Real-Time Rendering of Continuous Levels of Detail for Sparse Voxel Octrees

Szymon Jabłoński
Institute of Computer Science
Warsaw University of Technology
ul. Nowowiejska 15/19
00-665 Warsaw, Poland
s.jablonski@ii.pw.edu.pl

Tomasz Martyn
Institute of Computer Science
Warsaw University of Technology
ul. Nowowiejska 15/19
00-665 Warsaw, Poland
martyn@ii.pw.edu.pl

ABSTRACT

In this paper, we present a novel approach to real-time, continuous and symmetrical level of detail (LOD) management of a 3D object represented by a sparse voxel octree (SVO). We propose a new method for continuous and symmetrical transition between two detail levels. The method is based on a SVO representation extended by redundant, helper nodes which are used to achieve a proper interpolation of geometry and material data. We extend redundant nodes with a transition direction attribute. Additional memory requirements are minimized by storing indices in a direction vector lookup table in object space. The new method is applied for an accurate evaluation of the required LOD. It uses an image-based evaluation function, i.e. the standard level transition function based on camera distance is extended by the real-time calculation of the current LOD pixel fill rate. We extend typical level transition function based on distance with real-time calculation of the current LOD pixel fill rate. Two different image based methods of SVO node pixel fill rate calculations using compute shaders or GPU queries and parallel reduce are presented. The developed LOD management algorithm is applicable for a raytracing and a rasterization-based rendering pipeline. The LOD transition algorithm allows to perform a dynamic and continues control of the SVO based objects which have not been available in other works. Moreover, the proposed fading algorithm based on the fade out direction and scaling allows for a LOD change without any graphical artifacts or loss of the virtual scene immersion.

Keywords

Computer graphics, level of detail, sparse voxel octree, voxel rendering, parallel reduce, image processing

1 INTRODUCTION

Computer graphic engines are perfect examples of the soft real-time systems [Tanen07]. A key requirement for the real-time system is the processing time measured in tenths of seconds or shorter. Interactive graphic applications, such as computer games or virtual reality, require that all necessary logic computation and rendering is performed within a few milliseconds. Due to the limited memory of GPUs, achieving satisfactory rendering results requires an implementation of several optimization methods in our graphics engine pipeline.

An important observation is that with the perspective projection objects that are far away from observer appear on the screen much smaller than objects that are near the observer. This implies that they can be rendered with less geometrical or material details. Thus, different techniques for controlling the object's current level of detail (LOD) have been developed to adapt the object's complexity to their importance within the virtual scene.

The LOD is one of the oldest problems in computer graphics. It was first presented in an article by James H. Clark in 1976 [Clark90] and was defined as the complexity of the 3D object located at a suitable distance from the observer. The main aim of using LOD control algorithms is to increase rendering efficiency by minimizing the data consumption, e.g. a number of polygons or voxels. Increasing computation power of today's GPUs allowed game developers to create highly detailed virtual scenes represented by millions of polygons. As a consequence, efficient LOD management algorithms are needed more than ever.

The tendency in recent years has been to improve the features of continuous LOD algorithms by using the possibilities offered by the graphics hardware, such as the tessellation shaders [Schaf14]. The main problem with these methods is that there are no special-

ized shaders to decimate geometry on GPU. Moreover, there is still the need to store a complex geometry in the VRAM. As a result, the most commonly used solution in today's graphics engines is to prepare several objects with different LODs and change them in real-time based on their importance within the virtual scene [Lueb02].

In this paper, we present a novel approach to real-time, continuous and symmetrical management of a 3D object LOD represented by the sparse voxel octree (SVO).

2 RELATED WORK

There is a wide selection of literature on performing the LOD evaluation and control algorithms. Over the years, many methods of geometry simplification and continuous LOD controlling have been developed. However, most of them are actually using polygonal representation of geometry and are connected to the LOD of virtual scene terrains based on height fields. We will focus on the papers that are most directly related to our work.

As we mentioned in the introduction, the most commonly used method is based on preparing a finite count of objects in different LODs by artists and change the currently used one based on the specified distance function [Lueb02]. It is very easy to implement and control in real-time. However, the objects representing a different LOD have to be stored in GPU memory. Alternatively, a streaming functionality must be implemented in the graphic engine. Moreover, the continuous transition between triangle based objects with a significantly different number of polygons is quite difficult.

Schoeder et al. introduced algorithms to decimate a triangle mesh [Schroeder92]. Rossignac and Borrel extended their approach by developing vertex clustering method that uses the bounding box of the source mesh divided into the grid with all of the vertices in a given cell replaced with a single vertex [Rossignac93]. Other groups of algorithms use an iterative approach based on primitive simplification operators [Lueb02] such as edge collapse or vertex removal. One of the commonly applied iterative decimation technique is the QSlim algorithm [Garland97]. In this method the pair collapse operator is iteratively replacing two vertices with one, causing neighboring faces to become degenerated. However, the mesh simplification has been traditionally viewed as a non-interactive process not readily amenable to the GPU acceleration. Using the modern programmable GPU, many algorithms have been extended and developed on the GPU. Ramos et al. developed a method of continuous geometry complexity controlling based on GPU triangle strip operations [Ramos06]. The vertex clustering method has been extended by using geometry shaders and a general-purpose data structure called the probabilistic octree. It enabled a simultaneous construction of multiple LODs and out-of-core simplification of extremely large polygonal meshes [DeCoro07, Schiffner15, Willmott11].

Modern GPUs offer functionality to increase geometry complexity in real-time [Schaf14]. By using programmable tessellation shaders, we can increase the number of mesh triangles by dividing triangle patches. It is possible to achieve full control of how the object geometry is dived and to calculate all required data attributes such as normal vector and texture coordinates. This method has been effectively used in case of terrain rendering based on height field [Ripol12]. Even by using low-resolution height map one can create highly detailed terrain with continuous, distance dependent LOD. Using tessellation shaders one can increase the complexity of the processing object exclusively on the selected areas. Unfortunately, today's graphics adapters do not offer any programmable shaders to decrease the complexity of processing object. In order to create a symmetrical method to increase and decrease object LOD, it is necessary to combine simplification and tessellation approach into one algorithm.

Although all of the presented methods propose interesting ideas related to the LOD management, none of them is able to provide a symmetrical, continuous and universal LOD control. A common feature of all of the presented methods is the polygonal representation of 3D objects. The polygonal representation does not provide any hierarchical information. A solution to that problem is the usage of the voxel representation. The SVO is the current standard method for representing and rendering voxels [Laine10, Bau11, Wil13]. Cyril Crassin was able to perform visualization of global illumination based on an SVO and voxel cone tracing [Crassin11]. The SVO is a hierarchical structure that, in addition to the significant voxel memory compression, offers object's LODs. Based on the voxelization resolution one can calculate the maximum tree depth which is the number of the object's LODs. A dynamic LOD is quite often mentioned in various articles. However, no one has ever described an algorithm showing how to change the current LOD and how to perform a transition between the levels in a continuous way.

3 LOD MANAGEMENT ALGORITHM

This section describes the fundamental features of LOD management algorithms. All algorithms can be divided into the two main components:

- **LOD evaluation** — in this part one needs to determine when to change the object's current LOD. The initiation of the change and its direction depends on, for example, the distance between the object and the observer or object size on the screen.

- **LOD transition** — this part of the algorithm deals with the way of how the current LOD is changing. In the discrete model algorithm, the change is realized by simple swapping of the object geometry or material data. In the case of the continuous algorithm in order to achieve proper object transition, one needs to implement the interpolation between two selected LODs.

The features of the developed algorithm are as follows:

- The 3D object is represented by an SVO.
- The LOD management algorithm is independent of the rendering method. It can be used with both the ray tracing approach and the voxel visualization achieved with the triangle rasterization pipeline.
- Helper data needed for the algorithm execution are minimized. In the case of ray tracing visualization, all necessary data can be calculated on the fly.
- The LOD transition is done smoothly using interpolation between two levels of SVO.
- The object geometry and material data LODs are increasing or decreasing symmetrically.

4 LOD EVALUATION

The most commonly used measurement to evaluate the object's current LOD is the distance between the object and the observer position. However, to do that, the scene designer must manually find the proper distance for each object to achieve satisfying results. The algorithm proposed in this paper estimates the distance using a newly developed image-based method utilizing the object rendering results achieved in pre-processing stage or by using rendering results of the previous frame.

4.1 Continuous evaluation function

In general, the LOD evaluation function can be expressed as:

$$y = f(x) \tag{1}$$

where:

y = object LOD
x = evaluation parameter

Based on the computed distance between the object and observer position the current LOD is determined in a way presented in Fig. 1. For simplicity, the linear dependence between the distance value and the LOD is assumed.

In this case Eq. 1 can be written in the following form:

$$y = max\left(0, min(N - \frac{dist}{x}, N)\right) \tag{2}$$

where:

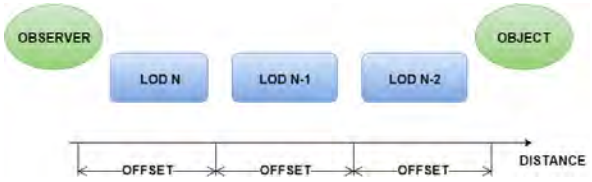

Figure 1: Linearly changing object LOD based on the distance.

y = object LOD
N = number of object LODs
$dist$ = distance between object and observer
x = defined distance offset between adjacent LODs

The computation results must be clamped to <*0,N*> in order to operate only on the existing object LODs. Assuming $N = 4$, $x = 2.0$ and $dist = 2.5$ we get:

$$y = max\left(0, min(4 - \frac{2.5}{2.0}, 4)\right) = 2.75 \tag{3}$$

Using mathematical round or truncate for the obtained result, we gain an integer value which represents the discrete LOD index. In addition, the floating result gives the possibility to control the continuous transition weight between two LODs.

The evaluation function presented above is based on the linear dependence between the distance value and the LOD. However, this approach can hardly give satisfying results on the virtual scene viewed with the perspective projection. More accurate results could be achieved using exponential or power functions.

The biggest problem with distance based evaluation functions is that there is no exact relationship between the distance and the resulting image. The rendering result will be different, but there are no algorithmic tools to describe scene complexity changes. In the next section, we propose a LOD evaluation method considering the image pixels as the main information carrier [Shannon48].

4.2 Pixel fill rate based evaluation function

Using compute shaders, data obtained from the rendering pass or GPU query objects according to the visualization algorithm, we can calculate how many pixels are filled by the draw operation [Wright10]. Thus, we can calculate how many pixels will be filled by some part of the object. The question is when we actually want to change the current LOD. If the object is so far away from the observer that there is no need to render it with the current LOD because we won't see any differences in rendering results, we can optimize the rendering process by decreasing the current LOD. Similarly, when the object is close enough to be rendered with a more complex geometry or material details we increase the current LOD.

5 PATTERN NODE SELECTION METHOD

In order to decide if the object is rendered with enough details, we analyze the rendering results of the smallest part of the object, the SVO node nearest to the observer position. In this paper, we define this special node as a *pattern node*. In order to find the pattern node, we propose two methods which differ in calculation precision and computing performance.

5.1 Approximate pattern node selection

The first method is based on a simple and naive approximation. For simplicity, we assume that the pattern node is exactly in the object bounding volume center position (even if actually there is no node in the selected 3D space). We can consider this method as an extension of the distance based evaluation method.

Having the object bounding volume and the node size of the current LOD, we render a single node off-screen. If our rendering pipeline is based on the ray tracing approach, we simply render the root node of the object scaled to the size of the current LOD node. In order to optimize the additional ray tracing step, we can perform ray tracing to a smaller render target than screen size. The minimum size of the target viewport can be easily calculated based on the object bounding volume and the camera matrices. Using, for example, atomic counters, we can calculate how many pixels will be filled by the draw operation. In the case of using a polygonal representation of the object, the same results can be achieved using GPU query objects.

As we mentioned before, it is a naive approach but can give satisfying results, especially on a low-power target like mobile devices.

5.2 Accurate pattern node selection

The accurate approach is much more advanced than the previous one. We want to find which SVO node of the object is the biggest on the screen. In other words, which one fills the most pixels of the resulting image. We must find the node in the current level of the SVO that is the closest to the observer position. We can calculate it on CPU by performing a simple software renderer but it is a computationally expensive solution hardly executable in real-time.

In order to obtain an accurate result, we need to perform the additional rendering and computation pass before the SVO visualization step. The accurate selection method will depend on the rendering algorithm. In this section we describe the pattern node selection for the ray tracing approach as well as for the triangle-based rasterization pipeline.

The foundation of the accurate pattern node selection algorithm is finding an SVO node with the smallest depth value for the current viewpoint. It means that before performing the final rendering, we need to find the node and count have many pixels will be filled by the node. The proposed solution can be implemented in a various way. In our work, we decided to use compute shaders. However, the developed solution can be implemented in other GPGPU interfaces such as CUDA or OpenCL.

5.2.1 Depth and node id texture generation

We propose an image based solution to find the pattern node utilizing the node depth information. By using the depth buffer, we find which pixel of the resulting image belongs to the object closest to the observer. The first step of our algorithm is an off-screen z-pass rendering. We use single channel render target texture with e.g. R32F or R16F internal format and save the linearized depth information. Further, the depth information has to be correlated to the SVO nodes.

The first step of our method requires defining unique ids for all SVO nodes. To optimize the node finding operation with a specified index it is recommended to create a lookup table for the tree nodes. Then, we extend the first pass of the algorithm by the saving node indices to the second texture channel. With the depth-only solution we could use floating point textures but unfortunately, we cannot save and retrieve integer or unsigned integer data from a float texture without data loss. A high-resolution voxel object will have ids counted in millions and we could lose the information. To solve this problem we use the unsigned integer type texture. Moreover, when saving the linearized depth information we perform normalization to the defined data range by multiplying depth values. We use a texture in the RG32UI internal format.

5.2.2 Minimum depth node seeking

The next step of our algorithm is to find the minimum depth from the rendered texture. If we need this information on CPU, we could transfer the texture data from GPU memory into the RAM. Due to the high cost of this data transfer, we might not be able to evaluate the LOD in real-time. In order to achieve real-time results, we can use two different solutions.

First of them is based on the parallel reduce algorithm on GPU [Buck04]. Using a compute shader, we create a new texture with half size of the source texture. Then, using a simple code we seek for the smallest depth value of N neighbors and store it in the new texture. By the defined number of iterations, we create a small texture with candidate nodes. We perform the parallel reduce algorithm until we create a 1x1 resolution texture and transfer it or read it on CPU. It is also possible to stop the iteration when our texture is small enough for being transferred to the CPU efficiently. In our implementation, we used the 1280x720 render target and 4

iterations of the parallel reduce algorithm resulting in a 80x45 resolution texture. We iterate through and find the pattern node on the CPU.

With the parallel reduce algorithm we can efficiently find the pattern node but we recommend an alternative solution. When performing the rendering operation, we can save the minimum depth of the rendered nodes with the corresponding node id by using the atomic operations and shader storage buffer objects. Thanks to that, we can use this information in the next step of our algorithm. Additionally, the access to the data stored in the shader storage buffer object on CPU side is very efficient. With a compute shader, we calculate how many pixels are filled by the found SVO node. This operation is performed in the same way for the ray tracing rendering approach and triangle based rasterization.

5.2.3 Optimizations and restrictions

The proposed solution requires an additional rendering step for finding the accurate pattern node. In some cases, this might be too expensive in order to fit the defined time requirements. We must render the 3D objects twice. In order to optimize our algorithm, we can use the previously rendered frame. In that case, apart from rendering the final image, we need to save the depth and voxel id data to an additional render target. After that, using the obtained minimum depth value and the node id stored in the shader storage buffer object, we can calculate the fill rate value by means of a compute shader.

The described algorithms give us the accurate results only when all voxels of a 3D object have exactly the same size. Otherwise, it is necessary to perform an additional calculation to obtain the proper pattern voxel. In order to obtain the correct interpolation weight, which will be used in the LOD transition stage, we must take into account the render target resolution.

5.3 Function boundary conditions

Last but not least an important part of the LOD evaluation algorithm is the boundary conditions of the LOD evaluation function. The defined evaluation boundaries are as follows:

- **Minimum condition** — defines the object minimum LOD. If the rendering of some object on the virtual screen produces N pixels corresponding to the minimum condition, there is no need to execute the evaluation and transition algorithm. This is the universal minimum boundary condition that is be used for any kind of voxel visualization algorithm. We defined N as a parameter, but the perfect function should use N defined as 1.
- **Maximum condition** — defines the object maximum LOD based on the position on the scene in relation to the observer. This boundary is very important because it defines when to stop the execution of the LOD evaluation function. It represents the situation when the object is rendered with the highest possible complexity. The maximum boundary condition is reached when exactly one voxel of the object corresponds to exactly one pixel of the resulting image. We can check this condition by comparing the filled pixel number with the voxel number used to render the image. The atomic counter can be used to calculate how many times a SVO node was rendered on the screen. After that, the sum of all used SVO nodes can be calculated.

6 PATTERN NODE BASED EVALUATION FUNCTION

The object's current LOD can be found using the pattern node pixel fill rate as an argument for the level evaluation function. The proposed evaluation method is based on the extended distance based function presented in section 4.1. The main difference is the usage of the pixel fill rate instead of the distance as the evaluation function parameter. Another very important difference is the type of the propagation function. The distance based approach discussed earlier assumed a linear dependence on the distance. In the case of objects that are represented by the SVO, an object rendered with the N-th LOD fills about four times more pixels than the same object rendered with the N-1-th level. Based on this fact we propose the LOD evaluation function described by Eq. 4 and 5.

$$y = max\Big(0.0, min(\frac{fillRate}{\sum_{i=0}^{N-LOD} maxRate * x}, 1.0)\Big) \quad (4)$$

$$y = \begin{cases} < minRate \Rightarrow \text{decrease level} \\ 1.0 \Rightarrow \text{increase level} \\ (0.0, 1.0) \Rightarrow \text{interpolate levels} \end{cases} \quad (5)$$

where:

y	= LOD interpolation weight
$fillRate$	= pattern node pixel fill rate
N	= number of object LODs
LOD	= object current LOD
x	= defined geometric progression value
$maxRate$	= defined max fill rate
$minRate$	= defined min fill rate

An additional step, which is not necessary with the distance-based approach is the calibration of the object's LOD. Before scene rendering, we must calculate the current LOD for all SVO based objects. In order to do that we use the following algorithm:

1. Find the pattern nodes for all LODs.
2. Calculate the pixel fill rate for all found pattern nodes.
3. Find the minimum fill rate value. Neglect values lower than the specified value or equal to zero.
4. Set the object's current LOD to the level with the minimum fill rate value.

As in the case of using the distance-based evaluation function, our algorithm requires the user input for the minimum and maximum fill rate for each object LODs defined with geometric progression.

For rendering results presented in section 8 we used the following parameters: maxRate = 16 pixels, minRate = 1 pixels and x = 1.

7 LOD TRANSITION

The core stage of the LOD management algorithm is the implementation of the current level transition method. The most straightforward solution to change the current LOD is to just stop the ray tracing algorithm at a specified level acquired from the evaluation pass. In the case of using triangle meshes, change object's vertex data buffers and materials. However, it may produce visible model swapping artifacts that adversely affect the perception and immersion of the virtual scene. Thanks to the voxel representation, the proposed algorithm is free from the limitations imposed by the polygons graphic representation.

The LOD control algorithm aims to change two attributes of the 3D object — geometry and material. In the voxel representation, both these attributes are related. With the SVO structure, each node can be divided into the maximum eight new nodes with different attribute values. When changing to a higher LOD, some child nodes may disappear creating changes in the object's geometry. Other nodes will just change the values of their attributes. A similar situation exists when the LOD decreases. Fig. 2 shows how the object geometry and material complexity changes between three LODs.

We can observe that if some node has a full set of childrens it is quite easy to perform the data interpolation between the parent node and child nodes. For example, using linear interpolation. The problem arises when some potential child node is missing, or when a parent has only one child. In order to accomplish the proper level transition, it is necessary to solve these issues. Below we describe the proposed algorithm for the SVO based object geometry and material transitions.

7.1 Object material transition

The new type of an SVO node is introduced. We call it *redundant node* based on its actual meaning for the object representation. The redundant node can be the

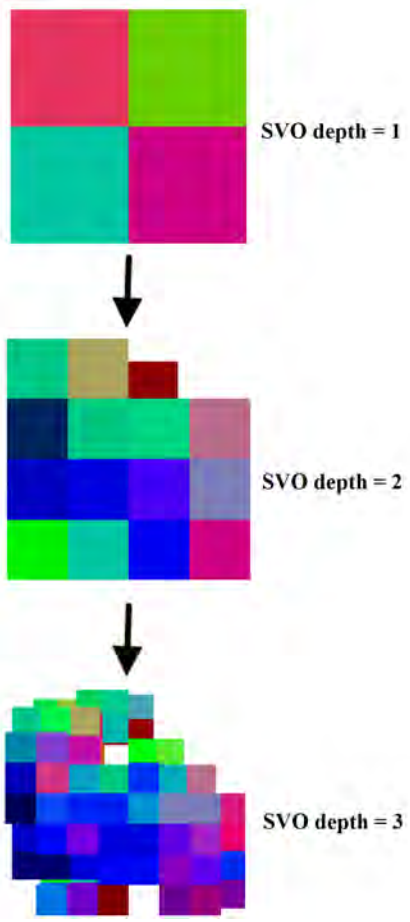

Figure 2: Object differences between three levels of Stanford Bunny [Stanford11].

actual part of the SVO, or it is just an information about the additional node in the tree. If the last traversing node has not the full set of children - we fulfill the gaps with the redundant nodes. We treat the last traversing level of the tree as it has a full set of child's. The main idea behind the redundant node is that in order to perform the interpolation between the parent and the child nodes the number of nodes at both levels must be the same. This is necessary for a continuous transition from one level to another.

All interpolation operations are performed between the parent and child node. Let's start with the parent node representation. As we mentioned before, each node can be divided into maximum eight child nodes. This means that we can treat the parent node as eight identical nodes with the same attributes. The more complex issue is with the children nodes. If the current node does not have eight children we must replace the missing nodes with the redundant nodes. Such nodes will have identical attributes as the parent node. Thanks to that, on both tree levels we will have an identical number of nodes. Fig. 3 presents the idea of using redun-

dant nodes for the two-dimensional grid. For a 3D data structure like the SVO, the method is identical.

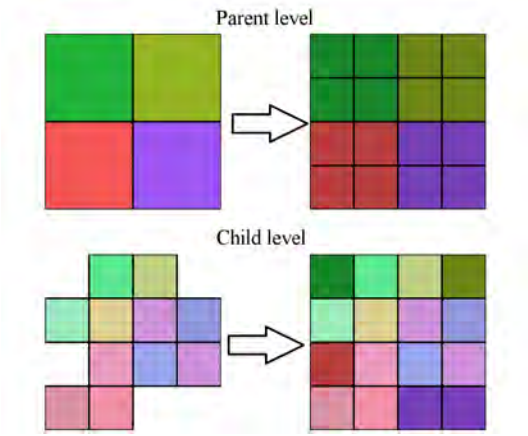

Figure 3: The idea of the dividing a parent node to eight identical nodes with redundant child nodes. Divided parent nodes and redundant nodes are highlighted with stripped lines.

7.2 Object geometry transition

Using the parent node division with the redundant child nodes we achieved the possibility to perform the data interpolation between two LODs. However, the redundant nodes must somehow disappear at the end of the level transition. The most straightforward way to achieve this is by using alpha blending with transparency. Unfortunately, this solution affects other nodes and create unacceptable artifacts.

We propose an alternative solution based on scaling the redundant nodes. Parallel to the data interpolation, with the interpolation weight parameter we change the size of the redundant nodes from the initial values to zero. However, the scale transformation in the defined origin causes the formation of holes at the edges of the objects. Thus, there is the need for an additional redundant node position control, so that they will be absorbed by the nearest neighbor and disappear in a more natural way.

In order to implement redundant nodes fading out we need to find the node's nearest neighbor and calculate the fading direction. The SVO structure guarantees that each node has a connected neighbor. If we cannot find any candidates on the children level, we seek for it at the parent level. The only exception to this rule is the tree root node. The direction vector for the root node will never be required. In the case of the ray tracing approach, we have access to the neighbor nodes during object rendering. If we do not have an access to the neighbor nodes during object rendering, we need to store pre-calculated values in an additional node attribute. Fortunately, we have a finite number of possible directions. A node can have maximum 7 possible neighbors on the node level and 8 possible candidates on the parent level. In order to minimize memory requirements, we can create a lookup table for direction vectors and store only an index to the vector. However, it is only required when our visualization algorithm is not based on the ray tracing approach. Fig. 4 presents an example of performing a continues transition.

Figure 4: Example of level transition based on the developed method.

8 RESULTS

In this section, we present rendering results of the developed algorithm. It is very difficult to present continuous LOD management on static images or even video samples. A fundamental feature of the continuous LOD transition is to hide level changing from the observer. Fig. 5 - 6 demonstrates rendering the result of the developed LOD management algorithm.

9 CONCLUSIONS AND FUTURE WORK

We have developed a novel approach for efficient real-time rendering and controlling the SVO LOD. Our method can be used to algorithmically evaluate the current LOD and perform a transition between two levels. The pixel fill rate method instead of a distance parameter allows for better control of virtual scene rendering results. Moreover, regardless of the chosen rendering method, we propose a universal pattern node evaluation method that can be used in real-time. In the case of the ray tracing approach, the required additional data can be obtained from the rendering pass or based on the previous frame. In the case of the triangle based rasterization method based on the parallel reduce algorithm and GPU queries offers real-time performance.

The LOD transition algorithm allows to perform a dynamic and continues control of the SVO based objects which is our main contribution. By extending the SVO structure with a new type of node called redundant node we achieved the full control of the level interpolation stage. Moreover, the proposed fading algorithm based on the fade out direction and scaling allows for a LOD change without any graphical artifacts or loss of the virtual scene immersion. The developed method is applicable for various voxel rendering algorithms. Moreover, the potential increase of memory consumption for additional data has been minimized.

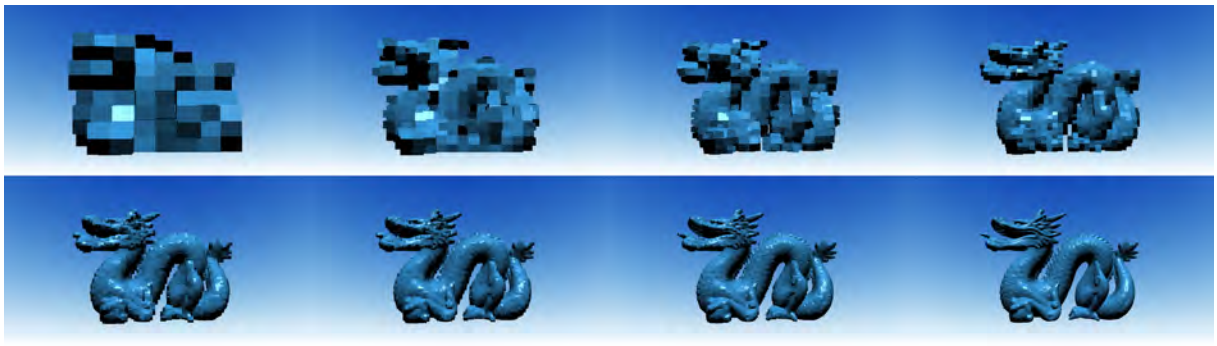

Figure 5: Object geometry and material transition example.

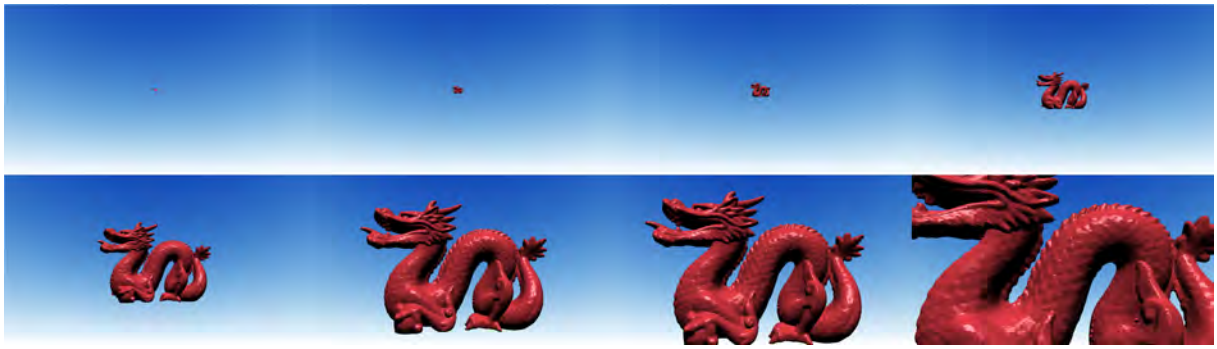

Figure 6: Example of the LOD management algorithm based on pixel fill rate evaluation for the single test object.

An obvious step forward would be to experiment with the control of the entire virtual scene with numerous SVO objects. The current method is dedicated to controlling a per object LOD. Moreover, the proposed method does not perform the LOD evaluation in the view-depended style. We control the whole object details even when we see just part of the object.

Last but not least, an interesting research can be conducted on the situation when we reached the highest LOD of the current object and still could get closer to the object. In that case, the interesting solution seems to be the procedural generation of the geometric complexity using e.g. displacement maps and tessellation.

10 REFERENCES

[Bau11] Bautembach D., Animated sparse voxel octrees, Bachelor Thesis, University of Hamburg, 2011.

[Buck04] Buck, I., and Purcell, T., A Toolkit for Computation on GPUs, In GPU Gems, Addison-Wesley, 2004, pp. 621-636.

[Clark90] Clark, J.H, Seminal graphics. New York, NY, USA: ACM, 1998, ch. Hierarchical Geometric Models for Visible Surface Algorithms, pp. 43-50. [Online]. Available: http://doi.acm.org/10.1145/280811.280921

[Crassin11] Crassin, C., Neyret, F., Sainz, M., Green, S., and Eisemann, E., Interactive indirect illumination using voxel cone tracing, Computer Graphics Forum (Proceedings of Pacific Graphics 2011), vol. 30, no. 7, sep 2011.

[DeCoro07] DeCoro, C., and Tatarchuk, N., Real-time Mesh Simplification Using the GPU. Symposium on Interactive 3D Graphics (I3D) 2007, pp. 6, April 2007.

[Garland97] Garland, M., and Heckbert, P. S. 1997. Surface simplification using quadric error metrics. Proceedings of ACM SIGGRAPH 1997, 209-216.

[Laine10] Laine, S., and Karras, T., Efficient sparse voxel octrees, in Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, ser. I3D 2010. New York, NY, USA: ACM, 2010, pp. 55-63.

[Lueb02] Luebke D., Watson B., Cohen, J., D., Reddy, M., and Varshney, A., Level of Detail for 3D Graphics. New York, NY, USA: Elsevier Science Inc., 2002.

[Ramos06] Ramos, F., Chover, M., Ripolles, O., and Granell, C., DGCI, volume 4245 of Lecture Notes in Computer Science, page 460-469. Springer, 2006

[Ripol12] Ripolles, O., Ramos, F., Puig-Centelles, A., and Chover, M., 2012. Real-time tessellation of terrain on graphics hardware. Comput. Geosci. 41 (April 2012), 147-155.

[Rossignac93] Rossignac, J., and Borel, P., 1993. Multi-resolution 3D approximations for rendering complex scenes. Modeling in Computer Graphics:

Methods and Applications (June), 455-465.

[Schaf14] Schäfer, H., Nießner, M., Keinert, B., Stamminger, M., and Loop, C., State of the art report on real-time rendering with hardware tessellation, 2014.

[Schiffner15] Schiffner, D., Stockhausen, C., Ritter, M. Surfaces for Point Clouds using Non-Uniform Grids on the GPU, Short papers proceedings WSCG2015.

[Schroeder92] Schroeder, W. J., Zagre, J. A., and Lorense, W.E.1992. Decimation of triangle meshes. In SIGGRAPH 92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques, ACM Press, New York, NY, USA, 65-70.

[Shannon48] Shanno, C., E., A Mathematical Theory of Communication, Bell System Technical Journal 27(3).

[Stanford11] The Stanford 3D Scanning Repository, Stanford University, 22 Dec 2010, Retrieved 17 July 2011.

[Tanen07] Tanenbaum, A. S., Modern Operating Systems, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007.

[Wil13] Willcocks, C. G., Sparse volumetric deformation, Ph.D. dissertation, Durham University, 2013.

[Willmott11] Willmott, A., Rapid Simplification of Multi-attribute Meshes, Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics, 2011.

[Wright10] Wright, R., S., Haemel, N., Sellers, G., Lipchak, B., OpenGL SuperBible: Comprehensive Tutorial and Reference, Addison-Wesley Professional, 2010.