GPU-based Rendering of Arbitrarily Complex Cutting Surfaces for Black Oil Reservoir Models

Bernardo Franceschin, Frederico Abraham, Luiz Felipe Netto, Waldemar Celes

Tecgraf/PUC-Rio Institute
Computer Science Department
Pontifical Catholic University of Rio de Janeiro, Brazil
{bfrances,fabraham,netto,celes}@tecgraf.puc-rio.br

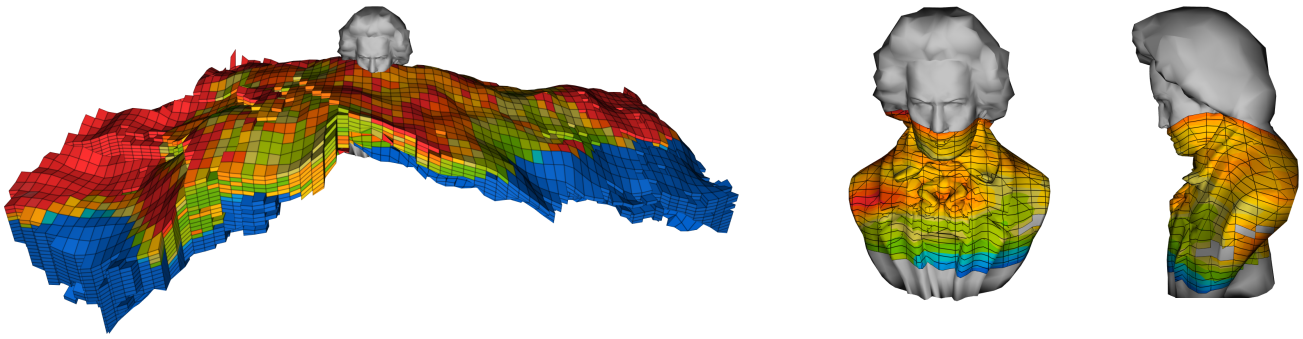


Fig. 1: A complex cutting surface in a reservoir model: cutting surface embedded in the model (left); views of the cutting surface in isolation with mapped scalar field (right).

***Abstract*—Numerical simulation of black oil reservoir models is extensively used by the oil industry to predict and plan field exploration. Such simulations produce a large amount of volume data that need to be inspected. One popular visualization technique to inspect volume data is the rendering of cutting surfaces, shaded by mapping properties associated with model elements. In this work, an efficient GPU-based algorithm for rendering arbitrarily complex cutting surfaces for reservoir models is presented. The rendering strategy is based on an efficient point location algorithm. The proposal includes a compact representation of reservoir models in the GPU memory, the use of a compact regular grid as the acceleration technique, and an accurate point location algorithm for handling hexahedral elements with non-planar faces. Computational experiments have demonstrated the effectiveness and efficiency of the proposed approach, even when applied to large models. A set of applications is discussed in the context of inspecting reservoir simulation results.**

I. Introduction

Sousa et al. [1] discuss the analysis of complex processes present in the oil and gas industry. They advocate for visual computing technologies to help professionals in different areas of expertise to gain insight when dealing with multidisciplinary datasets and processes. Among the different disciplines, the present work concentrates on visualizing results of black oil reservoir simulations or, more specifically, on efficient rendering techniques to support these visualizations. The industry relies on numerical simulations to plan the exploration of oil and gas fields. The output of such simulations includes a large amount of volume data that needs to be analyzed.

Interactive inspection of volume data uses two different approaches. In the first approach, the user specifies a transfer function, which maps opacities to scalar values, and volume rendering techniques are used to reveal the associated isosurfaces or isovolumes. Therefore, the user chooses the scalar value of interest to exhibit the spatial position of the related regions. In the second dual approach, the user specifies cutting surfaces and requests for viewing the scalar field distribution over such surfaces. Hence, the user defines the spatial position of interest, and cut-view techniques reveal the related information.

The focus here is on the second approach, on interactive cut-view rendering techniques. While trivial for regular volume data (it suffices to use texture mapping), the implementation of efficient cut-view techniques for irregular volumes is quite challenging. In particular, this work addresses the problem of interactively investigating the result of black oil reservoir simulations through the definition of general cutting surfaces. The conventional strategy is to employ hierarchical structures to accelerate geometry intersection computations. However, for complex models, a CPU-based approach tends to become the performance bottleneck for interactive visualization applications.

This work proposes a novel GPU-based strategy for efficient cut-view rendering algorithms applied to black oil reservoir models. The proposed solution supports cut-view visualizations through an efficient point location query algorithm on the GPU,

exploiting the characteristics of such models. To this end, the proposal encompasses a compact representation of reservoir models in the graphics memory. As a result, arbitrarily complex cutting surfaces can be directly rendered; for each rasterized fragment, the point location algorithm informs the reservoir cell that contains the point, if any, and the corresponding parametric coordinates are then used to map the associated scalar field.

In summary, this work presents the following main contributions:

- A compact GPU-based representation of reservoir models;
- An efficient point location algorithm for reservoir models;
- An efficient rendering algorithm of arbitrarily complex cutting surfaces based on fragment shaders.

Such contributions serve as the basis for powerful visualization tools for inspecting 3D reservoir models. Figure 1 shows an unrealistic but intricate cutting surface to illustrate the achievement of the proposed strategy.

The rest of this paper is organized as follows. Section II presents related works on point location for unstructured grids and cutting surface rendering for reservoirs. Black oil reservoir models are briefly described in Section III. Section IV presents the proposed model representation in the GPU, followed by the GPU-based point location method in Section V. Surface shading, specially wireframe rendering, is presented in Section VI. Experimental results are discussed in Section VII, demonstrating the efficiency of the proposed solution. Section VIII presents some real applications of the method. Concluding remarks and future work discussion are located in Section IX.

II. Related Work

The conventional approach to render cutting surfaces through irregular volumetric meshes is to compute the intersection among surface polygons and mesh elements on the CPU. Hierarchical subdivision structures [2] are employed to accelerate this computation. An intersecting mesh that includes the associated scalar field values is formed and then rendered. The interactive model inspection often requires manipulating the cutting surfaces, quickly hitting a CPU performance bottleneck.

This work pursues a different approach: a rendering algorithm based on point location. For each rasterized fragment, a point location query is issued to determine the associated scalar field. The challenge relies on the irregularity of the underlying mesh. The following paragraphs discuss GPU-based representation and point location algorithms for unstructured grids represented by general meshes, which are related to the current problem.

Andrysco and Tricoche proposed *matrix trees* [3], an efficient storage scheme for octrees and kd-trees. They encode the tree levels as sparse matrices in a CSR (compressed sparse row) representation and demonstrate GPU-based point location queries on unstructured grids, but model elements overlapping more than one tree leaf limit the method scalability and performance.

Langbein et al. [4] proposed a point location method for large unstructured grids. Their model representation stores the typical vertex coordinates and the indices of vertices composing each element. It also stores the elements incident to each vertex and element adjacency. Point location is supported by a complete point-based kd-tree, storing the index of a single mesh vertex on each leaf. The kd-tree is traversed to locate an element vertex close to the target point. Their proposal identifies an element incident to the vertex and propagates a ray from the vertex to the target point. The propagation is carried by element walking, eventually finding the target element. Special care must be taken if the ray reaches the model boundary, requiring a new kd-tree traversal on nearby elements.

Garth and Joy [5] presented a simple and compact data structure for point location in large unstructured grids. Their structure, the *celltree*, is built upon the concept of bounding interval hierarchies, being able to accommodate large and complex grids while delivering a good performance. Their approach does not suffer from the numerical instabilities of [4], while also working well on model boundaries and with mesh T-junctions. Although originally targeted at large general meshes, the application of such hierarchy for reservoir models deserves investigation. As part of this work, such hierarchies were experimentally compared to the proposed solution but, as it shall be demonstrated, they require too many texture accesses for each fragment, decreasing performance, as in [4].

The NVIDIA Turing architecture introduced ray-tracing hardware processors. Hardware-accelerated geometric queries are now allowed, given the description of a mesh and its bounding volume hierarchy (BVH). Wald et al. [6] employ these processors to perform point location in unstructured tetrahedral meshes entirely inside the GPU. Their basic idea is to trace a ray from the queried point in an arbitrary direction. Assuming that the tetrahedron faces are oriented inwards, if the point lies inside a tetrahedron element, the ray will necessarily first intersect one of its front faces. Otherwise, it will either hit a back-face or not hit the model. The hardware performs both BVH traversal and ray-triangle intersection tests, quickly identifying the target element. Per-element scalar values or interpolated per-vertex scalar values are also easily obtained. This method cannot be directly applied to reservoir models, which contain non-planar faces and concave elements. Nevertheless, it is an interesting approach to the problem and has to be observed with care.

Carvalho et al. [7] implement cutaway visualizations of oil reservoir models by defining a cutting surface as a depth map and rendering it indirectly: instead of drawing the cutting surface, they draw the internal faces of all elements, discarding fragments between the observer and the depth map. For correctly shading the cutting surface, the normal of each fragment is extracted from the depth map. However, the cost of rendering all faces of all elements is prohibitive, even for medium size reservoir models.

III. Black oil reservoir simulation models

A black oil reservoir model is created first by discretizing the reservoir domain in 3D. A $ni \times nj \times nk$ topological grid composed of hexahedral elements is formed. Each hexahedral

element with topological coordinates $[i, j, k]$ has the following elements as its topological neighbors: $[i-1, j, k]$, $[i+1, j, k]$, $[i, j-1, k]$, $[i, j+1, k]$, $[i, j, k-1]$, and $[i, j, k+1]$.

Although simple in terms of topological coordinates, the mesh geometry in cartesian space is usually irregular. Geological faults are modeled as geometric discontinuities; they cause elements that are neighbors in the topological grid to not share vertices and faces. Element faces can be highly non-planar, and some elements may not be well-formed, containing collapsed edges or having zero volume. Furthermore, some elements of the topological grid can be set as inactive, not being part of the reservoir model. The presence of inactive elements can yield irregular and even disconnected active element groups. The groups of elements with the same k coordinate are defined as the reservoir *layers*. Layers resemble terrains containing some discontinuities when seen from above.

The initial reservoir characterization is completed by specifying rock and fluid types and attributing geophysical and geological properties to each mesh element.

The simulator computes oil (and gas) flows and pressures based on its numerical model and the specified well arrangements and production plans. For each simulation time step, the simulator outputs well production data and physical properties associated with each mesh element, such as oil, gas and water saturations, pressures, and temperatures.

Scientific visualization techniques aid the reservoir engineer in understanding the underlying physical phenomena and its correlation with quantitative results such as oil production. Different visualization techniques are offered for better inspection of the 3D volume data.

IV. Compact model representation

The first requirement is a compact and efficient data structure to represent the *active* model elements in the GPU memory. The model elements lay on a topological grid of dimension $ni \times nj \times nk$, where ni, nj, and nk represent the total number of cells on directions I, J, and K, respectively. Therefore, each element is uniquely identified by its topological coordinates $[i, j, k]$.

One can think that the model vertices also lay on a topological grid, now with dimensions $(ni+1) \times (nj+1) \times (nk+1)$. However, due to geometric discontinuities, there may be more than one vertex instance at given topological coordinates $[i, j, k]$, and a consistent way to store all vertex instances is needed.

A *base vertex* of an element is defined as the incident vertex that comes first in the three directions, $I+$, $J+$, and $K+$. In the present proposal, the base vertices of all elements are stored at their respective $[i, j, k]$ coordinates, including base vertices of inactive elements that are incident to active neighbors. The other vertex instances, sharing the same topological coordinates, are listed explicitly and appended to the vertex storage.

A. Element classification

An active model element is classified as *type 1* if its seven vertices other than the base vertex are stored at their topological coordinates. These seven vertices are the base vertices of their respective elements. Otherwise, the element is classified as *type 2*. Figure 2 illustrates the proposed element classification.

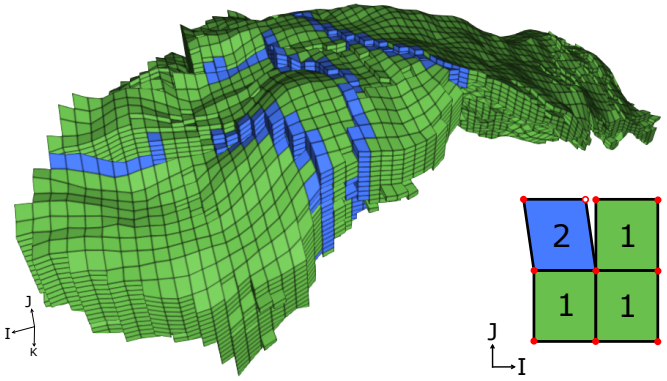


Fig. 2: The classification of elements in *type 1*, shown in green, and *type 2*, shown in blue, for an actual reservoir model. The schematic illustration depicts the classification criterion: elements of *type 2* do not have all their incident vertices as base vertices (represented as red solid circles).

An element of *type 1* has its incidence implicitly given, based on its topological coordinates. The incidence of an element of *type 2* has to be explicitly stored. This implicit representation represents a significant saving of memory since, in practice, more than 90% of the elements are of *type 1*.

B. Data storage and sparsity treatment

Only active elements are stored in the GPU (together with their incident vertices). This sparsity in the ijk domain is exploited to achieve a compact structure.

The *perfect spatial hashing* technique, proposed by Lefebvre and Hoppe [8], is employed. Their work defines a simple, multidimensional perfect hash function. In the 3D case, it packs data that is sparse in a discrete ijk domain into a compact 3D table H with dimensions $\bar{m}^3$, being $\bar{m}$ sufficient to store all input data [8]. It also defines an offset table ϕ with dimensions $\bar{r}^3$, typically smaller than those of the original table. Each offset table entry contains integer offsets in the ijk space. The offset value $[\phi_i, \phi_j, \phi_k]$ for given $[i, j, k]$ coordinates in the original domain is obtained from the following position in ϕ: $[i \bmod \bar{r}, j \bmod \bar{r}, k \bmod \bar{r}]$. The data value associated to $[i, j, k]$ coordinates in the original domain is stored in the following position in H: $[(i+\phi_i) \bmod \bar{m}, (j+\phi_j) \bmod \bar{m}, (k+\phi_k) \bmod \bar{m}]$. The method both computes the offset table and assigns data locations in H while avoiding collisions, which delivers a perfect hash function. The algorithm for offsets and data locations also attains good spatial coherence, with a good chance of two samples that are near in the original domain to be near in H. The tables can be stored as simple 3D textures in the GPU. The indexing of the offset table and the subsequent access to the associated data on the GPU are straightforward and efficient.

Figure 3 illustrates the proposed model data structure. For the sake of simplicity, the present proposal implements a unique

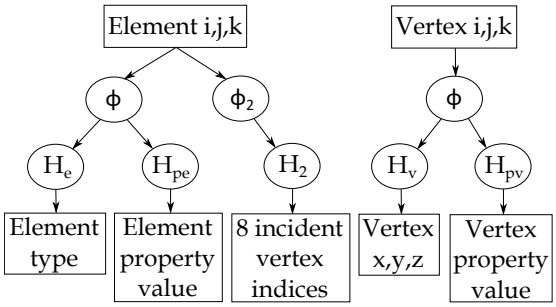


Fig. 3: Element and vertex data access on the GPU. The model structure components are: ϕ: hash offsets for both elements and vertices; ϕ_2: hash offsets for elements of *type 2*; H_e: storage of element types; H_{pe}: storage of element properties; H_2: storage of incident vertex indices for element of *type 2*, pointing directly to the H_v and H_{pv} storages; H_v: storage of vertex coordinates; H_{pv}: storage of vertex properties. Once we obtain the element type, it is possible to access vertex coordinates, per-element properties, and per-vertex properties.

mapping table ϕ for storing element and vertex data. The valid input indices cover the range $i \in [0, ni]$, $j \in [0, nj]$, and $k \in [0, nk]$, needed for vertex mapping. The offset table maps to actual data storage. There are four of them: one H_v for vertex $[x, y, z]$ coordinates, one H_e bitmap for element types (*type* 1 or *type* 2), one H_{pv} for vertex properties, and one H_{pe} for element properties. Maps H_v and H_{pv} are appended with data related to the additional vertex instances.

Elements of *type 2* need an extra hash: ϕ_2 and H_2. The table ϕ_2 maps the original $[i, j, k]$ element coordinates to H_2, which explicitly stores the indices of all 8 incident vertices of the element. These indices may refer to *base* or *non-base* vertices, and they already represent access to the storage table H_v (or H_{pv}).

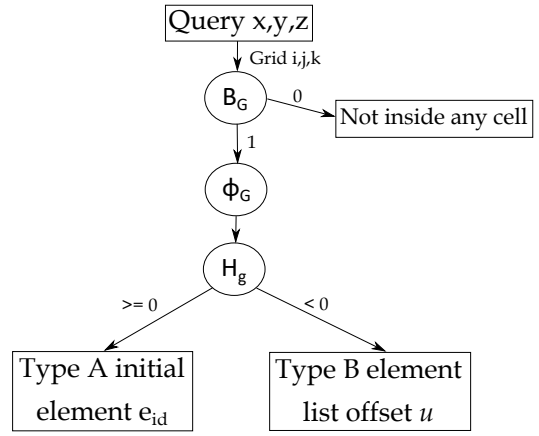
C. Element adjacency queries

Although not used by the point location algorithm discussed in this work, querying element adjacencies is useful in different rendering algorithms. The structure can be easily extended to include this information. Adjacency requires six additional bits per element, which can be packed alongside the bit representing the element type. Each bit indicates if the corresponding element face is shared by an active sibling element in the topological grid. The siblings, if present, have their $[i, j, k]$ coordinates obtained implicitly.

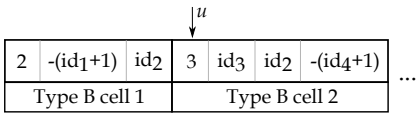
V. Point location algorithm

The point location algorithm uses a regular cell grid as the acceleration technique. Given a point in space (x, y, z), the containing grid cell is quickly found.[1] Given the axis-aligned bounding box (AABB) enclosing all active elements, the model discretizations ni, nj, and nk, and a grid refinement factor r, the AABB is discretized in $\lceil ni \times r \rceil \times \lceil nj \times r \rceil \times \lceil nk \times r \rceil$ cells. Figure 4 illustrates the point location structure and algorithm.

[1]Note that the name *element* refers to model entities and *cell* to grid entities.



(a) Regular grid structure components: B_g informs if the cell is non-empty; ϕ_g and H_g store one integer associated to each cell in a perfect hash. If in a cell of *type A*, the integer holds the identifier e_{id} of the topological search initial element. Otherwise, a regular search on the list pointed by u is started.



(b) Packed element list. In this illustration, the second cell element list contains three elements: id_3 and id_2 of *type 1* and id_4 of *type 2*.

Fig. 4: Point location structure and algorithm: (a) path from the query coordinates x, y, z to the list of elements to search; (b) example of packed element lists for cells of *type B*.

A. Grid cell classification

Grid cells may be *empty* or *non-empty*. Empty cells do not intersect any model element; non-empty cells intersect at least one element. The proposal further classifies non-empty cells in two types:

- Cell of *type A*: the grid cell volume is entirely covered by intersecting elements of *type 1*.
- Cell of *type B*: the grid cell intersects at least one element of *type 2*, or its volume is not entirely covered by intersecting elements.

Cells of *type A* only store one intersecting element identifier: $e_{id} = i + ni \times (j + nj \times k)$. The stored identifier corresponds to the element with highest volume overlap with the grid cell. Cells of *type B* require explicit access to all intersecting elements. In this case, the cell stores a negative value that encodes an index to an auxiliary array of integers: $-(u + 1)$, with u being the array index. This auxiliary array stores lists of intersecting elements. Each list begins with an element count followed by each element identifier. If the element is of *type 1*, its identifier e_{id} is stored; if the element is of *type 2*, its identifier is encoded in a negative number: $-(e_{id} + 1)$. The element identifiers in each list are sorted by decreasing order of volume overlap between element and cell.

In practice, there may be a large number of empty grid cells; these cells do not need to be represented. To save memory, a

bit array B_g and another perfect hash are used. B_g informs if the global cell with $[i, j, k]$ coordinates is *non-empty*. The offset table ϕ_g maps a *non-empty* $[i, j, k]$ cell to the grid cell storage table H_g.

B. Point location within a model element

A model element is represented by an irregular hexahedron with non-planar faces. Given the incident vertex coordinates and the element shape function N_i [9], the coordinate (x, y, z) of an internal point with parametric location $(s, t, r) \in [-1, 1]^3$ is expressed as:

$$x = f(s,t,r) = \sum_{i=1}^{8} N_i x_i$$

$$y = g(s,t,r) = \sum_{i=1}^{8} N_i y_i$$

$$z = h(s,t,r) = \sum_{i=1}^{8} N_i z_i$$

where (x_i, y_i, z_i) represents the incident vertex coordinates.

The inverse, to find the parametric location given the coordinate, is not trivial and ends up in a root finding problem:

$$x - f(s,t,r) = 0$$
$$y - g(s,t,r) = 0$$
$$z - h(s,t,r) = 0$$

A Newton-Raphson procedure is performed to find the (s, t, r) location. Experimental results have shown that four iterations are sufficient to deliver accurate results. If $(s, t, r) \in [-1, 1]^3$, the point is inside the element. Moreover, this procedure also provides significant parametric values for points outside the element, as described in the next subsection.

C. Point location within a grid cell

A point location query in a cell of *type A* is performed by a topological search. The search begins at the element stored in the cell; if the given point is not inside the element, the returned parametric values are used to choose the closest neighbor to continue the search. As soon as an element containing the point is found, the query is complete, and the element identifier, together with the corresponding parametric location, is returned. Note that this search tends to be efficient: it starts at the element with the highest volume overlap and continues to the neighbors in a coherent way.

In cells of *type B*, the search is performed conventionally. The listed elements are tested one by one, in decreasing order of volume overlap.

It should be noted that our cell classification (types A and B) is crucial for the structure compactness and efficiency. Cells of *type A* only require the storage of one element identifier versus the explicit element list for cells of *type B*. The topological search in cells of *type A* also tends to find the desired element faster than the simple search performed on cells of *type B*.

The grid refinement factor r has a significant influence on search performance. As r increases, each cell overlaps fewer elements, decreasing the number of visited elements. On the other hand, increasing r considerably enlarges the memory space needed to store the grid.

VI. Surface shading

A fragment shader is responsible for shading the cutting surfaces. For each fragment, given the element that contains the point and the corresponding parametric coordinate (s, t, r), the associated scalar value is first computed, and the corresponding color is assigned following the color scale of the application. For visualizing cell properties, the scalar value associated with the element is retrieved and assigned to the fragment. For vertex properties, the scalar values associated with the element vertices are retrieved, and the parametric coordinates are used to interpolate the scalar value at the fragment: $\alpha = \sum N_i \alpha_i$.

The challenge resides on rendering the wireframe. Revealing the reservoir mesh wireframe on the cutting surfaces may be crucial for a better understanding of the numerical simulation. The fragment gains the wireframe color if it is close to the element boundaries. However, surface triangles parallel to element faces should not be rendered as part of the wireframe to avoid coloring the entire triangle as wireframe.

Bærentzen et al. [10] suggested a procedure to draw antialiased wireframes. Their strategy combines the original fragment color with the wireframe color according to the distance from the fragment to the closest triangle edge. Their work is extended here to consider the distance to the element boundary in 3D while avoiding parallel faces. Considering the parametric coordinates s, t, and r, the fragment is snapped to the closest boundary, setting the corresponding parametric coordinate to -1 or 1. With the fragment on the boundary, the tangent plane at this point is computed (remember the faces are non-planar) and intersected with the triangle plane. The distance to the boundary is the distance from the fragment to the resulting line of intersection between the two planes.

VII. Experimental Results

This section presents an evaluation of the proposed solution, beginning with model representation statistics. The proposed regular grid structure is directly compared with the previously discussed *celltree* structure [5]. Then, an analysis of the influence of the regular grid refinement factor r is conducted, both in terms of memory footprint and performance.

The computational tests considered four reservoir models, with discretizations ranging from 300,000 to 6,000,000 elements. The models are identified here by the letters *H*, *L*, *T*, and *P*, all based on actual reservoir models. Models *H* and *L* contain many discontinuities and inactive element areas. Model *T* has large contiguous regions with some discontinuities. Model *P* does not possess any discontinuity on its domain, being entirely composed of elements of *type 1*.

All tests were performed on a computer equipped with a 3.3 GHz Intel i7 processor, 40 GB of RAM and an NVIDIA Geforce GTX 1080 Ti GPU with 11 GB of graphics memory. The implementation uses C++, OpenGL, and GLSL in its entirety. All rendering performance measurements were made

mapping per-element properties. The cost of the proposed rendering algorithm is dominated by fragment operations; the geometry complexity of the cutting surfaces has a low impact on the achieved performance.

A. Model representation

Table I shows the active element count and the percentage of elements of *type 1* for each model. The table also presents the GPU memory usage of each component of the model internal representation. As can be noted, the optimizations for elements of *type 1* and the use of perfect hashes result in compact structures, even for models containing millions of elements.

	Model			
Statistics	**H**	**L**	**T**	**P**
Active element count (M)	0.3	1.1	6.0	6.2
Elements of *type 1* (%)	90%	90%	93%	100%
Element and vertex offsets (ϕ) (MB)	0.4	1	10	13
Element types (H_e) (MB)	0.06	0.2	0.8	0.8
Vertex coordinates (H_v) (MB)	6.2	22	83	81
Type 2 incidences (ϕ_2 and H_2) (MB)	1.2	4	15	-
Total geometry memory (MB)	**8**	**27**	**109**	**95**
Element property (H_{pe}) (MB)	2.0	7	26	27
Vertex property (H_{pv}) (MB)	2.1	7	28	27

TABLE I: Element count statistics and GPU memory usage for each tested model.

B. Comparison with the celltree structure

The *celltree* structure, proposed by Garth and Joy [5], served as a reference to evaluate the regular grid and point location proposal. The *celltree* structure was implemented and tested, replacing the regular grid as the acceleration technique, maintaining the proposed model representation. The bounding interval tree was recursively built, forming leaves with at most two elements. In this way, no element list is needed: the two integers stored at each leaf can directly store one or two element identifiers. Moreover, the same encoding optimization when storing identifiers was applied: if the element is of *type 1*, its identifier e_{id} is stored; if the element is of *type 2*, its identifier is encoded in a negative number: $-(e_{id}+1)$.

The *celltree* structure is very compact, avoiding element overlaps altogether, with each element being stored only once in the entire hierarchy. Nevertheless, its point location search ends up testing a larger set of cells, due to many situations where it is necessary to traverse the two children of a given hierarchy node to locate the desired element. This cost is critical in the intended applications, which tend to perform a massive number of point location queries per frame. The regular grid has shown to be more appropriate in this case, requiring the test of a smaller set of elements, resulting in a better performance.

The test measured performance by rendering, for each model, one vertical cutting plane in one setting and twenty equidistant horizontal planes in another. The regular grid refinement factor r was adjusted to produce regular grids with the same GPU memory usage[2] as the *celltree* structure. Table II shows

[2]Represented by the GPU memory needed to store B_g, ϕ_g, H_g, and the element lists for cells of *type B*.

Model	**Method**	**GPU Memory Usage (MB)**	**1 plane fps**	**20 planes fps**
H	Celltree	5.0	65	3
	Regular grid	5.0 ($r = 0.5$)	**79**	**6**
L	Celltree	16.4	30	3
	Regular grid	16.4 ($r = 0.43$)	**35**	**5**
T	Celltree	91.5	27	3
	Regular grid	91.3 ($r = 1.033$)	**77**	**16**
P	Celltree	93.1	121	11
	Regular grid	92.7 ($r = 1.842$)	**286**	**33**

TABLE II: Comparison between the *celltree* structure and the proposed regular grid structure. The grid factor r was adjusted to match the GPU memory usage of the *celltree* structure. The rendering performance measurements were taken with a single cutting plane and with twenty parallel cutting planes.

the comparison measurements, with the proposed method performing better in all tests.

The observed gain in performance for the two first models (22% and 17%) are small if compared to the gain for the last two (85% and 136%). The main reason is the presence of discontinuities and inactive elements, generating more grid cells of *type B*, which decreases search performance.

C. Regular grid refinement factor influence

The next test measured the influence of the grid refinement factor r on memory usage and rendering performance. The performance was measured by rendering twenty equidistant horizontal planes cutting through the *T* model (with 6 million elements). Figure 5 presents the achieved results.

As mentioned before, the greater r is, the greater is the expected point location search performance, since smaller grid cells tend to intersect fewer elements; on the other hand, memory usage increases. The plot shows the memory costs growing proportional to r^3, as expected, with the rendering performance increasing. From this and other similar computational experiments, setting $r = 1.6$ seemed appropriate, which delivers a good trade-off between memory usage and performance for most models.

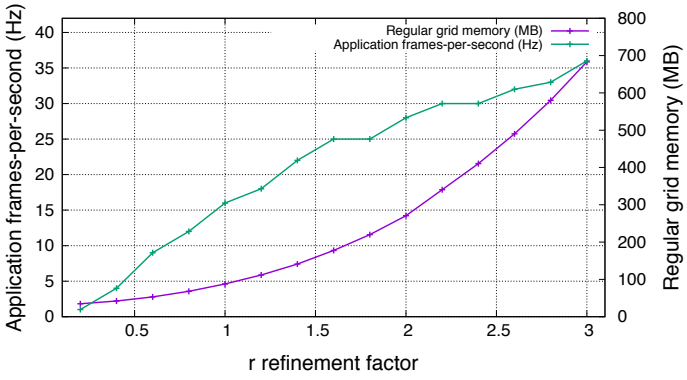


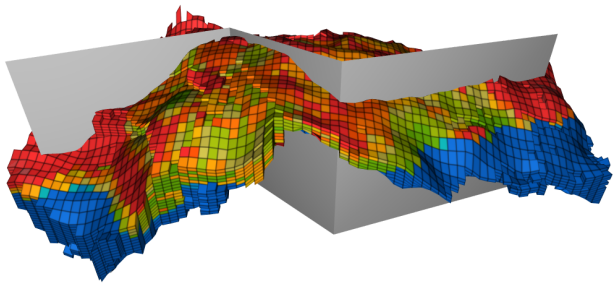
Fig. 5: Influence of the grid refinement factor r on memory usage and rendering performance. The test renders 20 plane surfaces cutting model *T*.

VIII. Applications

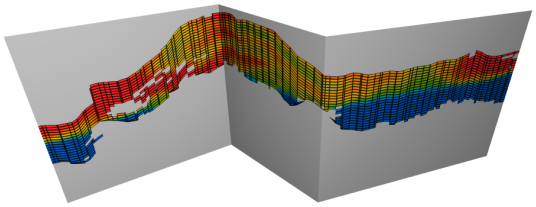
This section illustrates some actual applications of the proposed method for inspecting reservoir simulation results.

A. Fence diagram and fault surfaces

Visualization of fence diagrams is a traditional tool for inspecting reservoir models. A fence diagram is created by specifying a set of vertical planar panels. In general, the user needs to evaluate the model property distribution along sections passing through the wells. Figure 6 illustrates one achieved visualization. The proposed rendering technique allows modifying the fence position interactively, even for large reservoir models.



(a) Fence diagram embedded in the model



(b) Fence diagram viewed in isolation, with property mapping

Fig. 6: Visualization of fence diagrams.

Another common practice in the oil industry is the use of faults as cutting surfaces. While inspecting reservoir simulation results, specialists need to investigate and understand the fluid behavior in the vicinity of geological faults. To this end, it is desired to use the original faults as cutting surfaces, mapping reservoir properties over them, which is directly obtained with the proposed algorithm.

B. Reservoir properties mapped over streamlines

Reservoir simulators also output oil, gas, and water velocity fields. Streamlines are then traced to investigate the fluid paths. Visualizing the streamlines allows a set of different analyses. One important analysis examines the quality of reservoir drains. For this end, it is needed to visualize model properties along streamlines. This visualization is easily achieved with the proposed rendering algorithm; it suffices to draw the streamlines as "cutting surfaces". As the proposed method is based on point location, the same algorithm can be employed for line fragments, as illustrated in Figure 7.

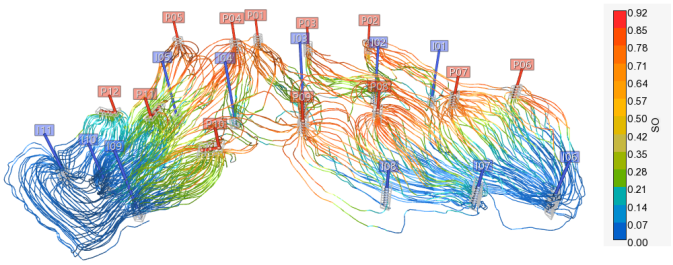


Fig. 7: Visualization of streamlines with a reservoir property mapped over them. The image illustrates the mapping of oil saturation (SO). Injector wells are shown in blue, while producers are shown in red.

C. Visualization of convex probes

One possible application of our method is the extraction of convex probes to cut out parts of the reservoir model. The main idea is to simulate the drawing of the probe shape inside the model and allow the user to move the cut object apart from it. An example of this visualization can be seen in Fig 8.

The proposed visualization can be obtained by rendering the reservoir model twice, inspired by Volume Clipping from [11]. In the first rendering pass, the cut inside the reservoir is revealed; in the second, the cut object is rendered at another position interactively chosen by the user.

The reservoir model cut is performed with the following steps:

1) Draw the back faces of the convex probe object inside the reservoir model, writing the fragment depth to a texture buffer, keeping fragments far away from the camera.
2) Draw the front faces of the convex object inside the reservoir model, writing the fragment depth to a texture buffer, keeping fragments closer to the camera.
3) Draw the reservoir model hull, discarding fragments that have a depth greater than the front faces of the convex object and less than its back faces.
4) Draw the cut surface intersection with the proposed point location algorithm by sampling the depth from the back-face texture map.

The probe is drawn in isolation by simply setting it as a cutting surface, using the proposed method. However, when the probe object intersects the reservoir external hull surface, we need to combine the model hull with the cutting surface. The final image is achieved then by drawing the model hull discarding fragments outside the probe, using the front and back face texture maps.

D. Decoupled cut visualization

To analyze the behavior of the simulation in the vicinity of wells is one of the most important tasks while inspecting a reservoir model. The proposed technique offers different views for this inspection. Each well is represented by a cylindrical shape following its trajectory. Along the well, there are several *completions*. Well completions connect the reservoir to the surface, allowing fluids to be produced or injected. They

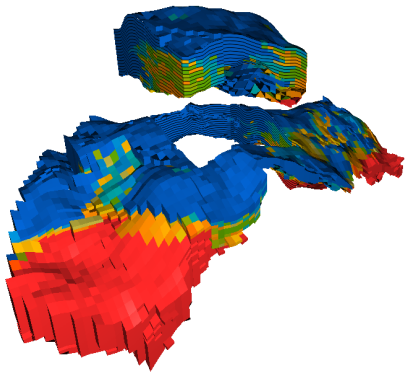


Fig. 8: Convex cut visualizations can rely on our GPU-based rendering method. They are quite useful to inspect the interior of the reservoir, particularly around its wells.

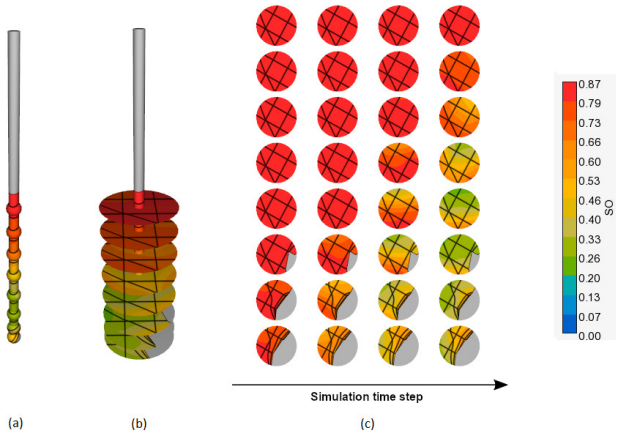


Fig. 9: Different visualization techniques to inspect reservoir behavior in the vicinity of a well: (a) well trajectories and completions rendered as cutting surfaces; (b) disks positioned at the well completions, showing the property variation on the regions around them; (c) disks laid out in a 2D view, presenting the property variations around each completion over simulation time.

usually consist of perforations along the well path. The reservoir regions close to the well completions are of particular interest to reservoir engineers. Figure 9 illustrates some proposed well visualizations.

First, the well and completion shapes can be used as cutting surfaces, as illustrated in Figure 9a. Furthermore, one can render disks centered at the well completions, perpendicular to the well trajectory, as in Figure 9b.

Additionally, it is possible to decouple the rendered object from the cutting coordinates. Figure 9c illustrates an array of disks along the well trajectory in a 2D view. Each column represents a different time step of the simulation. To obtain such images, the disks are rendered in the 2D space, with their 3D coordinates passed as vertex attributes, which are used by the point location algorithm. This example shows that the proposed technique can be seen as a *unstructured 3D texture* representing the reservoir model.

IX. Conclusion

This work presented an approach for efficiently rendering cutting surfaces with arbitrary geometry in reservoir models. The proposal includes a compact data structure for storing the model in the GPU memory, an acceleration technique based on a regular grid, and an efficient algorithm for point location. For accuracy, the point location algorithm uses a Newton-Raphson iteration to handle hexahedral elements with non-planar faces. The compactness of the proposed data structure allowed the storage of large reservoir models on the GPU. Computational tests have demonstrated that the proposed regular grid brought better performance than the one achieved with the *celltree* data structure as the acceleration technique. The effectiveness of the proposed rendering algorithm was demonstrated by its use in different reservoir visualization techniques, allowing interactive inspection even for large reservoir models.

A natural extension of the proposed approach is to handle unstructured meshes. Another topic of interest is the integration of the proposed compact representation with level-of-detail techniques for massive reservoir models.

Acknowledgment

The first author was financially supported by CAPES, the Brazilian government agency for higher education personnel improvement, during his master program.

Tecgraf/PUC-Rio is a research institute mainly funded by Petrobras.

References

[1] M. C. Sousa, E. Vital Brazil, and E. Sharlin, "Scalable and Interactive Visual Computing in Geosciences and Reservoir Engineering," *Geological Society, London, Special Publications*, vol. 406, no. 1, pp. 447–466, 2015.

[2] T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki, and S. Hillaire, *Real-Time Rendering 4th Edition*. Boca Raton, FL, USA: A K Peters/CRC Press, 2018.

[3] N. Andrysco and X. Tricoche, "Matrix trees," *Computer Graphics Forum*, vol. 29, no. 3, pp. 963–972, 2010.

[4] M. Langbein, G. Scheuermann, and X. Tricoche, "An efficient point location method for visualization in large unstructured grids," in *Proceedings of the Vision, Modeling, Visualization*, 2003, pp. 27–35.

[5] C. Garth and K. I. Joy, "Fast, memory-efficient cell location in unstructured grids for visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 6, pp. 1541–1550, Nov 2010.

[6] I. Wald, W. Usher, N. Morrical, L. Lediaev, and V. Pascucci, "RTX Beyond Ray Tracing: Exploring the Use of Hardware Ray Tracing Cores for Tet-Mesh Point Location," in *Proceedings of the Conference on High-Performance Graphics (HPG)*, 2019.

[7] F. M. de Carvalho, E. Vital Brazil, R. G. Marroquim, M. C. Sousa, and A. Oliveira, "Interactive cutaways of oil reservoirs," *Graphical Models*, vol. 84, pp. 1 – 14, 2016.

[8] S. Lefebvre and H. Hoppe, "Perfect Spatial Hashing," *ACM Transactions on Graphics*, vol. 25, no. 3, pp. 579–588, Jul. 2006.

[9] P. I. Kattan, *MATLAB Guide to Finite Elements*. Springer Berlin / Heidelberg, 2008.

[10] A. Bærentzen, S. L. Nielsen, M. Gjøl, B. D. Larsen, and N. J. Christensen, "Single-pass wireframe rendering," in *ACM SIGGRAPH 2006 Sketches*, ser. SIGGRAPH '06. New York, NY, USA: ACM, 2006.

[11] K. Engel, M. Hadwiger, J. M. Kniss, A. E. Lefohn, C. R. Salama, and D. Weiskopf, "Real-time volume graphics," in *ACM SIGGRAPH 2004 Course Notes*, ser. SIGGRAPH '04. New York, NY, USA: ACM, 2004.