Real Time Visual Mapping and Localization on a GPU

18.337 Spring 2009
Brent Tweddle
tweddle@mit.edu

Introduction:

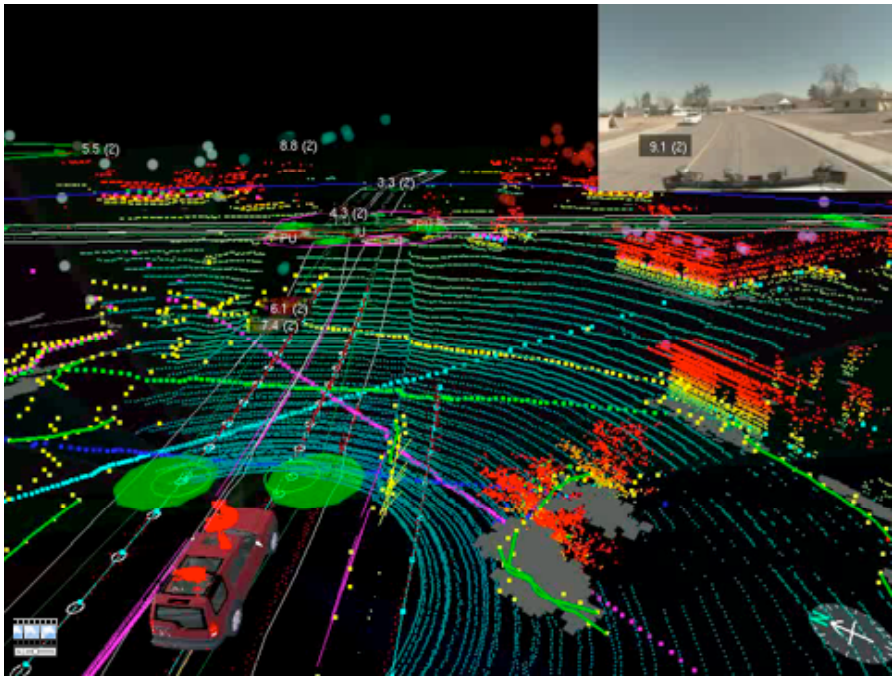
Autonomous robotics is concerned with designing robotic vehicles that can detect their environment and react to it in real-time, in the same way a human would. The vehicles entered in the DARPA Urban Challenge currently represent the state of the art in autonomous mobile robotics. In this competition, autonomous cars drove in a real urban environment along side other human drivers and robotic cars. Most of the cars successfully navigated the roads, and fully complied with California driving laws. MIT's entry to the Urban Challenge is pictured below.

In order to drive autonomously, a car must do three things: navigation, path planning and control. Navigation is the process of estimating the location of the vehicle and mapping the environment around it. Path planning is the act of determining a traversable route through the environment from the vehicles current location to a goal. Control is the act of applying the actuator forces to move the vehicle. In this project I will focus exclusively on a subset of the navigation problem.

Computational Requirements of Localization and Mapping:

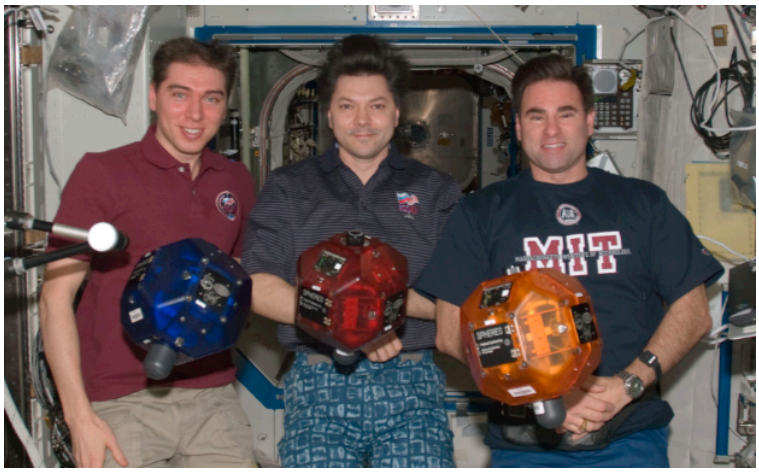
The only available information to the car comes from 5 cameras,12 single axis lidars, 1 rotating lidar and 15 radars (the sensor data is shown in the figure below). In the localization and mapping problem, GPS positioning is assumed to be unavailable, which commonly occurs in a variety of robotic applications including underground, outdoor and outer space environments.

As you can imagine, there is a lot of data that must be computed to generate a map and accurately local the vehicle in a complex 3D environment. The MIT Urban Challenge vehicle used a 10 Blade Server Cluster where each computer is a Quad-Core 2.3 GHz Xeon processor. This required a total power consumption of 4000 Watts, which required an additionally generator and air conditioner to be installed in the vehicle.



The Need for Power Efficient Implementations:

For many applications, especially aerospace, these mass and power requirements are simply not feasible. For example, the MIT SPHERES satellites (see below) are currently on the International Space Station. If these types of satellites were to employ the types of algorithms that were used on the DARPA Urban Challenge vehicle, they would need much more power efficient implementations.



The (typically aerospace) requirement to minimize the power consumption of complex artificial intelligent algorithms causes aerospace robotics engineers to look for power efficient hardware accelerators. This type of requirement has driven significant research on FPGA based computer vision [1]. However, given the complexity of a VHDL/Verilog computer vision implementation, these technologies were not considered for this project. An important figure of merit for selecting a hardware acceleration method is the Watts per GFLOP. The table below shows this metric for a number of common hardware platforms (note that FPGA's are not included since they do not have

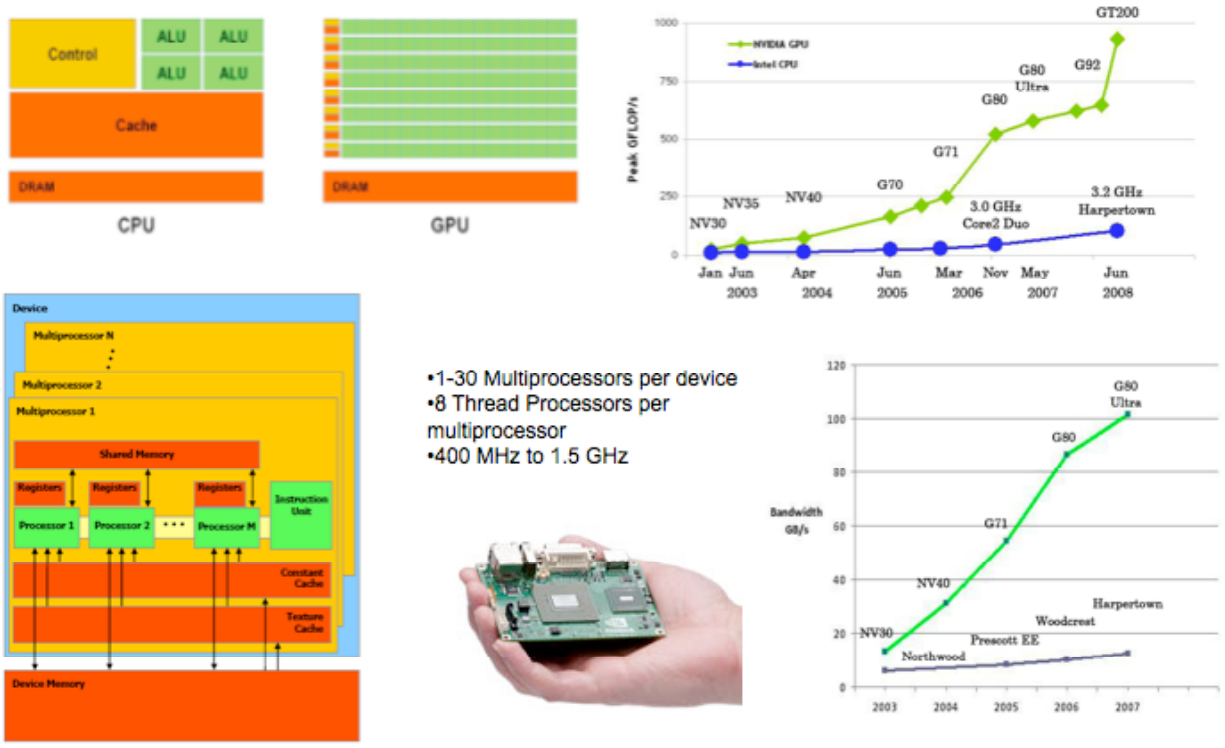
a well defined GFLOPS). It is important to note, that this is the theoretical peak GFLOPS and maximum thermal design power (TDP). In order to properly compare apples to apples, the same algorithm must be implemented on all platforms and measurements of both power and real world GFLOPS must be taken. Since this process is definitely too time consuming, I will use the Watts per GFLOP as a metric to select a processor architecture and compare my actual GFLOPS with the theoretical peak GFLOPS.

Processor	Theoretical Peak GFLOPS	Watts	Watts per GFLOPS
Quad "Bloomfield" Xeon 3.2 GHz	25.6 GFLOPS	130 W	5.078
Core 2 Duo "Penryn" 2.53 GHz	20.2 GFLOPS	25 W	0.810
Cell Processor	152 GFLOPS	80 W	0.526
NVIDIA Tesla C870	518 GFLOPS	170 W	0.328
NVIDIA GeForce 9800 GT	504 GFLOPS	105 W	0.208
NVIDIA GeForce 8800M GTS	240 GFLOPS	35 W	0.145

Since the GPU's claim to deliver better GFLOPS/Watt by an order of magnitude, it is worthwhile to determine whether they can be used.

Introduction to Graphics Processing Units and CUDA

Graphics processing units were originally designed to accelerate 3D video games and graphics applications using libraries such as OpenGL or DirectX. These are intrinsically data parallel operations, and hence GPU's have been designed with a large number of parallel algorithmic units, and very little control logic or on-chip cache (see figures below). This tradeoff means that very high GFLOPS are achievable, but at the expense of a much more difficult programming environment. Recently the CUDA programming environment has been released which allows the user to program NVIDIA GPU's in a restricted version of C. Much more details are available on CUDA, however it is important to realize that in order to obtain the maximum performance, memory, registers and parallel threads must be managed and fine-tuned by hand.



Dense Stereo Correspondence

A simultaneous localization and mapping algorithm would include 3 main steps:

- Dense Stereo Vision
- 3D Scan Matching
- Grid Map Estimation (Particle Filter)

For this project I will be implementing dense stereo vision algorithms, which generate depth maps based on a stereo camera system (see below). This algorithm typically contains a calibration stage that is followed by rectification, correspondence and post-processing. A large body of work exists on these methods [3,4,5].

Fig. 6. Results with simulated stereo images. (a) Virtual left stereo image. (b) By S&S DP without noise. (c) By WTA MW5 lr with noise. (d) By S&S DP with noise.

NVIDIA has provided an implementation of stereo correspondence [6], however it is a very "quick & dirty" implementation; in other words it runs very fast, but does not produce high quality results. A recently published paper [7] has shown how to implement a high quality stereo

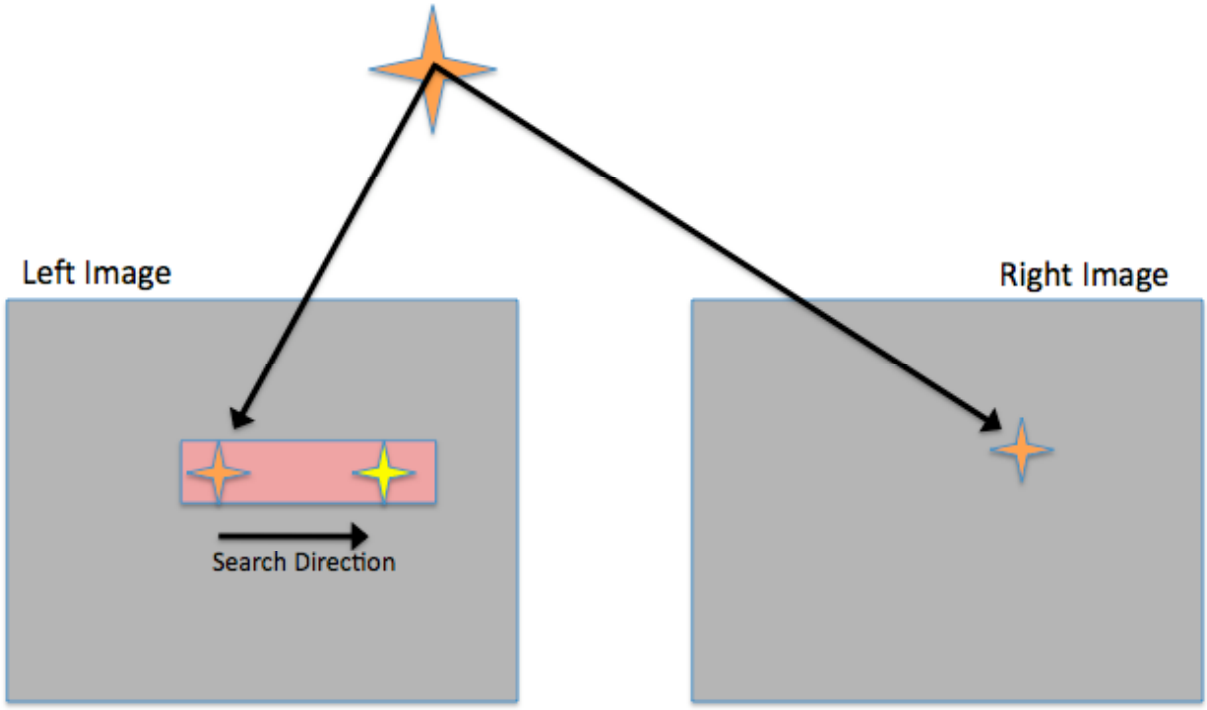
correspondence on SIMD CPU hardware such as SSE instruction sets. My primary goal in this project is to "upgrade" NVIDIA's code to produce high quality stereo correspondence without significantly increasing the algorithm execution time.

Stereo Correlation

Dense stereo correlation algorithms take two images of the same object from slightly different perspectives and attempt to match every pixel in one image with its corresponding pixel for the same object in the other image. For images that have been correctly rectified and are taken from perfectly parallel but offset positions, this search can be limited to a search along the baseline (or horizontal axis) in one direction (see image below). This results in significant computational savings for the algorithm.

The search is a minimization over the search direction of a cost function that signifies the matching error. This cost function is the Sum of Square Differences (SSD) over a small window (10x10 pixels).

In the NVIDIA sample code this is all that was implemented, however there are a number of improvements that can be made to refine this search, which I have implemented in my code.



Performing a Left-Right consistency check will improve accuracy at depth discontinuities [4]. This requires the same correlation and minimization to be performed on the left-to-right and right-to-left image. When this is complete, the results are verified and if they match, it further confirms a correct correlation; however if they do not match it indicates a problem (most likely an occlusion at a depth discontinuity) and the result is inconclusive.

A naïve implementation of the left-right consistency check will double the processing that needs to be done, however this is not necessary as all of the correlations have already been computed. In

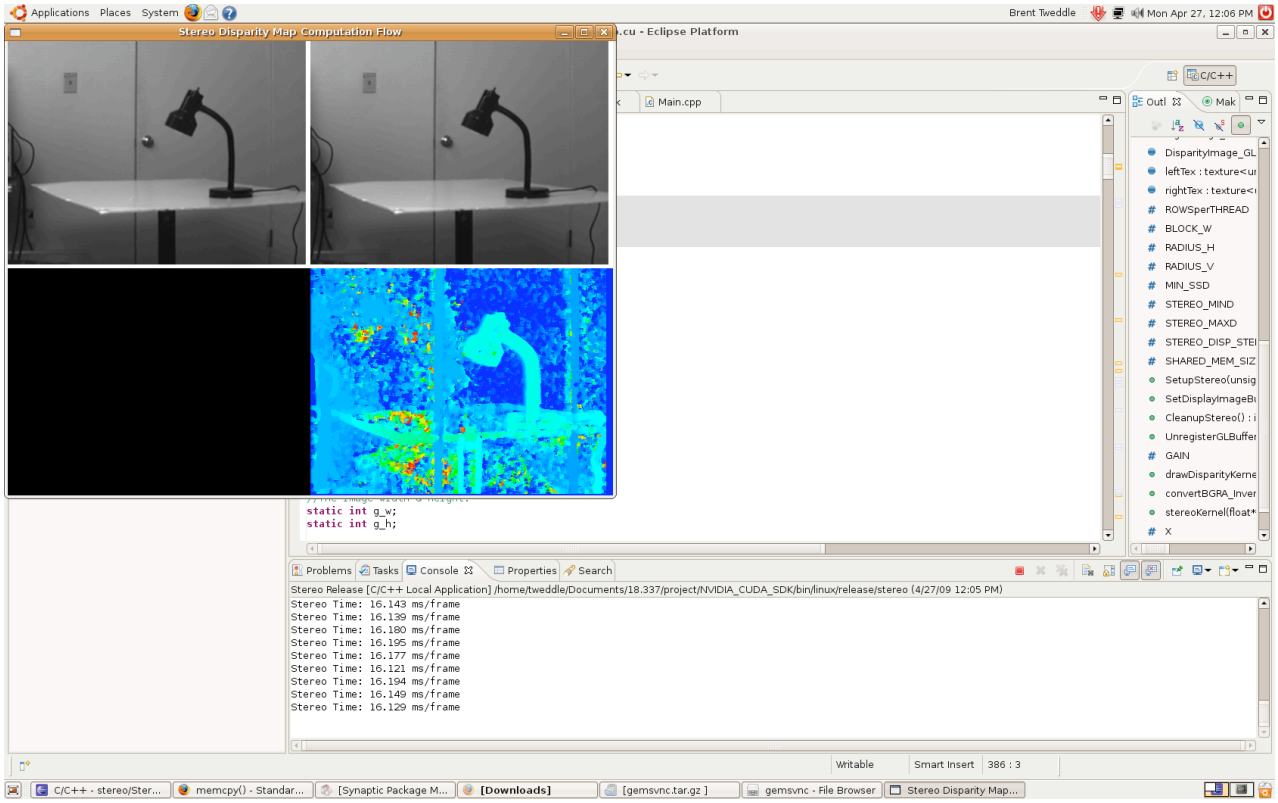
my final implementation, I have implemented a left right check without adding any extra correlations (however the memory requirements have significantly increased).

Additionally, if there is not enough texture in the surface to correctly match the images, the SSD will have a flat profile across the disparities. In order to check for a uniqueness test is performed where the new minimum value must be have a SSD value of 250 less than the previous best SSD.

First Steps in Stereo CUDA

The first step in my project is to get the stereo imaging code provided by NVIDIA to run on my computer. The source code is available along with a PDF that describes the algorithm's theory and optimized implementation. The first problem was that the source code is compiled for Windows and operates using a camera that I don't have. Since I want to run my algorithm on Linux as well as use stock images to evaluate accuracy, a significant "recompilation" effort was needed to get the code to this state.

The below image shows the results of this work; the code is running on Linux (within Eclipse), using OpenCV for file handling, and CUDA for stereo image processing. The two stereo pair images were downloaded from the web [8]. The bottom right window shows the disparity map where lighter colors represent points closer to the camera and darker colors represent points further from the camera.



The second step in my project was to time the original code. I have a GeForce 9800GT, which has 14 multiprocessors (112 Cores) that are running at 1.51 GHz. In the timing analysis I included the transfer time from the memory to the GPU as well as the computation of the disparity map;

however I did not include the time to draw the image on the screen (which is done on the graphics card by OpenGL).

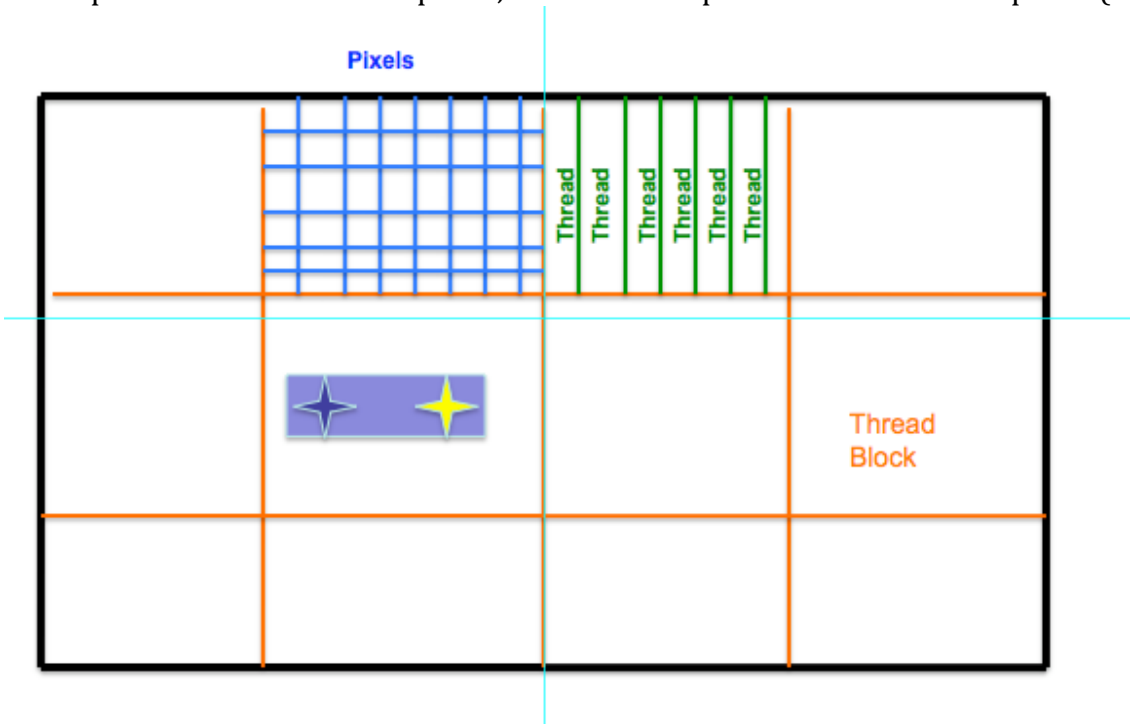
On my GPU, a 640x480 image takes approximately 16 ms, of which less than 1 ms is used for the memory transfers (CPU->GPU), and allocations take by commenting out the CUDA kernel code. I determined this by commenting out the kernel code and running the timing analysis. I tried improving the memory transfer time by using non-paged CPU memory, but it only resulted in a 0.2 second improvement.

A performance analysis of similar algorithms was published in [7] that used a 512x512 image on a 3.2 GHz Pentium 4. The standard OpenCV implementation took 10.6 seconds to run, however this used dynamic programming and is not a good comparison against the algorithm that I implemented.

In this paper another algorithm was implemented that is very similar to the NVIDIA stereo algorithm, and is highly optimized using SSE2 instructions on the Pentium processor. This algorithm took 87 ms to run the slightly smaller 512x512 image.

NVIDIA Stereo Approach

In the stereo sample provided by NVIDIA [6], a high speed approach is used. Each CUDA thread corresponds to one column of pixels, a block corresponds to a 2D block of pixels (see figure below.



At each iteration of a thread block, all of the pixels are offset by the same disparity value and their individual squared difference is calculated and stored in shared (local on chip) memory. The threads access their nearby squared differences to produce the sum of squared differences over the window size. This is compared to a minimum value in the disparity image, and if it is less than the stored value the disparity image is updated. The threads repeat this process for all disparity values within the search range.

Post Processing Upgrades

In order to implement the left right check without performing any additional correlations, we can store each (left to right) correlation in both the Left_Disparity{X,Y} and Right_Disparity{X-d,Y} location. In my implementation, rather than storing the disparity values in a 2D array, they are stored in a 3D grid, which is searched after all of the disparity values are calculated. Originally this was thought to be the only way to implement a LR check with subpixel resolution, however in retrospect it may be possible to avoid this.

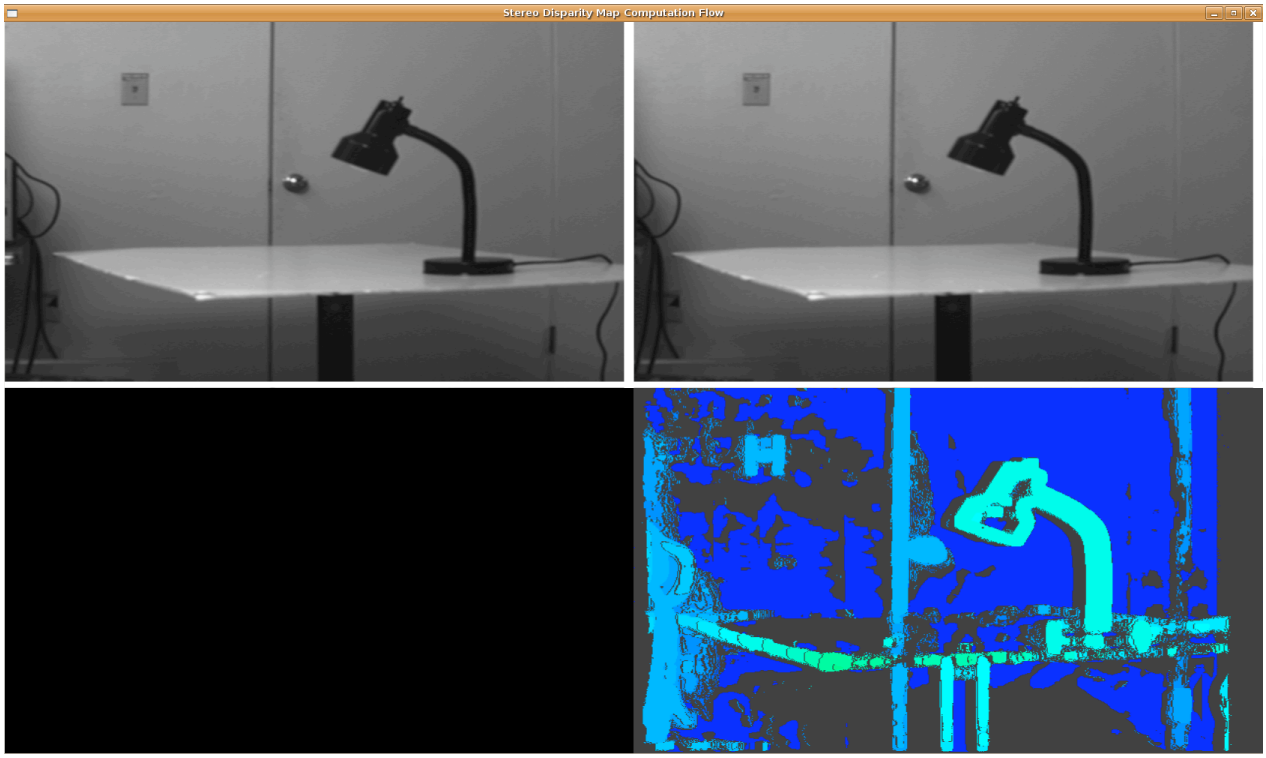
After all of the disparity values are calculated, each thread loops through each pixel in the column searching for the best disparity and checking the uniqueness ratio. Once this is done a Left-Right check can be performed.

Left-Right Check Race Conditions

Since each thread is executed in parallel and the order in which they are executed is not deterministic, race conditions may exist if pixels from other threads are being read and written to. I found that this was the case in my initial implementation, and it caused flickering in the disparity values of pixels as values were defined out of order.

To solve this, I created another CUDA kernel that has one thread block for each pixel. In this model, I read all of the pixels, synchronized the threads and then wrote back all of the LR corrections. This removed the race conditions and the flickering from the images.

The resulting image can be seen in the below figure. It is clear that this image has less noise and has removed (grayed out) occluded sections of the image.

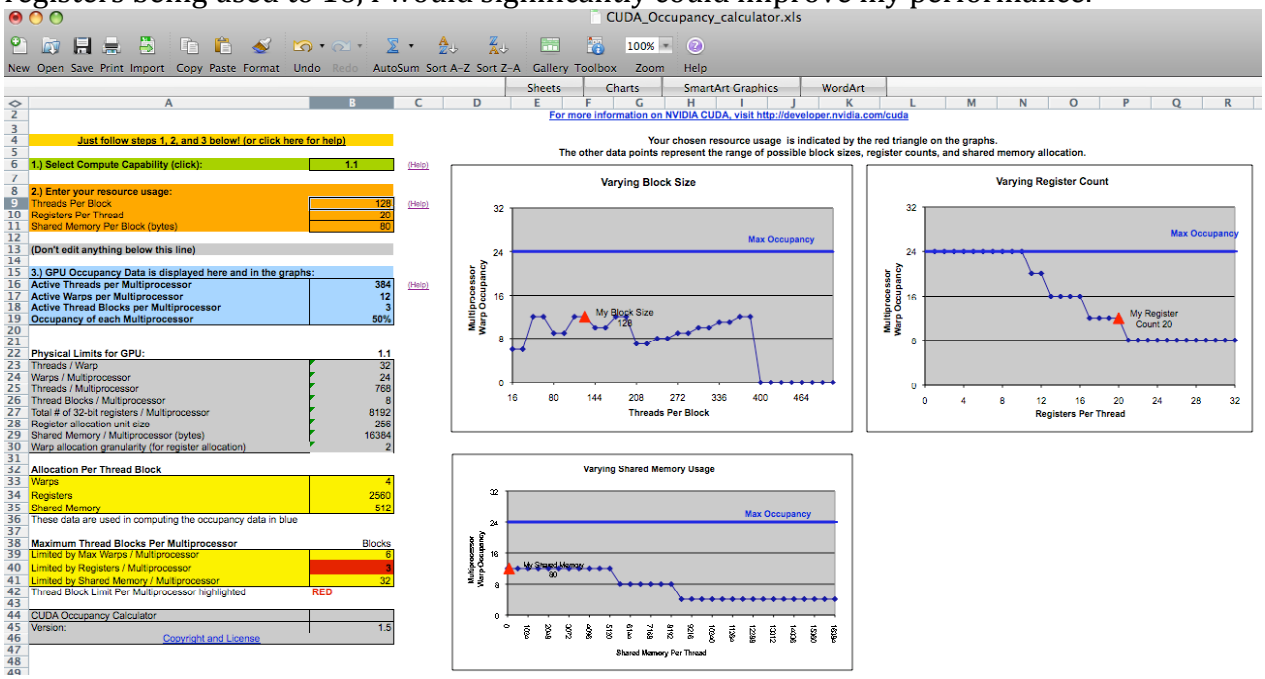


Performance Analysis

The improved algorithm with the 3D array and LR checks now runs in 25.7 seconds. Although this is a significant slowdown from the original code, it is still better than existing CPU algorithms.

Because the number of independent threads far exceeds the number of available thread processors, this code can hide the memory latency but not completely remove it. The main limiting factor in this application is memory latency. Since the total amount of data is relatively small the CUDA code is also not "memory bandwidth limited." Therefore the code's secondary limitation is by the number of concurrent threads that are being processed. This is referred to as the GPU occupancy, and NVIDIA has provided an Excel spreadsheet to analyze this. Each thread in my main kernel requires 20 registers, 80 bytes of shared memory, and 20 bytes of constant memory. If this exceeds the resources of each thread processor, another thread processor must be used to store the "spillover" data, which therefore decreases the number of thread processors being used for useful work.

The Occupancy Calculator can be seen in the below figure: I entered the compute version of my device (1.1), the number of registers used per thread (20), the number of bytes of shared memory (80) and the number of bytes of constant memory (80) and it indicates that I am being limited by the number of registers. The right hand plot shows that if I were to decrease the number of registers being used to 16, I would significantly could improve my performance.



Conclusion

In conclusion, I have shown that a GPU's would be usable for real-time stereo vision for Simultaneous Localization and Mapping. I showed a method to improve the stereo accuracy without significantly degrading performance. I analyzed and profiled this code to determine that its primary performance limitation was the number of registers required by each thread. Much further development is required to improve the implementation to where it can be usable on an actual vehicle. The CUDA source code for this algorithm is provided in the Appendix.

[1] Zhou, Wh., Du, X., Ye, Xq., Gu, Wk. (2005), "Binocular stereo vision system based on FPGA", Journal of Image and Graphics, Vol. 10 pp.1166-70.
[2] www.nvidia.com/cuda
[3]Scharstein, Szeliski "A Taxonomy and Evaluation of Dense Tow-Frame Stereo Correspondence Algorithms", IJCV 2002
[4]Brown, Burschka, "Advances in Computational Stereo", IEEE PAMI, 2003
[5] Bradski & Kaehler, "Learning OpenCV", OReilly 2008
[6] Stam, J. "Stereo Imaging with CUDA," MIT IAP 6.963 Website Link, Jan 2008
[7] Van der Mark, Gavrila, "Real-Time Dense Stereo for Intelligent Vehicles", IEEE Trans. ITS, 2006
[8] S. Birchfield and C. Tomasi, "Depth Discontinuities by Pixel-to-Pixel Stereo," Int Jorn. Computer Vision, 1999 (www.ces.clemson.edu/~stb/research/stereo_p2p)
[9] Hirschmuller, H; Innocent, P. R. and Garibaldi, J. "Real-Time Correlation-Based Stereo Vision with Reduced Border Errors" Int Jour. Computer Vision, 2002

Appendix: Source Code: Stereo.cu

```
/*
 * Copyright 1993-2007 NVIDIA Corporation.  All rights reserved.
 *
 * NOTICE TO USER:
 *
 * This source code is subject to NVIDIA ownership rights under U.S. and
 * international Copyright laws.  Users and possessors of this source code
 * are hereby granted a nonexclusive, royalty-free license to use this code
 * in individual and commercial software.
 *
 * NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
 * CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
 * IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
 * REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
 * MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
 * IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
 * OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
 * OF USE, DATA OR PROFITS,  WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
 * OR OTHER TORTIOUS ACTION,  ARISING OUT OF OR IN CONNECTION WITH THE USE
 * OR PERFORMANCE OF THIS SOURCE CODE.
 *
 * U.S. Government End Users.   This source code is a "commercial item" as
 * that term is defined at  48 C.F.R. 2.101 (OCT 1995), consisting  of
 * "commercial computer  software"  and "commercial computer software
 * documentation" as such terms are  used in 48 C.F.R. 12.212 (SEPT 1995)
 * and is provided to the U.S. Government only as a commercial end item.
 * Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
 * 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
 * source code with only those rights set forth herein.
 *
 * Any use of this source code in individual and commercial software must
 * include, in the user documentation and internal comments to the code,
 * the above Disclaimer and U.S. Government End Users Notice.
 */
// includes, system
// includes, system
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

#ifdef _WIN32
#  define WINDOWS_LEAN_AND_MEAN
#  include <windows.h>
#endif
#include <GL/glew.h>
#include <cuda.h>
#include <cutil.h>
#include <cudaGL.h>
#include <cuda_gl_interop.h>


#include "Stereo.h"

#define SQ(a) (__mul24(a,a))  // sad or ssd

int divUp(int a, int b)
{
              if(a%b == 0)
                 return a/b;
```

```
            else
               return a/b + 1;
}

//The Image width & height.
static int g_w;
static int g_h;

//Pointers to emory for the disparity value (d) and the current minimum SSD, also on the GPU:
//3d disparity map
static cudaPitchedPtr g_disparityLeft3D;
static cudaPitchedPtr g_disparityRight3D;
static cudaExtent dispExtent;

static float *g_disparityLeft;
static float *g_disparityRight;
static int *g_minSSD;
static size_t g_floatDispPitch;
static size_t g_floatDispPitchRight;

//Pointers to cudaArrays, which contain a copy of the image data for texturing
cudaArray * g_leftTex_array;
cudaArray * g_rightTex_array;

//These values store OpenGL buffer ID¬ís which are used to draw the image on the screen using CUDA¬ís OpenGL interop
capability.  These are set from the main program after the bufferes are created..
unsigned int LeftImage_GLBufferID;
unsigned int RightImage_GLBufferID;
unsigned int DisparityImage_GLBufferID;

//Declaration of the textures used for accessing the image data
texture<unsigned char, 2, cudaReadModeNormalizedFloat> leftTex;
texture<unsigned char, 2, cudaReadModeNormalizedFloat> rightTex;

//Defined constants:
#define ROWSperTHREAD 40   // the number of rows a thread will process
#define BLOCK_W 128                     // the thread block width
#define RADIUS_H 5              // Kernel Radius 5V & 5H = 11x11 kernel
#define RADIUS_V 5
#define MIN_SSD 500000   // The mimium acceptable SSD value
#define STEREO_MIND 0.0f   // The minimum d range to check
#define STEREO_MAXD 48.0f       // the maximum d range to check
#define STEREO_DISP_STEP 1.0f // the d step, must be <= 1 to avoid aliasing
#define SHARED_MEM_SIZE ((BLOCK_W + 2*RADIUS_H)*sizeof(int) ) // amount of shared memory used
#define DISP_VALLEY 10
#define UNIQUENESS_RATIO_THRESH 1.0005f


/********
 The function SetupStereo allocates GPU memory and sets critical parameters in stored in global variable.
 We pass the width & height into the function, but let CUDA allocate the host memory.
 Memory is allocated as pinned using cudaMallocHost or traditionally using malloc.
 Pinned memory can provide nearly double the transfer speed to the GPU, but is not always compatible
 with the video capture source, as it the case with the Point Grey camera driver.
 *********/
extern "C" void SetupStereo(unsigned int w, unsigned int h, unsigned char** leftHost, unsigned char** rightHost)
{

            g_w = w;
            g_h = h;

            if (cudaSuccess != CUDA_SAFE_CALL(cudaMallocHost((void**) leftHost, g_w*g_h))) {
               printf("Memory Allocation Error\n");
               exit(0);
```

```
            }

            if (cudaSuccess != CUDA_SAFE_CALL(cudaMallocHost((void**) rightHost, g_w*g_h))) {
               printf("Memory Allocation Error\n");
               exit(0);
            }

            dispExtent = make_cudaExtent(w*sizeof(float), h, (size_t) STEREO_MAXD);
            CUDA_SAFE_CALL(cudaMalloc3D(&g_disparityLeft3D, dispExtent));
            CUDA_SAFE_CALL(cudaMalloc3D(&g_disparityRight3D, dispExtent));

    CUDA_SAFE_CALL(cudaMallocPitch((void**)&g_disparityLeft,&g_floatDispPitch,w*sizeof(float),h));
    CUDA_SAFE_CALL(cudaMallocPitch((void**)&g_disparityRight,&g_floatDispPitchRight,w*sizeof(float),h));
            // CUDA_SAFE_CALL(cudaMallocPitch((void**)&g_minSSD, &g_floatDispPitch,w*sizeof(int),h));
            //left-right consistency check

            g_floatDispPitch /= sizeof(float);
            g_floatDispPitchRight /= sizeof(float);

            cudaChannelFormatDesc U8Tex = cudaCreateChannelDesc<unsigned char>();
            cudaMallocArray(&g_leftTex_array, &U8Tex, g_w, g_h);
            cudaMallocArray(&g_rightTex_array, &U8Tex, g_w, g_h);

            // for debug - show the memory on the GPU
            unsigned int free,total;
            cuMemGetInfo(&free,&total);
            printf("Memory After Allocation - Free: %d, Total: %d\n",free/(1024*1024),total/(1024*1024));
}

/*********
 SetDisplayImageBuffers
 Simply sets the OpenGL buffer IDs and registers them for CUDA GLinterop
 *********/
void SetDisplayImageBuffers (unsigned int Left, unsigned int Right,unsigned int Disparity)
{
            LeftImage_GLBufferID = Left;
            RightImage_GLBufferID = Right;
            DisparityImage_GLBufferID = Disparity;

            cudaGLRegisterBufferObject(LeftImage_GLBufferID);
            cudaGLRegisterBufferObject(RightImage_GLBufferID);
            cudaGLRegisterBufferObject(DisparityImage_GLBufferID);
}

/**********
 CleanupStereo - releses host & GPU memory allocated
 **********/
extern "C" int CleanupStereo()
{

            cudaFreeArray(g_leftTex_array);
            cudaFreeArray(g_rightTex_array);
            cudaFree(g_disparityLeft3D.ptr);
            cudaFree(g_disparityLeft);
            cudaFree(g_disparityRight3D.ptr);
            cudaFree(g_disparityRight);
            cudaFree(g_minSSD);
            return 0;
}

/**********
 Used at cleanup to unregister OpenGL buffers
 **********/
void UnregisterGLBufferForCUDA(int buffer)
```

```
{
    CUDA_SAFE_CALL(cudaGLUnregisterBufferObject(buffer));
}

/**********
 drawDisparityKernel()
 This function converts the floating point disparity values into a false color RGBA image.
 The image is allocated as an Op20enGL buffer from the main program
 *********/

#define GAIN (1.0f / STEREO_MAXD)

__global__ void drawDisparityKernel(   uchar4 * out_image,
                                                                        size_t out_pitch,
                                                                        float *disparity,
                                                                        size_t disparity_pitch,
                                                                        int width,
                                                                        int height)
{
    const int x = __mul24(blockIdx.x,blockDim.x) + threadIdx.x;
    const int y = __mul24(blockIdx.y,blockDim.y) + threadIdx.y;
             uchar4 temp;
    if(x < width && y < height) {
        float d = disparity[__mul24(y,disparity_pitch)+x];

                // first draw unmatched pixels in grey
        if(d == -1.0f)
                {
                        temp.x = 50;
                        temp.y = 50;
                        temp.z = 50;
                        temp.w = 255;
                }
                else
                {
                        float val = ((float)d)*GAIN;
                        float r = 1.0f;
                        float g = 1.0f;
                        float b = 1.0f;
                        if (val < 0.25f) {
                                r = 0;
                                g = 4.0f * val;
                        } else if (val < 0.5f) {
                                r = 0;
                                b = 1.0 + 4.0f * (0.25f - val);
                        } else if (val <  0.75f) {
                                r = 4.0f * (val - 0.5f);
                                b = 0;
                        } else {
                                g = 1.0f + 4.0f * (0.75f - val);
                                b = 0;
                        }
                        temp.x = 255.0 * r;
                        temp.y = 255.0 * g;
                        temp.z = 255.0 * b;
                        temp.w = 255;
                }
                out_image[__mul24(y,out_pitch)+x] = temp;
    }



}
```

```
/*******
 stereoKernel
 Now for the main stereo kernel:  There are five parameters:
 disparityPixel & disparityMinSSD point to memory containing the disparity value (d)
 and the current minimum sum-of-squared-difference values for each pixel.
 width & height are the image width & height, and out_pitch specifies the pitch of the output data in words
 (i.e. the number of floats between the start of one row and the start of the next.).
 *********/

__global__ void stereoKernel(cudaPitchedPtr disparityPitchedPtr,
                                                        cudaPitchedPtr disparityPitchedRightPtr,
                                                        cudaExtent disparityExtent,
                                                        float *disparityPixel,   // pointer to the output memory for the
disparity map
                                                        float *disparityPixelRight,   // pointer to the output memory
for the disparity map
                                                        //                                                      int
*disparityMinSSD,
                                                        int width,
                                                        int height,
                                                        size_t out_pitch)             // the pitch (in pixels) of the
output memory for the disparity map
{


              extern __shared__ int col_ssd[]; // column squared difference functions

              float d;     // disparity value
              int diff;  // difference temporary value
              int ssd;   // total SSD for a kernel
              float x_tex; // texture coordinates for image lookup
              float y_tex;
              int row;   // the current row in the rolling window
              int i;       // for index variable
              //int z;
              char* dispSlice;
              float* disparity_row;
              float best_ssd;
              float value;

              // use define's to save registers
#define X (__mul24(blockIdx.x,BLOCK_W) + threadIdx.x)
#define Y (__mul24(blockIdx.y,ROWSperTHREAD))

              // for threads reading the extra border pixels, this is the offset
              // into shared memory to store the values

              int extra_read_val = 0;
              if(threadIdx.x < (2*RADIUS_H)) extra_read_val = BLOCK_W+threadIdx.x;

              char* dispDevPtr = (char*) disparityPitchedPtr.ptr;
              size_t dispPitch = disparityPitchedPtr.pitch;
              size_t dispSlicePitch = __mul24(dispPitch,disparityExtent.height);

              char* dispDevRightPtr = (char*) disparityPitchedRightPtr.ptr;
              size_t dispPitchRight = disparityPitchedRightPtr.pitch;
              size_t dispSlicePitchRight = __mul24(dispPitchRight,disparityExtent.height);

              x_tex = X - RADIUS_H;
              for(d = STEREO_MIND; d < STEREO_MAXD; d += STEREO_DISP_STEP)
              {

                 col_ssd[threadIdx.x] = 0;
```

```
		if(extra_read_val>0)   col_ssd[extra_read_val] = 0;

		// do the first row
		y_tex = Y - RADIUS_V;

		for(i = 0; i <= 2*RADIUS_V; i++)
		{
			diff = (int)(255.0f*tex2D(leftTex,x_tex,y_tex)) - (int)(255.0f*tex2D(rightTex,x_tex-d,y_tex));
			col_ssd[threadIdx.x] += SQ(diff);

			if(extra_read_val > 0)
			{
				diff = (int)(255.0f*tex2D(leftTex,x_tex+BLOCK_W,y_tex)) -
(int)(255.0f*tex2D(rightTex,x_tex+BLOCK_W-d,y_tex));
				col_ssd[extra_read_val] += SQ(diff);
			}
			y_tex += 1.0f;
		}
		__syncthreads();

		// now accumulate the total
		if(X < width && Y < height)
		{
			ssd = 0;
			for(i = 0;i<=(2*RADIUS_H);i++)
			{
				ssd += col_ssd[i+threadIdx.x];
			}

			// the 1.0e-5 factor is a bias due to floating point error in the accumulation of the SSD value
			//                     if( ssd  < disparityMinSSD[__mul24(Y,out_pitch) + X])
			//                     {
			dispSlice = dispDevPtr + __mul24(d,dispSlicePitch);
			disparity_row = (float*) (dispSlice + __mul24((Y),dispPitch));
			disparity_row[X] = ssd;

			if (X-d > 0)
			{
				dispSlice = dispDevRightPtr + __mul24(d,dispSlicePitchRight);
				disparity_row = (float*) (dispSlice + __mul24((Y),dispPitchRight));
				disparity_row[(unsigned int) (X-(unsigned int)d)] = ssd;
			}
		}
		__syncthreads();

		// now do the remaining rows
		y_tex = Y - RADIUS_V; // this is the row we will remove
		for(row = 1;row < ROWSperTHREAD && (row+Y < (height+RADIUS_V)); row++)
		{
			// subtract the value of the first row from column sums
			diff = (int)(255.0f*tex2D(leftTex,x_tex,y_tex)) - (int)(255.0f*tex2D(rightTex,x_tex-d,y_tex));
			col_ssd[threadIdx.x] -= SQ(diff);

			// add in the value from the next row down
		    diff = (int)(255.0f*tex2D(leftTex,x_tex,y_tex + (float)(2*RADIUS_V)+1.0f)) -
(int)(255.0f*tex2D(rightTex,x_tex-d,y_tex + (float)(2*RADIUS_V)+1.0f));
			col_ssd[threadIdx.x] += SQ(diff);

			if(extra_read_val > 0)
			{
				diff =  (int)(255.0f*tex2D(leftTex,x_tex+(float)BLOCK_W,y_tex)) -
(int)(255.0f*tex2D(rightTex,x_tex-d+(float)BLOCK_W,y_tex));
				col_ssd[extra_read_val] -= SQ(diff);
```

```
                        diff = (int)(255.0f*tex2D(leftTex,x_tex+(float)BLOCK_W,y_tex +
(float)(2*RADIUS_V)+1.0f)) - (int)(255.0f*tex2D(rightTex,x_tex-d+(float)BLOCK_W,y_tex + (float)(2*RADIUS_V)+1.0f));
                        col_ssd[extra_read_val] += SQ(diff);
                }
                y_tex += 1.0f;
            __syncthreads();

                if(X<width && (Y+row) < height)
                {
                        ssd = 0;

                        for(i = 0;i<=(2*RADIUS_H);i++)
                        {
                                ssd += col_ssd[i+threadIdx.x];
                        }
                        //                              if(ssd < disparityMinSSD[__mul24(Y+row,out_pitch) + X])
                        //                              {
                        dispSlice = dispDevPtr + __mul24(d,dispSlicePitch);
                        disparity_row = (float*) (dispSlice + __mul24((Y+row),dispPitch));
                        disparity_row[X] = ssd;

                        if (X-d > 0)
                        {
                                dispSlice = dispDevRightPtr + __mul24(d,dispSlicePitchRight);
                                disparity_row = (float*) (dispSlice + __mul24((Y+row),dispPitchRight));
                                disparity_row[(unsigned int)(X-(unsigned int)d)] = ssd;
                        }

                        //                                      disparityPixel[__mul24(Y+row,out_pitch) + X] = d;
                        //                                      disparityMinSSD[__mul24(Y+row,out_pitch) + X] =
ssd;
                        //                              }
                }
                __syncthreads();  // wait for everything to complete
          } // for row loop
        } // for d loop

        if(X<width )
        {

           for(i = 0;i<ROWSperTHREAD && Y+i < height;i++)
           {
                   disparityPixel[__mul24((Y+i),out_pitch)+X] = -1.0f;

                   dispSlice = dispDevPtr + __mul24(STEREO_MAXD-1,dispSlicePitch);
                   disparity_row = (float*) (dispSlice + __mul24((Y+i),dispPitch));
                   best_ssd = disparity_row[X];

                   for(d = STEREO_MIND; d < STEREO_MAXD; d += STEREO_DISP_STEP)
                   {
                           dispSlice = dispDevPtr + __mul24(d,dispSlicePitch);
                           disparity_row = (float*) (dispSlice + __mul24((Y+i),dispPitch));
                           value = disparity_row[X];
                           if (value < (best_ssd-DISP_VALLEY))
                           {
                                   disparityPixel[__mul24((Y+i),out_pitch)+X] = d;
                                   best_ssd = value;
                           }

                   }

                   disparityPixelRight[__mul24((Y+i),out_pitch)+X] = -1.0f;

                   dispSlice = dispDevRightPtr + __mul24(STEREO_MAXD-1,dispSlicePitchRight);
```

```
                        disparity_row = (float*) (dispSlice + __mul24((Y+i),dispPitchRight));
                        best_ssd = disparity_row[X];

                        for(d = STEREO_MIND; d < STEREO_MAXD; d += STEREO_DISP_STEP)
                        {
                                dispSlice = dispDevRightPtr + __mul24(d,dispSlicePitchRight);
                                disparity_row = (float*) (dispSlice + __mul24((Y+i),dispPitchRight));
                                value = disparity_row[X];
                                if (value < (best_ssd -DISP_VALLEY))
                                {
                                        disparityPixelRight[__mul24((Y+i),out_pitch)+X] = d;
                                        best_ssd = value;
                                }

                        }

                  }
                }
                /*
                 __syncthreads();
                 if(X<width )
                 {
                 for(i = 0;i<ROWSperTHREAD && Y+i < height;i++)
                 {
                 d = disparityPixel[__mul24((Y+i),out_pitch)+X];
                 if ((X-d) < 0)
                 {
                 disparityPixel[__mul24((Y+i),out_pitch)+X] = -1.0f;
                 }
                 else if (disparityPixelRight[__mul24((Y+i),out_pitch)+X-(unsigned int)d] != d)
                 {
                 disparityPixel[__mul24((Y+i),out_pitch)+X] = -1.0f;
                 //                                disparityPixelRight[__mul24((Y+i),out_pitch)+X-(unsigned int)d] = -1.0f;
                 }

                 }
                 }
                 */
                __syncthreads();
}

__global__ void LRCheck(        float *disparityPixel,   // pointer to the output memory for the disparity map
                                        float *disparityPixelRight,   // pointer to the output memory for the
disparity map
                                        size_t out_pitch)              // the pitch (in pixels) of the output
memory for the disparity map
{
#define XX (blockIdx.x)
#define YY (blockIdx.y)
                float d;
                int check = 0;

                d = disparityPixel[__mul24(YY,out_pitch)+XX];
                if ((XX-d) < 0)
                {
                   check = 1;
                }
                else if (disparityPixelRight[__mul24(YY,out_pitch)+XX-(unsigned int)d] != d)
                {
                   check = 1;
                   //                                disparityPixelRight[__mul24((YY+i),out_pitch)+XX-(unsigned int)d] = -
1.0f;
                }
                __syncthreads();
```

```
            if (check == 1) {
               disparityPixel[__mul24(YY,out_pitch)+XX] = -1.0f;
            }
}

/*********
 stereoProcess
 The main function called for every frame is stereoProcess.
 p_hostLeft & p_hostRight contain teh source data
 **********/
extern "C" float stereoProcess(unsigned char * p_hostLeft,unsigned char * p_hostRight, int ImageType)
{

            unsigned char * GLtemp;

            unsigned int timer;
            dim3 grid(1,1,1);
            dim3 threads(16,8,1);
            dim3 gridLR(1,1,1);
            dim3 threadsLR(16,8,1);

            threads.x = BLOCK_W;
            threads.y = 1;
    grid.x = divUp(g_w, BLOCK_W);
            grid.y = divUp(g_h,ROWSperTHREAD);



            threadsLR.x = 1;
            threadsLR.y = 1;
            gridLR.x = g_w;
            gridLR.y = g_h;
            //printf("Copying: %d, %d\n",g_w, g_h);
            // Greyscale Image, just copy it.
            // printf("Copy Location: %X\n", (int) p_hostLeft);

            CUT_SAFE_CALL(cutCreateTimer(&timer));
            CUT_SAFE_CALL(cutStartTimer(timer));

            cudaMemcpyToArray(g_leftTex_array, 0, 0, p_hostLeft,g_w * g_h, cudaMemcpyHostToDevice);
            cudaMemcpyToArray(g_rightTex_array, 0, 0, p_hostRight,g_w * g_h, cudaMemcpyHostToDevice);

            // Set up the texture parameters for bilinear interpolation & clamping
            leftTex.filterMode = cudaFilterModeLinear;
            cudaBindTextureToArray(leftTex, g_leftTex_array);
            rightTex.filterMode = cudaFilterModeLinear;
            cudaBindTextureToArray(rightTex, g_rightTex_array);


            stereoKernel<<<grid,threads,SHARED_MEM_SIZE>>>(g_disparityLeft3D,g_disparityRight3D,dispExtent,g_disparit
yLeft,g_disparityRight, /*g_minSSD,*/g_w,g_h,g_floatDispPitch);
            //stereoKernel<<<grid,threads,SHARED_MEM_SIZE>>>(g_disparityLeft,g_minSSD,g_disparityLeft_Check,g_minSSD_
Check,g_w,g_h,g_floatDispPitch);

            cudaThreadSynchronize();
            LRCheck<<<gridLR,threadsLR>>>(g_disparityLeft,g_disparityRight,g_floatDispPitch);
            cudaThreadSynchronize();

            cudaUnbindTexture(leftTex);
            cudaUnbindTexture(rightTex);


            CUT_SAFE_CALL(cutStopTimer(timer));  // don't time the drawing
            float retval = cutGetTimerValue(timer);
```

```
            // now for OpenGL's copy to display
            cudaGLMapBufferObject( (void**)&GLtemp, LeftImage_GLBufferID);
            cudaMemcpyFromArray(GLtemp,g_leftTex_array,0,0,g_w*g_h,cudaMemcpyDeviceToDevice);
            cudaGLUnmapBufferObject(LeftImage_GLBufferID);

            cudaGLMapBufferObject( (void**)&GLtemp, RightImage_GLBufferID);
            cudaMemcpyFromArray(GLtemp,g_rightTex_array,0,0,g_w*g_h,cudaMemcpyDeviceToDevice);
            cudaGLUnmapBufferObject(RightImage_GLBufferID);

            // Now draw the disparity map
            threads.x = 16;
            threads.y = 8;
            grid.x = divUp(g_w , threads.x);
            grid.y = divUp(g_h , threads.y);
            uchar4 * DisparityMap;
            size_t DispPitch;
            cudaMallocPitch((void**)&DisparityMap,&DispPitch,g_w*sizeof(uchar4),g_h);

            drawDisparityKernel<<<grid,threads>>>(DisparityMap,DispPitch/sizeof(uchar4),g_disparityLeft,g_floatDispPi
tch,g_w,g_h);
            cudaThreadSynchronize();
            CUT_CHECK_ERROR("Kernel execution failed writing disparity map!");

            // now copy the output for openGL
            cudaGLMapBufferObject( (void**)&GLtemp, DisparityImage_GLBufferID);
            cudaMemcpy2D(GLtemp,g_w*sizeof(uchar4),DisparityMap,DispPitch,g_w*sizeof(uchar4),g_h,cudaMemcpyDeviceToDe
vice);
            cudaGLUnmapBufferObject(DisparityImage_GLBufferID);
            cudaFree(DisparityMap);
            return retval;

}
```

Appendix: Source Code: Main.cpp

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <cv.h>
#include <highgui.h>

#include <cuda.h>
#include <cutil.h>

// includes for OpenGL
#include <GL/glew.h>
#include <GL/glut.h>


#include "Stereo.h"

//#include "AVIReader.h"

// Define this symbol if a point grey camera is used. The user must provide
// digiclops.lib & triclops.lib along with the corresponding dll's from the Point Grey SDK
//#define ALLOW_POINT_GREY 1

// Camera aquisition and processing parameters
static unsigned ImageWidth = 630;
static unsigned ImageHeight = 480;

static GLuint LeftImageBuffer = 0;                    // This is an identifier of a OpenGL Buffer Object which will
hold the image to be displayed
static GLuint RightImageBuffer = 0;
static GLuint DispImageBuffer = 0;

static GLuint LeftImageTexture = 0;        // We also need an OpenGL texture object to draw the image.  We will bind
the data referenced by ImageBuffer to this texture
static GLuint RightImageTexture = 0;
static GLuint DispImageTexture = 0;

static GLuint GLWindowWidth = 800;
static GLuint GLWindowHeight = 600;

unsigned char * leftImg;
unsigned char * rightImg;

static int frame_count;  // counts the number of frames for timing
static float time_avg;

// forward declerations for glut callback functions
void display(void);
void idle(void);
void keyboard( unsigned char key, int x, int y);
void reshape (int x, int y);

bool use_camera = false;

//AVIReader * LeftAVI;
//AVIReader * RightAVI;
IplImage* LeftImg;
IplImage* RightImg;


////////////////////////////////////////////////////////////////////////////
// Program main
```

```
////////////////////////////////////////////////////////////////////////////
int
main( int argc, char** argv)
{

              LeftImg = 0;
              RightImg = 0;
              printf("Hello World\n");

   // First parse command line to see if we are using a camera or file
  if(argc > 3) {
/*                if(strcmp( "-c",argv[1]) == 0)
                  {
                     use_camera = true;
                     ImageWidth = atoi(argv[2]);
                     ImageHeight = atoi(argv[3]);
                  }
*/                if(strcmp( "-f",argv[1]) == 0)
                  {
                     use_camera = false;
/*
                     LeftAVI = new AVIReader(argv[2]);
                     RightAVI = new AVIReader(argv[3]);
                     if(!LeftAVI->IsOpen())
                     {
                            printf("Error Opening Left File\n");
                            exit(0);
                     }
                      if(!RightAVI->IsOpen())
                     {
                            printf("Error Opening Right File\n");
                            exit(0);
                     }

                     ImageWidth = LeftAVI->Width();
                     ImageHeight = LeftAVI->Height();
                     if(RightAVI->Width() != ImageWidth || RightAVI->Height() != ImageHeight)
                     {
                            printf("Video Files are not of the same size\n");
                            exit(0);
                     }
                     LeftAVI->loop = true;
                     RightAVI->loop = true;
*/
                     printf("loading images\n");
                     LeftImg = cvLoadImage(argv[2], 0);
                     RightImg = cvLoadImage(argv[3], 0);
                     if (LeftImg == 0 || RightImg == 0) {
                            printf("Error Loading Images\n");
                            exit(0);
                     }
                     ImageWidth = LeftImg->width;
                     ImageHeight = LeftImg->height;

                     printf("Width, Height, Depth: %d, %d, %d\n", ImageWidth, ImageHeight, LeftImg->depth);
                  }
   }
   else
   {
                  printf("       StereoCamera.exe -f [left filename] [right filename] \n");
                  exit(0);
   }
  printf("Setting Up Stereo\n");
   // Allocate GPU memory
```

```
   SetupStereo(ImageWidth, ImageHeight, &leftImg, &rightImg);

   if(!use_camera)
   {

   }


            // Now we are going to setup the OpenGL Utility Library Stuff to create our window and respond to user
I/O
            glutInit(&argc,argv);
            glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
            glutInitWindowSize(GLWindowWidth, GLWindowHeight);
            glutCreateWindow("Stereo Disparity Map Computation Flow");
            glutDisplayFunc(display);               // set function pointer to call on display
            glutKeyboardFunc(keyboard);             // set function pointer to call on keyboard input
            glutReshapeFunc(reshape);
            glutIdleFunc(idle);                             // set function pointer to call on Idle
            glewInit(); // initi the extension wrangler to allow use of the buffer objects
            if (!glewIsSupported("GL_VERSION_2_0 GL_VERSION_1_5 GL_ARB_vertex_buffer_object
GL_ARB_pixel_buffer_object")) {
               fprintf(stderr, "Required OpenGL extensions missing.");
               exit(-1);
            }


            // Now we need to create a couple memory buffers for drawing.  OpenGL
            // and CUDA will share these buffers. OpenGL Buffer Objects are simply
            // blocks of memory allocated on the Graphics card.  These buffers can contain
            // pixel data, vertex data, etc.  Using CUDA's OpenGL Interop capabilities
            // these buffers become accessable for read/write access.  When done, the buffers
            // can be returned to OpenGL control and used to draw whatever was placed into them

            // The following code is all OpenGL stuff to create and allocate the buffers, no CUDA yet.

            // Create a set of OpenGL Buffers to hold the image data we wish to draw
            unsigned int size = ImageWidth * ImageHeight *  sizeof(GLubyte);  // Image will be in 8-bit greyscale
format
            glGenBuffers( 1, &LeftImageBuffer);  // This creates the buffer
            glBindBuffer( GL_PIXEL_UNPACK_BUFFER, LeftImageBuffer);  // this make the buffer current
            glBufferData( GL_PIXEL_UNPACK_BUFFER, size, NULL, GL_DYNAMIC_DRAW);  // this allocates the proper size
and sets the data null
            glGenBuffers( 1, &RightImageBuffer);
            glBindBuffer( GL_PIXEL_UNPACK_BUFFER, RightImageBuffer);
            glBufferData( GL_PIXEL_UNPACK_BUFFER, size, NULL, GL_DYNAMIC_DRAW);

            size = ImageWidth * ImageHeight *  sizeof(GLubyte) * 4;
            glGenBuffers( 1, &DispImageBuffer);
            glBindBuffer( GL_PIXEL_UNPACK_BUFFER, DispImageBuffer);
            glBufferData( GL_PIXEL_UNPACK_BUFFER, size, NULL, GL_DYNAMIC_DRAW);

            glBindBuffer( GL_PIXEL_UNPACK_BUFFER, 0);  // this make the buffer current

            // Left Image Texture
            glGenTextures( 1, &LeftImageTexture);  // Create the texture reference
            glBindTexture( GL_TEXTURE_2D, LeftImageTexture);    // Make it current
            glTexImage2D( GL_TEXTURE_2D, 0, GL_LUMINANCE, ImageWidth, ImageHeight, 0, GL_LUMINANCE, GL_UNSIGNED_BYTE,
NULL); // Actually allocate the memory
            // Right Image Texture
            glGenTextures( 1, &RightImageTexture);  // Create the texture reference
            glBindTexture( GL_TEXTURE_2D, RightImageTexture);    // Make it current
            glTexImage2D( GL_TEXTURE_2D, 0, GL_LUMINANCE, ImageWidth, ImageHeight, 0, GL_LUMINANCE, GL_UNSIGNED_BYTE,
NULL); // Actually allocate the memory
            // Disparity Map Texture
```

```
            glGenTextures( 1, &DispImageTexture);  // Create the texture reference
            glBindTexture( GL_TEXTURE_2D, DispImageTexture);    // Make it current
            glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA, ImageWidth, ImageHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL); //
Actually allocate the memory

            glBindTexture(GL_TEXTURE_2D, 0);

            // pass these buffer ID's for the stereo algorithm and register them for CUDA interop
            SetDisplayImageBuffers(LeftImageBuffer,RightImageBuffer,DispImageBuffer);

            // zero the timer variables
            time_avg = 0;
            frame_count = 0;

            glutMainLoop();  // that's all the setup!  now kick off the main loop


}

void reshape(int x, int y) {
    glViewport(0, 0, x, y);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0, 1, 1, 0, -1, 1);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glutPostRedisplay();
}

int count = 0;

/* This is the main function where the stereo process is kicked off */
void display(void) {

            // count how many times we run for timing
            count++;
            int ImgType;

            // !!This where image aquisition occurs!!

            // Get a frame.  For the point grey camera, the pointers to host memory
            // where the images get stored were set during the call to ptGreyCam::init
            // in the main function.
   if(!use_camera)
   {

//              leftImg = LeftAVI->GetNextFrame();
//              rightImg = RightAVI->GetNextFrame();

                memcpy(leftImg, LeftImg->imageData, ImageWidth*ImageHeight);
                memcpy(rightImg, RightImg->imageData, ImageWidth*ImageHeight);

//              leftImg = (unsigned char*) LeftImg->imageData;
//              rightImg = (unsigned char*) RightImg->imageData;


                if(leftImg == NULL || rightImg == NULL)
                {
                        printf("Video Stopped\n");
                        exit(0);
                }
                ImgType = IMAGE_TYPE_GRAY_U8;

   }
```

```
            time_avg += stereoProcess(leftImg, rightImg, ImgType);


            // Now we Draw!
            glDisable(GL_DEPTH_TEST);
            glEnable(GL_TEXTURE_2D);
            glClear(GL_COLOR_BUFFER_BIT);  // Clear the screen

            glBindTexture( GL_TEXTURE_2D, LeftImageTexture);  // Select the Image Texture
    glBindBuffer( GL_PIXEL_UNPACK_BUFFER, LeftImageBuffer); // And Bind the OpenGL Buffer for the pixel data
    glTexSubImage2D( GL_TEXTURE_2D, 0, 0, 0, ImageWidth, ImageHeight, GL_LUMINANCE, GL_UNSIGNED_BYTE, 0);  // Set the
texture parameters
            glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);  // then unbind

            glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
            glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
            glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
            glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

            // Now just draw a square on the screen and texture it with the image texture
            glBegin( GL_QUADS);
               glVertex3f(0.0,0.0,0.5);
               glTexCoord2f(1.0, 0.0);

               glVertex3f(0.5,0.0,0.5);
               glTexCoord2f(1.0, 1.0);

               glVertex3f(0.5,0.5,0.5);
               glTexCoord2f( 0.0, 1.0);

               glVertex3f(0.0,0.5,0.5);
               glTexCoord2f( 0.0,0.0);

            glEnd();

            // right image
            glBindTexture( GL_TEXTURE_2D, RightImageTexture);  // Select the Image Texture
    glBindBuffer( GL_PIXEL_UNPACK_BUFFER, RightImageBuffer); // And Bind the OpenGL Buffer for the pixel data
    glTexSubImage2D( GL_TEXTURE_2D, 0, 0, 0, ImageWidth, ImageHeight, GL_LUMINANCE, GL_UNSIGNED_BYTE, 0);  // Set the
texture parameters
            glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);  // then unbind

            glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
            glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
            glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
            glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
            glBegin( GL_QUADS);
               glVertex3f(0.5,0.0,0.5);
               glTexCoord2f( 1.0, 0.0);

               glVertex3f(1.0,0.0,0.5);
               glTexCoord2f(1.0, 1.0);

               glVertex3f(1.0,0.5,0.5);
               glTexCoord2f( 0.0, 1.0);

               glVertex3f(0.5,0.5,0.5);
               glTexCoord2f( 0.0,0.0);

            glEnd();

            // disparity map
            glBindTexture( GL_TEXTURE_2D, DispImageTexture);  // Select the Image Texture
```

```
    glBindBuffer( GL_PIXEL_UNPACK_BUFFER, DispImageBuffer); // And Bind the OpenGL Buffer for the pixel data
    glTexSubImage2D( GL_TEXTURE_2D, 0, 0, 0,ImageWidth, ImageHeight, GL_RGBA, GL_UNSIGNED_BYTE, 0);  // Set the texture
parameters
              glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);  // then unbind

              glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
              glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
              glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
              glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
              glBegin( GL_QUADS);
                 glVertex3f(0.5,0.5,0.5);
                 glTexCoord2f( 1.0, 0.0);

                 glVertex3f(1.0,0.5,0.5);
                 glTexCoord2f(1.0, 1.0);

                 glVertex3f(1.0,1.0,0.5);
                 glTexCoord2f( 0.0, 1.0);

                 glVertex3f(0.5,1.0,0.5);
                 glTexCoord2f( 0.0,0.0);

              glEnd();

              glBindTexture( GL_TEXTURE_2D, 0);  // Select the Image Texture
    glutSwapBuffers();
              frame_count++;
              if(frame_count == 10)
              {
                 char s[256];
                 time_avg /= 10.0f;
#ifdef _WIN32
                 sprintf_s(s, "Stereo Time: %3.3f ms/frame\n",time_avg);
#else
         sprintf(s, "Stereo Time: %3.3f ms/frame\n",time_avg);
#endif
                 printf(s);
                 time_avg = 0;
                 frame_count = 0;
    }

    glutPostRedisplay();
}

void idle(void) {
    glutPostRedisplay();
}

void keyboard( unsigned char key, int x, int y) {
    switch( key) {
        case 27:
                        CleanupStereo();
                        UnregisterGLBufferForCUDA(LeftImageBuffer);
                        UnregisterGLBufferForCUDA(RightImageBuffer);
                        UnregisterGLBufferForCUDA(DispImageBuffer);
                        if(use_camera)  // only need to free memory if we are using the point grey camera.  Open CV
manages it's own
                        {
                                if(leftImg != NULL) free(leftImg);
                                if(rightImg != NULL) free(rightImg);
                        }
                        exit(0);
                        break;
        default: break;
```

```
    }
    glutPostRedisplay();
}
```