Parallel Treecut-Manipulation for Interactive Level of Detail Selection

Daniel Schiffner
Goethe Universität
Robert-Mayer-Str. 10
Germany, 60054, Frankfurt (Main)
dschiffner@gdv.cs.uni-frankfurt.de

Detlef Krömker
Goethe Universität
Robert-Mayer-Str. 10
Germany, 60054, Frankfurt (Main)
kroemker@gdv.cs.uni-frankfurt.de

ABSTRACT

We present a dynamic system that allows to alter the Level of Detail (LOD) of a treecut-based object. The adaptation and selection is made in a parallel process which avoids stalling or locks because of expensive calculations and LOD changes. We present a method to control the exchange between the independent threads. Based on this separation, we present multiple strategies to perform the LOD-selection for point-based representations.

Keywords

Level Of Detail, Parallel LOD-selection, LOD-strategies, Thread Management.

1 INTRODUCTION

Level of Detail (LOD)-techniques are required in today's rendering environments to assure interactivity because of the ever growing number of primitives used [Hol11]. The selection of a LOD-representation, for example, can be based on the current view or object-related properties. However, these selections may require expensive computations, and thus, discrete LODs are preferred over continuous methods. We address this issue and present a system to allow parallel LOD-selection.

This LOD-selection can be derived using different strategies. These range from a simple recursive algorithm up to a priority-based selection. Using a perceptual metric in combination with a prioritization, the visual quality of an existing representation is preserved with respect to the human visual system. As the necessary calculations made by a perceptual metric can be expensive, the LOD-selection is performed in a parallel thread. So, stalling of the rendering is reduced to a minimum.

In this work, we describe our point-based rendering system and show how to manage the individual threads. Furthermore, we present multiple LOD-strategies that evolve an existing representation using only local operations.

Following to this introduction, we will give an insight into related work and then present our framework. This includes the synchronization of the evaluation as well as the LOD-strategies. We present performance and visual results that were achieved with the proposed framework and conclude with an outlook regarding future work.

2 RELATED WORK

Several methods to create a LOD-hierarchy exist. Typically, these levels are created by the hand of a designer who may be supported by a reduction algorithm. These pregenerated LODs are then used during rendering and are exchanged by some kind of metric.

It is possible to avoid the generation of a hierarchy or pregenerated LODs to reduce the amount of stored information. Especially interesting in this context are progressive representations. These produce intermediate solutions and are not restricted to fixed, i.e. discrete, levels. For this reason, these methods are referred to as continuous LOD. Hoppe [Hop96] presented a mesh-based reduction method, which was extended to point-based representations by [Wu05].

To create the individual levels, the reduction methods can exploit geometric information to increase details. [Gar97], for example, apply an error metric to increase the quality of the reduced versions. If such an algorithm is applied sequentially and the results are stored, a hierarchy is created. The current representation is defined by a vertex front or a cut / treecut.

Instead of generating details, the vertex front can be altered to select the representation based on the current hierarchy. Schiffner and Krömker [Sch10] use a treecut to adapt the representation by prioritizing nodes with high curvature. A similar approach is presented

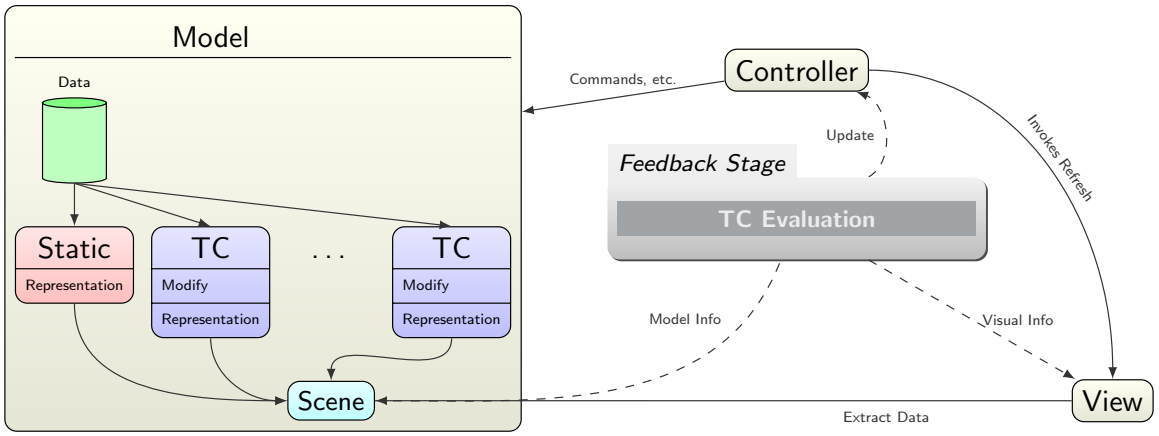


Figure 1: The design of our framework is based upon a Model-View-Controller (MVC) pattern. We introduce a Feedback Stage, which consists of a LOD-strategies and generates the new representation of the *TreeCut*. This component extracts information from both the Model and the View. Changes are relayed via the Controller.

by [Car11], where an octree cut is used to select data for visualization of large data sets. Both methods avoid a retraversal of the hierarchy and preserve the current vertex front, similar to Progressive Meshes [Hop96] or Progressive Splats [Wu05]. Both cuts includes a method to alter the current distribution. This idea will be used as a so-called LOD-strategy within this work.

Multi-threaded or parallel applications for rendering often focus on splitting the current representation into multiple viewports. Applications here range from Global Illumination [Hol11] to efficient large data set visualization [Gos12]. Recently, Peng et al. [Pen11] presented an approach to generate large crowds by parallel processing the individual models. Similar to our work, a scene is optimized for interactive renderings. They, however, use a fixed evaluation method.

3 SYSTEM DESIGN

As common in graphics applications [Shi10], our framework is based upon a Model-View-Controller (MVC) pattern. The Controller invokes the changes of the Model, i.e. the *TreeCut* [Sch10], which holds the current LOD representation. The View uses this current representation to derive a visual output using a graphics API.

We extend this default pattern by introducing a Feedback Stage (see figure 1). It is similar to an observer component, but with the ability to influence the Controller. Thus, the Feedback Stage can alter the Model which results in a different View. The Feedback Stage utilizes a strategy to apply the necessary LOD-operations. Therefore, the strategy may need to extract information from the rendered scene. This is visualized by the connection between the Feedback Stage and the View in figure 1. In the following, we will refer to the extraction of information and application of the strategy as an iteration.

Thread Management

We separate the Feedback Stage from the Controller, as it is an independent component. As it has access to all information required, it will be executed in parallel to the default rendering. During evaluation, the rendering of the old representation is continued. Due to the parallelization of both processes, some kind of synchronization needs to be included to avoid deadlocks or race-conditions.

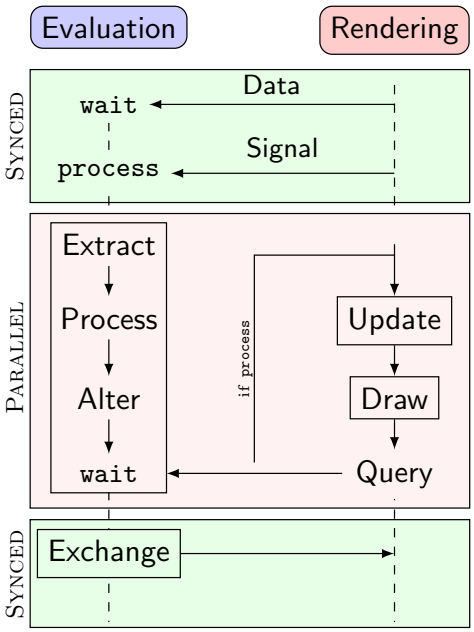


Figure 2: The processing sequence used to control the evaluation and rendering threads. While the evaluation is processing, the rendering thread displays the current LOD-version. Once the evaluation has completed, the data is exchanged and the LOD is updated gracefully. The query from the rendering thread is made without lock. This conditionally triggers the exchange of a new representation, which must be performed synchronized.

We propose a synchronization-strategy based on two states that are queried by the rendering thread: `wait` and `process`. This allows to add the evaluation with only small changes to the rendering code. The rendering thread only has to query for a new representation, while the Feedback Stage will handle the complete strategy evaluation and LOD-selection. The performed steps are visualized in figure 2.

We leverage the fact that the *TreeCut* is only represented with an index-list. The derivation of a new LOD thus only requires to generate a new index-list, which will be swapped or blended with the current one. This minimizes stall and flicker once a new representation is available. Only a pointer, or the VBO id, needs to be replaced. No copying of this data is performed during synchronization.

On initialization of the evaluation thread, the state is set to `wait`. In this state, all data can be accessed safely from the rendering thread. Here, no locking the data is required as no processing is performed. Only in this state the data will be exchanged. As stated before, the evaluation will generate an index-list, which can be swapped with the current one used for display.

If a new representation is requested, because the scene has changed or is considered invalid, the rendering process issues an update request to the evaluation via a signal. The evaluation thread is then set to the `process`-state. The rendering continues displaying the old representation as long as the evaluation is generating an updated version. After rendering a single frame, the evaluation is queried.

When starting to generate a new representation, the evaluation extracts the required data. This includes to copy the current index-list used by the rendering thread. As this is an read-only operation, no lock is required. LOD-strategies may need to aquire additional data from the View or the Model which can also be copied without a lock.

After completion of a single iteration, the evaluation thread will change its state to `wait`. As it is possible that the current representation is optimal for the applied strategy, a flag is used to indicate this case. This also allows to accelerate the LOD-strategies as they may terminate prematurely.

When the evaluation is in `wait`-stage again, the Exchange is executed. The Exchange does not cause a race-condition in both threads, because neither the rendering nor the evaluation requires access to the crucial data at this time. Additionally, we only require to exchange, i.e. swap, the used index-list if a new representation has been generated.

The Feedback Stage can be invoked again directly after the completion of the iteration. No additional updates to the Feedback Stage are required, as it extracts the current information in parallel.

4 LOD-STRATEGIES

Once a *TreeCut* has been established, only two core operations are applied (`refine`, `coarse`), which represent the changes in the detail. To alter the representation in a global manner, we apply LOD-strategies that are based solely on the current cut.

We include a threshold value to control the application of the individual operations. This counteracts repetitive `refine`s or `coarse`s of nodes. We, hereby, mean that a node is `refine`d in an iteration while it is `coarse`ned in the next.

In the following, we will present three different types of strategies: An optimization, a bucket-based approach and a recursive traversal of the hierarchy. The latter differs from the first ones because it operates on the complete LOD-hierarchy instead. It is included to show the universal applicability of the proposed thread management.

Optimization Strategy

The optimization LOD-strategy evaluates the current cut and applies a partial sorting based on a priority value, similar to [Car11]. This strategy requires some kind of limitation regarding the cut-size, e.g. a maximal node count. The priorities of the parent nodes should to be larger than their children to avoid artifacts. Otherwise, a parent node, i.e. a coarser representation, is favoured over a more detailed one.

During the Extract in the evaluation thread (refer to figure 2), the priorities are aquired. In our implementation, we use the curvature from the cut-nodes as priorities. The partial sorting is applied by iterating the complete cut and storing only the nodes with highest and lowest priority.

The algorithm selects nodes with highest priority for `refine`ment, while nodes with lowest priority are

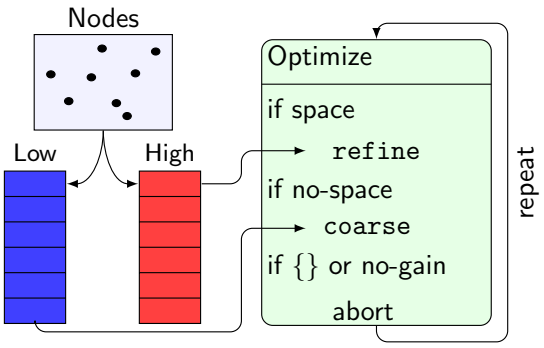


Figure 3: The optimization LOD-strategy for *TreeCut*-evaluation. Only the nodes with highest and lowest priority are processed. This accelerates the evaluation as it reduces the theoretical time complexity [Car11]. The most important ones are `refine`d, while the least important ones are `coarse`ned. This strategy requires a maximal node count to be applied

coarsened in the representation. A coarse frees space for further refinement with more important nodes. The operations performed by this strategy are visualized in figure 3.

As the partial sorting size can be considered constant during run-time, e.g. the size is not changed during an iteration, a linear time-complexity is given: $O(n \log k)$ with k being the constant partial sorting size and n the size of the current cut.

The threshold is defined as the minimal gain in priority required when altering the *TreeCut*. Therefore, the primitives in the sorted sequences (low and high) are compared before a coarse is applied. A refine is always executed as long as space is available.

Bucket-based Strategy

The second strategy assigns a target bucket to each node within the cut. The strategy alters the *TreeCut* to match a certain distribution as closely as possible. This strategy requires a method that determines the target bucket for a node.

An example for application is the generation of a stippling-like appearance of an object. The target bucket for each node is derived by using the illumination at the current node's location. The darker the current location, the more nodes are used within this region, i.e. the hierarchy is refined. The node is coarsened if a lighter representation is required.

In figure 4, an exemplary application of the LOD-strategy is shown. For each node, a target bucket is calculated. If this bucket differs from the currently assigned bucket, the delta is used to determine the according cut-operation. In the figure, a + denotes a positive delta and a refine needs to be applied, a - is a coarse. The 0 is the special case, that the node already has the correct bucket and no operation is necessary.

After the buckets have been calculated for all siblings, the operations are validated. As in the optimization LOD-strategy, a refine has higher precendence than a coarse. For this reason, the left branch is expanded in figure 4.

Special care has to be taken, if a coarse-operation needs to be applied. The parent node needs be inspected as well. The operation is only executed, if the bucket of the parent does not invalidate it. A small example will illustrate this scenario.

In the right branch of the tree in figure 4, a coarse needs to be applied. Therefore, the parent node is inspected (visualized by the question mark). In this case, the target bucket for the parent does not have a different delta (it is 0), i.e. it does not invalidate the operation. Thus, the coarse can be applied safely. The same applies, if the delta would be negative. If the parent would have a positive value, the node would be expanded in the next iteration. This would invalidate the operation and introduce a flicker into the represenation and the coarse is not executed.

As each node within the cut is evaluated, a linear time complexity is given: $O(n)$ with n being the cut-size.

For this LOD-strategy, the threshold is defined as the minimal delta that is required to force a coarse. We have achieved good results by a threshold of 0.

Recursive Strategy

The last strategy evaluates the complete LOD-hierarchy instead of the current cut. This method is inspired by the QSplat rendering system [Rus00]. Starting at the root node, the new cut is defined by the individual nodes when aborting the recursion. This abort is either due to culling, small splat area, or when no further refinement is possible, i.e. a leaf node is reached.

For this method it is required to additionally store the complete hierarchy, which is not the case for the other two LOD-strategies. During the Extract-step, this hierarchy is mapped to be accessible. The evaluation then starts the recursive traversal on a plain index-list.

The worst time complexity of this algorithm is $O(N)$ where N is the number of nodes within the tree. As the abort criterion includes culling, an acceleration is achieved, which results in an average logarithmic time complexity for large objects.

As opposed to the other two methods, the recursive strategy generates a new cut instead of manipulating an existing one. Thus, the definition of the threshold is not applicable to this strategy.

5 RESULTS

We tested the different evaluation strategies with our rendering system. We measured the rendering times and the overhead introduced by the usage of our system. The proposed LOD-strategies are compared to each other and the evaluation times in dependency of the original primitive count and the current count will be given as well.

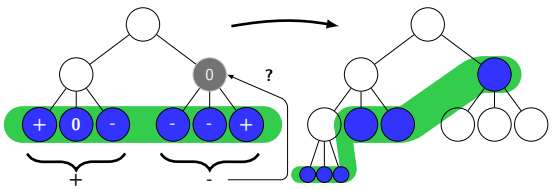


Figure 4: The bucket-based strategy for *TreeCut*-evaluation. Each node is assigned a target node. All siblings and the parent define the operation to be applied. If a coarse-operation is requested, the parent node is inspected (indicated by the question mark). Only if the operation is considered save, it is executed.

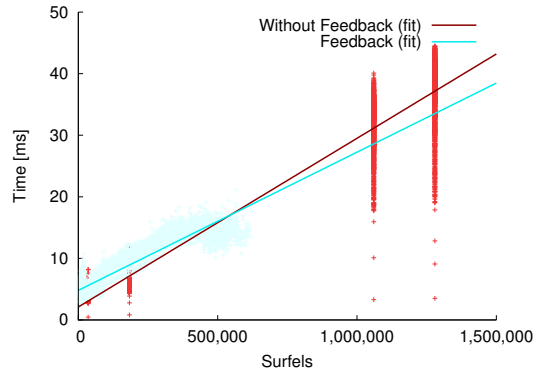


Figure 5: The performance impact when using the proposed Feedback System. The new representation generated is swapped from the evaluation thread to the rendering. Note that the increase is not required every frame, but only when an iteration has been completed. In our prototype, no changes are made to the representation during rendering. The average overhead is the difference between the fitted lines.

A sequential comparison is not included using the proposed strategies, but can be derived easily by summing up the rendering and evaluation times.

Additionally, we present some visual outputs generated by our renderer. For all renderings, we use a point-based rendering method that utilizes the Phong Splatting technique presented by [Bot05].

As noted before, we use the curvature as the priority value in case of the optimization LOD-strategy. The curvature identifies important regions on the surface of the object, and detail is preserved in regions where the surface changes. This was also presented in [Sch10] and [Lee05]. We preprocessed the curvature and included it within the LOD-hierarchy. Additionally, we assure that parent nodes have a higher curvature value than their children. During rendering of the scene, the maximal node count is set to be idendtical to the count given by the recursive LOD-strategy.

The bucket-based strategy uses the illumination information to derive a target bucket for each node. We use a fixed splat size for each node, and so create a stippling-like appearance of an object.

Finally, the recursive method implements the QSplat hierarchy and enables the basic QSplat method to be rendered efficiently using the Phong Splatting technique [Bot05] without further adaptation.

Time Measurements

Our prototype is written in `C++` and openGL. The tests were made with a Intel i5 with 3.47 GHz, 8.0 GB RAM and a nVidia GeForce 260 GTX with 896MB RAM. The graphics in figure 5 show the overhead introduced due to the exchange of the newly generated LOD-version after an iteration has been completed. Note that this increase is only generated if the evaluation is in `wait`-stage and a new LOD-version was created. The overall time falls below the version without the Feedback Stage (labeled Without Feedback) because rendering is accelerated and less primitives are required.

The graphs in figure 6 show the performance of the proposed strategies. We tested each strategy with multiple objects that are drawn in a predefined scene. During rendering, only one object and one light source is used. Both are rotated and moved to assure a large number of update request for the LOD-strategies. The objects are taken from the Standford 3d repository [3DScan]. As the same scene is used for all objects and LOD-strategies, the aquired evaluation times are comparable.

We omit information of the transfer of the data from the evaluation thread to the rendering thread as the generated data is only swapped.

As expected, both *TreeCut*-based strategies perform with linear time complexity. The bucket-based (refer to figure 6a) LOD-strategy does not change the size of the *TreeCut* as much as in the optimization LOD-strategy (refer to figure 6b). This is due to the fact that the bucket-based strategy is not limited by a maximal node count.

The optimization LOD-strategy allows to include any priority into an existing hierarchy. The evaluation remains linear despite the performed sorting. In the shown case, 8k elements are sorted.

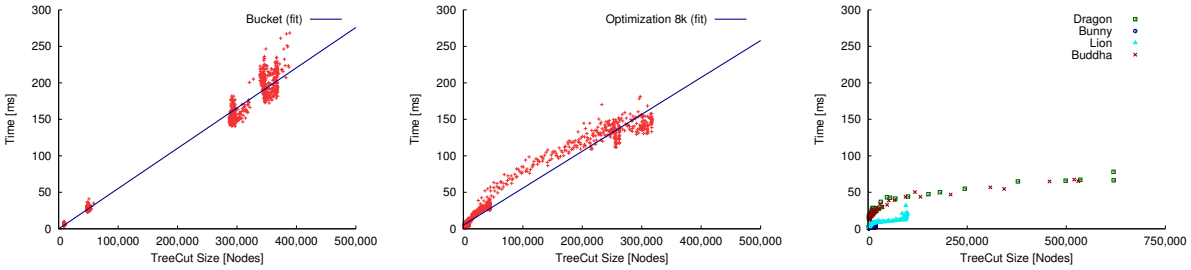
The recursive strategy applies the QSplat traversal presented by [Rus00]. The abort criterion includes backface culling, and for this reason, multiple nodes or surfels can be rejected early. This results in an overall acceleration of the traversal. For simple objects, the gain is not as large as with objects with higher geometric complexity. However, the strategy does not allow fine-tuning of a single representation. Only the recursive algorithm can be altered.

Visual Results

Some results achieved with the proposed LOD-strategies are shown in figure 7. A directional light source is used for illumination.

In figure 7a, we applied the bucket-based strategy along with an illumination-based target bucket function. We determine the target bucket by weighting the current illumination with respect to the depth of the node and maximal depth of the tree. This creates a stippling-like appearance of the drawn object. We enhanced Phong Splatting technique for the bucket-based approach to generate both a closed surface and equal-sized surfels.

The visual quality of the rendered version depends mainly on the used hierarchy. We enhanced the generation by using a node as parent instead of the average



(a) Bucket-based LOD-strategy performance graph.

(b) Optimization LOD-strategy performance graph.

(c) Recursive LOD-strategy performance graph.

Figure 6: The performance graphs of the proposed LOD-strategies. The values are given for an average single iteration. The LOD-strategies have been applied to mulitple objects with varying sizes. The first two offer linear time complexity, but the priority strategy has a larger overhead due to the required partial sorting. The recursive strategy is able to fast reject large portions of the hierarchy, which results in logarithmic time.

as proposed by [Rus00]. This suppresses motion of surfels when changing the LOD. Also, the more leaf nodes are available, the better the dark regions can be displayed.

A result generated with the optimization LOD-strategy is shown in figure 7b. The surfels are prioritized by the local curvature and surfel-size. This preserves details at regions where the object's surface is changing. Larger surfels are used in flat regions resulting in a reduction of the number of used surfels. In the shown image, only 45k surfels (original 183k) are used. The surfel-size has been added to avoid generation of too large splats that could mask detailed areas. This additional information is solely required for point-based representations.

Figure 7c shows the result created with the recursive LOD-strategy. Obviously, there is no difference in the visual quality compared to the original QSplat algorithm if plain splatting is used. However, the parallelization increases performance of the rendering. This is because the rendering can leverage VBOs and so avoids repetitive transfer of rendering data.

A higher visual quality is achieved by using the Phong Splatting technique. This can be used without any adaptation as the LOD-strategy is independent of the rendering. With the original QSplat, the multiple renderpasses of the Phong Splatting would require to traverse the hierarchy more than once, which would massively penalize the performance.

A rendering with the maximal available detail of a sample object (the Stanford lion) is depicted in figure 7d. It uses all 183408 leaf nodes and does not offer more details than the reduced versions (shown in figures 7b (45K) and 7c (87K)).

6 CONCLUSION AND OUTLOOK

Our approach increases the range of parallel processing existing LOD-hierarchies. The different LOD-strategies account for many scenarios, ranging from budget-based restrictions with perceptual optimization up to a bucket-based selection where nodes are assigned a specific level. Also, non-cut-based methods can benefit from the proposed system, which has been shown as well by including the QSplat algorithm.

As the object and selection is made in parallel, no stalling of the actual rendering occurs. In addition, the exchange between old and new representation can be made with blending to avoid flicker artifacts. The newly deduced LOD is not generated in advance, but created using information from the current representation. This allows a finer grained adaptation to a given scenario. Due to the design, the system can be included into existing LOD-management systems as a data provider for new LOD-versions.

All presented strategies have a low theoretical time complexity and show good performance in our prototype. The parallel processing of the data preserves interactivity of the rendering without being restricted to a fixed LOD-set. The synchronization is achieved by a simple query and thread-safe exchange is assured by design.

The LOD-strategies can be extended to account for information that is present in the scene. For example, the optimization strategy can include perceptual information acquired from the current scene. This increases the quality of the representation, while no new data needs to be generated. Especially interesting is the application of the *TreeCut* methods within the GPU to completely avoid transfer of data between CPU and GPU.

Yet, the system and the strategies are not optimal and need to be refined. Similar to other approaches, we plan to evaluate our system using many objects. The question arises, whether a centralized thread or a agent-based approach provides better results. Developers should be supported to decide which is the best for their scenario.

(a) Bucket-based strategy. The Stanford dragon is altered in dependency of the illumination.

(b) Optimization strategy. The surfels for rendering are prioritized by their curvature and surfel-size. This version uses 45k surfels.

(c) Recursive strategy. The surfels are generated by the QSplat algorithm (87k surfels).

(d) High quality rendering without the Feedback Stage (183k surfels).

Figure 7: Results generated with proposed LOD-strategies. A stippling like appearance can be created by determining the target bucket based on the current illumination. The second figure has been generated with the curvature and surfel-size as priority. The recursive splatting method presented by QSplat can be accelerated by employing a recursive evaluation strategy. The figure on the right shows the high detailed version of the Stanford lion.

We plan to include a complete perception model in the optimization LOD-strategy. In this case, a full 3d model, like [Sch11] or [Lee05], seems most suitable, as the perception information is extracted directly from the 3d data.

The bucked-based method does currently not include important properties like blue-noise [Hil01]. This information needs to be encoded within the hierarchy during generation of the LOD and has to be ensured during selection of the individual nodes as well. Also, the target bucket function needs to carefully select nodes for replacement.

Finally, the system itself needs be enhanced, so that the performance and the selection quality increases. We plan to extend it with environmental information, e.g. processing power or battery. This allows to selectively apply the LOD-selection on different hardware platforms, while being restricted to a universal, system-independent criterion.

7 REFERENCES

[3DScan] `http://graphics.stanford.edu/data/3Dscanrep/`.

[Bot05] Botsch, M., Hornung, A., Zwicker, M., Kobbelt, L. P. High-Quality Surface Splatting on Today's GPUs. Eurographics Symposium on Point-Based Graphics. pp.17-24. 2005.

[Car11] Carmona, R., Froehlich, B. Error-controlled real-time cut updates for multi-resolution volume rendering. Computers & Graphics. pp.934–944. 2011.

[Gar97] Garland, M., and Heckbert, P. S. Surface Simplification using quadric errormetrics. The art and interdisciplinary programs of SIGGRAPH 97. pp.209–216. 1997.

[Gos12] Goswami, P., Erol, F., Mukhi, R., Pajarola, R., Gobbetti, E. An Efficient Multiresolution Framework for High Quality Interactive Rendering of Massive Point Clouds using Multi-way kd-Trees. The Visual Computer 28. 2012.

[Hil01] Hiller, S., Deussen, O., Keller, A. Tiled Blue Noise Samples. Vision, modeling and visualization. pp.265–272. 2001.

[Hol11] Hollander, M., Ritschel, T., Eisemann, E., and Boubekeur, T. ManyLoDs: Parallel Many-View Level-of-Detail Selection for Real-Time Global Illumination. Computer Graphics Forum 30. pp.1233–1240. 2011.

[Hop96] Hoppe, H. Progressive Meshes. SIGGRAPH 96 conference proceedings. pp.99–108. 1996.

[Lee05] Lee, C. H., Varshney, A., Jacobs, D. W. Mesh saliency, Proceedings of ACM SIGGRAPH 2005. pp.659–666. 2005.

[Pen11] Peng, C., Park, S., Cao, Y., and Tian, J. A Real-Time System for Crowd Rendering: Parallel LOD and Texture-Preserving Approach on GPU. Lecture notes in computer science vol. 7060. pp.27–38. 2011.

[Rus00] Rusinkiewicz, S. and Levoy, M. QSplat: a multiresolution point rendering system for large meshes. SIGGRAPH 2000 conference proceedings. pp.343–352. 2000.

[Sch10] Schiffner, D., Krömker, D. Tree-Cut: Dynamic Saliency Based Level of Detail for Point Based Rendering. Sensyble 2010. pp.37–43. 2010.

[Sch11] Schiffner, D., Krömker, D. Three Dimensional Saliency Calculation Using Splatting. Sixth International Conference on Image and Graphics (ICIG). pp.835–840. 2011.

[Shi10] Shirley, P., Marschner, S. R., and Ashikhmin, M. Fundamentals of computer graphics 3rd edition. 2010.

[Wu05] Wu, J., Zhang, Z., and Kobbelt, L. P. Progressive Splatting. Eurographics Symposium on Point-Based Graphics. pp.25–32. 2005.