Office Apocalypse

Indledning

I denne synopsis forklares hvilken relevant programmeringsteori og metoder som er brugt til at udvikle computerspillet Office Apocalypse. Dernæst analyseres konkrete eksempler på hvorfor vi har gjort som vi har gjort. Til sidst testes på de opstillede krav for at undersøge, hvorvidt programmet kan gøre hvad det skal.

Programbeskrivelse

Office Apocalypse er et 3D first-person-shooter (FPS) spil, der finder sted i en zombie apokalypse. Spillet tager sted i en lukket kontorbygning. Det er et voxel-baseret spil.

Vi ville gerne lave noget udfordrende, så spillet er bygget fra bunden nærmest uden brug af biblioteker.

Kravspecifikation

Need to Have	
Fps-controller	
Zombies som kan dræbe spilleren	
Spilleren skal kunne dræbe zombies	
Spillet skal udspilles på et lukket kontor	
Text-rendering	
Zombies spawner tilfældigt på banen uden for spillerens fov	
Nice to Have	
Procedural animation	
Realistisk lys	
Ødelæggeligt terræn	
Partikler	
Lyskilder	
Tåge system	
Atmosfærisk lighting	
Bloom	
Won't have	
Multiplayer / lokal co-op	
Zombies kan ikke have våben	

Baseret på programbeskrivelsen og mængden af tid til opgaven, har vi opsat ovenstående krav til vores produkt.

Funktionalitet

Målet med spillet, var at lave en FPS, i voxel stilen, der havde destructible terrain og procudurally generated terræn. Voxel stilarten er kendetegnet ved, at spilmodeller er opbygget af kuber. Dette tillader at opbygge terrænet i et grid af kuber. Denne type terrænkonstruktion gør det muligt at implementere forskellige niveauer af destruktion og procedural generering af terrænet. Ved at gøre brug af destruktionsteknikker kan spillerne skade og ødelægge terrænet, hvilket kan skabe en mere realistisk oplevelse af kamp og

kampeffekter. Procedural generering af terrænet kan også give spillerne en unik og uforudsigelig oplevelse hver gang de spiller spillet.

En yderligere fordel ved voxel-stil er, at det ofte kræver mindre computerkraft end andre 3D-stile, at simulere de førnævnte ting ift. trekantsbaserede spil, hvilket kan gøre spillet mere tilgængeligt for en bredere gruppe af spillere.

Metode

Vi har hovedsageligt anvendt test driven development, til udvilklingen af spillet. Dette er en metode hvor man skriver test, der beskriver den forventede adfærd af ens kode, på forhånd. Her opdeler man sin kode i units, som er mindre dele af koden, så som datastrukturer og metoder. Her kan man så lave en model for hvilken adfærd der forventes for en række forskellige parametre. Her vil man så skrive sin test. Før man skriver sin kode, lader man som om man har lavet koden, og anvender den i testen.

```rust
#[test]
▶ Run Test | Debug
fn graph() {
    let mut g = SceneGraph::new();
    let root = g.root();

    let transform = Mat4::from_cols_array_2d(&[[1., 2., 3., 4.]; 4]);

    let a = g.insert_entity(
        Object {
            transform,
            model: Model::default(),
            tag: None,
        },
        &root,
    );
    let b = g.insert_entity(
        Object {
            transform,
            model: Model::default(),
            tag: None,
        },
        &a,
    );

    g.evaluate_all();

    assert_eq!(
        g.mutated_entity(&b).unwrap().transform().unwrap(),
        &(transform * transform)
    );
}
```

Rust har en indbygget test suite i sit buildsystem. Dette tillader at annotere funktioner med ‘test’ makroen, som herefter eksekverer unit tests når man kører ‘cargo test’ terminalen. Dette gør det super let at bruge test-driven development, da det er direkte understøttet i sproget og man ikke behøver at sætte alle mulige biblioteker op først.

```
running 5 tests
test scene::graph ... ok
test ai::straight ... ok
test terrain::block_coordinates ... FAILED
test terrain::map ... ok
test format::vox::tests::test_parse ... ok
```

Her ses et eksempel på 4 unit tests, som giver det forventede output. Dette kaldes også for at de ”passer” og en som ikke giver det forventede resultat.

Denne måde at udvikle kode på er specielt praktisk i store projekter, da den tillader at lave ændringer på implementerings detaljer senere, og så kunne se om koden stadig lever op til de samme forventninger, da den blev skrevet i første omgang.

Det er dog specifikt i spiludvikling, svært at lave unit tests på ting som gameplay og movement, da man typisk er afhængig af visuelt feedback, for at kunne afgøre om noget virker. Dog er der mange af de andre systemer som godt kan unit testes.

Brugergrænseflader

Programmet kan interageres med gennem WASD-tasterne samt med musen. WASD-tasterne styrer kameraets position, mens musen styrer retningen som kameraet peger.

Venstremusetast bruges til at skyde med våbnet som kan ses på nedenstående billede. Derudover kan spilleren hoppe på mellemrumstasten.

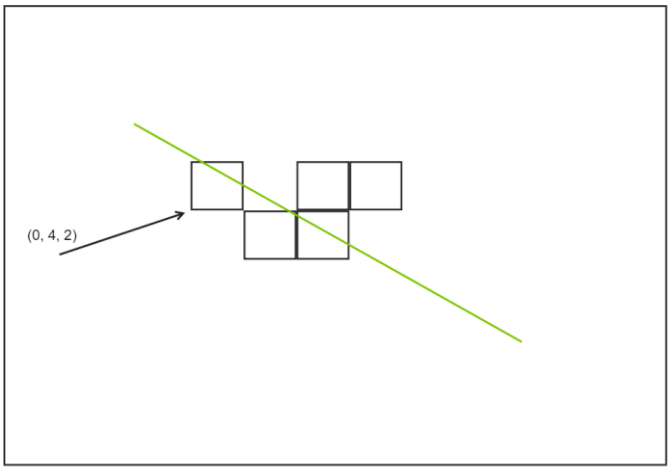
Man kan skifte mellem forskellige typer af våben på 1 og 2.

Figur 1 Screenshot fra spillet, hvor vi kan se en enemy zombie og en flyvende pistol.

Pseudokode

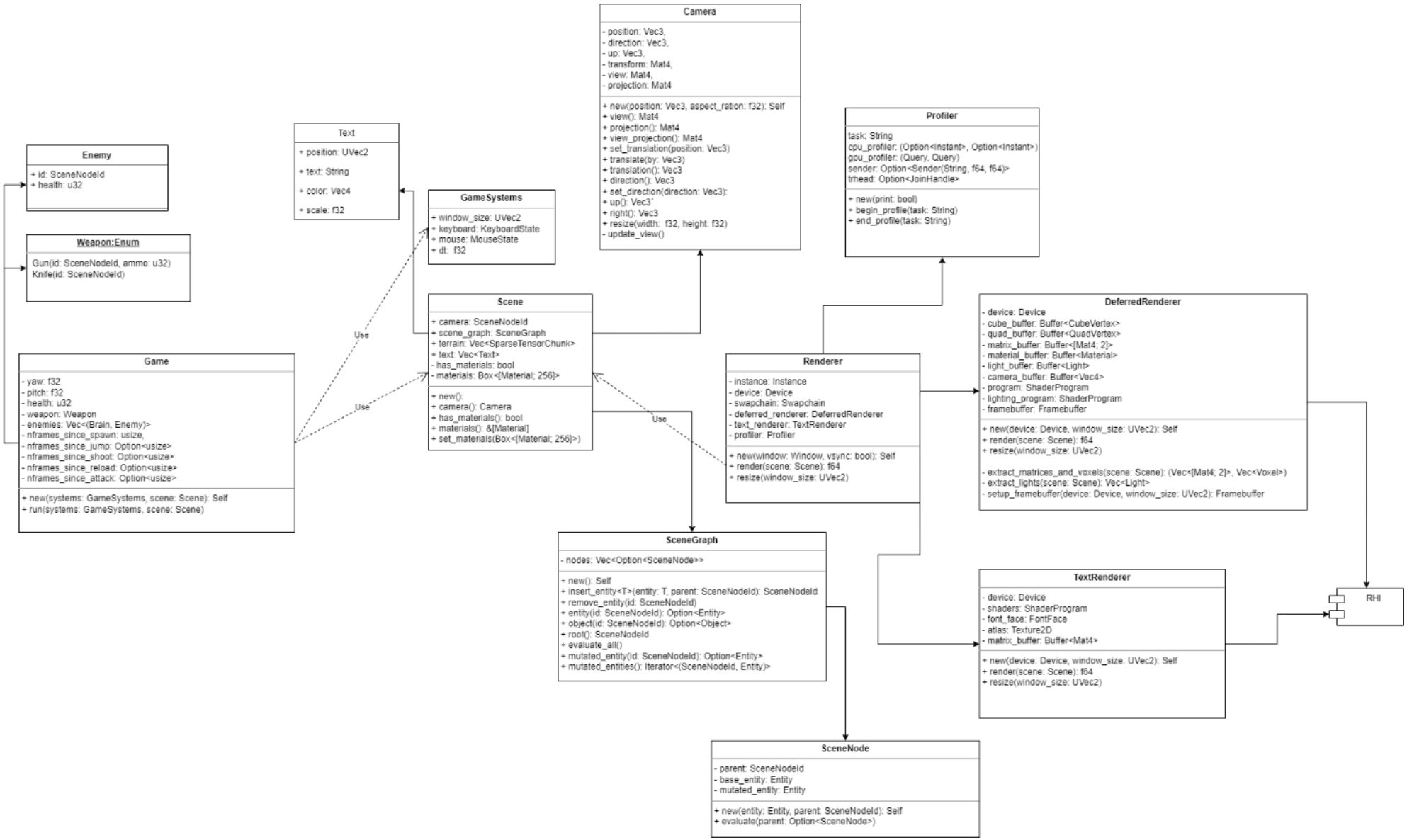
```
hitEnemy(enemy, damage):
    ray = Ray(origin: cameraPosition, direction: cameraDirection, len: 100)
    hit = castRay(ray, enemy)

    if hit:
        enemy.health -= damage

castRay(ray, enemy):
    t = 0
    while t != ray.len:
        v = floor(ray.origin + ray.direction * t);

        foreach voxel in enemy:
            if voxel == v:
                return true

        t += 0.25

    return false

```

Spillet indeholder en shooting mechanic som virker ved at trace en stråle gennem scenen. Hvis fjenden er er ramt af skuddet fratrækkes dennes antal liv med damage. castRay funktionen er funktionen som detekterer om fjenden er ramt. Dette gør den ved at opstillet en parameterfremstilling for en linje og så variere på variablen t med et skridt. Øges dette skridt, så bliver algoritmen mere upræcis. Jo større skridt, jo mere upræcis og jo mindre beregningstung.



På ovenstående billede er en voxel givet i punktet (0, 4, 2). Men vores scene er delt op i flere forskellige koordinatsystemer. Voxels Voxel-space, hvor koordinaterne er heltal, mens kameraet virker i world-space som er floats. Vi ved at en voxel er ramt af en stråle, hvis alle af strålens kompossanter er indenfor voxlen. Når vi tracer strålen bliver vi nødt til at konvertere mellem world-space og voxel-space. Dette kan gøres rigtig let, fordi vi har valgt at en voxels origo skal ligge nederst til venstre af kuben. Vi kan derfor bare floor decimaltallet og dette burde give os den voxel som den rammer. Havde vi valgt, at origo var i centrum, så havde det været lidt mere kompliceret at beregne, hvor den rammer.

Klassediagram



På klassediagrammet fremgår de vigtigste klasser i projektet og hvordan de relaterer til hinanden. Udover hvad man kan se på diagrammet er der et udførligt RHI modul som er ca. 700 linjer kode.

Diagrammet består af to dele, Game og Renderer. Det er scenen og GameSystems binder de to dele sammen. Der er altså meget lav kobling mellem selve spillogikken i Game og de bagvedliggende systemer. Dette betyder at havde man lyst kunne man fjerne alt spillogikken og skrive et nyt spil. Det bliver nærmest til en mini-spilmotor. Derudover kunne man ret let lave en ny implementering af renderen og udskifte den skulle man have lyst til det. I en forstand kan man se Renderer og Game som interfaces. Da vi ikke havde mange spil eller renderer vi skulle bygge - bare en enkelt. Valgte vi at lave dem til konkrete klasser og ikke interfaces, da vi ikke havde brug for modulariteten.

Scene & Scenegraph

Som vi har set på klassediagrammet, så kommunikerer spillet med de bagvedliggende systemer via en scene og scenegraph.

Scenen indeholder informationer om hvilket camera som er spillerens kamera. Man kan altså have flere kameraer i scenen. Derudover opbevarer den også materiale data for de forskellige modeller.

Scenen indeholder en scenegraph, som er der, hvor alle dynamiske entities bliver specificeret. Grafen specificerer hvilke objekter som er underobjekter af andre.

```
        let Scene { scene_graph, .. } = scene;
        let gun_id = scene_graph.insert_entity(gun, &scene_graph.root());
        let _ = scene_graph.insert_entity(magazine, &gun_id);
```

Objekter i verdenen har en position ligesom GPS koordinater i den virkelige verden. Ved at bruge scenegraphen kan vi specificere at objekter hænger sammen. F.eks. en pistol med et magasin, som vist på ovenstående kode. Vi vil gerne have at magasinet følger med pistolen, så ved at i insert_entity kaldet specificerer gun_id, så det andet argument følger magasinet med.

Under the hood itererer scenegraphen over alle objekterne og ganger et objekts transformationsmatrix sammen med deres forældres transformationsmatrix sammen. Et objekt kan have mange forældre i scenegraphen. En transformationsmatrix bruges til at finde ud af hvordan et objekt er i translateret, drejet og skaleret i verdenen. Dette kan ses i nedenstående kode:

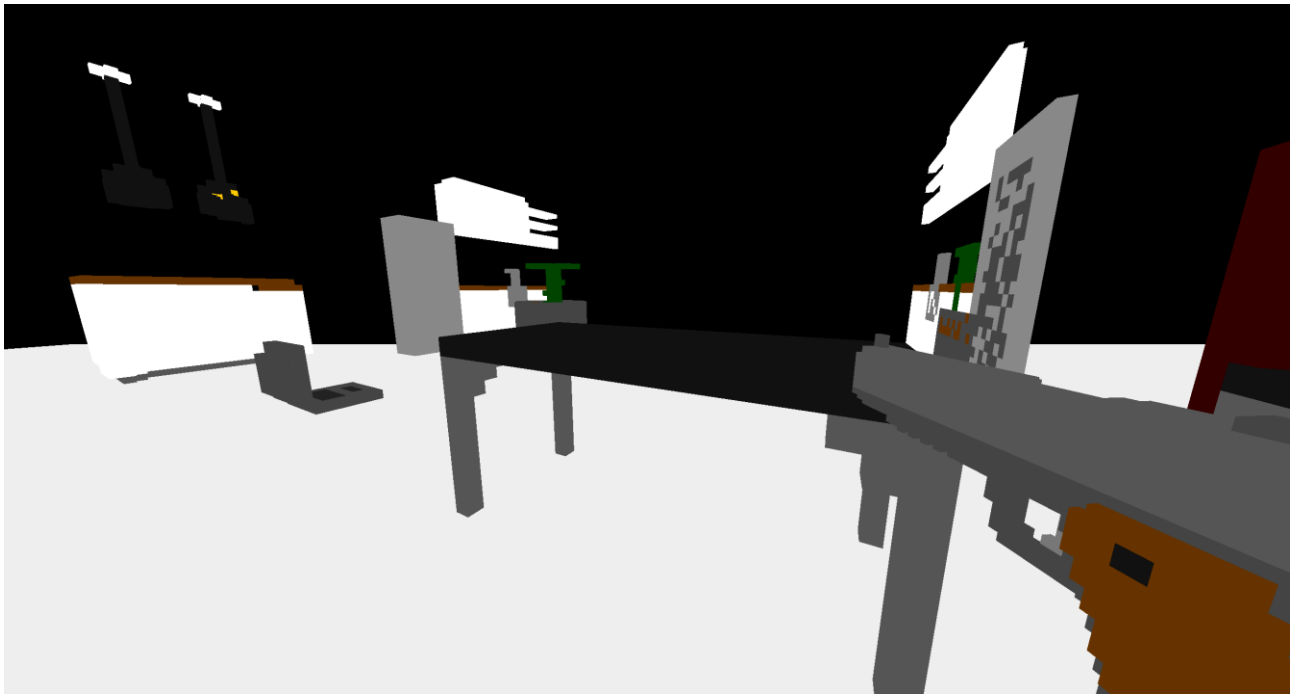
```rust
    pub fn evaluate_all(&mut self) {
        for n in 0..self.nodes.len() {
            let parent = if let Some(node) = &self.nodes[n] {
                self.nodes[node.parent.0].clone()
            } else {
                continue;
            };
            if let Some(node) = &mut self.nodes[n] {
                node.evaluate(&parent);
            }
        }
    }
```

SparseTensorChunk

For at kunne implementere destruktivt terræn, terrængeneration og kollisionsdetektion, har vi valgt at bruge en tensor af voxels. Ideen er at have en datastruktur, der kan indekseres med koordinater for at finde mulige voxels (eller en tom plads). Dette er implementeret på en måde, der har mindre overhead end at iterere over alle voxels i en liste og tjekke deres position. I stedet for at det tager O(n) for at søge igennem voxels vil denne måde at gøre det på være O(1).

Rendereren forventer imidlertid, at modeller er i et format, der kan itereres over positioner og voxels, hvor tomme pladser bliver ignoreret. Vi har eksperimenteret med forskellige måder at implementere voxel-tensoren på, såsom HashMaps og en brugerdefineret type. Vi har også prøvet med et array, der indeholder information om placeringen af tomme pladser. Selvom det stadig var nødvendigt at iterere over voxels for at finde placeringen med et givet sæt koordinatorer, kunne det gøres hurtigere med den ekstra information.

En af de store fordele ved at bruge en voxel-tensor til at repræsentere kontorlokaler i vores spil er, at det gør det nemmere at skabe variation i lokalerne og gøre dem mere realistiske og interessante. Ved at repræsentere cubicles, skriveborde og andre kontormøbler som voxels-tensore, kan man let manipulere og justere placeringen af disse elementer i lokalerne. Dette giver mulighed for at skabe variation i layoutet af kontorlokalerne, så de ikke føles ensformige og kedelige.



Terræn generingen fungere ved, først at generere et tilfældigt map. Det kan dog ikke være helt tilfædligt, der det samme terræn skal generes hvis spilleren går tilbage til et tidligere spot igen. Derfor er en pseudo-randomiseret funktion brugt, der bliver seedet af spiller koordinaterne. Dette betyder altså at den generer rimeligt tilfældige værderier, dog vil den altid generere det samme resultat, med det samme seed.

```rust
fn random(v: Vec3, r: Range<usize>, variant: usize) -> usize {
    let a: usize = match r.start_bound() {
        std::ops::Bound::Included(a) => *a,
        _ => panic!("invalid bound for random number generation"),
    };
    let b: usize = match r.end_bound() {
        std::ops::Bound::Excluded(b) => *b,
        _ => panic!("invalid bound for random number generation"),
    } - a;

    let x = (v.x * SEED).abs() as usize;
    let y = (v.y * SEED).abs() as usize + variant;
    let z = (v.z * SEED).abs() as usize;

    let r = (x | z) & y;

    r % b + a
}
```

Denne funktion bliver således brugt til at vælge en tilfældig model, der kan placeres i hvert ‘cubical’, som er et segmnet af mappet. På denne måde kan koden også udvides til at anvende en mere avanceret algoritme til at producere cubicals, end den nuværende, der bare placerer en enkelt random model i den.

Ud fra den resulterende MapChunk struktur, kan en voxel-tensor produceres (kaldet en SparseTensorChunk i koden). Dette gøres simpelt, ved at iterere over alle cubicals, or kalde metoden der producere en tensor for den cubical, og så translate positionen af modelen, med en simpel funktion, blk_pos

```rust
fn blk_pos(x: usize, y: usize, center: Vec3) -> Vec3 {
    let min = (FOV as f32 / -2.) * CUBICAL_SIZE as f32;
    let min = vec3(min, 0., min);
    let p = vec3(
        x as f32 * CUBICAL_SIZE as f32,
        y as f32 * CUBICAL_SIZE as f32,
        125.,
    );

    center + min + p
}
```

Det ville dog give en urealistisk oplevelse, hvis MapChunks kun blev loadet ind, når man nåede kandten af den eksisterende MapChunk. Derfor kan en TerrainMask struktur, brugt til at kontinuelt identificere nye cubicals der skal loades ind. Så der er en radious af 6 cubicals loadet på et givendt tidspunkt.

Her indeholder TerrainMask strukturen information om hvilke Cubicals der allerede er blevet loadet ind, baseret på spilerens position og den sidste position en MapChunk blev genereret fra. Her genereres TerrainMask'en med mask metoden

```rust
pub fn mask(&self, old_pos: Vec3) -> TerrainMask {
    let mut tmp = TerrainMask([[true; FOV]; FOV]);

    for y in 0..FOV {
        for x in 0..FOV {
            let new_pos = blk_pos(x, y, self.center);
            if old_pos.abs_diff_eq(new_pos, CUBICAL_SIZE as f32) {
                tmp.0[y][x] = false;
            }
        }
    }

    tmp
}
```

Afkodning af .vox Format

Vi har benyttet os af et program kaldet MagicaVoxel til at modellere de forskellige objekter i spillet. Vi har så skulle indlæse de her modeller i spillet, så vi kan rendere dem. Problemet er, at der findes ikke noget bibliotek i Rust som kan bruges til indlæse det her format.

En af opgaverne vi så skulle igennem var at afkode det her format og læse op på udvikleren af MagicaVoxel's dokumentation om hvordan formatet virker.

Parseren vi har skrevet, er godt eksempel på del og behersk tankegangen man bruger i computational thinking. Her har vi nedkogt hver type af data i sin egen lille funktion. Filen starter med en magisk header, som indeholder en signatur og hvilken version af formatet filen encoded som.

```rust
fn parse_header(input: &mut impl ReadBytesExt) -> ([u8; 4], i32) {
    let signature = {
        let mut buf = [0; 4];
        input.read_exact(&mut buf).unwrap();
        buf
    };

    let version = input.read_i32::<VoxEndian>().unwrap();
    (signature, version)
}
```

Ovenstående kode starter med at læse signaturen som er tekststrengen ”VOX ”. Derefter læses versionen af formatet. Her er der specificeret en endianness.

Der findes to typer af endianness: little endian og big endian. Endianness specificerer hvilken retning bitene er gemt i hukommelsen. Little endian betyder at den mindst signifikante bit er til venstre i hukommelsen, mens big endian betyder, at den mest signifikante bit er til venstre i hukommelsen. Hvilken man bruger afhænger af CPU’ens arkitektur og betyder faktisk at skriver man til en fil på en computer og åbner på en anden kan værdien af en tal være ”flippet”. Derfor specificerer udviklere hvilken endian de bruger i deres formater. MagicaVoxel bruger little endian.

Udvikleren har defineret forskellige datatyper i hans format. En anden er en string, som består af 4 bytes som er dens længde i antal tegn efterfulgt af alle tegnene

```rust
fn parse_string(input: &mut impl ReadBytesExt) -> String {
    let len = input.read_u32::<VoxEndian>().unwrap() as _;
    let mut buf = vec![0; len];
    input.read_exact(&mut buf).unwrap();
    String::from_utf8(buf).unwrap()
}

fn parse_dict(input: &mut impl ReadBytesExt) -> Vec<(String, String)> {
    let n = input.read_u32::<VoxEndian>().unwrap();

    let mut dict = Vec::new();
    for _ in 0..n {
        let key = parse_string(input);
        let value = parse_string(input);

        dict.push((key, value));
    }

    dict
}
```

Et eksempel på dette er et dictionary som består af 4 bytes som er antallet af par n efterfulgt af det antal (nøgle, værdi) par, hvor både nøglen og værdien er af typen string. Da vi har delt problemet og i flere problemer kan vi undgå at genimplementere string-parsing igen, ved at bruge parse_string funktionen.

Deferred Rendering

Modsat normal rendering betyder deferred rendering, at lysberegningerne bliver postponed til et andet render pass. Dette betyder at de dyre lysberegninger kun laves for de pixels som faktisk ender op på skærmen. Dette betyder at tidskompleksiteten bliver ændret:

$$O(geometry \cdot lights \cdot pixels) \rightarrow O(geometry + lights \cdot pixels)$$

Og går vi ud fra at pixels er konstante i begge renderingsalgoritmer, så går vi fra $O(n^2)$ til $O(n)$.

Så hvordan virker deferred rendering?

Vi opretter en framebuffer og binder mange textures til den. Vi kan så skrive data til de her textures. I vores program skriver vi pixelens position i world-space og en albedo texture, som opbevarer materiale data omkring en pixel.

Derefter binder vi nye shaders ind som modtager de texture vi lige har skrevet til og den bruger så disse værdier sammen med information omkring lyskilder til at beregne farven på den pixel.

Text Rendering

En god måde at kommunikere information til en bruger på er gennem tekst. I spillet skulle tekst rendering bruges til at kommunikere til spilleren deres score som tæller op og hvor meget ammunition som deres

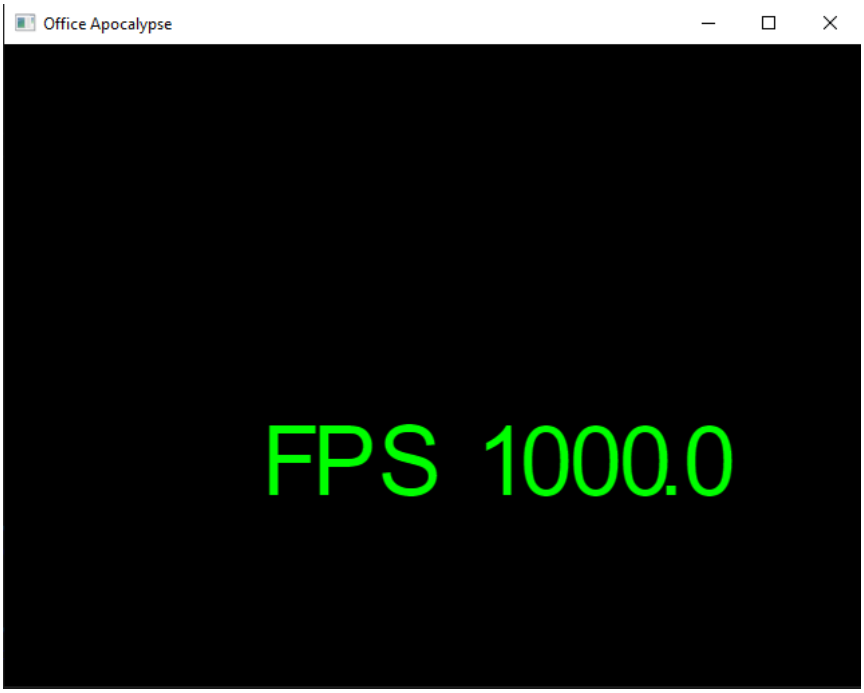
pistol har tilbage i magasinet. Til os som udviklerne af spillet kunne det være fedt med et stykke tekst som viste antallet af frames som blev renderet i sekundet.

Text Rendering virker ved at man indlæser et billede kaldet et texture atlas, hvor alle bogstaver er pakket sammen i et større billede. Derudover skal man bruge en fil som fortæller, hvor de forskellige bogstaver befinder sig i billedet. F.eks. kan vi se på billedet at bogstavet med ascii-værdien 50 starter på (215, 168) og er 38 pixels bred og 55 pixels høj. Dette format er autogenereret af et online værktøj, men da der ikke findes nogen parser til det format heller, skulle vi også skrive sådan en.

```
info face="sans-serif" size=72 bold=0 italic=0 charset="" unicode=1 stretchH=100 smooth=1 aa=1 pa
common lineHeight=72 base=56 scaleW=386 scaleH=387 pages=1 packed=0
page id=0 file="Unnamed.png"
chars count=66
char id=32 x=0 y=0 width=0 height=0 xoffset=0 yoffset=0 xadvance=21 page=0 chnl=15
char id=33 x=55 y=55 width=12 height=54 xoffset=4 yoffset=3 xadvance=21 page=0 chnl=15
char id=44 x=55 y=110 width=11 height=19 xoffset=4 yoffset=48 xadvance=21 page=0 chnl=15
char id=46 x=55 y=130 width=11 height=9 xoffset=5 yoffset=48 xadvance=21 page=0 chnl=15
char id=48 x=254 y=211 width=37 height=55 xoffset=1 yoffset=3 xadvance=41 page=0 chnl=15
char id=49 x=215 y=333 width=23 height=54 xoffset=6 yoffset=3 xadvance=41 page=0 chnl=15
char id=50 x=215 y=168 width=38 height=54 xoffset=0 yoffset=3 xadvance=41 page=0 chnl=15
char id=51 x=169 y=275 width=38 height=55 xoffset=1 yoffset=3 xadvance=41 page=0 chnl=15
char id=52 x=169 y=220 width=39 height=54 xoffset=-1 yoffset=3 xadvance=41 page=0 chnl=15
char id=53 x=215 y=223 width=38 height=54 xoffset=1 yoffset=4 xadvance=41 page=0 chnl=15
char id=54 x=254 y=267 width=37 height=55 xoffset=1 yoffset=3 xadvance=41 page=0 chnl=15
char id=55 x=71 y=334 width=37 height=53 xoffset=2 yoffset=4 xadvance=41 page=0 chnl=15
char id=56 x=215 y=0 width=38 height=55 xoffset=1 yoffset=3 xadvance=41 page=0 chnl=15
char id=57 x=215 y=56 width=38 height=55 xoffset=1 yoffset=3 xadvance=41 page=0 chnl=15
char id=65 x=0 y=255 width=52 height=54 xoffset=-2 yoffset=3 xadvance=49 page=0 chnl=15
char id=66 x=169 y=110 width=42 height=54 xoffset=4 yoffset=3 xadvance=49 page=0 chnl=15
char id=67 x=71 y=222 width=49 height=56 xoffset=2 yoffset=2 xadvance=52 page=0 chnl=15
char id=68 x=123 y=0 width=46 height=54 xoffset=4 yoffset=3 xadvance=52 page=0 chnl=15
char id=69 x=169 y=165 width=42 height=54 xoffset=4 yoffset=3 xadvance=49 page=0 chnl=15
char id=70 x=215 y=278 width=38 height=54 xoffset=4 yoffset=3 xadvance=44 page=0 chnl=15
char id=71 x=71 y=0 width=51 height=56 xoffset=2 yoffset=2 xadvance=57 page=0 chnl=15
char id=72 x=123 y=223 width=44 height=54 xoffset=4 yoffset=3 xadvance=52 page=0 chnl=15
char id=73 x=55 y=250 width=10 height=54 xoffset=5 yoffset=3 xadvance=21 page=0 chnl=15
char id=74 x=330 y=194 width=32 height=55 xoffset=0 yoffset=3 xadvance=36 page=0 chnl=15
```

Man specificerer via scenen, hvor man gerne vil have tekst på skærmen henne, tekstens størrelse og tekstens farve.

Derefter itererer man over alle bogstaver i et stykke tekst, hvor man genererer texture koordinater som svarer til lokationen af den pixel.



Ovenstående billede viser hvordan tekst renderingen så ud da den virkede.

```
let to_opengl = |texcoord: Vec2| {
    let x = texcoord.x / self.font_face.width as f32;
    let y = 1.0 - (texcoord.y / self.font_face.height as f32);
    vec2(x, y)
};

vertices.extend from slice(&[
    // top left -> top right -> bottom left
    TextVertex {
        position: position - glyph_offset + advance,
        texcoord: to_opengl(glyph_position),
    },
    TextVertex {
        position: position + glyph_width - glyph_offset + advance,
        texcoord: to_opengl(glyph_position + glyph_width),
    },
    TextVertex {
        position: position - glyph_height - glyph_offset + advance,
        texcoord: to_opengl(glyph_position + glyph_height),
    },
    // top right -> bottom right -> bottom left
    TextVertex {
        position: position + glyph_width - glyph_offset + advance,
        texcoord: to_opengl(glyph_position + glyph_width),
    },
    TextVertex {
        position: position +glyph_width-glyph_height-glyph_offset +advance,
        texcoord: to_opengl(glyph_position + glyph_size),
    },
    TextVertex {
        position: position - glyph_height - glyph_offset + advance,
        texcoord: to_opengl(glyph_position + glyph_height),
    },
]);
```

Hvert bogstav opfattes som en lille 2D-firkant på skærmen om texturen kopieres ned på. Derfor genereres 6 vertices som beskriver to retvinklede trekanter. Ovenstående kan ses hvordan på firkanten, men også texture koordinaterne genereres. Koordinaterne skal clampes ned i til 0..1, hvilket er hvad to_opengl funktionen gør. Desuden flipper den y-koordinaten, da OpenGL bruger omvendte koordinater på y-aksen.

Derefter er det bare at binde verticesne og billedet til GPU'en og køre et par meget simple shaders.

Test

I kravene fremgår følgende fire krav, som skal opfyldes af programmet. Når vi har testet programmet, har vi selv spillet. Vi har altså ikke fået andre til at afprøve det for os. Dette kan betyde, at mulige bugs er vi ikke stødt ind i, fordi vi spiller spillet på en særlig måde, da vi har udviklet det.

Vi har fundet frem til at det kun er et af kravene som er opfyldt fyldestgørende, mens de andre var vi på vej med. Det eneste opfyldte krav er en kameracontroller, som virker. Text-rendering har virket, men da vi lavede refactoring fik vi det ikke genimplementeret. Dette krav kan ikke ses som opfyldt, da det ikke virker lige nu.

Selvom vi fik lavet selve raycasting mekanikken, nåede vi ikke at fikse de sidste fejl, så zombien faktisk døde og blev fjerne fra scenen. I stedet sker der bare ingenting. Dette krav er altså heller ikke opfyldt.

Zombien kan heller ikke dræbe spilleren da det er den samme teknik som bruges til begge (som ikke virker). Dog kan den bevæge sig ved at give den koordinater, så det er meget let at integrere den med en pathfinding algoritme.

Vi udviklede og testet den underlæggende kode til terræn generering, og fik det til at virke, med nogen få bugs. Dog havde vi ikke tid til at implementere den kontinuere generering af terræn i selve spillet.

Perspektivering

Der er rigtig meget som er gået godt i projektet som ikke kan ses. Som vi kan se på klassediagrammet består spillet faktisk af pænt mange komponenter. DeferredRenderer og TextRenderer er avancerede komponenter, som ligger til grund for meget af programmet. De er bare ikke direkte krav som bliver testet på. Programmet er faktisk meget tæt på at virke. En uge med programmeringstimer mere, så havde alle krav været opfyldt.

Kigger man tilbage på projektet er det klart at der har været problemer. Meget af programmet afhænger af renderingen, for at kunne visualisere spillet. Dette betyder, at det har været svært at arbejde parallelt, fordi det kun en af os som ved hvordan GPU-programmering virker og GPU-programmering er ikke noget man bare lige sætter sig ind i.

Det meste af tiden er gået på at bygge de underliggende systemer til spillet i stedet for at bygge selve spillet. Nu var en af forudsætningerne, at man ikke måtte bruge game engine, så dette har vi ikke gjort, men en anden gang ville vi nok klart foretrække en løsning som bevy (en tekstbaseret game engine) i stedet for at skulle skrive det hele selv fra bunden.

Konklusion

Det kan konkluderes at der er mange komponenter og avancerede systemer der ligger til grund for et computerspil og selvom vi har fået implementeret mange af disse, så har der simpelthen ikke været tid til at få selve spillet til at virke, da rendering har været en kæmpe bottleneck som forudsagt. Mange af kravene er ikke opfyldt, men dog er en del tæt på. Med mere tid kunne de godt komme til at virke.

Bilag

Tidsplan

12/12/2022:

Implementer et Renderer Hardware Interface (RHI) og begynd på physically-based-rendering motoren - Sebastian

Implementer audiosystem og ”tensor” scene - Aksel

09/01/2023:

Implementer en fps-controller, så man kan styre spilleren - Aksel

Begynd på at skrive en simpel state-machine til zombierne - Aksel.

11/01/2023:

(forhåbentligt er 3D-motoren samt text-motoren færdig her)

(forsat fra sidste gang - Aksel)

Implementer hit-detection på zombier - Sebastian

16/01/2023:

Implementer damage fra zombies - Aksel

17/01/2023:

Ekstra tid, hvis noget skrider ellers implementer ”nice-to-have” features

06/02/2023:

Ekstra tid, hvis noget skrider ellers implementer ”nice-to-have” features

08/02/2023:

Ekstra tid, hvis noget skrider ellers implementer ”nice-to-have” features

Logbog

28/11/2022:

Begge syge ☹

30/11/2022:

Begge syge ☹

05/12/2022:

Aksel syg ☹

Sebastian begyndte på RHI interface og lavede research på hvordan man bedst bygger en voxel engine

06/12/2022:

Aksel og Sebastian diskuterer om hvorvidt de skal bruge en scene-graph eller en kæmpe sparse-tensor, hvor vi kom frem til at lave begge dele.

Desuden blev der skrevet mere på RHI interfacet.

12/12/2022:

RHI virker tilfredsstillende til at begynde at bruge det til at bygge en basic renderer. Aksel kigger på at bygge en scene-graph.

09/01/2023:

Sebastian Implementerede en fps-controller, så man kan styre spilleren og lavede renderen om til deferred rendering i stedet for forward rendering for bedre performance.

Aksel fik skrevet unit tests og begyndt på at implementere sparse tensoren.

11/01/2023:

Deferred rendering er ikke færdig, så denne dag bruges på at debugge sort-skærm problemer. Der kommer ikke noget frem på skærmen.

Aksel er done med sparse tensor, men kan ikke visualisere om det virker, da renderen ikke kan tage imod den endnu.

16/01/2023:

Deferred rendering virker nu, men den kan stadigvæk ikke understøtte både scenegraphen og sparsetensoren.

17/01/2023:

I dag begyndte Aksel så småt at kigge på hvordan man kunne gå til den opgave at generere terræn.

Scenegraphen er nu implementeret, så denne kan visualiseres og testes

06/02/2023:

Sebastian fixede flere issues med text-rendering. Flere sort-skærms debugging helvede.

Aksel implementerde en FPS-controller vha. scenegraphen og gjorde så man kunne kigge rundt med musen.

08/02/2023:

Sparsetensoren kan endelig visualiseres, så Aksel kigger på terrængenerering.

18/02/2023:

Sebastian lavede således at pistolen følger med spilleren og en jump animation.

Aksel arbejdede på terrængenerering.

19/02/2023:

Sebastian kodede, at fjender kunne spawnes ind og lavede på skydemekanikken

Aksel arbejdede igen på terrængenerering.