TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1 Per-Pixel Calibration and 3D Reconstruction on GPU

1.1 *RGBD* to $X^C Y^C Z^C RGB$

As described in chapter **??**, there are applications like SLAM and KinectFusion that require D to be converted into $X^C Y^C Z^C$ coordinates on a per-pixel basis. From chapter **??**, we know that the depth sensor measures Z^C, and a pinhole camera model (more specifically the intrinsic matrix) in homogeneous coordinates offers the relationship between Z^C and X^C/Y^C respectively. In this section, we will introduce how to generate the camera space 3D coordinates without calibration, and draw a colored camera space 3D reconstruction on GPU using a KinectV2 camera.

The KinectV2 depth sensor measures Z^C in millimeter and supports its positive data in unsigned-short data-type. Those data will be automatically converted into single-floating type with its range from 0.0f to +1.0f when uploaded onto GPU. Considering that in practical Ds from KinectV2 are always positive whereas Z^Cs should be always negative, we will add a negative sign in the un-scaling step to recover the Z^C in metric on GPU:

$$Z^C[m,n] = -\beta D[m,n], \tag{1.1}$$

where β constantly equals to 65535.0 (range of unsigned short in single-floating) for all pixels. Besides the depth stream, KinectV2 supports both of the horizontal and vertical field of view (FoV) that can help generate per-pixel X^C and Y^C, which share the same credits with the intrinsic parameters in a pinhole camera model. Figure 1.1 shows an intuitive

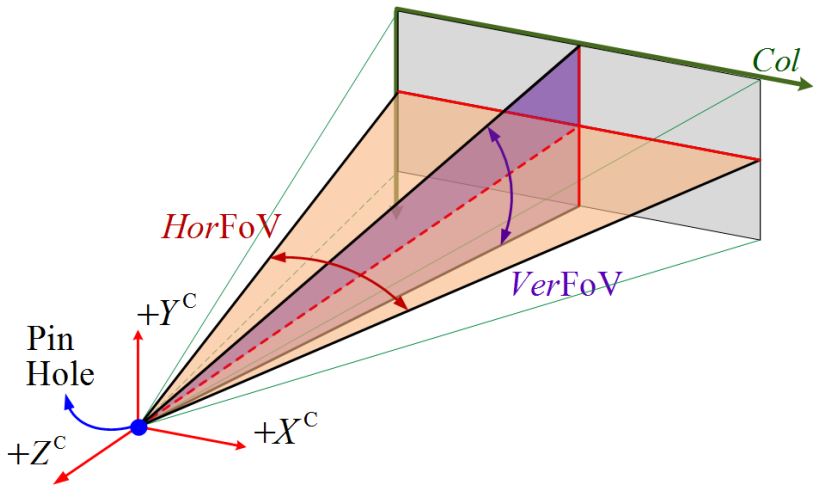


Figure 1.1: Field of View in Pinhole-Camera Model

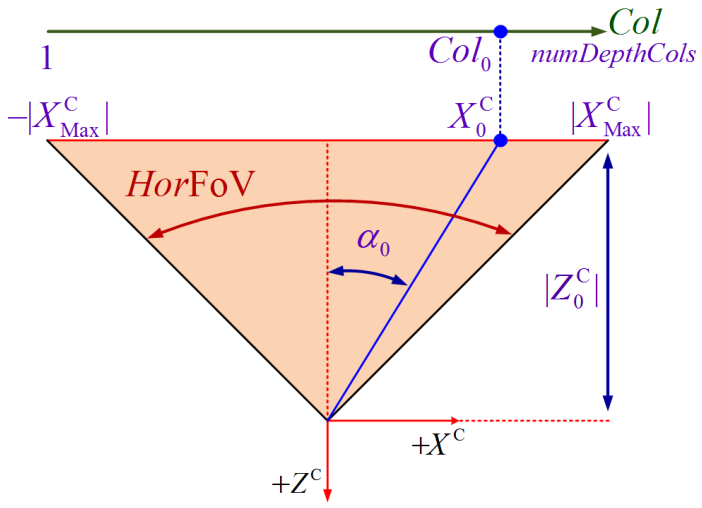


Figure 1.2: map Col to X^C via horizontal FoV

view of the horizontal and vertical FoVs in a pinhole camera model, based on which we can derive the X^C and Y^C values given a random Z^C value on the per-pixels basis. Assuming the depth sensor, with its size numDepthRows by numDepthCols, is observing right perpendicularly to a wall where all pixels share the same $|Z^C_0|$ from the sensor to the wall. Talking about the horizontal FoV only, it is easy to get the range of the camera space FoV along X^C-axis from $-|X^C_{\text{Max}}|$ to $|X^C_{\text{Max}}|$, as shown in fig. 1.2. The horizontal range value $|X^C_{\text{Max}}|$ depends on $|Z^C_0|$ and the horizontal FoV, given by:

$$|X^C_{\text{Max}}| = |Z^C_0| \cdot tan(horFov/2) \tag{1.2}$$

where the pixel observing on $X^C = |X^C_{\text{Max}}|$ has its column address of numDepthCols. Similarly, given a random pixel of column address Col_0, its horizontal view X^C_0 could be expressed based on its own horizontal view angle α_0.

$$|X^C_0| = |Z^C_0| \cdot tan(\alpha_0) \tag{1.3}$$

To combine eqn. 1.2 and eqn. (1.4), we get

$$\frac{X^C_0}{|X^C_{\text{Max}}|} = \frac{tan(\alpha_0)}{tan(horFov/2)} = \frac{Col_0}{numDepthCols} - 0.5\,, \tag{1.4}$$

which shows how to get the per-pixel X^C_0 from $|X^C_{\text{Max}}|$ based on its column address, while $|X^C_{\text{Max}}|$ depends on the depth sensor's horizontal filed of view. It is intuitively better to change eqn. (1.4) a little by substituting $|X^C_{\text{Max}}|$ with eqn. (1.2) such that we can get eqn. (1.5),

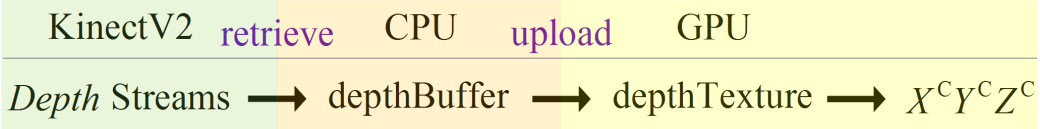


Figure 1.3: Diagram for Camera Space 3D Reconstruction without calibration

a proportional per-pixel mapping based on the column addresses from Z^C to X^C.

$$X^C[m,n] = tan(horFov/2) \cdot (\frac{n}{numDepthCols} - 0.5) \cdot |Z^C[m,n]|\,, \tag{1.5}$$

where $[m,n]$ is the discrete space *row* and *column* coordinate of each pixel in the depth sensor. Similarly, we can also get the proportional per-pixel mapping from Z^C to Y^C, based on the vertical FoV and row addresses.

$$Y^C[m,n] = tan(verFov/2) \cdot (\frac{m}{numDepthRows} - 0.5) \cdot |Z^C[m,n]| \tag{1.6}$$

Note that, *horFov* and *verFov* are constant during image processing, such that the mapping functions from per-pixel Z^C to per-pixel X^C/Y^C totally depend on a pixel's address $[m,n]$. Therefore eqn. (1.5) and eqn. (1.6) could be expressed as

$$\begin{aligned} X^C[m,n] &= a[m,n] \cdot |Z^C[m,n]| \\ Y^C[m,n] &= b[m,n] \cdot |Z^C[m,n]| \end{aligned} \tag{1.7}$$

where

$$\begin{aligned} a[m,n] &= tan(horFov/2) \cdot (\frac{n}{numDepthCols} - 0.5) \\ b[m,n] &= tan(verFov/2) \cdot (\frac{m}{numDepthRows} - 0.5)\,. \end{aligned} \tag{1.8}$$

Now that we have the per-pixel mapping from Z^C to $X^C Y^C$, it is time to draw the colored camera space 3D image. Figure 1.3 shows the diagram of drawing using GPU. We will retrieve *Depth* and *RGB* streams from the KinectV2 camera, and save them into corresponding buffers respectively. Then those three streams will be uploaded from CPU onto GPU as textures for fragment shader processing. The camera space 3D coordinates $X^C Y^C Z^C$ will be generated from depth texture based on eqn. (1.7), while *RGB* values can be looked up from the color texture. Six channels of data $X^C Y^C Z^C RGB$ for all pixels will be saved into a framebuffer object, whose texture can be utilized for looking up during the final 3D image drawing. The fragment shader is programmed as below.

```
uniform  sampler2D qt_depthTexture;
```

```
uniform  sampler2D qt_spherTexture;

layout(location  =0,index  =0)out  vec4 qt_fragColor;
void main()
{
    ivec2 textureCoordinate=ivec2( gl_FragCoord.x, gl_FragCoord.y);

    floatz = texelFetch(qt_depthTexture,textureCoordinate,0).r *65535.0;
    vec4 a = texelFetch(qt_spherTexture,textureCoordinate,0);

    qt_fragColor.x = -z*a.r;
    qt_fragColor.y = -z*a.g;
    qt_fragColor.z = -z;
}
```

This fragment shader handles both of $X^C Y^C Z^C$ generation and *RGB* texel fetch. There are four uniform input textures uploaded on GPU. The *qt_colorTexture* is RGB stream, and *qt_depthTexture* is depth stream. *qt_mappingTexture* is a mapping stream also given by KinectV2 camera, which contains two channels Row/Col mapping data. It offers a link between RGB stream and Depth stream, helping align RGB values to depth pixels. *qt_spherTexture* is a Z^C to X^C/Y^C proportional mapping texture, which contains the per-pixel parameters a/b based on eqn. (1.8). Let's view the fragment shader step by step, and then have a view of the 3D reconstruction in camera space.

We will first get the texture coordinates from the fragment coordinates.

```
ivec2 textureCoordinate=ivec2(gl_FragCoord.x/2,gl_FragCoord.y);
```

And then have a check if this fragment shader should be assigned for $X^C Y^C Z^C$ generation, or *RGB* texel fetch.

```
if  (int(gl_FragCoord.x)%2 ==0){
    // X^C Y^C Z^C generation
}else {
    // RGB texel fetch
}
```

The depth steam measures Z^C, which is always negative and will be rescaled from D.

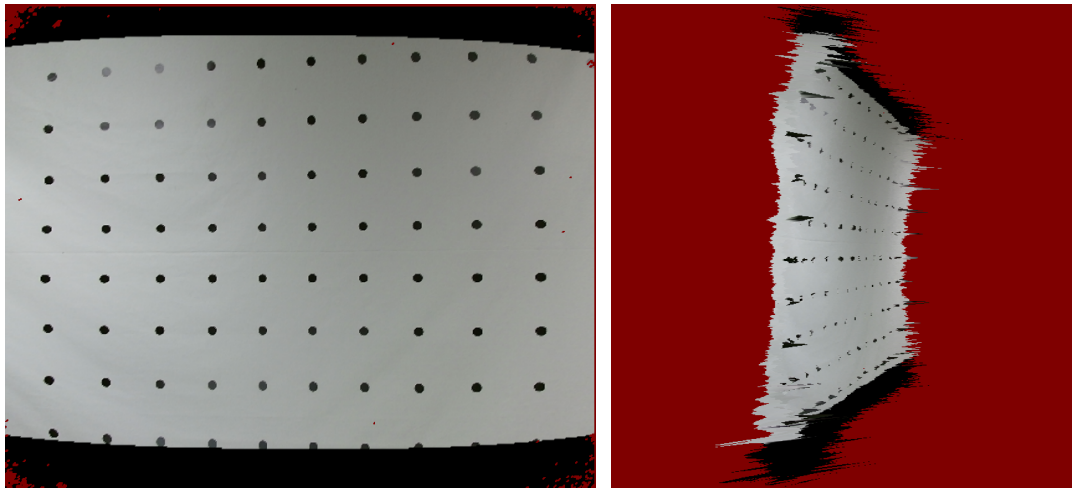
```
int col=textureCoordinate.x/4;
int chn=textureCoordinate.x%4;
float z=texelFetch(qt_depthTexture
```

```
                ,ivec2(col,textureCoordinate.y),0)[chn]*65535.0;
    qt_fragColor.z=-z;
```

The per-pixel X^C/Y^C can be generated through a proportional mapping from Z^C, as discussed in eqn. (1.7). And the parameters the per-pixel parameters a/b can be acquired from *qt_spherTexture*.

```
    vec4  a=texelFetch(qt_spherTexture,textureCoordinate,0);
    qt_fragColor.x=-z*a.r;
    qt_fragColor.y=-z*a.g;
```

The whole parallel image processing above shows a natural 3D reconstruction on GPU. With the per-pixel $X^C Y^C Z^C$ coordinates generated and saved together with corresponding RGB values, we are ready to draw the 3D image. Figure 1.4 shows the view of colored camera space 3D reconstruction when observing uniform grid dots pattern on the flat wall. We can tell from the image that, this reconstruction is totally based on raw data, without calibration at all. Not only the radial dominated lens distortions are apparent in the front view, but the *depth distortion* is also material that cannot be ignored. Therefore, calibration must be done for an undistorted 3D image.



(a) Front View (b) Left View

Figure 1.4: Colored Camera Space 3D Reconstruction

1.2 Rail Calibration System

Talking about camera calibration, the pinhole camera matrix M will come up in most people's mind. As discussed in chapter **??**, the pinhole-camera matrix M consists of an intrinsic matrix K and an extrinsic matrix $[R_{3*3}, T_{3*1}]$. The camera space 3D reconstruction method we discussed in section 1.1 utilizes horizontal and vertical field of view (FoV)s, which works in the same way with the intrinsic matrix K's principle. However, a camera's calibration needs external help from world space objects, which means neither of the FoVs and intrinsic matrix K alone are able to do calibration, and extrinsic parameters that can link to world space are necessary in calibration.

Kai [1] did a good job on structed light 3D scanner parallel calibration on GPU, and derived the per-pixel beam equation (1.9) directly from a pinhole camera matrix M, which offers the possibility of natural 3D reconstruction on GPU similar to eqn. (1.7).

$$\begin{aligned} X^W[m,n] &= a[m,n]Z^W[m,n] + b[m,n] \\ Y^W[m,n] &= c[m,n]Z^W[m,n] + d[m,n] \end{aligned} \tag{1.9}$$

where $[m,n]$ is the discrete space *row* and *column* coordinate of each pixel in a M by N sensor. This per-pixel beam equation shows per-pixel linear mappings from Z^W to X^W/Y^W, and the per-pixel Z^W can be mapped from features of structured light. It is not specially proportional like eqn. (1.7), because it contains space translation infos from camera space to world space. Although it is in world space now and the intrinsic parameters are able to be determined, however, lens distortions are still not able to be handled. To make it ideal, we need to not only remove the lens distortions and *depth distortion*, but also realize the 3D reconstruction on GPU in a natural method similar to eqn. (1.9).

In this section, we will find a best-fit calibration system for a KinectV2 camera's natural calibration and reconstruction, which is able the handle both of lens distortion and *depth distortion*. To easily show 3D reconstruction in a parallel way on the GPU, we would like our calibration system to be able to offer a per-pixel mapping from D to Z^W, which then could be used to map to X^W/Y^W using eqn. (1.9). In this way, the *depth distortion* could also be corrected during the per-pixel D to Z^W mapping. Of all different kinds of calibration systems, a camera on rail system with a planar pattern on wall is finally decided, which offers a moving plane with respect to the camera when the camera moves along the rail. As fig. 1.5 shows, a canvas on which printed an uniform grid dots pattern is hung on the wall, and the rail is required to be perpendicular to the wall. The RGB-D camera waiting to calibrate is mounted on the slider. Note that, in this calibration system, the only unit that needs to be perpendicular to the wall is the rail, whereas the RGB-D camera has no

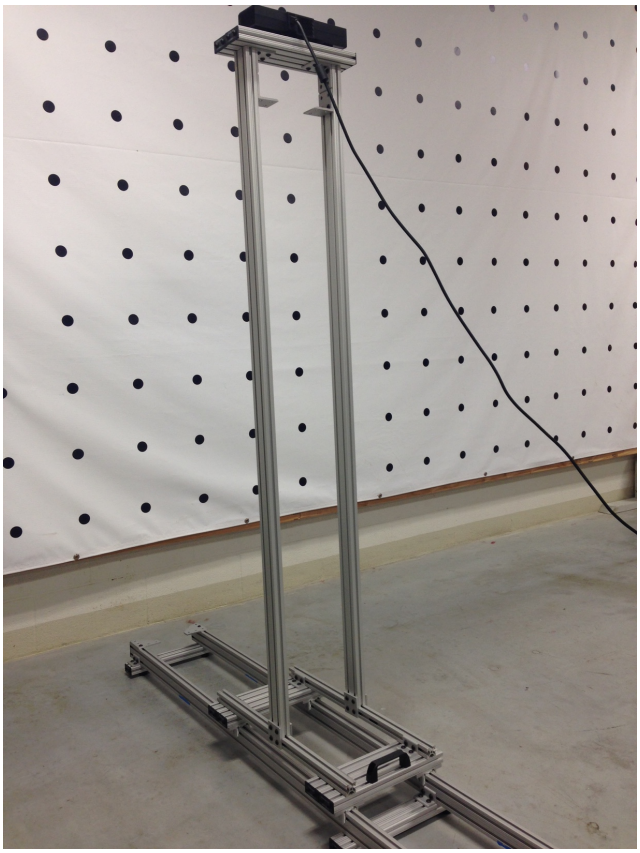


Figure 1.5: KinectV2 Calibration System

need to require its observation orientation. Because the per-pixel calibration requires only accurate world space coordinates which will be decided by the rail and the wall, whereas the camera's space is not considered at all. We will assign the pattern plane as the $X^W Y^W$ plane in world space, and the rail to be along with (not exactly on) Z^W-axis. The world coordinate is static with the camera on the slider.

Figure 1.6 shows one frame of NearIR $X^W Y^W Z^W$ 3D reconstruction, which can help a lot explaining how the world space origin is assigned. Inside the figure, both of the origin and Z-axis are high-lighted in blue, and the origin of world space is on the left end of the blue line. On the pattern plane with dot-clusters marked in circles, we can see one dot-cluster is high-lighted inside a thick circle, and its center point is where $X^W/Y^W = 0$. The dot-cluster which will be sitting on the Z^W-axis is the one whose center point is closest to the center pixel of the sensor. All pixels in this frame share the exact same Z^W, which is also why we require the rail to be perpendicular to the pattern. And the value of Z^W is measured by a laser distance measurer that static with the camera. The final origin of the world space will be decided by both of the camera's observing orientation and the laser distance measurer's position. This kind of world coordinate assignment is totally for

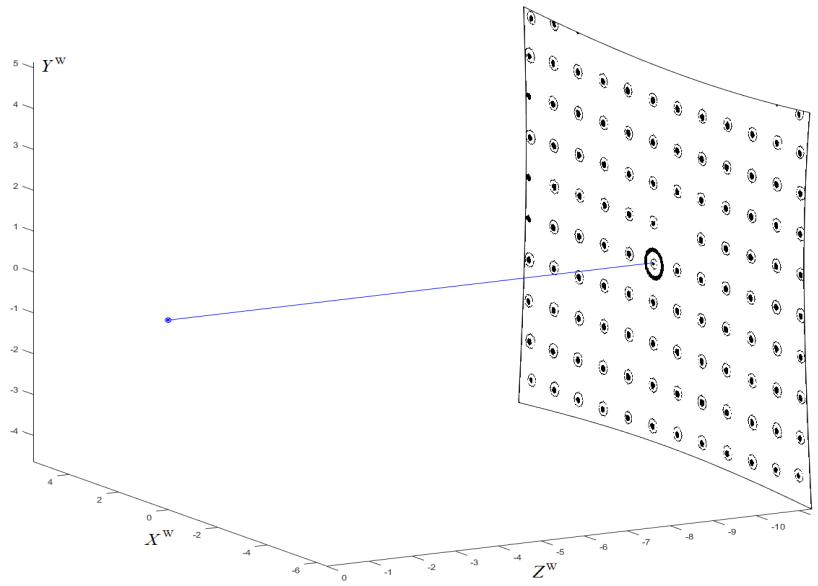


Figure 1.6: NearIR $X^W Y^W Z^W$ 3D Reconstruction

simplifying image processing during calibration. Practically, we do not even care where exactly the origin is, as long as the rail is perpendicular to the pattern and the distance measurer is static with the camera.

With this rail calibration system, infinite number of frames with infinite number of calibration points could be utilized for training a calibration model. Besides, as the slider moves along the rail, the amount and distribution of the grid dots captured by the camera will change, which means a dynamic pattern for calibration instead of static. With more and more dots walking into the camera's field of view under a certain rhythm as the slider moves further from the pattern plane, the dots (which will be extracted as calibration points) are able to cover all pixels of a sensor. What's more, a moving-plane system (multiple frames calibration instead of one frame calibration) makes it possible to do dense D to Z^W mapping, which will handle *depth distortion*.

1.3 Calibration Procedures

With the calibration system built up and world coordinate assigned, we are now ready to calibrate. Z^W values for all pixels of every frame will be supported from external laser distance measurer. To simplify potential calculation during image processing, we assign the world coordinate *Unit One* based on the uniform grid dots patter, to be same with the side of pattern's unit-square. Concretely, the distance between every two adjacent dots' centers in real-world is 228mm. Therefore, Z^W = -Z(mm) / 228(mm), where Z is the vertical distance to the pattern plane in reality measured by the laser distance measurer.

Note that, Z^W values are always negative, based on the assignment of Cartesian world coordinate. The outline of calibration procedures is listed below.

1. Mount both of the camera and laser distance measurer onto the slider.
2. Move the slider to the nearest position to the pattern plane.
3. Record one frame of RGB data and one frame of NearIR data at this position.
    a) measure $|Z^W|$ using the laser distance measurer.
    b) grab RGB, NearIR and Depth streams from KinectV2 camera.
    c) extract center points (R/C) of dot-clusters from RGB and NearIR streams respectively.
    d) assign X^W/Y^W values to the extracted points, RGB and NearIR respectively.
    e) train and determine the best-fit high order polynomial model that map from RC to X^W/Y^W, RGB and NearIR respectively.
    f) generate dense X^W/Y^W for all pixels using the model, for NearIR and RGB streams respectively.
    g) save 2 frames of RGB and NearIR all pixels' data respectively: $X^W Y^W Z^W RGBD$ for RGB stream, and $X^W Y^W Z^W ID$ for NearIR stream, where the channel I in NearIR frame denotes *Intensity*.
4. Move the slider to the next position, and repeat step 3.
5. Check all X^W/Y^W values of all frames, unify the staggered origin by adding or subtracting a corresponding integer (make sure the point $X^W Y^W = 0$ is at the same dot-cluster for all frames).
6. Determine the per-pixel mapping from D to Z^W and from Z^W to X^W/Y^W for all pixels using the NearIR frames of data, and generate a *row*-by-*column*-by-6 look-up table.
7. Determine a pinhole-camera matrix that can map $X^W/Y^W/Z^W$ to R/C using RGB frames of data (for the RGB values' alignment to pixels after 3D reconstruction).

Bibliography

[1] Kai Liu. *Real-time 3-D Reconstruction by Means of Structured Light Illumination*. PhD thesis, University of Kentucky, 2010.