

CUDA Report Zach

by Zach Port-Lincoln

FILE	3805_ZACH_PORT-LINCOLN_CUDA_REPORT_ZACH_107698_2092147783.DOCX (46.14K)		
TIME SUBMITTED	08-MAY-2019 05:43PM (UTC+0100)	WORD COUNT	2910
SUBMISSION ID	106679845	CHARACTER COUNT	14637

Creating shapes from procedurally generating Marching Squares (MS)

Abstract

In this paper the developer is going to look at author's steps in creating meshes from the MS algorithm and to see if there are any variants or things to add in, in order to guide their work to a specific final render. The developer will be giving a brief over view of what the algorithm is and how it works. The developer will also be explaining how their code is working and has been implemented. The developer will then be looking at the results, evaluating them and finally seeing what can be implemented in the future.

Contents

Abstract.....	1
Introduction	3
Literature Review.....	3
My Algorithm	5
Parallelization and implementation.....	5
Results.....	6
Evaluation	6
Conclusion and future work.....	7
References	8
Appendix	9

Introduction

In this paper the developer is going to be talking about the Marching Squares (MS) algorithm. The MS algorithm is where it takes a 2D grid of scalar data, splits this data up into grid cells of equal size. The MS algorithm then visits each cell sequentially and turns that cell into a contour image, whether each corner of the grid or (node) is either inside, outside, or on the border of the shape. When it finds a node that's on the border it will convert that section of the grid into one of the predefined images from a look up table until all grid segments have been evaluated accordingly.

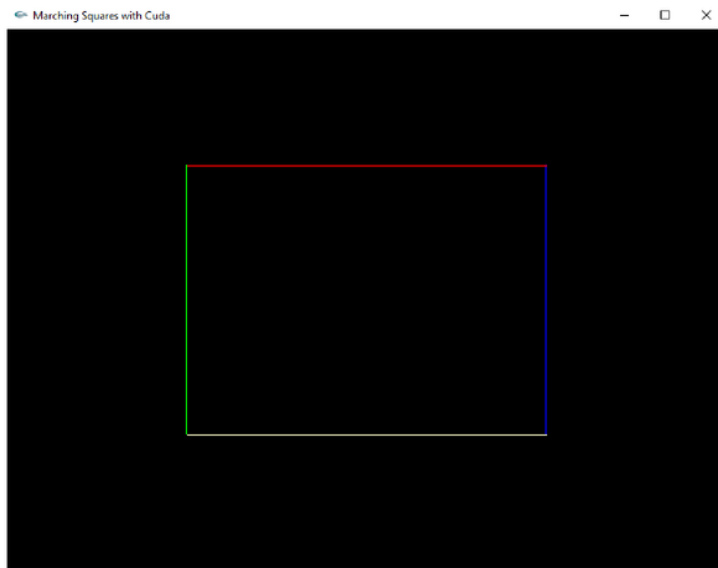
Literature Review

When using Marching Squares (MS), you can look at flat 2D shapes and determine what to find and/or draw. You can also use MS in conjunction with itself, that is called Cubical or Dual Marching Squares, to draw 3D objects, instead of doing another algorithm called Marching cubes (MC). The advantage of doing Cubical Marching Squares is that you will have no "gaps" in the final render of the object where in MC, some shapes used next to each other can produce said gaps in the final render. One paper that has done this dual MS approach was written by Chien-Chang Ho, Cheng-Han Tu and Ming Ouhyoung in 2005, and their paper is called "Detail Sculpting using Cubical Marching Squares" [1] In this paper the authors start the MS algorithm the typical way but when it comes to the sculpting section they have to change course slightly. They split the space into sub divisions of 2D space and render each square that needs rendering in each of the 2D sub divisions to get their detail and joined up 2D shapes in the 3D space. Another paper that has worked with cubical marching squares is one from the authors Chien-Chang Ho, Bing-Yu Chen, Ming Ouhyoung, and Jyh-Horng Chen in 2006, the paper is called "Extended Cubical Marching Squares for Surface Extraction from Various Kinds of Volumetric Structure" [2] in this paper the authors have worked out another way to render cubes but only using marching squares. They unfold the cubes into squares, run the marching squares algorithm to get the desired shape then folds them all back up together. This also achieves the same result as the previous paper, as in the final rendered image will have no "gaps" what this paper has also achieved is that they can use far thinner objects and still manage to get the same results, as the cubical marching cubes algorithm needs set scalar values, sometimes these values can be too big for parts of the object so some accuracy can be lost. Another similar algorithm is using the MC algorithm and then the MS algorithm. One paper that uses this is one by HAN Hongling and CUI Zhiheng called "Study on Three-Dimensional Scene Modelling and Interactive Technology Based on Marching Square Algorithm"[3] in this paper they split the object into a grid as normal, then they get contour lines on each side using the MS algorithm. Then using the contour lines, they generate a wire like frame to fill in for the final

shape. Finally, if they need to do any extra things like making it look more real then this is the stage, they will do that. In this paper their work has been displayed for a virtual reality system. Another paper that uses this same format for using Marching Cubes and then Marching Squares is one by Carsten Maple, this paper is called "Geometric Design and Space Planning Using the Marching Squares and Marching Cube Algorithms"[4] What Carsten Maple does in this paper is she will use Marching Cubes to approximate the area and volume of 3D space. Maple also does the same thing but with Marching Squares. They also visualise what the MC and MS have done. Finally using either the MC or MS algorithm they can work out the distance between 2 points which they find useful for the space planning, they show an example and the steps to work this out. Instead of using 2 algorithms, Sui Gong and Timothy S. Newman, have introduced a new method called Dual Marching Squares (DMS), they have wrote a paper on it called "Dual Marching Squares: Description and Analysis" [5] In this paper they talk about how they can find the midpoint of the iso contour, and then use the midpoint of that intersection and connect it to other mid points for a, in some cases, a smoother image. In the DMS algorithm it starts off the same as MS where it finds what square is overlapping and does the required steps to find what case that cell is, then using linear interpolation it would find the exact point the line would cross. Then the DMS intersection point is found, this is found at the midpoint of the two MS edge intersection points. Finally, it connects up all the mid points to form the shape. From their calculations, DMS seem to generate a smoother image than the MS in comparison. Lastly the authors, Sui Gong and Timothy S. Newman have produced paper called "A Corner Feature Sensitive Marching Squares"[6] which is very similar to the "Dual Marching Squares: Description and Analysis" [5] but this one was written first. In this paper they only concentrate on the segments of the grid that have sharp corners in them, where the original MS algorithm would turn this corner into a smoother one, here they have changed that to go to the midpoint of the square and then draw the intersection lines in order to make the corner as sharp as possible. They also say, as this only concentrates on a small but crucial part of the algorithm, that this is only an extension to the algorithm.

My Algorithm

In this project the developer is going to be using the marching squares algorithm in order to draw a square. What this algorithm will do is split the scene up into a grid of cells. Then it will scan each cell and see what corners are inside or on the border of the shape. Depending on what corners are inside the shape or outside the shape it will draw a line accordingly. In this project the developer is going to be making a larger square out of the marching squares algorithm. This will first be ran and timed on the CPU and then ran and timed on the graphics card.



This is the final result of the CPU code, every cell, at a cell size of 1 is getting checked, and in other words the cell size is the amount of pixels. Size one is 1 plus the starting pixel, 5 would be 5 plus the starting pixel and so on.

Parallelization and implementation

What the developer is going to parallelize is going to be each cell of the grid. The developer is going to pass the X and Y coordinates of the cell in question over from the CPU to the graphics card and from there the graphics card will then run the distance checks to see if either one of the 4 corners are inside the shape. Then once this has been completed it will then run the function to draw the line with its own coordinates if they are within the specified boundaries of the shape. This should be all the developer needs to parallelize as this is the only part of the code that is ran multiple times on the CPU. The developer is also going to test to see if it is faster to get the graphics card to run the line drawing function or if it faster to pass the co-ordinates and a true or false bool back over to the CPU in order for the CPU to run the line drawing algorithm.

Results

As the CUDA side of the programming has not been fully implemented yet there are no direct results to compare. What the developer thinks will happen is as the CPU can only deal with one cell of the grid at a time, that the CPU will in fact be slower than the graphics card. As the graphics card can deal with all these cells of the grid all at once the developer is expecting the graphics card to run the code at a much faster rate. As the developer is going to be running an OpenGL function call in order to draw a line on the screen, the developer doesn't know if this will cause problems for the time to run to increase or not. To the developers understanding OpenGL is supported with CUDA so using the graphics card to call a function to draw a line shouldn't be a problem but this will be tested in the code.

Evaluation

To evaluate the results the developer will have to time the code on both the CPU and the graphics card. As the CUDA section of the code is not fully implemented the developer is unable to accurately compare results. According to a profiling report the code that has been completed to run on the CPU has been timed, and according to this time it took 1.719 seconds to run but 98.69% of this time was idling, so with this high percent taken off the full time, the more accurate time that the CPU code took to run in order to fully render this square is 0.0178776 seconds. Although the time for this is very low, there is still an amount of code that has to wait in order for code within a loop to finish. Integrating CUDA into this code will therefore take this loop out, and in theory, make the program even faster. Although this is possible there is still a set up cost to establish a working CUDA so this could hinder the initial time it takes to complete.

Conclusion and future work

In this paper the developer has carried out a literature review through the Marching Squares algorithms and slight deviations and additions. The developer has also explained the different steps authors followed in undertaking reviews on this topic. The developer has then explained the algorithm the developer is going to be using and how. Also the developer explained how the CPU code is going to be parallelized and implemented and finally the possible results that could come of the graphics card nullifying the loop.

For the future the developer would like to implement the CPU code fully into CUDA. The developer would also like to take the shape out altogether and set each cell to randomly draw a line. This gets rid of the containment check but the developer could see if it is causing issues with OpenGL.

References

- [1] C.-C. Ho, C.-H. Tu, and M. Ouhyoung, "Detail Sculpting using Cubical Marching Squares."
- [2] C.-C. Ho, B.-Y. Chen, M. Ouhyoung, and J.-H. Chen, "Extended cubical marching squares for surface extraction from various kinds of volumetric structure," *ACM SIGGRAPH 2006 Res. posters - SIGGRAPH '06*, vol. d, p. 165, 2006.
- [3] H. Han and Z. Cui, "Study on three-dimensional scene modeling and interactive technology based on marching square algorithm," in *2010 The 2nd Conference on Environmental Science and Information Application Technology*, 2010, pp. 416–419.
- [4] C. Maple, "Geometric design and space planning using the marching squares and marching cube algorithms," in *2003 International Conference on Geometric Modeling and Graphics, 2003. Proceedings*, 2003, pp. 90–95.
- [5] S. Gong and T. S. Newman, "Dual Marching Squares: Description and analysis," in *2016 IEEE Southwest Symposium on Image Analysis and Interpretation (SSIAI)*, 2016, pp. 53–56.
- [6] S. Gong and T. S. Newman, "A corner feature sensitive marching squares," in *2013 Proceedings of IEEE Southeastcon*, 2013, pp. 1–6.

Appendix

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "glut.h"

//Size of window
const int WIDTH = 800;
const int HIGHT = 600;

int topOfSquare, bottomOfSquare, leftOfSquare, rightOfSquare;
int startX, startY, endX, endY;

bool topLeft = false, topRight = false, bottomRight = false, bottomLeft = false;
bool leftLeft = false, topTop = false, rightRight = false, bottomBottom = false;
bool allPointsInside = false;

struct Grid
{
    int x = 0, y = 0;

    int gridSize = 1, offsetX = 0, offsetY = 0;
}cell;

struct Vector
{
    int x = 0, y = 0;
};

Vector vector1;

void init()
{
    topOfSquare = HIGHT - (HIGHT / 4);
    bottomOfSquare = HIGHT / 4;
    leftOfSquare = WIDTH / 4;
    rightOfSquare = WIDTH - (WIDTH / 4);

    glClear(GL_COLOR_BUFFER_BIT);

    glColor3f(1.0, 0.0, 0.0);
}

void drawLine(int startX, int startY, int endX, int endY) //this function will draw
the line fromthe start x & y to the end x & y
{
    glBegin(GL_LINES);
```

```

        glVertex2f(startX + cell.offsetX, startY + cell.offsetY);    // start of line
        glVertex2f(endX + cell.offsetX, endY + cell.offsetY);        // end of line

    glEnd();

}

void draw()
{
    //int count = 0;
    //printf("ToS %d, BoS %d, LoS %d, RoS %d\n", topOfSquare, bottomOfSquare,
    leftOfSquare, rightOfSquare);

    for (int i = 0; i < HIGHT;)
    {
        for (int j = 0; j < WIDTH;)
        {
            //*****
            //This section gets the location of the top left vertex of the
cell
            //*****
            vector1.x = cell.x + cell.offsetX;
            vector1.y = cell.y + cell.offsetY + cell.gridSize;

            if ((vector1.y == topOfSquare) && (vector1.x > leftOfSquare) &&
(vector1.x < rightOfSquare))
            {
                topTop = true;
            }

            if ((vector1.x == leftOfSquare) && (vector1.y > bottomOfSquare)
&& (vector1.y < topOfSquare))
            {
                leftLeft = true;
            }

            //*****
            //This section gets the location of the top right vertex of the
cell
            //*****
            vector1.x = cell.x + cell.offsetX + cell.gridSize;
            vector1.y = cell.y + cell.offsetY + cell.gridSize;

            if ((vector1.y == topOfSquare) && (vector1.x > leftOfSquare) &&
(vector1.x < rightOfSquare))
            {
                topTop = true;
            }
        }
    }
}

```

```

    }

    if ((vector1.x == rightOfSquare) && (vector1.y > bottomOfSquare)
    && (vector1.y < topOfSquare))
    {

        rightRight = true;

    }

    //*****
    //This section gets the location of the bottom right vertex of
the cell

    //*****
    vector1.x = cell.x + cell.offsetX + cell.gridSize;
    vector1.y = cell.y + cell.offsetY;

    if ((vector1.x == rightOfSquare) && (vector1.y > bottomOfSquare)
    && (vector1.y < topOfSquare))
    {

        rightRight = true;

    }

    if ((vector1.y == bottomOfSquare) && (vector1.x > leftOfSquare)
    && (vector1.x < rightOfSquare))
    {

        bottomBottom = true;

    }

    //*****
    //This section gets the location of the bottom left vertex of the
cell

    //*****
    vector1.x = cell.x + cell.offsetX;
    vector1.y = cell.y + cell.offsetY;

    if ((vector1.y == bottomOfSquare) && (vector1.x > leftOfSquare)
    && (vector1.x < rightOfSquare))
    {

        bottomBottom = true;

    }

    if ((vector1.x == leftOfSquare) && (vector1.y > bottomOfSquare)
    && (vector1.y < topOfSquare))
    {

        leftLeft = true;

    }

    //This if statement will see if all points are inside the circle
and set the bool true if they are

```

```

        if ((topTop == true) && (rightRight == true) && (bottomBottom ==
true) && (leftLeft == true))
        {

            allPointsInside = true;

        }

//This if will draw the top part of the square
if ((topTop == true) && (allPointsInside == false))
{
    //printf("Drawing top part - \n");

    startX = cell.x;
    startY = cell.y + (cell.gridSize / 2);

    endX = cell.x + cell.gridSize;
    endY = cell.y + (cell.gridSize / 2);

    glColor3f(1.0, 0.0, 0.0);
    drawLine(startX, startY, endX, endY);

}
//This if will draw the left part of the square
if ((leftLeft == true) && (allPointsInside == false))
{
    //printf("Drawing left part - \n");

    startX = cell.x + (cell.gridSize / 2);
    startY = cell.y;

    endX = cell.x + (cell.gridSize / 2);
    endY = cell.y + cell.gridSize;
    glColor3f(0.0, 1.0, 0.0);
    drawLine(startX, startY, endX, endY);

}

//This if will draw the right part of the square
if ((rightRight == true) && (allPointsInside == false))
{
    //printf("Drawing right part - \n");

    startX = cell.x + (cell.gridSize / 2);
    startY = cell.y;

    endX = cell.x + (cell.gridSize / 2);
    endY = cell.y + cell.gridSize;
    glColor3f(0.0, 0.0, 1.0);
    drawLine(startX, startY, endX, endY);

}

//This if will draw the bottom part of the square
if ((bottomBottom == true) && (allPointsInside == false))
{
    //printf("Drawing bottom part - \n");

    startX = cell.x;
    startY = cell.y + (cell.gridSize / 2);

    endX = cell.x + cell.gridSize;

```

```

        endY = cell.y + (cell.gridSize / 2);
        glColor3f(1.0, 1.0, 0.8);
        drawLine(startX, startY, endX, endY);

    }

    cell.offsetX += cell.gridSize;
    j += cell.gridSize;
    //printf("Cell j = %d\n", j);
    //Sets the containment checks back to false
    bottomBottom = false;
    topTop = false;
    leftLeft = false;
    rightRight = false;
    allPointsInside = false;

}

cell.offsetX = 0;
cell.offsetY += cell.gridSize;

i += cell.gridSize;

}

cell.offsetY = 0;

glFlush();// forces a render to the screen
glutSwapBuffers();

}

void main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitWindowPosition(900, 200);
    glutInitWindowSize(WIDTH, HIGHT);
    glutInitDisplayMode(GLUT_RGB);
    glutCreateWindow("Marching Squares with Cuda");
    init();
    glutDisplayFunc(draw);
    //glutKeyboardFunc(keyboardread);
    //glutIdleFunc(wallCheck);
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glOrtho(0.0, WIDTH, 0.0, HIGHT, 0.0, 100.0); //the minus parts were 0.0

    glutMainLoop();
}

```

CUDA Report Zach

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

38/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

GRADING FORM: SMU GENERIC_GJ

ZACH PORT-LINCOLN

37.75

RESEARCH-INFORMED LITERATURE



Refs included. Some incompletely formatted. Only 6?

4.25

KNOWLEDGE AND UNDERSTANDING



A more complete/clearer description of the method would help here (an example with diagrams perhaps). Regarding aspect(s) of the algorithm that could be parallelised, show an example that includes (x.y) co-ords and diagrams of the squares indicating inside and/or outside tests.

7

ANALYSIS



Aspects of the algorithm that can be parallelised have been identified and discussed - please indicate explicitly this part. Relevant topics identified and discussed in review.

4

PRACTICAL APPLICATION AND DEP.



Only CPU aspects tested here. Some timing data has been supplied but is limited. Explain how you timed and gathered the results. A lack of clarity here.

20

SKILLS FOR PROFESSIONAL PRAC.



Report structured, refs included. Could have made better use of graphs and diagrams to support discussions and evaluation.

2.5