Tuoris: A middleware for visualizing dynamic graphics in scalable resolution display environments

Víctor Martínez[a,b,1], Senaka Fernando[b,1], Miguel Molina-Solana[b,a,*], Yike Guo[b]

[a]*Dept. Computer Science and AI, Universidad de Granada, Spain*
[b]*Data Science Institute, Imperial College London, United Kingdom*

Abstract

In the era of big data, large-scale information visualization has become an important challenge. Scalable resolution display environments (SRDEs) have emerged as a technological solution for building high-resolution display systems by tiling lower resolution screens. These systems bring serious advantages, including lower construction cost and better maintainability compared to other alternatives. However, they require specialized software but also purpose-built content to suit the inherently complex underlying systems. This creates several challenges when designing visualizations for big data, such that can be reused across several SRDEs of varying dimensions. This is not yet a common practice but is becoming increasingly popular among those who engage in collaborative visual analytics in data observatories. In this paper, we define three key requirements for systems suitable for such environments, point out limitations of existing frameworks, and introduce Tuoris, a novel open-source middleware for visualizing dynamic graphics in SRDEs. Tuoris manages the complexity of distributing and synchronizing the information among different components of the system, eliminating the need for purpose-built content. This makes it possible for users to seamlessly port existing graphical content developed using standard web technologies, and simplifies the process of developing advanced, dynamic and interactive web applications for large-scale information visualization. Tuoris is designed to work with Scalable Vector Graphics (SVG), reducing bandwidth consumption and achieving high frame rates in visualizations with dynamic animations. It scales independent of the display wall resolution and contrasts with other frameworks that transmit visual information as blocks of images.

Keywords: distributed visualization, large-scale visualization, SVG

1. Introduction

Data visualization is a fundamental step in most modern data analytics workflows, and is often used more than once in these processes – during the initial exploration as well as a mechanism to present and communicate results. In fact, the term *visual analytics* was coined to describe the process in which additional and relevant insights may be gained by representing data in a pictorial way [1]. The ever increasing demand for big data, which is the driving force behind a number of novel technologies, presents not only unique research and business opportunities, but also significant challenges and problems [2, 3]. One key challenge concerns the interpretation and understanding of large and diverse datasets, which require expertise from multiple domains. Data science researchers encourage *collaborative visual analytics* as a resolution to this challenge [4, 5].

Immersive data observatories [6, 7, 8] and scalable display walls [9, 10, 11] are proven to be excellent platforms for collaborative visual analytics. Such systems have been in existence for nearly three decades [12] and have seen tremendous commercial success offering a dramatic increase in the ability to generate valuable insight from big data. Despite computational power, storage capacity, and network bandwidth improving in orders of magnitude over the past decades, display devices have seen a much slower evolution and large display environments remain costly to build and hard to maintain [13]. Building large high-resolution displays by tiling smaller and cheaper screens has therefore become a popular choice, and we find a variety of such *scalable resolution display environments* (SRDEs) [14, 15] in existence.

The Data Science Institute of Imperial College London has several such SRDEs which are capable of accommodating audiences of various sizes: (a) an immersive data observatory in a cylindrical layout configuration of 64 HD screens providing a 132.7 megapixel display which can accommodate 20 individuals, (b) a 2D display wall made from 8 4K screens providing a 66.35 megapixel display which can accommodate 6–8 individuals, and (c) a 3D display wall made from 6 glassless 3D screens providing a 12.44 megapixel display which can accommodate 3–5 individuals. This makes it possible for multidisciplinary teams to explore big data in various settings.

One of the key challenges in operating SRDEs is the inability to use general purpose software in such environments as they do not support high performance rendering and high resolution output, restricting us to a select few specially designed middleware such as SAGE2 [16, 17], DisplayCluster [18], CGLX [19]

[*]Corresponding author

Email addresses: `victormg@acm.com` (Víctor Martínez), `senaka.fernando15@imperial.ac.uk` (Senaka Fernando), `mmolinas@ic.ac.uk` (Miguel Molina-Solana), `y.guo@imperial.ac.uk` (Yike Guo)

[1]These authors contributed equally to this work.

and Equalizer [20, 21]. Despite the individual merits of these frameworks we find major limitations in performance, scalability, and transparency preventing us adopting them across multiple purpose-built SRDEs. These limitations are common to any institution planning to operate more than one SRDE, or for a group of institutions that plan to engage in collective decision making on big data. We therefore define three requirements for systems suitable for collaborative visual analytics across a range of several SRDEs:

Performance of the system regardless of scale and complexity of the visualization, achieving a high frame rate suitable for producing smooth and satisfactory animated graphics for the human eye. The system must operate within a reasonable and predictable CPU and memory footprint and be suitable for general purpose networking infrastructure.

Scalability of a visualization such that it would properly render at any resolution making it possible for an audience to view the same content in a single display, across a few screens driven by a single computer, or a large high-resolution display of an SRDE. This scalability must not lead to degraded performance.

Transparency or the ability to design a visualization once and use it across environments of various scales without any reduction of quality, modifications to its contents and code, or having to introduce special adaptations for them to be usable at higher resolutions.

In addition to meeting the requirements mentioned above, other aspects such as *reliability* (does it crash frequently or not), *maintainability* (how easy is it to maintain and evolve), and *usability* (how easy is it to control interactive applications at runtime), must also be considered when designing middleware for SRDEs.

This manuscript introduces Tuoris, a novel open-source middleware for visualizing dynamic graphics in SRDEs. Tuoris is capable of distributing graphics in an SVG format while synchronizing predefined animations and interactive operations across tiled displays of variable resolutions. It is capable of driving an SRDE using commodity hardware and aims to offer a framework for collaborative visual analytics while meeting the performance, scalability and transparency requirements defined earlier.

In this paper, we start off by looking at the evolution of large high-resolution displays and high performance rendering pipelines and review contemporary middleware for SRDEs pointing out their limitations which prevent their adoption for our collaborative visual analytics use-cases. We then describe the design decisions underpinning Tuoris and explain its architecture in Section 3. This is followed by Section 4 where we discuss different use-cases along with a thorough performance analysis conducted within the SRDEs at the Data Science Institute of Imperial College London. Our manuscript ends with a conclusion and a list of further lines of action.

2. Related Work

The idea of using high resolution images to explore large datasets and also the use of high performance rendering pipelines to prepare such images has been a topic of interest in the computer graphics community for many years. Whitman [22] explained the history of attempts to parallel rendering spanning from the late 1970s. Molnar's classification of parallel rendering [23] in the mid 1990s paved the way for many interesting approaches to dynamic rendering of high-resolution graphics. However, the visualization of such high resolution graphics in tiled display walls was not popular until recently. Early examples such as Chromium [24], SAGE [10] and Parallel-SG [25] were introduced some 15 years ago. Modern SRDEs are an evolution of these early examples, with various infrastructures proposed in the past 5–10 years. These systems are diverse but solve similar problems using different strategies, and are not always suitable for each and every problem requiring an SRDE.

The standardization of middleware for SRDEs is still a pending task, but several authors have made attempts to categorize them in different ways. Chen et al. [26, 27] divided them into two groups (*master-slave* and *client-server*) based on their execution model. Ni et al. [14] classified them based on their data distribution architectures (as *display data streaming* software and *distributed rendering* software). Chung et al. [28] divided them into four groups based on applications they target: (a) transparent frameworks (for legacy applications), (b) distributed scene graph (DSG) frameworks (for 3D graphics applications), (c) interactive application frameworks, and (d) scalable rendering frameworks. A final grouping, noted by Renambot et al. [17], is according to their deployment models (*browser-based* vs. *desktop-application*).

Given these various classifications of middleware for SRDEs, we started off by looking at some window management systems. These systems intend to provide a unified workspace for distributed data visualization, making it possible to run different applications at the same time across several tiled displays. *Distributed Multihead X* (DMX[2]) was designed to provide multi-head support for the X-Windows desktop in systems composed of multiple displays. DMX follows a client-server model, where the server node distributes the visual elements, which are later rendered in client nodes.

Another popular window management architecture is *Scalable Adaptive Graphics Environment* (SAGE) [29] of which a second version SAGE2 [16, 17] is also available and now known as *Scalable Amplified Group Environment*. SAGE2 is a browser-based client-server cross-platform middleware distributed with different useful applications, which was designed to provide a powerful solution for remote collaboration in data intensive environments. These two systems have achieved significant popularity and have a large number of deployments around the world.

Though not as popular as SAGE2, DisplayCluster [18] is another dynamic windowing environment with built-in capabili-

[2]`http://dmx.sourceforge.net/`

ties for viewing media, which includes the display and streaming of ultra high-resolution images and video content. Despite these being transparent frameworks featuring a non-invasive programming model, they heavily rely on pixel streaming to distribute most visual information from server to clients. This greatly reduces the scalability of such systems in terms of supported resolution and restrict their applicability to environments where high-performance networks are available.

Unlike window management approaches, parallel graphics rendering middleware such as the *Image Composition Engine for Tiles* (IceT) [30, 31] distribute the rendering workload across multiple nodes of a distributed system. This allows applications to perform sort-last parallel rendering by splitting the display area into tiles and assigning them to one or more processors. And, at the same time, each processor can also be rendering content for more than one tile. IceT distributes the rendering of graphical content for M tiles across N processors and rely on multiple strategies such as binary trees and map-reduce to compose the resulting tiles and generate a unique image. While IceT met both the performance and scalability requirements we anticipated, it requires visualizations to be purpose-built to suit its rendering pipelines, limiting its scope.

A different approach is found in Chromium [24], a framework based on the older WireGL [32] system from Stanford. It operates by manipulating and distributing streams of graphic API commands on clusters of computers. Chromium can run OpenGL-based applications while intercepting OpenGL commands, which are distributed to client nodes. These clients execute the commands they receive in order to render their corresponding portion of a much larger scene. The main limitation of Chromium is that it incurs on a high network usage —even when the scene does not change— which is a result of its low-level focus giving priority to precision over performance.

Garuda [33] was proposed to solve limitations of Chromium. It reduces network bandwidth consumption by caching and managing the transmitted geometry at rendering nodes. Only required parts of the scene are transmitted to each client by using an adaptive algorithm that culls the scene graph to a hierarchy of frustums to determine which objects are visible in each tile of the wall. While Garuda closely matched many of our needs, it has performance implications such as the need for a high-end server machine and a gigabit ethernet in order to provide a stable frame rate for animations. But, the biggest weakness of Garuda was that it could only support applications of a scene graph type, which makes it less popular among those who engage in collaborative visual analytics.

Equalizer [20] is a toolkit for scalable parallel rendering based on OpenGL. It provides an application programming interface (API) to develop scalable graphics applications, that can run in different configurations including tiled displays. The *Cross-Platform Cluster Graphic Library* (CGLX) [19] provides another API that allows to run the same copies of an OpenGL-based application on all clients and replicate visualization data on all the clients. CGLX was also developed as an enhancement to Chromium, but had a much broader scope compared to Garuda and was not restricted to scene graph type applications, similar to Equalizer. But, unlike Equalizer it was much easier to maintain and was better in terms of transparency. The main weakness of CGLX was its scalability: we found that its performance (measured in FPS, common for animated content) reduced as the number of display tiles increased, due to the synchronization overhead introduced by the head node. This means that animations would run much faster on displays with fewer screens compared to displays with many screens.

Having carefully surveyed alternatives, we found that 1) frameworks supporting dynamic windowing environments were dependent on pixel streaming and performed poorly, 2) frameworks that were focusing on DSG had a limited scope and corresponding transparency implications, and 3) others had scalability limitations which prevented their use on SRDEs of different dimensions. For these reasons, we developed Tuoris following a different approach to address these limitations, but borrowed many key concepts from existing frameworks, which greatly influenced our design.

3. Implementation of Tuoris

Tuoris is an open-source middleware designed to visualize highly dynamic graphics across multiple SRDEs using commodity hardware. Its objective is to support multidisciplinary teams engage in collaborative visual analytics on big data. All design decisions underpinning its software architecture and implementation are driven by these two core aspects.

3.1. Design decisions and limitations

We chose to develop Tuoris as a **web-based** middleware instead of an OpenGL-based desktop application. We were inspired by the choice of a web-based architecture in SAGE2, in contrast to their predecessor SAGE and also by the growing popularity of web technology in general and particularly its widespread adoption in industry and academia for data visualization and visual analytics. This would make it possible for us to cater for a much larger audience intending to use SRDEs for collaborative visual analytics. Furthermore, a deployment of Tuoris involves a minimum installation overhead as it runs on a web browser. It also does not require specific communication and rendering libraries as it relies on a rich ecosystem of features provided by web browsers such as the support for HTML, JavaScript and WebSocket standards.

Another important choice was to use the **SVG file format**, which makes the system scalable. It not only provides the ability to render both vector and raster image formats but also interactive and dynamic drawings which can include animations that are declaratively defined and triggered or scripted using JavaScript. SVG (which is based on XML and an open standard by W3C since 1999) is the most widely adopted mechanism for describing vector graphics in modern web browsers. SVG drawings can be efficiently rendered and distributed across display environments spanning many screens with minimum impact on underlying networks as the specification itself is resolution independent. This enables displaying very high quality graphics in tiled displays spanning multiple screens without aliasing problems. An additional advantage of supporting the

SVG specification is that there are a large number of browser-based software frameworks that can be used to develop SVG-based visualizations, D3 [34] being one of the most popular ones. On top of that, software such as Convertio[3] can be used to convert files from over 100 different formats into the SVG format. This makes it possible for us to use Tuoris to render a large number of documents of different types at variable resolutions, or to embed files of various types within a visualization designed for an SRDE. However, SVG has its limitations, and there are a number of browser based content formats that Tuoris does not support such as playing audio and video or rendering 3D visualizations that are GPU-intensive.

We also opted for **retained rendering** [28] to increase the performance. With this mode of distributed rendering, the display nodes of Tuoris cache a copy of the visual elements making it possible to optimize network bandwidth consumption in subsequent frames by having the server only forward updates to the clients. The alternative to this is *immediate rendering* where the server passes all visual elements on a per-frame basis. All pixel streaming based middleware (including SAGE2) does not support retained mode rendering. Parallel rendering techniques are used to preserve the scalability of these systems, but this still introduces significant network overheads when working with high resolution graphics. Our choice of the SVG file format makes it possible to support retained rendering.

The implementation of Tuoris follows a **client-server architecture**, in which a client initiates a request that is processed by a server. The server then responds back to the requestor or all clients in general. This contrasts with the *master-slave* approach, in which the application is either replicated or broken down into pre-defined segments and distributed among the slaves. The master node drives the system and takes charge of ensuring each slave executes its code at a precise time. As pointed out by Ni et al. [14], the master-slave approach requires lesser network bandwidth and therefore performs better. But, we discarded the master-slave architecture as it presented us with several limitations in terms of synchronizing state and timing animations across multiple screens. Secondly, as pointed out by Nirnimesh et al. [33], our chosen client-server architecture does not require replicating all information on each and every client. This is particularly useful for scalability on SRDEs.

The next important decision was to focus on a **non-invasive programming model** similar to most transparent frameworks and DSG frameworks. As a fully transparent middleware, Tuoris only requires either a URL of an SVG drawing or a valid HTML document with an SVG element containing the visualization; and does not require any modification to existing SVG-based web applications.

One of the key design decisions of Tuoris is also one key limitation. The **single-controller multi-viewer** deployment model of Tuoris is both scalable and also very easy to deploy and manage. The limitation is that all of those who would want to interact with Tuoris would have to share a single user interface such as a webpage rendered on a *tablet*. Compared to other contemporary middleware for SRDEs such as SAGE2, which allow collaboration even across multiple sites, Tuoris cannot be controlled by more than one user interface at a time. However, this does not prevent multiple people interacting through a single control UI, which is achieved by large multi-touch screens. This key decision made it possible for us to greatly simplify the design and deliver real-time interactivity for fairly complex drawings rendered at a very high resolution.

In summary, Tuoris is a browser-based client-server cross-platform middleware like SAGE2 but specifically designed for working with graphics defined in an SVG format. It is similar to DSG frameworks (such as Garuda) which are restricted to scene graph type applications. This limits the scope of our middleware compared to that of SAGE2, but completely avoids need for pixel streaming and instead uses a specialized distributed-rendering algorithm, which we describe later. Tuoris works at an element-level, distributing differences to an SVG canvas as a series of messages similar to frameworks such as Chromium and CGLX. And, it supports retained rendering and distributes messages using a multicast-model while culling them to suit each tile as in the case of Garuda. This makes Tuoris scale to suit an SRDE of any dimension and perform independent of the display resolution, while the non-invasive programming model ensures transparency. One of the main implications of our design is that Tuoris is distinctively different to existing frameworks from a technological, algorithmic and application standpoint. Therefore, to confirm its suitability, we provide an extensive list of examples and testcases in Section 4. We also compare the performance, scalability and transparency of Tuoris with SAGE2 and discuss how these two frameworks meet other specific requirements such as reliability, maintainability and usability.

3.2. System architecture

Tuoris is made up of three loosely coupled components (a server, a control client, and a set of display clients) designed to be hardware and operating system platform independent. The server component requires a *Node.js* runtime environment which can be installed on a physical server, virtual machine or Docker environment. The display clients only require a modern web browser. The choice of client-side as well as server-side JavaScript [35] technology was based on their extensive support for event-driven architectures and asynchronous input and output capabilities. Together with the adoption of the SVG file format, these language-specific features contribute to the scalability of Tuoris.

Providing Tuoris with a URL of an SVG drawing (or alternatively an HTML document in which an SVG element is defined) triggers its server-side processing pipeline. This starts by creating a new HTML document which includes content from the original SVG file and injects a number of JavaScript elements to monitor changes on the underlying SVG element. These scripts adopt the *MutationObserver* specification[4], which is an interface provided by modern web browsers to keep track of

[3]`https://convertio.co/`

[4]`https://dom.spec.whatwg.org/#interface-mutationobserver`

mutations to a tree of Document Object Model (DOM) nodes. Tuoris observes changes to children and attributes of the entire HTML document, but ignores everything that is not within an SVG element.

There are two types of execution modes supported by Tuoris (see Figure 1):

Interactive mode: where Tuoris exposes a *control* webpage which contains the new HTML document. This webpage can be rendered on a separate web browser which the users would interact with. Any mutations to the SVG elements that happen as a result of these interactions will be tracked and broadcasted as a series of events to the server.

Non-interactive mode: where Tuoris renders the new HTML document using a headless Chromium web browser[5], which plays the role of the client. All changes that happen to the SVG element as a result of any existing scripts (such as pre-set animations) would be tracked by this client and broadcasted as a series of events to the server.

We collectively refer to the headless browser and the control webpage as the *control client*. In both execution modes the original SVG is rendered on this client. In the interactive mode the dimensions of this SVG would be suitable for a human viewer and scale to fit the browser on which it runs. In the non-interactive mode this would match the pre-defined internal dimensions (set to the minimum) of the headless browser. The dimensions of the SVG rendered on the control client are fixed regardless of the resolution of the display clients. This makes the performance of the rendering step independent of the SRDE in use. This is an advantage of using SVG, and it also makes our control client highly responsive and deterministic in terms of performance (as the browser spends a fixed and minimum amount of effort in terms of the rendering and painting of the interactive visualization).

The communication between the server and control client uses the *WebSocket* protocol, a standard for establishing a duplex communication channel over TCP [36]. The communication between the server and display clients also use the same protocol, which we implemented using the *socket.io* library. Similar to other browser based visualization frameworks for SRDEs such as SAGE2, Tuoris leverages the WebSocket protocol to provide an efficient communication mechanism between web servers and display clients while introducing a minimum connection overhead. But unlike most other frameworks Tuoris reduces both the size and frequency of the messages exchanged between each of these components, by means of a specially designed algorithm:

1. The control client tracks all mutations to the SVG DOM tree and stores them in an array as and when they happen, but process them at the rate at which a display refreshes (usually *60 FPS*).

[5]`https://pptr.dev/`

2. All mutations are recorded along with an identifier corresponding to the node that changed, which makes it possible to correlate each of them with that specific node.

3. Mutation processing stops either when there are no more mutations to process or when `30ms` has elapsed (`30ms` duration gives us sufficient time to process changes and broadcast them). By throttling the rate at which we process mutations, we make it possible for the display clients to maintain a stable refresh rate of *60 FPS*.

4. All mutations are then recorded on a changeset and sequenced according to the order in which they must be executed. Any mutation that can be executed immediately will have an order of *0* while others will have some positive integer value.

5. If the changeset is not empty it will be broadcasted to the server at the end of each processing cycle.

6. The server application intercepts this changeset and determines whether they must be broadcasted to the display clients. When each display client establishes a connection with the server, it specifies a *viewbox* with the boundaries of the specific portion of a visualization that will be rendered within it. Mutations outside this *viewbox* are not relevant to a display client, and are culled by the server to ensure a minimum communications overhead.

7. The server then forwards relevant mutations to the respective display clients, which interpret them. This means that the server component would not have any significant performance impact.

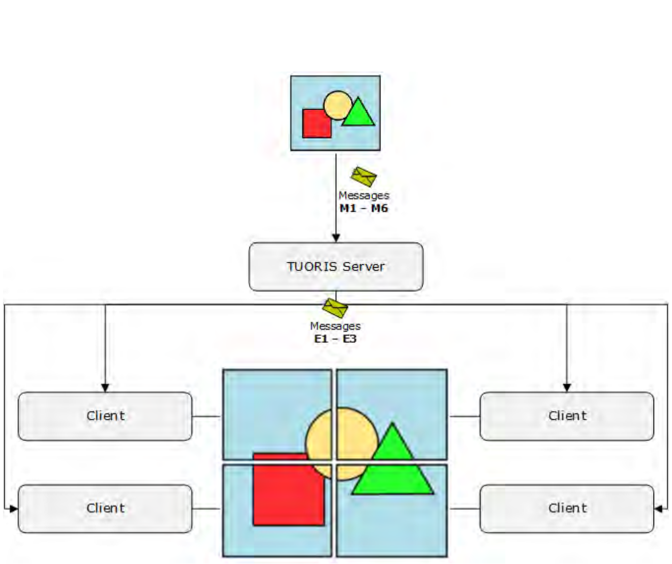
The server receives messages from the control client in six different formats, which are explained in Table 1.

	message description
M1	adds or updates a DOM node and includes a tag name, namespace, parent identifier, payload and an order.
M2	removes an existing DOM node.
M3	updates a value of a named DOM attribute.
M4	provides details of the bounding box of a given node as left, top, right and bottom.
M5	defines a collection of CSS rules.
M6	defines the *viewbox* for the parent SVG element.

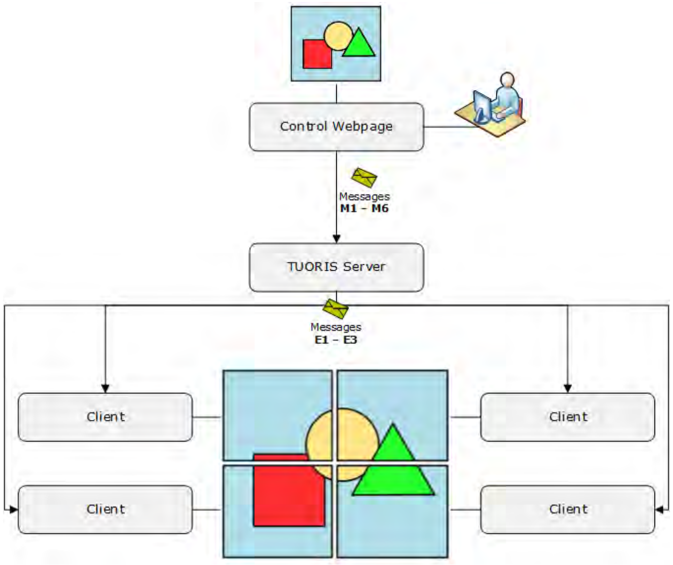
Table 1: Different types of messages received by the Tuoris server.

All messages include a *type* attribute, and messages of type **M1**–**M4** also include a *node identifier*. Messages of types **M5** and **M6** are different to other messages. **M5** is only sent once when the drawing is first loaded and **M6** is only sent again if the SVG element's *viewbox* changed due to resizing the webpage, for example. The server broadcasts these messages to the display clients in four different formats:

E1: represents a DOM node using a sequence of properties: type, node identifier, parent identifier, tag name, namespace, attributes, text content and bounding box details.



(a) In the **non-interactive mode** the input SVG is rendered on the headless browser. The different parts are then sent to the appropriate display clients.



(b) In the **interactive mode** the input SVG is rendered on the *control* webpage. Users interact with the content on this webpage. The different parts are then sent to the appropriate display clients.

Figure 1: Diagram of the Tuoris system, depicting the two modes of operation.

E2: includes the type and a collection of CSS rules.

E3: includes the type and the *viewbox* for the parent SVG element.

Messages of type **E1** are very frequent, while messages of type **E2** and **E3** are infrequent and may only be sent when the page first loads. All of these are broadcasted to the display clients in batches as and when the server receives new messages.

The display clients treat messages of types **E1**–**E3** as events and enqueues them onto a queue on receipt which will be processed afterwards. The display client pushes a special type of event, **E4**, on to the same queue at the end of each batch to instruct the system to clean up any nodes that are dangling or invisible within the given range for a specific display client as a result of a transformation.

The client-side processing of the event queue is also at the same rate at which the display refreshes (usually *60 FPS*). And similar to the server-side algorithm it too processes updates until there are no more or up to 30ms at a time, which ensures that the system is never overloaded. Due to culling at the server-side the workload at the client-side is generally much lower and avoids lags and tearing of content at each refresh interval. The special event of type **E4** makes the clean-up a part of this iterative workflow and does not introduce any additional overheads to the system.

To avoid lags and inter-screen tearing, our implementation uses the *requestAnimationFrame* method, and therefore relies on synchronized clocks on the client-side machines. The videos provided as supplementary material confirm that there are no lags and no visible tearing.

A frame rate of around 15FPS is deemed sufficient for a human to perceive something as interactive [37, 38]. Film and television cameras use 24FPS and 30FPS frame rates and most displays available today achieve a maximum frame rate of 60FPS. Figure 2 is a photograph of Tuoris achieving up to 60FPS for a typical visualization. We provide a separate video of the same as supplementary material to confirm that Tuoris is capable of supporting interactive visualizations.

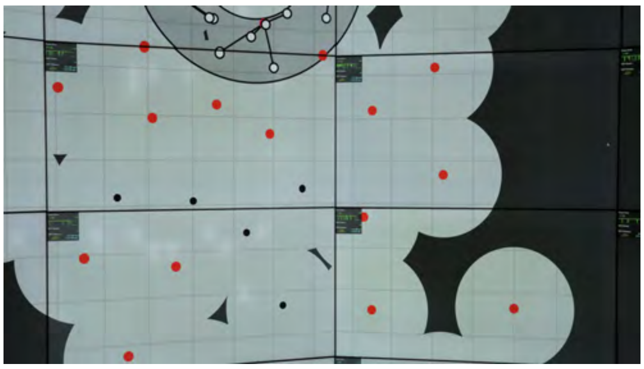


Figure 2: Photograph of Tuoris achieving up to 60FPS framerate (measured using the *FPS meter* of the Google Chrome web browser).

Tuoris is designed to handle highly dynamic visualizations. In each interactive step the elements of an SVG drawing may be transformed or moved to different areas of the display area and thus becoming no longer relevant to a specific monitor of an SRDE. To make rendering very efficient, these elements are discarded and recreated only if they reappear by means of a specific *element lifecycle*. The element is created if there are events corresponding to a non-existing element. It is retained as long as there are no events to mutate it. If one or more attributes of this element are changed they will be added or removed, accordingly. If the element has no parent or lies outside the *viewbox* of the display client it would be removed.

The design of Tuoris demands a moderately performant server-node but the clients need not have specialized hardware resources. These are confirmed in the tests that we carried out (explained in the next section). We also find that the flexibility as well as the scalability of Tuoris is due to its modular design and rate-limited message processing architecture. This also ensures consistency of visualization regardless of the volume of messages that are exchanged between the server and client components.

4. Examples and performance analysis

In order to validate our design hypothesis and subsequent implementation, in this section we present different examples of visualizations powered by the Tuoris system, covering a wide spectrum of use cases. We also use them as means of demonstrating and evaluating different aspects of performance, scalability and transparency of Tuoris.

Performance and scalability are fundamental to any design of a distributed system, be it specialized in visualization or otherwise. But, the complex nature of such systems and their dependencies on various other software and hardware elements makes it hard to make an accurate measurement of these characteristics. Therefore, in order to perform a fair evaluation, we designed a battery of tests and ran them at Imperial College London's immersive data observatory. It is composed of 64 full HD Samsung UD46D-P professional video wall monitors, arranged in a cylindrical layout in 4 rows and 16 columns; they are powered by 32 rendering nodes. This system offers a scalable display wall with a total resolution of 30720 pixels of width and 4320 pixels of height, resulting in a display of 132.7 megapixels, and have been also showcased in previous related publications [4, 5]. The Google Chrome web browser was used in the client nodes.

To demonstrate that our middleware supports transparency, we used previously created web-hosted content with no modifications in all of our use-cases; we simply provided Tuoris with a URL using its interactive mode. For our tests, we chose over 10 different SVG visualizations available on the internet, with different types of content. The particular visualizations, their URLs, and a reference to a screenshot of it running are given in Table 2.

4.1. Transparency

Our tests on transparency are grouped into five different categories. The first category of tests comprises of graphs developed using the popular D3.js framework:

T1a: *Les Misérables* character co-occurrence network represented as an adjacency matrix diagram which covered scenarios such as animation, selection of nodes and redrawing.

T1b: *Les Misérables* character co-occurrence network drawn as a force-directed graph. We were able to re-position a node and observe the dynamic re-drawing of the graph based on the forces.

T1c: a clustered force layout based on multiple custom forces to test the interactivity and the responsiveness of multiple viewers driven by a single controller. It proved that the effect of random forces does not cause the diagram to be drawn differently on individual display clients.

T1d: a multi-line chart rendered on a white and a black background to check the sharpness of rendering of the lines, text and also the axes.

T1e: a hive-plot that displayed 764 dependencies among 220 classes, where we could select nodes or links to understand how connected they were.

The next category includes other diagrams with animations:

T2a: a star map of the Northern Hemisphere using a flipped stereographic projection.

T2b: an animation that demonstrated how Bridson's Poisson disc sampling algorithm [39] works.

The third, fourth and fifth categories are content of various formats such as 3D graphics, converted office documents and geospatial content:

T3: the capability of rendering 3D content using an example from Three.js,

T4: the ability to render PowerPoint presentations and PDF documents converted into the SVG format using the Convertio online image converter,

T5: a geographic map of the Czech Republic (a large -15MB-SVG file which included many features). The same SVG is reused in subsequent tests to validate scalability (Section 4.2) and performance (Section 4.3).

Using these 10 visualizations, we confirm that Tuoris is able to properly render animated graphics of various formats. We made sure to include non-animated (and therefore non-interactive) content (**T1d**) as well as animations that require no user interaction (**T2b**, **T3** and **T5**). Figure 3 (a–f) includes photographs of each of the above test cases and we have included videos of some of them as supplementary material. Our tests did not cover all types of graphs that can be produced using D3 as well as graphs that can be rendered using other similar software such as Raphaël [40] and Graphviz [41], but we made sure to cover reasonable ground to confirm successful adoption of Tuoris to render a wide majority of graphs developed in the SVG format at very high resolutions spanning many screens.

Published research show that people prefer larger displays [42] and that maximizing the available screen real estate demonstrably affects visualization [11]. Larger displays are particularly useful for rendering high resolution content that may only be readable and interpretable at such dimensions. With Tuoris it is easy to switch between various resolutions and select what best suits the content that is displayed. To show the benefits we have provided two screenshots of Tuoris rendering a modified version of the adjacency matrix diagram from **T1a**

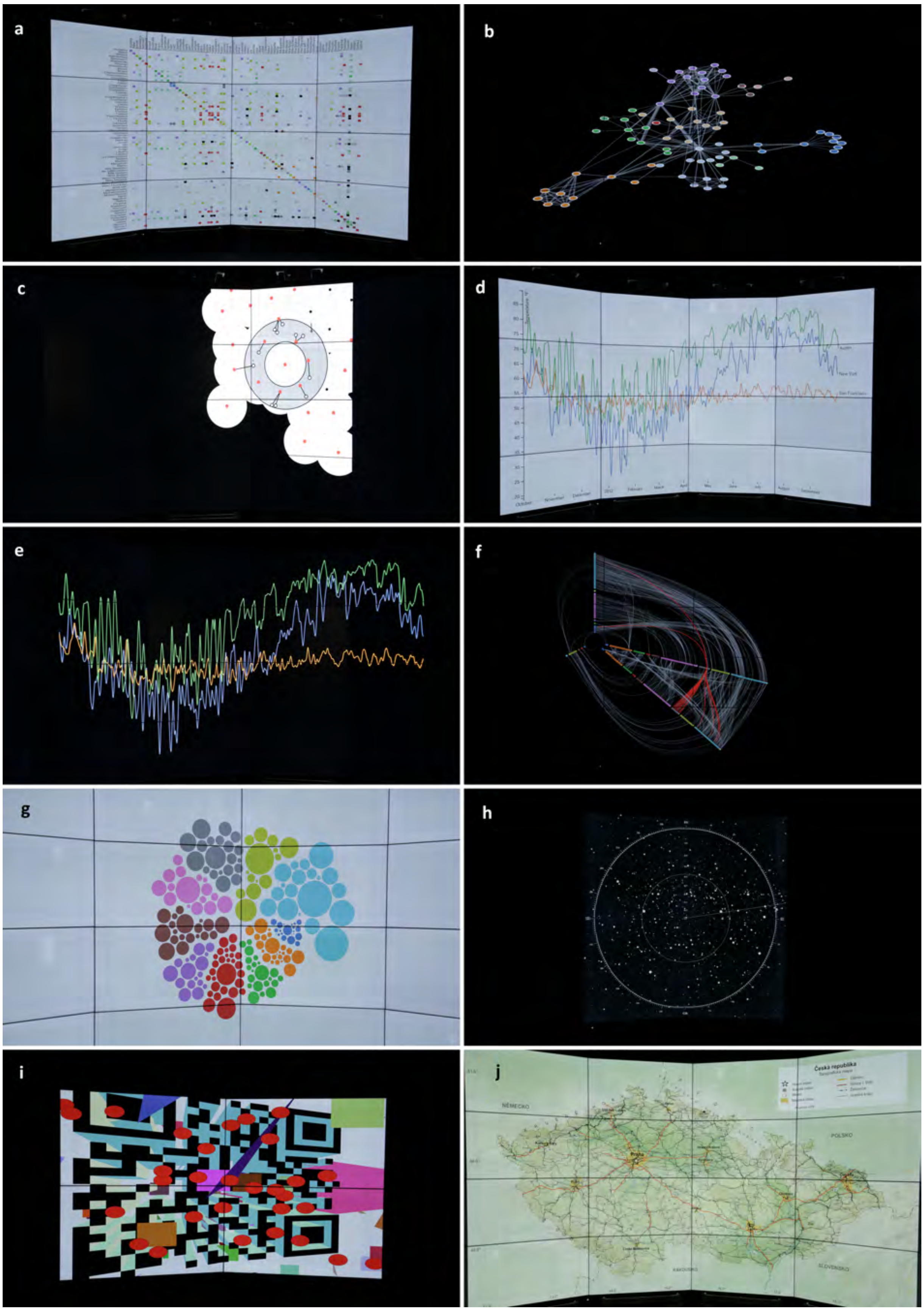


Figure 3: Photographs taken for various tests carried out on Tuoris

Test	URL	Figure
T1a	https://bost.ocks.org/mike/miserables/	3a
T1b	https://bl.ocks.org/mbostock/raw/4062045/5916d145c8c048a6e3086915a6be464467391c62/	3b
T1c	https://bl.ocks.org/mbostock/raw/7881887/c4cad93c5eaf159cccb8d6858d72894c45fbb6be/	3g
T1d	https://bl.ocks.org/mbostock/raw/3884955/95ccdeac9bbf2012300eb16f8109514e5ea234a2/	3d,3e
T1e	https://bost.ocks.org/mike/hive/	3f
T2a	https://bl.ocks.org/mbostock/raw/c7e85d2b47d11982db38/941bfa0419ef6b246e05f2d8525f8020dae36c89/	3h
T2b	https://bl.ocks.org/mbostock/raw/dbb02448b0f93e4c82c3/ae0487ceddd1d0763e5437c4e445f3f25319e698/	3c
T3	https://threejs.org/examples/svg_sandbox.html	3i
T4	https://convertio.co/image-converter/	–
T5	https://upload.wikimedia.org/wikipedia/commons/0/08/CzechRepublic-geographic_map-cz.svg	3j
F1	https://www.jasondavies.com/maps/sphere-spirals/	–
F2	https://bl.ocks.org/mbostock/raw/4636377/15ee95d79908587bc6fa3c1b1e7019b45739b9a6/	–

Table 2: Visualizations used on the different test scenarios. Their original URLs are provided (a majority of these are from D3 examples developed by Mike Bostock). The corresponding images are all found in Figure 3.

displaying 10 times as much information at two different resolutions in Figures 4 and 5 Furthermore, this also makes a case for why SRDEs displaying high resolution content is actually useful.

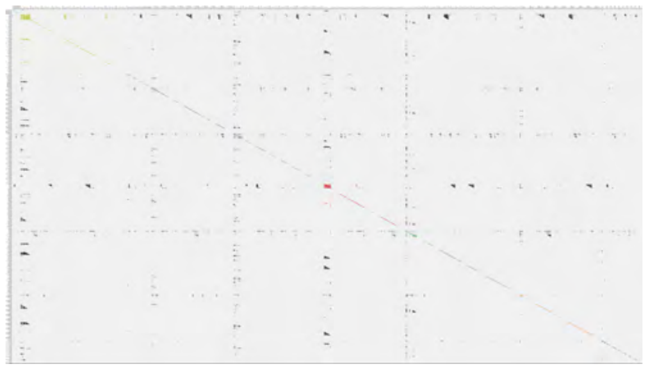


Figure 4: Screenshot of a modified version of **T1a** displaying 10 times as much information at a 1920 x 1080 resolution.

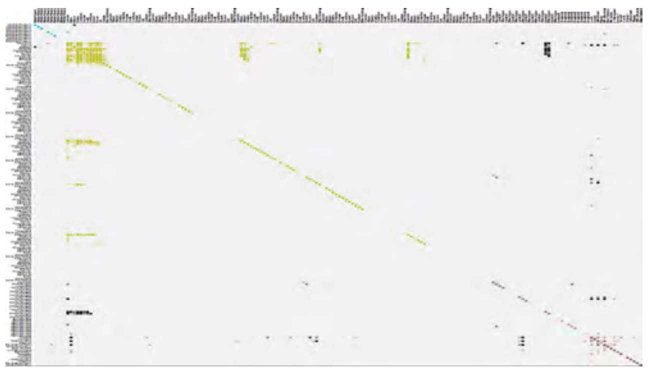


Figure 5: Screenshot of a modified version of **T1a** displaying 10 times as much information, same as Figure 4, but only shows the top-left $\frac{1}{16}$th at a 7680 x 4320 resolution.

Tuoris is an open source[6] cross-platform middleware which is straightforward to install; and, as we have used publicly available content in our tests with no modifications, interested readers only need to provide the system the same URLs to repeat these tests, or URLs of other similar visualizations to revalidate its capabilities. Tuoris can be deployed on an SRDE of any dimension using commodity hardware and it is not necessary to match the specification of our environment.

4.2. Scalability

To validate the scalability of our system we studied the server performance (total CPU and RAM consumption, as well as the time to complete the processing) while varying the display resolution and the number of clients. We reused the same SVG from **T5** above, as it was of a reasonable size (a large 15MB SVG) including many features. We ran tests in each configuration 10 times in order minimize potential interferences and ensure the reliability of our results. We tested 7 different combinations of resolutions and display nodes with a fixed server specification having 8 CPU cores and 16 GB RAM (see Figures 6 and 7 for results).

Based on our tests, we are able to confirm that the system is scalable despite the computation complexity of these visualizations and the resource consumption is proportional to the number of elements in the visualization rather than the number of pixels on the display wall, confirming its suitability for SRDEs. In addition to that the images always had smooth borders and corners (as we utilize the SVG standard), and occupied a number of pixels proportional to size of the displays and resolution of the individual monitors.

The load on RAM (see Figure 6) was almost constant at around 12% explaining that there are no memory bottlenecks (at least for this particular example). Also, in Figure 6 and by means of a box-and-whisker plot, we show that the amount of CPU used does vary but mostly according to the number of display nodes and not the resolution. Using fewer display nodes would reduce the CPU overhead, but we were also able to confirm that the increase in CPU consumption (with many display nodes) was due to the data transfer between the server and the display nodes rather than the computational overheads. This is confirmed by the varying server processing time (total CPU

[6] `https://github.com/fvictor/tuoris`

time instead of the average time spent per node) that we show in Figure 7, again as a box-and-whisker plot. In all of these tests the time taken by the clients to render the content was the same.

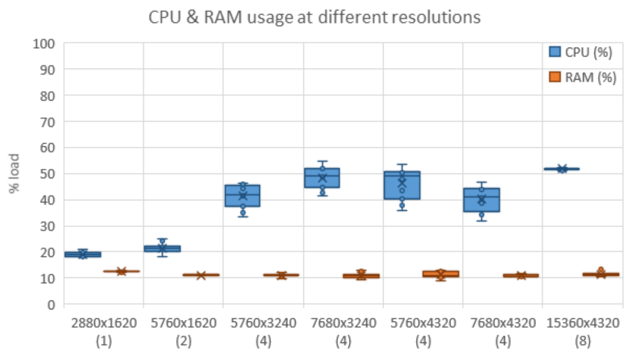


Figure 6: Comparison for the percentage of CPU load and RAM usage for the different resolutions (and number of display nodes) tested, represented as a box-and-whisker plot, summarizing 10 runs. The box in the middle represent the two central quartiles, and the line separating them is the median. The ends of the whiskers represent the minimum and maximum values.

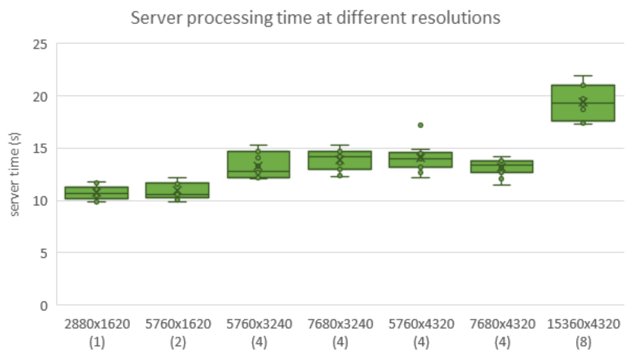


Figure 7: Comparison for the server time for the different resolutions (and number of display nodes) tested; represented as a box-and-whisker plot that summarizes 10 runs.

4.3. Performance

Next, we were also interested in evaluating the server-side performance of Tuoris and its associated overheads. For that, we carried out two additional tests with the aim of evaluating:

- performance of the system against different server resources (CPU/RAM) combinations.
- performance of the system while varying the complexity and file sizes of the SVG visualizations.

The corresponding test setups, results and main findings are described in the following subsections.

4.3.1. *Performance vs. server specification*

In these experiments, we fixed the resolution of the displays at 2880 x 1620 pixels and reused the SVG from **T5** as above. And, as before we tested each configuration a total of 10 times. The results are displayed in Figures 8 and 9. Note that

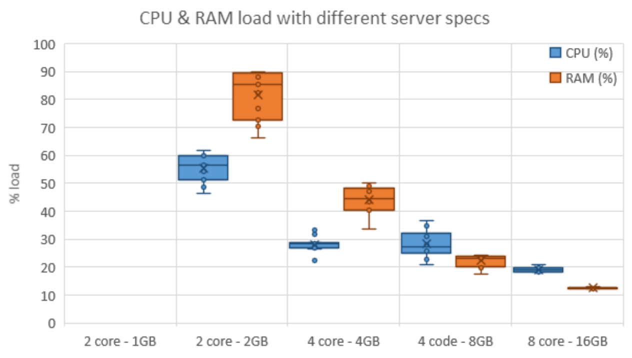


Figure 8: Comparison for the percentage of total CPU load and RAM usage for different combinations of CPU+RAM in the server; represented as a box-and-whisker plot that summarizes 10 runs. Note that Tuoris was not able to run in the first configuration.

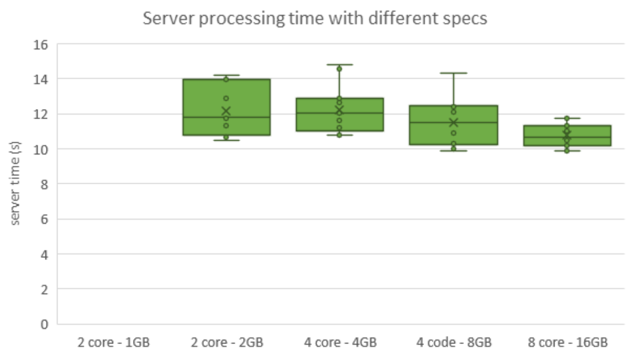


Figure 9: Comparison of server time to process the visualization for different combinations of CPU+RAM in the server; represented as a box-and-whisker plot that summarizes 10 runs. Note that Tuoris was not able to run in the first configuration.

Tuoris was not able to run in the configuration with the lowest system specification (2 CPUs and 1 GB of RAM).

In our results we find that the server processing time is very similar in all cases, and the variance reduces with the specification of the server. We also find that the percentage of memory used is linearly proportional to the amount of total memory available. The CPU requirement is non linearly proportional and it is noted that a reduction of the number of cores does not necessarily mean that the system demands a proportionally higher CPU. This confirms that the system is suitable for most commodity hardware configurations though it would certainly benefit from a server with a high number of CPU cores. It must be noted that a 15MB SVG with such high complexity is rare and the typical CPU and RAM consumption would be much lower for the average case. This is confirmed by the tests explained in the next subsection.

4.3.2. *Performance vs. complexity of the SVG*

Next, we fixed both the server specification (to 8 CPU cores and 16 GB RAM) and the display resolution (to 2880 x 1620 pixels), and evaluated with three SVGs: **T2a** of size 0.75 MB, **T1d** of size 0.12 MB, and **T5** of size 15 MB.

As expected, the CPU and RAM overheads (see Figure 10) as

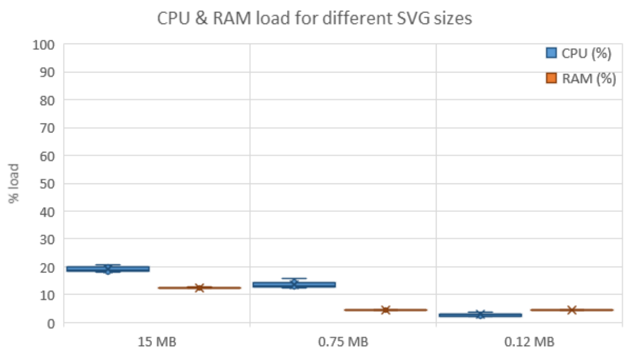


Figure 10: Box-and-whisker plot comparing the percentage of total CPU load and RAM usage of three visualizations with different number of elements.

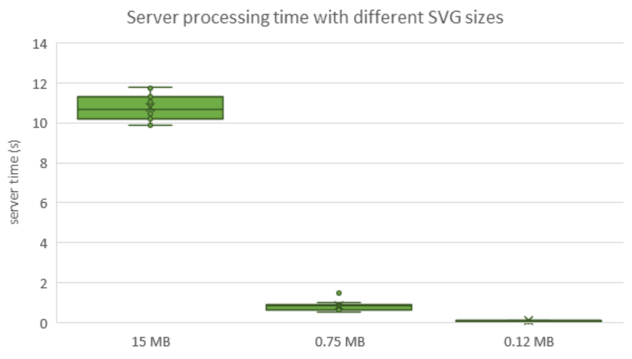


Figure 11: Box-and-whisker plot comparing server time to process 3 visualizations with different number of elements.

well as the server-side processing time (see Figure 11) reduces sharply with the size of the visualizations (i.e. the number of elements in the SVG) decreases. There is indeed a noticeable reduction of CPU overhead as there is a much lesser amount of data to be transferred from server to client as previously explained in Section 4.2.

The design of Tuoris has a limitation of having to transfer a new HTML document injected with JavaScript and SVG elements that are broadcasted not all at once but as a series of batches at a rate of *60 FPS*. This indirectly results in a prolonged server time and a correspondingly high initial CPU consumption, but the subsequent operations require much less CPU once the display nodes have built up their corresponding caches of SVG elements. Though this suits a wide majority of use-cases covering all the popular types of visualizations, it also has some limitations which we discuss in Section 4.5.

4.4. Comparison with SAGE2

Having measured the performance and scalability characteristics of Tuoris, we were eager to understand how well it compares against other web-based frameworks such SAGE2. Unlike Tuoris, SAGE2 is not a transparent framework and does not support a non-invasive programming model. For SVGs to be visible on SAGE2 they need minor code modifications, which is counterproductive. Therefore, should Tuoris compare reasonably well with SAGE2, it becomes a worthy alternative despite its limited scope.

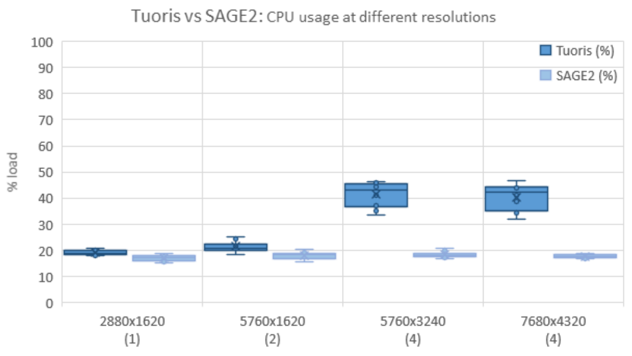


Figure 12: Box-and-whisker plot comparing the percentage of total CPU load on Tuoris and SAGE2, for different resolutions (and number of display nodes) of the SVG from **T5**, which is 15MB in size.

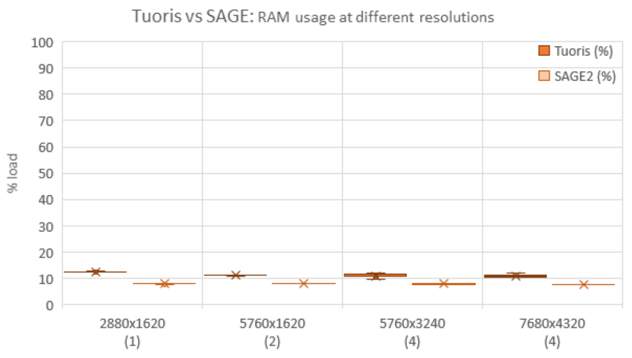


Figure 13: Box-and-whisker plot comparing total RAM usage on Tuoris and SAGE2, for different resolutions (and number of display nodes) of the SVG from **T5**, which is 15MB in size.

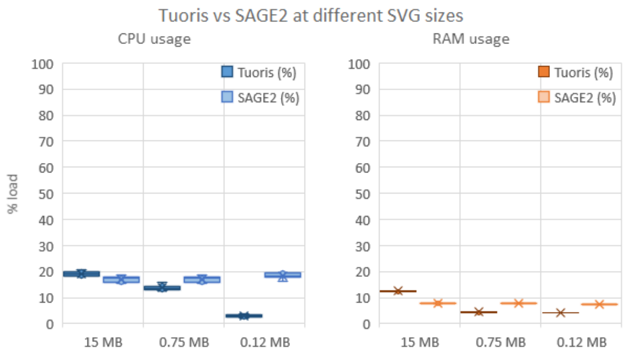


Figure 14: Box-and-whisker plot comparing percentage of CPU load and RAM usage for SVGs of different sizes on Tuoris and SAGE2.

Figures 12 and 13 show a comparison of CPU load and RAM usage on Tuoris and SAGE2, using once again box-and-whisker plots. From both of these charts, it becomes clear that Tuoris requires more CPU and RAM as the resolution increases while SAGE2 requires constant CPU and RAM. This shows that SAGE2 is a much more scalable framework in comparison. But, these observations are for a large SVG (**T5**, 15MB). Figure 14 provides a comparison for SVGs of various sizes using three SVGs from **T2a**, **T1d** and **T5**. From this chart, we can conclude that Tuoris performs better than SAGE2 for SVGs that are smaller in size.

Another important aspect is how well these two frameworks compare from a reliability, usability and maintainability point of view. As Tuoris supports a non-invasive programming model, it is much easier to use. Existing content can be directly loaded on an SRDE by simply providing a URL. Also, as there are no code modifications involved, the maintenance overhead is less. And, from a reliability point of view both frameworks compare equally.

But, SAGE2 has a much bigger limitation impacting D3-based visualizations; they all need to match the D3 version that is embedded within the SAGE2 framework (and similarly for other core libraries). This embedded version of D3 keeps changing between major releases of SAGE2. Therefore, a framework version change in SAGE2 becomes a major maintenance overhead that impacts all visualisations in contrast to Tuoris which has no overhead at all. Therefore, for projects using many SVG-based visualizations Tuoris is much better.

In summary, these observations suggest that Tuoris is a better option for visualizing content of a moderate size and a worthy contender for those with a much larger file size.

4.5. Limitations

Lastly, we discuss two scenarios where Tuoris consistently failed to render the SVG content within a reasonable amount of time on a computer with a reasonable hardware specification (8 CPU cores and 16 GB RAM):

Sphere spirals (F1): a visualization to explain the concept of spherical curves taken by ships that travel from the South Pole to the North Pole of the planet Earth while keeping a fixed angle with respect to the meridians. We found that the JavaScript code behind this visualization made rapid changes to the entire structure causing Tuoris to broadcast a very high volume of messages between the control client, server and display clients leading to a very poor performance (recall that Tuoris relies on broadcasting changes to the SVG structures rather than replicated execution of JavaScript on the browser). Tuoris worked well with a modified version of the code with less frequent changes.

Rotating Voronoi (F2): An animation based on Fortune's algorithm for computing the Voronoi diagram [43] or Delaunay triangulation of a set of two-dimensional points. In this example, we found that the web browser was struggling to paint the animation at very high resolutions (due to complexity in re-painting the image). Though this is out of scope for Tuoris, its heavy dependence on the browser makes this another key limitation.

These limitations can be reproduced by running Tuoris at a 7680 x 4320 resolution and providing the URLs of visualizations **F1** and **F2** (see Table 2).

5. Conclusions

This manuscript introduced Tuoris, an open-source middleware for visualizing dynamic graphics in SRDEs. This new tool eases the creation of complex, dynamic and interactive visualizations for immersive data observatories and scalable display walls and thereby enables collaborative visual analytics on big data. Tuoris is designed to be used by multidisciplinary teams, by hiding the underlying complexity from the developers, who can make use of standard web technologies and a number of popular programming languages, frameworks and libraries to create visualizations as if they were meant to run on a single computer with a standard display. The simplification of effort makes it much easier to explore big data in various settings.

Compared to other frameworks described in Section 2, such as SAGE2, Tuoris has a limited scope and focuses exclusively on content described in an SVG format. But, this trade-off makes it possible for Tuoris to meet three key requirements, performance, scalability and transparency – which are all required for collaborative visual analytics across a range of several SRDEs, such as the diverse visualization environment landscape at the Data Science Institute of Imperial College London.

Our tests confirm the suitability of Tuoris for these requirements, while highlighting its key feature which is the ability to simply provide a URL to a pre-created web-hosted SVG-based visualization and render it on an SRDE at the desired resolution with no loss of image quality, without making a single modification. These include a wide variety of graphs, drawings and a number of document formats, geospatial content along with 3D graphics and interactive animations. It meets the standard scalability and performance requirements of a distributed middleware suitable for SRDEs, and also is cross-platform and capable of being installed and run over completely heterogenous systems.

Among current limitations of Tuoris, we can cite that only content that can be represented through SVG is allowed. Therefore, videos cannot be displayed with this tool. We also pointed out two scenarios **F1** and **F2**, where Tuoris fails to render content of certain types within a reasonable amount of time. Also, in the interactive mode, it only supports one controller UI at a time.

Despite being designed to exclusively support SVG content, Tuoris can also be extended to support other vector image file formats such as PostScript, WebGL or CartoVL[7] (a proprietary geospatial data format). We also believe that further enhancements to the server-side algorithm should make it possible to use Tuoris for use-cases such as **F1**. **F2** however, would continue to remain out of scope. And, as future work, we plan to find a way to complement Tuoris with other technologies by running it as a library within frameworks such as SAGE2.

We have made Tuoris available as an open source middleware[8] under a permissive MIT license. It is straightforward to install and run and is immediately usable on display environments regardless of their scale and resolution. The simple URL-based approach makes it suitable for users with limited technical knowledge. We invite readers to download Tuoris and use it as a platform to develop new visualizations or use it to visualize

[7]https://carto.com/developers/carto-vl/
[8]https://github.com/fvictor/tuoris

their own pre-existing creations in immersive data observatories and scalable display walls.

Acknowledgments

V. Martínez was supported by the Spanish Ministry of Economy and the European Regional Development Fund (ERDF) under project TIN2012-36951 and the program "Ayudas para contratos predoctorales para la formación de doctores 2013" (grant BES-2013-064699). M. Molina-Solana was funded by the EU's H2020 programme (MSCA g.a. No 743623).

References

[1] R. Chang, C. Ziemkiewicz, T. M. Green, W. Ribarsky, Defining insight for visual analytics, IEEE Computer Graphics and Applications 29 (2) (2009) 14–17. doi:10.1109/MCG.2009.22.

[2] D. Che, M. Safran, Z. Peng, From big data to big data mining: challenges, issues, and opportunities, in: International Conference on Database Systems for Advanced Applications, Springer, 2013, pp. 1–15. doi:10.1007/978-3-642-40270-8_1.

[3] H. Jagadish, J. Gehrke, A. Labrinidis, Y. Papakonstantinou, J. M. Patel, R. Ramakrishnan, C. Shahabi, Big data and its technical challenges, Communications of the ACM 57 (7) (2014) 86–94. doi:10.1145/2611567.

[4] M. Molina-Solana, D. Birch, Y. Guo, Improving data exploration in graphs with fuzzy logic and large-scale visualisation, Applied Soft Computing 53 (2017) 227–235. doi:10.1016/j.asoc.2016.12.044.

[5] D. McGinn, D. Birch, D. Akroyd, M. Molina-Solana, Y. Guo, W. Knottenbelt, Visualizing dynamic Bitcoin transaction patterns, Big Data 4 (2) (2016) 109–119. doi:10.1089/big.2015.0056.

[6] C. Cruz-Neira, D. J. Sandin, T. A. DeFanti, R. V. Kenyon, J. C. Hart, The CAVE: audio visual experience automatic virtual environment, Communications of the ACM 35 (6) (1992) 64–73. doi:10.1145/129888.129892.

[7] A. Febretti, A. Nishimoto, T. Thigpen, J. Talandis, L. Long, J. D. Pirtle, T. Peterka, A. Verlo, M. Brown, D. Plepys, D. Sandin, L. Renambot, A. Johnson, J. Leigh, CAVE2: a hybrid reality environment for immersive simulation and information analysis, in: IS&T/SPIE Electronic Imaging, International Society for Optics and Photonics, 2013, p. 864903. doi:10.1117/12.2005484.

[8] S. Fernando, J. Amador, O. Şerban, J. Gómez-Romero, M. Molina-Solana, Y. Guo, Towards a large-scale Twitter observatory for political events, Future Generation Computer Systemsdoi:10.1016/j.future.2019.10.013.

[9] K. Li, H. Chen, Y. Chen, D. W. Clark, P. Cook, S. Damianakis, G. Essl, A. Finkelstein, T. Funkhouser, T. Housel, A. Klein, Z. Liu, E. Praun, J. P. Singh, B. Shedd, J. Pal, G. Tzanetakis, J. Zheng, Building and using a scalable display wall system, IEEE Computer Graphics and Applications 20 (4) (2000) 29–37. doi:10.1109/38.851747.

[10] S. Nam, S. Deshpande, V. Vishwanath, B. Jeong, L. Renambot, J. Leigh, Multi-application inter-tile synchronization on ultra-high-resolution display walls, in: Proc. first annual ACM SIGMM conference on Multimedia systems, ACM, New York, NY, USA, 2010, pp. 145–156. doi:10.1145/1730836.1730854.

[11] C. Papadopoulos, K. Petkov, A. E. Kaufman, K. Mueller, The Reality Deck–an immersive gigapixel display, IEEE Computer Graphics and Applications 35 (1) (2014) 33–45. doi:10.1109/MCG.2014.80.

[12] G. E. Marai, J. Leigh, A. Johnson, Immersive analytics lessons from the electronic visualization laboratory: A 25-year perspective, IEEE Computer Graphics and Applications 39 (3) (2019) 54–66. doi:10.1109/MCG.2019.2901428.

[13] G. Wallace, O. J. Anshus, P. Bi, H. Chen, Y. Chen, D. Clark, P. Cook, A. Finkelstein, T. Funkhouser, A. Gupta, et al., Tools and applications for large-scale display walls, IEEE Computer Graphics and Applications 25 (4) (2005) 24–33. doi:10.1109/MCG.2005.89.

[14] T. Ni, G. S. Schmidt, O. G. Staadt, M. A. Livingston, R. Ball, R. May, A survey of large high-resolution display technologies, techniques, and applications, in: Virtual Reality Conference, IEEE, 2006, pp. 223–236. doi:10.1109/VR.2006.20.

[15] J. Leigh, A. Johnson, L. Renambot, T. Peterka, B. Jeong, D. J. Sandin, J. Talandis, R. Jagodic, S. Nam, H. Hur, et al., Scalable resolution display walls, Proceedings of the IEEE 101 (1) (2013) 115–129. doi:10.1109/JPROC.2012.2191609.

[16] T. Marrinan, J. Aurisano, A. Nishimoto, K. Bharadwaj, V. Mateevitsi, L. Renambot, L. Long, A. Johnson, J. Leigh, SAGE2: A new approach for data intensive collaboration using scalable resolution shared displays, in: Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2014 International Conference on, IEEE, 2014, pp. 177–186. doi:10.4108/icst.collaboratecom.2014.257337.

[17] L. Renambot, T. Marrinan, J. Aurisano, A. Nishimoto, V. Mateevitsi, K. Bharadwaj, L. Long, A. Johnson, M. Brown, J. Leigh, SAGE2: A collaboration portal for scalable resolution displays, Future Generation Computer Systems 54 (2016) 296–305. doi:10.1016/j.future.2015.05.014.

[18] G. P. Johnson, G. D. Abram, B. Westing, P. Navrátil, K. Gaither, DisplayCluster: An interactive visualization environment for tiled displays, in: Cluster Computing (CLUSTER), 2012 IEEE International Conference on, IEEE, 2012, pp. 239–247. doi:10.1109/CLUSTER.2012.78.

[19] K.-U. Doerr, F. Kuester, CGLX: a scalable, high-performance visualization framework for networked display environments, IEEE Transactions on visualization and computer graphics 17 (3) (2011) 320–332. doi:10.1109/TVCG.2010.59.

[20] S. Eilemann, M. Makhinya, R. Pajarola, Equalizer: A scalable parallel rendering framework, IEEE Transactions on Visualization and Computer Graphics 15 (3) (2009) 436–452. doi:10.1109/TVCG.2008.104.

[21] S. Eilemann, A. Bilgili, M. Abdellah, J. Hernando, M. Makhinya, R. Pajarola, F. Schürmann, Parallel rendering on hybrid multi-gpu clusters, in: Eurographics Symposium on Parallel Graphics and Visualization, The Eurographics Association, Goslar, Germany, 2012, pp. 109–117. doi:10.2312/EGPGV/EGPGV12/109-117.

[22] S. Whitman, A task adaptive parallel graphics renderer, in: Proc. 1993 IEEE Parallel Rendering Symposium, IEEE, 1993, pp. 27–34. doi:10.1109/PRS.1993.586082.

[23] S. Molnar, M. Cox, D. Ellsworth, H. Fuchs, A sorting classification of parallel rendering, IEEE Computer Graphics and Applications 14 (4) (1994) 23–32. doi:10.1109/38.291528.

[24] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, J. T. Klosowski, Chromium: a stream-processing framework for interactive rendering on clusters, ACM Transactions on Graphics (TOG) 21 (3) (2002) 693–702. doi:10.1145/566654.566639.

[25] H. Peng, H. Xiong, J. Shi, Parallel-SG: Research of parallel graphics rendering system on PC-cluster, in: Proc. 2006 ACM International Conference on Virtual Reality Continuum and Its Applications, VRCIA '06, ACM, New York, NY, USA, 2006, pp. 27–33. doi:10.1145/1128923.1128929.

[26] H. Chen, Y. Chen, A. Finkelstein, T. Funkhouser, K. Li, Z. Liu, R. Samanta, G. Wallace, Data distribution strategies for high-resolution displays, Computers & Graphics 25 (5) (2001) 811–818. doi:10.1016/S0097-8493(01)00123-6.

[27] Y. Chen, H. Chen, D. W. Clark, Z. Liu, G. Wallace, K. Li, Software environments for cluster-based display systems, in: Proc. 1st IEEE/ACM Int. Symposium on Cluster Computing and the Grid, IEEE, 2001, pp. 202–210. doi:10.1109/CCGRID.2001.923194.

[28] H. Chung, C. Andrews, C. North, A survey of software frameworks for cluster-based large high-resolution displays, IEEE Transactions on Visualization and Computer Graphics 20 (8) (2014) 1158–1177. doi:10.1109/TVCG.2013.272.

[29] B. Jeong, J. Leigh, A. Johnson, L. Renambot, M. Brown, R. Jagodic, S. Nam, H. Hur, Ultrascale collaborative visualization using a display-rich global cyberinfrastructure, IEEE Computer Graphics and Applications 30 (3) (2010) 71–83. doi:10.1109/MCG.2010.45.

[30] K. Moreland, B. Wylie, C. Pavlakos, Sort-last parallel rendering for viewing extremely large data sets on tile displays, in: Proc. IEEE 2001 Symposium on parallel and large-data visualization and graphics, IEEE Press, 2001, pp. 85–92.

[31] K. Moreland, Image Composition Engine for Tiles, https://www.osti.gov/servlets/purl/1231491 (Aug. 2011).

[32] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, P. Hanrahan,

WireGL: a scalable graphics system for clusters, in: Proceedings of the 28th annual conference on Computer graphics and interactive techniques, ACM, 2001, pp. 129–140. doi:10.1145/383259.383272.

[33] N. Nirnimesh, P. Harish, P. Narayanan, Garuda: A scalable tiled display wall using commodity pcs, IEEE Transactions on Visualization and Computer Graphics 13 (5) (2007) 864–877. doi:10.1109/TVCG.2007.1049.

[34] M. Bostock, V. Ogievetsky, J. Heer, D^3 data-driven documents, IEEE Transactions on visualization and computer graphics 17 (12) (2011) 2301–2309. doi:10.1109/TVCG.2011.185.

[35] S. Tilkov, S. Vinoski, Node.js: Using javascript to build high-performance network programs, IEEE Internet Computing 14 (6) (2010) 80–83. doi:10.1109/MIC.2010.145.

[36] V. Pimentel, B. G. Nickerson, Communicating and displaying real-time data with websocket, IEEE Internet Computing 16 (4) (2012) 45–53. doi:10.1109/MIC.2012.64.

[37] J. Davis, Y. Hsieh, H. Lee, Humans perceive flicker artifacts at 500, Scientific Reports 5. doi:10.1038/srep07861.

[38] S. Balter, Fluoroscopic frame rates: not only dose, American Journal of Roentgenology 203 (3) (2014) W234–W236. doi:10.2214/AJR.13.11041.

[39] R. Bridson, Fast Poisson disk sampling in arbitrary dimensions, in: ACM SIGGRAPH 2007 Sketches, SIGGRAPH '07, ACM, New York, NY, USA, 2007, p. 22. doi:10.1145/1278780.1278807.

[40] D. Baranovskiy, Raphaël–javascript library, http://raphaeljs.com (2010).

[41] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, G. Woodhull, Graphviz–open source graph drawing tools, in: International Symposium on Graph Drawing, Springer, 2001, pp. 483–484. doi:10.1007/3-540-45848-4_57.

[42] R. Ball, C. North, Realizing embodied interaction for visual analytics through large displays, Computers & Graphics 31 (3) (2007) 380–400. doi:10.1016/j.cag.2007.01.029.

[43] S. Fortune, A Sweepline Algorithm for Voronoi Diagrams, in: Proceedings of the Second Annual Symposium on Computational Geometry, ACM, New York, NY, USA, 1986, pp. 313–322. doi:10.1145/10515.10549.