//'''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''

// Kinect to mouse/keyboard control

// Casimir Sowinski, 10/10/2015

//

// Using code from:

// - Joshua Blake

// \* URL: https://kinectpowerpoint.codeplex.com/

// \* Licence: http://kinectpowerpoint.codeplex.com/license

// - Saad Mousliki, hosted on CodeProject

// \* URL: http://www.codeproject.com/Tips/371718/The-Kinect-Mouse-Controller-Csharp

// \* Licence: http://www.codeproject.com/info/cpol10.aspx

//

// Comments key:

// '\*' Added reference or namespace or disambiguation resulting from new, added namespaces

// '#' Ideas for future expansion/features

//.....................................................................................................

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Data;

using System.Windows.Documents;

using System.Windows.Input;

using System.Windows.Media;

using System.Windows.Media.Imaging;

using System.Windows.Navigation;

using System.Windows.Shapes;

using Microsoft.Kinect;

using Microsoft.Speech.Recognition;

using System.Threading;

using System.IO;

using Microsoft.Speech.AudioFormat;

using System.Diagnostics;

using System.Windows.Threading;

using KinectMouseController;//\*

using System.Windows.Forms;//\*

using Coding4Fun.Kinect.Wpf;//\*

namespace KinectPowerPointControl

{

public partial class MainWindow : Window

{

// Init systems

KinectSensor sensor;

SpeechRecognitionEngine speechRecognizer;

//string filePath = @"C:\Users\Casimir\Google Drive\PSU\ECE 578 ROBOTICS 1\Kinect\button.wav";

System.Media.SoundPlayer player\_1 = new System.Media.SoundPlayer(@"C:\Users\Casimir\Google Drive\PSU\ECE 578 ROBOTICS 1\Kinect\button.wav");

DispatcherTimer readyTimer;

byte[] colorBytes;

Skeleton[] skeletons;

// Flags

bool isCirclesVisible = true;

bool isForwardGestureActive = false;

bool isMouseGestureActive = false;

bool isBackGestureActive = false;

bool isKeyTABActive = false;

bool isKeyENTERActive = false;

bool isClicked = false;

bool clickToggle = false;

bool enableMouse = false;

// Handles determination of if a mouseclick is being requested from the user

int reclickTimer = 0;

int reclickTimerInit = 15;

int clickDurTimer = 1;

int clickDurTimerInit = 1;

int headTrack = 1;

double setPoint = 0.5; // distance between r hand and head for leftclick

double clickComplete = 0; // range 0 - 100, tracks percentage of a in progress click

// Colors for ellipses over hands/head

SolidColorBrush activeBrush = new SolidColorBrush(Colors.Green);

SolidColorBrush inactiveBrush = new SolidColorBrush(Colors.Red);

SolidColorBrush newBrush = new SolidColorBrush(Colors.Azure);

SolidColorBrush enableBrush = new SolidColorBrush(Colors.Purple);

// For smooth mouse control [not implemented yet]

const int samples = 10;

public int[] cursorXFilter = new int[samples];

public int[] cursorYFilter = new int[samples];

public const float SkeletonMaxX = 0.6f;

public const float SkeletonMaxY = 0.4f;

public MainWindow() // Initialize main window

{

// Runtime initialization is handled when the window is opened. When the window

// is closed, the runtime MUST be unitialized.

InitializeComponent();

// Handle input from the keyboard

this.Loaded += new RoutedEventHandler(MainWindow\_Loaded);

// Handle the content obtained from the video camera, once received.

this.KeyDown += new System.Windows.Input.KeyEventHandler(MainWindow\_KeyDown);

}

void MainWindow\_Loaded(object sender, RoutedEventArgs e) // Initialize sensor and subsystems

{

// Construct KinectSensor object

sensor = KinectSensor.KinectSensors.FirstOrDefault();

// Close if no Kinect Sensor detected

if (sensor == null)

{

System.Windows.MessageBox.Show("This application requires a Kinect sensor.");

this.Close();

}

// Initialize KinectSensor with ColorStream, DepthStream, SkeletonStream,

// ColorFrameReady, SkeletonFrameReady, and angle

sensor.Start();

sensor.ColorStream.Enable(ColorImageFormat.RgbResolution640x480Fps30);

sensor.ColorFrameReady += new EventHandler<ColorImageFrameReadyEventArgs>(sensor\_ColorFrameReady);

sensor.DepthStream.Enable(DepthImageFormat.Resolution320x240Fps30);

sensor.SkeletonStream.Enable();

sensor.SkeletonFrameReady += new EventHandler<SkeletonFrameReadyEventArgs>(sensor\_SkeletonFrameReady);

sensor.ElevationAngle = -15;

// Initialize ExitEventHandler to shutdown propery

System.Windows.Application.Current.Exit += new ExitEventHandler(Current\_Exit); //\*added System.Windows.

// Initialize SpeechRecognition

InitializeSpeechRecognition();

}

void Current\_Exit(object sender, ExitEventArgs e) // Handle exit gracefully

{

// Shutdown SpeechRecognizer if started

if (speechRecognizer != null)

{

speechRecognizer.RecognizeAsyncCancel();

speechRecognizer.RecognizeAsyncStop();

}

// Shutdown KinectSensor if started

if (sensor != null)

{

sensor.AudioSource.Stop();

sensor.Stop();

sensor.Dispose();

sensor = null;

}

}

void MainWindow\_KeyDown(object sender, System.Windows.Input.KeyEventArgs e) // Handle keyboard input events

{

// Handle KeyEvents

if (e.Key == Key.C)

{

ToggleCircles();

}

else if (e.Key == Key.M)

{

// Toggle enableMouse

//#Find a way to make KeyEvents to work when the window isn't in focus/active

if (enableMouse)

{

enableMouse = false;

}

else

{

enableMouse = true;

}

}

}

void sensor\_ColorFrameReady(object sender, ColorImageFrameReadyEventArgs e)

{

using (var image = e.OpenColorImageFrame())

{

// Exit if ColorFrame not ready

if (image == null)

return;

// Construct new image if applicable

if (colorBytes == null || colorBytes.Length != image.PixelDataLength)

{

colorBytes = new byte[image.PixelDataLength];

}

image.CopyPixelDataTo(colorBytes);

//You could use PixelFormats.Bgr32 below to ignore the alpha,

//or if you need to set the alpha you would loop through the bytes

//as in this loop below

int length = colorBytes.Length;

for (int i = 0; i < length; i += 4)

{

colorBytes[i + 3] = 255;

}

BitmapSource source = BitmapSource.Create(image.Width,

image.Height,

96,

96,

PixelFormats.Bgra32,

null,

colorBytes,

image.Width \* image.BytesPerPixel);

videoImage.Source = source;

}

}

//VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV

void sensor\_SkeletonFrameReady(object sender, SkeletonFrameReadyEventArgs e)

{

using (var skeletonFrame = e.OpenSkeletonFrame())

{

// Exit if SkeletonFrame not ready

if (skeletonFrame == null)

return;

// Construct new Skeleton if applicable

if (skeletons == null || skeletons.Length != skeletonFrame.SkeletonArrayLength)

{

skeletons = new Skeleton[skeletonFrame.SkeletonArrayLength];

}

skeletonFrame.CopySkeletonDataTo(skeletons);

}

// Construct closestSkeleton

Skeleton closestSkeleton = skeletons.Where(s => s.TrackingState == SkeletonTrackingState.Tracked)

.OrderBy(s => s.Position.Z)

.FirstOrDefault();

// This used outside-in bias too

//Skeleton closestSkeleton = skeletons.Where(s => s.TrackingState == SkeletonTrackingState.Tracked)

// .OrderBy(s => s.Position.Z \* Math.Abs(s.Position.X))

// .FirstOrDefault();

// Exit if closestSkeleton not created

if (closestSkeleton == null)

return;

// Initialize limbs

var head = closestSkeleton.Joints[JointType.Head];

var rightHand = closestSkeleton.Joints[JointType.HandRight];

var leftHand = closestSkeleton.Joints[JointType.HandLeft];

//var leftThumb = closestSkeleton.Joints[JointType.];

// If we don't have a good read on the joints so we cannot process gestures

if (head.TrackingState == JointTrackingState.NotTracked ||

rightHand.TrackingState == JointTrackingState.NotTracked ||

leftHand.TrackingState == JointTrackingState.NotTracked)

{

return;

}

// Create 2d projection of right hand limb

Joint scaledRight = closestSkeleton.Joints[JointType.HandRight].ScaleTo(

(int)SystemInformation.PrimaryMonitorSize.Width,

(int)SystemInformation.PrimaryMonitorSize.Height,

SkeletonMaxX, SkeletonMaxY);

//\*\*\*\*\*\*\*\*\*\*\*\*{Add support for left handed}

// Assign variables to projected locations

cursorXFilter[0] = (int)scaledRight.Position.X;

cursorYFilter[0] = (int)scaledRight.Position.Y;

// Running average function

int cursorX = runningXAve();

int cursorY = runningYAve();

//Debug.WriteLine("cursorX: {0}, cursorY: {1}", cursorX, cursorY);

// Send mouse info if enabled

if (enableMouse)

{

KinectMouseController.KinectMouseMethods.SendMouseInput(

cursorX, cursorY, SystemInformation.PrimaryMonitorSize.Width,

SystemInformation.PrimaryMonitorSize.Height, isClicked);

}

// Handles timer to make you only click if its been some time after the last click, (30 fps)

if (reclickTimer != 0)

{

reclickTimer -= 1;

}

// Handles how long a click is engaged before unclicking (30 fps)

if (clickDurTimer != 0 && isClicked)

{

clickDurTimer -= 1;

}

else

{

clickDurTimer = clickDurTimerInit;

isClicked = false;

}

// Send counter values to screen

//Console.WriteLine("reClickTimer: {0}, clickDurTimer: {1}", reclickTimer, clickDurTimer);

//Debug.WriteLine("poopy poop, pooper");

// Debug.WriteLine("reClickTimer: {0}, clickDurTimer: {1}", reclickTimer, clickDurTimer);

// Update indicator ellipse positions

SetEllipsePosition(ellipseHead, head, false, 20);

SetEllipsePosition(ellipseLeftHand, leftHand, false, 20);

SetEllipsePosition(ellipseRightHand, rightHand, isClicked, clickComplete);

//SetEllipsePosition(ellipseEnable, rightHand, isForwardGestureActive);

// Update gesture recognition methods

ProcessForwardBackGesture(head, rightHand, leftHand);

ProcessMouseGesture(head, rightHand, leftHand);

}

private void SetEllipsePosition(Ellipse ellipse, Joint joint, bool isHighlighted, double size)

// This method is used to position the ellipses on the canvas according to correct movements

// of the tracked joints.

{

double sizeBounded = 0.5 \* size;

if (sizeBounded < 10)

{

sizeBounded = 10;

}

// Assign channel values between rgb(48,230,38) and alpha=100 to rgb(242,26,26) and alpha =255

// R G B A

//1st 48 230 38 100

//2nd 242 26 26 250

double aDummy = 100 + (150 / 100) \* size;

double rDummy = 48 + (194 / 100) \* size;

double gDummy = 230 - (204 / 100) \* size;

double bDummy = 38 - (12 / 100) \* size;

// Bound values between 0 and 255

aDummy = byteBound(aDummy);

rDummy = byteBound(rDummy);

gDummy = byteBound(gDummy);

bDummy = byteBound(bDummy);

// Cast to bytes

byte aColor = System.Convert.ToByte(aDummy); // Alpha chanel

byte rColor = System.Convert.ToByte(rDummy); // r channel

byte gColor = System.Convert.ToByte(gDummy); // g channel

byte bColor = System.Convert.ToByte(bDummy); // b channel

SolidColorBrush mySolidColorBrush = new SolidColorBrush();

mySolidColorBrush.Color = Color.FromArgb(aColor, rColor, gColor, bColor);

//Debug.WriteLine("rColor[0]: {0}, rColor[1]: {1}", rColor[0], rColor[1]);

//Debug.WriteLine("byteDummy: {0}", byteDummy);

//Debug.WriteLine("size: {0}", size);

//Debug.WriteLine("rColor: {0}", rColor);

if (isHighlighted)

{

ellipse.Width = sizeBounded;

ellipse.Height = sizeBounded;

ellipse.Fill = mySolidColorBrush;

}

else

{

ellipse.Width = sizeBounded;

ellipse.Height = sizeBounded;

ellipse.Fill = mySolidColorBrush;

}

// Construct CoordinateMapper object to get 2D projection data from 3D joint data

CoordinateMapper mapper = sensor.CoordinateMapper;

// Extract appropriate data from mapper object

var point = mapper.MapSkeletonPointToColorPoint(joint.Position, sensor.ColorStream.Format);

// Update indicator ellipse position

Canvas.SetLeft(ellipse, point.X - ellipse.ActualWidth / 2);

Canvas.SetTop(ellipse, point.Y - ellipse.ActualHeight / 2);

}

double byteBound(double valToBound)

{

if (valToBound > 255)

{

valToBound = 255;

}

else if (valToBound < 0)

{

valToBound = 0;

}

return valToBound;

}

int runningXAve()

{

// Init vars

int average = 0;

// Back propagate old values

for (int j = samples - 1; j > 0; j--)

{

cursorXFilter[j] = cursorXFilter[j - 1];

}

// Sum values

for (int j = 0; j < samples; j++)

{

average += cursorXFilter[j];

}

average = average / samples;

return average;

}

int runningYAve()

{

// Init vars

int average = 0;

// Back propagate old values

for (int j = samples - 1; j > 0; j--)

{

cursorYFilter[j] = cursorYFilter[j - 1];

}

// Sum values

for (int j = 0; j < samples; j++)

{

average += cursorYFilter[j];

}

average = average / samples;

return average;

}

private void ProcessMouseGesture(Joint head, Joint rightHand, Joint leftHand)

{

// Handle clickComplete that shows how close a click is to being completed

//clickComplete = 100 / setPoint \* (head.Position.Z - rightHand.Position.Z);

clickComplete = 100 / setPoint \* Math.Sqrt(Math.Pow(head.Position.X - rightHand.Position.X,2)

+ Math.Pow(head.Position.Y - rightHand.Position.Y, 2)

+ Math.Pow(head.Position.Z - rightHand.Position.Z, 2));

//Debug.WriteLine("clickComplete: {0}", clickComplete);

//Debug.WriteLine("headX: {0}, headY: {1},headZ{2}",head.Position.X,head.Position.Y,head.Position.Z);

if (clickComplete > 100)

{

clickComplete = 100;

// Play full click sound

if (enableMouse)

{

//player\_1.Stop();

player\_1.Play();

}

}

else if (clickComplete < 0)

{

clickComplete = 0;

}

//Debug.WriteLine("clickComplete: {0}", clickComplete);

// Processes mouse left-click state from left hand relative to head WRT Z

//if (norm2(head.Position, rightHand.Position) > 0.30)

if (Math.Abs(head.Position.Z - rightHand.Position.Z) > setPoint)

{

//if (!isMouseGestureActive)

//{

// if (!isClicked && reclickTimer == 0)

// {

// isMouseGestureActive = true;

// //System.Windows.Forms.SendKeys.SendWait("{TAB}");

// isClicked = true;

// reclickTimer = reclickTimerInit;

// //clickToggle = true;

// }

//}

isClicked = true;

}

else

{

isMouseGestureActive = false;

isClicked = false;

//clickToggle = false;

}

////if (Math.Abs(rightHand.Position.X - leftHand.Position.X) < 0.2)

//if (Math.Abs(rightHand.Position.Z - head.Position.Z) > 0.5)

//{

// if (!isMouseGestureActive)

// {

// if (!isKeyENTERActive)

// {

// //sensor.ElevationAngle = (int) head.Position.Y;

// //isMouseGestureActive = true;

// //System.Windows.Forms.SendKeys.SendWait("{ENTER}");

// //System.Windows.Forms.MouseEventArgs.X

// }

// }

//}

//else

//{

// isKeyENTERActive = false;

// isMouseGestureActive = false;

//}

}

private void ProcessForwardBackGesture(Joint head, Joint rightHand, Joint leftHand)

{

// Processes gesture of either hand being farther than a set amount from the head

if (rightHand.Position.X > head.Position.X + 0.55 && enableMouse)

{

if (!isForwardGestureActive)

{

isForwardGestureActive = true;

System.Windows.Forms.SendKeys.SendWait("{Right}");

}

}

else

{

isForwardGestureActive = false;

}

if (leftHand.Position.X < head.Position.X - 0.55)

{

if (!isBackGestureActive)

{

isBackGestureActive = true;

System.Windows.Forms.SendKeys.SendWait("{Left}");

}

}

else

{

isBackGestureActive = false;

}

}

private double norm2(Point headPos, Point rightHandPos)

{

double dX = Math.Pow(headPos.X - rightHandPos.X, 2);

double dY = Math.Pow(headPos.Y - rightHandPos.Y, 2);

//double dZ = Math.Pow(headPos.Z - rightHandPos.Z, 2);

//double d = Math.Sqrt(dX + dY + dZ);

double d = 0;

//Single d = Math.Pow(headPos.Position.X - rightHandPos.Position.X, 2)

// + Math.Pow(headPos.Position.Y - rightHandPos.Position.Y, 2)

// + Math.Pow(headPos.Position.Z - rightHandPos.Position.Z, 2)

//Single distance = 0; //Math.Sqrt(0);

return d;

}

void ToggleCircles()

{

if (isCirclesVisible)

HideCircles();

else

ShowCircles();

}

void HideCircles()

{

isCirclesVisible = false;

ellipseHead.Visibility = System.Windows.Visibility.Collapsed;

ellipseLeftHand.Visibility = System.Windows.Visibility.Collapsed;

ellipseRightHand.Visibility = System.Windows.Visibility.Collapsed;

}

void ShowCircles()

{

isCirclesVisible = true;

ellipseHead.Visibility = System.Windows.Visibility.Visible;

ellipseLeftHand.Visibility = System.Windows.Visibility.Visible;

ellipseRightHand.Visibility = System.Windows.Visibility.Visible;

}

private void ShowWindow()

{

this.Topmost = true;

this.WindowState = System.Windows.WindowState.Normal;

}

private void HideWindow()

{

this.Topmost = false;

this.WindowState = System.Windows.WindowState.Minimized;

}

#region Speech Recognition Methods

private static RecognizerInfo GetKinectRecognizer()

{

//\*\*\*\*\*\*\*\*\*\*\*\*{Add support for multiple languages}

Func<RecognizerInfo, bool> matchingFunc = r =>

{

string value;

r.AdditionalInfo.TryGetValue("Kinect", out value);

return "True".Equals(value, StringComparison.InvariantCultureIgnoreCase)

&& "en-US".Equals(r.Culture.Name, StringComparison.InvariantCultureIgnoreCase);

};

return SpeechRecognitionEngine.InstalledRecognizers().Where(matchingFunc).FirstOrDefault();

}

private void InitializeSpeechRecognition()

{

RecognizerInfo ri = GetKinectRecognizer();

if (ri == null)

{

// added System.Windows.

System.Windows.MessageBox.Show(

@"There was a problem initializing Speech Recognition.

Ensure you have the Microsoft Speech SDK installed.",

"Failed to load Speech SDK",

MessageBoxButton.OK,

MessageBoxImage.Error);

return;

}

try

{

speechRecognizer = new SpeechRecognitionEngine(ri.Id);

}

catch

{

// added System.Windows.

System.Windows.MessageBox.Show(

@"There was a problem initializing Speech Recognition.

Ensure you have the Microsoft Speech SDK installed and configured.",

"Failed to load Speech SDK",

MessageBoxButton.OK,

MessageBoxImage.Error);

}

var phrases = new Choices();

phrases.Add("computer show window");

phrases.Add("computer hide window");

phrases.Add("computer show circles");

phrases.Add("computer hide circles");

phrases.Add("computer close window");

phrases.Add("computer enable mouse");

phrases.Add("computer turn mouse on");

phrases.Add("computer turn on mouse");

phrases.Add("computer disable mouse");

phrases.Add("computer turn off mouse");

phrases.Add("computer turn mouse off");

phrases.Add("computer switch window");

phrases.Add("powerpoint full screen");

phrases.Add("powerpoint return");

phrases.Add("computer go back");

phrases.Add("computer undo");

phrases.Add("computer print screen");

phrases.Add("computer toggle lock");

phrases.Add("computer open paint");

phrases.Add("computer maximize window");

var gb = new GrammarBuilder();

//Specify the culture to match the recognizer in case we are running in a different culture.

gb.Culture = ri.Culture;

gb.Append(phrases);

// Create the actual Grammar instance, and then load it into the speech recognizer.

var g = new Grammar(gb);

speechRecognizer.LoadGrammar(g);

speechRecognizer.SpeechRecognized += SreSpeechRecognized;

speechRecognizer.SpeechHypothesized += SreSpeechHypothesized;

speechRecognizer.SpeechRecognitionRejected += SreSpeechRecognitionRejected;

this.readyTimer = new DispatcherTimer();

this.readyTimer.Tick += this.ReadyTimerTick;

this.readyTimer.Interval = new TimeSpan(0, 0, 4);

this.readyTimer.Start();

}

private void ReadyTimerTick(object sender, EventArgs e)

{

this.StartSpeechRecognition();

this.readyTimer.Stop();

this.readyTimer.Tick -= ReadyTimerTick;

this.readyTimer = null;

}

private void StartSpeechRecognition()

{

if (sensor == null || speechRecognizer == null)

return;

var audioSource = this.sensor.AudioSource;

audioSource.BeamAngleMode = BeamAngleMode.Adaptive;

var kinectStream = audioSource.Start();

speechRecognizer.SetInputToAudioStream(

kinectStream, new SpeechAudioFormatInfo(EncodingFormat.Pcm, 16000, 16, 1, 32000, 2, null));

speechRecognizer.RecognizeAsync(RecognizeMode.Multiple);

}

void SreSpeechRecognitionRejected(object sender, SpeechRecognitionRejectedEventArgs e)

{

Trace.WriteLine("\nSpeech Rejected, confidence: " + e.Result.Confidence);

}

void SreSpeechHypothesized(object sender, SpeechHypothesizedEventArgs e)

{

Trace.Write("\rSpeech Hypothesized: \t{0}", e.Result.Text);

}

void SreSpeechRecognized(object sender, SpeechRecognizedEventArgs e)

{

//This first release of the Kinect language pack doesn't have a reliable confidence model, so

//we don't use e.Result.Confidence here.

if (e.Result.Confidence < 0.70)

{

Trace.WriteLine("\nSpeech Rejected filtered, confidence: " + e.Result.Confidence);

return;

}

Trace.WriteLine("\nSpeech Recognized, confidence: " + e.Result.Confidence + ": \t{0}", e.Result.Text);

if (e.Result.Text == "computer show window")

{

this.Dispatcher.BeginInvoke((Action)delegate

{

ShowWindow();

});

}

else if (e.Result.Text == "computer hide window")

{

this.Dispatcher.BeginInvoke((Action)delegate

{

HideWindow();

});

}

else if (e.Result.Text == "computer hide circles")

{

this.Dispatcher.BeginInvoke((Action)delegate

{

this.HideCircles();

});

}

else if (e.Result.Text == "computer enable mouse"

|| e.Result.Text == "computer turn mouse on"

|| e.Result.Text == "computer turn on mouse")

{

enableMouse = true;

}

else if (e.Result.Text == "computer disable mouse"

|| e.Result.Text == "computer turn off mouse"

|| e.Result.Text == "computer turn mouse off")

{

enableMouse = false;

}

else if (e.Result.Text == "computer switch window")

{

this.Dispatcher.BeginInvoke((Action)delegate

{

System.Windows.Forms.SendKeys.SendWait("%{TAB}");

//System.Windows.Application.Current.MainWindow.Close();

});

}

else if (e.Result.Text == "computer go back" || e.Result.Text == "computer undo")

{

this.Dispatcher.BeginInvoke((Action)delegate

{

System.Windows.Forms.SendKeys.SendWait("{F2}");

//Debug.WriteLine("undo");

});

}

else if (e.Result.Text == "powerpoint full screen")

{

this.Dispatcher.BeginInvoke((Action)delegate

{

System.Windows.Forms.SendKeys.SendWait("{F5}");

//System.Windows.Application.Current.MainWindow.Close();

});

}

else if (e.Result.Text == "powerpoint return")

{

this.Dispatcher.BeginInvoke((Action)delegate

{

System.Windows.Forms.SendKeys.SendWait("{ESC}");

//System.Windows.Application.Current.MainWindow.Close();

});

}

else if (e.Result.Text == "computer close window")

{

this.Dispatcher.BeginInvoke((Action)delegate

{

System.Windows.Application.Current.MainWindow.Close();

});

}

else if (e.Result.Text == "computer show circles")

{

this.Dispatcher.BeginInvoke((Action)delegate

{

this.ShowCircles();

});

}

else if (e.Result.Text == "computer print screen")

{

this.Dispatcher.BeginInvoke((Action)delegate

{

System.Windows.Forms.SendKeys.SendWait("%{PRTSC}");

});

}

else if (e.Result.Text == "computer toggle lock")

{

this.Dispatcher.BeginInvoke((Action)delegate

{

System.Windows.Forms.SendKeys.SendWait("^`");

});

}

else if (e.Result.Text == "computer open paint")

{

this.Dispatcher.BeginInvoke((Action)delegate

{

System.Windows.Forms.SendKeys.SendWait("^{ESC}PAINT{ENTER}");

//System.Windows.Forms.SendKeys.SendWait("^{ESC}{UP}");

});

}

else if (e.Result.Text == "computer maximize window")

{

this.Dispatcher.BeginInvoke((Action)delegate

{

//System.Windows.Forms.SendKeys.SendWait("^{ESC}PAINT{ENTER}");

//System.Windows.Forms.SendKeys.SendWait("^{ESC}+{UP}");

//InputSimulator.SimulateModifiedKeyStroke(VirtualKeyCode.LWIN, VirtualKeyCode.VK\_E);

});

}

}

#endregion

}

}