Rendering with Marching Cubes, looking at Hybrid Solutions

Patrik Andersson

Sakarias Johansson

Abstract

Marching Cubes is a rendering technique that has many advantages for a lot of areas. It is a technique for representing scalar fields as a three-dimensional mesh. It is used for geographical applications as well as scientific ones, mainly in the medical industry to visually render medical data of the human body. But it's also an interesting technique to explore for the usage in computer games or other real-time applications since it can create some really interesting rendering.

The main focus in this paper is to present a novel hybrid solution using marching cubes and heightmaps to render terrain; moreover, to find if it’s suitable for real-time applications. The paper will follow a theoretical approach as well as an implementational one on the hybrid solution.

The results across several tests for different scenarios show that the hybrid solution works well for today's real-time applications using a modern graphics card and CPU (Central Processing Unit).

Table of Contents

Introduction

The Marching Cubes algorithm [1] is not entirely new. It was first developed in the eighties [5] and was then quickly patent. It was not until 2005 [5] the patent was released and the algorithm was open for implementations. Before this other similar techniques, such as marching tetrahedrons [2], had been developed to avoid patent issues.
Marching Cubes is a volumetric algorithm that builds up a polygonal mesh from a scalar field taking in eight neighboring points, forming a cube. It's used to represent three dimensional data which is also known as volumetric data.
Marching Cubes can be seen as a rectangle or cube built up with other smaller blocks all of the same size. These blocks are also known as voxels, volumetric pixels or volumetric picture elements. Each voxel holds a certain value describing how filled the voxel is.
Together all these voxels represents a scalar field. The Marching Cubes algorithm can then use this volumetric data (the scalar field) to generate a polygonal mesh iterating through each voxel and finding fitting pieces for the mesh. The result is a three dimensional mesh.

The algorithm is most commonly used to visualize medical data, such as an M.R.I scan. The M.R.I scans can generate 3D data for the Marching Cubes algorithm that can be used to represent 3D models of human tissue. Some really interesting results can be achieved using the algorithm when considering real-time applications such as video games. Compared to a regular heightmap where only height is the one thing that can be changed, Marching Cubes can render advanced terrain with tunnels and overhangs.

A heightmap is a 2D greyscale picture where every pixel represents a height in the terrain. The position of the pixel (x and y) represents two of the axis in the world and the value that relies in that pixel represents the z-axis. This means that a 3D terrain can be created from the 2D texture.
The only feature it supports is variation in height. It's not possible to create caves in heightmaps simply because it would require an overlap of z values in the texture which it does not support.

Moreover, Marching Cubes can be used for interesting particle rendering or fluid simulation. The algorithm has not been used that much in this industry since there are several draw backs using the algorithm, mainly the massive memory usage.

Goal & Purpose

The paper focuses on a hybrid solution using marching cubes and heightmaps to render terrain. To see how well the solution runs on modern hardware for real-time applications. The purpose of using this hybrid technique is not to use Marching Cubes entirely over the whole terrain but only when necessary. To see if a middle-ground can be found that yields the interesting results of marching cubes alongside heightmaps for performance gain regarding both memory usage and mesh calculation time.

The paper will mainly focus on the following:

1. Is it possible to create a hybrid between heightmaps and the marching cubes algorithm?
2. How will the hybrid solution work on real-time applications (execution time, memory usage)?
3. Will it be possible to get any performance increase in execution time by moving parts of the marching cubes algorithm to the GPU (Graphics Processing Unit)?

Hypothesis

Since the majority of the implementation relies on creating a hybrid solution between heightmaps and marching cubes to actually be able to sample results, it was of upmost importance to work the idea of this technique as a priority. The paper itself will thoroughly go through the first question, the creation of the hybrid solution.

As for how the hybrid would work on real-time applications the outcome is already, in a sense, somewhat clear. Using a heightmap itself, without marching cubes, will perform the best result. But without the interesting features such as overhangs and tunnels or other advanced terrain features. On the other hand, recreating the entire terrain using marching cubes is not feasible for the real-time applications. Thus, using heightmap together with marching cubes will naturally give a performance decrease but a necessary one to give the results we are looking for. Our hypothesis for the performance of the novel hybrid solution is as follows:

The novel hybrid solution integrates benefits from the marching cubes algorithm for its unique terrain generation and is able to perform satisfactory on a modern hardware.

As for moving parts of the algorithm to the GPU we had a strong belief that we would gain significant performance increase in execution time as compared to the CPU implementation. However, our results tell otherwise (see Results page 13). This will be further discussed and explained later into the paper in Discussion (page 15).

Delineation

This paper will strongly focus on algorithm-construction. There are already many papers on what Marching Cubes is and how it works. So instead the focus is more on the hybrid solution as an implementation. Both practical and theoretical aspects will be brought up. The implementation is done using DirectX 11 API and the algorithm will be tested on both the CPU and the GPU by using CUDA.

Target group

The reader should have knowledge in a graphical API such as DirectX or OpenGL. The paper won't explain basic 3D elements, but instead focus on the implementation of Marching Cubes as a hybrid solution with heightmaps and later focus on validation of performance.

Related Work

For the interested reader related work [6, 7, 8, 9] can be found for similar techniques or ideas in the References (page 20). The ideas are not quite the same but similar. Stack-based Terrains[7] and grounded heightmap trees [9] are two other ways to add support for overhangs and caves on heightmaps. Stack-based Terrains enabled complex geometry by adding materials on top of each other at the heightmap so each pixel of the heightmap could be seen as a stack of materials. The grounded heightmap tree enabled support for carving and such by linking smaller heightmaps to the larger root heightmap. Whenever a carving event occurred two heightmaps were created and linked to the root node (root heightmap) where one defines the cave entry and the other defines the cave end. Game Engine Gems 1 [6] describes the Marching Cubes algorithm in more detail, stating important facts and taking you through the implementation steps of the marching cubes algorithm. There have also been attempts to accelerate the marching cubes algorithm by moving parts to the GPU. Gunnar Johansson and Hamish Carr[8] managed to accelerate the marching cubes algorithm by testing various methods.

There is not much to find on the subject of hybrid techniques between heightmaps and the marching cubes algorithm considering how new marching cubes is since the patent expired. And today an accepted standard for terrain generation is simply by using heightmaps.

Design & Construction

The idea of our hybrid solution is to create a hybrid implementation between the Marching Cubes algorithm and heightmaps. Using a heightmap alone is very limited to what can be achieved, but instead with the hybrid solution we can extend the heightmap's features using marching cubes merged into the terrain itself. With this, interesting features such as caves and overhangs can be accomplished.

When modifying the heightmap, marching cubes fields are added to the world as a separate mesh that's merged, but not necessarily, into the heightmap. The part that's affected on the heightmap is removed and instead replaced by marching cubes fields. The fields can be either carved into or extended upon to create advanced terrain features that cannot be achieved by a heightmap alone.

With the merging technique performance of a heightmap is attained at best but extended upon when necessary with the fields of marching cubes as limited as possible. Naturally the worst result would be when the whole heighmap itself is replaced by marching cubes. This is solved by using chunks instead of filling the whole world with marching cubes. The world is split up into an oct-tree like grid where each chunk can hold several fields of marching cubes, but there is a limit to its maximum size. Once fields intersect with each other they will merge into a larger field to limit the draw calls. Data is extracted from the intersecting fields to the larger field and when that's done the intersecting fields will be removed. The merge size is dependent on the size of each chunk node. If the merged size of a marching cubes field exceeds the chunk node boundaries then a new marching cubes field will be created at the neighboring chunk node the field intersected at. Keeping the marching cubes fields to a limited size works well when modifying the mesh because having it too large will yield slow performance when the new mesh is generated. Even a small change in a marching cubes field will cause it to generate the whole mesh all over again. Performance is gained by making smaller chunks so that less data has to be recalculated. Keeping the chunk nodes relatively small sized, the marching cubes fields inside the node will have a mesh that is not larger than the node itself. This is simply so that the generation of the mesh does not get too complicated, as it can by having no fixed size on the marching cubes fields.

Scenarios

When modifying the terrain a box marker is used that works as an area of effect where the heightmap is to be modified as illustrated in Figure 1 below.

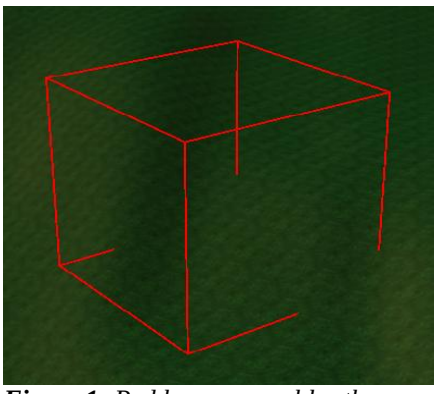


Figure 1: *Red box reassembles the area of effect.*

There are five different scenarios that can occur when modifying the world with the area of effect. First scenario is when modifying the world where no marching cubes field is present, the second is when the area of effect is fully inside a field modifying it, the third is when the area of effect is partially inside a field and the field is then extended to fit the total size of the current field and the area of effect, and fourth scenario is when the area of effect intersects two or more marching cubes fields causing them to merge into one larger field and at last the fifth scenario occurs when the area of effect intersects two or more chunks.

Scenario 1:

Here no previous field is present so a new marching cubes field is created to fit the area of effect and the affected terrain is removed and instead replaced by this new field as illustrated in Figure 2. Surrounding terrain of the affected area is glued to the neighboring points in the marching cubes field to avoid gaps and give a fluid look.

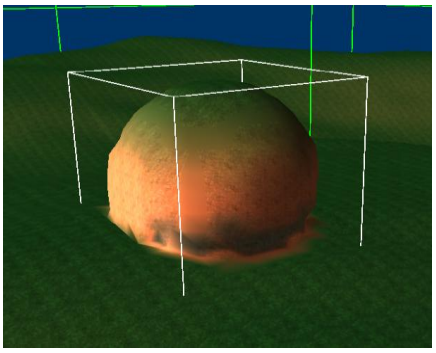


Figure 2: *Field replacing the terrain.*

Scenario 2:

In this case no new marching cubes fields is needed since the area of effect is fully inside the field itself. Instead the area of effect modifies the field and a new mesh is generated as seen in Figure 3. This is the best case scenario performance wise since there is no new data to be created; only existing data is modified.

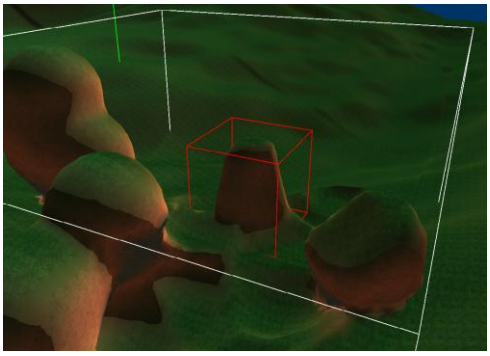


Figure 3: *Existing field is changed.*

Scenario 3:

Here the area of effect is partially inside an already existing marching cubes field. The field is then extended to cover the whole total area of effect. This gives some performance decrease because new data has to be created. Figure 4 shows the field before it's extended and Figure 5 shows the complete extension of the field.

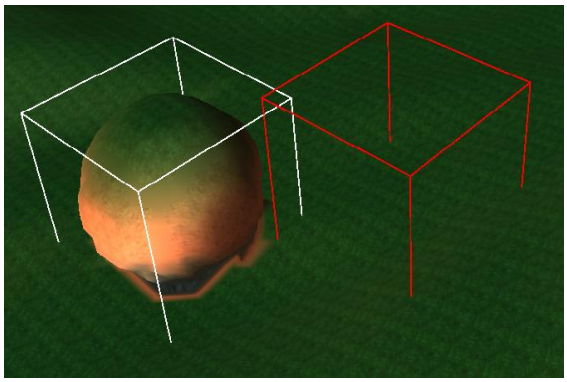


Figure 4: *Small field that's about to get extended.*

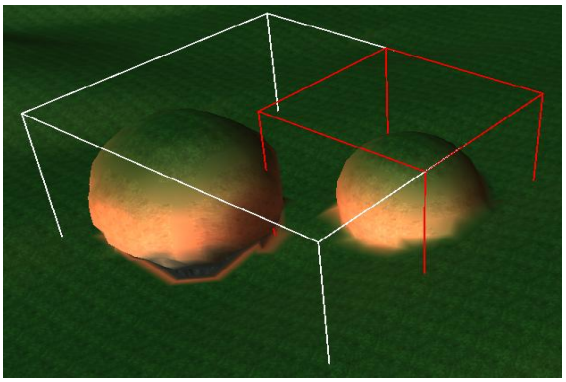


Figure 5: *Field extended to fit the area of effect.*

Scenario 4:

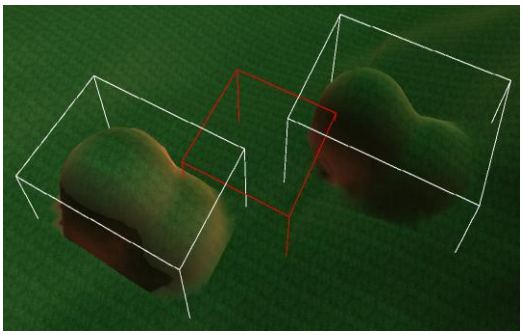


Figure 6: *Two fields about to merge*

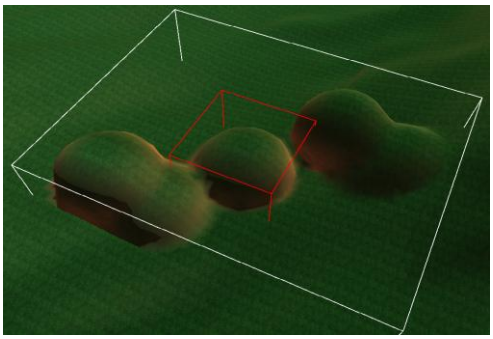


Figure 7: *Two fields merged to a larger field.*

Much like scenario 3 but causes some overhead since several separate marching cubes fields are now supposed to merge together and in most cases we have more data to process and more data to create than in scenario 3. Figure 6 illustrates two fields before a merge and Figure 7 illustrates the two fields fully merged into one.

Scenario 5:

When the area of effect partially intersects two or more grid chunks this causes the area of effect to be sent to all the intersecting chunks to be processed individually and one of the four first scenarios may occur in each chunk. In Figure 8 the area of effect has been placed at the border of four different chunks. In Figure 9 the area of effect was applied to the four different chunks and the result is 4 different marching cubes fields that cover the area of effect.

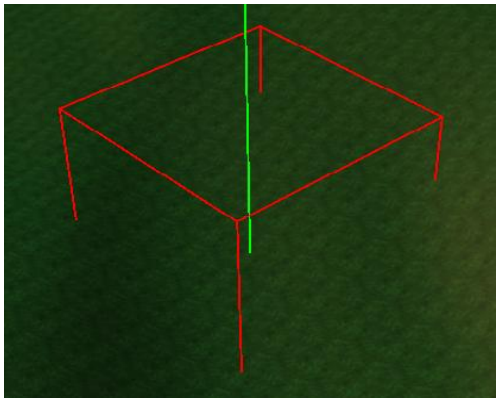


Figure 8: *Area of effect on the edge of four fields*

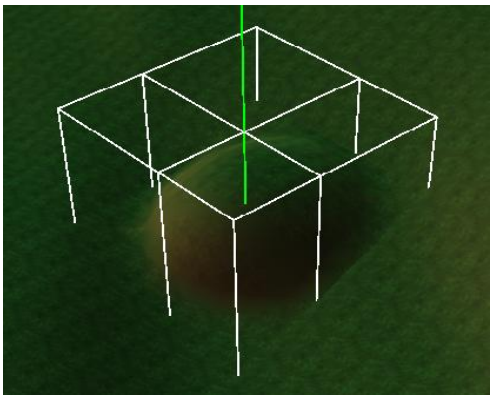


Figure 9: *Four fields created each in a separate chunk*

With these five scenarios the hybrid implementation is complete. They cover all issues that might occur and with using chunks to split up the marching cubes field merged into the terrain. The number of draw calls can be decreased the more filled a chunk is with marching cubes fields. Mesh generation time is also decreased and so is memory usage. This is because the time spent generating a mesh is less the smaller the marching cubes field is and since a field can only reach the size of its parent chunk. And less memory is used because marching cubes field exist only where the terrain is affected so to keep it as limited as possible.

CUDA

CUDA is a programming language created by NVIDIA [4] to let developers program multi-threaded kernels (functions) on the GPU, also called the device, making use of the speed both in processing and in memory usage of a graphics card as compared to a regular CPU, also called the host. This is ideal to make use of if there is a heavy load on a certain algorithm or computation on the CPU side of the code.

The process works so that the data is copied, or transferred, from the CPU (host) part to the GPU (device). Once on the GPU the data can be modified in a kernel doing the same computations as it had on the CPU only faster. With CUDA the advantage of launching a kernel function with

multiple threads and blocks is possible giving the developer the power to parallel parts of the algorithm for even faster execution. A block is like a grid that has the power to launch several threads in its own grid. So a kernel could be launched using only one block but several threads. Or the opposite, many blocks but only one thread per block. Using heavy computations the developer sometimes wish to make use of even more power so combining blocks and threads together yields a very large amount of parallelism. Imagine one block being launched for each iteration in a large for-loop and the threads in this block handles iterations inside this loop so the blocks represents the outer for-loop while the threads represents the inner for-loop. Parallelizing nested for-loops can get complex very fast, but using blocks and threads it is possible to replace loops all together and instead just launch a kernel for each iteration instead.
Once the execution is done on the device the modified data has to be once again copied, but this time back to the host, so the programmers can make some sense of it as the regular data most are used treating with.

The limitations of using CUDA are solely based on the system's graphic card. NVIDIA graphics cards from 2006 or newer has support for the CUDA architecture [4]. The graphics card also decides the amount of blocks and threads that can be launched simultaneously during an execution. Later graphics cards can naturally launch more blocks and with more threads for each block than older graphics cards.

For this paper parts of the Marching Cubes algorithm has been moved to the GPU in hope to speedup performance. This will be explained further in Results (page 13) and Discussion (page 15). Mainly the function GenerateMesh(...), that actually does just so, generates the mesh for marching cubes, has been moved to the GPU and treated as a kernel function instead.
Moving code from the CPU to the GPU can be a really easy matter depending on what you are moving. If there are a lot of dependencies it can be a mess just to copy everything over to the device, which itself is a bottleneck, especially if there is a lot to copy. And let’s not forget the final step, after kernel execution, where everything has to be copied back to the host.
So moving too much data from the CPU to the GPU back and forth is not worth while in the long run, especially if there is a lot of dependencies that are needed on both sides to do the necessary calculations. If that is the case one should really consider moving all code to the GPU or just stay on the CPU.
For the novel hybrid solution there is both a CPU and GPU implementation and both implementations are tried and tested and will be explained later in the thesis.

Methodology

The results are sampled by counting the seconds between starting and finishing each time a field is added or affects the world. This way the other update functions or render calls aren't included in the time measurements. The total, average, minimum and maximum time is considered for each test as well as memory usage. All tests run the same sort of scenario: generating a world using the hybrid technique with a fixed generation seed. However, the last scenario is different because there the whole terrain is replaced by marching cubes from the start, so it is different compared to the other tests. The test environment adds fields of different sizes at different positions with varying polarity, meaning the fields either carve or add onto the world.

Four different scenarios were created for each test, each with different input data for the hybrid technique such as chunk and vertex buffer size.
First test runs the hybrid technique on the CPU, testing the four different scenarios. The same test is repeated with parts of the hybrid solution running on the GPU instead. And finally, the last test, the whole terrain is replaced by marching cubes from the beginning but this test still runs the same sampling scenarios as the other two.
The tests were also initiated twice on each scenario for both CPU and GPU. This is because the first time the test is running there is no data allocated for the marching cubes fields and the second time the data is already created and is only changed. The second run should give a cleaner test result when only measuring the speed of the algorithm since no new memory allocation is needed. However the last test already has memory allocated to begin with since the terrain have already been fully replaced by one big marching cubes field.

All scenarios were repeated ten times to a total of 240 different unique tests.
The tests are run on the following hardware:
OS: Windows 7 64-bit
Graphics Card: GeForce GTX 570, 1280MB Memory
Processor: Intel Core i5-2500 CPU 3.30 Ghz (4 CPUs)
RAM: 4.0 GB, 669 MHz

Results

Chunk size is the maximum allowed size for a marching cubes field. Marching cubes fields cannot go outside the chunk it resides in and the fields can only be merged with other fields within the same chunk. Time is measured in seconds and memory is shown in kilo bytes.

Table 1 and 2 displays the results of the CPU implementation of the novel hybrid solution while table 3 and 4 uses the GPU implementation. Table 5 show the result of the whole terrain being replaced by one large marching cubes field and is running the CPU implementation.

There are some cases where the time is zero. This is of course not entirely true. The time measurement couldn't handle the small timespans that occurred in some of the tests and the result is a truncated value which simply means that the time span is too small to be measured.

Table 1: CPU, First run, chunk size 25 to 100.

CPU first run	**Chunk Size 25**	**Chunk Size 50**	**Chunk Size 75**	**Chunk Size 100**
Total Time	14.955078	6.681641	6.123047	6.855469
Average Time	0.149551	0.066816	0.061230	0.068555
Max Time	1.041016	0.451172	0.363281	0.541016
Min Time	0.000000	0.003906	0.009766	0.001953
Memory	970,636 K	325,192 K	342,020 K	536,464 K

Table 2: CPU, Second run, chunk size 25 to 100.

CPU second run	**Chunk Size 25**	**Chunk Size 50**	**Chunk Size 75**	**Chunk Size 100**
Total Time	0.949219	1.533203	2.462891	3.646484
Average Time	0.009492	0.015332	0.024629	0.036465
Max Time	0.048828	0.044922	0.060547	0.083984
Min Time	0.001953	0.001953	0.005859	0.000000
Memory	976,692 K	331,564 K	353,719 K	536,384 K

Table 3: GPU, First run, chunk size 25 to 100.

GPU first run	**Chunk Size 25**	**Chunk Size 50**	**Chunk Size 75**	**Chunk Size 100**
Total Time	16.369141	7.160156	6.73628	7.212891
Average Time	0.163691	0.071602	0.067363	0.07219
Max Time	1.087891	0.453125	0.373047	0.556641
Min Time	0.003906	0.003906	0.011719	0.003906
Memory	979,628 K	354,084 K	375,580 K	559,296 K

Table 4: GPU, Second run, chunk size 25 to 100.

GPU second run	**Chunk Size 25**	**Chunk Size 50**	**Chunk Size 75**	**Chunk Size 100**
Total Time	2.203125	2.350781	3.011719	4.039063
Average Time	0.022031	0.0220508	0.030117	0.040391
Max Time	0.093750	0.054688	0.074219	0.107422
Min Time	0.003906	0.005859	0.007813	0.001953
Memory	1,003,080 K	356,188 K	376,284 K	559,704 K

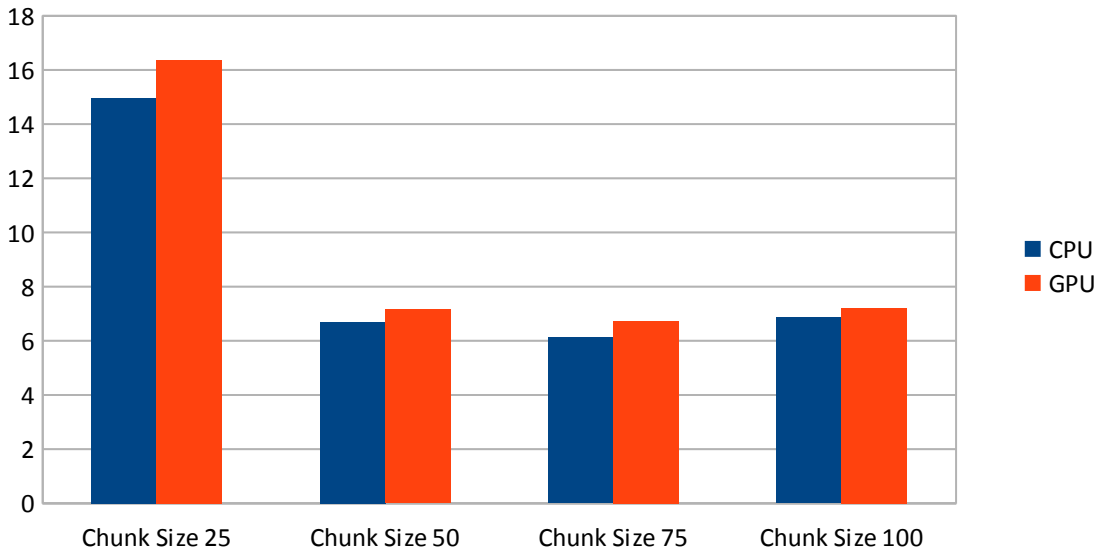
Table 5: CPU, First run and Second run, chunk size 256 (whole terrain).

Full Marching Cubes	**First run**	**Second run**
Total Time	29.783081	29.792974
Average Time	0.297831	0.297980
Max Time	0.978760	0.978760
Min Time	0.117920	0.117676
Memory	869,576 K	872,564 K

Discussion

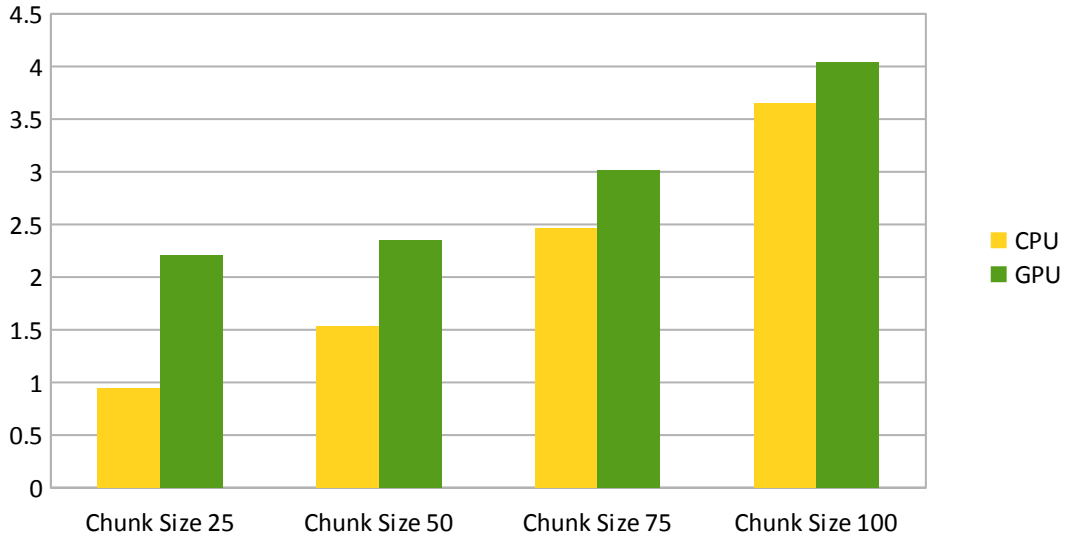
The tests show that the GPU in all cases yields worse results than the CPU both on the first and second run. The worst case is the full marching cubes test which was expected.
The Y-Axis on all diagrams represents time in seconds.

First run



First Run: *Results between CPU and GPU for different chunk sizes using the hybrid solution.*

Second Run

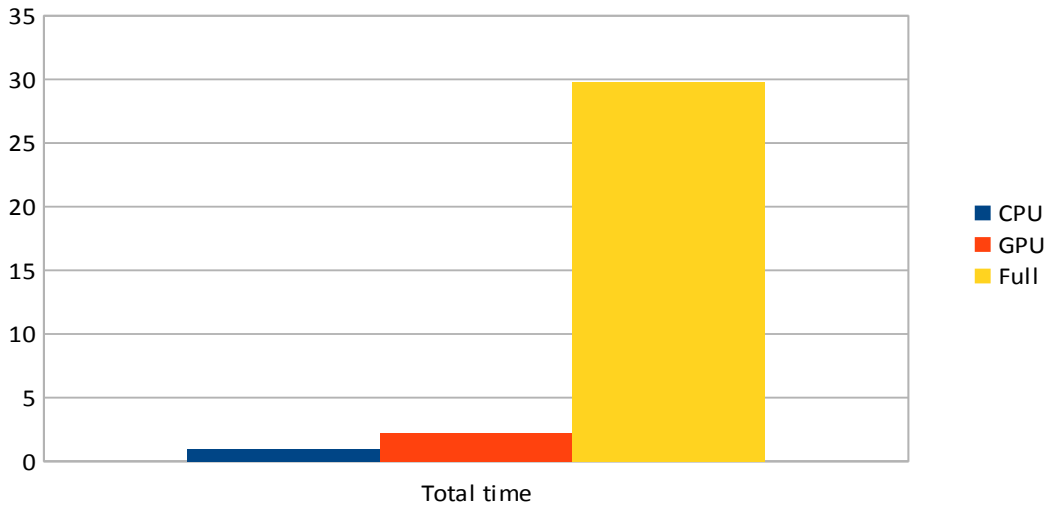


Second Run: *Results between CPU and GPU on the second run using the hybrid solution.*

So why does the GPU always end up with worse results than the CPU?
It's probably because the core of the Marching Cubes algorithm itself isn't that heavy. The bottleneck of the GPU is the pipeline in between where data is sent. The algorithm takes care of huge amounts of data which has to be sent to the GPU and then back to the CPU each time in order to recalculate the mesh. While the calculation of the mesh itself may perhaps be faster on the GPU the transfer of data will slow it down enough for the CPU to get better results.

So how could an implementation be done on the GPU that's faster than the CPU?
Simply by moving parts with heavy calculations to the GPU. The core of the Marching Cubes algorithm is just a lookup and vertex points are pushed to a buffer, no heavy calculations are made. One thing that also might increase performance of the algorithm would be if it could be done with less data transfer to the GPU.
Perhaps future graphics cards with higher memory bandwidths would handle the GPU implementation better but we would still recommend moving other parts to the GPU with heavier calculations and less memory dependencies.
The chunk size also affects the algorithm in two ways, both negative and positive.
Looking at the first run and comparing the chunk sizes by time shows that time spent is greatly reduced when the chunk sizes are increased. The difference is over 50% when comparing both CPU and GPU with chunk size 25 to 50. The performance peak seems to be somewhere between chunk size 50 and 75 because at chunk size 100 the performance starts to degrade again both regarding time spent and memory usage. However when taking a closer look at the second run it's always slower the larger the chunk is. This proves that when fields and data are already allocated beforehand small chunks should be favored but when data isn't allocated larger chunks should be used. On the other hand data only has to be created once and you can still earn that time back by using smaller chunks in long term conditions. On larger chunks the smallest change will cause it to recalculate the mesh, if smaller chunks were used it would have to recalculate smaller parts of the mesh and less time will be spent recalculating parts of the mesh that are still the same.

CPU vs. GPU vs. Full marching cubes



CPU vs. GPU vs. Full MC: *The best total time for all three scenarios: CPU, GPU & Full Marching Cubes*

The diagram displays a time comparison between CPU and GPU with chunk size 25 and the full marching cubes field at the second run.
The CPU chunk solution has a performance increase of 3138% compared the full marching cubes version. This is simply because less time is spent recalculating the meshes vertex points and only the necessary parts are recalculated on the CPU chunk solution. The chunk solution could be used for other things as well such as frustum culling which would allow one to cull away chunks not being currently visible by the camera, meaning unseen chunks will not be rendered. Another option would be when a chunk isn't seen it's saved to the hard drive and removed from the application and once it's seen again it will be loaded in the application again in order to have less memory allocated.

From what we've seen in our results it is clear to us that the hybrid solution works well, at best it will give us the results of a regular heightmap if no marching cube fields are added to it, as expected, and as we go on adding marching cubes to the heightmap the performance drops but it's a necessary drop to make use of the marching cubes algorithm. Sooner or later once more and more of the heightmap is covered by marching cubes we'll get some really nasty performance degradation. However performance will never get as bad as the full marching cubes example because of the usage of chunks that makes sure that less time are spent recalculating meshes. That's why if this technique should be used in a real-time application limitations should be considered of

how much the terrain can be affected or speedup the algorithm in other ways. Say move the whole technique to the GPU or add culling for the chunk based oct-tree to limit the draw calls on the GPU.

An issue we've struggled with was the actual snapping of the terrain and marching cubes vertices. We would get some artifacts seen as cuts and holes where the two techniques merged, and it usually depended where on the terrain we did the merge. The steeper the terrain then more likely we get artifacts. This was fixed by adding vertex points at the edge from the marching cubes field to a list in the terrain. The terrain then used that list to pick out the closest points to snap its vertex points to. The results are much better now and we get less artifacts but the issue still remain. This could be solved by forcing the marching cubes field to stretch on the Y-axis to the min and max of the terrains vertices it will cover.

Another issue we had was when actually removing vertices from the heightmap when adding marching cubes to it. The idea was to keep the vertices as they are in the heightmap but only change the index buffer, meaning how they will be drawn. If we can change the index buffer to only draw the vertices outside of the affected area of marching cubes then we are good to go. And we did get it to work but it took about 15 seconds to calculate the new index buffer, so we scrapped that pretty much all together. We didn't feel like wasting too much time on fixing that since it was not entirely important for our main implementation itself. What we did instead was to simply make the affected vertices on the heightmap invisible. Not the ideal solution but it’s cheap to do and the heightmap itself is not getting any worse.

Conclusion

So this brings us to the conclusion of our paper and we would like to wrap things up by saying the novel hybrid solution works well beyond our initial concerns; both implementation wise and performance.

Our goals have been met and the hypothesis, to create a hybrid solution with marching cubes fields and heightmaps to be able to perform satisfactory on a modern hardware, is supported by the evidence from the results section between Table 2 and Table 5. So using the hybrid solution, with marching cubes and heightmaps working together, works well for modern real time applications. We save both memory and execution time using the hybrid instead of full marching cubes.

Currently the way the implementation works there is one major limitation, the more the terrain is affected by marching cubes fields the more memory is used up and performance will degrade. The implementation could be improved by saving chunks that are culled away to the hard drive and then load them again once they can be seen. This way the amount of memory used will be limited by the amount of marching cubes fields you see and not limited by how much you have affected the terrain.

Reference

[1] Cline. H. E, Lorensen. W. E, "Marching Cubes: A high resolution 3D surface construction algorithm", SIGGRAPH Computer Graphics, July 1987.

[2] Bourke. P, "Polygonising a scalar field", May 1994.
Can be viewed: http://local.wasp.uwa.edu.au/~pbourke/geometry/polygonise/
Last visited, 2012-5-23.

[3] Frank D. Luna, "Introduction to 3d Game Programming With Directx 10", Wordware Publishing Inc., 2008, ISBN 9781598220537

[4] Sanders. J, Kandrot. E, "CUDA by example: an introduction to general-purpose GPU programming", Addison-Wesley Professional, 2010, ISBN 0131387685

[5] Cline. H. E, Lorensen. W. E, inventors; General Electric Company, assignee. System and method for the display of surface structures contained within the interior region of a solid body. United States Patent 4710876. 1987 Jun 5

[6] Lengyel .E, ”Game Engine Gems 1”, Jones and Bartlett, 2010, ISBN 0763778885.

[7] Löffler. F, Müller. A , Schumann. H, "Real-time Rendering of Stack-based Terrains", Vision, Modeling, and Visualization, 2011.
Can be viewed: http://www.informatik.uni-rostock.de/~fl/assets/vmv_2011.pdf
Last visit: 2012-6-9

[8] Johansson. G, Carr. H, "Accelerating Marching Cubes with Graphics Hardware", Linköping University, Sweden, University College Dublin, Ireland, 2006.
Can be viewed: http://ivg.ucd.ie/files/shared/JC06_GPUMarchingCubes.pdf
Last visit: 2012-6-9

[9] Alonso. J.A, Joan-Arinyo. R, "THE GROUNDED HEIGHTMAP TREE - A New Data Structure for Terrain Representation", Barcelona Tech, Spain, 2008.
Can be viewed:
http://nguyendangbinh.org/Proceedings/VISIGRAPP/2008/VISIGRAPP%202008/GRAPP%202008/Short%20Papers/C1_096_Joan-Arinyo.pdf
Last visit: 2012-6-9