On the use of WebAssembly for rendering and segmenting medical images

Sébastien Jodogne[0000−0001−6685−7398]

Computer Science and Engineering Department (INGI),
Institute of Information and Communication Technologies, Electronics and Applied Mathematics (ICTEAM), UCLouvain, 1348 Louvain-la-Neuve, Belgium
sebastien.jodogne@uclouvain.be
https://www.info.ucl.ac.be/~sjodogne/

Abstract. Rendering medical images is a critical step in a variety of medical applications, from diagnosis to therapy. There is a growing need for advanced viewers that can display the fusion of multiple layers, such as contours, annotations, doses, or segmentation masks, on the top of image slices extracted from volumes. Such viewers obviously necessitate complex software components. But desktop viewers are often developed using technologies that are different from those used for Web viewers, which results in a lack of code reuse and shared expertise between development teams. Furthermore, the rise of artificial intelligence in radiology calls for Web viewers that integrate deep learning models and that can be used outside of a clinical environment, for instance to evaluate algorithms or to train skilled workers. In this paper, we show how the emerging WebAssembly standard can be used to tackle these challenges by sharing the same code base between heavyweight viewers and zero-footprint viewers. Moreover, we introduce a fully functional Web viewer that is entirely developed using WebAssembly and that can be used in research projects or in teleradiology applications. Finally, we demonstrate that deep convolutional neural networks for image segmentation can be executed entirely inside a Web browser thanks to WebAssembly, without any dedicated computing infrastructure. The source code associated with this paper is released as free and open-source software.

Keywords: Medical imaging, DICOM, image segmentation, Web applications

1 Introduction

The volume of medical images that are generated worldwide is in constant growth. This can be explained by the fact that the treatment of an increasingly number of chronic diseases requires a longitudinal and multimodal exploitation of medical images. For a highly effective, personalized treatment, it is indeed often important to consider images that have been acquired throughout the entire life of the patient and to combine multiple cues produced by several types

of imaging modalities. Artificial intelligence has enormous potential to assist radiologists in analyzing this growing body of imaging data [1].

The DICOM standard [21] specifies how medical images are encoded and exchanged. Fortunately, DICOM is nowadays adopted by any hospital dealing with digital medical images. Thanks to the fact that DICOM is an open standard, a large ecosystem of free and open-source software exists to handle medical images. For instance, when it comes to the need of storing, communicating and archiving DICOM images in full interoperability with the information systems of an hospital, free and open-source DICOM servers such as dcm4chee [30], Dicoogle [5] or Orthanc [10,11] can be used. Such free and open-source software can notably be used to deploy task-specific DICOM stores in the hospital that are dedicated to some medical specialisms or to some artificial intelligence application.

Obviously, once stored inside a DICOM server, the medical images must still be displayed. Various use cases exist for viewers of medical images: clinical review integrated within the electronic health records, specialized tools dedicated to one pathology or treatment, external access for the patient, quality control of modalities, second medical opinion, telemedicine, clinical trials... Moreover, the huge interest in artificial intelligence applied to medical imaging requires the development of advanced viewers that can be used to annotate or review regions of interest for the training of the algorithms, and to display the outputs of such algorithms (e.g. segmentation masks).

Depending on the clinical or pre-clinical scenario, one might either need a desktop software, a mobile application, or a Web platform to display the medical images. This led many community projects or commercial products to focus only on one of those three categories of platforms, or to have separate development teams working on distinct viewers for the several types of platforms. In the latter case, developing similar features in different languages depending on the target platform (JavaScript, C++, Java...) comes at a large cost because of non-shared source code. Furthermore, distinct code bases might suffer from different issues, or conversely might feature optimizations that are not available for the other platforms. Expertise is also less shared between development teams who are focused on different platforms and languages.

Two major open-source software libraries are currently used to render medical images. The first one is VTK [25], that is mature and mostly written in the C++ language. VTK is used by many heavyweight desktop viewers, including the 3D Slicer platform [15], the well-known OsiriX radiology application [24] and its fork Horos. Note that VTK is also often used for scientific visualization beyond medical imaging. The second library is Cornerstone3D [29], that is more recent and that is written in the JavaScript language to be used as a building block for higher-level Web applications. As it takes its roots in modern Web technologies, Cornerstone3D is at the core of many zero-footprint, lightweight Web viewers, notably OHIF [29], and is often encountered within cloud-based proprietary solutions.

The emerging WebAssembly standard (often abbreviated as "wasm") might be a game changer in this divided landscape between heavyweight and Web ap-

plications. WebAssembly is an open standard that defines a portable binary code (bytecode) for high-performance applications that can be run by Web browsers [8]. Its core specification was released as an official W3C recommendation in December 2019. WebAssembly is actively backed by the industry, and is now supported by the major Web browsers. Thanks to the Emscripten compiler, C++ source code can be transformed to WebAssembly bytecode, which can then be executed by Web browsers. This opens the path to the creation of C++ libraries that can be used both by native software (using a native compiler) and by Web platforms (using the Emscripten toolchain). Such C++ libraries could also be used by native mobile applications through Android NDK or an Objective-C wrapper for iOS. WebAssembly has thus an immense potential to improve code reuse in multi-platform developments, particularly in the field of medical imaging.

According to this discussion, the first main contribution of this paper is to propose a novel, free and open-source C++ library named "Stone of Orthanc," whose internal architecture was driven by the constraints of Web development while providing compatibility with the native compiler toolchains. The Stone of Orthanc library is designed to be as portable and versatile as possible, with a small footprint[1]. Historically, the Stone of Orthanc project started back in 2016 by leveraging the Google's PNaCl and Mozilla's `asm.js` technologies that are now superseded by WebAssembly. Additionally, this paper presents different applications for medical imaging that were built using the Stone of Orthanc library. In particular, the "Stone Web viewer" is introduced as a fully functional Web viewer for end users to review medical images.

A preliminary version of this paper appeared in the Proceedings of the 15th International Joint Conference on Biomedical Engineering Systems and Technologies (BIOSTEC 2022), which already introduced the Stone of Orthanc and the Stone Web viewer [14]. The present paper expands this preliminary work by exploring the topic of artificial intelligence algorithms applied to the automated segmentation of organs or biological structures in a Web environment.

Indeed, many clinical deployments of deep learning models require the medical images to be sent either to a dedicated on-premises server, or to a remote cloud infrastructure. The former on-premises paradigm requires a computation cluster to be deployed and maintained within the hospital, which is complex and costly for smaller facilities, whereas the latter cloud-based approach is often problematic with respect to the protection of clinical data, which requires careful legal agreements associated with an effective de-identification of the DICOM images. The complexity of this procedure also slows down the evaluation, and thus the adoption of artificial intelligence algorithms by hospitals. Furthermore, introducing skilled workers to artificial intelligence or making deep learning models

[1] This intimacy with the lightweight architecture of the Orthanc project for medical imaging [10,11] explains the name of the Stone of Orthanc library, which is a tribute to the *palantír* that resides inside the tower of Orthanc in *The Lord of the Rings* by J. R. R. Tolkien.

available to researchers is difficult in practice, as such processes typically require the access to some professional clinical deployment of artificial intelligence.

In this paper, we introduce an extended version of the Stone Web viewer that can locally execute deep learning models directly inside Web browsers, without requiring any dedicated infrastructure or network communication. More precisely, this extended Web viewer was designed to apply U-Net architectures, that have proved to be remarkably successful on segmentation tasks for biomedical imaging [23]. This is made possible by implementing a platform-independent C++ code in the Stone of Orthanc library to apply CNN (Convolutional Neural Networks) models to 2D images. Thanks to the fact that the Stone of Orthanc library also offers primitives to render scenes composed of multiple layers, the Stone Web viewer can overlay the segmentation masks that are produced by the U-Net models, on the top of their input medical images. This novel contribution demonstrates that WebAssembly provides a new framework for rendering and segmenting medical images.

2 Stone of Orthanc

As explained in the Introduction, Stone of Orthanc is a C++ library to render medical images, with the goal of being compatible with WebAssembly. In the current section, we review how the high-level architecture of Stone of Orthanc has been engineered.

2.1 Motivations

The objective of creating a library that can be used by heavyweight software, by mobile applications as well as by Web platforms puts strong constraints on the architecture of the library. Most importantly, traditional native applications are multi-threaded and sequential, whereas Web applications are single-threaded and driven by asynchronous callbacks (closures). Stone of Orthanc was designed to accommodate with such extremely different platforms by adding different abstractions that will be explained in Section 2.2.

Besides this constrained computing architecture, the accurate rendering of medical images requires a careful mapping between pixel coordinates and 2D or 3D physical coordinates. The library must also be extensible to display the various kinds of overlays (such as segmentation masks or annotations) that might have been produced by artificial intelligence algorithms or manually added by physicians. Furthermore, medical specialties such as nuclear medicine, radiotherapy or protontherapy necessitate to display fusions of different layers (for instance, a dose over a CT-scan, or a contour over a PET signal). To fulfill such needs, the architecture of Stone of Orthanc integrates a portable 2D rendering engine for vectorized graphics with support of overlays that will be discussed in Section 2.3.

Figure 1 provides a summary of the software architecture of Stone of Orthanc. Because the goal of this paper is not to document the internal components of

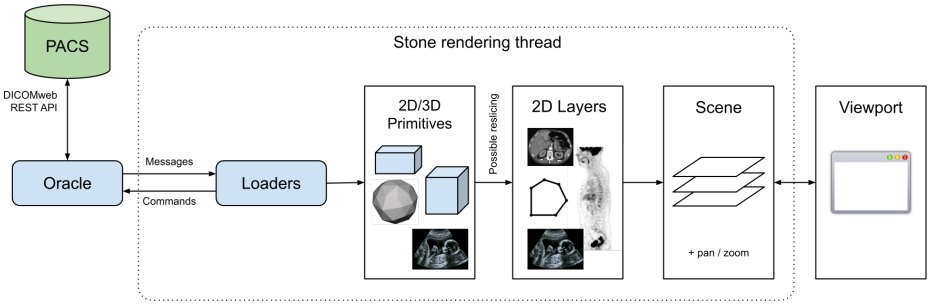

Fig. 1. High-level overview of the components of the Stone of Orthanc library [14]. The Oracle abstracts the runtime environment. The Loaders are responsible for loading the graphical primitives that have to be displayed. These graphical primitives are then converted into a set of 2D Layers (in particular, primitives corresponding to 3D volumes can be sliced along a cutting plane to extract a 2D Layer). The Layers are then superimposed to define a Scene that is finally composed onto the rendering surface of the Viewport. The user can interact with the Viewport to update the affine transform (pan/zoom) that is associated with the Scene.

Stone of Orthanc, only the main concepts that drove the design of Stone of Orthanc are presented. The interested reader is kindly invited to dive into the source code of the project.

2.2 Loaders and Oracle

The Stone of Orthanc library provides developers with many primitives that are necessary to create medical imaging applications. In particular, the library offers facilities to download images from a DICOM server (notably using the DICOMweb standard, but a custom REST API can be used as well). In this way, Stone of Orthanc features C++ classes that contain 2D or 3D medical images, and that are filled by classes that are referred to as "Loaders." In the case of 3D dense volumes, the associated Loaders automatically sort the individual 2D slices of the volume with respect to their 3D location[2].

However, as written above, the engines of JavaScript and WebAssembly are inherently single-threaded. Therefore, to be compatible with the Web runtimes, the core of the Stone of Orthanc library is not allowed to launch threads or to maintain a pool of threads for the Loaders to download the images, which contrasts with what would be done in a traditional C++ application. The solution adopted by Stone of Orthanc is to introduce a so-called "Oracle" singleton object that abstracts the way asynchronous operations such as downloads are processed on the target platform. The Oracle is the key component that enables

[2] The 3D location is determined by the "image position patient" `(0x0020,0x0032)` and "image orientation patient" `(0x0020,0x0037)` DICOM tags if available, or by the "instance number" `(0x0020,0x0013)` DICOM tag.

the writing of source code that can be run indifferently by WebAssembly and by native applications.

The Stone of Orthanc library provides a different Oracle for the different target platforms. In the case of WebAssembly, the Oracle boils down to a wrapper that calls the primitives offered the Web browser itself using the Emscripten C headers. In the case of native applications, the Oracle abstracts the system SDK associated with the target platform: For instance, to handle downloads, the Oracle internally manages a set of worker threads connected to a shared queue of pending HTTP requests. Besides downloading DICOM instances, other examples of asynchronous operations implemented by the Oracles include reading files from the filesystem, or sleeping for a certain amount of time.

According to this discussion, the Oracle object receives a stream of "Command" objects that specify the asynchronous operations that must be carried on. Following the observer design pattern [7], each Command to be processed is associated with an "Observer" object that will receive a "Message" object that contains the result of the Command once it has been processed. For instance, in the case of the download of a DICOM instance from some REST API, the Command would contain the URL to download, the Observer would correspond to the Loader object that populates the target medical image, and the Message would contain the raw bytes of the DICOM instance. Sending a Command to the Oracle is a non-blocking operation, which reflects the asynchronous nature of the Oracle. The Observer is specified as a C++ "pointer to member function," with a slight use of C++ templates as syntactic sugar, and the Oracle invokes this member function upon completion of the Command.

The Oracle is part of the global context of the Stone of Orthanc library: Each class that executes asynchronous operations must keep a reference to the Oracle. In order to map the single-threaded aspect of WebAssembly into native environments supporting multi-threading, it is assumed that the global context is only accessed from one distinguished user thread that is responsible for the rendering. A mutex is included in the global context to prevent the Messages emitted by the Oracle from interfering with the rendering thread. Reference counting using shared pointers ensures that the Observer objects are not destructed before all the scheduled Commands they are receivers of have been processed[3].

It is worth noticing that the Oracle mechanism can be used as an accelerator for certain operations that can be time-intensive. For instance, Stone of Orthanc introduces a Command to download then parse DICOM instances as a single operation: In the case of a multi-threaded environment, this allows to offload the DICOM parsing to a thread without locking the mutex of Stone of Orthanc, which brings more concurrency in native environments[4]. This also allows to maintain a cache of parsed DICOM files.

[3] The observer design pattern and reference counting are used elsewhere in Stone of Orthanc for sending messages from one object to another, and for broadcasting messages from one object to a set of Observers using a subscription mechanism.

[4] In the future, similar optimizations could be offered in the WebAssembly environment by leveraging Web workers.

2.3 Viewports and Layers

When it comes to the client-side rendering of medical images, the 2D operations of panning, zooming, and changing windowing must be as fast as possible. This calls for taking advantage of the 2D hardware acceleration that is provided by any GPU (graphics processing unit). To this end, both native applications and WebAssembly applications have access to the OpenGL API, the former through development libraries such as Qt or SDL, the latter through WebGL.

On the other hand, to reduce the network bandwidth in the presence of large images (typically 3D volumes), the developers of viewers might want to use streaming: In this scenario, a central server generates a stream of "screenshots" of a medical imaging scene, and those renderings are then sent to a thin client in charge of their actual display to the user. In the case of streaming, GPU is not necessarily available on the server, as multiple users might be connected to the same server. GPU might also be unavailable if printing an image, if generating a secondary capture image, or if the viewer runs on a low-power or embedded computer (such as the popular Raspberry Pi).

To accommodate with all such situations in a cross-platform and lightweight way, the architecture of Stone of Orthanc is designed to support both hardware acceleration (using straight OpenGL) and software rendering (using the widespread `cairo` drawing library). Furthermore, to maximize portability, the Stone of Orthanc library implements its own OpenGL shaders for the GPU-accelerated drawing of lines, texts, and bitmaps.

In Stone of Orthanc, each rendering surface (i.e. the application widget, the HTML5 canvas, or the memory buffer) is associated with a so-called "Viewport" abstract class that is responsible for the rendering of a 2D "Scene." A single application is allowed to manage multiple Viewports to show different Scenes.

The Scene is an object that encodes a superposition of "Layer" objects, each Layer being a 2D vectorized graphics with an alpha layer to allow for transparency. For instance, in a nuclear medicine application, a Scene would consist of three superimposed Layers, the foremost being the contours (a set of 2D polygons stored in a DICOM RT-STRUCT instance), the second being the texture associated with the PET-scan signal (with a heat colormap providing an alpha value for transparency), and the background being the CT-scan slice (a texture containing floating-point values expressed in Hounsfield units). Besides its constituting Layers, the Scene also contains a 2D-to-2D affine transform that maps the coordinates of Layers to the Viewport coordinates. Altering this transform is used to pan and zoom the Scene onto the Viewport.

Each Scene and Layer is agnostic of the actual rendering surface: It is up to the Viewport to compose the Scene as a bitmap, before putting it onto the rendering surface. Depending on the type of their associated rendering surface, Viewport objects will use a different algorithm to render a Scene, following the strategy design pattern [7]. This separation of concerns between Viewports and Scenes, is like the one between the Oracle and Commands. If a Layer containing a bitmap must be rendered, the Viewport is also responsible for the proper conversion of the pixel values (that typically correspond to floating-point num-

bers or 16 bit integers) to a bitmap that is suitable for the rendering surface (in general, red/green/blue/alpha values in 8 bits per pixel). This conversion must take into account the windowing parameters of the Layers, if any.

Summarizing, the rendering engine of Stone of Orthanc is inherently oriented toward 2D vectorized graphics. This choice is similar to that of the original Cornerstone.js library, but contrasts with the VTK library, that is more 3D-oriented. The advantage of focusing on 2D rendering is to make the rendering of multi-layer scenes more natural to the developers than if 3D must also be considered. Importantly, just like the Oracle/Commands/Observers architecture, this Viewport/Scene/Layers architecture is compatible with both native applications and WebAssembly applications. It is also worth noticing that Stone of Orthanc could be used as a rendering engine for vectorized graphics in native, mobile or Web applications out of the field of medical imaging.

Viewers for medical imaging need to display 3D volumes, not just 2D images. Section 2.2 explained how Loaders are used to download 3D images and store them into RAM. To render 3D volumes, Stone of Orthanc introduces the concept of "VolumeReslicer" that is an abstract class taking as its inputs one 3D image and one cutting 3D plane, and producing a 2D slice as its output. This 2D slice can then be injected as a Layer into the Scene to be rendered. Volume reslicers are currently implemented for MPR (Multiplanar Reconstruction, cf. Section 5.1) and for reslicing along an arbitrary cutting plane (cf. Section 5.2). To maximize the portability of Stone of Orthanc, the reslicing of volumes is done entirely using the CPU. Additional reslicers will be implemented in the future for MIP (Maximum Intensity Projection) rendering, possibly using the Oracle as an accelerator for the computations.

2.4 Sample desktop viewer

The distribution of the Stone of Orthanc library comes with the source code of a sample, basic viewer that provides an example about how Stone of Orthanc can be used to display one 2D slice of a DICOM instance downloaded from an Orthanc server. This viewer is a standalone desktop application that is started from the command line by providing the URL of the Orthanc REST API, together with the identifier of the DICOM instance to be displayed. This example uses SDL to manage its window, making it eminently cross-platform.

A screenshot of this application can be found in Figure 2. Zooming, panning, and windowing are supported, as well as taking measures. The mouse interactions are handled by a dedicated abstract class called "ViewportInteractor" that is responsible for tracking mouse events, and that is part of the Stone of Orthanc library. Interestingly, because the DICOM image is downloaded from the Orthanc server, it is decoded server-side, which frees the viewer from handling the various transfer syntaxes of the DICOM standard, hereby simplifying the source code. The full application only consists of 700 lines of code.

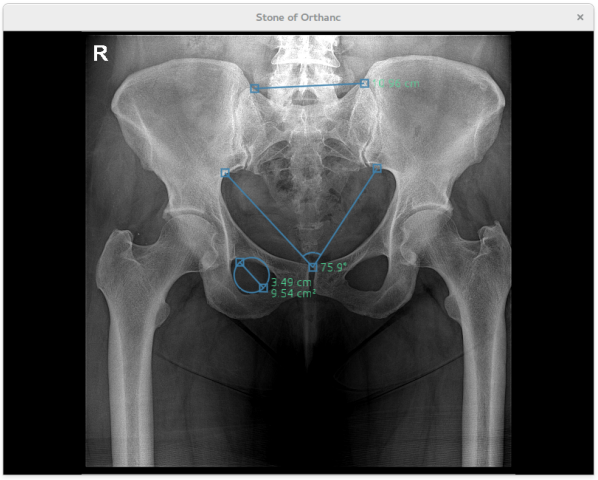

Fig. 2. Simple 2D viewer displaying a radiography image [14]. In this case, the Viewport is a SDL window, with one background Layer containing a 16bpp bitmap. The windowing can be changed using the mouse. The viewer can be used to take measures of lines, circles and angles in physical coordinates. These measures are geometric primitives stored in the foreground Layers of the Scene.

3 Stone Web viewer

The so-called "Stone Web viewer" is currently the main application that leverages the Stone of Orthanc library. The Stone Web viewer is a fully functional teleradiology solution, as depicted in Figure 3. It can simultaneously display multiple DICOM series, and it implements some advanced features such as measurements, synchronized browsing, 3D cross-hair, and 3D reference lines. The Stone Web viewer communicates with a DICOM server using the DICOMweb protocol, making it theoretically compatible with any PACS (Picture Archiving and Communication System) environment supporting DICOMweb.

To the best of our knowledge, the Stone Web viewer is the first viewer of medical images that entirely relies on WebAssembly. Most of its source code is written in C++ and compiled using the Emscripten toolchain. The buttons and the HTML5 canvas are managed by a small Vue.js application that calls the WebAssembly bytecode using an automatically generated JavaScript wrapper. Contrarily to the sample desktop viewer described in Section 2.4, the Stone Web viewer does not rely on the remote server to decode the DICOM files: It embeds the DCMTK library in the WebAssembly bytecode to this end.

The Stone Web viewer is a client-side application that is entirely run by the Web browser. Because its binary assets consist of a mere set of static resources (HTML files, JavaScript sources and WebAssembly bytecode), the Stone Web viewer can be served by any HTTP server (such as nginx, Apache or Microsoft IIS). To fulfill the same-origin security policy, the HTTP server must be config-

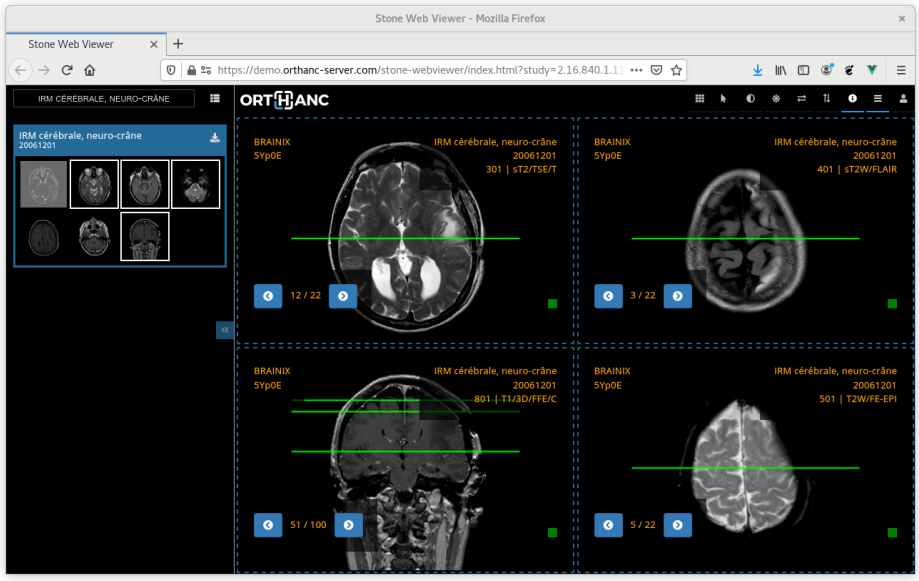

Fig. 3. Screenshot of the Stone Web viewer [14].

ured as a reverse proxy to map the REST API of the DICOMweb server next to the assets of the Stone Web viewer. The configuration file of the Stone Web viewer has an option to provide the root of the DICOMweb API to this end.

However, this manual configuration of a HTTP server might sound complex to users without a background in network administration. To ease the deployments, a plugin for Orthanc has been developed to directly serve the assets of the Stone Web viewer using the embedded HTTP server of Orthanc. Thanks to this Stone Web viewer plugin and thanks to the DICOMweb plugin for Orthanc, it is extremely easy to start the Stone Web viewer. Furthermore, this plugin is readily available in the Docker images, the Microsoft Windows installers and the macOS packages provided by the Orthanc project[5].

Figure 4 provides the big picture of the different components that are involved in the Stone Web viewer. The Stone Web viewer is a high-level application that is built on the top of the Stone of Orthanc library. As can be seen, the Stone of Orthanc library reuses some classes from the Orthanc project that are referred to as the "Orthanc Framework." The Stone Web viewer can either be served from a HTTP server or be run as a plugin to the main Orthanc server.

[5] An online demo of the Stone Web viewer plugin is also available at: https://demo.orthanc-server.com/

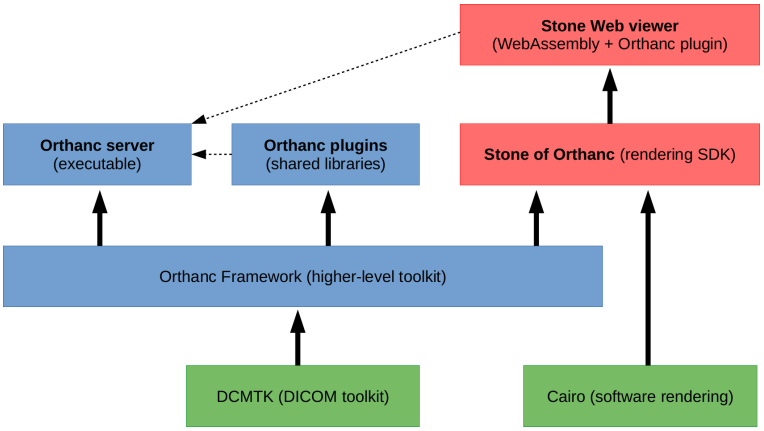

Fig. 4. Overview of the high-level architecture of the Stone Web viewer [14]. The green boxes indicate the major third-party libraries, the blue boxes represent components provided by the Orthanc project [10], and the red boxes correspond to the Stone of Orthanc project.

4 Client-side segmentation using deep learning

Thanks to the flexibility of the Stone of Orthanc library, the Stone Web viewer can be extended in many ways. As explained in Section 2.3, Stone of Orthanc can notably render stacks of superimposed Layers. It is thus possible to overlay a segmentation mask on the top of a medical image. In this section, this capability will be used to apply a deep learning model for lung segmentation to the slice that is currently displayed by the Stone Web viewer, then to display the output of this segmentation. The segmentation mask will be computed locally, entirely inside the Web browser, thanks to WebAssembly.

4.1 U-Net architecture

U-Net is a remarkably successful deep learning architecture for bioimaging segmentation [23]. Contrarily to the sliding-window approach, in which all the possible patches of a fixed size in an image are scanned by a CNN producing one binary output for each patch [3], the U-Net architecture considers the input image as a whole. U-Net models first apply a contraction path to generate high-level features, then apply an expansion path to output the segmentation mask, resulting in the typical "U" shape that is depicted in Figure 5. This architecture allows for the global spatial structure of the medical images to be considered. Generating a segmentation mask given an input image is also much faster than using the sliding-window approach.

In this work, the U-Net models were trained on the individual 2D axial slices of segmented CT-scan images to minimize a smoothed version of the Dice loss

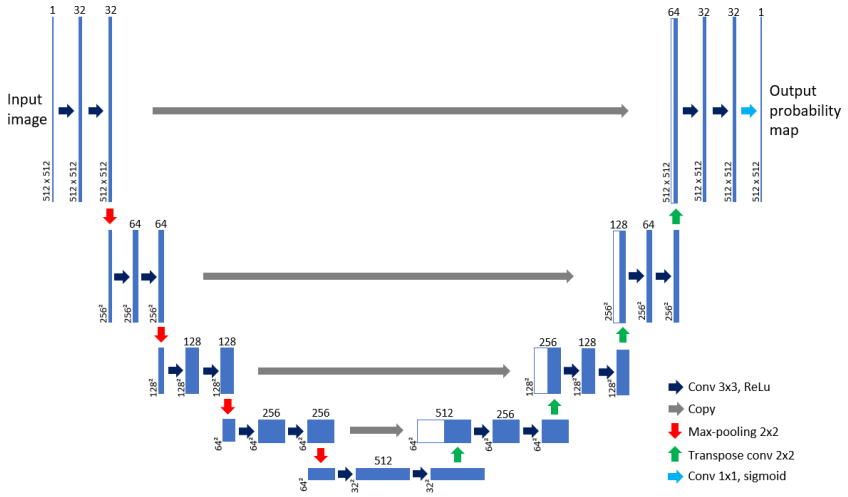

Fig. 5. The U-Net architecture that is used in this paper. The main differences with respect to the original U-Net architecture [23] are the size of the input and output images (both 512×512 pixels), the number of filters used for the convolutions (it ranges from 32 to 512 filters), and the zero-padding that is used in convolutions (which preserves the image sizes throughout the network). The same architecture was used in our recent work related to federated learning for the segmentation of the heart [20]. The resulting U-Net models contain exactly 7,759,521 floating-point parameters.

function [28], that is defined as:

$$\mathcal{L}(\hat{Y}, Y) = -\frac{2\left(\sum_{(x,y)} \hat{Y}(x,y)Y(x,y)\right) + \epsilon}{\left(\sum_{(x,y)} \hat{Y}(x,y)\right) + \left(\sum_{(x,y)} Y(x,y)\right) + \epsilon},$$

where $Y(x,y) \in \{0,1\}$ (resp. $\hat{Y}(x,y)$) denotes the expected binary value in the segmentation mask (resp. the actual prediction of the U-Net) of the pixel at coordinates (x,y), and where ϵ is a coefficient that prevents divisions by zeros as well as vanishing gradients induced by the slices that do not contain any pixel belonging to the segmentation mask.

4.2 Dataset for lung segmentation and data augmentation

A U-Net model for the segmentation of the lungs was trained on the *Lung CT Segmentation Challenge 2017* (LCTSC) dataset [31,32] provided by *The Cancer Imaging Archive* project [4]. This dataset contains DICOM CT-scan images together with the delineation of both lungs encoded as DICOM RT-STRUCT instances. The latter DICOM RT-STRUCT format is routinely used by radiotherapy and nuclear medicine departments to encode the contours of organs

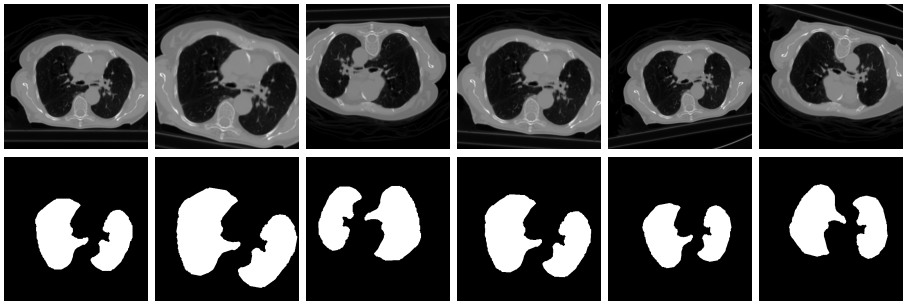

Fig. 6. A illustrative set of training slices (top) together with their segmentation mask of the lungs (bottom), as obtained through data augmentation. The first column shows the original slices from the LCTSC dataset, and the other columns some of their augmented versions. Those images were all generated by REST API calls to the rendering plugin.

and tumors. The LCTSC dataset contains 60 DICOM studies, each study corresponding to one 3D CT-scan, and is divided in 36 studies for the train set and 24 studies for the test set. This defines 3393 training 2D slices and 2063 testing 2D slices. A validation set containing 20% of the train set was retained. This validation set was used for the early stopping of the training to avoid overfitting.

Importantly, data augmentation was used to improve the performance of the model while reducing overfitting [26]: This well-known technique consists in artificially increasing the size of a dataset using random label-preserving transformations, such as rotations, offsets, flips or zooms. An illustration of data augmentation applied to the LCTSC dataset is shown in Figure 6. Most research works dealing with CNN include a preprocessing pipeline that consists in converting the source DICOM images to a set of files in dedicated folders that can easily be opened as NumPy arrays, possibly including data augmentation, before carrying on the actual training of the models in Python.

In this paper, we preferred to use the Orthanc PACS server as the sole source for the training data. The DICOM instances of the LCTSC dataset were rapidly stored and indexed in Orthanc using the folder indexer plugin [13]. Secondly, we developed a so-called "rendering plugin" for the Orthanc server that generates NumPy arrays on-the-fly through REST API requests. The latter requests specify the DICOM identifier of their CT-scan slice or their RT-STRUCT instance of interest, as well as the parameters for possible data augmentation. This plugin was developed on the top of the primitives offered by the Stone of Orthanc library. Thanks to this plugin, the training process can be entirely isolated from the rendering of the augmented training images. This approach facilitates the data management by providing a robust training environment, and it avoids the duplication of the training data on the filesystem. The resulting training pipeline does not contain any preprocessing step.

4.3 Training

The U-Net model for lung segmentation was trained in Python using Keras with the TensorFlow back-end [2]. The Adam optimizer was used, with random initial weights, a learning rate of $2 \cdot 10^{-5}$ and a batch size of 1. Note that this batch size of 1 corresponds to the value that was used in the original U-Net paper [23]. Four dropout layers [27] were introduced at the four max-pooling layers (cf. Figure 5) to prevent overfitting. The smoothing parameter ϵ of the Dice loss function $\mathcal{L}(\hat{Y}, Y)$ was set to 10.

Online data augmentation was applied through Keras data generators calling the REST API of the rendering plugin, which means that the augmented images were different at each epoch. The following transformations for data augmentation were experimentally found to provide good results: rotations between $-10°$ and $10°$, zoom-in and zoom-out up to 5%, displacements up to 5 pixels in both axes, flips along both axes, and slight modifications of the Hounsfield values of the CT-scan up to 1%.

The training took 7 hours on a standard desktop computer (Intel Core i7-9700 CPU equipped with a NVIDIA GeForce GTX 1650 GPU). The resulting Dice score was 95%, which indicates average performance. Note that the objective of this paper was not to train the best-performing U-Net model. Instead, our goal was to train a sensible U-Net model on an average computer, to release open-source code for other people to independently reproduce this result, then to release the U-Net model in open-access to demonstrate its use in the context of WebAssembly. Further development could release better U-Net models in open-access, for instance by taking advantage of recent meta-optimization techniques such as nnU-Net [9].

4.4 U-Net models in the Stone Web viewer

Once the U-Net model for lung segmentation is available, it still must be applied to some slice displayed in the Stone Web viewer. This implies that the model generated by Keras must somehow be executed by WebAssembly, without any access to Python.

To this end, we implemented inside the Stone of Orthanc library, a highly portable sub-library written in plain C++ to apply CNN models. Our CNN toolkit is focused on 2D images, not on generic tensors, which implies that both its footprint and its complexity are small. This library provides a reference implementation for all the distinct types of layers that are encountered in the U-Net architecture (cf. Figure 5). The library is currently single-threaded to match the constraints of WebAssembly, and it only uses the CPU for maximum portability. Future work will improve its performance by leveraging multi-threading and GPU. It is nonetheless worth noticing that convolution layers are already implemented as matrix products, which is a well-known technique to improve the locality of the data, hereby greatly speeding up the convolutions.

Another difficulty lies in the complex file format that is used by Keras to save the parameters of the CNN models. To alleviate this problem, we decided

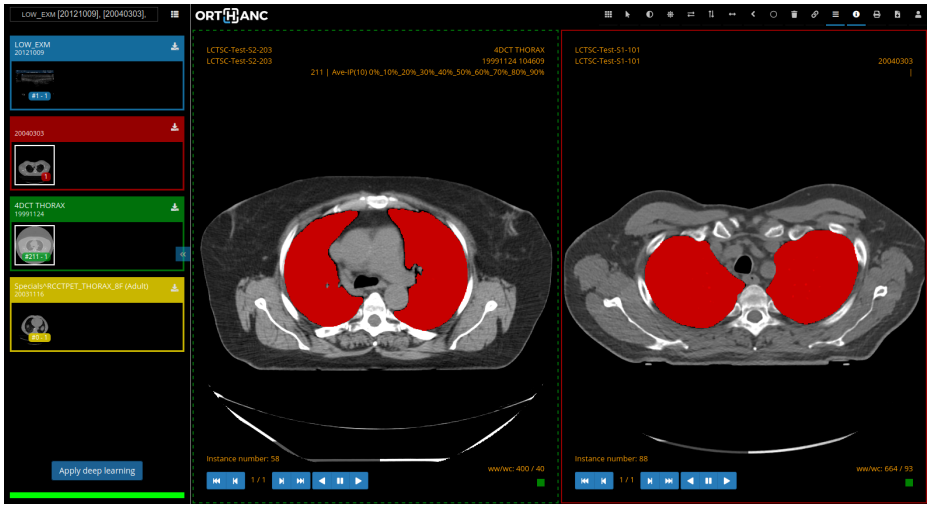

Fig. 7. Segmentation using U-Net models in the Stone Web viewer, applied to two test slices of the LCTSC dataset. The user launches the segmentation by clicking on the "*Apply deep learning*" button at the bottom left of the Web interface. Then, a green progress bar below the button provides feedback about the computation by the Web worker. The segmentation mask is finally rendered as a red, semi-transparent overlay on the top the slice of the CT-scan.

to convert the models to a simpler binary format using Google's Protocol Buffers (Protobuf) library. A conversion script written in Python uses the Keras API to explore the parameters of the various layers in the trained model, then to save them using the Protobuf format. Protobuf being a highly portable library, it can be cross-compiled to WebAssembly. Altogether, this provides a pipeline to load Keras models into the Stone Web viewer.

Combining these different elements, Figure 7 shows an extended version of the Stone Web viewer that supports deep learning for lung segmentation. The CNN toolkit applies the U-Net model in a dedicated Web worker, which amounts to running the segmentation in a separate thread, as Web workers are the official HTML way of executing tasks in the background of a Web application. This use of a Web worker avoids to freeze the user interface while applying the U-Net model. The full deep learning pipeline can be found in Figure 8. On a standard laptop computer equipped with an Intel i7-1165G7 CPU, applying the U-Net model of Figure 5 necessitates 18 seconds, which is only twice longer than the same code executed as a native executable. As indicated above, future work will try and reduce this execution time. The key point here is that U-Net models can be applied through WebAssembly directly in a teleradiology application, without resorting to any dedicated computing cluster or to any cloud infrastructure. The diffusion in open-access of U-Net models usable directly in a free and open-source

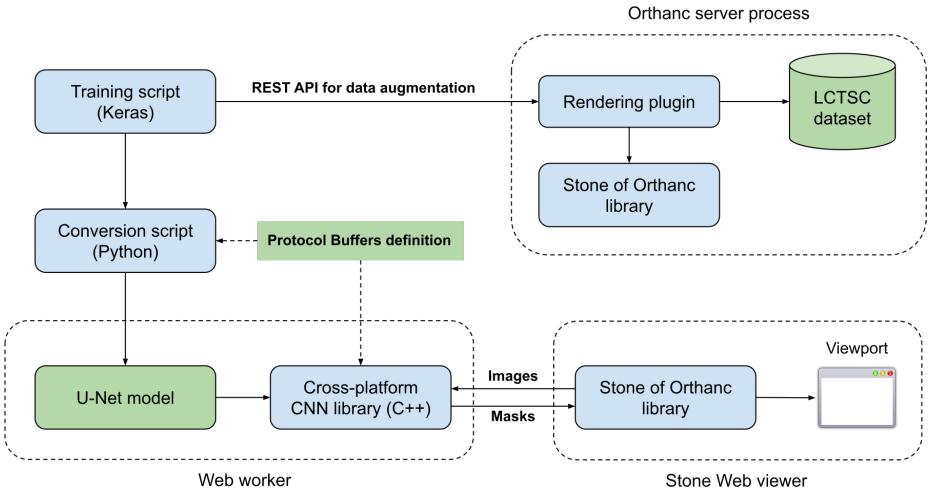

Fig. 8. The full deep learning pipeline for U-Net models. Data files are highlighted in green, whereas the blue boxes indicate the software components. Google's Protocol Buffers are used as an intermediate binary file format between Keras and the Stone Web viewer. The U-Net model is applied in a dedicated Web worker, asynchronously of the Stone Web viewer, because this is a time-consuming operation that would freeze the user interface. Evidently, the training and conversion scripts are executed offline, prior to the use of the U-Net model in the Stone Web viewer.

Web viewer may have an impact on the teaching and research related to artificial intelligence in medical imaging.

5 Other applications

The core features of the Stone of Orthanc library were explained in Section 2. Section 3 introduced the Stone Web viewer as a full teleradiology solution that was extended with deep learning in Section 4. In this section, several other higher-level applications that were built using the Stone of Orthanc library are reviewed.

5.1 Rendering oncology images

In the context of nuclear medicine, radiotherapy and protontherapy, patients rarely have the opportunity to view the images of their own treatment plan from home. This is because the Web portals used in hospitals are mostly focused on the radiology workflow, and thus are typically unable to display advanced information such as contours or dose delivery. Yet, viewing one's own treatment plan might have positive impact on the empowerment of the patient and on the reduction of stress during the treatment.

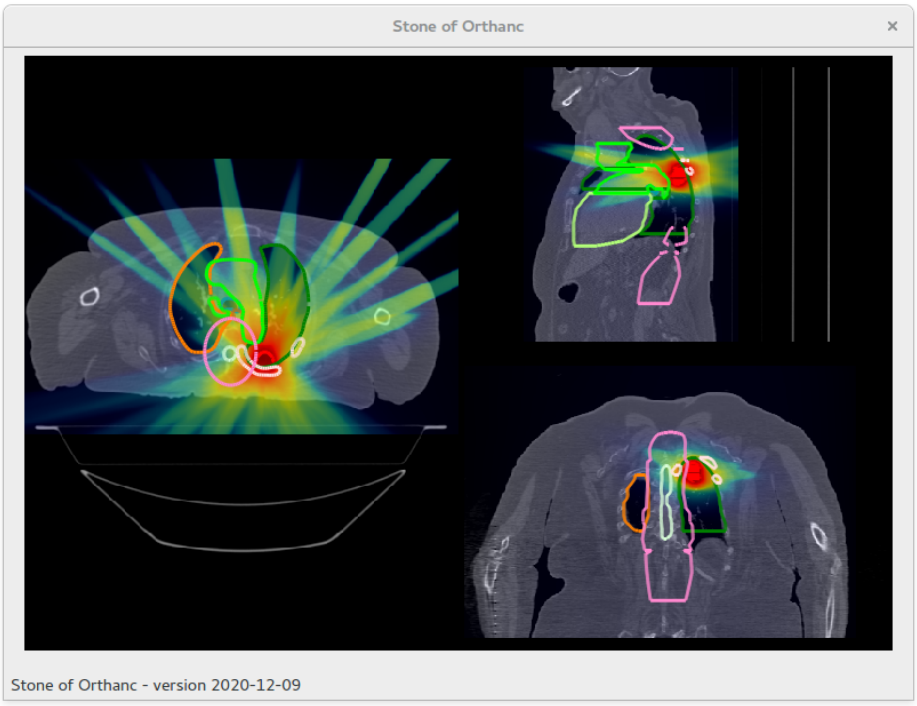

Fig. 9. Desktop application for patient education in the context of external radiotherapy [14,16].

In collaboration with the radiotherapy department of the University Hospital of Liège, we have developed a simple radiotherapy viewer as a cross-platform desktop application built using the Qt toolkit and the Stone of Orthanc library. This viewer was used in a controlled randomized clinical trial to evaluate the feasibility of using a viewer of radiotherapy plans in the context of patient education [16]. In this clinical trial, the patients received a USB key containing their own images and the viewer application, after a personal meeting with their oncologist and with a nurse who explained how to use the viewer.

The user interface of the viewer is depicted in Figure 9. It is designed to be as simple as possible. It displays the fusion of the simulation CT-scan, the planned dose (DICOM RT-DOSE), and the different contours (DICOM RT-STRUCT) following a MPR layout. The user can select which contours are displayed.

Following the terminology of Section 2.3, the user interface of this application is made of three different Scene objects: one for the axial projection, one for the sagittal projection, and one for the coronal projection. In turn, each of these three Scene objects contains three Layer objects: The foremost Layer contains the polygons that correspond to the contours of the delineated structures, the central Layer displays a colorized version of the slice of the dose (with an alpha channel for transparency), and the backward Layer shows the CT-scan slice. The

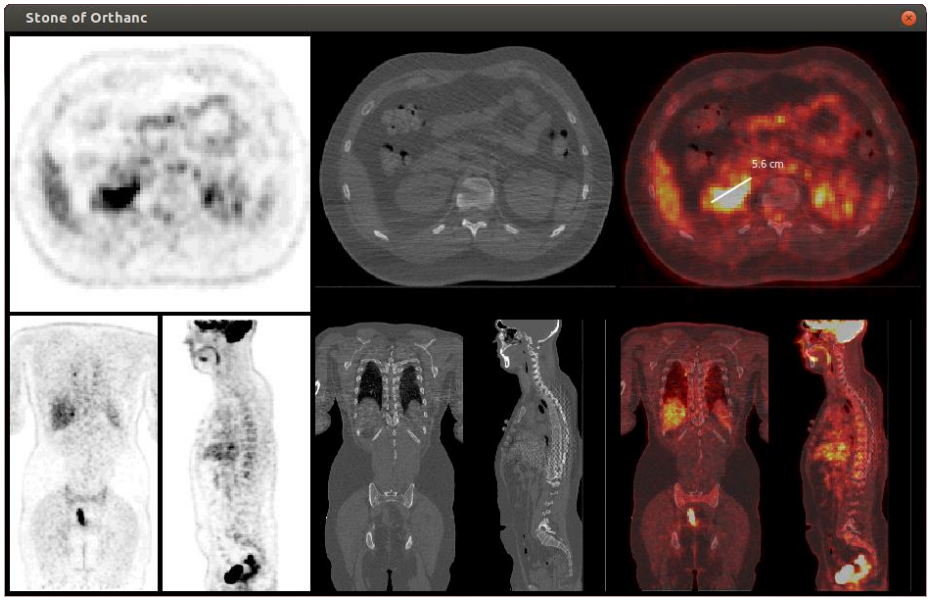

Fig. 10. Desktop application for the rendering of PET-CT-scans according to a clinical-like layout [14].

multimodal fusion is then obtained by blitting the three 2D Layer objects of each Scene onto the corresponding drawing surface.

From an algorithmic point of view, the surface rendering of the 3D contours is done by reading the content of one DICOM RT-STRUCT instance. In such an instance, the contours of each structure are specified as a list of polygons on the individual axial slices. Consequently, the rendering of one specific slice in the axial projection simply consists in drawing the polygons that are contained in this slice of interest. The sagittal and coronal projections are more complex to deal with. We first select the polygons that intersect the given sagittal or coronal slice. Each of those intersections define a 2D box in the plane of interest. The final rendering is done by drawing a set of polygons that corresponds to the union of all those 2D boxes. Well-known, efficient computational geometry algorithms based on the sweep line paradigm are available to compute such unions of rectangles [19,22].

On the other hand, the extraction of an axial, sagittal or coronal slice out of the CT-scan volume stored in RAM whose voxels are indexed by (i, j, k) integers is straightforward: It consists in setting i, j or k to the value of interest, then implementing two nested loops over the two other dimensions. The same algorithm can be applied to the PET-scan volume, if its axes are aligned with those of the CT-scan volume. If the two volumes are not aligned, the PET-scan volume can be resliced, a process that will be described in Section 5.2.

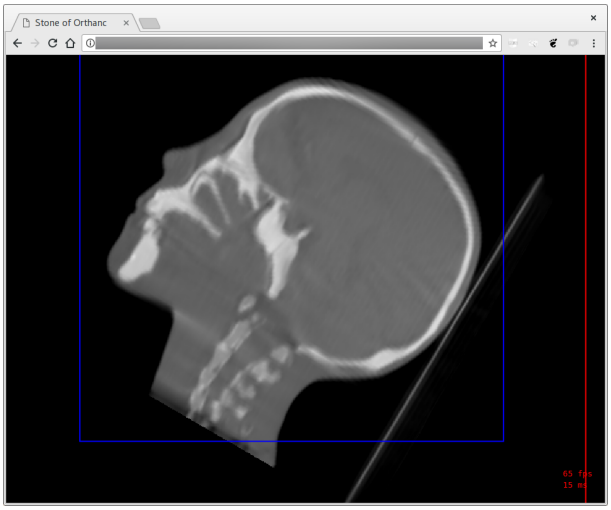

Fig. 11. Real-time reslicing of a 3D volume by a Web browser [14].

Note that besides radiotherapy, the Stone of Orthanc library can be used to render similar oncology images in the field of nuclear medicine. For instance, Figure 10 illustrates another desktop application that provides an advanced rendering of a PET-CT-scan using a layout that is commonly encountered in professional applications. Those two applications could be ported to Web environments thanks to WebAssembly.

5.2 Real-time volume reslicing

As indicated at the end of Section 2.3, Stone of Orthanc does not use the GPU if extracting one slice out of a 3D volume along an arbitrary cutting plane: Volume reslicing is done entirely in RAM by the CPU. This technical choice was motivated by simplicity and portability, but it raises concerns related to performance. To evaluate the performance, a simple WebAssembly application was developed that is depicted in Figure 11. The mouse was used to change the orientation of the cutting plane.

From an algorithmic point of view, the cutting plane is specified by one 3D point indicating the origin of the plane, and by two orthonormal 3D vectors indicating the directions of the X and Y axes in the plane. The intersection of the source volumetric image (which is a rectangular cuboid) with the cutting plane is first computed. If non-empty, this intersection defines a polygon containing between 3 and 6 vertices in the cutting plane [6, Figure 39-6]. Secondly, the bounding box of this polygon in the reference system of the cutting plane defines the physical extent of the 2D image to be generated. The width and height of the extracted bitmap is derived from the desired pixel spacing. Finally, the extracted bitmap is filled by looping over its pixels, looking for the value of the voxel in the

source volume that is the closest to the coordinates of each 2D pixel on the 3D cutting plane. Bilinear or trilinear interpolation can also be applied to improve the visual quality of the reslicing.

On a standard computer (Intel Core i7-3770 at 3.4GHz), the rendering of one slice of size $512 \times 512 \times 502$ voxels in 16bpp takes about 25 milliseconds (40 frames per second), which is fully compatible with real-time requirements. On an old, entry-level smartphone (Samsung Galaxy A3 2017) running the Mozilla Firefox browser, the same rendering takes about 100 milliseconds (10 frames per second), which is still usable in practice.

5.3 Digitally reconstructed radiographs

The Stone of Orthanc library also features some computational geometry primitives to deal with volume rendering. It notably contains C++ classes to generate a digitally reconstructed radiograph (DRR) from a CT volume, which is frequently used in the context of radiotherapy or protontherapy to position the patient in the treatment machine. The DRR is the 2D radiograph that would be imaged using a virtual X-ray camera centered at a given 3D point with a given focal length.

DRR can be computed by raytracing each voxel of the CT volume according to the perspective transform, which is a costly computation because it involves divisions. Stone of Orthanc optimizes this computation by using the so-called "shear-warp transform" [17,18]. The intuitive idea behind shear-warp is to postpone as much as possible the application of the perspective transformation, and to apply it only once to a single "*intermediate image*" (this is the "warp" step). This intermediate image is created from the summation of the individual slices of the volume after the application of an affine transform in which the X and Y axes are left uncoupled, which enables a fast computation (this is the "shear" step that is separately applied to each slice). The shear-warp transform provides an efficient algorithm for CPU-based volume rendering.

Figure 12 shows a WebAssembly application that computes a DRR in near-real-time. On a standard computer, the full rendering of a $512 \times 512 \times 502$ volume in 16bpp takes about 1 second. While moving the virtual camera, the number of slices used for the rendering can be reduced to improve speed, thus enhancing the user experience. The GPU could be used to speed up this computation by implementing an appropriate Oracle command. Besides DRR, the shear-warp algorithm could also be applied to compute MIP rendering by putting the virtual camera at infinity, and by replacing the summation with a "max" operation in the intermediate image.

6 Conclusions

This paper introduces Stone of Orthanc as a library to render medical images, in a way that is compatible with desktop, mobile and Web applications. Stone of

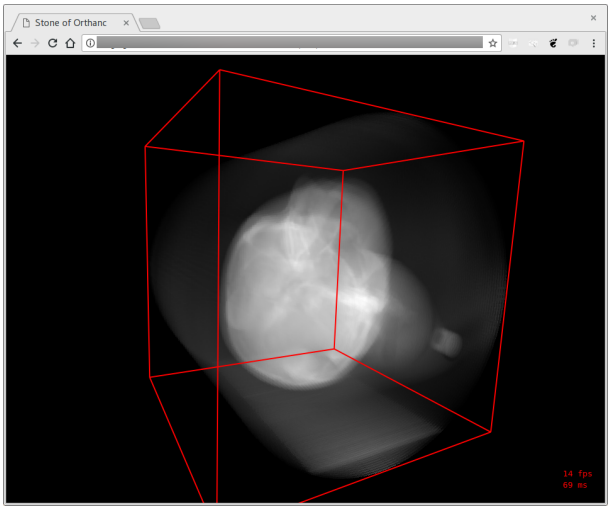

Fig. 12. Web rendering of a digitally reconstructed radiograph (DRR) from a CT-scan [14]. The projection is computed client-side by the Web browser in near-real-time (a few frames per second).

Orthanc is fully written in C++, and leverages the emerging WebAssembly standard if targeting Web applications. Sample applications of Stone of Orthanc are described as well. In particular, the Stone Web viewer is introduced as the first free and open-source teleradiology solution entirely developed using WebAssembly. As shown in this paper, WebAssembly can be used to maximize code reuse in medical imaging, which should also be true in many other scientific fields that involve numerical computations.

This paper also demonstrates how U-Net models can be executed client-side by Web browsers to segment images in a teleradiology solution, without running the computation on a dedicated server, and without having to transfer the medical images to a remote cloud infrastructure. This contribution could notably help the teaching of artificial intelligence for medical imaging by sharing technical knowledge about deep learning to a much larger audience, as it avoids the need to resort to proprietary solutions that are not available outside of large-scale clinical environments.

The source code of the Stone of Orthanc library is publicly released under the LGPL license, and the Stone Web viewer is licensed under the AGPL. The whole Stone of Orthanc project currently contains more than 70,000 lines of code[6].

Future work will pursue the exploitation of Stone of Orthanc for artificial intelligence applications in medical imaging. We notably plan to design applications supporting the "AI Results (AIR)" IHE profile that is designed to standardize how the inputs and outputs of artificial intelligence algorithms are exchanged.

[6] The source code is located at: https://hg.orthanc-server.com/orthanc-stone/

The integration of MPR layout, MIP rendering and whole-slide imaging [12] directly within the Stone Web viewer will also be investigated. Finally, as far as client-side deep learning is concerned, we plan to improve the performance of our deep learning toolkit, to establish an open-access library of traceable U-Net models, and to extend the Stone Web viewer to other types of deep learning networks beyond image segmentation (such as image classification and object detection).

Acknowledgments The author wants to thank Benjamin Golinvaux, Alain Mazy and Osimis for their contributions to the development of the Stone of Orthanc project.

References

1. Cheng, P.M., Montagnon, E., Yamashita, R., Pan, I., Cadrin-Chênevert, A., Perdigón Romero, F., Chartrand, G., Kadoury, S., Tang, A.: Deep learning: An update for radiologists. RadioGraphics **41**(5), 1427–1445 (2021). https://doi.org/10.1148/rg.2021200210
2. Chollet, F., et al.: Keras (2015), https://github.com/fchollet/keras
3. Ciresan, D., Giusti, A., Gambardella, L., Schmidhuber, J.: Deep neural networks segment neuronal membranes in electron microscopy images. In: Pereira, F., Burges, C.J.C., Bottou, L., Weinberger, K.Q. (eds.) Advances in Neural Information Processing Systems. vol. 25. Curran Associates, Inc. (2012), http://papers.nips.cc/paper/by-source-2012-1292
4. Clark, K., Vendt, B., Smith, K., Freymann, J., Kirby, J., Koppel, P., Moore, S., Phillips, S., Maffitt, D., Pringle, M., Tarbox, L., Prior, F.: The Cancer Imaging Archive (TCIA): Maintaining and operating a public information repository. Journal of Digital Imaging **26**(6), 1045–1057 (Jul 2013). https://doi.org/10.1007/s10278-013-9622-7
5. Costa, C., Ferreira, C., Bastião, L., Ribeiro, L., Silva, A., Oliveira, J.L.: Dicoogle – an open source peer-to-peer PACS. Journal of Digital Imaging **24**(5), 848–856 (Oct 2011). https://doi.org/10.1007/s10278-010-9347-9
6. Fernando, R.: GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics. Pearson Higher Education (2004)
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1st edn. (1994)
8. Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J.F.: Bringing the Web up to speed with WebAssembly. In: Cohen, A., Vechev, M.T. (eds.) Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017. pp. 185–200. ACM (2017). https://doi.org/10.1145/3062341.3062363
9. Isensee, F., Jaeger, P.F., Kohl, S.A.A., Petersen, J., Maier-Hein, K.H.: nnU-Net: A self-configuring method for deep learning-based biomedical image segmentation. Nature Methods **18**(2), 203–211 (Feb 2021). https://doi.org/10.1038/s41592-020-01008-z
10. Jodogne, S.: The Orthanc ecosystem for medical imaging. Journal of Digital Imaging **31**(3), 341–352 (Jun 2018). https://doi.org/10.1007/s10278-018-0082-y

11. Jodogne, S., Bernard, C., Devillers, M., Lenaerts, E., Coucke, P.: Orthanc – A lightweight, RESTful DICOM server for healthcare and medical research. Proceedings, IEEE International Symposium on Biomedical Imaging: from Nano to Macro pp. 190–193 (2013). https://doi.org/10.1109/ISBI.2013.6556444
12. Jodogne, S., Lenaerts, E., Marquet, L., Erpicum, C., Greimers, R., Gillet, P., Hustinx, R., Delvenne, P.: Open implementation of DICOM for whole-slide microscopic imaging. In: VISIGRAPP (6: VISAPP). pp. 81–87 (2017). https://doi.org/10.5220/0006155100810087
13. Jodogne, S.: Importing and serving open-data medical images to support artificial intelligence research. In: EuSoMII Annual Meeting. vol. 13(S1). SpringerOpen (2021). https://doi.org/10.1186/s13244-022-01168-w
14. Jodogne, S.: Rendering medical images using WebAssembly. In: Proc. of the 15th International Joint Conference on Biomedical Engineering Systems and Technologies (Volume 2). pp. 43–51 (2022). https://doi.org/10.5220/0000156300003123
15. Kikinis, R., Pieper, S.D., Vosburgh, K.G.: 3D Slicer: A Platform for Subject-Specific Image Analysis, Visualization, and Clinical Support, pp. 277–289. Springer New York, New York, NY (2014). https://doi.org/10.1007/978-1-4614-7657-3_19
16. Kirkove, D., Barthelemy, N., Coucke, P., Mievis, C., Ben Mustapha, S., Jodogne, S., Dardenne, N., Donneau, A., Pétré, B.: Étude de faisabilité : utilisation de l'imagerie médicale en éducation thérapeutique en radiothérapie. Cancer/Radiothérapie (2022). https://doi.org/10.1016/j.canrad.2022.04.004
17. Lacroute, P.: Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation. Ph.D. thesis, Stanford University (1995), technical Report CSL-TR-95-678
18. Lacroute, P., Levoy, M.: Fast volume rendering using a shear-warp factorization of the viewing transformation. In: Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques. pp. 451–458. SIGGRAPH '94, ACM, New York, NY, USA (1994). https://doi.org/10.1145/192161.192283
19. Lipski, W., Preparata, F.P.: Finding the contour of a union of iso-oriented rectangles. Journal of Algorithms **1**(3), 235–246 (1980). https://doi.org/10.1016/0196-6774(80)90011-5
20. Misonne, T., Jodogne, S.: Federated learning for heart segmentation. In: 2022 IEEE 14th Image, Video, and Multidimensional Signal Processing Workshop (IVMSP). pp. 1–5 (2022). https://doi.org/10.1109/IVMSP54334.2022.9816345
21. National Electrical Manufacturers Association: ISO 12052 / NEMA PS3, Digital Imaging and Communications in Medicine (DICOM) standard (2021), http://www.dicomstandard.org
22. Preparata, F.P., Shamos, M.I.: Computational Geometry: An Introduction. Texts and Monographs in Computer Science, Springer (1985). https://doi.org/10.1007/978-1-4612-1098-6
23. Ronneberger, O., Fischer, P., Brox, T.: U-Net: Convolutional networks for biomedical image segmentation. In: Navab, N., Hornegger, J., III, W.M.W., Frangi, A.F. (eds.) Medical Image Computing and Computer-Assisted Intervention - MICCAI 2015 - 18th International Conference Munich, Germany, October 5 - 9, 2015, Proceedings, Part III. Lecture Notes in Computer Science, vol. 9351, pp. 234–241. Springer (2015). https://doi.org/10.1007/978-3-319-24574-4_28
24. Rosset, A., Spadola, L., Ratib, O.: OsiriX: An open-source software for navigating in multidimensional DICOM images. Journal of Digital Imaging **17**(3), 205–216 (Sep 2004). https://doi.org/10.1007/s10278-004-1014-6
25. Schroeder, W., Martin, K., Lorensen, B.: The Visualization Toolkit: An object-oriented approach to 3D graphics. Kitware, Inc., 4 edn. (2006)

26. Shorten, C., Khoshgoftaar, T.M.: A survey on image data augmentation for deep learning. Journal of Big Data **6**(1) (Jul 2019). https://doi.org/10.1186/s40537-019-0197-0
27. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: A simple way to prevent neural networks from overfitting. Journal of Machine Learning Research **15**(56), 1929–1958 (2014)
28. Sudre, C.H., Li, W., Vercauteren, T., Ourselin, S., Cardoso, M.J.: Generalised Dice overlap as a deep learning loss function for highly unbalanced segmentations. In: Deep Learning in Medical Image Analysis and Multimodal Learning for Clinical Decision Support, pp. 240–248. Springer International Publishing (2017). https://doi.org/10.1007/978-3-319-67558-9_28
29. Urban, T., Ziegler, E., Lewis, R., Hafey, C., Sadow, C., Van den Abbeele, A.D., Harris, G.J.: LesionTracker: Extensible open-source zero-footprint web viewer for cancer imaging research and clinical trials. Cancer Research **77**(21), e119–e122 (2017). https://doi.org/10.1158/0008-5472.CAN-17-0334
30. Warnock, M.J., Toland, C., Evans, D., Wallace, B., Nagy, P.: Benefits of using the DCM4CHE DICOM archive. Journal of Digital Imaging **20**(Supp. 1), 125–129 (2007). https://doi.org/10.1007/s10278-007-9064-1
31. Yang, J., Sharp, G., Veeraraghavan, H., Van Elmpt, W., Dekker, A., Lustberg, T., Gooding, M.: Data from lung CT segmentation challenge (2017). https://doi.org/10.7937/K9/TCIA.2017.3R3FVZ08
32. Yang, J., Veeraraghavan, H., Armato, S.G., Farahani, K., Kirby, J.S., Kalpathy-Kramer, J., van Elmpt, W., Dekker, A., Han, X., Feng, X., Aljabar, P., Oliveira, B., van der Heyden, B., Zamdborg, L., Lam, D., Gooding, M., Sharp, G.C.: Autosegmentation for thoracic radiation treatment planning: A grand challenge at AAPM 2017. Medical Physics **45**(10), 4568–4581 (Sep 2018). https://doi.org/10.1002/mp.13141