View point reconstruction using a single image and depthmap

Vikko Smit
1293478
V.J.Smit@tudelft.nl

July 2015

1 Introduction

In the last decade 3D imaging has reached a level where it is quite common. Many technologies have been developed to determine the depth on images. Prior to my master I helped to develop a technology that does such a thing, depth from luminance. The goal of this technology is to extract depth information from single images and produce a depth map.

Human makes use of stereo triangulation to view 3D, so if we want to display our 3D reconstruction in a convincing way, we should predict the view of the scene for both eyes as if it were in a 3D area (Figure 1). The obvious problem here is that when you take data from a single picture, there are gaps in the reconstruction where one object covered a part of another object from that single view point.

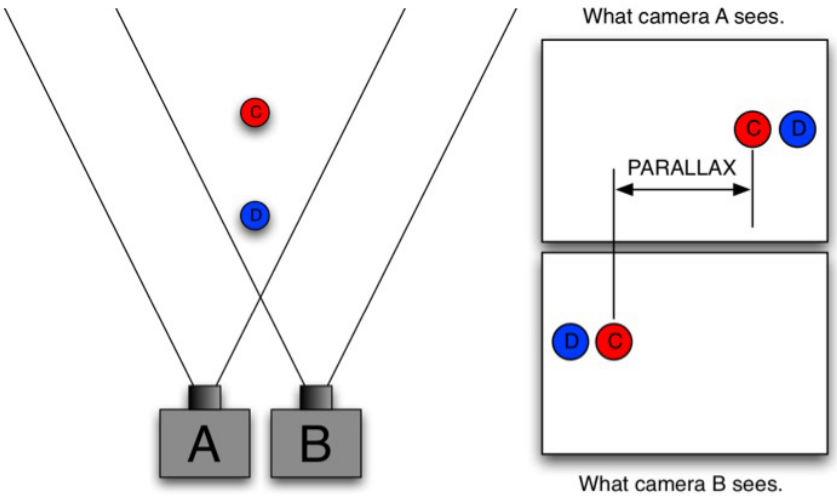

Figure 1: Naive (streaming) inpainting algorithm

In this project I try to find a approach where the missing data is reconstructed in such fashion that it is as close and plausible as the real view from another other angle.

2 Inpainting

2.1 Naive inpainting

A simple approach to generate a view from another angle is simply by shifting the pixels based on the value in the depthmap. Take an image with corresponding depthmap from that view point. Assume that the image I you have is a recording from the left camera, to get the right camera's view the pixels shift to left depending on the value of the depthmap 1.

$$I_{right} = I_{left}(i + c^1 * depth(i,j), j), \forall i,j \in I_{left} \tag{1}$$

To test the differences in reconstruction a dataset of Middlebury[2] was used that contains a left and right camera recording with corresponding depthmaps. A simple way to deal with missing data is Naive inpainting, where you use the knowledge of objects shifting to the left to fill missing data with the value from the left iteratively. This simple algorithm was developed by me last summer to inpaint a streaming video signal on an FPGA. For each pixel in the stream it overwrites all pixels up to the desired disparity. If a higher value of depth is streamed in, it will overlap the previous object as desired (Figure 2). Note that this only does linewise inpainting with only taking the previous values into

[1] c is a constant depending on where the right camera is located.
[2] http://vision.middlebury.edu/stereo/data/

Algorithm 1 Naive streaming inpainting

```
1: procedure INPAINT(Inputstream S)
2:     for each (pixel,depth) i, d ∈ S do
3:         disparity = c * d
4:         while disparity ≥ 0 do
5:             output(i-disparity) = I(i)
6:             disparity = disparity - 1;
7:         end while
8:     end for
9:     return output(i-c)
10: end procedure
```

account, therefor it would only gives good results if the missing pixels are just a few in a homogeneous region. However, propagating the values to the missing data line wise is a very fast approach that may help as a initial value for the next inpainting algorithms.

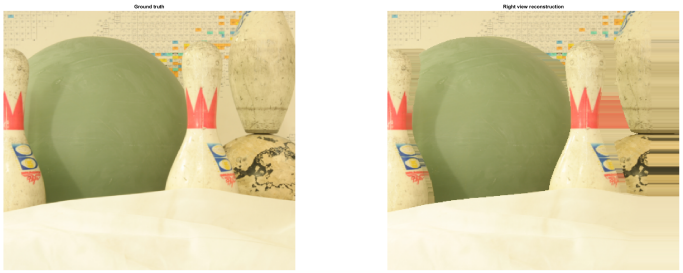

Figure 2: Naive (streaming) inpainting algorithm

2.2 Map of uncertainty

Using the Naive inpainting algorithm we can determine the pixels that need to be inpainted quite easily. We know that the most left pixel of each naive pixel inpainting is the actual pixel visible in the left camera view. We can exploit this knowledge by writing a 'false' for each pixel that is shifted to the left and write 'true' for all pixels on the right that inherit their value from this pixel. When applying the streaming algorithm with this setting the result is a value of 'false' for each pixel where we are certain what the value is and 'true' where the pixel is a naive estimation of the value, therefore the mask to inpaint is found

(Figure 3).

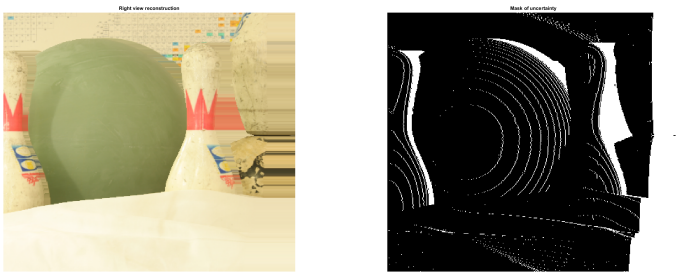

Figure 3: Mask of Uncertainty, constructed during naive inpainting.

2.3 Fast pixel Inpainting

Since the algorithm is supposed to be used in a realtime system, the first paper [1] I reviewed is on what is claimed to be 'a very fast inpainting algorithm with similar results as state of the art algorithms'. The approach they take is quite easy. You initialize a mask Ω to inpaint, then convolve masked regions with a kernel of your choice (Figure 2.3).

```
initialize Ω;
for (iter =0; iter < num_iteration; iter++)
  convolve masked regions with kernel;
```

a	b	a
b	0	b
a	b	a

c	c	c
c	0	c
c	c	c

Figure 4: Fast inpainting algorithm and used kernels. The diffusion kernels have values a=0.073235, b=0.176765, c=0.125

Eventhough the paper states that it has very good results, in my opinion that is due that it requires *human annotations of borders*! To keep it fair with other techniques I decided to implement it without annotations to see if it still

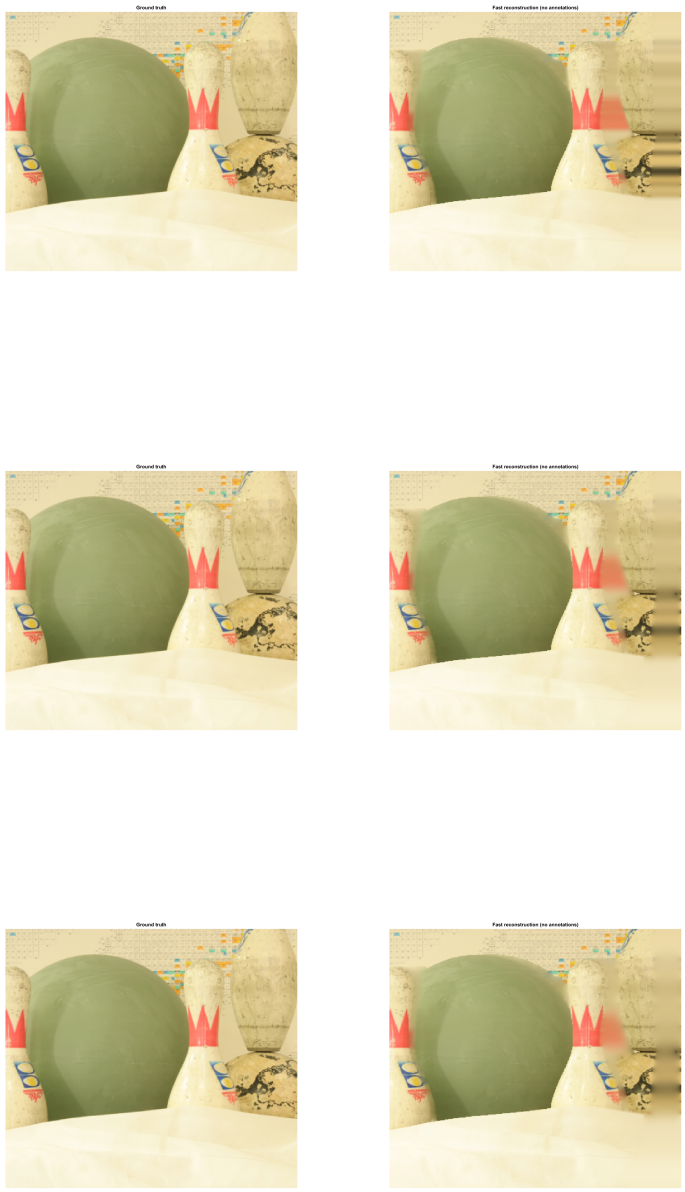

Figure 5: Fast inpainting using 10, 50 and 100 iterations from top to bottom respectively.

had any good results. Testing the results of different amounts of iterations on this kernel without annotations gave me quite similar results. The kernels act like a smoothing operator, which means that with enough iterations the full masks will get a blur effect. This is visible in Figure 5, where the blur on the mask increases with the iteration count.

2.4 Structure Inpaiting

A totally different approach then propagating pixels colors into the inpaint region is an algorithm I implemented imspired by the findings in [2]. In this paper the authors describe a non-parametric method to synthesize texture. They take an initial seed of texture, then with the assumption of an Markov Random Field they grow the initial patch by 1 pixel at the time. It considers the local area of the pixel that needs to reconstructed and tries to find the best match that keeps the texture intact. I implemented such a model that processes one pixel at the time, but instead of taking a random sample from the structure that it needs to match, I decided to create a vector of all possible patches in the original un-shifted image using a sliding window.

Next a pixel of the mask is compared with all windows of the original. To determine the best matching patch, the L2 norm is computed on the difference between the target and each window. Note that the image is converted in Lab colorspace here to make the correlation between those patches less sensitive to shifts in luminance by an uneven displacement in the patch.

The results of the norm vector is an index of pixels where the minimum value can be tracked back to the pixel value in the original. This value is copied into the target pixel and this process is repeated for every other pixel in the mask. Results of this approach don't look very natural either (Figure 6), probably because the inpainted pixels are on the edge where half of the patch is missing its very hard to get a proper match in the old pixel. This results in the same pixel being copied multiple times along a line while you would like a repeating pattern there. This effect led me to another approach: Instead of copying pixel by pixel, once a matching patch has been found, copy the whole patch on the mask locations.

This approach would keep the spatial local information of the pixel intact, however the result of this patch processing had the same issue: repeating patterns, but now with a bigger space in between (Figure 7). Downside is that if the original didn't have a smooth transition from one object to the other in the original, it is practically impossible to predict a transition in the reconstruction since that information does not exist. For the case where objects do not overlap this approach is expected to work pretty well, but on the current test image this is not the case.

2.5 PDE Inpainting

The last inpainting algorithm is a more complex approach, proposed in [3], using Partial Difference Equations to inpaint the mask. This approach is inspired by

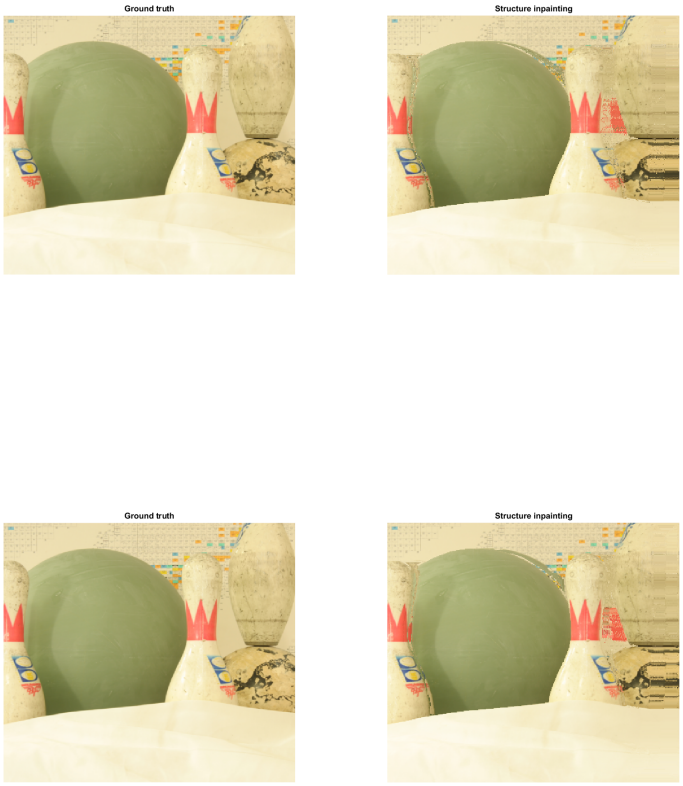

Figure 6: Structure inpainting pixel by pixel using a 5x5 and 9x9 window size respectively.

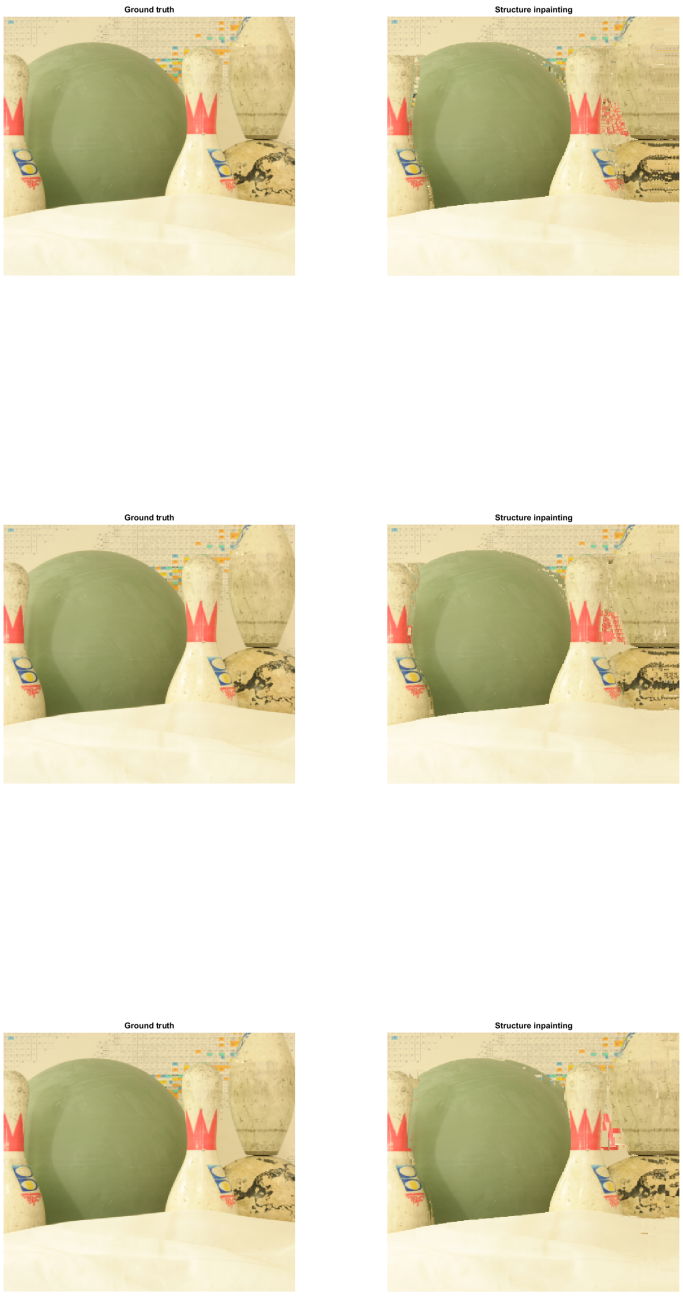

Figure 7: Structure inpainting patch by patch using a 5x5, 9x9 and 21x21 window size respectively.

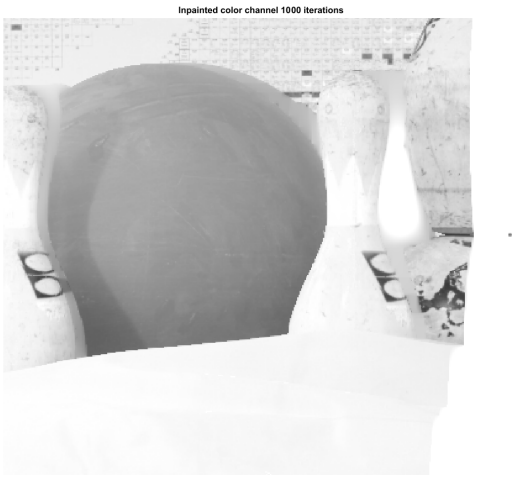

Figure 8: First attempt of PDE inpainting on a single color channel, using 1000 iterations.

art restorers that usually follow edges of the image that go into the damages area, then fill up the rest of the parts.

$$I_t(i,j) = \left(\overrightarrow{\delta L}(i,j) \cdot \frac{\overrightarrow{N}(i,j)}{\left|\overrightarrow{N}(i,j)\right|} \right) |\nabla I^n(i,j)| \tag{2}$$

The algorithm has a similar process: For each pixel on the border or the mask Ω you calculate the normals by taking the gradient in x and y direction, find the perpendicular vector $\overrightarrow{N}(i,j)$ (this is the direction you want to propagate your information) and normalize this direction. The x,y gradient represents the maximum change in intensity, so by taking the perpendicular vector to that you find the direction of the change. Next you need to measure the direction of the change of information $\overrightarrow{\delta L}$. To have a smooth propagation of information this is computed by taking the simple discrete Laplacian defined by $L(i,j) = I_{xx}(i,j) + Iyy(i,j)$, the double derivative in both x and y direction summed up. The change of information is projected on the direction we want to propagate the information on, then multiplied by the slope-limited version of the norm of the gradient of the image $|\nabla I^n(i,j)|2$. After a few iterations of this inpainting, anisotropic diffusion is applied to smoothen out the information along the edges. The script for this was available on MATLAB Central.

The first result is shown in Figure 8, visible is that the middle of the bigger regions that needed to be inpainted were not correctly filled. Each iteration the

Figure 9: PDE inpainting using multiscale image inpainting.

border calculates a pixel update and propagates a small bit of the color inside. The issue when you need to propagate a color all the way to the center it takes a exponential time to reach that place. I tried different approaches of smoothing to help the propagation, TV inpainting, but the most efficient seemed to be multiresolution inpainting. First different scales of the picture are created, then the smallest is inpainted and upscaled. The upscaled result is used as a seed for the next scale to inpaint and this process is repeated until the original image size is reached. With this approach the result in Figure 9 was obtained.

Eventhough the texture is still missing and there are still artifacts, it is a improvement compared to the previous approaches. There is one thing that still needs some finetuning: The edge preservation is not optimal. Even on very simple synthetic test images the edges turned out to be slightly blurred.

3 Conclusion & Future work

The goal of this project was to find a suitable inpainting technique to fill missing data after a displacement due to another viewpoint. I started off with very simple approaches that would be feasible to work realtime, then tried to improve the results by integrating more complex approaches.

I ended up with a system that would take an hour of MATLAB processing to inpaint and the result was still not something natural. However, I am convinced that in other settings this inpainting algorithm should work quite well, especially when the depthmap does not change that rapidly and the inpaint regions are smaller. The test image I used was a hard example: Objects partially covered, complex textures, similar colors.

For now I can conclude that it would not be feasible to do image inpainting in a very natural way if you require realtime results. There are a lot of different approaches out there, but the more quality you require of the result, the less realtime it gets.

Another problem that occurs with all these techniques is reproducing a natural texture in larger regions. I would suggest to combine one of the edge-inward-propagation techniques with the texture synthesis approach, this would keep a smooth transition from 'real' data to inpainted data, while making the inpainted data have a natural structure instead of a huge blurred area.

I really enjoyed the research on inpainting, eventhough the process was frustrating now and then. Some of the techniques when not fully optimized took over 2.5 hours to converge, many variables caused the system to become instable with a slight tweak. If you are very patient and handle the variable delicately, you're welcome to try it out yourself, the sourcecode is available on https://github.com/Vikko/CV_Project_Inpainting.

4 References

References

[1] R. McKenna M. Oliveira, B. Bowen and Y. Chang. Fast digital image inpainting. 2001.

[2] T. Leung A.Efros. Texture synthesis by non-parametric sampling. 1999.

[3] T. Leung A.Efros. Texture synthesis by non-parametric sampling. 1999.