Eurographics Symposium on Parallel Graphics and Visualization (2012)
H. Childs, T. Kuhlen, and F. Marton (Editors)

Auto Splats: Dynamic Point Cloud Visualization on the GPU

Reinhold Preiner Stefan Jeschke Michael Wimmer

Institute of Computer Graphics and Algorithms, Vienna University of Technology, Austria

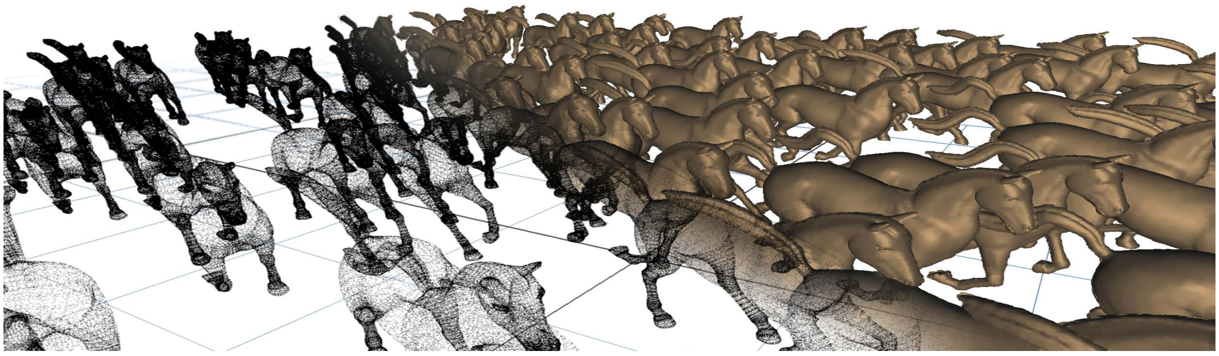

Figure 1: *Visualization of a dynamic point cloud scene with 1 Million points. The raw point position data is streamed onto the GPU, where our algorithm performs an instant surface reconstruction that produces surface-aligned splats for illuminated rendering. The complete frame is computed in 94 ms at a resolution of of 1700 $\times$ 900 pixels.*

Abstract

Capturing real-world objects with laser-scanning technology has become an everyday task. Recently, the acquisition of dynamic scenes *at interactive frame rates has become feasible. A high-quality visualization of the resulting point cloud stream would require a per-frame reconstruction of object surfaces. Unfortunately, reconstruction computations are still too time-consuming to be applied interactively. In this paper we present a local surface reconstruction and visualization technique that provides interactive feedback for reasonably sized point clouds, while achieving high image quality. Our method is performed entirely on the GPU and in screen space, exploiting the efficiency of the common rasterization pipeline. The approach is very general, as no assumption is made about point connectivity or sampling density. This naturally allows combining the outputs of multiple scanners in a single visualization, which is useful for many virtual and augmented reality applications.*

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms I.4.1 [Image Processing an Computer Vision]: Digitization and Image Capture—Imaging geometry

1. Introduction

Laser scanning devices have become a common tool to acquire 3D object geometry. Recent work [WLG07, WAO*09] has made it possible to scan *dynamic* objects in real time, which allows for instant acquisition and processing of real-life film sequences. This trend to more flexible acquisition at ever increasing quality is very likely to grow in the near future. Common applications can be found in the film industry, rapid prototyping and augmented reality. The scanned point clouds are typically converted to a polygonal mesh, which is still a very time-consuming process. In particular, reconstructing a complete mesh is not possible if dynamic point clouds should be visualized directly from the scanner feed or during editing operations. A different, well-established

DOI: 10.2312/EGPGV/EGPGV12/139-148

approach is to visualize point clouds directly. A continuous surface can be obtained by drawing a *splat* for each point. A splat is a circular or elliptical surface element that is aligned with the surface tangent and covers an area according to the local point density. This method requires knowledge of the local surface tangents and point densities, which again implies lengthy preprocessing.

In this paper, we propose a technique to *interactively* estimate the above splat parameters for every output frame on the GPU. No prior knowledge is required about the incoming points, except for their 3D positions (and optional colors). We introduce a new screen-space algorithm to compute the k nearest neighbors (KNNs) of each point for local surface fitting. One main idea of our approach is to work directly in *screen space*, utilizing the features of the common graphics pipeline to process many points in parallel. Most importantly, working in screen space reduces the reconstruction workload to the required minimum, i.e., the *visible* points. Using a KNN search naturally accounts for sampling problems that arise from noisy data or spatially varying point density, for example, if point data coming from different sources are processed simultaneously. Furthermore, outliers are identified and excluded in order to maintain a high reconstruction quality. Our on-the-fly splat generation for large, dynamic point data enables high-quality visualizations using state-of-the-art splatting techniques without any preprocessing.

2. Related Work

A huge amount of work has been done in the field of *point rendering*, and we refer the interested reader to an excellent book [GP07] and surveys on this topic [Gro09, KB04]. Almost all existing techniques assume per-point normals as input (and optionally splat sizes), with a few exceptions that will be discussed below. In contrast, the main goal of this paper is to interactively compute normals and splat radii on the fly for the visible part of a given point set, to further apply any appropriate point-rendering technique for visualization.

Numerous offline algorithms have been presented for *surface reconstruction* for both static (e.g., Mullen et al. [MDGD*10]) and dynamic scenes [WJH*07, SAL*08, LAGP09]. Mitra et al. [MNG04] gives a broad discussion on normal estimation in noisy point cloud data and analyzes the influence of the noise factor, error bounds and similar factors. A further related problem is the consistent propagation of normal *orientations* in point clouds. König and Gumhold [KG09] proposed an offline algorithm.

In contrast to the above methods, we want to perform the reconstruction *online*. Zhou et al. [ZGHG11] sample the points into an octree structure and extract an implicit surface. To obtain high resolution and thus high reconstruction quality, the octree should tightly enclose the scene. For many real-world applications, this is not easily possible, especially considering large scanned data sets. A standard approach for per-point splat estimation is to compute each point's k nearest neighbors and fit a surface in this neighboring point set [HDD*92]. For these methods, the performance-critical part is the computation of the K-neighborhood for a large number of points, which mostly requires spatial acceleration data structures like grids or trees. Various techniques have been developed to perform KNN-Search interactively, mostly by utilizing modern graphics hardware [ZHWG08, PLM10, QMN09, LTdF*09, GDB08, JGBZ10]. While these approaches are able to reach fast peak performances, their efficiency mostly depends on carefully chosen parameters, which only perform well up to a certain data size due to hardware limitations (number of threads per block, shared memory occupancy, etc).

Aiming at a surface reconstruction for point cloud visualization, it is not efficient to reconstruct the surface on the complete data set, most of which might not even be visible. Our approach thus tries to limit the necessary reconstruction computations to the set of *visible* points. We compare our method to the work of Zhou et al. [ZHWG08], who perform a surface reconstruction on a deforming point model with a constant number of points by building a kd-tree on the complete point set and searching for the point's KNNs on the GPU in each frame. While fast performance can be achieved, the user needs to choose an initial search radius to provide an efficient reconstruction. Also, the KNN search efficiency strongly depends on parameters that were chosen for the different stages in the tree build-up phase. As our technique performs the KNN search *after* projecting the points in screen space, we do not rely on user-defined input parameters for KNN search acceleration. Furthermore, in our approach the size of the rendered point set is not restricted by GPU memory as for an object-space based approach. Computations are only bounded by scene complexity in image space.

Several point rendering methods apply a screen-space rendering approach as we do [MKC07, RL08, DRL10], but assume precomputed per-point normals as input. Some methods lift this restriction by also performing reconstruction in screen-space: Diankov and Bajcsy [DB07] applied erosion and dilation operations to fill holes, but the resulting image quality is not very high. Kawata et al. [HK04] base their reconstruction on a downscaled image, which is not applicable to input with varying point densities due to a user-provided, fixed grid size.

3. Overview

Our system allows for instant high-quality visualization of dynamic point sets from raw position data by performing a per-frame computation of surface aligned splats. The input to the algorithm is a set of n 3D points $S = \{x_i, y_i, z_i | i = 1..n\}$ containing only the point positions of the current frame. In addition, a parameter k defines the number of neighbors to take into account for splat normal and radius estimation.

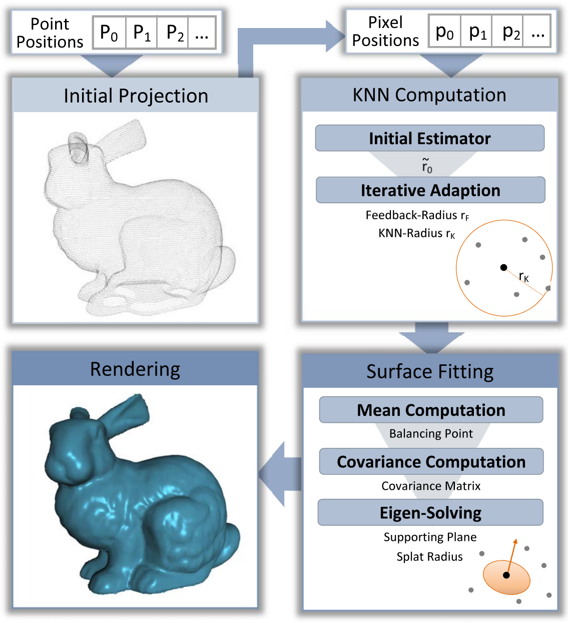

Figure 2: *Overview of the main steps of the algorithm*

This parameter defines a tradeoff between feature preservation and noise reduction.

The complete computation pipeline consists of three main phases, as depicted in Figure 2. First, each point is projected to its 2D pixel position in the current output image, where its 3D position information is stored. Then, the occupied output pixels are read back to a pixel position buffer that is used to address the points in the screen in the following steps without having to reproject the whole point cloud again. In the next step, each screen point finds its k nearest neighbors (Section 5). Based on the KNN, we perform surface fitting by computing a normal and a radius for each point (Section 6). Finally, we render the according splats to create the output image (Section 7). Note that all above computations are performed directly in screen space, utilizing the common graphics pipeline and programmable shaders to efficiently process points in parallel. In particular, the KNN search and normal and radius computation use a new parallel algorithm for communication between different screen-space splats which we describe in Section 4.

4. Parallel Splat Communication

This section describes how neighboring points can exchange information through screen-space rendering operations, which is a central building block for the KNN computation and surface estimation. Van Kooten et al. [vKvdBT07] use screen space splats in a particle system to *distribute* information from a point to all its neighbors within a given influence radius. This is similar to the distribution pass we describe below, but we also show how to efficiently *gather* information from a point's neighbors. Using both distribution and gathering allows us to iteratively compute the KNNs of each individual point.

Let P be a point in world space and let r be a radius that defines a neighborhood around P. We call any 3D point with an Euclidean distance to P smaller than r a *neighbor* of P. In screen space, this neighborhood projects to a general ellipse on the view plane, which we approximate by a tightly covering 2D *neighborhood box* (Figure 3). This box is guaranteed to contain the projections of all world-space neighbors of P. A naive way for P to gather information from its neighbors is to carry out texture lookups for all pixels covered by this neighborhood box. However, this would be prohibitively expensive due to the large number of required texture lookups especially for larger neighborhoods. We therefore introduce a gathering operation that is based on a *distribution* pass:

Distribution To pass information from P to all its neighbors in parallel fashion, we assign this information to the rendered box splat. For each pixel containing a point in the splat, we test (via a simple distance comparison) whether this point is a neighbor of P in world space. If so, the assigned information is written to the respective pixel, otherwise, the pixel is discarded.

Gathering If P needs to gather information from its neighbors (an operation that is used multiple times in our approach), we perform a distribution pass so that all neighbors write their information into the pixel of P. Note that for this task, since it is carried out for all points in parallel, the radius of the sphere that defines the screen splat for a point Q has to be the distance to the furthest point P that has Q as its neighbor. We call this distance the *feedback radius* r_f. Contrary to the distribution pass, for each point P covered by the splat of Q we test whether Q lies within the neighborhood sphere of P. If so, Q is a neighbor to P and we can perform a feedback write at the pixel coordinates of P. To compute r_f for the neighbors of P, we perform a distribution pass that writes the world-space distance $|Q-P|$ to the splat pixel of each neighbor Q. Using MAX-Blending, each point ends up with the distance r_f to the furthest point for which it serves as neighbor.

Using distribution and gathering, we can accumulate information from each point's neighbors in parallel. Note that by the above definition, each point P is its own neighbor. However, we can always choose whether to operate on the entire neighbor point set or only on the neighbors $Q \neq P$, by discarding a pixel at Q in the fragment program if it equals the position of the distributing point P.

To minimize the number of fragment threads required for splat communication, we take advantage of the hardware's Early-Z culling ability that is implemented in most modern GPUs. We only need the GPU to start a fragment thread for the "non-empty" pixels within a neighborhood box, i.e.,

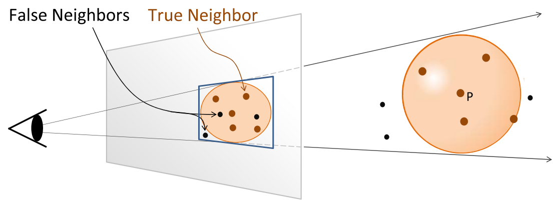

Figure 3: *Projecting the sphere (P, r) to the viewplane reduces the search space for the neighbors of P inside r to an elliptical region containing all points inside the frustum that is defined by the ellipse. An inside-outside test with the sphere on each point in this region yields the neighbors of P.*

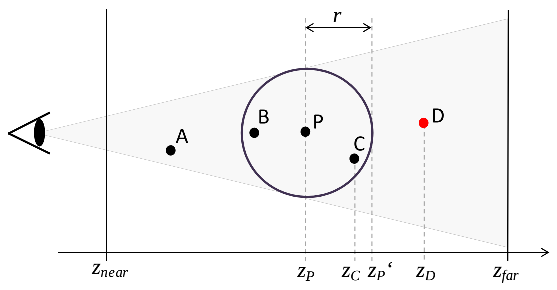

Figure 4: *P communicates with its neighbors within a range r by rendering a neighborhood splat, which in screen space contains false positives A and D. By choosing a splat depth of z'_P, depth test discards all splat fragments containing points with $z > z'_P$. Thus, D is Early-Z culled.*

those which actually contain a projected point. To achieve this, we create a depth mask in the z-buffer at the time of initial point projection, which causes all empty pixels to fail the depth-test and to not launch a fragment thread. The remaining pixels in the neighborhood box contain both true neighbors (actually within world-space range) and false neighbors, as depicted in Figure 3.

To further increase the efficiency, the depth buffer is set up in a way that allows the hardware to discard about 50% of false neighbors at Early-Z: After initial projection at the beginning of each frame, the depth buffer contains the normalized depth footprint for all visible points and depth 1 for all empty pixels. When drawing the splat that defines the neighborhood box for a given point P in the screen, we pass its fragments a biased depth $z'_P = f(z_P + r)$, where z_P is the view space z-coordinate of P, r is the neighborhood radius that defines the communication range of P, and f is a mapping from view-space depth to clip space depth. Setting the depth comparison function to "GREATER" lets the z-buffer cull all points at Early-Z that lie beyond the depth-border represented by z'_P (see Figure 4), while still maintaining Early-Z discards for empty pixels. Note that depth-writes have to be disabled during the whole splat communication phase of the algorithm to maintain the state of the initial depth footprint.

5. Neighborhood Computation

We define the k-neighborhood of a point P by the k-radius r_k that encloses the k nearest neighbors of P. Once this radius is found, we are able to perform operations on the KNNs in parallel by using it as communication radius threshold for the distribute and gather mechanisms in our system. The k-radius r_k is found by an iterative range search in the non-continuous, monotonically increasing neighbor-count function $\sigma(r)$ over the radii r. Starting with an initial estimator $\tilde{r}_0$, in each iteration, the number of points in the current range is determined and used to update the search range until a range $\tilde{r}_i$ is found that counts $\sigma(\tilde{r}_i) = k$ points.

5.1. Initial Estimator

Having a good estimator for the initial range $\tilde{r}_0$ is critical for fast convergence of the iterative search. We propose an automatic approach to determine $\tilde{r}_0$ individually for each point, which is important for scenes with spatially varying point densities. In contrast, in their online surface reconstruction technique, Zhou et al. [ZHWG08] require the user to manually set $\tilde{r}_0$, and to obtain correct results, their approach requires $\tilde{r}_0$ to be conservative, i.e., encompass r_k.

We apply a screen-space approach to obtain $\tilde{r}_0$: A low-resolution grid that contains the projected points is laid over the screen, and the number of points within each grid cell is counted by performing an accumulation pass. Each frame, the grid resolution is chosen based on the current point count in such a way that on average, k screen points fall into one cell. For each grid cell, we choose $\tilde{r}_0$ based on its point density. Since we presume to operate on points that describe an object surface, it makes sense to assume a two-dimensional distribution of the points within a cell. Let A be the cell's pixel area and n the number of points in that cell. The average cell area covered by k points is estimated by $A\frac{k}{n}$, and the pixel radius r_{screen} of a circle covering the area of k points by

$$r_{screen} = \sqrt{\frac{A}{\pi}\frac{k}{n}}.$$

The initial world-space estimator $\tilde{r}_0$ of a point can then be derived by unprojecting r_{screen} based on the point's view-space depth. If the points in the cell describe one single unwrapped surface, this estimator roughly pinpoints the expected KNN radii. On the other hand, if several depth layers are projected to the screen (e.g., the front- and the backfacing part of a closed object), the number of cell points will be too high. This is acceptable, however, since in the worst case, this estimator causes the initial screen splat defined by $\tilde{r}_0$ to be smaller than intended, and it will be expanded in the next step.

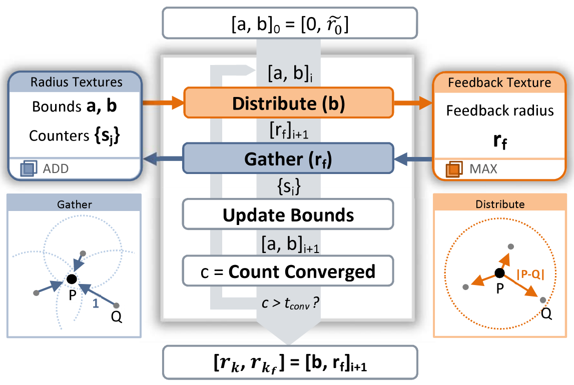

Figure 5: *K-radius search algorithm and textures used for data writes and reads. The texture boxes show the stored main information and the used blend modes. After each iteration, the number c of converged points is counted. Iteration is finished if c exceeds some desired threshold t_{conv}.*

5.2. Iterative Radius Search

Figure 5 illustrates the iterative k-radius search. For each point, we use the initial estimator $\tilde{r}_0$ as starting value for searching the target value r_k on the function $\sigma(r)$. Similar to a histogram-based KNN search in a kd-tree [ZHWG08], the search is performed using a multi-sampled bracketing approach that iteratively narrows the location of r_k by two bounding radii a (lower bound) and b (upper bound). In every iteration, each point's upper bound defines its current neighborhood. The number of neighbors σ_i within this neighborhood can be queried using a distribution pass to the neighbors to obtain the corresponding feedback radius r_{f_i}. A following gathering pass with r_{f_i}-sized feedback splats then accumulates a counter from each neighbor, yielding the current neighbor count σ_i. Instead of taking just one neighbor-count function sample σ_i at b, multiple samples $\{s_j | j = 1..m\}$ are taken at m regular steps between a and b. The total number of neighbors σ_i is therefore represented by s_m. To query multiple samples, m feedback counters are accumulated in each gathering pass, stored in several target texture channels in the radius textures (we use $m = 4$ in our implementation). Multisampling results in faster convergence since in each iteration it significantly raises the chance to find r_k and allows for a much tighter narrowing of the bounds. Based on the current bounds a_i, b_i and the counter samples $\{s_j\}$, the adapted bounds a_{i+1}, b_{i+1} for the next iteration are then computed. This adaptation occurs in two phases:

Expanding As long as $\sigma_i < k$ holds, b is enlarged. The new upper radius bound b is chosen by extrapolating the current neighbor count σ_i to k assuming a constant two-dimensional point density. This linear relation between surface area and point count yields the radius increase factor

$$\alpha = \sqrt{\frac{k}{\sigma_i}}.$$

Bracketing For a point count $\sigma_i > k$, a and b are iteratively narrowed until a radius with corresponding neighbor count sample $s_j = k$ is found. Bracketing is necessary to ensure a view-independent reconstruction. A faster but more naive approach would be an approximate nearest neighbors search that stops after the expanding stage and uses the resulting neighbor count $\kappa \geq k$ for reconstruction. However, since the iterative search is initialized based on the point distribution in image space, this would lead to a different κ and thus a different splat reconstruction under different views, resulting in temporal flickering artifacts under camera movement, even for static point scenes.

Implementation Details We use a single feedback texture T_F and two radius textures $T_{R1,2}$ to store all required data. Besides the texture data shown in Figure 5, additional data is stored and ping-ponged between T_F and T_R, e.g., the current radius bounds a and b. Using separate blend functions for the RGB and the alpha channel, we achieve accumulating $\{s_j\}$, MAX-computing r_f, and passing along the additional data at the same time. To reduce fragment writes to a minimum, we do not actually feed back the counters from every neighbor within b, but only from those located within the delta region between a and b. To obtain the required counter samples $\{s_j\}$, we also store the last neighbor count at a and add it to the accumulated counters. If a converged point is about to distribute, a 1-pixel sized splat is drawn to still pass along the converged k-radius result until iteration is finished. Similarly, if a point that accumulated a zero r_{f_i} is about to feed back, a 1-pixel splat is required to pass along the point's data. The number of converged points is efficiently counted using occlusion queries by looking up each point's current data in the radius textures and emitting a fragment if it is found to be converged.

5.3. Robustness

This section discusses some robustness issues that arise in noisy scenes with many outliers and due to the information loss we trade against performance when reconstructing on a point set reduced through screen projection.

Outliers Points in the framebuffer that have no neighbors in their immediate neighborhood can appear due to outliers in the point data set (e.g. scanner data) or because their neighbors are occluded by closer points in the depth buffer. In the expanding phase of the iterative search for r_k (Section 5.2), such points would continuously increase their search radius $\tilde{r}$ without finding any neighbor. This leads to huge screen splats when projecting the search sphere onto the framebuffer, which can significantly reduce performance. Generally, for our approach the classification of such points is undecidable, since during iterative search we cannot distinguish between outliers with no real neighbors and points belonging to a coarsely sampled surface whose neighbors

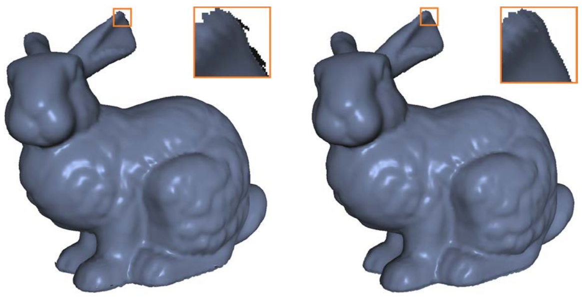

Figure 6: *Comparison of a bunny model rendered with Auto Splats (left) and with precomputed normals using the same normal and radius estimation procedure (right).*

have just not been found yet. To reduce visual artifacts produced by outliers, we discard points with no neighbors after a certain number e_0 of expanding iterations. In all our test scenes we found that $e_0 = 3 \sim 4$ is sufficient for a good reconstruction.

Small Point Groups Outlying point groups containing $\kappa < k$ points represent a similar problem as single outlier points. To prevent our system from expanding the search radii up to k neighbors by bridging large gaps with no points, the radius expansion is further constrained. In each search iteration, the distance d_{max} of the furthest current neighbor is tracked. If the circular area defined by the expanding search radius $\tilde{r}_i$ grows by a certain factor λ without finding a new neighbor, expansion is aborted and the radius r_k is clamped to the reduced κ-neighborhood $r_\kappa = d_{max}$. In our scenes we used a $\lambda = 4$ to cover surfaces of moderately irregular point distribution while avoiding too large bridging splats.

6. Surface Fitting

The supporting plane of the splat attached to a point P is computed by fitting a plane π to the set of points $S = \{x_i | i = 1 \ldots k\}$ in a local neighborhood of P by using linear regression [HDD*92]. A common method to find the parameters of π in the form $\pi : \ n \cdot (x - \bar{x}) = 0$ is to compute $\bar{x}$ as the mean

$$\bar{x} = \frac{1}{k} \sum_i x_i \tag{1}$$

of the point set S, and n as the eigenvector to the lowest eigenvalue of the scaled covariance matrix

$$cov(S) = \sum_i (x_i - \bar{x})(x_i - \bar{x})^T . \tag{2}$$

With the KNN radius r_k at hand, this computation can be carried out in three steps in our system (see Figure 2):

Mean Accumulation First we perform a *gathering pass* that accumulates the mean $\bar{x}$ of the points in the neighborhood of P by ADD-blending each neighbor's world-space position as well as the counter value 1 for counting the number of accumulated values (Equation 1). The latter is necessary since we cannot be completely sure that each point has found exactly k neighbors in the KNN-radius computation pass before (see also Section 5.3).

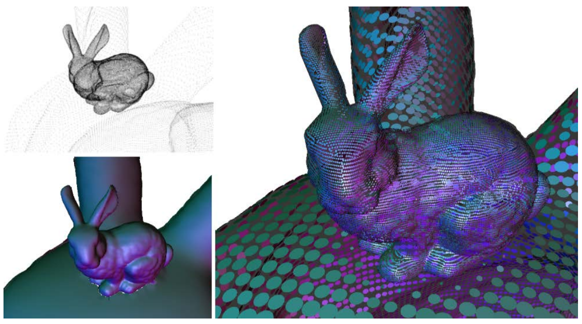

Figure 7: *Reconstruction quality in a scene exhibiting large differences in point density, here showing a Stanford Bunny sitting on the head of another, larger Bunny (left top). Surface aligned splats can be computed for both the large scale and the small scale model (left bottom). In the right image, splat size was scaled down for better visibility of the splats.*

Covariance Accumulation In a second gathering pass, we accumulate the terms required for summing the covariance matrix from all neighbors (Equation 2). Each neighbor contributes the symmetric matrix $(x - \bar{x})(x - \bar{x})^T$, where the mean $\bar{x}$ is calculated by dividing the accumulated position by the counter value calculated in the previous pass. Since the covariance matrix is symmetric, it is sufficient to accumulate only the 6 values of its upper triangle matrix, which can be stored compactly within only two render-target textures.

Eigen Solving A final per-pixel pass reads the covariance values and computes the eigenvector to the least eigenvalue using a standard eigensolver procedure [Ebe11] in a fragment shader program. This eigenvector defines the non-oriented normal of the supporting plane of the splat. Since we do not intend to consistently reconstruct the complete surface, but only the visible parts of the point cloud needed for rendering, we simply orient the normals towards the eye point. We can also render elliptical splats by additionally computing the remaining two eigenvectors, which represent the minor and major axes of the ellipse. The fraction of their two eigenvalues is used as the proportion of their respective lengths [Paj03].

To determine the splat radius, we use a quick estimator based on the average area coverage of the points in the local neighborhood. At k neighbors, the radius of a circle enclosing the average area covered by each neighbor is defined by $\bar{r} = \sqrt{\frac{r_k^2}{k}}$. We choose the splat radius to be the average distance between neighboring points, which we approximate

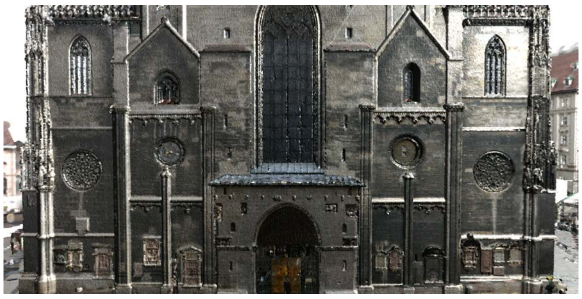

Figure 8: *A huge laser scan of a cathedral (∼500M points) rendered by an out-of-core point renderer with Auto Splats. The renderer streams an amount of ∼10M points to the GPU each frame.*

by $r_{splat} = 2\bar{r}$. For elliptical splats, this value is used for the length of the semi-minor axis.

7. Auto Splat Rendering

After surface reconstruction has finished, we can use the splat normals and radii for rendering using any existing splat-based rendering technique. We implemented Surface Splatting using Gaussian Blending (see [BHZK05]). This method employs three passes: First, a *visibility* pass producing a depth map; second, an *attribute* pass that uses the depth map for proper occlusion culling and front-surface attribute accumulation, and last, a *normalization and shading* pass, that applies deferred shading computation. In the following, we describe two extensions to the technique which improve the quality and speed of rendering reconstructed splats.

Depth Refinement Our reconstruction started with a rendering of all points as one-pixel sized point primitives. In this rendering, there are regions where the foreground surface is quite densely sampled, but there are still holes where pixels from background surfaces are visible. Most of the actual neighbors of these background pixels are occluded by foreground pixels, and therefore KNN estimation can produce large splats, which can lead to artifacts because they extend beyond the silhouette of the foreground surface in the final rendering. Because these points appear mostly in backfacing or occluded regions, we can get rid of most of these artifacts by using vertex-based occlusion culling against the initial depth map. Unfortunately, since the depth map from the first visibility pass is rendered *with* those incorrect splats, the visibility information in the depth map might be corrupt and we would miss a number of surface points for rendering.

Thus, we extend the usual surface splatting pipeline by an additional depth buffer *refinement* stage that produces an improved depth map without artifacts. This is done by two additional depth passes. First, all points in the screen are culled against the initial coarse depth map to render a new, mostly artifact-free depth map. Then, we cull the points against this second map to remove possible holes and obtain a final refined depth map which is used as input for the attribute pass.

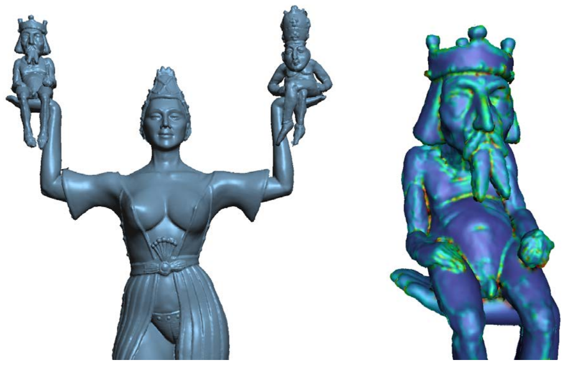

Figure 9: *Left: Autosplatted image of a range scan of the Imperia Statue. Right: closeup on a part of the statue, visualizing the curvature estimates for the points. Like normals and splat radii, curvature is dynamically computed from the KNNs provided by our algorithmn.*

Grid Culling In larger point clouds with higher depth complexity of the reconstructed scene, we often face the situation that due to perspective projection, the foreground surface is sampled only sparsely in screen space, while background surfaces densely cover the screen. This can become inefficient in our framework, as we would spend most of the time reconstructing the background surfaces, which will finally be culled against the foreground surface. To improve such scenarios, we apply a simple optional approximate culling technique based on the low-resolution grid used to accumulate densities for the initial KNN radius estimator (see Section 5.1). While accumulating point counts in this grid, we also determine the depth d_i of the nearest point per grid cell i. For each cell, a culling plane is then defined at depth $d'_i = d_i + s_i$ to set all culled points in *passive* state. Here, s_i represents the unprojected side length of screen cell i in world space at depth d_i. We then apply reconstruction (KNN search, fitting, visibility) only for the remaining *active* points. The passive point set is, however, still used for communication with the active points, i.e., they are still involved in gathering passes (KNN search, fitting) to maintain a correct reconstruction for the active set. The depth map obtained from the active set can then be used to perform an accurate culling pass for the passive points, and apply reconstruction for the small set of points incorrectly classified by the approximate culling step. For large point clouds, the overhead incurred for the additional reconstruction pass is easily outweighed by the reduced workload in the first pass. We have observed a speed-up of up to 60% for large scenes.

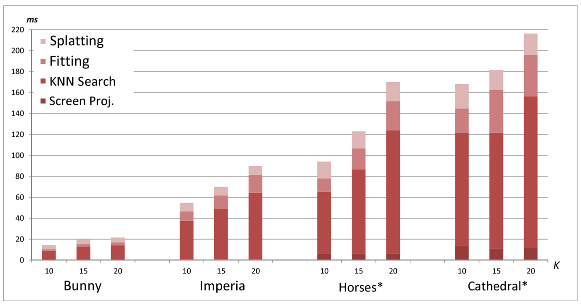

Figure 10: *Performance decomposition of the computation pipeline for our test scenes for different neighborhood sizes K and 99.9% convergence. The main part of the frame time is drawn by the KNN search. The actual time for plane fitting lies in the magnitude of the final splatting stage. The horses and the cathedral scene were drawn using grid culling.*

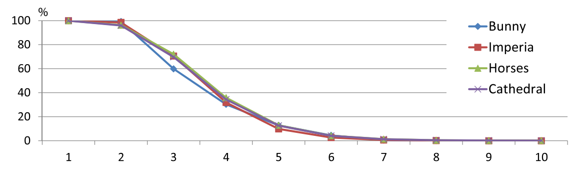

Figure 11: *Convergence behaviour of the k-radius search. The y-axis denotes the percentage of unconverged points before each iteration.*

	Points	Pixels	kdTree	AS	Speedup	
Scene	#	#	ms	ms	p.pt.	**ALL**
Bunny	36K	34K	14	9	1,47	**1,56**
Imperia	546K	291K	95	37	1,37	**2,57**
Horses	1M	690K	169	117	1,00	**1,44**
Horses*	1M	690K	169	71	1,64	**2,38**
Cathedral	10M	1.3M	-	264	-	-
Cathedral*	10M	1.3M	-	123	-	-

Table 1: *k-radius search times in ms at $k = 10$ achieved with Auto Splats compared to a GPU kd-tree [ZHWG08] with supposed preknown ideal parameters. The horse and the cathedral scene were measured both with* and without grid culling. The kd-tree times represent time for tree build-up plus k-radius search. Speedups are listed per element (p.pt), i.e., per point or pixel, and overall (all), i.e., taking into account the savings of handling only visible points in Auto Splats.*

8. Results and Discussion

8.1. Reconstruction Quality

Figure 6 compares the reconstruction quality of Auto Splats with preprocessed normals that were computed using the same neighborhood size and normal and radius estimation procedure than used in our algorithm. Despite some minor artifacts due to information loss at silhouettes, we observe a similar visualization. Since we are especially interested in the algorithm's behavior in scenes with point sets of strongly varying sample densities, we placed two bunny models with a large difference in scale into the same scene (Figure 7). Note that we reduced the splat radii in this image for better visualization of the reconstructed splats. Our algorithm is able to perform a reconstruction that correctly adapts the splat sizes to the local point densities in world space without relying on user input. Note that in the method of Zhou et al. [ZHWG08], such a scene can only be rendered without artifacts if the user-specified initial radius estimator $\tilde{r}_0$ is chosen large enough for the larger model, which is very inefficient for the smaller model.

8.2. Performance

The performance of our system has been profiled for four different scenes (Figures 6, 9, 1 and 8) exhibiting different characteristics in the geometry and density of the points. All measurements have been taken at a resolution of 1760x900 (Imperia was rendered in portrait format) using a GeForce GTX580 Graphics Card with 1536MB VRAM and an Intel i7-930 CPU with 2.8 GHz. Figure 10 shows the required rendering times for the test scenes at different neighbor counts and decomposes the frame computation times according to the amount of time spent in each pass. The main part of the computation time is spent in the KNN search stage. For the shown test scenes, our KNN search requires about 7–10 iterations to converge for 99,9% of the points at $k = 10$. Moreover, as depicted in Figure 11, we observe a characteristic convergence graph that is common to all our test scenes, independent of the number of points. It can be seen that generally, 6–7 iterations already provide a good quality.

We also compared the performance of our method to the GPU kd-tree of Zhou et al. [ZHWG08], which also allows reconstructing some point-based models on-the-fly. We reimplemented this method and analyzed its performance characteristics. Since tree build-up parameters are not reported in the paper, we provide a best-case comparison for their method. We profiled the tree build-up and k-radius search using several parameter combinations (including the user-specified initial range $\tilde{r}_0$) for our test scenes. As suggested by the authors, we used a histogram resolution of $n_{hist} = 32$ and $n_{iter} = 2$ range search iterations. We picked the lowest achieved timings that came reasonably close to the k-radius accuracy of 99,9% we use in our scenes and compared them to the timings of the KNN-search in the Auto Splatting system (Table 1). Note that the kd-tree timings include the tree build-up since we assume dynamic scenes. In all scenes, our system outperforms the GPU kd-tree variant. On the one hand, this is explained by the fact that our system performs the search only on the reduced set of visible points. On the other hand, even assuming the same number of points (by considering the average time spent on a KNN-

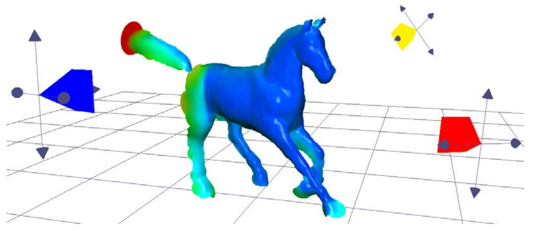

Figure 12: *A dynamic object scanned by three scanners.*

query for a point), Autosplats provide a speedup over GPU kd-trees. Furthermore, the GPU kd-tree algorithm was not able to handle larger scenes like the Cathedral scene at all.

8.3. Applications

Real-Time Scan Visualization One application where our technique is especially useful is the setup of real-world scanning sessions of static and dynamic content, possibly using multiple scanners. We simulated such a setup, as depicted in Figure 12. A dynamic object is scanned by up to four scanners, each providing a registered 3D point cloud (assuming mutual scanner registration). The scanners simulate varying amounts of Gaussian noise. The auto-splatting technique allows us to display the scanned surfaces with high quality in real time. This enables positioning the scanners for optimizing surface coverage and minimizing occlusions in a scene during a film sequence, for example. In addition, we can apply helpful visualization techniques, like the size of the K-neighborhood as depicted in Figure 12, where the red spectrum suggests too sparse sampling. Note that for such scenarios, we can turn off the outlier removal of our system to not distort the result. We believe that in the near future, more real-time scanning devices will be available (like Microsoft Kinect), making a fast but high-quality preview particularly important for many film, VR and AR applications.

Normal and Curvature Estimation Preview Offline algorithms performing normal or curvature estimation on massive point datasets can require up to several hours of processing time. However, the reconstruction quality of the hole dataset is not known in advance and errors in the choice of the parameters that could require a recomputation are often only recognized after the processing is finished. Our algorithm can be used to provide an instant preview of the reconstruction of different parts of a data set by an interactive walk-through. A user might wish to test different parameters for the k neighborhood to find a smooth but still feature-preserving optimum, or wants to analyze whether a certain parameter choice leads to uneven quality among the points. See for example Figure 9 for an instant visualization of curvatures.

Modeling Applications Applications that allow the user to modify the point cloud cannot rely on a lengthy preprocessing phase for normal vector estimation. An example is an application that allows archaeologists to modify a scene to experiment with different reconstructions of an archaeological site.

8.4. Discussion and Limitations

Our design choice to locally reconstruct surfaces based on the points sampled in screen has many advantages. For example, there are practically no parameters that influence the reconstruction quality (except for k for the KNN search), making the system readily useful for many applications. On the other hand, because each pixel only stores the front-most point, a number of points get lost. Generally, this is intentional, especially in large or dense point clouds with high pixel overdraw where we only want to perform computation on the the potentially small fraction of visible points in the front. However, not storing a point can become a problem if either its splat would still contribute to the final rendering, or it would contribute to the normal direction of a visible splat in its neighborhood. This can happen for example at object silhouettes where spatially neighboring points can get rasterized to the same image pixel, leading to misaligned splats.

Another limitation is that we currently do not account for splat orientation, as this would require computing a minimum spanning tree in the weighted K-neighborhood graph [HDD*92], which seems too costly for a real-time setup. This sometimes leads to wrongly illuminated splats, mostly at silhouettes. We are currently investigating methods to reduce these artifacts.

9. Conclusions and Future Work

We have presented an algorithm for producing interactive high-quality visualizations from dynamic point clouds by performing a surface reconstruction of the point clouds on the fly. Our algorithm uses the frame buffer as search data structure for nearest-neighbor computation. This comes with the advantage that we obtain a reduced subset of the possibly huge amount of points in the scene for reconstruction. The main strength of our algorithm is that it can be used to interactively render reconstructions for point clouds from any source, including dynamic point clouds from real-time scanners, or point data from an out-of-core rendering system.

We have shown that our method outperforms a GPU-based kd-tree approach for KNN search in each of our tests, even if the parameters required for the kd-tree are chosen favorably. Our method, on the other hand, does not require manual parameter tuning, and also works for larger scenes. In the future, we want to investigate ways to reduce the remaining small artifacts that can appear at object silhouettes, where information for fitting is lost. Also, we will investigate methods to choose k adaptively in order to further increase visual quality.

10. Acknowledgements

This work was funded by the Austrian Research Promotion Agency (FFG) through the FIT-IT project "Terapoints" and the Austrian Science Fund (FWF) project P20768-N13. The Imperia Statue was provided by the University of Konstanz. The horse animation was made available by Robert Sumner and Jovan Popovic from the Computer Graphics Group at MIT.

References

[BHZK05] Botsch M., Hornung A., Zwicker M., Kobbelt L.: High-quality surface splatting on today's gpus. In *Point-Based Graphics, 2005. Eurographics/IEEE VGTC Symposium Proceedings* (june 2005), pp. 17 – 141. 7

[DB07] Diankov R., Bajcsy R.: Real-time adaptive point splatting for noisy point clouds. In *GRAPP (GM/R)* (2007), pp. 228–234. 2

[DRL10] Dobrev P., Rosenthal P., Linsen L.: Interactive image-space point cloud rendering with transparency and shadows. In *Communication Papers Proceedings of WSCG, The 18th International Conference on Computer Graphics, Visualization and Computer Vision* (Plzen, Czech Republic, 2 2010), Skala V., (Ed.), UNION Agency – Science Press, pp. 101–108. 2

[Ebe11] Eberly D.: Eigensystems for 3x3 symmetric matrices (revisited). `http://www.geometrictools.com/Documentation/EigenSymmetric3x3.pdf`, 2011. [Online; accessed 29-Sept-2011]. 6

[GDB08] Garcia V., Debreuve E., Barlaud M.: Fast k nearest neighbor search using gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on* (june 2008), pp. 1 –6. 2

[GP07] Gross M., Pfister H.: *Point-Based Graphics (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. 2

[Gro09] Gross M.: Point based graphics: state of the art and recent advances. In *ACM SIGGRAPH 2009 Courses* (New York, NY, USA, 2009), SIGGRAPH '09, ACM, pp. 18:1–18:68. 2

[HDD*92] Hoppe H., DeRose T., Duchamp T., McDonald J., Stuetzle W.: Surface reconstruction from unorganized points. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1992), SIGGRAPH '92, ACM, pp. 71–78. 2, 6, 9

[HK04] Hiroaki Kawata Alexandre Gouaillard T. K.: Interactive point-based painterly rendering. In *Proc. International Conference on Cyberworlds 2004 (CW2004)* (Nov. 2004), pp. 293–299. 2

[JGBZ10] Jie T., Gangshan W., Bo X., Zhongliang G.: Interective point clouds fairing on many-core system. In *Proceedings of the International Symposium on Parallel and Distributed Processing with Applications* (Washington, DC, USA, 2010), ISPA '10, IEEE Computer Society, pp. 557–562. 2

[KB04] Kobbelt L., Botsch M.: A survey of point-based techniques in computer graphics. *Comput. Graph. 28* (December 2004), 801–814. 2

[KG09] König S., Gumhold S.: Consistent propagation of normal orientations in point clouds. In *VMV* (2009), pp. 83–92. 2

[LAGP09] Li H., Adams B., Guibas L. J., Pauly M.: Robust single-view geometry and motion reconstruction. *ACM Trans. Graph. 28* (December 2009), 175:1–175:10. 2

[LTdF*09] Leite P., Teixeira J., de Farias T., Teichrieb V., Kelner J.: Massively parallel nearest neighbor queries for dynamic point clouds on the gpu. In *Computer Architecture and High Performance Computing, 2009. SBAC-PAD '09. 21st International Symposium on* (oct. 2009), pp. 19 –25. 2

[MDGD*10] Mullen P., De Goes F., Desbrun M., Cohen-Steiner D., Alliez P.: Signing the unsigned: Robust surface reconstruction from raw pointsets. *Computer Graphics Forum 29*, 5 (2010), 1733–1741. 2

[MKC07] Marroquim R., Kraus M., Cavalcanti P. R.: Efficient point-based rendering using image reconstruction. In *SPBG* (2007), Botsch M., Pajarola R., Chen B., Zwicker M., (Eds.), Eurographics Association, pp. 101–108. 2

[MNG04] Mitra N. J., Nguyen A., Guibas L.: Estimating surface normals in noisy point cloud data. In *special issue of International Journal of Computational Geometry and Applications* (2004), vol. 14, pp. 261–276. 2

[Paj03] Pajarola R.: Efficient level-of-details for point based rendering. In *Computer Graphics and Imaging* (2003), pp. 141–146. 6

[PLM10] Pan J., Lauterbach C., Manocha D.: Efficient nearest-neighbor computation for gpu-based motion planning. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on* (oct. 2010), pp. 2243 –2248. 2

[QMN09] Qiu D., May S., Nüchter A.: Gpu-accelerated nearest neighbor search for 3d registration. In *Proceedings of the 7th International Conference on Computer Vision Systems: Computer Vision Systems* (Berlin, Heidelberg, 2009), ICVS '09, Springer-Verlag, pp. 194–203. 2

[RL08] Rosenthal P., Linsen L.: Image-space point cloud rendering. In *Proceedings of Computer Graphics International* (2008), pp. 136–143. 2

[SAL*08] Sharf A., Alcantara D. A., Lewiner T., Greif C., Sheffer A., Amenta N., Cohen-Or D.: Space-time surface reconstruction using incompressible flow. *ACM Trans. Graph. 27* (December 2008), 110:1–110:10. 2

[vKvdBT07] van Kooten K., van den Bergen G., Telea A.: Point-based visualization of metaballs on a gpu. *GPU Gems*, 3 (2007). 3

[WAO*09] Wand M., Adams B., Ovsjanikov M., Berner A., Bokeloh M., Jenke P., Guibas L., Seidel H.-P., Schilling A.: Efficient reconstruction of nonrigid shape and motion from real-time 3d scanner data. *ACM Trans. Graph. 28* (May 2009), 15:1–15:15. 1

[WJH*07] Wand M., Jenke P., Huang Q., Bokeloh M., Guibas L., Schilling A.: Reconstruction of deforming geometry from time-varying point clouds. In *Proceedings of the fifth Eurographics symposium on Geometry processing* (Aire-la-Ville, Switzerland, Switzerland, 2007), Eurographics Association, pp. 49–58. 2

[WLG07] Weise T., Leibe B., Gool L. V.: Fast 3d scanning with automatic motion compensation. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR'07)* (June 2007). 1

[ZGHG11] Zhou K., Gong M., Huang X., Guo B.: Data-parallel octrees for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics 17* (2011), 669–681. 2

[ZHWG08] Zhou K., Hou Q., Wang R., Guo B.: Real-time kd-tree construction on graphics hardware. In *ACM SIGGRAPH Asia 2008 papers* (New York, NY, USA, 2008), SIGGRAPH Asia '08, ACM, pp. 126:1–126:11. 2, 4, 5, 8