Conservative Out-of-Core Rendering of Large Dynamic Scenes Using HDLODs

Vitaly Semenov [1,2,3], Vasily Shutkin [1] and Vladislav Zolotov [1]

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, Alexander Solzhenitsyn st., 25, Moscow, 109004, Russia*
[2] *Moscow Institute of Physics and Technology, 9 Institutskiy per., Dolgoprudny, 141701, Russia*
[3] *National Research University Higher School of Economics, 11 Pokrovsky Bulvar, Moscow, 109028, Russia*

Abstract
Rendering of large scenes in external memory is one of the most important problems of computer graphics, which is used in such areas as CAD/CAM/CAE, geoinformatics, project management, scientific visualization, virtual and augmented reality, computer games and animation. Unlike static scenes, for which a number of effective approaches have been proposed, in particular, levels of detail (LOD), rendering of large dynamic scenes with a given level of the tolerance and trustworthiness remains a big challenge. This paper discusses and investigates the possibility of using the previously proposed method of hierarchical dynamic levels of detail (HDLOD) for conservative rendering of dynamic scenes with a deterministic nature of events in external memory. The carried out series of computational experiments prove the feasibility and effectiveness of the method for real industrial scenes when employed in combination with memory management techniques.

Keywords
Out-of-core rendering, dynamic 3D scenes, levels of detail, HDLOD

1. Introduction

While the problem of out-of-core rendering of static scenes has been successfully solved and a lot of promising approaches and graphic systems have been developed and introduced into practice, rendering of dynamic scenes larger than main memory remains an acute urgent problem. The class of graphic applications in which scenes exhibit dynamic behavior and occupy extremely large amount of memory is quite wide and includes CAD/CAM/CAE systems, GIS, 4D project modelling systems, scientific visualization, computer games and animation, virtual and augmented reality. Unfortunately, the continuous growth of the complexity of displayed scenes is not compensated by the modernization of graphics hardware, although there are quality improvements. For this reason, new approaches are needed to render complex dynamic scenes with reasonable realism and fidelity, and when possible, with interactive frame rates. As opposed to the interactive or approximate mode, the presented paper focuses on the so-called conservative rendering aimed at ensuring a given level of tolerance and trustworthiness. The conservative mode is especially important for industrial applications that detect spatio-temporal collisions and interferences in scenes, as well as for applications that create photorealistic graphics and video-presentations. The performance requirements in such cases are relaxed in favor of high-quality conservative rendering.

The problem of rendering complex scenes has been studied intensively for over many decades. Proposed methods include polygonal model simplifications, replacing distant geometry with imagery, levels of detail (LOD) and hierarchical levels of detail (HLOD), view-frustum and occlusion culling, spatial indexing (primarily octrees and kd-trees), from-point and from-region visibility analysis, parallel

GraphiCon 2021: 31st International Conference on Computer Graphics and Vision, September 27-30, 2021, Nizhny Novgorod, Russia
EMAIL: sem@ispras.ru (V. Semenov); v451ly@ispras.ru (V. Shutkin); vladislav.zolotov@ispras.ru (V. Zolotov)
ORCID: 0000-0002-8766-8454 (V. Semenov); 0000-0002-9238-5029 (V. Shutkin); 0000-0002-1988-6548 (V. Zolotov)

loading and rendering, speculative prefetching, geometry caching, approximate display, draw call optimizations, batching and texture packing.

Out-of-core rendering systems usually employ one or another combination of these methods. iWalk [1] uses octree to build on-disk hierarchical representation for a large model. It uses multi-threaded rendering approach that combines speculative prefetching with from-point occlusion-culling algorithm based on prioritized-layered projection (PLP). Papers [2] and [3] both propose algorithms that are based on the HLOD method [4]. In [2], the scene graph representation is decoupled into skeleton and object-reps. The skeleton stores connectivity information of the scene graph, as well as auxiliary data like nodes bounding boxes and geometric error. The skeleton must be loaded into main memory, although it is typically not a problem because the skeleton has very small memory footprint. Object-reps (that are levels of details for the nodes of the scene graph) are stored on disk. The algorithm utilizes two processes that work in parallel. The first process traverses the scene graph and computes so-called front — the list of object representations that have to be rendered on this frame. The second process loads the object-reps in parallel and performs prefetching when there is time for that. Algorithm in [3] uses different approach — nodes are loaded successively starting from root with maintaining the following property: if a node has geometry loaded, so does all of its ancestors. This way the algorithm doesn’t require the full skeleton to be in main memory. Quick-VDR algorithm [5] represents the model as a clustered hierarchy of progressive meshes (CHPM). The cluster hierarchy is used for coarse-grained view-dependent refinement, whereas the progressive meshes provide fine-grained local refinement. The algorithm also performs occlusion culling. One frame of latency is added to fetch newly visible clusters from disk. Paper [6] presents an interactive out-of-core rendering frameworks that uses real-time ray tracing and a hierarchical, hybrid volumetric/lightfield-like approximation scheme for representing not-yet-loaded geometry. MMR system [7] renders near geometry using levels of details and far geometry is rendered as a textured depth mesh. The system automatically balances the screen-space errors resulting from geometric simplification with those from textured-depth-mesh distortion. Cesium 3D Tiles [8] is an open specification for sharing, visualizing, fusing, and interacting with massive heterogenous 3D geospatial content. It is based on HLOD concept and implies using JSON file for storing skeleton data and glTF files for geometric representations of the hierarchy nodes.

HLOD method is a good basis for designing an out-of-core rendering algorithm. However, HLOD and most of existing out-of-core rendering algorithms are developed primarily for static scenes. They preliminary build certain data structures that are then utilized during rendering. When dynamic changes occur in the scene those data structures must be recomputed. The time required to recompute is typically greater than that for rendering the scene. That is why the authors of HLOD method admit that it is most suitable for the case of “infrequent motion” [4]. The situation with out-of-core algorithms is even worse, because in this case updating the data on disk would be required.

The previously proposed method of hierarchical dynamic levels of detail (HDLOD) looks more promising for the purposes declared. It also assumes preliminary hierarchical clustering of objects, but takes into account not only their spatial proximity, but also the similarity of dynamic behavior [9]. As a result, each cluster encapsulates both geometry model, material and dynamic behavior and, therefore, can be reused unchanged over entire time intervals, rather than in particular time moments. HDLOD trees can be systematically traversed and processed to render the original scene with a certain threshold of the geometric error. It is noteworthy that the specified threshold can be provided for each viewing position at any time. Previous studies have proven the high efficiency of the method when HDLOD trees are allocated in main memory. Finally, the method is applicable for wide classes of large dynamic scenes with a deterministic nature of temporal events such as the appearance, removal and movement of objects.

This paper presents an out-of-core generalization of the HDLOD method, suitable for conservative rendering of very large dynamic scenes, whose HDLOD representations are located in external memory and can be loaded into main memory only in parts. The underlying HDLOD method is shortly described in section 2 with emphasis on sorts and representations of spatio-temporal clusters. The out-of-core generalizations of the method, in particular, splitting the HDLOD cluster tree into a skeleton and a complement are specified in section 3. It also provides some details on memory management and caching techniques necessary for the method implementations. The carried out computational experiments are specified in section 4 and the results are summarized in conclusions.

2. HDLOD method

Suppose that a scene $S = \{s(g_s, p_s(t), v_s(t))\}$ is represented as a set of objects s that are defined in a 3D Euclidean space E^3 on a simulation time interval $[0, T]$. Each object $s \in S$ has an invariable geometric representation g_s. The presence (or availability) of the object s in the scene is determined by its presence function $v_s(t)\colon [0, T] \to \{0,1\}$ in such a way that the function $v_s(t)$ is one if the object appears in the scene at the instant t and it is zero if it does not. The position of the object s is determined by its position function $p_s(t)\colon [0, T] \to P$, where P is a set of 4×4 matrices or any other structures used for determining the location and orientation of the scene objects.

In terms of dynamic behavior, all objects in the scene can be divided into three non-overlapping classes.

The class of *static* objects includes the objects whose presence and position do not change throughout the entire simulation period, i.e., $\forall\, t \in [0, T]$ $v_s(t) = 1$, $p_s(t) = p_s$, where p_s is the unchanged position of the object s. In further consideration, for clarity, such objects will be denoted as $s(g_s, p_s, v_s)$.

The class of *pseudo-dynamic* objects consists of the objects that appear and disappear without changing their positions: $\forall\, t \in [0, T]$ $p_s(t) = p_s$, but $\exists\, t_1, t_2 \in [0, T]$ such that $v_s(t_1) \neq v_s(t_2)$. Further, such objects will be denoted as $s(g_s, p_s, v_s(t))$.

Finally, the class of *dynamic* objects implies that both the presence function and position function of the object vary with time: $\exists\, t_1, t_2, t_3, t_4 \in [0, T]$, such that $v_s(t_1) \neq v_s(t_2)$ $p_s(t_3) \neq p_s(t_4)$. In further, to emphasize the dynamic character of such objects the notation $s(g_s, p_s(t), v_s(t))$ will be used.

We refer to the hierarchical dynamic levels of detail (HDLOD) as a tree of clusters $C = \{c(g_c, p_c(t), v_c(t), b_c(t), \delta_c(t))\} \cup \{\prec\}$ represented by a set of clusters c and a set of agglomeration relations $\prec$. Each cluster c is specified by its geometric representation g_c and the following behavior functions defined over the entire simulation interval $[0, T]$. These are presence function $v_c\colon [0, T] \to \{0,1\}$, position function $p_c\colon [0, T] \to P$, bounding volume function $b_c\colon [0, T] \to B$, where B is a set of volumes in E^3, and geometric error function $\delta_c\colon [0, T] \to [0, \Delta_c]$, where Δ_c is the maximum local deviation of the cluster geometry from its originated objects throughout the simulation interval $[0, T]$. Geometric error function $\delta_c(t)$ quantifies the spatial error of the simplified geometry compared with the original objects taking into account its presence status at a particular time point $t \in [0, T]$. The function $b_c(t)$ enables to determine the bounding volume of the cluster c at every moment $t \in [0, T]$. The bounding volume can be axis aligned bounding box (AABB), oriented bounding box (OBB) or any other volumetric structure, including bounding volume hierarchies, potentially mobile and deformable in time. The only significant requirement for bounding volumes, is computationally inexpensive checks of the belonging of volumes to given primitive shapes, e.g. visibility cones, and making verdicts on the need to display the clusters contained within them.

Each agglomeration relation $c' \prec c$ reflects the order in which the parent cluster $c \in C$ was formed from simplified geometric and behavioral representations of the child cluster $c' \in C$. Clusters at the top of the tree contain representations of the renderable content with lowest levels of detail, while bottom clusters provide content with higher levels of detail. Geometry simplification methods are well studied and have been widely used in the HLOD implementations. Methods of simplifying behavior have not yet been studied, therefore, more clarifications are needed. Since the objects agglomerated into clusters can exhibit different behaviors, the concept of presence takes on a broader interpretation related to the visual fidelity, which can be achieved by displaying some of the clusters, or vice versa, by ignoring them. The concept provides for cases when a parent cluster is considered absent at some point in time, while some of its children are present at the same moment, and, conversely, the parent cluster can be considered present, while some children are absent. Therefore, the presence status should always be understood in the context of the tolerance that can be provided when the cluster is displayed or, conversely, ignored.

Similar to the introduced classification of scene objects, HDLOD clusters can also be divided into static, pseudo-dynamic and dynamic. Static clusters are those that agglomerate only static objects and clusters. Their positions, bounding volumes, presence statuses and tolerances do not depend on the time parameter and therefore can be denoted as $c(g_c, p_c, v_c, b_c, \delta_c)$. Pseudo-dynamic clusters are clusters that agglomerate pseudo-dynamic and possibly static objects. Their positions and bounding volumes are fixed, while presence statuses and tolerances may depend on the time parameter $c(g_c, p_c, v_c(t), b_c, \delta_c(t))$. Dynamic clusters are those that agglomerate dynamic and possibly pseudo-dynamic and static objects and therefore have all attributes as functions $c(g_c, p_c(t), v_c(t), b_c(t), \delta_c(t))$.

Regardless of the specific HDLOD clustering method used, the following assumptions are made and must be guaranteed:

1. The bounding volume of the parent cluster $b_c(t)$ covers the bounding volume of any child cluster $b_{c'}(t)$ so that $b_{c'}(t) \subseteq b_c(t)$ at every time point $t \in [0, T]$;
2. The geometric error of the parent cluster $\delta_c(t)$ exceeds the error of any child cluster $\delta_{c'}(t)$ so that $\delta_{c'}(t) \leq \delta_c(t)$ at every time point $t \in [0, T]$;
3. The geometric error for leaf clusters is taken to be zero $\delta_c(t) = 0$ ($\forall c \in C\ such\ that\ \nexists c' \in C\ c' \prec c\ \delta_c(t) = 0$), thereby assuming that their geometric and behavioral representations are accurate and can satisfy any tolerance requirements. In most cases, this can be achieved by using the original objects of the scene as leaf clusters.

More detailed information on clustering method can be found in [9]. Under these assumptions, the scene can be effectively rendered with a certain screen-space error (sse) explicitly expressed in terms of pixels of the screen. When rendering the scene, the cluster tree is traversed. At each cluster, the functions are evaluated to determine if the cluster can be immediately rendered or further traversal of the subtree is required to find and employ proper clusters.

First of all, it is checked whether the bounding volume of the cluster $b_c(t)$ falls within the view frustum. If so, the next check is performed. Otherwise, traversal is stopped and the cluster and its subtree are excluded from further analysis. Then it is checked whether the screen-space error caused by the cluster tolerance exceeds the given threshold sse. For this purpose, the geometric error of the cluster at the current modelling time is transformed to the screen-space error according to a standard perspective projection and is compared with the threshold:

$$\delta_c(t) \leq 2\ d\ tg \frac{\left(\frac{\alpha}{2}\right) sse}{r},$$

where r is the resolution (or height) of the rendering screen in pixels, d is the distance from the camera point to the point on the bounding volume of the cluster which is closest to the camera, α is the opening angle of the view frustum. If the condition is satisfied, the cluster is selected for rendering. If not, the traversal of the cluster subtree continues and its children also undergo the same checks.

```
void display_tree (Hdlod tree, Time time, Frustum frustum, Real sse)
{
        visit (tree.root, time, frustum, sse, true);
}
void visit (HdlodCluster cluster, Time time, Frustum frustum, Real sse, Bool frustum_culling)
{
        OverlapType overlap = INSIDE;
        if (frustum_culling)
                overlap = check_overlap (cluster.bounding_volume (time), frustum);
        if (overlap EQUAL OUTSIDE)
                return;
        if (cluster.children.size() EQUAL 0) {
                display_cluster (cluster, time);
                return;
        }
        if (geometric_error_to_sse (cluster.error (time), frustum) LESS_THAN sse) {
                display_ cluster (cluster, time);
                return;
        }
        for each (HdlodCluster child in cluster.children)
                visit (child, time, frustum, sse, overlap NOT_EQUAL INSIDE);
}
void display_ cluster (HdlodCluster cluster, Time time)
{
        if (NOT cluster.is_present (time))
                return;
        render (cluster.geometry, cluster.matrix (time));
}
```

Figure 1: Pseudocode of the HDLOD rendering algorithm

If the cluster is selected for rendering, it's presence status $v_c(t)$ is determined. If the cluster is present, its content is rendered. Otherwise, traversal is stopped and nothing is displayed. The traversal is guaranteed to terminate when it reaches leaf clusters with exact content. Thus, checks against bounding volume, error and presence status are done when traversing the cluster tree to decide on the need of rendering the cluster.

The pseudocode of the HDLOD rendering procedure *display_tree* with formal parameters corresponding to the HDLOD tree representation, current simulation time, current view frustum, and required screen-space error is presented in Figure 1. The procedure is based on the visiting function *visit* with similar parameters. The visiting function is recursively called to traverse clusters and render them using the *display_cluster* function. At the start of each frame the visiting function is invoked for the root of the tree.

The procedure *render* is intended to render the geometric model of the cluster at a given location with a proper orientation as determined by the position matrix. The function *check_overlap* determines whether the specified bounds are outside, inside, or intersecting the view frustum and returns one of the enumerated values *OUTSIDE, INSIDE, INTERSECTS*. If bounding volume of the cluster is completely inside the view frustum, then all of its descendants are inside the view frustum and there is no need to perform further frustum culling checks. The function *geometric_error_to_sse* transforms the geometric error of the cluster in accordance with the formula above and returns screen-space error.

3. Out-of-core HDLOD algorithm

The HDLOD method allows for conservative out-of-core rendering based on splitting the cluster tree into a skeleton, which is mostly in main memory, and cluster contents, which are preferably in external memory and loaded into main memory on the fly when needed. From formal point of view, the cluster tree can be thought as the structure $C = C' \cup C''$, where $C' = \{c'(b_c(t), \delta_c(t), v_c(t))\} \cup \{\prec\}$, $C'' = \{c''(g_c, p_c(t))\}$. In what follows, the first structure C' is called the skeleton, which contains both the cluster headers c' and agglomeration relations $\prec$ of the original tree. The second structure C'' is referred to as the complement that contain the cluster contents c''. It is assumed that each cluster header is linked with the corresponding cluster content, regardless of where it is allocated either in external memory or already loaded into main memory. It does not matter whether the links are implemented as memory pointers, identifiers, URL references, or any other associative structures and mechanisms. The assumptions above do not prevent the decomposition of the skeleton into parts, which will be stored, loaded and processed separately. It is also allowed that simple clusters can be an integral part of the skeleton, while more complex ones are the parts belonging to the complement. The only requirement is that the skeleton consumes a small amount of main memory compared to the complement and entire representation.

It is noteworthy that the idea of extracting the skeleton is not original and has been successfully applied for out-of-core rendering of static scenes within the HLOD methods [2]. However, the presented study proves that this idea can be feasible and fruitful for rendering deterministic dynamic scenes as well. Moreover, we argue that the presented in Figure 1 pseudocode for displaying HDLOD trees in main memory is suitable for conservative rendering of these structures stored in external memory. The only reasonable assumption is that the skeleton has been preliminary allocated in the main memory.

Unlike interactive rendering, which focuses on viewing and manipulating scenes in real time, usually at the expense of image trustworthiness and accuracy, conservative rendering provides the required fidelity of the generated images, but with lower frame rates and less stringent performance requirements. The main reason is the inevitable costs of additional operations for loading and unloading the contents of the clusters. However, conservative rendering is important for industrial applications that face the challenges of accurate scene analysis, collision and interference detection, and proper visualization of the detected issues. It is also necessary for applications that generate photorealistic graphics and video-presentations. Conservative rendering must quickly analyze the whole HDLOD tree and select clusters that will provide the desired level of fidelity, that is why the skeleton is required to be in main memory.

The out-of-core algorithm employs the pseudocode shown in Figure 1, except for the *display_cluster* function. Instead of immediately rendering the cluster the out-of-core algorithm must first ensure that the cluster contents (geometry and position) are loaded into main memory. To efficiently implement this without stalling the main thread, two additional threads are used. The threads communicate via queues (Figure 2). The main thread traverses HDLOD tree and makes decisions on what clusters have to be rendered, similarly to in-core HDLOD rendering algorithm. The checks against bounding volumes, tolerances and presence statuses can be done immediately since they access attributes of the skeleton which are always in main memory under the assumption above. If the cluster is selected for rendering, it is added to the load queue. Load thread monitors load queue and performs loading of requested cluster contents into main memory. If the cluster is loaded, load thread sends the command for rendering it via render queue. Rendering thread monitors render queue and performs all necessary API calls to set state, load cluster mesh into video memory and run its rendering on GPU.

To minimize number of loading operations, load thread maintains a cache in main memory. It keeps track of the loaded clusters and stores their contents in main memory. A memory limit is given to the cache manager to ensure that no more memory than specified is used. The only restriction to this limit is that the skeleton and the rendering buffers have to fit into memory too. When the limit is reached, some clusters have to be removed from cache in order to allow for loading of the new clusters. To decide on which clusters to remove, the least recently used (LRU) replacement policy is used. The cache manager maintains the list of loaded clusters sorted by the time they were last accessed. Each time a cluster is selected for rendering it is placed in the head of the list. This way, the tail of the list contains the least recently used cluster that is the first candidate for removal. The LRU policy is commonly used in out-of-core rendering algorithms to determine what data to release [2][3]. However, implementers must take care to not release clusters that are in use by the rendering thread. For this purpose, a special Boolean flag, indicating that the cluster is in use, can be employed. Another solution is to use smart pointers to ensure that the data is not freed from memory prematurely.

Often out-of-core algorithms employ some sort of prefetching technique that is aimed at speculative preliminary loading of those clusters that are likely to be needed in the next frames. However, the experiments done have found situations where prefetching can lead to significant degradation of the total performance, thereby offsetting minor acceleration effects. For example, such situations happen if the memory cache is completely consumed for rendering clusters on the current frame and there is no extra space to preload clusters that presumably will be needed in the next frames. Forced prefetching of new clusters will inevitably require unloading of previously used clusters, including those that may be required again. Accuracy requirements can also impact the reasonable balance of memory cache sizes required for the current frame and next frames. Finally, effectiveness of prefetching highly depends on the quality of prediction of the camera movement, changes of modelling time and other user actions. Some specific scenarios (for example, animation playthrough) allow for accurate predictions, other scenarios (for example, navigation after a long stay) are poorly predicted and will most likely lead to additional unproductive costs for loading unnecessary clusters. In order not to lose in performance, the prefetching technique must take into account all the mentioned factors, which is a serious problem. Since the technique cannot be straightforwardly used and widely recommended, we did not use it in the experiments and focused on the fundamental possibility of using the HDLOD method for conservative out-of-core rendering of large dynamic scenes which is the main topics of the presented research.

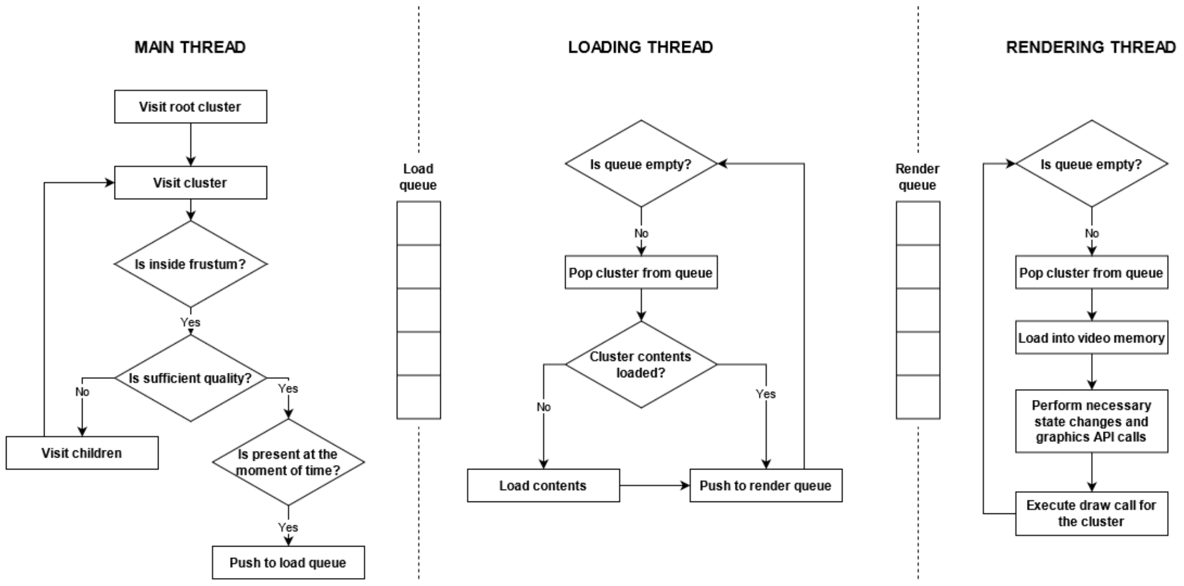


Figure 2: Overview of the out-of-core HDLOD rendering algorithm

4. Computational experiments

To validate the proposed method a series of computational experiments has been carried out. As a benchmark, synthetic scene has been generated using method described in [10]. The scene consists of 9 buildings. Each building is made of 30 by 30 bricks and has 30 floors. In total, the scene contains 1078920 triangles and 89910 meshes. HDLOD has been generated for the scene and stored in files on the data storage device. The computational experiments have been organized in such a way that the camera has been moving through the scene over 500 frames and the frame times have been measured. At the start of the trajectory, the camera is far away from the scene. For the first 100 frames it moves closer the scene and ends up at such a distance that the scene is still inside view frustum. For the next 100 frames camera rotates 90 degrees to the right around the center of the scene. Then the camera moves close to the central building through one of the buildings, some buildings are culled by the view frustum. After that the camera again rotates around the central building with some buildings appearing and some disappearing from the view frustum. In the last 100 frames the camera rapidly moves far away from the scene through one of the buildings. Experiments have been carried out for static and dynamic scenarios. Dynamic behavior has been generated for the objects of the static scene emulating simultaneous construction of the buildings. Screenshots taken during the experiment on dynamic scene are presented in Figure 3, clusters are visualized with different colors. Clusters are selected for rendering based on screen-space error, which is calculated from their geometric error. To ensure high fidelity of conservative rendering only clusters whose screen-space error is no more than the size of the pixel are displayed. This way, the difference between original model rendering and rendering with HDLODs is barely visible. The rendering performance has been measured on a typical hardware configuration: Intel Core i7-4790 CPU (3.6 GHz), 16 GB of RAM, GeForce GTX 750 Ti (2 GB), Seagate ST2000DM001 HDD (2 TB, 7200 RPM).

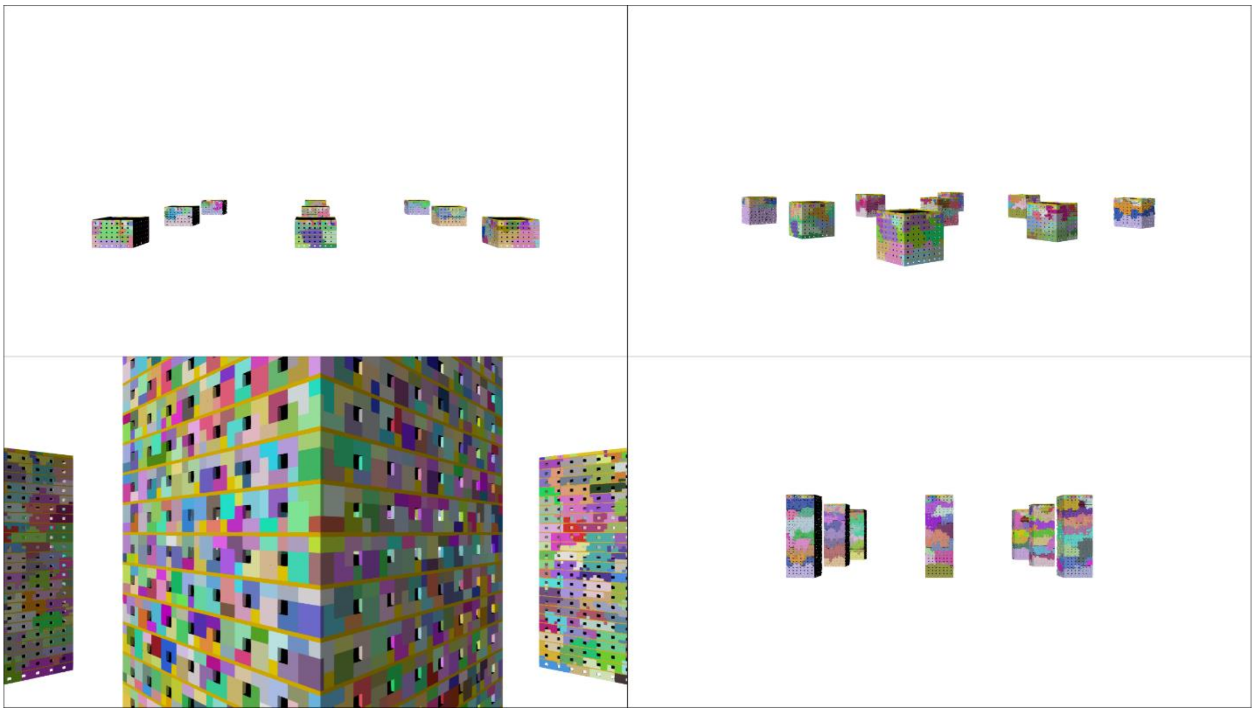


Figure 3: Screenshots taken during experiment on dynamic scene. The buildings grow while the camera is flying through the scene. Clusters are visualized with different colors

It is worth noting that data storage device performance affects cluster contents loading time. To validate the proposed method of out-of-core rendering an HDD storage has been selected as a worst-case scenario. SSDs tend to have much better performance. There is also an important factor that can affect the rendering performance. To render geometry on a GPU it must first be buffered in video memory. Most optimal way is to use some sort of caching for geometry in video memory, in order to minimize loading operations (i.e., video buffer with paging and LRU policy for unloading). Since this paper is focused on loading cluster contents into main memory, to eliminate the video memory buffering factor from the experiments, loading cluster geometries into video memory was performed on the fly before rendering them. This increases frame times but also helps emphasize simplification culling effectiveness.

The resulting frame time graphs for static and dynamic scenarios are presented in Figure 4. “In-core no HDLOD” graph illustrates the performance when rendering only original scene objects (leaf HDLOD clusters), which are residing in main memory. In this case the frame times are stable except for the part when frustum culling comes in play. The “in-core HDLOD” case is HDLOD rendering with all cluster contents preliminary loaded into main memory. It produces much lower frame times than “no HDLOD” due to simplification culling. The “Out-of-core HDLOD infinite cache” test starts with an empty cache; the cluster contents are loaded on the go as they get selected for rendering. The results are comparable with in-core test, except for the frame time spikes which are caused by loading large numbers of new clusters. This is especially noticeable when the camera moves through buildings (frames 250-280 and 400-410), since moving through the buildings requires loading large numbers of leaf clusters. In the dynamic scenario, not only the camera has been moving through the scene, but also modelling time has been changing. The test starts with an empty scene. “In-core no HDLOD” graph shows gradual increase in frame time (except for the part with frustum culling in frames 200-400) which corresponds to growth of the buildings and increase of triangle and mesh count with time. It also shows steps in appearing of new building elements. “Out-of-core HDLOD infinite cache” shows large periodic spikes. They are connected with dynamic behavior, because new clusters are selected for rendering due to changing geometric error functions of the clusters.

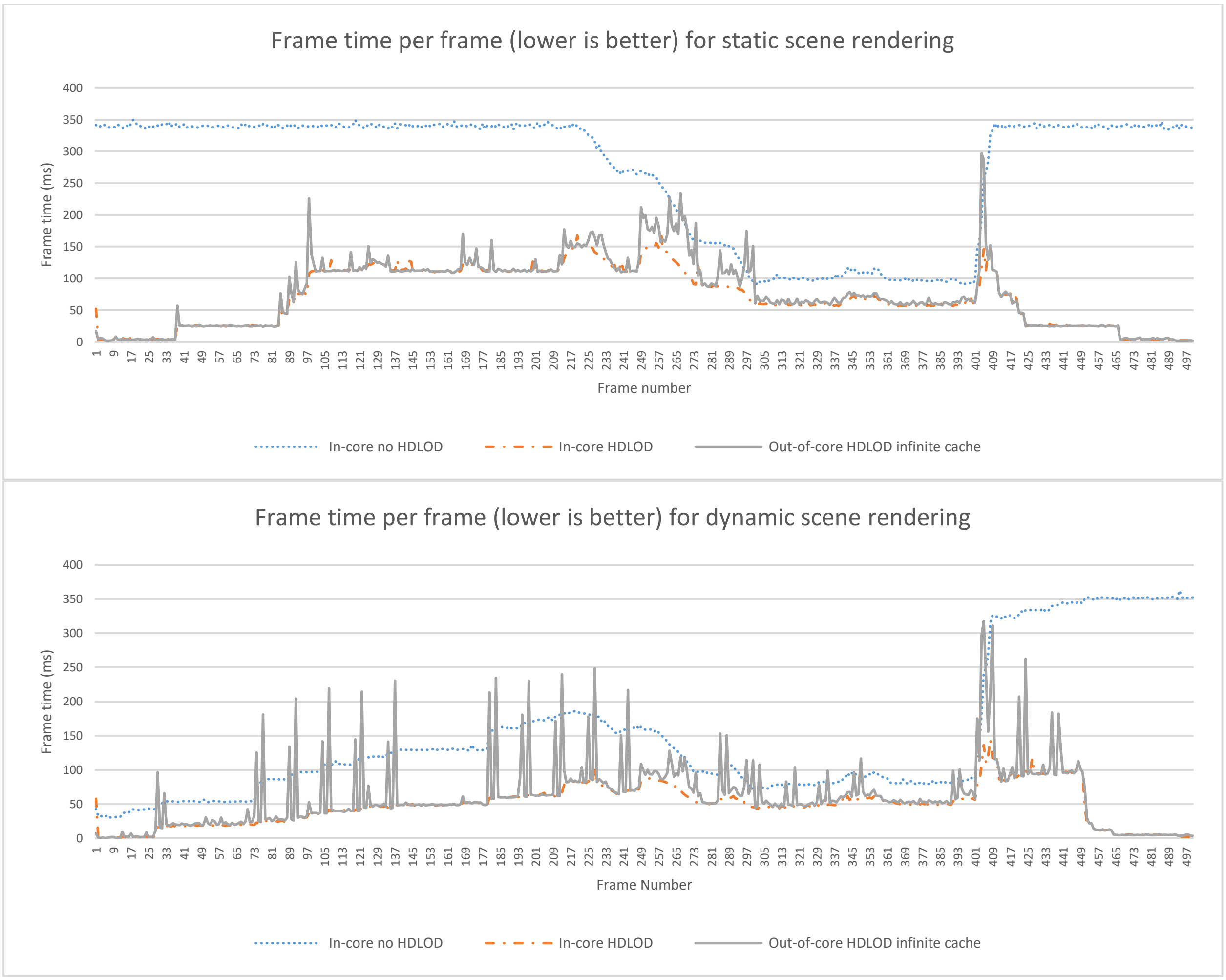
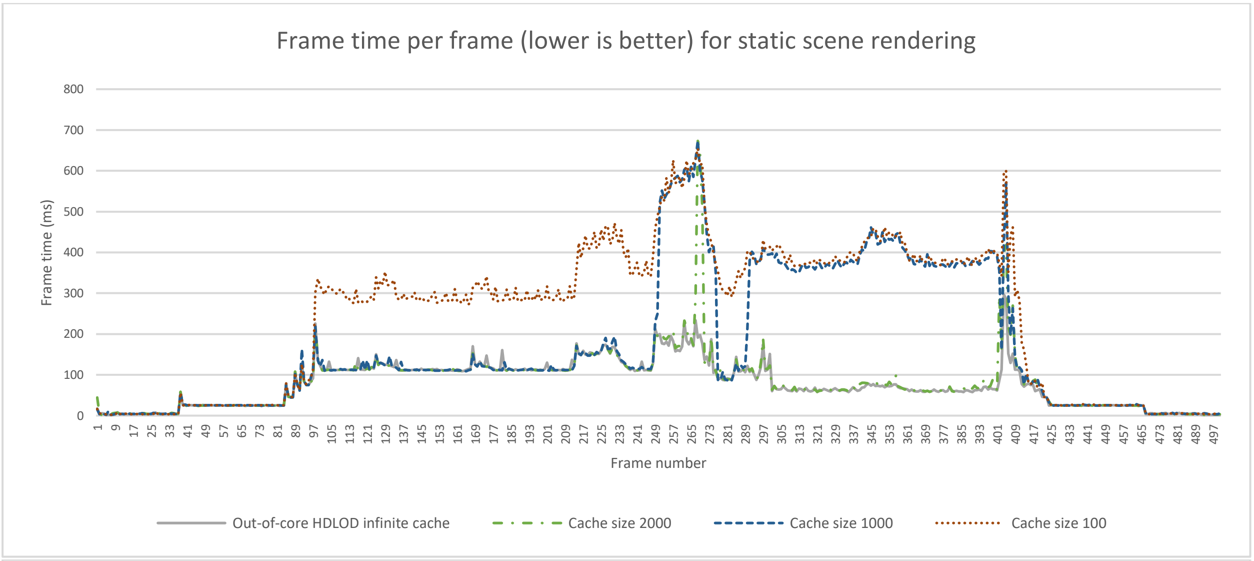


Figure 4: Performance comparison of in-core rendering without HDLOD, in-core HDLOD rendering and out-of-core HDLOD rendering with infinite cache for static and dynamic scenarios. In-core performance without HDLOD is the worst. Out-of-core performance with infinite cache size matches in-core performance with the exception of frame time spikes that occur when new clusters are loaded

In “infinite cache” test the cluster contents never get unloaded. The series of experiments with different cache sizes gives an idea of how cache size affects performance (Figure 5). For simplicity, cache size represents the number of clusters that can fit into cache. Overall, the results are comparable with infinite cache, the significant difference is observed only on very low cache sizes, when the size is too small to fit all clusters required for the frame. In such case cluster contents are unloaded after they are rendered to allow for loading next cluster contents, which causes permanent loading/unloading. For example, in the static scenario with the size of the cache for 1000 clusters it happens during frames 250-280 (moving through one of the buildings), frames 290-400 (closely rotating around the central building) and frames around 410 (moving through one of the buildings as camera rapidly flies away from the scene). With the cache size of 100 number of rendered clusters per frame exceeds the cache size most of the time, which results in significantly degraded performance, except for the beginning and the end of the experiment, where the camera is far away from the scene which causes small number of simplified clusters to be rendered. In the dynamic scenario there are periods when no dynamic changes occur. They correspond to the lower frame time periods with cache sizes 100 and 1000. This happens because at the moments of modelling time when no changes occur more simplified clusters are selected for rendering (their geometric error functions take lower values for these moments of modelling time), thus reducing number of clusters required per frame. Results with cache size larger than 5000 are practically indistinguishable from “infinite cache” and are not included in the graphs. This means that the LRU replacement policy allows to efficiently remove old clusters that are no longer needed.

With low cache sizes the out-of-core HDLOD performance is worse than in-core no HDLOD performance, but these are extreme scenarios. Typically, the cache size should be selected as large as possible, to occupy most of main memory. Moreover, "no HDLOD" test has contents of 89910 clusters residing in main memory, which is much more than 100 or 1000 clusters.



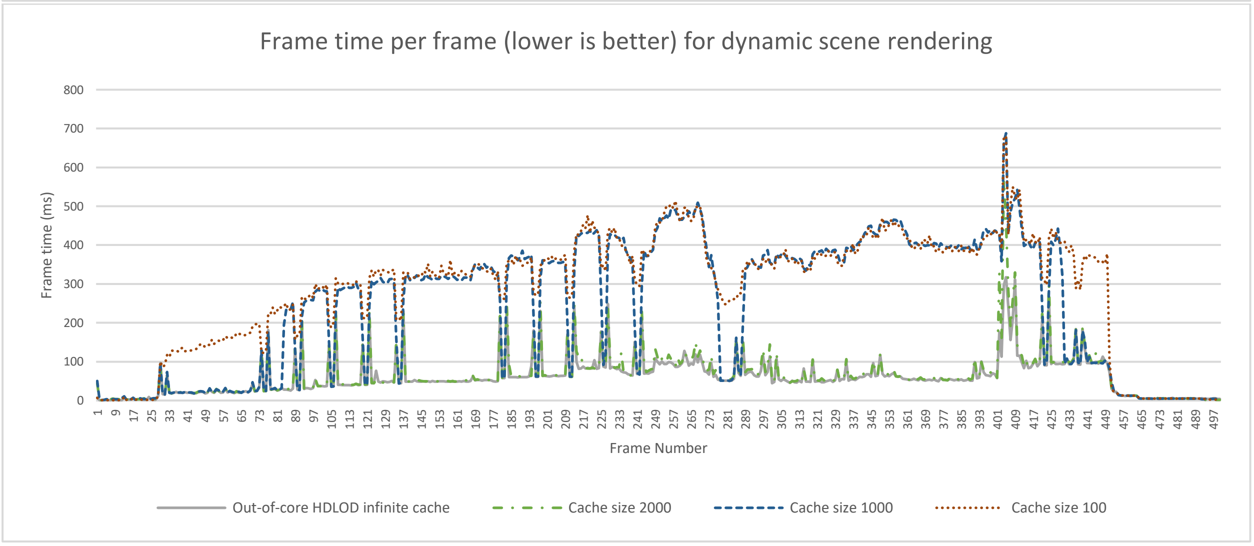


Figure 5: HDLOD out-of-core performance for low cache sizes compared to infinite cache for static and dynamic scenarios. Larger frame times occur when size of the cache is too small to fit all the clusters required to conservatively render the frame, which causes constant loading/unloading during frame rendering. Other than that, frame times are comparable with infinite cache which is due to effectiveness of LRU replacement policy

5. Conclusions

Thus, the carried-out studies have proven the high efficiency of the HDLOD method when applied for rendering of large dynamic scenes with deterministic events. The method is suitable for scenes allocated both in main memory and in external memory.

The out-of-core version of the method is based on splitting the preliminary formed HDLOD cluster tree into a skeleton, which is mostly in main memory, and cluster contents, which are preferably in external memory and loaded into main memory when needed. This is not a burdensome limitation, since memory overhead is mainly caused by geometric and behavioral models, rather than by the tree structure and simple cluster attributes such as bounding volumes, time intervals and geometric errors. The research has proven the possibility for conservative out-of-core rendering of large dynamic scenes with a given level of tolerance and trustworthiness. Quite an expected result is that the larger the size of the available memory cache, the more successfully the rendering problem can be solved using the HDLOD method. The research is planned to be continued to analyze the possibilities of using the method for interactive out-of-core rendering, which is expected to alleviate the memory requirements.

6. References

[1] W. Corrêa, J. T. Klosowski, C. T. Silva, iWalk: Interactive Out-Of-Core Rendering of Large Models, 2002. URL: https://www.researchgate.net/publication/2862878_iWalk_Interactive_Out-of-Core_Rendering_of_Large_Models.

[2] G. Varadhan, D. Manocha, Out-of-core rendering of massive geometric environments, in: IEEE Visualization, VIS 2002, 2002, pp. 69–76. doi:10.1109/VISUAL.2002.1183759.

[3] P. J. Cozzi, Visibility Driven Out-of-Core HLOD Rendering, Master’s thesis, University of Pennsylvania, Philadelphia, USA, 2008.

[4] C. Erikson, D. Manocha, W. V. Baxter, HLODs for faster display of large static and dynamic environments, in: Proceedings of the 2001 symposium on Interactive 3D graphics, I3D '01, Association for Computing Machinery, New York, NY, USA, 2001, pp. 111–120. doi:10.1145/364338.364376.

[5] S.-E. Yoon, B. Salomon, R. Gayle, D. Manocha, Quick-VDR: Interactive View-Dependent Rendering of Massive Models, in: Proceedings of the conference on Visualization '04, VIS '04, IEEE Computer Society, USA, 2004, pp. 131–138. doi:10.1109/VISUAL.2004.86.

[6] I. Wald, A. Dietrich, P. Slusallek, An interactive out-of-core rendering framework for visualizing massively complex models, in: ACM SIGGRAPH 2005 Courses, SIGGRAPH '05, Association for Computing Machinery, New York, NY, USA, 2005, pp. 17–es. doi:10.1145/1198555.1198756.

[7] D. Aliaga, J. Cohen, A. Wilson, E. Baker, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stuerzlinger, R. Bastos, M. Whitton, F. Brooks, D. Manocha, MMR: an interactive massive model rendering system using geometric and image-based acceleration, in: Proceedings of the 1999 symposium on Interactive 3D graphics, I3D '99, Association for Computing Machinery, New York, NY, USA, 1999, pp. 199–206. doi:10.1145/300523.300554.

[8] Cesium 3D Tiles, Specification for streaming massive heterogeneous 3D geospatial datasets. URL: https://github.com/CesiumGS/3d-tiles.

[9] V. Semenov, V. Shutkin, V. Zolotov, S. Morozov, V. Gonakhchyan, Visualization of Large Scenes with Deterministic Dynamics, Programming and Computer Software, vol. 46(3) (2020), pp. 223–232. doi:10.1134/S036176882003007X.

[10] V. Gonakhchyan, O. Tarlapan, V. Semenov, Generating Dynamic 3D Scenes for Rendering Benchmarks, in: Proceedings of the International Conference on Computer Graphics, Visualization, Computer Vision and Image Processing 2019, CGVCVIP 2019, 2019, pp. 485–488. doi:10.33965/cgv2019_201906P076.