```
namespace kinectScan
{
    using System;
    using System.ComponentModel;
    using System.Globalization;
    using System.IO;
    using System.Threading.Tasks;
    using System.Drawing;
    using System.Diagnostics;
    using System.Windows;
    using System.Windows.Controls;
    using System.Windows.Media;
    using System.Windows.Media.Imaging;
    using System.Windows.Media.Media3D;
    using System.Windows.Threading;

    using HelixToolkit.Wpf;

    using Microsoft.Kinect;
    using Microsoft.Kinect.Toolkit;

    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {

        /// <summary>
        /// Timestamp of last depth frame in milliseconds
        /// </summary>
        private long lastFrameTimestamp = 0;

        /// <summary>
        /// Timer to count FPS
        /// </summary>
        private DispatcherTimer fpsTimer;

        /// <summary>
        /// Timer stamp of last computation of FPS
        /// </summary>
        private DateTime lastFPSTimestamp;

        /// <summary>
        /// Event interval for FPS timer
        /// </summary>
        private const int FpsInterval = 5;

        /// <summary>
        /// The counter for frames that have been processed
        /// </summary>
        private int processedFrameCount = 0;

        /// <summary>
        /// Active Kinect sensor
        /// </summary>
        private KinectSensor sensor;

        /// <summary>
        /// Kinect sensor chooser object
        /// </summary>
        private KinectSensorChooser sensorChooser;

        /// <summary>
        /// Format of depth image to use
        /// </summary>
        private const DepthImageFormat dFormat = DepthImageFormat.Resolution320x240Fps30;

        /// <summary>
```

```
69          /// Format of color image to use
70          /// </summary>
71          private const ColorImageFormat cFormat = ColorImageFormat.InfraredResolution640x480Fps30;
72
73          // stores furthest depth in the scene
74          public ushort greatestDepth = 0;
75
76          // array for all of the depth data
77          private int[] Depth = new int[320 * 240];
78
79          // stores all of the 3D trianlges with normals and points
80          Model3DGroup modelGroup = new Model3DGroup();
81
82          // material placed over the mesh for viewing
83          public GeometryModel3D msheet = new GeometryModel3D();
84
85          // collection of corners for the triangles
86          public Point3DCollection corners = new Point3DCollection();
87
88          // collection of all the triangles
89          public Int32Collection Triangles = new Int32Collection();
90
91
92          public MeshGeometry3D tmesh = new MeshGeometry3D();
93
94          // collection of all the cross product normals
95          public Vector3DCollection Normals = new Vector3DCollection();
96
97          // add texture to the mesh
98          public PointCollection myTextureCoordinatesCollection = new PointCollection();
99
100         // storage for camera, scene, etc...
101         public ModelVisual3D modelsVisual = new ModelVisual3D();
102
103
104         public Viewport3D myViewport = new Viewport3D();
105
106         // test variable
107         public int samplespot;
108
109         // variable for changing the quality 1 is the best 16 contains almost no data
110         public int s = 1;
111
112         // depth point collection
113         public int[] depths_array = new int[4];
114
115         // collection of points
116         Point3D[] points_array = new Point3D[4];
117
118         // collection of vectors
119         Vector3D[] vectors_array = new Vector3D[5];
120
121         //used for displaying RGB camera
122         public byte[] colorPixels;
123         public WriteableBitmap colorBitmap;
124
125         public MainWindow()
126         {
127             InitializeComponent();
128         }
129
130         private void WindowLoaded(object sender, RoutedEventArgs e)
131         {
132             // Start Kinect sensor chooser
133             this.sensorChooser = new KinectSensorChooser();
134             this.sensorChooserUI.KinectSensorChooser = this.sensorChooser;
135             this.sensorChooser.KinectChanged += this.OnKinectSensorChanged;
136             this.sensorChooser.Start();
```

```
137
138             // Start fps timer
139             this.fpsTimer = new DispatcherTimer(DispatcherPriority.Send);
140             this.fpsTimer.Interval = new TimeSpan(0, 0, FpsInterval);
141             this.fpsTimer.Tick += this.FpsTimerTick;
142             this.fpsTimer.Start();
143
144             // Set last fps timestamp as now
145             this.lastFPSTimestamp = DateTime.Now;
146         }
147
148         /// <summary>
149         /// Execute shutdown tasks
150         /// </summary>
151         /// <param name="sender">object sending the event</param>
152         /// <param name="e">event arguments</param>
153         private void WindowClosing(object sender, System.ComponentModel.CancelEventArgs e)
154         {
155             // Stop timer
156             if (null != this.fpsTimer)
157             {
158                 this.fpsTimer.Stop();
159                 this.fpsTimer.Tick -= this.FpsTimerTick;
160             }
161
162             // Unregister Kinect sensor chooser event
163             if (null != this.sensorChooser)
164             {
165                 this.sensorChooser.KinectChanged -= this.OnKinectSensorChanged;
166             }
167
168             // Stop sensor
169             if (null != this.sensor)
170             {
171                 this.sensor.Stop();
172                 this.sensor.DepthFrameReady -= this.SensorDepthFrameReady;
173                 this.sensor.ColorFrameReady -= this.SensorColorFrameReady;
174             }
175
176             // Empty the canvas
177             this.ClearMesh();
178         }
179
180         /// <summary>
181         /// Handles adding a new kinect
182         /// </summary>
183         /// <param name="sender">object sending the event</param>
184         /// <param name="e">event arguments for the newly connected Kinect</param>
185         private void OnKinectSensorChanged(object sender, KinectChangedEventArgs e)
186         {
187             // Check new sensor's status
188             if (this.sensor != e.NewSensor)
189             {
190                 // Stop old sensor
191                 if (null != this.sensor)
192                 {
193                     this.sensor.Stop();
194                     this.sensor.DepthFrameReady -= this.SensorDepthFrameReady;
195                     this.sensor.ColorFrameReady -= this.SensorColorFrameReady;
196                 }
197
198                 this.sensor = null;
199
200                 if (null != e.NewSensor && KinectStatus.Connected == e.NewSensor.Status)
201                 {
202                     // Start new sensor
203                     this.sensor = e.NewSensor;
204                     this.StartCameraStream(dFormat, cFormat);
```

```
205                 }
206             }
207 
208             if (null == this.sensor)
209             {
210                 // if no kinect clear the text on screen
211                 this.statusBarText.Content = Properties.Resources.NoKinectReady;
212                 this.IR_Title.Content = "";
213                 this.Model_Title.Content = "";
214                 this.RGB_Title.Content = "";
215             }
216         }
217 
218         /// <summary>
219         /// Handler for FPS timer tick
220         /// </summary>
221         /// <param name="sender">Object sending the event</param>
222         /// <param name="e">Event arguments</param>
223         private void FpsTimerTick(object sender, EventArgs e)
224         {
225 
226             if (null == this.sensor)
227             {
228                 // Show "No ready Kinect found!" on status bar
229                 this.KinectStatusText.Content = Properties.Resources.NoReadyKinect;
230             }
231             else
232             {
233                 // Calculate time span from last calculation of FPS
234                 double intervalSeconds = (DateTime.Now - this.lastFPSTimestamp).TotalSeconds;
235 
236                 // Calculate and show fps on status bar
237                 this.KinectStatusText.Content = string.Format(
238                     System.Globalization.CultureInfo.InvariantCulture,
239                     Properties.Resources.Fps,
240                     (double)this.processedFrameCount / intervalSeconds);
241             }
242 
243             // Reset frame counter
244             this.processedFrameCount = 0;
245             this.lastFPSTimestamp = DateTime.Now;
246         }
247 
248         /// <summary>
249         /// Reset FPS timer and counter
250         /// </summary>
251         private void ResetFps()
252         {
253             // Restart fps timer
254             if (null != this.fpsTimer)
255             {
256                 this.fpsTimer.Stop();
257                 this.fpsTimer.Start();
258             }
259 
260             // Reset frame counter
261             this.processedFrameCount = 0;
262             this.lastFPSTimestamp = DateTime.Now;
263         }
264 
265         /// <summary>
266         /// Start depth stream at specific resolution
267         /// </summary>
268         /// <param name="format">The resolution of image in depth stream</param>
269         private void StartCameraStream(DepthImageFormat dFormat, ColorImageFormat cFormat)
270         {
271             try
272             {
```

```
273                 // Enable streams, register event handler and start
274                 this.sensor.DepthStream.Enable(dFormat);
275                 this.sensor.DepthFrameReady += this.SensorDepthFrameReady;
276                 this.sensor.ColorStream.Enable(cFormat);
277                 this.sensor.ColorFrameReady += this.SensorColorFrameReady;
278                 this.sensor.Start();
279             }
280             catch (IOException ex)
281             {
282                 // Device is in use
283                 this.sensor = null;
284                 this.ShowStatusMessage(ex.Message);
285 
286                 return;
287             }
288             catch (InvalidOperationException ex)
289             {
290                 // Device is not valid, not supported or hardware feature unavailable
291                 this.sensor = null;
292                 this.ShowStatusMessage(ex.Message);
293 
294                 return;
295             }
296 
297             // Allocate space to put the pixels we'll receive
298             this.colorPixels = new byte[this.sensor.ColorStream.FramePixelDataLength];
299 
300             //// This is the bitmap we'll display on-screen
301             this.colorBitmap = new WriteableBitmap(this.sensor.ColorStream.FrameWidth, this.sensor.
    ColorStream.FrameHeight, 96.0, 96.0, PixelFormats.Gray16, null);
302         }
303 
304         /// <summary>
305         /// Event handler for Kinect sensor's ColorFrameReady event
306         /// </summary>
307         /// <param name="sender">object sending the event</param>
308         /// <param name="e">event arguments</param>
309         void SensorColorFrameReady(object sender, ColorImageFrameReadyEventArgs e)
310         {
311             using (ColorImageFrame colorFrame = e.OpenColorImageFrame())
312             {
313                 if (colorFrame != null)
314                 {
315                     // Copy the pixel data from the image to a temporary array
316                     colorFrame.CopyPixelDataTo(this.colorPixels);
317 
318                     // Write the pixel data into our bitmap
319                     this.colorBitmap.WritePixels(
320                         new Int32Rect(0, 0, this.colorBitmap.PixelWidth, this.colorBitmap.PixelHeight),
321                         this.colorPixels,
322                         this.colorBitmap.PixelWidth * colorFrame.BytesPerPixel,
323                         0);
324                 }
325 
326                 // set the RGB image to the RGB camera
327                 this.KinectRGBView.Source = this.colorBitmap;
328 
329             }
330         }
331 
332         /// <summary>
333         /// Event handler for Kinect sensor's DepthFrameReady event
334         /// Take in depth data
335         /// </summary>
336         /// <param name="sender">object sending the event</param>
337         /// <param name="e">event arguments</param>
338         void SensorDepthFrameReady(object sender, DepthImageFrameReadyEventArgs e)
339         {
```

```
340
341             DepthImageFrame imageFrame = e.OpenDepthImageFrame();
342             if (imageFrame != null)
343             {
344                 double maxDepth = Far_Filter_Slider.Value;
345                 short[] pixelData = new short[imageFrame.PixelDataLength];
346                 imageFrame.CopyPixelDataTo(pixelData);
347                 this.greatestDepth = 0;
348                 for (int y = 0; y < 240; y++)
349                 {
350                     for (int x = 0; x < 320; x++)
351                     {
352                         // scale depth down
353                         this.Depth[x + (y * 320)] = ((ushort)pixelData[x + y * 320]) / 100;
354 
355                         // finds the furthest depth from all the depth pixels
356                         if ((this.Depth[x + y * 320] > this.greatestDepth) && (this.Depth[x + y * 320] ↙
    < maxDepth))
357                         {
358                             this.greatestDepth = (ushort)this.Depth[x + y * 320];
359                         }
360 
361 
362                     }
363                 }
364                 // Blur Filter  -- Guassian
365                 if (Filter_Blur.IsChecked == true)
366                 {
367                     for (int i = 641; i < this.Depth.Length - 641; ++i)
368                     {
369 
370                         short depthaverage = (Int16)((this.Depth[i - 641] + (2 * this.Depth[i - 640]) + ↙
     this.Depth[i - 639] +
371                                                     (2 * this.Depth[i - 1]) + (4 * this.Depth[i]) + (2 ↙
     * this.Depth[i + 2]) +
372                                                     this.Depth[i + 639] + (2 * this.Depth[i + 640]) + ↙
    this.Depth[i + 641]) / 16);
373 
374                         this.Depth[i] = depthaverage;
375                         if ((this.Depth[i] > this.greatestDepth) && (this.Depth[i] < maxDepth))
376                         {
377                             this.greatestDepth = (ushort)this.Depth[i];
378                         }
379                     }
380                 }
381 
382                 // Set the depth image to the Depth sensor view
383                 this.KinectDepthView.Source = DepthToBitmapSource(imageFrame);
384             }
385         }
386 
387 
388         /// <summary>
389         /// Flag check for a point within the bounding box
390         /// </summary>
391         /// <param name="x">location on the x plane</param>
392         /// <param name="y">location on the y plane</param>
393         private bool PointinRange(int x, int y)
394         {
395             double minDepth = Near_Filter_Slider.Value;
396             double maxDepth = Far_Filter_Slider.Value;
397             return ((this.Depth[x + (y * 320)] >= minDepth && this.Depth[x + (y * 320)] <= maxDepth) ||
398                 (this.Depth[(x + s) + (y * 320)] >= minDepth && this.Depth[(x + s) + (y * 320)] <=     ↙
    maxDepth) ||
399                 (this.Depth[x + ((y + s) * 320)] >= minDepth && this.Depth[x + ((y + s) * 320)] <=     ↙
    maxDepth) ||
400                 (this.Depth[(x + s) + ((y + s) * 320)] >= minDepth && this.Depth[(x + s) + ((y + s) *   ↙
    320)] <= maxDepth));
```

```
401
402         }
403
404         /// <summary>
405         /// Create the mesh
406         /// </summary>
407         void BuildMesh()
408         {
409             double maxDepth = Far_Filter_Slider.Value;
410             int i = 0;
411             for (int y = (int)Top_Slider.Value; y < ((int)Bot_Slider.Value - s); y = y + s)
412             {
413                 for (int x = (int)Left_Slider.Value; x < ((int)Right_Slider.Value - s); x = x + s)
414                 {
415                     //Any point less than max
416                     if (PointinRange(x, y))
417                     {
418                         if (this.Depth[x + ((y + s) * 320)] >= maxDepth)
419                         {
420                             depths_array[0] = -this.greatestDepth;
421                         }
422                         else
423                         {
424                             depths_array[0] = -this.Depth[x + ((y + s) * 320)];
425                         }
426
427                         if (this.Depth[x + (y * 320)] >= maxDepth)
428                         {
429                             depths_array[1] = -this.greatestDepth;
430                         }
431                         else
432                         {
433                             depths_array[1] = -this.Depth[x + (y * 320)];
434                         }
435
436                         if (this.Depth[(x + s) + (y * 320)] >= maxDepth)
437                         {
438                             depths_array[2] = -this.greatestDepth;
439                         }
440                         else
441                         {
442                             depths_array[2] = -this.Depth[(x + s) + (y * 320)];
443                         }
444
445                         if (this.Depth[(x + s) + ((y + s) * 320)] >= maxDepth)
446                         {
447                             depths_array[3] = -this.greatestDepth;
448                         }
449                         else
450                         {
451                             depths_array[3] = -this.Depth[(x + s) + ((y + s) * 320)];
452                         }
453
454                         // triangle point locations
455                         points_array[0] = new Point3D(x, (y + s), depths_array[0]);
456                         points_array[1] = new Point3D(x, y, depths_array[1]);
457                         points_array[2] = new Point3D((x + s), y, depths_array[2]);
458                         points_array[3] = new Point3D((x + s), (y + s), depths_array[3]);
459
460                         // create vectors of size difference between points
461                         vectors_array[0] = new Vector3D(points_array[1].X - points_array[0].X,
    points_array[1].Y - points_array[0].Y, points_array[1].Z - points_array[0].Z);
462                         vectors_array[1] = new Vector3D(points_array[1].X - points_array[2].X,
    points_array[1].Y - points_array[2].Y, points_array[1].Z - points_array[2].Z);
463                         vectors_array[2] = new Vector3D(points_array[2].X - points_array[0].X,
    points_array[2].Y - points_array[0].Y, points_array[2].Z - points_array[0].Z);
464                         vectors_array[3] = new Vector3D(points_array[3].X - points_array[0].X,
    points_array[3].Y - points_array[0].Y, points_array[3].Z - points_array[0].Z);
```

```
                        vectors_array[4] = new Vector3D(points_array[2].X - points_array[3].X,
    points_array[2].Y - points_array[3].Y, points_array[2].Z - points_array[3].Z);

                        // add the corners to the 2 triangles to form a square
                        corners.Add(points_array[0]);
                        corners.Add(points_array[1]);
                        corners.Add(points_array[2]);
                        corners.Add(points_array[2]);
                        corners.Add(points_array[3]);
                        corners.Add(points_array[0]);

                        // add triangles to the collection
                        Triangles.Add(i);
                        Triangles.Add(i + 1);
                        Triangles.Add(i + 2);
                        Triangles.Add(i + 3);
                        Triangles.Add(i + 4);
                        Triangles.Add(i + 5);

                        // find the normals of the triangles by taking the cross product
                        Normals.Add(Vector3D.CrossProduct(vectors_array[0], vectors_array[2]));
                        Normals.Add(Vector3D.CrossProduct(vectors_array[0], vectors_array[1]));
                        Normals.Add(Vector3D.CrossProduct(vectors_array[1], vectors_array[2]));
                        Normals.Add(Vector3D.CrossProduct(vectors_array[1], vectors_array[2]));
                        Normals.Add(Vector3D.CrossProduct(vectors_array[3], vectors_array[4]));
                        Normals.Add(Vector3D.CrossProduct(vectors_array[0], vectors_array[2]));

                        i = i + 6;
                    }

                }
            }

            // add the flat back wall
            int numcorners = corners.Count;
            for (int p = 0; p < numcorners; p++)
            {
                Point3D cornertocopy = corners[p];
                corners.Add(new Point3D(cornertocopy.X, cornertocopy.Y, -this.greatestDepth));
                Triangles.Add(i);
                Normals.Add(new Vector3D(0, 0, 1));
                i = i + 1;
            }


        }

        /// <summary>
        /// Create depth image from depth frame
        /// </summary>
        /// <param name="imageFrame">collection of depth data</param>
        BitmapSource DepthToBitmapSource(DepthImageFrame imageFrame)
        {
            short[] pixelData = new short[imageFrame.PixelDataLength];
            imageFrame.CopyPixelDataTo(pixelData);
            BitmapSource bmap = BitmapSource.Create(
             imageFrame.Width,
             imageFrame.Height,
             96, 96,
             PixelFormats.Gray16,
             null,
             pixelData,
             imageFrame.Width * imageFrame.BytesPerPixel);
            return bmap;
        }

        /// <summary>
        /// take a photo when button is clicked
```

```
532         /// </summary>
533         /// <param name="sender">object sending the event</param>
534         /// <param name="e">event arguments</param>
535         private void Begin_Scan_Click(object sender, RoutedEventArgs e)
536         {
537             //clear the canvas
538             this.ClearMesh();
539 
540             // add light to the scene
541             DirectionalLight DirLight1 = new DirectionalLight();
542             DirLight1.Color = Colors.White;
543             DirLight1.Direction = new Vector3D(0, 0, -1);
544 
545             // add a camera to the scene
546             PerspectiveCamera Camera1 = new PerspectiveCamera();
547 
548             // set the location of the camera
549             Camera1.Position = new Point3D(160, 120, 480);
550             Camera1.LookDirection = new Vector3D(0, 0, -1);
551             Camera1.UpDirection = new Vector3D(0, -1, 0);
552 
553             // create the mesh from depth data
554             this.BuildMesh();
555 
556             // add texture to all the points
557             tmesh.Positions = corners;
558             tmesh.TriangleIndices = Triangles;
559             tmesh.Normals = Normals;
560             tmesh.TextureCoordinates = myTextureCoordinatesCollection;
561             msheet.Geometry = tmesh;
562             msheet.Material = new DiffuseMaterial((SolidColorBrush)(new BrushConverter().ConvertFrom("#↙
    52318F")));
563 
564             // build the scene and display it
565             this.modelGroup.Children.Add(msheet);
566             this.modelGroup.Children.Add(DirLight1);
567             this.modelsVisual.Content = this.modelGroup;
568             this.myViewport.IsHitTestVisible = false;
569             this.myViewport.Camera = Camera1;
570             this.myViewport.Children.Add(this.modelsVisual);
571             KinectNormalView.Children.Add(this.myViewport);
572             this.myViewport.Height = KinectNormalView.Height;
573             this.myViewport.Width = KinectNormalView.Width;
574             Canvas.SetTop(this.myViewport, 0);
575             Canvas.SetLeft(this.myViewport, 0);
576 
577         }
578 
579         /// <summary>
580         /// Export the completed mesh to a .obj file
581         /// </summary>
582         /// <param name="sender">object sending the event</param>
583         /// <param name="e">event arguments</param>
584         private void Export_Model_Click(object sender, RoutedEventArgs e)
585         {
586             //function from Helix Toolkit
587             string fileName = Model_Name.Text + ".obj";
588 
589             using (var exporter = new ObjExporter(fileName))
590             {
591                 exporter.Export(this.modelGroup);
592             }
593 
594             // test code for seeing depth frame values
595             Process.Start("explorer.exe", "/select,\"" + fileName + "\"");
596 
597             string fileName2 = "depth.txt";
598 
```

```
599             using (System.IO.StreamWriter file = new System.IO.StreamWriter(fileName2))
600             {
601                 //file.Write(string.Join(",", this.Depth));
602                 file.Write(greatestDepth);
603             }
604 
605         }
606 
607         /// <summary>
608         /// Show exception info on status bar
609         /// </summary>
610         /// <param name="message">Message to show on status bar</param>
611         private void ShowStatusMessage(string message)
612         {
613             this.Dispatcher.BeginInvoke((Action)(() =>
614             {
615                 this.ResetFps();
616                 this.KinectStatusText.Content = message;
617             }));
618         }
619 
620         /// <summary>
621         /// clear everything from the scene and canvas
622         /// </summary>
623         public void ClearMesh()
624         {
625             KinectNormalView.Children.Clear();
626             modelGroup.Children.Clear();
627             myViewport.Children.Clear();
628             modelsVisual.Children.Clear();
629             tmesh.Positions.Clear();
630             tmesh.TriangleIndices.Clear();
631             tmesh.Normals.Clear();
632             tmesh.TextureCoordinates.Clear();
633 
634         }
635 
636         /// <summary>
637         /// Clear canvas button click
638         /// </summary>
639         /// <param name="sender">object sending the event</param>
640         /// <param name="e">event arguments</param>
641         private void End_Scan_Click(object sender, RoutedEventArgs e)
642         {
643             this.ClearMesh();
644         }
645     }
646 }
```