Progressive Transmission of Large Models by using a Bounding Planes Hierarchy.

F.J. Melero, P. Cano and J.C. Torres

Dept. Lenguajes y Sistemas Informáticos, ETSIIT, Univ. Granada, Spain

Abstract

In this paper we present a new data structure that makes it possible to transmit and visualize large three-dimensional models through a network, the BP-Octree (Bounding-Planes Octree). The BP-Octree is based on a spatial decomposition of the model using an octree, and offers a very tight hierarchy of convex bounding volumes which leads to simplified models. It is done by assigning to each node of the tree a set of planes that, intersected one among others, creates a very bounding volume of the part of the model contained in that node. These planes are taken from the real polygons of the model, and are selected at each level guaranteeing that they bound completely all boundings of deeper levels.

By using this scheme, the transmission of a model through a network is not achieved by sending the polygons, but using just the planes coefficients needed at each level, or just the index of each plane if it has been previously transmitted. On the client side, the visualization algorithm is the responsible for reconstructing the polygonal geometry from the set of planes just applying a mesh clipping algorithm.

Key words:
PACS:

1 Introduction

One of the fields that have not been yet completely resolved by the Computer Graphics community is the visualization of 3D models through internet. Several standards and file format have been proposed to represent three-dimensional objects (VRML, X3D, etc..), and many papers can be found in

Email address: fjmelero@ugr.es, pcano@ugr.es, jctorres@ugr.es (F.J. Melero, P. Cano and J.C. Torres).

the literature with techniques useful in remote visualization [1,2], but none of them has been widely accepted by the internet industry.

Computer graphics end users always require very detailed models, what makes the management of large datasets necessary. The interactive visualization of those complex models requires a great effort of the graphics hardware, due to the large number of polygons needed to represent the geometry at the expected level of detail and realism. This resource requirement may lead to a lack of interactivity. But if this problem is translated into a distributed system, it is neccesary to involve some other techniques of progressive and adaptive visualization in order to optimize as much as possible the bandwith usage.

The conflict between the rendered level of detail and the visualization speed has motivated the development of several techniques to reach both goals. Under the generic name of level-of-detail (LOD) techniques we gather techniques that represent a complex geometry in a simplified way when the observer is not close enough to the object.

Geometry-based LOD techniques can be classified as: *discrete* [3], i.e. the object is represented in several instances, each one at different LOD; *progressive* [1], i.e. the detail is extracted from an unique data structure during runtime; and *view-dependent* [4], which are an extension of progressive techniques, in a way that the LOD is not uniform along the object, it is *anisotropic* depending on the point of view.

Another approach to represent volumes and solids with variable LOD is to represent them with schemes based on spatial decomposition, using hierarchical structures. Among these methods we can find the *Octree* and several extensions to these (Extended-Octrees, PM-Octrees, SP-Octrees [5–7]).

All these techniques are successfully used in progressive or adaptive visualization, not only for remote visualization, but also for standalone applications, solving the limitations on memory and processing time. Relevant work on this area can be found in [8–11].

Our proposed data structure is based on an octree, assigning to each node a set of planes that form a convex bounding volume of the part of the model contained in it (see a detailed node in figure 1, and two different bounding levels in figure 2). These planes are restricted to be face planes or planes parallel to faces, in order to avoid including more geometric information. That is why it is named *BP-Octree* (Bounding-Planes Octree).

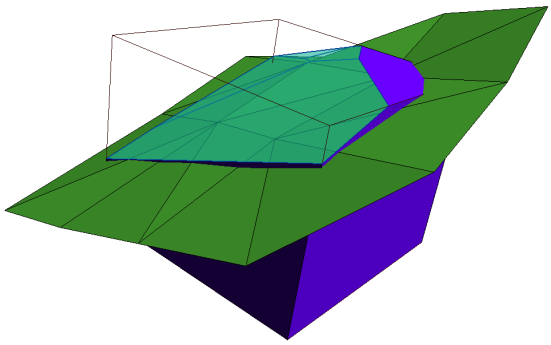
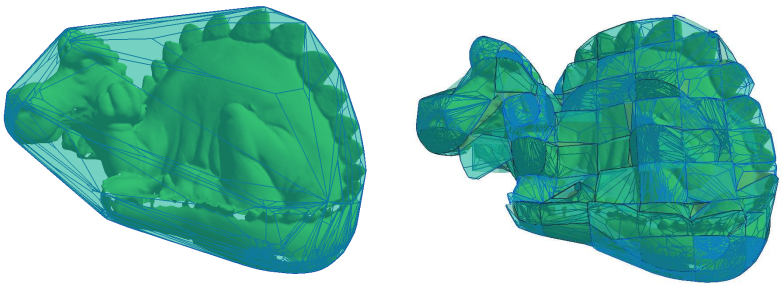


Fig. 1. Example of node. Green, model geometry; cyan, computed bounding.



(a) (b)

Fig. 2. Phlematic Dragon and its bounding planes (cyan) at root node (a) and at level 4 of the BP-Octree (b).

2 BP-Octree. Data Structure Overview

As told before, in each node we store a set of planes whose intersection half spaces -intersected with the cell of the node- include the part of the model that falls in the octree cell. At leaf nodes, in addition, is stored the final geometry that was initially assigned to them. Pseudocode of the data structure is shown in figure 3 and a general schema of the whole data structure is illustrated in figure 4. Figure 5 shows the root node of the BP-Octree for several models.

As planes we want to restrict us to planes of the polygons of the model wherever possible. This works fine when the model restricted to the cell forms a convex geometry, but it is not usually the case. When having a non pure convex geometry, new planes have to be introduced. Then, to avoid defining new planes, we use planes that are offsets to faces of the model (field d of the `BPlane` structure is just that offset).

At each node we store just the indices of the planes, so it means that all the

```
typedef long int Index;
typedef struct {
     Index planeIndex;
     double d;
} BPlane;
typedef long int Octcode; // 4 bytes long

class BPNode {
    vector<BPlane> boundingPlanes;
    Octcode oct;
}

class BPLeafNode:public BPNode {
    vector<Index> faces;
};
```

Fig. 3. Basic Data Structure of our BP-Octree

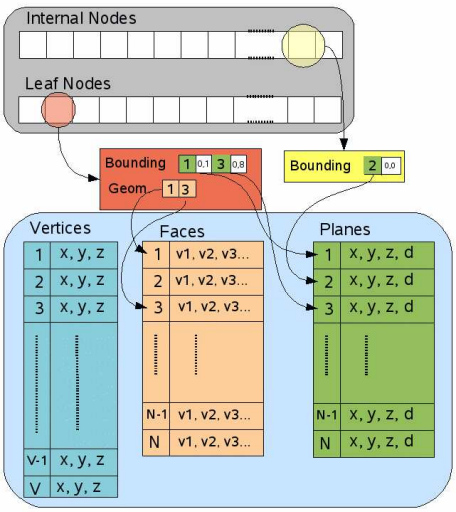


Fig. 4. General Scheme of BP-Octree data structure.

planes are stored in an external data structure, accessed by the octree. By doing so we save space and eliminate redundancies. The same idea applies to the final geometry stored at leaf nodes. When using huge models, this approach will make it easier to have the data structure in main memory and to access the planes and geometry information via external files, delegating into the operating system the caching procedure of these data.

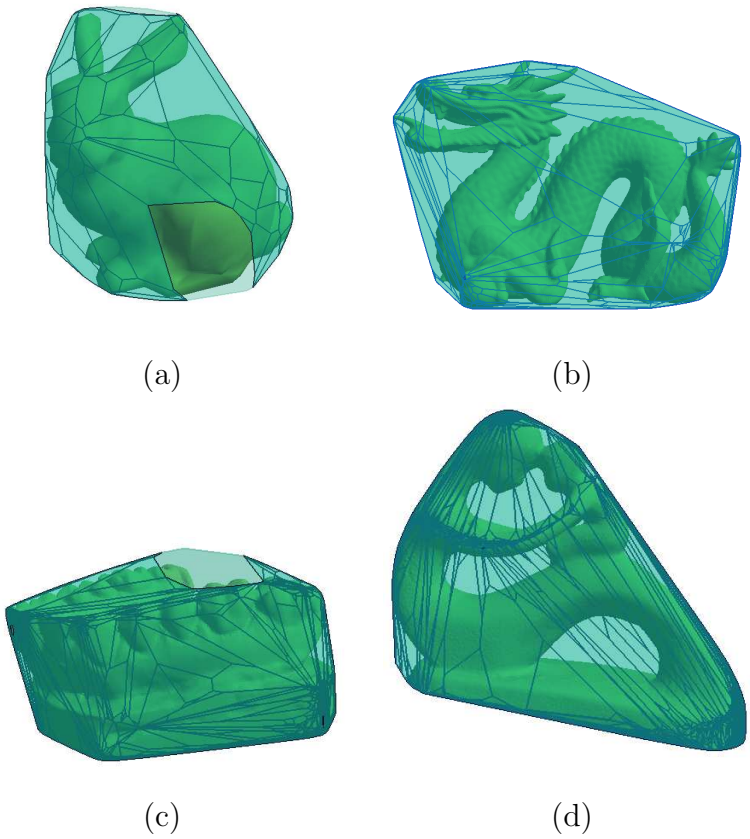


Fig. 5. Boundings at level 0.

Another goal of this work is to achieve the building of the whole data structure in a reasonable computation time. This is achieved by using a bottom-up approach: first, we classify the polygons in a three-dimensional grid, which is the deepest level of the octree; then, these polygons are assigned to each leaf node; then, all leaf nodes are proccessed in order to have a minimum number of planes, and then their bounding planes are selected, and finally, internal nodes are built by following the path from the root node to each leaf, computing their bounding planes in a bottom-up recursion

When designing the data structure, several problems arose that required our attention in order to get a complete and correct solution:

- *Management of large datasets.* Our data structure should be able to deal with large models. Our goal was not only to optimize space consumption by using a hierarchical representation, but also reduce building time. To solve this, we have used a spatial indexation of the polygons, making easier to address and classify the polygons and also the management of such amounts of data via external memory.
- *Guarantee that every polygon contained in the node is used when computing its bounding.* As the octree is a discrete environment, we have to assign

continuous primitives (polygons) to voxels, and do it quickly enough. The obvious solution would be to clip all polygons to fit them exactly into our three-dimensional voxels, but this would lead to prohibitive computation time. We use a 3DDDA algorithm to traverse each polygon and determine which voxels are containing it.

- *Ensure that computed boundings at level* n *are completely inside of bounding at level* n-1. It is important to keep coherence between levels, because it makes no sense to be in a given level of the BP-Octree and when asking a more detailed level, we got a less tight bounding. To achieve this, we compute the bounding at leaf nodes by using the real geometry of the model, but for internal nodes, we use the geometry generated by the bounding planes of their children nodes.
- *Avoid cracks between adjacent nodes.* As we use approximated bounding, it is almost impossible that two adjacent nodes get identical bounding plane at their shared face, so when rendering it, user would see a hole in the surface. We add a set of fictitious planes, which are never transmitted, corresponding to the node bounding box faces that are still visible (see purple polygons in figure 1).
- *Which plane to use when polygons in the node create a concavity?.* When node geometry is not a pure convex geometry, it is rather complex to find a face whose plane bounds the whole geometry. Then, we allow these planes to be translated along its normal direction in order to become a bounding plane. We name them *displaced planes.*
- *Convex meshes get less simplification rate than irregular ones.* If we have a sphere as object, all its face planes satisfy the bounding criterion, so we would get as bounding the sphere itself. To avoid such situations, we limit the number of planes at each level, and these are selected by using an statistical criterion. An example of this situation can be found in figure 6

2.1 Spatial Indexation

Being the octree a discrete structure, and the 3D space a continuous environment, it is clear that the first task to do is to discretize our model, i.e., to determine which nodes are intersected by the polygons. This is done by using an octcode, computed as a traditional *Morton code* [12].

The root node of the octree is an *Axis Aligned Bounding Box* [13], i.e. it is not necessarily a cube, but it has its eight faces axis aligned. It is computed while loading the model, and it is defined by two significant points: $BB_{min} = (x_{min}, y_{min}, z_{min})$ and $BB_{max} = (x_{max}, y_{max}, z_{max})$.

The basic step to index all polygons is to determine the cell of the discrete grid which a given point p belongs to. For each dimension D, at a given level l

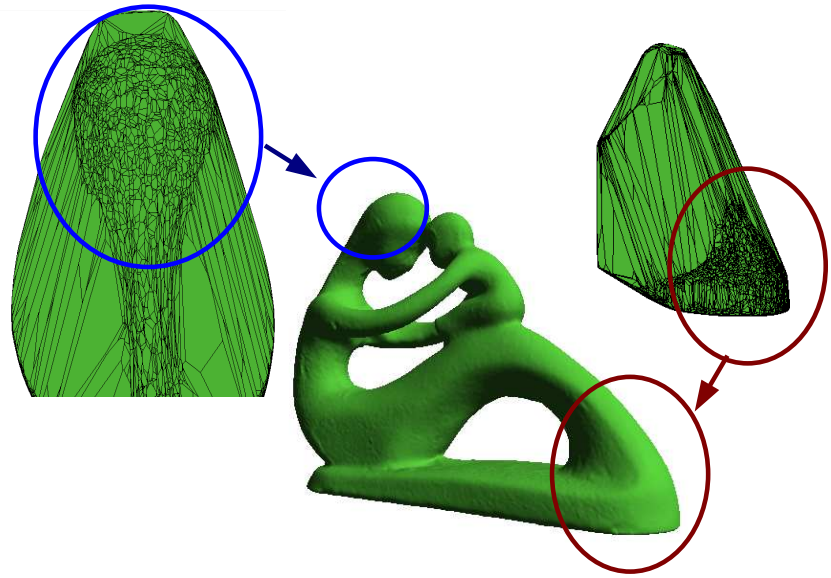


Fig. 6. High density of planes due to strong convex bounding areas of the model.

the discrete coordinate N_D is computed as described by the following formula:

$$N_D = floor(\frac{2^l}{w_d}(d_p - d_{min})) \tag{1}$$

where w_d is the length of the root node at that dimension, d_{min} is the minimum value of that node for the dimension D and d_p is the coordinate at dimension D of the point p.

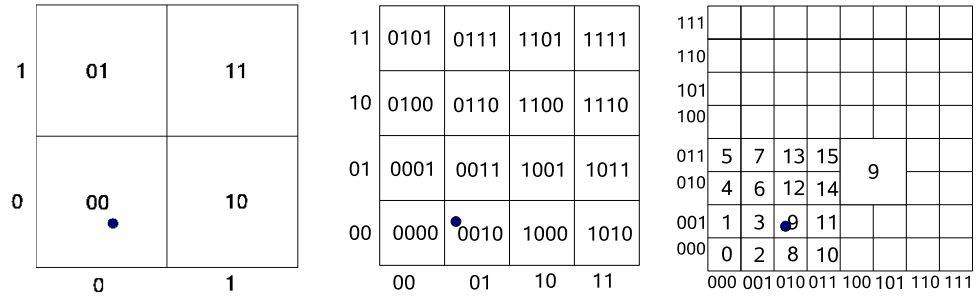
2.1.1 Morton Code

The *Morton Code* of a given voxel is computed by interleaving the discrete coordinates (expressed in binary) of any of the points that belong to that voxel.

In figure 7 we can see in 2D how for each node X and Y coordinates are interleaved, resulting in an integer number, unique for each voxel in its level, but not unique among all levels. In level 3 we can see that the number 9 can identify a cell in 2nd or 3rd level, being both of different sizes and location.

To solve this ambiguity, we append extra information to the *octcode* about the level that it belongs to, also in binary. It converts the Morton Code into a locational code. There are two options:

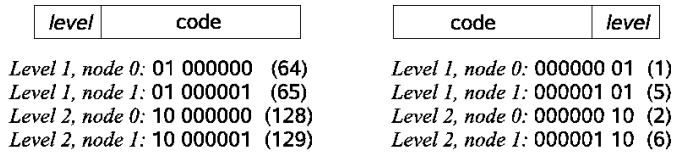
- Append the level bits as the most significant. (figure 8a). This causes a breadth-first ordering of the octcodes.
- Append the level bits as the less significant (figure 8b). This causes a depth-



(Level 1) (Level 2) (Level 3)

Fig. 7. Morton-code based nodes numbering

first ordering of the octcodes.



(a) (b)

Fig. 8. Morton-based locational code: a) Depth-first ordering; b) breadth-first ordering

In our approach, we have chosen the breadth-first ordering of the octcodes, because when saving them into a file, this sorting keeps the spatial coherence for a progressive transmission of the model.

2.1.2 Defining Maximum Depth

Given the locational code expressed as before, we can easily calculate the memory needed to address each voxel. The octcode can be splitted into two parts: the level and the cell index related to that level (figure 8). Each depth level l adds three bits to the cell index (so it is $3l$ bits), and the level l is represented with $log_2 l$ bits.

Therefore, with 4 bytes we can have $2^{27} = 134217728$ leaf nodes, i.e. more than 134 million leaf nodes. Up to now, models over 1.5M polygons have never used more than 5% of maximum depth nodes. So we could assume that 4 bytes, a `long int` in most of the implementations, is a safe code length for such models.

To identify which polygons belong to each node, we run a 3DDDA algorithm [14] over each polygon, identifying whose voxels are traversed by the polygon by computing the *octcode* as shown in equation 1. Later on, this polygon is taken into account when computing the bounding volume at those traversed nodes.

By applying this algorithm, we guarantee that all nodes traversed by the polygon will use it to compute the bounding volume. If we were not restricted to have at each node a bounding volume of the model, it would be faster and less memory consuming to select just a small subset of these traversed nodes, e.g. by selecting only the nodes where the vertices lie.

3 Building the Octree

After indexing the polygons, we can determine exactly the leaf nodes that our octree will hold. First we create all leaf nodes, identified by their *octcode*, and attach to each leaf node the index of each polygon that traverse it.

It is possible to reduce the number of leaf nodes just grouping leaf nodes until a given criterion is fulfilled. Some of these criteria might be:

- Group while new leaf node has less than N polygons assigned, or
- Group until new leaf node has more than M polygons assigned, or
- Group until average number of polygons at brother nodes is over X, etc...

By choosing any of these options, we get a significant reduction in the number of leaf nodes, up to 85%. In the experiments shown in this work we have used the criterion *group while new leaf node has less than 50 polygons*. Taking one or other criterion only affects the lower levels of the octree, being not relevant at top levels. The selected criterion gives an average of about six polygons per leaf node in all models.

It is simple, by looking at the *octcode* of each leaf, to extract the path from the root to the leaf node, and create the internal nodes that are necessary to reach the leaf depicted in figure 9.

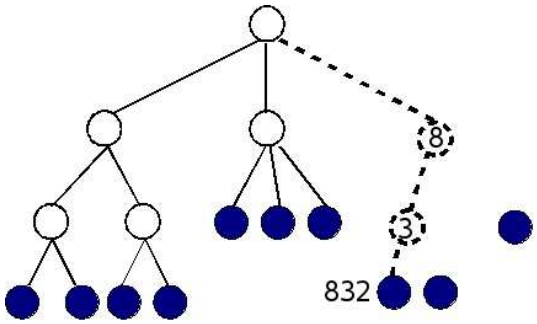


Fig. 9. Building octree

4 Computing Bounding Volumes

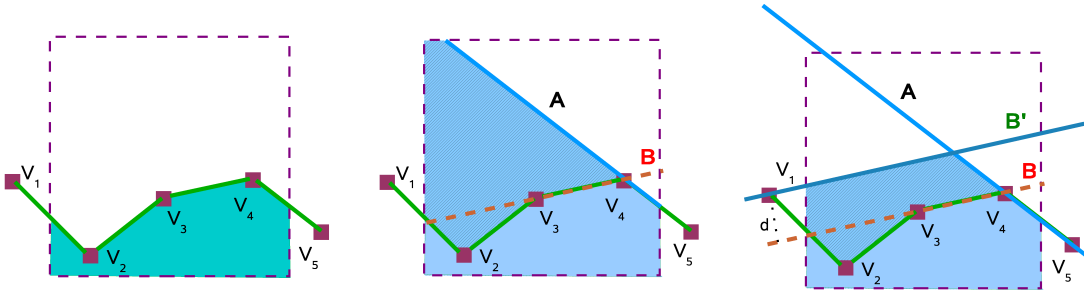
At each node we store a set of planes that includes completely the portion of the 3D model contained in the node. To compute this set, we follow a recursive bottom-up procedure, so at each node just a small number of computations is performed, i.e. we follow the divide-and-conquer paradigm.

As is briefly described in figure 10, at each node we select just the planes that bound the geometry of its children (or the real geometry of the model contained in the node if we are at a leaf) using the `addBVPlanes` method. Furthermore, only planes belonging to descendant's boundings are used to compute the current node bounding , and only bounding vertices of its descendants (`addBVVertices`) are used to guarantee that the bounding is increasing in volume as we ascend through the octree.

The method that we follow to compute the bounding is quite simple, as shown in 2D in figure 11. For each node n we create a set of *candidate* planes, that are those that were selected as bounding planes at children of n (or the original geometry in case we are in a leaf node). The, we test sequentially every candidate plane against contained bounding vertices. The plane is taken as it

```
computeBounding(Node_T node){
 if isLeaf(node)
  node->selectBoundingPlanes();
 else {
  for each ch child of node {
   computeBounding(ch);
   node->addBPlanes(ch.getBPlanes());
   node->addBVertices(ch.getBVertices());
  }
  node->selectBoundingPlanes();
 }
}
```

Fig. 10. Computing recursively the bounding at each node



(a) (b) (c)

Fig. 11. Computing boundings. a) Node, b) bounding without displaced planes, c) bounding with B'

was in the child node, using the displacement value. If all vertices lie inside or on the plane, i.e., signed distance is less or equal to zero, the plane belongs to the bounding as it is. This is the case of plane A in figure 11.b, which is the only plane that bounds all the vertices. The resulting bounding volume is the cyan one.

As we search for a closer bounding, in figure 11.c is shown that we can insert into the set of bounding planes the plane B but displaced until all vertices lie inside. Note as vertex V_1 initially lies outside B (11.b), but displacing the plane to B', it is now in the inner halfspace. The distance d is exactly the distance from V_1 to B.

Each relevant plane -displaced or not- is selected as part of the bounding. The key point is to define what is a *relevant plane*. We could consider that all bounding planes are relevant enough, but this would lead to have no advantage of our method when using it with completely convex models. So, we have decided to use a classification method to determine which planes best describe the original surface. This is done by using a k-medoids algorithm [15] over the plane normals. We simplify the problem running the algorithm over a 2D space, representing the planes just by their normals in polar coordinates. By using this method, planes with similar orientation are discarded, selecting just the most relevant one, and its guaranteed that at most k planes are selected and that they are homogeneously distributed in the 2D normal space.

In table 1 are shown building times for several models. Fertility model is built in more time that Angelo, although it has 80K polygons less. It is due its very convex and smooth surface, which makes a lot of planes to be selected and hence propagated to the upper levels.

Model	Polygons	Secs.
Bunny	69.451	23,16
Dragon	202.520	55,72
Teeth	233.204	86,75
Igea	268.686	127,88
Fertility	483.226	171,51
Angelo	562.879	131,12
Phlegmatic Dragon	715.933	197,24

Table 1
BP-Octree construction time for several models.

Level	Bunny	Dragon	Fertility	Phlegmatic
0	391	529	1777	743
1	1225	1658	3267	2166
2	3087	4230	7418	6031
3	7120	10822	18691	17663
4	16019	27637	42056	43295
5	33502	61272	91574	102967
6	46534	116649	185686	230607
7	-	52331	296797	423267
8	-	67	3399	144246

Table 2
Number of planes at each level of the BP-Octree.

5 Progressive Transmission of Planes

The intrinsic hierarchical feature of the BP-Octree makes easy to use it to achieve a progressive transmission of the model through Internet, by start sending planes from root level or a lower one, depending on the available bandwith.

In table 2 is shown the number of planes stored at each level for four representative models: Stanford Bunny (69K polygons), Stanford Dragon (202K polygons), Fertility (483K polygons) and Phlegmatic Dragon (715K polygons). It can be seen in table 3 as at first level is used less than 0.50% of total planes, and in figure 12 is shown as with around a 10% of planes we obtain very tight approximations.

Level	Bunny	Dragon	Fertility	Phlegmatic
Level 0	0,56%	0,26%	0,37%	0,10%
Level 1	1,16%	0,54%	0,30%	0,19%
Level 2	2,45%	1,20%	0,82%	0,52%
Level 3	5,05%	2,90%	2,17%	1,52%
Level 4	10,27%	7,11%	4,35%	3,24%
Level 5	19,11%	13,24%	8,65%	7,21%
Level 6	19,82%	21,08%	15,39%	14,57%
Level 7	-	7,77%	20,57%	22,75%
Level 8	-	0,01%	0,22%	7,27%

Table 3
Percentage of total number of planes that are used at first time at each level.

We remark that the number of planes at one level does not mean the transmission of four floating point coefficients per plane, plus its offset value. Most of them have been used at upper levels or appear more than once at that level, so in that cases just an index value and the offset are sent.

We have dealed in this section only to bounding-planes transmission, i.e. assuming that the final geometry is never transmitted, but a very fine approx-

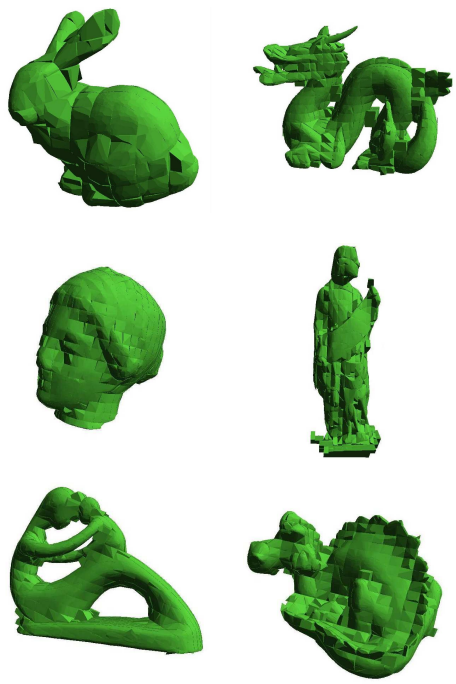


Fig. 12. Several models at level 4, using less than 10% of planes.

imation of it is reached at leaf nodes by their bounding. If we wanted to transmit the final geometry, each polygon is only transmitted the first time it is found in a leaf node. Then, at client side, it is assigned to every leaf node where it will be used again, as described in subsection 2.2. This means that we do not transmit any redundant data over the net, and that our structure overhead is enough to get at early stages a good approximation of the model.

In figure 16 we show the appearance of two models at each level of the BP-Octree, and the accumulated volume of data neccesary to reach each level is displayed in figure 14. It can be appreciated as transferring up the sixth level of the multiresolution hierarchy is equivalent to send the original full detailed model, and looking at figure 16 is shown that at first levels we obtain an acceptable image quality.

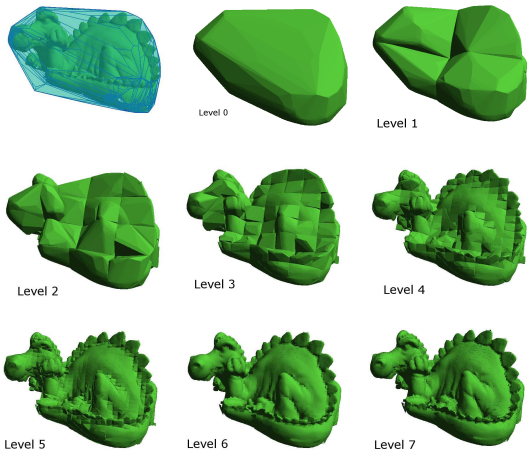


Fig. 13. Phlematic dragon at different levels of the BP-Octree.

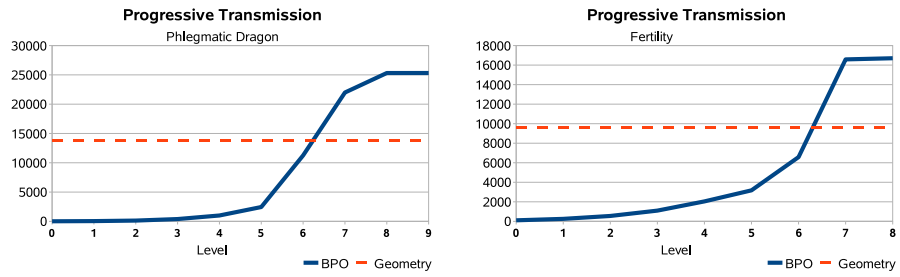


Fig. 14. Accumulated byte count for the progressive transmission of Fertility and Phlegmatic Dragon. It is displayed also the size of the full geometry model.

6 Adaptive Visualization

Underlying BP-Octrees data structure allows to adaptively render the models. The criteria to decide if descend or not in a given node are not fixed, being possible to use the observer distance to the object of any other criterion such as the screen size of the node.

figures in 15 have been taken using the following criterion: each node is displayed at most as 40 pixels width. At foreground, we can recognize the original geometry at maximum level of detail, while in background are shown low resolution nodes.

7 Conclusions and Future Works

We have developed a novel data structure that handles large polygonal models, regarless of whether they have holes or not, of whether their faces are triangles or polygons. BP-Octree construction is achieved in a reasonable timing, taking into account that it is done just once per model, and planes are arranged through levels in such way that transmission is done progressively and depending on the observer.

It is still needed to improve the visualization of the hierarchical model, either using the original normals of the model or using image based rendering techniques, as impostors [16].

Among other applications of this data structure, we are working on the collision detection applyied to haptic devices and speeding up raytracing algorithms.

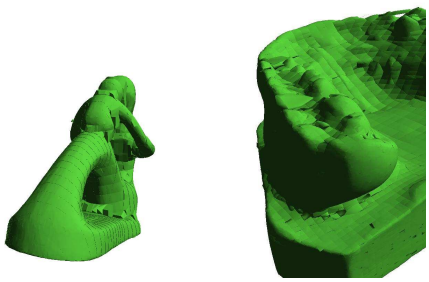


Fig. 15. Adaptive visualization of Fertility and Teeth.

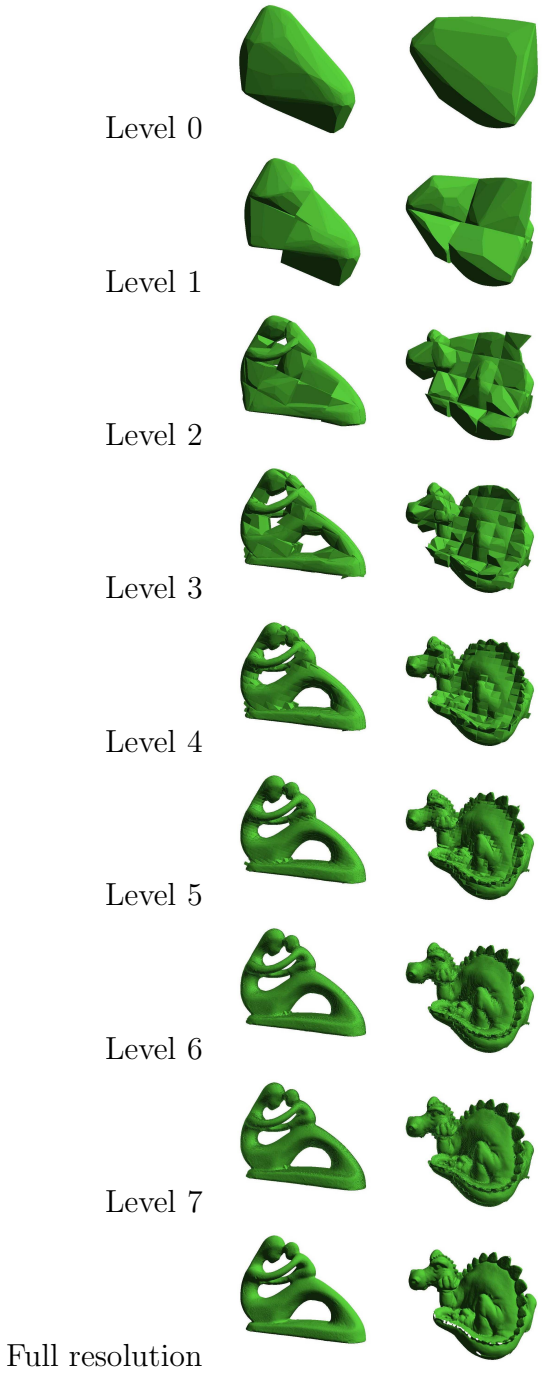


Fig. 16. Progressive transmission of Fertility and Phlegmatic Dragon models.

8 Acknowledgments

This work has been partially funded by the Spanish Ministry of Science and Technology and EU FEDER funds (projects TIN2004-06326-C03-02 and TIN2007-67474-C03-02) and by the Andalusian Ministry of Innovation, Science and Enterprise (project PE-TIC-401).

Bunny and Dragon models are courtesy of Stanford Univ. Computer Graphics Lab. Phlegmatic Dragon is courtesy of Czech Sciences Academy and Czech TU. Fertility, Igea y Teeth models are courtesy of Aim@Shape Repository. Angelo model is courtesy of VIHAP3D project.

References

[1] H. Hoppe, Progressive meshes, in: SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, ACM Press, New York, NY, USA, 1996, pp. 99–108.

[2] I. M. Martin, Arte, an adpative rendering and transmission environment for 3d graphics, in: MULTIMEDIA '00: Proceedings of the eighth ACM international conference on Multimedia, ACM Press, New York, NY, USA, 2000, pp. 413–415.

[3] J. H. Clark, Hierarchical geometric models for visible surface algorithms, Commun. ACM 19 (10) (1976) 547–554.

[4] H. Hoppe, View-dependent refinement of progressive meshes, Computer Graphics 31 (Annual Conference Series) (1997) 189–198.

[5] I. Carlbom, I. Chakravarty, D. Vandershcel, A hierarchical data structure for representing the spatial decomposition of 3d objects, IEEE Comp. Graphics and Applications 5 (4) (1985) 24–31.

[6] D. Ayala, P. Brunet, R. Joan, I. Navazo, Object representation by means of nonminimal division of quadtrees and octrees, ACM Transaction on Graphics 4 (1).

[7] P. Cano, J. Torres, F. Velasco, Progressive transmission of polyhedral solids using a hierarchical representation, Journal of WSCG 11 (1) (2003) 81–86.

[8] B. Chamberlain, T. DeRose, D. Lischinski, D. Salesin, J. Snyder, Fast rendering of complex environments using a spatial hierarchy, in: W. A. Davis, R. Bartels (Eds.), Graphics Interface '96, Canadian Human-Computer Communications Society, 1996, pp. 132–141.

[9] M. Callieri, P. Cignoni, F. Ganovelli, G. Impoco, C. Montani, P. Pingi, F. Ponchio, R. Scopigno, Visualization and 3d data processing in david's restoration, Tech. rep., ISTI, CNR, Pisa, Italy (Jan. 2004).

[10] C. Erikson, D. Manocha, I. William V. Baxter, Hlods for faster display of large static and dynamic environments, in: SI3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics, ACM Press, New York, NY, USA, 2001, pp. 111–120.

[11] P. Cignoni, C. Montani, C. Rocchini, R. Scopigno, External memory management and simplification of huge meshes, IEEE Transactions on Visualization and Computer Graphics 9 (4) (2003) 525–537.

[12] G. Morton, A computer oriented geodetic data base and a new technique in file sequencing, Tech. rep., IBM Ltd. (1966).

[13] V. D. Bengen, Efficient collision detection of complex deformable models using aabb trees, Journal of Graphics Tools 2 (4) (1997) 1–13.

[14] A. Fujimoto, T. Tanaka, K. Iwata, ARTS: accelerated ray-tracing system, Computer Science Press, Inc., New York, NY, USA, 1988, pp. 148–159.

[15] L. Kaufman, P. J. Rousseeuw, Finding Groups in Data – An Introduction to Cluster Analysis, John Wiley & Sons, 1990.

[16] F. Melero, P. Cano, J. Torres, Combining sp-octrees and impostors for multiresolution visualization, Computer and Graphics 29 (2005) 225–233.