Rapport lab 6 – Gruppe 10

DAT155 – 20. november 2020

Gruppa:

Gudsteinn Arnarson – 577854

Stian Grønås - 578007

Kristian Eliassen – 578055

Bilete av spelet:

Innhald

Abstrakt

Gruppen sitt mål med dette prosjektet var å starte på å lage et FPS (First Person Shooter) spill, hvor brukeren/spilleren interagerer med en verden. Kontrollene til en FPS er vanligvis gjort slik at en styrer hvor spilleren går i verden med taster og ser seg rundt ved å bruke data-musen. Spilleren skal holde en pistol som skal kunne skytes. I tillegg vil spilleren kunne hoppe opp på objekter. I spillet kan også spilleren bytte mellom å holde en pistol og en lommelykt. Lommelykten skal kunne lyse opp et lite område foran spilleren.

I dette prosjektet ble det laget en scene som fulgte en oppgåve for faget. Gruppen valgte å lage en scene med en del interaktivitet for brukeren/spilleren. I tilegg ble det laget en verden som spilleren befinner seg i. denne verdenen består av et terreng, en innsjø, vann/dam, trær i terrenget, et hus, en stei, noen kuber og en himmel. I tillegg kan spilleren styre en sol som beveger seg langs en kurve. Det ble brukt forskjellige avanserte teknikker for å legge til de forskjellige delene i scenen.

1 Innleiing

Siden gruppens fokus var å lage noe interaktivt, ville gruppen lage en FPS. Spilleren må altså ha noe å gå på som et terreng. I tilegg ble det lagt til en del modeller som trær, vann, en innsjø, et hus og noen kuber. Spilleren kunne også bytte mellom en lommelykt og et våpen å skyte med. Gruppen la derfor til en fugl som en kunne skyte slik at fulgen falt i bakken. Gruppen la også til en sol som en kan kontrolere hvilke retning den går på en kurve over terrenget.

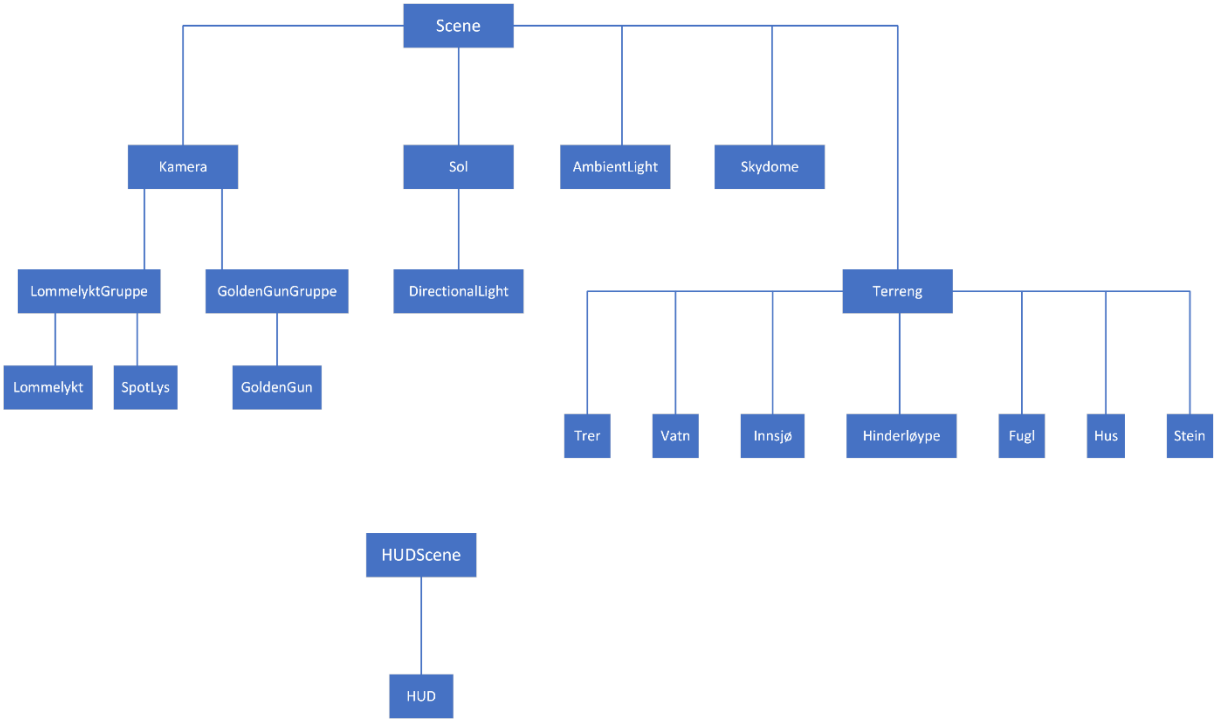
I rapporten blir oppgåva forklart og scenegrafen blir vist som eit bilde og så kort forklart. Vidare blir det kort fortalt val av nokre metodar og grunngjeving for desse. Arkitekturen til spelet blir vist som eit klassediagram. Så blir det forklart i detalj korleis dei ulike delane av spelet verker med litt bilete og kode for å vise. Til slutt kjem ein konklusjon av kva me fekk til og det me lærte og kva me ville eventuelt jobba vidare med på spelet vårt.

2 Oppgåvebeskrivelse

Oppgaven gruppen gikk ut på å lage en scene i threejs med bruk av avanserte teknikker. I oppgaven var det noen absolutte krav, noen teknikker som burde være med og til slutt noen eksempler på andre teknikker en kunne integrere i scenen. De absolutte kravene var å ha en scene implementert i threejs, bruk av avanserte teknikker og bruk av klasser. I tillegg var det også et krav å lage en custom shader, legge til et terreng i scenen og at en skulle kunne bevege seg rundt om i scenen.

I bilete 2.1 kan ein sjå scenegrafen. Den er delt opp i «3 delar» der me har dei tinga som ligg direkte under scenen, som sol, anna lys og skydome, og dei andre delane med kamera og terrenget. Kameraet er normalt ikkje med i ein scenegraf, men sidan me ville ha modellar og slikt som føl med kameraet måtte også kameraet liggje i scenegrafen. Kameraet har lommelykta og våpenet som er lagt til i to ulike grupper for å enkelt kunne skjule og vise det ein ynskjer. Under terrenget ligg alle dei tinga som er plasserte ut i terrenget for at dei skal kunne følgje med om me gjer noko med terrenget elles. Det er også lagt med scenegrafen for HUD (kap. 3.3.15) og den er liten då det er berre HUD som er med i den grafen.

Bilete 2.1 – Scenegrafen



3 Løysingsmetode

3.1 Val av metode

Sidan me ikkje ville fokusera på terrenget så mykje brukte me berre det som kom med startkoden. Via nettsida me henta ut høgdedata kunne me også hente ut ei punktsky av eit område så dette kunne alternativt vert ein måte å få laga terrenget vårt på.

For kamera og kontroll brukte me PointerLockControls og kode frå Pointer Lock eksempelet fordi det passar betre til eit FPS spel, og det brukar Pointer Lock API-et som er viktig for oss slik at musepeikaren ikkje går utanfor vindauget til nettlesaren. Her var det andre alternativ som å bruke den koden som kom med startkoden og andre eksemplar frå three.js men dei passa ikkje like godt og måtte ha gjort meir endringar for å få det til å verke med vårt program.

For å få til vatn kunne me brukt three.js sine eksemplar med vatn og hav, men me ville prøve å få til noko med modifisering av shader-kode som var enklare å forstå. Me brukte three.js sitt CubeCamera for å få til refleksjon men her kunne me også ha brukt meir avanserte metodar som eksempla til three.js.

Med HUD var det fleire ulike alternativ, me kunne brukt sprite som me plasserte på kameraet ved å for eksempel manuelt opprette ein tekstur og lage ein sprite som ein legg på kameraet, eller gjort alt gjennom HTML. Det er også fleire andre alternativ [49]. Det me valte var enkelt å forstå og lett å oppdatere og endre mens programmet køyrde og det var stor moglegheit for å få til ulike ting som tekst og bilete.

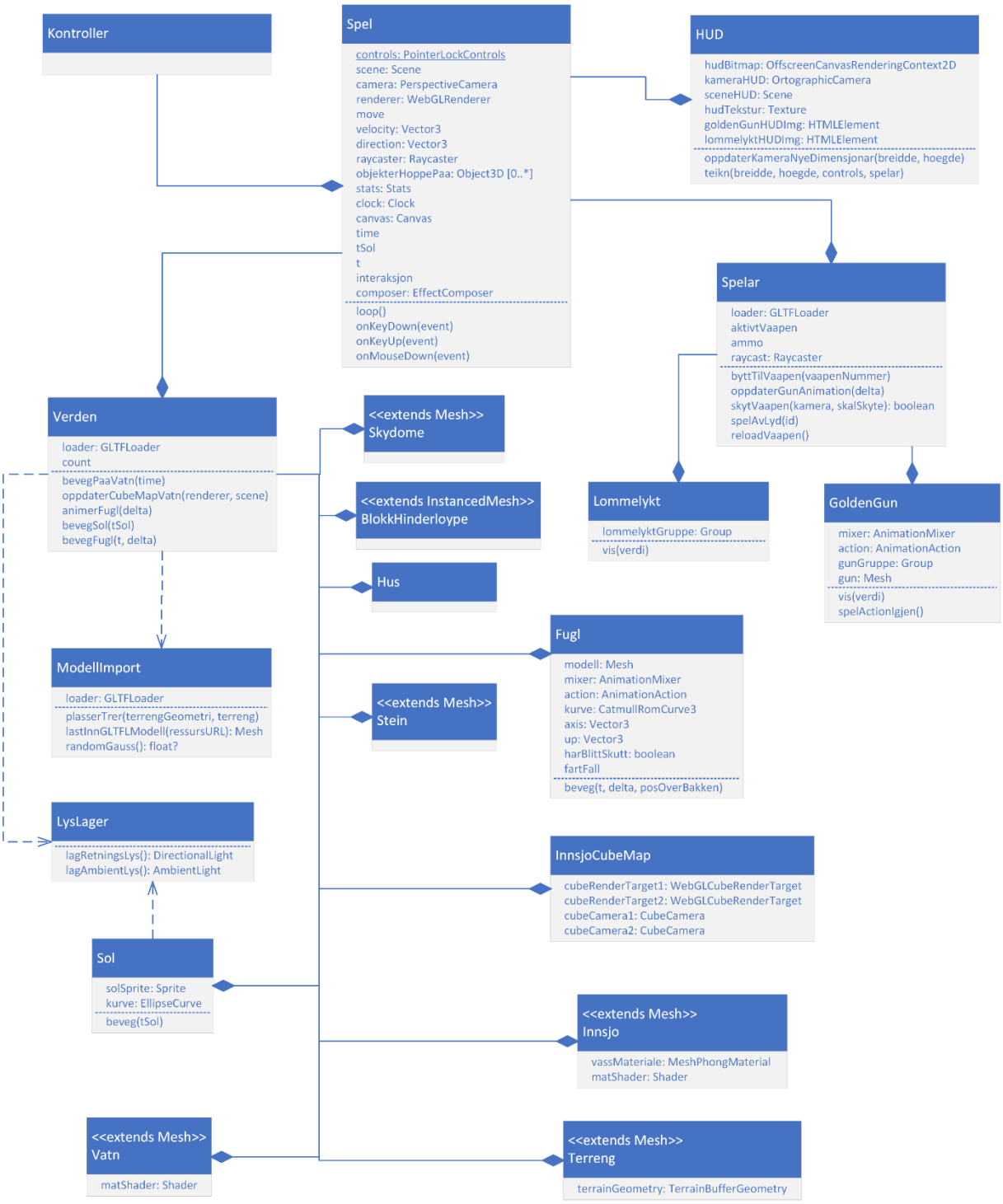
For valget av sol valgte gruppen å bruke sprite siden det var noe nytt og enkelt en kunne sette seg inn i. i tillegg valgte gruppen å lage en kurve som solen kunne følgje slik at lyset som var barn av solen, ble dynamisk. Dette var relativt simpelt og veldig morsomt og lærerikt.

Grunnen til at huset ble lagt til i scenen var å gi scenen litt meir utfylling og gruppen ville også da lære seg å importere .gltf modeller frå nettet.

Skydome ver også en viktig komponent av scenen som ga meir realisme til at brukeren befinner seg i en meir virkelig verden. I tillegg var dette en simpel og lærerik teknikk å få til.

3.2 Arkitektur

Bilete 3.2.1 – Klassediagram over arkitekturen



3.3 Detaljert design

3.3.1 – Kamera, kontroll og bevegelse

Kameraet me brukar er berre eit vanleg PerspectiveCamera frå three.js for å få ein perspektivvisning. For styring av kamera brukar me eksempelet PointerLockControls [1]. Denne klassen gjer det enkelt å styre kameraet med musa, der ein blant anna kan setje kor stor vinkel ein skal kunne sjå opp og ned (kamera pitch) med max/minPolarAngle. Standardinnstilliingar gjer at ein berre kan sjå mellom rett opp og rett ned, altså at ein ikkje kan fortsette å flytte musa nedover og så til slutt ende opp på same punkt. Klassen tar i bruk PointerLock API-et [2] som «fangar» musepeikaren til vindauget og gjer at klassen kan lese av hendingar slik at den kan hente ut rørsla til musa og setje kamera slikt. Den oppdaterer quaternion-en til kameraet når den har rekna ut euler-vinkelen for å rotere kameraet. Objektet av denne klassen blir lagra som eit statisk objekt på Spel-klassen då det var litt trøbbel med å få den inn i ein metode som skulle køyrast når ei hending hadde skjedd.

PointerLockControls har også to metodar, moveForward og moveRight, som gjer det enkelt å flytte seg parallelt med xz-planet uavhengig av kor kameraet peikar i scenen. Via hendingar kan me sjekke om tastane «wasd», mellomrom og fleire har blitt trykka, og så beveg kameraet basert på dette i loop-funksjonen vår som driv spelet. Her bruker me metodane onKeyDown, onKeyUp og onMouseDown i Spel-klassen som blir køyrd når ein trykker på knappar eller på musa. Desse metodane set boolske variablar som ligg i «move» og «interaksjon»-variablane til Spel-klassen som blir brukt for å finne ut kva ein skal gjere i loop-metoden.

Det er ein eigen styringsbolk i loop-metoden til Spel-klassen som har med å oppdatere posisjonen til kameraet via PointerLockControls, og blir berre køyrd når PointerLockControls har fanga musepeikaren til vindauget via isLocked attributten. Denne koden er henta frå eit eksempel (controls / pointerlock) og så modifisert til vårt bruk [3]. Det blir lagra fart og retning som vektorar, og dette blir oppdatert for kvart bilde eller «frame». Farta gjer at det blir meir naturleg rørsle då ein har ein liten «oppstart» og at ein ikkje stoppar med ein gong ein slepp tasta. Delta (som er kor lenge sidan førre frame var vist [4], og då vil vere ca. 16,7 ms for 60 FPS) blir brukt for å kunne få like stor forflytting uavhengig av bildefrekvens (liten bildefrekvens vil derimot kunne gje hopping rundt om kring). I kodesnutt 3.3.1.1 vil ein kunne sjå korleis oppdateringa av rørsla i xz-planet har blitt gjort. Number() blir brukt for å gjere om ein boolsk verdi til 1 eller 0 for å finne ut om det skal leggjast til eller fjernast på farta [5].

Kode 3.3.1.1 – Oppdatering av rørsle i Spel.loop()

```
this.velocity.x -= this.velocity.x * 10.0 * delta;
this.velocity.z -= this.velocity.z * 10.0 * delta;
this.direction.z = Number(this.move.forward) - Number(this.move.backward);
this.direction.x = Number(this.move.right) - Number(this.move.left);
if (this.move.forward || this.move.backward) {
      this.velocity.z -= this.direction.z * 400.0 * delta;
}
if (this.move.left || this.move.right) {
      this.velocity.x -= this.direction.x * 400.0 * delta;
}
Spel.controls.moveRight(- this.velocity.x * delta);
Spel.controls.moveForward(- this.velocity.z * delta);
```

For å få kameraet til å følgje terrenget brukar me terrenggeometrien sin getHeightAt(x,z) metode for å finne ut kva høgda i terrenget er og set kameraet til å bruke den y-verdien og litt ekstra for å få den over terrenget. Denne metoden brukar bilinær interpolasjon for å finne høgda og er uendra frå startkoden. Ein kan sjå koden for at kamera før terrenget i kodesnutt 3.3.1.2. Det blir også sjekka om ein er i lufta eller på eit objekt som ein kan stå på slik at ein ikkje blir «låst» til terrenget.

Kode 3.3.1.2 – Oppdatering av y-posisjon til kameraet i Spel.loop()

```
let terrengPosHogde = this.verden.terreng.terrengGeometri.getHeightAt(this.cam
era.position.x, this.camera.position.z);

if (!faller && !erOverBakken && !onObject && this.move.canJump) {
      Spel.controls.getObject().position.setY(terrengPosHogde + 3);
}
```

Me har også implementert hopping, og hopping på visse objekt i scenen ved hjelp av three.js sin Raycaster [6]. Kode er henta og så modifisert frå same eksempel for å bevege seg med fart i terrenget [3]. Som med fart i x- og z-retning, blir det også lagt til fart i y-retning. Her er farta basert på formelen for tyngdekraft: F = mg med g = 9,8 m/s. Når ein trykker på mellombaren blir det lagt til fart i y-retning oppover i onKeyDown-metoden og kameraet vil bevege seg oppover for å så nå maks høgde og falle ned mot terrenget igjen.

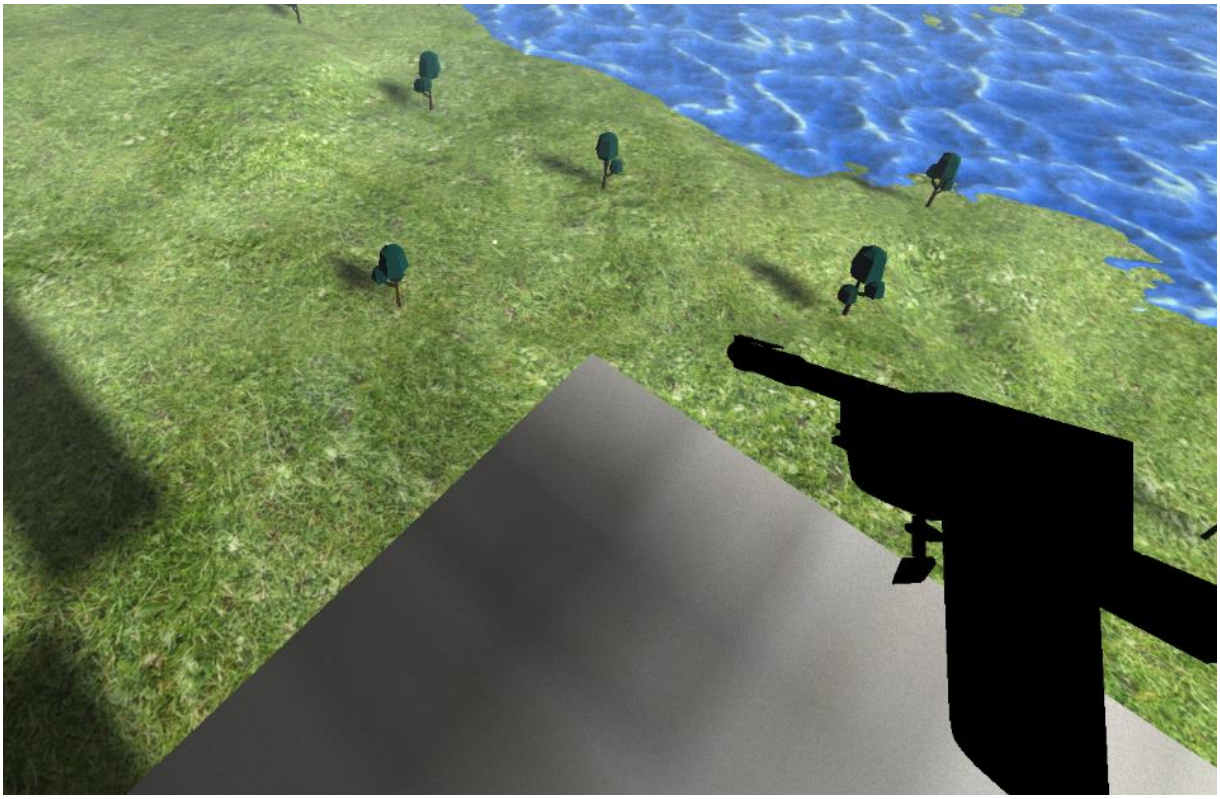
For å hoppe på objekt brukar har me ein Raycaster som peikar ned (negativt langs y-aksen). Når loop køyrer blir starten til stråla som Raycaster-en sender ut satt til posisjonen til kameraet, og så sjekkar me mot ein tabell av objekt om stråla har treft eit av dei objekta med Raycaster sin intersectObjects. Her vil det enkelt vere mogleg å leggje til Object3D eller barn av denne for å kunne hoppe på dei, men med fleire objekt vil den måtte sjekke meir og det vil ta meir prosessorressursar. Om stråla

trefte eit objekt vil farta bli satt til 0 så lenge den er negativ og ein kan hoppe på nytt. I kode 3.3.1.3 kan ein sjå kode for oppdatering og sjekking av Raycasteren. I kode 3.3.1.2 kan ein sjå at me ikkje set y-posisjonen til kameraet om ein har hoppa på eit objekt eller er i lufta. Bilete 3.3.1.4 visar at spelaren har hoppa opp på eit objekt i scenen og ser ned på terrenget.

Kode 3.3.1.3 – Raycaster for å hoppe på objekt i Spel.loop()

```
this.raycaster.ray.origin.copy(Spel.controls.getObject().position);
this.raycaster.ray.origin.y -= 3;
let intersections = this.raycaster.intersectObjects(this.objekterHoppePaa);
let onObject = intersections.length > 0;
if (onObject === true) {
      this.velocity.y = Math.max(0, this.velocity.y);
      this.move.canJump = true;
}
Spel.controls.getObject().position.y += (this.velocity.y * delta / 5);
```

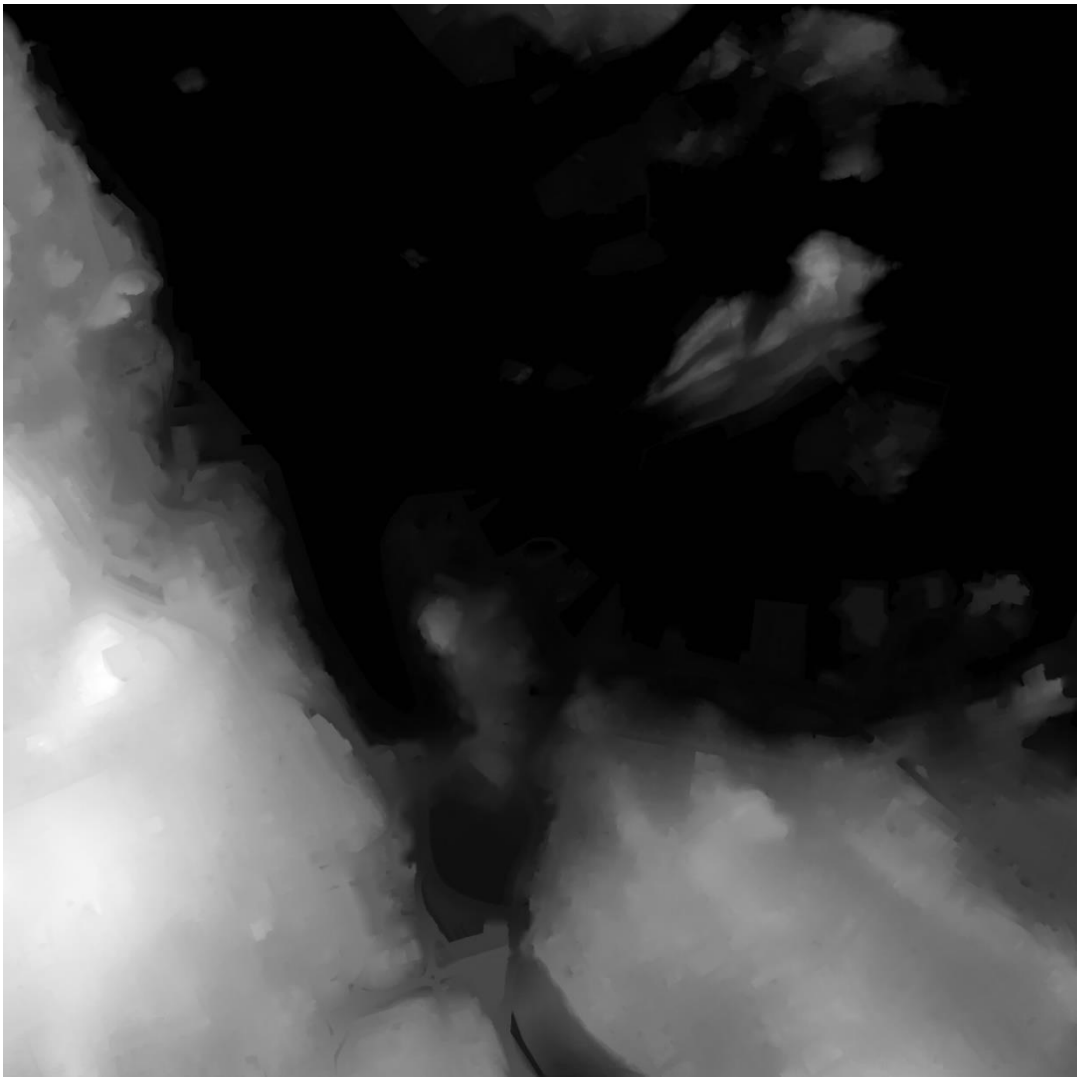
Bilde 3.3.1.4 – Kamera på eit objekt over terrenget



3.3.2 – Terreng

For terrenget ville me ha eit terreng frå Stavanger-området. Me henta ut eit DTM datasett i GeoTIFF-format frå kartverket si høgdedata-nettside over ein del av Stavanger [7]. Så ved hjelp av eit program som heitar QGIS [8] kunne me setje saman GeoTIFF-filene til eit komplett høgdekart ved hjelp av merge-funksjonen i programmet [9]. Deretter kunne ein klippe ut eit område som var kvadratisk for å bruke terrenggenereringa frå startkoden. Så kan ein eksportere til eit bilde ved hjelp av translate og export-funksjonane i QGIS og bruke dette i terrenggenereringa [10]. Bilete 3.3.2.1 visar det høgdekartet me brukte i programmet vårt.

Bilete 3.3.2.1 – Høgdekart over Stavanger



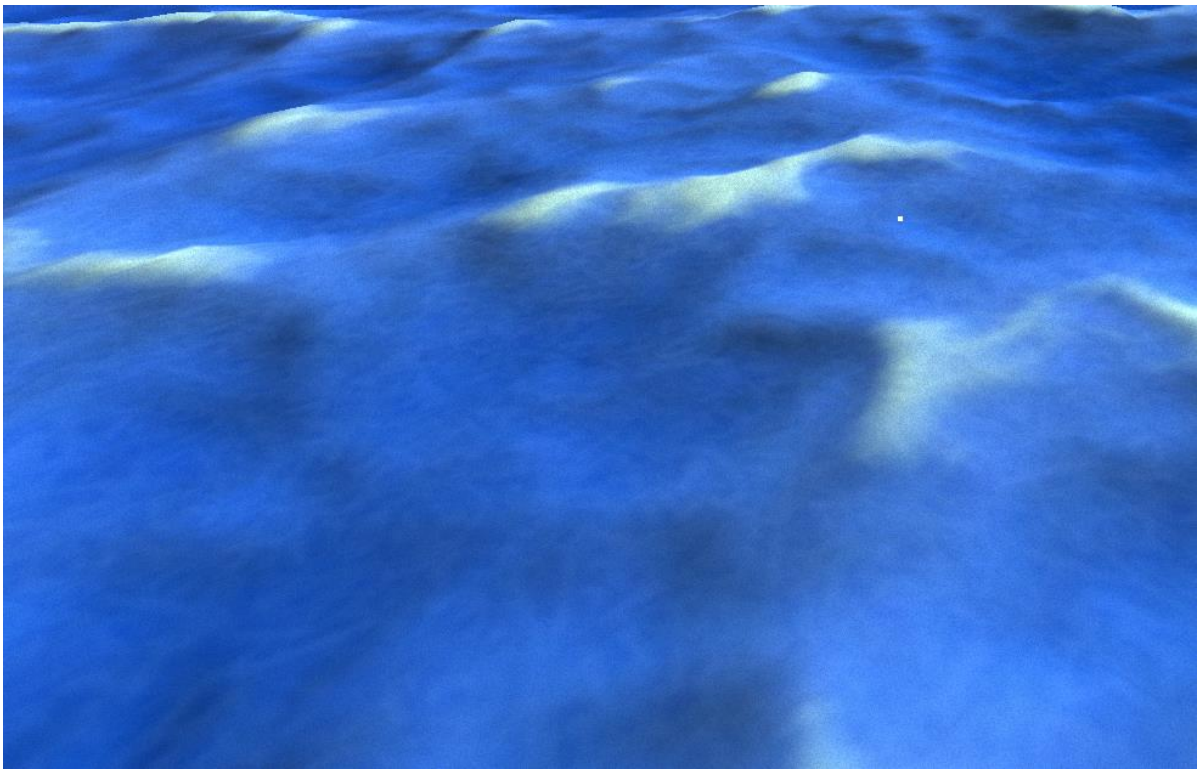
Me endra ikkje noko særleg på resten av koden for terrenget enn det som kom med startkoden frå Canvas. Det blei endra litt på høgde og inndelingar for å passe betre med vårt bilde. For splatting av teksturar og kva slags teksturar så brukte me berre dei som kom med startkoden og endra ikkje noko på det.

3.3.3 – Vatn og innsjø

Vatnet i terrenget laga me i klassen Vatn ved å bruke PlaneBufferGeometry for å ha ein litt meir effektiv representasjon [11], og så MeshStandardMaterial for å få eit materiale som blir mørklagt ved hjelp av Phong, men også at den er «fysisk korrekt» [12]. Den er ikkje heilt konfigurert rett og me burde ha brukt MeshPhongMaterial for å spare ressursar mot at den ikkje ser like realistisk ut [13]. Her henta me inn nokon teksturar frå nettet som ga ein fin utsjånad på vatnet [14]. Her var det eit displacement map som gjer at vertex-ane til mesh-en blir flytta basert på den teksturen [28]. Det var også eit ambient occlusion map som gjer at ein kan rekne ut kor mykje kvart punkt er eksponert til ambient lys [15]. Til slutt var det eit normal map for å fikse på normalane.

For å få litt bevegelse i vatnet modifiserte me vertex-shaderen ved å leggje til ekstra glsl-kode før kompilering av shaderen [16]. Her fant me enkle sin/cos-funksjonar for å oppdatere posisjonen til kvar vertex og fikse normalen på nettet som ein kan sjå i kode 3.3.3.2 [17]. Desse funksjonane gjer at ein simulerer bølgjer i vatnet, men dei vil følje eit mønster og ikkje vere tilfeldige. Her kunne me ha brukt three.js sine Ocean eksemplar for å gjere det meir realistisk [18]. I bilete 3.3.3.1 kan ein sjå vatnet som me fekk til. For å oppdatere uniformvariabelen som me la til (time) slik at vertex-ane får ulik posisjon over tid blir denne oppdatert i Spel sin loop() metode som kallar på Verden sin bevegPaaVatn() metode som då set uniformen til ein ny verdi. Time er berre ein variabel som aukar med deltaverdien for kvar gong.

Bilete 3.3.3.1 – Vatn



Kode 3.3.3.2 – Bølgje og normal-funksjonar i vertex-shader

```
float wave(float time, float freq, float amp) {
      float angle = (time+position.y)*freq;
      return sin(angle)*amp;
}
float waveNorm(float time, float freq, float amp) {
      float angle = (time+position.y)*freq;
      return -amp * freq * cos(angle);
}

// ---

transformed.z += wave(time,freq,amp)
               + wave(time,freq*2.0,amp/2.0)
               + wave(time,freq*3.5,amp*0.2);
//likt for normalen også^
```

For innsjøen prøvde me å få til refleksjon ved hjelp av dynamisk cube map. For å lage eit cube map brukte me three.js sine CubeCamera [19] og WebGLCubeRenderTarget [20] klassar. CubeCamera vil lage 6 kamera som då rendrer det som er i scenen rundt CubeCamera til eit CubeRenderTarget. Ut frå dette CubeRenderTarget kan ein hente ut eit environment map (eller cube map) tekstur som ein set på materialet til innsjøen. Innsjøen blir laga med eit PlaneBufferGeometry som geometri og eit MeshPhongMaterial som materiale. EnvMap-parameteren blir satt til teksturen frå CubeRenderTarget.

For å oppdatere cube map til innsjøen har me ein metode i Verden, oppdaterCubeMapVatn(…), som blir køyrd i loop() i Spel-klassen. Her har me brukt 2 CubeCamera og CubeRenderTarget for å få til ein ping-pong / dobbel-buffer effekt der me oppdaterer det eine CubeCamera-et og hentar ut teksturen frå den andre CubeRenderTarget-en, og så bytter om ved neste kall på metoden [21]. Dette kan ein sjå i kode 3.3.3.3 under.

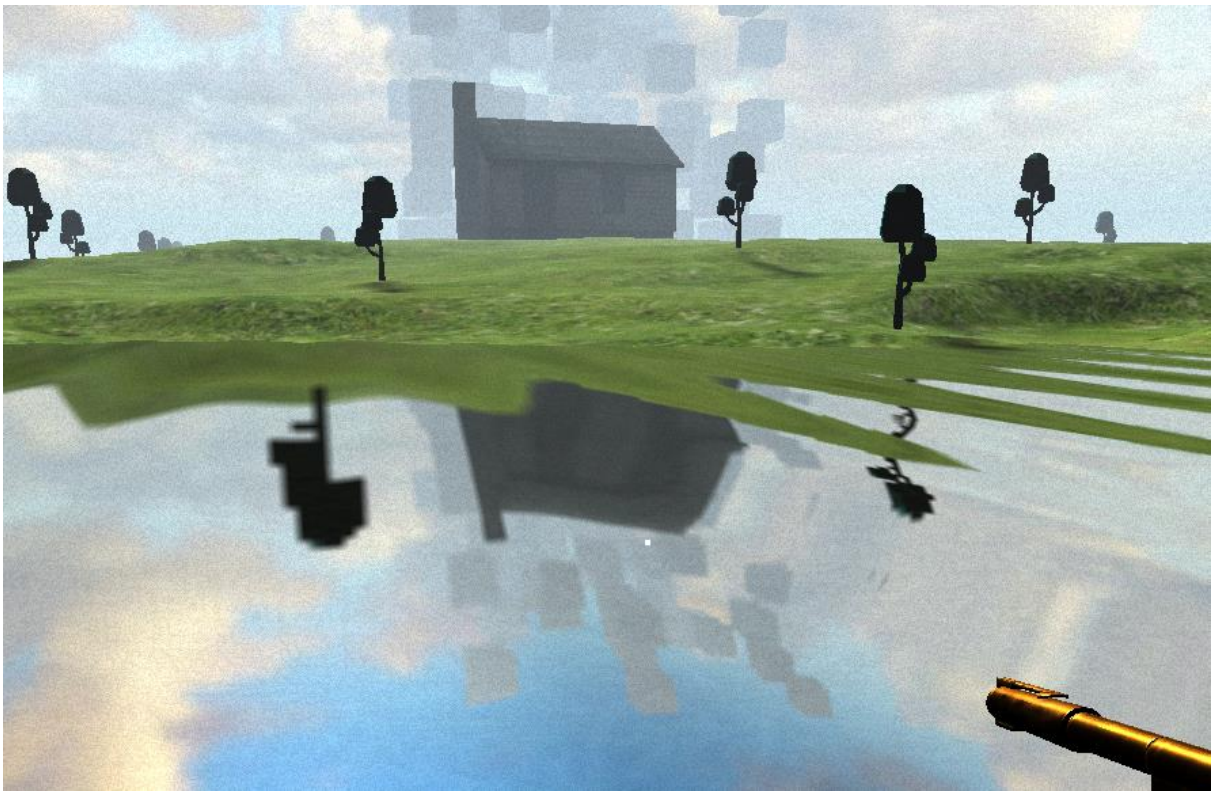
Kode 3.3.3.3 – Oppdatering av CubeCamera og henting av tekstur

```
this._innsjo.hidden = true;
if (this.count % 2 === 0) {
       this._innsjoCubeMap.cubeCamera1.update(renderer, scene);
       this._innsjo.vassMateriale.envMap = this._innsjoCubeMap.cubeRenderTarget2.texture;
} else {
       this._innsjoCubeMap.cubeCamera2.update(renderer, scene);
       this._innsjo.vassMateriale.envMap = this._innsjoCubeMap.cubeRenderTarget1.texture;
}
this._innsjo.hidden = false;

this.count++;
```

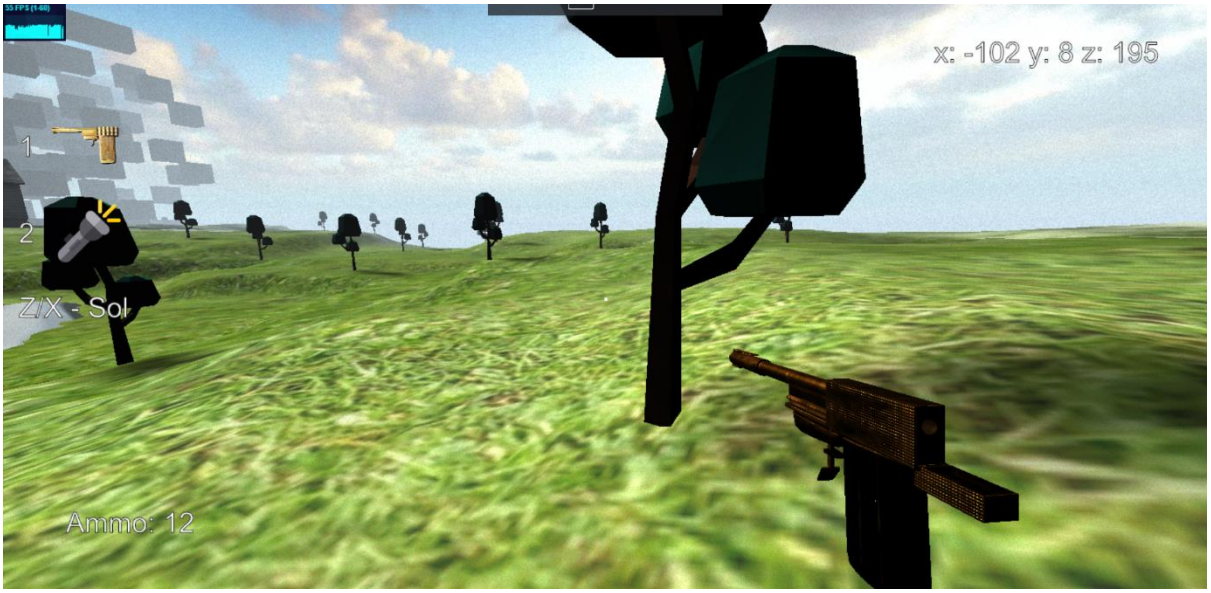
For å animere innsjøen med litt bølgjer blir vertex-shaderen modifisert med ein litt enklare bølgjefunksjon enn som blei gjort med vatnet, der me berre bruker wave og waveNorm funksjonane i kode 3.3.3.2 ein gong. Resultatet kan ein sjå i bilete 3.3.3.4. Oppløysinga er litt låg for å spare ressursar, då me skrudde ned størrelsen på CubeRenderTarget til 256.

Bilete 3.3.3.4 – Innsjø



3.3.4 – Trer og utplassering

Bilde 3.3.4.1 - Tre



Tre-modellen som er brukt er funnet frå startkode frå Canvas. Vi har lagd ei dobbel for-løkka som definere pixelområdet som trea kan bli plassert i. Disse verdiane er satt til –400 < x < 400 og x += 30, -400 < z < 400 og z += 30. x/z += 30 er den minste distansen som er tillaten mellom trea, for å forhindra at de blir plassert oppå kvarandre, eller for tett. For å få ein litt "random" og variert utplassering så har vi brukt randomGauss frå Canvas og lagt den til i plasseringskoordinatene til kvart tre slik:

Kode 3.3.4.2 – Koordinatar for trer

```
//litt tilfeldig plassering rundt dei punkta
const px = x + this.randomGauss(x);
const pz = z + this.randomGauss(z);
```

For at trea ikkje skal bli plassert i sjøen eller i innsjøen så lagde vi ein const height, som finner høgda, i y-koordinat, til det utplasserte treet. Om height > 2 && height < 9 så blir treet plassert ut. Høgda til vatnet går opp til 1.8, så 2 > height < 9, gjer at trea ikkje er heilt nede i vasskanten, som er lite realistisk. Skulle gjerne hatt tid til å fått trea plassert i skogar, men fekk ikkje tid til å legge dette til.

3.3.5 – Hus

Gruppen brukte en GLTFLoader til å laste inn huset for så å legge det til i scenen. I starten brukte gruppen en bygning, men dette ble veldig krevende for datamaskiner. Dette var nok fordi at det var veldig mange teksturer/maps og en del kompleks geomtri på bygningen. Til slutt endte gruppen med å velje et litt meir simplere hus med 3 forskjellige tekstrurer/maps. Huset og bygningen ble hentet på Sketchfab saman med teksturer/maps. Huset ble manuelt plassert på en plass i scenen hvor det så mest naturlig ut.

Bilde 3.3.5.1 – Hus

3.3.6 – Fugl som føl kurve

Her importerte me ein fuglemodell frå Sketchfab [22]. Med denne modellen var det også animasjonar som me kunne leggje til i spelet vårt. Me brukte three.js sin AnimationMixer til å animere objektet [23]. Først oppretta me ein AnimationMixer med den importerte scenen frå GLTF-objektet, så henta ut ein animasjon (her var det berre ein) og lagra denne som ein AnimationAction med mixer sin clipAction(…) funksjon [24]. Køyrer ein action.play() vil då animasjonen kunne bli spelt av. Kode kan ein sjå i kodesnutt 3.3.6.1. For at animasjonen skal bli oppdatert mens spelet køyrar må ein også i loop() i Spel køyre AnimationMixer sin update(…) funksjon. Dette blir gjort via verden sin animerFugl() metode.

Kode 3.3.6.1 – Hente ut og starte animasjon

```
this._mixer = new AnimationMixer(object.scene);
this.action = this._mixer.clipAction(object.animations[0]);
this.action.play();
```

For å få fuglen til å fly over terrenget brukte me three.js sin CatmullRomCurve3 for å lage ei glatt kurve [25]. Dette er ein spline-kurve laga med Catmull-Rom algoritmen. Me laga 4 punkt rundt innsjøen i terrenget med litt ulik høgde for å få ein litt meir variert bane på fuglen og satt den til å vere ei lukka kurve. Kurva har to metodar, getPoint() og getTangent(), for å hente ut posisjon og tangenten langs ei kurve. Med posisjonen kan ein enkelt få modellen til å følgje kurva ved å kopiere over vektoren til modellen sin posisjon. Ved å gjere litt vektor-utrekning kan ein få satt rotasjonen fuglen skal ha som ein quaternion, her fant me ein guide på nettet [26]. Men me hadde eit problem ved at fuglen snur seg rundt på kurva og er opp-ned langs halve del av kurva som me ikkje fekk til å fikse. I kodesnutt 3.3.6.3 kan ein sjå utdrag av kode for å oppdatere posisjon og retninga til modellen. Me snur retninga på tangenten med multiplyScalar(-1) fordi modellen var feil retning når den blei importert og gjorde at den «flydde baklengs» i scenen utan denne endringa.

Rørsla til fuglen blir utført via loop() som kallar på Verden sin bevegFugl() som igjen kallar på fuglen sin beveg()-metode. Bilete 3.3.6.2 visar fuglen mens den er i korrekt «tilstand», altså at beina er nedst. I andre halvdel av banen vil beina vere øvst.

Bilete 3.3.6.2 – Fugl som går i bane

Kode 3.3.6.3 – Oppdatering av posisjon og retning for fuglen

```
this.axis = new Vector3();
this.up = new Vector3(0, 1, 0);

// ---

let pos = this.kurve.getPoint(t);
let tan = this.kurve.getTangent(t).normalize().multiplyScalar(-1);

if (this._modell) {
	this.axis.crossVectors(this.up, tan).normalize();
	let radians = Math.acos(this.up.dot(tan));
	this._modell.position.copy(pos);
	this._modell.quaternion.setFromAxisAngle(this.axis, radians);
}
```

3.3.7 – Stein

Me lagde ein stein som me plasserte i terrenget. For å lage steinen brukte me ein SphereBufferGeometry og eit MeshStandardMaterial. Me fant ein teksturpakke for stein på nettet som inkluderte displacement map for å gjere at det såg ut som ein stein, normal map, roughness map, ambient occlusion map og sjølve fargeteksturen [27]. Dette la me inn som parametre på materiale og då fekk steinen ein utsjånad som likna ein stein som ein kan sjå i bilete 3.3.7.1.

Bilete 3.3.7.1 – Stein

3.3.8 – Sol og anna lys i scenen

For å legge til en sol ble det diskutert at vi måtte legge til en kurve i scenene for så å la solen følgje denne kurven. Kurven var en EllipseCurve som konstrueres i Sol.js klassen saman med solen, og lyset solen skulle gi frå seg. Solen var av typen Sprite og lyset var et DirectionalLight med «target» eller retning mod midten av scenen. I starten følgte solen kurven, men lyset ble veldig rart når solen skulle bevege seg under terrenget. Vi valgte så å legge til at solen ble styrt ved å holde inne taster for hvilke retning solen skulle bevege seg. Vi satte også en maks grense på at solen skulle stoppe når den er rett under horisonten/terrenget.

Sjølv om Sol.js klassen legger til lyset på solen og retningen i scenen, så ble det også laget en LysLager.js klasse som har funksjoner for å opprette Directional light og bestemmer skyggene. Det er også en metode for å lage ambient light som blir lagt til i scenen i Verden.js klassen. Dette ble lagt til for å få .gltf modellene til å se lysere ut, siden de var veldig mørke når de ble lagt til i scenen.

Bilde 3.3.8.1 - Sol

3.3.9 – Våpen

Sidan dette skulle vere ein FPS fant me ein modell av eit våpen på Sketchfab [29] og importerte via GLTFLoader. Dette inkluderte også ein animasjon som fuglen og er animert på same måte. Det er ein liten forskjell på animeringa her, då på action-en som har animasjonen er den satt til å kun gjenta seg ein gong med setLoop(LoopOnce) og at ein set attributten clampWhenFinished til true. For å kunne spele av animasjonen på nytt kan ein kalle på action.play(). Våpnet har også blitt skalert ned og plassert nærme kameraet for at den ikkje skal klippe igjennom terrenget og andre ting ein er nærme.

Med modellen er det også fleire teksturar som ein diffus tekstur, ein normaltekstur for å få bump mapping og sjølve fargeteksturen. Loop() kallar Spelar sin oppdaterGunAnimation() for å oppdatere animasjonen til våpenet.

Bilete 3.3.9.1 – Våpen

3.3.10 – Skyting av fuglen

For å leggje til litt interaktivitet i spelet har me lagt til at ein kan skyte fuglen slik at den fell mot bakken etter at den har blitt skutt. Her bruker me igjen ein Raycaster som blei brukt til å få til å hoppe på objekt i scenen. Det er i Spelar-klassen logikken ligg for å «skyte» med Raycaster-en og sjekke om den treff noko. Logikken er lik som for hopping for å utføre sjekk om stråla treff noko, men nå set me at startposisjonen til Raycaster-en skal vere ut frå kameraet ved hjelp av setFromCamera() metoden slik at den sender eit stråle ut langs negativ z-akse frå kameraet (altså der midten av skjermen peikar) [30]. For å gjere sjekken om ein treff noko raskare blir det berre sjekka mot fuglen og ikkje noko anna, men her kunne ein enkelt lagt til fleire objekt som ein kunne ha skutt på.

Når ein trykkar med musa blir attributten interaksjon sin harSkutt satt til sann med mouseDown hendinga. Metoden skytVaapen() i Spelar blir utført i loop() når harSkutt er sann og returnerer om ein skøyt fuglen eller ikkje. Om den har blitt skutt blei ein attributt «harBlittSkutt» på fuglen satt til true. Når då fugl sin beveg() blir kalla på gjennom loop() og verden-objektet bytter den logikk til at animasjonen stoppar ved å kalle på action.stop(), og det blir lagt til fart i negativ y-retning slik at den fell mot bakken. Når y-posisjonen er likt som ved bakken blir ikkje farta oppdatert meir og fuglen stoppar og blir værande der til ein startar spelet på nytt. Kode kan ein sjå under i kodesnutt 3.3.10.1.

Kode 3.3.10.1 – Oppdatering av fuglen når den har blitt skutt

```
beveg(t, delta, posOverBakken) {
      if (!this._harBlittSkutt) {
            //sjå kode 3.3.6.3
      } else {
            this.action.stop();
            if (posOverBakken < this._modell.position.y) {
                this.fartFall -= 9.8 * 5.0 * delta; // 5 = massen
                this._modell.position.y += this.fartFall * delta;
            } else {
                this.fartFall = 0;
                this._modell.position.y = posOverBakken;
            }
        }
```

Det er også lagt til ein attributt på spelaren som held styr på kor mykje ammunisjon våpenet har, er det tomt vil ein ikkje kunne skyte fuglen og må lade om våpenet med r-tasta der ei hending plukkar det opp og legg på verdi på attributten igjen. I bilete 3.3.10.2 kan ein sjå at fuglen ligg på bakken etter at den har blitt skutt. Om fuglen er på den delen av kurva der føtene er mot himmelen vil modellen klippe inn i terrenget då origo til objektkoordinatane ligg i føtene til fuglen.

Bilde 3.3.10.2 – Fugl er skutt og ligg på bakken

3.3.11 – Lommelykt

For lommelykta fant me igjen ein modell som me importerte frå Sketchfab [31]. Den blei skalert ned og flytta nærme kamera for å ikkje klippe inn i andre objekt like lett som med våpenet. For å få til eit lys frå lommelykta brukte me three.js sin SpotLight [32]. For parametre til lyset valte me ein litt varm farge og satt distansen lyset skulle verke og vinkelen til lyset til å vere mindre verdiar slik at ein får eit lys som er ei kjegle ut frå kameraet inn i scenen ein kort distanse. Her brukte me 100 på distanse, 0.5 på vinkelen og 2 på decay for å få litt meir realistisk dimming av lyset desto lengre ut lyset gjekk. Me skrudde ikkje skygge på lyset då ein ikkje ser noko særleg av skygga fordi objektet blokkerer sjølv for at ein kan sjå skygga. Dette vil også spare litt ressursar. I bilete 3.3.11.1 kan ein sjå at me har peikt lyset på eit hus og lyser berre opp ein mindre del av det me ser på (den litle kvite firkanten er midten av skjermen).

For å få lyset til å lyse ut frå kameraet som om ein hadde ei lommelykt i auga sat me posisjonen til spotlyset rett bak kamera og så spesifiserte at målet til lyset skulle vere kameraet [33]. Dette gjer at når ein flytter på musa for å bevege kameraet vil dermed lyset oppdatere retninga den lyser etter kor kameraet peiker nå. Det skjedde derimot ein feil der når lyset blir skrudd på vil det bli svarte kantar på terrenget i distansen (sjå bilete 3.3.11.2), og me fant ikkje ut kvifor dette skjedde.

Me la også til det å bytte mellom våpenet og lommelykta ved å bruke hendingar og så berre setje dei respektive objekta sine «hidden» attributt som sann eller usann. For å enklare få med lyset laga me ei three.js Group [34] og la til lommelykt-modellen og lyset på gruppa slik at ein kunne enkelt skjule eller vise begge to på likt. Gruppa ligg så under kameraet for å følgje med rørsla spelaren gjer elles.

Bilete 3.3.11.1 – Lommelykt og lys

Bilete 3.3.11.2 – Svart kant i terrenget

3.3.12 – Instansierte kuber («hinderløypa»)

Frå eksempelet som me brukte for å implementere rørsle og kontroll av kameraet var det også brukt blokker til å vise korleis ein kan få til å hoppe [3]. Dette ville me vidareføre ved å ta i bruk three.js sin InstancedMesh for å spare på teiknekall til grafikkprosessoren og raskt og enkelt kunne teikne mange blokker på likt [35]. Me brukte 100 instansar for våre blokker. For geometrien brukte me BoxBufferGeometry og for materiale brukte me 6 materiale (eitt for kvar side) for å enkelt lage ein tekstur for kuben med fleire ulike teksturar brukt på dei ulike sidene [36]. Me brukte MeshPhongMaterial på alle materiala for at dei skulle få korrekt lys på seg. Kunne også brukt CubeTextureLoader men me fekk den ikkje heilt til med vår konfigurasjon. Teksturane var henta frå ein Minecraft teksturpakke [37], og ein kan sjå resultatet i bilete 3.3.12.2.

For å få plassert kubane brukte me den random-funksjonen som var i eksempelet [3] med PointerLockControls og modifiserte litt slik at dei blir flytta til ein annan plass i terrenget og at den passar med vår størrelse på kubane (som er 10x10x10). Me laga så ei 4x4 matrise og brukte setPosition() metoden for å lage ein translasjonsmatrise [38]. Med InstancedMesh sin setMatrixAt() metode kunne me oppdatere matrisa til kvar enkel kube slik at den fekk rett posisjon i terrenget, og i kode 3.3.12.1 kan ein sjå koden for dette. Mesh-en sin instanceMatrix sin needsUpdate blei satt til sann for at renderen skal få med seg at me har endra matrisene til instansane. Ein kan også hoppe på alle desse ved at ein legg til mesh-en i tabellen som Raycaster-en skal sjekke.

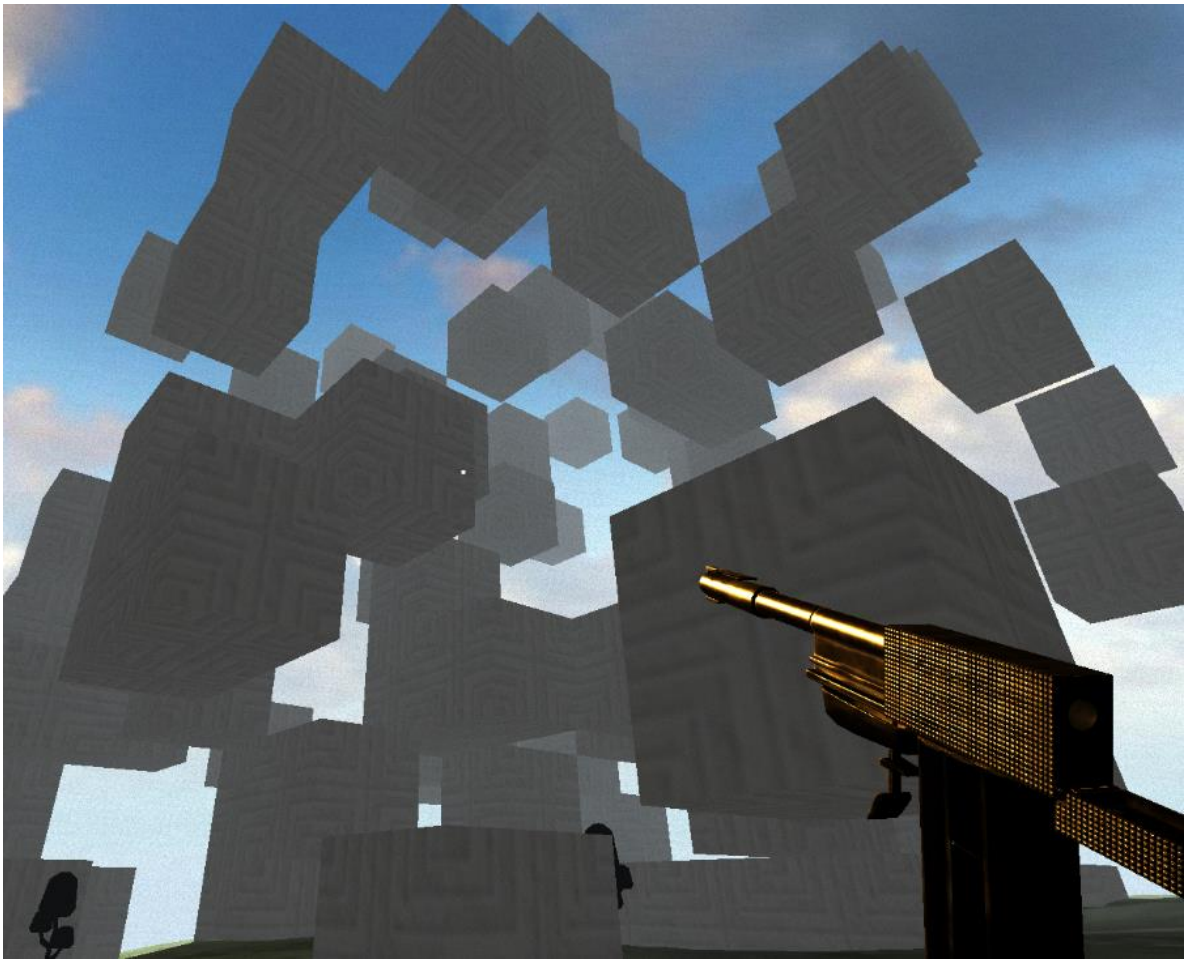
Kode 3.3.12.1 – Random-funksjon og oppdatering av posisjon for kvar instans

```
for (let i = 0; i < antInstanced; i++ ) {
      let x = 300 + Math.floor( Math.random() * 10 - 10 ) * 10;
      let y = Math.floor( Math.random() * 10 ) * 10 + 10;
      let z = 300 + Math.floor( Math.random() * 10 - 10 ) * 10;

      let matrise = new Matrix4().setPosition(x, y, z);

      this.setMatrixAt(i, matrise);
}
this.instanceMatrix.needsUpdate = true;
```

Bilete 3.3.12.2 – Instansierte kuber



3.3.13 – Skydome

Skydome som ble opprettet i scenen var veldig enkelt. Klassen Skydome.js oppretter en sfære som er ganske stor, med en tekstur som laster inn et bilde vi fant på nettet. For å vende om på hvor en ser polygonene frå velger vi å sette «side: backside». For at spilleren skunne kunne se skydomen med tåke i scenen ble også fog attributten satt til false i materialet slik at den blir synlig uavhengig av hvor mye tåke det er i scenen.

Bilete 3.3.13.1 – Skydome

3.3.14 – Post-processing

For post-prosessering har me testa litt med EffectComposer eksempelet frå three.js [39]. Denne tar inn renderen og så kan ein leggje til fleire ulike «Pass» som endrar det som skal blir vist på skjermen på ein eller annan måte. Det neste leddet i denne kjeda vil verke på det førre slik at rekkjefølgje vil vere viktig for kva ein vil oppnå med post-prosesseringa. For kvart ledd så rendrer denne EffectComposer-en til ein rendertarget, og rendrer ikkje til skjerm før det siste leddet (pass) [40]. I loop() metoden til Spel er det nå EffectComposer-en ein kallar render-metoden på i staden for renderer-objektet.

Her testa me litt med BokehPass først som skal leggje til depth of field i biletet [41], men me fekk det ikkje til. Me la også til FilmPass som legg til støy eller «film grain» og litt skannlinjer i biletet [42]. Over lengre tid vil skannlinjene og effekten bli litt vrengt og ser feil ut.

Bilete 3.3.14.1 – FilmPass

3.3.15 – HUD / UI

For HUD-en (heads-up display) ein ser i bilete 3.3.15.3 er det henta og kopiert direkte frå eksempelet om pointerlock [3] der berre tekst er modifisert. Denne er berre laga med HTML og CSS for å gjere den gjennomsiktig og mørker til skjermen når musa ikkje er låst til spelet.

I bilete 3.3.15.4 er det derimot blitt laga noko via three.js. Her opprettar me eit HTML OffscreenCanvas [43] og hentar ut eit OffscreenCanvasRenderingContext2D som me ved hjelp av Reference Canvas API-et kan teikne direkte til [44] [45]. Her kan ein setje font, størrelsen på tekst, fargen på tekst og meir. Ein kan også teikne med fillText(), strokeText(), drawImage() og meir.

For å få dette canvas-objektet til å bli rendra til skjermen laga me ein ny scene og eit OrthographicCamera [45]. Det blir brukt ortografisk kamera fordi me vil ikkje ha noko perspektiv på HUD-en. Vidare lagar me ein ny tekstur av canvas-objektet me laga, lagar eit MeshBasicMaterial då det ikkje skal reagere til lys og eit PlaneBufferGeometry. Så lagar me eit Mesh med materialet og geometrien og legg dette til den nye scenen. Sjå kode 3.3.15.1 for detaljar.

Kode 3.3.15.1 – Oppretting av HUD med canvas

```
let hudCanvas = new OffscreenCanvas(breidde, hoegde);
this._hudBitmap = hudCanvas.getContext('2d');

this._kameraHUD = new OrthographicCamera(-breidde/2, breidde/2, hoegde/2, hoegde/2, 0, 30);

this._sceneHUD = new Scene();

this._hudTekstur = new Texture(hudCanvas);
this._hudTekstur.needsUpdate = true;
let materiale = new MeshBasicMaterial({
       map: this._hudTekstur
});
materiale.transparent = true;

let planGeometri = new PlaneBufferGeometry(breidde, hoegde);
let plan = new Mesh(planGeometri, materiale);
this._sceneHUD.add(plan);
```

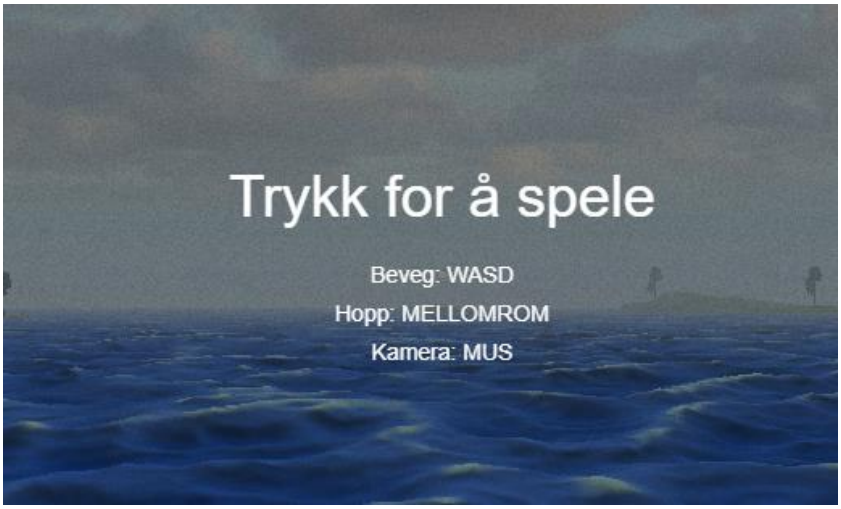
Nå kan ein i loop() løkka bruke scenen og kameraet me oppretta til å rendre HUD-en til skjerm over den originale scenen. Det er viktig at ein skrur av autoClear på renderer før ein rendrer HUD-en slik at den ikkje teiknar over scenen med clear-fargen og berre får HUD ut på skjermen (kode 3.3.15.2).

Kode 3.3.15.2 – Render av HUD

```
this.renderer.autoClear = false;
this.renderer.render(this.hud.scene, this.hud.kamera);
this.renderer.autoClear = true;
```

I HUD-klassen er det lagt til ein metode teikn() som vil bli kalla i loop() og som vil oppdatere HUD-en slik at den er dynamisk og kan endre seg. For eksempel blir det skreve tekst om kva xyz-koordinatane til kameraet er. Andre ting som me skriv til canvas-et er tal på ammunisjon som blir oppdatert, og litt info om kva knappar ein kan trykke på for å bytte våpen eller flytte på sola. I bilete 3.3.15.4 kan ein sjå denne HUD-en.

Bilete 3.3.15.3 – HUD med HTML



Bilete 3.3.15.4 – HUD med canvas og ny scene

3.3.16 – Andre småting

Me la også til Stats.js [46] for å enkelt kunne sjå bildefrekvens og anna statistikk. Me la også til three.js sin FogExp2 [47] med ein farge som me henta ut frå teksturen til skydome-en i området der terrenget låg på skydome-en for å få ein finare overgang til skydome-en. Her brukte me ein låg verdi på density for at berre ting langt vekke skulle bli tåkelagt. Bilete 3.3.16.1 visar forskjell mellom ting som er nærme og ikkje er tåkelagt mens det i bakgrunnen glir i lag med skydome-en som ligg bak terrenget.

Bilete 3.3.16.1 – Tåke

4 Konklusjon

Resultatet til gruppa var at me fekk laga eit interaktivt FPS. Ein kan blant anna bevege seg i terrenget med mus og tastatur. Vidare kan ein hoppe på visse objekt, som ein stein og ei «hinderløype» med kuber. Det går også ann å skyte ein fugl slik at den fell til bakken, og fuglen føl ein glatt kurve i scenen. Terrenget er laga etter eit høgdekart henta frå kartverket over Stavanger-området, men på litt stor skala. I terrenget blei også to ulike typar vatn lagt til, eit med meir fokus på å få til bølgjer og den andre for å få til refleksjon. Det blei lagt til andre ting som ei sol ein kan flytte på med tastaturet og at ein kan bytte mellom eit våpen og ei lommelykt som lyser opp framføre kameraet. Til slutt var det også nokon andre modellar som trer som blei utplassert og eit hus som var importert frå nettet.

Gruppa fekk ikkje implementert alt som var planlagt. Me ville blant anna fått til betre utplassering av trer slik at det dannar ein skog med mange trer og få utanfor denne skogen. Bruk av billboarding og sprite var det lite av og me fekk ikkje laga noko gras som me kunne plassert i terrenget. Post-prosesseringa var litt i siste liten og me fekk ikkje testa og implementert dei effektane me kunne ha ynskja oss. Me prøvde også med ein ray-tracing renderer [48] frå nettet som skulle fungera med ein del eksisterande three.js klasser, men det fungerte ikkje heilt med vårt oppsett. Lyd når ein skøyt og lyd på fuglen blei prøvd til slutt men me fekk det ikkje heilt til. Til slutt fekk me ikkje implementert noko annleis teksturs platting enn det som kom med startkoden og gjere terrenget meir «glatt».

Me føler målet vårt blei nådd. Me fekk til ein god del interaksjon og spel-logikk implementert med at ein kan bevege seg rundt, hoppe, skyte og bytte mellom ulike ting framføre kameraet. Men det var kanskje litt masse fokus på interaksjon og logikk enn meir avanserte grafiske metodar som ein kunne implementert med WebGL og three.js.

Gruppa har gjennom oppgåva lært ein del nytt om det å bruke three.js til å lage eit program som kan vere lærerikt å ta med seg til ei eventuell bacheloroppgåve om VR. Det me lærte mest var å få til interaksjon slik at det likna meir eit spel og ikkje berre noko som gjekk automatisk. Då spesielt med raycaster og bevegelse av kamera. Andre ting var å hente ut og få konvertert høgdedata til noko me kunne bruke i vårt spel. Me lærte også korleis me kunne importere modeller som me henta på nettet og korleis ein kunne spele av animasjonar som kom med desse modellane. Vidare lærte med litt om ulike typar lys og korleis ein kunne animere vatn.

5 Vidare arbeid

Om gruppa skulle ha jobba meir med oppgåva er det ein del ting me skulle ha fått til og gjort. Den største tingen er å optimalisere spelet då det køyrer dårleg på eldre og innebygde grafikkprosessorar i blant anna Intel-prosessorar, då alle på gruppa hadde kraftige dedikerte grafikkort på maskinane sine. Ting me kunne ha gjort var å bruke LOD for å gjere ting lengre vekke frå kameraet til enklare modellar som var lettare å teikne. Andre ting er å ikkje skyggelegge alt i scenen men berre nokon få ting, og ha ein lågare kvalitet på skuggen til objekt og refleksjonar i vatnet. Me ville også ha prøvd å lagt til lyd via three.js sine lyd-klassar. Ein stor ting me ville gjerne hatt var meir interaktivitet, som ein modell av ein person som gjer noko og ein kan interagere med denne modellen eller noko liknande. Betre terreng med korrekte teksturar ved bruk av meir teksturspLatting og andre ting. Fikse utplassering av trer med normalfordeling og også få inn gras som billboard og gjerne med krysning for at det ser meir realistisk ut. Til slutt kunne me ha fortsett på vatnet og innsjøen vår for å få dei meir realistiske, kanskje ved å sjå meir på three.js eksempla og finne meir avanserte shader-teknikkar.

Referanseliste

Referert til i teksten:

[1] https://threejs.org/docs/#examples/en/controls/PointerLockControls

[2] https://developer.mozilla.org/en-US/docs/Web/API/Pointer_Lock_API

[3] https://github.com/mrdoob/three.js/blob/master/examples/misc_controls_pointerlock.html

[4] https://threejs.org/docs/index.html#api/en/core/Clock

[5] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number

[6] https://threejs.org/docs/#api/en/core/Raycaster

[7] https://hoydedata.no/laserinnsyn/

[8] https://www.qgis.org/en/site/

[9] https://somethingaboutmaps.wordpress.com/blender-relief-tutorial-getting-set-up/

[10] https://gis.stackexchange.com/questions/332234/converting-grayscale-geotiff-to-png-using-qgis-results-in-all-black-image/363919

[11] https://threejs.org/docs/#api/en/geometries/PlaneBufferGeometry

[12] https://threejs.org/docs/#api/en/materials/MeshStandardMaterial

[13] https://threejs.org/docs/#api/en/materials/MeshPhongMaterial

[14] https://3dtextures.me/2017/12/28/water-001/

[15] https://en.wikipedia.org/wiki/Ambient_occlusion

[16] https://medium.com/@joshmarinacci/customizing-vertex-shaders-86527c5693b2

[17] https://medium.com/@joshmarinacci/water-ripples-with-vertex-shaders-6a9ecbdf091f

[18] https://github.com/mrdoob/three.js/blob/master/examples/webgl_shaders_ocean.html

[19] https://threejs.org/docs/index.html#api/en/cameras/CubeCamera

[20] https://threejs.org/docs/index.html#api/en/renderers/WebGLCubeRenderTarget

[21] https://sites.google.com/site/threejstuts/home/dynamic-cubemap

[22] https://sketchfab.com/3d-models/low-poly-bird-animated-82ada91f0ac64ab595fbc3dc994a3590

[23] https://threejs.org/docs/index.html#api/en/animation/AnimationMixer

[24] https://threejs.org/docs/index.html#api/en/animation/AnimationAction

[25] https://threejs.org/docs/index.html#api/en/extras/curves/CatmullRomCurve3

[26] https://stackoverflow.com/questions/11179327/orient-objects-rotation-to-a-spline-point-tangent-in-three-js

[27] https://3dtextures.me/2020/02/03/rock-039/

[28] https://threejs.org/docs/#api/en/materials/MeshStandardMaterial.displacementMap

[29] https://sketchfab.com/3d-models/james-bond-golden-gun-b9d719a0bf504059a4bbcff1b20c8874

[30] https://stackoverflow.com/questions/31220969/three-js-raycast-from-camera-center

[31] https://sketchfab.com/3d-models/flashlight-fc5a0e9799de4eda932f2714f63f8d0c

[32] https://threejs.org/docs/index.html#api/en/lights/SpotLight

[33] https://stackoverflow.com/questions/16456912/point-spotlight-in-same-direction-as-camera-three-js-flashlight

[34] https://threejs.org/docs/index.html#api/en/objects/Group

[35] https://threejs.org/docs/index.html#api/en/objects/InstancedMesh

[36] https://discourse.threejs.org/t/cubetextureloader-mapping-of-textures-to-skybox/13238

[37] https://www.curseforge.com/minecraft/texture-packs/clarity

[38] https://threejs.org/docs/#api/en/math/Matrix4.setPosition

[39] https://threejs.org/docs/index.html#examples/en/postprocessing/EffectComposer

[40] https://threejsfundamentals.org/threejs/lessons/threejs-post-processing.html

[41] https://github.com/mrdoob/three.js/blob/master/examples/jsm/postprocessing/BokehPass.js

[42] https://github.com/mrdoob/three.js/blob/master/examples/jsm/postprocessing/FilmPass.js

[43] https://developer.mozilla.org/en-US/docs/Web/API/OffscreenCanvas

[44] https://www.w3schools.com/tags/ref_canvas.asp

[45] https://www.evermade.fi/story/pure-three-js-hud/

[46] https://github.com/mrdoob/stats.js/

[47] https://threejs.org/docs/index.html#api/en/scenes/FogExp2

[48] https://github.com/hoverinc/ray-tracing-renderer

[49] https://github.com/mrdoob/three.js/issues/225

Andre som er blitt brukt men ikkje referert til i teksten:

https://threejs.org/docs/#examples/en/loaders/GLTFLoader

https://discourse.threejs.org/t/easiest-way-to-play-skeletal-animation-from-gltf/7792

https://www.hiclipart.com/free-transparent-background-png-clipart-ngwzp

https://www.hiclipart.com/free-transparent-background-png-clipart-iuuxx

https://threejs.org/examples/#misc_controls_pointerlock

https://github.com/mrdoob/three.js/blob/master/examples/webgl_materials_cubemap_dynamic.html

https://medium.com/@soffritti.pierfrancesco/dynamic-reflections-in-three-js-2d46f3378fc4

https://threejsfundamentals.org/threejs/lessons/threejs-fog.html

https://stackoverflow.com/questions/12667507/drawing-ui-elements-directly-to-the-webgl-area-with-three-js

https://codepen.io/jaamo/pen/MaOGZV

https://stackoverflow.com/questions/38371768/three-js-cubemap-and-renderer-autoclear-false

https://threejs.org/docs/#api/en/lights/shadows/SpotLightShadow

https://threejs.org/examples/#webgl_lights_spotlight

https://threejs.org/docs/#api/en/textures/CubeTexture

https://threejs.org/docs/#api/en/loaders/CubeTextureLoader

https://stackoverflow.com/questions/47023151/how-to-use-a-three-js-curve-as-a-path-for-moving-a-mesh-along-at

http://jsfiddle.net/SCXNQ/891/

https://threejs.org/docs/#api/en/audio/Audio

https://threejs.org/docs/#api/en/audio/PositionalAudio

https://threejs.org/docs/#api/en/audio/AudioListener

https://github.com/mrdoob/three.js/tree/master/examples/jsm/postprocessing

https://threejs.org/docs/#api/en/objects/Sprite

https://sketchfab.com/3d-models/house-test-8ccdacecef714a4bb1e7eaa7075695c7

https://threejs.org/docs/#api/en/extras/curves/EllipseCurve

https://threejs.org/docs/index.html#api/en/lights/DirectionalLight

https://threejs.org/docs/index.html#api/en/lights/shadows/DirectionalLightShadow

https://threejs.org/docs/index.html#api/en/lights/shadows/LightShadow

https://www.giantbomb.com/angry-sun/3015-6840/images/