GEOMETRY – VISIBILITY – ILLUMINATION

GAMETOOLS

Advanced Tools for Developing Highly Realistic Computer Games

Initially Working Visibility Modules

Document identifier:	**GameTools-3-D3.3-03-1-1-Initially Working Visibility Modules**
Date:	**08/05/2006**
Work package:	**WP03: Visibility**
Partner(s):	**VUT**
Leading Partner:	**VUT**
Document status:	**Final Version**
Deliverable identifier:	**D3.3**

Abstract: This technical report describes the initially working modules of the visibility work package.

Delivery Slip

	Name	Partner	Date	Signature
From	Jiri Bittner	VUT	04/05/2006	
Reviewed by	WP Managers	All		
Approved by	WP Managers	All		

Document Log

Issue	Date	Comment	Author
1-0	30/04/2006	Draft Version	Jiri Bittner
1-1	04/05/2006	Final Version	Jiri Bittner

Document Change Record

Issue	Item	Reason for Change

Files

Software Products	User files / URL
Word	gametools-ist-2-004363-3-d3.3-03-1-1-initially working visibility modules.doc

CONTENT

1. INTRODUCTION

1.1. OBJECTIVES OF THIS DOCUMENT

This document describes the initially working modules for the Visibility Work Package. Its aim is to describe the modules and explain how they work.

1.2. DOCUMENT AMENDMENT PROCEDURE

Any project partner may request amendments but each amendment must be analysed and approved by the GameTools Project Coordinator or Project Manager.

1.3. TERMINOLOGY

Glossary

GTP	GameTools Project
PC	Project Coordinator
PM	Project Manager
WP	Work Package

2. DESCRIPTION OF THE MODULES

2.1. MODULES OVERVIEW

The visibility work package (WP3) consists the following modules:

- View space partitioning (task 3.1)
- Computation of Potentially Visible Set (PVS computation - task 3.2)
- Online visibility culling (task 3.3)

The view space partitioning and PVS computation are implemented as a standalone application. The view space partitioning takes as input a static part of the scene and outputs a set of view cells. The PVS computation uses these view cells and to determine potentially visible sets which are saved in a single XML file together with the view cells. Additionally there is a library version of the view space partitioning module, which can be used as a runtime interface to the preprocessed data. The online visibility culling module is implemented as a specialized scene manager integrated into the OGRE engine. By linking with the view space partitioning library this module also allows the engine to use the preprocessed data: it can load the XML view cells + PVS description and use it in runtime to further speedup rendering or schedule prefetching for out-of-core or network based rendering.

2.2. VIEW SPACE PARTITIONING MODULE

2.2.1. Overview

The view space partitioning module deals with subdividing the space of all possible viewpoints and viewing directions into view cells. The result of the computation is a BSP tree describing the geometry of the view cells and the view cell hierarchy describing the view cells at different levels of detail. Note that on the contrary to common approaches the BSP tree is constructed according to the actual visibility in the scene: it aims to minimize the average rendering time for the PVS associated resulting set of view cells. The process of view space partitioning consists of three main steps:

- Visibility sampling
- View space subdivision
- View space merging

The first step estimates visibility in the scene, which allows basing the view cell construction on scene visibility without incurring the overhead of having to calculate complete visibility. We use stochastic sampling by casting rays that are uniformly distributed in the whole view space. As a result we obtain a set of maximal free line segments which we call *visibility segments*. The visibility segments provide information about scene visibility for the subsequent steps of the view cells construction.

The second step performs an adaptive hierarchical subdivision of view space. The subdivision is driven by heuristics which aim to minimize the estimated rendering cost of the resulting subdivision. The result is set of elementary view cells which satisfy certain termination criteria. We choose a very fine subdivision in order to allow more choices for the subsequent merging step.

The third step merges the elementary view cells to larger ones while minimizing the increase of the estimated rendering cost for each merging step. The merging progress is recorded in a *merge history tree*. The merge history defines a *view cell hierarchy*, allows retrieving an optimal set of view cells for a specified granularity of the view space subdivision. Additionally, this hierarchy can be used to compress the PVSs in a simple and efficient way. The merging step will implicitly approximate important visibility events in the scene. The three steps of our algorithm are illustrated in

Figure 1.

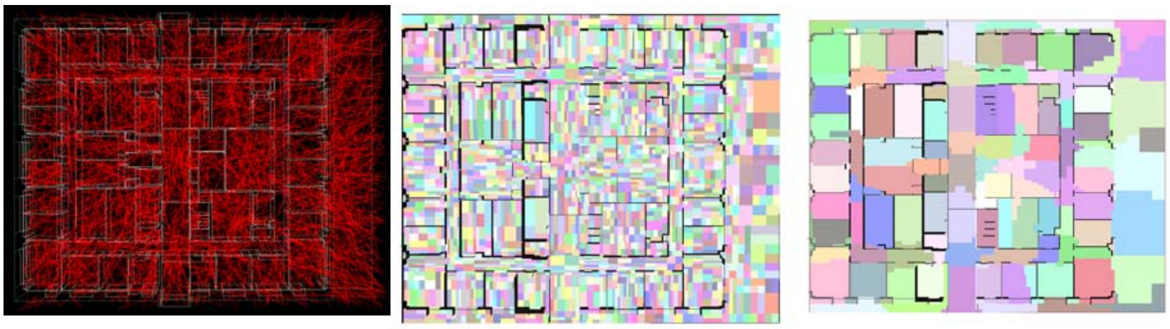


Figure 1. ***Illustration of the view space partitioning process in a building interior (soda scene). (left) Visibility sampling. (center) Subdivision into elementary view cells. (right) Merged view cells***

2.2.2. Cost Model

The view cell construction is driven by a cost model which estimates the expected rendering time based on the sampled visibility information. The cost of a given a set of view cells S is estimated from the expected value of the rendering time as follows:

$$c(\mathcal{S}) = \sum_{i \in \mathcal{S}} p(i) r(PVS_i),$$

where $r(PVS_i)$ is a rendering time estimation for the approximate PVS of cell i and p(i) is the probability of the viewpoint being located in view cell i. Assuming that viewpoints will be distributed uniformly in the whole view space, p(i) can be chosen as the ratio of the volume V_i of the given view cell and the total volume V of view space:

$$p(i) = \frac{V_i}{V}.$$

In general, the user can specify any probability density d for viewpoint locations, so that areas where the user is more likely to move receive more attention in the view cell construction (note that this feature is not yet supported by the module). p(i) is then given by:

$$p(i) = \frac{\int_{x \in i} d(x)}{\int d(x)}.$$

The rendering time for a view cell is estimated from the rendering times for the objects in the PVS:

$$r(PVS) = \sum_{o \in PVS} r(o).$$

The rendering time estimation function r(o) is difficult to establish exactly since it depends not only on the particular set of objects and their attributes, but also on the actual implementation and hardware. On the other hand, the view cell subdivision should not be tied too much to a specific hardware, neither do we have an accurate PVS to determine the absolute value of the rendering time. Therefore we propose to loosely calibrate an analytic rendering time estimation function to a small number of target machines. Since current graphics hardware is CPU limited for small batches, the following function provides good results:

$$r(o) = \max(a, bt_o, cp_o),$$

where a, b and c are positive constants, and t_o and p_o are the number of triangles and the number of projected pixels of object o (estimated from some points in the cell) respectively.

2.2.3. Representation of View Cells

We maintain the view space partition as a binary space partition tree which is constructed top-down. The leaves of this tree are convex polyhedra which form a set of elementary view cells. The final view cell partition is constructed from these elementary view cells using a bottom-up merging procedure which is recorded in the merge history tree. Note that merging is not tied to the original subdivision and therefore usually results in a completely different, more optimal tree. Both steps of the view cell hierarchy construction will be detailed in the next section. The representation of view cells using the two hierarchies is shown in Figure 2.

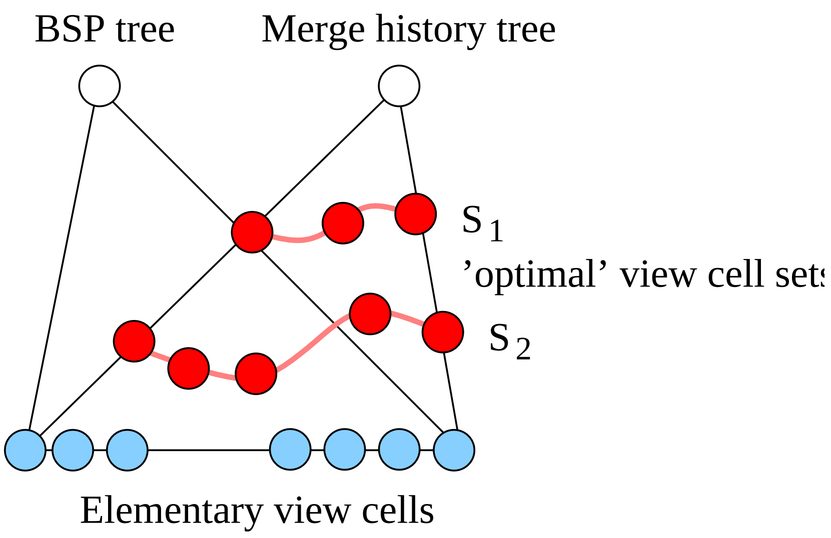


Figure 2. The view cell hierarchy is represented using a BSP tree and the merge history tree. The BSP tree provides geometrical description for elementary view cells. The merge history tree provides logical grouping of the view cells which allows extracting set of view cells with specified granularity of the subdivision. The example shows two sets S_1 and S_2 where the desired number of view cells for S_2 is larger than for S_1.

Note that we partition the view space only in the spatial domain, since the observer can quickly move through the whole directional space within a few frames by changing her viewing direction. Fortunately, culling of directional space is efficiently handled by view-frustum culling.

2.2.4. Visibility Sampling

We gain information about global visibility in the scene by sampling the whole view space. A view space sample is a 5D entity corresponding to a ray in primal space. For simplicity, let's assume that the view space is defined by a 3D spatial box of possible ray origins (*view space box*) and contains all possible ray directions.

The view space is then sampled using the following strategy:

1. Select a point p inside the view space box and a direction d using uniform distributions.
2. Cast a `forward' ray from p in direction d and a `backward' ray from p in the opposite direction -d.
3. Construct a line segment formed by the calculated termination points of the forward and backward rays. If at least one of the two rays hits an object, we call the resulting line segment a visibility segment and store it for later use.

The determination of visibility segments is illustrated in Figure 3.

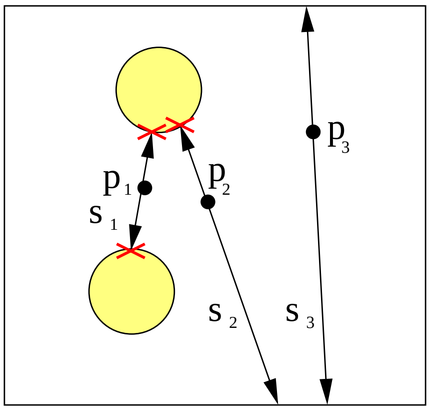


Figure 3. Illustration of determination of visibility segments. Three visibility samples are generated from points p_1, p_2 and p_3, resulting in two valid visibility segments. s_1 carries information about visibility of two objects, s_2 about one object. s_3 is not a valid visibility segment since it does not hit any object.}

There is an interesting subtlety involved in polygon orientations. For a general scene, the above procedure will create visibility segments in the interiors of objects if those regions are not explicitly exempted from view space. If, however, the input model is guaranteed to be watertight and the polygons have a consistent orientation, the algorithm can detect *empty view space* at no additional cost: if a ray hits the back side of a polygon, the ray starting point is simply shifted to the intersection point and the ray is re-cast (see Figure 4). In this way, no visibility segments will be generated in empty space. We have found that empty view space detection can improve the final view cell hierarchy, because visibility regions are more clearly separated (see the Preprocessor.detectEmptyViewSpace parameter of the module).

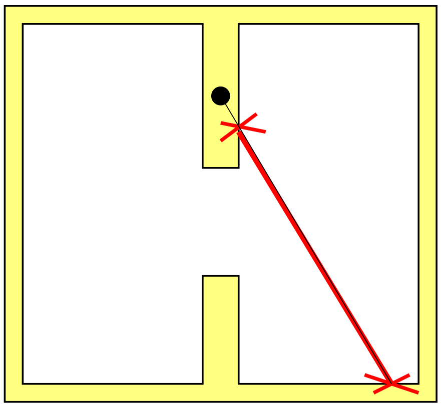


Figure 4. Example of empty view space detection. The origin of the ray lies inside the wall of the building and so the ray first hits a back-facing polygon. We shift the ray origin to the intersection point and the ray is re-cast. The resulting visibility segment is shown in red.

2.2.5. View Space Subdivision

The view space subdivision uses a top-down approach to create a set of elementary view cells. In particular we use *binary space partitioning* (BSP) maintained by a BSP tree. Starting with a single view cell corresponding to the whole view space, we recursively subdivide the current view cell using either axis-aligned planes or planes aligned with scene geometry. Each cell of the subdivision also references all visibility segments that intersect it.

The BSP construction uses a greedy optimization for the best-next split. Note that in contrast to most previous work on BSP or kD-tree construction we use global optimization procedure with a priority queue for selecting the splitting plane candidates. An entry in the priority queue consists of a reference to leaf node, the best splitting plane candidate inside this leaf, and a cost reduction which would be achieved when splitting the leaf by the plane. For each step of the subdivision we select the node for which its best splitting plane provides the highest global render cost reduction. The subdivision is thus refined progressively and regions with the highest potential render cost decrease are subdivided first.

The best splitting plane for a node is established as follows: we generate axis-aligned splitting plane candidates aligned with the endpoints of rays intersecting the node. Additionally, we generate planes aligned with the geometry contained in that node (if any). For each candidate, we calculate the reduction of the expected rendering cost c(S) that would result from subdividing the node by that plane. This is done by partitioning the current set of visibility segments according to the plane (note that a segment can be assigned to both sets), computing the PVSs (i.e., the union of the objects hit by the visibility segments) for the front and back segment sets, and using those to evaluate the reduction of the cost. Finally, we choose the candidate plane that provides the most reduction in expected rendering cost and put a corresponding entry in the priority queue.

The subdivision is terminated when one of the following termination
criteria is met:

- A specified maximum number of elementary view cells have been generated. It ensures that the algorithm stays within reasonable memory bounds.
- The cost reduction for the best splitting plane is below a specified threshold. As the cost reduction can temporarily stagnate, we only terminate when the reduction was below the threshold in several successive subdivision steps in that branch of the tree.

Due to the priority driven subdivision the view space will be evenly subdivided no matter of the termination point. This is not the case for depth-first approaches, where for example a termination on low memory would leave whole view space regions unsubdivided. Additionally, the local termination criteria used in the depth-first approach are hard to tune.

Note that the time required for the subdivision is dominated by the cost evaluation for the candidate planes. To accelerate this process, we limit the number of axis-aligned as well as geometry-aligned candidates. If the number of visibility segments or geometry planes is above these limits we select their random subsets. This speeds up the selection especially for nodes near the root of the subdivision tree.

2.2.6. View Space Merging

View space merging is a bottom-up process which aims to reduce the number of view cells while minimizing the cost of the merged view cell set. We use a greedy algorithm that always merges the pair of view cells resulting in the minimal cost increase. This is done by maintaining a priority queue of view cell merge candidates. Each pair of neighboring view cells forms a merge candidate. The relative cost increase due to the merge candidate consisting of view cells x and y is given as:

$$r(x,y) = \frac{c_r(\mathcal{S}_{merged})}{c_r(\mathcal{S})}$$

where S is the current set of view cells and S_{merged} is the set resulting from merging x and y. The priority of the merge candidate p(x,y) is given by:

$$p(x,y) = \frac{1}{r(x,y)} = \frac{c_r(\mathcal{S})}{c_r(\mathcal{S}_{merged})}$$

In the beginning, the queue is initialized with all pairs of neighboring view cells. At every step, we select the merge candidate with the highest priority (smallest relative cost increase) and merge the associated view cells. The PVS and the estimated rendering cost of the new view cell is calculated. After the merge, the priority queue is updated by removing entries corresponding to the merged view cells and inserting new entries corresponding to the created view cell and its neighbors. Note that the set of neighbors for a view cell is determined using the BSP tree.

The merging process provides a sequence of view cell sets: at every merge step we obtain a new set of view cells with exactly one view cell less than in the previous set. We record the whole merging process in a *merge history tree*. The leafs of this tree correspond to elementary view cells. Every internal node corresponds to a merged view cell. With each internal node we associate the current cost of the subdivision resulting from the corresponding merge.

Once the complete merge history tree is built, there are several ways how to create a view cell partition from the tree. The easiest way is to specify a target number n of view cells and extract these from the tree. These view cells can then be used as input for a visibility preprocessing algorithm.

2.2.7. Extracting View Cells

The view cell hierarchy allows to extract the set of view cells most suitable for the target application. Bellow we discuss three possibilities of view cell extraction.

Getting a specified number of view cells. Often, it is most convenient to specify a desired number of view cells to use for visibility calculation. This limits both the preprocessing time and the storage required for PVS data, as well as restricting the frequency with which new PVS data has to be fetched due to crossing into a new view cell at runtime. To obtain a given number of view cells, we perform a priority traversal of the merge history tree. The priority of a node is given by the cost associated with the node. When reaching a leaf node, we add it to the list of resulting view cells. When the sum of traversed leaves and the nodes in the priority queue becomes equal to the desired number of view cells, we terminate the traversal and add the contents of the priority queue to the resulting view cells. The collected view cells form a cut of the merge history tree at optimal depths with respect to the specified granularity.

Fulfilling a given memory budget. The procedure described above can be extended to allow specifying an approximate memory budget for the complete PVS representation (it is only approximate because it relies on the approximate PVS from the sampling step). At every step of the tree traversal, we can easily calculate the memory requirements for the current set of view cells and their approximate PVSs. We can terminate the traversal when the budget is reached and collect the resulting view cells as described above. Note, that this step can also be applied after computing the final visibility classification. In this case the memory budget would be the real budget for storing the PVS representation.

Extracting important view cells. An alternative to the methods described above is view cell extraction based on their estimated rendering cost. In particular we can use the maximal tolerance of the increase of the estimated rendering cost over the minimal cost, i.e. the cost provided by the densest view space partition (elementary view cells). The view cells are extracted again by a priority traversal of the merge history tree. At each step we evaluate the ratio of the current cost (stored with the processed node) and the minimal cost. If the ratio falls below a threshold, we terminate the traversal and collect the resulting view cells as described above.

Interactive specification. In practice, the user can combine these these methods in an interactive setup. The selection process can easily be accomplished with a real-time visualization of the view cells and a depiction of the associated cost and memory budgets, as well as the just described cost ratio. Typically, the user would start by setting an initial cost ratio and refining the result interactively. This allows the user interactive control over the process, which is a feature often desired by practitioners.

2.2.8. Using the View Space Partitioning module

The view space partitioning is implemented as a standalone application. Alternatively the module can be compiled as a library which can be linked to a 3rd party code. An example of such an application is the online occlusion culling module (integrated with OGRE engine) which links the library in order to load the preprocessed data and use them in runtime.

The module uses a number of parameters which can be specified by the user through the "environment file" or command line parameters. The parameters can have one of the following types: string, int, float, and boolean. All parameters have implicit values set by the preprocessor (see the Environment.cpp source file). This value can be redefined by the environment file (see for example preprocess_visibility.env). This value can further be overwritten by a command line argument. There are two possibilieties how to specify the command parameter: using a shortcut or define construct. The define construct uses the following syntax: -Dparameter_name=value. With the exception of Boolean parameters the shortcut version is specified as follows: -parameter_shorcut=value. The boolean parameter shortcut is directly followed by + or - sign (e.g. -preprocessor_use_gl_render+). The usage of the parameters is illustrated in the example scripts and environment files (preprocess_visibility, preprocess_visibility.env, generate_viewcells, generate_viewcells.env).

Bellow we give a list of the most important parameters. For other parameters see the environment file (generate_viewcells.env) and the documentation included in the source code of the module.

Scene.filename (-scene_filename=) [string]

Scene description file. Currently simplified X3D (.x3d), Unigraphics (.dat), UNC (.ply) formats are supported. This option has to be specified!

Preprocessor.type (-preprocessor=) [string, one of: vss, rss, exact, sampling, render]

Type of the preprocessor to use. Currently only the "rss" type is supported for PVS computation (rss stands for Ray Space Subdivision explained in the previous chapters of the documentation). For view cells construction the "vss" preprocessor should be used (vss stands for View Space Subdivision).

Preprocessor.useGlRenderer (-preprocessor_use_gl_renderer) [boolean]

Tells the preprocessor to open an OpenGL window which serves for visualization, testing and visual debugging of the preprocessor. This functionality is currently implemented using Qt OpenGL widget.

ViewCells.type (-view_cells_type=) [string, one of: vspBspTree, ...]

Type of view cells to be generated. Use "vspBspTree", which corresponds to our optimized algorithm.

VspBsp.Termination.maxViewCells (-vsp_bsp_term_max_view_cells=) [int]

Maximal number of generated view cells by the subdivision

VspBsp.Construction.samples (-vsp_bsp_construction_samples=) [int]

Number of visibility samples to be used for the construction of the view cell subdivision.

VspBsp.Termination.minGlobalCostRatio (-vsp_bsp_term_min_global_cost_ratio=) [float]

Defines the minimal render cost decrease where the view cell subdivision is terminated

VspBsp.maxPolyCandidates (-vsp_bsp_max_poly_candidates=) [int]

Number of geometry aligned splits evaluated for the next best split.

VspBsp.useCostHeuristics (-vsp_bsp_use_cost_heuristics=) [bool]

If spatial mid split is always taken for axis aligned splits or split plane is evaluated using the cost model.

ViewCells.PostProcess.merge (-view_cells_post_process_merge=) [bool]

If the final subdivision should be merged according to the cost model to optimize the view cells.

ViewCells.Construction.samples (-view_cells_construction_samples=) [int]

The number of samples cast before the final merge step (the subdivision is handled by VspBsp.Construction.samples)

2.3. PVS COMPUTATION MODULE

This task deals with calculating visibility in a preprocess. It determines a potentially visible set (PVS) of objects for every view cell. In runtime the view cell containing the current viewpoint is located and only the objects contained in the associated PVS have to be rendered.

This concept is straightforward and it is known from early 90s. The major problem is the difficulty of determining a PVS for view cells in large scene of arbitrary type. This problem has been addressed by numerous papers dealing with computing from-region visibility. The previous research addressed mostly conservative algorithms, with few exceptions dealing with exact visibility. Currently there is plenty of conservative methods and several exact methods. However, the conservative methods are limited to a particular scene type (indoor, 2.5D, large polygons) whereas the exact methods are computationally costly, difficult to implement and prone to numerical errors.

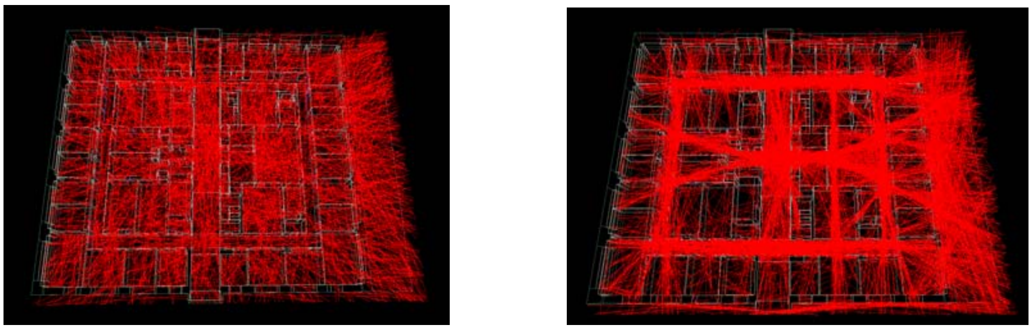


Figure 5. (left) Uniform visibility sampling. (right) Importance sampling driven by the 5D ray space subdivision.

Our module provides a visibility preprocessing framework which redefines the classical preprocessing scheme. Instead of solving visibility for individual view cells we progressively compute global visibility, i.e. visibility for all view cells. A coarse global visibility solution is obtained within few seconds or minutes. As the time passes more visible objects are discovered. The summary of the differences of our framework to commonly used techniques is given in the following table:

Common methods	**Our solution**
Sequential computation	Progressive computation
Coherence of visibility between view cells not exploited	Coherence exploited by computing true 'global visibility' (for all view cells at once)
Not usable for very large scenes	Very large scenes

(work on triangle level)	(works on the object level)
Scene type dependent	Arbitrary scenes (indoor + outdoor)
Complicated implementations	Relatively simple implementation: (ray casting, alternatively HW rendering)
Binary visibility only	Integral visibility estimate

Table 1. Differences of common visibility preprocessing methods to the implemented solution.

The module uses progressive global visibility computation, which consists of initial visibility sampling and further refinement using a combination of stratified and importance sampling. The importance sampling aims to place more samples at places of visibility changes, identifying of which can make a contribution to the currently computed PVSs.

The algorithm uses adaptive subdivision of ray space maintained by a 5D kD-tree. The subdivision is driven by homogeneity of visibility in the corresponding ray space cells. The sampling density is then varied depending on the estimated visibility contributions of rays cast for different cells of the subdivision. The algorithm works in stages, where the samples cast at previous stages drive the sampling for the next stages.

2.3.1. Algorithm outline

The algorithm implemented in the PVS computation module builds on the following ideas:

- Cast rays from object surfaces to possible viewpoints and obtain maximal free ray segments.
- Compute visibility contributions of the ray segments to the view cells pierced by the segment.
- Organize the sampling domain using adaptive 5D ray space subdivision.
- Cast more rays to places with higher estimated contributions.

The first two points shift the classical computation of visibility from the view cells into computing visibility from the objects: we cast rays from the objects and determine which view cells can see the object. This computation is performed easily by determining which view cells intersect the ray before the ray is terminated or leaves the scene. The major benefit of this approach is that a single ray can contribute to many view cells.

The second two points deal adaptive with sampling. The 5D sampling domain is too large to quickly capture important rays by regular sampling. Therefore we use adaptive hierarchical subdivision of the relevant part of ray space. The subdivision is a tool for providing spatial and directional focus for the computation depending on the estimated contribution in that region of ray space.

2.3.2. Evaluating sample contributions

Once the maximal free line segments are determined their contributions to the view cells are evaluated by computing intersections of the view cells with these segments. Since the view cells are organized in a BSP tree (as described in the previous chapter), we can easily determine which view cells are intersected by each line segment. This process is implemented by a simple traversal through the view cells BSP tree and the view cell hierarchy. Then for each view cell we compute a relative contribution of the sample r as 1/(n(t(r)) + 1), where n(t(r)) is the number of samples with the same termination object t(r) already associated with the view cell. Thus if we discover a new object for the view cell's PVS it will provide us a relative contribution of 1. For further samples associated with the same object their contribution is reduced.

2.3.3. Ray Space Subdivision

Ray space subdivision aims to serve as a tool for estimating the contribution of casting a particular set of rays. We use a 5D-tree similar to the tree used by Arvo and Kirk for speeding up ray object intersection computation. However our approach differs not only in the purpose of the tree, but also in the way how the tree is constructed. The 5D subdivision is maintained by a kD-tree. The tree is constructed in order to provide a good separation of estimated ray contributions. The root of the tree is associated with a spatial bounding box of the scene and directional bounding box of the view space.

Each leaf of the tree is then associated with an axis aligned 3D box and a 2D directional rectangle. Whenever we decide to split a leaf node of the tree, we evaluate the benefit of the split using all of the 5 axes (3 spatial + 2 directional). The benefit of the split is measured by balancing the number of objects associated with rays in the front and back cells with the volumes of the cells.

The estimated contribution of the rays in a leaf of the RSS tree is evaluated as a sum of contributions of rays associated with the leaf. We use a temporal filter which gives higher weight to rays from the recent passes.

2.3.4. Generating sample rays

Based on the contribution estimation heuristics we decide how many rays to generate for the 5D cell corresponding to a leaf of the RSS tree: the number of rays generated per leaf is proportional to the estimated contribution of rays in that leaf.

For every generated ray we first compute its intersection with spatial box of the 5D cell in order to eliminate undesired intersections inside the box. Such a ray is cast through the spatial data structure as described in Section 2.3.5.

For generating the new sample rays we use a combination of the following methods:

- **Locally uniform sampling.**
  The locally uniform sampling uniformly distributes new samples across the given 5D cell. We use a 5D halton sequence to generate 5D points which represent the origin and the direction of the sample ray.
- **Silhouette sampling.** Silhouette based sampling aims to place new samples along the silhouettes of the objects associated with the 5D cell. We use a rejection sampling in order to discart rays which cannot be silhouette rays. The objects in the leaf are sorted based on the probability that they get hit by a ray. This probability is derived from the relative number of ray samples stored in the cell which are associated with each object.

2.3.5. Casting rays

We compute the first intersection of the ray with a scene object. If there is no such intersection we compute an intersection with a bounding volume of the view space. Then we recalcute the origin of the ray in order to obtain a maximal free line segment corresponding to the ray. The new origin is computed by casting the ray from its initial origin backwards.

Currently the rays are cast using a kD-tree and an unoptimized software ray caster. As a further development of the module we want to optimize the current implementation and also to implement a HW accelerated variant of ray casting.

2.3.6. Filtering visibility

In order to reduce undersampling errors we filter the computed visibility information. We use two different filters: view space filter and spatial filter. The view space filter merges the PVSs of neighboring view cells. The spatial filter propagates visibility to objects in proximity of visible objects. Currently these filters are box filters and have to be specified as a ratio to bounding box the scene (see Preprocessor.visibilityFilterWidth parameter further in the text).

2.3.7. Implementation

We have implemented the PVS computation module as a multithreaded application . Initially the scene and the view cells are loaded into memory and the initial set of rays is cast. Casting the initial ray set is performed on the CPU and takes a few seconds. After propagating the ray visibility contributions to the view cells the scene can already be rendered using the view cell based PVS. At this stage visible errors appear due to undersampling however the most of the visible objects is already rendered correctly. The user can freely walk in the scene and while the visibility processing thread updates the PVSs. Depending on the scene it takes few tens of seconds to few minutes till the PVSs capture enough objects so that most frames contain no visible errors.

2.3.8. Using the PVS computation module

The PVS computation module is implemented as a standalone application. The parameters for the module can be specified by the user through the "environment file" or on the command line as described for the view space partitioning module. Bellow we give a list of the most important parameters. For other parameters see the environment file (preprocess_visibility.env) and the documentation included in the source code of the module.

Scene.filename (-scene_filename=) [string]

Scene description file. Currently simplified X3D (.x3d), Unigraphics (.dat), UNC (.ply) formats are supported. This option has to be specified!

Preprocessor.type (-preprocessor=) [string, one of: vss, rss, exact, sampling, render]

Type of the preprocessor to use. Currently only the "rss" type is supported for PVS computation (rss stands for Ray Space Subdivision explained in the previous chapters of the documentation). For view cells construction the "vss" preprocessor should be used (vss stands for View Space Subdivision).

Preprocessor.useGlRenderer (-preprocessor_use_gl_renderer) [boolean]

Tells the preprocessor to open an OpenGL window which serves for visualization, testing and visual debugging of the preprocessor. This functionality is currently implemented using Qt OpenGL widget and thus requires compiling the preprocessor with Qt library.

Preprocessor.detectEmptyViewSpace (-preprocessor_detect_empty_viewspace=) [boolean]

Empty view space detection allows more efficient sampling for scenes with properly modelled watertight objects (those which can be correctly rendered with backface culling on).

Preprocessor.applyVisibilityFilter (-preprocessor_apply_visibility_filter) [boolean]

Tells the preprocessor if the visibility filter should be applied.

Preprocessor.visibilityFilterWidth (-preprocessor_visibility_filter_width=) [float]

Width of the visibility filter specified as a ratio of the size of the bounding box of the scene. A reasonable value for common scenes is 0.01.

Preprocessor.visibilityFile (-preprocessor_visibility_file=) [string]

Name of the file into which the preprocessed visibility information should be exported.

RssPreprocessor.initialSamples (-initial_samples=) [int]

Number of initial samples to use. These samples are distributed uniformly in the whole view space.

RssPreprocessor.vssSamples (-rss_vss_samples=) [int]

Number of samples cast using importance and stratified sampling based on RSS.

RssPreprocessor.useImportanceSampling (-rss_use_importance) [bool]

Tells the preprocessor whether to use RSS based sampling at all. If false then even the RssPreprocessor.vssSamples are cast using the uniform distribution.

RssPreprocessor.vssSamplesPerPass (-rss_vss_samples_per_pass=) [int]

Number of samples cast per pass of the RSS algorithm. After each pass the RSS tree is updated and the new set of rays is generated according to the estimated sample contributions.

2.4. ONLINE VISIBILITY CULLING MODULE

This module works in runtime to quickly cull occluded objects given a particular viewpoint and a viewing direction. We have designed and implemented an online culling algorithm which efficiently uses occlusion queries supported by the recent graphics hardware. Large regions of similar visibility are culled early by using a hierarchy. To eliminate CPU stalls and GPU starvation, we exploit temporal coherence and schedule the occlusion queries using a priority queue. To reduce unnecessary queries, we pull up and push down visibility information in the hierarchy.

This modules provides a prototype implementation of online culling in the OGRE engine. The integrated algorithm is reusable with most plugins and hierarchies without major changes, and it is flexible enough so that it’s applicable in both a rendering and a query mode.

2.4.1. Hardware Occlusion Queries

Hardware occlusion queries follow a simple pattern: To test visibility of an occludee, we send its bounding volume to the GPU. The volume is rasterized and its fragments are compared to the current contents of the z-buffer. The GPU then returns the number of visible fragments. If there is no visible fragment, the occludee is invisible and it need not be rendered. There are several advantages of hardware occlusion queries:

- Generality of occluders. We can use the original scene geometry as occluders, since the queries use the current contents of the z-buffer.
- Occluder fusion. The occluders are merged in the z-buffer, so the queries automatically account for occluder fusion. Additionally this fusion comes for free since we use the intermediate result of the rendering itself.
- Generality of occludees. We can use complex occludees. Anything that can be rasterized quickly is suitable.
- Exploiting the GPU power. The queries take full advantage of the high fill rates and internal parallelism provided by modern GPUs.
- Simple use. Hardware occlusion queries can be easily integrated into a rendering algorithm. They provide a powerful tool to minimize the implementation effort, especially when compared to CPU-based occlusion culling.

2.4.2. Algorithm Overview

Our method is based on exploiting temporal coherence of visibility classification. In particular, it is centered on the following three ideas:

- We initiate occlusion queries on nodes of the hierarchy where the traversal terminated in the last frame. Thus we avoid queries on all previously visible interior nodes.

- We assume that a previously visible leaf node remains visible and render the associated geometry without waiting for the result of the corresponding occlusion query.

- Issued occlusion queries are stored in a query queue until they are known to be carried out by the GPU. This allows interleaving the queries with the rendering of visible geometry.

The algorithm performs a traversal of the hierarchy that is terminated either at leaf nodes or nodes that are classified as invisible. Let us call such nodes the termination nodes, and interior nodes that have been classified visible the opened nodes. We denote sets of termination and opened nodes in the i-th frame Ti and Oi, respectively. In the i-th frame, we traverse the kD-tree in a front-to-back order, skip all nodes of Oi−1 and apply occlusion queries first on the termination nodes Ti−1. When reaching a termination node, the algorithm proceeds as follows:

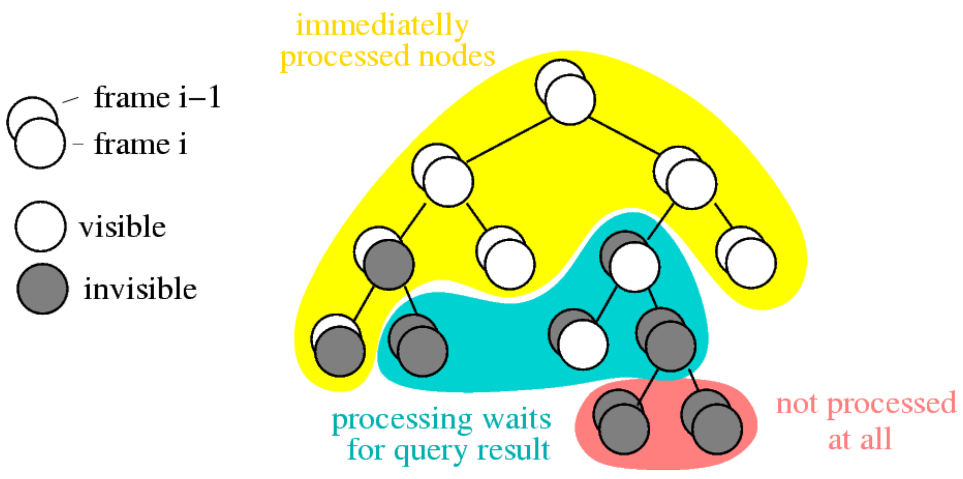
- For a previously visible node (this must be a leaf), we issue the occlusion query and store it in the query queue. Then we immediately render the associated geometry without waiting for the result of the query.

- For a previously invisible node, we issue the query and store it in the query queue.

When the query queue is not empty, we check if the result of the oldest query in the queue is already available. If the query result is not available, we continue by recursively processing other nodes of the kD-tree as described above. If the query result is available, we fetch the result and remove the node from the query queue. If the node is visible, we process its children recursively. Otherwise, the whole subtree of the node is invisible and thus it is culled.

In order to propagate changes in visibility upwards in the hierarchy, the visibility classification is pulled up according to the following rule: An interior node is invisible only if all its children have been classified invisible. Otherwise, it remains visible and thus opened. An example of the behavior of the method on a small kD-tree for two subsequent frames is depicted Figure 6. The pseudo-code of the complete algorithm is given in Figure 10.

The sets of opened nodes and termination nodes need not be maintained explicitly. Instead, these sets can be easily identified by associating with each node an information about its visibility and an id of the last frame when it was visited. The node is an opened node if it is an interior visible node that was visited in the last frame (line 23 in the pseudocode). Note that in the actual implementation of the pull up we can set all visited nodes to invisible by default and then pull up any changes from invisible to visible (lines 25 and line 12). This modification eliminates checking children for invisibility during the pull up.


immediatelly processed nodes
frame i−1
frame i
visible
invisible
processing waits for query result
not processed at all

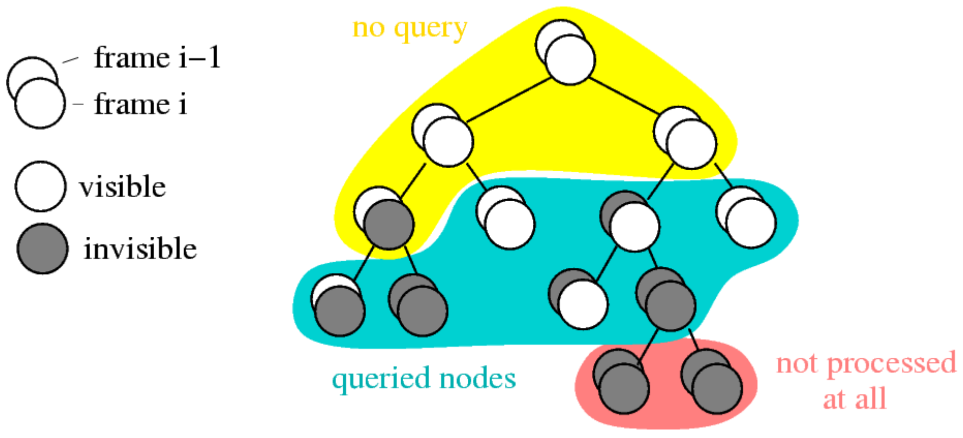


Figure 6. (top) Illustration of processing requirements of nodes of the spatial hierarchy at frame i. (bottom) Illustration of query requirements of nodes of the spatial hierarchy.

2.4.3. Reduction of the number of queries

Our method reduces the number of visibility queries in two ways: Firstly, as other hierarchical culling methods we consider only a subtree of the whole hierarchy (opened nodes + termination nodes). Secondly, by avoiding queries on opened nodes we eliminate part of the overhead of identification of this subtree. These reductions reflect the following coherence properties of scene visibility:

- Spatial coherence. The invisible termination nodes approximate the occluded part of the scene with the smallest number of nodes with respect to the given hierarchy, i.e., each invisible termination node has a visible parent. This induces an adaptive spatial subdivision that reflects spatial coherence of visibility, more precisely the coherence of occluded regions. The adaptive nature of the subdivision allows to minimize the number of subsequent occlusion queries by applying the queries on the largest spatial regions that are expected to remain occluded.

- Temporal coherence. If visibility remains constant the set of termination nodes needs no adaptation. If an occluded node becomes visible we recursively process its children (pull-down). If a visible node becomes occluded we propagate the change higher in the hierarchy (pull-up). A pull-down reflects a spatial growing of visible regions. Similarly, a pull-up reflects a spatial growing of occluded regions. By avoiding queries on the opened nodes, we can save 1/k of the queries for a hierarchy with branching factor k (assuming visibility remains constant). Thus for the kD-tree, up to half of the queries can be saved. The actual savings in the total query time are even larger: the higher we are at the hierarchy, the larger boxes we would have to check for occlusion. Consequently, the higher is the fill rate that would have been required to rasterize the boxes. In particular, assuming that the sum of the screen space projected area for nodes at each level of the kD-tree is equal and the opened nodes form a complete binary subtree of depth d, the fill rate is reduced (d + 2) times.

2.4.4. Reduction of CPU stalls and GPU starvation

The reduction of CPU stalls and GPU starvation is achieved by interleaving occlusion queries with the rendering of visible geometry. The immediate rendering of previously visible termination nodes and the subsequent issuing of occlusion queries eliminates the requirement of waiting for the query result during the processing of the initial depth layers containing previously visible nodes. In an optimal case, new query results become available in between and thus we completely eliminate CPU stalls. In a static scenario, we achieve exactly the same visibility classification as the hierarchical stop-and-wait method.

If the visibility is changing, the situation can be different: if the results of the queries arrive too late, it is possible that we initiated an occlusion query on a previously occluded node A that is in fact occluded by another previously occluded node B that became visible. If B is still in the query queue, we do not capture a possible occlusion of A by B since the geometry associated with B has not yet been rendered. In Section 3.6 we show that the increase of the number of rendered objects compared to the stop-and-wait method is usually very small.

2.4.5. Further Optimizations

This section discusses a couple of optimizations of our method that can further improve the overall rendering performance. In contrast to the basic algorithm from the previous section, these optimizations rely on some user specified parameters that should be tuned for a particular scene and hardware configuration.

Conservative visibility testing

The first optimization addresses the reduction of the number of visibility tests at the cost of a possible increase in the number of rendered objects. This optimization is based on the idea of skipping some occlusion tests of visible nodes. We assume that whenever a node becomes visible, it remains visible for a number of frames. Within the given number of frames we avoid issuing occlusion queries and simply assume the node remains visible. This technique can significantly reduce the number of visibility tests applied on visible nodes of the hierarchy. Especially in the case of sparsely occluded scenes, there is a large number of visible nodes being tested, which does not provide any benefit since most of them remain visible. On the other hand, we do not immediately capture all changes from visibility to invisibility, and thus we may render objects that have already become invisible from the moment when the last occlusion test was issued.

In the simplest case, the number of frames a node is assumed visible can be a predefined constant. In a more complicated scenario this number should be influenced by the history of the success of occlusion queries and/or the current speed of camera movement.

Approximate visibility

The algorithm as presented computes a conservative visibility classification with respect to the resolution of the z-buffer. We can easily modify the algorithm to cull nodes more aggressively in cases when a small part of the node is visible. We compare the number of visible pixels returned by the occlusion query with a user specified constant and cull the node if this number drops below this constant.

Complete elimination of CPU stalls

The basic algorithm eliminates CPU stalls unless the traversal stack is empty. If there is no node to traverse in the traversal stack and the result of the oldest query in the query queue is still not available, it stalls the CPU by waiting for the query result. To completely eliminate the CPU stalls, we can speculatively render some nodes with undecided visibility. In particular, we select a node from the query queue and render the geometry associated with the node (or the whole subtree if it is an interior node). The node is marked as rendered but the associated occlusion query is kept in the queue to fetch its result later. If we are unlucky and the node remains invisible, the effort of rendering the node's geometry is wasted. On the other hand, if the node has become visible, we have used the time slot before the next query arrives in an optimal manner.

To avoid the problem of spending more time on rendering invisible nodes than would be spent by waiting for the result of the query, we select a node with the lowest estimated rendering cost and compare this cost with a user specified constant. If the cost is larger than the constant we conclude that it is too risky to render the node and wait till the result of the query becomes available.

Initial depth pass

To achieve maximal performance on modern GPU's, one has to take care of a number of issues. First, it is very important to reduce material switching. Thus modern rendering engines sort the objects (or patches) by materials in order to eliminate the material switching as good as possible. Next, materials can be very costly, sometimes complicated shaders have to be evaluated several times per batch. Hence it should be avoided to render the full material for fragments which eventually turn out to be occluded. This can be achieved by rendering an initial depth pass (i.e., enabling only depth write to fill the depth buffer). Afterwards the geometry is rendered again, this time with full shading. Because the depth buffer is already established, invisible fragments will be discarded before any shading is done calculated. This approach can be naturally adapted for use with the CHC algorithm. Only an initial depth pass is rendered in front-to-back order using the CHC algorithm. The initial pass is sufficient to fill the depth buffer and determine the visible geometry. Then only the visible geometry is rendered again, exploiting the full optimization and material sorting capability of the rendering engine. If the materials require several rendering passes, we can use a variant of the depth pass method. We render only the first passes using the algorithm (e.g., the solid passes), determining the visibility of the patches, and render all the other passes afterwards. This approach can be used when there are passes which require a special kind of sorting to be rendered correctly (e.g., transparent passes, shadow passes). In Figure 7, we can see that artifacts occur in the left image if the transparent passes are not rendered in the correct order after applying the hierarchical algorithm (right image). In a similar fashion, we are able to handle shadows as shown in Figure 8.

Batching multiple queries

When occlusion queries are rendered interleaved with geometry, there is always a state change involved. To reduce state changes, it is beneficial not to execute one query at a time, but multiple queries at once. Instead of immediately executing a query for a node when we fetch it from the traversal stack, we add it to the pending queue. If n of these queries are accumulated in the queue, we can execute them at once. To obtain an optimal value for n, several some heuristics can be applied, e.g., a fraction of the number of queries issued in the last frame.

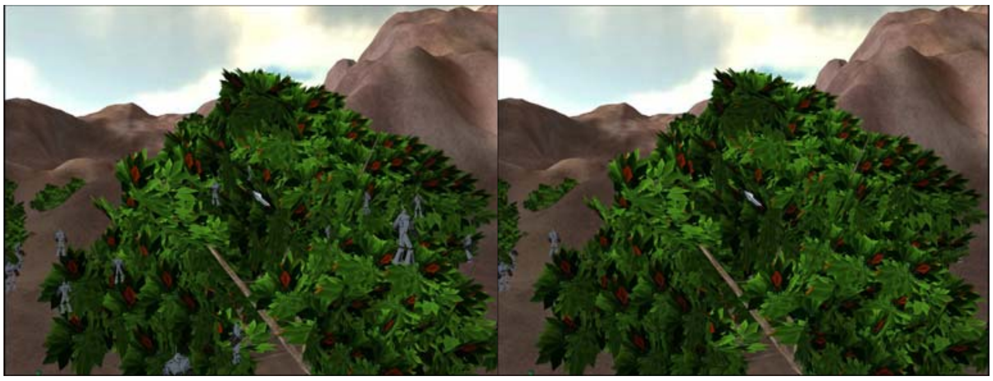


Figure 7. (left) all passes are rendered with CHC. Note that the soldiers are visible through the tree. (right) Only the solid passes are rendered using CHC, afterwards the transparent passes.

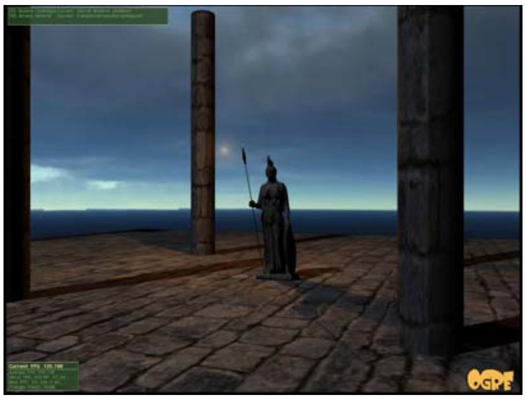


Figure 8. The online culling module correctly handles shadow volumes.

2.4.6. Implementation

The major focus in the implementation of the online culling module was a careful separation of the CHC algorithm from the platform dependent code, which allows to reuse the same algorithm code for both Ogre3D and Shark3d game engines.

The main class of the module is the CullingManager . The particular algorithm is realized as class which inherits from CullingManager and implements the RenderScene method. The CHC algorithm is implemented by the CoherentCullingManager class. The class diagram of the integration into Ogre is given in Figure 9.

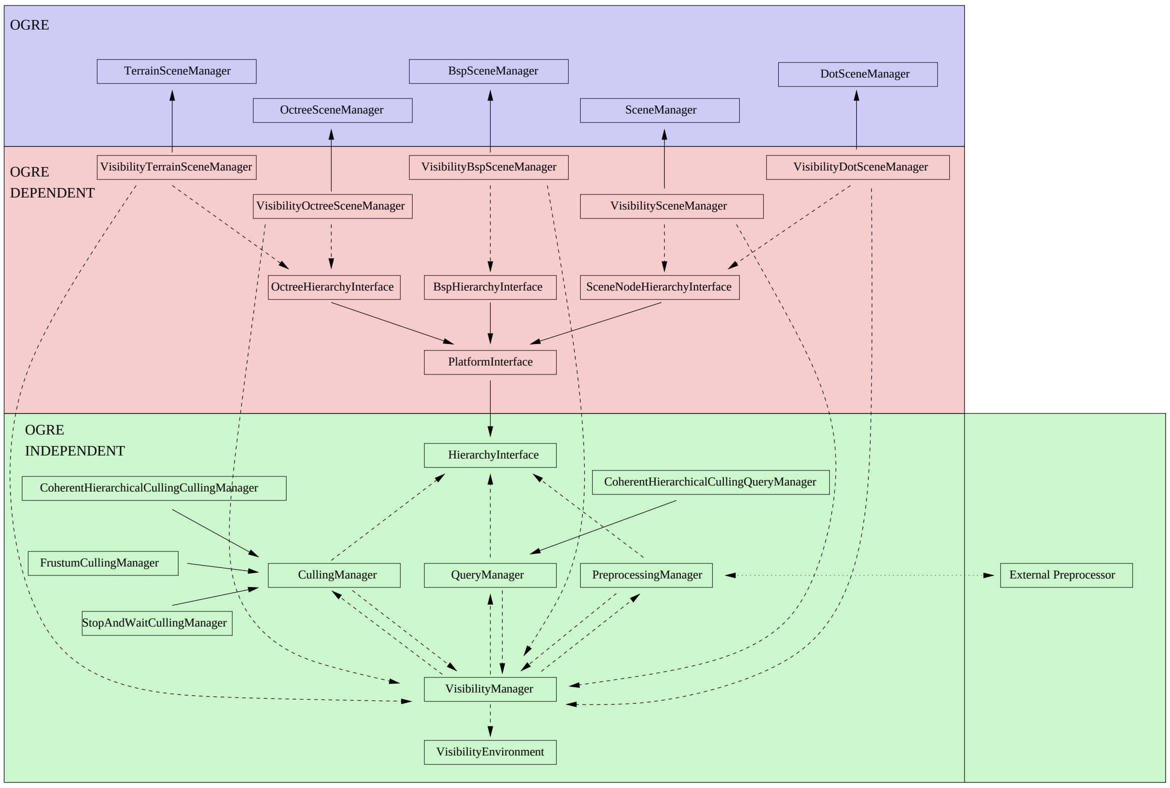


Figure 9. Class diagram depicting the integration of the online culling module into Ogre engine.

The Ogre3D implementation was realized using a specialized scene manager plugin. We implemented the single pass version of the algorithm as well as the initial depth pass method. The algorithm runs nearly orthogonal to other Ogre3D features and supports Ogre texture shadow as well as shadow volumes. The second functionality of the online module is the integration of the preprocessed visibility: it provides loading and using the preprocessed view cells in Ogre. The view cells are loaded from a XML file which is computed by the PVS computation module. The scene objects corresponding to the PVS for the current view point are set to visible while all other objects are set invisible and thus not rendered.

To keep the algorithm independent from the platform, the algorithm builds on the HierarchyInterface abstract class. This class has to be implemented for each supported platform. The most important methods of the HierarchyInterface class are:

OcclusionQuery IssueNodeOcclusionQuery (HierarchyNode ** node, *const bool* wasVisible*)*

Issues a occlusion query for this node and returns the query. wasVisible indicates if the node was visible in the last frame.

*void TraverseNode (HierarchyNode ** node *)*

Traverses and renders the given hierarchy node.

*void RenderNode (HierarchyNode ** node *)*

Renders the given hierarchy node.

*void PullUpVisibility (HierarchyNode ** node *) const*

Pulls the visibility classification up the tree.

*void SetNodeVisible (HierarchyNode ** node, *const bool* visible *) const*

Sets the node to be visible / invisible in the current frame.

*void SetLastVisited (HierarchyNode ** node, *const unsigned int* frameId *)*

The flag stores when the node was visited the last time.

*bool IsLeaf (HierarchyNode ** node *) const*

Returns true if the node is a leaf, false if the node is an interior node of the hierarchy.

DistanceQueue GetQueue ( )*

Returns pointer to the priority queue storing the nodes based on the distance to the view point in a front-to-back order.

The pseudocode of the CHC algorithm is given at Figure 10.

```
TraversalStack.Push(hierarchy.Root);
while ( not TraversalStack.Empty() or
        not QueryQueue.Empty() )
{
  //-- PART 1: process finished occlusion queries
  while ( not QueryQueue.Empty() and
         (ResultAvailable(QueryQueue.Front()) or
          TraversalStack.Empty()) )
  {
    node = QueryQueue.Dequeue();
    // wait if result not available
    visiblePixels = GetOcclusionQueryResult(node);
    if ( visiblePixels > VisibilityThreshold )
    {
      PullUpVisibility(node);
      TraverseNode(node);
    }
  }

  //-- PART 2: hierarchical traversal
  if ( not TraversalStack.Empty() )
  {
    node = TraversalStack.Pop();
    if ( InsideViewFrustum(node) )
    {
      // identify previously visible nodes
      wasVisible = node.visible and
                   (node.lastVisited == frameID - 1);
      // identify nodes that we cannot skip queries for
      leafOrWasInvisible = not wasVisible or IsLeaf(node);
      // reset node's visibility classification
      node.visible = false;
      // update node's visited flag
      node.lastVisited = frameID;
      // skip testing previously visible interior nodes
      if ( leafOrWasInvisible )
      {
        IssueOcclusionQuery(node);
        QueryQueue.Enqueue(node);
      }
      // always traverse a node if it was visible
      if ( wasVisible )
        TraverseNode(node);
    }
  }
}

TraverseNode(node)
{
  if ( IsLeaf(node) )
    Render(node);
  else
    TraversalStack.PushChildren(node);
}

PullUpVisibility(node)
{
  while (!node.visible)
  {
    node.visible = true;
    node = node.parent;
  }
}
```

Figure 10. **CHC algorithm pseudocode.**