Bachelor of Science in Computer Science
June 2019

Real-time Terrain Deformation with Isosurface Algorithms

Olle Nässén, Edvard Leiborn

Faculty of Computer Science, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden

Contact Information:
Author(s):
Olle Nässén
E-mail: olna15@student.bth.se

Edvard Leiborn
E-mail: edle16@student.bth.se

University advisors:
Stefan Petersson and Hans Tap
DIDA

ABSTRACT

Background. Being able to modify virtual environments can create immersive experiences for video-game players. Storing data as volumetric scalar fields allows for highly modifiable 3D environments that can be converted into GPU-friendly triangles with isosurface algorithms. Using scalar fields and isosurface algorithms can be more computationally expensive and require more data than the more commonly used polygonal models.

Objectives. The aim of this thesis is to explore solutions to modifying real-time 3D environments with isosurface algorithms. This will be done in two parts. First in terms of observing how modern games deal with storing scalar fields, researching which isosurface algorithms are being used and how they are being used in games. The second part is to create an application and limit the data storage required while still running at a real-time speed.

Methods. There are two methods to achieve the aim. The first is to research and see which data structures and isosurface algorithms are being used in modern games and how they are utilized. The second method will be done by implementation. The implementation will use the GPU through compute shaders and use marching cubes as isosurface algorithm. It will utilize Christopher Dyken's Histogram Pyramids for stream compaction. Two different versions will be implemented that differ in terms of what data types will be used for storage. The first using the data type char and the second int. Between these two versions, the runtime speed will be measured and compared on two different hardware configurations.

Results. Finding good data on what algorithms games use is difficult. Modern games are using scalar fields in many different ways: Some allow almost complete modification of terrain, others only use it for a 3D environment. For data storage, octrees and chunks are two common ways to store the fields. Dual Contouring appears to be the primary isosurface algorithm being used based on the researched games. The results of the implementation were very fast and usable in real time environments for destruction of terrain on a large scale. The less storage intensive variation of this implementation(char) gave faster results on modern hardware but the opposite(int) was true on older hardware.

Conclusions. Modifying scalar field terrain is done at a very large scale in modern games. The choice of using Dual Contouring or Marching Cubes depends on the use-case. For areas where sharp features can be important Dual Contouring is the preferred choice. Likely for these reasons Dual Contouring was found to be a popular choice in the studied games. For other areas, like many types of terrain, Marching Cubes is very fast, as can be seen in the implementation. By using the char version of the implementation, interacting with the environment in real-time is possible at high frame-rates.

Keywords: Isosurface, Marching Cubes, Terrain Deformation, OpenGL, Compute Shader

Table of Contents

List of Figures

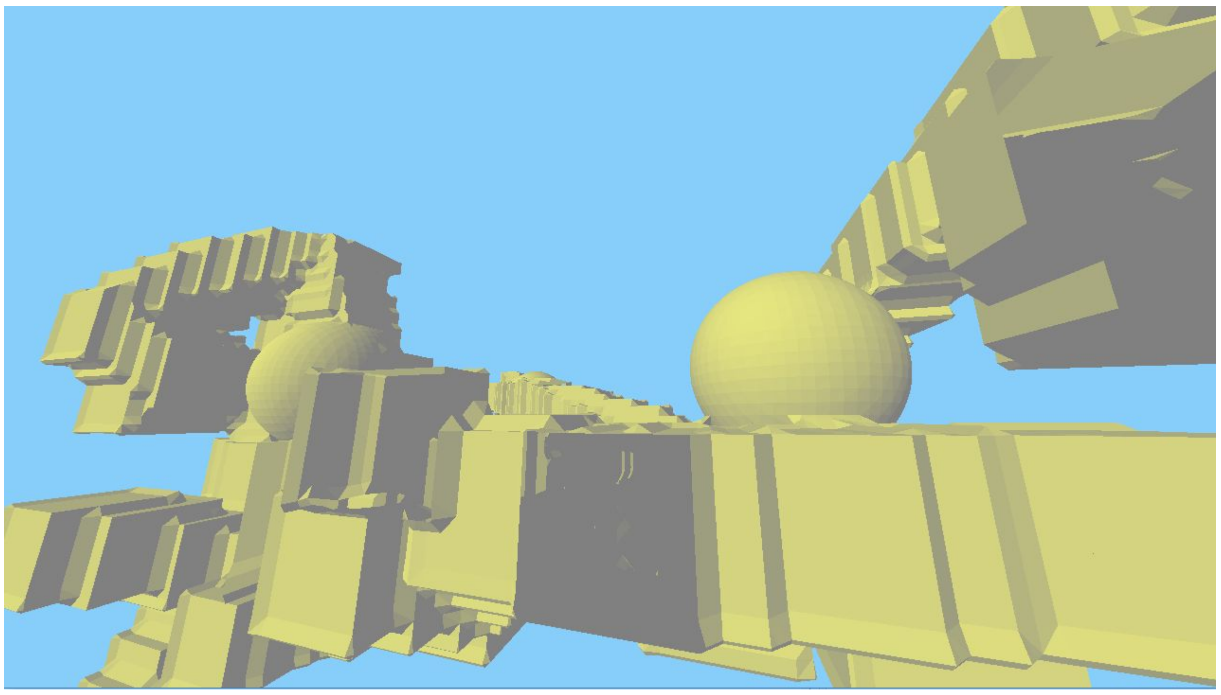

Figure 1: Marching Cubes Implementation

1. Introduction

For video game players to be able to modify the worlds they are playing in can be a very immersive experience. Video games have allowed players to interact and modify their environment for many years now. Back in 1975 the arcade game "Gun Fight" was released which allowed players to shoot down parts of the trees the players could use for cover. There are other early examples such as "Space Invaders" where similarly the cover can be destroyed piece by piece by the aliens. As game developers have had access to increasing computing power over the years, the scope of what players could potentially modify has increased significantly. The 2001 shooter Red Faction managed to implement a terrain that was highly modifiable in a 3D environment. In Red Faction you were not only able to use parts of terrain as cover but you could also create new paths by shooting and destroying/molding the environment.

3D games today have two main ways of storing terrain, height-maps and volumetric scalar fields[3]. Height-maps are 2D textures where each pixel represents the height of a given point in the terrain. The vast majority of software products such as games use height-maps today[3]. Height-maps are easy to use, relatively fast and does the job in most cases. There are limitations with height-maps however. Because of the limitation of having one height per point, it is not possible to create true 3D terrain like tunnels, cliffs and caves. Volumetric scalar fields is a collection of points in 3D space and would thus solve this problem by having many height values. However, it is also more expensive computationally and naturally requires more data[16].

A volumetric scalar field is a room filled with points which represents a scalar value. The scalar value of each point will represent one of three things, if it is inside, outside or on the edge of an entity that exists in the world. What this means is that if you have for example a box in a room, the points inside the box will be positive, the points outside will be negative and if a point lies on the edge of the box they will be on the edge. The scalar values can be described as a distance to where the surface will be created. 3D games today are rendered with triangles, so all the points in the scalar field possibly needs to be translated into triangles depending on their scalar value. There are many solutions for translating scalar fields into triangles, some of which will be featured in related work.

The first part of this thesis is going to explore how terrain deformation is currently being done in games that utilize scalar field based systems. One of the disadvantages of scalar fields is as previously mentioned, the large amount of data required. Because of this, the second part of this thesis will be to implement two different versions and compare both of those methods in terms of which can store data most efficiently while still maintaining an acceptable real-time frame rate. Real-time acceptable frame-rate in all cases in this thesis refers to above 60 frames per second.

The first version will store the distance data as char (1 byte) which will need slightly more processing on the GPU. This is for technical reasons as some conversions will have to be done on the GPU, on the fly, to have data types of this size. The second version stores the data as ints (4 bytes), which means that this version will use four times as much memory but will require slightly less processing.

1.2 Research Questions

RQ1: How feasible is it to modify scalar field terrain in modern games?
RQ2: How can scalar field terrain data be stored and pipelined to the GPU in an efficient way?

The first research question, “How feasible it is to modify scalar field terrain in modern games?” will be looked at from two different viewpoints. To what extent is modifying scalar field terrain being done in modern games today and how the games achieve those standards are two questions that will be discussed in this context.

The second research question will be tested by measuring the amount of data needed to store the scalar field on the CPU and how to efficiently pipeline this data to the GPU for surface extraction and rendering as quickly as possible.

1.3 Hypothesis

The expectations for RQ1 are that it is feasible to a large extent but we are unsure of how it is achieved. For RQ2, storing and sending the scalar field data as the data type char/byte seems to be an efficient method. The processing requirements this will need compared to the int version are expected to be negligible.

2. Related Work

2.1 Marching Cubes

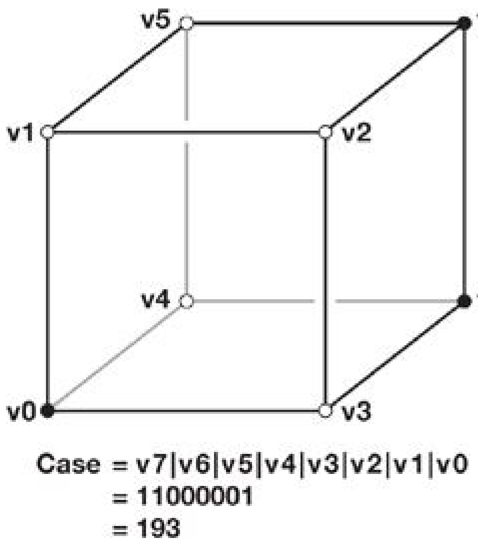

Figure 2: Marching Cubes Cell, picture from GPU GEMS 3[11]

The Marching Cubes algorithm was invented in 1987 to process 3D medical data and convert it into polygons which is then used in a 2D representation[1]. Marching Cubes has become the de-facto standard for solving the problem of converting volumetric data into polygons[2]. A brief explanation of how Marching Cubes works: in the the scalar field, eight points represents as a cell[Figure 2]. By adding the distance values of the points into a single byte**,** you get a case number which is then used to index a couple of look-up tables that will output how many vertices should be created and along which edges of the cell those vertices should be created on.

As GPUs have become progressively more powerful over time, many Marching Cubes implementations have been created on the GPU[2, 11]. Putting parts or all of the Marching Cubes algorithm on the GPU for large data sets has shown to be superior in speed compared to doing the previous full CPU implementations[11].

2.2 Surface Nets

Like Marching Cubes, Surface Nets was first created to solve problems in the medical field. Surface nets is a dual method as described by Sarah F.F.Gibson[5] that creates vertices in all bipolar cells. The position of each vertex is then relaxed to reduce an energy measure in the links.

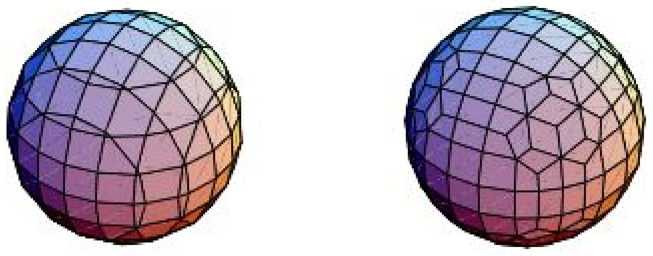

Figure 3. Marching Cubes to the left, Surface Nets to the right. Picture from[6].

In Figure 3, notice the edges of the image to the right in comparison to the image to the left vertices in the right image is dual to the ones on the left. In other words the vertices are placed within the cell a opposed to on the edges like in the left image.

2.3 Dual Contouring

Another dual method that is commonly used is Dual Contouring. Just like the Surface Net algorithm Dual contouring is an isosurface extraction technique the creates a mesh dual to the surface and places the vertices inside the cell in comparison to marching cubes that places vertices in grid edges. However the Dual Contouring technique can also keep sharp edges and corners within cells while normal marching cubes can not. Dual contouring generally creates higher quality meshes from being a dual method. It typically also has a lower triangle count[6] compared to Marching Cubes as vertices are reused. A minimizing quadratic error function is used to extract the exact position of the vertex within a cell. The original implementation used an octree representation with recursive functions to generate the mesh[6]. The Dual Contouring algorithm works on hermite data which allows for sharp edges and corners[6], however the hermite data also uses up more space and memory for storage.

2.4 Histogram Pyramid/HistoPyramid

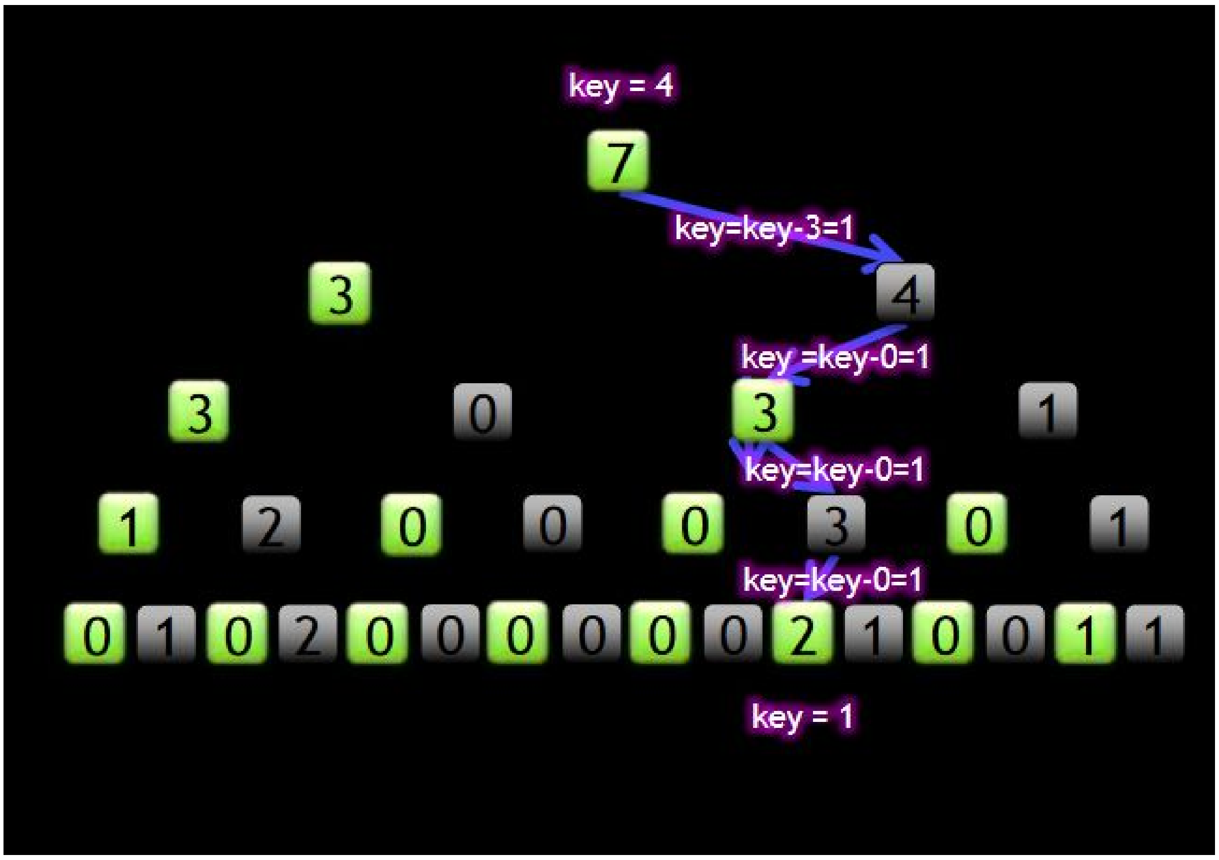

Figure 4: Histogram pyramid lookup, picture from Nvidia[2]

Histogram Pyramids is a method to solve the problem of stream compaction in compute shaders[2, 4]. Histogram Pyramids can be used in many GPU related contexts like rendering and tessellation. The use-case for this article is a Marching Cubes based solution. The implementation in the method section will use the 4-to-one Histogram Pyramid solution as presented by Christopher Dyken et al.

Histogram pyramids creates a lookup table for arrays of sparse data by adding values of a lower layer together and place them in a higher layer of the pyramid until the top is reached. This means the top contains all the values in starting array added together. This can be done on multiple values at once. For example 4-to-1 Histogram Pyramid adds four values together at once for each layer in the pyramid. This pyramid can then later with the help of a key be traversed from top to bottom to find the corresponding index in the original array of sparse data. This traversal is done by subtracting each value from the ones assigned by the pyramid layer until on that is greater than the key is found. When found traversal will continue in the next layer corresponding to the position that was greater that the remainder of the key. Figure 4 shows this process for a 2-to-1 Histogram Pyramid. The process is the same apart from only stepping through two values per layer instead of four.

2.5 Data Storage: Octrees and Chunks

An octree subdivides data into eight nodes(children) per node for each level as can be seen below in Figure 5. One of the benefits of Octrees and a common use-case is that for every level in the octree have a progressively higher level of detail, with the lowest level having the highest. Subnautica stores the scalar field data in batches where every batch is subdivided into three dimensions by 5^3 octrees[14]. In addition to the scalar field distance data, they also store material per voxel and pointers to the children.

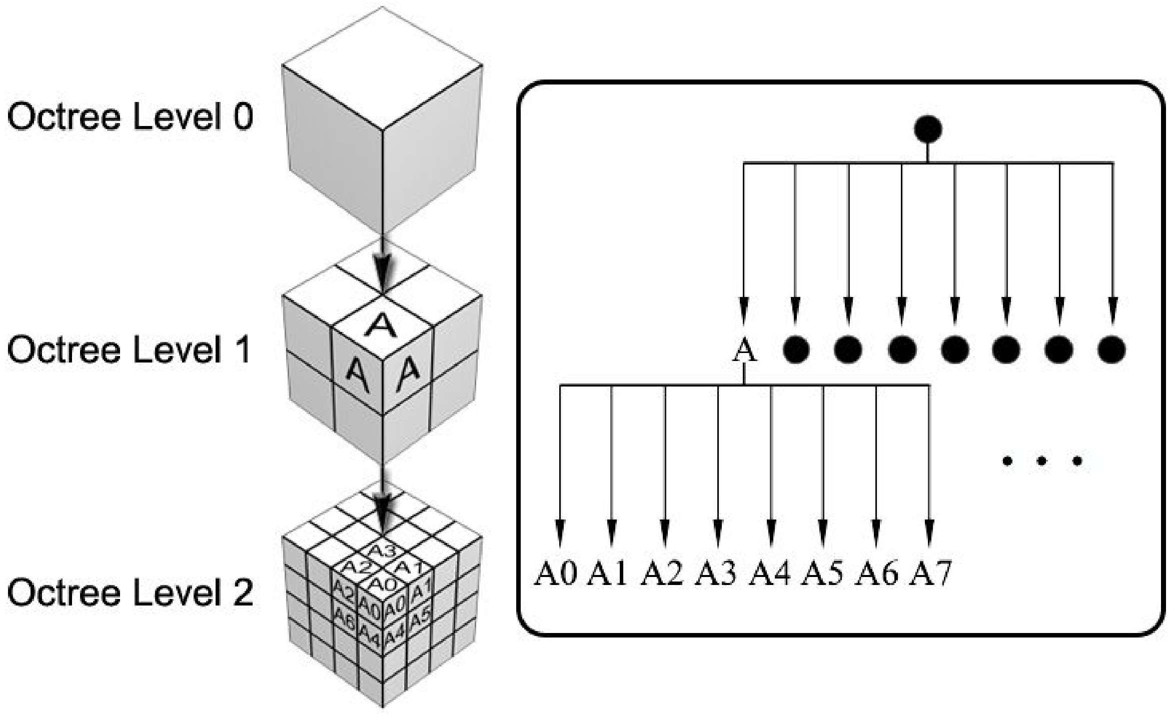

Figure 5: Octree subdivision, picture from[19]

Chunks are 3D arrays that are loaded/unloaded into/from memory when needed. Chunks are not subdivided like octrees, instead each point in space is equally spaced. This generally results in higher memory usage than octrees, but because all points in space are known it is easier to work with in environments where a lot of parallel processing is available. This makes chunks a good fit for solutions that utilizes the GPU, since the GPU is very parallel in processing data.

3. Method

3.1 Game Research

There are many techniques relating to deformable terrain, unfortunately not all of it is published as game engines tend to use proprietary techniques[13]. For that reason it was difficult to find good data on the topic. Because of that, some of the references and citations will be from developers posting on blogs and forums.

The games/engines that we found info on were: Subnautica, EverQuest Next/Landmark(Voxel Farm), Voxel Plugin and No Man’s Sky.

3.2 Implementation

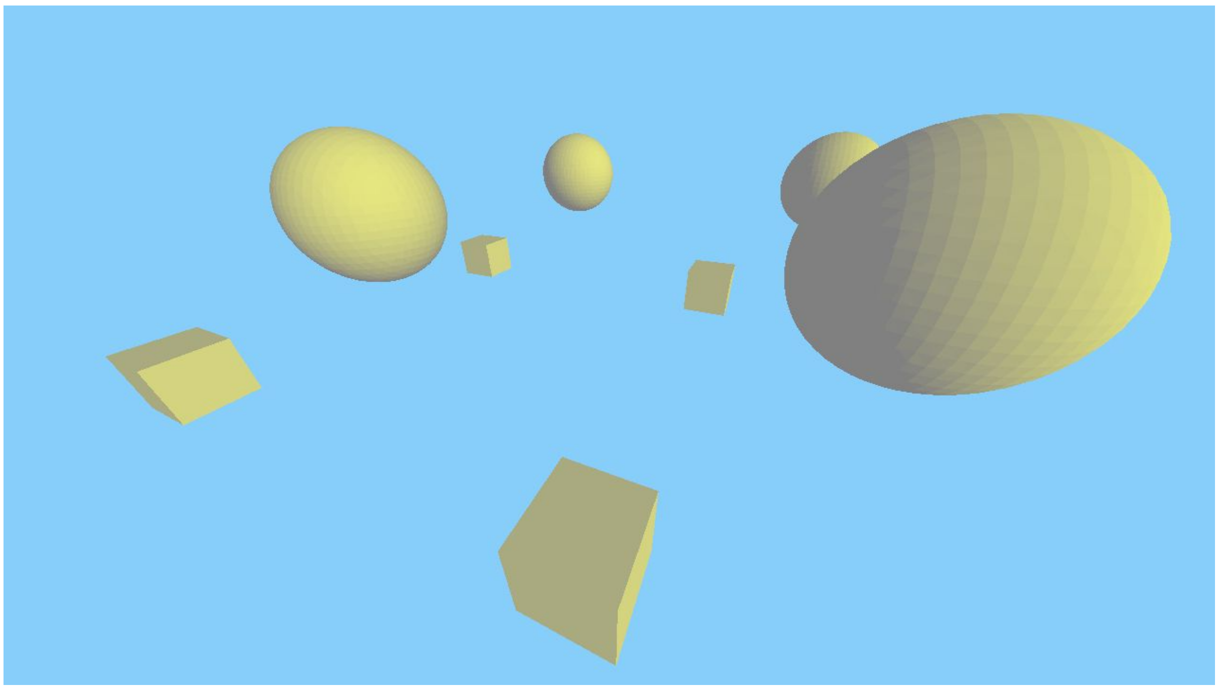

Figure 6: Marching Cubes Environment with Addable Primitives

As seen in Figure 6, the user can add primitives such as spheres and cubes to the testing environment in real-time. This is done by modifying the scalar field on the CPU.

Two lookup tables are stored in a buffer on the GPU. The first table has the number of vertices that should be created in each of all the 256 Marching Cubes cases. The second table stores 256x16 values and they represent the edges of which interpolation will occur when creating the vertices.

The scalar field is created on the CPU in chunks. 65x65x65 distance values are stored in chars per chunk. The isovalue is 0 which represents terrain surface, positive values means they are inside solid terrain, negative values means they are outside. Each char has the value -100 at the beginning representing all empty space. The scalar field is passed on to the GPU every frame.

Compute shaders are utilized in two steps. First for processing each cell in parallel and determining how many vertices should be created in every cell. Determining how many vertices belongs to a cell is done by checking the Marching Cubes case and using that output to index the number of vertices in the first lookup table. Each cell stores this information in the first layer of a buffer. This bottom layer is 64x64x64(total number of cells) of int values, which again represents the number of vertices per cell. This bottom layer will be used as the foundation of the histogram pyramid.

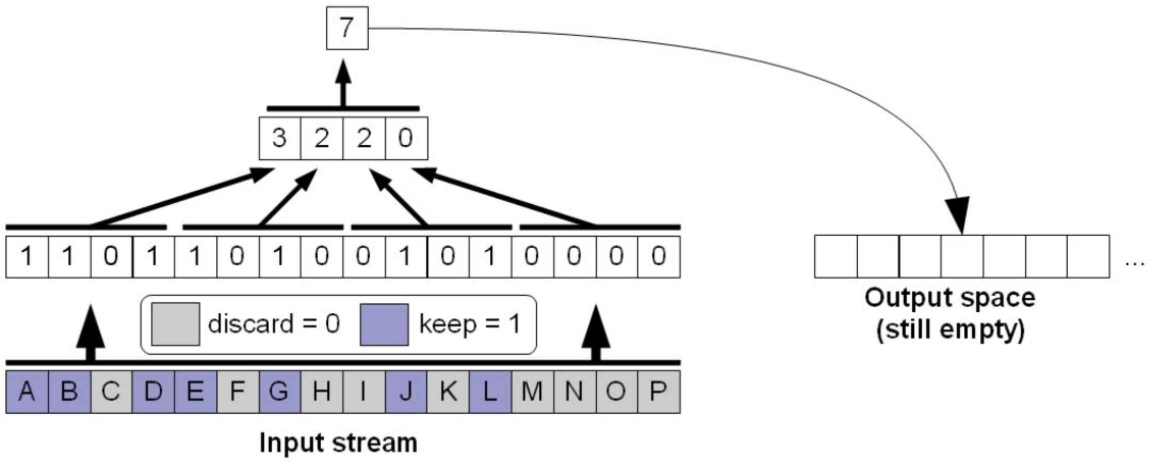

Figure 7: HistoPyramid buildup, picture from[20]

The second compute shader step is to create the rest of the histogram pyramid, in accordance with Christopher Dyken et al. 4-1 Histogram pyramid. Each layer is the size of the layer below divided by four until you get to the top. The layers of the pyramid are generated by for each layer, in sequence summing up four values and adding that sum to the next layer. As can be seen in figure 7 the top will then eventually consist of a single value that represents the total number of vertices that should be generated from all cells. This second step could be seen as many steps as one pass is dispatched per pyramid level until the top.

Once the pyramid has been generated the histogram pyramid buffer is bound to the vertex shader.

When calling to use the vertex shader glDrawArraysIndirect is used. glDrawArraysIndirect allows the number of vertices to be defined in the previous compute shader stage and can therefore avoid any GPU to CPU to GPU passes.

In the vertex shader the histogram pyramid will be traversed from top to bottom. This is achieved by using gl_VertexID as key to find the correct vertex and then pass it to the fragment shader for shading.

In order to answer RQ2, two different versions were developed. The first sends the scalar field as an int array to the GPU. The second sends the scalar field as an int array but instead stores four chars in each int. They are later extracted in the compute shader and in the vertex shader.

Keeping data storage sizes down while still maintaining high speed was the goal of the second research question. That is why tests were done on two versions. The first one using chars and the second test with ints. Because there is no built in char data type in glsl extra work needs to be done on the GPU to extract the char data but the space required is significantly less. The int version uses 1098500 bytes (1.05MB), the char version uses only 274625 bytes (0.26MB). This is almost four times the data for ints. This data needs to be stored on both GPU and CPU which means it takes up twice the amount of space required.

Because glsl has no built in support for the char data type (1 byte), the bits had to be manipulated by hand by placing 4 of the in one int (4 bytes). The data therefore had to be extracted on the fly in the GPU compute shader stage and vertex shader stage to find the correct distance values. By storing four of the distances in one int, the data had to first be bit shifted to the proper location. After that the correct amount of bits were kept and the rest discarded. To get the correct sign of the data a built in function in glsl called bitfieldExtract was used. The data was then temporarily stored in full ints again because there is no char data type by default in glsl.

The implementation used 65x65x65 distance values in the isosurface scalar field. This field contained either char or int data representing the distance to the surface of the mesh in millimeters. A single cell was 100 millimeters in size if the specific cell was further than 100 millimeters away from the surface it stored either -100 or 100 depending on if it was outside or inside the mesh. This was done to save memory since char (1 byte) only can hold values in the range -128 to 127 and because values outside this range was not needed to represent the mesh.

This data could then be used to determine how many times the vertex shader needed to run to create the mesh of the field. In the vertex shader the 4-to-1 Histogram Pyramid was used to determine the index inside the cell this specific vertex was with help of the gl_VertexID. Knowing the cell and the vertex id the lookup table could then be used to find the correct edge case for this specific vertex as can be seen in figure 8. The interpolation could then be done on the fly in the vertex shader to find the final position of the vertex. Because of this the isosurface scalar field could be manipulated on the CPU in real-time while running the application without it affecting the performance of the program significantly. The only extra work needed is the lookup and calculation of the shape that is being placed and where to place it in the field. The extraction of the mesh is not affected, but higher triangle count could potentially affect the performance of the vertex shader in very large meshes.

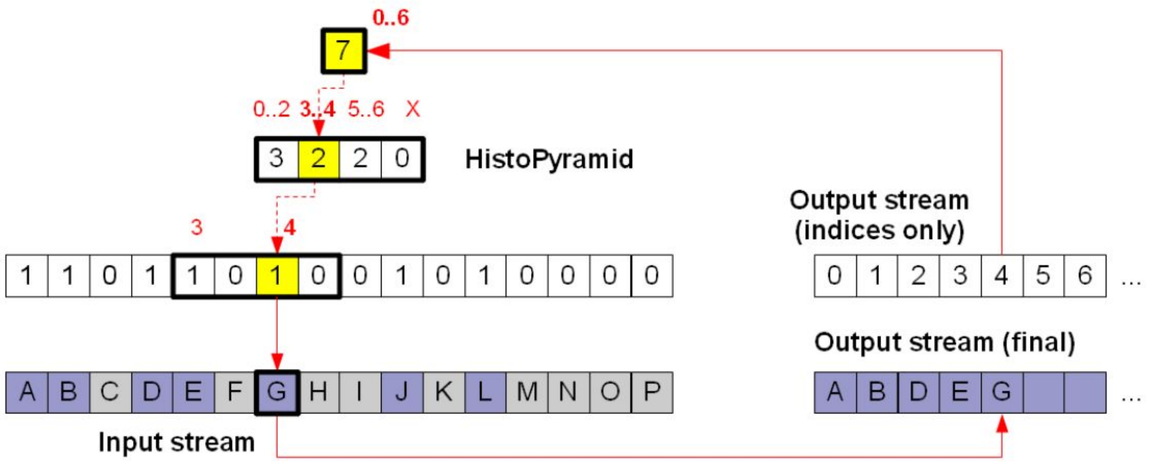

Figure 8: HistoPyramid traversal, picture from[20]

3.3 Experimental Setup / Benchmark

Testing was performed on two different sets of hardware.

1. Windows 10, Intel i7-8750H 2.2 GHz, 4.1 GHz w/Turbo, Nvidia GeForce GTX 1070(Max-Q) with 8GB VRAM(GDDR5)
2. Windows 10, Intel i5-3210M, Nvidia GeForce GTX 660M with 2GB VRAM(GDDR5)

The application ran at 720p and was using OpenGL version 4.3 and GLSL version 430. The tests for the GPU versions are a time based average(in seconds) of the entire game loop per 30 frames. Testing was done on 1-4 chunks of data. Each chunk containing 65x65x65 distance values of either char or int type. Each chunk is processed independently of all other chunks doing all the processing steps before continuing with the next chunk. This was done up to four times per frame (1-4 chunks).

The environment was built on C++, OpenGL and its built in Compute Shaders. The user can add primitives to the running environment.

The goal of these tests is to answer RQ2 and find an efficient method to pipeline the scalar field data to the GPU. The experiment will research whether it is most efficient to store all scalar field data as the datatype char or as int. As previously mentioned the char version will need to bit extracted on the GPU but will require less data.

Surface normals are calculated per triangle in the geometry shader which are then used in the fragment shader. The fragment shader shades everything with a phong-like shader that only has one directional light as seen in Figure 9.

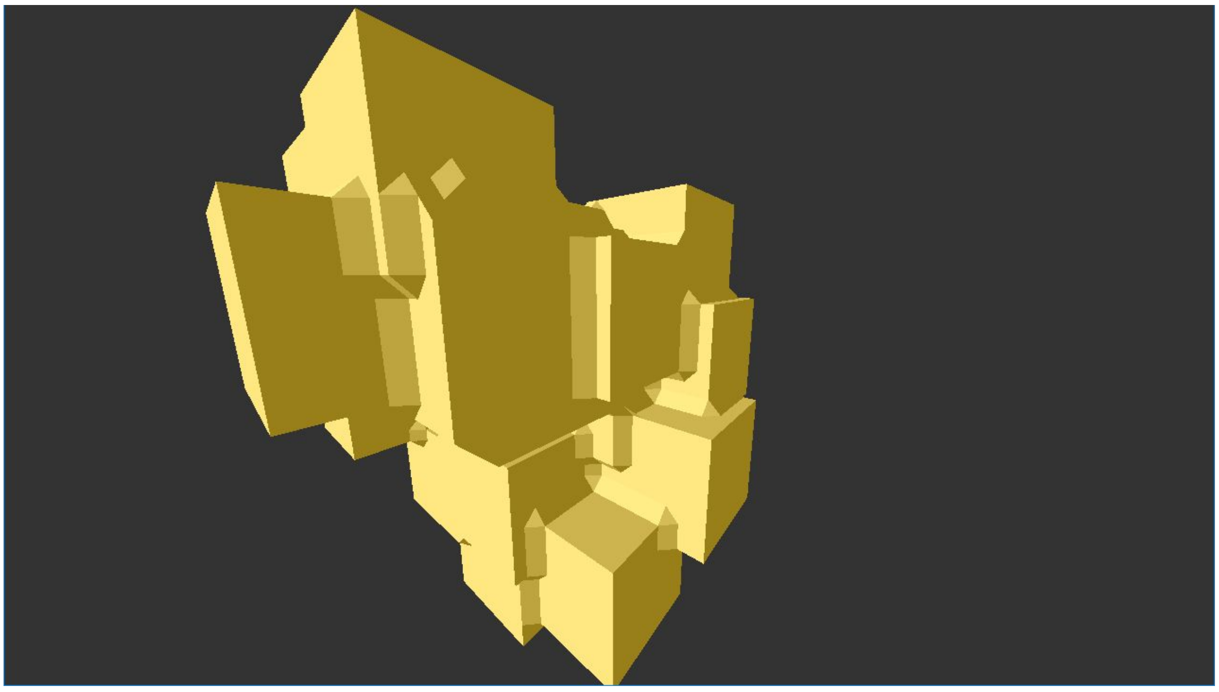

Figure 9: Simple shading with surface normals

4. Results

4.1 Game Research Results

Subnautica uses Dual Contouring to create their polygon surfaces[14]. For Subnautica to use an isosurface algorithm is a logical choice as the game revolves around swimming in underwater- tunnels and caves. This would not be possible with a height-map based terrain. Data is stored using an octree[14]. Worth noting is that in the release version of Subnautica it is not possible to modify the terrain.

Everquest Next was a game that was cancelled in 2016. It used the Voxel Farm editor which is a plugin for the game engines Unity and Unreal Engine. Voxel Farm allows developers to create highly interactive and modifiable voxel based worlds. The full source code for Voxel Farm is proprietary but the author has a blog where some information has been shared[16]. Some of this information is used in our implementation below. Similar to Subnautica, Voxel Farm uses Dual Contouring which is discussed in the above mentioned blog. Everquest Next was one of the first games to use Voxel Farm. However the game was later cancelled and the reasons provided by the president of the company was that the game wasn't fun to play[18].

Voxel Plugin is a tool for Unreal Engine which similarly to Voxel Farm allows users to create modifiable voxel worlds. Voxel Plugin uses a Marching Cubes implementation based on Transvoxel which was invented by Eric Lengyel[3].

No Man's Sky is another game that utilizes scalar field terrain with dual contouring[17]. Instead of using the GPU to generate the meshes No Man's Sky generates all meshes on the CPU[17]. It is not clear if this is done once or every frame. There is also no detail on the size of the scalar fields used.

4.2 Implementation Results

Figure 10 shows all the results for 1-4 chunks of data of both the char and int version of the algorithm using GTX 660M. Figure 11 shows the result for one chunk. Figure 12 for two chunks, figure 13 for three chunks and figure 14 for four chunks respectively. The increase in performance per chunk is around 0,01 seconds more per chunk for both versions. For this set of hardware the char version is around 0,005 seconds slower in all cases.

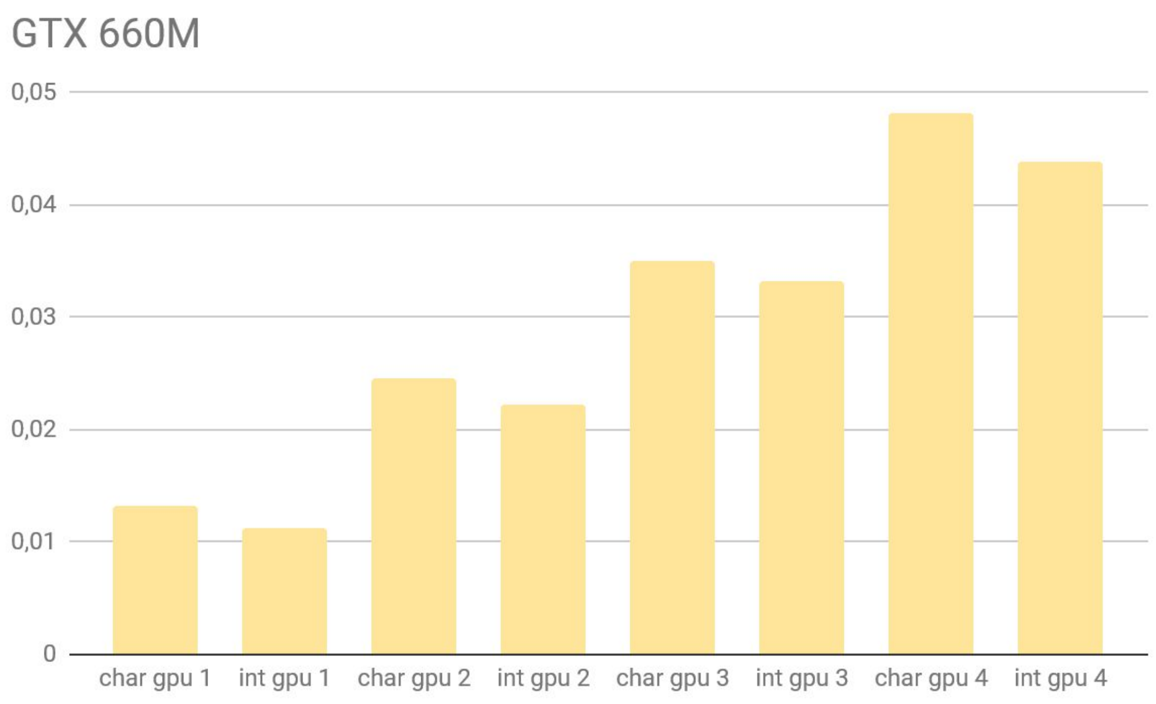

Figure 10. Comparison of char and int versions on GTX 660M for one to four chunks of data 65x65x65.

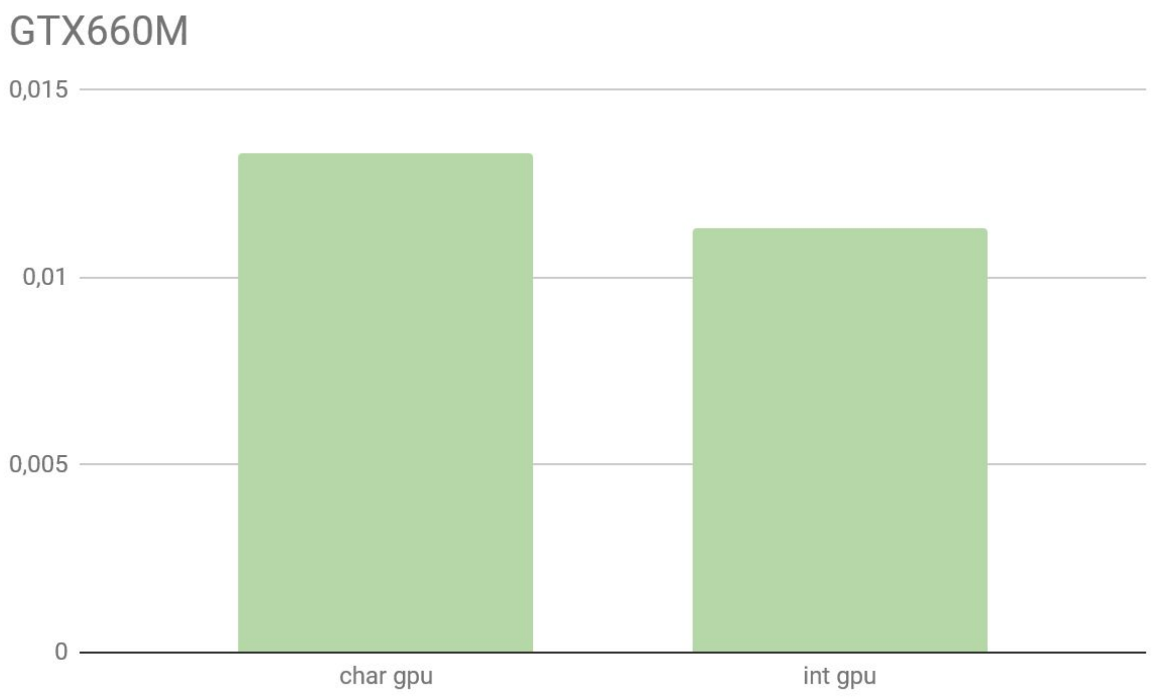

Figure 11. Using 1 chunk 65x65x65 with GTX660M

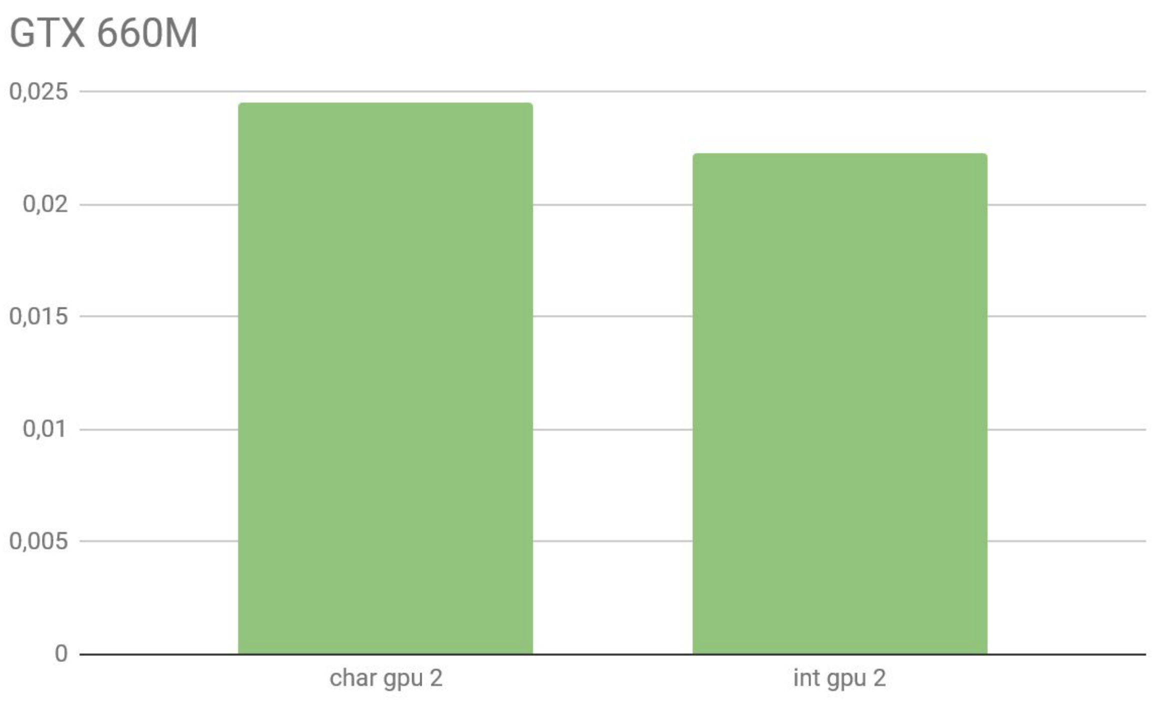

Figure 12. Using 2 chunks 65x65x65 with GTX660M

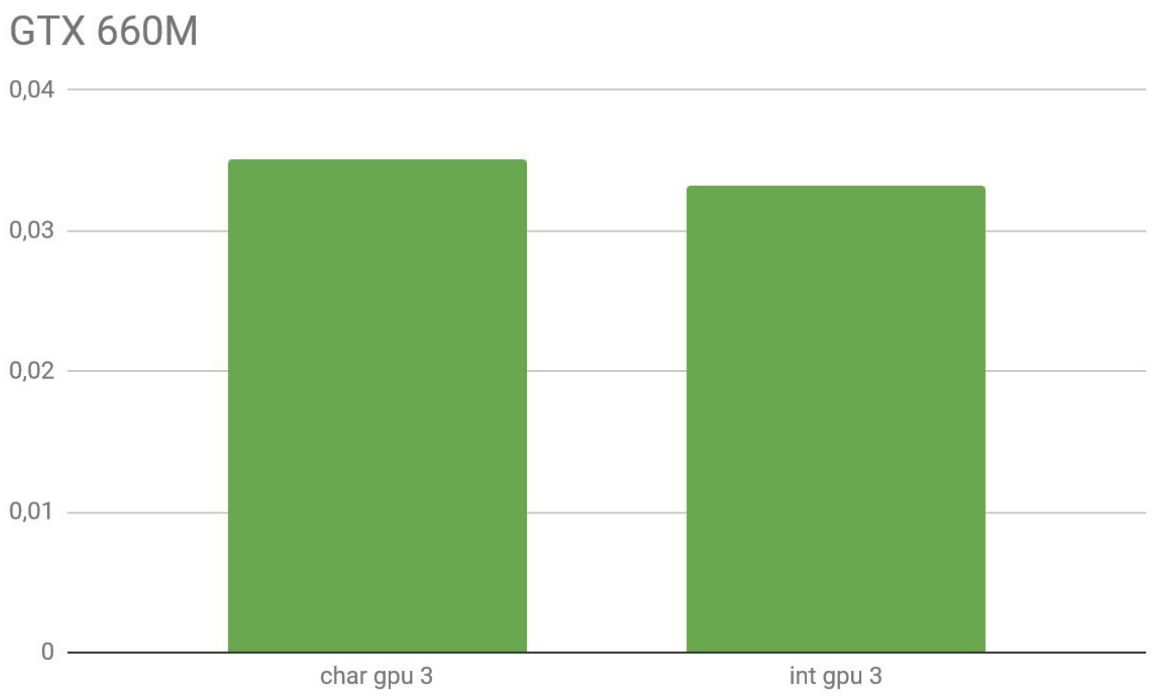

Figure 13. Using 3 chunks 65x65x65 with GTX660M

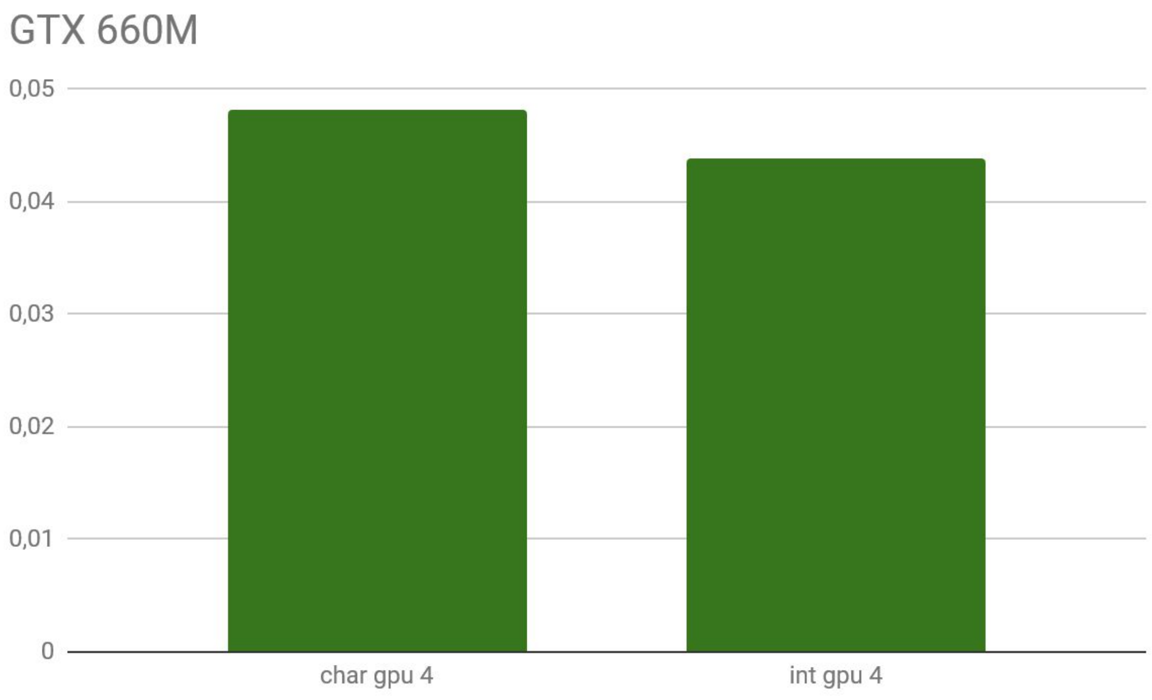

Figure 14. Using 4 chunks 65x65x65 with GTX660M

Figure 15 shows all the result for 1-4 chunks of data of both the char and int version of the algorithm using GTX 1070(Max-Q). Figure 16 shows the results for one chunk. Figure 17 for two chunks, figure 18 for three chunks and figure 19 for four chunks respectively. The increase in performance per chunk is around 0,001 seconds more per chunk for both versions. For this set of hardware the int version is around 0,005 seconds slower in all cases.

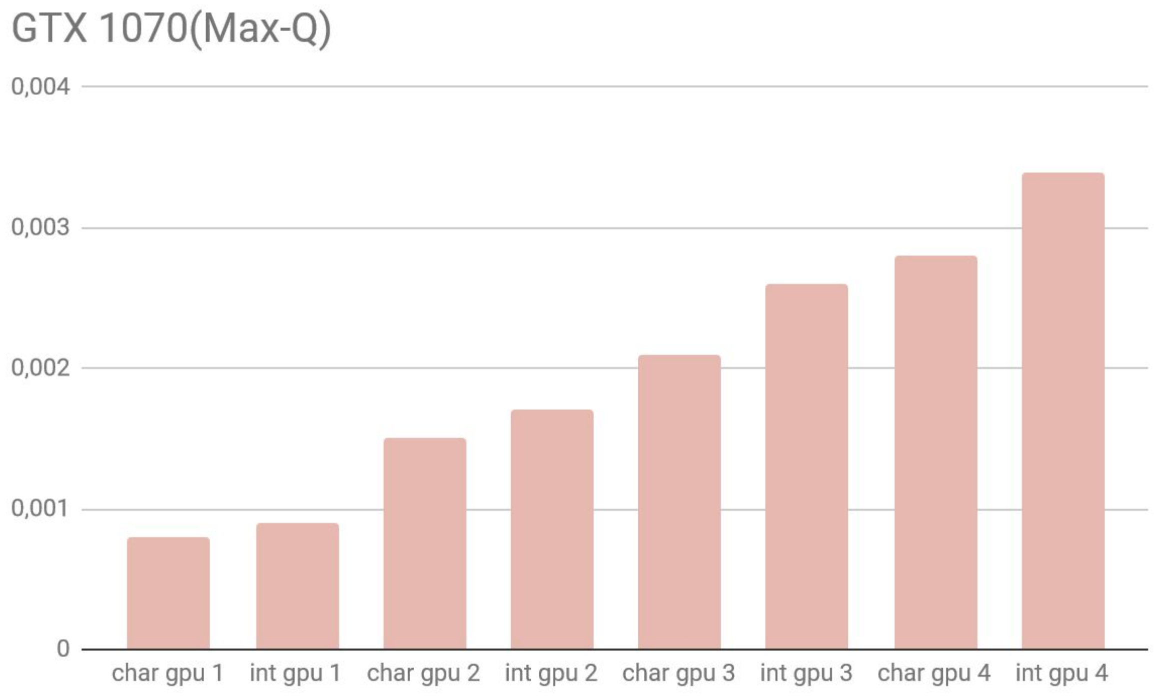

Figure 15. Comparison of char and int versions on GTX 1070(Max-Q) for one to four chunks of data 65x65x65.

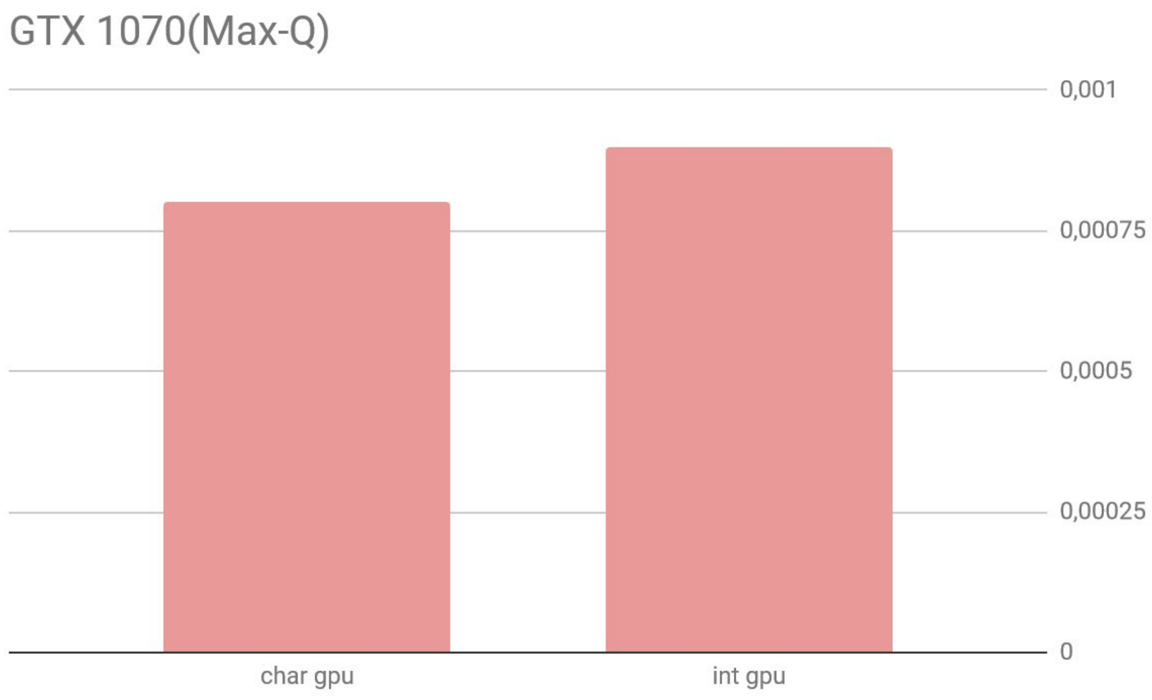

Figure 16. Using 1 chunk 65x65x65 with GTX 1070(Max-Q)

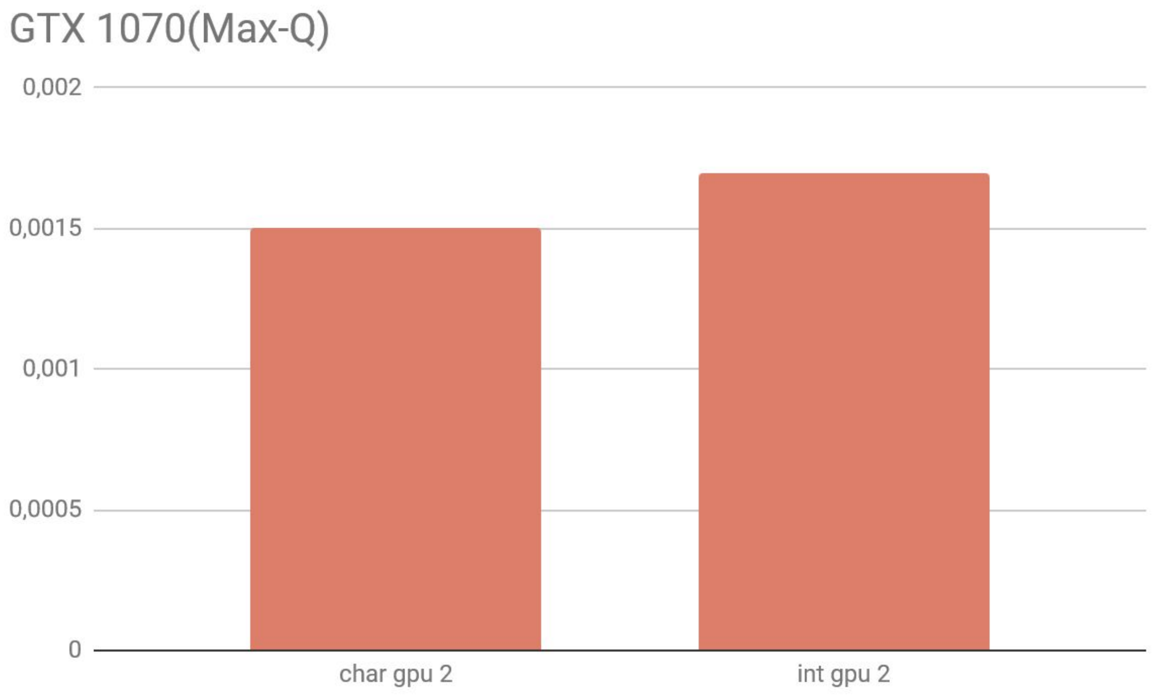

Figure 17. Using 2 chunks 65x65x65 with GTX 1070(Max-Q)

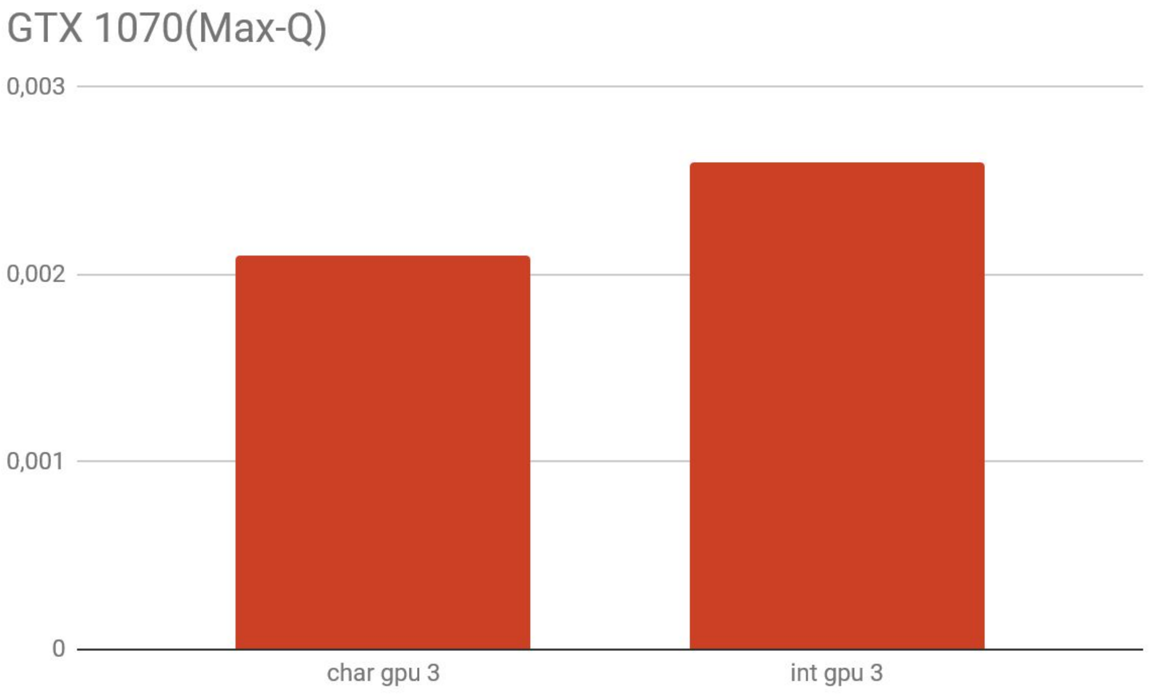

Figure 18. Using 3 chunks 65x65x65 with GTX 1070(Max-Q)

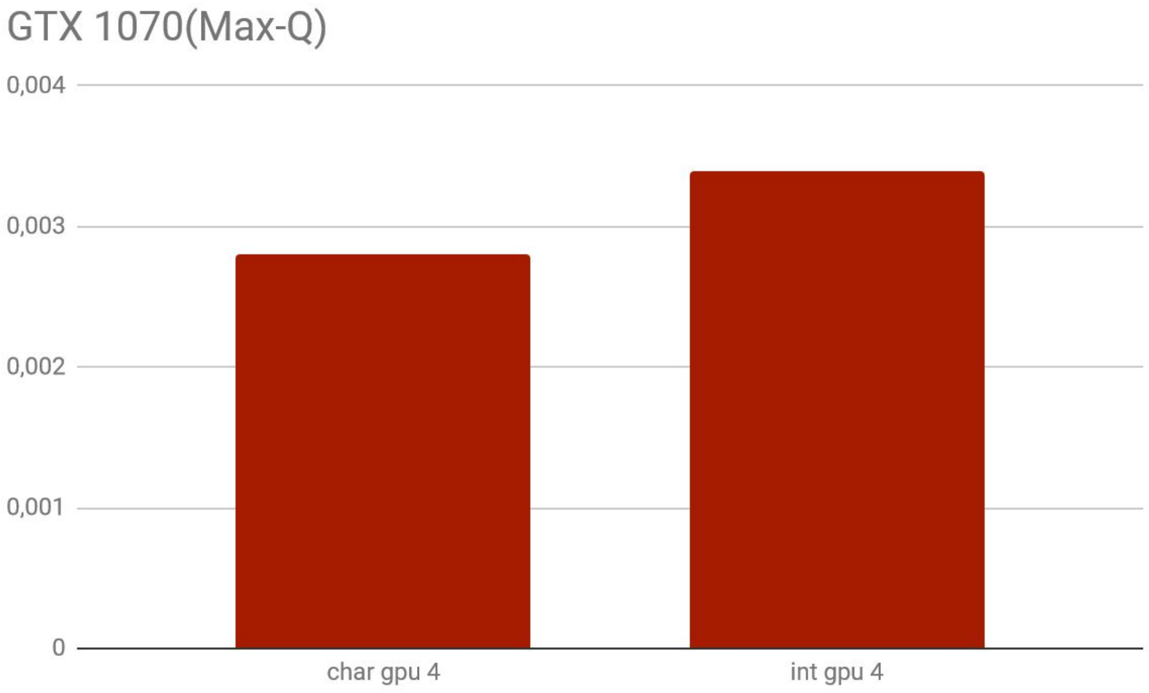

Figure 19. Using 4 chunks 65x65x65 with GTX 1070(Max-Q)

5. Analysis and Discussion

5.1 Game Research Analysis

It was difficult to find good information regarding isosurface algorithms in games but most of the researched titles used Dual Contouring with a few using Marching Cubes. Dual Contouring has the benefits of a lower vertex count, sharp edges and triangle quality. All of these qualities are probable causes for its frequent use. In games Dual Contouring has a wide area of use: terrain, structures and character models were all seen created with it. Marching Cubes has the benefit of being very fast which can be seen in Nvidia's work[2] as well as in the provided GPU implementation. In games, one use-case for Marching Cubes appear to be terrain based. This is possibly because the lack of sharp edges might not matter as much.

For data structures, chunks and octrees both seem to be popular choices. The original Dual Contouring paper[6] provided an octree implementation which could be one cause for its popularity.

Few games allow complete modifiable terrains on very large scale. Subnautica used to have a tool in the beta which allowed users to deform the terrain but it seems to have created more problems than solved as it was later removed citing performance and design reasons[14]. One of the few games that allow terrain deformation on a large scale is No Man's Sky which allows players to destroy very large pieces of terrains by shooting it with guns. The explosions generated by shooting the terrain also creates small fractions of terrain which are removed after a short period of time, possibly for performance reasons.

Voxel Farm's use of scalar fields allows complete worlds/environments to be built. The environments created in Voxel Farm look very similar to ones created with traditional tools while also being highly modifiable in real time with acceptable frame-rates. This is a good showcase of the capabilities of isosurface and scalar field based solutions to terrain creation and modification.

Some games like Everquest Next had issues not necessarily on the performance side of things but with making a fun game as it was abandoned before launch citing the "lack of fun"[18].

5.2 Implementation Analysis

For large scale terrain deformation in real time on the GPU both sets of hardware performed well as can be seen in [Figure 10-14] for the GTX660M and in [Figure 15-19] for the GTX1070. Figures 11-14 and 16-19 are the individual tests for each set of hardware and chunk count. In all of the cases the GTX660M had slightly better performance on the int version and the GTX1070 slightly better performance on the char version.

In the GTX660m configuration the int version had slightly better results than the char version. In the GTX1070 config, the char version had slightly better results than the int version. One possibility for this is that because the GTX660m is less powerful than a GTX1070 and therefore the extra processing power needed to convert chars to ints on the GPU is not worth it. Since the GTX1070 is significantly more powerful and can process data faster and thereful the bottleneck is the time it takes to transfer the data from CPU to GPU which means that the char version is faster. This can be seen in figures above. The implementation had very fast results and could easily be scaled up more and still maintain acceptable real-time frame rates.

Storing the scalar field data as chars on the CPU and bitpacked ints on the GPU was not only an efficient way to store the data in terms of the storage size, but also proved to be the fastest performing on modern hardware. Having all the work done on the GPU prevents the GPU to CPU to GPU passes that significantly slows down the process and proved to give the best results. Piping a lot of data back and forth between the CPU and the GPU takes a lot of time and therefore doing all the work on the GPU every frame instead of sending data back and forth gave the best test results.

As can be seen in[Figure 15] the cost to generate and render any of the GPU versions of the algorithm modern hardware is very small, even in the test with four chunks. The cost of adding more chunks increases linearly with the number of chunks and can therefore easily be scaled up and down to fit the limitations of the software and hardware.

6. Conclusion and Future Work

To answer RQ1, it's certainly possible for games to use detachable terrain at large scales, as games like No Man's Sky and EverQuest Next have shown at different scales. If the need for a a true 3D environment exists, using scalar fields makes a lot of sense. Whether or not parts should be detachable from the environment is more dependent on what type of game or experience the developers is trying to create.

As GPU's continue to improve, some parts of the computational tradeoff could become less of a factor so potentially developers could be more inclined to find further use-cases in games in the future.

Dual Contouring and Marching Cubes are both being used. Dual Contouring has the advantage of sharp edges which can make a big difference in terms of mesh quality/accuracy. In areas where that might not matter as much like terrain, Marching Cubes is a good option. The GPU version of Marching Cubes runs very fast.

A follow-up question for future work could be surrounding whether or not what you gain from using it is sufficiently interesting as compared to what is computationally required. As seen in Subnautica, being able to dig in terrain might not improve the feeling of immersion to an extent that it warrants the increased costs. In that case it could even be argued to hamper the user experience, as referenced by the developer.

To answer RQ2, depending on the hardware storing the data as either plain ints or chars can be preferred. For computation speed, storing as chars and doing bit extraction is preferred on modern hardware while ints should be preferred on less powerful hardware. Generally storing as chars is preferred for their small storage size and also because of the good performance results.

On modern hardware, putting as much work as possible the GPU seems to be a good choice in terms of Marching Cubes. Modern hardware is more limited by memory access than compute power, as can be seen in the results. Maintaining low memory usage opens up potential for larger terrain scalar fields.

The current version updates the data every frame, however a potential performance benefit would be to only update the chunks that changed that frame. This would cost more memory because it would need to store all the result but would only process a single chunk at most every frame.

The smallest possible representation for the implementation used in this paper is a char (1 byte), however by having a different world representation it is possible to store data in even less bits. This could be achieved by having more points in space that would result in less accuracy needed but more data because the world would be more dense. The implementation used in this paper used millimeter based representation with 100 millimeters between every point. Another way to decrease bit use is to represent space less accurately. For example using centimeter accuracy within each cell instead of using millimeter accuracy. This would result in less accuracy but could potentially still be enough depending on the detail level of the geometry being presented. The values needed would then be in the range of -10 to 10 instead which would only need 5 bits instead of 8 bits (1 byte).

References

[1] William E. Lorensen, Harvey E. Cline. 1987. *Marching Cubes: A High Resolution 3D Surface Construction Algorithm*. Published in Siggraph ‘87.

[2] Christopher Dyken, SINTEF Norway and Gernot Ziegler, NVIDIA UK. GPU-accelerated data expansion for the Marching Cubes algorithm. Presented by Nvidia in GPU Technology Conference 2010.

[3] Eric Stephen Lengyel. *Voxel-Based Terrain for Real-Time Virtual Simulations*. Doctoral Thesis. University of California Davis. Published in 2010.

[4] Christopher Dyken, Gernot Ziegler, Christian Theobalt and Hans-Peter Seidel. *High-speed Marching Cubes using Histogram Pyramids*. Published in EUROGRAPHICS 2007.

[5] Sarah F.F.Gibson. *Constrained Elastic Surface Nets: generating smooth surfaces from binary segmented data*. TR99-24 December 1999.

[6] Tao Ju, Frank Losasso, Scott Schaefer, Joe Warren. *Dual Contouring of Hermite Data*. Siggraph 2002.

[7] Scott Schaefer, Joe Warren. *Dual Contouring: “The Secret Sauce”*. Rice University.

[8] Paul Bourke. *Polygonising a scalar field*. http://paulbourke.net/geometry/polygonise/ - Viewable 2019-05-19.

[9] John Congote, Aitor Moreno, Iñigo Barandiaran, Javier Barandiaran, and Oscar Ruiz. *Extending Marching Cubes with Adaptative Methods to obtain more accurate iso-surfaces*. Conference Paper in Communications in Computer and Information Science · January 2010.

[10] Rephael Wenger. *Isosurfaces Geometry, Topology, And Algorithms*. A K Peter. 2013.

[11] Hubert Nguyen. 2007. Nvidia. *GPU Gems 3*. Addison-Wesley Professional (August 12, 2007). Available here: https://developer.nvidia.com/gpugems/GPUGems3 - Viewable 2019-05-19.

[12] Gregory M. Nielson. *Dual Marching Cubes*. IEEE Visualization 2004.

[13] Justin Crause, Andrew Flower, Patrick Morais. *A System for Real-Time Deformable Terrain*. SAICSIT '11, October 3–5, 2011.

[14] Jonas Bötel. *Terrain data format in Subnautica*. https://unknownworlds.com/subnautica/terrain-data-format/ - Forum post, viewable 2019-05-19.

[15] “Flayra”, Subnautica developer. *Forum post detailing why terraformer was removed by developer*. https://forums.unknownworlds.com/discussion/142291/no-more-terraformer-digging - Forum post, viewable 2019-05-19.
[16] Miguel Cepero. Blog by Voxel Farm creator. http://procworld.blogspot.com/ - blog, viewable 2019-05-19.

[17] Nick. Blog by Nick, “ngildea”.
http://ngildea.blogspot.com/2015/12/improving-generation-performance.html - blog, viewable 2019-05-19.

[18] Russell Shanks .
https://www.daybreakgames.com/news/daybreak-president-community-letter-everquest-next-2016 - blog, viewable 2019-05-19.

[19] Mark J.W. Laprairie and Howard J. Hamilton. *Isovox: A Brick-Octree Approach to Indirect Visualization*. Department of Computer Science, University of Regina.

[20] Gernot Ziegler. *Real-time Visual Computing: The HistoPyramid*. 3D Video and Vision-Based Graphics Group. Collection of slides,
http://www.astro.lu.se/compugpu2010/resources/bonus_histopyramid.pdf - viewable 2019-05-19.