GPU-based raytracing for accelerating deflectometric asphere measurements

Alexander Puschke[1],*, *Marc* Fischer[1], *Marcus* Petz[1], and *Rainer* Tutsch[1]

[1]Institut für Produktionsmesstechnik, Technische Universität Braunschweig, Schleinitzstraße 20, 38106 Braunschweig, Germany

Abstract. Deflectometry in transmission offers the ability to measure simultaneously the front and back surface geometries of transparent test specimen, while keeping their geometric relation. The acquired data consist of measured rays between a pattern generator and an image sensor. To obtain the 3-dimensinal geometry from the measurement data, a model-based evaluation method is used, which iteratively determines the lens parameters. For each iteration step, hundreds of thousands of light rays have to be traced through the model. This results in a calculation time in the dimension of one hour, which prevents the competitiveness of this measuring approach. In order to optimize the computation time parallelizing the raytracing calculations on graphics hardware seems promising. To achieve this task, NVIDIA OptiX, a high level raytracing API, is used to develop a ray tracer.

1 Introduction

Obtaining the 3-dimensional geometry of any object may represent an extensive task. However, adding transparency to the material properties complicates it, for most established measurement techniques.

To measure an aspherical transparent object like a lens, actual used technologies show some disadvantages. Scanning, contact-making techniques exhibit long measurement times and the possibility of damaging the specimen. Interferometric approaches offer a high resolution, but suffer from a small measuring range influencing their robustness.

The measured signal, of many optical techniques, is acquired separately for front and rear surface in reflection. This leads to multiple challenges, like the small signal from the remaining reflectivity and additional reflexes originating from the rear surface. Also the geometric relation is lost due to the separate measurement of front and rear surface. Therefore additional arrangements, to gain this information, are needed.

To avoid these challenges, we proposed a measurement of the incident and refracted light rays similar to a deflectometric measurement system. However, the two refracting surfaces prevent an absolute clarity of which path a given ray takes. This problem can be solved by taking multiple measurements from different directions and supply a model-based reconstruction algorithm with this data. In the following steps the software iteratively reconstructs the geometry of the specimen, as figure 1 illustrates. Due to the sheer amount of needed calculations the measurement time is not yet competitive compared to established techniques [1].

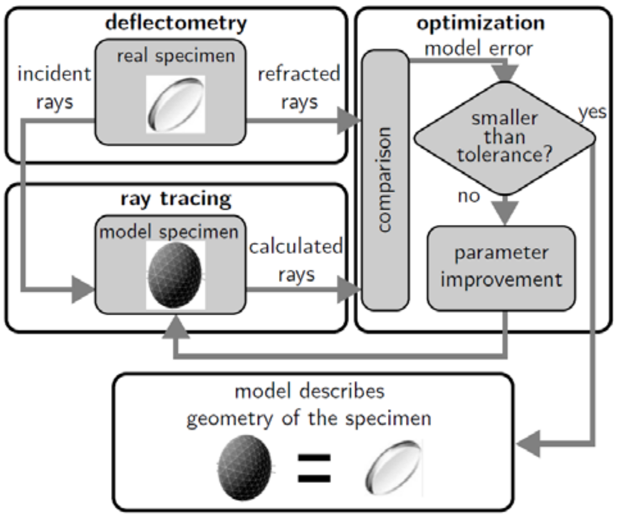

Figure 1. Model-based reconstruction based on [2]

In each iteration step, hundreds of thousands of light rays have to be traced through the model of the specimen multiple times. This leads to computation times in the order of an hour with an implementation on the CPU. Although the inherent parallel nature of the raytracing process suggests itself for an implementation on a modern GPU, a closer inspection reveals some challenges that need to be overcome before a considerable acceleration of the computation can be achieved. In this paper we discuss these aspects and evaluate the usage of GPU-raytracing in our model-based deflectometric surface reconstruction algorithm. The implementation is based on OptiX, which is a high-level general purpose raytracing API by NVIDIA.

* Corresponding author: a.puschke@tu-braunschweig.de

2 General purpose GPU computing

A graphics processing unit is normally integrated in a video card to render images for a display device. However, an application programming interfaces (API) like OpenCL or NVIDIA CUDA allows the conduction of general purpose computations on this hardware. Due to their architecture, GPU's excel at executing tasks in parallel while conducting the same instruction on multiple data. This means that a given task has to exhibit a general parallel characteristic to achieve fast computation times.

2.1 Raytracing on GPU's

Basic raytracing to calculate visible points on an object can be parallelised because the same intersection algorithm has to be applied to every ray. However, to simulate an effect like refraction of a light ray, another ray has to be spawned, which leads to recursion and cannot be parallelised completely.

To implement the raytracing algorithm on a GPU, NVIDIA OptiX, a dedicated high level raytracing-API, is used. The reason for using OptiX is that the programming style focusses on the algorithm [3]. This means an algorithm only has to be defined for a single ray and OptiX applies him to every casted ray. Furthermore the API handles the distribution of given tasks to the GPU for optimal performance.

2.2 Constructing a ray tracer

A basic ray tracer mainly consists of a ray generation program, intersection and material program. The ray generation program defines the number of primary rays, their origin and each direction of propagation. In our case we use backward raytracing, where the light rays spawn, contrary to the physical reality, in the observing camera. Depending on the used camera model the emerging rays differ in their direction. For simplicity a pinhole camera model is used.

To insert geometries into the scene two methods are used. Starting with simple geometric primitives which can be implemented by an object-specific intersection program. The intersection program tests if these rays hit any of the given geometries. In case of an existing hit point the coordinates are transferred to the corresponding material program, which defines the ray's outcome or further behaviour like colour, reflection and refraction.

Acceleration structures are used to lower the computation time by reducing the amount of necessary calculations to fulfil a given task. They are implemented mostly by bounding boxes, which surround geometric primitives to determine a possible hit on the underlying geometries as Figure 2 illustrates. If no hit point at the bounding box is found the intersection algorithm can be skipped.

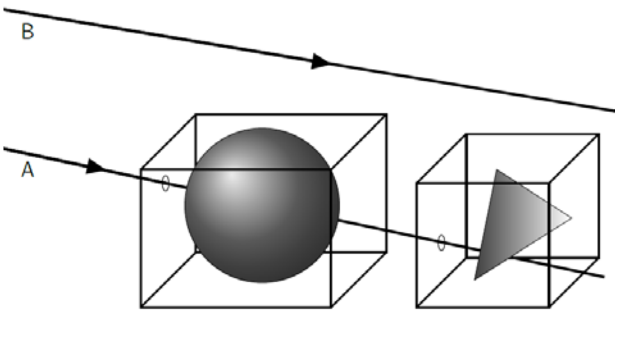

Figure 2. Surrounding bounding boxes to determine a possible hit on a given geometry

3 Ray tracer for deflectometry

The following steps are used to evaluate the usage of a GPU-based ray tracer for the application in the model-based reconstruction.

3.1 Scene

The first step to create a ray tracer for the described purpose was to recreate a typical measurement setup as a traceable scene. This setup consists of a pinhole camera to generate 512*512 rays, and cast them into the scene to intersect with a model specimen in front of a pattern generator as seen in Figure 3.

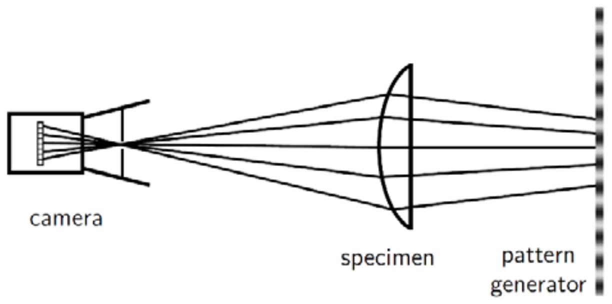

Figure 3. Basic principle of the test setup

Initially, the model specimen was created out of three geometric primitives (front surface, back surface, and a cylinder to connect those) by an intersection algorithm. Based on this setup the material implementation of the glass and pattern generating plane could be tested.

However, three basic geometric primitives are either insufficient or their definition tends to grow to a bad performing algorithm for representing an aspheric lens. For reasonable high-performance and variable shapes, the model specimen were constructed from polygon meshes with different amounts of primitives (44 to 658,734).

Each point of a polygon allows a local adjustment of the specimen. As a sophisticated intersection algorithm exists [4], the meshes consist of triangles.

3.2 Features

Furthermore, some useful transformations were implemented like the rotation and translation of the specimen as well as rotation, translation and tilting of both lens surfaces in reference to each other.

An example mesh with a low amount of triangles put into the scene is shown in Figure 4. Caused by the low amount of triangles the model specimen does not provide a decent image quality.

Improvement can be done by optimising the ray's calculated direction of refraction. This can be achieved by interpolating a local surface normal similar to the Phong-shading [5]. The local normal is calculated from the relative position inside the triangle in reference to the three point normals. But this does not change the physical dimensions of the specimen, which can be seen at the edges of the lens in Figure 5.

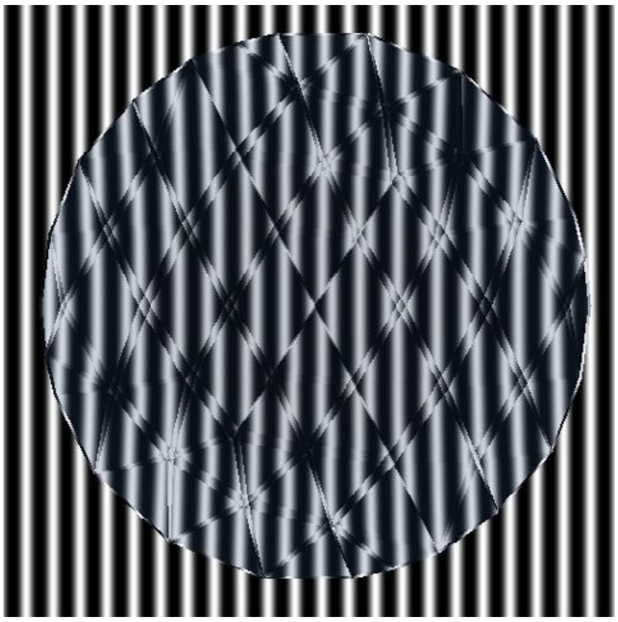
Figure 4 Live traced model-specimen consisting of a 176-piece triangle mesh (low amount of triangles for illustration)

A first estimation, for the needed amount of triangles per model specimen, can be obtained by comparing the incident rays with the number of triangles per lens surface. The currently used resolution results in 262,144 rays casted into the scene. Around 160,000 of these will intersect with the first lens surface which consists of up to 329367 triangles. This results in around two triangles per incoming ray and should be a decent basis for the following reconstruction.

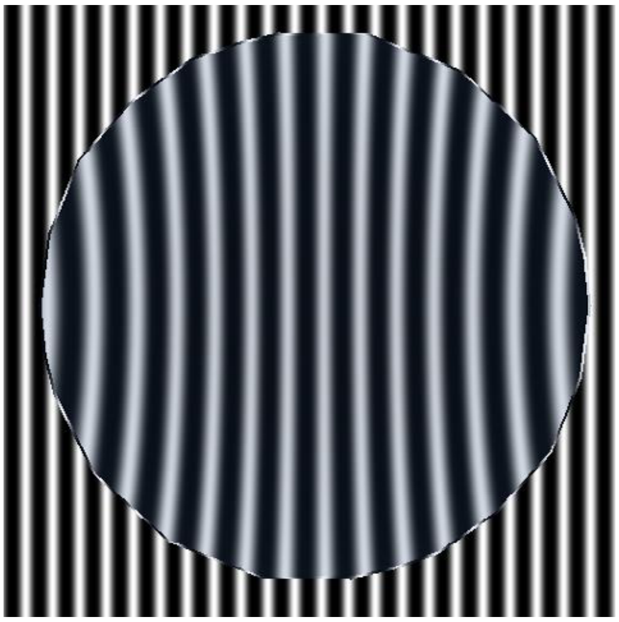
Figure 5. Same triangle mesh as Figure 4 with interpolated face normals.

3.3 Performance considerations

The hardware setup used for developing and testing this application consists of an INTEL Core i3-6100 and a NVIDIA GeForce GTX 650ti.

By analysing the different parts of the ray tracer, their relative impact on the calculation time varies depending on the given settings.

The time needed for loading the polygon mesh is linear to the amount of triangles. But the execution time per trace scales "sub-linear". For example increasing the amount of triangles from 176 to 658,734 results in a change of the computation time from 6 ms to 23 ms.

Computation time depending on the resolution is caused by two effects:

First of all by the amount of pixels because a ray needs to be traced for each pixel. But with a higher amount of pixel more calculations of the same type need to be done, which leads to a better parallelisation. In our case improvements at high resolutions are limited by the GPU-memory.

The second effect regards the initialisation time until the tracing starts. This is implemented to check for changed parameters and adds a constant delay which affects a fast repeating ray tracer more than a slow one, as Table 1 shows.

If the model specimen changes its shape, the acceleration structures need to be recompiled and will increase the computation time.

For the actual implementation the currently implemented visualisation of the traced image is no longer needed and only some specific data will be transferred to the measurement software.

Table 1. Ray tracer performance with a 658734-triangle mesh

Resolution	Rays per second	Time per trace	Time per pixel
128*128	4.39 million	3.73 ms	228 ns
256*256	7.01 million	9.35 ms	107 ns
512*512	11.6 million	22.6 ms	86 ns
1024*1024	15.4 million	68.1 ms	64 ns
2048*2048	19.5 million	215 ms	51 ns
4096*4096	23.8 million	705 ms	42 ns
6144*6144	20.6 million	1832 ms	49 ns

4 Conclusion

Using parallel computing on graphics hardware shows a high potential for accelerating the raytracing calculations in our model-based reconstruction of the test specimen. However the actual implementation into the measurement software still needs to be done to identify the achievable acceleration.

In addition to that, some improvements can be made regarding the construction of model specimen because the used ones were custom built for the purpose of testing. The next step is to automatize their construction based on input parameters like diameter, surface radius and thickness including a local mesh refinement.

Acknowledgment. The authors gratefully acknowledge the funding of this work by Deutsche Forschungs-gemeinschaft (DFG) under grant PE 1402/2-2

References

1. M. Fischer, M. Petz, R. Tutsch, *Model-Based Deflectometric Measurement of Transparent Objects*, Fringe 2013 – 7th International Workshop on Advanced Optical Imaging and Metrology, 573-576 (2013)
2. J.E. Nitsche, M. Fischer, M. Petz, R. Tutsch, *Measurement of the Geometry of Transmissive Optical Elements with Deflectometry*, International Symposium of Optomechatronics (2013)
3. S. D. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robinson, M. Stich, *Optix: A General Purpose Ray Tracing Engine*, ACM Trans. Graph. **4**, 1-13 (2010)
4. T. Möller, B. Trumbore, *Fast, Minimum Storage Ray-triangle Intersection*, Journal of Graphics Tools **2**, 21-28 (1997)
5. B. T. Phong, *Illumination for Computer Generated Pictures*, Communications of the ACM 18, **6**, 311-317 (1975)