

# CS7495 Computer Vision Project

## Dual Photography

Ang Lee / gtg082v

### Abstract

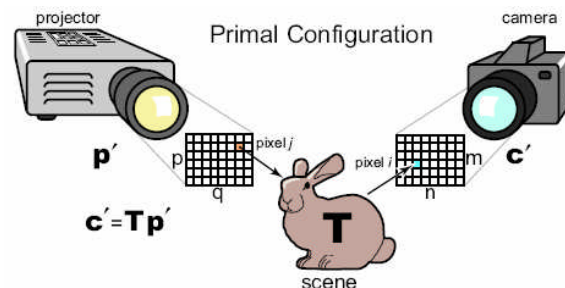
Dual Photography is a paper appeared in SIGGRAPH 2005. It exploits Helmholtz reciprocity to interchange the light (projector) and camera in a scene. I intended to implement the part with the scene that has one project and one camera. Both brute force method and the quick method have been implemented. However, due to the limitation of the device available to me, the results are too poor to be shown. I start at briefly going through the method and then discuss what I have done. In the end, I discuss what is poor and what I have learned from this project.



Using dual Photography We can simulate what a camera sees if it is put at the position of the projector.

### Method / Pseudo-Code

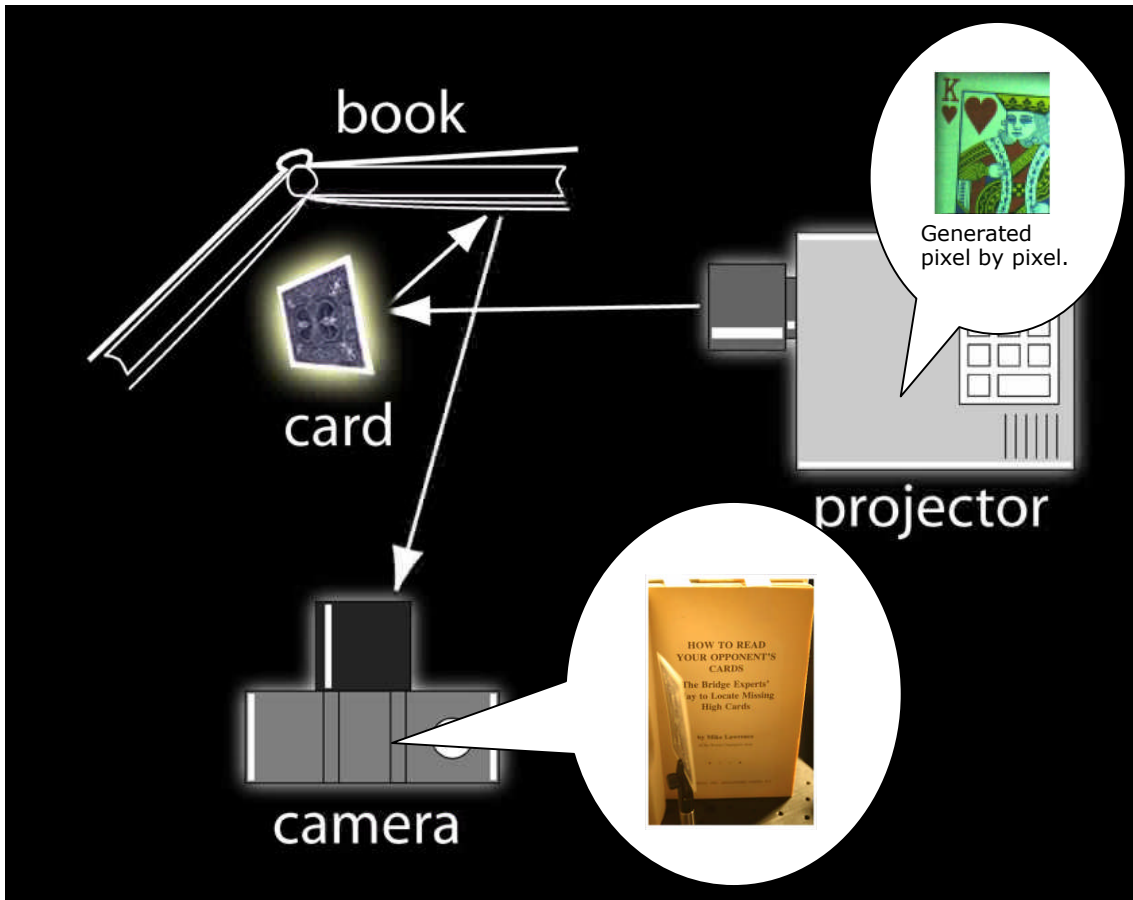
In this project I focused on the sections that deals with the scene with one projector and one camera which I am going to discuss here. In the original paper, the authors also discussed how this scheme could be extended to the scenes with multiple camera/projectors and then could be use in scene relighting.



The idea here is to exploit Helmholtz reciprocity – the symmetry of BRDF's. In the simplest form, a photon travels from a light source, say a pixel of a projector, bouncing one or several times and reaches a sensor among the array of sensor of a camera. There is a number describes this relationship between the source and the destination. Now the projector has  $p \times q$  pixels, and the size of sensor array of the camera is  $m \times n$ , for each pair of one from a first group and the other from another, there is such number. All these  $p \times q \times m \times n$  numbers together form a big array  $T$ . All above can be described as  $c' = T p'$ , as shown in the figure above, where  $c'$  is an array of size  $m \times n \times 1$ ,  $T$  is an array of size  $m \times n \times p \times q$ , and  $p'$  is an array of size  $p \times q \times 1$ .

### Brute force method

Helmholtz reciprocity tells us  $p'' = T^T C''$ . Intuitively, it means, once we have  $T$  we can get  $P''$ , (i.e. the picture) seen by the projector by  $c''$ , (i.e. the camera.) I personally understand it this way. To simulate the picture seen by the projector, we can do the following: set up a black room. Lit only one pixel of the projector. This ray then bounce in the scene full of objects and may defuse reflect...etc. The camera captures part of this ray. When the ray bounce at some place reddish, the energy other than red would be absorbed and thus the array of sensor in the camera gets red light. For the same reason, catching blue light means the surface is blue. Hence, the follow cool demo provided by the authors becomes very intuitive. The camera can not see the face of the playing card. When the ray projected on to the red region of the card the page of the book will be illuminated red, and then be caught by the camera. Projecting many rays in raster order, and the image of king of heart will be generated pixel by pixel in this manner.

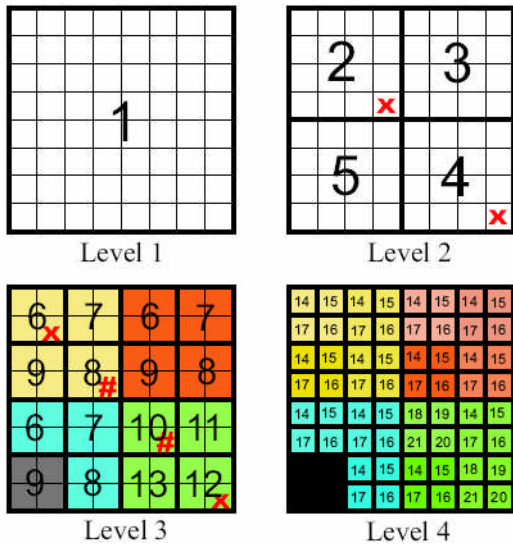


### Adaptive Multiplexed Illumination

The problem of the above scheme (brute force) is that it is very time consuming. The reason is  $T$  is a huge matrix (say  $1024 \times 768 \times 320 \times 240 = 60397977600$  entries). The paper proposed a much more efficient method. The idea is that if projecting two rays at the same time and they end not affecting each other, then we can save our time by doing so. "Affecting" means when projecting

these two rays separately, the region been illuminated in the camera images do not overlap. This is called "Adaptive Multiplexed Illumination". The method recursively divide a region of projection into 4 sub-regions. In an illuminating level (divide regions once), all the sub-regions have same figure. And in each lit within a level, all the sub-regions lit do not affect each other. This is achieved by considering two constraints: 1. If two regions don't affect each other, nor do their children. 2. Any pair the children come from the same region can not be lit together. This is done until the sub-region is 1x1 pixel.

Here is an example:



Example - The numbers indicate the order of the regions to be lit. In the image captured in lit 2 and lit 4 more than one pixel are lit in both rounds, so 2 & 4 are considered to affected each other. Thus, their children could affect. On the other hand children of 2, 3, 5, won't affect. So, in lit 6, 7, 8, 9 we lit the children of 2, 3, 5, and followed by 10, 11, 12, and 13 which are the children of 4. And when a region lit and contribute nothing, that region no longer need to be lit (no need to sub-divide it into four children) and is culled, as the lower left 9 illustrated in level

The **pseudo-code** of this algorithm is:

(Provided in the paper, modified and annotated (part) by me)

**Main Flow:**

```

1. Initialization();
Repeat{
    //construct a conflict-free lists of blocks that could be processed in parallel
    2. ConstructConflictFreeLists();
    //illuminate scenes with patterns constructed from each list and acquire with camera
    3. AcquireImages();
    //process images, store results, generate new lists of blocks for next iteration
    4. ProcessResults();
} until lowest level is reached.

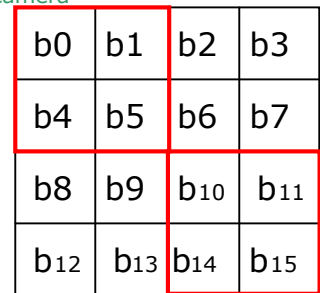
```

**Sub-functions**

```

1. Initialization() {
for each camera pixel k {
    //Initially assume every camera pixel is affected by block 0; the floodlit image
    Bk = {b0} // take the example above, say pixel 256 has been affect by lit 2 and 4 in level 2, i.e. that
                pixel has value > 0 (or some threshold) in the camera image of lit 2 and 4, then in level 3
                (current level) B256 = {b0, b1, b4, b5, b10, b11, b14, b15};

```



```

}
C = empty; // initialize set of conflicts to empty
// C is a List of Pairs of blocks which can not be lit at the same time.
// take the example above, in iteration 3 (level 3), C = {(b0,b1), (b0,b4), (b0,b5), (b1,b4),
(b1,b5), (b4,b5), (b2,b3), (b2,b6), (b2,b7), (b3,b6), (b3,b7), (b6,b7), (b8,b9), (b8,b12),
(b8,b13), (b9,b12), (b9,b13), (b12,b13), (b10,b11), (b10,b14), (b10,b15), (b11,b14),
(b11,b15), (b14,b15), (b0,b10), (b0,b11), (b0,b14), (b0,b15), (b1,b10), (b1,b11), (b1,b14),
(b1,b15), (b4,b10), (b4,b11), (b4,b14), (b4,b15), (b5,b10), (b5,b11), (b5,b14), (b5,b15)}
}
2. ConstructConflictFreeLists(){
//from graph structure
B = Union(Bk) //Nodes B, edge C,
(L[0], ..., L[N-1]) = GraphColor(graph(B,C)); //N Lists of node returned
// continue the above example,
// L[0] = {0, 2, 8}, L[1] = {1, 3, 9, 11}
// L[2] = {5, 7, 13}, L[3] = {4, 5, 12}
// L[4] = {10}, L[5] = {11}
// L[6] = {14}, L[7] = {15}
}
3. AcquireImages(){
//We now have N conflict-free lists L[i]'s
for i = 0 to N-1 {
generate patten P[i] from L[i]; //light pixels from all blocks in L[i]
illuminate pattern P[i];
capture HDR image I[i];
}
}
4. ProcessResults(){
C = empty;
for each camera pixel k {
new_Bk = empty;
for i=0 to N-1{
// find block (if any) that affects current pixel
current_block = intersect(Bk, L[i]); // because L[i] was conflict free.
// This can be at most one block (per L[i])
// for example in lit #6, the blocks 0, 2, 8 are lit
// if a pixel is lit in the camera image is lit by one
// and exactly one of these three.

if (current_block= empty)
continue; // pixel k not affected by L[i];
if (pixel k in I[i] = 0)
continue; // no value measured, do nothing
if (pixel k in I[i] < threshold) or last iteration {
// below the threshold so store the energy here.
// T() is the hierarchical representation of the matrix
// indexed by block in the subdivision tree and camera pixel k
T (current pixel, k) = pixel k in I[i];
Continue; // no futtter subdivision
else
//request subdivision for this block
insert 4 children of current_block into new_Bk
}
}
}

```

```

    // set Bk for the next iteration
    Bk = new_Bk;
    // collect conflicts and add to C for the next iteration
    for each pair (s, t) where s and t are in Bk and s != t {
        insert (s,t) into C;    // see the example in Initialization above
    }
}
}

```

## Brute force method

## What I have done

I started with adaptive multiplexed illumination, and then switch to brute force method, both failed to get decent result. The reason is also provided below in the next section. The tool I used is C++ / OpenGL and OpenCV APIs. OpenGL was used to show the project pattern and OpenCV takes the camera control part.

### Adaptive Multiplexed Illumination

This is harder than I expected. To figure out the meaning of pseudo-code took me some time. I used arrays of link lists as the data structure for both C and B to have the storage consumption dynamical. However, using link lists makes the search, delete and append to be harder. Most importantly, as in many other image processing applications, the process time and storage amount used need to be take carefully. The memory consumption is big here. A pixel could be affect by several blocks together, and each pixel has a link list recording this. To find conflict pairs we need to go through each of these for several time (finding pairs) the pixel number is on the order of 80000, In extreme cases, when the block size gets smaller and smaller, the length of each list  $B_k$  of a pixel could be on the order of (800000) That is  $800000 * 800000 = 64 * 10^{10}$  integers!!

Here I select some most challenging parts and describe my methods:

1. To do 'Union' of link lists  $B_k$ ,
  - 1.1 links all the  $B_k$  to be a very long link list.
  - 1.2 Dynamically declare a array of size equals to the link list and put the nodes into this array.
  - 1.3 Merge sort the array members and combine the repeated terms.
2. Create lit lists, base on C
  - 2.0 Note that the C has been set up that the first entry in each pair is smaller than the second one.
  - 2.1 put the result of Union( $B_k$ ) into an array. Prepare another array as a buffer.
  - 2.2 use two for loops to go through all the pairs in the array and to check C at the same time.
    - If the pair appears in C. Remove one of them and put it into the buffer. The result is that

all the nodes in the array do not conflict. Put the nodes left in the array into a lit link list, thus the array becomes empty, and we have a lit list that all its members do not conflict.

2.3 if the buffer is not empty. Move all the nodes in the buffer to the array. And repeat 2.2 until the buffer is empty.

3. Number the blocks.

3.1 I adopt the naming rule of the original paper, i.e. clockwise name the four blocks starting from the top-left one. And in each level the first block is numbered 0.

3.2 The size of block can be determined once one knows what level currently is. The function DivideSquareToFourBlocks () take the coordinates of left, right, up, and bottom of a block, and recursively divide it into four blocks by calling itself four times.

3.3 Note that number the block this way is good because, if the parent block ID is X, its four children will simply be  $4*X$ ,  $4*X+1$ ,  $4*X+2$ ,  $4*X+3$  respectively. When processing result. The Bk list could be create easily:

```
temp1 = new BkNode; temp1->node = currentBlock*4;   temp1->nxt = NULL;   temp2 = temp1;
temp1 = new BkNode; temp1->node = currentBlock*4+1; temp1->nxt = NULL; temp2->nxt = temp1;
temp1 = new BkNode; temp1->node = currentBlock*4+2; temp1->nxt = NULL; temp2->nxt->nxt = temp1;
temp1 = new BkNode; temp1->node = currentBlock*4+3; temp1->nxt = NULL; temp2->nxt->nxt->nxt = temp1;
```

4. Generate Project Pattern

4.1 create a class 'block', its members are three integers: left, bottom, LitOrder, and Boolean AlreadyCulled; go through the lit lists and put the ID of lit list as the LitOrder of the block.

5. Collect conflicts and add to C for next iteration

5.1 Go through  $b_k$  and put all the possible combination of pairs of node into C.

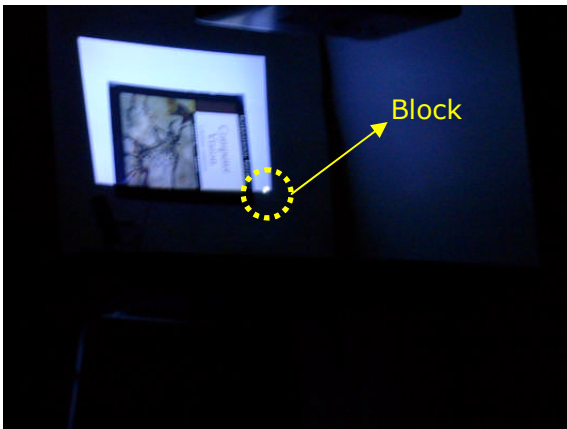
5.2 keep adding entries of Bk into C make the size of C explode very quickly. And all these are store in an array. The program checks the number of nodes being added as compared to the max length of the array. Once this exceeds the threshold, say 80% of the max length, do a merge sort and combine the common terms to reduce the consumption. Since among Bk's the nodes contained usually have a lot of portion overlapped, this could usually reduce the consumption of memory significantly.

### **Brute force method**

Realizing adaptive multiplexed illumination method failed, I try the Brute force method as well. Using the brute force method, I simply light the blocks of 3x3 in raster order, it takes 5 hours to generate 1/3 of the result, however, the result is too dark, I change to 6x6, and it took 4 hours to generate 1/3 of the result but still failed to get good result. Finally I tried 30x30, which makes the final result only has the size 35x26, can finally get something but it is still very bad.

## Device Set Up and preprocessing

I set up the experiment in a room at TSRB. The project resolution was set to 1024x768 and the camera I used was Logitech Notebook Pro the resolution was set to 320x240. The capture time was very late mid night with all the light near turned off. However, there are still some light pass through the window and from rooms in the distance. The bottle neck of time consumption was the response time of camera. As a result, as describe above it take several hours to get results to be judged. As the paper suggests, before starting I took a floodlit image and subtract it from all the images captured later.



## Result



This is the only "ok" result I have gotten so far, with block size 30x30 used the image size is 35x26 pixel<sup>2</sup>. All the others are too dark. With 3x3 block size or 6x6 block size, the difference between the maximum and minimum pixel value gotten is one or two which is too bad to be shown. This result took about 1 hour. Note this image has been adjusted using photoshop.

## Discussion (Why failed)

In this section, I discuss the general reasons for why the result is not good for both Adaptive Multiplexed Illumination method and brute force method. And this is followed this by some issues applied to Adaptive Multiplexed Illumination only.

Camera automatically compensate the light

The camera I used like many automatic digital cameras, it adjusts and balances the image automatically which is not desirable in this project. I have change and optimize all the adjustable parameters. But the camera still automatically compensates the light change. For example when the scene change from very dark to mid-level light, the image captured changed from all black to very bright, then back to mid-level.

Not totally dark environment

As described above, the environment is not totally dark, light permeates through the shutter. And other light came from room in the distance also make the room not totally dark.

Focus of projector

The focal length of the project is limited. The object can't be move too close to the projector. This makes the neighboring pixels affect each other a lot. This is like to applied a blur filter upon the result image. This is one of the main reasons to make Adaptive Multiplexed Illumination didn't work.

Low-resolution camera

The camera I used has only 320x240 resolution. However, this should not be a big issue. Since the resolution of the image I tried to generate is decided by the block size and the resolution of projector mainly.

### Issues about Adaptive Multiplexed Illumination

The problem is that the memory consumption is too big to use the code. Only in level 5, the block number is already  $16 \times 12 = 192$ . In the worst case, if all of them affect each other. The C list is already  $192 \text{ select } 2 = 18336$  pairs, in level 10, this number is 309237252096, and there are totally 10 levels to finish. Multiply this number by the pixel number of camera 76800 (this could happened when processing result from Bk to C) the memory consumption explode after only a few level.

It is important to discuss how can the worst case happened. The worst case happens if the scene

is full of diffusive material such as paper of book cover as I was using. The reason is that if the scene is diffusive, when a ray hit the diffusive surface, it is scattered in to rays along many directions by the surface. These rays later will be caught by many camera sensors. The Adaptive Multiplexed Illumination method save time because it can process the source rays which do not affect each other in the camera image at the same time. When there are lots of diffusion, there are more chances that two light sources affect, i.e. illuminate the same pixel in the camera image. As a result, the worst case happens and memory consumption exploded very quickly.

So in my case, I don't have camera with high resolution, what even worse it the issue of the focal length of the projector as discussed. Hence, Adaptive Multiplexed method is not practical. Another tricky thing is that: when using Adaptive Multiplexed Illumination, we don't want diffusive materials to apper in the scene; on the other hand, when using brute force method, diffusive materials is desirable since it makes more light (energy) go to the camera sensor and result in better contrast of the final image.

### **What I have learned**

Through out the process of completing this project, I feel that choosing the data structure to be used is very important when dealing with the tradeoff between memory storage space and time. I spent too much time on coding the Adaptive Multiplexed Illumination algorithm which turns out to be not practical. And though I complete the brute force code fast, I didn't realize to run it take such a long time. I should start running it earlier. Also I learned that, this paper deals with the problem that needs good equipment, before starting, I should google how much the camera they used cost first :)