```c
#include <time.h>
#include "phiADT.h"

// Local Method Declarations
void parse_cmdLineArgs(int argc, char *argv[]);

// Global variables
char *fileToRead; // name of the file to read
char *prefix;     // prefix for the output file

/*
 * Main method requires two command line arguments
 * Usage: program <file_to_process> <output_prefix>
 */
int main (int argc, char *argv[]) {

        time_t start, stop;
        time(&start);

        parse_cmdLineArgs(argc, argv);

        // open the file as read-only
        FILE *inputFile = fopen( fileToRead, "r" );
        if ( inputFile == NULL ) exit(1);

        char fileName[255];
        sprintf(fileName, "%s_distfield", prefix);

        Phi *phiFncn = phi_create(inputFile);
        phi_calc_distance_field(phiFncn);
        phi_gen_file(phiFncn, VTI|DAT, fileName);
        phi_destroy(phiFncn);

        fclose(inputFile);

        time(&stop);
        printf("\n-----------------------------\n");
        printf("Finished in about %f s \n", difftime(stop, start));
        printf("-----------------------------\n\n");

        return EXIT_SUCCESS;
}

/*
 * Method to parse and check the command line
 * arguments passed when running the code.
 * Arguments:
 *        int [in] - total number of arguments passed
 *   char* [] [in] - pointer to char array
 * Returns:
 */
void parse_cmdLineArgs(int argc, char *argv[]) {
        if ( argc != 3 ){
                printf("\nUsage: distance_field <file_to_process> <output_prefix>\n\n");
                exit(EXIT_FAILURE);
        }

        fileToRead = argv[1];
        prefix = argv[2];
}
```

```c
/*
 * phiADT.h
 * Abstract Data Type for the Distance Field
 * Preprocessor.
 *
 *  Created on: Jun 04, 2015
 *      Author: Anup Shrestha
 *
 *  Audit Trail:
 *     Date:
 *  Changes:
 */
#ifndef PHIADT_H_
#define PHIADT_H_

#include <stdio.h>
#include <stdlib.h>
#include "file_writer.h"

// Type Definitions
typedef struct phi_type Phi;

// Method Declarations
Phi *phi_create(FILE *);
void phi_destroy(Phi *);
void phi_calc_distance_field(Phi *);
void phi_gen_file(Phi *, FileOut, char *);

#endif /* PHIADT_H_ */
```

```
/*
 * phi3D.h
 * Main header file for 3D Phi Function.
 *
 *  Created on: Jun 04, 2015
 *      Author: Anup Shrestha
 *
 *  Audit Trail:
 *     Date:
 *  Changes:
 */

#ifndef PHI3D_H_
#define PHI3D_H_

#include "phiADT.h"
#include "vti_parser.h"
#include <math.h>

/* speed of propagation [Equation (1) in the report] */
#define SPEED                    1
#define DEFAULT_BORDER_LOCATION  -1
#define DEFAULT_BORDER_DISTANCE  INFINITY
#define DEFAULT_INTERIOR_DISTANCE 90000

/*
 * Completing the struct declared in the
 * phiADT.h This definition is for a 3D phi.
 *  x, y, z    - interior dimensions
 *  dx, dy, dz - node spacing
 *  F          - speed
 *  location   - location values
 *  distance   - distance values
 */
struct phi_type {
  int      x, y, z;
  double   dx, dy, dz, F;
  int    * location;
  double * distance;
};

#endif /* PHI3D_H_ */
```

```c
/*
 * phi3D.c
 * This is a wrapper file that calls appropriate
 * methods from other different files that:
 *   1) Creates and initializes the phi function
 *      variables
 *   2) Calculates the distance field
 *   3) Writes the data to file
 *   4) Performs cleanup
 *
 *  Created on: Jun 04, 2015
 *      Author: Anup Shrestha
 *
 *  Audit Trail:
 *     Date:
 *  Changes:
 */
#include "phi3D.h"

// private method declarations
static void adjust_boundary(Phi *p, Grid3D gridEx);
static int linear3dIndex(int x, int y, int z, int max_x, int max_y);


// external method declarations
extern void init_phiFncn(Phi *p, double *d);
extern void free_phiFncn(Phi *p);
extern void calc_distfield(Phi *p, Grid3D g3d);


/*
 * Creates phi function, calls the file parser
 * methods to get the dimensions, location
 * and distance data from the file. Also,
 * allocates memory and initializes variables
 * required by phi function.
 * Arguments:
 *      FILE* [in] - pointer to a file
 * Returns:
 *   Phi* - pointer to phi function
 */
Phi* phi_create(FILE *vti) {

        Phi *p = (Phi *) malloc(sizeof(Phi));

        // parsing the information about the dimensions
        // and the spacing of the nodes from the file and
        // storing them in an array
        double *dims = (double *) malloc(sizeof(double) * 6);
        vti_get_dimensions(vti, dims);

        // now using the information about the dimensions
        // and the spacing create a phi function by
        // initializing all the attributes and allocating
        // space for the location and distance data
        init_phiFncn(p, dims);

        // the phi function stores the internal dimensions
        // so store the external dimensions in a global
        // Grid3D struct variable for future use.
        // Grid3D and make_grid3D are defined in vti_parser.h
        Grid3D gridEx = make_grid3D(p->x + 2, p->y + 2, p->z + 2);

        // now parse the location and distance data from
        // the file and store it in appropriate arrays
        // inside the phi function.
        vti_get_data(vti, p->location, DEFAULT_BORDER_LOCATION,
                                p->distance, DEFAULT_BORDER_DISTANCE, gridEx);
```



```c
        return p;
}

/*
 * Calls the method that deallocates
 * memory allocated for the phi function.
 * Arguments:
 *    Phi* [in] - pointer to phi function
 * Returns:
 */
void phi_destroy(Phi *p) {

        free_phiFncn(p);

}

/*
 * Calls the method that calculates the distance
 * field.
 * Arguments:
 *    Phi* [in/out] - pointer to phi function
 * Returns:
 */
void phi_calc_distance_field(Phi *p) {

        Grid3D gridEx = make_grid3D(p->x + 2, p->y + 2, p->z + 2);
        calc_distfield(p, gridEx);

}


/*
 * Updates the boundary values of the distance
 * array, then calls the method that writes
 * distance field to different files based on
 * the FileOut parameter
 * Arguments:
 *      Phi* [in] - pointer to phi function
 *   FileOut [in] - Output file type enumerator
 *     char* [in] - name of the output file
 * Returns:
 */
void phi_gen_file(Phi *p, FileOut out, char *fileName) {

  Grid3D gridEx = make_grid3D(p->x + 2, p->y + 2, p->z + 2);
  adjust_boundary(p, gridEx);
  FileGrid fg = make_fileGrid(gridEx.x, gridEx.y, gridEx.z, p->dx, p->dy, p->dz)
;
  FileType ft = make_fileType(fileName, p->distance, out, fg);

  // generate file(s)
  file_generate(&ft);

}

/*
 * Adjusts the boundary values
 * Arguments:
 *      Phi* [in/out] - pointer to phi function
 *   Grid3D [in]     - dimensions of grid
 * Returns:
 */
static void adjust_boundary(Phi *p, Grid3D gridEx) {
        // before writing to file we need to update the
  // boundary values
  int x, y, z, i, j, k;
  x = gridEx.x;
  y = gridEx.y;
  z = gridEx.z;
```

```
  for(i = 0; i < z; i++){
    for(j = 0; j < y; j++){
      for(k = 0; k < x; k++){
        int I = i, J = j, K = k;
        I = (i == z-1) ? I-1 : (!i) ? I+1 : I;
        J = (j == y-1) ? J-1 : (!j) ? J+1 : J;
        K = (k == x-1) ? K-1 : (!k) ? K+1 : K;
        if( i != I || j != J || k != K) {
                p->distance[linear3dIndex(k, j, i, x, y)] = p->distance[linear3d
Index(K, J, I, x, y)];
        }
      }
    }
  }
}

/*
 * Convert 3D indexing to 1D indexing
 * Arguments:
 *   int x, y, z [in] - 3D coordinate
 *   int max_x   [in] - size of x-dimension
 *   int max_y   [in] - size of y-dimension
 * Returns:
 */
static int linear3dIndex(int x, int y, int z, int max_x, int max_y) {
  return z * max_y * max_x + y * max_x + x;
}
```

```
/*
 * phi3D_cuda.h
 * Phi CUDA header file.
 *
 *  Created on: Jun 04, 2015
 *      Author: Anup Shrestha
 *
 *  Audit Trail:
 *     Date:
 *  Changes:
 */

#ifndef PHI3D_CUDA_H_
#define PHI3D_CUDA_H_

  #include "phi3D.h"

  /*
   * Structure for storing the cudaExtent and the
   * cudaPitchedPtr for the 3D array memory in
   * device (GPU) for location and distance.
   */
  typedef struct  {
    cudaExtent loc_ext, dst_ext;
    cudaPitchedPtr loc_dPitchPtr, dst_dPitchPtr;
  } Phi3D_d;

  /*
   * Structure for storing information used during
   * sweeping to manage internal grid dimensions,
   * sweep directions, position of the node on the
   * array and its offset in each kernel block
   */
  typedef struct {
    int level;
    int xDim, yDim, zDim;
    int xOffSet, yOffset;
    int xSweepOff, ySweepOff, zSweepOff;
    double dx, dy, dz;
  } SweepInfo;

  // Macro for checking CUDA errors following a CUDA launch or API call
  #define cudaCheckError() {\
          cudaError_t e = cudaGetLastError();\
          if( e != cudaSuccess ) {\
                  printf("\nCuda failure %s:%d: '%s'\n",__FILE__,__LINE__,cudaGetErrorS
tring(e));\
                  exit(EXIT_FAILURE);\
          }\
  }

  // External Linkage Method Declarations
  extern "C" {
    void init_phiFncn(Phi *p, double *d);
    void free_phiFncn(Phi *p);
    void calc_distfield(Phi *p, Grid3D g3d);
  }

#endif /* PHI3D_CUDA_H_ */
```

```
/*
 * phi3D_cuda.cu
 * Phi3D CUDA source file.
 *
 * Parallel 3D implementation of Fast Sweeping Method
 * using CUDA C/C++.
 * Takes the information about locations and IB distances
 * from a VTI file and propagates it. The algorithm
 * implemented for parallel fast sweeping method is from
 * a paper in the Journal of Computational Physics titled
 * "A parallel fast sweeping method for the Eikonal Equation"
 * by Miles Detrixhe, Federic Gibou, and Chohong Min.
 * DOI: http://www.sciencedirect.com/science/article/pii/S002199911200722X
 *
 *  Created on: Jun 04, 2015
 *      Author: Anup Shrestha
 *
 *  Audit Trail:
 *     Date:
 *  Changes:
 */
#include "phi3D_cuda.h"


static int iDivUp(int a, int b) {
        return ( (a % b) != 0 ) ? (a / b + 1) : (a / b);
}

// private method declarations
static void update_distance(int *loc_d, double *dst_d, Grid3D g3d);
static void fast_sweep(Phi* p, int noOfTimesToSweep, cudaPitchedPtr dst_dPitchPt
r);
static void set_distance_negative_inside(Phi3D_d *d_p, Grid3D g3d);
static void cudaMemcpy3D_P2D(Phi3D_d *d_p, double *dst, int *loc);
static void cudaMemcpy3D_D2P(Phi3D_d *d_p, double *dst, int *loc);

// Kernel Declarations
__global__ void update_distance_kernel(int *loc, double *dst, int totalNodes);
__global__ void set_distance_negative_inside_kernel(cudaPitchedPtr dst_dPitchPtr
, cudaPitchedPtr loc_dPitchPtr, Grid3D g3d);
__global__ void fast_sweep_kernel(cudaPitchedPtr dst_dPitchPtr, SweepInfo s);
__device__ double solve_eikonal(double cur_dist, double minX, double minY, doubl
e minZ, double dx, double dy, double dz);


/*
 * Initializes all the attributes of the phi function.
 * Also allocates pinned memory for location and
 * distance arrays. This function is called after the
 * input VTI file has been processed, and the dimensions
 * of the grid are known.
 * Arguments:
 *    Phi3D [out] - pointer to phi function
 *   double* [in] - pointer to array of double that
 *                  has the dimension and spacing values
 * Returns:
 */
void init_phiFncn(Phi *p, double *d) {

        p->x = (int) d[0] + 1; p->dx = d[3];
        p->y = (int) d[1] + 1; p->dy = d[4];
        p->z = (int) d[2] + 1; p->dz = d[5];
        p->F = SPEED;

        // allocating pinned memory for the
        // location and distance arrays
        int totalNodes = (p->x + 2) * (p->y + 2) * (p->z + 2);
        size_t l_arr = sizeof(int) * totalNodes;
        size_t d_arr = sizeof(double) * totalNodes;
```



```
        cudaHostAlloc((void **)&p->location, l_arr, cudaHostAllocMapped);
        cudaCheckError();
        cudaHostAlloc((void **)&p->distance, d_arr, cudaHostAllocMapped);
        cudaCheckError();

}

/*
 * Deallocates memory used by the phi function
 * Arguments:
 *    Phi* [in] - pointer to phi function
 * Returns:
 */
void free_phiFncn(Phi *p) {
        cudaFreeHost(p->location); cudaCheckError();
        cudaFreeHost(p->distance); cudaCheckError();
        free(p);
}


/*
 * This method updates the distance array based
 * on the location values. Then allocates 3D
 * memory on the device, copies the memory from
 * pinned memory to device memory, runs the fast
 * sweeping method, sets the inside distance to
 * negative based on the location values and
 * copies the device memory back to pinned memory.
 * Arguments:
 *     Phi* [in] - pointer to phi function
 *   Grid3D [in] - dimensions of the grid
 * Returns:
 */
void calc_distfield(Phi *p, Grid3D g3d) {

        // get the device pointers to the pinned memory
        int *loc_d; double *dst_d;
        cudaHostGetDevicePointer(&loc_d, p->location, 0); cudaCheckError();
        cudaHostGetDevicePointer(&dst_d, p->distance, 0); cudaCheckError();

        update_distance(loc_d, dst_d, g3d);

        /*
         * Setup for running the fast sweeping method on the
         * device (GPU). For faster performance, we want to
         * allocate memory for the arrays using cudaMalloc3D.
         * Even though the pinned memory has a device pointer
         * and can be accessed directly by kernels, it has not
         * been allocated using cudaMalloc3D, hence it degrades
         * performance. Therefore, we need to:
         * 1) Allocated memory using cudaMalloc3D
         * 2) Copy the pinned memory data to that memory
         * 3) Perform calculations
         * 4) Copy the data back to pinned memory
         * 5) Deallocate device memory
         */

        Phi3D_d* d_p = (Phi3D_d *) malloc (sizeof(Phi3D_d));

        /* (1) */
        d_p->loc_ext = make_cudaExtent(g3d.x * sizeof(int), g3d.y, g3d.z);
        d_p->dst_ext = make_cudaExtent(g3d.x * sizeof(double), g3d.y, g3d.z);

        cudaMalloc3D(&d_p->loc_dPitchPtr, d_p->loc_ext); cudaCheckError();
        cudaMalloc3D(&d_p->dst_dPitchPtr, d_p->dst_ext); cudaCheckError();

        /* (2) */
        cudaMemcpy3D_P2D(d_p, dst_d, loc_d);
```

```
        /* (3) */
        fast_sweep(p, 3, d_p->dst_dPitchPtr);

        set_distance_negative_inside(d_p, g3d);

        /* (4) */
        cudaMemcpy3D_D2P(d_p, dst_d, loc_d);

        /* (5) */
        cudaFree(d_p->loc_dPitchPtr.ptr);
        cudaFree(d_p->dst_dPitchPtr.ptr);
        free(d_p);

}



/*
 * Sets up the kernel call that updates the distance
 * array based on the location values.
 * Arguments:
 *      int* [in]     - location array in device
 *   double* [in/out] - distance array in device
 *    Grid3D [in]     - full grid dimensions
 * Returns:
 */
static void update_distance(int *loc_d, double *dst_d, Grid3D g3d) {

        // Setup 3D-Grid and 3D-Block for Kernel launch
        // Running 256 threads per block
        dim3 bs(8, 8, 8);
        dim3 gs(iDivUp(g3d.x, bs.x), iDivUp(g3d.y , bs.y), iDivUp(g3d.z, bs.z));

        int totalNodes = g3d.x * g3d.y * g3d.z;
        update_distance_kernel<<<gs, bs>>>(loc_d, dst_d, totalNodes);
        cudaThreadSynchronize(); cudaCheckError();

}

/*
 * Kernel that updates the distance array based
 * on the location values.
 * Arguments:
 *      int* [in]     - location array in device
 *   double* [in/out] - distance array in device
 *       int [in]     - total number of nodes
 */
__global__ void update_distance_kernel(int *loc, double *dst, int totalNodes) {
        int blockId = blockIdx.x + blockIdx.y * gridDim.x + gridDim.x * gridDim.
y * blockIdx.z;
        int tid = blockId * (blockDim.x * blockDim.y * blockDim.z) +
                            (threadIdx.z * (blockDim.x * blockDim.y))+
                            (threadIdx.y * blockDim.x) + threadIdx.x;

        if (tid < totalNodes)  {
                int l = loc[tid];
                double d = dst[tid];
                if(l != DEFAULT_BORDER_LOCATION &&
                   d != DEFAULT_BORDER_DISTANCE ) {
                        dst[tid] = (l == 1 && d == INFINITY) ? -1 : (d > 0.0 ||
d < 0.0) ? d : DEFAULT_INTERIOR_DISTANCE;
                }
        }
}

/*
 * Set up for the parallel FSM implementation
 * Determines the total number of levels and
```



```
 * the nodes on each level that are executed
 * in parallel.
 * Then determines the direction of the sweep,
 * chooses an offset, to translate the coordinates
 * for sweeping from different directions.
 * Finally, calls the CUDA kernel that calculates
 * distance field for a 3D grid using Gauss-Seidal
 * iterations.
 * (1) i = 1:I, j = 1:J, k = 1:K
 * (2) i = I:1, j = 1:J, k = K:1
 * (3) i = I:1, j = 1:J, k = 1:K
 * (4) i = 1:I, j = 1:J, k = K:1
 * (5) i = I:1, j = J:1, k = K:1
 * (6) i = 1:I, j = J:1, k = 1:K
 * (7) i = 1:I, j = J:1, k = K:1
 * (8) i = I:1, j = J:1, k = 1:K
 *
 * This is the sweeping step discussed in
 * section 2.3 of the report.
 * Arguments:
 *                Phi* [in]     - pointer to the phi function
 *                 int [in]     - number of sweep iterations
 *   cudaPitchedPtr [in/out] - pointer to distance array in
 *                             device memory
 *      Returns:
 */
static void fast_sweep(Phi* p, int noOfTimesToSweep, cudaPitchedPtr dst_dPitchPt
r) {
        // Information regarding sweeping and linear indexing
        int meshDim = 3;
        SweepInfo sw;
        sw.xDim = p->x; sw.dx = p->dx;
        sw.yDim = p->y; sw.dy = p->dy;
        sw.zDim = p->z; sw.dz = p->dz;

        int totalLevels = sw.xDim + sw.yDim + sw.zDim;

        // loop till the number of times to sweep
        int fastSweepLoopCount = 1;
        while( fastSweepLoopCount <= noOfTimesToSweep){
                printf("Please wait. Sweeping...[%d/%d]\n", fastSweepLoopCount, noOfTimes
ToSweep);
                for(int swCount = 1; swCount <= 8; ++swCount){
                        int start = (swCount == 2 || swCount == 5 || swCount ==
7 || swCount == 8 ) ? totalLevels : meshDim;
                        int end = ( start == meshDim ) ? totalLevels + 1 : meshD
im-1;
                        int incr = ( start == meshDim ) ? true : false;

                        // sweep offset is used for translating the 3D coordinat
es
                        // to perform sweeps from different directions
                        sw.xSweepOff = (swCount == 4 || swCount == 8 ) ? sw.xDim
 + 1 : 0;
                        sw.ySweepOff = (swCount == 2 || swCount == 6 ) ? sw.yDim
 + 1 : 0;
                        sw.zSweepOff = (swCount == 3 || swCount == 7 ) ? sw.zDim
 + 1 : 0;

                        for(int level = start; level != end; level = (incr) ? le
vel+1 : level-1){
                                int xs = max(1, level-(sw.yDim + sw.zDim)), ys =
 max(1,level-(sw.xDim + sw.zDim));
                                int xe = min(sw.xDim, level-(meshDim-1)),   ye =
 min(sw.yDim, level-(meshDim-1));

                                int xr = xe-xs + 1, yr = ye-ys + 1;
                                int tth = xr * yr; // Total number of threads ne
eded
```

```
                                dim3 bs(16, 16, 1);
                                if(tth < 256){
                                        bs.x = xr;
                                        bs.y = yr;
                                }
                                dim3 gs(iDivUp(xr, bs.x), iDivUp(yr , bs.y), 1);

                                sw.level = level;
                                sw.xOffSet = xs;
                                sw.yOffset = ys;

                                fast_sweep_kernel<<<gs, bs>>>(dst_dPitchPtr, sw)
;
                                cudaThreadSynchronize();
                                cudaCheckError();
                        }
                }
                printf("Sweeping finished!.....[%d/%d]\n", fastSweepLoopCount, noOfTimesT
oSweep);
                ++fastSweepLoopCount;
        }
}

/*
 * Kernel for fast sweeping method
 * Arguments:
 *   cudaPitchedPtr [in/out] - pointer to distance array in
 *                             device memory
 *        SweepInfo [in]     - sweep information
 */
__global__ void fast_sweep_kernel(cudaPitchedPtr dst_dPitchPtr, SweepInfo s) {
        int x = (blockIdx.x * blockDim.x + threadIdx.x) + s.xOffSet;
        int y = (blockIdx.y * blockDim.y + threadIdx.y) + s.yOffset;
        if(x <= s.xDim && y <= s.yDim) {
                int z = s.level - (x+y);
                if(z > 0 && z <= s.zDim){
                        int i = abs(z-s.zSweepOff);
                        int j = abs(y-s.ySweepOff);
                        int k = abs(x-s.xSweepOff);

                        char *devPtr = (char *) dst_dPitchPtr.ptr;
                        size_t pitch = dst_dPitchPtr.pitch;
                        size_t slicePitch = pitch * (s.yDim + 2);

                        double *c_row = (double *) ( (devPtr + i * slicePitch) +
 j * pitch);          // center row
                        double center = c_row[k];
                      // center distance
                        double left = c_row[k-1];
                      // left distance
                        double right = c_row[k+1];
                      // right distance
                        double up = ((double *) ( (devPtr + i * slicePitch) + (j
-1) * pitch) )[k];    // upper distance
                        double down = ((double *) ( (devPtr + i * slicePitch) +
(j+1) * pitch))[k];   // lower distance
                        double front = ((double *) ( (devPtr + (i-1) * slicePitc
h) + j * pitch) )[k]; // front distance
                        double back = ((double *) ( (devPtr + (i+1) * slicePitch
) + j * pitch) )[k];  // back distance

                        double minX = min(left, right);
                        double minY = min(up, down);
                        double minZ = min(front, back);
                        c_row[k] = solve_eikonal(center, minX, minY, minZ, s.dx,
 s.dy, s.dz);
                }
        }
```



```
}

/*
 * Solves the Eikonal equation at each point of the grid.
 * Arguments:
 *   double - current distance value
 *   double - minimum distance in the x-direction
 *   double - minimum distance in the y-direction
 *   double - minimum distance in the z-direction
 *   double - spacing in the x-direction
 *   double - spacing in the y-direction
 *   double - spacing in the z-direction
 */
__device__ double solve_eikonal(double cur_dist, double minX, double minY, doubl
e minZ, double dx, double dy, double dz) {
        double dist_new = 0;
        double m[] = { minX, minY, minZ} ;
        double d[] = { dx, dy, dz};

        // sort the mins
        for(int i = 1; i < 3; i++){
                for(int j = 0; j < 3-i; j++) {
                        if(m[j] > m[j+1]) {
                                double tmp_m = m[j];
                                double tmp_d = d[j];
                                m[j] = m[j+1]; d[j] = d[j+1];
                                m[j+1] = tmp_m; d[j+1] = tmp_d;
                        }
                }
        }

        // simplifying the variables
        double m_0 = m[0], m_1 = m[1], m_2 = m[2];
        double d_0 = d[0], d_1 = d[1], d_2 = d[2];
        double m2_0 = m_0 * m_0, m2_1 = m_1 * m_1, m2_2 = m_2 * m_2;
        double d2_0 = d_0 * d_0, d2_1 = d_1 * d_1, d2_2 = d_2 * d_2;

        dist_new = m_0 + d_0;
        if(dist_new > m_1) {

          double s = sqrt(- m2_0 + 2 * m_0 * m_1 - m2_1 + d2_0 + d2_1);
          dist_new = ( m_1 * d2_0 + m_0 * d2_1 + d_0 * d_1 * s) / (d2_0 + d2_1);

          if(dist_new > m_2) {

            double a = sqrt(- m2_0 * d2_1 - m2_0 * d2_2 + 2 * m_0 * m_1 * d2_2
                            - m2_1 * d2_0 - m2_1 * d2_2 + 2 * m_0 * m_2 * d2_1
                            - m2_2 * d2_0 - m2_2 * d2_1 + 2 * m_1 * m_2 * d2_0
                            + d2_0 * d2_1 + d2_0 * d2_2 + d2_1 * d2_2);

            dist_new = (m_2 * d2_0 * d2_1 + m_1 * d2_0 * d2_2 + m_0 * d2_1 * d2_
2 + d_0 * d_1 * d_2 * a) /
                       (d2_0 * d2_1 + d2_0 * d2_2 + d2_1 * d2_2);
          }
        }

        return min(cur_dist, dist_new);
}


/*
 * Sets up the kernel call to set the inside distance
 * values in the array to negative based on location.
 * Arguments:
 *   Phi3D_d* [in/out] - pointer to phi function (GPU)
 *     Grid3D [in]     - dimensions of grid
 * Returns:
 */
static void set_distance_negative_inside(Phi3D_d *d_p, Grid3D g3d) {
```

```
        // Setup 3D-Grid and 3D-Block for Kernel launch
        // Running 256 threads per block
        dim3 bs(8, 8, 8);
        dim3 gs(iDivUp(g3d.x, bs.x), iDivUp(g3d.y , bs.y), iDivUp(g3d.z, bs.z));
        set_distance_negative_inside_kernel<<<gs, bs>>>(d_p->dst_dPitchPtr, d_p-
>loc_dPitchPtr, g3d);
        cudaThreadSynchronize(); cudaCheckError();

}


/*
 * Kernel for fast sweeping method
 * Arguments:
 *   cudaPitchedPtr [out] - pointer to distance array in
 *                          device memory
 *   cudaPitchedPtr [in]  - pointer to location array in
 *                          device memory
 *           Grid3D [in]  - dimensions of grid
 */
__global__ void set_distance_negative_inside_kernel(cudaPitchedPtr dst_dPitchPtr
, cudaPitchedPtr loc_dPitchPtr, Grid3D g3d) {
        int x = blockIdx.x*blockDim.x+threadIdx.x + 1;
        int y = blockIdx.y*blockDim.y+threadIdx.y + 1;
        int z = blockIdx.z*blockDim.z+threadIdx.z + 1;

        if (x < g3d.x-1 &&  y < g3d.y-1 && z < g3d.z-1 )  {

                char *dist_devPtr = (char *) dst_dPitchPtr.ptr;
                size_t dist_pitch = dst_dPitchPtr.pitch;
                size_t dist_slicePitch = dist_pitch * g3d.y;
                char* dist_slice = dist_devPtr + z * dist_slicePitch;
                double* dist_row = (double *) (dist_slice + y * dist_pitch);

                char *loc_devPtr = (char *) loc_dPitchPtr.ptr;
                size_t loc_pitch = loc_dPitchPtr.pitch;
                size_t loc_slicePitch = loc_pitch * g3d.y;
                char* loc_slice = loc_devPtr + z * loc_slicePitch;
                int* loc_row = (int *) (loc_slice + y * loc_pitch);

                if(loc_row[x] == 1) dist_row[x] = -1;
        }
}


/*
 * Copies location and distance array memory
 * from device pinned memory to device memory
 * Arguments:
 *   Phi3D_d* [in/out] - pointer to d_Phi (Device - Destination)
 *    double* [in]     - pointer to distance (Pinned - Source)
 *          int* [in]     - pointer to location (Pinned - Source)
 * Returns:
 */
static void cudaMemcpy3D_P2D(Phi3D_d *d_p, double *dst, int *loc) {
        cudaMemcpy3DParms mcp = { 0 };
        mcp.kind = cudaMemcpyDeviceToDevice;

        // copy parameters for distance
        mcp.dstPtr = d_p->dst_dPitchPtr;
        mcp.srcPtr.ptr = dst;
        mcp.srcPtr.pitch = d_p->dst_ext.width;
        mcp.srcPtr.xsize = (size_t) (d_p->dst_ext.width/sizeof(double));
        mcp.srcPtr.ysize = d_p->dst_ext.height;
        mcp.extent = d_p->dst_ext;
        cudaMemcpy3D(&mcp); cudaCheckError();

        // copy parameters for location
```



```
        mcp.dstPtr = d_p->loc_dPitchPtr;
        mcp.srcPtr.ptr = loc;
        mcp.srcPtr.pitch = d_p->loc_ext.width;
        mcp.srcPtr.xsize = (size_t) (d_p->loc_ext.width/sizeof(int));
        mcp.srcPtr.ysize = d_p->loc_ext.height;
        mcp.extent = d_p->loc_ext;
        cudaMemcpy3D(&mcp); cudaCheckError();
}

/*
 * Copies location and distance array memory
 * from device memory to pinned memory
 * Arguments:
 *   Phi3D_d* [in/out] - pointer to d_Phi (Device - Source)
 *    double* [in]     - pointer to distance (Pinned - Destination)
 *          int* [in]     - pointer to location (Pinned - Destination)
 * Returns:
 */
static void cudaMemcpy3D_D2P(Phi3D_d *d_p, double *dst, int *loc) {
        cudaMemcpy3DParms mcp = { 0 };
        mcp.kind = cudaMemcpyDeviceToDevice;

        // copy parameters for distance
        mcp.srcPtr = d_p->dst_dPitchPtr;
        mcp.dstPtr.ptr = dst;
        mcp.dstPtr.pitch = d_p->dst_ext.width;
        mcp.dstPtr.xsize = (size_t) (d_p->dst_ext.width/sizeof(double));
        mcp.dstPtr.ysize = d_p->dst_ext.height;
        mcp.extent = d_p->dst_ext;
        cudaMemcpy3D(&mcp); cudaCheckError();

        // copy parameters for location
        mcp.srcPtr = d_p->loc_dPitchPtr;
        mcp.dstPtr.ptr = loc;
        mcp.dstPtr.pitch = d_p->loc_ext.width;
        mcp.dstPtr.xsize = (size_t) (d_p->loc_ext.width/sizeof(int));
        mcp.dstPtr.ysize = d_p->loc_ext.height;
        mcp.extent = d_p->loc_ext;
        cudaMemcpy3D(&mcp); cudaCheckError();
}
```

```c
/*
 * vti_parser.h
 *
 *  Created on: Jun 04, 2015
 *      Author: Anup Shrestha
 *
 *  Audit Trail:
 *     Date:
 *  Changes:
 */

#ifndef VTI_Parser_H_
#define VTI_Parser_H_

#include <stdio.h>
#include <string.h>

/*
 * Structure for storing the dimension
 * and spacing of the grid
 */
typedef struct {
        int    x, y, z;
} Grid3D;

// public method declarations
Grid3D make_grid3D(int x, int y, int z);
void vti_get_dimensions(FILE *vti, double *d);
void vti_get_data(FILE *vti, int *l, int b_l, double *d, double b_d, Grid3D g);

#endif /* VTI_Parser_H_ */
```

```c
/*
 * vti_parser.c
 * This file parses information from the VTI file
 * and stores them in the appropriate output
 * parameters. Also responsible for setting the
 * border values for the arrays.
 *
 *  Created on: Jun 04, 2015
 *      Author: Anup Shrestha
 *
 *  Audit Trail:
 *     Date:
 *  Changes:
 */
#include "vti_parser.h"

// private method declarations
static void move_file_pointer(FILE *file_ptr, int lineNumber, int r);
static void get_location(FILE *vti, int *l, int b_l, Grid3D g);
static void get_distance(FILE *vti, double *d, double b_d, Grid3D g);

/*
 * Creates a Grid3D struct.
 * Arguments:
 *   int x, y, z [in] - dimensions
 * Returns:
 *   Grid3D - struct representing 3D grid
 */
Grid3D make_grid3D(int x, int y, int z){
        Grid3D g;
        g.x = x; g.y = y; g.z = z;

        return g;
}

/*
 * Moves the pointer of a FILE to specified line.
 * Argument(s):
 *   FILE* [in/out] - file pointer that is to be moved
 *         int [in]     - line number to move the pointer to
 *         int [in]     - 0 - rewind; 1 - no rewind
 * Returns:
 */
static void move_file_pointer(FILE *file_ptr, int lineNumber, int r) {
        char tmpStr[512];
        if(r) rewind(file_ptr);
        while (lineNumber > 0){
                fgets (tmpStr, 511, file_ptr);
                lineNumber--;
        }
}

/*
 * Searches for x, y, z, dx, dy, dz information
 * in the VTI file and puts it into the array
 * taken as the second argument.
 * Argument(s):
 *        FILE*  [in]  - vti file to be parsed
 *       double* [out] - array to store parsed data
 * Returns:
 */
void vti_get_dimensions(FILE *vti, double *d) {
        char tmpStr[512];
        rewind(vti);
        while (1) {
                fgets (tmpStr, 511, vti);
                if ( strstr(tmpStr, "ImageData WholeExtent") ) {
                        sscanf(tmpStr, "  <ImageData WholeExtent=\"0 %lf 0 %lf 0 %lf\" Spacing
=\"%lf %lf %lf\">",
```



```c
                                        &d[0], &d[1], &d[2], &d[3], &d[4], &d[5]
);
                        break;
                }
        }
}

/*
 * Calls the appropriate methods to parse the
 * file for location and distance values and
 * stores them in an int and double array
 * respectively.
 * Arguments:
 *         FILE* [in]  - vti file to be parsed
 *      int* [out] - array to store location values
 *       int [in]  - border value for location
 *      double* [out] - array to store distance values
 *       double [in]  - border value for distance
 *   Grid3D* [in]  - dimensions of the grid
 * Returns:
 */
void vti_get_data(FILE *vti, int *l, int b_l, double *d, double b_d, Grid3D g) {

        // move the file pointer to
        // line 6 from beginning
        move_file_pointer(vti, 6, 1);

        get_location(vti, l, b_l, g);

        // move the file pointer 2 lines
        // forward from its last position
        move_file_pointer(vti, 2, 0);

        get_distance(vti, d, b_d, g);

}

/*
 * Parses the file for location values and
 * stores them in the int array. Also adds
 * the default border values for location
 * in the array.
 * Arguments:
 *         FILE* [in]  - vti file to be parsed
 *      int* [out] - array to store location values
 *       int [in]  - border value for location
 *   Grid3D* [in]  - dimensions of the grid
 * Returns:
 */
static void get_location(FILE *vti, int *l, int b_l, Grid3D g) {
        int i, j, k, *t = &l[0];
        for (i = 0; i < g.z; i++){
                for (j = 0; j < g.y; j++) {
                        for (k = 0; k < g.x; k++) {
                                // Border
                                if (k == 0 || k == g.x-1 || j == 0 || j == g.y-1
 || i == 0 || i == g.z-1 ) {
                                        *(t++) = b_l;
                                }
                                else{ // Interior
                                        fscanf(vti, "%d ", t++);
                                }
                        }
                }
        }
}

/*
 * Parses the file for distance values and
```

```
 * stores them in the double array. Also adds
 * the default border values for distance
 * in the array.
 * Arguments:
 *          FILE* [in]  - vti file to be parsed
 *        double* [out] - array to store distance values
 *         double [in]  - border value for distance
 *   Grid3D* [in]  - dimensions of the grid
 * Returns:
 */
static void get_distance(FILE *vti, double *d, double b_d, Grid3D g) {
        int i, j, k;
        double *t = &d[0];
        for (i = 0; i < g.z; i++){
                for (j = 0; j < g.y; j++) {
                        for (k = 0; k < g.x; k++) {
                                // Border distance
                                if (k == 0 || k == g.x-1 || j == 0 || j == g.y-1
 || i == 0 || i == g.z-1 ) {
                                        *(t++) = b_d;
                                }
                                else{ // Interior distance
                                        fscanf(vti, "%lf ", t++);
                                }
                        }
                }
        }
}
```

```c
/*
 * file_writer.h
 *
 *  Created on: Jun 04, 2015
 *      Author: Anup Shrestha
 *
 *  Audit Trail:
 *     Date:
 *  Changes:
 */

#ifndef File_Writer_H_
#define File_Writer_H_

#include <stdio.h>
#include <stdlib.h>
#include <netcdf.h>

/*
 * Enumerator for types of output files
 *  1) VTI - VTK Image Data
 *  2) DAT - Data file
 *  3) NCF - NetCDF file
 */
typedef enum {
        VTI = 0x01,
        DAT = 0x02,
        NCF = 0x04
} FileOut;

/*
 * Structure for storing the dimension
 * and spacing of the grid
 */
typedef struct {
        int    x, y, z;
        double dx, dy, dz;
} FileGrid;

/*
 * Structure for information required
 * to generate an output file
 */
typedef struct {
        char       *name; // name of the output file
        double     *data; // data to be put into file
        FileOut    out;   // type of file to output
        FileGrid   grid;  // grid dimensions/spacing
} FileType;

// NetCDF Macros
// Handle errors by printing an error message and exiting with a
// non-zero status.
#define ERRCODE 2
#define ERR(e) {printf("Error: %s\n", nc_strerror(e)); exit(ERRCODE);}

// public method declarations
FileGrid make_fileGrid(int x, int y, int z, double dx, double dy, double dz);
FileType make_fileType(char *n, double *d, FileOut o, FileGrid g);
void file_generate(FileType *f);

#endif /* File_Writer_H_ */
```

```c
/*
 * file_writer.c
 * Generates file of different types based
 * on the information provided through the
 * FileType structure. The types of file that
 * can be generated are:
 *  1) VTI - VTK Image Data
 *  2) DAT - Data file
 *  3) NCF - NetCDF file
 *
 *  Created on: Jun 04, 2015
 *      Author: Anup Shrestha
 *
 *  Audit Trail:
 *     Date:
 *  Changes:
 */
#include "file_writer.h"

// private method declarations
static void file_gen_dat(double *data);
static void file_gen_vti(double *data);
static void file_gen_ncf(double *data);

// global variables
static char fileName[255]; // stores the full filename
static int x, y, z;        // grid dimensions
static double dx, dy, dz;  // node spacing

/*
 * Creates a FileGrid struct.
 * Arguments:
 *         int x, y, z [in] - 3D dimension of grid
 *   double dx, dy, dz [in] - spacing of the nodes on
 *                            each dimension of grid
 * Returns:
 *   FileGrid - struct with dimension and spacing
 *              information
 */
FileGrid make_fileGrid(int x, int y, int z, double dx, double dy, double dz) {
        FileGrid g;
        g.x = x; g.dx = dx;
        g.y = y; g.dy = dy;
        g.z = z; g.dz = dz;

        return g;
}

/*
 * Creates a FileType struct.
 * Arguments:
 *       char* [in] - name of the file
 *    double* [in] - data to be written
 *    FileOut [in] - type of file to generate
 *   FileGrid [in] - dimensions/spacing of the grid
 * Returns:
 *   FileType - struct containing all the information
 *              to generate a file
 */
FileType make_fileType( char *n, double *d, FileOut o, FileGrid g) {
        FileType f;
        f.name = n;
        f.data = d;
        f.out  = o;
        f.grid = g;

        return f;
}
```



```c
/*
 * Determines what types of file to output
 * generates filename for each type and
 * calls the appropriate methods to generate
 * the file.
 * Arguments:
 *   FileType* [in] - pointer to FileType
 * Returns:
 */
void file_generate(FileType *f) {

        // initialize global variables
        x = f->grid.x; dx = f->grid.dx;
        y = f->grid.y; dy = f->grid.dy;
        z = f->grid.z; dz = f->grid.dz;

        // check if DAT bit is set
        if(f->out & DAT) {
                sprintf(fileName, "%s.dat", f->name);
                file_gen_dat(f->data);
                printf("DAT file created: %s\n", fileName);
        }

        // check if VTI bit is set
        if(f->out & VTI) {
                sprintf(fileName, "%s.vti", f->name);
                file_gen_vti(f->data);
                printf("VTI file created: %s\n", fileName);
        }

        // check if NCF bit is set
        if(f->out & NCF) {
                sprintf(fileName, "%s.nc", f->name);
                file_gen_ncf(f->data);
                printf("NCF file created: %s\n", fileName);
        }
}

/*
 * Generates .dat output file
 * Arguments:
 *   double* - data to be written
 * Returns:
 */
static void file_gen_dat(double *data) {
        file_gen_vti(data);
}

/*
 * Generates .vti output file
 * Arguments:
 *   double* - data to be written
 * Returns:
 */
static void file_gen_vti(double *data) {

        FILE *fp = fopen(fileName, "w");

        // Header Information + Data for VTI file
        fprintf(fp, "<?xml version=\"1.0\"?>\n");
        fprintf(fp, "<VTKFile type=\"ImageData\" version=\"0.1\" byte_order=\"LittleEndian\">\n");
        fprintf(fp, "\t<ImageData WholeExtent=\"0 %d 0 %d 0 %d\" Origin=\"0 0 0\" Spacing=\" %f %f %
f \">\n", x-1, y-1, z-1, dx, dy, dz);
        fprintf(fp, "\t\t<Piece Extent=\"0 %d 0 %d 0 %d\">\n", x-1, y-1, z-1);
        fprintf(fp, "\t\t\t<PointData Scalars=\"Distance Field\">\n");
        fprintf(fp, "\t\t\t\t<DataArray type=\"Float32\" Name=\"Distance Field\" format=\"ascii\">\n");

        int i,j,k;
        double *t = &data[0];
```

```
        for(i = 0; i < z; i++){
                for(j = 0; j < y; j++){
                        for(k = 0; k < x; k++) {
                          fprintf(fp, "%f ", *(t++));
                        }
                        fprintf(fp, "\n");
                }
                fprintf(fp, "\n");
        }

        // Closing matching entities
        fprintf(fp, "\t\t\t\t</DataArray>\n");
        fprintf(fp, "\t\t\t</PointData>\n");
        fprintf(fp, "\t\t\t<CellData>\n");
        fprintf(fp, "\t\t\t</CellData>\n");
        fprintf(fp, "\t\t</Piece>\n");
        fprintf(fp, "\t</ImageData>\n");
        fprintf(fp, "</VTKFile>\n");

        fclose(fp);
}

/*
 * Generates .nc output file
 * Arguments:
 *   double* - data to be written
 * Returns:
 */
static void file_gen_ncf(double *data){

        int ndims = 3;

  // IDs for the netCDF file, dimensions, and variables
  int ncid, x_dimid, y_dimid, z_dimid, retval;
  int dist_varid, x_varid, y_varid, z_varid;
  int dimids[ndims];

  // Data for x, y, z variables
  float lats[y], lons[x], deps[z];
  int i,j,k;
  for(k = 0; k < x; k++)
    lons[k] = dx * k;
  for(j = 0; j < y; j++)
    lats[j] = dy * j;
  for(i = 0; i < z; i++)
    deps[i] = dz * i;

  // Create the file. The NC_NETCDF4 flag tells netCDF to
  // create a netCDF-4/HDF5 file.
  if((retval = nc_create(fileName, NC_NETCDF4  | NC_CLOBBER, &ncid)))
    ERR(retval);

  // Define the dimensions in the root group. Dimensions are visible
  // in all subgroups
  if ((retval = nc_def_dim(ncid, "x", x, &x_dimid)))
    ERR(retval);

  if ((retval = nc_def_dim(ncid, "y", y, &y_dimid)))
    ERR(retval);

  if ((retval = nc_def_dim(ncid, "z", z, &z_dimid)))
    ERR(retval);


  // The dimids passes the IDs of
  // the dimensions of the variable
  dimids[0] = z_dimid;
  dimids[1] = y_dimid;
  dimids[2] = x_dimid;
```



```
  // Defining the coordinate variables
  if ((retval = nc_def_var(ncid, "x", NC_FLOAT, 1, &x_dimid, &x_varid)))
    ERR(retval);

  if ((retval = nc_def_var(ncid, "y", NC_FLOAT, 1, &y_dimid, &y_varid)))
    ERR(retval);

  if ((retval = nc_def_var(ncid, "z", NC_FLOAT, 1, &z_dimid, &z_varid)))
    ERR(retval);

  // Assign delta attribute to coordinate variables
  if ((retval = nc_put_att_double(ncid, x_varid, "delta", NC_DOUBLE, 1, &dx)))
    ERR(retval);

  if ((retval = nc_put_att_double(ncid, y_varid, "delta", NC_DOUBLE, 1, &dy)))
    ERR(retval);

  if ((retval = nc_put_att_double(ncid, z_varid, "delta", NC_DOUBLE, 1, &dz)))
    ERR(retval);

  // Define a float variable in root, using dimensions
  // in the root group.
  if ((retval = nc_def_var(ncid, "Distance Field", NC_FLOAT, ndims, dimids, &dist_va
rid)))
    ERR(retval);

  // End define mode
  if ((retval = nc_enddef(ncid)))
    ERR(retval);

  // Put the data values for the dimensions and distance
  if ((retval = nc_put_var_float(ncid, x_varid, &lons[0])))
    ERR(retval);

  if ((retval = nc_put_var_float(ncid, y_varid, &lats[0])))
    ERR(retval);

  if ((retval = nc_put_var_float(ncid, z_varid, &deps[0])))
    ERR(retval);

  if ((retval = nc_put_var_double(ncid, dist_varid, data)))
    ERR(retval);

  // Close the file.
  if ((retval = nc_close(ncid)))
    ERR(retval);
}
```