G12

Simulation environment for point cloud scanning augmented reality application development

User Manual

Veikko Svanström
Eetu Luoma
Valtteri Tiitinen
Tomi Miettinen
Anton Rantamäki
Niilo Rannikko
Veeti Vanha-Kämppä

Version history

Version	Date	Author	Description

1.0	27.04.2025	Eetu L.	Compile the manual

Contents

1 Introduction

This document comprises the user manual for the GamiLiDAR point cloud scanning simulation project, with the purpose of being a sort of super-document that users or developers can refer to.
The document will contain a short guide on how to get going using the simulation product, and then compile the documentation written about the components of the product separately.

2 Getting started

2.1 Install dependencies

- Install Unity Hub from https://unity.com/download
- Install Unity 6 LTS **6000.0.35f1** from https://unity.com/releases/editor/archive

2.2 Clone our repository

Clone this repository to your machine using your favorite Git client.

2.3 Usage

Open Unity Hub, navigate to `Projects -> Add -> Add project from disk` and add the repo you cloned.

Click on the project in Unity Editor and start development!

2.4 CI Pipeline

See our readme for pipeline information

3 Individual component documentation

In this section the documentation of individual components are compiled into their own sections. The same contents can be found in their respective documents in the project repository.

3.1 Terrain generation

3.1.1 Overview

Generation of the simulation's game world and its terrain is handled through a dedicated MapGenerator class. At the start of the game the class generates the

ground surface, a road through the area, bordering mountains to block access to outside of the play area and vegetation for the player to scan and see in the simulation. The class uses a perlin noise -based algorithm to procedurally create the height variations of the world and borders. The class has a seed parameter that can be changed in the Unity editor to change the layout of the generated world. The class has a variety of other parameters that can be changed in the code which affect the generated world size, borders, road and form. By default the values are set to a reasonable level and shouldn't require changing unless specifically wanted. The MapGenerator class also uses VegetationGenerator, MapDisplay, MeshGenerator and TextureGenerator classes to divide parts of the terrain generation process.

3.1.2 Key features

Terrain generation:
- Uses Perlin noise to generate a height map for the terrain.
- Supports configurable parameters like mapWidth, mapHeight, noiseScale, heightMultiplier, and more.
- Includes a cubic easing function to smooth transitions near the map's borders.

Road generation:
- Creates a simple road running vertically through the map.
- Smoothly blends the road into the terrain using easing functions.
- Marks road positions on a roadMap for further processing.

Boundaries generation:
- Automatically generates walls around the map to act as boundaries.
- Walls are created or updated dynamically based on the map's dimensions.

Rendering modes:
- Supports two rendering modes: NoiseMap (visualizes the noise map as a texture) and Mesh (renders the terrain as a 3D mesh).
- Dynamically loads materials for terrain, roads, and mountains.

Vegetation integration:
- Integrates with a VegetationGenerator to add vegetation to the terrain.
- Clears and regenerates vegetation when the map is updated.

Validation:
- Ensures that parameters like mapWidth, mapHeight, noiseScale, and others are clamped to valid ranges using the OnValidate method.

3.1.3 Key methods

GenerateMap():
- Main method for generating the terrain.

- Creates the noise map, applies height adjustments, generates roads, and renders the terrain.

GenerateRoad():
- Generates a vertical road on the terrain.
- Smoothly transitions the road into the surrounding terrain using easing functions.

CreateBoundaries():
- Creates walls around the map to act as boundaries.
- Dynamically adjusts wall size and position based on the map dimensions.
- Uses helper function CreateWall to make individual walls

SetDefaults():
- Resets all parameters to default values.

OnValidate():
- Ensures parameter values are valid and triggers map generation if autoUpdate is enabled.

3.1.4 Fields and properties

Public fields:
- seed: Random seed for noise generation.
- autoUpdate: Automatically updates the map when parameters are changed.

Private fields:
- renderMode: Determines the rendering mode (NoiseMap or Mesh).
- lastWarningTime and warningCooldown: Prevent spamming warnings in play mode.

Inspector fields (hidden):
- Terrain dimensions: mapWidth, mapHeight, borderWidth, roadWidth.
- Noise parameters: noiseScale, octaves, persistence, lacunarity.
- Materials: terrainMaterial, roadMaterial, mountainMaterial.
- Vegetation: vegetationGenerator.

3.1.5 Usage

- Attach the MapGenerator script to a GameObject in Unity.
- Configure the parameters in the Inspector or use the SetDefaults method.
- Call GenerateMap() to create the terrain.

3.1.6 Summary

The MapGenerator class is a robust and flexible tool for procedural terrain generation in Unity. It combines noise-based algorithms, road generation, and boundary creation to produce detailed and customizable terrains. Its integration with vegetation and rendering options makes it suitable for a wide range of applications, such as games or simulations.

3.2 Point cloud scanning and saving

3.2.1 Overview

The point cloud scanning system in Unity captures and stores 3D environment data by shooting rays, detecting surface hits, and recording positional and color information. The data is saved in a structured format for further visualization or processing.

3.2.2 How it works

1. Point Structure
Each scanned point consists of:

- Position: A Vector3 representing the 3D coordinates of the point.
- Color: Stored as a 32-bit RGBA value for compatibility with Unity's VFX Graph.

2. Scanning Process

- The camera emits multiple rays in random directions (NumRays adjustable for accuracy).
- Each ray detects collisions with scene objects within the MaxDistance.
- If a valid surface is hit:
    - The position and color of the hit point are recorded.
    - The system avoids capturing unnecessary objects like the player or known scene meshes (e.g., "SeenMesh").
    - For objects with LOD groups, the closest mesh in LOD0 is used for scanning.
- The SeenMeshManager manages the hit triangle for future use by the AR games

3. Point Cloud Accumulation

- A scan is triggered when the camera moves significantly (CameraTranslationThreshold) or rotates (CameraRotationThreshold).
- Collected points are stored in a list (_points) for long-term storage and _scannedPoints for the current scan.

4. Visualization

- Captured points can be displayed using the PointCloudVisualizer component.
- A GameObject named PointCloud holds the visual representation.
- Materials can be applied for rendering and highlighting points.
- Visualization can be toggled on or off using the KeyCode.U.

5. Data Storage

- For each ScanObject() call:
    - One camera entry is written.
    - All point cloud entries of the current scan are stored.
- Data is written to text files asynchronously to avoid blocking the main thread.
- File Structure:
    - Point Cloud Data: Stored in pointcloudstr.txt with each line containing:
        - x y z r g b timestamp
    - Camera Data: Stored in camerastr.txt with each line containing:
        - x y z α β γ timestamp
- Files are saved in the folder:
    - PointCloudDataSets/{timestamp}_{identifier}/

- The system uses StringBuilder to buffer data and flushes it at regular intervals (_flushStringsTime).

3.2.3 Code Flow

1. **Initialization (Start):**
    - Sets up the SeenMeshManager and PointCloudVisualizer.
    - Creates necessary directories and files for saving data.
    - Initializes variables like _points, _scannedPoints, and timers.
2. **Scanning (Update and ScanObject):**
    - Checks if the camera has moved or rotated enough to trigger a scan.
    - Shoots rays, detects hits, and records point data.
    - Stores hit triangles using SeenMeshManager.
3. **Saving (SavePointCloud and SavePoint):**
    - Saves camera and point data to StringBuilder.
    - Flushes data to files asynchronously to prevent performance issues.
4. **Visualization (TogglePointCloudDisplay):**
    - Toggles the visibility of the point cloud in the scene.
5. **Cleanup (OnDestroy and OnDisable):**
    - Ensures all buffered data is flushed and files are closed properly.

3.2.4 Key Components

- PointCloudVisualizer:
    - Handles the visual representation of the point cloud.
    - Adds points to a graphics buffer for rendering.
- SeenMeshManager:
    - Manages and updates triangles that have been scanned.
- PointCloud GameObject:
    - Holds the visualized point cloud in the scene.

3.2.5 Error handling

- If the PointCloud GameObject is missing, a warning is logged, and visualization is disabled.
- If the MainCamera is not found, scanning is skipped, and an error is logged.

3.2.6 Asynchronous saving

- Writing data to files on the main thread can cause performance issues, especially in real-time applications.
- By using asynchronous saving (FlushStringBuildersCoroutine), the system ensures smooth operation without freezing the game.

3.2.7 Summary

The system efficiently scans and saves a 3D point cloud in Unity, leveraging structured storage and visualization. It is designed for real-time data collection, making it suitable for AR applications, simulations, and research projects. The use of asynchronous saving and modular components ensures high performance and flexibility.

3.3 Simulated AR Mesh

3.3.1 Overview

The SeenMeshManager is a component that manages and visualizes triangles that have been scanned or "seen" during a point cloud scanning process. It tracks scanned triangles, updates a mesh representation of the seen areas, and provides functionality for toggling visibility and updating mesh colliders. This system is essential for ensuring that previously scanned areas are not redundantly processed, improving performance and accuracy in AR or simulation applications. Creating a completely new mesh using only the scanned point cloud data would be difficult to implement and require heavy calculations which would substantially slow down the simulation. Instead of creating a completely new mesh from the point cloud data, we use the existing meshes of the scene to create a mesh that will be updated with new triangles in real-time as they are hit by ray casts. This way we can easily build a new mesh using existing meshes and without any calculations. The performance of SeenMeshManager is optimized by updating colliders only once every second and caching large lists of vertices and triangles.

3.3.2 How it works

1. Core Components
The SeenMeshManager consists of the following key components:

- Mesh (_seenMesh): A Unity Mesh object that stores the vertices and triangles of the seen areas.
- Triangle Tracker (_seenTriangles): A HashSet<int> that ensures each triangle is only processed once.
- Vertex and Index Lists (_seenVertices, _seenIndices): Lists that store the vertices and indices of the seen triangles.
- MeshCollider and MeshRenderer: Components for collision detection and visibility control.
- Dictionaries for the seen mesh triangles (_meshTrianglesCache) vertices (_meshVerticesCache): Caches to avoid repeated access to large lists

2. Initialization
The SeenMeshManager is initialized with the following steps:

- A new Mesh object is created with 32-bit indices enabled to support large meshes.
- Internal data structures (_seenTriangles, _seenVertices, _seenIndices) are initialized to track the scanned data.
- A new GameObject is created to hold the mesh, with MeshFilter, MeshCollider, and MeshRenderer components attached.
- The MeshFilter and MeshCollider are assigned the shared mesh (_seenMesh).

- The GameObject is tagged as "ARMesh" and assigned to the "ARMesh" layer for filtering purposes.

3. Adding Triangles

The AddTriangleToSeenList method processes triangles hit by a raycast:

Triangle Validation:

- The triangle index from the RaycastHit is checked to ensure it is valid and not already processed.
- Bounds are validated to prevent out-of-range errors.

Vertex Transformation:

- The triangle's vertices are transformed from local to world space using the MeshFilter's transform.

Data Storage:

- Unique vertices are added to the _seenVertices list.
- Indices are updated to reference the newly added vertices.

Mesh Update:

- The UpdateSeenMesh method is called to update the mesh with the new vertices and triangles.

4. Updating the Mesh

The UpdateSeenMesh method ensures the SeenMesh is up-to-date.

- Clears the existing mesh data.
- Assigns the updated vertices and triangles to the mesh.
- Recalculates normals for proper lighting and shading.
- Calls UpdateMeshCollider in intervals of 1 second to synchronize the MeshCollider with the updated mesh.

5. Visibility Control

The ToggleVisibility method allows toggling the visibility of the SeenMesh:

- Uses the MeshRenderer to enable or disable rendering of the mesh.
- Updates the mesh to ensure it is vizualized correctly

3.3.3 Code Flow

Initialization

- The SeenMeshManager is instantiated and initialized with empty data structures.
- A GameObject is created to hold the mesh, and its components are configured.

Scanning and Adding Triangles

- When a raycast hits a triangle, the AddTriangleToSeenList method is called.
- The mesh’s triangles and vertices are fetched from the cache if they have been cached, otherwise fetch them directly and cache them for later use
- The triangle's vertices are transformed, stored, and added to the mesh.

Mesh Updates

- The UpdateSeenMesh method is called whenever new triangles are added.
- The MeshCollider is updated once every second to reflect the changes.

Visibility Toggling

- The ToggleVisibility method is used to show or hide the SeenMesh.

3.3.4 Key Components

Mesh (_seenMesh)

- Stores the vertices and triangles of the seen areas.
- Updated dynamically as new triangles are added.

Triangle Tracker (_seenTriangles)

- Ensures that each triangle is only processed once.
- Prevents redundant calculations and improves performance.

MeshCollider and MeshRenderer

- MeshCollider: Synchronizes with the SeenMesh for collision detection.
- MeshRenderer: Controls the visibility of the SeenMesh.

Cache for mesh triangles and vertices

- _meshTrianglesCache: Stores triangle lists in a dictionary where the mesh is the key
- _meshTrianglesCache: Stores vertex lists in a dictionary where the mesh is the key

3.3.5 Error handling

- Invalid Triangle Index: Logs a warning if the triangle index is out of bounds.
- Missing Components: Logs a warning if the renderer is missing when trying to set a material to the renderer and if the collider is missing when trying to update the collider.

3.3.6 Integration with PointCloudManager

The SeenMeshManager is tightly integrated with the PointCloudManager:

- The PointCloudManager uses the SeenMeshManager to track and highlight triangles hit during the scanning process.
- The AddTriangleToSeenList method is called whenever a valid triangle is detected by a raycast.
- The ToggleVisibility method is mapped to a key press (KeyCode.Y) to allow toggling the visibility of the seen mesh. (This will soon be delegated to a separate Input class)

3.3.7 Summary

The SeenMeshManager is a robust system for managing and visualizing scanned triangles in Unity. It efficiently tracks seen areas, updates a mesh representation, and provides tools for collision detection and visibility control. This makes it an essential component for AR applications, simulations, and real-time 3D scanning projects. Its modular design ensures flexibility and ease of integration into larger systems.

3.4 Point cloud visualization

See in-code documentation.

4 Game development

4.1 Spider game port

We succeeded in porting the pre-existing spider game onto our simulation. (Issue #25) (Branch of port)
This section describes what steps were necessary to achieve this, with the aim of being a useful reference for future porting of games between the core application and our platform. The description may not be exact or exhaust all the actions that were done in the Unity editor, but it should be a sufficient reference for this purpose.

The port has happened in stages, due to the simulation platform being continuously developed during the project.

4.1.1 Porting to Simulation platform

- Export package from the core application (from context menu of the spider game Scene). You are exporting a package of the Scene of the spider game.
- Import into the simulation Unity project
- Exporting the package will miss the ConeCollider prefab object (Assets/ConeCollider/Resources/Prefab). Copy the whole ConeCollider folder by hand into the simulation project.
- fix possible errors (delete files or implement fix depending on situation)
    -errors to fix:
    - Delete various AR scripts not needed in the simulation which have errors, delete them
    - There is prompt for api changes, asking if you want to fix them. Select [yes, for these files only.]
- There are errors in some TextMesh Pro –scripts if you imported those from the core application, about some UnityEngine Vector types. Fix those by hand by changing to appropriate types

- Copy-paste Terrain-scene objects into SpiderGame-scene
- Put vacuum_collider object from under XR Origin to under Player->Main Camera
- Remove XR Origin-object and AR Session-object
-Remove duplicate Directional light
- Activate EventSystem –object and press the Fix button (Replace with InputSystemUIInputModule) for the Input System UI Input Module component of the object in the Inspector
- change shader for spider material (URP - Lit), put spider texture in its basemap
- Add the Tags that the code needs (will prompt exceptions when missing)
- Attach ARMesh –Tag to the terrains wanted to be used for spider spawning (we dont have the point cloud mesh that the real application has, yet....)
- add hack for Default-ParticleSystem (conflicts with our pointcloud scanning)
(See spider_game_port –branch commit history)

- add hack for vacuum_collider functionality
(See spider_game_port –branch commit history)
- Add LayerMask “ARMesh” to spiderspawner.cs raycast to only spawn spiders on surfaces tagged as ARMesh
- Add asmdef file to SceneSelector to have StaticVariables available
- Add asmdef file to com.8bitgoose.asciifbxexporter Editor and Runtime folders and add the Runtime asmdef to the Editor asmdef’s refrences. The editor asmdef should have “Editor” in includedPlatforms
- Do the same as above for jp.keijiro.pcx@1.0.1 Editor and Runtime folders.
- Add other missing dependencies to asmdef files to prevent errors
- See project gitlab issue #100 (https://gitlab.tuni.fi/cs/gamilidar/simulated_forest_scanning/-/issues/100): for some reason, our raycast for point scanning scanned points of the spiders if the spiders were in the air (walking on player collider / map edge). To prevent this from happening, spiderspawner.cs had a line of code added which sets a layer for the spawned spiders telling the raycast to ignore them: **createdObject.layer = LayerMask.NameToLayer(“Ignore Raycast”);** This is the simplest solution, as at this point of the project the group shouldn’t start refactoring the point scanning for this reason.

- The class StaticVariables from StaticVariables.cs must be given the public scope

- Copy the Keybind Manager object into the spider scene

4.2 UFO game

We created a ufo shooting game at the proof-of-concept level (Issue #72). The customer wished to know / see how this game would function on top of the core application, so we ported the game onto it for this purpose.
This document describes what steps were necessary to achieve this.

4.2.1 Porting to GamiLiDAR core app

- Export ufo scene from simulation, Import to core
- Remove our tests that came with the package
- Copy Xr origin, ar session to ufo scene
- Move rifle from under Player->main camera to under XR origin ->main camera
- Delete fog objects, map generator, terrain mesh, vegetation generator, roadmesh, pointcloud, keybind manager, UI canvas
- Copy lighting settings from some other existing game (dont want to use our simulation skybox etc.)
- Delete our scripts files (pointcloudmanager, pointcloudvisualizer, pointcloud, seenmeshmanager, vegetationgenerator, mapgenerator, legendmanager, keybindmanager, playermovement)
- Restore StaticVariables.cs into its contents before importing was done as our version overwrites it

- Delete assets/editor/generatoreditor
- Set ufo prefab in UfoSpawner object (prefab UfoFinal in assets)
- Set rifle transform to (0.035, -0.03, 0.06) so it's positioned well with camera
- Add UFO2 scene to the list in SceneSelector scene's "selector" object

4.3 Things we learned

- Avoid relying on simulation specific scripts/objects (MapGenerator etc.)
- Use canvas objects for game input. E.g. ufo game and the spider game use a UI Image object with an Event Trigger component.
- In order to avoid .asmdef -dependency jungles, it may be best to not use the Unity test framework

5 Interface documentation

We have used xmldoc to document the various interfaces of the project codebase (issue 99).
Doxygen is used for compiling the code documentation via the CI pipeline. The generated PDF file can be found here.