UNIVERSITÄT SIEGEN

Theorie
und Praxis
für Karrieren
von morgen

Technical Report

Fast Hierarchical 3D Distance Transforms on the GPU

Nicolas Cuntz and Andreas Kolb

Version: May, 15th, 2007

Faculty 12
Computer Graphics and Multimedia Group
Prof. Dr. Andreas Kolb

Abstract

This paper describes a novel approach for the GPU-based computation of approximate 3D Euclidean distance transforms, i.e. distance fields with associated vector information to the closest object point. Our hierarchical method works on discrete voxel grids and uses a propagation technique, both on a single hierarchy level and between the levels. We assume a closed object, so that a signed distance of a voxel w.r.t. the object boundary is defined. The object's boundary is given by means of the grid voxel classification as interior or exterior and the pre-initialization of voxels lying directly next to the boundary. The propagation method can be seen as a filtering process, where the voxel's distance transform is updated by computing alternative distances according to the distance transforms for neighbor voxels. Using our hierarchical approach, the effort to compute the distance transform is significantly reduced.

Our technique is purely GPU-based. We build upon a specific approach to work on a 3D distance transform using *Multiple Render Targets (MRT)*. All hierarchical operations are performed on the GPU.

ACM Categories: I.3.5 Computer Graphics (Computational Geometry and Object Modeling - Curve, surface, solid, and object representations)

1. Introduction

Signed or unsigned distance fields have many applications in computer graphics, scientific visualization and related areas. They can be used for implicit surface representation and collision detection [KLRS04], for skeletonization [ST04] or for accelerated volume raytracing [HSS*05], to state just a few.

Computing a 3D Euclidean distance transform is a well-studied problem (see [Cui99] for an overview). Depending on the initial object representation, as voxel grid or as explicit geometric representation, different approaches have been proposed. Concerning the voxel grid approach, there are two major categories, propagation methods and methods based on Voronoi diagrams. Propagation methods propagate the distance information to the neighboring voxels, either by spacial sweeping or by contour propagation.

GPU-based approaches have been presented for distance transform computation in the 2D case for voxel grid input [ST04, RT06] and for 3D polygonal input [SPG03, SGGM06].

The method presented in this report works on 3D voxel grid input models and is based on the propagation approach. The approach uses a specific hierarchical technique consisting of *push-downs* and *pull-ups* to halve and double the grid resolution, respectively, thus exponentially reducing the number of propagation steps required for the computation of an approximate distance transform. Our system demonstrates that the error can efficiently be canceled with minor computational costs.

The remainder of this paper is structured in the following way: Sec. 2 discusses related research results. Our hierarchical approach is described in Sec. 3, whereas Sec. 4 contains a discussion of the error occuring in our algorithm. Sec. 5 gives details on the implementation and Sec. 6 provides experimental results obtained using two different test cases.

2. Prior Work

This section describes the main concepts of voxel grid techniques based on Voronoi diagrams (Sec. 2.1) and propagation (Sec. 2.2) utilizing programmable Graphics Processing Units (GPUs) to compute distance transforms.

Consider a voxel grid and a closed object boundary $\delta\Omega$ marked in the voxel grid. The Euclidian distance transform dt (also called Feature or complete distance transform) for a voxel $\mathbf{P}$ is defined as

$$\mathrm{dt}(\mathbf{P}) = \left(\min_{\mathbf{Q}\in\delta\Omega} \{\|\mathbf{P}-\mathbf{Q}\|\}, \arg\min_{\mathbf{Q}\in\delta\Omega} \{\|\mathbf{P}-\mathbf{Q}\|\} \right),$$

i.e. $\mathrm{dt}(\mathbf{P})$ stores the distance plus the point $\mathbf{Q} \in \delta\Omega$ closest to $\mathbf{P}$. We refer to the distance and the closest point of $\mathrm{dt}(\mathbf{P})$ as $\mathrm{dt}_d(\mathbf{P})$ and $\mathrm{dt}_\delta(\mathbf{P})$, respectively. In case of a signed distance transform, $\mathrm{dt}_d(\mathbf{P})$ is negative in the inner region and positive in the outer region.

2.1. The Voronoi Diagram Approach

A 2D Voronoi diagram is a partitioning of the plane into cells w.r.t. a fixed set of points (*seeds*), where each cell contains all points in the planes closest to one seed. It is clear that the distance transform can be obtained by setting Voronoi seeds onto the object's boundary.

2D-Voronoi diagrams can easily be determined using rasterization techniques. Therefore, cones with a common opening angle are placed over each seed and the resulting scene is rendered from top-view using the OpenGL depth buffer function `GL_LESS`. Automatically, the resulting image is the Voronoi diagram of the given seeds and the depth buffer contains the distance information to the corresponding seed. Hoff et al. [HKL*99] extend this approach to other geometric objects and to 3D. Sigg et al. [SPG03] and Sud et al. [SGGM06] present

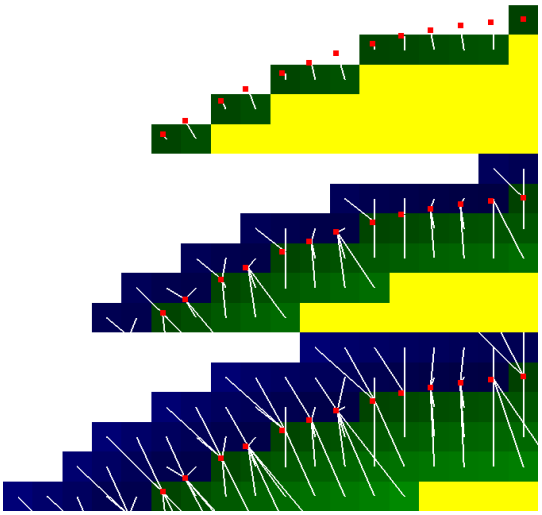

Figure 1: *Two propagation steps. Note that* dt *is initialized with precise sub-pixel references in this example. We use a* 3×3 *(*$\times 3$*) structure element in order to propagate the distance information.*

a GPU-based implementation of the Voronoi-based approach.

2.2. The Propagation Approach

For this approach, the object boundary is given in a voxel grid. The original approach was introduced for 2D images, but can easily be adapted to 3D. An extensive survey of related techniques is given in [Cui99]. The (signed) distance transform is initialized in the following way:

$$\mathrm{dt}^0(\mathbf{P}) = \begin{cases} (0, \mathbf{P}) & \text{if } \mathbf{P} \in \delta\Omega \\ (M, *) & \text{if } \mathbf{P} \in \Omega_+ \text{ (obj. exterior)} \\ (-M, *) & \text{if } \mathbf{P} \in \Omega_- \text{ (obj. interior)} \end{cases}$$

where M is greater than any possible distance between grid voxels. A propagation step for the distance works using a structure element $\mathcal{M}$, defining a local neighborhood, by taking a minimum:

$$\mathrm{dt}_d^{i+1}(\mathbf{P}) = s_{\mathbf{P}} \min_{\mathbf{Q} \in \mathcal{M}(\mathbf{P})} \{ \left\| \mathrm{dt}_\delta^i(\mathbf{Q}) - \mathbf{P} \right\| \}$$

The sign $s_{\mathbf{P}}$ is taken from $\mathrm{dt}_d^i(\mathbf{P})$, and $\mathrm{dt}_\delta^{i+1}(\mathbf{P})$ is updated using the selected $\mathbf{Q}$. This algorithm, using a $3 \times 3 \times 3$ structure element, is known to produce an approximate distance transform if applied a sufficient number of times for all voxels in parallel (refer to [CM99] for a very comprehensible description). See Fig. 1 for a visualization of two sequential propagation steps.

A fast variant of this algorithm is presented by Tsitsiklis [Tsi95]. This approach uses a priority queue approach to optimize the order of the distance transform updates. Strzodka and Telea [ST04] present a GPU-based 2D approach using an arc length parametrization for $\delta\Omega$, which, in general, does not carry over to the 3D case.

Rong and Tan [RT06] present a GPU-based technique called Jump Flooding primarily designed for the generation of Voronoi diagrams for a given set of seeds, but also suitable to compute unsigned distance transform. Here, propagation is performed within a neighborhood of voxels that are k steps away from the central point $\mathbf{P}$. In order to reach all voxels in a logarithmic number of steps, the offset k is halved after each step. This can be seen as an hierarchical procedure where k specifies the grid resolution during one step. Rong and Tan attempt to correct the resulting error by performing one or two additional propagation steps after termination. However, no qualitative information is given about the remaining error. Moreover, no performance results are given for 3D grids.

Besides speed, the main benefit of our approach is a fully hierarchical design and the idea to control the error independently for each hierarchy level. In addition, the use of Multiple Render Targets allows for direct output into a 3D grid of resolution 256^3 (according to current graphics hardware specifications). In comparison to Jump Flooding, our algorithm is asymptotically faster, as the levels in the hierarchy are decreasing in size. In the Jump Flooding approach, each step involves the same complexity w.r.t. to the grid resolution. Moreover, the caching capability of modern graphics hardware is exploited by storing hierarchy levels in separate textures.

3. Hierarchical 3D Distance Transforms

This section describes the main steps involved in our hierarchical approach. The algorithm is introduced and explained in the following section.

3.1. Algorithmic Overview

The input for our algorithm is a voxel grid representing the object boundary separating the space into an inner and outer region. This is done by initializing dt with exact value for negative (inner) voxels next to the boundary. We store precise sub-pixel references rather than setting the initial voxel distances to 0 in order to get an exact representation of our (implicitly defined) test geometries (see Sec.6). The object boundary $\delta\Omega$ could be as well defined by selecting an appropriate set of 0-distance voxels.

Thus, dt is defined for voxels directly next to the boundary. Using super-sampling, the grid resolution is repeatedly reduced by a factor of 2 and, in parallel, the distance transform is (implicitly) propagated (*push-down*, Sec. 3.2). The recursion breaks for the smallest level, for

which we compute the complete distance transform using the propagation method described in Sec. 2.2 by repeatedly propagating until the complete grid is filled. Note that, because of the low resolution, we can do this without performance deficit. The push-down is then inverted by a *pull-up* (Sec. 3.3) in combination with k_p additional propagation steps, where k_p is a small positive integer.

The algorithm is summarized in the following listing:

```
01 initialize +/-
02 compute distance next to object boundary
03
04 // push-down
05 for level = 1 to n do
06   reduce to 1/2 resolution
07 done
08
09 compute dt for level n
10
11 // pull-up
12 for level = n-1 to 1 do
13   recursively combine with level+1
14   perform k_p propagation steps
15 done
```

3.2. Push-down

In the push-down pass (from fine to coarse resolution), distance information for voxels touching the object boundary are propagated to lower hierarchy levels. In our 128^3 example, we push-down until a resolution of 8^3 voxels.

The push-down is done by super-sampling surrounding voxels, using a factor of 2 in each dimension. The distance transform ${}^{j}\,\mathrm{dt}$ is combined and propagated from level j to level $j+1$ using the following update rule (see Fig. 2):

$$
\begin{aligned}
&{}^{j+1}\,\mathrm{dt}_d({}^{j+1}\mathbf{P}) = \\
&s_{{}^{j+1}\mathbf{P}} \cdot \min_{{}^{j}\mathbf{Q} \in \mathcal{N}_2({}^{j+1}\mathbf{P})} \{ \left\| {}^{j}\,\mathrm{dt}_\delta({}^{j}\mathbf{Q}) - {}^{j+1}\,\mathbf{P} \right\| \}
\end{aligned}
$$

with sign taken from ${}^{j}\,\mathrm{dt}_d({}^{j+1}\mathbf{P})$, and ${}^{j+1}\,\mathrm{dt}_\delta({}^{j+1}\mathbf{P})$ is updated using the selected ${}^{j}\mathbf{Q}$. Here, ${}^{j}\mathbf{P}$ and $\mathcal{N}_2$ denote a voxel on level j and the super-sampling neighborhood for the reduction factor 2, respectively.

Note that the voxels initialized with $\pm M$ are properly handled in the next level, i.e. the correct references are set for ${}^{j+1}\mathbf{P}$, if at least one ${}^{j}\mathbf{Q} \in \mathcal{N}_2({}^{j-1}\mathbf{P})$ has a valid reference. Thus, the distance information is also propagated in a spacial sense.

3.3. Pull-up

For now, we assume that the correct distance transform ${}^{j}\,\mathrm{dt}$ for the coarse level is given. In principle, this is the case for the coarsest level, because for this level, we calculate ${}^{j}\,\mathrm{dt}$ explicitly (Sec. 3.1).

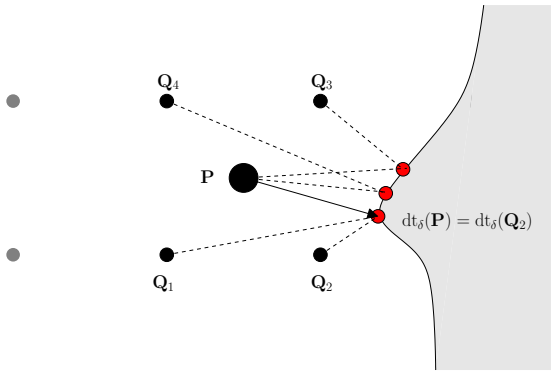

Figure 2: *Push-down: Computing* ${}^{j+1}\,\mathrm{dt}$ *for samples in a coarse grid. The closest reference point w.r.t. level* j *triggers the push-down of the distance transform to level* $j+1$.

The pull-up pass (from coarse to fine resolution) works in much a similar way as the reduction described in Sec. 3.2. Eight surrounding samples in the coarse grid around a voxel $\mathbf{P}$ are checked and the minimal distance determines the reference point for $\mathbf{P}$. Since we have already computed distance information nearby the object boundary, this step is only performed for points which contain $\pm M$ as distance. Note that this can be done easily because each level is stored separately.

As mentioned (see above), the result after a pull-up pass is an approximation of the distance transform ${}^{j-1}\,\mathrm{dt}$. To correct the error, we follow two strategies:

First, we perform k_p additional propagation steps as described in Sec. 2.2 for each level. The first motivation for this correction is the fact that the error in the distance value is constantly bounded (see Sec. 4.1). This strategy is particularly useful in cases where we have no larger displacements in the distance transform, because then, the (constant) expansion of the propagation steps will annihilate the constant error. A second motivation is the observation that the continuous distance field strongly varies near to the object boundary $\delta\Omega$, whereas the variation far from the object boundary is smaller. Thus, based on the exact values next to the boundary, the area close to the boundary will be adjusted by the additional propagation steps.

Secondly, the reference point $\mathbf{S}' = {}^{j-1}\,\mathrm{dt}_\delta({}^{j-1}\mathbf{P})$ resulting from the pull-up step is tracked, and the surrounding voxels of this point at the same level are used for a refinement ${}^{j-1}\,\mathrm{dt}'_d({}^{j-1}\mathbf{P})$. For this purpose, super-sampling as in Sec. 3.2 is performed around $\mathbf{S}'$:

$$
\begin{aligned}
&{}^{j-1}\,\mathrm{dt}'_d({}^{j-1}\mathbf{P}) = \\
&s_{{}^{j-1}\mathbf{P}} \cdot \min_{{}^{j-1}\mathbf{Q}' \in \mathcal{N}_2(\mathbf{S}')} \{ \left\| {}^{j-1}\,\mathrm{dt}_\delta({}^{j-1}\mathbf{Q}') - {}^{j-1}\,\mathbf{P} \right\| \}
\end{aligned}
$$

with sign taken from ${}^{j-1}\,\mathrm{dt}_d({}^{j-1}\mathbf{P})$, and ${}^{j-1}\,\mathrm{dt}'_\delta({}^{j-1}\mathbf{P})$ is updated using the selected ${}^{j-1}\mathbf{Q}'$. This is an improvement especially for far reference points which are displaced by a small offset.

Ideally, one would like a pull-up method that generates ${}^{j-1}\,\mathrm{dt}$ from ${}^{j}\,\mathrm{dt}$ without producing any error, using a minimal number of propagation steps k_p on each level. Unfortunately, an optimal value k_p is hard to determine, even though a constant error bound for the distance component ${}^{j-1}\,\mathrm{dt}_d$ can be determined for the general situation (see Sec. 4.1). This is related to the fact, that a reference can, in some cases, point in a wrong direction, independently from the actual distance error in ${}^{j-1}\,\mathrm{dt}$.

4. Error Analysis

This section discusses the distance error that occurs when performing one pass of the hierarchical algorithm (see Sec. 3). Refer to Sec. 6.2 for measurements of the overall error in our sample geometries.

4.1. General discussion

First, we consider the error that generally occurs during one super-sampling step. This examination is valid for the push-down pass as well as for the pull-up pass (see Sec. 4.2).

An error is generated in point $\mathbf{P}$, if and only if the following situation is given: Assuming that the input grid is filled with the correct distance transform, there exists a reference point $\mathbf{S} \in \delta\Omega$ in the input grid with

$$\begin{array}{ll} (1) & \mathrm{dt}_d(\mathbf{Q}) \;< \|\mathbf{Q}-\mathbf{S}\| \quad \text{and} \\ (2) & \|\mathbf{P}-\mathbf{S}\| < \|\mathbf{P}-\mathrm{dt}_\delta(\mathbf{Q})\|, \end{array}$$

where $\mathbf{P}$ is a point for which we compute a new distance value by taking the reference $\mathrm{dt}_\delta(\mathbf{Q})$ stored in the input grid point $\mathbf{Q}$. This reference is taken instead of the closer point $\mathbf{S}$ (see Fig. 3), leading to a wrong value for the distance $\mathrm{dt}_d(\mathbf{P})$ and an incorrect reference $\mathrm{dt}_\delta(\mathbf{P})$. In the following, we will discuss the distance error ϵ_d in $\mathrm{dt}_d(\mathbf{P})$ rather than the displacement of $\mathrm{dt}_\delta(\mathbf{P})$.

The error ϵ_d is given by the difference between the reference taken from $\mathbf{Q}$ and the missed reference $\mathbf{S}$:

$$\epsilon_d = \|\mathbf{P}-\mathrm{dt}_\delta(\mathbf{Q})\| - \|\mathbf{P}-\mathbf{S}\|$$

Property 1 *The error ϵ_d is bounded by $\sqrt{3}\Delta$ where Δ is the grid spacing in the fine grid.*

Proof: Because of $\|\mathbf{P}-\mathbf{Q}\| = \frac{\sqrt{3}}{2}\Delta$, we can write

$$\epsilon_d \le \mathrm{dt}_d(\mathbf{Q}) + \frac{\sqrt{3}}{2}\Delta - \|\mathbf{P}-\mathbf{S}\|\,.$$

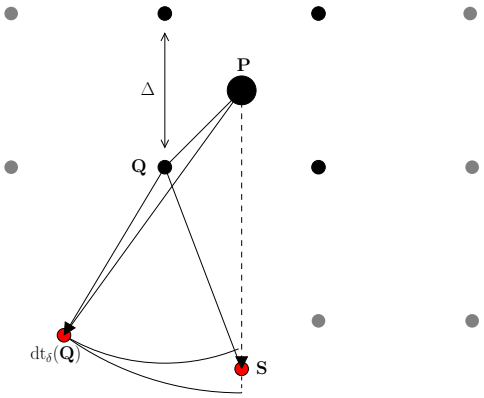

Figure 3: *Situation where the hierarchical propagation produces an error – $\mathrm{dt}_\delta(\mathbf{Q})$ is closer to $\mathbf{Q}$ than $\mathbf{S}$ while $\mathbf{S}$ is closer to $\mathbf{P}$ than $\mathrm{dt}_\delta(\mathbf{Q})$.*

We know that $\mathrm{dt}_\delta(\mathbf{Q})$ must be closer to $\mathbf{Q}$ than $\mathbf{S}$, yielding:

$$\epsilon_d \le \|\mathbf{Q}-\mathbf{S}\| + \frac{\sqrt{3}}{2}\Delta - \|\mathbf{P}-\mathbf{S}\|$$

From the triangle inequality follows $\|\mathbf{Q}-\mathbf{S}\| - \frac{\sqrt{3}}{2}\Delta \le \|\mathbf{P}-\mathbf{S}\|$, so we get

$$\epsilon_d \le \sqrt{3}\Delta.$$

□

This estimation is a sharp bound for a very degenerated situation (see below). Two worst cases for push-down are visualized in Fig. 4. Note that the figure addresses the 2D case with bound $\sqrt{2}\Delta$. However, the 3D case is similar.

The first figure (a) shows a scenario where the corners $\mathbf{Q}_1,\ldots,\mathbf{Q}_4$ have references to some distant points marked in red. The white circles around $\mathbf{Q}_1,\ldots,\mathbf{Q}_4$ indicate the regions where no other references can exist, because otherwise one reference would have been missed in the fine grid. Point $\mathbf{S}$ cannot be in region $\mathcal{B}$ (dark-gray), because we want $\mathbf{S}$ to be closer to $\mathbf{P}$ than $\mathrm{dt}_\delta(\mathbf{Q})$. Thus, the only regions where $\mathbf{S}$ can lie are those marked as $\mathcal{A}$.

Remark 1 *By simple calculus, it can be shown that for push-down, the worst case for (a) is bounded by $\frac{\sqrt{2}}{2}\Delta$, if assumed that $\mathrm{dt}_\delta(\mathbf{Q})$ is located outside the sample square. This is the case for $\|\mathbf{Q}-\mathrm{dt}_\delta(\mathbf{Q})\| = \frac{\Delta}{2}$, and $\mathrm{dt}_\delta(\mathbf{Q}_i) = \mathbf{Q}_i$ for $\mathbf{i} \in 1,\ldots,4$. Correspondingly, the error is bounded by $\frac{\sqrt{3}}{2}\Delta$ in the 3D case.*

Fig. 4 (b) visualizes in a similar way an extreme case where $\mathbf{S}$ lies exactly on top of $\mathbf{P}$. Here, the error bound given in Property 1 ($\sqrt{3}\Delta$ in the 3D case) is sharp.

4.2. Error in Push-down

In principle, the above observation applies for the push-down step as well as for the pull-up step in our algorithm.

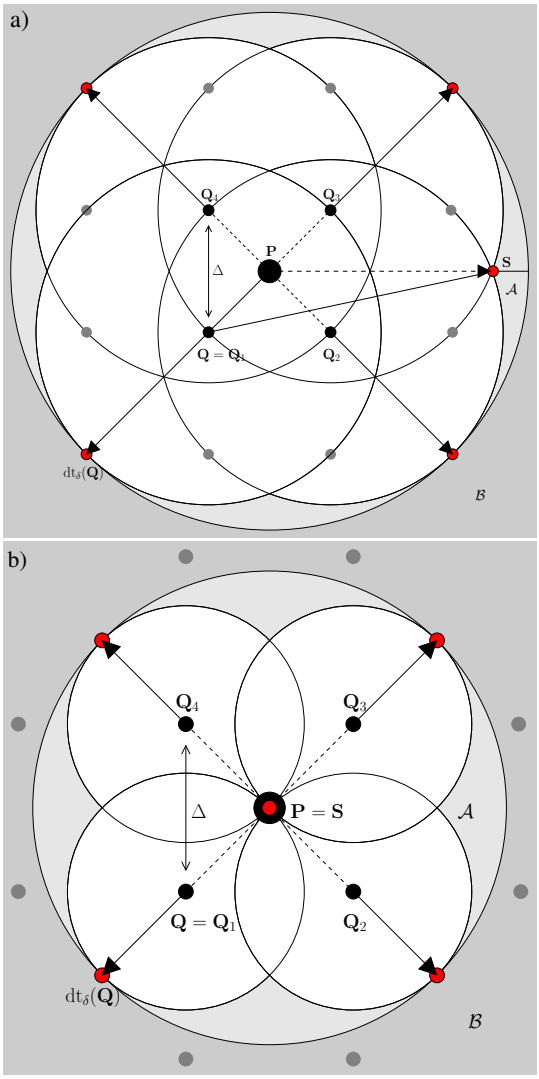

Figure 4: *Push-down (in 2D): Worst case scenario for exterior* $\mathbf{S}$ *(a) – worst case where* $\mathbf{S}$ *lies within the square defined by* $\mathbf{Q}_1, \ldots \mathbf{Q}_4$ *(b)*

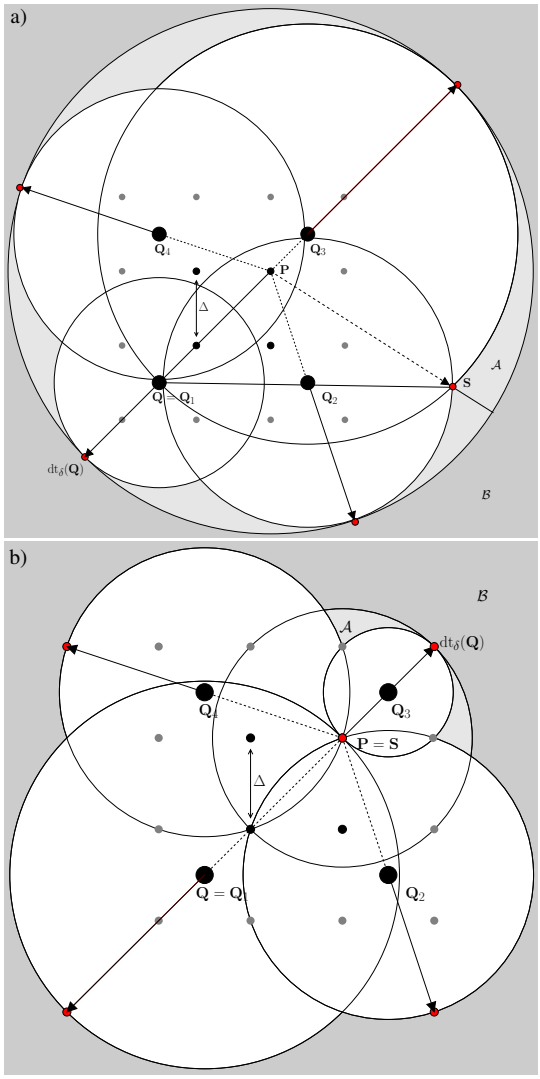

Figure 5: *Pull-up (in 2D): Worst case for exterior references (a) – degenerated case (b)*

However, only references close to a thin line of voxels (with distance $\leq \Delta$) are pushed down to the coarser level. In addition, all references that will be stored in the coarser grid are contained within the fine grid, which itself represents an accurate distance transform. This excludes the extreme case shown in Fig. 4 (b). Typically, the error is much smaller than $\sqrt{3}\Delta$ in the push-down step, more precisely, bounded by $\frac{\sqrt{3}}{2}\Delta$ (see Remark 1).

4.3. Error in Pull-up

Compared to the push-down step, the surrounding voxels pulled up to the finer level are not equally far from the currently considered point $\mathbf{P}$ (see Fig. 5). Nevertheless, Property 1 is satisfied because of the fact that the distance between $\mathbf{P}$ and the next surrounding point is $\frac{\sqrt{3}}{2}\Delta$ as in the push-down case.

Similar situations as in Fig. 4 can be constructed for the pull-up step. Fig. 5 (a) shows a worst case situation where the missed reference lies outside the sample square. Fig. 5 (b) is an extreme situation where Property 1 is a sharp bound.

Altogether, we have shown that the error occurring in one step (push-down or pull-up) in the algorithm is bounded by $\sqrt{3}$ times the spacing w.r.t. the finer grid. Typically, this error will be corrected by the additional propagation step we do after each pull-up step in our algorithm.

Cases where the error may not be canceled appear when a wrong reference points in a completely incorrect direction. Still, this effect can also be compensated by performing more than only one additional propagation step and the reference tracking described in Sec. 3.2. See Sec. 6.2 for measurements of the error in our test examples.

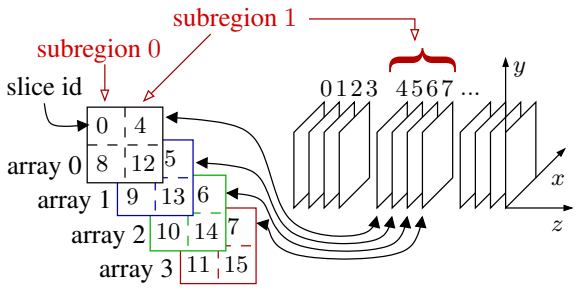

Figure 6: *Referencing of slices in four larger 2D textures using sub-regions*

5. Implementation Details

This section is devoted to technical details concerning the GPU realization of the algorithm. The implementation is based on OpenGL as graphics API and uses GLSL as GPU programming language.

5.1. 3D Grid Processing on the GPU

Our system relies heavily on the ability to dynamically update 3D grids on the GPU using fragment programs. Current graphics hardware does not provide 3D render-to-texture functionality, thus we cannot store 3D textures explicitly. We use a technique we presented in [KC05] which represents a grid as a stack of 2D *slices* and offers fast 3D texture processing.

In order to maximize the number of voxels processed in one pass, so-called *multiple render targets*, supported by the `GL_ARB_draw_buffers`-extension, are used to handle up to four slices in parallel. Additionally, each of the four render targets is partitioned in sub-regions, which represent the slices. Thus, four textures can be used to represent the 3D grid, theoretically allowing a maximal resolution of 256^3 because of the 2048-limit for 2D textures on current graphics hardware. Since the MRT mechanism allows only computation at the same output position for all four arrays, the slice reference to the sub-regions in the four render targets in an alternating manner in order to process four adjacent slices in parallel (see Figure 6).

5.2. GPU Programming Aspects

The distance transform is represented by a structure as described in Sec. 5.1. Frame buffer objects (supported by `GL_EXT_framebuffer_object`) are used to store the render targets. A quad exactly fitting into the output buffer is rendered and rasterized in order to write directly into the structure using a fragment program. This is a commonly used approach to perform general purpose computations on the GPU (see [OLG*05]). Because input and output must be different in this approach, voxel processing is done by binding a double buffered texture containing the data of the previous pass. The levels of the hierarchy are stored in separate textures. Note that super-sampling as in push-down and pull-up greatly takes advantage of the caching capability of the GPU, because adjacent texels are sampled.

Super-sampling is done by checking samples for the closest reference point. The same operation is used for propagation of the distance information. The following listing shows how this operation is implemented in GLSL. The parameters `position` and `center` contain the information related to the output voxel, `sample` is the sample we want to compare:

```
01 vec4 minDistance(const vec3 position,
02                  const vec4 center,
03                  const vec4 sample)
04 {
05   float val_pos = distance(position,
06                            sample.gba);
07
08   if (val_pos < abs(center.r))
09     return vec4(sign(center.r)*val_pos,
10                 sample.gba);
11   else
12     return center;
13 }
```

We use textures storing 4 floating point components. In order to obtain precise distances and best possible framerates, we choose `GL_RGBA_FLOAT16_ATI` as internal texture format. The first component stores the (signed) distance, the remaining components contain the coordinates of the closest point to the object boundary.

The run-time analysis of the fragment programs shows that the push-down and the pull-up are texture-fetch-bounded. On the other hand, the distance propagation using a $3 \times 3 \times 3$ structure element is computation-bound.

6. Results

The presented approach has been implemented and tested using implicit geometries stored as voxel data. Figure 7 shows the computed distance field for a notched sphere using our hierarchical approach. Figure 8 and 9 show a more detailed example obtained by joining and intersecting several implicit geometries.

6.1. Performance

We use a GeForce 7600 GT as graphics card for our performance tests. The fragment programs in our implementation leave some room for improvements. Still, in the notched sphere example we have a frame rate of 30 FPS ($\approx$ 33ms) in a grid with resolution 64^3 when using four additional hierarchy levels and performing one additional propagation step for each level ($k_\mathrm{p} = 1$). For compar-

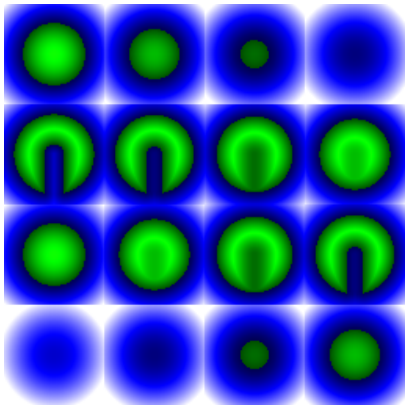

Figure 7: *Notched Sphere: Object used in our testing framework – the volume has resolution* 64^3*. Note that only one of four render target is shown, thus we see* 16 *grid slices. Negative distances are shown in green/yellow, positive in blue/white, both scaled and biased for better visual output.*

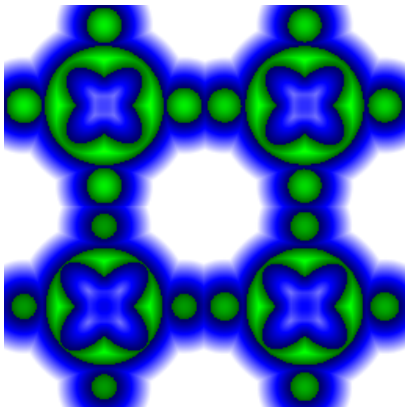

Figure 8: *Detailed Object: Four slices taken from our second example in a* 128^3 *volume.*

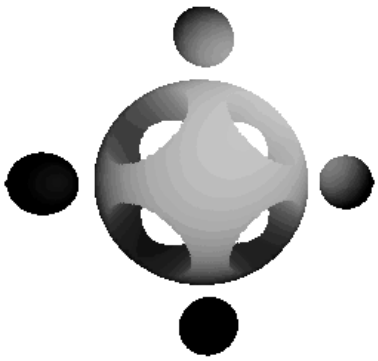

Figure 9: *The detailed object in 3D*

ison, the frame-rate falls to approximately 3 FPS when propagation is used without hierarchy (until all voxels are reached). Thus, the hierarchical structure yields a significant performance benefit.

Taking a 128^3 volume instead of 64^3 with the hierarchy approach leads to 4 FPS.

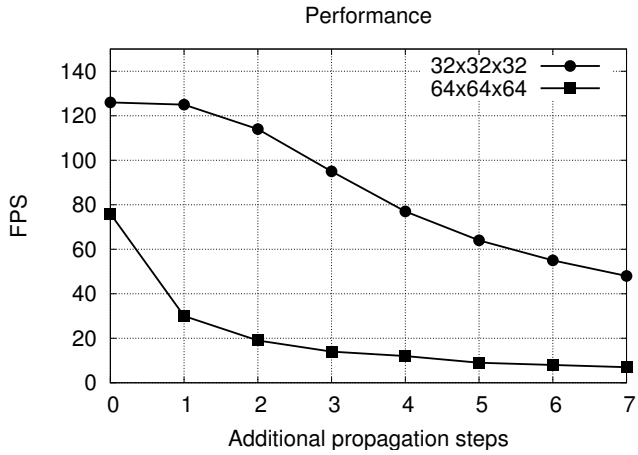

Figure 10: *Performance for the notched-sphere example in a* 32^3 *and a* 64^3 *volume using* 3 *additional hierarchy levels*

Fig. 10 shows the performance depending on k_p. Notice the typical non-linear behavior due to optimizations performed by the GPU architecture.

Although the propagation step is very well-suited for parallelization on the GPU, this step is still a bottle neck if applied on the finest level in the hierarchy. Thus, it is important to choose a compromise between the resulting error and the loss of interactivity. A second bottle neck in the current implementation is the pull-up step described in Sec. 3.3. The overall performance is essentially bounded by the number of texture-fetches.

6.2. Error

The same configuration as in Sec. 6.1 produces an average of the relative error of 0.000247059 in a 64^3 volume and 0.000021961 in 128^3. The correct distance transform is computed using a (slow) brute force method which checks all pairs of voxels.

Fig. 11 shows how the error evolves depending on k_p in a 32^3 volume. The error greatly decreases for both geometries already for $k_\mathrm{p} = 1$. It nearly disappears for a larger k_p.

Fig. 12 demonstrates the usefulness of additional propagation steps. As one can see, switching from $k_\mathrm{p} = 0$ to $k_\mathrm{p} = 1$ significantly improves the accuracy of the references.

7. Conclusion

A method for computing 3D distance transforms in real-time has been presented. The algorithm computing the distance transform is organized hierarchically. We provide a bound for the resulting distance error, which, in

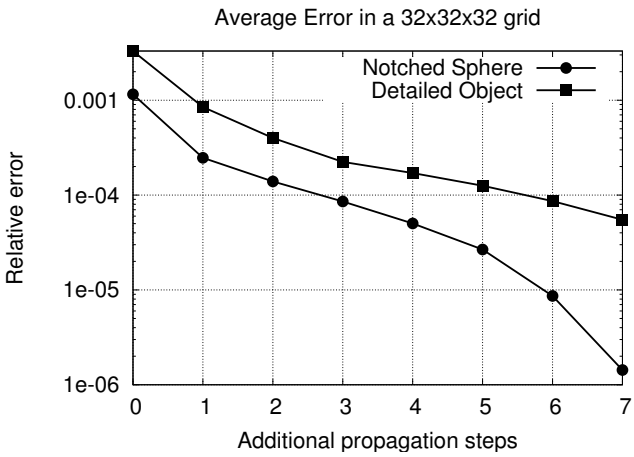

Figure 11: *Average Error for both examples in a* 32^3 *volume using* 3 *additional hierarchy levels*

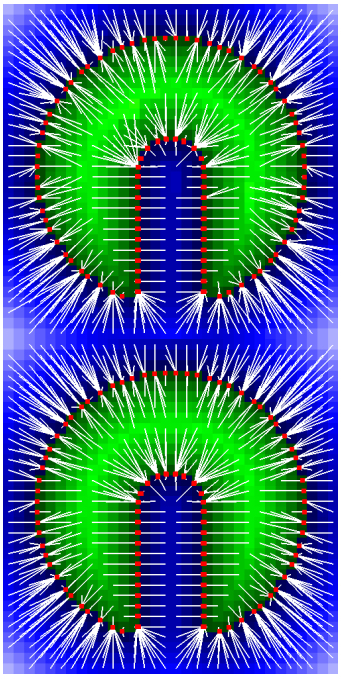

Figure 12: *References computed by the hierarchical method, without additional propagation step (top image), with one additional propagation step (bottom image) – references to other slices are left out.*

our experiments, is thinned out without a large impact in the performance by means of k_p additional propagation steps. Thus, an error estimation depending on k_p would be an interesting task for further investigations.

References

CM99. CUISENAIRE O., MACQ B.: Fast euclidean distance transformation by propagation using multiple neighborhoods. In *Computer Vision and Image Understanding, Vol. 76, No. 2* (November 1999), pp. 163–172.

Cui99. CUISENAIRE O.: Distance transformations: Fast algorithms and applications to medical image processing. *Ph.D. Thesis, UCL, Louvain-la-Neuve, Belgium* (October 1999).

HKL*99. HOFF K., KEYSER J., LIN M., MANOCHA D., CULVER T.: Fast computation of generalized Voronoi diagrams using graphics hardware. *ACM Proceedings SIGGRAPH* (1999), 277–286.

HSS*05. HADWIGER M., SIGG C., SCHARSACH H., BÜHLER K., GROSS M.: Real-time ray-casting and advanced shading of discrete isosurfaces. In *Proc. EUROGRAPHICS* (2005), pp. 303–312.

KC05. KOLB A., CUNTZ N.: Dynamic particle coupling for GPU-based fluid simulation. In *Proc. 18th Symposium on Simulation Technique, ISBN 3-936150-41-9* (Sep. 2005), pp. 722–727.

KLRS04. KOLB A., LATTA L., REZK-SALAMA C.: Hardware-based simulation and collision detection for large particle systems. In *Proc. Graphics Hardware* (2004), pp. 123–131.

OLG*05. OWENS J., LUEBKE D., GOVINDARAJU N., HARRIS M., KRÜGER J., LEFOHN A., PURCELL T.: A survey of general-purpose computation on graphics hardware. In *Proc. EUROGRAPHICS, State of the Art Reports* (2005), pp. 21–51.

RT06. RONG G., TAN T.-S.: Jump flooding in gpu with applications to vornoi diagram and distance transform. In *ACM Symposium on Interactive 3D Graphics and Games, 14–17 March, Redwood City* (2006), pp. 109–116.

SGGM06. SUD A., GOVINDARAJU N., GAYLE R., MANOCHA D.: Interactive 3d distance field computation using linear factorization. In *Proc. Symp. on Interactive 3D graphics & games* (2006), pp. 117–124.

SPG03. SIGG C., PEIKERT R., GROSS M.: Signed distance transform using graphics hardware. In *Proc. IEEE Conf. on Visualization* (2003).

ST04. STRZODKA R., TELEA A.: Generalized distance transforms and skeletons in graphics hardware. In *VisSym* (2004), pp. 221–230.

Tsi95. TSITSIKLIS J. N.: Efficient algorithms for globally optimal trajectories. In *IEEE Trans. on Automatic Control* (1995), pp. 1528–1538.