Kernel Patching Intel Realsense R200 ToF Images

Barry, Daniel.
UC Robotics
Email: danbarry[2^4][@]googlemail[dot]com

June, 2017

Abstract: *In this paper we explore several methods of repairing the images from the Intel Realsense R200 ToF sensor [1], reviewing both the time taken for each method and visual inspection of the depth reconstruction. Our findings show that a recursive cross-shaped kernel is both lightweight and effective at reconstructing missing depth data.*

1 Introduction

1.1 Motivation

This project has been motivated by the depth images produced by the Intel Realsense R200 time of flight (ToF) camera [1], where it's limited focus on depth and imperfect capture of ToF data produces images where holes are visibly seen in the resulting image. This is highlighted in figure 1, where we can clearly see we have missing data, despite the depth ranges being within the focus area as seen in figure 2.

Figure 1: Example depth image produced by Intel Realsense R200.

Figure 2: Example colour image produced by Intel Realsense R200.

The reason for using this device over a much more powerful sensor array would be motivated by both it's size and power consumption, therefore as we look to recover the missing data, we should look to use as little processing power as possible. For most uses, it will be required that the 640x480 pixel image is processed in less than 33.33ms in order to allow the sensor data to be processed at 30FPS.

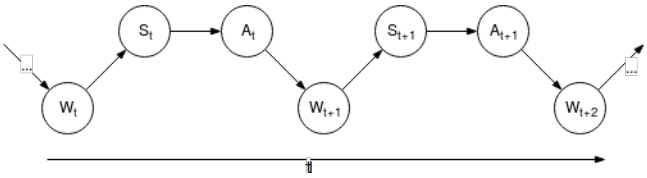

Figure 3: Sensor-actuator loop time series.

In figure 3 we have a diagram of a sensor-actuator time series, `A` is the actuator, `S` in the sensor, `W` is the world and `t` is time. The result of not being able to process `S_t` before `S_t+1`, means that `A_t+1` uses information from a previous state. In that time, the world state, `W`, could have changed between `W_t` and `W_t+1`. This behaviour is undesired, as the robot may appear slow to respond and potentially exhibit a hunting behaviour when attempting to react to the environment.

1.2 Background

The problem we have is that a 2D array of data is "corrupt", leaving holes signified as black (or zero) entries. This makes object detection more difficult, as it's harder job to figure out if the gap bridges two objects or represents the same object, as can be seen in figure 1. To fix this problem, we look to kernels.

A kernel is a simple image processing matrix that applies simple arithmetic over a small scope, typically a square with the 2D array. We pass this kernel over the image and in turn generate a new image representing the missing data. For our purposes, we can assume that the data that we do know is correct (or that we have no way of telling it isn't) and that the missing data is to be filled. This means we can do our kernel in place, to save on additional memory usage, copying time and information we're trying to store in the CPU cache.

2 Methodology

In this section we look at the kernels used and the method we use for testing them.

2.1 Kernels

2.1.1 Expanding Ring Cross

The main concept of this algorithm is to keep expanding a cross (in all four diagonal directions) until two or more pixels are found to get an average from.
The following is the implementation:

```
int width = depth.length - size;
int height = depth[0].length - size;
for(int y = size; y < width; y++){
  for(int x = size; x < height; x++){
    if(depth[y][x] != 0xFF000000){
      resultBuffer[y][x] = depth[y][x];
    }else{
      int s = 1;
      int sum = 0;
      int num = 0;
      while(s <= size){
        int nw = depth[y - s][x - s] & 0xFF;
        int ne = depth[y - s][x + s] & 0xFF;
        int sw = depth[y + s][x - s] & 0xFF;
        int se = depth[y + s][x + s] & 0xFF;
        if(nw != 0){
          sum += nw;
          ++num;
        }
        if(ne != 0){
          sum += ne;
          ++num;
        }
        if(sw != 0){
          sum += sw;
          ++num;
        }
        if(se != 0){
          sum += se;
          ++num;
        }
        if(num > 1){
          int avg = sum / num;
          resultBuffer[y][x] = (avg << 16) | (avg
              << 8) | (avg) | 0xFF000000;
          s = size;
        }
        s += 2;
      }
    }
  }
}
```

2.1.2 Depth From Image

Here we try to infer the depth of the missing information by looking at the maximum and minimum local values and trying to extrapolate where our data would fit in this range.
The following is the implementation:

```
for(int y = size; y < depth.length - size; y++){
  for(int x = size; x < depth[y].length - size;
      x++){
    if(depth[y][x] == 0xFF000000){
      int colMin = 255;
      int colMax = 0;
      int depMin = 255;
      int depMax = 0;
      int num = 0;
      for(int z = y - size; z < y + size; z++){
        for(int w = x - size; w < x + size; w++){
          int d = depth[z][w] & 0x000000FF;
          if(d != 0){
            int p = (
              (0x000000FF & (color[z][w] >> 16)) +
              (0x000000FF & (color[z][w] >>  8)) +
              (0x000000FF & (color[z][w]      ))
            ) >> 2;
            if(p < colMin){
              colMin = p;
            }
            if(p > colMax){
              colMax = p;
            }
            if(d < depMin){
              depMin = d;
            }
            if(d > depMax){
              depMax = d;
            }
            ++num;
          }
        }
      }
      if(num > 0){
        int colRng = (colMax - colMin) >> 2;
        int depRng = (depMax - depMin) >> 2;
        colMin += colRng;
        colMax -= colRng;
        depMin += depRng;
        depMax -= depRng;
        double m = (double)(depMax - depMin) /
            (double)(colMax - colMin);
        double c = depMax - (m * colMax);
        int p = (
          (0x000000FF & (color[y][x] >> 16)) +
          (0x000000FF & (color[y][x] >>  8)) +
          (0x000000FF & (color[y][x]      ))
        ) >> 2;
        int val = Math.max(depMin,
            Math.min(depMax, (int)((m * p) + c)));
        resultBuffer[y][x] = (val << 16) | (val
            << 8) | (val) | 0xFF000000;
      }
    }else{
      resultBuffer[y][x] = depth[y][x];
    }
  }
}
```

2.1.3 Expanding Ring Kernel

The expanding ring kernel method looks to keep expanding a ring until two or more pixels are found in an expanded ring.
The following is the implementation:

```
for(int y = size; y < depth.length - size; y++){
  for(int x = size; x < depth[y].length - size;
      x++){
    if(depth[y][x] != 0xFF000000){
      resultBuffer[y][x] = depth[y][x];
    }else{
      int s = 1;
      while(s > 0 && s <= size){
        int sum = 0;
        int num = 0;
        for(int i = -s; i < s; i++){
          if(depth[y - s][x + i] != 0xFF000000){
            sum += depth[y - s][x + i] &
                0x000000FF;
            ++num;
          }
          if(depth[y + s][x + i] != 0xFF000000){
            sum += depth[y + s][x + i] &
                0x000000FF;
            ++num;
          }
          if(depth[y + i][x - s] != 0xFF000000){
            sum += depth[y + i][x - s] &
                0x000000FF;
            ++num;
          }
          if(depth[y + i][x + s] != 0xFF000000){
            sum += depth[y + i][x + s] &
                0x000000FF;
            ++num;
          }
        }
```

```
        if(num > 1){
          int avg = sum / num;
          resultBuffer[y][x] = (avg << 16) | (avg
              << 8) | (avg) | 0xFF000000;
          s = -1;
        }else{
          ++s;
        }
      }
    }
  }
}
```

2.1.4 Linear Processing

This kernel is similar to `depthFromImage()`, except it tries to consider all of the pixels when creating a linear representation (`y = mx + c`) as to get the best pixel translation.
The following is the implementation:

```
for(int y = size; y < depth.length - size; y++){
  for(int x = size; x < depth[y].length - size;
      x++){
    if(depth[y][x] != 0xFF000000){
      resultBuffer[y][x] = depth[y][x];
    }else{
      int sumCol = 0;
      int sumDep = 0;
      double num = 0;
      for(int z = y - size; z < y + size; z++){
        for(int w = x - size; w < x + size; w++){
          if(gray(depth[z][w]) > 0){
            sumCol += gray(color[z][w]);
            sumDep += gray(depth[z][w]);
            num += 1.0;
          }
        }
      }
      if(num > 0){
        double avgCol = sumCol / num;
        double avgDep = sumDep / num;
        double diffCol = 0;
        double diffDep = 0;
        for(int z = y - size; z < y + size; z++){
          for(int w = x - size; w < x + size;
              w++){
            if(gray(depth[z][w]) > 0){
              diffCol += avgCol -
                  gray(color[z][w]);
              diffDep += avgDep -
                  gray(depth[z][w]);
            }
          }
        }
        diffCol /= num;
        diffDep /= num;
        double m = (avgDep - diffDep) / (avgCol -
            diffCol);
        m /= 32.0;
        double c = avgDep - (m * avgCol);
        int pixel = Math.max(0, Math.min(255,
            (int)((m * gray(color[y][x])) + c)));
        resultBuffer[y][x] = (pixel << 16) +
            (pixel << 8) + pixel + 0xFF000000;
      }else{
        resultBuffer[y][x] = depth[y][x];
      }
    }
  }
}
```

2.1.5 Stretch Horizontal Kernel

This "kernel" (possibly not quite in this category) simply stretches the last known valid pixel to cover any blank spaces in the horizontal direction.
The following is the implementation:

```
for(int y = 0; y < depth.length; y++){
  int p = 0xFF000000;
  for(int x = 0; x < depth[y].length; x++){
    if(depth[y][x] == 0xFF000000){
      resultBuffer[y][x] = p;
    }else{
      p = depth[y][x];
      resultBuffer[y][x] = p;
    }
  }
}
```

2.2 Experiment

The experiment was performed for all of the above methods, using kernel sizes of `{0, 1, 2, 4, 8, 16, 32, 64, 128}`, over 100 iterations of each. A simple average was then taken so that the results could be compared and the fundamental speed differences down to computer and operating system itself could be statistically negated from the results.
Finally, there will be a visual inspection of the kernel results as both the height maps views through a point-cloud application to help identify how well they perform. This will allow us to make the trade-off between time taken to process the image and the results of the processing.

3 Results

This experiment was performed on a computer with the following specifications:

- Intel Core i7-6700HQ CPU @ 2.60GHz x 8
- 15.6 GiB DDR4 RAM

Note that this experimenting used one core of this many core machine, which should hopefully also help reduce the affects of the underlying operating system and fairly random execution of various tasks.

3.1 Data

Type	Kernel Size	Average Time
c	0	5.74563856
c	1	10.48270523
c	2	10.73666647
c	4	11.99379303
c	8	16.99396152
c	16	17.97773062
c	32	13.5901129
c	64	15.30741013
c	128	13.58260142
d	0	8.57698502
d	1	15.74986771
d	2	22.34381421
d	4	49.40358899
d	8	188.70524135
d	16	587.9495269
d	32	2006.11533239
d	64	6118.13089812
d	128	11266.4092583
e	0	5.82274667
e	1	15.20993678
e	2	22.77032388

e	4	28.15856514
e	8	37.87871734
e	16	42.71119977
e	32	37.91791815
e	64	34.89005538
e	128	25.34426004
l	0	8.8967575
l	1	29.82275601
l	2	38.5474815
l	4	93.31674083
l	8	347.1734455
l	16	1269.37634313
l	32	4050.40232619
l	64	12393.2870707
l	128	22898.96308828
s	0	6.51492969
s	1	6.49016443
s	2	6.49409448
s	4	6.42818826
s	8	6.53791844
s	16	6.91076957
s	32	6.5586434
s	64	6.66761187
s	128	6.54124983

3.2 Output

Below are the various kernels of size 16, with corresponding point clouds at the same angle for each comparison. These are shown in figures 4 to 9.

Figure 4: Unprocessed data.

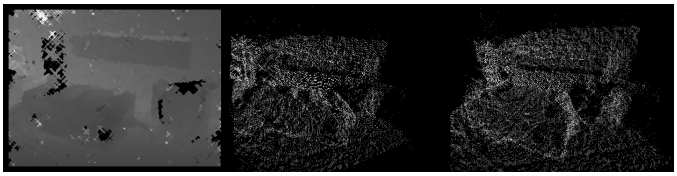

Figure 5: Expanding ring cross.

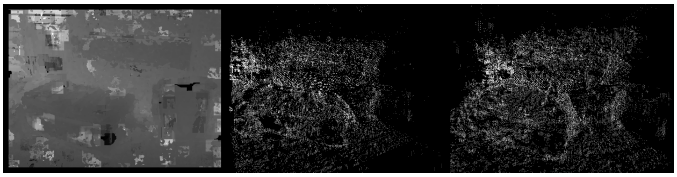

Figure 6: Depth from image.

3.3 Graphs

Below are the produced graphs, showing a plot of kernel size vs time (ms), to allow us to judge both whether the kernel can be run without our time frame of 33ms and whether the algorithm scales well. The results are shown in figures 10 to 14.

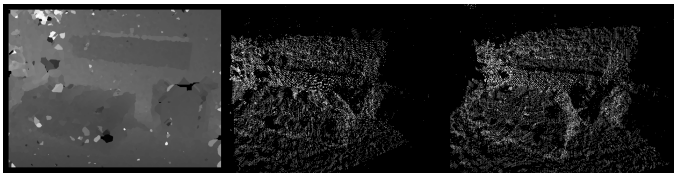

Figure 7: Expanding ring kernel.

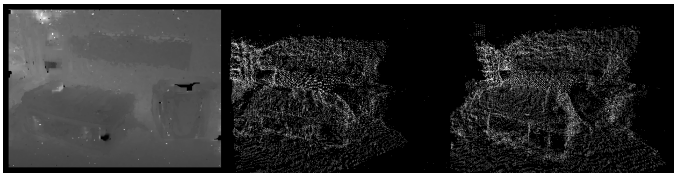

Figure 8: Linear processing.

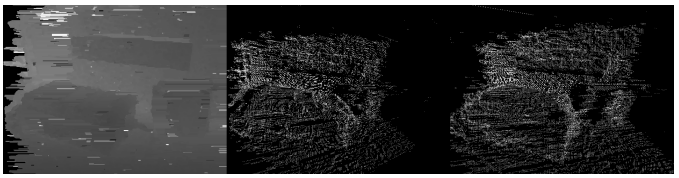

Figure 9: Stretch horizontal kernel.

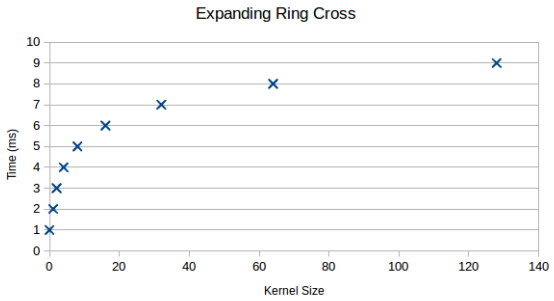

Figure 10: Expanding ring cross.

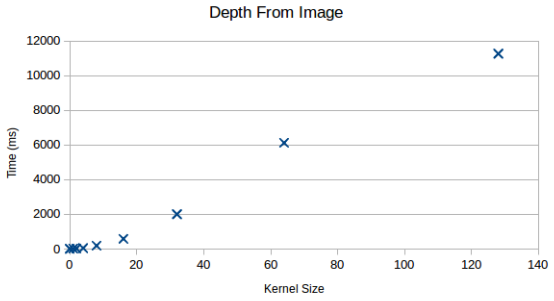

Figure 11: Depth from image.

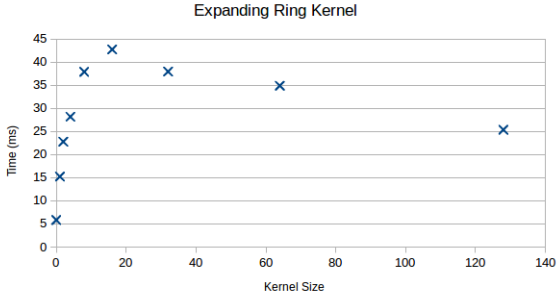

Figure 12: Expanding ring kernel.

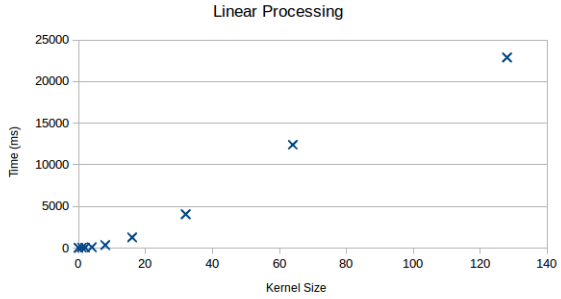

Figure 13: Linear processing.

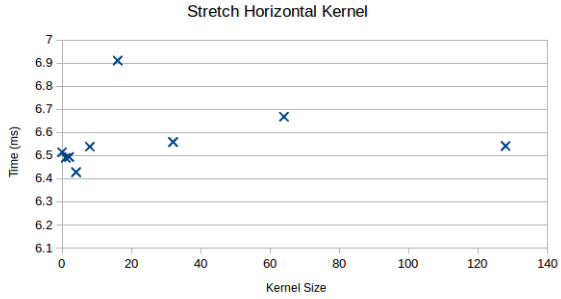

Figure 14: Stretch horizontal kernel.

4 Conclusion

Both models `d` and `l`, based on processing depth from colour image pixels, are too slow with any reasonably sized kernel. It appears that building a local translation between depth and colour is both not very effective and too process intensive for real-time application. Models `c`, `e` and `s` appear to have a good relationship between time taken and kernel size - some even reducing time for larger kernels. This is because as the kernel size increases, the further from the edge of the image the kernel starts in order to process the depth data. Eventually, the kernel size could grow to such a size that it ends up not processing anything at all. `128` is the largest size we thought was reasonable to have the kernel at, any larger and the FOV would drop too much to be useful.

Model `s` was by far the cheapest, being just a simple smudge of the last valid depth value - but produces the worst results when viewing the point cloud. This leaves models `c` and `e`, with `e` being both the more expensive operation and viewing the worst in the point cloud view. `c` appears to be our clear depth image repairer, with one of the smallest run times and good results.

Our testing method should not be considered fully conclusive, as the testing data did not fully represent what we shall see in our usage of the Intel Realsense R200 sensor. It was one example capture used for all testing to keep the tests repeatable and the experiments easy to compare. What we can conclude from these tests is that there is not a simple relationship between colour information and ToF in the depth image, so exploring this avenue further is unlikely to be fruitful.

Method `c`, expanding ring cross, shall certainly become our benchmark for future improvements to our ToF image patching and possibly be used as our initial test implementation on real hardware.

5 Future Work

One of the steps we are now discussing is to use a better sensor, such as that offered by the Xbox Kinnect, to artificially introduce our corrupted depth image data and then measure how close a kernel is able to come to the original image. We also discussed using a learning method for automatically producing better kernels based on a better wealth of test data, possibly using a method such as genetic programming to produce better kernels. The search space can be reduced to multiplication and finally addition for the relevant values after a threshold is met, also making use of the kernel's horizontal and vertical symmetry to reduce search time.

Additional method worth exploring include recurrent neural networks, deep neural networks, convolution neural networks and possibly the use of a hybrid fuzzer for producing and testing kernels.

Eventually, we would like to use this data to aid object detection within the VEX competition environment, requiring the fast detection of objects that vary in size and colour.

6 References

(1) Intel