INTEL® PERCEPTUAL COMPUTING SDK

How to Capture Raw Image

LEGAL DISCLAIMER

THIS DOCUMENT CONTAINS INFORMATION ON PRODUCTS IN THE DESIGN PHASE OF DEVELOPMENT.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

INTEL MAY MAKE CHANGES TO SPECIFICATIONS AND PRODUCT DESCRIPTIONS AT ANY TIME, WITHOUT NOTICE. DESIGNERS MUST NOT RELY ON THE ABSENCE OR CHARACTERISTICS OF ANY FEATURES OR INSTRUCTIONS MARKED "RESERVED" OR "UNDEFINED." INTEL RESERVES THESE FOR FUTURE DEFINITION AND SHALL HAVE NO RESPONSIBILITY WHATSOEVER FOR CONFLICTS OR INCOMPATIBILITIES ARISING FROM FUTURE CHANGES TO THEM. THE INFORMATION HERE IS SUBJECT TO CHANGE WITHOUT NOTICE. DO NOT FINALIZE A DESIGN WITH THIS INFORMATION.

THE PRODUCTS DESCRIBED IN THIS DOCUMENT MAY CONTAIN DESIGN DEFECTS OR ERRORS KNOWN AS ERRATA WHICH MAY CAUSE THE PRODUCT TO DEVIATE FROM PUBLISHED SPECIFICATIONS. CURRENT CHARACTERIZED ERRATA ARE AVAILABLE ON REQUEST.

CONTACT YOUR LOCAL INTEL SALES OFFICE OR YOUR DISTRIBUTOR TO OBTAIN THE LATEST SPECIFICATIONS AND BEFORE PLACING YOUR PRODUCT ORDER.

COPIES OF DOCUMENTS WHICH HAVE AN ORDER NUMBER AND ARE REFERENCED IN THIS DOCUMENT, OR OTHER INTEL LITERATURE, MAY BE OBTAINED BY CALLING 1-800-548-4725, OR BY VISITING INTEL'S WEB SITE HTTP://WWW.INTEL.COM.

ANY SOFTWARE SOURCE CODE REPRINTED IN THIS DOCUMENT IS FURNISHED UNDER A SOFTWARE LICENSE AND MAY ONLY BE USED OR COPIED IN ACCORDANCE WITH THE TERMS OF THAT LICENSE ANY SOFTWARE SOURCE CODE REPRINTED IN THIS DOCUMENT IS FURNISHED UNDER A SOFTWARE LICENSE AND MAY ONLY BE USED OR COPIED IN ACCORDANCE WITH THE TERMS OF THAT LICENSE

INTEL, THE INTEL LOGO, INTEL CORE, INTEL MEDIA SOFTWARE DEVELOPMENT KIT (INTEL MEDIA SDK) ARE TRADEMARKS OR REGISTERED TRADEMARKS OF INTEL CORPORATION OR ITS SUBSIDIARIES IN THE UNITED STATES AND OTHER COUNTRIES.

MPEG IS AN INTERNATIONAL STANDARD FOR VIDEO COMPRESSION/DECOMPRESSION PROMOTED BY ISO. IMPLEMENTATIONS OF MPEG CODECS, OR MPEG ENABLED PLATFORMS MAY REQUIRE LICENSES FROM VARIOUS ENTITIES, INCLUDING INTEL CORPORATION.

*OTHER NAMES AND BRANDS MAY BE CLAIMED AS THE PROPERTY OF OTHERS.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Table of Contents

Introduction to RawImageCapture Sample

The Intel® Perceptual Computing SDK provides a set of C++ interfaces that define the functionalities of core frameworks, I/O modules, and algorithms modules. The **`PXCCapture`** interface is responsible for retrieving color and depth images data as well as audio data from input sensors.

The **`RawImageCapture`** sample shows three different ways to capture color and image data, and to manipulate and display the data. These samples demonstrate the flexibility of programming with the SDK by using hierarchical API interfaces (see Figure 1.)

1. Capture images directly via **`PXCCapture`** interface.
2. Capture images with **`UtilCapture`** utility.
3. Capture images with **`UtilPipeline`** utility.

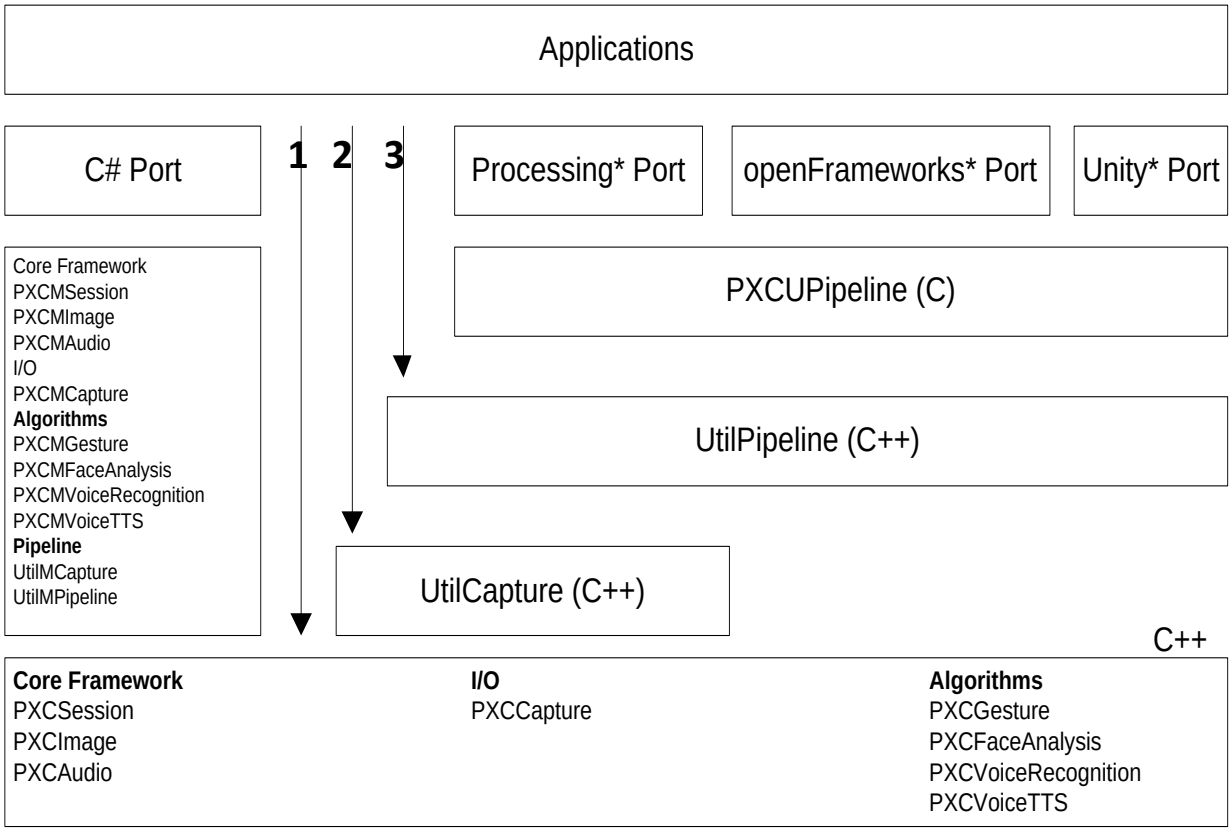

Figure 1: SDK API Interface Hierarchy

RawImageCapture Sample

The **RawImageCapture** sample uses three approaches to capture color and depth data

1. Capture images directly via **PXCCapture** interface.
2. Capture images with **UtilCapture** utility.
3. Capture images with **UtilPipeline** utility.

The sample displays color and depth images. However, the sample does not show how to render images. It also illustrates how to map depth pixels to color pixels if UVMap data is available. For how to create a new application, refer to the "**Getting Started Guide.**"

To build and run the sample

- In Windows Explorer, navigate to the **RawImageCapture** folder
- Double-click the icon for the **.sln** solution file to open the file in Microsoft* Visual Studio*.
- Build the application.
- Press CTRL + F5 to run the sample application

Header Files

The following are header files:

```
#include <stdio.h>
#include <windows.h>
#include <wchar.h>
#include <vector>

#include "pxcsession.h"
#include "pxcsmartptr.h"
#include "pxccapture.h"
#include "util_render.h"
#include "util_capture_file.h"
#include "util_pipeline.h"
```

- Use **pxcsession.h**: to create a session
- Use **pxcsmartptr.h**: to use SDK smart pointers
- Use **pxccapture.h**: to use **PXCCapture** interface APIs
- Use **util_render.h**: to render color and depth images
- Use **util_capture_file.h**: to use **UtilCapture** utility
- Use **util_pipeline.h**: to use **UtilPipeline** utility

The Main Program

The **wmain** function in **RawImageCapture.cpp** is the entry point of the sample program

```
int wmain(int argc, WCHAR* argv)
{
      CaptureDirect();
      CaptureUtility();
      CapturePipeline();
      CaptureUtility_UVMap();
}
```

- **CaptureDirect()**: capture color and depth data directly
- **CaptureUtility()**: capture color and depth data via **UtilCapture** utility
- **CapturePipeline()**: capture color and depth data via **UtilPipeline** utility
- **CaptureUtility_UVMap()**:capture color and depth data via **UtilCapture** utility and use UVMap data to map depth pixel coordinates to color pixel coordinates.

CapturePipeline

Both **CaptureDirect** and **CaptureUtility** require an application to write a roughly fixed amount of code even though the latter tries to provide some simpiflication.

This section explains how to leverage **UtilPipeline**, an SDK utility, to capture both color and depth data. The total number of lines of code is greatly reduced.

To use the `UtilPipeline` utility, let's first introduce a class that inherits the `UtilPipeline` class.

```
class RawPipeline: public UtilPipeline {
public:
      RawPipeline(void):UtilPipeline(), m_color_render(L"Color"),
                  m_depth_render(L"Depth"), m_total_frames (0) {
          EnableImage(PXCImage::IMAGE_TYPE_COLOR, 640, 480);
          EnableImage(PXCImage::IMAGE_TYPE_DEPTH, 320, 240);
      }

      virtual bool OnNewFrame(void) {
          bool status =
          m_color_render.RenderFrame(QueryImage(PXCImage::IMAGE_TYPE_COLOR));
          if (!status) return false;

          status =
          m_depth_render.RenderFrame(QueryImage(PXCImage::IMAGE_TYPE_DEPTH));
          if (!status) return false;

          m_total_frames++;
          if (m_total_frames == 300) return false;

          return true;
      }
```

```
protected:
    UtilRender m_color_render;
    UtilRender m_depth_render;
    Int        m_total_frames;
};
```

The constructor of the **RawPipeline** class enables the following images:

- color image with the VGA resolution (640x480)
- depth image with the QVGA resolution (320x240)

OnNewFrame is the callback function. When it is invoked, new color and depth frames are available. Then the function displays both image frames, and increase the total number of captured frames. When 300 frames have been captured, **OnNewFrame** returns false and the main **loop** is exited.

The actual implementation of **CapturePipeline** method is very simple. It first defines a **RawPipeline** object. Then its **LoopFrames** method is invoked.

```
int CapturePipeline()
{
      RawPipeline pipeline;
      Pipeline.LoopFrames()
      return 0;
}
```

In summary, **UtilPipeline** greatly simplifies the coding work to capture raw color and depth images. Please notice that **CapturePipeline** can also be used to enable both gesture and face recognition.

CaptureUtility

We will skip explanation of code that appears in the previous method.

```
int CaptureUtility()
{
      pxcStatus sts;

      // Create session
      PXCSmartPtr<PXCSession> session;
      sts = PXCSession_Create(&session);
      if (sts<PXC_STATUS_NO_ERROR) {
          wprintf(L"Failed to create a session\n");
          return 3;
      }

      UtilCaptureFile capture(session, 0, false);
```

```
// Set source device search critieria
capture.SetFilter(L"DepthSense Device 325");
PXCSizeU32 size_VGA = {640, 480};
capture.SetFilter(PXCImage::IMAGE_TYPE_COLOR, size_VGA);
PXCSizeU32 size_QVGA = {320, 240};
capture.SetFilter(PXCImage::IMAGE_TYPE_DEPTH, size_QVGA);

PXCCapture::VideoStream::DataDesc request;
memset(&request, 0, sizeof(request));
request.streams[0].format=PXCImage::COLOR_FORMAT_RGB32;
request.streams[1].format=PXCImage::COLOR_FORMAT_DEPTH;

sts = capture.LocateStreams (&request);
if (sts<PXC_STATUS_NO_ERROR) {
    wprintf(L"Failed to locate color and depth streams\n");
    return 1;
}
PXCCapture::DeviceInfo device_info;
sts = device->QueryDevice(&device_info);
```

```
UtilCaptureFile capture(session, 0, false);
```

This statement creates an **`UtilCaptureFile`** object. The last input value is **`false`**, indicating reading data frames from the sensor. If it is **`true`**, then data frames are read from a video file. This sample does not cover the details about how to read data from a file.

`SetFilter` are invoked three times to set up the following filters:

- device name, **`"DepthSense Device 325"`**
- color image type, and its resolution (VGA, 640x480)
- depth image type, and its resolution (QVGA, 320x240)

Furthermore, **`PXCCapture::VideoStream::DataDesc`** can be used to set up stream formats.

Finally,

```
sts = capture.LocateStreams(&request);
```

`LocateStreams` method of **`UtilCaptureFile`** is invoked to locate the streams that meet all the specified filtering conditions and create streams of the specified formats. The application does not need to explicitly create streams any more.

The following code is similar to that of **`CaptureDirect`** method. The only difference is that at this stage two streams (color and depth) have already been created. The depth stream is created only for **`DEPTHMAP`**.

```
PXCCapture::Device* device = capture.QueryDevice();

std::vector<UtilRender*> renders;
for (int idx=0;;idx++) {
```

```
        PXCCapture::VideoStream *stream_v = capture.QueryVideoStream(idx);
         if (stream_v) {
            PXCCapture::Device::StreamInfo sinfo;
            sts = device->QueryStream(idx, &sinfo);

            WCHAR stream_name[256];
            switch (sinfo.imageType) {
            case PXCImage::IMAGE_TYPE_COLOR:
                swprintf_s(stream_name, L"Stream#%d (Color)", idx);
                renders.push_back(new UtilRender(stream_name));
                break;
            case PXCImage::IMAGE_TYPE_DEPTH:
                swprintf_s(stream_name, L"Stream#%d (Depth)", idx);
                renders.push_back(new UtilRender(stream_name));
                break;
            default:
                break;
            }
         } else
            break;
    }
```

The main loop is slightly different.

```
    PXCSmartArray<PXCImage> images(2);
    PXCSmartSP sp;
    for (int f = 0; f<300; f++) {
          capture.ReadStreamAsync(images.ReleaseRefs(),sp.ReleaseRef());
          sp->Synchronize();

          if (renders[0]) {
              if (!renders[0]->RenderFrame(images[0])){
                   delete renders[0];
                   renders[0]=0;
                   break;
              }
          }

          if (renders[1]) {
              if (!renders[1]->RenderFrame(images[1])){
                   delete renders[1];
                   renders[1]=0;
                   break;
              }
          }
    }
```

Again the main loop reads and displays total 300 frames. Invoking **`ReadStreamAsync`** method of the **`capture`** object intends to ready both color and depth frames.

The **`Synchronize`** method returns whenever both new color and depth frames are available.

The **UtilRender** objects renders are used to display color and depth images. **RenderFrame** is invoked with a **PXCImage** object as input.

Before reading the next frame, the internal pointers of the image object and **SyncPoint** object are released.

Finally, the resource of the **renders** object are destroyed.

```
    if (renders[0]) delete renders[0];
    if (renders[1]) delete renders[1];

    return 0
}  // End of CaptureUtility
```

In summary, with **UtilCapture** (or **UtilCaptureFile**), an application does not need to explicitly:

- Create a device object
- Create streams

CaptureDirect

```
int CaptureDirect()
{
      pxcStatus sts;

      // Create session
      PXCSmartPtr<PXCSession> session;
      sts = PXCSession_Create(&session);
      if (sts<PXC_STATUS_NO_ERROR) {
          wprintf(L"Failed to create a session\n");
          return 3;
      }

      // Create capture
      PXCSession::ImplDesc desc;
      memset(&desc,0,sizeof(desc));
      desc.subgroup = PXCSession::IMPL_SUBGROUP_VIDEO_CAPTURE;
      desc.iuid = PXC_UID('S','K','D','2');
      PXCSmartPtr<PXCCapture> capture;
      sts = session->CreateImpl(&desc, PXCCapture::CUID, (void**)&capture);

      // Create capture device
      PXCSmartPtr<PXCCapture::Device> device;
      sts = capture->CreateDevice(0, &device);
      if (sts<PXC_STATUS_NO_ERROR) {
          wprintf(L"Failed to create device #%d\n", 0);
          return 3;
      }
      PXCCapture::DeviceInfo device_info;
      sts = device->QueryDevice(&device_info);
```

PXCSession_Create creates an SDK session that is the very first object that any SDK application needs to create. A **session** object is a context that SDK uses to host all I/O and/or algorithm modules.

ImplDesc structure is used to describe details of an I/O or algorithm module. It contains the following fields:

- **group**
- **subgroup**
- **algorithm**
- **iuid**
- **version**
- **acceleration**
- **merit**
- **vendor**
- **cuids**
- **friendlyName**
- **reserved**

The sample uses **group**, **subgroup**, and **iuid** fields to identify a proper capture module. In this sample, it is a capture module for **DepthSense** sensor. The invocation of the **session->CreateImpl** method is to create an instance of the capture module.

```
    sts = capture->CreateDevice(0, &device);
```

In this statement, capture object is used to create a device object with index 0.

```
    sts = device->QueryDevice(&device_info);
```

Device information can be obtained via **QueryDevice**. The device information is not utilized in this sample.

A device can output various video and/or audio streams. In the following block of code, stream information is queried. Rendering objects are created for color and depth streams correspondingly. Please note that for image type **IMAGE_TYPE_DEPTH**, there are two types of streams: one for **DEPTHMAP** and the other for **VERTICES**. This sample only cares about **DEPTHMAP** data.

```
    int i, depth_stream_index= -1;
    std::vector<UtilRender*> renders;
    for (i=0;;i++) {
        PXCCapture::Device::StreamInfo sinfo;
        sts = device->QueryStream(i, &sinfo);
        if (sts < PXC_STATUS_NO_ERROR) break;

        WCHAR stream_name[256];
        switch (sinfo.imageType) {
        case PXCImage::IMAGE_TYPE_COLOR:
```

```
            swprintf_s(stream_name, L"Stream#%d (Color)", i);
            renders.push_back(new UtilRender(stream_name));
            break;
        case PXCImage::IMAGE_TYPE_DEPTH:
            if (depth_stream_index < 0) {
                swprintf_s(stream_name, L"Stream#%d (Depth)", i);
                renders.push_back(new UtilRender(stream_name));
                depth_stream_index = i;
            }
            break;
        default:
            break;
        }
    }
```

It is time to create color and depth streams with **`CreateStream`**. Once a stream is created, profiles can be received and set through **`QueryProfile`** and **`SetProfile`** APIs.

```
    int num_streams = renders.size();
    if (!num_streams) {
        wprintf(L"No streams found in device!\n");
        return 1;
    }

    std::vector< PXCCapture::VideoStream* > streams(num_streams);
    for (i=0; i<num_streams; i++) {
        sts = device->CreateStream(i, PXCCapture::VideoStream::CUID,
                                   (void **)&streams[i]);
        if (sts<PXC_STATUS_NO_ERROR) {
            wprintf(L"Failed to create stream #%d\n", i);
            continue;
        }

        PXCCapture::VideoStream::ProfileInfo stream_profile;
        sts=streams[i]->QueryProfile(0, &stream_profile);
        if (sts<PXC_STATUS_NO_ERROR) {
            wprintf(L"Failed to QueryProfile for a video stream #%d\n", i);
            continue;
        }
        sts=streams[i]->SetProfile(stream_profile);
        if (sts<PXC_STATUS_NO_ERROR) {
            wprintf(L"Failed to SetProfile for a video stream #%d\n", i);
            continue;
        }
    }
```

Now let’s add code to start color and depth streams and then read image frames.

```
    PXCSmartSPArray sps(num_streams);
    PXCSmartArray<PXCImage> images(num_streams);

    for (i=0; i<num_streams; i++) {
        streams[i]->ReadStreamAsync (&images[i], &sps[i]);
```

Here variables **sps** and **images** are declared in arrays to hold **SyncPoint** objects and **PXCImage** objects, with array size as **num_streams (= 2)**. The loop is to read one color frame and one depth frame asynchronously.

The main loop is as follows:

```
      int nwindows = renders.size();
      for (int f = 0; f<300 && nwindows>0; f++) {
            pxcU32 stream_index = 0;
            if (sps.SynchronizeEx(&stream_index)<PXC_STATUS_NO_ERROR)
                break;

            if (renders[stream_index]) {
                if (!renders[stream_index]->RenderFrame(images[stream_index])){
                    delete renders[stream_index];
                    renders[stream_index]=0;
                    nwindows--;
                }
            }

            if (streams[stream_index]->ReadStreamAsync(
                images.ReleaseRef(stream_index), sps.ReleaseRef(stream_index))
                < PXC_STATUS_NO_ERROR)
                break;
      }
      sps.SynchronizeEx();
```

The main loop reads and displays a total of 300 frames. Color and depth data are displayed in two separate windows. When a window is killed, its corresponding images are not displayed any more. When both windows are killed, the program exits the loop.

SynchronizeEx returns whenever any of the two image frames is available. Here the return value **stream_index** indicates what kind of image frame.

The **UtilRender** objects **renders** are used to display color and depth images. When a frame is available, **RenderFrame** is invoked with a **PXCImage** object as input.

Before reading the next frame, the internal pointers of the image object and **SyncPoint** object are released.

Finally, the resource of the **renders** object is destroyed.

```
     if (renders[0]) delete renders[0];
     if (renders[1]) delete renders[1];
     return 0
}  // End of CaptureDirect
```

CaptureUtility_UVMap

Most code is same as or similar to **`CaptureUtility`**. We will skip explanation but focus on UV mapping.

```
int CaptureUtility_UVMap()
{
      pxcStatus sts;

      // Create session
      PXCSmartPtr<PXCSession> session;
      sts = PXCSession_Create(&session);
      if (sts<PXC_STATUS_NO_ERROR) {
          wprintf(L"Failed to create a session\n");
          return 3;
      }

      UtilCaptureFile capture(session, 0, false);

      // Set source device search critieria
      capture.SetFilter(L"DepthSense Device 325");
      PXCSizeU32 size_VGA = {640, 480};
      capture.SetFilter(PXCImage::IMAGE_TYPE_COLOR, size_VGA);
      PXCSizeU32 size_QVGA = {320, 240};
      capture.SetFilter(PXCImage::IMAGE TYPE DEPTH, size QVGA);

      PXCCapture::VideoStream::DataDesc request;
      memset(&request, 0, sizeof(request));
      request.streams[0].format=PXCImage::COLOR_FORMAT_RGB32;
      request.streams[1].format=PXCImage::COLOR_FORMAT_DEPTH;

      sts = capture.LocateStreams (&request);
      if (sts<PXC_STATUS_NO_ERROR) {
          wprintf(L"Failed to locate color and depth streams\n");
          return 1;
      }

      PXCCapture::Device* device = capture.QueryDevice();

      PXCCapture::VideoStream::ProfileInfo pinfo1;
      capture.QueryVideoStream(0)->QueryProfile(&pinfo1);
      PXCCapture::VideoStream::ProfileInfo pinfo2;
      capture.QueryVideoStream(1)->QueryProfile(&pinfo2);

      std::vector<UtilRender*> renders;
      renders.push_back(new UtilRender(L"Color with UVMap"));
      renders.push_back(new UtilRender(L"Depth"));

      for (int f=0;f<300;f++) {
          PXCSmartArray<PXCImage> images(2);
          PXCSmartSP sp;

          capture.ReadStreamAsync(images, &sp);
          sp->Synchronize();
```

```
        if (renders[0]) {
          PXCImage::ImageData data0;
          PXCImage::ImageData data1;
          images[0]->AcquireAccess(PXCImage::ACCESS_READ_WRITE,
                        PXCImage::COLOR_FORMAT_RGB32,&data0);
          images[1]->AcquireAccess(PXCImage::ACCESS_READ,&data1);

          float *uvmap=(float*)data1.planes[2];
          float *depthmap = (float*)data1.planes[0];
          for (int y=0;y<(int)pinfo2.imageInfo.height;y++) {
             for (int x=0;x<(int)pinfo2.imageInfo.width;x++) {
                int xx=(int)(uvmap[(y*pinfo2.imageInfo.width+x)*2+0]
                             *pinfo1.imageInfo.width+0.5);
                int yy=(int)(uvmap[(y*pinfo2.imageInfo.width+x)*2+1]
                             *pinfo1.imageInfo.height+0.5);
                if (xx>=0 && xx<(int)pinfo1.imageInfo.width && yy>=0 &&
                                        yy<(int)pinfo1.imageInfo.height)
                   ((pxcU32 *)data0.planes[0])[yy*pinfo1.imageInfo.width+xx]
                     = 0x80FF0000;
             }
          }
          images[0]->ReleaseAccess(&data0);
          images[1]->ReleaseAccess(&data1);

          if (!renders[0]->RenderFrame(images[0])) {
              delete renders[0];
              renders[0]=0;
              break;
          }
      }

      if (!renders[1]->RenderFrame(images[1])) {
          delete renders[1];
          renders[1]=0;
          break;
      }

      images.ReleaseRef(0);
      images.ReleaseRef(1);
   }

   // destroy resources
   if (renders[0]) delete renders[0];
   if (renders[1]) delete renders[1];

   return 0;
}
```

Let's concentrate on the main loop.

First, once new color and depth images arrive, **`AcquireAccess`** is invoked for data access:

- For color data, the access type is **`PXCImage::ACCESS_READ_WRITE`**. Its data will be read first. With UV mapping, some of the data will be overwritten.
- For depth data, the access type is **`PXCImage::ACCESS_READ`**, read only.

Secondly, **`PXCImage::ImageData`** has data "**`planes`**." Color data is always stored in **`planes[0]`**. Depth data is organized as follows:

- **`planes[0]`** stores **`DEPTHMAP`** data or **`VERTICES`** data
- **`planes[1]`** stores **`CONFIDENCEMAP`** data
- **`planes[2]`** stores **`UVMAP`** data

The **`UVMAP`** data can be used to map depth pixel coordinates to color pixel coordinates. The mapping algorithm in this sample is straightforward:

- For each depth pixel, compute the color pixel coordinates based on its **`(u, v)`** map values.
- For the mapped color pixel, change its RGB values to the red color value **`0x80FF0000`**.

The color screen in Figure 2 shows a color image with uvmapping. The red pixels are mapped from depth pixels. You may notice that UVmap data are sensitive to the distance and light source.

Figure 2: UV Map Visualization

Project Settings

`UtilCapture` and **`UtilPipeline`** are provided by the **`libpxcutils`** library. To use the library, you will need to configure the following project settings (see Figure 3):

- Add **`$(PCSDK_DIR)\include`** and **`$(PCSDK_DIR)\sample\common\include`** to the "**`Include Directories`**" of "VC++ Directories"
- Add **`$(PCSDK_DIR)\lib\$(Platform)`** and **`$(PCSDK_DIR)\sample\common\lib\$(PlatformName)\$(PlatformToolset)`** to the "**`Library Directories`**" of "VC++ Directories"
- In "**`Debug`**" mode, add **`libpxc_d.lib`** and **`libpxcutils_d.lib`** to your project additional libraries. Remove "**`_d`**" if it is "**`Release`**" mode.

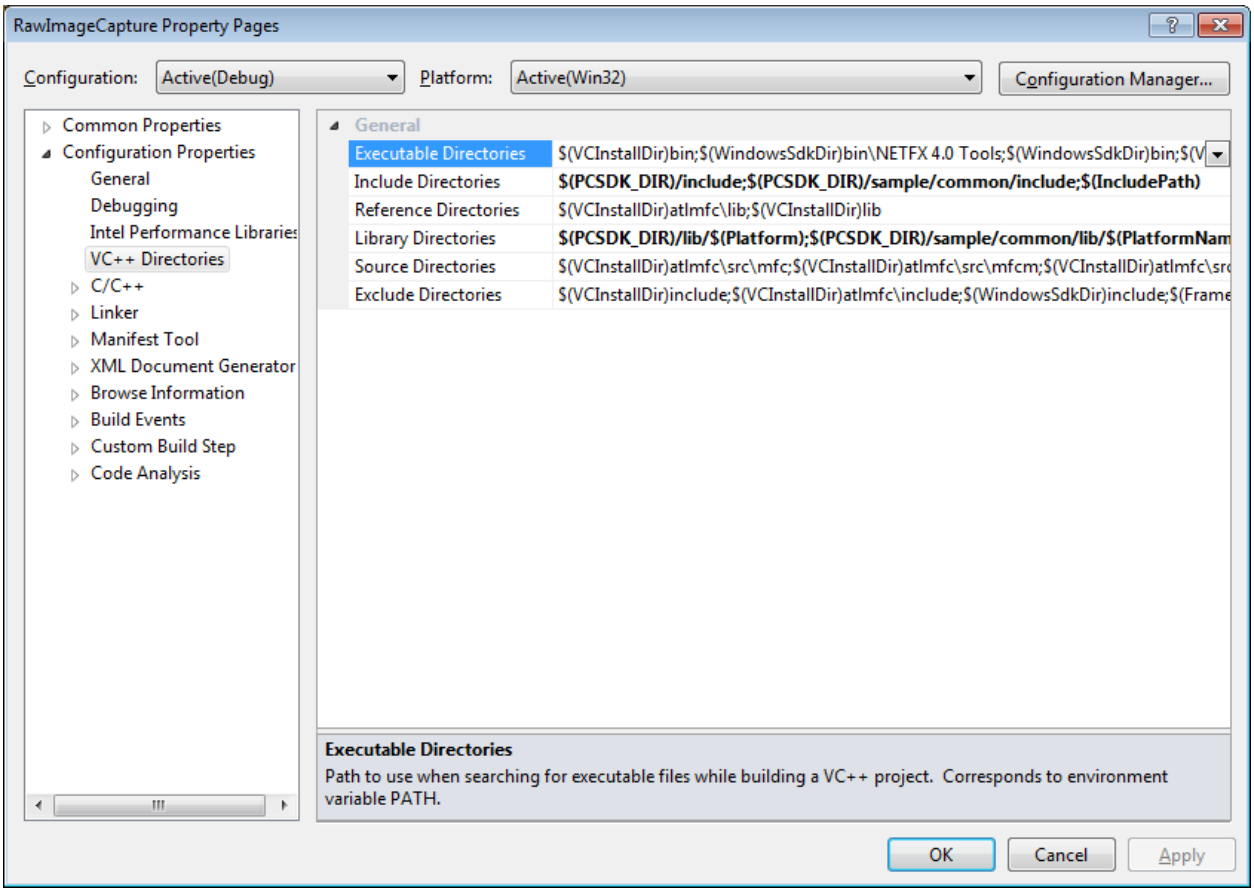

Figure 3: Project Settings