Technisch verslag

Door:
Joeri Kok,
Joris Heemskerk,
Koen Brave &
Rick Horeman

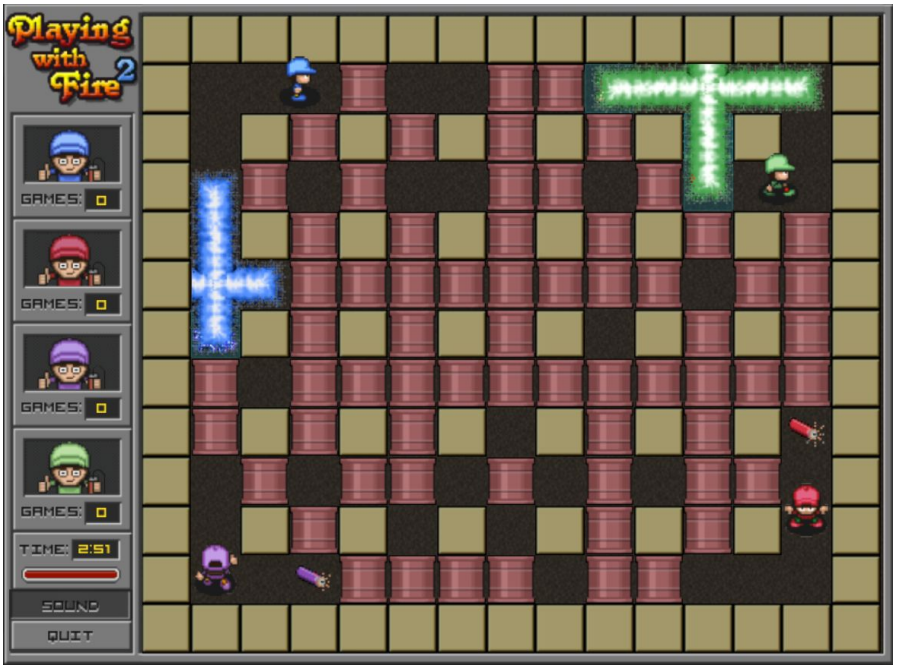
Inhoudsopgave

Inleiding

In jaar twee van de HBO-ICT opleiding op de Hogeschool Utrecht is een van de leukere projecten voor de studierichting TI (volgens studenten) de Themaopdracht Gaming.
De opdracht voor dit project was om een game te maken met behulp van de kennis die we tot nu toe hebben opgedaan gedurende onze opleiding. Het project moest in C++ geschreven worden en de visualisatie zou met behulp van de SFML library gedaan moeten worden.

Wij hadden als groep besloten de oude game Bomberman na te maken. Een game die we allen herkende uit onze jeugd. De versie van Spele.nl was hier voor ons de primaire bron van inspiratie. https://spele.nl/bomberman-4-spel/
We konden ons nog goed herinneren dat we dit spel op de basis- en middelbare school met onze vrienden op de schoolcomputers speelde, pure nostalgie.

Het idee van de game is als volgt. Je speelt het spel met maximaal vier personen op een speelveld. Elke speler heeft een oneindig aantal bommen en kan om de zoveel tijd een bom neerleggen die na een bepaalde tijd explodeert. Als de bom explodeert komt er een explosie in alle windrichtingen die na drie tegels (standaard) stopt. Als de explosie een opblaasbaar object tegenkomt, bijvoorbeeld een speler of breekbare muur, zal dit object worden opgeblazen en de explosie hier stoppen.
Het doel van het spel is om alle andere spelers op te blazen en zelf als langst in leven te blijven. Deze speler wint dan het spel.

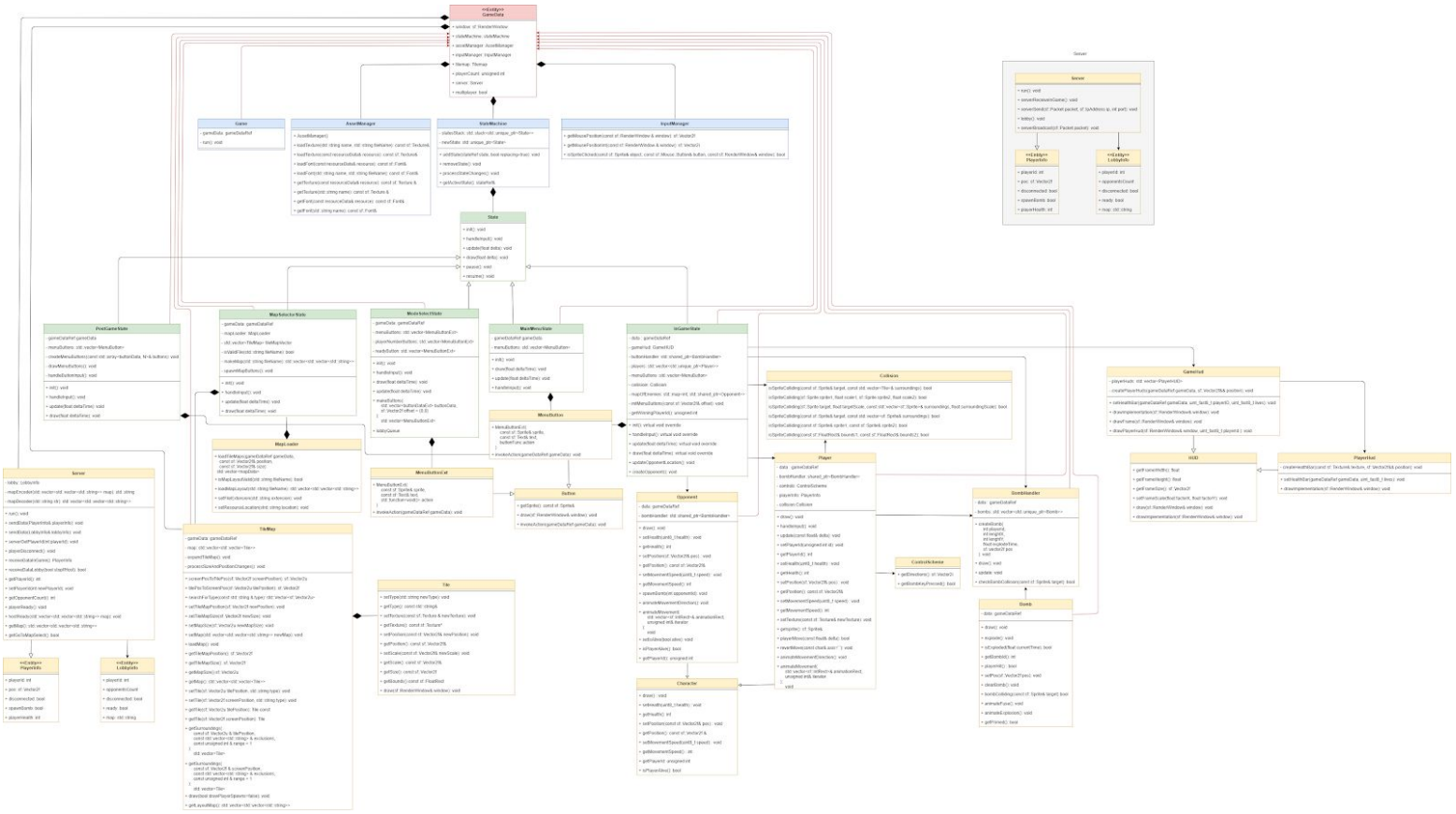


(afbeelding van de bomberman versie op spele.nl)

Naast het doel om het spel na te maken hadden we ook van te voren besloten onszelf extra uit te dagen en te proberen een ster te halen voor dit project. We zouden dit de doen door middel van het toevoegen van een zeer uitdagende feature. We waren van plan een online multiplayer optie toe te voegen.
In hoeverre het ons gelukt is dit toe te voegen is verderop te lezen.

In dit technische verslag zullen we ons klassendiagram laten zien en zullen we uitleggen hoe we tot bepaalde keuzes zijn gekomen. Ook hebben we doxygen toegevoegd aan onze files, waar dit te vinden is staat in hoofdstuk *Doxygen*.

Klassendiagram



(de png staat ook los in de github, gezien de tekst niet te lezen is en sommige pijlen niet goed zichtbaar zijn)

We hebben naar aanleiding van ons project een klassendiagram gemaakt. Hierin hebben we twee aparte systemen weergegeven, de game zelf en de server (rechtsboven in het aparte vakje).

Het belangrijkste object is de GameData (aangegeven in het rood), dit is een entity object waar heel veel andere classes dingen uit nodig hebben.

Dan hebben we de game engine, dit zijn de blauwe classes. Hier wordt geregeld dat verschillende states met elkaar op een stack kunnen werken, dat er input afgehandeld kan worden en dat er textures globaal ingeladen kunnen worden.

Vervolgens zijn alle groene classes aan de beurt. Dit zijn allemaal State classes of instanties van State.

De groene classes hebben weer pijlen naar en van de gele classes. Dit zijn alle overige classes. We hebben hier verder geen onderscheid in gemaakt.

Design keuzes

Game engine

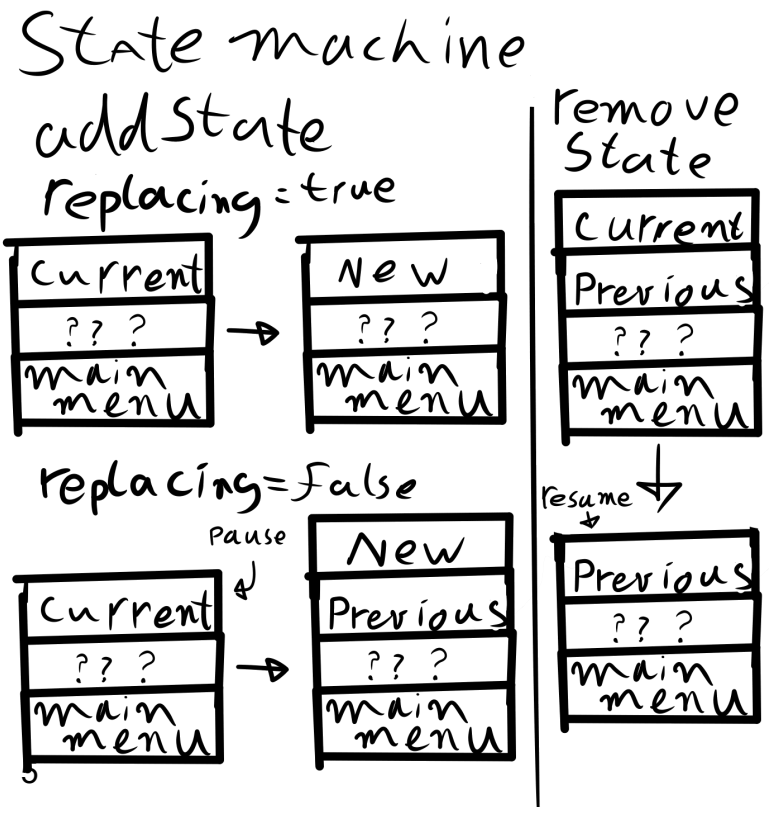
De game engine bestaat uit de volgende onderdelen:

- asset manager
- input manager
- state machine
- state(s)
- game loop
- gamedata

De asset manager heeft een functie om textures en fonts uit bestanden te openen en deze (met een naam) op te slaan in een map. Ook is er een functie om deze assets vervolgens weer te laden waar je ze nodig hebt met de gegeven naam. Het mooie hieraan is dat je één keer centraal een texture kunt laden die vervolgens misschien wel 50 keer gebruikt wordt door een tile, zo neemt hij maar één keer geheugen op in het VRAM en hoeft het, relatief langzame, uitladen uit een bestand maar één keer te gebeuren.

De input manager is bedoeld als centrale plek om gebruiker invoer af te vangen. Zo zijn er functies om de muis positie op te vragen en een functie die checked of er op een bepaalde sprite geklikt wordt. Dit kan uitgebreid worden om ook toetsenbordinvoer af te vangen (en nog meer), maar dit doen wij in praktijk direct via SFML.

De state machine heeft de taak om de verschillende states waarin het spel kan zijn te controleren. Hiervoor heeft de state machine een stack waar de states opgezet kunnen worden. Om naar een nieuwe state te gaan kan je de addState functie aanroepen om een nieuwe state op de stack te zetten, deze functie heeft ook een boolean parameter die aangeeft of de huidige state vervangen moet worden, of dat de nieuwe state daar bovenop geplaatst moet worden.
Om terug te gaan naar de vorige state (dit kan uiteraard niet als die vervangen is) kan je de removeState functie aanroepen, deze haalt



simpelweg de bovenste state van de stack.
Wanneer er een nieuwe state op de stack wordt gezet roept de state machine eerst de pause functie aan van de huidige state aan, dan kunnen dingen eerst netjes gepauzeerd worden, mocht dat nodig zijn.
Wanneer er een state van de stack wordt gehaald, wordt de resume functie van de state daaronder (weer) aangeroepen om netjes verder te kunnen gaan, mocht dat nodig zijn.
Daarnaast is er ook nog een functie die een referentie naar de huidig actieve state teruggeeft.

De state klasse is een basis voor elke state in het spel. Om een nieuwe state te maken kan je een klasse maken die afgeleid is van de state en implementaties toevoegen voor de verschillende functies: init, handleInput, update, draw, en optioneel; pause en resume.
De update en draw functies krijgen van de game loop de verstreken tijd sinds de vorige loop mee, hiermee kan er “framerate independent movement” gemaakt worden, anders zou een iemand die 60fps krijgt twee keer zo snel kunnen lopen als iemand met maar 30fps.

De game loop heeft als taak om telkens de handleInput, update en draw functies van de huidig actieve state aan te roepen. Daarnaast zorgt de game loop ervoor dat de draw functie niet vaker wordt aangeroepen dan nodig/gewenst. In plaats daarvan blijft hij heel vaak de handleInput en update functies aanroepen om input en beweging responsive te houden zonder onnodig veel last op de GPU te leggen.
De run functie van de game klasse hoeft slechts één keer aangeroepen te worden in de main, daarna blij hij zelf de loop uitvoeren.

De gamedata is een struct met centrale instanties van de state machine, asset manager en input manager, een SFML render window en wat centrale variabelen voor de game, waaronder een tilemap.

Collision

Het collision mechanisme van het spel is vrij straight forward. De collision klasse bestaat uit een aantal functies die controleren of verschillende drawable SFML objects met elkaar overlappen. Dit komt uiteindelijk neer op de global bounds van een drawable pakken en daar de intersects functie op aanroepen.

Er zijn een aantal verschillende functies aangemaakt die het gebruik van het checken op collisions gemakkelijker maken. Zo is er bijvoorbeeld een overload aangemaakt voor de isSpriteColliding() functie waaraan een lijst van sprite objects meegegeven kan worden samen met een losse sprite. Vervolgens wordt er door deze lijst van sprite object heen gelopen om te zijn of de losse sprite met een van deze objecten intersect. Deze functie was in het bijzonder handig om te koppelen aan de getSurroundings() functie van de TileMap klasse, aangezien de TileMap klasse, afhankelijk van een bepaalde range en positie, een lijst van sprites teruggeeft.

In de Sprite klasse worden er geen variabelen of states bijgehouden waardoor het in principe niet nodig is om een apart object aan te maken voor het detecteren van collisions. Toch is het makkelijker om hier alsnog een aparte klasse voor te gebruiken omdat op die manier de mogelijkheid om de Sprite klasse uit te breiden open blijft.

Map Loader

De map loader is een klasse die is ontstaan nadat het duidelijk werd dat de Map Selector klasse te groot begon te worden. Ideaal gezien heeft een klasse een enkele verantwoordelijkheid en bij het reviewen van de Map Selector klasse werd het duidelijk dat de Map Selector klasse te veel verantwoordelijkheden op zich nam. Op dat moment is er toen het besluit genomen om de Map Selector klasse op te splitsen in twee verschillende klassen.

Het laden van de de verschillende mappen wordt nu door de Map Loader afgevangen. De klasse heeft de mogelijkheid om een folder locatie en een file extension mee te krijgen waarin gezocht zal worden naar verschillende mappen die de spelers van het spel kunnen spelen. Standaard wordt de locatie en file extension gebruikt die in het definitions.hpp bestand worden gespecificeerd. Bij het laden van een map wordt er gekeken of de map geldig is. Als dit het geval is, dan wordt er gelijk een TileMap object aangemaakt aan de hand van de zojuist ingeladen map layout.

De Map Selector communiceert met de Map Loader en houdt een lijst van TileMap objecten bij. Zodra het maximum aantal mappen geladen is, dat overigens ook

gespecificeerd wordt in het definitions.hpp bestand, dan is het laden van de mappen voltooid en wordt er een preview weergave van de map op het scherm getoond. De gebruiker kan op die manier een duidelijke keuze maken tussen de verschillende mappen die geladen zijn.

Buttons

Door het gehele spel heen wordt er gebruik gemaakt van buttons die interacteren met de speler en de mogelijkheid bieden om te schakelen tussen de verschillende game states. Om te zorgen dat de buttons gemakkelijk gebruikt kunnen worden is er een aparte klasse aangemaak voor de buttons.

De belangrijkste eigenschappen van een button is dat het een titel moet hebben en een action. Daarnaast heeft een button ook een sprite object waar een texture op weergeven kan worden en een text object voor het renderen van de text op een button. Om het gebruik van de buttons zo flexibel mogelijk te houden, bestaat er een abstracte Button klasse met een draw() en een abstracte invokeAction() methode.

Oorspronkelijk was het alleen nodig om een simpele actie uit te kunnen voeren zodra er op een button geklikt wordt. Hier is een MenuButton klasse voor aangemaakt die een action opslaat aan de hand van een simpele function pointer. Gedurende het project begon er ook vraag te ontstaan om complexere acties uit te voeren wanneer er op een button geklikt wordt. Om dit handig af te kunnen vangen is er gebruik gemaakt van een aparate derived klasse MenuButtonExt. Deze aanvullende klasse maakt gebruik van een std::function object om een action te kunnen gebruiken die lambda captures ondersteunt.

```
18    constexpr std::array buttons{
19        buttonData{"Play", Util::switchState<ModeSelectState>},
20        buttonData{"Exit game", [](gameDataRef gameData){gameData->window.close();}}
21    };
```

Voorbeeld van het aanmaken van buttons.

Aan de hand van de buttonData array wordt er een lijst van button objecten aangemaakt. Op die manier kan er heel gemakkelijk een nieuwe button worden toegevoegd. Het enige dat opgegeven hoeft te worden is een titel een actie.

Button Factory

Het aanmaken van verschillende buttons wordt op dit moment gedaan met behulp van een simpele for-loop die door de buttonData objecten heen loopt, een sprite en een texture aanmaakt en vervolgens de scaling/positie zodanig aanpast zodat de nieuwe button goed op het scherm past. Zie ook het voorbeeld hieronder.

```
22      for (std::size_t index = 0; index < buttons.size(); ++index) {
23          static const auto& texture = gameData->assetManager.getTexture("default button");
24          auto sprite = sf::Sprite{texture};
25          sprite.setScale(windowSize / texture.getSize() / sf::Vector2f{5, 10});
26          const auto& spriteBounds = sprite.getGlobalBounds();
27          sprite.setPosition(Util::centerRectMargin(windowSize, spriteBounds, index, buttons.size()));
28
29          static const auto& font = gameData->assetManager.getFont("default font");
30          auto text = sf::Text{buttons[index].title, font};
31          text.setFillColor(Resource::globalFontColor);
32          text.setOrigin(Util::scaleRect(text.getGlobalBounds(), {2, 2}));
33          text.setPosition(Util::centerVector(sprite.getPosition(), spriteBounds, {2, 2.2}));
34
35          menuButtons.emplace_back(std::move(sprite), std::move(text), buttons[index].action);
36      }
```

Aanmaken, positioneren en scalen van een reeks buttons.

Ideaal gezien wordt hier een soort van factory klasse voor gebruikt zodat andere klasse op gemakkelijke wijze nieuwe buttons kunnen aanmaken. Hier is dit weekend nog een poging aan gewaagd maar er was niet genoeg tijd om deze factory klasse volledig in gebruik te nemen door de gehele code base heen. De Button Factory klasse kan in GitHub teruggevonden worden onder de de maintenance/menuInterface branch. Hieronder nog een voorbeeld van hoe de factory klasse gebruik zou kunnen worden.

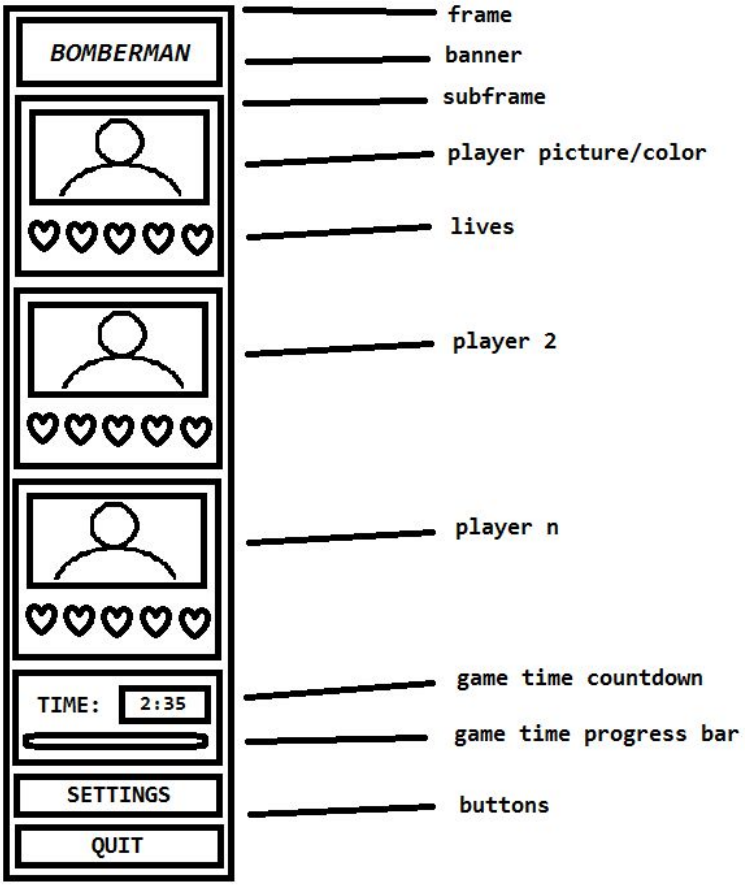
```
61      menuButtons = gameData->buttonFactory.createButtonsHorizontal<MenuButton>(
62          gameData,
63          std::vector{
64              ButtonData{"Play Again", [](gameDataRef gameData){
65                  gameData->tileMap.loadMap();
66                  Util::replaceState<InGameState>(gameData);
67              }},
68              ButtonData{"Main Menu", [](gameDataRef gameData){
69                  gameData->stateMachine.removeState();
70              }}
71          },
72          85/100.f
73      );
74  }
75
```

Voorbeeld gebruik Button Factory

HUD

De HUD, ook wel heads-up display genoemd, is de status bar van het spel waar informatie over de spelers wordt weergegeven, zoals levens, score etc. Het is belangrijk dat de HUD actuele informatie weergeeft op het scherm zonder de spelers te veel af te leiden van de gebeurtenissen die in het spel plaatsvinden. Wanneer een nieuwe game wordt gestart, wordt de HUD geladen en op het scherm getoond.

De HUD is opgedeeld in twee verschillende onderdelen, de game HUD en de player HUD. Bij het starten van een nieuwe game wordt de game HUD geladen die bestaat uit een frame, een banner en een x aantal player HUD's, afhankelijk van de hoeveelheid spelers die aan een game meedoen. Elke player HUD heeft ook een frame en bestaat daarnaast nog uit een profile picture en een healthbar. Zie de mockup hieronder om een duidelijk beeld te krijgen van de samenstelling van de complete HUD.



Mockup van de game HUD met daarop ook de player HUD en aanvullende buttons.

In de mockup is te zien dat de game HUD ook nog uit een game timer bestaat waarmee de tijd aangegeven kan worden hoe lang een spel duurt. Door tijdgebrek zijn we hier echter niet meer aan toegekomen. Dankzij de manier waarop de game HUD klasse is opgebouwd, is het uitbreiden van extra onderdelen vrij eenvoudig.

Aangezien er een aantal overeenkomsten bestaan tussen de game HUD en de player HUD, is er gekozen om een algemene base klasse aan te maken genaamd HUD. Deze base klasse heeft bevat een sprite waar de frame van een HUD mee op het scherm getekend kan worden. Vanwege het gebruiksgemak is er gekozen om het NVI-pattern toe te passen. De base klasse tekent altijd een frame op het scherm en roept daarna de abstracte functie drawImplementation() aan om de derived klassen eventuele andere objecten te laten tekenen.

Bij het initialiseren van de HUD's worden de benodigde sprites gescaled afhankelijk van de grootte van geladen textures en de grootte van het scherm. Op die manier blijft alles netjes op scherm staan indien er later alsnog besloten wordt om andere textures te gebruiken voor de HUD's of om een andere dimensie voor het game scherm. Bij het aanmaken van de game HUD kan er een positie meegegeven worden waarmee bepaald wordt waar de HUD op het scherm wordt aangemaakt. De positie duidt de top-left corner van de HUD aan. De hoeveelheid player HUD's die de game HUD aanmaakt wordt bepaald door de playerCount variabele uit de GameData structure.

Bij het aanmaken van de player HUD's wordt er een player ID meegegeven en een positie waar de HUD op het scherm moet worden weergegeven. De player ID is nodig om de bijbehorende profile picture te laden. Door iedere speler een andere profile picture mee te geven is het makkelijk voor de speler om elkaar te kunnen onderscheiden. De player HUD klasse bevat daarnaast nog een functie voor het aanmaken van de health bar en het wijzigen van de health bar. De health bar is ook dynamisch omgebouwd waardoor het gemakkelijk is om het maximale aantal levens aan te passen zonder dat de vormgeving van de HUD aangepast hoeft te worden.

Met de setHealthBar() functie kan het aantal levens dat een player HUD moet weergeven bepaald worden. De game HUD houdt een lijst bij van alle player HUD's en kan de health bar van een bepaalde player HUD aanpassen afhankelijk van de player ID van buitenaf wordt meegegeven.

Door de manier waarop het lobby mechanisme van de game server werkt, ontstond er de vraag om minder player HUD's te tekenen dan dat er worden aangemaakt. Dit is de reden dat de drawFrame() en drawPlayerHud() geïmplementeerd zijn. Deze functies maken het mogelijk om in plaats van alle sprites van de game HUD, alleen de gewenste sprites te tekeken.

Makefile

Voor dit project hebben we een aangepaste makefile moeten gebruiken om de gehele code base op een handige manier te kunnen compilen. Aangezien niet ieder groepslid gebruik maakt van dezelfde code editor/IDE, is er enigszins een splitsing ontstaan in de manier waarop de code base gebuild wordt. De twee verschillende situaties kwamen op het volgende neer:

Methode 1

Deze methode werd door Visual Code gebruikers gebruikt en bestond uit een constructie waarbij de makefile niet aangepast hoefde te worden wanneer een nieuwe source file werd aangemaakt. Het nadeel was echter wel dat alle object files opnieuw opgebouwd moesten worden bij iedere compilatie.

Methode 2

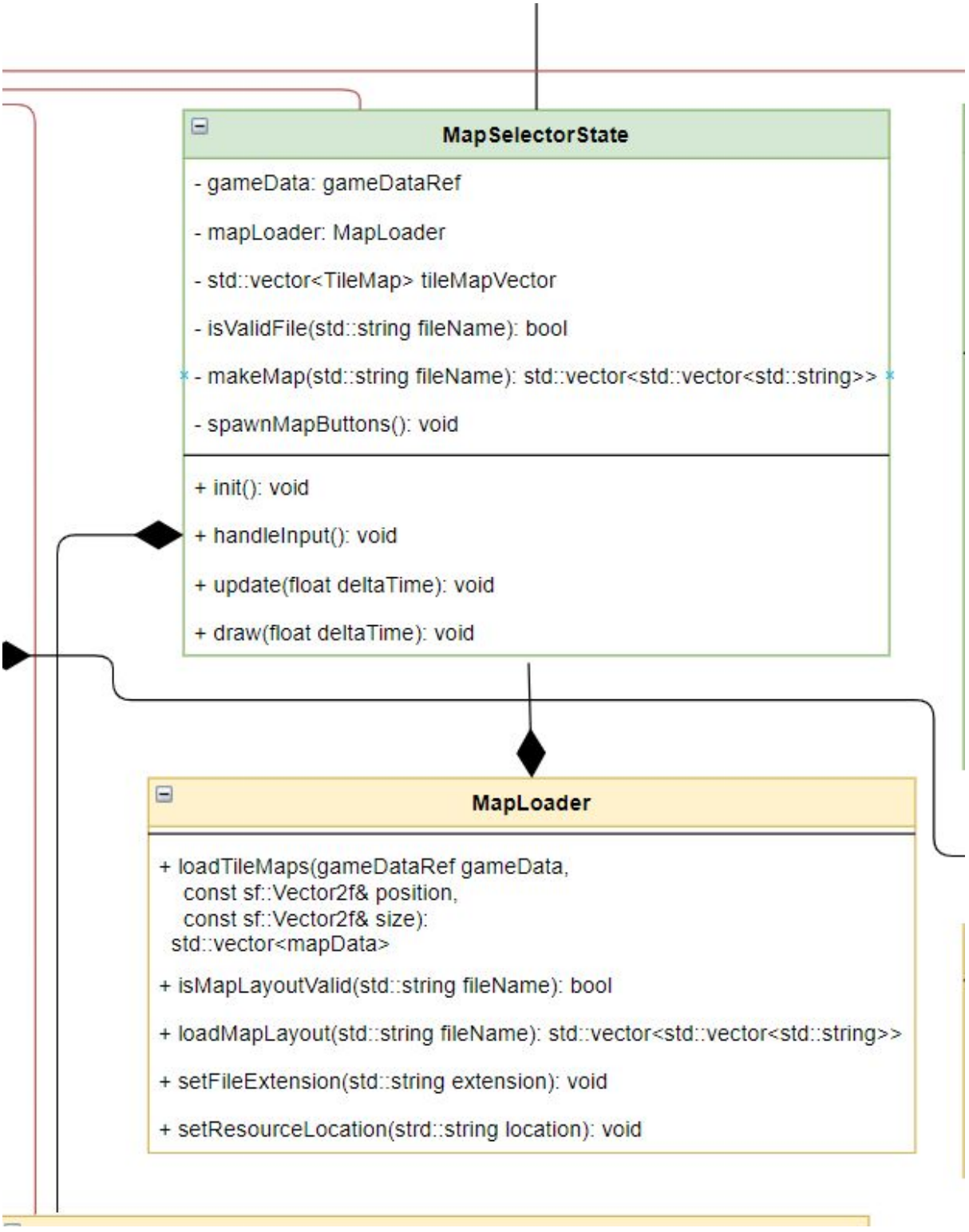
Deze methode werd door de Sublime gebruikers gebruikt en bestond uit een constructie waarbij de makefile handmatig aangepast moest worden voor iedere nieuwe source file die aan het project werd toegevoegd. Alleen de object files waarvan de bijbehorende source en/of header file was aangepast hoefde opnieuw gebuild te worden waardoor het compileren heel snel ging. Het toevoegen van nieuwe targets in de makefile was op zich nog niet zo'n probleem maar het onderhouden van het geheel werd uiteindelijk heel lastig en onoverzichtelijk.

Deze situatie heeft er toe geleid dat er het laatste weekend gewerkt is aan een makefile die niet meer met de hand aangepast hoeft te worden en alleen nieuwe objects build die ook daadwerkelijk nodig zijn. Hieronder een voorbeeld daarvan:

```
43  # extract file names from all CPP files within the SRC_DIR directory
44  # and replace the .cpp extensions with .o
45  targets := $(patsubst %.cpp,%.o,$(notdir $(wildcard $(SRC_DIR)/*.cpp)))
46
47  # search through the HPP and CPP file specified by the $1 (first) argument and
48  # only return the strings that start with '#include "' (note the double quote)
49  find_includes = findstr 2^>nul /b /i /c:"\#include "^" "$(HDR_DIR)\\$1.hpp" "$(SRC_DIR)\\$1.cpp"
50
51  # walk through the output of the find_includes command, returning the 2nd token
52  # delimited by double quotes while only keeping the file name and extension
53  find_dependencies = for /f tokens^=2^ delims^=^" %%e in ('$(find_includes)') do echo %%~nxe
54
55  # creates a recipe for a target based on the 1st argument it receives and runs
56  # the find_dependencies command to look for all corresponding header files
57  define create_target_recipe
58      $1.o: $1.cpp $(shell $(find_dependencies))
59          g++ $(CFLAGS) -c $$< -I $(SFML_INC) -o $$@
60  endef
61
62  build: $(targets)
63      g++ $(CFLAGS) $(targets) -o $(OUT_FILE) -L $(SFML_LIB) $(SFML_STL) $(WIN_STL)
64
65  # iterate over all the targets found and create a recipe for each of them
66  $(foreach target, $(basename $(targets)), $(eval $(call create_target_recipe,$(target))))
67
68  clean:
69      del $(targets) 2>nul
70
71  .PHONY: build clean
72
```

Dynamische makefile, meer informatie:
https://github.com/BraveKoen/Bomberman/blob/maintenance/menuInterface/makefile

Map selector



Bij het maken van de MapSelectorState class hebben we rekening gehouden met eventuele uitbreiding van onze game. We hebben deze class gemaakt met in ons achterhoofd een eventuele map editor waarin een nieuwe map gemaakt zou kunnen worden.

We hebben daarom gekozen de map op te slaan in txt bestanden en deze in de MapSelectorState uit te lezen zodat een eventuele map editor een apart programma zou kunnen zijn die deze bestanden aan zou kunnen passen.

We zijn er niet aan toegekomen om een map editor te maken. Toch kan de gebruiker de map files makkelijk aanpassen. We hebben namelijk een tile map layout gebruikt in txt files (zie hieronder).

resources > maps > map2.txt

```
1   solid solid solid solid solid solid solid solid solid solid solid solid solid solid solid
2   solid spawn empty empty empty empty break empty empty empty empty empty empty spawn solid
3   solid empty solid break solid break solid empty solid empty solid empty solid empty solid
4   solid empty empty empty break break break empty empty empty empty break empty empty solid
5   solid empty solid empty solid break solid empty solid empty solid empty solid empty solid
6   solid break empty empty empty break break empty empty empty empty break empty empty solid
7   solid break solid empty solid break solid empty solid break solid empty solid empty solid
8   solid empty empty empty empty break empty break empty break break empty empty break solid
9   solid empty solid empty solid break solid empty solid break solid break solid break solid
10  solid break empty empty break empty empty empty empty empty empty empty empty break solid
11  solid break solid empty solid break solid break solid break solid empty solid break solid
12  solid break empty empty empty empty empty break empty empty break break empty empty solid
13  solid break solid empty solid empty solid break solid break solid break solid empty solid
14  solid spawn empty empty empty empty empty empty empty empty empty break empty spawn solid
15  solid solid solid solid solid solid solid solid solid solid solid solid solid solid solid
```

Elk vakje in het spel wordt aangeduid door een van de volgende 5 letter woorden:
-solid (een onbreekbare muur)
-empty (een leeg vakje)
-spawn (een plek voor een player om te spawnen)
-break (een opblaasbare muur)

Bij het gaan naar de MapSelectorState wordt gekeken welke files er staan in de map folder. Van al de txt bestanden in deze folder wordt gekeken of de format valide is; of de enige woorden die er in voorkomen (gescheiden door spaties) een van de vier bovenstaande mogelijkheden zijn.
Als de file valide is maakt hij de map aan met behulp van de TileMap klasse (hierover meer uitleg in het volgende kopje).
Ook wordt de naam van de txt file in de button gezet. Een te lange naam wordt afgekort en er wordt “…” achter gezet.

Tile map

De tilemap is een klasse met een twee dimensionale vector van tiles (een klasse, zie hieronder).
Er zijn te veel tilemap functies om hier allemaal te benoemen, dus zie voor omschrijving per functie de doxygen.
Het is mogelijk om op verschillende manieren op te vragen welke tile waar zit en om de tiles aan te passen, je kunt zelfs in één keer alle tiles aanpassen door een hele nieuwe map er in te gooien. Daarnaast heeft de tilemap een eigen draw functie die alle tiles tekent.

Tiles

De tile klasse is vrij simpel, met een string als type en een sprite. Beide hebben setters en getters en de klasse heeft een draw functie.

Player

Opponent en Player zijn beide een inheritance van Character. Helaas moet ik toegeven dat hier niet correct gebruik van is gemaakt. Om het correct te doen hadden we de variabelen die player en opponent beide hebben in de character class moeten doen.

Bij de player class wordt er rekening gehouden met player input. Ook wordt er gekeken of de player tegen muren aanloopt of in een bomb staat.

Bij de opponent wordt er niet veel bijgehouden door de client zelf. De opponent wordt constant geupdate door data dat binnenkomt vanuit de server. Dus op het moment dat de health van de player omlaag gaat wordt dat verzonden en bij de andere clients wordt de health geupdate.

Server

```
struct PlayerInfo{
    int      playerId;
    sf::Vector2f pos;
    bool disconnected;
    bool spawnedBomb;
    int playerHealth;
};
```

```
struct LobbyInfo{
    int playerId;
    int opponentsCount;
    bool disconnected;
    bool ready;
    std::string map;
};
```

De server kent twee data structs: PlayerInfo en LobbyInfo. Op het moment dat de server wordt opgestart, wordt er gebruikt gemaakt van LobbyInfo. Bij het ontvangen wordt het pakketje omgezet naar de LobbyInfo struct zo kan de server uitlezen wat er moet worden gedaan met de ontvangen data. Op het moment dat de server een LobbyInfo.ready binnenkrijgt van de host, wordt er een broadcast gedaan naar alle spelers dat ze kunnen beginnen met de game. Dus de host krijgt ook zijn eigen bericht binnen.

Op het moment dat de server naar de serverReceiveInGame functie gaat, wordt er ook gewisseld naar PlayerInfo. Als de server nu LobbyInfo zou binnenkrijgen kan die daar niks mee en zal die daar ook niks meer mee doen.

Multiplayer

Op het moment dat er wordt geklikt op de button “Join online” start er een aparte thread die bezig gaat met het ontvangen van data die binnenkomt. Parallel daarvan wordt er een bericht naar de server gestuurd dat de client wil connecten. De server zelf houd bij welk player nummer hij moet sturen. Als de client het nummer 1 terug krijgt wordt de MapSelectState gestart. Dit betekent automatisch dat je de host bent en jij dus bepaald wanneer de game gestart mag worden. Helaas zijn we er niet aan toegekomen om visueel te laten zien hoeveel mensen er op het moment geconnect zijn naar de server. Als de game wordt gestart zal elke server constant zijn locatie verzenden bij elke beweging die de player doet. Ook op het moment dat de player wordt geraakt door een bom wordt er nieuwe data verzonden.

Control schemes

Voor onze game wilde we het mogelijk maken om met een maximum van vier spelers op één toetsenbord te spelen. Dit zou betekenen dat we vier verschillende plekken op het toetsenbord moeten aanwijzen waar een speler (het liefst met één hand) zou kunnen bewegen met vier bewegings toetsen en een bom kunnen plaatsen met een vijfde toets.
De standaard control schemes zijn als volgt:

- speler 1: WASD + spatie
- speler 2: pijltjes + RCtrl
- speler 3: IJKL + AltGR
- speler 4: Num lock 8456 + Num Lock Enter

Voor de vierde speler is een Num Lock nodig, dit heeft niet ieder toetsenbord. Maar ook niet iedereen heeft de behoefte met vier spelers te spelen. Op een toetsenbord zonder Num Lock is het erg onhandig om vier control schemes te maken. De handen van de spelers zitten elkaar dan te snel in de weg.
Mocht iemand de standaard schemes willen aanpassen kan dat in de InGameState klasse in de InGameState::init functie.

Doxygen

Om de door ons geschreven code tot in detail uit te kunnen leggen zouden we een heel boek kunnen schrijven. Helaas is dat toch echt te veel moeite; voor ons om te schrijven, maar ook voor de beoordelaar om het terug te lezen. We hebben daarom, volgens de opdracht, onze code voorzien van doxygen kommentaar. De gegenereerde versie hiervan is te vinden in de github.

Uitbreiding

Na afloop van het project hebben we nog een aantal punten bedacht waar ons project nog op uitgebreid kan worden. Als we zelf nog verder willen met wat we tot nu toe hebben gemaakt of als iemand anders er later op uit wilt bereiden hebben we een lijstje van dingen die we zelf graag in de game terug zouden zien.

- Power-ups
  Als een speler een breekbare muur opblaast zou er een kans kunnen zijn dat er een powerup spawnt. Als een speler deze activeert door er overheen te lopen zou de powerup een effect kunnen hebben op de player.
  Bijvoorbeeld een power-up die de speler sneller laat lopen, een power-up die de speler een extra bom laat plaatsen (nog voor de vorige is afgegaan), een power-up die de speler onsterfelijk maakt voor een bepaalde tijd, etc.

- Teleporters
  Een paar van tiles die een speler die op een van de twee staat naar de ander verplaatst. Zo zou een speler van A naar B kunnen met maar één stap. Dit zou het ontsnappen aan spelers uitdagender kunnen maken.

- Map maker
  Een apart programma (of een aparte state) waar een user door middel van een grafische interface een map kan aanmaken en die vervolgens kan spelen in de game.
  Momenteel is dit mogelijk, door de txt bestanden met de hand aan te passen, maar dat is niet heel intuïtief voor gebruikers.

- Custom graphics
  Een manier voor een speler om de standaard textures voor de map, de achtergrond en de muren aan te kunnen passen. Dit zou bijvoorbeeld gerealiseerd kunnen door middel van een ingebouwde omgeving waar een gebruiker zelf iets in zou kunnen tekenen of een file in zou kunnen plaatsen.

- Audio
  Het toevoegen van audio aan de game. Het liefst verschillende audio voor bijvoorbeeld de menu’s dan in het spel zelf.
  Ook zou het leuk zijn een geluidje af te spelen als een speler geraakt wordt of als een bom ontploft.

- Splash screen
  Een startscherm met het logo van de game om een gebruiker het gevoel te geven dat de game aan het opstarten is.

- Statistieken
  Het toevoegen van statistieken van een gespeelde game. Nadat een speler het spel gewonnen heeft zou er een scherm moeten verschijnen waar niet alleen wordt weergegeven wie er gewonnen heeft maar ook wie welke welke speler heeft geraakt en hoe vaak. Dit geeft spelers het incentief om agressiever te spelen en zo veel mogelijk punten te scoren.

- Uitbreiding online multiplayer
  De vorm van online multiplayer heeft momenteel nog een vrij beperkte functionaliteit. Het zou leuk zijn als hier een lobby systeem wordt gemaakt waar elke speler kan zien wie er al zijn gejoined, welke map er geselecteerd wordt door de host en bijvoorbeeld welke kleur welke speler is.

- Character customisation
  Het maken van een apart programma of een aparte state waarin de gebruiker een andere sprite sheet mee kan geven om mee te spelen. Hiervoor zou de manier waarop animaties worden gemaakt op basis van de sprites moeten worden aangepast om ook andere formaten van spritesheets te kunnen accepteren.

- Schermgrootte aanpasbaar maken
  De optie toevoegen dat een gebruiker de grootte van het scherm zou kunnen aanpassen door de randen van het scherm te verslepen. Het overgrote deel van de code is al op een manier gemaakt dat de objecten worden geplaatst op basis van de afmetingen van het scherm. Echter hebben we niet genoeg tijd gehad dit voor alle objecten te updaten als een schermgrootte wordt aangepast.

Dit zijn enkele suggesties die we zelf zouden willen toevoegen maar helaas geen tijd voor hebben gehad. Mocht iemand die met deze code verder zou willen werken zelf ideeën hebben om toe te voegen verwelkomen we dit uiteraard met open armen.

Slot

We zijn enorm trots op het resultaat wat we neer hebben kunnen zetten. Ieder van ons heeft erg hard gewerkt om het product wat er nu staat te realiseren.

Van alle voorgaande projecten vonden we dit een van de leukste. Het project daagt je niet alleen uit om nieuwe dingen toe te passen en te leren, maar ook om de kennis die we in het blok dat bij het project hoort hebben opgedaan toe te kunnen passen. Zo hebben we dit project veel meer gelet op de verschillende patterns, algoritmes en de manier waarop we functies een naam geven zoals geleerd in ALDS en CPSE2. Maar ook hebben we nieuwe dingen geleerd, zoals bijvoorbeeld het werken met SFML, het maken van een server die data kan versturen of bijvoorbeeld het maken en animeren van grafische objecten.

Naast alle programmeer kennis die we hebben opgedaan binnen dit project hebben we ook beter leren omgaan met de documenterende taken. Zo hebben we excessief gewerkt met GitHub en we hebben meer geleerd over Scrum en hoe dat binnen een project wordt toegepast.

Kortom, we hebben zeer veel geleerd en het uitermate naar ons zin gehad gedurende de drie projectweken. We kijken uit naar volgende projecten waar we hopelijk net zo blij mee kunnen zijn als dit project.