5
Results

For the evaluation of our proposal, we implemented a ray-casting using CUDA that handles both tetrahedral and hexahedral meshes. We measured the rendering performance with four known volumetric datasets: Blunt-fin, Fuel, Neghip, Oxygen. We tested the following algorithms:

1. **Hexa**$_{CONST}$: Ray-casting for hexahedral meshes that uses a fixed number of 100 steps in each cell to accurately compute the illumination assuming a constant scalar field at each step.

2. **Hexa**$_{ACC}$: Our proposal for accurate volume rendering of hexahedral meshes.

3. **Hexa**$_{FAST}$: Our proposal for fast volume rendering of hexahedral meshes.

4. **Tetra**$_{HARC}$: Ray-casting using a pre-integrated table (3), with each hexahedral cell subdivided in six tetrahedra.

Although the $Hexa_{Const}$ algorithm is far from being an efficient solution, it produces accurate results, and we use it as our rendering quality reference. We ran the experiments on Windows 7 with an Intel Core 2 Duo 2.8 GHz, 4 GB RAM and a GeForce 460 GTX 1 GB RAM. The screen size was set to 800 x 800 pixels in all experiments.

In Section 5.1.1 we compare a tetrahedral and our hexahedral proposal for isosurface rendering regarding quality. Then, in Section 5.1.2, we compare the volume rendering quality. Section 5.2 presents a time comparison among the algorithms. We compare the rendering quality of the $Hexa_{FAST}$ algorithm in Section **??**.

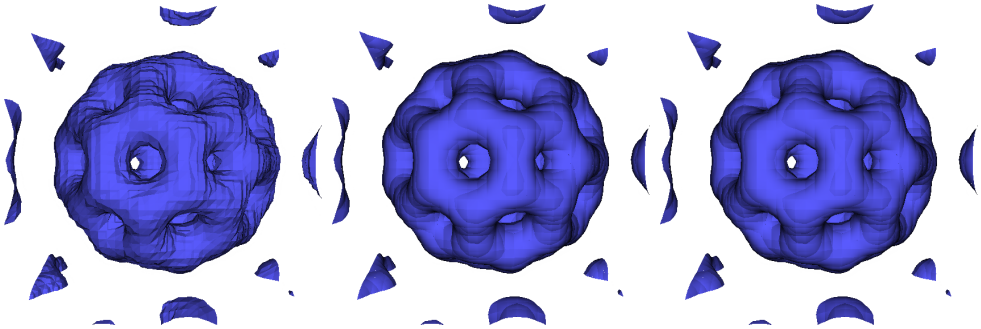
5.1
Rendering quality

In this section, we will discuss how our proposals compares with the hexahedron subdivision approach, both in isosurface rendering and volume rendering.

5.1.1 Isosurface rendering

Figure 5.1 compares the isosurface of the Bucky dataset.

As can be noted, our proposal presents smoother isosurfaces, as expected, since we consider a trilinear scalar function. Both algorithms ($Hexa_{ACC}$ and $Hexa_{FAST}$) presents similar results.



5.1(a): $Tetra_{HARC}$ 5.1(b): $Hexa_{ACC}$ 5.1(c): $Hexa_{FAST}$

Figure 5.1: Bucky dataset isosurfaces.

5.1.2 Volume Rendering Quality

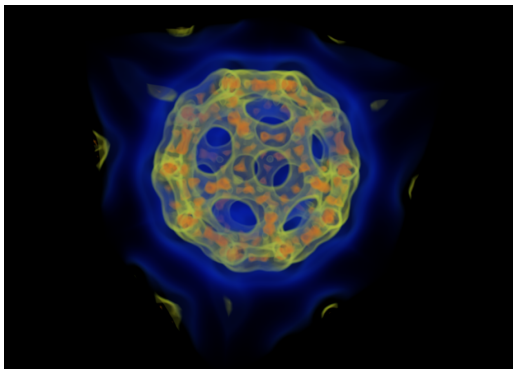
We evaluate our ray-casting algorithms regarding rendering quality. We first devised an experiment using synthetic volume data composed by only one hexahedral cell. Figure 5.2 shows the scalar values set to each vertex and presents the four images achieved by the four algorithms. For these images, we used a transfer function with six thin spikes, isolating six different slabs. Even though such a scalar field variation is rare in actual datasets, this synthetic test aims to show how the subdivision of an hexahedron can lead to unrecognizable results. The test also demonstrates that both of our proposals are capable of rendering scalar fields with complex variations.

Figure 5.3 shows and compares the results achieved by the four algorithms for rendering the Bucky model. Figures 5.3(e), 5.3(f) and 5.3(g) shows the difference between the images achieved with our two proposals and the subdivision scheme when compared to our quality reference. It's possible to notice a clear difference between the rendering images, with our proposals more similar to our quality reference. The $\mathbf{Hexa}_{FAST}$ proposal presents a subtle difference, expected because of our approximations of the ray integral.

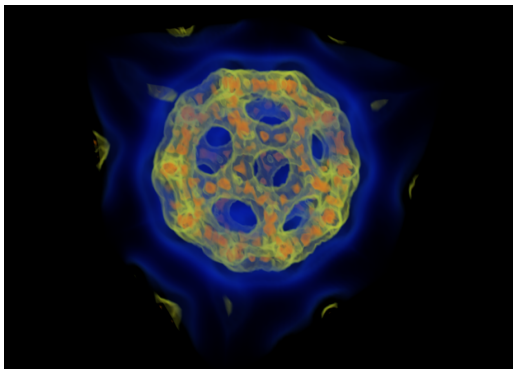
Figure 5.4 shows the result of a similar experiment but now considering the Bluntfin model. As can be noted, the results are equivalent: our algorithm produces images with a better quality. In this example, our two proposal present similar results.

5.2(a): Scalar values at vertices | 5.2(b): $\mathbf{Hexa}_{CONST}$ | 5.2(c): $\mathbf{Tetra}_{HARC}$ | 5.2(d): $\mathbf{Hexa}_{FAST}$ | 5.2(e): $\mathbf{Hexa}_{ACC}$

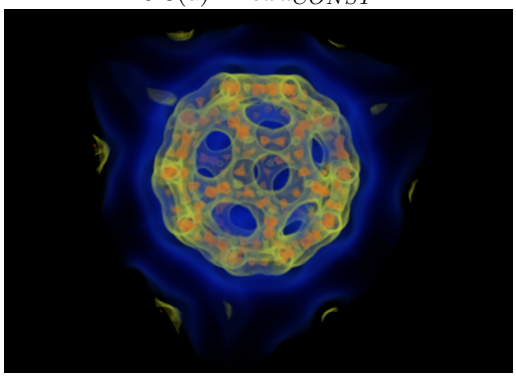
Figure 5.2: Rendering on a synthetic model composed by on hexahedron.



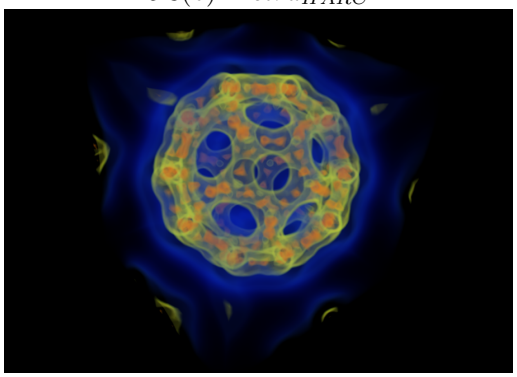
5.3(a): $Hexa_{CONST}$



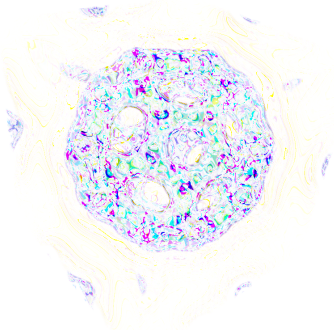
5.3(b): $Tetra_{HARC}$



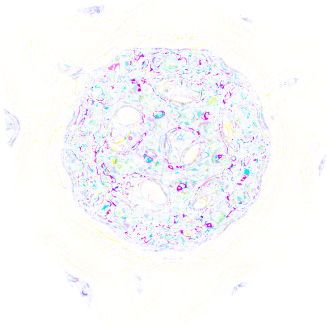
5.3(c): $Hexa_{FAST}$



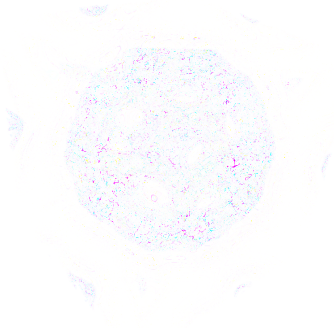
5.3(d): $Hexa_{ACC}$



5.3(e): Difference $Tetra_{HARC}$

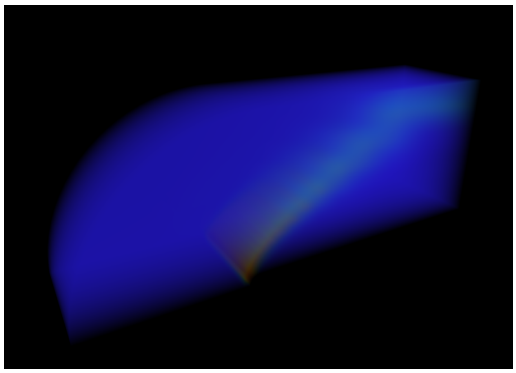


5.3(f): Difference $Hexa_{FAST}$

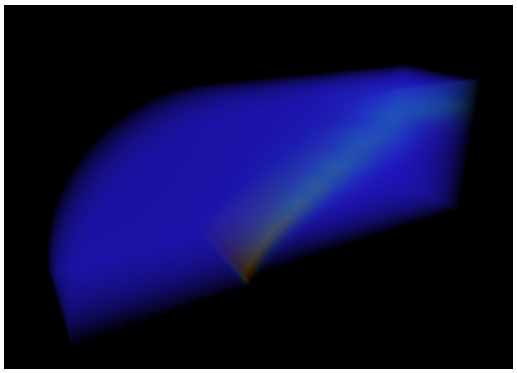


5.3(g): Difference $Hexa_{ACC}$

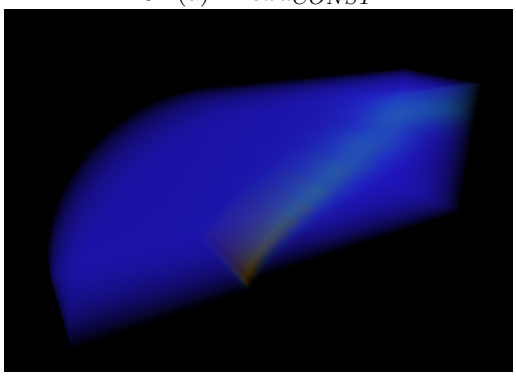
Figure 5.3: Images of the Bucky model.



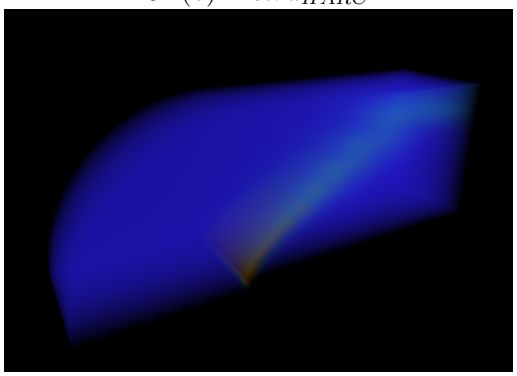
5.4(a): $Hexa_{CONST}$



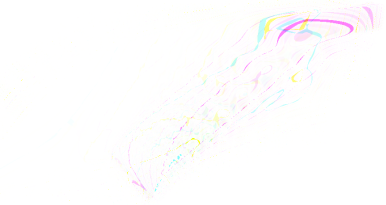
5.4(b): $Tetra_{HARC}$



5.4(c): $Hexa_{FAST}$

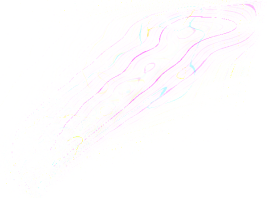


5.4(d): $Hexa_{ACC}$



5.4(e): Difference $Tetra_{HARC}$

5.4(f): Difference $Hexa_{FAST}$



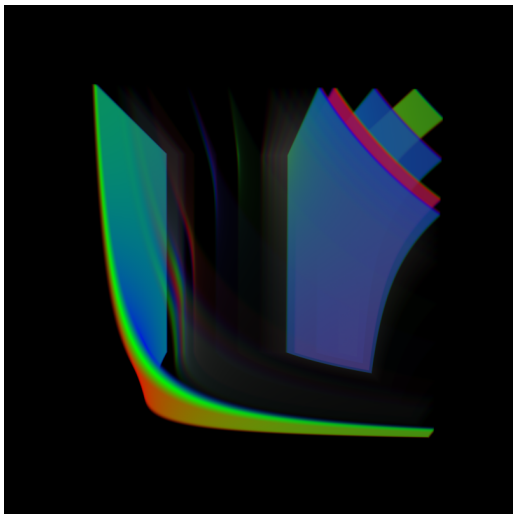
5.4(g): Difference $Hexa_{ACC}$

Figure 5.4: Images of the Bluntfin model.

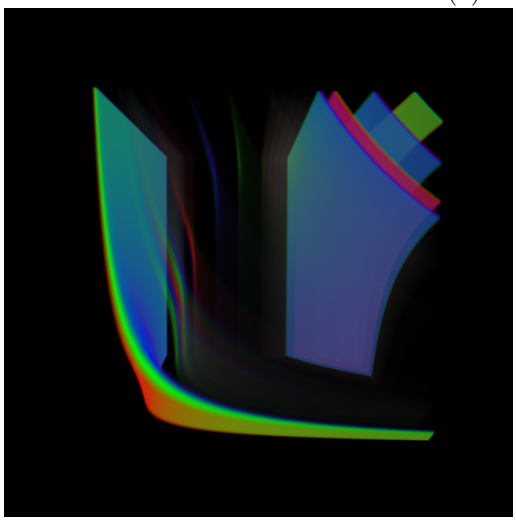
5.1.3
Comparassion with regular data rendering

We also compared our algorithm with a regular data ray-casting, detailed in Appendix D. Figure 5.5 shows the difference between the different algorithms. To highlight the differences, we used a transfer function with 100 control points.

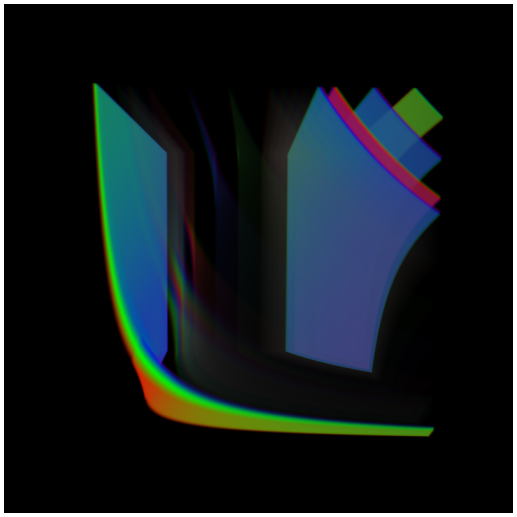
As can be seen, our proposal (both using quadrature and linear approximation) presents a better representation of the volume. Due to the trilinear interpolation during the 3D texture fetch, the rendering of the regular ray-casting algorithm does not appear as smooth as our proposal. Another problem is that, even with a 100 steps pre-integrated table, the regular ray-casting algorithm misses some of the transfer function control points, unlike our proposal, that stops at every control point.



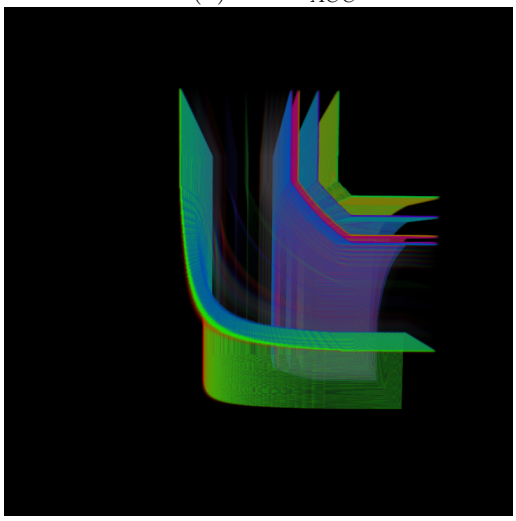
5.5(a): $Hexa_{Const}$



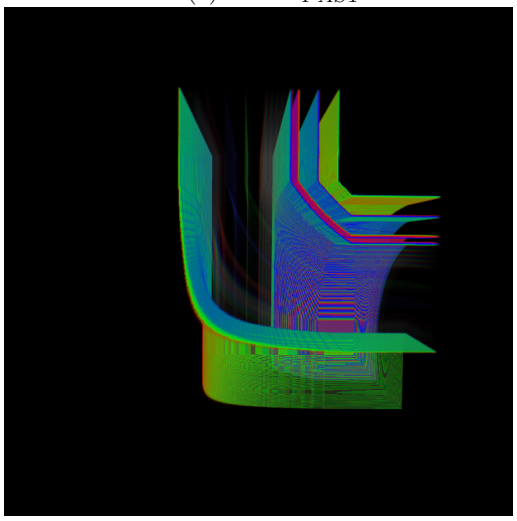
5.5(b): $Hexa_{ACC}$



5.5(c): $Hexa_{FAST}$



5.5(d): Regular ray-casting, using pre-integrated table



5.5(e): Regular ray-casting

Figure 5.5: Achieved images using a synthetic model.

5.2 Time results

Table 5.2 shows a comparison of memory consumption and rendering time. The time reported in the table represents the rendering time for one frame. For each entry, we repeated the experiment 5 times, and, for each run, we changed the camera to 64 different positions, averaging the measured time.

As can be noted, when compared to the subdivision scheme of 6 tetrahedra per hexahedron cell, our proposal reduces memory consumption by a factor of 2.2. Our algorithm also presents competitive performance. The rendering time of our Hex_{ACC} algorithm is in average 12% worse than the algorithm based on the subdivision scheme. To achieve accurate results, we need to perform part of the integration computation in double precision. That is the main reason for losing performance. If we had used single precision, our algorithm would be 15 to 20% faster than the subdivision scheme, but this would bring numerical inaccuracy. Figure 5.6 presents a comparison of the Bluntfin model rendered with both single and double precision. As can be noted, the use of single precision does result in inaccurate results.

Our $Hexa_{FAST}$ algorithm, however, presents a performance around 12% better than the $Tetra_{HARC}$ algorithm.

Model	Algorithm	# cells	Mem (MB)	Time (ms)
Fuel	$Tetra_{HARC}$	1,572,864	144	76.08
	$Hexa_{ACC}$	262,144	62	79.23
	$Hexa_{FAST}$	262,144	62	59.74
Neghip	$Tetra_{HARC}$	1,572,864	144	89.82
	$Hexa_{ACC}$	262,144	62	109.23
	$Hexa_{FAST}$	262,144	62	80.47
Oxygen	$Tetra_{HARC}$	658,464	60.2	32.46
	$Hexa_{ACC}$	109,744	25.9	35.73
	$Hexa_{FAST}$	109,744	25.9	29.62
Bluntfin	$Tetra_{HARC}$	245,760	22.5	27.64
	$Hexa_{ACC}$	40,960	9.6	31.60
	$Hexa_{FAST}$	40,960	9.6	25.89

Table 5.1: Rendering times and memory footprint of the subdivision scheme and our proposal.

5.6(a): Single precision 5.6(b): Double precision

Figure 5.6: Volume rendering of the Bluntfin Dataset.