2VSIM101

Visualisation and simulation 2025

Candidate number: 431

GitHub link: https://github.com/alanhaugen/gameengine

1 Introduction

Processing real-world terrain data and integrating it into an interactive simulation requires handling irregular point cloud data, transforming them into a structured mesh, and efficient rendering and collision detection implementations. This assignment explores ways to develop an experimental game engine and an application centred around loading data from the Dovre mountain region, Norway.

The purpose of the work is to demonstrate how to take unstructured LiDAR point data and transform it into a height-map terrain mesh suitable for real-time simulations. The results give insight into creating terrain meshes, point-cloud processing, and simulation techniques in graphics programming.

Section 2 describes the methods used to preprocess the LiDAR dataset, generate the terrain mesh, and implement rendering and simulation features in the engine. Section 3 presents the results of the terrain reconstruction and the behaviour of the simulation running on top of it. Section 4 discusses the strengths and limitations of the chosen approach and concludes the report with a summary of findings and possible directions for future work.

2 Methods

A point cloud consists of a large amount of x, y, z coordinates which can be rendered with APIs such as OpenGL and Vulkan.

The Norwegian state has LiDAR data taken the entire country available at the website https://hoydedata.no. The dataset used represents a geographical location known as Snøhetta (Kartverket 2016). See figure 1 and 2.

The downloaded LiDAR data has been compressed with LAzip and has to be converted into point cloud data. After this, to visaulise the point cloud as a mesh, it needs to be converted via technique called regular triangulation (Nylund 2025 page 166). Regular triangulation is a mesh with equal length between the vertices, making looking up data from the mesh trivial. See figure 3.

2.1 Point cloud visualisation

See figure 9 shows a visualisation of the point clouds rendered with a points pipeline in Vulkan. The points have been calculated with the accompanying

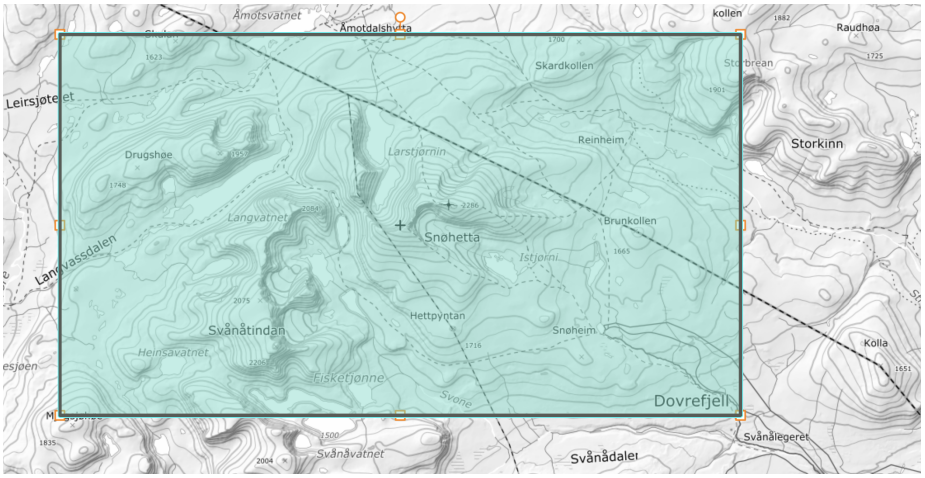

Figure 1: Snøhetta 62,343036°, 9,179474° as seen on the Kartverket website.

Figure 2: Snøhetta 62,343036°, 9,179474° as seen on Google Maps.

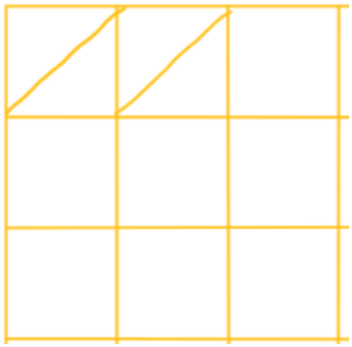

Figure 3: Regular triangulation mesh (Dag 2025 page 168).

Python script importlas.py.

Table 1: An excerpt of the resulting importlas.py conversion of LiDAR data:

509599.89	6908638.38	1746.08
509599.41	6908642.74	1749.23
509599.89	6908644.68	1750.72
509599.89	6908643.71	1749.92

The engine written to visualise the terrain is Entity Component System (ECS) based, terrain is considered to be a Component. The following code renders the terrain as points.

```
Terrain* terrainMeshPoints = new Terrain("Assets/output_smallest.txt", true);
```

There are different techniques available to turn a point cloud into a mesh. One of the better known algorithms is the Delaunay triangulation algorithm (Delanuay 1934). The technique is described in the lecture notes (Nylund 2025 page 161). It is possible to create the mesh offline, with a custom made program, and load in the data precomputed to save CPU-time. Another approach is to simply take all the points, and use a regular triangulation technique as described by Nylund on page 166 in the lecture notes. This was the approach was taken here, as it seemed simple. So, in order to render the height-map shown in figure 6, the point cloud was converted into a 2D image with a custom program, pointconverter. Pointconverter can be found in tools/pointconverter in the repository.

2.2 Pointconverter and terrain mesh visualisation

Pointconverter is a program written for this course to convert point data into a height-map.

The program will read the output of the python script importlas.py, and turn the points into a 2D greyscale image. The output will be stored into the file output.png.

```
$ ./pointcloud lazoutput.txt
```

The program will first figure out how large the data is and store all the point data in an array.

```
float x, y, z;
while (infile >> x >> z >> y) // Notice z and y are swapped
{
    static vec3 offset = vec3(x, y, z);

    vec3 pos = vec3(x, y, z) - offset;

    ...

    points.push_back(pos);
}
```

The data is then used to create a 2D array which represents the height-map.

```
for (auto point : points)
{
    float normX = (point.x - smallestX) / float(largestX - smallestX);
    float normZ = (point.z - smallestZ) / float(largestZ - smallestZ);
    float normY = (point.y - smallestY) / float(largestY - smallestY);

    int x = std::min(width  - 1, int(normX * (width  - 1)));
    int z = std::min(height - 1, int(normZ * (height - 1)));

    vertices[x][z] = u8(normY * 255.0f);
}
```

stb_image_write is used to record the image. The resulting image can then be loaded into the engine, which will calculate vertices, normals and UVs based on the data. Figure 3 shows an example image of a height-map generated from point cloud data

The benefit of using height-maps is the speed of loading the data and the widespread use of height-maps elsewhere in the games industry.

The game engine Terrain component also supports height-maps. To load height-map data, the following code can be used:

```
Terrain* terrainMesh = new Terrain("Assets/output.png");
```

2.3 Simulation

Code was written to make spheres (balls) which can roll along the normals of the terrain. These balls don't use realistic physics, instead simply glide along the surface.

A component called RigidBody was designed. Since the engine is using ECS (Entity Component System), a rolling ball can be devised like this:

```
GameObject* ball = new GameObject("Ball");
ball->AddComponent(new Sphere("Assets/Textures/orange.jpg"));
ball->AddComponent(new TrackingSpline);
ball->AddComponent(new SphereCollider(ball));
ball->AddComponent(new RigidBody(terrainMesh));
```

This will create a Game Object called Ball which can be edited in the editor, moved around, and it will roll automatically around on the terrain. It will also create a spline-based trail after it due to the TrackingSpline Component. This demo can be found in the demos project, games/demos/rollingball.cpp. Please note the engine is divided into many parts, and is designed to be used to make many different projects. See the screenshot below:

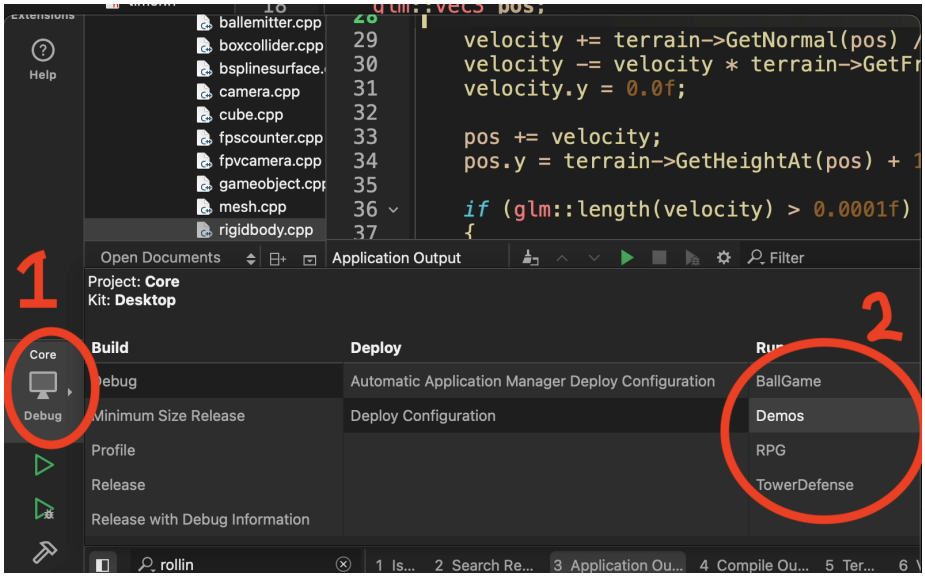

Figure 4: When using the game engine written for this course, please make sure to select the right project, this is done by selecting the Kit Selector (1), shown with the Computer icon, and then selecting which project to run (2).

Most of the logic is found in the RigidBody component, found in source/core/components/rigidbody.cpp:

```
glm::vec3 pos = gameObject->GetPosition();

velocity += (terrain->GetNormal(pos) / 500.0f) * deltaTime;
velocity -= velocity * terrain->GetFriction(pos) * deltaTime;
velocity.y = 0.0f;

pos += velocity;
pos.y = terrain->GetHeightAt(pos) + 1.0f;

mesh->SetPosition(pos);
```

The code above is the Update method for the component RigidBody. If it is attached to a game object, like in the rolling ball demo, it will move the ball along with the terrain mesh, and it will use the friction from the terrain to slow down. Some of the terrain has higher friction, which is shown in red. See figure 7. Note the physics are not realistic. This is to keep the simulation simple.

Phong shading is used to render the meshes, while a simpler colour shader is used for lines. The game engine supports as many pipelines as you like, and it is easy to swap out and try out new shader programs.

Another part of the assignment was to create a ball emitter with trails for the balls. This was accomplished by using a object pool with many spheres, spline tracking curve and rigidbody components (Object Pool). See figure 5 below.

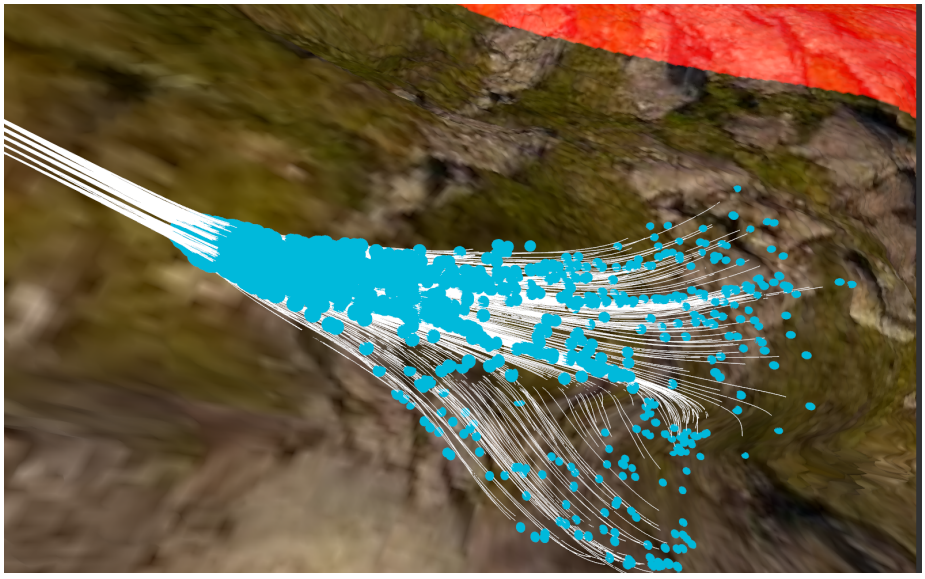

Figure 5: An example of 1500 rolling balls spawning from a water emitter component, together with 1500 splines showing their paths.

3 Results

The point cloud shown in figure 9 consists of over 4 GiB of data, and renders fine with the Vulkan renderer. All the Snøhetta laz data were concatenated into one large point cloud datafile. It does take a few minutes to load. The datafile is not included in the Assets folder in the repository, but all other necessary assets are included.

The pointcloud converter converts the data into the image.

The rolling balls simulation never allows balls to bounce off the surface. This is unrealistic, but it does simplify writing the code for the simulation. The algorithm described in section 9.6 of the lecture notes was applied (Dag 2025 page 136).

First, the triangle the ball is on was calculated. Thanks to the regular triangulation, looking up the two possible triangles it is on is trivial and very fast, calculating barycentric coordinates is only necessary for two triangles per look-up.

The normal vektor then gets calculated. The speed and position are updated for the ball according to the following formulas from the lecture notes:

$$v_k + 1 = v_k + a\Delta t$$

$$p_k + 1 = p_k + v_k\Delta t$$

Acceleration is simply terrain normal devided by 500 to slow it down, gravity and velocity in the y-direction is ignored for simplicity.

$$a_{slope} = \frac{terrain \quad normal}{500.0f}$$

GetHeight, GetNormal and GetFriction in terrain.cpp are all O(1) thanks to using a lookup table into the regular grid (generated with regular triangulation) instead of looping through all of the vertices and doing barycentric coordinate calculations on each vertex. This speeds up the simulation considerably.

The water emitter uses the Object Pool pattern (Nystrom 2021a). This pattern makes the simulation run faster by pre-allocating all the needed components up-front, and reusing them as needed. Sadly, this was not achieved with the tracking splines. Tracking splines take some processing to generate as old splines are removed and new ones are generated every so often, based on a timer. This results on noticeable lag, even on modern systems.

The engine has a built in collision detection and response engine, a physics engine. It is called AAPhysics, and is one of the Systems. You can use it to

create Sphere Components and Box Components, which can be attached to game objects to make them register and react to collisions. By adding Box Collider to the rollingsphere game objects, the spheres collide and stop when touching other objects with box or sphere colliders attached to them.

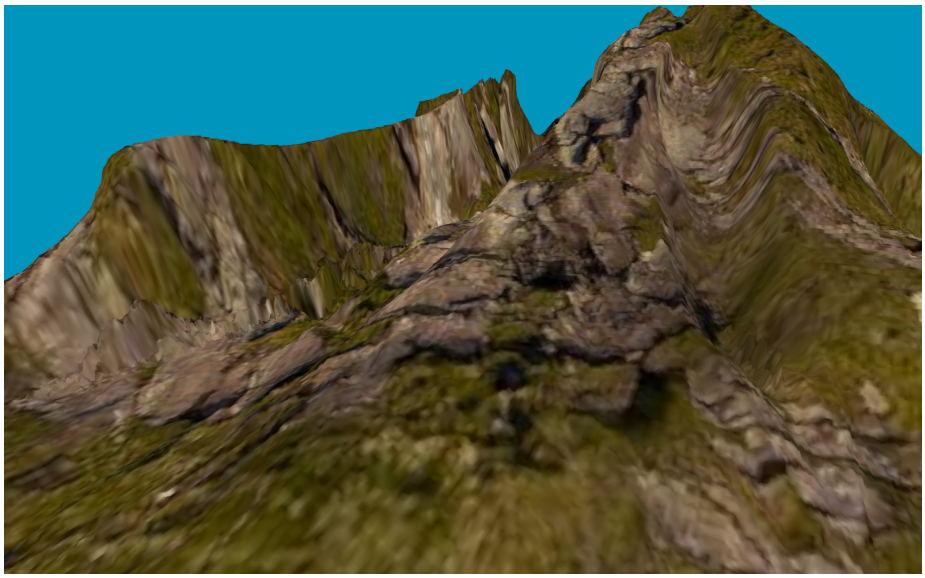

Figure 6: Textured mesh with normals of Snøhetta 62,343036°, 9,179474°, at Dovre, Norway.

4 Discussion

Using a hieght-map to visualise point cloud data worked well, it is both fast to load and easy to understand. Post processing effects, such as Gaussian blurr can be added to the image with existing tools to improve the results.

The editor allows customization and one can play with the simulation, adding components, game objects, and so on is both easy and fun.

All the tasks were completed. The hardest part was making RemoveDrawable in the renderer to work correctly. It has a dense array of render components, called drawawbles, and making holes in it can't be done. To make the program run as fast as possible, it is desired to make all the systems use components which are stored in contiguous data structures (Nystrom 2021b). The component lookup therefore uses sparse index array to make sure that already given out ids will still work when the size of the dense array changes.

The data from the pointconverter progam, the height-map it produces, can be smoothed with gaussian blurr or similar techniques to prevent errors as seen in figure 8 , depending on the quality of the attained point cloud data.

Figure 7: The high friction area is tinted red.

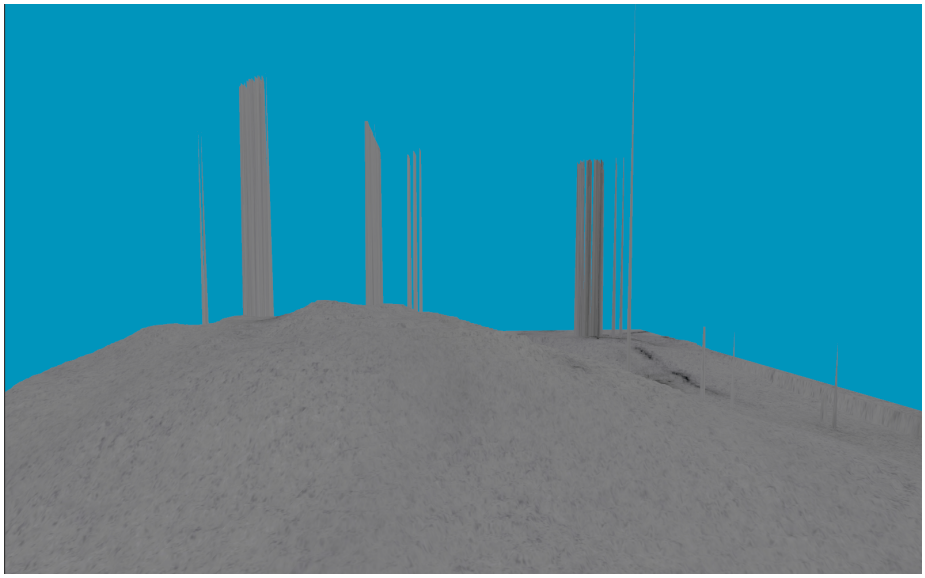

Figure 8: Here are some rendering artifacts which occur for some data sets. LiDAR data is messy, some points are erroneously placed off the ground. This results in strange geometry as seen here. Solutions to solve this are discussed in the next section.

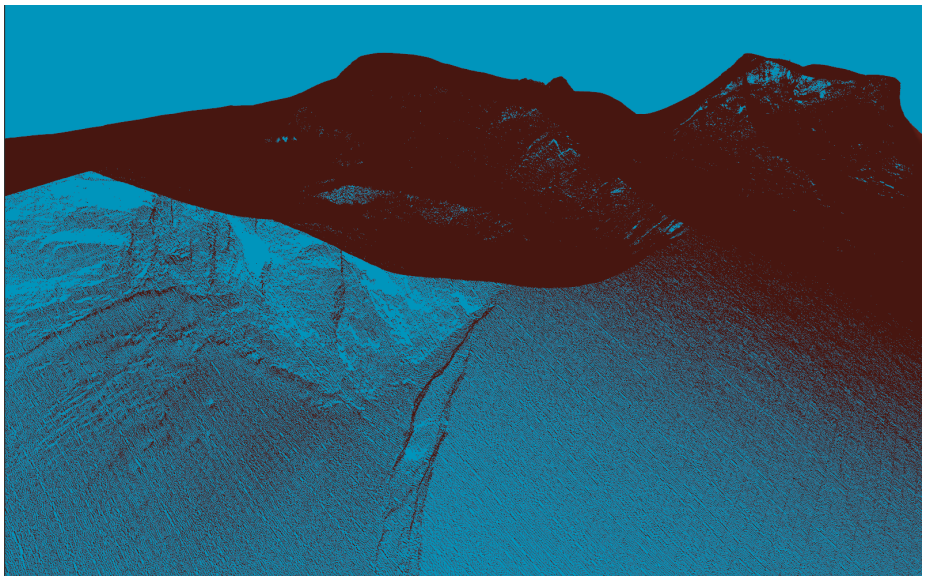

Figure 9: Point cloud data from Kartverket of a hill at Dovre.

To improve realism, a fully dynamic rigid body system could be implemented, incorporating Newtonian gravity, proper acceleration due to slopes, and correct rotation and collision response for rolling objects.

Due to the ECS (Entity Component System) implementation, the performance is really good and a total of 1500 water particles are used in the water emitter. Thanks to fast GetHeight, GetNormal and GetFriction implementations, and thanks to data locality (ECS), the game engine ends up being performant.

References

Delaunay, Boris (1934). “Sur la sphère vide” [On the empty sphere]. Bulletin de l’Académie des Sciences de l’URSS, Classe des Sciences Mathématiques et Naturelles

Kartverket (2016) Fotogrammetrisk generert punktsky Snøhetta klassifisert med bakkepunkt, uklassifisert og støy.

Nylund, Dag (2025) MAT301 Matematikk III VSIM101 Visualisering og simulering forelesningsnotater og oppgaver

Nystrøm, Robert (2021a) Object Pool https://gameprogrammingpatterns.com/object-pool.html

Nystrøm, Robert (2021b) Data Locality https://gameprogrammingpatterns.com/data-locality.html