

# Integrating Occlusion Culling and Hardware Instancing for Efficient Real-Time Rendering of Building Information Models

Mikael Johansson

*Chalmers University of Technology, Gothenburg, Sweden*

*jomi@chalmers.se*

Keywords: Real time rendering, Occlusion culling, Hardware instancing, BIM

**Abstract:** This paper presents an efficient approach for integrating occlusion culling and hardware instancing. The work is primarily targeted at Building Information Models (BIM), which typically share characteristics addressed by these two acceleration techniques separately – high level of occlusion and frequent reuse of building components. Together, these two acceleration techniques complement each other and allows large and complex BIMs to be rendered in real-time. Specifically, the proposed method takes advantage of temporal coherence and uses a lightweight data transfer strategy to provide an efficient hardware instancing implementation. Compared to only using occlusion culling, additional speedups of 1.25x-1.7x is achieved for rendering large BIMs received from real-world projects. These speedups are measured in viewpoints that represents the worst case scenarios in terms of rendering performance when only occlusion culling is utilized.

## 1 INTRODUCTION

With the creation of Building Information Models (BIM), the content produced by architects and designers has evolved from traditional 2D-drawings to semantically-rich, object-oriented 3D-models. With all of the data available in 3D, this concept further facilitates the use of real-time visualizations in various contexts. However, as primarily created to describe a complete building in detail, many 3D datasets extracted from BIMs provides a challenge to manage in real-time without additional acceleration techniques (Steel et al., 2012).

In this context, occlusion culling has been shown to provide a suitable option (Johansson and Roupé, 2012). Given that typical building models naturally exhibit a lot of occlusion this is an efficient approach to increase rendering performance for many viewpoints. Still, for viewpoints where many objects are, in fact, visible, occlusion culling alone may not always be able to provide sufficiently high frame rates. Common examples include exterior views of whole building facades where the sheer number of draw calls, and hence CPU burden, easily becomes the limiting factor in terms of rendering performance (Wloka, 2003).

Another characteristic of typical BIMs is the frequent reuse of identical building components. As similarity tends to reduce design, production and maintenance costs, use of multiple identical components, such as doors and windows, is common in any building (Sacks et al., 2004). For viewpoints where many objects are visible it is therefore a high probability that many of these objects are identical, albeit placed at different locations. One way to take advantage of this is to utilize the hardware instancing functionality of modern GPUs. With hardware instancing it is possible to render multiple copies of the same geometry with a single draw call, thereby reducing CPU-burden for scenes with much repetition. However, even if the reduction of draw calls improves performance in CPU-limited scenarios, the GPU still has to process all the instantiated geometry. As such, culling of invisible instances is still important to reduce overall workload

This paper presents a method to integrate occlusion culling and hardware instancing in order to provide efficient real-time rendering of large BIMs. By using an efficient occlusion culling algorithm hardware instancing can be restricted to visible replicated objects only. For viewpoints when many objects are visible, hardware instancing complements the occlusion culling by providing an

efficient rendering path for visible replicated objects. The key component to realize this is an efficient dynamic hardware instancing implementation that takes advantage of temporal coherence and uses a lightweight data transfer strategy.

## 2 RELATED WORK

### 2.1 Occlusion Culling

With occlusion culling the aim is to identify and reject occluded regions of 3D scenes in order to improve rendering performance. Within this category of acceleration techniques a vast amount of research has been conducted and for a general overview interested readers are referred to the surveys provided by (Cohen-Or et al, 2003) and (Bittner and Wonka, 2003). In essence, available algorithms can be classified according to whether they require time-consuming offline computations or not.

When considering online approaches, which require no pre-computations, the support for hardware accelerated occlusion queries has provided a simple mechanism to detect visibility. With hardware occlusion queries the GPU returns the number of pixels that passes the depth test when rasterizing a given object. This way, proxy geometries can be used to detect occlusion before the actual object is rendered. However, due to the delayed processing in the graphics pipeline the result of the query is not immediately available on the CPU which makes an efficient implementation more complex. This problem was addressed with the Coherent Hierarchical Culling (CHC) algorithm, which exploits spatial and temporal coherence in order to reduce latency and overhead of the queries (Bittner et al., 2004). However, although the CHC algorithm works well in highly occluded scenes, wasted queries and unnecessary state changes makes it less reliable for viewpoints when many objects are visible. In order to reduce the number of wasted queries, (Guthe et al., 2006) proposed a method, called Near Optimal Hierarchical Culling (NOHC), based on a statistical model for occlusion probability and a hardware calibration step. Assuming proper hardware calibration their approach always performs better than view-frustum culling. In (Mattausch et al., 2008) an improved version of the CHC algorithm, called CHC++ was presented. Although the core ideas of the algorithm remain the same, the additional components introduced by CHC++ provide a significant improvement in rendering

speed compared to both NOHC and CHC. Mainly, this was achieved by introducing batching of queries as a means to reduce costly state changes.

When considering the case of rendering complex BIMs, the efficiency of the CHC++ algorithm has been recently demonstrated (Johansson and Roupé, 2012). Compared to view-frustum culling, CHC++ provided significant speedups for a number of fairly large BIMs during both interior and exterior viewpoints.

As an alternative to occlusion queries, (Hill and Collin, 2011) recently proposed a modern variant of the hierarchical z-buffer (Green et al., 1993), where all visibility tests are performed on the GPU. The state of visibility is then read-back to the CPU, so that un-occluded objects can be rendered in a single stage. However, although reported as being successfully used in recent computer games, the performance implications are still largely unknown for general 3D models. In addition, this approach requires a set of good occluders in order to initiate the z-buffer.

A problem common to practically all visibility culling methods is that of granularity. On the one hand, in order to maximize culling efficiency, we ideally want to perform visibility determination on the level of granularity provided by the individual objects contained in a 3D scene. On the other hand, for viewpoints with many visible objects, this is not an optimal organization of the 3D scene, considering the aim of keeping a low draw call count (Wloka, 2003). In this case, reduction of draw calls can often be addressed by geometry batching, where spatially coherent objects (with similar material properties) are combined into larger ones during a pre-process (Buchholz and Döllner, 2005). However, even if this process enhance rendering performance for certain viewpoints, it potentially reduces culling efficiency, and hence, performance, for other viewpoints. Besides requiring a dedicated pre-process, geometry batching also complicates the use of additional acceleration techniques applied per-object, such as level-of-detail (LOD).

The proposed method addresses this situation by performing *implicit* geometry batching. By taking advantage of hardware instancing capabilities of modern GPUs, culling can be performed at fine granularity at the same time as the amount of draw calls is reduced for viewpoints with many visible objects.

## 2.2 Hardware Instancing

For 3D scenes where many individual objects have to be rendered it is not uncommon that the large number of draw calls (and related state changes and buffer binds) becomes the limiting factor in terms of performance (Wloka, 2003). Given a large amount of replicated geometry, hardware instancing is one way to address this problem. The idea behind this concept is to use a single draw call when rendering multiple copies of the same geometry. By using a previously uploaded buffer or texture object containing per-instance data (i.e. transformation matrix), each instance can then be transformed to its correct location on the GPU. Typical applications that can benefit from hardware instancing include rendering of crowds and vegetation, which usually require a large number of instances at the same time as there exists much repetition. In (Park et al., 2009), (Dudash, 2007), and later (Ramos et al., 2012), examples on how to render several thousands of animated characters in real-time with the use of hardware instancing is presented. Recently, (Bao et al., 2012) presented a GPU-driven framework for rendering large forests. Hardware instancing, together with level-of-detail selection on the GPU, allow them to render several thousands of detailed trees, with shadows, in real-time.

However, even if hardware instancing reduces the number of draw calls, and hence CPU-burden, the GPU still have to process all the geometry that is instantiated. Without any type of visibility culling, this may lead to unnecessary high GPU-burden for 3D scenes with many instances. In order to limit the number of instances, (Park et al., 2009) and (Bao et al., 2012) perform view-frustum culling on the GPU.

Still, for highly occluded scenes, such as Building Information Models, view-frustum culling only allows a subset of the invisible geometry to be rejected. The proposed method addresses this problem by an efficient dynamic hardware instancing implementation. By taking advantage of temporal coherence together with a lightweight data transfer approach, occlusion culling can be performed at object level at the same time as replicated geometry is efficiently rendered using hardware instancing.

## 3 THE IFC BUILDING MODEL

For the majority of BIM authoring tools the underlying data-model closely resembles that of the Industry Foundation Classes (Eastman et al., 2011).

Instead of pure geometrical entities, this scheme represents a building or facility in terms of its individual building components, such as walls, doors, windows and floors. For each component a visual representation is then provided in the form of one or several geometrical entities (i.e. triangular meshes). When considering instancing, this concept is performed at the building component level. As an example, all instances of a specific window type will be considered a unique building component but share the same visual representation. For the implementation and tests presented in this paper, no additional processing of the input 3D-data has been performed except organizing it in a bounding volume hierarchy. In this hierarchy, leaf nodes represent the individual building components. As such, culling is performed at a granularity corresponding to the individual building components. However, hardware instancing is performed at a level corresponding to the geometrical entities that represent each component.

For the rest of this paper replicated components that are suitable for hardware instancing are referred to as *instanceable*. The specific geometry being instanced is referred to as the *geometry reference*.

## 4 ALGORITHM OUTLINE

The proposed method consists of three main steps. In order to give an overview of the algorithm all three steps are briefly discussed below.

**Determine visible instances** Using an efficient occlusion culling system, we inherently have access to the set of potentially visible objects in a certain frame. Based on the assumption that hardware instancing is the most efficient way to render multiple copies of the same geometry, this set is searched for replicated components. These objects are then scheduled for rendering with hardware instancing in the *next* frame.

**Upload required data to GPU** Given a set of visible objects to be rendered using hardware instancing, per-instance data need to be uploaded to the GPU. To reduce per-frame data transfer, an indexed approach is used: During scene loading, transformation matrices for all potential instances are uploaded to the GPU. During rendering, only a single index per instance needs to be transferred in order to locate the corresponding transformation matrix in GPU memory. Thus, at the end of each frame, data in the form of indices is uploaded to the GPU for processing during the next frame.

**Render using hardware instancing** At the beginning of each frame opaque instances collected during the previous frame are rendered using hardware instancing. However, for semi-transparent geometry hardware instancing introduces complexities. As correct depth ordering no longer can be maintained, an order-independent transparency rendering technique (Everitt, 2001) is needed to support hardware instancing of semi-transparent geometry.

## 5 INTEGRATING OCCLUSION CULLING AND HARDWARE INSTANCING

As outlined, the general idea behind the proposed method is to dynamically select candidates for hardware instancing, based on visibility knowledge provided by the occlusion culling system. For the purpose of this, any efficient occlusion culling algorithm can be used as long as it provides visibility classification for all objects in a scene. For the implementation and tests presented in this paper the latest version of the Coherent Hierarchical Culling algorithm, CHC++ has been used. This choice is based on the simplicity of the algorithm and the fact that it has already been proven to work well when applied to large Building Information Models. As such, it provides a good basis for further enhancements. In the followings subsections the details of the algorithm is presented, starting with a review of the hardware instancing API together with important aspects of the CHC++ algorithm.

### 5.1 Hardware Instancing API

Taking OpenGL as an example, the instancing API extends the conventional draw call by exposing the option to specify the number of times a particular batch of geometry should be rendered. In the vertex shader an internal counter (`gl_InstanceID`) is then accessible which advances for each iteration. Using the internal counter as an index, the per-instance transformation matrix can then be sourced from any type of previously uploaded array, texture or buffer object. However, the arrangement of per-instance data must reflect the fact that the internal counter advances with a fixed step. In order to render a specific set of instances with a single draw call, the per-instance data must be arranged sequentially.

### 5.2 CHC++

The original CHC algorithm takes advantage of spatial and temporal coherence in order to provide efficient scheduling of hardware occlusion queries. The state of visibility from the previous frame is used to initiate queries in the current frame (temporal coherence) and by organizing the scene in a hierarchical structure (i.e. bounding volume hierarchy) it is possible to test entire branches of the scene with a single query (spatial coherence). While traversing a scene in a front-to-back order, queries are only issued for previously invisible interior nodes and for previously visible leaf nodes of the hierarchy. The state of visibility for previously visible leaves is only updated for the next frame and they are therefore rendered immediately (without waiting for the query results to return). The state of visibility for previously invisible interior nodes is important for the current frame and they are not further traversed until the query results return. By interleaving the rendering of (previously) visible objects with the issuing of queries, the algorithm reduces idle time due to waiting for queries to return.

Although the core ideas remain the same, CHC++ introduced several optimizations which make it perform very well even in situations with low occlusion. Most notably, the improved version addressed the problem of redundant state changes due to the interleaved rendering and querying. Instead of directly querying a node, it is appended to a queue. When this queue reaches a certain size, the rendering state is changed to querying and an occlusion query is issued for each node in the queue. In addition, this mechanism allows an application to perform material sorting before rendering visible objects in order to reduce costly API calls.

In order to reduce the number of queries, the original CHC algorithm introduced an important optimization based on temporal coherence - A visible object is assumed to stay visible and will only be tested for visibility again after a user-specified amount of frames (typically 10-20). This optimization, together with the assumption that hardware instancing is the most efficient way to render multiple copies of the same geometry, is the entry-point for the proposed method. When an object suitable for instancing is found visible, it is scheduled to be rendered using hardware instancing in the following frame.

### 5.3 Data Preparation

In a static situation, where the same set of instances should be rendered every frame, per-instance data can be uploaded to GPU-memory once, and then, during subsequent draw calls, be fetched in the vertex shader based on the value of the internal instance counter. In the proposed approach, however, a dynamic behaviour is needed in order to support per-frame selection of which geometry to render using hardware instancing. As per-instance data needs to be arranged sequentially in order to facilitate a single draw call per geometry reference, this requires complete or partial updates of the shared buffer or texture every frame. In order to optimize this process an indexed approach is used, as explained visually in Figure 1. During scene loading, transformation matrices for all instanceable objects are collected and placed in a single array, denoted M. The location of each instance's transformation matrix within this array is recorded for later use. During rendering, a single index is then needed to locate each instance's transformation matrix within M. For a 4x4 transformation matrix, this approach effectively reduces the required data transfer by a factor of 16. Both the index array (I) and matrix array (M) are implemented as a Texture Buffer Objects.

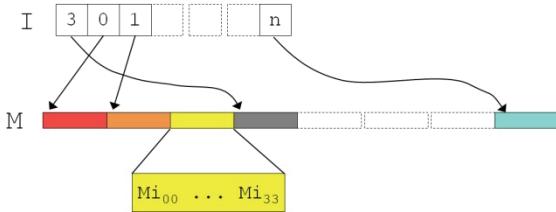


Figure 1: An array of indices (I) is used to locate each instance's transformation matrix, encoded in a single, shared array (M).

### 5.4 Collecting Visible Instances

Depending on the occlusion culling algorithm of use, the state of visibility for objects in a scene might be known at different stages. For instance, the GPU-based implementation of the Hierarchical Z Buffer proposed by (Hill, 2011), resolves visibility for all objects in a single phase in the beginning of a frame. The CHC++ algorithm, on the other hand, distributes this process by interleaving the rendering of objects with the issuing of queries, effectively delaying the complete visibility knowledge of a scene towards the end of the frame. In order to cope with different implementations and to provide additional time for the required data transfer, the

rendering of instanceable geometry is deferred by one frame. Thus, an object detected visible in frame n will be scheduled for rendering using hardware instancing in frame n+1. Figure 2 presents the modifications to the original CHC++ algorithm that is needed in order to implement this behavior. When a node of the spatial hierarchy is found visible, the *TraverseNode* function is called for its children (For a complete picture of the algorithm the reader is referred to the original CHC++ paper). During traversal of an instanceable leaf node the algorithm first checks if it is scheduled for rendering using hardware instancing in the current frame. If this is not the case it is rendered in a conventional way by adding it to a render queue. In a second step, it is scheduled for rendering using hardware instancing in the next frame.

```
TraverseNode(N) {
    if isLeaf(N) {
        if isInstanceable(N) {
            if N.nextInstFrameId != frameId {
                Render(N);
            }
            EnqueueForInstInNextFrame(N);
            N.nextInstFrameId = frameId + 1;
        }
        else {
            Render(N);
        }
    }
    else {
        DistanceQueue.PushChildren(N);
        N.IsVisible = false;
    }
}
```

Figure 2: Pseudo-code for the collection of visible instances. Difference to the CHC++ algorithm is marked in blue.

### 5.5 Data Transfer

At the end of frame n, a set of objects suitable for rendering using hardware instancing in frame n+1 has been collected. As illustrated in Figure 3, this set is sorted by geometry reference to generate a single array of indices (I) to upload to GPU memory. Thus, for m unique geometry references the array will contain m regions of indices. Within each region, the array is populated with indices corresponding to the location of each instance's transformation matrix in M. While generating the array the offset to each specific region is also recorded. This offset is needed during rendering in order to use a single indices array for all geometry references (Section 5.6).

During this stage, before the actual upload, the minimum number of instances per geometry

reference is also considered. Geometry rendered with hardware instancing uses a more complex vertex shader and require additional data transfer, which itself introduce a performance penalty. In order to gain an increase in performance the reduction of draw calls must reflect this. If the number of collected instances per geometry reference is below a user-defined parameter,  $N_{\text{Imin}}$ , they are not scheduled for instancing in the next frame. Instead the parameter  $\text{nextInstFrameId}$  (Figure 2) is set to zero on the corresponding objects in order to render them using a non-instanced approach in the next frame. For the different BIMs evaluated in this paper (Section 6), empirical tests have shown that a minimum requirement of three (3) instances per geometry reference is a suitable choice. However, ultimately, this parameter should be set per geometry reference, taking number of triangles into account.

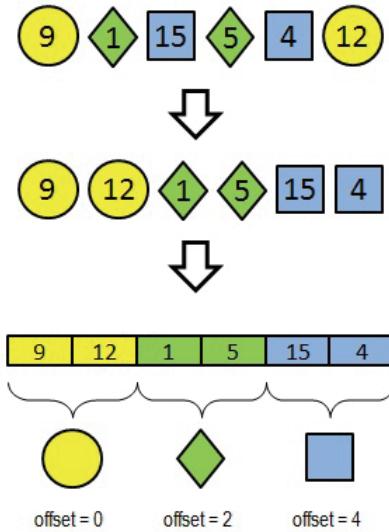


Figure 3: Collected instances (top) are sorted by geometry reference (middle) to generate the final index array and corresponding offsets (bottom). The numbers (indices) corresponds to the location of each instance's transformation matrix in M.

## 5.6 Rendering

As based on hardware occlusion queries, the CHC++ algorithm requires that visible objects are rendered before occluded ones in order to properly detect occlusion. Thus, in order to preserve the state of visibility, opaque instances are rendered using hardware instancing in a single step at the beginning of each frame. However, if the visibility determination system is separated from the

conventional rendering, this step can be performed at a later stage.

The actual rendering of all collected instances is performed by a single draw call per geometry reference. During this stage, the transformation matrix array (M) and indices array (I) are bound to the context. In Figure 4, GLSL code fragments from the vertex shader are shown. Here, the internal counter (`gl_InstanceID`) is used to fetch the current index from the indices array (I). This index is then used to locate the correct transformation matrix in the transformation matrix array (M). However, when invoking an instanced draw call, the internal counter will start its iteration from zero (0). As a single array is used for all indices an offset is required to define which region to fetch values from. This offset is recorded during the actual forming of the global indices array (Section 5.5), and during rendering it is supplied as a uniform per geometry reference.

```
uniform samplerBuffer M; //Matrices
uniform samplerBuffer I; //Indices
uniform int _offset;

void main()
{
    //Fetch index by offset
    int id = gl_InstanceID + _offset;

    int idx =
        int(texelFetchBuffer(I,id).x);

    mat4 OT =
        mat4(texelFetchBuffer(M,idx*4),
            texelFetchBuffer(M,idx*4+1),
            texelFetchBuffer(M,idx*4+2),
            texelFetchBuffer(M,idx*4+3));

    gl_Position =
        gl_ModelViewProjectionMatrix *
        OT * gl_Vertex;

    //-----
    //Other per-vertex calculations.
    //-----
}

;
```

Figure 4: Vertex shader used for the instanced rendering path (GLSL-code).

### 5.6.1 Semi-Transparent Geometry

Using conventional methods, semi-transparent geometry is rendered after opaque objects, in a back-to-front order, using alpha blending (Akenine-Möller et al., 2008). With hardware instancing correct order among transparent objects can no



Figure 5: Transparency rendering modes: Depth peeling (left), Weighted average (middle) and sorted by object (right).

longer be preserved and, consequently, an order-independent transparency rendering technique is needed. A common technique within this category of algorithms is depth peeling (Bavoil, 2008), where transparent fragments are sorted by rendering the geometry several times, peeling off one transparent layer at a time. However, although accurate, the performance penalty of depth peeling is rather high which makes it unsuitable in practice. As a performance efficient alternative, a single pass approximation of depth peeling is suggested in (Bavoil, 2008). The technique, referred to as weighted average transparency, calculates the final color as the alpha-weighted sum of colors divided by the average alpha. When blending pixels of equal color and transparency, this technique produces correct results. However, when colors and transparency values differ too much, the result of the weighted average technique starts to deviate in terms of correctness compared to depth peeling. Still, despite being an approximation, it works very well for typical building models. As the use of transparency and color for windows and glazed structures is usually coherent within a building, the technique produces plausible results. Even in situations when many transparent layers are visible, the difference between the correct and the approximate method is hard to detect, as seen in Figure 5.

In the proposed method, both instanced and non-instanced semi-transparent geometry are rendered in a final stage each frame. For the implementation and tests the weighted average transparency technique has been primarily used. However, in the results section, the findings in terms of performance for the depth peeling approach are also reported.

## 6 RESULTS

The proposed method has been tested on four different Building Information Models. The models were created in Autodesk Revit 2012, and all four represents planned or existing buildings (see Table 2

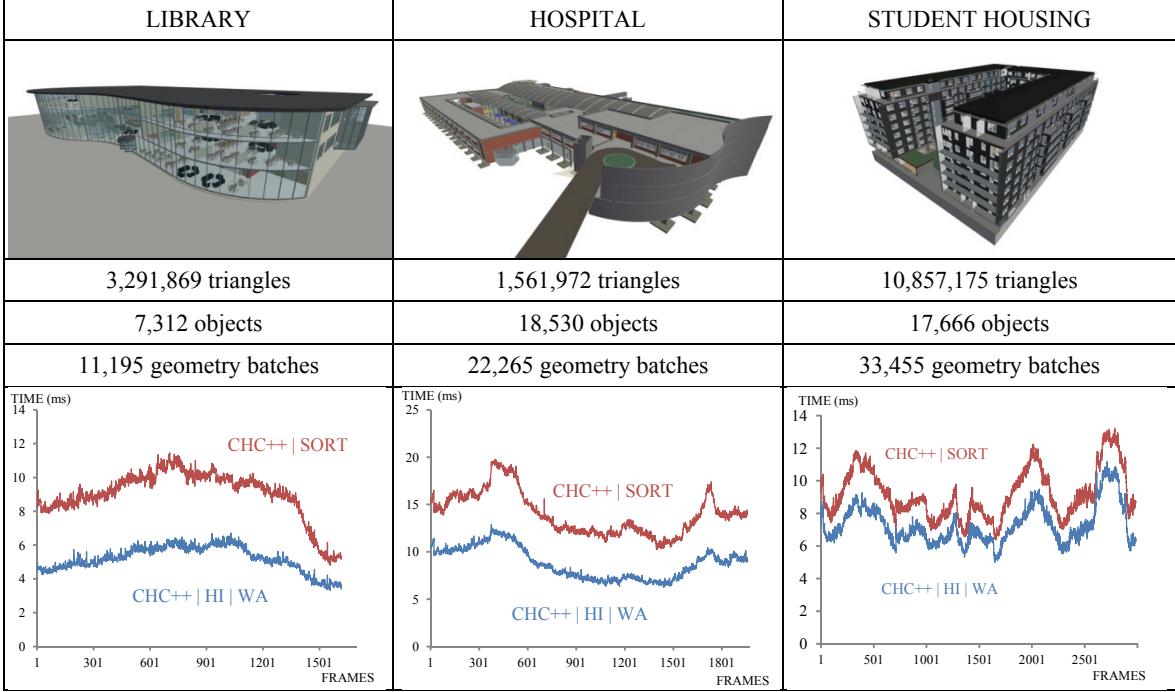
Table 1: Frame times (in ms) for view-frustum culling (VFC) and occlusion culling (CHC++) for one exterior and one interior viewpoint for each of the four test models.

Scene	INTERIOR		EXTERIOR	
	VFC	CHC++	VFC	CHC++
Library	13.1	1.9	17.2	7.5
Hospital	26.6	1.1	37.8	18.8
Student Housing	40.1	1.2	68.2	11.3
Hotel	56.3	1.4	130.2	47.8

and 3 for detailed information). For all of the tests an Intel Core i7 3.07 GHz CPU and an Nvidia GeForce 570 GTX graphics card was used. The CHC++ occlusion culling algorithm was used together with a bounding volume hierarchy built according to the surface area heuristics (Macdonald, 1990), and the screen resolution was set to 1280 x 720. Unless otherwise stated, the following parameters were used: maximum triangle count for instancing Tmax=3000, assumed visible frames Nav=20, minimum number of instances NImin=3. The weighted average technique (WA) was used for rendering semi-transparent geometry when hardware instancing (HI) was activated. Without instancing activated semi-transparent geometry was rendered using a conventional sort-by-object approach (SORT). However, for the Hotel model the performance numbers with depth peeling (DP) is also presented.

The CHC++ algorithm has previously been found to perform very well compared to only using view frustum culling for typical BIMs. These findings were confirmed for all of the test-models. Table 1 presents a comparison of frame times for one interior (highly occluded) and one exterior (same as seen in the screenshots in Table 2 and 3) viewpoint for each of the four models. As can be seen, the CHC++ algorithm provides a significant speedup, especially for the interior viewpoints. Given this, subsequent tests were focused on the worst case scenarios provided by the test models -

Table 2: Statistics for three of the test models and frame times for the proposed approach with occlusion culling (CHC++), hardware instancing (HI) and weighted average transparency (WA) compared to occlusion culling (CHC++) and conventional sort-by-object transparency (SORT) for the predefined walkthroughs.



viewpoints when many objects are visible. In such situations the sheer number of objects that has to be rendered becomes the limiting factor in terms of performance. For all test models these scenarios were found in exterior viewpoints and a set of camera paths were constructed accordingly. These walkthroughs represent the worst case scenarios in terms of rendering performance when only occlusion culling was enabled. Table 2 (left) presents the frame times with and without hardware instancing enabled for the Library model. This model features a large glazing façade and consequently, many interior objects are visible at the same time when viewed from the outside. Here, the camera path provides an orbital camera movement from one side of the building to the other, while facing the center of the building. Compared to only using occlusion culling, the proposed method provides an average speed-up of 1.7x during this walkthrough. For the Hotel model the results are similar (Table 3). However, in this case the number of visible objects is mainly a result of a vast façade composed by many replicated windows, curtain wall elements and façade stones. The walkthrough sequence is similar as for the Library model, however, in the end interior, more occluded regions of the building are also visited. During these viewpoints the number of visible

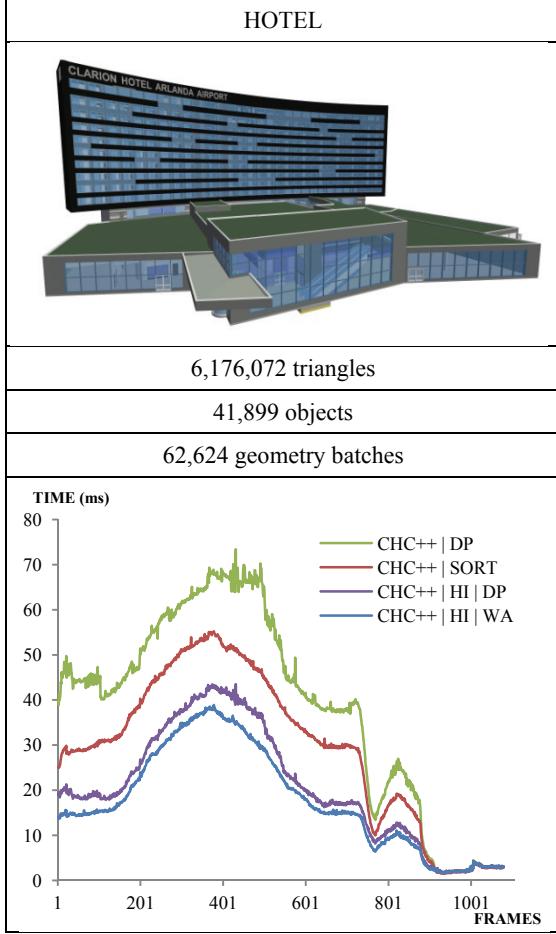
instanceable objects is low and, hence, few or none of them are rendered using hardware instancing. Still, in such viewpoints the occlusion culling system alone is able to deliver high performance and the important thing to note is that the proposed method only introduces a slight overhead, noticeable only in terms of relative numbers. For the non-interior parts of the walkthrough sequence an average speed-up of 1.7x was achieved with hardware instancing.

Table 3 also presents the performance results with depth peeling (DP). This approach guarantees a correct results but the performance penalty is higher compared to the weighted average technique. Nevertheless, compared to conventional sorting (SORT) a 1.5x speed-up was still achieved with instancing. On the other hand, when depth peeling was used in both cases (instanced and non-instanced) the average speed-up was almost 2x.

Figure 6 presents the number of draw calls with and without hardware instancing enabled for the Hotel model. This plot reveals the source of the performance gain. As can be seen, the numbers of draw calls are greatly reduced and, as a consequence, the performance is increased.

Table 2 also presents the performance numbers for the Hospital (middle) and Student Housing

Table 3: Statistics for the Hotel model and frame times for the proposed approach with occlusion culling (CHC++), hardware instancing (HI) and weighted average transparency (WA) compared to occlusion culling (CHC++) and conventional sort-by-object transparency (SORT) for the predefined walkthrough. In addition, the frame times for depth peeling (DP) are presented.



model (right). For the Hospital model an average speed-up of 1.6x is achieved with instancing. For the Student Housing model the performance gain of the proposed method is more moderate. Although an average speed-up of 1.25x is achieved, it is less than expected considering the model still has a fairly large amount of replicated components. However, compared to the other models, the animation sequence for the Student Housing model does not feature viewpoints equally beneficial in terms of instancing. First, the relative amount of visible replicated geometry is not as high and, second, the number of different geometry references is higher.

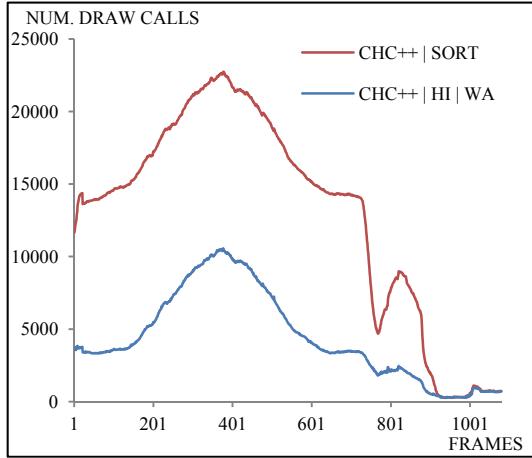


Figure 6: Number of draw calls with and without hardware instancing for the Hotel model.

## 7 CONCLUSION

This paper has presented a simple, yet efficient approach for integrating occlusion culling and hardware instancing. Compared to only using occlusion culling, average speed-ups of 1.25x – 1.7x were achieved for all test models in viewpoints where many objects are visible. Without dynamic hardware instancing activated, these viewpoints represent the worst case scenarios in terms of rendering performance.

The only aspect to consider a limitation is the requirement of an order-independent transparency rendering technique for semi-transparent geometry. A simple solution to remove this restriction would be to skip the use of hardware instancing for transparent geometry. Still, such geometry often possesses characteristics suitable for hardware instancing, which makes them tempting to include. For the tested models the weighted average technique was found to provide plausible results with high performance. In addition, depth peeling was shown to provide a viable option if a fully correct result is important.

For future work it would be interesting to test the proposed method together with other occlusion culling algorithms. The CHC++, although efficient, tightly integrates visibility determination and actual rendering of geometry. This puts restrictions on when collection, upload and rendering of instanced geometry can be performed. If these restrictions were relaxed, it is possible that a more efficient implementation of hardware instancing could be achieved.

Another area of further investigations would be the parameters NImin (minimum number of instances per geometry reference) and Tmax (maximum number of triangles for instanced geometries) for different scenes and hardware setups. Although the results show that uniform values for these parameters works in practice, it is likely that the performance could be further enhanced by letting NImin depend on triangle count (i.e. demanding a higher instance count for geometries with many triangles).

## REFERENCES

- Akenine-Möller, T., Haines, E., Hoffman, N. (2008). Real-Time Rendering 3rd Edition. A. K. Peters, Ltd., Natick, MA, USA.
- Bao, G., Li, H., Zhang, X., Dong, W. (2012). Large-scale forest rendering: Real-time, realistic, and progressive. *Computers & Graphics*, Vol. 36, Issue 3, Pages 140-151.
- Bavoil, L., Myers, K. (2008). Order Independent Transparency with Dual Depth Peeling. Tech. rep., NVIDIA Corporation.
- Bittner, J., Wimmer, M., Piringer, H., Purgathofer, W. (2004). Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful. *Computer Graphics Forum* 23, 3, pages 615–624.
- Bittner, J., Wonka, P. (2003). Visibility in Computer Graphics. *Environment and Planning B: Planning and Design* 30, 5, pages 729–756.
- Buchholz, H., Döllner, J. (2005). View-Dependent Rendering of Multiresolution Texture-Atlases. *Proceedings of the IEEE Visualization 2005*, Minneapolis, USA.
- Cohen-Or, D., Chrysanthou, Y. L., Silva, C. T., Durand F. (2003). A Survey of Visibility for Walkthrough Applications. In *IEEE Transactions on Visualization and Computer Graphics* 09, 3, pages 412–431.
- Dudash, B. (2007). Animated crowd rendering. In *GPU Gems 3*. Addison-Wesley, pages 39–52.
- Eastman, C., Teicholz, P., Sacks, R., Liston, K. (2011) BIM Handbook (2nd Edition) A guide to building information modeling for owners, managers, designers, engineers and contractors, John Wiley & Sons, New Jersey.
- Everitt, C. (2001). Interactive Order-Independent Transparency. Tech. rep., NVIDIA Corporation.
- Greene, N., Kass, M., Miller, G. (1993). Hierarchical ZBuffer Visibility. In *SIGGRAPH '93*, pages 231–238.
- Guthe, M., Balazs, A., Klein, R. (2006). Near Optimal Hierarchical Culling: Performance Driven Use of Hardware Occlusion Queries. In *Eurographics Symposium on Rendering 2006*.
- Hill, S., Collin, D. (2011). Practical, Dynamic Visibility for Games. In *Gpu Pro 2*.
- Johansson, M., Roupé, M. (2012). Real-Time Rendering of large Building Information Models. In proceedings of CAADRIA 2012 - Beyond Codes & Pixels, pages 647-656.
- Macdonald, J. D., Booth, K. S. (1990). Heuristics for ray tracing using space subdivision. *Visual Computer* 6, 6, pages 153–65.
- Mattausch, O., Bittner, J., Wimmer, M. (2008). CHC++: Coherent Hierarchical Culling Revisited. *Computer Graphics Forum (Proceedings Eurographics 2008)* 27, 2, pages 221–230.
- Park, H., Han, J. (2009). Fast Rendering of Large Crowds Using GPU. In *Entertainment Computing - ICEC 2008 (Lecture Notes in Computer Science, 5309)*, pages 197-202.
- Ramos, F., Ripolles, O., Chover, M. (2012). Continuous Level of Detail for Large Scale Rendering of 3D Animated Polygonal Models. In *Articulated Motion and Deformable Objects (Lecture Notes in Computer Science, 7378)*, pages 194–203.
- Sacks, R., Eastman, C.M., Lee, G. (2004). Parametric 3D modeling in building construction with examples from precast concrete. In *Automation in Construction* 13, pages 291–312.
- Steel, J., Drogemuller, R., Toth, B. (2012). Model interoperability in building information modelling. In *Software and Systems Modeling*, 11, 1, pages 99-109.
- Wloka, M. (2003). Batch, Batch, Batch: What Does It Really Mean? Presentation at Game Developers Conference 2003.