The Kinect/Grasshopper Multi-touch Interface

Viswanathan Kumaragurubaran
Summer Internship Report
Autumn, 2011
The Kinect/Grasshopper Multi-touch Interface

This document explains the organization, algorithms and work flow of the Kinect + Grasshopper touch interface using the Kinect for Windows PC SDK.

**Required resources**

- PC with Windows 7 OS capable of dual display output
- Microsoft Kinect for Xbox with external power source and USB cables
- Projector capable of resolutions 1024x768 and higher
- Overhead mount for the Kinect
- Table with translucent top capable of transmitting light from the projector reflected off a mirror

**Hardware Setup**

The table acts as the interaction space wherein the required user interface is projected onto the table top using the projector installed beneath. The table top should typically be capable of allowing light from the projector to be dispersed on the surface to facilitate viewing as well as not exhibit specular behavior. This is important since the Kinect sensor operates by projecting infra-red light through the field of view and light bouncing off any reflective surface would be reproduced as noise or invalid depth data.

The depth sensor should be installed overhead with a field of view covering the table top. While the minimum distance of the sensor (distance to table) is restricted by the Kinect’s capabilities (typically around 0.75m), the maximum distance to table determines the accuracy of touch calculations. A typical depth value (and recommended) is 1.00 m for the specific table dimensions.


**Software Setup**

The source code (written in C++ and targeting the .NET framework 4.0) for the application can be checked out to any $path from the following location using SVN:

`file:///L:/admin/Tech Studio/Misc Projects/Kinect`

Three software executable files are located at:

`$path\SDK\src\simple\Release`

- SurfaceCalibration.exe
- ExtentsCalibration.exe
- MissionMode.exe

The executable files should be run in the following order (one at a time):
1. **SurfaceCalibration.exe** - necessary for depth estimation across the table top. It may be useful to push the calibration window to the table top display in order to track the progress of surface calibration.

Once the executable completes, a dump file called `DepthAverage.dump` is created in the `Release` directory containing the depth information in millimeters for each of the 640x480 pixels.

2. **ExtentsCalibration.exe** - is required to map the 2D table top space to the projected Grasshopper canvas. Run this executable and move the display window to the table top. This executable displays a square pixelated area on the display which acts as the target point to be mapped onto the 2D depth space (which is normally restricted to a subset of the table space). Once the square is displayed (starting bottom-right and moving clockwise for each corner, ending in the center), the user is expected to touch and hold the square for a few seconds until it moves to the next calibration location. This continues until all the 5 points are displayed and logged by the calibration utility.
At the end of the process, a data file called Calibration.dat is created in the Release directory containing the co-ordinates logged during the extents calibration process.

The controls in the UI are required to adjust the touch parameters and thresholds during image processing. This section displays the frame rate, provides control sliders to modify the DSurface, Finger and Mean filter threshold value. The filter kernel size can also be modified using the text control box. The default values in the UI are usually sufficient for a typical use case.

3. MissionMode.exe - this is the actual touch and image processing executable that processes touch inputs, identifies and translates the touch inputs as gestures and mouse events on the Grasshopper canvas. This executable should always be open and running to enable touch gestures tracking. The window displays the touch points at 30 frames per second.
Once the calibration procedure is complete, the Grasshopper canvas can be pushed to the table top display and the interaction process can be started.

**Gesture Recognition and Library**

Refer `\titan\lnm\admin\Tech Studio\Misc Projects\Kinect\Project Planning\Gesture Vocabulary.xlsx`

**Implementation and Code walkthrough**


The basic steps in the software (a typical Mission mode data flow) can be summarized by the following flowchart:

![Flowchart](image)

For each mode of operation (SurfaceCalibration mode, ExtentsCalibration mode and Mission mode), the data flow is different.
Gesture and Emulation Library Classes

The Gesture and Emulation library is contained in the classes defined in TouchGestures.h.

```cpp
class GestureLibrary {
public:
    // Gesture detect function
    IMAGE Gelib_DoSurfaceCalibration( USHORT *rawDepth, USHORT Frames);
    IMAGE Gelib_ReadSurfaceCalibration( void );
    IMAGE Gelib_CalcTouchImage( USHORT* dBuffer, IMAGE SurfaceVal, POINT S, POINT L, UINT32 DSURFACE_TOL, UINT32 FINGER );
    IMAGE Gelib_MeanFilter( IMAGE Raw, UINT32 Threshold, BYTE KERNELSIZE );
    IMAGE Gelib_Dilate( IMAGE B );
    IMAGE Gelib_Erode( IMAGE A );
    IMAGE Gelib_CCL8( IMAGE C );
    PARTICLES Gelib_ParticleAnalysis( IMAGE T );
    CALIBRATION Gelib_CalibrateExtents( CALIBRATION C );
    POINT Gelib_TransformToTablespace( POINT P, CALIBRATION CAL );
    float Gelib_TwoFingerZoom( PARTICLES Z, float PreviousZoomDistance );
    POINT Gelib_PanCanvas( PARTICLES PC, POINT PreviousPoint, bool KeyboardMode );
    POINT Gelib_DragWireAdd( PARTICLES DW, POINT PreviousPoint );
    POINT Gelib_NpointAvg( BYTE N, POINT M );
};
```

```cpp
class EmulationLibrary {
public:
    void Emulib_SetMousePos( LONG x, LONG y );
    void Emulib_MouseScroll( int Direction, POINT Midpoint );
    void Emulib_RightClick( void );
    void Emulib_RightClickHold( void );
    void Emulib_RightClickRelease( void );
    void Emulib_LeftClick( void );
    void Emulib_LeftClickHold( void );
    void Emulib_LeftClickRelease( void );
    void Emulib_KeyDown( void );
    void Emulib_KeyShift( void );
    void Emulib_KeyShiftRelease( void );
    void Emulib_Key_Control_Alt_O( void );
    void Emulib_Key_Alt_F4( void );
};
```

Surface Calibration Mode

The ‘SurfaceCalibration’ mode is responsible for modeling the table surface over multiple frames and a fixed spatial resolution of 640x480 depth points. The process does not calculate depth information adaptively. Hence, any objects placed on the table surface during calibration will be modeled as being part of the touch space. It is important to keep the field of view devoid of interference from direct light sources and objects (even a few seconds of interference would yield result in errors of ~10% in calibration). The methodology of modeling the Surface and extracting touch information is explained in the link [http://research.microsoft.com/en-us/um/people/awilson/publications/WilsonITS2010/WilsonITS2010.html](http://research.microsoft.com/en-us/um/people/awilson/publications/WilsonITS2010/WilsonITS2010.html)

SurfaceCalibration mode involves a single step as soon as the depth buffer is ready and locked. This step contains the routine (Gelib_DoSurfaceCalibration) that translates the depth values to millimeters and stores the cumulative sum over a period. The number of frames to track the depth values (defined by DUMPFOR parameter) and the frame interval (defined by DUMPEVERY parameter) are specified in the header file TouchGestures.h. The routine also enables a progress bar to indicate the calibration progress. Once the depth values are accumulated, the final depth values for each pixel (one of 640x480) are calculated by finding the average values (dividing each pixel accumulated value by DUMPFOR). The average depth values are stored in the file DepthAverage.dump (with semicolon-space-newline delimitation).

The SurfaceCalibration.exe application may be terminated as soon as this file is generated (or as soon as the progress bar indicates completion).
ExtentsCalibration Mode

During the ‘ExtentsCalibration’ mode, the system works in partial mission mode. Touch gestures are processed but not translated and emulated as mouse events. Instead, touch points are recorded for a certain time period and the corresponding co-ordinates are logged in a file.

The data flow is shown below:

Upon initialization and depth buffer filled event, the first step is to read the surface calibration depth values as an IMAGE structure from the stored file ‘DepthAverage.dump’. The IMAGE data structure is defined as a 2D unsigned short integer array with sizes specified as DWIDTH and DHEIGHT (640, 480).

```
typedef struct tagIMAGE {
    USHORT Img[DWIDTH][DHEIGHT];
} IMAGE;
```

The image analysis and processing routines are similar to that of the MissionMode and will be explained in that section. The additional function `CALIBRATION Gelib_CalibrateExtents( CALIBRATION C )` displays and logs the touch points in the sequence. The five points are iterated through a counter (cIteration) and the calibration mode swaps to regular mode (qswap = 0) once the calibration is completed.
The `Calout` data structure contains the calibration reference points. These points are stored in the `Calibration.dat` file. The data flow switches to touch display mode hence forth (displays transformed touch points and does not translate and emulate mouse events).

**Transforming touch points:** An affine transform of the touch points is performed to translate the touch points from the depth space to the table space.

The depth space is defined as the 2D region as seen by the Kinect depth camera (this is different from the 2D video space seen by the Kinect). This is expected to be a subset of the table space which is the physical 2D region of the glass top over the table. The transforms applied are:

1. Translation in X,Y direction ($T[C0(x,y)]$) towards and relative to the table space origin) of the point $C0$
2. Scale in X,Y direction ($S[M(x,y)]$) towards the upper boundaries of the table space

The above process determines factors to calculate the transformation offset and scale factors. The translation offset and scale factors are then applied to any incoming touch particle set over a loop.
The `gestureCanvas` image is necessary for debugging and visualizing the touch points in real time by assigning the gesture points to a square 5x5 super pixel.

**Mission Mode**

The Mission Mode is the run time mode that tracks touch gestures in real time and translates it to mouse events for the Windows 7 OS.

**Reading Surface Calibration Values**

During the system boot-up (`Frames = 0`), after the global run time variables are initialized, the surface calibration values are read off of the `DepthAverage.dump` file.

```c
for(int c=0;c<4;c++) {
  if(Calout.coordinates[c][0] < Small.x) {
    Small.x = Calout.coordinates[c][0];
  }
  if(Calout.coordinates[c][1] < Small.y) {
    Small.y = Calout.coordinates[c][1];
  }
  if(Calout.coordinates[c][0] > Large.x) {
    Large.x = Calout.coordinates[c][0];
  }
  if(Calout.coordinates[c][1] > Large.y) {
    Large.y = Calout.coordinates[c][1];
  }
}
```

`gSurface` is an image data structure that holds the average depth values of the table surface obtained during calibration. Following this, the calibration reference co-ordinates obtained from the ExtentsCalibration Mode is read back from the file `Calibration.dat` and stored in the structure `Calout`. For each calibration co-ordinate obtained from the file, an envelope bounding box (rectangular) is applied and the co-ordinates are redefined. This is done to adjust for any rotational shifts in the Z-axis during calibration (note: this is not a transform, but a linear extrapolation).

```c
for(int pa = 1; pa <= Part.Particles; pa++) {
  gTouch.x = Part.coord[pa][4];
  gTouch.y = Part.coord[pa][5];
  POINT TransformPt = mGestureLib.Gelib_TransformToTablespace(gTouch, Calout);
  Part.coord[pa][4] = (USHORT) TransformPt.x;
  Part.coord[pa][5] = (USHORT) TransformPt.y;
  for(int fx=0; fx<5; fx++) {
    for(int fy=0; fy<5; fy++) {
      gestureCanvas.Img[(USHORT)(TransformPt.x) +fx-2][(USHORT)(TransformPt.y) +fy-2] = 0xFF;
    }
  }
}
```

**Calculating Touch from Depth**

Following the above operations of reading the calibration files and loading it in the run time memory variables, the conversion of depth image to touch is performed by calling:

```c
prefilterImg = mGestureLib.Gelib_CalcTouchImage(pBufferRun, gSurface, Small, Large, this->DSURFACE_TOL, this->FINGER);
```

The output of the `Gelib_CalcTouchImage` function is an image structure containing binary touch information (a 640x480 array of 0’s and 255’s). The function takes inputs of the buffer containing depth information passed on to the application from the SDK as a memory pointer, bounding box co-ordinates
for the calibration points (this is required to exclude points outside the touch space), the constants DSURFACE_TOL and FINGER passed through the GUI.

The touch image is calculated by first reducing the depth buffer and extracting the masked depth values (by a logical bitwise &0x0ffff). The classification of depth values as touch is explained in http://research.microsoft.com/en-us/um/people/awilson/publications/WilsonITS2010/WilsonITS2010.html

The values of Dmax and Dmin are calculated as shown in the code snippet above. Dmax is the maximum value of depth beyond which any value of depth is classified as not belonging to the touch range. Dmin is the minimum depth value below which depth values do not count as meaningful touch values. The small range of values between Dmax and Dmin is classified as a valid touch point. (Note: To avoid spurious calibration parameters to be counted in the process, any value of Depth equaling 0 or greater than 1200 is classified as a non-touch point). Dmax can be assumed to be the actual surface value as calculated during the surface calibration procedure. However, owing to temporal depth noise, a tolerance parameter (DSURFACE_TOL) is included in the calculation of Dmax. The thickness of the finger dictates the Dmin calculation. We include the parameter FINGER to determine the Dmin value.

2D Spatial Depth Filter

As recommended by A. Wilson, a 2D separable mean filter (without weights) is implemented to remove boundary noise during interactions. The kernel size is fixed at 9x9 (although this is configurable using the GUI). A binary threshold is applied following the two stages of filtering. The threshold is adjustable from the MissionMode GUI. A value of 96 to 128 worked well at the time of testing.
Connected Component Labeling

The process of identifying blobs of connected particles (in this case, separate one finger from another in the touch space) and labeling them progressively for each frame is known as connected component labeling. The algorithm used here is 8 connected (Moore Neighborhood) component labeling. Refer [http://en.wikipedia.org/wiki/Connected-component_labeling](http://en.wikipedia.org/wiki/Connected-component_labeling) for the algorithm implementation and discussion.

The CCL algorithm is run in two passes. The first pass labels each valid pixel incrementally depending on the 8-connectivity rule. The second pass creates an equivalence table for labels that are associated with the same particle but have a different primary value (a primary value is the dominant value in the particle).

```csharp
LabelImg = mGestureLib.Gelib_CCL8(postFilterImg);
```

The connected-8 kernel is defined as:

```
(1,-1) (0,-1) (1,-1)
(-1,0)  (0,0)  (-1,0)
```

**First Pass:** For each pixel (represented by the solid black cell), the surrounding connected pixels are compared with the current pixel. The smallest label above 0 in the connected 8 kernel is assigned as the new label for the current pixel. If the smallest label in the kernel cannot be found (probably connected by zeros), then the pixel value is assigned a new label which is defined as a counter.

**Second pass:** The first pass output is rerun to derive the equivalence table. The equivalence table associates boundary particles that may have a different label than the primary value of the particle. The same procedure of 8-connecting is applied and the table structure is formed based on the variance in pixel values between the current pixel and the surrounding pixels. Once the association is derived, the equivalence table is normalized and the frame is redrawn. The resulting output is a well-connected and a uniquely labeled blob set.

Particle Analysis

```csharp
Part = mGestureLib.Gelib_ParticleAnalysis(LabelImg);
```

Particle analysis deals with performing morphology and intensity based parameter measurements on binary images. Typical operations involve measuring areas, lengths, coordinates, chords and axes, shape equivalence and shape features. Before particle analysis, Histogram based Thresholding and equalization is done to remove small particles and noise. Currently, the threshold is a fixed parameter of 32, which means that any particle with less than 32 pixels area is ignored for gesture calculations.
We limit the particle analysis operations in this application to a “bounding-box and center point” approach. More accurate measurements are unnecessary considering the errors in touch registration. For each particle identified as a label, the bounding box and center values are found as the maximum and minimum values in X and Y direction.

```c
for( int y = 0 ; y < DHEIGHT; y++ )
{
    for( int x = 0 ; x < DWIDTH; x++ )
    {
        USHORT px = P.Img.Img[x][y];
        if(px > 0 && px <=NUM_PARTICLES) {
            if(x < P.coord[px][0]) //minX
                P.coord[px][0] = x;
            if(x > P.coord[px][1]) //maxX
                P.coord[px][1] = x;
            if(y < P.coord[px][2]) //minY
                P.coord[px][2] = y;
            if(y > P.coord[px][3]) //maxY
                P.coord[px][3] = y;
        }
    }
}
for(int pa = 1; pa <= P.Particles; pa++) {
    P.coord[pa][4] = (P.coord[pa][0] + P.coord[pa][1])/2;
    P.coord[pa][5] = (P.coord[pa][2] + P.coord[pa][3])/2;
}
```

The particle analysis structure (P) returns the measured values for each valid particle back to the application.

**Detecting Gestures and Emulating I/O Device Actions**

The particle structure is used as the input to the gesture filter switch-case structure. Based on the number of particles detected (label count from the particle analysis routine), the gesture filter redirects I/O control (typically mouse actions or keyboard events) to the corresponding functions. These functions process gestures temporally and spatially, hence the accuracy of gesture detection depends on well-trained coefficients (these coefficients are currently hard-wired in the source code. These can be exposed as controls during the Surface Calibration process or as a completely different calibration/training process. Coefficients maybe determined as OTP values (hard-wired), dynamic and linear values or intelligence based values such as using neural networks). At the time of writing, up to 5 particles per frame are handled by the gesture filter (up to five fingers or five cases in the switch-case structure), although there is no limit to the number of particles (NUM_PARTICLES parameter) that the particle analysis method can process and the gesture filter can handle.

Note: For multi-user interactions, it may be necessary to process multiple particles through deeper particle measurements (palm recognition etc.). This would also require the Windows Multipoint SDK for multi-mouse interactions.

The gesture detection mechanism is designed in a way to facilitate smooth hand off between gestures and gesture fade-in and fade-out. As an example, there may be cases where a two-finger zoom gesture may transition and fade-out from 2 particles to 1 particle followed by 0. These transitions have been accounted for in the gesture tracking mechanism.
**Idle mode** - During the idle mode, the gesture filter and class variables are initialized and the filter waits for a valid gesture particle set. Any value of the number of particles above 1 is assumed to be a valid gesture. In addition, all emulation events are reset (mouse buttons are released). In order to avoid spurious gesture resets to case 0 (noisy frames), a time out counter is set. For any value of the time out beyond 3 (3 frames) the previous gesture is assumed to have completed. Certain gestures like Select/Move have special time out values that are unique to the behavior of those gestures. Every gesture case also keeps track of the previous gesture in the ‘pGesture’ variable (similar to the function of a state machine’s state variable).

**Select/Move mode** - This mode (case 1) handles gestures such as Select/Move components and delete component as well as (left and right) mouse click gestures.

**Handling Mouse Right Click**

```csharp
if(dtTimeout >= 4 && dtTimeout < 10) {
    cGesture = 1;
    mEmulationLib.Emulib_SetMousePos(Part.coord[1][4], Part.coord[1][5]);
    mEmulationLib.Emulib_RightClick();
}
```

The right click event is handled based on a timed gesture response. This is required to differentiate between gesture transitions as well as other single particle gestures. The right click event is determined based on the dtTimeout counter value.

**Handling component select, move and delete**

```csharp
sePreviousPt = mGestureLib.Gelib_SelectElement(Part, sePreviousPt, KBmodeON);
```

The Gelib_SelectElement function processes all touch gestures that require translation to corresponding events on the Grasshopper canvas. The select gesture is valid when the spatial difference between the current touch point and the previous touch point (the previous frame) is less than a tolerance value defined by SE_TOLERANCE. The tolerance value is set to a strict 1 currently. To distinguish between spurious or stray touch points, we track the gesture for stability. If the current touch point is stable for 10 frames, the gesture is locked down by setting the mouse pointer position at the translated position and a 'mouse left-click and hold' event is sent to the operating system. For the 11th frame (special case), the mouse pointer is jittered for a pixel in the Y direction to ensure that the component information (usually achieved by mouse hover) is displayed.

The component move gesture is opposite in implementation to the move gesture. In this case, we check if the difference in coordinates of the current and previous touch point is above a certain threshold (SE_TOLERANCE + 2) and proceed to send the ‘mouse left-click and hold’ event to the OS. The delete component gesture is also implemented under the same category wherein we check if the difference in current and previous touch coordinates is above the delete threshold of DE_TOLERANCE. Once the threshold limit is hit, the delete gesture identifier proceed to a binary encoded style state machine that tracks for successive gestures (back and forth movement of the finger). After 3 successive move gestures (back and forth), the gesture recognition loop proceeds to send the ‘Delete’ key event to the OS.

**Zoom mode** - The two finger zoom gesture is similar in implementation to the Select/Move mode.

```csharp
pZoomDistance = mGestureLib.Gelib_TwoFingerZoom(Part, pZoomDistance);
```
On function entry, the floating point distance between the two fingers is calculated. This is required to differentiate between a zoom in and zoom out gesture. For each frame, the current distance between the fingers and the previous frame’s distance between the fingers are compared. If the difference is positive, the gesture is a zoom-out. In this case, the mouse-scroll event is sent to the OS with a positive WHEEL_DELTA. A zoom counter in each case is incremented to verify switching between one zoom-mode to the other.

**Drag-wire and add mode** - This is a 3 finger gesture similar to the move gesture with the additional shift key event included with the mouse left click hold event.

```c
DWPrevPt = mGestureLib.Gelib_DragWireAdd(Part, DWPrevPt);
```

It is important to identify the optimum relative positioning of the 3 fingers on the touch space. For natural interaction the index finger, middle finger and the thumb are used. The middle finger shall be used as the actual pointer position for the gesture. To calculate the validity of this condition, a conditional check is performed on the 3 coordinates passed to the function. Following the classification of the touch points, the final touch point (the middle finger) is extracted and checked for motion across frames using the same method described for the move gesture. The shift key is also held in addition to the Left-Click-Hold event. The 3 finger hold gesture is implemented as well for the sake of uniformity in behavior.

**Pan Canvas mode** - Cases 4 and 5 are used in tandem for the pan mode. The pan gesture is a 5 finger gesture, however there may be instances where two fingers combine to one particle and yield 4 particle gestures. In those cases, the gesture filter operates in the same pan canvas mode. The start of the gesture however requires 5 unique particles.

```c
PanPrevPt = mGestureLib.Gelib_PanCanvas(Part, PanPrevPt);
```

The actual mouse position is determined as the average X and Y position of each of the (4 or) 5 particles passed as the parameter to the Pan Canvas function. The pan canvas gesture is easily susceptible to temporal noise owing to the inconsistency in the ability to determine a fixed mouse pointer coordinate. To nullify this problem, a 2 point frame average of the coordinates is calculated as the final pan point. Whenever the current touch point changes position, the movement delta values in X and Y directions are calculated. If the delta values exceed a threshold (PAN_TOLERANCE), the pan gesture is simulated by a mouse right-click and hold event.