaalde voorwaarden moet voldoen, zal er pathfinding[2] moeten geïmplementeerd worden ook. Deze pathfinding zal dan later in C++ geport[3] worden, zodat ook daar pathfinding kan gebruikt worden, maar dan voor het volledig spel.

In dit onderdeel van het project zal er ook iemand zich moeten bezighouden met testing. Het op voorhand opstellen van testfuncties die een afgesproken interface gebruiken om code te testen, kan bij het ontwikkelen van de pathfinding zeer belangrijk zijn om kleine foutjes in randvoorwaarden op tijd te detecteren.

Verder informatie om jullie op weg te helpen in dit eerste deel, vind je in hoofdstuk 3.

2.2 Spel (C++)

Het uiteindelijke doel van het project is om een variant op het spel “Advance Wars” te ontwikkelen in C++. In “Strategic Wars” wordt initieel een speelveld ingeladen. Op dit speelveld zijn een aantal obstakels aanwezig en een hoofdkwartier voor elke speler. Ook krijgt elke speler bij het begin van het spel een aantal troepen (=units). Een voorbeeld van het originele spel vind je hier https://youtu.be/RSLSlJPX5LM.

Spelers mogen elk om beurt hun zetten doen, waarna ze de beurt overdragen naar de tegenspeler. Een speelbeurt bestaat uit het selecteren van een unit, deze bewegen op het speelveld of laten aanvallen, tot er geen Action Points (=AP) meer over zijn voor deze unit. Daarna kan de volgende unit die nog APs over heeft bewogen worden/aanvallen. Eens alle units hun APs hebben verdaan (of de speler het beslist), wordt de beurt beëindigd en is het aan de andere speler.

In jullie implementatie zijn er 2 spelers: de menselijke speler en een computergestuurde tegenspeler. Er zal dus een zekere vorm van intelligentie ingebouwd moeten worden.

Winnen doe je wanneer je aan één van de mogelijke win-condities voldoet:

- Alle units van de tegenspeler zijn vernietigd.
- Het hoofdkwartier van de tegenspeler is vernietigd.

[2] Het zoeken van het (kortste) pad van punt A naar B.
[3] Herschrijven in een andere programmeertaal.

Uiteraard is er heel wat meer te vertellen over al deze aspecten, maar dit kort overzicht geeft alvast een houvast over hoe een speelsessie opgebouwd is.

2.3 Deadlines

Het project heeft 2 grote deadlines. De eerste voor het afleveren van het C gedeelte, de tweede voor het afleveren van het C++ gedeelte.

- Deadline C (LevelEditor/Pathfinding)
  - Zondag 25 oktober 23:59
  - Laatste versie voor deze tijd wordt automatisch van github gehaald
- Deadline C++ (Volledig spel)
  - Zondag 6 december 23:59
  - Laatste versie voor deze tijd wordt automatisch van github gehaald
  - Document op dropbox indienen (zie 8)

2.4 Puntenverdeling

Het project geldt voor 5 punten van het examen. Deze 5 punten kunnen als volgt verdiend worden:

- 2,5 punten op het C gedeelte
  - 1,75 punten voor de implementatie van de basis-level editor
  - 0,75 punten voor implementatie van 3 uitbreidingen (de *bugfix* uitbreiding telt hierbij niet mee als een volwaardige uitbreiding)
    (vrij te kiezen uit de lijst van uitbreidingen behalve de bug fix) (*)
- 2,5 punten op het C++ gedeelte.

(*) Het staat een groep vrij om méér dan 3 uitbreidingen te implementeren, dan quoteren we de 3 best geïmplementeerde uitbreidingen.

2.5 Allegro library

Bij zowel de editor als het volledige spel zal gebruik gemaakt worden van de allegro library. Deze library zal het eenvoudiger maken om vanuit C of C++ naar het scherm te tekenen.

Waar in het C gedeelte deze library bijna rechtstreeks zal worden aangeroepen, zal dit wat meer geabstraheerd worden in het C++ gedeelte. Een stuk wordt daar voor jullie afgeschermd via de AllegroLib klasse. Dit betreft alles in verband met initialisatie van allegro, input opzetten, timing en events (de code eens bekijken en proberen snappen, kan zeker geen kwaad). Het tekenen zelf moeten jullie wel implementeren, maar dit zal in een aparte wrapper rond allegro gebeuren, die jullie zelf dienen te schrijven. (zie 4.5.7).

Een deel van software ontwikkeling is het integreren en gebruiken van libraries. Om meer informatie over de API en documentatie omtrent de allegro library (5.0.10) te vinden gebruik je volgende URLs:

- http://liballeg.org/a5docs/5.0.10/
- http://liballeg.org/
- https://www.allegro.cc/ (community website)

3 Implementatie Level Editor (C)

3.1 Algemeen

De bedoeling van deel 1 van het project is om een level editor te schrijven. Deze moet in C geschreven worden, en is onafhankelijk van het eigenlijke spel dat in deel 2 van het project in C++ geschreven zal worden.

De C11 standaard wordt gebruikt in dit deel van het project. Dat betekent o.a. dat initialisatie van variabelen niet vooraan in een functie moet gebeuren. Dit is voordelig voor de leesbaarheid van de code.

De editor laat toe om spel levels uit bestanden te lezen, ze te bewerken, en ze opnieuw op te slaan. Het spel uit deel 2 van het project zal deze levels kunnen inlezen en gebruiken.

Figuur 1: Level Editor GUI

3.2 Details level editor GUI

Elk level bestaat uit een matrix van “tegels” (“Cell” in de opgave code). De level editor gaat uit van een vaste grootte voor de levels (25 tegels breed en 12 hoog). Het huidige level wordt centraal op het scherm getoond. Links van het level kan de gebruiker een type tegel kiezen. Van sommige types tegels zijn er 2 versies, namelijk elke speler een versie (zie kleur linksboven tegel). Als de gebruiker op het level klikt, zal de tegel onder de muis vervangen worden door de tegel die links geselecteerd is.

Met de 3 knoppen bovenaan kan de gebruiker het level opslaan, een level inlezen, of het huidige level leeg maken.

De level editor geeft onder het level boodschappen weer wanneer een level niet geldig is.

In een geldig level heeft elke speler een hoofdkwartier, en is er een pad tussen de hoofdkwartieren dat minstens 100 eenheden lang is (Later meer over de berekening van deze afstand). Indien aan één of meerdere van deze voorwaarden niet voldaan is, wordt er een tekstboodschap getoond die het probleem meldt.

Hoofdkwartieren worden door de editor speciaal behandeld. Ze zijn altijd 4 tegels groot, en er kan per speler slechts 1 hoofdkwartier zijn. Als er door de gebruiker een nieuw hoofdkwartier geplaatst wordt, moet het oude automatisch verwijderd worden. Het nieuwe moet ook onmiddellijk 4 tegels groot zijn. De editor garandeert ook dat een hoofdkwartier altijd uit 4 tegels bestaat. Als er een tegel van het hoofdkwartier wordt aangepast waardoor dit niet meer zo is, moet het hoofdkwartier verwijderd worden.

Als er een pad mogelijk is tussen de 2 hoofdkwartieren, wordt het kortste pad weergegeven op het scherm (zie de groene voeten die het pad weergeven tussen de twee hoofdkwartieren in Figuur 1).

3.3 Functionele blokken

3.3.1 Common

common.h definieert gemeenschappelijke structs en functies die door de meeste andere bestanden gebruikt worden.

- De struct *CellType* bevat de verschillende mogelijke types die een tegel kan zijn. De laatste waarde, CELL_TYPE_COUNT, is echter geen type, dit is een waarde die indien ze naar int gecast wordt, het aantal types terug geeft. Deze truc wordt dikwijls gebruikt bij enums in C. Lees voor meer uitleg het commentaar in de header file zelf.
- De struct ***Owner*** bevat de verschillende mogelijke “eigenaars” van een tegel. Vele tegels hebben geen eigenaar (o.a. lege grond), dus ook dat is een geldige waarde.
- De functie *cell_type_is_unit* geeft de waarde “waar” terug, als het opgegeven *cell_type* een door de speler of AI beweegbare eenheid voorstelt (Hoofdkwartieren geven dus *false* terug.)
- De functie *cell_type_is_player_owned* geeft de waarde “waar” terug, als het opgegeven *cell_type* altijd in het bezit van de speler of de AI moet zijn. Dit is waar voor alle eenheden en hoofdkwartieren. Dit is niet waar voor neutrale tegeltypes, zoals grond en water.
- De struct *Pos* wordt gebruikt om de positie van een tegel bij te houden. Deze positie wordt bijgehouden aan de hand van kolom- en rijnummer van de tegel. Belangrijk: de rijen zijn genummerd van boven naar onder, de kolommen van links naar rechts.
- De functie *pos_init* is een handige kleine functie, die een *Pos* struct teruggeeft (via “pass by value”) met een opgegeven kolom- en rijwaarde.
- De struct *Path* stelt een pad voor, dat uit verschillende stappen bevat. Elke stap is een positie. De posities worden bijgehouden in een rij waar de pointer *steps* naar verwijst.

Het aantal stappen wordt bijgehouden in *step_count*. De afstand wordt bijgehouden in *distance*.

- De functie *path_alloc* geeft een pointer terug naar een nieuw gealloceerd stuk geheugen waar zich een *Path* struct bevindt. De pointer *steps* is bovendien geïnitialiseerd, en verwijst naar een rij waar plaats is voor minstens *step_count Pos* structs. De *step_count* en *distance* in de *Path* struct worden gedigitaliseerd volgens de opgegeven waarden.
- De functie *path_free* geeft het geheugen waar de opgegeven pointer naar een *Path* struct naar verwijst vrij, en zorgt dat ook het geheugen waar *step* binnen dat *Path* naar verwijst vrijgegeven wordt.

Opgave: de functies die in header file common.h gedefinieerd worden, moeten worden geïmplementeerd in common.c. Bovendien moeten deze functies getest worden. Voor functies zoals *path_alloc* en *path_free* moet enkel getest worden wat eenvoudig getest kan worden. Zo kan er maar beperkt getest worden of alloceren en vrijgegeven van geheugen correct werkt (je kan enkel bij het alloceren nakijken of er een NULL pointer is).

3.3.2 Level

De levels van het spel bestaan uit tegels ("Cells" in de opgave code). Levels hebben een vaste grootte van 25 tegels breed en 12 hoog. Elke tegel heeft een type en (mogelijk) een eigenaar. Het level wordt bijgehouden met behulp van de structs *Cell* en *Level*.

Er zijn verschillende functies gedefinieerd in level.h, die in level.c geïmplementeerd moeten worden. Levels worden met behulp van de funties *level_write_to_file* en *level_alloc_read_from_file* van en naar bestanden omgezet. Hieronder vind je een beschrijving van het bestandsformaat.

Een level wordt in tekstueel formaat opgeslaan. Elke lijn wordt afgesloten met een "newline" symbool (ASCII code 10, symbool '\n' in C). De eerste lijn bevat de dimensies van het level. Eerst het aantal kolommen, dan een '|' symbool, en vervolgens het aantal rijen. Aangezien deze dimensies vast zijn, is de eerste lijn dus: "25|12\n". De volgende lijnen bevatten het eigenlijke level. Per tegel geeft 1 karakter het type en de eigenaar weer (zie onderstaande tabel). De eerste lijn karakters komt overeen met de eerste rij tegels. Het 2de karakter geeft dus het type en de eigenaar weer van de tegel op rij 0 kolom 1.

Mapping tussen karakters, types en eigenaars:

Karakter	Tegel Type	Eigenaar
*	Lege grond	OWNER_NONE
W	Water	OWNER_NONE
B	Brug	OWNER_NONE
R	Rots	OWNER_NONE
h	Hoofdkwartier	OWNER_HUMAN
H	Hoofdkwartier	OWNER_AI

1	Eenheid type 1	OWNER_HUMAN
2	Eenheid type 2	OWNER_HUMAN
3	Eenheid type 3	OWNER_HUMAN
7	Eenheid type 1	OWNER_AI
8	Eenheid type 2	OWNER_AI
9	Eenheid type 3	OWNER_AI

Ter illustratie een voorbeeld level bestand:

```
25|12
****************WW*******
************WWWW********
*hh***R******W*****R*****
*hh**********W***********
************W********9**
*1*********WBW********8**
*2*********WBW********7**
*3***R*****W*************
**********W*****R****HH*
**********W**********HH*
********WWWW************
*******WW****************
```

Opgave: Implementeer in level.c de functies die gedeclareerd zijn in level.h en ontwikkel ook de nodige testen om correcte functionaliteit te kunnen aantonen.

3.3.3 Gui

De GUI interface is reeds voor jullie geschreven. Voor de basisopgave hoeven de files gui.h en gui.c in principe niet aangepast te worden.

De GUI kan gebruikt worden aan de hand van de volgende functies, die in gui.h gedefinieerd zijn:

- *gui_initialize* initialiseert de GUI. Het te gebruiken “theme” voor de levels is als parameter meegegeven.
- *gui_free* stopt de gui en geeft alle geheugen vrij.
- *gui_set_level* geeft aan welk level weergegeven moet worden. Merk op dat een pointer doorgegeven wordt. Als het level verandert, zal de GUI automatisch het gewijzigde level weergeven de volgende keer dat *gui_draw_frame* opgeroepen wordt.
- *gui_draw_frame* update de weergave van de GUI.
- *gui_set_level_highlight* toont een selectiekader rond een bepaalde tegel in het level. Dit is typisch de tegel waarboven de muis zich bevindt. Als er geen kader getekend moet worden, kan deze functie aangeroepen worden met ongeldige coördinaten (i.e buiten de

geldige rij- en kolomwaarden voor het level, dus coördinaten met negatieve getallen, of met getallen groter dan de dimensies).

- *gui_set_build_highlight*: tekent een selectiekader rond een bepaalde tegel in de linkse lijst met tegels in de GUI. Om geen selectiekader rond een tegel te zetten, kan een ongeldige combinatie van *cell_type* en *owner* opgegeven worden.
- *gui_add_message*: voegt een tekstboodschap toe aan de GUI. Deze functie neemt 1 of meerdere argumenten, op exact dezelfde manier als de "*printf*" functie. Het eerste argument is een format string in het formaat dat ook *printf* verwacht, de eventueel volgende argumenten komen overeen de variabelen in die format string (%s %d etc). Er worden maximaal 5 boodschappen tegelijk weergegeven, daarna wordt telkens de oudste boodschap verwijderd.
- *gui_clear_messages*: verwijdert alle tekstboodschappen in de GUI.
- *gui_show_path*: toont een bepaald pad in de GUI. Om geen pad weer te geven, kan NULL aan deze functie meegegeven worden.
- *gui_get_next_game_event*: deze functie wacht tot het volgende game event binnen komt, en geeft dat vervolgens terug. Meer uitleg over events kan verderop gevonden worden.
- *gui_show_save_file_dialog*: deze functie maakt een dialoog die de user toelaat te kiezen waar een bestand op te slaan. De functie wacht tot de gebruiker een keuze gemaakt heeft en geeft vervolgens de naam van het bestand terug. Als de gebruiker geen bestand koos, wordt NULL terug gegeven.
- *gui_show_load_file_dialog*: Bijna identiek aan *gui_show_save_file_dialog*, maar deze versie wordt gebruikt om een bestand om in te laden te kiezen.

De GUI maakt gebruik van events. De header file event.h definieert hiervoor de nodige structs. Er zijn verschillende types events die kunnen optreden. Om te beginnen is er een *TimerEvent*, dat periodiek opgeroepen wordt. Elke keer dit timer event voor komt, moet het scherm getekend worden. Er is ook een *LevelEvent*, dat voorkomt als de gebruiker de muis beweegt, of met de muis klikt, op het deel van de GUI dat het level weer geeft. Dit event bevat informatie over de tegel waar de muis zich bevindt, en of er met de muis geklikt is of enkel bewogen. Het *BuildSelectorEvent* komt voor als de gebruiker de muis beweegt, of met de muis klikt, op het links deel van de GUI dat alle mogelijke tegeltypes oplijst. Dit event bevat informatie over het tegeltype en de eigenaar van het keuzevak waar de muis op staat, en of er met de muis geklikt is of enkel bewogen. Ten slotte is er het *ButtonEvent* , dat voor komt als er op een GUI knop geklikt wordt door de gebruiker. Ook als de GUI afgesloten wordt komt dit event voor. Het event bevat info over op welke knop er geklikt is door de gebruiker.

De functie *gui_get_next_game_event* uit gui.h wacht op het volgende event, en geeft dat dan terug. Er is voor elk type event een bijhorende struct gedefinieerd in event.h. De functie *gui_get_next_game_event* verwacht geen van deze structs, maar verwacht een pointer naar een "union". Deze pointer moet naar een reeds gealloceerde event union verwijzen. Die union wordt aangepast naar gelang welk event voor komt.

De union voegt alle mogelijke event type structs samen, zodat die dus allemaal via 1 gezamenlijk type gebruikt kunnen worden. **Lees grondig het stuk over unions in de cursus om te bekijken hoe dit exact werkt.**

Merk op dat de union bovendien ook de enum "EventType type" bevat. Dit is strikt gezien niet nodig, aangezien elk van de event structs reeds start met deze enum. Maar net omdat elke struct dus start met de EventType enum, kunnen we, door deze enum ook in de union op te nemen, zorgen dat de EventType enum ook rechtstreeks beschikbaar is in de union zonder eerst een bepaald event struct type op te kiezen. Een voorbeeld:

```
Event event;

gui_get_next_game_event (&event);

//moest "EventType type" geen deel van de "Event" union zijn,

//dan moesten we het type zo achterhalen:

if (event.timer_event.type == EVENT_LEVEL)

    printf("Dit is een Level Event.");

//of even goed, zo:

if (event.button_event.type == EVENT_LEVEL)

    printf("Dit is een Level Event.");

//Maar omdat "EventType event" wél deel van de union is,

//kan het ook korter op deze manier:

if (event.type == EVENT_LEVEL)

    printf("Dit is een Level Event.");
```

Merk op dat in de 3 bovenstaande manieren om het type te vinden, altijd exact hetzelfde stuk geheugen wordt geïnspecteerd.

Opgave: er hoeft niets gewijzigd te worden in events.h, gui.h of gui.c

3.3.4 Editor

editor.c bevat de *main* functie van de level editor applicatie. Deze laadt een eerste level, en toont vervolgens de GUI. De "game loop" (zie 1.2.1) wordt gestart en staat in voor de weergave van de GUI, en voor de interactie. De editor GUI gebruikt een eenvoudige game loop, die gebruik maakt van hulp functies in gui.c, zoals *gui_get_next_game_event* en *gui_draw_frame*. De voornaamste taken van de game loop in editor.c zijn:

- Bij *TimerEvents* het scherm hertekenen.

- Bij *LevelEvents* een selectiekader rond de huidige tegel zetten, en bij een klik het level aanpassen.
- Bij *BuildSelectEvents* bijhouden en weergeven welk type tegel geselecteerd is.
- Bij *ButtonEvents* de bij de knop passende actie uitvoeren, zoals het opslaan of inlezen van levels.

Opgave: de main loop in editor.c moet uitgebreid worden om alle nodige functies van de editor te implementeren.

3.3.5 Path Finding

De editor kijkt na of de hoofdkwartieren van de 2 spelers bereikbaar zijn en of ze ver genoeg uit elkaar liggen. Hiervoor moet het kortste pad tussen de 2 hoofdkwartieren berekend worden. Deze functionaliteit wordt geïmplementeerd in de bestanden pathfinder.h en pathfinder.c

De afstand tussen 2 tegels wordt als volgt gedefinieerd:

- De afstand tussen 2 tegels die enkel 1 rij of een kolom verschillen, en dus slechts een horizontale of verticale stap vereisen, is 12
- De afstand tussen 2 tegels die zowel 1 rij als 1 kolom verschillen, en dus een diagonale stap vereisen, is 17

Om een pad te bepalen, maken we gebruik van een gerichte gewogen graaf. Met behulp van die graaf wordt dan het kortste pad gezocht. Het kortste pad algoritme kan vrij gekozen worden, een klassieke keuze is echter Dijkstra's algoritme. Het A* algoritme is een andere optie. Info over onder andere het Dijkstra's algoritme, kan je vinden in de cursus "Algoritmen en datastructuren I". Er is ook zeer veel informatie online te vinden. Beide algoritmen maken gebruik van een "priority queue". Er wordt in pqueue.h en pqueue.c een eenvoudige implementatie van een priority queue meegegeven. Merk op dat deze implementatie allesbehalve efficiënt is. Voor een eerste versie van deze level editor is dat echter geen probleem.

De graaf wordt opgebouwd en bewaard in een struct van het type "DWGraph". De declaratie hiervan moet aangevuld worden in dwgraph.h. De implementatie van de functies nodig om met deze graaf te werken (zelf te declareren in dwgraph.h) komt in dwgraph.c. Om de graaf te bouwen, wordt het level overlopen, worden van elke tegel alle mogelijke stappen bekeken, en wordt voor elke stap de overeenkomstige boog toegevoegd aan de graaf.

In de header file pathfinder.h is de functie *find_path* gedeclareerd. Deze neemt als argument een Level, een start tegel en een doel tegel. Deze functie geeft een *Path* struct terug die het korste pad bevat, of een NULL pointer als er geen pad is.

Opgave: De functionaliteit voor het zoeken van kortste paden moet geïmplementeerd worden. De implementatie begint met het implementeren van de functie *find_path* in pathfinder.c. Eventuele hulpfuncties nodig voor deze implementatie, moeten ook in pathfinder.h gedeclareerd worden, zodat ze ook getest kunnen worden. Ook zal een structuur nodig zijn om een directe

gewogen graaf voor te stellen. Hiervoor moet de declaratie van de struct *DWGraph* in dwgraph.h aangevuld worden (zowel struct als hulpfuncties), en moet de implementatie verzorgd worden in dwgraph.c. Ook hier moeten de nodige testen toegevoegd worden. De *pqueue* implementatie is reeds gegeven en hoeft niet gewijzigd te worden. Deze moet opnieuw wel getest worden.

3.4 Testen

Het bestand test_editor.c bevat alle testen voor de editor. Deze moeten worden aangevuld, om zo de functionaliteit van de editor te testen.

Voor het schrijven van testen wordt dikwijls van een test framework gebruik gemaakt, zeker in talen zoals Java. Ook voor C en C++ bestaan hiervoor frameworks. Als kennismaking met het schrijven van testen, maken we voor deze opgave gebruik van een zeer eenvoudig en minimalistisch test framework, gebaseerd op enkele eenvoudige preprocessor macro's.

In test_editor.c zijn ter illustratie reeds enkele tests gedefinieerd. Elke test is een afzonderlijke functie. Het return type van de functie is in dit framework steeds "char*": de functie zal een pointer naar een foutboodschap teruggeven als een test faalt. Als de test succesvol is, wordt de NULL pointer teruggegeven. Daarom staat er "*return NULL;*" op het einde van elke test functie. Elke test functie zal de waarheid van 1 of meerdere statements testen. Hiervoor moeten de testfuncties de functie[4] *mu_assert* gebruiken. Die veronderstelt dat de meegegeven uitdrukking *waar* is. Als dat niet het geval is, faalt de test.

Een goede test zal typisch slechts 1 functie testen, en slechts 1 bepaalde input waarde voor die functie testen. Uiteraard is het voor de leesbaarheid soms beter om van dit basisprincipe af te wijken, maar het is goed om elke test zo eenvoudig en minimalistisch mogelijk te houden.

De functie *all_tests()* roept alle test functies op via de functie *mu_run_test* en geeft daarna ook NULL terug. Voor het uitvoeren van alle testen wordt een afzonderlijke *main* functie gebruikt, die de functie *all_tests()* oproept.

Het gebruikte framework zal stoppen van zodra een test faalt. Complexere frameworks zullen steeds alle testen doen, en zullen ook typisch complexere "assert" functies voorzien. Een test toevoegen kan door een functie die een *char** terug geeft toe te voegen, en die in *all_tests()* op te roepen via *mu_run_test*.

[4] *mu_assert* en *mu_run_test* zijn preprocessor macro's. Dat betekent dat het geen functies zijn die opgeroepen worden, maar dat er code gegenereerd en ingevoegd wordt waar *mu_assert* en *mu_run_test* gebruikt worden. Deze gegenereerde code bevat o.a. een "return", die gebruikt wordt als een assert of test faalt. Daardoor wordt de "return 0" onderaan de test functies en de *test_all()* functie dus enkel opgeroepen als de test succesvol is. Merk op dat het gebruik van macro's op deze manier zeer krachtig is, maar het ook moeilijker maakt om de code snel te begrijpen.

Opgave: Schrijf voor elke functionaliteit van de applicatie een test. Deze testen kunnen voor, tijdens en na het eigenlijke implementeren van de functionaliteit zelf geschreven worden. De functionaliteit van de GUI (gui.h, gui.c en event.h), hoeft hierbij niet getest te worden. De functionaliteit van de game loop implementatie (editor.c) hoeft eveneens niet getest te worden. Er zullen echter extra functies geschreven moeten worden voor de implementatie van de game loop in editor.c. Deze functies moeten wel getest worden.

3.5 Extra informatie

3.5.1 Compileren op Windows

Op Windows raden we aan om "Visual Studio 2015 Community" te gebruiken. Je kan deze software downloaden op https://www.visualstudio.com/downloads/download-visual-studio-vs

Eenmaal geïnstalleerd, kan je de "solution file" openen. Dit is in de opgave code de file "*LevelEditor\LevelEditor.sln*".

Er zitten 3 projecten in de "solution":

- LevelEditor: Dit is een library met gemeenschappelijke code. Bijna alle code zit hierin. Dit project bevat geen main functie, en zal dus een foutboodschap geven indien je probeert uit te voeren.
- GuiLevelEditor: Dit bevat *editor.c* met de main functie. Dit project start de GUI.
- TestLevelEditor: Dit project bevat *test_editor.c* met main functie voor de testen. Dit start de testen.

Om de GUI of de testen te starten, ga je als volgt te werk:

- Klik met de rechtermuisknop op het project (GuiLevelEditor of TestLevelEditor)
- Selecteer "Set as StartUp Project"
- Klik vervolgens op de Run knop bovenaan (of gebruik de F5 toets)

De opgave bestanden zullen zonder wijzigingen reeds de (beperkte) GUI weergeven, en de testen starten. Uiteraard zal de GUI nog niet veel doen (oa geen level tonen), en zal de eerst gestarte test al falen.

3.5.2 Compileren op Linux

Om allegro op Linux te compileren, moet Allegro 5.0 eerst geïnstalleerd worden. Hoe Allegro installeren hangt af van de specifieke Linux distro.

Voor Ubuntu probeer je best eerst deze methode:
https://wiki.allegro.cc/index.php?title=Install_Allegro_from_Ubuntu_PPAs

Als dat niet lukt, kan je proberen allegro zelf te compileren. Op Ubuntu (of andere Debian gebaseerde distro's) kan je daarvoor deze instructies volgen:
https://wiki.allegro.cc/index.php?title=Install_Allegro5_From_Git/Linux/Debian

Deze instructies zijn gelijkaardig voor andere Linux distributies.

Om de project code te compileren moeten er enkele packages, zoals de compiler, geïnstalleerd zijn. Op vele Linux systemen zal dat al het geval zijn. Op Ubuntu kan je ze indien nodig installeren op de volgende manier:

```
sudo apt-get install build-essential pkg-config
```

Het compileren zelf is eenvoudig. Ga naar de map waar allegro geïnstalleerd is en voer daar het volgende commando uit:

```
make
```

Dit zal automatisch de allegro installatie detecteren en alles compileren. Je kan de editor starten en de testen laten lopen met de volgende commando's:

```
make run

make test
```

3.6 Opgegeven bestanden

Een deel van de code wordt bij de opgave meegegeven. Het is de bedoeling de bestanden waar nodig aan te passen om zo tot een werkende oplossing te komen.

Volgende bestanden zijn geheel of gedeeltelijk gegeven:

- gui.h, event.h en gui.c: De grafische interface. Deze is reeds volledig geïmplementeerd.
- common.h en common.c: Gemeenschappelijke structs en functies. In common.c moeten nog enkele functies ingevuld worden.
- level.h en level.c: structs en functies voor het bijhouden van levels. Level.h moet niet aangepast worden. In level.c moeten alle functies geïmplementeerd worden.
- editor.c: Bevat de main functie en de game loop. Hier is een minimale basis gegeven, die uitgebreid moet worden.
- pathfinder.h, pathfinder.c: Hier moet de path finding functionaliteit geïmplementeerd worden.
- dwgraph.h en dwgraph.c: De kortste pad zoektocht maakt gebruik van een kortste pad algoritme in een gerichte gewogen graaf ("directed weighted graph", vandaar dwgraph). Er moet beslist worden hoe de graaf bijgehouden wordt, en dit moet geïmplementeerd worden.
- pqueue.h en pqueue.c: Het gebruikte kortste pad algoritme maakt gebruik van een priority queue, die in deze bestanden reeds geïmplementeerd is.
- test_editor.c: In dit bestand komen de tests die de verschillende functionele onderdelen zullen testen. Enkele tests zijn als voorbeeld gegeven.

3.7 Uitbreidingen

Eenmaal de basisopgave van deel 1 van het project klaar is, kunnen er uitbreidingen aan toegevoegd worden. (zie ook de puntenverdeling in 2.4)

3.7.1 Optimaliseren Priority Queue

De opgegeven implementatie van de priority queue werkt, maar is zeer inefficiënt qua snelheid en geheugengebruik. Verbeter deze implementatie. Zorg er voor dat de interface gelijk blijft. Het kan hierbij handig zijn om een kleine test-applicatie te schrijven die zeer veel gebruik maakt van

de priority queue, en de tijd meet. Op die manier kan vergeleken worden hoe snel beide implementaties zijn.

3.7.2 Wegen

Voeg een nieuwe type tegel toe aan de level editor: een weg. Eenheden die over wegen bewegen gebruiken slechts de helft zoveel action points als normaal. Tegels met wegen staan dus op halve afstand (tegenover gewone tegels) van elkaar in de graaf. Deze bonus geldt enkel voor stappen tussen 2 tegels met een weg, dus niet van een tegel zonder weg naar een tegel met weg. Bovendien geldt de bonus enkel voor horizontale en verticale stappen, en dus niet voor diagonale stappen.

Er zijn verschillende wijzigingen nodig om wegen te implementeren:

- Het tegeltype “weg” moet toegevoegd worden aan het level. Het moet ook bij het opslaan en inlezen ondersteund worden.
- De berekening van het kortste pad moet met wegen rekening houden.
- De visualisatie (gui.c) moet aangepast worden. De afbeelding die wordt weergegeven voor een tegel met een weg, hangt af van of er wegen aanwezig zijn naast die tegel. Er zijn daardoor 16 verschillende afbeeldingen voorzien voor een weg, en de juiste afbeelding moet gekozen worden.

3.7.3 Flexibele level dimensies

Momenteel hebben de levels vaste dimensies: 25x12. Zorg er voor dat de dimensies flexibel zijn. De level editor moet dus levels kunnen inlezen die een andere dimensie dan de standaard 25x12 hebben. Levels met een andere dimensie moeten na wijzigingen ook terug kunnen weggeschreven worden. Voeg ook een command line argument toe aan de applicatie, dat bepaalt welke dimensies een leeg level heeft als er op de “Clear” knop geklikt wordt. Om de GUI testen kan je ook manueel level-bestanden maken met verschillende dimensies.

3.7.4 Bugfix

Er zit een bug in gui.c. De bug duikt op als er meer dan 4 tekstboodschappen worden getoond. Soms lukt het om 5 boodschappen te tonen, maar meestal loopt dit mis en worden slechts de 4 laatste boodschappen getoond. Zoek de oorzaak en los deze zo eenvoudig mogelijk op. Deze (kleine) uitbreiding telt niet mee voor de verzameling vrij te kiezen uitbreidingen die gequoteerd worden.

3.7.5 Toevoegen flexibele thema’s

De level editor ondersteunt reeds verschillende thema’s. Eenzelfde level, wordt met behulp van een thema anders weergegeven (i.e. de gebruikte afbeeldingen veranderen). Zorg dat de editor een extra knop heeft, die het thema verandert. Er zijn momenteel 2 thema’s voorzien, en de knop moet dus bij elke muisklik naar het alternatieve thema overschakelen. Het gekozen thema moet ook worden opgeslagen in de levelbestanden en moet ook ingesteld worden bij het inlezen uit een levelbestand. Hiervoor mag je zelf kiezen hoe je het level-bestandsformaat aanpast.

3.7.6 Binair formaat levels

In de basis opgave worden de levels opgeslagen in een tekstueel formaat. Hieronder vinden jullie de beschrijving van een binair formaat, waarin de levels opgeslagen en ingelezen moeten kunnen worden. Maak gebruik van de door de gebruiker gekozen extensie bij het opslaan en inlezen om te detecteren of het level in binair of tekstueel formaat moet worden weggeschreven of ingelezen.

De bestandsextensie voor het tekstuele formaat is: .world

De bestandsextensie voor het binaire formaat is: .wld

Er zijn in de opgave levels voorzien in het binaire formaat. Volgende tabel geeft weer hoe een level in het binaire formaat omgezet wordt:

<table>
<tr><th>Byte count</th><th>Data</th></tr>
<tr><td>4</td><td>Identificatie van het bestandsformaat. De meeste bestandstypes beginnen met enkele vaste bytes die helpen om het formaat te herkennen. Voor deze opgave zijn volgende 4 vaste tekst karakters gekozen: .SOI</td></tr>
<tr><td>1</td><td>Versie nummer: 1</td></tr>
<tr><td>2</td><td>Breedte (type: unsigned short, big endian)</td></tr>
<tr><td>2</td><td>Hoogte (type: unsigned short, big endian)</td></tr>
<tr><td>breedte x hoogte</td><td>1 byte per tegel.
Volgorde cellen: alle tegels in een rij na elkaar, en rij 0 eerst. Dus eerst rij 0 kolom 0, dan rij 0 kolom 1, dan rij 0 kolom 2, …, dan rij 1 kolom 0, dan rij 1 kolom 1, …
Met volgende formaat voor de bits (hoogste bits eerst):
<table>
<tr><th>Bit count</th><th>Data</th></tr>
<tr><td>2</td><td>Owner (none = 0, human = 1, AI=2)</td></tr>
<tr><td>3</td><td>Terrain type (ground=0, water=1, rock=2, hq=3, bridge=4)</td></tr>
<tr><td>1</td><td>Road (0 if no road, 1 if road)</td></tr>
<tr><td>2</td><td>Unit (0 als geen unit, anders nummer unit (1,2 of 3))</td></tr>
</table>
</td></tr>
</table>

Merk op dat dit formaat meer toelaat dan de Level struct aan kan, het laat bijvoorbeeld een brug met een eenheid op toe. Zo'n ongeldige situaties moeten bij het inlezen opgevangen worden door de ongeldige tegel als lege grond te zien.

Opmerking: de "shorts" voor breedte en hoogte in de header vereisen endian-aware code. Het omzetten van een unsigned short in "host system" order (typisch little endian op x86 hardware) naar big endian, kan met de functie htons(). Omzetten van big endian naar "host system" endian kan met ntohs().

De reden hiervoor is dat het bestand op dezelfde manier moet opgeslaan worden op verschillende systemen. Als hier geen aandacht aan besteed wordt, zal het bestand op big-endian systemen levels wegschrijven die niet compatibel zijn met little-endian systemen, en vice versa.

Een voorbeeld programma:

```
#ifdef WIN32
#include <Winsock2.h>
#pragma comment(lib, "Ws2_32.lib")
#else
#include <arpa/inet.h>
#endif

#include <stdio.h>

int main() {
    unsigned short example = 42;
    printf("Host order value: %d  (hexadecimal: %04X)\n",
       example, example);
    unsigned short bigendian = htons(example);
    printf("Big endian value: %d  (hexadecimal: %04X)\n",
       bigendian, bigendian);
    unsigned short back = ntohs(bigendian);
    printf("Restored host order value: %d  (hexadecimal:
%04X)\n",
       back, back);
    return 0;
}
```

Output:

```
Host order value: 42  (hexadecimal: 002A)
Big endian value: 10752  (hexadecimal: 2A00)
Restored host order value: 42  (hexadecimal: 002A)
```

4 Implementatie Strategic Wars (C++)

4.1 Spelregels

4.1.1 Spelwereld

Een spelwereld heeft hetzelfde bestandsformaat als wat er als output wordt gegenereerd in de level editor (voor het formaat zie 3.3.2). Het spel moet dergelijk bestand kunnen inlezen en correct de spelwereld weergeven. Op basis van het level moet het spel dus geïnitialiseerd kunnen worden. Een level bestaat normaal uit een grid van maximaal 25x12 cellen.

4.1.2 Units

We verwachten drie types van units. Deze drie type units moeten in een driehoeksverhouding sterker zijn tegen elkaar (blad-steen-schaar principe). Bijvoorbeeld: soldaten met zwaard zijn goed tegen pijl en boog, pijl en boog soldaten zijn goed tegen kanon, kanon soldaten zijn goed tegen zwaard. Verder heeft elke unit een aantal 'stats': health points (hp), damage point (dp), action points (ap), minimum rang (range_min) en maximim range (range_max).

Volgende tabel geeft een voorbeeld van dergelijke 'stats' die kunnen gebruikt worden:

	Infantry (zwaard)	Archery (boog)	Fire (kanon)
HP	10	10	10
DP	2	3	4
AP	2	3	1
range_min	0	0	2
range_max	1	3	5

Een aanval van een type unit die goed is tegen en ander type unit, zal dan +50% damage points doen op diens health points.

4.1.3 Movement

Units kunnen verticaal, horizontaal en diagonaal over het speelveld bewegen. Een beweging kost hen 1 AP. Units mogen nooit met 2 op 1 vakje staan. Ze mogen ook niet door een andere unit heen passeren.

De pathfinding zoals jullie die implementeerden voor de level editor in C kan als basis dienen voor de pathfinding in het spel. Probeer dus hetzelfde algoritme van je C code te porten naar C++. Probeer hier aandacht te hebben voor die plaatsen waar je verbeteringen kan aanbrengen, en wees niet terughoudend om iets anders te doen omdat de C++ taal zich daar beter voor lijkt te lenen.

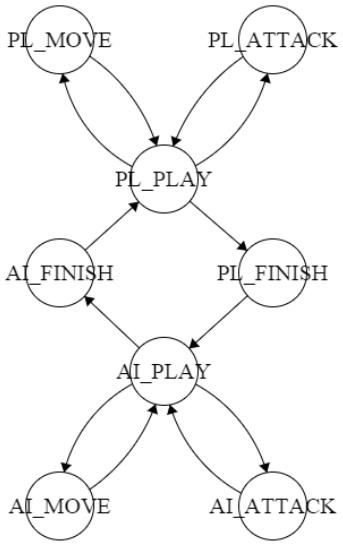
4.1.4 Aanvallen

Aanvallen kost ook 1 AP. Bij een aanval wordt éénmalig het aantal DP (+ eventueel 50% bij sterker type) van het doelwit zijn HP afgetrokken. In het geval er meerdere APs over zijn, kan opnieuw aangevallen worden.

Behalve units, kunnen ook gebouwen aangevallen worden (hier enkel de headquarters). Deze headquarters hebben dus ook hp, worden ze volledig vernietigd, dan kan je zo het spel ook winnen.

4.1.5 Beurtensysteem

Het betreft een turn-based spel, dus er wordt elk om de beurt gehandeld. Dit kan het eenvoudigst voorgesteld worden in verschillende game states. Hieronder een voorbeeld.



Figuur 2: State diagram

Men start in de PL(ayer)_PLAY state. Daar kan je een unit selecteren en die verplaatsen of laten aanvallen, respectievelijk wordt de state naar PL_MOVE of PL_ATTACK gezet. Zo weet het RenderSysteem dat het nu een beweging of aanval moet renderen. Is die animatie afgerond, dan wordt de state terug op PL_PLAY gezet en heeft de speler weer controle over de resterende units.

Als alle APs op zijn, of wanneer de speler dat manueel beslist (door bijvoorbeeld de spatiebalk in te drukken) wordt de state naar PL_FINISH gezet. Deze state zal dan bijvoorbeeld een informatieboodschap renderen die zegt dat het aan de computer is, en de state wordt naar AI_PLAY gezet, waar een analoge sequentie gestart wordt.

Dit is slechts een voorbeeld van hoe het zou kunnen werken, je bent natuurlijk vrij om dit anders te doen ook, maar probeer het zeker overzichtelijk te houden!

4.1.6 Input via muis

Het spel moet speelbaar zijn met een muis. De allegro library heeft hier ook support voor. Hoe je dit exact doet, kan je uitzoeken in de documentatie.

De bedoeling is dat het spel volledig speelbaar is met de muis wat betreft selectie van units en hen bewegen, laten aanvallen. Het eindigen van een beurt mag gerust met een toets als spatie gebeuren. Signaleren dat je klaar bent met een unit ondanks dat er nog APs over zijn, kan eventueel ook met een knop gebeuren (enter bijvoorbeeld).

Op het moment dat er een bewegingspad of aanvalsdoelwit ingegeven is, neemt het spel even de controle over en kan er niets geselecteerd worden door de speler. Na de animatie wordt de controle terug gegeven aan de speler.

4.1.7 Computer AI

De computer zal een aantal AI strategieën moeten hebben. Dit hoeft niet geavanceerd te zijn, maar als je zelf geen weerstand biedt, moet de computer wel kunnen winnen en units aanvallen.

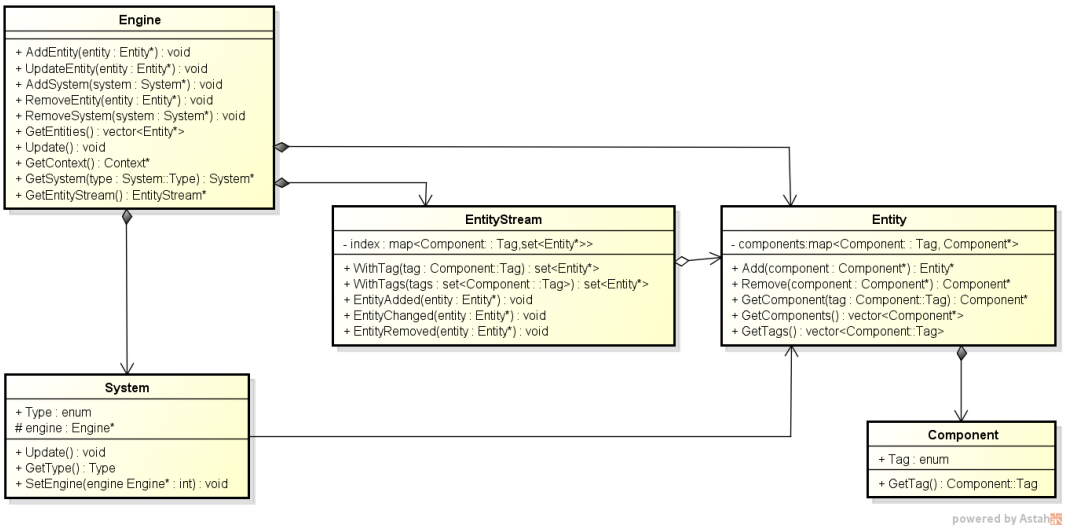
4.1.8 Game over condities

Zoals eerder vermeld, zijn er twee verschillende manieren om het spel te winnen (en dus ook verliezen):

- Alle units van de tegenspeler zijn vernietigd.
- Het hoofdkwartier van de tegenspeler is vernietigd.

4.2 Basis entity system framework

Meer informatie over een entity system framework was te vinden op eerder vermelde bronnen. In het project zullen jullie een basisversie van een dergelijk systeem dienen te implementeren. Het resulterende framework wordt dan gebruikt om het spel verder te ontwerpen. In Figuur 3 zien we meteen een overzicht van het framework.



Figuur 3: Entity System Framework

Het framework bestaat uit een aantal belangrijke klassen, die nu kort één per één overlopen worden.

4.2.1 Component

Een Component is een heel eenvoudige data-container. De bedoeling is conceptuele eigenschappen in een klasse te steken die overerft van Component. De attributen mag je voor de eenvoud publiek houden aangezien deze dikwijls zullen moeten uitgelezen worden.

Voor elk nieuw Component-type wordt er dus een nieuwe klasse aangemaakt die overerft van Component.

4.2.2 Entity

Een entity stelt om het even welk object voor in het spel. Entities worden opgevuld door een verzameling Components toe te voegen die de eigenschappen van het object voorstellen.

De belangrijkste methodes van Entities zijn:

- Entity* Add(Component * component): Voegt een Component toe via een pointer naar deze component. Een pointer naar de Entity wordt teruggegeven om eventueel via een Builder-patroon stijl verder te kunnen doen, maar dit hoeft niet.
  (eg. `entity.Add(new XCoponent()).Add(newXXComponent);`)
- Component* Remove(Component * component): Verwijdert een Component via een pointer naar de te verwijderen component. Een pointer naar de verwijderde Component wordt teruggegeven. (verwijderen betekent niet vernietigen!)
- Component* GetComponent(Component::Tag tag): Geeft een pointer naar een Component terug door opgeven van zijn Tag (enum-waarde).
- std::vector<Component*> GetComponents(): Geeft een vector van Component pointers terug die wijzen naar alle Componenten van deze entiteit.
- std::vector<Component::Tag> GetTags(): Geeft een vector terug met alle Tags (enum-waardes) van de Componenten van deze entiteit.

4.2.3 System

Een System stelt een bepaald aspect van het spel voor. Bijvoorbeeld de rendering, de movement, etc. Alle Systems zullen moeten overerven van de hoofdklasse System. Systems zullen een reeks van Entities verwerken in elke iteratie van de game-loop.

System beschikt over de volgende methodes:

- virtual void Update() = 0 : Dit is een methode die verplicht te implementeren is door alle overervende System klassen. Hierin gebeurt het merendeel van de logica. Deze methode wordt 1 keer per game loop iteratie aangeroepen (ongeveer 60 game loop iteraties per seconde).
- virtual Type GetType() = 0: Dit is een verplicht te implementeren methode voor alle overervende System klassen en geeft de Type enum waarde terug voor dit systeem.
- void SetEngine(Engine* _engine): Dit is een eenvoudige setter om een pointer naar de engine beschikbaar te hebben in alle overervende systemen. (protected veld engine wordt hiermee ingesteld).

Bovendien beschikt System ook over een enum Type die het Type van System voorstelt. Deze worden gebruikt om de Systems aan een map toe te voegen met als sleutel hun Type. Zo kunnen de Systems efficiënt opgevraagd worden.

4.2.4 Engine

De Engine is de hoofdklasse van het framework. Hier komt alles functioneel samen. Zowel Entities als Systems worden toegevoegd aan de engine. De Engine heeft een Update() methode die zal opgeroepen worden voor elke iteratie van de game loop. In deze methode wordt elk System aangeroepen. Systems zullen via de EntityStream (zie hieronder) de nodige Entities kunnen opvragen om deze te verwerken. Systemen kunnen ook met andere Systemen communiceren door ze op te vragen bij de Engine.

Engine beschikt over de volgende methodes:

- void AddEntity(Entity* entity): Voegt een Entity toe aan de Engine via een pointer.
- void UpdateEntity(Entity* entity): Zorgt dat de Entity correct gereflecteerd wordt na een verandering van de Components. Dit kan bijvoorbeeld door verwijderen en opnieuw toevoegen van de Entity. (zie ook uitleg 4.2.5)
- void AddSystem(System* system): Voegt een System toe aan de Engine via een pointer.
- void RemoveEntity(Entity* entity): Verwijdert een Entity van de Engine via een pointer.
- void RemoveSystem(System* system): Verwijdert een System van de Engine via een pointer.
- std::vector<Entity*> GetEntities(): Geeft een vector met pointers naar alle Entities uit de Engine terug.
- void Update(): Roept om de beurt de Update methode aan op alle Systems die momenteel aan de Engine toegevoegd zijn.
- Context* GetContext(): Geeft een pointer naar de Context klasse terug. (zie ook 4.5.5)
- System* GetSystem(System::Type type): Geeft het System terug die hoort bij de Type enum waarde type. Indien niet aanwezig, wordt NULL teruggegeven.
- EntityStream * GetEntityStream(): Geeft een pointer terug naar het EntityStream object (zie volgende subsectie).

4.2.5 EntityStream

Dit is een hulpklasse. Aan de EntityStream kan een set van Entities opgevraagd worden door op te geven welke Componenten ze moeten hebben. Een RenderSystem zal bijvoorbeeld de Entities willen met een PositionComponent en een TextureComponent.

EntityStream moet dus op de hoogte zijn van alle Entities die toegevoegd en/of verwijderd worden van de Engine. Dit is mogelijk door in de Engine klasse bij toevoegen / verwijderen van een Entity de respectievelijke methodes van EntityStream op te roepen: EntityAdded en EntityRemoved. Deze zullen de interne mapping in een std::map van (sleutel) Component::Tag naar (waarde) std::set<Entity*> correct aanpassen.

EntityStream beschikt dus over de volgende methodes:

- std::set<Entity*> WithTag(Component::Tag tag): Geeft een set terug met pointers naar de Entities die toegevoegd zijn aan de Engine én die een component met de opgegeven tag bevatten.
- std::set<Entity*> WithTags(std::set<Component::Tag> tags): Geeft een set terug met pointers naar de Entities die toegevoegd zijn aan de Engine én die een component bevat met één of meerdere van de opgegeven tags.
- void EntityAdded(Entity* entity): Wordt aangeroepen door de Engine bij toevoegen van een Entity. Een pointer naar de Entity wordt aangeleverd via het argument. Daarmee wordt de interne mapping aangepast, zodat de WithTag(s) methodes correct kunnen gebruikt worden.
- void EntityChanged(Entity* entity): Wordt aangeroepen door de Engine bij veranderen van (de Componentes van) een Entity. Een pointer naar de Entity wordt aangeleverd via het argument. Daarmee wordt de interne mapping aangepast, zodat de WithTag(s) methodes correct kunnen gebruikt worden. [Enkel nodig als Components added of removed worden terwijl de Entity al toegevoegd werd aan de Engine, niet bij het gewoon aanpassen van velden van een bestaande Component]

- void EntityRemoved(Entity* entity): Wordt aangeroepen door de Engine bij verwijderen van een Entity. Een pointer naar de entity wordt aangeleverd via het argument. Daarmee wordt de interne mapping aangepast, zodat de WithTag(s) methodes correct kunnen gebruikt worden.

4.2.6 Opmerking

Deze klassen bepalen het basis Entity System Framework dat zal ontwikkeld en gebruikt worden. Concretere informatie over de overervende klassen volgt nog, maar dit geeft al het belangrijkste idee van de onderlinge interacties tussen de delen van het framework.

4.3 Components

De verschillende componenten hangen uiteraard af van de functionaliteit in het spel en de manier waarop je zelf kiest om de aspecten in componenten op te delen. Er zijn zeker andere logische opsplitsingen mogelijk, uiteraard kan een volledige onlogische opsplitsing zich wel reflecteren in een lagere score.

Wij hebben alvast volgende componenten gebruikt voor de voorbeeldoplossing:

PositionComponent	Grid pos	De huidige positie-coördinaat in de spelwereld
	int z	De z-as (diepte). Dit wordt gebruikt als een laag-systeem om te weten in welke volgorde Entities die overlappen, moeten gerenderd worden. Dit lukt wanneer de Entities ingevoegd worden in een set met een custom compare die sorteert op het z veld indien een PositionComponent aanwezig is.
TextureComponent	Allkit::Sprite texture	Een enumwaarde die de sprite voorstelt
	Color color	De kleur waarin getekend moet worden indien texture gelijk is aan Allkit::Sprite::SPRT_NONE
UnitComponent	Type type	Het type unit (Archery, Infantry, Fire).
	int player	De speler die de units bezit.
	int hp	Stats van de unit zoals beschreven in 4.1.2.
	int ap	
	int dp	
	int range_min	
	int range_max	

4.4 Systems

Ook de verschillende systemen hangen wederom af van de functionaliteit in het spel. Net als voorheen mag je gerust andere systemen maken of gebruiken, zo lang deze nog een logische opsplitsing blijven.

Wij hebben alvast onder andere volgende Systems gebruikt voor de voorbeeld oplossing:

- **MouseSystem:** Reageert op alle acties die met de muis ondernomen kunnen worden, zoals muisbeweging, klikken, etc.
- **AnimationSystem:** Zorgt ervoor dat de status van te animeren Entities correct aangepast wordt, zodat het RenderSystem deze kan renderen als gelijk wel andere Entity.
- **RenderSystem:** Zorgt voor het renderen van alles wat een TextureComponent en PositionComponent heeft, alsook de tekst op het scherm.

Let op: Je zal zeker nog andere systemen moeten maken! Je hoeft ook niet alle Systems te gebruiken die hierboven aangegeven zijn.

Hieronder volgen een aantal systemen die we nog wat nauwkeuriger uitleggen.

4.4.1 MouseSystem

Wanneer de speler controle heeft, toon je een selectievakje op de cell waarboven de muis hovert. Bij een linkerklik op een unit van jezelf, wordt deze geselecteerd. Nu wordt een pad getekend naar het vakje waarboven de muis hovert.

Indien dit pad te ver is (wat betreft het aantal APs die de unit over heeft), wordt dit ook aangetoond. Je kan dit aantonen zoals je zelf wilt, met een anders gekleurde pad indicator, of door de cell waarnaartoe het pad getekend wordt met een andere sprite aan te duiden (dit zijn slechts voorbeelden).

Bij nogmaals links klikken wordt een movement bevestigd. Bij links klikken boven een vijand, wordt aanvallen bevestigd.

Om een unit te deselecteren mag de toets ESC gebruikt worden.

4.4.2 AnimationSystem

Wanneer een movement of aanval bevestigd wordt, wordt dit (lees: de nodige informatie) op de één of andere manier in een (bv. Animation)component opgeslaan en aan die unit (Entity) gehangen.

Het AnimationSystem leest dan alle Entities met een AnimationComponent. Dit System kan dan waarden aanpassen die nodig zijn om de correcte animatie weer te geven.

Bijvoorbeeld, bij movement zal er waarschijnlijk een aanpassing gedaan worden aan de positie. Anderzijds wordt dan waarschijnlijk in de AnimationComponent bijgehouden hoe ver tussen twee zulke posities we zitten. Als de animatie voltooid is, kan de speler weer units selecteren om zijn/haar beurt verder te zetten.

4.4.3 RenderSystem

Het RenderSysteem zal voor elk frame de rendering voor zich nemen. De puntjes hieronder zijn heel belangrijk en moeten opgevolgd worden!

Belangrijk:

- Renderen is een computationeel intensief aspect van het spel. Het wordt daarom zo min mogelijk nodeloos gedaan. Dit wordt normaal gezien door de programmeur gedetecteerd, we hebben deze functionaliteit echter ingebouwd in AllegroLib (4.5.6). De bedoeling is dat jullie in het begin van de Update functie kijken of er wel degelijk moet getekend worden met bool AllegroLib::Instance ().IsRedraw();
- Als er dan effectief getekend wordt, moet eerst en vooral het scherm leeggemaakt worden met Graphics::Instance().ClearScreen(); zodat op een zwart leeg canvas kan begonnen worden met tekenen.
- Als allerlaatste in de Update() methode van RenderSystem, moet de Grahpics::Instance().ExecuteDraws() methode aan geroepen worden. Elke tekenoperatie wordt namelijk naar een bitmap in het geheugen gedaan, bij oproepen van ExecuteDraws() wordt de bitmap in het geheugen gewisseld met de bitmap op het scherm. Deze techniek, beter gekend als buffering, zorgt ervoor dat er geen flikkering in het scherm komt bij het renderen.

4.5 Hulpklassen

4.5.1 Grid

Deze hulpklasse gebruik je voor het bijhouden van de locatie in de speelveld tabel. Het bevat een row en column veld, meer niet. Je kan dit voor het gemak zelfs met public velden doen. Een Grid stellen we typisch voor door een koppel (row, column).

Een aantal operators overloaden is ook handig en soms nodig. Denk hierbij aan:

- operator==
- operator!=
- operator<

4.5.2 Vector2

Deze hulpklasse stelt ook twee int velden voor, net zoals Grid, maar noemt anders en dit zal helpen om coördinaten in pixels voor te stellen. Een coördinaat stellen we typisch voor door een koppel (x,y). Let erop dat dit anders is dan bij een Grid. (y -> row, x -> column).

Het voordeel van ook een klasse Vector2 te hebben, is dat het zo heel duidelijk wordt in de signatuur van methoden en functies, welke soort argument je nu verwacht een cel-locatie of een pixel coördinaat.

Ook hier geldt dat dezelfde extra operatoren handig kunnen zijn als bij Grid.

4.5.3 Color

Dit is een hulpklasse die toelaat een kleur voor te stellen. Ofwel als rgb met waarden van 0-255. Ofwel als rgba met waarden van 0,0-1,0.

4.5.4 Tags

Dit is een hulpklasse die je toelaat om aan de hand van een builder patroon een set van Compontent::Tag tags op te stellen. Dit zal handig zijn bij het opvragen van Entities aan de EntityStream klasse. Voorbeeld gebruik:

```
std::set<Component::Tags> tags = (Component::POSITION).And(Component::MOVEMENT).List();
```

4.5.5 Context

Dit is een hulpklasse waar een aantal variabelen in staan die je kunnen helpen. Overkoepelende state die je wil bijhouden (zoals de GameState van het spel op dit moment, zie 4.1.5), is daar slechts een voorbeeld van.

Hint: Aangezien Context meegegeven wordt aan een Game en vervolgens aan een Engine. Kan via het protected engine attribuut de context in elk System opgevraagd worden.

4.5.6 AllegroLib

- void Init(Vector2 _screensize, float _fps): Initialiseert de allegro bibliotheek. Dit moet aangeroepen worden alvorens ook maar iets te tekenen.
- void StartLoop(): Deze methode start de achterliggende timer die 60 keer per seconde de game loop laat uitvoeren. Ze moet **net voor** de while van de gameloop komen!
- void StartIteration(): Deze methode wacht totdat het volgende allegro event ontvangen wordt en houdt bij wanneer er moet hertekend worden als er een TIMER event ontvangen werd. Ze moet **als eerste** opgeroepen worden in een iteratie van de while van de gameloop!
- void Destroy(): Ruimt alle interne structuren van allegro op. Roep dit als laatste in je programma op, anders heb je sowieso memory leaks!
- bool IsWindowClosed(): Geeft aan dat het laatste opgevangen allegro event al dan niet een window close event was en dat dus het scherm werd gesloten door middel van het kruisje rechts boven.
- bool IsTimerEvent(): Geeft aan dat het laatste opgevangen allegro event al dan niet een timer event was.
- bool IsKeyboardEvent(): Geeft aan dat het laatste opgevangen allegro event al dan niet een keyboard event was.
- bool IsMouseEvent(): Geeft aan dat het laatste opgevangen allegro event al dan niet een muis event was.
- ALLEGRO_EVENT GetCurrentEvent(): Staat toe om het laatste allegro event op te vragen om het zelf te kunnen verwerken.

AllegroLib is een wrapper rond de allegro bibliotheek. Allegro laat ons toe om op relatief eenvoudige wijze naar het scherm te tekenen en een game te ontwikkelen. Via AllegroLib hebben we Allegro nog sterk vereenvoudigd naar jullie toe. **Het is dan ook de bedoeling dat jullie steeds gebruik maken van AllegroLib waar mogelijk. Allegro specifieke methodes (beginnen typisch met al_) moeten dus meestal niet gebruikt worden!**

AllegroLib werd als **singleton** geïmplementeerd. **Opvragen van de instance doe je dus steeds met de statische methode AllegroLib::Instance().** Dan heb je een (de enige) instantie van AllegroLib en kan je de nodige methodes oproepen.

4.5.7 Graphics

Zoals hierboven uitgelegd doet de AllegroLib wrapper al events, timing, initialisatie en opruiming voor jullie. Jullie moeten ook zelf een aantal functies wrappen die met tekenen te maken hebben. Dit doen jullie in de Graphics klasse.

De Graphics klasse is ook als singleton geïmplementeerd en die code krijgen jullie al. De bedoeling is dus dat het aanroepen van volgende methodes, resulteert in een call naar de allegro instance die dit dan correct doet volgens de allegro API.

- void LoadFonts(): De fonts worden ingeladen en opgeslaan in een lokale variabele. (fonts staan in de assets folder)
- void LoadSpriteCache(): De sprites worden in geladen in een vector van ALLEGRO_BITMAP*, deze kan als cache gebruikt worden. De vector is geïndexeerd door middel van een enum met de Sprite namen.
- void UnLoadFonts(): Verwijder de ingeladen fonts.
- void UnLoadSpriteCache(): Verwijder de ingeladen sprites.
- void ExecuteDraws(): Wissel de bitmap in het geheugen met die op het scherm.
- void ClearScreen(): Leeg het gehele scherm (of de dus de bitmap) door het helemaal zwart te schilderen.
- void GenerateBackgroundSprite(World* world): Stelt de bitmap waarnaar getekend wordt op de correct ALLEGRO_BITMAP* van de sprite vector (cache). Teken de volledige level (achtergrond) naar deze bitmap. Zet daarna de bitmap waarnaar getekend wordt opnieuw op de backbuffer van de ALLEGRO_DISPLAY*.
- void DrawBitmap(Sprite sprite, float dx, float dy): Tekent een sprite aangegeven door de enum, op locatie (dx,dy) op het scherm. (let op: (0,0) links boven!)
- void DrawString(std::string str, float dx, float dy, Color c, Align align, bool hugeFont): Tekent de opgegeven string op het scherm op de locatie (dx,dy) met de kleur c en alignatie align. hugeFont is een boolean om te kiezen tussen een groot of klein font.
- void DrawRectangle(float dx, float dy, float width, float height, Color c, float thickness)): Tekent een rechthoek op het scherm op locatie (dx,dy) met een breedte width en hoogte height, in kleur c, met een lijndikte van thickness pixels.
- Vector2 ToPx(Grid gridloc): Zet de gegeven Grid locatie om in de correcte pixel coördinaat op het scherm.
- int GetGridSize(): Geeft de ingestelde grootte van de sprites en dus de Grid cellen terug.

4.6 Opgegeven bestanden

4.6.1 Assets

- De folders **fonts** en **images** folders bevatten de respectievelijke resources.
- De folder **levels** is leeg en kan gebruikt worden om standaard levels in op te slaan of uit te lezen.

4.6.2 Header bestanden

- Allegro
  - AllegroLib.h
    Deze header file is volledig geïmplementeerd voor jullie. Het is een wrapper voor de meer ingewikkelde functionaliteit van de allegro bibliotheek (timer, events, etc)
  - Graphics.h
    Deze header file is slechts gedeeltelijk geïmplementeerd voor jullie. Het is een wrapper voor de meer eenvoudige functionaliteit van de allegro bibliotheek (draw calls, sprites, fonts, etc)
- Components
  - Component.h
    Deze header file is nagenoeg volledig geïmplementeerd voor jullie. Het is de base class voor een Component uit het Entity System. Bij toevoegen van nieuwe overervende Components, zal je hier de Tag enum moeten aanvullen, zodat elke Component zichzelf kan identificeren met de GetTag() methode.
  - TextureComponent.h
    Deze header file is volledig geïmplementeerd voor jullie. Je bent niet verplicht deze Component te gebruiken of te laten staan. Het dient als voorbeeld van een Component. Deze houdt een Graphics::Sprite enum waarde bij of een Color. Het kan dus gebruikt worden om een texture te bewaren van een Entity, of een fill-color.
- Systems
  - System.h
    Deze header file is nagenoeg volledig geïmplementeerd voor jullie. Het is de base class voor een System uit het Entity System. Bij toevoegen van nieuwe overervende Systems, zal je hier de Type enum moeten aanvullen, zodat elk System zichzelf kan identificeren met de GetType() methode.
  - RenderSystem.h
    Deze header file is volledig geïmplementeerd voor jullie. Je bent niet verplicht dit System te gebruiken of te laten staan. Het dient als voorbeeld van een System. De implementatie van de Update() methode krijgen jullie natuurlijk niet mee.
- Utility
  - Color.h
    Deze header file is volledig geïmplementeerd voor jullie. Het is een wrapper om eenvoudig kleuren te kunnen doorsturen via de Graphics methodes naar allegro.
  - debug_file.h
    Zie 9 Appendix A: Eenvoudig memory leaks detecteren in VS
  - Grid.h
    Deze header file is gedeeltelijk geïmplementeerd voor jullie. Het is een

eenvoudige klasse om een row en column bij te houden. Een aantal operator overloadings zouden deze klasse makkelijker kunnen maken in gebruik.

- Vector2.h
  Deze header file is gedeeltelijk geïmplementeerd voor jullie. Het is een eenvoudige klasse om een x en y bij te houden. Een aantal operator overloadings zouden deze klasse makkelijker kunnen maken in gebruik.

- Other
  - constants.h
    Gebruik deze file om alle #defines te verzamelen, je include de file gewoon als je een constante nodig hebt.
  - Context.h
    Deze header file is gedeeltelijk geïmplementeerd voor jullie. Het is een klasse waarvan een object meegegeven wordt aan de Game klasse. Ze kan gebruikt worden om overkoepelende data op te slaan. De Engine kan immers de Context opvragen.
  - Engine.h
    Deze header file is gedeeltelijk gedefinieerd voor jullie. Het is het hart van het Entity System. De implementatie is grotendeels jullie verantwoordelijkheid.
  - Entity.h
    Deze header file is gedeeltelijk gedefinieerd voor jullie. Het is de Entity van het Entity System. De implementatie is grotendeels jullie verantwoordelijkheid.
  - EntityStream.h
    Deze header file is gedeeltelijk gedefinieerd voor jullie. Deze klasse implementeert de functionaliteit om een set van Entities op te vragen aan de hand van aanwezige Components. Een instantie van deze klasse zal in de Engine opgevraagd kunnen worden.
  - Game.h
    Deze header file is gedeeltelijk gedefinieerd voor jullie. Met een instantie van de Game klasse zullen de resources ingeladen, de gameloop opgestart, en de resources terug opgeruimd worden. De implementatie is grotendeels jullie verantwoordelijkheid.
  - Tags.h
    Deze header file is volledig geïmplementeerd voor jullie. Het is een hulpklasse om eenvoudig een set van Component::Tag te maken. Gebruik als volgt:

```
std::set<Component::Tag> tags =
Tags(Component::TEXTURE).And(Component::POSITION).List();
```

4.6.3 Implementatie bestanden

- AllegroLib.cpp
  Deze cpp file is volledig geïmplementeerd voor jullie.

- Graphics.cpp
  Deze cpp file is gedeeltelijk geïmplementeerd voor jullie.
- Engine.cpp
  Deze cpp file moet nog geïmplementeerd worden.
- Entity.cpp
  Deze cpp file moet nog geïmplementeerd worden.
- EntityStream.cpp
  Deze cpp file moet nog grotendeels geïmplementeerd worden.
- Game.cpp
  Deze cpp file moet nog grotendeels geïmplementeerd worden.
- Main.cpp
  Deze cpp file is grotendeels geïmplementeerd voor jullie. Het is de starting-point voor jullie spel. Heel veel moet hier niet meer veranderd worden, maar je bent vrij om hier aanpassingen aan te maken.

4.6.4 Concept: World

Van World hebben jullie geen bestanden gekregen, die moet je dus nog zelf maken. Je kan hier verschillende kanten mee op. Eén manier is van World ook gewoon een Entity maken met een Component die de wereld bijhoudt bijvoorbeeld. Een andere is World een apart klasse op zich laten zijn die kan opgevraagd worden via de Context klasse.

Deze klasse stelt de wereld voor en moet dus uiteraard een level bestand (zoals gemaakt met de level editor) kunnen uitlezen. Je zal deze World onder andere nodig hebben voor het berekenen van de pathfinding bij het verplaatsen van units.

5 Meetings en contact met assistent

Gedurende het project zullen regelmatig meetings gehouden worden met jullie assistent. **Op deze meetings moet iedereen aanwezig zijn.** We gaan er ook vanuit dat jullie voorbereid naar de meetings komen, want de assistenten moeten dit tussen overige afspraken in plannen en hebben dus niet de tijd om met iedereen uren te vergaderen. Zorg dat je dus eventuele problemen of vragen op voorhand geïdentificeerd hebt.

5/10	Introductie deel 1	Gezamenlijke introductiesessie
12/10	Meeting 1	Eerste meeting met assistent
19/10	Meeting 2	Stand van zaken deel 1
26/10	Deadline deel 1	
9/11	Introductie deel 2 + feedback deel 1	Gezamenlijk introductiesessie + algemene feedback deel 1
23/11	Meeting 3	Stand van zaken deel 2
6/12	Deadline project	

Naast deze vaste afspraakmomenten kunnen ook ad-hoc afspraken geregeld worden indien nodig, dit hoeft dan ook niet met iedereen te zijn.

Uiteraard zijn de assistenten ook steeds bereikbaar via e-mail, waarna ze zo snel mogelijk zullen antwoorden op jullie vragen.

6 Hints en opmerkingen

- Het is zeker toegestaan (en soms nodig!) om opgegeven header files en cpp files aan te passen.
- Begin niet onmiddellijk te programmeren. Lees eerst aandachtig de opgave en denk na hoe je de verschillende problemen zou oplossen (vb. welke klassen ga je gebruiken, wat houden ze bij, …).
- Het voordeel van een Entity Framework System is dat je bepaalde Systemen kan uitschakelen tijdens het debuggen door ze gewoon even niet aan de Engine toe te voegen.
- Maak zoveel mogelijk gebruik van de Standard Template Library (STL), Hoofdstuk 14 van de cursus. Volgende link kan daarbij ook van pas komen: http://www.cppreference.com/wiki/stl/start
- Alle opmerkingen bij de oefenset-opgaves 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…
- Als referentiecompiler wordt Microsoft Visual C++ Community Edition 2015 gebruikt. Zorg er dus voor dat je project daarmee compileert en uitvoerbaar is! Behalve het keyword *auto* is gebruik van C++11 is toegestaan.
- Probeer zoveel mogelijk compiler warnings te vermijden; deze duiden meestal op fouten die Visual Studio C++ 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!**
- ***Sluit het spel altijd af met het kruisje op de UI. Als je de console zelf afsluit zal de allegro bibliotheek niet altijd correct opgeruimd worden en zal je dus eigenaardige errors krijgen.***

7 Opzetten github repository

Na de introductieles zullen jullie een email krijgen van de voor jullie groep verantwoordelijke assistent. Die zal vragen om de github-naam van de github verantwoordelijke in jullie groep terug te mailen.

Deze namen zullen toegevoegd worden aan de private repository van de assistent met de opgave code. Het is aan jullie groepsverantwoordelijke om daarna een 'fork' te nemen van het project en zo de eigen repository op te zetten (dit kan gewoon op de wegpagina van de repository van de assistent eens toegevoegd, door op het knopje fork te klikken, meer info in de e-mail).

Daarna moet de github verantwoordelijke nog zijn eigen groepsleden toevoegen op de eigen repository. Deze repository gebruiken de groepsleden dan als centraal samenwerkingspunt om te clonen, checkout, push/pullen etc. Ook de assistent heeft normaal volledig access tot jullie github space.

8 Indienen

Alles wordt georganiseerd rond de github repository op github.ugent.be (zie ook cursus). De bedoeling is dat jullie daar gebruik maken van de **issue tracker** voor problemen en de **wiki** voor documentatie & informatie.

De assistent kan hierop meevolgen en eventueel zelf antwoorden op issues. Ook zal in plaats van indienen door jullie, de laatste gepushte commit op de github repository als de finaal ingediende versie gezien worden.

Nog een aantal extra puntjes:

- Het project wordt gemaakt in groepen van 4 studenten. Het is verplicht hiervoor een groep op minerva te kiezen. Indien iemand geen groep vindt (na via Minerva project-forum proberen een groep te vinden, dan vragen we **in de eerste week van het project** zo spoedig mogelijk contact met ons op te nemen via soi@intec.ugent.be)
- Hou een kort verslag bij op de github wiki (max. 4 bladzijden) met daarin:
    - naam, voornaam en richting van de groepsleden + groepsnummer van minerva
    - Een extra woordje uitleg bij je ontwerpbeslissingen
    - De taakverdeling: wie heeft wat gedaan?
- De laatste commit op de github repository voor de deadline wordt als final code/state gezien van jullie project.
- Naast de GitHub stuur je ook een document met groepsnummer, werkverdeling en de locatie van je github repository (URL) naar de dropbox van Bruno Volckaert op Minerva, en dit voor de deadline van deel 2 van het project.

Veel succes!

9 Appendix A: Eenvoudig memory leaks detecteren in VS

Naast de gekende methode voor het vinden van memory leaks (gebruikt in de practica), bestaat er een manier om dit proces te stroomlijnen. In plaats van de locatie van het geheugenlek te moeten gebruiken om dan tijdens het opnieuw runnen van je programma een breakpoint te zetten met _CrtSetBreakAlloc(long), kan je gewoon dubbelklikken op het gevonden lek, om direct naar het bestand en de lijn van de allocatie te gaan.

Hoe je dit moet doen, staat in onderstaande bijlage.

Memory leaks vinden in Visual Studio

Via forced include(*) in Visual Studio gaan we bij elke file de volgende file includen: debug_file.h

Dit bestand bevat volgende regels:

```
#ifndef _DEBUG_FILE_H
#define _DEBUG_FILE_H
        #ifdef _DEBUG
                #define _CRTDBG_MAP_ALLOC
                #include <stdlib.h>
                #include <crtdbg.h>

           #ifndef DEBUG_NEW
                  #define DEBUG_NEW new ( _NORMAL_BLOCK , __FILE__ , __LINE__ )
                  #define new DEBUG_NEW
           #endif
        #endif  // _DEBUG
#endif // _DEBUG_FILE_H
```

We overlopen de code:

```
#ifndef _DEBUG_FILE_H
#define _DEBUG_FILE_H
```

Dit zorgt dat er geen dubbele includes gebeuren.

```
#ifdef _DEBUG
        …
        …
#endif  // _DEBUG
```

Dit #ifdef statement zorgt ervoor dat het enkel uitgevoerd wordt als _DEBUG defined is. Bij het builden in debug mode in Visual Studio wordt automatisch de _DEBUG vlag gedefined.

```
        #define _CRTDBG_MAP_ALLOC
        #include <stdlib.h>
        #include <crtdbg.h>
```

De includes zijn normaal gekend van de normale manier van debuggen via de crtdbg bibliotheek. Het define statement ervoor vervangt de malloc statements in de code door een malloc met argumenten, waar de constanten __FILE__ en __LINE__ in meegegeven worden. Dit zijn compiler gedefinieerde constanten die de bestandsnaam en het lijnnummer meegeven waar ze worden opgeroepen.

```
   #ifndef DEBUG_NEW
      #define DEBUG_NEW new ( _NORMAL_BLOCK , __FILE__ , __LINE__ )
      #define new DEBUG_NEW
   #endif
```

Hier gebeurt exact hetzelfde, maar dan manueel bijgevoegd voor het new commando.

Vanaf nu zullen memory leaks op het einde van het programma dubbelklikbaar zijn, zodat rechtstreeks naar het bestand en de regel van de allocatie kan gegaan worden, zonder nog gebruik te moeten maken van _CrtSetBreakAlloc(long _BreakAlloc).

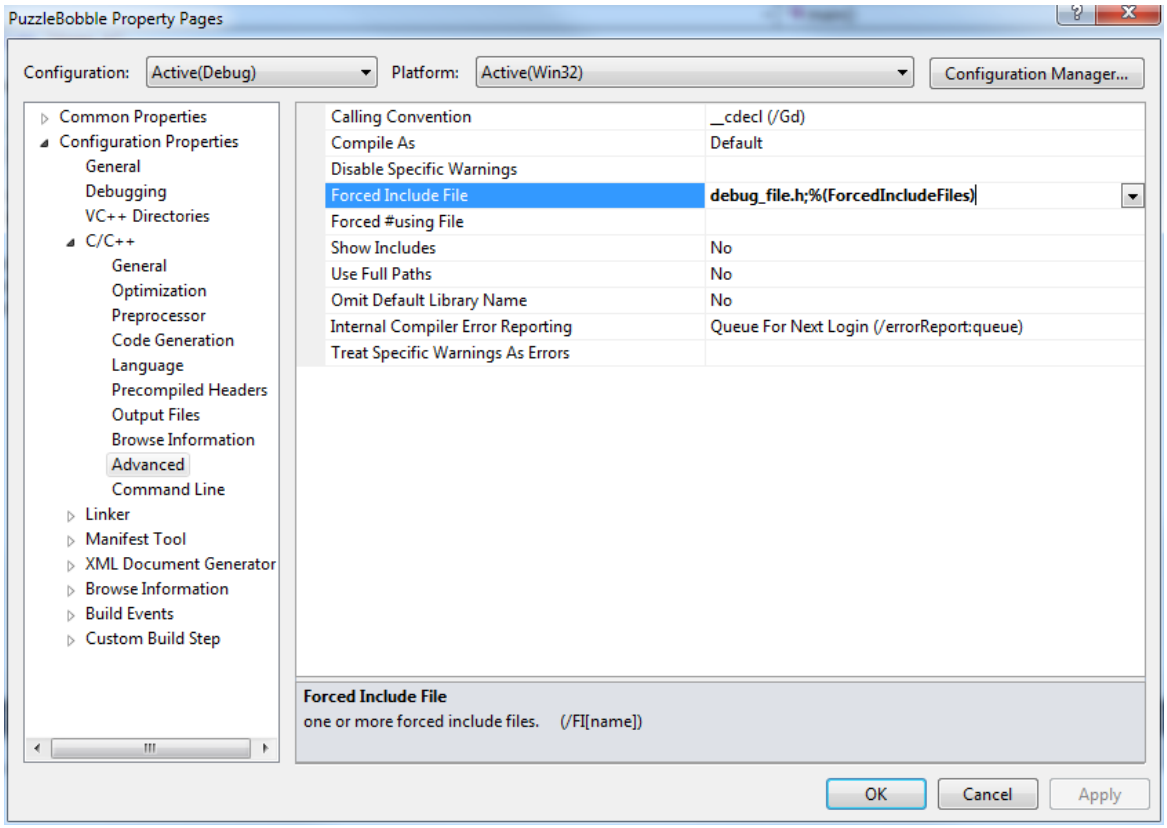
Om zeker te zijn dat geheugen slechts gedumpt wordt na het eindigen van de main methode en dus ook geen statische variabelen die nog in het geheugen zitten op dat moment te rapporteren, gebruik je in plaats van _CrtDumpMemoryLeaks() op het einde van je programma, beter volgende vlag in het begin van je programma:

```
int main()
{
        // Debug flag
        _CrtSetDbgFlag ( _CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF );

        // Programma code
}
```

(*) De **forced include** functionaliteit in Visual Studio kan je terugvinden bij de Project properties (rechtsklikken op je project in de Solution Explorer), onder:

Configuration Properties > C/C++ > Advanced > Forced Include File



[i] https://en.wikipedia.org/wiki/Advance_Wars