Mappeoppgave H2023 VisSim

Stål Brændeland

Introduksjon

For dette prosjektet valgte jeg å jobbe i Unity. Det gjør noen ting som innlesing av punkter og visualisering av de samme punktene som punktsky noe mer involvert, men flere andre ting er mye bedre tilrettelagt - i tilegg er det da mulighet for å jobbe i Rider fremfor Qt Creator.

Metode og resultater

2 Konstruksjon og visualisering av 3D-terreng

2.1

Her valgte jeg et vilkårlig areal i nordmarka på hoydedata.no, som ble tilsendt meg på mail som en .laz fil. Deretter konverterte jeg det til enkelt innlesbar xyz data med las2txt, et redskap fra LAStools-samlingen (`https://github.com/LAStools/LAStools`). Filen kan åpnes direkte i fil-utforsker fra prosjekt-mappen (Assets/StreamingAssets/terrain.txt) eller via Unity-prosjektet.

2.2

Referer til Unity-prosjektet for å se det endelig meshet. For å få ok performance (over 20FPS) når jeg skulle tegne mange millioner kuber var det eneste som ga bra nok resultater indirekte GPU-instansiering. (Jeg valgte en kube til å representere ett punkt - her hadde noe som GL_Points i openGL vært nyttig.) For å få indirekte GPU-instansiering til å funke brukte jeg en youtube-video fra Tarodev[4] og tilsvarende GitHub-prosjekt[3] som referanse.

2.3

Siden jeg ikke trenger flere millioner punkter for å lage en triangulering leser jeg inn 100.000 av punktene fra LiDAR-dataen til er array. Maks antall vertekser du kan ha i et enkelt mesh i Unity er 65355 (16 bit buffer), så jeg satt en oppløsning o slik at $o^2 < 65355$. I dette tilfellet satte jeg den til 60. Da vår vi et grid for triangulering av o^2 vertekser, som også gir oss $(o-1)^2$ firkanter i gridet - det betyr at til slutt får vi $2(o-1)^2$ trekanter. Prosessen følger tett den beskrevet i 10.6 i forelesningsnotatene[2], men med xMin,zMax til xMax,zMin i stedet for yMax og yMin, siden y er opp i Unity. Z er også gitt som opp i høydedataen, så i koden kan du se at jeg bytter om z og y-koordinatene. Jeg justerer

også alle koordinat-verdiene ned (i praksis bare å flytte meshet nærmere origo) så ikke terrenget skal ende opp så fryktelig langt unna. Jeg bruker en klasse, Triangle, for å holde oversikt over nabo-trekanter. Tekstfil med indekserings-data og nabodata kan du finne i prosjekt-mappen (Assets/StreamingAssets/triangles.txt), og interaktivt i Unity-prosjektet. Den regeneres hver gang du trykker på Generer tekstfil med triangulering", med den samme dataen som meshet i **2.4** tegnes med.
For å slippe at den skulle regenereres hver gang man åpner menyen, men fortsatt sikre at TriangleSurface-klassen er ferdig med å generere dataen på sin side når man trykker på knappen brukte jeg "coroutines".

```
public void GetTriangleInfo(Action<int[]> callback)
{
    StartCoroutine( routine: FetchTriangleInfo(callback));
}

 Frequently called   1 usage   Stål Brændeland
private IEnumerator FetchTriangleInfo(Action<int[]> callback)
{
    yield return new WaitUntil(() => _trianglesGenerated);

    var output :int[] = GenerateOutput();

    callback?.Invoke(output);
}
```

Fig. 1: *GenerateOutput()* lager trianguleringsdataen, men det antar at Triangles (en List<Triangle>) er populært med riktig info, som vi ikke kan vite om den er når denne funksjonen kalles, siden *GetTriangleInfo* kalles fra en annen klasse - i dette tilfellet MainMenu. Her kan vi bruke en coroutine for å vente på et bool flagg som lar oss vite at trianglene er generert og vi kan genere trianguleringsinfoen på tekst-format.

2.4

Referer til Unity-prosjektet.

3 Simulering av nedbør og vassdrag

3.1

Newtons andre lov sier at når et legeme blir påvirket av én eller flere krefter, vil det få en akselerasjon i den retningen kreftene virker. Summen av kreftene på legemet er lik legemets

masse ganger dets akselerasjon. Denne loven kan beskrives med formelen

$$F = ma$$

Massen til ballen er 1. Tyngdekraften er representert av Unity sin Physics.Gravity, altså vektoren

$$\vec{g} = [0, -9.81, 0]$$

Physics.Gravity har enheten m/s^2, så vi ganger med massen for å få en ny vektor som har enheten N

$$\vec{G} = \vec{g} \times 1$$

Ballen bruker så i hvert tidssteg barysentriske koordinater for å finn ut hvilken trekant den befinner seg på. Når vi vet hvilken trekant det dreier seg om kan Triangle-klassen finne enhetsnormalen sin ved å ta kryssproduktet av to av kantene sine og normalisere det. Hvis en trekant består av vertekser a, b, c

$$\vec{ab} = b - a$$

$$\vec{ac} = c - a$$

$$\vec{n} = ab \times ac$$

Vi kaller $\vec{n}$ normalisert for $\hat{n}$. Normalkraften blir da minus en ganger prikkproduktet av denne vektoren og vektoren for tyngdekraften. Vi ganger den med enhetsnormalen for å gi den retning og representere den som en vektor.

$$\vec{N} = -(\hat{n} \cdot \vec{G}) \times \hat{n}$$

Den endelige kraften som virker på ballen, x-komponenten til tyngdekraften langs trekant-planet kan vi finne ved å legge sammen de to vektorene.

$$\vec{F} = \vec{G} + \vec{N}$$

Ved å bruke formelen fra Newtons andre lov kan vi finne akselerasjonen

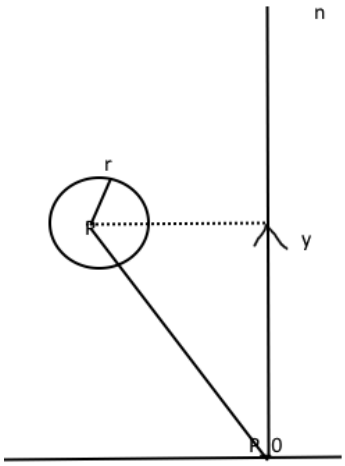
$$a = \frac{F}{m}$$

Med akselerasjonen er det trivielt å finne fart og deretter posisjon (bare å gange med tid) - slik kan vi simulere ballens posisjon ettersom den beveger seg i terrenget[2].

```
var surfaceNormal:Vector3 = -Vector3.Dot(lhs:Gravity, rhs:unitNormal) * unitNormal;
var force:Vector3 = Gravity + surfaceNormal;
Vector3 acceleration;
if (_rolling)
    acceleration = force / Mass;
else
    acceleration = Gravity / Mass;

var velocity:Vector3 = _oldVelocity + acceleration * Time.fixedDeltaTime;
```

Fig. 2: Implementasjonen av fysikken i RollingBall-klassen. _*rolling* er her en bool som er sann hvis ballen er i kontakt med bakken.



(a) Konsept for kollisjon med bakken.

```
unitNormal = triangles[_triangleID].Normal;
Vector3 point = triangles[_triangleID].Vertices[0];
var p:Vector3 = transform.position - point;
var y:Vector3 = Vector3.Dot(lhs:p, rhs:unitNormal) * unitNormal;
_rolling = y.magnitude < _radius;
```

(b) Implementasjon i koden.

Fig. 3: Jeg setter _*rolling*==true hvis lengden på $\vec{y} < r$.

3.2

Regndråpene bruker akkurat samme logikk basert på samme fysiske lover som ballene til å bevege seg over terrenget. Det er samme klasse (RollingBall), bare med en boolean som bestemmer annen logikk i forhold til vassdrag og ekstrem-vær. Nedbørs-klassen (RainManager) genererer nå bare 180 regndråper før regnet skrur seg av, men det er fortsatt mulig for brukeren å interaktivt skru på nedbøren igjen, og å stoppe og starte det når som helst. Det vil alltid stoppe seg selv hvis det eksisterer for mange regndråper på banen for øyeblikket, for å holde performance under kontroll. I tidligere versjoner ble regnet også generert over hele terrenget, men for å etterligne virkeligheten (For i virkeligheten blir fuktig luft som presses over fjell nedkjølt, og dannes deretter til nedbør[1]) ble området begerenset til et mindre areal over et et høydepunkt i terrenget. Dette hjelper også og tydeliggjøre vassdragene og retningene deres.

3.3

Når regndråpene har mistet så og si all fart og satt seg i ro danner de et lite vann som kan gro ettersom nye regndråper faller på samme sted. Mens de ruller lagrer de kontrollpunkter i xz-planet hvert 200. tidssteg. En spline konstrueres med disse kontrollpunktene og visualiseres med en Line Renderer. For hvert punkt vi konstruerer i kurven (Line Render-klassen) interpolerer vi for xz-posisjon og bruker så barysentriske koordinater til å finne y-verdien til terrenget der kontrollpunktet ligger, slik at vi kan interpolere høyden og løfte kurven opp på overflaten.

```
// Calculate the positions
// the t starts from 2 and ends at controlPoints.Count
for (float t = 2; t <= controlPoints.Count; t += step)
{
    var position:Vector3 = Vector3.zero;
    for (int i = 0; i < controlPoints.Count; i++)
    {
        float basis = BSplineBasis(i, degree:2, t, knotVector);
        position += basis * new Vector3(controlPoints[i].Value.x,
            y:0,
            z:controlPoints[i].Value.y);
        // get the adjusted y-value for each point in the line renderer according to terrain
        position += basis * new Vector3(x:0,
            y:_triangleSurface.Triangles[controlPoints[i].Key].HeightAtPoint(controlPoints[i].Value) + 0.1f,
            z:0);
    }

    positions.Add(position);
}

_lineRenderer.positionCount = positions.Count;
_lineRenderer.SetPositions(positions.ToArray());
```

Fig. 4: Utklipp fra Spline-klassen hvor vi regner ut punktene til kurven. controlpoints er en liste med attributt–verdi-par hvor attributtet er ID-en til trekanten kontrollpunktet befinner seg i slik at vi kan lett regne ut høyde med barysentriske koordinater uten å trenge å søke gjennom alle trekantene. Verdien er en Vector2 med xz-koordinatene til punktet.

Knutevektoren er ikke "clampeddet vil si at vi får en åpen spline som ikke går gjennom endepunktene.

```
private static float BSplineBasis(int i, int degree, float t, IReadOnlyList<float> knots)
{
    if (degree == 0)
    {
        if (knots[i] <= t && t < knots[i+1]) return 1.0f;

        return 0f;
    }

    var a:float = (t - knots[i]) / (knots[i + degree] - knots[i])
            * BSplineBasis(i, degree:degree - 1, t, knots);
    var b:float = (knots[i + degree + 1] - t) / (knots[i + degree + 1] - knots[i + 1])
            * BSplineBasis(i:i + 1, degree:degree - 1, t, knots);
    return a + b;

}
```

Fig. 5: Basis-funksjonen.

3.4

Baller som har lagt seg til ro i terrenget vil flyte oppå vannet som dannes, slik at hvis det blir nok vann vil de rulle avgårde til lavere terreng.

```
// Ekstrem-vær effekt
else if (WithinTriangles(_balls[i].gameObject.transform.position))
{
    // print("Floating ball" + ball.gameObject.name);

    // push away normal balls (non-raindrops)
    var height:float = _vertices[0].y;
    // to world space
    height += transform.position.y;
    ball.DoFloat(height);
}
```

Fig. 6: Utklipp fra WaterBody-klassen (en regndråpe som har lagt seg i ro transformerer seg til en WaterBody).

```
public void DoFloat(float worldHeight)
{
    _floating = true;
    _height = worldHeight;
}
```

Fig. 7: Utklipp fra RollingBall-klassen. *height*-variabelen bestemmer høyden i verdenen ballen korrigerer til hvert tidssteg.

Diskusjon

Det var flere oppgaver med noe til en del rom for tolkning, men slik jeg forsto det fra oppgaveteksten så var det meningen. En relativt stor del av kodeprosjektet er WaterBody.cs - en klasse som ikke nødvendigvis var spesifisert av noen av oppgavene, og sikkert kommer til å være forskjellig fra mange andre sine løsninger. Denne klassen er min tolkning av oppgave 3.4. Det er en del den gjør som ikke er strengt tatt nødvendig for å løse den oppgaven som er gitt. Selve effekten av å dytte ballen opp på overflaten er relativt simpel, det mer involverte er hvordan vannet gror ettersom det samler nye regndråper. Det sjekker posisjonen på hjørnene sine i forhold til trekantflater i området, og bruker barysentriske koordinater for å prøve å

bare gro slik at ikke hjørnene på planet (fordi den representeres av et enkelt plan) ikke skal stikke ut (komme til syne ovenifra) for å opprettholde illusjonen av en vannmasse. Andre småting å nevne er at tekstfilene for både punktdata og triaungleringsdata burde gå an å åpne på macOS og Linux automatisk via menyen, gjerne prøv hvis du har en tilgjengelig pc.

Referanser

[1] Sigbjørn Grønås. *Hvorfor regner det så mye i Bergen*. Universitetet i Bergen. 5. jan. 2009. URL: `https://www.uib.no/gfi/56974/hvorfor-regner-det-s%C3%A5-mye-i-bergen` (sjekket 24.11.2023).

[2] Dag Nylund. *MAT301 Matematikk III VSIM101 Visualisering og simulering forelesningsnotater og oppgaver*. Google Docs. 5. okt. 2023. URL: `https://drive.google.com/file/d/1cKrJ_vrKrcS1igT96_c09cjgJrRo442p/edit?usp=embed_facebook` (sjekket 24.11.2023).

[3] Matthew Spencer. *Pushing Unity's rendering capabilities to the max!* original-date: 2023-03-21T03:22:15Z. 23. nov. 2023. URL: `https://github.com/Matthew-J-Spencer/pushing-unity` (sjekket 24.11.2023).

[4] Tarodev. *How To Render 2 Million Objects At 120 FPS*. 26. mar. 2023. URL: `https://www.youtube.com/watch?v=6mNj3M1il_c` (sjekket 24.11.2023).