Development of Real-Time Smart City Mapping Utilizing Game Engines

Tucker Clark, Evan Brock, Dalei Wu, Yu Liang
Department of Computer Science and Engineering
University of Tennessee at Chattanooga, Chattanooga, TN 37403

***Abstract*— Game engines are ideal platforms to generate and visualize digital twins of smart cities in real time. The real-time mapping of a smart city faces two challenges: (1) streaming data from Internet of Things (IoT) devices , (2) rendering a high-throughput and heterogeneous digital twin. Current game engine infrastructure is not built to handle the influx of real time data streams from a diverse array of IoT devices, nor can they render a real-time dynamic mesh streamed from a scanning device such as a LIDAR. As meshes are the basic framework needed to render digital twin objects that represent their real word counterparts and real time IoT streams are necessary for modeling an accurate digital twin, both of these issues must be resolved in order to use game engines to create a digital twin of a smart city. In this paper, we propose a networking infrastructure capable of handling a wide variety of IoT devices and a novel mesh rendering algorithm. Additionally, we provide a quantitative error analysis. Experimental results show that the proposed streaming method and rendering algorithm could enable a game engine to efficiently generate a digital twin of a smart city.**

Keywords: LIDAR, mapping, GIS, augmented reality

I. Introduction

Visualization is one of the most powerful tools used by computer scientists to portray the information that we gather. Recently, visualization has become close to a requirement for all subfields, due to the large amount of data generated by the numerous different sources found in each domain. As fields commingle, and interdisciplinary work becomes more and more common, we will continually push our current tools to their limits as they attempt to integrate with systems that they were not built to mesh with. Every so often, a tool will fit perfectly into an interdisciplinary application at a high level, but will not have the underlying low level framework built to interface with new and unique software. This is currently the case with geographic information systems (GIS) and game engines [1].

Game engines are often used to render large, detailed, 3D environments, the same kind that geospatial experts seek to replicate. The coordinate system within any game engine can be used to replicate 3D localization of objects and terrain, while taking advantage of their optimization and portability. Both interactable and performant, game engines seem to be the perfect candidate to visualize and interact with the geographic environment, and thus are a near perfect candidate to visualize a smart city [2]; industry clearly agrees. Both Google and Mapbox have built APIs and SDKs in order to bring their infrastructure and frameworks into the Unity game engine [3], [4].

But, game engines are just not built with IoT devices in mind, which predominantly power the data pipeline of any smart city. Game engines are simply not built to handle live streaming data from unsupported objects, nor are they built to render dynamically changing meshes defined by live streaming data. In this paper, we describe both a framework used to connect IoT devices to game engines through the use of low level networking and a novel algorithm used to circumvent current mesh rendering limitations found in modern game engines, along with experimental results verifying our theoretical findings. This enables both a data feed and data visualization of a smart city in a game engine, a desirable framework for geospatial experts seeking to model infrastructure.

II. Related Work

LIDARs are very desirable instruments for three-dimensional mapping, because a basic mapping algorithm involves two operations: acquisition of the LIDAR data, and localization of sensors. Most previous work seek to localize an object through deducing their own location through LIDAR data [5], [6]. Other work uses a combination of telemetry sensors and LIDAR data to achieve the same purpose [7]. While these work great for object detection or short term scans, they do not support collaborative scans, where multiple scans can be stitched together automatically through the geographical significance of their vertices in any three-dimensional environment.

Using game engines for visualization of real GIS data is uncommon, but [8] makes a notable step towards the normalization of game engine mapping by developing an application for 3D viewing of real data inside the game engine, Virtools. They specifically note that their choice of a game engine for 3D viewing is because of their "powerful render engines that allow the visualisation of complex, highly detailed landscapes in 3D in real-time". These render engines are desirable for allowing us to process and render data in real-time, with acceptable performance

In this paper, a GPS is used for absolute localization, and telemetry sensors for precise movements to store scans with respect to geographical coordinates. Our algorithm also allows for the reconstruction of an environment to be observed in real-time. This is not an uncommon feature for mapping technologies [9], but the implementation of our live

This work was supported by the National Science Foundation under grant numbers 1647175 and 1924278.

maps on such a large scale inside a game engine has proven to be very intuitive in our testing.

III. Problem Statement

Game engines have pros and cons for smart city mapping applications. Game engines are development environment that are created solely for the purpose of making video games. Abstracting that, they are development environments containing tools that make it easy to manipulate a virtual environment.

To use a game engine to aggregate data streamed from IoT devices, the software must be designed in such a way that connectivity is supported to a wide variety of devices, diverse in both operating system and in computational power. This is the case with IoT devices. IoT devices are often low power, low performance microcontrollers that run on some type of embedded operating system, or run on a lightweight distribution of Linux. In either case, the data sources for a smart city, IoT devices, cannot connect to game engines as a client using the networking modules built into most game engines.

Another issue underlying game engines, is that they are built to render objects fetched from secondary memory periodically, such as when a new level or map is loaded into RAM in order for the player to interact with it in game [10]. In all cases, this object comes in the form of a 3D mesh. In a smart city application, real world object data will need to be streamed into the engine. In this case, the mesh used to represent the object will not be known in advance, but will be built procedurally as the information is streamed into the engine. Game engines are currently not optimized to handle this operation, as this feature would never need to be present in a traditional game. Thus, it is the case that the underlying data structures supporting mesh generation, do not lend themselves to the scenario of a live, constantly mutating mesh.

There has been research has been conducted on the different rendering methods available inside of current game engines. But, to the best of our knowledge, no research has revealed whether or not a game engine can handle live streaming data from IoT devices. Most often, meshes begin as raw point cloud data, coming from a scanning device such as a LIDAR, which we will be using in our experiment to generate point cloud data. So we ask: can a game engine support the live streaming data from a wide variety of IoT devices? If it can, can the game engine render this data in real time?

IV. Data Preprocessing

The data that is used to generate a scan is interpreted as a collection of the geographical data from the drone and the relative data from the LIDAR. The drone is responsible for recording the offset of each scan, which is the vertical, horizontal and orientation components difference from the user-defined geographical zeroing point. The LIDAR detects all objects within range and records their relative position to the drone. The offset of the drone and the relative data are processed every frame such that the offset is added to each relative point to give each point geographical significance, a process described in Equation 2. This allows us to take multiple scans that will align automatically if parts of them overlap. Keep in mind that the offset of the drone also includes the orientation of the drone, so all recorded data will always be placed on the same plane the drone is on when a certain scan is recorded. This processed data is used for all of the following rendering methods.

We consider X, Y, and Z to be the Cartesian components for the three dimensional LIDAR data relative to the drone. For a given point, n, the components of roll, pitch, and yaw are κ_n, ρ_n, and ψ_n, respectively. The distance measured by the lidar is returned as d_n. For each point, κ_n and ρ_n are usually the same as κ_{n-1} and ρ_{n-1} and only vary once the LIDAR redefines as new scan with the new telemetry data, which results in κ_n and ρ_n being updated. ψ_n will never be the same as ψ_{n-1}, as well as d_n This is because the LIDAR will scan many points, returning a different ψ_n and d_n for each point before the κ_n and ρ_n are updated. The Cartesian components X_n, Y_n, and Z_n of point n can be computed as

$$\begin{bmatrix} X_n \\ Y_n \\ Z_n \end{bmatrix} = \begin{bmatrix} d_n sin(\psi_n) * cos(\kappa_n) \\ d_n cos(\psi_n) * cos(\rho_n) \\ d_n \sqrt{(sin(\kappa_n)sin(\psi_n))^2 + (sin(\rho_n)cos(\psi_n))^2} \end{bmatrix} \quad (1)$$

The derivation of components X_n, Y_n, and Z_n can be explained as follows by assuming the drone rolls along the Y-axis, pitches along the X-axis, and yaws along the Z-axis. X_n is determined by every rotation of the drone except for the rotation along the X-axis, which is pitch, ρ. The value of $d_n sin(\psi_n)$ increases as ψ approaches 90° and 270°, because those are the angles perpendicular to the X-axis, and therefore, the farthest away from it, increases the value of this component. $d_n sin(\psi_n)$ is then multiplied by $cos(\kappa_n)$ to account for the return toward the X-axis as the drone rolls. As the drone rolls farther, the X component becomes closer to 0 as ρ approaches 90°, because the scanned point would be directly above or below the X axis in such a case.

Y_n can be determined very similarly to the X component, except it is unaffected by roll, κ. It approaches maximum distance as ψ approaches 0° and 180°. The value of the Y component can also equal zero if ρ were to equal 90°, where the drone is in completely vertical pitch, in which case, the Y component would be directly above or below the Y axis.

Determining Z_n is slightly more complex, because all orientations of the drone affect the vertical location of a data point. The ρ and κ both affect the vertical position of a point. The significance of ρ and κ varies based on ψ, because the closer a point is to an axis that it is rotated around, the less its vertical position changes. For example the significance of the first half of the equation, $d_n sin(\kappa_n) sin(\psi_n)$, increases as ψ approaches 90° or 270°, at these points a change in κ is most significant because the point is as far away as possible from the axis it is being rotated around. Similar is true for the second half of the equation, $d_n sin(\rho_n) cos(\psi_n)$, except a change in ρ is most significant at a ψ of 0° and 180°. Because

of this relationship, as the significance of a change in one component increases, the significance of a change in the other component decreases. This is because as distance from one axis of rotation increases, the distance from the other axis of rotation decreases. The square root of the squared components, $d_n sin(\kappa_n) sin(\psi_n)$ and $d_n sin(\rho_n) cos(\psi_n)$, is taken to calculate the entire Z component.

It is important to note that ψ_n is representative of the same axis that both the LIDAR rotates on and the drone yaws on. Because of this, we compensate for the drone's rotation such that the ψ_n is always relative to an absolute point, and is not affected by the yaw of the drone. We consider D to be the

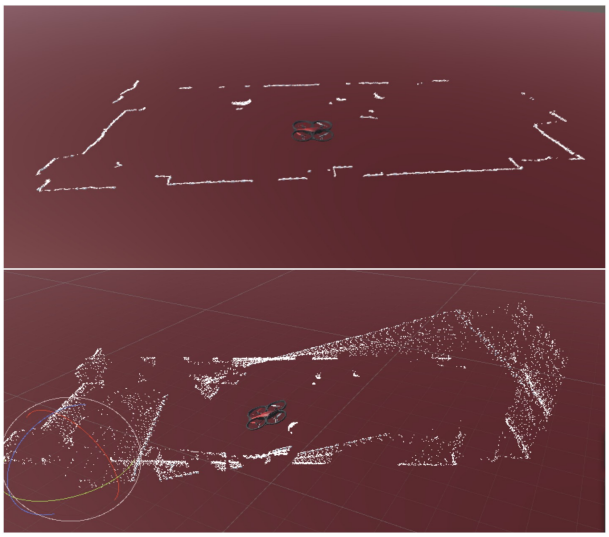

Fig. 1: *Application of a rotation of the drone in an indoor environment*

three-dimensional position of the drone in meters, and can add geographical significance to any point by adding the X, Y, and Z components of D (i.e., D_x, D_y, and D_z), to the relative components of any point: X_n, Y_n, and Z_n. This will allow us to produce the geographic set of coordinates, G_x, G_y, and G_z. This is important because sometimes, when there are no objects to represent the global location of the drone, we need to place vertices for our point cloud with respect to the global origin, rather than relative to the location of the drone at a certain time.

$$\begin{bmatrix} G_x \\ G_y \\ G_z \end{bmatrix} = \begin{bmatrix} X_n \\ Y_n \\ Z_n \end{bmatrix} + \begin{bmatrix} D_x \\ D_y \\ D_z \end{bmatrix} \tag{2}$$

V. 3D Rendering

When rendering the data in a 3D environment, it is important that the objects used to render can hold a global position and be combined with old data seamlessly. The amount of data rendered increases throughout a scan, meaning a lightweight rendering method is highly favorable. It is important that the algorithm chosen to do this performs well, because if the frames per second (FPS) drops below the scan rate of any scanning device, data will begin to be thrown out, as it will possibly be updated before the next frame is processed.

A. Rendering Methods

Point clouds have many properties that make them ideal for mapping LIDAR data in 3D environments. The problem is that we needed a point cloud that we could constantly add data to, circumventing the array restrictions of most game engines. There are several different ways to implement this. We will be using the Unity game engine for the following rendering methods.

1) Single Mesh

Game engines' array restrictions prevent us from directly appending new data to old data, but they do not prevent us from redefining the data entirely, and including the newest data. This can be used to restrict the amount of objects we use to render the scene to only one, and hosting all the data from it. This creates an object with rapidly changing geometry, as more vertices are added to it each second. Every vertex is positioned using the pre-processing algorithm described section 4. This does not scale well because the more data is being rendered, the longer the array is that must be entirely redefined every time new data is available. This linear growth can be seen in Figure 3.

2) Parallelized Meshes

Using a hybrid system of smaller meshes hosted by individual objects, we were able to significantly improve scalability. The premise is to use the drone's location at a given time to generate an object every frame. When that object is generated, it retrieves the current data from the LIDAR and hosts the data to itself. This essentially creates an object whose virtual location is representative of the drone's location at a given time, and whose mesh vertices represent each individual point that was recorded while the drone was at that position. This results in many objects with vertices that resemble a LIDAR scan. In our implementation, about 70 points are hosted per object, which has a considerable impact on performance since 16.67 objects are being generated per second, but has a logithmic growth function. It does not drop below 60 FPS until about 60 seconds of scanning and bottoms out at around 30 FPS, which is sufficient, but still undesirable. This can be seen in Figure 3.

3) Compressed Parellelized Meshes

Compressed parallelized meshes strongly resemble the previous method, but with an additional measure of optimization. Almost all of the load from the previous method comes from the amount of 3D objects required to host all of the vertices, the vertices themselves have very little effect on performance. This can be fixed by hosting more than 70 vertices per object, so that less of them are required for the same data. By recording global data using the method described in equation 2, we are able to use it periodically to generate a new object that hosts thousands of points, and then deleting all the objects hosting identical information. This compression results in far fewer objects required, and results in superior performance and scalability, as seen in figure 3.

B. Dynamic Scan Construction

The performance of this solution can be further improved by combining all of these smaller meshes once the scan is saved. Since a save deems the scan is complete, it is no longer necessary to create new objects or have more than one to represent the most recent changes to the scan. This is done by recording the global coordinates of all vertices throughout the scan, then reconstructing the scan attached to a single object. This allows the scan to still be dynamically constructed during the scan for operator convenience and preserves resources once it is no longer necessary to modify the scan.

VI. Experimental Results and Discussion

In order to evaluate the performance of the mentioned rendering algorithms, we are conducting an experiment that connects a LIDAR, carried by drone, to a Unity environment, where rendering takes place. The data being rendered is supplied from the LIDAR in the form of a point cloud. A Raspberry Pi acts as the drone's micro controller controlling data streams to and from the LIDAR and drone.

A. Experiment Setup and Data Flow

Our networking scheme is built from scratch and uses only the built-in socket functions on each machine, the differing languages are primarily used as a way to access these functions. Our networking scheme is divided into two sections: the Unity server and the raspberry pi micro-controller.

Networking in any game engine is one of the most important features it can offer. Communication speed and integrity are of the highest priority in multiplayer games, which our application essentially is. Unity offers a wide suite of networking features, but the majority of these features only work between two Unity environments, where client and server must be on a machine that is capable of running Unity. In the case of smart cities, almost all devices used for data acquisition do not meet the requirements to run Unity. Our code on both ends: client and server, must be built from basic networking libraries in languages that each device supports. This puts limitations on the code that we use in Unity. We use the basic C# library, which uses OS level sockets no matter what platform the application is running on. When building our project to a Windows, iOS, or Android machine, it will always use OS level sockets to communicate. It follows that our communication format must also be language and platform agnostic, as a variety of IoT devices will be used. One of Unity's best features is the ability to parse a string in JavaScript Object Notation (JSON) format directly into the 3D environment through the use of Game Objects, which are discussed later. It is a safe bet to assume that a given IoT device can send a string over the network, which means it can send a string formatted as a JSON. We do just that. Once the JSON is received through the Unity socket, it is parsed directly into the 3D environment.

For our testing, we used a DJI M100 with a SLAMTEC RPLiDAR A2 mounted on top, producing two-dimensional scans aligned with the drone.

B. Localization of the Drone in a Virtual Environment

Localizing the drone correctly and proportionally in a virtual environment is necessary to properly generate a virtual map. Most 3D environments' coordinate system consists of arbitrary units, so it's important to unify all sensors and hardware to use the same unit. It required considerable post-processing on the host side of the network to convert various inputs with inconsistent formats into meters.

1) GPS Localization

The GPS coordinates in the DJI SDK are represented in geographic radians, which is a simple conversion in geographic degrees. From there, we must use a function of latitude to convert longitude to meters, since the significance of a degree of longitude varies at different latitudes.

$$40075 * \cos(latitude) / 360 \tag{3}$$

The conversion for latitude is simply done with a constant. Once the data is in the correct unit, we can properly represent the horizontal position of the drone using the two values to represent both axes.

2) Orientation of Drone

Knowing the orientation of the drone is required due to the variability of the LIDAR data with respect to the physical drone's orientation. This must be represented in the virtual environment to generate map data along the plane of the drone's current orientation. The orientation of the drone is represented by a compass and a gyroscope. These components need to be mapped to an object in the virtual environment that represents the orientation of the physical drone. The result is a virtual drone that yaws, rolls, or pitches identically to the physical drone.

3) Derivation of Drone's Altitude

The altitude of the drone is measured by barometer, which are not suited for accurate sea level measurement, as fluctuations in air pressure can easily cause tens of meters of error in the measured altitude in the same location when measured at different times. However, we can retrieve the sea level altitude at any global coordinate from an online database and use it as an offset, and use the barometer as a relative measurement. This makes any fluctuations in air pressure limited to the duration of the flight, since the drone can be re-zeroed once it lands in the same spot it took off from.

4) Localization Inaccuracies

The accuracy of our scans is limited by our hardware, primarily our GPS and barometer. With the solution to our barometer inconsistencies resolved, we can now focus on the accuracy problems relating to the GPS when relying on it for centimeter-level precision. The GPS itself is rated to be accurate within a 15 meter radius. The drone uses this GPS only for calibrating itself via satellite, and the precision of the location information that it sends is the result of it using an accelerometer in combination with the GPS

to estimate more precisely the difference even the smallest movements make in the recorded coordinates. This does not affect the local scan quality, but it does affect the accuracy of the recorded coordinates each scan is associated with. This present problems only when multiple adjacent scans are stitched together. They do not always align properly, but they are generally always in the right orientation, and the right relative direction from adjacent scans. This could be improved by using more accurate hardware, or by matching similar sections of the scans, which would require all scans to be done adjacently.

C. Error Breakdown

To further characterize the errors introduced during the aforementioned drone-positioning process, Figure 2 provides a visual representation of all possible sources of error relative to the orientation of the drone, presented from three different views of the drone: top view, rear view, and side view, respectively. The drone has lateral error on all three axes, as well rotational error on all three axes. In addition to this, it has rotational error from the LiDAR, and distance error from the LiDAR's laser. All of these are defined relative to the airframe of the drone and are listed in Table I [11].

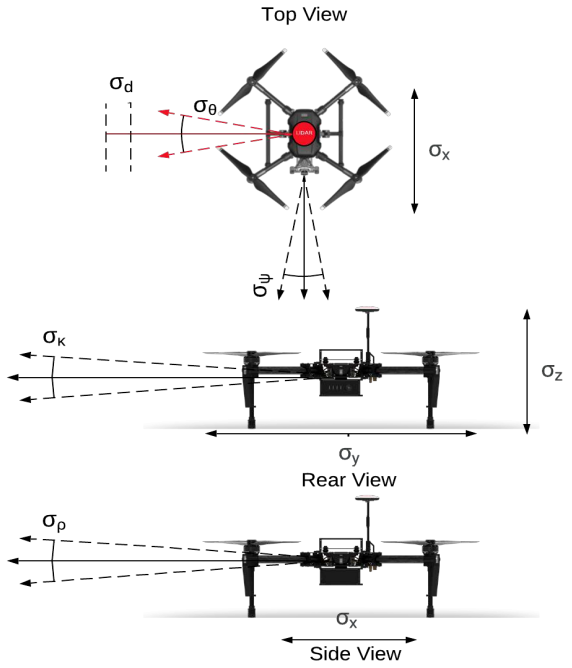

Fig. 2: *Error Components*

$$\begin{bmatrix}\sigma_{tx}\\ \sigma_{ty}\\ \sigma_{tz}\end{bmatrix} = \begin{bmatrix}\sigma_x & w_x & \cos(\theta)\\ \sigma_y & w_y & \sin(\theta)\\ \sigma_z & w_z & r_z\end{bmatrix}\begin{bmatrix}1\\ d\\ \sigma_d\end{bmatrix} \quad (4)$$

TABLE I: *Comprehensive Sources of Positional Error*

GPS Error	σ_x, σ_y
Barometer Error	σ_z
Drone Orientation	ρ (pitch), κ (roll), ψ (yaw)
Orientation Error	σ_ρ (pitch), σ_κ (roll), σ_ψ (yaw)
The angle of LiDAR	θ
LiDAR Angle Error	σ_θ
LiDAR Range Error	σ_d

where

$$w_x \triangleq \sin(\theta)\sin(\sigma_\theta) + \sin(\theta)\sin(\sigma_\phi) + \cos(\theta)\sin(\sigma_\phi) \quad (5)$$

$$w_y \triangleq \cos(\theta)\sin(\sigma_\theta) + \cos(\theta)\sin(\sigma_\phi) + \sin(\theta)\sin(\sigma_\kappa) \quad (6)$$

$$w_z \triangleq \sqrt{(sin(\theta)sin(\sigma_\kappa))^2 + (cos(\theta)sin(\sigma_\rho))^2} \quad (7)$$

$$r_z \triangleq \sqrt{(sin(\theta)sin(\kappa))^2 + (cos(\theta)sin(\rho))^2} \quad (8)$$

Equations (4)-(6) can be described as a breakdown of the significance of certain error sources under certain circumstances. As such, many of the sources of error are amplified or reduced depending on the recorded angle of the scanned point. One linear source of error is distance, as all sources of error except σ_x, σ_y, σ_z, and σ_d are increased by the distance of the scanned point.

The quantification for σ_{tz} is slightly different because the effects of pitch and roll both manipulate the vertical position of a scanned point, while the yaw of the drone, when compensated for in pre-processing, does not.

D. Results and Discussion

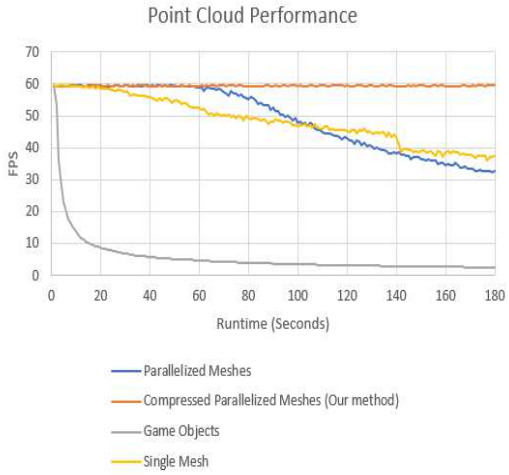

Fig. 3: *Comparison of performance of rendering methods.*

The combination of geographic localization and relative mapping results in a powerful application that can be used to update current geographic databases easily, or construct a new one from scratch. This is due to all scans containing data relating to their real world coordinates, which makes them very easy to locate in a 3D environment.

When we take the real world environment in Figure 5 from Google Earth, and scan it in two separate flight sessions, the

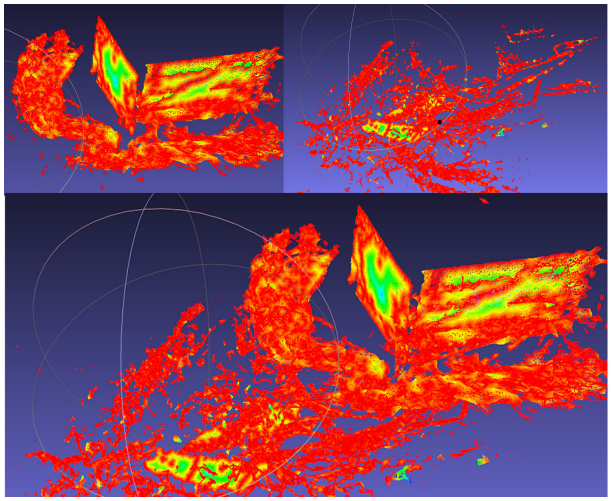

Fig. 4: *The building and the field are scanned separately, but are combined automatically in a 3D environment*

Fig. 5: *Building and nearby field for reference. Coordinates: 35.04276671,-85.29910206*

result is still a single large scan, with a seamless border between the two.

The results shown in Figure 4 very closely represents the distinctive features in the environment, such as the walls of the facility and the trees surrounding it, but it does struggle to represent the ground when scanning the field. This is because our results were recorded with a two-dimensional LiDAR where the horizontal configuration of the LiDAR made it difficult to scan the ground. This resulted in a low point density on areas defining the ground, and during post-processing, the density was not high enough to produce a distinctive plane. While some areas of the scan could be improved, the project shows promising application for real time mapping with respect to previous data.

The quality and density of the scan could be improved greatly by using a three-dimensional LIDAR, which would also help scan the ground, since the drone would not have to perform aggressive maneuvers to point the sensor at the ground. However, the concept and functionality can be applied to virtually any scanning hardware.

VII. Conclusions

In this paper, we have proposed both a networking paradigm and a unique algorithm for rendering real time dynamic meshes in a game engine, both key features in using a game engine for the modeling of a smart city. Experimental implementations have shown FPS increases that are 25% higher for a runtime of 120 seconds and 33% higher for a runtime of 180 seconds than the current methods implemented by Unity, an industry leader in game engine design. This paper brings to light a fundamental issue underlying the implementation of mesh rendering in game engines, specifically for dynamic meshes that change according to a real time stream.

Acknowledgement

The authors would like to thank Dakila Ledesma and Maxwell Omwenga for their helpful discussion and suggestions.

References

[1] C. Andrews, "Gamification in gis and aec," [online] Available at: https://www.esri.com/arcgis-blog/products/arcgis/3d-gis/gamification-in-gis-and-aec/ [Accessed on May 30, 2020], Feb. 2020.

[2] C. Rusu, "Ar, vr, gamification: cutting-edge technologies applied in smart cities," [online] Available at: http://citisim.org/ar-vr-gamification-cutting-edge-technologies-applied-in-smart-cities/ [Accessed on May 30, 2020], 2018.

[3] Maps SDK for Unity Overview - Google Maps Platform Gaming Solution [Online]. Available: https://developers.google.com/maps/documentation/gaming/overview_musk [Accessed on May 30, 2020].

[4] Maps for Unity. [online] Available at: https://www.mapbox.com/unity/ [Accessed on May 30, 2020].

[5] Z. J. Chong, B. Qin, T. Bandyopadhyay, M. H. Ang, E. Frazzoli, and D. Rus, "Synthetic 2D LIDAR for precise vehicle localization in 3D urban environment," in *Proc. IEEE International Conference on Robotics and Automation*, Karlsruhe, Germany, May 2016.

[6] ——, "Mapping with synthetic 2D LIDAR in 3D urban environment," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, Tokyo, Japan, Nov. 2013.

[7] I. Toroslu and M. Doğan, "Effective sensor fusion of a mobile robot for SLAM implementation," in *Proc. The 4th International Conference on Control, Automation and Robotics*, Auckland, New Zealand, April 2018.

[8] P. Greenwood, J. Sago, S. Richmond, and V. Chau, "Using game engine technology to create real-time interactive environments to assist in planning and visual assessment for infrastructure," in *Proc. International Congress on Modelling and Simulation*, Cairns, Australia, July 2009.

[9] P. Agrawal, A. Iqbal, B. Russell, M. K. Hazrati, V. Kashyap, and F. Akhbari, "PCE-SLAM: A real-time simultaneous localization and mapping using LiDAR data," in *Proc. IEEE Intelligent Vehicles Symposium (IV)*, Los Angeles, CA, Jun. 2017.

[10] B. Klein, "Managing the scalability of visual exploration using game engines to analyse UHI scenarios," *Procedia Engineering*, vol. 169, pp. 272–279, 2016.

[11] N. May and C. Toth, "Point positioning accuracy of airborne LiDAR systems: A rigorous analysis," in *Proc. Photogrammetric Image Analysis*, Munich, Germany, Sept. 2007.