Hardware Acceleration Techniques for 3D Urban Field Strength Prediction

Michael Reyer, Tobias Rick, Rudolf Mathar

Institute for Theoretical Information Technology
RWTH Aachen University
D-52074 Aachen, Germany
Email: {reyer,rick,mathar}@ti.rwth-aachen.de

***Abstract*— Planning and optimization of radio networks are active research areas. Therefore, both fast and accurate radio wave propagations predictions are required. To fulfill those requirements we propose specialized algorithms on ordinary graphics cards. We present an efficient algorithm for determining the visibility between objects. Therefore, we exploit the discrete pixel structure on graphics processing units (GPU). This leads to a tremendous acceleration of up to 140 times compared to a version on a CPU.**

I. Introduction

Accurate radio wave propagation predictions are a prerequisite for effective dimensioning of cellular radio networks. In [4], [5] ordinary graphics cards are used to accelerate the calculation of certain wave guiding effects. Automatic cell planning algorithms have to explore a vast amount of network configurations to find an optimal solution. Hence, on one hand the reliability of cell planning solutions is strongly influenced by the prediction quality of the propagation algorithms. On the other hand, planning time is directly dominated by the runtime of the underlying field strength prediction algorithms. Therefore, low computation times of accurate field strength predictions are essential for automatic cell planning algorithms to find appropriate solutions. An overview of radio wave propagation models is given in [1], [2].

In spite of their very poor prediction quality, statistical propagation models are frequently used, due to short runtimes. This may be sufficient for simple rural environments. However, complex urban scenarios demand for a great emphasis on site-specific details in the propagation environment which are not covered by such statistical approaches. Ray tracing algorithms compute paths through a scene due to wave guiding effects like reflection and diffraction and are well-known to achieve extremely accurate prediction results at the cost of very large runtimes. To cope with high runtimes, usually all necessary propagation predictions are precomputed and stored in large databases [3], to be accessed later by planning algorithms.

Ray optical models are classified in ray tracing and ray launching, depending on the way the ray paths are determined. In ray tracing models all possible paths from receiver point to transmitter point are searched. Multiple calculations of nearly identical ray path pieces are needed, particularly, if receiver points are located nearby. Ray launching approaches [6], [7] emit a finite set of rays in predetermined directions. As the rays disperse, important deflection sources may be missed. Alternatively, in [8], [9] 3D cones are used instead of single rays. Beyond this work, mixed models have been investigated, which follow partly rays and partly use empirical parameters, cf. [10]. Additional work on prediction algorithms, which is based on ray optical approaches, can be found in [11], [12].

In [7] a cube oriented ray launching algorithm (CORLA) has been proposed to counteract the high runtimes of classical ray tracing algorithms while maintaining high prediction accuracy. This has successfully reduced runtimes down to roughly 10 seconds for a 7 km^2 urban area with a mean squared error (MSE) of less than 7 dB. The key idea of this ray launching algorithm is to represent urban environments by a grid of discrete blocks. Costly ray-object intersections are then replaced by traversing those blocks via an algorithm which samples a continuous line into discrete components.

Graphics cards for personal computers offer a very high computing power (up to 300 GFLOPS). In this paper we therefore propose to exploit such hardware in order to accelerate the above mentioned ray launching algorithm. Our current results indicate that runtimes are significantly reduced by this approach.

The advantages of the enormous runtime reduction is twofold. First, field strength predictions can be delivered to cell planning algorithms on demand, i.e., there is no longer a need for precomputations. Second, statistical propagation models can be replaced by accurate ray launching algorithms improving overall results, without increasing planning time.

This paper is organized as follows. In Section II the basic structure of graphics cards is described. After introducing the model for radio wave propagation in Section III, we explain the underlying principles of determining the necessary information for evaluating the system model in Section IV. The transfer of those principles onto graphics hardware is given in Section V. Finally, Section VI provides results and concludes this work.

II. Graphics Hardware

In the last few years, the programmable graphics processing units have evolved into an extremely powerful computing

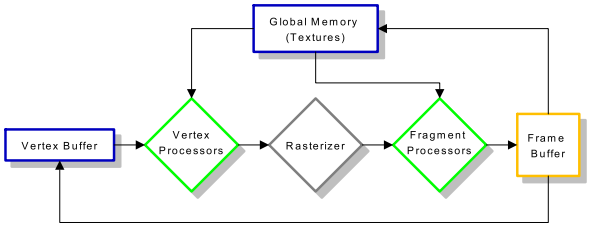

Fig. 1. The Graphics Rendering Pipeline.

device. The main reason for the high throughput is that the *Graphics Processing Unit* (GPU) is specialized for computational intensive, highly parallel calculations. That is, the GPU is especially designed to support data processing, rather than data caching and flow control as is the *Central Processing Unit* (CPU). Thus, the architecture of graphics cards is a *Single Instruction Multiple Data* (SIMD) architecture, i.e., many parallel processors simultaneously execute the same instructions at a time on different parts of data. In order to perform radio wave propagation on graphics hardware the key challenge is to correctly map the problem related tasks to the graphics rendering context.

The programming paradigm of today's graphics hardware is best described by the stages of the *Graphics Rendering Pipeline* (Fig. 1). The input of the pipeline consists of planar geometric objects (triangles or quadrangles) which are described by their coordinates (*vertices*) and additional arbitrary numerical information (*textures*). In the first processing step of the rendering pipeline, multiple *vertex processors* execute in parallel the instructions from a user-written program on the vertices. Commonly, geometric transformations like translations and rotations are applied here.

In the subsequent step, the processed geometry is sampled (*rasterized*) into discrete points (*fragments*). Each fragment has a pixel position on the screen, a depth value and additional data. Analogous to the vertex processors, multiple *fragment processors* execute a user-written program on each fragment in parallel, producing the final result of the GPU computation. Usually, the output consists of a three-dimensional vector which is commonly interpreted as color information.

Finally, all fragments are collected and recorded in the *frame buffer*. If multiple fragments are mapped to the same pixel position, the *depth test* specifies which one is written into the frame buffer by evaluating the fragments' depth values. One cycle through the rendering pipeline is called *rendering pass*. For more details on the programming of today's graphics hardware see [13].

III. Radio Wave Propagation Model

The well-known free space propagation model (cf. [14]) describes the *received power* $P_r(d)$ of a receiver antenna with distance d to a transmitter antenna when they have an unobstructed, clear *line of sight* (LOS) path as

$$P_r(d) = \frac{P_t G \lambda^2}{(4\pi)^2 d^2} \tag{1}$$

with transmitted power P_t, antenna gains G and wavelength λ. The corresponding *path loss* in dB is than given by

$$PL^{\text{dB}} = 10 \log_{10} \left(\frac{P_t}{P_r(d)} \right). \tag{2}$$

The path loss prediction in regions with *no line of sight* (NLOS) is known to be more complex. In common radio frequencies wave guiding effects like reflection, diffraction and scattering typically have a substantial effect on the radio wave attenuation. As we are interested in an average received power we neglect the influence of multipath effects and concentrate on the path with the dominant contribution. The resulting error is rather low as the following example illustrates. Consider two equally strong ray paths, if their contribution is added in logarithmic scale the overall error of the approximation is less than three dB. This error is negligible, regarding since mean squared errors of 5 to 10 dB are considered as excellent, see [1]. Applying this approximation reduces computation time significantly.

We take into account different wave guiding effects by introducing distinct attenuation functions. Let α be the change of direction of the corresponding deflection, than the attenuation functions due to reflection, vertical diffraction and horizontal diffraction are denoted by $PL_R^{\text{dB}}(\alpha)$, $PL_V^{\text{dB}}(\alpha)$ and $PL_H^{\text{dB}}(\alpha)$. In our model the path loss at a point in NLOS is evaluated by taking the strongest ray path and adding its attenuations due to the corresponding wave guiding effects

$$\begin{aligned} PL_{\text{NLOS}}^{\text{dB}} = & PL^{\text{dB}} + \sum_{i=1}^{N_R} PL_R^{\text{dB}}(\alpha_{R,i}) \\ & + \sum_{j=1}^{N_V} PL_V^{\text{dB}}(\alpha_{V,j}) + \sum_{k=1}^{N_H} PL_H^{\text{dB}}(\alpha_{H,k}), \end{aligned} \tag{3}$$

where N_R, N_V and N_H denote the number and $\alpha_{R,i}$, $\alpha_{V,j}$ and $\alpha_{H,k}$ depict the corresponding direction changes of deflection points for each type of wave guiding effect.

IV. Cube Oriented Ray Launching

The cube oriented ray launching algorithm, CORLA for short, is applicable for determining the ray path information needed for evaluating the system model of the preceding section. The main idea is to rasterize the given environment into cubes. If the center of a cube lies within a polyhedron describing the environment the cube is marked as *filled*. Otherwise, it is marked as *empty*. If a cube intersects with a surface section of a polyhedron it is called a *reflection source*. Additionally, a reflection source is a *horizontal or vertical diffraction source* if it intersects with the boundary of a surface section. Cubes below ground level are also marked as filled. Given two points p_1 and p_2, p_2 is called *visible to* p_1 if no

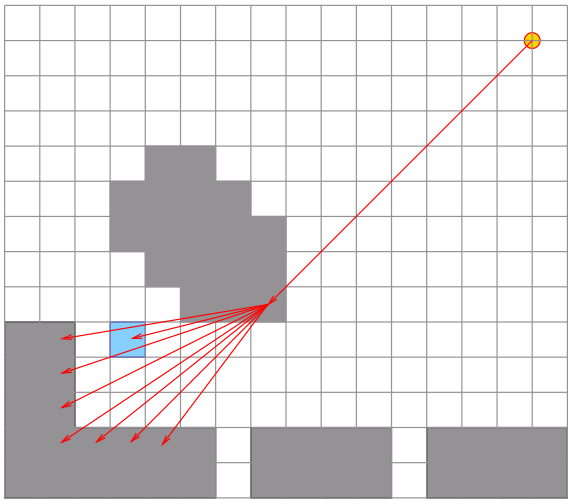

Fig. 2. Cube Oriented Ray Launching Principle

cube on the straight line between p_1 and p_2, other than the cubes containing either p_1 or p_2, is filled. Fig. 2 illustrates this concept.

The main task in CORLA is to determine the set of cubes visible to a source p. If p represents a source of deflection the search for visible cubes is limited to the corresponding deflection cone. The task of determining visible cubes can be accomplished relatively easy and accurate when exploiting the cubical representation of the environment. The required density for hitting all cubes with potential deflection sources can easily be determined. Hence, deflection sources are only ignored if the cube resolution is chosen too low. Determining visibility is than achieved by traversing the discretized (sampled) three-dimensional direct line between source and destination point and look up if there is a filled cube in between. Line discretization can be efficiently performed by the well-known Bresenham-algorithm, cf. [15].

V. Ray Launching on the GPU

The key operation in CORLA is the sampling of cubes along a certain ray or cone (e.g. diffraction cone). In the standard CPU implementation this sampling is achieved with the well-known Bresenham-algorithm [15]. Basically, a continuous line segment is represented by a set of pixels.

According to the ray launching principle, the cubes of a ray are processed, beginning from the start point until a filled cube is reached. Hence, from an algorithmic point of view, we simply generate the sampling of cubes plus an additional memory access at every cube to check for a filled one. This memory access presents the main obstacle when implementing this algorithm on the graphics hardware, since the number of memory accesses (*texture fetches*) in a shader program is limited. Hence, this algorithm cannot directly be implemented as a fragment shader program.

In order to cope with the rather large amount of memory

Algorithm 1 ProcessSuperCubes()

```
T ←getTransmitterPos()
M ←getMaxAllowedMemAccesses()
setQuadSize(M)
clearBuffer(A, B)
enableShader(TraceRay() )
for i = 0 ... maxSteps do
  setDrawBuffer(A)
  fillBufferWith(B)
  for all sc ∈ getNeighboringSuberCubes(T, i) do
    setBoundaryCoords(T, sc)
    drawQuad(sc) {TraceRay() is invoked on every pixel
    of this quad}
  end for
  swap(A, B) {Update results for next rendering pass}
end for
disableShader(TraceRay() )
```

Algorithm 2 TraceRay()

```
T ←getTransmitterPos()
M ←getMaxAllowedMemAccesses()
l ←getBoundaryCoords()
r ←getFragmentPos()
{Determine new start point q}
d ←normalize(r − T)
λ ←max {(M · l.x)/d.x, (M · l.y)/d.y}
q ←T + (λ − 1)d
{Continue with Bresenham-algorithm from q to r}
bresenhamLine(q,r)
```

accesses, we propose the following *multi-pass* approach. We execute multiple subsequent rendering passes such that rendering pass $i+1$ can access the results from rendering pass i. The idea is sketched in Algorithm 1.

The goal is to sample a line from a source point T to every other point. Let the maximum number of allowed memory accesses per shader invocation be M. Hence, we can only traverse lines with at most M pixels length per shader instance.

The algorithm operates on a grid of cubes (pixels). We define a *supercube* to be a quad with edge length of M pixels. Then a set of supercubes is constructed such that T lies on the corner of four supercubes (as illustrated in Fig. 3 top left) and all supercubes are a partition of the whole area.

Then the algorithm proceeds as follows. Supercubes are processed by simply rendering a quad of the same size as the supercube. On rendering, a fragment shader (cf. Algorithm 2) is invoked on every pixel that belongs to the quad. The shader program traverses the pixels from a start point q to its corresponding fragment position r and passes the information if a filled cube has been reached.

The coordinates of the vertical and horizontal *boundary lines* are given as those coordinates of the current supercube sc, which are nearest to the transmitter T. Thus, in the first loop of Algorithm 1 the boundary coordinates and the start point q are of course given by the transmitter coordinates itself. In the subsequent steps of Algorithm 1 the intersection

point between the line from the position of the fragment r to the transmitter T and the boundary lines are determined in Algorithm 2. The intersection point with the larger distance to the transmitter determines the new starting point q, which is a direct neighbor of the current supercube sc. This cube contains the information, if a filled cube has been seen on the way from the transmitter point T to q, i.e., if the direct path is obstructed yet.

Then a standard Bresenham-algorithm can be applied in the shader to update the cube-information inside the supercube, if there is a filled cube on the way. As the edge length of each supercube is M, it is ensured that there will be at most M texture fetches to check for filled cubes for each shader instance.

This procedure is repeated as sketched in Algorithm 1 until all supercubes are processed.

VI. Results

For comparison of the runtimes of line rasterization on CPU and GPU hardware we have implemented the rasterization for both. The reference scenario consisted of a 512×512 grid where we have sampled rays from the center to every point in the grid. At each sampling step we performed a memory access and a summation of four floating point values. Total runtime on the CPU (Intel(R) Xeon(TM) CPU 2.40GHz) was about 1.24 seconds. The GPU (NVIDIA GPU GeForce 8800 GTX) implementation exhibits a runtime of only 8.93 milliseconds. Hence, as Fig. 4 illustrates, the CPU version with roughly 211,000 rays per second is clearly outperformed by orders of magnitude by the GPU which was able to process about 29 million rays per second, which is approximately 140 times faster than the CPU version. Note, the grid size has been chosen, such that the number of iteration ($maxSteps$) in Algorithm 1 is zero. Otherwise the GPU version would benefit even more, because pixels outside the four supercubes are not followed to the transmitter but only to the neighboring supercube. We conclude that ray launching based on graphics hardware can significantly speed up radio wave propagation predictions.

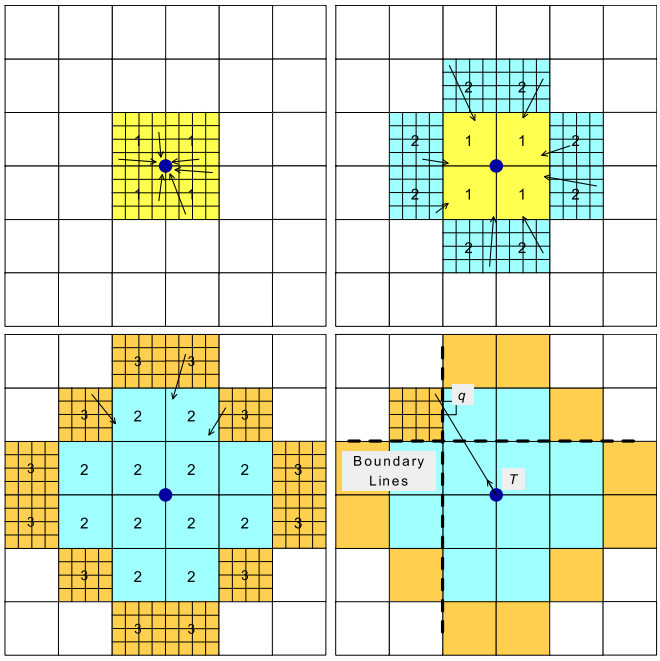

Fig. 3. Processing of supercubes and boundary line

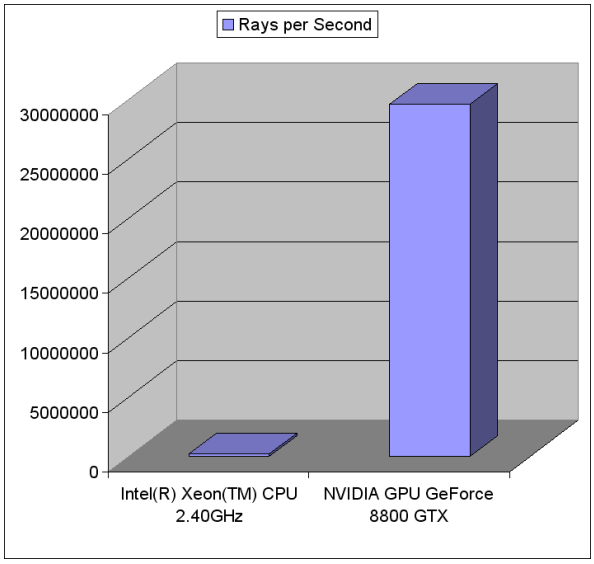

Fig. 4. Rays per second, comparison CPU vs. GPU

References

[1] E. Damosso, Ed., *COST Action 231: Digital mobile radio towards future generation systems, Final Report*. Luxembourg: Office for Official Publications of the European Communities, 1999.

[2] N. Geng and W. Wiesbeck, *Planungsmethoden für die Mobilkommunikation*. Berlin: Springer, 1998.

[3] G. Wölfle, R. Hoppe, and F. Landstorfer, "A fast and enhanced ray optical propagation model for indoor and urban scenarios, based on an intelligent preprocessing of the database," in *Proceedings PIMRC*, Osaka, Japan, 1999.

[4] D. Catrein, M. Reyer, and T. Rick, "Accelerating radio wave propagation predictions by implementation on graphics hardware," in *Proceedings: IEEE VTC Spring*, 2007.

[5] T. Rick and R. Mathar, "Fast edge-diffraction-based radio wave propagation model for graphics hardware," in *ITG INICA*, 2007.

[6] G. Durgin, N. Patwari, and T. S. Rappaport, "An advanced 3D ray launching method for wireless propagation prediction," in *Proceedings IEEE VTC Spring*, Phoenix, AZ, 1997, pp. 785–789.

[7] R. Mathar, M. Reyer, and M. Schmeink, "A cube oriented ray launching algorithm for 3D urban field strength prediction," in *Proceedings IEEE International Conference on Communications*, Glasgow, Scotland, June 2007.

[8] M. Nidd, S. Mann, and J. Black, "Using ray tracing for site-specific indoor radio signal strength analysis," in *Proceedings IEEE VTC Spring*, Phoenix, AZ, 1997, pp. 795–799.

[9] T. Frach, "Adaptives hierarchisches Ray Tracing Verfahren zur parallelen Berechnung der Wellenausbreitung in Funknetzen," Ph.D. dissertation, RWTH Aachen University, 2003.

[10] J. Beyer, "Ausbreitungsmodelle und rechenzeiteffiziente Methoden für die Feldstärkeprognose in städtischen Mikrozellen," Ph.D. dissertation, Universität-Gesamthochschule Siegen, 1997.

[11] R. Wahl, G. Wölfle, P. Wertz, P. Wildbolz, and F. Landstorfer., "Domi-

nant path prediction model for urban scenarios," in *14th IST Mobile and Wireless Communications Summit*, 2005.

[12] P. Wertz, R. Wahl, G. Wölfle, P. Wildbolz, and F. Landstorfer., "Dominant path prediction model for indoor scenarios," in *German Microwave Conference (GeMiC)*, 2005.

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

[14] T. S. Rappaport, Ed., *Wireless Communications: Principles and Practice*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1995.

[15] J. E. Bresenham, "Algorithm for computer control of a digital plotter," *IBM Systems Journal*, vol. 4, pp. 25–30, 1965.