**ABSTRACT**

A new approach to constraint-based path selection for dynamic routing and wavelength allocation in optical Air ride suspension carries the load on each axle with a pressurized air bag much like a high pressure balloon. Air ride suspension systems have been in common use for over forty years and have proven to provide the smoothest and most shock-free ride of any known vehicle suspension system. Modern air bags are constructed using the same methods as a tire by using high strength cords which are then encapsulated in rubber. These units are very durable in service and have a proven life of many years. In addition to providing extremely smooth ride quality, air ride suspension also provides other important features. First, the system automatically adjusts air pressure in the air bag so that the trailer always rides at the same height, whether lightly loaded or heavily loaded. This allows the suspension system to always provide the maximum usable wheel travel independent of trailer load. In addition, the higher air bag pressure associated with higher trailer loads automatically provides a stiffer suspension which is exactly what is required for a smooth ride.

The lower air bag pressure for lightly loaded conditions automatically provides for a softer suspension, thus providing the same ride quality for all trailer loading conditions. Since each axle is independently supported by its own air bag, the air ride suspension is a truly, fully independent suspension system. The automatic control of the air bag pressure is accomplished by a solid state electronic control system specifically designed and packaged for vehicle use. This system continuously monitors the "ride height" of the trailer suspension and increases air pressure if the ride height is too low, by turning on an on-board air compressor. The air compressor stops automatically when the proper ride height is reached. If the ride height is too high, an automatic vent valve vents excess air pressure and stops venting when the proper ride height is reached. All required electrical power is provided by a 12 volt battery contained in the trailer equipment compartment.

Along with this suspension system the IP sensor is going to attached and this IP sensor will automatically alert the system and make the call to the ambulance and to the number given. So this emergency system is very helpful in highways.

**BLOCK DIAGRAM CODING**

**BRAKE SENSOR**

**DRIVER SEAT BELT SENSOR**

**CRASH SENSOR**

**POWER**

**THROTTLE POSITION SENSOR**

**ENGINE RPM SENSOR**

**IP SENSOR**

**AIR BAGS**

**WARNING DISPLAY**

**SPEED SENSOR**

**TRANSMISSION CONTROL SENSOR**

**EDR CONTROLLER**

**CODING**

using System;

using System.IO;

using System.Diagnostics;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Linq;

using System.Text;

using System.Threading;

using Microsoft.Kinect;

using Microsoft.Kinect.Toolkit;

using Microsoft.Kinect.Toolkit.FaceTracking;

using Microsoft.Xna.Framework;

using System.Windows;

using System.Windows.Forms;

using System.Windows.Media;

using System.Windows.Media.Imaging;

using System.Drawing;

using System.Drawing.Imaging;

using System.Runtime.InteropServices;

using System.Runtime.Serialization.Formatters.Binary;

namespace WindowsFormsApplication1

{

public partial class M5IAS : Form

{

public M5IAS()

{

//Load Splash Screen while Kinect and Main screen load

Form2.ShowSplashScreen();

InitializeComponent();

//Timer to Process new data

RefreshTimer.Interval = 50;

RefreshTimer.Tick += RefreshWindow;

// Memory Allocation for Bitmaps

Marshal.FreeHGlobal(colorPtr);

colorPtr = Marshal.AllocHGlobal(1228800);

Marshal.FreeHGlobal(depthPtr);

depthPtr = Marshal.AllocHGlobal(1228800);

Marshal.FreeHGlobal(compositePtr);

compositePtr = Marshal.AllocHGlobal(1228800);

// Start Kinect and timer. Close splash screen

Kinect();

RefreshTimer.Start();

Form2.CloseForm();

}

// SubModule Handles GUI Refresh and Kinect Data Refresh

public void RefreshWindow(object sender, EventArgs e)

{

//Begin Data Processing

//If Kinect data exsists copy into local arrays for processing

Stopwatch stopwatch = new Stopwatch();

stopwatch.Start();

if( colorDataTemp != null && depthDataTemp != null)

{

KinectLock.WaitOne();

colorDataTemp.CopyTo(colorData,0);

depthDataTemp.CopyTo(depthData,0);

skeletonData = Skeleton\_Clone(skeletonDataTemp);

KinectLock.ReleaseMutex();

}

//Image Processing

if(colorData != null && depthData != null )

{

//Convert Kinect image arrays into Bitmaps

kinectDepthBitmap = DepthDataFormatter(depthData);

kinectCompositeBitmap = PixelDataToBitmap(colorData, compositePtr);

GetSkeleton();

//Check Skeleton and Face Tracking status

if (skeleton != null && driver == true)

{

SkelTrackVal.Text = "Tracking";

SkeletonToComposite();

}

else

{

SkelTrackVal.Text = "Untracked";

this.LOFHeadProxVal.Text = "0.00";

this.LOFDistanceVal.Text = "0.00";

this.HeadOrienRollVal.Text = "0.00";

this.HeadOrienYawVal.Text = "0.00";

this.HeadOrienPitchVal.Text = "0.00";

this.textBox1.Text = "0.00";

this.textBox2.Text = "0.00";

//this.textBox3.Text = "0.00";

this.textBox4.Text = "0.00";

this.textBox5.Text = "0.00";

this.textBox6.Text = "0.00";

this.textBox7.Text = "0.00";

//this.textBox8.Text = "0.00";

//this.textBox9.Text = "0.00";

this.textBox10.Text = "0.00";

}

// Check for Face Tracking

if (facePoints != null && faceTriangles != null)

{

FaceMaskToComposite();

FaceTrackingStatus.Text = "Tracking";

HeadOrienRollVal.Text = headRoll.ToString("F");

HeadOrienYawVal.Text = headYaw.ToString("F");

HeadOrienPitchVal.Text = headPitch.ToString("F");

}

else

{

FaceTrackingStatus.Text = "Untracked";

}

try

{

headX = skeleton.Joints[JointType.Head].Position.X \* 1000;

headY = skeleton.Joints[JointType.Head].Position.Y \* 1000;

headZ = skeleton.Joints[JointType.Head].Position.Z \* 1000;

shoulderCenterX = skeleton.Joints[JointType.ShoulderCenter].Position.X \* 1000;

shoulderCenterY = skeleton.Joints[JointType.ShoulderCenter].Position.Y \* 1000;

shoulderCenterZ = skeleton.Joints[JointType.ShoulderCenter].Position.Z \* 1000;

}

catch (Exception error)

{

}

}

//Update Displays, Calculate Biometrics if Skeleton found.

if(colorData != null && depthData != null)

{

kinectDepthBitmap.RotateFlip(RotateFlipType.RotateNoneFlipX);

kinectCompositeBitmap.RotateFlip(RotateFlipType.RotateNoneFlipX);

KinectDepth.Image = kinectDepthBitmap;

KinectComposite.Image = kinectCompositeBitmap;

if(CordMapper != null)

{

UpdateBiometrics();

}

}

stopwatch.Stop();

textBox10.Text = stopwatch.ElapsedMilliseconds.ToString();

}

// Sub-module Handles all Biometric calculations

private void UpdateBiometrics()

{

// Depth Point manually found using Kinect System Explorer

DepthImagePoint dpToW = new DepthImagePoint();

dpToW.Depth = 708;

dpToW.X = 9;

dpToW.Y = 184;

//Arbitrary Depth Point A for finding center of wheel

DepthImagePoint dpPtA = new DepthImagePoint(); // Point A

dpPtA.Depth = 936;

dpPtA.X = 194;

dpPtA.Y = 280;

//Arbitrary Depth Point B for finding center of wheel

DepthImagePoint dpPtB = new DepthImagePoint(); // Point B

dpPtB.Depth = 1073;

dpPtB.X = 283;

dpPtB.Y = 245;

// Convert the three points from Depth Reference to Reference

SkeletonPoint skToW = CordMapper.MapDepthPointToSkeletonPoint(

DepthImageFormat.Resolution640x480Fps30, dpToW);

SkeletonPoint skpa = CordMapper.MapDepthPointToSkeletonPoint(

DepthImageFormat.Resolution640x480Fps30, dpPtA);

SkeletonPoint skpb = CordMapper.MapDepthPointToSkeletonPoint(

DepthImageFormat.Resolution640x480Fps30, dpPtB);

// Skeleton Point for Center of Wheel

SkeletonPoint skCoW = new SkeletonPoint();

skCoW.X = (float)(skpa.X+skpa.X-skpb.X);

skCoW.Y = (float)(skpa.Y+skpa.Y-skpb.Y);

skCoW.Z = (float)(skpa.Z+skpa.Z-skpb.Z);

CalculateDeploymentVectors(skCoW);

if (driver == true)

{

FindUserHeight();

DetectHandCollision(dpPtA, dpPtB);

//Distance from Top of Wheel

textBox4.Text = ((Math.Sqrt(Math.Pow((headX - (skToW.X \* 1000)), 2) +

Math.Pow((headY - (skToW.Y \* 1000)), 2) +

Math.Pow((headZ - (skToW.Z \* 1000)), 2)) / 25.4)\*1.0216 -

5.2369).ToString("F");

//Distance from center of wheel

textBox5.Text = ((Math.Sqrt(Math.Pow((headX - (skCoW.X \* 1000)), 2) +

Math.Pow((headY - (skCoW.Y \* 1000)), 2) +

Math.Pow((headZ - (skCoW.Z \* 1000)), 2)) / 25.4)\*1.014 -

4.4682).ToString("F");

}

}

private void DetectHandCollision(DepthImagePoint dpPtA, DepthImagePoint dpPtB)

{

SkeletonPoint skLHand;

SkeletonPoint skRHand;

if (skeleton == null)

{

return;

}

jointCollection = skeleton.Joints;

skLHand = jointCollection[JointType.HandLeft].Position;

skRHand = jointCollection[JointType.HandRight].Position;

DrawHandCollision(dpPtA, dpPtB, skLHand);

DrawHandCollision(dpPtA, dpPtB, skRHand);

}

private void DrawHandCollision(DepthImagePoint dpPtA, DepthImagePoint dpPtB, SkeletonPoint skHandPt)

{

// Center of the chest

SkeletonPoint skptA = CordMapper.MapDepthPointToSkeletonPoint(

DepthImageFormat.Resolution640x480Fps30, dpPtA);

// Center of steering wheel

SkeletonPoint skptB = CordMapper.MapDepthPointToSkeletonPoint(DepthImageFormat

DepthImageFormat.Resolution640x480Fps30, dpPtB);

// Line of Deployment Vector Points

float x1 = skptA.X;

float y1 = skptA.Y;

float z1 = skptA.Z;

float x0 = skptB.X;

float y0 = skptB.Y;

float z0 = skptB.Z;

// Hand Points

float x3;

float y3;

float z3;

x3 = skHandPt.X - x0;

y3 = skHandPt.Y - y0;

z3 = skHandPt.Z - z0;

// Invert the X skeleton point value so it displays correctly (not sure why)

skHandPt.X = skHandPt.X \* -1;

Graphics g = Graphics.FromImage(kinectCompositeBitmap);

Vector3 u = new Vector3(x1 - x0, y1 - y0, z1 - z0);

Vector3 pq = new Vector3(x3, y3, z3);

float distance = Vector3.Cross(pq, u).Length() / u.Length();

//textBox4.Text = distance.ToString();

if (distance < 0.203) // 203mm or 8 inches

{

DrawSkeletonDot(g, bluBrush, skHandPt);

}

}

private void FindUserHeight()

{

//Bottom of Seat Depth Point

DepthImagePoint dpSeat = new DepthImagePoint();

dpSeat.Depth = 1136;

dpSeat.X = 306;

dpSeat.Y = 478;

SkeletonPoint skSeat = CordMapper.MapDepthPointToSkeletonPoint(DepthImageFormat.Resolution640x480Fps30, dpSeat);

double HeadToNeck = Math.Sqrt(Math.Pow((headX - shoulderCenterX), 2) +

Math.Pow((headY - shoulderCenterY), 2) +

Math.Pow((headZ - shoulderCenterZ), 2)) / 25.4;

double NeckToSeat = Math.Sqrt(Math.Pow((skSeat.X \* 1000) - shoulderCenterX, 2) +

Math.Pow(((skSeat.Y \* 1000) - shoulderCenterY), 2) +

Math.Pow(((skSeat.Z \* 1000) - shoulderCenterZ), 2)) / 25.4;

textBox1.Text = (HeadToNeck + NeckToSeat).ToString("F");

textBox2.Text = ((HeadToNeck + NeckToSeat) / 0.43).ToString("F");

textBox6.Text = Math.Floor(((HeadToNeck + NeckToSeat) / 0.43) / 12).ToString();

textBox7.Text = (((HeadToNeck + NeckToSeat) / 0.43) % 12).ToString("F1");

}

private void CalculateDeploymentVectors(SkeletonPoint CenterOfWheel)

{

BestX = 0;

BestY = 0;

float Y = 0;

short depth;

float LOFDepth = 0;

DepthImagePoint dpCenterOfWheel = CordMapper.MapSkeletonPointToDepthPoint(CenterOfWheel,

DepthImageFormat.Resolution640x480Fps30);

// \*\*\*\*\*\*\*\*\*\*\*\*\*\* Torso Collision Detection \*\*\*\*\*\*\*\*\*\*\*\*/

// Search for collision point between Line of Deployment

// and person's torso from color camera X-coord 128-351

for (int X = 128; X < 351; X++)

{

Y = (float)((-0.39167 \* X) + 356);

depth = depthPixelData[(int)((Math.Round(Y) \* 640) + X)];

// F(X) = Y -- where Y is LOFDepth

// F(X) = 0.6928X + 736.3

LOFDepth = (float)((1.14167 \* X) + 714.5);

if (Math.Round((double)(depth)) == Math.Round((double)(LOFDepth)))

{

BestX = X;

BestY = (int)(Y);

driver = true;

break;

}

else if ((Math.Round((double)(depth)) - Math.Round((double)(LOFDepth)) <= 10) &&

(Math.Round((double)(depth)) - Math.Round((double)(LOFDepth)) >= -10))

{

BestX = X;

BestY = (int)(Y);

driver = true;

}

// No collision found

else if (X == 350 && BestX == 0)

{

BestX = 0;

BestY = 0;

driver = false;

break;

}

}

// If collision found draw deployment vector

if (BestX != 0)

{

DepthImagePoint dpLOF = new DepthImagePoint();

dpLOF.Depth = (int)(LOFDepth);

dpLOF.X = 640 - BestX;

dpLOF.Y = BestY;

CenterOfWheel.X = 0 - CenterOfWheel.X;

SkeletonPoint skLOF = CordMapper.MapDepthPointToSkeletonPoint(DepthImageFormat.Resolution640x480Fps30, dpLOF);

Graphics g = Graphics.FromImage(kinectCompositeBitmap);

if (skLOF.X != 0)

{

DrawSkeletonLine(g, greenPen, skLOF, CenterOfWheel);

}

LOFDistanceVal.Text = (Math.Sqrt(Math.Pow(((CenterOfWheel.X \* 1000) - (skLOF.X \* 1000)), 2) +

Math.Pow(((CenterOfWheel.Y \* 1000) - (skLOF.Y \* 1000)), 2) +

Math.Pow(((CenterOfWheel.Z \* 1000) - (skLOF.Z \* 1000)), 2)) / 25.4).ToString("F");

//Find Head Proximity

LOFHeadProxVal.Text = (Math.Sqrt(Math.Pow((headX - (skLOF.X \* 1000)), 2) +

Math.Pow((headY - (skLOF.Y \* 1000)), 2) +

Math.Pow((headZ - (skLOF.Z \* 1000)), 2)) / 25.4).ToString("F");

}

}

private void Form\_FormClosing(object sender, FormClosingEventArgs e)

{

//Stop the sensor

StopKinect(sensor);

}

private void Kinect()

{

//Set up the Kinect Sensor and Variables

if (KinectSensor.KinectSensors.Count == 0)

{

return;

}

else

{

//Create sensor, enable streams, set resolutions, setup skeletal

sensor = KinectSensor.KinectSensors[0];

sensor.ColorStream.Enable(ColorImageFormat.RgbResolution640x480Fps30);

sensor.DepthStream.Range = DepthRange.Near;

sensor.DepthStream.Enable(DepthImageFormat.Resolution640x480Fps30);

sensor.SkeletonStream.EnableTrackingInNearRange = true;

sensor.SkeletonStream.TrackingMode = SkeletonTrackingMode.Seated;

sensor.SkeletonStream.Enable(

new TransformSmoothParameters() {

Correction = 0.5f,

JitterRadius = 0.05f,

MaxDeviationRadius = 0.05f,

Prediction = 0.5f,

Smoothing = 0.5f

});

//Create New Event for Kinect data

sensor.AllFramesReady += new EventHandler<AllFramesReadyEventArgs>(SensorAllFramesReady);

//Initialize data structures

KinectLock.WaitOne();

skeletonDataTemp = new Skeleton[6];

skeletonData = new Skeleton[6];

colorDataTemp = new byte[sensor.ColorStream.FramePixelDataLength];

colorData = new byte[sensor.ColorStream.FramePixelDataLength];

depthData = new DepthImagePixel[sensor.DepthStream.FramePixelDataLength];

depthDataTemp = new DepthImagePixel[sensor.DepthStream.FramePixelDataLength];

depthColorData = new byte[sensor.ColorStream.FramePixelDataLength];

KinectLock.ReleaseMutex();

//Start the Kinect sensor

sensor.Start();

faceTracker = new FaceTracker(sensor);

CordMapper = new CoordinateMapper(sensor);

}

}

private void StopKinect(KinectSensor sensor)

{

//Stop the Kinect sensor

if (sensor != null)

{

sensor.Dispose();

sensor.Stop();

}

}

private void SensorAllFramesReady(object sender, AllFramesReadyEventArgs e)

{

//Lock Mutex before data exchange

KinectLock.WaitOne();

//Get color frame

using (ColorImageFrame colorImageFrame = e.OpenColorImageFrame())

{

if (colorImageFrame == null)

{

return;

}

colorImageFrame.CopyPixelDataTo(colorDataTemp);

}

//Get depth frame

using (DepthImageFrame depthImageFrame = e.OpenDepthImageFrame())

{

if (depthImageFrame == null)

{

return;

}

depthImageFrame.CopyDepthImagePixelDataTo(depthDataTemp);

}

//Get skeletal frame

using (SkeletonFrame skeletonFrame = e.OpenSkeletonFrame())

{

if (skeletonFrame == null)

{

return;

}

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

{

skeletonDataTemp = new Skeleton[skeletonFrame.SkeletonArrayLength];

}

skeletonFrame.CopySkeletonDataTo(skeletonDataTemp);

}

//Process Skeletal for facial tracking data

if (colorDataTemp != null && depthDataTemp != null && skeleton != null)

{

short[] temp = new short[depthDataTemp.Length];

for(int i=0;i<depthDataTemp.Length;i++)

{

temp[i] = (short)(depthDataTemp[i].Depth);

}

FaceTrackFrame faceFrame = faceTracker.Track(

ColorImageFormat.RgbResolution640x480Fps30,

colorDataTemp,

DepthImageFormat.Resolution640x480Fps30,

temp,

skeleton);

if (faceFrame.TrackSuccessful)

{

facePoints = faceFrame.GetProjected3DShape();

faceTriangles = faceFrame.GetTriangles();

headRoll = faceFrame.Rotation.Z;

headPitch = faceFrame.Rotation.X;

headYaw = faceFrame.Rotation.Y;

}

else

{

facePoints = null;

faceTriangles = null;

}

}

//Release mutex

KinectLock.ReleaseMutex();

}

private static Skeleton[] Skeleton\_Clone(Skeleton[] skOrigin)

{

//Copy skeletal data

MemoryStream ms = new MemoryStream();

BinaryFormatter bf = new BinaryFormatter();

bf.Serialize(ms, skOrigin);

ms.Position = 0;

object obj = bf.Deserialize(ms);

ms.Close();

return obj as Skeleton[];

}

private static Bitmap PixelDataToBitmap(byte[] inputData, IntPtr Ptr)

{

//Convert Color byte array into bitmap

Marshal.Copy(inputData, 0 , Ptr, inputData.Length);

return new Bitmap(640, 480, 2560, System.Drawing.Imaging.PixelFormat.Format32bppRgb,Ptr);

}

private static Bitmap DepthDataFormatter(DepthImagePixel[] inputData)

{

//Format depth short array into bitmap

int colorPixelIndex = 0;

depthPixelData = new short[inputData.Length];

for (int i=0; i < inputData.Length;++i)

{

short depth = inputData[i].Depth;

ColorRGB c = HSL2RGB(((double)(depth-200)/8192),0.66,0.5);

if (depth == 0)

{

depthColorData[colorPixelIndex++] = (byte)(0); //Blue

depthColorData[colorPixelIndex++] = (byte)(0); //Green

depthColorData[colorPixelIndex++] = (byte)(0); //Red

depthColorData[colorPixelIndex++] = (byte)(255);

}

else

{

depthColorData[colorPixelIndex++] = (byte)(c.B); //Blue

depthColorData[colorPixelIndex++] = (byte)(c.G); //Green

depthColorData[colorPixelIndex++] = (byte)(c.R); //Red

depthColorData[colorPixelIndex++] = (byte)(255);

}

try

{

depthPixelData[i]=depth;

}

catch(Exception error)

{

}

}

return PixelDataToBitmap(depthColorData, depthPtr);

}

private static void SkeletonToComposite()

{

// Draw Skeleton to Bitmap

jointCollection = skeleton.Joints;

Graphics g = Graphics.FromImage(kinectCompositeBitmap);

// DrawDots

DrawSkeletonDot(g, redBrush, jointCollection[JointType.Head].Position);

DrawSkeletonDot(g, redBrush, jointCollection[JointType.ShoulderCenter].Position);

DrawSkeletonDot(g, redBrush, jointCollection[JointType.ShoulderLeft].Position);

DrawSkeletonDot(g, redBrush, jointCollection[JointType.ShoulderRight].Position);

DrawSkeletonDot(g, redBrush, jointCollection[JointType.ElbowLeft].Position);

DrawSkeletonDot(g, redBrush, jointCollection[JointType.ElbowRight].Position);

DrawSkeletonDot(g, redBrush, jointCollection[JointType.WristLeft].Position);

DrawSkeletonDot(g, redBrush, jointCollection[JointType.WristRight].Position);

DrawSkeletonDot(g, redBrush, jointCollection[JointType.HandLeft].Position);

DrawSkeletonDot(g, redBrush, jointCollection[JointType.HandRight].Position);

// DrawLines

DrawSkeletonLine(g, redPen, jointCollection[JointType.Head].Position,

jointCollection[JointType.ShoulderCenter].Position);

DrawSkeletonLine(g, redPen, jointCollection[JointType.ShoulderCenter].Position,

jointCollection[JointType.ShoulderLeft].Position);

DrawSkeletonLine(g, redPen, jointCollection[JointType.ElbowLeft].Position,

jointCollection[JointType.ShoulderLeft].Position);

DrawSkeletonLine(g, redPen, jointCollection[JointType.ElbowLeft].Position,

jointCollection[JointType.WristLeft].Position);

DrawSkeletonLine(g, redPen, jointCollection[JointType.HandLeft].Position,

jointCollection[JointType.WristLeft].Position);

DrawSkeletonLine(g, redPen, jointCollection[JointType.ShoulderCenter].Position,

jointCollection[JointType.ShoulderRight].Position);

DrawSkeletonLine(g, redPen, jointCollection[JointType.ElbowRight].Position,

jointCollection[JointType.ShoulderRight].Position);

DrawSkeletonLine(g, redPen, jointCollection[JointType.ElbowRight].Position,

jointCollection[JointType.WristRight].Position);

DrawSkeletonLine(g, redPen, jointCollection[JointType.HandRight].Position,

jointCollection[JointType.WristRight].Position);

}

private static void DrawSkeletonDot(Graphics g, System.Drawing.SolidBrush brush, SkeletonPoint skPt)

{

//Convert skeleton point into X,Y and draw dot

DepthImagePoint dpPt = CordMapper.MapSkeletonPointToDepthPoint(skPt,DepthImageFormat.Resolution640x480Fps30);

g.FillEllipse(brush,dpPt.X-5,dpPt.Y-5,10,10);

}

private static void DrawSkeletonLine(Graphics g, System.Drawing.Pen pen, SkeletonPoint skPt1, SkeletonPoint skPt2)

{

//Draw line between two skeleton points

DepthImagePoint dpPt1 = CordMapper.MapSkeletonPointToDepthPoint(skPt1,DepthImageFormat.Resolution640x480Fps30);

DepthImagePoint dpPt2 = CordMapper.MapSkeletonPointToDepthPoint(skPt2,DepthImageFormat.Resolution640x480Fps30);

g.DrawLine(pen,dpPt1.X,dpPt1.Y,dpPt2.X,dpPt2.Y);

}

private static void FaceMaskToComposite()

{

//Draw face mask points onto bitmap and build triangles on bitmap

Graphics g = Graphics.FromImage(kinectCompositeBitmap);

foreach(FaceTriangle triangle in faceTriangles)

{

try

{

DrawFaceTriangle(g,bluPen,

facePoints[(FeaturePoint)(triangle.First)],

facePoints[(FeaturePoint)(triangle.Second)],

facePoints[(FeaturePoint)(triangle.Third)]);

}

catch (Exception error)

{

continue;

}

}

}

private static void DrawFaceTriangle(Graphics g, System.Drawing.Pen pen,

Microsoft.Kinect.Toolkit.FaceTracking.PointF FacePt1,

Microsoft.Kinect.Toolkit.FaceTracking.PointF FacePt2,

Microsoft.Kinect.Toolkit.FaceTracking.PointF FacePt3)

{

//Draw triangle using face points

g.DrawLine(pen,FacePt1.X,FacePt1.Y,FacePt2.X,FacePt2.Y);

g.DrawLine(pen,FacePt1.X,FacePt1.Y,FacePt3.X,FacePt3.Y);

g.DrawLine(pen,FacePt2.X,FacePt2.Y,FacePt3.X,FacePt3.Y);

}

private static void GetSkeleton()

{

//Find useable skelton

if (skeletonData != null)

{

skeleton = null;

foreach(Skeleton skel in skeletonData)

{

if(skel != null)

{

if(skel.TrackingState == SkeletonTrackingState.Tracked)

{

skeleton = skel;

}

}

}

}

}

public struct ColorRGB

{

public byte R;

public byte G;

public byte B;

public ColorRGB(System.Drawing.Color value)

{

this.R = value.R;

this.G = value.G;

this.B = value.B;

}

public static implicit operator System.Drawing.Color(ColorRGB rgb)

{

System.Drawing.Color c = System.Drawing.Color.FromArgb(rgb.R,rgb.G,rgb.B);

return c;

}

public static explicit operator ColorRGB(System.Drawing.Color c)

{

return new ColorRGB(c);

}

}

public static ColorRGB HSL2RGB(double h, double sl, double l)

{

// Given H,S,L in range of 0-1

// Returns a Color (RGB struct) in range of 0-255

double v;

double r,g,b;

r = l; // default to gray

g = l;

b = l;

v = (l <= 0.5) ? (l \* (1.0 + sl)) : (l + sl - l \* sl);

if (v > 0)

{

double m;

double sv;

int sextant;

double fract, vsf, mid1, mid2;

m = l + l - v;

sv = (v - m ) / v;

h \*= 6.0;

sextant = (int)h;

fract = h - sextant;

vsf = v \* sv \* fract;

mid1 = m + vsf;

mid2 = v - vsf;

switch (sextant)

{

case 0:

r = v;

g = mid1;

b = m;

break;

case 1:

r = mid2;

g = v;

b = m;

break;

case 2:

r = m;

g = v;

b = mid1;

break;

case 3:

r = m;

g = mid2;

b = v;

break;

case 4:

r = mid1;

g = m;

b = v;

break;

case 5:

r = v;

g = m;

b = mid2;

break;

}

}

ColorRGB rgb;

rgb.R = Convert.ToByte(r \* 255.0f);

rgb.G = Convert.ToByte(g \* 255.0f);

rgb.B = Convert.ToByte(b \* 255.0f);

return rgb;

}

private static Mutex KinectLock = new Mutex();

private static KinectSensor sensor;

private static FaceTracker faceTracker = null;

private static EnumIndexableCollection<FeaturePoint, Microsoft.Kinect.Toolkit.FaceTracking.PointF> facePoints = null;

private static Microsoft.Kinect.Toolkit.FaceTracking.FaceTriangle[] faceTriangles = null;

private static FaceTrackFrame faceFrame = null;

private static Skeleton skeleton = null;

private static Skeleton[] skeletonData = null;

private static Skeleton[] skeletonDataTemp = null;

private static DepthImagePixel[] depthDataTemp = null;

private static DepthImagePixel[] depthData = null;

private static byte[] colorDataTemp = null;

private static byte[] colorData = null;

private static byte[] depthColorData = null;

private static short[] depthPixelData = null;

private static JointCollection jointCollection = null;

private static CoordinateMapper CordMapper = null;

// Bitmaps

private static Bitmap kinectCompositeBitmap = null;

private static Bitmap kinectDepthBitmap = null;

// Pointers

private static IntPtr colorPtr;

private static IntPtr depthPtr;

private static IntPtr compositePtr;

//Brushes

private static System.Drawing.SolidBrush redBrush = new System.Drawing.SolidBrush(System.Drawing.Color.Red);

private static System.Drawing.SolidBrush greenBrush = new System.Drawing.SolidBrush(System.Drawing.Color.Green);

private static System.Drawing.SolidBrush bluBrush = new System.Drawing.SolidBrush(System.Drawing.Color.Blue);

private static System.Drawing.Pen redPen = new System.Drawing.Pen(System.Drawing.Color.Red, 3);

private static System.Drawing.Pen bluPen = new System.Drawing.Pen(System.Drawing.Color.Blue, 1);

private static System.Drawing.Pen greenPen = new System.Drawing.Pen(System.Drawing.Color.Green, 4);

// Joint points and orientations

private static float headX = 0;

private static float headY = 0;

private static float headZ = 0;

private static float shoulderCenterX = 0;

private static float shoulderCenterY = 0;

private static float shoulderCenterZ = 0;

private static float headRoll = 0;

private static float headYaw = 0;

private static float headPitch = 0;

private static int BestX = 0;

private static int BestY = 0;

private static bool driver = true;

//Timer event

System.Windows.Forms.Timer RefreshTimer = new System.Windows.Forms.Timer();

}

}