GPGPU Assignment #3

Author: Yu Sheng Lin Instructor: Wei Chao Chen

March 29, 2016

1 Goals

You have to implement Poisson Editing for image cloning in this assignment.

2 Description

2.1 Image Cloning

Figure 1. shows possible inputs of image cloning algorithms: (a) is the background, (b) is the target image and (c) is a boolean mask. A very naive image cloning example is shown in Figure 2. It just copy and paste the image from the target image to the background image if the mask is true. The is just what we do in the sample codes we provide.

Of course we won't let you implement that in this assignment. You will implement Poisson Editing in this assignment, which will be described in more details later in this document.

2.2 Function Signature

The function signature of this assignment is Listing 1.. `background` and `output` are interlaced RGB images of size $W_b \times H_b$ and range from 0 to 255. `target` is the same except that it's size is $W_t \times H_t$. `mask` is the mask which only has one color channel and we use `0.0f/255.0f` to represent false/true.

```
void PoissonImageCloning(
    const float *background,
    const float *target,
    const float *mask,
    float *output,
    const int wb, const int hb, const int wt, const int ht,
    const int oy, const int ox
);
```

Listing 1: Function signature

We will also assign the offset O_y, O_x, which means the offset of the target image in the background image (from the top-left corner). To generate the image shown in Figure 2. with the the sample code, run the program by Listing 2..

The image format we are using is the PGM/PPM format, which can be edited by many image processing softwares. You can generate new testcases if you desire.

```
./a.out img_background.ppm img_target.ppm img_mask.pgm 130 600 output.ppm
```

Listing 2: Execute your code

(a) The background image $W_b \times H_b$.

(b) The target image which will be pasted to the background, $W_t \times H_t$.

(c) The mask $W_t \times H_t$.

Figure 1: The input images of this assignment.

2.3 Poisson Editing

Ignoring the abstruse math of the paper, the 4-neighbor linear system of Poisson Editing is simply Equation 1. (also refer to Figure 3.). (If you are interested, please refer to the original paper of Poisson Editing.)

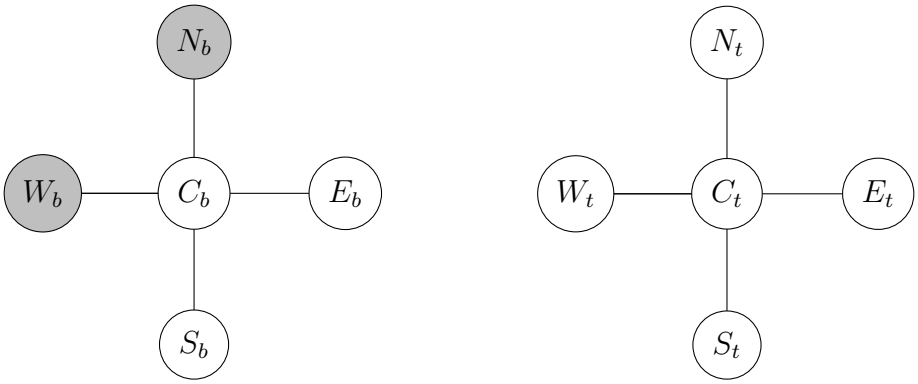
$$4C_b - (S_b + E_b) = 4C_t - (N_t + W_t + S_t + E_t) + (N_b + W_b) \tag{1}$$

With the Jacobi Iteration, the iteration step is Equation 2..

$$C'_b = \frac{1}{4}\left[\underbrace{4C_t - (N_t + W_t + S_t + E_t) + (N_b + W_b)}_{\text{Fixed during iterations}} + \underbrace{(S_b + E_b)}_{\text{Current value}}\right] \tag{2}$$

It's your task to generalize (just by intuition) and figure out what if (1) the distrubution of gray points changes and (2) the point only has fewer neighbors.

Figure 2: A very naive (and stupid) image cloning.



(a) The values to be solved.

(b) The corresponding target image.

Figure 3: The gray nodes are the boundary (the black pixels in the mask).

2.4 Acceleration

This part is counted as bonus. And if you implement this part, you should analyze the acceleration in order to actually get the bonus. A good metric is to compare convergence against the number of iterations or execution time. Besides, the parameters in this subsection were NOT tried and are TA's CONJECTURES.

First, acute readers might notice that the time for a value to propagate from the boundary is just propotional to the of distance from the boundary, and the time to converge may even be square of it! This method will require thousands of iterations to converge, which is very impratical. Figure 4. shows TA's baseline implementation, and you can see that it need 20000 iterations to converge.

A simple solution is to use the multigrid method: solve at lower resolution initially then upsample and solve again, and I "guess" that 1/64x, 1/16x, 1/4x and 1x scales conjunted with the nearest-neighbor sampling is acceptable.

Second you can try the successive over-relaxation (SOR), which makes the iteration step becomes Equation 3..

$$C'_{b,SOR} = \omega C'_b + (1-\omega)C_b \tag{3}$$

SOR is just a interpolation/extrapolation between current values and the values of the next iteration. Usually, we use $\omega < 1$ to ensure the convergence while $\omega > 1$ to accelerate the convergence. Also note that when $\omega = 1$, the SOR is exactly the original method. You could choose larger ω (1.2, I "guess") initially and decrease it to 1 later after a few iterations.

3 Grading

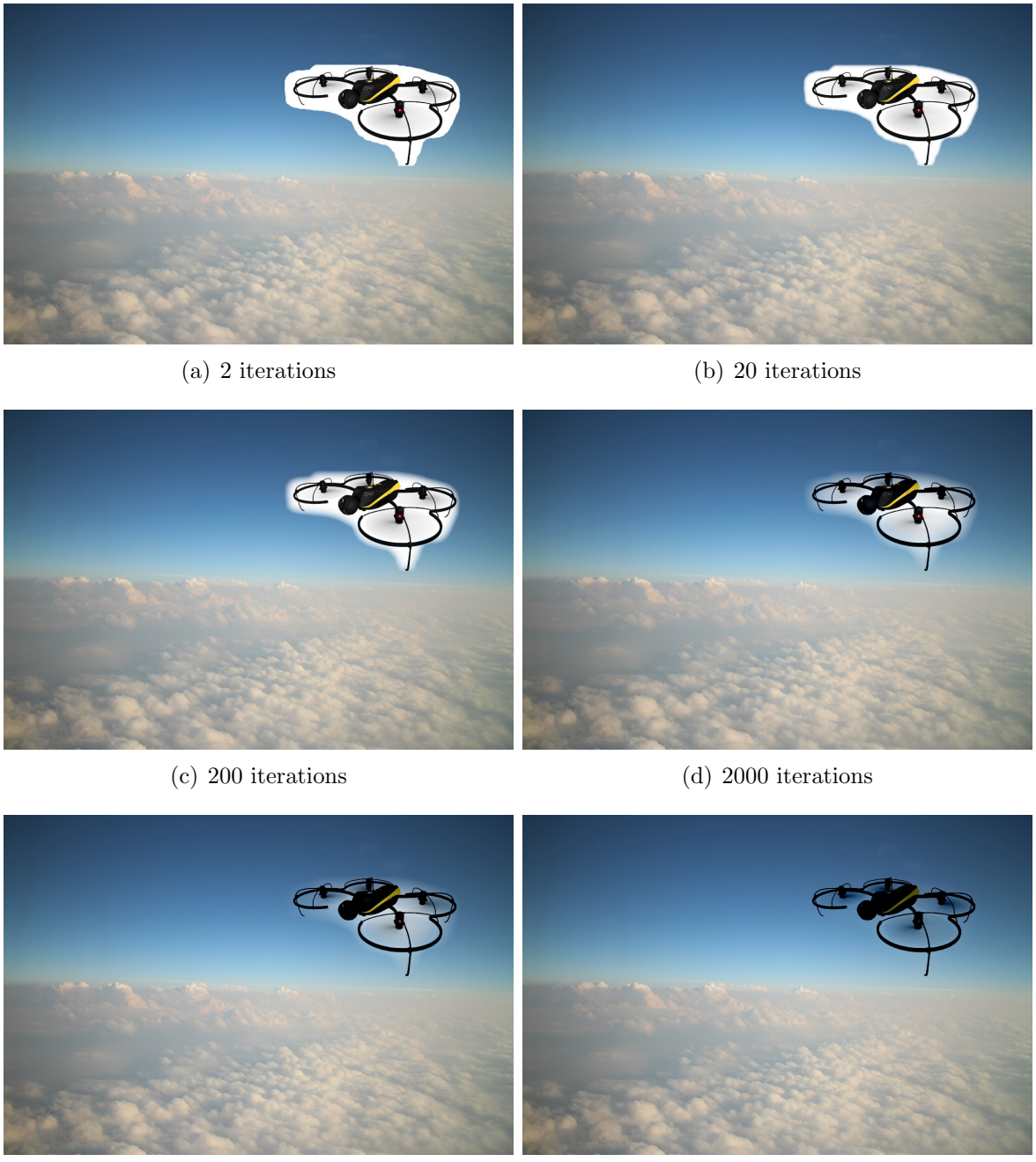
1. 100pts if you finish baseline (Jacobi, no acceleration).
2. At most 10pts bonus if you sumbit other interesting artifacts.
3. At most 30pts if you accelerate and analyze (mainly by the report you submit).

4 Submission

- The submission deadline is [3 weeks or more].
- Please submits the result in JPG format `lab3/results/***.jpg`.
- Also submit your code `lab3/lab3.cu`. Again, you should only modify this file.
- We will test your code with the command listed in Listing 2..
- If you utilize multigrid or SOR, then you shell submit a report `lab3/report.pdf` to get the bonus.

5 Hint

Listing 3. is part of TA's code. Feel free to use it.



(a) 2 iterations

(b) 20 iterations

(c) 200 iterations

(d) 2000 iterations

(e) 6000 iterations

(f) 20000 iterations

Figure 4: The convergence with Jacobi Iteration.

```cpp
void PoissonImageCloning(
    const float *background,
    const float *target,
    const float *mask,
    float *output,
    const int wb, const int hb, const int wt, const int ht,
    const int oy, const int ox
) {
    // set up
    float *fixed, *buf1, *buf2;
    cudaMalloc(&fixed, 3*wt*ht*sizeof(float));
    cudaMalloc(&buf1, 3*wt*ht*sizeof(float));
    cudaMalloc(&buf2, 3*wt*ht*sizeof(float));

    // initialize the iteration
    dim3 gdim(CeilDiv(wt,32), CeilDiv(ht,16)), bdim(32,16);
    CalculateFixed<<<gdim, bdim>>>(
        background, target, mask, fixed,
        wb, hb, wt, ht, oy, ox
    );
    cudaMemcpy(buf1, target, sizeof(float)*3*wt*ht, cudaMemcpyDeviceToDevice);

    // iterate
    for (int i = 0; i < 10000; ++i) {
        PoissonImageCloningIteration<<<gdim, bdim>>>(
            fixed, mask, buf1, buf2, wt, ht
        );
        PoissonImageCloningIteration<<<gdim, bdim>>>(
            fixed, mask, buf2, buf1, wt, ht
        );
    }

    // copy the image back
    cudaMemcpy(output, background, wb*hb*sizeof(float)*3, cudaMemcpyDeviceToDevice);
    SimpleClone<<<gdim, bdim>>>(
        background, buf1, mask, output,
        wb, hb, wt, ht, oy, ox
    );

    // clean up
    cudaFree(fixed);
    cudaFree(buf1);
    cudaFree(buf2);
}
```

Listing 3: Hint