TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN
*University of Kaiserslautern*

**Fachbereich Informatik**
*Department of Computer Science*

**Bachelor Thesis**

# FPM Post Processing:
## Real-Time Ray Tracing of Point Set Surfaces in OptiX

**Tim Biedert**

t_biedert09@cs.uni-kl.de

September 25th, 2012

Supervisors: Prof. Dr. Hans Hagen
Dr. Jörg Kuhnert

# Abstract

This thesis is an initial attempt to create an interactive visualization technique tailored to the Finite Pointset Method, a grid-free particle simulation method developed at Fraunhofer ITWM in Kaiserslautern. Typical applications are problems in the context of flow and continuum mechanics, such as sloshing fluids, refueling of tanks or the unfolding of airbags. In this thesis, a novel approach is presented by implementing a classic rendering technique on a state of the art programming architecture: Surfaces are approximated by moving least squares interpolation polynomials and rendered by means of ray tracing, as initially proposed by Adamson & Alexa. OptiX, a ray tracing acceleration engine designed by NVIDIA, is employed in order to harness the highly parallel computing capabilities of modern CUDA-enabled graphics devices.

First, a context is established by summarizing and discussing the suitability of several approaches in point based graphics, i.e. triangulation-, splatting-, and smooth interpolation-based techniques. The chosen iterative surface intersection algorithm is presented afterwards. Based on the construction of local polynomial approximations at several locations along each ray, convergence to the actual surface intersection is achieved. In each iteration, a local support plane is determined, serving as coordinate system for a moving least squares interpolation of the surrounding points, which are weighted using a Gaussian bell curve combined with a global smoothing parameter. The resulting quadratic or cubic polynomials are intersected by analytical or numerical means, respectively. Subsequently, a brief overview of the programmable ray tracing pipeline introduced by OptiX is provided, followed by technical details on the application structure, data management, and actual implementation. An extensive evaluation discussing various benefits and shortcomings of the presented approach concludes this thesis.

The novel combination of moving least squares point set surfaces and hardware-accelerated ray tracing offers great potential in the grid-free and highly interactive visualization of the Finite Pointset Method, where classic contemporary mesh-based rendering techniques fail by nature. Several demonstrations and benchmarks indicate a limited applicability of the original algorithm in unmodified form. While very smooth results can be achieved in surface regions defined by dense point sets featuring well-oriented normals, sparsely distributed point clusters with inconsistent normal orientations lead to conspicuous rendering artifacts. Loading procedures of several seconds render the presented approach useless for the interactive navigation through data sets consisting of hundreds to thousands of time steps.

# Zusammenfassung

Ziel dieser Arbeit ist die Erstellung einer interaktiven Visualisierungstechnik, welche auf die Finite Pointset Method, einer am Fraunhofer ITWM in Kaiserslautern entwickelten gitterfreien Methode zur Partikelsimulation, zugeschnitten ist. Typische Anwendungsgebiete sind strömungs- und kontinuumsmechanische Problemstellungen, wie z.B. das Betanken von Fahrzeugen oder die Entfaltung eines Airbags. Oberflächen werden durch Moving Least Squares Interpolation approximiert und mittels Ray Tracing dargestellt, wie ursprünglich von Adamson & Alexa vorgeschlagen. OptiX, eine Ray Tracing Beschleunigungs-Engine für moderne CUDA-fähige Grafikprozessoren von NVIDIA, wird eingesetzt, um eine Visualisierung in Echtzeit zu realisieren.

Zu Beginn werden verschiedene Ansätze auf dem Gebiet der punktebasierten Grafik, d.h. triangulierungs-, splatting- und interpolationsbasierte Techniken, zusammengefasst und deren Eignung diskutiert. Der gewählte iterative Algorithmus zur Schnittberechnung wird anschließend ausführlich vorgestellt. Konvergenz wird dabei durch die Konstruktion mehrerer lokaler polynomialer Approximationen der Oberfläche entlang eines jeden Strahls erreicht. In jeder Iteration wird eine lokale Hilfsebene bestimmt, die als Koordinatensystem für eine anschließende gewichtete Moving Least Squares Interpolation dient. Die Schnittberechnung mit den resultierenden quadratischen oder kubischen Polynomen wird analytisch bzw. numerisch durchgeführt. Nach einer kurzen Einführung in die frei programmierbare Ray Tracing Pipeline von OptiX, werden technische Details der Anwendungsstruktur, des Datenmanagements und der eigentlichen Implementierung erläutert. Eine ausführliche Evaluierung schließt die Arbeit durch eine Diskussion verschiedener Vor- und Nachteile des gewählten Ansatzes ab.

Im Gegensatz zu klassischen gegenwärtigen gitterbasierten Darstellungstechniken, beweist die präsentierte neuartige Implementierung des Moving Least Squares Approximationsverfahrens auf einer hochaktuellen Grafikkartenarchitektur großes Potential für die gitterfreie Visualisierung der Finite Pointset Method. Die gezeigten Darstellungen und Leistungsmessungen zeigen eine eingeschränkte Eignung des ursprünglichen Algorithmus in unmodifizierter Form. Während sehr glatte Ergebnisse in durch dichte Punktewolken mit wohldefinierten Normalen gegebenen Oberflächenregionen zu erreichen sind, führen isolierte Punkteanhäufungen mit inkonsistenten Normalenorientierungen zu auffallenden Darstellungsfehlern. Ladezeiten von mehreren Sekunden verhindern die interaktive Navigation durch Simulationen, welche aus hunderten bis tausenden von Zeitschritten bestehen.

# Contents

# Lists

## Figures



## Tables

# 1 Introduction

The visualization of numerical simulation results based on mere point clouds is a challenging task. Such data sets emerge from the *Finite Pointset Method* (FPM), a particle simulation method in the context of flow and continuum mechanical problems. Developed at the department of Transport Processes at Fraunhofer ITWM in Kaiserslautern, FPM is a grid-free method which is especially suited for time-dependent problems, such as fluid dynamical computations with free surfaces. Classical grid-based numerical procedures, e.g. Finite Elements or Finite Volumes, fail in these scenarios due to their inherent necessity for remeshing. Typical examples are the dynamic pressure of sloshing diesel in a truck tank or the unfolding of an airbag, as illustrated in figures 1.1 and 1.2.

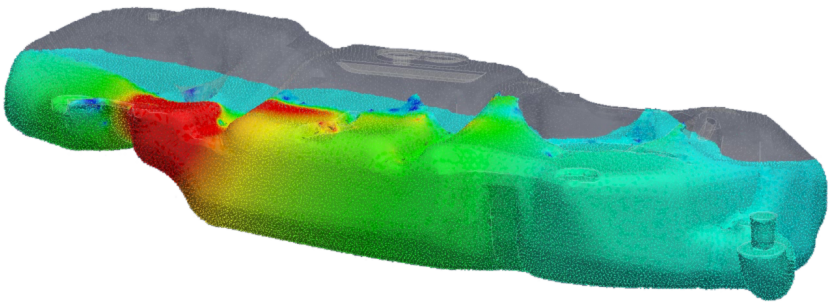

Figure 1.1: Pressure of sloshing diesel in a truck tank upon an impact situation. The HSV-based color scale indicates regions of normal (blue), increased (green) and high (red) pressure.

However, currently there exist no well-elaborated visualization approaches tailored to grid-free methods. A multitude of contemporary rendering techniques are grid-based, and thus inappropriate for the evaluation and analysis of FPM results. Furthermore, for high density point clouds with great geometric complexity relative to the rastered image, it seems natural to stay within the context of point-based shape representation and directly use the surface points as display primitives. At the same time, achieving interactive frame rates while rendering highly detailed point-based data sets is another desirable objective. This allows the simulation and visualization of technical processes in real-time via FPM, thus enabling an early intervention by the user at certain states or the detection of faulty input.

This bachelor thesis presents a simple and intuitive visualization of FPM simu-

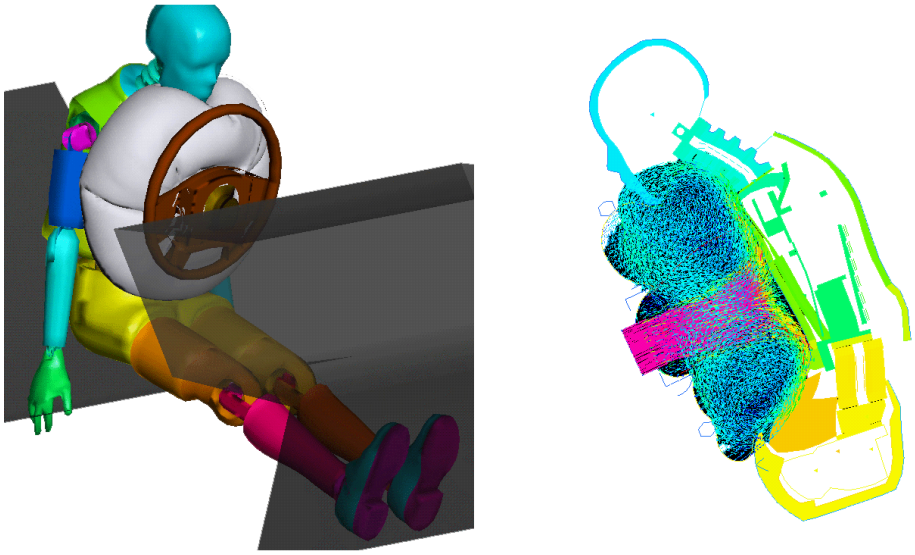

Figure 1.2: Impact of an unfolding airbag on the human body. Left: Triangulation-based rendering of airbag boundary and crash test dummy. Right: Cross section revealing the actual simulated air particles.

lations. The employed ray tracing algorithm restricts itself to the resulting point clouds and does not require any additional topological information, such as triangulations. Interactive frame rates are achieved by implementing ray-surface intersection computations in OptiX, a ray tracing acceleration engine developed by NVIDIA, harnessing the highly parallel computing capabilities of modern CUDA-enabled GPUs. This novel combination of a classic rendering technique implemented on a state of the art programming architecture successfully fulfills the aforementioned requirements of a grid-free and real-time visualization especially tailored to FPM data sets.

The structure of this thesis is as follows. After giving a brief overview of related work and the development of point-based graphics, chapter 2 presents the employed moving least squares surface definition and the corresponding ray-surface intersection algorithm. Chapter 3 introduces the OptiX ray tracing engine and contains all implementation specifics, such as data management and technical details on the iterative intersection algorithm. Visual quality and performance are evaluated in chapter 4. Finally, a summary and an outlook of future work are located in the concluding chapter 5.

# 2 Point Set Surfaces

## 2.1 Overview

This thesis presents the implementation of an iterative moving least squares surface intersection scheme in NVIDIA's ray tracing engine OptiX. In order to put the algorithm in context, a brief overview of various approaches in the history and development of point-based graphics is provided. The algorithm itself is outlined in section 2.2. More detailed surveys of point-based graphics and techniques can be found in Kobbelt & Botsch [2004] or Gross & Pfister [2007].

In contrast to the interactive real-time visualization of point set surfaces targeted in this work, the early days of point-based graphics focused on offline surface reconstruction, e.g. creating models from data acquired by laser scanners. Due to the success story of the hardware-accelerated rasterization pipeline, mesh-based reconstruction and rendering seemed natural. A popular idea was the surface definition based on the points' Delaunay triangulation (Boissonnat [1984], Edelsbrunner & Mücke [1994]). Several algorithms utilized this approach and only differed in their identification of surface triangles, e.g. the Crust (Amenta et al. [1998], Amenta et al. [2000]) or Cocone algorithms (Dey & Goswami [2003]). For interactive visualization these methods are not applicable as the construction of the Delaunay complex of millions of points is computationally expensive. A fast approach by Hoppe et al. [1992] simply defines an implicit signed distance function based on the normal component of the distance to the closest point, yielding a surface consisting of planes bounded by the points' Voronoi cells. The aforementioned algorithms share the common problem of creating only $C^0$ approximations. This can be visually improved by building a piecewise smooth subdivision-based surface over the triangulation (Hoppe et al. [1994]). Nevertheless, considering the flexible programmability of modern processors and graphics hardware, triangulation-based methods tailored to the legacy fixed-function graphics pipeline seem outdated. In addition, a visualization method especially designed for FPM data sets should go hand in hand with its grid-free distinctiveness.

Pure point-based rendering is a completely different field of work related to point set surfaces. Some key developments in this area are surfels (Pfister et al. [2000]), differential points (Kalaiah & Varshney [2001]) or splatting-based algorithms such as QSplat (Rusinkiewicz & Levoy [2000]) or EWA (Zwicker et al. [2002]). Sainz & Pajarola [2004] present a detailed survey of various point-based rendering techniques. A critical drawback of point-based rendering algorithms is their lack of

robust surface interrogation means, such as ray-surface intersections which are required in modeling and CSG. Furthermore, best visual results are achieved with extremely high point counts (e.g. hundreds of millions of points for QSplat), which is contradictory to typical FPM data set sizes, ranging from a few thousand surface particles for quick prototyping to some millions for precise simulation.

Rapid ray tracing-based intersection evaluations are achieved by Schaufler & Jensen [2000] without an intermediate surface definition. By collecting points within a cylinder around the ray, a weighted average surface can be determined. Although very fast, the resulting location and orientation of the surface geometry depends on the particular ray intersecting the surface. Consequently, irritating view-dependent artifacts in interactive real-time visualizations render this approach unusable. Nevertheless, as fluid dynamical computations are a central application of FPM simulation, appropriate and intuitive photorealistic effects such as shadows, reflections and refractions are desirable. Thus, ray tracing still seems very advantageous as the rendering method of choice.

Fitting smooth interpolants solves the aforementioned view-dependence and provides several benefits. A costly triangulation is no longer required as rendering and surface interrogation can be achieved by means of ray casting techniques, where intersections are either computed explicitly or approximated numerically. Furthermore, smooth interpolants directly provide differential geometric properties of the surface such as normals or curvature and facilitate the handling of noisy data. Radial basis functions (RBF) can be used to construct a global and smooth interpolant for scattered data (Savchenko et al. [1995], Turk & O'brien [2002]). However, the required computationally expensive solution of large linear systems gave rise to several optimization approaches, such as compactly supported functions of minimal degree (Wendland [1995]), the fast evaluation and solution using multipole expansions (Beatson & Newsam [1998]), or thinning and hierarchical clustering techniques (Floater & Iske [1998], Iske [1999], Dyn et al. [2002], Ohtake et al. [2003b]). Other methods involve the local fitting of globally smooth interpolants (Goshtasby & O'Neill [1993]), performing purely local fits (Pratt [1987], Lei et al. [1996]) and blending local surface approximations (Ohtake et al. [2003a]).

Moving least squares (MLS) (Levin [1998]) takes the idea of local fits to the extreme by constructing a smooth interpolant for every point on the surface and was first introduced to 3D computer graphics by Alexa et al. [2001]. Nowadays, MLS surfaces are increasingly adopted as the standard definition of point set surfaces, with a multitude of different applications in surface reconstruction, processing (deformation, animation, editing, simplification) and rendering. A detailed survey and classification of various MLS methods can be found in Cheng et al. [2008], also addressing advanced topics such as adaptive sampling, complex surfaces with sharp features and theoretical guarantees.

This bachelor thesis focuses on the rendering of moving least squares surfaces by means of ray tracing, as initially proposed by Adamson & Alexa [2003a] and Adamson & Alexa [2003b]. Successfully employed to visualize various scientific models, e.g. the animation of deforming point-sampled surfaces (Adams et al. [2005]), fracturing materials (Pauly et al. [2005]) or point-sampled thin shells (Wicke et al. [2005]), the MLS-based surface definition seems suitable for the visualization of FPM particle simulations. Furthermore, the attainability of interactive frame rates was successfully demonstrated using efficient ray-surface intersections combined with highly optimized acceleration structures (Wald & Seidel [2005]) or exploiting GPU hardware acceleration (Tejada et al. [2006]). However, the latter approach is inflexibly tailored to vertex and pixel shaders of the standard rasterization pipeline. The implementation in OptiX benefits from GPU parallelization while at the same time providing extensive freedom to the programmer.

## 2.2 Moving Least Squares Surface Definition

This section outlines the employed moving least squares surface definition and iterative ray-surface intersection algorithm, as initially proposed by Adamson & Alexa [2003a].

Let a set of points $p_i \in \mathbb{R}^3$, $i \in \{1, ..., N\}$, be sampled from a smooth surface $S$, possibly including noise. A typical scenario in this context is surface reconstruction based on reverse engineering, e.g. the optimization of worn-out parts. The general idea of the surface definition bases on moving least squares approximation by building local polynomial approximations at any point in space. A point $\boldsymbol{s}$ belongs to the surface if its local polynomial approximation contains $\boldsymbol{s}$. This scheme is applied iteratively to converge to the surface $S$ along a ray $\boldsymbol{r} = \boldsymbol{o} + t \cdot \boldsymbol{d}$.

The following three steps are iterated until the sequence $\boldsymbol{r}_0$, $\boldsymbol{r}_1$, $\boldsymbol{r}_2$, ... converges to the ray-surface intersection or the procedure is terminated (see Figure 2.1):

1. **Support plane:** The evaluation of a moving least squares approximation requires the construction of a local coordinate system, which is defined using a support plane $H$ in $\boldsymbol{r}_i$ with normal $\boldsymbol{n}$. As FPM provides normal information for all surface particles, a suitable normal for the support plane can be easily computed as the normalized weighted average of the neighboring points $\boldsymbol{p}_j$ of $\boldsymbol{r}_i$:

$$\mathbf{n} = \frac{\bar{\mathbf{n}}}{\|\bar{\mathbf{n}}\|} \quad \text{with} \quad \bar{\mathbf{n}} = \frac{\sum_{\mathbf{j=1}}^{\mathbf{N}} \theta\left(\|\mathbf{p_j} - \mathbf{r_i}\|\right) \mathbf{n_j}}{\sum_{\mathbf{j=1}}^{\mathbf{N}} \theta\left(\|\mathbf{p_j} - \mathbf{r_i}\|\right)} \tag{2.1}$$

   However, in order to avoid mathematic cancellation effects, this approach requires consistently oriented normals with little deviation. Alternatively, as

originally presented by Adamson & Alexa [2003a], the normal of the support plane can be determined by minimizing the weighted distances of the points $\boldsymbol{p}_j$ to $H$, which is formulated in the following weighted least squares minimization problem:

$$\min \sum_{j=1}^{N} \left(\langle \mathbf{n}, \mathbf{p_j} - \mathbf{r_i} \rangle\right)^2 \theta\left(\|\boldsymbol{p}_j - \boldsymbol{r}_i\|\right) \tag{2.2}$$

Equation (2.2) can be rewritten in bilinear form:

$$\min_{\|\boldsymbol{n}\|=1} \boldsymbol{n}^T B \boldsymbol{n} \tag{2.3}$$

where $B = \{b_{kl}\}$ is the matrix of weighted co-variances

$$b_{kl} = \sum_{j=1}^{N} \theta\left(\|\boldsymbol{p}_j - \boldsymbol{r}_i\|\right) \left(p_{j_k} - r_{ik}\right) \left(p_{j_l} - r_{il}\right) \tag{2.4}$$

The minimization problem (2.3) is solved by computing the eigenvector of $B$ corresponding to the smallest eigenvalue. The resulting H is approximately parallel to the surface $S$ in its nearest approach to $\boldsymbol{r}_i$.

2. **Polynomial approximation:** Using the support plane $H$ in $\boldsymbol{r}_i$ as reference, a local bivariate polynomial approximation $a_i$ of $S$ is computed. The coefficients of $a_i$ are determined by solving the following weighted least squares problem:

$$\min \sum_{j=1}^{N} \left(a_i\left(x_j, y_j\right) - f_j\right)^2 \theta\left(\|\boldsymbol{p}_j - \boldsymbol{r}_i\|\right) \tag{2.5}$$

where $(x_j, y_j)$ is the projection of $\boldsymbol{p}_j$ onto $H$ in normal direction and $f_j = \langle \mathbf{n}, \mathbf{p_j} - \mathbf{r_i} \rangle$ is the height of $\boldsymbol{p}_j$ over $H$. This is solved using standard numerical methods, e.g. applying Gaussian elimination or the conjugate gradient method to the linear system of normal equations. Numerical stability can be improved by QR decomposition using Householder reflections. The resulting polynomial is expected to be a good local approximation of the surface $S$ around the reference point $\boldsymbol{r}_i$, if $\boldsymbol{r}_i$ is sufficiently close to $S$. Notably, this approach is not restricted to bivariate polynomials. The same general scheme can be applied to polynomial approximations of higher dimension.

3. **Intersection:** If the ray intersects $a_i$, the resulting intersection point $\boldsymbol{r}_{i+1}$ serves as the starting point for the next iteration and can be used to converge to $S$. In order to restrict the algorithm to local intersections close to $\boldsymbol{r}_i$, a region of trust $T$ is defined. Only intersections within $T$ are considered. For quadratic approximation $a_i$ the ray-polynomial intersection can be computed directly, whereas for cubic polynomials numerical schemes such as bisection or Newton's method can be applied.

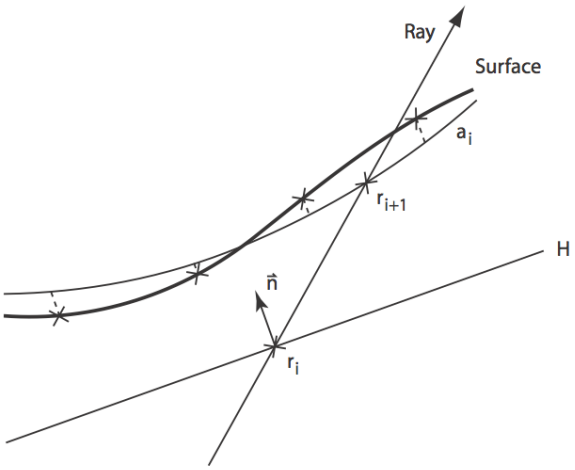


Figure 2.1: Iterative ray-surface intersection algorithm. At each point $\boldsymbol{r}_i$, a bivariate surface approximation polynomial $a_i$ is determined by local moving least squares interpolation. The intersection point $\boldsymbol{r}_{i+1}$ serves as the starting point for the next iteration and can be used to converge to the surface. (Source: Adamson & Alexa [2003a])

The above weighting function $\theta : \mathbb{R} \to \mathbb{R}$ is a smooth, positive, monotone decreasing function, specifying the influence of a point based on its euclidean distance to the respective reference point. In practice, Gaussian weights are used, i.e. $\theta(d) = \exp(-d^2/h^2)$, as illustrated in Figure 2.2. $h$ is a global parameter, specifying the locality of the approximation, and can be used to smooth out small variations in the surface (e.g. noise).

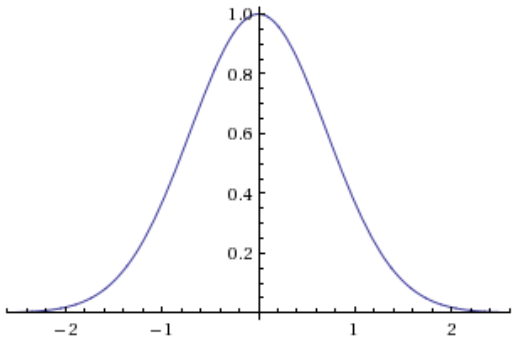


Figure 2.2: Gaussian weight function $\exp(-d^2/h^2)$ with $h = 1$.

In order to converge to the surface $S$, the illustrated iterative intersection procedure requires a suitable starting point $\boldsymbol{r}_0$. This is solved by constructing a set of balls $B_i$ of radius $\rho_i$ around the points $\boldsymbol{p}_i$. If the surface $S$ is contained in the union

of balls, i.e. $S \subseteq \sum_{i=1}^{N} B_i$, the balls form a bounding volume of $S$ which is easy to test for intersection and constrains the set of suitable starting points. An intersected ball indicates a potential ray-surface intersection, since a ray intersecting $S$ must also intersects at least one ball containing the intersection. As $S$ is unknown, the radii $\rho_i$ need to be estimated. Conservative values such that $B_i$ encloses the $k$-nearest neighbors of $\boldsymbol{p}_i$ yield satisfactory results, e.g. $k = 6$.

The intersection of a ray with a set of balls can be computed efficiently. Furthermore, the set of potentially intersected balls is greatly reduced by using an octree as acceleration structure, as illustrated in figure 2.3. Since the first ray-surface intersection is of interest, the intersected balls are sorted along the ray. However, this is not necessarily required, depending on the rendering algorithm, e.g. for secondary shadow rays. Each ray-ball intersection is handled as follows: The center $\boldsymbol{p}_i$ of $B_i$ is used as initial reference point for the construction of local coordinate system and polynomial approximation $a_i$ of $S$. $\boldsymbol{p}_i$ is expected to be close to the surface $S$, thus providing good a approximation around $\boldsymbol{p}_i$. Intersecting the ray with $a_i$ yields the starting point $\boldsymbol{r}_0$ for the above iterative procedure, where $B_i$ is used as the region of trust $T$. If no ray-surface intersection is found within $B_i$, the next intersected ball along the ray is inspected. Interestingly, this approach provides a certain speedup as the initial approximations are independent of the ray and can be precomputed for each ball.

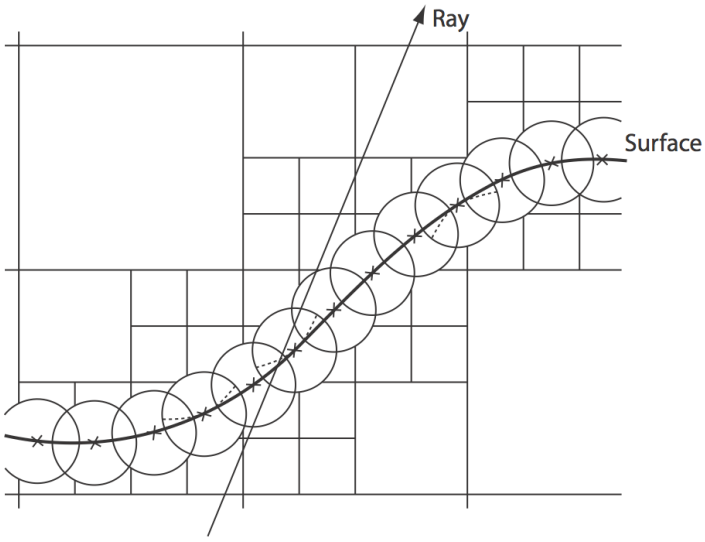


Figure 2.3: Region of trust balls encompassing the surface. Each ball contains an initial precomputed local surface approximation using its center as reference point. An octree is used for intersection test acceleration. (Source: Adamson & Alexa [2003a])

# 3 Implementation in OptiX

## 3.1 Real-Time Ray Tracing on GPUs

First CPU-based approaches to real-time ray tracing were implemented in the 1990's on highly parallel supercomputers, e.g. Green & Paddon [1990], Muuss [1995] or Parker et al. [1999]. Due to faster hardware and algorithmic improvements, real-time ray tracing is finally feasible on contemporary consumer computers. While there exists a multitude of different approaches to interactive ray tracing of animated scenes, these techniques usually depend heavily on scene complexity, model and data structure, type of motion or ray coherency. Thus, most algorithms are designed with a focus on specific applications or classes of rendering algorithms. A detailed discussion and evaluation of various state of the art methods in real-time ray tracing is given in Wald et al. [2007]. Furthermore, efficient implementations of several ray tracing algorithms were successfully demonstrated on GPUs, such as Horn et al. [2007], Popov et al. [2007] or Aila & Laine [2009]. Again, these are mainly focused on application-specific performance, rather than flexibility.

A different approach is taken by OptiX, a general purpose programmable ray tracing engine designed for CUDA-enabled GPUs by NVIDIA. The framework was first presented at SIGGRAPH 2010 (Parker et al. [2010]) and will be described in this section. For detailed information and API documentation see the official developer website[1].

The core idea bases on the observation that most ray tracing algorithms share a common structure and can be refined with a small set of user-programmable operations for ray generation, scene traversal, object intersection and material shading. This flexible architecture allows for a wide range of applications. e.g. interactive rendering, collision detection or scientific simulations such as sound propagation. In order to allow this high degree of freedom, OptiX was designed with the following trade-offs and design decisions in mind. As a general low level engine, it focuses on fundamental ray tracing operations instead of rendering specifics such as lighting models or material definitions and creates an abstract execution model as a sequence of user-specified programs. Complexity is reduced by hiding ray management and acceleration structures from the user. Nevertheless, an efficient scene representation based on a flexible node graph system supporting level-of-detail, instancing and nested acceleration structures provides a multitude of optimization possibilities for

[1] http://developer.nvidia.com/optix

various scenarios.

Interestingly, the programmable ray tracing pipeline can be seen as a direct analog to the programmable rasterization pipelines found in OpenGL/Direct3D: an abstract rasterizer combined with lightweight user-supplied vertex, geometry, tessellation and fragment shaders allows the implementation of various rasterization-based techniques.

Figure 3.1 illustrates the control flow of the configurable ray tracing pipeline. The core function *rtTrace* alternates between locating an intersection and applying the corresponding shading operations. Data can be read and written to either user-defined ray payloads or global device memory buffers, allowing arbitrary computations during scene traversal.

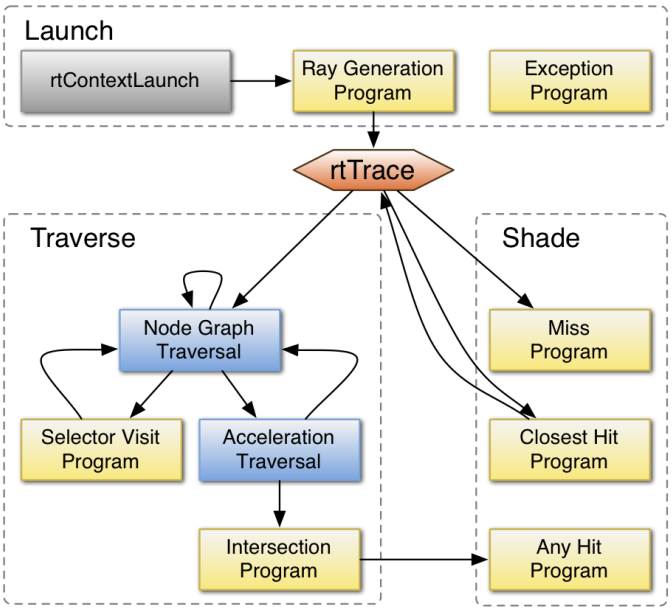


Figure 3.1: Control flow of the OptiX ray tracing pipeline. User-supplied programs are represented by yellow boxes. The core function *rtTrace* alternates between scene traversal and applying shading operations. (Source: Parker et al. [2010])

The following list outlines the programmable elements of OptiX's ray tracing pipeline.

**Ray Generation** programs define the entry point of the pipeline. A single call to *rtContextLaunch* creates many invocations of this type, e.g. for each pixel in a pinhole camera model.

**Intersection** programs perform ray-geometry intersection tests during acceleration

structure traversal. The intersection location and relevant properties such as surface normals, texture coordinates or other user defined attributes solely based on hit position are computed.

**Bounding Box** programs compute the axis-aligned bounding box associated with each primitive, allowing the creation of acceleration structures over arbitrary geometry. Typically, tighter bounds result in improved performance.

**Closest Hit** programs are invoked once scene traversal has determined the intersection of a ray with the geometry closest to the ray origin. Typical applications are surface shading and casting secondary rays for effects such as shadows, reflection or refraction. The resulting data, e.g. pixel color value, is stored in the ray payload.

**Any Hit** programs are invoked for each ray-object intersection detected during scene traversal. They complement the intersection programs by allowing material properties to participate in the intersection decision, e.g. alpha transparency based on texture lookup or early ray termination for shadow rays.

**Miss** programs are executed for rays without any geometry intersection. This is useful for environment map lookups or simply setting a background color.

**Exception** programs allow the handling of errors and exceptional conditions, such as recursion stack overflow or out of range buffer indices. Typically, this involves printing debugging messages or highlighting the problem by setting the corresponding output buffer's pixel value to a special striking color value.

**Selector Visit** programs expose programmability for coarse-level node graph traversal, e.g. the implementation of level-of-detail techniques.

The above user-specified programs are provided to OptiX in the form of Parallel Thread Execution (PTX) functions, a virtual assembly language being part of the CUDA architecture. Conveniently, these programs can be implemented in CUDA C/C++ and compiled to PTX by the *nvcc* compiler. Data-dependent optimizations and workarounds for recursion and function pointer operations on legacy hardware are performed by a PTX Just-In-Time (JIT) compiler, which is central to the OptiX host runtime.

Scene representation and acceleration structures represent another essential component of the OptiX ray tracing system. A flexible container object called the *context* stores scene geometry and associated programs. The geometry is organized in a hierarchical directed graph which can be augmented by special affine transformation nodes, enabling the creation of dynamic scenes. Geometry programs (*Intersection*, *Bounding Box*) and material programs (*Closest Hit*, *Any Hit*) are assigned to certain geometry and material classes, respectively. These can be instanced multiple times in the scene graph, thus allowing the sharing of common data. OptiX offers

several types of acceleration structures to be integrated in the scene graph which are used by the traversal algorithm to quickly determine candidates of ray-geometry intersection. The SBVH algorithm (Stich et al. [2009]) focuses on hierarchy quality, whereas HLBVH2 (Garanzha et al. [2011]) provides rapid construction speed. Multiple balance levels in between ensure flexibility. In addition, the possibility of applying different acceleration structures to different sub graphs allows a more efficient traversal of mixed scenes featuring both static and dynamic groups of geometry. During context launch, acceleration structures marked for rebuild are constructed in two stages by first invoking the respective bounding box programs and performing the actual build afterwards. Available host and device parallelism is exploited to increase construction speed.

Steady improvements since its first presentation at SIGGRAPH 2010 indicate NVIDIA's ongoing interest in establishing OptiX as a state of the art key technology in real-time ray tracing. Consequently, the evaluation of its performance regarding ray tracing-based point set rendering techniques is of great interest. Section 4.1 compares the achieved performance to the original results presented by Adamson & Alexa [2003a].

## 3.2 Framework and Data Management

By outlining the application structure and initialization procedures, this section presents the framework required to perform interactive real-time ray tracing of moving least squares point set surfaces on the GPU.

The developed application was written in C++ and utilizes several libraries besides OptiX. Window and input management is provided by *freeglut*[2], a platform independent interface for OpenGL applications. The *OpenGL Extension Wrangler (GLEW)*[3] allows comfortable access to advanced OpenGL extension such as vertical synchronization. *Armadillo*[4] is used to solve the initial per-sphere weighted least squares approximations of the surface. Making heavy use of recursive templates and template meta-programming, the linear algebra library performs compile-time combination of operations and reduction of temporaries. Screenshots are saved in PNG format using *FreeImage*[5].

After the main application window is created and callbacks for drawing and input handling are registered, buffers and variables used by OptiX are initialized. An OpenGL buffer object of type GL_ARRAY_BUFFER serves as output buffer to OptiX and stores the final ray traced image. Each frame, the contents of this buffer

[2] http://freeglut.sourceforge.net
[3] http://glew.sourceforge.net
[4] http://arma.sourceforge.net
[5] http://freeimage.sourceforge.net

are copied to an OpenGL texture which is rendered fullscreen using very basic orthographic projection. Typically, the texture resolution equals window size in order to avoid interpolation. Simple grid-based supersampling can be implemented with little effort, though, by increasing the output buffer resolution and activating linear texture filtering in OpenGL. If buffer objects are not available, the ray traced image can be drawn by mapping the OptiX output buffer to host memory and copying data to the framebuffer using OpenGL functions. However, this fallback solution suffers from a grave performance penalty due to the CPU-based data transfer, which should be avoided if possible.

Besides the aforementioned image output memory, a multitude of other global variables and buffers required for the actual intersection algorithm need to be allocated. Most variables are crucial to interactivity and can change each frame. Examples are camera position, lighting conditions and algorithm parameters such as the smoothing value $h$ or iteration and recursion limits. On the contrary, buffers are used to store the actual geometry data which is constant during the currently loaded FPM time step. While logically a multitude of different attributes is associated with each point in the data set, i.e. an *array of structs*, the equivalent *struct of arrays* approach is preferred in GPU programming in order to optimize global memory access patterns. Thus, the following buffers are allocated, each holding the respective values for all points in the data set:

- Point coordinates and radii of trust
- Normals
- List of neighbor indices
- Polynomial coefficients of the initial surface approximations
- Initial local coordinate frames

Memory alignment is optimized for each point by the combination of its coordinates and region of trust radius. Padding bytes are introduced in the remaining buffers to achieve the same effect. The number of neighbor indices is constant for each point in order to avoid control flow divergence in the OptiX code.

Once variables and buffers are set up, the various required OptiX programs to complete the abstract ray tracing pipeline are created and assigned. Section 3.3 provides extensive implementation details of the central programs. The successful initialization of OptiX and allocation of GPU memory is followed by the actual reading and preprocessing of an FPM data set in *EnSight 6* file format. While general information has been determined at application startup, e.g. the number of available time steps, attributes and respective file paths, the first time step is now loaded.

By computing the global axis-aligned bounding box of the surface point set, a reasonable scaling factor can be derived, improving overall numerical properties in conjunction with the GPU's single precision arithmetics. Also, an initial useful camera position and orientation is determined. The scaled surface points are filled in an octree data structure, accelerating the computation of each point's nearest neighbors relevant for interpolation and the region of trust radii. At this stage neighbors can be filtered, e.g. by skipping points with opposing normals which would otherwise cause artifacts on flat volumes. The fixed number of stored nearest neighbors is obviously related to the smoothing radius $h$ and must be defined accordingly. Afterwards, an initial moving least squares surface approximation is computed for each point in its sphere of trust, as described in section 2.2. All resulting data is transferred to the aforementioned respective global buffers in GPU memory. Occupied memory will be deallocated in subsequent time step loading procedures or upon application termination. Finally, the OptiX scene graph is created. As recommended in the OptiX developer guide, a very flat hierarchy is employed by assigning all surface points directly to the root node. Interestingly, the attributes of each point could directly be stored in variables of the geometry instances in the scene graph. However, this would effectively result in an *array of structs* implementation. Thus, only a single integer is assigned to each point instance, indexing its attribute values in the above global buffers. Though requiring a longer construction time, the SBVH acceleration structure (Stich et al. [2009]) yields best performance for this static scene. The loading procedure is finalized by validating and compiling the OptiX context.

Eventually, the application's main loop is started. Each frame, keyboard and mouse input is handled, providing interactivity. Thus, mutable OptiX variables such as camera position or changes in the aspect ratio due to window resize events are updated on the GPU. The actual rendering of the scene is invoked by launching the OptiX context, as outlined in figure 3.1. The loop iteration is ended by minor instructions such as drawing text overlays or measuring frames per second. Optionally, the loading of a whole new FPM time step can be initiated, resulting in the application of the routine illustrated above.

## 3.3 Ray-Surface Intersection Programs

Implementing the MLS point set surfaces approach in OptiX requires the breakdown and adaption of the algorithm to the various programs in the ray tracing pipeline. The content of each program is presented separately in the following subsections, sticking to the same partitioning as in section 3.1. All programs are executed solely on the GPU.

### 3.3.1 Ray Generation

Each frame, the launch of the OptiX context creates many invocations of the *Ray Generation* program, i.e. for each pixel in the current image output buffer. By

using the globally stored camera position and orientation, the launch index of each invocation is mapped to a ray in world space. Additionally, a ray payload consisting of the final output color value and the current recursion depth is allocated on stack. Both ray and its payload are input to *rtTrace*, the core procedure initiating scene graph traversal in OptiX. Once the function returns, either a surface intersection was detected and handled in the *Closest Hit* program or the *Miss* program was executed as fallback. In any case, the resulting color value is copied from the ray payload to the respective position in the output buffer.

### 3.3.2 Bounding Box

The capability of building acceleration structures over arbitrary geometry is achieved with user-supplied *Bounding Box* programs. These are executed for each primitive in the scene graph, i.e. for each point. As in the original implementation of the algorithm by Adamson & Alexa [2003a], the sphere of trust around each point is closely encompassed by an axis-aligned bounding box. Thus, the program simply reads the center and radius of the corresponding point from the global buffer and returns the minimum and maximum corner by componentwise subtraction and addition, respectively.

### 3.3.3 Intersection

Once scene traversal detects a bounding volume intersection, the *Intersection* program is executed for the validation and exact computation of the potential surface intersection. As the axis-aligned bounding box of each point tightly encompasses its sphere of trust, the principal part of the algorithm is implemented in this program, closely following the schematic outline presented in section 2.2. By intersecting the initial precomputed surface approximation, a suitable starting point is determined, with all required data being stored in the global buffers listed in section 3.2. Iteratively constructing a local coordinate system, computing the moving least squares approximation and intersecting the resulting polynomial yields the final ray-surface intersection. Each iteration is checked against the sphere of trust, with failure directly resulting in program termination.

The implementation of the first two steps directly follows the mathematical illustration of the algorithm in section 2.2, i.e. the local support plane is determined based on a weighted average of the neighboring normal vectors provided by FPM and the moving least squares polynomial approximation is solved by standard numerical means, e.g. Gaussian elimination, conjugate gradient method or QR decomposition. However, the actual ray-polynomial intersection computation is yet to be described and will be discussed in the following.

Computational complexity is reduced by transforming the intersection problem into the local coordinate system of the moving least squares approximation, defined

by an origin $\boldsymbol{o}$ and perpendicular vectors $\boldsymbol{u}$, $\boldsymbol{v}$ and $\boldsymbol{n}$, where $\boldsymbol{n}$ is the normal of the support plane spanned by $\boldsymbol{u}$ and $\boldsymbol{v}$. The parametrized interpolated surface can be expressed in world space as the linear combination

$$\boldsymbol{s}(x, y) = \boldsymbol{o} + x \cdot \boldsymbol{u} + y \cdot \boldsymbol{v} + f(x, y) \cdot \boldsymbol{n} \tag{3.1}$$

where $f(x, y)$ is the bivariate interpolating polynomial. In this thesis, quadratic and cubic polynomials are evaluated, i.e.:

$$f_2(x, y) = a_0 + a_1 x + a_2 y + a_3 x^2 + a_4 y^2 + a_5 xy$$
$$f_3(x, y) = a_0 + a_1 x + a_2 y + a_3 x^2 + a_4 y^2 + a_5 xy + a_6 x^3 + a_7 y^3 + a_8 x^2 y + a_9 x y^2$$

Given a ray $\boldsymbol{r}(t) = \boldsymbol{o}_{ray} + t \cdot \boldsymbol{d}_{ray}$ in world space, projection yields the transformed local ray components:

$$\begin{pmatrix} o_x \\ o_y \\ o_f \end{pmatrix} = \begin{pmatrix} \langle \mathbf{u}, \mathbf{o_{ray}} - \mathbf{o} \rangle \\ \langle \mathbf{v}, \mathbf{o_{ray}} - \mathbf{o} \rangle \\ \langle \mathbf{n}, \mathbf{o_{ray}} - \mathbf{o} \rangle \end{pmatrix} \qquad \begin{pmatrix} d_x \\ d_y \\ d_f \end{pmatrix} = \begin{pmatrix} \langle \mathbf{u}, \mathbf{d_{ray}} \rangle \\ \langle \mathbf{v}, \mathbf{d_{ray}} \rangle \\ \langle \mathbf{n}, \mathbf{d_{ray}} \rangle \end{pmatrix} \tag{3.2}$$

As the distance of the ray origin, i.e. the virtual camera, is expected to be large in comparison to the sphere of trust radii, the projected ray origin is shifted along the ray towards the vicinity of the estimated intersection location, in order to improve the numerical properties of the subsequent intersection computation:

$$\begin{pmatrix} o'_x \\ o'_y \\ o'_f \end{pmatrix} = \begin{pmatrix} o_x \\ o_y \\ o_f \end{pmatrix} + t_{offset} \cdot \begin{pmatrix} d_x \\ d_y \\ d_f \end{pmatrix} \tag{3.3}$$

where $t_{offset} = ||\boldsymbol{o}_{ray} - \boldsymbol{o}||$ is the distance of the ray origin from the local coordinate system origin in world space. Hence, the ray in the local coordinate system can be expressed as:

$$\begin{pmatrix} x(t) \\ y(t) \\ r(t) \end{pmatrix} = \begin{pmatrix} o'_x \\ o'_y \\ o'_f \end{pmatrix} + t \cdot \begin{pmatrix} d_x \\ d_y \\ d_f \end{pmatrix} \tag{3.4}$$

Substituting $x(t)$ and $y(t)$ in the bivariate moving least squares polynomial $f(x, y)$ yields the one-dimensional auxiliary polynomial $f(t)$, basically representing the height of the surface above the orthogonal projection of the ray onto the local support plane spanned by $\boldsymbol{u}$ and $\boldsymbol{v}$. At this stage, the aforementioned $t_{offset}$ shift results in numerically stable coefficients of $f(t)$, each close to unit magnitude.

Consequently, the ray-surface intersection can be transformed to a simple one-dimensional root-finding problem:

$$f(t) = r(t) \iff h(t) = f(t) - r(t) = 0 \tag{3.5}$$

While the quadratic case can be solved directly with little computational effort, a numerical approximation scheme is beneficial for cubic polynomials. In this thesis, both Newton's method and a bisection approach were implemented. While the former requires a starting point close to the actual root, a suitable starting interval containing a single change of sign is necessary for the latter in order to converge. Although minimum and maximum values of $t$ are obviously given by the intersections of the ray with the sphere of trust, the acquired limits are not sufficiently restrictive. Thus, both algorithms share a common initialization procedure, performing uniform sampling of the potential intersection interval for sign changes. The number of required subdivision steps evidently depends on the complexity and curvature of the interpolation polynomials. In practice, small values around 5 to 10 yield stable results and little performance impact.

As multiple ray-surface intersections may occur, i.e. two in the quadratic and three in the cubic case, each must be validated individually by checking its location in world space against the sphere of trust. The closest valid intersection, i.e. smallest $t$, serves as reference point for the next iteration or the *Intersection* program is terminated if validity can not be achieved by any candidate.

Eventually, as soon as the iterative algorithm has converged to a final ray-surface intersection at $t_{intersect}$, the associated (unnormalized) normal vector $\boldsymbol{n}_{intersect}$ is determined based on the surface tangents, i.e. the partial derivatives of equation (3.1):

$$\boldsymbol{n}_{intersect} = \frac{\delta \boldsymbol{s}}{\delta x}(\bar{x}, \bar{y}) \times \frac{\delta \boldsymbol{s}}{\delta y}(\bar{x}, \bar{y}) \tag{3.6}$$

where

$$\begin{pmatrix} \bar{x} \\ \bar{y} \end{pmatrix} = \begin{pmatrix} o'_x \\ o'_y \end{pmatrix} + t_{intersect} \cdot \begin{pmatrix} d_x \\ d_y \end{pmatrix} \tag{3.7}$$

### 3.3.4 Closest Hit

Once scene traversal has determined the ray-surface intersection closest to the ray origin, shading operations are performed in the *Closest Hit* program. The normal computed by the *Intersection* program is input to a classic Phong illumination model. Optionally, real recursive specular reflection and refraction can be incorporated. If the recursion limit is not reached, two additional invocations of *rtTrace* perform scene traversal for the respective secondary rays. Performance can be improved with negligible impact on visual quality by reducing the maximum iteration limit of the surface intersection algorithm for secondary rays.

### 3.3.5 Others

The *Miss* program is executed for rays without any geometry intersection during scene traversal. Thus, the corresponding pixel in the image output buffer is set

to a constant background color. Alternatively, the ray can be intersected with a checkerboard plane, providing better visual cues when rendering with reflection and refraction effects.

Exceptional conditions like stack overflows are handled in the *Exception* program by printing a log message and setting the pixel color to a striking value. While relatively uninteresting for release candidate software, this mechanism is particularly beneficial during development and debugging.

Both *Any Hit* and *Selector Visit* programs are not required for the implementation of moving least squares point set surfaces in OptiX. While the former could be used for early shadow ray termination, the latter is definitely irrelevant due to the employed very flat scene graph hierarchy, with each point directly being a child of the root node.

# 4 Evaluation

Achieving high frame rates while maintaining a solid surface representation is crucial to the interactive visualization of point-based FPM data sets. This chapter evaluates the performance and visual quality of the presented implementation in OptiX at various configurations. All benchmarks and renderings were performed on the following system:

- Intel Core i5 750
- NVIDIA GeForce GTX 470 (1280MB VRAM)
- Windows 7 64 Bit
- NVIDIA Driver 301.42
- CUDA 4.2
- OptiX 2.5.0

## 4.1 Performance Analysis

This section discusses the general performance and presents a comparison to the original results by Adamson & Alexa [2003a] using common laser scanned models in computer graphics, i.e. the Stanford Bunny by the Stanford University Computer Graphics Laboratory[1] and the Cyberware Rabbit[2].

### 4.1.1 Stanford Bunny

Consisting of approximately 35000 surface points, the Stanford Bunny is comparable to typical medium FPM data set sizes and will be used for the general evaluation of the achievable performance at various settings.

Figure 4.1 shows a rendering based on two iterations of the iterative intersection algorithm. The surface is approximated by quadratic polynomials, which are determined using LU decomposition. At a resolution of 800x800, the scene is rendered at 1.67 frames per second, which is barely interactive. However, performing only one iteration yields very fluent 39.81 frames per second. The same effect is clearly noticeable in table 4.1 across all configurations. A single iteration allows rendering

[1] http://graphics.stanford.edu/data/3Dscanrep/
[2] http://www.cyberware.com/products/scanners/desktopSamples.html

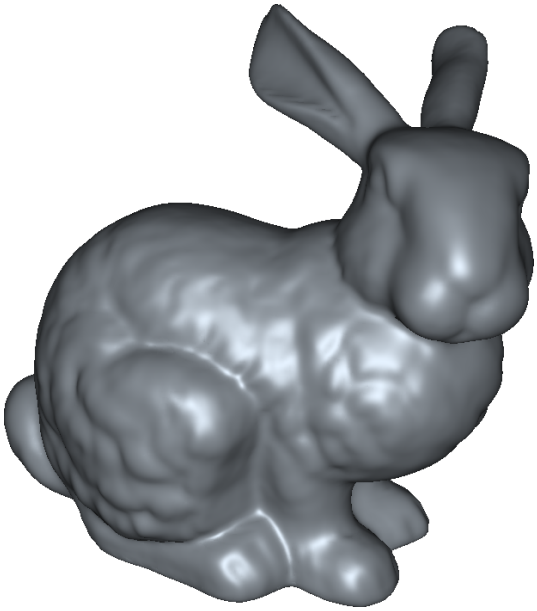

Figure 4.1: Stanford bunny. 2 iterations, quadratic polynomials, LU decomposition, 800x800 resolution, 1.67 fps.

at high frame rates, while multiple iterations suffer from great performance penalty. The former requires little computational effort as only the ray-surface intersections with the precomputed initial polynomials are determined on the GPU, whereas the latter results in the parallel creation and solution of many moving least squares systems. Reading and writing varying data from different memory locations in each thread contradicts the preferred memory access patterns in the CUDA architecture, thus quickly revealing its limitations.

| Iterations | Quadratic LU | Quadratic CG | Cubic Bisection LU | Cubic Bisection CG | Cubic Newton LU | Cubic Newton CG |
|---|---|---|---|---|---|---|
| 1 | 39.81 | 40.10 | 38.72 | 34.86 | 38.54 | 35.71 |
| 2 | 1.67 | 1.61 | 1.04 | 1.03 | 1.04 | 1.03 |
| 3 | 0.85 | 0.82 | 0.52 | 0.53 | 0.52 | 0.52 |
| 4 | 0.58 | 0.55 | 0.35 | 0.35 | 0.35 | 0.35 |
| 5 | 0.43 | 0.41 | 0.26 | 0.26 | 0.26 | 0.26 |

Table 4.1: Frame rates of Stanford Bunny in figure 4.1 at various iteration limits and configurations. The notable performance drop occuring in multiple iterations is caused by the parallel creation and solution of many moving least squares systems, i.e. for each ray. Quadratic/cubic polynomials, LU decomposition/conjugate gradient (CG) method, 800x800 resolution.

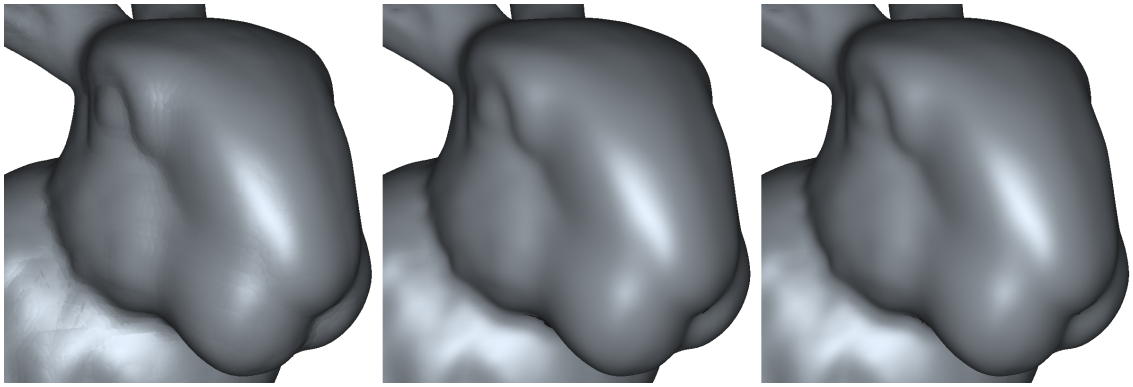

Figure 4.2: Close up of Stanford Bunny at various iterations: 1 (left, 34.1 fps), 2 (center, 1.53 fps) and 3 (right, 0.78 fps). Each rendered at 800x800 resolution, quadratic polynomials, LU decomposition.

In general, LU decomposition is only marginally faster than the conjugate gradient (CG) method, as the occurring linear systems are rather small. Likewise, there is no measurable performance difference between the bisection and Newton methods for the approximation of the exact ray-surface intersections, as described in section 3.3.3. Thus, the default configuration for cubic polynomials is set to the combination of LU decomposition and bisection, as these are expected to be the theoretically more stable procedures, even though they are visually indistinguishable. A basic QR decomposition using Householder reflections has been implemented, but requires an extraordinary amount of stack memory and performs extremely slow. In fact, the Windows watchdog timer terminates the process due to timeout. Notably, this effect contradicts general emperical knowledge, as QR decomposition usually provides better performance than LU decomposition for small matrices, indicating a lot of optimization potential in the implementation. Nevertheless, QR decomposition will not be further considered in this thesis, since LU decomposition and the conjugate gradient method are not affected by numerical problems.

There are multiple important observations regarding the development of the surface smoothness depending on the number of performed iterations. Since a single iteration results in very fluent and interactive frame rates, it is significant to examine its visual quality and usefulness. Figure 4.2 shows several close-up renderings of the Stanford Bunny using one (left), two (center) and three (right) iterations, respectively. Interestingly, a single iteration is already relatively smooth and shows only minor edges at the boundaries of the regions of trust around each point. While there is a distinct improvement using two iterations, the immense drop in rendering performance, i.e. from 34.1 fps down to 1.53 fps, definitely needs to be considered. By contrast, performing three iterations results in no visual improvement of the depicted scene, even at high magnification. Consequently, the presented implementation in OptiX implies a severe trade-off between high rendering performance and

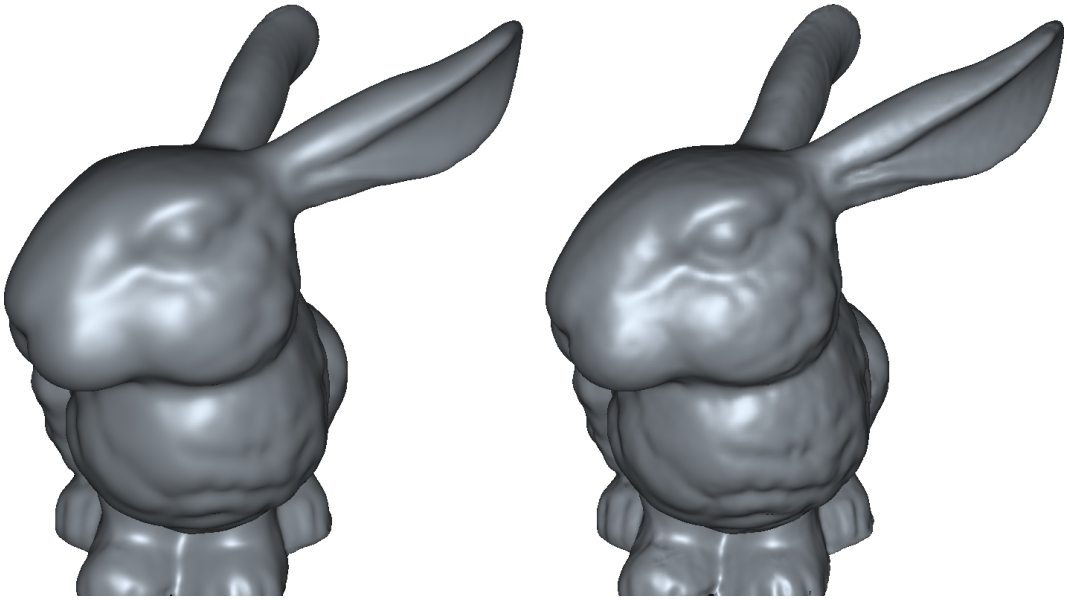

Figure 4.3: Comparison of quadratic (left, 1.77 fps) and cubic (right, 1.10 fps) polynomials applied to Stanford Bunny. Both using 2 iterations, 800x800 resolution, LU decomposition. Cubic polynomials are intersected using bisection approach with 10 sampling and 10 refinement steps.

smooth surface quality, which is basically reduced to a single optional refinement iteration performed on the GPU.

While the various solution approaches to the moving least squares system, i.e. LU decomposition and conjugate gradient method, are rather interchangeable, the choice of the actual polynomials to be determined is of central importance. Figure 4.3 depicts a comparison of quadratic (left) and cubic (right) interpolation polynomials. Cubic polynomials clearly provide a greater richness of detail and structure, whereas in comparison quadratic polynomials look fairly smoothed out. As stated in table 4.1, the difference in performance of the sole intersection computation on the GPU is only marginal, i.e. rendering a single iteration yields 39.81 fps using quadratic and 38.82 fps using cubic polynomials with bisection approach. However, performing multiple iterations reveals a noticeable performance penalty for the computation of cubic polynomial coefficients on the GPU.

So far only the attainability of certain frame rates has been discussed in various scenarios and configurations. However, the implementation of moving least squares point set surfaces in OptiX requires a time-consuming initialization procedure, which needs to be taken into consideration when discussing performance, as well. A break down of the loading procedures, computations and data transfers involved during initialization is presented in table 4.2. Since the Stanford Bunny is parsed from a

| Process | Seconds | Percentage |
|---|---|---|
| Read data set | 1.061 | 12.9% |
| Create OptiX context | 0.102 | 1.2% |
| Fill octree | 0.023 | 0.3% |
| Determine nearest neighbors | 1.826 | 22.2% |
| Compute initial MLS surface | 1.309 | 15.9% |
| Copy buffers to GPU | 0.007 | 0.1% |
| Create scene graph | 1.218 | 14.8% |
| Validate & compile context | 2.675 | 32.6% |
| Total | 8.221 | 100.0% |

Table 4.2: Loading time break down of the Stanford Bunny. 35 000 points, 50 nearest neighbors per point, quadratic initial polynomials.

PLY file in ASCII format, reading takes a relatively long time. Nevertheless, this process can be drastically reduced by the utilization of optimized binary file formats. In contrast, summing up to almost 40%, the determination of each point's nearest neighbors and the subsequent moving least squares computation of the initial interpolation polynomials represent a major contribution to the loading time, primarily dependent on the data set size and bound by CPU processing power. The creation of the scene graph and the compilation of the OptiX context constitute another central activity, mainly performed on the GPU. Besides the construction of the acceleration structure, this process is a black box, thus offering little potential for optimization. Although the total loading time of 8.2 seconds is negligible for a static scene such as the Stanford Bunny, the situation is worse for animated data sets with each time step being represented by a different point set. This drawback will be further discussed in section 4.2.

### 4.1.2 Cyberware Rabbit

Defined by 67000 points, the surface of the Cyberware Rabbit features a data set size twice as large as the Stanford Bunny. Initially employed by Adamson & Alexa [2003a] in order to evaluate the original algorithm, the rabbit model is used in this section to assess the performance boost of the presented GPU-based implementation in OptiX.

Figure 4.4 shows three renderings of the Cyberware Rabbit model. At a resolution of 400x765, the left image is rendered at 42.4 fps using one iteration, while both the center and right scenes yield a frame rate of 0.68 fps using two iterations. Furthermore, the left and center renderings employ the default approximation locality parameter $h$ of 1.5 times the average region of trust radius, whereas the right image demonstrates a clearly visible smoothing effect by applying a 20-times increase to $h$. Interestingly, the left (one iteration) and center (two iterations) images

are visually hardly distinguishable at this distance, though differing greatly in rendering performance. As expected, the frame rate is not influenced by the choice of $h$.

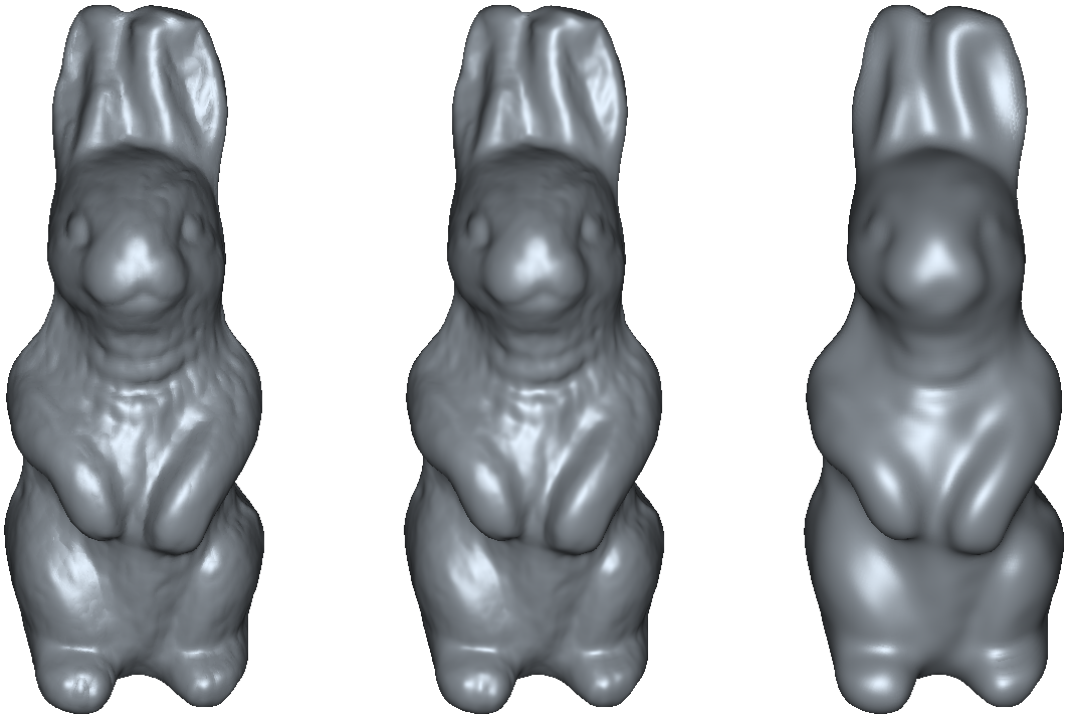

Figure 4.4: Cyberware Rabbit. Left: 1 iteration, 42.4 fps. Center: 2 iterations, 0.68 fps. Right: 2 iterations, 0.68 fps, h increased by factor 20x. All images rendered at 400x765 using quadratic polynomials and LU decomposition.

The scene depicted in the center of figure 4.4 was rendered by Adamson & Alexa using quadratic polynomials at a resolution of 200x400 in order to evaluate the performance of the original implementation. Table 4.3 summarizes their findings and shows the speedup that can be achieved using the presented GPU-based approach

| Iterations | OptiX | Adamson & Alexa | Speedup |
|---|---|---|---|
| 1 | 52.93 fps | - | - |
| 2 | 5.63 fps | 7.9 sec | 44x |
| 3 | 2.99 fps | 11.5 sec | 34x |
| 5 | 1.54 fps | 18.9 sec | 29x |
| 10 | 1.45 sec | 42.7 sec | 29x |

Table 4.3: Frame rate comparison of the OptiX implementation versus the original results by Adamson & Alexa [2003a], using Cyberware Rabbit model in figure 4.4 at 200x400 resolution and quadratic polynomials.

in OptiX. Although the frame rate increase of 44x using two iterations seems very promising at first glance, the results are relativized considering the original benchmarks were executed on a 2 GHz Pentium 4 processor, eight years before the actual release of the NVIDIA GTX 470 graphics processor employed in this thesis. Therefore, an up to date measurement on a contemporary CPU would be very interesting.

Concludingly, a loading time break down of the Cyberware Rabbit data set is listed in table 4.4. The general distribution on a percentage basis is similar to the break down of the Stanford Bunny loading procedure presented in table 4.2. Since the rabbit is available in binary PLY format, the required time for reading the data set is drastically reduced. As expected, the double amount of point data compared to the Stanford Bunny almost linearly manifests itself in the determination of the nearest neighbors, the moving least squares computation of the initial polynomials and the construction of the scene graph. Again, the validation and compilation of the OptiX context marks an uncontrollable major contribution to the total loading procedure, presumably including a longer construction time of the internal acceleration structure due to the increased point count.

| Process | Seconds | Percentage |
|---|---|---|
| Read data set | 0.124 | 1.0% |
| Create OptiX context | 0.103 | 0.9% |
| Fill octree | 0.043 | 0.4% |
| Determine nearest neighbors | 3.549 | 29.4% |
| Compute initial MLS surface | 2.553 | 21.2% |
| Copy buffers to GPU | 0.013 | 0.1% |
| Create scene graph | 2.304 | 19.1% |
| Validate & compile context | 3.363 | 27.9% |
| Total | 12.052 | 100.0% |

Table 4.4: Loading time break down of the Cyberware Rabbit. 67 000 points, 50 nearest neighbors per point, quadratic initial polynomials.

## 4.2 Visualization of the Finite Pointset Method

In order to create an interactive visualization method tailored to the grid-free Finite Pointset Method (FPM), this thesis presents an adaption of the moving least squares point set surfaces approach by Adamson & Alexa [2003a] to NVIDIA's GPU-based real-time ray tracing engine OptiX. While previous paragraphs discussed the general performance and visual quality using common models in computer graphics, this section focuses on the rendering of actual FPM data sets. Figure 4.5 shows several time steps of a sloshing fluid, each rendered at 3-4 frames per second using two iterations and quadratic polynomials with 1280x800 rays. The data set was

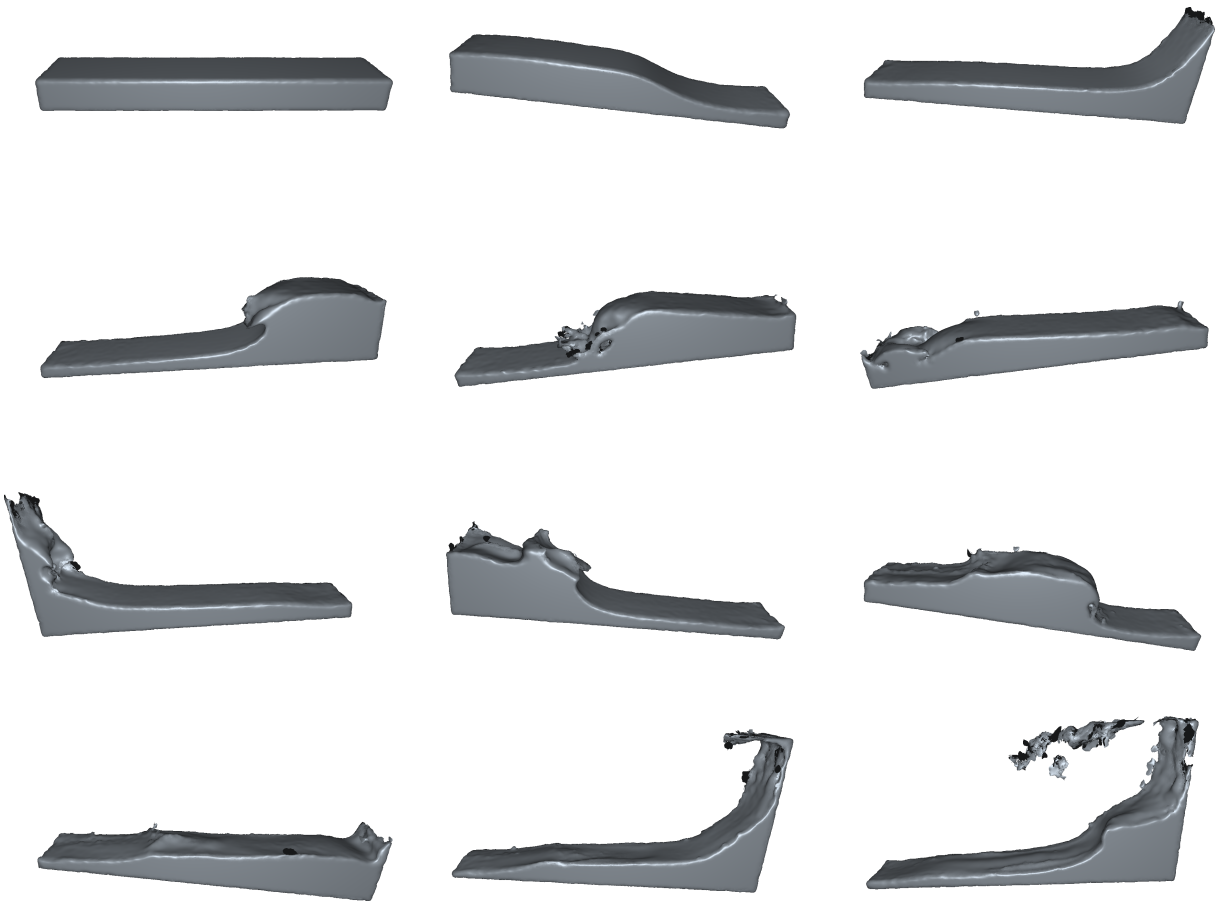

Figure 4.5: Several time steps of a sloshing fluid simulated in FPM, approximately consisting of 10000 surface points and ranging from time step 0 (top left) to time step 1100 (bottom right). Each using 2 iterations, quadratic polynomials, LU decomposition. 3-4 fps at 1280x800 resolution.

simulated in FPM at various resolutions, i.e. approximately 2500, 10 000 and 100 000 surface points. Including internal particles, the total number of points in the data set obviously is of greater magnitude. However, for the purpose of visualization in this thesis, only the surface subset flagged by FPM is relevant. Unless otherwise specified, all renderings and benchmarks provided in this section are based on the medium data set consisting of about 10 000 points.

Figure 4.6 depicts two close up renderings of time step 375 using two iterations, comparing quadratic (left) and cubic (right) interpolation polynomials. As already demonstrated in figure 4.3, cubic polynomials provide greater structural complexity and details of the underlying point set. Admittedly, the resulting frame rate of only 0.56 frames per second at a resolution of 1280x800 is impractical. Although more

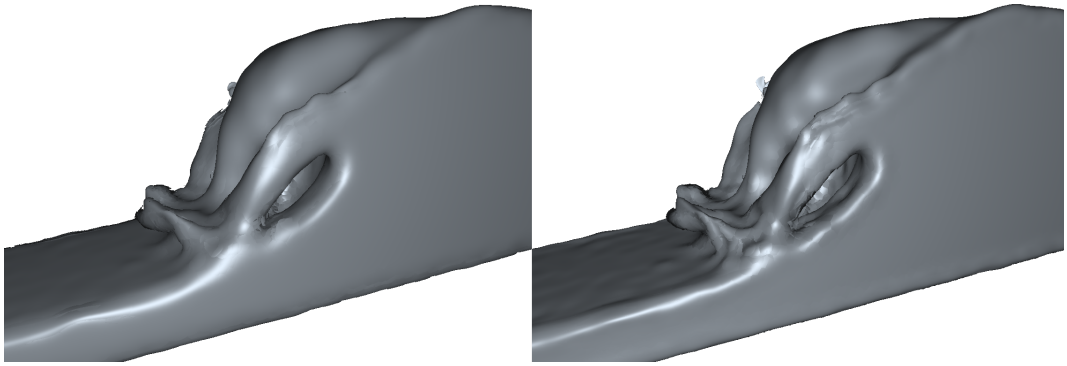

Figure 4.6: Comparison of quadratic (left, 1.20 fps) and cubic (right, 0.56 fps) polynomials at time step 375. Both using 2 iterations, 1280x800 resolution, LU decomposition.

than twice as fast, i.e. 1.20 fps, quadratic polynomials are still far from interactive. Looking closely, slight jumps of smoothness are observable at the borders of the regions of trust, which can be eliminated by increasing the number of precomputed nearest neighbors for each point. However, this approach intensifies the bottleneck effects of GPU memory access, thus further decreasing performance. Consequently, aiming at preferably high frame rates, only quadratic polynomials will be considered in the following. Nevertheless, cubic polynomials are still an interesting option for the creation of offline renderings with increased precision.

The impact of resolution and iteration limit on rendering speed is illustrated in table 4.5, showing various benchmarks of the left scene in figure 4.6. As expected, multiple iterations result in barely interactive frame rates, while a single iteration yields extraordinary performance even at 1600x800. Furthermore, quadrupling the resolution only results in approximately half rendering speed. This obviously leads to the central question, if and in which cases restricting the algorithm to a single iteration yields sufficiently precise and smooth results for FPM surface visualization.

Figure 4.7 compares time step 460 of the sloshing fluid data set using one (left)

| Resolution | 1 Iteration | 2 Iterations | 3 Iterations |
|---|---|---|---|
| 200x100 | 212.57 | 19.16 | 10.19 |
| 400x200 | 145.27 | 7.98 | 4.14 |
| 800x400 | 83.40 | 3.11 | 1.60 |
| 1600x800 | 36.24 | 1.17 | 0.55 |

Table 4.5: Frame rate of time step 375 depicted in left scene of figure 4.6 at various resolutions and iterations.

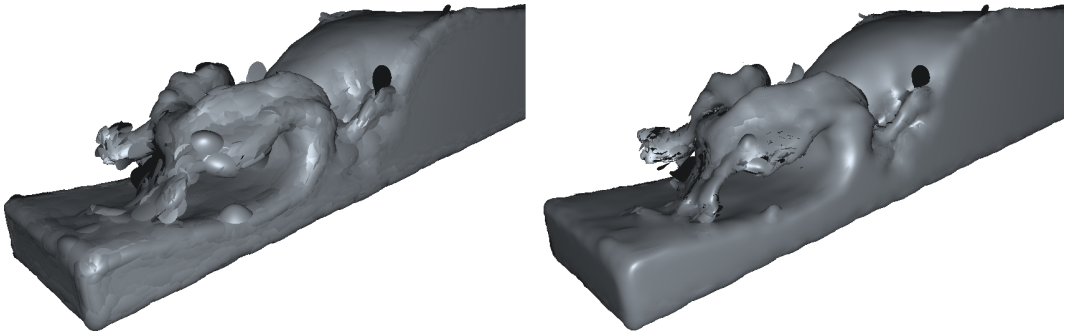

Figure 4.7: Comparison of one (left, 42.20 fps) and two (right, 1.64 fps) iterations at time step 460. 1280x800 resolution, quadratic polynomials, LU decomposition.

and two (right) iterations. As expected, while both rendered at 1280x800, the frame rate difference is notable, i.e. 42.20 fps versus 1.64 fps, respectively. Interestingly, although looking exaggerated and bulged at various locations combined with jumps in smoothness, the initial interpolation polynomials (left) nevertheless produce a remarkably accurate approximation of the converged results (right). Thus, it is recommendable to exploit the performance of a single iteration, i.e. solely using the GPU for intersecting the precomputed initial surfaces, during interactivity, e.g. camera movement, and execute further refinement iterations upon reaching idle state. The same scheme can be applied to cubic polynomials due to their slower yet similar performance behavior, as presented in table 4.1.

However, at closer inspection, figure 4.7 also reveals several rendering artifacts inherent to the algorithm. Various salient black spots and fragments caused by the employed Phong lighting model indicate back-facing surfaces which should not be visible. Especially prominent at the sloshing wave front, regions of sparse point density combined with non-uniform normals featuring high disparities in orientation provoke this effect. These critical areas contradict the algorithm's central assumption of interpolating a smooth sufficiently sampled surface.

The scene depicted in figure 4.8 takes this exceptional situation to the extreme by containing multiple clusters of sparsely distributed point clouds with inconsistent normal orientations, thus creating a striking amount of visualization artifacts. More importantly, the image is rendered using five iterations, thereby indicating the insufficiency of solely increasing the maximum number of refinement operations. While drastically increasing the radii of trust and employing higher-order polynomials might visually resolve surface smoothness and coherence, this approach contradicts the real cause of the artifacts. The aforementioned regions of sparsely distributed point sets actually represent clusters of droplets occurring in particle-based fluid simulation. Obviously, a rendering algorithm tailored to the needs of

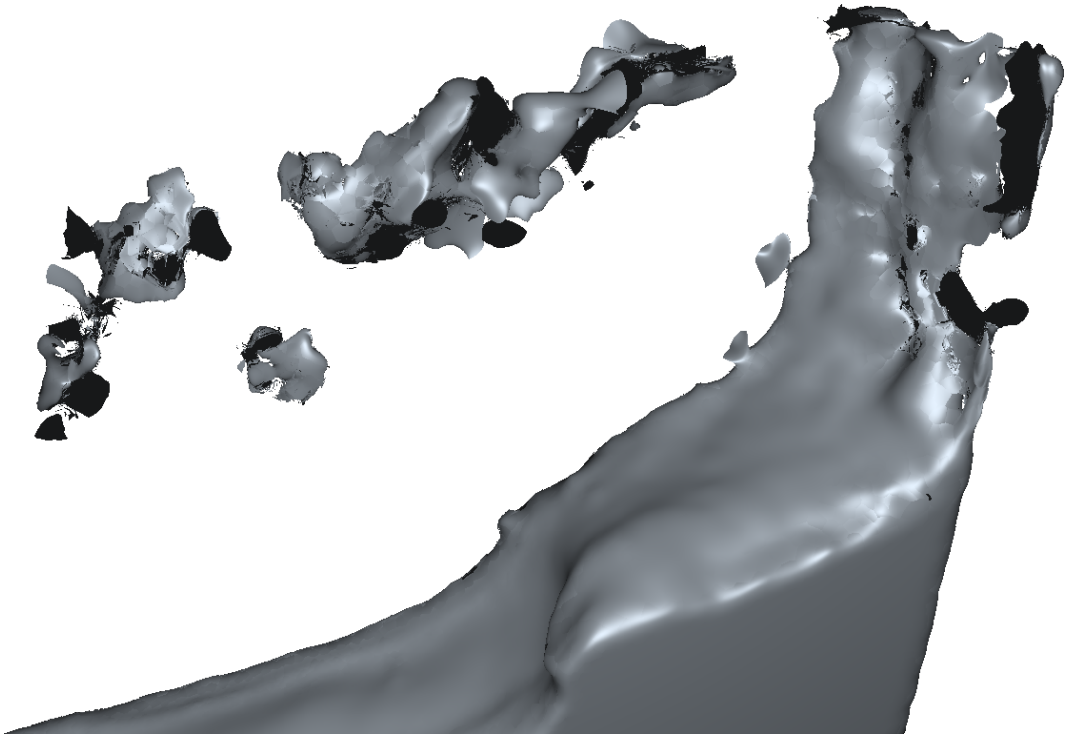

Figure 4.8: Clearly visible artifacts in sparse regions at time step 1100. 5 iterations, 0.4 fps, 1280x800 resolution, quadratic polynomials, LU decomposition.

the grid-free Finite Pointset Method should incorporate proper handling of these very common conditions. Isolation factors are computed for each point at any time step during FPM simulation, which might serve as additional input for advanced preprocessing of the point set in future work.

Since all previous figures demonstrate various effects using the same medium sized data set consisting of approximately 10 000 points, it is interesting to further analyze the impact of data set size on the rendering performance. Table 4.6 list a frame rate comparison of the initial time step as depicted in the top left of figure 4.5 for

| Points | 1 Iteration | 2 Iterations | 3 Iterations |
|---|---|---|---|
| 2500 | 73.05 | 4.18 | 2.17 |
| 10000 | 65.35 | 2.81 | 1.44 |
| 100000 | 28.43 | 1.47 | 0.75 |

Table 4.6: Frame rate of time step 0 (top left of figure 4.5) at various data set sizes and iterations. 1280x800 resolution, quadratic polynomials, LU decomposition.

| Process | 2500 Points Seconds | 2500 Points Percentage | 10000 Points Seconds | 10000 Points Percentage | 100000 Points Seconds | 100000 Points Percentage |
|---|---|---|---|---|---|---|
| Read data set | 0.054 | 5.4% | 0.046 | 2.2% | 0.032 | 0.2% |
| Create OptiX context | 0.109 | 10.8% | 0.111 | 5.3% | 0.113 | 0.8% |
| Fill octree | 0.001 | 0.1% | 0.005 | 0.2% | 0.070 | 0.5% |
| Determine nearest neighbors | 0.071 | 7.0% | 0.397 | 19.0% | 2.730 | 19.3% |
| Compute initial MLS surface | 0.080 | 7.9% | 0.315 | 15.1% | 3.341 | 23.6% |
| Copy buffers to GPU | 0.001 | 0.1% | 0.001 | 0.0% | 0.023 | 0.1% |
| Create scene graph | 0.137 | 13.6% | 0.349 | 16.7% | 3.241 | 22.0% |
| Validate & compile context | 0.555 | 55.1% | 0.869 | 41.5% | 4.626 | 32.6% |
| Total | 1.008 | 100.0% | 2.093 | 100.0% | 14.176 | 100.0% |

Table 4.7: Loading time break down of the initial time step in the sloshing fluid simulation (top left of figure 4.5) at various data set sizes. 50 nearest neighbors per point, quadratic initial polynomials.

various data set sizes and iteration limits, using quadratic polynomials at 1280x800 resolution. Interestingly, a ten-fold increase in point set size results in a performance drop by only factor two, thus emphasizing the logarithmic nature of the underlying bounding volume hierarchy acceleration structure.

Besides the attainability of real-time rendering speeds, fast loading of new time steps is another crucial factor for interactivity in a FPM simulation consisting of hundreds to thousands of independent time steps. Table 4.7 presents a detailed loading time break down for various data set sizes. As expected, the main contributions are the determination of the nearest neighbors, the computation of the initial moving least squares surface approximations, the construction of the scene graph and eventually the compilation of the OptiX context. Apart from the latter, an almost linear increase in computation time is evident. File loading is very fast, since FPM simulations are stored in binary EnSight 6 format. Again, the creation of the OptiX context requires constant time, as already seen in the break downs of the Stanford Bunny and the Cyberware Rabbit models. Interestingly, the percentage share of the final validation and compilation of the context decreases with data set size, indicating a partially constant overhead. Even the smallest simulation consisting of only 2500 points already requires a total preprocessing and loading time of one second, not to mention the 14 seconds necessary for the greater version based on about 100 000 particles. Obviously, this renders dynamic reloading of data useless for an interactive navigation through all simulated time steps. While GPU memory may provide enough space for storing multiple precomputed time steps depending on the number of points, the total accumulated loading times are critical and far from negligible.

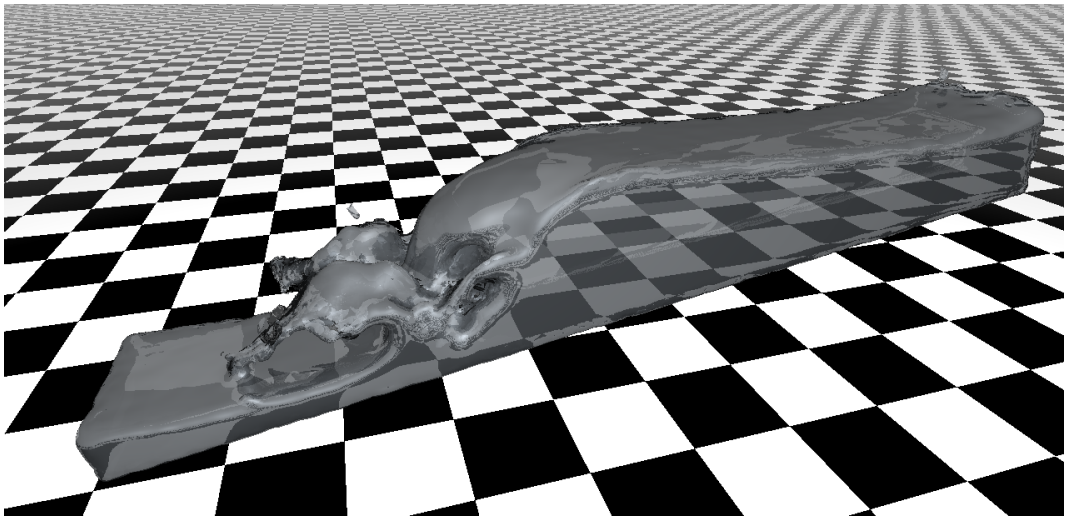

Figure 4.9: Reflection and refraction used in water shader for rendering of time step 440. 2 iterations, 2 recursions, 1270x610 resolution, quadratic polynomials, LU decomposition, 4.59 seconds per frame.

This evaluation is concluded by a demonstration of recursive ray tracing in OptiX in order to realize a water shader, incorporating reflection and refraction computation. Figure 4.9 depicts time step 440 of the sloshing fluid simulation based on two iterations of the surface intersection algorithm and a maximum secondary ray depth of two recursive invocations. A checkerboard replaces the constant background color for highlighting purposes by producing interesting refraction effects, e.g. at the top bulge of the wave front. Based on quadratic polynomials, an image of 1270x610 resolution requires a rendering time of 4.59 seconds. While basically irrelevant for offline rendering, e.g. the creation of high fidelity presentable demonstration videos, this low frame rate is close to the critical time out threshold induced by the aforementioned Windows display driver watchdog. There are several solutions to circumvent this exception, e.g. by using a separate graphics device for CUDA computations or employing a Linux-based operating system.

# 5 Conclusion

## 5.1 Summary

In order to develop an interactive real-time visualization of the Finite Pointset Method, the iterative moving least squares approach by Adamson & Alexa has been successfully implemented in NVIDIA's ray tracing framework OptiX.

Performing a single iteration of the algorithm allows rendering at highly interactive frame rates, while multiple iterations suffer from great performance penalty. The former requires little computational effort as initial surface approximations are precomputed and stored for each sphere of trust, thus only resulting in the determination of ray-surface intersections on the GPU. In contrast, the additional creation and solution of a multitude of moving least squares systems involved in the latter quickly reveals the memory access bottleneck of modern graphics hardware. Thus, it is beneficial to exploit the performance of a single iteration during interactivity, e.g. camera movement, and execute further refinement operations in idle state. Typically, two iterations are sufficient for convergence. However, even the precomputed initial surface approximations can provide adequate precision in certain situations.

This approach is practical for both quadratic and cubic interpolation polynomials. Interestingly, the impact of polynomial order on frame rate is only of minor degree, thus allowing a choice primarily depending on the desired level of surface smoothing and richness of detail.

Although yielding very smooth results in surface regions defined by dense point sets featuring well-oriented normals, the original algorithm is only suitable for the visualization of the Finite Pointset Method to a limited extent. Sparsely distributed point clusters with inconsistent normal orientations lead to striking rendering artifacts, which are not affected by an increased iteration limit. Furthermore, loading procedures of several seconds render the presented approach useless for the interactive navigation through simulations consisting of hundreds to thousands of time steps.

## 5.2 Open Questions and Future Work

Although the performance of the OptiX implementation can theoretically be compared to the initial results published by Adamson & Alexa, the obtained speedups

obviously have to be treated with caution due to more than eight years in hardware development between the respective contemporary system setups.

Thus, a benchmark of the original implementation on a modern high-end consumer CPU, e.g. Intel Ivy Bridge, versus the OptiX adaption on the latest general purpose GPU generation by NVIDIA, i.e. Kepler, would be of great interest. Besides advancements in processing power and parallelization, up-to-date graphics devices feature reduced memory access penalties and presumably lower the constant initialization overhead introduced by OptiX, which is crucial for faster loading times.

However, not only the attainability of higher frame rates needs to be tackled. As demonstrated and evaluated in this thesis, the application of the original iterative algorithm without enhancements is not sufficient for the visualization of the Finite Pointset Method. Further preprocessing combined with the consideration of advanced attributes and parameters provided by FPM, e.g. the isolation factor of each point, needs to be incorporated in the determination of surface clusters and separated droplets.

# Bibliography


Adams, B., Keiser, R., Pauly, M., Guibas, L. J., Gross, M., & Dutré, P. (2005). Efficient raytracing of deforming pointsampled surfaces. *Computer Graphics Forum*, 24, 677–684.

Adamson, A. & Alexa, M. (2003a). Approximating and intersecting surfaces from points. In *Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, SGP '03 (pp. 230–239). Aire-la-Ville, Switzerland, Switzerland: Eurographics Association.

Adamson, A. & Alexa, M. (2003b). Ray tracing point set surfaces. In *Proceedings of the Shape Modeling International 2003*, SMI '03 (pp. 272–282). Washington, DC, USA: IEEE Computer Society.

Aila, T. & Laine, S. (2009). Understanding the efficiency of ray traversal on gpus. In *Proc. High-Performance Graphics 2009* (pp. 145–149).

Alexa, M., Behr, J., Cohen-Or, D., Fleishman, S., Levin, D., & Silva, C. T. (2001). Point set surfaces. In *Proceedings of the conference on Visualization '01*, VIS '01 (pp. 21–28). Washington, DC, USA: IEEE Computer Society.

Amenta, N., Bern, M., & Kamvysselis, M. (1998). A new voronoi-based surface reconstruction algorithm. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98 (pp. 415–421). New York, NY, USA: ACM.

Amenta, N., Choi, S., & Kolluri, R. K. (2000). The power crust, unions of balls, and the medial axis transform. *Computational Geometry: Theory and Applications*, 19, 127–153.

Beatson, R. K. & Newsam, G. N. (1998). Fast evaluation of radial basis functions: Moment-based methods. *SIAM J. Sci. Comput.*, 19(5), 1428–1449.

Boissonnat, J.-D. (1984). Geometric structures for three-dimensional shape representation. *ACM Transactions on Graphics*, 3(4), 266–286.

Cheng, Z.-Q., Wang, Y.-Z., Li, B., Xu, K., Dang, G., & Jin, S.-Y. (2008). A Survey of Methods for Moving Least Squares Surfaces. In *Eurographics/IEEE VGTC on Volume and Point-Based Graphics* (pp. 9–23). Los Angeles, California, USA: Eurographics Association.

Dey, T. K. & Goswami, S. (2003). Tight cocone: a water-tight surface reconstructor. In *Proceedings of the eighth ACM symposium on Solid modeling and applications*, SM '03 (pp. 127–134). New York, NY, USA: ACM.

Dyn, N., Floater, M. S., & Iske, A. (2002). Adaptive thinning for bivariate scattered data. *J. Comput. Appl. Math*, 145, 505–517.

Edelsbrunner, H. & Mücke, E. P. (1994). Three-dimensional alpha shapes. *ACM Trans. Graph.*, 13(1), 43–72.

Floater, M. S. & Iske, A. (1998). Thinning algorithms for scattered data interpolation. *BIT*, 38, 705–720.

Garanzha, K., Pantaleoni, J., & McAllister, D. (2011). Simpler and faster hlbvh with work queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11 (pp. 59–64).

Goshtasby, A. & O'Neill, W. D. (1993). Surface fitting to scattered data by a sum of gaussians. *Comput. Aided Geom. Des.*, 10(2), 143–156.

Green, S. A. & Paddon, D. J. (1990). A highly flexible multiprocessor solution for ray tracing. *The Visual Computer*, 6, 62–73.

Gross, M. & Pfister, H. (2007). *Point-Based Graphics*. Morgan Kaufman.

Hoppe, H., DeRose, T., Duchamp, T., Halstead, M., Jin, H., McDonald, J., Schweitzer, J., & Stuetzle, W. (1994). Piecewise smooth surface reconstruction. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '94 (pp. 295–302). New York, NY, USA: ACM.

Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., & Stuetzle, W. (1992). Surface reconstruction from unorganized points. *SIGGRAPH Comput. Graph.*, 26(2), 71–78.

Horn, D. R., Sugerman, J., Houston, M., & Hanrahan, P. (2007). Interactive k-d tree gpu raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, I3D '07 (pp. 167–174). New York, NY, USA: ACM.

Iske, A. (1999). Hierarchical scattered data filtering for multilevel interpolation schemes. In *In Mathematical methods for curves and surfaces* (pp. 211–221).: Vanderbilt Univ. Press.

Kalaiah, A. & Varshney, A. (2001). Differential point rendering. In *In Proceedings of the 12th Eurographics Workshop on Rendering* (pp. 139–150).

Kobbelt, L. & Botsch, M. (2004). A survey of point-based techniques in computer graphics. *Comput. Graph.*, 28(6), 801–814.

Lei, Z., Blane, M. M., & Cooper, D. B. (1996). 3l fitting of higher degree implicit polynomials. In *In Proceedings of Third IEEE Workshop on Applications of Computer Vision* (pp. 148–153).

Levin, D. (1998). The approximation power of moving least-squares. *Mathematics of Computation*, 67, 1517–1531.

Muuss, M. J. (1995). Towards real-time ray-tracing of combinatorial solid geometric models.

Ohtake, Y., Belyaev, A., Alexa, M., Turk, G., & Seidel, H.-P. (2003a). Multi-level partition of unity implicits. *ACM Trans. Graph.*, 22(3), 463–470.

Ohtake, Y., Belyaev, A., & Seidel, H.-P. (2003b). A multi-scale approach to 3d scattered data interpolation with compactly supported basis functions. In *Proceedings of the Shape Modeling International 2003*, SMI '03 (pp. 153–). Washington, DC, USA: IEEE Computer Society.

Parker, S., Martin, W., pike J. Sloan, P., Shirley, P., Smits, B., & Hansen, C. (1999). Interactive ray tracing.

Parker, S. G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., & Stich, M. (2010). Optix: a general purpose ray tracing engine. In *ACM SIGGRAPH 2010 papers*, SIGGRAPH '10 (pp. 66:1–66:13). New York, NY, USA: ACM.

Pauly, M., Keiser, R., Adams, B., Dutré, P., Gross, M., & Guibas, L. J. (2005). Meshless animation of fracturing solids. *ACM Trans. Graph.*, 24(3), 957–964.

Pfister, H., Zwicker, M., van Baar, J., & Gross, M. (2000). Surfels: surface elements as rendering primitives. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00 (pp. 335–342). New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.

Popov, S., Günther, J., Seidel, H.-P., & Slusallek, P. (2007). Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26(3), 415–424. (Proceedings of Eurographics).

Pratt, V. (1987). Direct least-squares fitting of algebraic surfaces. *SIGGRAPH Comput. Graph.*, 21(4), 145–152.

Rusinkiewicz, S. & Levoy, M. (2000). Qsplat: a multiresolution point rendering system for large meshes. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00 (pp. 343–352). New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.

Sainz, M. & Pajarola, R. (2004). Point-based rendering techniques. *Computers & Graphics*, 28, 869–879.

Savchenko, V., Pasko, E. A., Okunev, O. G., & Kunii, T. L. (1995). Function representation of solids reconstructed from scattered surface points and contours. *Computer Graphics Forum*, 14, 181–188.

Schaufler, G. & Jensen, H. W. (2000). Ray tracing point sampled geometry. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000* (pp. 319–328). London, UK, UK: Springer-Verlag.

Stich, M., Friedrich, H., & Dietrich, A. (2009). Spatial splits in bounding volume hierarchies. In *Proceedings of High-Performance Graphics 2009*.

Tejada, E., Gois, J. P., Nonato, L. G., Castelo, A., & Ertl, T. (2006). Hardware-accelerated extraction and rendering of point set surfaces. In *EuroVis* (pp. 21–28).

Turk, G. & O'brien, J. F. (2002). Modelling with implicit surfaces that interpolate. *ACM Trans. Graph.*, 21(4), 855–873.

Wald, I., Mark, W. R., Günther, J., Boulos, S., Ize, T., Hunt, W., Parker, S. G., & Shirley, P. (2007). State of the art in ray tracing animated scenes. In D. Schmalstieg & J. Bittner (Eds.), *STAR Proceedings of Eurographics 2007* (pp. 89–116).: The Eurographics Association.

Wald, I. & Seidel, H.-P. (2005). Interactive ray tracing of point-based models. In *ACM SIGGRAPH 2005 Sketches*, SIGGRAPH '05 New York, NY, USA: ACM.

Wendland, H. (1995). Piecewise polynomial, positive definite and compactly supported radial functions of minimal degree. *Advances in Computational Mathematics*, 4, 389–396. 10.1007/BF02123482.

Wicke, M., Steinemann, D., & Gross, M. H. (2005). Efficient animation of point-sampled thin shells. *Comput. Graph. Forum*, (pp. 667–676).

Zwicker, M., Pfister, H., van Baar, J., & Gross, M. (2002). Ewa splatting. *IEEE Transactions on Visualization and Computer Graphics*, 8(3), 223–238.

# Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, habe ich durch die Angabe der Quelle, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht.

Kaiserslautern, den 25. September 2012 _____

Tim Biedert