Digitally Representing Physical Objects for Collision Avoidance in Virtual Reality

Siddartha Devic, Alec Moore, James Eubanks, Chengyuan Lai, Dr. Ryan P. McMahan*
The University of Texas at Dallas, FIVE Lab

I. Introduction

Relatively inexpensive virtual reality (VR) headsets have allowed for unprecedented consumer participation in the sector. However, one problem that the typical consumer may face is the lack of a large open play area for tracking. Tables, chairs, even the pillars of a house may be occluding and taking up valuable meters in the tracked space.

By creating a way to digitally represent physical objects in a tracked space, the user will be able to expand and keep obstacles within that tracked space. As the user draws closer the digital representation of the object, the digital object will appear to fade into view, and a simple wireframe outline of the selected physical object will become visible.

The greatest problem in implementing this solution was in fact user selection: How does the user select or "trace" the physical object regardless of visual feedback? The initial idea was the "Box Method", described below. Eventually, a more efficient and intuitive method was conceived of and implemented.

The implementation was built in Unity3D with the SteamVR plugin and an HTC VIVE headset and controllers for testing.

II. The Box Method

The "Box Method" allows the user to instantiate a rectangular prism by defining two opposing and diagonal vertices. (Figure 1)

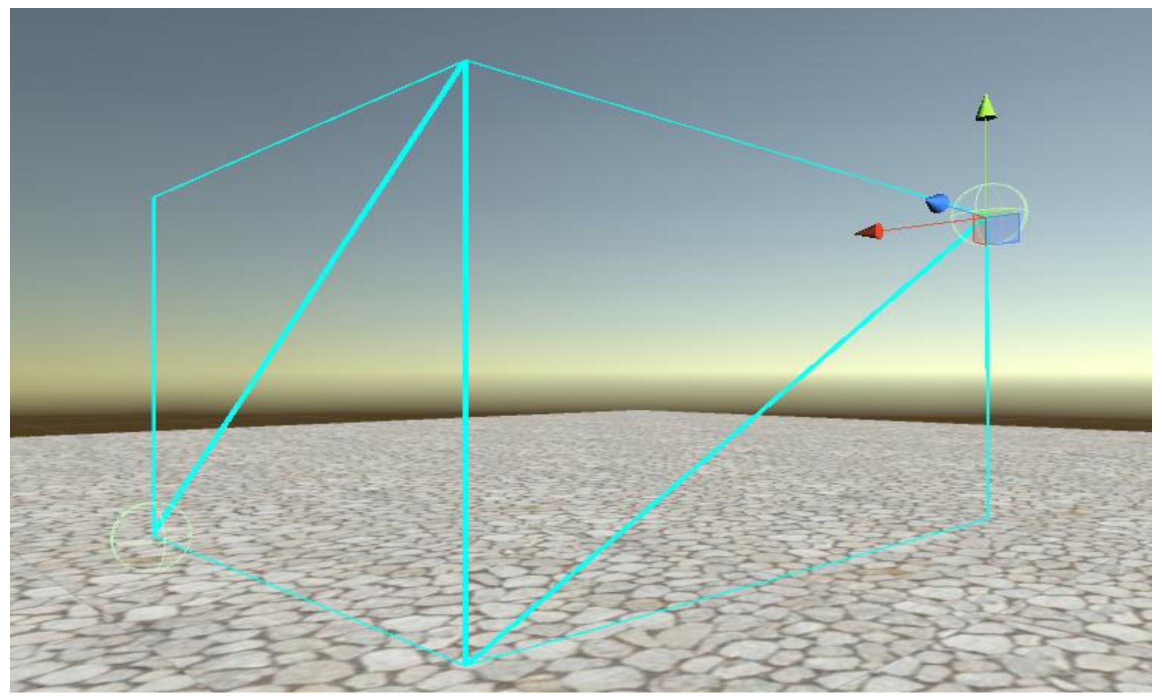


Figure 1: "Box Method". Opposing vertices are defined by the user in upper right corner and lower left corner.

In the "Box Method", the user defines each vertex by moving either controller to the desired location in physical space, and pressing the trigger on the corresponding VIVE controller. The initially 1x1x1 selection box is then scaled with respect to the relative difference of position along the x, y, and z axes between the two user defined vertices. The box is then translated to the correct position in virtual space.

Due to the fact that the object is not instantiating at runtime—simply changing the scale of an existing object and making it visible—this method was relatively easy to implement.

However, the ease of implementation is reflected in the poor accuracy of representation. This method is only suitable for uniform, box-like objects. If given more complicated objects to trace, such as an object with a zig-zag edge, there would be a noticeable difference in position between the physical or real edge of the object, and the virtually represented edge of the object. When dealing with already minimized spaces, maximizing the playable area is an important priority.

III. The Convex Hull Method

The inaccuracy problem of the "Box Method" can be solved in a fairly simple way: by giving the user more control over the placed vertices of the object.

In the "Convex Hull Method", the user places a series of vertices on the ground, around an object, as well as a single point used for calculating the height of the object. In case the user places points that are not incident on the edge or are not vertices of the largest creatable shape, we find the convex hull of the placed points. The convex hull is largest shape that encompasses all defined points, where each interior angle is less than 180°. (Figure 2)

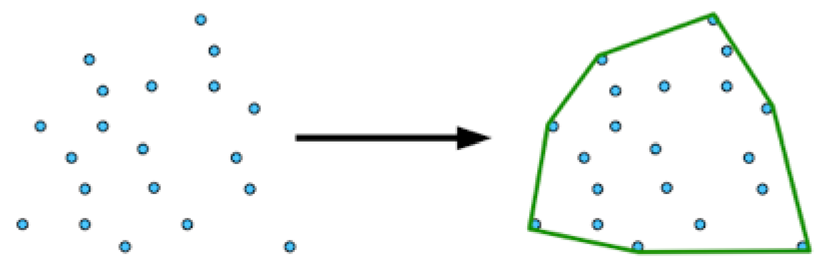


Figure 2: Convex hull creation with user defined vertices. (credit: Brigham Young University)

The resulting object has the ability to be more complicated than a simple box (Figure 3), which allows for more accurate digital representations of objects, and by extension, larger play spaces.

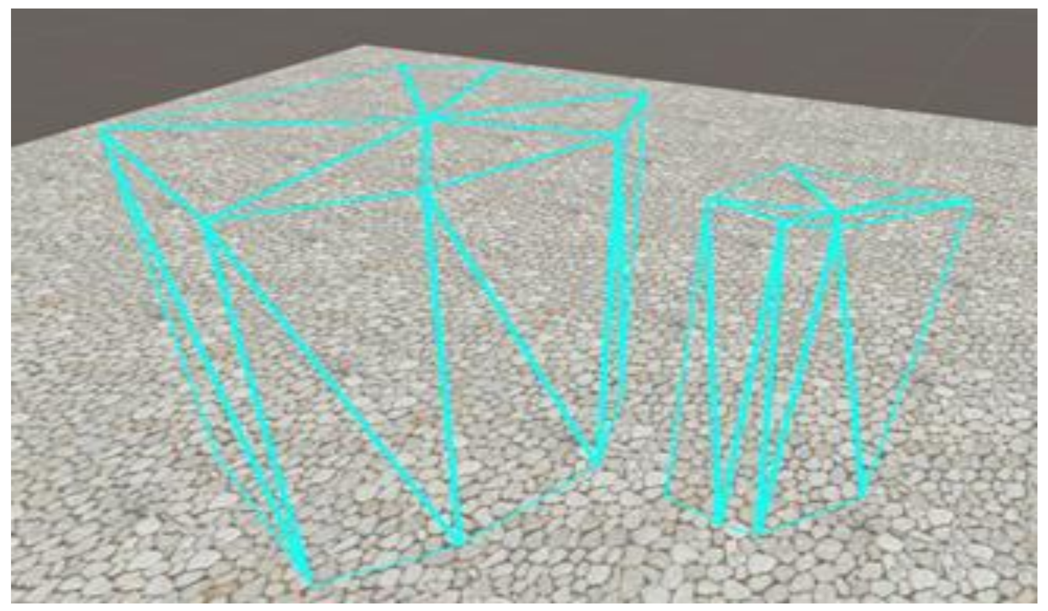


Figure 3: Convex user-created objects

IV. Procedural Meshes in Unity

When creating "meshes"—Unity's name for dynamically modifiable game objects—through Unity's script system, there needs to be three different types of information passed through for the mesh to not only be rendered, but be rendered correctly.

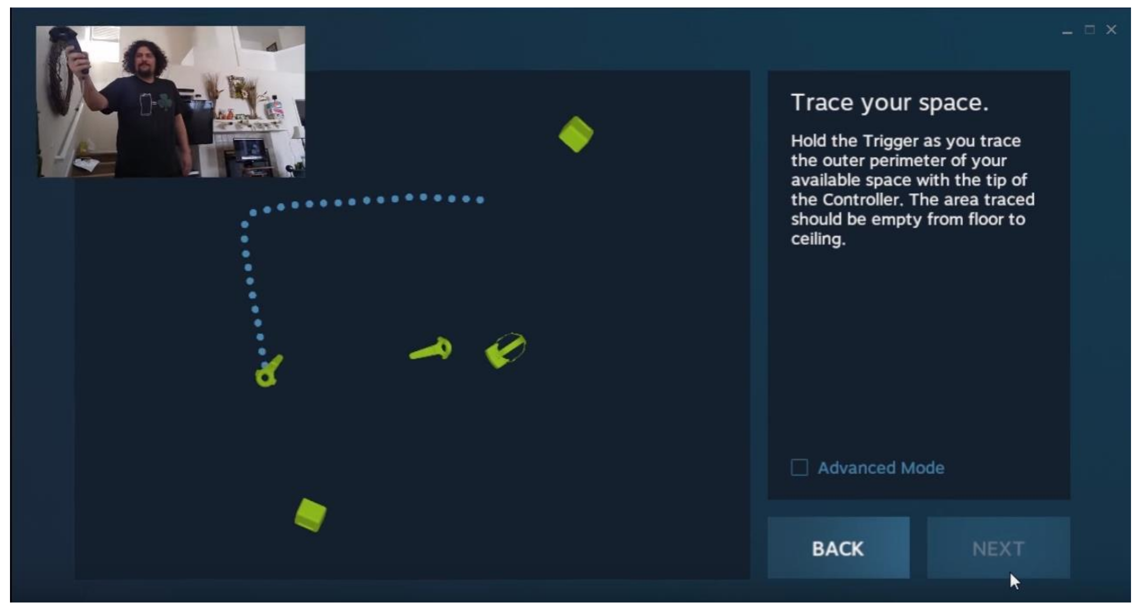
1. The mesh vertices
2. The mesh normals (used for lighting calculations and texture assigning)
3. The mesh triangles (used for rendering, see: Figure 1)

The mesh vertices are user defined. The mesh normals—because the object is using a transparent wireframe shader whose appearance is not affected by the surrounding lighting—are arbitrary. The mesh triangles, however, need to be assigned at runtime. The algorithm for assigning these triangles must be able to run regardless of the number of vertices the user places.

Once all the pieces of the mesh are defined, we can enable the mesh renderer and the object will now be visible to us on the screen and in VR.

V. The VIVE Input Method

While the "Convex Hull" method proved to be effective, the method itself is unintuitive for new users. The user must use 3 or 4 different buttons in a certain order, each with their own role in the creation of the object. To create a more intuitive user experience, we chose to replicate a method that SteamVR uses when setting up the tracked space (Figure 4), but in a different context.



Trace your space.

Hold the Trigger as you trace the outer perimeter of your available space with the tip of the Controller. The area traced should be empty from floor to ceiling.

Figure 4: SteamVR room setup, the user holds the controller trigger and traces the edge of the playable space. (credit: Jeffrey Grubb, YouTube)

Our implementation of this method has the user hold the controller trigger down and "trace" the edge of the physical object. (Figure 5)

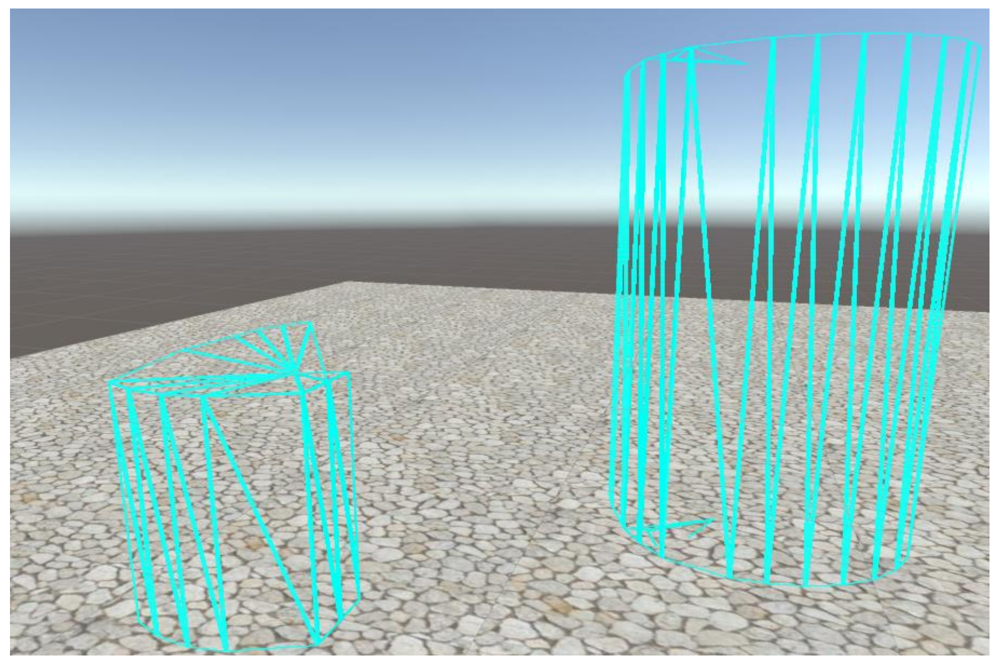


Figure 5: Some interesting creatable shapes using the "VIVE Input Method", a pseudo-tracing algorithm.

VI. Implications and Future Work

If game developers or SteamVR were to implement this as part of game setup, users would be able to play games in VR without removing large objects like tables or chairs from their play area, and previously unusable areas would now be available to play in.

A comprehensive user study comparing different shaders and user immersion levels is still needed before this implementation becomes ubiquitous.

VII. Acknowledgements

I would like to thank Alec, Jack, and Coleman for always being there to answer my sometimes trivial questions. I would also like to thank Dr. McMahan for allowing me to work in this lab, as well as Adam and Alan for keeping me company during long days. Finally, thank you to the Clark scholars program for giving me this wonderful opportunity even before my freshman year.