Motion Blur with point-based rendering

Motion Blur with Point-Based Rendering

Johanna Wolf

Master Thesis
March - August 2008

Prof. Dr. Markus Gross

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

cgl
Computer Graphics Laboratory ETH Zurich

Abstract

This thesis seeks to apply photo-realistic motion blur to point-based rendering. Its solution is built upon EWA splatting, a technique for rendering point sampled geometry by elliptical splat primitives with high-quality anisotropic antialiasing. We present a consistent extension of the theoretical base in the time dimension and introduce ellipsoids with spatial and temporal dimensionality as new rendering primitives. Specifically, the surface reconstruction kernels are extended by a temporal dimension along the instantaneous velocity. Finally, we describe an implementation of the entire adapted point rendering pipeline using vertex and fragment programs of current GPUs.

Zusammenfassung

Diese Arbeit nimmt sich zum Ziel, fotorealistisches Motion Blur auf punktbasiertes Rendering anzuwenden. Die vorgestellte Lösung basiert auf EWA Splatting, einer Methode welche punktbasierte Objekte mittels elliptischer Splat Primitive rendert und hochqualitatives anisotropisches Antialiasing erzielt. Wir stellen eine konsistente Erweiterung der zugrundeliegenden Theorie in zeitlicher Dimension vor und führen Ellipsoide als Renderinprimitive ein, welche eine räumliche und zeitliche Dimension in sich vereinen. Spezifisch werden die Kernels zur Oberflächenrekonstruktion durch eine zeitliche Dimension entlang des Geschwindigkeitsvektors erweitert. Desweiteren beschreiben wir eine Implementation der gesamten angepassten Punktrendering-Pipeline unter Verwendung von Vertex- und Fragmentprogrammen gegenwärtiger GPUs.

Contents

List of Figures

List of Tables

1

Introduction

Motion blur denotes the visual effect that appears in still images or film sequences when objects moving with rapid velocity are captured. The image is perceived as smeared or blurred along the direction of relative motion to the camera. The reason for this is that the image taken by a camera does not represent a single instant of time, but in fact an integration of the incoming light over the period of exposure. This appears natural because the human eye behaves in much the same way.

In computer animation, however, each rendered frame displays a perfect instant in time. Due to this, the motion in the image sequence will be perceived as staggered or even incorrect if the frame rate falls below a certain number of frames per second. In other words, temporal aliasing occurs if the temporal sampling rate does not meet the Nyquist frequency of the screen space signal of the captured scene. If motion blur is applied on the other hand, these disturbing artifacts are alleviated and the sequence will appear realistic and continuous at the same frame rate. Similarly, in contrast to instantaneous still images, a motion blurred image gives the viewer hints about the ongoing motion and adds a sensation of dynamicism.

Therefore, the simulation of motion blur effects results in a method for temporal antialiasing. It seeks to reduce or completely remove the effects of insufficient temporal sampling by either supersampling, that is, producing frames as a composite of many time instants which are sampled below the displayed frame rate. Or - which is theoretically more rectified and therefore to be preferred - the incoming signal is bandlimited before sampling to guarantee that its Nyquist frequency is met. To the commonly used approximations of the motion blur effect count techniques such as image space post-processing, geometric deformations or blending of temporal supersamples.

On the other hand, because of their conceptual simplicity and superior flexibility, point-based geometries have evolved into a valuable alternative to surface representations based on polygo-

nal meshes. 3D scanning devices often produce huge volumes of point-sampled surface data that are difficult to process, edit and visualize. Instead of reconstructing consistent triangle meshes or higher order surface representations from the acquired data, point-based approaches work directly with the sampled points without requiring the computation of connectivity information or imposing topological manifold constraints or requirements on the sample distribution. Point rendering is particularly interesting for highly complex models, whose triangle representation requires millions of tiny primitives. The projected area of those triangles is often less than a few pixels, resulting in inefficient rendering because of the overhead for triangle setup and the rasterization stage becomes a bottle-neck. Point-based methods have proven to be efficient for processing, editing and rendering such data [RL00].

Well-established among these methods is the technique of surface splatting [ZPvBG01] which allows for rendering high-quality images of geometric objects that are given by a sufficiently dense set of sample points. The idea is to approximate local regions of the surface by planar ellipses in object space and then render the surface by accumulating and blending these ellipses in image space. From the geometric point of view, the set of ellipses defines a piecewise linear approximation of the given geometry, and the size and aspect ratio of the ellipses depend on the local principal curvatures (and the approximation tolerance prescribed by the user). To achieve high-quality anti-aliasing capabilities surface splatting is in general applied with the sophisticated *elliptical weighted average* (EWA) filtering which has been introduced by Heckbert [Hec89] in the context of texture mapping. The rendering algorithm basically represents the surface splats by elliptical Gaussian kernels and projectively maps them to image space. Before sampling at the pixel grid locations, the projected kernels are combined with a low-pass filter such that most sampling artifacts such as holes or aliasing can be effectively avoided.

Since the publication of the original software-based EWA splatting [ZPvBG01], several authors tried to map this technique to the GPU in order to exploit hardware acceleration. Due to the lacking support for splat primitives, these methods always have to find a trade-off between rendering quality and rendering performance. With the current generation of graphics processors, however, it is now possible to control a large part of the rasterization process [BHZK05].

This thesis seeks to apply motion blur to point-based rendering and will be structured as follows: First, Chapter 2 discusses previous work on related issues, Chapter 3 derives the theoretical basis of our proposed method and Chapter 3.6.4 describes the applied rendering algorithms. Chapter 4.6 presents results and performance measurements and Chapter 5 concludes the thesis.

1.1. Contribution

We extend the conceptional basis of the EWA splatting framework in a consistent way to mathematically represent motion blurred images. We achieve this by replacing the 2-dimensional Gaussian kernels which continuously reconstruct the point-sampled surface by 3-dimensional Gaussian kernels which unify a spatial and temporal component. By use of these the scene can be reconstructed continuously in space as well as time. The derived result naturally fits into the EWA splatting algorithm such that the final image can be computed as a weighted sum of warped and band-limited kernels. Accordingly, we implement the derived solution by means of a 3-pass rendering algorithm with strong parallels to the original 2D splatting algorithm.

We discuss the introduced approximations and their effect on image quality and rendering performance. Furthermore, we compare the visual quality of these approaches with ground truth images generated by ray-tracing and investigate on rendering performance.

2

Related Work

In this chapter we briefly review the most essential work on EWA splatting and previous approaches to motion blur. For a more detailed discussion on point-based rendering topics the reader is referred to [Poi07].

2.1. EWA Splatting

This thesis is based on the concept of resampling filters as introduced by Heckbert in his groundbreaking work on texture mapping [Hec89]. Resampling filters unify a Gaussian reconstruction filter with a low-pass filter which leads to rendering algorithms with high-quality anisotropic antialiasing capabilities. Using an affine approximation of a general projective mapping, Heckbert derived a Gaussian resampling filter, known as the *elliptical weighted average* (EWA) filter, which can be computed efficiently. While Heckbert originally developed resampling filters in the context of texture mapping, the technique has been reformulated and applied to point rendering by Zwicker et al. [ZPvBG01] in the so-called EWA splatting technique. Here, a Gaussian filter kernel, also called a splat, is assigned to each point sample. Following Heckbert's EWA approach, each kernel is transformed to image space by an affine approximation of the projective mapping. The corresponding resampling filter is then computed by band-limiting the projected kernel by an image space filter. Finally, the resampling filters of all points are rasterized and blended. Using resampling filters for point rendering, most sampling artifacts such as holes or aliasing can be effectively avoided.

Building on this work, several point-based rendering approaches have been developed, the early ones being implemented in software and therefore putting a high load on the CPU. To reduce the computational complexity of EWA splatting, an approximation using look-up tables has been

presented by Botsch et al. [BWK02].

[BSK04] and its follow-up work [ZPvBG01] present methods to interpolate normals between the splat primitives and render high-quality Phong shaded images.

The paper [ZRB+04] implements perspectively exact EWA splatting and handles arbitrary elliptical reconstruction kernels, where in previous techniques the projection may be exact at the ellipse center but the shape is only an affine approximation. Their method is based on the formulation of the 2D projective mappings from local tangent frames to image space using homogeneous coordinates. In contrast to the previous affine approximations of the projective mapping, they choose the affine mapping such that it matches the projective mapping of a conic isocontour of the Gaussian kernels. Since they choose an isovalue of the conic to correspond with the cut-off value of the kernels, the shape of the truncated kernels is correct under projective mappings. This leads to more accurate reconstruction of surfaces in image space and avoids holes even under extreme perspective projections. Although their method computes the perspectively correct splat shape, the resampling kernel is still based on an affine approximation of the perspective projection. Furthermore, the authors describe a direct implementation of their algorithm on GPU. Instead of rendering each splat as a quad or a triangle as proposed before, they use OpenGL point primitives and fragment programs to rasterize the resampling filters. Hence, they avoid the overhead of sending several vertices to the graphics processor for each point. In the rasterization stage each pixel in the image plane is tested whether it is inside or outside the resampling filter. To minimize the number of pixels that are tested a tight axis aligned bounding box is computed.

Most of the above methods neglect the screen space filter of the EWA framework and restrict to the Gaussian reconstruction filter in object space. While this leads to sufficient anti-aliasing in magnified regions, it cannot prevent aliasing artifacts in minified areas. In contrast, the method of [ZRB+04] implements the full EWA splatting approach on the GPU.

Botsch et al. [BHZK05] show how to exploit the increased capabilities of latest graphics hardware at that time for GPU-based surface splatting, such that the trade-off between quality and efficiency is effectively minimized. The availability of multiple render targets with floating point precision and blending capabilities enabled the implementation of all computations required for high-quality surface splatting directly on the GPU. Hence, this paper does not introduce genuinely new concepts, but focuses on the efficient implementation of a hardware-accelerated deferred shading framework and a simple and effective approximations of the EWA prefilter to achieve fast and high-quality surface splatting. Using per-pixel Phong shading and a simple but effective approximation to the screen space filter, the approach presented in this paper provides results comparable to the original EWA splatting. The algorithm which we propose in this thesis is built upon this 3-pass algorithm.

2.2. Temporal Antialiasing

The only mathematically strict method for generating realistic motion blur is said to be temporal supersampling, that is, to render the scene multiple times and then composite the rendering results. However, rendering each image and averaging them can be a time-consuming affair.

Getting around this and still maintaining accuracy is difficult since for each object instance along the motion path both shading and visibility change. All existing techniques attempt to simulate the effect by making certain assumptions and approximations.

A well-argued discussion on this is provided in the work of Sung et al. [SPW02]. The paper presents the motion blur technique implemented by means of ray casting in the Maya Rendering system. The authors categorize previous work on motion blur based on the approaches to solving the rendering equation

$$i(w,t) = \sum_l \int_\Omega \int_T r(w,t) g_l(w,t) L_l(w,t) dt dw, \tag{2.1}$$

where w denotes the solid angle of the viewing ray and T is the exposure time. The sum iterates through all l objects in the scene to determine the visible object in the w solid angle direction at time t and computing the incoming luminance of the visible object $L_l(w,t)$. The $g_l(w,t)$ term describes the visibility function, where it is 1 only for the visible geometry and 0 otherwise. Ω denotes the total solid angle from the environment towards the pixel (x,y). The $r(w,t)$ term describes a reconstruction filter which may model the physical movement of the camera shutter closing during the exposure time. Figure 2.1 summarizes the different classifications of previous work on temporal antialiasing. See [SPW02] for further details. Apart from implementations which utilize Monte Carlo integration, other approaches approximate the equation by making assumptions about the visibility, g, and the luminance, L, terms.

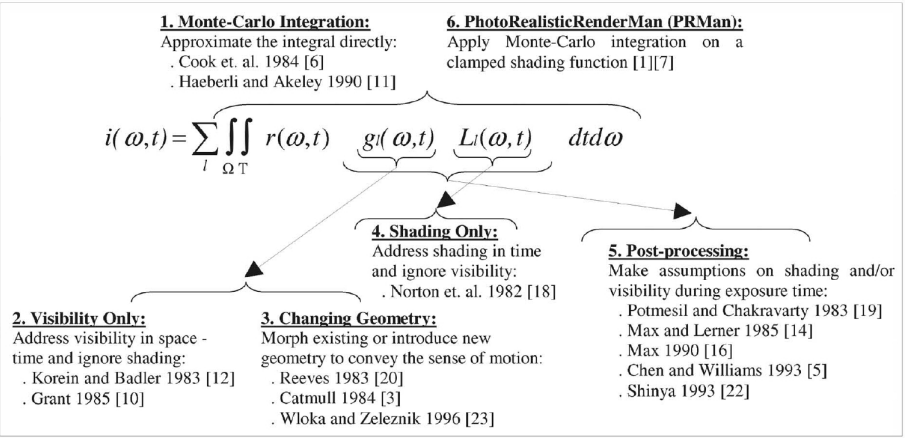

Figure 2.1.: *Classification of previous work on temporal antialiasing ([SPW02], Figure 1)*

With the Distributed Raytracing method by Cook et al. [CPC84], Monte Carlo integration has been proposed as an approximative solution to the image generation process. Because of the generality of the underlying Monte Carlo approach, this is the only existing approach so far that is capable of approximating $L_l(w,t)$ in general. However, as in all Monte Carlo methods, a large number of samples is usually required to generate images without excessive noise. For this reason it is difficult to achieve real-time constant rendering rates. In this thesis, we will use a Monte Carlo raytracer to generate ground truth images (see Sections 3.6.4 and 4.5).

Assuming constant shading within the interval where an object is visible, [KB83] and [Gra85] solve for the visibility, that is, the geometry term. They may solve the temporal geometric

aliasing problem for a spatial sample point, but do not address the temporal shading. As pointed out by the authors of both articles, in general, spatial-temporal antialiasing problems (for both geometric and shading) cannot be solved separately. That is, it is not possible to separate $g_l(w,t)$ into $g_l(w)g_l(t)$ or $L_l(w,t)$ into $L_l(w)gL_l(t)$ and solve the visibility or shading problem in the spatial and temporal domains separately. The authors of [SPW02] introduce a simplification by separating the shading problem from the visibility problem which still generates high quality images. However, since it requires adaptive super sampling, the rendering speed cannot be guaranteed.

On the other hand, geometric morphing methods deform or introduce new geometry, such as motion volumes, strategically placed with respect to the actual movements. All these approaches approximate motion effects by transforming $g_l(w,t)L_l(w,t)$ to $g_{l'}(w,t)L'_{l'}(w,t)$. Since the visible geometry and/or the luminance function must be altered, it is difficult for these approaches to adequately address issues involved in complicated motions and/or shadings. The volumetric kernels solution presented in this thesis (see Sections 3.6.3 and 4.1.2) falls under the geometric morphing category since we extend the 2-dimensional Gaussian kernels to volumetric kernels and additionally place these along the motion trajectory of a point sample.

[NRS82] addresses the temporal shading problem by proposing an image space method that interpolates between samples and provides a continuous transition from a sampled signal to another signal. Their method can be naturally extended for motion blur generation, however, the visibility is assumed to be constant. Our time bucketing method (see Sections 3.6.2 and 4.4) is tangent to this category since it makes simplifying assumptions on visibility while solving for the shading term.

Another field is constituted of post-processing techniques which operate on the synthesized images to simulate a motion blur effect and can be applied to any geometric representation. [PC83] proposes an approach for producing motion blur images by time convolution of the normal image with the motion function. However, in general, solutions to post-processing approaches only address the 2D problem and therefore cannot adapt to local properties of the images, and cannot address the situation where motion objects cannot be separated into non-overlapping layers in depth. [ML85] and [Shi93] extend the problem to 2.5D.

Sung et al. [SPW02] build their work upon a simplified alternative formulation of the rendering equation based on describing energy arriving at the screen image plane. The presented solution is to integrate the object-time visible area, that is, the projection of the subpixel region where an object is visible onto the moving object. They do this by means of the corresponding image-time visible volume, which is the per-object visible volume in the image-time domain. Computing spatial-temporal shading is solving the integration of the shading function over the object-time visible areas. For each pixel, the results of a point sample in space are stored as a list of continuous temporal coverage grouped according to the corresponding objects. Samples in time are sampled stochastically. As such, this solution of the visibility function is analytic in the temporal domain and stochastic in the spatial domain.

In the domain of volume splatting, [MMI+98] presents an antialiasing extension to the basic splatting algorithm that mitigates the spatial aliasing for high-resolution volumes. It furthermore outlines an analysis of the common approximation errors at that time in the splatting process for perspective viewing and, what is of interest in this context, discusses current work on extensions for temporal antialiasing. Specifically, it describes a framework for motion blur generation in

the context of splatting-based volume rendering. For volume rendering, the geometric visibility problem persists when semitransparent or opaque compositing is used. The method of Mueller et al. [MMI+98] simplified the problem and did not address the visibility problem, but only the integrated energy across the time domain. The elongated Gaussian kernel is constructed as a rectangle that spans the motion vector of a voxel, with two half-spherical Gaussian splats at both ends. This idea is similar to [WZ96] by creating a motion volume for each geometric elements.

The work by Mueller et al. has been extended by [GM04]. Since the method proposed in this paper is tailored to point-based surface rendering it is situated closest to our work. In contrast to [MMI+98], they only render a motion blurred isosurface of the volumetric object instead of all object voxels and use point-based surface rendering instead of full volume splatting. Rather than trying to create theoretically correct motion blur effects, their aim is to quickly provide strong motion hints for the viewer. The method tries to mimic an effect in conventional photography art: First the picture is shot with dim light while the shutter is kept open for a long enough time to capture the motion trail. Then the scene is given a strong flash for a very short time interval to capture the geometric and material information of the object as if the object stays still in that interval.

The approach accomplishes these effects by sending two primitives down the rendering pipeline: One for a sharp, realistic rendering of the surface object and the other one to simulate the motion trail. They use a simplified version of EWA surface splatting as described in [ZPvBG02]: The surface is composed of 2D round Gaussian splats and rendered as texture-mapped squares in space. The corresponding motion volume is generated by low-pass filtering the object along the time dimension. This makes some assumptions on the linearity of the motion path per primitive. I.e., a Gaussian filter is applied to a splat for temporal blurring which produces a Gaussian ellipse. The center is placed slightly behind to achieve the motion hint. Additionally, circular splats are rendered at the start and end times of the frame. The screen space ellipse is obtained by a projection transform. An alternative and more efficient method skips the 3D construction of the motion ellipsoid entirely and instead constructs the 2D motion ellipse from a convolution of the 2D Gaussian functions of the projected point and the projected motion vector. Conceptually this would be equivalent to a post-processing method.

A downside of the approach is, that it does not handle the visibility function of the moving objects properly because the point samples are first bucket-sorted with respect to their depth and the corresponding ellipses are rendered in back-to-front order. simply rendered by depth-sorting. Also, shading changes over time are not regarded. Currently it only handles small rotations, where motion vectors can be approximated by a straight line. This approach is similar to our volumetric kernels solution (see Sections 3.6.3 and 4.1.2) in a sense, that the 2-dimensional surface reconstruction kernels are extended by a temporal dimension. The benefit of our method is that we position several interpolating temporal supersamples along the trajectory of a point sample and compute the motion blurred image as a composite of these. This means that the rendered scene is continuously reconstructed in temporal dimension in a piece-wise linear way. The method thus can handle changing shading and visibility over time and more complex motions and, therefore, enables a higher photo-realsim. Also, our proposed method leads to a straight-forward adaption of the three-pass algorithm using GL point primitives which is commonly applied with EWA splatting (see Section 4.1.1). Motion hints can be achieved in our case by choosing an appropriate temporal weighting function (see Section 3.1) which gives more weight to later time instants.

2.3. Commercial Software

In this section, we give some examples of commercial software which is able to create motion blur effects and shortly describe their applied techniques.

The free open source 3D content creation suite Blender generates a motion blurred image by accumulating multiple intermediate frames rendered inbetween the real frames at equidistant time steps. In presence of fast motions, a high number of temporal samples is necessary to achieve visually pleasing results.

ReelSmart Motion Blur is a plug-in for video editing software such as Adobe After Effects. It works on image data and therefore implements motion blur by post-processing. A motion field is computed by tracking the pixel motions between two frames and used for convolving the respective frames.

The rendering software Mental Ray is the rendering component of many leading 3D content-creation tools such as Maya. Its raytracer is able to create fuzzy effects like depth-of-field or motion blur based on the concept of Distributed Raytracing [CPC84].

Pixar's RenderMan first clamps the shading function frequency by interpolating between a discrete set of temporal samples for each object. Then visibility is computed by stochastically taking samples in space and time. Positions within a single frame are piecewise linearly interpolated.

3

Core Theory

In this chapter, we derive the theoretical basis to our motion blur algorithms.

3.1. Screen space intensity of a motion blurred image

We interpret an image as a 2D signal in screen space. For an instantaneous image at time t, the intensity at screen space position $\mathbf{x}$ is given by the continuous screen-time space signal $g_c(\mathbf{x}, t)$. A motion blurred image which captures the scene over the period of exposure $[t_0, t_1]$ is represented by $G_{c,t_0,t_1}(\mathbf{x})$. The intensity value at position $\mathbf{x}$ is generated by a weighted integration of incoming intensities over the exposure time:

$$G_{c,t_0,t_1}(\mathbf{x}) = \int_{t_0}^{t_1} a(\mathbf{x}, t) g_c(\mathbf{x}, t) dt, \tag{3.1}$$

where the weighting function a is used to simulate the medium which captures the image (Figure 3.1). For simulating the behavior of a camera, the weights are only time-dependent. We restrict our theory to this case.

$$a(\mathbf{x}, t) := a(t). \tag{3.2}$$

We model the rendering process of $G_{c,t_0,t_1}(\mathbf{x})$ as a resampling problem of $g_c(\mathbf{x}, t)$ in time and space. If the sampling rate does not meet the Nyquist criterion of $g_c(\mathbf{x}, t)$, spatial and temporal aliasing artifacts occur. Two approaches to reduce aliasing problems exist: We can either sample the continuous signal at a higher frequency, or we eliminate frequencies above the Nyquist limit

Quantity	*Notation*
t	Time
$\mathbf{x}$	Screen space
$\mathbf{u}$	Source space: Local object surface parametrization
$\{P_k(t)\}$	Set of surface point samples with index k
$r_k(\mathbf{u}, t)$	Reconstruction kernel
$w_k(t)$	(Normalizing) weight, surface attribute sample
$\mathbf{m}(\mathbf{u}, t)$	Projective mapping from source to screen space: $\mathbb{R}^2 \times T \rightarrow \mathbb{R}^2$
$a(\mathbf{x}, t)$	Intensity contribution of screen signal at time instant t and screen position $\mathbf{x}$ to the motion blurred image
$g_c(\mathbf{x}, t)$	The continuous screen-time space signal
$h(\mathbf{x}, t)$	The band limiting prefilter
$g'_c(\mathbf{x}, t)$	The band limited continuous screen-time space signal
$g(\mathbf{x}, t)$	The band limited screen-time space signal discretized in time and screen space
$G_{c,t_0,t_1}(\mathbf{x})$	The motion blurred screen signal over exposure time $[t_0, t_1]$
$[t_0, t_1]$	Exposure time
$v_k(\mathbf{x}, t)$	Visibility of a surface splat in screen-time space
$f_c(\mathbf{u}, t)$	The continuous surface attribute function

Table 3.1.: *Notation.*

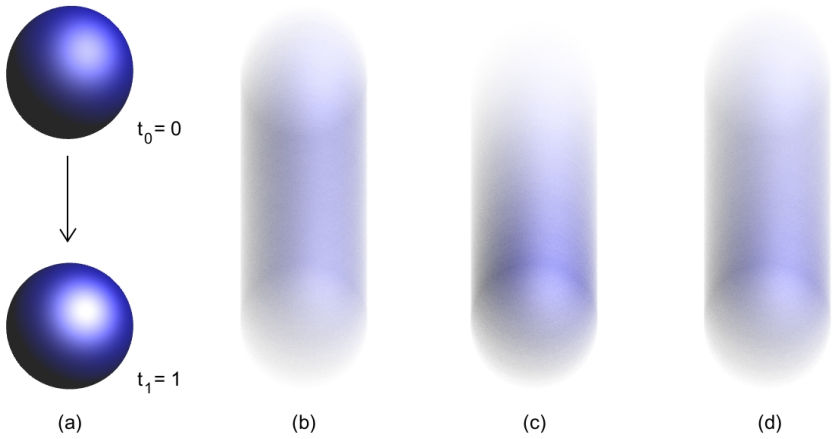

Figure 3.1.: *(a) A blue sphere moves in downward direction with constant velocity during the captured time interval $[t_0 = 0, t_1 = 1]$. This picture displays the instantaneous screen space signals $g_c(\mathbf{x}, t_0)$ and $g_c(\mathbf{x}, t_1)$, respectively. (b) The motion blurred image $G_{c,t_0,t_1}(\mathbf{x})$ with constant weighting $a(\mathbf{x}, t)$. (c) Here, the temporal weight decreases linearly with increasing distance of the sampling time t to the weighting center $t_c := 1$. (d) In this case, the weighting function is chosen to decrease exponentially, that is, $a(t) := 1.0 - exp(-|t_c - t|)$. The images are generated with raytracing. Per pixel 4 spatial and 100 temporal samples were taken.*

before sampling, which is called prefiltering or equivalently band limiting. The latter is realized by convolving the signal with a spatio-temporal antialiasing filter $h(\mathbf{x}, t)$. This results in the band limited continuous signal $g'_c(\mathbf{x}, t)$:

$$g'_c(\mathbf{x}, t) = g_c(\mathbf{x}, t) \otimes h(\mathbf{x}, t) \tag{3.3}$$

$$= \int_T \int_{\mathbb{R}^2} g_c(\xi, \tau) h(\mathbf{x} - \xi, t - \tau) d\xi d\tau. \tag{3.4}$$

Then we convert the continuous signal to a discrete signal by evaluating $g'_c(\mathbf{x}, t)$ at discrete frames t_i and pixel positions $\mathbf{x}_j$. Analytically this is expressed by multiplying the signal with a spatio-temporal pulse field $i(\mathbf{x}, t)$ (Figure 3.2):

$$g(\mathbf{x}, t) = g'_c(\mathbf{x}, t) \cdot i(\mathbf{x}, t). \tag{3.5}$$

3.2. Computation of the continuous screen-time space signal

We require that the scene objects can be represented by a continuous surface function $f_c(\mathbf{u}, t)$. This multidimensional function is defined on a local surface parametrization - the source space - and describes object attributes such as surface textures or normals at time t. The continuous

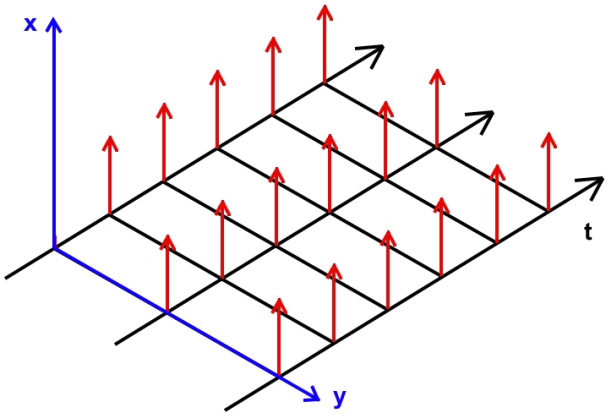

Figure 3.2.: *The sampling pulse field* $i(\mathbf{x}, t) = \sum_{i,j} \delta(\boldsymbol{x} - i, t - j)$

screen-time space signal $g_c(\mathbf{x}, t)$ is then retrieved by perspective projection of $f_c(\mathbf{u}, t)$ from source to screen space.

The projective mapping depends on the possibly time-variant camera parameters and is denoted by

$$\mathbf{m}_t(\mathbf{u}) : \mathbb{R}^2 \rightarrow \mathbb{R}^2. \tag{3.6}$$

Note that this mapping is invertible for a fixed time instant t since it is defined over the 2D domain of the local surface parametrization and not over 3D object space. This means by definition that the inverted mapping at time t assigns to each screen position $\mathbf{x}$ the visible surface point $\mathbf{u}$.

Using this, the continuous screen-time space signal can be formulated as follows:

$$\begin{aligned} g_c(\mathbf{x}, t) &= (f_c \circ \mathbf{m}^{-1})(\mathbf{x}, t) && (3.7) \\ &= f_c(\mathbf{m}^{-1}(\mathbf{x}, t), t) && (3.8) \end{aligned}$$

3.3. Representation of $f_c(\mathbf{u}, t)$

In our framework, animated graphics models are represented as a set $\{P_k(t)\}$ of time-dependent irregularly spaced point samples in three dimensional object space without connectivity. A point $P_k(t)$ has a certain position $\mathbf{u}_k(t)$ at time t. It is associated with a reconstruction kernel $r_k(\mathbf{u} - \mathbf{u}_k(t), t)$ centered at point position $\mathbf{u}_k(t)$. Each point represents a scalar sample $w_k(t)$ of the continuous surface attribute function $f_c(\mathbf{u}, t)$. The kernels are chosen to be elliptical 2D Gaussians. They interpolate the attribute samples between the points. Generally, they do not guarantee a partition of unity. Therefore, an interpolated value has to be divided by the sum of weights of the contributing kernels. In practice, the Gaussians are truncated to a finite support, i.e., the reconstruction kernels are evaluated only within conic isocontours at a specified cutoff value. (Figure 3.3).

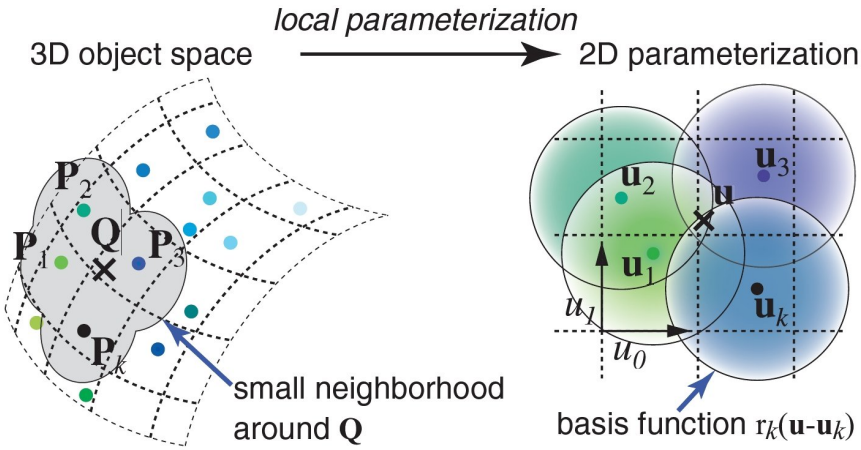

Figure 3.3.: *Defining a texture function on the surface of a point-based object ([ZPvBG01], Figure 2).*

Under the assumption that the objects are non-deformable, that is, relative point locations do not change over time, and surface attributes are constant over time, the weights and kernel functions become time-independent. In the following, these assumptions are taken:

$$r_k(\mathbf{u} - \mathbf{u}_k(t), t) := r_k(\mathbf{u} - \mathbf{u}_k(t)) \tag{3.9}$$

and

$$w_k(t) := w_k. \tag{3.10}$$

Then, the continuous surface attribute function $f_c(\mathbf{u}, t)$ at time t is reconstructed by the weighted sum

$$f_c(\mathbf{u}, t) = \sum_{k \in \mathbb{N}} w_k r_k(\mathbf{u} - \mathbf{u}_k(t)). \tag{3.11}$$

We define the projective mapping $\mathbf{m}(\mathbf{u}, t)$ individually for each reconstruction kernel as $\mathbf{m}_k(\mathbf{u}, t)$.

3.4. Geometric Interpretation

A reconstruction kernel $r_k(\mathbf{u} - \mathbf{u}_k(t))$ in form of an elliptical 2D Gaussian can be geometrically interpreted as a splat primitive centered at point sample position $\mathbf{u}_k(t)$ and with normal orientation. We are now considering the geometric behavior of a single splat primitive over the exposure time $[t_0, t_1]$. In object space, the volume which is defined by sweeping the silhouette of the splat along the motion trajectory of $\mathbf{u}_k(t)$ has a 3D tubular shape. At the ends, the tube is closed up by the splats centered at $\mathbf{u}_k(t_0)$ and $\mathbf{u}_k(t_1)$ respectively. The resulting shape of a perspective projection of this volume to screen space is a planar tube (Figure 3.4).

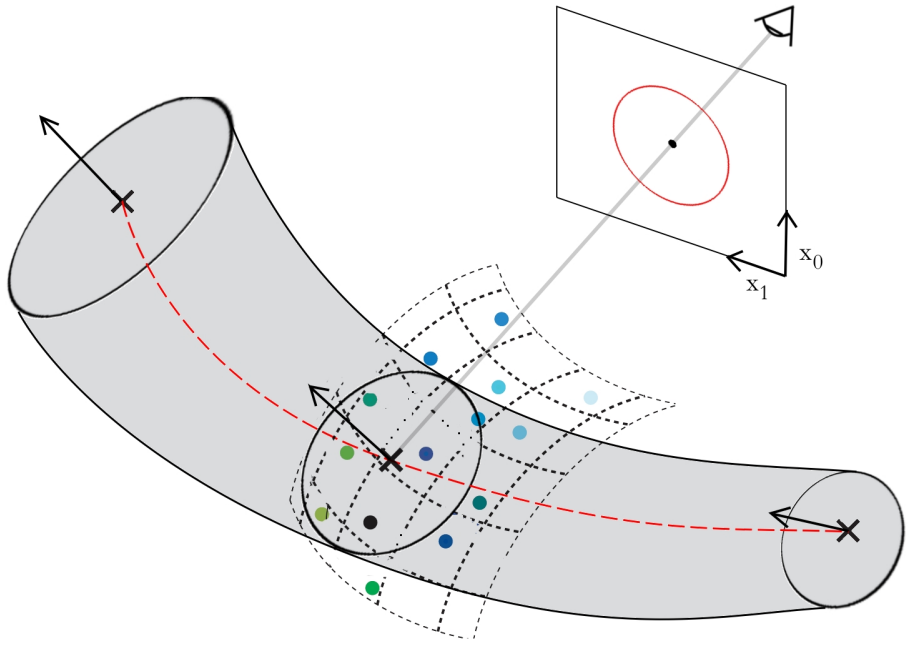

Figure 3.4.: *The time-space volume of a single surface kernel and one time slice projected to the screen.*

3.5. Extended EWA surface splatting

For each surface point $\mathbf{u}(t)$ a visibility value can be defined which denotes whether the point is fully visible from the camera viewpoint or occluded by other surface parts. We formalize this by introducing per point sample visibility functions $v_k(\mathbf{x}, t)$ over the screen-time space domain. For each $\mathbf{u}(t) = \mathbf{m}_k^{-1}(\mathbf{x}, t)$ in the local parametrization plane of point sample $P_k(t)$ they define the visibility at time t. Since we do not consider transparency, the the visibility is binary, that is, either fully opaque or completely occluded. Using this and inserting Equation (3.11) into Equation (3.8) we arrive at the following equation for the continuous screen-time space signal $g_c(\mathbf{x}, t)$:

$$
\begin{aligned}
g_c(\mathbf{x}, t) &= (f_c \circ \mathbf{m}^{-1})(\mathbf{x}, t) && (3.12)\\
&= f_c(\mathbf{m}^{-1}(\mathbf{x}, t), t) && (3.13)\\
&= \sum_{k \in \mathbb{N}} w_k v_k(\mathbf{x}, t) r_k(\mathbf{m}_k^{-1}(\mathbf{x}, t) - \mathbf{u}_k(t)) && (3.14)\\
&= \sum_{k \in \mathbb{N}} w_k v_k(\mathbf{x}, t) r'_k(\mathbf{x}, t), && (3.15)
\end{aligned}
$$

where $r'_k(\mathbf{x}, t) = r_k(\mathbf{m}_k^{-1}(\mathbf{x}, t) - \mathbf{u}_k(t))$ are the reconstruction kernels projected to screen space.

An explicit expression for the band limited screen-time space signal of Equation (3.4) can then be derived as follows:

$$
\begin{aligned}
g'_c(\mathbf{x},t) &= g_c(\mathbf{x},t) \otimes h(\mathbf{x},t) && (3.16)\\
&= \int_T \int_{\mathbb{R}^2} \sum_{k\in\mathbb{N}} w_k v_k(\xi,\tau) r_k(\mathbf{m}_k^{-1}(\xi,\tau) - \mathbf{u}_k(\tau)) \\
&\quad h(\mathbf{x}-\xi, t-\tau) d\xi d\tau && (3.17)\\
&= \sum_{k\in\mathbb{N}} w_k \int_T \int_{\mathbb{R}^2} v_k(\xi,\tau) r_k(\mathbf{m}_k^{-1}(\xi,\tau) - \mathbf{u}_k(\tau)) \\
&\quad h(\mathbf{x}-\xi, t-\tau) d\xi d\tau. && (3.18)
\end{aligned}
$$

We approximate this equation by pulling the visibility function and kernels apart and filtering them separately. This extremely simplifies computation since these two very different types of functions can now be solved separately, probably in different ways. It can be exploited that the attribute function is usually arbitrary while the visibility function is well-defined by the geometry and associated motion.

$$
\begin{aligned}
g'_c(\mathbf{x},t) &\approx \sum_{k\in\mathbb{N}} w_k \\
&\quad \int_T \int_{\mathbb{R}^2} v_k(\xi,\tau) h(\mathbf{x}-\xi, t-\tau) d\xi d\tau \\
&\quad \int_T \int_{\mathbb{R}^2} r_k(\mathbf{m}_k^{-1}(\xi,\tau) - \mathbf{u}_k(\tau)) h(\mathbf{x}-\xi, t-\tau) d\xi d\tau && (3.19)\\
&= \sum_{k\in\mathbb{N}} w_k v'_k(\mathbf{x},t) \rho_k(\mathbf{x},t) && (3.20)
\end{aligned}
$$

with the filtered visibility functions $v'_k(\mathbf{x},t)$ and ideal resampling kernels $\rho_k(\mathbf{x},t)$.

$$
v'_k(\mathbf{x},t) = \int_T \int_{\mathbb{R}^2} v_k(\xi,\tau) h(\mathbf{x}-\xi, t-\tau) d\xi d\tau, \tag{3.21}
$$

$$
\rho_k(\mathbf{x},t) = \int_T \int_{\mathbb{R}^2} r_k(\mathbf{m}_k^{-1}(\xi,\tau) - \mathbf{u}_k(\tau)) h(\mathbf{x}-\xi, t-\tau) d\xi d\tau. \tag{3.22}
$$

The above equations state that we can first project and filter each reconstruction kernel $r_k(\mathbf{u},t)$ individually to derive the resampling kernels $\rho_k(\mathbf{x},t)$ and then sum up the contributions of these kernels in screen space where occluded parts are masked out by the filtered visibility function $v'_k(\mathbf{x},t)$.

These derivations exactly match the original EWA splatting framework for an instantaneous image at time t with the difference that visibility is formalized here and filtering band limits the signal over time as well as space.

3.6. Computation of the motion blurred screen signal

Combining Equation (3.20) with Equation (3.1) results in:

$$G_{c,t_0,t_1}(\mathbf{x}) \approx \int_{t_0}^{t_1} a(t) \sum_{k \in \mathbb{N}} w_k v'_k(\mathbf{x}, t) \rho_k(\mathbf{x}, t) dt. \tag{3.23}$$

This section presents different solution approaches to approximate this equation.

3.6.1. Temporal sampling of $g'_c(\mathbf{x}, t)$

The time integral of Equation (3.23) is solved by Monte Carlo integration:

$$G_{c,t_0,t_1}(\mathbf{x}) \approx \frac{1}{t_1 - t_0} \frac{1}{N} \sum_j a(t_j) g'_c(\mathbf{x}, t_j) \tag{3.24}$$

where the time samples t_j are chosen stochastically. If the temporal sampling is not sufficiently dense, temporal artifacts appear. E.g., in presence of high velocity motion, the image appears to be jagged. In other words, they will appear to be multiple exposures which is the case if the same time samples t_i apply to all pixels.

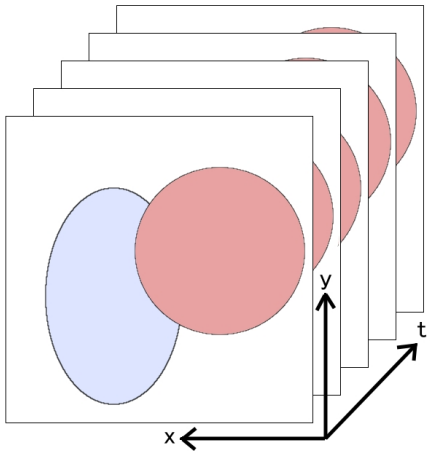

Figure 3.5.: *Sampling the screen-time volume in the temporal dimension.*

In other words, this approach is equivalent to averaging temporal samples of the filtered continuous screen-time function $g'_c(\mathbf{x}, t)$ weighted by $a(t)$ (Figure 3.5).

3.6.2. Temporal sampling of $P_k(t)$

We insert Equation (3.20) into Equation (3.24) and apply the following reorganization:

$$G_{c,t_0,t_1}(\mathbf{x}) \approx \frac{1}{N_j}\sum_j a(t_j)\sum_{k\in\mathbb{N}} w_k v'_k(\mathbf{x},t)\rho_k(\mathbf{x},t) \tag{3.25}$$

$$\approx \sum_{k\in\mathbb{N}}\frac{1}{N_{j_k}}\sum_{j_k} a(t_{j_k})w_k v'_k(\mathbf{x},t_{j_k})\rho_k(\mathbf{x},t_{j_k}) \tag{3.26}$$

$$\tag{3.27}$$

This means that the Monte Carlo integration is performed over the individual point sample trajectories $P_k(t)$ (Figure 3.4). The time samples t_{j_k} differ and can be chosen adaptively.

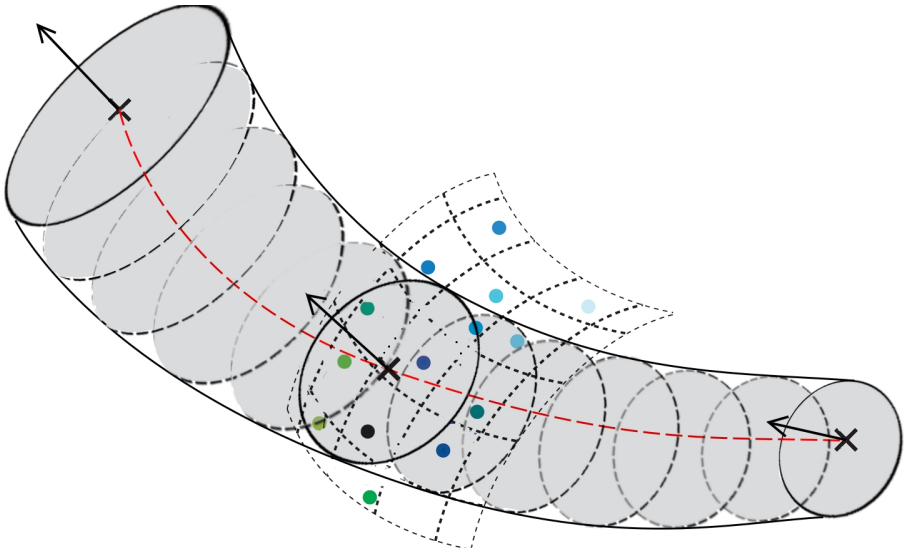

Figure 3.6.: *Approximation of the space-time volume by sampling along the motion trajectory.*

While the filtered resampling kernels $\rho_k(\mathbf{x},t_{j_k})$ are well-defined, we have to provide a computation method for the visibility function $v'_k(\mathbf{x},t_{j_k})$.

A very crude simplification is to evaluate visibility once in the exposure time $[t_0,t_1]$ against all generated time samples:

$$v'_k(\mathbf{x},t_{j_k}) = 0 \text{ if } \mathbf{m}_k^{-1}(\mathbf{x},t_k) \text{ is occluded by a 2D surface splat associated with any } P_k(t_{j_k}), \text{ else } 1. \tag{3.28}$$

The temporal artifacts which arise due to this simplification are illustrated in Figure 3.7.

To alleviate this problem, we suggest to quantify visibility in time. That is, we subdivide $[t_0,t_1]$ into smaller subintervals $[t_i,t_{i+1}]$ where $v'_k(\mathbf{x},t_{j_k})$ is evaluated separately:

$$v'_k(\mathbf{x},t_{j_k}) = 0 \text{ if } \mathbf{m}_k^{-1}(\mathbf{x},t_k) \text{ is occluded at any time within } [t_i,t_{i+1}], \text{ else } 1. \tag{3.29}$$

As the subinterval sizes become infinitely small, $v'_k(\mathbf{x},t_{j_k})$ approaches the exact visibility function.

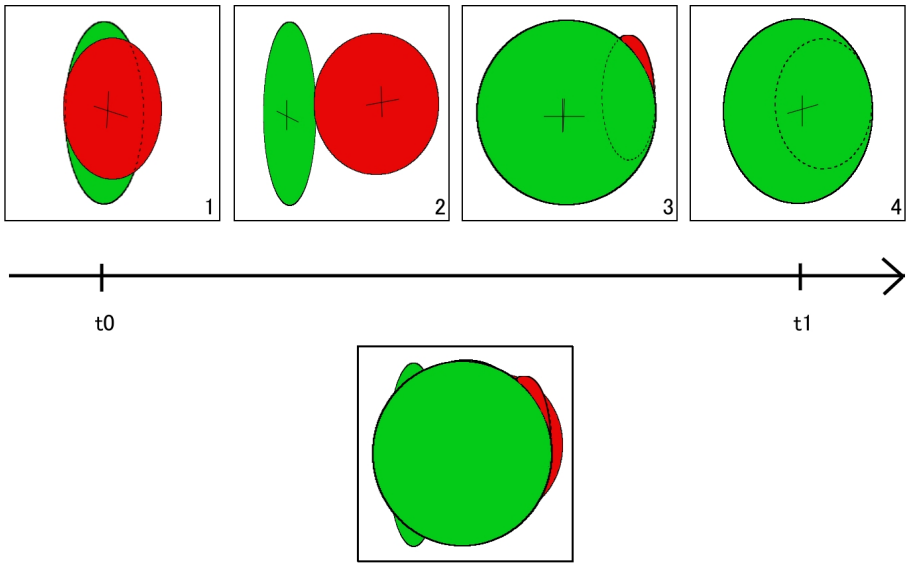

Figure 3.7.: *Top: Within the time span $[t_0, t_1]$, four temporal samples are taken of the green and red surface splats respectively and are treated as new, separate splats. Bottom: Assume the green splat at time step 4 is closest to the eye compared to all other splats. Splatting all samples using the same z-Buffer then causes all other splats to be occluded. However, splats from the other time steps 1, 2 and 4 should not be occluded, but blended. A correct solution would be to use a separate z-Buffer for each time step and finally blend the results.*

3.6.3. Continuous Temporal Reconstruction

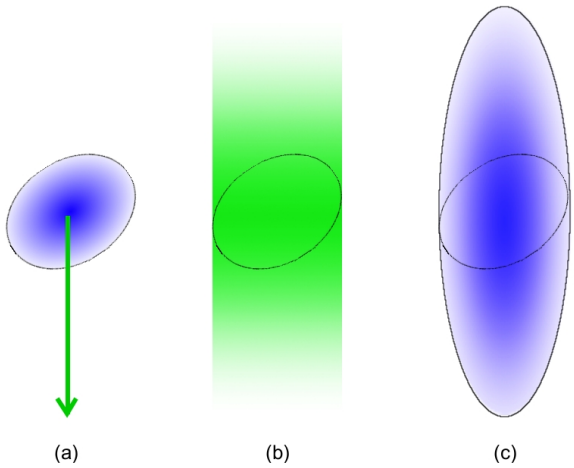

Figure 3.8.: *Construction of a volumetric kernel. (a) The 2D surface splat is moving downwards along the instantaneous velocity vector which is indicated by the green arrow. The blue gradient the splat is filled with represents the underlying elliptical Gaussian function. (b) Each point on the 2D surface splat is convolved with a 1D Gaussian along the velocity vector which is illustrated by the green gradient. (c) The resulting volumetric kernel.*

In analogy to a spatial reconstruction of $f_c(\mathbf{u}, t)$ by centering kernels at point samples, we sample the motion trajectories of $P_k(t)$ in time. Ellipsoidal 3D reconstruction kernels $R_{kt}(\mathbf{u})$ with time dimensionality are centered at the point sample positions $\mathbf{u}_{kt}$. The new kernels $R_{kt}(\mathbf{u})$

are constructed based on the elliptical 2D Gaussian kernels $r_k(\mathbf{x}, t)$ by convolution with a 1D Gaussian along the velocity vector at the time t (Figure 3.8. The variance of the 1D Gaussian is determined by the velocity magnitude and the sampling density. Since Gaussian kernels are closed under convolution, this results in an ellipsoidal 3D Gaussian (see Appendix C) which describes the linearized motion trail at time instant t. Thus, we assume the motion to be a low-passing of the object location over time. The surface attribute function is now continuously reconstructed as follows:

$$f_c(\mathbf{u}, t') = \sum_{k\in\mathbb{N}} \sum_{t_k} w_k R_{kt_k}(\mathbf{u} - \mathbf{u}_{kt_k}). \tag{3.30}$$

Inserting this into Equation (3.8) leads to the following equation for the continuous screen-time space signal $g_c(\mathbf{x}, t')$:

$$\begin{aligned}
g_c(\mathbf{x}, t') &= (f_c \circ \mathbf{m}^{-1})(\mathbf{x}, t') && (3.31)\\
&= f_c(\mathbf{m}^{-1}(\mathbf{x}, t'), t') && (3.32)\\
&= \sum_{k\in\mathbb{N}} \sum_{t_k} w_k v_{kt_k}(\mathbf{x}, t') R_{kt_k}(\mathbf{m}_k^{-1}(\mathbf{x}, t') - \mathbf{u}_{kt_k}) && (3.33)\\
&= \sum_{k\in\mathbb{N}} \sum_{t_k} w_k v_{kt_k}(\mathbf{x}, t') R'_{kt_k}(\mathbf{x}), && (3.34)
\end{aligned}$$

where $R'_{kt_k}(\mathbf{x}) = R_{kt_k}(\mathbf{m}_k^{-1}(\mathbf{x}, t') - \mathbf{u}_{kt_k})$ are the reconstruction kernels projected to screen space.

An explicit expression for the band limited screen-time space signal of Equation (3.4) can then be derived as follows:

$$\begin{aligned}
g'_c(\mathbf{x}, t') &= g_c(\mathbf{x}, t') \otimes h(\mathbf{x}, t') && (3.35)\\
&= \int_T \int_{\mathbb{R}^2} \sum_{k\in\mathbb{N}} \sum_{t_k} w_k v_{kt_k}(\xi, \tau) R_{kt_k}(\mathbf{m}_k^{-1}(\xi, \tau) - \mathbf{u}_{kt_k}) \\
&\quad h(\mathbf{x} - \xi, t' - \tau) d\xi d\tau && (3.36)\\
&= \sum_{k\in\mathbb{N}} \sum_{t_k} w_k \int_T \int_{\mathbb{R}^2} v_{kt_k}(\xi, \tau) R_{kt_k}(\mathbf{m}_k^{-1}(\xi, \tau) - \mathbf{u}_{kt_k}) \\
&\quad h(\mathbf{x} - \xi, t' - \tau) d\xi d\tau && (3.37)\\
&\approx \sum_{k\in\mathbb{N}} \sum_{t_k} w_k \\
&\quad \int_T \int_{\mathbb{R}^2} v_{kt_k}(\xi, \tau) h(\mathbf{x} - \xi, t' - \tau) d\xi d\tau \\
&\quad \int_T \int_{\mathbb{R}^2} R_{kt_k}(\mathbf{m}_k^{-1}(\xi, \tau) - \mathbf{u}_{kt_k}) h(\mathbf{x} - \xi, t' - \tau) d\xi d\tau && (3.38)\\
&= \sum_{k\in\mathbb{N}} \sum_{t_k} w_k v'_{kt_k}(\mathbf{x}, t') \rho_{kt_k}(\mathbf{x}). && (3.39)
\end{aligned}$$

with the filtered visibility functions $v'_{kt_k}(\mathbf{x}, t')$ and ideal resampling kernels $\rho_{kt_k}(\mathbf{x})$.

$$v'_{kt_k}(\mathbf{x}, t') = \int_T \int_{\mathbb{R}^2} v_{kt_k}(\xi, \tau) h(\mathbf{x} - \xi, t' - \tau) d\xi d\tau, \tag{3.40}$$

$$\rho_{kt_k}(\mathbf{x}) = \int_T \int_{\mathbb{R}^2} R_{kt_k}(\mathbf{m}_k^{-1}(\xi, \tau) - \mathbf{u}_{kt_k}) h(\mathbf{x} - \xi, t' - \tau) d\xi d\tau. \tag{3.41}$$

In analogy to the 2D case (Equations (3.28) and (3.29)), visibility is defined as a per point sample function $v_{kt_k}(\mathbf{x}, t')$.

Since Gaussians are closed under perspective projection and convolution with a Gaussian filter, the projected filtered resampling kernels $\rho_{kt_k}(\mathbf{x})$ represent elliptical 2D Gaussians.

With this approach, an approximation of Equation (3.23) is computed as follows:

$$G_{c,t_0,t_1}(\mathbf{x}) \approx \frac{1}{N_j} \sum_j a(t_j) g'_c(\mathbf{x}, t_j) \tag{3.42}$$

$$= \frac{1}{N_j} \sum_j a(t_j) \sum_{k \in \mathbb{N}} \sum_{t_k} w_k v'_{kt_k}(\mathbf{x}, t_j) \rho_{kt_k}(\mathbf{x}). \tag{3.43}$$

We now choose the temporal samples t_j to be equal to the times t_k where the motion trajectories were sampled, in other words, the kernel centers in time dimension. We then arrive at the following equation:

$$G_{c,t_0,t_1}(\mathbf{x}) \approx \sum_{k \in \mathbb{N}} \sum_{t_k \in [t_0, t_1]} a(t_k) w_k v'_{kt_k}(\mathbf{x}) \rho_{kt_k}(\mathbf{x}), \tag{3.44}$$

where $v'_{kt_k}(\mathbf{x}) = v'_{kt_k}(\mathbf{x}, t_k)$ is the visibility function at the time t_k where the kt_k-th point sample was taken.

We can approximate visibility by setting it constant per sample P_{kt_k}:

$$v'_{kt_k}(\mathbf{x}) := v'_{kt_k}. \tag{3.45}$$

In this case, the motion blurred image is simply a result of a weighted accumulation of the resampling kernels $\rho_{kt_k}(\mathbf{x})$ with $t_k \in [t_0, t_1]$.

To handle visibility more correctly, notice that the visibility $v'_{kt_k}(\mathbf{x})$ of a kernel is the result of integrating its visibility along the viewing ray through the pixel $\mathbf{x}$.

3.6.4. Reference Implementation

A general solution to the image generation problem is an approximation based on Monte Carlo integration. I.e., the integral of Equation (3.1) is directly approximated by

$$G_{c,t_0,t_1}(\mathbf{x}) \approx \frac{1}{t_1 - t_0} \cdot \frac{1}{N_j} \frac{1}{N_k} \sum_j \sum_k a(t_k) g_c(\mathbf{x}_j, t_k). \tag{3.46}$$

With this method visibility and shading is simultaneously approximated by point sampling.

4

Rendering Algorithms

This chapter will provide a detailed description of the algorithms underlying the temporal supersampling (Section 4.3), time bucketing (Section 4.4) and volumetric kernels (Section 4.2.2) approaches, starting with a description of the basic multipass 2D splatting algorithm. Figure 4.1 gives an overview of the terminology concerning coordinate systems and transforms.

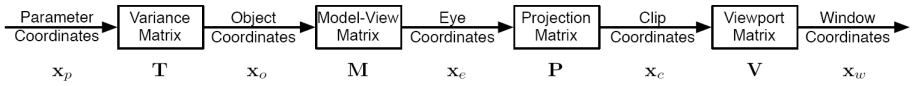

Figure 4.1.: *The OpenGL vertex transformation sequence is preceded by an additional transformation from parameter coordinates to object coordinates. In parameter coordinates, the conic matrix defining the quadratic surface is a diagonal matrix. ([SWBG06], Figure 3)*

4.1. Multipass Algorithm

This section describes the multipass algorithms for rendering 2D splats (Section 4.1.1) and the modified version for volumetric kernels (Section 4.1.2).

4.1.1. 2D splats

For rendering 2D splats we follow the three pass algorithm (Figure 4.2) which is commonly used for surface splatting and achieves correct blending of spatially overlapping kernels. An outline of each pass is given in the following. For a more detailed discussion refer to [BHZK05].

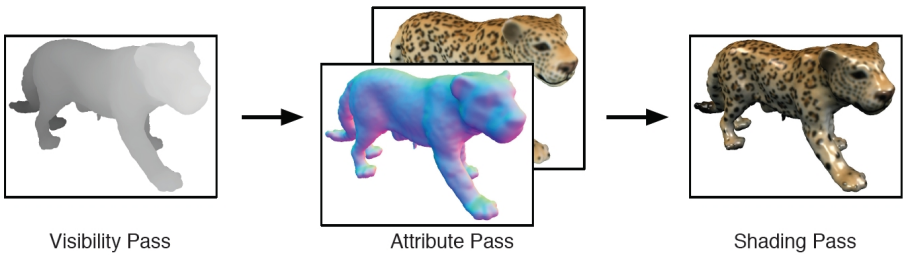

Figure 4.2.: *The deferred shading pipeline for GPU-based splatting. The visibility pass fills the z-buffer, such that the attribute pass can correctly accumulate surface attributes, like color values and normal vectors, in separate render targets. The final shading pass computes the actual color value for each image pixel based on the information stored in these render targets ([BHZK05], Figure 1).*

Visibility pass

In the first pass the object is rendered without lighting in order to fill the depth buffer only [RL00]. The object is slightly shifted away from the camera by a small ϵ value to achieve a Gouraud-like blending of overlapping splats whose depths differ less than ϵ during the next pass. Still, this leads to correct occlusions for splats with larger depth offsets.

Attribute pass

Having lighting and alpha blending options enabled, we use multiple render targets to splat and accumulate normal vectors as well as material properties during this pass. The corresponding pixel shader performs the rasterization computations outlined in Section 4.2, but instead of shading each accepted pixel, its (weighted) normal vector and color values are output to the two render targets. These buffers and the depth buffer are then used as textures for the final normalization and shading pass, for which a window-size rectangle is drawn in order to send each pixel through the rendering pipeline again [ZPvBG01].

Normalization and shading pass

In a final normalization pass each pixel is normalized by dividing the accumulated normal vector and material properties stored in its RGB components by the sum of weights stored in its alpha component. From the depth texture, the corresponding 3D position can easily be derived by inverting the viewing and projection mappings. Having position, normal and color information at hand then enables deferred per-pixel shading computations. The resulting Phong shading clearly improves the rendering quality over the Gouraud shading used by most previous methods. Since lighting computations are performed only once for each pixel of the projected object in the final image instead of for each object pixel, deferred shading also yields noticeable performance improvements.

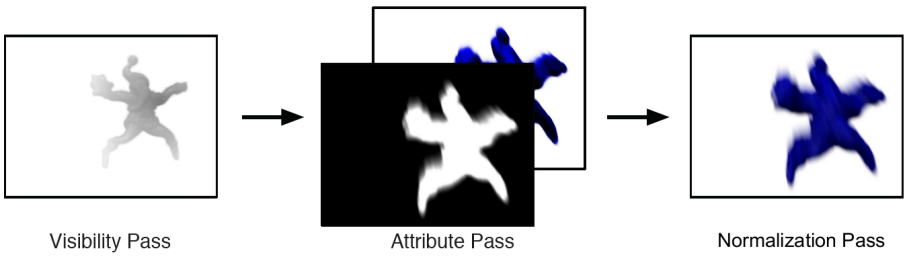

Figure 4.3.: *The shading pipeline for GPU-based rendering of volumetric kernels. The visibility pass fills the depth texture which is used as an input to the attribute pass, which computes the depth of a fragment by tracing back its source point. Surface attributes, like color values, and temporal alpha values can then be correctly accumulated in separate render targets. The final shading pass blends the color value at a pixel with the background based on the information stored in these render targets.*

4.1.2. Volumetric kernels

The volumetric kernels approach basically follows the three-pass rendering algorithm with modifications on depth testing (Figure 4.3). Also, deferred shading cannot be applied anymore. The reason for this is that the temporal samples interpolate the normals over the time interval which causes incorrect results (see Figure 4.4). Furrhtermore,the lighting would eventually only be evaluated once with deferred shading, that is, at the end of the interval. This means, that the specular highlights are not blurred. A correct solution has to evaluate lighting for each temporal sample separately.

Visibility pass

Notice that we can only perform occlusion testing for samples which have been taken at the same time instant. Otherwise, artifacts like in the time bucketing approach occur (see Section 4.4). Therefore, it is necessary to perform a separate visibility pass for each time instant where samples are generated. This can be achieved by using a layered depth buffer, however, since the number of layers is limited, samples within a certain time interval have to be collected in the same depth buffer layer. This causes the artifacts as observed in the time bucketing approach. In our implementation, we perform a loop over the sampled time instants and clear the depth buffer each time again. Also, depth testing has to be done based on the underlying 2D splats and not on the ellipsoids since these represent not a single time instant but a range in time. Furthermore, the modified attribute pass needs to access the depth buffer at arbitrary window coordinates as described in Section 4.1.2. Therefore, instead of writing depth values to the frame z-buffer, we add a depth texture which will be used as an input texture in the attribute pass. Apart from this, the visibility pass works as described before. The 2D splats which belong to the volumetric kernels are rendered with a slight offset for blending of spatially overlapping splats for small depth differences.

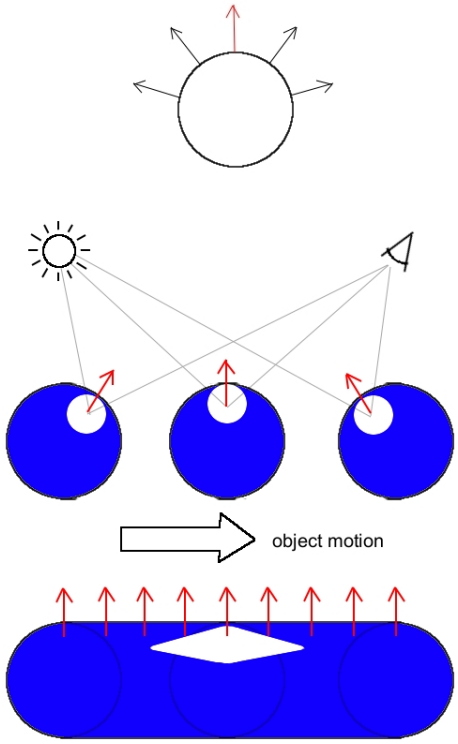

Figure 4.4.: *Problem with normal interpolation: Top: When the normals of temporal samples which are spatially located very closely are interpolated on the sphere surface, the surface normal of the swept volume points straight upwards. Middle: The light source is at the left and the eye at the right corner. The image displays three temporal samples with the correct specular reflection at the time instant. Bottom: In this case, the normals are interpolated between the temporal samples. As a result, the surface normals of the swept volume are perpendicular to the motion trail. This incorrectly causes the specular reflection to vanish at the beginning and end of the interval of exposure.*

Attribute pass

In the attribute pass material properties are rendered to the two render targets. The ellipsoidal kernels are rasterized as described in Section 4.2.2. The major difference to the previous algorithm is the way in which depth testing is performed which is illustrated in Figure 4.5. To compute the depth value of a fragment which has been accepted because it is within the projected ellipsoid boundary, we have to determine its source on the 2D surface splat. In our implementation, we use the midpoint of the viewing ray intersection points with the ellipsoid surface. Along the direction of the velocity vector this point is traced back onto the 2D surface splat. The resulting intersection point has depth value d and window coordinates (x_w, y_w). The depth value now has to be compared at the location (x_w, y_w) and not at the fragment location as in the fixed functionality pipeline. That is, a fragment passes the depth test if its source on the 2D splat is not occluded:

$$d \leq depth\_texture(x_w, y_w) \tag{4.1}$$

where $depth\_texture(x_w, y_w)$ denotes the value stored at location (x_w, y_w) in the depth texture

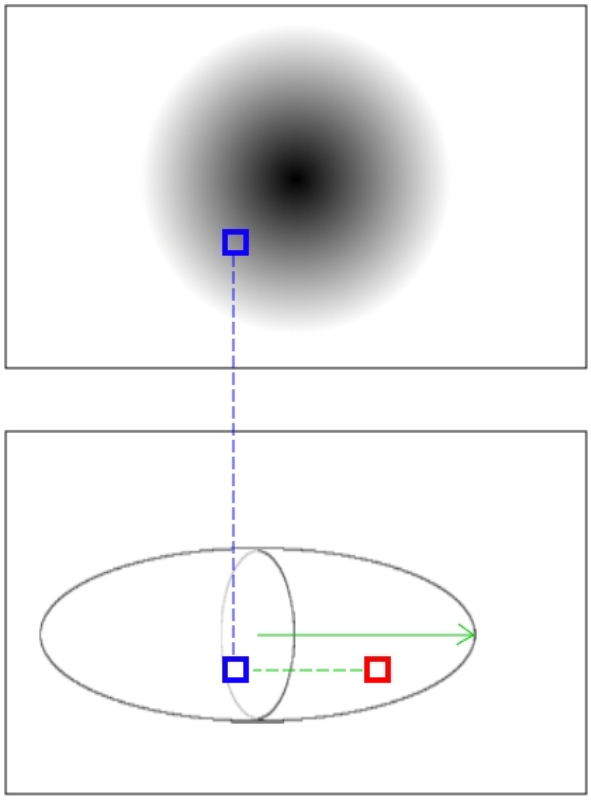

Figure 4.5.: *Depth test for volumetric kernels. The top rectangle displays the depth texture content after the visibility pass. The bottom rectangle shows an ellipsoid in window space during the fragment stage of the attribute pass. Within the ellipsoid, the original 2D splat is indicated. The green arrow signifies the velocity vector. To compute the depth of a fragment (red square) and perform depth testing, the source of the fragment is determined first. This is done by following the midpoint of the ray intersection points back along the velocity vector direction onto the 2D splat in eye space. The resulting point (blue square) determines the depth d of the fragment. This value is now compared with the depth texture content at the same pixel location (blue square).*

which has been filled in the visibility pass. Figures 4.7 and 4.8 illustrate exemplary results of this depth testing algorithm. Since ϵ depth testing is applied, a fragment is accepted if its source point depth is within the ϵ depth offset and, as a result, spatial blending is performed. In contrast to the depth texture, the material render targets are not cleared during the loop and samples of different time instants are blended in the same buffer. To be able to finally blend the image with the background, we write the blending weights to a further target texture which we call alpha texture. In contrast to other material properties, these alpha values are not computed as a weighted average, but as a weighted sum (see Figure 4.6). For 2D splatting, phong shading calculations were done in the final normalization pass. Here however, a corresponding 3D location of a pixel does not exist anymore, this is why we have to abandon deferred shading and move all shading computations to the attribute pass. The color and alpha buffers are then used as textures for the final normalization pass, as before.

The blending weight is a combination of spatial blending due to the continuous surface reconstruction (ϵ depth testing) and temporal blending due to a continuous reconstruction over time.

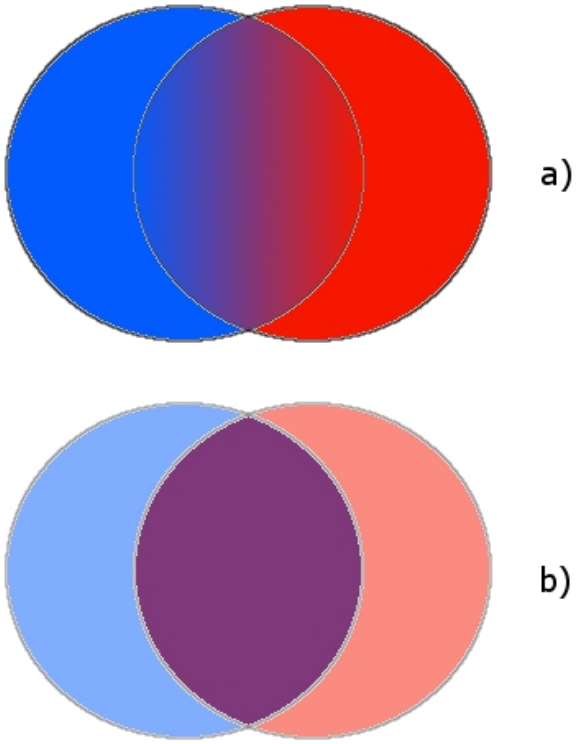

Figure 4.6.: *This picture displays the different behavior of spatial and temporal blending. a) The two surface splats are sampled at the same time instant. They are overlapping with a distance which is within the epsilon offset. Therefore, the color in the overlapping area is a weighted average of red and blue. b) The surface splats have been sampled at two different time instants. Even though they are spatially closely overlapping, there is no spatial blending due to the time difference. However, there is a temporal blending which is handled differently for material property and transparency. The color value is averaged with equal weight. The transparency, however, is summed up.*

The latter has to ensure that the intensity integral of a kernel is constant independent of the length in time which the kernel represents. We calculate the blending weight of a fragment by transforming the intersection midpoint to parameter space and insert it into a 3D Gaussian function.

Normalization pass

In a final normalization pass each pixel is normalized by dividing the accumulated material properties stored in its RGB components by the sum of weights stored in its alpha component. The values in the alpha buffer are not normalized, but clamped to the range $[0, 1]$. Finally, the resulting image is blended with the background by use of these alpha values.

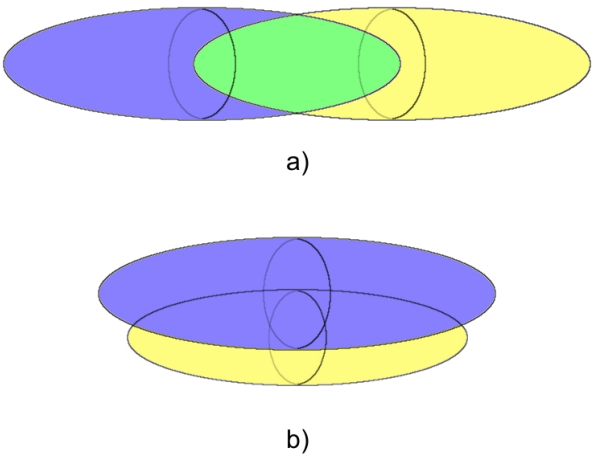

Figure 4.7.: *This picture illustrates visibility for volumetric kernels for two point samples which are sampled at the same time instant t. The underlying surface splats are outlined in the center of each volumetric kernel. The motion is along the x-Axis direction, where it does not matter whether it is in negative or positive direction. a) Here, the two surface splats are completely visible. This means, that the visibility $v'_{kt}(\mathbf{x})$ for both volumetric kernels is constantly 1 and the overlapping parts are blended according to their weights w_k. b) This case shows a surface splat (yellow) in the back which is partly occluded by a surface splat (blue). As a result, the front volumetric kernel will occlude the one located in the back.*

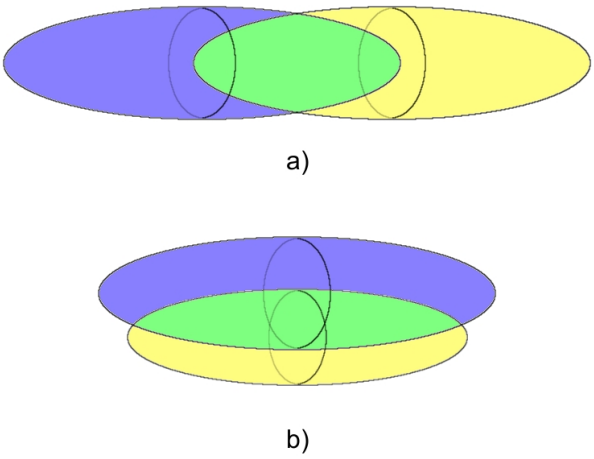

Figure 4.8.: *This picture illustrates visibility for volumetric kernels for two point samples which are sampled at different times t_0 (blue) and t_1 (yellow). The underlying surface splats are outlined in the center of each volumetric kernel. The motion is along the x-Axis direction, where it does not matter whether it is in negative or positive direction. Since occlusion cannot be tested against samples of different time instants, both volumetric kernels are completely visible for both cases a) and b) and are blended according to their weights w_k.*

4.2. Kernel rasterization

In this section we shortly describe the perspectively correct rasterization of projected kernels in the case of 2D splats (Section 4.2.1) and volumetric kernels (Section 4.2.2). Each projected kernel is represented by one OpenGL vertex and pixel shaders are used for their rasteriza-

tion [BK03].

4.2.1. 2D splats

For rasterizing planar elliptical surface splats we follow the per-pixel projectively correct ray casting approach as introduced by [BSK04]. Following the notation of [BSK04], a splat S_j is defined by its center $\mathbf{c}_j$ and two orthogonal tangent directions $\mathbf{u}_j$ and $\mathbf{v}_j$. These tangent vectors are scaled according to the principal radii of the elliptical splat such that an arbitrary point $\mathbf{q}$ in the splat's embedding plane lies in the interior of the splat if its local parameter values u and v satisfy the condition

$$u^2 + v^2 = (\mathbf{u}_j^T(\mathbf{q} - \mathbf{c}_j))^2 + (\mathbf{v}_j^T(\mathbf{q} - \mathbf{c}_j))^2 \leq 1. \tag{4.2}$$

The rasterization of a splat S_j is performed by sending its center $\mathbf{c}_j$, tangent axes $(\mathbf{u}_j, \mathbf{v}_j)$, and optional material properties to OpenGL, which are then processed by custom shaders for both the vertex and the pixel stage. The vertex shader conservatively estimates the size d of the projected splat based on a perspective division of the larger of the ellipse radii by the eye-space depth value c_z of the splat center, followed by a window-to-viewport scaling.

This causes the single OpenGL vertex $\mathbf{c}$ to be rasterized as a $d \times d$ image space square, each pixel (x, y) of which is then tested by a pixel shader to lie either inside or outside of the projected elliptical splat contour. Local ray casting through the corresponding projected point $\mathbf{q}_n$ on the near plane yields the eye space point $\mathbf{q}$ on the splat's supporting plane. From this projectively exact 3D position the local parameter values, (u, v), can be determined and tested as shown in Equation 4.2. While pixels corresponding to points outside the splat are discarded, pixels belonging to the splat are accepted and processed further. If a pixel (x, y) is accepted, its weighting factor is determined as

$$w(x, y) = h(\sqrt{u^2 + v^2}), \tag{4.3}$$

where $h(\cdot)$ is typically chosen as a Gaussian. To allow for exact blending and occlusion the pixel's depth value has to be adjusted in order to correspond to the computed 3D position $\mathbf{q}$. This finally results in a per-pixel projectively correct rasterization of elliptical splats.

4.2.2. Volumetric kernels

For the rasterization of volumetric kernels which can be regarded as ellipsoid primitives, we follow the computations described in the work of Sigg et al. [SWBG06]. They propose an efficient perspective correct rendering technique for quadric primitives based on GPU-accelerated splatting using per-pixel ray-casting. In the following, we will outline the necessary computation steps.

In general, quadratic surfaces are defined as the set of roots of a polynomial of degree two:

$$f(x,y,z) = Ax^2 + 2Bxy + 2Cxz + 2Dx + Ey^2 + 2Fyz + Gy + Hz^2 + 2Iz + J = 0. \quad (4.4)$$

The shape of the quadric is solely determined by the coefficients A through J. Using homogeneous coordinates $\mathbf{x} = (x, y, z, 1)^T$ the quadric can compactly be written using the bilinear form $\mathbf{x}^T\mathbf{Q}\mathbf{x} = 0$ with the conic matrix

$$\mathbf{Q} = \begin{pmatrix} A & B & C & D \\ B & E & F & G \\ C & F & H & I \\ D & G & I & J \end{pmatrix}, \quad (4.5)$$

which is invariant under perspective projections. Figure 4.1 illustrates the complete coordinate system transformation pipeline. Due to the fact that this matrix is symmetric, it can be put into a normalized diagonal form by a basis transformation **T**:

$$\mathbf{Q} = \mathbf{T}^{-T}\mathbf{D}\mathbf{T}^{-1}, \quad (4.6)$$

with **D** diagonal, $d_{ii} \in \{0, \pm 1\}$. Coefficient matching of Equation 4.4 with the polynomial which defines an ellipsoid

$$x^2 + y^2 + z^2 - 1 = 0 \quad (4.7)$$

results in **D** being of the form

$$\mathbf{D} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} \quad (4.8)$$

The transformation matrix **T**, called variance matrix, expresses the basis of the parameter space in object coordinates. The columns contain the axes **u**, **v**, **w** and center **c** of the quadric

$$\mathbf{T} = \begin{pmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{c} \\ 0 & 0 & 0 & 1 \end{pmatrix}. \quad (4.9)$$

The ellipsoid represents the bounding ellipsoid of the 2D splat axes and the velocity. As, in general, the velocity vector is not perpendicular to the splat plane, the orthogonal ellipsoid axes

u, **v** and **w** need to be recomputed first. For a detailed description of the involved computations see Appendix C.

In analogy to the case of 2D splats, the kernel center in clip coordinates and GL point primitive size needs to be calculated in the vertex stage of the shader. The point size is determined by the bounding box of the kernel projected to screen. A tight axis-aligned bounding box $[b_{x,1}, b_{x,2}] \times [b_{y,1}, b_{y,2}]$ of the projected ellipsoid in clip coordinates is computed first, which is defined by four intersecting half-spaces. Each half-space is given by an equation of the following form:

$$\mathbf{n}_c^T \mathbf{x}_c \leq 0. \tag{4.10}$$

This condition can easily be enforced in parameter space, where the quadric is defined by the normalized diagonal matrix $\mathbf{D}$. In parameter space, the ellipsoid coincides with the S^2 sphere and each point on the sphere also corresponds to a normal of a tangent plane. Therefore, the touching condition in parameter space becomes

$$\mathbf{n}_p^T \mathbf{D} \mathbf{n}_p = 0. \tag{4.11}$$

This condition is then transformed to clip space and the resulting quadratic equation is solved for the bounding box coordinates. Finally, the viewport transformation is applied to the bounding box size and half of the larger value (width or height) is set as the GL point size. The vertex position of the point primitive coincides with the center of the bounding box in clip coordinates.

The task of the fragment shader is to kill fragments that are not covered by the ellipsoid. For the corresponding ray intersection problem, again the roots of a quadratic equation need to be found. If the equation has no real solution, the ray does not intersect the quadric and the fragment can be killed.

Finally, the weighting factor of the pixel (x, y) needs to be computed in analogy to Equation 4.3. The mathematically strict solution would be to integrate the volumetric kernel over the ray segment between the intersection points. A further difficulty is added to this problem by the fact that the volumetric kernel does not represent an exact 3D Gaussian unless the velocity vector is perpendicular to the 2D splat plane. In our implementation, however, we approximate this value based on the observation that the opaqueness has to grow with the length of the line segment between the intersection points. To calculate the length of the intersecting ray segment in parameter space, we transform the previously computed intersection points back to parameter space and retrieve $\mathbf{x}_{p,1}$ and $\mathbf{x}_{p,2}$. Since the points are located on the S^2 sphere, their distance lies within the interval $[0, 2]$. The weight is then computed as

$$w(x, y) = h(\frac{||\mathbf{x}_{p,1} - \mathbf{x}_{p,2}||}{2}), \tag{4.12}$$

where $h(\cdot)$ is chosen as a Gaussian.

While normal vector and material properties of a fragment can simply be set as the underlying 2D splat's normal and material properties since these are constant over the 2D splat, the depth calculation is more involved. For this, we have to recapitulate that the volumetric kernel is

generated by convolving the 2D Gaussian with a 1D Gaussian along the direction of motion. This means, each point in the interior of the volumetric kernel lies on the motion trail of its 2D splat source point which can be found by tracing back along the instantaneous velocity direction. The correct way of determining the fragment depth would be to take sample points on the intersecting ray segment and average the depth values of their corresponding source points. For simplicity, we resort to using the depth value of the intersection mid-point.

4.2.3. EWA Approximation

The complete EWA filter is composed of an object space reconstruction kernel and a band-limiting screen space prefilter. As the required computations are quite involved, many rendering approaches simply omit the screen space filter and use the reconstruction kernel only. However, in the case of extreme minification, when the size of projected splats falls below one pixel, the signal corresponding to the accumulated projected splats may have frequencies higher than the Nyquist frequency of pixel sampling grid, resulting in the alias artifacts shown in the top image of Figure 4.9.

The work of Swan et al. [JESMM+97] proposes a simple - and hence efficient - heuristic for approximating the EWA screen-space filter which still provides high-quality anti-aliasing in magnified and minified regions. By clamping the size of projected splats to be at least 2x2 pixels it is guaranteed that enough fragments are generated for antialiasing purposes, even for splats projecting to sub-pixel areas. This restriction on the minimum size can easily be incorporated into the vertex shader.

Instead of computing the weight $w(x, y)$ based only on the reconstruction filter, the pixel shader is adjusted to compute two radii $r_{3D} := u^2 + v^2$ (see Equation 4.12) and $r_{2D} := d(x, y)^2/r^2$, with $d(x, y)$ being the 2D distance of the current fragment from the respective projected kernel center and $r = \sqrt{2}$ being the band-limiting screen-space filter radius. A given fragment is then accepted if it lies within the union of the low-pass and the reconstruction filter (Figure 4.10).

$$\tilde{r}(x, y) := min\{r_{2D}(x, y), r_{3D}(x, y)\} \leq 1, \tag{4.13}$$

i.e., either if it corresponds to a 3D point within the kernel's interior, or if it lies within a certain radius around the projected kernel center. The final weight corresponding to Equation 4.12 is computed as $w(x, y) = h(\sqrt{\tilde{r}(x, y)})$.

Notice that the minimal splat size is only enforced in the attribute pass, but not in the visibility pass. This means that the ϵ-depth test, which is simulated by the two rendering passes, is not applied to those pixels which are additionally generated on silhouettes by the screen space filter. In contrast, these pixels are blended with the surface parts behind them, which results in a pseudo edge antialiasing for object silhouettes.

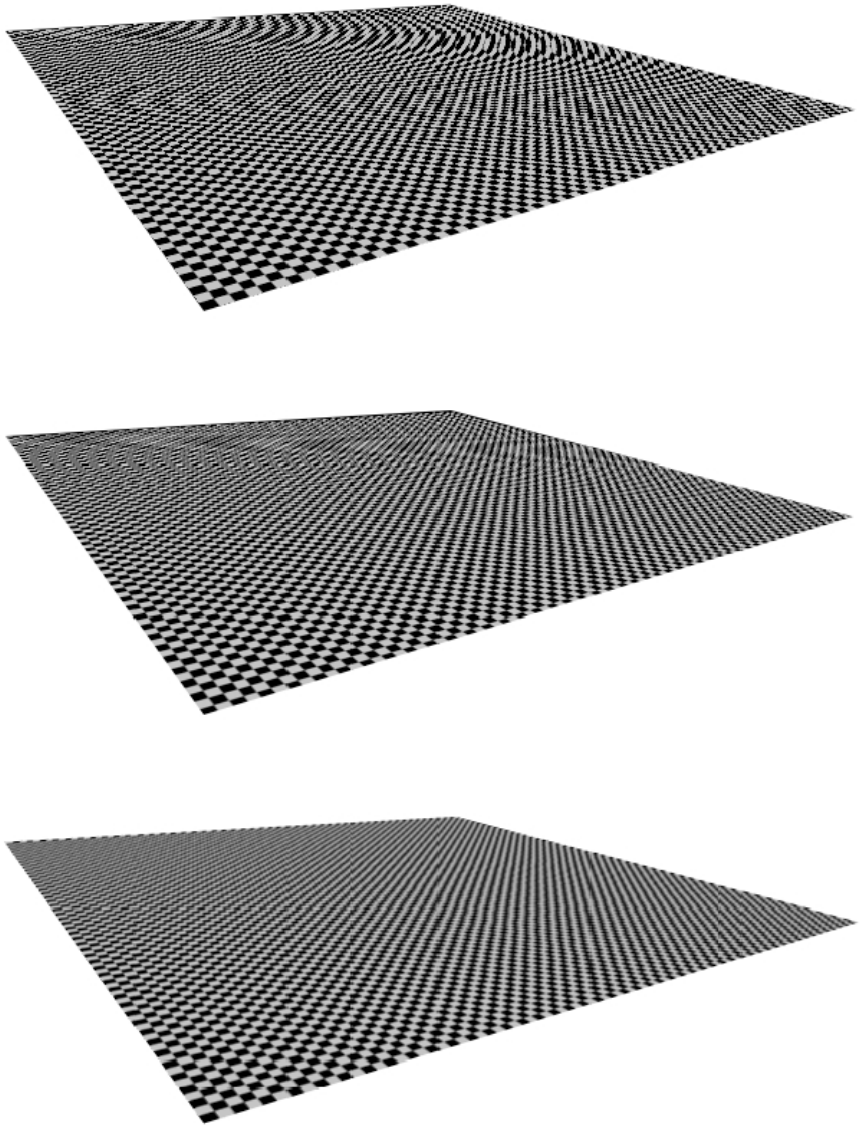

Figure 4.9.: *Antialiasing Comparison. (top) Use of only a reconstruction filter. (middle) Full Screen Anti-Aliasing using supersampling. (bottom) EWA approximation. [Botsch:2005:HSS], Figure 4.*

4.3. Temporal supersampling

For blending the temporal supersamples within the exposure interval, we added a further accumulation texture buffer. To generate an output image, we loop over the temporal samples, transform the scene objects according to their current location and render the point model with 2D splats or volumetric kernels as described above. The corresponding temporal weighting function value is passed to the shaders of the final normalization pass and written into the output fragment alpha value. The output value is then blended with the accumulation texture color by the blending function

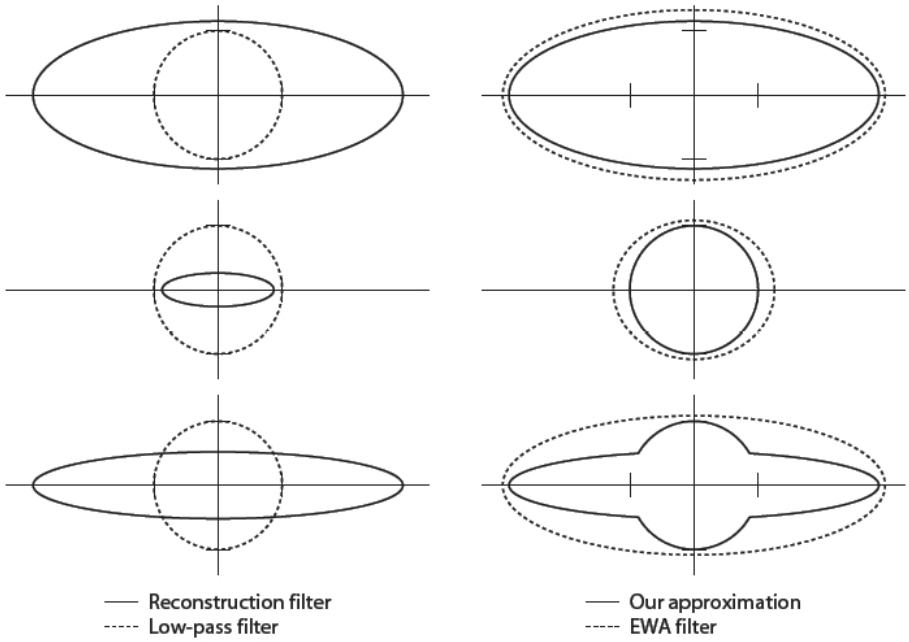

Figure 4.10.: *(left) Three configurations with a reconstruction filter and a low-pass filter. (right) Comparison between the combined approximation and the EWA filter. [Botsch:2005:HSS], Figure 4.*

$$(Rd, Gd, Bd, Ad) = (As * Rs + Rd, As * Gs + Gd, As * Bs + Bd, As + Ad), \qquad (4.14)$$

where (Rd, Gd, Bd, Ad) is the accumulation texture RGBA value at the fragment location, and (Rs, Gs, Bs, As) is the shader output.

4.4. Time bucketing

The drawing procedure works in a similar way as for the accumulation of temporal samples. The difference of this approach consists of first looping over the individual time buckets. When performing the visibility and attribute pass, all temporal samples which fall into the current bucket are rendered at once as if they were a single object. The normalization and shading pass only needs to be performed once per time bucket. Thus, correctness of the visibility is sacrificed to rendering speed.

4.5. Reference implementation

A motion blurred image $G_{c,t_0,t_1}(\mathbf{x})$ is computed by means of Quasi Monte Carlo integration and raytracing. We define the image as an array of pixels with screen coordinates (x, y) and corresponding color values $c(x, y)$. The number of spatial and temporal supersamples is denoted by s_s and s_t, respectively.

To compute the color value $c(x, y)$, repeat s_s times:

1. Choose a random spatial supersample position $(x, y)_i$ within the pixel area. To prevent an aggregation of sample positions, the sampling distribution is based on a jittered 2D grid as described in [KK].
2. Repeat s_t times:
   a) Choose a random temporal supersample time t within the exposure interval $[t_0, t_1]$. In analogy to the spatial sampling, the temporal sampling distribution follows a 1D jittered grid. To control the trade-off between visual disturbing mach-banding artifacts and noise, the jittered grid is sampled more densely and for each spatial supersample, the temporal supersamples are chosen as a random set out of this finer jittered grid.
   b) Transform the objects of the scene to their positions at time t.
   c) Shoot a ray from the viewpoint through the screen location $(x, y)_i$ and intersect the object splats. Determine the intersection point which is closest to the viewpoint. To perform ϵ-blending of the splats, determine all further intersection points within ϵ distance from the closest intersection. To compute the color c_i of the sample, blend the color values of the intersection points. The respective alpha values are determined by the underlying splat kernel functions.
   d) Add c_i to the final color $c(x, y)$, weighted by the time weighting function $\mathbf{a}(t)$. Accumulate $\mathbf{a}(t)$ to the normalization value a.
3. Finally divide $c(x, y)$ by the normalization value a.

With this sampling strategy, we do not exploit the whole advantages of jitterd grid sampling since for each pixel grid location we sample at each time grid location. It would be better to either jitter x, y and t together, choose one random t for each (x_i, y_i), or to use one of the multisampling methods suggeseted in Chapter 3 of the book [SM03].

4.6. Animation framework

The scene to be rendered is described in a simple xml format where point model files (internal bin file format or sfl file) are listed with their corresponding animation file (anim file format as generated by the Maya animImportExport plug-in). An example scene xml is shown in Figure 4.11.

According to the user's choice, the animation transforms inbetween the keyframes are either linearly interpolated or calculated by cubic spline interpolation. For the latter, we use the code of [Bur].

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
- <scene>
  - <object>
      <pointModel>scenes\models\sphere.bin</pointModel>
      <animation>scenes\animations\thrown.anim</animation>
    </object>
  - <object>
      <pointModel>scenes\models\santa.sfl</pointModel>
      <animation>scenes\animations\bouncing.anim</animation>
    </object>
  - <object>
      <pointModel>scenes\models\igea.sfl</pointModel>
      <animation>scenes\animations\rolling.anim</animation>
    </object>
  </scene>
```

Figure 4.11.: *An example scene xml file.*

5

Results

In this chapter, we show results of the volumetric kernel approach for a varying number of temporal samples per frame and compare their visual quality with images as generated by an accumulation of temporal supersamples with original 2D surface splatting. The accompanying Table 5.1 displays the corresponding framerates. All measurements were taken on a computer with Pentium Intel Core2 Quad Q6700 CPU and NVidia GeForce 8800 Ultra GPU. In the following, we are going to discuss the results.

Figure 5.1 illustrates that for volumetric kernels a very small number of temporal samples is sufficient to create smoothly motion blurred images. Especially in the case of linear motion, even a single temporal sample produces satisfactory results since a volumetric kernel represents a piece-wise linear approximation of the motion trajectory. Sampling beyond one temporal sample is in fact superfluous for this example since there are no visibility or shading changes which would require a higher sampling rate. On the other hand, motion blur effects which are generated by accumulating images taken at a higher frequency than the framerate require a fairly large number of samples to alleviate disturbing staggering which severely presses the framerate down. In Figure 5.2 (e), the lower left tentacle of the octopus has partly vanished. This is due to the fact, that the instantaneous velocity vector of each of the three temporal samples in this region differs relatively largely in direction and the volumetric kernels do not overlap anymore. This implies that by adjusting the velocity vectors we have to make sure that volumetric kernels of subsequent time instants overlap sufficiently such as to provide a smooth interpolation over time. Figure 5.3 reveals a problem with the current solution on how to handle transparency which is introduced by sweeping an object along its trajectory over time. Since the 2D Gaussian surface kernels and therefore also the volumetric kernels do not guarantee a partition of unity, we tend to get too small or too large values at areas where the kernels overlap if we compute transparency by summing up the corresponding Gaussian alpha value. For densely sampled models with a high ratio of overlapping area for a kernel as in the case of

the shell model, the temporal transparency becomes too opaque.

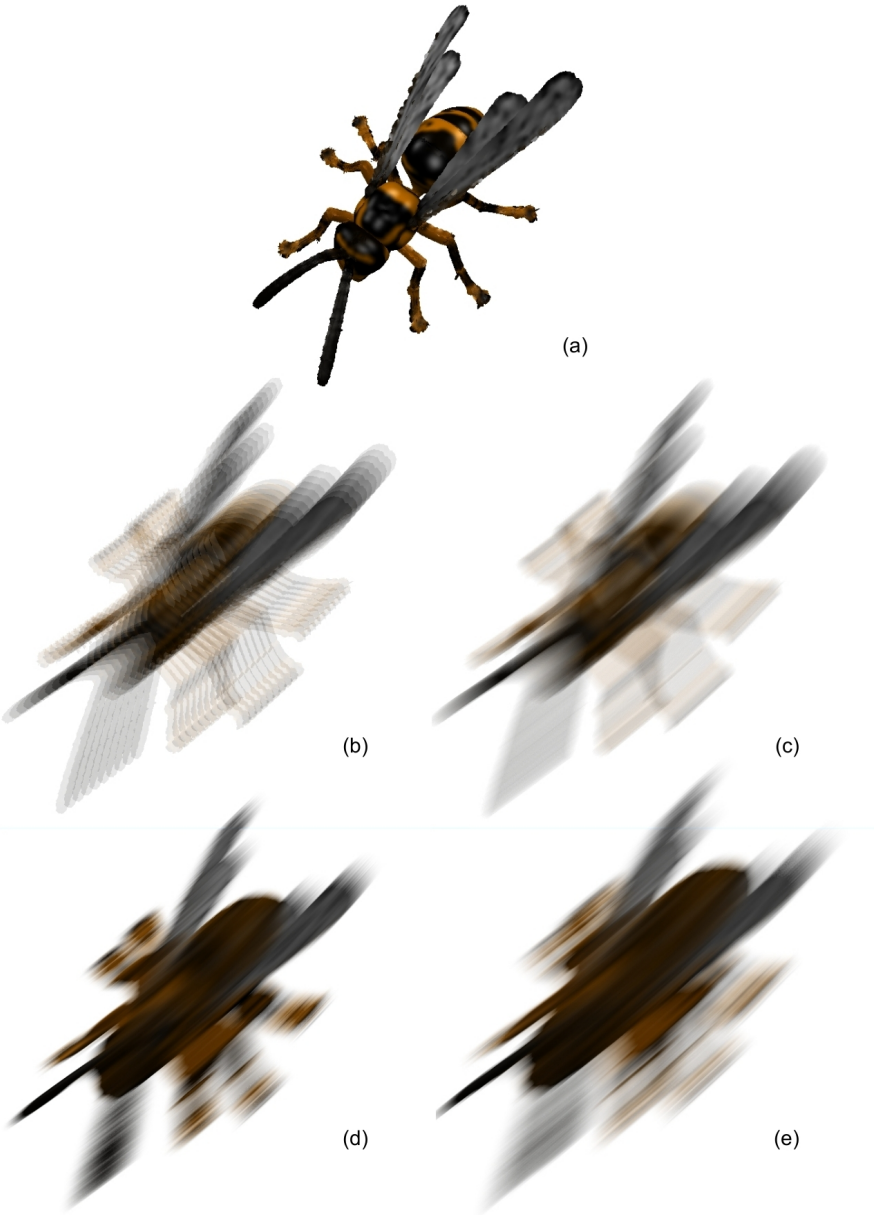

Figure 5.1.: *Wasp model (49604 points) moving in diagonal direction from the upper right to the lower left corner. (a) Rendering without motion blur and 2D surface splats. (b)-(c) show motion blurred images generated by accumulating (b) 10 and (c) 50 temporal supersamples, respectively, and 2D surface splats. (d)-(e) are rendered with volumetric kernels and (d) 1 and (e) 3 temporal samples, respectively. Table 5.1 enlists the corresponding framerates.*

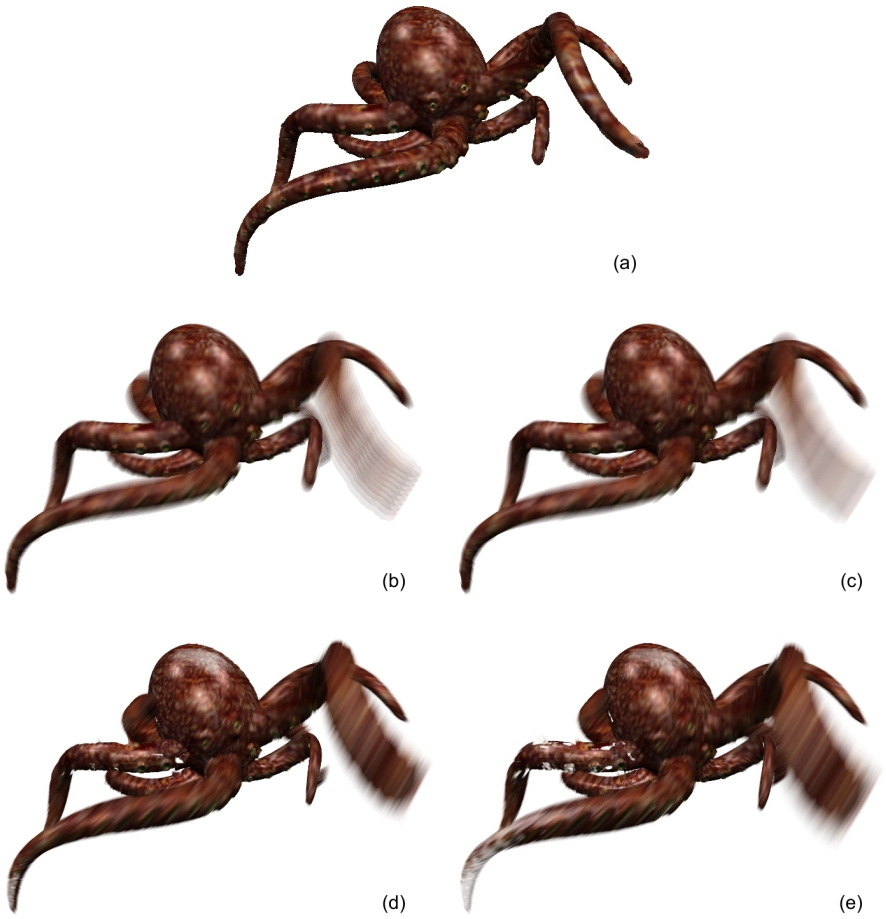

Figure 5.2.: *Octopus model (465878 points) rotating about its center. (a) Rendering without motion blur and 2D surface splats. (b)-(c) show motion blurred images generated by accumulating (b) 10 and (c) 50 temporal supersamples, respectively, and 2D surface splats. (d)-(e) are rendered with volumetric kernels and (d) 1 and (e) 3 temporal samples, respectively. Table 5.1 enlists the corresponding framerates.*

Image	*Temporal samples*	*Kernel type*	*Fps Figure 5.1*	*Fps Figure 5.2*	*Fps Figure 5.3*
(a)	1	2D splats	39.6	29.9	37.9
(b)	10	2D splats	7.9	4.7	10.3
(c)	50	2D splats	2.3	1.1	4.4
(d)	1	volumetric	3.5	7.0	0.8
(e)	3	volumetric	1.7	3.8	0.4

Table 5.1.: *Framerates of the rendered images in Figures 5.1, 5.2 and 5.3*

Figure 5.3.: *Ammonite and shell model (277269 and 168120 points, respectively). The ammonite is moving towards the camera and rotating about its center while the shell is moving horizontally from left to right. (a) Rendering without motion blur and 2D surface splats. (b)-(c) show motion blurred images generated by accumulating (b) 10 and (c) 50 temporal supersamples, respectively, and 2D surface splats. (d)-(e) are rendered with volumetric kernels and (d) 1 and (e) 3 temporal samples, respectively. Table 5.1 enlists the corresponding framerates.*

6

Conclusion and Outlook

6.1. Summary

In this thesis, we have described a method for rendering point-based models with motion blur effects. First, we presented an overview of related work on the topics of temporal antialiasing and EWA splatting in Chapter 2. Chapter 3 derived the theoretical basis by extending EWA surface splatting in the temporal dimension. As central idea of our approach, we introduced a volumetric kernel which unifies a spatial and temporal component and allows for a continuous reconstruction of the scene in space as well as time dimension. In the following Chapter 3.6.4, we described the modified three-pass algorithm in contrast to the original 2D surface splatting. Finally, result images and performance measurements were presented in Chapter 4.6.

6.2. Future Work

For the time being, when sampling at a certain time instant, we take a temporal sample at the motion trajectory of every point of the model. However, since individual point motion paths may differ strongly in length and shape, it would be very advantageous to choose the sampling density for the individual point samples of the model adaptively. This would allow for incorporating the stochastic sampling strategies of distributed raytracing [CPC84] and frameless rendering [BFMZ94] [DWWL05]. While these are applied on a per-pixel level, our idea would in contrast sample on the level of volumetric kernels. The problem which arises is, that the evaluation of the visibility function has to be adjusted. One could think of grouping the temporal samples as in the time bucketing solution.

Also, it is necessary to rethink the solution for computing the temporal alpha texture. As has

been revealed in Figure 5.3, the result becomes incorrect if the kernels do not guarantee a partition of unity. An option would be to move temporal alpha evaluation to the visibility pass. That is, the texture would be computed separately for each sampled time instant or time bucket. This would make it possible to guarantee a partition of unity by calculating the intermediate result as a weighted average instead of a sum. In the attribute pass, the intermediate values for a time instant could then be summed up in the final temporal alpha texture.

Since volumetric kernels tend to be elongated and thin, a rasterization process which exploits the OpenGL point primitive becomes inefficient. Therefore it could make sense to resort to rendering a textured quad for each volumetric kernel.

A

Accompanying CD Contents

This appendix gives an overview of the contents on the accompanying DVD. It contains the executables for the motion blur splatting and the reference applications, the respective source codes and the thesis PDF.

A.1. Application executables

A.1.1. Motion Blur Splatting

The application for motion blur splatting can be found in the folder `\MotionBlurSplatting`. In order to install the application simply copy the folder to your hard disk. The application executable requires Windows XP or Vista and an installation of .NET 3.2 or later versions. Moreover, a NVidia GeForce GPU 8 or later series is required. To start the application, execute `splatting.exe`.

A.1.2. Reference Implementation

The executable of the reference implementation is stored in the folder `\ReferenceImplementation`. In order to install the application copy the folder to your hard disk. The application executable requires Windows XP or Vista and an installation of .NET 3.2 or later versions. Since computations are performed completely on the CPU, there are no requirements on the GPU. Execute `ReferenceImplementation.exe` to start the application.

A.2. Implementation source code

The source code of the motion blur splatting and reference implementation projects can be found in the folders `\Source\MotionBlurSplatting` and `\Source\ReferenceImplementation`, respectively. The subfolders which separate different parts of the projects are organized in almost the same way for both applications. The source code files and Microsoft Visual Studio 9 projects are stored in the root folder in the case of the motion blur splatting application and in the subfolder `\Source\ReferenceImplementation` for the reference implementation. The subfolder `\scenes` contains a collection of example scene xml files which refer to point models in the subfolder `\scenes\models` and Maya .anim files in the subfolder `\scenes\animations` (for details see Section 4.6).

A.3. Thesis PDF

The thesis PDF is located in the folder `\Thesis`.

B

Implementation Overview

In this appendix, we give a brief overview of the central classes in our C++ implementations of the motion blur splatting application (Section B.1) and reference implementation (Section B.2).

B.1. Motion Blur Splatting

B.1.1. Class: GLSplat

The class `GLSplat` provides an interface to the class `GLSplat2D` for 2-dimensional surface splat rendering which itself is the base class for `GLSplat3D`. The latter implements the volumetric kernels rendering algorithm. Each class contains a version of the following main functions:

```
void draw();
```

This is the function which is called on a redraw callback. It retrieves the input parameters from the GUI, updates the scene and draws the current frame.

```
void draw_arrays(unsigned int shaderId);
```

For a drawing pass with the ID `shaderID`, this function loops over all models in the scene, applies the corresponding animation transforms and sends the velocity and time weight parameters to the shader. Finally, it sends the range of vertices in the VBO which is occupied by the current model down the drawing pipeline by calling `glDrawArrays`.

```
void normalization();
```

Here, the normalization pass is performed.

File	*Description*
`vp_pass1.glsl`	2D surface splats: Vertex shader
`fp_pass1.glsl`	2D surface splats: Fragment shader
`vp_pass1_quadsurf.glsl`	Volumetric kernels: Vertex shader
`fp_pass1_quadsurf.glsl`	Volumetric kernels: Fragment shader

Table B.1.: *Visibility pass shaders.*

```
void update_shader_parameters(GLhandleARB program);
```

Being called once every time before drawing a frame, this function sends the viewing transform parameters to a shader specified by `program`.

B.1.2. Frame buffers

For storing frame buffer data, we use two frame buffer objects (FBOS). The first one, `fbo_`, holds textures which are updated for every temporal sample (or time bucket), that is, the depth texture `depth_tex_`, the color texture `color_tex_` and the normal texture `normal_tex_`. The latter is used for storing the temporal alpha values in the case of volumetric kernel rendering as deferred shading is not applied here and a normal texture becomes superfluous. The second FBO, `fbo_accum_`, is used for accumulating the intermediate results of each temporal sample (or time bucket).

B.1.3. Class: GLState

This class stores all relevant OpenGL states and can therefore provide some nice and efficient functions like projecting, unprojecting, eye point or viewing direction. All state changes are applied by using functions of this class instead of calling the OpenGL functions directly to maintain consistency.

B.1.4. Class: Spline

The code for this class is taken from [Bur]. It provides different spline interpolation methods which we use for interpolating data between animation keyframes.

B.1.5. Shaders

In this section, we shortly describe how the shaders are structured.

For the visibility pass (Table B.1), the shader code for the 2-dimensional surface splat and volumetric kernel rendering algorithms differ only slightly. The earlier uses OpenGL fixed

File	*Description*
	2D surface splats:
`vp_pass2.glsl`	Vertex shader
`fp_pass2_gouraud_accum.glsl`	Fragment shader (Gouraud shading)
`fp_pass2_phong_accum.glsl`	Fragment shader (Phong shading)
	Volumetric kernels:
`vp_pass2_quadsurf.glsl`	Vertex shader
`fp_pass2_quadsurf.glsl`	Fragment shader

Table B.2.: *Attribute pass shaders.*

File	*Description*
	2D surface splats:
`fp_normalize_gouraud_accum.glsl`	Fragment shader (Gouraud shading)
`fp_normalize_phong_accum.glsl`	Fragment shader (Phong shading)
`fp_normalize_quadsurf.glsl`	Volumetric kernels: Fragment shader
`fp_normalize_color_accum.glsl`	Fragment shader (color texture display)
`fp_normalize_depth_accum.glsl`	Fragment shader (depth texture display)
`fp_normalize_normal_accum.glsl`	Fragment shader (normal texture display)
`fp_normalize_alpha_quadsurf.glsl`	Fragment shader (alpha texture display)

Table B.3.: *Normalization pass shaders.*

functionality depth testing in the attribute pass and, therefore, updates the values in the FBO depth buffer. To make it possible to display the content of the depth buffer, the same values are stored in the depth texture `depth_tex_`. This texture is used as an input to the attribute pass in the case of volumetric kernels rendering.

In the attribute pass (Table B.2) for 2-dimensional surface splat rendering, two separate fragment shaders implement Gouraud and Phong shading, respectively. For Gouraud shading, lighting computations are performed on the color values before storing them to the color texture. On the other hand, the fragment shader for Phong shading does not yet evaluate lighting for the current fragment, but accumulates the normal in the normal texture for deferred shading.

The last four enlisted shaders in Table B.3 display the content of the color, depth, normal and temporal alpha textures, respectively, on the screen. As the volumetric kernels rendering does not accumulate normal values, we reuse the normal texture for storing the temporal alpha values.

B.2. Reference Implementation

B.2.1. Class: MainApp

This is the central class of the application which controls the frame buffers, camera state, scene and GUI and handles the drawing callback.

B.2.2. Class: Scene

This class contains the models of the scene and their animations and controls their state. The most important function is

```
void TraceRay(Vec3f* cameraPos, Vec3f* ray, GLfloat time, ColorRGB*
resColor);,
```

which traces the viewing ray `ray` with eye position `cameraPos` at time `t` and returns the resulting color in `resColor`.

B.2.3. Class: Animation

This class provides an interface to the derived classes `StaticAnimation`, `LinearAnimation`, `RotationalAnimation` and `KeyframeAnimation`. The latter is defined by reading in data from a Maya .anim file. The essential functions of the `Animation` interface are as follows.

```
void TransformRay(Vec3f* origin, Vec3f* ray, GLfloat time);
```

Instead of transforming the object itself which is very expensive for large point models, the viewing ray `ray` and eye position `origin` are transformed by the inverse modelview transform. The transform is determined by either linearly interpolating between the start and end states of the object inbetween two frames or by cubic spline interpolation in the case of keyframed animations. `time` denotes the time instant between the current two frames.

```
void TransformObjectIntersection(Vec3f* intersectionPoint, Vec3f*
intersectionNormal);
```

Since during the intersection computation the viewing ray and eye position have been transformed instead of the object, the determined intersection point location `intersectionPoint` and normal `intersectionNormal` have to be adjusted accordingly. This is done by applying the transforms which have been calculated during the call of the function `TransformRay`.

The animation sequence can be controlled by calling the functions

```
void NextFrame(void);
void PreviousFrame(void);
void Reset(void);
```

B.2.4. Class: SceneObject

This class represents an object of the scene. It provides an interface for the derived classes `Sphere` for an ideal sphere and `PointModel` for a point model. The most important functions of the interface are listed below.

```
GLfloat Intersect(Vec3f* origin, Vec3f* ray);
```

This function intersects a viewing ray `ray` with the object and returns the intersection depth value. `origin` denotes the eye position. A bounding box tree over the point model splats accelerates the computation. The normal and position of the last intersection point are set and can be retrieved by the functions

`Vec3f* GetLastIntersectionNormal(void);` and
`Vec3f* GetLastIntersectionPoint(void);`.

B.2.5. Class: QMCSampler

Based on the sample code in [KK], the `QMCSampler` class implements a Quasi Monte Carlo sampling algorithm.

C

Ellipsoid axes computation

This appendix will first derive the 3D Gaussian function which is created by convolving a 2D Gaussian surface kernel with a 1D Gaussian along the direction of instantaneous velocity. Then, it describes how to calculate the orthogonal axes **s**, **t** and **u** of the ellipsoid which represents this 3D Gaussian. Based on the surface splat axes $\mathbf{a}_0$ and $\mathbf{a}_1$ and the instantaneous velocity vector **v**, we want to calculate a tight spanning ellipsoid (Figure C.1). Section C.2 concerns itself with the case of general elliptical splats, whereas Section C.3 will state the calculation for the more simple case of circular surface splats. Table C.1 provides an overview over the notation.

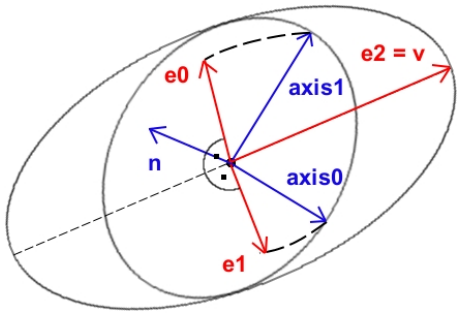

Figure C.1.: *Computation of the orthogonal ellipsoid axes* $\mathbf{e}_0$, $\mathbf{e}_1$ *and* $\mathbf{e}_2$.

The center of the kernel is chosen as the center of the original surface splat. It can be ignored in the following calculations, that is, set to $(0, 0, 0)$, without any loss of generality.

Quantity	*Notation*
$\mathbf{x} = (x, y, z)$	An arbitrary point within the volumetric kernel
$\mathbf{c} := (0, 0, 0)$	Kernel center
$\mathbf{a}_0$, $\mathbf{a}_1$	Axes of the 2D surface splat
$\mathbf{a}_i = (a_{ix}, a_{iy}, a_{iz})$	
$\mathbf{n}$	Normal of the 2D surface splat
$\mathbf{v}$	Instantaneous velocity vector
$\mathbf{v'} := \frac{\mathbf{v}}{\|\mathbf{v}\|}$	Normalized instantaneous velocity vector
$\mathbf{v'} = (v'_x, v'_y, v'_z)$	
$\mathbf{e}_0$, $\mathbf{e}_1$, $\mathbf{e}_2$	Axes of the volumetric kernel
$\mathbf{e}_i = (e_{ix}, e_{iy}, e_{iz})$	
$\mathbf{o}$	Source point of point $\mathbf{x}$ on the 2D surface splat

Table C.1.: *Notation.*

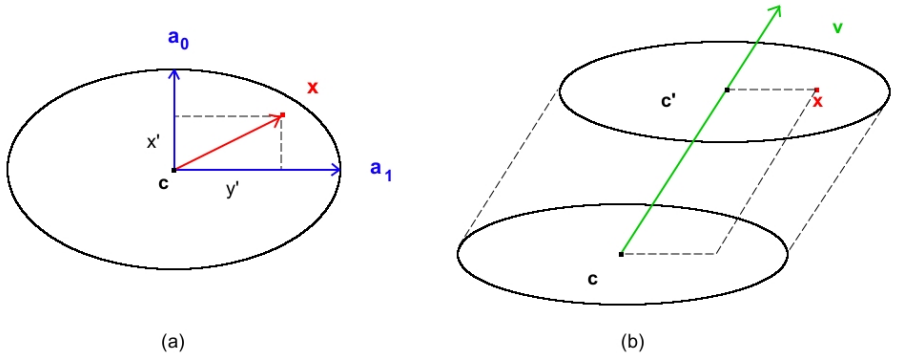

Figure C.2.: *Construction of the 3D Gaussian kernel. (a) Point* **x** *lies on the ellipse. (b) The ellipse through point* **x** *which is shifted parallelly along the velocity vector* **v**.

C.1. 3D Gaussian function

A point $\mathbf{x} = (x, y, z)$ lies on a 2D ellipse, if the segments x' and y' which are calculated by normal projection to the ellipse axes $\mathbf{a}_0$ and $\mathbf{a}_1$ (see Figure C.2 (a)) fulfill the following condition:

$$\frac{x'^2}{||\mathbf{a}_0||^2} + \frac{y'^2}{||\mathbf{a}_1||^2} \leq 1. \tag{C.1}$$

We can solve for x' and y' by calculating the normal projection of $\mathbf{x}$ to the splat axes:

$$\cos\alpha = \frac{x'}{||\mathbf{x}||} \tag{C.2}$$

$$\mathbf{x}\cdot\mathbf{a}_0 = ||\mathbf{x}||\cdot||\mathbf{a}_0||\cdot\cos\alpha \tag{C.3}$$

$$= ||\mathbf{a}_0||\cdot x' \tag{C.4}$$

$$\rightarrow x' = \frac{\mathbf{x}\cdot\mathbf{a}_0}{||\mathbf{a}_0||} \tag{C.5}$$

$$y' = \frac{\mathbf{x}\cdot\mathbf{a}_1}{||\mathbf{a}_1||}, \tag{C.6}$$

where α denotes the angle which is enclosed by $\mathbf{x}$ and $\mathbf{a}_0$. Inserting Equation C.6 into Equation C.7 gives us the following ellipse equation:

$$\frac{(\mathbf{x}\cdot\mathbf{a}_0)^2}{||\mathbf{a}_0||^4} + \frac{(\mathbf{x}\cdot\mathbf{a}_1)^2}{||\mathbf{a}_1||^4} \leq 1. \tag{C.7}$$

The 2D Gaussian kernel G_{2D} which corresponds to this ellipse now has the following form:

$$G_{2D} = Ae^{-(\frac{(\mathbf{x}\cdot\mathbf{a}_0)^2}{||\mathbf{a}_0||^4}+\frac{(\mathbf{x}\cdot\mathbf{a}_1)^2}{||\mathbf{a}_1||^4})}, \tag{C.8}$$

where the parameter A ensures that the Gaussian integrates to 1.

As a next step, we derive the 3D Gaussian function of a volumetric kernel. For this, we consider a point $\mathbf{x}$ which is located at a certain distance from the 2D ellipse (Figure C.2 (b)). The value of the 3D Gaussian at $\mathbf{x}$ is calculated by convolving the source point on the ellipse with a 1D Gaussian along the direction of velocity. The value of the 2D Gaussian at the source point can be retrieved by placing an ellipse through $\mathbf{x}$ which is shifted parallely along the velocity vector. The center $\mathbf{c}'$ of this shifted ellipse is

$$\mathbf{c}' = s\cdot\mathbf{v'} + \mathbf{c} \tag{C.9}$$

$$= s\cdot\mathbf{v'} \tag{C.10}$$

$$= (s\cdot v'_x, s\cdot v'_y, s\cdot v'_z), \tag{C.11}$$

where $s \in \mathbb{R}$ is a scaling parameter. It can be determined by the fact that

$$s\cdot v'_z = z' \rightarrow s = \frac{z'}{v'_z}, \tag{C.12}$$

where z' is the normal projection of $\mathbf{x}$ to the ellipse normal. The ellipse equation of the shifted ellipse at $\mathbf{x}$ then becomes

$$\frac{((\mathbf{x}-\mathbf{c}')\cdot\mathbf{a}_0)^2}{||\mathbf{a}_0||^4} + \frac{((\mathbf{x}-\mathbf{c}')\cdot\mathbf{a}_1)^2}{||\mathbf{a}_1||^4} \leq 1. \tag{C.13}$$

and the corresponding 2D Gaussian function is

$$G_{2D}* = Be^{-(\frac{((\mathbf{x}-\mathbf{c}')\cdot\mathbf{a}_0)^2}{||\mathbf{a}_0||^4}+\frac{((\mathbf{x}-\mathbf{c}')\cdot\mathbf{a}_1)^2}{||\mathbf{a}_1||^4})}, \tag{C.14}$$

where the parameter B ensures that the Gaussian integrates to 1. The form of the 1D Gaussian which takes the role of the temporal filter is

$$G_{1D} = Ce^{-(\frac{s^2}{||\mathbf{v}||^2})} = Ce^{-(\frac{z'^2}{v'_z\cdot||\mathbf{v}||^2})}. \tag{C.15}$$

After combining Equation C.14 with Equation C.14, we arrive at the 3D Gaussian kernel G_{3D}

$$G_{3D} = G_{1D}\cdot G_{2D} = De^{-(\frac{((\mathbf{x}-\mathbf{c}')\cdot\mathbf{a}_0)^2}{||\mathbf{a}_0||^4}+\frac{((\mathbf{x}-\mathbf{c}')\cdot\mathbf{a}_1)^2}{||\mathbf{a}_1||^4}+\frac{s^2}{||\mathbf{v}||^2})} \tag{C.16}$$

and the corresponding ellipsoid equation

$$\frac{((\mathbf{x}-\mathbf{c}')\cdot\mathbf{a}_0)^2}{||\mathbf{a}_0||^4}+\frac{((\mathbf{x}-\mathbf{c}')\cdot\mathbf{a}_1)^2}{||\mathbf{a}_1||^4}+\frac{s^2}{||\mathbf{v}||^2}\leq 1. \tag{C.17}$$

Here, again, the parameter D has to normalize the Gaussian such that it integrates to 1. If $||\mathbf{v}|| = 0$ or v'_z holds, that is, the velocity is zero or it coincides with the ellipse plane, the ellipsoid degenerates to an ellipse.

C.2. General elliptical splats

Based on the 3D Gaussian kernel we have derived in the previous section, we can now proceed to calculating the axes of the ellipsoid primitive which represents the kernel. We begin with determining the conic matrix Q of the ellipsoid which appears in an alternative representation of the ellipsoid equation, that is, a point $\mathbf{x}$ lies within the ellipsoid if the following condition holds.

$$\mathbf{x}^TQ\mathbf{x} = Ax^2+By^2+Cz^2+2Dxy+2Exz+2Fyz-1\leq 0, \tag{C.18}$$

where the components of the conic matrix are

$$Q = \begin{pmatrix} A & D & E \\ D & B & F \\ E & F & C \end{pmatrix}. \tag{C.19}$$

The ellipsoid axes can now be easily calculated by principal component analysis, that is, we have to perform an Eigenvalue decomposition of the conic matrix Q. Since this matrix is sym-

metric, we can rely on the properties that its Eigenvalues are real and the Eigenvectors form an orthogonal basis. The Eigenvalue decomposition into the Eigenvector matrix V

$$V = \begin{pmatrix} \mathbf{ev}_0 & \mathbf{ev}_1 & \mathbf{ev}_2 \\ & & \end{pmatrix} \tag{C.20}$$

with Eigenvectors $\mathbf{ev}_i$ and Eigenvalue matrix Λ

$$\Lambda = diag\lambda_0, \lambda_1, \lambda_2 \tag{C.21}$$

with the corresponding Eigenvalues λ_i is

$$\begin{aligned} Q &= V\Lambda V^{-1} && \text{(C.22)} \\ &= T^{-T}DT^{-1} && \text{(C.23)} \\ &= TT^{-1}. && \text{(C.24)} \end{aligned}$$

In the last equation, the matrix D determines the type of quadric in the implicit representation. Since we are working with ellipsoids, D equals the identity matrix. Furthermore, the matrix T denotes the variance matrix of the ellipsoid whose columns consists of the ellipsoid axes

$$T = \begin{pmatrix} \mathbf{e}_0 & \mathbf{e}_1 & \mathbf{e}_2 \\ & & \end{pmatrix}. \tag{C.25}$$

This implies that the ellipsoid axes are computed by scaling the Eigenvectors with the square root of their corresponding Eigenvalues

$$\mathbf{e}_i = \sqrt{\lambda_i} \cdot \mathbf{ev}_i. \tag{C.26}$$

Sections 2 and 3 in Chapter 11 of the book [PFTV88] provide source code for a numerical computation of the Eigenvalue decomposition. First, the matrix Q is reduced to a tridiagonal form by the Housholder algorithm. The resulting matrix is then used as an input to the QL algorithm which computes the Eigenvalues and Eigenvectors. We ported the source code to GLSL and limited the number of iterations in the Housholder algorithm to the 5 iterations which are necessary for a three-by-three matrix. However, we could not achieve the expected results and assume that the code overstrains the capabilities of the GPU. If we make the simplifying assumption that the surface splats have circular shape, the ellipsoid axes computation is far less involved and realizable on GPU. The necessary derivations are provided in the following section.

C.3. Simplified case: circular 2D splats

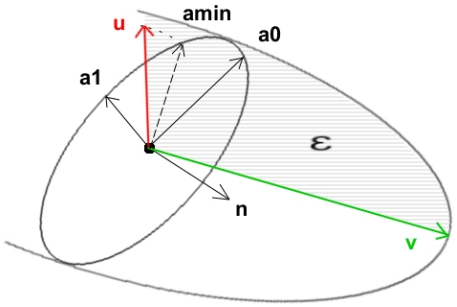

Figure C.3.: *The shortest ellipsoid axis lies in the plane which is spanned by the longest vector in the configuration and the minimal direction in the plane of the other two vectors.*

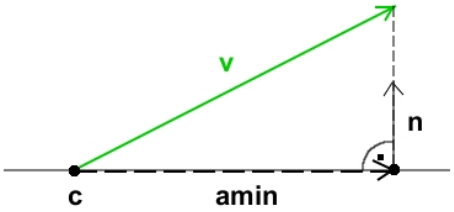

Figure C.4.: *Calculation of the minimum direction.*

In this section, we derive the calculation of the ellipsoid axes in the case of circular 2D surface splats. Note that this derivation does not deliver the accurate shape since we assume that the longest vector in the configuration of the 2D splat axes and the velocity vector equals the longest ellipsoid axis. In fact, the longest ellipsoid axis may tend towards the direction of the longest vector in the configuration, but they are not identical. Consider Figure C.3 where we show the geometrical setting of the 2D surface splat, the velocity vector and the ellipsoid.

In the following derivation, $\mathbf{a}_0$, $\mathbf{a}_1$ and $\mathbf{v}$ are interchangeable and we consider the following examplary configuration:

$$\begin{aligned} ||\mathbf{a}_0|| &= ||\mathbf{a}_1|| = r \\ ||\mathbf{v}|| &> r, \end{aligned} \tag{C.27}$$

where r denotes the splat radius. In this case, the longest ellipsoid axis is chosen to be the velocity vector:

$$\mathbf{e}_0 := \mathbf{v} \tag{C.28}$$

Next we assume that the shortest ellipsoid axis lies in the plane spanned by the velocity vector $\mathbf{v}$ and the vector $\mathbf{a}_{min}$. The latter is planar to the 2D splat and has the smallest incident angle with $\mathbf{v}$. It can be determined by calculating the normal projection of the velocity vector to the 2D splat plane. In other words, the endpoint of $\mathbf{a}_{min}$ is the point in the 2D splat plane which has

the shortest distance to the endpoint of **v**. Since $\mathbf{a}_{min}$ is only used to define the plane in which the shortest ellipsoid axis lies, its length is irrelevant. If we consider the 2D splat plane equation

$$\begin{aligned} Ax + By + Cz + D &= 0 \\ \mathbf{c} := 0 &\rightarrow D = 0 \\ (A, B, C)^T &:= \mathbf{n}, \end{aligned} \tag{C.29}$$

the shortest distance of endpoint of $\mathbf{v} = (v_x, v_y, v_z)^T$ to this plane is

$$dst = \frac{Av_x + Bv_y + Cv_z}{\sqrt{A^2 + B^2 + C^2}} \tag{C.30}$$

$$\rightarrow \mathbf{a}_{min} = \mathbf{v} - dst \cdot \mathbf{n} \tag{C.31}$$

The remaining ellipsoid axes are then chosen as follows:

$$\begin{aligned} \mathbf{e}_1 &:= e_1 \cdot (\mathbf{v} \times \mathbf{a}_{min}) \\ \mathbf{e}_2 &:= e_2 \cdot (\mathbf{e}_0 \times \mathbf{e}_1) \end{aligned} \tag{C.32}$$

The unknowns e_1 and e_0 can be calculated by inserting the splat axis enpoints $\mathbf{a}_0$ and $\mathbf{a}_1$ into the ellipsoid equation . We still need to handle certain degenerate cases where the velocity vector happens to coincide with the 2D splat plane or its length equals zero. This will be described in the following.

C.3.1. Degenerate case: Velocity vector coincides with 2D surface splat plane

If the velocity vector lies in the plane of the 2D surface splat, the ellipsoid degenerates to an ellipse. One option tho handle this is to project the velocity vector to the 2D splat plane and compute the spanning ellipse of this vector and the original 2D splat. The resulting ellipse could then be rendered by switching back to the original EWA surface splatting. However, it would be preferable to apply the same rendering algorithm as with ellipsoid axes. The difficulty which arises here is the necessity of computing the inverse of the ellipsoid variance matrix T. If one of the ellipsoid axes equals zero, this matrix cannot be inverted, however. Therefore, we have to resort to an approximative solution which is described as follows. First, the normal projection of **v** to the 2D splat plane is computed, which we denoted by $\mathbf{v}*$. Depending on the length of $\mathbf{v}*$, two different cases have to be regarded. In the first case, the endpoint of $\mathbf{v}*$ lies within the original 2D splat and the ellipsoid axes are chosen as

$$\begin{aligned} \mathbf{e}_0 &:= \mathbf{a}_0 \\ \mathbf{e}_1 &:= \mathbf{a}_1 \\ \mathbf{e}_2 &:= \epsilon \cdot \mathbf{n}, \end{aligned} \tag{C.33}$$

where ϵ is a sufficiently small scaling parameter while ensuring that the inversion of the variance matrix T is still stable. In the second case, the endpoint of $\mathbf{v}*$ lies outside of the original 2D splat. This makes it necessary to calculate the spanning ellipse of $\mathbf{v}*$ and the original 2D splat. For circular splats, this is straightforward.

$$\begin{aligned} \mathbf{e}_0 &:= \mathbf{v}* \\ \mathbf{e}_1 &:= r \cdot \frac{\mathbf{v}* \times \mathbf{n}}{||\mathbf{v}* \times \mathbf{n}||} \\ \mathbf{e}_2 &:= \epsilon \cdot \mathbf{n}. \end{aligned} \tag{C.34}$$

C.3.2. Degenerate case: Velocity vector length equals zero

For this degenerate case, we choose the 2D surface splat axes and the small scaled splat normal as the ellipsoid axes.

$$\begin{aligned} \mathbf{e}_0 &:= \mathbf{a}_0 \\ \mathbf{e}_1 &:= \mathbf{a}_1 \\ \mathbf{e}_2 &:= \epsilon \cdot \mathbf{n}. \end{aligned} \tag{C.35}$$

C.3.3. Special case: Velocity vector parallel to splat normal

In this case, we can skip the computations of a general configuration as described in Section C.3. The first two ellipsoid axes become the splat axes and the third one is the velocity vector projected to the splat normal:

$$\begin{aligned} \mathbf{e}_0 &:= \mathbf{a}_0 \\ \mathbf{e}_1 &:= \mathbf{a}_1 \\ \mathbf{e}_2 &:= (\mathbf{n} \cdot \mathbf{v}) \cdot \mathbf{n}. \end{aligned} \tag{C.36}$$

C.4. Footprint function of a volumetric kernel

Since Gaussian kernels are closed under projective transformations, the perspective projection of a 3D Gaussian kernel G_{3D} to screen results in a elliptical 2D Gaussian G_{2D} which is denoted

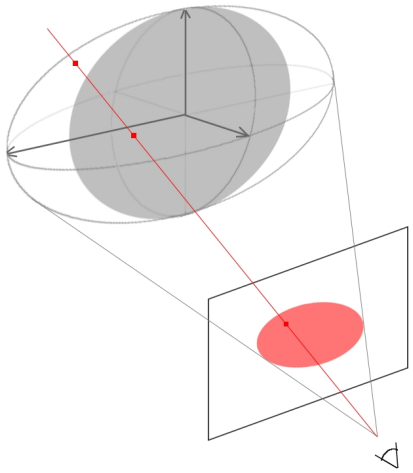

Figure C.5.: *A 3D Gaussian kernel represented by an ellipsoid is intersected with the viewing ray.*

by the term footprint function. To evaluate this function at a certain screen location **x**, we have to calculate the integral along the viewing ray through the screen location (Figure C.5). The ray equation is stated as

$$\mathbf{x}_x = \mathbf{eye} + t\mathbf{dir}_x, \tag{C.37}$$

where $\mathbf{x}_x$ is a point on the viewing ray through the screen location **x**, **eye** is the ray origin, $t \in \mathbb{R}$ is a scaling parameter and $\mathbf{dir}_x$ denotes the viewing direction. Inserting the ray equation into the Gaussian G_{3D} and integrating over t then delivers the required value of the footprint function G_{2D}:

$$G_{2D}(\mathbf{x}) = \int_{-\liminf}^{\liminf} G_{3D}(\mathbf{eye} + t\mathbf{dir}_x). \tag{C.38}$$

Because we work with Gaussians which have only a finite support, the integral reduces to the range between the two intersection points $\mathbf{x}_1$ and $\mathbf{x}_2$ of the ray and the conic isosurface at the cutoff value, that is, the ellipsoid. In our application, the screen space size of a single footprint function is fairly small and exact gradient changes barely perceptible, which means that an approximative value is sufficient. We exploit the fact that the Gaussian value increases with the length of the ray segment between the intersection points. In detail, we transform the intersection points back to parameter space and compute their distance:

$$d = ||(MT)^{-1}\mathbf{x}_1 - (MT)^{-1}\mathbf{x}_2||. \tag{C.39}$$

Since the ellipsoid is a unit sphere in parameter space, this value lies in the range $[0, 1]$ and is plugged into a spherical Gaussian to compute the final value.

C.5. Attribute functions of a volumetric kernel projected to screen

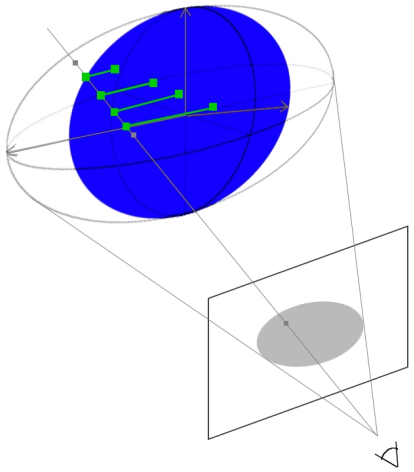

Figure C.6.: *The attribute function value at a certain screen location is determined by sampling along the viewing ray, tracing back the corresponding source points and averaging their attribute values.*

During the ellipsoid rasterization stage, it becomes necessary to evaluate the attribute function, for example color, texture parameter, normal or visibility, at a certain screen location. However, these functions are all defined over the 2D surface splat and not the volumetric kernel. To determine the attribute value of an arbitrary point within the ellipsoid, we have to be aware of the fact about how this point has been generated. That is, the stems from a point on the 2D surface splat which has been convolved with a 1D Gaussian along the velocity vector. This point, which we call the source point, can be calculated as explained in the next Section C.6 and determines the attribute function value of all points along the direction of velocity. If we denote by σ the operator which traces back a point $\mathbf{x}_x$ to its source point, the attribute function value of a volumetric kernel projected to screen becomes

$$h(\mathbf{x}) = \frac{1}{t_f - t_c} \int_{t_c}^{t_f} h(\sigma(\mathbf{eye} + t\mathbf{dir}_x)), \tag{C.40}$$

where t_c and t_f denote the scaling parameters corresponding to the closer and farther intersection point and h is a placeholder for the attribute function. Instead of computing the integral, the intersection line segment can be sampled and the corresponding attribute values averaged (Figure C.6).

$$h(\mathbf{x}) \approx \frac{1}{N} \sum_{t_k} h(\sigma(\mathbf{eye} + t_k\mathbf{dir}_x)). \tag{C.41}$$

Note that the visibility function, which is defined to be binary on the 2D surface splat, now can have an arbitrary value within the interval $[0, 1]$. For our application, the evaluation of

the color function is particularly simple since we assume a constant color value for each 2D surface splat. Visibility is in general not constant over the 2D surface splat area, however, and we determine its value by sampling the intersection segment once at the midpoint of the intersections. Resultingly, we cannot handle splat intersections.

C.6. Source point on 2D splat

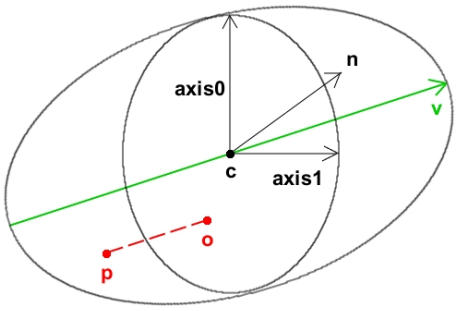

Figure C.7.: *Source point on the original 2D splat.*

This section derives how the source point of a point **x** within the volumetric kernel is determined. By the term of source point we denote a point on the 2D surface splat. In the process of sweeping the 2D splat along the velocity vector by convolving the Gaussian kernel with a 1D Gaussian, this point generates a line. The material properties and visibility of every point on this line is defined by the corresponding value at the source point. In other words, the material properties and visibility of a point within the volumetric kernel is determined by its generating source point on the 2D surface splat. Therefore, we need a way to trace back a point **x** along the velocity vector onto the 2D surface splat. The involved calculations are illustrated in Figure C.7 and described in the remainder of this section.

The problem is posed as a intersection of the 2D splat plane with the line which runs through the point **x** along the direction of the velocity vector **v**. The plane equation is

$$\begin{aligned} \mathbf{c} \cdot \mathbf{n} &= d \\ \mathbf{o} \cdot \mathbf{n} &= d. \end{aligned} \tag{C.42}$$

and the line equation

$$\mathbf{x} + \mathbf{v} \cdot t_o = \mathbf{o} \tag{C.43}$$

To solve for t_0, insert Equation C.42 into Equation C.43:

$$t_o = \frac{d - \mathbf{x} \cdot \mathbf{n}}{\mathbf{v} \cdot \mathbf{n}}. \tag{C.44}$$

The required source point **o** is then calculated by evaluating Equation C.43.

Bibliography

[BFMZ94] Gary Bishop, Henry Fuchs, Leonard McMillan, and Ellen J. Scher Zagier. Frameless rendering: double buffering considered harmful. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 175–176, New York, NY, USA, 1994. ACM.

[BHZK05] Mario Botsch, Alexander Hornung, Matthias Zwicker, and Leif Kobbelt. High-quality surface splatting on today's gpus. In *Symposium on Point-Based Graphics 2005*, pages 17–24, jun 2005.

[BK03] Mario Botsch and Leif Kobbelt. High-quality point-based rendering on modern gpus. In *PG '03: Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, page 335, Washington, DC, USA, 2003. IEEE Computer Society.

[BSK04] Mario Botsch, Michael Spernat, and Leif Kobbelt. Phong splatting. In *In Proc. of symposium on Point-Based Graphics 04*, 2004.

[Bur] John Burkardt. Spline - interpolation and approximation of data.

[BWK02] Mario Botsch, Andreas Wiratanaya, and Leif Kobbelt. Efficient high quality rendering of point sampled geometry. In *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering*, pages 53–64, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[CPC84] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *SIGGRAPH Comput. Graph.*, 18(3):137–145, 1984.

[DWWL05] Abhinav Dayal, Cliff Woolley, Benjamin Watson, and David Luebke. Adaptive frameless rendering. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*,

page 24, New York, NY, USA, 2005. ACM.

[GM04] Xin Guan and Klaus Mueller. Point-based surface rendering with motion blur. In *Eurographics Symposium on Point-based Graphics 2004*, 2004.

[Gra85] Charles W. Grant. Integrated analytic spatial and temporal anti-aliasing for polyhedra in 4-space. *SIGGRAPH Comput. Graph.*, 19(3):79–84, 1985.

[Hec89] Paul S. Heckbert. Fundamentals of texture mapping and image warping. Technical report, Berkeley, CA, USA, 1989.

[JESMM+97] II J. Edward Swan, Klaus Mueller, Torsten Möller, Naeem Shareef, Roger Crawfis, and Roni Yagel. An anti-aliasing technique for splatting. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 197–ff., Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[KB83] Jonathan Korein and Norman Badler. Temporal anti-aliasing in computer generated animation. *SIGGRAPH Comput. Graph.*, 17(3):377–388, 1983.

[KK] Thomas Kollig and Alexander Keller. Efficient multidimensional sampling. pages 557–563.

[ML85] Nelson L. Max and Douglas M. Lerner. A two-and-a-half-d motion-blur algorithm. *SIGGRAPH Comput. Graph.*, 19(3):85–93, 1985.

[MMI+98] Klaus Mueller, Torsten Möller, J. Edward Swan II, Roger Crawfis, Naeem Shareef, and Roni Yagel. Splatting errors and antialiasing. *IEEE Transactions on Visualization and Computer Graphics*, 4(2):178–191, 1998.

[NRS82] Alan Norton, Alyn P. Rockwood, and Philip T. Skolmoski. Clamping: A method of antialiasing textured surfaces by bandwidth limiting in object space. *SIGGRAPH Comput. Graph.*, 16(3):1–8, 1982.

[PC83] Michael Potmesil and Indranil Chakravarty. Modeling motion blur in computer-generated images. *SIGGRAPH Comput. Graph.*, 17(3):389–399, 1983.

[PFTV88] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical recipes in C: the art of scientific computing*. Cambridge University Press, New York, NY, USA, 1988.

[Poi07] *Point-Based Graphics*. Morgan Kaufmann Series in Computer Graphics, jul 2007.

[RL00] Szymon Rusinkiewicz and Marc Levoy. Qsplat: a multiresolution point rendering system for large meshes. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 343–352, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[Shi93] Mikio Shinya. Spatial anti-aliasing for animation sequences with spatio-temporal filtering. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 289–296, New York, NY, USA, 1993. ACM.

[SM03] Peter Shirley and R. Keith Morley. *Realistic Ray Tracing*. A. K. Peters, Ltd.,

Natick, MA, USA, 2003.

[SPW02] Kelvin Sung, Andrew Pearce, and Changyaw Wang. Spatial-temporal antialiasing. *IEEE Transactions on Visualization and Computer Graphics*, 8(2):144–153, 2002.

[SWBG06] Christian Sigg, Tim Weyrich, Mario Botsch, and Markus Gross. Gpu-based ray-casting of quadratic surfaces. pages 59–65, Boston, 2006.

[WZ96] Matthias M. Wloka and Robert C. Zeleznik. Interactive real-time motion blur. *The Visual Computer*, 12(6):283–295, 1996.

[ZPvBG01] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Surface splatting. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 371–378, New York, NY, USA, 2001. ACM.

[ZPvBG02] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Ewa splatting. *IEEE Transactions on Visualization and Computer Graphics*, 08(3):223–238, 2002.

[ZRB+04] Matthias Zwicker, Jussi Räsänen, Mario Botsch, Carsten Dachsbacher, and Mark Pauly. Perspective accurate splatting. In *GI '04: Proceedings of Graphics Interface 2004*, pages 247–254, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. Canadian Human-Computer Communications Society.