A Virtual Theremin Using Kinect

Niall Wingham

April 8, 2013

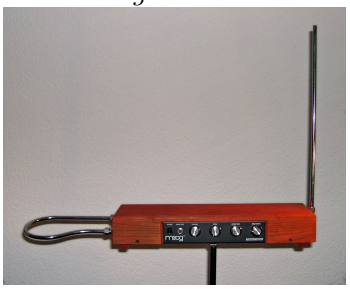
Abstract

The theremin is an electronic synthesizer that is played without being touched. Because it gives no haptic feedback, it a natural candidate for virtualization. In this paper we discuss our design and implementation of a virtual theremin using the Microsoft Kinect. We demonstrate algorithms that reliably track hands and recognize simple gestures at 30 frames per second.

1 Introduction

A theremin consists of two perpendicular metal rods which each act as a plate in a capacitor. The performer's hands act as the corresponding ground plates, and the hand moving to and from the rod changes the frequency of an associated oscillator. Typically, the distance between the right hand and the vertical rod controls the pitch of the theremin while the height of left hand above the horizontal rod controls the volume. For a video of Leon Theremin playing his invention, see http://youtu.be/w5qf9O6c20o.

Figure 1: A *Moog* Etherwave Theremin



1.1 Motivation

The theremin is a fascinating instrument, but it has several limitations. Playing a theremin is quite challenging because the pitch changes over a continuous range (most instruments play only discrete pitches). The performer needs an excellent ear to constantly "tune" the instrument. Theremins are also difficult to obtain: a would-be hobbyist must be willing to spend a few hundred dollars on Ebay, or be able to assemble one from a kit. Finally, the theremin's characteristic ethereal sound is only appropriate for certain types of music.

1.2 Objectives

By building a virtual theremin, we aim to address each of these limitations through software. In particular, our objectives are to:

1. Provide visual feedback to the performer, so it is easier to play.
2. Use only commodity hardware, so it is cheaper and more accessible.
3. Synthesize a variety of sounds, so it is more expressive.

1.3 Outline

In Section 2, we introduce background material and terminology relevant to our implementation. In Section 3, we describe the implementation of our virtual theremin. In Section 4, we show test data and discuss the strengths and weaknesses of our algorithms. In Section 5, we raise possibilities for future work.

2 Preliminaries

2.1 Synthesizers

A synthesizer consists of one of more voices, each of which generates sound through a chain of oscillators and effects. A sine wave is a simple oscillator, while raising or lowering the volume is a simple effect on an oscillator. To play a synthesized sound, each voice is sampled several thousand times per second. An excellent introduction to synthesizers can be found here.

2.2 Kinect

The Microsoft Kinect provides three sets of data at each frame: a colour image, a depth image, and a skeleton image. The colour and depth images both have pixel-coordinates, but they are slightly offset from each other due to the relative position of the cameras on the device. The skeleton exists in real X,Y,Z space and consists of joints such as `HandLeft` or `HipCenter`. The Kinect provides methods for transforming points between these three sets of coordinates. More information on the Kinect can be found here.

2.3 Pseudocode

Our project is implemented in C#, and we often make use of its functional-style LINQ methods, so it will help to be familiar with them. For example, `[1..100].Where(Even).Take(5)` would return the (lazily generated) list `[2,4,6,8,10]`. In the context of a function that returns a list, the keyword `yield` means to return the next item in a lazily generated list. More information can be found here.

3 Implementation

Our implementation of the theremin consists of two main components: a sound synthesizer, and a hand tracking & gesture recognition system. The main thread is responsible for drawing the user interface and synthesizing a pitch based on the current hand positions. A second worker thread receives and processes data from the Kinect, updating the hand positions and generating gesture events. Pseudocdoe for each algorithm is included as an appendix, and the full source code of the implementation is available at https://github.com/niallwingham/theremin.

3.1 Synthesizer

Our synthesizer consists of a single voice that can choose between oscillators. We have implemented sine, triangle, sawtooth, and square wave oscillators. A fifth oscillator demonstrates harmonic overtones using the sine oscillator.

3.2 Hand Tracking

Our hand tracking algorithm builds on segmentation by depth, from e.g. Raheja et al. [1]. The idea is to (a) locate a starting point in a depth image that is part of a hand, and (b) filter out pixels that are far away from the starting point. We use the left and right hand joints identified in the skeleton image as our starting points. We then transform the depth image to skeleton space and select only the pixels within a 15cm cube centered 10cm in front of the hand joint.[1] We call this process segmentation by volume.

To determine the center of the hand, we simply take the 3D geometric average of the filtered points. We use this point to calculate the X and Y offsets of the hands for controlling pitch and volume. This is preferable to using the raw hand joint for two reasons. First, we found that the raw joint had significant jitter even when the hand was kept still.[2] Secondly, the geometric center of the hand is easy for the performer to control, e.g. by extending or closing a single finger, which helps to switch between nearby pitches.

3.3 Gesture Recognition

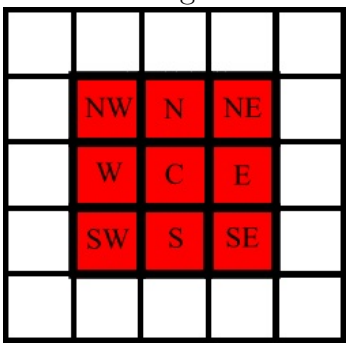
Using the depth image for each hand, we are able to identify fingertips using a method from [2]. First, we calculate the convex hull around the depth points in our image using Andrew's Monotone Chain Algorithm [3]. It is similar to Graham's scan [4] but has better performance as it avoids calculating the angles between points.

Next, we calculate the contour of the hand using the Moore Neighbourhood Tracing Algorithm. The Moore neighbourhood of a pixel in an image is the set of eight pixels surrounding it. The tracing algorithm finds the outline of a "black" image on a "white" background. In our case, black pixels are ones that passed segmentation by volume. It works by finding an arbitrary black point on the outline (e.g. the one with the lowest X coordinate) and circling through the point's Moore neighbourhood in a clockwise direction until an adjacent black point is found. In turn, this point's neighbourhood

[1] We found that the hand joint was often at the base of the palm or the wrist, so we position the segmentation volume *in front* of the joint to ensure the forearm is not included.

[2] We could simply smooth the jitter, but then the hand tracking would not be as responsive when the performer *wants* to move quickly.

Figure 2: The Moore Neighbourhood of a pixel C



is circled, starting from the previous point on the contour, to find a third point. The process continues until we return to the starting point.

Finally, we identify fingertips in the image. Fingertip candidates are all points appearing on both the convex hull and the contour. In addition to the actual fingertips, the set of candidates will contain points along the wrist and palm. To eliminate them, we score each candidate by taking the points eight spots to the left and right along the contour from each candidate, and considering the angle they form with the candidate point. Candidates with an angle greater than $\pi/3$ are removed. There may also be several candidate points on the same fingertip; in this case we take the candidate with the best score (i.e. lowest angle). If we identify the same number of fingertips in four consecutive frames, we recognize it as a gesture.

4 Results

We found that our implementation of the hand tracking and gesture recognition algorithms operated smoothly at 30 frames per second, and were successful in ordinary playing conditions. A video demonstration is available at http://youtu.be/GUCRyYfLES4.

One limitation of our algorithm is that segmentation by volume is quite naive. If the hands are held near the body or other objects, then those objects will be included in the segmentation volume and treated as part of the hand. To handle these cases, we need to filter out all unconnected pixels in the volume.

A second limitation is that we do not take advantage of depth data when

Figure 3: Fingertip Identification

Green points are the convex hull,
white points are the contour, and
red points are the fingertips.

identifying fingertips, so our algorithms begin to fail when the hand is held nearly horizontally. Making better use of depth data could potentially increase our accuracy. For example, a promising technique in [5] treats the depth image pixels as nodes in a graph, with the relative depth as the cost between nodes, and examines path lengths along the hand to identify fingertips.

5 Future Work

In addition to addressing the limitations above, we think there many ways to continue to enhance the virtual theremin. The performer's hands still have many degrees of freedom which could control other options. For example, the pitch hand could move toward and away from the camera to control a vibrato effect, or flick up and down to synthesize attacks on the current pitch.

Another line of work involves synthesizing more than one voice at once. Using the Microsoft Kinect, we are also able to identify up to four different "players", so we could implement a multiplayer mode where several people perform together. We could also implement a recording mode that allowed a single performer to record and play back tracks, performing on top of them.

Finally, we would like to try virtualizing other instruments. A trumpet, for example, has only a few options for finger configurations which should

be quite recognizable using the algorithms we have already implemented (the second hand could indicate the tightness of the embouchure). Given an accurate enough camera, we fully intend to implement a rockin' air guitar.

References

[1] Jagdish L. Raheja, Ankit Chaudhary, and Kunal Singal: Tracking of Fingertips and Centres of Palm using KINECT. In *Third International Conference on Computational Intelligence, Modelling & Simulation (CIMSiM'11)*, pp. 248-252. IEEE, 2011.

[2] Yi Li: Hand Gesture Recognition Using Kinect. In *Third International Conference on Software Engineering and Service Science (ICSESS'12)*, pp. 196-199. IEEE, 2012.

[3] A.M. Andrew: Another efficient algorithm for convex hulls in two dimensions. In *Information Processing Letters*, vol. 9, no. 5, pp. 216-219. 1979. http://dx.doi.org/10.1016/0020-0190(79)90072-3

[4] R.L. Graham: An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set. In *Information Processing Letters*, vol. 1, no. 4, pp. 132-133. 1972.

[5] Hui Liang, Junsong Yuan, and Daniel Thalmann: 3D Fingertip and Palm Tracking in Depth Image Sequences. In *Proceedings of the 20th ACM International Conference on Multimedia (MM'12)*, pp. 785-788 ACM, 2012.

A Algorithms

A.1 Segmentation by Volume

```
define DepthPoint[] HandImage(SkeletonPoint handJoint,
                              DepthPoint[] Image) as
    for depthPoint in Image do
        skeletonPoint ← MapToSkeletonSpace(depthPoint)
        if |skeletonPoint.X - handJoint.X| < 15cm and
           |skeletonPoint.Y - handJoint.Y| < 15cm and
           |skeletonPoint.Z - handJoint.Z + 10cm| < 15cm
        then
            yield depthPoint
```

A.2 Monotone Chain Algorithm

```
define bool RightTurn(Point a, Point b, Point c) as
    v1 ← (b.X - a.X, b.Y - a.Y)
    v2 ← (c.X - a.X, c.Y - a.Y)
    cross ← v1.X × v2.Y - v1.Y × v2.X
    if cross >= 0 then
        return True
    else
        return False

define Point[] ConvexHull(Point[] points) as
    h ← Point[]
    n ← 0
    points ← points.OrderBy(X)
    for p in points do
        while n >= 2 and RightTurn(h[n-2], h[n-1], p) do n--
        h[n++] ← point
    min ← n + 1
    for p in points.Reverse() do
        while n >= min and RightTurn(h[n-2], h[n-1], p) do n--
        h[n++] ← point
    return h.Take(n - 1)
```

A.3 Moore Neighbourhood Tracing

```
define Point[] Neighbourhood(Point center, Point start) as
    x ← start.X
    y ← start.Y
    for i in [0,8)
        dx ← x - center.X
        dy ← y - center.Y
        if dx + dy != 0 then x ← x - dy
        if dx - dy != 0 then y ← y + dx
        if x in [0, Width) and y in [0, Height) then
            yield Point(x, y)

define Point[] Contour(Point[] points) as
    first ← points.First(HasDepth)
    current ← first
    previous ← (current.X - 1, current.Y - 1)
    do
         yield current
         neighbourhood ← Neighbourhood(current, previous)
         previous ← current
         current ← neighbourhood.First(HasDepth)
    until current = first or not HasDepth(current)
```

A.4 Fingertip Identification

```
define decimal Score(Point candidate, Point[] contour) as
    index ← contour.IndexOf(candidate)
    left ← contour[index - 8 % contour.Length]
    right ← contour[index + 8 % contour.Length]
    v1 ← Normalize(left - candidate)
    v2 ← Normalize(right - candidate)
    return ArcCos(DotProduct(v1, v2))

define Point[] Fingertips(Point[] hull, Point[] contour) as
    candidates ← hull.Intersect(contour)
                     .Where(Score(candidate) < π / 3)
                     .OrderBy(Score(candidate))
    while not candidates.Empty() do
        finger ← candidates.First()
        yield finger
        candidates ← candidates.Where(|finger - candidate| > 20)
```