GPU Ray-Casting for Scalable Terrain Rendering

Christian Dick[1], Jens Krüger[2], and Rüdiger Westermann[1 †]

[1]Computer Graphics and Visualization Group, Technische Universität München, Germany
[2]Scientific Computing and Imaging Institute, University of Utah, USA

Abstract

With the ever increasing resolution of scanned elevation models, geometry throughput on the GPU is becoming a severe performance limitation in 3D terrain rendering. In this paper, we investigate GPU ray-casting as an alternative to overcome this limitation, and we demonstrate its advanced scalability compared to rasterization-based techniques. By integrating ray-casting into a tile-based GPU viewer that effectively reduces bandwidth requirements in out-of-core terrain visualization, we show that the rendering performance for large, high-resolution terrain fields can be increased significantly. We show that a screen-space error below one pixel permits piecewise constant interpolation of initial height samples. Furthermore, we exploit the texture mapping capabilities on recent GPUs to perform deferred anisotropic texture filtering, which allows for the rendering of digital elevation models and corresponding photo textures. In two key experiments we compare GPU-based ray-casting to a rasterization-based approach in the scope of terrain rendering, and we demonstrate the scalability of the proposed ray-caster with respect to display and data resolution.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Viewing Algorithms I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Texture, Raytracing, Virtual Reality

1. Introduction and Contribution

Due to the ever increasing size and resolution of scanned digital elevation models (DEMs) and corresponding photo textures, geo-spatial visualization systems are more and more facing the problem of dealing with TB data sets. Figure 1 shows such a gigantic model, which covers a 56 km $\times$ 85 km area of the Alps at a resolution of 1 m and 12.5 cm for the DEM and the photo texture, respectively. This amounts to over 860 GB of data, bearing the risk of severe bottlenecks both in data access and rendering.

To avoid these bottlenecks, a number of previous efforts have tackled the problem of bandwidth and rendering throughput limitations by using dedicated compression schemes [Ger01, LH04, GMC*06], efficient data management and streaming strategies [LP02,CGG*03,CKS03], and adaptive level-of-detail triangulations [LKR*96, DWS*97, LP01,LP02], to name just a few. Due to these advancements, it is now possible on commodity PCs to stream spatially extended high-resolution terrain fields to the GPU at rates allowing interactive rendering at reasonable quality. Nevertheless, as it has been shown recently by Dick et al. [DSW09], with increasing display resolution and a maximum geometric screen-space error below one pixel, geometry throughput on the GPU is becoming a severe performance limitation. For instance, the particular view shown in Figure 1 already requires rendering of about 30 million triangles on a 2 megapixel display.

To overcome this limitation, we present a GPU terrain rendering algorithm using ray-casting. Our method is similar to recent work by Tevs et al. [TIS08] in that it uses maximum mipmaps of the height field to speed up ray traversal on the GPU. Compared to this work, we propose a more efficient and numerically stable ray traversal scheme for the regular height field pyramid that is used as ray-casting acceleration structure. In addition, we have integrated the GPU ray-caster into a tile-based visually continuous terrain rendering method, which enables rendering from a LOD hierarchy with respect to a given screen-space error. Interestingly, we will show that a screen-space error below one pixel permits piecewise constant interpolation of initial height samples at no visual quality degradation. This allows us to avoid the expensive computation of ray intersection points with the bilinearly interpolated height field as proposed by Tevs and co-workers.

To support interactive rendering of TB data sets including scanned photo textures, we further present a novel deferred

† dick@in.tum.de, jens@sci.utah.edu, westerma@in.tum.de

© The Eurographics Association 2009.

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

Figure 1: *A textured DEM of Vorarlberg, Austria (56 km × 85 km) is rendered on a 1920 × 1080 view port using our method. The spatial resolution of the DEM and the texture is 1 m and 12.5 cm, respectively. Even though we render to a 2 megapixel view port, an average frame rate of about 30 fps is achieved at a geometric screen-space error of below one pixel.*

texturing method including full anisotropic texture filtering. For a ray's intersection point with the terrain height field, we compute the texture derivatives along the screen-space axes and let the GPU perform hardware-supported anisotropic texture sampling.

By means of our developments, we show that ray-casting can effectively reduce geometry load and per-fragment overdraw on the GPU, and therefore yields superior frame rates than rasterization-based approaches for high terrain resolutions. Since the maximum mipmap for a tile's height field is built on the GPU once a tile becomes visible, maximum mipmaps do not require any additional information to be stored. Especially in the current application, where memory access and bandwidth limitations are a major concern, maximum mipmaps thus have a clear advantage over alternative ray-casting acceleration methods like cone stepping [Dum06, PO07] or precomputed distance fields [Don05].

The remainder of this paper is organized as follows: In the next section, we outline previously published LOD methods and techniques for terrain rendering. Next, we describe our height field ray traversal algorithm, and show how it is extended to utilize the maximum mipmap acceleration structure. We then focus on the anisotropic texture filtering, and show how to exploit tile to tile occlusions to further improve the rendering performance. Next, we describe the integration of the new method into a large-data out-of-core terrain rendering engine. In Section 5, we compare our new ray-casting approach to a highly optimized rasterization rendering method. The paper is concluded with a discussion and remarks on future work.

2. Related Work

Previous work in the field of terrain rendering can roughly be categorized into mesh- and grid-based techniques. Most of the mesh-based methods use the rasterization approach to render the terrain and focus on efficient LOD schemes to reduce the per-frame workload. Most of the grid-based solutions utilize some sort of ray-casting or grid traversal approach to directly operate on the height field to generate the image.

2.1. Mesh-based Terrain Rendering

Over the last decade, a number of view-dependent LOD techniques for terrain rendering have been proposed, which differ mainly in the hierarchical structures used. Previous work can be classified into dynamic remeshing strategies, region-based multi-resolution approaches, and regular nested grids, all of which allow for visually continuous LOD rendering. For a thorough overview of the field let us refer here to the recent survey by Pajarola and Gobbetti [PG07].

2.2. Terrain Ray-Casting

Early terrain ray-casting implementations such as Musgrave [Mus88] were based on a combination of the digital difference analyzer (DDA) algorithm and traditional triangle/ray intersection ideas. Musgrave later presented the quasi-analytic error-bounded (QAEB) ray-tracing (formally published in the book Texturing and Modeling [EMP*02]) to render fractal terrains directly from their analytic description. For photo-realistic flight simulator applications, Cohen

© The Eurographics Association 2009.

et al. [CS93, CORLS96] proposed hierarchical ray-tracing in a pyramidal data structure to speed up the image generation. While Lee and Shin [LS95] presented more efficient grid traversal strategies, later, Henning and Stephenson [HS04] improved the performance of the traversal by replacing the cell-based method entirely by a run-based approach.

While CPU-based terrain ray-casting systems have a long history, only recently—with the advent of sophisticated GPU features—hardware-accelerated terrain ray-casting methods have been published. Qu et al. [QQZ*03] presented a GPU-based ray-casting system for elevation data but did not integrate sophisticated acceleration structures. Mantler and Jeschke [MJ06] focused on the efficient rendering of vegetation integrated into a GPU-based ray-casting approach. Their method like many other GPU-based techniques [OP05, PO06], however, only uses a fixed step size to trace over the height-field and thus may miss fine structures, producing only approximate results. To improve the approximation, nested intervals were used on the GPU, but still these methods were only approximations and were therefore primarily applied for secondary effects [Ohb03, Wym05] or in combination with precomputed safety radii [Dum06, PO07, Mic08, Don05, BD06, JMW07]. To utilize both the GPU as well as the CPU power Balciunas et al. [BDZ06] presented a hybrid model that, in a first pass, rasterizes a low resolution version of the terrain on the GPU and uses the depth information as starting point for a CPU-based ray-casting system; for a medium sized model they report timings of 3-10 fps. Most closely related to our approach are the recent contributions by Oh et al. [OKL06] and Tevs et al. [TIS08], both using a similar idea of the traversal of a maximum quadtree on the GPU, which is a simplified version of a maximum/minimum mipmap [GBP06, CHCH06].

In contrast to all of the previous approaches, our rendering system, however, is able to handle arbitrarily large data sets via a tiling mechanism together with a highly GPU optimized quadtree traversal scheme that outperforms any previous implementations, while still guaranteeing an exact ray/terrain intersection.

3. GPU-based Terrain Ray-Casting

Our terrain rendering technique works on a tile-based multi-resolution representation of the terrain model. Each tile consists of a height field of size $N \times N$ samples and an orthographic photo texture. In our current implementation, a tile size of $N = 512$ is used. For details, we refer the reader to Section 4. In every frame, the set of tiles representing the terrain at the current view is determined, and these tiles are rendered in front-to-back order using the ray-casting approach described in the following sections.

To render a tile, we cast rays of sight through the centers of those pixels which are covered by the tile. For each ray, we determine the first intersection point with the tile's height field. The tile's photo texture is then sampled at that location to obtain the color of the respective pixel.

3.1. Ray Traversal Algorithm

The ray-casting of a tile is performed in a single rendering pass and is initiated by rendering the back faces of the tile's bounding box, which generates a fragment for each pixel covered by the tile. Rendering back instead of front faces has the advantage that the case when the viewer position is inside the bounding box of the tile does not have to be treated in a special way [Mic08]. The ray-casting is done in the height field's local texel-space for the x- and y-coordinate, and in normalized world-space for the z-coordinate. The coordinates of the vertices of the bounding box are issued as per-vertex attributes, thus for each ray the exit point *BoxExit* from the bounding box is available in the fragment shader. Furthermore, the current eye point *Eye* is issued as a constant buffer variable.

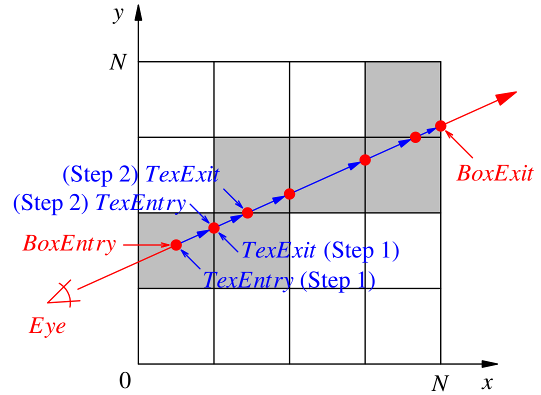

Figure 2: *The ray-casting is implemented by marching from texel to texel. In each step, the ray's exit point from the current texel is computed, which is equal to the entry point to the next texel along the ray and thus also yields the index of that texel. In the figure, the ray enters the tile's bounding box on the top and leaves on the right.*

The actual ray-casting is performed in the fragment shader. First, we compute the ray's direction *Dir* as $Dir = BoxExit - Eye$ and we determine the ray's entry point *BoxEntry* by intersecting the ray with the tile's bounding box. To simplify the computations and thus to speed up the ray-casting process, we only handle the case $Dir_x \geq 0 \wedge Dir_y \geq 0$. All other cases are reduced to this case by mirroring, e.g., if $Dir_x < 0$, we set $Dir_x \leftarrow -Dir_x$ and $BoxEntry_x \leftarrow N - BoxEntry_x$. Note that this requires to mirror the texture coordinates to access the tile's height field accordingly. To reduce the number of conditional branches and needed registers within the ray-casting loop, we test for the sign (≥ 0, < 0) of each of Dir_x, Dir_y, and Dir_z at the beginning, and replicate the ray-casting loop for each of the eight branches.

Starting from the bounding box entry point, we cast the ray until it hits the height field or leaves the domain of the tile. Let *TexEntry* denote the ray's entry point to the current height field texel. At the beginning, *TexEntry* is initialized with $TexEntry \leftarrow BoxEntry$. The ray-casting loop is run while the ray does not intersect the height field and does not leave the domain of the tile, i.e., $TexEntry_x < N \wedge TexEntry_y < N$. In each ray-casting step, we fetch the current height field texel $(\lfloor TexEntry_x \rfloor, \lfloor TexEntry_y \rfloor)$, and we compute the ray's exit point *TexExit* from that texel. If the ray does not intersect the height field within the texel, we set $TexEntry \leftarrow TexExit$ and proceed with the next ray-casting step. The entry/exit points as well as the correspond-

© The Eurographics Association 2009.

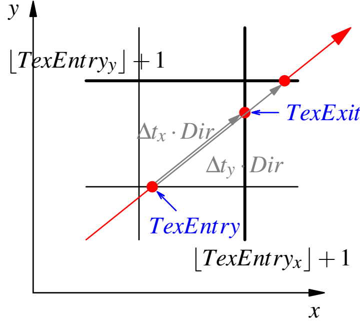

Figure 3: *To determine the exit point of a ray from a texel, two of its edges (bold) have to be considered. Of the ray's two intersection points with these edges, TexExit is the one with the smaller ray parameter.*

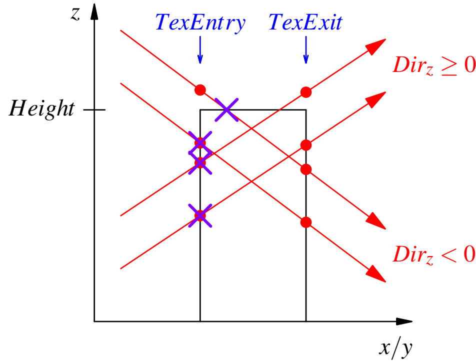

Figure 4: *Dependent on whether the ray is ascending or descending, for the intersection test the texel's height value has to be compared with the height value of either the texel entry or exit point, respectively.*

ing height field texels along a ray are illustrated in Figure 2.

Figure 3 shows the computation of the ray's exit point $TexExit$ from the current height field texel. Of the two intersections with texel edges, $TexExit$ is the one with the smaller ray parameter t. Let $\Delta t_x := \frac{(\lfloor TexEntry_x \rfloor + 1) - TexEntry_x}{Dir_x}$, $\Delta t_y := \frac{(\lfloor TexEntry_y \rfloor + 1) - TexEntry_y}{Dir_y}$ and $\Delta t := \min\{\Delta t_x, \Delta t_y\}$. Then, $TexExit = TexEntry + \Delta t \cdot Dir$. To avoid infinite looping due to roundoff errors, we explicitly set the coordinate corresponding to the intersected texel edge, i.e., if $\Delta t = \Delta t_x$, then $TexExit_x = \lfloor TexEntry_x \rfloor + 1$, else $TexExit_y = \lfloor TexEntry_y \rfloor + 1$.

The intersection test is illustrated in Figure 4. Let $Height$ denote the height value of the current height field texel. If the ray is running upwards, i.e., $Dir_z \geq 0$, the ray intersects the texel iff $TexEntry_z \leq Height$. If the ray is running downwards, i.e., $Dir_z < 0$, the ray intersects the texel iff $TexExit_z \leq Height$. In the former case, if an intersection is detected, $TexEntry$ is the intersection point. In the latter case, the intersection point within the texel's domain has to be computed explicitly as $TexEntry + \max\left\{\frac{Height - TexEntry_z}{Dir_z}, 0\right\} \cdot Dir$.

When the casting of the ray is completed, we sample the photo texture at the intersection point to obtain the color value for the respective pixel, or we discard the fragment if the ray does not intersect the height field. The anisotropic texture filtering is described in Section 3.3.

In the fragment shader, we also compute the screen-space depth of the intersection point and output this value as depth buffer value. Thus, the ray-casting approach can be combined with rasterization-based rendering of polygonal geometry [Mic08].

3.2. Acceleration Structure

To speed up the ray-casting process, we employ an acceleration structure which effectively reduces the number of ray-casting steps to find a ray's intersection point with the height field. The basic idea is to use precomputed information during ray-casting, which enables to aggressively advance the ray for multiple texels per step, without intersecting the height field. Such a technique is cone step mapping [Dum06], which also has been used in the context of terrain rendering [Mic08]. This technique places a circular cone above each height field texel. In a preprocess, the maximum angle for each cone is computed such that the cone does not intersect the height field. During ray-casting, the ray can then safely be advanced inside a cone, without intersecting the height field. The disadvantage of this technique is that the precomputation has to be done in an offline preprocess, which requires to store the results on disk. For height fields of several hundred gigasamples, this would consume tremendous amounts of disk space and disk bandwidth.

In our work, we therefore favor a maximum mipmap pyramid of a tile's height field as proposed in [TIS08]. Each texel in this pyramid is the maximum of the corresponding 2×2 texels in the next finer level. Considering a texel at some level in the pyramid, if a ray does not intersect this texel, it also does not intersect the original height field within the entire domain of that texel. The pyramid is built directly on the GPU by using a simple multi-pass approach, and only introduces additional GPU memory requirements of $\frac{1}{3}$ of the memory needed to store the height field. In the following, the original height field is associated with level number 0, and the coarser levels are associated with ascending level numbers.

To utilize this acceleration structure, the ray traversal algorithm presented in Section 3.1 is extended as follows. Let ℓ denote the current level in the pyramid which is used to test for ray intersection. At the beginning, we start with the second coarsest level consisting of 2×2 texels, i.e., $\ell \leftarrow MaxLevel - 1$ (we skip the coarsest level, since every ray starts at the bounding box of the tile and thus intersects the single texel of that level). In each ray-casting step, we first fetch the current texel $\left(\left\lfloor \frac{TexEntry_x}{2^\ell} \right\rfloor, \left\lfloor \frac{TexEntry_y}{2^\ell} \right\rfloor\right)$ from the current level ℓ, and we compute the exit point $TexExit$ of the ray from that texel as described in Section 3.1, with $\lfloor TexEntry_{x/y} \rfloor + 1$ in the computation being replaced by $\left(\left\lfloor \frac{TexEntry_{x/y}}{2^\ell} \right\rfloor + 1\right) \cdot 2^\ell$. If the ray intersects the texel, we advance the ray to the intersection point, i.e., if $Dir_z < 0$, we set $TexEntry \leftarrow TexEntry + \max\left\{\frac{Height - TexEntry_z}{Dir_z}, 0\right\} \cdot Dir$.

© The Eurographics Association 2009.

Furthermore, if $\ell > 0$, we step one level down in the pyramid, i.e., $\ell \leftarrow \ell - 1$ (if $\ell = 0$, $TexEntry$ is the intersection of the ray with the height field, and the algorithm is finished). If the ray does not intersect the texel, we advance the ray to the texel's exit point by setting $TexEntry \leftarrow TexExit$, and step one level up if the ray leaves a 2×2 texel block, i.e., $\ell \leftarrow \min\{\ell + 1 - (Edge \bmod 2), MaxLevel - 1\}$ with $Edge = \left\lfloor \frac{TexExit_x}{2^\ell} \right\rfloor$ if $\Delta t = \Delta t_x$, and $Edge = \left\lfloor \frac{TexExit_y}{2^\ell} \right\rfloor$ otherwise. We then proceed with the next ray-casting step.

This technique greatly accelerates the ray-casting process. In our experiments, we observed a speedup of about 5 compared with the original approach described in Section 3.1.

3.3. Anisotropic Texture Filtering

After the intersection point of the ray and the terrain has been found, we sample the tile's photo texture at that location to obtain the color for the respective pixel. We use the SampleGrad function provided by Direct3D 10, which enables anisotropic texture filtering by specifying the sampling location as well as two vectors spanning a parallelogram that approximates the pixel's projection into texture-space. Typically, these are the vectors $\left(\frac{\partial u}{\partial x}, \frac{\partial v}{\partial x}\right)$ and $\left(\frac{\partial u}{\partial y}, \frac{\partial v}{\partial y}\right)$, where $\frac{\partial u}{\partial x}$, $\frac{\partial u}{\partial y}$, $\frac{\partial v}{\partial x}$ and $\frac{\partial v}{\partial y}$ are the partial derivatives of the texture coordinates u and v as function of screen-space position (x, y), evaluated at the pixel's center. These derivatives are computed automatically by the graphics hardware in rasterization-based rendering of polygonal surfaces.

In our ray-casting method, we compute the pixel's footprint manually by employing a two step approach. First, we project a texel located at the sampling position from texel-space into screen-space. Assuming that the projection is locally linear, we then use the texel's footprint in screen-space to approximate the pixel's footprint in texel-space.

Let P denote the intersection point of the ray and the height field. We start with projecting the vectors $(1,0)$ and $(0,1)$, spanning a texel located at (P_x, P_y) in texel-space, orthographically onto the tangent plane to the height field at the intersection point P. This corresponds to the orthographic mapping of the photo texture onto the height field. The resulting vectors are $\left(1, 0, \frac{\partial h(P_x,P_y)}{\partial u_t}\right)$ and $\left(0, 1, \frac{\partial h(P_x,P_y)}{\partial v_t}\right)$, with the x/y-coordinates being in texel-space (u_t, v_t) and the z-coordinate being in normalized world-space. The height field's partial derivatives $\frac{\partial h}{\partial u_t}$ and $\frac{\partial h}{\partial v_t}$ at (P_x, P_y) are computed by using central differences (except for the tile border, where forward/backward differences are used). We then reproject the vectors into screen-space, returning vectors a and b which span the texel's footprint in this space.

In the second step, the vectors $(1,0)$ and $(0,1)$, spanning a pixel in screen-space, are expressed in the basis formed by the vectors a and b. The resulting coordinate vectors span a parallelogram which approximates the pixel's footprint in texel-space, and are used as input to the SampleGrad texture sampling function (after scaling by $\frac{1}{N}$ to switch from texel- to texture-space).

3.4. Occlusions Between Tiles

In this section, we show how the rendering of multiple tiles using the presented ray-casting approach can further be optimized by exploiting occlusions between tiles.

The basic idea is to render the tiles in front-to-back order, and to spawn a ray only if for the respective pixel no intersection with previously rendered tiles has been found yet, i.e., for each pixel at most one intersection is determined. The early-z test, which skips the fragment shader invocation for a fragment if it will fail the depth test, would be perfectly suited to implement that optimization, but we are discarding fragments in the fragment shader (if a ray does not intersect the height field), which deactivates early-z testing on current graphics hardware. We therefore pursue a different approach, which is similar to the idea of the early-z test in that we immediately exit the fragment shader and thus skip the expensive ray-casting loop, if we detect that an intersection for the respective pixel has already been found.

This is implemented by using an additional rendering pass for each tile, which precedes the actual ray-casting pass, and in which we detect those pixels covered by the tile for which no intersection has been found yet. In this pass, we render the back faces of the tile's bounding box into an offscreen render target, consisting of a one component, unsigned integer texture which is cleared with 0 at the beginning of every frame. With each fragment, we store a unique tile ID, which is obtained from a counter enumerating the tiles being rendered in each frame. In this pass, depth testing is enabled, but writing into the depth buffer is disabled. Thus, exactly for those pixels covered by the tile's bounding box for which no intersection has been found yet, the tile ID is written into the offscreen render target. Then, in the ray-casting pass, we again render the back faces of the tile's bounding box, as described in Section 3.1. In this second pass, depth testing is disabled (i.e., set to "pass always"), and writing to the depth buffer is enabled. In the fragment shader, we first fetch the ID stored in the offscreen render target for the respective pixel. If this ID is equal to the current tile ID, we spawn a ray. Otherwise, we discard the fragment and immediately exit the fragment shader, without entering the ray-casting loop. Thus, for each pixel at most one intersection point is determined.

Dependent on the amount of occlusion, in our experiments we observed an average speedup of about 2 by using this optimization.

4. System Integration

In this section, we outline the integration of the proposed ray-casting method into a terrain rendering system, which we have presented in our previous work [DSW09].

This system is capable of rendering very large out-of-core terrain data sets by utilizing CPU and GPU memory paging and prefetching techniques, and provides continuous level of detail by using a tile-based multi-resolution representation of the terrain model. In a preprocessing step, we first build a Gaussian pyramid of the terrain's entire height field and photo texture by averaging blocks of 2×2 samples to obtain a sample in the next coarser level. Then, each level is tiled into square regions, with each tile covering exactly four tiles in the next finer level. The tiles are organized in a quadtree, which we refer to as the tile tree. For each tile, a

© The Eurographics Association 2009.

restricted quadtree mesh is constructed, which approximates the tile's height field within a prescribed world-space error tolerance. This mesh, along with the tile's photo texture, is stored on disk. To reduce storage and bandwidth requirements, we have developed a geometry compression scheme for restricted quadtree meshes, and we use the S3TC DXT1 codec to compress the photo texture. Both schemes allow for GPU-based decoding.

During runtime, out-of-core data loading is handled asynchronously by a separate IO-thread, which works independently of the rendering thread and dynamically builds and destroys the tile tree in main memory, dependent on the movements of the viewer. In every frame, the tile tree currently available in main memory is traversed in preorder, and the tiles to be rendered at the current view are determined by view frustum culling and level of detail computation, maintaining a screen-space error tolerance of $\frac{2}{3}$ pixels. In the preorder traversal, the children of a tile are visited in an order that yields a front-to-back sorting of the tiles to be rendered. Exploiting frame-to-frame coherence, only tiles not already residing in graphics memory are uploaded to the GPU. For each of these tiles, a mipmap pyramid of the tile's photo texture is gathered from the tile hierarchy on-the-fly, and the mipmap pyramid as well as the compressed geometry is then loaded into GPU memory. In our previous work, the geometry was then expanded into a triangle list, which was rendered using rasterization-based graphics.

For this work, we have replaced the rendering part of the system. Instead of expanding the compressed geometry into a triangle list, we now rasterize the tile's mesh into a one component, 16 bit floating point (UNORM) texture to reconstruct the tile's height field. It is worth noting that for high-detail data sets with a large number of triangles per tile, this height field even requires less GPU memory than the triangle list. The height field, overlaid with the tile's photo texture, is then rendered using our ray-casting approach presented in Section 3.

5. Results

In this section, we give a detailed analysis of the performance of both the rasterization-based and the ray-casting-based approach. All benchmarks were run on a standard desktop PC, equipped with an Intel Core 2 Quad Q9450 2.66 GHz processor, 8 GB of RAM, and an NVIDIA GeForce GTX 280 graphics card with 1024 MB of local video memory. For all tests, the far plane was set to 600 km, and the screen-space error tolerance was set to $\frac{2}{3}$ pixels.

5.1. Data Sets

For our tests, we used two different data sets. The first data set is a digital model of Vorarlberg, Austria, consisting of a digital surface model at a resolution of 1 m and an orthographic photo texture at a resolution of 12.5 cm for a region of 56 km × 85 km, resulting in a total of 860 GB of data (see Figure 1). The height field of this data set is extremely detailed and clearly exhibits vegetation and buildings.

The second data set is a digital elevation model of the State of Utah at a resolution of 5 m, accompanied by an orthographic photo texture of 1 m (see Figure 5). With a spacial extent of 460 km × 600 km, this data set has a size of 790 GB. Contrary to the first data set, its height field does not contain vegetation and buildings, which have been removed by the provider during data processing. Thus, its height field is much smoother than the one of the Vorarlberg data set.

Figure 5: *Screenshot of the Utah data set, rendered with the ray-caster. With an extent of 460 km × 600 km at a resolution of 5 m and 1 m for the height field and texture, respectively, this data set amounts to 790 GB of data.*

Figure 6: *Color coded iteration step count until a hit is found (from black = 0 steps to white = 50 steps).*

5.2. Performance Analysis

In the first qualitative analysis in Figure 6 we show the number of traversal steps until a hit with the surface is found, color coded from black (= 0 steps) to white (= 50 steps), for the scene shown in Figure 1. Without our pyramidal acceleration structure we would expect up to 1024 steps for a tile size of 512 × 512. In the image, however, it can be seen clearly how effective the quadtree traversal is. Only relatively few rays come close to 50 steps. Overall, we observe a speedup of about 5 compared to an unoptimized ray-caster that simply steps through the highest resolution of the quadtree. As can be seen in Figure 7 and 8 in which the frame rates for a flight over the Vorarlberg and Utah data set are shown, the performance scales linearly with the screen resolution and remains independent of the data set. This shows that our approach scales excellently in the complexity of the terrain. Note that the fluctuations in frame rate are mainly due to the varying sky/terrain coverage of the screen during the course of the flight. The massive peaks in the frame rate show up when the camera is facing down onto the terrain. In this case only very few traversal steps are necessary to find the intersection.

© The Eurographics Association 2009.

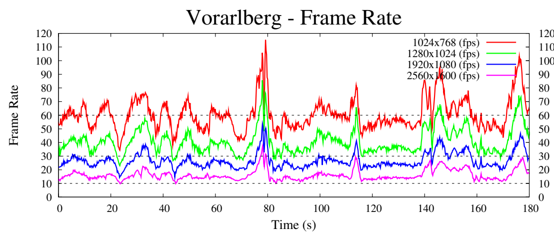

Figure 7: ***Ray-casting*** *frame rate over the course of a flight over the Vorarlberg data set. A screen captured video of the flight can be seen in the accompanying video.*

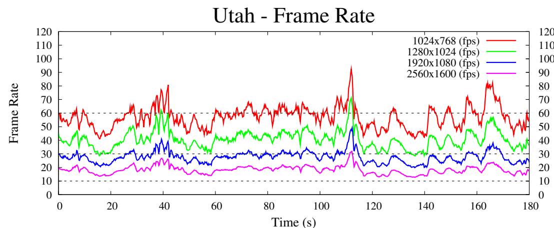

Figure 8: ***Ray-casting*** *frame rate over the course of a flight over the Utah data set.*

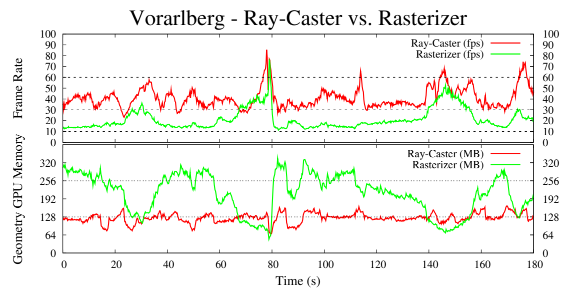

Figure 9: *Direct comparison of the Ray-Casting vs. Rasterization frame rate and GPU memory consumption over the course of a flight over the Vorarlberg data set.*

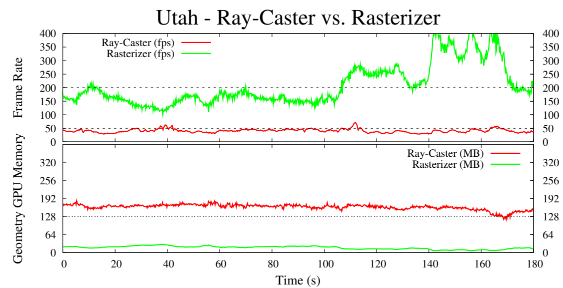

Figure 10: *Direct comparison of the Ray-Casting vs. Rasterization frame rate and GPU memory consumption over the course of a flight over the Utah data set.*

In the upper diagrams of Figures 9 and 10, we compare the frame rates of our novel ray-casting approach and our previous highly optimized rasterization-based solution [DSW09], using a 1280×1024 view port. For the Vorarlberg data set (Figure 9), the ray-caster achieves about twice the performance of the rasterizer, even though our optimized rasterizer has an average throughput of about 350 million triangles per second. In the few cases where the two curves come close to each other or the rasterizer even overtakes the ray-caster, a relatively small number of triangles are sufficient to render the terrain, i.e., the camera is looking at a smooth region of the terrain or is facing down onto the terrain.

In the lower diagrams of Figures 9 and 10, we compare the GPU memory consumption of the two approaches for the tiles' triangle lists (96 bits/triangle) or height fields (16 bits/sample + mipmaps), respectively (the GPU memory required for the photo textures is the same for both approaches). For the Vorarlberg data set (Figure 9), the ray-caster requires much less memory and has an almost constant memory footprint of only about 128 MB to interactively render a data set of about one terabyte. In contrast to this, the rasterizer requires much more memory although it already operates on a compressed representation. Furthermore, as the memory consumption depends on the "roughness" of the terrain currently in view, memory consumption is fluctuating significantly over the path of the flight for the rasterizer, but not for the ray-caster.

For the lower resolution Utah data set (Figure 10), however, the compression of the regularly sampled height field into a mesh pays off and the rasterizer is both faster and has a smaller memory footprint than the ray-caster. Note that the ray-caster does not slow down for this data set and still runs at about 40-50 fps, but the rasterization approach simply becomes much faster due to a significantly reduced number of triangles to be rendered per frame. This observation leads us to the conclusion that for low and medium resolution data sets a hybrid approach may be the fastest solution. Fortunately, our system allows us to render every tile with a different strategy, which should depend on the number of triangles of the tile and the distance to the viewer. Therefore, in the near future we plan to realize such a hybrid approach.

6. Conclusion and Future Work

In this work, we have tackled scalability limitations of terrain viewers, with special emphasis on the development of an advanced rendering back-end. Our goal was to analyze the potential of GPU ray-casting for scalable rendering of gigantic elevation maps and corresponding photo textures on high-resolution display systems. Specifically, we wanted to find out whether ray-casting on the GPU can be positioned as a real alternative to rasterization-based rendering approaches in geo-spatial visualization systems.

Based on current work in the field of GPU ray-casting of height fields, we have proposed some novel contributions in the scope of terrain rendering. These contributions include the integration of GPU ray-casting into a tile-based continuous LOD renderer for large digital elevation models, and a method for fast and stable traversal of regular height field pyramids on the GPU. We have further proposed a novel technique to perform deferred anisotropic texture filtering at visible terrain samples, thus reducing texture fetch operations significantly.

Based on these developments we have carried out a detailed performance analysis of terrain rendering using ray-casting, including the comparison to a highly efficient rasterization-based approach. Our analysis has shown that

© The Eurographics Association 2009.

for moderate data resolutions GPU ray-casting can not achieve the performance of its rasterization-based counterpart. The reason lies in the computation and memory access overhead that is introduced by ray-traversal. At large height field resolutions, however, this overhead can be amortized and the assumed logarithmic complexity of ray-casting in the number of height samples can deploy its full potential.

With respect to our findings, in the future we will continue research on alternative geometry compression schemes for DEMs. At first glance this does not seem to be directly related to the investigations in this work, but if we have a closer look it turns out that the use of a ray-caster for terrain rendering eventually allows us to use more effective compression schemes as we do so far. In the current work we have employed a geometry compression scheme that is based on a particular adaptive triangulation method for height fields. One of the reasons why this method was chosen is that it can effectively reduce the number of triangles that are needed to represent the height field within a certain error tolerance. To render the DEM using ray-casting, however, we first convert the adaptive representation into a regular height map. Consequently, one future challenge will be the development of advanced compression schemes for the initial regular height field, and then to either perform GPU ray-casting on the compressed representation or to first convert it into the kind of height-field pyramid that is used in the current work.

Acknowledgments

The authors wish to thank the Landesvermessungsamt Feldkirch, Austria and the State of Utah for providing high-resolution geo data.

This work was made possible in part by the NIH/NCRR Center for Integrative Biomedical Computing, P41-RR12553-10.

References

[BD06] BABOUD L., DÉCORET X.: Rendering geometry with relief textures. In *Proc. Graphics Interface* (2006), pp. 195–201.

[BDZ06] BALCIUNAS D. A., DULLEY L. P., ZUFFO M. K.: GPU-assisted ray casting of large scenes. In *Proc. IEEE Symposium on Interactive Ray Tracing* (2006), pp. 95–103.

[CGG*03] CIGNONI P., GANOVELLI F., GOBBETTI E., MARTON F., PONCHIO F., SCOPIGNO R.: Planet-sized batched dynamic adaptive meshes (P-BDAM). In *Proc. IEEE Visualization* (2003), pp. 147–154.

[CHCH06] CARR N. A., HOBEROCK J., CRANE K., HART J. C.: Fast GPU ray tracing of dynamic meshes using geometry images. In *Proc. Graphics Interface* (2006), pp. 203–209.

[CKS03] CORRÊA W. T., KLOSOWSKI J. T., SILVA C. T.: Visibility-based prefetching for interactive out-of-core rendering. In *Proc. IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (2003), pp. 1–8.

[CORLS96] COHEN-OR D., RICH E., LERNER U., SHENKAR V.: A real-time photo-realistic visual flythrough. *IEEE TVCG 2*, 3 (1996), 255–265.

[CS93] COHEN D., SHAKED A.: Photo-realistic imaging of digital terrains. *Computer Graphics Forum 12*, 3 (1993), 363–373.

[Don05] DONNELLY W.: *GPU Gems*, vol. 2. Addison-Wesley, 2005, ch. Per-Pixel Displacement Mapping with Distance Functions.

[DSW09] DICK C., SCHNEIDER J., WESTERMANN R.: Efficient geometry compression for GPU-based decoding in realtime terrain rendering. *Computer Graphics Forum 28*, 1 (2009).

[Dum06] DUMMER J.: Cone step mapping: An iterative ray-heightfield intersection algorithm. http://www.lonesock.net/files/ConeStepMapping.pdf, 2006.

[DWS*97] DUCHAINEAU M., WOLINSKY M., SIGETI D. E., MILLER M. C., ALDRICH C., MINEEV-WEINSTEIN M. B.: ROAMing terrain: Real-time optimally adapting meshes. In *Proc. IEEE Visualization* (1997), pp. 81–88.

[EMP*02] EBERT D. S., MUSGRAVE F. K., PEACHEY D., PERLIN K., WORLEY S.: *Texturing and Modeling: A Procedural Approach*, 3 ed. Morgan Kaufmann, 2002.

[GBP06] GUENNEBAUD G., BARTHE L., PAULIN M.: Real-time soft shadow mapping by backprojection. In *Proc. Eurographics Symposium on Rendering* (2006), pp. 227–234.

[Ger01] GERSTNER T.: Fast multiresolution extraction of multiple transparent isosurfaces. In *Proc. Eurographics - IEEE TCVG Symposium on Visualization* (2001), pp. 35–44.

[GMC*06] GOBBETTI E., MARTON F., CIGNONI P., DI BENEDETTO M., GANOVELLI F.: C-BDAM – compressed batched dynamic adaptive meshes for terrain rendering. *Computer Graphics Forum 25*, 3 (2006), 333–342.

[HS04] HENNING C., STEPHENSON P.: Accelerating the ray tracing of height fields. In *Proc. ACM GRAPHITE* (2004), pp. 254–258.

[JMW07] JESCHKE S., MANTLER S., WIMMER M.: Interactive smooth and curved shell mapping. In *Proc. Eurographics Symposium on Rendering* (2007), pp. 351–360.

[LH04] LOSASSO F., HOPPE H.: Geometry clipmaps: Terrain rendering using nested regular grids. In *Proc. ACM SIGGRAPH* (2004), pp. 769–776.

[LKR*96] LINDSTROM P., KOLLER D., RIBARSKY W., HODGES L. F., FAUST N., TURNER G. A.: Real-time, continuous level of detail rendering of height fields. In *Proc. ACM SIGGRAPH* (1996), pp. 109–118.

[LP01] LINDSTROM P., PASCUCCI V.: Visualization of large terrains made easy. In *Proc. IEEE Visualization* (2001), pp. 363–370.

[LP02] LINDSTROM P., PASCUCCI V.: Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. *IEEE TVCG 8*, 3 (2002), 239–254.

[LS95] LEE C.-H., SHIN Y. G.: An efficient ray tracing method for terrain rendering. In *Proc. Pacific Graphics* (1995), pp. 180–193.

[Mic08] MICROSOFT: DirectX Software Development Kit. http://www.microsoft.com/directx, Nov 2008. RaycastTerrain Sample.

[MJ06] MANTLER S., JESCHKE S.: Interactive landscape visualization using GPU ray casting. In *Proc. ACM GRAPHITE* (2006), pp. 117–126.

[Mus88] MUSGRAVE F. K.: *Grid Tracing: Fast Ray Tracing for Height Fields*. Tech. Rep. RR-639, Yale University, Department of Computer Science, 1988.

[Ohb03] OHBUCHI E.: A real-time refraction renderer for volume objects using a polygon-rendering scheme. In *Proc. Computer Graphics International* (2003), pp. 190–195.

[OKL06] OH K., KI H., LEE C.-H.: Pyramidal displacement mapping: A GPU based artifacts-free ray tracing through an image pyramid. In *Proc. ACM Symposium on Virtual Reality Software and Technology* (2006), pp. 75–82.

[OP05] OLIVEIRA M. M., POLICARPO F.: *An Efficient Representation for Surface Details*. Tech. Rep. RP-351, Universidade Federal do Rio Grande do Sul, 2005.

[PG07] PAJAROLA R., GOBBETTI E.: Survey on semi-regular multiresolution models for interactive terrain rendering. *The Visual Computer 23*, 8 (2007), 583–605.

[PO06] POLICARPO F., OLIVEIRA M. M.: Relief mapping of non-height-field surface details. In *Proc. ACM Symposium on Interactive 3D Graphics and Games* (2006), pp. 55–62.

[PO07] POLICARPO F., OLIVEIRA M. M.: *GPU Gems*, vol. 3. Addison-Wesley, 2007, ch. Relaxed Cone Stepping for Relief Mapping.

[QQZ*03] QU H., QIU F., ZHANG N., KAUFMAN A., WAN M.: Ray tracing height fields. In *Proc. Computer Graphics International* (2003), pp. 202–207.

[TIS08] TEVS A., IHRKE I., SEIDEL H.-P.: Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering. In *Proc. ACM Symposium on Interactive 3D Graphics and Games* (2008), pp. 183–190.

[Wym05] WYMAN C.: Interactive image-space refraction of nearby geometry. In *Proc. ACM GRAPHITE* (2005), pp. 205–211.

© The Eurographics Association 2009.