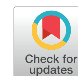
Journal of Cultural Heritage 41 (2020) 142–151

Available online at
ScienceDirect
www.sciencedirect.com

Elsevier Masson France
EM|consulte
www.em-consulte.com/en

Original article

An efficient GPU approach for designing 3D cultural heritage information systems☆



Luis López*, Juan Carlos Torres, Germán Arroyo, Pedro Cano, Domingo Martín

Virtual Reality Laboratory, University of Granada, Granada, Spain

A R T I C L E I N F O

Article history:
Received 29 September 2018
Accepted 14 May 2019
Available online 3 August 2019

Keywords:
3D information systems
Information layers
3D models
3D digitisation
GPU programming

A B S T R A C T

We propose a new architecture for 3D information systems that takes advantage of the inherent parallelism of the GPUs. This new solution structures information as thematic layers, allowing a level of detail independent of the resolution of the meshes. Previous proposals of layer based systems present issues, both in terms of performance and storage, due to the use of octrees to index information. In contrast, our approach employs two-dimensional textures, highly efficient in GPU, to store and index information. In this article we describe this architecture and detail the GPU algorithms required to edit these layers. Finally, we present a performance comparison of our approach against an octree-based system.

© 2019 Elsevier Masson SAS. All rights reserved.

1. Introduction

Documentation of cultural heritage artefacts is one of the most important tasks in terms of understanding and preserving tangible heritage. This process usually involves handling huge sets of heterogeneous data: photographs, illustrations, recordings, logbooks, plans, diaries, databases, digitized 3D models, etc. In traditional information systems, these data are usually stored in the absence of any kind of cross-referencing between the 3D model and the database.

The development of 3D scanning technologies over the last decade has allowed us to capture highly accurate representations of cultural heritage artefacts. Following the application of the proper processing techniques, the resulting 3D models possess sufficient geometric detail for them to be useful in terms of taking measurements and performing geometric operations.

The geometric information offered by these digitized 3D models is connected with every other type of meaningful data used to document artefacts. After all, pictures are taken to accurately document specific areas, illustrations are drawn to emphasize finer details, logbooks are used to register the work done in different sections at specific times, etc. It thus seems natural to organize and store this information on the surface of the 3D models.

2. Research aim

This article presents a new architecture for cultural heritage information systems that borrows from the foundations and functionality of Geographic Information Systems (GISs) and applies them to 3D models. Our solution organizes the information in thematic layers that are mapped onto the surface of the 3D model (Fig. 1). The data of these layers is stored on 2D textures and texture coordinates are used to properly index that information. This new approach takes advantage of the parallelism and efficiency of the Graphics Processing Units (GPUs) to handle and operate these structures. Our approach also allows to associate information independently of the geometry of the 3D model.

3. Previous works

In this section, we classify previous works related to information systems into four categories according to the dimension of the space employed in the analysis and visualization of data.

3.1. 2D space

These methods represent the information by means of 2D structures and employ existing GISs with minor modifications to document tangible heritage. This choice provides a vast range of solid tools with powerful analytic and information retrieval capabilities. However, these methods lack a bidirectional connection between the stored information in a 2D plane and the digitized 3D

☆ The article is part of a special issue entitled/XXX, edited by XXX.

* Corresponding author.
E-mail addresses: luislopez@ugr.es (L. López), jctorres@ugr.es (J.C. Torres), arroyo@ugr.es (G. Arroyo), pcano@ugr.es (P. Cano), dmartin@ugr.es (D. Martín).

https://doi.org/10.1016/j.culher.2019.05.003
1296-2074/© 2019 Elsevier Masson SAS. All rights reserved.

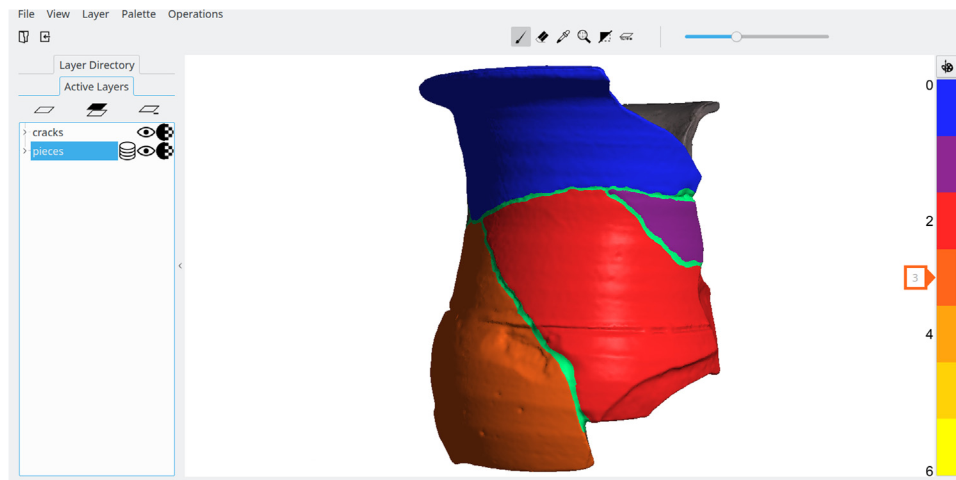


Fig. 1. Prototype of the proposed architecture. The user has associated two information layers to the 3D model of a vessel. The first layer *cracks* identifies the area of the cracks using a green color. The second layer *pieces* identifies the different pieces of the shattered vessel using the selected palette.

source model. It is, therefore, necessary to process this 3D model before using it as a valid input in these systems.

Naglič [1] and Ioannidis et al. [2] are good examples of this approach. They both use GISs for their work on large-scale archaeological sites, as these systems allow them to index large areas by geographical coordinates. Likewise, Parkinson et al. [3] employ GISs to study tooth marking patterns created by large contemporary canids on the bones of their preys and compare them with earlier fossils. They photograph the bones and manually create vector layers of the markings. In each of these cases they have to compromise the 3D nature of the source material in order to use GISs to analyse the data, limiting their entire work-flow to just one point-of-view at a time.

3.2. 2.5D space

These methods work with images and GISs to analyse and process the data. In contrast to the 2D approaches, the images are rasterized elevation models that contain height information. Therefore, they work in the restricted 3D space that this type of images provides. It is a more flexible and powerful method but, at the same time, it shares the issues of the 2D approaches. They need to convert the digitized 3D models into the appropriate image format and they are restricted to only one point-of-view.

Benito et al. [4] use this approach to classify stone tools employed by wild chimpanzees. They divide this classification in several stages: first they scan the tools, then they transform the resulting 3d models into digital elevation models and finally, they use morphometric GIS classification functions to discriminate between active and passive pounders in lithic assemblages.

3.3. Hybrid space

The methods under this category work in a 2D space during the stages of analysis and data processing and then visualize the resulting information in a 3D space. Two different transformation steps are required for this process. First, they need to project the initial 3D model, used as geometric reference, onto different 2D planes. The resulting images or digital elevation models are later processed in a GIS. Once this process has finished, the output needs to be projected onto the 3D model again as a texture. Although these methods lack the point-of-view restrictions of those described above, their work-flow is more complex and it suited only to tangible heritage easily divisible in 2D planes.

Campanaro et al. [5] used this approach to tackle the preservation of architectonic structures. They project the façade of buildings into multiple images, process them in different GISs and finally, they project the results onto a simplified version of the original 3D models of the buildings for their visualization.

3.4. 3D space

Unlike the previous approaches, these methods work directly with the digitized 3D models. There are no point-of-view restrictions and no transformations between different workspaces.

3.4.1. Annotation systems

The main goal of these systems is to associate information on specific sections of the 3D model surface and offer robust information retrieval tools. There is a wide spectrum of indexation mechanisms under this paradigm, ranging from submeshes of the original 3D model to lines or points in 3D space.

Durand et al. [6], Meyer et al. [7] and Mateos et al. [8] propose several online information systems to document archaeological sites. These systems require that the original 3D model is segmented into smaller and distinct entities. The analysis, processing and dissemination of information are done for each entity instead of the complete model.

Giunta et al. [9] use AutoCAD models to structure the architectural information and diagnostic investigation results of Milan's Cathedral Façade. Additionally, the system allows the user to insert pictures, texts and documents in a geo-referenced way.

Serna et al. [10] describe a distributed information system to annotate multimedia objects (3D models, images, text) using the concept of areas. Apollonio et al. [11] develop a web information system to document the restoration project of Neptune's Fountain in Bologne, Italy. The system allows the user to annotate the 3D model using three different primitives (points, polylines and areas) and to gather the stored data by means of robust information retrieval tools.

3.4.2. Layer based systems

These systems structure information attached to 3D models as a set of layers, where each layer stores the value of one attribute. Data layers can be considered as functions that associate a property value to points on the surface of the object. This type of systems can include the same functionality as the annotation systems and they can also perform complex operations between data layers. However, the main problem that these systems need to solve is how

to design efficient structures and methods to represent those data layers.

Torres et al. [12] divide the surface of the original 3D model into cells by means of an octree. Specifically, surfaces are recursively subdivided by eight cubic cells or voxels of the same size up to a predefined resolution level. Hence, each cell stores the triangles of the mesh that intersects. The level of detail depends on the size of these voxels and therefore, the number of subdivisions (levels) applied. The octree structure allows the user to work independently of the geometric irregularities and resolution. This way, fairly simple meshes are able to store information with better accuracy than the geometric mesh is able to offer. However, this spatial indexation becomes expensive in terms of memory and performance when dealing with the highest resolution levels.

The information layers are stored as sequences of properties assigned to the leaf nodes crossed by the surface [13,14]. This system not only is able to annotate or look up information, but it also allows the user to make complex computations with heterogeneous layers. Some of these computations include arithmetic, logic, geometric and topological operations or database queries, which can be used to analyze already existing data or to produce new information. Soler et al. [15] improve this system in order to solve specific topological problems at the expense of using a more complex data structure.

In this article we propose a new solution for these layer based systems that works in the 3D space and uses information layers to organize the information annotated on the 3D model. Unlike the Torres et al. [12] approach that requires an octree, our solution takes advantage of the modern GPU hardware by means of using textures and texture coordinates for data storage and indexation of the information. Our layers always reside in the memory of the GPU provided that there is memory available and all the operations are computed in the GPU. Consequently, the data-transfers between CPU and GPU are nonexistent and the editing of the layers is really efficient. For instance, the size of the edited area does not affect the performance. Moreover, the required time for the creation of these structures is insignificant in comparison to octrees and the size of our meshes is also independent of the layer resolution.

Next section describes in detail all the aspects of the data structures and algorithms required to create the proposed architecture.

4. Proposed approach

This section offers a comprehensive look at our proposed architecture. Section 4.1 introduces the concept of using textures to store data other than colours. Section 4.2 describes the layers and the rest of the structures. Finally, Section 4.3 details the algorithms required to edit the layers.

Ideally, whenever possible, structures and algorithms should be written with the strengths of the GPUs in mind to minimize the involvement of the CPU. Our architecture (Fig. 2) follows this principle and it defines all the structures directly in the memory of the GPU. Moreover, all the algorithms are solved in the GPU as well, so the CPU only has a management role. Layer manager handles the loading, storage and creation of layers. Texture manager is part of layer manager and issues commands to the GPU to create or destroy these structures but it does not store any of their information. Geometry manager handles the loading and storage of geometry files and uploads the geometry data efficiently to the GPU. Database manager resolves the creation and update of tables, tuples and queries. The disk drive is only used to load and store layers, databases and geometry permanently. Finally, the editing process requires a graphical user interface, therefore we have implemented a prototype (Fig. 1) based in our architecture.

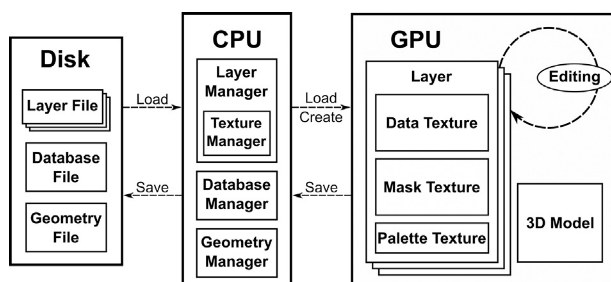


Fig. 2. General overview of the proposed architecture.

4.1. Texture as a heterogeneous data structure

Textures are usually defined as containers of one or multiple images in computer graphics. These images are arrays of texels of a certain dimensionality (1D, 2D or 3D) and they store the information following a specific format. Traditionally, textures have been widely used to add color information to the surface of 3D models. However, the birth of the programmable graphics and GPU computing pipelines have dramatically expanded their use during the last two decades.

Therefore, textures are multi-purpose structures nowadays. In fields like General Purpose GPU (GPGPU) computing, textures are usually treated as simple arrays or computation matrices. However, in Computer Graphics, it is a *de facto* standard to work with 3D models and textures coordinates, which are necessary to access and store the information. These coordinates are the result of projecting every primitive of the mesh, such as triangles, onto the texture space, which is usually a 2D space.

Since our information system works with 3D models, we use texture coordinates to index the data stored in 2D textures. This data is represented as integers or floating numbers of a certain size based on the type of the information layers.

4.2. Information layer

Our system handles two types of information layers:

- numeric layers store numeric properties such as curvature, rugosity, age and so on. The properties can be integer or real values. These layers are normally employed for annotating and operating with quantitative information;
- database layers have a relational database table associated. Regardless of how the user defines the table schema, the primary keys are always unsigned integers and they are also the values stored in the layer. Therefore, these layers establish a bidirectional relationship between the table of a database and the geometry of the 3D model. The use of tables is an elegant solution that allows the storage of complex heterogeneous data (text, dates, documents, pictures) and greatly enhances the semantic information of the 3D models. These layers are used for annotating and operating with qualitative information.

Fig. 2 depicts how layers are implemented in our system. Specifically, information layers consist of two different 2D textures and one 1D texture:

- Data Texture (2D) stores a number per texel. These numbers are integers (8, 16 or 32 bits) or float (16 or 32 bits) values. This texture holds the actual information of the layer, meaning property values in case of numeric layers, or primary keys in case of database layers;

- Mask Texture (2D) stores a boolean value per texel to determinate whether the texel of the Data Texture contains valid information or not;
- Palette Texture (1D) stores the necessary color information per texel to visualize the contents of the Data Texture. Each texel holds a four-component vector with the following color format: red, green, blue and alpha.

This wide range of texture options presents a challenge when the system needs to handle numerous instances of each option simultaneously. Our texture manager accomplishes this task using texture arrays [16] and sparse [17,18] bindless textures [19].

Traditionally GPUs can only use simultaneously as many textures as the number of texture image units they have available per shader stage. For instance, in modern NVIDIA graphics cards, fragment shaders can only access up to 32 different textures. To bypass this limit, our texture manager implements bindless textures. This feature allows our system to access textures in shaders by means of a handle without having to bind each texture to one texture image unit.

When there are many instances of the same type of textures, it is not efficient to use one handle for each texture because they can be grouped in categories. To make this possible, our texture manager classifies the textures using a hash function at the time of their creation. The hash key is constructed by joining their width, height and data type in a unique field. This way, the manager can group multiple textures that share similar features and use texture arrays to store them. These structures have multiples levels and each level stores a texture. At the same time, they only need one texture handle. Therefore, they are very useful for grouping textures. Whenever a new texture is required, the texture manager computes its hash key. If the key does not exist, the manager issues the creation of a new texture array to the GPU, stores the texture array identifier and assigns the texture to its first level. On the contrary, if the key already exists, the new texture can be part of an existing texture array and occupies the next available level.

Creating multiple textures arrays with a high number of levels can rapidly fill up the memory of the GPU. Even if we only need a small amount of textures per texture array, the graphic driver has to allocate all the space at the time of creation. Decreasing the number of textures per texture array does not solve the problem because there is still allocated memory that is not currently in use. Here is where sparse or partially resident textures offer an excellent solution. They allow the texture manager to allocate virtual memory space for texture arrays without wasting any physical memory until it is specifically requested. Only when the manager needs new textures, this virtual memory is committed and allocated into physical memory.

Memory management is, therefore, simpler and more flexible. There is no physical memory wasted. More texture arrays with a higher number of levels can be created, hence more instances of different types of layers can exist simultaneously. Sparse textures also improve the performance due to there is no data-transfer between CPU and GPU until the physical memory of the GPU is almost fully occupied. This offers another advantage: the textures always reside in the memory of the GPU, with the exception of two cases. In the first case, the user wants to save the layer to the hard disk and the three textures of the layer (Data, Mask and Palette) are transferred between GPU and CPU. In the second case, there is no enough space in the GPU memory to store a new layer and the graphic driver transfers unused textures to CPU memory to make room for the new ones.

Our solution results in an efficient approach due to GPUs excel at working with textures and textures coordinates. Modern GPU architectures have specialized hardware in the form of texture caches and texture mapping units that optimize the operations with these structures [20–22]. The cache exploits the spatial and temporal locality in accesses to reach high hit rates and the texture data is available to the shader processor with high throughput and low read latencies. Another advantage of this approach it is the independence between data and geometry in our model. Therefore, high resolution textures can be used to store information with a high level of detail on simple models.

Since these 2D textures do not store colour information but rather heterogeneous data, our system uses palettes that transform values into colours in order to render the layers. Our solution consists of a 1D texture that stores the colour information of the palette defined as a sequence of control points. Using the numeric value of the information layer per texel, another structure that contains the lower and upper limits of the palette and a single linear transformation, we can properly index the 1D texture and retrieve the correct colour to display.

4.3. Editing layers on the 3D model

Regarding to the user interface, once the editing mode is activated, an editing tool in the shape of a circle appears under the mouse cursor. The user can interactively add information to the selected layer by pressing the left button of the mouse and moving the tool over the desired area. This is a complex process that involves two distinct algorithms:

- Texture Editing Algorithm (TEA*)*, detailed in Section 4.3.1, translates the user inputs into valid texture coordinates and it stores the new values in the appropriate texels of the layer textures;
- Texture Padding Algorithm (TPA), described in Section 4.3.2, eliminates the visual artefacts that could appear when the application renders the layer on the surface of the 3D model.

4.3.1. Texture Editing Algorithm

TEA is a multi-step algorithm that projects the area of the editing tool, expressed in window coordinates, onto the space of the texture coordinates and stores the selected value on the appropriate textures permanently. Fig. 3 depicts a diagram of the whole process, which progressively discards regions of the texture until the edited area matches the orange shape of the editing tool. The inputs of each step are displayed on the left side. The 3D views with the results, on the right side. The center shows how the texture of the layer evolves through the different steps.

This algorithm requires two input textures in addition to other 3D model attributes, such as vertices, normals and texture coordinates and the information relative to the camera:

- Depth Texture is created as a part of the system pipeline. This texture stores the depth values of the scene from the current point-of-view. The user triggers an update when she modifies the viewpoint of the active camera or loads a new 3D model into the system;
- Editing Tool Shape Texture is created as a part of the system pipeline. This texture stores the shape of the current editing tool. The user triggers an update when she selects another shape for the editing tool.

Since the goal of the algorithm is the editing of textures, an off-screen framebuffer is set up and three textures are attached to it: the 2D textures of the layer, Data and Mask, and a temporal mask, Edited Area Mask. Each editing operation updates them as follows:

- Data Texture is updated with the value selected for the editing operation in the area marked by the user;

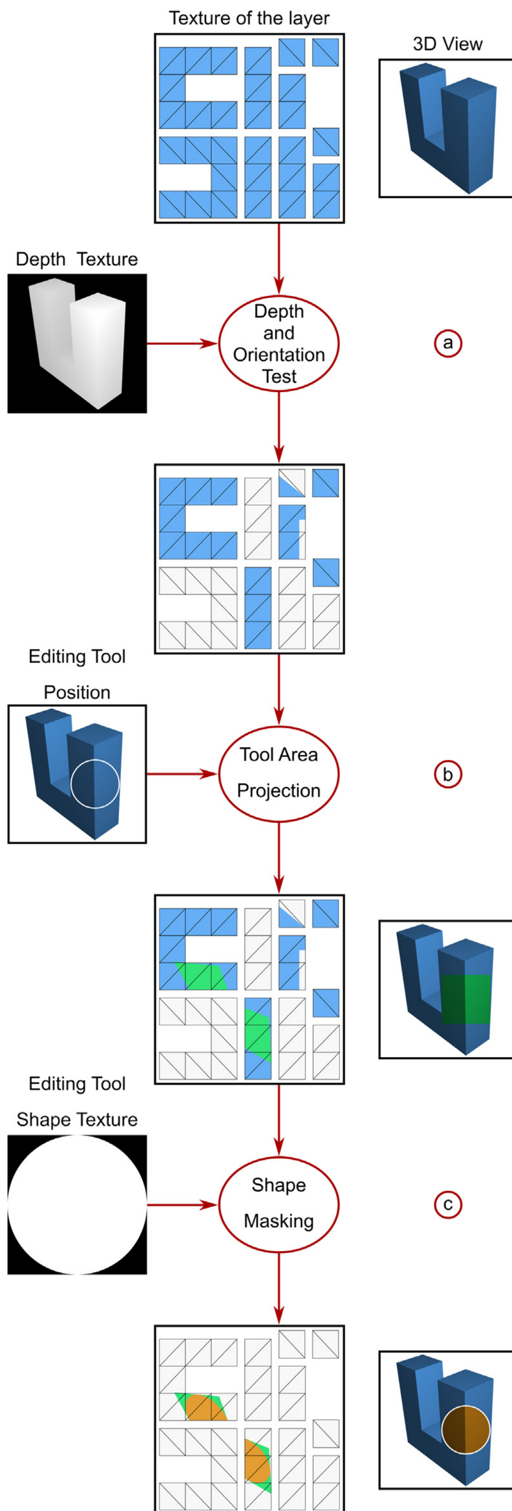


Fig. 3. Our algorithm filters areas of the texture progressively. (a) Depth and Orientation Test discards the occluded and back-facing parts of the 3D model; (b) Tool Area projection projects a squared area of the appropriated size onto the remaining area. Finally, (c) Shape Masking discards the texels that fall outside of the editing tool shape.

- Mask Texture is updated with the value *true* in the area marked by the user;
- Edited Area Mask is also updated with the value *true* but the system cleans it after each operation. Hence this is a mask of the area updated by the current editing operation.

The scene is rendered from the viewpoint of an orthogonal camera using the texture coordinates of the 3D model as vertex positions. That way, the mesh parametrization is rendered on a 2D plane or texture space. Fig. 3 shows the mesh parametrization of an u-shaped 3D model at the top of the central column. The black lines represent the 2D triangles created by the texture coordinates and the blue color, the actual rendered area in the texture.

TEA makes extensive use of projective texture mapping [23,24]. This technique assumes that textures are projected onto the scene by slide projectors. It is a similar procedure to projecting the scene to the screen but, in this case, the scene is projected to the slide projector. Specifically, the vertices of the 3D model are transformed using the modelview and projection matrices of the slide projector. This transformation maps the homogeneous coordinates of the vertices to clip coordinates:

$$\begin{bmatrix} X_{clip} \\ y_{clip} \\ Z_{clip} \\ W_{clip} \end{bmatrix} = M_{projection} \bullet M_{viewing} \bullet \begin{bmatrix} X_{obj} \\ Y_{obj} \\ Z_{obj} \\ W_{obj} \end{bmatrix}$$

where $(x_{obj}, y_{obj}, z_{obj}, w_{obj})$ and $(x_{clip}, y_{clip}, z_{clip}, w_{clip})$ are the vertex and clip coordinates, $M_{projection}$ and $M_{viewing}$ are the projection and modelview matrices. An additional scale and bias transformation is applied to translate the domain of clip coordinates [-1,1] to the domain of texture mapping [0,1]:

$$\begin{bmatrix} x_{prj} \\ y_{prj} \\ z_{prj} \\ w_{prj} \end{bmatrix} = M_{sb} \bullet \begin{bmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{bmatrix}$$

where $(x_{clip}, y_{clip}, z_{clip}, w_{clip})$ and $(x_{prj}, y_{prj}, z_{prj}, w_{prj})$ are the clip and projective coordinates, M_{sb} is the scale-bias matrix.

Unlike standard texture mapping that uses real texture coordinates (s,t), projective texture mapping uses homogeneous coordinates or coordinates in the projective space (s,t,w) where $s = x_{prj}$, $t = y_{prj}$ and $w = w_{prj}$. These coordinates are interpolated over the primitive and then at each fragment. The interpolated homogeneous coordinates are projected to the real texture space s/w (s/w, t/w) in order to access the texture image. The sampled value is the output of the projective texture mapping for the fragment.

Although TEA is actually solved in one single pass, we explain the process in three different conceptual steps:

(a) Depth and Orientation Test

It is undesirable that the system allows the user to edit the occluded areas of the 3D models. Since the scene is rendered on a 2D plane using the texture coordinates, TEA cannot use the default depth buffer to apply the depth test. To solve this issue, our algorithm makes use of the Depth Texture provided by the system pipeline, which stores the depth values of the 3D scene viewed from the active camera point-of-view.

The projector shares the same viewing and projection information of the 3D scene camera. The vertex positions of the 3D model are transformed and interpolated over the primitives which are the triangles represented by the texture coordinates. Therefore, each fragment has the depth value of the 3D scene, z_{prj}/w_{prj}, and this depth is tested against the value of the Depth Texture sampled by the coordinates (s/w, t/w) using a nearest-neighbor interpolation. A small bias is applied in this computation to fight possible precision issues. If the depth of the fragment falls behind the depth value of the texture, the

fragment is discarded. Otherwise, the rest of the algorithm proceeds normally.

The system also does not allow the user to edit back-facing triangles. To test their orientation, our algorithm computes the dot product between the camera vector and the mesh normals. If the result is less than zero, the triangle is back-facing and the fragment is discarded. Otherwise, the algorithm moves to the next step.

Fig. 3a shows what sections of the mesh parametrization, displayed in blue, are discarded after applying the depth and orientation test. The discarded areas are shown in white.

(b) Tool Area Projection

In this step the projector shares the same world position of the 3D scene camera, hence its viewing transformation is also identical. However, its projection transformation needs to be adjusted to the texture frustum. If the projection transformation were the same, the texture would be projected onto the whole screen. Therefore, the projection has to be limited to the 2D area occupied by the editing tool. The basic equation for 2D coordinate transformation accomplishes that:

$$T_c = T_f + S_f \bullet S_c$$

where T_c is the target coordinate, T_f is the translate factor, S_f is the scale factor and S_c is the source coordinate. The scaling and translate factors are computed with the following equations:

$$S_f = \left(\frac{W_w}{2T_w}, \frac{W_h}{2T_h}\right)$$

$$T_f = \left(\frac{T_{px}\text{-}0.5 \bullet W_w}{T_w}, \frac{T_{py}\text{-}\ 0.5 \bullet W_h}{T_h}\right)$$

where S_f and T_f are the scale and translate factor respectively, W_w and W_h are the window size (width and height), T_w and T_h are the texture size (width and height), T_{px} and T_{py} are the horizontal and vertical coordinates of the editing tool.

Fig. 3b shows the projection of the squared area containing the shape of the editing tool on the texture space using the position and size of the tool as inputs. The projection only takes place on the coordinates not discarded by the previous tests, which are the blue area of the texture. The green squared area is the output of this step.

(c) Shape Masking

At this point the coordinates need to be tested against the texture that stores the shape of the tool. The Editing Tool Shape texture is sampled using nearest-neighbor interpolation and this simple masking operation discards those coordinates that fall outside the shape. The final coordinates project the right shape onto the 3D model and determine which areas of the textures need to be updated.

Fig. 3c shows the masking operation using the editing tool shape as an input. The output of this algorithm is the orange rounded area and this step only takes place on the coordinates validated by the previous projection, which are the green area of the texture.

4.3.2. Texture Padding Algorithm

While our TEA solves the editing of the data textures, the usage of textures also entails some visualization artefacts. When GPUs display textured 3D models, the texture coordinates define how the texels are sampled and they are usually organized as sets of isles of different size and shape along a 2D plane. The space of texture coordinates is continuous while the textures or images are discrete. This disparity makes the conversion between both spaces prone to small inaccuracies around the edge of the isles. If there is no redundant information beyond the borders of the texture isles, visual discontinuities or artefacts can appear when these borders are sampled. In order to prevent this rendering issue, our padding algorithm expands the border data on some additional texels. The general idea of the process is shown in Fig. 4, where the inputs are displayed on the left side; the output, on the right side and the zoom-in area depicts the padding in detail.

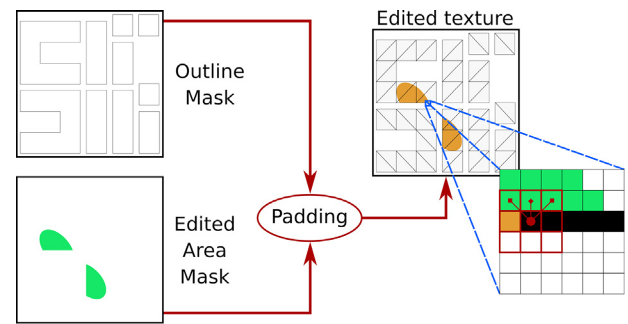


Fig. 4. In the zoomed area, the algorithm has verified that the current texel is part of the Outline Mask (black line) and the kernel (red matrix) is checking whether there are texels of the Edited Area Mask nearby (green shape). If so, it stores the value selected for the editing operation on the textures (orange color).

The algorithm requires two input textures:

- Outline Mask. This texture contains the outlines of the texture islands. It is created as a part of the system pipeline by a two-pass algorithm. The first pass uses the texture coordinates to render the mesh parametrization of 3D model into a 2D texture. The second pass is a 2D image filter that use the output of the first pass and checks whether each texel is at a distance of the islands less than or equal to one texel. It is updated only when a new 3D model is loaded into the system;
- Edited Area Mask. This texture contains the shape of the current edited area. It is updated every time the user edits the layer.

Both textures are sampled using nearest-neighbor interpolation.

This algorithm modifies the 2D textures of the layer edited in the previous subsection: Data and Mask textures. An off-screen framebuffer is set up with these two textures attached to it and each padding operation updates them as follows:

- Data Texture is updated with the value selected for the editing operation in the padded areas;
- Mask Texture is updated with the value *true* in the padded areas.

The scene is rendered from the viewpoint of an orthogonal camera and it consists of two triangles that represent a quadrilateral polygon of the same size of the textures.

The process is a quite straightforward image processing algorithm. It requires a kernel and the radius of the kernel is the width, in texels, of the desired padding. In this case, one texel is enough because the system uses nearest-neighbor interpolation to sample and render the layers. Each texel is tested as follows: the algorithm samples Outline Mask to check whether the texel is part of the outline of a texture island. If that is the case, the algorithm samples then Edited Area Mask to check whether its distance to the edited area is less than or equal to the kernel radius. The texels that satisfy both conditions are part of the padded area and therefore updated by the algorithm.

This padding adds some redundant information to the textures and guarantees the right colour when the GPU samples and renders the texture island borders. While this reduces the usable space in

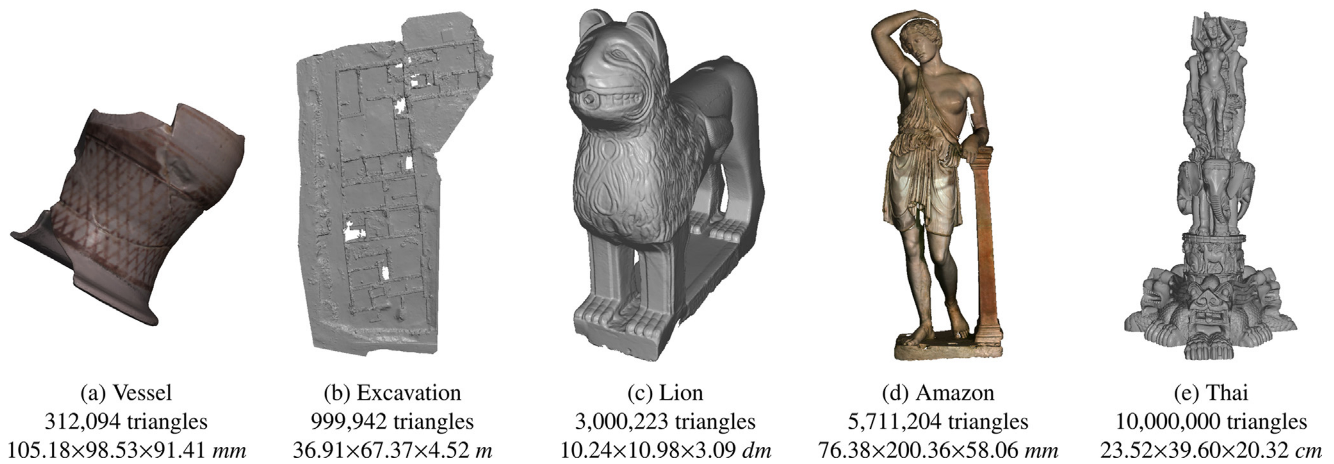


Fig. 5. 3D models used in the tests. Models (a) and (d) have colors per vertex while (b), (c) and (d) use a generic gray color. They are organized by number of triangles from left to right. The dimensions are also detailed for each model.

the texture, the reduction is not significant enough to be too costly because our padding is only one texel wide.

5. Results and discussion

In this section we compare the performance of the prototype based on our architecture against another system based on an octree. Specifically, we selected the system designed by Torres et al. [12], noted as OCT-TR, because both systems work directly with 3D models, implement structures to associate information independently of the number of triangles of the 3D models, use information layers to organize the data and therefore, they are very similar in terms of functionality.

The hardware specifications of the computer used to conduct these tests are as follows: CPU Intel i7 4790k at 4.00 GHz, 16 GB DDR3 RAM memory at 1866 MHz and NVIDIA GTX 970 at 1.114 GHz with 4 GB GDDR5 RAM memory at 7 GHz.

All the tests measured the performance of the layer editing process in equivalent scenarios. For our solution, this involves the two algorithms explained in the last section: Texture Editing and Texture Padding. For OCT-TR, it involves the CPU casting multiples rays to find which voxels of the octree they collide with, updating the layer values accordingly and transferring the new version of the layer to the GPU.

We chose three levels of detail for the information layers to analyse the performance when the cell size decreases: three different texture resolutions for our solution and three different depths for OCT-TR. Moreover, for each one of those three cases, we selected five different editing tool sizes to analyse the performance when the edited area growths. All these tests used the same five 3D models (depicted in Fig. 5).

Since octrees are volumetric spatial structures and textures are bi-dimensional spatial structures, there is no possible direct comparison between the area of the surface contained in one voxel and the area contained in one pixel. Therefore, we performed multiple tests to establish a correspondence between the cell size provided by texture resolutions and octree depths and empirically we reached to the following results: the 2048×2048 texture resolution is similar in terms of cell size to an octree depth of 11; the 4096×4096 texture resolution, to an octree depth of 12 and the 8192×8192 texture resolution, to an octree depth of 13. Table 1 shows the mean cell size, in square millimetres, achieved by both systems for the lion model. The results for the other four models are included in the appendix that accompany this article and they follow a similar pattern where our solution usually offers smaller mean cell size than OCT-TR.

Table 1
This table shows the mean cell size achieved by both algorithms for the lion model. The measurements are in square millimetres. The header of the columns shows the texture resolution and depth of octree that are tested against each other.

Method	2048 - depth 11	4096 - depth 12	8192 - depth 13
Our approach	0.7702	0.1926	0.0481
OCT-TR	0.8752	0.2188	0.0547

While our method had no problem to handle the Thai statue, it is important to note that the tests of OCT-TR with the depths of 12 and 13 for this model could not be completed. OCT-TR required a high amount of memory to create the octree itself and the rest of its structures. The memory manager of the operative system showed that the system was using over 30 GB of virtual memory before the application crashed.

Fig. 6 shows how the performance evolves when the edited area changes under a logarithmic scale. After a detailed examination, we can conclude that our approach exhibits a significantly better behaviour: the results growth linearly in contrast with those of OCT-TR. Though the theoretical behaviour of our algorithm seems to be linear, this turns to be constant in practice due to the almost zero slope of the line, no matter the size of the editing tool. The reasons behind this excellent performance are explained by the good use of the GPU resources. Our algorithm allocates the workloads between GPU cores evenly and minimises the stalls in the GPU execution pipeline because there is no interdependencies in the calculations. All the operations are independent and inexpensive in terms of cost; they also use structures (textures) that are completely optimised by the architecture of GPUs. In contrast, the poor results obtained by OCT-TR are due to the own nature of the octree. Unlike our solution, this structure resides in the primary memory and the CPU computes all the operations. Concretely, the editing process involves the casting of multiple rays over the different voxels of the structure. The bigger is the editing area, the higher is the number of rays and collisions to check. The depth of the octree is also critical in terms of complexity because the voxels are halved by two in each dimension between consecutive depths and therefore, the number of collision tests growths exponentially.

Fig. 7 shows, under a logarithmic scale, how the performance evolves when each 3D model is edited with an editing tool of the same size. After analysing the results, we conclude that our solution exhibits a better performance and a completely consistent behaviour across the five models. The time required to complete the editing operation increases when the number of triangles is higher or when the texture grows bigger. These results are reasonable

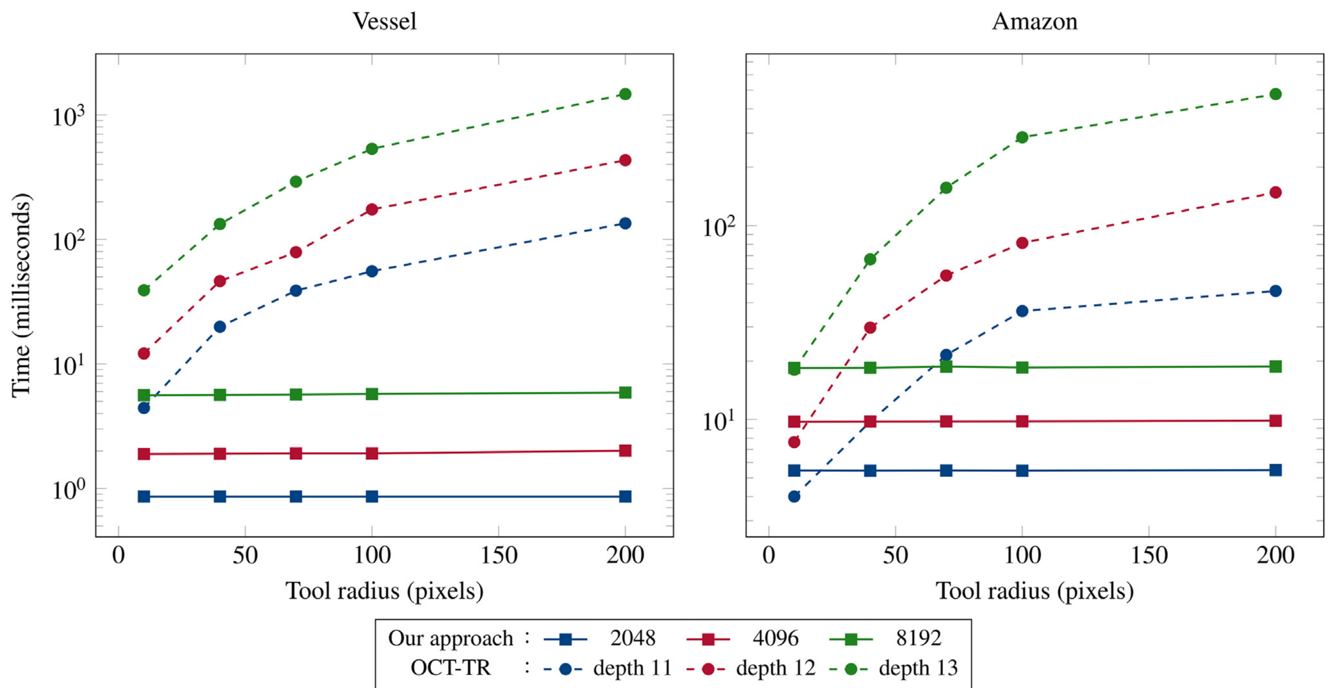


Fig. 6. These graphs show the tests results, under logarithmic scale, for the editing of layers of two models: a: vessel; b: Amazon. Tool radius corresponds to the radius of the editing tool in pixels. The solid lines correspond to our approach and the dashed lines to OCT-TR.

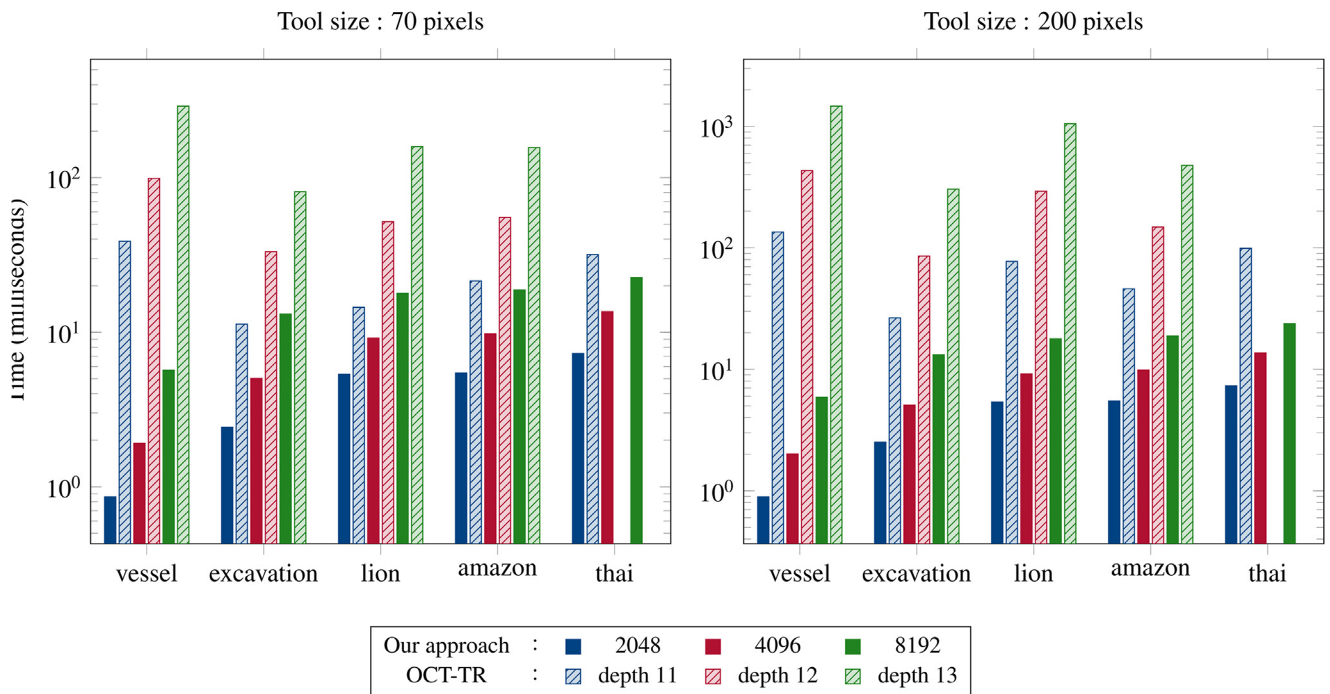


Fig. 7. These graphs show the tests results, under logarithmic scale, for the editing of layers using the same tool size on each model. Seventy pixel radius was used on the left graph and two hundred pixel radius on the right graph. The 3D models are organized by number of triangles. The solid colour bars correspond to our approach and the hatched bars to OCT-TR.

because even though our solution takes advantage of the parallelism of GPUs, GPU resources are limited. At the same time, our algorithm performs better than expected: the difference between the results of the vessel and the Thai statue is always less than one order of magnitude even though there is a difference of almost two orders of magnitude in terms of geometry. In contrast, OCT-TR shows inconsistencies between models because the performance is highly dependent on how well balanced the octree is and which area is edited. When objects are projected onto octrees, one of their main features is the ability to discard complete octants in order to reduce the number of collisions to check. The shape of 3D models and changes in their orientation can make the central region of the octree heavily populated. Projecting the editing tool onto that region can make impossible to discard any octant in advance due to all of them contain sections of the model, affecting negatively to the performance. Therefore, the shape and orientation of 3D

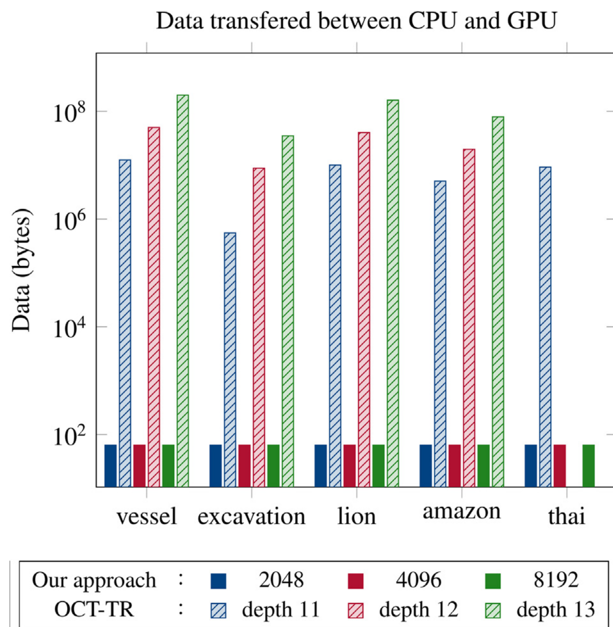


Fig. 8. This graph shows the amount of data, under logarithmic scale, that needs to be transferred between CPU and GPU during each editing operation. The solid colour bars correspond to our approach and the hatched bars to OCT-TR.

models are critical for spatial structures such as octress. The vessel and its sloped orientation is a perfect example of this disadvantage. It is the more demanding model even though it has the lowest number of triangles. Moreover, the performance is two orders of magnitude worse than our solution in the worst case tested, taking over a second to complete one single editing operation.

Fig. 8 shows, under a logarithmic scale, the amount of data that our approach and OCT-TR transfer to the GPU for each editing operation. After analysing the results, it is evident that our approach performs significantly better. Since our layers always reside in GPU, our solution only transfers the 64 bytes of the matrix that represents the position of the editing tool. In contrast, OCT-TR uses two different representations for its layers: the primary memory stores the data and the GPU memory stores the colours. Therefore, OCT-TR have to send the updated version of the layer colours to the GPU in order to visualize the changes. These transfers require almost 200 Megabytes (MB) for the more detailed layers (depth 13) of the vessel every time an editing operation is performed.

Finally, our solution is usually more space efficient too. While our layers are bigger in size, our 3D model representation is more compact because it is constant in terms of space independently of the resolution of the layers. Unlike our solution, OCT-TR subdivides its meshes when the octree depth increases. Therefore, our approach is better when the number of layers used simultaneously is below a threshold. Using the vessel as example, our system is more efficient in terms of space with less than eleven layers. Specifically, the size of our 3D model is 12 MB while OCT-TR requires 1.268 GB to store its model at depth of 13. However, for the highest resolution, the size of our layers is 327 MB while the size of OCT-TR layers is 199 MB.

6. Conclusions

In this article we have proposed an efficient architecture for cultural heritage information systems. We also have carried out empirical tests comparing our approach to OCT-TR, clearly demonstrating that our representation is more efficient and can handle larger models. The strongest advantages of our approach are summarized in the following list:

- the size of our meshes is constant while OCT-TR subdivide its meshes when layer resolution is increased. This more compact format is valuable when researchers need to share information during field works;
- the required time for the creation of our structures is insignificant in comparison with the creation time of the octree;
- our structures always reside in GPU;
- all the operations are computed in GPU;
- the data-transfers between CPU and GPU are close to zero;
- during the editing of the layers, the size of the editing tool does not affect the performance of the algorithm.

In conclusion, our architecture structures the information in thematic layers, uses 2D textures to store them and texture coordinates to index their information. Furthermore, it takes advantage of the inherent parallelism of GPUs to manage and operate these layers.

Acknowledgements

This research was funded by the Spanish Ministry of Science, Innovation and Universities (grants TIN2014-60965-R and TIN2017-85259-R), including FEDER funds from the European Union.

Thanks to Council of the Alhambra and Generalife, Stanford Computer Graphics Laboratory, Museo de Puebla de Don Fadrique, Museo Histórico Municipal de Écija and Centro Andaluz de Arqueología Ibérica for the 3D models used in this article.

Appendix A. Supplementary data

Supplementary data associated with this article can be found, in the online version, at: https://doi.org/10.1016/j.culher.2019.05.003.

References

[1] K.K. Naglič, Cultural Heritage Information System in The Republic of Slovenia, in: ARIADNE 5 Work, Doc. Interpret. Present. Publ. Cult. Herit. (2001) 1–9 (Prague).
[2] C. Ioannidis, C. Potsiou, S. Soile, An integrated spatial information system for the development of the archaeological site of mycenae, Int. Arch. Photogramm. Remote Sens. Spat. Inf. Sci. 34 (2003) 1–8.
[3] J.A. Parkinson, T.W. Plummer, R. Bose, A GIS-based approach to documenting large canid damage to bones, Palaeogeogr. Palaeoclimatol. Palaeoecol. 409 (2014) 57–71, http://dx.doi.org/10.1016/j.palaeo.2014.04.019.
[4] A. Benito-Calvo, S. Carvalho, A. Arroyo, T. Matsuzawa, I. de la Torre, First GIS analysis of modern stone tools used by wild chimpanzees (Pan troglodytes verus) in Bossou, Guinea, West Africa, PLoS One 10 (2015) e0121613.
[5] D.M. Campanaro, G. Landeschi, N. Dell'Unto, A.-M.L. Touati, 3D GIS for cultural heritage restoration: a "white box" workflow, J. Cult. Herit. 18 (2016) 321–332, http://dx.doi.org/10.1016/j.culher.2015.09.006.
[6] A. Durand, P. Drap, E. Meyer, P. Grussenmeyer, J.-P. Perrin, Intra-site level cultural heritage documentation: combination of survey, modeling and imagery data in a web information system, 2006 (CoRR. abs/cs/061).
[7] É. Meyer, P. Grussenmeyer, J.-P. Perrin, A. Durand, P. Drap, A web information system for the management and the dissemination of Cultural Heritage data, J. Cult. Herit. 8 (2007) 396–411.
[8] F.J. Mateos Redondo, L. Valdeón Menéndez, A. Rojo Álvarez, A. Armisén Fernández, B. García Fernández-Jardón, Plataforma virtual para el diseño, planificación, control, intervención y mantenimiento en el ámbito de la conservación del patrimonio histórico "PETROBIM", Constr. Pathol. Rehabil. Technol. Herit. Manag. (2016) (REHABEND).
[9] G. Giunta, E. Di Paola, B.M.V. Castiglione, L. Menci, Integrated 3D-database for diagnostics and documentation of Milan's cathedral façade, CIPA 2005 XX Int. Symp, Torino, Italy, 2005, pp. 1–9.
[10] S.P. Serna, R. Scopigno, M. Doerr, M. Theodoridou, C. Georgis, F. Ponchio, et al., 3D-centered media linking and semantic enrichment through integrated searching, browsing, viewing and annotating, Int. Symp. Virtual Reality, Archaeol. Cult. Heritage. 12 (2011) 89–96 (VAST).

[11] F.I. Apollonio, V. Basilissi, M. Callieri, M. Dellepiane, M. Gaiani, F. Ponchio, et al., A 3D-centered information system for the documentation of a complex restoration intervention, J. Cult. Herit. 29 (2018) 89–99, http://dx.doi.org/10.1016/j.culher.2017.07.010.

[12] J.C. Torres, P. Cano, J. Melero, M. España, J. Moreno, Aplicaciones de la digitalización 3D del patrimonio, Virtual Archaeol. Rev. 1 (2010) 51–54, http://dx.doi.org/10.4995/var.2010.4768.

[13] J.C. Torres, L. López, C. Romo, G. Arroyo, P. Cano, F. Lamolda, et al., Using a cultural heritage information system for the documentation of the restoration process, Digit. Herit. Int. Congr. 2013 (2013) 249–256 (DigitalHeritage).

[14] F. Soler, J.C. Torres, A.J. León, M.V. Luzón, Design of cultural heritage information systems based on information layers, J. Comput. Cult. Herit. 15 (1) (2013) 15–17, http://dx.doi.org/10.1145/2532630.2532631.

[15] F. Soler, F.J. Melero, M.V. Luzón, A complete 3D information system for cultural heritage documentation, J. Cult. Herit. 23 (2017) 49–57, http://dx.doi.org/10.1016/j.culher.2016.09.008.

[16] The Khronos Group Inc, Texture array, 2008.

[17] The Khronos Group Inc, Sparse texture, 2013.

[18] C. Everitt, T. Foley, J. McDonald, G. Sellers, Approaching zero driver overhead in OpenGL, Game Dev. Conf, San Francisco, CA, USA, 2014.

[19] The Khronos Group Inc, Bindless texture, 2013.

[20] Z.S. Hakura, A. Gupta, The design and analysis of a cache architecture for texture mapping, SIGARCH, Comput. Arch. News 25 (1997) 108–120, http://dx.doi.org/10.1145/384286.264152.

[21] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, A. Moshovos, Demystifying GPU microarchitecture through microbenchmarking 2010, IEEE Int. Symp. Perform. Anal. Syst. Softw. (2010) 235–246, http://dx.doi.org/10.1109/ISPASS. 2010.5452013.

[22] M. Doggett, Texture caches, IEEE Micro 32 (2012) 136–141, http://dx.doi.org/10.1109/MM. 2012.44.

[23] M. Segal, C. Korobkin, R. Van Widenfelt, J. Foran, P. Haeberli, Fast shadows and lighting effects using texture mapping, ACM Siggraph Comput. Graph. 26 (2) (1992) 249–252.

[24] C. Everitt, Projective texture mapping, white pap, 4, NVidia Corp, 2001.