Efficient Collision Detection While Rendering Dynamic Point Clouds

Mohamed Radwan* Stefan Ohrhallinger† Michael Wimmer‡

Vienna University of Technology, Austria

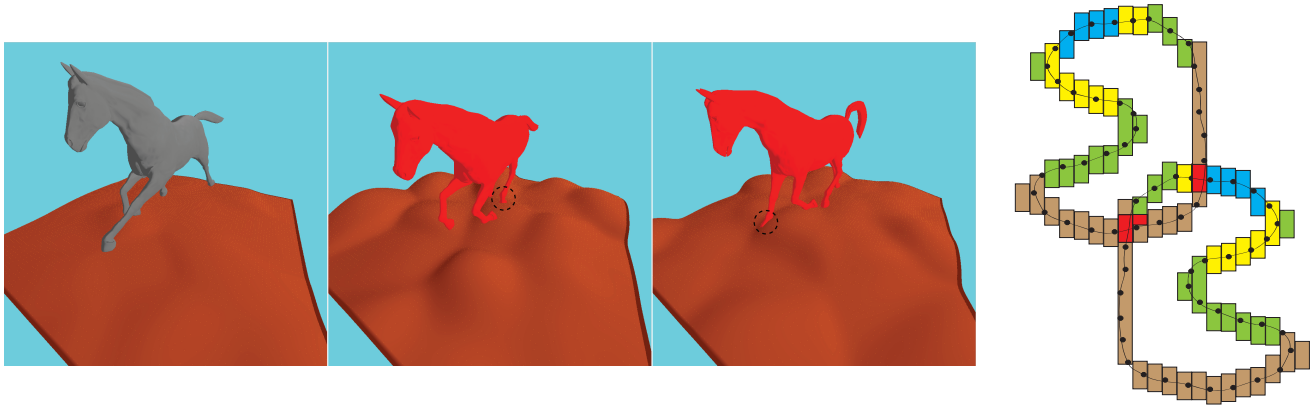


Figure 1: Left: Subsequent snapshots of an animated HORSE traversing continuously oscillating ground. Collisions between those two dynamic point clouds are marked by circles and HORSE is shaded red. Right: TLDIs of two objects. Colliding extents are shaded red.

ABSTRACT

A recent trend in interactive environments is the use of unstructured and temporally varying point clouds. This is driven by both affordable depth cameras and augmented reality simulations. One research question is how to perform collision detection on such point clouds. State-of-the-art methods for collision detection create a spatial hierarchy in order to capture dynamic point cloud surfaces, but they require $O(NlogN)$ time for N points. We propose a novel screen-space representation for point clouds which exploits the property of the underlying surface being 2D. In order for *dimensionality reduction*, a 3D point cloud is converted into a series of *thickened layered depth images*. This data structure can be constructed in $O(N)$ time and allows for *fast surface queries* due to its increased compactness and memory coherency. On top of that, parts of its construction come for free since they are already handled by the rendering pipeline. As an application we demonstrate online collision detection between dynamic point clouds. It shows *superior accuracy* when compared to other methods and *robustness to sensor noise* since uncertainty is hidden by the thickened boundary.

Index Terms: Computer Graphics [I.3.5]: Computational Geometry and Object Modeling—Hierarchy and Geometric Transformations Image Processing and Computer Vision [I.4.8]: Scene Analysis—Surface Fitting

1 INTRODUCTION

This paper proposes a novel accelerated approach for constructing and querying the underlying surface of dynamic point clouds. When those point clouds are rendered, calculations from the point-based rendering (PBR) pipeline are reused in the surface construction for the points inside the view frustum.

Collision detection requires determining the distance from the shape boundary of the object. For point clouds, especially noisy ones, reconstructing the surface as a triangulated mesh is a tedious process which currently is not feasible to do online. Applications where collision detection between dynamic point clouds is relevant include moving and posing of objects, as well as touch and grip. Such point clouds are dynamically changing environments, e.g., acquired by sensors attached to drones in a disaster scenario as simultaneous location and mapping (SLAM), human avatars captured by a Kinect, or deforming virtual objects for augmented-reality applications. Physically remote point clouds may be transposed into a common coordinate system to allow for interaction. Finally, user interaction can lead to non-rigid deformation or fragmentation.

We target medium-to-large and possibly noisy point clouds which are dynamic in the sense of having little or no temporal coherence. Constructing a spatial hierarchy for geometry, e.g., bounding volume hierarchies (BVH) [12], or tree structures [20], adds a logarithmic time factor to collision processing with respect to the number of handled points. This setup time is amortized only for static point clouds. Using a BVH allows for deformations and local rigid transformations, but not for entirely dynamic point sets. With interactive applications, the interest is often concentrated inside the view frustum, since it determines what the viewer can see and manipulate.

Our main goal is to enable online processing of medium-to-large dynamic point clouds without temporal coherence, such as Kinect input. We achieve this by avoiding construction of spatial hierarchies altogether and instead discretize the surface underlying the points into a screen-space grid. This two-dimensional structure reduces the dimensionality of the grid and thus results in more compact storage and faster intersection testing. The advantages of using a grid remain, namely that construction and evaluation can be parallelized well on the GPU.

*e-mail: radwan@cg.tuwien.ac.at
†e-mail:ohrhallinger@cg.tuwien.ac.at
‡e-mail:wimmer@cg.tuwien.ac.at

Our contributions are:

- *Efficient reconstruction of connectivity* for point clouds where an estimation of local sampling density is available, even in the presence of noise.
- *Compact boundary discretization* of point clouds by extending layered depth images with range, adapted to screen space.
- *Reuse of parts of the rendering pipeline* for constructing the boundary data structure for the point cloud.
- *Precise and online collision detection of dynamic point clouds* as an example application for surface distance queries.

2 Related Work

Bounding volume hierarchies (BVHs). These are spatial object representation structures that have been widely used in many applications. Different volume types are used to bound the geometric primitives, such as AABBs [2], OBBs [8], DOPs [13], and convex hulls [5]. BVHs are efficient in processing proximity queries, with $O(logN)$ time. Their construction of $O(NlogN)$ is also considered efficient, since for static objects the structure is constructed only once at set up. However, updating a BVH of an entirely dynamic data set is also of $O(NlogN)$ time. Therefore, for data sets with continuous temporal updates as we consider in this paper, BVHs suffer from inefficiency, whether they are updated or constructed from the start with every update.

Voxelization. Our work is related to scene voxelization approaches. Eisemann and Decoret [6] utilized the capabilities of the GPU to construct voxel-based representations which need not be aligned in one (i.e., the view) axis but are restricted to a fixed number of constant-size intervals, while Hinks et al. [11] use a similar representation to construct solid models for computational modeling. An older approach [4] also reconstructs a sampled surface implicitly at grid cells using a signed distance function. We compute discrete screen-space aligned layers instead, each layer represented by non-aligned depth ranges.

Image-based techniques for collision detection. Such techniques [14, 9, 10] do not require any pre-processing, and thus are appropriate for dynamically deforming objects. In [10], layered depth images (LDI) are computed for both objects, then volume representations are constructed and compared to find intersection regions. The algorithm, and almost all image-based collision detection (CD) algorithms as well, targets triangulated meshes. To our knowledge, only one approach [1] uses image space to detect collision between point clouds, but is restricted to movement in 2.5D space. They divide the space into slices and compute a height map for each. The approach assumes that obstacles are nearly parallel to YZ plane and perpendicular to XY plane, and uses these assumptions to infer obstacle information and save them with each pixel.

Static point-cloud collision detection. The most important approach [12] is both robust to noise and fast (interactive if need be, depending on the time budget). They construct a BVH and use a collision probability measure between pairs of nodes, in order to traverse the two objects' hierarchies ordered by priority. In the second stage, they sample the implicit surface at the leaf nodes to measure separation distance. However, as mentioned before, construction of a BVH is slow and can be memory-intensive, and the entire data needs to reside in memory as well. Thus it is impractical for large point sets and even detrimental to build such a structure for points sets which change dynamically and are not queried often enough to amortize its building cost. Our approach targets different application scenarios, but it surpasses the accuracy achieved by Klein's algorithm [12], as is shown in Section 7. Pan et al. [16] robustly detect collisions between noisy point clouds by defining the detection as a two-class classification problem and estimate collision probability with support vector machines, but runtime is comparatively slow. Our approach hides sensor noise by querying a thickened boundary.

Dynamic point-cloud collision detection. A very recent paper [17] uses BVHs and/or octrees to detect collisions and compute distances between sensor-captured point clouds. They propose two ideas, one is appropriate for static environments and the other for dynamic ones. For dynamic environments, they propose to mutually traverse an octree (environment point cloud) and an AABB (robot), and do an unspecified, probably simple collision test at leaf nodes. Although this approach for dynamic environments is simple, we avoid building a spatial hierarchy at all and can keep the GPU pipeline more occupied by streaming coherent data.

3 Overview

Our input data are *unstructured points*. We assume that the rendering pipeline has already culled points against the view frustum (or respective bounding box) and projected them into screen space. Further we assume that the *sampling density* for the individual input points is given, either globally uniform or, e.g., estimated from sensor device properties.

In Section 4 we define a *thickened boundary* which envelops the implicit surface of the point cloud, provided that the sampling is sufficiently dense. We then transform it into projected space and finally discretize it in screen space, adapted to the view point. Based on this representation, we show how the contained implicit surface can be queried quickly. Then we explain in Section 5 how we efficiently construct this boundary representation in parallel as a *thick layered depth image* extended with depth range (TLDI). For this we show reuse of several parts of a standard point-based rendering pipeline. We describe in Section 6 that detecting collisions by querying the surface in this TLDI data structure is straightforward to do. In Section 7 we compare our collision detection algorithm with sampled meshes as ground truth and show that it is significantly more precise than prior methods, robust in the presence of noise and fast enough to handle dynamic point clouds at *interactive frame rates*. We give concluding arguments in Section 8 along with an outlook to the extensions we are currently working on.

4 Surface Definition

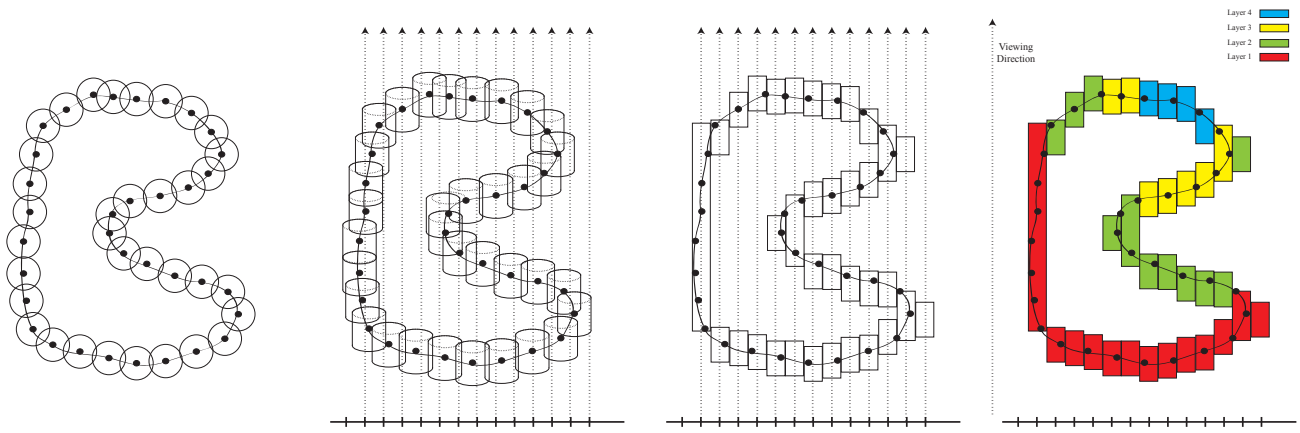
Our goal is to determine the distance of a point $p \in \mathbb{R}^3$ to the manifold, possibly bounded surface Σ that is implicitly defined by a set of points S, sampled on or close to it. All distances are in the Euclidean sense, unless otherwise noted. Since Σ is not known, we first define a *thickened boundary* Ω that contains such a surface near S, similar to an adaptive *spherical cover* as proposed in [15]. For precise evaluation of proximity queries to Σ we require that Ω bounds it as closely as possible, but also want to avoid holes in Ω that are not present in S. Evaluating the distance $\|p,\Omega\|$, which in turn allows us to approximate $\|p,\Sigma\|$, requires representation of Ω by a discrete spatial structure. Our design requirements are that it is compact and can be both constructed and evaluated quickly.

4.1 Spherical Cover Ω Containing the Surface Σ

First, we want to define a volume Ω which covers the surface Σ underlying the samples so that we can perform distance queries to Σ. Let $B_i(s_i,r_i)$ be the balls centered at samples $s_i \in S$ in $\mathbb{R}^3$ with radii r_i chosen such that Σ is enclosed entirely in the union of balls Ω (see Figure 2a):

$$\Omega = \bigcup_{i=0}^{N} B_i(s_i,r_i)$$

If S is sampled non-uniformly densely, balls which are close but from geodesically remote parts of the surface may merge in Ω, and then Ω is not homeomorphic to Σ. This is not a problem for our



(a) Spherical cover (b) Cylinders (c) Blended and discretized (d) TLDI

Figure 2: Representations of the volume bounding the surface Σ: a) Union of balls Ω centered at samples. b) Projected onto the view plane as cylinders in object space Ω'. c) Blended depth intervals $\hat{\Omega}$. d) TLDI shaded per layer.

use case, since determining the distance from a point $p \in \mathbb{R}^3$ to a surface neither requires that surface to be manifold nor orientable.

If the radii r_i associated with the samples are just sufficiently large with respect to local sampling density, the B_i will overlap such that Σ is entirely contained in Ω. We assume r_i either to be a global constant, estimated from range image properties or determined in preprocessing, as for out-of-core huge point clouds [19]. Alternatively, r_i could be estimated locally by determining k-nearest neighbors in screen space, as shown in [18]. Note that real holes in the surface which are smaller than r_i could disappear in the representation.

Since Ω consists of balls, its thickness perpendicular to Σ will be large and oscillate considerably between samples. Determining the connectivity between samples would allow us to blend their balls and result in a more equally thickened boundary. As mentioned above, inserting the balls into a spatial hierarchy in $\mathbb{R}^3$ to recover the connectivity is slow because we have to sort in three dimensions. Instead, we show how to achieve this more efficiently in projected (2-dimensional) space, which has something in common with splat rendering, as described next.

4.2 Blending Cylinders in Projected Space

We define samples in S as connected if their balls overlap. Now we want to locate the connectivity between the samples so that we can blend their associated balls for neighbors to equalize boundary thickness. This is easier if we project them from $\mathbb{R}^3$ onto a plane. Then we just need to locate overlapping disks in that plane and check if they also overlap in depth with their radii, similar to rendering view-plane aligned splats. In object space this represents testing plane-parallel cylinders which contain the balls and are of minimum size (see Figure 2b). We name the union of cylinders Ω'.

Each point $\hat{\mathbf{x}}$ in the projection plane (i.e., the view plane) represents a view ray in object space and may intersect Ω' multiple times. Therefore each $\hat{\mathbf{x}}$ maps to a set of depth ranges (entry-exit point pairs of Ω') which we call its *layers*, represented by the function $F_i(\hat{\mathbf{x}})$ for layer i:

$F_i(\hat{\mathbf{x}}) = \{d_{i,near}, d_{i,far}\}$

We want to equalize the boundary thickness of Ω' since its associated values of $F(\hat{\mathbf{x}})$ change discretely at cylinder boundaries. So we blend its values (both *near* and *far*) for the N connected samples $\mathbf{s}_i$ whose cylinders overlap with the corresponding entry-exit pair along the view ray of $\hat{\mathbf{x}}$ as follows:

$$\hat{F}_i(\hat{\mathbf{x}}) = \sum_{i=1}^{N} d_i r(\|\mathbf{x} - \mathbf{s}_i\|)$$

where $r(x) = e^{x^2}$. We call $\hat{\Omega}$ the volume defined by the depth range layers of $\hat{F}$.

In regions where the surface is mostly parallel to the view plane, the set of cylinders intersected by a view ray in one layer is such that each cylinder overlaps with each other in that set. Where the surface is oblique, this may not hold because the depth range of a layer becomes large. We call such a set of cylinders containing non-overlapping subsets as *stacked*. For such stacks, we blend the frontmost cylinder only with its overlapping cylinders in the stack to get $d_{i,near}$, and similar for the backmost cylinder to get $d_{i,far}$. Figure 3 explains the two cases.

Σ is not known but implicitly assumed through its set of samples S. Nevertheless, we would like Σ to be bound by $\hat{\Omega}$, so we attempt to define it to lie centered in $\hat{\Omega}$. The way in which Σ approximates S can then be thought of as similar as a blended surface of splats. Our results in Section 7 confirm that $\hat{\Sigma}$ is reasonably close to S.

We would like to define $\hat{\Sigma}$ as the set of centers of maximum balls contained in $\hat{\Omega}$ which touch both sides of its boundary. However, boundary sides of $\hat{\Omega}$ are not clearly defined, but shooting view rays through it results in entry/exit pairs. Based on that information we can define $\hat{\Sigma}$ as the set of centers of maximum balls contained in $\hat{\Omega}$ which are centered along a view ray and growing monotonically either from its entry or exit point. A view ray then contains for each layer $\hat{F}_i$ either one or two balls. $\hat{\Sigma}$ is similar to a subset of the medial axis [3] of $\hat{\Omega}$ as the maximizing of balls along the view ray prunes spurious branches in that direction. However, it may contain spurious branches in the other axes.

4.3 Discretization of $\hat{\Omega}$ in Screen Space

For efficient spatial sorting, we discretize $\hat{\Omega}$ into a 2D grid with screen-space resolution. This data structure is well suited to parallel processing as the point primitives are streamed onto the GPU and connectivity has local extent in screen space so there is not much interdependency.

The result is a kind of non-aligned voxelization, since each pixel can reference multiple layers in $\hat{F}$, but their depth range does not

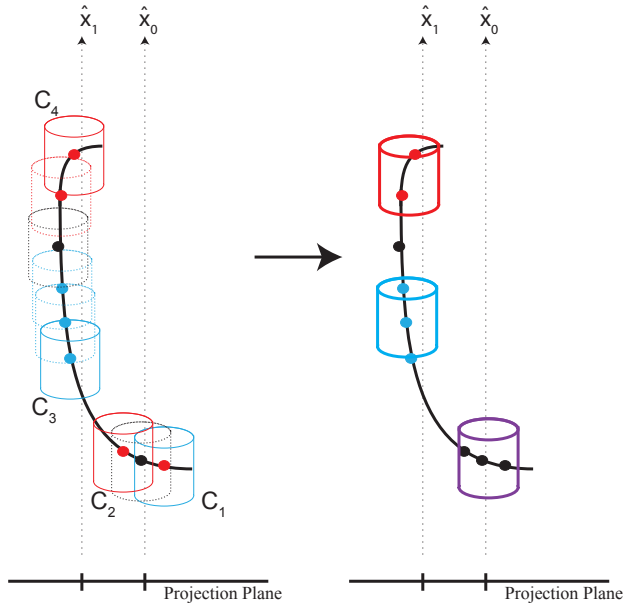


Figure 3: Left: This figure shows blending at stacked ($\hat{x}_1$) and non-stacked ($\hat{x}_0$) view rays. Frontmost (cyan) and backmost cylinders (red) are drawn with continuous lines, other cylinders as dashed. View ray $\hat{x}_0$ enters the layer at cylinder C_1 and leaves at C_2, intersecting three cylinders in total which all overlap and thus are blended together to a single d_i. View ray $\hat{x}_1$ on the other hand intersects a stack of cylinders, C_3 in the front and C_4 at the back. $d_{i,near}$ is then the result of blending C_3 with its overlapping (light cyan) cylinders and $d_{i,far}$ similar for C_4. Right: the result of blending are the cylinders drawn with thick stroke.

correspond between pixels (see Figure 2c). A closely related concept are *layered depth images* (LDI), which are typically used to peel off surface layers from a mesh as shown in [7]. In our case, layers represent depth ranges instead of scalar values, so we extend the depth of an LDI with a second value to represent the near (entry) and far (exit) intersection of the thickened boundary. We name this a *thickened LDI* (TLDI).

5 Construction of TLDI

Constructing the TLDI for a point cloud peels off layers similar as does depth peeling for a mesh (see Figure 2d). Since operations such as visibility culling, blending and normalization are involved, we can partially reuse work already done in the standard PBR pipeline which processes the points sequentially:

The Standard Three-Pass PBR Pipeline. The common pipeline of surface splatting employed by PBR algorithms is generally composed of three shader passes:

- *Visibility Pass:* All splats are simply rendered, depth culled, leaving only the front-most fragments in the output buffer.
- *Blending Pass:* Fragments of the splats that are within a certain threshold from the front depth values are rendered to accumulate the weighted colors and the weights themselves.
- *Normalization Pass:* The accumulated weighted colors value is divided by the accumulated weights value to get the blended depth.

For the blending and normalizing passes, we simply replace values of color with depth (front and back values respectively).

Modifications for TLDI layer computation:

For constructing a layer of the TLDI, we insert three passes between the visibility and blending pass:

- *Stacking Pass:* Since points are not processed in order, cylinders in a stack may occur after each other such that they do not overlap. We maintain a zero-initialized bit array for an assumed optimal stack size of size 128 bits, 32 bits for each one of the RGBA channels, quantized by the radius of the first encountered point. Subsequent cylinders encountered at that pixel and inside its range fill up the bits corresponding to their depth (see Figure 4).
- *Counter Pass:* The number of contiguous filled bits is determined, starting from the first filled bit.
- *Back Visibility Pass:* The previous count determines the backmost cylinder in that stack and also in the current layer.

The two pipelines are displayed in Figure 5. For each pixel in the TLDI, a pair of depth values (d_{near}, d_{far}) is output. In our implementation, we actually store their average d_{avg} along with half their distance, because for non-stacked pixels, d_{avg} already represents Σ.

In our experiments, we managed to capture all layers entirely within our assumed stack size of 128 bits. However it is important to note that layers exceeding this size would simply be split up into two, adding another layer to the data structure but not changing the underlying representation. We expect this to minimally decrease performance, but accuracy would not be affected.

We execute the above pipeline for each layer of the point cloud, however the collision detection application that we present next often terminates already after a single layer has been constructed.

6 Collision Detection as an Application

We now present collision detection as one application of querying the TLDI representation of the implicit surface of S. We show that it can be implemented efficiently by merging TLDI construction and collision testing into an existing PBR pipeline.

Simply put, collisions are detected by intersecting view rays from the camera for each pixel with the TLDI for each point cloud and testing if their depth ranges along that ray (since close to $\hat{\Sigma}$) intersect. For non-colliding point clouds, this also infers the separation distance in view direction, which especially makes sense for an object moving with the camera, such as an avatar. We describe next how the two point clouds' collision and distance queries are processed.

6.1 TLDIs Comparison

Comparisons between layers are performed pixel wise. A collision is detected if at a pixel the depth ranges for layers from two objects overlap. Since the boundary is thickened, we expect a number of false positives, i.e., $\hat{\Omega}_0, \hat{\Omega}_1$ of the point clouds intersect while the actual surfaces $\hat{\Sigma}_0, \hat{\Sigma}_1$ do not. In our experiments we discovered that we could compress the thickness of $\hat{\Omega}$ in view direction by a significant factor in order to eliminate most false positives while keeping the number of false negatives small. Since $\hat{\Omega}$ is projected in view direction, thinning it in this axis does not affect the general observations made in Section 4, in fact it approximates Σ more closely.

Since a collision may already be detected in the first layer (which terminates our method), we do not have to construct all layers of the object and compare them against each layer of the other object. Instead, we compute them in depth order on demand as long as no collision is detected, as outlined below. This limits the number of

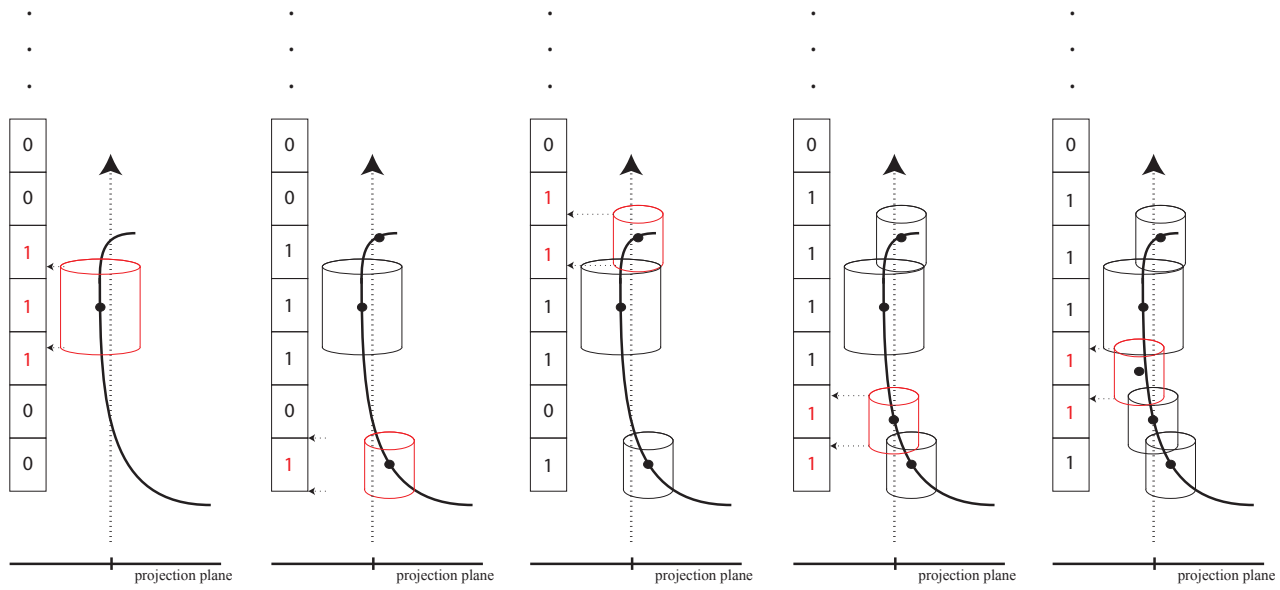


Figure 4: The figure demonstrates how the connectivity of a stack inside a layer is tracked by a bit array in the *stacking pass*. Depth is quantized into segments, where each segment is of height equal to the diameter of the first encountered cylinder in the initial *visibility pass*, and the first segment aligned to its lower disc. In the subsequent *stacking pass*, cylinders of encountered points are projected to gain occupancy information, as shown in order. Each cylinder fills the segments in the buffer bit that intersects with its cylinder depth interval. The order in which points are projected is assumed to be random, and cylinders in the figure are not all the same size.

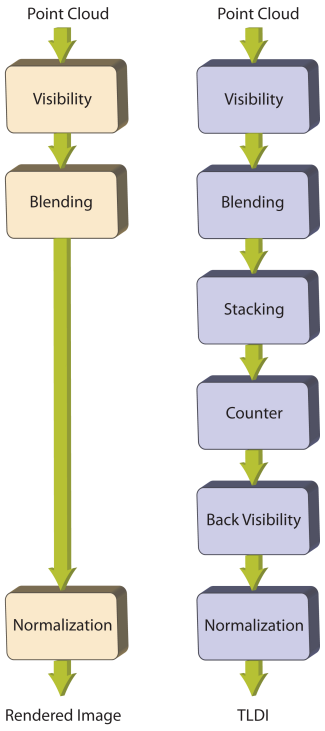


Figure 5: For TLDI construction we insert three additional passes into a standard PBR pipeline.

depth comparisons by the number of layers per object, corresponding to its view-dependent depth complexity.

Let there be two point clouds O_0, O_1. For O_0, the first and second layers – $l_{0,0}$ and $l_{0,1}$ – are computed. Then we consider the subset of O_1 which is clipped by the depth range of – and between – $\hat{\Omega}$ of $l_{0,0}$ and $l_{0,1}$, construct its layers and compare it against those of O_0. The clipping is repeated similar between subsequent layer pairs $l_{0,i-1}$, $l_{0,i}$, for $i \geq 2$, until the backmost layer of O_0 has been reached or a collision is detected.

6.2 Early Rejection Test

Since entire TLDIs have to be constructed in case of non-collision, we add a quick rejection test in the beginning, where we simply compare the front layer of O_0 to the back layer of O_1, and vice versa. Since the front layer is the closest to the camera, a non-collision is reported if for all pixels d_{near} of the front layer of O_0 is greater than d_{far} of the back layer of O_1. If the depth intervals of those layers intersect, a collision is detected early as well. This test can be performed quickly inside the standard PBR pipeline, as only one depth interval is required to be blended. If the test does not deliver any result, we continue the normal procedure of comparing the layers, in which the already computed layers can be reused.

6.3 Integration into Existing PBR Pipeline

The overlap between the standard PBR pipeline and TLDI construction suggests an integration between those two. The early rejection test already uses the same standard pipeline of rendering to create the two front layers. In fact, the only difference is what is being blended, color or depth. However, in rendering the two objects are processed and z-culled together, while TLDI construction processes one object at a time. The preferred integration scenario is to blend both colors and depth while creating the front TLDIs of O_0 and O_1. While the TLDIs are used for collision detection, the two frames with the blended colors can be merged into one by performing z-culling at each pixel using the corresponding TLDIs to choose which color value is copied to the merging frame pixel.

We note that TLDI construction profits from the reuse of PBR pipeline calculations for point-cloud data located inside the view frustum. For many application scenarios, only these data are of interest anyway, e.g., an avatar moving with the camera.

Point clouds are almost always perspectively projected onto the screen by rendering pipelines, whereas the cylinders in sections 4, 5, 6 are assumed to be orthogonally projected. The integrated pipeline has to use the same projection for both rendering and TLDI and so we use perspective projection in our experiments. This results in perspective foreshortening of the cylinders and thus turns them into truncated cones. However, since the resulting surface is only an approximation, accuracy is not affected significantly as the results in Section 7 confirm.

6.4 Distance Queries

In addition to collision queries, our method can also incidentally answer distance queries in case of non-collision. Since we do not consider $\hat{\Sigma}$ as orientable, the distance function we calculate with respect to it is unsigned. We can therefore not decide if the distance between two objects is one of separation or of penetration. When layers $l_{0,i}, l_{1,j}$ are compared to determine a collision, their absolute depth difference per pixel is calculated as follows:

$d(\hat{\mathbf{x}}) = min(|\hat{F}_i(near) - \hat{F}_j(far)|, |\hat{F}_j(near) - \hat{F}_i(far)|)$

We keep track of $d_{min}(\hat{\mathbf{x}})$ at each pixel and then report its overall minimum in case of non-collision, which yields the separation distance in view direction.

7 Results

The algorithm is implemented in C++, OpenGL and GLSL. Tests were run on Core2 Quad processor, 2.4 GHz, with 4 GB RAM, and GeForce GTX 680 graphics processor.

We used the benchmark proposed by [21] for testing, in which two copies of the same model are tested for collision against each other. Both objects are normalized to fit in a 2^3 cube. The center of one object is positioned at the origin, and the center of the second is positioned at a distance d_0 from the origin along the +x direction. The second object is compositely rotated about the y-axis and the z-axis, with a number of small steps. The composite rotations are iteratively repeated, each iteration starting from a position at a distance $d_i = d_0 - i\Delta d$ from the origin. We initialized d_0 with 3.0, 31 iteration/distance, and 30 steps per rotation, which makes 900 cases per iteration/distance, and 27900 total cases. The camera is positioned at coordinates $(0,0,+4)$, and facing the origin. Point clouds are perspectively projected with view angle 90° at planes $z = +1$ and $z = +6$ from the camera position.

Accuracy and runtime results are computed for each distance by averaging the results of all steps in the corresponding rotation. Accuracy tests are performed by comparing our outcomes with those of an exact mesh collision procedure. We call this percentage the accuracy error, but it should be noted that *it is not actually an error*. The polygonal mesh of a point cloud is an approximation of the surface, but not the surface itself. So, we consider the mesh collision test results as an *approximation of the ground truth*.

Four models were used for testing: Stanford BUNNY, ARMADILLO, DRAGON, and HAPPY polygonal models. For each model, the point set has been extracted from the mesh and the r_i for its points determined by *kNN* with $k = 7$ for testing purposes, where r_i is set to 0.65 of the computed distance. An efficient screen-space method to determine a radius containing *kNN* has been demonstrated in [18]. They project cylinders onto a rather large frame, 1024×1024, to achieve accurate results.

Besides synthetic models, we also tested collision detection between the huge data set of the houses of EPHESUS (> 5M points) captured by a laser scanner (see Figure 6), and a single HAPPY model. We tried to imitate an interactive navigation experience by considering the HAPPY model a human discovering the big model.

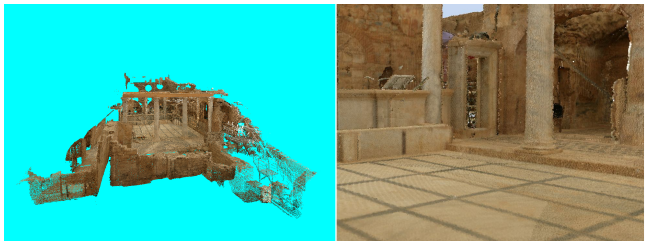


Figure 6: The houses of EPHESUS. Left: View from above of the point cloud. Right: Closer view with splat rendering.

Table 1: The percentages of false negatives and false positives with various compression ratios ρ for: **B**UNNY, **A**RMADILLO, **D**RAGON, **H**APPY.

ρ	false negatives (%)				false positives (%)			
	B	A	D	H	B	A	D	H
1.0	0	0	0	0	0.92	0.47	0.47	0.12
0.5	0	0	0	0	0.68	0.29	0.29	0.08
0.25	0	0	0	0	0.52	0.23	0.21	0.03
0.1	0	0.1	0.01	0	0.43	0.18	0.15	0.02
0.05	0	0.01	0.02	0.01	0.39	0.17	0.14	0.02
0.01	0.01	0.03	0.05	0.05	0.33	0.16	0.13	0.01
0.005	0.05	0.04	0.06	0.06	0.3	0.16	0.13	0.01

The same benchmark described above is used, where EPHESUS keeps its size, HAPPY scaled to the size of a human – relative to EPHESUS – and the view emanates from the eyes of HAPPY and directed forward. HAPPY is initially positioned at coordinates $(0,0,+D)$, where D is the x-extent of the EPHESUS bounding box. Number of iterations and number of steps per iteration are the same as above. A polygonal mesh of EPHESUS is not available, so only runtime is measured.

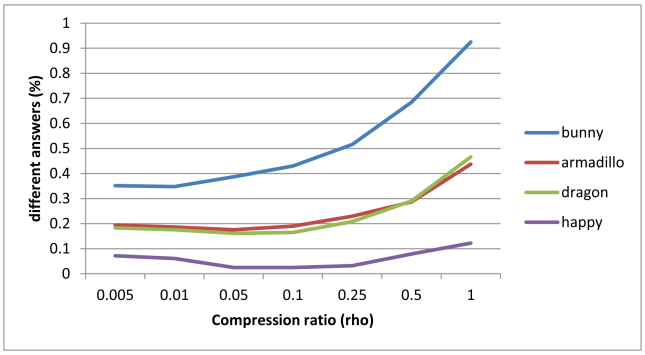


Figure 7: Accuracy error of collision detection compared with the sampled mesh as approximate ground truth. Based on these results we chose $\rho = 0.05$ for subsequent tests.

When querying the intersection between thickened bounding intervals as an indicator of collision, we have encountered very few cases of false negatives in our experiments (none for most object pairs tested), but the ratio of false positives is rather high (see Table 1). Compressing the intervals as mentioned in Section 6.1 with a factor ρ decreases this number effectively while yielding only an insignificant number of false negatives. The accuracy resulting from different compression values is plotted in Figure 7, which shows that accuracy peaks for ρ between 0.01 and 0.05. Smaller values of ρ yield less false negatives, but more false positives, which results in less overall accuracy. Based on that, we chose $\rho = 0.05$ for all

Table 2: Point clouds with total TLDI construction time (in millisec). CD runtime, time overlap with rendering, and error from mesh ground truth are averaged following the benchmark of Section 7.

Model	Size	TLDI	CD	Render	Error
Bunny	36k	13.9	2.2	0.9	0.39 %
Armadillo	173k	43.7	8.1	3.1	0.18 %
Dragon	438k	122.5	15.8	6.7	0.16 %
Happy	544k	134.9	18.3	8.2	0.03 %

following tests, to maximize accuracy (= minimizing sum of false negatives and false positives).

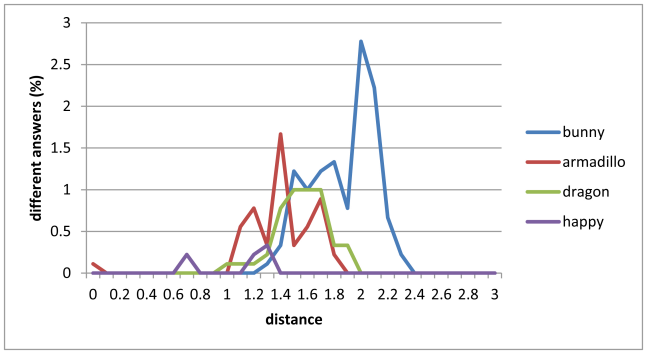


Figure 8: Accuracy error vs distance for all models.

Accuracy

Figure 8 shows a plot of accuracy error against distance d_i. For all models, the error is zero when the two objects centers are either far or close, and increases in between, where the object surfaces collide. The accuracy error stays below 3% for all models for any distance. Table 2 shows accuracy error averaged over distances, and is always below 0.4%. Note that false negatives result from $\hat{\Omega}$ not covering $\hat{\Sigma}$ entirely. This can occur if either ρ is too small, overly compressing $\hat{\Omega}$, or if radii are estimated too small.

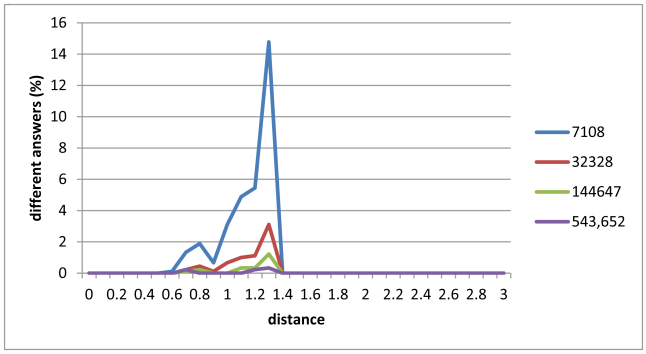


Figure 9: Accuracy error for different resolutions of HAPPY.

We are particularly satisfied by the accuracy results of our method. In Figure 10, [21] showed accuracy error of different resolutions of HAPPY. We reproduce this plot based on the same benchmark in Figure 9, albeit with the different resolutions of HAPPY which were available to us. Interestingly their accuracy do not improve much when increasing sampling density (always $< 7\%$). Our results improve significantly with increasing sampling density as we expect the TLDI to approximate the surface better, down to $< 0.3\%$ for the original resolution.

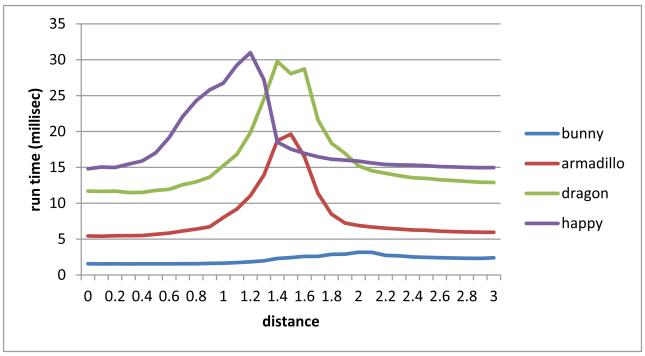


Figure 10: Runtime vs distance.

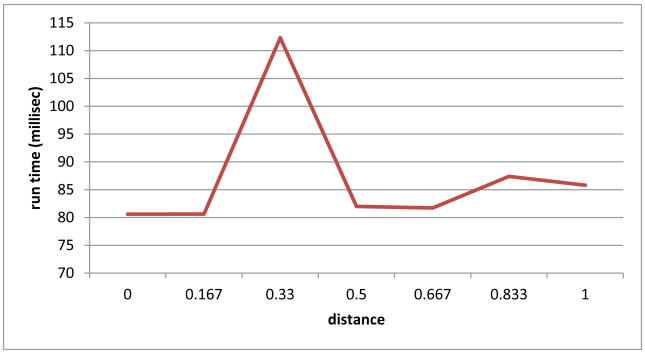


Figure 11: Runtime for colliding large EPHESUS (5M points) with HAPPY averaged over different distances (using benchmark). The numbers on the x-axis are the distances normalized to $[0, 1]$.

Runtime

The result of the benchmark shows that runtime increases approximately linearly with point cloud size (see Table 2). The same table also shows the large overlap of collision detection with the rendering pipeline: about 40% of collision detection runtime is removed if the point clouds are rendered as well. Colliding the large EPHESUS model with HAPPY is still possible at interactive frame rates (see Figure 11). Similar to accuracy, runtime also decreases for near or far object centers and increases in between where surfaces collide, as Figure 10 demonstrates.

Figure 12 confirms that the early rejection test outlined in Section 6.2 reduces runtime significantly as well.

The runtime of TLDI construction is directly proportional to point cloud size, and number of captured layers, whereas the number of layers in turn depends on how tight the TLDI is. The more the TLDI adheres to the actual surface, the more layers are captured and the longer the construction time. TLDI tightness is controllable via scaling the splats radius and the frame resolution. For collision detection, it is necessary to construct a tight TLDI to achieve accurate results, but this is balanced by the fact that it is not necessary to construct the whole TLDI as explained in Section 5. For other applications where the TLDI functions as a bounding volume rather than a surface estimator, the construction time can be traded off with the tightness level. Table 3 shows the runtime of full TLDI construction for HAPPY and DRAGON models, against different frame resolutions and splat scales. The table shows the accuracy error values as well. The smallest error is achieved by a radius scale of 1.0 and frame resolution of 1024×1024. There is no specific rule how accuracy error changes with the two parameters, but the trend is that it decreases as the radius scale increases.

Comparing the runtime of our method to others, e.g. [21], is of

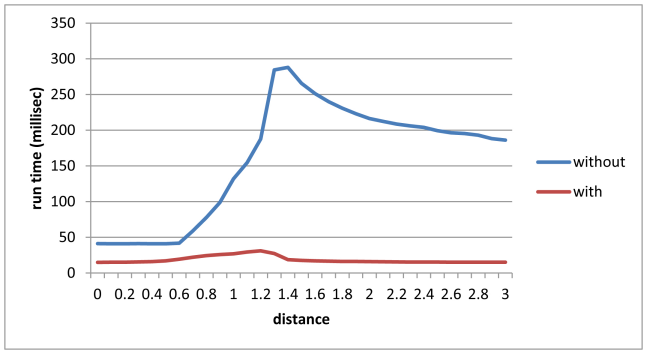


Figure 12: Runtime for HAPPY (using benchmark), without (blue) and with (red) the early rejection test. It clearly shows that it increases efficiency significantly.

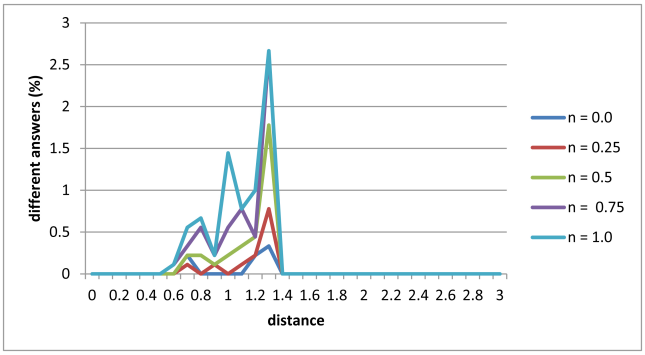


Figure 13: Accuracy error for different levels of uniform Gaussian noise ($\sigma = nr_{avg}$) added to HAPPY.

Table 3: The average TLDI construction time for HAPPY and DRAGON models at different frame resolutions and splat scales. Average number of captured layers are denoted inside brackets, and accuracy error values are denoted below in italic. The values on top of the columns indicate the scaling value of the splat radius. The numbers show how the TLDI construction time decreases as the frame resolution decreases and the splat radius scale increases.

Model	Resolution	1.0	3.0	5.0
Happy	1024×1024	134.9(10.4)	88.2(6.9)	67.5(5.4)
		0.16%	*0.79%*	*1.9%*
	512×512	121.8(9.5)	85.3(6.8)	68.6(5.6)
		0.18%	*0.67%*	*1.7%*
	256×256	113.4(9.0)	75.9(6.2)	63.2(5.1)
		0.71%	*0.57%*	*1.4%*
Dragon	1024×1024	122.5(11.5)	98.2(9.1)	93.4(6.4)
		0%	*0.20%*	*0.24%*
	512×512	116.2(11.1)	94.2(9.0)	64.6(6.2)
		0.16%	*0.19%*	*0.63%*
	256×256	108.3(10.1)	89.9(8.7)	62.5(6.2)
		0.67%	*0.25%*	*0.42%*

limited usefulness. Their algorithm was designed for static objects, as the underlying structure takes considerable time to construct. For those pre-processed data structures it performs collision queries faster than ours (being of $O(logN)$ versus our $O(N)$ complexity), but for dynamically changing objects, hierarchy construction time needs to be added to each query, and for that our algorithm is faster by orders of magnitude, even considering that they were measured on older hardware. The construction of their underlying BVH may become faster if parallelized and performed on modern hardware. The tested models contain about 1M points. For objects of that size, a BVH-based algorithm of $O(NlogN)$ complexity would require an extra time factor $O(logN)$ of 20, which is quite large.

Robustness to Noise

Since the boundary of Ω is thickened to the extent of sampling density, we expect it to smooth noise up to a similar level. We tested the robustness of our approach by adding Gaussian noise with different σ to HAPPY, the most densely sampled synthetic model used in our tests. We set $\sigma = nr_{avg}$, where r_{avg} is the average over r_i, and random $n = [0,1]$. Runtime did not change significantly and accuracy error was always below 3% (see Figure 13).

Real Data and Dynamic Simulation

Point clouds captured with the Kinect often exhibit noise and holes, as ROOM (300k points, captured by Kinect) shown in Figure 14a. Figures 14b-d show snapshots of collision detection between BUNNY and ROOM. Collisions are robustly detected near flat surfaces and small holes. We also simulated a dynamic environment of an animated model (HORSE) (10 frames, 8.5k points each) traversing the EPHESUS model. Figure 15 shows snapshots from the simulation. BUNNY and HORSE are rendered as meshes in the figures for visual plausibility.

7.1 Complexity Analysis

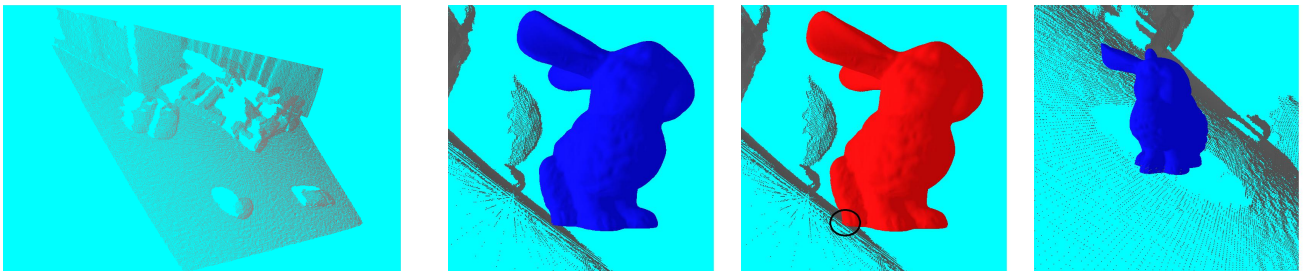
Worst-case time complexity between pairs of objects occurs only if there is no collision and the early rejection test does not detect that. An example is an object that is partially obscured from the view point by a concavity in the other object. This requires construction of all TLDIs per point cloud and comparing all of those for one point cloud against the subsets of TLDIs clipped between them. TLDI construction is linear in the size of the point clouds, with the added factor of depth complexity, as points are processed in order for each layer and therefore $O(LN)$.The collision test is output-sensitive with $O(LXY)$ for screen space resolution $X \times Y$ and scales with the depth complexity of the point cloud being clipped.

For collision detection among a set of more than two point clouds, using the proposed algorithm would make the overall runtime (both construction and collision detection) quadratic, as the construction of a cloud TLDI is dependent on the other cloud TLDI and thus would be reconstructed for each comparison. However, if the number of clouds is large and the size of each cloud is relatively small, we could also construct the TLDI of each point cloud just once and separately. The then linear TLDI construction time has the trade-off that the previously linear time of collision detection becomes $O(L_1L_2XY)$.

If we compare a single large point cloud (environment) against multiple small ones (avatars), we could also use another approach. In that case, all avatars are treated as a single combined point cloud, and the same complexity of a single pair comparison holds. Increasing the number of avatars in that scenario increases N_2 in the above expression, and therefore construction time increases linearly. In order to know which avatars collide with the environment, labels at the points would have to be stored as well, which would result in a small increase in memory storage.

8 CONCLUSION AND FUTURE WORK

We have proposed a novel data structure for representing the surface of dynamic point clouds. We show that it can be constructed efficiently and reuse computation from an existing PBR pipeline. As an application we have demonstrated online collision detection for large models. Our results show that our surface extraction is significantly more precise than for a previous method [12], especially where points are densely sampled, and that it is also robust to noise since the surface underlying the points is thickened.



(a) (b) (c) (d)

Figure 14: Collision detection between BUNNY and the (a) Kinect captured ROOM. BUNNY is blue in cases of non collision, and turns red in cases of detected collisions. (b) shows BUNNY near a flat surface, and crosses it in the next frame (c). Both cases are correctly detected. BUNNY passes through a wide hole in (d) which is not recognized as part of the surface, and thus no collision is detected.

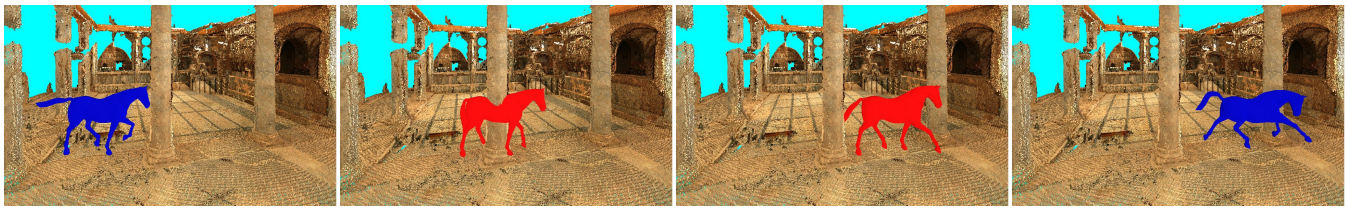


Figure 15: Animated HORSE inside EPHESUS, passing through a column.

We are currently working to improve our data structure in terms of compactness and efficiency of construction and traversal. Implementation of the more exact surface extraction for stacks would even further increase accuracy, since we currently simply assume the center of the depth range of a layer along a view ray to be the surface intersection. We think that augmenting the TLDI with data from sampling such as normals and uncertainty information could permit even more precise surface extraction. TLDI could also be used instead of a voxelization as a more compact representation, for example to accelerate global illumination computations.

ACKNOWLEDGEMENTS

This research was supported by the EU FP7 project HARVEST4D (no. 323567).

REFERENCES

[1] R. K. Anjos, J. M. Pereira, and J. F. Oliveira. Collision detection on point clouds using a 2.5+d image-based approach. *J. of WSCG*, 20(2):145–154, 2012.

[2] G. V. D. Bergen. Efficient collision detection of complex deformable models using AABB trees. *J. of Graphics Tools*, 4(2):1–14, 1997.

[3] H. Blum. A Transformation for Extracting New Descriptors of Shape. In W. Wathen-Dunn, editor, *Models for the Perception of Speech and Visual Form*, pages 362–380. MIT Press, Cambridge, 1967.

[4] B. Curless and M. Levoy. A volumetric method for building complex models from range images. *Proc. SIGGRAPH*, pages 303–312, 1996.

[5] S. A. Ehmann and M. C. Lin. Accurate and fast proximity queries between polyhedra using convex surface decomposition. *cgforum*, 20:500–510, 2001.

[6] E. Eisemann and X. Dècoret. Fast scene voxelization and applications. *ACM SIGGRAPH Symp. on Interactive 3D Graphics & Games*, pages 71–78, 2006.

[7] C. Everitt. Interactive order-independent transparency. Technical report, NVIDIA, 2001.

[8] S. Gottschalk, M. Lin, and D. Manocha. OBB-tree: A hierarchical structure for rapid interference detection. *SIGGRAPH 96 Conf. Proc.*, pages 171–180, Aug 1996.

[9] N. K. Govindaraju, M. C. Lin, and D. Manocha. Fast and reliable collision culling using graphics hardware. *Vis. and Computer Graphics, IEEE Trans. on*, 12(2):143–154, Mar-Apr 2006.

[10] B. Heidelberger, M. Teschner, and M. H. Gross. Detection of collisions and self-collisions using image-space techniques. In *J. of WSCG*, volume 17, pages 145–152, 2004.

[11] T. Hinks, H. Carr, L. Truong-Hong, and D. Laefer. Point cloud data conversion into solid models via point-based voxelization. *Surveying Engineering*, 139(2):7283, 2013.

[12] J. Klein and G. Zachmann. Point cloud collision detection. In *Eurographics 2004*, volume 23, pages 567–576, Sep 2004.

[13] J. T. Kloswski, M. Held, J. S. B. Mitchell, H. Sowrizal, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Trans. on Vis. & Com. Graphics*, 1(4):21–36, Jan 1998.

[14] D. Knott and D. K. Pai. Cinder: Collision and interference detection in real-time using graphics hardware. In *Graphics Interface*, pages 73–80, May 2003.

[15] Y. Ohtake, A. Belyaev, and H.-P. Seidel. An integrating approach to meshing scattered point data. In *Proc. of 2005 ACM symp. on Solid & physical modeling*, pages 61–69. ACM, 2005.

[16] J. Pan, S. Chitta, and D. Manocha. Probabilistic collision detection between noisy point clouds using robust classification. *Int. Symp. on Robotics Research*, 2011.

[17] J. Pan, I. A. Sucan, S. Chitta, and D. Manocha. Real-time collision detection and distance computation on point cloud sensor data. In *IEEE Int. Conf. on Robotics & Automation*, pages 3593–3599, 2013.

[18] R. Preiner, S. Jeschke, and M. Wimmer. Auto splats: Dynamic point cloud visualization on the gpu. In H. Childs and T. Kuhlen, editors, *Proc. of Eurographics Symp. on Parallel Graphics & Vis.*, pages 139–148. Eurographics Association 2012, may 2012.

[19] C. Scheiblauer and M. Wimmer. Out-of-core selection and editing of huge point clouds. *Computers & Graphics*, 35(2):342–351, Apr 2011.

[20] D. Steinemann, M. Otaduy, and M. Gross. Efficient bounds for point-based animations. *Symp. Point-Based Graphics*, pages 57–64, 2007.

[21] G. Zachmann. Minimal hierarchical collision detection. In *ACM Symp. on Vir. Reality Software and Tec.*, pages 121–128, Nov 2002.