Department of Computer Science

João Oliveira
Research Student

University College London
Gower Street
London WC1E 6BT UK

Tel: +44 (0)20 7679 3687
Fax: +44 (0)20 7387 1397
Email: Joao.Oliveira@cs.ucl.ac.uk

Technical Report Number

RN/05/13

An Efficient Octree For Interactive Large Model Visualization

João Fradinho Oliveira and Bernard Francis Buxton

June 17, 2005

An Efficient Octree For Interactive Large Model Visualization

João Fradinho Oliveira Bernard Francis Buxton

University College London

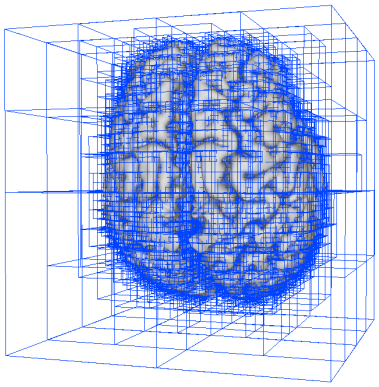

Figure 1. This image shows an extensively edited model of the human brain using our system. The interactive editing task was carried out over several sessions of approximately 40 minutes to one hour each, without using secondary memory or specialized graphics cards features. This was possible because of our system's novel compact data structures that kept the model in-core, and because of its automatic adaptation to the model's size for a comfortable synchronous rendering of 20-40 fps on a G4 500MGhz with an ATI 128 graphics card. The final model of 1.1 million triangles is shown being rendered progressively with the idle function.

ABSTRACT

As main memory continues to increase in size, larger 3D models can be loaded into memory. As these models grow in size, in spite of continuing improvements in graphics pipelines, they continue to present new challenges for the rendering hardware. Solutions for decreasing the load on graphics hardware, such as level of detail and vertex hierarchies, work well if enough main memory is present for the model at hand.

We present a new data structure (RenderArray) that allows load balancing with little memory overhead. The RenderArray allows fast display of any size model that fits in main memory, by only rendering the parts of the model that are near to the foremost intersection point of the line of sight with the model. An octree is used for this calculation, but in order to accommodate models near the main memory limit, without causing the system to page, we introduce a novel memory-friendly way of building a compact octree that does not store triangles at leaf nodes.

The RenderArray is computed to adapt to the size complexity of the models in order to meet a target frame rate for rendering. The frame rate may be re-set by the user and the RenderArray recomputed during use. Additionally, we use a standard level of detail technique to create a low-resolution version of the model (navigation skin), to assist navigation around the large model.

CR Categories and Subject Descriptors: E.1 [Data Structures]: Trees; I.3.6 [Computer Graphics]: Methodology and Techniques --- Graphics data structures and data types; I.3.8 [Computer Graphics]: Applications

Additional Keywords: mesh simplification

1 INTRODUCTION

Several algorithms have been designed to address/solve some of the technical issues raised by massive models that inherently don't fit in main memory [see e.g. Lindstrom [1], Cignoni et. al [2]]. Such methods have been successful in simplifying massive models, often by moving intermediate data structures out of core as well [Lindstrom and Silva [3]]. More recently asynchronous pre-fetching and caching of pre-calculated levels of detail/strips of surface clusters/parts, has allowed fast out-of-core, view dependent inspection of massive models [Quick-VDR; Yoon *et. al* [5], TetraPuzzles; Cignoni *et. al* [4]].

However, as RAM increases and new random access memory technologies are developed, there is a wealth of models[1], each of which fits in core memory that would benefit from direct raw visualization for inspection and editing, without using secondary memory. The balance of RAM capacity and rendering capability on many platforms is such that these models, though they are wholly resident in core memory, cannot be rendered at useful frame rates for interactive use. This is particularly important if one is carrying out an extensive edit of a model, say, over a period of hours, or in a museum setting, with several visitors using the same system throughout the day. Furthermore, accessing the model from disk in order to free computational resources for rendering frequently does not solve the problem as, until recently, out of core view dependent systems imposed a direct refinement cost on the rendering throughput [XFastMesh; DeCoro *et. al* [20]]. Asynchronous rendering and pre-fetching have now indirectly allowed high frame rates, making visualization GPU bound [Cignoni *et. al* [4]], by trading speed for visual quality. However, it is not clear what long periods of read/write operations derived from high quality refinement updates might do to a mechanical hard disk in the context of intensive applications. Reliable Raid hard disks can address this problem, but with a considerable price tag, limiting their widespread use.

The motivation for development of the system described here came from a request by colleagues interested in medical imaging to carry out a significant edit (25 000 triangles had to be user chosen and deleted) to create a mirror right hemisphere from the left hemisphere of a human brain. The resulting model is comprised of 1.1million triangles (Figure 1). For such an edit, we required interactive, stable rendering rates of at least 10-20 frames per second and surface connectivity data structures, such as edge,

[1] The 8 million triangle statue model of David, previously regarded as needing to be stored out-of-core on most platforms, currently fits in 1.5 GB of core memory (basic flat shaded geometry, no connectivity), with desktop machines such as Apple Computer's G5, and PC P4 shipping with 4 GB of main memory.

vertex-face relationships, that ultimately left no room in memory for fast rendering data structures such as vertex hierarchies and level of detail load balancing strategies.

Main Contribution: We present a new rendering algorithm for interactive display of large models of millions of triangles that fit in core memory. In the absence of main memory space for traditional rendering acceleration data structures such as vertex hierarchies, and more recent out of core structures such as clustered hierarchy of progressive meshes (CHPM, [5]), or hierarchical surface patches (TetraPuzzles, [4]), we achieve interactive rendering rates by directly accessing the original maximum resolution model via a compact RenderArray. This RenderArray is a one-dimensional array of pointers to nodes at different levels of an octree. This collection of pointers may be regarded as the active front of a vertex hierarchy of a view dependent system that spans the whole extent of the model. One of the key aspects that makes CHPM and TetraPuzzles successful is that the original scene/surface is represented as smaller manageable clusters/surfaces, and the hierarchies are built on these smaller structures, rather than being built at the fine granularity of the vertex level. Like the CHPM scene representation and the hierarchical tetrahedral volumes, our octree represents the entire model/scene in a spatially hierarchical manner, but our hierarchy accesses the same original resolution geometry at all levels.

Our system is view independent in that the pointers in the RenderArray remain the same for every frame. However, we are interested in rendering first the triangles, which are closest to the foremost intersection point of the line of sight with the model. To accommodate this rendering priority, in every frame, distances to all the octnodes to which the RenderArray points are computed and sorted accordingly. Since sorting and computing distances for all pointers may take appreciable computation time, an appropriate number of pointers that can be sorted in the frame time is created for the RenderArray.

Theoretically any large model that fits in core may be sorted in a coarse way by means of 8 octnode pointers from the root although in such a case, triangles from the first octnode would be rendered, and not necessarily those closest to the foremost intersection point of the line of sight with the model. However, our observation, illustrated in Figure 2, 3rd column from left, is that very quickly the octnodes start to have a spatial size granularity that allows an approximate rendering of the closest bucket of triangles to the foremost intersection point of the line of the sight with the centre of the model. Furthermore, with large models, we have found that it is feasible to use more than 8 pointers at a level in the tree deeper than the first to allow direct access to the triangles that are most relevant, i.e. close to the point of interest on the model, for example, for editing. At run time the RenderArray can be destroyed at the user's request and be recomputed using higher or lower octnodes in order to adjust the balance between the sorting and rendering frame rate.

Our system does not impose any computational burden for refinement updates because, when a user is editing a model up close, a few thousand of the original triangles fill the entire screen, making it unnecessary to consult a hierarchy. For assisting navigation of the model, we can render the octnode bounding boxes in wire frame as in previous systems [2], at the desired level of the tree, where more depth in the tree indicates the presence of more detailed geometry. Additionally we can render high-resolution geometry, together with a navigation skin, that is comprised of a very coarse level of detail of the model in wireframe mode. Surprisingly, we have found this two level of detail rendering methaphor to be quite useful in editing and navigating a model.

Other Contributions: Our second contribution is the compact octree that is generated in place, without extra temporary memory and without storing triangles at leaf nodes. Finally, our third contribution is that, as a side effect of the octree's in place sorting of the triangle order, we generate a mesh that is typically more coherent than the original which in turn is good for mesh streaming and compression [6]. This happens because we change the triangle order of a mesh according to successive sorts of finer grain regular lexicographical order of the octnodes rather than according to the longest diagonal of the bounding box. For example, as the first column of Figure 2 shows, the first 16128 triangles of the final octree's root node are locally coherent and their mutual proximity is apparent.

Advantages: Whilst we do not address massive models that currently only fit out of core, we believe that our system has some characteristics that make it very convenient for today and the future, as larger models migrate in-core.

1. **Large Models:** We show that our visualization system with a navigation skin, allows practical interactive visualization of large models that fit in core (e.g 1.7-8.8 million triangles) without using traditional rendering acceleration data structures that would not fit in core. We tested our system with a volumetric derived surface model of the brain, and both scanned and CAD models.

2. **Runtime performance:** We make no assumptions on the graphics card used. We achieved 8 fps by synchronous rendering in software of the 1.7 million triangle turbine blade model, at full screen (1152x768) without a graphics card on a G4 500.

3. **Compact octree:** Our in-place octree creation does not use extra temporary memory that could cause a system to page with large models. This can be useful in several other applications.

4. **Easy implemention**: The pseudo code in Figure 3 reveals the simplicity of our in-place octree.

Paper overview: We briefly review data structures that are related to our system in section 2 and present our system in section 3. In section 4, we describe an extension to our system that uses a navigation skin. Results are shown at the end of each of the corresponding sections. In section 5, we describe the application that motivated the design and implementation of our system. Finally we conclude in section 6.

2 RELATED WORK

Our system addresses the need for the interactive visualization of large models when the computational platform does not have the memory required for traditional rendering acceleration data structures. Whilst our non-photo realistic rendering system may be regarded as an inspection and editing tool, it builds on existing real-time rendering techniques.

2.1 Level of detail

Mesh simplification has been an active area of research ([7],[8],[9],[10],[11],[12]) for providing alternative, simpler representations of complex meshes, by optimizing a range of cost functions. Edge collapse operations are performed with increasing cost to produce either a discrete lower resolution mesh, or a history of collapses that can be used to transmit, or refine a mesh at run time. In section 4, we show how a single coarse mesh (the 'Navigation skin') rendered in wire-frame, combined with high resolution geometry can be an extremely useful metaphor for navigating and editing a complex object, without the memory overheads associated with other levels of detail or refinements. We believe that such a metaphor is not only useful for inspection/editing of large models, but for inspection and editing of more complex scenes, for example, for planning off-line CG rendering for film applications.

2.2 View dependent methods

The linear sequence of edge collapses in a level of detail simplification can be re-organized spatially into a vertex hierarchy (Xia *et al.* [13], Hoppe [14]) suitable for local refinement/coarsening updates ideal for view dependent rendering. Since querying individual vertex normals for view-dependent mesh refinement could be prohibitive, Luebke *et. al* [15] used view cones that represented the normal of a group of local vertices to reduce the number of queries at each frame. QSplat uses a hierarchy of normals stored in spheres as a more flexible way to tackle the view query load [16]. Recently view dependent out of core methods ([4],[5]) switch the resolution of entire sub-surfaces or clusters, by comparing the view parameters and stored pixel error at each element of the hierarchies. As noted above, this aspect of managing larger blocks of data has proved useful in our load balancing of larger or smaller octree nodes.

Correspondences and dependencies in a hierarchy can often take significant computational time to resolve. One variant of our system thus uses two octrees, one for the original object and the other for the navigation skin. In this setting, one does not wish to render parts of the navigation skin that would obstruct the high-resolution geometry of interest that, for example, is being edited. For this purpose, we developed a way of implicitly inferring the correspondences between the two octrees so as to not render obstructing parts of the navigation skin at no extra cost.

We implemented view frustum culling, but found that the user's changes of viewpoint were unpredictable when editing a model and consequently caused discomforting frame rate variations. Instead, we render only those triangles closest to the foremost intersection point of the line of sight with the model, until we have exhausted the rendering budget. This has the advantage that the rendered load may be assigned a constant budget for every frame.

2.3 Out of core methods

Lindstrom used out of core quadrics with a regular grid to cluster massive models of any size [1], Fei et. al [17] used curvature information derived from the quadric error [18], to adaptively re-sample the out-of-core mesh in the directions of highest curvature. Garland and Schafer [19] used quadric information to non-uniformly adapt a BSP tree and improve the quality of the reduced mesh. Lindstrom and Silva [3] moved temporary data structures out of core to make the output mesh memory insensitive, whilst Cignoni et. al [2] used wireframe octree bounding boxes to allow a user to select the part of the mesh to load in core. We similarly allow the user to preview and adjust the level of the scene's wire-frame bounding boxes so as to enable him/her to zoom to areas of more interest or detail. In their work, Cignoni *et. al* [2] also re-label vertex indices so that they belong to a range of indices defined in the octnode in which the vertices are contained. In contrast, our octree reorders only the triangles and keeps track of the number of sorted triangles within each octnode. This reordering of the triangles can lead to a more compact mesh suitable, for example, for streaming and compression, as illustrated in Figure 2. Quantitatively, it compares well with the spectral sequencing, or single axis vertex sorting of Isenberg *et al.* [6]. For example, the Stanford dragon initially has a *front width* of 1.05%, and after our triangle re-ordering it drops to 0.58% and compares to 0.18% (of spectral sequencing). It is fair to say that the original dragon had some coherence already, unlike models that are largely incoherent from the start such as the stanford bunny at 26.22% (1.9% after our triangle re-ordering and 0.78% with spectral sequencing). We plan to re-label our vertices, within octnodes, for example as in Cignoni et al.[2], to improve also on the *front span* of our re-ordered models.

DeCoro and Pajarola [20], presented a solution for out of core view dependent refinement. Recently ([4],[5]) decouple rendering from out of core fetching, a strategy which allows the graphics card to render at full speed always what is stored in the graphics card cache and prevents the graphics pipeline from stalling. Out of core updates are either inserted in a priority queue and phased-in, or are limited in scope and thus allow the rendering to be GPU bound. Our method does not assume graphics card capabilities, so switching the rendering context of our navigation skin to wire-frame typically incurs a 50% frame-rate reduction. However, since the navigation mesh is typically comprised of not much more than a thousand triangles and the rendering budget that fills the screen is roughly 4000 triangles, we can, for example, render the turbine blade model in software at 1152x768 pixels on a G4 500MGhz at 8 fps. Using the graphics card at 800x600 pixels allows rendering of the same model at an average of 35-40 fps without the navigation skin, and half that speed with the wire-frame skin.

2.4 Octrees

Spatial data structures such as the octree have long been useful for tasks such as fast distance queries (see Samet [21]). The way in which they are built, however, can affect their application and robustness. One widely used method of octree construction in out of core methods is first to scan all the vertices of a model one by one to find the dimensions of the model, and then either create files of leaf node triangles [2], or perform several external memory sorts on vertices or on fundamental quadrics of each octnode [3].

The classical way of creating an Octree in main memory, and indeed of creating other space division trees such as a BSP tree, involves creating and deleting temporary memory to reassign triangles to child nodes. This can easily cause a system to page when sub-dividing the root of a large scene.

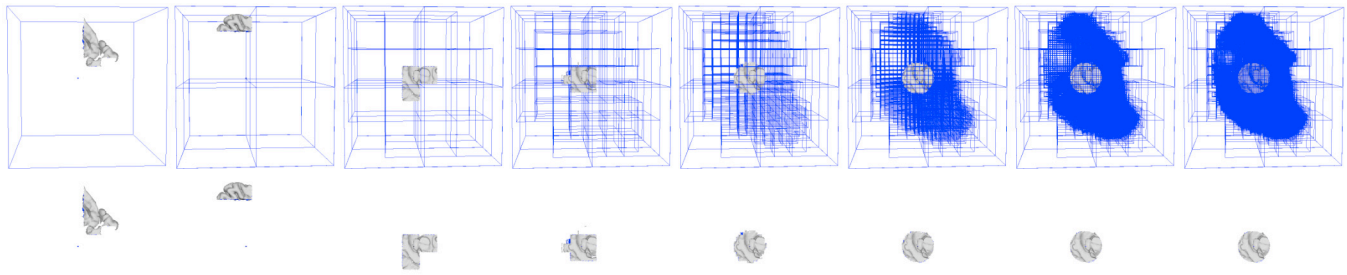

Figure 2. The first column shows that the final octree root's first 16128 triangles are locally coherent, ideal for mesh streaming (Isenberg [6]). Bottom: 16128 rendered triangle budget of RenderArray at depths (from left: 0-5, 5, 5); top: visualization of associated octree of depth 0 to 7, left to right.

We present a new octree that does not create temporary memory for the triangles, and does not store triangles at leaf nodes. We use the fact that octnodes are defined uniquely in 3D space to allow in-place sorting of the local, 1-8, lexicographical partial id-tags, and recorded only the start and end of the triangle indices in each octnode. Provided the partitions uniquely bound space, this idea is easily extended to other space partition trees such as the KD-tree.

3 BASE SYSTEM

The central idea in our system is that our octree is not just used for distance queries, but also represents the entire scene in a hierarchical way of increasing size manageable blocks, which become fewer in number at larger scale. In this section, we start by describing our new way of building an octree in-place without temporary memory, and without storing triangles at leaf nodes.

We then describe the RenderArray that, in essence, is like the Active front across a vertex tree. The RenderArray is an array of pointers to octnodes in the tree. At run time, the tree is queried to find the foremost intersection point of the line of sight with the model. For every frame, this intersection point is calculated and the distance to all the RenderArray's octnode corners and centres are computed and stored. The RenderArray is sorted according to the smallest of these distances and the rendering triangle budget is spent on the first encountered octnodes in the RenderArray.

Finally, we describe how the RenderArray is initialized automatically to adapt to the size complexity of the input, in-core model, and how it can be adjusted at run time for display. We present results obtained from the base system at the end of this section.

3.1 Octree construction

Several octree implementations start by creating a duplicate array or list of the chosen geometric primitive of the original model. This array is then either reset, or destroyed as the geometry is passed to new lists in child nodes. Unfortunately, this characteristic can undermine algorithms that would otherwise be robust. Our approach starts by using directly the object's global array of pointers to individual geometric primitives. Our root node of the octree has two *triangle index* numbers, one that records the first position of the first triangle pointer of the global array, and the second the last position. We then proceed to find the maximum extent of the object and to subdivide space regularly at half distances. At each subdivision step there are three phases, a counting phase (O(N)), a sorting phase (O(N*log(N))), and a node creation phase (O(N)) (see Figure 3).

Phase one – counting: The first phase of the subdivision process is to make one pass on the array of triangle pointers in the range defined by the node's two index numbers, and test in which of the 8 subspaces each individual triangle is contained. When we have determined where a triangle is contained, we mark the triangle with a number tag from 1-8 corresponding to the lexicographical order of the subspaces. Note that Cignoni *et al* [2] tag vertices and write them to leaf addresses, whereas we tag triangles and perform in memory sorting of one-dimensional tags. At the same time, we also keep a counter for each of the 8 subspaces and increment them according to how many triangles were contained in each subspace after the pass. We use the centroid of each triangle to determine if a triangle is contained in a subspace. The subspaces are thus unique in space.

Phase two - in place sorting of partial id: The next phase is to sort the face pointers in the range of the node, directly in their original data structure according to the tag given. It is important to note that, since the child nodes are mutually exclusively contained within the parent octnode's 3D volume, sorting within the child nodes does not affect the global order of the triangles with respect to higher nodes. This triangle reordering makes the mesh more compact and coherent [6] (Figure 2, first columns) than that of the original maximum resolution model.

Phase three - sub node creation: In the third and final phase, we check each counter in turn, creating new sub-nodes if the counters were non-zero, and recursively subdividing them until either MAXTRI, a threshold determining the maximum number of triangles the child nodes may contain, or MAXL a maximum depth threshold, is reached. The starting index of the first node, if there were triangles contained in it, is the same as the starting index of the root node, whilst the ending index of the node is the starting index plus the number of triangles encountered in that node. The starting index of the second node is therefore the same as the number of triangles contained in the first node, whilst the last index adds the number of triangles contained in the second node to the starting index of the sub-node. The other sub-nodes are created with the same procedure as outlined in pseudo-code in Figure 3 below.

```
Subdivide(thenode)
if((level<MAXL)&(thenode->end-thenode->start>=MAXTRI))
   o1count=0 o2count=0 o3count=0 o4count=0
```

```
o5count=0 o6count=0 o7count=0 o8count=0
//PHASE ONE - COUNTING
for i=thenode->start i<=thenode->end i++
      onode=0
      face=atFaceArray(i)
      p=face->calculate_cenrtroid()
      if((p.X()>=thenode->X())&(p.Y()>=thenode-
            >Y())&(p.Z()<=thenode->Z()) {o1count++ onode=1}
      elseif((p.X()>=thenode->X())&(p.Y()>=thenode-
            >Y())&(p.Z()>thenode->Z()) {o2count++ onode=2}
      elseif((p.X()<thenode->X())&(p.Y()>=thenode-
            >Y())&(p.Z()<=thenode->Z()) {o3count++ onode=3}
      elseif((p.X()<thenode->X())&(p.Y()>=thenode-
            >Y())&(p.Z()>thenode->Z()) {o4count++ onode=4}
      elseif((p.X()>=thenode->X())&(p.Y()<thenode-
            >Y())&(p.Z()<=thenode->Z()) {o5count++ onode=5}
      elseif((p.X()>=thenode->X())&(p.Y()<thenode-
            >Y())&(p.Z()>thenode->Z()) {o6count++ onode=6}
      elseif((p.X()<thenode->X())&(p.Y()<thenode-
            >Y())&(p.Z()<=thenode->Z()) {o7count++ onode=7}
      else/*((p.X()<thenode->X())&(p.Y()<thenode-
            >Y())&(p.Z()>thenode->Z())*/{o8count++ onode=8}

      if(onode!=0)
      {//face->set_groupid(face->get_groupid()*10+onode)
         face->set_groupid(onode) //use above for global index
      }

      //PHASE TWO - SORTING
      st=thenode->get_startf()
      numFaces=thenode->get_endf()-thenode->get_startf()+1
      sort(st, st+numFaces, groupid_compare())

      //PHASE THREE - SUB NODE CREATION
      tmpstart=thenode->get_startf()
      if(o1count>0) {
      tmp=newNODE(thenode,1,tmpstart,tmpstart+o1count-1)
      thenode->set_o1(temp)
      tmpstartt=tmpstart+o1count}
      if(o2count>0) {
      tmp=newNODE(thenode,2,tmpstart,tmpstart+o2count-1)
      thenode->set_o2(temp)
      tmpstartt=tempstart+o2count}
      - - -
      // subdivide further
      if(thenode->get_o1()!=0)
      {level++ subdivide(thenode->get_o1() level--}
      if(thenode->get_o2()!=0)
      {level++ subdivide(thenode->get_o2() level--}
      -       - -
      end for

end if
end
```

Figure 3. Pseudo code *for the octree construction step.*

3.2 RenderArray

Once the scene has been represented hierarchically in the octree, we can build an analogous structure for the active front of a vertex tree, with the appropriate number of nodes that we can sort and to which we can compute distances within a preset time. The default for this time has been set to 0.1s.

The RenderArray essentially consists of an array of pointers to octnodes with a variable RMAXTRI, similar to the threshold MAXTRI used in the octree construction phase described in the previous subsection. The RenderArray is created by checking the *triangle index range* of the rootnode. If there are more triangles than RMAXTRI, then we don't store this node pointer in our RenderArray, we access the sub-nodes instead and insert a node in the array only when the range is less than or equal to RMAXTRI. Initially, RMAXTRI is set to the same value as MAXTRI so that the resulting RenderArray will have as many elements as there are leaf nodes.

3.3 Initialization

As mentioned above, the system initially defaults RMAXTRI to the threshold MAXTRI used in the octree at start-up time. It then times how long it takes to complete one pass of computing distances of the RenderArray node corners and centres to a 3D point, and to sort the RenderArray according to the smallest of these distances. If the system takes more than a default time of 0.1s, the system destroys the RenderArray and creates a new one by repeating the process described in section 3.2 with RMAXTRI increased by a factor of ten so as to have fewer nodes to which to compute distances and to sort. The effect of this adjustment may be seen from Table 1, where it corresponds to moving one row down the Table.

An interesting observation can also be made from Figure 2. In the first left image in the upper row, the RenderArray depth is 0 and the number of nodes to be sorted and for which we have to compute distances is just 1, so the triangle budget is spent on the first triangles of the RenderArray (in this case the rootnode). These triangles are clearly far away from the foremost intersection point of the line of sight with the model, and rendering them is like rendering the first triangles of any un-organized triangle soup, although our mesh gains coherence through the triangle reordering during the octree construction phase. As we increase the depth of the RenderArray, by dividing RMAXTRI by ten, we have more nodes to sort and for which we have to compute distances, but the triangle budget is spent on nodes that are closer to the line of sight intersection point, as may be seen in the second image from left in the upper row of Figure 2. Increasing the depth of the RenderArray from depth 4 to 5 adds little visual benefit, whilst adding more computation. At depth 4, the triangle budget is already spent mostly near the foremost intersection point of the line of sight with the model.

3.4 Display

Once this RenderArray has been created, it is retained and used for rendering all frames, unless the user chooses to make a smaller or larger RenderArray, by changing RMAXTRI by multiples of ten. For each frame being rendered, we first determine the foremost intersection point of the line of sight with the model using the octree. Whilst traversing the octree, we use the fast ray rejection test defined by Xu *et al*. [22] to quickly reject finding an intersection point with an irrelevant octnode. A ray is likely to intersect more than one octnode, and more than one triangle in the model, so we keep track of the smallest positive distance from the viewer's position defining the start of the ray and each of the planes defined by the bounding box of the octnodes. This allows one to navigate within a volume. Once the relevant octnode is found, we intersect the ray with the geometry at that node, to find the intersection point.

Once we have determined the foremost intersection point of the line sight, we compute the distance of this point to the 8 corners of each nodes pointed to in the RenderArray. In addition, we also

compute the distance to the centres of these nodes, and store the smallest of these 9 distances in each node of the RenderArray. Finally, we sort the RenderArray according to the distances stored and render a triangle budget set by the user, by first rendering every triangle in the first RenderArray node, and so on for sub-nodes. Finally, to help the user have an idea where the un-rendered parts of the model are, the system can render the bounding boxes of the octree nodes, at the depth set by the user.

3.5 Results

All the results from this paper were obtained using a 500MHz PowerBookG4, with 1 GB of main memory.

We note that frequently when editing a model the user does not move around the model very much. The distance calculations and sorting order thus remain almost constant, which is ideal for constant frame-rate editing. This constant property is also useful to enable us to compare different fill-rate capabilities of different graphics cards, as the only variable is then the number of pixels to be rasterized according to the distance from the viewer to the model. In contrast, if the user pulls-back from a model and a visibility culling algorithm were used, this would very likely affect the number of elements to be processed. We also note that in our experiments, in which we achieved high frame rates (48fps) by consistently rendering a budget of 4500 triangles, this number of triangles filled a considerable viewing area as illustrated in Figure 4 (left image).

We also found that the bounding boxes or contours of the nodes can help the user navigate to regions of interest by moving to areas in the model that look more subdivided. These additional boxes do not need to be rendered to great depths in the octree whilst the user is changing viewpoint. We found this feature quite useful in the context of navigating the hierarchical model of a power plant (see Figure 5).

Finally we implemented an idle function that can be switched on or off at the user's discretion. This function increases the triangle rendering budget gradually if the viewpoint is not changing. This was again, very useful, for example, when we were navigating the brain model (see Figure 1).

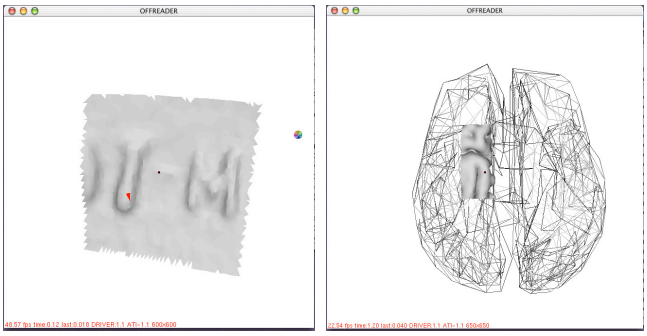

Figure 4. left: *Editing the turbine blade model. 4694 triangles out of 1.7x10^6 triangles are being rendered at render depth 3. The RenderArray is sorting 485 nodes in 0.0015seconds at 48 fps.* right: *Navigation skin rendering of the created 1.1 million triangle brain model. ~5000 triangles are being rendered at render depth 3, with the OpenGL wireframe context switch approximately halving the rendering speed to 22 fps.*

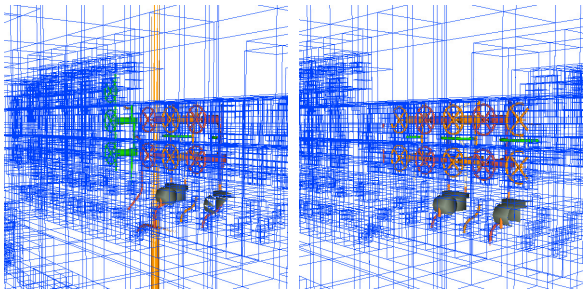

Figure 5. Inspection of a power plant model. 28874 triangles out of 556,374 triangles are being rendered at render depth 3, the RenderArray is sorting 1755 nodes in 0.0058 seconds, and the wire frame bounding boxes have been temporarily rendered at a greater depth to locate detailed geometry that may be of interest. The bounding-box rendering depth is reduced prior to changing the viewpoint.

Model	brain (left)	turbine blade	power plant	Budd ha
#triangles	596872	1.7x10^6	556374	1x10^6
octree depth	7	8	15	10
Octree mem.	1.4Mb	3.8Mb	1.4Mb	2.4Mb
Construct time	11.4	43.2	14.5	24.8
RMAXTRI 100 #nodes memory time	 20814 83k 0.11538	 51734 206k 0.296	 18464 73k 0.09	 32518 130k 0.16
1000	2708 10k 0.0114	5961 23k 0.0265	1755 7k 0.0058	3596 14k 0.001
10000	251 1k 0.00073	485 1.9k 0.0015	240 0.9k 0.0007	347 1.3k 0.001
100000	8 32bytes 4.8e-5	65 260byt. 0.0001	8 32byt. 4.7e-5	15 60byt. 5.7e-5
1000000	1 4bytes 2.8e-5	8 32bytes 4.5e-5	1 4bytes 2.7e-5	8 32byt. 4.4e-5
10000000	-	1 4bytes 2.5e-5	-	1 4bytes 2.5e-5

Table 1. Memory and timings (s) for the octree and different Renderarrays (rows) at depth 0 (bottom) to 5 (top). An RMAXTRI of 100 is equal to the MAXTRI used in the octree construction. At this value, the Renderarray has the same number of elements as leaf nodes and, for the larger models, takes longer to sort than the default target of 0.1s. This is detected at start-up and a new Renderarray at depth 4 with RMAXTRI = 1000 is built. At run-time the user can chose to destroy and create a new Renderarray, for example at depth 3, in order to obtain a faster rendering time as shown in Figure 4.

4 NAVIGATION SKIN

In the previous section, we presented the simple solution of drawing the contours of the octree nodes, at different depths of the octree, as a way of giving the user an idea of the extent of the object, and to help him/her navigate to areas of detailed geometry. In order to give the user a more faithful impression of the extent of the object in space, we have created an extension of the previous octree, by creating a dual octree that keeps track of an approximate, discrete low-level of detail 'skin'. To create this skin, we used a standard high quality level of detail technique [12]. Whilst we note that keeping track of the correspondence between the maximum resolution model and the navigation skin could be potentially useful, for example, as one way of ensuring that the navigation skin does not obscure the high resolution model, in practice we found that a coarse wire-frame does not get in the way of the editing, because of the different rendering style. Thus, we just render the one RenderArray of the high-resolution model and render the coarse mesh over it in wireframe.

4.1 Dual Octree construction

In this case, we use the strategy described in Section 3.1 to create two octrees, one for the high-resolution model, the other only for the coarse skin. Since the skins have only approximately 1000 triangles, we create the skin octree with MAXTRI set to 1.

4.2 Dual RenderArray

We construct two Renderarrays, one for each octree, as described in Section 3.2.. Table 2 shows the additional computer resources needed in this approach.

4.3 Display

The rendering is slightly different when the navigation skin is used. Since both models are in the same space, we infer a correspondence between the low level of detail skin and the high-resolution model from the relationship between the two octrees and thus do not render from the first node of the sorted RenderArray associated with the skin. The user can, if desired, adjust the depth of the RenderArray associated with the skin so as to have control of the granularity of overlapping areas. The default depth is 0. However, in practice we chose to just render the navigation skin as one display list.

4.4 Results

If we use the dual octree, and dual RenderArray, sorting the extra small RenderArray for the skin adds a small cost in computation. In practice, we use just one octree and RenderArray by rendering the navigation skin over it in wire-frame as described above. In this setting, switching the mode to wireframe slows the rendering by ~50% when compared to the speed of the base system which does not display the navigation skin. Thus, we achieved 8 fps by synchronous rendering in software of the 1.7 million triangle turbine blade model with a navigation skin, at full screen (1152x768) without a graphics card on a G4 500 (Figure 6).

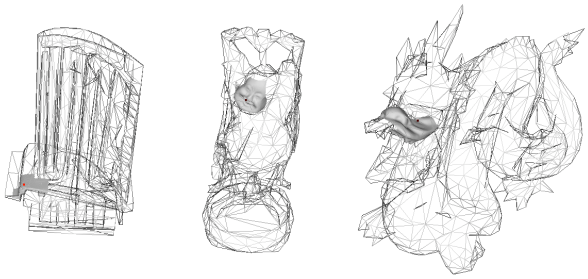

Figure 6. Navigation skin rendering of large models, from left: turbine blade, Buddha statue, Stanford Dragon.

Model	turbine	Buddha
#triangles	1351	1480
RMAXTRI 1		
#nodes	1351	1480
time (s)	0.005576	0.0059
RMAXTRI	406	391
10	0.0016	0.049
RMAXTRI	55	40
100	0.0002	0.0002
RMAXTRI	8	8
1000	0.00013	6.9e-5
RMAXTRI	1	1
10000	8.6e-5	4.4e-5

Table 2. Additional memory and times to those shown in Table 1 associated with the navigation skin for different Renderarrays, from depth 0 (bottom) to 4 (top).

5 APPLICATION OF OUR SYSTEM: EDITING THE BRAIN

The model of the brain whose editing led to development of the system described in this paper was created using a surface reconstruction tool called FreeSurfer (see Fischl et. al [25]) on an MRI scan. The scan was acquired using 2 MP-RAGE scans (8.5 minutes/scan) motion corrected and averaged, collected on a Siemens 1.5T Sonata.

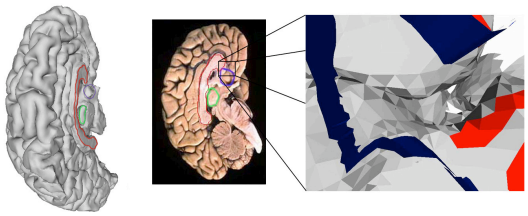

Figure 7. Topological constraints

The model of the brain presented to us consisted of a left hemisphere, with a white matter surface together with a grey matter surface closely following the white matter, but exterior to it (see Figure 7, left). A full, symmetrical model of the brain was required as a triangulated surface as the starting point for several other parameterised and smoother representations to be utilised in an optical tomography system (Zacharoupoulos et. al [24]). The grey and white matter surfaces had together 596872 triangles. In

order to build a symmetrical right hemisphere, whilst keeping the same topological genus, the following steps were taken: *a)* Images such as the one in Figure 7 (middle) were used to plan the perimeter of the area to be removed from the side of the grey matter. *b)* The resulting hole boundary was extruded into the yz plane of the mirror (see red triangles, Figure 7, right), and the model mirrored on this plane.*c)* The vertices on the yz plane were re-used on the right hand side, keeping the mesh connected. In practice, these steps were more easily performed first on the white matter (see extruded triangles in blue passing through the grey matter, Figure 7, right). Then, on the grey matter, we selected red triangles around the dark blue extrusion. In total 25,781 triangles had to be selected and removed, subject to the constraints of non-self intersection, nor intersection between the two surfaces. In addition, features such as deep chasms on either individual surface required careful 3D inspection planning (Figure 7, right). The end result can be seen in Figure 1, & Figure 4 right).

5.1 Editing

The algorithm we used to select triangles for editing represents a ray as two non-parallel planes passing through the origin as defined by Xu *et al*. [22]. Although the triangle picking algorithm is fast, selecting thousands of triangles by hand would be quite time consuming and prone to error. To assist the user, we created the following tools *i)* a selection tool that allows the user to define a radius threshold which is used to tag and show as highlighted all triangles connected to a selected triangle to the depth set in a breadth first traversal of the connected geometry, *ii)* a purge button that enables the user to de-select small triangle groups and to retain only the largest connected group of tagged triangles, *iii)* an undo select button. We found these tools very useful. In particular, *i)* was used with large radius thresholds to select triangles that were otherwise difficult to access, and a radius threshold of 10000 was used to select all the triangles inside the perimeter of the areas to be removed from the side of the model.

6 Conclusions

We have introduced a memory-friendly octree generated in place and without storing triangles at leaf nodes. This representation proves to be very useful for representing a scene in hierarchically manageable parts and has been incorporated in a system that automatically adjusts to the size complexity of the input mesh in order to display at a regular frame-rate the part of the model of interest, for example in an interactive editing task. Like other hierarchical scene representations that use graph-partitioning algorithms [5], to create equal triangle size parts, we can't always guarantee that our octnodes are at the same depth for the line of sight intersection. In practice, the observed frame rate variation was no more than 5-8 fps, at frame rates above 20. Such a variation does not disturb the user's performance of a task such as editing [23]. The way we created the octree changes the triangle ordering with the beneficial side effect of making the mesh more coherent. To further improve mesh coherence, we plan to use the octnode ids to re-label the vertex indices as in [2]. Rendering one coarse level of detail in wire-frame together with a shaded high-resolution, variable size portion of a model, has proved to be an invaluable metaphor for navigating a mesh where multiple levels of detail can't be stored, even though the resulting display-mode context switches reduce the achievable frame-rate by about 50%.

Acknowledgements: We would like to acknowledge Matti Hämäläinen & Bruce Fischl for providing the brain model. Simon Arridge & Athanasios Zacharopoulos for originating this project. Stanford University for the happy Buddha statue & dragon model. As-Built Solutions, for the power plant model. GE Aircraft Engines for the turbine blade model. In addition, we would like to thank Fundação Calouste Gulbenkian, Ministério da Ciência e Tecnologia JNICT/PRAXIS XXI, & ALFAMICRO for financial support. Peter Lindstrom and Martin Isenberg for stream data.

Referernce

[1] Peter Lindstrom. Out-of-Core Simplification of Large Polygonal Models. In *SIGGRAPH 00 Proc*., 259-262, 2000.

[2] Paolo Cignoni and Claudio Rocchini and Claudio Montani and Roberto Scopigno. External Memory Management and Simplification of Huge Meshes. In *IEEE Transactions on Visualization and Computer Graphics*, 9(4), 525-537, 2003.

[3] Peter Lindstrom, and Claudio Silva. A memory insensitive technique for large model simplification. In *IEEE Visualization '01*, 121-126.

[4] Paolo Cignoni and Fabio Ganovelli and Enrico Gobbetti and Fabio Marton and Federico Ponchio and Roberto Scopigno. Adaptive TetraPuzzles: Efficient Out-of-core Construction and Visualization of Gigantic Polygonal Models. In *SIGGRAPH 04 Proc*., 2004.

[5] Sung-Eui Yoon and Brian Saloman and Russel Gayle and Dinesh Manocha. Quick-VDR: Interactive View Dependent Rendering of Massive Models. In *IEEE Visualization '04*, 2004.

[6] Martin Isenberg and Peter Lindstrom. Streaming Meshes, 2004.

[7] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. In *SIGGRAPH 92*, 65-70.

[8] André Guéziec. Surface simplification inside a tolerance volume. Technical report, IBM Research Report RC 20440, 1996.

[9] Hugues Hoppe. Progressive Meshes. In *SIGGRAPH 96,* 99-108.

[10] Jonathan Cohen and Amitabh Varshney and Dinesh Manocha and Greg Turk and Hans Weber and Pankaj Agarwal and Frederick Brooks and William Wright. Simplification Envelopes. In *SIGGRAPH 96 Proc*., 119-128, 1996.

[11] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH 97 Proc*., 209-216, 1997.

[12] Peter Lindstrom and Greg Turk. Fast and Memory Efficient Polygonal Simplification. In *IEEE Visualization 98*, 279-286.

[13] Julie C. Xia and Amitabh Varshney. Dynamic view-dependent simplification for polygonal models.*IEEE Visualization '96*, 327-334

[14] Hugues Hoppe. View-dependent refinement of progressive meshes. In *SIGGRAPH 97 Proc*., 189-198, 1997.

[15] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. In *SIGGRAPH 97*, 99-208.

[16] Szymon Rusinkiewicz and Marc Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes. *SIGGRAPH 00*,343-352.

[17] Guangzheng Fei and Kangying Cai and Baining Guo and Enhua Wu. An Adaptive Sampling Scheme for Out-of-Core Simplification. In *Computer Graphics Forum*, (21):2, 11-119, 2002.

[18] Michael Garland. CS PhD thesis. Carnegie Mellon University, 1998.

[19] Eric Shaffer and Michael Garland. Efficient adaptive simplification of massive meshes. In *IEEE Visualization '01*, 127-134, 2001.

[20] Christopher DeCoro, and Renato Pajarola. XFastMesh: Fast View-dependent Meshing from External Memory. *IEEE Visualization'02*.

[21] Hanan Samet. The Design and Analysis of Spatial Data Structures. Addison-Wesley, ISBN 0-201-50255-0, 1990.

[22] ZY Xu and ZS Tang and L Tang. An efficient rejection test for ray/triangle mesh intersection. *Journal of Software*, (14):10,1787-95.

[23] Benjamin Watson and Neff Walker and Victoria Spaulding and William Ribarsky. Evaluation of the Effects of Frame Time Variation on VR Task Performance. In *IEEE VRAIS 96*, 38-52.

[24] Athanasios Zacharopoulos and Jan Sikora and Simon Arridge. Parametric surface models in medical imaging. In *Proc. Institute of Physics and Engineering in Medecine '04*, 10th Annual Sci, Meeting.

[25] Bruce Fischl and Martin I. Sereno and Anders Dale. Cortical Surface-Based Analysis II: Inflation, Flattening, and a Surface-Based Coordinate System. *NeuroImage*, (9):2, 195-207, 1999.