PERCEPTUAL COMPUTING: GENERATING 3D FROM DEPTH DATA

By: Lee Bamber

1. Introduction

Working with perceptual computing hardware, you will soon learn that to move beyond the confines of the Intel® Perceptual Computing SDK you need a good understanding of the depth data provided by the camera and how to manipulate it. This article will cover the basics of obtaining this data, how it relates to the camera, and how to convert it to a textured 3D model for rendering.

As a recent participant in two of Intel’s Ultimate Coder challenges, I’ve had the opportunity to contend with this technology over the last two months, and my background in designing and developing 3D game engines helped me overcome many of the hurdles you will doubtless encounter on your own journey.

As a prerequisite, you should have a broad understanding of 3D concepts such as vectors, vertex formats, and a basic knowledge of C.

2. Why Is 3D from Depth Important

If you are new to perceptual computing, you will likely start with the Intel Perceptual Computing SDK and explore the methods described by the documentation and the code samples. When you want to explore functionality beyond those methods, you will need to harness the raw data directly. By converting the depth data to a 3D format such as a mesh or point cloud, you gain access to additional techniques and opportunities for creativity.

The primary reason for obtaining a 3D representation of the world in front of the camera is to do simple rendering, or for more advanced control systems such as finger and body tracking.

This article will explore this technique using the C++ API, but the concepts are identical whether you are a Unity® or C# developer. Unity developers would need to create their own third-party module to implement this functionality in their apps.

3. The Technique Explained

The approach is divided into two main steps. The first is to read the depth data from the camera and store the values in an array. The second step is to filter this data and convert it to a three dimensional coordinate from which a mesh can be created.

The idea behind the technique is very simple. No data is adversely changed during the process; it is simply translated from one context to another. The depth data exists as a two dimensional array of 16-bit integers. The destination format exists as a one dimensional array of typed data suitable for rendering or 3D interrogation.

The Raw Depth Data
Resolution will change from device to device, but the first generation Creative Interactive Gesture camera offers raw depth data at 320x240 with 16-bit integers used to store a value that represents the distance of any object from the camera at that pixel. By initializing the camera’s hardware and requesting a depth data stream, you can pull this data into your application at a default speed of 30 frames per second.

The raw data can be accessed once the stream has synchronized with your application, and the easiest and quickest way to extract the depth values is to use a nested for loop and run a scan line through the data until you have read each pixel.

The Destination Array

To retain high performance in your applications, you want to avoid repeated scanning of the depth data, and so preparing your destination array before you begin scanning the depth data is best approach. To further improve performance, we will also ensure that our array can be used directly for rendering, so the data structure will be large enough to store the vertex position, vertex normal, and vertex UV coordinates.

In DirectX*, the term FVF (flexible vertex format) is used to describe the structure of a point in 3D space, which contains more than special information. It also stores which direction the vertex faces and what color the vertex might have. As this subject exceeds the scope of this article, it is assumed that you are familiar with how a 3D model is stored and rendered.

For our technique, our FVF structure consists of three 32-bit floats for position (XYZ), three 32-bit floats for normal/direction (NXNYNX), and two 32-bit floats for the texture coordinates (UV). Note the location of the U and V values as we will be tackling this later. With each vertex taking 8 floats, we then multiply that with the resolution of the depth data to produce the final size of our destination array.

The Process

Once the source and destination are prepared, the process itself is relatively simple. Each 16-bit integer is read in at specific X and Y coordinates from the depth resolution. The depth value itself represents distance, which in this case we will convert to a new value we will call Z. We can now feed our destination array with the X, Y, and Z values that represent a position in 3D space. By doing this for each pixel in the depth data, we produce a 3D matrix of X,Y,Z values that is 320 vertices wide, 240 vertices high, and an arbitrary range along the depth axis.

At this point you have un-textured 3D mesh data as a real-time representation of the depth data streaming from the camera.

From here, you can take one of two roads. You can work with the 3D data in its present form, perhaps converting it to a point cloud for further 3D analysis. The second road, and the one we’ll be taking, is to use the data for direct rendering to the screen.

Texture Considerations

In addition to translating the depth information to X,Y,Z values, you need to translate the X and Y values once more into UV coordinates. As we want to texture our 3D model, each vertex needs to know which point it represents on the texture image we are using.
The calculation simply involves dividing the X value by the overall width of the depth data and storing the result as the U value. Similarly, dividing the Y value by the height of the depth data will produce the V value required by the vertex format. The UV coordinates within each vertex should now be float values between zero and one.

**Normal Considerations**

Were you to render the 3D mesh at this stage, you would be horrified to find that although you can see the depth data portrayed in 3D and the texture image spread evenly across the mesh, your lighting would not work. For the mesh to be lit correctly, the normal values (NXNYN) also need to be populated.

As a refresher, the normal vector needs to point away from the position vertex in such a way that when a light position is specified the correct light attenuation can be determined, and the only way to calculate this normal vector is to know the position of the vertex neighbors. As this information is unknown during the primary process, this needs to be applied to the mesh after all positions have been written.

In a second pass, we step through each vertex within the mesh and based on neighboring vertex positions, work out the ideal smoothing normal for that vertex. Once complete, the 3D mesh will be ready for rendering against a light source.

At this point, our 3D mesh has now been populated with position, texture, and lighting information and represents a real-time shape of the depth data coming from the camera. Further techniques can be applied to either the extraction loop or the normal processing loop to further refine the desired rendering. Some concepts include processing the data into separate meshes, or eliminating the background pixels from the mesh.

**4. Tricks and Tips**

**DOs**

Keep the number of passes through the data to a minimum. Given the amount of data produced by the depth data and generated by the 3D mesh data, multiple traversals through this data will impact performance of your application.

Remember to free up all memory usage and interfaces at the proper stage in your application. A good way to ensure this is to add the termination or release code to your app the moment you finish adding the creation half. The consequences are not so bad if your application is simply terminating, but perceptual computing apps often de-activate then re-activate the camera several times during the lifecycle of your application and ensuring your program remains leak free will reduce support incidents later.

When capturing the depth data, ensure you activate depth smoothing in the driver as this will help calm the erratic values you might observe at the very edges of your 3D representation. This is caused by the infrared beam from the camera scattering, creating confusion in the depth data as to whether the object is near or far. It might also be worthwhile experimenting with edge detection to help clarify the true edge of the foreground object.

**Don’ts**
Do not try to set the frame rate properties of the color and depth streams to different values as the driver will not permit this. If you only need your color data (which has a larger 640-pixel wide resolution) every 30 fps, but you need your 3D model to update at 60 fps, then you will need to create two instances of the device so they can synchronize at different rates.

Do not use a 16-bit index buffer when rendering a 320x240 3D model as it will not fit. Use a 32-bit index buffer or a vertex-only buffer so you can render the whole mesh.

Do not share vertex positions if you later want to separate out various features from within the 3D mesh. Your 3D mesh must be made up of truly independent polygons so that you can separate from any background meshes in real time.

5. Code Overview

The following snippets highlight the key elements of the technique.

```cpp
pxcStatus sts=PXCSession_Create(&session);
UtilCmdLine cmdl(session);

At this point we have created a session, which we need before we can start to capture camera data.

pcapture = new UtilCapture(session);
for (std::list<PXCSizeU32>::iterator itr=cmdl.m_csize.begin();itr!=cmdl.m_csize.end();itr++)
    pcapture->SetFilter(PXCImage::IMAGE_TYPE_COLOR,*itr);
if (cmdl.m_sdname) pcapture->SetFilter(cmdl.m_sdname);

We have now created a Capture interface we will be using later.

memset(&request, 0, sizeof(request));
request.streams[0].format=PXCImage::COLOR_FORMAT_RGB32;
request.streams[1].format=PXCImage::COLOR_FORMAT_DEPTH;
sts = pcapture->LocateStreams (&request);

We request a color and a depth stream from the Capture interface.

pcapture->QueryDevice()->SetProperty(PXCCapture::Device::PROPERTY_DEPTH_SMOOTHERING,1);
pcapture->QueryVideoStream(0)->QueryProfile(&pcolor);
pcapture->QueryVideoStream(1)->QueryProfile(&pdepth);
swprintf_s(line,sizeof(line)/sizeof(pxcCHAR),L"Depth %dx%d", pdepth.imageInfo.width, pdepth.imageInfo.height);
pdepth_render = new UtilRender(line);
swprintf_s(line,sizeof(line)/sizeof(pxcCHAR),L"UV %dx%d", pcolor.imageInfo.width, pcolor.imageInfo.height);
puv_render = new UtilRender(line);
sts=pcapture->QueryDevice()->QueryPropertyAsUID(PXCCapture::Device::PROPERTY_PROJECTION_SERIALIZABLE,&prj_value);
pcapture->QueryDevice()->QueryProperty(PXCCapture::Device::PROPERTY_DEPTH_LOW_CONFIDENCE_VALUE,&dvalues[0]);
pcapture->QueryDevice()->QueryProperty(PXCCapture::Device::PROPERTY_DEPTH_SATURATION_VALUE,&dvalues[1]);
session->DynamicCast<PXCMetadata>()->CreateSerializable<PXCPayment>(prj_value, &projection);
```

The above code prepares pdepth_render and puv_render, which are used later to read the streams. This code was taken directly from a working prototype, with only the error trapping removed for easier reading. Follow the above sequence and you should have trouble-free initialization of your camera device.

The above sample can be found in the Intel Perceptual Computing SDK in the sample called "camera-uvmap," which is a very good stripped-down example of how to get up and running very quickly.
Let's have a look at the data structure for our destination array:

```c
struct sVertexType {
    float fX;
    float fY;
    float fZ;
    float fNX;
    float fNY;
    float fNZ;
    float fU;
    float fV;
}
sVertexType * pVertexMem = new sVertexType[320*240];
```

Here we see the data structure we are using for our 3D mesh data, so that we can render, texture, and light the final 3D model.

Now let's look at the processing loop:

```c
Int n=0;
for ( int y=0; y<240; y++ )
{
    for ( int x=0; x<320; x++ )
    {
        float fX = (float)x;
        float fY = (float)y;
        float fZ = 0.0f;
        float fU = 0.0f;
        float fV = 0.0f;
        pxcU16 depthvalue = ((pxcU16*)ddepth.planes[0])[y*pdepth.imageInfo.width+x];
        if ( depthvalue>10 && depthvalue<1500 )
        {
            fZ = (depthvalue-10) /10.0f;
            fU = fX / 320.0f;
            fV = fY / 240.0f;
            pVertexMem[n].fX = fX;
            pVertexMem[n].fY = fY;
            pVertexMem[n].fZ = fZ;
            pVertexMem[n].fU = fU;
            pVertexMem[n].fV = fV;
            n++;
        }
    }
}
```

As you can see, a simple nested loop can be used to traverse all the depth data pixels and translate them into 3D XYZ coordinates. We are also converting the XY to UV coordinates too, so that our 3D model can have a texture when rendered.

And finally, for those of you who want to get the camera color stream into DirectX so your 3D model can have a texture:

```c
LPDIRECT3DTEXTURE9 lpTexture = pMesh->pTextures[0].pTexturesRef;
if ( lpTexture )
{
    D3DLOCKED_RECT d3dlock;
    DWORD bitdepth = 32/8;
    RECT rc = { 0, 0, 320, 240 };
    if(SUCCEEDED(lpTexture->LockRect ( 0, &d3dlock, &rc, 0 )) )
    {
```
This code sample might be quite a lot to take in, but it breaks down quite easily. The first few lines simply locate a texture surface you will have created earlier during the initialization of your application. You then lock the surface so you can write to it safely, and then create a nested loop that will go through every pixel of the depth data. Notice we don’t say color data and also notice the nest loop iterations are for 320x240, and not 640x480, which is the resolution of the color data.

Once inside the inner nest, three things are happening. The first step is that COLX and COLY are filled with the reference coordinate within the color data of the pixel belonging to the depth data coordinate specified by X and Y loop nest variables. Remember that we are texturing a mesh 320 vertices wide, so we don’t need all 640 color pixels from the camera! We do this by using the UVMAP array that will have been read earlier in the code when the stream had finished synchronizing. For more information on this, checkout the ‘camera_unmap’ sample in the Intel SDK.

Using the reference coordinates in the color data, we can read the pixel color, apply a solid white alpha color using (255<<24), and then write the final color to the locked texture. The final step is to advance the write pointer and wait until the nest is complete. We free the texture surface by unlocking it, and the next time you use the texture surface, you will find the latest camera color data present and ready for rendering.

6. Technique Gallery

Here are some screen shots taken from a prototype during the course of development that employed this technique as part of a teleconferencing app.
Mesh and Lighting: With vertex position and lighting in place, you can see how the depth data was represented as a 3D mesh. The color stream from the camera was not used at this early stage.

The Sun: While working in the early hours, you might find that the sun’s rays disintegrate your 3D data. This is the result of the infrared from the sun confusing the infrared detector in the camera.

Too Few Polygons: This early attempt at texturing the model shows that by reducing the number of polygons to represent the depth data can be unsightly.
Too Many Polygons: Conversely, too many polygons is wasteful and performance intensive, and as you can see from the shot above it does not necessarily produce the best visual result. To appreciate how dense the 3D mesh was, an alternative technique was implemented by including a diffuse component in the Vertex Format and the color of the vertex was determined by looking up the camera color pixel based on the UV map provided with the depth data stream.

Final Render: This final shot shows the 3D model wrapped with a texture taken directly from the camera color stream and applied to the render. The mesh was stitched together and normal vectors reset to allow the camera color data to render cleanly. You will notice that despite subtle inaccuracies between depth and color coordinates, the final model contains sufficient detail to create the impression you’re being projected into a virtual conference.

7. About The Author

When not writing articles, Lee Bamber is the CEO of The Game Creators (http://www.thegamecreators.com), a British company that specializes in the development and distribution of game creation tools. Established over 13 years ago, the company and surrounding community of game makers are responsible for many popular brands including Dark Basic, FPS Creator, and most recently App Game Kit (AGK).
The application that inspired this article and the blog that tracked its seven week development can be found here: http://ultimatecoderchallenge.blogspot.co.uk/2013/02/lee-going-perceptual-part-one.html

Lee also chronicles his daily life as a coder, complete with screen shots and the occasional video here: http://fpscreloaded.blogspot.co.uk
Notices

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 go to: http://www.intel.com/design/literature.htm

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark* and MobileMark*, are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

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.