A Fast and Simple Solution to Visualising Isosurfaces

H. Perrier[1], J. Dupuy[1, 2], J.-C. Iehl[1], and J.-P. Farrugia[1]

[1]Université de Lyon, CNRS , Université Lyon 1, LIRIS, UMR5205, F-69622, France
[2]LIGUM, Dept. I.R.O., Université de Montréal

Abstract

We present a tool to visualise volumetric data by extracting at each frame a new adaptive triangulated surface from the object. This lets us visualise in real-time static scenes as well as fully dynamic data, as well as keeping the implementation simple since it relies only on the GPU's processing power instead of sophisticated optimisation schemes. We extend an existing adaptive marching cubes solution and increase its execution speed by developing a pipeline that is fully data parallel and implemented on the GPU. We also solve the aliasing issues due to the rasterization, by combining it with a criterion which controls the projected size of the extracted triangles.

1 Introduction

Volumetric objects are being more and more used in computer graphics. They represent a volume with a function $f(\boldsymbol{x}), \mathbb{R}^3 \to \mathbb{R}$, that associates a density to every point in space. This function can be defined analytically or with a dataset. Since this representation does not contain topological information, volumetric objects are easy to manipulate. They are hence a good candidate for procedural generation [Peytavie et al., 2009]. They can be also be found in medical applications, considering they are the output of acquisition scanners. Finally, this representation is widely used in gas and fluids simulation [Bridson and Müller-Fischer, 2007].

One of the most popular solution to visualise those objects in real time is isosurface extraction. Those methods consists in converting the volumetric object into a triangulation that can then be rasterized. The main reason those solutions are widely used is be-

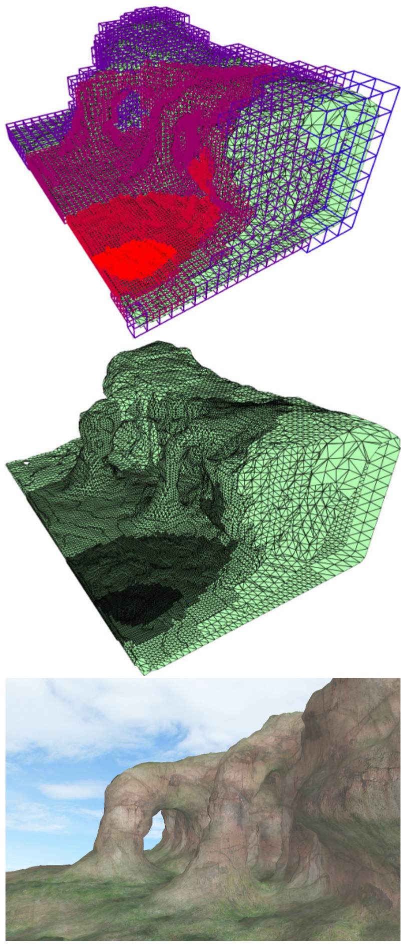
Figure 1: Our method is based on Lengyel's algorithm [Lengyel, 2010] combined with a linear restricted octree (top) which allows us to create adaptive meshes (middle) from volumetric data in real-time (bottom).

cause they fit the GPU pipeline and therefore are very fast.

They however come with two constraints. First, a triangulated surface needs to be extracted from the data. A very popular algorithm to do this triangulation is Marching Cubes (MC) [Lorensen and Cline, 1987]. This algorithms samples the object with a regular square grid. Then, for each cell of this grid it generates triangles along the surface. This is done efficiently by looking up into a triangulation table, associating a list of triangles to every possible corner values of the cell. This algorithm is data parallel, it is therefore well suited for a GPU implementation [Tatarchuk et al., 2007].

Second, during the rasterization process, each pixel of the framebuffer samples a unique triangle. As a result, when several triangles project onto the same pixel, aliasing artefacts appear due to undersampling. To avoid such artefacts, the surface extraction must be adaptive to maintain a good sampling rate, using level of details (LoD) algorithms.

In this paper, we present a method that extracts a triangulated surface from a volumetric object while maintaining a minimal triangle size to avoid the aliasing issues. Our contribution is actually twofold

- First, a method that is both fast enough to visualize those scenes in real time and very simple to implement. We achieved this by extracting a new surface at each frame, relying heavily on the data parallel structure of modern GPUs.
- Second, we addressed the issue of geometric aliasing, that to our knowledge has been left aside until now, by controlling the screen size of the extracted triangles with a data-parallel LoD algorithm.

Our method runs in three pass. The first pass uses a data parallel octree to create a hierarchical partition of the scene. The second pass triangulates the cells of this partition using an adaptive marching cubes. This leads to cracks when two different level of details meet. Those are fixed by "stitching" them in a last pass, filling them with additional triangles.

We will present the state of the art of visualising volumetric object in real time with level of detail in Section 2. Then, we will develop our implementation and the algorithms it is based on in Section 3. Finally, in Sections 4 and 5, we will show our results and then conclude about our work.

2 Related Work

In this section, we will focus on the methods that use isosurface extraction to visualize volumetric objects in real time. We will not talk about the solutions that extract those triangulations as an offline process, nor will we talk about the whole Marching Cube like family of methods [Schaefer and Warren, 2004, Shu et al., 1995, Schmitz et al., 2009] that is far too wide to be correctly covered in this paper. We also omit all solutions that are based on ray casting [Crassin et al., 2009, Dick et al., 2009], or hybrid methods [Gobbetti and Marton, 2005, Ammann et al., 2010].

Real time adaptive surface extraction has been an extensively studied topic. A numerous amount of solutions were presented to extract an adaptive surface from 2.5D datasets [Lindstrom et al., 1996, Bösch et al., 2009, Dupuy et al., 2014]. One of the seminal paper in this domain is the paper of Duchaineau et al. [Duchaineau et al., 1997]. By using two priority queues, they maintain an optimal triangulation given a fixed triangle count and a fixed fps. A few years later, Losasso and Hoppe [Losasso and Hoppe, 2004] presented Geometry clipmaps, a solution so efficient that is still competes with actual state of the art method thanks to a very efficient texture caching scheme on the GPU. But all those solutions only renders heightmaps dataset.

Another numerous amount of methods were developed for 3D datasets [Westermann et al., 1999, Scholz et al., 2014, Löffler et al., 2011]. We can cite Zhou et al. [Zhou et al., 2011] that uses data parallel octrees to extract a triangulation from point clouds. However their solution is very difficult to implement. Lengyel [Lengyel, 2010] presented a solution that also uses an octree to divide the scene before triangulating it with a marching cubes and stitching the cracks with a second marching cube. Our method iterates on this work. In the original paper, the octree is implemented on the CPU. Here, we replaced it by a data parallel structure that fits the GPU pipeline. Scholz et al [Scholz et al., 2014] also presented a solution that is very close from our work. They rely on a tetrahedral structure instead of an octree to prevent cracks from happening which lets them use a MC process to extract the surface, and to avoid a heavy stitching process. However, this method relies on caching and preprocess, precomputing the triangulations and saving them for reuse. Even though this leads to very high framerates, it prevents the use of dynamic data rendering in real time. Another issue is the difficulty of implementation of their work. Here, we chose to focus on a method that is both fast and simple.

An interesting point was presented by Bosch et al. [Bösch et al., 2009]. They mentioned how the extraction of adaptive dataset can lead to aliasing issues if it is not performed carefully. However, none of the actual solutions really investigated this issue. For most methods, the LoD criteria is based on a geometric error or on a distance to the viewer, and never ensures a minimal triangle size to prevent rasterization issues. Some solutions such as [Löffler et al., 2011, Fang et al., 2003] used a level of detail criterion based on the projected size of triangles but none of them uses this criterion to ensure a minimal triangle size. To our knowledge, the only solution that used a similar criterion for triangle size control was the one of Dupuy et al. [Dupuy et al., 2014]. However, this solution is limited to 2.5D datasets.

3 Background

The solution of Lengyel *et al.* [Lengyel, 2010] is based on extracting an adaptive triangulated surface by applying MC on an octree instead of a regular grid. However, due to its recursive nature, the octree is not suited for the GPU pipeline. Hence, to take full advantage of the GPU, we must convert this tree to a non recursive structure. Such representations were first introduced by Gargantini [Gargantini, 1982] as linear quadtrees. They have been implemented on the GPU by Dupuy *et al.* [Dupuy et al., 2014].

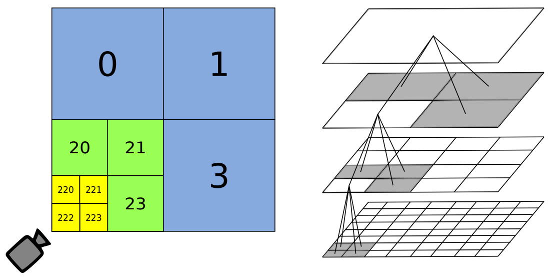

Figure 2: In [Dupuy et al., 2014], a linear quadtree is used to maintain a 2D hierarchical partitioning of space. (Left) presents such a partitioning. A cell is more or less subdivided depending on its distance to the camera. They are also encoded with morton code, that identifies each cell with its position relatively to its parent. (Right) shows the corresponding active front in the full tree. This partition is entirely represented by storing the list of codes of the gray cells.

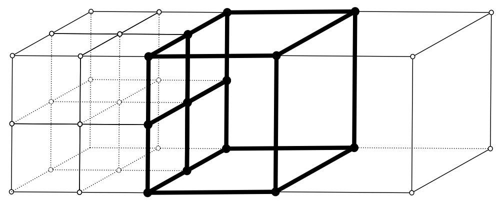
Figure 3: Triangulating a hierarchical grid with MC creates cracks at level changes. To fix those, Lengyel *et al.* inserts transition cells (bold). Those cells are more subdivided on one side to patch the change of resolution. Triangulating those cells finally generates the triangles that will patch the cracks.

3.1 Adaptive surface extraction

Lengyel *et al.* [Lengyel, 2010] developed a solution to visualise volumetric data in real-time. They start by subdividing the space with an octree. At each frame, they select a set of cells within this tree, called the active front, that partitions the space. The resolution of the partition varies along the space, depending on the chosen cells. This idea is illustrated for quadtrees in the Figure 2. Whether a cell is part of this active front is determined by evaluating a LoD criterion. The selected cells can finally be triangulated using a method similar to MC.

When two cells of different resolution meet, T-Junctions appear. They are due to some vertices in the high resolution cells that have no match in the low resolution cell. The problem is that when using a MC algorithm, those inconsistencies create cracks in the final triangulation. To fix this, Lengyel *et al.* inserts transition cells (Figure 3) where two cells of different level meet. Those cells are more subdivided on one side than on the other to patch the difference of resolution. They are then triangulated with a modified MC, producing a crack-free surface for visualisation. However, it requires having triangulation tables adapted to the unusual structure of the transitions. It also demands the use of a restricted octree, since the transitions only allow a single level of difference between neighbouring cells. As the triangulation is based on modified MC, it is data parallel.

The adaptivity of the final triangulation is controlled by a LoD criterion. Lengyel *et al.* use a measure based on the distance between the cell and the camera, which must therefore be re-evaluated every time the camera moves. We will use a similar criterion here, so we will have to change the octree at every frame.

On the GPU, always recomputing the whole active front from the root node would be too expensive. Instead, we update the tree by merging or splitting the cells, relying on temporal coherence. Since we want to extract an adaptive triangulation entirely on the GPU, we require a non recursive representation for the tree, as well as data parallel update operations. Those properties can be found using linear trees [Gargantini, 1982].

3.2 Linear Quadtrees

Linear quadtrees were presented by Gargantini [Gargantini, 1982] as a non recursive representation for quadtrees. Such structures were reused by Dupuy *et al.* [Dupuy et al., 2014] who presented a solution to handle them efficiently on the GPU.

Linear quadtrees associate a code to each cell, identifying it by its path to the root node. The full tree is thus entirely represented by the list of its leaves' codes, which removes the recursivity. In their implementation, Dupuy *et al.* used Morton codes (Figure 2). Those codes store a spatial position, encoding the location of the cell relatively to its parent. The final code of a cell is thus the succession of locations between the cell and the root node. Therefore, the maximal depth of the tree depends on the number of quadrants that can be encoded. In a quadtree, a cell is divided into 2^2 quadrants, so encoding one subdivision requires two bits. Dupuy *et al.* store for each cell its Morton code and its depth within the tree. Therefore, with a code stored on 32 bits, using 28 bits for the locations and 4 bits for the depth, the tree is limited to 14 levels.

Dupuy *et al.* used those quadtrees to maintain a 2D hierarchical space partition. This partition was updated at every frame.

During an update, there are three possible operations on a cell: it can be kept, merged or split. A merge operation removes the codes of the children's cells from the list and replaces them with the code of the parent cell. A split operation removes the code of a cell replacing it by the codes of its children. A kept operation maintains the code of the current cell. An efficient GPU implementation relies on the fact that those operations can be done independently on each cell. This is true as long as the LoD criterion is also evaluable independently on each cell.

With this implementation of linear tree, it is possible to maintain a quadtree on the GPU. By generalizing this to the octree, it could be used with the triangulation presented by Lengyel *et al.* to develop a fully data parallel pipeline. There is however three conditions on the LoD criterion.

- It must ensure that the extracted triangles will project onto more than one pixel, to ensure a good rasterization sampling.
- The tree must also stay restricted at all times as the transition cells of Lengyel *et al.* only patch cells with a single change in resolution.
- Finally, this criterion must allow to detect the position of the transition cells. Those cells are inserted when a neighbouring cell is more subdivided than the current cell. We must thus be able to evaluate this criterion on the neighbours of a cell. However, to do this on the GPU, we must do this with a data parallel algorithm.

3.3 LoD on projected size

When rasterizing a triangulated surface, all triangles are projected on the screen. Then, each pixel selects a sample from one or several of the triangles that projects onto itself and combines those samples to compute the final color. However, since it is expensive to take many samples, a fixed number of samples is used for each pixel. But is there are more triangles projecting onto the pixel that the number of samples taken, we get aliasing artefacts due to undersampling. Therefore, to limit this, we aim at ensuring a minimal triangle projected size. If all triangles project onto more than a pixel, a single sample is sufficient to determine the final color.

When using a MC algorithm, the size of the generated triangles is related to the size of the cell they are extracted from. Therefore, we will first ensure that the cells all project onto more than one pixel. To do so, we use the criterion in Equation 1 where, $2 \cdot s(z)$ is the projected screen size at a distance z from the camera, and $2 \cdot \alpha$ is the horizontal viewing angle (the fovy).

$$s(z) = z \cdot \tan(\alpha) \tag{1}$$

Then, as in Dupuy et al. [Dupuy et al., 2014], to determine if a cell should be merged/split, the ratio

$$\frac{\text{cell\_size}}{2 \cdot s(z)}$$

is evaluated. This ratio is compared to a factor k that controls the density of the triangulation.

$$\text{If} \quad \frac{\text{cell\_size}}{2 \cdot s(z)} \geq k \quad \text{then split} \tag{2}$$

$$\text{Else if} \quad \frac{\text{parent\_cell\_size}}{2 \cdot s(z_{\text{parent}})} \leq k \quad \text{then merge} \tag{3}$$

$$\text{Else keep}$$

Evaluating (2) splits a cell if its project size exceeds a certain amount. Equation (3) determines if a cell should merge by testing if the parent of this cell should split. This guarantees that all the children of a cell will merge at the same time, since they will all evaluate the criterion on the same parent. This criterion also maintains a restricted quadtree while the condition

$$\frac{\text{cell\_size}}{2 \cdot s(z)} < 2 \cdot k$$

is always verified. For $k = 2$, this is ensured as long as the horizontal viewing angle doesn't exceed 142^o. We prove this in Appendix 1.

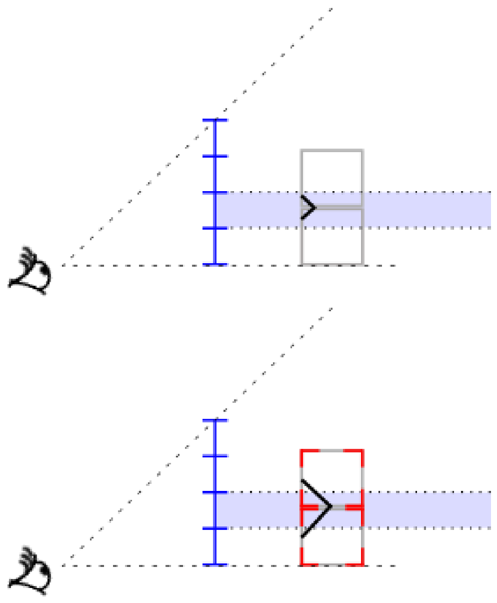

Figure 4: On the top image, the cells (in gray) both project onto more than a pixel (identified by the light blue zone, the image plane is in dark blue). However, since the triangles are too small inside the cell, they both project onto the same pixel, leading to aliasing issues. On the bottom figure, we limited the placement for the vertices (the red zones are forbidden to place a vertex). Therefore, the size of the cell can guarantee a minimal triangle size and if the cell do project onto more than a pixel, no aliasing is possible.

However, it is worth noting that when used with a MC process, this kind of LoD in itself is not sufficient to guarantee a minimal projected triangle size. It only guarantees a minimal cell projected size, and a cell only guarantees a maximal triangle size. When Marching cubes generates the triangulation, it determines for each edge of the cell if it is crossed by the surface. Then, it decides where to put a vertex on this edge and finally it connects all vertices. Therefore, if the vertices can be put anywhere on the edge, we might get a triangle whose vertices are so close from the corner of the cell that it will project on less than a pixel.

To guarantee a minimal triangle size, it is therefore mandatory to constraint the vertex position. For example, forbidding to set a vertex in the first or last 1/10th of the edge. Then, adapting the k factor to ensure that 1/10th of the cell will always project onto more than a pixel guarantees that all triangles will project onto more than a pixel. This is illustrated Figure 4. Of course, with such a process, some geometrical precision is lost. This will be discussed in the last section.

Despite this issue, this LoD criterion has the four characteristics that we need:

- It is data parallel.
- It can ensure that the cells project onto more than one pixel.
- It maintains a restricted tree.
- Its evaluation returns if a cell should split or merge.

Therefore, by generalizing it to the octree, we could use it with Lengyel *et al.* triangulation method to implement an adaptive visualisation pipeline on the GPU.

4 GPU implementation

We present a new tool for isosurfaces visualisation that is adapted to rasterization issues and runs in real-time. Our pipeline is divided in 3 parts.

- The first step consists in updating a linear octree that maintains the active front of cells.
- The second step performs both a culling on the octree cells, and creates the transition cells required for a crack-free MC triangulation.
- The last step takes the cells that passed the culling, the transition cells, and re-subdivide them before triangulating them.

This whole process is illustrated in Figure 5. In the following sections, we present each step of our solution.

4.1 Maintaining the linear octree

Our method triangulates a set of cells. Therefore, the first step of our pipeline evaluates a LoD criterion on each cell to determine if it is kept, split or merged. In this section, we present the GPU representation for those trees, as well as the criterion we use to evaluate the new partition.

Linear octree on the GPU

Dupuy *et al.* [Dupuy et al., 2014] proposed an implementation to handle efficiently quadtrees on the GPU. Here, we want to manipulate an octree, hence, we generalize their solution to the third dimension.

To represent the cells, we use Morton codes. Those codes are a succession of locations relatively to the root node. In three dimensions, there are $2^3 = 8$ possible locations for a node so 3 bits are required to encode one subdivision. Thus, if we keep the codes on 32 bits as in [Dupuy et al., 2014], we will be limited to a maximal subdivision depth of 9. This is too coarse to do a correct triangulation of the surface. Therefore, we used 64 bits for the encoding. This doubles the memory size of the front but, by using 5 bits for the depth of the cell and the other 59 bits for the code, we can subdivide 19 times.

At each frame we have the active front of the previous frame, t, and we want to update it to get the front at frame $t + 1$. To do so, we evaluate the LoD criterion given in equation (1) on every cell at frame t. We then test the equations (2) and (3) to determine if the cell should split or merge. If a cell splits, it emits the codes of its children. If a cell is kept it re-emits its own code. If a cell merges, there are two possibilities. If it is the first child (with the lowest Morton code), it emits the code of its parent. Otherwise, it does nothing. This ensures that the merging operation produce only one cell. The new cells are written into a new buffer that represents the active front at frame $t + 1$. This is illustrated Figure 6.

Improved LoD criterion

Dupuy *et al.* manipulated quadtrees, a two dimensional structure, which contained a reasonable amount of cells in their active front. This amount is greater when using an octree as those structures are in three dimensions. We thus extended the LoD

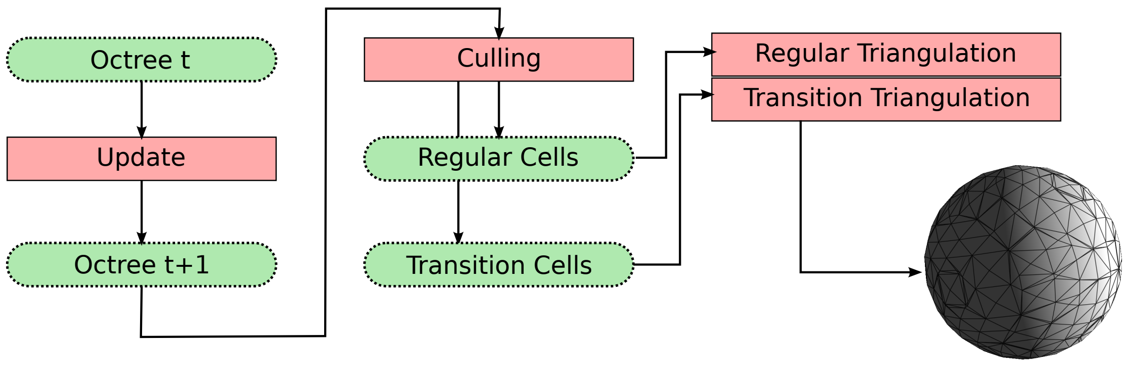

Figure 5: This figure summarizes our GPU pipeline. Data buffers are represented in green and computations in red. Each computation retrieves data from a buffer and fills a new one.

criterion determining the partition to minimize the size of the front. Note that our extended criterion could still be used in two dimensions.

Here, we use the criterion given equation (1). Its evaluation is data parallel and controls the projected size of the cells. However, it is based on the Euclidean distance between a cell and the camera. Since this distance is absolute, cells behind the camera are highly subdivided even though they do not contribute to the final image. Therefore, we combine this Euclidean distance with a radial angle to the camera's forward direction. Cells behind the camera are then less subdivided, which reduces the number of cells in the octree. The new criterion is given Equation (4) and illustrated Figure 7.

$$z' = z + \begin{cases} w \cdot z \cdot (\theta - \alpha) & \text{if } \theta \geq \alpha \\ 0 & \text{else.} \end{cases} \tag{4}$$

In this equation, z is the Euclidean distance between the center of the cell and the camera, θ is the angle and α is the fovy, both expressed in radians. The parameter w controls the importance of the radial distance over the Euclidean distance. With this new z', the criterion evaluated is now $s(z')$ instead of $s(z)$. Note that we may lose the guarantee of maintaining a restricted octree outside the frustum. This should not create visual artefacts as those cells do not contribute to the final image. Empirically, we found that using $w \leq 6$ prevents artefacts.

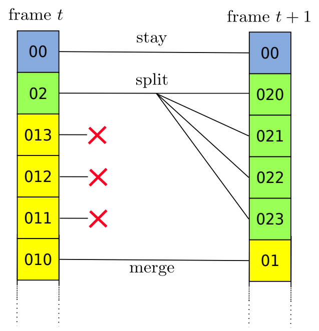

Figure 6: The octree at frame t is stored as a list of Morton codes in a buffer. To update the tree, we fill a new buffer representing its state at frame $t+1$. On the GPU, each code of the first buffer is processed in parallel. If the cell is to be subdivided, it writes the codes of its children in the new buffer. If it merges, it writes either nothing, or the code of its parent (if it is the first child). If it is kept, it just re-writes its own code.

4.2 Culling and transitions determination

Once the active front has been updated, it contains the list of the cells to triangulate. We then apply frustum culling to remove the nodes that are outside the view frustum. We also remove empty cells; a cell that is fully inside or outside the object will not generate triangles and therefore should not be triangulated. The remaining cells are stored in a new buffer that is used as input for triangulation.

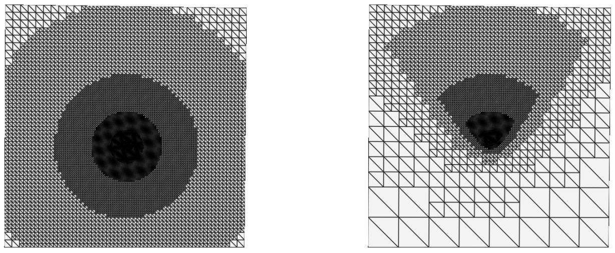

Figure 7: Adaptive triangulation of a plane. The camera is located in the center and looks up. (Left) uses the criterion in [Dupuy et al., 2014] and (Right) uses our criterion. The total number of cells is (Left) 1037114 and (Right) 210323.

Once the culling is done, we must then identify the level changes in the octree to create the corresponding transitions cells. We do so by checking if a cell has a neighbour of higher resolution than itself. In our implementation, the chosen LoD criterion is fully data parallel and allows to determine if a cell has been split or merged. Therefore, we can determine the transitions by evaluating this criterion on the neighbours of a cell.

Our LoD criterion is based on the projected size of a cell. We thus need the position of the neighbours of the current cell. This is easy to compute as we know the position of the cell and its size. We then test each of them to determine if it has been split, using equation (2). If this test is verified, we emit a transition cell. Note that each cell can then emit a maximum of 3 transitions, one for each axis. We finally have two buffers containing the octree and transition cells.

4.3 Triangulation and further subdivision

When a deeper octree is used to subdivide the scene, the resulting triangulation is more precise. However, since it contains more cells, updating this tree and culling the cells become computationally expensive. To compensate this tradeoff, we added a subdivision pass after the culling and before the triangulation. This allows us to get a more precise triangulation with a coarser octree.

Every cell of the octree and every transition cell goes through this subdivision. Each octree cell is turned into a regular grid of smaller cells with a resolution of $(2^N)^3$, N being a positive integer controlling the new level of subdivision. The transition cells have a fixed width so they become a regular grid with a resolution of $(2^N)^2$.Since this subdivision is done after the update and culling, those steps stay very efficient and the triangulation itself is the only step that runs on the subdivided tree. However, this implies that no culling is done on the new smaller cells.

The triangulation step in itself is a classical MC algorithm, using the triangulation tables given by Lengyel et al. [Lengyel, 2010].

5 Results

We tested our method with an Intel Pentium 3550M processor and a NVIDIA GeForce GTX 850M GPU. We implemented it with C++ and GLSL. The pipeline runs with geometry shaders, combined with transform feedbacks.

We ran our method on different volumetric datasets. The first one is a stack-based representation of a terrain, called Moria, created with the ARCHES framework [Peytavie et al., 2009]. It contains 512^3 voxels, which we uploaded on the GPU using a 3D texture. In a second test, we visualised a set of metaballs. Those implicit surfaces are defined analytically to be evaluated on the GPU and are updated at every frame. This scene creates objects that goes through high geometrical changes and thus cannot be precomputed. All those scenes are illustrated in Figure 9 and in the accompanying video. In all our experiments, we used the LoD criterion given in equation (4).

5.1 Stack-based terrain

We ran our method on the Moria scene [Peytavie et al., 2009], enhanced with a noise function to add high resolution geometric details. We present the average execution times for each step of the pipeline, as well as the number of cells in input (Table 1).

Step	Time	#Cells
Update	1.13	219295
Culling	3.3	219295
Triangulation	0.94	5570
GPU time	5.4	
CPU Time	2.3	

Table 1: Average execution time for each step of the pipeline during a flythrough of the Moria scene. The first column gives the average execution time in ms and the second contains the number of cells in input.

We can note that updating the octree is very fast (less than 2 ms) whereas the culling step is slower (by roughly 300%). This is due to the fact that this pass runs tests on the whole partition, which can be computationally expensive. Nonetheless, this efficient culling allows the triangulation to run on a very small amount of cells, thus reaching very good performances. Finally, the total GPU time needed to update and triangulate the new frame is under 6 ms (180 FPS). Furthermore, as every computation is done on the GPU, the CPU is only active for 2 ms to initiate the shaders.

5.2 Animated metaballs

The cells inside the frustum are re-triangulated at each frame. Therefore, it is possible to visualise fully dynamic data without any performance loss. We tested this while visualising animated metaballs. When running our experiments, the animation was stopped between the frames 5000 and 10000. The camera was static during the whole experiment to remove all variability that would have been induced if the number of cells to evaluate was changing. The resulting graph is shown in Figure 8.

As can be seen on this graph, the triangulation timings are constant. This is explained by the fact that we do not rely on saving and reusing the computed trangulation. Furthermore, since our LoD criteria is only based on the triangles projected size, the number of cells in the octree is not affected by the geometry but only by the camera's position. The only variability that can be measured is induced by the task scheduling on the GPU.

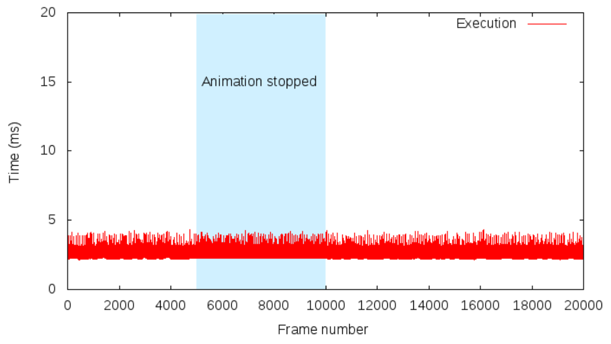

Figure 8: Rendering time for an animated scene. The average rendering time is about 3 ms even though the object is re-triangulated on each frame. Those timings do not include the shading. The blue box identifies the frames during which the object was static. As can be seen, the fact that the object is or not animated has no impact whatsoever on the rendering time.

6 Conclusion

We presented a new tool for real-time visualisation of volumetric data. Our method runs entirely and efficiently on the GPU. We also triangulate the surface on the fly and therefore are able to visualize dynamic scenes such as moving metaballs. Furthermore, with a LoD criterion that is based on the cells projected size, we prevent geometrical aliasing artefacts during rasterization. To top it all, this method is very simple to implement, which makes it a valuable candidate for the industry.

It however has several flaws compared to state of the art solutions. The major one is that popping artefacts occur at level change. Therefore, the main future work on this method would be to add a geomorphing process [Giegl and Wimmer, 2007, Heidrich et al., 1998, Lindstrom et al., 1996]. Ideally, this process should keep the simplicity of the method and would allow for an efficient GPU implementation. Another important flaw is the complete absence of geometric error measurement. Most methods determine if a triangulation must be subdivided or not depending on how close it is from the real geometry of the object. Here, a major constraint is that the LoD criteria must be evaluated in a data parallel process while also ensuring a restricted tree; this means that we cannot access the informations

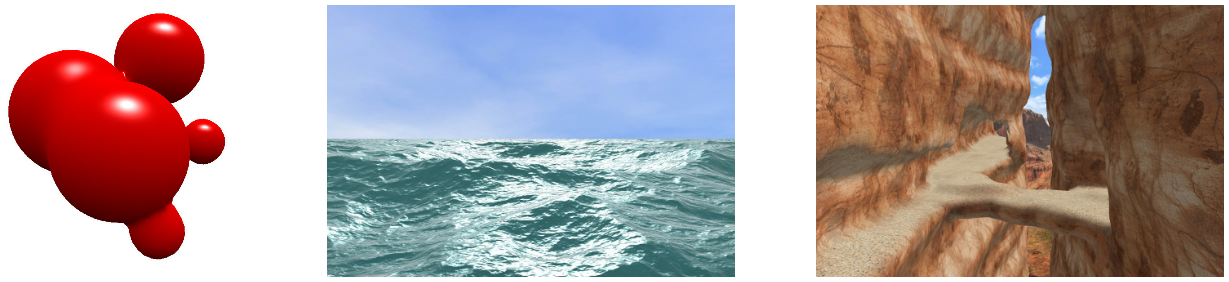
Figure 9: This shows some outputs of our program. (Left) is an example of moving metaballs. (Middle) is an animated ocean. (Right) is a stack based terrain from the ARCHES plateform [Peytavie et al., 2009]

of a cell's neighbours, but we still need to ensure that we will not have more than one level of difference. We also do not want to add any preprocess on the method to maintain its ability to render dynamic datasets. And finally, we want a criteria that fulfill those constraints while maintaining the antialiasing property. Last drawback, our method only generates a triangle soup, instead of a well defined mesh. However, it may be fast enough to add a retriangulation process after the extraction, similarly to what is done in [Scholz et al., 2014].

Our choice to retriangulate the object at every frame could also be arguable. This makes our method slower than methods based on saving and reusing parts of the triangulation. However, this kind of caching has a memory cost that is not negligible. Furthermore, it can be a real issue when visualising dynamic datasets.

References

[Ammann et al., 2010] Ammann, L., Génevaux, O., and Dischler, J.-M. (2010). Hybrid rendering of dynamic heightfields using ray-casting and mesh rasterization. In *Proceedings of Graphics Interface 2010*, pages 161–168. Canadian Information Processing Society.

[Bösch et al., 2009] Bösch, J., Goswami, P., and Pajarola, R. (2009). Raster: Simple and efficient terrain rendering on the gpu. *Eurographics Areas Papers*, pages 35–42.

[Bridson and Müller-Fischer, 2007] Bridson, R. and Müller-Fischer, M. (2007). *Fluid simulation: SIGGRAPH 2007 course notes.* ACM.

[Crassin et al., 2009] Crassin, C., Neyret, F., Lefebvre, S., and Eisemann, E. (2009). Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, Boston, MA, Etats-Unis. ACM, ACM Press. to appear.

[Dick et al., 2009] Dick, C., Krüger, J., and Westermann, R. (2009). Gpu ray-casting for scalable terrain rendering. In *Proceedings of EUROGRAPHICS*, volume 50. Citeseer.

[Duchaineau et al., 1997] Duchaineau, M., Wolinsky, M., Sigeti, D. E., Miller, M. C., Aldrich, C., and Mineev-Weinstein, M. B. (1997). Roaming terrain: real-time optimally adapting meshes. In *Proceedings of the 8th Conference on Visualization'97*, pages 81–88. IEEE Computer Society Press.

[Dupuy et al., 2014] Dupuy, J., Iehl, J.-C., and Poulin, P. (2014). *GPU Pro 5*, chapter Quadtrees on the GPU.

[Fang et al., 2003] Fang, D. C., Gray, J. T., Hamann, B., and Joy, K. I. (2003). Real-time view-dependent extraction of isosurfaces from adaptively refined octrees and tetrahedral meshes. In *Electronic Imaging 2003*, pages 103–114. International Society for Optics and Photonics.

[Gargantini, 1982] Gargantini, I. (1982). An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905–910.

[Giegl and Wimmer, 2007] Giegl, M. and Wimmer, M. (2007). Unpopping: Solving the image-space blend problem for smooth discrete lod transitions.

In *Computer Graphics Forum*, volume 26, pages 46–49. Wiley Online Library.

[Gobbetti and Marton, 2005] Gobbetti, E. and Marton, F. (2005). Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. In *ACM Transactions on Graphics (TOG)*, volume 24, pages 878–885. ACM.

[Heidrich et al., 1998] Heidrich, W., Slusallek, P., and Seidel, H.-P. (1998). Real-time generation of continuous levels of detail for height fields. In *Proc. WSCG98*, pages 315–322.

[Lengyel, 2010] Lengyel, E. S. (2010). *Voxel-based terrain for real-time virtual simulations*. University of California at Davis.

[Lindstrom et al., 1996] Lindstrom, P., Koller, D., Ribarsky, W., Hodges, L. F., Faust, N., and Turner, G. A. (1996). Real-time, continuous level of detail rendering of height fields. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 109–118. ACM.

[Löffler et al., 2011] Löffler, F., Müller, A., and Schumann, H. (2011). Real-time rendering of stack-based terrains. In *VMV*, pages 161–168.

[Lorensen and Cline, 1987] Lorensen, W. E. and Cline, H. E. (1987). Marching cubes: A high resolution 3d surface construction algorithm. In *ACM Siggraph Computer Graphics*, volume 21, pages 163–169. ACM.

[Losasso and Hoppe, 2004] Losasso, F. and Hoppe, H. (2004). Geometry clipmaps: terrain rendering using nested regular grids. *ACM Transactions on Graphics (TOG)*, 23(3):769–776.

[Peytavie et al., 2009] Peytavie, A., Galin, E., Grosjean, J., and Merillou, S. (2009). Arches: a framework for modeling complex terrains. In *Computer Graphics Forum*, volume 28, pages 457–467. Wiley Online Library.

[Schaefer and Warren, 2004] Schaefer, S. and Warren, J. (2004). Dual marching cubes: Primal contouring of dual grids. In *Computer Graphics and Applications, 2004. PG 2004. Proceedings. 12th Pacific Conference on*, pages 70–76. IEEE.

[Schmitz et al., 2009] Schmitz, L., Dietrich, C., Comba, J. L., et al. (2009). Efficient and high quality contouring of isosurfaces on uniform grids. In *Computer Graphics and Image Processing (SIBGRAPI), 2009 XXII Brazilian Symposium on*, pages 64–71. IEEE.

[Scholz et al., 2014] Scholz, M., Bender, J., and Dachsbacher, C. (2014). Real-time isosurface extraction with view-dependent level of detail and applications. *Computer Graphics Forum*, pages 1–13.

[Shu et al., 1995] Shu, R., Zhou, C., and Kankanhalli, M. S. (1995). Adaptive marching cubes. *The Visual Computer*, 11(4):202–217.

[Tatarchuk et al., 2007] Tatarchuk, N., Shopf, J., and DeCoro, C. (2007). Real-time isosurface extraction using the gpu programmable geometry pipeline. In *ACM SIGGRAPH 2007 courses*, pages 122–137. ACM.

[Westermann et al., 1999] Westermann, R., Kobbelt, L., and Ertl, T. (1999). Real-time exploration of regular volume data by adaptive reconstruction of isosurfaces. *The Visual Computer*, 15(2):100–111.

[Zhou et al., 2011] Zhou, K., Gong, M., Huang, X., and Guo, B. (2011). Data-parallel octrees for surface reconstruction. *Visualization and Computer Graphics, IEEE Transactions on*, 17(5):669–681.

A Ensuring a restricted octree

To partition the space, Dupuy *et al.* use the criterion given in equation (1). They then test the size of the cell over this criterion. If equation (5) is verified, it means that the cell will be twice more subdivided than its neighbour. To maintain a restricted tree, we need to ensure that it never happens.

$$\frac{\text{cell\_size}}{2 \cdot s(z)} < 2k \tag{5}$$

We illustrate this limit case Figure 10. Because the only variable is α, we need to bound it. From Thales, we can deduce:

$$\frac{2k \cdot s(z)}{s(z)} = \frac{z + s(z)}{z}$$

From this, we can express z in terms of $s(z)$:

$$z = \frac{s(z)}{2k - 1}$$

Finally, trigonometry gives us:

$$tan(\alpha) = 2k - 1$$

As α is half the horizontal viewing angle, the octree is constraint as long as $\alpha < 2\arctan(2k-1)$ For $k = 2$, this implies that the fovy must not exceed 142^o.

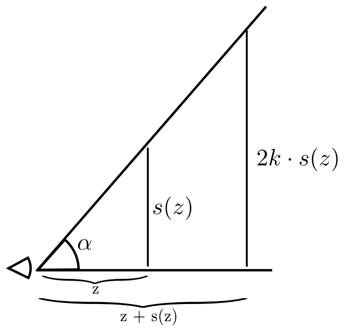

Figure 10: If this situation happens, the LoD criterion no longer ensures to maintain a restricted tree. The parameter α is thus bounded. Over a certain value, a cell can have two level of differences with its neighbours. This figure is willingly wrongly scaled for readability purposes.