Vision, Modeling, and Visualization (2015)
D. Bommes, T. Ritschel, and T. Schultz (Eds.)

Efficient GPU Based Sampling for Scene-Space Video Processing

F. Klose[1], O. Wang[2], J.-C. Bazin[2], M. Magnor[1] A. Sorkine-Hornung[2],

[1]TU Braunschweig
[2]Disney Research Zurich

Abstract

We describe a method to efficiently collect and filter a large set of 2D pixel observations of unstructured 3D points, with applications to scene-space aware *video processing. One of the main challenges in scene-space video processing is to achieve reasonable computation time despite the very large volumes of data, often in the order of billions of pixels. The bottleneck is determining a suitable set of candidate samples used to compute each output video pixel color. These samples are observations of the same 3D point, and must be gathered from a large number of candidate pixels, by volumetric 3D queries in scene-space. Our approach takes advantage of the spatial and temporal continuity inherent to video to greatly reduce the candidate set of samples by solving 3D volumetric queries directly on a series of 2D projections, using out-of-core data streaming and an efficient GPU producer-consumer scheme that maximizes hardware utilization by exploiting memory locality. Our system is capable of processing over a trillion pixel samples, enabling various scene-space video processing applications on full HD video output with hundreds of frames and processing times in the order of a few minutes.*

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Picture/Image Generation— I.3.3 [Computer Graphics]: Parallel processing—

1. Introduction

Recent work has shown that many complex video processing tasks can be effectively formulated in *scene-space* [KWB*15] using a novel sampling based method, as opposed to traditional *image-space* formulations. This approach enables fundamental operations such as denoising, deblurring, and superresolution, as well as more advanced effects such as inpainting, virtual aperture synthesis, computational shutter functions, and 3D action shots on monocular video sequences. Until now, 3D enabled video processing techniques have been limited by the need for exact scene representations such as accurate meshes, which are challenging to obtain in real world application scenarios. Sampling based scene-space processing [KWB*15], on the other hand, employs a sample collection and filtering strategy which is robust to inevitable noisy and erroneous scene information, such as depth maps and camera poses estimated from monocular 2D video or measured by depth sensors.

A key computational challenge is how to efficiently gather an appropriate set of 3D samples (observations of the same scene point), given the considerable amount of data contained in video. For example, a 1080p, 30fps, 30 second long video contains 1.8 billion pixel, and therefore requires 1.8 billion volumetric queries to collect required pixel samples. If each sample gathering step collects roughly 1000 samples for each output pixel, about 1.8 *trillion* samples have to be considered to process an entire video.

In this paper we first provide a brief overview of sampling based scene-space video processing [KWB*15], and then describe in detail how such a framework can be efficiently implemented on a modern GPU. The basic approach is divided into two steps. First, a general purpose gathering step collects for each output pixel, the complete set of potential observations of the same scene point. Then, an application-specific filtering step computes a weighted sum of these samples to produce the final output pixel color.

While the work by [KWB*15] introduces the basic, high-level concept of sampling based scene-space video processing and its applications, the goal of this work is a full and detailed description of how one can implement sampling

DOI: 10.2312/vmv.20151264

based scene-space video processing *efficiently*. Given the huge amount of data processed, making this concept work efficiently in practice requires a carefully designed technical solution. The contributions of this work include a dedicated producer-consumer scheme (Section 4), designed to maximize data parallelism and memory locality and thereby optimize GPU thread coherence. Additionally, it minimizes compute kernels, leading to more efficient register and device compute utilization. Another technical contribution of this work is an out-of-core data tiling and streaming approach (Section 5) which allows the method to scale to arbitrary input/output resolutions and video lengths by bounding the maximum memory usage. This is an essential property for processing high resolution videos as well as compatibility on GPU devices with limited memory. In addition to the algorithmic description, we provide a detailed investigation of memory and runtime performance and conduct a scalability analysis with respect to GPU memory.

2. Related Work

Our paper describes in detail the approach of Klose et al. [KWB*15], and a full overview of scene-space processing related work can be found there. Here, we concentrate on prior work concerning acceleration techniques for similar problems.

Nearest neighbor search At the heart of our method is a fast out-of-core volume query implemented on the GPU using a series of projections. In this sense, our approach is a specific case of general nearest-neighbor search methods. Common approaches for this kind of query are KD-trees [Ben75], locality sensitive hashing [DIIM04], or product quantization [JDS11]. Some of these techniques have been also deployed on GPU, for example efficient hashing [AVS*11, ASA*09, GLHL11]. These approaches are general purpose acceleration techniques, but are typically designed for querying nearest neighbors using some metric distance function, rather than directly querying *convex volumes*. Additionally, we are interested in datasets containing over a billion points, and perform over a billion queries on these datasets. Most of these approaches are not designed for tasks that require this quantity of data. As opposed to these methods, we propose a volumetric query technique dedicated to (unstructured) 3D points obtained from *structured* 2D observations (image points), which allows us to achieve higher performance.

A recent GPU-accelerated approach by Tsai et al. [TSPP14] describes how image structure can be used to reduce the query space for an approximate nearest neighbor search. Rapid queries can then be used for example, for image denoising by NL-means [BCM05]. Their method outperforms existing nearest neighbor techniques such as [GDNB10, SH08]. We perform at the same speed as [TSPP14], but provide a more general query framework that operates on significantly larger amounts of data. Additionally, Tsai et al. [TSPP14] gain performance by computing approximate nearest neighbor, whereas our method computes exact nearest neighbors.

Techniques for out-of-core proximity tests such as Kim et al. [KSK*14] can handle point clouds in the order of tens of millions. They perform spherical neighborhood queries based on a working-block grid maintained by multiple CPUs and handling local NN-queries on the GPU. The overall amount of input data for video makes the use of existing GPU accelerated nearest neighbor algorithms difficult, since any precomputed data structure would have to be streamed in and out of the graphics memory. Our approach is designed to be a general out-of-core spatial query framework, allowing us to handle long and high resolution video sequences.

Object intersection Computing object intersections is a very common task in computer graphics. Many techniques have been proposed, for example in the context of object collision [PKS10, LG07] and to detect whether a point lies in a 2D polygon or 3D volume [Gla90]. While these techniques are efficient, they are general purpose. In contrast, we can take advantage of the fact that our (unstructured) 3D points come from structured 2D observations, which allows us to more quickly remove a large number of samples lying outside the query frustum volume in an early step, and then efficiently check the remaining samples.

Point-based methods Similar to our approach, point-based rendering techniques deal with large numbers of unstructured points, mostly for rendering virtual view points. These methods work by "splatting" unstructured, oriented point clouds (e.g., acquired by laser scans) into a virtual camera view for displaying complex scenes [RL00, ZPvBG01]. Similar to our approach, they accumulate surface samples in screen space. QSplat [RL00] uses a bounding volume hierarchy to accelerate queries and a workstation renders between 1.5 to 2.5 million points per second (in the year 2000). Later work significantly increased the number of rendered points using GPU-based implementations for surface splatting [RPZ02]. For instance, using deferred shading techniques, Botsch et al. [BHZK05] render up to 20M elliptical splats per second on a GPU. These point-based methods are designed to render unstructured point clouds and can use straightforward depth-based visibility tests to discard hidden samples. Our 3D information is noisy therefore we cannot make similar assumptions. In the context of geometry fusion from multiple RGB-D cameras, Kuster et al. [KBÖ*14] perform 3D neighborhood queries by projecting the query 3D point onto all the cameras and collecting the points within an area in the 2D domain. In contrast we aim for volumetric queries in a defined 3D frustum.

3. Overview of scene-space video processing

The idea behind sampling based scene-space video processing [KWB*15] is to gather and process all potential observations of the same scene point. Consider all pixels in a video

```
foreach pixel p of output image O do S(p) ← ∅
foreach input image I^f do
    foreach pixel p of output image O do
        Construct 3D query shape V(p)
        foreach pixel q of I^f do
            s_xyz ← compute the 3D position of q from
            its depth value and the camera matrix C^f
            if s_xyz inside V(p) then
                s_rgb ← I^f(q)
                s_t ← f
                S(p) ← S(p) ∪ {(s_xyz, s_rgb, s_t)}
            end
        end
    end
end
foreach pixel p of O do
    O(p) ← compute output pixel color from S(p)
end
```

Algorithm 1: Outline of a naive implementation of scene space processing for one output image $\boldsymbol{O}$. Note that the loop over all output image pixels can be executed independently and in parallel.

projected into an unstructured 3D point cloud; the viewing frustum of one pixel from an output frame contains all of these possible samples. However, due to errors in 3D as well as occlusions, some samples may correspond to different objects, and some correct observations may be missing. A subsequent filtering step therefore removes the contribution of these erroneous samples to the final output color. More specifically, the input to the method is a monocular video with known camera pose parameters and per-frame depth maps. A pixel p of frame f in the input video $\boldsymbol{I}$ is written as $\boldsymbol{I}^f(p)$. Our goal is to compute the colors of all output pixels $\boldsymbol{O}^g(p)$ for all output frames g. For each $\boldsymbol{O}^g(p)$, we draw a set of samples $\boldsymbol{S}^g(p)$ directly from the input video $\boldsymbol{I}$ based on a 3D query volume $\boldsymbol{V}^g(p)$. The query shape $\boldsymbol{V}^g(p)$ is constructed for each $\boldsymbol{O}^g(p)$ individually. The most common query shape for our application is the view frustum of pixel $\boldsymbol{O}^g(p)$, as this frustum is the volume of all scene samples that could be potentially observed by $\boldsymbol{O}^g(p)$. However, our method could be used to compute volumetric queries of any convex shape. In the following we drop the index g for clarity when considering sequentially processed output frames.

Each of these selected *samples*, $s \in \mathbb{R}^7$, is composed of color ($s_{rgb} \in \mathbb{R}^3$), scene-space position ($s_{xyz} \in \mathbb{R}^3$), and frame time ($s_t \in \mathbb{R}$). We denote the camera matrix used to project a pixel from frame f together with its depth into scene-space as C^f. The final output color of $\boldsymbol{O}^g(p)$ is then determined from $\boldsymbol{S}^g(p)$ in a filtering operation based on a weighted sum of these input samples:

$$\boldsymbol{O}^g(p) = \frac{1}{\boldsymbol{W}} \sum_{s \in \boldsymbol{S}^g(p)} w(s) s_{rgb} \tag{1}$$

where $w(s)$ is an application specific weighting function and $\boldsymbol{W}$ is the sum of all $w(s)$ used for normalization. In order to determine which samples in $\boldsymbol{S}^g(p)$ are reliable, we rely heavily on the concept of a *reference* sample (s_{ref}). Because our task is to process an existing video, we can, in most cases, use the sample derived from projecting the input pixel $\boldsymbol{I}^f(p)$ into scene-space as a reference. Where such a reference exists, weights can be computed based on the distance of each sample to this reference, similar to the patch center in bilateral filtering. For example, a weighting function suitable for video denoising is:

$$w_{denoise}(s) = \exp\left(-\frac{(s_{ref} - s)^2}{2\sigma^2}\right). \tag{2}$$

We use the above notation for clarity, while samples are actually represented in a 7D space and we use a diagonal covariance matrix σ. We call the diagonal entries σ_{rgb} for the three color dimensions, σ_{xyz} for the scene-space position and σ_f for the frame time. The weight falls off exponentially with distance to the reference sample s_{ref}. For more application specific weighting functions we refer the reader to the original work [KWB*15].

While the naive sample checking of Algorithm 1 outlines the general principle very well and gives insight into the scale of the problem to be solved, it is not a practical solution for scene-space video processing. Prior work [KWB*15] proposed a technique to take advantage of the continuity of image data to quickly reject a large number of samples by checking whether they fall in the 2D projections of the 3D query frustum $\boldsymbol{V}(p)$. One of the advantages of this approach is that it operates directly in the image domain, avoiding any intermediary, explicit 3D representation. The performance gain of this step is a function of the camera motion and scene structure. Algorithm 2 illustrates the modified algorithm only checking the samples within the convex bounds of the projected query shape. We extend this concept and introduce a producer-consumer scheme with out-of-core data streaming. In cases where the filtering functions perform a weighting based on the frame number, we can further reduce the set of tested samples to frames that lie within $3\sigma_f$ of the current reference frame.

Please note that the image domain projection and testing does not restrict the maximum number of samples tested. It is in fact a necessary step in setting up the problem in such a way, that we can take full advantage of the parallel GPU processing power in our producer/consumer scheme.

4. Producer-Consumer Model

While the sample selection method above greatly reduces computation, it is still not sufficient for practical running times. As memory transfers and access are the major bottlenecks when performing out-of-core GPU computing, we break the sampling process into a producer and a consumer

foreach *pixel p of output image* $\boldsymbol{O}$ **do** $\boldsymbol{S}(p) \leftarrow \emptyset$
foreach *input image* $\boldsymbol{I}^f$ **do**
  foreach *pixel p of output image* $\boldsymbol{O}$ **do**
    Construct 3D query shape $\boldsymbol{V}(p)$
    $\boldsymbol{U}(p) \leftarrow$ project vertices of $\boldsymbol{V}(p)$ into view $\boldsymbol{I}^f$
    $\boldsymbol{V}_O(p) \leftarrow$ compute the convex hull of $\boldsymbol{U}(p)$
    foreach *pixel q of* $\boldsymbol{I}^f$ *inside* $\boldsymbol{V}_O(p)$ **do**
      $s_{xyz} \leftarrow$ compute the 3D position of q from its depth value and the camera matrix of $\boldsymbol{C}^f$
      if s_{xyz} *inside* $\boldsymbol{V}(p)$ **then**
        $s_{rgb} \leftarrow \boldsymbol{I}^f(q)$
        $s_t \leftarrow f$
        $\boldsymbol{S}(p) \leftarrow \boldsymbol{S}(p) \cup \{(s_{xyz}, s_{rgb}, s_t)\}$
      end
    end
  end
end
foreach *pixel p of* $\boldsymbol{O}$ **do**
  $\boldsymbol{O}(p) \leftarrow$ compute output pixel color from $\boldsymbol{S}(p)$
end

Algorithm 2: Outline scene-space processing for one output image $\boldsymbol{O}$, incorporating the reduced sample tests based on convex hulls.

step. The reason for splitting is twofold; first, we can compute the required data in the producer and then only upload the needed amount on-demand before we run consumer kernels to collect the samples, thereby limiting memory transfers (see Section 5). Second, by splitting the process we get two smaller kernels with better block coherency and less resource and register usage to maximize occupancy on the compute device.

The producer computes a 2D polygon outline of the area of sample candidates that need to be checked, and writes line segments into a per output pixel segment storage, for each pixel row the polygon covers. The consumer kernel then reads these line segments and performs acceptance tests for all pixels on the line. This process is described in Algorithm 3, where $\boldsymbol{R}(p)$ is the segment storage. The bounding boxes of the convex hulls computed by the producer kernel are used to upload the required input color and depth data from the CPU to the GPU. The consumer kernels then only operate on local data areas, which is beneficial for memory locality and therefore efficient caching.

4.1. Producer

The producer kernel is responsible for computing and storing line segments that fall under the convex hull of the projected query volume vertices. The created row line segments need to cover the entire projected 2D polygon created from the 3D query shape for each output pixel. Each line segment is aligned with one row y and is described by its start and end x-coordinate $(y, x_{\text{left}}, x_{\text{right}})$. Since the storage only holds

foreach *pixel p of output image* $\boldsymbol{O}$ **do** $\boldsymbol{S}(p) \leftarrow \emptyset$
foreach *input image* $\boldsymbol{I}^f$ **do**
  foreach *pixel p of output image* $\boldsymbol{O}$ **do**
    Construct 3D query shape $\boldsymbol{V}(p)$
    $\boldsymbol{U}(p) \leftarrow$ project vertices of $\boldsymbol{V}(p)$ into view $\boldsymbol{I}^f$
    $\boldsymbol{V}_O(p) \leftarrow$ compute the convex hull of $\boldsymbol{U}(p)$
    $\boldsymbol{R}(p) \leftarrow \emptyset$
    foreach *row y in* $\boldsymbol{V}_O(p)$ **do**
      compute x_{left} and x_{right}
      $\boldsymbol{R}(p) \leftarrow \boldsymbol{R}(p) \cup \{(y, x_{\text{left}}, x_{\text{right}})\}$
    end
    foreach *line segment* $(y, x_{left}, x_{right}) \in \boldsymbol{R}_p$ **do**
      for $x = x_{left}$ **to** x_{right} **do**
        $s_{xyz} \leftarrow$ compute the 3D position of $q = (x, y)$ from its depth value and the camera matrix of $\boldsymbol{C}^f$
        if s_{xyz} *inside* $\boldsymbol{V}(p)$ **then**
          $s_{rgb} \leftarrow \boldsymbol{I}^f(q)$
          $s_t \leftarrow f$
          $\boldsymbol{S}(p) \leftarrow \boldsymbol{S}(p) \cup \{(s_{xyz}, s_{rgb}, s_t)\}$
        end
      end
    end
  end
end
foreach *pixel p of* $\boldsymbol{O}$ **do**
  $\boldsymbol{O}(p) \leftarrow$ compute output pixel color from $\boldsymbol{S}(p)$
end

Algorithm 3: Outline scene-space processing for one output image $\boldsymbol{O}$, incorporating the reduced sample tests based on convex hulls and splitting the inner part into a line segment producer and a consumer.

line segments for one input frame at a time, this six bytes (2 bytes per value) representation is sufficient to unambiguously describe the raster lines of the polygon.

To create the 2D convex hull we use an augmented version of the Jarvis March (JM) convex hull computation [Jar73] that jointly computes the convex hull and polygon edge tables for rasterization. The JM algorithm is also known as gift wrapping, because it starts at one vertex and adds vertices to the convex hull polygon in a defined order. This enables us to keep track of a left and right edge list for the polygon rasterization step later, while computing the convex hull. We start from the lowest left coordinate in $\boldsymbol{U}(p)$ and add vertices to the convex hull in a counter-clockwise manner. Starting from the first vertex, all edges of the hull polygon that have an increasing y-coordinate belong to the left edge list T_l, and any edge after the tipping point to the right edge list T_r. This approach for computing a convex hull is well suited for GPU implementation, as it does not require additional data structures to be maintained. Since we compute a convex hull for each output pixel in every query into an input image, even small memory overhead has a huge impact.

$T_l \leftarrow$ Left edge table
$T_r \leftarrow$ Right edge table
$e_l \leftarrow 0$
$e_r \leftarrow 0$
for $y = y_{min}$ *to* y_{max} **do**
  $e_l \leftarrow$ increment to edge containing y
  $e_r \leftarrow$ increment to edge containing y
  $x_l \leftarrow$ evaluate edge $T_l(e_l)$ at y
  $x_r \leftarrow$ evaluate edge $T_r(e_r)$ at y
  output (y, x_l, x_r)
end

Algorithm 4: Polygon row line segment creation as used in the producer kernel.

While computing the convex hull we also keep track of the upper and lower bound of the polygon $y_{\max}$ and $y_{\min}$, respectively. After computing the left and right edge lists, the producer iterates over all rows spanned by the polygonal hull. We keep two indices into the left and right edge table, evaluating the edges at that index for the current y coordinate (see Algorithm 4).

Since line segment data has to be stored for each individual output pixel, it is necessary to constrain the number of rows stored. We define a fixed maximum of stored segments $r_{\max} = 512$ that the producer kernel will create in one call. At the end of each producer run we store the y_{last} row for each output pixel that was written and call the consumer kernel to process the line segments. Upon the next producer kernel launch, we continue from that row y_{last} until we reach $y_{\max}$ and terminate. The number of alternating producer and consumer calls can be computed from $ceiling((y_{\max} - y_{\min})/r_{\max})$. The required iteration count depends on the number of rows in the projected 2D polygon of $\boldsymbol{V}(p)$. For small camera motions in the horizontal direction, usually only one or two producer-consumer iterations are needed. For diagonal or upward camera motions, the worst case iteration count is the input image height divided by $r_{\max}$. Since the increased kernel launch counts introduce a small overhead, and the producer kernel has to recreate the 2D polygon to the point where it stopped computing, it is desirable to choose $r_{\max}$ as large as the GPU memory allows.

4.2. Consumer

The consumer kernel determines which samples actually fall into the query volume. One CUDA kernel is run per output pixel, which reads line segments stored by the producer. It loops over all pixel coordinates (x, y) with $x \in [x_{\min}, x_{\max}]$ of defined by the line segment. After loading the z component from the input $\boldsymbol{I}^f$ at (x, y) it computes the sample candidates scene-space position $s_{\text{xyz}} = \boldsymbol{C}^f \cdot (x, y, z)^T$ with the current input frames camera matrix $\boldsymbol{C}^f$.

The fact that the current pixel is checked by the consumer, is evidence of it falling into the convex hull of the projected query volume $\boldsymbol{V}(p)$. This is a necessary condition for the sample s_{xyz} falling into $\boldsymbol{V}(p)$. For simple convex geometries thresholding the distance of s_{xyz} to the center of the query shape is sufficient, resulting in a box for L_1 distance, and a sphere for L_2. Since we want to query a frustum shape, we reproject the sample s_{xyz} into the current output view, and accept the sample based in its 2D distance to the current query point p and depth value. Note that there are highly efficient ways to perform inside tests for 3D points on general convex shapes and we refer to the related work in Section 2 for further reading. If the final inside test is passed, the sample candidate is added to the sample set $\boldsymbol{S}(p)$.

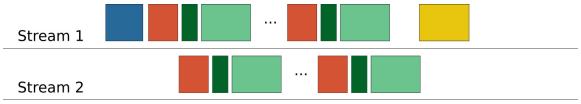

Figure 1: Two parallel streams are executing the producer (dark green) and consumer (light green) computations, entirely hiding the input data upload (red). Before the first upload in stream one, the required input data bounds are computed (blue) and the filtering (yellow) takes place after the last producer-consumer block has finished.

5. Out-of-Core Data Streaming

Up to this point, we explained how the sample gathering step loops over the input frames for each output frame. As the input can potentially be hundreds of frames, the resulting data is often in the order of multiple gigabytes and cannot fit entirely on the limited memory available on current GPUs. With the demand for video processing of ever larger resolutions this will become even more of a problem in the future. We therefore do not upload the entire input video, but rather stream the input data one frame at a time into GPU memory.

Looking at the memory requirements for the collected sample sets reveals a similar observation. Given 13 bytes per sample (3 rgb + 4 depth + 4 xy-position + 2 time) and 1000 samples to store, an estimated 11.9 gigabytes of sample storage is necessary to hold the samples for a single output frame at 1280×720 pixels resolution. Note that we store the sample position s_{xyz} implicitly in form of its 2D image location xy and its depth value from the depth map, resulting in 8 bytes of storage, rather than 12 bytes required for 3 floating point values. The full scene-space position can be recomputed at any time using the sample 2D location, depth and camera matrix of the samples input frame s_t.

In order to handle arbitrary output resolutions, we split the output image into multiple tiles. By performing a producer dry-run (as in Section 4) to compute bounding boxes on each respective input frame, we can conservatively estimate the input pixels required for the current output tile. Uploading only the input pixels that will be used by the current output tile, limits the memory transfer times for each input frame. It is still desirable to have fewer output tiles, since re-

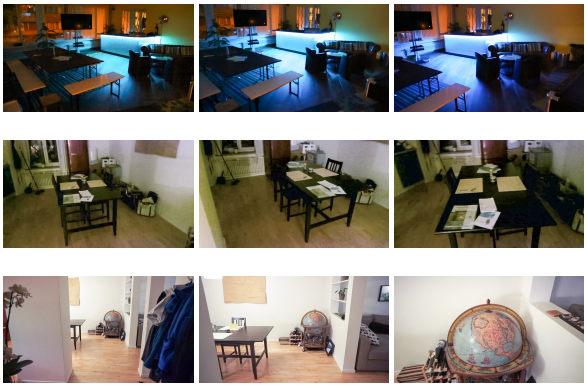

Figure 2: Three frames from each input sequence. (top) OFFICE dataset featuring a mostly rotation camera motion, (mid) KITCHEN dataset with predominant left to right camera motion, (bottom) GLOBE dataset with a forward motion.

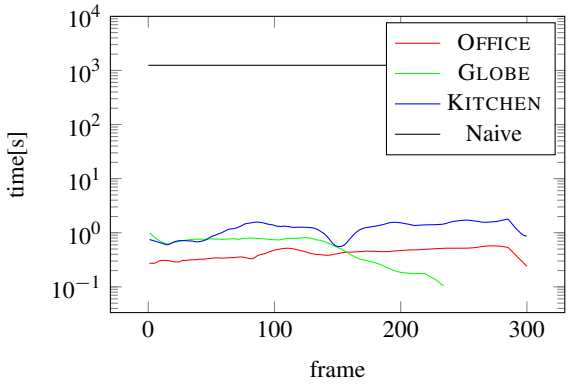

Figure 3: Sample gathering times per frame, for different datasets. The per-frame computation time for the GPU version of the naive solution does not vary with the dataset. We are able to outperform the baseline by three orders of magnitude. The variance within as well as in between the datasets stems from the impact camera motion has on the projected shape of the query volumes in other input frames.

dundant uploads appear in between output tiles that require overlapping input data. The size of the tiling is mainly governed by the available GPU memory. While larger tiles result in faster processing, the approach can be run on GPUs with less memory available by using smaller tiles. The tiling of the output has the additional benefit of resulting in a natural way to distribute the algorithm onto multiple GPUs, where each GPU processes one of the independent output tiles.

To achieve maximum resource usage, we run two CUDA streams in parallel, processing producer-consumer collections on two input frames in parallel. The general kernel execution scheme is illustrated in Figure 1. This interleaved process enables us to hide the upload of input data behind computation, as well as to make use of parallel kernel execution whenever the GPU resources permit.

Please note that, for applications requiring simple weighted sums, it would be possible to accumulate a running sum instead of explicitly collecting the sample set $\boldsymbol{S}(p)$. However, to harness the full versatility of scene-space sample processing for other tasks such as video inpainting or occlusion reasoning within the sample sets, the collection of the entire sample sets is required. If we disable the explicit storage of sample sets, we observe an average performance gain of 15%. The computational bottleneck of the algorithm lies not with writing the accepted samples, but rather testing the sample candidates.

To optimize the memory access when storing and reading the sample sets we store the output sample tile grouped by sample count first. Therefore every thread storing the first sample results in coalesced writes. Although the per output pixel sample set size is bound to diverge over querying multiple input frames, in practice this access order results in a high degree of aligned writes due to spatially coherent structure inherent in the color and depth data.

6. Memory and Timing Analysis

We implemented our approach using CUDA on a NVIDIA GeForce GTX 980 with 4GB of GPU memory. The host code is written in Python and run on a desktop Intel(R) Core(TM) i7 CPU. The datasets for which we provide timings are videos processed at 720p resolution.

One of the advantages of our out-of-core processing is that it can easily scale to accomodate large memory consumption requirements. The most important factor determining the GPU memory footprint is the out-of-core sample tile size u_t. To get better insight of the scalability with respect to memory we describe the required memory usage in Table 1 in two separate parts. The first part (Table 1, rows 1-5) shows reference image data and the currently processed frame, as well as the resulting image and camera parameters. These rows constitute the small fixed memory requirement of our algorithm that is independent of the output sample tile size. For 720p resolution this data is roughly 30 megabytes in size. The second part stores intermediary data such as line segments as well as the collected samples. These scale mostly with the output sample tile size and constitutes the largest part of the GPU memory (Table 1, rows 6-10). The choice of the tile size influences execution time and for best performance on our hardware (NVidia GeForce GTX 980), we choose the output tile size to be $u_t = 512 \times 384$ pixels, such that our input frames fall into six tiles at a memory requirement of 3.6 GB. By reducing the sample tile size to 128×128, we observe that the required memory drops drastically to 304 MB. This hugely reduced footprint comes at a moderate computational cost, i.e. the average per frame computation time for the KITCHEN sequence increases from 1.213 seconds to 1.424 seconds (+17%). The increased com-

	COMPONENTS	BYTES PER ELEMENT	MEMORY u_{t_1} [KB]	MEMORY u_{t_2} [KB]
Reference Frame RGB-D	u_i	3+4	6152	6152
Camera Matrices	$n \times 3 \times 4$	4	46	46
Precomputed Bounding Boxes	$n \times 4$	4	15	15
Input RGBD	$u_i \times 2$ streams	3+4	12 304	12 304
Result	$u_o \times 4$	4	14 063	14 063
Sample Count	u_t	4	63	750
Tile Storage	$u_t \times u_s$	13	208 000	2 496 000
Segment Storage	$u_r \times r_{\max} \times 2$ streams	6	96 000	1 152 000
Producer State	$u_t \times 2$ streams	2	62	750
TOTAL MEMORY REQUIRED			304 125	3 649 500

Table 1: Detailed overview of the GPU memory consumption. Numbers are computed for input and output resolutions of $u_i = u_o = 1280 \times 720$ in a sequence of $n = 1000$ frames with $u_s = 1024$ samples per pixel. In some cases, two CUDA streams are used for better parallelism considering memory transfer, which results in twice the memory consumption but faster processing. We also show the memory usage with different output tile sizes $u_{t_1} = 128 \times 128$ and $u_{t_2} = 512 \times 384$.

	TIME [MS]	TIME %
precompute upload	93	2.9
producer kernel	527	16.5
consumer	2464	76.6
output color computation	128	4.0

Table 2: Average per-frame runtime distribution over individual kernel calls computed over a 100 frame, 1280×720 input video. Note that the memory upload is entirely hidden by computation and therefore does not appear in this table. The majority of the time is spent on checking candidate samples in the consumer kernel.

putational cost stems from the overhead of the out-of-core streaming.

To illustrate the time consumption of individual kernels, we present the runtime distribution of one frame in Table 2. The timings shown are averages computed over a 100 frame sequence, where each output frame used all 100 input frames to gather samples. The majority of the time is spent in the projection steps and inside tests of the consumer kernel. The kernel launch configurations for all algorithms and timings is one GPU thread per output tile pixel with 16×16 threads launched concurrently per block. For each input frame Algorithm 1 and 2 each launch one monolithic kernel performing the projection, sample checking and collection. Algorithm 3 alternately launches producer and consumer kernels multiple times, that only project frustums or check samples respectively. The alternation of kernel launches is stopped, once the producer kernel does no longer create new line segments to be checked by the consumer kernel. Although the number of samples to be checked varies with camera motion, the coherence in number of sample checks between different output pixels within one frame is high. This fact can be observed in the low average divergence of 5 % within thread blocks of the consumer kernel.

The technical specifications of the NVIDIA GTX 980 state the theoretical limit for floating point operations per second (Flops) to 5.632 TFlops. Using the NVIDIA profiler we measure an average of 5.1 TFlops for our produce-consumer algorithm implementation. While the difference that is lost in program structure overhead and memory/cache latencies could be mitigated even further in a CPU implementation, our reported performance exceeds that of a modern day CPU by a factor of 50 (Intel(R) Broadwell i7-5557U: 32 FLOP/cycle at 3.4 GHz $\approx$ 0.1 TFlops).

The final filtering operations are single or multiple linear passes over the gathered sample sets. By storing the samples indexed by count first, we obtain a high memory transfer utilization due to coalesced reads by all neighboring output pixel threads in the final filter call.

We demonstrate the performance of the presented approach on three datasets shown in Figure 2. Since the size of the convex hull of the projected query shape is dependent on the camera motion in the input data, we choose three real-world sequences featuring typical camera motions. The OFFICE dataset is filmed with a mostly rotational camera motion, whereas the KITCHEN dataset has a mostly translational left to right sideways motion. The third dataset is called GLOBE and shows a translational forward motion into the scene. The OFFICE and KITCHEN dataset consist of 300 input frames, and the GLOBE dataset has 234 frames.

Table 3 shows the overall average per output frame computation times. We give timings for the fully GPU parallelized naive implementation (Algorithm 1), the variant using the convex hull projection method (Algorithm 2) and the fully optimized producer-consumer version (Algorithm 3). All three variants are executed within the same framework and identical launch configurations. The producer-consumer

	NAIVE	CB	CB + PC
KITCHEN	1244	1.843	1.213
GLOBE	1244	0.827	0.568
OFFICE	1244	0.535	0.427

Table 3: Running time in seconds with various parts of our method. NAIVE is the naive, fully parallelized solution, CB is the solution with convex bounds, and CB+PC is the full method, including the producer-consumer model.

implementation performs up to 1.5 times faster than the monolithic kernel version and up to 2913 times faster than the naive method.

To give an insight into the computation time variation within and in between sequences, we show the per frame timings for the individual sequences in Figure 3. We can see that a rotational movement of the camera, as featured in the OFFICE dataset, results in faster per frame computation times, since the projected query volume remains small due to smaller parallax. On the other hand the translational motion in the GLOBE and KITCHEN datasets results in higher per frame runtime. Note that the dip around frame 150 in the graph of the KITCHEN dataset (blue curve), is a point in the video where the camera moves slower, resulting in lower per frame computation time. The KITCHEN sequence exhibits the most timing variation due to camera motion. Per frame timings vary around the mean of 1.213 seconds, from 0.55 seconds to 1.78 seconds with a standard deviation of 0.35.

7. Conclusion

We have described in detail a general-purpose framework for parallel queries of 3D volumes for large amounts of RGB-D video data. Due to the flexibility of sampling based scene-space effects, there is a variety of potential future applications that can leverage the presented methods. By presenting the technical details required to achieve reasonable runtimes, we hope to encourage future development in this direction.

As with many GPU based applications, our algorithm can directly benefit from future hardware advances, particularly larger onboard memory and wider memory interfaces. Exploring the capabilities and applications on small scale graphics units (as can be found in mobile devices) will be an additional interesting area for further study. We also hope to extend our work to dynamic scenes in the future. To accomplish this the query mechanism of our algorithm would need to be made aware of scene motion. This can then enable spatio-temporal queries of all points residing in or passing through a spatio-temporal neighborhood.

References

[ASA*09] ALCANTARA D. A., SHARF A., ABBASINEJAD F., SENGUPTA S., MITZENMACHER M., OWENS J. D., AMENTA N.: Real-time parallel hashing on the GPU. *ACM TOG (SIGGRAPH Asia) 28*, 5 (2009). 2

[AVS*11] ALCANTARA D. A., VOLKOV V., SENGUPTA S., MITZENMACHER M., OWENS J. D., AMENTA N.: Building an efficient hash table on the GPU. In *GPU Computing Gems*, vol. 2. Morgan Kaufmann, 2011, ch. 4, pp. 39–53. 2

[BCM05] BUADES A., COLL B., MOREL J.: A non-local algorithm for image denoising. In *CVPR* (2005), pp. 60–65. 2

[Ben75] BENTLEY J. L.: Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9 (1975), 509–517. 2

[BHZK05] BOTSCH M., HORNUNG A., ZWICKER M., KOBBELT L.: High-quality surface splatting on today's GPUs. In *Symposium on Point Based Graphics* (2005), pp. 17–24. 2

[DIIM04] DATAR M., IMMORLICA N., INDYK P., MIRROKNI V. S.: Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the ACM Symposium on Computational Geometry* (2004), pp. 253–262. 2

[GDNB10] GARCIA V., DEBREUVE E., NIELSEN F., BARLAUD M.: K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching. In *ICIP* (2010), pp. 3757–3760. 2

[Gla90] GLASSNER A.: *Graphics Gems*. Academic Press, '90. 2

[GLHL11] GARCÍA I., LEFEBVRE S., HORNUS S., LASRAM A.: Coherent parallel hashing. *ACM TOG (SIGGRAPH Asia) 30*, 6 (2011), 161. 2

[Jar73] JARVIS R. A.: On the identification of the convex hull of a finite set of points in the plane. *Inf. Process. Lett. 2*, 1 (1973), 18–21. 4

[JDS11] JÉGOU H., DOUZE M., SCHMID C.: Product quantization for nearest neighbor search. *IEEE PAMI 33*, 1 (2011), 117–128. 2

[KBÖ*14] KUSTER C., BAZIN J.-C., ÖZTIRELI A. C., DENG T., MARTIN T., POPA T., GROSS M.: Spatio-temporal geometry fusion for multiple hybrid cameras using moving least squares surfaces. *Eurographics 33*, 2 (2014), 1–10. 2

[KSK*14] KIM D., SON M.-B., KIM Y. J., HONG J.-M., EUI YOON S.: Out-of-core proximity computation for particle-based fluid simulations. In *HPG* (2014), pp. 1–9. 2

[KWB*15] KLOSE F., WANG O., BAZIN J. C., MAGNOR M. A., SORKINE-HORNUNG A.: Sampling based scene-space video processing. *ACM Trans. Graph. 34*, 4 (2015), 67. 1, 2, 3

[LG07] LE GRAND S.: Broad-phase collision detection with CUDA. In *GPU Gems 3*. 2007, ch. 32. 2

[PKS10] PABST S., KOCH A., STRASSER W.: Fast and scalable CPU/GPU collision detection for rigid and deformable surfaces. *CGF (Symposium on Geometry Processing) 29*, 5 (2010). 2

[RL00] RUSINKIEWICZ S., LEVOY M.: QSplat: a multiresolution point rendering system for large meshes. In *SIGGRAPH* (2000), pp. 343–352. 2

[RPZ02] REN L., PFISTER H., ZWICKER M.: Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering. *Eurographics 21*, 3 (2002), 461–470. 2

[SH08] SILPA-ANAN C., HARTLEY R.: Optimised KD-trees for fast image descriptor matching. In *CVPR* (2008), pp. 1–8. 2

[TSPP14] TSAI Y., STEINBERGER M., PAJAK D., PULLI K.: Fast ANN for high-quality collaborative filtering. In *HPG* (2014), pp. 61–70. 2

[ZPvBG01] ZWICKER M., PFISTER H., VAN BAAR J., GROSS M.: Surface splatting. In *SIGGRAPH* (2001), pp. 371–378. 2