Real-time Visualization of Wave Propagation

Arne Schmitz and Leif Kobbelt
Computer Graphics Group
RWTH Aachen University
{aschmitz,kobbelt}@cs.rwth-aachen.de

Abstract: In this work we present a method to visualize the wave propagation mechanisms of wireless networks at interactive rates. The user can move around transmitting nodes and immediately sees the resulting field strength for the complete scenario. This can be used for rapid optimization of antenna placement, or for visualizing the coverage of mobile stations, as they move through a simulation. In a preprocessing step we compute the wave propagation for a distinct set of transmitter positions. Whereas in the visualization phase we use these precomputed maps to do a fast interpolation using a current graphics card.

1 Introduction

The computation of wave propagation for microwave transmitters is a well researched topic and of interest in different areas of the networking community. First we can use wave propagation prediction to optimize the placement of antennas for fixed transmitter stations. In many cases it is not practical to tentatively setup a transmitter and change its location if area coverage turns out to be insufficient. Therefore a simulation of the pathloss for the area of interest is needed to choose an optimal transmitter location.

Second, wave propagation prediction is of interest when simulating mobile wireless networks. Here we want to know the pathloss for two possibly moving stations. This usually requires recomputing a lot of propagation paths every time a sender moves and thus prohibits a real-time evaluation of the simulation. We will show how this can be avoided.

1.1 Related work

Different methods of wave propagation are for example described in [SM00], [WB88], [WWW$^+$05b], [WWW$^+$05a]. These methods use different approaches for the prediction of the field strength. In general radio waves can be described by geometrical optics, as a propagation along rays. Every time a ray hits an object, it will be scattered, reflected, transmitted, diffracted or absorbed. This leads to quite complex propagation patterns, all of which should be modelled correctly to get a good result. This behavior can be achieved by different algorithms that are either based on empirical or physical models. Both differ in computation complexity and the accuracy that can be achieved.

First there are empirical models, that try to fit a more or less simple function to the pathloss. One of the most widely used empirical models is the Walfish-Ikegami-Model [WB88]. It takes into account multiple rooftop-to-street diffractions, depending on the distribution of buildings. This is a good approximation for microcellular urban scenarios where rooftop diffraction is dominant. Although this model is very fast to compute, it fails to represent propagation paths that are produced by reflection on building walls. Therefore this model is only partially useful.

Most of the other algorithms fall into the category of deterministic models. These methods compute the wave propagation by tracing rays through the scene from which the pathloss can be derived. Usually this can be done either from the radiation source, or from the receiving position. Ray launching algorithms shoot rays from the sender, while ray tracing algorithms go the other way around. The problem with these models is that they require an enormous amount of rays for a decent estimate of the pathloss. Therefore most algorithms take half a minute to about an hour for the computation of the pathloss at a single transmitter position, considering every point in the scene. Interactive simulation rates with moving senders are thus not possible.

One example of a ray launching algorithm is given in [SM00]. Here the rays are traced through a uniform grid to produce a pathloss sampling. A hybrid model that uses both ray propagation techniques and an underlying empirical model is described in [WWW$^+$05b] and [WWW$^+$05a]. This so called dominant path prediction is able to cut down on the computation complexity, so that one simulation can be computed in about 30 seconds to a couple of minutes at most. Yet this is still not enough for interactive visualization.

1.2 Overview of the algorithm

In our work we also use a ray launching algorithm to produce a pathloss image of the scene, called the Photon Path Map [SK06]. It does not really matter which algorithm one chooses, since the visualization step is independent from the chosen propagation algorithm. We use a precomputed set of pathloss images to interpolate on the fly a new image for the updated sender position. It will be shown that this can be done in real time using current graphics hardware that is able to do the interpolation step.

The structure of the algorithm is very simple and straightforward. First, we let the user specify the scene geometry and mobile transmitter properties. This includes the different materials for buildings and walls, as also the frequency and power of the senders used. Following this, the user can specify an arbitrary number of points that are used for the precomputed pathloss maps.

The next step is to use a ray tracing algorithm to compute the pathloss images for the specified sender points. These maps are uploaded to the graphics card, where a new image can be interpolated. Doing this via conventional methods using the main CPU is too costly and results in non-interactive response times of the visualization tool.

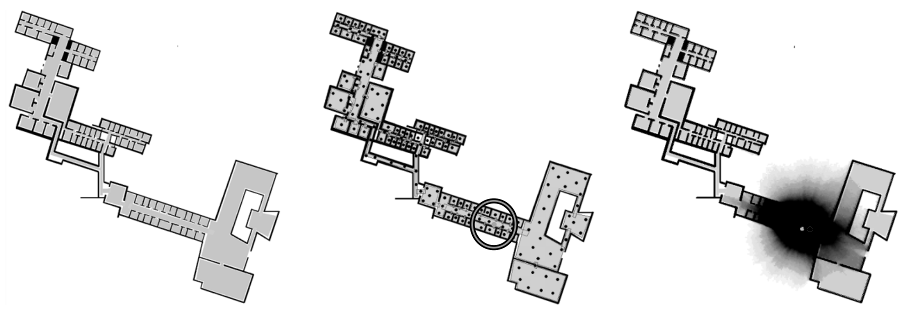

Abbildung 1: On the left the geometry specified by the user can be seen. Then some seed-points are created and initial pathloss maps computed. For a new sender point we choose the nearest maps (i.e. the ones in the circle) and interpolate them to get a new pathloss map (right). The example is two dimensional, but the scenarios can also be specified in three dimensions.

2 Simulation map generation

The precomputation of the pathloss- or simulation-maps consists of two steps. First the user has to specify the scenario with all parameters and has to supply positions of the initial simulation maps. Second the ray-tracing program has to generate the pathloss-maps. The latter part is done by using an algorithm called the Photon Path Map, which is related to the Photon Map known from computer graphics [JC95].

2.1 Geometry specification

The scene consists of three dimensional polygonal meshes, in particular triangular meshes. They can be specified by the user through an editing tool or exported using standard 3D software. Furthermore the user supplies material properties for the building walls, like reflectivity and transparency. These values can be taken from direct material measurements or parameters can be fitted for a scene using a set of on-site measurements. In our tests we have used a map of our institute's building and fitted the material parameters according to measurements taken inside the building. The model can be seen in Figure 1. The same image also shows the initial sender positions that are specified by the user.

2.2 Photon Path Map generation

For the computation of the pathloss-maps we use an algorithm called the Photon Path Map. It is derived from the Photon Map algorithm, which is widely used in Computer Graphics for illumination purposes [JC95]. For details on the Photon Map we refer to the original work. Here we just sketch out how our extension to the algorithm works.

Similar to the original Photon Map approach, we trace photon particles emanating from the sender through the scene. When the photon hits an object, it is either reflected, transmitted or absorbed. Also photons passing close to object corners might get diffracted. Every time such an interaction happens, we store a new path segment in our Photon Path Map, noting the trajectory of the photon. After shooting many thousands of these photons, we have a data structure that has a record of all photon trajectories (or paths) through the scene.

Initially one single photon represents $\frac{1}{n}$th of the flux emitted from the sender, if the sender has emitted n photons. To get an estimate for the pathloss at a given point in space, we need to make a density estimation over the photons. We do this by modelling a kernel density estimator that integrates over all the photons passing through or near the same point in space. How this is done explicitly is beyond the scope of this paper and is described in more detail in [SK06]. We just note that this is done in a matter of seconds to a few minutes at most and leaves us with a volume image containing the pathloss in the scene for one given sender position.

3 Real-time display

The goal of this work is to be able to visualize the radio wave propagation of moving senders in real-time. In the last section we have seen how to precompute a set of wave propagation simulations. So now we have a set of points p_i with an associated pathloss map m_i (in the following also called volume image). These maps are defined as a set of voxels $v_{jkl} \in \{0..X\} \times \{0..Y\} \times \{0..Z\}$, where X, Y and Z are the number of voxels of each dimension of the 3D pathloss map. A voxel is a discrete cell in a uniform three-dimensional grid, similar to the pixel in a two-dimensional image.

3.1 General interpolation

It is now an obvious step to interpolate these known data points to get a new one. For this we sort all the points into a kd-tree [Ben75] in advance and do a K-nearest neighbor lookup when we want to do an interpolation. This can be done very efficiently and leads us to the following formulation, which is similar to a standard kernel density estimator:

$$m_s = \frac{\sum_{i=1}^{k} \frac{m_j}{\|p_j - p_s\|^d}}{\sum_{i=1}^{k} \frac{1}{\|p_j - p_s\|^d}} \tag{1}$$

Where p_s shall denote the new position of the sender and p_j with $1 \leq j \leq k$ are the K-nearest neighbors of p_s. The m_j are the according pathloss-maps. The value d is a damping factor which influences the weight of more distant neighbors. Typically one would use values in the range of $d = 1$ to 3 for this, where $d = 1$ translates to a linear filter kernel for the sample points.

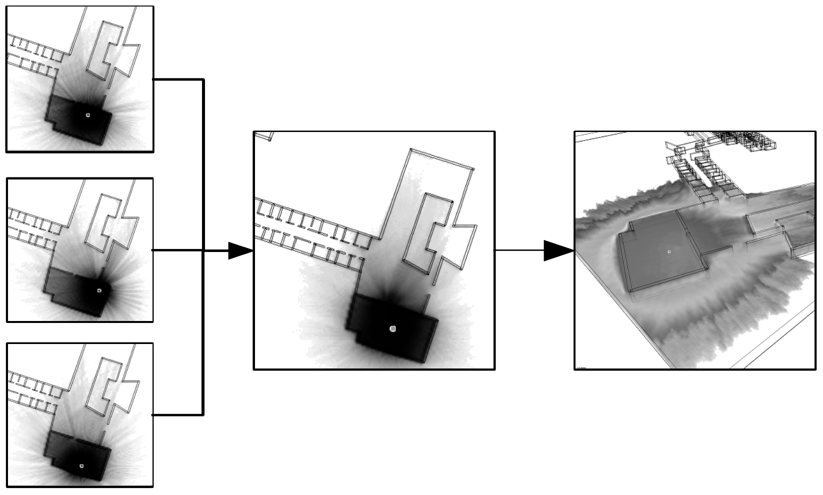

Abbildung 2: First the algorithm selects the nearest matching pathloss maps (left), blends them to a new image (middle) which can then be displayed for interactive viewing (right).

In the above equation, when we write $\frac{m_j}{\|p_j - p_s\|^d}$ the pathloss-image m_j is said to be weighted with the factor $\frac{1}{\|p_j - p_s\|^d}$. It is clear that this can amount to a lot of multiplications, since every voxel of m_j has to be weighted, and then all maps have to be summed up and as a last step the resulting image has to be normalized. Doing all this on the conventional CPU is very slow, since each pathloss map can be up to several megabytes in size. In our tests we achieved approximately 1 to 4 frames per second on a 2.2 GHz machine, which is far from being interactive, let alone real-time. One of the reasons for this procedure being so slow is also the relatively slow upload of texture images onto the graphics card. After each interpolation we have to upload the new 3D texture to the graphics hardware. Once the textures are residing in the GPUs RAM we can manipulate them without great delay. The GPU also has the advantage of a very high-bandwidth local memory access, in our case more than 30 gigabyte per second. This fact, combined with the parallel architecture of a GPU, allows us to overcome the problems that we see when using the main CPU for interpolation.

Additionally a visibility test can be used to select only sample points that are visible from the new sender position. For example it might not make much sense to incorporate a pathloss image in the interpolation, if the new sender position and the interpolation sample are divided by a massive concrete wall. Then we would get a wrong interpolation estimate. The visibility test itself is done efficiently by utilizing the ray tracing algorithm from the preprocessing step. The number of rays needed is constant and the ray tracer scales logarithmically with the complexity of the underlying scene geometry.

3.2 GPU Interpolation

Since interpolation of the maps on the CPU is prohibitively expensive, we can instead do the interpolation on a modern graphics processing unit (GPU). The pathloss-maps are volume images, i.e., three dimensional sets of voxels. Current generation graphics cards can handle three dimensional images with ease and use them as textures for direct visualization using incremental slicing techniques [WE98]. Interpolation on the CPU cannot compete with this technique, because GPUs are highly parallelized. In our case for example we take up to 16 volume images and blend them together. The GPU used is able to do the 16 texture lookups entirely in parallel, whereas a standard CPU can only do them sequentially, or at most do a few lookups in parallel if the volume images are stored interleaved in memory. Also the GPU can process several voxels in parallel, i.e. the weighted adding can happen for more than one voxel. On a CPU this also has to be done sequentially.

With the advent of programmable graphics hardware it has become possible to extend the once fixed rendering pipeline and use it to do the costly image interpolation. The first step is to upload the precomputed pathloss-maps onto the graphics card. There the maps are stored as a set of 3D-textures, which we can use for rendering purposes, also see Figure 2. In our implementation we use OpenGL as the basic graphics programming interface.

The interpolation of the images is done by rendering textured quadrilaterals and using so called fragment programs [LBB03] to do an exact weighting of each texture. The algorithm can be sketched like this:

```
gpu_interpolate(K, w):
  generate new framebuffer
  load fragment program p[K]
  for each 3D texture t[i], i=1..K
    compute weight w[i] and pass to fragment program
    bind 3D texture t[i] to texture unit i
  for each slice in the 3D texture
    render quad using fragment program p[K]
    copy framebuffer into 3D texture for display
```

The weights $w[i]$ in the above code correspond to the weights computed in section 3.1. The fragment program $p[K]$ does K simultaneous texture lookups and blends the values according to the interpolation weights, which are passed as local parameters to the fragment program. The fragment program itself is:

```
!!ARBfp1.0
# Temporary variables for color lookup:
TEMP col0, ..., colK;
TEMP sum;
# Parallel texture lookups:
TEX col0, fragment.texcoord[0], texture[0], 3D;
...
TEX colK, fragment.texcoord[0], texture[K], 3D;
```

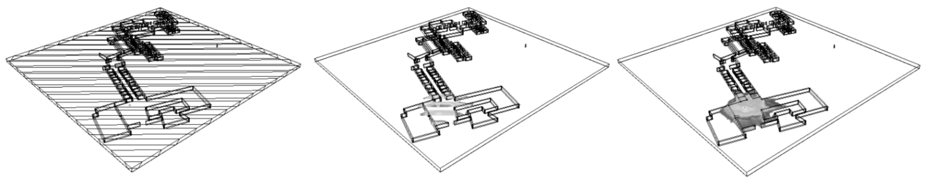

Abbildung 3: A set of screen aligned planes is intersected with the 3D volume resulting in a set of slices (left). Then the slices are textured with data from the 3D volume (middle). Using a high number of slices a continuous appearance can be achieved (right).

```
# Weighted blending:
MUL sum, program.local[0], col0;
MAD sum, program.local[1], col1, sum;
...
MAD sum, program.local[N], colK, sum;
# Return resulting color:
MOV result.color, sum;

END
```

Note that this interpolation happens in only one render pass, which ensures that the blending happens with the highest possible accuracy. Usually graphics cards store values only with 8 bits of accuracy, but when doing arithmetical operations in a fragment shader, one automatically uses floating point arithmetics.

It is also noteworthy that a single render pass can only be achieved, if the GPU used has enough texture units. If we want to do an interpolation of the K-nearest neighbors, the GPU has to provide K texture units, which can process all the volume images in parallel. Otherwise one has to do several render passes. For example if the GPU supplies n texture units and we want to do a K-nearest interpolation, with $K > n$, we need $\lceil \frac{n}{K} \rceil$ render passes. In each pass, we can interpolate n textures in parallel, and we have to incrementally blend all render passes until the interpolation is completed, which gives a performance penalty. However modern graphics cards supply quite a large number of texture units, so that in our experiments we used a K-nearest neighbor interpolation with $K = 16$, on an nVidia GeForce 7800 GTX GPU which also supplies 16 texture units. This is more than enough to get a smooth looking interpolation.

3.3 Volume visualization

For the visualization of the interpolated image we use an algorithm described in [WE98]. It allows us to display a 3D volumetric image on a standard graphics card. The idea is to intersect the 3D volume cube with n screen-aligned planes, which produces a set of n slices through the volume (see Figure 3). The graphics card is able to compute texture coordinates for these slices and use the 3D volumetric image to apply a texture to the slices. When using a high number of slices (i.e. more than 100), a continuous appearance can be achieved.

This rendering is very fast and scales with the number of slices used. If the number of slices is too high, the pixel fill rate of the graphics card will be saturated and the rendering frame rate will drop significantly. However, in our experiments we used between 500 and 1000 slices for rendering the images without a great performance loss. One can say that for window sizes of up to 1000^2 pixels, 500 slices are a good tradeoff between visual quality and efficient rendering.

4 Results

For testing of our method we have used a model of our institute's building with 170 precomputed sender positions. The initial pathloss maps have been generated by a grid of AMD Opteron nodes in less than 15 minutes. Each pathloss map is of size $256 \times 256 \times 4$ and is 256 KBytes in size, using a logarithmic byte encoding for extended dynamic range. This amounts to 42.5 MBytes of space, that are also needed for storage on the graphics card. As one can see, much more sender positions or higher resolutions for the maps are possible, since the used GeForce 7800 GTX is equipped with 512 MBytes of RAM. Additionally the GPU can use up to 1 GByte of main RAM, accessing it through the PCI-Express bus. This way almost 200 pathloss maps of size $256 \times 256 \times 128$ can be used. This is enough for fairly large 3D buildings. In general many scenarios will only be 2.5D and a resolution of $256 \times 256 \times 4$ will be enough.

Using the 170 pathloss maps and 16 nearest neighbors, we achieve the following rendering speeds, when tracing some random movement path through the buildings on a 2.2 GHz computer:

Method	Resolution	Slices	FPS
CPU	$256 \times 256 \times 4$	1000	2.6
GPU	$256 \times 256 \times 4$	1000	102
CPU	$512 \times 512 \times 4$	1000	0.5
GPU	$512 \times 512 \times 4$	1000	45

The main problem here is the fill rate limitation of the GPU when rendering the complete volume to the screen. The interpolation itself can run even faster, when using less slices for the visualization. With 500 slices we get roughly 150 frames per second for the smaller dataset. But obviously the visual quality gets better the more slices are used.

5 Conclusion

In this work we have shown a way to visualize the field strengths of moving senders in real-time using current graphics hardware. Up to now it was not possible to interactively move a sender around and have live feedback on the signal strength in the whole scenario. We have shown that this is possible in a smooth manner using off-the-shelf graphics cards. Compared to the naive approach using the computer's main CPU we get 10 to 50 times the rendering performance, depending on the size of the dataset. Interesting enough the interpolation is much less a bottleneck than is the final rendering of the 3D image.

One possible next step is to combine this with a network simulator to have a live feedback of the simulation in progress. For this multiple sender images might have to be blended together, for example to show interfering ranges of the sending nodes. This would be a good tool for designing, analyzing and understanding wireless networks.

Literatur

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[JC95] Henrik Wann Jensen und Niels Jørgen Christensen. Photon Maps in Bidirectional Monte Carlo Ray Tracing of Complex Objects. *Computers and Graphics*, 19:215–224, March 1995.

[LBB03] Benj Lipchak, Bob Beretta und Pat Brown. ARB Fragment Program Extension. OpenGL Registry, August 2003.

[SK06] Arne Schmitz und Leif Kobbelt. Wave Propagation Using the Photon Path Map. Bericht, Computer Graphics Group, RWTH Aachen University, 2006.

[SM00] M. Schmeink und R. Mathar. Preprocessed indirect 3D-ray launching for urban microcell field strength prediction. In *AP 2000 Millennium Conference on Antennas and Propagation*, April 2000.

[WB88] J. Walfisch und H.L. Bertoni. A theoretical model of UHF propagation in urban environments. *IEEE Transactions on Antennas and Propagation*, 36(12):1788–1796, December 1988.

[WE98] Rüdiger Westermann und Thomas Ertl. Efficiently using graphics hardware in volume rendering applications. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, Seiten 169–177, New York, NY, USA, 1998. ACM Press.

[WWW$^+$05a] R. Wahl, G. Wölfle, P. Wertz, P. Wildbolz und F. Landstorfer. Dominant Path Prediction Model for Urban Scenarios. *14th IST Mobile and Wireless Communications Summit, Dresden (Germany)*, 2005.

[WWW$^+$05b] P. Wertz, R. Wahl, G. Wölfle, P. Wildbolz und F. Landstorfer. Dominant Path Prediction Model for Indoor Scenarios. *German Microwave Conference (GeMiC) 2005, University of Ulm*, 2005.