BqR-Tree: a Data Structure for Flights and Walktroughs in Urban Scenes with Mobile Elements

J.L Pina[1], F. Seron[1] and E. Cerezo[1]

[1]Advanced Computer Graphics Group (GIGA)
[1]Computer Science Department, University of Zaragoza
[1]Engineering Research Institute of Aragon (I3A), Zaragoza, Spain
[1]jlpina@zaragoza.es, seron@unizar.es, ecerezo@unizar.es

Abstract

BqR-Tree, the data structure presented in this paper is an improved R-Tree data structure based on a quadtree spatial partitioning which improves the rendering speed of the usual R-trees when view-culling is implemented, especially in urban scenes. The city is split by means of a spatial quadtree partition and the block is adopted as the basic urban unit. One advantage of blocks is that they can be easily identified in any urban environment, regardless of the origins and structure of the input data. The aim of the structure is to accelerate the visualization of complex scenes containing not only static but dynamic elements. The usefulness of the structure has been tested with low structured data, which makes its application appropriate to almost all city data. The results of the tests show that when using the BqR-Tree structure to perform walkthroughs and flights, rendering times vastly improve in comparison to the data structures which have yielded best results to date, with average improvements of around 30%.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.6]: Graphics data structures and data types—

1. Introduction

Among the largest and most complex scenes are urban environments; urban scenes are particularly interesting for several simulation applications: crowds in motion, emergency evacuations, virtual tours, urban planning, military operations, traffic management and its impact of noise to the surrounding buildings, etc.

Not only the size and the complexity of actual virtual urban environments are increasing, but the need to populate them with more than just a few mobile elements (characters, cars,...). In fact, real-time rendering of highly populated urban environments requires the resolution of two separate problems: the interactive visualization of large-scale static environments, and the visualization of animated crowds or other mobile elements. Both tasks are computationally expensive and only now are beginning to be addressed by developers. Several techniques commented in next section, have been developed to accelerate the rendering of complex scenes. This paper focuses on one of them, view-culling and in particular, in the use of a proper data structure to accelerate it.

As it is well-known, a scene graph is a directed acyclic graph, which describes the entities and dependencies of all the graphic elements in the scene. A view frustum culler culls away all the scenegraph's nodes that lay outside the view frustum, i.e. those objects that are outside the observer's field of view, by using the nodes' bounding volumes. Faster view frustum cullers are particularly important if complex and large scene graphs are traversed. Dividing complex geometry, consisting of multiple triangles, into an adequate data structure can greatly improve the ability to cull away triangles that lie outside the view frustum, resulting in less triangles to be sent to the rendering pipeline. The advantage of using a data structure is that the work of creating the data structure is performed in pre-processing time with little addition to the real-time computation. Afterwards, other techniques can be applied to further improve the framerate. Data structures are commonly designed considering

only static environments, but they should also be effective in environments populated by mobile elements, therefore, both type of elements should be taken into account. R-tree is the most common data structure implemented in urban scenes because its management of bounding volumes in general leads to an efficient view-culling. Its main drawback is overlapping, especially concerning large objects, which are common in urban scenes.

On the other hand Quadtree is the most common data structure implemented in terrain scenes. It leads to a spatial decomposition without overlapping and in which objects which are near to each other remain close in the data structure, which requires less culling time for the scene graph.

It was originally developed for point scenes and is not well suited for complex objects such as urban blocks or buildings. When compared with an R-Tree decomposition, the Quadtree decomposition is more adequate to the spatial distribution of urban elements and requires less culling time, but the bounding volumes of an R-tree are more efficient than the quadrants of the Quadtree for testing the view-frustum.

The aim of this paper is to find a data structure capable of interactively performing urban walkthroughs and flights containing numerous dynamic entities without the use of predefined paths. In order to fulfill these requirements, all elements of the city are grouped into a basic unit: the urban block, since the basic urban unit is a natural way to integrate static and mobile elements in the data structure. Setting out from a typical quadtree decomposition, a data structure named BqR-Tree or Block-Quadtree R-Tree is built. The use of a Quadtree decomposition leads to a very efficient spatial decomposition of the urban scene; afterwards, the data structure is populated like an R-tree to take advantage of the use of bounding volumes of urban objects. This data structure leads to a great increase in frames per second (FPS), both in flights and walkthroughs, even in environments populated by mobile elements. BqR-Tree is capable of being applied to urban data with low structure or no structure at all. Therefore, the proposed data structure can be applied to data acquired from several sources: 2D GIS, terrain measurements, etc. Another advantage of this method is that identification of buildings is not required. The basic unit, the urban block, can be automatically identified under any circumstances and is well suited to integrate, seek and perform fast culling of the mobile elements in the scene graph.

The organization of the paper is the following: next section is devoted to discuss previous related work. Section 3 presents the new data structure and section 4 shows the results, whereas in section 5 conclusions and future work are outlined.

2. Related work

A great variety of solutions can be found in the bibliography related to the rendering of large data models in real time. Methods can be classified into five main groups of techniques: LOD or level of detail, billboards, occlusion culling, view-culling and ray tracing. Several of them are often combined to produce better results as well. Nonetheless, most of these techniques were developed for the efficient management of large static polygonal models, so their application to the management of thousands of complex dynamic entities, such as virtual actors, is not a trivial matter.

LOD is one of the most frequently used techniques: continuous LODs have been tested [DB05], but they reduce rendering speed; so have hierarchical models [FS93] that choose the proper resolution according to the node of the tree being displayed. However, it is very difficult to avoid bothersome drops between frames of different resolutions.

Billboards and impostors [ADB08], are based on the use of images of objects instead of the 3D objects themselves. They are very effective in reducing rendering time. But, in fact, the task of visualizing models with many polygons is not really solved, but avoided. Relief impostors techniques have been implemented recently with a very good performance, by means of a quadtree [ADB08] or by means of texturised blocks [CMG*07] which outperforms hierarchical Lod [EMB01] in urban scenes. A minor variation of billboards is the image reuse technique implemented in static environments, or the technique implemented by Ulicny et al. [UCT04] for animated individuals by storing pre-computed deformed meshes. Dobbyn et al [DHOO05] have presented an hybrid system for animated crowds, combining image-based representations and geometric representations.

The occlusion culling technique resorting only to software techniques or taking advantage of the GPU [BWPP01], involves culling away objects that are not visible due to their being hidden. The identification of hidden objects from the camera's point of view for each frame every time the camera or an object moves is its main drawback. This technique is widely used in urban environments [COCSD03] [GBSF05] but it is not suited to urban flights in real time. In its native form, it is not well suited to the treatment of mobile elements.

View frustum culling is an acceleration technique well studied in static environments [SMM08], [Pla05], [AM00] but it is rarely implemented in dynamic scenes. Seron et al. [SRCP02] proposed two new types nodes for the integration of mobile elements into the scene graph, Nirnimesh et al. [NHN06] proposed multiple view-frustum.

Ray tracing algorithms are very powerful for computing advanced lighting effects. Off-line rendering has primarily used ray tracing for this reason. Unfortunately, ray tracing is computationally demanding and has not yet benefited from special purpose hardware. Consequently, it could not be supported at interactive frame rates until very recently. Searching to obtain real time frame rates ray tracing has taken advantage of data structures. Kd-trees could be the most popular and have been declared the "best known

method" for fast ray tracing [Sto05]. But Kd-Trees are not well suited to dynamic updates, because even small changes to the scene geometry use to invalidate the tree. Thus, researchers have again started to actively explore better ways to support dynamic scenes with other acceleration structures as BVHs (Bounding Volume Hierarchies). Most Kd-Tree and BVH based ray tracers today employ a surface area heuristic (SAH) for building the data structure. The heuristic SAH [Wac07] it is optimal, although the minimization cost function poses demands which can not be fulfilled in real life (np-complete problem) and leads to extremely long pre-calculation times. An approximation of the cost-function will result in a decreased quality of the hierarchy. Most research incorporates the coherence within successive frames [AM00] to reduce the number of rays to be traced, but it is not well suited to walkthroughs with large rotations and movements of the camera. Nowadays, ray tracing is still too slow for rendering urban dynamic scenes. Nevertheless, fast construction techniques are being implemented on manycore processors or GPUs [LGS*09] [WIP08].

Most of the previous algorithms incorporate an indexing data structure to accelerate rendering which results in a great improvement in rendering times, and this is done in preprocessing time. Two main categories can be found among the most widely used indexing structures: Hierarchical structures and Bounding Volume Hierarchy (BVH). The main difference between these two categories lies in their approach to dividing data space. Structures which belong to the first category use space partitioning methods that divide data space along predefined hyper-planes regardless of data distribution. The resulting regions are mutually disjoint and their union completes the entire space. Kd-Tree and Quadtree belong to this category. Structures which belong to the second category use data-partitioning methods which divide data space in buckets of MBV (Minimum Bounding Volumes) according to its distribution, which can lead to possible overlapping regions. R-Tree belongs to this category. Those data structures which belong to the R-Tree families yield the best performance [GG98], being the VamSplit R-Tree the best performance R-tree [KS97] [WJ96]. This is why this structure has been chosen to make comparisons with the proposed data structure. The R-tree data structure is very sensitive to the order in which objects are inserted, but its real drawback is the overlap, especially affecting large objects common in urban scenes [GG98]. The criterion of object insertion, minimal bounding volumes, usually leads to the separation of large objects on the tree although they may be near to each other in the scene. On the other hand, Quadtree is not sensitive to the order in which objects are inserted and there is no overlapping, it keeps objects that are near to each other in the scene together in the scene graph, therefore it is a very efficient spatial decomposition. Nevertheless, it has been designed like a point access method and although there are more complex versions capable of managing polygonal data, they are not well suited to manage large urban objects.

Hybrid data structures have been proposed to overcome the drawbacks of the traditional spatial index structures. The K-D-Tree with overlapping [XP03], or the Q+Rtree [CM99], which is an hybrid tree structure consisting on an R-tree and a Quadtree. The Rtree component indexes quasi-static objects and the Quadtree component indexes fast moving objects.

In this paper, an hybrid indexed data structure to be built in preprocessing time for acceleration purposes is proposed. Instead of using MBV as is usual in an R-Tree, and in most urban applications, a Quadtree decomposition [ABJN85] has been chosen, which is rare but not unique in urban environments [Man08] [ADB08]. The Quadtree decomposition implemented is a region quadtree (detailed in section 3), but should not be confused with a Loose-quadtree [Sam90], in which elements of a given size are always in a given tree level. Details of the proposed data structure are facilitated in the next section.

3. The BqR-Tree

3.1. Building the scene graph from the blocks

The proposed BqR-Tree is based on the decomposition of a city into blocks: the block is considered the minimum and indivisible unit of the city as well as the basis of the proposed structure. The term block is used to name the group of urban elements completely surrounded by streets, i. e., the usual meaning of urban block.
The BqR-Tree structure is an improved R-Tree whose method of space partitioning is a quadtree decomposition, once the tree structure is built, it's populated with MBVs and geometry like an R-tree, therefore the BqR-Tree structure requires less culling time. The structure also incorporates a second improvement. The MBVs of the nodes are bounding-spheres according to the results of [KS97]. This produces improvements in node access speed and storage requirements. Nevertheless, the use of bounding-spheres produces too much overlapping, which is the reason for the use of the intersection of bounding-boxes and spheres for the leaves (from now on bounding-boxes-spheres), as it will be explained later on.
For every city it is necessary to identify all the basic indivisible elements. In particular, in order to build the BqR-tree, it is necessary:

- To identify all the blocks in the town and calculate its geometric centers and bounding spheres,
- To assign every graphic element of the town to an urban block.

In Figure 1 a part of the city under consideration (Zaragoza, Spain) with the blocks and its centers already identified is displayed. Identification of blocks takes place setting out

from the streets, which are identified from the outset. Block status is assigned to any portion of terrain completely surrounded by streets.

The data used to test our model has been exported from a 2D-GIS. In the initial files neither buildings nor blocks were already identified as entities. Therefore, filtering, structuring and 3D elevation have been necessary. To build the BqR-

Figure 1: *Identifying the blocks and its centers in a city*

Tree, the first step is to take the smallest rectangle bounding the city. This rectangle is divided into four equal rectangles called quadrants or buckets; all quadrants are recursively subdivided in this manner. The BqR-Tree tree is formed by recursive division of the city into quadrants. A quadrant is considered to be indivisible if it contains only one block. No splitting of the objects (urban blocks) is implemented, since this would involve a scattering along the tree of the pieces of urban blocks, which would in turn lead to a poorer performance of the data structure. Should a division cross a block, the entire block is assigned to a single quadrant, the one that contains the geometric center of the block. At the end of the process, every final quadrant contains a single block or remains empty. Figure 2 presents a quadtree decomposition performed on the example shown in Figure 1.

The second step is to build the tree. To do this, every quadrant is assigned to a node or sub-node and every block is assigned to a leaf. Figure 3 shows a graphic representation of the BqR-Tree structure corresponding to the example of Figure 1 and 2. Leaves are represented by squares and nodes by ellipses. Empty quadrants are not assigned to nodes, thus categorizing the quadtree as an adaptive quadtree. Each leaf of the tree stores the entire geometry of the block as well as the bounding volume for that block. Each node stores a pointer to its descendants and to the bounding-sphere of the node. In the case of the leaves, the bounding-box-sphere of each block (intersection of the sphere and the box of the block) (see Figure 4) is calculated and stored. For each node, the bounding sphere is composed of the union of bounding-volumes of its descendants, in a bottom-up manner.

Therefore, the resulting structure is a tree of the nodes'

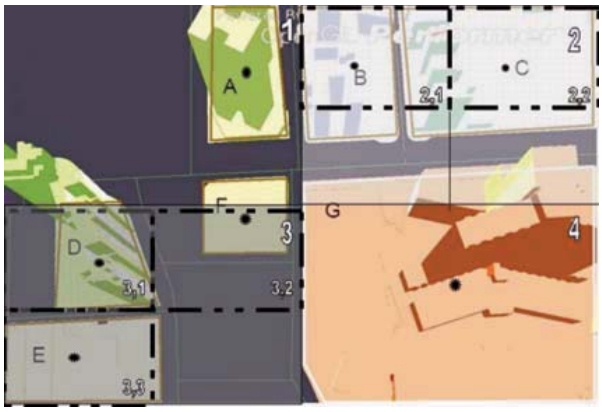


Figure 2: *Quadtree decomposition of the example seen in Figure 1*

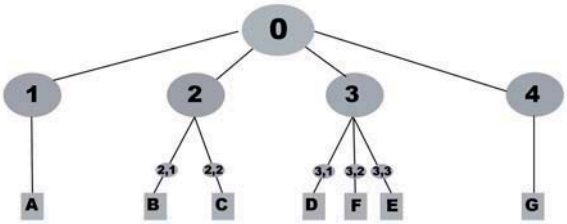


Figure 3: *Tree representation of the example seen in Figure 1*

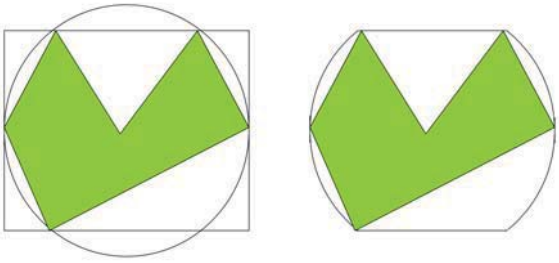


Figure 4: *Bounding Volume of a Block. The left image corresponds to the block (seen from above) with its bounding box and bounding sphere (2D). The right image shows the bounding volume from the intersection of its bounding box and bounding sphere (2D)*

bounding-spheres which ends with the bounding-boxes-spheres and contains the geometry of the blocks as well. A linear codification, corresponding with its quadtree decomposition, based on a numeric key, is used to identify all the elements of the tree, nodes and leaves. Storage of the key of each element of the tree is required due to the non-representation of empty nodes. The quadrants' dimensions

are not needed to be stored in the tree, they are only needed for the spatial distribution, also the bounding-volumes of the quadrants are not calculated.
Regarding the performance of object-location, there are two possibilities:

- If the object-location is referred to an object initially indexed, such as a block, location is trivial because of the quadrants decomposition and indexing of the blocks and quadrants.
- If the object-location is referred to an object not initially indexed, as a point, the procedure is the usual point-location operation in R-Tree family. Location is easy and fast due to the bounding-volumes.

Summarizing, the properties of the BqR-Tree data structure are the following:

- The structure is a tree of the nodes' bounding-spheres (quadrants are disjoint but not the bounding-spheres of the nodes associated with them) and ends with the bounding-boxes-spheres and the geometry of the blocks.
- Although very similar to an SR-Tree and VamSplit R-Tree, the most important difference is the use of the quadtree decomposition with no splitting blocks, and the use of bounding-boxes-spheres for the leaves.
- The complexity of the search for an element is proportional to the depth of the tree, being O(n) in the worst case, where n is the maximum depth of the tree, which tends to be small. Union, intersection and complement share the same complexity, all of which leads to very efficient tree culling.

3.2. Incorporation of the mobile elements to the scene graph

Every mobile element (ME) is always associated with a street and a direction. Each street is composed of two directions and every direction (semi-street) is associated with a block. Therefore, every ME is associated in an unambiguous way to only one block, and the geometry of each mobile element is stored in the corresponding leaf of a final node. Thus, the position and speed of every ME is easy and quickly known every time. When the MEs are moving, it is necessary to inspect their connections. Only the MEs whose movement leads them to another street should be inspected. If the new street belongs to a different block, then the ME must be disconnected from its old block and connected to the new block. But in fact, not all MEs have to be re-connected, only those that are inside the view-frustum or have just abandoned it. For the rest of the Mes, it is enough to annotate the change and implement it when they come into the view-frustum. Connections are performed by means of a pointer to the ending node of the block (the parent of the geometry of the block). If the ME is defined as a simple geometry model, without Lod, the geometry of the ME is stored into a leaf which has a pointer to its parent, the node of the block. Thus leaves of the block and the ME are siblings. If the geometry of the ME is more complex, with Lod, the root node of the tree of the ME is connected with a pointer to the parent node block. Re-connection is performed by a re-writting of the pointer of the ME, from its old parent to its new parent.

4. Tests

The data used belong to the city of Zaragoza (Spain) and have been kindly provided by the city council. The initial file was in Microstation format and its data were converted to a 2,013,900 point text format. The BqR-Tree structure built was comprised of 145 nodes and 96 leaves. Auxiliary files for the 96 blocks as well as one file with the 300Mb 3D model and 1,688,218 triangles were created. All tests were performed with a computer provided with a DualP4 Xeon Pentium 4 processor at 2.8 GHz and a memory of 2Gb. The graphic card is a GeForce Fx 6800 Ultra with a memory of 256 Mb. The operating system is Windows XP.
The scene graph OpenGL Performer has been used to test the proposed BqR-Tree data structure. In order to load the structure, a file with all the geometry and the BqR-Tree structure elements arranged in nodes and leaves is supplied to Performer. A dll reads it, nodes are loaded as pfGroup, blocks are loaded as pfGeoset and pfBuilder creates the tree in Performer format.
Of course, any other scene graph may be used, as only a new dll, which can be generated for the new format, is required. Besides, the preprocessing programs are in tcl, and the associated dll in C. Both are independent from the operating system and hardware and are, therefore, totally portable. Although Performer is equipped with tools for speeding up rendering, in order to show that the increase in rendering speed is caused only by the new structure, the only acceleration technique implemented is view culling, which is directly derived from the culling of the structures. The aim of the next tests is to show the increase in rendering speed using the Bq-Rtree data structure at the view culling process, it is not the aim to perform a complete application which should moreover implement backface culling, occlusion culling and LOD. Anyway, backface culling and LOD equally would affect to all data structures and occlusion culling would be applied after the view culling; therefore the effect of all of them would be an increase of the rendering speed for all data structures.
Performer, as the other scene graphs, implements its view culling in a top-down manner. The bounding-volume of the first node, the root, is tested against the view frustum, if the bounding-volume of the node is completely or partially inside the view frustum, the culling process continues with its descendant, otherwise the node and its descendant are culled away because they are not visible.

4.1. Tests on a city model

The starting point for the tests was the previously mentioned 3D model of the city, which is comprised of 1,688,023 triangles (see Figure 5). A program with the essential elements allowing free or guided camera movements was implemented. A route containing all the usual camera movement in flights and walkthroughs was designed; this route remains the same for all tested structures. The route (see Figure 6) begins at the outskirts of the city at ground level, and undertakes a walkthrough to the heart of the city. Once there, the camera rises vertically until it is above the rooftops facing the ground. Finally, a diagonal flight is performed from this position to the ground, along with camera rotations pointing in the direction of the movement.

Figure 5: *3D City model. In order to facilitate the identification of urban blocks these are coloured instead of texturised.*

To determine the real improvement resulting from the use of BqR-Tree, it was decided to compare it with the VAM-Split data structure [WJ96] an optimized R-Tree. It is a static index structure (an insert or delete operation implies a reorganization of the whole tree), unlike quadtree. The tree construction algorithm is based on the K-D-Tree and it is created by recursively selecting dataset splits using the maximum variance dimension and choosing a split that is approximately the average.

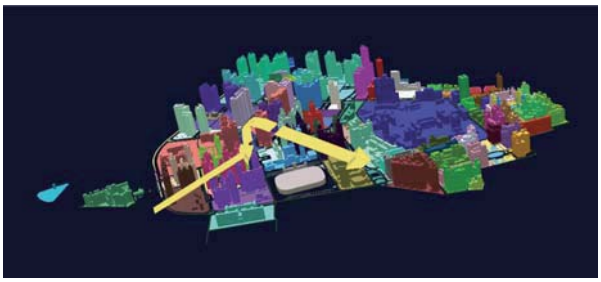


Figure 6: *Camera route*

The view culling is applied for both data structures in the same manner: the bounding volumes of the nodes and leaves are tested against the view frustum pyramid. An unstructured model was also used as a reference.

The evolution of the time required (in seconds) to render each frame for each of the previously mentioned three data structures can be seen in Figure 7:

- The yellow (top line) line corresponds to the unstructured mode.
- The red line (middle line) corresponds to the VamSplit R-Tree structure, and the blue line (bottom line) corresponds to the BqR-Tree line.

As explained, the only acceleration technique implemented is view culling, which is derived from the culling of the structures. The yellow line is constant, and therefore the time required for rendering is always the same for each frame. This was predictable, since there can be no view culling if there is no data structure. As might also have been anticipated, it is slower than any of the other two data structures. Regarding the other two lines, the first consideration is that the blue line (which represents the BqR-Tree) is the line with the quickest rendering time for each frame, and is markedly inferior than the red line (which represents the VamSplit R-Tree). In fact, the red line is not better at any point of the route; at the very most, it is equal to the blue line in a small number of frames. The BqR-Tree line is also noticeably more regular than the line which represents the VamSplit R-Tree. The latter shows two great depressions and multiple oscillations. If the results are analyzed the reason for this becomes apparent.

At the beginning of the route, the camera sees the city from far away, and therefore with no opportunity for view culling, thus the initial equivalence of the red and blue lines. Nevertheless, the more the camera advances into the city, the more important view culling becomes and the more both lines deviate. The ascent of the camera takes place between frames 48 and 140; this is the interval where both structures approximate the most. Later on, a diagonal flight towards the ground separates them once again. In the last 20 frames both lines come together again when the camera is facing the floor and is about to land and view culling possibilities are similar for both structures.

The explanation for these differences between the two data structures is confirmed analysing the visible triangles per frame (see Figure 8). The yellow line (top line) is constant once again, as might have been expected: absence of view culling forces the system to test every triangle in the city for each frame. The behaviour of the other two lines is similar to that of Figure 7. The same oscillations are repeated once again.

Therefore, it can be clearly concluded that the new data structure BqR-Tree (blue bottom line), allows for more efficient culling.

Additionally, the average performance of each structure is shown in Table 1. If the ratio between the average rendering time of both structures is calculated, a marked improvement may be appreciated: the result of the BqR-Tree/VamSplit R-

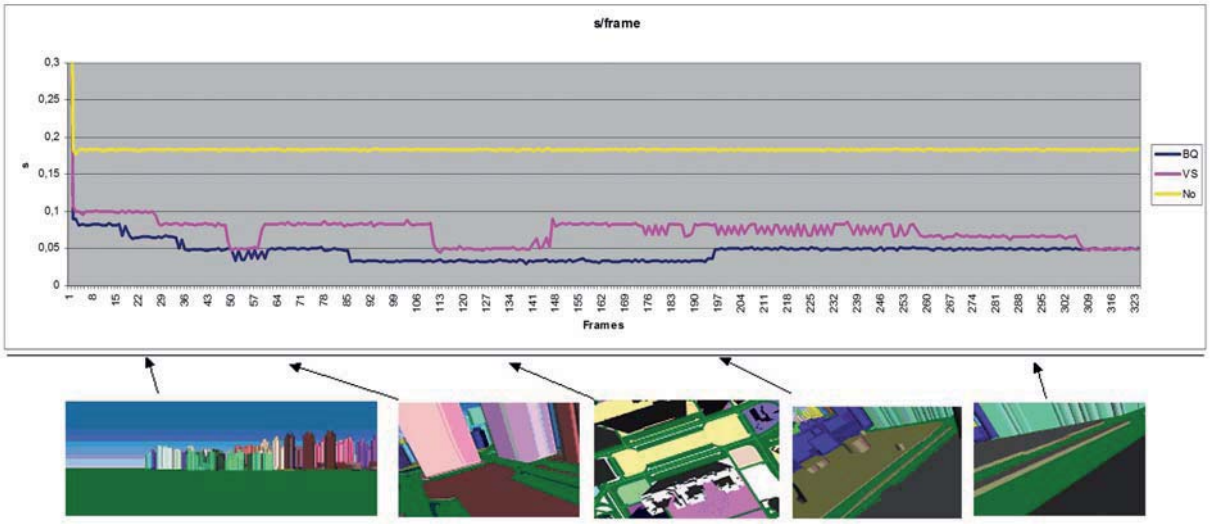


Figure 7: *Evolution of rendering time for each structure. A few images of what the camera sees at key moments have been added.*

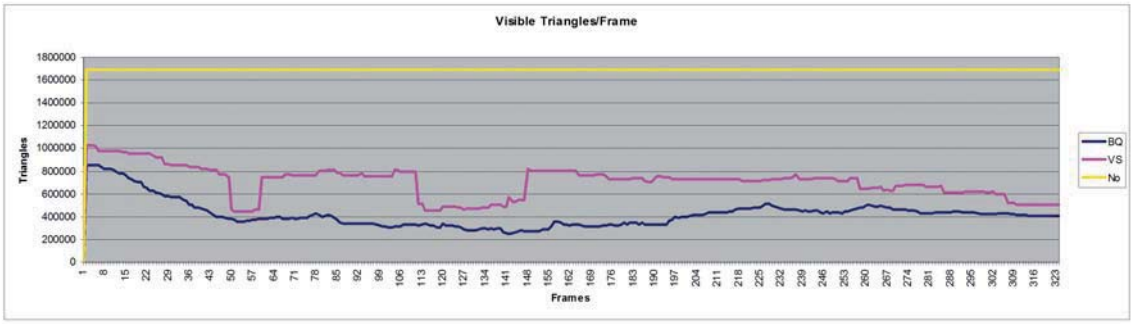


Figure 8: *Evolution of visible triangles per frame*

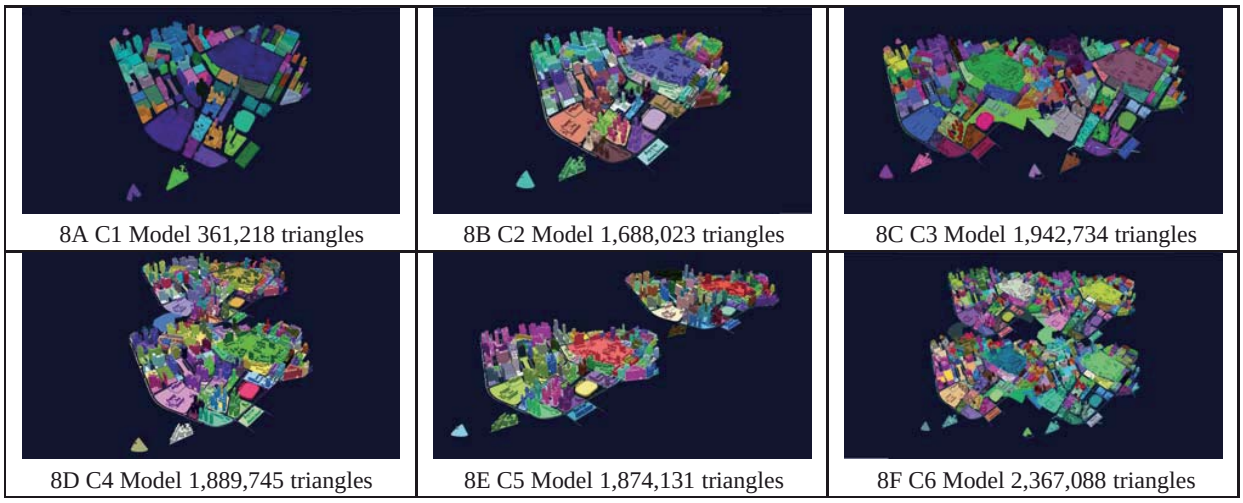
Tree division is 0.638 and that of the BqR-Tree/No Tree division is 0.261. Therefore an improvement of almost 40% can be observed in relation to the VamSplit R-Tree structure and the results are 75% better than when no structure is used.

Table 1: *Average values for each structure*

	seconds/frame	triangles/frame
No tree	0.1850	1,688,023
VamSplit	0.0758	704,873
BqR-Tree	0.0483	420,894

4.2. Impact of the size and distribution of the city models

The final step was to ascertain how the size and distribution of the city affect the rendering time results. In order to establish this, six different models of cities were built (Figure 9). The first of them (see Figure 9A), was named C1 and has less triangles than any other but the same volume of the initial city. It consists of 361,218 triangles obtained by elimina-



8A C1 Model 361,218 triangles | 8B C2 Model 1,688,023 triangles | 8C C3 Model 1,942,734 triangles

8D C4 Model 1,889,745 triangles | 8E C5 Model 1,874,131 triangles | 8F C6 Model 2,367,088 triangles

Figure 9: *Models of the city generated for the tests*

Table 2: *Average rendering times (s/frame) for each city model*

	C1	C2	C3	C4	C5	C6
Bq	0.021	0.048	0.053	0.055	0.057	0.064
Vs	0.031	0.076	0.079	0.084	0.079	0.094

tion of every element of the initial city except for buildings. This allows to test the influence of triangle density on the performance of each structure. The second city, C2, (see Figure 9B), is the initial model, which has already been tested. The third city, C3, (see Figure 9C), is made up of 1,942,734 triangles and is the result of joining the two previous models, C1 and C2. Model C1 is added to C2 in the direction of the axis of the Xs. The fourth city, C4, (see Figure 9D), is like C3 but with C1 added to C2 in the direction of the Y axis, and is made up of 1,889,745 triangles.

The fifth city, C5, (see Figure 9E), is like C3 and C4 but now C1 has been added in the diagonal XY direction. It is made up of 1,874,131 triangles. Finally, the sixth city, C6, (see Figure 9F), is C2 added to three C1, one in each axis direction, X, Y, XY, and is made up of 2,367,088 triangles.

It is very interesting to observe the evolution of rendering time for each structure in relation to the number of triangles of the cities. Table 2 shows the average values of rendering time in seconds/frame for each data structure. As it might have been expected, a larger number of triangles in the model entails more rendering time per frame, but the increase in rendering time is not linear with the increase in the number of triangles. In order to analyze the comparative evolution of rendering time in relation to the number of triangles for each city model, each column of Table 2 was divided by the value of the first column (C1). Figure 10 plots the results for the BqR-Tree. The yellow line (top line) represents just the number of triangles of each city divided by the ones in C1. The blue line (bottom line) represents the average value of BqR-Tree's rendering time in relation to the C1 value. As it can be observed:

- Model C2 has 4.6 times as many triangles as C1, but the rendering time of BqR-Tree is only 2.3 times greater than C1's (exactly the half).
- Model C6 has 6.5 times as many triangles as C1, but the rendering time is only 3 times greater than of C1's (less than the half).

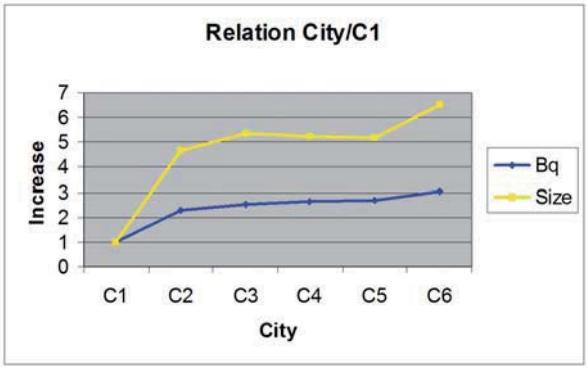


Figure 10: *Number of triangles and rendering times in relation to C1 for the different city models (using the BqRT)*

So, the increase in rendering time is much lower than the increase in the number of triangles; the separation between the two lines even increases when the number of triangles does likewise.

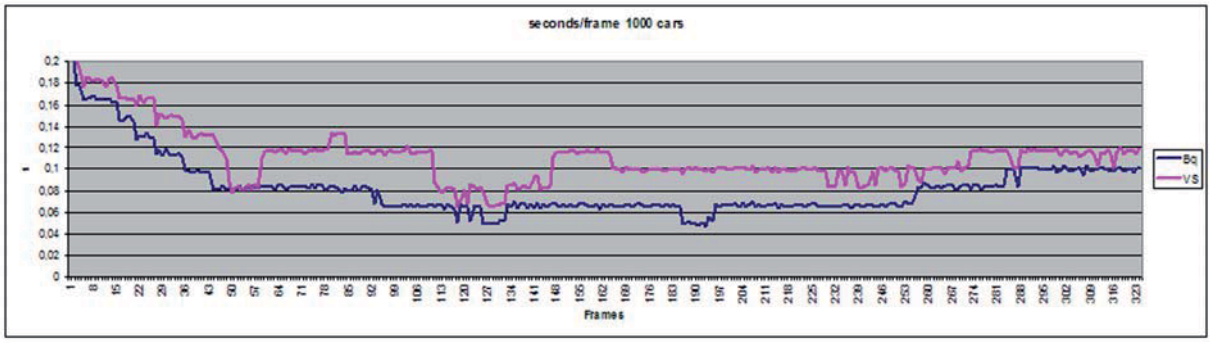


Figure 11: *Evolution of rendering time for each structure with 1,000 cars, each car is composed of 387 triangules.*

4.3. Tests with Mobile Elements

Cars have been used as mobile elements for tests purposes. The car model used is composed of 387 triangles.
The cars' geometry is stored in an independent node, to which neither LOD nor backface-culling is applied. Tests are implemented in the previously mentioned C2 city model, and cars are initially randomly situated on the streets of the city. In every frame every car is displaced a fixed amount. Tests with cars attached to the BqR-tree data structure and the previously mentioned VamSplit-Rtree data structure have been performed.
The evolution of the time required (in seconds) to render each frame for each of the previously mentioned two data structures with 1,000 cars moving in the city can be seen in Figure 11: where the red line (middle line) corresponds to the VamSplit R-Tree structure, and the blue line (bottom line As shown in Figure 11, many little peaks appear for both lines. This is caused by the re-connection of the MEs. However, after the addition of MEs, the differences between both data structures remain equal. In Figure 12, shows average rendering times plotted for both data structures with 100 and 1,000 cars moving in the city. As shown in the figure, the differences between both data structures remain after the addition of MEs. In order to improve rendering times it is necessary the addition of other acceleration techniques. The usual solution is the implementation of LODs. Therefore, the same previous tests have been performed with cars with different levels of details: the models range from 18 triangles to 1,701 triangles. The evolution of the average rendering times for both data structures when increasing the number of cars is plotted in Figure 13. It is easy to realize that the differences between both data structures persist although the absolute rendering time has decreased in both cases due to the implementation of LOD.

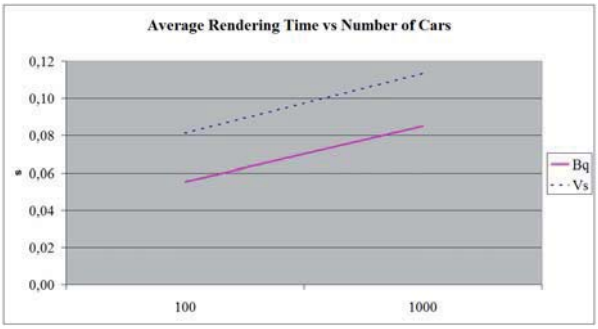


Figure 12: *Evolution of the average rendering time (in seconds per frame) with the number of cars in the city for both structures (Vs dotted line: VamSplit-Rtree, Bq continuous line: BqR-tree)*

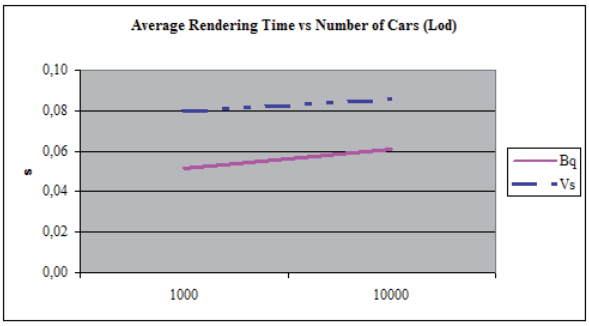


Figure 13: *Evolution of the average rendering time of both data structures (Vs dotted line: VamSplit-Rtree, Bq continuous line: BqR-tree) when increasing the number of LOD cars*

5. Conclusions and Future Work

We have presented a data structure, the BqR-Tree. This structure is especially suited to urban environments and its main features are:

- It is defined by considering the city block as the basic and logical unit. The advantage of the block as opposed to the traditional unit, the building, is that it is easily identified

regardless of the data source format, and allows the inclusion of mobile elements in a natural way.
- The usefulness of the structure has been tested with low structured city data, which makes its application appropriate to almost all city data.
- The results of the tests show that when using the BqR-Tree structure to perform city walkthroughs and flights (with or without mobile elements populating the city) rendering times improve an average of 30% in comparison to the data structures which have yielded best results to date.

Regarding next steps, the idea is:

- To perform semantic view culling, ie, to take into account semantic information when performing the view culling.
- To implement relief impostors based on blocks or quadtrees, because both techniques allow a direct implementation on the BqR-tree data structure.

6. Acknowledgements

This work has been partly financed by the Spanish Dirección General de Investigación, contract number TIN2007-63025 and by the Government of Aragon by way of the WALQA agreement.

References

[ABJN85] AYALA D., BRUNET P., JUAN R., NAVAZO I.: Object representation by means of nonminimal division quadtrees and octrees. *ACM Trans. Graph. 4,1* (1985), 41–59.

[ADB08] ANDUJAR C., DÍAZ J., BRUNET P.: Relief impostor selection for large scale urban rendering. *IEEE Virtual Reality Workshop on Virtual Citiscapes: Key Research Issues in Modeling Large-Scale Immersive Urban Environments* (2008).

[AM00] ASSARSSON U., MOLLER T.: Optimized view frustum culling algorithms for bounding boxes. *J. Graph. Tools 5*, 1 (2000), 9–22.

[BWPP01] BITTNER J., WIMMER M., PIRINGER H., PURGATHOFER W.: Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum Proceedings of EUROGRAPHICS 2004* (2001), 615–624.

[CM99] CHAKRABARTI K., MEHROTRA S.: The hybrid tree: An index structure for high dimensional feature spaces. *In Proc. Int. Conf. Data Engineering* (1999), 440–447.

[CMG*07] CIGNONI P., M D. B., GANOVELLI F., GOBBETTI E., MARTON F., SCOPIGNO R.: Ray-casted blockmaps for large urban models visualization. *Computer Graphics Forum 26 3* (2007), 405–413.

[COCSD03] COHEN-OR D., CHRYSANTHOU Y., SILVA C.-T., DURAND F.: A survey of visibility for walkthrough applications. *IEEE Transaction on Visualization and Computer Graphics 9 3* (2003), 412–431.

[DB05] DOLLNER J., BUCHHOLZ H.: Continuous level-of-detail modeling of buildings in 3d city models. *Proceedings of the 13th annual ACM international workshop on Geographic information systems* (2005), 173–181.

[DHOO05] DOBBYN S., HAMILL J., O'CONOR K., O'SULLIVAN C.: Geopostors: A real-time geometry/impostor crowd rendering system. *In Proceedings Of Symposium On Interactive 3d Graphics And Games* (2005), 95–102.

[EMB01] ERIKSON C., MANOCHA D., BAXTER W.: Hlods for faster display of large static and dynamic environments. *SI3D 01: Proceedings of the 2001 symposium on Interactive 3D graphics* (2001), 111–120.

[FS93] FUNKHOUSER T., SEQUIN C.: Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. *Computer Graphics 27*, Annual Conference Series (1993), 247–254.

[GBSF05] GRUNDHÖFER A., BROMBACH B., SCHEIBE R., FRÖHLICH B.: Level of detail based occlusion culling for dynamic scenes. *In Proceedings of the 3rd international Conference on Computer Graphics and interactive Techniques in Australasia and South East Asia,. GRAPHITE 05. ACM* (2005), 37–45.

[GG98] GAEDE V., GUNTHER O.: Multidimensional access methods. acm comput. surv. *Computer Graphics 30 2*, Annual Conference Series (1998), 170–231.

[KS97] KATAYAMA N., SHINICHI S.: The SR-Tree: an index structure for high-dimensional nearest neighbor queries. *Proceedings of ACM SIGMOD* (1997), 369–380.

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast bvh construction on gpus. *Computer Graphics Forum 28,2* (2009), 375–384.

[Man08] MANOCHA D.: Real-time motion planning for agent-based crowd simulation. *Proceedings of 2008 IEEE Virtual Reality Workshop on Virtual Cityscapes* (2008).

[NHN06] NIRNIMESH, HARISH P., NARAYANAN P.: Culling an object hierarchy to a frustum hierarchy. *Springer Berlin 4338/2006, Computer Vision, Graphics and Image Processing* (2006), 252–263.

[Pla05] PLACERES F.: Improved frustum culling. *in Pallister, K. (Ed.). Game Programming Gems 5* (2005), 65–77.

[Sam90] SAMET H.: The design and analysis of spatial data structures. *Addison-Wesley Longman Publishing Co., Inc.* (1990).

[SMM08] SHAHRIZAL M., MOHD A., MOHD T.: Improved view frustum culling technique for real-time virtual heritage application. *The International Journal of Virtual Reality*, 3 (2008), 43–48.

[SRCP02] SERON F., RODRIGUEZ R., CEREZO E., PINA A.: Adding support for high-level skeletal animation. *IEEE Transactions On Visualization And Computer Graphics 8, 4* (2002), 360–372.

[Sto05] STOLL G.: Part ii: Achieving real time , optimization techniques. *Slides from the Siggraph 2005 Course on Interactive Ray Tracing* (2005).

[UCT04] ULICNY B., CIECHOMSKI P.-D.-H., THALMANN D.: Crowdbrush: Interactive authoring of real-time crowd scenes. *Proc. ACM SIGGRAPH Symposium on ComputerAnimation* (2004).

[Wac07] WACHTER C. A.: Quasi-monte carlo light transport simulation by effcient ray tracing. *geb. in Ochsenhausen Institut fur Medieninformatik, 2007, thesis* (2007).

[WIP08] WALD I., IZE T., PARKER S.: Special section: Parallel graphics and visualization: Fast, parallel, and asynchronous construction of bvhs for ray tracing animated scenes. *Comput. Graph. 32*, 1 (2008), 3–13.

[WJ96] WHITE D., JAIN R.: Similarity indexing: Algorithms and performance. *Proc SPIE Vol .2670, San Diego, USA* (1996), 62–73.

[XP03] XIA Y., PRABHAKAR S.: Q+rtree: Efficient indexing for moving object databases. *Proceedings of the Eighth International Conference on Database Systems for Advanced Applications* (2003), 175.