UNIVERSITY OF HAMBURG
Department of Informatics

# Animated Sparse Voxel Octrees

**Dennis Bautembach**

Bachelor's Thesis
February 25, 2011

Supervisors: Dr. Werner Hansmann (Betreuer),
Prof. Dr. Leonie Dreschler-Fischer (Zweitbetreuerin)

**Abstract**

In this thesis a general way of animating sparse voxel octrees during the rendering process is presented. It is shown how one can apply it to rotation and skinning. Furthermore, a rasterizer is introduced that yields real-time frame rates rendering a model consisting of 3M voxels with phong shading, shadow mapping and skinning on a 800 x 800 screen resolution. The rasterizer makes it possible to render mixed graphical content consisting of both triangles and voxels that interact with each other (e.g. cast shadows on one another).

# Contents

*Figure 1: An example of a (low resolution) voxel model.*

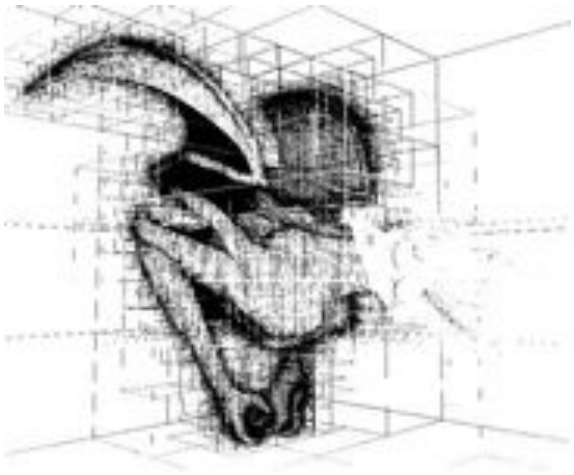

*Figure 2: The concept of SVO construction illustrated.*

# 1 Introduction

There are plenty of ways to represent three dimensional (3D) models, such as triangle meshes, voxels, CSG (constructive solid geometry), etc. Triangle meshes are the most common format among the others due to their use in computer and video games and hardware support through GPUs. Voxels have many advantages over triangle meshes. Such advantages are memory efficiency, implicit level of detail and resolution boundedness (rather than geometry boundedness). They are not being used in mainstream real-time graphics applications, mainly due to the fact that they are static. It turns out to be quite impossible to animate them.

## 1.1 What voxels are

Voxels are cuboids. The idea is to approximate 3D models with many (several million) little cuboids, as can be seen in figures 1 and 12. If the resolution of a voxel model is considerably high (and one does not zoom in too much), its voxels become smaller than the pixels of the display and are not distinguishable anymore. The model is perceived smooth rather than blockish. Image enhancing techniques like anti-aliasing, supersampling and interpolation finally completely hide the underlying cuboid structure of a voxel model – the same way they hide the mesh structure of triangle based models.

### 1.1.1 What voxels are suited for

Voxels are suited for representing models with great mesostructural[1] detail, such as rocks, mountains, terrain, plants, vegetation, nature. Especially if those models have surface

[1]Object geometry is usually defined on three scales, the macrostructure level, the mesostructure level, and the microstructure level. The mesostructure level includes higher frequency geometric details that are relatively small but still visible such as bumps on a surface[8].

characteristics like hangovers which cannot be faked by techniques like parallax mapping[2].

Under these conditions it can be stated that voxels are more memory efficient than triangles, due to the fact that as many triangles as voxels are needed to accurately represent such surfaces and a simple voxel implementation uses 1.25 bytes positional data per voxel whereas a triangle typically consumes 14 bytes positional data, and two triangles are needed to simulate one voxel (to form a rectangle that fills one pixel).

### 1.1.2 What voxels are inappropriate for

Voxels are inappropriate for representing flat surfaces, which can be found in architecture and furniture for example. Such surfaces can be modeled with very few triangles, whereas voxel models must always have maximal resolution to appear smooth – no matter the kind of shape they represent.

This is the main reason why one should not stick to one technology exclusively but mix both, tapping their full potential.

## 1.2 Storage format

Probably, the simplest storage format for voxel models is a uniform grid. It can be realized as a 3D array wherein every single cell describes one voxel (its appearance). This data structure is very memory inefficient. In order to only store the shape of a 3D model with a resolution of $1024^3$ voxels, 1 GB of memory is needed – and that would not even be remotely detailed.

A more efficient solution and the most commonly used one is the sparse voxel octree (SVO). It is a tree data structure in which every node can have **up to** eight children. It is sparse in the sense that it does not include the subtrees corresponding to empty space. The major idea is to start with one large cuboid and evenly split it up into eight smaller cuboids and repeat this process for every child recursively until the leaves sufficiently approximate the shape of an object. Figure 2 illustrates this concept.

SVOs are very memory efficient (state of the art voxel rendering engines can compress the memory footprint of one voxel down to one bit[1]) and their hierarchical nature benefits many operations used in computer graphics.

## 1.3 Advantages of sparse voxel octrees over triangle meshes

Even though the octree structure adds a little bit of hierarchical overhead to all computations done on SVOs, the resulting benefits outweigh it by far.

- Implicit level of detail (LOD): Traversal needs not to continue when voxels reach the size of a pixel. Thus, only as many voxels are rendered as the model occupies pixels and SVO rendering becomes **resolution bound**. That means, independently of the geometrical detail, the rendering process will take a fixed amount of time for a certain pixel occupancy, as can be seen in figure 3. Therefore, SVOs are only limited by memory.

[2]An image enhancing technique in 3D computer graphics that increases geometrical detail without actually changing the geometry[8].

*Figure 3: The frame rate increases as the projected size of the model decreases (58 fps vs. 97 fps), since the octree needs not be traversed as deeply to achieve the same visual quality.*

- Streaming: During the rendering process the nodes are only traversed as they are visible and still need further subdivision. This information, namely which nodes are traversed, can be used as feedback to implement automatic streaming, so that it can be possible to render models that do not fit completely into memory.
- Many algorithms like culling, collision detection, etc. benefit from the hierarchy of the octree and also from the fact that all nodes are axis-aligned cuboids.

## 1.4 Disadvantages of sparse voxel octrees

SVOs are static, not in the sense of immutability (of course the data structure can be altered (at runtime)), but in the sense of animation. The approximation of a 3D model through a SVO can be imagined as the set of cells within the uniform grid spanned by the octree boundaries that are intersected by the model's surface. If this model changes (for example a moving character), the set of intersected cells will also change and the octree which describes the altered model looks totally different from the original one. Voxel octrees have no sense of *organizational* hierarchy (like model→arm→hand→fingers) which could be animated. Therefore, the only way to animate a model is to construct

a distinct octree for every pose of the character, or to reconstruct it from another (an animatable) 3D model storage format in every single frame. The first solution consumes too much memory, while the second one consumes too much time.

There are still use cases for SVOs, for example representing static geometry like rocks, mountains, terrain, etc. Upcoming engines and games are going to do exactly that[2, 3]. But, considering the growing percentage of animated objects in games, voxels end up being of limited use.

# 2 Previous work

Samuli Laine and Tero Karras[4] evaluated the viability of voxels as a generic data structure for representing arbitrary geometry on current GPUs. They introduced ,,contour information" which greatly enhances image quality at high zoom levels.

Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre and Elmar Eisemann[5] focused on rendering extremely large data sets that exceed GPU memory by far, using voxels. They use information gained during the rendering process to steer the streaming process. Their technology ,,Gigavoxels" is not limited to solid geometry but is able to represent atmospheric effects such as clouds.

Branislav Siles[1] created ,,Atomontage", a voxel rendering engine featuring highly detailed, visually appealing scenes. What makes this technology stand out among the others is the possibility to render voxels and triangle meshes simultaneously and let them interact with each other. Furthermore, the storage format has to be considered very efficient and it supports runtime decompression. Finally, Atomontage's goal is to use voxels not only for visualization, but also for the simulation of physics, chemistry and some other atom-based scientific models.

# 3 Animating SVOs

Triangle meshes consist of vertices (points). Connected via lines they form triangles. Triangles are chosen because they are the simplest 3D structure. They define a plane and therefore make visualization and interpolation easy. A special feature of triangle meshes is that the triangles form a surface which defines the shape of a 3D model. This means that any transformations can be applied on the vertices and – as long as the transformations are not that drastic – preserve the 3D model's general shape and recognizability.

This is not possible for SVOs because they are rigid: Every node's coordinates are derived by its position inside the octree hierarchy and the octree's minimum and maximum vector. The nodes themselves do not store any position/orientation that could be altered by simulated physical forces or other means of transformation. The only "real" ways of transforming voxel octrees are the ones described in chapter 1.4. Both are unsuitable for real-time applications.

This thesis does not solve the animation problem with one all-mighty algorithm. Neither does it provide specific solutions to all imaginable types of animation. What it does, is to present a general concept, that can be easily applied to particular animation types. Furthermore, two examples are shown to illustrate what this application can look

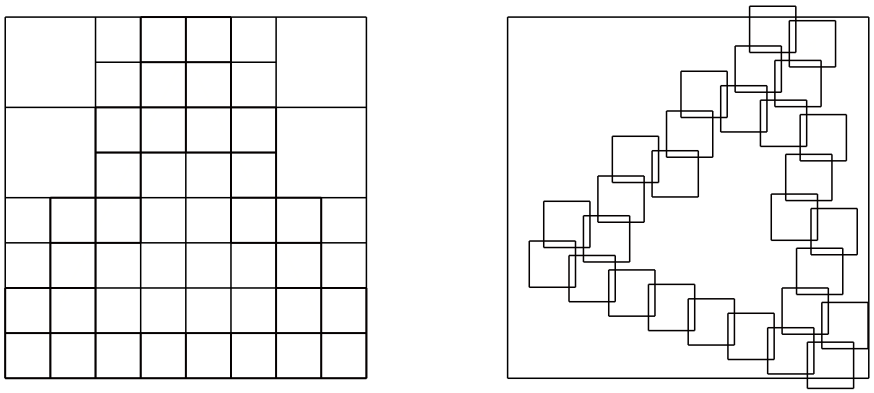

*Figure 4: Left: A quadtree representation (blue rectangles) of a 2D model (red triangle). Right: The same quadtree rotated by x degrees.*

like: Rotation (as an example of a rigid transformation[3]) and Skinning (as an example of a non-rigid transformation).

## 3.1 The core idea of animating sparse voxel octrees

1. Treat every node of a SVO like an atom; a rigid, axis-aligned cuboid; that can be transformed **independently from all other nodes** inside the octree.
2. Apply the transformation to each node during the rendering phase, when the node's final screen position is computed on the fly anyway.

This is the general concept. Two examples of its application follow.

## 3.2 Rotation

Figure 4 (left) shows a SVO representing a 3D model. 2D quadtrees are used for simplicity but the principles shown are valid for 3D octrees as well.

In order to rotate this model by $x$ degrees, the following steps must be executed (for simplicity LOD is not taken into account and the octree is traversed to the leaf level and rendered on the highest available resolution):

1. Traverse the octree to the leaf level and compute every node's minimum and maximum vectors on the fly by deriving them from the node's position inside the octree hierarchy and the octree's minimum and maximum vector.
2. Apply the transformation, in this case a simple rotation matrix, to every leaf node by determining the center point of each leaf node, rotating it, and reconstructing the node at the new position (figure 4 right). (Actually, no data structures have to be altered at all to "reconstruct" a node. The node only has to be **drawn** at its new position, leaving the octree itself untouched.)

[3]A transformation that treats the model like a rigid body. The distance between any two given points of a rigid body remains constant in time[9].

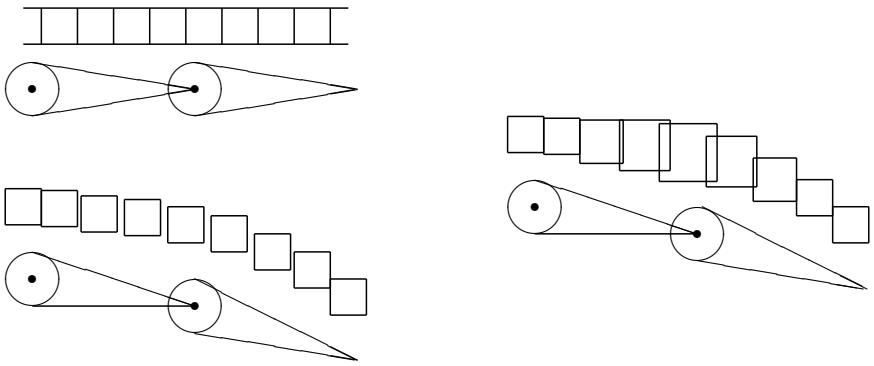

*Figure 5: Left: A trivial skinning implementation causes holes in the surface. Right: Voxels in regions where the surface stretches are enlarged to prevent holes.*

This procedure obviously introduces a visual error. The "real" way of transforming SVOs, or even the construction of the octree in the very first place, introduce a visual error as well, because in both cases not necessarily axis-aligned geometry is approximated with axis-aligned cuboids. However, triangle meshes are also an approximation. Furthermore, the results will show that the visual error is negligible if a high enough sampling rate is used and that the sampling rate for animated SVOs needs not be higher than the one for static SVOs to achieve the same visual quality.

In Fact, the procedure described above can be applied to every rigid transformation and every non-rigid transformation that preserves a 3D model's proportions (like scaling). Non-rigid transformations that do not preserve a 3D model's proportions are a bit more complicated to implement. The problem is that this kind of transformations changes the distance between voxels. As for triangle meshes, this is no problem since the triangles stretch "naturally" across the vertices like a skin. But it can lead to holes in the case of SVOs, if neither the voxel's size nor their sampling rate is adapted accordingly. There does not seem to be a general procedure that can handle **every** non-rigid transformation as there is for rigid transformations. Rather, one has to come up with an individual one for every single case. This should not be too difficult or too much work. The next chapter shows, how the problem was solved in the case of skinning.

## 3.3 Skinning

Skinning[4] causes certain areas of the 3D model to stretch which creates holes in the surface (figure 5 left, figure 11). The implementation of skinning does not differ from the one of rotation, other than it circumvents holes by adapting the size of voxels in those regions (figure 5 right). The stretch factor $S$ by which the voxel is enlarged is approximated with a very simple formula: $S = min(2, W)$, where $W$ is the number of the voxel's bone weights that are greater than 0. While this technique is very primitive,

[4]A technique in computer animation, in which a model is represented in two parts: A surface representation used for visualization (called the skin) and a hierarchical set of bones used for animation (called the skeleton)[10].

*Figure 6: Render an octree and store the depth information in the depth buffer and the visual data (normals in this example) in a set of textures.*

it produces acceptable results. Chapter 6 covers its implied limitations in more depth and chapter 8.1.3 discusses possibilities to improve it and make it more robust.

# 4 Rendering ASVOs

The presented approach solves the animation problem for SVOs, but it introduces a new one: Visualization or rendering. SVOs favor ray tracing as the rendering mechanism because of their hierarchical nature. The fact, that all child nodes geometrically lie inside their parent node makes octrees a very good acceleration structure for intersection tests – which ray tracing is all about. ASVOs clearly destroy this hierarchical nature. The fact that a parent node's boundaries do not necessarily contain the boundaries of its child nodes, makes ray tracing unsuitable as a rendering mechanism since it prohibits the implementation of efficient intersection tests. Of course an acceleration structure could be applied right after the transformation, but that would be too computationally expensive. Therefore a rasterizer implemented on NVIDIA's compute unified device architecture (CUDA) is used to render ASVOs.

## 4.1 The rasterizer

Before the functionality of the renderer is explained, some great advantages of using a rasterizer shall be noted:

- Since rasterizers are the first choice for rendering triangle meshes, triangle meshes and SVOs become compatible and can interact with each other (e.g. cast shadows on one another, reflect each other, etc.).
- All existing visual effects in form of shaders can be used and do not have to be reinvented.
- A fast hardware implementation is possible on current GPU architectures.

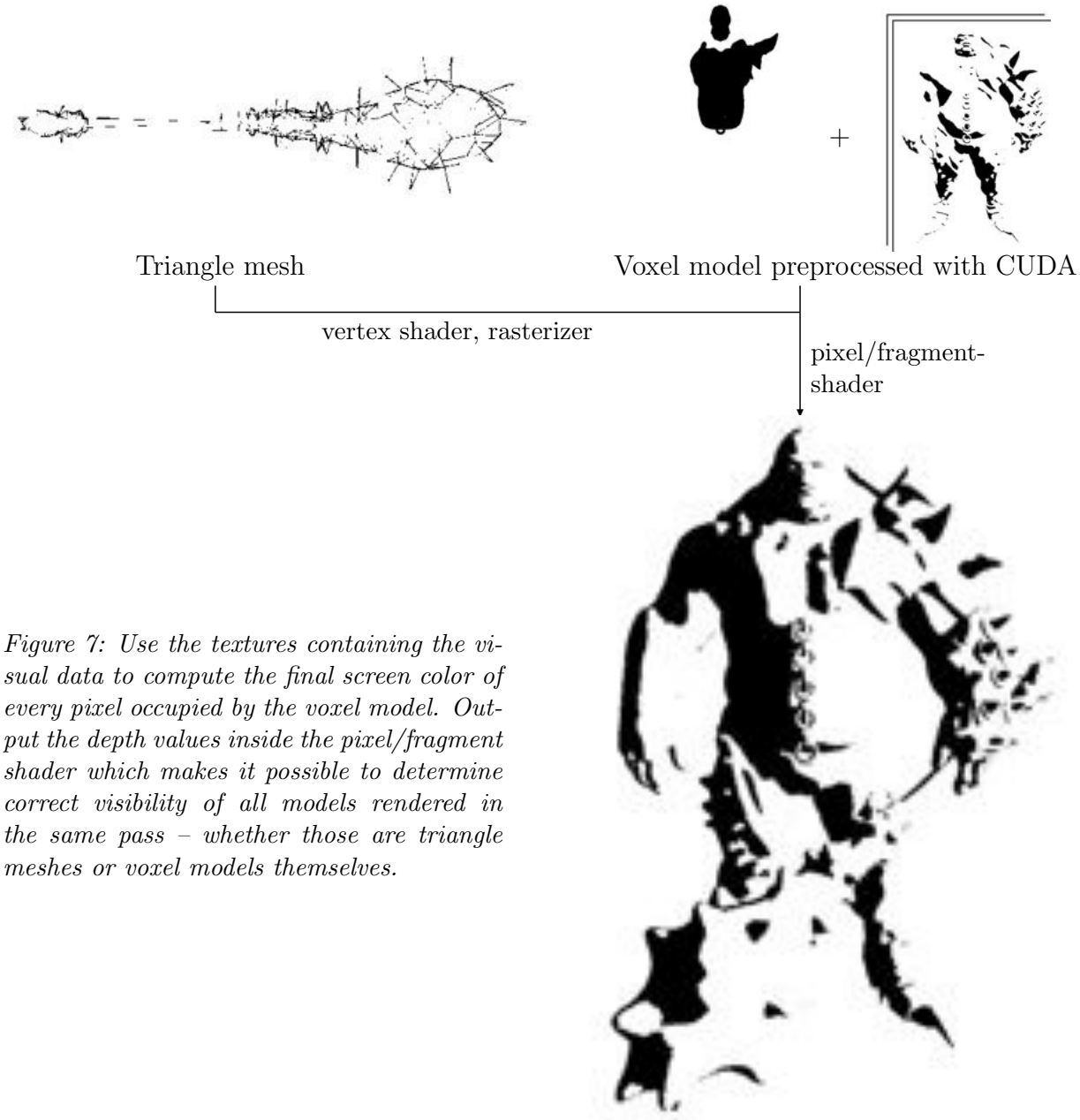


*Figure 7: Use the textures containing the visual data to compute the final screen color of every pixel occupied by the voxel model. Output the depth values inside the pixel/fragment shader which makes it possible to determine correct visibility of all models rendered in the same pass – whether those are triangle meshes or voxel models themselves.*

The actual functionality of the rasterizer is not very special:

1. Using the approach described in chapter 3.1, traverse the octree and determine each visible node's final screen position and depth.
2. Write the depth value in the depth buffer and the visual information (texture coordinates, normals, etc.) associated with the voxel in a separate set of textures.
3. Render a huge quad which occupies the whole screen and therefore forces the graphics card to launch a pixel/fragment shader for each pixel on the screen.
4. Using the depth information from the depth buffer, determine visibility, making it possible to render multiple objects (no matter whether voxel octrees or triangle meshes) correctly.

5. Using the visual information stored in the extra textures, compute the final appearance (color) of each pixel with existing shader effects.

Figures 6 and 7 illustrate the rendering process.

## 4.2 CUDA implementation

The abstract rendering algorithm described can be implemented in many different ways – a GPU implementation with CUDA being just one. That is why this chapter does not go too much into detail, but it spends some words on the implementation of the rendering process for several reasons:

- A hardware implementation is the obvious choice when performance matters.
- As a proof of concept: To show that a real-time voxel rasterizer exists and produces visually appealing results.
- The GPU implementation of the renderer was one of the most outstanding challenges regarding this thesis. Some of its specialties should be revealed.

The CUDA renderer is only responsible for the first two steps of the rendering process. The remaining steps are realized by classical graphics code and shaders.

GPUs are **massive** SIMD (Single Instruction, Multiple Data) architectures. Modern GeForce GPUs consist of one to at most a few so called ,,multiprocessors" and a couple hundred so called ,,streaming processors". A streaming processor is only capable of arithmetic and logic operations, texture lookups, etc., but has no means to handle program flow. This work is done by the multiprocessor. A multiprocessor is able to do context switching, scheduling, task prioritizing, etc. It steers the streaming processors. One conclusion is that GPUs can perform a vast number of parallel operations due to the many streaming processors. Another conclusion is that the previous statement is only true, if all streaming processors are assigned the same task, because if the program flow diverges (takes different execution paths – for example because of a conditional branch whose condition was evaluated differently among the streaming processors) the multi processor will choose one execution path and halt every streaming processor not executing this path. The execution proceeds until the path merges back to the main program flow and this behavior is repeated for all execution paths until every one is terminated.

These circumstances contradict the desire to traverse SVOs, since traversal of hierarchical data structures in its nature is everything but divergence-free. But traversal has to be done in order to realize LOD and derive the voxel's final screen positions. Additionally, most NVIDIA GPUs do not support recursive function calls and on the ones that do, they are very expensive. The local memory per thread is low which makes it hard to implement an explicit (call-) stack. The global memory on the other hand is not scarce at all, but it is much slower.

From all constraints arose the need for a **stackless, divergence-free, parallel octree traversal algorithm** - a subject worth its own thesis. The problem was solved with a queue based rasterizer. The result was a stackless, low-divergence, parallel octree traversal algorithm. The algorithm is given in figure 8.

```
1: { Calling code for the rasterizer-kernel }
2: q ← init() { q: Queue }
3: insert(q, rootNode)
4: while q is not empty do
5:    launchKernel(length(q))
6: end while
7:
8: { Rasterizer-kernel }
9: v ← removeAny(q)
10: if projectedSize(v) > pixelSize and hasChildren(v) then
11:    for all c in getChildren(v) do
12:       insert(q, c)
13:    end for
14: else
15:    draw(v)
16: end if
```

*Figure 8: Rasterizer code.*

In the first part a queue ($q$) storing voxels is initialized and the root node of the octree is inserted. As long as $q$ is not empty a kernel is launched with as many threads as $q$ contains elements.

Every thread operates on a single voxel in the queue. It determines its position and size and also decides whether the voxel needs further subdivision. If so, its children are inserted into the queue to be processed in the next kernel call. Otherwise, the voxel is drawn.

This is a very basic implementation only featuring LOD. Optimizations like viewing frustum culling or occlusion culling could be easily added in form of a predicate (between lines 9 and 10) that determines if a voxel should be processed at all. This solution fulfills nearly all the requirements stated before. It is not divergence-free, but only has very low divergence, mainly when some of the threads launched draw voxels and the others subdivide voxels. This results in a maximum of one branch per warp[5].

[5]On CUDA devices, the multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps[6].

# 5 Results

A new way of animating SVOs was introduced, it is called Animated Sparse Voxel Octree, or ASVO. The animation takes place inside the renderer when the final screen positions of voxels would have to be computed anyway and does not alter the initial data structure, making it very efficient. It was shown how the presented general concept can be applied to specific types of animation and two examples – rotation and skinning – were provided.

Furthermore, a hardware based rasterizer was presented. It accomplishes a frame rate of ∼40 fps on a screen resolution of 800 x 800 and a virtual model resolution of $1024^3$ voxels with phong shading, shadow mapping and hardware skinning. Hereby it is only limited by the GPU memory, since the memory consumption grows exponentially with every octree level. Model resolutions of up to $4096^3$ voxels could have been achieved if it focused on storage efficiency which it does not: State of the art voxel rendering engines use ∼one bit per voxel[1] whereas this rasterizer uses 48 bytes per voxel. Figure 13 shows the results.

# 6 Limitations

The presented technique relies on model animations to be moderate. If the boundaries of a node were *completely* unrelated to the boundaries of its children, it would not be possible to make any assumptions on the children's positions which would prevent LOD: A node could not be culled (and therefore all its children be dismissed), because some of its children might still be visible in the viewing frustum. Traversal could not be stopped when a node reaches the desired size, because some of its children might be 10 times closer to the camera and therefore still require further subdivision. But all of that is possible if the animations are moderate. Because then, LOD and culling operations can depend on a threshold and only be applied when voxels meet certain conditions plus/minus an experimentally determined margin. With the same argumentation ASVOs could be rendered with a ray-tracer. For example, voxels could be artificially enlarged to be hit by more rays and not be rejected too early. A rasterizer was chosen because it seemed easier to implement at the time of writing this thesis.

The computation of the stretch factor in its current form has several drawbacks:

1. It is very inflexible. The stretch factor can only have one of the values 1 and 2 (theoretically it could also have the value 0, but normally voxels with zero bone weights don't occur in skinned models). So either a voxel is stretched or it is not. When it is stretched there are situations in which a factor of 2 is too high, unnecessarily enlarging the voxel and either forcing further subdivision or occupancy of multiple pixels on the screen. There are also situations in which a factor of 2 is too low and holes are still visible in the surface.

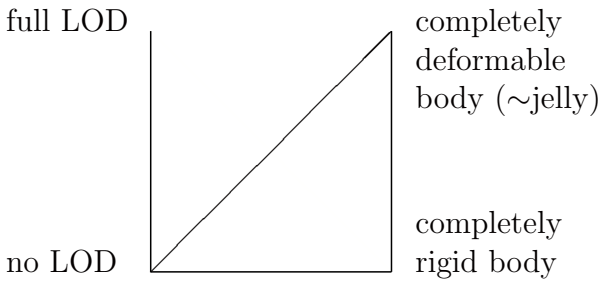


*Figure 9: Relationship between the animation grade of a model and how effectively LOD can be applied to it.*

2. It is static. The stretch factor is applied independently of whether the bones a voxel belongs to are animated in the current frame or not. By not taking this information into account, LOD is mostly prevented: The more bones and animations a model contains, the more often voxels are enlarged and subdivision forced even when it might not be needed. Figure 9 illustrates the relationship between the animation grade of a model and how effectively LOD can be applied to it, given the current form of stretch factor computation.

# 7 Conclusion

The current study on SVOs proves their suitability for being used in real-time graphics applications and for representing arbitrary geometry. id software's$^{\text{Inc}}$ ,,Rage" and Crytek's$^{\text{GmbH}}$ ,,Crysis 2" will be the first mainstream games which feature voxels[2, 3]. Voxels have great benefits over triangle meshes such as:

- Geometrical detail independency (screen resolution bound)
- Memory efficiency (for models with great mesostructural detail)
- Being an "all-in-one" data structure whose hierarchical nature enables (including, but not limited to):
  - Level of detail
  - Live streaming of huge data sets that do not fit into device memory
  - Simulation of atom-based physical models

Not only possibilities to render voxels in a fast way have been researched but also techniques to circumvent some of their major drawbacks:

- The fact that voxels are suitable for representing objects with great mesostructural detail, but their animation is not feasible, led to making voxels and triangle meshes compatible and appear inside the same scene, interacting with each other.

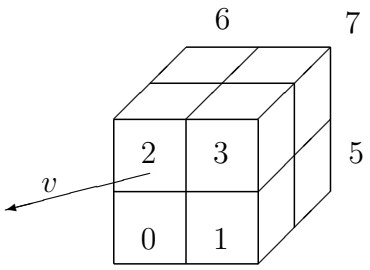


*Figure 10: A plane perpendicular to the viewing direction v moving towards the node center, would intersect the voxels in the order ,,2 0 3 1 6 4 7 5". Rendering them in this order fills up the depth buffer early with small values and many nodes can be rejected faster.*

- The present thesis explored a completely different way of rendering and animating SVOs in the hope that it serves as an inspiration for others and voxels eventually become usable for more than just static scenes one day. Plants, vegetation and nature are perfect examples of what can be visualized with animatable voxels.

For a long time the lack of freedom in GPU programmability prohibited extensive use of voxels. As it increases and considering the recent work on the topic, voxels most likely will continue to be a current research field – especially with the release of the new video game console generation (XBOX 3 and PlayStation 4).

# 8 Future work

I would like to keep on working on this great technology. Despite the progress in voxel graphics, much work still has to be done. Future research topics will not only cover improving visual quality, rendering performance and making voxels more powerful, but the whole voxel infrastructure: Modeling tools, integration of physics simulation (including destructible models), and more.

## 8.1 Possible enhancements for ASVOs

### 8.1.1 Viewing frustum culling

The viewing frustum (VF) is the region of space that may appear on the screen. It is determined by the camera.

After projection each voxel is given in normalized screen coordinates. Every point (x, y, z) inside the viewing frustum satisfies the equations -1 $\leq$ x $\leq$ 1, -1 $\leq$ y $\leq$ 1 and 0 $\leq$ z $\leq$ 1. All that is left now is to test whether two axis-aligned rectangles intersect which can be done very fast. This test would be much slower between a rectangle and a triangle to perform.

### 8.1.2 Occlusion culling

Another reason for a model not to be visible in the rendered image is because it is occluded (completely or partially) by another model. This issue is generally solved by using the

z- or depth buffer. However, the depth buffer is only able to dismiss pixels, so it will not save one from traversing the octree, but only guarantee correct visibility.

The hierarchical visibility algorithm[7] solves this problem in a more efficient way: Instead of using a flat depth buffer, it uses a quadtree in which every node stores the depth value of the child which is nearest to the camera. Again, this algorithm is suited to be used along with voxels as intersection tests between them and quadtree nodes are very fast.

The earlier the depth buffer fills up with small values, the better, because future octree nodes can be dismissed faster. This fact does not satisfy to sort all scene objects by distance to the camera, but each octree can be traversed in a way that visits nodes close to the camera first. This technique is illustrated in figure 10.

### 8.1.3 Improved skinning

The problem of holes in surfaces of animated voxel models could be solved very easily: Bone weights and indices would have to be stored per voxel corner, rather than per voxel. Instead of transforming the voxel center, all eight corners would get transformed and the new voxel would be defined as the axis-aligned bounding box of those eight points. Although this technique is not quite optimal as it leads to too big voxels in many cases, it eliminates holes completely, since all voxels form a closed surface in the binding pose and share corners with each other which therefore would be animated equally across all voxels. Unfortunately this technique is too computationally expensive.

More opportunities arise to improve the current implementation, if ASVOs are created with modeling tools that natively support voxels. Stretch factors could be automatically computed, manually refined and stored per voxel. However, it most likely would not be possible to store adjusted stretch factors for every animation frame as this would consume large amounts of memory.

A better solution has still to be found.

## 8.2 Modeling Tools

Modeling tools that support creating voxel models natively will become inevitable for several reasons. A modeling tool should give the designer WYSIWYG results, which is not possible with triangle→voxel conversion. Furthermore, memory consumption of SVOs depends less on the voxel count than on the number of octree levels (assuming real world conditions). Thus, a designer needs not to fear exceeding the hardware budget by adding more nodes. He will choose an appropriate octree depth at the start and then freely sculpt the model. Designers need to really step out of the box and make models that otherwise could not be created with triangles or shading tricks – for example by creating hang overs at the mesostructure level which cannot be simulated with parallax mapping. Finally the modeling process will change from moving vertices around to actual sculpting, very similar to making pottery. Zbrush[6] and 3D-Coat[7] are two modelers that already support this technique.

[6] A 3D modeling tool. `http://www.pixologic.com`

[7] A 3D modeling tool. `http://www.3d-coat.com`

## 8.3 Physics simulation and more

Most voxel rendering systems (including the one presented here) use static SVOs. However, deformable and destructible objects will need a voxel data structure that is alterable at runtime. A dynamic data structure does not offer as many optimization possibilities regarding space and time requirements as a static data structure does. It has to be researched whether such a format is still feasible for real-time applications.

As stated before, voxels have the potential to become an "all-in-one" data structure. This characteristic could be strengthened by using voxels for more than only visualization: Simulation of atom-based physical and chemical models whereby voxels would play the role of atoms. It is questionable if compression schemes will allow to use such a high number of voxels inside a simulation.

**Acknowledgments**. I thank Onno van Braam and Dimitry Parkin for providing me with 3D models for testing and illustration purposes. I thank Sylvain Lefebvre and Arjan Westerdiep for providing me with illustrations for this thesis. I thank Dr. Werner Hansmann and Prof. Dr. Leonie Dreschler-Fischer for their great and professional support and advice.

*Figure 11: The effects of surface stretching with voxel size adaption turned off.*

*Figure 12: Extreme zoom levels reveal the underlying voxel structure.*

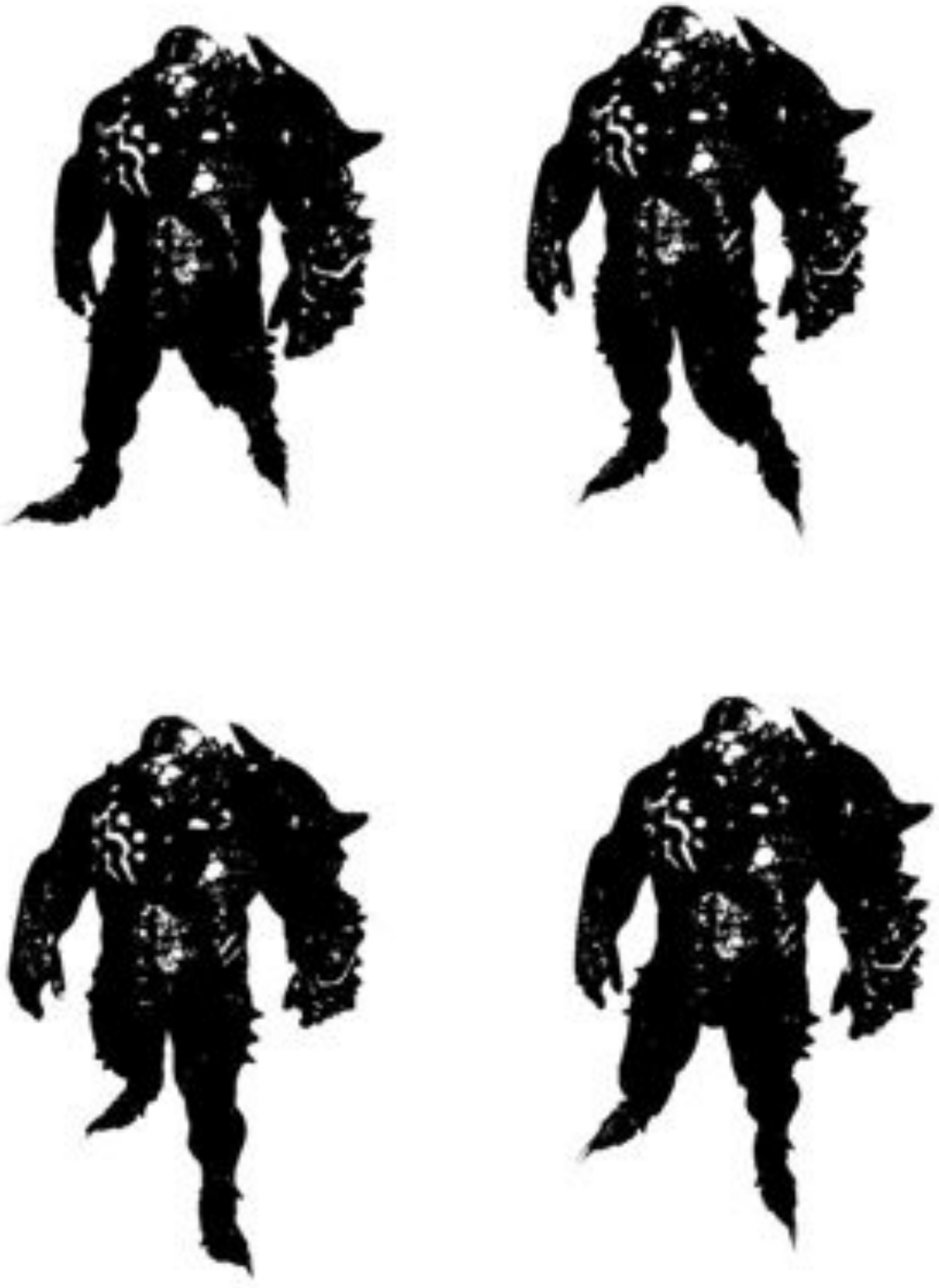

*Figure 13: Imrod walk cycle; voxel count: ∼3M; frame rate: 36 fps; GPU: Nvidia GeForce GTX 460 with 2 GB of memory; resolution: 800 x 550 pixels; effects: Phong shading, normal mapping, shadow mapping, hardware skinning.*

# References


[1] Branislav Siles:
Atomontage engine.

A voxel rendering engine.
http://atomontage.com/?id=home

[2] Ryan Shrout:
„John Carmack on id Tech 6, Ray Tracing, Consoles, Physics and more".

Interview with John Carmack, in *PC Perspective*, March 12, 2008.
http://www.pcper.com/article.php?aid=532

[3] Cevat Yerli:
„Future of Gaming Graphics".

Keynote at *GCDC 2008*.
http://video.golem.de/games/1577/gcdc-2008-panel-future-of-gaming-graphics.html

[4] Samuli Laine and Tero Karras:
„Efficient Sparse Voxel Octrees".

In *Proceedings of ACM SIGGRAPH 2010 Symposium on Interactive 3D Graphics and Games*, pages 55–63. ACM Press, 2010.
http://www.tml.tkk.fi/~samuli/

[5] Crassin, Cyril and Neyret, Fabrice and Lefebvre, Sylvain and Eisemann, Elmar:
„GigaVoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering" (author's version).

In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, to appear. ACM Press, Boston, MA, Etats-Unis, February 2009.
http://artis.imag.fr/Publications/2009/CNLE09/

[6] „NVIDIA CUDA C Programming Guide".

http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf

[7] Tomas Möller and Eric Haines:
„Occlusion Culling Algorithms".

Excerpt from *Real-Time Rendering*, in *Gamasutra*, November 9, 1999.
http://www.gamasutra.com/view/feature/3394/occlusion_culling_algorithms.php?page=2

[8] László Szirmay-Kalos, Tamás Umenhoffer:
,,Displacement Mapping on the GPU - State of the Art".

In *Computer Graphics Forum*, Volume 27, Number 6, pages 1567–1592. 2008.
`http://sirkan.iit.bme.hu/~szirmay/egdisfinal3.pdf`

[9] ,,Rigid body".

In *Wikipedia*. Retrieved September 11, 2010
`http://en.wikipedia.org/wiki/Rigid_body`

[10] ,,Skeletal animation".

In *Wikipedia*. Retrieved September 11, 2010
`http://en.wikipedia.org/wiki/Skeletal_animation`

## Erklärung

Ich versichere, dass ich die Arbeit selbstständig verfasst und keine anderen, als die angegebenen Hilfsmittel - insbesondere keine im Quellenverzeichnis nicht benannten Internetquellen benutzt habe, die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ich bin mit der Einstellung der Bachelor-Arbeit in den Bestand der Bibliothek des Departments Informatik einverstanden.

---

Dennis Bautembach
Hamburg, den 25. Februar 2011