3D Scanner

Jessy Lin & Evan Tey
lnj@mit.edu, tey@mit.edu

6.111 Fall 2016 Final Project

Contents

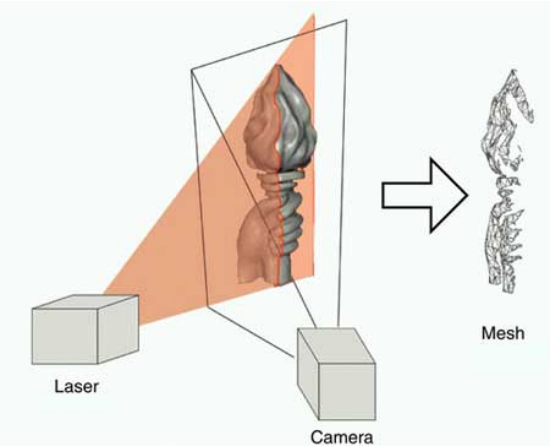

Figure 1: Laser scanning setup

1 Introduction

The demand for fast and cheap 3D scanning has skyrocketed since 3D printing was popularized recently on the mass market. Previously an expensive process reserved for specialized applications such as animation or industrial design, 3D scanning has expanded to day-to-day purposes such as scanning objects, people, and scenes so they can be manipulated virtually before re-printing into new models.

Potential implementations include stereoscopic imaging, photogrammetry, and triangulation from laser profiles, to name a few. The core technical challenge for most of these 3D scanners lies in reconstructing 3D data from typical 2D data, such as images from conventional cameras. This task is both complex and computationally intensive.

In this project, we implement a fast, hardware-based version of a 3D laser scanner, using triangulation to reconstruct a point cloud for the object given the parameters for the physical setup and images from a NTSC camera. The 3D point cloud will be rendered on the display in real-time.

All the code for this project can be found in our Github repository.

2 Technical Background

2.1 Depth Reconstruction

A 3D laser scanner projects a thin laser line on to the object to be scanned, revealing an object profile that is captured by the camera (Figure 1). The laser and camera are positioned at an angle to one another. Knowing the location of the laser and certain properties of the camera, by looking at the laser profile we

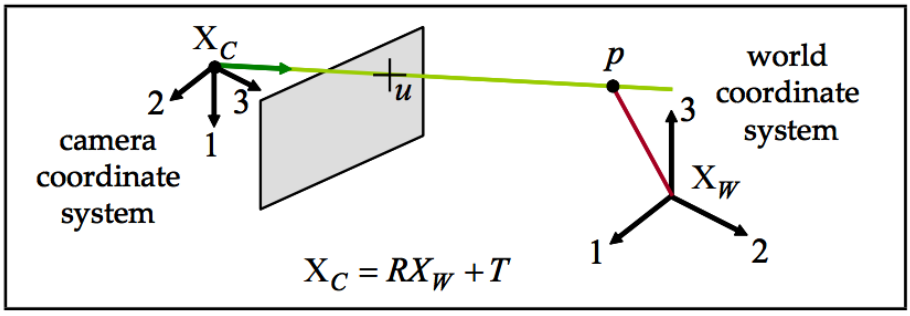

Figure 2: Transformation from camera to world coordinates. Figure taken from [1].

can calculate the corresponding points in 3D space that are illuminated. This section will describe how to perform this transformation.

Given a pixel in the processed image, we translate its image coordinates to world coordinates by applying the transformation between the camera and world coordinate systems (Figure 2). We can express this in terms of the extrinsic parameters of the camera, position and orientation, as represented by a translation vector T and a rotation matrix R with respect to the origin of the world coordinate system. If p_C, p_W are the 3D vectors representing the point in camera and world coordinates, respectively, then

$$\vec{p_C} = R \cdot \vec{p_W} + \vec{T}$$

.

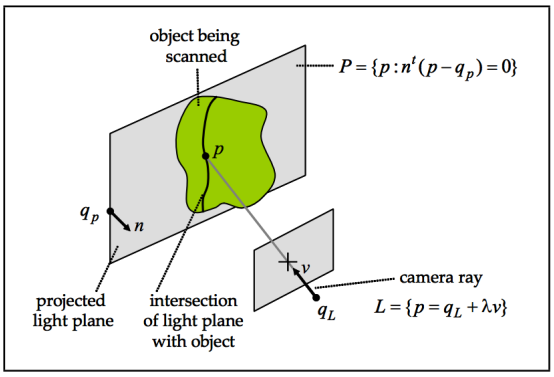

Figure 3: Finding the intersection of the laser plane and the camera. Figure taken from [1].

We model the setup as an intersection between the plane of laser light and the camera ray to the projected laser line (the object's profile) to calculate p_C (Figure 3).

Thus, we can do some matrix math convert given 2D pixel coordinates (x, y), we can convert them to corresponding 3D world coordinates $(x_{world}, y_{world}, z_{world})$.

First, we note that we need to correct for the intrinsic parameters of the camera (e.g. focal length) at some point during the conversion to pixels to real-world coordinates. The intrinsic parameters of the camera can be determined with a one-time calibration procedure and represented in the form of an intrinsic matrix:

$$KK = \begin{bmatrix} fc_1 & \alpha_c * fc_1 & cc_1 \\ 0 & fc_2 & cc_2 \\ 0 & 0 & 1 \end{bmatrix}$$

where:

- fc_1, fc_2 are the focal lengths in x and y pixels respectively. While the physical focal length of the camera is fixed, the focal length in x and y pixels may be different if we have rectangular pixels.
- α_c is the skew coefficient (the angle between the x and y pixels; nonzero if pixels are not perfectly rectangular)
- cc_1, cc_2 are coordinates of the *principal point*, i.e. the projection of the center of the camera on to the image plane (the "true center").

.

The coordinates of the *pixel*, on the image plane, with respect to the origin of camera is:

$$u = \begin{bmatrix} x - cc_1 \\ y - cc_2 \\ 1 \end{bmatrix}$$

where the z dimension arises if we assume a physical focal length of 1.

The coordinates of the object point in camera coordinates lies on the ray from the origin of the camera to p, extended to the object. Thus the 3D object point with respect to the origin of the camera is some multiple of the pixel in camera coordinates, $p_C = \lambda u$.

Now we consider what we know about the laser plane: in our hardware setup, we can set it to pass through the origin of the world coordinate system, for simplicity. Referring to Figure 4, if the laser is at θ degrees to camera, given the world coordinate system, the normal to the laser plane is:

$$n = \begin{bmatrix} 0 \\ a \\ b \end{bmatrix}$$

where $tan(90 - \theta) = \frac{a}{b}$. Since the plane passes through the origin, the parameterized equation of the laser plane is $\{p : n^T p = 0\}$.

We wish to find the unknown λ, so then the point of interest in camera coordinates is λu. Starting from the equation of the laser plane, we know both the intrinsic matrix and the rotation matrix have inverses, so

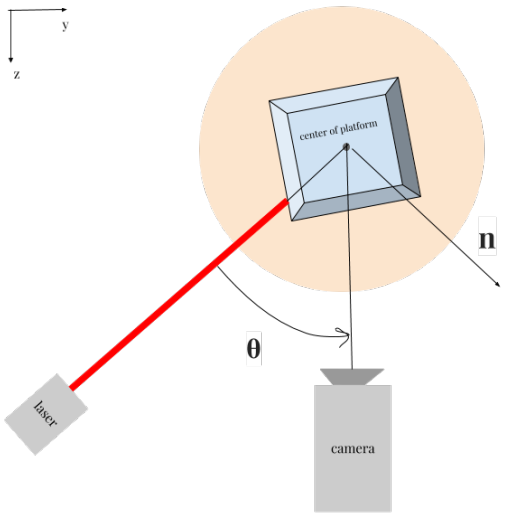

Figure 4: Hardware setup, with θ and n indicated.

$$\begin{aligned} p_c &= K(Rp_W + T) \\ \lambda u &= K(Rp_W + T) \\ R^{-1}(K^{-1}\lambda u - T) &= p_W \end{aligned}$$

Multiplying both sides by n^T and knowing that the point lies on the laser plane with equation $\{p : n^T p = 0\}$,

$$\begin{aligned} n^T R^{-1}(K^{-1}\lambda u - T) &= 0 \\ \lambda &= \frac{n^T R^{-1} T}{n^T R^{-1} K^{-1} u} \end{aligned}$$

Thus the coordinates of the point relative to the camera origin is λu, and relative to the world origin is $\boxed{p_W = R^{-1}(K^{-1}\lambda u - T)}$.

2.2 3-D Rendering

In order to render 3 dimensional points on the monitor, we make a few assumptions. These limit the abilities of the renderer, but are necessary to allow implementation practicality.

First, we assume that we're observing the object from very far away. As we observe any object from farther and farther, we lose perspective skew. So,

for example in Figure 5, if we're looking at a box up close, we can distinguish the top and front faces, but when the box is very far away, we can only see the front edge. This allows us to simply use the flat projection of an object onto a plane to generate 2 dimensional points, then scale that projection so it fits in monitor. (A separate way to think about this is that the monitor is a camera with a focal length of infinity.)

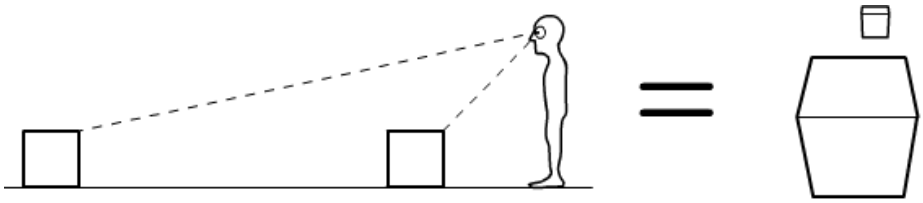

Figure 5: Perspective skew of a box.

Making this assumption allows us to save many trigonometric calculations associated with skew and a focal point, because given (x, y, z) coordinates, we simply have to scale and shift (x, y) until the object can be seen in the monitor.

So

$$(x, y, z) \rightarrow \begin{array}{c} x_{vga} = a * x + x_{offset} \\ y_{vga} = a * y + y_{offset} \end{array}$$

where a is a scaling factor.

Second, we assume lighting always comes from the monitor's direction. This can be thought of as having a flashlight mounted on our pretend monitor-camera pointed directly at the object, so pixels in the front appear brighter than pixels in the back.

This assumption allows us to simply use an appropriately scaled z-value as luminance rather than introducing light ray tracing (this could be an entire project on its own).

So

$$(x, y, z) \rightarrow pixel[7:0] = z[9:2]$$

(where x, y, and z are each 10 signed bits.)

Given all of this, we'd like to rotate around our objects so we can view the point cloud from many angles.

To start with, we need rotations around the y axis – the equivalent of walking around an object to the left or right. Given any point and angle, we can calculate the point's new position after rotation by applying a rotation matrix to it, so

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} cos(\theta) & 0 & -sin(\theta) \\ 0 & 1 & 0 \\ sin(\theta) & 0 & cos(\theta) \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Using the new (x', y', z'), we can project into the xy-plane and use z for luminance. Confining ourselves to a single rotation puts a limit on the calculations required during any clock cycle. This ensures that we can still store the new points in ZBT memory because that process is extremely time-sensitive.

Allowing rotations around the x-axis (looking from the top or bottom of the object) does not require much more mathematical work. It is simply the multiplication of another rotation matrix so

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos(\phi) & -sin(\phi) \\ 0 & sin(\phi) & cos(\phi) \end{pmatrix} \begin{pmatrix} cos(\theta) & 0 & -sin(\theta) \\ 0 & 1 & 0 \\ sin(\theta) & 0 & cos(\theta) \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

3 Design and Dataflow

Altogether, the process for scanning an object is as follows:

1. Place object on rotating platform. The object should be small enough to be contained within the camera image. Additionally, less reflective objects and a dark environment are ideal for accurately capturing the laser profile.
2. Camera data is streamed through the preprocessing stages where noise is filtered out and the laser profile is cleaned to the width of a single pixel.
3. The camera coordinates of each of these pixels is then converted to real world coordinates and stored in ZBT memory.
4. These points are then received from memory and written to a visual RAM (a bitmap representing the monitor.
5. This visual RAM is then read out through the VGA port to the monitor.

3.1 Block diagram

A more detailed diagram of how the scanner will work can be seen in 6.

4 Hardware setup (Jessy)

Our hardware setup consisted of a circular platform mounted on a 28BYJ-48 stepper motor + stepper motor driver (Figure 7). The motor was controlled by an Arduino with the Stepper library and made to rotate 10 degrees (57 steps) every 300 ms. A 65 nm, 5 mW red line laser was mounted at the height of the platform at $\theta = 40$ deg to the camera and passing through the axis of rotation (the center) of the platform. We ensured that the line laser was approximately perpendicular to the platform by placing a paper cup on the platform and checking that the line aligned with the seam of the cup.

The NTSC camera location and height, as well as the focus on the lens, were fixed for calibration. We were able to determine the extrinsic parameters with calibration procedure (described later), and the camera parameters were left constant after calibration.

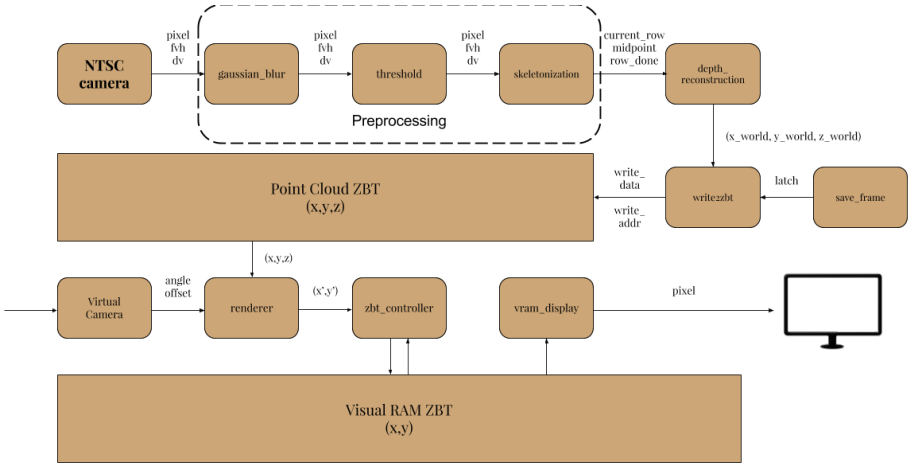

Figure 6: Detailed flowchart of 3D scanner

5 Image processing (Jessy)

The input from the NTSC camera has a resolution of 640 x 840 pixels, with 30 bits of YCbCr color per pixel. The 3D scanner must distinguish between bright pixels of the laser line on a black background, so only the 8 bits of luminance (Y) are necessary for our application.

5.1 Module: *gaussian_blur*

The *gaussian_blur* module averages each pixel with its neighbors to reduce noise from the camera. In our implementation, we stream the bits directly from the camera into the module and implement preprocessing in real time.

Figure 8 illustrates the circuitry we used for preprocessing. When pixels are streamed in from the camera, they are initially passed to the set of 25 registers holding the 25 values currently under the filter. As the filter moves to the right in the image, filter values are shifted to the left. Once the pixel passes $(0, 4)$, it is no longer under the filter and it is passed to a line buffer (which also shifts all its values to the left every clock cycle) of width IMG_WIDTH $- 5$ to store for the remainder of the current line. When the filter starts the beginning of the next line, the pixel at the end of the line buffer will be passed to the second row of the filter. This ensures that the current values in the filter registers are always correct, while maintaining pixels that already came from the camera, are not currently under the filter, but will be used again later as the filter shifts down.

We used 5x5 kernel with standard deviation 2, using coefficient approximations with denominator 1024 from Matt Hollands and Patrick Yang's Fall 2015 Project[2]. On each clock cycle, the values that are currently in the filter register are multiplied by the corresponding coefficient and summed together. The

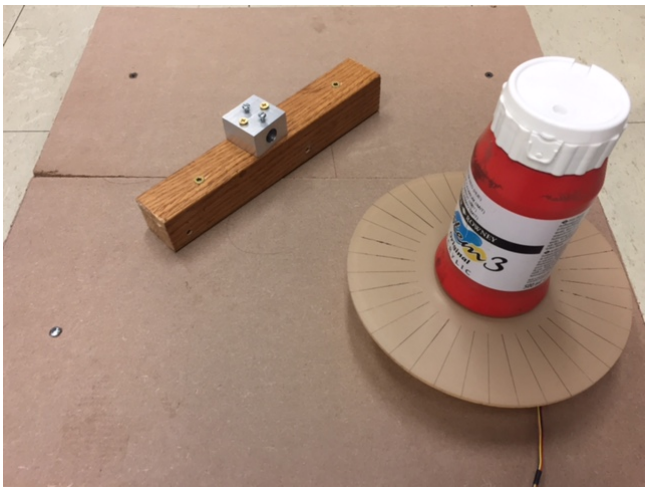

Figure 7: Hardware setup, without the camera

result is the blurred value for the pixel currently at location $(2, 2)$, the center of the filter.

With a 5x5 filter as shown, a latency of $2 * \text{IMG\_WIDTH} + 2$ clock cycles is introduced from when a pixel is first output from the camera to when its blurred value is calculated. This appears to be the minimum possible to achieve for real-time blurring, since it is absolutely necessary to wait for the camera to provide two rows of pixels below the current one in order to calculate the correct blurred value. The throughput is 1 clock cycle; we continuously output pixels at the same rate that the camera provides them.

5.1.1 Challenges

Gaussian filters from typical projects in previous years have calculated blurred values based on pixels stored in memory. However, since our application demanded a fast and real-time solution we strove to implement it with only registers.

Our circuitry relies heavily on knowing the correct image width (since it parameterizes the width of the line buffers). Although the NTSC camera specified 640 pixels per line, through the process of debugging we discovered that this may not be the case for our practical purposes; 2D filtering outputted a duplicated image that suggested our line buffers were not the correct size. We reverted to using repeated applications of a long 1D filter to achieve sufficient noise reduction for our application (viewing a clear laser line). The implementation seems otherwise correct (see GitHub code), and it may be worthwhile for future teams to experiment further with the line buffer lengths to achieve a successful real-time 2D filter implementation.

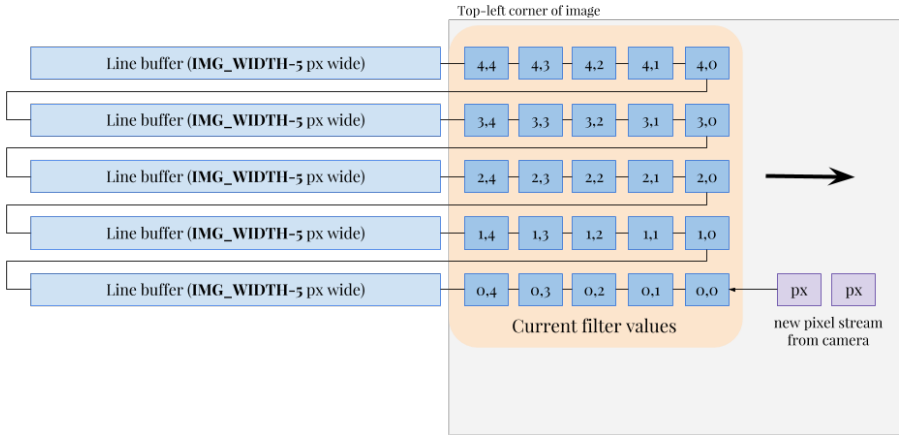

Figure 8: Line buffer for real-time Gaussian blur

5.2 Module: *threshold*

The *threshold* module is used to separate the bright laser line more clearly from a dark background. Given a threshold, the module reassigns white to all the pixels that are above that threshold, and black to all pixels below. Because we found that the clarity of the red laser line sometimes depended on the color of the object itself, we made the threshold adjustable using the labkit buttons, which would allow us to increment or decrement the threshold by 5 as needed and immediately see the results. We chose thresholds that allowed us to see the largest possible laser line without exposing any of the background or introducing excessive noise from the camera.

This module is also real-time, taking in the (delayed) pixel stream from *gaussian_blur* and outputting the updated value one clock cycle later.

5.3 Module: *skeletonize*

The *skeletonize* module is used to thin a thick laser blob into a line, in order to increase the precision of depth reconstruction (which calculates the depth of individual pixels given to it).

At a high level, the skeletonization module takes the pixel stream from threshold as input and keeps track of the longest block of white pixels on the line and its midpoint using a simple running count. At the end of each row, we output row and the midpoint of the longest block on the row. These pixel coordinates will be converted to a corresponding point in 3-space by the *depth_reconstruction* module.Figure 9 shows the effect of skeletonization.

White noise on the black background is not problematic, but the module accounts for potential black noise in the middle of white blocks, which could disrupt the midpoint calculation. The implementation uses an finite state ma-

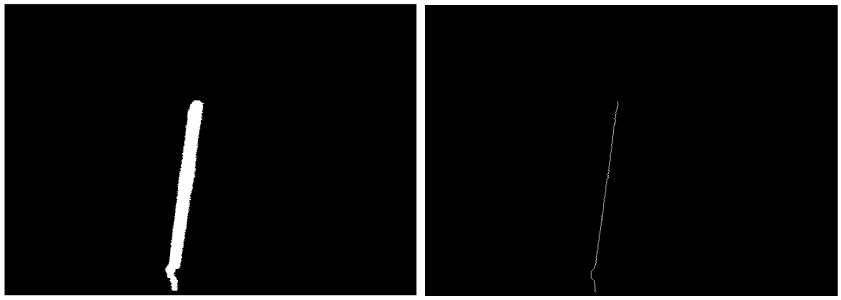

Figure 9: Pre-skeletonization thresholded (left) and post-skeletonization (right) image of a cup

chine with three states: WHITE, NOISE, and BLACK. If we are in the middle of a white block (WHITE) and we encounter a black pixel, we enter the NOISE state and keep a running count of the number of black pixels we have encountered since we entered the state. If the number exceeds 3 (an arbitrary threshold), we end the white pixel count and enter the BLACK state. If the we encounter a white pixel before the noise threshold exceeds 3, we continue the previous running count and fill in the noisy pixels, adding them to the running count.

This module was thoroughly tested in simulation to ensure the timing and values were correct.

5.4 Module: *save_frame (Evan)*

Up until this point, data has been constantly streaming through the pipeline – being blurred, thresholded and skeletonized. We don't want to keep all of this information, however, because some camera frames will contain laser profiles that are somewhere in between a 10 degree rotation and others will simply have redundant information. We just want to keep profiles for every 10 degree mark, which means we need a way to capture frames at a given interval.

To do this, we want to assert a signal for an entire camera frame – at first in response to a button. So once a button is pressed, we wait until the next *vsync* and hold a *latch* signal at 1 until the next *vsync* happens. This *latch* signal can then be used to allow / not allow the further passage of data.

We use it by only allowing the writing of coordinates when *latch* is high. So just before any data is written, we AND *latch* with the write signal (which expresses that coordinates have successfully been calculated and are ready to be written).

Due to troubles with depth reconstruction, *save_frame* was only successfully appended to skeletonization, where *latch* was ANDed with the *row_done* signal from *skeletonize*. This was then written to ZBT memory and rendered for testing.

6 Depth reconstruction(Jessy)

The core part of our project involves calculating the corresponding point in 3D space (a point on the object illuminated by the laser) for a given pixel coordinate. In order to calculate the 3D point properly, we needed to know intrinsic properties of the camera such as focal length, as well as extrinsic properties such as its position and orientation relative to the origin of the world coordinate system. To find these parameters, we transfer images of a standard checkboard from the labkit to a laptop and run a calibration procedure in MATLAB.

6.1 Module: *rs232_transmit* (Evan)

RS-232 is a serial communication standard which we use to transmit data from the FPGA to a computer. We were provided with a simple Verilog file readily sent bytes of data (specified by switch configuration) to a laptop at the press of a button. From the computer side, we used minicom to allow the receiving of serial data. Transmission also required an RS-232 to USB converter so the cable could run from the FPGA to my laptop.

6.1.1 Challenges

The most difficult aspect of setting up working RS-232 transmission was the timing. First, we had to update the baud rate to match the clock we were using. Originally, the file provided a timing setup using a 27 Mhz clock which we had to update for a 40.5 Mhz clock. This just consisted of updating the clock *Divisor* to match the 115200 baud rate.

With this working, we needed to find a way to send a single frame across the transmission. To do this, we froze all NTSC writing to ZBT in response to a button. This button additionally triggered an iteration through all ZBT memory addresses containing pixel data.

At first, we tried a naive approach of simply displaying pixels from all lines of ZBT memory. This, however, only produced static because we couldn't find an appropriate image width to use when displaying the image in Matlab.

After some experimentation, we found that the *vram_display* module shows the monitor only reads out addresses of the form

$$wire[18:0]vram\_addr = 1'b0, vcount_f, hcount_f[9:2];$$

where *hcount* and *vcount* max out at 1048 and 805 respectively and where each address holds 4 pixels (hence ignoring the last two pixels of *hcount*). So these are also the pixels we want to iterate through.

Now given the correct address lines of ZBT, we needed to send over 4 pixels for each line. This process takes several clock cycles because protocol bits as well as information bits have to be sent over. So while sending any given bit, we ran a counter to allow the transmission module enough time to send over the correct bit.

Altogether, this meant we needed an FSM to keep track of what state we were in. At a high level, there are two main states – reading from ZBT and sending pixel data. Within the *SEND* state, we separated the sending phases for each of the 4 pixels in a line of memory. Upon entering each of these substates, we pulse the send signal to start the transmission procedure and wait until a 14-bit counter finishes one full iteration. Next we move to the next pixel, and once all pixels are finished, return to the *READ* state so we can read the next appropriate line of ZBT.

After all of this was done, we got more coherent images on the computer, but still had to make an adjustment. By careful inspection, we found the pixel that should appear first in the image was routinely being displayed last in its group of four, meaning we needed to cycle the order which we were sending each pixel. After this, the RS-232 transmission was successful.

6.2 Camera calibration

To perform camera calibration, we use Jean-Yves Bouguet's calibration toolkit. The procedure involves capturing images of a checkerboard mounted on a rigid, flat surface in various orientations (Figure 10) [3]. By specifying the actual width of the checkerboard squares and extracting the corners manually, the toolbox is able to calculate camera parameters that effectively specify the correspondence between a pixel and a unit of length in the world.

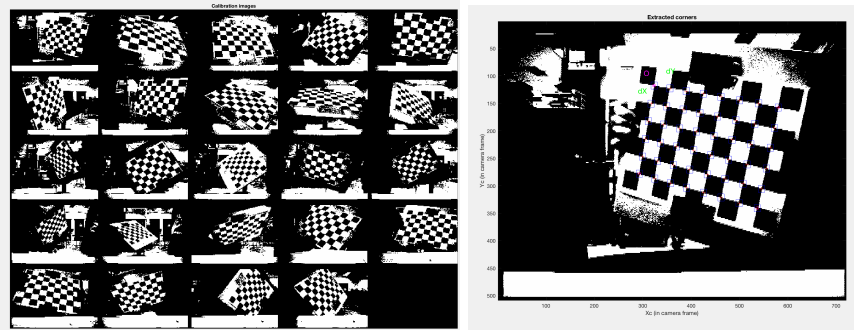

Figure 10: Twenty-four checkboard images used for calibration and the extracted corners

The toolbox outputs several intrinsic parameters. The particular ones of interest to us are contained in the intrinsic matrix, which we will use to reconstruct depth later. For our particular camera,

$$KK = \begin{bmatrix} 764.374 & 0 & 373.826 \\ 0 & 723.787 & 262.280 \\ 0 & 0 & 1 \end{bmatrix}$$

Once we have an intrinsic matrix, we can calculate the extrinsic parameters (translation/rotation) for the camera coordinate system relative to any arbitrary

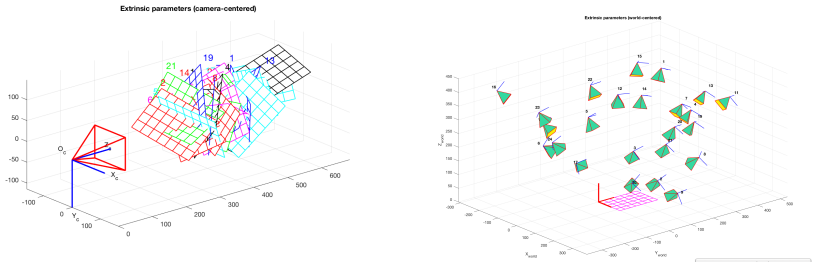

Figure 11: Visualization of the extrinsic parameters: position and orientation of our camera relative to each calibration checkerboard in a camera-centric (left) and world-centric (right) view

origin. Figure 11 shows the result of calculating the extrinsics for the calibration images – we are able to show how the camera is positioned and oriented relative to the checkerboard in each image.

We captured a picture of a checkerboard with the top-left corner (which we define as the origin) over the center of the rotation platform, so that the laser plane would pass through the origin, simplifying calculations. The rotation and translation matrices to transform the camera coordinate system into the world coordinate system are:

$$Rc = \begin{bmatrix} -0.0679 & 0.9977 & -0.0025 \\ 0.9811 & 0.0663 & -0.1819 \\ -0.1813 & -0.0148 & -0.9833 \end{bmatrix}$$

$$Tc = \begin{bmatrix} 3.6920 \\ -70.4300 \\ 243.2743 \end{bmatrix}$$

6.3 Module: *depth_reconstruction*

We implement the calculations as described in section 2.1 with 36-bit signed arithmetic throughout the module. The module takes the (x, y) pixel coordinates ($(0, 0)$ at the top-left corner of the camera image) as inputs. The pixel coordinates are technically each 11 bits unsigned, but we used 12 bits signed wires so we maintain signed arithmetic throughout the module. The module outputs 13 bit signed $(x\_world, y\_world, z\_world)$ coordinates $((0, 0, 0)$ at the origin of the world coordinate over the center of the platform, as defined during the calibration procedure).

6.3.1 Challenges

We implemented a working version in MATLAB first. Transferring the code to Verilog was relatively straightforward, but we needed to approximate the

non-integer camera parameters. Similar to the Gaussian blur module, we use fractions with powers of 2 in the denominators so we shift bits instead of implementing division. The calculation of λ requires dividing by a variable, so we use the divider.v module provided in lecture to calculate the quotient over $BIT\_WIDTH = 36$ clock cycles of pipelined division. ii The hardware implementation threw us unexpected challenges. In simulation, we compared the calculated 3D points with those the MATLAB implementation yielded, and discovered the following problems:

1. **Loss of precision.** Repeatedly multiplying by decimal factors that were then truncated by Verilog caused imprecise calculations to build up and output grossly incorrect values. To resolve this, we multiplied by a power of two early onto carry decimal places throughout the calculation, and then shift out the bits right before outputting the final 3D coordinates.

2. **Tradeoff between precision and overflow.** We initially chose a large decimal factor, which made our simulation-outputted values much closer to those from MATLAB. However, they still were as much as 40% off from the true values, so we attempted to make our fractional approximations more accurate. In some cases, this visibly caused overflow (as reflected in coordinates that were suddenly very far off), so for the sake of divider latency and resource efficiency we decided to settle for less precision by decreasing our decimal factor and accepting less-than-ideal approximations.

After experimenting with these modifications in simulation, we were able to attain coordinate calculations that were within 5% of the true values as outputted by MATLAB.

After synthesizing the module, however, we encountered further problems. We attempted to debug by outputting to the logic analyzer and hex display.

1. **Apparent timing glitches.** We created a signal that pulsed whenever the inputted pixel coordinate changed to signal to the divider that it needed to start a new calculation. However, often the signal did not pulse despite the point changing, or the divider did not output a done signal despite the assertion of a valid start and inputs. We still do not know the cause of the glitches, and it is uncertain whether this had an effect on the module's failure to compute correct values.

2. **Incorrect calculations, but no overflow.** We ensured that the calculations were all smaller than 36 bits, but probing into the intermediate calculations of the module revealed that some numbers were still being truncated.
   We are still uncertain of the cause, but our running hypothesis is that although the numbers are small enough to fit in the bit width, multiplying such large numbers still exceeds timing or resource constraints. Decreasing the precision of our approximations further and precomputing any values we could (such as multiplying our coefficients by the decimal factor in advance, instead of letting it be an adjustable parameter) often fixed the

problem.
The debugging process during this particular step was painstakingly slow; we had to put each of the 36-bit signals involved in the lengthy matrix calculations out to our debugging tools one-by-one, since they were so large. We had to experiment with different approximations and modifications until something worked. Sometimes the modifications were arbitrary – in one case, switching the order of multiplication of two signed numbers of the same width changed the result to the correct value!

The latest problem we were unable to debug was the divider module outputting zero, despite valid inputs of a start pulse and a non-zero dividend and divider. We did not complete this part of the project.

7 Rendering (Evan)

Given three dimensional coordinates from truction, we'd like to store them in memory, find their "display" coordinates, and store them in a visual RAM so they can be piped out to VGA and appear on a monitor. Additionally, we want the object to be shaded and viewable from various angles to really highlight the 3D quality of it.

7.1 Memory

In order to render points, we first need some form of visual RAM. This visual RAM acts as a bytemap for the monitor to read out pixels. Just like in the example NTSC code, *vram_display* sequentially reads out lines of ZBT to the VGA output. Where a certain three dimensional point, however, will appear on the monitor, depends the angle and position from which the monitor is viewing the object. This means, we need to store the original three dimensional points as well as a visual RAM. Unfortunately these two memory sources fill out the two banks of ZBT that we have. Unlike other projects, we aren't able to store camera data directly or have a second visual RAM buffer for rendering (to remove lag and glitches).

7.2 Module: *write2zbt*

With this in mind, once the (x, y, z) coordinates of a single point come out of depth reconstruction, we store them in ZBT memory. Based on a data valid signal from depth reconstruction, we hold increment our write address and hold our new write data, using the protocol

$$write\_data = \{6'b0, x, y, z\}$$

where x, y, and z are all signed 10 bit numbers.

There were no significant difficulties in creating this module, however because of where this module is in the pipeline, it was difficult to test this module for correctness.

7.3 Module: *renderer*

The goal of the *renderer* module is to retrieve these (x, y, z) points, then use the math outlined in 2.2 to translate them into monitor coordinates and pixel value so they could be used for updating the visual RAM.

7.3.1 Challenges

The greatest challenge here didn't come from the implementation of the module, but instead from our definition of coordinate systems. Without really any thought, I defined right on the monitor to be positive x, down on the monitor to be positive y, and towards the viewer to be positive z, this meant that first, I had to translate the (x, y, z) from the world coordinate system in depth reconstruction to monitor coordinates, and second I had to rotate in reverse. The rotation I mentioned in 2.2 only works for right-handed coordinate systems, so instead I had to use

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ sin(\theta) & 0 & -cos(\theta) \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Additionally, the actual implementation of the math had to be played with until it got into a working form. Altogether, this required lots of testing with various point clouds to ensure the correct effect occurred.

7.3.2 Module: *virtual_camera*

In order to perform the rotation, we needed to control a user's angle around the object. Additionally, we needed to control horizontal and vertical offsets so users could center the object in the screen. These were all taken care of by the *virtual_camera* module. For each of these actions – $angle$, $left\_offset$, and $right\_offset$, I created an *incrementer* module which incremented and decremented each of these according to a step size, initial value, and max value. When the appropriate button (for example buttons 0 and 1) are held down long enough, the value (angle) is incremented or decremented by step size. If the value ever goes past 0 or above max, it gets wrapped around.

7.3.3 Module: *trig_LUT*

Once we have an angle, we need to evaluate $cos(\theta)$ and $sin(\theta)$ to do the rotation calculation. Since Verilog can't evaluate trigonometric functions, I created a lookup table to just grab the value. I calculated both $cos(\theta)$ and $sin(\theta)$ for every angle multiple of 5 from 0 to 255, then multiplied the result by 64 to maintain precision. Later in *renderer* I shift 6 bits to the right to renormalize the value.

7.3.4 Blackout

An additional effect of rotations was that although rotations meant writing to a new point, the visual RAM still had the old monitor position of the point in memory. This meant that as you rotated, points would leave "trails" of where they had once been as a result of rotation. In order to solve this, I inserted a short blackout whenever *angle*, *left_offset*, or *right_offset* changed. This meant that when any of these values changed, a blackout signal switched all ZBT writing to simply black values for several frames. Though theoretically, a single frame should be enough, lots of noise appeared on the screen after a blackout. This prompted increasing blackout time to several frames. Further investigation could be done into decreasing blackout time, as it did cause a noticeable delay in rendering (whenever you rotate, the screen appeared black for a couple tenths of a second), but we did not have time to make this smoother (also again, we did not have a second ZBT memory to use as a visual RAM buffer).

7.4 Module: *zbt_controller*

With *renderer* complete, we need to update the second ZBT memory bank (the visual RAM) with the appropriate pixel. First, we determined the correct line of ZBT memory we needed to edit

$$addr = \{y, x[9:2]\}$$

because this mirrors the addresses read by *vram_display*

$$addr = \{vcount, hcount[9:2]\}$$

This led to issues, though, because when multiple points appear at the same monitor coordinates, we want only the closest (brightest) point. This naive approach doesn't consistently deliver the brightest pixel value, instead switching between the multiple valid pixel values.

To fix this, we introduce an extra read request. We read the ZBT line we want to write to then compare the old pixel value with the new, and on the next write cycle, write whichever pixel value is higher to memory (while maintaining the rest of the pixels on that line of ZBT).

7.5 Timing

Each of these modules was fairly easy to create individually, but the biggest challenge was integrating them all together so they worked out timing-wise. Trying to keep with the original sample NTSC ZBT interaction, we divide up clock cycles based on the value of *hcount[1:0]*. During *hcount[1:0]* = 2, *write2zbt* writes to the point cloud ZBT. In the following cycle, on *hcount[1:0]*=1, *renderer* reads the valid point from the point cloud ZBT. *zbt_controller* needs to use this information to update its address then read from the visual RAM ZBT so it can compare pixel values to maintain the brightest pixels in front. It does this in the

following cycle on *hcount[1:0]*=1, then on *hcount[1:0]*=2, ZBT controller uses this information as well as the old point information to write to the visual RAM ZBT. Additionally, *vram_display* is reading on *hcount[1:0]*=0 out to VGA.

Developing this timing scheme took a lot of precision and planning. Being one clock cycle off in almost any direction led to a complete failure of rendering.

7.6 Testing

In order to test the renderer, I wrote a *manual_write2zbt* module which acts like a write2zbt module, but writes points of my choice into memory. At first, this was limited simply to 4 points on a diagonal, but as the rendering developed, I could test overlayed points with different z-values, and in the end shapes generated in Python to test rotations. Unfortunately, I was unable to take any pictures of these objects. If I had time, I planned on generating a more complex point cloud, like a duck, to demonstrate the full powers of the renderer.

Like write2zbt, this module output a write address as well as a write value. The write address was incremented every 4 cycles (so that it would be held for an entire *hcount* cycle, then a large case switch was used to select write data based on address.

This module was extremely useful for testing each part of the rendering pipeline.

8 Conclusion

While we were unable to complete our project, we feel like we made a lot of progress towards completing a working 3D scanner. We wish we could have another day or two to complete the project and think that if we did, we likely would be able to finish, but given the time constraints did almost as much as we could to complete the project.

At this point, all parts of the project are working except the depth reconstruction math. As the math does require precise floating point values and many multiplications and divisions, it is a difficult process to debug, especially as we've started to see it become dependent on Verilog's synthesis procedure. Once this module is working the 3D scanner altogether will be functional because all the preprocessing is completed and verified and all the rendering is completed and verified.

Overall, we are glad that we took on this challenging project. At first when we proposed this project, we didn't have a good idea how to implement it – how to take it from an abstract idea to hard modules that we could actually code. This entire process helped us learn how to break larger problems into much smaller ones that were more achievable and verifiable.

For example, the process in which we developed the renderer consisted of first simply moving data from the first ZBT to the second and then rendering it. Next we included a dummy module (which would become *zbt_controller*) to take data from the first ZBT and write to the second. Next, we created the

manual_write2zbt module to render my own points to the screen. From there we continued to build up in extremely small steps so that we didn't have to worry about testing large modules and trying to find where they broke.

We also both feel that we learned a lot about how to think in a data / signal processing mindset. Both coming with more of 6-3 backgrounds, we were used to high level, abstract thinking. This project forced us to really consider how data is physically flowing through our system and about what should happen to it at each stage.

Last, we are both glad that we were really challenged by this project. We don't think we would've learned this much if we had chosen a less interesting, but easily more practical project. The entire process taught us about FPGAs, but also about teamwork and designing and planning.

For any future students, we advise you to stay organized and communicate well, just like in any team setting. We also highly recommend thinking about developing testing frameworks for each little development step, whether visual (through the LED display or monitor) or through a testbench, because once a module is completed, it can be deceptively difficult to see where things went wrong.

To end, we would like to thank Gim, Victor, and Shawn for advising the project throughout the process and being so dedicated to our success, from thinking through the debugging process with us to having faith in our ambitious project.

9 References

[1] http://mesh.brown.edu/byo3d/notes/byo3D.pdf
[2] http://web.mit.edu/6.111/www/f2015/projects/pbyang_Project_Final_Report.pdf
[3] http://www.vision.caltech.edu/bouguetj/calib_doc/index.html