Intel® RealSense™ Enhanced Photography

With the R200 Camera Sample Application

By Rick Blacker

Contents

[Introduction - 3 -](#_Toc432511176)

[Requirements - 3 -](#_Toc432511177)

[Project Structure - 3 -](#_Toc432511178)

[Visual Studio Project Structure - 4 -](#_Toc432511179)

[Simple high level sequence diagram - 5 -](#_Toc432511180)

[Code Walkthrough - 6 -](#_Toc432511181)

[High level overview of the source code files and form - 7 -](#_Toc432511182)

[Forms - 7 -](#_Toc432511183)

[Source Code - 14 -](#_Toc432511184)

[Source\CustomEventArgs Code - 14 -](#_Toc432511185)

[In depth Understanding - 14 -](#_Toc432511186)

[RSStreaming - 15 -](#_Toc432511187)

[RSEnhancedPhotography - 17 -](#_Toc432511188)

[RSUtility - 20 -](#_Toc432511189)

[RSPaintUtil - 22 -](#_Toc432511190)

[RSPoints - 23 -](#_Toc432511191)

[RSSampleArg - 23 -](#_Toc432511192)

[RSEnhancedImageArg - 24 -](#_Toc432511193)

[RSMeasureArg - 24 -](#_Toc432511194)

[FormMain - 25 -](#_Toc432511195)

[FormMeasure - 28 -](#_Toc432511196)

[FormDepthEnhance - 30 -](#_Toc432511197)

[FormDepthResize - 31 -](#_Toc432511198)

[FormDepthRefocus - 32 -](#_Toc432511199)

[FormDepthPasteOnPlane - 33 -](#_Toc432511200)

[FormDepthBlending - 34 -](#_Toc432511201)

# Introduction

In this document and sample application, I will show you how to use the Intel® R200 camera along with the Enhanced Photography functionality out of the RealSense™ SDK.

# Requirements

Hardware requirements:

* 4th generation Intel® Core™ processors based on the Intel microarchitecture code name Haswell
* 8GB free hard disk space
* Intel® RealSense™ R200 3D camera (required to connect to a USB\* 3 port)

Software requirements:

* Microsoft\* Windows\* 10 OS 64-bit
* Intel RealSense SDK which can be downloaded [here](https://software.intel.com/en-us/intel-realsense-sdk/download). R5
* Microsoft Visual Studio\* 2010-2015 with the latest service pack
* Microsoft .NET\* 4.0 Framework for C# development
* Unity\* 4.1.0 or higher for Unity game development

Note: This particular sample project was created in Visual Studio 2015 using the latest .NET release.

# Project Structure

In this sample application, I have tried to separate out the RealSense functionality from the GUI layer code to make it easier for a developer to focus on the R200 Enhanced Photography functionality. I’ve done this by creating a couple C# wrapper classes.

* [RSStreaming](#Overview_RSStreaming) – Wraps the RealSense streaming functionality
* [RSEnhancedPhotography](#Overview_RSEnhancedPhotography) – Wraps the RealSense enhanced photography

Also, to try and make things cleaner and simpler, I created individual WinForms for each type of Enhanced Photography function. While I understand this creates a little more duplicated code, it offers up the benefit of keeping things very precise in trying to demo each Enhanced Photography functionality.

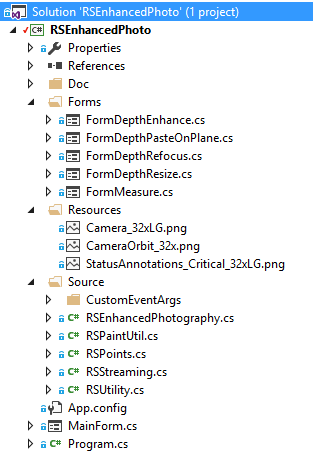
While there is more to the project structure than what I have mentioned here, I will go into full detail on all the source code files and how they work throughout this document.

You will notice that I make use of creating my own events. I do this to demonstrate how to keep the RealSense functionality loosely coupled from the GUI source code as much as possible. I find this to be a cleaner solution that passing an entire form into a class so that a given class can access properties on the form.

It’s also necessary to note that this sample application does not try to enforce proper software engineering techniques. There is very little if any runtime checking, no try catch blocks. The reason behind this is to keep the code as clean as possible without introducing extra distractions by providing a simple clean example project to learn from.

While none of the forms in this sample application are earth shattering beautiful, they do serve the purpose of demonstrating how to use Enhanced Photography.

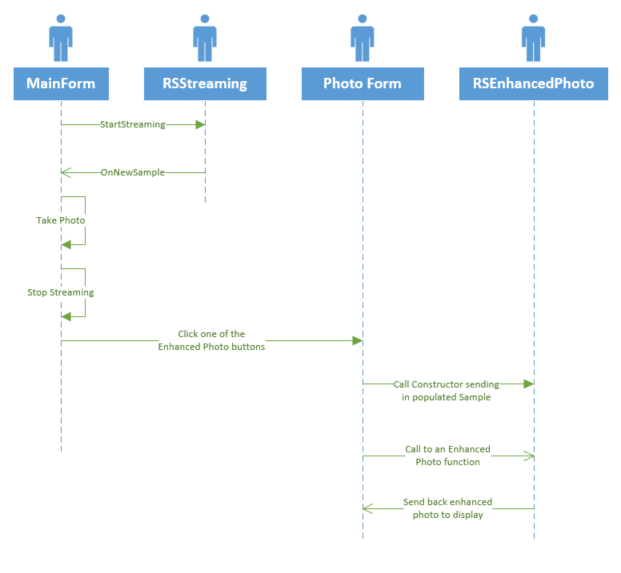
## Visual Studio Project Structure



This image shows what the Visual Studio 2015 solution looks like. The various folders contain:

* Forms – All the various WinForms that demonstrate a different Enhanced Photography functionality
* Source – All the source code that goes along with the project
* Source\CustomEventArgs – This folder contains classes that have been derived from the native EventArgs class.
* MainForm.CS – This is the main form of the application.

# Simple high level sequence diagram



When the user runs the sample application, FormMain will display. On this form the user will can start streaming by clicking on the Streaming button. When this button is clicked, the application will kick off the streaming by making use of functionality wrapped up in [RSStreaming](#Overview_RSStreaming) class.

The RSStreaming class is constantly causing updates to FormMain by calling its internal event OnNewStreamingSample. This happens every frame from the camera.

As the streaming is running, the user has the ability to stop streaming by clicking on the Stop Streaming button. If this happens, the streaming simply stops running and the user does not have any option to do anything else other than start streaming again. However, if the user clicks on the Photo button, the image data will be saved to disk and streaming stops. Once streaming has stopped, the photo saved to disk, the Photo Enhancement buttons will become active allowing for the various Enhanced Photography dialogs to be activated demonstrating each of the capabilities included in this sample application.

When one of the Enhanced Photo dialogs are loaded up, the sample image that was saved to disk is loaded up and utilized. I will go into deeper explanation later in this document.

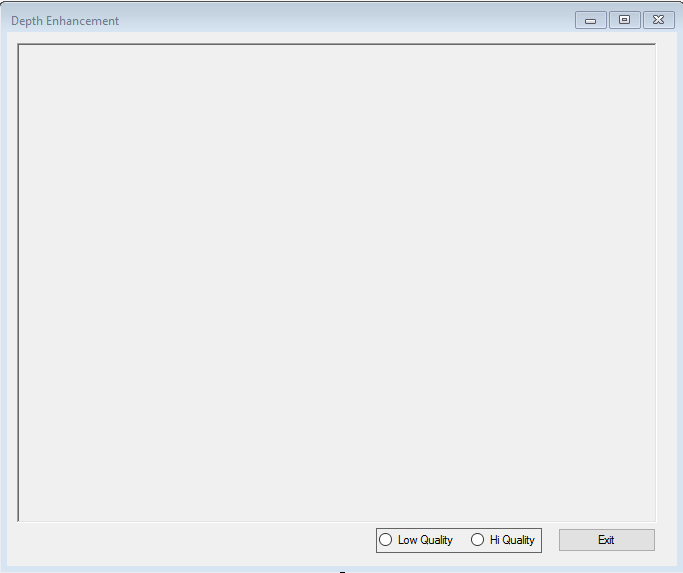
# Code Walkthrough

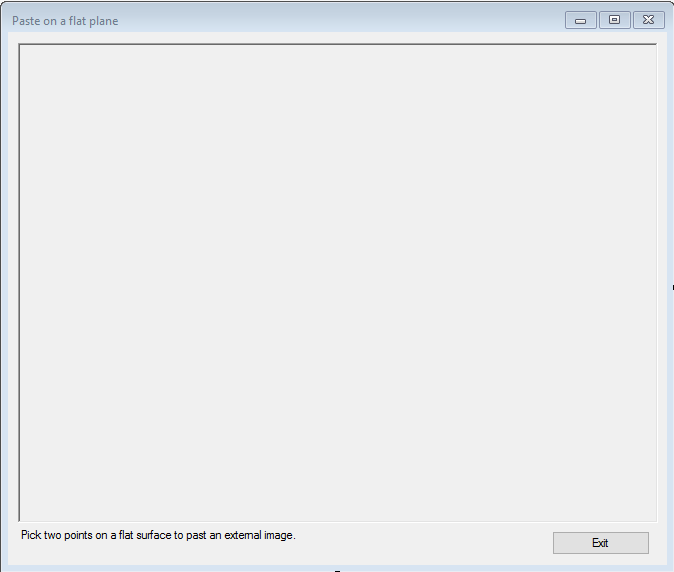
The following sections will walk you through how the entire application works detailing out the flow of the application and descriptions of the various classes.

## High level overview of the source code files and form

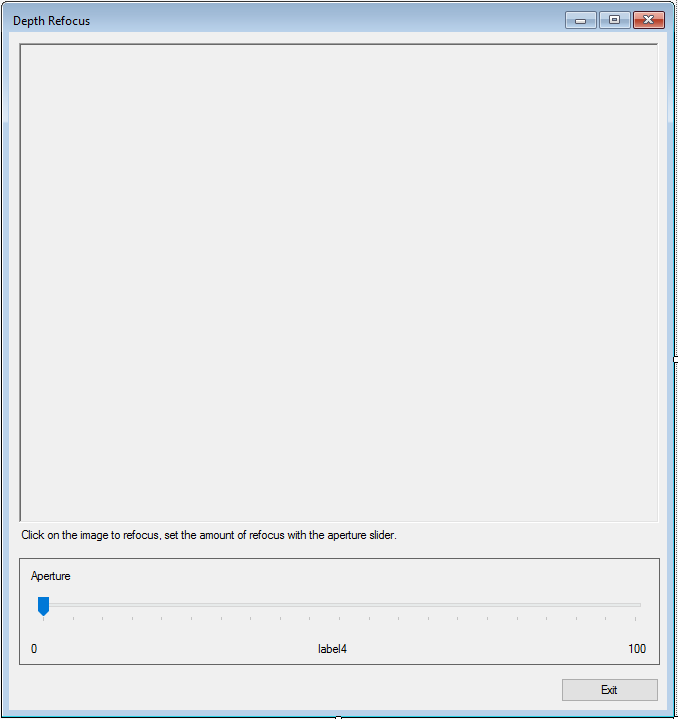
### Forms

[FormDepthEnhance](#Detail_Form_DepthEnhance) – Demonstrates how use two different depth quality settings to display depth data. The user can either choose Low Quality or High Quality.

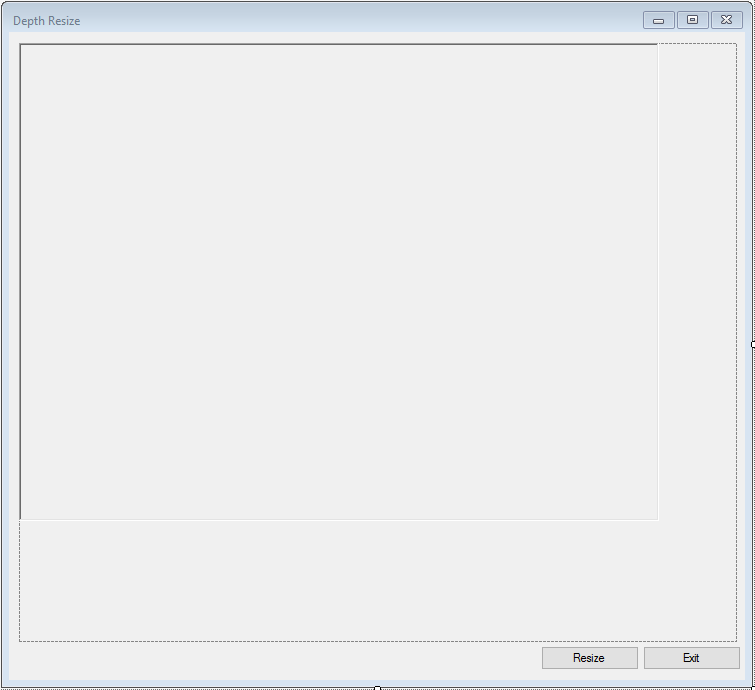


[FormDepthPasteOnPlane](#Detail_Form_DepthPasteOnPlane) – This form demonstrates how to use the paste on plane functionality to import an external image by clicking two points on a flat surface. 

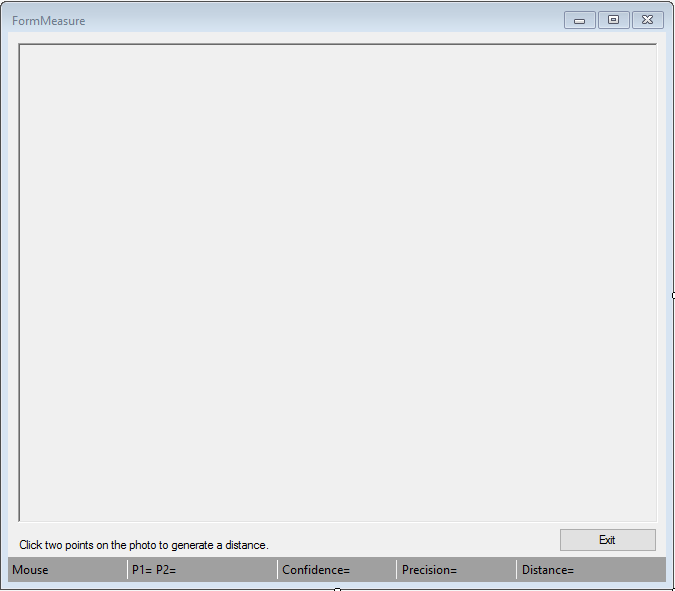
[FormDepthRefocus](#Detail_Form_DepthRefocus) – This form shows how to click on a point and have the focus point on an image brought to light by blurring the rest of the image. A user will click on a spot on the image which becomes the focus point. Also, allows a user to adjust the simulated aperture of the camera lens by changing the slider.



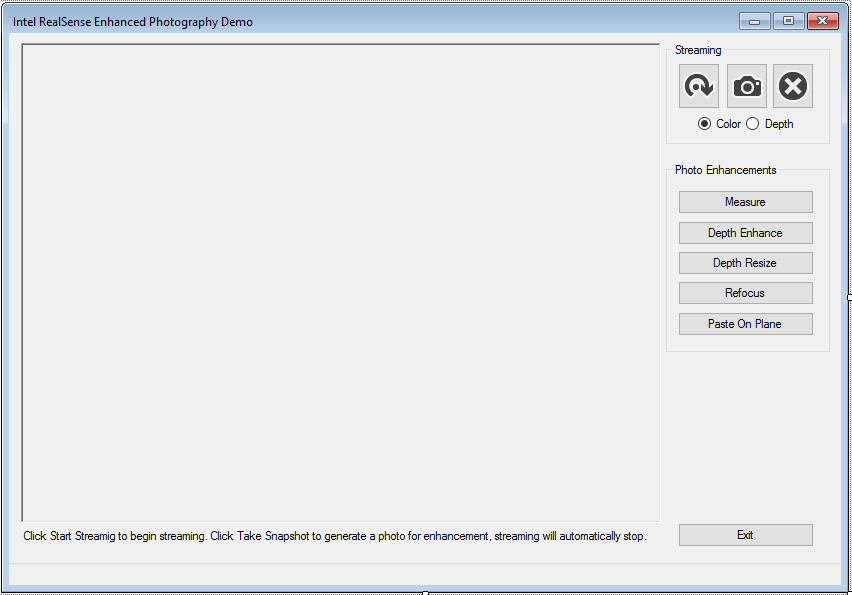
[FormDepthResize](#Detail_Form_DepthResize) – Shows the resize functionality can upsize the depth image to be the same size as the RGB image.



[FormMeasure](#Detail_Form_Measure) – Demonstrates how to use the enhanced photography measure capabilities to obtain distance, precision and confidence values.



[FormMain](#Detail_Form_Main) – This is the main form for the application. It allows the user to start the RealSense streaming, stop streaming and capture a snapshot. Once a snapshot has been taken, the user then has the ability to perform various enhanced photography functions on the image.



### Source Code

**RSEnhancedPhotography.CS** – This is a wrapper class that encapsulates the RealSense Enhanced Photography functionality. Its primary reason for existence is to try and remove as much of the RealSense functionality from the GUI layer as possible. It uses custom events to publish data back to the client app. It will use the RSEnhancedImageArg class to contain the new image which gets displayed.

**RSPaintUtil.CS** – This is a static utility class that assists in drawing mouse click points and lines onto the C# PictureBox controls.

**RSPoints.CS** – This helper class encapsulates PXCMPointI32 point objects and creates functionality to store points, validate point data as well as report point data to be displayed on the GUI.

**RSStreaming.CS** – This is a wrapper class that encapsulates RealSense streaming. It will stream data and publish an event back out to the client. The event uses the RSSampleArg class to store data to be used by the client.

**RSUtility.CS** – This is a static class that contains source code that is, as the name implies, utility. None of the functionality really belongs in any particular class.

### Source\CustomEventArgs Code

**RSEnahncedImageArg** – Extends EventArgs by containing a PXCMImage object. This object will contain an image that has been manipulated by the RealSense Enhanced Photography functionality. This image is to be used to display on the individual WinForms PictureBox control.

**RSMeasureArg.CS** – Extends EventArgs by containing measure data returned from the RealSense Enhanced Photography functionality. This data is used on the WinForm “FormMeasure” to report measurement information back to the user.

**RSSampleArg.CS** – Extends EventArgs by containing a PXCMCapture.Sample object. This object will contain the latest frame captured by the camera which is used for streaming data and displaying it on the WinForm FormMain.

# In depth Understanding

I’m going to talk about the underlying supporting classes that support the forms first. I think learning how the underlying code base works first is better than talking about the forms first. So, what that in mind let’s start with the first class RSStreaming

I won’t be covering every single detail such as getter and setter functions. Those are very self-explanatory. Nor will I cover any function that is clearly obvious for other reasons.

## RSStreaming

RSStreaming is a wrapper class around the RealSense streaming capabilities. This is a very light class in that it's not over complex. The intent here is to show a very simple example of how to stream data from the RealSense camera. It has the ability to both stream and take an Enhanced Photo image and send both back to the client via events.

public event EventHandler<RSSampleArg> OnNewStreamingSample;

As mentioned previously in this document, I am using Events to send data back to the client apps (Forms). RSStreaming sends data back to the client; in this case, FormMain via the event OnNewStreamingSample. As you can see it will take one parameter, [RSSampleArg](#Detail_RSSampleArg) which will contain the newest sample from the camera.

public bool Initialized

A simple getter flag that indicates if the class has been initialized.

public bool IsStreaming

A simple getting flag indicating if the class is currently streaming data.

public void StartStreaming( )

A public function that a client will use to start the streaming from the camera. Ensures that the class has been properly initialized and if so calls the InitCamera() function initialize the camera.

One key feature that I would like to explain about my sample is that I’m using a feature that does not get a lot of focus. As you have probably seen in a lot of samples for streaming, the sample will show a while loop and the AcquireAccess function with some type of mechanism to cancel streaming via a boolean flag. This sample uses a different approach.

My approach is to utilize the PXCMSenseManager’s StreamFrames function. This will cause the SenseManager to kick off its own internal thread and send data back via event handling. This is done by assigning the PXCMSenseManager.Handler object to a function. More on that later in the InitCamera( ) function.

private void InitCamera( )

InitCamera is a private function that will does just that… Initializes the camera and streaming. Starting out we create the PXCMSenseManager and PXCMSession objects. Next, we need the device information (Camera). This is gotten by making use of the [RSUtility](#Detail_RSUtility) GetDeviceByType() static function passing in the session and the type of camera we want.

Then I create two PXCMVideoModule.DataDesc objects, one for color streaming, one for depth streaming. From there I configure each stream. After the streams have been configured, I prompt the PXCMSenseManager to enable the streams.

As mentioned in the function StartStreaming(), I’m using an event based approach to streaming and gathering data. This is done by creating and intitializing a PXCMSenseManager.Handler event handler object and assigning it to the OnNewSample function. Every time the camera captures a new frame, the OnNewSample event handler will be called.   
Once this has all be accomplished, I then initialize the SenseManager sending it in the handler object telling it to use this object and it’s event hander.

private pxcmStatus OnNewSample( int mid, PXCMCapture.Sample sample )

OnNewSample is the event handler for the PXCMSenseManager.Handler object.

Parameters

* **mid** - The stream identifier. If multiple streams are requested through the EnableVideoStream[s] function, this is PXCCapture.CUID+0, or PXCCapture.CUID+1....
* **PXCMCapture.Sample** – The sample image that came from the camera.

When this function is called, I capture the image out of the Sample argument and put it into a new [RSSampleArg](#Detail_RSSampleArg) object then call this classes OnNewStreamingSample event. This forces the event to notify the client; [FormMain](#Detail_Form_Main) that a new image is ready to be displayed.

Release the frame and then return; as required, a pxcmStatus, which is not being used in this case.

public void StopStreaming( )

Stops the streaming by closing the streams and calling Dispose( ).

private void Dispose( )

Frees up resources for garbage collection.

## RSEnhancedPhotography

The RSEnhancedPhotography class was created to wrap the Enhanced Photography functionality functionality into one easy to use class. It works on an event principle. Once an image has been processed, an event is raised returning the newly created image or measurement data back the client app/class.

public RSEnhancedPhotography( PXCMPhoto photo )

Constructor initializes several of the global variables that are used in the class. The single input parameter is the original photo that was taken by the main form. It’s fully initialized with image data and used to initialize the local \_colorPhoto object.

public void Dispose( )

Releases the memory to be garbage collected.

public void SendOriginalImage( )

Returns the original image back to the calling application by making use of the OnImageProcessed event.

public void MeasurePoints( RSPoints points )

MeasurePoints receives a populated RSPoints object. First I ensure that there are indeed two valid points in this object, the start and end points. Once this has been determined a MeasureData object is created and sent into the PXCMEnhancedPhoto objects MeasureDistance function.

Next I take the data from the populated measureData object and populate the RSMeasureArg object. Notice the ( mesaureData.distance / 10 ) parameter. This is converting to centimeters. Once the arg object has been populated, I send it back to the client via the OnImageMeasured event.

public void RefocusOnPoint( RSPoints point, float aperture = 50 )

Imagine how a camera works, you set the aperture on a camera to get either a large or shallow depth of field. A small aperture and you get a large depth of field, a wide open aperture you get a shallow depth of field blurring items in front of or behind your subject.

RefocusOnPoint has the same affect. The function allows you to change your focal point in the image.

Yes, I know aperture settings don’t work in values of 0-100, but for the purposes of this example, they do. ☺ If you feel so inclined, please feel free to convert them to proper f-stops and send me the updated code.

RefocusOnPoint uses the PXCMEnhancedPhotos DepthRefocus function to create a new image with new depth focus using the original color photo, the point where the user clicked on the screen and an aperture setting. Once we have the newly created PXCMPhoto I get the reference image out by calling the QueryContainerImage() then supplying the PXCMImage to the RSEnahncedImageArg instance. From there it’s as simple as passing it back to the client app via the OnImageProcessed event.

public void DepthEnhancement( PXCMEnhancedPhoto.DepthFillQuality quality )

Enhances/Changes the depth quality of an image between two values, either high or low. This is specified in the DepthFillQuality parameter.

R5 seen the creation of a new class, PXCMEnhancedPhot.PhotoUtils. This class now contains the EnhanceDepth functionality. I create an instance of this class so that we can call the EnhanceDepth function.

After this class has been created the local PXCMPhoto gets initialize by calling the PhotoUtils’s EnhanceDepth function supplying the original PXCMPhoto and the quality specified.

Then to initialize the PXCMImage I use the enhancedPhoto‘s QueryDepthImage to supply us with the newly created depth image.

OnImageProcessed.

public void DepthResize( )

This shows a very simplistic way to resize a depth image. In this case it resizes the depth image to be the same size as the color image specified in the original PXCMPhoto that was created in the constructor.

First I need the size information from the color photo. This is accomplished by querying this out of the original PXCMPhoto object specified in the constructor. I then create the instance of the PhotoUtils object that contains the DepthReize function.

I get the size of the original image, I store the width and height in the required PXCMSizeI32 object.

From there it’s a simple process of telling the PhotoUtils to resize the depth image by calling the DepthResize function specifying the PXCMPhoto and target size.

Once the resizing is done, it’s the same thing. Create the image by querying for enhancedPhoto, populating the RSEnhancedArg and sending it back to the client via OnImageProcessed.

public void PastOnPlane( RSPoints points )

This function shows how a user can have a PXCMPhoto and paste a new image onto a flat surface. When doing so, the image being pasted will adapt to the environment. Meaning, if the image is pasted onto a wall, the image will look upright and have the same angle. If the image is pasted onto a desktop surface, the image will appear to lay down flat on the desk.

First thing we need to do is ensure that there are two valid points. This functionality requires to valid points, then create the objects needed to do the Pasting functionality.

After we know there are two valid points, we load up the image we want to paste by using the [RSUtility’](#Detail_RSUtility)s LoadBitmap function.

They key object in working on this is the the PXCMEnhancedPhoto.Paster class. This is a new class with the R5 release. Paster has a function PasteOnPlane that used to be directly on the PXCMEnhancedPhoto class, but was moved into this new class.

You will notice that in this function I’m taking a little bit extra caution by looking to see if the return value from PastOnPlane is not null. This is because there is no guarantee that the PasteOnPlane function was able to successfully perform the operation. If the surface between the two points is no flat, the function will not succeed for instance. I’m simply ensuring that I don’t use a null object.

If we have a successful return value, as you may have guessed, I get the reference image, store it, pass it into the RSEnhancedImageArg object and post it back to the client app.

## RSUtility

RSUtility is a static utility class that contains functionality that does not appropriately fit into any of the other classes.

public static PXCMCapture.DeviceInfo GetDeviceByType( PXCMSession session, PXCMCapture.DeviceModel deviceType )

This function is a helper function that is focused on getting the detail information about a device. In this case, the R200 camera. This functionality has been seen any multiple RealSense examples.

The first thing we do is setup a filter by specifying that we are looking for a sensor as the main group, then for the subgroup we specify a video capture sensor.

Because multiple devices can be on a given system we must iterate over all the possible devices. On each iteration I am populating the current PXCMSession.ImplDesc data in the currentGroup object. If there was no error we move onto the next check which is the need to populate the PXCMCapture object.

Once we have the PXCMCapture object I iterate over the various devices attached to this object. Check to see if the current device info is the camera we are looking for and if it is, we break out of the loop. If not, we move onto the next device information attached to the PXCMCapture device until all devices have been checked.

Once the device has been found, we return it to the client which in this case is the [RSStreaming](#Detail_RSStreaming) object.

public static Bitmap ToRGBBitmap( PXCMCapture.Sample sample )

A polymorphic function simply turns around and calls ToRGBBitmap( PXCMImage image ) passing it the sample argument’s Image.

public static Bitmap ToRGBBitmap( PXCMImage image )

Simple wrapper function that will use the PXCMImage objects functionality to get bitmap data using a PXCMImage.ImageData object. Data is pulled out and stored into a .NET bitmap object and returned to the client.

public static PXCMImage LoadBitmap( PXCMSession session )

This function is used to support PasteOnPlanes. It loads up a predetermined bitmap into a PXCMImage object and returns it to the client.

First it gets the path to the file and ensures the file exists. If not, returns null. If the file exists creates a brand new .NET bitmap object by loading it from a file.

A PXCMImage.ImageInfo object is used to hold basic information about the bitmap. This is used when creating the PXCMImage object, they are the specifications for the image we will create.

Next we need a PXCMImage.ImageData, this will contain and hold the actual bitmap data itself. A .NET BitmapData object is created and initialized with data that describes the format and structure of the data we need.

To fully understand what Bitmap.Lockbits is doing, refer to <https://msdn.microsoft.com/en-us/library/5ey6h79d(v=vs.110).aspx>.

The PXCMImage object releases access to the image data to free up the memory used, the bitmap unlocks its bits and the PXCMImage is returned to the client.

public static int GetDepthAtClickPoint( PXCMPhoto photo, PXCMPointI32 point )

**This function receives a PXCMPhoto and a PXCMPointI32. The photo is the image that we want to get depth from and the point is the location on the image where the mouse was clicked.**

**First thing that happens is creating the necessary PXCMImage objects and initializing them. I NEED TO COM BACK AND FIND OUT IF THERE IS A BETTER WAY TO DO THIS FUNCTION**

public static bool SavePhoto( PXCMSession session, PXCMCapture.Sample sample )

Saves the photo out to the disk. Uses the session to create a new PXCMPhoto object which has the functionality to save to disk. The PXCMPhoto object uses its import from preview sample to import the image data into itself. Find out of the file already exist, if so delete it and save the file.

public static PXCMPhoto OpenPhoto( PXCMSession session )

Straightforward function, ensures that the XDM photo exists on the hard drive, if so, uses the PXCMSession object to create the PXCMPhoto. The photo object then loads the XDM file. The function then returns the PXCMPhoto back to the client.

## RSPaintUtil

The RSPaintUtil class is a utility class to encapsulate the drawing of points and lines onto the picture boxes photos. Also, it reduces code duplication between different forms that rely on this functionality.

Ensures that there is a valid start point. If there is, create a new .NET Point object specifying the start points x,y values.

Calls the draw circle function to draw a circle around the point that was clicked.

static public void DrawTwoPointsAndLine( RSPoints points, PaintEventArgs e )

Ensures that both points are valid, creates two new .NET Point objects. The function draws circles at those points by calling DrawCircle for each. Then a line is drawn between them via DrawLine.

static private void DrawCircle( Point p, PaintEventArgs e )

Function draws a circle around the x,y coordinates of the .net Point object. This is done by creating a new Pen object. A rectangle is needed for drawing any circle, it’s what defines the size of the circle to be drawn. I created a utility bounding rectangle function to build this. Once the rectangle has been created, I use the paint event args DrawEllipse function to draw the circle the size of the rectangle.

static private void DrawLine( Point pointA, Point pointB, PaintEventArgs e )

As with the circle, we need to create a .NET Pen object, tell it the mode to use and draw the line using the event args DrawLine function between the start point and end point.

static public Rectangle BuildBoundingRectangle( Point p )

Builds a .NET rectangle object centered around the x,y values in p. This is done by creating a new Rectangle object, I wanted the bounding rectangle to be 10px by 10px. 10x10 was just an arbitrary value I selected.

## RSPoints

RSPoints is a simple wrapper for managing two different possible points. The points represent where a user clicked on a given PXCMPhoto being shown in a .NET PictureBox control

It users two PXCMPointI32 objects which represent a start point and and end point. In some situations a RSPoints instance will only need the start point, this would be the case for functionality such as RefocusOnPoint, other times two valid points will be need for functionality such as MeasurePoints which needs both start and end points.

The class will operate in two modes, single point mode, meaning we are doing operations that only require one valid point, or multi point mode which requires both start and end points to be valid for things like MeasurePoints.

public RSPoints( )

Constructor puts point mode into single, the calls ResetPoints to set all x,y values to 0.

public void AddPoint( int x, int y )

Yes, I realize this would be more akin to adding an object to an array or list. However as you can see there is no array or list. Just two points. But, I wanted to give this class a list type feel from the outside in the event I decided to add an array to contain points later. Was this class well thought out? Probably not, but at this point, I’m not too worried about this class, it’s just a supporting class at this time and does what I need it to. Anyway…

This is a little involved, so I will do my best to explain it. If we are in single point mode, AddPoints will always replace the start point. This is done via the ResetPoints object

If the mode is single point mode, I don’t necessarily want to add more points, rather just clear out the existing start point and set a new start point.

## RSSampleArg

RSSAmpleArg inherits from EventArgs. The intent of this class is to be used with the [RSStreaming](#Detail_RSStreaming). When RSStreaming is streaming, an instance of this class will be created on every new frame and populated with the data from the camera, then sent back to the client via an event class.

public RSSampleArg( PXCMCapture.Sample sample )

The constructor initializes the class’s local PXCMCapture.Sample \_sample object with the parameter.

public PXCMCapture.Sample Sample

Simple getter returns the local PXCMCapture.Sample object

## RSEnhancedImageArg

RSEnhancedImageArg inherits from EventArgs. The intent of this class is to be used with [RSEnhancedPhotography](#Detail_RSEnhancedPhotography). When the RSEnhancedPhotography has completed modifying an image, an instance of RSEnhancedImageArg will be create, populated with the newly created image and sent back to the client by the usage of an Event class that the client subscribes to.

public RSEnhancedImageArg( PXCMImage image )

The constructor initializes the only variable in the class, the PXCMImage object.

public PXCMImage EnhancedImage

A simple getter returns the PXCMImage instance

## RSMeasureArg

RSMeasureArg inherits from EventArgs. This event is used inside the RSEnhancedPhotography classes MeasurePoints function. When the MeasurePoints function has calculated the distance, an instance of this class is used to contain the distance, confidence and precision data returned. Once this object has been populated, it will be used with an Event object to send the data back to the client app.

public RSMeasureArg(float distance, float confidence, float precision)

Parameters

* Float – measurement distance between two points.
* Float – the level of confidence the SDK has regarding the measurement.
* Float – the precision level the SDK used.

Constructor populates the local data with the input parameters

public float Distance

Simple getter returns the distance between the two points

public float Confidence

Simple getter returns the confidence level calculated when the SDK processed the distance between the two points.

public float Precision

Simple getter returns the precision level calculated when the SDK processed the distance between the two points.

## FormMain

FormMain is the main form (entry point) for the application, it allows the user to control when to stream, stop streaming and taking a snapshot to be used by the other forms for the various Enhanced Photography functionality.

This form uses the RSStreamingRGB object to stream data from the camera. The data is then rendered to a .NET PictureBox control with the help of a RealSense utility object named D2D1Render.

The form gets its updates from RSStreamingRGB via subscribing to RSStreaming’s OnNewStreamingSample event.

public MainForm( )

This is the forms constructor so to speak. It initializes the forms global objects, sets up the even handler for the samples as they come in from RSStreamingRGB sets the buttons to the proper state.

private void rsRGBStream\_NewSample( object sender, RSSampleArg sampleArg )

The event handler for the RSStreamingRGB object. Checks to see if we want to view the color image or RGB image and updates the \_render utility object with the proper sample.

After that, checks to see if a new snapshot needs to be taken and if so, uses the RSUtilty to save to photo to disk.

private void btnStream\_Click( object sender, EventArgs e )

The event handler when the Stream button is clicked. Instructs the RSStreamingRGB object to start streaming, and sets the buttons according to the state of the app.

private void btnStopStream\_Click( object sender, EventArgs e )

The stop streaming button event handler, just calls the StopStreaming function.

private void btnTakeDepthPhoto\_Click( object sender, EventArgs e )

Event hander the snapshot button. Sets the flag to take a snapshot to true so that in the streaming event hander, it will know to save the current image data to disk, then instructs the form to stop streaming.

private void EnableEnhancedPhotoButtons( bool enable )

Sets the buttons according to the state of the app via the enable value.

You are going to probably look at this function and wonder what’s going on. You didn’t see any multithreading going on. Nothing spun up a thread. So, why am I using multithreading syntax to update the buttons? The reason is this. Even though the form is not launching any threads, no is RSStreamingRGB, there IS a separate threading executing.

Inside RSStreaming’s StartStreaming function, you will see a line of code

**\_senseManager.StreamFrames( false );**

Behind the scene this line of code (imbedded inside the RealSense SDK) is actually spawning a new thread. And because of this, we need to wrap up this functionality in multi-threaded syntax.

To start out, I have to check to see if the group box surrounding the various enhancement buttons requires an invoke. This is something that is required in dealing with windows controls in multi-threaded applications. If it’s required, then you have to create a new delegate, in this I’m creating an instance of the EnableEnhancedButtons delegate created at the top of the source code file. When creating an instance of a delegate, you must supply the name of the function you wish to call. In this case, we want to simply call the same function EnableEnhancedPhotobuttons. After the delegate has been created, we tell the form to invoke it sending in the original boolean value.

When the delegate calls the function, this time, the function will not pass the InvokeRequired test and fall into the else statement enabling the group box surrounding the enhancement buttons. The group box will either be enabled or disabled depending on the value in enable boolean value.

private void EnableStreamButtons( bool enable )

I’m not going into any detail here. This works exactly the same as EnableEnahcnedPhotoButtons does with the exception of it working with different controls to enable and or disable.

private void StopStreaming( )

Checks to ensure that the RSStreaming object has been properly initialized and if so, calls it’s stop streaming function. This kills the streaming from the camera and closes the thread that was generated.

Then I set the buttons accordingly.

private void btnExit\_Click( object sender, EventArgs e )

Click event handler for the exit button. Calls the form Close() function.

private void Form1\_FormClosing(object sender, FormClosingEventArgs e)

Handles the forms FormClosing event. Checks to see if the RSStreaming object was properly initialized and if so, forces it to stop streaming.

private void btnDepthResize\_Click( object sender, EventArgs e )  
private void btnDepthEnhance\_Click( object sender, EventArgs e )  
private void btnRefocus\_Click( object sender, EventArgs e )  
private void btnPasteOnPlane\_Click( object sender, EventArgs e )  
private void btnBlending\_Click( object sender, EventArgs e )

I grouped all these functions together in one explanation. They are all doing the exact same thing with the exception that each is initializing a different form. In each I am creating a new session object which is required by the RSUtility.OpenPhoto function. The OpenPhoto function will open the image that was created and saved to disk. Once the photo has been retrieved I creating a new form object passing it the photo then showing the form. Pretty straightforward stuff.

private void btn\_MouseEnter( object sender, EventArgs e )

Event hander when the mouse rolls over either the Start Streaming, Take Photo or Stop Streaming buttons. I get the button that was rolled over, then look at its “Tag” field. After determining which button was hovered over, I set the toolstrips text value to indicate what each button does.

private void btn\_MouseLeave( object sender, EventArgs e )

Sets the toolstrips text back to an empty string.

## FormMeasure

FormMeasure demonstrates how to use the Enhanced Photography’s measuring capabilities. It utilizes a RSEnhancedPhotography instance to talk to RealSense Enhanced Photography functionality.

public FormMeasure( PXCMPhoto photo )

The forms constructor. Accepts a PXCMPhoto that contains the image to be measured. The constructor initializes any and all global variables included in the class, and registers an OnImageMeasured event handler.

private void InitStatStrip( )

Creates default entries for the status strip. Sets values to empty strings and or 0,0 for x,y positions.

private void rsEnhanced\_OnImageMeasured(object sender, RSMeasureArg e)

The event handler for the OnImageMeasured event on the RSEnhancedPhotography object. Accepts the RSMeasuredArg which contains the information about the measurement. Updates the status strips text values.

private void pictureBox1\_MouseClick( object sender, MouseEventArgs e )

Event handler when a mouse is clicked on the picture box. The event handler starts of by trying to determine how many mouse clicks have been selected. If the there is no start point, then we know that this is the first time a user has clicked on the picturebox/photo. If this is the case then add the click point. If there is a valid start point, then we need to verify the existence of the end point. If it does not exist, add it.

Once we have determined the start and end points, I update the points in the status strip by calling UpdatePointsStatus.

Once we have to valid points; start point, end point, we can then call into the RSEnhancedPhoto’s MeasurePoints(…) function passing in the points object. We don’t need these points anymore so I clear them out by calling the RSPoint object’s ResetPoints() function.

After that, I call the pictureboxe’s invalidate to draw the points on the screen. Keep in mind this function will be called regardless of taking a measurement. This is so that we can always see the where the picture was clicked on.

private void UpdatePointStatus( )

Updates the status strips two point text values which shows the x,y on the image where it was clicked.

private void pictureBox1\_MouseMove( object sender, MouseEventArgs e )

Mouse move event handler, simply tracks where the mouse is at while moving around the image. Updates the status strips mouse x,y values.

private void pictureBox1\_Paint( object sender, PaintEventArgs e )

The picture boxes paint event handler. Responsible for drawing the two points and the line between them onto the image.

Checks to see if we have two valid points and if so, will draw both points and the line. If there is only the start point, it will only draw the start point marker onto the image.

private void Cleanup( )

Cleans up resources.

private void btnExit\_Click( object sender, EventArgs e )

The exit buttons click event handler forces the form to close.

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

The forms closing event handler. Calls cleanup to release resources.

## FormDepthEnhance

This form shows the how a depth image can be enhanced and displays the newly enhanced image in the picturebox control. It has the ability to show either low quality or high quality.

public FormDepthEnhance( PXCMPhoto photo )

The forms constructor initializes the variables used by the class sets up the picturebox control with the initial depth image which is going to be defaulted to LOW quality.

private void rdo\_Click( object sender, System.EventArgs e )

Event handler when one of the two radio buttons are selected. Calls the RSEnhancedPhoto’s DepthEnhancement(…) function passing in the value indicating high or low quality.

private void \_rsEnhanced\_OnImageProcessed( object sender, RSEnhancedImageArg e )

The OnImageProcessed event hander when RSEnhancedPhotography sends out the event. Simply updates the picturebox via the renderer helper object.

private void btnExit\_Click( object sender, System.EventArgs e )

Exit button click event handler forces the form to close

private void Cleanup( )

Nullify the objects to allow for garbage collection.

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

Form closing event handler. Forces cleanup of the variables.

## FormDepthResize

This form is used to show how a depth image can be resized. In this case, it gets resized to the size of the RGB image from the PXCMPhoto object.

public FormDepthResize( PXCMPhoto photo )

The forms constructor. Just as with the other form constructors, initializes the variables used by the form and sets the OnImageProcessed event handler.

private void btnResize\_Click( object sender, System.EventArgs e )

Button resize event handler simply calls the RSEnahancedPhoto’s DepthResize(…) function.

private void \_rsEnhanced\_OnImageProcessed( object sender, RSEnhancedImageArg e )

The OnImageProcessed event handler, resizes the picturebox and updates the image displayed via the renderer utility object.

private void btnExit\_Click( object sender, System.EventArgs e )

Exit button click event handler. Forces the form to close.

private void Cleanup( )

Nullifies the objects so they can be garbage collected.

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

Form closing event handler simply ensures that Cleanup is called.

## FormDepthRefocus

FormDepthRefocus demonstrates the refocusing capabilities of the RealSense Enhanced Photography functionality. When a user clicks on the image, the focal point around that spot is manipulated bringing it into focus while putting everything else out of focus.

Along with clicking on a spot on the image to focus on, there is a slider which can be used to simulate changing a cameras aperture.

public FormDepthRefocus( PXCMPhoto photo )

The forms constructor initializes the variables used by the form. Registers the enhanced photography objects OnImageProcessed event to a function and setups up the picturebox control.

private void pictureBox1\_MouseClick( object sender, MouseEventArgs e )

Click event handler for the picturebox. Sets the points objects start point to the cooridnates of the mouse click, calls the RSEnhancedPhotography’s RefocusOnPoint(…). The RSPoints ResPoints() removes the start point values, essentially resetting the object.

private void tbAperture\_Scroll( object sender, System.EventArgs e )

The scroll controls scroll event handler. Gets the value of the scroll, sets the forms text value representing the scroll value then calls the RSEnhancedPhotography’s RefocusOnPoint this time sending in a second parameter, the aperture size.

private void \_rsEnhanced\_OnImageProcessed( object sender, RSEnhancedImageArg e )

The event handler for the RSEnhancedObjects OnImageProcessed event. Simply uses the renderer utility class to update the picturebox control.

private void btnExit\_Click( object sender, System.EventArgs e )

Exit button click event handler. Forces the form to close.

private void Cleanup( )

Nullifies the objects so they can be garbage collected.

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

Form closing event handler simply ensures that Cleanup is called.

## FormDepthPasteOnPlane

This form demonstrates how a user can click on a flat surface (plane) that is present in the image and force a second image to be rendered onto that surface. A user will click two points on a flat surface and if the RealSense functionality can determine the surface, the external image will be pasted onto that flat surface.

public FormDepthPasteOnPlane( PXCMPhoto photo )

The forms constructor initializes the variables used by the form. Registers the enhanced photography objects OnImageProcessed event to a function and setups up the picturebox control.

private void \_rsEnhanced\_OnImageProcessed( object sender, RSEnhancedImageArg e )

The event handler for the RSEnhancedObjects OnImageProcessed event. Simply uses the renderer utility class to update the picturebox control.

private void pictureBox1\_MouseClick( object sender, MouseEventArgs e )

Event handler when a mouse is clicked on the picture box. The event handler starts of by trying to determine how many mouse clicks have been selected. If the there is no start point, then we know that this is the first time a user has clicked on the picturebox/photo. If this is the case then add the click point. After adding the point, I need to invalidate the picturebox control so that the click point can be drawn onto the image.

If there is a valid start point, then we need to verify the existence of the end point. If it does not exist, add it.

Once we have to valid points; start point, end point, we can then call into the RSEnhancedPhoto’s MeasurePasteOnPlane(…) function passing in the points object. We don’t need these points anymore so I clear them out by calling the RSPoint object’s ResetPoints() function.

private void btnExit\_Click( object sender, System.EventArgs e )

Exit button click event handler. Forces the form to close.

private void Cleanup( )

Nullifies the objects so they can be garbage collected.

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

Form closing event handler simply ensures that Cleanup is called.

## FormDepthBlending

FormDepthBlending is similar to plaste on planes. However this is different in that once an image has been imbedded, that image can be manipulated by changing the Yaw, Pitch, Roll, Zoffset and Scale which are all indicated by the sliders present on the form.

public FormDepthBlending( PXCMPhoto photo )

The forms constructor initializes the variables used by the form. Registers the enhanced photography objects OnImageProcessed event to a function and setups up the picturebox control.

private void pictureBox1\_MouseClick( object sender, MouseEventArgs e )

Event handler for the mouse clicks on the picturebox. Ensures that there are only ever one point in the points object which will be the StartPoint. It’s the only point that’s needed by this functionality. I capture the points then call the Blend() to do the actual blending of the external image into the PXCMPhoto.

private void Blend( )

Gets the values from each slider, creates the rotation array needed then calls the RSEnhancedPhotograpy’s DepthBlend function passing in all the parameters.

private void tbBlend\_Scroll( object sender, EventArgs e )

Event handler for any and all of the scrollers. Simply turns around and calls the Blend()

private void \_rsEnhanced\_OnImageProcessed( object sender, RSEnhancedImageArg e )

The event handler for the RSEnhancedObjects OnImageProcessed event. Simply uses the renderer utility class to update the picturebox control.

private void btnExit\_Click( object sender, System.EventArgs e )

Exit button click event handler. Forces the form to close.

private void Cleanup( )

Nullifies the objects so they can be garbage collected.

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

Form closing event handler simply ensures that Cleanup is called.