# Real-Time Re-Textured Geometry Modeling Using Microsoft HoloLens

Samuel Dong* Tobias Höllerer†

University of California, Santa Barbara

**ABSTRACT**

We implemented live-textured geometry model creation with immediate coverage feedback visualizations in AR on the Microsoft HoloLens. A user walking and looking around a physical space can create a textured model of the space, ready for remote exploration and AR collaboration. Out of the box, a HoloLens builds a triangle mesh of the environment while scanning and being tracked in a new environment. The mesh contains vertices, triangles, and normals, but not color. We take the video stream from the color camera and use it to color a UV texture to be mapped to the mesh. Due to the limited graphics memory of the HoloLens, we use a fixed-size texture. Since the mesh generation dynamically changes in real time, we use an adaptive mapping scheme that evenly distributes every triangle of the dynamic mesh onto the fixed-size texture and adapts to new geometry without compromising existing color data. Occlusion is also considered. The user can walk around their environment and continuously fill in the texture while growing the mesh in real-time. We describe our texture generation algorithm and illustrate benefits and limitations of our system with example modeling sessions. Having first-person immediate AR feedback on the quality of modeled physical infrastructure, both in terms of mesh resolution and texture quality, helps the creation of high-quality colored meshes with this standalone wireless device and a fixed memory footprint in real-time.

**Index Terms:** Computing methodologies—Computer graphics—Graphics systems and interfaces—Mixed / augmented reality; Computing methodologies—Computer graphics—Image manipulation—Texturing

## 1 INTRODUCTION

Microsoft released the HoloLens, a standalone augmented reality headset with a stereoscopic 3D display, in early 2016. Equipped with an infrared depth sensor and an inertial measurement unit, the HoloLens continuously builds up a model of the environment it is in, allowing objects that were positioned to remain fixed in the location no matter how far or where the user moves to. Due to its standalone nature, it is meant to be worn as the user walks around their environment, expanding the stored model as they go. It splits its processing into three units: the CPU, GPU, and HPU, or holographic processing unit. This allows the HoloLens to dedicate hardware to continuously track and model the environment while also supporting intensive 3D graphical applications.

The environment is scanned using the infrared laser and sensor, generating an internal point cloud. The HoloLens automatically processes the point cloud and converts it into a triangular mesh. This mesh is used in a variety of manners, ranging from showing a 3D cursor to providing anchor points for 3D objects. The mesh is also commonly used to provide occlusion for virtual objects, making them feel truly positioned in the real space. Games can also use this information to build a custom experience based on the structure of the surrounding environment, such as rendering decals or choosing where units spawn.

While the HoloLens does scan the geometry of the environment, it does not automatically apply color detail at all. The mesh only contains positions and normals. This prevents the out-of-the-box HoloLens from being used for 3D scanning and other applications that use mesh color and/or texture for exporting or rendering. Examples of potential uses of color include scanning an object with surface texture and rendering virtual objects with real-world reflections and ambient color.

In this paper, we propose a real-time solution using the HoloLens's built-in camera. Because the HoloLens is a device with limited memory, we did not want to have a memory requirement that scales with the size of the environment, so we utilize a fixed-size texture. This texture operates as a standard UV texture map, making exporting and integrating into existing applications very straightforward. We show the results of modeling both a small and large environment to show the effects of the adaptive texture and consider the time required to capture these environments and the quality of the outcome.

## 2 RELATED WORK

Interactively rendering realistic image-based models of the real environment has been a research topic for a while. Early research focused on using image-based techniques to extract the geometric data and provide a virtual model. With the advent of depth cameras and laser scanning, and the popularity of structure-from-motion techniques [5], acquiring geometric data is extremely practical. Color data can be acquired by taking pictures of the geometry from various angles, and the camera position and orientation is sometimes available as well.

Newcombe et al.'s KinectFusion [12] shows building a dense geometric model of interior environments in real-time using an inexpensive depth sensor. This has been improved upon in Whelan et al.'s Kintinuous paper [16], allowing for growing environments and conversion to a triangular mesh.

Once a model has been built, image-based rendering still needs to be done to make them look realistic. Debevec et al. [4] improved upon their previous view-dependent image-based rendering technique by introducing a real-time method for rendering geometric data from a set of images and poses. In their paper, the best images were chosen by the current view angle and projected onto the geometry. Because projective texturing colors geometry no matter its visibility, the occlusion was dealt with by clipping the geometry with occluding volumes. Buehler et al. [2] introduced unstructured lumigraph rendering, a combination of view-dependent image-based rendering with light field techniques to support arbitrary view locations and directions. It factors in the resolution of each view and creates a blend of the best views based on their quality. Chen et al. [3] uses KinectFusion to acquire a high-quality mesh from the scene along with image frames. After some offline processing, view-dependent texture projection is used to render the scene while factoring in occlusion. Care is taken to balance smooth transitions by blending many frames and preserving view-dependent features, such as specular highlights and reflections, by selecting fewer frames. These methods render the model in real-time with highly accurate

*e-mail: samuel_dong@umail.ucsb.edu

†e-mail: holl@cs.ucsb.edu

2018 IEEE Conference on Virtual Reality and 3D User Interfaces
18-22 March, Reutlingen, Germany
978-1-5386-3365-6/18/$31.00 ©2018 IEEE

results, but they rely on a provided model and a dense set of images.

If the real-time constraint is lifted, further optimizations for texture quality can be performed. Bernardini et al. [1] used both the depth and color information when blending together multiple views based on a weight calculated on the confidence of individual pixels. Views were also aligned by utilizing features detected in the color information to prevent ghosting. Similarly, Zhou and Koltun [17] reduced artifacts by optimizing the camera location and orientation and image distortion based on photometric consistency. This allows for better texturing when geometric errors or lens distortion cause misalignment. Maier et al. [11], on the other hand, improve visual quality by merging multiple lower resolution color frames into higher resolution ones with some filtering. This allows the texture to have higher resolution data for an object than what is provided by the original RGB camera frame.

Using RGB-D (color and depth) information, Fechteler et al. [7] were able to texture a provided articulated model with the Kinect. The articulated mesh provides UV coordinates for a texture file. The images come from a continuous stream of RGB data. Because not all triangles are visible in any one frame, a set of frames are kept based on their angle and resolution. Each triangle references only one frame, and frames that do not provide much data are discarded to save memory.

The Kinect has also been used to generate both color and geometry data. By using RANSAC and feature extraction to align views, Henry et al. [8] use frame to frame RGB-D data to construct a sparse surface patch array. It also detects loops and closes them. Their method runs in real time and provides a compact representation of indoor environments, but must store a set of keyframes for loop detection. Instead of using image features, Kerl et al. [9] use photometric error between all pixels in subsequent frames to provide higher quality pose estimation. By selecting keyframes and closing loops, accumulated drift between frames is minimized. Steinbrücker et al. [15] extend on that paper to use the tracking data to merge depth and color into a sparse octree data structure for compact storage of this data. The octree allows for real-time reconstruction and viewing of both the geometry and color information by marching through the tree and rendering the voxels to create the image.

An alternative to dense reconstructions of the environment is the use of predefined 3D models. By maintaining a database of common objects, a scene can be represented by a set of these objects, removing sensor artifacts and providing semantic labeling. Sankar and Seitz [14] use interaction with the user to guide a program in placing 3D models of objects. Salas-Moreno et al. [13] extend the simultaneous localization and mapping (SLAM) with these predefined objects, allowing for both automatic detection and placement of the objects, as well as more efficient pose estimation. Dou and Fuchs [6] utilize several RGB-D sensors mounted on walls in a room in an effort to provide an extremely robust model of potentially dynamic scenes. By utilizing both existing 3D models for rigid and static geometry and reconstructed models for dynamic geometry, higher quality scene reconstruction is possible for potentially dynamic environments.

The Microsoft HoloLens provides the geometric data and modeling in real-time automatically. Our work focuses on using a video stream augmented with pose information to color this dynamic mesh in real-time. For easy integration, we use a UV texture approach, which is almost universally accepted by modeling and model viewing programs and simple to render. We also maintain a fixed-memory footprint, storing only the latest frame of the camera stream.

## 3 Methodology

The HoloLens API provides its geometric data in the form of a set of meshes that are identified by a GUID. Each mesh contains a position buffer, a normal buffer, and an index buffer. We keep the set of meshes stored in a dictionary using the GUID as a key. To UV texture the set of meshes, UV coordinates must be provided for each vertex in every mesh. While there are many automated methods of unwrapping a 3D mesh onto a UV map, we chose a simple, efficient method to cope with regular changes in geometry as the HoloLens is building up the environment.

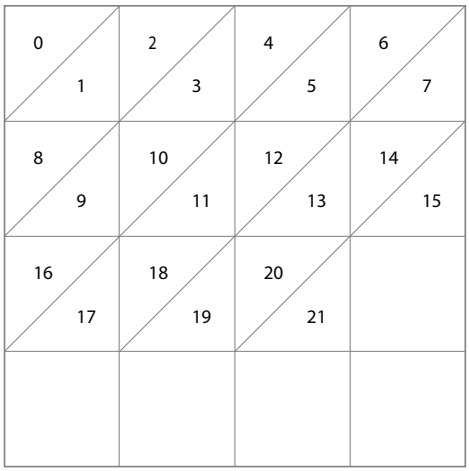


Figure 1: Example layout with 22 triangles. The numbers show where the triangles with specific global indices would be mapped to. The size is 4, which holds a maximum of 32 triangles.

### 3.1 Mapping Triangles to a Fixed-Size Texture

Our method first assigns each triangle in the set of meshes a global index. We iterate over the dictionary in order by GUID, which ensures a consistent ordering of the meshes. Each mesh has an internal ordering of triangles. Since we need to index each triangle globally with respect to the whole set of meshes, an offset must be calculated to apply to the internal ordering. The offset of a mesh is calculated as the sum of the number of triangles of every mesh before it. Our global index is then defined as the offset plus the internal triangle index.

Given a fixed-size square texture, we divide it equally into squares based on the total number of triangles that need to fit in it. By using two triangles to form one square, the number of squares on one side can be computed as:

$$numberOfSquares = \left\lceil \sqrt{\frac{totalNumberOfTriangles}{2}} \right\rceil \quad (1)$$

We will refer to the number of squares on one side as the size of the layout. Given the global index and the size, one can calculate the triangular region reserved for that specific triangle by first computing which square the triangle lies in, then which half of the square the triangle is mapped to based on the index being even or odd. An example is shown in Fig. 1.

In our approach, assuming the UV origin is in the top-left corner, squares are laid out in a left-to-right, top-to-bottom order. Even indices map to the top-left half while odd indices map to the bottom-right half.

### 3.2 Accounting for Rasterization Issues

Due to rasterization rules, pixels whose center are not within a triangle are not rasterized as that triangle. Rasterization rules ensure that every pixel is uniquely mapped to one triangle when two triangles are adjacent. This causes some parts of some triangles to contain the color of an adjacent triangle as shown in Fig. 2. To prevent this, the triangle's UV coordinates are mapped to an inner triangle with a sufficient gap from the reserved triangle. Fig. 3 shows the positions

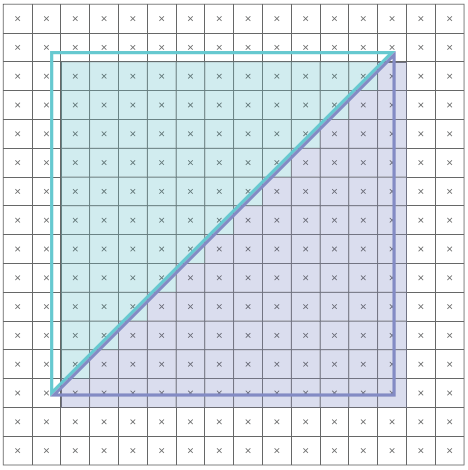

Figure 2: The issue with using the outer triangles as is due to rasterization rules. Parts of the light purple triangle near the diagonal are teal. If a UV coordinate lands in those teal parts, the color they sample comes from the teal triangle instead of the light purple triangle.

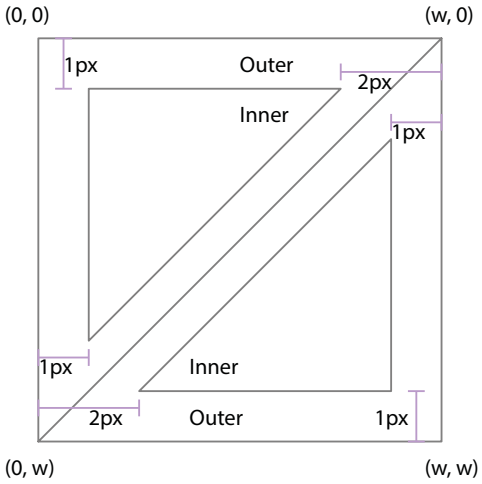


Figure 3: The coordinates and distances of the inner and outer triangles relative to the top-left corner of a square. w is the width of a square in UV space, which is the reciprocal of the size. A pixel in UV space is the reciprocal of the width of the texture in pixels.

of the vertices relative to the top-left corner of the square. To ensure that every part of the triangle is rasterized, a form of overestimated conservative rasterization must be used, in which any pixel that is partially inside a triangle is to be rasterized. DirectX 11.3 is the first feature level that allows conservative rasterization, but it is not supported by the HoloLens. Instead, we use the geometry shader to build a one-pixel border out of triangles, duplicating the interpolants at the corners to ensure the border matches the edges of the triangle. One such construction is shown in Fig. 4.

### 3.3 Projecting Camera Images Onto the Texture

The HoloLens provides video streaming capabilities from its camera. The color buffer is encoded in NV12 YUV color format, and both a view and projection matrix are provided. This information is used to project colors onto the UV texture. First, blurry frames should be rejected. This is done by detecting when the rotational or translational velocity of the HoloLens is greater than a threshold. Because the HoloLens does not expose its inertial measurement unit (IMU) data, we cannot use the accelerometer or gyroscope. Instead, the previous and current view matrices are used to transform both the origin and the forward unit vector. The difference between the previous and current origin represents the translation while the angle between the previous and current forward unit vectors represents the rotation.

If a frame is stable enough, it needs to be projected onto the UV texture. Projective texturing normally causes occluded objects to share the color of the object in front of it. To prevent the color from bleeding through objects, the projection is done in a two-pass fashion.

#### 3.3.1 Depth-Only Pass

The first pass uses the camera's view and projection matrix to perform a depth-only pass onto a separate depth texture using the set of meshes. This is analogous to creating a shadow map from the perspective of the camera. It must be noted that the projection matrix provided by the HoloLens does not properly map the $z$-coordinate and must be modified to factor in a near and far plane.

#### 3.3.2 Projection Pass

The second pass is the actual projection. The set of meshes are passed into the pipeline. The output of the vertex stage contains a clip-space position and a world-space position.

- **Vertex Stage**: Outputs an arbitrary clip-space position (this will be set in the geometry shader) and a world-space position from multiplying the vertex position by the mesh's model matrix.

- **Geometry Stage**: The offset and size are passed in as a constant buffer. For each input triangle, the geometry stage outputs 13 triangles (cf. Fig. 4). One of the triangles is the inner triangle, while the rest form the one-pixel border to conservatively rasterize the inner triangle. First, loop over every vertex of the input triangle and use the primitive ID, vertex ID (the loop parameter), offset, and size to calculate both the outer triangle's and inner triangle's UV coordinates using Alg. 1 and Alg. 2. We then project the inner triangle's vertices onto the outer triangle's edges to construct the 12 border triangles. The clip-space position corresponds to the UV coordinates mapped onto clip-space to rasterize the triangles on the texture in the correct location. The world space position is copied to the extra vertices to ensure the border contains the same colors as the inner triangle.

- **Pixel Stage**: The camera's view and projection matrices, the camera's luminance and chrominance textures, and the depth texture from the first pass are passed in. The world position is first transformed into UV coordinates by first projecting them into clip-space with the view and projection matrices, taking care to reject coordinates outside of normalized device coordinates (NDC) and with a negative $w$-coordinate. Then, the $x$ and $y$ coordinates of the NDC are mapped to UV coordinates for both the depth and camera textures. The depth texture is used to check if the pixel is occluded or not, discarding it if it is. If it is not, the pixel shader returns the RGB color from converting the camera textures from the NV12 YUV color format.

### 3.4 Updating Texture when Geometry Changes

When geometry is added by the HoloLens, two changes can occur to the layout. One, the GUID of the added mesh is in between existing meshes or before all meshes, causing the global indices of the existing meshes to shift. Two, the total number of triangles exceeds the amount allowed by the size of the grid being used. In these cases, the old texture cannot continue to be used. To preserve

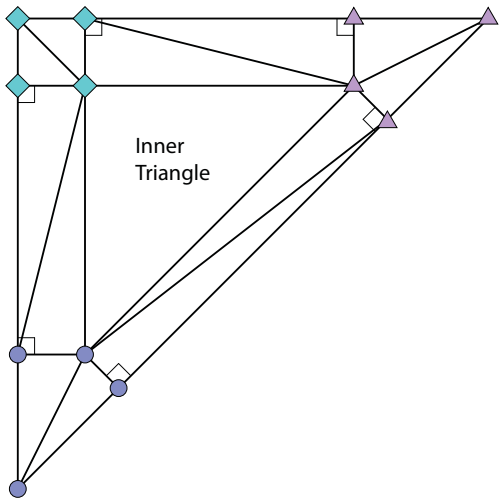


Figure 4: The construction of the 12 border triangles and inner triangle to be outputted by the geometry stage. Given the vertices of the inner and outer triangles, extra vertices are created by projecting the inner vertices onto the outer triangle's edges. The outer vertex and the two projections of the inner vertex share the same interpolants as the inner vertex. These are indicated with the same shape and color. The diagonals shown inside the border rectangles are arbitrary.

as much existing color data as possible, we transfer the data from the old texture to a new texture with the updated layout. Because the transfer requires both the old and new textures to be distinct, two textures are used in a double-buffered style, one being the current texture used in both projection and rendering, the other being the new texture to be copied to and then swapped.

Meshes can also be updated and refined by the HoloLens. Due to the lack of spatial coherency in our texture mapping scheme, this would cause that mesh's texture data to be invalidated upon update. Because most meshes are updated continuously by the HoloLens, we decided to ignore updates to existing meshes to avoid having gathered texture data be invalidated immediately upon looking elsewhere, causing an unsatisfactory experience for users trying to scan a large environment.

The procedure for texture updates on geometry changes is very similar to the projection step. First, two sets of offsets and sizes are calculated for both the old and new layouts based on the old and new sets of meshes. The old texture is bound as an input while the new texture is bound as the render target. Then, for each mesh that is in both the old and new sets of meshes, each triangle is rendered in the new layout's position while sampling colors from the old layout's position of the same triangle.

- **Vertex Stage**: Outputs an arbitrary clip-space position (this is the minimum required output of a vertex shader).

- **Geometry Stage**: Both the old offset and size and the new offset and size are passed in as a constant buffer. The new offset and size are used to calculate the clip-space positions of the inner triangle and 12 border triangles as per the projection step. The output vertex format also includes the UV coordinates with which to access the old texture with. These are calculated as the inner triangle using the old offset and size, and they are duplicated in the same fashion as the projection step.

- **Pixel Stage**: Samples the old texture at the UV coordinate and returns it.

**Algorithm 1** Calculating Outer UV Coordinates

Parameters: $primitiveID, vertexID, offset, size$
$width \leftarrow 1.0/size$
$evenOffsets[vertexID] \leftarrow [\langle 0.0, 0.0\rangle, \langle width, 0.0\rangle, \langle 0.0, width\rangle]$
$oddOffsets[vertexID] \leftarrow [\langle width, 0.0\rangle, \langle width, width\rangle, \langle 0.0, width\rangle]$

$globalID \leftarrow primitiveID + offset$
$squareID \leftarrow globalID/2$
$position \leftarrow \langle squareID \bmod size, squareID/size\rangle$
$topLeftCorner \leftarrow position * width$
**if** $globalID$ is even **then**
  return $topLeftCorner + evenOffsets[vertexID]$
**else**
  return $topLeftCorner + oddOffsets[vertexID]$
**end if**

**Algorithm 2** Calculating Inner UV Coordinates

Parameters: $primitiveID, vertexID, offset, size$
$p \leftarrow 1.0/resolution$
$w \leftarrow 1.0/size$
$even[vertexID] \leftarrow [\langle p, p\rangle, \langle w - 2.0 * p, p\rangle, \langle p, w - 2.0 * p\rangle]$
$odd[vertexID] \leftarrow [\langle w - p, 2.0 * p\rangle, \langle w - p, w - p\rangle, \langle 2.0 * p, w - p\rangle]$

$globalID \leftarrow primitiveID + offset$
$squareID \leftarrow globalID/2$
$position \leftarrow \langle squareID \bmod size, squareID/size\rangle$
$topLeftCorner \leftarrow position * w$
**if** $globalID$ is even **then**
  return $topLeftCorner + even[vertexID]$
**else**
  return $topLeftCorner + odd[vertexID]$
**end if**

### 3.5 Rendering and Exporting the Mesh

When using the texture to render the current set of meshes, the UV coordinates are calculated using the coordinates of the inner triangle. In our experiments, we render the mesh back in front of the user, allowing a user to compare the mesh with the real environment in AR. Parts of the environment that have not been mapped are transparent and therefore not visible to the user. When the mesh is textured, it will appear in stereoscopic 3D and approximately overlay the real objects. This will occur on-the-fly as the user looks around and grows to match their visible field of view. Artifacts of digital video capture, such as noise and color differences, and geometric inaccuracies, causing inaccurate colors from different viewpoints, are visible in this overlay. By comparing this 3D textured mesh and the expected color of the real objects being modeled, the user can determine the quality of the current texture and parts of the mesh that need to be textured or improved.

We pass the set of meshes into the pipeline. Two view-projection matrices are provided by the HoloLens to render stereoscopic 3D. By using hardware instancing, one draw call is sufficient to draw to both render targets simultaneously.

- **Vertex Stage**: Outputs the position of the vertex transformed with the model matrix of the mesh and the view-projection matrix array indexed with the instance ID. The SV_RenderTargetArrayIndex semantic is used to specify which render target to render to and is filled with the instance ID as well.

- **Geometry Stage**: The UV coordinates of the inner triangle are calculated from the offset and size passed in and appended to the vertex format.

Table 1: Small and Large Environment Results

| | # of Triangles | Time for capture (seconds) |
|---|---|---|
| Small | 3,656 | 40 |
| Large | 80,501 | 120 |

- **Pixel Stage**: The texture is sampled using those UV coordinates for the final color. This is equivalent to texturing a mesh with no lighting.

We also provide functionality to export the mesh by looping over the vertices on the CPU and calculating the UV coordinates of the inner triangle. These are then used to export a Wavefront OBJ file. The texture is exported as an image file, and a simple material file is created to specify that the texture is to be used by the OBJ file.

## 4 Results

In our experiments, we use the application to scan both a small and large environment. We use a texture size of 4096 by 4096 pixels. Care is taken by the operator to try and scan every portion of the environment. The time taken is noted as well.

Currently, interaction with our application is done through voice commands. The user can pause camera and geometry updates independently, which allows them to check on the current status of their scan. They can also export the mesh to the HoloLen's local storage.

In our first environment, we scan a section of a wall with a chest of drawers in front of it. The mesh ended up containing 3,656 triangles. Because of the lower amount of triangles, the amount of resolution available to each triangle is fairly high. The HoloLens failed to pick up parts of the chest, but the objects on top of it and wall are mostly filled in. While there is slight warping and blurriness, the overall quality of the scan is decent. This scan took 40 seconds, but with the geometry of the drawer not picked up, the actual time scanning the rest of the mesh was 30 seconds.

In our second environment, we scan a sizable living room. The mesh ended up consisting of 80,501 triangles. Most of the environment, with the exception of glass and the dark chest of drawers and sofas, was filled in with geometry. Because the geometry for the sofas was missing, the color was projected onto the wall behind them, requiring time to rescan those areas. In total, it took 2 minutes to complete the scan. Even though the number of triangles is significantly higher, the texture still provides a reasonable amount of detail to the mesh.

As shown by these two examples in Fig. 5, our method allows the user to easily scan even a large environment in a small amount of time. The adaptive texture mapping allows both environments to take the same amount of memory by spreading out the resolution between the triangles. This is evident when comparing the quality of a close-up of both meshes as shown in Fig. 6. In the UV texture itself, the triangles for the smaller mesh are significantly larger than their counterparts in the larger mesh. Even so, using just 64MB for an 8-bit RGBA texture, the quality of the texture is satisfactory for the larger mesh.

On the other hand, because we ignore mesh updates from the HoloLens, once an area has been scanned, its geometry will not be improved during the operation of our program. While, algorithmically, changes in existing meshes are handled correctly, practically, due to the near constant nature of these updates, as well as how they invalidate our texture's data for that mesh, it is nearly impossible for a user to texture an environment without pausing geometric changes. Currently, the only way to update existing meshes is to exit the program, allow the HoloLens to continue to refine the mesh, and then relaunch the application. An alternative would be to allow to user to choose when to start ignoring mesh updates and to perform one more full sweep of the environment to texture it.

In order to solve this issue, a mapping from the original mesh to the updated mesh is required. One potential method would involve projecting the triangles from the updated mesh onto the original mesh and retrieving the list of triangles, some of which are clipped, from which to copy the data to each new triangle. This can also be interpreted as clipping the original mesh by the extrusion of each triangle of the updated mesh in both directions. This could potentially be calculated on CPU with an acceleration structure to be executed in a single draw call on the GPU for real-time performance. Given the high rate of mesh updates along with the number of meshes updated per frame, finding a mapping that allows for fast copying and high data retention is an area for future improvement.

Another drawback arises when the triangles in the mesh differ significantly from the half-square shape they map to in the texture. This is caused by two different phenomena. First, while most triangles generated by the HoloLens are fairly regular in size and shape, long, thin triangles are oversampled in the short direction and undersampled in the long direction. This leads to blurriness in the long direction. Second, because the orientation of the triangle in the texture is arbitrary, alignments of the axis of the pixels do not necessarily line up between adjacent triangles in the mesh. This leads to visible differences in larger environments where each triangle has a lower resolution, which allows the edges of pixels to be discerned. Bilinear filtering alleviates some of this issue but does not solve it.

In addition, the spatially incoherent mapping of the triangle poses challenges for traditional hardware techniques such as mipmapping and anisotropic filtering. Mipmapping can still be utilized by manually building the mip chain using the same method as copying the texture data when the geometry changes. The number of mip levels must be limited based on the size of individual triangles rather than the resolution of the entire texture. This can be thought of as having a separate mip chain for each triangle. Since the border must remain as one pixel, there is increasing levels of wasted texture space for higher mip levels. Anisotropic filtering is extremely challenging because the size of the sample area can extend particularly far at glancing angles. Exploration into more spatially coherent mappings, such as automatic UV unwrapping algorithms that minimize distortion and minimize waste, could provide a more elegant solution to these issues.

Due to the real-time performance and fixed memory footprint, our method is viable for integration into other rendering applications. The algorithm works independently of the texture size and even aspect ratio, although the more different actual mesh triangles are from their mapped counterparts, the more distorted the distribution of resolution is.

The quality of the mesh generated by the HoloLens has a large impact on the quality of the scan. Missing geometry causes projections to bleed through their objects, overwriting the correct data. Our current method also does not take into account the quality of camera frame when projecting onto geometry. This leads to glancing angles and distance views to overwrite potentially better data from past frames.

User experience with the application can also be improved. Currently, the application does not guide the user toward areas of the mesh that have not been textured yet. Possible solutions include a heads-up display style arrows that point in the direction the user should look and so forth. The status and quality of scan are also not reported to the user. Potentially, our application could show a coverage amount or simplified image that shows the relative quality of the texture so far in that area. Also, our application does not differentiate between lack of texture and lack of geometry data. Both show up as transparent, which might cause confusion or make the user waste time trying to scan an area with no geometry. Because the HoloLens treats black as transparent, dark areas are also hard to differentiate between the overlay and the real world. Fixing this issue would require balancing color accuracy and user visibility.

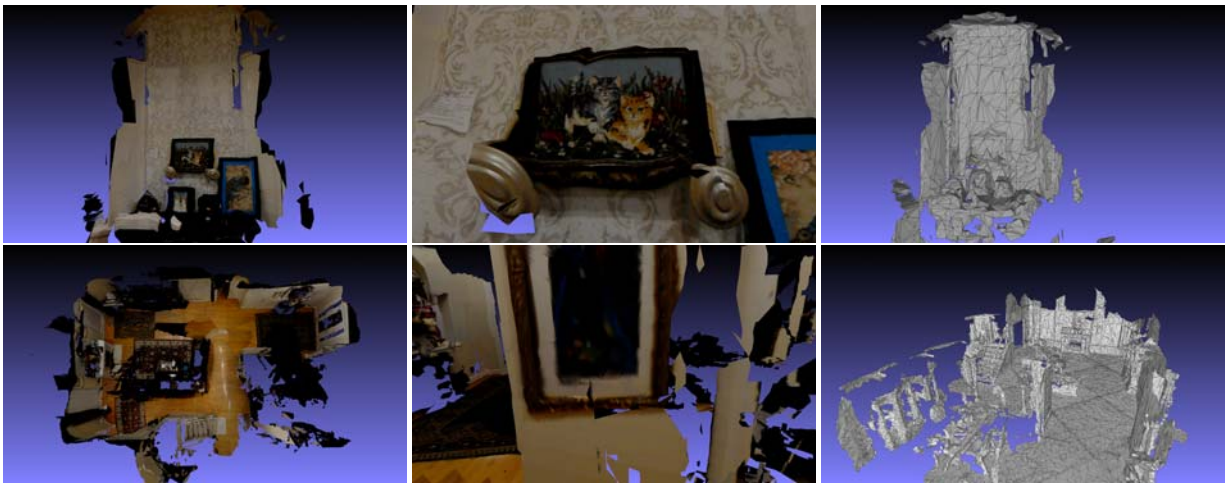

Figure 5: Scan of the small (top) and large (bottom) environments. From left to right: textured mesh, close-up of textured mesh, mesh geometry. Notice the difference in resolution in the two close-ups.

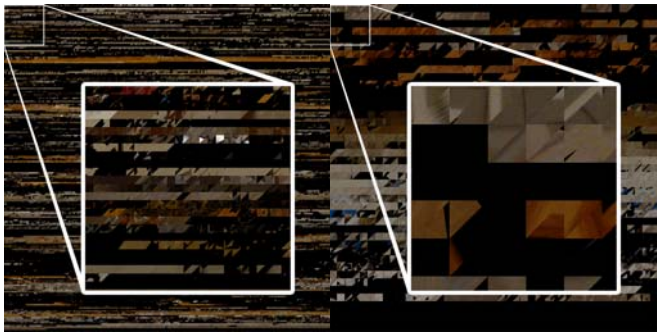

Figure 6: Comparison of textures for small (right) and large (left) environments. The textures are the same size, but because there are significantly more triangles in the large environment, the triangles are smaller relative to the small environment.

## 5 Conclusion

In this paper, we have demonstrated a real-time method to generate a standard UV texture of fixed size to color a dynamically changing mesh provided by the HoloLens. By utilizing the mobility and convenience of a headset device such as the HoloLens, we demonstrate modeling both small and fairly sizable environments in short periods of time. Because the user can view the creating of the mesh in real-time, it is also very simple to use, as they can see which portions of the mesh still need to be scanned. Because we provide a standard UV texture, it is trivial to integrate the mesh into common rendering pipelines and 3D applications, while also providing the fastest possible render time with no texture state changes necessary. The method is also fast enough to be integrated directly into a real-time pipeline.

There are still several shortcomings and extensions that we have observed. One, we currently do not apply any criteria for projecting camera frames onto the mesh. By utilizing a quality method, as was done in [2], we can prevent low-quality frames from overwriting better past data. We would still recommend factoring in how stale the data is to keep the texturing as up-to-date as possible. If such a method is integrated, adjacent triangles would most likely come from different frames, causing potential seams due to color or quality. A seamless mosaicing technique, such as [10], can be used to alleviate most of these issues.

Our method is heavily influenced by the quality of the geometry, especially because of its use as a proxy for both occlusion and rendering. Due to a variety of reasons, the HoloLens has difficulty scanning geometry from specific materials and complex objects. When projecting, occlusion is approximated with the proxy geometry, which can lead to projections onto occluded geometry. The quality of the output is also inferior to view-dependent methods such as [4] because access to multiple views allows for the illusion of more detailed proxy geometry and multiple accurate viewpoints.

Potential extensions include using the mesh as environment input into image-based lighting (IBL) techniques. Reflections can be dynamic where the user is looking while past data can fill in reflections in the other directions. By using IBL and possibly a combination of local cubemaps and screen-space reflections, it can be possible to more realistically render virtual objects in the real world in real-time.

## Acknowledgments

The authors wish to thank Dr. Pradeep Sen for discussions and comments. This work was supported in part by ONR grant N00014-16-1-3002.

## References

[1] F. Bernardini, I. Martin, and H. Rushmeier. High-quality texture reconstruction from multiple scans. *IEEE Trans. on Visualization and Computer Graphics*, 7(4):318–332, 2001. doi: 10.1109/2945.965346

[2] C. Buehler, M. Bosse, L. McMillan, S. Gortler, and M. Cohen. Unstructured lumigraph rendering. In *Proc. of the 28th Annual Conf. on Computer Graphics and Interactive Techniques - SIGGRAPH '01*, pp. 425–432. ACM Press, New York, New York, USA, 2001. doi: 10.1145/383259.383309

[3] C. Chen, M. Bolas, and E. S. Rosenberg. View-dependent virtual reality content from RGB-D images. In *IEEE Int. Conf. on Image Processing*, pp. 2931–2935, 2017.

[4] P. Debevec, Y. Yu, and G. Borshukov. Efficient View-Dependent Image-Based Rendering with Projective Texture-Mapping. pp. 105–116. Springer, Vienna, 1998. doi: 10.1007/978-3-7091-6453-2_10

[5] F. Dellaert, S. Seitz, C. Thorpe, and S. Thrun. Structure from motion without correspondence. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition. CVPR 2000 (Cat. No.PR00662)*, vol. 2, pp. 557–564. IEEE Comput. Soc. doi: 10.1109/CVPR.2000.854916

[6] M. Dou and H. Fuchs. Temporally enhanced 3d capture of room-sized dynamic scenes with commodity depth cameras. In *2014 IEEE Virtual Reality (VR)*. IEEE, Mar. 2014. doi: 10.1109/vr.2014.6802048

[7] P. Fechteler, W. Paier, and P. Eisert. Articulated 3D model tracking with on-the-fly texturing. In *2014 IEEE Int. Conf. on Image Processing*

*(ICIP)*, pp. 3998–4002. IEEE, Oct. 2014. doi: 10.1109/ICIP.2014.7025812

[8] P. Henry, M. Krainin, E. Herbst, X. Ren, and D. Fox. RGB-d mapping: Using kinect-style depth cameras for dense 3d modeling of indoor environments. *The International Journal of Robotics Research*, 31(5):647–663, Feb. 2012. doi: 10.1177/0278364911434148

[9] C. Kerl, J. Sturm, and D. Cremers. Dense visual SLAM for RGB-d cameras. In *2013 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*. IEEE, Nov. 2013. doi: 10.1109/iros.2013.6696650

[10] V. Lempitsky and D. Ivanov. Seamless Mosaicing of Image-Based Texture Maps. In *2007 IEEE Conf. on Computer Vision and Pattern Recognition*, pp. 1–6. IEEE, Jul. 2007. doi: 10.1109/CVPR.2007.383078

[11] R. Maier, J. Stuckler, and D. Cremers. Super-resolution keyframe fusion for 3d modeling with high-quality textures. In *2015 Int. Conf. on 3D Vision*. IEEE, Oct. 2015. doi: 10.1109/3dv.2015.66

[12] R. A. Newcombe, A. J. Davison, S. Izadi, P. Kohli, O. Hilliges, J. Shotton, D. Molyneaux, S. Hodges, D. Kim, and A. Fitzgibbon. KinectFusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE Int. Symp. on Mixed and Augmented Reality*, pp. 127–136. IEEE, Oct. 2011. doi: 10.1109/ISMAR.2011.6092378

[13] R. F. Salas-Moreno, R. A. Newcombe, H. Strasdat, P. H. Kelly, and A. J. Davison. SLAM++: Simultaneous Localisation and Mapping at the Level of Objects. In *2013 IEEE Conf. on Computer Vision and Pattern Recognition*, pp. 1352–1359. IEEE, Jul. 2013. doi: 10.1109/CVPR.2013.178

[14] A. Sankar and S. M. Seitz. In situ CAD capture. In *Proc. of the 18th Int. Conf. on Human-Computer Interaction with Mobile Devices and Services - MobileHCI '16*, pp. 233–243. ACM Press, New York, New York, USA, 2016. doi: 10.1145/2935334.2935337

[15] F. Steinbrucker, C. Kerl, and D. Cremers. Large-scale multi-resolution surface reconstruction from RGB-d sequences. In *2013 IEEE Int. Conf. on Computer Vision*. IEEE, Dec. 2013. doi: 10.1109/iccv.2013.405

[16] T. Whelan, J. McDonald, M. Kaess, M. Fallon, H. Johannsson, and J. J. Leonard. Kintinuous: Spatially extended kinectfusion. In *RSS Workshop on RGB-D: Advanced Reasoning with Depth Cameras*. Pittsburgh, PA, Jul. 2012.

[17] Q.-Y. Zhou and V. Koltun. Color map optimization for 3d reconstruction with consumer depth cameras. *ACM Trans. Graph.*, 33(4):155:1–155:10, Jul. 2014. doi: 10.1145/2601097.2601134