3D Modeling of Cities for Virtual Environments

Calvin Davis\textsuperscript{1}, Jared Collins\textsuperscript{1}, Joshua Fraser\textsuperscript{1}, Haoxiang Zhang\textsuperscript{1}, Shizeng Yao\textsuperscript{1}, Emily Lattanzio\textsuperscript{3}, Bimal Balakrishnan\textsuperscript{2}, Ye Duan\textsuperscript{1}, Prasad Calyam\textsuperscript{1}, Kannappan Palaniappan\textsuperscript{1}

\textsuperscript{1}Dept. of Electrical Engineering and Computer Science, \textsuperscript{2}Architectural Studies, University of Missouri, Columbia, MO \textsuperscript{3}High Point University, High Point, NC

Abstract—Modeling and simulation of large urban regions is beneficial for a range of applications including intelligent transportation, smart cities, infrastructure planning, and training artificial intelligence for autonomous navigation systems including ground vehicles and aerial drones. Immersive environments including virtual reality (VR), augmented reality (AR), mixed reality (MR or XR) can be used to explore city scale regions for planning, design, training and operations. Virtual environments are in the midst of rapid change as innovations in display technologies, graphics processors and game engine software present new opportunities for incorporating modeling and simulation into engineering workflows. Game engine software like Unity with photorealistic rendering and realistic physics have plug-in support for a variety of virtual environments and typically model the scene as meshes. In this paper, we develop an end-to-end workflow for creating urban scale real world accurate synthetic environments that can be visualized in virtual environments including the Microsoft Hololens head mounted display or the CAVE VR for multi-user interaction. Four meshing algorithms are evaluated for representation accuracy and city-scale meshes imported into Unity for assessing the quality of the immersive experience.

Index Terms—3D Reconstruction, Mixed Reality, Synthetic Environment, Pipeline, Meshes, Texture Mapping, Point Cloud

I. INTRODUCTION

Mixed reality (MR) has significantly influenced technology by immersing users in synthetic virtual environments, providing capabilities such as communication [1] and autonomous vehicle testing [2]. Our work focuses on utilizing the immense capabilities of MR with 3D modeling to display cities as meshes in MR devices. This paper explores an effective pipeline with publicly available software for photo-realistic mesh reconstruction from a point cloud, its challenges, and its use cases on a city scale.

The current work that has been done in this field either does not address the entirety of a city-to-synthetic environment pipeline or has been intended for a different application and would not work well with city-scale data. In addition, we experimentally compare three different four different meshing algorithms, available in free software that we use, to determine which is the most suitable for 3D modelling of urban cities.

In this paper, a pipeline is presented to view 3D point clouds of cities in MR. Realistic synthetic environments can be useful in many areas, such as drone flight planning, flight training for aircraft, city planning, car safety research in real-time traffic simulations, and video games [1]. We provide a method for generating city-scale meshes using free software, and offer different recommendations for tuning parameters depending on the intended application.

II. RELAT ED WORK AND SIMILAR PIPELINES

Maiti and Chakravarty [3] study surface reconstruction using point clouds resulting from feature-based photogrammetry. They evaluate both Poisson surface reconstruction [4] and the Ball-Pivoting Algorithm (BPA) [5] with a discussion on tuning algorithm parameters and demonstrating the impact these parameters have on the resulting watertight mesh.

Bosch et al. created an open source ground-truth and metric evaluation pipeline for urban areas sourced from commercial satellite imagery along with benchmark datasets [6]. They establish evaluation metrics for photogrammetric point cloud accuracy to LiDAR, horizontal accuracy to public mapping vector products, semantic labeling, volumetric accuracy, curvature and roughness, and triangle mesh model simplicity.

Poulis and You directly use LiDAR for city reconstruction [7]. Gaussian distributions are used to segment similar regions in a nadir view of a city to find roofs. Then, boundaries are extracted from the segmentation results. Planes are fitted to the boundaries and then extruded downward, producing a watertight mesh. For texture mapping, available images are matched to scenes of the extruded 3D mesh and bundle adjusted. Texture coordinates are found by projecting the mesh’s triangles into all images and using the image with the highest projected area.

Kuschk shows that city mesh reconstructions can be produced by connecting neighboring points of a created digital surface model [8]. First, bundle adjusted camera poses and corresponding images are used for dense stereo reconstruction to obtain height information. Then, in the digital surface model, 4-connected neighborhoods of pixels are naïvely connected into two triangles. Since the resulting mesh has too many polygons, two simplifications are performed: plane fitting to remove redundant vertices and removal of nearly colinear triangles. After simplification, the texturing method projects all triangles to all the images, choosing the image with the highest projected area.

These methods either do not address city-scale data, or do not provide source code. The largest item Bosch et al. [6] reconstructs is an outdoor staircase with few steps. Poulis and You [7] use city-scale LiDAR instead of images to create a mesh, but we show a dense reconstruction from images is enough. Finally, Kuschk [8] achieves impressive results using multi-view stereo and images, but does not provide source...
code. We show that city-scale point clouds can be created using free software such as CloudCompare and MeshLab, and we detail our approach so that our results can be reproduced.

III. METHODS

This section details our pipeline for modeling and visualization of urban environments from 3D point clouds as shown in Figure 1. First a triangle mesh is generated from a city-scale point cloud using freely available software packages, then our custom high-resolution texturing algorithm is applied, and finally we import our results into Unity to create synthetic environments for HoloLens and CAVE.

A. Mesh Generation

We begin with dense 3D point clouds of two urban areas Columbia, MO and Albuquerque, NM created using the VB3D aerial multiview stereo algorithm [9]. Both city point clouds are roughly on a 1 meter grid. The aerial imagery are from a high resolution metric camera [10] with the onboard exterior camera orientations refined using the fast BA4S bundle adjustment algorithm [11]. Additionally, LiDAR is available for Columbia, MO as an additional source of point data for algorithm comparison and evaluation.

To generate high quality meshes suitable for an MR environment we evaluate the following meshing algorithms with freely available implementations: ball-pivoting, greedy triangulation, Poisson and screened Poisson.

The ball-pivoting algorithm (BPA) uses a ball of fixed radius to traverse a point cloud creating triangles when three distinct points are in contact with the ball, as the ball pivots around edges created during the previous step(s). Holes resulting from irregularly sampled surfaces are filled by iteratively running BPA with increasing radii [5].

The greedy triangulation mesh generation method [12] incrementally adds edges that never cross one another, so edges are never deleted. Each point is assigned a fixed number of neighbors in a sphere determined by local density. A weighted least squares plane fitted to points in the neighborhood is used to estimate the surface normal. Points are pruned based on visibility, connected to the original point, and consecutively connected to each other to form triangles. Maximum and minimum angle criteria are used to reduce the smoothing of corners. This method focuses on fast triangulation so that it can be used in real-time applications where speed and robustness to noise is important.

Poisson surface reconstruction formulates mesh reconstruction as a Poisson problem resulting in a global solution producing smooth surfaces that are robust to noise [4]. The surface is extracted from oriented point samples by first calculating an inside-outside indicator function discretized to an octree then extracted using the marching cubes algorithm. Once the marching cube traverses to the desired octree depth, the triangulated 3D mesh is created by interpolating the points between the cube vertices.

The Screened Poisson surface reconstruction method adds point constraints as a regularization term to reduce the original method’s tendency to over smooth [13]. The point constraints are key points that should be preserved and included in the final surface.

We use the surface reconstruction algorithms from the following freely available software packages: the Ball-Pivoting Algorithm (BPA) [5] and Screened Poisson [13] available in MeshLab [14], Poisson Surface Reconstruction [4] in CloudCompare [15], Greedy Surface Triangulation [16] in Point Cloud Library (PCL) [16], and Poisson Reconstruction with Delaunay Triangulation [17] in the Computational Geometry Algorithms Library (CGAL) [18].

B. Tuning Mesh Generation

Each surface reconstruction algorithm provides parameters that tune its robustness to noise, holes, and non-uniform sampling. Tuning these parameters not only affects the quality of the resulting surface, but also significantly affects the compute time and memory usage.

The Ball-Pivoting Algorithm (BPA) [5] is tuned by the ball radius, clustering radius, and angle threshold parameters. The ball radius parameter controls the size of the ball that traverses the point cloud, where a small radius will be sensitive to noise due to including most points in the mesh, and a large radius pivoting over noisy points will result in a smoother surface at the cost of finer details. Clustering radius filters out points that lie too close together in a neighbor as a percentage of the ball radius. Because clustering is defined as a percentage of the ball radius, it is important to note that these are not independent
parameters. Angle threshold stops the ball traversal at an edge that would require a pivot angle greater than the threshold; for example this should be $\leq 90$ otherwise the ball will also roll along the underside of the surface.

The Greedy Surface Triangulation [12] uses maximum number of neighbors, farthest neighbor maximum edge length, minimum and maximum angles, and maximum surface angle. Defining a maximum number of neighbors and the farthest possible neighbor filters local influence since a plane is estimated from the neighborhood. The maximum edge length, minimum angle, and maximum angle limit resulting triangles, but the minimum angle might not be honored. The maximum surface angle determines the maximum angle between two triangle normals, and an optional consistency check is available to ensure adjacent normals lie on the same side of the surface.

The Poisson algorithm [4] controls the voxel resolution by setting the maximum depth of the octree for surface reconstruction. Increasing this depth results in higher-resolution triangle meshes. Samples per node is the minimum number of points that should lie in a node during octree construction as it adapts to sampling density. Five or fewer samples per node may be suitable in the absence of noise, while twenty or more may be required to smooth noisy data.

The Screened Poisson algorithm [13] reduces the tendency to over-smooth with an additional interpolation weight parameter. A lower interpolation weight puts more importance on fitting the gradients, while a higher value constrains the mesh more to the points.

C. Evaluation of Meshing Algorithms

Figure 3 shows a few results of our meshing. To find the best mesh with BPA and greedy triangulation, a grid search was performed with their many parameters. Figure 6 shows a few BPA results for Boone County Courthouse in Columbia, MO, pictured in 5. Clustering radius is, to the best of our knowledge, only documented within the implemented method in MeshLab. It can be used to be more robust to noise. Angle threshold is mostly fixed to 90 for buildings, and the ball radii should be picked to conform to the underlying data.

CloudCompare’s default Cloud-to-Mesh Distance function [19] is used to compute the distances between the point cloud and mesh, shown in Eqs 1 and 2. More specifically, let $M$ be the (triangulated) mesh generated from the point cloud, and $G$ be the ground truth LiDAR 3D point cloud. For one 3D LiDAR point $g \in G$, its distance to the closest polygon/triangle in the mesh is defined as,

$$r_{g \rightarrow M} = \min_{m \in M} |g - m| \quad (1)$$

These distances can be aggregated to define the precision of the reconstructed mesh $M$:

$$P(M, G) = \frac{\sum_{g \in G} (r_{g \rightarrow M})}{\text{size}(G)} \quad (2)$$

D. Texture Mapping Meshes Using View Selection

The resulting polygon meshes contain only per-vertex color retained from the source point clouds. Achieving photo-realism requires applying high-resolution textures to our model. Rather than create a single texture atlas and UV parameterization, we divide the mesh into triangle sets texture mapped to one of a small set of the original high-resolution aerial images evenly spaced about a circular orbit of our scene. We use the associated bundle adjusted camera poses for each image and face normals to choose the view to use as a texture for each triangle. One triangle set is created for each texture image and triangles are assigned to a set based on the smallest angle between the camera view vector and the face normal as shown in Algorithm 1.

```
Algorithm 1 Match Faces to Views for Texture Mapping
1: Input
2: $M$ mesh with faces in winding order
3: $n$ number of image views to texture from
4: $Z_n$ array of camera look-at directions, size $n$
5: Output
6: $R_n$ list of meshes with removed faces
7: procedure REMOVEFACES($M$, $n$, $Z_n$)
8: initialize $R_n = M$, $\forall n$
9: for all face $f \in M$ do
10: $v_0, v_1, v_2 \leftarrow$ vertex 0, 1, 2 $\in f$
11: $u \leftarrow v_1 - v_0$
12: $v \leftarrow v_2 - v_0$
13: $p \leftarrow u \times v / |u \times v|$
14: $i \leftarrow \text{argmin} \{p \cdot z, z \in Z_n\}$
15: delete $f$ from $R_n$, $n \neq i$
16: end for
17: end procedure
```

After assigning each triangle to an associated image set, the face must be UV parameterized to the image for texture display. Generating UVs for each triangle only requires projecting vertex coordinates to the image plane, normalizing image plane coordinates, and vertically flipping to match texture origin conventions as shown in Algorithm 2. The triangles with UV texture coordinates are written to a mesh file associated with the image used for texturing resulting in
Fig. 3: Visual comparison of VB3D meshes with VB3D$_{pc}$ point cloud for ABQ dataset (see Figure 2 for an aerial image from the same viewpoint). The grid shows meshing results for the full ABQ scene with errors between 0 to 1 meter shown in gray and errors more than 1 meter colored using a jetmap: blue colors are lower errors, while red colors are higher errors. Figure 3a is a cropped mesh of ABQ for BPA since MeshLab crashes on high volume data when running the method.
a separate mesh for each of the images used in Algorithms 1 and 2. We use MeshLab to merge the separate point clouds into a single Stanford polygon format (.ply) file with texture references. Figure 7 shows the result of texture mapping using eight images. Texture mapping could be improved with depth testing to remedy incorrectly textured triangles resulting from occlusions using a z-buffer algorithm [20].

E. Unity Game Engine and Mixed Reality Plugins

We use the Unity game engine as the rendering backend due to its ubiquitous support on many platforms and mixed reality devices. However, Unity is not an open source platform possibly limiting its use. The city-scale texture mapping applied in the previous section results in several image-associated face sets exported as separate meshes. These individual non-overlapping but adjoined meshes are imported into Unity with their images applied as textures to create a cohesive model of the urban environment. Figure 8 shows a generated mesh using our pipeline inside of Unity.

We target two mixed reality platforms: Hololens 1 and CAVE. To use the Hololens 1, Microsoft’s Mixed Reality toolkit is downloaded and ‘3D’ is selected as the Unity project. Although the plugin sets up the Unity environment, there are many steps to deploy the application to HoloLens. In the build settings, the Universal Windows Platform is set to a D3D project with x86 architecture and default compression.

Algorithm 2 Generate Texture Map Coordinates for Triangles

Input
1: $M$ mesh with faces removed
2: $P$ camera matrix associated with desired image
3: $w$ image width
4: $h$ image height

Output
5: $T$ textured mesh coordinates

for all face $f \in M$
7: for all vertex $v \in f$
8: convert $v$ to homogeneous
9: $(u, v) \leftarrow P v$, project vertex to image
10: $u \leftarrow u/w$, normalize
11: $v \leftarrow v/(1 - (v/h))$, normalize and flip

end for
end for
The platform is then to Universal Windows. To interact with the mesh hologram we add interaction profiles to the project, which include TaptOPlace and BoundsControl, which allows clicking to move and scaling of the hologram, respectively. A step-by-step guide on the rest of the process is provided by Microsoft [21].

MiddleVR 2.0 and UniCAVE 2.0 are the plugins used in the CAVE experiments. These plugins make it convenient for users to set up the VR environment, connect VR headset and wands, and customize the scripts to control the wand event. MiddleVR is a Windows-only plugin that supports Unity projects running under multi-display, stereoscopy, and VR systems. It configures the connection between VR equipment and VR systems such as the CAVE environment and joystick. MiddleVR supports multi-PC, multi-GPUs, and multiple projectors suitable for a wide variety of projects. MiddleVR has many built-in time-saving scripts allowing users to just add the components to the game object and customize parameters. Although MiddleVR has many advantages for interfacing Unity with the CAVE VR environment, it is proprietary and expensive.

UniCAVE is an open source plugin for Unity made by University of Wisconsin-Madison. UniCAVE does not have many built-in functions, so users have to write their own code to implement. Compared to MiddleVR, UniCAVE does not automatically set up the connection between the VR equipment and environment. Configuring the unity scene for the University of Missouri’s CAVE is involved, so we provide a tutorial [22].

IV. RESULTS

We first generate the highest quality mesh we can achieve on a single building, the Boone County Courthouse, with each method, shown in Figures 9, 10, 11, 12. After finding the best parameters, they are tuned on a whole city, Albuquerque, New Mexico. Both meshes are compared to its point cloud by computing the nearest points to mesh distances.

A. Ball-pivoting Algorithm

Figure 9 shows the overview of results with BPA. The authors recommend running successively with increasing radii.
(a) Best greedy triangulation result on a single building (with PCL).

(b) Distances between point cloud and best greedy triangulation result on a single building (with PCL). $\mu = 0.005, \sigma = 0.123$

(c) Best greedy triangulation result on a whole city (with PCL).

(d) Distances between point cloud and best greedy triangulation result on a whole city (with PCL). $\mu = 0.009, \sigma = 0.137$

Fig. 10: Greedy triangulation algorithm with PCL overview. For subfigures 10a and 10c, the color gradient corresponds to the relative height. Subfigures 10b and 10d have colors tied to the distance between a point and the best mesh generated. Colors are gray from 0 to 1, and then a jet gradient is used from 1 to 4, where 4 is fully saturated as red. The more gray, the better.

to help close holes and reduce noise in the final mesh [5]. However, the MeshLab implementation of BPA has a known bug that prevents it from running multiple times without crashing. Additionally, MeshLab crashes when running BPA on large point clouds, in our case, about 1.5 million points. It may appear to run fine, but in our experience, it can run over 12 hours and then crash before it finalizes. To ameliorate these issues, we run separate radii on clones of the input point cloud and render them together. For our whole city result, we crop a small portion of the downtown area with 123,640 points.

Creating a mesh from the courthouse required runs with three radii: $\sqrt{2}$, 2, and 3. Angle threshold is set to 90, which is the default. Clustering radius, expressed as a percent of the main diagonal in world, is set to 0. Also note that this parameter, to the best of our knowledge, is not documented in any literature, including the original paper, and it is only mentioned in the MeshLab documentation for the algorithm. Points are mostly uniformly distributed in a grid of 1 meter units, so the smallest possible triangles are made first with the radius equal to $\sqrt{2}$. The other two radii are used to close holes.

Figure 9a shows our final courthouse mesh. One large hole is noticeable and can’t be closed with a larger radius. Our quantitative measurements on the courthouse and Albuquerque crop show a low mean and standard deviation, mostly due to points that were not reachable by the ball, so triangles were not created with them.

B. Greedy Triangulation

Our greedy triangulation result summary is shown in Figure 10. This method appears to produce the least quantitative errors between the other methods. Its mesh depends mostly on the points it considers at each step, which can be controlled by a max distance, max number of points, and max edge length. Experimentally, we also find that normal estimation with the same search radius as greedy triangulation itself within PCL is important to create good results, so any previously estimated normals should be overwritten. We set mu to 3, search radius for both normal estimation and meshing to 8, and 100 max neighbors. The max surface angle, max angle, min angle, and normal consistency are set to 45 degrees, 10 degrees, 120 degrees, and false, respectively.

PCL has a known bug during greedy triangulation where it flips the orientation of normals. If this is an issue, normals can
Fig. 11: Poisson reconstruction with CloudCompare overview. For subfigures 11a and 11c, the color gradient corresponds to the relative height. Subfigures 11b and 11d have colors tied to the distance between a point and the best mesh generated. Colors are gray from 0 to 1, and then a jet gradient is used from 1 to 4, where 4 is fully saturated as red. The more gray, the better.

C. Poisson Reconstruction

Our Poisson reconstruction results are summarized in Figure 11. The original implementation has few parameters in CloudCompare. Of note, point weight needs to be set to 0 to run as non-screened. The maximum octree depth is set to 12, boundary is set to Neumann, and samples per node is set to 3 since there is relatively little noise in our input data. Increasing the sample per node will yield smoother bases and reduced height for each building.

The resulting meshes have the highest point to mesh distance errors. Figures 11a and 11c appear good at first glance, but fail to meet the height of the input point cloud. Figure 11a may appear sharper than its screened counterpart in Figure 12a, but the non-screened result has higher error because its faces are further from its input point cloud. The tallest buildings in Figure 11c end early, which is more clearly shown in red in Figure 11d.

D. Screened Poisson Reconstruction

Figure 12 shows our best screened Poisson results. We increase the samples per node to 10, since a lower value increases the number of pits generated in the final mesh. Point weight, the parameter that causes it to be screened, is set to 0.5, which is enough to better fit the points in the input point cloud. The tops of buildings are no longer removed and the overall city mesh has much lower distances between its input and faces.

V. Conclusion

Meshing a city-scale point cloud produces challenges that are hard to overcome with a single algorithm. We test four popular methods and evaluate them for guiding the process for others in their different applications. We use mostly uniformly sampled point clouds, but there are parameters for each method that can help with noise.
(a) Best screened Poisson reconstruction result on a single building (with CloudCompare).
(b) Distances between point cloud and best screened Poisson reconstruction result on a single building (with MeshLab). \( \mu = 0.490, \sigma = 0.564 \)
(c) Best screened Poisson reconstruction result on a whole city (with MeshLab).
(d) Distances between point cloud and best screened Poisson reconstruction result on a whole city (with MeshLab). \( \mu = 0.357, \sigma = 0.409 \)

Fig. 12: Screened Poisson reconstruction with CloudCompare overview. For subfigures 12a and 12c, the color gradient corresponds to the relative height. Subfigures 12b and 12d have colors tied to the distance between a point and the best mesh generated. Colors are gray from 0 to 1, and then a jet gradient is used from 1 to 4, where 4 is fully saturated as red. The more gray, the better.

For applications that require the most accurate and true to input meshes, such as flight or building planning, BPA or greedy triangulation is recommended. Faces will be produced with existing vertices, with the only error coming from points that are not used in creating faces. Running BPA with successive and increasing radii can produce the least amount of holes, but for large data, its implementation in MeshLab is lacking, so we recommend looking elsewhere. Greedy triangulation can produce the least noisy city-scale meshes due to its ability to use many more points than BPA, but it is prone to producing more holes.

If the intended application needs a watertight mesh, such as visualization in MR devices like the CAVE 13, then screened Poisson reconstruction is recommended. A high enough point weight can ensure that faces are produced mostly by the input point cloud. An overall smoothed result while adhering to points can still be produced by increasing the samples per node. Additionally, the boundary should be set to Neumann if the input is not a closed volume, and Dirichlet otherwise. To the best of our knowledge, the third option—free—produces the

Fig. 13: Lab Members inside of CAVE at University of Missouri
same meshes as a Neumann boundary. Non-screened Poisson is not generally recommended, unless the application requires a mesh that is especially smooth. It is possible to get balloon-like structures using a Poisson method, and we conclude that they are produced by inaccurate normals, since the method itself is dependent on normal accuracy. If balloons are produced, we recommend re-estimating normals.

We also demonstrate a pipeline to transform point clouds into an immersive environment using common surface meshing algorithms and custom texture mapping to explore the possibilities this emerging technology enables. Highly detailed city scale reconstructions are challenging due to sample density, noise, and missing data resulting from occlusions. All the presented meshing algorithms produce suitable results with carefully selected parameters as demonstrated in our experimental results. We address the demands of increasing immersiveness for city scale environments by creating high fidelity textures congruent with our high quality meshes through a custom algorithm. Finally, we examine and provide details for visualizing our results in mixed reality environments.

ACKNOWLEDGMENTS

Emily Lattanzio (High Point University) and Calvin Davis (University of Missouri) were supported by the NSF REU program at the University of Missouri EECS Department. The research was partially supported by the National Science Foundation under awards CNS-1950873 (REU) and CNS-2018850 (MRI) and U.S. Army Research Laboratory W911NF-1820285. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the U.S. Government or agency thereof.

REFERENCES