GPU-Assisted Ray Casting of Large Scenes

Daniel A. Balciunas[*] Lucas P. Dulley[†] Marcelo K. Zuffo[‡]

Laboratory of Integrable Systems
University of São Paulo

Abstract

We implemented a pipelined rendering system that pre-renders a reduced set of a scene using the raster method built in the graphics hardware. The computation performed by the graphics card is used as an estimate for evaluating the initial traversal points for a ray caster running on the CPU. This procedure replaces the use of complex spatial subdivision structures for primary rays, offloading work that would traditionally be executed by the CPU and leaving additional system memory available for loading extra scene data. The ray traversal algorithm skips even narrow empty spaces, which are usually hard to map using conventional spatial subdivision. We achieved interactive frame rates (3–10 frames/s) running the system on a single computer with conventional hardware.

Keywords: Ray casting, graphics hardware, height field.

Index Terms: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing; I.3.8 [Computer Graphics]: Applications; I.3.3 [Computer Graphics]: Methodology and Techniques

1 Introduction

For historical reasons, graphics hardware uses the raster method for 3D rendering. Its execution on current graphics cards is much faster than any ray caster running on a CPU and it became the most popular choice for generic image rendering.

However, rendering large scenes can be an arduous task for graphics hardware, particularly when the amount of data is far greater than its available memory. This may restrict quick camera displacement movements, hindering the visualization of a scene with fine details, because scene visualization relies on scene paging algorithms [8], occlusion culling [5] and on the use of level of detail [30] to increase performance.

Ray casting systems are an excellent choice for visualization of huge data sets [9]. They render scenes with billions of polygons at interactive frame rates. However, these systems usually have to run on computer clusters to achieve interactive frame rates, and they do not make intensive use of a valuable and common hardware resource: the graphics card. Even when using a ray tracer, the power of graphics hardware processing is essential for real-time visualization.

Our work proposes and implements a ray casting acceleration system that makes intensive use of the graphics hardware rastering algorithm to evaluate the starting ray traversal points for primary rays. The starting traversal points are positioned very close to the intersection point, avoiding the need to traverse through complex spatial subdivisions. A simple implementation achieved interactive frame rates for large scene data sets, running the test program on a single computer (Figure 1).

[*]e-mail: balciunas@ieee.org
[†]e-mail: dulley@ieee.org
[‡]e-mail: mkzuffo@lsi.usp.br

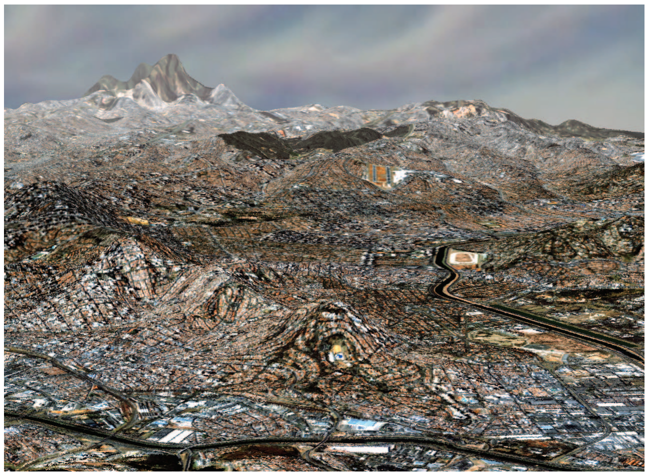

Figure 1: Interactive frame rate acheived for a 268 million voxels height field (equivalent to 537 million triangles) with a 16,384 x 16,384 texture.

This paper is organized as follows: Section 2 presents the motivation for this paper. Section 3 discusses the related work. Section 4 presents a case study, the visualization of high definition height fields. Section 5 describes the use of bounding volumes in our system. Section 6 discusses the system architecture. Finally some results are presented in Section 7 and conclusions in Section 8.

2 Motivation

Since graphics hardware has become a commodity, we decided to propose and implement a rendering system that makes use of this common computational resource and the CPU to render large scene data sets at interactive frame rates. These scenes do not fit the graphics card memory, but they do fit the main memory.

We had basically two choices: either to implement a common rendering system using the graphics hardware as the main renderer, *or* a ray caster rendering system running on the CPU, using the graphics hardware as a secondary resource for speedup. The first approach is widely used, but requires complex additional algorithms, such as occlusion culling [5], scene paging [8] and level of detail [30]. The second approach would use the graphics card computational power to replace a spatial subdivision algorithm that would usually be implemented on the CPU to speedup the ray casting time.

We chose the ray casting approach instead of the graphics card rendering approach. This work shows the speedup achieved when using a rendering system that makes use of the graphics card's original rendering capabilities in order to accelerate a ray caster running on the CPU, without the use of complex algorithms or spatial subdivision for the ray caster. We did not focus on the graphics card rendering approach as it already has been widely tested. Our aim is to compare the acceleration of our method with other ray casting acceleration techniques.

Our acceleration method uses the GPU to render a distance buffer of a simplified set of bounding volumes of the objects of a large scene. This is done computing the distance from the camera to the nearest bounding volume for each pixel. The distance-buffer is then used to evaluate the starting ray traversal points by the ray caster running on the CPU, it places these starting traversal points very close to the intersection, skipping most of the empty regions of the scene.

The speedup comes from the fact that the CPU does not have to traverse each primary ray through any complex spatial subdivision. Whether the bounding volumes are a good approximation of the represented objects, rays do not traverse long distances across the scene before hitting an object, strongly reducing the total ray casting time. For the most generic scenes with large number and kinds of objects, a simple grid [1] would be enough for object culling.

The architecture is basically a pipeline: the graphics card processes the next frame while the ray caster renders the actual frame (we implemented it using multi-thread parallelism). As the graphics card has its own rendering method initially designed for performance and not for visual accuracy, it becomes a good choice to render an approximate scene (stored in the GPU), outputting an approximate distance buffer as the result of its computation. The distance-buffer is then used for ray casting optimization, rendering the full resolution scene (stored on the system memory) with maximum quality. This procedure is illustrated in Figure 2.

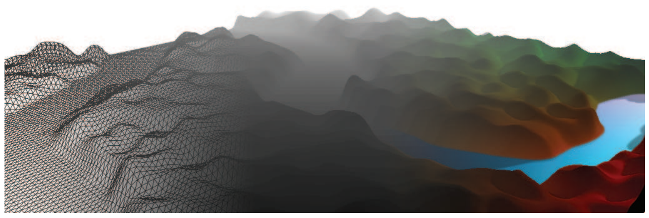

Figure 2: The stages of our rendering system: from simplified scene geometry to depth-buffer and then ray casting.

The proposed system as described is flexible and can be reproduced easily, as it uses only the basics of GPU and thread programming techniques. It requires only hardware that is readily available on recent computers. It does not use any aggressive or relaxed algorithm that would generate artifacts. It also preserves storage space by avoiding the use of complex spatial subdivision for primary rays.

Since the whole scene is available for the ray caster renderer, the camera displacement movements are not restricted: we can quickly navigate through different parts of the scene, and position the camera to display the entire scene (with fine details) without having to wait for long cache operation delays.

The ray caster also may handle object data on its native form: it is not necessary to transform them into a triangle mesh for the ray-object intersection test. This saves a considerable amount of memory for the ray caster scene. It is possible to create specific surface reconstruction intersection tests [15], and choose the appropriate one depending on the distance between the camera and the object (surface reconstruction level of detail). These are advantages over the graphics card raster method rendering approach.

Simple implementations of ray tracers on the CPU usually does not make any use of the GPU. The graphics card is an important computation resource commonly present in most computers nowadays, with a considerable processing power, fast local memory access and recently improved communication interface with the motherboard — PCI Express technology [29]. Currently, to achieve real-time performance goals with ray tracing applications, it is essential the use of graphics hardware as an additional computational resource.

3 Related Work

Rendering large scenes using the raster method of current graphics hardware relies basically on the use of frustum culling, scene paging and level of detail algorithms.

Scene paging [8] is limited by the transfer rate between the graphics card and the system memory, which might hinder the rendering response time of the first frame after a quick camera move or any requisition of a reasonable amount of scene data. This can result in a camera movement restriction imposed by the visualization application.

Level of detail [30] helps to decrease the amount of data processed by the GPU, increasing the frame rate, but it may severely increase the total amount of data stored in memory and increase the traffic between CPU and GPU when rendering large scenes, specially when soft transitions between the levels are required. Level of detail algorithms might not mimic the scene with fidelity when it is necessary to use higher levels with fewer triangles to preserve the frame rate.

Occlusion culling [5] is a feature unavailable in the graphics card raster method. It must be implemented apart and consumes extra processing power. Simple implementations of ray casters already have native object occlusion culling.

A simple approach for rendering large terrains on the GPU is the use of geometry clipmaps [2], combining the techniques presented above and the fact that terrains can be treated as 2D elevation images, resulting in real-time frame rates.

An interesting ray tracing solution for detailed terrain rendering using displacement mapped triangles has been proposed in [26], but interactive frame rates could not be achieved then.

The use of rasterization to find primary ray intersections is introduced in [28]. Our work uses the graphics card rasterization method to obtain an approximation for the starting ray traversal points — a subtle but important difference.

An outstanding solution for huge scene data visualization has been proposed by [9]. An iterative ray tracer uses scene paging techniques to provide interactive frame rates. Billions of polygons can be visualized while moving through the scene. The fine details are loaded when the camera stops moving quickly through the scene. Although the scene data used for this solution does not even fit the system memory, the time required to obtain a fine detail frame is considerable. It also requires the use of several cluster nodes to render at interactive frame rates.

An implementation of a ray tracer on the GPU [4, 21] would be limited for our purpose: interactive frame rates were achieved only for scenes with less than one million triangles. The graphics hardware was not originally designed for this purpose, and ray tracing usually may not fit well on GPU's streaming architecture due to the high number of branches. In addition, if one renders a large scene with a GPU implemented ray tracer, the local graphics card memory would not be enough for loading the entire scene, and scene paging would be required, as in a GPU rendering approach.

4 Case Study: Height Fields

We chose the visualization of high definition height fields as an example to test our system because it was a faster way to obtain preliminary results without developing a complete bounding volume generator solution for generic objects. In future work we plan to extend our system for GPU accelerated ray tracing of generic huge scenes. Our aim here is to obtain the frame rate speedup for the proposed system through the case study of height fields, using a specific bounding volume generator.

High definition height fields are frequently used for a tridimensional visualization of large satellite images of the surface, combining them with the topographic height map of the same region.

Spatial subdivision (e.g., quadtree [7], Figure 3) and object-specific acceleration algorithms (e.g., vertical coherence [6], Figure 4) allow us to accelerate the ray-object intersection test and reduce the main memory access, positioning the starting traversal point after large empty regions.

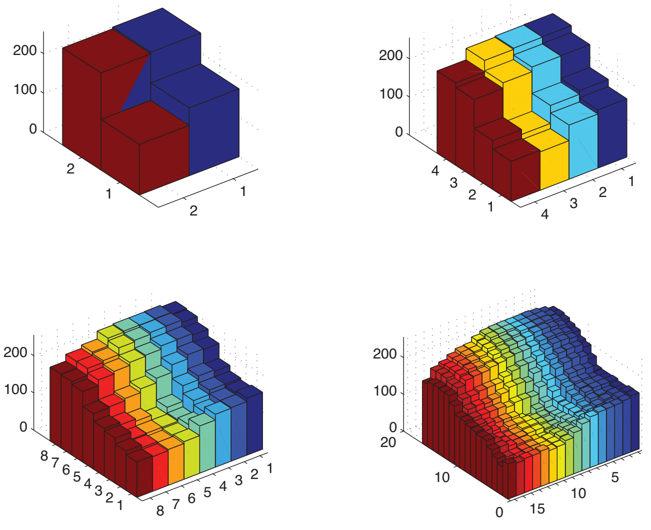
Figure 3: Quadtree spatial subdivision for height fields.

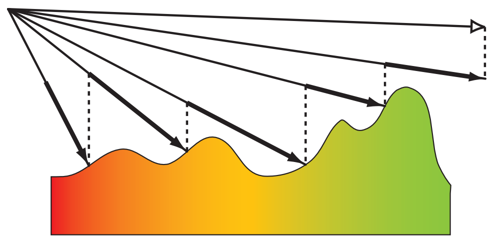
Figure 4: A height field specific algorithm: vertical coherence among rays belonging to the same frame.

Because the number of sampling points of a height field grows quadratically with its dimensions, polygon reducing techniques [10, 23] would allow us to load larger height fields. However, in a ray casting approach, the sampling points may be easily stored in the form of a height matrix and interpreted as voxels during the intersection test and surface reconstruction process, saving considerable memory space. Thus, we do not use triangle reduction techniques for the **ray caster** height field scene.

To keep this solution and still reduce the height field definition on particular regions, it would be necessary to use an additional spatial subdivision structure such as a quadtree. But this would not improve the ray casting time for our rendering system, which already provides empty space skipping. It just would load an extra unnecessary structure in the system memory, reducing the space previously reserved for the scene. The quadtree speedup is also limited to the efficiency of its ray traversal, which depends basically on the memory access rate, usually the bottleneck.

In the case of the traditional vertical coherence algorithm, some parts of the terrain that would intersect the ray are skipped, while in fact they should have been intersected [12]. This usually happens when the camera direction is not parallel to the height fields base plane, so it may be considered as an aggressive algorithm in these cases. Another disadvantage of this algorithm is the restricted camera movement [27]: it is not possible to rotate the camera around its own direction axis (roll), because *up* vector must always be perpendicular to the height field's base plane.

5 Bounding Volume Scene

A bounding volume is a closed volume that completely contains an object. In our system, a set of bounding volumes is created from the original scene's objects to compose a new simplified scene.

The main objective of this operation is to create a scene with a reduced number of polygons, so that it fits the graphics card memory. Fewer polygons also increase the efficiency when the GPU renders the distance buffer.

The secondary purpose is to approximate the objects with bounding volumes as closely as possible, avoiding empty spaces inside them (Figure 5 and Figure 6). This should increase the efficiency of our system, because the starting traversal points will be placed closer to the intersection point. The concept of a good bounding triangle volume is shown in Figure 6.

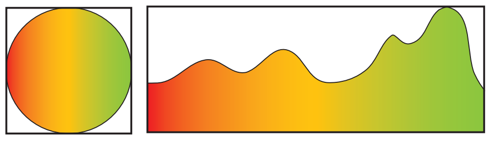
Figure 5: Simple bounding box leaves large empty spaces inside the volume.

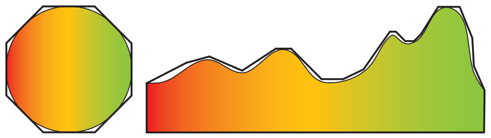
Figure 6: A bounding triangle mesh is a better approximation for the object.

However, the specifications above are conflicting: raising the number of triangles for the bounding volumes leads to a better approximation of the object, but increases the distance buffer rendering time, and vice versa when reducing the number of triangles. Evidently, there is a strong dependency on the hardware specification.

To create a height field bounding triangle mesh, we used a simple method, which reduces the number of triangles by four on each level of iteration, similar to a quadtree constructor, starting from the full-resolution height field. We basically compute the maximum value of a height element and its eight neighbors to get the new height value for the reduced height field. This ensures that the triangle on the reduced height field will stay above the original height field triangles. This can be combined with triangle reduction techniques [10, 23] to improve the performance of the **GPU module**. The mathematical analysis of optimality of this algorithm is an interesting subject for future research.

6 System Architecture

Since the graphics card is capable of evaluating the distances from the camera's position to the projected triangles for each pixel, one can use this information to accelerate the primary rays of a ray caster rendering a similar scene from the same point of view.

We propose a system with a composed rendering method, mixing the graphics card rendering and ray casting rendering. It replaces the use of complex spatial subdivision for primary rays, using the graphics hardware raster method computation for this purpose.

The system is subdivided in three modules: GPU module, Ray Casting Module and Display Module. The GPU module runs mainly on the graphics card, while the Ray Casting module runs on the CPU. The Display module basically shows a color-buffer on screen.

6.1 Data Flow: Workspaces

We decided to create workspaces to implement data flow and synchronization between the modules. Each workspace is associated to a frame. It is merely a data structure composed by a framebuffer (a color buffer and a distance buffer), camera attributes (position, direction and field of view), window attributes (width, height), and scene changes.

The workspaces are necessary because the GPU module and the Display Module run in parallel with the Ray Casting module, and each module processes its own frame. Workspaces can be easily implemented and avoid data mismatching and memory reallocation. Workspace swapping is fast because it is just a pointer attribution operation. Data is synchronized before the execution of the modules, using the swapping process described in the sequel (Figure 7).

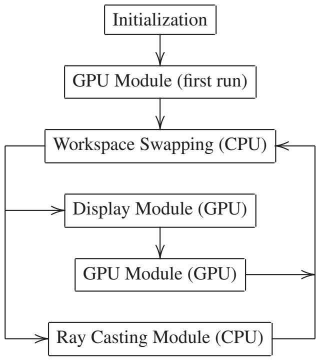

Figure 7: Parallel execution of modules.

The data flow starts on the GPU module, whose workspace receives the latest changes in the scene (window, camera and scene changes). The GPU module renders the distance-buffer (distance between the camera and the nearest object, for a given pixel) and stores it on its workspace.

At this point, workspaces are swapped: the Ray Casting module receives its workspace from the GPU module, which in turn receives a clear workspace from the Display module. The Ray Casting module reads the workspace information to collect scene change information and renders the scene using the distance-buffer previously outputted by the GPU module. The resulting color buffer rendered by the Ray Casting module is then stored on the workspace.

Again, the workspaces are swapped, and the Display module receives the workspace previously used by the Ray Casting module. It displays the color buffer on screen and clears the workspace. This cleanup is needed because GPU module will receive this workspace on next workspace swap.

All the swapping process is shown in Figure 8. Data flows as in Figure 9.

6.2 The Pipeline

As both the GPU module and the Display module use graphics card resources they do not need to run in parallel. This also avoids GPU resource sharing, which might degrade performance.

The Ray Casting module uses only the CPU and can run in parallel with the GPU and Display modules. This is easily implemented through the use of threads.

Both the CPU and the GPU can be used simultaneously, taking advantage of all processing resources available in the computer system (Figure 7).

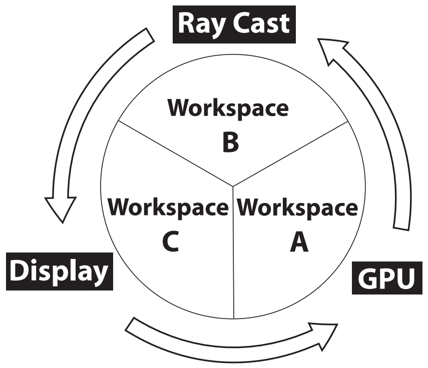

Figure 8: Modules' workspace swapping: an efficient way to transfer a module's output and the scene data changes to the next one.

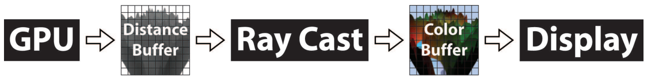

Figure 9: Data flow along modules.

In Table 1 it is possible to follow the rendering of different frames along the pipeline. The greatest disadvantage of this approach is the delay to render the scene changes. The pipeline needs two rendering cycles (instead of one rendering cycle) to transmit the scene changes to the output image. This issue is more noticeable when frame rates are lower than eight frames per second.

Table 1: Frames processed along the rendering pipeline.

Cycle	GPU	Ray Cast	Display
0	Frame 1	Blank Frame	Idle
	Idle	Frame 1	Blank Frame
1	Frame 2	Frame 1	Idle
	Idle	Frame 2	Frame 1
2	Frame 3	Frame 2	Idle
	Idle	Frame 3	Frame 2
3	Frame 4	Frame 3	Idle
	Idle	Frame 4	Frame 3
4	Frame 5	Frame 4	Idle
⋮	⋮	⋮	⋮

6.3 GPU Module

The GPU module is essentially responsible for the early evaluation of the distance estimate of each frontmost (visible) object to the camera position to be used by the Ray Casting module as ray traversal starting points. The general idea of this module is to estimate the distance of each visible object to the camera position by rendering the scene from the camera's point of view and using a GPU generated depth-buffer to evaluate that distance. The distances are stored in the distance-buffer, which is the desired output of the GPU module (Figure 10).

The initialization stage of the GPU module is executed once at our rendering system start-up. During the initialization, we load the height field from a file and create a reduced bounding volume for it (triangle mesh), which is stored on a display list, as the height field is a static object. There is no need to load any texture for the display list geometry on the GPU, as we are only interested in the geometry itself. This leaves memory available for extra geometry.

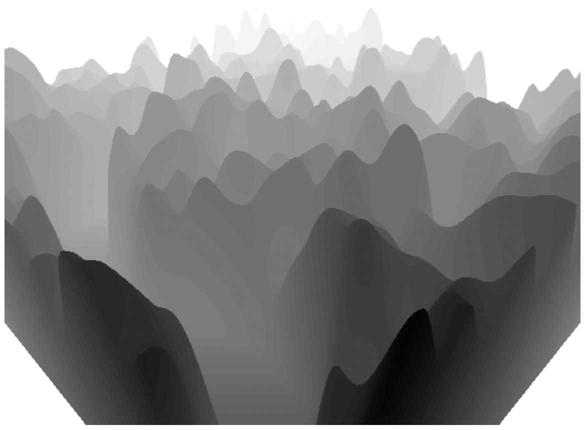

Figure 10: Screenshot of the pre-rendered distance-buffer; white areas are marked with a special flag for ray casting bypass.

The operation of GPU module can be subdivided in three stages: the first GPU rendering pass, the second GPU rendering pass and the distance-buffer readback. These stages are executed on every rendering cycle of our system. The first rendering pass is responsible for rendering the depth of each pixel to a texture using the GPU's original functionality. The second one uses a GLSL [22, 24] fragment shader program running on the GPU's fragment processor. It accesses the previously generated depth texture and evaluates an estimate of the distance from the triangle associated with each pixel to the camera position. The output of the second GPU rendering pass is written to another texture, the distance-buffer. Finally, this distance-buffer is readback to the system memory to be used by the Ray Casting module running on the CPU.

In the first-pass, the OpenGL's fixed functionality renders the scene directly to a depth-texture-object. In fact, only the depth-test is performed and its results are stored in the depth-texture-object. The depth-texture-object is attached to a framebuffer object (first FBO) [17]. The first FBO has no color buffer attached to it, as nothing is rendered to a color buffer (`GL_DRAW_BUFFER` set as `GL_NONE`).

In the second-pass, the drawing of a screen sized quadrilateral (quad) [19] triggers the shader execution. For each screen pixel, the fragment shader evaluates an estimate of the distance between the front-most triangle associated with the pixel and the camera position. The shader evaluates this estimate from the supplied *near* and *far* values and from the depth value of the related pixel stored in the depth-texture, obtained in the first-pass. For each pixel, the object's distance to the camera (z in camera coordinates) is evaluated by Equation 1.

$$\mathrm{dist}(x,y) = \frac{\mathrm{near}\cdot\mathrm{far}}{(\mathrm{far} - \mathrm{depth}\cdot(\mathrm{far}-\mathrm{near}))} \qquad (1)$$

Due to the GPU's architecture, many of these operations are processed in parallel [20]. The obtained distance value is stored in a single channel color-buffer (distance-buffer) attached to a FBO (second FBO). Moreover, if a pixel distance value is greater than or equal to *far*, it is tagged for ray casting bypass. Finally, the distance values from the distance texture are then read back to main memory to be used by the Ray Casting module to evaluate the starting points for the ray traversal.

In our implementation, we use OpenGL 2.0 [25, 24] and GLSL. At the present time, the best way to do proper direct render-to-texture in OpenGL is using `the EXT_framebuffer_object` (FBO) extension [17], which allows texture objects to be used as render targets. We also use the GLEW library [16], which is only used to easily enable and initialize the required OpenGL extensions [18] available in the graphics hardware. We used two FBOs: the first one has only one texture attached (a 32-bit depth texture buffer) and is a depth-only framebuffer, this way the OpenGLs fixed functionality depth buffer data is rendered directly to the depth texture. The second FBO also has only one texture attached to it but this one is a 32-bit single-color-channel buffer where the shader program will render to, storing the the 32-bit floating point distance value to be read back.

As an alternative method, the generation of the distance-buffer might also be done by using standard OpenGL calls: we would render the scene on the GPU to a regular framebuffer (e.g., back buffer) and then read it back to main memory with `glReadPixels`. But, as is widely known, the range of depth buffer values is from 0.0 to 1.0, representing the *near* and *far* planes respectively, and it is not linear. So we would have to do the depth-to-distance transformation for each pixel on the CPU. The additional computational cost of this operation added to the read back cost would hinder the speedup gains obtained by using the GPU. In our system the evaluation is done in the GPU rather than on the CPU, offloading the CPU of performing the depth-to-distance evaluation for the whole depth buffer.

6.3.1 Graphics Hardware Dependency

The readback transfer rate (GPU to CPU) for the GeForce 7900 GTX is 1.3 GB/s, while the download rate (CPU to GPU) is 575 MB/s [3], although according to the the PCIe specification [29] the nominal maximum data PCIe x16 transfer rate in each direction is 4 GB/s. The AGP 8x readback rate is 170 MB/s, while the download rate is 500 MB/s. We measured the total time spent with readback and download operations (Table 2). There is a reasonable difference between AGP 8x and PCIe x16 total transfer time, thus the performance of this module is directly linked to the graphics card communication interface transfer rate and obviously to its performance.

Table 2: Data transfer time in our rendering system for a 800×600 frame size (*for a 10 frames/s rate).

Interface	Readback	Download	Total transfer	Rendering time %*
AGP (8x)	10.77 ms	3.66 ms	14.43 ms	14.4
PCIe (16x)	1.41 ms	3.18 ms	4.59 ms	4.6

6.3.2 Static Scene Data Optimization

From [14] and from initial tests we did, the geometry submission relative performance from best to worst is display lists, vertex buffer objects, vertex arrays, immediate mode (`glBegin` / `glEnd`). This knowledge allows us to minimize CPU-GPU copies, function call overheads and command buffer traffic.

The solution for the time-consuming data transfer is the use of display lists, so that we can cache OpenGL commands on graphics hardware, since the use of display lists allows the storage of OpenGL commands on server side (graphics hardware). Since we are handling static vertex data, its use would be a suitable solution for the optimization of the drawing of the height map mesh. The known drawbacks of using display lists are not an issue for our solution, because the time overhead to compile a display list appears only in the program initialization. As the scene in our test is static, there is no need to use vertex buffer objects, hence we opted to create a display list for it. This optimizes the GPU module scene rendering, as display lists store a group of OpenGL commands in the graphics card memory that can be optimized by its driver [25].

6.3.3 Two-pass vs. Single-pass

Although our first and successful choice was the two-pass approach, we could not disregard trying a single-pass one. We did face major drawbacks when using a single-pass with a fragment shader: the performance was worse than expected and, more important, it was worse than the original two-pass solution.

In a single-pass, the fragment shader has to use the depth data of the fragment `gl_FragCoord.z` to evaluate the distance estimates and store the result in a color-buffer which has an associated depth-buffer for depth-testing. Since *shader computing* equals *drawing* [11], the fragment shader is run for every primitive (inside the view frustum) stored in a display list, whether they were visible or occluded. This can introduce a substantial overhead and loss of performance. Even one of the simplest fragment shaders such as `gl_FragColor.x = -1.0` runs slower than OpenGL fixed functionality (Table 3).

One might think that an occlusion query may help. But an occlusion query test prior to an eventual single-pass — with the purpose of reducing the amount of geometry to be processed by the fragment shader — would be pointless, if not redundant, as the data we actually need is the depth value of the frontmost polygons.

Table 3: GPU Module Performance Test: two-pass vs. single-pass comparison (*very simple shader).

Height Field Size	Single-pass	Single-pass*	Two-pass
2 Million triangles 1024×1024 voxels	20.34 (frames/s)	42.81 (frames/s)	43.17 (frames/s)
8 Million triangles 2048×2048 voxels	6.33 (frames/s)	13.67 (frames/s)	14.74 (frames/s)
33 Million triangles 4096×4096 voxels	3.19 (frames/s)	4.12 (frames/s)	4.63 (frames/s)

In the two-pass approach the inherent hardware optimization of the OpenGL's fixed functionality combined with the direct-rendering to a depth-texture-object make the first pass very efficient. In the second pass, the fragment shader — which evaluates the camera-object distance estimate — is executed for each screen pixel. The only geometry involved in this pass is the *quad* drawn for a texel to pixel one-to-one mapping, thus the second pass is independent of the geometry complexity of the scene.

For the reasons above the original two-pass approach was used since it suit our needs, runs faster and is more optimized than the single pass approach.

6.4 Ray Casting Module

After the ray direction generation, the Ray Casting module sets the ray origin. Instead of using the camera position, it sets the ray origin of each ray using the distance buffer generated by the GPU module as in Equation 2:

$$\text{origin}(x,y) = \text{camera} + \text{direction} \cdot \text{dist}(x,y) \qquad (2)$$

Where (x, y) are the coordinates of each pixel on the screen. This procedure skips all the empty space between the camera and the closest bounding volume of an object, since the scene in the GPU module is essentially a set of bounding volumes of the Ray Casting module's scene.

To reconstruct the height field surface near to the camera, we implemented a bilinear analytical intersection test. For regions far from the camera, the intersection test is a simple height plane reconstruction test, which allows a quick intersection test without hindering the image quality. By implementing this kind of *reconstruction level of detail* we can efficiently save ray casting time.

A screenshot of the resulting image is shown in Figure 11.

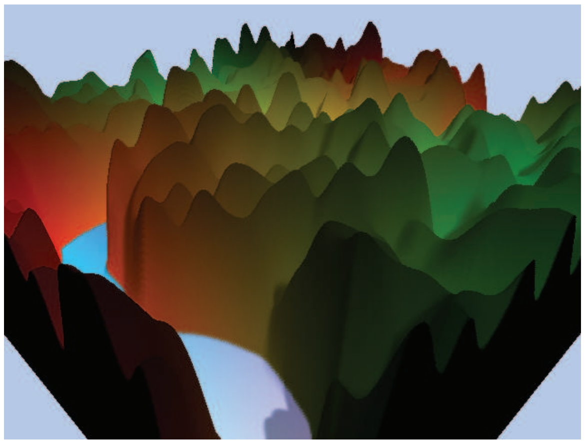

Figure 11: Ray casting resulting image.

As there is no dependency between rays, it is possible to use any number of threads to render different parts of the screen, significantly raising the resulting frame rate for multi-processed CPUs.

For a better image quality, it is possible to use MIP mapping for the height fields texture. This is a good choice when there are too many fine details in the texture, as in satellite images of large cities, preventing texture aliasing. It also reduces the access to the main memory when rendering regions far from the camera. Shadows may also be pre-processed on the texture using a ray-tracing algorithm, adding more realism to the scene.

The Ray Casting module's main output is the color buffer (Figure 11). Its performance is fundamentally linked to the CPUs processor performance and to the data transfer rate of the system memory bus.

6.5 Display Module

This module downloads the color buffer rendered by the Ray Casting module to the graphics card to show it on screen. Due this operation, this module's performance is directly linked to the graphics card communication interface transfer rate.

7 RESULTS

Different height field sizes were used to test our rendering system (Figure 12). The adopted screen size for these tests was 800×600 pixels. The height field object covers approximately 90% of the resulting image (Figure 11), and it is possible to view the entire scene from the rendered point of view. The test computer is a Pentium D 3.0GHz, with 4GB DDR 400MHz system memory, GeForce 7900 GTX 512MB graphics card.

We did not implement our system using specific MMX/SSE instruction set, as this is not the main objective of this work. We just compiled it using MMX/SSE optimization flags, substantially increasing the performance.

In our case study, we compared our method to a naïve ray caster [13], to a classic object-specific ray traversal acceleration algorithm (vertical coherence [6]) and to a hierarchical spatial subdivision tree algorithm (quadtree [7]). We implemented all of them using multi-thread parallelism. In all cases we used the same surface reconstruction method (described in section 6.4). The proposed method achieved better frame rates compared to any of these algorithms.

The frame rate values obtained for the naïve ray traversal exceeded the expected, relatively to the other algorithms. This happens because the ray caster accesses the same amount of main memory to get the texture color of the intersection point, for each pixel in the color buffer. In other words, even using a ray caster, the shading process still has its weight in the final frame rate measure. In

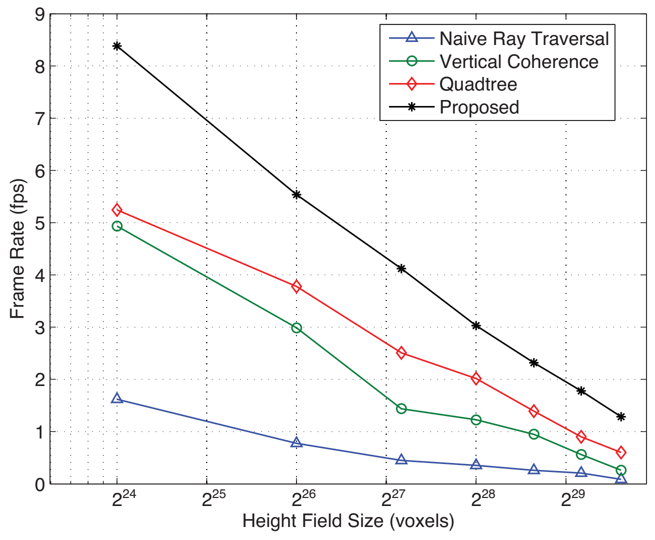

Figure 12: Resulting frame rates of different height field ray casting algorithms and our implemented system.

this way, we should consider the gain of our method at least higher than the gain shown through the frame rate measured.

The performance of the vertical coherence algorithm raises as the screen resolution grows and the height fields shrinks. It gets worse as the intersected terrain becomes more distant from the camera, as expected, because each subsequent ray must traverse longer distances. The performance of the quadtree is slightly better than the vertical coherence one, as memory caching works better for it, reducing system memory access, as expected too.

For the given test computer, we used a reduced scene with 4 Million triangles for the GPU, as the resulting GPU module frame rate for this number of triangles (Table 3) is always above the rendering system frame rate (Figure 12). This is vital, because the bottleneck of our system cannot be the GPU Module, as its main functionality is to accelerate the Ray Casting module. Thus, although there was a frame rate speedup, the bottleneck of our system must always be the Ray Casting module, which is the core of our rendering system.

8 Conclusion and Future Work

The cost of the GPU module operation is balanced by the gain obtained when positioning the ray traversal starting points close to the intersection, resulting in interactive frame rate. The main disadvantage of the proposed method is that it accelerates only primary rays. As secondary rays have different starting traversal points, it would be necessary to change the camera position on the GPU module and reduce the depth buffer size / the field of view for each secondary ray. It would also be necessary to increase the number of steps (modules) in the pipeline to support multiple reflections/refractions, increasing the total rendering time of a frame, loosing the interactivity.

Since our method requires only a small portion of the system memory, it does not exclude the use of additional spatial subdivision algorithms — this would be useful to accelerate the ray casting of generic scenes. The method also supports ray oversampling for border anti-aliasing purposes (this could be easily implemented simply by increasing the distance-buffer size). This method may also accelerate ray tracing of complex objects through the use of bounding triangle meshes.

Our work proposed the use of GPU raster method and shader to assist ray tracers runing on the CPU. Ray casting of generic scenes and shadow mapping using a similar technique is currently being developed.

In future work, an algorithm for dynamic scene complexity adjustment of the GPU module could also be proposed, removing the bottleneck from the Ray Caster module and resulting in a better usage of processing power for the whole system. It is also possible to implement this method for dynamic scenes, simply by extending the module scene data synchronization code.

Acknowledgements

We would like to thank Felipe Miguel Pait, Bruno Barberi Gnecco and Marcelo de Paiva Guimarães for many useful suggestions, helpful discussions and revision, and the anonymous reviewers for their feedback. This work was supported by FINEP (Studies and Projects Funding Body, an agency linked to the Brazilian Ministry of Science and Technology) and by the CAVERNA Digital (a facility located at Laboratory of Integrable Systems at University of São Paulo).

References

[1] J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. In *Proceedings of Eurographics'87*, pages 3–10, Amsterdam, North-Holland, 1987. Elsevier Science Publishers.

[2] A. Asirvatham and H. Hoppe. Terrain rendering using gpu-based geometry clipmaps. In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 27–44, Massachusetts, USA, march 2005. Addison Wesley.

[3] I. Buck, K. Fatahalian, and P. Hanrahan. Gpubench: Evaluating gpu performance for numerical and scientific applications. In *Proceedings of the 2004 ACM Workshop on General-Purpose Computing on Graphics Processors*, 2004. Go online to `http://graphics.stanford.edu/projects/gpubench/`.

[4] N. A. Carr, J. Hoberock, K. Crane, and J. C. Hart. Fast gpu ray tracing of dynamic meshes using geometry images. In *Proceedings of Graphics Interface 2006*, Amsterdam, North-Holland, 2006.

[5] D. Cohen-Or, Y. Chrysanthou, C. Silva, and F. Durand. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics*, 2002.

[6] D. Cohen-Or, E. Rich, U. Lerner, and V. Shenkar. A real-time photo-realistic visual flythrough. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):255–265, 1996.

[7] D. Cohen-Or and A. Shaked. Photo-realistic imaging of digital terrains. *Comput. Graph. Forum*, 12(3):363–373, 1993.

[8] W. T. Corrêa, J. T. Klosowski, and C. T. Silva. Visibility-based prefetching for interactive out-of-core rendering. In *Proceedings of PVG 2003 (6th IEEE Symposium on Parallel and Large-Data Visualization and Graphics)*, pages 1–8, 2003.

[9] A. Dietrich, I. Wald, and P. Slusallek. Large-Scale CAD Model Visualization on a Scalable Shared-Memory Architecture. In *Proceedings of 10th International Fall Workshop - Vision, Modeling, and Visualization (VMV) 2005*, pages 303–310, Erlangen, Germany, November 2005. Akademische Verlagsgesellschaft Aka.

[10] L. D. Floriani and E. Puppo. Hierarchical triangulation for multiresolution surface description. *ACM Trans. Graph.*, 14(4):363–411, 1995.

[11] D. Göddeke. Gpgpu–basic math tutorial. Technical report, FB Mathematik, Universität Dortmund, Nov. 2005. Ergebnisberichte des Instituts für Angewandte Mathematik, Nummer 300, `http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu`.

[12] L. C. Guedes, M. Gattass, and P. C. P. Carvalho. Real-time rendering of photo-textured terrain height fields. In *Computer Graphics and Image Processing, 1997. Proceedings. X Brazilian Symposium on Computer Graphics and Image Processing*, Outubro 1997.

[13] L. C. Guedes, M. Gattass, and P. C. P. Carvalho. Real-time rendering of photo-textured terrain height fields. In *Computer Graphics and Image Processing, 1997. Proceedings. X Brazilian Symposium on Computer Graphics and Image Processing*, Outubro 1997.

[14] E. Hart, B. Licea-Kane, S. Green, M. Harris, and C. Everitt. Opengl performance tuning, mar 2006. Go online to `http://www.ati.com/developer/gdc/2006/GDC06-OpenGL_Tutorial_Day-Hart-OpenGL_03_Performance.pdf`.

[15] C. Henning and P. Stephenson. Accelerating the ray tracing of height fields. In *GRAPHITE '04: Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 254–258, New York, NY, USA, 2004. ACM Press.

[16] M. Ikits and M. Magallon. The opengl extension wrangler library. Go online to `http://glew.sourceforge.net/`.

[17] J. Juliano and J. Sandmel. Opengl framebuffer object extension. Go online to `http://www.opengl.org/registry/specs/EXT/framebuffer_object.txt`.

[18] OpenGL. Opengl extentions. Go online to `http://opengl.org/documentation/extensions/`.

[19] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, Aug. 2005.

[20] M. Pharr and F. Randima, editors. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 453–545. Addison Wesley, Massachusetts, USA, march 2005.

[21] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

[22] R. J. Rost. *OpenGL Shading Language*. Addison-Wesley Professional, 2nd edition, January 2006.

[23] S. Röttger, W. Heidrich, P. Slusallek, and H.-P. Seidel. Real-time generation of continuous levels of detail for height fields. In *WSCG'98 Conference Proceedings*, 1998.

[24] M. Segal and K. Akeley. The opengl graphics system: A specification (version 2.0), October 2004. http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf.

[25] D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2*. Addison-Wesley Professional, 5th edition, July 2005.

[26] B. E. Smits, P. Shirley, and M. M. Stark. Direct ray tracing of displacement mapped triangles. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pages 307–318, London, UK, 2000. Springer-Verlag.

[27] M. Wan, H. Qu, and A. Kaufman. Virtual flythrough over a voxel-based terrain. In *VR '99: Proceedings of the IEEE Virtual Reality*, page 53, Washington, DC, USA, 1999. IEEE Computer Society.

[28] H. Weghorst, G. Hooper, and D. P. Greenberg. Improved computational methods for ray tracing. *ACM Trans. Graph.*, 3(1):52–69, 1984.

[29] A. H. Wilen, J. P. Shade, and R. Thornburg. *Introduction to PCI Express: A Hardware and Software Developer's Guide*. Intel Press, Oregon, USA, 2003.

[30] J. C. Xia, J. El-Sana, and A. Varshney. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):171–183, Apr. 1997. ISSN 1077-2626.