Sprite tree: an efficient image-based representation for networked virtual environments

Minhui Zhu, Géraldine Morin, Vincent Charvillat, Wei Tsang Ooi

Sprite tree: an efficient image-based representation for networked virtual environments

Minhui Zhu[1] · **Géraldine Morin**[2] · **Vincent Charvillat**[2] · **Wei Tsang Ooi**[1]

Abstract We propose a new and efficient image-based representation for networked virtual environments, called the sprite tree. A sprite tree organizes multiple reference images efficiently and compactly for accelerating the rendering of complex virtual scenes. Using our basic construction and rendering methods, the results show that a sprite tree can efficiently organize the pixels from hundreds of distinctive reference images and accelerate the rendering of a complex scene. Furthermore, we propose the sprite view similarity to (i) largely reduce the lighting artifacts in the rendered images, and (ii) significantly reduce the redundancy and the tree size with little loss of the visual quality.

Keywords 3D image warping · Image-based rendering · Remote rendering · Networked virtual environments

1 Introduction

Networked virtual environments (NVEs) have several popular applications such as multiplayer online games (e.g. World of Warcraft[1], League of Legends[2]), social virtual worlds (e.g. Second Life[3]), and virtual heritage (e.g. Google Cultural Institute[4]). These modern NVEs have exhibited several challenging features. First, they are displaying more and more densely distributed 3D models. When these objects could have significantly complex geometry, the traditional visibility culling techniques (e.g. view frustum culling) cannot reduce the rendering complexity enough. For example, the Lucy statue[5] is scanned and modeled with more than 28 million polygons. Second, modern NVEs may allow users to create or alter the 3D models in real time, such as in Second Life.[6] It then requires a high bandwidth to transmit these models for real-time updates. For example, it requires 324.8 s to download the highly compressed 81.2 MB Thai Statue model[7] [42] with an average bandwidth of 2 Mbps. Third, modern NVEs may need to support thousands or millions of concurrent users. For example, as a massively multiplayer online game (MMOG), League of Legends reported over 7.5 million concurrent players during each day's peak play time.[8] Fourth, modern NVEs should be accessible by mobile devices. Compared with PC machines, ordinary mobile devices are usually equipped with small display screens and without high-end graphics hardware, run under a low bandwidth and with limited battery capacities. These devices may not be fully capable of rendering a virtual

✉ Minhui Zhu
minhui7zhu@gmail.com

Géraldine Morin
Morin@enseeiht.fr

Vincent Charvillat
Vincent.Charvillat@enseeiht.fr

Wei Tsang Ooi
ooiwt@comp.nus.edu.sg

[1] National University of Singapore, Singapore, Singapore

[2] University of Toulouse, Toulouse, France

[1] http://battle.net/wow/.

[2] http://www.leagueoflegends.com.

[3] http://secondlife.com.

[4] https://www.google.com/culturalinstitute/.

[5] http://graphics.stanford.edu/data/3Dscanrep/.

[6] http://secondlife.com/.

[7] http://graphics.stanford.edu/data/3Dscanrep/.

[8] http://www.riotgames.com/our-games.

scene in real time. It may also be expensive or time consuming to download the complex 3D models over wireless networks. Finally, some NVEs may be copyrighted and the downloading of 3D models in the scene is not allowed, especially since 3D printing became popular. A possible scenario is that a virtual museum may need to display the ancient objects in 3D without transferring their 3D forms to the viewers over the network.

The most common 3D representation is mesh based: in this context, 3D meshes are transferred to the client for local rendering with good visual quality and low interaction latency. But these meshes may require a high bandwidth for real-time downloading, such as the Thai Statue mentioned above. Even being downloaded, high-end graphics hardware is needed to render them with an acceptable frame rate. Such context is not adapted for light client, as bandwidth and rendering capacities may be insufficient. Additionally, it does not support copyrighted meshes. Another popular solution is the video-based approach. Relying on the server for rendering, encoding, and streaming each frame, the clients including the mobile devices explore the virtual scene like watching a video remotely without much effort other than decoding and displaying. It requires a reasonable bandwidth comparable to video streaming and suits the copyrighted 3D models. As for mesh-based approach, the workload between the server and the client is unbalanced: in a video-based approach, workload is pushed on the server. The NVEs may need to be deployed with powerful remote servers. The servers render for each client and download video frames that are usually not reusable on the client side.

We opted for a third, image-based, solution. It represents the 3D models as image samples, for example, a planar impostor [1] (also called a billboard) or image with its depth map (called a depth image). The complexity of image-based rendering algorithms [24,26] does not depend on the scene complexity and resolution, unlike rendering through a geometry-based rendering pipeline. For this reason, it is much cheaper for the clients to render with these image samples than the 3D models. This solution has similar advantages to the video-based solution, but the workload is balanced between the server and the client. It can easily adapt to the increasing number of 3D models in a scene without overloading either the server or the client for three reasons.

First, these image samples are reusable on the client side. They can also be shared among concurrent users exploring the same virtual scene, since the users may share the viewing content during their navigation.

Second, the clients, even the resource-constrained devices, can take a reasonable amount of work via image-based rendering techniques.

Third, the copyrighted and/or highly complex 3D models can be transmitted using image samples, while other 3D models in the scene can still be rendered with other techniques including the mesh-based and video-based solutions.

Despite these features, traditional image-based solutions do have their limitations in efficiency. They may need a large number of image samples [4,14] to show a complex virtual scene with acceptable visual quality, increasing the memory and bandwidth requirements for exploring the NVEs. It may also become computationally intensive if too many image samples have to be rendered at the same time. Therefore, we propose a new and efficient image-based representation, named the sprite tree, for the acceleration of rendering in modern NVEs. Specifically, a sprite [32] is a group of image pixels extracted from a depth image, and a sprite tree is an octree storing and organizing the sprites. The main intuition for proposing the sprite tree is to efficiently organize and utilize a number of image samples to accelerate the rendering of a complex virtual scene. One main application of the sprite tree is the following remote rendering system. The server maintains a sprite tree and streams the sprites to the clients upon requests. There is no need to constantly render or stream the geometry data for each client, and thus the server can support more clients. The clients, resource-constrained or not, can cache and reuse the sprites in a local sprite tree for the local rendering, reducing the interaction latency and server-side rendering workload.

Our main contributions are summarized as follows.

- We propose the sprite tree to significantly accelerate the rendering of static 3D models in a complex virtual scene. The sprite tree can efficiently organize the pixels from hundreds of distinctive reference images for the acceleration of rendering. The proposed view similarity criteria reduce the redundancy in the sprite tree by inserting only the distinctive reference images. Moreover, traditional visibility culling techniques are combined with the sprite tree.
- Further, the sprite view similarity measure avoids the lighting artifacts in rendered images, by considering both visibility and lighting conditions for selecting valid sprites for rendering. We also use this measure to further reduce redundant sprites in the sprite tree.

2 Related work

In this section, we first review classical image-based techniques and representations that use explicit geometry same as our work. Second, we review classical approaches for remote rendering.

2.1 Image-based rendering

Image-based rendering is a process of sampling and reconstruction of the world. But pure image-based representations usually require a huge number of image samples, such as Lumigraph [12] and the light field [19]. To make the data size manageable, a commonly used strategy is to introduce geometry. We now review several representations introduced with different types of geometry.

Maciel and Shirley [23] introduced the idea of planar impostors, also called billboard. A planar impostor is easy to generate and cheap to display, but due to the flatness, it causes visibility errors and can be easily detectable when the viewer moves around it. A way to tackle the visibility issue is to make image impostors view independent. Decoret et al. proposed billboard clouds [9]. They optimize image impostors and choose a set of representative planes given a geometric error threshold. But pre-processing such impostors takes long time, and a significant amount of texture memory is required for storage. Our sprites are similar to image impostors, but they are not rendered on well-chosen quadrilateral plane as a texture. It is extracted from a reference image in a straightforward way (see Sect. 3.2). Moreover, a sprite can be warped [24–26] to different views without suffering the visibility issue of planar impostors. Viewers are hence not constrained to certain viewing cells.

Shade et al. [33] employed image impostors in their representation called hierarchical image cache to accelerate walk-through in virtual environments. The image impostors are stored in a BSP tree, reused as long as they are valid according to an error metric, and updated once they become invalid. Schaufler et al. [31] proposed a similar image cache but with a different space partitioning tree, a kd tree. We also use a spatial partitioning tree. Only distinctive sprites are added to a sprite tree, and selection of sprites is enable for the visibility and lighting issues instead of updating. For this reason, sprites do not need frequent updates like the hierarchical image cache when the view changes rapidly.

Textured Depth Mesh (TDMs) can be seen as impostors augmented with 3D information, also called 3D impostors or meshed impostors [39]. It can be seen as a simplified mesh mapped with detailed scene appearance information from depth images. Although TDMs offer a better reconstruction and solve the visibility issue, it may have rendering artifacts called the rubber sheet effect [10]. To address such artifacts, multi-mesh impostors (MMI) [10] control the visibility errors by generating multiple layers of textured meshes with dynamic updates. To eliminate the visibility errors, Jeschke and Wimmer [16] present a new algorithm that generates a TDM with a special error bound metric. Wilson and Manocha [41] also minimized such errors by sampling the geometry incrementally, generating an incremental textured depth mesh (ITDMs). Generating a TDM is usually too slow to be done in real time, and thus TDMs are pre-processed. Ghiletiuc proposed a real-time creation of TDMs [11]. The TDM is able to present details like contours better than impostors and sprites. Unlike TDMs, our sprite tree does not rely on a mesh, that is, the underlying structure is not piecewise linear. But, because of this ad'hoc piecewise linear mapping, creating TDMs requires heavier processing than creating sprite trees. In addition, TDMs are used efficiently to model urban scenes, including mostly flat objects like buildings and streets. Objects like plants require numerous small polygons, and thus are difficult to be represented as meshes. By contrast, the sprite tree has no such constraints, as we shall see in Sect. 5 in particular on the San Miguel scene.

3D Image Warping [24–26] is widely adopted for image-based rendering systems [36,38,43]. This technique has occlusion errors [26]. To reduce these errors, multiple reference images [24] can be taken from different views and warped to the same target view. These reference images must be carefully chosen [3,14,24,37] so that mixing their warped views compensates the missing information in single frames. On top of using warping from multiple sprites, we filter sprites to be warped depending on views to ensure coherent lighting for the target view (see Sect. 4). Layered Depth Image (LDI) [32] offers an alternative way to reduce occlusion errors. LDI generalizes the depth images by associating each pixel location with possibly several depth pixels. It can still be warped like a normal depth image [29]. Meanwhile, it reduces artifacts due to single-layered depth image. But current hardware does not support the maintenance of multiple samples along the viewing ray per pixel. Moreover, the fixed resolution of the LDI is a limitation. To address this problem, Chang et al. [4] proposed the LDI tree. The LDI tree organizes the LDIs into an octree, and coarser versions of the pixel are also added to the parent octree nodes to get a multi-resolution representation. We also adopt 3D image warping and organize samples using an octree. Figure 1 illustrates the relations between a sprite in the sprite tree and an LDI in the LDI tree, by showing two sprites (9 pixels and 4 pixels respectively) and their corresponding LDIs. In an LDI, pixels are orthographically projected onto one side of its bounding box, choosing the axis-aligned plane with the closest orientation. Pixels projected to the same LDI grid location and also from the same surface are merged, eliminating redundant samples. Once projected, a pixel cannot be traced back to the view it originates from. As a comparison, sprites and their associated pixels are view dependent. We reduce redundancy from the source, that is, only dissimilar images or sprites are inserted (see Sects. 3.3 and 4.2).

The following disadvantages of the LDI and the LDI tree (mentioned by their authors) motivated us to propose the sprite tree for rendering complex 3D scenes. First, pixels undergo two resampling steps from the input image to output image, one at the construction stage and the other at the

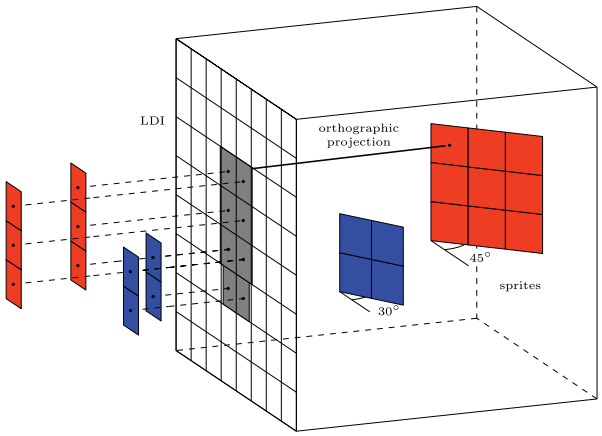

Fig. 1 An illustration of an LDI [4] and two sprites (used in our approach)

rendering stage. This potentially degrades the image quality. Qu et al. extended LDI to overcome this disadvantage with the help of O-buffer [30]. It records the position of pixels in LDI and delays the resampling at the construction stage to the rendering stage. It, however, involves storing the origin information for each pixel and applying a two-step warping algorithm. For our representation, we rather choose not to resample. Second, merging pixels at the same location only works for diffuse surfaces with little view-dependent variance. Lischinski et al. [21] proposed to use a second collection of LDIs for the view-dependent part of the scene. This method increases the complexity to create, maintain, and use the LDIs. We do not carry out such separation for sprites. Instead, we choose to handle this problem by selecting only qualified sprites for the target view (see Sect. 4.2).

2.2 Remote rendering

Based on what data are transmitted from the server to the client, we review existing work on remote rendering in terms of the following three approaches: (i) mesh-based approach; (ii) video-based approach; and (iii) image-based approach.

In mesh-based remote rendering, the server streams the 3D meshes to the clients for local rendering. Since the downloading time depends on the network bandwidth and the mesh size, large meshes are not well suited for interactive remote rendering. One solution is progressive meshes [13], which transmits a coarser mesh and progressively refines the geometry. Although the user can view a simple mesh almost immediately, progressive meshes have to be generated by preprocessing. In addition, progressive meshes are not suitable for scenes with dense and separate objects, since they are object-based representation. Anyhow, mesh-based approach assumes that the clients have enough rendering capability and the meshes are allowed for downloading, which are not always true.

In video-based approach, a high-end server renders the 3D models and streams the resulting videos to the remote clients [17]. It relies on server-side rendering and efficient video compression, making the scene visualization easy for mobile devices. Videos generated from a virtual scene can also benefit from additional information, for example, the information on the rendering process can be used to predict [7,27] or calculate [6] the motion vectors. On the other hand, it is not a scalable solution due to the heavy server-side rendering and the client-side interaction delay. In other words, the server can be easily overwhelmed if there are too many clients requesting at the same time.

Image-based approach can alleviate the burden of concurrent clients on the server side. Image-based rendering techniques reduce the complexity of 3D rendering by replacing parts of the geometry with images. The difficulty is to trade off between the rendering quality and the interaction latency. 3D warping [24–26] is often used in this approach. Specifically, the server renders the 3D model into depth images (called reference images), which are transmitted to the client for local warping. Light clients can be benefited significantly from this kind of remote rendering systems [5]. To reduce occlusion errors, Mark et al. [24] chose two reference images, respectively, near a previous and a future reference view. But, as they mentioned, occlusion errors cannot always be avoided by two reference images. Hudson and Mark [14] proposed to use three sets of reference images surrounding the user's view, warping twelve images per view, while some of the images may not be useful. Shi et al. [37] proposed to predict a future reference view to be used with a main one so that the warping errors can be compensated. The quality of the prediction, however, depends on the actual future camera motion. Instead of using reference views along the user path, Bouatouch et al. [3] proposed a camera placement algorithm to capture a relatively small set of reference images so that occlusion errors are avoided. Their approach suits street networks, but is difficult to generalize to other scenes. Different from the above work, we propose to store many reference images in terms of distinctive sprites in a sprite tree and select carefully only those sprites useful to the target view for warping; the sprite filtering both improves the rendering quality and limits the complexity of the warping computation. The reference views can come from the user movement or from previously well-chosen reference views [3,14,24,37]. As image samples accumulated in a sprite tree, there are many more images to choose from for fewer occlusion errors, compared to a few chosen reference images. Similar to the camera placement method [3], we also choose a relatively small set of images; however, we rely on the user movement history and our filter determines what image samples are actually stored.

3 Sprite tree

We now present the structure of sprite tree. Following that, we introduce the basic methods to build a sprite tree with controlled redundancy and to render with a sprite tree.

3.1 Structure

A sprite tree is a set of sprites stored in an octree. As explained in Sect. 2.1, image-based representation stored in hierarchical spatial partitioning tree has been proposed for rendering, for LDIs [4,32] and billboards [31,33]. In this section, we detail the structure of the proposed sprite tree.

The initial sprite tree is a preprocessed octree, used to subdivide the 3D space of the scene for frustum culling. It means that the root node of this octree has the bounding box of the scene. Only the leaf nodes will be inserted with sprites.

A sprite is a group of image pixels with depth rendered from the same view: each sprite corresponds to a view (rendering parameters are stored in the indexed view) and consists of an array of depth pixels. Each pixel stores its original index (i.e. coordinates in the image which it belongs to), color and depth, as shown in the following representation:

DepthPixel:		
	Index:	integer
	Color:	32 bit integer
	Depth:	float
Sprite:		
	View Index:	integer
	Pixels[N]:	array of DepthPixel

All the depth pixels in a sprite are located in the same leaf node, which is how they are grouped into one sprite (see Sect. 3.2). With the root bounding box determined by the scene, the depth of the octree determines how much scene information the sprites in the same leaf node could represent. An octree with more levels can lead to better accuracy of the frustum culling, but it also decomposes an image into smaller sprites, leading to more processing in the insertion and selection processes. In the experiments, we set the level of the sprite tree to be seven.

Sprites in the same leaf node are extracted from different reference images and thus have different views. Depending on the views, these sprites may be rendered with different amount of scene details for the geometry contained by this leaf node. To render the geometry in a particular leaf node from a target view, only sprites with a certain depth range are sufficient and hence required. This depth range should be comparable to the depth range of the leaf node perceived from the target view. For this reason, we store the depth ranges of sprites. There are two reasons why we prefer the depth range to the sampling resolution or density. First, the depth range can be directly acquired from the depth map. Second, it can be easily estimated for any leaf node visible to the target view for selecting suitable sprites. Furthermore, in order to avoid storing the depth range of each sprite and searching all of them, we adopted another method that assigns levels to sprites.

Specifically, sprites are distinguished by levels within each leaf node according to the minimum depth values of their pixels. Note that the depth is the actual linear depth in the viewing space not the inverse depth stored in the Z-buffer.

The viewing range $[Z_N, Z_F]$ defined by the near plane Z_N and the far plane Z_F of the camera is divided into a fixed number of uniform sub-ranges, each of which corresponds to a level. A sprite, once created, will be stored on the level whose sub-range its minimum depth value falls into. In this way, given a desired depth range, we can directly refer to the corresponding levels that cover this depth range to retrieve sprites. Besides the minimum depth values of a sprite, the maximum depth or the average depth could also be used for this purpose.

3.2 Inserting A reference image

We now explain how the sprites from a reference image are inserted into the sprite tree.

Before the insertion of a reference image I, an octree is built for the scene as the initial empty sprite tree (see Sect. 3.1). We project each pixel x in the image I into the 3D space by Eq. 1 presented by McMillan [26], where C is the camera position, P is the mapping matrix from image space to rays, and δ is the disparity.

$$X = C + Px\frac{1}{\delta}. \tag{1}$$

The sprite tree is traversed from the root to leaf nodes for each projected pixel. By checking whether the bounding box of the traversed node contains the 3D location X of pixel x, we find the leaf node o that the pixel x falls within. After all the pixels from image I are processed in the above way, those pixels falling into the same node are grouped into a sprite. The minimum depth of these pixels is also computed for assigning levels to sprites. We denote a sprite as $s(o, v)$, in which o indexes the leaf node and v is the view of image I. Then, we locate the level $l(o, v)$ for the sprite $s(o, v)$ with the minimum depth.

This insertion process performs two tasks, which decomposes the image into sprites and attaches these sprites to the octree leaf nodes they are located in. It would be possible to decompose images based on the bounding boxes of the objects, to shorten the traversal of the tree, or organize sprites by a different space partitioning tree such as a k-d tree. We

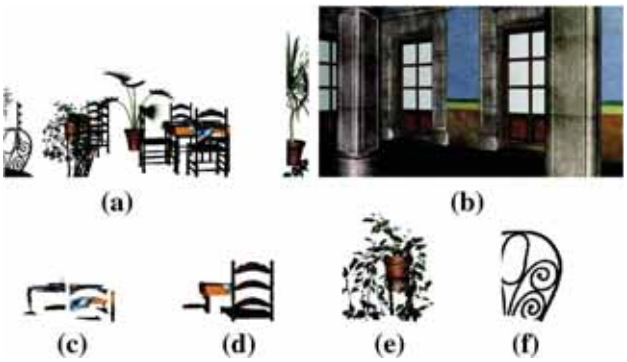

Fig. 2 An example of a reference view rendered into a background image and a foreground image. **a** Foreground image, **b** background image, **c–f** are four of the sprites created after inserting the foreground image into the sprite tree

choose the octree for its simplicity and decompose the images naturally using the octree subdivision.

Note that we separate the objects in the scene into two categories: (i) the background objects, such as the walls, windows, pillars, and the ground, which have simple geometry; and (ii) the foreground objects, such as the tables and plants, which have much complex geometry than the background objects. We only insert pixels from the foreground objects using the method described above. The reason is that these background objects can be rendered easily, but they may generate a large number of pixels. The cost of storing pixels from background objects is not worth the savings in rendering cost. After such separation for the San Miguel Model (see Sect. 5.1) used in the evaluations, the foreground objects contain about 90 % of the geometry in the whole scene. It means that 90 % of the geometry will be replaced with sprite tree for rendering. To separate the pixels from foreground and background objects, we render them separately into two depth images using multiple render buffers. The foreground image is used for insertion, and the two images are merged for display. With this setting, a sprite tree can be built by servers or capable clients without interrupting their rendering tasks.

Figure 2a, b shows an example of a reference view rendered into a foreground image and a background image. After inserting the foreground image into the sprite tree, four of the sprites created are shown in Fig. 2c–f. Note that these sprites are only displayed as images. As mentioned in Sect. 3.1, the pixels in any sprite are stored sequentially as an array.

3.3 Redundancy control in building sprite tree

Since any image rendered in the scene can be inserted into the sprite tree, it is necessary and critical to control and reduce the redundancy introduced by inserting images from similar views. For example, if the images are captured from a user trace, successive views along the trace are usually similar due to the spatial and temporal coherence; thus, inserting all of them would lead to a large amount of redundancy in the sprite tree. One straightforward method is to insert dissimilar images. We, therefore, propose a simple view similarity measure for this purpose. Note that, we will use the phrases "inserting view(s)" and "inserting image(s)" interchangeably from now on.

We define the view similarity by two criteria: (i) the distance $d(v_1, v_2)$ between the camera positions of two views v_1, v_2 is less than threshold ϵ; and, (ii) the angle $a(v_1, v_2)$ between their viewing directions is less than another threshold θ. In other words, two views are considered similar if they are spatially close and looking in similar directions. According to these two criteria, only dissimilar views are used for building the sprite tree. By adjusting θ and ϵ, we control the redundancy and the size of the sprite tree.

This view similarity measure is a simplistic measure that is cheap to compute, since it does not consider the scene complexity or the existing sprites. In Sect. 4.1, we will propose a more sophisticated way to insert reference images by considering the similarity between the sprites to be inserted and the existing sprites based on the sprite view similarity measure. For comparison in the evaluations, we will refer to the insertion method based on the view similarity as view-based insertion and the insertion method based on the sprite view similarity as sprite-based insertion.

3.4 Rendering with sprite tree

We now introduce how to render a target view with a sprite tree. Given a target view, the process starts with the frustum culling. From the root node, only those children nodes not culled will be traversed recursively until that the leaf nodes are reached. After reaching a leaf node, the minimum and maximum depth values of the node to the target view are computed through its bounding box. Each of the depth values corresponds to a level. Sprites on these two levels and the levels in-between are selected and considered adequate to represent the geometry inside the node for the target view. By comparing the levels first, we ensure that the pixels used for warping are originally rendered at a comparable depth, that means, a comparable resolution. These selected sprites can all be used to render the target view, but not all of them may be useful in terms of reconstruction. To avoid unnecessary warping, one basic decision can be made according to the visibility of a sprite as a planar billboard using the back-face culling technique.

Figure 3 shows an illustration of this technique. The plane represents a sprite $s(o, v)$ in the in-view node o, and it has a fixed orientation $\mathbf{n}$ which is the reverse viewing direction of its sprite view. We compute the product of the vector $\mathbf{n}$ and the vector $\mathbf{r}$ which points from the position of the target view v_t to the center of the node o. If the product $\mathbf{n} \cdot \mathbf{r} < 0$, the

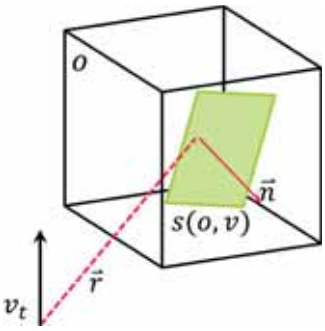

Fig. 3 An illustration for determining the visibility of a sprite in an in-view node

sprite is considered visible from the target view v_t, otherwise it is discarded. These visible sprites are send to the GPU for warping. The warping is done as proposed by McMillan [26] but is applied to sprites rather than images.

After warping, the warped sprites are merged with the background image rendered with the background geometry on GPU before display.

There is one possible drawback in the above rendering process, that is, the lighting conditions in different sprites are not considered for selecting adequate sprites. Warping sprites with different lighted colors to the same surface may lead to lighting artifacts due to the inconsistency. We, therefore, propose the sprite view similarity to consider both the visibility and the lighting condition. For comparison, we will refer to the above rendering method using the back-face culling as culling-based rendering and the improved method based on the sprite view similarity as similarity-based rendering. In the evaluations, we will show the lighting artifacts in the culling-based rendering and that the similarity-based rendering can largely reduce such artifacts.

3.5 Sprite tree for NVE

NVEs are displaying more and more 3D models that could have significantly complex geometry in the viewing frustum. When the traditional visibility culling techniques cannot reduce much geometry complexity, they can still work well with the sprite tree. More importantly, the storage, rendering cost and bandwidth consumption of using the sprite tree depend on the size of the sprites instead of the complexity or the number of 3D models in NVEs. Additionally, in case of real-time changes to the objects in NVEs, sprites can be added or removed conveniently from the sprite tree with only certain specific sprites related to the changes to be transmitted. While NVEs can have thousands or millions of concurrent users, a sprite tree can be shared among users and cached to prevent the server from rendering the same static scene repeatedly for the users. Mobile clients can also access NVEs using the sprite tree without the need of high-end graphics hardware or high bandwidth. Last but not least, NVEs with copyrighted 3D models can be rendered by letting the users download the sprite tree freely.

4 Similarity in sprite tree

An important factor for the performance of a sprite tree is the selection of sprites. During insertion of reference images, the view similarity criteria (see Sect. 3.3) detect when two views are similar. But there could be similar sprites from views considered dissimilar. Inserting views without further considering the similarity in sprites potentially increases redundancy in the sprite tree. This redundancy would increase both the number of sprites selected for reconstructing a target view and also the storage size required by the sprite tree. Similarly, selecting a sprite without considering the similarity between the sprite view and the target view may lead to inconsistent lighting. That is to say, results for the approach including the view-based insertion (see Sect. 3.3) and the culling-based rendering (see Sect. 3.4) can be further improved in terms of performance and quality.

Having introduced the basic work of the sprite tree in Sect. 3, we now present how to improve the efficiency of the sprite tree beyond the performance of regular rendering via a measure called the sprite view similarity. This measure takes into account both the visibility and the lighting conditions and allows two important improvements: (i) sprite-based insertion, we insert only the distinctive sprites rather than a whole reference image, based on the similarity between the to-be-inserted sprite and the inserted sprites; (ii) similarity-based rendering, we select the visible sprites whose lighting conditions are suitable for the target view for the rendering. We will highlight the performance increase of using the sprite view similarity measure in Sect. 5.

4.1 Modelling sprite view similarity

The sprite view similarity measures the quality of a sprite $s(v, o)$ for a target view v_t, and is denoted $SML(o, v, v_t)$. The sprite $s(v, o)$ is generated from view v and corresponds to the geometry in tree leaf node o. Whereas the view similarity criteria between v and v_t (Sect. 3.3) consider the two views generally regarding the whole scene, the sprite view similarity considers the two views regarding a node o only.

This measure is twofold: first, we consider the quantity of shared content (visibility), second, we consider the difference in appearance (lighting). A sprite captures the appearance of the 3D geometry in a node, while the complexity of the 3D geometry in the node is unknown. The performance would be unpredictable if we compute the similarity based on the actual geometry. We could compute the simplification of the geometry based on the actual model and material, but this

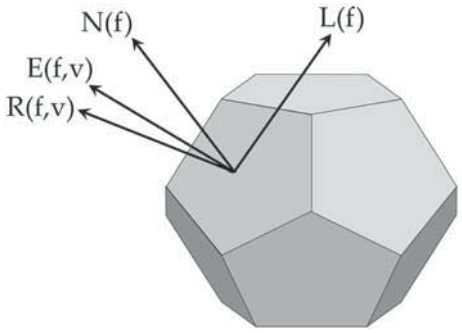

Fig. 4 Vectors that affect surface lighting

implies to store and maintain the simplified geometry and material data along with the sprite tree. We propose instead to consider a canonical representation, which models the geometry in o and possible surface orientations. We could take a sphere included in the cell, but, in order to limit the computational complexity, we quantify the normals by considering a dodecahedron instead of a sphere. Among the five regular polyhedrons candidates, the dodecahedron offers a better trade off between the accuracy and computational cost.

For assessing the difference in visibility: we first set the visibility $F(f, v) = 1$ for each face f, if the center point of face f is visible from the view v; otherwise, $F(f, v) = 0$. Similarly, $F(f, v, v_t) = 1$ if the center point of face f is visible from both views v and v_t. Note that $F(f, v)$ and $F(f, v, v_t)$ are the same visibility function that can take different number of views as input variables.

The appearance of the geometry in o depends on the view because of the lighting. Lighting does affect pixel colors: when sampling the same surface with the same amount of pixels from two different views, the color of the surface might be different enough and cause warping artifacts (see Sect. 5.2). For each node, its sole dodecahedron is always placed in the center with a fixed radius and a fixed orientation, which is independent of the views. As such, we can quantify the difference in appearance of a node from two views via the dodecahedron. Let f be a face index, where $f = 0, 1, \cdots, 11$. The three vectors necessary for the computations are $N(f)$, the unit outwards normal vector of the face f, $E(f, v)$, the unit vector pointing from the center point of the face f to the viewing position of the view v, and $L(f)$, the light direction from the center point of the face f to the light source (Fig. 4). The illumination $I(f, v)$ at the center point of face f as perceived from v is computed choosing a lighting model. For example, taking the Phong model [28]

$$R(f, v) = 2 \cdot \big(N(f) \cdot L(f)\big) \cdot N(f) - L(f), \tag{2a}$$

$$I(f, v) = I_A + I_D \cdot \big(N(f) \cdot L(f)\big) + I_S \cdot \max\big\{R(f, v) \cdot E(f, v), 0\big\}^{\alpha} \tag{2b}$$

Table 1 Examples for the setting of threshold σ

$\sum_{f=0}^{11} F(f, v_t)$	1	2	3	4	5	6
σ example I	1	2	3	4	5	6
σ example II	1	2	3	3	4	5

where $R(f, v)$ is the computed reflection ray, I_A is the ambient intensity, I_D is the diffuse intensity, I_S is the specular intensity, and α is the shininess. Similarly, we also compute the illumination $I(f, v_t)$ at the face f from v_t. Note that there is no need to consider the view-independent intensity in this computation of the illumination, that is, neither the ambient component, nor the diffuse intensity when it is not affected by the light setting such as a headlight.

Finally, we determine a binary sprite view similarity measure such that $SML(o, v, v_t) = 1$ if the following criteria are both satisfied (otherwise, $SML(o, v, v_t) = 0$):

$$\sum_{f=0}^{11} F(f, v, v_t) \geq \sigma \tag{3}$$

$$|I(f, v) - I(f, v_t)| < \delta, \text{ for all } f \text{ with } F(f, v, v_t) = 1 \tag{4}$$

Criterion (3) insures the similarity of visible geometry, by requiring a minimal number of shared visible faces. Note that, for dodecahedron, the maximum number of visible faces is six . Table 1 shows two examples of σ setting: example I ensures a strict constraint, for which all faces visible from the sprite view v_t also need to be visible from v; example II is a little looser, which is also used in our experiments. Criterion (4) bounds the lighting intensity difference of the shared dodecahedron faces from the two views.

Depending on the above criteria and their thresholds, it is possible that certain useful sprites could be ignored, which may lead to certain objects missing from the resulting image. Since the thresholds can be adjusted to allow sprites with more appearance or visibility difference to be selected, such artifacts of missing objects can also be adjusted. The corresponding experiment results are shown in Sect. 5.2.

4.2 Applications to the rendering and insertion

We now introduce how to apply the sprite view similarity measure for the insertion and the rendering.

Sprite-based insertion For a reference image rendered at view v_r to be inserted into the sprite tree S, the sprite tree S is traversed for this view v_r. We denote the set of visible leaf nodes from view v_r as $G(v_r)$. For each node $o \in G(v_r)$, the minimum and maximum depth are computed using its bounding box for finding the level $l(o, v_r)$ where the sprite $s(o, v_r)$ will be stored. Among the set $S(o, v_r)$ of sprites at level

$l(o, v_r)$ in node o, we search for a sprite $s(o, v) \in S(o, v_r)$ that is considered qualified to reconstruct the sprite view $v_r(o)$, i.e. the sprite view similarity $SML(o, v, v_r) = 1$. If at least one of such sprite $s(o, v)$ is found, we say that the sprite $s(o, v_r)$ is redundant to an existing sprite in the sprite tree S and thus unnecessary to be inserted. We collect such nodes in the set $G(v_r, S) = \{o | o \in G(v_r), \exists s(o, v) \in S(o, v_r), SML(o, v, v_r) = 1\}$ for which the corresponding sprite $s(o, v_r)$ is redundant; no sprite from v_r will be inserted in nodes in $G(v_r, S)$. The other sprites from view v_r for nodes in the set $G(v_r) - G(v_r, S)$ are inserted. Figure 5 illustrates the above insertion process. There are two views v_1 and v_2 in the figure, each having two sprites (one sprite for the red sphere and the other for the wireframe sphere). If using the view similarity criteria (see Sect. 3.3), the two views are dissimilar and all their sprites will be inserted into the sprite tree as described in Sect. 3.2. But there could be redundant sprites. For example, the sprites of the wireframe sphere in both views may be similar enough that one can reconstruct the other. To insert only the distinctive sprites and avoid the redundancy, we propose to use our sprite view similarity criteria. After the sprites $s(o_1, v_1)$ and $s(o_2, v_1)$ of the two spheres from the view v_1 are inserted, we can identify that the sprite $s(o_2, v_2)$ of the wireframe sphere from v_2 is similar and thus redundant to the inserted sprite $s(o_2, v_1)$ of the same sphere according to the sprite view similarity ($SML(o_2, v_2, v_1) = 1$). As a result, the sprite $s(o_2, v_2)$ is not inserted. By contrast, the sprite $s(o_1, v_2)$ of the red sphere is not similar to the inserted sprite $s(o_1, v_1)$ and is thus inserted. As such, three sprites are inserted using the sprite view similarity measure, less than using the view similarity criteria. In this way, images can be inserted without considering their view similarities. Meanwhile, sprites are inserted greedily as long as they are not redundant. Since only redundant sprites are reduced, the sprites inserted earlier and capable of reconstructing these redundant sprites are still preserved in the sprite tree. Therefore, the visual quality of the output image is not affected significantly, while the sprite tree is smaller.

Similarity-Based Rendering The rendering process is the same as that in Sect. 3.4, except that we are replacing the back-face culling with the sprite view similarity check. In other words, instead of checking visibility based on the traditional technique, we measure the sprite view similarity between the sprite view and the target view. If they are similar, the sprite is selected for reconstructing the target view. Figure 6 shows an example of rendering a target view with the sprite tree. After locating the octree leaf nodes visible in the target view, qualified sprites are selected in each leaf node. Only five leaf nodes are shown in this example, and their approximate projected areas on the target image plane are plotted as five red rectangles layered on the image. The available sprites in these nodes are selectively shown in the five gray dashed rectangles. In this case, one qualified sprite for each of these nodes is selected (marked by red edges instead of gray) and warped (represented by red arrows pointing from the selected sprite to the corresponding projected area) to the target view.

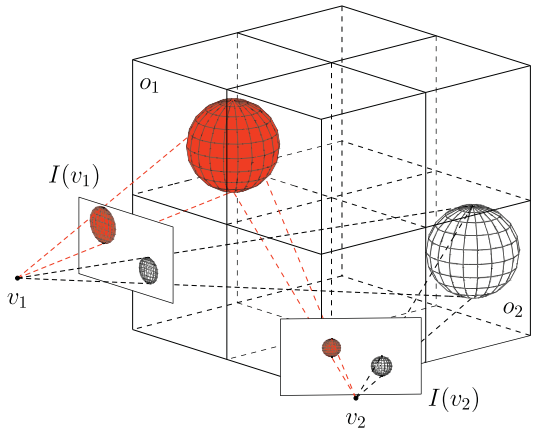

Fig. 5 A scene of two spheres, with two views generating four sprites

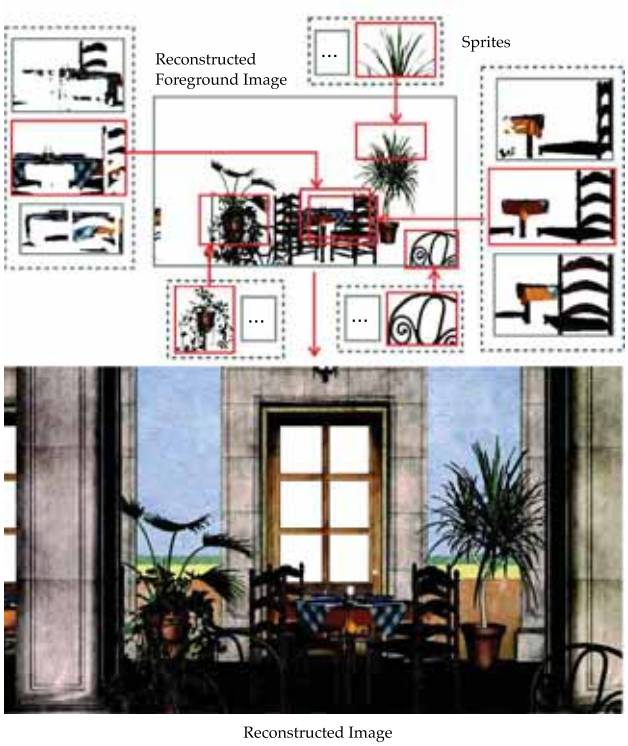

Fig. 6 An example of rendering a target view with the sprite tree. The *top image* shows five of the sprites for the foreground as an example, selected (*on the side*) and warped (*in the middle*) for rendering the middle image with our method

4.3 Similarity in NVEs

According to the mobility characteristics [20,34] in NVEs, there are certain popular areas that users often visit. Even for

areas not so popular, it is likely that multiple users may view the same part of the scene. Considering the huge number of concurrent users in NVEs, it is highly likely that high redundancy would exist in sprites without any proper precaution. Such redundancy in sprites would then unnecessarily consume computation and network resources when using the sprite tree for NVEs. The sprite view similarity measure that we propose for the sprite tree will not only (i) prevent such redundancy from accumulating in the sprite tree but also (ii) make sure the similarity in NVEs can be used to improve the quality of the reconstructed image by identifying the more qualified sprites.

5 Evaluation

5.1 Experiment setup

We now introduce the data sets used in our evaluation, including the virtual scene and the user traces mentioned above.

Virtual scene We use a 3D virtual scene *San Miguel*, modeled by Guillermo M. Leal Llaguno of Evolución Visual, Mexico. The scene has 2.5 million unique polygons and is rendered with 10.7 million polygons using the object instancing technique. We render this scene with a 60° vertical field of view at a resolution of 1280 × 720. A headlight is used with the camera to light the scene. If placing a camera in the courtyard, many objects in the viewing direction are densely distributed within the view frustum, meaning the geometry complexity is high even after frustum culling. Note that there are no transparent objects or common graphics effects like shadows in the rendering of this virtual scene. Based on some existing techniques [8,15,22], we could apply these effects to our rendering system in the future.

User trace We use synthetic user traces in our evaluation. They are generated using human mobility models that contain the mobility characteristics [20,34] (e.g. distributions of flight length and pause time, popularity of areas) in virtual scenes, such as the SLAW [18] mobility model and the SAMOVAR [35] mobility model. We used the SLAW model to generate synthetic user traces. The generation code is available from the authors[9] and BonnMotion.[10] In the SLAW model, there are several parameters that can be adjusted, such as the size of the scene area, the number of waypoints, the minimum and maximum pause time of the user, the duration of the trace, and the number of simultaneous users. We adjust only these five parameters in the SLAW model according to the scene San Miguel. Specifically, we make the following settings: (i) we restrict the user movements to the courtyard because there are much fewer objects outside that area, and thus the size of the scene area is set to the size of the courtyard; (ii) the number of waypoints is set to 100, and Fig. 7 shows an example of 100 waypoints generated by SLAW for the courtyard area of the scene San Miguel; (iii) the minimum and maximum pause time are set to 10 seconds and 60 seconds respectively; and, (iv) the duration of the trace is set to one hour, and the user's speed is fixed to 1 m/s. As an example, Fig. 7 shows one user trace simulated with the above setting.

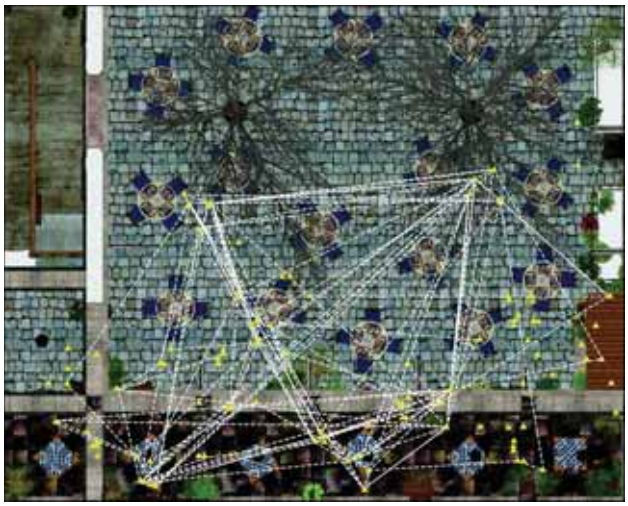

Fig. 7 A waypoint map with 100 waypoints (*yellow triangles*) generated by the SLAW mobility model, shown from the top view of the courtyard in the model San Miguel. The *white dashed line* shows the trajectory of one simulated user

Reference-based visual quality assessment The visual quality of the reconstructed views is an important performance measure in our evaluations. Traditional image quality metrics such as PSNR and SSIM [40] are not suitable for assessing reconstructed images by 3D image warping [2]. We propose to assess the visual quality by quantifying how many pixels are reconstructed for a target view. This approach requires to compare the reconstructed image with the ground truth image rendered with geometry. We simply count how many pixels in the ground truth image are matching in the reconstructed image after rendering a target view v_t with sprite tree S, that is to say, we count in the reconstructed image, the number $W(v_t, S)$ of pixels whose depth values match the depth values of their corresponding pixels in the ground truth image. For normalization, this number is divided by the total number $Z(v_t)$ of pixels in the ground truth image. This computation of reference-based visual quality for reconstructing a target view v_t with sprite tree S is

$$Q(v_t, S) = \frac{W(v_t, S)}{Z(v_t)}. \tag{5}$$

[9] http://research.csc.ncsu.edu/netsrv/?q=content/human-mobility-models-download-tlw-slaw.

[10] http://sys.cs.uos.de/bonnmotion/

Table 2 Performance measurements of the rendering methods for the sprite tree and their relative differences

Rendering method	N_{warped}	T_{frame} (ms)	Q_{visual}
Culling-based (A)	3, 765, 586	44.30	0.81
Similarity-based (B)	1, 317, 264	38.88	0.76
Relative difference ($\frac{A-B}{A}$) (%)	65	12	6

N_{warped} is the average number of pixels warped for one target view; T_{frame} is the average frame time; and, Q_{visual} is the average visual quality

We will call this assessment Q_{visual} in evaluations.

Note that for the similarity-based rendering and sprite-based insertion in the following evaluations, threshold $\delta = 0.15$ and the setting example II of threshold σ in Table 1 are used for the sprite view similarity measure.

5.2 Similarity-based *vs.* culling-based rendering

We first compare the two rendering methods of the sprite tree: similarity-based (in Sect. 4.2) and culling-based (in Sect. 3.3). To compare the two methods, we render 210 dissimilar target views selected from a five-users mobility simulation, with the same sprite tree built using the view-based insertion method with view similarity thresholds $\epsilon = 2$ and $\theta = 20$ (see Sect. 3.3). The visual quality of each reconstructed image is compared to its ground truth image rendered with 3D geometry as the reference (i.e. the reference-based approach, see Sect. 5.1).

Table 2 shows the average number N_{warped} of warped pixels for one target view, the average frame time T_{frame}, and the average visual quality Q_{visual} of the two rendering methods. It also highlights the relative difference between the measurements of the two methods. As shown, N_{warped} drops significantly from 3.7 million to 1.3 million when using the similarity-based rendering method, which is 65 % fewer, since this method additionally considers the lighting conditions for the selection of qualified sprites. The rendering time with the similarity-based rendering is thus 12 % less. By contrast, the average frame time measured for rendering the target views with geometry (the ground truth) is 98.70 ms, which is more than twice the frame time of rendering with the sprite tree even using the culling-based rendering. The reduction of qualified sprites also leads to a slight decrease (about 6 %) in the average visual quality Q_{visual}, which can be explained with Fig. 8.

Figure 8 shows one of the target views reconstructed by the two rendering methods. The visual quality Q_{visual} of the two rendering methods is both high (around 0.8) and similar, but the rendering of objects is different. In Fig. 8b, the image reconstructed by the similarity-based rendering method has higher fidelity to the ground truth image (Fig. 8c), which can be seen from two details including the ivy leaves on the wall and the three chair backs around the table. In general, the rendered objects in Fig. 8b have no noticeable lighting artifacts when compared to those in Fig. 8a, as the result of avoiding the sprites with unsuitable lighting conditions.

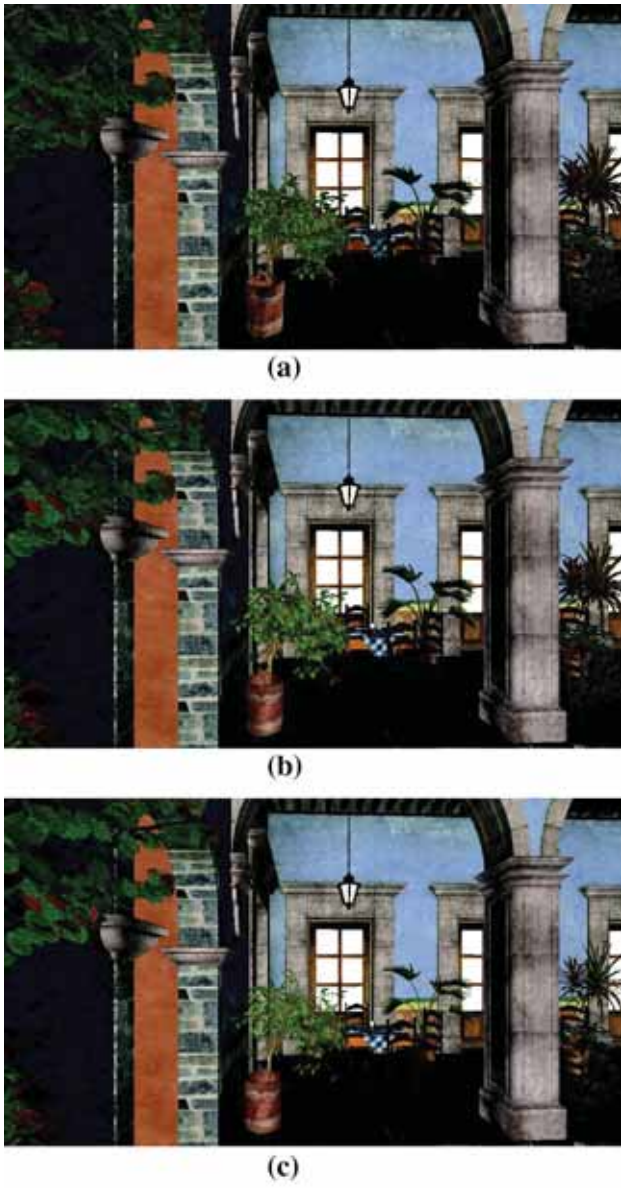

Fig. 8 An example of a reconstructed view using the culling-based rendering and the similarity-based rendering with the same sprite tree. **a** Culling-based rendering, $Q_{\text{visual}} = 0.84$, **b** similarity-based rendering, $Q_{\text{visual}} = 0.88$, **c** ground truth

Therefore, the similarity-based rendering method achieves better visual quality in terms of lighting conditions and is also flexible for adjustment.

5.3 Sprite-based vs. view-based insertion

We now compare the two insertion methods. For the comparison, we first build two sprite trees with the sprite-based insertion and view-based insertion, respectively. The reference images used to build the trees are from the same synthetic user traces that are generated by the SLAW mobility model [18], simulating five users (see Sect. 5.1). Second, we render the same 210 target views (see Sect. 5.2) with the two sprite trees using the similarity-based rendering method. The size of the two sprite trees is compared in terms of the total number N_{total} of pixels in sprites. Their rendering performance is also measured and averaged over the target views, including the average number N_{warped} of pixels warped for one target view, the average frame time T_{frame}, and the average reference-based visual quality Q_{visual}.

Figure 9 shows that the tree size (in pixels) grows slower when using the sprite-based insertion method, as more and more reference images are inserted. Moreover, the final tree using the sprite-based insertion method contains only about 50 million pixels in total, which is much smaller than the 70 million pixels in the other tree. The reason is that the sprite-based method ignores the redundant sprites while the view-based method inserts the dissimilar images without considering the redundancy of sprites among them.

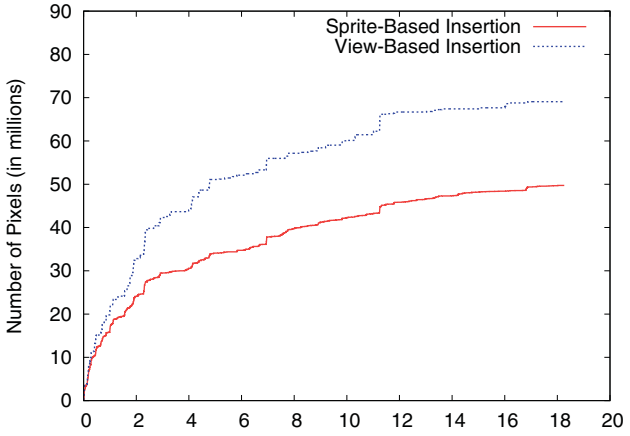

Fig. 9 The sprite tree size (i.e. the total number of pixels)

Furthermore, as shown in Table 3, the number N_{warped} of pixels warped is also 28 % smaller when using the sprite-based insertion method, since the redundancy among these pixels (or their sprites) is reduced as well. Only about 0.64 million pixels (fewer than the number of pixels in a 1280 × 720 image) are selected and warped, which is 52 % fewer compared to the 1.32 million pixels (almost two 1280 × 720 images) used by the view-based insertion. The resulting frame time also decreases from 38.88 to 37.76 ms.

A smaller sprite tree is easier to maintain in memory or to transfer over the network. With a more compact sprite tree, the sprite-based insertion method still leads to an average visual quality (0.72) close to the quality (0.76) of using the view-based insertion method. An example of a reconstructed view is shown in Fig. 10, with similar visual quality achieved: 0.83 (sprite-based) and 0.88 (view-based). It means that the sprite-based insertion method reduces the size and the redundancy of the sprite tree significantly with only a small decrease in visual quality.

5.4 LDI tree vs. sprite tree

We now compare the LDI tree to our sprite tree, using the sprite-based insertion and similarity-based rendering methods based on the sprite view similarity measure. The comparisons are in terms of rendering quality and tree size. For the visual comparison of rendering quality, we insert the same images corresponding to three test views into the two empty trees, as shown in Fig. 11a. The three test views are looking at the same table and its chairs from the left (Fig. 11b), the right (Fig. 11c), and the middle side (Fig. 12a), respectively. They show different lighting conditions of the rendered objects, e.g. the chair backs and the yellow tablecloth.

We then compare between the reconstructed images of the middle test view rendered by the sprite tree and the LDI tree. Ideally, since the image of the middle view has been inserted, both trees are expected to fully reconstruct and render the middle view. As expected, the resulting image (Fig. 12b) of the sprite tree is highly similar to the ground truth image (Fig. 12a), but the resulting image (Fig. 12c) of the LDI tree is not. In the image (Fig. 12c) of the LDI tree,

Table 3 Performance measurements of the insertion methods for the sprite tree and their percentage differences

Insertion method	N_{total}	N_{warped}	T_{frame} (ms)	Q_{visual}
View-based (A)	69,070,185	1,317,264	38.88	0.76
Sprite-based (B)	49,724,435	638,775	37.76	0.72
Percentage difference ($\frac{(A-B)}{A}$) (%)	28	52	3	6

N_{total} is the number of total pixels in the sprite tree; N_{warped} is the average number of pixels warped for one target view; T_{frame} is the average frame time; and Q_{visual} is the average visual quality

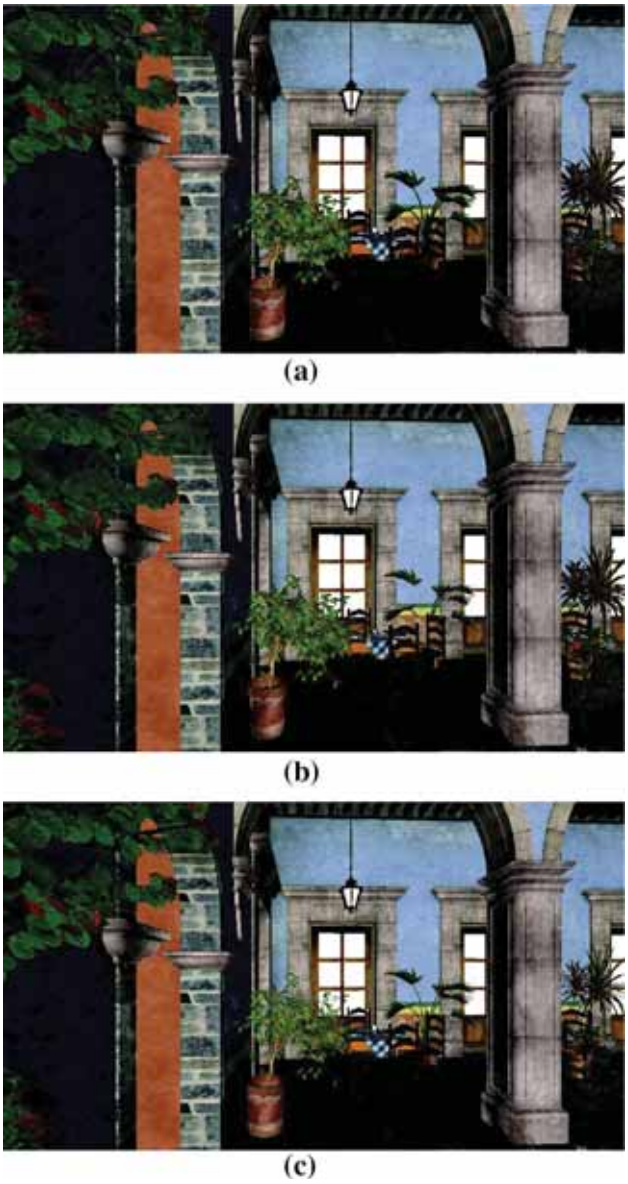

Fig. 10 An example of a reconstructed view by the similarity-based rendering with the sprite trees using different insertion methods. **a** View-based insertion, $Q_{\text{visual}} = 0.88$, **b** sprite-based insertion, $Q_{\text{visual}} = 0.83$, **c** ground truth

darker and lighter colors are inconsistently shown on the chair backs and the tablecloth. Obviously, besides the image samples from the middle view, samples from the other two test views are also warped and mixed together in the resulting image of the LDI tree, leading to these noticeable lighting artifacts.

Therefore, we can better select the suitable image samples for the rendering using the sprite tree. When both the suitable and unsuitable image samples are available, it is critical to identify and select only the suitable ones. LDI tree is a pixel-based representation. Pixels are stored independently in an LDI without the information of their original views. Without such information, the LDI tree cannot identify the suitable image samples similar to the target views. In contrast, the sprite tree preserves and uses the view information of image samples (inserted as the sprites) for the consideration of lighting conditions. As the result, the image samples from the left and the right test views can be avoided based on the sprite view similarity measure. In this way, only the samples from the middle view are correctly selected for the rendering.

Furthermore, the construction cost of a sprite tree is more sustainable than that of an LDI tree, because: (i) the construction cost of an LDI tree may depend on which render pipeline is used, for example, an LDI can be constructed directly from a ray tracer for synthetic scenes [4,32] but requires an additional pixel resampling process if using the traditional rasterization pipeline; (ii) the sprites are simply the depth pixels from the rendered images, regardless of the underlying rendering pipeline; and (iii) the main computation cost of building a sprite tree is to assess the sprite similarity, which is easily controlled by various similarity parameters and thresholds.

Although the sprite similarity also takes additional computational cost during the rendering, the cost is linear to the number of sprites in all the octree leaf nodes within the target view. Let $S(o)$ be the set of sprites stored in the leaf node o within the target view v_t, and let $O(v_t)$ be the set of octree leaf nodes within v_t. The total calculations of the sprite similarity based on illumination difference (see Sect. 4.1) are linear to $\sum_{o \in O(v_t)} |S(o)|$. The maximum number of in-view nodes $|O(v_t)|$ is constant if the camera setting and the octree setting stay unchanged. We also largely reduce the number $|S(o)|$ of sprites in each leaf node o by inserting only the dissimilar sprites. In our simulation using the San Miguel scene with dense objects, there are only about one thousand leaf nodes in total in an octree of seven levels and about 26 sprites on average in each leaf node. In summary, the computational cost to identify the similarities among sprites is also manageable.

Finally for the comparison of the tree size, Fig. 13 shows the total number of pixels in the sprite tree and the LDI tree, with the same set of reference images inserted. The size of the sprite tree grows to 30 million pixels, while the size of the LDI tree grows to 52 million pixels that is approximately 1.7 times larger. The reasons are that the LDI tree maintains the pixel samples of LDIs with multiple sampling rates. These sampling rates correspond to the octree levels. The traversal needs to reach only the octree level with the sampling rate comparable to the target view. For each pixel to be inserted to an LDI in an octree node, it will also be inserted to the parent LDI in the parent octree node. Such insertion leads to a huge LDI tree. In contrast, the sprite tree maintains the pixels in sprites with only their original sampling rate.

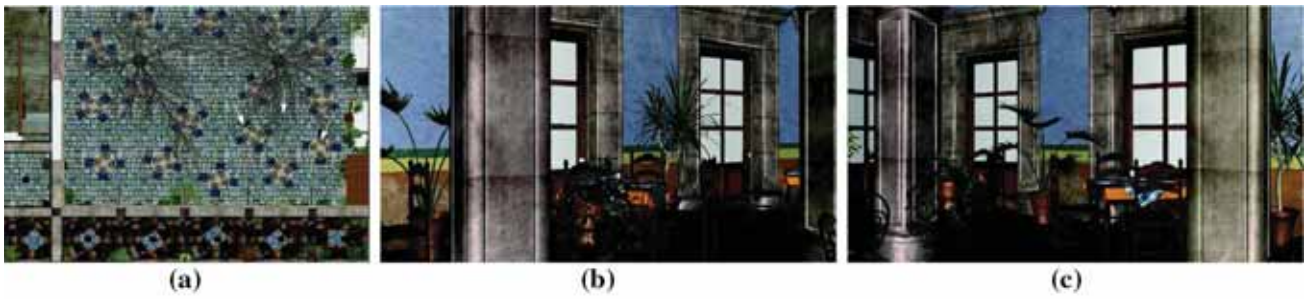

Fig. 11 Reference views for evaluation: **a** a *top view* of the scene with *three white arrows* indicating the reference views, **b** *left view*, and **c** *right view*. The *middle view* is shown in Fig. 12a

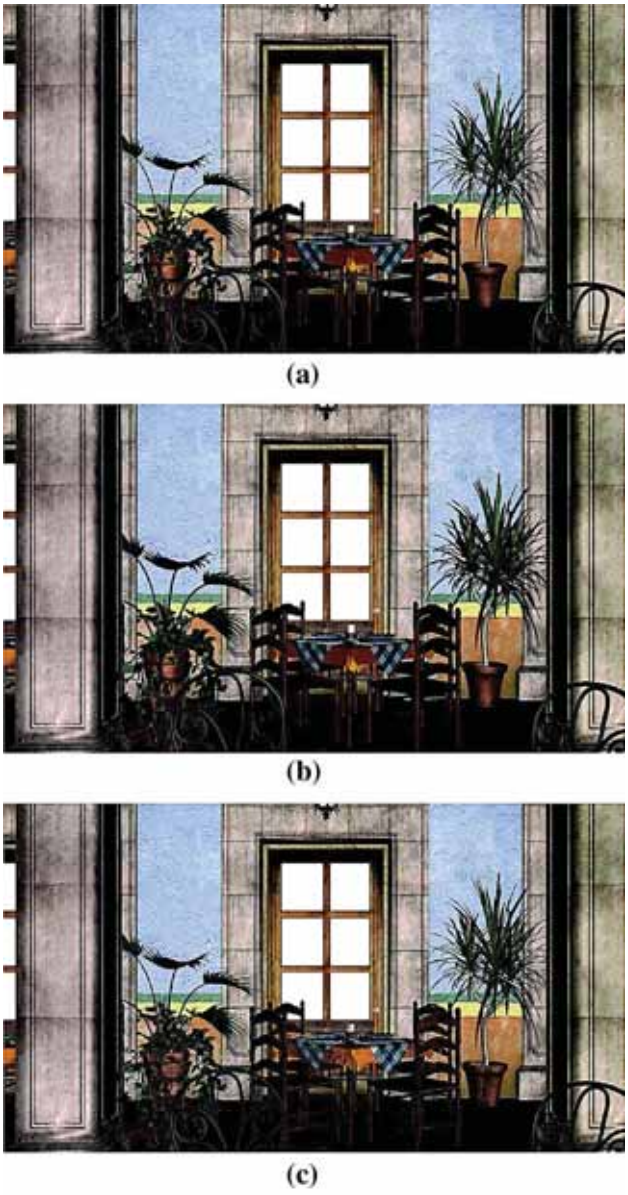

Fig. 12 Rendered images of the middle testing view as indicated in Fig. 11a: **a** ground truth image, **b** rendered with the sprite tree, and **c** rendered with the LDI tree

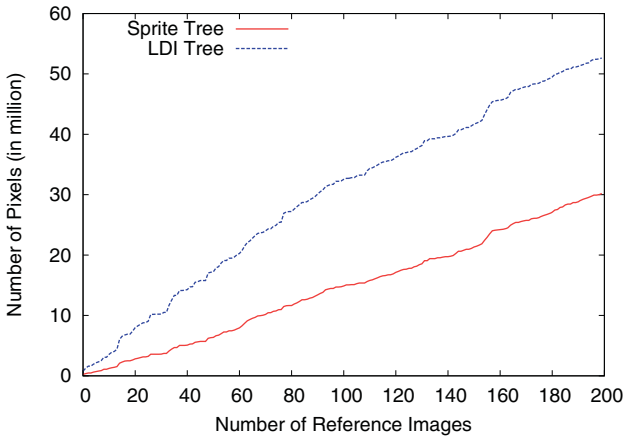

Fig. 13 The total number of pixels in the sprite tree and the LDI tree

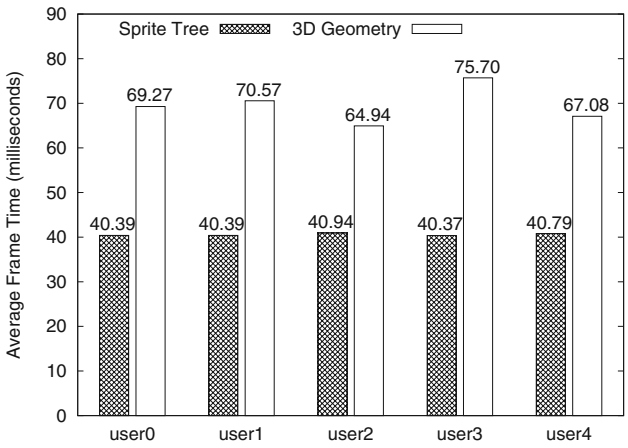

Fig. 14 Comparison of the frame time between rendering with the sprite tree and rendering with the 3D geometry

5.5 Comparison with the geometry-based rendering

We now compare the rendering with the sprite tree and the rendering with the 3D geometry. Two sets of synthetic user traces are used, each of which simulates five users and contains about 18,000 views. One set is used to build a sprite tree with the sprite-based insertion method and the other set

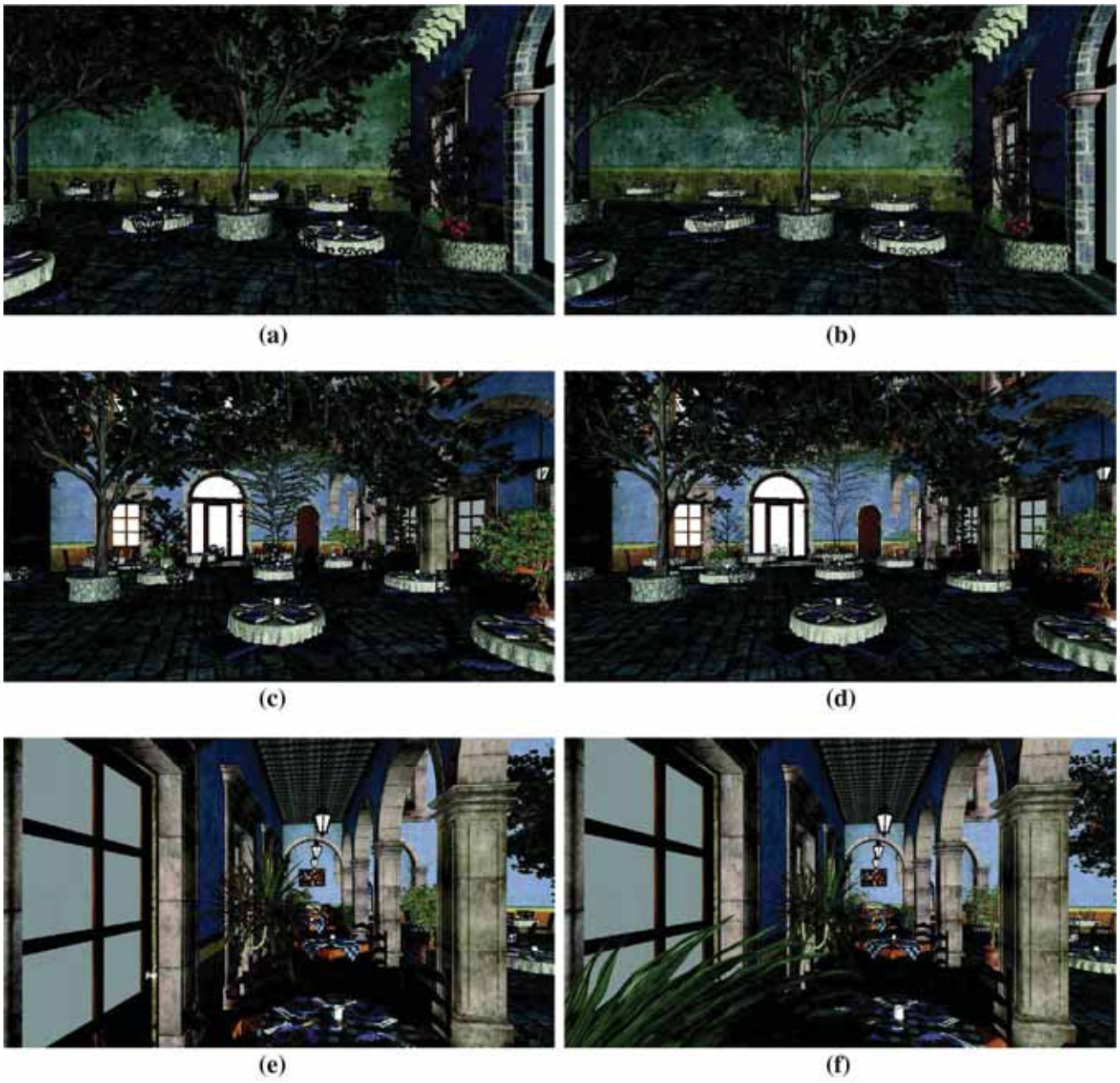

Fig. 15 **a**, **c** are examples of rendered images in acceptable quality using the sprite tree. **e** is an example with more rendering errors. The *right column* is the ground truth images rendered with the geometry

is used as the target views of the target users. We render these target views with the sprite tree and with the geometry of the virtual scene (as the ground truth). The similarity-based rendering method is used with the sprite tree.

Figure 14 plots the average frame time of the two rendering methods for each target user. We can see that rendering with the sprite tree requires about 40 ms on average and is about 30 ms faster than rendering with the geometry. Moreover, the frame time of using the sprite tree among different target users is not as fluctuating as that of geometry, because the complexity of rendering with the sprite tree does not depend on the geometry complexity as perceived in the target views. Additionally, the average visual quality Q_{visual} measured is 0.82, which is acceptable based on our observations.

Figure 15a, c shows two representative frames in acceptable quality, as the examples of the rendered images with the sprite tree. As shown, the rendered images are highly similar to the ground truth images with two exceptions. First, the rendered objects' silhouettes are thicker and with more edge aliasing, for example, the plants in the flowerpot in Fig. 15a. This is caused by the splatting after a pixel is warped to the resulting image. We use a simple splatting method by

updating four pixel positions closest to the position of the warped pixel. In other words, the influence area of any warped pixel is a 2×2 pixel area. Despite this problem, splatting is an effective technique to resolve the visibility issues in 3D image warping, although it cannot resolve all of them and also affects the smoothness of lighted colors on a surface. Without splatting, there would be more holes in the resulting image. The LDI tree also adopts splatting. Second, the table at the bottom left in Fig. 15a has incorrect lighting conditions. This is a compromise made to avoid missing rendered objects by lowering the sprite view similarity criteria (see Sect. 4.1) in the similarity-based method. Because the disappeared objects may be more noticeable than the lighting artifacts in terms of rendering errors. As such, the sprites with unsuitable lighting conditions are also used, if no better sprite is available in the sprite tree. As a comparison, Fig. 15e shows a rendered image with more rendering errors. Compared with the ground truth image (Fig. 15f), it is obvious that there is a part of plant leaf not rendered in Fig. 15e. The reason is that there is no sprite qualified for rendering the missing leaf in this target view. It is possible that a low-resolution sprite is available, but it is not considered qualified to deliver satisfying resolution and may lead to many small holes. In Fig. 15e, the missing plant leaf does not hurt the harmony in the image.

There is more to discover about the above-mentioned errors. Besides the close-up views of objects, many errors happen to those objects near the walls of the courtyard, according to observations during the experiments. Those spots are less visited than the center of the courtyard by the simulated user traces; therefore, less sprites are saved, leading to the higher probability that qualified sprites could not be found in the sprite tree. As a comparison, acceptable quality is achieved for those views popularly visited, such as the view in Fig. 15c which are around the center of the scene.

5.6 Discussions on real-time graphics effects

We apply the sprite tree method for the dense and static objects that enrich the context of a virtual scene. Our method can handle limited real-time graphic effects, like the lighting effects with a headlight in our simulation or the shadow casting among static objects. This type of shadows is captured into sprites and warped properly.

The proposed version of the sprite tree models static objects and the dynamic part of the scene can be rendered in a separate pass; moving transparent objects and particle effects can be rendered in the same separate pass with the background objects (see Sect. 3.2), as long as they have little interaction with the static objects. Another solution, for handling slow motion, is to time-stamp sprites and select valid sprites based on the time frame. This could also apply for predictable lighting changes (e.g. daylight changes in a natural scene). To support more real-time graphics effects, we could add additional geometry-related sprite attributes or even different sprites that help improve the interaction between sprites and dynamic scene elements. As an example, for shadow casting, dedicated "depth-sprites" could model shadow maps, and would be stored in a sprite tree and used for rendering, similarly to our proposed geometric sprites.

6 Conclusion

We propose the sprite tree as an efficient image-based representation to accelerate the rendering in NVEs. We show how to build a sprite tree by inserting only dissimilar views and how to render target views with a sprite tree using visible sprites only. Furthermore, we improve the performance of the sprite tree in terms of the size, the rendering quality and the rendering speed by modelling the sprite view similarity. This measure is used to (i) insert dissimilar sprites without considering the view similarity for building a compact sprite tree, and (ii) select suitable sprites considering both the visibility and the lighting conditions of sprites regarding any target view. The results show that the sprite tree does accelerate the rendering of the complex scene even using the basic insertion and rendering methods. After applying the sprite view similarity measure in the insertion and rendering, not only the sprite tree is more compact, but also the lighting artifacts are largely reduced, as compared to our basic methods and the LDI tree.

In this paper, we have proposed a framework for sprites modelling static scenes that simplifies the storage and rendering pipeline of the static part of the scene. This proposal allows users with limited bandwidth or rendering capabilities to improve their frame rates, benefiting from the rendering of other users. In future work, we would like to further consider modelling real-time graphics effects with sprites, as long as their use remains more efficient than direct rendering, either in storage space or in rendering time.

Acknowledgments We would like to thank Guillermo M. Leal Llaguno of Evolución Visual, Mexico for giving us permission to use the 3D model San Miguel in our work.

References

1. Aliaga, D.G.: Visualization of complex models using dynamic texture-based simplification. In: Proceedings of the 7th Conference on Visualization '96, VIS '96, pp. 101–ff (1996)
2. Bosc, E., Pepion, R., Le Callet, P., Koppel, M., Ndjiki-Nya, P., Pressigout, M., Morin, L.: Towards a new quality metric for 3-D synthesized view assessment. IEEE J. Sel. Top. Signal Process. **5**(7), 1332–1343 (2011)
3. Bouatouch, K., Point, G., Thomas, G.: A client-server approach to image-based rendering on mobile terminals. Research Report

RR-5447, French Institute for Research in Computer Science and Automation (2005)
4. Chang, C.F., Bishop, G., Lastra, A.: LDI tree: a hierarchical representation for image-based rendering. In: Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '99, pp. 291–298 (1999)
5. Chang, C.F., Ger, S.H.: Enhancing 3D graphics on mobile devices by image-based rendering. In: Proceedings of the Third IEEE Pacific Rim Conference on Multimedia: Advances in Multimedia Information Processing, PCM '02, pp. 1105–1111 (2002)
6. Cheng, L., Bhushan, A., Pajarola, R., Zarki, M.E.: Real-time 3D graphics streaming using MPEG-4. In: Proceedings of the IEEE/ACM Workshop on Broadband Wireless Services and Applications, BroadWise '04, pp. 1–16 (2004)
7. Cohen-Or, D., Noimark, Y., Zvi, T.: A server-based interactive remote walkthrough. Proceedings of the Sixth Eurographics Workshop on Multimedia **2001**, 75–86 (2002)
8. Decaudin, P., Neyret, F.: Volumetric billboards. Comput. Graph. Forum 28(8), 2079–2089 (2009)
9. Décoret, X., Durand, F., Sillion, F.X., Dorsey, J.: Billboard clouds for extreme model simplification. ACM Trans. Graph. **22**(3), 689–696 (2003)
10. Decoret, X., Sillion, F., Schaufler, G., Dorsey, J.: Multi-layered impostors for accelerated rendering. Comput. Graph. Forum **18**(3), 61–73 (1999)
11. Ghiletiuc, J., Färber, M., Brüderlin, B.: Real-time remote rendering of large 3D models on smartphones using multi-layered impostors. In: Proceedings of the 6th International Conference on Computer Vision / Computer Graphics Collaboration Techniques and Applications, MIRAGE '13, vol. 14, pp. 1–8 (2013)
12. Gortler, S.J., Grzeszczuk, R., Szeliski, R., Cohen, M.F.: The lumigraph. In: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '96, pp. 43–54 (1996)
13. Hoppe, H.: Progressive meshes. In: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '96, pp. 99–108 (1996)
14. Hudson, T.C., Mark, W.R.: Multiple image warping for remote display of rendered images. University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, Tech. rep. (1999)
15. Im, Y.H., Han, C.Y., Kim, L.S.: A method to generate soft shadows using a layered depth image and warping. IEEE Trans. Vis. Comput. Graph. **11**(3), 265–272 (2005)
16. Jeschke, S., Wimmer, M.: Textured depth meshes for real-time rendering of arbitrary scenes. In: Proceedings of the 13th Eurographics Workshop on Rendering, EGRW '02, pp. 181–190. Eurographics Association (2002)
17. Lamberti, F., Sanna, A.: A streaming-based solution for remote visualization of 3D graphics on mobile devices. IEEE Trans. Vis. Comput. Graph. **13**(2), 247–260 (2007)
18. Lee, K., Hong, S., Kim, S.J., Rhee, I., Chong, S.: SLAW: self-similar least-action human walk. IEEE/ACM Trans. Netw. **20**(2), 515–529 (2012)
19. Levoy, M., Hanrahan, P.: Light field rendering. In: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '96, pp. 31–42 (1996)
20. Liang, H., Silva, R.N., Ooi, W.T., Motani, M.: Avatar mobility in user-created networked virtual worlds: measurements, analysis, and implications. Multimedia Tools Appl. **45**(1–3), 163–190 (2009)
21. Lischinski, D., Rappoport, A.: Image-based rendering for non-diffuse synthetic scenes. In: Proceedings of the Eurographics Workshop on Rendering Techniques '98, pp. 301–314 (1998)
22. Ma, L., Duan, Q.: Image-based rendering of transparent object with caustic shadow. In: IEEE Youth Conference on Information, Computing and Telecommunication, YC-ICT '09, pp. 538–541 (2009)
23. Maciel, P.W.C., Shirley, P.: Visual navigation of large environments using textured clusters. In: Proceedings of the 1995 Symposium on Interactive 3D graphics, I3D '95, pp. 95–ff (1995)
24. Mark, W.R., McMillan, L., Bishop, G.: Post-rendering 3D warping. In: Proceedings of the 1997 Symposium on Interactive 3D graphics, I3D '97, pp. 7–ff (1997)
25. McMillan, L., Bishop, G.: Plenoptic modeling: an image-based rendering system. In: Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '95, pp. 39–46 (1995)
26. McMillan Jr., L.: An image-based approach to three-dimensional computer graphics. Ph.D. thesis, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA (1997)
27. Noimark, Y., Cohen-Or, D.: Streaming scenes to MPEG-4 video-enabled devices. IEEE Comput. Graph. Appl. **23**(1), 58–64 (2003)
28. Phong, B.T.: Illumination for computer generated pictures. Commun ACM **18**(6), 311–317 (1975)
29. Popescu, V., Lastra, A., Aliaga, D., de Oliveira Neto, M.: Efficient warping for architectural walkthroughs using layered depth images. In: Proceedings of the Conference on Visualization '98, VIS '98, pp. 211–215 (1998)
30. Qu, H., Kaufman, A., Shao, R., Kumar, A.: A framework for sample-based rendering with o-buffers. In: Proceedings of the 14th IEEE Visualization 2003, VIS '03, p. 58 (2003)
31. Schaufler, G., Stürzlinger, W.: A three dimensional image cache for virtual reality. Comput. Graph. Forum **15**(3), 227–235 (1996)
32. Shade, J., Gortler, S., He, L.w., Szeliski, R.: Layered depth images. In: Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '98, pp. 231–242 (1998)
33. Shade, J., Lischinski, D., Salesin, D.H., DeRose, T., Snyder, J.: Hierarchical image caching for accelerated walkthroughs of complex environments. In: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '96, pp. 75–82 (1996)
34. Shen, S., Brouwers, N., Iosup, A., Epema, D.: Characterization of human mobility in networked virtual environments. In: Proceedings of Network and Operating System Support on Digital Audio and Video Workshop, NOSSDAV '14, pp. 13–18 (2014)
35. Shen, S., Iosup, A.: Modeling avatar mobility of networked virtual environments. In: Proceedings of International Workshop on Massively Multiuser Virtual Environments, MMVE '14, pp. 1–6 (2014)
36. Shi, S., Jeon, W.J., Nahrstedt, K., Campbell, R.H.: Real-time remote rendering of 3D video for mobile devices. In: Proceedings of the 17th ACM International Conference on Multimedia, MM '09, pp. 391–400 (2009)
37. Shi, S., Kamali, M., Nahrstedt, K., Hart, J.C., Campbell, R.H.: A high-quality low-delay remote rendering system for 3D video. In: Proceedings of the International Conference on Multimedia, MM '10, pp. 601–610 (2010)
38. Shi, S., Nahrstedt, K., Campbell, R.: A real-time remote rendering system for interactive mobile graphics. ACM Trans. Multimedia Comput. Commun. Appl. **8**(3s), 1–20 (2012)
39. Sillion, F., Drettakis, G., Bodelet, B.: Efficient impostor manipulation for real-time visualization of urban scenery. Comput. Graph. Forum **16**(3), C207–C218 (1997)
40. Wang, Z., Bovik, A.C., Sheikh, H.R., Simoncelli, E.P.: Image quality assessment: from error visibility to structural similarity. IEEE Trans. Image Process. **13**(4), 600–612 (2004)
41. Wilson, A., Manocha, D.: Simplifying complex environments using incremental textured depth meshes. In: ACM SIGGRAPH 2003 Papers, SIGGRAPH '03, pp. 678–688 (2003)
42. Zhao, S., Ooi, W.T., Carlier, A., Morin, G., Charvillat, V.: 3D mesh preview streaming. In: Proceedings of the 4th ACM Multimedia Systems Conference, MMSys '13, pp. 178–189 (2013)

43. Zhu, M., Mondet, S., Morin, G., Ooi, W.T., Cheng, W.: Towards peer-assisted rendering in networked virtual environments. In: Proceedings of the 19th ACM International Conference on Multimedia, MM '11, pp. 183–192 (2011)

Minhui Zhu joined the Ph.D. program in National University of Singapore in 2009. She received her Bachelor degree in Computer Science from Xi'an Jiaotong University, Xi'an, China. Her research work is in image-based rendering for networked virtual environments.

Géraldine Morin is an associate professor at the University of Toulouse in France. Her initial work is in geometric modeling and she has since proposed different multi-resolution models, and compact and progressive representations for 3D models in multimedia. She also worked on collaborative analysis and synthesis of images for reconstructing 3D models from images (for plants or circular shapes), on manipulation of 3D shapes (visual tracking, analysis of similarity).

Vincent Charvillat received the Eng. degree in Computer Science and Applied Mathematics from ENSEEIHT, Toulouse France and the M.Sc. in Computer Science from the National Polytechnic Institute of Toulouse, both in 1994. He received the Ph.D. degree in Computer Science from the National Polytechnic Institute of Toulouse in 1997. He joined the Computer Science and Applied Mathematics department of ENSEEIHT in 1998 as an assistant professor. He obtained the habilitation degree in Computer Science in 2008 and is currently a professor at the University of Toulouse, IRIT research lab, ENSEEIHT Eng. School. He is the head of VORTEX research team at ENSEEIHT (Visual Objects: from Reality To EXpression). His main research interests are visual processing and multimedia applications. Current topics of research include visual object extraction (object tracking, detection and coding), compositing (augmented reality and hypermedia), interactive delivery (multimedia adaptation, mobile applications).

Wei Tsang Ooi is an associate professor in the Department of Computer Science, National University of Singapore. His research interest is in the area of systems support for multimedia applications, including video streaming, graphics streaming, and networked virtual environment.