A Comparative Study on a Novel Drawcall-Wise Visibility Culling and Space-Partitioning Data Structures

Yvens Rebouças Serpa[1] * Ygor Rebouças Serpa[2] † Maria Andréia Formico Rodrigues[1] ‡

[1]Universidade de Fortaleza (UNIFOR), Programa de Pós-Graduação em Informática (PPGIA), Brazil
[2]Universidade de Fortaleza (UNIFOR), Centro de Ciências Tecnológicas (CCT), Brazil

Abstract

Computer games and animations often require high frame rates per second, which motivates the study of increasingly efficient and robust solutions in visibility culling, particularly in terms of processing time of 3D scenes. In this paper, we present a comparative analysis focused on the number of triangles and drawcalls necessary for the generation of varied scenes, taking into account combinations of features such as positioning, organization and partition of geometric elements. Furthermore, from the above analysis, we performed tests with different Space-Partitioning Data Structures (Octree, KDtree and Grid) and different visibility culling techniques (View-Frustum Culling and Backface Culling). The contributions of our work are therefore twofold: (1) a performance impact analysis of drawcall counts generated by several Space-Partitioning Data Structures for visibility culling; and (2) a novel approach to visibility culling that is drawcall-wise optimized. The results show that more competitive frame rates can be obtained if there is an optimum balance between visibility culling precision and number of drawcalls.

Keywords: visibility culling, space-partitioning data structures, walkthroughs.

1 Introduction

The increasing demand for games and graphical applications, specially digital games, with high frame rates has fostered the study of techniques to improve the efficiency and robustness of the rendering pipeline [8] [10] [19]. Among the pipeline stages, the visibility culling process [5] has attracted notorious attention in this regard, being responsible for discarding geometrical primitives unseen by the viewer [11] [15]. However, a more fine-grained culling, *per se*, does not necessarily implies a better performance [4] [19]. Often, the more precise this process is, the more rendering calls to the graphical API are required. Such calls are known as *drawcall*s. Drawcalls are associated with a fixed-cost and, thus, should be kept to a minimum [19]. On the other hand, a coarser visibility culling may submit too many unseen triangles, degrading the overall performance.

Traditionally, different techniques exist for visibility culling. Among these, we highlight *View-Frustum Culling* and *Backface Culling* [2] [21]. The former concerns with the culling of geometrical primitives that lie outside of the viewing *frustum* [1] [2], while the latter focuses on culling primitives within the frustrum, but that present their back face to the viewer, *i.e.*, with the normal vector pointing to the same direction as the view vector [21]. Optimization strategies have been proposed to accelerate these tasks. Basically, they employ Space-Partitioning Data Structures (SPDS) to solve the problem in a divide-and-conquer manner [3] [14] [20]. However, although several distinctive approaches have been explored, to the best of our knowledge, the role of drawcalls in the context of visibility culling is so far scarcely mentioned on the academic literature, being treated mostly on the industrial domain [19]. In the light of this context, the contributions of our work are twofold: (1) a performance impact analysis of drawcall counts generated by several SPDS for visibility culling; and (2) a novel approach to visibility culling that is drawcall-wise optimized.

We divide our work into three main parts: (1) an analysis of the relationship between triangles and drawcalls counts to the frames-per-second (FPS) metric, using a geometry generator; (2) the definition of three data-structures (Grid, Octree and KDtree) and three culling approaches, including our novel Drawcall-Wise method, which combined produce 9 different culling strategies; and (3) a case-study of these strategies applied to a walkthrough on three-dimensional complex scenes (taken from a known computer game), which extensively evaluates the solutions on both the traditional FPS metric and the number of drawcalls metric.

*e-mail: yvensre@gmail.com
†e-mail: ygor.reboucas@gmail.com
‡e-mail: andreia.formico@gmail.com

2 Related Work

Samet and Webber [14] present a comprehensive study on the main algorithms for visibility culling: View-Frustrum Culling, Backface Culling, Occlusion Culling and several SPDS used in this context. Later on, Cohen *et al.* revisited this theme, updating it to more recent contributions [5]. On the theoretical side, Laakso [11] contributed with a reformulation of the problem as determining the concept of the potentially visible set (PVS) of triangles. Based on it, several authors employed this terminology for a more mathematical foundation [1].

More specifically to the View-Frustrum Culling, studies have been performed on spatial optimizations [2] [3], such as the use of: bounding boxes, *e.g.*, AABBs and OBBs [7]; SDPS construction algorithms [1] [13]; and temporal optimization techniques, such as temporal coherency [6] [17], which preconizes that the PVS of the next frames of an animation sequence will usually keep the visibility status of the current frame during a short amount of time [12].

More in depth on the SPDS side, Naylor [13] presents a SPDS construction algorithm based on probability theory. In another direction, Andrysco and Trichoche [1] present an approach to outsource the SPDS construction to graphics processing units (GPUs), obtaining significant results for both dense and sparse geometrical meshes. Finally, some works make a distinction between aggressive and conservative culling, *i.e.*, if the technique is suitable for culling visible primitives or not, respectively [9] [15] [20].

In parallel, several works have been developed on the Backface Culling and Occlusion Culling [21] [22]. Zhang and Hoff proposed an approach in which the Backface Culling algorithm is performed in a discrete way, partitioning triangles on usually 8 major viewing directions based on *bit* masks [21]. Given the view-vector, it is possible to safely exclude at least 3 of such directions, as being backfaced to the user, by testing the view-vector *bit* mask against the viewing-direction *bit* masks. On the other hand, the Occlusion Culling algorithm proposed by Zhang *et al.* uses a hierarchical tech-

nique for discarding triangles occluded by others [22]. In common, both works share the use of *bit* masks to optimize several look-up operations, leading to fast running time.

Exploring all these approaches combined and how they relate, Silva and Rodrigues [15] present a comparative study on several algorithms for Visibility Culling paired with Grids, BSP-Trees, Octrees and Portal-Octrees applied to 3D scenes on mobile devices. Moreover, the authors focus on a conservative Visibility Culling. Wald *et al.* explore a novel way for using Grids for the visibility culling problem, however, focused on ray-tracing and dynamic scenes [18].

Applied directly to digital games, Gregory [8] reinforces the need for an optimized graphical rendering pipeline, thus, most of its time should be spent on triangles that will be effectively seen by the player. To the same end, Wloka [19] stresses the importance of minimizing the number of drawcalls to maximize the visual throughput. Intuitively, both remarks should be followed. However, to the best of our knowledge, works regarding the Visibility Culling problem seldom address the number of additional drawcalls generated by their approaches or its impact on the FPS metric. Furthermore, hierarchical structures, if used, can easily increase the number of drawcalls exponentially, aggravating this issue. On this path, this work proposes both an analysis and solution strategies for the Visibility Culling problem in the context of drawcall minimization.

3 Visibility Culling Algorithms

For digital games and other interactive graphical applications, the process of discarding triangles unseen by the viewer must occur at each frame, updating what is considered visible and what is not, in terms of the viewing vector and position. In this work, we simplify this task by excluding the Occlusion Culling algorithm, leaving only the View-Frustrum and Backface Culling algorithms. This simplification by no means reduces the generality of the work and was carried out to avoid too many combinations of algorithms. In the following subsections, we describe in more detail the two main culling approaches and how they were implemented.

3.1 *View-Frustum Culling* and *Backface Culling*

The View-Frustum Culling algorithm is responsible for determining the portions of the scene that lie outside the view *frustum*. Commonly, a SPDS is employed to query which portions are inside the *frustum*, solving the task in a divide-and-conquer manner. While sophisticated solutions for this task exists, they are out of the scope of this paper. Thus, we have developed three SPDS: Grid, Octree and KDtree. These SPDS will be further explained in Section 3.2.

Executed after the View-Frustum Culling, the Backface Culling algorithm is responsible for excluding primitives that face the same direction as the viewer. Such primitives are backfacing the user, thus, shall not be rendered. In comparison to the earlier task, this is a much more fine grained process, since backfaced triangles are likely scattered all over the viewing *frustum*. For this task, we developed a single solution, based on the Zhang Backface Culling algorithm[21]. Both these methods are illustrated in Figure 1.

The Zhang Backface Culling algorithm processes the scene and divides it into N groups. Each group contains all triangles that are facing $1/N$ of all possible directions. For instance, considering only the $x-z$ plane and taking N as 8, the groups correspond to the following angle ranges [0°, 45°], [45°, 90°], ..., [315°, 360°]. This is shown in Figure 2. To cull triangles, the viewing vector is considered. Let G_i be the group the viewing vector belongs to, we can safely discard all triangles of the groups G_{i-1}, G_i and G_{i+1}.

When paired with the View-Frustum Culling, the Zhang Backface Culling algoritm partitioning strategy is pushed down to a per-cell or per-leaf scale. For instance, on a KDtree, the geometry of each leaf is partitioned into the N Zhang's partitions. Finally, in the context of drawcalls, for each visible cell/node, there can be up to $N-1$ drawcalls, *i.e.*, all groups are non-empty.

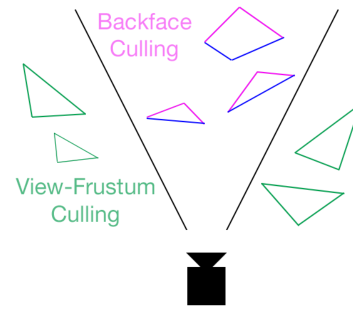


Figure 1: Illustrative representation of the View-Frustum and Backface Culling algorithms.

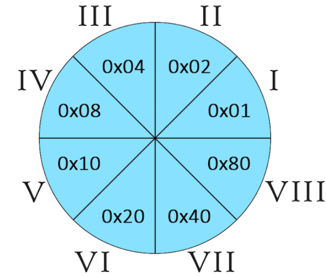


Figure 2: Zhang masks inside an unity circle. Given that the view vector is one of the eight groups, the given group and its two neighbours can be safely discarded. For instance, if the view vector lies on section IV, we can discard sections III, IV and V.

In our implementation, we disregarded the $x-y$ plane for generating groups, focusing only on the $x-z$ plane. As a consequence, faces in which the normal vector are not sufficiently aligned to the $x-z$ plane are grouped into a special partition, called Non-Zhang Partition, that is ignored by the Zhang Backface Culling algorithm and are always rendered with one drawcall.

3.2 Space-Partitioning Data Structures

In the present work, we have implemented the following SPDS: Grid [18] [20], Octree [15] and KDtree [1]. In the following sections, we discuss in details the main features of these SPDS.

3.2.1 Grid

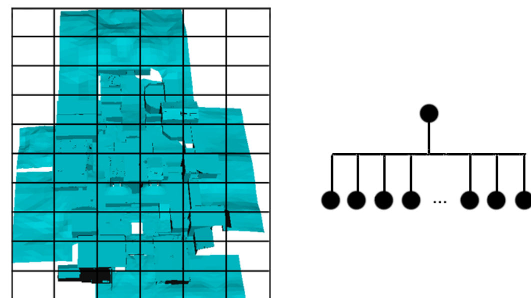


Figure 3: Illustration of the Grid implementation on the $x-z$ plane.

Among the implemented structures, the Grid is the simplest one, being the less flexible and less expressive in terms of partitioning capabilities. This structure divides the entire scene in equally sized

cells divided over a number of rows and columns. Three dimensional Grids are possible, but seldom used within the visibility context. For simplicity, we only allow square Grids, *i.e.*, same number of rows and columns, as show in Figure 3.

The Grid visibility culling consists in identifying all cells that intercept the viewing *frustum*. These cells, are considered visible while the others are discarded. When compared to the other structures, while unable to cull on the y axis, the Grid is able to achieve a higher precision on the remaining x and z axes. Therefore, it is recommended for scenarios where the y component is not really explored, such as outdoors scenes with huge open areas.

3.2.2 Octree

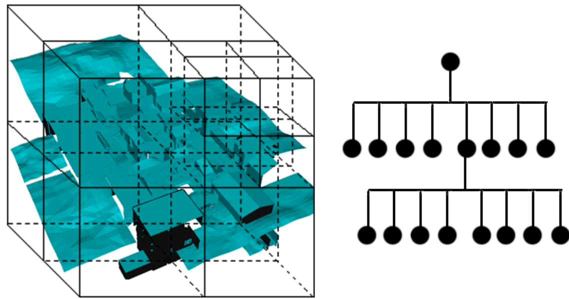


Figure 4: Illustration of the Octree implementation.

While the Grid is a flat structure, the Octree presents a hierarchical relationship of cells. Starting from the root node, which encompasses the entire scene, each sub-sequent level divides the scene in all three axes on 8 sub-regions, as shown in Figure 4.

The Octree visibility culling process is performed by exploiting the hierarchical nature of the structure. If a node and the *frustum* have an intersection the process is recursed to all 8 children nodes, if they do not, then we can safely discard the node and all its children nodes. At the end of the recursive process, the algorithm returns the list of visible leaf nodes. Compared to the other SPDS, the Octree excels at scenarios where all three axes are equally well distributed, since it is unable to favor one axis over the other. Examples include dense urban scenes with lots of buildings and architectural structures.

3.2.3 KDtree

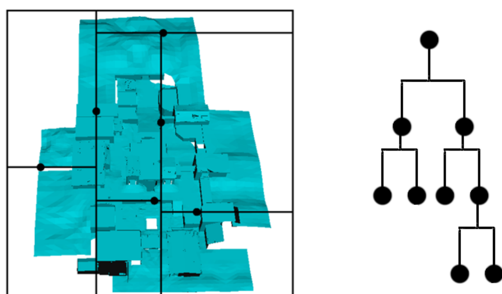


Figure 5: Illustration of the KDtree implementation.

Finally, the KDtree is the most expressive and flexible of all three structures, consisting of a binary tree where each level selects an axis and a point to build a splitting plane. For that, a heuristic is employed to guide the splitting process. For instance, one simple heuristic would be to alternately choose the x and z axes and split them on their midpoint. This heuristic generates a partitioning equivalent to a Grid on the $x-z$ plane. Likewise, it is possible to build an Octree-equivalent partitioning by alternating between all three axes. However, the strength of this structure is the possibility to express a partitioning that lies in-between the two approaches. In this work, at each level, we chose as splitting axis the one with the highest variance on its triangles and we use the mean axis component value of all triangles as the splitting point (Figure 5).

Given a node and the frustrum, a KDtree performs the visibility culling considering three cases: the frustrum is completely to the left; the *frustum* is completely to the right; or, it is intercepting the plane. If it is located to the left, we recurse to the left sibling. Likewise, if it is to the right, we recurse to the right. In case it is intercepting the plane, we have no option but to recurse to both children nodes. As for the Octree, by the end of the procedure, the algorithm yields the list of all visible leaf nodes. Finally, when comparing this algorithm to the others, it presents more overhead, since it needs to be much deeper to express the same amount of regions. However, it is able to shift its focus from axis to axis, suiting itself to the scenario's properties. This SPDS is a good all-around choice and stands out on the range of scenarios between the preferred ones for the Grid and the ones for the Octree.

3.3 Triangles Redundancy

During the SPDS construction, triangles may end up intercepting a partitioning plane when a region is subdivided, not belonging uniquely to neither left nor right side. For instance, consider an Octree of height 2 processing a scene with a triangle as big as the scene itself, such triangle would inevitable belong to all of its nodes, which would all reference and draw it. While such a triangle rarely exists, it is actually pretty common for a triangle to cross two or three nodes. This redundancy affects the overall performance. However, in the structures we have developed, this was not identified as a significant issue and, thus it is out of scope of this work.

4 Drawcall Minimization

For each SPDS, we have developed 3 variant implementations, combining the SPDS with the use or not of Backface Culling and the Drawcall minimization strategy. These combinations are: Simple SPDS; Drawcall-Wise SPDS; and Zhang Drawcall-Wise SPDS.

4.1 Simple SPDS

The simple approach basically ignores the drawcalls issue, employing the SPDS normally. When computing the SPDS, all the vertex information that should go on a certain cell, is likely scattered throughout the entire vertices lists. Therefore, each cell has to perform several drawcalls, rendering each contiguous set of vertex one at a time. For that reason, it generates several drawcalls per node. For a deep structure, this is prohibitive. On the other hand, only one copy of the entire geometry is needed, saving up memory.

4.2 Drawcall-Wise SPDS

Unlike the Simple approach, the Drawcall-Wise strategy maintains no global copy of the entire geometry, but several subsets of it, one for each node in the structure. When visible, the node only has to submit its share of the model, yielding a single drawcall. As a drawback, some of the geometry ends up replicated, since some primitives are shared over a couple of nodes. This leads to a higher memory footprint, at the expense of a greatly reduced number of drawcalls.

4.3 Zhang Drawcall-Wise SPDS

Finally, the Zhang Drawcall-Wise SPDS works similarly to the Drawcall-Wise SPDS. However, each leaf node divides its mesh in

9 submeshes, 1 for the Non-Zhang partition and 8 for the Zhang partition, according to the Zhang Backface Culling algorithm. When rendered, each leaf node does 1 drawcall for the Non-Zhang partition and 1 drawcall for each visible partition based on the view vector. The Zhang Drawcall-Wise SPDS is the more precise in terms of culling than the others. While still more efficient than Simple in terms of drawcalls, this approach performs some trade-off of additional drawcalls per node in exchange for a more fine-grained culling.

5 Triangles, Drawcalls and FPS

To stress the importance of correlating both triangles and drawcalls counts to the FPS metric simultaneously, we generated a dataset of half a million triangles and subjected it to several rendering settings to assess the performance, independently of any application and prior/posterior optimizations.

Each test consists of taking the first N triangles of the dataset, splitting it into D equally sized groups and, then, for each frame, render all groups sequentially, *e.g.*, for $N = 10.000$ and $D = 10$, we generate 10 drawcalls per frame of a thousand triangles each. Table 1 presents several combinations of number of triangles and drawcalls with their respective FPSs.

Table 1: Mean FPS for several combinations of triangle and draw counts, generated from a dataset of 500.000 randomly generated triangles.

	Number of Drawcalls					
Triangles	**1**	**10**	**100**	**1000**	**10000**	**100000**
100k	577,03	528,54	496,21	465,59	233,28	47,19
150k	372,10	348,93	296,58	273,56	178,52	47,44
200k	279,89	246,21	222,36	208,47	152,74	41,06
250k	222,98	202,46	187,30	181,78	129,13	41,36
300k	188,79	179,01	149,34	138,93	106,63	38,65
350k	161,04	151,23	126,91	118,00	108,92	38,89
400k	143,09	103,33	115,66	115,94	94,94	38,76
450k	126,02	118,80	100,35	92,16	84,97	38,42
500k	115,14	104,58	89,84	85,07	75,09	38,79

Considering only the number of triangles, an almost linear relationship to the FPS measure can be seen in column 1 of Table 1. However, as we increase the number of drawcalls, the relative importance of the number of triangles to the final FPS value gradually decreases, as the overhead of more drawcalls increases. Numerically, for a single drawcall, 100.000 triangles are drawn at 570 FPS and 500.000 triangles at 115 FPS, *i.e.*, a 5 times lower FPS for a 5 times bigger 3D mesh. For a hundred thousand drawcalls, 100.000 triangles are drawn at 47 FPS and 500.000 triangles are drawn at 38 FPS, only 20% slower. Furthermore, when drawing the entire dataset, *i.e.*, 500.000 triangles, with one single drawcall, it still performs better than when drawing only a fifth of it with one hundred thousand drawcalls. Figure 6 shows the graphs representing this behavior.

From the perspective of optimization analysis, several possibilities exist to improve the FPS in terms of triangle counts and drawcalls. Consider the case of 300,000 triangles with 100 drawcalls. Obviously, lowering either the number of triangles, drawcalls or both, will result on a better performance. However, two other options exist: (1) the culling can be relaxed up to 50,000 triangles as long as drawcalls are reduced by 90, and (2) the culling can be tightened up to remove more 50,000 triangles under the budget of 900 drawcalls. In other words, the performance can be improved by moving diagonally on the triangle-drawcalls space.

While this is just an example, it is pretty much a best-case scenario according to Wloka [19]. In a real-world application, such as digital games, several user operations are performed for each

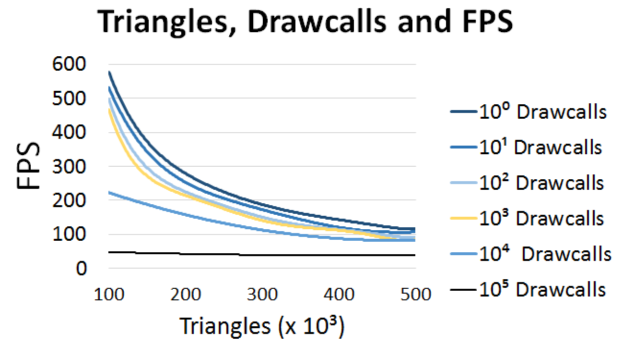


Figure 6: Polynomial regression of triangle, drawcall and FPS data displayed in Table 1. From top-to-bottom, the increased number of drawcalls gradually decreases the performance. For 10^5, the overhead completely dominates the execution time, insensible to the number of triangles.

drawcall, aggravating this issue, such as what occurs when dealing with skinning meshes or texture/shader changes. Furthermore, many drawcalls are hard to avoid, such as those that are typical of dynamic objects. Therefore, a significant amount of drawcalls is an inevitable reality. For instance, a naive Octree of height H for View-Frustrum Culling algorithm can yield up to 8^H drawcalls. While the goal is to cull as many unseen primitives as possible, it is prohibitive to do so in a completely drawcall unaware fashion.

6 Tests and Results

This section deals with the type of tests we have conducted and the results obtained.

6.1 Scenarios and Walkthroughs

To comprehensively analyse the drawcall issue within a real-world dataset, we have selected and used 3D scenes from the critically aclaimed *Team Fortress 2* game [16]. The scenes, named Coaltown, Manor and Coldfront, were chosen due to their for having distinctive dimensions, triangle counts, irregular and rocky terrain and mixed indoor and outdoor scenery. We show these 3D game scenes and their respective number of triangles in Figure 7.

For each scenario, a different walkthrough was developed to be used as a test case. Each walkthrough covers important regions of the scenario, getting in and out of indoor zones and exposing the underlying algorithms to several stressing settings, such as balconies with view to most of the outdoor scenery.

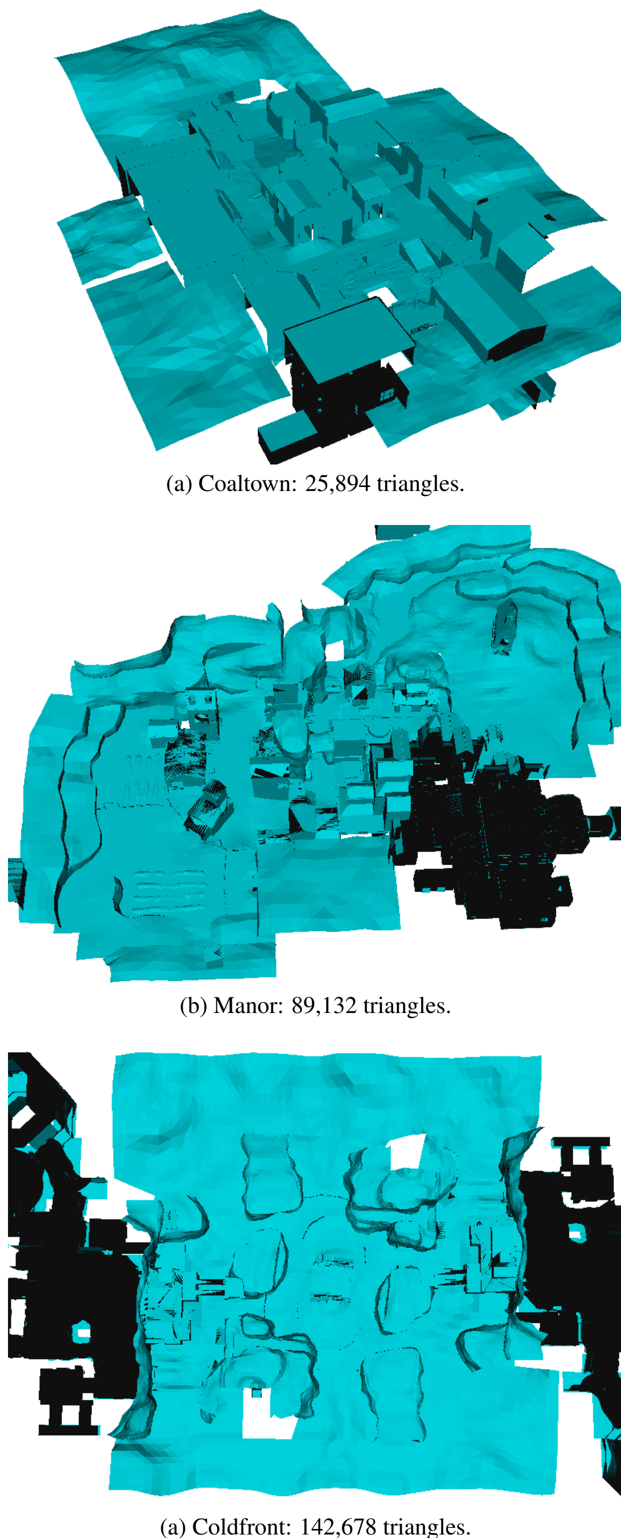
6.2 Framework

We composed several ensembles of algorithms by combining the SPDS, the use or not of the Zhang Backface Culling and the drawcall minimization strategy. For comparative purposes, we added the "None" algorithm, which consists in no culling usage, *i.e.*, plainly drawing the entire geometry with a single drawcall.

All paths were executed for all algorithm compositions and the FPS metric, drawcall numbers and number of triangles sent to the rendering pipeline were accounted. The tests were run using a AMD FX 8320 Eight-Core Processor, 3.50 GHZ with 8.00 GB RAM memory, running on a GeForce GTX 960 4.00 GB GPU.

6.3 Results

On all carried out tests, it is possible to visualize that the Simple approach is always the least efficient among the tested approaches. When compared to the others, the Simple variation reaches up to 100 times more drawcalls than all other approaches. We can note



(a) Coaltown: 25,894 triangles.

(b) Manor: 89,132 triangles.

(a) Coldfront: 142,678 triangles.

Figure 7: 3D game scenes taken from the *Team Fortress 2* game and their respective number of triangles. Outdoor regions were shaded in blue while the indoor regions were shaded in black.

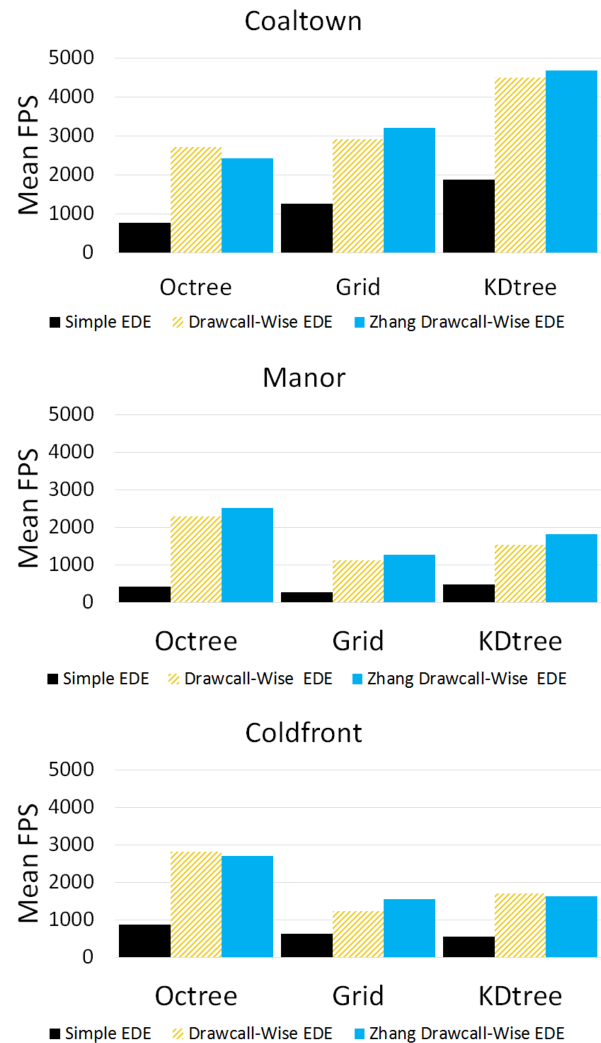


Figure 8: Mean FPS comparison of all 6 combinations of SPDS, using the Backface Culling algorithm and the drawcall minimization strategy.

that the Drawcall-Wise and Zhang Drawcall-Wise attempting to minimize the drawcall counts, perform far better, reaching more than twice the FPS, when compared to the Simple approach. On average, the Zhang Drawcall-Wise approach performs better than the pure Drawcall-Wise. However, in a real-world application, where the cost to render each triangle is usually higher than on our testing framework (that is, due to more demanding shaders and effects) the Zhang approach is expected to perform better, since it culls a higher number of triangles.

Regarding the impact of the structure size, results show that the height of 4 is the most suitable height for all scenarios. The significant exceptions are the Octree on Coaltown, which performed better with a height of 2, and the Zhang Drawcall-Wise Grid on Coldfront, which performed better with a size of 5.

6.3.1 FPS and Triangles

Collectively, all scenarios presented distinctive results, favouring one technique over the others. Additionally, there is a strong correlation between a high FPS and a high culling precision. However, plainly maximizing the culling has shown to be the worst approach (Simple), confirming the importance of weighting the number of drawcalls when designing culling algorithms.

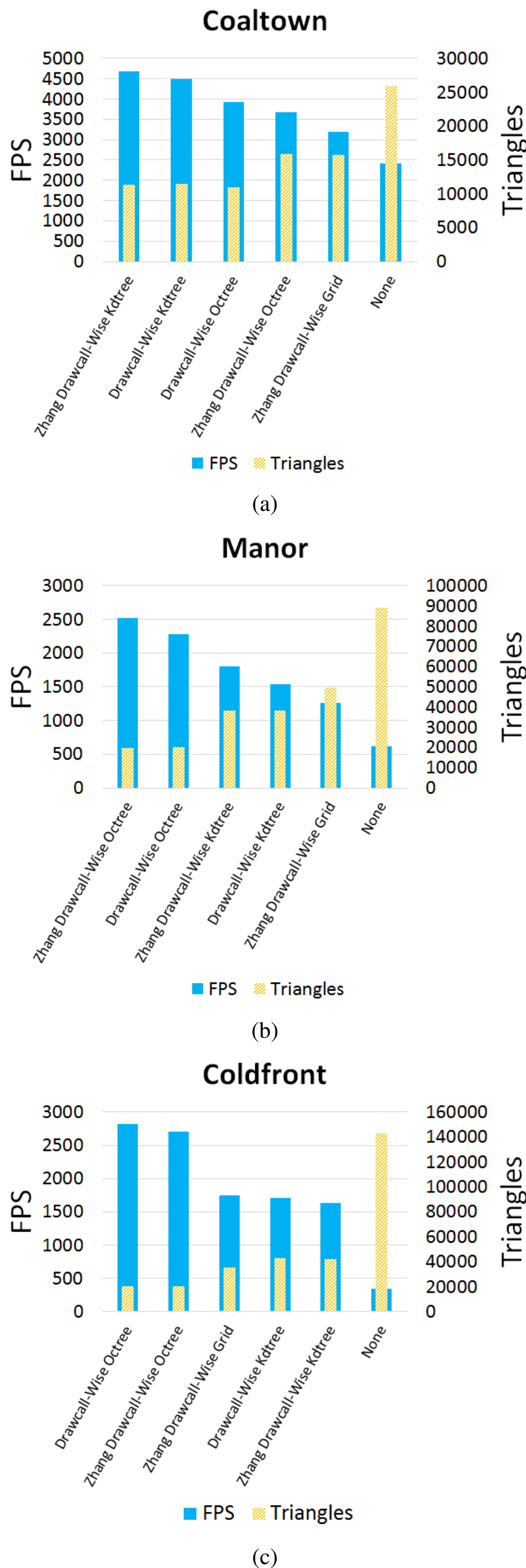


Figure 9: Combined FPS and triangle counts for the best performing approaches in each scene and the baseline None approach.

Individually, each scenario is either biased to the Zhang or non-Zhang approach. This can be explained by analysing their geometries one by one. The first, Coaltown, is composed by simple houses and walls, with a overall simple geography. Due to its smaller dimension, often, most of the scenery is considered visible for the Viewing-Frustrum Culling algorithm, increasing the importance of the Backface Culling. Thus, it favors Zhang approaches over no Backface Culling.

The Manor scene is also biased towards Zhang approaches. However, unlike the previous scene, this is not an implication of its size, but of its topology. As the name suggests, the scene contains a mansion, thus, many walls and other $x-y$ aligned geometry, *i.e.*, triangles with normal vectors on the $x-z$ plane. Therefore, a significant share of its triangles belong to one of the 8 Zhang partitions, allowing the Backface Culling to be more effective.

Finally, the biggest and most complex of all three scenes, Coldfront, favors the non-Zhang methods. Again, this is a result of its topology. Similar to Manor, this scene presents a mixture of indoor and outdoor regions, yet, it has a much higher presence of non $x-y$ aligned geometry, reducing the amount of culling opportunities available to Zhang approaches. For instance, a significant portion of this scene corresponds to an irregular rocky terrain. Additionally, due to its large dimensions on all 3D axes and significantly higher triangles count, the Octree approach was the best solution because of its higher amount of nodes.

6.3.2 Walkthrough Analysis

We have analyzed each walkthrough individually, as shown in Figure 10. The walkthrough on Coaltown starts near the edge of the scene, with the observer looking directly to the outside, hence, most of the scene is discarded by the algorithms. This is the highest FPS peak of the entire walkthrough for all algorithms. For almost the entire path, due to the scene limited dimensions and number of far triangles, the Octree solutions become rather excessive, generating too many drawcalls for a not so significant additional culling, when compared to the KDtree, for instance. The same applies to the Grid. For this reason, the best Octree height for this scene was 2, unlike all other scenes, where it was 4.

The walkthrough on Manor scene starts at the center of the scene, where a huge amount of triangles are visible. At this point, the methods that more accurately cull triangles excel. In sequence, the viewer moves towards the Manor, where the Backface Culling is the most effective. For this reason, the Zhang methods have outstanding performance during most of this path, even though they generate more drawcalls. As an example, the Zhang Drawcall-Wise Octree exhibits the highest FPS of all approaches for most of this test. Numerically, this approach takes about 60 more drawcalls more in exchange for culling about a thousand triangles more, in average. On the worst algorithms side, the None approach, again, exhibits the lowest FPS, followed by the Simple approaches. Besides FPS, we analysed the SPDS usage for this scenario. As we mentioned previously, the best performing size for all structures was found to be four. The Octree, for instance, produces 8^4 or 4096 nodes. However, most of these are empty. In more practical terms, only about 256 nodes are actually created. Moreover, in average, only 50 nodes are visited per frame, considering both internal and leaf nodes.

Finally, on Coldfront, the observer starts at the middle of the scene, between the two main buildings. During the initial moments, the observer does a complete turn around itself, visualizing the entire scene and, thus, demanding a very good culling from the algorithm. Additionally, most of the landscape around the viewer is highly irregular and rocky, inadequate for a good action of Zhang-based approaches. After that, the observer starts to move towards the left building. From that moment, the KDtree starts to obtain a higher FPS metric since its space-partitioning technique is able to cull almost half of the scene on fewer steps than the other SPDS.

Since the scene has a good distribution on triangles, the initial splitting planes of the KDtree approaches are located near the scene's center, explaining the good performance while the observer moves towards the edge. With the exception of the Grid, which performed better with a height of 5, all other SPDS had a better performance at a height of 4, like the previous scene. The Drawcall-Wise Octree achieved the better FPS metric among the other SPDS, even with the highest number of nodes, thus drawcalls, testing a mean of 51 nodes per frame. The Zhang Drawcall-Wise Octree performed a mean of 126 drawcalls per frame, culling approximately 150 more triangles per frame, but performed worse than the Drawcall-Wise Octree, since its culling have not improved enough to counteract the higher number of drawcalls. Differently than the previous scenes, on this the Zhang Drawcall-Wise Grid outperformed the Zhang Drawcall-Wise and Drawcall-Wise KDtree, culling more effectively, by sending over seven thousand triangles less and performing approximately 23 drawcalls, on average, compared to the KDtree approaches.

7 Conclusions and Future Work

In this work, we presented a comparative study of the number of triangles, drawcalls and FPS on a graphical application, showcasing a direct correlation between the three mentioned variables, thus implying its importance to the decision-making process of Visibility Culling design.

Following these footsteps, we presented the design of several culling approaches without drawcalls in mind and a novel approach on drawcall-wise visibility culling, which were compared to each other in three real-world problems for benchmarking. When running the approaches for each scene, the drawcall-unaware solution was strictly worse than its drawcall-aware counterparts. Moreover, not always the extra drawcalls related to Backface Culling provided enough extra culling to outweight its overhead. Actually, a strong relationship exists between the scene topology and the presence or not of profit in performing the additional Backface Culling step.

Finally, and most important, the findings of our later analysis corroborated our prior observations on the triangles-drawcalls trade-off to achieve maximum throughput. Furthermore, as expected, on a more complex setting, such as a digital game, the impact of drawcalls is much higher than on our original sample dataset. In this work, besides the API overhead itself, the visibility lookup(s) operation is part of the drawcall impact, *e.g.*, when increasing the level of an Octree, not only a higher number of drawcalls occur, but a higher number of nodes are tested. On a full-fledged digital game, this is even worse, since many other operations besides a structure look-ups may be tied to increased drawcall amounts.

As future work, we anticipate that drawcall counts can be pushed even further down by employing either smarter SPDS and memory locality. Additionally, the idea of a parametric structure that is able to adapt its culling precision in runtime is appealing in this context. Besides further SPDS improvements, the impact of Occlusion Culling algorithm and a Zhang Backface Culling algorithm extended to both $x-y$ and $x-z$ planes needs to be investigated to generate a more complete picture of the problem. In the same line, the number of testing scenarios could be extended to delve on much more deeply and effectively into higher triangle counts, in the order of tens of millions.

Acknowledgments

Yvens R. Serpa, Ygor R. Serpa, and Maria Andréia F. Rodrigues would like to thank the Brazilian Agencies FUNCAP-CE (Process Number PEP-0094-00005.01.09/2014) and CNPq (SWG Process Number 212628/2014-3), as well as the Universidade de Fortaleza - UNIFOR (Process Number 2038) for the financial support.

References

[1] N. Andrysco and X. Tricoche. Implicit and dynamic trees for high performance rendering. In *Proceedings of the 2011 Graphics Interface*, pages 143–150. Canadian Human-Computer Communications Society, 2011.

[2] U. Assarsson and T. Moller. Optimized view frustum culling algorithms for bounding boxes. *Journal of Graphics Tools*, 5(1):9–22, 2000.

[3] J. Bittner. *Hierarchical techniques for visibility computations*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2002.

[4] A. Chandak, L. Antani, M. Taylor, and D. Manocha. Fastv: From-point visibility culling on complex models. *Computer Graphics Forum*, 28(4):1237–1246, 2009.

[5] D. Cohen-Or, Y. L. Chrysanthou, C. T. Silva, and F. Durand. A survey of visibility for walkthrough applications. *Visualization and Computer Graphics, IEEE Transactions on Visualizations and Computer Graphics*, 9(3):412–431, 2003.

[6] N. K. Govindaraju, A. Sud, S.-E. Yoon, and D. Manocha. Interactive visibility culling in complex environments using cclusion-switches. In *Proceedings of the 2003 Symposium on Interactive 3D graphics*, pages 103–112. ACM, 2003.

[7] D. Green and D. Hatch. Fast polygon-cube intersection testing. *Graphics Gems V*, pages 375–379, 1995.

[8] J. Gregory. *Game engine architecture*. CRC Press, 2009.

[9] S. Gummerus. Conservative from-point visibility. *Master's Thesis, University of Tampere*, 77:22, 2003.

[10] T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff, and H. Zhang. Accelerated occlusion culling using shadow frusta. In *Proceedings of the XIII Annual Symposium on Computational Geometry*, pages 1–10. ACM, 1997.

[11] M. Laakso. Potentially visible set (pvs). *Helsinki university of technology*, 2003.

[12] O. Mattausch, J. Bittner, and M. Wimmer. Chc++: Coherent hierarchical culling revisited. In *Computer Graphics Forum*, volume 27, pages 221–230. Wiley Online Library, 2008.

[13] B. Naylor. Constructing good partitioning trees. *Graphics Interface*, pages 181–181, 1993.

[14] H. Samet and R. E. Webber. Hierarchical data structures and algorithms for Computer Graphics I - Fundamentals. *Computer Graphics and Applications, IEEE*, 8(3):48–68, 1988.

[15] W. B. Silva and M. A. F. Rodrigues. Interactive rendering of indoor and urban environments on handheld devices by combining visibility algorithms with spatial data structures. *International Journal of Handheld Computing Research*, 2(1):55–71, Jan. 2011.

[16] VALVE. Team Fortress 2. `http://www.teamfortress.com/`. Accessed: 2016-06-07.

[17] G. van den Bergen. Collision detection in interactive 3D environments. 2004.

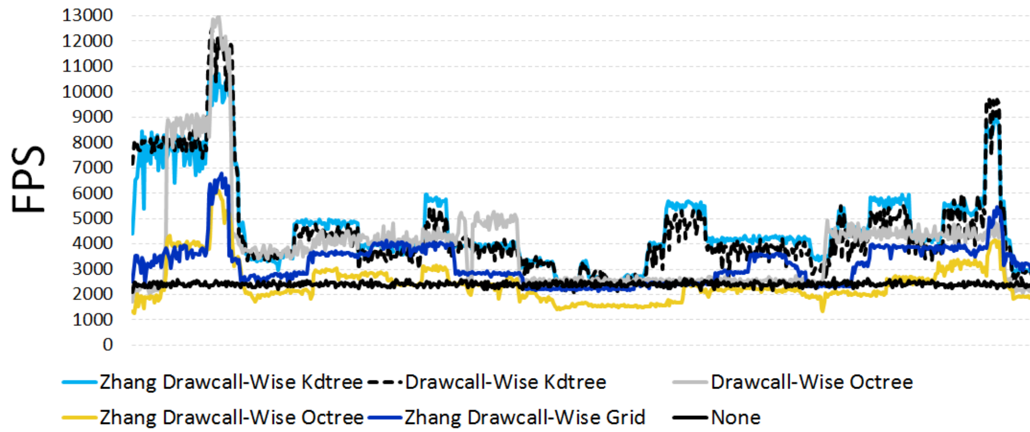
[18] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker. Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics (TOG)*, 25(3):485–493, 2006.

[19] M. Wloka. Batch, batch, batch: What does it really mean? Presentation at Games Developer Conference, 2003.

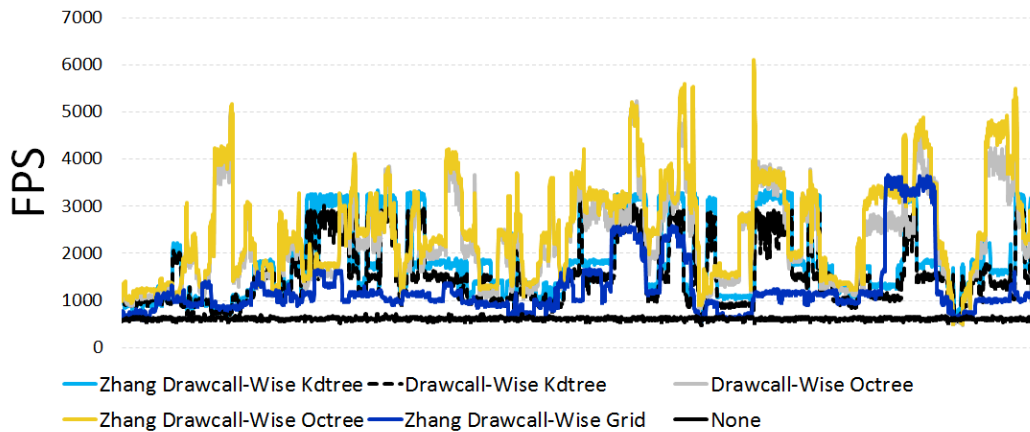
[20] T. Yılmaz and U. Güdükbay. Conservative occlusion culling for urban visualization using a slice-wise data structure. *Graphical Models*, 69(3):191–210, 2007.

[21] H. Zhang and K. E. Hoff III. Fast backface culling using normal masks. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, pages 103–ff. ACM, 1997.

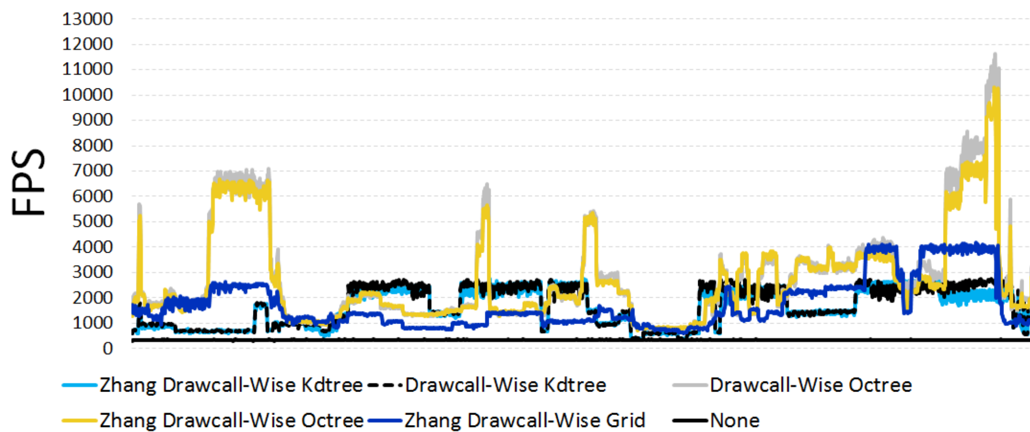
[22] H. Zhang, D. Manocha, T. Hudson, and K. E. Hoff III. Visibility culling using hierarchical occlusion maps. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, pages 77–88. ACM Press/Addison-Wesley Publishing Co., 1997.



(a) Coaltown walkthrough and its SPDS.



(b) Manor walkthrough and its SPDS.



(c) Coldfront walkthrough and its SPDS.

Figure 10: Walkthroughs on all 3D scenes.