15-468/668 Project 2: Poisson Blending

Due Tuesday, 2/15/11 at 11:59pm

This project is an introduction to solving linear systems in parallel on the GPU. In particular, you will iteratively find the solution to a Poisson equation of the form $A\mathbf{x} = \mathbf{b}$ in order to blend two images together.

Requirements:

You must implement basic Poisson blending as described in the lecture slides in order to seamlessly blend masked images on top of similar backgrounds.

The equation you will be solving is of the form $A\mathbf{x} = \mathbf{b}$, where A is the adjacency matrix that blends surrounding pixels, $\mathbf{x}$ is the target blended image, and $\mathbf{b}$ is the pre-computed gradient field of the foreground image. For this project, you will need to write kernel functions that compute the gradient and iteratively solve for the vector $\mathbf{x}$.

In order to store the gradient field, you can use an MAC-style grid, which means that you keep track of the gradients on the edges between pixels, but store them in adjacent grid cells. For example, pixel `[i,j]` might store the horizontal component of color gradient between pixel `[i-1,j]` and `[i,j]`, while it would also store the vertical gradient between pixel `[i,j-1]` and `[i,j]`.

The simplest method of solving the Poisson equation is by performing repeated Jacobi iterations. Because A is a predictable adjacency matrix, you don't need to explicitly store it as a sparse matrix; you can just look up the values of the adjacent pixels directly from the old working buffer.

Skeleton Code:

The main file that you are responsible for is `poisson.cu`, which contains the code relevant for blending the supplied foreground with the background. Upon initialization, it calls `computeGradients`, which should calculate the vector **b** for the supplied foreground image.

Once the application starts blending the foreground with the background, `runPoissonKernel` is called every frame, which should perform one Jacobi iteration on the working blend buffer, **x**. Each of these two functions should call their own helper kernel functions, which you will have to write and launch from scratch.

The Poisson solver makes extensive use of memory maps and Pixel Buffer Objects, so the classes `PixelBufferObject` and `MemoryMap` have been provided to abstract the OpenGL API calls. When you construct a `MemoryMap`, it opens a map between a PBO and CUDA. Kernel functions can access this memory directly with the pointer returned by `getPointer`. Once the `MemoryMap` loses scope and its destructor is called, the map closes.

It may help to compile the automatic code documentation – just run the command "`make docs`" in order to generate HTML files that outline the starter code.

Setup:

The C++ runtime libraries and CUDA binaries and libraries aren't visible by default, so before you can use the makefile to build the project, you'll need to add some folders to your `PATH` and `LD_LIBRARY_PATH`.

Add the following to your `PATH` variable: `"/usr/local/cs/cuda/open64/bin"`

Add the following to your LD_LIBRARY_PATH variable:
`"/usr/local/lib64:/usr/local/cs/cuda/lib:/usr/local/cs/cuda/lib64"`

Also, in case you use VIM and would like syntax highlighting (for the C++ portions of your code), add the following line to your `.vimrc`:

```
au BufNewFile,BufRead *.cu set ft=cpp
```

Hardware:

CUDA is proprietary and only works on NVIDIA hardware, so if you only have Intel or ATI graphics available to you, you'll have to work using one of the Gates machines in the 5th floor or 3000 clusters (to be specific, the ghcXX.ghc.andrew.cmu.edu machines, where XX is from 01 to 82). Makefiles for Mac OSX and Linux are also provided, so if you do have modern NVIDIA hardware, you can try using them with some minor adjustments.

Handin and grading:

By 11:59:59pm on Tuesday, February 15th (2/15/11), you must copy the code that builds on the Gates machines to your handin directory at `/afs/cs/academic/class/15668-s11/handin/YOUR_ANDREW_ID/p2`.

Before you can copy your code to the handin directory, however, you must receive credentials to the CS realm by running the command "`aklog cs.cmu.edu`". For convenience, you can add that command to your `.cshrc` or `.bashrc` file to automatically run it.

If you have any trouble accessing your handin directory, let the TA know immediately at dboehle@andrew.cmu.edu.

This project will be graded on the correctness of the Poisson blend; you must correctly implement kernels for both gradient computation and Jacobi iteration.

For every day that this project is late, 15 points will be deducted from the grade. If you have any special circumstances, let the professor or TA know as soon as possible.

Bonus:

Any speed improvements over the base requirements will earn you extra points, especially if you implement a better solver technique such as conjugate gradient.

Reference materials:

For writing code for CUDA threads, this overview will help serve as a refresher to the basic concepts: http://www.nvidia.com/content/cudazone/download/Getting_Started_w_CUDA_Training_NVISION08.pdf

For convenience, the official CUDA documentation is currently being hosted on the course website at http://www.cs.cmu.edu/afs/cs/academic/class/15668-s11/www/cuda-doc/, where you can access both the PDF manuals and the html documentation. This has documentation for the version of CUDA installed on the Gates machines.

One good overview of the Poisson blending technique other than the course slides is here: http://www.cs.brown.edu/courses/csci1950-g/results/proj2/edwallac/. It is based upon a subset of a fairly technical paper, so feel free to read the original paper if you want extra clarification or justification: http://www.cs.brown.edu/courses/cs195-g/asgn/proj2/resources/PoissonImageEditing.pdf