Real-Time Shadows for Animated Crowds in Virtual Cities

Céline Loscos  
c.loscos@cs.ucl.ac.uk  

Franco Tecchia  
f.techhia@cs.ucl.ac.uk  

Yiorgos Chrysanthou  
y.Chrysanthou@cs.ucl.ac.uk

University College London  
Computer Science Department  
Gower Street  
WC1E 6BT London, UK

ABSTRACT
In this paper, we address the problem of shadow computation for large environments including thousands of dynamic objects. The method we propose is based on the assumption that the environment is 2.5D, which is often the case for virtual cities, thus avoiding complex visibility computation. We apply our method for virtual cities populated by thousands of walking humans, which we render with impostors, allowing real-time simulation.

In this paper, we treat the cases of shadows cast by buildings on humans, and by humans on the ground. To avoid 3D computation, we represent the shadows cast by buildings onto the environment with a 2.5D shadow map. When humans move, we quickly access the shadow information at the current location with a 2D grid. For each new position of a human, we compute its coverage by the shadow, and we render the shadow on top of the impostor with low cost using multi-texturing hardware. We also use the property of an impostor to display the shadow of humans on the ground plane, by projecting the impostor relatively to the light source.

The method is currently limited to sharp shadows and a single light source. However, approximations could be made to allow non-accurate soft-shadows. We show in the results that the computation of the shadows, as well as the display is done in real time, and that the method could be easily extended to real-time moving light sources.

Keywords
Shadow computation, dynamic shadows, real-time rendering, populated virtual cities, multi-texturing.

1. INTRODUCTION
The interactive visualisation of complex and realistic environments has always been one of the goals of virtual reality. Over the years, the performance of graphics hardware have greatly and stably improved, so that it is now possible to render in real-time even complex polygonal models. A common example of very complex scenarios is cities and other urban environments which, especially in recent years, have been widely utilised in virtual reality and virtual presence experiments.

As the available computational power increases, it becomes feasible to introduce a higher level of complexity in city simulation; ongoing research is analysing the issue of populating these empty models with a crowd of intelligent virtual agents. In fact, only in this way is it possible to obtain proper virtual cities, where the user can navigate and, up to a certain extent, "live". Populating virtual cities can considerably boost the realism of such models with the presence of crowds and traffic, elements that we are used to in everyday life. The presence of crowd helps also to give a much better idea of the size and the dimensions of buildings and roads, and also to give a "semantic" to the cities, where densely populated spots normally highlight important features of blocks or single buildings of the urban environment. Such simulations are important for applications such as architectural walkthrough, urban simulation, games and the movie industry, as well as for internet applications.

Examples of such research can be found in [13, 9, 10, 19, 5, 7, 6]. In some of these works many efforts are devoted to give realistic behaviours to the virtual humans, so that is possible to simulate very interesting behaviours such as flow of crowds or car traffic. Being virtual reality application, most of these works normally share a common strong requirement: the frame rate of the simulation must be sufficiently high so to make the user feel immersed in a truly interactive virtual city. For this reason, the accuracy and quality of the graphical representation are often limited in order to obtain higher rendering speeds, and, when it comes to choosing the visual feature to implement in the application, an important detail that is often dropped is the use of shadows. Unfortunately the absence of shadows can greatly diminish the visual realism of a simulation; at early stage of computer graphics shadows had been acknowledged to be very important in order not only to contribute to the realism, but also to help the user to situate objects relatively to each other [16]. Shadows are so natural for the human eye, that their presence greatly improves the overall perceived realism even if the scene is not rendered in a photo realistic way. Typically, shadows are not implemented because current shadow computation techniques are too computationally intensive to be performed in real-time on complex scenarios; also, the human eye is very sensible to consistency, and, in the case of urban environments, shadows need then to be generated taking into account complex geometric relations. This is especially true if a population is added, so that there can be thousands of dynamic objects (humans, cars and so on).

In this paper we propose methods for computing real-time shadows in such highly dynamic scenes. We make the assumption that the static model - the city buildings - is 2.5D and we do not address models with objects such as bridges. In [19], an urban simulation
Figure 1: Example of our system, allowing real time shadow update for thousands of humans walking around a city.

system capable of rendering thousands of avatars in real time was presented. These results were achieved by using image-based rendering to represent each avatar with a single dynamic impostor. In the present paper, we describe how real-time approximate shadows can be added to this original system introducing just a relatively small speed reduction. It is our experience that introducing shadows greatly improves realism even if the shading is not perfectly accurate. The basic idea of our approach is to use a discretization of the environment to track the position of each virtual agent, and to use 2.5D representation of the shadow volumes to locate shadow areas. Because in our case the dynamic objects are visualised using impostors, we render the shadows by placing a second texture on top of the original impostor image. To minimise the rendering cost of such a texture, we use the common multi-texturing capabilities of modern hardware and at run-time we compute appropriate texture coordinates to fit the required coverage of the shadow. Finally, we propose a simple (approximate) procedure to cast shadows from impostors onto the static environment. An example image of the final result is shown in Fig. 1.

2. PREVIOUS WORK

A large number of techniques exist in the literature for shadow generation. They can be broadly subdivided into two categories. Soft shadows are produced by taking the light source to be a finite area or volume, and sharp shadows when the light is assumed to be at infinity or a single point. The former case is much more accurate and realistic, but at the same time they also require complex and expensive visibility computations. In general they are not really applicable to real-time rendering for complex dynamic scenes. Techniques such as discontinuity meshing are the most accurate ones; they find not only the shadow boundaries but also the edges where the direct illumination on the surfaces varies discontinuously [14, 8]. However, they are slow to compute and render because of the resulting complex mesh. More approximate techniques that use texture mapping instead of partitioning the input polygons into a mesh [17, 11] are faster to render. However, they still cannot be computed in real time for any sizeable scene and they require separate textures for different surfaces.

For real time applications, the rendering of shadows has often been restricted to sharp one. The simplest method is the fake shadows technique [1]. After rendering the model from the given viewpoint, a matrix is added on the stack that has the property that it projects every vertex onto a plane (usually the floor). The scene is rendered again with the projected objects as "grey", appearing as shadows. An additional rendering of the scene is needed for each receiving surface. Shadows are computed at each frame. Some methods, such as the SVBSP tree [2] precompute the shadows so they just need to be rendered for each new viewpoint. The computation of a full solution on all surfaces is possible but is time consuming. Although incremental update is possible for a small number of dynamic objects [3], this would be too costly for a highly dynamic setting such as ours. Also, even ignoring the update cost, the number of extra shadow polygons that are needed for the display of shadows can greatly affect the rendering performance.

Currently the most popular techniques use the graphics hardware to compute the shadows at a per pixel level, thus avoiding the shadow storage problem. Shadow volumes methods [4] were used to delimit a spatial region in shadow between the object surface and a receiver. Using the stencil buffer [12], regions in shadow can be automatically detected by the hardware. An interesting alternative method for computing shadow planes was suggested by McCool [15]. The scene is drawn first from the light position and the z-buffer is read. The shadow volumes are then reconstructed based on the edges of the discretized shadow map. The discretization can introduce artefacts though. One of the drawbacks of this method is that the shadow planes tend to be very large and they have a detrimental effect on the rendering time. In practice this effect can be limited by using the method not for a complete solution but rather for shadows only from the ‘important’ objects. Similar to the shadow Z-buffer this method can be used to cast shadows onto any object that can be scan converted. But unlike the shadow Z-buffer the objects causing the shadows need to be polygonal.

Methods based on shadow maps [21] are pixel-based mixing depth information provided both from the light source and the user points of view. Shadow areas are determined by comparing the depth of the points from both the light source viewpoint and the user viewpoint. Such methods are now implemented in hardware allowing fast shadow computation for complex geometrical environments. However they suffer from two major drawbacks due to image resolution. First images can be aliased if the resolution does not permit to accurately decide on depth. In particular this is often the case for large scenes for which objects might be represented by few pixels. Second, because it is pixel-based, the frame rate depends on the displayed image size.

Depth images from light sources had been therefore widely used to determine fast visibility without complex visibility computation. We based our method in these ideas, taking advantage however of the volume described by shadows rather than visibility boundaries. As explained in the following sections, our solution is viewpoint-independent and the quality of the displayed results does not depend on the resolution of the displayed image. Changes in visibility are detected quickly using a 2D discretization of the shadow depth, but shadows are computed locally reducing artefacts when displayed.

3. OVERVIEW OF THE METHOD

3.1 The target rendering system

Before we describe the shadow method, let us motivate it by giving a short description of the system it will be used in. Tecchia et al. [19] developed an image-based rendering system to simulate thousands of animated humans inside virtual cities. Instead of using a polygonal model for each human, the complexity is reduced using one impostor with an appropriate texture mapped on it. The textures are pre-computed from a number of view positions dis-
tributed on a hemisphere placed around the avatar model and for several frames of animation (see Fig. 2). For rendering, the chosen image to display is the closest to the current viewpoint given a frame of animation. This results in animated virtual humans walking around a city. The model of the city is 2.5D, which allows for information access in 2D. When humans move in space, they access the height information of the target cell to check for obstacles. This allows for fast access to local information and this had been used for fast collision detection against the buildings and the other humans. Collision detection with the environment is done by checking the height of the static objects stored in a depth map. The algorithm is explained in [20]. An example of a collision detection decision is shown in Fig. 3. A grid is then used as well to detect inter-collision between humans. Although human are placed on the 2D grid, they can still occupy a discrete set of positions inside each cell allowing smooth animation. The extension of using the 2D grid to control general behaviour had been expressed in [18].

As presented in [19], the system did not compute or display dynamic shadows. In our approach, we also use the 2D grid to store local information on shadows. Since humans are represented by impostors, it is difficult to compute exact shadow boundaries. Traditional shading techniques run on polygonal models. A straightforward way to shade an impostor is to go back to its polygonal model and re-generate the images. Another is to pre-compute different lighting configurations and to save images for each of them. Since we deal with many impostors interacting in a complex polygonal environment, we decided to reduce the image storage to one given shading and approximate changes when they occur working directly on the image. The shading chosen is diffuse, to reduce the visual artefacts that specular effects may cause during reshading.

In this paper we address case (2) and partly case (3). However we use fake shadows [1] to display shadows from the buildings on the ground and the OpenGL lighting to shade buildings. Although it does not allow us to treat case (1) for moving light sources, it allows the method proposed to situate shadows. We do not address case (4) in this paper since it is difficult to compute shadows between thousands of moving objects. We do, however, have some ideas to extend the method used for case (3).

Addressing case (2) is not obvious. When humans walk in a virtual city, it means that thousands of dynamic objects (and their shadow) need to be updated in real time. Especially, we need to update shadows for every dynamic object. This problem is extremely complex when considering it in a general case. However, in our case, the static scene with which humans need to interact, can be assumed to be 2.5D and therefore the volume covered by the shadows can be approximated by a 2.5D map. The idea is to discretize the shadow planes and to store them as a 2D height map. Then it is possible to compare the height of the people with the height of the shadows and to compute the degree of coverage of a human by a shadow. The way we compute the 2.5D shadow map and the way dynamic objects access it, are described in section 4.

The shadow detection works for any kind of animated object. If the objects are polygonal, the information stored in the shadow height map can be used to quickly compute shadows onto the polygons. In our case, we compute shadows for moving objects represented by impostors. We decided to use a shadow texture mapped on the top of the impostor to darken the part in shadow. The way the texture coordinates are computed and the texture applied is described in section 5.

Finally we use impostors to display shadows cast by the humans. Instead of using the viewpoint as a reference, we consider the light source position. Therefore the same images as the one used for impostors can be used for shadows. This enables high-quality shadows representing the exact shape of the walking human with extremely low additional cost. We describe this in section 6.

4. DISCRETIZED INFORMATION FOR SHADOW DETECTION

In order to compute the shadows cast by the buildings onto the animated humans, we take advantage of the 2.5D attributes of a city. We use a height map to represent the area covered by the shadows of the static scene. In the next section 4.1, we explain how we compute the shadow height map, and how we access it in section 4.2.

4.1 2.5D representation of shadow coverage

In order to get a 2.5D map to represent the shadows, we compute for a given light source, the shadow planes that make shadow volumes. The shadow planes can be computed in any of the standard
ways. We decided to compute shadow volume by setting a plane below the whole scene, and then finding the intersection with this plane of every ray going from the light source to each vertex of the scene. For each model edge, the original vertices with the projected ones define the shadow plane.

Once the shadow volume boundary planes are computed, the rendering is done offscreen and from a viewpoint placed on the top of the model, using an orthogonal projection. From the z-buffer of this image, we extract the depth information that we store at the usual grid resolution. The saved map is shown in Fig. 4 (a). The height of the shadows relatively to the height of the objects is given by the difference between our shadow height map and the original depth of the geometry (shown in Fig. 4 (b)). This results to 0 in area non-covered by shadows. The result map is called shadow height map, and is the one we use to detect shadow in the following.

The important key point of this method is the use of the 2.5D map to represent shadows in space. The way we computed the shadow height map could be optimised using hardware. Quicker ways to compute the shadow volume information could allow interactive updates for a moving light source.

4.2 Access of shadow information

We check the shading in 2D, using a regular grid. At each cell occupied by an agent, we check whether the current height of the human is higher than the height of the shadow stored in a 2.5D shadow map. The difference in height gives the percentage of coverage of the shadow on the top of the human. When humans move, their position is located in 2D. When reaching a new cell, the particle checks the depth of the shadow. Since we have a convex building configuration, the number of cases is limited to uncovered, partially covered and fully covered.

5. APPROXIMATE SHADING OF ANIMATED IMPOSTORS

5.1 Display using multi-texturing

When detecting in which case the shading of the impostor corresponds, we set up the tag for the display. When fully covered and uncovered, the impostor support polygon could be rendered either in white or grey. When partially covered, we use a texture mapped onto the impostor to reflect the shadow boundary.

We approximate the solution by the assumption that when a shadow is cast, it covers everything under it, given the 2.5D configuration of the scene. This excludes the case of roofs overhanging the edge of buildings, bridges, and other kind of “non 2.5D” objects. We use therefore a single texture representing 2 colours, white on the top and black on the bottom. The texture is shown in Fig. 5 (a). Before mapping this texture we need to compute the appropriate texture coordinates. This is explained in the next section. This texture is then sent to the rendering pipeline using the multi-texturing feature of the graphic cards. If a second texture unit is not available, two rendering passes can be used instead. Because of the multi-texturing is virtually cost-free, we render the second texture for every polygon using texture coordinates in the white area of the texture for uncovered case, and in the black area for fully covered. An example of the three different cases is shown in Fig. 5.

5.2 Shadow texture coordinate computation

A simple approach to compute texture coordinates is to compute the percentage of coverage of the shadow comparing the height of the human with the height of the shadow (see Fig. 6 (a)). This first solution gives a very good impression on the shading. However shadows rarely describe horizontal boundaries in a city lit by the sun. To detect the inclination of the shadow, we therefore check the cells around the occupied one, and check the difference in depth (see Fig. 6 (b)). This may results in inaccuracy in the shadow boundary, but it reflects the transition in getting into a shadow, and avoids shadow popping when going from one cell to the other. At this stage, we have not yet implement an algorithm for checking the cells to compute the inclination of the shadow. The idea is to compute an average out of the shadow height of the neighbour cells, weighted by the position of the particle inside the current cell. Since the particle has a discreet position inside a cell, we need to compare its distance relatively to the centre of each cell. We compute a shadow boundary for each of the side of the impostor. This needs to be done at the rendering time since the borders of the impostor are view dependant.

For faster computation, we separate the neighbourhood into four quadrants, described by the connection of the middle of each cell (see Fig. 6 (c)). The weighted average is done only considering the cells of the quadrant the particle borders are. This interpolation allows not only to have a better description of the inclination of the shadow, but also to have smoother transitions when the humans move.

6. SHADOWS OF PEOPLE ONTO THE ENVIRONMENT

In order to add realism, it is very important that humans cast shadows onto the environment. We do not provide a complete so-
Figure 6: Computation of texture coordinates, based on the height stored at each pixel. (a) The texture coordinates are uniform while the human stays inside the same cell. (b) If we interpolate with the adjacent cells, it can provide smoother shadow boundaries. (c) Interpolation is done for the adjacent cells situated in the quadrant were the human is. In this example, the human represented by a red dot, is situated in the upper left quadrant in green. The four cells to be taken in consideration for the shadow texture coordinate computation are the cells covered by the green colour.

Figure 7: (a) Image of the avatar viewed from the light source. (b) Impostor computed from this position (see Fig. 2). (c) Projection of the avatar impostor on the floor relatively to the light position.

Figure 8: (a) Model with human. No shadows are simulated. (b) Same as (a) but with fake shadows, representing shadows of the buildings. (c) Same as (b) with shadows of humans on the ground added. (d) All shadows simulated, impostors are darken in shadow area.

7. IMPLEMENTATION AND RESULTS

This system was tested on a PC Pentium 833Mhz with an NVIDIA GeForce GTS2 64 Mb graphic card. We tested the shadow method for thousands of humans walking around a village modelled with 41,260 polygons, performing collision avoidance against the building and themselves. The tests were done for 1,000, 5,000 and 10,000 humans. The display window is set to be 1024x1024. Creating the 2.5D shadow map takes 1 second for such a model, including the shadow volume vertices computation (which takes only 0.1 second). This map is 1024x1024 large and the stored depth is represented in 24 bits.

To evaluate the additional cost of the shadows, we activated step by step the different components of shadow computation and display to highlight the time consumption of each step of the method. We also set up a fixed walkthrough path since the number of human and polygons rendered at each frame has a consequence on the rendering frame rate. Each simulation is composed of 1160 frames. For each animation we computed the average number of rendered frames per second as well as the total time of computation for the all 1160 frames. The results are summarised in Table 1. The second column shows the timings for the scene rendered with the static model and the avatars without any shadows (see Fig. 8 (a)). For the timings of the third column we added in the simulation the display of the fake shadows, representing shadows of the buildings on the ground (see Fig. 8 (b)). The forth column shows timings for the simulation including shadow projection of the humans on the floor (see Fig. 8 (c)). The fifth column shows the results when everything is simulated including shadows projected by the buildings on the humans (see Fig. 8 (d)). The video of each case is shown on http://www.cs.ucl.ac.uk/staff/Y.Chrysanthou/Crowds/VRST01/.

The display of the static model as well as the fake shadows appears to be the most time-consuming task. The table shows as well that the frame rate decrease when more humans are simulated, but
were taken with 10,000 animated people. The pictures in Fig. 9 show shadow map, we could modify the light source position and perform real-time updates of the shadows. The pictures in Fig. 9 show different lighting conditions and different views of the system, and were taken with 10,000 animated people.

8. CONCLUSION AND FUTURE WORK

We presented new methods to compute and update shadows for thousands of dynamic objects, moving in a 2.5D environment. We focused on improving consistency of positioning between objects rather than accurate shadow casting. This method is adapted to the context of a virtual city simulation with animated crowd of humans. We extended the method presented by Tecchia et al. [19] that use image-based rendering techniques to allow interactive frame rates. In this system, to reduce the required number of polygons to display, humans are replaced by impostors.

To enable fast shadow detection, we use a 2.5D map to locate shadows, avoiding complex visibility computations. This method can be used for any type of dynamic scene with 2.5D configuration, and is not restricted to scenes simulated with impostors. Because in our case humans are displayed using impostors, we use multitexture rendering to add a shadow texture on the top of each human. This texture is simple and the same for every human. We compute texture coordinates to reflect the height of the coverage of the shadow. Although the texture coordinates are easier to compute for a single polygon, the application of a texture could also be used for polygonal dynamic objects rendered. We also take advantage of the impostor representation to quickly cast shadows of humans onto the ground.

We show in this paper that the additional cost is relatively low, and that the system still allows for interactive walkthrough in a crowded virtual city. The display of shadows greatly improves the visual quality of the simulation. However, we do not compute accurate shadows, and the quality of the result is limited by the resolution of the 2.5D shadow map, the resolution of the depth component and the resolution of the impostor images. This work is at an early stage, and some of the implementation could be optimised. There is definitely room to accelerate the rendering of the shadows of the buildings as well as the computation of the 2.5D shadow map.

We also still have configuration of shadows to take into account, computing shadows between buildings, between avatars, and of avatars onto the building. For the two latest cases, we want to take benefit of the 2D grid to quickly locate grid cells affected by the shadow of a human. Visibility computation should then just be computed only for the humans or the buildings occupying the occluded cells. We can also reduce the visibility computation when humans and buildings are close to the field of view of the user, avoiding complex computation for the non-visible part.

In the long term, we would like to extend this system to take into account multiple moving light sources as well as approximate soft shadows, while keeping the real-time constraint. A first step to display soft shadows would be to soften the border of the shadow textures.

Acknowledgement

This work was in part supported by the EPSRC project GR/R01576/01 and the EPSRC Interdisciplinary Research Centre equator.

9. REFERENCES

Figure 9: Results for 10,000 humans with different light source positions. (a) Noon shadows. (b) Mid-afternoon shadows. (c) Late afternoon shadows. (Figure reproduced in color on page 199.)


