EUROGRAPHICS 2004 / M.-P. Cani and M. Slater
(Guest Editors)

Volume 23 (2004), Number 3

Deferred Splatting

G. Guennebaud and L. Barthe and M. Paulin†

IRIT - CNRS - Université Paul Sabatier - Toulouse - France

Abstract

In recent years it has been shown that, above a certain complexity, points become the most efficient rendering primitives. Although the programmability of the lastest graphics hardware allows efficient implementation of high quality surface splatting algorithms, their performance remains below those obtained with simpler point based rendering algorithms when they are used for scenes of high complexity. In this paper, our goal is to apply high quality point based rendering algorithms on complex scenes. For this purpose, we show how to take advantage of temporal coherency in a very accurate hardware accelerated point selection algorithm allowing the expensive computations to be peformed only on visible points. Our algorithm is based on a multi-pass hardware accelerated EWA splatting. It is also suitable for any rendering application since no pre-process is needed and no assumption is made on the data structure. In addition, we briefly discuss the association of our method with other existing culling techniques and optimization for particular applications.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Viewing algorithms

1. Introduction

Recently, it has been shown that points are conceptually the most efficient primitives for the rendering of complex geometry. This is mainly due to the size of triangles becoming smaller than a single pixel when the scene complexity increases. The rendering of these tiny triangles is then inefficient because of the necessary overhead for triangle setup.

A fundamental issue for point based rendering is hole filling. The challenge is the reconstruction of a continuous image of the point cloud when the space between two neighbour points reaches or exceeds the size of a pixel. This task can be done using an image-based filtering technique, by adjusting on the fly the sampling density or using the so-called surface splatting technique. In this last case, each point, also called *surfel*, is associated with a 2D reconstruction kernel (defined in the surfel tangent plane) which is projected onto the image space. Two recent papers [BK03, GP03] show that modern GPUs now support efficient point rendering with high quality reconstruction. However, these techniques only achieve a point rate from 7M to 10M high quality filtered

Figure 1: *Our point based rendering algorithm applied on a landscape. This frame contains 1500 visible trees (750k points per tree). 95% of points declared visible by the high level culling and LOD are culled by our accurate point selection, increasing performance from 1.1 fps to 12 fps.*

† e-mail: {guenneba | lbarthe | paulin}@irit.fr

© The Eurographics Association and Blackwell Publishing 2004. Published by Blackwell Publishing, 9600 Garsington Road, Oxford OX4 2DQ, UK and 350 Main Street, Malden, MA 02148, USA.

splats per second because it is necessary to use both a multi-pass approach and a slow fragment programmability. The same hardware attains 70M-80M small unfiltered squares per second[DVS03]. Hence, in order to efficiently use one of these high quality point rendering techniques on complex scenes (figure 1), we should limit the expensive filtering computations to visible points only. We do this by adding *selection algorithms* on top of the rendering step. These include view frustum, back-face and occlusion culling techniques. We also include level-of-detail (LOD) methods that remove superfluous primitives in case of minification. These selections can be performed at different levels:

High level selection algorithms work on bounding volumes of objects or primitives, commonly unified within a hierarchy.

Low level selection algorithms work at primitive level and, in the final scan convertion process at pixel level.

Previous research has only focused on high level methods that perform fast and coarse selection on top of the accurate low level selection. Commonly, this last visibility computation is entirely performed by the GPU itself, which processes only per primitive view frustum and back-face culling and per pixel occlusion culling. However, as we show in this paper, in the case of complex shaders it is very useful to perform low level selection ourselves before the rendering step.

Since the rendering of simple points is significantly faster than the rendering of high quality filtered splats, the main idea of this paper is to perform the expensive rendering process only on visible splats (similar to deferred shading techniques). To do so, we present an efficient and accurate point selection algorithm and show how to take advantage of temporal coherency. The main features of our algorithm are:

- **no preprocess**: suitable for dynamic point clouds
- **accuracy**: per primitive view frustum and occlusion culling and LOD
- **low level**: takes as input the result of any high level culling
- **temporal coherency**
- **hardware acceleration**

Before describing our algorithm in section 4, we briefly present the previous work (section 2) and in particular the GPU-based EWA splatting algorithm of Guennebaud et al. [GP03] which is the base of our work (section 3). In addition to our generic algorithm, we also present a full hardware accelerated implementation suitable for the specific case where our accurate point selection is enabled on a unique object (section 6). Finally, we discuss about the association of our method with other existing high level techniques such as occlusion culling and LOD selection (section 7).

2. Related Work

In this section we focus on methods which take points as a display primitive for high quality rendering. A recent and more general survey on point based representation can be found in [Kri03].

2.1. High Quality Point Based Rendering

Point based rendering systems have been the product of recent research, especially for methods that provide high quality. In 2001, Zwicker et al. [ZPvBG01] introduced elliptical weighted average (EWA) surface splatting that provides high quality anisotropic texture filtering [Hec89]. A reconstruction kernel (a 2D Gaussian) is associated to each point and warped to elliptical splats in image space. Before the rasterization, this warped reconstruction kernel is band limited by an image space low pass filter that removes high frequencies. Although their software based approach provides the highest visual quality, it only handles 250k splats per second.

Ren et al. [RPZ02] reformulate the image based EWA filtering in the object space in order to allow a hardware implementation. They use a multi-pass approach for hidden surface removal and render each splat as a textured rectangle in the object space. This concept increases by a factor of four the number of processed points, slowing down the rendering to about 2M-3M splats per second.

In a hybrid point-polygon rendering system, Coconu and Hege [CH02] render surfels back to front using elliptical Gaussian and alpha blending (fuzzy splats) without depth test. Although it avoids an expensive additional pass for precomputing the depth buffer, this concept leads to a coarse texture filtering that prohibits large magnification.

More recently, Botsch et al. [BK03] and Guennebaud et al. [GP03] present two high quality hardware-accelerated point rendering algorithms. These two methods are very similar, both using the hardware point primitive with per fragment depth correction, elliptical gaussian with additive blending mode and multi-pass rendering (visibility splatting, reconstruction, normalization). The main difference is that the second method is based on the EWA splatting of Zwicker et al. [ZPvBG01] and hence provides better anti-aliasing for minification. Another difference is that Botsch et al. perform approximation in the perspective computation and this can lead to artefacts in the case of large magnification.

2.2. Data Structure For High Level Point Selection

In order to accelerate rendering, most point based rendering systems use hierarchical data structures to store the set of points [PZvG00, RL00, BWK02]. These data structures allow hierarchical culling and local LOD point selection but they are not designed to be used with the GPU which, for efficiency, should take large data chunks as input. In [SD01, DCSD01] the point sampling rate is dynamically adjusted by rendering list prefix of pre-computed random point sets. Due to the use of random sampling, the two previous approaches are only suitable for highly irregular objets (such as plants). More recently, Dachsbacher et al. [DVS03]

© The Eurographics Association and Blackwell Publishing 2004.

present sequential point trees: an efficient data structure for points adaptively rendered by sequential processing on the GPU.

Although most of these data structures allow frustum culling and back-face culling, none present techniques for occlusion culling. However, many occlusion culling algorithms work on the bounding box, and so, could be easily adapted for point based geometry. A recent survey of visibility algorithms for interactive application can be found in [COCSD03].

3. GPU-Based EWA Splatting

We have chosen to base our work on our point based rendering algorithm [GP03] because it offers high reconstruction quality and takes advantages of recent GPU capabilities. We point out that due to their similarities, the methods of Botsch et al. [BK03] and Ren et al. [RPZ02] can also be used. In this section we present an overview of the our previous multi-pass algorithm (figure 2):

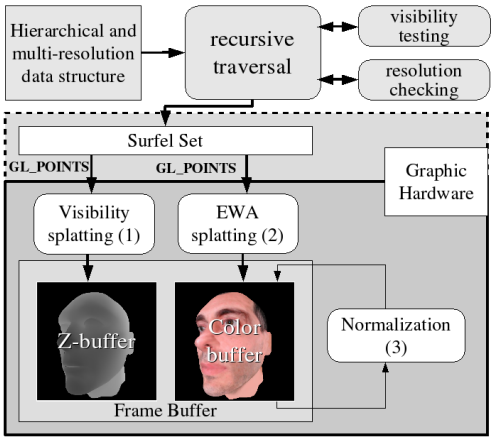


Figure 2: *Overview of the previous multi-pass rendering algorithm.*

1. **Visibility Splatting**
   The goal of this pass is to pre-compute a correct depth buffer without any hole. To do so, all surfels are rendered into the depth buffer via the hardware point primitive with per fragment depth and shape correction.
2. **EWA Splatting**
   During this second pass, all surfels are rendered a second time onto the color buffer via the hardware point primitive with additive blending mode. Due to the previous pass, only visible splats are blended. The weight of each fragment is computed from a Gaussian centered at the current surfel position in the screen space, using a fragment program. This Gaussian is the result of the convolution between the surfel reconstruction filter and the low pass filter, which is computed on the fly by a GPU vertex program.
3. **Normalization**
   In this last pass, the resulting color buffer is rendered as a full screen textured rectangle with a simple fragment program that divide RGB components by the alpha component which contains the sum of weights. This pass is independent of the scene complexity and is negligible compared with the rendering time of the two previous passes.

Note that the two splatting passes are relatively expensive because they need the use of both complex vertex programs and simple but slow fragment programs. Compared with the rendering of simple point primitives, they show a slowdown by a factor of 3 to 4.

4. Our Rendering Algorithm with Accurate Point Selection

The motivation of our work is the reduction of the slowdown produced by the expensive rendering computations of the previous algorithm. To do so, our approach is to perform these expensive computations only on visible surfels. The algorithm presented here is based on accurate point selection and temporal coherency, and similar to deferred shading technique, the framebuffer is used to store information about visible primitives.

4.1. Overview

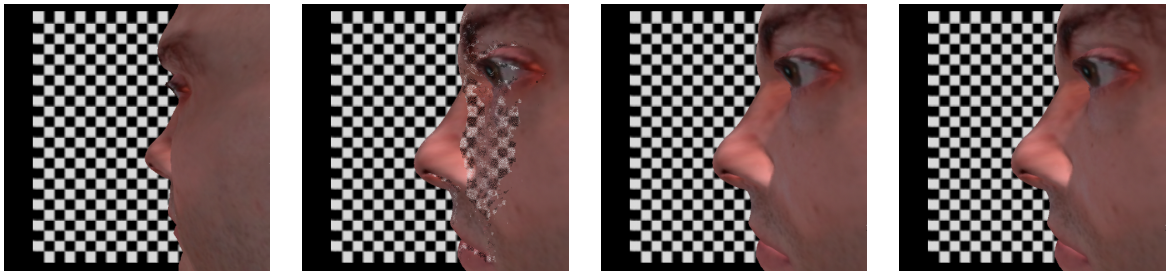
The main idea of our algorithm is the following: between the visibility splatting and EWA splatting of the previous algorithm we first add a pass (number 2) where all input surfels are rendered as a single pixel of the color buffer (section 4.2). But, instead of the surfel color we use the surfel index. Then, after this fast and simple pass, the color buffer contains the list of visible surfels; all hidden points have been discarded by the depth test. We call this sub-set of surfels B_i, where i is the number of the current frame. Now, we are able to perform the EWA splatting (pass 4) only on visible surfels, considerably accelerating this pass.

In order to also accelerate the first pass, i.e. perform the visibility splatting only on visible surfels, we take advantage of temporal coherency (section 4.3) and perform the visibility splatting only on the surfels which are visible in the previous frame (pass 1). Of course, for a correct reconstruction with EWA splatting we must also perform visibility splatting on surfels visible in the current frame and hidden in the previous one (pass 3). To summarize, the multi-pass rendering algorithm of the frame i becomes:

1. Visibility Splatting **with** B_{i-1}
2. **Render surfel indices into the color buffer**
   (we obtain B_i)
3. **Visibility Splatting with** $B_i - B_{i-1}$
4. EWA Splatting **with** B_i
5. Normalization of the color buffer

In the next sections (4.2 and 4.3) we explain in detail our

© The Eurographics Association and Blackwell Publishing 2004.



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

Figure 3: *Two point based models: a checkerboard and a head. (a) Rendered without the second visibility splatting pass. (b) The same scene after a large rotation of the head between 2 consecutive frames. This illustrates the artefacts due to temporal coherency approximations. (c) The next frame. Previous holes are filled and since no object have moved, no new hole appears. (d) As in (b) but with the second visibility splatting pass enabled.*

additional passes 2 and 3. We also discuss the side effects of our algorithm in section 4.4.

4.2. Low Level Point Selection

This section explains in detail the second pass of our rendering algorithm. The challenge is to render as quickly as possible all surfels declared visible by the high level point selection with a unique identifier instead of its color. We use the hardware point primitive with a constant screen space size of one pixel. Since the identifier is stored in the RGBA color buffer, it is limited to 32 bits. Moreover, we should be able to efficiently sort each surfel by object because each object has different transformation matrices, material properties, etc. Thus, we use a very simple combination for the identifier where n bits are reserved for the surfel index (i_s) into the object and the remaining $(32-n)$ bits are used for the object index (i_o). The value of n is a compromise between the maximum number of surfels per object and the maximum number of visible objects. Typically, we take $n = 20$ to allow 1M surfels by object and a maximum of 4096 visible objects. This combination is computed on the fly by the GPU via a very simple vertex program (1 instruction) for which i_o is a uniform parameter. For efficiency, the surfel indices are stored in a single video memory buffer used for all objects. This buffer stores 2^n consecutive integers from 0 to $2^n - 1$. Notice that the identifier corresponding to the clear color value before this pass should never represent a real surfel index. Hence, the clear value must be set to the most improbable identicator value, i.e. `0xFFFFFFFF` which corresponds to the last allowed surfel of the last allowed visible object.

However, if 1M points per object seems to be a good maximum, the number of visible objects can quickly exceed the limit of 4096, e.g. in an aerial view of a forest. More sophisticated combinations can be used to optimize the use of bits, but the extraction of the object index and the surfel index from the identifier must remain as fast as possible. It is also possible to increase the number of bits, by using the 8 bits of the stencil buffer in order to store a part of the object index. However, it is not recommended because it increases the mass of data that must be read from the video memory. In fact, the number of surfels per object is often reduced by LOD. Indeed, very few objects are close to the viewpoint and hence most of the others require less points to be accuratly rendered. Thus, our solution for very large scenes is to use different repartitions of bits in function of the object's size. So, we should reserve a few bits in order to represent the repartition scheme. For instance, in large landscapes, as our forest test scene, we use two schemes (11/20 and 14/17) that allow 2048 objects with more than 131k points and 16384 small objects. In this case, a single bit is sufficient to store the repartition scheme.

Finally, the color buffer is read from the video memory and a vector of indices is built for each object. These index buffers are used both for the EWA splatting pass of the current frame and for the first visibility pass of the next frame as explained in the next section.

4.3. Temporal Coherency

After the accurate point selection of the previous section we are able to perform the expensive EWA splatting only on visible surfels. Thus the cost of this last pass becomes independent of the scene complexity. However, the visibility splatting is also an expensive pass since we must perform a per-fragment depth correction. In this section we show how the visibility splatting pass can be also accelerated by taking advantage of temporal coherency between successive frames.

Indeed, a major part of the visible surfels of the frame number i are still visible in the next frame. Hence, we use only the visible surfels of the frame i for the visibility splatting pass of the frame $i+1$ (instead of all visible surfels resulting of the high level selection). However, all hidden points that become visible are missing, introducing small holes in the depth buffer. Although this depth buffer is accurate enough for our point selection, these holes generate artefacts in the final image because some hidden splats are accumulated into the color buffer (figure 3-b).

These artefacts may be acceptable in some applications

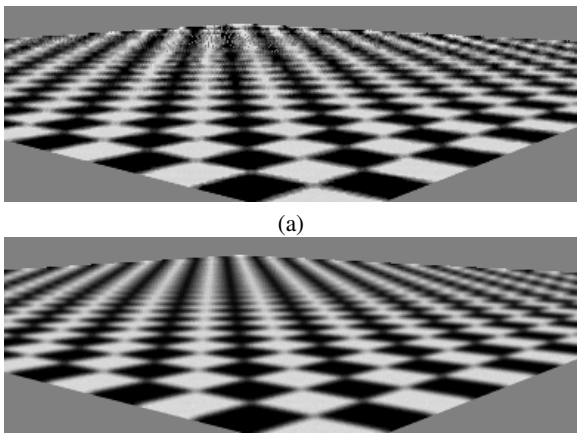
© The Eurographics Association and Blackwell Publishing 2004.

but can be removed without expensive computation by adding a second visibility splatting pass before the EWA splatting. This new pass is performed only with surfels that are potentially visible (holes in the depth buffer make B_i imperfect) in the current frame but hidden in the previous frame, i.e. we perform a visibility splatting pass on $B_i - B_{i-1}$. Since the exact computation of $B_i - B_{i-1}$ is expensive, we only compute a conservative approximation. For that, we use an array of boolean flags where each flag is associated with a group of m consecutive surfel indices and it indicates if one of the surfels in the group belongs to the set B_{i-1}. Thus, we can know immediately if a point of B_i is in B_{i-1}. This difference is computed while sorting the result of the previous pass (section 4.2). Although this approach implies to render more surfels in the first visibility pass, $B_i - B_{i-1}$ becomes smaller and less scattered between objects. Hence, the cost of this second visiblity splatting pass is negligible. The choice of the groups size is a compromize between the accuracy and the memory cost. In our system we use groups of 64 surfels that need 8Mb for stroring flags in the worst case.

Note that during the visibility splatting of pass 1, we use each object's transformation matrix at the current frame i and not the one at the previous frame. This allows our algorithm to also work on dynamic scenes. Since B_i stores only the surfels indices, our algorithm also deals with dynamic point clouds. The robustness is essentially guaranteed by the visibility splatting pass with $B_i - B_{i-1}$.

4.4. Filtering

Since our algorithm allows the projection of only one surfel per pixel, aliasing and flickering artefacts should appear. Indeed, in the case of oversampling, many visible but superfluous points are discarded by our point selection. This means, we lose texture information and introduce aliasing artefacts for high frequency textured models (figure 4-a). Like Pfister et al. in [PZvG00], our solution is to use several prefiltered texture samples per surfel instead of only one texture color by surfel. Such a surfel is also called *surfel mipmap*. In this case, the vertex program of the EWA splatting also performs a linear interpolation between the two closest mipmap levels (figure 4-b). The interpolation coefficient can be computed from the Jacobien matrix which has been already computed during the resampling filter determination (see [PZvG00, ZPvBG01, GP03] for all details).



(a) (b)

Figure 4: *Illustration of the lost of texture information. (a) Aliasing artefacts introduced by high frequency texture. (b) Smoothing using on the fly mipmap levels interpolation.*

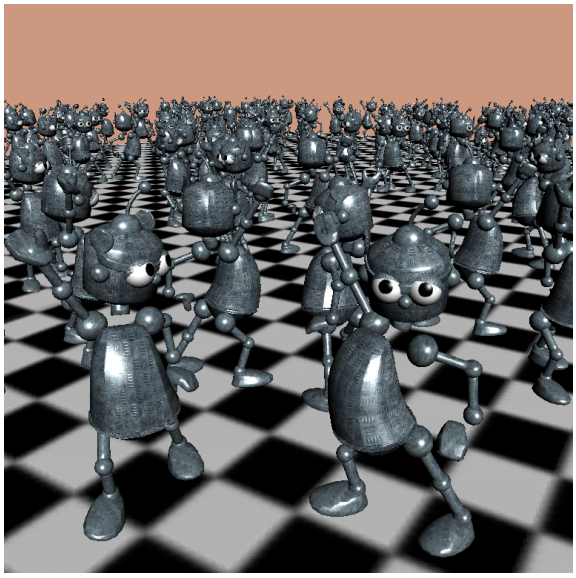


Figure 5: *A dynamic scene (200 obj.) of the Hugo model (450k points) rendered at 33 fps.Laurence Boissieux @ INRIA 2003*

5. Implementation and Results

We have implemented our point selection algorithm with OpenGL under Linux. Performances have been measured on a 2GHz AMD Athlon system with a NVidia GeforceFX 5900 graphic card. All vertex and fragment programs are implemented using `ARB_vertex_program` and `ARB_fragment_program` vendor independent extensions. In order to get the best performance, all the geometries are stored in the video memory via the `ARB_vertex_buffer_object` extension. We also build one vertex buffer object that contains a list of successive indices for pass 2. This buffer is shared by all point based geometries.

In the high level point selection, we have implemented per-object view frustum culling and LOD selection. We have also implemented simple per-object occlusion culling as described in section 7.1. All measures are done with a screen resolution of 512x512 pixels. Our algorithm has been tested

© The Eurographics Association and Blackwell Publishing 2004.

on different types of scenes, from the simpler (a head, figure 3-d) to the more complex (a forest of 6800 trees, figure 7) and on a dynamic scene (figure 5). The rendering times, with and without our point selection, are given in table 1. This table shows that we obtain a maximum speedup by a factor of 10 on very complex scenes and even for a relatively simple model as the head, the overhead due to our additional passes is fully compensated by the gain given by our point selection procedure.

Scene	FPS without	Percentage of culled points	FPS with
Head (285k pts)	34	70%	39
Tree (750k pts)	8.6	88%	32
Dynamic scene	11.5	90%	33.5
Forest (6800 trees)	0.8-1.5	90-98%	10-16

Table 1: *Rendering performance with (column 4) and without (column 2) our point selection algorithm. The column 3 shows also the percentage of points culled by our algorithm.*

Average raw performances for each pass are summed up in table 2, but, for pass 2 we cannot give any average rendering time because it depends highly on the scene complexity and the high level culling, shown by the graph 6. It is also this pass that becomes the most expensive for large scenes. In order to accelerate this pass, we could add more efficient/accurate high level culling algorithms. This observation means that our algorithm and other high level culling methods are not in competition but are complementary and must work together. Since high level culling is not the focus of this paper, performance of this step does not appear in our tables. However, we point out that the view frustum culling and LOD selection are performed in parallel with the first visibility splatting pass, and we have also measured that the occlusion culling step takes from 3 to 6ms for the forest.

Pass	primitves/second	time (ms)
1 - Visibility Splatting	11.5-13.5M/s	7-11
2 - Render indices	90-110M/s	-
Reading the color buffer	-	6.6
Sort indices	-	1.7-2.2
3 - Visibility Splatting	11.5-13.5M/s	0.8-1.2
4 - EWA Splatting	11-13M/s	9-11.5
5 - Normalization	-	0.7

Table 2: *Performances of each pass for a screen resolution of 512x512. The last column indicates the average rendering time of several scenes (The second value is the upper bound).*

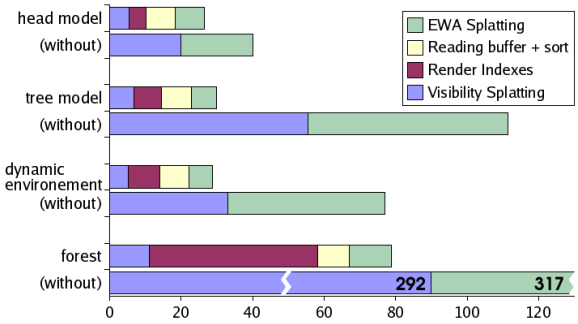


Figure 6: *Rendering time of each passes for different scenes. The seconds rows are obtained without our point selection. Negligible passes (second visibility splatting and normalization) does not appears.*

6. Full Hardware Accelerated Version

In the specific case where only one object is rendered with our accurate point selection, we can optimize our algorithm since we do not need to re-sort surfels by object after pass 2. Indeed, all indices in the color buffer now correspond to the same object. Hence, it is possible to directly copy this buffer into a GPU index buffer. In fact, this will soon be possible with the OpenGL extensions called *SuperBuffers* or *PixelBufferObject*. At the moment data must transfers through the CPU memory, making this process very slow and not CPU free. Also, all pixels with invalid indices have to be removed. This is done using the NVidia `GL_NV_primitive_restart` which allows the user to define an index value which is interpreted as a pair of *glEnd();glBegin(<current primitive>);*. With the GL_POINTS primitive the only effect of this extension is the skipping of this index. The index value used for the `GL_NV_primitive_restart` is the clear color value of the color buffer before pass 2 (see section 4.2 for its choice). Since the `GL_ARB_super_buffers` extension is not available yet, we cannot give accurate results, and today, we obtain very similar performances to the generic version.

Typical applications where this full hardware version will be useful are applications which manipulate dynamic point clouds for editing [ZPKG02], modelling [PKKG03], implicit surface visualisation (via particles system), etc. Indeed, such applications need a lot of CPU resources, and it becomes very important to free the CPU of all the rendering tasks. Moreover, the user interacts often with only a single object at once. Through modifications, it becomes very tedious to maintain a high level data structure while our low level structure remains well suited.

7. Discussion

During the first visibility splatting pass we precompute a depth buffer with points that are visible in the previous

© The Eurographics Association and Blackwell Publishing 2004.

Figure 7: *A landscape with 6800 trees. At the higher resolution, the tree model contains 750k points. This frame contains approximately 2300 visible trees and is rendered at the rate of 11fps. The ground is a polygonal mesh and it illustrates the use of our algorithm mixed with traditional triangle based rendering.*

frame. In a way, the same result can be obtained using image warping. However, a first drawback of an image warping approach is that it is impossible to take into account a dynamic scene: only the camera can move. Moreover, although a forward warping could be done by the GPU, the hole filling process becomes very difficult since we do not know the orientation of each pixel. An inverse warping approach like in [MMB97] solves this last problem but it remains slow.

Since we know the exact visibility percentage of each object (after reading the color buffer in pass 2), a first and simple optimisation of the high level culling step would be to perform visibility compution only on objects that have, in the previous frame, a visibility percentage higher than a given threshold. With our efficient multi-pass approach, it is also possible to add high level occlusion culling (section 7.1). Moreover, we show in section 7.2 that the sequential point tree data structure also takes advantage of our algorithm, and vice versa.

7.1. High Level Occlusion Culling

Today, graphics hardware support features for occlusion culling. The standard OpenGL `ARB_occlusion_query` extension determines, by scan conversion, whether the specified primitives are visible or not. Commonly, these queries are performed on bounding boxes of a set of primitives after initialization of the depth buffer, by rendering a set of selected occluders. This selection is a difficult problem. In [HSLM02], the occluders selection is replaced by a grid decomposition of the scene and a front-to-back traversal that needs preprocess, making this solution unsuitable for dynamic scenes. All these problems are solved with our algorithm because we already have a depth buffer which is initialized by the efficient first visibility pass. Thus, in the high level point selection, any algorithm and data structure can be used for performing hardware occlusion queries on this depth buffer. Their choice depends on the application.

7.2. Sequential Point Trees

As we have mentioned above, the sequential point tree (SPT) [DVS03] is a poweful hierarchical data structure for fine LOD point selection that takes texture and local curvature into account. It is also designed for a best GPU usage because the traditional CPU-based hierarchical traversal is replaced by sequential processing on the graphics processor. The CPU only does a first efficient pre-culling, providing a large chunk of points. The fine granularity culling is performed by the GPU in a simple vertex program (they report a percentage of culled points going from 10% to 40%). Thus, using the SPT in a pure high quality point based rendering system is not efficient because much unnecessary data is processed by complex vertex programs (needed for high quality). However, used with our new high quality point based rendering algorithm, SPT becomes very efficient because it is only used during pass 2 with a very simple vertex program (simpler than the one used in the original SPT because there are no shading computations). In their system, they also perform back face culling by splitting the initial SPT list into an array of point lists with equal quantized normals (they use 128 quantized normals). But, for large complex scenes

© The Eurographics Association and Blackwell Publishing 2004.

we advocate replacing the normal clustering by a position clustering in order to perform more accurate high level view frustum and occlusion culling, by working on a small bounding box hierarchy instead of a single but large bounding box per object.

8. Conclusion and Future Work

In this paper we have presented a new accurate point selection algorithm that gives us the capability to perform expensive rendering computations only on visible points. Thus, it is particularly efficient for high quality point based rendering on the GPU. Our algorithm is easy to implement, independent of high level data structure, suitable for dynamic scenes and dynamic point clouds. We have also shown that it is very simple to add efficient high level occlusion culling and efficiently use the sequential point tree in a pure high quality point based rendering system.

Our results show that we can add more efficient/accurate high level culling algorithms to our system. For non-deformable objects, this task is relatively simple and many octree-based systems have already been presented. However, for deformable objects, it becomes much harder. With our accurate point selection it is also possible to add a more complex lighting model (e.g. for foliage or skin) without an expensive overhead since these additional computations are totally independent of the scene complexity. We also attempt to see if our methods can be used for efficient shadow generation on point based models.

Acknowledgements

We would like to thank Mohamed Hassan from the University of Cambridge for proof-reading the paper.

References

[BK03] BOTSCH M., KOBBELT L.: High-Quality Point-Based Rendering on Modern GPUs. In *11th Pacific Conference on Computer Graphics and Applications* (2003), IEEE, pp. 335–343. 1, 2, 3

[BWK02] BOTSCH M., WIRATANAYA A., KOBBELT L.: Efficient high quality rendering of point sampled geometry. In *Proceedings of the 13th Eurographics workshop on Rendering* (2002), pp. 53–64. 2

[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. 2

[COCSD03] COHEN-OR D., CHRYSANTHOU Y., SILVA C. T., DURAND F.: A survey of visibility for walkthrough applications. *IEEE TVCG* (2003), 412–431. 3

[DCSD01] DEUSSEN O., COLDITZ C., STAMMINGER M., DRETTAKIS G.: Interactive visualization of complex plant ecosystems. In *Proceedings of IEEE Visualization* (2001), pp. 37–44. 2

[DVS03] DACHSBACHER C., VOGELGSANG C., STAMMINGER M.: Sequential point trees. In *Proceedings of ACM SIGGRAPH 2003, Computer Graphics Proceedings* (2003), pp. 657–662. 2, 7

[GP03] GUENNEBAUD G., PAULIN M.: Efficient screen space approach for Hardware Accelerated Surfel Rendering. In *Vision, Modeling and Visualization* (2003), IEEE Signal Processing Society. 1, 2, 3, 5

[Hec89] HECKBERT P. S.: *Fundamentals of Texture Mapping and Image Warping*. Master's thesis, University of California at Berkley, 1989. 2

[HSLM02] HILLESLAND K., SALOMON B., LASTRA A., MANOCHA D.: *Fast and simple occlusion culling using hardware-based depth queries*. Tech. rep., Department of Computer Science, University of North Carolina, 2002. 7

[Kri03] KRIVANEK J.: *Representing and Rendering Surfaces with Points*. Tech. Rep. DC-PSR-2003-03, Department of Computer Science and Engineering, Czech Technical University in Prague, 2003. 2

[MMB97] MARK W. R., MCMILLAN L., BISHOP G.: Post-rendering 3d warping. In *Symposium on Interactive 3D Graphics* (1997), pp. 7–16, 180. 7

[PKKG03] PAULY M., KEISER R., KOBBELT L. P., GROSS M.: Shape modeling with point-sampled geometry. In *Proceedings of ACM SIGRAPH 2003, Computer Graphics Proceedings* (2003), pp. 641–650. 6

[PZvG00] PFISTER H., ZWICKER M., VAN BAAR J., GROSS M.: Surfels: surface elements as rendering primitives. In *Proceedings of ACM SIGGRAPH 2000, Computer Graphics Proceedings* (2000), pp. 335–342. 2, 5

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

[RPZ02] REN L., PFISTER H., ZWICKER M.: Object space ewa surface splatting: A hardware accelerated approach to high quality point rendering. In *Proceedings of Eurographics 2002* (2002). 2, 3

[SD01] STAMMINGER M., DRETTAKIS G.: Interactive sampling and rendering for complex and procedural geometry. In *Proceedings of the 12th Eurographics workshop on Rendering* (2001), pp. 151–162. 2

[ZPKG02] ZWICKER M., PAULY M., KNOLL O., GROSS M.: Pointshop 3d: an interactive system for point-based surface editing. In *Proceedings of ACM SIGGRAPH 2002, Computer Graphics Proceedings* (2002), pp. 322–329. 6

[ZPvBG01] ZWICKER M., PFISTER H., VAN BAAR J., GROSS M.: Surface splatting. In *Proceedings of ACM SIGGRAPH 2001, Computer Graphics Proceedings* (2001), pp. 371–378. 2, 5

© The Eurographics Association and Blackwell Publishing 2004.