Eurographics Symposium on Point-Based Graphics (2005), pp. 1–8
M. Pauly, M. Zwicker, (Editors)

Interactive Ray Tracing of Point-based Models

Ingo Wald Hans-Peter Seidel
MPI Informatik, Saarbrücken, Germany
{wald,hpseidel}@mpi-inf.mpg.de

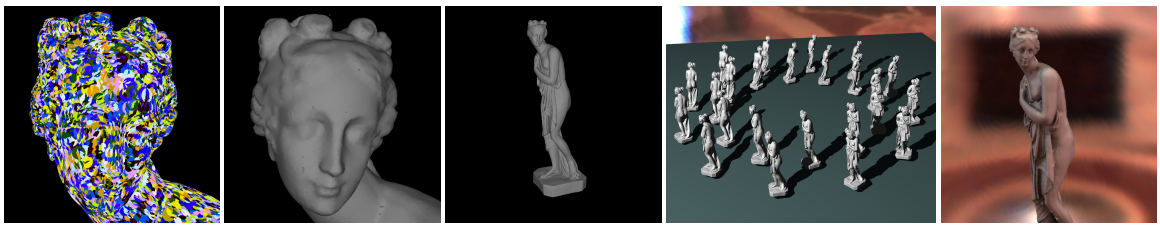

Figure 1: *Several examples of interactively ray tracing point-based scenes: Interactive ray tracing of point-based scenes: a) "Iphigenia" head, each point represented by a disc. b) The splats blended to an implicit function and intersected using an acceleration structure (*$6.8\ fps@512\times512$ *pixels). c) The full model (*$15.9\ fps@512\times512$ *pixels). d) A complex scene of 24 Iphigenias (24 million points) with phong shader and shadows (*$\sim 2\,fps@640\times480$ *pixels). e) Iphigenia, displayed interactively with a (precomputed) global illumination solution (*$\sim 4\,fps@400\times600$ *pixels). All frame rates are measured on a single PC.*

Abstract

Point-based methods have recently gained significant interest, as their simplicity and independence of connectivity make them a simple and powerful tool in both modelling and rendering. Still, their use for high-quality and photorealistic rendering is still in its infancy, in particular for interactive applications. This paper describes a framework for interactively ray tracing point-based models based on a combination of an implicit surface representation, an efficient surface intersection algorithm, and a specially designed acceleration structure. Using this framework allows for interactively ray tracing even highly complex models on a single PC, including global illumination effects and the interactive visualization of a 24-million-point model with ray traced shadows.

1. Introduction

In recent years, point-based methods have gained significant interest. In particular their simplicity and total independence of topology and connectivity make them an immensely powerful *and* easy-to-use tool for both modelling and rendering. For example, points are a natural representation for most data acquired via measuring devices such as range scanners [LPC*00], and directly rendering them without the need for cleanup and tessellation makes for a huge advantage.

Second, the independence of connectivity and topology allow for applying all kinds of operations to the points without having to worry about preserving topology or connectivity [PZvBG00, OBA*03, PKKG03]. In particular, filtering operations are much simpler to apply to point sets than to triangular models. This allows for efficiently reducing aliasing through multi-resolution techniques [PZvBG00, RL00, WS03], which is particularly useful for the currently observable trend towards more and more complex models: As soon as triangles get smaller than individual pixels, the rationale behind using triangles vanishes, and points seem to be the more useful primitives.

1.1. Ray Tracing and Global Illumination on PBMs

Albeit their growing importance, using point-based models (PBMs) for high-quality and photorealistic rendering is still in its infancy, in particular for interactive applications. Nowadays, point-based models are rendered almost exclusively via splatting them into a Z-buffer. This operation corresponds naturally to rasterization for triangles, and shares many of its advantages – namely, simplicity – and deficiencies. In particular, it gets costly for complex models, and high-quality shading effects such as shadows, reflections, or even global illumination are hard to achieve at all.

Computing such global effects also on point based models eventually requires ray tracing. As both rays and points are "singular" primitives, this requires one to either trace "thick" rays (such as cones, beams, etc [Ama84, SJ00, WS03]), or to somehow make the points cover a finite area (using, e.g., disks, ellipses, or spheres [RL00]; or via blending them to an implicit function [AA03a, OBA*03]). Unfortunately, both variants are quite costly. Additionally, most of the recent progress towards fast and realtime ray tracing [Wal04] have bypassed point-based models, as these so far have concentrated exclusively on supporting triangles.

In this paper, we present a framework for interactively

ray tracing even highly complex point-based models using a combination of an implicit surface representation, an efficient surface intersection algorithm, and a specially modified acceleration structure. Using this framework allows for interactively ray tracing even highly complex models of several million points on a single PC, including global effects like shadows and even (precomputed) global illumination.

2. Previous Work

Point-based methods have a long history in computer graphics. In this paper we will concentrate on (interactive) point based rendering and ray tracing. For an overview of using point-based representations in acquisition and modelling, see for example the recent survey by Kobbelt at al. [KB04].

Points as display primitives have already been used in '85 by Levoy and Whitted [LW05]. Since then, most approaches have been targeting the efficient and/or high-quality rendering of point-based models by various forms of splatting [PZvBG00, RL00, ZPvBG01, CH02, WK04]. In contrast to splatting-based approaches, we want to ray trace the point sets. As both points and rays are singular primitives, the intersection of a ray with a point has zero probability, which leaves only two options for ray tracing points: Either to grow the primitives such that they cover a non-differential surface, or, alternatively, to trace "thick" rays in the form of cylinders, cones or beams.

Following the second of these approaches, Schaufler et al. [SJ00] have traced cylinders, and have computed the intersection depending on the local density of the points along the "ray". Unfortunately, this operation is quite expensive, and may lead to inconsistencies as the outcome of the surface intersection depends on the direction of the ray.

Another approach of tracing tracing thick rays has been proposed by Wand et al. [WS03], who trace (anisotropic) ray cones into a multi-resolution hierarchy, thereby directly computing effects like anti-aliasing or blurry reflections without the need for stochastic sampling. However, tracing cones is is usually far from interactivity. Quite generally, "thick ray" approaches suffer from the fact that such extended rays tend to traverse large portions of the acceleration structure, and visit lots of primitives.

The alternative to growing the rays is to grow the primitives such as to cover a non-differential area. In its most simple form, this means replacing each point by a disk or ellipse, which in turn corresponds to splatting with flat shading. Though conceptually quite simple, this approach leads to shading artifacts in particular at the silhouettes, and where different primitives intersect each other (see Figure 2).

These artifacts can only be removed by defining a smooth and continuous surface. To this end, Adamson et al. [AA03b, AA03a] have proposed an implicit surface model that equips points with a spherical support that allows for "blending" overlapping splats together to a smooth surface (see Figure 2). We will eventually use the same surface model as well, and will discuss it in more detail below. Note that a similar approach has also been proposed by Ohtake at al. [OBA*03], in the form of MPU Implicits.

Figure 2: *Using disks as primitives (top row) vs. Adamsons implicit surface model also used in our approach (second row). From left to right: Iphigenia model with 2,000, 32,000 and 1,000,000 splats, respectively. Whereas the disks result in shading artifacts, the implicit surface is smooth and continuous. The effect of a non-continuous surface such as disks is further emphasized when adding highlights or shadows (bottom row). Artifacts persist even in the highest resolution available (see zoom of the nose).*

3. Interactive Ray Tracing of Point Based Models

As discussed above, ray tracing points sets requires one to either trace thick rays, or to extend the points to cover a surface. In our experience, tracing thick rays significantly complicates the traversal, and is too costly for interactivity. For this reason, we have ruled out thick rays, and have opted on the second approach. In particular, tracing "usual" rays allowed for taking benefit from vast experience in fast ray tracing that has been developed for polygonal ray tracing [Wal04].

Having decided on using thin rays, ray tracing a point set can be broken down into the following sub-problems:

1. What surface representation to use,
2. How to efficiently compute the surface intersection,
3. What acceleration structure to use, and how to construct it optimally to achieve minimal cost.

In the following, we will discuss these issues step by step.

3.1. Surface Model

Given today's extremely high ray tracing performance for polygonal models [Wal04], our first attempt towards point based ray tracing was to extend an existing realtime ray tracer, and to extend it to support disk primitives as well. Most of the existing knowledge on fast ray tracing could be directly applied to this new framework as well, including fast traversal [WSBW01], construction of high-quality kd-trees [Wal04, Hav01], parallelization, etc. Though this yielded relatively good performance results, the rendering quality was not acceptable, in particular close to silhouette edges, or in highly curved regions (see Figure 2).

The second approach we tried was based on the observation that the shading artifacts in the disk-approach are mostly due to two issues: Disks "sticking through" other disks they are intersecting, and shading discontinuities due to a discontinuous surface normal. In order to remove those problems, we experimented with cutting off those parts of the disks that stick through others by suitable modifications to the primitive intersection code. If that had been successful, a smooth appearance could have been achieved by smoothly interpolating the normal from neighboring splats. Unfortunately, this approach was largely unsuccessful as well, as the modified intersection code was highly nontrivial, very costly, and numerically unstable. In particularly the latter led to thin hole and crack artifacts that were not tolerable.

3.1.1. Adamson and Alexas Implicit Surface Model

Finally, we experimented with *smoothly* blending the individual splats together, and finally arrived at Adamson et al.'s implicit surface model [AA03b]: Each primitive splat[†] $S_i = (p_i, n_i, r_i)_{i=1}^N$ is defined by its position x_i, its normal n_i, and its radius of influence r_i. Inside this radius of influence, the point is surrounded by a weight function

$$w_i(x) = W(\frac{||x - p_i||}{r_i}),$$

where $W(r)$ is a decreasing weight function, usually a (truncated) Gaussian, a spline, etc. In our experiments, a simple hat function

$$W(r) = \begin{cases} 1-r & ;r < 1 \\ 0 & ;r \geq 1 \end{cases}$$

has shown to yield reasonable results.

Once the p_i, n_i, r_i, and W are defined, for each point $x \in \Re^3$ we can define a weighted average of the position

$$\bar{p}(x) = \frac{\sum w_i(x) p_i}{\sum w_i(x)},$$

and normal

$$\bar{n}(x) = \frac{\sum w_i(x) n_i}{\sum w_i(x)}$$

of the surrounding splats. These define a local plane approximation, which in turn allows for defining an implicit function

$$f(x) = (x - \bar{p}(x))\bar{n}(x),$$

whose root is a smooth, continuous surface.

3.2. Surface Intersection

In this surface model, each splat only has a small, local support. Thus, during each evaluation of $f(x)$, only a small number of splats will actually have a non-zero contribution. In particular, large parts of the ray interval will not overlap any splat at all, and cannot have an intersection. Thus, efficient intersection requires skipping the regions where there is no overlap, and to always only consider those splats that potentially have any influence at all. Therefore, we build a kd-tree over the model, such that each cell of the kd-tree stores a reference to all the splats whose support overlaps it. The exact way that kd-tree is built is very important, and will be discussed in more detail below in Section 3.4.

3.2.1. Fast Ray/Implicit Surface Intersection

Once a kd-cell is encountered while traversing a ray $R(t)$ through the kd-tree, the ray has to be intersected (only) with the splats overlapping that cell, and computing an intersection requires finding the closest root of $f(t) := f(x = R(t))$. The obvious approach of using an iterative procedure (like e.g., Newton-iteration or the method outlined in [KB04]) unfortunately turned out to be numerically problematic, in particular close to silhouettes. Instead, we regularly sample the ray interval: Taking k samples $t_0, t_1, ...$ along the ray, there is a surface intersection if there is an i with $sign(f(t_i)) \neq sign(f(t_{i+1}))$. In that case, we linearly interpolate the hitpoint between t_i and t_{i+1}, depending on $f(t_i)$ and $f(t_{i+1})$, respectively. Obviously, only samples in the interval $[t_{near}, t_{far}]$ in which the ray overlaps the current cell have to be considered. This interval is already known from kd-tree traversal without any further effort [Wal04].

Though the above surface model is quite simple, it also is computationally expensive. For that reason, we have spent significant time in optimizing the intersection code. Apart from low-level optimizations (like storing precomputed divisions etc.), a particularly interesting optimization is to exchange the function f for a simpler one with the same root. To this end, we take $W(x) = \sum w_i(x)$ and define

$$\begin{aligned} F(x) &= W^2(x) f(x) && (1) \\ &= (W(x)x - \sum w_i(x) p_i) \sum w_i(x) n_i. && (2) \end{aligned}$$

Except for those x where $W(x) = 0$ (i.e., where no single splat overlaps x, and where f was undefined, anyway), F

[†] Note that we call these primitives "splats" even though they are are obviously not used as such. This naming convention was chosen to emphasize that a splat is more than only a "point", but – though being defined by position, normal, and radius – is also not a disk.

has the same root and signs as f. This, it defines the same surface), but is much simpler, and has no divisions any more, yielding the simple intersection routine:

```
bool INTERSECT(Ray R, Splats S[])
float oldF = 0; oldT = t_near;
for (i=0..k-1)
   t = Interpolate(i/(k-1),t_near,t_far);
   x = origin + t * direction;
   W = 0; N = 0; P = 0;
   for (each splat S)
      w = S.w(x);
      if (w == 0) continue;
      W += w;   N += w * S.n;   P += w * S.p;
   end
   if (W==0) continue; // not contd in any S
   F = (W*x - P) * N;
   if (F*oldF < 0 /* different signs ! */)
      t_hit = Interpolate(oldF/(oldF-F),
                          oldT,t));
      n_hit = ... /* same loop as above */
      return HIT;
   end
   oldF = F; oldT = t;
end
return NO_HIT;
```

Obviously, the performance of this intersection largely depends on the number of iterations, and on the number of splats in the leaf. Much of that can be influenced by a proper choice of the splat radii (see Section 3.3 below) and by a well-built kd-tree (see Section 3.4 below). The number of iterations required to reach a certain accuracy depends particularly on how well the kd-tree encloses the surface, as the kd-cell width directly affects the sample spacing.

3.2.2. SIMD Acceleration

The intersection can be further accelerated by computing four $f(t_i)$ in parallel using SIMD operations [Int02]. Using a well-build kd-tree, a constant number of $N = 4$ samples are sufficient, allowing for computing all four samples in a single sweep without any iteration at all. Additionally, in SIMD part of the conditionals can be replaced by cheap min/max operations, and the sign operations are trivial as well. Taken together, the computational density of the code is quite high, and SIMD acceleration provides good results.

Note, however, that using only $N = 4$ sample points inside the cell – and thus, the ability to use the SIMD variant – *only* works if the cell tightly encloses the surface, and is prone to sub-sampling artifacts if the cells are too large.

3.3. Choosing optimal Splat Radii

The performance of the surface intersection also depends on the splat radii, as too large splats result in each splat covering a large volume, i.e., in lots of splats per cell on average. On the other hand, too small splat radii will result in holes in the model. Note that the splat radii cannot be changed interactively, as they need to be known during kd-tree construction.

The "optimal" splat radii for our purposes would require

1. that the splats cover the entire surface without holes, and
2. that each splat is as small as possible without violating the previous condition.

A method to generate such a coverage has recently been proposed by Wu et al. [WK04]: In a first step, a splat is grown from each input point, such that a certain error tolerance is met. These splats are then subsampled greedily in a way that guarantees that complete surface coverage is maintained. Finally, a global optimization procedure is applied to further optimize the placement and size of the splats.

Though the purpose of this method originally was to optimally sub-sample a model at any desired resolution, its output perfectly fits our requirements also without using its subsampling capabilities at all. In fact, we did not even have to re-implement that method, as the authors have graciously made their already preprocessed data available to us. Obviously, using the subsampling capabilities directly lend for a level-of-detail approach also for our approach. This is already being investigated, but required building a kd-tree that encodes multiple model resolutions at the same time.

3.4. Building a High-Quality kd-Tree

Though we have already mentioned that we are using a kd-tree, we have not yet discussed its actual construction method. This however is quite important, as it has to fulfill several demands:

1. It should minimize the number of traversal steps, and – in particular - the number of costly surface intersections.
2. It should enclose the surface as tightly as possible, in order to guarantee that our surface intersection works reliably and efficiently
3. It should minimize the number of splats that have to be considered per ray, to minimize intersection cost.

The first item can best be achieved by building the kd-tree using a cost estimation function such as a surface area heuristic (SAH) [Wal04, Hav01]. This heuristic estiates the cost C of splitting cell V into cells V_l and V_r as

$$C(V) = C_{trav} + \frac{SA(V_l)}{SA(V)} C_{est}(n_l) + \frac{SA(V_r)}{SA(V)} C_{est}(n_r), \quad (3)$$

where $SA(V)$ is the surface area of cell V, $C_{est}(n) = C_{isec} \times n$ is the (estimated) cost of traversing a child with n primitives, n_l and n_r are the number of primitives overlapping V_l and V_r, respectively, and C_{isec} and C_{trav} are constants representing the cost of a traversal and primitive intersection, respectively. In order to optimally place the split planes the SAH needs a good estimate on the extent of the primitives it considers. Without an explicit representation of the surface,

this unfortunately is not available. For that reason, we simply enclose each splat with an axis-aligned box that exactly encloses it, and apply the SAH to these kinds of "primitives".

These boxes work quite well on a coarse scale, but have the disadvantage that the extent of the surface is significantly overestimated, in particular for coarse models with large radii. This has two disadvantages: First, split planes are often further apart from the surface than necessary, resulting in unnecessary traversals and intersections. Second, the automatic termination criterion of the SAH – which stops further subdivision if the expected cost for a split is larger than the expected cost for making a cell – often terminates subdivision too early, as the children's cost is overestimated[‡].

Obviously, these deficiencies could best be remedied by a better estimate of the surface's extent. As no such estimate is (yet) available, we use several heuristics to improve on them.

3.4.1. KD-Cell Shrinking and Splat Culling

Once the SAH decides not to subdivide the cell any further, we first try to shrink the cell towards the surface as follows: First, we calculate the average of the normals of all splats in the cell. Orthogonal to the dominant dimension of that normal we then slice the cell into K equidistant slices.

For each of these slices V, we estimate $F_{min} = \min\{F(x \in V)\}$ and $F_{max} = \max\{F(x \in V)\}$ by sampling F (see eq. 1) with random $x \in V$. If those values are beyond certain thresholds (i.e., $F_{min} > \varepsilon$ or $F_{max} < -\varepsilon$), this slice is likely not to contain the surface. Thus, the cell can be shrunk respectively by inserting a new split plane. In particular, if no slice contained the surface at all (which is perfectly possible), the cell is completely marked empty. Using this method, much of the overestimated space can be correctly classified as empty. In practice, we use 7–13 slices and 100-200 samples per slice. Clearly, sampling can also lead to missing a "full" voxel due to undersampling. This happens in particular close to highly curved regions, but is usually only visible when zooming in closely, and is usually quite tolerable. Using interval arithmetic for (conservatively) estimating F_{min} and F_{max} would avoid for both getting rid of these artifacts, as well as for significantly reducing precomputation time. So far however this is not implemented yet.

After the leaf has been shrunk, it is possible that splats from the original cell will no longer overlap the shrunken cell. This will be checked, and those splats get removed.

3.4.2. Encouraging of Splits for Smaller KD-cells

As mentioned above, the overestimation of the surface extent leads in too early termination of the subdivision procedure. This effect unfortunately cannot be helped by the above cell shrinking, as the cell shrinking will only shrink the bounds of a cell to the surface contained within it, but will not further split the surface itself. Introducing new splits into an already-shrunk cell is not trivial, either, as good candidates for split planes are not obvious.

Instead, we artificially encourage splitting by modifying the cost estimation function. Instead of $C_{cell}(n) = C_{isec} \times n$, we introduce an additional factor $E(n)$, that artificially lowers the cost for small cells, yielding

$$C_{cell}(n) = C_{isec} \times n \times E(n),$$

In practice, we use $E(n) = clamp(1, \frac{95+5n}{200})$, but similar parameters are possible as well. Note that this term intentionally affects only already small cells: The SAH already works well on the coarse scale, and modifying the cost estimate on the coarse scale often leads to unexpected results.

3.4.3. KD-Tree Post-Collapse

By design and intention, encouraging splits leads to lots of small cells, most of which are later on significantly shrunk, or culled. We therefore perform an additional cleanup pass, in which splits with two equal siblings – i.e., two empty children – get merged by collapsing their parent split node.

4. Results and Applications

Once all individual parts of our method are now together, we can evaluate its performance. Figure 3 shows two views of the Iphigenia statue, rendered at 512×512 pixels. These will be used in the following experiments. All experiments are performed on a single 2.4GHz dual-Opteron PC.

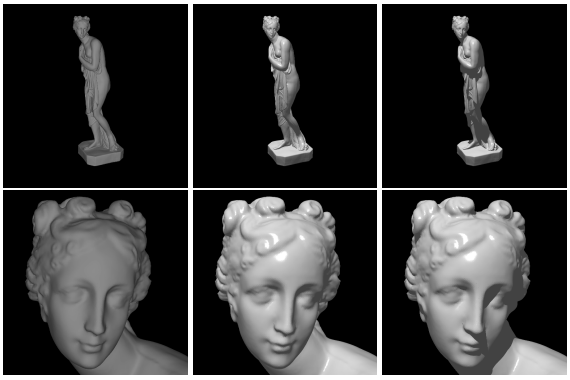

Figure 3: *The Iphigenia model used in our experiments, available in various resolutions (shown: 125k points). Top: Full model. Bottom: Zoom towards the head. From left to right: Diffuse, Phong with highlights, and shadows.*

4.1. Overall Performance

Our test model is available in several resolutions: 2k, 32k, 125k, and 1M points. To quantify overall performance, have measured the performance for each of them with all optimizations turned on:

[‡] $E(n) = C_{isec} \times n$ assumes the cell to be a leaf. Since the cell might be subdivided later on, its actual cost can be significantly lower.

(in frames per sec.)	2k	32k	125k	1M
head	10.3	8.3	7.8	6.8
full	30	26.1	22	15.9

With simple GL-like shading per ray, we achieve 6.8–10.3 and 15.9–30 frames per second for the head and the full statue, respectively. Note that even though it shows much more points, the entire statue is much faster than the head only, as significantly fewer pixels are covered.

4.2. Scalability in Model Resolution

As can also be seen from these measurements, model resolution has only a relatively small impact: While the model size increases by almost three orders of magnitude, performance drops only by about 30% for the head view, respectively 50% for the full model. This is a particularly interesting feature since model size is still one of the most limiting factors in splatting based approaches.

As a stress test, Figure 4 shows a scene with 24 Iphigenias, totalling 24 million points. Even with additional shadows, we achieve interactive performance of $\sim$2 frames per second at 640×480 pixels. Note that – though this is trivially possible – this scene does *not* use multiple instantiation, but really consist of 24 million individual points.

4.3. Traversal Statistics

To best understand the good scalability in model size, it is helpful to have a closer look at traversal statistics. Thus, for each of the experiments above we have measured the (average) number of kd-tree node traversal steps, leaf cells encountered (including empty ones), computed surface intersections, and number of splats encountered per ray:

view	resolution	2k	32k	125k	1M
head	traversals	32.0	47.1	50.6	55.2
	cells	5.99	9.02	9.47	9.98
	intersections	1.07	1.06	0.99	0.97
	splats	10.8	7.87	8.03	6.90
full	traversals	23.4	30.2	35.2	40.6
	cells	4.81	5.77	6.50	7.34
	intersections	0.64	0.61	0.62	0.62
	splats	6.02	4.76	5.06	4.41

As can be seen, the number of cell intersections – one of the main cost factors – stays almost constant, and only the number of traversal steps increases. With increasing resolution, the lessening effect of surface overestimation even *reduces* the number of intersections.

Effect of kd-tree Optimizations: To roughly quantify the impact of the kd-tree optimizations outlined in Section 3.4, we have performed the same experiments also with all kd-tree statistics turned off, albeit only for one model resolution (125k):

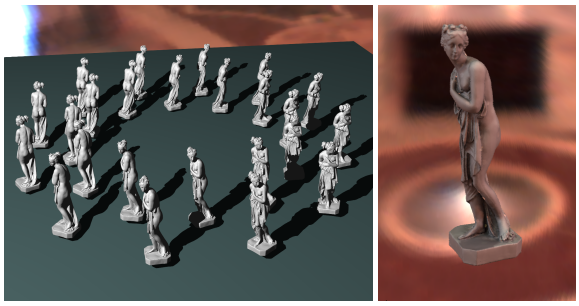

Figure 4: *Two examples from the final system: a) 24 Iphigenias (24 million points total), with Phong shading and shadows. b) Iphigenia with precomputed global illumination, showing the Iphigenia illuminated from an HDR environment map of St.Peters. At* 640×480 *and* 400×600 *pixels, these examples render at* $\sim$*2 and* $\sim$*4 frames per second, respectively, on a single 2.4GHz dual-Opteron PC.*

Iphigenia, 125k		kd-node trvsals	kd-cell trvsals	surf isecs	splats visited
head	no opt	46.3	8.7	1.6	15.8
	opt	50.6	9.5	0.99	8.0
full	no opt	32.0	5.9	1.1	10.8
	opt	35.2	6.5	0.6	5.06

As expected, by slighly increasing the traversals the kd-tree optimizations significantly reduce the number of surface intersections and splats considered per ray. Note that "no opt" already refers to a highly tuned SAH implementation.

Hot-spot Analysis Given the previous data, it is interesting to see where the individual time is spent. We have therefore measured how the total rendering time splits up into surface intersection, traversal, and overhead (ray generation, shading, etc), respectively (relative numbers, in percent):

Trv:Isec:Other	32k	125k	1M
head	28:48:24	27:49:24	27:53:20
full	18:32:50	18:31:51	18:32:50

Note that these number are *very* coarse, due to the simplistic measurement procedure involved. As can be seen, the overall traversal and intersection performance is already high enough to make ray generation and shading consume significant portions of time (up to 50% of total time!).

4.4. Rendering Quality

Apart from performance data, it is also interesting to investigate rendering quality. As the surface is smooth and continuous, rendering artifacts are quite low. For really coarse models (e.g., 2k resolution), the blending does not work well, in particular at silhouettes. Also, zooming closely onto the surface shows some high-frequency noise in the normal, probably a consequence of the simple hat filter used for blending. As it is only visible from a close distance, it is quite tolerable. For typical model resolutions (125k or more), the overall quality is quite high, and few artifacts appear, if any.

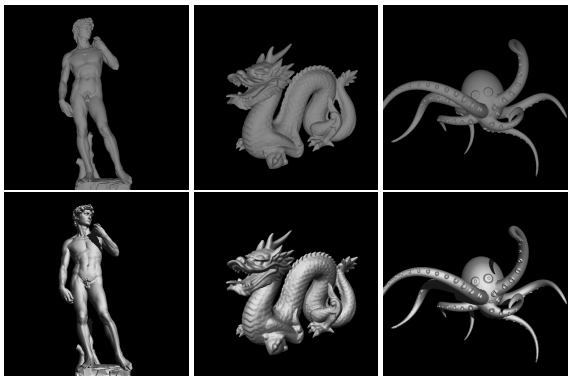

Figure 5: *Our method applied to the David (1,501,177 points), Dragon (1,309,059 points), and Octopus (465,878 points) models. Note that for the Octopus the radii have* not *been optimized using Wu's technique, and contains holes. At* 512×512 *pixels on a single dual-Opteron PC, we can render these models at 10.6, 7.5, and 8.8 frames per second using simple shading, and 4.1, 5.7, and 4.1 frames per second using a Phong shader with shadows, respectively.*

4.5. Global Effects

Of course, shading is always performed "per pixel", and not interpolated from the nearest splats. Arbitrary secondary rays can be shot, e.g., for computing accurate shadows or reflections (see Figures 3 and 4).

Apart from "typical" ray traced effects like hard shadows and reflections, it is interesting to also apply global illumination effects to point based model. For this purpose, we have designed a new precomputation-based global illumination method [WS05] that – similar to Photon Mapping – has been explicitly designed to be independent of geometry, and is thus applicable also to point based models.

Figure 4 shows the full-resolution Iphigenia lighted from an HDR environment map of St.Peters. Using the afore mentioned precomputed global illumination method, all kinds of global illumination effects are present: Direct as well as indirect illumination, self-shadowing and self-illumination, color bleeding, highlights, arbitrary BRDFs, etc. As all illumination is precomputed, only a single ray has to be shot per pixel, and the fully illuminated model can be viewed interactively at $\sim$4 frames per second at 400×600 pixels.

5. Summary and Conclusion

In this paper, we have sketched a complete framework for interactively ray tracing point based models. This framework consists of a whole suite of different techniques. In particular, we have decided to trace "thin" rays, which are intersected with a smooth surface that is defined by a combination of Adamson et al.'s implicit surface model [AA03b, AA03a] Wu et al.'s near-optimal coverage technique [WK04]. This is combined with a highly optimized and SIMD-accelerated intersection code, together with a highly optimized kd-tree that is particularly built to suit the demands of the chosen surface representation and intersection computation. Thus, the power of the approach does not lie in the individual techniques, but in the way that these optimally play together, and emphasize their respective strengths.

Taken together, these methods allow for interactive ray tracing performance of 7 to 30 frames per second at 512×512 pixels, for non-trivial models, on a single dual-2.4GHz Opteron PC. Additionally, the framework allows for tracing arbitrary rays, thus allowing for high-quality and global shading effects like shadows, reflections, and even (interactive) global illumination. The framework is fully integrated into the OpenRT realtime ray tracing system [Wal04], and can be used with all existing shaders, surface types, parallelization features, etc.

In the near future, we plan to extend our system with a multiresolution approach, in particular for visualizing much more complex models. Also, the investigation of dynamic data structures in the spirit of [AKP*05] appears interesting.

Acknowledgements

This paper would not have been possible without the support by Leif Kobbelt, who has graciously made the readily preprocessed Iphigenia, David, and Dragon models available. Thanks also to Mark Pauly for the Octoput model and to Anders Adamson, Marc Alexa, and Johannes Günther for the helpful discussions. Finally, many thanks also to the reviewers for the very detailed and helpful comments.

References

[AA03a] ADAMSON A., ALEXA M.: Approximating and intersecting surfaces from points. In *SGP '03: Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing* (2003), pp. 230–239.

[AA03b] ADAMSON A., ALEXA M.: Ray Tracing Point Set Surfaces. In *SMI '03: Proceedings of the Shape Modeling International 2003* (2003), p. 272.

[AKP*05] ADAMS B., KEISER R., PAULY M., GUIBAS L. J., GROSS M., DUTRÉ P.: Efficient Ray Tracing of Deforming Point-Sampled Surfaces. In *Proceedings of Eurographics 2005* (2005). to appear.

[Ama84] AMANATIDES J.: Ray tracing with cones. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques* (1984), pp. 129–135.

[CH02] COCONU L., HEGE H.-C.: Hardware-Accelerated Point-Based Rendering of Complex Scenes. In *Proceedings of the 13th Eurographics Workshop on Rendering* (2002), pp. 43–52.

[Hav01] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001.

[Int02] INTEL CORP.: Intel Pentium III Streaming SIMD Extensions. http://developer.intel.com, 2002.

[KB04] KOBBELT L., BOTSCH M.: A survey of point-based techniques in computer graphics. *Computers & Graphics 28*, 6 (Dec. 2004), 801–814.

[LPC*00] LEVOY M., PULLI K., CURLESS B., RUSINKIEWICZ S., KOLLER D., PEREIRA L., GINZTON M., ANDERSON S., DAVIS J., GINSBERG J., SHADE J., FULK D.: The digital Michelangelo project: 3D scanning of large statues. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (2000), pp. 131–144.

[LW05] LEVOY M., WHITTED T.: *The use of points as display primitives*. Tech. rep., CS Department, University of North Carolina at Chapel Hill, 2005.

[OBA*03] OHTAKE Y., BELYAEV A., ALEXA M., TURK G., SEIDEL H.-P.: Multi-level partition of unity implicits. *ACM Trans. Graph. 22*, 3 (2003), 463–470.

[PKKG03] PAULY M., KEISER R., KOBBELT L. P., GROSS M.: Shape modeling with point-sampled geometry. *ACM Trans. Graph. 22*, 3 (2003), 641–650.

[PZvBG00] PFISTER H., ZWICKER M., VAN BAAR J., GROSS M.: Surfels: Surface Elements as Rendering Primitives. In *Proc. of ACM SIGGRAPH* (2000), pp. 335–342.

[RL00] RUSINKIEWICZ S., LEVOY M.: QSplat: A Multiresolution Point Rendering System for Large Meshes. In *Proc. of ACM SIGGRAPH* (2000), pp. 343–352.

[SJ00] SCHAUFLER G., JENSEN H. W.: Ray Tracing Point Sampled Geometry. In *Proceedings of the Eurographics Workshop on Rendering Techniques* (2000), pp. 319–328.

[Wal04] WALD I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. Available at http://www.mpi-sb.mpg.de/~wald/PhD/.

[WK04] WU J., KOBBELT L.: Optimized Sub-Sampling of Point Sets for Surface Splatting. In *Proceedings of Eurographics 2004* (2004), vol. 23 of *Computer Graphics Forum*, pp. 643–652.

[WS03] WAND M., STRASSER W.: Multi-Resolution Point-Sample Raytracing. In *Graphics Interface 2003 Conference Proceedings* (2003).

[WS05] WALD I., SEIDEL H.-P.: High-Quality Global Illumination Walkthroughs using Discretized Incident Radiance Maps. (submitted for publication), 2005.

[WSBW01] WALD I., SLUSALLEK P., BENTHIN C., WAGNER M.: Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum 20*, 3 (2001), 153–164. (Proceedings of Eurographics).

[ZPvBG01] ZWICKER M., PFISTER H., VAN BAAR J., GROSS M.: Surface splatting. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), pp. 371–378.