CamCube

SOFTWARE DEVELOPMENT TUTORIAL

Written by Martin Profittlich
Manager Software Development Group
PMDTechnologies GmbH

Table of Contents

1 INTRODUCTION 3

2 GETTING 3D DATA 4

3 OPTIMIZING THE DATA QUALITY 6

4 OPTIMIZING THE FRAME RATE 8

5 USING THE DATA TO DO SOMETHING USEFUL 10

6 FINAL WORDS 12

1 Introduction

In this paper, we will take a look at the PMD[vision]® CamCube time-of-flight camera and how to develop applications for it using the PMDSDK2.

The PMD[vision]® CamCube has a PMD chip with 204x204 pixels. This means we get more that 41 thousand distance values for each frame with up to 25 frames per second. The camera operates at a standard modulation frequency of 20 MHz, which gives us an unambiguous range of (very close to) 7.5 meters. The integrated suppression of background illumination (SBI) provides the CamCube with an enhanced dynamic range so that it can operate even in bright environments.

The PMDSDK2 is used to communicate with the PMD[vision]® CamCube. It provides a simple C API that allows us to retrieve the 3D and 2D data and to configure the camera, e.g. set the integration time (exposure time) of the CamCube. We will create some simple C programs to get to know the camera and the API and then write an application that uses the CamCube to do something useful.

Full source code and Visual Studio project files for all examples in this document are available so it is not necessary to recreate the examples by hand. For every chapter with an example, there is a corresponding chapterX.zip file.

Before we start, we should take a look at the different types of data that the PMD[vision]® CamCube generates. The most important image is of course the *distance* image. In each pixel, it provides us with a distance from the camera to the observed object. If the field of view (and possibly other lens properties) of the camera is known, it is possible to calculate Cartesian coordinates from the distances.

The *amplitudes* provide additional useful information: The higher the amplitude value of a pixel, the more reliable is its corresponding distance value. If the camera observes a scene with good reflectivity (in the infrared spectrum), the amplitude values will be high. On the other hand, if the camera is directed at the sky, where there is no object to be observed, the amplitudes will be close to zero. Objects with very low reflectivity will produce noisy distance values but will also produce low amplitudes. Therefore it is possible to assess the quality of the distance value of a pixel by looking at its amplitude value. We can then choose to ignore distances below a certain amplitude threshold to ensure that we only work with reliable values.

Finally, the *intensity* image is similar to a simple grayscale image from a traditional 2D camera: The more light reaches a pixel, the higher is its intensity value. However, cameras with an SBI – such as the PMD[vision]® CamCube – will produce counter-intuitive intensities when the SBI kicks in. As soon as the SBI starts to suppress background light, the intensity value for the respective pixel will drop again, despite the still increasing amount of light that reaches the sensor. In these situations, it might be better to use the amplitude image for grayscale display or 2D image processing.

2 Getting 3D data

To be able to connect to the camera and capture distance images, we must first understand how the PMDSDK2 works. The PMDSDK2 provides a set of functions to interact with PMD cameras or other devices or data sources. However, it does not contain any real functionality – the actual work is done in plugins. For each connection, a set of two plugins is needed. The first one – the *Source Plugin* – handles the connection to the device, its configuration and the image acquisition. The other one – the *Processing Plugin* – uses the acquired source data to convert or calculate the distance images and other kinds of data. For the CamCube, these plugins are called *camcube* and *camcubeproc.*

To use the CamCube, we need to open a connection first. The following program does nothing except open a camera connection and close it immediately:

```
#include <stdio.h>
#include <stdlib.h>
#include <pmdsdk2.h>

void doStuff (PMDHandle hnd);

int main (void)
{
  /* The handle is used to identify a connection */
  PMDHandle hnd;
  /* Result code for PMDSDK2 calls */
  int res;

  /* Open connection using the plugins camcube and
     camcubeproc. hnd is set if the call succeeds.
     The empty strings are parameters for the plugins */
  res = pmdOpen (&hnd, “camcube”, “”, “camcubeproc”, “”);

  /* Check if the call succeeded */
  if (res != PMD_OK)
    {
      char err[256];
      /* The handle parameter in pmdGetLastError is 0
         because no connection was established */
      pmdGetLastError (0, err, 256);
      fprintf (stderr, “Could not connect: %s\n”, err);
```

```
      return 1;
    }

  doStuff (hnd);

  /* Close the connection */
  pmdClose (hnd);

  return 0;
}

void doStuff (PMDHandle hnd)
{
  /* We will fill this function later on */
}
```

Now that we have established a connection to the PMD[vision]® CamCube, we can get to the interesting part: Retrieving image data. For this, we use the *doStuff()* function in the above example, but before that, we introduce a simple error handling function to keep *doStuff()* clutter free:

```
/* We put error handling in an extra function to keep
   the interesting code short
*/
void checkError (PMDHandle hnd, int code)
{
  /* Only do something if the result code is not PMD_OK */
  if (code != PMD_OK)
    {
      char err[256];

      /* Get an error description from the PMDSDK2 */
      pmdGetLastError (hnd, err, 256);
      /* Display the error message */
      fprintf (stderr, "Error: %s\n", err);
      /* Close the connection */
      pmdClose (hnd);
      /* For our example programs, we just quit */
      exit (1);
    }
}
```

And now the interesting part:

```
void doStuff (PMDHandle hnd)
{
  /* We know how much memory we need for a distance image.
     In a real application, it is better to query this
     information with pmdGetSourceDataDescription() */
  float dist[204*204];
  int res;

  /* Take a picture */
  res = pmdUpdate (hnd);
  checkError (hnd, res);
```

```
  /* Get a distance image */
  res = pmdGetDistances (hnd, dist, sizeof (dist));
  checkError (hnd, res);

  printf (“A distance from the middle: %f m\n”,
          dist[204 * 102 + 102]);
}
```

We use *pmdUpdate()* to grab a frame from the CamCube and *pmdGetDistances()* to calculate and collect the distance data. When *pmdUpdate()* is called, a new image acquisition is started by the CamCube. The call returns when the image is complete and transferred to the PC. Successive *pmdGetDistances()* calls will produce the same data until *pmdUpdate()* is called again. For the other image types, there are functions that behave essentially the same: *pmdGetAmplitudes()*, *pmdGetIntensities()* and *pmdGet3DCoordinates().*

3 Optimizing the data quality

Now that we can acquire distance data from the camera, we can take a look at what we have to do to get the best possible images from the CamCube.

The first step to good quality distance information is to adjust the integration time of the PMD[vision]® CamCube to an adequate value. A general suggestion for an integration time cannot be made, because the data acquisition depends on too many factors. In general, the integration time has to be adjusted to the total amount of light that reaches the PMD sensor. The amount of light on the chip depends primarily on the following factors:

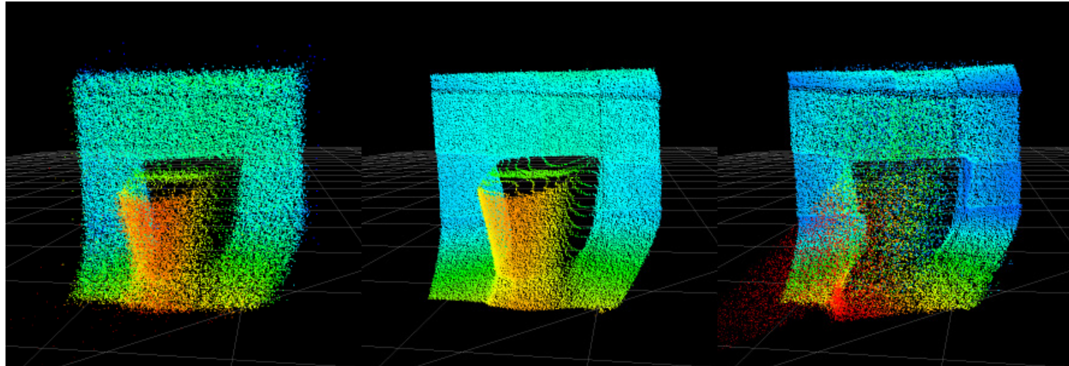
- The integration time setting of the camera
- The reflectivity of the observed object
- The strength of the camera’s light sources
- The amount of background light in the scene

Under normal circumstances, of the above, only the integration time can be influenced. This is done by the call *pmdSetIntegrationTime()*:

```
/* Set an integration time of 1300 microsec. The second
   parameter is for cameras that internally support
   multiple integration times. For the CamCube, this
   is always 0. */
res = pmdSetIntegrationTime (hnd, 0, 2300);
checkError (hnd, res);
```

When choosing the right integration time, we have to take signal strength and saturation into account. If the integration time is too short, the amplitudes, i.e. the signal strength, will be low and the distance values will be noisy. Too long integration times lead to saturation. The SBI feature (suppression of

background illumination) pushes the limit of saturation far ahead of what a traditional sensor can do, but even with SBI, there is a limit to the dynamic range. When too much light hits the sensor, some pixels will experience saturation. Saturation can also be automatically detected, but not this is more difficult and not within the scope of this paper.



Integration times of 140 µs, 1400 µs and 14000 µs. Note the low signal strength on the left and the saturation on the right side due to unsuitable integration times.

The trick is to choose an integration time that ensures that all pixels have a strong signal (as determined by the amplitude image), but no pixels are saturated. This can be seen in the middle picture above. In some situations, this is not possible. In scenes with objects of very different reflectivities, there might be situations in which there are always some noisy and some saturated pixels. There are two possible strategies to deal with these situations. The first one is to choose an integration time that minimizes the number of erroneous pixels and to live with having to remove some values from the distance image. The second option is to alternate between a long and a short integration time. While this approach reduces the effective frame rate of the CamCube, it can greatly increase the dynamic range, similar to HDR images in photography.

The following function shows how to implement a simple merging algorithm.

```
void hdrImage (PMDHandle hnd)
{
  int res;
  float amplitude[2][204*204];
  float distance[2][204*204];

  /* Take a picture with the short integration time */
  res = pmdSetIntegrationTime (hnd, 0, SHORT_TIME);
  checkError (hnd, res);

  res = pmdUpdate (hnd);
  checkError (hnd, res);

  res = pmdGetAmplitudes (hnd, amplitude[0],
                          sizeof (float) * 204 * 204);
  checkError (hnd, res);

  res = pmdGetDistances (hnd, distance[0],
                         sizeof (float) * 204 * 204);
  checkError (hnd, res);
```

```
  /* Take a picture with the long integration time */
  res = pmdSetIntegrationTime (hnd, 0, LONG_TIME);
  checkError (hnd, res);

  res = pmdUpdate (hnd);
  checkError (hnd, res);

  res = pmdGetAmplitudes (hnd, amplitude[1],
                          sizeof (float) * 204 * 204);
  checkError (hnd, res);

  res = pmdGetDistances (hnd, distance[1],
                             sizeof (float) * 204 * 204);
  checkError (hnd, res);

  /* Check every pixel: If the amplitude is too low, use
     the long measurement, otherwise keep the short one */
  for (unsigned i = 0; i < 204 * 204; ++i)
    {
      if (amplitude[0][i] < AMPL_THRESHOLD)
        {
          amplitude[0][i] = amplitude[1][i];
          distance[0][i] = distance[1][i];
        }
    }

  /* amplitude[0] and distance[0] now contain the merged
     data */
}
```

This approach can be extended to any number of measurements to further improve the dynamic range. With more sophisticated merging algorithms, the results are even better.

4 Optimizing the frame rate

To ensure the highest possible frame rate, we must keep the time between *pmdUpdate()* calls as short as possible. We can do this by calling *pmdUpdate()* repeatedly in its own thread and letting the main thread (or any other thread) handle the calculation and image processing. To keep even the distance calculation out of the image acquisition thread, we only use *pmdGetSourceData()* to obtain the raw data that the camera produces. The source data is then handed over to the main thread for further processing with *pmdCalcDistances(), pmdCalcAmplitudes()* etc. This is especially useful, because the CamCube starts the image acquisition upon calling *pmdUpdate()* and every delay between these calls affect the frame rate directly.

For setting up threading and inter-thread communication in our application, refer to the example source code. Here, we will focus on what to do with the PMDSDK2 inside the threads.

In the receiver thread, we repeatedly call *getData()*:

```
void getData (PMDHandle hnd,
              PMDDataDescription * dd, char ** sourceData)
{
  int res;

  /* Take a picture */
  res = pmdUpdate (hnd);
  checkError (hnd, res);

  /* Get information about the frame */
  res = pmdGetSourceDataDescription (hnd, dd);
  checkError (hnd, res);

  /* Reserve memory for the frame */
  if (sourceSize != dd->size)
    {
      delete[] *sourceData;
      *sourceData = new char[dd->size];
      sourceSize = dd->size;
    }

  /* Get the frame raw data */
  res = pmdGetSourceData (hnd, *sourceData, dd->size);
  checkError (hnd, res);
}
```

This is the smallest possible amount of work we can do in the receiver thread.

In the processing thread, we do this:

```
void calcDistance (PMDHandle hnd, PMDDataDescription dd,
                   char * sourceData)
{
  int res;
  unsigned numPixels = dd.img.numRows * dd.img.numColumns;

  /* Reserve memory for the distance image */
  if (distSize != numPixels)
    {
      delete[] dist;
      dist = new float [numPixels];
      distSize = numPixels;
    }

  /* Calculate distances from the raw data */
  res = pmdCalcDistances (hnd, dist,
      numPixels * sizeof (float),
      dd, sourceData);
  checkError (hnd, res);

  /* Output a distance value from the center */
  printf ("Middle distance: %f m\n",
          dist[numPixels / 2 + dd.img.numColumns / 2]);
}
```

This function takes the data that was retrieved in the receiver thread and calculates and displays distance values. In a real application, we could then proceed to do something useful with the distances.

5 Using the data to do something useful

Now that we have the ability to get accurate and fast distance information, we want to use the data for something useful. In this example, we will use the CamCube as a simple human input device. We will be able to move the mouse pointer by pointing our hand towards the camera and moving it in the desired direction. We will use a relatively simple algorithm for this. It is left as an exercise to the reader to make the detection more robust for changing scenes and to add further functionality such as clicking the mouse buttons.

To implement this little tool, we take the code from the last chapter and replace the *calcDistance()* function with a function called *moveMouse()*:

```
#define MAX_DIST      0.8
#define MIN_AMP       50.0
#define LEFT          80
#define RIGHT         120
#define TOP           80
#define BOTTOM        120
#define MOUSE_SPEED   3

void moveMouse (PMDHandle hnd, PMDDataDescription dd,
                               char * sourceData)
{
  int res;

  /* Get the distances */
  float * dist = new float [dd.img.numRows *
                            dd.img.numColumns];

  res = pmdCalcDistances (hnd, dist,
      dd.img.numColumns * dd.img.numRows * sizeof (float),
      dd, sourceData);
  checkError (hnd, res);

  /* Get the amplitudes */
  float * amp = new float [dd.img.numRows *
                            dd.img.numColumns];

  res = pmdCalcAmplitudes (hnd, amp,
      dd.img.numColumns * dd.img.numRows * sizeof (float),
      dd, sourceData);
  checkError (hnd, res);

  /* Filter the distances to suppress shot noise */
  median (dd, dist, amp);

  /* Find the pointer position.
     We just use the closest pixel we can find. If all
```

```
   pixels are more than 80 cm away, we do nothing.
   Pixels with low amplitudes are ignored. */
unsigned closest = 0;
unsigned i;
bool foundPointer = false;

for (i = 0; i < dd.img.numRows * dd.img.numColumns; ++i)
  {
    if (amp[i] > MIN_AMP &&
        dist[i] < dist[closest] && dist[i] < MAX_DIST)
      {
        closest = i;
        foundPointer = true;
      }
  }

unsigned row, col;
row = closest / dd.img.numColumns;
col = closest - row * dd.img.numColumns;

/* Do something with the found position: Move the mouse
   pointer in the desired direction */
printf ("Position: ");

if (!foundPointer)
  {
    printf ("none");
  }
else
  {
    if (row < TOP)
      {
        printf ("top    ");
        POINT pt;
        GetCursorPos (&pt);
        SetCursorPos (pt.x, pt.y - MOUSE_SPEED);
      }
    else if (row > BOTTOM)
      {
        printf ("bottom ");
        POINT pt;
        GetCursorPos (&pt);
        SetCursorPos (pt.x, pt.y + MOUSE_SPEED);
      }
    else
      {
        printf ("center ");
      }

    printf ("\t");

    if (col < LEFT)
      {
        printf ("left ");
        POINT pt;
```

```
          GetCursorPos (&pt);
          SetCursorPos (pt.x - MOUSE_SPEED, pt.y);
        }
      else if (col > RIGHT)
        {
          printf ("right ");
          POINT pt;
          GetCursorPos (&pt);
          SetCursorPos (pt.x + MOUSE_SPEED, pt.y);
        }
      else
        {
          printf ("center ");
        }
    }

  printf ("   \r");

  delete[] dist;
  delete[] amp;
}
```

First, we calculate the distances and amplitudes and apply a median filter to the distances to remove noisy pixels. Then we look for the closest pixel in the image that has a high enough amplitude (*MIN_AMP*). To prevent the background of the scene from interfering with the mouse, we only look at pixels that are closer than a certain distance threshold (*MAX_DIST*). If we have found such a pixel, we determine where it is in the image and move the mouse pointer accordingly.

For best results, adjust the thresholds as well as the integration time to suit the scene. It is a good idea to use *CamVis 3* to check whether there is noise or saturation and to get a feel for interacting with the camera.

6 Final words

As we have seen, it is comparatively easy to get meaningful data from the PMD[vision]® CamCube and use it in an application. With just a few simple commands like *pmdUpdate()* and *pmdGetDistances()* it is possible to integrate the CamCube – or any other PMD camera – into powerful applications. Further information about the different commands of the PMDSDK2 can be found in the *PMDSDK2 Programming Manual*. For more information about the PMD[vision]® CamCube, this tutorial or PMD technology in general, visit *http://www.pmdtec.com* or contact us at *info@pmdtec.com*.

PMDTechnologies GmbH
Am Eichenhang 50

57076 Siegen
Germany
Phone +49(0)271 / 238 538- 800
Fax +49(0)271 / 238 538- 809
info@PMDTec.com
www.PMDTec.com