Implementing Software Occlusion Culling for Real-Time Applications

Matias N. Leone, Leandro R. Barbagallo, Mariano M. Banquiero, Diego Agromayor, Andres Bursztyn

Proyecto de Investigación "Explotación de GPUs y Gráficos Por Computadora", GIGC - Grupo de Investigación de Gráficos por Computadora, Departamento de Ingeniería en Sistemas de Información, UTN-FRBA, Argentina

{ mleone, lbarbagallo, mbanquiero, dagromayor }@frba.utn.edu.ar
andresb@sistemas.frba.utn.edu.ar

Abstract. The visualization of complex virtual scenes can be significantly accelerated by applying Occlusion Culling. In this work we introduce a variant of the Hierarchical Occlusion Map method to be used in Real-Time applications. To avoid using real objects geometry we generate specialized conservative Occluders based on Axis Aligned Bounding Boxes which are converted into coplanar quads and then rasterized in CPU using a downscaled Depth Buffer. We implement this method in a 3D scene using a software occlusion map rasterizer module specifically optimized to rasterize Occluder quads into a Depth Buffer. We demonstrate that this approach effectively increases the number of occluded objects without generating significant runtime overhead.

Keywords: Occlusion culling, Hierarchical Occlusion Map, Occluder skin, visibility determination, Depth Buffer, Occlusion query, Occluder fusion.

1 Introduction

In Real-Time Computer Graphics, it is desirable to show complex scenes consisting of a large number of triangles with as good quality as possible. Because current hardware is not capable of supporting these kinds of complex scenes at an acceptable frame rate, optimization techniques are absolutely needed.

One of these optimization techniques is Frustum Culling which eliminates models that are outside of the viewing volume at an early stage in the pipeline. However, its major drawback is the fact that it does not consider the case where one object is not visible because it is being entirely blocked by another object. To solve this issue, Occlusion Culling technique needs to be implemented.

One of the main advantages of this method is that it reduces the overdraw of fragments that has a large computation effort in applications with intensive use of pixel shaders, like those with dynamic lighting or fake geometry generation.

In this work we focus on implementing an image space Occlusion Culling solution using software rasterization, based on a variation of the Hierarchical Occlusion Map (HOM) suggested by Zhang et al [1]. Unlike HOM, instead of rasterizing the objects full geometry, we rasterize simple conservative volumes called Occluder Skins. These

volumes are designed exclusively for the Occlusion Culling process assuring its conservativeness. This work presents a technique for automatic Occluder Skins generation based on Axis-Aligned Bounding Box (AABB).

Since we use Occluder Skins which are completely opaque, there is no need for the Opacity Map proposed in HOM. Finally, Occluder Skins are scan converted into a memory Depth Buffer using a streamlined software rasterizer designed specifically to support coplanar quad primitives.

2 Related work

A strategy for Occlusion Culling is proposed [2] which creates in Real-Time the shadow frusta produced by the objects chosen as Occluders. This technique works in the Occluder geometry-space but its main drawback is the fact that it does not take advantage of the effect called Occluder Fusion, which helps to accelerate to a large degree the rendering performance.

The technique is refined proposing a particular Occluder projection operator [3], in order to achieve Occluder Fusion.

Another solution [4] proposed consists in pre-computing a conservative visibility solution of the scene, so then this information can be used at Real-Time. For this to be effective, it normally requires a particular kind of scenes, like Indoor environments and architectural buildings, where discrete cells and connections can be delimited. Moreover, this technique does not work well with dynamic environments.

Current GPU architecture has the ability to perform Occlusion Queries in order to detect whether mesh would be finally visible on screen. Although these techniques have a great potential, there are still many difficulties to solve, especially when reading from CPU the result of an executed query in GPU at a predictable frame rate. Different techniques are proposed [5][6][7] to solve these issues.

There is a technique proposed for automatic Occluder mesh generation in urban environments [8], but the strategy is restricted to 2.5D objects, and is not clear how to extend the same approach to other types of meshes.

3 Algorithm overview

The proposed strategy consists of the following steps:

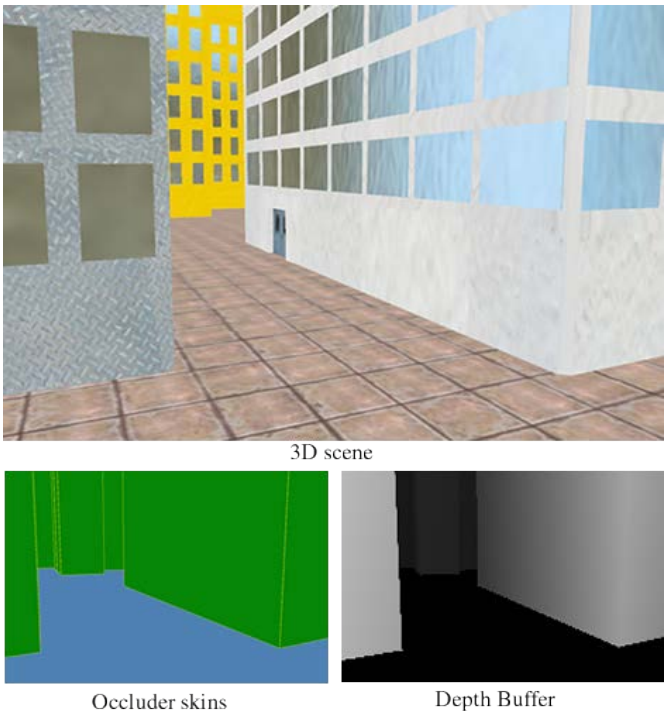

Fig. 1. An urban scene with Occluder skins and its respective Depth Buffer.

- *Occluder Generation:* Creates in offline time a simplified conservative volume that serves as an Occluder object in the scene.
- *Occluders Selection:* Determines in Real-Time which Occluders are inside the Viewing Frustum and selects the best candidates for rasterization.
- *Occluders Rasterization:* Detects the visible faces of the Occluders volumes and sends them to the specialized rasterizer. As a result, it updates a downscaled Depth Buffer version of the real frame buffer screen composed of depth points.
- *Occludees Test:* first it determines the objects of the scene that are inside the Viewing Frustum. Then for every object it generates the 2D screen projected Bounding Box and within that region compares each depth values stored in the Depth Buffer with a single conservative Occludee depth. If the Bounding Box depth is completely behind all the values of the Depth Buffer in that region, then we consider the object occluded and we avoid sending it to the rendering pipeline.

The Depth Buffer is updated in every frame. As the Occluders are rasterized and aggregated together in the Depth Buffer, the area covered by each Occluder is added

together resulting in larger Occluder areas, generally maximizing the number of Occlude rejections. This property is known as Occluder Fusion.

After all the Occluders are rasterized, every point in the Depth Buffer will hold the depth closest to the camera viewpoint, and it will be used to determine whether scene object is visible or not by testing if it is being occluded by a single Occluder or portions of different Occluders.

The visibility test performed is always conservative. The whole area of projected pixels of an object must be completely behind the values stored in the Depth Buffer to be considered as non-visible. Conversely, if at least one pixel of the object is closer to the depth stored in the Depth Buffer, the whole mesh is considered visible.

4 Occluder generation

The original HOM technique uses real meshes of the scene to populate the Depth Buffer. This can become an important bottleneck when the geometry of the meshes is complex. To avoid the issue, the method proposed in this work employs “impostor” or “proxy” objects that represent the volume of the Occluders. These objects must fulfill the conditions described [9] in order to be considered as a valid Occluder:

- *Simple:* Its geometry must be as reduced as possible to be efficient at software rasterization time. Usually this implies being convex.
- *Conservative:* The object must be completely contained inside the original mesh. Its volume should be less or equal than the real mesh volume.
- *Large:* Must occupy the largest volume possible of the original mesh. This way the Depth Buffer will be filled with more pixels and will help to discard more Occludee objects in the Occlusion Culling phase.

The Occluders generation is an Offline task, done prior to the runtime execution of the Real-Time application so they can be constructed manually with the help of a design tool. Although this method usually produces the most efficient Occluders, it can require a significant human effort. Therefore it is desirable to have an automated Occluder generation process, which may not be the most efficient, but creates a set of base Occluders that can be later refined manually.

In order to compute an Occluder automatically, we can use mesh reduction techniques as proposed by Hoppe et al [10], which decrease the geometrical complexity of a mesh but doesn’t guarantee the conservativeness of the result.

Another approach [11] presented consists of a heuristic to reduce the mesh complexity assuring that at all times the result is conservative. Nonetheless, the resulting simplified mesh can still be too complex to be rasterized in Real-Time for Occlusion Culling purposes.

In this work’s implementation we opted to calculate the least quantity of Axis-Aligned Bounding Boxes that approximate to a mesh that are conservative and as large as possible. The technique derives from the one suggested by Danell[9] and comprises the following steps:

- Voxelize all the space occupied by the mesh AABB, defining an adequate voxel size.

- Determine which voxels represent the external surface of the mesh.
- Determine which voxels are interior. These voxels are the ones contained inside the outer surface of the mesh.
- Compute the largest AABB based on the interior voxels of the mesh, which does not collide with any surface voxel.

To voxelize the space occupied by the mesh Bounding Box, we first define a voxel size and then perform a voxel-triangle collision detection against all triangles of the mesh, using the AABB/Triangle method presented by Moller [12]. In the case of the colliding voxels we mark them as "Surface Voxels", and for the rest we proceed to calculate the "Inner Voxels" in which we need to find at least one voxel that is completely surrounded by Surface Voxels. If we take any non Surface Voxel, we can consider it an Inner Voxel only if there exists at least one Surface Voxel (be it adjacent or not) in all the six directions that define the voxel.

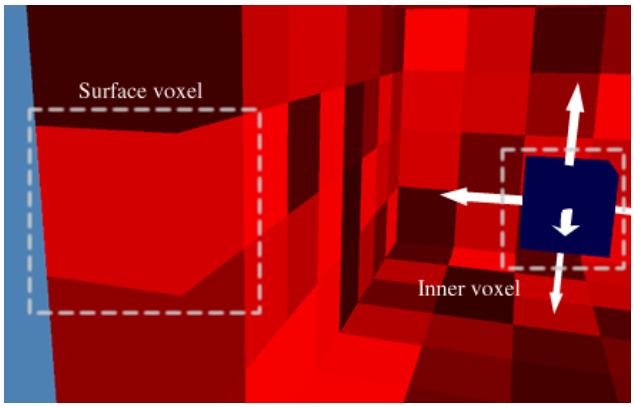

Fig. 2. Surface voxels and Inner voxels. The Inner voxels must have at least one Surface voxel in all of its six directions.

However, in order to apply this method, the mesh must fulfill the precondition of being completely sealed or "Water Tight". Even though non Water Tight voxelization methods exist, like the one suggested by Haumont [13], their results cannot be directly applied to the computation of conservative Occluders.

In order to find the first Inner voxel we proceed to check all the non Surface voxels of the mesh's space, until we find the one which satisfies the conditions mentioned earlier. Once the first Inner voxel is found, all the neighboring voxels are added to the list, except the ones that are Surface Voxels. The neighboring voxels are those which stay next to the current voxel, again in their six directions.

This process is repeated recursively until a voxel cannot expand to any further extent, because it is located next to a Surface voxel or because the whole voxel space has been checked.

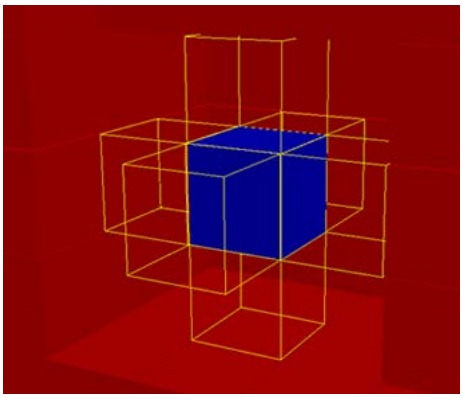

Fig. 3. Inner voxel expansion, until it reaches the Surface voxel frontier.

At this point, if the mesh was not Water Tight, the Inner voxel expansion would leak through one of the holes of the non sealed object, generating an invalid Inner voxel space. In this work we have not found a method yet to overcome this limitation. Once we have the set of Inner voxels, we proceed to search in an exhaustively manner the larger AABB which fits only within the Inner voxels. We start with an AABB of the size of an Inner voxel and we expand this AABB by a voxel length in every one of the six directions. In each expansion we check that all the voxels contained in the AABB are Inner voxels. We proceed recursively until the AABB cannot expand anymore because it contains Surface voxels.

Of all the possible AABB we select the one with the bigger volume and then all the Inner voxels contained in this AABB are tagged as “taken”.

Then we proceed to choose another initial inner voxel which is not “taken” and we execute again this sequence of steps, always avoiding the use of “taken” voxels.

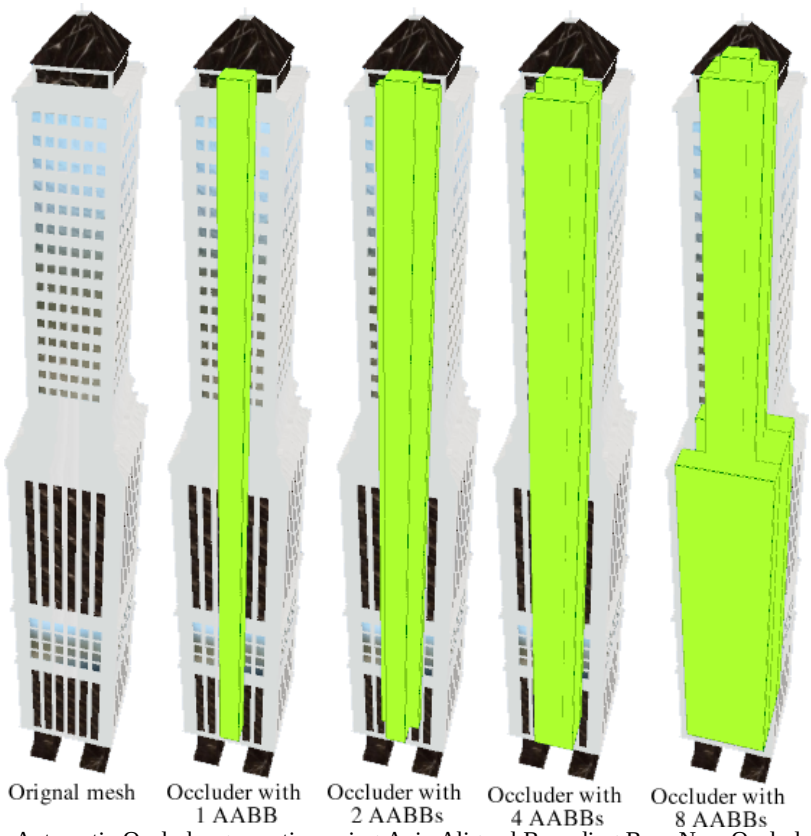

Fig. 4. Automatic Occluder generation using Axis-Aligned Bounding Box. New Occluders are computed for each mesh until a threshold is reached. The original mesh is shown behind the Occluders for volume comparison.

A threshold must be defined for this procedure, based on the following criteria:

- The maximum amount of AABB allowed per mesh.
- The minimum AABB acceptable volume for an Occluder. If it does not pass this minimum value, the AABB is rejected and the search for Occluders of this mesh is stopped.

One possible option to select the first Inner voxel that triggers the construction of a new AABB is to choose the “densest” one, as suggested by Danell [9]. The “densest” voxel can be obtained by searching for the one that has the largest average distance to all the Surface voxels of the mesh.

5 Depth Buffer generation

During the execution of the Real-Time application, the Occluders obtained with the previous method are used in the following way:

First Frustum Culling is performed with all the Occluders of the scene, which discards the Occluders that lay completely outside the View Frustum. To achieve this, a Frustum-AABB collision test is performed with the Occluder's AABB. Spatial subdivision techniques, such as Octree and KD-Tree, can be integrated at this point to speed the Culling process.

Then, for each Occluder we need to detect its visible faces, i.e. the faces that the Occluder's AABB possesses inside the View Frustum. An Occluder could have one, two or at least three visible faces at the same time. Since the Occluder has an AABB geometry, each face is made of a four-vertex coplanar polygon or "Quad". In order to detect which Quads are visible we need to inspect the angle between the Quad's normal (N) and the direction vector from the Quad to the Frustum Viewpoint (L):

$$N \text{ dot } L < 0$$

If the dot product between N and L is less than zero then the Quad is considered visible. For each visible face we have a 3D polygon that must be projected to screen. We project the four vertices of the Quad to obtain the 2D polygon in screen-space so that we can send it to the rasterizer.

This polygon may not fit completely inside the screen bounds, however the traditional rasterization techniques proceed to clip the polygon in homogeneous space, as detailed by Blinn [14]. In this particular case there is no need for a perfect Quad clipping, because the only goal of the rasterizer is to generate a Depth Buffer. The rasterizer simply computes the fragments that are inside the screen bounds.

The Quad clipping can be avoided without problems except when a polygon edge has one vertex with a negative projected value of W and the other vertex with a positive W. In these cases, it is necessary to perform some sort of clipping in order to avoid invalid projection results.

To deal with this problem we project the Quad to View-Space and then we clip the edges that intersect the Frustum Near Plane. In this way we perform a polygon clipping procedure but only against one plane, when the clipping techniques are usually employed to perform tests with six planes, like the one detailed by Blinn [14].

It must be noted that the clipping procedure may not always generate a four-vertex polygon. Sometimes a three or five-vertex polygon is generated. For the first case only a triangle is sent to the rasterizer, and for the five-vertex case we send a Quad and a triangle to rasterize.

The rasterizer received the (X, Y) projected coordinate for each polygon's vertex, along with its project Z coordinate (after being divided by W). Finally the rasterizer computes the depth value for each fragment by using a scan-line conversion procedure.

An optimal choice is to select only one depth value for the whole polygon, for example the farthest Z value. In this way there is no need for the rasterizer to compute the depth value of each fragment, reducing expensive fragment computations. The problem with this approach is that it is extremely conservative for some types of Occluder geometries, like the ones with high Z variation in their points (in View Space).

6 Testing the Occludees

Only the objects that passed the Depth Buffer test are sent to the GPU. We first discard those who lay completely outside the View Frustum. Then for every mesh we generate its 2D screen projected Bounding Box and from this projection we choose only one Z value, by selecting the one closest to the View Point. In this way we guarantee the conservativeness of the test.

Then this 2D rectangle is sent to the rasterizer, where its unique Z value is checked against the corresponding fragment's Z value of the Depth Buffer. If at least one value of the rectangle is closer to the View Point than its corresponding in the Depth Buffer, then the rasterizer stops and the mesh is considered visible. This enables an early-out strategy. The mesh is considered non visible when the rectangle Z value is behind all the fragment Z values of the Depth Buffer.

7 Implementation

The software rasterizer designed by Barbagallo et al. [15] was used for the task of Depth Buffer generation and Occludees testing. The rasterizer is designed with the only purpose of fast generation of the Depth Buffer and for efficient Occludee rejection against it. Due to its design, it does not have the overhead of other typical stages of the pipeline, like Shading and Texture mapping. This rasterizer has support for triangles and Quads primitives. The Quads are supported in a native way, when most of other rasterization solutions convert them to triangles. A tiled rasterization approach is used which divides the Depth buffer in fixed section to accelerate the procedure. Each tile rasterization is deferred until it is completely necessary.

A very important aspect is to select a proper size for the Depth Buffer. A Depth Buffer with the same size of the Frame Buffer will generate too much overhead in the rasterizer, and usually there is no need of such fragment precision for Occlusion Culling purposes. On the contrary a Depth Buffer that is too small will be overly conservative, reducing the performance boost of the Occlusion Culling procedure. For this work we choose to use a Depth Buffer with a quarter size of the screen.

8 Results

A 3D city model was built, composed of 22 meshes, adding up a total of 50.189 triangles. For this scene 40 Occluders were generated in Offline time. In order to analyze the algorithm performance, eight representative scene View Points were taken. For each position we compute the following metric: $Value = (t - v) / t * 100$, where *t:* total scene meshes and *v:* total visible meshes.

This metric allows us to see the percent of discarded meshes that Occlusion Culling prevented from sending to the GPU in each frame. The metric is computed with Occlusion Culling deactivated and then with it activated. We also include the frames per second that resulted from rendering the scene with and without Occlusion Culling. The results were computed using a PC with Intel Core 2 Duo 1.86GHz processor with 2GB RAM and an ATI Radeon Xpress 1100 GPU.

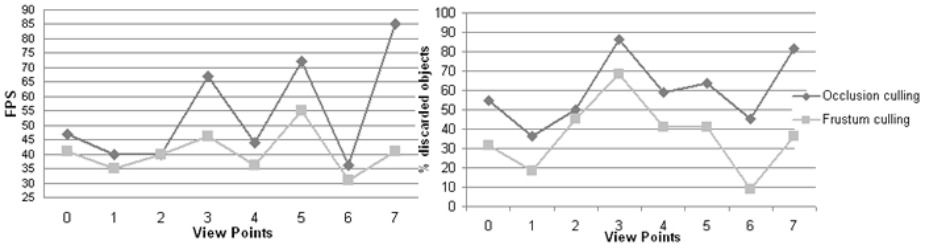

Fig. 5. Left: FPS rendering performance only with Frustum Culling and then with Occlusion Culling activated, at the eight different selected View Points. Right: Discarded mesh percent, first with only Frustum Culling and then activating Occlusion Culling, at the eight different selected View Points.

9 Conclusions

We have demonstrated herein that it is viable to apply HOM in Real-Time applications using modern graphics hardware. Using full object geometry can be prohibitive, so specially generated Occluders are to be used. The manual generation of these Occluders can take too much work, and for that reason it should be combined with the automatic Occluder generation techniques introduced in this work. We think it is wise to perform manual adjustments to the Occluders as part of the last stage of the scene creation procedure.

To be able to efficiently rasterize the Occluders using a software approach it is recommended to make the most of a specialized rasterizer, focused only on the Depth Buffer generation, ignoring other pipeline stages such as Shading or Texturing. A native support for Quads rasterization is also important to exploit the geometry features of Occluder skins. Finally, choosing the right Depth Buffer dimensions is essential in order to limit the Occlusion Culling process computational time to only a fraction of the total frame rendering time.

10 Future Work

Occluder generation could be extended to other types of volumes such as Oriented Bounding Boxes and Quads. An algorithm that supports these new volumes ought to choose automatically the right type for each particular mesh.

In addition, it is necessary to study which approach must be followed to generate Occluders from non Water Tight meshes. As a first step, this new approach should detect these cases preventing the generation of invalid Occluders, and as a second step the approach should also contemplate the creation of valid Occluders for these cases.

The Occluder clipping process should be enhanced to avoid cases where the polygons end up formed with more than four vertices, which makes the rasterizer overload with extra primitives.

11 References

1. Zhang, H., Manocha, D., Hudson, T., Hoff, I.: Visibility culling using hierarchical occlusion maps. In : Proceedings of the 24th annual conference on Computer graphics and interactive techniques, New York, NY, USA, pp.77-88 (1997)
2. Hudson, T., Manocha, D., Cohen, J., Lin, M., Hoff, K., Zhang, H.: Accelerated occlusion culling using shadow frusta. In : Proceedings of the thirteenth annual symposium on Computational geometry, New York, NY, USA, pp.1-10 (1997)
3. Durand, F., Drettakis, G., Thollot, J., Puech, C.: Conservative visibility preprocessing using extended projections. In : Proceedings of the 27th annual conference on Computer graphics and interactive techniques, New York, NY, USA, pp.239-248 (2000)
4. Teller, S., Sequin, C.: Visibility preprocessing for interactive walkthroughs. In : Proceedings of the 18th annual conference on Computer graphics and interactive techniques, New York, NY, USA, pp.61-70 (1991)
5. Bittner, J., Wimmer, M., Piringer, H., Purgathofer, W.: Coherent hierarchical culling: Hardware occlusion queries made useful. In : Computer Graphics Forum, vol. 23, pp.615-624 (2004)
6. Hillesland, K., Salomon, B., Lastra, A., Manocha, D.: Fast and simple occlusion culling using hardware-based depth queries. Chapel Hill: University of North Carolina (2002)
7. Staneker, D., Bartz, D., Meissner, M.: Improving Occlusion Query Efficiency with Occupancy Maps. In : Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics, Washington, DC, USA, pp.15-- (2003)
8. Germs, R., Jansen, F. W.: Geometric simplification for efficient occlusion culling in urban scenes. In : Proc. of WSCG, vol. 2001 (2001)
9. Danell, N.: Automated Occluders For GPU Culling. (Sep 2011)
10. Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., Stuetzle, W.: Mesh optimization. In : Proceedings of the 20th annual conference on Computer graphics and interactive techniques, New York, NY, USA, pp.19-26 (1993)
11. Sub, T., Koch, C., Jahn, C., Fischer, M.: Approximative occlusion culling using the hull tree. In : Proceedings of Graphics Interface 2011, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, pp.79-86 (2011)
12. Akenine-Moller, T.: Fast 3D triangle-box overlap testing. In : ACM SIGGRAPH 2005 Courses, New York, NY, USA (2005)
13. Haumont, D., Warzee, N., Bruxelles, U.: Complete Polygonal Scene Voxelization. (2002)
14. Blinn, J., Newell, M.: Clipping using homogeneous coordinates. In : Proceedings of the 5th annual conference on Computer graphics and interactive techniques, New York, NY, USA, pp.245-251 (1978)
15. Barbagallo, L., Leone, M., Banquiero, M., Agromayor, D., Bursztyn, A.: Techniques for an Image Based Occlusion Culling Engine., Buenos Aires (2012)