Graz University of Technology
Institute of Computer Graphics &
Knowledge Visualization

Bachelor Thesis
Ray Tracing Point Clouds

November 18, 2011

Christoph Wiesmeier 0730989

c.wiesmeier@student.tugraz.at

Supervisor: Dr. Dipl.-Math T. Ullrich and Dipl.-Ing. T. Schiffer

Statutory declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

....
date (signature)

Abstract

Currently most common rendering technologies are designed to render triangle based geometry. These technologies are usually not appropriate for rendering infinitesimal small objects like points. To provide a renderer suitable for point based models we introduce a ray tracing based implementation which supports interactive rendering utilizing a CUDA compatible graphics processor. We investigate spheres and discs to solve the problem of intersecting infinitesimal small points. Furthermore, we evaluate different methods to increase the shading quality using intersection with multiple points on the surface. Finally, we discuss methods for normal estimation and point set reduction.

1 Introduction

In the last years technologies like laser scans and 3D reconstruction from images became more and more popular. They result in point models, which represent samples of the surface, containing several millions of points. These point-based models can't be rendered in a satisfying way using conventional rendering pipelines like OpenGL [9].

The main problem when rendering such models is that a point is infinitesimal small and therefore has no surface. Due to this fact, rasterization will not produce a realistic image. Also a standard ray tracing attempt will fail because an intersection of a ray with a point is very unlikely. Furthermore typical point models contain no direction information which is the basis for all common lighting calculation.

This work presents a ray tracing approach to render point models. The implementation is based on the OpenGI framework [8] and supports interactive rendering. Most of the computation is done on the Graphics Processing Unit (GPU) using the Nvidia Compute Unified Device Architecture (CUDA) [5]. Furthermore, a preprocessing tool which supports different methods for point set reduction and normal estimation is introduced.

Contrary to other implementations the presented solution is capable of rendering point and triangle data simultaneously. Additionally, the design allows rendering of animated scenes.

The remainder of this thesis is structured as followed. In Section 2 we discuss some other implementations. Section 3 explains the basics of ray tracing and introduces the used OpenGI framework. Rendering approaches for point models based on ray tracing are introduced in Section 4. Section 5 explains methods for point set reduction and normal estimation. Finally Section 6 summarizes the results and discusses topics of further work.

2 Related Work

One of the fundamental works is presented by G. Schaufler et al. [7]. They used an approach based on cylindric rays, and some sort of density test to search for intersections. If an intersection is detected, they use the points and their precomputed normals within a short part of the cylinder to calculate the intersection point and surface normal. Another interesting approach using GPU computing has been presented by K. Sriram [4]. He investigated splatting approaches similar to this work and further used implicit surface octrees, to avoid artifacts in areas of high bending. A concept based on point set surfaces is presented by A. Adamson and M. Alexa [2]. They show that high quality images can be generated by using higher order functions to define the surface. Therefore iterative algorithms based on local polynomial approximations were used to find intersections. However compared to splatting this leads to increased render times. Another approach which is worth mentioning because of its efficiency, is presented by I. Wald and H. Seidel [10]. Their approach is able to render large scenes in real time on the CPU. Their final solution is based on an implicit surface model.

There are a lot of other approaches to render point based models. Especially there

are some methods based on surface reconstruction using triangles, which are beyond the scope of this thesis.

3 Ray Tracing Basics

Ray tracing is a rendering technology based on fundamental optical laws. This allows the generation of high quality images including effects like shadows, reflection, refraction and transparency. This introduction is focused on Whited style ray tracing [11].

Contrary to the common optical model the rays are tracked backwards from the camera towards the lights. For each pixel on the view plane a ray is started. The rays are intersected with the objects of the scene. Based on the resulting intersection points new rays are started to handle shadows, reflection, refraction and transparency. The contribution of each ray to the intensity and color of the pixel is defined by the material of the intersected object. To achieve an accurate representation this is done recursively for all rays except the shadow rays.

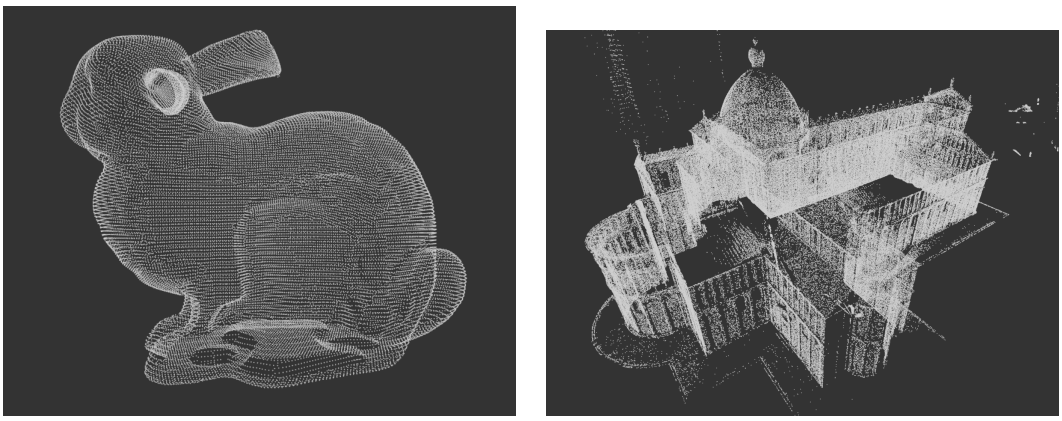
Ray Tracing Framework

This work is based on the ray tracer implemented in the OpenGI framework [8]. OpenGI is capable of rendering scenes using different back ends like a rasterizer or a ray tracer. The scene handling uses the scene graph library OpenSG [6], which includes a direct OpenGL back end. The ray tracer of OpenGI performs most calculations on the GPU using the CUDA toolkit. As an acceleration structure a bounding volume hierarchy is used. Due to the restrictions of CUDA regarding recursion there are some limitations in the lighting model (e.g. only 3 levels of indirect light are supported). The intersections are searched along a ray within the distance range of $[tmin, tmax]$, where $tmin$ is used to avoid self intersections and $tmax$ is used to exclude objects behind the nearest intersection. The bounding volume hierarchy is traversed using a stack-based depth first algorithm.

4 Rendering

One method for rendering points is using a rasterizer like OpenGL. The projection of a point to the view plan is infinitesimal small, usually it is rounded up to one pixel. The resulting pixels are set to the material color (Figure 1). To provide a fundamental lighting model a normal vector can be given to each point which allows to compute the intensity of the point-based on the light direction. This solution permits no shadow calculation and is also not capable of generating closed surfaces.

As shown in Figure 1 rendering points using rasterization does not result in satisfying images. To work around this problem there are approaches to reconstruct the surface rather than rendering the points directly. One approach is to build a triangle mesh between the points. Reconstruction is quite complex and time consuming but allows using a standard rasterization based renderer.



(a) Standford Bunny [1] (b) Laser scan Pisa (300k Points)

Figure 1: The images show datasets rendered using OpenGL, without using normal-based lighting. The points are represented using OpenGL points. Each data point is rendered as a single pixel.

Another well known rendering approach is to use ray tracing. This method has two major advantages. First it is capable of calculating shadows, reflection, refraction and transparency. Second the time complexity of ray tracing using a tree based acceleration structure is usually proportional to $p \cdot log(n)$ with n the number of points and p the number of pixels of the image. This allows handling of big datasets with interactive performance as can be seen in Section 4 (Performance).

Ray Tracing Point Models

Ray tracing is a simple but powerful technique which can be split into ray-scene intersection and the lighting calculation which uses the results. The ray-scene intersection must be adapted for each object type. Because an intersection test with an infinitesimal small point would not provide meaningful results, we need to find an appropriate representation to use instead. The lighting calculation does not directly depend on the object type. The only information required is the orientation of the object, which is represented by a surface normal. This information is no integral component of a point, but for most algorithms we assume it is part of the data set.

Spheres

The simplest approach is to replace points by spheres for intersection. The replacement requires additional information about the size of the spheres. This information is determined during the preprocessing of the data set. Rendering results using spheres can be seen in Figures 2 and 3. This method is not capable of generating a flat surface. Furthermore the resulting surface is shifted nearly one radius in front of the actual sampled surface. The fact that the points are visible can be useful to analyze the dataset.

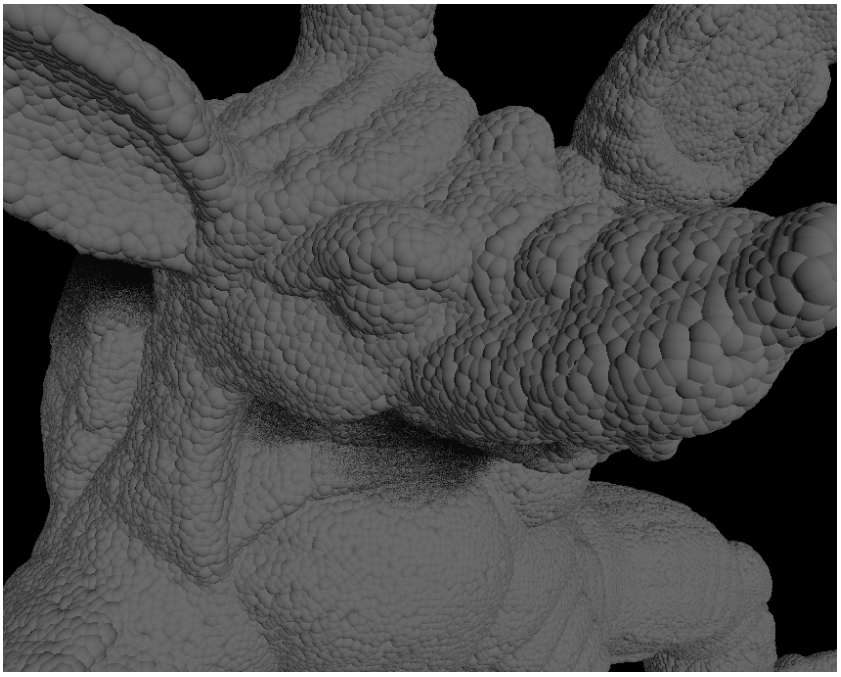


Figure 2: Armadillo [1] (173k Points) rendered using spheres. The sphere segments can be seen easily. The resulting surface is slightly bigger than the sampled data set.

Figure 3: Laser scan Pisa (1M Points) rendered using points as spheres. The huge number of points makes the result look quite well, but the flat areas appear noisy because of the rippled surface.

Discs

Another approach is to use a disc centered at the point for rendering. This allows representation of flat surfaces, without introducing the distortion caused by spheres. Also the intersection is now closer to the actual sampled surface. These improvements come with the additional requirement of a surface normal. If this information is not provided by the dataset, it can be calculated using local patches as introduced in Section 5 (Normal Calculation). An example using discs can be seen in Figure 4. Noticeable on this image are the black borders of some discs. This is caused by shadowing of neighboring discs. To remove these artifacts we introduce a minimum distance for intersections when spawning new rays. This leads us to the rendering results of Figures 5 and 6.

Figure 4: Bunny [1] (36k Points) rendered using discs. Shadow effects between neighboring discs can be seen.

Figure 5: Bunny [1] (36k Points) rendered using discs. Using a minimum intersection distance to avoid shadow artifacts. The boundaries of the discs can be seen slightly.

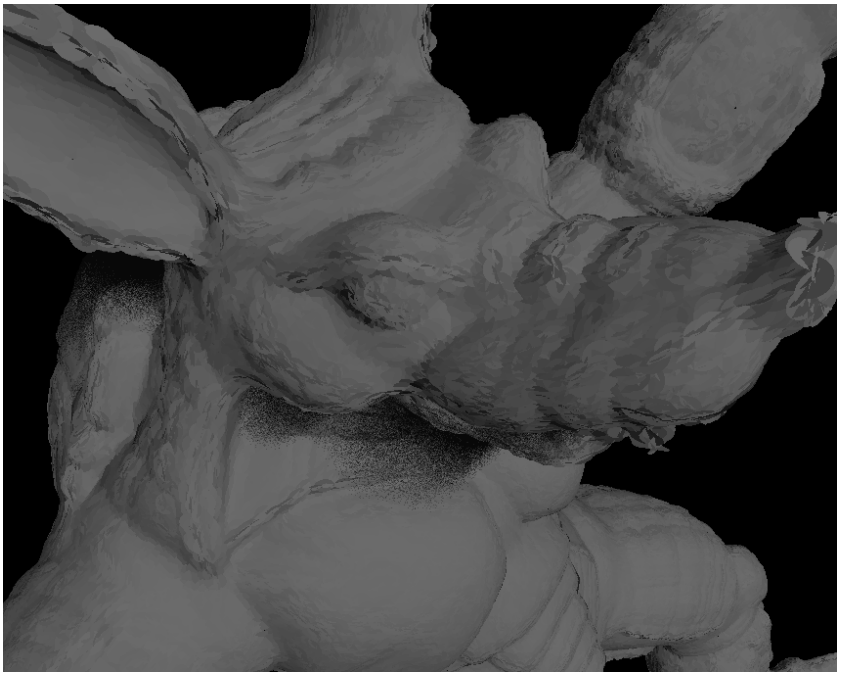


Figure 6: Armadillo [1] (173k Points) rendered using discs. Using a minimum intersection distance to avoid shadow artifacts. The segments of the disks can be seen and create a cornered image.

Splatting

The disc approach renders the whole disc using the same normal vector. This results in a nearly constant intensity for the whole disc and therefore in visible edges on the boundaries. To create a smooth intensity gradient we combine the information of several points in the neighborhood to calculate a surface normal. Therefore all discs in a range of one radius, starting with the nearest intersection, are collected. The current implementation is limited up to 16 discs.

One way to calculate the normal vector is using a weighted average. Therefore, the weighting function is defined as $w = 1 - \frac{d}{r}$ with d as the intersection distance in the plane of the disc, and r as the radius of the disc. This algorithm only works if the normals are aligned consistently, to point into or out of the object. This is usually not the case if the normals are calculated from the dataset, because a normal is only defined in its direction not in the magnitude, so n and $-n$ are defining the same surface orientation. When normals are not aligned adding them can deliver arbitrarily wrong normals. The result of using non aligned normals can be seen in Figures 7 and 8. Some tools try to solve this problem by aligning the normal vectors to the outside of the object after estimating them. But this task is a global problem and requires to consider the whole dataset at once. On some objects like the Möbius strip it is even impossible to calculate a consistent direction for normal orientations.

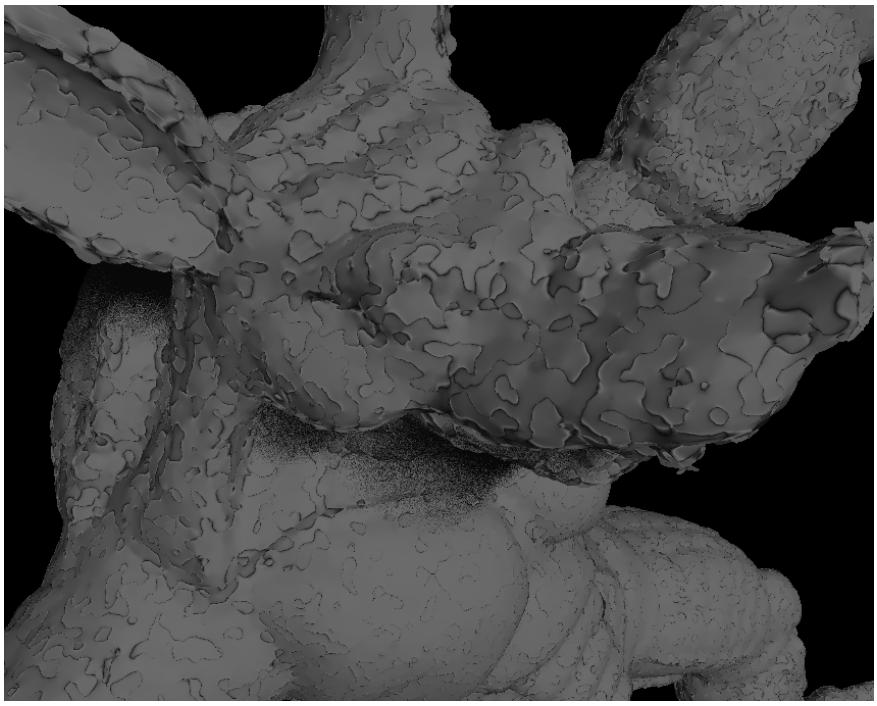


Figure 7: Armadillo [1] (173k Points) rendered using a weighted average of not aligned normal vectors. The dark edges which can be seen are a result of the sum of normal vectors which point in opposite directions.

Figure 8: Bunny [1] (36k Points) rendered using a weighted average of not aligned normal vectors.

To remove the requirement of prior aligned normals we use an approach presented by G. Schaufler and W. Jensen [7]. The normal vector is aligned at runtime towards the ray origin, to fulfill the condition $ray.direction \cdot normal < 0$. This results in a correct alignment in most situations as can be seen in Figures 9 and 10. There are also some situations on sharp edges, and if the ray and the normal are nearly orthogonal, which can lead to a misalignment. Figure 11 demonstrates the problem, when a ray intersects with a sharp edge and the alignment results in a wrong normal.

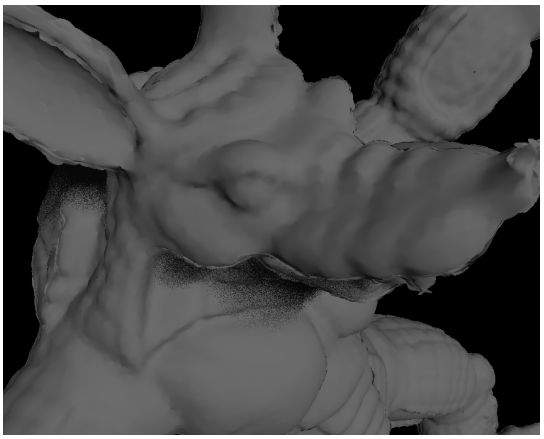
An alternative of using normals provided by the data points would be to use the points themselves to estimate the normal. Therefore we calculate the mean square error approximation of the surface based on the collected points. This method will be introduced in Section 5 (Normal Calculation). This method has the theoretical advantage that no normal information on the points is required. But the current implementations uses discs for the first intersection test, therefore the normal is required during the intersection calculation. An alternative would be using spheres for intersection which doesn't require normal information. But the intersection with a sphere will occur about $1r$ before the actual surface and therefore the intersection calculation would select wrong points if the ray is nearly parallel to the surface.

All mentioned splatting algorithms have problems on the object boundaries. This occurs when the discs which are used for intersection are to big for the local bending. The result is an intersection which does not represent the surface. Such situations cannot be detected before the scene traversal is finished. The problem is that the implementation is not capable of performing a second scene traversal. Without this additional intersection

Figure 9: Armadillo (173k Points) rendered using a weighted average of aligned normal vectors. The result looks very good except some artifacts on sharp boundaries.

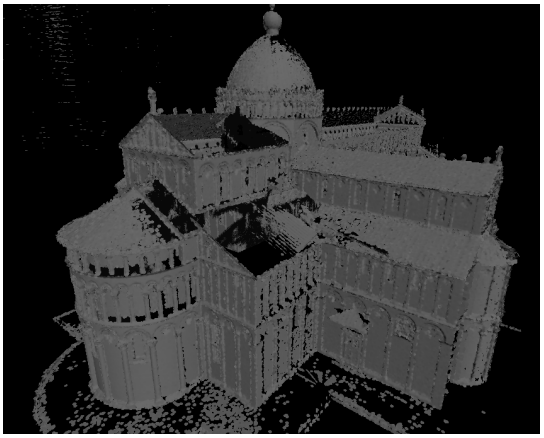
test, there is no way to decide if a better intersection would be found after discarding the first one. To resolve this problem we have to provide features to perform an additional intersection test. Using the current implementation this could result in a significant increase in render time. Another solution is presented by Sriram Kashyap [4] which extends the acceleration structure with additional information to avoid such artifacts.

(a) Standford Bunny [1]



(b) Standford Armadillo [1] detail view

(c) Laser scan Pisa detail view



(d) Laser scan Pisa

Figure 10: These images show the rendering results on several datasets using multiple intersections and aligned normals. The weighted averaging of the normal vectors results in smooth results on slightly curved surfaces. On sharp edges some border effects are the result e.g. the edge on the ear of the bunny.

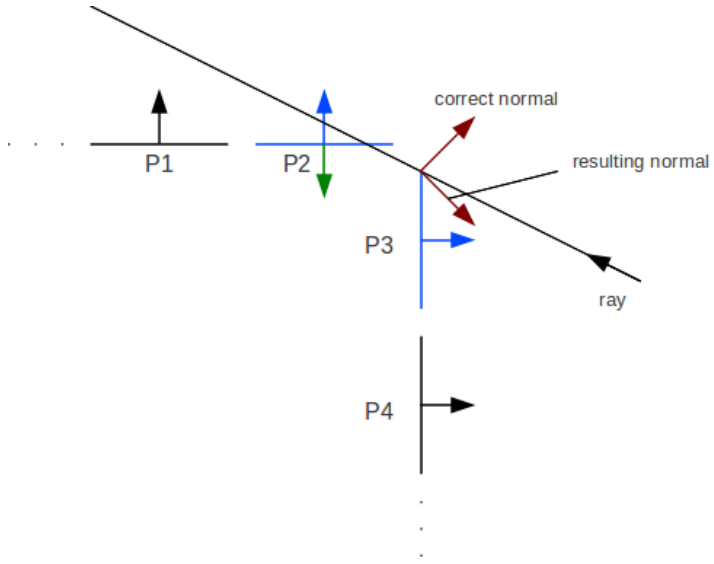


Figure 11: This figure illustrates problems on a sharp edge when aligning the normal towards the ray origin. The points P2 and P3 are intersected (using disc intersection), the alignment algorithm decides to flip the normal vector of P2, which results in a wrong normal.

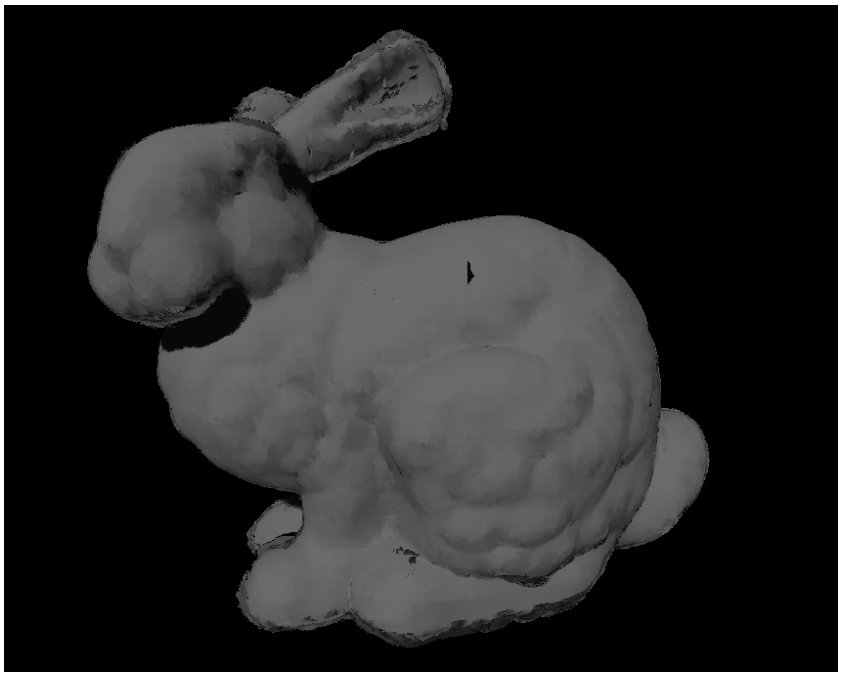


Figure 12: Rendering results for the bunny (36k points) using runtime normal estimation. The disc size is doubled in comparison to the approaches using normals. This is required to generate enough intersections for the normal estimation. The black artifact on the ridge of the bunny is supposedly a result of too few intersections to calculate a surface normal.

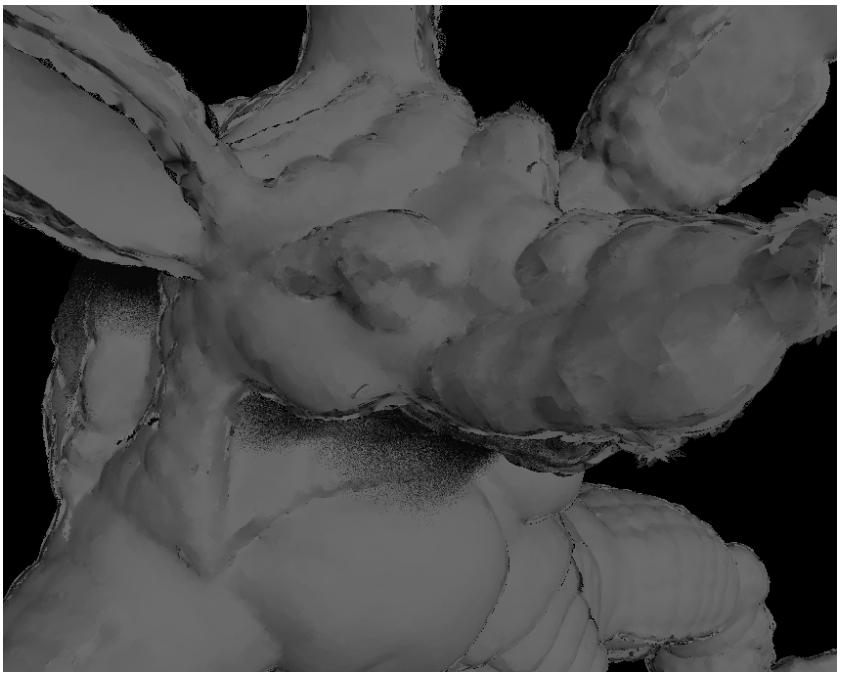


Figure 13: Rendering results for the Armadillo (173k points) using runtime normal estimation. The disc size is doubled in comparison to the approaches using normals. This is required generate enough intersections for the normal estimation. The increased disc size results in less quality on fine structure.

Performance

The ray tracer is currently capable of handling about 1.5 million points with interactive frame rates of more than 5 frames per second in any test. The current limitation concerning the number of points is the 32-bit host program which cannot allocate enough memory to build up the bounding volume hierarchy. The GPU memory requirements are 32 Byte for each point and 64 Byte per bounding bounding volume. Using up to 4 points (typically 3) per bounding volume and a bounding volume hierarchy with 2 children per node, this results in a combined memory requirement of about 75 Byte per Point. The test environment uses a single core of a NVIDIA GeForce GTX 590. The theoretical limit for this card (1,536 MB Memory) is about 21 Million points.

Dataset NrPoints	Bunny 36k		Cube 300k		Pisa 1078k		Armadillo 173k	
	trace ms	render ms	trace ms	render ms	trace ms	render ms	trace ms	render ms
Sphere	5.2	25.3	6.4	17.8	16.5	48.7	5.9	19.7
Disc	7.4	38	10.3	21.9	26.3	119	9.2	25.3
Multi mean	9.2	38	14.5	27.9	38.9	135	12.4	28
Multi aligned	9.3	38.5	14.6	26.4	39	135	12.3	28
Multi runtime	39	102	74	87	85	183	46.5	67.8

Table 1: Render times for the different datasets and render methods (Sphere, Disc, Multi intersection aligned, Multi intersection mean, Multi intersection with runtime normal estimation). For the Multi intersection with runtime normal estimation the disc radius is doubled to generate the required amount of intersections.

Nr Points ∗1000	trace time ms	render time ms
10	7.9	41
30	11.7	41.9
100	18.6	44.9
300	22.7	78.2
1000	44.3	140

Table 2: Trace and render times for different numbers of points, rendered on the Pisa data set

As can be seen in Table 1 the sphere intersection results in smallest render times. The disc intersection test is basically an extension of the sphere test and therefor results in increased render times. The additional overhead for multiple intersections is quite small, because this only requires additional time for storing and updating multiple results, but doesn't effect the actual intersection code. The increase in render time of the runtime alignment is almost unnoticeable, because this are only view instructions on

the intersected points. A major increase in render time can be noticed when using the runtime normal estimation. This is a result of the increased disc size, which is necessary to provide enough intersections for the normal estimation, and also of the eigenvector/eigenvalue decomposition which is currently not optimized for GPU usage.

The render times in Table 2 show a logarithmic increase of the trace time up to 300 thousand Points. The the results for 1 Million show a significant higher trace time. The reason for this is probably the reduced thread coherence which results in a inferior utilization of the GPU.

5 Preprocessing

Selection of Points

Some datasets contain several million points, this results in long computation time and huge memory requirements. To adjust the size of the model several algorithms are implemented. The simplest approach is to use the first part of the input data. Another method takes a subset where the points are selected according to $\lfloor i * s \rfloor : i \in \mathbb{N}, s \in \mathbb{R}, 1 \leq s$ with i as an index and s as step size. This provides a selection well suited for spacial sorted data sets e.g. scanned lines of a lacer scanner. A simple and fast algorithm is using a random selection. Finally a minimum distance filter is implemented. This is useful, if the distances of the points are very different, because it will reduce the number of points on positions with high density. The min distance algorithm is quite the same as the 3D sparse Matrix algorithm which will be mentioned below, it simply deletes all neighbor points instead of calculating a normal vector.

Neighbor Calculation

The neighbor algorithms are required to calculate the surface normals. A neighbor is defined as any point with a distance lower the $neighborDistance$. Two approaches have been implemented to calculate these points. Both are based on a grid with a regular size of one $neighborDistance$ to store the points. This method allows a quite fast calculation because all neighbors are in the 27 $(3 \cdot 3 \cdot 3)$ bounding blocks. The difference of the algorithms is now how to build and store this gird structure to perform a fast search of neighbor points.

Sparse Matrix

This algorithm is based on the idea of simply dividing the space in a 3D grid as mentioned before. The direct implementation would result in an infeasible memory requirement. To avoid unnecessary memory usage for empty boxes, only boxes containing points are stored. This is done by storing the points in a std::map (in other languages dictionary) which is basically a tree structure that supports fast data access. Based on this data structure the algorithm is quite simple: Load all points into the grid and then walk

through all blocks which contain points and check for neighbors in the surrounding blocks.

Scan Line

The scan line algorithm uses a 2D array with the size of the smallest side plane of the imaginary 3D grid. Now the points are sorted by the remaining direction and processed starting with the smallest value. Based on the position of the current point all points within a distance smaller than the $neighborDistance$ in scan line direction are stored in the 2D array. The update of the 2D array is handled incremental each time a point is selected. Now all neighbors must be within the block assigned to the current point and the 8 neighbor blocks. This algorithm works quite fast, but can require a significant amount of memory, if the side plane gets big. This can be caused by outliers for example.

Normal Calculation

The normal calculation is based on an approach presented by Marc Alexa et al. [3]. They calculate the normal vector based on the neighbor points. This is done by fitting a plane into the points. The algorithm minimizes the squared distances from the plane to the points. This is done by calculating the covariance matrix of the distance vectors to all neighbor points. Now the eigenvalues and eigenvectors of the covariance matrix are calculated. The eigenvector which corresponds to the smallest absolute value of the eigenvalue is chosen as the direction of the normal.

Point Size Selection

Another problem is to choose an appropriate value for the size of the points. The renderer would allow this to be selected for each point. The size of the points is stored within the normals. The length of the normal vector equals the radius of the disc or sphere. Currently there are two ways of selecting the size of the points. First entirely manual or based on an automatic calculated mean distance. Therefore a selectable number C of points will be chosen randomly from the dataset. Now for each of these points the nearest k points will be searched. For this search a simple brute force algorithm is used which calculates the distance to all other points and searches the nearest k points. As soon as all of this data is calculated, all $C \cdot k$ distance values are used together to calculate the median distance.

This algorithm has some interesting properties. It requires no information about the data set and doesn't need any precomputed or sorted data. Assuming an equally distributed set of points on a surface the average number of points n, which should be inside the neighborhood of a points, can be chosen easily by selecting $k = 2 \cdot n$.

6 Conclusion

Interesting results have been taken from the sphere approach which requires no normal information and renders very fast. Therefore only a selection of the point size is required, which works quite well using the heuristic presented at Section 5 (Point Size Selection). Few requirements and the fact that the data points can be seen make this method most suitable for data visualization.

The rendering method which generates the highest quality pictures is multi intersection using aligned normals. This results in smooth surfaces under most circumstances. The handling of sharp edges would be a topic for further work. Therefore a dynamic selection of the number of points depending on the curvature, or the implicit surface octrees presented in K. Sriram [4] could be a basis for further investigation.

The most interesting approach for further work is the runtime normal estimation. Therefore the requirement for normals during the intersection test should be removed. A solution could be an intersection test which checks the radial distribution of the points around the ray to prevent false intersections. The reduced memory requirement due to the removed normals can then be used to store more points which are required by this method to achieve the same quality.

All algorithms result in rendering times acceptable for interactive scene navigation and viewing.

References

[1] The stanford 3d scanning repository. http://graphics.stanford.edu/data/3Dscanrep/.

[2] A. Adamson and M. Alexa. Ray tracing point set surfaces. *Shape Modeling and Applications, International Conference on*, 0:272, 2003.

[3] M. Gross, M. K. H. Pfister (eds.), et al. *Point-Based Graphics*. Elsevier, 2007.

[4] S. Kashyap. Fast raytracing of point based models using gpus. Master's thesis, Indian Institute of Technology Bombay, 2010.

[5] NVIDIA Corporation. *CUDA Programming Guide*, 1 edition, 06 2007.

[6] D. Reiners, G. Voss, and J. Behr. Opensg: Basic concepts. *Proceedings of OpenSG Symposium 2002*, 2002.

[7] G. Schaufler and H. W. Jensen. Ray tracing point sampled geometry. *Rendering Techniques 2000*, 2000.

[8] T. Schiffer, A. Schiefer, R. Berndt, T. Ullrich, V. Settgast, and D. W. Fellner. Enlightened by the web – a service-oriented architecture for real-time photorealistic rendering. In *Tagungsband 05. Kongress Multimediatechnik Wismar*, pages 1–8, 2010.

[9] M. Segal, K. Akeley, et al. The opengl graphics system: A specification. Technical report, Khronos Group, Aug. 2008.

[10] I. Wald and H.-P. Seidel. Interactive ray tracing of point-based models. Technical report, MPI Informatik, Saarbrücken, Germany, 2005.

[11] T. Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23:343–349, June 1980.