EARTH OBSERVATION DATACUBES MULTI-VISUALIZATION TOOLBOX

Christophe Muller[1], Antoine Lestrade[1], Mathias Marty[1], Artan Sadiku[1], Joep Neijt[2], Yann Voumard[2], and Stéphane Gobron[1*]

[1] Department of Engineering, University of Applied Sciences West Switzerland (HES-SO), Neuchâtel, Switzerland.
(christophe.muller, antoine.lestrade, artan.sadiku, stephane.gobron)@he-arc.ch, mathias.marty@epfl.ch
[2] Solenix Schweiz GmbH, Härkingen, Switzerland. - (joep.neijt, yann.voumard)@solenix.ch

Commission IV, WG IV/4

KEY WORDS: Earth Observation, Data Visualization, Raycasting, 3D Rendering, Real-time visualization, GPGPU, WebGL

ABSTRACT:

Too much information often kills information. With the increasing number of satellites and their ever-increasing performance, new tools must be made available to deal with this data onslaught. We noticed that a number of computer graphics tools were largely under-exploited to help scientists better interpret and find relevant information in large datasets. A modern approach to run large processes efficiently is the use of GPUs, but nowadays the emphasis is often put on the parallel processing of geospatial datasets rather than focusing on their visualization. Considering geospatial data using GPU resources for intermediate computation and visualization is this paper main contribution. Given the increasing interest in interacting directly with this data using Web pages or Notebooks, this article presents tools allowing a program to run on the GPU and display, using the WebGL API, these matrices of data, also called datacubes. This paper shows a range of models applicable to datacubes deployed in the context of terrestrial observation. The end goal is to display on a PC very large (*i.e.* 1024^3) datacubes rendered on the fly and in real time. Furthermore, results show our models can process large amounts of data and render them in real time. All these innovative rendering models are assembled in a toolbox dedicated to datacube visualization. Finally, we give several examples of how to use this toolbox which enables the retrieval of raw data from an external server to real-time rendering on a local Web page.

1. INTRODUCTION

Data visualization is at least as important as the data itself. As the amount of data generated nowadays is getting bigger and bigger, we need more efficient tools. Earth observation (EO) generates multidimensional large-sized data, called datacubes. Datacubes can have different formats combining spatial and temporal dimensions such as (x, y), (x, y, z), (x, y, t) or even (x, y, z, t). A representation of these datacubes is shown in Figure 1. Data scientists need adequate and efficient tools not only to display these datacubes, but also to highlight pertinent data intuitively.

With more and more EO data becoming available, especially from satellites programs like Copernicus (Jutz and Milagro-Pérez, 2020), there is a need for an easier access. Efforts to ease the use of such data have led to the development of the Rasdaman server concept (Baumann, 1993, Baumann et al., 1998) and nowadays open source solutions like OpenEO are becoming more prevalent (Pebesma et al., 2018).

Datacubes are a reference to handle EO data (Liang et al., 2014, Baumann et al., 2018) and several works have already detailed techniques to visualize them (Kruger and Westermann, 2003, Gobron et al., 2011, Hassan et al., 2012). NASA has also developed *Web WorldWind*, an open-source interface to visualize satellite data on a virtual globe (Bell et al., 2007). It provides interactive features and is meant to be integrated in web pages.

In more recent works we find a focus on the preparation of large scale geospatial data (Mazroob Semnani et al., 2020), which is a highly technical subject and could benefit from some optimizations. Specifically for airborne LiDAR data, researchers developed a structure using a 3rd-party application (CesiumJS) to handle and display this 3D data in a web application (Vo et al., 2020). QGIS is another tool frequently used in EO, it can be enhanced by developing new plugins (Rufin et al., 2021), for example to plot timeseries of data from the Google Earth Engine cloud processing platform, thus interacting directly with data on the web. There are also several developments made in the web and mobile application fields (Lühr Sierra et al., 2021), which tackle the problem of waste management in a city by developing a web and mobile application to track the route of waste collection trucks via GPS satellites, with a focus on the user experience. The emphasis on the user is central when developing with Jupyter notebooks too as they can be used to query, manipulate, assemble and visualize spatial data in an educational context (Camara et al., 2021).

The efficiency issue is ever so present in geospatial data, due to the large scales that have to be dealt with. A more modern way to tackle it is to use GPUs instead of standard CPU processing. When reviewing the use of GPUs to process geospatial data, the emphasis is often put on the parallel processing of big geospatial datasets but not on their visualization (Saupi Teri et al., 2022). The next necessary step is the visualization of geospatial data using GPU resources.

As overall there is more and more data being put online, we observe an increasing interest in interacting directly with this data using web pages. This article will present tools enabling a program to run on the GPU and display the desired datacubes in an HTML canvas that uses the WebGL API. This can result in high performance visualizations thanks to its low level control and possibility of using GPGPU programs. With WebGL, running natively on most web browsers, another benefit will be the end-user ease of use since no additional software has to be installed.

* Corresponding author

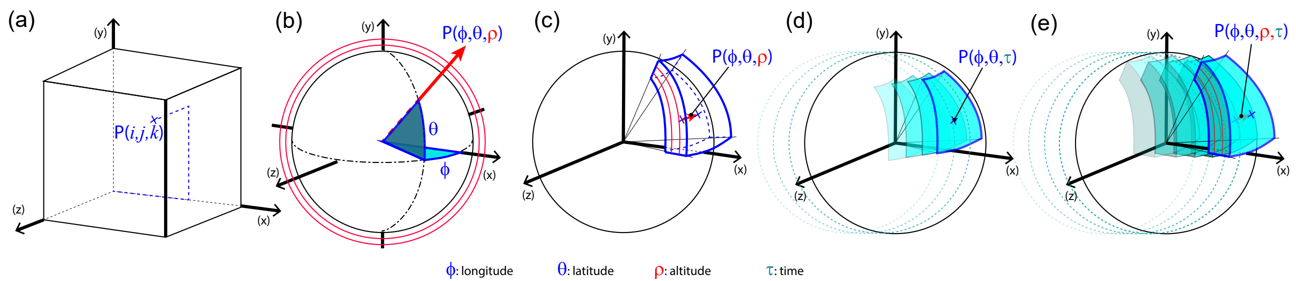


Figure 1. Actual representation of possible datacubes: (a) 3D Cartesian (x, y, z) datacube; (b) spherical (ϕ, θ, ρ) datacube with ρ constant; (c) partial multi-layer spherical (ϕ, θ, ρ) datacube with ρ as a variable; (d) time-related partial spherical (ϕ, θ, t) datacube; (e) time-related partial multi-layer spherical (ϕ, θ, ρ, t) datacube.

To oversee the project an advisory group of international experts was formed to represent the interests of actors in the EO industry and academic research. They are mainly associated with the European Space Agency (Europe), EURAC (Italy), GISAT (Czech Republic), Terrasigna (Romania), TU Wien (Austria), VITO (Belgium) as well as a former NASA (USA) analyst. They have been regularly interviewed to get constant feedback on our developed application suitability.

The end goal of our project is to build a toolbox – we titled *CubeViz4EO* – of models to efficiently visualize large datacubes of different formats on the fly and in real time. This will be done by fetching data directly from a rasdaman server as described in Figure 2 and processing it directly on the GPU to render the result locally in a web page.

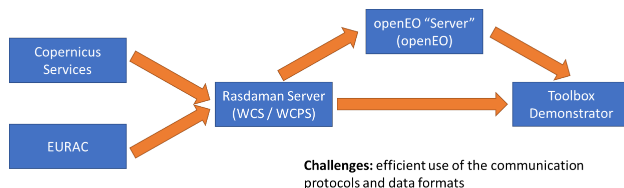


Figure 2. Relationship between data servers, Rasdaman server and our demonstrator toolbox providing the end user with data visualizations.

This paper is organized as follows: first of all, there is no state of the art section dedicated to the presented rendering models, because two full papers of co-authors (accepted in parallel) describe them in depth (Lestrade et al., 2022, Marty et al., 2022). The section 2 presents a set of innovative approaches to visualize Earth observation data: (1) discrete and spline-based implicit curves models; (2) massive 3D volume raycast: x-ray, implicit-surface simulation, and derived raycasting models. Corresponding results are illustrated in the section 3 and discussed in section 4. Finally, the section 5 concludes the paper by including a set of potential perspectives.

2. EARTH OBSERVATION DATA VISUALIZATION MODELS

The models presented in this paper include several visualizations: the level lines of 2D (x, y) datacubes with two rendering methods called *implicit curves* in 2D or 3D, the *derived raycasting* for (x, y, t) data to show differences between a base layer and a given time range. Other models were initially developed to handle (x, y, z) datacubes, but observations from the advisors panel mentioned that relevant EO data mostly consisted of 2D and 2D with time dimension formats. So the main goal is to display 2D and 3D data, (x, y) and (x, y, t), in real-time and in the most intuitive way for the user. In our approach to reduce the complexity of processing these large datasets,reducing the amount of to-be-displayed data is a basic but efficient method. When rendering 3D datacubes our tools only process the fraction of the data the user is interested in, whether it is a time range or a spatial region depending on the selected datacube format.

2.1 Implicit curves: discrete and math-based models

Curves are extracted from a gray scale two-step image illustrated in Figure 3. First, we use an *iso* value as a threshold to binarize the image. The obtained binary image is composed of zeros and ones. The border of the regions containing ones are the curves we are interested in and we extract them by using a convolution product with two kernels for edge detection. All these steps are done in shaders, so they can be executed parallel for each pixel on the GPU. Once this processing is done, the curves are rendered and stored in another 2D texture to be accessed directly.

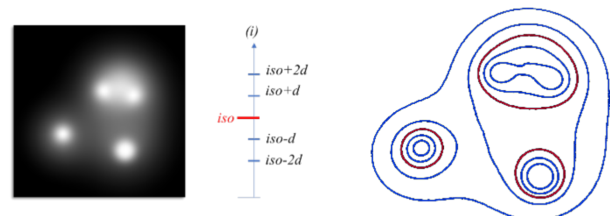
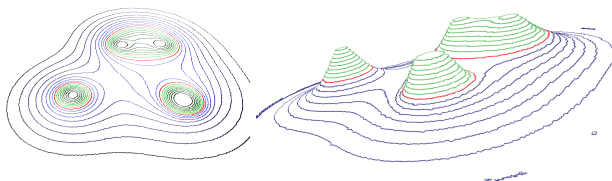


Figure 3. 2D curves concept: a desired *iso* and δ values are selected and corresponding levels from a scalar heightmap (left) are extracted and processed as curves (right). The *iso* reference level is in red.

Rendering of 3D surfaces with curves uses the displacement map algorithm, which is parallelized for each pixel in the shader. This shader then tests if each pixel lies inside a curve. This is equivalent to projecting the curve onto the surface. If it does, it colors itself with a different color so that we distinguish the curve of the surface. This enables surface rendering while also displaying the curves. By clicking on the surface, the user can get that point's (x, y, z) values, which is done by creating a G-Buffer using a deferred rendering technique. It is created with the displacement map algorithm in a first render pass, and then the final render is performed in a second render pass using the information contained in this G-Buffer. The G-Buffer is a framebuffer texture and stores information such as depth, texture coordinates of the base binary image and other values needed for the final render in the second pass. These values

differ according to the camera point of view, because they are related to the screen space. This should be understood as an intermediate view of the final render containing only the necessary information for the second render pass, which applies the true colors on the surface, the illumination model and any additional effects on the screen. When a click is detected on the screen, it is possible to sample the G-Buffer for that position and obtain specific information encoded within this buffer.

The surface color is computed from another texture representing this same region data. The user can use real colors (with a set or customized color palette) or colors representing some other information, thus cumulating the visualization of two different types of data.



(a) 2D curves (b) 3D level curves equivalent to Figure 4a.

Figure 4. Renderings of heightmap from Figure 3 with *iso* level 120 and $\delta = 15$. *Iso* lines are red, lines above *iso* level are in shades of green, lines below in blue.

Although level curves can be displayed on the surface, their width is not constant since they are projected into the surface. To solve this problem, we can use a more advanced method described in (Marty et al., 2022). The latter proposed a way of storing and rendering these curves using WebGL primitives, thus enabling constant width of the curve. A comparison of the two methods is shown in the results, in Figure 10. This is achieved by first computing the curves as sequence of points, then storing only a small subset of points to finally reconstruct a curve approximation using this list of points. On the whole reference gray scale image, a compression ratio of 3% can be achieved with this method.

Using a surface would be another way of displaying a desired level on a 3D map; having a horizontal plane cutting through the topography makes it easier to see what areas are above or below the level. Representing a desired *iso* with a surface is particularly useful in the context of water levels. The simulation of water surfaces is especially pertinent in the case of natural disasters like floods, global rising water levels, or infrastructure failures (dyke or dam break). Displaying water levels gives us the opportunity of using the light refraction optical effect described by Snell-Descartes law, which can be directly implemented on the GPU. As we reproduce the natural optical effect we experience in real life, this visual effect applied to the render gives a more intuitive way of displaying a water surface.

Each cast ray progresses in 3D space by steps of minimum distance in order to sample all discrete values encountered on the path. This raycast operation can be more or less intensive in terms of performance depending on the size of the 3D space. To help smaller hardware configurations achieve a smooth and stable rendering, we implemented a dynamic precision system on the raycast algorithm. The method consists of increasing or decreasing the cast rays step size according to the frame number per second in order to obtain an average performance level around 30 frames per second. If the number of frames per second is higher than 30, the algorithm tries to increase accuracy, thus decreasing the step size, and inversely if the number of frames per second is lower than 30. This feature is optional but allows a performance gain at the expense of the final rendering quality.

2.2 3D volumes raycasting

We have developed three rendering models for the visualization of 3D datacubes of dimension (x, y, z) and (x, y, t) using a raycasting algorithm. As WebGL version 1.0 has no 3D textures support, we must represent these 3D data in a 2D space as a standard image. The datacube is decomposed into layers along one of its dimensions and, following a special arrangement and indexing, we can find any point in the datacube space on a 2D texture (see Fig. 5). These algorithms can currently support datacubes of dimensions 16^3, 64^3, 256^3 and 1024^3. However, a 1024^3 datacube is represented by a texture of dimension $32'768^2$, which exceeds the maximum dimension supported for 2D textures in WebGL. Our solution to reduce the dimension by half is to compress the base image using RGBA color channels of a smaller texture. In our case, a datacube contains only one discrete value per voxel, so its representation in 2D space is a grayscale image and the RGBA values are redundant. The idea is to separate the datacube image into four equal parts and combine each into a smaller image color channel (illustrated in Fig. 6). Thanks to this method, the pixel number and the space occupied in memory are reduced by a factor of 4, enabling us to load datacubes of dimensions 1024^3 in our rendering models. More details on these optimization methods can be found in the twin paper "Real-time Renderings of Multidimensional Massive DataCubes on Jupyter Notebook" (Lestrade et al., 2022).

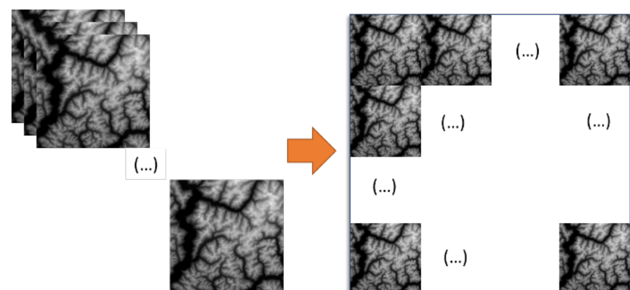


Figure 5. Sorting a 3D dataset into a 2D texture to improve performance.

2.2.1 X-Ray rendering

We designed two models that are specifically adapted for the visualization of 3D datacubes with (x, y, z) dimensions. The first is an X-Ray-like rendering model, which allows the visualization of 3D spatial data in the form of a volume with gray level transparency in the scene. The raycasting algorithm, as shown in Figure 7, accumulates the values encountered for each ray until it reaches the limits of the explored volume. This model gives an appearance similar to an X-Ray image, hence its name.

2.2.2 Implicit surface simulation

This rendering model for (x, y, z) datacubes is a simulation of implicit surfaces for 3D data. This model is a simulation because the surfaces are computed and only exist in the screen space context and not in an algebraically correct context. Here, the rays do not need to accumulate values but must stop as soon as a surface is encountered in the volume (as shown in Figure 8). The boundary of this surface is defined by a threshold *iso* value and a color is applied according to a color matching table. In addition, a normal vector is computed by sampling and averaging the 26 neighboring

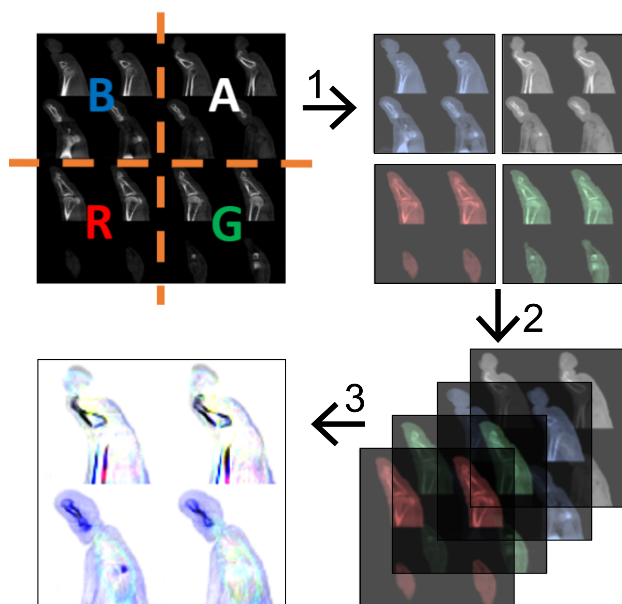


Figure 6. From left to right : (1) Slicing the image into 4 parts, each corresponding to a color channel, (2) Combining each part into one smaller RGBA image, (3) The resulting image is 4 times smaller in size than the original.

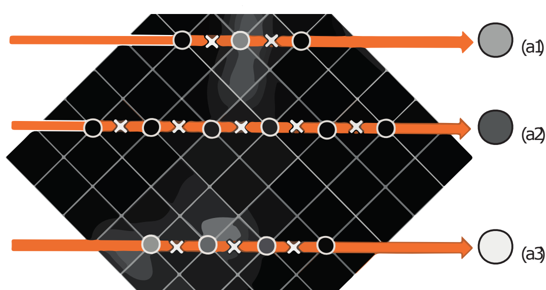


Figure 7. Representation of how the voxels are sampled by the raycast algorithm using the X-Ray model: voxel values are accumulated along the rays.

voxels. Then a Phong illumination model is applied on it to highlight these surfaces.

2.2.3 Derived raycasting The Derived model is one of our three rendering models specially designed for 3D datacubes of dimension (x, y, t) where t is a time dimension. With this model, the aim is to visualize the evolution in time of a geographical area by highlighting the temporal differences within a volume. By choosing a reference layer representing the state of an area at a defined time and a time interval, the differences between the reference and this time range are visible in color in the sub-volume defined by the interval (see Fig. 9). Positive differences are shaded in red and negative differences are shaded in blue while the reference layer is colored in gray. In order to add more contextual information in the visualized geographical area, we added the possibility to display an additional map at the reference layer level within the volume. In our case, thanks to the API of the Mapbox service, we could display a topographic map corresponding to the geographical area of the visualized data.

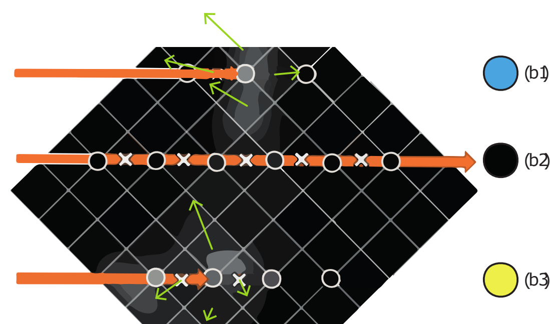


Figure 8. Representation of how the voxels are sampled by the raycast algorithm using the implicit surface model: samples the first voxel value at a given threshold is sampled.

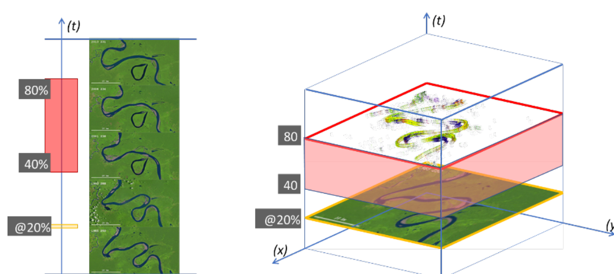


Figure 9. Representation of the rendering model with derivatives: (left) stack of images, the selected range is in red and the reference layer in yellow; (right) illustration of a potential view.

3. RESULTS

In this section, we present different examples of our rendering models in specific use case contexts as well as their integration in our Web visualization toolbox. The data shown here were rendered with the highest precision possible for the raycast algorithm to ensure maximum fidelity to the original data. All the visuals below were captured using a modern high-end hardware configuration (in 2022). The hardware and software configuration of the test machine is the following :

- OS : Windows 11 v21h2 64-bit;
- CPU : Intel Core i9-10850K @ 3.60GHz;
- RAM : 16GB DDR4;
- GPU : NVIDIA GeForce RTX 3070 8GB;
- Storage : SSD NVme 500GB;
- Web browser : Google Chrome 64-bit (with no v-sync).

The Table 1 is here to remind us which model we can find in what figure.

Table 1. List of figures representing each feature and rendering model described in this section.

Model / Feature	Figures
Implicit curves (IC)	10, 11
IC with 3D surface (3DS)	12, 13, 14
IC 3DS and water level	16
Custom colormap	15
X-Ray & implicit surface simulation	17
3D derived rendering	18, 21
Toolbox usecase	19, 20, 21

3.1 Implicit curves

Here are some application examples for the two models based on implicit curves. The scalar heightmaps used in this section have all a size of 2048×2048.

Firstly, using the implicit curves model without rendering the 3D surface allows the compression of level curves as illustrated in Figure 10. Here we compare the reconstruction method employing vectorized reconstructed curves with the naive discrete approach that only uses a raycasting algorithm.

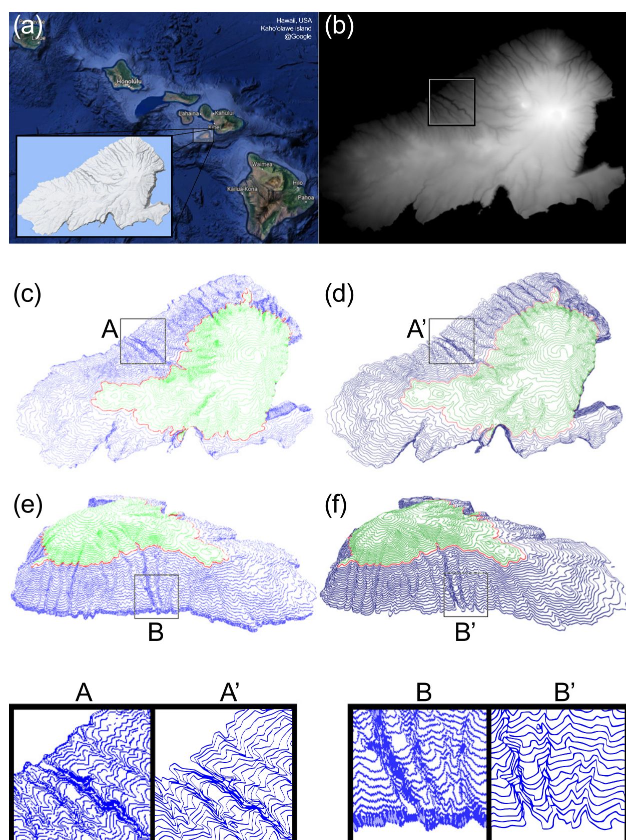
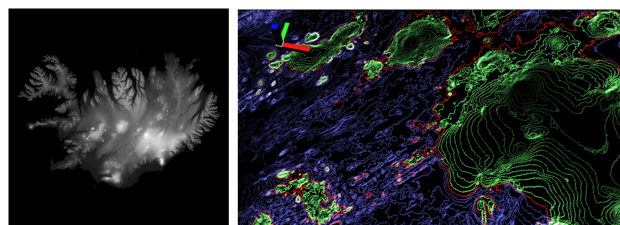


Figure 10. Implicit surface model with a real data set: (a) Kaho'olawe, Hawaii's island Google maps; (b) input depth map; (c) and (d) top views and (e) and (f) side views; discreet output are shown in (c)+[A] and (e)+[B]; and implicit curve output in (d)+[A'] and (f)+[B'].

Figure 11 illustrates the 2D implicit curves model applied to a high resolution scalar-map of Iceland. A set *iso* level is represented by red lines, whereas the green and blue lines represent levels above and below the *iso* value separated by the set δ value. The result is a semi-perspective view of the discrete contour simulation (Fig. 11b is a close-up view due to the large amount of details).

The same data (Fig. 11a) is used in Figure 12. Here the implicit curves are displayed in 3D on a simulated surface. In addition to the curves information, a colormap is applied to render the image values as a range of colors and an illumination model using diffuse reflection simulation gives a feeling of topography.

To add more context or combine the render with additional data of the same area, a second image can be used to replace the displayed colormap. Figure 13 illustrates this concept by using the same render as Figure 12, but coloring the surface with a map



(a) Iceland heightmap (b) 2D curves on Iceland map with small δ

Figure 11. Input test heightmap of Iceland (a) and close-up view of resulting 2D curves visualization (b).

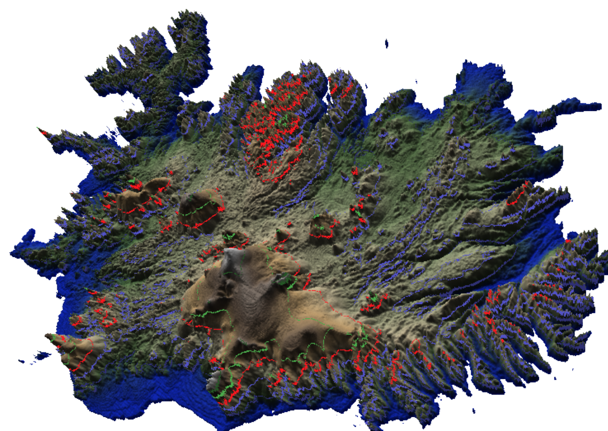


Figure 12. Implicit curves render with 3D surface simulation performed on the Fig. 11a scalar heightmap.

of rainfall data. This last result provides an interesting observation: indeed, as this simulation allows to navigate *inside* the information of relief and humidity, correlations between precipitations and the mountain range appear clearly.

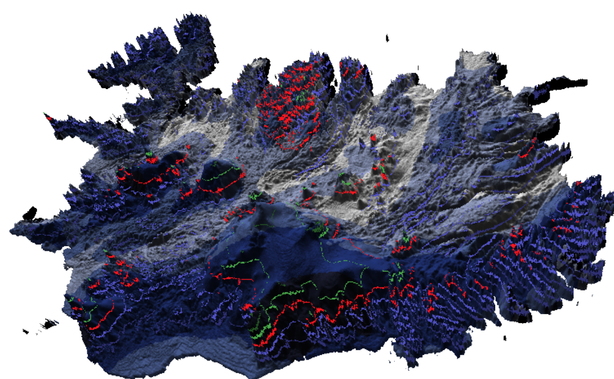


Figure 13. Render identical to Fig. 12 using rainfall data to change the surface colormap.

Figure 14 shows the 3D surface as initially rendered in Figure 12 with seven steps corresponding to the *iso* and δ levels instead of the usual more regular surface. This is convenient to bring together specified ranges of values. This render answers the needs of information specialists who had expressed their desire to easily visualize information in increments. Indeed, in this article we mostly used elevation maps, but specialists will often deal with data that do not only represent elevation, but also terrain categories, moisture rates, agricultural or urban areas.

https://doi.org/10.5194/isprs-archives-XLVIII-4-W1-2022-321-2022 | © Author(s) 2022. CC BY 4.0 License.

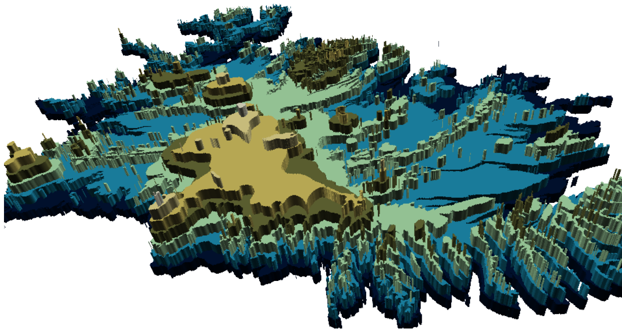


Figure 14. Rendering with steps: a first plateau is defined by the *iso*, the others are $\pm k\delta$ away.

The 3D implicit curves tool also includes the ability to define custom colormaps, another feature that the expert group suggested we add. This option is a simple and on the fly way to visually identify data with specific values. Figure 15 illustrates the versatility of this option on another synthetic heightmap.

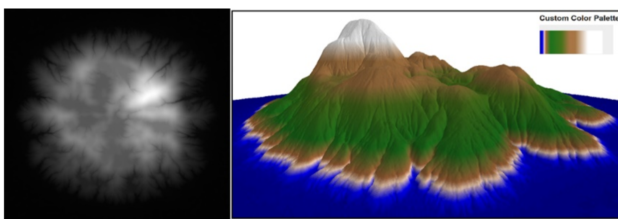


Figure 15. Illustration of creating a custom colormap to render a virtual heightmap (left) ; It is worth noting the paradigm shift in the interpretation of information.

The water surface simulation is illustrated in Figure 16, which shows the use of a synthetic heightmap rendered with the 3D curves model and fills the terrain up to a given *iso* with water. In this example, with a surface size of 2048×2048, this model can achieve real-time rendering with an average of 360 FPS (or 450 FPS without the water surface simulation).

3.2 3D X-Ray & implicit surfaces

Figure 17 shows the renderings for 3D datacubes using the X-Ray and implicit surface simulation. In this case, we display a datacube of size 256^3 representing a CT scan of a foot. In fact, these two types of rendering are well suited for 3D (x, y, z) datacube visualization and this kind of data is common in medical imaging, e.g. CT scans. That is why we use them in our results to provide the optimal visual representation of these rendering models. After testing these models with a 256^3 datacube on our test machine, we get an average of 170 FPS for the X-Ray model and an average of 160 FPS for the surface simulation model. We also obtained real-time renderings for datacubes of size 1024^3, but with a reduced number of samples for the raycast algorithm. Further details on our performance results with these models are described in (Lestrade et al., 2022).

3.3 3D derived rendering

Figure 18 shows an example of 3D derived rendering, with a datacube of size 256^3. From a selected reference layer and a given time range within the (x, y, t) datacube, the variations at a given location are rendered as shades of red and blue. The option to display more geographical context is displayed in Figure 18b where the base layer is replaced by a topographic map

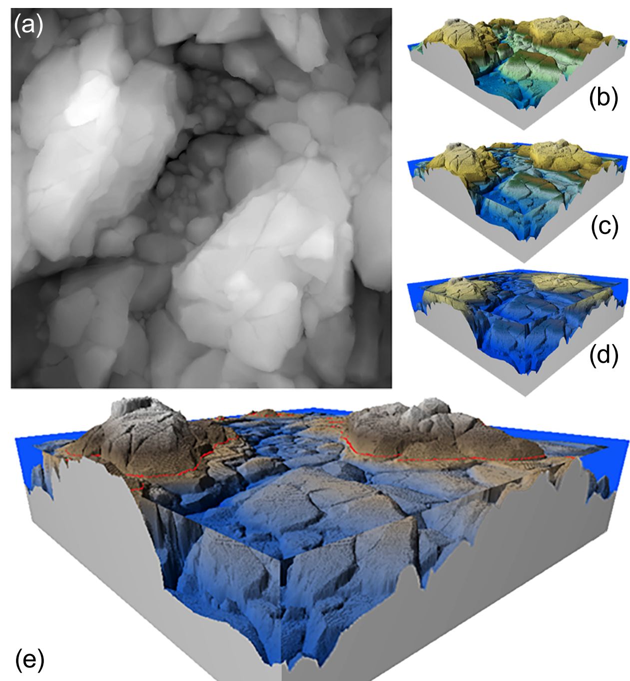


Figure 16. Example of the water level simulation in a 3D surface render on a virtual terrain: (a) synthetic heightmap; (b) standard 3D render with low water level and (c) and (d) simulations with rising water levels; (e) illustrates a close-up view with another ground color palette.

of the current location. In terms of rendering performance on our test machine, we can reach an average of 130 FPS for a datacube of size 256^3 and a full sampling of voxels along the rays with this model. For a billion cell datacube, we have measured real-time performance around 100 FPS, but with a number of samplings reduced to 1% of the total number of voxels. (Lestrade et al., 2022) describe in more detail the comparison of our performance metrics according to the datacube size and the number of samples for this particular model.

3.4 Toolbox integration

All the results presented so far in section 3 have been implemented in the *CubeViz4EO* visualization toolbox. The following figures (19, 20, 21) show the implementation of a web application used to run the toolbox models and designed for the end-user. This toolbox can retrieve a list of coverages from WCS servers as mentioned earlier in Figure 2, the user is then able to load a coverage from the server into a visualization. Before using a rendering model, a map shows a satellite view of the selected region, which can be modified on the fly as illustrated in Figure 19.

Figure 20 shows the basic render for a selected coverage corresponding to a 3D (x, y, t) datacube. As detailed in section 2.2 and illustrated in Figure 5 the data has been converted to a 2D GPU texture. In Figure 21 we show the resulting render using the 3D derived model visualization, where the user can freely select the desired base layer and range to display.

This toolbox includes all our render models and the user can display each one of them depending on their use case and the data he has access to. The Angular based web application integrates the rendering models by using a custom TypeScript library which can be extended to include other rendering models in the

https://doi.org/10.5194/isprs-archives-XLVIII-4-W1-2022-321-2022 | © Author(s) 2022. CC BY 4.0 License.

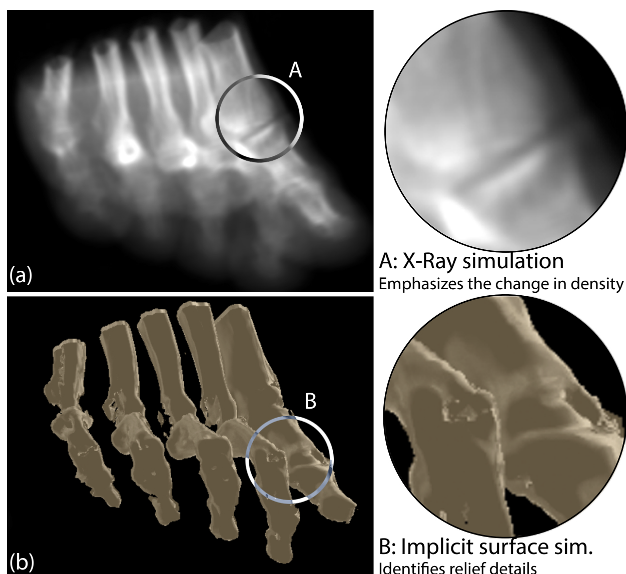
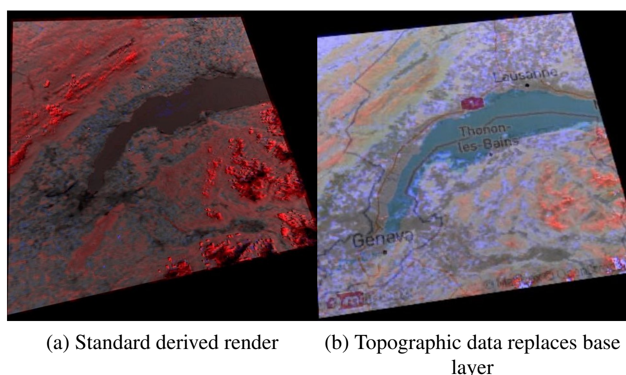


Figure 17. Two volume rendering on a (x, y, z) medical datacube: (a) X-Ray-like rendering model and (b) simulation of implicit surfaces.



(a) Standard derived render (b) Topographic data replaces base layer

Figure 18. Derived rendering illustration: red/blue areas show positive/negative differences with the selected base layer. In (b) the gray base layer is replaced by a topographic map.

future and can also be employed in other tools to facilitate the rendering capabilities reuse.

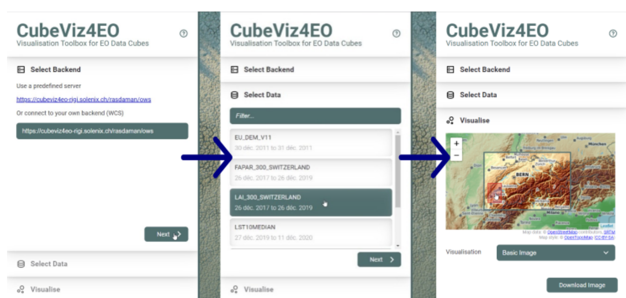


Figure 19. Toolbox process: accessing raw data server (left); setting up specific data set (middle); selecting regions (right).

4. DISCUSSION

Results show our models can process large amounts of data and render them in real time. Where large 3D datasets would normally become problematic to handle for any GPU, we developed specialized tools to overcome software and hardware limitations. For instance, a 3D datacube can be sorted into a 2D texture to be directly loaded into GPU memory and improve performance. When the textures become too big to work with WebGL, their data can be split in the RGBA channels of standard 2D textures for a four-fold decrease in memory use. Furthermore, when displaying our rendering models, and in the case of machines without sufficiently powerful graphics cards, we propose to display only the data fraction that interests the user. The developed toolbox runs in any web browser supporting JavaScript since Jupyter notebooks are also widely used among data scientists; an implementation of our models focusing on large datacube rendering with Jupyter is described in another twin publication (Lestrade et al., 2022).

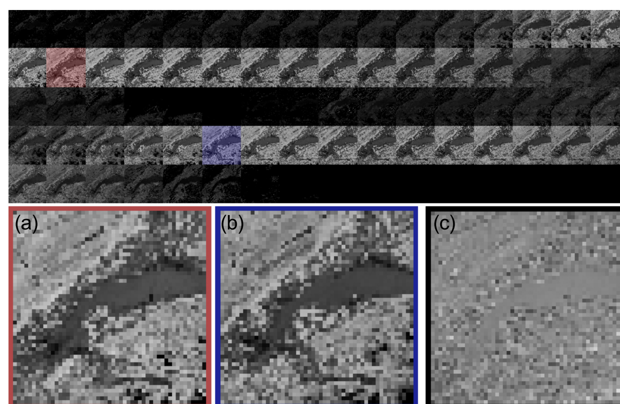


Figure 20. Result after selection from Figure 19: automatic accelerated 2D texture generation in toolbox. As shown with the red and blue samples –zoomed in (a) and (b)–, the direct derivation (c) to detect differences over time does not result into obvious interpretation.

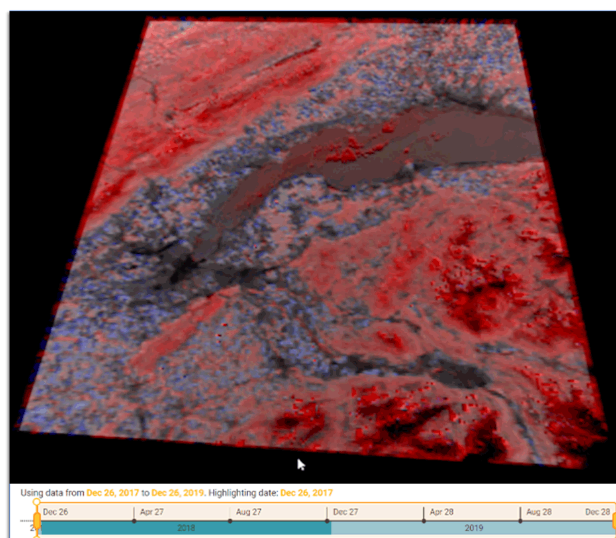


Figure 21. 3D Derived rendering corresponding to the texture generated in Figure 20.

Figures 13 and 18b showcase examples of additional context added to the render from external data. This data could also be another geographical context like borders or various weather data. The water level simulation for the *3D implicit curves* render is also a first step towards more realistic floods or watershed simulations, with our deferred rendering technique adapted to quickly generate water sources. Concerning the de-

veloped models mentioned for 3D datacubes in section 2.2, only the *derived rendering* for (x, y, t) data was presented with geospatial data, because this sort of data in (x, y, z) and (x, y, z, t) datacubes format is much less frequent.

5. CONCLUSION

In this paper we demonstrated an example application retrieving raw data from a server, formatting it for local use with GPGPU, and rendering it using several original models. The models produced innovative results for different datacube formats and sizes. Thanks to a web application and Jupyter Notebook integration developed alongside our models, the visualizations are usable even on modest computer configurations. The development does not rely on commercial third-party software and the rendering models presented do not build on specialised geospatial analysis software: they build directly on WebGL itself. The code for the models and integrated Web application will be freely available under the Apache 2.0 license from the GitHub repository of the *CubeViz4EO* project.

ACKNOWLEDGEMENTS

This work was supported by the Swiss Space Office / SEFRI, grant number 'REF-1131-61001' as a part of the *Enhanced Data Cube Visualisation for Earth Observation* project. We would like to thank the members of our advisory board for their support and advice. Special thanks are also due to Ms Gutierrez Montserrat for her precious English review of the manuscript.

REFERENCES

Baumann, P., 1993. Language Support for Raster Image Manipulation in Databases. ZGDV, Zentrum für Graphische Datenverarbeitung e. V., M. Göbel, J. C. Teixeira (eds), *Graphics Modeling and Visualization in Science and Technology*, Springer Berlin Heidelberg, Berlin, Heidelberg, 236–245.

Baumann, P., Dehmel, A., Furtado, P., Ritsch, R., Widmann, N., 1998. The multidimensional database system RasDaMan. *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, Association for Computing Machinery, New York, NY, USA, 575–577.

Baumann, P., Misev, D., Merticariu, V., Huu, B. P., Bell, B., 2018. Datacubes: A Technology Survey. *IGARSS 2018 - 2018 IEEE International Geoscience and Remote Sensing Symposium*, 430–433.

Bell, D. G., Kuehnel, F., Maxwell, C., Kim, R., Kasraie, K., Gaskins, T., Hogan, P., Coughlan, J., 2007. NASA World Wind: Opensource GIS for Mission Operations. *2007 IEEE Aerospace Conference*, 1–9.

Camara, G. S., Camboim, S. P., Bravo, J. V. M., 2021. Using jupyter notebooks for viewing and analysing geospatial data: Two examples for emotional maps and education data. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLVI-4-W2-2021, Copernicus GmbH, 17–24.

Gobron, S., Çöltekin, A., Bonafos, H., Thalmann, D., 2011. GPGPU Computation and Visualization of Three-Dimensional Cellular Automata. *The Visual computer*, 27(1), 67–81.

Hassan, A. H., Fluke, C. J., Barnes, D. G., 2012. A Distributed GPU-Based Framework for Real-Time 3D Volume Rendering of Large Astronomical Data Cubes. *Publ. – Astron. Soc. Aust*, 29(3), 340–351.

Jutz, S., Milagro-Pérez, M. P., 2020. Copernicus: The European Earth Observation Programme. *Revista de Teledetección*, V-XI.

Kruger, J., Westermann, R., 2003. Acceleration techniques for GPU-based volume rendering. *IEEE Visualization, 2003. VIS 2003.*, 287–292.

Lestrade, A., Marty, M., Sadiku, A., Muller, C., Neijt, J., Voumard, Y., Gobron, S., 2022. Real-time renderings of multidimensional massive datacubes on jupyter notebook. *ICGG 2022 Proceedings of the 20th International Conference on Geometry and Graphics*. in press.

Liang, J., Gong, J., Li, W., Ibrahim, A. N., 2014. Visualizing 3D Atmospheric Data with Spherical Volume Texture on Virtual Globes. *Computers & Geosciences*, 68.

Lühr Sierra, D. V., Balinos, M., Gatica, J., Lagomarsino, C., 2021. Ciudad Limpia Valdivia: A mobile and web based smart solution based on FOSS technology to support municipal and household waste. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLVI-4-W2-2021, Copernicus GmbH, 97–102.

Marty, M., Lestrade, A., Muller, C., Sadiku, A., Neijt, J., Voumard, Y., Gobron, S., 2022. Implicit curves: from discrete extraction to applied formalism. *ICGG 2022 Proceedings of the 20th International Conference on Geometry and Graphics*. in press.

Mazroob Semnani, N., Breunig, M., Al-Doori, M., Heck, A., Kuper, P., Kutterer, H., 2020. Towards intelligent geo-database support for earth system observation: Improving the preparation and analysis of big spatio-temporal raster data. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLIII-B4-2020, Copernicus GmbH, 485–492.

Pebesma, E., Wagner, W., Soille, P., Kadunc, M., Gorelick, N., Schramm, M., Verbesselt, J., Reiche, J., Appel, M., Dries, J., Jacob, A., Neteler, M., Gebbert, S., Briese, C., Kempeneers, P., 2018. openEO: An Open API for Cloud-Based Big Earth Observation Processing Platforms. 4957.

Rufin, P., Rabe, A., Nill, L., Hostert, P., 2021. GEE timeseries explorer for QGIS – instant access to petabytes of earth observation data. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLVI-4-W2-2021, Copernicus GmbH, 155–158.

Saupi Teri, S., Musliman, I. A., Abdul Rahman, A., 2022. GPU Utilization in geoprocessing big geodata: A review. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLVI-4-W3-2021, Copernicus GmbH, 295–304.

Vo, A. V., Laefer, D. F., Trifkovic, M., Hewage, C. N. L., Bertolotto, M., Le-Khac, N. A., Ofterdinger, U., 2020. A highly scalable data management system for point cloud and full waveform LIDAR data. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLIII-B4-2020, Copernicus GmbH, 507–512.