Real-time 3D Model Difference Reasoning with HoloLens

Pierre Beckmann Mathis Lamarre ETHZ

Abstract

Augmented reality and the associated technologies have shown significant growth for the industry and education of tomorrow. Software development for recently developed hardware such as the HoloLens is rapidly increasing. A very relevant aspect of augmented reality is to enhance one's perception of reality through overlaying. Our objective is to highlight removed or added objects in a scene in real-time on the HoloLens. This could be particularly useful to track the advancement of construction sites for example. This report summarizes our project within the Computer Vision and Geometry group of ETHZ, for the 3D Vision Class. We successfully created a system that shows depth differences using C# scripts and shaders in Unity. However it still lacks an alignment function between the loaded model and live model for use in real-time with a pre-loaded scan.

1. Introduction

Augmented reality can allow better and faster understanding of knowledge and data. Visualizing in 3D and using real-time overlaying provides an intuitive way to understand and therefore use information.

The aim of this project is to compute and display the 3D differences between a pre-loaded model of a scene and a current scan in real-time using the Microsoft HoloLens.

After accessing the depth maps, the z-value of each pixel in the live and reference models are compared. If the difference is bigger than a certain threshold, the corresponding space is marked as a difference. Depending on the sign of the difference, it can be an addition (scanned data is closer) or a deletion (scanned data is further). For an intuitive experience, the differences are highlighted as an overlay.

1.1. Developing for the HoloLens using Unity

Unity

Originally, Unity is a cross-platform game engine. It allows to use a graphical interface with drag-and-drop functionalities and scripting to develop video games and simulations for computers and mobile devices.

Since 2015, Microsoft and Unity have been working together to provide tools to create mixed reality applications for the HoloLens. Unity tools support the spatial mapping of the HoloLens as well as its audio, gaze, gesture, voice recognition and many more.

Coding in Unity involves placing objects into a scene. To those objects one can then attach C# scripts and associate shaders. A classical example of this is to place a camera in a scene with a certain script and shader attached. Once built and deployed on the HoloLens using Visual Studio, the user is able to see the rendering of the shader on the HoloLens screen. If a user wants to draw blue lines on a wall, he will write a script that identifies the wall using the camera input of the HoloLens and some computations, as well as a shader that will draw blue lines on the screen accordingly.

Creating objects and attaching scripts in Unity simplifies the implementation considerably and allows the user to use GUI instead of writing many lines of code.

Scripts and Shaders

Scripting in Unity is done using either C# or JavaScript. In this project, C# is used. Scripts can respond to input and manipulate objects in the scene. Then, the video post-processing is done by the shaders.

Shaders are designed specifically to run on a GPU. It is ultimately what draws the triangles of 3D models. They are coded in HLSL (High-Level Shader Language) [1]. This shading language was developed by Microsoft for the Direct3D 9 API. It supports five different forms of shaders, in this project we only used the fragment shader. The use of HLSL makes debugging extremely difficult because no console output or error messages are possible.

HoloToolkit

The spatial mapping is done using scripts from the HoloToolkit [2]. The HoloToolkit is an open source library that offers multiple tools to simplify developing for the HoloLens. It results from a collaboration between SOTA (a Microsoft studio) and NASA (in the context of the Mars rover) and is widely used in the community.

2. Methods

2.1. Depth Difference Computing and Rendering

In order to compute the depth difference between the live and reference models, we first need to get the two depth maps. Then, they are both accessed directly on the GPU in a single shader. This shader then highlights each pixel according to the sign of the depth difference: added objects in blue and removed objects in red (see fig.1).

On the HoloLens, an simulated camera accesses a pre-loaded model to create the reference view which will be compared to the spatial mapping in the live view. This was implemented in Unity and is described in fig.2.

As a first step, both cameras were set to see very simple Unity scenes, consisting of objects of basic shapes. This is offline, as both scenes are pre-loaded. This allowed us to implement the shaders and test them in Unity.

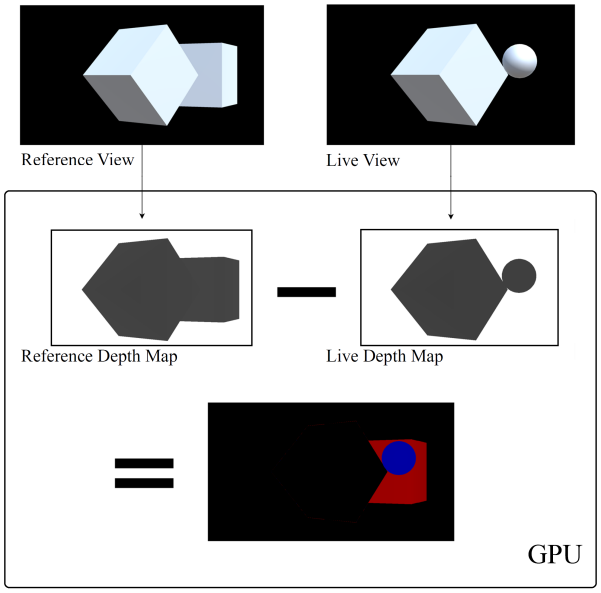

Figure 1. Basic principle of the algorithm. The depth maps of the reference view and live view are compared per-pixel directly on the GPU with the use of a shader. Removed objects are shown in red, added ones in blue.

2.2. Unity Implementation

In Unity, two cameras are used. The Reference Camera, which accesses the pre-loaded model. In a first implementation, this reference model consisted of simple virtual shapes like squares or spheres. Later, once the results were satisfactory, the reference camera was set to see the actual pre-loaded scan, saved as an object file. The Live Camera, which will eventually observe the live scan of the model. At first, it was also set to see simple shapes made in Unity. To see the live model, we attached scripts from the Spatial Mapping HoloToolkit, mainly `SpatialMappingObserver.cs`.

Then, to get the depth of the reference model, we implemented `LoadDepthToBuffer.cs` and `RenderDepth.shader` which are attached to the Reference Camera. At each frame update, the `Render()` function of the Reference Camera is called by the `RenderDepthDifference.cs` attached to the Live Camera. Since the Reference Camera has no target, the rendering is offscreen, but the depth does go into the buffer.

Finally, in `RenderDepthDifference.shader` which is attached to the live camera, both depths are accessed through the `_LastCameraDepthTexture` (reference) and `_CameraDepthTexture` (live) keywords. The difference is computed and if it is bigger than a threshold, the pixel is highlighted according to the sign. Blue for an added object (present in the live scene but absent in the reference) and red for a removed object.

To make sure that the pose of both cameras is always the same (the live camera is moving), the position and orientation of the reference camera are set to those of the live camera at each frame update.

For a more intuitive experience, the texture of the live model is overlaid with the highlighting colors so that the user can track the spatial mapping in real time. Also, to give an impression of depth, the intensity of the color is inversely proportional to the distance to the viewer. This way, instead of seeing all the objects as uniform surface, the user will be able to distinguish them. The intensity of the texture, the shading of the colors as well as the depth difference threshold are all parameters whose value can be changed.

2.3. Parameter tuning

A built-in Unity function `UNITY_SAMPLE_DEPTH` returns a value between zero and one for each pixel. They are respectively the maximum sensor range and the closest measurable distance. Using another Unity function, `Linear01Depth`, we get values between zero and one but with a linear scale. Also, zero is the closest and one is the maximum range. The HoloLens' depth range is between 0.85m and 3.1m.

The depth difference is therefore between zero and one or zeron and minus one depending on the sign of the difference. The parameter `DepthDifferenceThreshold` is the size of the region around 0 considered as not significant. In both directions, depth differences between `-DepthDifferenceThreshold/2` and `DepthDifferenceThreshold/2` are not highlighted.

This parameter often has to be tweaked according to the application. For real-world use on the HoloLens, 0.1 seems to be the optimal balance between noise and sensibility.

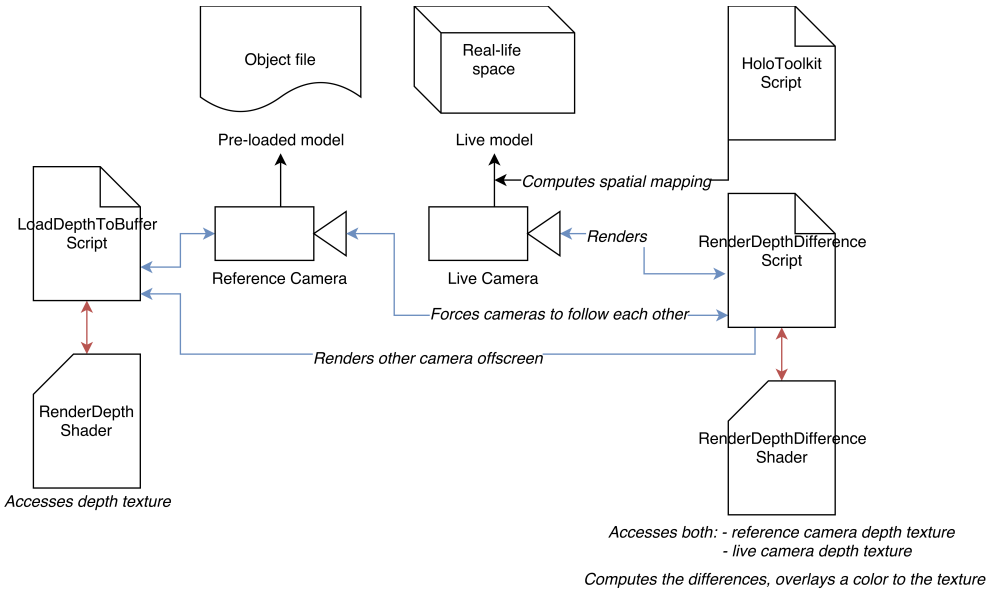

Figure 2. Flowchart of system organization in Unity. The reference camera sees the pre-loaded model whereas the live camera sees the spatial mapping of the real world using the HoloToolkit script. The RenderDepthDifference shader attached the corresponding script renders the depth difference onto the live camera by using the depth maps of both cameras. To access the reference depth it has to render the reference camera to call the LoadDepthToBuffer script and associated shader. CPU operations in blue GPU operations in red.

3. Results

3.1. Basic Scenes

With simple Unity objects, the algorithm works as expected (see fig.3). In a scene with two cubes, one was removed and a ball was added. The output does show the missing cube in red as a "ghost" and the added ball in blue. Since the texture of the live scene is kept, the added objects have a 3D impression while the removed ones don't.

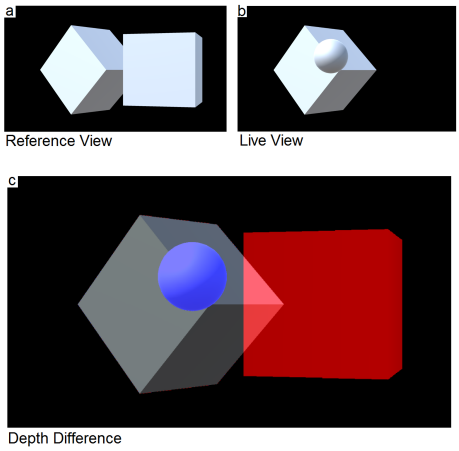

Figure 3. 3D model difference reasoning of basic Unity scenes. (a) shows the reference view. (b) shows the live view. (c) is the output of the algorithm, with removed objects in red, additions in blue and live texture in grey.

3.2. Immersion improvements

In order to make the experience more intuitive, especially when moving around, we wanted to give an impression of depth. To do so, we made the color intensity proportional to the distance to the viewer. In fig.4, we can clearly distinguish the person laying back on a chair in front of a window in the background.

Figure 4. Shading according to depth. The brighter the blue, the closer the scan is to our current position.

3.3. Offline results with real scans

After trying the algorithm on virtual objects, we used actual scans of the HoloLens. To do so, we scanned the lab before and after moving a chair. As we can see on fig.5, the old position of the chair is highlighted in red, while the new position is highlighted in blue. However, there is a lot of

noise due to the imprecision of the spatial mapping. This leads to false positive highlighting: lots of small areas are colored in blue or red when they should not.

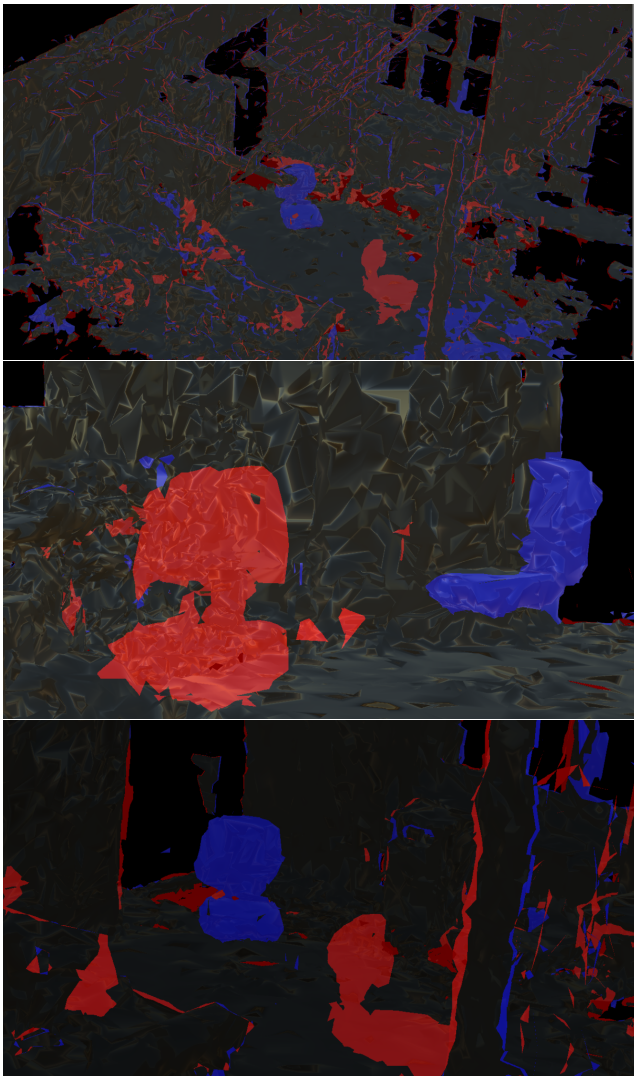

Figure 5. 3D model difference reasoning of two scans of the lab. Both scans were made with the HoloLens. Different views levels of texture intensity were chosen. Added objects are in blue and removed ones in red.

3.4. Real-time difference reasoning

Since the alignment of a pre-loaded model with the live scan of the HoloLens is not implemented yet, we tried to use the algorithm in real time with a very simple model instead of a scan. We simply chose a big pillar, as it relates to construction sites. As we can see on fig.6, it is shown in red in most of the room as it is missing. All the live spatial mapping done by the HoloLens is new, so it is shown in blue. An interesting portion of the picture is where the pillar crosses the table: since the difference is smaller than the threshold, it is not highlighted.

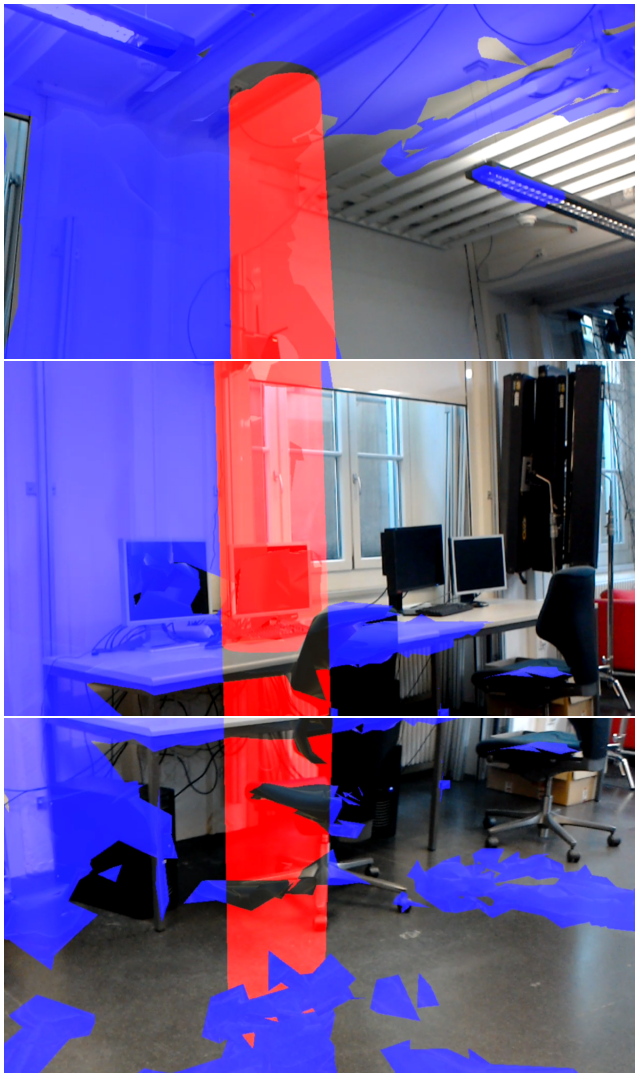

Figure 6. Real-time difference reasoning with a simple reference model in the lab. The pre-loaded model is a pillar. For some reason, the screenshots of the HoloLens barely show the textures, but they are visible on the device.

4. Workload distribution

We worked together through the entire project. A lot of time, especially at the beginning, went into internet research about shaders and HLSL. Nils Funk had a big role in this, but unfortunately he left our group and dropped the class midway through the semester. We also read the documentation and did some Microsoft Mixed Reality Academy Tutorials. Because so little documented work or forums exist about working with the HoloLens, a lot of time-consuming debugging was necessary.

5. Discussion

We successfully implemented an interactive visualization of model differences on the HoloLens. Our algorithm works really well offline with simple shapes and smooth surfaces. The overlay on the HoloLens is satisfying, and it is very fast (after the few seconds taken for the spatial mapping). Some problems arise when using real scans because of their imprecision. Satisfying results can be achieved when fine tuning parameters, but at a cost of less sensibility. Real-time use should work as soon as a way to align the pre-loaded model and the live scan is found.

5.1. Outlook and Improvements

Alignement

The alignement is a crucial problem to solve. ICP or other alignement techniques could be used. However we noticed that some HoloToolkit scripts seem to do just that. Unfortunately we didn't manage to use the functionality yet because of the lack of an explicit implementation and very sparse documentation.

Noise

Concerning the noise, even if the difference threshold reduces false positives, the results are stil not perfect. Filtering techniques, such as median filtering, could be useful to reduce the noise.

References

[1] R. J. Rost, B. Licea-kane, D. Ginsburg, J. M. Kessenich, M. Weiblen, B. Lucea-Kane, D. Ginsburg, J. M. Kessenich, B. Lichtenbelt, H. Malan, and M. Weiblen, *OpenGL Shading Language*, 2009. [Online]. Available: http://www.amazon.com/OpenGL-Shading-Language-Randi-Rost/dp/0321637631 1

[2] Microsoft, "HoloToolkit ReadMe," 2017. [Online]. Available: https://github.com/Microsoft/HoloToolkit/blob/master/README.md 1