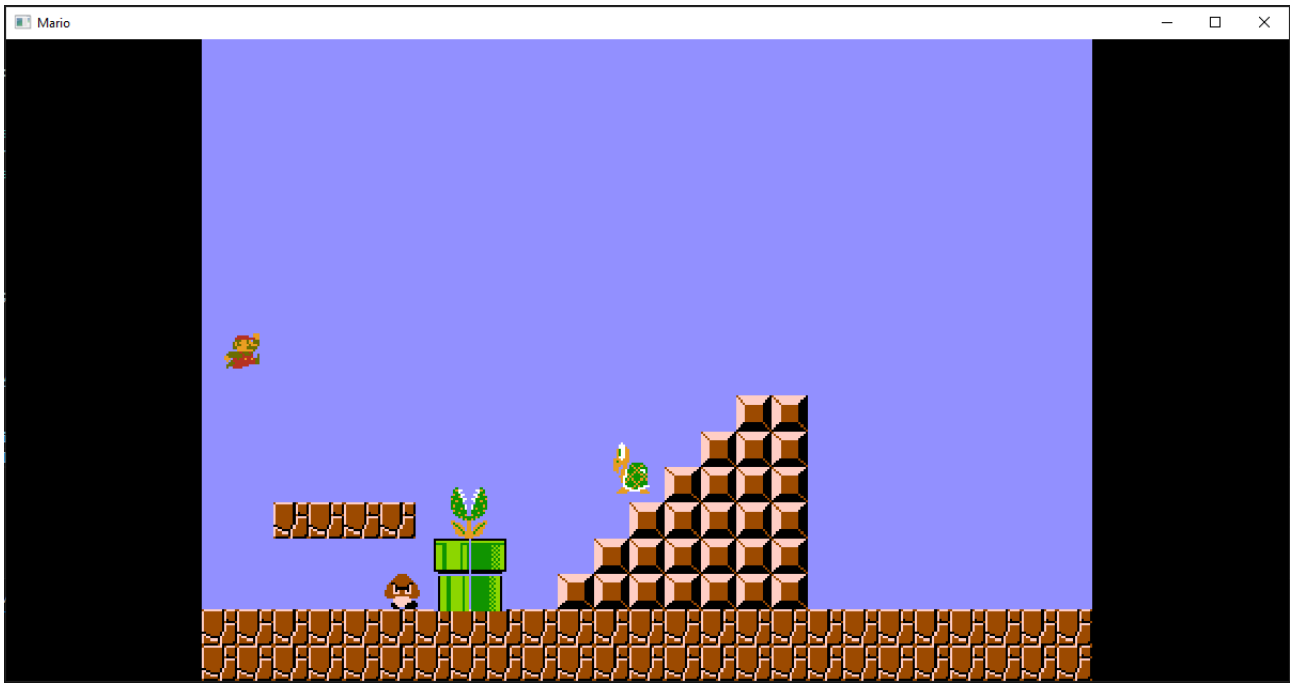
Super Mario Bros Lookalike



Indhold

Indledning

I denne synopsis redegøres, analyseres samt diskuteres implementeringen af et computerspil, som er stærkt inspireret af det originale Super Mario Bros computerspil fra 1985 udviklet af Nintendo til deres Nintendo Entertainment System (NES) spillekonsol. Programmet er skrevet i Rust som er et typestærkt system programmeringssprog med fokus på korrekt, hurtigt og læsbar kode. Formålet med projektet var at udvikle et større program interaktivt program i Rust. Der er brugt SDL2-biblioteket, image-biblioteket og futures-biblioteket. Der er indsat referencer til alle biblioteker i litteraturlisten.

Bemærk ingen assets er lavet af os. De er originale Nintendo assets fundet online.

Problemformulering

Hvordan kan man udvikle et program i Rust, der kan minde om det originale Super Mario Bros spil?

Kravspecifikation

For at udvikle og senere teste om programmet virker er følgende krav opstillet med en simplificeret MosCow-model:

Need to have:

- Spilleren skal kunne bevæge sig med WASD-tasterne
- Multithreading
- Kollision med fjender + objekter
- Tilemap generator
- Health system
- Spilleren skal kunne vinde, ved at hoppe ind i et flag (ligesom Mario)
- Sprites af objekter, så de ikke er kedelige og flade.
- Serialisering af objekter, så man kan gemme spillets tilstand mellem spilsessioner.

Nice to have:

- Animationer
- Lyd (Musik + lydeffekter)
- Multiplayer

Teori

Enums

Rust har indbygget understøttelse for noget som hedder ”tagged-unions”. En tagged-union er en struktur, som kan være en af en masse varianter afhængig af dens tag. En union-type er en type, hvor flere forskellige felter ligger i samme hukommelse i RAM. Overvej følgende kode skrevet i C++:

```cpp
#include <cstdint>

struct A {
  uint16_t a_data;
};

struct B {
  uint32_t b_data;
};

enum class ABTag : uint32_t { A, B };

struct AB {
  union {
    A a;
    B b;
  } data;
  ABTag tag;
};
```

Man skulle tro størrelsen af AB er 4 + 4 + 2 bytes = 10 bytes, men i virkeligheden er den kun 8, fordi A og B bliver opbevaret samme sted i hukommelsen. Havde der ikke været en union var størrelsen rigtig nok 10 bytes. Skal man manuelt implementere et tagged-union i C++, ville man starte med at definere sine to varianter som forskellige klasser med hver deres respektive fields. Herefter konstruerer man så en ny klasse som er overklassen over A og B (her kaldet for AB). Derudover laver man også en enum som indeholder det tag, der differentierer mellem de A og B, så man ved hvilken type man har at gøre med. Vil man så gerne have fat i A skal man ALTID huske at tjekke hvilket tag det er først, fordi det ikke er direkte bygget ind i sproget, beskytter compileren ikke en mod at lave dumme tastefejl.

```cpp
auto x = AB{ A(), ABTag::A };
if (x.tag == ABTag::A) {
  // x is of type A
} else if (x.tag == ABTag::B) {
  // x is of type B
}
```

Man kan altså tage fat i B selvom den data man har gemt i AB er af type A, fordi compileren ikke beskytter en. Dette kan resultere i spildte udviklingstimer for at finde fejlen og give brugeren en dårlig oplevelse, fordi det introducerer bugs. Dette kan man ikke i Rust! I Rust er denne konstruktion bygget ind i sproget. Man kan aldrig få fat i værdien, hvis den ikke findes. Det er umuligt! Compileren er efter dig! Prøver man at tilgå en af varianterne i en enum som man ikke har et if-statement udenom bliver compileren sur og skriver

at man skal tjekke alle muligheder. Desuden er der heller ikke hele den der work-around med at skulle definere 4 forskellige typer. Rust koden ville se ud som følgende:

```
enum AB {
    A(u32),
    B(u32),
}
```

Super simpelt og elegant og det kan aldrig være skyld i undefined behaviour. Bemærk at i Rust kaldes et tagged-union bare for et enum[1]. Men hvad kan de helt præcist og hvad kan de bruges til? I OOP bruger man ofte noget som hedder polymorfisme til at opbevare forskellige typer af data som alle deler noget logik, men alligevel ikke er helt ens. Et trivielt eksempel på dette kunne være, at man gerne vil have både en. Dog og Cat klasse i samme array i forbindelse med en dyrehandel, så man kan søge efter forskellige dyr. Her ville man bruge polymorfisme, men enums kan det samme. De er bare ikke helt så dynamiske. Når man har med enums at gøre, skal man kende alle de forskellige varianter som kan forekomme, fordi der skal oprettes en variant for hver af dem af compileren. Hvis man gerne vil have en type som er fuldt dynamisk, så skal man bruge polymorfisme. Man kan altså sige enums er en slags statisk polymorfisme. Modsat nedarvning og polymorfisme, så virker enums ”under the hood” som en betinget udførsel, mens polymorfisme er en dereferencing af en pointer og dermed et cache flush afhængig af, hvor objektet ligger i heapen. Dette betyder, at det er langt hurtigere at bruge statisk-polymorfisme versus dynamisk-polymorfisme, hvilket er vigtigt når man udvikler spil.

Statemachines

En statemachine er en abstrakt beregningsmodel som kan være i én af veldefineret række af stadier. Man kan gå mellem de forskellige med overgange eller transitions på engelsk. En statemachine er velegnet til programmer, hvor der er klar separation mellem et enkelt objekts rolle, når forskellige hændelser forekommer.[2]

Multi-threading

I har sikkert hørt, at når man skal ud og købe ny CPU i dag, skal man i stor grad overveje antallet kerner den CPU man er ved at købe har, hvor flere oftest er bedre.

I gamle dage havde man kun en enkelt kerne på en CPU. Koden i ens program bliver eksekveret på en kerne. En kerne kan kun køre en kun en enkelt instruktion ad gangen og er faktisk en mini-CPU inden i den store CPU. Dette betyder altså, at man i et system med en enkelt kerne, kun kan køre ét program ad gangen eller gøre én ting på samme tid[3].

En moderne CPU består af mange kerner (6+) og når et enkelt program udnytter ressourcerne på mange kerner, klassificeres dette program som multi-threaded. Multi-threading tillader at man kan køre mange beregninger på samme tid. F.eks. kan man have både Chrome og Visual Studio Code åben eller i et computerspil, på samme tid lave kollisionsberegninger, mens der læses input fra tastaturet. Man kan altså udnytte multi-threading til at få programmer til at køre hurtigere, fordi man kan parallelisere beregninger, så de kører på samme tid.

[1] (Rust-Lang, u.d.)
[2] (MDN, u.d.)
[3] (CMI, u.d.)

Når man bruger multi-threading skal man designe sine algoritmer så de virker på flere kerner. En godt designet algoritme kan ændre tidskompleksiteten, af en række af beregninger eller algoritme, til $\frac{O(n)}{c}$, hvor c er antallet af kerner i ens system. Dog er det oftest ikke helt så let, fordi der kommer en del ekstra arbejde i at holde styr på at dataene bliver sendt rigtigt frem og tilbage mellem de forskellige tråde[4].

Branch-prediction

En anden ting CPU'en kan gøre er at lave betinget-udførsel ud i fremtiden og på den måde øge antallet af instruktioner som kan blive eksekveret pr. sekund. Den prøver at gætte hvilken en af de to grene af programmet som tager. Hvis den gætter forkert, skal den arbejde sig baglæns, hvilket er langsomt, da CPU'ens kredsløb er pipelinede i forlængelse af hinanden[5] og den derfor bliver ned at resette og starte forfra.

Brugervejledning

Når man åbner programmet, er det vigtigt, at ens working directory er sat til root-mappen. Ellers er der fejl, når den prøver at indlæse de forskellige assets. Dette gøres automatisk, hvis man skriver "cargo run".

Derudover skal man åbne editoren sammen med spillet tilføres "--editor" flaget til programmets input argumenter gennem terminalen:

```
$ cargo run -- --editor
```

Figur 1 Kørsel af program, hvor editoren åbnes

Det er vigtigt at vide at man skal være fokuseret i det vindue man prøvet at klikke i. Derfor skal man trykke en gang, før vinduet begynder registrer input, hvis man lige har interageret med et andet vindue.

Game

I spillet kan man styre spilleren vha. A og D, som styrer retningen på Mario. Ved at hoppe ovenpå fjenderne kan man dræbe dem eller pacificere dem. Bortset fra den kødædende plante. Derudover kan man trykke på mellemrumstasten for at hoppe.

Editor

Editoren har også en del forskellige måder at interegere med.

Ved venstre-klik på et af de forskellige tiles, indsættes den nuværende valgt farve. Med højreklik kan denne fjernes igen.

Control + Z omgør den ændring man lige har lavet.

Control + O åbner en prompt så man kan indlæse et level (selve indlæsningen virker ikke).

Control + S gemmer et level på disken (ikke implementeret).

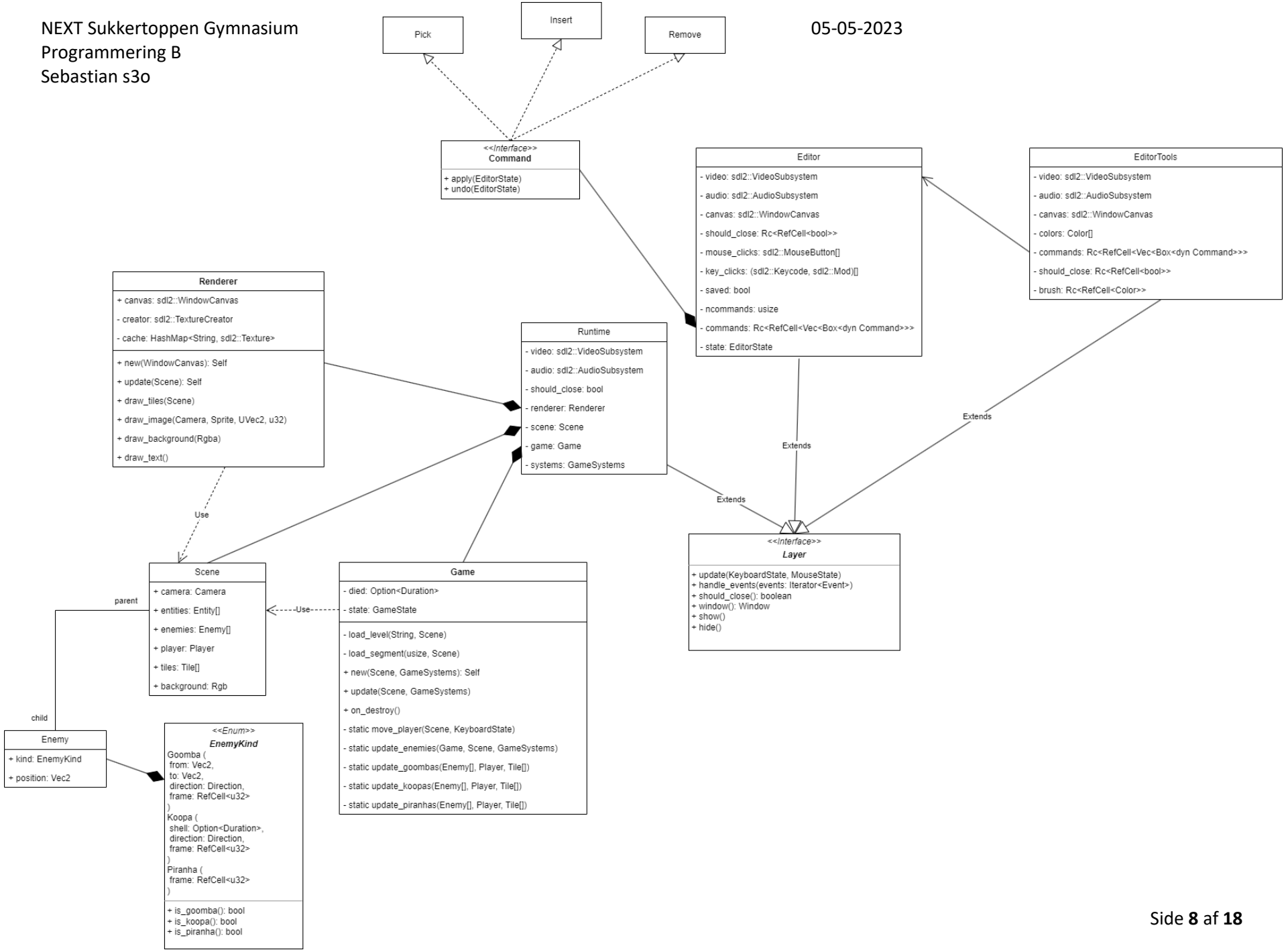
I " Tools"-vinduet kan man vælge en farve med musen ved at venstreklikke på den pågældende farve.

[4] (Bartleby, u.d.)
[5] (Computer Hope, u.d.)

Analyse

Programmet er opbygget i et event-loop som styrer programmet. Event-loopet er bygget ind i SDL2 og abstraherer over de operativsystem-specifikke måder at gøre tingene på. Derfor virker programmet på flere platforme. Det var meningen spillet skulle bestå af flere delapplikationer udover selve spillet.

Der skulle også have været implementeret en editor så man let kunne redigere i banerne. Derfor er der blevet designet og implementeret et layer-system som automatisk sender de forskellige events rundt til de applikationer de hører til. Dette gøres vha. af polymorfisme med ”Layer” interfacet. På følgende side ses et uddrag af de klasser der findes i programmet:



Figur 2 Uddrag af klassediagram

Man kan se på diagrammet, hvordan der er 3 over-klasser, der alle nedarver fra ”Layer”. Runtime står for selve spillet. Editor er editoren og EditorTools er et popup/ekstra vindue til at kunne vælge hvilken type af tile man skal indsætte i det level man redigerer I. Dette er også derfor der er en association mellem Editor og EditorTools, fordi når man vælger en ny blok, så ændrer man også i ”Editor”.

Editoren har et indbygget Undo-system, så man kan gå tilbage til forrige stadier. Dette er et must-have for alle editorer, så man ikke skal være bange for at eksperimentere med sine levels. Systemet virker ved at der er en række af kommandoer, som alle bruger dynamisk polymorfisme og nedarver fra ”Command”. Hver implementering af ”Command” ved både hvordan den skal udføres, men også hvordan man går tilbage fra en ændring. På den måde er det meget let, hvis man har et array af kommandoer at springe frem og tilbage i det ved at kalde hhv. ”apply”- og ”undo”-metoderne..

Selve spillet består af en ”Game”-klasse, hvor alt spillogikken forekommer, en ”Scene”-struktur til at kommunikere mellem ”Game” og ”Renderer”, samt en ”Renderer”, hvis job er at tegne scenen på skærmen. F.eks. kan man ændre baggrundens farve ved at ændre i ”Scene::background” eller tilføje flere fjender ved at skubbe et nyt element ind i ”Scene::enemies”. Game og Renderer har hver deres ”update”-metode som kaldes hver frame. Det er her al deres logik udspringer fra.

En klasse ikke vist på diagrammet er ”level”-klassen, som sørger for at serialisere og deserialisere alt relevant data om et enkelt level. Dette gør den ved at bruge biblioteket ”serde” som er Rusts go-to bibliotek til at lave seralisering[6] og deseralisering. Man bruger biblioteket ved at ”derive” Deserialize og Serialize på ens egen struktur og så finder biblioteket selv ud af at automatisk kunne gøre hhv. det ene og det andet.

```rust
#[derive(Debug, Clone, PartialEq, Deserialize, Serialize)]
pub struct Level {
    pub name: String,
    pub difficulty: Difficulty,
    pub start: Option<usize>,
    pub segments: Vec<Segment>,
}

#[derive(Debug, Clone, PartialEq, Deserialize, Serialize)]
pub struct Segment {
    pub spawn: Option<UVec2>,
    pub enemies: Vec<Enemy>,
    pub entities: Vec<Entity>,
    pub tiles: Vec<MapTile>,
    pub background: UVec3,
}
```

Figur 3 Opbygningen af "Level"-strukturen

[6] Serialisering og Deserialisering vil sige at skrive tal og værdier om til et andet format, så man kan gemme dataene på disk og så indlæse dem igen senere.

En bane er opdelt i segmenter for at repræsentere de individuelle dele af en hel bane. F.eks. når man går ind i et rør i Mario kommer man ind i en anden verden. På den måde bruges ikke ressourcer på at rendere de ting man alligevel ikke kan se.

Da alle typer af fjender er kendte på forhånd, bruges statisk polymorfisme i form af enums. Alle fjender har alle sammen en position, så dette felt er rykket ud af enumen og over i sin egen type som man kan se på klassediagrammet, men udover dette er alle de forskellige data unikke. F.eks. har Goombaer en defineret sti de skal følge frem og tilbage. Denne særlige logik håndteres i den tilsvarende ”update_goombas”-funktion. Faktisk har alle de forskellige fjendevarianter en dedikeret ”update”-funktion som håndterer logikken der tilhører dem. Alle disse funktioner kaldes fra en enkelt funktion ”update_enemies”, som sørger for at alle de forskellige fjender bliver opdateret i en enkelt frame.

”update_enemies” benytter multi-threading til at formindske tiden det tager at opdatere fjenderne. Der benyttes en thread-pool fra futures biblioteket, som tillader at sende data til andre tråde. Det er en dyr operation at oprette en ny tråd, så derfor laves en hel masse på forhånd (i en pool), som bare venter på, at der bliver brug for dem. Dette betyder, at det kun er første frame i spillet, som er langsommere, fordi den skal oprette alle trådene.

I nedenstående funktion startes med at hive ”thread_pool”-field ud af den samlede GameSystems struktur. Derefter filtreres de samtlige de forskellige fjender ud som er af goomba-varianten med ”filter”-funktionen.

```rust
pub fn update_enemies(game: &mut Game, scene: &mut Scene, systems: &GameSystems) {
    let GameSystems { thread_pool, .. } = systems;

    let updated_goombas = {
        let goombas: Vec<_> = scene
            .enemies
            .iter()
            .filter(|el| el.is_goomba())
            .cloned()
            .collect();

        let player = scene.player.clone();
        let tiles = scene.tiles.clone();
        async move { Self::update_goombas(goombas, &player, &tiles) }
    };

   /// ...
}
```

Figur 4 Uddrag af "update_enemies"-funktionen som viser, hvordan der oprettes en future

Derefter klones både player og de indsatte tiles i scenen. Dette er den overhead som oftest opstår, når man laver multi-threading. Derefter laves en future med ”async move” nøgleordene. En future repræsenterer en værdi som er beregnet i fremtiden. Det samme princip gøres for alle fjendevarianterne.

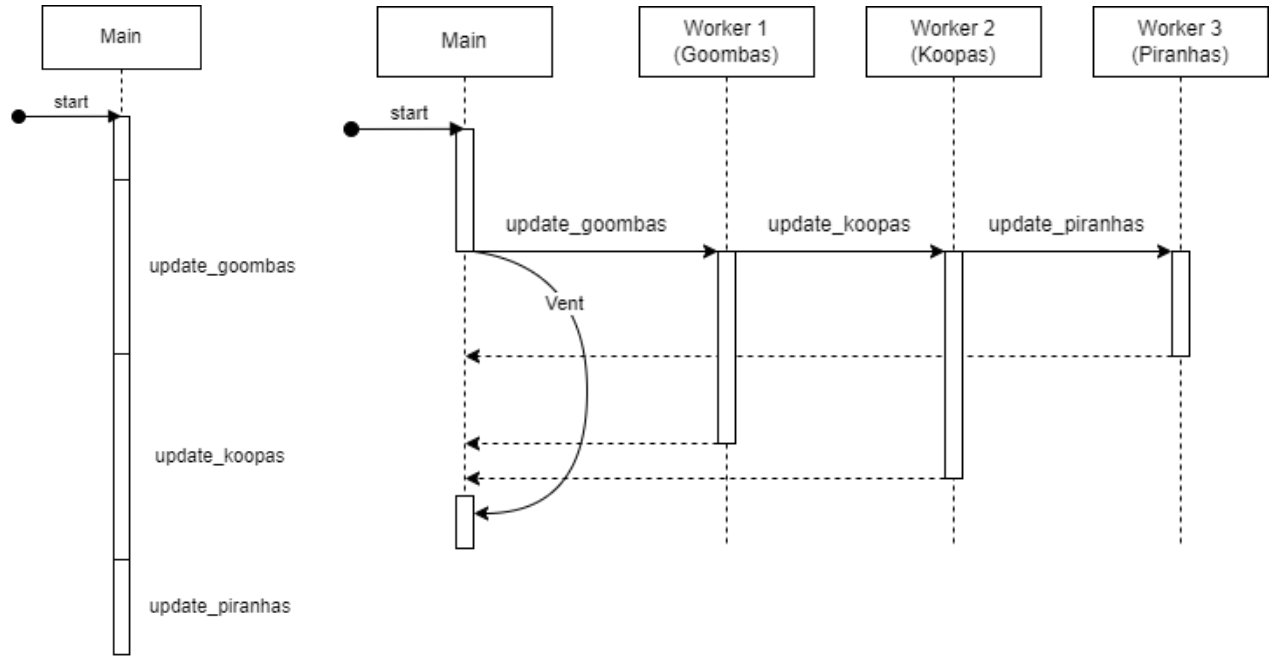
```
        let (goombas, koopas, piranhas) = futures::executor::block_on(join!(
            thread_pool.spawn_with_handle(updated_goombas).unwrap(),
            thread_pool.spawn_with_handle(updated_koopas).unwrap(),
            thread_pool.spawn_with_handle(updated_piranhas).unwrap(),
        ));

        scene.enemies.clear();
        scene.enemies.extend(goombas.0);
        scene.enemies.extend(koopas.0);
        scene.enemies.extend(piranhas.0);

        // ...
    }
```

Figur 5 Uddrag af "update_enemies"-funktionen, der viser på hvilken måde de forskellige futures sendes til thread-poolen.

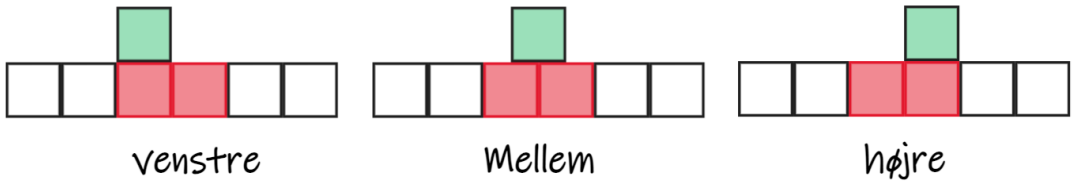
Derefter indsættes alle futures i thread-poolen for at køre dem på andre tråde og den nuværende tråd venter blot vha. et while-loop (som er skjul i ”block_on”-funktionen fra futures biblioteket) indtil alle fjender er færdig med at blive opdateret. Til sidst fjernes alle de outdaterede fjender i ”scene.enemies” og de opdaterede indsættes. Hvordan de forskellige tråde arbejder sammen, kan ses på nedenstående tråd-diagram:



Figur 6 tv. Single-threaded opdatering af fjender. th. Multi-threaded opdatering af fjender.

Til venstre ses hvordan, at i det enkelt-trådede eksempel bliver alle fjenderne opdateret efter hinanden, mens de overlapper i det multi-threadede program og på den måde er det hurtigere. Dog kan man også bemærke, at inden arbejdet sendes ud på de forskellige tråde er der langt større arbejde som skal gøre for at forberede overførslen til de andre tråde.

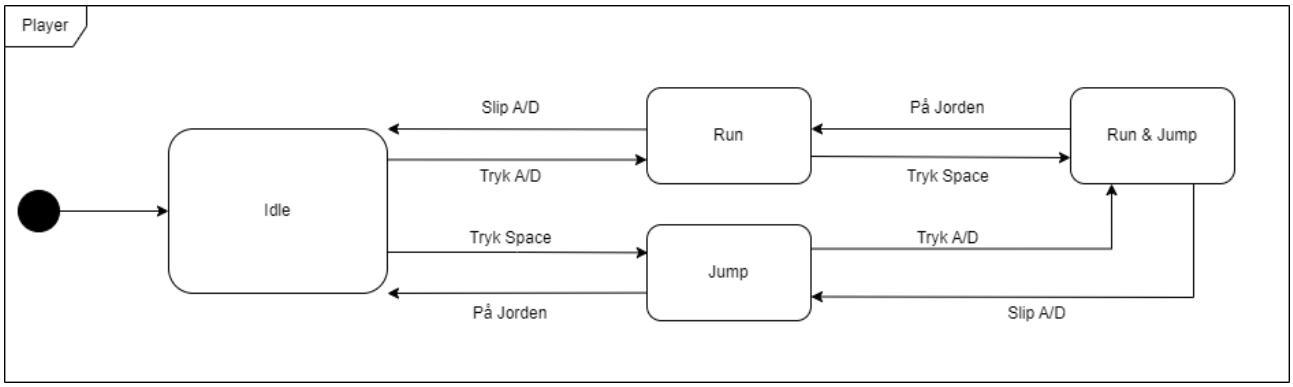
En anden vigtig funktion i programmet er ”below_of”-funktionen, som finder ud af om der er en tile under en given entity. Dette er super vigtigt ift. om der skal laves tyngdekraft på den entity. Dette gør den ved at konvertere den entity fra world-space til tile-space, som og tjekke om der er en tile der matcher i listen af tiles, der tages ind som argument.



Figur 7 De tre scenarier som er muligt for en entity at kunne kollidere med en tile på.

Da tile-space er i heltal, mens world-space er i decimaltal, skal der tjekkes to tiles, fordi der laves et implicit destruktivt cast fra f32 til u32. Det er vigtigt at tilen til højre altid tjekkes, fordi man ellers ville kunne falde igennem på højre siden af entitien. Ankoret/Origo sidder nemlig i toppen af venstre hjørne så når man konverter mister man mellem positionen og den konverteret ned til venstre scenariet. Det der er valgt at gøre i stedet for, er at plusse med bredden af den entity og så tjekke om der er en tile til venstre,der også rammer ved, at minus med et bagefter i tile-space. Dette giver det samme resultat, rækkefølgen er bare omvendt. For hele implementeringen af ”below_of”, jævnfør ”below_of”-funktionen i bilag.

Spilleren er implementeret som en state-machine der kan være i 4 forskellige stadier:

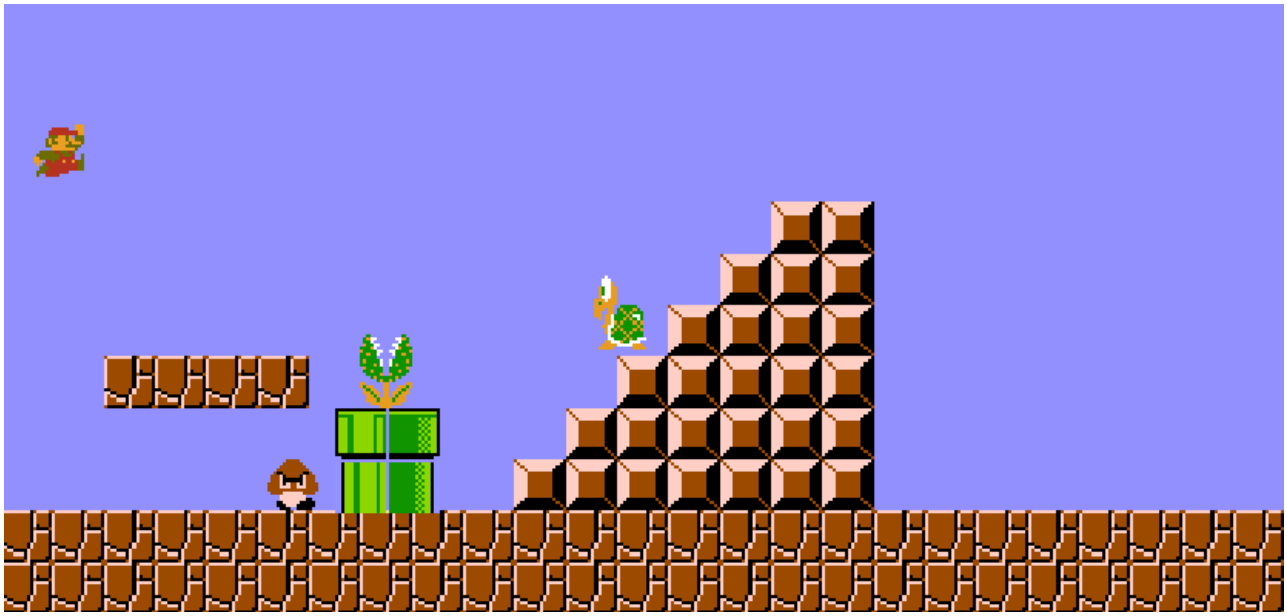


Figur 8 Diagram over de forskellige stadier spilleren kan være I

Når der ikke gøres noget ryger spilleren tilbage i ”idle” stadiet for at repræsentere at ingenting sker. Trykkes på space kommer spilleren over i ”jump” stadiet.

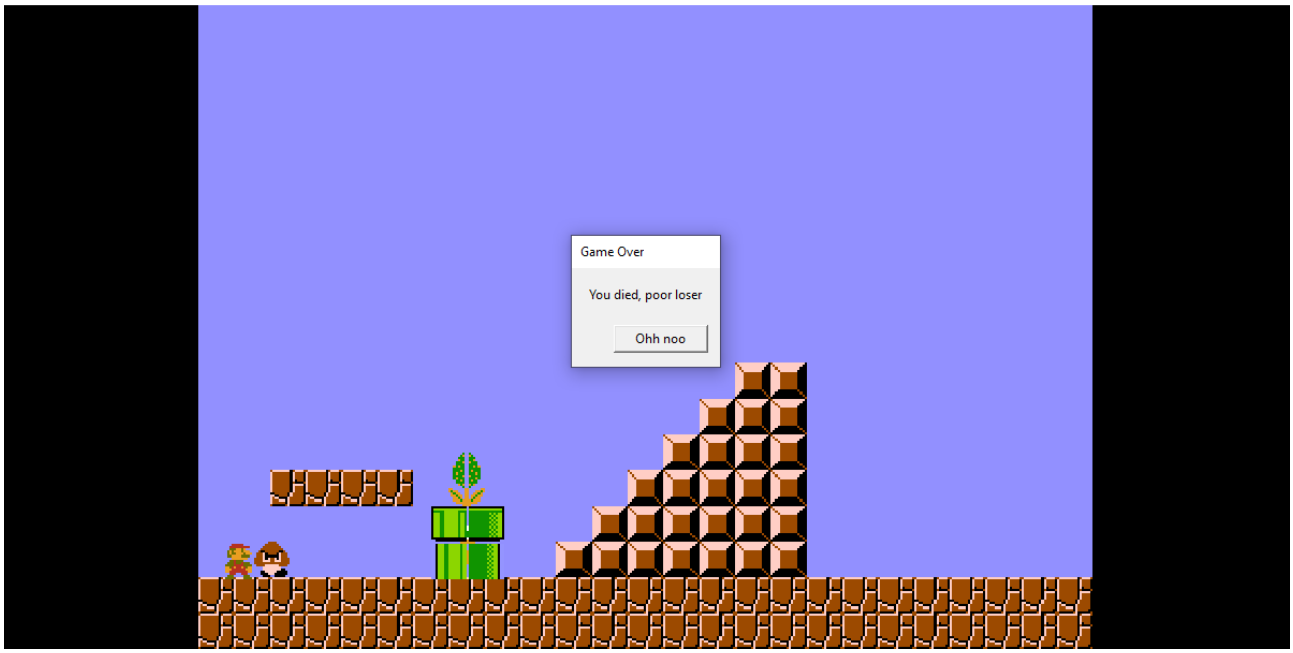
Test

Spilles spillet kan man se at mange dele fungerer. F.eks. kan fjender kollidere med jorden ud at falde igennem:



Figur 9 Screenshot som illusterer collision med jorden

Eller spilleren kan dø, hvis denne bliver ramt af fjender:



Figur 10 Screenshot som illusterer at man kan tabe spillet, hvis man bliver ramt af en fjende

Desuden er der ikke en mulighed for at vinde spillet ved at hoppe ind i et flag som i det rigtige Super Mario Bros spil. Man kan også bemærke at der er plads mellem røret, hvor der er gennemsigtigt. Dette skyldes at renderen ikke har support for, at en enkelt entity består af flere tiles.

En anden ting er, at koopaen vil crashe programmet, når den når til kanten af skærmen, fordi der kommer en heltal underflow fejl.

Man kan også bemærke at lyden virker, så der kommer en lyd, når Mario dør. Dog kan man ikke gemme sin progression, fordi man aldrig kan vinde.

Editoren er også kun halvt funktionerende. Den fungerer kun med farver og man kan ikke gemme levels.

Af de nødvendige krav er stort set alle funktionerende bortset fra det med at kunne vinde og bruge editoren. Dog er der også et par af nice-to-have kravene som er implementeret. Helt konkret er det animationer og lydeffekter som virker.

En anden bug er, at man kan klatre på væggene som en Ninja og gå igennem dem som et spøgelse, hvis man rammer helt rigtigt.

Diskussion

SDL2 biblioteket kom i vejen… I hvert fald bindings til Rust gjorde. Biblioteket er et hyppigt brugt bibliotek, men dokumentationen er lidt halvdårlig ift. det man kender fra Rust. Alle de hyppigt brugte biblioteker i Rust har fantastisk dokumentation, hvor alt er udførligt dokumenteret. Planen var at editoren skulle kunne bruge menuer, som man ofte ser i Windows-applikationer, men dette var ikke muligt, fordi SDL2 overtog event-loopet, så det var ikke muligt at læse menu events, da dette var implementeret for SDL2.

Et andet problem det besværliggjorde udviklingen af programmet var, at der ud af det blå kom ydelsesproblemer, fordi renderingsmetoderne i sdl2 sendte et draw-call til GPU'en, hver gang man tegnede en pixel, hvilket ikke var beskrevet i dokumentationen, så i starten kørte programmet 2fps. Dette tog en uges lektioner at fikse og da meget afhænger af renderingen, blev det umuligtt at udvikle på noget andet før dette var løst.

Man kan gå igennem vægge, hvis man bliver ved at bevæge spilleren ind i en væg. Dette er højst sandsynligt, fordi spillerens position er repræsenteret som floats og der kommer en upræcished på de små decimaler, når der tilføres en konstant hastighed og på den måde mener kollisionssystemet ikke, at de kolliderer. For at fikse dette skal man nok rykke spilleren tilbage hver frame med den mængde spilleren overlapper med tilen på.

En smart ting ved måden som update_enemies er implementeret på er at man får rigtig god performance med branch-prediction, fordi alle de forskelle fjender er filtreret ud i forskellige variabler i forvejen. Hvis de bare var splittet tilfældigt ud, var der ca. 33% chance for, at den gættede rigtigt, men der nu er op mod 100%.

Teoretisk set burde multi-threading også gøre så programmet kører hurtigere, men dette er ikke testet. Derfor kan det faktisk ikke vides om multi-threading gør programmet hurtigere.

Konklusion

Det kan konkluderes at man godt kan bygge et Mario lignende spil i Rust. Selvom mange elementer stadigvæk skal have finpudsning og der mangler ting som en hovedmenu og endnu vigtigere at man kan vinde. Multi-threading og parallelisering kan være en god måde at optimere programmer på, dog vides det ikke om det har gjort en forskel i dette program, da det ikke direkte er testet. Det kan også konkludres, at man godt kan benytte sdl2 til at udvikle spil, men en anden gang ville der nok skrives de ting som absolut skulle bruges fra bunden, fordi det endte med at bide os i halen, at SDL2 var dårligt dokumenteret.

Bibliografi

About SDL. (u.d.). Hentet fra Simple Direct Media Layer: https://www.libsdl.org/

alexcrichton, cramertj, & taiki-e. (u.d.). *futures*. Hentet fra DOCS.rs: https://docs.rs/futures/latest/futures/

Bartleby. (u.d.). *Fundamentals of Multithreaded Algorithms*. Hentet fra Bartleby: https://www.bartleby.com/subject/engineering/computer-science/concepts/fundamentals-of-multithreaded-algorithms

CMI. (u.d.). *How Many Cores? Is More Always Better?* Hentet fra CMI Blog: https://www.newcmi.com/blog/how-many-cores

Computer Hope. (u.d.). *Branch Prediction*. Hentet fra Computer Hope: https://www.computerhope.com/jargon/b/branch-prediction.htm

MDN. (u.d.). *State Machine*. Hentet fra MDN Web Documentation: https://developer.mozilla.org/en-US/docs/Glossary/State_machine

Rust-Lang. (u.d.). *Defining an Enum*. Hentet fra The Rust Programming Language: https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html

theotherphil, fintelia, & HeroicKatora. (u.d.). *image*. Hentet fra DOCS.rs: https://docs.rs/image/latest/image/

Bilag

”Below_of”-Funktionen

```rust
pub fn below_of(position: Vec2, size: UVec2, tiles: &[MapTile]) -> Option<BoundingBox> {
    let mut left: Option<MapTile> = None;
    let mut right: Option<MapTile> = None;
    for tile in tiles {
        if tile.coordinate == (position + (size.as_vec2())).as_uvec2() / Renderer::TILE_SIZE {
            left = Some(*tile)
        } else if tile.coordinate
            == (position + (size.as_vec2())).as_uvec2() / Renderer::TILE_SIZE - 1
        {
            right = Some(*tile);
        }

        if left.is_some() && right.is_some() {
            break;
        }
    }

    match (left, right) {
        (Some(left), None) => Some(left.collider()),
        (None, Some(right)) => Some(right.collider()),
        (Some(left), Some(right)) => {
            let mut left_collider = left.collider();
            let right_collider = right.collider();

            left_collider.width += right_collider.width;
            left_collider.height = left_collider.height.max(right_collider.height);
            Some(left_collider)
        }

        _ => None,
    }
}
```

Tidsplan

08-03-2023:

Færdiggørelse af projektbeskrivelse, og opstart på programmet.

Sebastian begynder på (de)serialisering af scenen og selve udformningen af scenestrukturen, da de to ting går hånd i hånd. Der skal udvikles en specifikation for hvilke felter der skal være i JSON-dataet. Samt en måde at indlæse, gemme og ændre i de gemte scener.

Hector begynder på renderen, som skal tegne pixels til skærmen. Dog skal rendereren også kunne tegne sprites ud fra et spritemap og så loade dem ind det rigtige sted på skærmen. Derudover skal renderen vigtigst af alt, kunne tage imod en scene, og så tegne den til skærmen.

13-03-2023:

Når Sebastian er færdig med at lave serialisering af scener, så skal der kigges på måder at indsamle input, så vi kan få en spiller til at bevæge sig. I Mario er kameraet fastlåst til spillerens position.

Hector: Samme som ovenstående.

14-03-2023:

Sebastian: Samme som ovenstående.

Hector: Samme som ovenstående.

20-03-2023:

Når Hector er done med rendereren, så skal han lave en level editor, hvor man kan plotte terræn, enemies og enemy paths vha. af musen m.m. Level-editoren skal integreres med serialiseringssystemet, så man kan eksportere de levels man laver og læse dem ind i spillet.

Sebastian begynder på en animation stat emachine, som kan animere fjender ud fra forskellige parametre f.eks. om de skifter retning, bevæger sig eller angriber.

22-03-2023:

Derefter overtager Sebastian level editoren fra Hector, hvor hector derefter kigger på at lave kollision mellem spilleren, enemies og terræn/andre objekter i verdenen.

27-03-2023:

Sebastian samme som ovenstående

Hector samme som ovenstående

28-03-2023:

Sebastian samme som ovenstående

Hector kigger på afspilning af lyd.

11/04

Det er ikke lykkedes at blive færdige med kollisionen endnu, og der er kommet nogle problemer med renderen så de skal fikses til næste gang

17/04

Renderen er blev optimeret med noget smart texture caching, dog virker levelleditoren ikke helt, men fokus ligger på at få enkelte assets ind, og så få lavet noget gamelogic

19/04

Kolission virker halvt nu, dog kun op og ned, til næste kan må vi få lavet det til siderne. Dog har vi også fået implementeret Goompa logik.

(Tidsplanen revurderes løbende og opdateres i påskeferien)