```
// ------------------------------------------------------------------------------------------
------------------------------
// Verdonschot, R.G., Guillemaud, H., Rabenarivo, H., & Tamaoka, K. (2015)
// The Microsoft KINECT: a novel tool for psycholinguistic research. The Open Journal of Modern Linguistics. 5, 291-301.
// Please see this publication for additional information concerning this code.
// ------------------------------------------------------------------------------------------
------------------------------

namespace FaceTrackingBasics
{
    using System;
    using System.Collections;
    using System.Collections.Generic;
    using System.Diagnostics;
    using System.Windows;
    using System.Windows.Controls;
    using System.Windows.Media;
    using Microsoft.Kinect;
    using Microsoft.Kinect.Toolkit.FaceTracking;
    using System.Windows.Media.Media3D;
    using System.Windows.Shapes;

    using Point = System.Windows.Point;

    using OfficeOpenXml;
    using System.IO;

    /// <summary>
    /// Class that uses the Face Tracking SDK to display a face mask for
    /// tracked skeletons
    /// </summary>
    public partial class FaceTrackingViewer : UserControl, IDisposable
    {
        public static readonly DependencyProperty KinectProperty =
DependencyProperty.Register(
            "Kinect",
            typeof(KinectSensor),
            typeof(FaceTrackingViewer),
            new PropertyMetadata(
                null, (o, args) =>
((FaceTrackingViewer)o).OnSensorChanged((KinectSensor)args.OldValue,
(KinectSensor)args.NewValue)));

        /// <summary>
        /// Boolean setting when we begin the face tracking
        /// </summary>
        public bool beginTracking = false;
        /// <summary>
        /// Boolean setting when we begin the speech detection
        /// </summary>
        public static bool _beginExperiment = false;
        public void setBeginExperiment(bool begin)
        {
            _beginExperiment = begin;
        }
        public bool getBeginExperiment()
        {
            return _beginExperiment;
```

```
}

private const uint MaxMissedFrames = 100;

#region FaceTrackingPointsIds

// left on the image, but is on the right of the person

private const int LeftEye = 72;

private const int RightEye = 74;

// These points are on the external border of the lip

private const int MouthUpperLeft = 79;

private const int MouthUpperRight = 80;

private const int MouthLowerLeft = 85;

private const int MouthLowerRight = 86;

private const int MouthCornerLeft = 88;

private const int MouthCornerRight = 89;

// These points on the internal border of the lip

private const int MouthInUpperLeft = 81;

private const int MouthInUpperRight = 82;

private const int MouthInLowerLeft = 83;

private const int MouthInLowerRight = 84;

private const int MouthInUpperMiddle = 87;

// Other points of the head

private const int TopHead = 0;

private const int LeftNose = 76;

private const int Chin = 10;

#endregion FaceTrackingPointsIds

#region ConstantVectors
private static Vector3D Horizontal3D = new Vector3D(1, 0, 0);

private static Vector Horizontal2D = new Vector(1, 0);

private static Vector Vertical2D = new Vector(0, 1);

#endregion ConstantVectors

/// <summary>
```

```
/// Number of frame per seconds captured by the Kinect
/// </summary>
private const int Fps = 15;

#region SpeechDetection
/// <summary>
/// Number of keys in hashDistances over which we consider
/// that the mouth has made a significant movement.
/// It is set during the calibration.
/// </summary>
private static int _thresholdKeys = -1;
public int thresholdKeys { get; set; }
/// <summary>
/// Value of the Opening we consider that the mouth is opened.
/// It is set during the calibration.
/// </summary>
private static double _mouthOpeningTrigger = -1.0;
public double mouthOpeningTrigger { get; set; }
/// <summary>
/// Margin multiplying factor for the mouth opening trigger
/// </summary>
private const double marginMouthOpening = 1.1;
/// <summary>
/// The 3D distances are too small numbers,
/// so we multiply them by this factor before treating them.
/// </summary>
private const int MultFactor = 1000;
/// <summary>
/// This is an approximated vertical opening of the mouth
/// </summary>
private static double mouthOpening;
/// <summary>
/// Boolean retaining if the last frame was detected as
/// being speaking
/// </summary>
private static bool lastFrameSpeaking = false;
/// <summary>
/// Queue containing all the mouth Distances calculated over the last second
/// </summary>
private static Queue<Distance> mouthDistances;
/// <summary>
/// Getter for the Distances in mouthDistances
/// </summary>
/// <returns>An array of Double containing only the distances</returns>
public int[] arrayMouthDistances()
{
    int[] res = new int[mouthDistances.Count];
    int counter = 0;
    foreach (Distance d in mouthDistances)
    {
        res[counter++] = d.GetHashCode();
    }
    return res;
}
/// <summary>
/// Hashtable containing the keys of all the Distances
/// present in mouthDistances at the time
/// </summary>
```

```
        private static Hashtable hashDistances;
        /// <summary>
        /// Queue containing all the mouth Openings calculated over the last second
        /// </summary>
        private static Queue<double> mouthOpeningsQueue;
        /// <summary>
        /// Hashtable containing the keys of all the Openings
        /// present in mouthOpeningsQueue at the time
        /// </summary>
        private static Hashtable hashOpenings;

        #endregion SpeechDetection

        #region HeadMovements
        /// <summary>
        /// Variables retaining the angles calculated at the last frame,
        /// to measure the difference with the current frame
        /// </summary>
        private static double oldHeadAngleX, oldHeadAngleY, oldHeadAngleZ;
        /// <summary>
        /// Determine whether the head moves or not on different axes
        /// </summary>
        private static bool headMovX, headMovY, headMovZ;
        /// <summary>
        /// We leave a margin after the end of the detection of a head movement,
        /// for the face tracking and the values we measure
        /// to have time to stabilize again
        /// </summary>
        private static int counterEndHeadMovement;
        /// <summary>
        /// Number of frame for the margin
        /// </summary>
        private const int DurationEndMovement = 5;
        /// <summary>
        /// Thresholds of differences between the old and current angles,
        /// alllowing to set the booleans headMov
        /// </summary>
        private const double MaxVariationHeadX = 3.0,
            MaxVariationHeadY = 3.0,
            MaxVariationHeadZ = 3.0;

        #endregion HeadMovements

        private readonly Dictionary<int, SkeletonFaceTracker> trackedSkeletons = new
Dictionary<int, SkeletonFaceTracker>();

        private byte[] colorImage;

        private ColorImageFormat colorImageFormat = ColorImageFormat.Undefined;

        private short[] depthImage;

        private DepthImageFormat depthImageFormat = DepthImageFormat.Undefined;

        private bool disposed;

        private Skeleton[] skeletonData;
```

```
/// <summary>
/// Watch measuring the time through the experiment.
/// </summary>
private static System.Diagnostics.Stopwatch watch;
public System.Diagnostics.Stopwatch getWatch()
{
    return watch;
}

#region Sound
/// <summary>
/// Computed energy of the sound,
/// assimilated to its volume
/// </summary>
public static double soundEnergy;
public void setSoundEnergy(double energy)
{
    soundEnergy = energy;
}
/// <summary>
/// Number of consecutive frames when the volume
/// of the sound exceeds a definite threshold
/// </summary>
public static int counterFrameSoundSpeaking = 0;
public void resetCounterFrameSoundSpeaking()
{
    counterFrameSoundSpeaking = 0;
}
/// <summary>
/// Number of consecutive frames when the volume
/// of the sound is below a definite threshold.
/// It is initialized greater than MinFramesNotSpeaking,
/// so that at the beginning of the experiment, we consider that
/// we are not in the middle of a word/ speaking
/// </summary>
public static int counterFrameSoundNotSpeaking = MinFramesNotSpeaking + 1;
public void resetCounterFrameSoundNotSpeaking()
{
    counterFrameSoundNotSpeaking = 0;
}
/// <summary>
/// Minimum number of consecutive frames when the volume is high enough
/// to consider that a word is being spoken
/// </summary>
public static int MinFramesSpeaking = 5;
/// <summary>
/// Minimum number of consecutive frames when the volume is low
/// to consider that a word has finished being spoken
/// </summary>
public static int MinFramesNotSpeaking = 5;
/// <summary>
/// Boolean indicating if we detect speech relating
/// on the sound volume (energy)
/// </summary>
public static bool soundSpeech = false;
/// <summary>
///
/// </summary>
```

```
        private bool soundSpeakingDetected = false;
        /// <summary>
        ///
        /// </summary>
        public bool unableSoundDetection = false;
        /// <summary>
        ///
        /// </summary>
        public void initSoundDetection()
        {
            unableSoundDetection = true;
            counterFrameSoundSpeaking = 0;
            counterFrameSoundNotSpeaking = MinFramesNotSpeaking + 1;
            soundSpeech = false;
            soundSpeakingDetected = false;
        }
        /// <summary>
        /// We set the speaking decision by hysteresis
        /// </summary>
        public void updateSoundSpeech()
        {
            if (unableSoundDetection)
            {
                if (soundSpeech)
                { // Last frame sound speaking detected
                    counterFrameSoundSpeaking++;
                    soundSpeech = soundEnergy > 0.2;
                    if (soundSpeech)
                    { // still sound speak detection
                        if (counterFrameSoundSpeaking >= MinFramesSpeaking
&& !soundSpeakingDetected)
                        {
                            counterFrameSoundNotSpeaking = 0;
                            soundSpeakingDetected = true;
                            BeginSoundSpeak();
                        }
                    }
                    else
                    { // on the falling edge
                        if (counterFrameSoundSpeaking < MinFramesSpeaking)
                        { // parasite noise
                            counterFrameSoundNotSpeaking += counterFrameSoundSpeaking;
                            counterFrameSoundSpeaking = 0;
                        }
                    }
                }
                else
                {
                    counterFrameSoundNotSpeaking++;
                    soundSpeech = soundEnergy > 0.3;
                    if (!soundSpeech)
                    {
                        if (counterFrameSoundNotSpeaking >= MinFramesNotSpeaking &&
soundSpeakingDetected)
                        {
                            counterFrameSoundSpeaking = 0;
                            soundSpeakingDetected = false;
                            EndSoundSpeak();
```

```
                }
            }
            else
            { // on the rising edge
                if (counterFrameSoundNotSpeaking < MinFramesNotSpeaking)
                { // pause in the middle of a word
                    counterFrameSoundSpeaking += counterFrameSoundNotSpeaking;
                    counterFrameSoundNotSpeaking = 0;
                }
            }
        }
    }
}
#endregion Sound

#region Calibration
/// <summary>
/// Indicates if we are during the calibration
/// </summary>
private static bool _isRunningCalibration = false;
public void setIsRunningCalibration(bool runningCalibration)
{
    _isRunningCalibration = runningCalibration;
}
public bool isRunningCalibration()
{
    return _isRunningCalibration;
}
/// <summary>
/// Indicates if we are in the phase preparing the subject
/// for the calibration
/// </summary>
private static bool _isBeginningCalibration = false;
public void setIsBeginningCalibration(bool beginningCalibration)
{
    _isBeginningCalibration = beginningCalibration;
}
public bool isBeginningCalibration()
{
    return _isBeginningCalibration;
}
/// <summary>
/// Timer used to time the calibration
/// </summary>
private static System.Diagnostics.Stopwatch _calibrationTimer;
public System.Diagnostics.Stopwatch getCalibrationTimer()
{
    return _calibrationTimer;
}
/// <summary>
/// Duration of the preparation for the calibration in seconds.
/// </summary>
public const int beginningCalibrationDurationSeconds = 1;
/// <summary>
/// Duration of the calibration in seconds.
/// </summary>
public const int calibrationDurationSeconds = 2;
/// <summary>
```

```
/// Indicates if the last calibration has succeeded or not.
/// </summary>
private static bool calibrationSucceeded = true;
public bool hasCalibrationSucceeded()
{
    return calibrationSucceeded;
}
public void setCalibrationSucceeded(bool success)
{
    calibrationSucceeded = success;
}
#endregion Calibration

#region ExcelOutput
private static ExcelPackage pck;
/// <summary>
/// Creates and excel file with the arrays containing
/// the results of the experiment
/// </summary>
/// <param name="newFile">Name and location of the file</param>
public void createExcelPackage(FileInfo newFile)
{
    pck = new ExcelPackage(newFile);
    // Add the Content sheet
    ws = pck.Workbook.Worksheets.Add("Results");
    ws.Cells["A1"].Value = "Times";
    ws.Column(1).Width = 12;
    ws.Cells["B1"].Value = "Events";
    ws.Column(2).Width = 14;
    ws.Cells["C1"].Value = "Times sound";
    ws.Column(3).Width = 12;
    ws.Cells["D1"].Value = "Events sound";
    ws.Column(4).Width = 17;
    ws.Cells["F1"].Value = "Target";
    ws.Column(6).Width = 14;
    ws.Cells["G1"].Value = "Condition";
    ws.Column(7).Width = 10;
    ws.Cells["H1"].Value = "Onset";
    ws.Cells["I1"].Value = "Offset";
    ws.Cells["J1"].Value = "Sound Onset";
    ws.Column(10).Width = 12;
    ws.Cells["K1"].Value = "Sound Offset";
    ws.Column(11).Width = 12;
}
/// <summary>
/// Worksheet in which we will write the results
/// </summary>
private static ExcelWorksheet ws;
public ExcelWorksheet getWs()
{
    return ws;
}
/// <summary>
/// Contains the path to the output file
/// </summary>
public string outputFile;
#endregion ExcelOutput
```

```
#region Events
public delegate void WindowEvent();
/// <summary>
/// Thrown when we detect the beginning of a speech
/// with the visual data.
/// Suggestion of improvement: should be thrown
/// only once for a word (as for BeginSoundSpeak)
/// </summary>
public static event WindowEvent BeginSpeak;
/// <summary>
/// Thrown when we detect the end of a speech
/// with the visual data
/// </summary>
public static event WindowEvent EndSpeak;
/// <summary>
/// Thrown each time a calibration is finished,
/// to verify that it has succeeded
/// </summary>
public static event WindowEvent CalibrationFinished;
/// <summary>
/// Thrown when the tracking of the face is lost
/// </summary>
public static event WindowEvent TrackingLost;
/// <summary>
/// Thrown when the tracking of the face is regained
/// </summary>
public static event WindowEvent TrackingRegain;
/// <summary>
/// Thrown when we detect the beginning of a speech
/// with the audio data
/// </summary>
public static event WindowEvent BeginSoundSpeak;
/// <summary>
/// Thrown when we detect the end of a speech
/// with the audio data
/// </summary>
public static event WindowEvent EndSoundSpeak;
#endregion Events

public FaceTrackingViewer()
{
    this.InitializeComponent();
    mouthDistances = new Queue<Distance>(Fps);
    hashDistances = new Hashtable();
    mouthOpeningsQueue = new Queue<double>(Fps);
    hashOpenings = new Hashtable();
    watch = new Stopwatch();
    _calibrationTimer = new Stopwatch();
    watch.Start();
    counterEndHeadMovement = 0;
}

~FaceTrackingViewer()
{
    this.Dispose(false);

}
```

```
public KinectSensor Kinect
{
    get
    {
        return (KinectSensor)this.GetValue(KinectProperty);
    }

    set
    {
        this.SetValue(KinectProperty, value);
    }
}

public void Dispose()
{
    if (pck != null)
    {
        pck.Save();
    }
    this.Dispose(true);
    GC.SuppressFinalize(this);
}

protected virtual void Dispose(bool disposing)
{
    if (!this.disposed)
    {
        this.ResetFaceTracking();

        this.disposed = true;
    }
}

protected override void OnRender(DrawingContext drawingContext)
{
    base.OnRender(drawingContext);
    foreach (SkeletonFaceTracker faceInformation in this.trackedSkeletons.Values)
    {
        faceInformation.DrawFaceModel(drawingContext);
    }
}

/// <summary>
/// Retrieves the canvas created in SkeletonFaceTracker
/// to draw the mouth of the subject
/// </summary>
/// <returns>Canvas in which the mouth of the subject is drawn</returns>
public Canvas canvasDrawingMouth()
{
    foreach (SkeletonFaceTracker faceInformation in this.trackedSkeletons.Values)
    {
        Canvas res = faceInformation.DrawMouthModel();
        if (res != null)
            return res;
    }
    return null;
}
```

```
        /// <summary>
        ///
        /// </summary>
        /// <returns>The window containing the information</returns>
        public WindowInformation getWindowInfo()
        {
            foreach (SkeletonFaceTracker faceInformation in this.trackedSkeletons.Values)
            {
                WindowInformation res = faceInformation.windowInfo;
                if (res != null)
                    return res;
            }
            return null;
        }

        private void OnAllFramesReady(object sender, AllFramesReadyEventArgs
allFramesReadyEventArgs)
        {
            ColorImageFrame colorImageFrame = null;
            DepthImageFrame depthImageFrame = null;
            SkeletonFrame skeletonFrame = null;

            // Wait for beginning of experiment
            if (!beginTracking)
            {
                return;
            }
            try
            {
                colorImageFrame = allFramesReadyEventArgs.OpenColorImageFrame();
                depthImageFrame = allFramesReadyEventArgs.OpenDepthImageFrame();
                skeletonFrame = allFramesReadyEventArgs.OpenSkeletonFrame();

                if (colorImageFrame == null || depthImageFrame == null || skeletonFrame
== null)
                {
                    return;
                }

                // Check for image format changes.  The FaceTracker doesn't
                // deal with that so we need to reset.
                if (this.depthImageFormat != depthImageFrame.Format)
                {
                    this.ResetFaceTracking();
                    this.depthImage = null;
                    this.depthImageFormat = depthImageFrame.Format;
                }

                if (this.colorImageFormat != colorImageFrame.Format)
                {
                    this.ResetFaceTracking();
                    this.colorImage = null;
                    this.colorImageFormat = colorImageFrame.Format;
                }

                // Create any buffers to store copies of the data we work with
                if (this.depthImage == null)
                {
```

```
                    this.depthImage = new short[depthImageFrame.PixelDataLength];
                }

                if (this.colorImage == null)
                {
                    this.colorImage = new byte[colorImageFrame.PixelDataLength];
                }

                // Get the skeleton information
                if (this.skeletonData == null || this.skeletonData.Length != 
skeletonFrame.SkeletonArrayLength)
                {
                    this.skeletonData = new Skeleton[skeletonFrame.SkeletonArrayLength];
                }

                colorImageFrame.CopyPixelDataTo(this.colorImage);
                depthImageFrame.CopyPixelDataTo(this.depthImage);
                skeletonFrame.CopySkeletonDataTo(this.skeletonData);

                // Update the list of trackers and the trackers with the current frame 
information
                foreach (Skeleton skeleton in this.skeletonData)
                {
                    if (skeleton.TrackingState == SkeletonTrackingState.Tracked
                        || skeleton.TrackingState == SkeletonTrackingState.PositionOnly)
                    {
                        // We want to keep a record of any skeleton, tracked or 
untracked.
                        if (!this.trackedSkeletons.ContainsKey(skeleton.TrackingId))
                        {
                            this.trackedSkeletons.Add(skeleton.TrackingId, new 
SkeletonFaceTracker());
                        }

                        // Give each tracker the updated frame.
                        SkeletonFaceTracker skeletonFaceTracker;
                        if (this.trackedSkeletons.TryGetValue(skeleton.TrackingId, out 
skeletonFaceTracker))
                        {
                            skeletonFaceTracker.OnFrameReady(this.Kinect, 
colorImageFormat, colorImage, depthImageFormat, depthImage, skeleton);
                            skeletonFaceTracker.LastTrackedFrame = 
skeletonFrame.FrameNumber;
                        }
                    }
                }

                this.RemoveOldTrackers(skeletonFrame.FrameNumber);

                this.InvalidateVisual();
            }
            finally
            {
                if (colorImageFrame != null)
                {
                    colorImageFrame.Dispose();
                }
```

```
                if (depthImageFrame != null)
                {
                    depthImageFrame.Dispose();
                }

                if (skeletonFrame != null)
                {
                    skeletonFrame.Dispose();
                }
            }
        }

        private void OnSensorChanged(KinectSensor oldSensor, KinectSensor newSensor)
        {
            if (oldSensor != null)
            {
                oldSensor.AllFramesReady -= this.OnAllFramesReady;
                this.ResetFaceTracking();
            }

            if (newSensor != null)
            {
                newSensor.AllFramesReady += this.OnAllFramesReady;
            }
        }

        /// <summary>
        /// Clear out any trackers for skeletons we haven't heard from for a while
        /// </summary>
        private void RemoveOldTrackers(int currentFrameNumber)
        {
            var trackersToRemove = new List<int>();

            foreach (var tracker in this.trackedSkeletons)
            {
                uint missedFrames = (uint)currentFrameNumber -
(uint)tracker.Value.LastTrackedFrame;
                if (missedFrames > MaxMissedFrames)
                {
                    // There have been too many frames since we last saw this skeleton
                    trackersToRemove.Add(tracker.Key);
                }
            }

            foreach (int trackingId in trackersToRemove)
            {
                this.RemoveTracker(trackingId);
            }
        }

        private void RemoveTracker(int trackingId)
        {
            this.trackedSkeletons[trackingId].Dispose();
            this.trackedSkeletons.Remove(trackingId);
        }

        private void ResetFaceTracking()
        {
```

```
            foreach (int trackingId in new List<int>(this.trackedSkeletons.Keys))
            {
                this.RemoveTracker(trackingId);
            }
        }

        private class SkeletonFaceTracker : IDisposable
        {
            private static FaceTriangle[] faceTriangles;

            private EnumIndexableCollection<FeaturePoint, PointF> facePoints;

            private EnumIndexableCollection<FeaturePoint, Vector3DF> facePoints3D;

            private FaceTracker faceTracker;

            private bool lastFaceTrackSucceeded;

            /// <summary>
            ///
            /// </summary>
            private bool faceTrackingLost = true;

            private SkeletonTrackingState skeletonTrackingState;

            public int LastTrackedFrame { get; set; }

            /// <summary>
            /// Special window to display information on
            /// variables of the experiment
            /// </summary>
            public WindowInformation windowInfo = new WindowInformation();

            public void Dispose()
            {
                if (this.faceTracker != null)
                {
                    this.faceTracker.Dispose();
                    this.faceTracker = null;
                }
                if (windowInfo != null)
                {
                    windowInfo.Close();
                }
            }

            private bool PointValid(int id)
            {
                return id < 90 && id >= 79;
            }

            private bool TriangleValid(FaceTriangle triangle)
            {
                return PointValid(triangle.First) || PointValid(triangle.Second) ||
PointValid(triangle.Third);
            }

            /// <summary>
```

```
            /// Draws a model of the mouth in a canvas
            /// </summary>
            /// <returns>Canvas</returns>
            public Canvas DrawMouthModel()
            {
                if (!this.lastFaceTrackSucceeded || this.skeletonTrackingState !=
SkeletonTrackingState.Tracked)
                {
                    return null;
                }
                Canvas res = new Canvas();

                // Adding the tracking points of the mouth
                Point mouthCornerLeft = new Point(facePoints[MouthCornerLeft].X,
facePoints[MouthCornerLeft].Y);
                Point mouthCornerRight = new Point(facePoints[MouthCornerRight].X,
facePoints[MouthCornerRight].Y);
                Point mouthBarycentre = new Point((mouthCornerLeft.X +
mouthCornerRight.X) / 2, (mouthCornerLeft.Y + mouthCornerRight.Y) / 2);
                for (int i = MouthUpperLeft; i < 90; i++)
                {
                    Ellipse ellipse = new Ellipse();
                    ellipse.Stroke = System.Windows.Media.Brushes.Black;
                    ellipse.Fill = System.Windows.Media.Brushes.Red;
                    ellipse.HorizontalAlignment = HorizontalAlignment.Left;
                    ellipse.VerticalAlignment = VerticalAlignment.Center;
                    ellipse.Width = 2;
                    ellipse.Height = 2;
                    double marginLeft = facePoints[i].X - mouthBarycentre.X;
                    double marginTop = facePoints[i].Y - mouthBarycentre.Y;
                    ellipse.Margin = new Thickness(marginLeft * 4, marginTop * 4, 0, 0);

                    res.Children.Add(ellipse);
                }

                // Adding the lines for the lips
                Line line1 = new Line();
                line1.Stroke = Brushes.Red;
                line1.X1 = (facePoints[MouthCornerLeft].X - mouthBarycentre.X) * 4;
                line1.Y1 = (facePoints[MouthCornerLeft].Y - mouthBarycentre.Y) * 4;
                line1.X2 = (facePoints[MouthUpperLeft].X - mouthBarycentre.X) * 4;
                line1.Y2 = (facePoints[MouthUpperLeft].Y - mouthBarycentre.Y) * 4;

                Line line2 = new Line();
                line2.Stroke = Brushes.Red;
                line2.X1 = (facePoints[MouthUpperLeft].X - mouthBarycentre.X) * 4;
                line2.Y1 = (facePoints[MouthUpperLeft].Y - mouthBarycentre.Y) * 4;
                line2.X2 = (facePoints[MouthUpperRight].X - mouthBarycentre.X) * 4;
                line2.Y2 = (facePoints[MouthUpperRight].Y - mouthBarycentre.Y) * 4;

                Line line3 = new Line();
                line3.Stroke = Brushes.Red;
                line3.X1 = (facePoints[MouthUpperRight].X - mouthBarycentre.X) * 4;
                line3.Y1 = (facePoints[MouthUpperRight].Y - mouthBarycentre.Y) * 4;
                line3.X2 = (facePoints[MouthCornerRight].X - mouthBarycentre.X) * 4;
                line3.Y2 = (facePoints[MouthCornerRight].Y - mouthBarycentre.Y) * 4;

                Line line4 = new Line();
```

```
line4.Stroke = Brushes.Red;
line4.X1 = (facePoints[MouthCornerRight].X - mouthBarycentre.X) * 4;
line4.Y1 = (facePoints[MouthCornerRight].Y - mouthBarycentre.Y) * 4;
line4.X2 = (facePoints[MouthLowerRight].X - mouthBarycentre.X) * 4;
line4.Y2 = (facePoints[MouthLowerRight].Y - mouthBarycentre.Y) * 4;

Line line5 = new Line();
line5.Stroke = Brushes.Red;
line5.X1 = (facePoints[MouthLowerRight].X - mouthBarycentre.X) * 4;
line5.Y1 = (facePoints[MouthLowerRight].Y - mouthBarycentre.Y) * 4;
line5.X2 = (facePoints[MouthLowerLeft].X - mouthBarycentre.X) * 4;
line5.Y2 = (facePoints[MouthLowerLeft].Y - mouthBarycentre.Y) * 4;

Line line6 = new Line();
line6.Stroke = Brushes.Red;
line6.X1 = (facePoints[MouthLowerLeft].X - mouthBarycentre.X) * 4;
line6.Y1 = (facePoints[MouthLowerLeft].Y - mouthBarycentre.Y) * 4;
line6.X2 = (facePoints[MouthCornerLeft].X - mouthBarycentre.X) * 4;
line6.Y2 = (facePoints[MouthCornerLeft].Y - mouthBarycentre.Y) * 4;

Line line7 = new Line();
line7.Stroke = Brushes.Red;
line7.X1 = (facePoints[MouthCornerLeft].X - mouthBarycentre.X) * 4;
line7.Y1 = (facePoints[MouthCornerLeft].Y - mouthBarycentre.Y) * 4;
line7.X2 = (facePoints[MouthInUpperLeft].X - mouthBarycentre.X) * 4;
line7.Y2 = (facePoints[MouthInUpperLeft].Y - mouthBarycentre.Y) * 4;

Line line8 = new Line();
line8.Stroke = Brushes.Red;
line8.X1 = (facePoints[MouthInUpperLeft].X - mouthBarycentre.X) * 4;
line8.Y1 = (facePoints[MouthInUpperLeft].Y - mouthBarycentre.Y) * 4;
line8.X2 = (facePoints[MouthInUpperRight].X - mouthBarycentre.X) * 4;
line8.Y2 = (facePoints[MouthInUpperRight].Y - mouthBarycentre.Y) * 4;

Line line9 = new Line();
line9.Stroke = Brushes.Red;
line9.X1 = (facePoints[MouthInUpperRight].X - mouthBarycentre.X) * 4;
line9.Y1 = (facePoints[MouthInUpperRight].Y - mouthBarycentre.Y) * 4;
line9.X2 = (facePoints[MouthCornerRight].X - mouthBarycentre.X) * 4;
line9.Y2 = (facePoints[MouthCornerRight].Y - mouthBarycentre.Y) * 4;

Line line10 = new Line();
line10.Stroke = Brushes.Red;
line10.X1 = (facePoints[MouthCornerRight].X - mouthBarycentre.X) * 4;
line10.Y1 = (facePoints[MouthCornerRight].Y - mouthBarycentre.Y) * 4;
line10.X2 = (facePoints[MouthInLowerRight].X - mouthBarycentre.X) * 4;
line10.Y2 = (facePoints[MouthInLowerRight].Y - mouthBarycentre.Y) * 4;

Line line11 = new Line();
line11.Stroke = Brushes.Red;
line11.X1 = (facePoints[MouthInLowerRight].X - mouthBarycentre.X) * 4;
line11.Y1 = (facePoints[MouthInLowerRight].Y - mouthBarycentre.Y) * 4;
line11.X2 = (facePoints[MouthInLowerLeft].X - mouthBarycentre.X) * 4;
line11.Y2 = (facePoints[MouthInLowerLeft].Y - mouthBarycentre.Y) * 4;

Line line12 = new Line();
line12.Stroke = Brushes.Red;
line12.X1 = (facePoints[MouthInLowerLeft].X - mouthBarycentre.X) * 4;
```

```
                line12.Y1 = (facePoints[MouthInLowerLeft].Y - mouthBarycentre.Y) * 4;
                line12.X2 = (facePoints[MouthCornerLeft].X - mouthBarycentre.X) * 4;
                line12.Y2 = (facePoints[MouthCornerLeft].Y - mouthBarycentre.Y) * 4;

                res.Children.Add(line1);
                res.Children.Add(line2);
                res.Children.Add(line3);
                res.Children.Add(line4);
                res.Children.Add(line5);
                res.Children.Add(line6);
                res.Children.Add(line7);
                res.Children.Add(line8);
                res.Children.Add(line9);
                res.Children.Add(line10);
                res.Children.Add(line11);
                res.Children.Add(line12);

                return res;
            }

            /// <summary>
            /// Computes the speech detection,
            /// and fills in the information window
            /// </summary>
            /// <param name="drawingContext"></param>
            public void DrawFaceModel(DrawingContext drawingContext)
            {
                if (!this.lastFaceTrackSucceeded || this.skeletonTrackingState !=
SkeletonTrackingState.Tracked)
                { // Tracking of the face lost, we do not continue
                    // We mask the information in the window
                    windowInfo.textBoxDist3D.Text = "/";
                    windowInfo.textBoxMouthOp.Text = "/";
                    windowInfo.textBoxMovHead.Text = "/";
                    windowInfo.textBoxMOTrigger.Text = _mouthOpeningTrigger < 0 ? "/" :
_mouthOpeningTrigger.ToString();
                    windowInfo.textBoxKeysTrigger.Text = _thresholdKeys < 0 ? "/" :
_thresholdKeys.ToString();
                    windowInfo.textBoxNbKeysDist.Text = "/";
                    windowInfo.textBoxNbKeysMO.Text = "/";
                    windowInfo.textBoxSoundEnergy.Text = soundEnergy.ToString();
                    windowInfo.textBoxSoundSpeaking.Text = soundSpeech.ToString();
                    // We throw an event when the tracking has just been lost
                    if (!faceTrackingLost)
                    {
                        faceTrackingLost = true;
                        calibrationSucceeded = false;
                        TrackingLost();
                    }
                    return;
                }

                // We throw an event when the tracking has just been regained
                if (faceTrackingLost)
                {
                    faceTrackingLost = false;
                    TrackingRegain();
                }
```

```
                // List of the tracking points of the face projected on 2D
                var faceModelPts = new List<Point>();
                for (int i = 0; i < this.facePoints.Count; i++)
                {
                    faceModelPts.Add(new Point(this.facePoints[i].X + 0.5f,
this.facePoints[i].Y + 0.5f));
                }


////////////////////////////////////////////////////////////////////////////// 3D
points
                var faceModelPts3D = new List<Vector3D>();
                for (int i = 0; i < this.facePoints3D.Count; i++)
                {
                    faceModelPts3D.Add(new Vector3D(this.facePoints3D[i].X * MultFactor,
this.facePoints3D[i].Y * MultFactor, this.facePoints3D[i].Z * MultFactor));
                }
                double length_vertical_3D =
Vector3D.Subtract(faceModelPts3D[MouthInUpperLeft],
faceModelPts3D[MouthInLowerLeft]).Length +
                    Vector3D.Subtract(faceModelPts3D[MouthInUpperRight],
faceModelPts3D[MouthInLowerRight]).Length;
                double length_horizontal_3D =
Vector3D.Subtract(faceModelPts3D[MouthCornerLeft],
faceModelPts3D[MouthCornerRight]).Length;
                double distance_opening_3D = length_horizontal_3D + length_vertical_3D;

//////////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////////// 2D
projected points
                double length_vertical = Point.Subtract(faceModelPts[MouthInUpperLeft],
faceModelPts[MouthInLowerLeft]).Length;
                double length_horizontal = Point.Subtract(faceModelPts[MouthCornerLeft],
faceModelPts[MouthCornerRight]).Length;
                double distance_opening = length_horizontal + length_vertical;

//////////////////////////////////////////////////////////////////////////////

                // Information displayed in the window
                windowInfo.textBoxDist3D.Text = distance_opening_3D.ToString();
                windowInfo.textBoxMouthOp.Text = mouthOpening.ToString();
                windowInfo.textBoxMovHead.Text = isMovingHead().ToString();
                windowInfo.textBoxMOTrigger.Text = _mouthOpeningTrigger < 0 ? "/" :
_mouthOpeningTrigger.ToString();
                windowInfo.textBoxKeysTrigger.Text = _thresholdKeys < 0 ? "/" :
_thresholdKeys.ToString();
                windowInfo.textBoxNbKeysDist.Text = hashDistances.Keys.Count.ToString();
                windowInfo.textBoxNbKeysMO.Text = hashOpenings.Keys.Count.ToString();
                windowInfo.textBoxSoundEnergy.Text = soundEnergy.ToString();
                windowInfo.textBoxSoundSpeaking.Text = soundSpeech.ToString();

                // Speaking detection
                if (_beginExperiment)
                {
                    if (mouthDistances.Count < Fps)
```

```
                    { // initialisation (after loss of tracking)
                        //drawingContext.DrawText(new FormattedText("init",
System.Globalization.CultureInfo.GetCultureInfo("en-us"), FlowDirection.LeftToRight, new
Typeface("Verdana"), 15.0, Brushes.Red), new Point(0, 25));
                        //drawingContext.DrawText(new FormattedText("mDcount: " +
mouthDistances.Count, System.Globalization.CultureInfo.GetCultureInfo("en-us"),
FlowDirection.LeftToRight, new Typeface("Verdana"), 15.0, Brushes.Red), new Point(20,
50));
                        lastFrameSpeaking = false;
                    }
                    else if (_isRunningCalibration)
                    {
                        lastFrameSpeaking = false;
                    }
                    else if (  // Speaking
                        (( // Speaking detected
                        // in the middle of a word
                        hashDistances.Keys.Count > _thresholdKeys && lastFrameSpeaking
                        // mouth open
                        || mouthOpening >= _mouthOpeningTrigger * marginMouthOpening
                        // big variations
                        || hashDistances.Keys.Count > _thresholdKeys + 2)
                        &&
                        ( // Not false detection
                        !isMovingHead())
                        ||
                        ( // Probable false detection end of speaking
                        lastFrameSpeaking
                        && isMovingHead()))
                        &&
                        ( // MouthOpening does not change
                        hashOpenings.Keys.Count > 1)
                        )
                    {
                        //drawingContext.DrawText(new FormattedText("Speaking",
System.Globalization.CultureInfo.GetCultureInfo("en-us"), FlowDirection.LeftToRight, new
Typeface("Verdana"), 15.0, Brushes.Red), new Point(0, 25));
                        //drawingContext.DrawText(new
FormattedText(hashDistances.Keys.Count.ToString(),
System.Globalization.CultureInfo.GetCultureInfo("en-us"), FlowDirection.LeftToRight, new
Typeface("Verdana"), 15.0, Brushes.Red), new Point(0, 50));
                        if (!lastFrameSpeaking)
                        { // First frame we detect a speaking
                            BeginSpeak();
                        }
                        lastFrameSpeaking = true;
                    }
                    else   // Not speaking
                    {
                        if (lastFrameSpeaking)
                        { // First frame we don't detect a speaking
                            EndSpeak();
                        }
                        //drawingContext.DrawText(new FormattedText("...",
System.Globalization.CultureInfo.GetCultureInfo("en-us"), FlowDirection.LeftToRight, new
Typeface("Verdana"), 15.0, Brushes.Red), new Point(0, 25));
                        lastFrameSpeaking = false;
```

```
                        //drawingContext.DrawText(new
FormattedText(hashDistances.Keys.Count.ToString(),
System.Globalization.CultureInfo.GetCultureInfo("en-us"), FlowDirection.LeftToRight, new
Typeface("Verdana"), 15.0, Brushes.Red), new Point(0, 50));
                    }
                }

            }

            /// <summary>
            /// Updates the face tracking information for this skeleton,
            /// and the data containing the distances measured.
            /// Also takes charge of the calibration.
            /// </summary>
            internal void OnFrameReady(KinectSensor kinectSensor, ColorImageFormat
colorImageFormat, byte[] colorImage, DepthImageFormat depthImageFormat, short[]
depthImage, Skeleton skeletonOfInterest)
            {
                this.skeletonTrackingState = skeletonOfInterest.TrackingState;

                if (this.skeletonTrackingState != SkeletonTrackingState.Tracked)
                {
                    // nothing to do with an untracked skeleton.
                    return;
                }

                if (this.faceTracker == null)
                {
                    try
                    {
                        this.faceTracker = new FaceTracker(kinectSensor);
                    }
                    catch (InvalidOperationException)
                    {
                        // During some shutdown scenarios the FaceTracker
                        // is unable to be instantiated.  Catch that exception
                        // and don't track a face.
                        Debug.WriteLine("AllFramesReady - creating a new FaceTracker
threw an InvalidOperationException");
                        this.faceTracker = null;
                    }
                }

                if (this.faceTracker != null)
                {
                    FaceTrackFrame frame = this.faceTracker.Track(
                        colorImageFormat, colorImage, depthImageFormat, depthImage,
skeletonOfInterest);

                    this.lastFaceTrackSucceeded = frame.TrackSuccessful;
                    if (!this.lastFaceTrackSucceeded || this.skeletonTrackingState !=
SkeletonTrackingState.Tracked)
                    {
                        mouthDistances.Clear();
                        hashDistances.Clear();
                    }

                    if (this.lastFaceTrackSucceeded)
```

```
                    {
                        if (faceTriangles == null)
                        {
                            // only need to get this once.  It doesn't change.
                            faceTriangles = frame.GetTriangles();
                        }

                        this.facePoints = frame.GetProjected3DShape();


////////////////////////////////////////////////////////////////////////////
Distance opening 3D
                        this.facePoints3D = frame.Get3DShape();
                        var faceModelPts3D = new List<Vector3D>();
                        for (int i = 0; i < this.facePoints3D.Count; i++)
                        {
                            faceModelPts3D.Add(new Vector3D(this.facePoints3D[i].X *
MultFactor, this.facePoints3D[i].Y * MultFactor, this.facePoints3D[i].Z * MultFactor));
                        }
                        double length_vertical_3D =
Vector3D.Subtract(faceModelPts3D[MouthInUpperLeft],
faceModelPts3D[MouthInLowerLeft]).Length +
                            Vector3D.Subtract(faceModelPts3D[MouthInUpperRight],
faceModelPts3D[MouthInLowerRight]).Length; double length_horizontal_3D =
Vector3D.Subtract(faceModelPts3D[MouthCornerLeft],
faceModelPts3D[MouthCornerRight]).Length;
                        Distance distance_opening_3D = new Distance(length_horizontal_3D
+ length_vertical_3D);

////////////////////////////////////////////////////////////////////////////


//////////////////////////////////////////////////////////////////////////// angles
of the head
                        Vector3D eyes = Vector3D.Subtract(faceModelPts3D[RightEye],
faceModelPts3D[LeftEye]);
                        Vector projectedEyesY = new Vector(eyes.X, eyes.Z);

                        Vector3D verticalHead =
Vector3D.Subtract(faceModelPts3D[TopHead], faceModelPts3D[LeftNose]);
                        Vector projectedVerticalHeadZ = new Vector(verticalHead.X,
verticalHead.Y);
                        Vector projectedVerticalHeadX = new Vector(verticalHead.Y,
verticalHead.Z);

                        double newHeadAngleX =
Vector.AngleBetween(projectedVerticalHeadX, Horizontal2D);
                        double newHeadAngleY =
Vector.AngleBetween(projectedVerticalHeadZ, Horizontal2D);
                        double newHeadAngleZ = Vector.AngleBetween(projectedEyesY,
Horizontal2D);

////////////////////////////////////////////////////////////////////////////

                        // Filling of mouthDistances
                        var faceModelPts = new List<Point>();

                        for (int i = 0; i < this.facePoints.Count; i++)
```

```
                        {
                            faceModelPts.Add(new Point(this.facePoints[i].X + 0.5f,
this.facePoints[i].Y + 0.5f));
                        }
                        double length_vertical =
Point.Subtract(faceModelPts[MouthInUpperLeft], faceModelPts[MouthInLowerLeft]).Length;
                        double length_horizontal =
Point.Subtract(faceModelPts[MouthCornerLeft], faceModelPts[MouthCornerRight]).Length;
                        Distance distance_opening = new Distance(length_horizontal +
length_vertical);

                        if (mouthDistances.Count == Fps)
                        {
                            Distance oldDistance = mouthDistances.Dequeue();
                            hashDistances.Remove(oldDistance.GetHashCode());
                            if
(!hashDistances.ContainsKey(distance_opening_3D.GetHashCode()))
                            {
                                hashDistances.Add(distance_opening_3D.GetHashCode(),
distance_opening_3D);
                            }
                            // Update bools movements head
                            headMovX = (Math.Abs(newHeadAngleX - oldHeadAngleX) >
MaxVariationHeadX);
                            headMovY = (Math.Abs(newHeadAngleY - oldHeadAngleY) >
MaxVariationHeadY);
                            headMovZ = (Math.Abs(newHeadAngleZ - oldHeadAngleZ) >
MaxVariationHeadZ);
                            if (headMovX || headMovY || headMovZ)
                            {
                                counterEndHeadMovement = DurationEndMovement;
                            }
                        }
                        else if (mouthDistances.Count > Fps)
                        {
                            while (mouthDistances.Count >= Fps)
                            {
                                hashDistances.Remove(mouthDistances.Dequeue());
                            }
                        }
                        mouthDistances.Enqueue(distance_opening_3D);

////////////////////////////////////////////////////////////////////////////////

                        // Update head angles
                        oldHeadAngleX = newHeadAngleX;
                        oldHeadAngleY = newHeadAngleY;
                        oldHeadAngleZ = newHeadAngleZ;

                        // Update head counterEndMovement
                        counterEndHeadMovement = counterEndHeadMovement <= 0 ? 0 :
counterEndHeadMovement - 1;

                        // Vertical mouth opening
                        Vector3D middleLowerLip = new
Vector3D((faceModelPts3D[MouthInLowerLeft].X + faceModelPts3D[MouthInLowerRight].X) / 2,
(faceModelPts3D[MouthInLowerLeft].Y + faceModelPts3D[MouthInLowerRight].Y) / 2,
(faceModelPts3D[MouthInLowerLeft].Z + faceModelPts3D[MouthInLowerRight].Z) / 2);
```

```
                        mouthOpening = Vector3D.Subtract(middleLowerLip,
faceModelPts3D[MouthInUpperMiddle]).Length;

                        // Filling of mouthOpeningsQueue
                        if (mouthOpeningsQueue.Count == Fps)
                        {
                            double oldOpening = mouthOpeningsQueue.Dequeue();
                            int oldHashCode = (int)Math.Round(oldOpening);
                            hashOpenings.Remove(oldHashCode);
                            int hashCode = (int)Math.Round(mouthOpening);
                            if (!hashOpenings.ContainsKey(hashCode))
                            {
                                hashOpenings.Add(hashCode, mouthOpening);
                            }
                        }
                        else if (mouthOpeningsQueue.Count > Fps)
                        {
                            while (mouthOpeningsQueue.Count >= Fps)
                            {
                                hashOpenings.Remove(mouthOpeningsQueue.Dequeue());
                            }
                        }
                        mouthOpeningsQueue.Enqueue(mouthOpening);

                        // Calibration
                        if (_calibrationTimer.Elapsed.Seconds >
beginningCalibrationDurationSeconds + calibrationDurationSeconds)
                        {
                            // End of the calibration
                            _calibrationTimer.Reset();
                            _isRunningCalibration = false;
                                CalibrationFinished();
                        }
                        else if (_calibrationTimer.Elapsed.Seconds >
beginningCalibrationDurationSeconds)
                        {
                            // Begin the calibration
                            _isBeginningCalibration = false;
                            // Re-initialize the maxima
                            _mouthOpeningTrigger = 0;
                            _thresholdKeys = 0;
                            _isRunningCalibration = true;
                        }

                        if (_isRunningCalibration)
                        {
                            // calibration mouthOpening
                            // measure the maximum mouth opening distance when closed
                            if (mouthOpening > _mouthOpeningTrigger)
                            {
                                _mouthOpeningTrigger = mouthOpening;
                            }

                            // calibration ThresholdKeys
                            // measure the maximum number of keys when the mouth is
closed
                            if (hashDistances.Keys.Count > _thresholdKeys)
                            {
```

```
                            _thresholdKeys = hashDistances.Keys.Count;
                        }

                        // check that there is no movement during calibration
                        if (isMovingHead())
                        {
                            calibrationSucceeded = false;
                        }
                    }

                }
            }

        }

        private bool isMovingHead()
        {
            return counterEndHeadMovement > 0;
        }

        private struct FaceModelTriangle
        {
            public Point P1;
            public Point P2;
            public Point P3;
        }

    }

    private struct Distance
    {
        public double dist;

        public Distance(double d)
        {
            dist = d;
        }

        public override bool Equals(object obj)
        {
            if (obj == null || !(obj is Distance))
                return false;
            else
                return dist == ((Distance)obj).dist;
        }

        public override int GetHashCode()
        {
            return (int)dist;
        }
    }

}
}
```