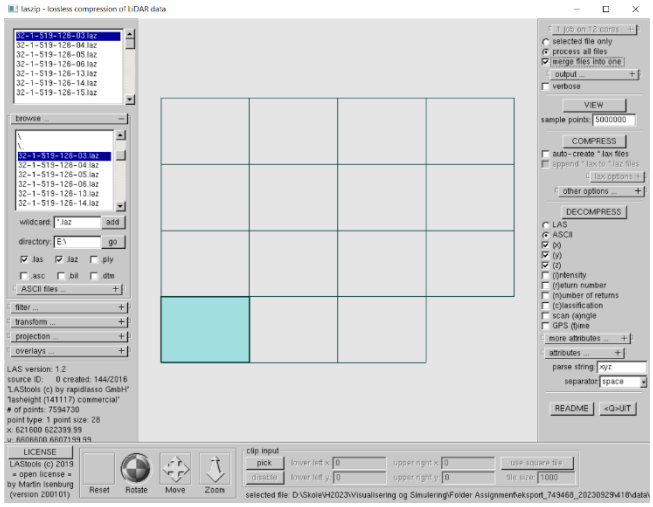
Oppgave 2.1:

«Last ned terrengdata fra Statens Kartverk for et selvvalgt område med noe kupert terreng, og valgfritt filformat SOSI eller LAS (Les om filformatene LAS og SOSI). Konverter høydedata fra valgt fil til eget format (x y z linjevis). Lag en tekstfil med ett punkt på hver linje (xyz koordinater adskilt slik at de enkelt kan leses inn). Øverst i tekstfila skal antall punkter stå på egen linje. Bruk gjerne tilgjengelig programvare til dette.»

- Området som er lastet ned fra «Statens Kartverk» er et utklipp fra Askim i gamle Østfold kommune, hvor jeg vokste opp.
- Filen ble mottatt som LAS-filformat via mail og ved bruk av programmet «Laszip.exe» har jeg fått konvertert punktskyen fra LAS til et lesbart «xyz»-format.
- Den konverterte tekstfilen består av koordinater på punktsky-form, der hvert koordinat får sin egen linje fordelt over 3 punkter (xyz-koordinater for punktet).
- For å få tak i øverste linje, hvor tallet over mengden linjer i dokumentet skal stå, brukte jeg «Notepad++» og la til linjetallet basert på ninjenummeret på siste linje i dokumentet. Jeg så på dette som den mest effektive måten å få inn linjetallet for videre bruk.

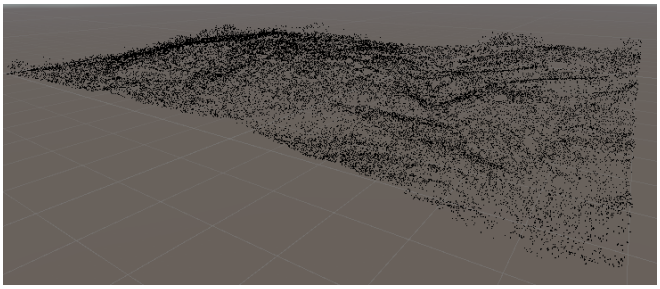



```
37672333
623200.00 6608009.00 151.20
623200.04 6608008.51 151.19
623200.07 6608008.03 151.21
623200.11 6608007.54 151.20
623200.15 6608007.05 151.19
623200.18 6608006.56 151.17
623200.21 6608006.08 151.22
623200.25 6608005.60 151.22
623200.29 6608005.11 151.21
623200.33 6608004.61 151.17
623200.36 6608004.14 151.20
623200.39 6608003.67 151.22
623200.42 6608003.20 151.22
623200.46 6608002.72 151.22
623200.49 6608002.24 151.23
623200.53 6608001.75 151.23
623200.57 6608001.26 151.21
623200.60 6608000.79 151.24
623200.63 6608000.31 151.24
623200.89 6608000.29 151.24
623200.86 6608000.74 151.23
623200.83 6608001.19 151.22
623200.80 6608001.65 151.23
623200.77 6608002.10 151.22
623200.73 6608002.54 151.20
623200.71 6608002.99 151.17
```

Oppgave 2.2:

«Tilpass koordinatsystemet i rendrevinduet og kameraposisjonen til datagrunnlaget. Test at de innleste punktene vises ved å tegne dem som punkter/punktsky.»

- Jeg benytter en selvlaget «TekstInnlesnings»-funksjon for å splitte opp linjene fra punktsky-filen, hente dataen, og legge «x, y, z»-koordinatene til i en ny Vector3-Liste i Unity.
- Fra denne listen er det en smal sak å visualisere punktene ved bruk av en innebygd «`OnDrawGizmos»-klasse med sin egne DrawCube()-funksjon`.
- Ettersom punktsky-filen består av enorme mengder linjer, velger jeg å hoppe over 100 punkter for hvert punkt jeg viser, slik at programmet ikke overgår 38 millioner punkter, som er det maksimalt antall Gizmos som kan vises på en gang i Unity. Bildet over skal likevel vise at punktene er lest inn korrekt.
- Det skal sies at terrenget i Askim var mindre kupert enn jeg hadde sett for meg, så jeg har valgt å legge inn en multiplier i koden når meshen scales, som ganger y-verdien i alle punkter med 2, slik at terrenget blir noe mer kupert. For å kompensere for dette, har jeg valgt å skyve hele terrenget ned 300 units. I tilfeller med mer kupert terreng ville y-verdien blitt beholdt uendret i scalingen av meshen, da høyden varierer svert lite ift. x-/ og z-verdiene til terrenget.



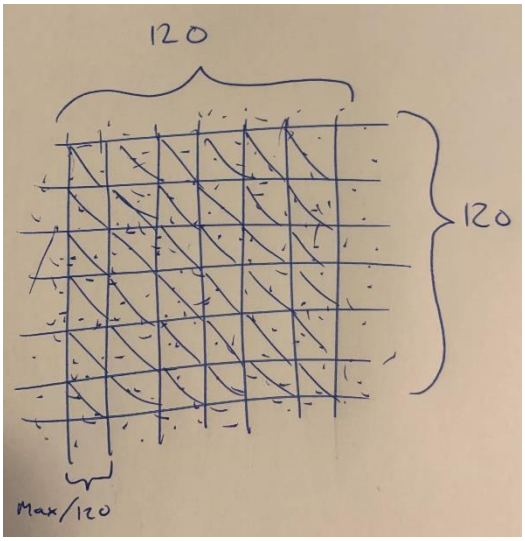
```
Unity Message | 0 references
void OnDrawGizmos()
{
    if (!showGizmo)
    {
        return;
    }

    //Draw squares on each "vertices_PointCloud[i]"-point
    for (var i = 0; i < vertices_PointCloud.Count; i += 3)
    {
        //Color
        Gizmos.color = gizmoColor;

        //Cube
        Gizmos.DrawCube
            (
                new Vector3
                (
                    vertices_PointCloud[i].x,
                    vertices_PointCloud[i].y,
                    vertices_PointCloud[i].z
                ),

                new Vector3(gizmoSize, gizmoSize, gizmoSize)
            );
    }
}
```

```
//Correct Point Position based on new Min/Max, and set new height
for (int i = 0; i < vertices_PointCloud.Count; i++)
{
    vertices_PointCloud[i] = new Vector3
        (
            vertices_PointCloud[i].x - minValue.x,
            vertices_PointCloud[i].y * 2 - 300,
            vertices_PointCloud[i].z - minValue.z
        );
}
```

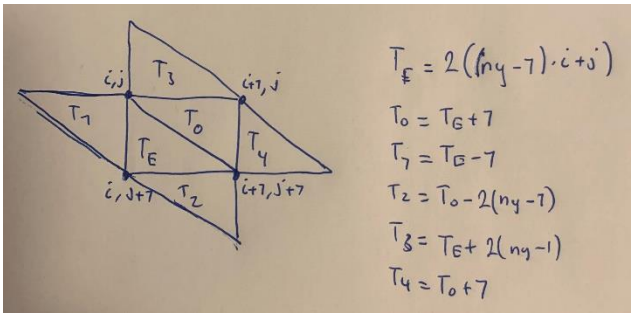
Oppgave 2.3:

«Lag en regulær triangulering for datasettet, som gjennomgått i kapittel 10.6 i forelesningsnotater. Angi nabotrekanter som i kapittel 6.3.8. Indeksering av vertekser/trekanter kan gjøres ved å bruke samme ide som i 2.5-2.6 (figur 2.1).»

- For mitt regulære grid har jeg valgt å benytte en størrelse på 120. Dvs. at jeg har delt terrenget som punktskyen består av, inn i 120 x 120 like store firkanter, for å sitte igjen med et 2D grid på 14 400 punkter for punktskyen på totalt 37 672 333 punkter.
    - Høyden til hver firkant bestemmes ut fra et gjennomsnitt over høyden til alle punktene fra punktskyen som holdes innenfor firkantens koordinater.



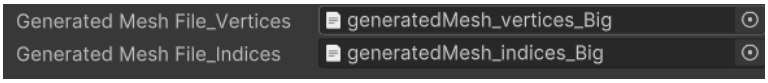
- Videre har jeg delt opp hver firkant inn i 2 trekanter. Nabotrekantene vil derfor utgjøre 2 andre trekanter for hver av mine 2 trekanter, som står utenfor firkanten som fokuseres på, i tillegg til hverandre.
- Fordelen med å utføre en regulær triangulering av et terreng, er at det blir veldig lett å referere til hvor på meshen et objekt er, ettersom vi ser etter firkantene via indekser og kan utføre utregning av barysentriske koordinater på hver av de to trekantene i firkanten, etter tur. Dette sparer oss for å gå gjennom alle punktene for meshen i hvert frame.
- Selve trianguleringen gjennomføres i «List<Vector3> GetVertices(List<Vector3> pos, int size)»-funksjonen i «PointCloudVisualize»-scriptet mitt.
- Indices lagres i funksjonen «List<int> GetIndices(int size)» og er kort nok til å ta med som kjermbilde her. Det er viktig å få med hele firkanten, så derfor legges det til for begge trekantene som firkanten består av.

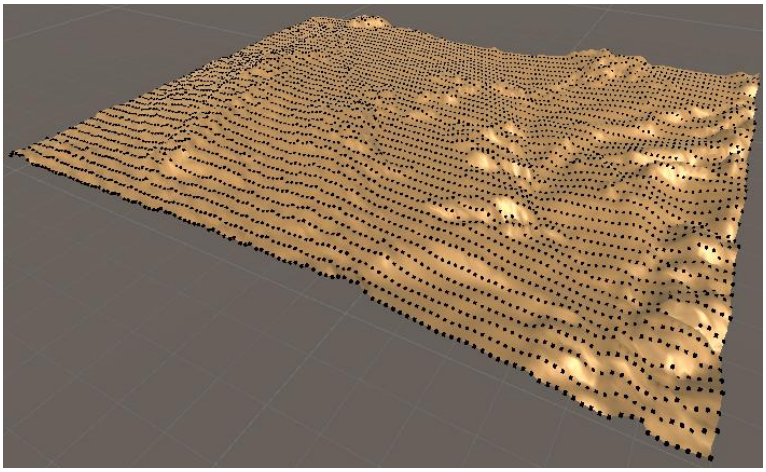



```
List<int> GetIndices(int size)
{
    List<int> indexList = new List<int>();

    //Add the indexes of the indices for both triangles in a square
    for (int x = 0; x < size - 1; x++)
    {
        for (int y = 0; y < size - 1; y++)
        {
            indexList.Add(x + (y * size));
            indexList.Add(x + (y * size) + 1);
            indexList.Add(x + (y * size) + size);

            indexList.Add(x + (y * size) + 1);
            indexList.Add(x + (y * size) + size + 1);
            indexList.Add(x + (y * size) + size);
        }
    }

    return indexList;
}
```

Oppgave 2.4:

«Visualiser terrenget. Velg en passende kameraposisjon og -orientering.»





- For å effektivt kunne visualisere terrenget, har jeg valgt å lage en funksjon som lager to nye .txt-filer; en for vertices og en for indices av det regulære terrenget jeg lagde i 2.3.
- Dette skaper to .txt-filer som til sammen er mange ganger mindre i størrelse enn punktsky-filen. Dette reduserer tiden det tar å visualisere terrenget i editoren med over 99%. Når jeg har et regulært datasett over både vertices og indices av et mesh, er det bare å lese inn dataene fra fil på samme måte som i 2.2, og putte Listene rett inn i Unity's tomme MeshObject for å fylle det med de riktige dataene.
- Videre trenger jeg kun å synliggjøre meshen i editoren for at den skal vise alle trianglene den består av.
- Her har jeg valgt å beholde Gizmosene på for å vise at terrenget nå er regulært i editoren.

Oppgave 3.1:

«Lag en simulering av en ball (eller flere) som beveger seg på dette terrenget (avsnitt 2.4) i henhold til Newtons 2. lov. Det skal være mulig å velge startpunkt for en ball interaktivt. Velg et startpunkt slik at ballen legger seg i ro i en forsenkning eller lavt punkt i terrenget.»

- Newtons 2. lov sier at akselerasjonen virker i retning av kreftene, med summen av kreftene som masse * akselerasjon (F = ma), så da må disse regnes ut for å finne ballens retning, i hvert frame.
- I hvert BallObject har jeg et script som bestemmer ballens bevegelse. Kreftene som viker inn på ballen er *KraftVektoren*, *Normalvektoren* og *Gravitasjonsvektoren*. Jeg har sett bort fra friksjon.

```
//Set physics vectors
Vector3 force_Vector = new Vector3(0, -9.81f * mass, 0);
Vector3 normal_Vector = new Vector3();
Vector3 gravity_Vector = mass * gravity;
```

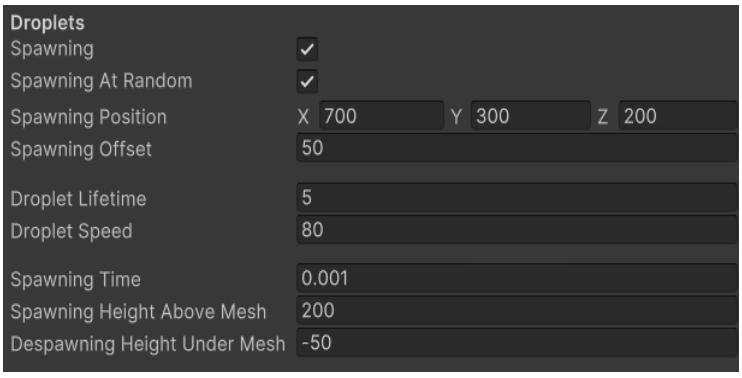
- Dersom ballen ikke kolliderer med terrenget, skal ballen falle i takt med gravitasjonen.
- For å finne ut om ballen kolliderer med terrenget, bruker vi mesh-indexen som utgangspunkt til å finne ballens posisjon. Dette for raskt å sirkles inn på riktig firkant å gjennomføre kalkulasjonen av barysentriske koordinater for hver av trekantene sine. (Funksjonen GetCollisionPoint() gjennomfører utregningen av barysentriske koordinater og returnerer trekanten på meshen hvor ballen kolliderer.

```
//Get the point where collission takes place
Vector3 collission = PointCloudVisualize.instance.GetCollissionPoint(transform.position, mapPoint, radius);

//Check if gameObject collides with the mesh
if (collission != Vector3.one * -1f)
{
    //Start gameObject's cooldown process, before repawning it
    cooldown = true;

    //Use physics to setup the change of the gameObject's movement direction
    normal_Vector = -Vector3.Dot(collission, force_Vector) * collission;
    Vector3 normal_Velocity = Vector3.Dot(velocity, collission) * collission;
    velocity = velocity - normal_Velocity;

    //Increase water level with the gameObject's water amount
    RainManager.instance.waterLevel += RainManager.instance.waterInDroplet;
}
```

```
//Change gameObject position based on acceleration and velocity
acceleration = (gravity_Vector + normal_Vector) / mass;
velocity += acceleration * Time.fixedDeltaTime * RainManager.instance.dropletSpeed;
transform.position += velocity * Time.fixedDeltaTime;
```

- Vi bruker så informasjonen fra denne trekanten til å endre normalvektoren, akselerasjonen og til slutt farten, slik at vi kan endre ballens posisjon i takt med underlaget den treffer. Her benyttes også ballens radius, slik at vi kjenner avstanden den skal ha til underlaget den kolliderer med.
- Jeg har *ikke* implementert rulling eller kollisjonsdetektering mellom ballene, som foreslått i 3.1.1

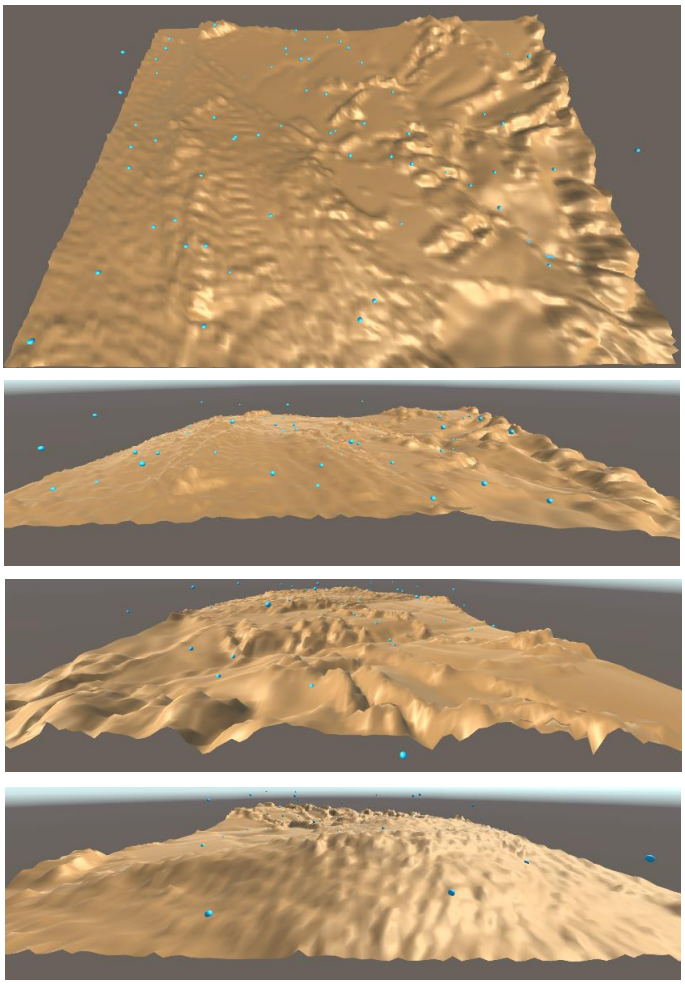
- I editoren har jeg satt det opp slik at man kan endre ulike parametre interaktivt, også om ballen skal spawne i et gitt punkt eller tilfeldig over terrenget.
- Det er også mulig å sette ballens levetid, slik at dersom det spawnes mange baller samtidig, så kan de forsvinne før pc'en knekker sammen. Jo lengre ballens levetid er, jo lenger ned i terrenget kommer ballen før den forsvinner.



Oppgave 3.2:

«Simulering av nedbør. Bruk mange random genererte små partikler til å simulere regndråper. Eksperimenter med antallet og med visualiseringen og dokumenter dette. Når regndråpene når bakken, kan de visualiseres som små baller og de skal bevege seg etter Newtons 2. lov. Kollisjon mellom disse regndråpene kan fint forekomme (uten kollisjonsdetektering/kollisjonsrespons).»

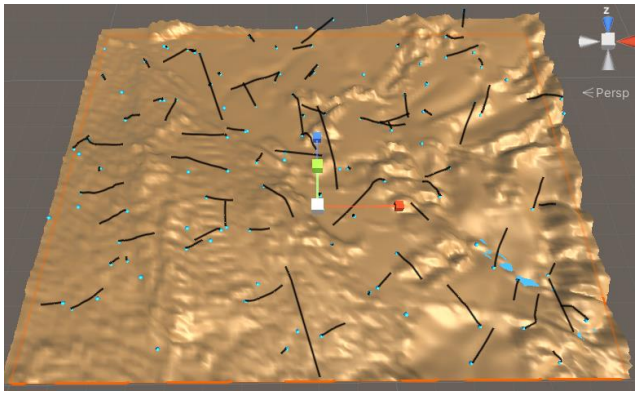
- Ballene som brukes i denne oppgaven er av samme type som i 3.1, som vil si at de beveger seg langs terrenget etter Newtons 2. lov.
- For å random-generere baller som regndråper som triller nedover terrenget, har jeg valgt å bygge en ObjectPooling som instansierer baller dersom poolen er full. Dette skaper noe frame drops i starten, men stabiliserer seg når de første ballene begynner å despawne. Ved en spawning av baller hvert 0.001s og en levetid på 4 sekunder fra ballene treffer terrenget, lages en pool på ca. 80 ballObjekter, noe som pc'en klarer på ca. 16fps.
- Fra ulike vinkler kan man se ballene trille nedover underlaget når de treffer det. Noen baller vil også trille utenfor terrenget og despawne etter kort tid.

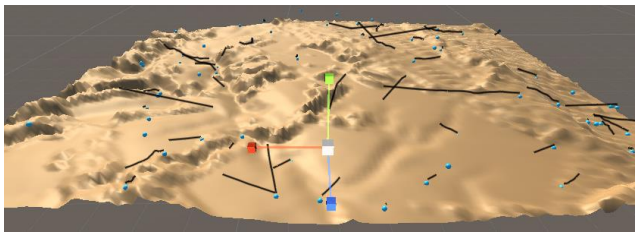


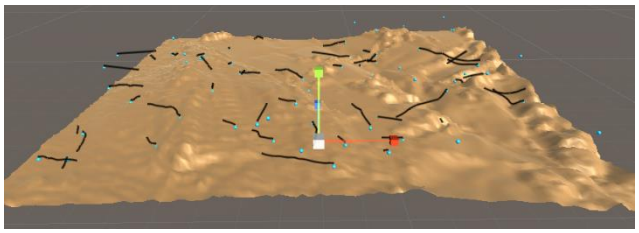
Oppgave 3.3:

«Simulering av vassdrag. Les av regndråpenes posisjon fra de treffer bakken og deretter for hvert tidssteg dt. Dette tidssteget bestemmes ekserimentelt, og du må velge hvor mange posisjoner du skal registrere. Bruk de avleste posisjonene (i planet) som kontrollpunkter for en kvadratisk splinekurve (i planet). Bruk dette til å lage og visualisere kvadratiske B-spline kurver som viser regndråpenes bevegelse langs bakken. Lag kurvene i planet, og bruk koordinatene i planet til å finne høyden på overflaten/terrenget. På denne måten løftes kurvene opp på overflaten.»

- På grunn av tiden denne «Folder Exam» har tatt, har jeg valgt å gjøre denne oppgaven på en noe forenklet måte, selv om effekten skal bli tilnærmet likt.
- Istedenfor å bygge opp kvadratiske splinekurver, så har jeg lagt til en «List<vector3> posList» i hvert ballObject som registrerer og legger til ballens posisjon hvert 0.1s fra den har truffet terrenget.
- Videre tegner jeg opp linjer mellom punktene som er registrert, slik at vi kan se en stripe der ballen har vært. (Jeg flytter linjen litt høyere enn ballen for at vi skal kunne se den skikkelig).
- På grunn av pc'ens kapasitet har jeg valgt å fjerne linjen når ballen despawner. Jeg kunne lagt listen over «ballens reise» inn i en hovedliste når ballen despawner, for å beholde linjen, men har valgt å ikke gjøre dette.

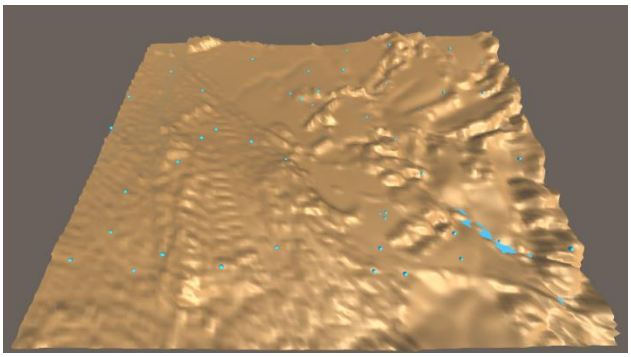


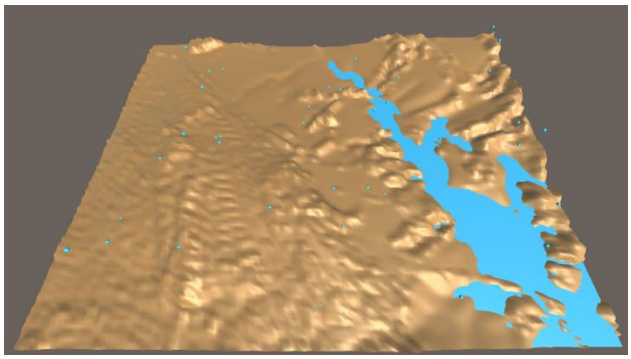




Oppgave 3.4:

«Ved mye nedbør skal ballen som har lagt seg til ro i oppgave 3.1 flyte sammen med vannet som dannes. Implementer denne effekten. Forklar/dokumenter hvordan du løser dette, og ta med viktigste beregninger. Det teller positivt hvis man legger til andre relevante objekter/effekter/funksjonalitet i tillegg til oppgavene som er gitt. Eksempel: bikvadratisk B-spline tensorproduktflate.»





- Ettersom jeg har en PC som ikke tåler å ha mange baller aktive samtidig, har jeg valgt å gi hver ball en mengde vann på 0.001.
- Hver gang en ball treffer meshen, registreres vannmengden og legges til en float som inneholder den totale vannmengden.
- Under terrenget har jeg plassert en blå kube som dekker terrengets mål. Y-verdien til denne kuben øker i takt med verdien i floaten, og får kuben til å forflytte seg oppover. På sikt lager dette en illusjon av at vannmengden stiger, basert på regndråpene som lander.
- Jo kraftigere det regner, jo fortere fylles vannet oppover.

- For å simulere mer lokal nedbør, så kunne jeg laget flere og mindre kuber under meshen. Hver kube kunne fått tildelt et område i det regulære gridet (eks. 5x5), hvor dråpene som lander på meshen innenfor dette området får vannstanden i kuben til å øke. Hver kube ville derfor økt sin høyde ulikt, og skapt flere områder med vanndannelse i terrenget.