Et bilde som inneholder sort, mørke

Automatisk generert beskrivelse

Fakultet for Audiovisuelle Medier og Kreativ Teknologi

**827**

**Mappeoppgave**

**Rapport**

Visualisering og simulering

2VSIM101

**2023**

1. **Introduksjon**

Jeg skal skrive kode som plasserer punkter i en punktsky rundt om i en 3D-scene. En annen del med kode jeg skal skrive skal konstruere et kvadratisk plan via regulær triangulering, som en eller flere kuler skal rulle over. Kulene skal kunne kollidere med hverandre. Siden valg av program eller spillmotor var valgfritt gikk jeg for spillmotoren Unity som bruker programmeringsspråket C#. Annet som er vert å vite om Unity er at som bruker et venstrehendt koordinatsystem, dvs. at Y som peker oppover og Z peker inn og ut av skjermen. Rekkefølge for indeksering av vertices og dermed triangel naboer må legges inn i rekkefølge som går med klokken, siden det er rekkefølgen Unity leser av indeksering for vertices. Kommersielle spillmotorer slik som Unity eller Unreal har ofte allerede implementert fysikk. Oppgaven baserer seg på at man ikke benytter seg av de allerede innebygde fysikk systemene, slik som de ulike *collider* klassene og *rigidbody*.

1. **Metode**

Jeg begynner med å få tak i punktskydataen jeg skal bruke for å rendere punktskyen og definere høyden til den genererte flaten. Her begynner jeg først med å laste ned jeg ned en LAS-fil fra <http://hoydedata.no>. LAS filer brukes til å representere geodata som en punktsky. Jeg bruker så programmet *Laszip* for å konvertere LAS filen om til en .txt fil, slik som er bedt om i oppgaveteksten. *Laszip* skriver ut hvert punkt på sin egen linje med XYZ-koordinater. Jeg lager en klasse *Punktsky* som leser gjennom .txt filen og lagrer alle punktdataene i en Vector3 array *points*. Siden punktene sine koordinater er veldig lagt unna origo, lagrer jeg også de største og minste verdiene for X, Y, og Z i respektive float variabler *xMin*, *xMax*, osv. Ved å ta gjennomsnitt av *Min* og *Max* verdiene for XYZ får jeg sentrert alle punktene rundt origo i scenen. Siden jeg bruker Unity er det også nødvendig å benytte seg av enten *Direct GPU Instansing* eller *Indirect GPU Instansing* for at programmet skal ha en brukbar bildefrekvens når jeg visualiserer punktene. I mitt tilfelle valgte jeg å bruke *Indirect GPU Instancing* siden det gir noe høyere bildefrekvens (Spencer, 2023). Disse punktene skal brukes til å bestemme høyden for flaten jeg konstruerer.

I en ny klasse skal jeg konstruere en kvadratisk flate satt sammen av mindre kvadrater, som igjen er satt sammen av triangler. For å kunne sette sammen flaten må jeg lagre noen forskjellige typer data. Dataene som lagres for dette er vertex-data, indeks-data for hver vertex, triangel nabodata (Nylund, 2023, s. 81) og uv-koordinater hvis man skal ha teksturer på flaten. Størrelsen på planet i XZ-retning bestemmer jeg ved å benytte den større av *xMax* eller *zMax* til en ny variabel *size*. Ved å dividere *size* på en annen variabel *resolution* får jeg et nytt tall som gir meg skrittlengen mellom hver vertex. *Resolution* bestemmer også hvor mange kvadrater i XZ-retning, som skal bygge opp flaten, altså oppløsningen til flaten. Høyden til hver vertex bestemmes ved at jeg sender inn hver vertex sin posisjon på XZ-planet til en funksjon *checkForPoint* og definerer et lite kvadratisk område rundt verticen. Jeg bruker så barysentriske koordinater for å sjekke om det eksisterer punkter innenfor det kvadratiske området, dersom det eksisterer punkter innenfor kvadratet lagrer jeg Y-koordinaten til punktet i et array og tar gjennomsnittet av de lagrede verdiene på slutten av funksjonen og setter verticen sin høyde til gjennomsnittet. Siden punktskyene kan inneholde flere millioner av punkter har jeg implementert variabelen *skip* som bestemmer antallet punkter som hoppes over i hver iterasjon av løkken som sjekker posisjonen punktene. Dette er for å unngå å vente veldig lenge på at CPU-en skal lese gjennom alle punktene når en skal teste fysikk simulasjonen.

Kulene som skal rulle/skli over flaten lages i en egen klasse/C#-fil *RollingBall* som har en referanse til en triangelflate klasse. For hvert nytt bilde som vises kalkuleres akselerasjonen og farten til kulen, dersom den er i kontakt med en flate kalkuleres også ut normalen til triangelet kulen er i kontakt med og projiserer fart vektoren mot triangelets normalvektor. For å sjekke om kulen er i kontakt med en flate må jeg først ha posisjonen til kula og bruke barysentriske koordinater for å finne ut av hvilket triangel kula kan være i kontakt med. Her utnytter jeg regulariteten av kvadratene som bygger opp flaten, og bruker kulas posisjon i XZ-retning og til å først finne hvilket kvadrat kulen er innenfor også tar de barysentriske koordinatene til kvadratets oddetall triangel (Åsbø, 2023). Om kula ikke er innenfor triangelet i XZ-retning brukes det barysentriske koordinaten som er mest negativt til å velge et nabotriangel i retning av kulas posisjon. Når jeg til slutt finner triangelet kula er innenfor kan det barysentriske koordinaten oversettes til verdens koordinater som kan benyttes til å kalkulere kontaktpunktet. De barysentriske koordinatene gir oss kontaktpunktet mellom kula og flaten, dersom distansen mellom kula og kontaktpunktet er mindre enn radiusen til kula i kontakt med flaten. Fart vertoren til en kule som er i kontakt med flaten projiseres langs flaten (Nylund, 2023, 113).

For at kulene skal kunne kollidere med hverandre må de ha en referanse til de andre kulene. Jeg lagrer alle kulene i et array *balls* i en ny klasse *BallManager*, og bruker to for-løkker på arrayet til å sjekke om to kuler er nære nok til å kollidere. Om distansen er mindre enn radiusen til de to kulene lagt sammen gjennomføres kollisjonsresponsen. *BallManager* klassen står for å plassere nye kuler på tilfeldige posisjoner over flaten, hver gang den plasserer en ny kule legges den til i *balls* arrayet. Her sjekker den om distansen mellom de to kulene er større eller mindre enn radiusen til begge kulene lagt sammen. Brukeren kan holde inne mellomromsknappen for på plassere en ny kule på et tilfeldig punkt hvert halve sekund.

1. **Resultat**

Jeg plasserer omtrent 960 000 punkter i scenen og representerer dem med kuber. Prosessen i å laste inn så mange punkter og deretter tegne dem hadde vært svært tungt for datamaskinen om jeg hadde gjort dette via CPU-en. Siden jeg istedenfor tok i bruk Unitys *Indirect GPU Instancing* funksjonalitet blir prosessen i å tegne punktene lastet over på GPU-en. Med *Indirect GPU Instancing* får jeg en gjennomsnittlig bildefrekvens på cirka 90 bilder i sekundet.

Et bilde som inneholder himmel, utendørs, vulkan, grunn

Automatisk generert beskrivelse

*Figur x: Punkskyen*

Flaten blir konstruert til et uniformt kvadrat satt sammen av mindre kvadrater (quads) som igjen er satt sammen av to triangler. Variabelen *resolution* som brukeren kan sette i motoren bestemmer antallet quads i XZ-retning. Størrelsen til flaten bestemmes av punktsky punktet som er lengst unna origo i XZ-retning, dette kan gi ganske store flater da jeg har opplevd at noen av punktsky dataene kan være ganske store. Høyden settes riktig, men denne prosessen kan bruke veldig lang CPU-tid dersom *skip* variabelen er lav. Med *skip* og *resolution* satt til 50 er ikke CPU-tiden mye lenger enn 8 sekunder. Siden jeg har regnet ut UV-koordinatene til flaten er det også mulig å legge til og vise teksturer på den.

Et bilde som inneholder tegnefilm, tegning, maling, kunst

Automatisk generert beskrivelseEt bilde som inneholder tegnefilm, Animasjon, Fiktiv karakter, skjermbilde

Automatisk generert beskrivelse

*Figur x: flaten sett fra oven med wireframe* *Figur x: Samme som høyre, men uten wireframe*

Kulene ruller/sklir over flaten. Om jeg søker gjennom alle trianglene for å finne hvilke triangel ballen kolliderer med resulterer lavere bildefrekvens når det er flere kuler eller antallet triangler i flaten er stort. Bildefrekvensen blir bedre når triangel søket utnytter regulariteten av flaten til å finne hvilken trekant kula er på og deretter søker i nabotrekanter dersom det første søket ikke er riktig.

Å sjekke for kollisjon mellom kule til kule funker som forventet. Kollisjonsresponsen resulterer i at kulene dytter jeg fra hverandre slik at de aldri overlapper (OneLoneCoder, 2018), men denne funksjonaliteten gjør også at kulene har en tendens til å dytte hverandre gjennom flaten når de kolliderer. I tillegg oppleves fysikken for å endre fart vektoren uforutsigbar.

1. **Diskusjon**

Fysikken bak kollisjonsresponsen er ikke riktig,

1. **Referanser**

Nagy, L. N

Åsbø, A. P. (2023).

Andreas

Nylund, D. A. (2023). *MAT301 Matematikk III VSIM101 Visualisering og simulering forelesningsnotater og oppgaver*.

Spencer, M. J. [Tarodev] (2023, Mars 26.) *How To Render 2 Million Objects At 120 FPS* [Video] Youtube. <https://www.youtube.com/watch?v=6mNj3M1il_c>

OneLoneCoder [javidx9] (2018, Januar 21.) *Programming Balls #1 Circle Vs Circle Collisions C++* [Video] Youtube. <https://www.youtube.com/watch?v=LPzyNOHY3A4>

Özcan, A. B. [NesquikPlus] (2021). <https://github.com/NesquikPlus/opengl_collision/blob/master/Game.cpp>