Analytic Isosurface Rendering and Maximum Intensity Projection on the GPU

Péter Józsa, Márton József Tóth, Balázs Csébfalvi
Budapest University of Technology and Economics
Department of Control Engineering and Information Technology,
Magyar tudósok krt. 2, H-1117, Budapest, Hungary
cseb@iit.bme.hu

ABSTRACT

It is well known that isosurfaces implicitly represented by volumetric data can be analytically rendered if a trilinear interpolation is assumed to be applied for the continuous reconstruction. However, to the best of our knowledge, it has not been investigated yet how this approach can be efficiently implemented on current GPUs and how much the analytic intersection point calculations slow down the rendering process compared to the traditional discrete approximation. In this paper, we propose a GPU friendly first-hit ray-casting algorithm that (1) minimizes the number of texture fetches, (2) significantly simplifies the arithmetic operations, and (3) avoids error accumulation during the ray traversal. We show that our analytic isosurface rendering optimized for the GPU is even faster than an equidistant discrete sampling, if the sampling frequency is set such that a comparable image quality is obtained. This is true even if the empty blocks of voxels are not processed along the rays. Therefore, the analytic approach can completely replace the traditional first-hit ray-casting implementations. Additionally, we show that the core of our algorithm can also be used for analytic Maximum Intensity Projection (MIP).

Keywords

First-hit ray casting, maximum intensity projection, reconstruction filtering, GPU acceleration.

1 INTRODUCTION

There are two fundamentally different approaches for rendering isosurfaces contained in volumetric data. The first one is indirect volume rendering by using the classical Marching Cubes (MC) algorithm [LC87], where the isosurfaces are extracted from the volumetric representation in a form of a triangular mesh. The other one is direct volume rendering [Lev88], where along each viewing ray just the first intersection point is determined [PSL+98], for which the interpolated density value is the same as the threshold that defines the isosurface. The drawbacks of the MC algorithm are the following:

- If the initial volumetric data is of high resolution, the MC algorithm produces a huge number of triangles.
- The computation of the triangular mesh is relatively expensive, and it has to be repeated whenever the user changes the isosurface threshold.

- The resulting geometric model is not smooth enough as it is a piecewise linear representation of the isosurface. The edges between the triangles become apparent if we zoom into the details.

To reduce the complexity of the mesh, triangle decimation algorithms [SZL92] were proposed, while for a real-time recalculation of the mesh, fast GPU implementations can be used exploiting the geometry shader [Gei08]. The roughness of the model, however, is more difficult to handle. Higher-order surface-extraction algorithms [The02, PVS+11] result in smoother isosurfaces, but their preprocessing and rendering costs are significantly higher. Although there exist constrained surface-fairing algorithms [Nie04] that iteratively modify the positions of the vertices to minimize the global curvature, the geometric model still remains piecewise linear. In contrast, direct isosurface-rendering techniques [PSL+98] provide visually more pleasing images than the MC algorithm, since in each cubic cell, they reproduce a level set of a cubic polynomial that is obtained by the trilinear interpolation. Direct isosurface rendering can be implemented on current GPUs by using either texture slicing [WE98] or ray casting [KW03]. While the former fits better to the architecture of the GPU, the latter is more flexible for using more sophisticated shading models [HLSR09]. In both cases, the image quality depends very much on the sampling frequency, that is, the number of texture fetches. On

the other hand, the rendering performance is inversely proportional to the number of texture fetches. Therefore, a given sampling frequency always determines a trade-off between image quality and rendering speed. To guarantee the best possible image quality for rendering the isosurfaces of trilinearly reconstructed volumes, an analytic calculation of the intersection points was proposed on a parallel CPU architecture [PSL+98]. This approach has not been adapted to the GPU so far, probably because of the following reasons:

- The cubic polynomial ray profiles inside a given cell are analytically determined from the eight corner voxels. Using a direct GPU implementation, this would require eight nearest-neighbor texture fetches. However, for the same cost, eight trilinear samples along the given ray segment can be evaluated, which might already provide sufficient image quality.
- The analytic intersection point calculation requires the roots of a cubic polynomial. The root finding is computationally expensive and has to handle several special cases, which is not GPU friendly.

In this paper, we show that despite these counterarguments, an analytic first-hit ray casting can be efficiently implemented on current GPUs after a thorough revision of the original CPU-based algorithm [PSL+98]. The contributions of our work are the following:

- It is known that the cubic polynomial ray profiles inside the intersected cells can be determined by taking only three trilinear samples per cell in average [AWC10], but this approach is first used in this paper for analytic isosurface rendering. Furthermore, we show that the roots of a cubic polynomial has to be determined for only one cell along each ray. For the other intersected cells, just the solution of a second-degree equation is needed, which is easy and efficient to obtain in the well-known closed form.
- As an analytic isosurface rendering requires precise intersection points between the faces of the cells and the rays, the accuracy of the applied ray-traversal algorithm is of crucial importance. We show that the previously applied incremental ray-traversal techniques [AW87] produce substantial accumulated error on the GPU. Therefore, we propose an integer-based ray traversal that completely avoids the error accumulation.
- We thoroughly compare our analytic first-hit ray-casting implementation to the traditional discrete approximation. We demonstrate that the traditional discrete sampling can guarantee approximately the same image quality only at the cost of a drastic oversampling. Therefore, it is slower than our analytic solution.
- We show that the core of our first-hit ray-casting implementation can also be used for analytic MIP rendering, which produces much higher image quality than the classical discrete approximation if the sampling frequency is set such that the rendering times are nearly the same.

2 RELATED WORK

An analytic ray profile evaluation [SGS95] was first proposed for MIP rendering [HMS95, SSN+98, MKG00, KJ08]. This approach was then adapted to first-hit ray casting [LC96, PSL+98]. All these methods were implemented on CPU architectures. Therefore, they calculate the analytic ray profile inside a cubic cell from its eight corner voxels. However, the direct implementation of this kind of evaluation on the GPU is inefficient, since the costly texture fetches reduce the performance. Recently, it has been shown that the analytic ray profiles inside the cubic cells can be determined from four trilinear samples [AWC10] as well, but this technique was proposed for a completely different purpose, namely, to obtain the extrema of the ray profiles for a precise preintegrated volume rendering [EKE01]. In this paper, we apply a similar approach, but for fast first-hit ray casting, and show that it is more efficient than the traditional discrete sampling, while it provides the best possible image quality if a trilinear interpolation is assumed. Although an optimized and numerically stable method has already been published [MKWF04] to analytically find the first intersection point inside a cell, to the best of our knowledge, we are the first to recognize that the computationally expensive root finding in the cubic polynomial ray profiles is not necessary for each single cell intersected by the given ray, and it has to be performed at most only once inside that particular cell, where the ray does hit the isosurface. Therefore, our major goal is to efficiently find this cell for each ray rather than to optimize the root finding [MKWF04]. Our algorithm is novel also regarding the applied ray-traversal technique. An analytic ray profile evaluation requires accurate entry and exit points for each cell. For this purpose, usually incremental algorithms [AW87, CW88] are applied. Nevertheless, we show that due to the floating-point state variables, these algorithms result in significant accumulated error on the GPU, which contradicts to the goal of the analytic ray profile evaluation. Therefore, we propose an improved ray-traversal algorithm that is based on integer state variables, and as such does not introduce an accumulated error. Although there exist similar integer-based ray-traversal methods [LZY04, LSZ08],

they determine only the intersected voxels without calculating the corresponding entry and exit points.

3 ANALYTIC FIRST-HIT RAY CASTING ON THE GPU

Our method processes the intersected cubic cells along each ray in front-to-back order. Inside each cell it is checked whether the ray hits the isosurface. This is done by analytically evaluating the extrema of the ray profile inside the given cell, which requires just the solution of a second-degree equation. If the isosurface threshold is between the minimum and the maximum of the ray profile, there must be an intersection point in the given cell, which is analytically determined by solving a third-degree equation. If the isosurface threshold is lower than the minimum of the ray profile or it is greater than the maximum of the ray profile then there is no intersection point in the current cell, so the next intersected cell needs to be processed. Overall, a computationally expensive solution of a third-degree equation is necessary at most only once for each ray, which is of negligible cost compared to the processing of all those cells, where the ray does not intersect the isosurface.

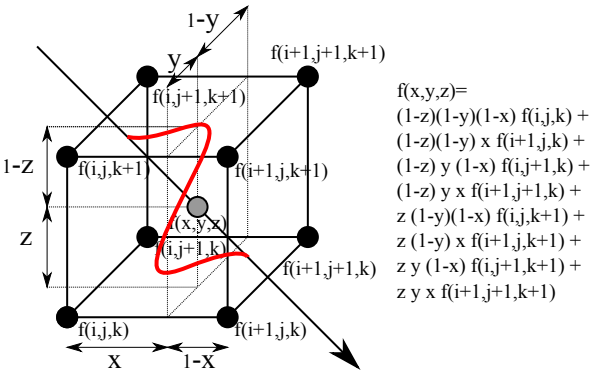


Figure 1: Trilinear interpolation.

4 TRILINEAR INTERPOLATION

Our algorithm assumes that the density function inside the cubic cells is reconstructed by a trilinear interpolation (see Figure 1). Although higher-order filters, such as the tricubic B-spline or the Catmull-Rom spline [MMK+98], provide higher image quality than the trilinear filter, they are an order of magnitude slower to evaluate on current GPUs even if the hardwired trilinear texture fetching is utilized in their implementation [SH05]. This is the major reason why the trilinear interpolation is still a de facto standard in the practice of interactive volume visualization.

5 ANALYTIC RAY PROFILES

It is easy to see that the trilinear interpolation results in piecewise cubic ray profiles. The parametric equation of a ray is expressed as follows:

$$\mathbf{r} = \mathbf{r}_0 + \mathbf{d} \cdot t, \tag{1}$$

where $r = [x,y,z]^T$ is an arbitrary point along the ray, $\mathbf{r}_0 = [x_0,y_0,z_0]^T$ is the starting position, $d = [dx,dy,dz]^T$ is the direction of the ray, and t is the ray parameter. Substituting Equation 1 into the trilinear interpolation formula (Figure 1), we obtain the ray profile inside a cubic cell as a cubic polynomial of the ray parameter t:

$$f(t) = at^3 + bt^2 + ct + d. \tag{2}$$

Sakas et al. [SGS95] proposed to determine the polynomial coefficients a, b, c, and d from the densities of the eight corner voxels. This evaluation scheme has originally been proposed for a CPU implementation [SGS95, LC96, PSL+98]. On the GPU, however, it is rather inefficient because of two reasons. First, it requires too many arithmetic operations. Second, it relies on eight nearest-neighbor texture fetches, which take approximately the same time as the evaluation of eight equidistant trilinear samples along the ray segment that is intersected by the given cell [SH05]. Generally, such a drastic oversampling already guarantees a sufficiently high image quality. Therefore, a direct GPU implementation of the method by Sakas et al. [SGS95] does not pay off, since for the same computational cost it is not expected to provide significantly higher quality than the traditional GPU implementation. Instead, we use a much simpler GPU-friendly evaluation scheme that well exploits the hardwired trilinear texture-fetching functionality. Our goal is to avoid all the nearest-neighbor texture fetches, and to determine the polynomial coefficients a, b, c, and d from the minimal number of trilinear samples. Since $f(t)$ is a cubic polynomial, it is unambiguously defined by its values at four different parameters t_1, t_2, t_3, t_4. Additionally, $f(t)$ is defined by a trilinear interpolation in a position that corresponds to parameter t. Therefore, we have to find four different positions along the ray segment that is intersected by the given cubic cell. Without a loss of generality, let us assume that $t = 0$ and $t = 1$ belong to the entry and exit points, respectively. In between, we apply a uniform subdivision, which results in four different parameters: $t_1 = 0$, $t_2 = 1/3$, $t_3 = 2/3$, $t_4 = 1$. The corresponding values of $f(t)$ are

$$f_1 = f(0) = d, \tag{3}$$

$$f_2 = f\left(\frac{1}{3}\right) = \frac{1}{27}a + \frac{1}{9}b + \frac{1}{3}c + d,$$

$$f_3 = f\left(\frac{2}{3}\right) = \frac{8}{27}a + \frac{4}{9}b + \frac{2}{3}c + d,$$

$$f_4 = f(1) = a+b+c+d.$$

From these four equations, the four unknown coefficients a, b, c, and d can be determined as follows:

$$a = -\frac{9}{2}f_1 + \frac{27}{2}f_2 - \frac{27}{2}f_3 + \frac{9}{2}f_4, \tag{4}$$

$$b = 9f_1 - \frac{45}{2}f_2 + 18f_3 - \frac{9}{2}f_4,$$

$$c = -\frac{11}{2}f_1 + 9f_2 - \frac{9}{2}f_3 + f_4,$$

$$d = f_1.$$

Thus, inside a cubic cell, an analytic ray profile is derived from four trilinear samples f_1, f_2, f_3, and f_4. In a GPU implementation, this is approximately twice as fast as the brute-force approach that derives the polynomial coefficients from eight nearest-neighbor samples [SGS95]. The derivation of the cubic polynomial ray profile from four trilinear samples is not new, since it was previously proposed by Ament et al. [AWC10]. However, it was not their goal to use this analytical form for first-hit ray casting. Instead, they produced a piecewise linear representation that has the same local extrema as the cubic polynomial ray profile. The obtained piecewise linear ray profiles were then used for preintegrated direct volume rendering [EKE01]. Although this approach leads to a more general ray-casting implementation that can also be used for isosurface rendering, the intersection points between the rays and the isosurface are still not determined analytically.

6 EXTREMA OF THE RAY PROFILES

The ray profile can take its local extrema at those values of parameter t, where its derivative is equal to zero:

$$f'(t) = 3at^2 + 2bt + c = 0. \tag{5}$$

Since this is a second-degree equation, the two roots can be easily determined as

$$\frac{-b \pm \sqrt{b^2 - 3ac}}{3a}. \tag{6}$$

If a root is greater than zero and less than one (the corresponding position is inside the given cubic cell), it is substituted into $f(t)$ to obtain a potential extremum. Additionally, the ray profile can also take its extrema at the entry and exit points. Therefore, f_1 and f_4 can also be potential maxima or minima. On the whole, the maximum and the minimum of four candidates have to be taken for each cell intersected by the given ray. Note that the entry point of a cubic cell is the same as the exit point of the previous cell. Therefore, only three trilinear samples need to be evaluated for each additional intersected cell. On current GPUs the bottleneck is the texture fetching, so the cost of the arithmetic operations necessary for the root finding is negligible compared to that of the three texture lookups.

7 INTEGER-BASED RAY TRAVERSAL

An analytic reconstruction of the ray profile within a cell requires the precise locations of the points, where the ray enters and leaves the cell. This means that during the ray traversal, the intersections of the ray with the planes separating the neighboring cells need to be calculated. An important characteristic of the intersection points is that one coordinate is always an integer, while the other two coordinates are typically fractional values. Many efficient algorithms have been developed for ray traversal through a voxel grid, but either they only determine the intersected voxels and do not explicitly calculate the entry/exit points [LZY04, LSZ08], or they employ incrementally updated floating-point variables [AW87, CW88]. Incremental algorithms implemented with floating-point arithmetic are prone to the accumulation of rounding errors, which eventually leads to traversing the wrong cells. This is especially a crucial problem in a GPU implementation, where double-precision floating-point arithmetic is not universally supported yet. Therefore, we propose a new ray-traversal algorithm for 3D voxel grids, which provides better accuracy than existing algorithms, while still remaining computationally efficient. The proposed algorithm avoids error accumulation during ray traversal and is particularly well-suited for GPU implementations.

The basic idea behind our algorithm is similar to that of the classic DDA algorithms [AW87, CW88]. During ray traversal, we keep track of the current position and the possible candidates for the next ray/cell intersection. In each step, the closest intersection is selected and the state variables are updated accordingly. Instead of tracking the distances to the next intersections along the ray, the integer coordinates of the intersections are tracked and the relative distances are derived from these integers in each step. In the following, we present in detail how our algorithm works in 2D. Due to its symmetrical nature, it can be trivially extended to 3D by adding the appropriate variables for the z-coordinate.

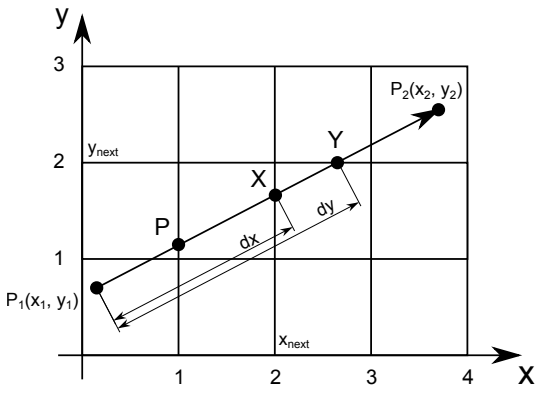


Figure 2: Integer-based ray traversal.

Figure 2 illustrates the operation of the algorithm. Let us denote the start and end points of the traversed ray segment by $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$, respectively. In the initialization phase, we calculate a couple of values that will be constant during the ray traversal. The differences between the endpoint coordinates are:

$$delta_x = x_2 - x_1,$$

$$delta_y = y_2 - y_1.$$

The length of the ray segment is:

$$length = |P_2 - P_1|.$$

The per-unit distance increments along the ray for each coordinate axis are:

$$deltaInv_x = length/delta_x,$$

$$deltaInv_y = length/delta_y.$$

If the ray is parallel to one of the coordinate axes then $delta_x$ or $delta_y$ is zero. In that case, the inverse value can be set to some arbitrarily large value. We also precalculate the integer coordinate increment per step for each axis, which is either 1 or -1 depending on the ray direction:

$$step_x = -1 \text{ if } delta_x < 0, 1 \text{ otherwise,}$$

$$step_y = -1 \text{ if } delta_y < 0, 1 \text{ otherwise.}$$

The algorithm proceeds as follows. Assume that the current position, where the ray intersects a grid line is at the point P in Figure 2. The intersections with the next vertical and horizontal grid lines are denoted by X and Y, respectively. The distance from P_1 to X can be calculated as

$$dx = (x_{next} - x_1) * deltaInv_x.$$

Similarly, the distance from P_1 to Y is

$$dy = (y_{next} - y_1) * deltaInv_y.$$

Comparing the distances, we can decide which intersection is closer to the starting point P_1 (and therefore, also to the current position P) and advance to that point along the ray. If $dx < dy$, as is the case in Figure 2, then the intersection with the vertical grid line is closer, so x_{next} is incremented by the value of $step_x$. If $dy < dx$ then y_{next} is incremented by the value of $step_y$. If $dx = dy$, that is, the ray passes through a cell corner, both x_{next} and y_{next} are incremented. The current position along the ray can be derived from the distance belonging to the current intersection (either dx or dy) and the initial ray parameters:

$$P = P_1 + dist * dir,$$

where dir is the normalized direction vector of the ray. Note that the explicit value of the current position is not used for finding the next intersection, so numerical inaccuracies in its computation do not affect the next step. The initial values of x_{next} and y_{next} are calculated as follows. First, the coordinates of the previous intersections of the ray with the grid are determined. This is calculated by rounding the starting position to the nearest integer coordinates in the direction opposite to the ray direction. This means that the coordinate is rounded down if the ray direction along the corresponding axis is positive, and rounded up if the direction is negative, yielding the values x_{prev} and y_{prev}. Then the per-step increment is added to them to obtain the initial values for the next intersections:

$$x_{prev} = ceil(x_1) \text{ if } delta_x < 0, floor(x_1) \text{ otherwise,}$$

$$y_{prev} = ceil(y_1) \text{ if } delta_y < 0, floor(y_1) \text{ otherwise,}$$

$$x_{next} = x_{prev} + step_x,$$

$$y_{next} = y_{prev} + step_y.$$

The only dynamic state variables that are required for the ray traversal are the grid coordinates of the next intersection for each axis. These are integer values, as well as the increment in each step. Consequently, our algorithm can be implemented without any dynamic floating-point state variable, thus avoiding the accumulation of the rounding errors. While being numerically much more stable, it is nearly as efficient computationally as the widely used Amanatides-Woo algorithm [AW87], as it requires only one additional floating-point multiplication per iteration step. Another advantage is that it can be implemented without any conditional branching, which is an important aspect of a fast GPU implementation. The pseudo code of the algorithm in 2D is shown in Listing 1.

```
void RayTraversal2D(int x1, int y1, int x2, int y2)
{
int iNextX, iNextY; int iStepX, iStepY;

int deltaX = x2 - x1, deltaY = y2 - y1;
float length =
  sqrt(deltaX * deltaX + deltaY * deltaY);
float dirX = deltaX / length;
float dirY = deltaY / length;
float deltaInvX = length / deltaX;
float deltaInvY = length / deltaY;
if (deltaX < 0) iStepX = -1; else iStepX = 1;
if (deltaY < 0) iStepY = -1; else iStepY = 1;
if (deltaX < 0) prevX = ceil(x1);
else prevX = floor(x1);
if (deltaY < 0) prevY = ceil(y1);
else prevY = floor(y1);
iNextX = prevX + iStepX; iNextY = prevY + iStepY;

float dist = 0.0;
do
  {
    float x = x1 + dist * dirX, y = y1 + dist * dirY;
    ProcessCell(x, y);
    float dx = (iNextX - x1) * deltaInvX;
    float dy = (iNextY - y1) * deltaInvY;
    float dist = min(dx, dy);
    if (dx == dist) iNextX = iNextX + iStepX;
```

```
    if (dy == dist) iNextY = iNextY + iStepY;
  } while (dist < length)
}
```

Listing 1: Pseudo code of our ray-traversal algorithm in 2D.

The 2D version is easy to extend to 3D by adding the appropriate variables for the z-coordinate. Note that within the traversal loop, only the `iNextX` and `iNextY` variables are updated incrementally by `iStepX` and `iStepY`, and these are defined as integers. The other variables are typically implemented in floating-point format, although depending on the application, some of them may be integers as well.

7.1 Implementation on the GPU

The GLSL shader code shown in Listing 2 demonstrates how the algorithm can be implemented by exploiting vector operations to eliminate any conditional branching.

```
void RayTraversal(vec3 startPos, vec3 endPos)
{
float   length; // length of the ray
vec3    ray;    // ray vector
vec3    dir;    // normalized ray direction
vec3    deltaInv; // distance increment per grid unit
ivec3   iStep;  // +/-1 for each axis
                // depending on the ray direction
ivec3   iNext;  // next plane intersection
                //in each direction
float   dist;   // current distance from
                //the starting point

ray = endPos - startPos;
length = length(ray);
dir = normalize(ray);
bvec3 isDirNegative = lessThan(dir, vec3(0));

// avoid division-by-zero
ray = mix(ray, vec3(1e-6),
  lessThan(abs(ray), vec3(1e-6)));
deltaInv = vec3(length) / ray;
iStep = ivec3(1) - ivec3(2) * ivec3(isDirNegative);

// round starting position to the
// previous grid intersection
vec3 prev = mix(floor(startPos),
  ceil(startPos), isDirNegative);
iNext = ivec3(prev) + iStep;

dist = 0.0;
do
  {
    // process cell at current position
    vec3 pos = fma(vec3(dist), dir, startPos);
    ProcessCell(pos);

    // calculate the distance to the next
    // intersection for each axis
    vec3 dNext =
      (vec3(iNext) - startPos) * deltaInv;

    // pick the closest intersection
    dist = min(min(dNext.x, dNext.y), dNext.z);

    // step along each axis that goes
    // through the intersection point
    bvec3 step =
      lessThanEqual(dNext, vec3(dist));
    iNext += iStep * ivec3(step);
  } while (dist < length);
}
```

Listing 2: GLSL code of our ray-traversal algorithm in 3D.

The function `ProcessCell()` is responsible for processing the cell that the ray enters at the current position. Note that some care must be taken for the calculation of `deltaInv` to avoid a possible division by zero. Also, we use the `lessThanEqual` comparison instead of `equal` for the sake of cautiousness.

7.2 Numerical Accuracy of the Ray Traversal

We have compared the accuracy of our algorithm to an optimized version of the Amanatides-Woo algorithm [AW87]. We implemented both methods in GLSL within the Voreen visualization framework.

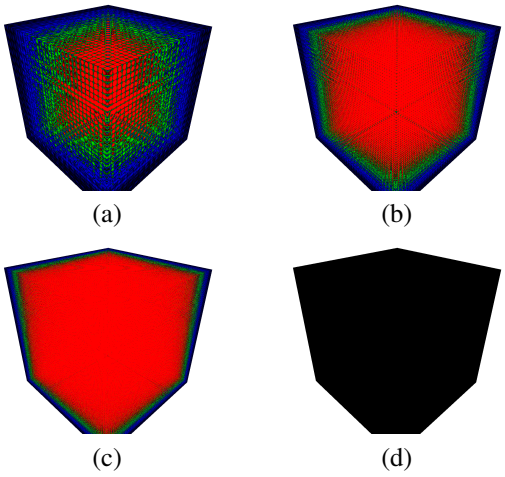


Figure 3: Visualization of the accumulated error for the Amanatides-Woo algorithm [AW87] ((a) - (c)), and for our integer-based ray-traversal algorithm (d). The resolutions of the grids are 16^3 (a), 32^3 (b), 64^3 (c), and 256^3 (d).

Figure 3 shows the visualization of the accumulated error. We calculated an error value in each step during the ray traversal, indicating the distance of the intersection position from the nearest grid plane. The absolute error values are summed along the ray. The corresponding pixel is colored according to the accumulated error. Ideally, the accumulated error should be zero, resulting in a completely black image. As we can see, this is the case for our integer-based algorithm. In contrast, the images corresponding to the floating-point-based Amanatides-Woo algorithm indicate that the accumulated error is already quite significant even for a low-resolution grid, and increases if the data resolution gets higher. A bright red color represents an accumulated error of 3.0 in grid units.

8 RESULTS

Figure 4 shows the results of our analytic first-hit ray casting for three different test data sets. For the same voxel/pixel ratio, an equidistant discrete sampling of

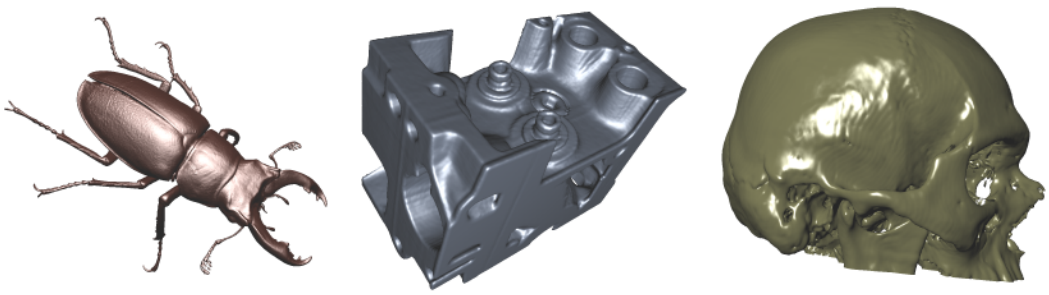


Figure 4: Images rendered by our analytic first-hit ray-casting algorithm.

the same computational cost provides similar image quality, but if we zoom into the details as demonstrated in Figure 5, the analytic approach is clearly superior. Measuring the running times (see Figure 5), we found that the bottleneck of the first-hit ray casting is indeed the texture fetching as it was expected. Note that an oversampling by a factor of four, which is already slower than our analytic evaluation, still results in visible sampling artifacts, while the analytic solution guarantees the best possible image quality for a trilinearly reconstructed volume. Although the trilinear interpolation kernel itself introduces postaliasing artifacts [ML94], at least the sampling artifacts are completely removed. The postaliasing artifacts could be suppressed by higher-order filters [MMK+98], for which the degree of the polynomial ray profiles is higher than three. In this case, however, the analytic root finding would be much more complicated and it could hardly be efficiently implemented on the GPU. Therefore, we think that our approach is a good compromise between rendering speed and image quality.

9 ADAPTATION TO MIP

Our first-hit ray-casting implementation is easy to adapt to MIP rendering. The core of the algorithm analytically determines the extrema of the cubic polynomial ray profiles in cells intersected by the given ray. Therefore, we have to visit all intersected cells and take the global maximum. Note that, for this purpose only second-degree equations need to be solved, and in average only three trilinear texture fetches per cell are necessary. Similarly to the first hit-ray casting, the analytic evaluation takes approximately the same time as a traditional discrete sampling taking three texture fetches per cell. Figure 6 shows the visual comparison of our analytic MIP evaluation to the traditional MIP rendering. The images were rendered practically at the same frame rate, since in case of traditional MIP, we set a sampling frequency that ensured nearly the same rendering speed as the analytic MIP evaluation. Note that our analytic MIP completely removes the sampling artifacts.

10 CONCLUSION

In this paper, we have shown that both isosurfaces and MIP images can be rendered interactively on current GPUs using an analytic evaluation. This approach results in the best possible image quality if a trilinear interpolation is assumed. Furthermore, the user does not have to specify the sampling rate as an additional parameter. We demonstrated that the analytic evaluation practically does not require a computational extra cost, since the traditional equidistant sampling is slower if the sampling frequency is set such that a comparable image quality is obtained. We have optimized our algorithm onto the GPU by reducing the number of texture fetches, simplifying the arithmetic operations, and avoiding the conditional branching. Moreover, to avoid the accumulation of the rounding errors, we developed a novel integer-based ray-traversal algorithm. Overall, according to our results, we believe that the analytic evaluation for both isosurface rendering and MIP can completely replace the traditional implementation that is based on a discrete approximation.

ACKNOWLEDGEMENTS

This work was supported by OTKA K-101527. We implemented our method using the Voreen framework (`voreen.uni-muenster.de`). The test data sets for MIP rendering were taken from the data base of the OsiriX DICOM Viewer (`www.osirix-viewer.com/datasets/`).

11 REFERENCES

[AW87] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *Proceedings of Eurographics*, pages 3–10, 1987.

[AWC10] Marco Ament, Daniel Weiskopf, and Hamish Carr. Direct interval volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1505–1514, 2010.

[CW88] John G. Cleary and Geoff Wyvill. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *The Visual Computer*, 4(2):65–83, 1988.

[EKE01] Klaus Engel, Martin Kraus, and Thomas Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 9–16, 2001.

[Gei08] Ryan Geiss. Generating complex procedural terrains using the GPU. In Hubert Nguyen, editor, *GPU Gems 3*, pages 7–37. Addison-Wesley, 2008.

[HLSR09] Markus Hadwiger, Patric Ljung, Christof Rezk Salama, and Timo Ropinski. Advanced illumination techniques for GPU-based volume raycasting. In *ACM SIGGRAPH Courses*, pages 2:1–2:166, 2009.

[HMS95] Wolfgang Heidrich, Michael McCool, and John Stevens. Interactive maximum projection volume rendering. In *Proceedings of the 6th Conference on Visualization*, pages 11–18, 1995.

[KJ08] Heewon Kye and DongKyun Jeong. Accelerated MIP based on GPU using block clipping and occlusion query. *Computers & Graphics*, 32(3):283–292, 2008.

[KW03] J. Kruger and R. Westermann. Acceleration techniques for GPU-based volume rendering. In *Proceedings of the 14th IEEE Visualization*, pages 287–292, 2003.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 163–169, 1987.

[LC96] Chyi-Cheng Lin and Yu-Tai Ching. An efficient volume-rendering algorithm with an analytic approach. *The Visual Computer*, 12(10):515–526, 1996.

[Lev88] Marc Levoy. Display of surfaces from volume data. *IEEE Computer Graphics Applications*, 8(3):29–37, 1988.

[LSZ08] Yong Kui Liu, H. Y. Song, and Borut Zalik. A general multi-step algorithm for voxel traversing along a line. *Computer Graphics Forum*, 27(1):73–80, 2008.

[LZY04] Yong Kui Liu, Borut Zalik, and H. Yang. An integer one-pass algorithm for voxel traversal. *Computer Graphics Forum*, 23(2):167–172, 2004.

[MKG00] L. Mroz, A. König, and E. Gröller. Maximum intensity projection at warp speed. *Computers & Graphics*, 24(3):343–352, 2000.

[MKWF04] Gerd Marmitt, Andreas Kleer, Ingo Wald, and Heiko Friedrich. Fast and accurate ray-voxel intersection techniques for iso-surface ray tracing. In *Proceedings of Vision, Modeling, and Visualization (VMV)*, pages 429–435, 2004.

[ML94] S. Marschner and R. Lobb. An evaluation of reconstruction filters for volume rendering. In *Proceedings of IEEE Visualization*, pages 100–107, 1994.

[MMK+98] T. Möller, K. Mueller, Y. Kurzion, R. Machiraju, and R. Yagel. Design of accurate and smooth filters for function and derivative reconstruction. In *Proceedings of IEEE Symposium on Volume Visualization*, pages 143–151, 1998.

[Nie04] Gregory M. Nielson. Dual marching cubes. In *IEEE Visualization*, pages 489–496, 2004.

[PSL+98] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter pike Sloan. Interactive ray tracing for isosurface rendering. In *Proceedings of IEEE Visualization*, pages 233–238, 1998.

[PVS+11] C. Pagot, J. Vollrath, F. Sadlo, D. Weiskopf, T. Ertl, and J. L. D. Comba. Interactive isocontouring of high-order surfaces. In *Dagstuhl Follow-Ups*, volume 2, pages 276–291, 2011.

[SGS95] G. Sakas, M. Grimm, and A. Savopoulos. Optimized maximum intensity projection (MIP). In *Proceedings of Rendering Techniques*, pages 51–63, 1995.

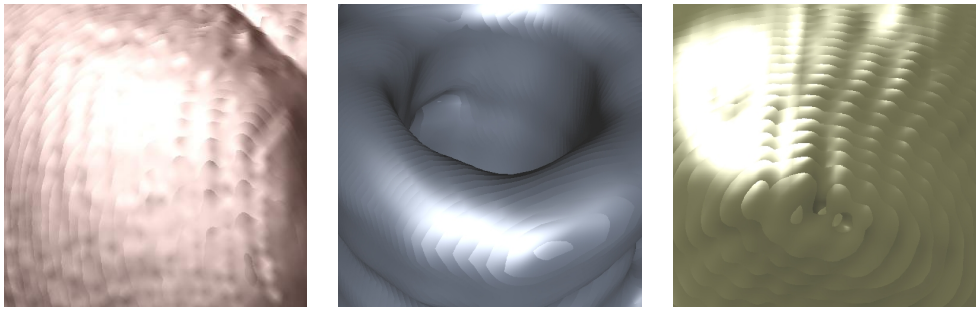
[SH05] C. Sigg and M. Hadwiger. Fast third-order texture filtering. In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 313–329. Matt Pharr (ed.), Addison-Wesley, 2005.

[SSN+98] Yoshinobu Sato, Yoshinobu Sato, Shin Nakajima, Nobuyuki Shiraga, Shinichi Tamura, and Ron Kikinis. Local maximum intensity projection (LMIP): A new rendering method for vascular visualization. *Computer Assisted Tomography*, 22(6):912–917, 1998.

[SZL92] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. In *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '92, pages 65–70, 1992.

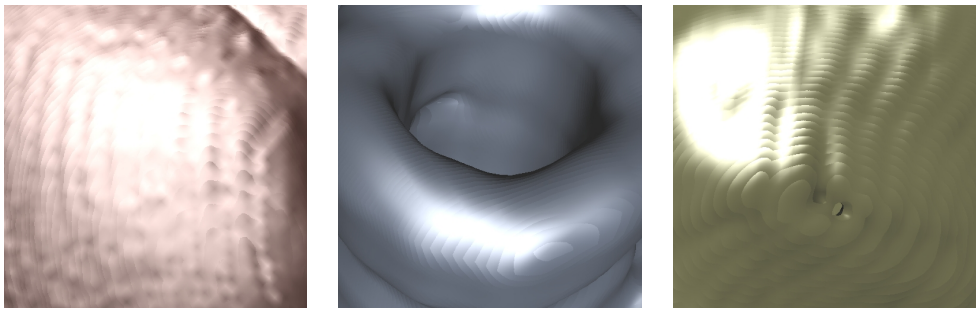
[The02] Holger Theisel. Exact isosurfaces for marching cubes. *Computer Graphics Forum*, 21(1):19–32, 2002.

[WE98] Rüdiger Westermann and Thomas Ertl. Efficiently using graphics hardware in volume rendering applications. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 169–177, 1998.



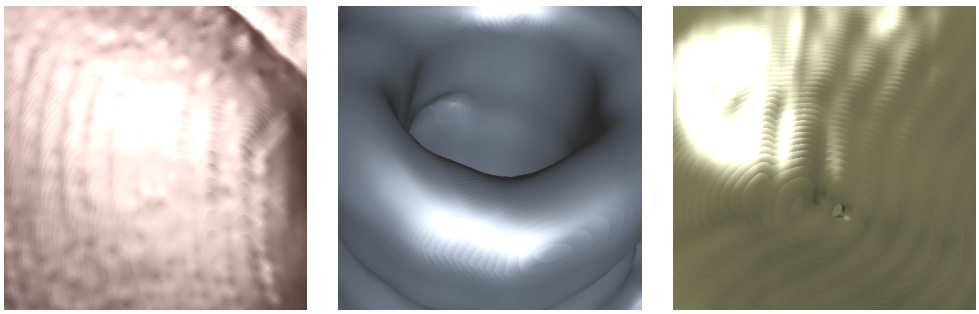
0.113 seconds 0.152 seconds 0.164 seconds

Traditional first-hit ray casting using an oversampling by a factor of two.



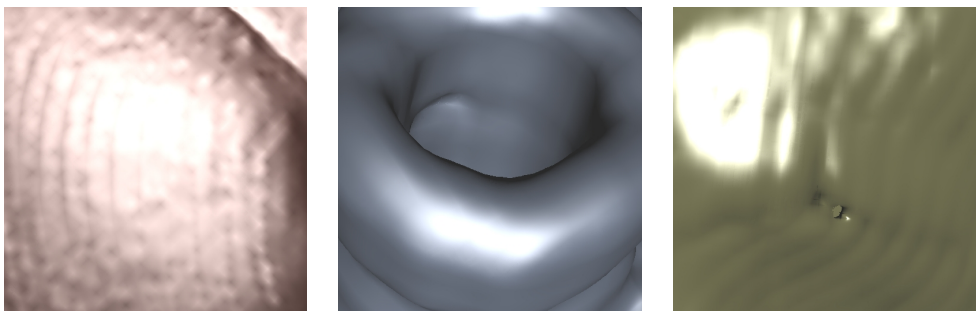
0.126 seconds 0.186 seconds 0.204 seconds

Traditional first-hit ray casting using an oversampling by a factor of three.



0.138 seconds 0.22 seconds 0.244 seconds

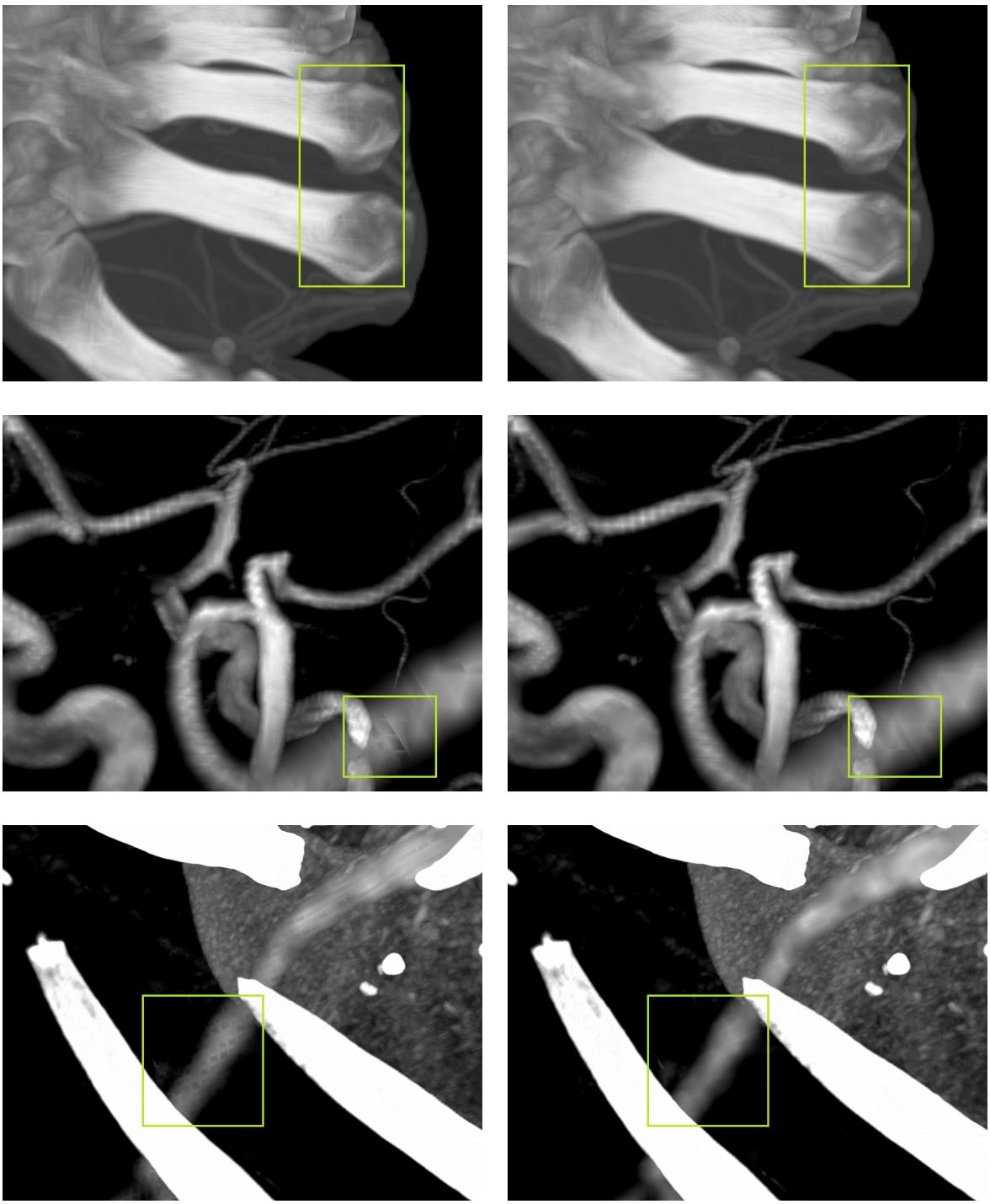
Traditional first-hit ray casting using an oversampling by a factor of four.



0.132 seconds 0.204 seconds 0.238 seconds

Our analytic first-hit ray casting.

Figure 5: Comparison of our analytic first-hit ray casting to traditional equidistant sampling in terms of visual quality. The rendering times were measured on an AMD Radeon HD5670 1GB graphics card.



Traditional MIP using discrete sampling. Analytic MIP.

Figure 6: Comparison of our analytic MIP evaluation to traditional MIP in terms of visual quality.