

GPU-Based Nonlinear Ray Tracing

D. Weiskopf, T. Schafhitzel, and T. Ertl

Institute of Visualization and Interactive Systems
University of Stuttgart

Abstract

In this paper, we present a mapping of nonlinear ray tracing to the GPU which avoids any data transfer back to main memory. The rendering process consists of the following parts: ray setup according to the camera parameters, ray integration, ray-object intersection, and local illumination. Bent rays are approximated by polygonal lines that are represented by textures. Ray integration is based on an iterative numerical solution of ordinary differential equations whose initial values are determined during ray setup. To improve the rendering performance, we propose acceleration techniques such as early ray termination and adaptive ray integration. Finally, we discuss a variety of applications that range from the visualization of dynamical systems to the general relativistic visualization in astrophysics and the rendering of the continuous refraction in media with varying density.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

1. Introduction

Ray tracing is a versatile technique for the computation of global illumination and, therefore, is widely used in computer graphics. It is based on geometric optics and is determined by two major components—the propagation of light between scene objects and the interaction between light and matter. The underlying mathematical framework can be formulated in the form of the rendering equation [Kaj86]. In traditional ray tracing, the interaction is restricted to reflection and transmission points on the objects' surfaces and the light propagation is assumed to be linear between these intersection points. Nonlinear ray tracing generalizes the light-propagation step by including curved light rays, while keeping the traditional reflection and transmission computations. This model is an appropriate description for a number of physical scenarios, such as gravitational lensing by strong gravitational sources (like galaxy clusters or neutron stars) or light propagation within a medium with a space-variant index of refraction (like mirages or in the vicinity of explosions).

Nonlinear ray tracing builds upon linear ray tracing and essentially extends the representation of light rays—bent rays are approximated by polygonal lines. Unfortunately, this polygonal representation leads to a significant increase in the number of ray-object intersections that have to be

computed. Typically, nonlinear ray tracing is slower by more than one or two orders of magnitude, compared to linear ray tracing for the same scene. The goal of this paper is to provide a fast GPU implementation of nonlinear ray tracing that exploits the strengths of current GPUs by realizing all ray-tracing steps without any data transfer back to main memory. In addition to this direct GPU mapping, we introduce a number of acceleration techniques to further improve the rendering performance, e.g., early ray termination and adaptive ray integration. Finally, we discuss examples from the visualization of dynamical systems, general relativistic visualization in astrophysics, and the rendering of the continuous refraction in media with varying density.

2. Previous Work

Related previous work can be separated into two different categories that have not yet been investigated together: GPU-based methods and nonlinear ray tracing. With respect to GPU techniques, the implementations of linear ray tracing [PBMH02] and the computation of intersections between rays and triangles [CHH02] are most relevant for this paper. Recently, this line of research has been extended to photon mapping [PDC*03], and radiosity and subsurface scattering [CHH03] on GPUs. In general, there is a prevailing trend towards using GPUs for a variety of general purpose com-

putations [GPG04], e.g., for matrix and optimization operations [BFGS03, HMG03, KW03], or the simulation of cloud dynamics [HBSL03].

In the second category, Gröller [Grö95] presents a generic approach for CPU-based nonlinear ray tracing and discusses a number of applications for visualizing mathematical and physical systems. A specific application of nonlinear ray tracing is the visualization of gravitational light bending within general relativity [HW01, KWR02, Wei00]. In the physics literature, the light deflection by neutron stars and black holes is of special interest [NRHK89, Nem93]. Finally, nonlinear ray tracing can be used to simulate the continuous refraction in a medium that exhibits a space-variant index of refraction, e.g., in the vicinity of explosions [YOH00] or for the refraction aspects found in mirages [BTL90].

3. Basic Architecture for Nonlinear Ray Tracing

Our architecture for nonlinear ray tracing is built upon the structure of GPU-based linear ray tracing [CHH02, PBMH02]. Two categories of entities have to be represented: scene objects and light rays. Neglecting possible acceleration data structures (like octrees), a key element of linear ray tracing is the intersection between n_{rays} rays and n_{objs} objects. With [CHH02], this $n_{\text{rays}} \times n_{\text{objs}}$ problem could be regarded to have a “crossbar” structure. On a GPU, the scene objects can be represented by vertex-based geometry, and the rays can be represented by textures. Fragment operations in programmable pixel shaders allow us to combine textures and geometry in such a “crossbar” fashion.

Nonlinear ray tracing introduces an additional dimension: each ray consists of a number of ray segments, n_{segs} . Stated differently, a three-dimensional “crossbar” structure would be required for this $n_{\text{rays}} \times n_{\text{objs}} \times n_{\text{segs}}$ problem. We separate the three-dimensional structure into two levels. In an inner loop, $n_{\text{rays}} \times n_{\text{objs}}$ intersection computations are performed, one for a single segment of each ray. This part is analogous to the implementation of linear ray tracing. The outer loop iterates over all segments n_{segs} .

We restrict ourselves to a ray casting model, traversing only primary eye rays. Shadow rays are problematic in nonlinear ray tracing because a linear projection from light sources to scene objects is generally not possible (see the discussion in [Grö95]). Therefore, secondary effects are neglected. Figure 1 illustrates the complete architecture for this basic nonlinear ray tracing. Ray segments are stored in 2D textures that have a one-to-one mapping to the corresponding pixels on the image plane. Scene objects are represented by vertex-based geometry. In the first step, the initial values for the first segment of each ray are set according to the camera parameters. The subsequent three steps are iterated over all segments n_{segs} . This loop begins with the computation of the current ray segment, guided by the ordinary differential equation (ODE) of an underlying model for the curved

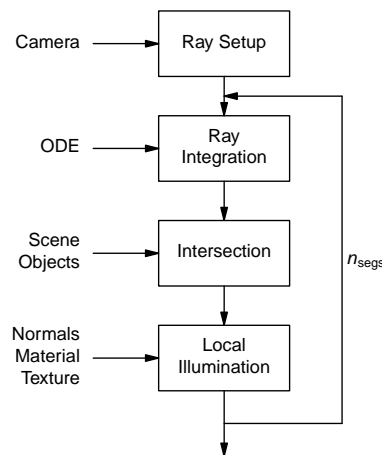


Figure 1: Architecture for nonlinear ray tracing.

rays. Then, the $n_{\text{rays}} \times n_{\text{objs}}$ intersection computations are performed for the current segment index. Finally, pixels are shaded via a local illumination model when an intersection is found. Since the number of ray segments is fixed and the same for all rays, the loop over segments can be conceptually unrolled. Therefore, the ray-tracing architecture is still compatible with the streaming model on GPUs [PBMH02]. In the following subsections, the previously mentioned steps are discussed in more detail.

3.1. Ray Setup

Ray setup computes the initial values for the primary rays. In general, the state of a point on a ray is described by its position $\mathbf{x} \in \mathbb{R}^3$ and its direction $\mathbf{v} \in \mathbb{R}^3$. We denote the combination $\mathbf{p} \equiv (\mathbf{x}, \mathbf{v})$ as an element in *ray phase space* $P = \mathbb{R}^6$. The ray phase space elements are stored in 2D textures whose texels correspond to the associated pixels on the image plane. The six components of \mathbf{p} are distributed over two textures: one texture for positions, the other texture for directions. In our implementation, the components of these textures are stored with floating-point precision for maximum accuracy and flexibility. During ray setup, the initial values for \mathbf{x} are set to the camera position, and the values for \mathbf{v} are determined by the direction towards the corresponding pixel. The texture for \mathbf{v} is filled by rendering a quadrilateral that covers the image plane, using a pixel shader program for the computation of the initial directions. The texture for \mathbf{x} is initialized by writing the (constant) camera position.

3.2. Ray Integration

In a generic approach to nonlinear ray tracing, the bending of rays is determined by an equation of motion in the form

of a system of ODEs,

$$\begin{aligned}\frac{d\mathbf{x}(\tau)}{d\tau} &= \mathbf{v}(\tau) \quad , \\ \frac{d\mathbf{v}(\tau)}{d\tau} &= \mathbf{f}(\mathbf{x}(\tau), \mathbf{v}(\tau), \dots) \quad ,\end{aligned}\quad (1)$$

where τ describes the parameterization along the rays. The properties of the equation of motion are determined by the function \mathbf{f} . The dots indicate that there could be additional parameters that affect the propagation of light. The initial value problem for these differential equations could be solved by any explicit numerical integration scheme. For simplicity we consider first-order Euler integration,

$$\begin{aligned}\mathbf{x}_{i+1} &= h\mathbf{v}_i + \mathbf{x}_i \quad , \\ \mathbf{v}_{i+1} &= h\mathbf{f}(\mathbf{x}_i, \mathbf{v}_i, \dots) + \mathbf{v}_i \quad ,\end{aligned}\quad (2)$$

with the stepsize h and the index i for the points along the polygonal approximation of the light rays. At each integration step, a read access to the textures with index i and a write access to the textures with index $i + 1$ is required. To save memory, only these two copies of textures are held on the GPU. After each integration step, the two copies are alternately exchanged in a ping-pong rendering scheme. The numerical operations for Eq. (2) are implemented in a pixel shader program that outputs its results to multiple render targets, namely the \mathbf{x} and \mathbf{v} textures. Once again, the fragments are generated by rendering a single viewport-filling quadrilateral.

We allow the user to choose the integration stepsize h independently from the ray segment length. Typically, a large number of integration steps is required to achieve an appropriate numerical accuracy. In contrast, the number of ray segments could be smaller without introducing inaccuracy artifacts in the ray-object intersection. A user-specified parameter $n_{\text{integration}}$ describes the number of internal integration steps before a single ray segment is output to the intersection process. A typical value for $n_{\text{integration}}$ is 10.

3.3. Ray-Object Intersection

A ray segment is defined by the line between \mathbf{x}_i and \mathbf{x}_{i+1} from the two latest points in ray phase space. Ray-object intersections are computed directly after the ray integration step. In this way, previous, older segments are no longer needed and can be discarded; i.e., this is a reason why only two copies of the ray phase space textures have to be stored simultaneously.

Our implementation supports triangles and spheres as primitive objects. For triangle-ray intersection, we have adopted the approach by Carr et al. [CHH02] and added a check for the finite extent of a segment. The computation of sphere-ray intersection is based on [Gla93] and also takes into account a finite segment length. Following [CHH02], a single scene object is represented by a vertex-based primitive. The object parameters are transferred to the pixel shader

program in the form of attached texture coordinates. The intersection between an object and all rays is triggered by rendering a viewport-filling quad. The pixel shader program checks whether an intersection takes place and, if that is the case, computes the intersection point.

Visibility is determined by including a test against the depth value that is stored in a depth texture. If the current intersection point is closer, the depth texture will be updated and the corresponding pixel will be drawn (as described in the following subsection on local illumination). Note that the z value (in eye or clip coordinates) is not appropriate to describe a depth ordering for nonlinear ray tracing. Instead, the monotonically increasing index i serves as a measure for the distance between camera and a point on a ray. For the location of a point within a segment, we use a local coordinate that has value 0 at the beginning of the segment and value 1 at the end. The summation of the integer index i and the fractional part from the local coordinate results in an accurate depth ordering. The depth buffer is implemented by a floating-point texture. The simultaneous read and write access for this depth buffer is realized by ping-pong rendering.

3.4. Local Illumination

The last step is the local illumination at the ray-object intersection points. The current implementation supports ambient lighting and light-emitting objects. Textures can be used to modulate the material properties of objects. In addition, “fake” Blinn-Phong illumination is implemented, based on straight connections between a hit point on the surface and the light sources. This model is mainly used for testing purposes.

Based on the position of the intersection, texture coordinates, normal vectors, and material colors are computed by interpolation and used to evaluate the local illumination model. Finally, the value is updated in the image buffer, which is a texture that holds the intermediate pixel colors. Since local illumination requires information about the intersection point, its implementation is combined with the above intersection computation in a single pixel shader program.

4. Acceleration Techniques and Extensions

4.1. Early Ray Termination

Nonlinear ray tracing builds upon a large number of ray segments of finite length. Therefore, the total length of rays directly determines the rendering time. The goal of early ray termination is to reduce the computational steps by pruning rays. We consider two different conditions under which a ray can be stopped without introducing any errors. The first pruning criterion exploits the fact that only the first hit is required for opaque scene objects: Once (at least) one intersection is found within a ray segment, no further segments need to be generated and traversed. The second criterion relies on

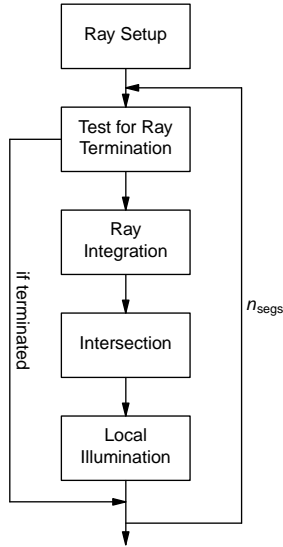


Figure 2: Data flow with early ray termination.

cutting rays that have left the scene and propagate to infinity without any chance of intersecting objects. Here, a bounding geometry is laid around the scene, and rays that intersect this geometry are discarded. Our implementation uses a bounding sphere that contains all scene objects and is placed in almost flat space in which rays cannot “turn around” and propagate back into the scene.

Early ray termination essentially leads to a conditional break in the loop over ray segments. Since breaks are not compatible with the streaming model of GPUs, we propose the following approach. As shown in Figure 2, a complete loop over all segments n_{segs} is performed for all rays. However, the expensive computations for ray integration, intersections, and local illumination are conceptually skipped for terminated rays. The early z test is effective in aborting pixel operations early and therefore cuts down computation times significantly. The early z test allows us to essentially skip the pixel shader operations for the integration, computation, and illumination of terminated rays, even though the corresponding fragments are still generated during rasterization.

The early z test is implicitly enabled on modern GPUs, but works only under some conditions. Most importantly, the depth value of a fragment must not be modified by a pixel shader program. In the architecture from Figure 2, the pixel shader for the “Test for Ray Termination” checks the aforementioned termination criteria. It outputs a depth value of 1 (i.e., the distance of the far clipping plane) if the ray is not terminated, and a value of 0 (i.e., near clipping plane) if the ray is terminated. This step does not yet use the early z test. The subsequent steps, however, are realized by rendering quadrilaterals with a constant z value of 0.5 and can

therefore exploit the early z test to discard pixel operations for terminated rays.

4.2. Adaptive Ray Integration

Adaptive ray integration is an approach to increase both rendering performance and quality. We adopt the idea of step-size control from adaptive numerical integration schemes. In a step-doubling approach, each integration step is taken twice, once as a full step, then, independently as two half steps. The difference between both results is a measure for the accuracy associated with the current stepsize. If the difference is below a given threshold, the stepsize is increased; if the difference is larger than another threshold, the stepsize is decreased. Since pixel shader programs do not support conditional branching, this decision is realized by using compare instructions (`cmp` command in DirectX).

The GPU implementation is illustrated in Figure 3. The steps with sizes h and $h/2$ are computed by the Euler integration process from Section 3.2. The current stepsize, which may differ from ray to ray, is stored in the previously unused fourth component of the \mathbf{v} texture. The “Compute New Step-size” pixel shader compares both results and determines the stepsize for the subsequent integration step.

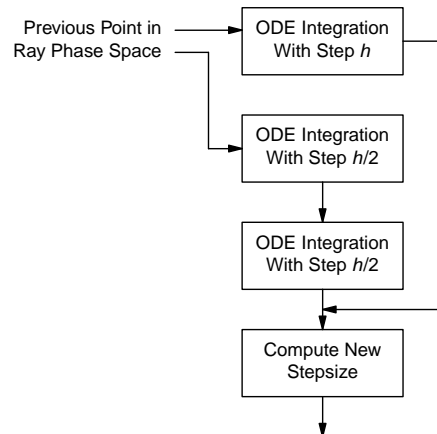


Figure 3: Adaptive ray integration.

Stepsize control not only improves the speed and quality of numerical integration, but, at the same time, can reduce the number of ray segments. For example, regions with only weakly curved rays are covered with large segments and, therefore, only few intersection computations are required. Of course, this speed-up is only effective in combination with early ray termination at scene boundaries and objects because otherwise integration and intersection computations would be performed for a constant, maximum number of segments.

4.3. Environment Mapping for Asymptotically Flat Sky

Light rays that leave the scene boundary usually result in the background color. As an alternative, we use an environment texture that represents light-emitting objects at infinity. A cube texture implements such a “sky box”, where the direction \mathbf{v} of a ray at the boundary serves as texture coordinates. This model is valid for scenarios in which the boundary geometry already is in (almost) flat regions (where rays are not or only slightly bent).

5. Applications

5.1. Visualization of Nonlinear Dynamics

One field of application for nonlinear ray tracing is the visualization of dynamical systems. Nonlinear dynamics and chaotic behavior can be investigated by examining paths in phase space that describe the temporal evolution of a dynamical system [ASY96]. We follow [Grö95] in discussing two examples for chaotic systems—the Lorenz and the Rössler systems. The Lorenz system [Lor63] is governed by

$$\mathbf{f}(\mathbf{x}) = \begin{pmatrix} \sigma(y-x) \\ \rho x - y - xz \\ xy - \beta z \end{pmatrix},$$

with $\mathbf{x} = (x, y, z)$. Figure 4 (b) shows an example for ray tracing with the parameters $\beta = 8/3$, $\sigma = 10$, $\rho = 28$. The Rössler system [Rös76] is described by

$$\mathbf{f}(\mathbf{x}) = \begin{pmatrix} -(x+z) \\ x + \alpha y \\ \beta + z(x-\gamma) \end{pmatrix}.$$

A possible choice of parameters is $\alpha = 3/8$, $\beta = 2$, and $\gamma = 4$.

5.2. Motion in a Potential Field

Another closely related example for nonlinear ray tracing uses the motion in a potential field to model curved paths. The equation of motion for a particle within a radial force field that is centered around the point \mathbf{x}_c yields:

$$\mathbf{f}(\mathbf{x}) = -\frac{(\mathbf{x} - \mathbf{x}_c)}{r} \xi(r),$$

where $r = \|\mathbf{x} - \mathbf{x}_c\|$ is the distance to the center point, and the function $\xi(r)$ describes the radial behavior. This force field subsumes the Yukawa potential,

$$V(\mathbf{x}) = \zeta \frac{e^{-\mu r}}{r},$$

which represents the effective potential for a large class of fundamental physical particle–particle interactions [Gro93]. The corresponding force is computed by determining the gradient of the potential, i.e., $\mathbf{f}(\mathbf{x}) = -\nabla V$. The parameter μ reflects the rest mass of the particles that mediate the interaction; ζ is a constant scaling factor. For example, electromagnetic interaction is mediated by massless photons (the

gauge bosons of the Maxwell field) and therefore has $\mu = 0$. Similarly, Newton’s law of gravitation also has vanishing μ . On the other hand, the strong interaction between nucleons (such as protons or neutrons) is mediated by heavy particles, which reduces the range of interaction and is described by a non-vanishing positive value for μ . Adopting a generalized radial field, $\xi(r)$ can represent any continuous function. The example in Figure 4 (c) uses

$$\xi(r) = 2\frac{r^3}{R^3} - 3\frac{r^2}{R^2} + 1,$$

for $0 \leq x \leq R$ [Grö95].

5.3. Air with Continuously Varying Index of Refraction

Within a medium with a space-variant index of refraction, light is subject to continuous refraction. Typically, a varying index of refraction is caused by a non-constant density of air, for example for mirages [BTL90] or in the vicinity of explosions [YOH00]. The model of [YOH00] allows for a discretization of continuous refraction: The index of refraction is updated along the light ray; and when the index of refraction changes by more than a threshold, the new direction of the ray is computed according to Snell’s law using the gradient of the refraction index as the surface normal.

Figure 4 (d) shows an example of light propagation in a medium with varying index of refraction. The index of refraction resembles the explosion model from [YOH00]. However, we do not use a numerical simulation to compute the spatial distribution of the index of refraction, but a noise-based procedural model that is not based on a physics simulation.

5.4. General Relativistic Visualization

Light is bent by gravitational sources and therefore nonlinear ray tracing is ideally suited for the visualization of the effects of general relativity on light propagation. Light rays are identical to null geodesics within the curved spacetimes of general relativity (see Appendix A). The underlying geodesic equation is a second-order ODE that can be transformed into the structure of Eq. (1). Figure 4 (e) illustrates light bending around a non-rotating black hole whose spacetime is described by the Schwarzschild metric. For comparison, the test scene from Figure 4 (a) is used. The astrophysical scenario in Figure 4 (f) shows a neutron star (blue) and a much less heavier, accompanying star (yellow) in front of a background star field. Checkerboard textures are attached to the two stars to visualize the distortions on the surfaces. The background is represented by an environment texture that is mapped onto a “sky box” as described in Section 4.3.

Relativistic visualization supports scientists in understanding numerical or analytical results from gravitational research because it provides a compact representation that is independent of the coordinate system [Wei00]. In addition,

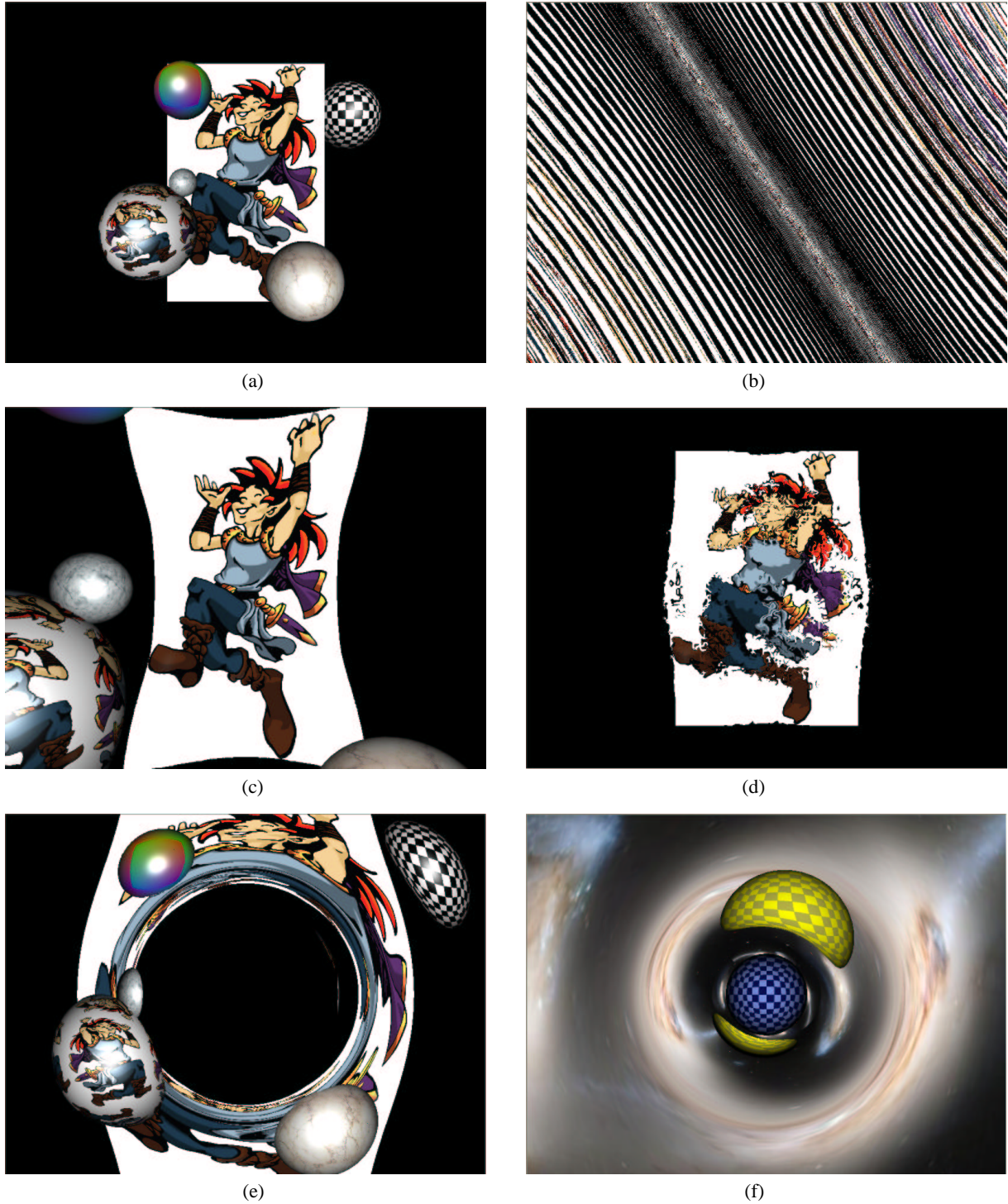


Figure 4: Images generated by GPU-based nonlinear ray tracing: (a) undistorted image of the test scene; (b) rays governed by the Lorenz system, with the same test scene as in (a); (c) rays in a radial potential field; (d) rays in a medium with varying index of refraction; (e) general relativistic ray tracing with a black hole (Schwarzschild spacetime). Image (f) visualizes an astrophysical scenario with a neutron star (blue) and a much less heavier, accompanying star (yellow) in front of a background star field. Checkerboard textures are attached to the two stars to reveal the distortions on the surfaces.

visualization also serves as a tool in teaching physics courses and explaining important aspects of relativity to the public, e.g., in popular-science films or exhibitions.

6. Implementation and Results

Our implementation is based on DirectX 9.0, and all fragment operations are formulated in the assembler-level Pixel Shader 2.0 language. We use 32 bit floating-point textures to represent the depth values and the positions \mathbf{x} and directions \mathbf{v} along rays. Tests with 16 bit floating-point textures led to a significantly degraded image quality and thus showed that the accuracy of integration and intersection computations was heavily affected. All images in Figure 4 were generated by our ray-tracing system on a Windows XP PC with an ATI Radeon 9700 (128 MB) GPU and a Pentium 4 (2.8 GHz) CPU. The following measurements were also performed with this hardware configuration.

Figure 5 shows the performance characteristics for a varying number of spherical scene objects. The other parameters are fixed: The viewport has a size of 800×600 pixels; 500 integration steps are computed within a Schwarzschild spacetime (Section 5.4), leading to 50 ray segments. The upper (slower) curve in Figure 5 represents the original, non-optimized implementation from Section 3, while the lower (faster) line displays the rendering performance for ray tracing with early ray termination and adaptive ray integration from Section 4. The vertical offset of the two curves for $n_{\text{objs}} = 1$ indicates how much time is spent to solve the ODE and construct the curved rays. Under the present test conditions, the acceleration techniques reduce the computation times by some forty percent. More importantly, the slope of the lower curve is much smaller than the slope of the upper curve, i.e., the acceleration methods improve the intersection computations by pruning the rays. Both curves are almost

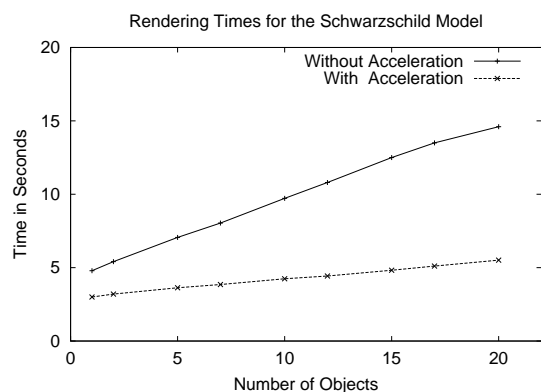


Figure 5: Comparison of rendering times for accelerated and non-accelerated nonlinear ray tracing. The number of spheres in the scene changes along the horizontal axis.

Table 1: Rendering times in seconds on a 800×600 viewport, with 10 scene objects, 30 ray segments, and 300 integration steps.

	No Acceleration Methods	Acceleration Methods
Lorenz System	3.87	3.13
Potential Field	4.22	1.16
Varying Index of Refraction	3.35	2.42
Schwarzschild	4.91	3.79

straight lines, which shows that the intersection and shading steps depend linearly on the number of scene objects.

Table 1 documents rendering times for different models of curved rays. We compare results for the Lorenz system, a potential field, a medium with varying index of refraction, and the Schwarzschild metric. The viewport has a size of 800×600 pixels, the scene consists of 10 spherical objects, and 300 integration steps were performed to build 30 ray segments each. The first column shows the rendering times for the non-optimized implementation. The Schwarzschild solver is slower than the solvers for the other models because its evaluation of the function \mathbf{f} involves a larger number of numerical instructions. The second column reflects the rendering times for the acceleration methods. Rendering is faster, although the speed-up heavily depends on the underlying ODE system. In our example for the potential field, only a weak attractive force is applied, leading to rather straight light rays. Therefore, a significant increase in speed can be achieved by adaptive integration.

7. Conclusion and Future Work

We have presented a fast GPU implementation of nonlinear ray tracing that avoids any data transfer back to main memory. Curved light rays are represented by polygonal lines that are constructed via an iterative numerical ODE solver. The rendering process can be mapped to the streaming model of current GPUs by subsequently executing ray setup, ray integration, ray-object intersection, and local illumination. We have proposed two acceleration techniques to improve the rendering performance: early ray termination and adaptive ray integration. In particular, we have investigated ways to introduce such acceleration techniques into the streaming architecture on GPUs.

In future work, some of the bottlenecks of the current implementation could be addressed. Especially, the inner loop for the internal ODE integration steps could be unrolled within a longer pixel shader program. In this way, much communication via the floating-point textures for ray positions and directions could be avoided. Furthermore, space partitioning strategies that are known from linear ray tracing

could be incorporated to achieve a better scalability with respect to the number of scene objects. Finally, deferred shading could be used to accelerate the illumination computation. By deferring the shading process to the very end of the ray-tracing algorithm, lighting would only be evaluated for actually hit points.

Acknowledgments

We would like to thank the anonymous reviewers for helpful remarks to improve the paper. Special thanks to Bettina Salzer for proof-reading, and to Joachim Vollrath for his help with the video.

Appendix A: Geodesics in Spacetime

Here, a brief discussion of the mathematical background of general relativity and, in particular, the propagation of light is given. For a comprehensive presentation we refer to the textbooks [MTW73, Wei72]. The concept of curved spacetime is the geometric basis for general relativity. Spacetime is a pseudo-Riemannian manifold and can be characterized by the infinitesimal distance ds ,

$$ds^2 = \sum_{\mu, \nu=0}^3 g_{\mu\nu}(\mathbf{x}) dx^\mu dx^\nu \quad ,$$

where $g_{\mu\nu}(\mathbf{x})$ are entries in a 4×4 matrix, representing the metric tensor at the point \mathbf{x} in spacetime. The quantities dx^μ describe an infinitesimal distance in the μ direction of the coordinate system. Trajectories of freely falling objects are identical to geodesics. Geodesics are the generalization of the idea of straightest lines to curved manifolds and are solutions to the geodesic equations, a set of second-order ODEs:

$$\frac{d^2 x^\mu(\tau)}{d\tau^2} + \sum_{\nu, \rho=0}^3 \Gamma^\mu_{\nu\rho}(\mathbf{x}) \frac{dx^\nu(\tau)}{d\tau} \frac{dx^\rho(\tau)}{d\tau} = 0 \quad ,$$

where τ is an affine parameter along the geodesic. The Christoffel symbols $\Gamma^\mu_{\nu\rho}$ are determined by the metric according to

$$\Gamma^\mu_{\nu\rho}(\mathbf{x}) = \frac{1}{2} \sum_{\alpha=0}^3 g^{\mu\alpha}(\mathbf{x}) \left(\frac{\partial g_{\alpha\nu}(\mathbf{x})}{\partial x^\rho} + \frac{\partial g_{\alpha\rho}(\mathbf{x})}{\partial x^\nu} - \frac{\partial g_{\nu\rho}(\mathbf{x})}{\partial x^\alpha} \right),$$

where $g^{\mu\alpha}(\mathbf{x})$ is the inverse of $g_{\mu\alpha}(\mathbf{x})$. Light rays are a special class of geodesics: lightlike or null geodesics, which fulfill the null condition,

$$\sum_{\mu, \nu=0}^3 g_{\mu\nu}(\mathbf{x}) \frac{dx^\mu(\tau)}{d\tau} \frac{dx^\nu(\tau)}{d\tau} = 0 \quad .$$

During ray setup, the initial position in spacetime and the initial spatial direction of the light ray are determined as in the other models from Section 5. The temporal component of the initial direction is then computed according to the null condition.

An important class of spacetimes is described by the Schwarzschild metric,

$$ds^2 = \left(1 - \frac{2M}{r}\right) dt^2 - \frac{dr^2}{1 - 2M/r} - r^2 (d\theta^2 + \sin^2 \theta d\phi^2).$$

This metric represents a vacuum solution for Einstein's general relativistic field equations and describes the spacetime around a non-rotating, non-charged, spherically symmetric distribution of matter and energy. It applies to many compact astrophysical objects, for example, to regular stars, neutron stars, or black holes. We choose units in which the speed of light and the gravitational constant are 1. The parameter M denotes the mass of the source of gravitation. In asymptotically flat outer regions of spacetime, the spherical Schwarzschild coordinates, r , θ , and ϕ , are identical to the standard spherical coordinates of flat space. In our implementation, the spherical Schwarzschild coordinates are transformed into pseudo-Cartesian Schwarzschild coordinates. In this way, the x , y , and z components in the ray setup can be directly used as input to the Schwarzschild metric. Finally, t denotes time. The temporal component of the positions $x^\mu(\tau)$ can be neglected in stationary scenes.

References

- [ASY96] ALLIGOOD K. T., SAUER T. D., YORKE J. A.: *Chaos: An Introduction to Dynamical Systems*. Springer, New York, 1996.
- [BFGS03] BOLZ J., FARMER I., GRINSPUN E., SCHRÖDER P.: Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Transactions on Graphics* 22, 3 (2003), 917–924.
- [BTL90] BERGER M., TROUT T., LEVIT N.: Ray tracing mirages. *IEEE Computer Graphics and Applications* 10, 3 (1990), 36–41.
- [CHH02] CARR N. A., HALL J. D., HART J. C.: The ray engine. In *Proceedings of the Eurographics/SIGGRAPH Workshop on Graphics Hardware* (2002), pp. 37–46.
- [CHH03] CARR N. A., HALL J. D., HART J. C.: GPU algorithms for radiosity and subsurface scattering. In *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware* (2003), pp. 51–59.
- [Gla93] GLASSNER A. S. (Ed.): *An Introduction to Ray Tracing*, 4th ed. Academic Press, London, 1993.
- [GPG04] GPGPU: General-Purpose Computation on GPUs. Web Page: <http://www.gpgpu.org>, 2004.
- [Gro93] GROSS F.: *Relativistic Quantum Mechanics and Field Theory*. John Wiley & Sons, New York, 1993.

- [Grö95] GRÖLLER E.: Nonlinear ray tracing: Visualizing strange worlds. *The Visual Computer* 11, 5 (1995), 263–276.
- [HBSL03] HARRIS M. J., BAXTER W., SCHEUERMANN T., LASTRA A.: Simulation of cloud dynamics on graphics hardware. In *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware* (2003), pp. 92–101.
- [HMG03] HILLESLAND K. E., MOLINOV S., GRZESZCZUK R.: Nonlinear optimization framework for image-based modeling on programmable graphics hardware. *ACM Transactions on Graphics* 22, 3 (2003), 925–934.
- [HW01] HANSON A. J., WEISKOPF D.: Visualizing relativity. *SIGGRAPH 2001 Course #15 Notes*, 2001.
- [Kaj86] KAJIYA J. T.: The rendering equation. *Computer Graphics (SIGGRAPH '86 Proceedings)* 20, 4 (1986), 143–150.
- [KW03] KRÜGER J., WESTERMANN R.: Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics* 22, 3 (2003), 908–916.
- [KWR02] KOBRAS D., WEISKOPF D., RUDER H.: General relativistic image-based rendering. *The Visual Computer* 18, 4 (2002), 250–258.
- [Lor63] LORENZ E. N.: Deterministic nonperiodic flow. *Journal of the Atmospheric Sciences* 20 (1963), 130–141.
- [MTW73] MISNER C. W., THORNE K. S., WHEELER J. A.: *Gravitation*. Freeman, New York, 1973.
- [Nem93] NEMIROFF R. J.: Visual distortions near a neutron star and black hole. *American Journal of Physics* 61, 7 (July 1993), 619–632.
- [NRHK89] NOLLERT H.-P., RUDER H., HEROLD H., KRAUS U.: The relativistic “looks” of a neutron star. *Astronomy and Astrophysics* 208 (1989), 153.
- [PBMH02] PURCELL T. J., BUCK I., MARK W. R., HANRAHAN P.: Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics* 21, 3 (2002), 703–712.
- [PDC*03] PURCELL T. J., DONNER C., CAMMARANO M., JENSEN H. W., HANRAHAN P.: Photon mapping on programmable graphics hardware. In *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware* (2003), pp. 41–50.
- [Rös76] RÖSSLER O.: An equation for continuous chaos. *Physics Letters A* 57, 5 (1976), 397–398.
- [Wei72] WEINBERG S.: *Gravitation and Cosmology: Principles and Applications of the General Theory of Relativity*. John Wiley & Sons, New York, 1972.
- [Wei00] WEISKOPF D.: Four-dimensional non-linear ray tracing as a visualization tool for gravitational physics. In *Proceedings of IEEE Visualization* (2000), pp. 445–448.
- [YOH00] YNGVE G. D., O'BRIEN J. F., HODGINS J. K.: Animating explosions. In *Proceedings of SIGGRAPH 2000 Conference* (2000), pp. 29–36.