International Conference on Artificial Reality and Telexistence
Eurographics Symposium on Virtual Environments (2016)
D. Reiners, D. Iwai, and F. Steinicke (Editors)

Real-Time Marker Tracking with Microsoft Kinect

Alexander Gunter[1], Andrew Robb[2], & J. Adam Jones[1]

[1]University of Mississippi, United States
[2]Clemson University, United States

Abstract

In this document, we discuss a work-in-progress which will implement real-time marker tracking with the Microsoft Kinect. The Kinect uses an infrared depth sensor to capture 3D video, and retroreflective surfaces are not visible to this sensor. Thus we can adhere anular markers made of retroreflective material to a surface and identify them in the depth sensor's output. By placing a non-reflective surface at the marker's center and identifying it in the depth image, we can track the marker a surface and map it to a point in a virtual space. We are able to identify potential markers using an optimized and modified version of the flood fill algorithm, and we will filter out noise by validating potential markers against the known topology and dimensions of the markers.Our goal is to demonstrate real-time tracking and virtual representation of the marker using the Unity Game Engine.

Categories and Subject Descriptors (according to ACM CCS): I.4.8 [Computer Graphics]: Scene Analysis—Tracking

1. Introduction

Currently the Microsoft Kinect SDK is only capable of tracking humans and human appendages, and consequently it is not optimized to track general objects with the Kinect [Kin]. This work-in-progress aims to extend the Kinect's capabilities by implementing real-time marker tracking. This could quickly and efficiently produce a flexible, low cost solution for general purpose motion tracking.

The Kinect produces two images, the color image and the depth image [Kin]. The depth image is produced using an infrared laser whose surface reflections is then used to estimate distances. Markers made of retroreflective material thus provide an invalid depth reading which is relatively easy to identify in the depth image.

The Unity3D Game Engine is a platform which supports the use of C# scripts and is able to display virtual objects in a 3D scene [Uni]. Microsoft also provides a Unity3D plugin for the Kinect SDK. Thus we can use Unity3D to easily access the Kinect's depth image and visualize a virtual environment.

2. Planned Implementation

The project requires two components: a trackable marker which can be affixed to a surface and software to identify and track the marker.

The marker's shape will be a symmetric shape with a square hole at the centroid, and it will be made of retroreflective tape. For initial testing, the marker will be affixed to a flat static surface which will remain in the Kinect's field of view. In the depth image, this marker will appear as a contiguous region of zeros with a non-zero island at the centroid.

The tracking software will have three primary calculations stages following acquisition of a depth image from the Kinect. These stages will be to search for features in the depth image which might be markers, filter out noise in the depth image, verify markers, and map the markers to points in a three dimensional space.

The searching stage will primarily rely on a flood fill algorithm. This algorithm is equivalent to a breadth first search in a graph where each pixel corresponds to a graph node. The program takes a pixel from a search queue and applies a decision function to the pixel's depth value. If the function returns true, the program records the pixel as such and adds its Moore neighborhood to the search queue. This produces a contiguous set of pixels satisfying the decision function, and recording pixels which fail the decision function also produces the borders of holes. For this program, the decision function will check whether the pixel's value is non-zero; and it will return true or false depending on what the program is currently searching for. Without optimizations, three applications of this algorithm, alternating from finding non-zero regions to zero regions and back, is sufficient to identify potential markers in the depth image.

The filtering stage will take the results from the searching stage and produce features which correspond to markers. The primary purpose of this stage is to filter out noise which may produce artifacts topologically identical to a marker. This noise can be filtered by verifying known properties of the markers. First, markers must have exactly one non-zero island. Additionally, that island must be at the centroid of the feature. If the program knows the real-world

DOI: 10.2312/egve.20161451

size of a marker, and a potential marker has exactly one non-zero island, then that island's average depth value indicates the real-world size of the feature. If this does not match the expected size, the feature is probably not a marker. Finally, if there is a set minimum and maximum distance which markers may reside, features lying outside that range can be filtered out.

The mapping stage will be the simplest due to pre-existing solutions built into the Kinect SDK. The SDK can already map a pixel in the depth image to a 3D coordinate. To map a marker to a 3D coordinate in Unity, we only need to pick the pixel closest to the centroid, use the Kinect SDK to convert it to 3D coordinates, and perform a change of basis to correctly map that point to a point in Unity's space.

There are a number of potential optimizations for improving execution time of this program. Some of the filtering checks are fast and could significantly reduce the search space in the searching stage if performed prior to searching. Specifically, if markers have a known size and have a maximum possible distance from the depth sensor, then there is a minimum possible feature size in the depth image for a marker. Then there is a minimum width and a minimum height for a marker. We can then map each pixel to one if it is non-zero and zero otherwise. We can then map each row and column to the sum of these Booleans. Columns and rows with no zero-regions will map to their own length, and we can subtract a marker's minimum height and width from a column and row's length respectively to obtain a reliable threshold. Columns and rows below this threshold might contain markers, and these correspond to a set of pixel coordinates to test which should be smaller than the set of all pixels.

3. Results to Date

So far we have implemented prototype programs for capturing a depth image, the searching stage of the tracking algorithm, and a part of the filtering stage. We have also estimated the relationship between the Kinect SDK's depth values and real world distance.

Of particular note are the searching and filtering prototypes. For this, we have used Python to implement the flood fill algorithm and the row and columnwise filtering discussed in Section 2. For this filtering, we have assumed a minimum marker height and width of fifty pixels. The searching stage uses the flood fill algorithm to search the remainder of the image for zero-regions.

To convert real world distances to the Kinect SDK's depth values, we sampled pixels from depth images with known real-world distances. Plotting real world distance against the corresponding depth value indicated a linear relationship where 1 mm roughly corresponds one unit on a scale from 0 to 4096 [Kin].

4. Future Work

Remaining work for this project includes completion of the filtering stage as well as the mapping stage. We also need to port the existing prototypes to Unity, and we need to implement a method for tracking markers across multiple consecutive frames.

Future developments include optimizing the design of markers for use on more general objects. With static objects and appropriately placed markers, it would be possible to record the relationship between markers and translate the virtual representations of corresponding obstructed markers based upon the movement of visible markers. These algorithms could also be ported to an importable C# library for use in other virtual reality projects as an unobtrusive supplement to the Kinect SDK.

Figure 1: *A sample depth image converted to a grayscale JPEG.*

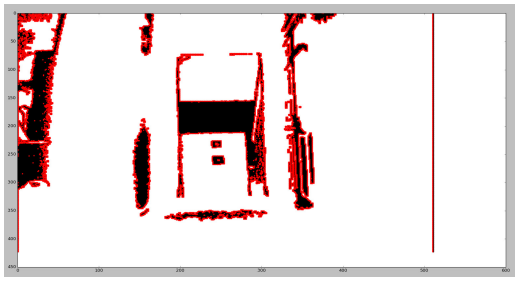

Figure 2: *Output from the prototype searching stage visualized with Pyplot. Red pixels represent non-zero pixels in the depth image. Black pixels represent pixels in the depth image with a depth of zero. The X and Y axes are the X and Y coordinates of the corresponding pixels in the depth image.*

References

[Kin] Kinect for windows sdk. https://msdn.microsoft.com/library/dn799271.aspx. Accessed: 2016-10-12. 1, 2

[Uni] Unity manual. https://docs.unity3d.com/Manual/index.html. Accessed: 2016-10-12. 1