3D INTERACTIVE INTERFACE USING A KINECT SENSOR

A Design Project Report
Presented to the Engineering Division of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering (Electrical)

By: Anandram Sundar(as2454), Adarsh Kowdle

Advisor: Prof. Tsuhan Chen

Degree Date: May, 2011

Abstract

Master of Electrical Engineering Program
Cornell University
Design Project Report

Project Title: 3D INTERACTIVE INTERFACE USING A KINECT SENSOR

Author: Anandram Sundar(as2454@cornell.edu)

Abstract:

The aim of the project is to develop an intuitive 3D interface for modeling applications. For this purpose we performed hand segmentation based on the depth map of the user and tracked the hands across frames. Based on the tracking information we detect and recognize various gestures. For each of the gesture recognized we then perform mouse actions to control a 3D modeling software.

Report Approved by

Project Advisor: Dr. Tsuhan Chen

Executive summary

3D modelers allow users to create and alter models. The entire process of 3D modeling can be made intuitive and easier with the help of an interactive 3D interface. The aim of the project is to develop a 3D interactive interface to enable 3D modeling.

For this purpose we use a Kinect sensor to get the depth map of a user which is then fed to openNI and NITE middleware components. These give us the hand points for multiple hands with a unique persistent id. We use this information to track how the hands move across frames with respect to each other as well as in the absolute space domain. We also keep track of the hand size of the current frame by segmenting the hand in the frame.

Using this tracking information, we identify different gestures like zoom-in (hands moving apart), zoom-out (hands closing in), click & drag (hand closed and move), click (push), swipes and other complex gestures based on these.

Once the gesture has been recognized we then control the 3D modeling software. For this we perform actions based on mouse events to simulate the actual mouse controls executed to control the application. For this purpose we use google sketchUp to demo our gesture recognition system. Though the actions performed are specific to this software, the gestures are universal.

The results were satisfactory. The cursor movement was still jerky after smoothing across 5 frames. The gestures were recognized to a good degree of accuracy. The only problem is the tracking is very good for uneven motion of the hands and there is a lot of calibrating needed.

There is still room for a lot of improvement. Not all the data available from the sensor has been used. More gestures can be performed based on finger tracking once the hand has been segmented.

Contents

Topic	Page
1. Introduction	5
2. Previous Work	5
3. Initial Approach	6
4. System Components	6
4.1. Kinect Sensor	6
4.2. Middleware components	8
4.2.1. OpenNI	8
4.2.2. NITE	10
4.3. User App	13
4.3.1. Calibration	13
4.3.2. Gestures	14
4.3.3. Functions	16
4.4. Basic Assumptions	19
5. Experiments and Results	20
6. Conclusions	21
7. Future Work	21
8. References	22
9. Appendix	23
A. System installation	23
B. Code	24

1. Introduction

3D modelers allow users to create and alter models via their 3D mesh. Many 3D modelers are designed to model various real-world entities, from plants to automobiles to people. The medical industry uses them to create detailed models of organs. The movie industry uses them to create and manipulate characters and objects for animated and real-life motion pictures. The video game industry uses them to create assets for video games. The science sector uses them to create highly detailed models of chemical compounds. The architecture industry uses them to create models of proposed buildings and landscapes. The engineering community uses them to design new devices, vehicles and structures as well as a host of other uses.

There are typically many stages in the "pipeline" that studios and manufacturers use to create 3D objects for film, games, and production of hard goods and structures. Users can add, subtract, stretch and otherwise change the mesh to their desire. Models can be viewed from a variety of angles, usually simultaneously. Models can be rotated and the view can be zoomed in and out.

With increasing importance in 3D modeling and numerous functionalities available, an interactive 3D interface would make the task of 3D modeling not only intuitive but much easier too. The idea is to come up with gestures which the users will find natural to use for performing a specific task. With this in mind the aim of the project is to develop a 3D interactive interface to enable 3D modeling using a Kinect sensor.

2. Previous Work

People have used the kinect sensor to perform various tasks ranging from a simple mouse control to complex functions like autonomous robotic explorations. There have been many projects to make the development using kinect easier. There are support software which provide simple functions like hand-point generation to complex functions like user pose detection and scene analyses. People have had the idea of integrating the sensor with modeling software. They have used the sensor to get a 3D model for AutoCAD (by Kean Walmsley) and

3D manipulation using both the mouse (2D cursor) as well as the hand (3D cursor). There has also been a 3D modeling hack using power gloves to facilitate the process (by Sebastian Pirch). But there has been no 3D modeling/manipulation application just using the kinect (to the best of my knowledge).

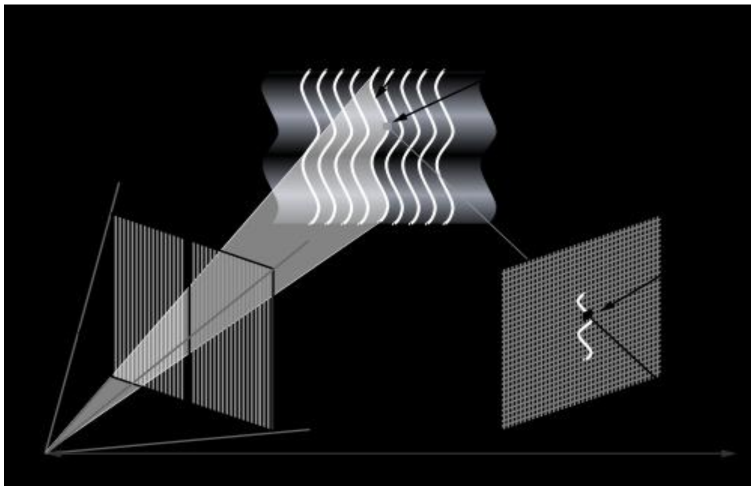
3. Initial Approach

The system originally used CLNUI drivers for the Kinect sensor and started off the sample program provided in the installation. Here we segmented the depth map using a distance threshold and concluded everything which crosses this imaginary plane to be a hand point. We then clustered the available points based on 8-connected component analysis and found the corresponding hand clusters and cluster size. But since the depth map wasn't very accurate, there were holes in the clusters thus leading to multiple clusters instead of a single one. We then changed the approach to cluster hand points based on distance between hand points being less than a particular threshold. This improved the performance, but still gave multiple clusters. Thus we changed our approach to using the middleware components to detect the hand points and focused on recognizing gestures.

4. System Components

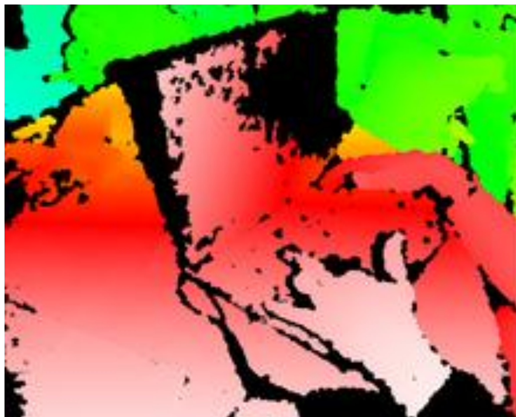
4.1. Kinect sensor

Kinect is based on range camera technology by PrimeSense, which interprets 3D scene information from a continuously-projected infrared structured light. This 3D scanner system called Light Coding employs a variant of image-based 3D reconstruction.



The Kinect sensor is a horizontal bar connected to a small base with a motorized pivot and is designed to be positioned lengthwise above or below the video display. The device features an RGB camera, depth sensor and a multi-array microphone. The Kinect sensor's microphone array enables acoustic source localization and ambient noise suppression.

The depth sensor consists of an infrared laser projector combined with a monochrome CMOS sensor, which captures video data in 3D under any ambient light conditions. The sensing range of the depth sensor is adjustable.



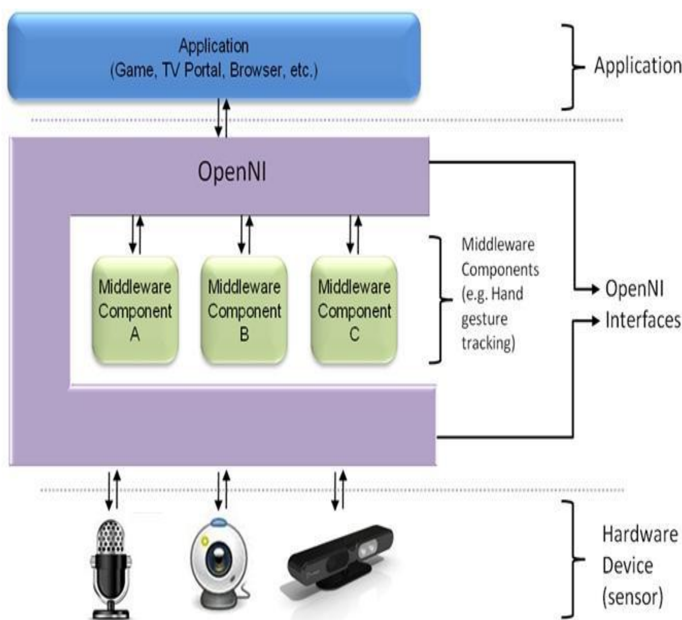
The depth map is visualized here using color gradients from white (near) to blue (far). The Kinect sensor outputs video at a frame rate of 30 Hz. The RGB video stream uses 8-bit VGA resolution (640 × 480 pixels), while the monochrome depth sensing video stream is in VGA resolution (640 × 480 pixels) with 11-bit depth, which provides 2,048 levels of sensitivity. The Kinect sensor has a practical ranging limit of 1.2–3.5 m (3.9–11 ft). The sensor can maintain tracking through an extended range of approximately 0.7–6 m (2.3–20 ft). The sensor has an angular field of view of 57° horizontally and 43° vertically, while the motorized pivot is capable of tilting the sensor up to 27° either up or down. The horizontal field of the Kinect sensor at the minimum viewing distance of ~0.8 m (2.6 ft) is therefore ~87 cm (34 in), and the vertical field is ~63 cm (25 in), resulting in a resolution of just over 1.3 mm (0.051 in) per pixel. The microphone array features four microphone capsules and operates with each channel processing 16-bit audio at a sampling rate of 16 kHz.

4.2. Middleware

4.2.1. OpenNI

OpenNI (Open Natural Interaction) is a multi-language, cross-platform framework that defines APIs for writing applications utilizing Natural Interaction. OpenNI APIs are composed of a set of interfaces for writing NI applications. The main purpose of OpenNI is to form a standard API that enables communication with both vision and audio sensors (the devices that 'see' and 'hear' the figures and their surroundings.) and vision and audio perception middleware (the software components that analyze the audio and visual data that is recorded from the scene, and comprehend it).

OpenNI supplies a set of APIs to be implemented by the sensor devices, and a set of APIs to be implemented by the middleware components. By breaking the dependency between the sensor and the middleware, OpenNI's API enables applications to be written and ported with no additional effort to operate on top of different middleware modules. OpenNI's API also enables middleware developers to write algorithms on top of OpenNI raw data formats, regardless of which sensor device has produced them, and offers sensor manufacturers the capability to build sensors that power any OpenNI compliant application.



The OpenNI standard API enables natural-interaction application developers to track real-life (3D) scenes by utilizing data types that are calculated from the input of a sensor. Applications can be written regardless of the sensor or middleware providers.

Abstract Layered View

Top: Represents the software that implements natural interaction applications on top of OpenNI.

Middle: Represents OpenNI, providing communication interfaces that interact with both the sensors and the middleware components, that analyze the data from the sensor.

Bottom: Shows the hardware devices that capture the visual and audio elements of the scene.

Production Nodes are a set of units that have a productive role in the process of creating the data required for Natural Interaction based applications. Each production node can use other lower level production nodes (read their data, control their configuration and so on), and be used by other higher level nodes, or by the application itself.

The production node types that are used by the application are:

Depth Generator: A node that generates a depth-map. This node should be implemented by any 3D sensor that wishes to be certified as OpenNI compliant.

Gestures Alert Generator: Generates callbacks to the application when specific gestures are identified.

Hand Point Generator: Supports hand detection and tracking. This node generates callbacks that provide alerts when a hand point (meaning, a palm) is detected, and when a hand point currently being tracked, changes its location.

The Context Object

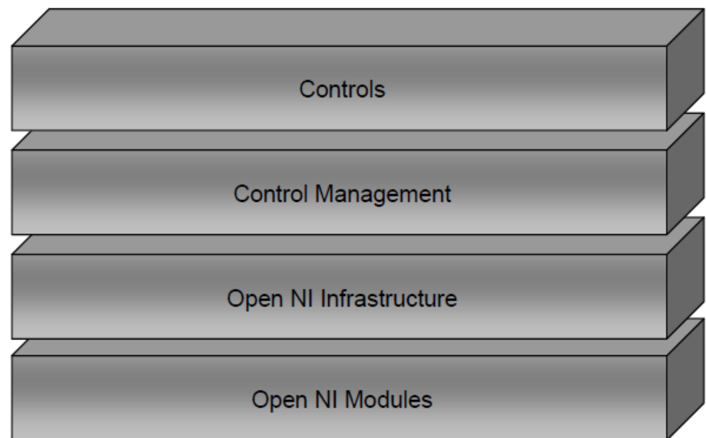
The context is the main object in OpenNI. A context is an object that holds the complete state of applications using OpenNI, including all the production chains used by the application. The same application can create more than one context, but the contexts cannot share information. For example, a middleware node cannot use a device node from another context. The context must be initialized once, prior to its initial use. At this point, all plugged-in modules are loaded and analyzed. To free the memory used by the context, the application should call the shutdown function.

4.2.2. NITE

NITE is a toolbox to allow application to build flows based on the user's hands movement. The hand movement is understood as gestures and is tracked, to provide 'hand points'. NITE works over OpenNI.

NITE works with the OpenNI framework and consists of the following layers:

OpenNI Modules: The OpenNI modules supported by NITE are Gesture Generator and Hands Generator.



Control Management: Receives a stream of points and routes them to the appropriate NITE control.

Controls: Each control receives a stream of points and translates it into a meaningful action specific to that control. The control then calls a callback from the application that can change the current active control in the control management layer, thus defining the application flow.

A session is a state in which the user is in control of the system using hand points. While in the session, hand points are tracked, and have persistent IDs.

There are 3 possible session states - Not in session, In session and Quick Refocus. A session starts when the user performs a focus gesture which is a click or a wave.

XnVSessionManager is the NITE object that handles the session. It turns on and off the gesture recognition when needed.

When in 'in session' state, and there are active hand points, it groups them together and sends them to interested controls.

Controls are objects that receive specific types of data, and perform some action on that data. Controls are actually listeners, getting data every frame.

The most common controls in NITE are the point controls.

Point controls receive the current active hand points from some source and try to understand these points as some action.

Controls have events, to which callbacks can be registered. When a control recognizes the movement it is supposed to, it will call all registered events.

Point Controls

Point controls allow registration to when new points are created, existing points move and when points that existed disappear. When a Point Controls is connected to a Session Manager, it gets the active hand points each frame, and can perform its action.

Each control allows registration to control-specific events. The registered callback functions will be called when that event happens.

List of controls used the system

Push Detector

This point control tries to recognize hand point movement as a push motion, which is a continuous motion towards the sensor and back again.

Swipe Detector

This point control tries to recognize hand point movement as a swipe motion, either up, down, left or right. A swipe motion is a short movement in a specific direction, followed by the hand resting. This was initially used in the system to detect planes and line gestures (compound gestures) but this was later removed as these detectors weren't reliable. The system has its own logic based on the history of the tracked points to detect these.

Wave Detector

This point control tries to recognize hand point movement as a wave motion. A wave is a number of direction changes within a timeout. By default 4 direction changes are needed to identify a wave.

Circle Detector

This point control tries to recognize hand point movement as a circular motion.

The Circle Detector needs a full circle in either direction in order to start generating output. Clockwise direction is considered the 'positive' direction, and anti-clockwise is considered the 'negative'.

Primary Point

All the aforementioned controls work on one of the hand points, not all of them. This point is the Primary Point. It is initially the first one recognized. If the primary point is no longer available (hand point isn't identified anymore), and other points exist, one of those points will take over as the primary point. The primary point is set by the session manager.

<u>NITE flow</u>

When NITE controls are connected directly to a Session Manager, it means that all the controls are always active, meaning they get hand points whenever there are any hand points.

A flow can be defined, so that controls receive data only when needed, when they are meaningful.

List of Flow objects used by the system

Flow Router

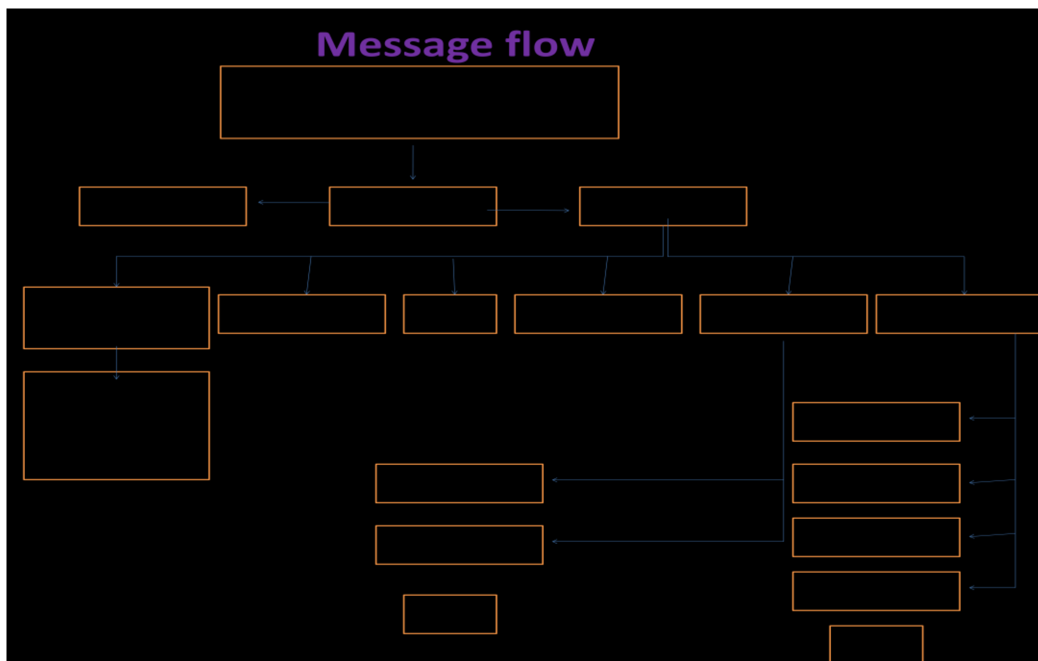
The flow router object sends all the data it receives to a single object that is connected to it. This single object may be changed. An object that is connected to the flow router it considered to be 'active', as it will receive the data. An object that was connected to the flow router but isn't anymore is considered to be 'inactive'. Initially these were used to recognize plane and line gestures by switching between listeners. This is also used to switch between calibrate and start listening modes.

Broadcaster

The broadcaster object sends all data it receives to all objects that are connected to it. Once the session has started and calibration is performed this object sends data to all the different listeners like the circle, wave, push and point detectors for recognizing gestures

4.3. User App

The system starts with a session initialization which looks for the focus gestures("click","wave"). Once it sees one, the system then does the user working space calibration. Once this is done the system then passes the sensor data to the broadcaster which then passes the data onto the various listeners to recognize gestures and perform various actions depending on what gesture was performed. The session ends if after a timeout occurs when no change in sensor data occurs. The program terminates when a key is hit.



4.3.1. Calibration

The calibration for the system requires some modification to the code to adjust to various users as well as runtime calibration which defines the user working space.

The user hand size varies and the click & drag function depends upon this. So before a particular user can start using the system, 4 parameters namely the far-distance and size of open hand at far-distance and the near-distance and size at near-distance have to be defined.

For the runtime calibration, the user first has to start the session with a push and then performs 2 pushes- one for the top left of the user space and one for the bottom right of the user space. This calibration is done so as to track the hand movement and map it to the cursor movement on the screen.

4.3.2. Gestures

There is a standard timeout of 2 seconds between and successive gestures. After this time the program calls the setback function to start listening for gestures afresh.

Click

The user just has to quickly move the hand forward and back. This triggers a push event to be thrown which is caught by push_pushed.

Erase

For this the user has to move his hand to the left and back multiple times(at least 5) which triggers an event to be thrown which is caught by the wave_detect.

Circle

For this the user has to move his hand in a circular fashion. This triggers an event to be thrown which is detected by the circle_detect.

Plane

For this the user has to move his hand to the right, down, left and then up in that order without giving too much time (2 secs) between the successive motions. This is detected in the point_update function and the corresponding actions are performed.

Line

For this the user has to move his hand to the right, down, and then up in that order without giving too much time between the successive motions. This is detected in the point_update function and the corresponding actions are performed.

Click and drag

For this the user closes his hand and moves the cursor hand. This function relies on the get_hand_size function to check the size of the hand. If it is below a threshold the system

performs a mouse left down and releases it when the hand is once again open. The threshold value is varied linearly for the distance of the hand point. This is because the hand appears smaller in the camera from a longer distance than from close up. The threshold also depends on a close to open hand size ratio, which I have approximated to .55. This means for a particular distance if the hand size returned by the function is less than .55 times the full hand size, it is considered close and hence a mouse down is performed.

Zoom out/in

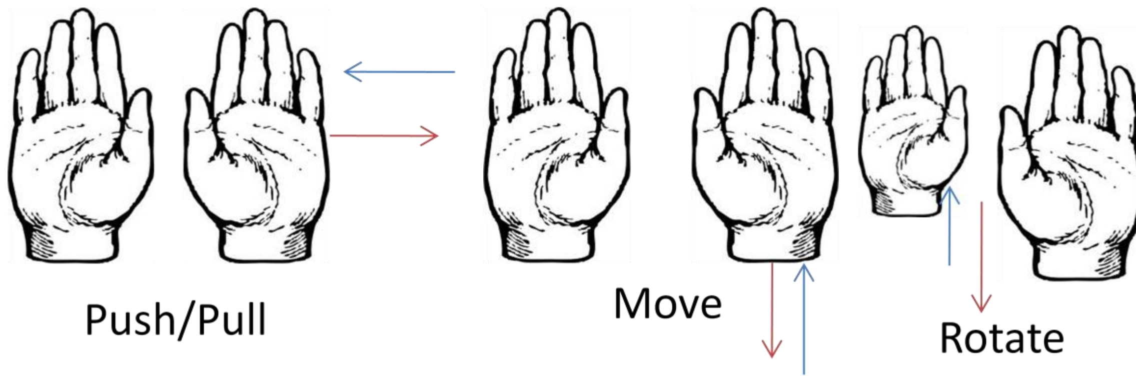
Here the user keeps the primary hand steady while moving the secondary hand towards/away from the primary hand in almost the same plane. These gestures have been mapped to going into and coming out of the push/pull mode in the Google SketchUp application.

Rotate X clockwise/anticlockwise

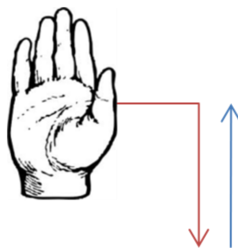
Here the user keeps the primary hand steady while moving the secondary hand up/downwards from the primary hand in almost the same plane. These gestures have been mapped to going into and coming out of the move mode in the Google SketchUp application.

Rotate Z clockwise/anticlockwise

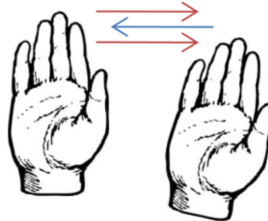
Here the user keeps the primary hand steady while moving the secondary hand front/back from the primary hand. These gestures have been mapped to going into and coming out of the orbit mode in the Google SketchUp application.



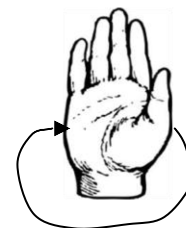
Grab/Click & Drag



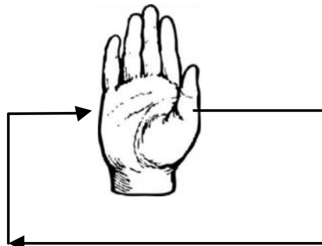
Line/cursor toggle



Erase/cursor toggle



Circle mode



Plane mode

4.3.3. Functions

Setback()

This function sets back everything to a state where there are no previous moves or no gestures in progress or detected. This is called when there is a timeout between successive swipes in recognizing a gesture or after action has been taken for a particular gesture or when hand points are lost.

Gethandsize()

This function gets the size of the hand from the depth map given the 3D position of the hand-point. It does this by assuming that all the pixels with a depth of ±40 of the given hand point are also points on the hand.

Check_swipe_()

These functions look at the positions of the primary hand point across multiple frames and checks if it is moving up/down/left/right consistently in these frames.

Check_rotate_()

These functions look at the positions of the secondary hand point across multiple frames and checks if it is moving up/down/front/back consistently in these frames along with the primary hand point being relatively stable.

Check_zoom()

This function looks at the positions of the secondary hand point across multiple frames and checks if it is moving left/right consistently in these frames along with the primary hand point being relatively stable.

Push_()

These functions move the cursor to a particular position on the screen (based on the mode buttons which are specific to the app-SketchUp), performs a click and returns the mouse cursor to the previous location.

Get_smoothed_()

These functions does a smoothing operation on the retrieved hand points by performing an averaging (LPF) to eliminate sudden jerks in the mouse movement which may be caused by the openNI modules by changing the primary hand point.

Session_start()

Sets the flow to the broadcaster so that all the various gesture listeners can start listening for the events. It also starts the session.

Push_pushed()

Performs a click at the current cursor location when it is not in calibration mode or performs calibration by storing the push positions as top left and bottom right of the user working space.

On_point_create()

Adds extra hand point and selects the primary hand point as the one that first came into view.

On_point_destroy()

This decreases the number of hand points by one and performs other operations like changing the mode back to the cursor mode and deleting history of the hand points and changing the primary hand point.

Circle_detect()

It listens for a circle event and changes the mode to circle mode if it detects one. This is disabled when the mode is in one of the 2 hand modes.

Wave_detect()

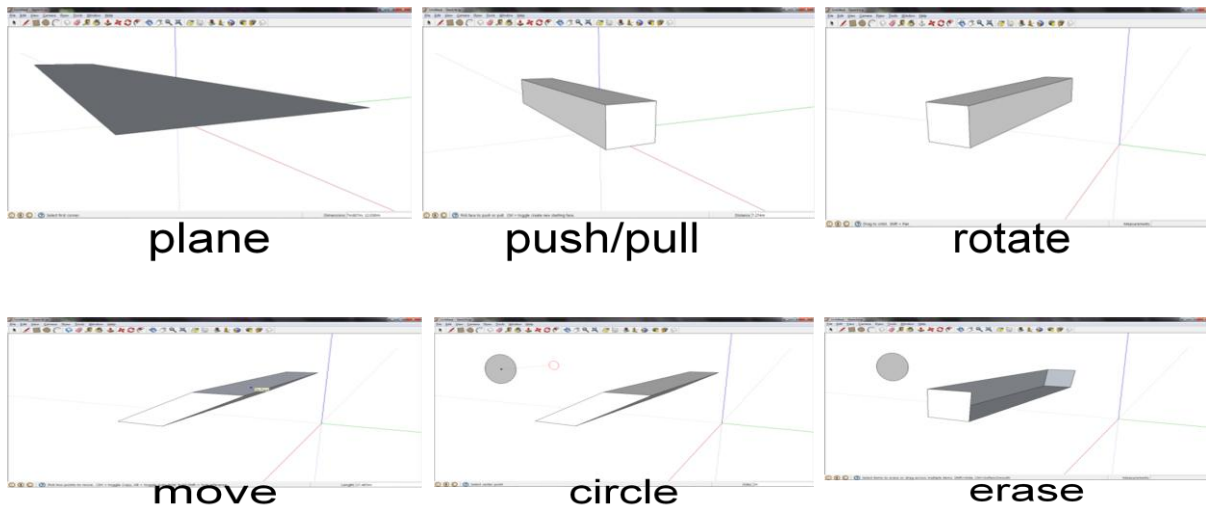
It listens for a wave event and changes the mode to erase if it detects one. This is disabled when the mode is in one of the 2 hand modes. It also acts as a toggle to switch between erase or cursor mode.

On_point_update()

This is the most important function in the entire system. It checks for the various possible gestures by calling the check_() functions and performs corresponding actions. It is also responsible for cursor movement based on the hand position. It also performs click and drag gestures. It also keeps updating the array which stores the history of the hand points.

Main()

This initializes the session and adds the various listeners to specific objects. It also instantiates the various generators like the hand and depth. It then keeps updating the context which contains all the sensor data every time there is a change in the depth.



4.4. Basic assumptions

- The controlling hand is inside the field of view, and is not occluded by other users and objects. Exiting the field of view may cause losing the control.
- Working distance of the controlling user should be 1m - 3.5m
- Hand that performs the gesture should be at a distance from your body.
- Palm should be open, fingers pointing up, and face the sensor.
- The movement should not be too slow or too fast.

- The “click” movement should be long enough.
- The “click” gesture should be performed towards the sensor.
- The “wave” movement should consist of at least 5 horizontal back-and-forth movements.
- Avoid touching other objects with your controlling hand while you are in control.

5. Experiments and results

I tried varying various system parameters like the number of frames across which to perform the gesture recognition, number of frames for smoothing the hand point and the open-to-close ratio of the hand size. For larger number of frame counts, the gesture recognition started performing poorly. The rotate gestures performed well even for frame counts of 15, but zoom-in/zoom-out all performed very badly for these counts. If the frame counts were very less, there were a lot of false positives because of improper tracking. 5 frames gave the minimum number of false positives while recognizing all the gestures.

For the smoothing parameter, too many frames introduced a lag in the system response while no smoothing meant too much jerky movement in the cursor. 5 frames seemed a good trade-off between these two.

The ratio of the hand size between closed and open was not varied too much. I checked in the range of 0.4 to 0.7 with step size of 0.05. If the ratio was too small sometimes because of the second hand being present in some of the gestures, the size does not go below the threshold even after fully closing. If it was too big the system gave false positives for click and drag. Even a slight difference in orientation of the hand when the hand moves across the frame gives false grabs. The system performance for click and drag was best for 0.55.

6. Conclusion

The overall system performance is satisfactory. The gestures are recognized most of the times when the conditions are suitable and the user adheres to the assumptions made. The erase, circle, push, orbit, push/pull and move are recognized every time. The click and drag works really well with proper calibrations of the user and size. It sometimes gives a false grab when the user changes the palm orientation as seen by the camera, thus giving a cross-sectional view and thereby decreasing the size of the hand. This makes the system think that the hand is closed. As for the plane, line modes they get recognized if the swipes are done cleanly with the palm facing the sensor the entire time and if they aren't too fast or too far apart (timeout).

The system does perform poorly under certain conditions. If another object is close to the hand, the hand point might move to the other object. It may happen that a hand point is lost. In general, very fast motions may cause tracking failure. Sometimes the swipes may not be recognized if there is too much variation in the depth of the hand that is being tracked. There might be false pushes because of bad tracking of the hand point. Even after smoothing the cursor movement is not very smooth. In some cases, overall tracking might be bad. Re-calibrating the user may resolve the problem. The depth map is unreliable when there is too much IR interference in the environment, for example in an open hall with sun light.

Though the actions performed after recognizing the gestures are specific to SketchUp, the system can be used to manipulate any app with minor changes to the code.

7. Future work

- Better hand segmentation to use it in a cluttered area
- Perform better smoothing to get reliable cursor movement
- Add confidence measures to the recognized gestures to increase the reliability of the system.

- Implement fingertip recognition to recognize more gestures
- Use multi-user calibrations to allow different users to use the app
- Provide a wrapper with throwable events after recognizing gestures to facilitate further application development, thus allowing for control of any 3D application based on a standard set of gestures.

8. References

- OpenNI User Guide.pdf
- NITE Algorithms 1.3.pdf
- NITE Controls 1.3 – Programmer's Guide.pdf
- http://msdn.microsoft.com/en-us/library/ms646260%28v=vs.85%29.aspx
- http://kindohm.com/2011/03/01/KinectCursorControl.html
- http://codelaboratories.com/kb/nui/
- http://en.wikipedia.org/wiki/Kinect
- http://asimmittal.net/blog/?p=121
- http://en.wikipedia.org/wiki/3D_computer_graphics_software
- http://sketchup.google.com/intl/en/index.html
- http://kinecthacks.net/

9. Appendix

A. System installation:

1) Install the primesense driver for kinect(mod).
2) Install openNI http://www.openni.org/downloadfiles/2-openni-binaries
3) Install NITE http://www.openni.org/downloadfiles/12-openni-compliant-middleware-binaries
4) Change the XML-configuration to include the primesense key and the correct VGA resolution

   < License vendor="PrimeSense" key="0KOIk2JeIBYClPWVnMoRKn5cdY4="/>

   < MapOutputMode xRes="640" yRes="480" FPS="30"/>

5) Check if the sample programs are working
6) Download the project folder onto the computer and execute*

*If you are copying only the code, set the dependencies in the project properties to include the library and include folders in the openNI and NITE installations.

B. Code:

```
#include <XnOpenNI.h>
#include <XnCppWrapper.h>
#include <XnVSessionManager.h>
#include <XnVPushDetector.h>
#include <iostream>
#include <conio.h>
#include <XnVSessionGenerator.h>
#include <XnVSwipeDetector.h>
#include <XnVFlowRouter.h>
#include <Windows.h>
#include <XnVCircleDetector.h>
#include <XnVBroadcaster.h>
#include <XnVWaveDetector.h>
#include <Windows.h>
#include <time.h>
#include <fstream>

#define FRAME_COUNT 5
#define TRAIL_COUNT 5
#define steady_thresh 10
#define FULLATNEAR 13000
#define FULLATFAR  2515
#define FAR 1500
#define move_thresh 20
#define zoom_thresh 8
#define NEAR 580
#define FTOCRATIO .55


//All for screen res 1280x800
#define CURSORX 18
#define LINEX 54
#define CURSORY 50
#define PLANEX 85
#define CIRCLEX 110
#define ERASEX 205
#define ROTATEX 418
#define PANX 450
#define PUSHX 300
#define MOVEX 330

enum Move{UP,DOWN,LEFT,RIGHT,NONE};
enum gest{ROTZCLOCK,ROTZACLOCK,ROTXCLOCK,ROTXACLOCK,ZOOMIN,ZOOMOUT,STOP};

using namespace std;

int tempcount=1;
int width = GetSystemMetrics(SM_CXSCREEN);
int height = GetSystemMetrics(SM_CYSCREEN);
bool grabbed=false;
bool g_detected=false;

time_t prevTime,currTime,lastTime;
Move prevMove=NONE;
gest currGest=STOP;
```

```
XnVFlowRouter g_flowRouter;
XnVPushDetector g_pushDetector;
XnVSwipeDetector g_swipeDetectorUp;
XnVSwipeDetector g_swipeDetectorDown;
XnVSwipeDetector g_swipeDetectorLeft;
XnVSwipeDetector g_swipeDetectorRight;
XnVBroadcaster g_broadcaster;
XnVWaveDetector wavedetector;
XnVFlowRouter planeRouter;
XnVFlowRouter lineRouter;
XnVSwipeDetector g_swipeDetectorUpline;
XnVSwipeDetector g_swipeDetectorDownline;
XnVPushDetector pushDetector;
xn::DepthGenerator depth;
xn::HandsGenerator hand;
XnVCircleDetector g_circle;

float handposx1=0,handposy1=0,handposx=0,handposy=0;
float handposx2=0,handposy2=0;
float hand_arr_x_1[TRAIL_COUNT],hand_arr_y_1[TRAIL_COUNT],hand_arr_Z_1[TRAIL_COUNT];
float hand_arr_x_2[TRAIL_COUNT],hand_arr_y_2[TRAIL_COUNT],hand_arr_Z_2[TRAIL_COUNT];
int xl=0,yl=0,xr=0,yr=0;



bool Calib=true;
bool cursor=true;
int numpoints=0;
int id=0;
int id1=0;

void setBack()
{
	g_detected=false;
	prevMove=NONE;
	prevTime=0;
	lineRouter.SetActive(&g_swipeDetectorUpline);
	planeRouter.SetActive(&g_swipeDetectorRight);
}

int gethandsize(XnFloat z,XnFloat x, XnFloat y)
{
	int size=0;
	const XnDepthPixel* pDepthMap = depth.GetDepthMap();
	XnUInt32 nEndIndex = XN_VGA_X_RES * XN_VGA_Y_RES;
	for(XnUInt32 i=0;i<nEndIndex;i++){
		int a=i%XN_VGA_X_RES;
		int b=i/XN_VGA_X_RES;
		if((pDepthMap[i]<(z+40)) && (pDepthMap[i]>(z-40))){
			size++;
		}
	}
	return size;
}

int check_swipe_horiz()
{
```

```
	int count=0;
	for(int i=0;i<FRAME_COUNT-1;i++){
		if(abs(hand_arr_x_1[i]-hand_arr_x_1[i+1])>move_thresh&&abs(hand_arr_y_1[i]-
hand_arr_y_1[i+1])<move_thresh&&abs(hand_arr_Z_1[i]-hand_arr_Z_1[i+1])<move_thresh){
			if(hand_arr_x_1[i]-hand_arr_x_1[i+1]>move_thresh){
				count++;
			}
			else{
				count--;
			}
		}
	}
	if(count>FRAME_COUNT-3)
		return -1;
	if(count<-(FRAME_COUNT-3))
		return 1;
	return 0;
}

int check_swipe_vert()
{
	int count=0;
	for(int i=0;i<FRAME_COUNT-1;i++){
		if(abs(hand_arr_y_1[i]-hand_arr_y_1[i+1])>move_thresh&&abs(hand_arr_x_1[i]-
hand_arr_x_1[i+1])<move_thresh&&abs(hand_arr_Z_1[i]-hand_arr_Z_1[i+1])<move_thresh){
			if(hand_arr_y_1[i]-hand_arr_y_1[i+1]>move_thresh){
				count++;
			}
			else{
				count--;
			}
		}
	}
	if(count>FRAME_COUNT-3)
		return -1;
	if(count<-(FRAME_COUNT-3))
		return 1;
	return 0;
}

int check_rotate_z()
{
	int count=0;
	for(int i=0;i<FRAME_COUNT-1;i++)
	{
		if(abs(hand_arr_x_1[i]-hand_arr_x_1[i+1])<steady_thresh && abs(hand_arr_y_1[i]-
hand_arr_y_1[i+1])<steady_thresh && abs(hand_arr_Z_2[i]-hand_arr_Z_2[i+1])>move_thresh)
		{
			if(hand_arr_Z_2[i]-hand_arr_Z_2[i+1]>0)
				count++;
			else
				count--;
		}
	}

	if(count>(FRAME_COUNT-3))
		return 1;
	else if(count<-(FRAME_COUNT-3))
```

```
			return -1;
		else
			return 0;
}

int check_rotate_x()
{
		int count=0;
		for(int i=0;i<FRAME_COUNT-1;i++)
		{
			if(abs(hand_arr_x_1[i]-hand_arr_x_1[i+1])<steady_thresh && abs(hand_arr_y_1[i]-hand_arr_y_1[i+1])<steady_thresh && abs(hand_arr_y_2[i]-hand_arr_y_2[i+1])>move_thresh)
			{
				if(hand_arr_y_2[i]-hand_arr_y_2[i+1]>0)
					count++;
				else
					count--;
			}
		}
		if(count>FRAME_COUNT-3)
			return 1;
		else if(count<-(FRAME_COUNT-3))
			return -1;
		else
			return 0;
}

int check_zoom()
{
		int count=0,prev_diff=0,diff=0;
		prev_diff=hand_arr_x_1[0]-hand_arr_x_2[0];
		for(int i=1;i<FRAME_COUNT;i++)
		{
			diff=hand_arr_x_1[i]-hand_arr_x_2[i];
			if((diff-prev_diff)>zoom_thresh && abs(hand_arr_y_1[i-1]-hand_arr_y_1[i])<steady_thresh && abs(hand_arr_y_2[i-1]-hand_arr_y_2[i])<steady_thresh)
				count++;
			else if((diff-prev_diff)<-(zoom_thresh) && abs(hand_arr_y_1[i-1]-hand_arr_y_1[i])<steady_thresh && abs(hand_arr_y_2[i-1]-hand_arr_y_2[i])<steady_thresh)
				count--;
			prev_diff=diff;
		}
		if(count>=(FRAME_COUNT-3))
			return 1;
		else if(count<=-(FRAME_COUNT-3))
			return -1;
		else
			return 0;
}

void pushOrbit()
{
		INPUT* buffer= new INPUT[5];
		buffer->type=INPUT_MOUSE;
		buffer->mi.dx=ROTATEX*55;
		buffer->mi.dy=CURSORY*85;
		buffer->mi.mouseData=0;
		buffer->mi.dwFlags=(MOUSEEVENTF_ABSOLUTE|MOUSEEVENTF_MOVE);
```

```
        buffer->mi.time=0;
        buffer->mi.dwExtraInfo = 0;
        (buffer+1)->type=INPUT_MOUSE;
        (buffer+1)->mi.dx=0;
        (buffer+1)->mi.dy=0;
        (buffer+1)->mi.mouseData=0;
        (buffer+1)->mi.dwFlags=(MOUSEEVENTF_LEFTDOWN);
        (buffer+1)->mi.time=0;
        (buffer+1)->mi.dwExtraInfo = 0;
        (buffer+2)->type = INPUT_MOUSE;
        (buffer+2)->mi.dx = 0;
        (buffer+2)->mi.dy = 0;
        (buffer+2)->mi.mouseData = 0;
        (buffer+2)->mi.dwFlags = MOUSEEVENTF_LEFTUP;
        (buffer+2)->mi.time = 0;
        (buffer+2)->mi.dwExtraInfo = 0;
        (buffer+3)->type=INPUT_MOUSE;
        (buffer+3)->mi.dx=(hand_arr_x_1[FRAME_COUNT-1]-xl)*width*55/(xr-xl);
        (buffer+3)->mi.dy=(height*85)-((hand_arr_y_1[FRAME_COUNT-1]-yr)*height*85/(yl-yr));
        (buffer+3)->mi.mouseData=0;
        (buffer+3)->mi.dwFlags=(MOUSEEVENTF_ABSOLUTE|MOUSEEVENTF_MOVE);
        (buffer+3)->mi.time=0;
        (buffer+3)->mi.dwExtraInfo = 0;
        (buffer+4)->type=INPUT_MOUSE;
        (buffer+4)->mi.dx=0;
        (buffer+4)->mi.dy=0;
        (buffer+4)->mi.mouseData=0;
        (buffer+4)->mi.dwFlags=(MOUSEEVENTF_LEFTDOWN);
        (buffer+4)->mi.time=0;
        (buffer+4)->mi.dwExtraInfo = 0;
        SendInput(5,buffer,sizeof(INPUT));
}

void pushMove()
{
        INPUT* buffer= new INPUT[4];
        buffer->type=INPUT_MOUSE;
        buffer->mi.dx=MOVEX*55;
        buffer->mi.dy=CURSORY*85;
        buffer->mi.mouseData=0;
        buffer->mi.dwFlags=(MOUSEEVENTF_ABSOLUTE|MOUSEEVENTF_MOVE);
        buffer->mi.time=0;
        buffer->mi.dwExtraInfo = 0;
        (buffer+1)->type=INPUT_MOUSE;
        (buffer+1)->mi.dx=0;
        (buffer+1)->mi.dy=0;
        (buffer+1)->mi.mouseData=0;
        (buffer+1)->mi.dwFlags=(MOUSEEVENTF_LEFTDOWN);
        (buffer+1)->mi.time=0;
        (buffer+1)->mi.dwExtraInfo = 0;
        (buffer+2)->type = INPUT_MOUSE;
        (buffer+2)->mi.dx = 0;
        (buffer+2)->mi.dy = 0;
        (buffer+2)->mi.mouseData = 0;
        (buffer+2)->mi.dwFlags = MOUSEEVENTF_LEFTUP;
        (buffer+2)->mi.time = 0;
        (buffer+2)->mi.dwExtraInfo = 0;
        (buffer+3)->type=INPUT_MOUSE;
```

```
		(buffer+3)->mi.dx=(hand_arr_x_1[FRAME_COUNT-1]-xl)*width*55/(xr-xl);
		(buffer+3)->mi.dy=(height*85)-((hand_arr_y_1[FRAME_COUNT-1]-yr)*height*85/(yl-yr));
		(buffer+3)->mi.mouseData=0;
		(buffer+3)->mi.dwFlags=(MOUSEEVENTF_ABSOLUTE|MOUSEEVENTF_MOVE);
		(buffer+3)->mi.time=0;
		(buffer+3)->mi.dwExtraInfo = 0;
		SendInput(4,buffer,sizeof(INPUT));
}

void pushPush()
{
		INPUT* buffer= new INPUT[4];
		buffer->type=INPUT_MOUSE;
		buffer->mi.dx=PUSHX*55;
		buffer->mi.dy=CURSORY*85;
		buffer->mi.mouseData=0;
		buffer->mi.dwFlags=(MOUSEEVENTF_ABSOLUTE|MOUSEEVENTF_MOVE);
		buffer->mi.time=0;
		buffer->mi.dwExtraInfo = 0;
		(buffer+1)->type=INPUT_MOUSE;
		(buffer+1)->mi.dx=0;
		(buffer+1)->mi.dy=0;
		(buffer+1)->mi.mouseData=0;
		(buffer+1)->mi.dwFlags=(MOUSEEVENTF_LEFTDOWN);
		(buffer+1)->mi.time=0;
		(buffer+1)->mi.dwExtraInfo = 0;
		(buffer+2)->type = INPUT_MOUSE;
		(buffer+2)->mi.dx = 0;
		(buffer+2)->mi.dy = 0;
		(buffer+2)->mi.mouseData = 0;
		(buffer+2)->mi.dwFlags = MOUSEEVENTF_LEFTUP;
		(buffer+2)->mi.time = 0;
		(buffer+2)->mi.dwExtraInfo = 0;
		(buffer+3)->type=INPUT_MOUSE;
		(buffer+3)->mi.dx=(hand_arr_x_1[FRAME_COUNT-1]-xl)*width*55/(xr-xl);
		(buffer+3)->mi.dy=(height*85)-((hand_arr_y_1[FRAME_COUNT-1]-yr)*height*85/(yl-yr));
		(buffer+3)->mi.mouseData=0;
		(buffer+3)->mi.dwFlags=(MOUSEEVENTF_ABSOLUTE|MOUSEEVENTF_MOVE);
		(buffer+3)->mi.time=0;
		(buffer+3)->mi.dwExtraInfo = 0;
		SendInput(4,buffer,sizeof(INPUT));
}

void pushCursor()
{
		INPUT* buffer= new INPUT[4];
		buffer->type=INPUT_MOUSE;
		buffer->mi.dx=CURSORX*55;
		buffer->mi.dy=CURSORY*85;
		buffer->mi.mouseData=0;
		buffer->mi.dwFlags=(MOUSEEVENTF_ABSOLUTE|MOUSEEVENTF_MOVE);
		buffer->mi.time=0;
		buffer->mi.dwExtraInfo = 0;
		(buffer+1)->type=INPUT_MOUSE;
		(buffer+1)->mi.dx=0;
		(buffer+1)->mi.dy=0;
		(buffer+1)->mi.mouseData=0;
		(buffer+1)->mi.dwFlags=(MOUSEEVENTF_LEFTDOWN);
```

```
		(buffer+1)->mi.time=0;
		(buffer+1)->mi.dwExtraInfo = 0;
		(buffer+2)->type = INPUT_MOUSE;
		(buffer+2)->mi.dx = 0;
		(buffer+2)->mi.dy = 0;
		(buffer+2)->mi.mouseData = 0;
		(buffer+2)->mi.dwFlags = MOUSEEVENTF_LEFTUP;
		(buffer+2)->mi.time = 0;
		(buffer+2)->mi.dwExtraInfo = 0;
		(buffer+3)->type=INPUT_MOUSE;
		(buffer+3)->mi.dx=600*55;
		(buffer+3)->mi.dy=400*85;
		(buffer+3)->mi.mouseData=0;
		(buffer+3)->mi.dwFlags=(MOUSEEVENTF_ABSOLUTE|MOUSEEVENTF_MOVE);
		(buffer+3)->mi.time=0;
		(buffer+3)->mi.dwExtraInfo = 0;
		SendInput(4,buffer,sizeof(INPUT));
}

void pushPlane(){
		if(currGest==ROTZCLOCK||currGest==ROTXCLOCK||currGest==ZOOMIN)
			return;
		INPUT* buffer= new INPUT[3];
		buffer->type=INPUT_MOUSE;
		buffer->mi.dx=PLANEX*55;
		buffer->mi.dy=CURSORY*85;
		buffer->mi.mouseData=0;
		buffer->mi.dwFlags=(MOUSEEVENTF_ABSOLUTE|MOUSEEVENTF_MOVE);
		buffer->mi.time=0;
		buffer->mi.dwExtraInfo = 0;
		(buffer+1)->type=INPUT_MOUSE;
		(buffer+1)->mi.dx=0;
		(buffer+1)->mi.dy=0;
		(buffer+1)->mi.mouseData=0;
		(buffer+1)->mi.dwFlags=(MOUSEEVENTF_LEFTDOWN);
		(buffer+1)->mi.time=0;
		(buffer+1)->mi.dwExtraInfo = 0;
		(buffer+2)->type = INPUT_MOUSE;
		(buffer+2)->mi.dx = 0;
		(buffer+2)->mi.dy = 0;
		(buffer+2)->mi.mouseData = 0;
		(buffer+2)->mi.dwFlags = MOUSEEVENTF_LEFTUP;
		(buffer+2)->mi.time = 0;
		(buffer+2)->mi.dwExtraInfo = 0;
		SendInput(3,buffer,sizeof(INPUT));
		delete buffer;
		prevTime=time(NULL);
		setBack();
		g_detected=true;
		planeRouter.SetActive(&g_swipeDetectorRight);
}

void pushLine(){
		if(currGest==ROTZCLOCK||currGest==ROTXCLOCK||currGest==ZOOMIN)
			return;
		if(prevMove==DOWN){
		if(cursor){
			cursor=false;
```

```
		cout<<endl<<"Going to line mode.";
		INPUT* buffer= new INPUT[3];
	buffer->type=INPUT_MOUSE;
	buffer->mi.dx=LINEX*55;
	buffer->mi.dy=CURSORY*85;
	buffer->mi.mouseData=0;
	buffer->mi.dwFlags=(MOUSEEVENTF_ABSOLUTE|MOUSEEVENTF_MOVE);
	buffer->mi.time=0;
	buffer->mi.dwExtraInfo = 0;
	(buffer+1)->type=INPUT_MOUSE;
	(buffer+1)->mi.dx=0;
	(buffer+1)->mi.dy=0;
	(buffer+1)->mi.mouseData=0;
	(buffer+1)->mi.dwFlags=(MOUSEEVENTF_LEFTDOWN);
	(buffer+1)->mi.time=0;
	(buffer+1)->mi.dwExtraInfo = 0;
	(buffer+2)->type = INPUT_MOUSE;
	(buffer+2)->mi.dx = 0;
	(buffer+2)->mi.dy = 0;
	(buffer+2)->mi.mouseData = 0;
	(buffer+2)->mi.dwFlags = MOUSEEVENTF_LEFTUP;
	(buffer+2)->mi.time = 0;
	(buffer+2)->mi.dwExtraInfo = 0;
	SendInput(3,buffer,sizeof(INPUT));
	}
	else{
		cursor=true;
		cout<<endl<<"Going to cursor mode.";
		INPUT* buffer= new INPUT[3];
	buffer->type=INPUT_MOUSE;
	buffer->mi.dx=CURSORX*55;
	buffer->mi.dy=CURSORY*85;
	buffer->mi.mouseData=0;
	buffer->mi.dwFlags=(MOUSEEVENTF_ABSOLUTE|MOUSEEVENTF_MOVE);
	buffer->mi.time=0;
	buffer->mi.dwExtraInfo = 0;
	(buffer+1)->type=INPUT_MOUSE;
	(buffer+1)->mi.dx=0;
	(buffer+1)->mi.dy=0;
	(buffer+1)->mi.mouseData=0;
	(buffer+1)->mi.dwFlags=(MOUSEEVENTF_LEFTDOWN);
	(buffer+1)->mi.time=0;
	(buffer+1)->mi.dwExtraInfo = 0;
	(buffer+2)->type = INPUT_MOUSE;
	(buffer+2)->mi.dx = 0;
	(buffer+2)->mi.dy = 0;
	(buffer+2)->mi.mouseData = 0;
	(buffer+2)->mi.dwFlags = MOUSEEVENTF_LEFTUP;
	(buffer+2)->mi.time = 0;
	(buffer+2)->mi.dwExtraInfo = 0;
	SendInput(3,buffer,sizeof(INPUT));
	}
	g_detected=true;
	setBack();
	lineRouter.SetActive(&g_swipeDetectorUpline);
	}
}
```

```
float getSmoothedX1(float x){
	float sum=0;
	for(int i=1;i<TRAIL_COUNT;i++)
		sum+=hand_arr_x_1[i];
	sum+=x;
	return (sum/TRAIL_COUNT);
}

float getSmoothedY1(float y){
	float sum=0;
	for(int i=1;i<TRAIL_COUNT;i++)
		sum+=hand_arr_y_1[i];
	sum+=y;
	return (sum/TRAIL_COUNT);
}

float getSmoothedX2(float x){
	float sum=0;
	for(int i=1;i<TRAIL_COUNT;i++)
		sum+=hand_arr_x_2[i];
	sum+=x;
	return (sum/TRAIL_COUNT);
}

float getSmoothedY2(float y){
	float sum=0;
	for(int i=1;i<TRAIL_COUNT;i++)
		sum+=hand_arr_y_2[i];
	sum+=y;
	return (sum/TRAIL_COUNT);
}

void XN_CALLBACK_TYPE SessionStart(const XnPoint3D& pFocus, void* UserCxt)
{
	cout<<endl<<"Session started"<<endl;
	g_flowRouter.SetActive(&g_broadcaster);
	cout<<endl<<"Do top left"<<endl;
}

void XN_CALLBACK_TYPE SessionEnd(void* UserCxt)
{
	cout<<endl<<"Session ended"<<endl;
}

void XN_CALLBACK_TYPE Push_Pushed(XnFloat fVelocity, XnFloat fAngle, void* cxt)
{
	cout<<endl<<"Push detected"<<endl;
	if(Calib)
	{
		if(xl==0)
		{
			xl=handposx;
			yl=handposy;
			cout<<endl<<"Do bottom right"<<endl;
		}
		else{
			xr=handposx;
			yr=handposy;
```

```
				Calib=false;
				cout<<endl<<"The points are: "<<xl<<" "<<yl<<" "<<xr<<" "<<yr<<endl;
			}
		}
		else
		{
			INPUT* buffer= new INPUT[2];
			buffer->type=INPUT_MOUSE;
			buffer->mi.dx=0;
			buffer->mi.dy=0;
			buffer->mi.mouseData=0;
			buffer->mi.dwFlags=(MOUSEEVENTF_LEFTDOWN);
			buffer->mi.time=0;
			buffer->mi.dwExtraInfo = 0;
			(buffer+1)->type = INPUT_MOUSE;
			(buffer+1)->mi.dx = 0;
			(buffer+1)->mi.dy = 0;
			(buffer+1)->mi.mouseData = 0;
			(buffer+1)->mi.dwFlags = MOUSEEVENTF_LEFTUP;
			(buffer+1)->mi.time = 0;
			(buffer+1)->mi.dwExtraInfo = 0;
			SendInput(2,buffer,sizeof(INPUT));
		}
}

void XN_CALLBACK_TYPE OnPointCreate(const XnVHandPointContext* cxt,void* h)
{
		if(id==0)
			id=cxt->nID;
		else
			id1=cxt->nID;
		numpoints++;
}

void XN_CALLBACK_TYPE OnPointUpdate(const XnVHandPointContext* cxt,void* h)
{
		bool flag=false;
		int ch=0;
		int SIZETHRESH=FTOCRATIO*(FULLATNEAR+(cxt->ptPosition.Z-NEAR)*(FULLATFAR-FULLATNEAR)/(FAR-NEAR));
		currTime=time(NULL);
		if(difftime(currTime,prevTime)>2)
			setBack();
		int size=gethandsize(cxt->ptPosition.Z,cxt->ptPosition.X,cxt->ptPosition.Y);
		ch=0;
		if(size>SIZETHRESH && grabbed){
			INPUT* buffer= new INPUT[1];
			buffer->type=INPUT_MOUSE;
			buffer->mi.dx=(cxt->ptPosition.X-xl)*width*55/(xr-xl);
			buffer->mi.dy=(height*85)-((cxt->ptPosition.Y-yr)*height*85/(yl-yr));
			buffer->mi.mouseData=0;
			buffer->mi.dwFlags=(MOUSEEVENTF_LEFTUP);
			buffer->mi.time=0;
			buffer->mi.dwExtraInfo = 0;
			SendInput(1,buffer,sizeof(INPUT));
			grabbed = false;
		}
		else if(size<SIZETHRESH && !grabbed){
			INPUT* buffer= new INPUT[1];
```

```
            buffer->type=INPUT_MOUSE;
            buffer->mi.dx=(cxt->ptPosition.X-xl)*width*55/(xr-xl);
            buffer->mi.dy=(height*85)-((cxt->ptPosition.Y-yr)*height*85/(yl-yr));
            buffer->mi.mouseData=0;
            buffer->mi.dwFlags=(MOUSEEVENTF_LEFTDOWN);
            buffer->mi.time=0;
            buffer->mi.dwExtraInfo = 0;
            SendInput(1,buffer,sizeof(INPUT));
            grabbed = true;
}
if(!Calib){
if(cxt->nID!=id){
            flag=true;
handposx2=getSmoothedX2(cxt->ptPosition.X);
handposx=handposx2;
handposy2=getSmoothedY2(cxt->ptPosition.Y);
handposy=handposy2;
for(int i=0;i<TRAIL_COUNT-1;i++){
                        hand_arr_x_2[i]=hand_arr_x_2[i+1];
                        hand_arr_y_2[i]=hand_arr_y_2[i+1];
                        hand_arr_Z_2[i]=hand_arr_Z_2[i+1];
            }
            hand_arr_x_2[TRAIL_COUNT-1]=handposx2;
            hand_arr_y_2[TRAIL_COUNT-1]=handposy2;
            hand_arr_Z_2[TRAIL_COUNT-1]=cxt->ptPosition.Z;
}
else{
            flag=true;
handposx1=getSmoothedX1(cxt->ptPosition.X);
handposx=handposx1;
handposy1=getSmoothedY1(cxt->ptPosition.Y);
handposy=handposy1;
for(int i=0;i<TRAIL_COUNT-1;i++){
                        hand_arr_x_1[i]=hand_arr_x_1[i+1];
                        hand_arr_y_1[i]=hand_arr_y_1[i+1];
                        hand_arr_Z_1[i]=hand_arr_Z_1[i+1];
            }
            hand_arr_x_1[TRAIL_COUNT-1]=handposx1;
            hand_arr_y_1[TRAIL_COUNT-1]=handposy1;
            hand_arr_Z_1[TRAIL_COUNT-1]=cxt->ptPosition.Z;
}
}
if(!flag){
handposx=cxt->ptPosition.X;
handposy=cxt->ptPosition.Y;
}
if(numpoints==1)
            ch=check_swipe_horiz();
if(ch==1){
            if(prevMove==NONE){
                        prevTime=time(NULL);
                        prevMove=RIGHT;
            }
}
if(ch==-1){
            if(prevMove==DOWN){
                        prevMove=LEFT;
                        prevTime=time(NULL);
```

```
		}
	}
	ch=0;
	if(numpoints==1)
		ch=check_swipe_vert();
	if(ch==1){
		if(prevMove==DOWN){
			prevTime=0;
			cout<<endl<<"Going to line mode.";
			pushLine();
			prevMove=NONE;
		}
		if(prevMove==LEFT){
			prevTime=0;
			cout<<endl<<"Going to plane mode.";
			pushPlane();
			prevMove=NONE;
		}
	}
	if(ch==-1){
		if(prevMove==RIGHT){
			prevTime=time(NULL);
			prevMove=DOWN;
		}
	}
	ch=0;
	if(numpoints==2)
		ch=check_rotate_z();
	if(ch==1){
		lastTime=time(NULL);
		cout<<endl<<currGest;
		if(currGest!=ROTZCLOCK)
		{
			currGest=ROTZCLOCK;
			pushOrbit();
		}
	}
	if(ch==-1){
		lastTime=time(NULL);
		if(currGest==ROTZCLOCK){
		currGest=STOP;
		INPUT* buffer= new INPUT[1];
	buffer->type=INPUT_MOUSE;
	buffer->mi.dx=0;
	buffer->mi.dy=0;
	buffer->mi.mouseData=0;
	buffer->mi.dwFlags=(MOUSEEVENTF_LEFTUP);
	buffer->mi.time=0;
	buffer->mi.dwExtraInfo = 0;
	SendInput(1,buffer,sizeof(INPUT));
		pushCursor();
		}
	}
	ch=0;
	if(numpoints==2)
		ch=check_rotate_x();
	if(ch==1)
	{
```

```
		lastTime=time(NULL);
		if(currGest!=ROTXCLOCK)
		{
			currGest=ROTXCLOCK;
			pushMove();
		}
	}
	if(ch==-1){
		lastTime=time(NULL);
		if(currGest==ROTXCLOCK){
			currGest=STOP;
		INPUT* buffer= new INPUT[1];
	buffer->type=INPUT_MOUSE;
	buffer->mi.dx=0;
	buffer->mi.dy=0;
	buffer->mi.mouseData=0;
	buffer->mi.dwFlags=(MOUSEEVENTF_LEFTUP);
	buffer->mi.time=0;
	buffer->mi.dwExtraInfo = 0;
	SendInput(1,buffer,sizeof(INPUT));
			pushCursor();
		}
	}
	ch=0;
	if(numpoints==2)
		ch=check_zoom();
	if(ch==1)
	{
		lastTime=time(NULL);
		if(currGest!=ZOOMOUT)
		{
			currGest=ZOOMOUT;
			pushPush();
		}
	}
	if(ch==-1){
		lastTime=time(NULL);
		if(currGest==ZOOMOUT){
			currGest=STOP;
		INPUT* buffer= new INPUT[1];
	buffer->type=INPUT_MOUSE;
	buffer->mi.dx=0;
	buffer->mi.dy=0;
	buffer->mi.mouseData=0;
	buffer->mi.dwFlags=(MOUSEEVENTF_LEFTUP);
	buffer->mi.time=0;
	buffer->mi.dwExtraInfo = 0;
	SendInput(1,buffer,sizeof(INPUT));
			pushCursor();
		}
	}

	if(cxt->nID==id&&!Calib){
	INPUT* buffer= new INPUT[1];
	buffer->type=INPUT_MOUSE;
	buffer->mi.dx=(cxt->ptPosition.X-xl)*width*55/(xr-xl);
	buffer->mi.dy=(height*85)-((cxt->ptPosition.Y-yr)*height*85/(yl-yr));
	buffer->mi.mouseData=0;
```

```
		buffer->mi.dwFlags=(MOUSEEVENTF_ABSOLUTE|MOUSEEVENTF_MOVE);
		buffer->mi.time=0;
		buffer->mi.dwExtraInfo = 0;
		SendInput(1,buffer,sizeof(INPUT));
		}
}

void XN_CALLBACK_TYPE OnPointDestroy(XnUInt32 nID,void* h){
		setBack();
		if(nID==id)
		{
			if(numpoints==2)
			{
				if(currGest!=STOP)
				{
					currGest=STOP;
					INPUT* buffer= new INPUT[3];
					buffer->type=INPUT_MOUSE;
					buffer->mi.dx=0;
					buffer->mi.dy=0;
					buffer->mi.mouseData=0;
					buffer->mi.dwFlags=(MOUSEEVENTF_LEFTUP);
					buffer->mi.time=0;
					buffer->mi.dwExtraInfo = 0;
					(buffer+1)->type=INPUT_MOUSE;
					(buffer+1)->mi.dx=CURSORX*55;
					(buffer+1)->mi.dy=CURSORY*85;
					(buffer+1)->mi.mouseData=0;
					(buffer+1)->mi.dwFlags=(MOUSEEVENTF_ABSOLUTE|MOUSEEVENTF_MOVE);
					(buffer+1)->mi.time=0;
					(buffer+1)->mi.dwExtraInfo = 0;
					(buffer+2)->type = INPUT_MOUSE;
					(buffer+2)->mi.dx = 0;
					(buffer+2)->mi.dy = 0;
					(buffer+2)->mi.mouseData = 0;
					(buffer+2)->mi.dwFlags = MOUSEEVENTF_LEFTDOWN;
					(buffer+2)->mi.time = 0;
					(buffer+2)->mi.dwExtraInfo = 0;
					(buffer+3)->type=INPUT_MOUSE;
					(buffer+3)->mi.dx=0;
					(buffer+3)->mi.dy=0;
					(buffer+3)->mi.mouseData=0;
					(buffer+3)->mi.dwFlags=(MOUSEEVENTF_LEFTUP);
					(buffer+3)->mi.time=0;
					(buffer+3)->mi.dwExtraInfo = 0;
					SendInput(4,buffer,sizeof(INPUT));
					delete buffer;
				}
				id=id1;
				for(int i=0;i<TRAIL_COUNT;i++)
				{
					hand_arr_x_1[i]=hand_arr_x_2[i];
					hand_arr_x_2[i]=0;
					hand_arr_y_1[i]=hand_arr_y_2[i];
					hand_arr_y_2[i]=0;
					hand_arr_Z_1[i]=hand_arr_Z_2[i];
					hand_arr_Z_2[i]=0;
				}
```

```
			}
			else
			{
				id=0;
				id1=0;
				for(int i=0;i<TRAIL_COUNT;i++)
				{
					hand_arr_x_1[i]=0;
					hand_arr_x_2[i]=0;
					hand_arr_y_1[i]=0;
					hand_arr_y_2[i]=0;
					hand_arr_Z_1[i]=0;
					hand_arr_Z_2[i]=0;
				}
			}
		}
		else
		{
			id1=0;
			for(int i=0;i<TRAIL_COUNT;i++)
			{
				hand_arr_x_2[i]=0;
				hand_arr_y_2[i]=0;
				hand_arr_Z_2[i]=0;
			}
		}
		numpoints--;
}

void XN_CALLBACK_TYPE CircleDetect(XnFloat fTimes, XnBool bConfident, const XnVCircle *pCircle, void *pUserCxt)
{
		if(currGest==ROTZCLOCK||currGest==ROTXCLOCK||currGest==ZOOMIN)
			return;
		INPUT* buffer= new INPUT[3];
		buffer->type=INPUT_MOUSE;
		buffer->mi.dx=CIRCLEX*55;
		buffer->mi.dy=CURSORY*85;
		buffer->mi.mouseData=0;
		buffer->mi.dwFlags=(MOUSEEVENTF_ABSOLUTE|MOUSEEVENTF_MOVE);
		buffer->mi.time=0;
		buffer->mi.dwExtraInfo = 0;
		(buffer+1)->type=INPUT_MOUSE;
		(buffer+1)->mi.dx=0;
		(buffer+1)->mi.dy=0;
		(buffer+1)->mi.mouseData=0;
		(buffer+1)->mi.dwFlags=(MOUSEEVENTF_LEFTDOWN);
		(buffer+1)->mi.time=0;
		(buffer+1)->mi.dwExtraInfo = 0;
		(buffer+2)->type = INPUT_MOUSE;
		(buffer+2)->mi.dx = 0;
		(buffer+2)->mi.dy = 0;
		(buffer+2)->mi.mouseData = 0;
		(buffer+2)->mi.dwFlags = MOUSEEVENTF_LEFTUP;
		(buffer+2)->mi.time = 0;
		(buffer+2)->mi.dwExtraInfo = 0;
		SendInput(3,buffer,sizeof(INPUT));
}
```

```
void XN_CALLBACK_TYPE waveDetect(void* UserCxt)
{
	if(currGest==ROTZCLOCK||currGest==ROTXCLOCK||currGest==ZOOMIN)
		return;
	if(!cursor)
	{
		pushCursor();
		cursor=true;
		return;
	}
	cursor=false;
	INPUT* buffer= new INPUT[3];
	buffer->type=INPUT_MOUSE;
	buffer->mi.dx=ERASEX*55;
	buffer->mi.dy=CURSORY*85;
	buffer->mi.mouseData=0;
	buffer->mi.dwFlags=(MOUSEEVENTF_ABSOLUTE|MOUSEEVENTF_MOVE);
	buffer->mi.time=0;
	buffer->mi.dwExtraInfo = 0;
	(buffer+1)->type=INPUT_MOUSE;
	(buffer+1)->mi.dx=0;
	(buffer+1)->mi.dy=0;
	(buffer+1)->mi.mouseData=0;
	(buffer+1)->mi.dwFlags=(MOUSEEVENTF_LEFTDOWN);
	(buffer+1)->mi.time=0;
	(buffer+1)->mi.dwExtraInfo = 0;
	(buffer+2)->type = INPUT_MOUSE;
	(buffer+2)->mi.dx = 0;
	(buffer+2)->mi.dy = 0;
	(buffer+2)->mi.mouseData = 0;
	(buffer+2)->mi.dwFlags = MOUSEEVENTF_LEFTUP;
	(buffer+2)->mi.time = 0;
	(buffer+2)->mi.dwExtraInfo = 0;
	SendInput(3,buffer,sizeof(INPUT));
}

int main()
{
	prevTime=0;
	XnStatus nRetVal=XN_STATUS_OK;
	xn::Context context;
	XnStatus rc;
	rc = context.InitFromXmlFile("C:\\Program Files (x86)\\Prime Sense\\NITE\\Data\\Sample-Tracking.xml");
	XnVSessionManager sessionManager;
	rc = sessionManager.Initialize(&context, "Wave,Click", "RaiseHand");
	sessionManager.RegisterSession(NULL, &SessionStart, &SessionEnd);

	XnVPointControl point;
	XnVPointControl::PointCreateCB create=&OnPointCreate;
	XnVPointControl::PointUpdateCB update=&OnPointUpdate;
	XnVPointControl::PointDestroyCB destroy=&OnPointDestroy;

	nRetVal=hand.Create(context);
	nRetVal = depth.Create(context);
	XnMapOutputMode mapMode;
	mapMode.nXRes = XN_VGA_X_RES;
	mapMode.nYRes = XN_VGA_Y_RES;
	mapMode.nFPS = 30;
```

```
	nRetVal = depth.SetMapOutputMode(mapMode);
	XnUInt32 nMiddleIndex = XN_VGA_X_RES * XN_VGA_Y_RES/2 + XN_VGA_X_RES/2;
	nRetVal = context.StartGeneratingAll();

	g_circle.RegisterCircle(NULL,&CircleDetect);
	pushDetector.RegisterPush(NULL, &Push_Pushed);
	wavedetector.RegisterWave(NULL,&waveDetect);
	point.RegisterPointCreate(NULL,create);
	point.RegisterPointUpdate(NULL,update);
	point.RegisterPointDestroy(NULL,destroy);

	g_flowRouter.SetActive(NULL);
	sessionManager.AddListener(&g_flowRouter);

	g_broadcaster.AddListener(&point);
	g_broadcaster.AddListener(&pushDetector);
	g_broadcaster.AddListener(&g_circle);
	g_broadcaster.AddListener(&wavedetector);

	while(!kbhit())
	{
		nRetVal=context.WaitOneUpdateAll(depth);
		sessionManager.Update(&context);
	}
	return 0;
}
```